SUMAT

Jonathan M. Borwein
Matthew P. Skerritt

An Introduction to Modern Mathematical Computing

With Mathematica®

Springer

Springer Undergraduate Texts
in Mathematics and Technology

For further volumes:
http://www.springer.com/series/7438

Jonathan M. Borwein • Matthew P. Skerritt

An Introduction to Modern Mathematical Computing

With Mathematica®

 Springer

Jonathan M. Borwein
Director, Centre for Computer Assisted Research
  Mathematics and its Applications (CARMA)
University of Newcastle
Callaghan, NSW 2308
Australia
jon.borwein@gmail.com

Matthew P. Skerritt
Centre for Computer Assisted Research
  Mathematics and its Applications (CARMA)
University of Newcastle
Callaghan, NSW 2308
Australia
matt.skerritt@gmail.com

To my grandsons Jakob and Skye, and granddaughter Zöe.
   Jonathan Borwein

To my late grandmother, Peggy, who ever urged me to hurry up with my PhD, lest she not be around to see it.
   Matthew Skerritt

Preface

Thirty years ago mathematical, as opposed to applied numerical, computation was difficult to perform and so relatively little used. Three threads changed that:

- The emergence of the personal computer, identified with the iconic Macintosh but made ubiquitous by the IBM PC.
- The discovery of fiber-optics and the consequent development of the modern Internet culminating with the foundation of the World Wide Web in 1989 made possible by the invention of hypertext earlier in the decade.
- The building of the "Three Ms": *Maple*™, *Wolfram Mathematica*®, and MATLAB. Each of these is a complete mathematical computation workspace with a large and constantly expanding built-in "knowledge base". The first two are known as "computer algebra" or "symbolic computation" systems, sometimes written *CAS*. They aim to provide exact mathematical answers to mathematical questions such as what is

$$\int_{-\infty}^{\infty} e^{-x^2} \, dx,$$

what is the real root of $x^3 + x = 1$, or what is the next prime number after 1,000,000,000? The answers, respectively, are

$$\sqrt{\pi}, \quad \frac{\sqrt[3]{108 + 12\sqrt{93}}}{6} - \frac{2}{\sqrt[3]{108 + 12\sqrt{93}}}, \quad \text{and } 1{,}000{,}000{,}007.$$

The third M is primarily numerically based. The distinction, however, is not a simple one. Moreover, more and more modern mathematical computation requires a mixture of so-called *hybrid* numeric/symbolic computation and also relies on significant use of geometric, graphic and, visualization tools. It is even possible to mix these technologies, for example, to make use of MATLAB through a *Maple* interface; see also [6]. MATLAB is the preferred tool of many engineers and other scientists who need easy access to efficient numerical computation.

Of course each of these threads rely on earlier related events and projects, and there are many other open source and commercial software packages. For example, *Sage* is an open-source CAS, *GeoGebra* an open-source interactive geometry package, and *Octave* is an open-source counterpart of MATLAB. But this is not the place to discuss the merits and demerits of open source alternatives. For many purposes *Mathematica* and *Maple* are interchangeable as adjuncts to mathematical learning. We propose to use the latter. After reading this book, you should find it easy to pick up the requisite skills to use *Mathematica* [14] or MATLAB.

Many introductions to computer packages aim to teach the *syntax* (rules and structure) and *semantics* (meaning) of the system as efficiently as possible [7, 8, 9, 13]. They assume one knows why one wishes to learn such things. By contrast, we intend to persuade that *Maple* and other like tools are worth knowing assuming only that one wishes to be a mathematician, a mathematics educator, a computer scientist, an engineer, or scientist, or anyone else who wishes/needs to use mathematics better. We also hope to explain how to become an experimental mathematician while learning to be better at proving things. To accomplish this our material is divided into three main chapters followed by a postscript. These cover the following topics:

- **Elementary number theory.** Using only mathematics that should be familiar from high school, we introduce most of the basic computational ideas behind *Maple*. By the end of this chapter the hope is that the reader can learn new features of *Maple* while also learning more mathematics.
- **Calculus of one and several variables.** In this chapter we revisit ideas met in first-year calculus and introduce the basic ways to plot and explore functions graphically in *Maple*. Many have been taught not to trust pictures in mathematics. This is bad advice. Rather, one has to learn how to draw trustworthy pictures.

In[1]:= `Plot[x * Sin[1 / x], {x, -1 / 2, 1 / 2}]`
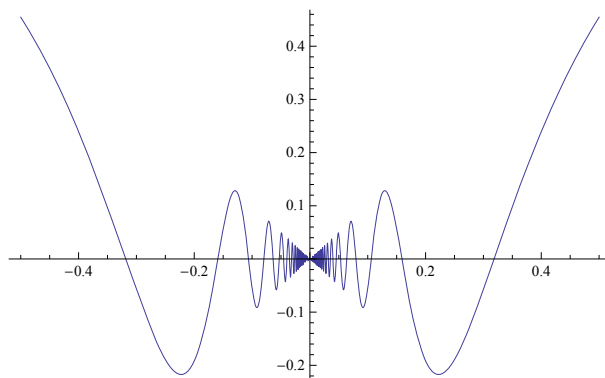
Out[1]=

- **Introductory linear algebra.** In this chapter we show how much of linear algebra can be animated (i.e. brought to life) within a computer algebra system. We suppose the underlying concepts are familiar, but this is not necessary. One of the powerful attractions of computer-assisted mathematics is that it allows for a lot of "learning while doing" that may be achieved by using the help files in the system and also by consulting Internet mathematics resources such as *MathWorld, PlanetMath* or *Wikipedia.*
- **Visualization and interactive geometric computation.** Finally, we explore more carefully how visual computing [10, 11] can help build mathematical intuition and knowledge. This is a theme we will emphasize throughout the book.

Each chapter has three main sections forming that chapter's core content. The fourth section of each chapter has exercises and additional examples. The final section of each chapter is entitled "Further Explorations," and is intended to provide extra material for more mathematically advanced readers.

Based on these principles, *An Introduction to Modern Mathematical Computing with Maple^{TM}* was published in July 2011, using *Maple* as the software tool. This book is, essentially, the same text corresponding to the *Mathematica* system. For the most part the same examples and techniques have been "translated" to *Mathematica*, but

occasionally the structure of *Mathematica*'s language, or other particulars of the system have necessitated a divergence from the previous book.

In particular, the entire section on geometric constructions from Section 4.2 needed to be performed in *Cinderella*. Additionally, several errors made by *Maple* to do with infinite sums and products which lead to interesting mathematical explorations are simply not made by *Mathematica*. In all cases, the author has endeavored to adhere to the principles described above.

A more detailed discussion relating to many of these brief remarks may be followed up in [2, 3, 4] or [5], and in the references given therein.

The authors would like to thank Wilhelm Forst for his comments, corrections and suggestions, Joshua Borwein-Nevin for his work helping to change *Maple* code to *Mathematica* code, and Shoham Sabach and James Wan for their help proofreading preliminary versions of the *Maple* version of the book.

Additional Reading and References

We also supply a list of largely recent books at various levels that the reader may find useful or stimulating. Some are technical and some are more general.

1. George Boros and Victor Moll, *Irresistible Integrals*, Cambridge University Press, New York, 2004.
2. Jonathan M. Borwein and Peter B. Borwein, *Pi and the AGM: A Study in Analytic Number Theory and Computational Complexity*, John Wiley & Sons, New York, 1987 (Paperback, 1998).
3. Christian S. Calude, *Randomness and Complexity, from Leibniz To Chaitin*, World Scientific Press, Singapore, 2007.
4. Gregory Chaitin and Paul Davies, *Thinking About Gödel and Turing: Essays on Complexity, 1970-2007,* World Scientific, Singapore, 2007.
5. Richard Crandall and Carl Pomerance, *Prime Numbers: A Computational Perspective,* Springer, New York, 2001
6. Philip J. Davis, *Mathematics and Common Sense: A Case of Creative Tension,* A.K. Peters, Natick, MA, 2006.
7. Stephen R. Finch, *Mathematical Constants*, Cambridge University Press, Cambridge, UK, 2003.
8. Marius Giaguinto, *Visual Thinking in Mathematics,* Oxford University, Oxford, 2007.
9. Ronald L. Graham, Donald E. Knuth, and Oren Patashnik, *Concrete Mathematics*, Addison-Wesley, Boston, 1994.
10. Bonnie Gold and Roger Simons (Eds.), *Proof and Other Dilemmas: Mathematics and Philosophy*, Mathematical Association of America, Washington, DC, in press, 2008.
11. Richard K. Guy, *Unsolved Problems in Number Theory*, Springer-Verlag, Heidelberg, 1994.
12. Reuben Hersh, *What Is Mathematics Really?* Oxford University Press, Oxford, 1999.
13. J. Havil, *Gamma: Exploring Euler's Constant*, Princeton University Press, Princeton, NJ, 2003.
14. Steven G. Krantz, *The Proof Is in the Pudding: A Look at the Changing Nature of Mathematical Proof*, Springer, New York, 2010.

15. Marko Petkovsek, Herbert Wilf, and Doron Zeilberger, *A=B,* A.K. Peters, Natick, MA, 1996.

16. Nathalie Sinclair, David Pimm, and William Higginson (Eds.), *Mathematics and the Aesthetic. New Approaches to an Ancient Affinity*, CMS Books in Math, Springer-Verlag, New York, 2007.

17. J. M. Steele, *The Cauchy-Schwarz Master Class*, Mathematical Association of America, Washington, DC, 2004.

18. Karl R. Stromberg, *An Introduction to Classical Real Analysis*, Wadsworth, Belmont, CA, 1981.

19. Richard P. Stanley, *Enumerative Combinatorics*, Volumes 1 and 2, Cambridge University Press, New York, 1999.

20. Terence Tao, *Solving Mathematical Problems*, Oxford University Press, New York, 2006.

21. Nico M. Temme, *Special Functions, an Introduction to the Classical Functions of Mathematical Physics*, John Wiley, New York, 1996.

22. Fernando R. Villegas, *Experimental Number Theory*, Oxford University Press, New York, 2007.

Many other useful links are maintained by the authors of *Mathematics by Experiment* [3] at http://www.experimentalmath.info/.

Finally, errata and other information relating to this book can be found online at http://carma.newcastle.edu.au/books/mathematicalcomputing/.

Jonathan Borwein
Matthew Skerritt
May 31, 2012

Contents

Conventions and Notation

Mathematica

For the purposes of this book, we assume that *Mathematica* is operating according to the system defaults that are set in version 8. *Mathematica* examples in this book are formatted to look like they would in a *Mathematica* notebook using these defaults, and look like the following

In[2]:= **Input**
      Input; Input
      ...
      Errors »

Out[2]= *Output*

Out[3]= *Output*

Each input—and its associated output—are numbered. Multiple commands may be input together at the same prompt, and the input may even be spread over multiple lines. Multiple input lines may form a block have a single number, however corresponding outputs will each be numbered individually. Note that in some cases *Mathematica* might produce output that is not numbered, however we have opted for the purposes in this book to number such outputs. Do not be overly concerned if your copy of *Mathematica* does not number output that the book has numbered.

Input is colored black and appears in a bold monospaced "typewriter" (Courier) font. Note that some input characters might have other colors. Output is black and is typeset like regular mathematics. Errors are orange-ish in a small sans-serif font, and terminate with a » character. Clicking on this character in a notebook will bring up extra help for the error in question.

When entering input into *Mathematica*, pressing the enter (or return) key will move to a new line of the same input block. In order to make the commands run, and generate output, then *shift-enter* must be pressed. That is, holding down the shift key while also pressing the enter (or return) key.

Basic arithmetic symbols are as follows. Note that multiplication may be achieved using either the asterisk (∗), or with a space. Using the asterisk leaves no ambiguity as to the nature of the calculation, whereas a space might easily be missed. As such the reader should feel free to use whichever method he or she prefers, but should be aware that the *Mathematica* examples in this book will endeavor to use the asterisk to

denote multiplication, unless there can be no ambiguity (for instance, with polynomial coefficients).

Desired Effect	Key Combination	Example
Addition	+	a + b
Subtraction	–	a – b
Multiplication	*	a * b
	␣	a b
Division	/	a / b
Raise to a power	^	a^b
New Line	*<enter>*	
Run Command	*<shift-enter>*	

Table P.1 Essential *Mathematica* Key Combinations

Standard order of operations is followed, and so the use of parentheses is required for explicit operation ordering.

The Documentation Center

Mathematica provides electronic documentation as part of its functionality. This documentation is called the *Documentation Center*. This book will, on many occasions, refer the reader to the Documentation Center for further reference.

Additionally, the reader is expected and encouraged to turn to the Documentation Center whenever they are trying to work out how to do something in *Mathematica*. Several exercises will require students to find functions in the Documentation Center.

The Documentation Center can be accessed through the Help menu of the program. Additionally, an online copy of the Documentation Center is available at:
http://reference.wolfram.com/mathematica/guide/Mathematica.html

Useful Functions

We list here, for convenience, a small list of *Mathematica* functions and key words which are arguably essential to know. These are commonly used constants, as well as functions to perform elementary mathematical operations (for example, square roots) or for simplification. The list is in Table P.2. Consult the Documentation Center about these functions for more specific details on their use. Note that many more functions are introduced and explained within the main text of this book.

Some comments:

- Powers of e may be obtained with the `Exp` function, or simply by raising the constant `E` to a power.
- The `Log` command may also be directed to perform logs to a base other than e if desired, but will perform natural logs unless specifically told otherwise. For example, the command `Log[3, 2]` will calculate the log of 3 to the base 2.

Desired Effect	Command
Value of π	`Pi`
Value of e	`E`
Value of ∞	`Infinity`
Square root of a number ($\sqrt{x}$)	`Sqrt[x]`
nth root of a number ($\sqrt[n]{x}$)	`x^(1/n)`
Power of e (e^x)	`Exp[x]`
	`E^x`
Natural logarithm ($\log x$)	`Log[x]`
Rearrange an algebraic expression	`Simplify[...]`
	`Factor[...]`
	`Expand[...]`
Force evaluation	`Evaluate[...]`
Numeric Computation	`N[...]`
	`NIntegrate[...]`
	`NSum[...]`

Table P.2 Essential *Mathematica* Functions

- The *Mathematica* keyword `Pi` must have a capital 'P' and a lower case 'i'. Any other combination of cases for these letters is a different name, and will not be recognized by *Mathematica* as π.
- The commands for algebraic rearrangement work primarily as one would expect from their names. However, it should be noted that when faced with a complicated answer from *Mathematica* it is sometimes the case that **Factor**, or **Expand** will produce a more simplified answer than the **Simplify** function does. Furthermore, these commands used in conjunction with each other can produce superior results.
- Numeric computations of symbolic expressions can be computed directly with the **N** function. However, *Mathematica* provides several functions for numerically computing specific things, such as integration (via **NIntegrate**) or sums (via **NSum**). For a full list, the Documentation Center entry named "Numerical Calculations" should be consulted.
- *Mathematica* numeric precision handled on a value by value basis, not on a notebook-wide basis. To force a particular precision, the **N** function can be invoked with an optional argument for precision, see Exercise 2 from Chapter 1. Additionally, the other numeric functions (such as **NIntegrate** and **NSum**) may be invoked with a `WorkingPrecision` option; the Documentation Center should be consulted for more information.

Mathematical Conventions

Included in Table P.3 is a list of mathematical notation, and its meaning.

Note: In the case of the log, sin, cos, and similar functions, if there is any ambiguity as to what the function in question is and is not to be applied to, then parentheses or brackets are used to make it clear. For example $2 + \sin(3x + 1)$.

Notation Meaning

$\mathbb{N}$	The set of natural numbers
$\mathbb{Z}$	The set of integers
$\mathbb{Q}$	The set of rational numbers
$\mathbb{R}$	The set of real numbers
$\mathbb{C}$	The set of complex numbers
$\log x$	The *natural* logarithm of x
$\log_b x$	The logarithm of x (base b)
$\sin x$	The sine function applied to x
$\sin^n x$	The nth power of the sine function applied to x. That is $(\sin x)^n$.
$\sin^{-1} x$	The inverse-sine function applied to x
$\arcsin x$	The inverse-sine function applied to x
$\cos x$	The cosine function applied to x
$\cos^n x$	The nth power of the cosine function applied to x. That is $(\cos x)^n$.
$\cos^{-1} x$	The inverse-cosine function applied to x
$\arccos x$	The inverse-cosine function applied to x

Table P.3 Mathematical Notation

Chapter 1
Number Theory

This chapter includes the basics of the use of *Mathematica*, illustrated by fairly simple examples mostly involving integers. For this chapter you need to know what a sequence is, an infinite sum, summation notation, what a function is, and what a polynomial is. By the end of the chapter you should be comfortable using *Mathematica* for moderately complex tasks, and should be ready to learn the new commands required for doing specific mathematics such as calculus or linear algebra.

1.1 Introduction to *Mathematica*

Before we can set about exploring mathematics with *Mathematica* we need to know how to input basic commands into it. This section will introduce *Mathematica* and its most basic commands.

1.1.1 Inputting Basic Mathematica Expressions

At its absolute most basic, *Mathematica* can be used as sort of an overblown pocket calculator. We give it an expression to calculate, and *Mathematica* performs the calculation. Note, that to perform the computation, it is required to use *shift-enter* after typing the command; that is to hold the *shift* key down whilst typing the *enter* key.

In[1]:= `1 + 2`

Out[1]= 3

In[2]:= `2 * 3^5 + 12 − 2`

Out[2]= 496

Mathematica input is formatted, by default, as text and is not typeset in the same manner as we would write the mathematics on paper. In the above example we asked *Mathematica* to calculate $2 \cdot 3^5 + 12 - 2$. Order of operations is important here, and if we are ever in doubt, we can always force the desired ordering by using parentheses, just as we would when writing mathematics on paper.

We can have *Mathematica* compute even more complicated statements involving factorials, trigonometric functions, and a lot more besides. In the following example, we calculate $\left(\sin(\pi/2) + 12! \cdot \sqrt{12} \right) / e^4$.

In[3]:= `(Sin[Pi / 2] + 12! * Sqrt[12]) / E^4`

Out[3]= $\dfrac{1 + 958003200\,\sqrt{3}}{\mathrm{e}^4}$

Notice that in this last example that *Mathematica* didn't provide a decimal number as the answer to the input. This rather nicely illustrates a key difference between *Mathematica* and your pocket calculator. *Mathematica* is a Computer Algebra System (CAS), and performs its calculations as exactly as possible. When no exact number occurs, *Mathematica* provides an exact expression. So e^4 and $\sqrt{3}$ are exact values, whereas 54.59815003 and 1.732050808 (respectively) are decimal approximations. *Mathematica* gives us the exact value unless we specifically ask it otherwise. To accomplish this we either use the **N** command, or put a decimal point next to a constant.

In[4]:= `N[(Sin[Pi / 2] + 12! * Sqrt[12]) / E^4]`

Out[4]= 3.03913×10^7

In[5]:= `(Sin[Pi/2] + 12! * Sqrt[12.]) / E^4`

Out[5]= 3.03913×10^7

Attention should be drawn, in the second example, to the period after the 12 in the square root. This is a shorthand for—in this case—12.0, and tells *Mathematica* that the value is a decimal. The inclusion of a decimal in an expression is one way to tell *Mathematica* that we wish a numeric (rather than symbolic) calculation to be performed.

Inasmuch as *Mathematica* is a CAS, we ought to expect that it can do some basic algebra. In point of fact it can, and a good start is to work with basic polynomials.

In[6]:= `3 x^2 + 2 x^3 + 3`

Out[6]= $3 + 3x^2 + 2x^3$

In[7]:= `4 x^2 + 9 x^2`

Out[7]= $13x^2$

Notice that *Mathematica* automatically adds the like terms together.

In[8]:= `3 x^4 * 3 y^2`

Out[8]= $9x^4 y^2$

Notice the coloring on the variable. A blue variable signifies that the variable is unknown, or has no value. We'll make sense of this shortly. Notice, also, that we did not need to specify the ***** operator for the multiplication **x** and it's coefficient. Mathematica automatically understands that a number followed by an algebraic variable (see below) signifies that the two are multiplied. Furthermore, we may use a space instead of the ***** operator to signify multiplication if we wish.

In general, *Mathematica* considers any word it does not otherwise know to be an algebraic variable. We could use the strings "alice" and "bob" in place of x and y and *Mathematica* will treat them just like any other algebraic variable.

In[9]:= `alice + bob`
        `% + 2`
        `% + 5`

Out[9]= $alice + bob$

Out[10]= $2 + alice + bob$

Out[11]= $7 + alice + bob$

The previous example demonstrates a couple of things that we have not yet seen before, which we take some time to highlight.

First there were three commands inside of the one input. Each command was on its own line, which is how *Mathematica* identifies the individual commands. Alternatively, a command may be ended by a semicolon (;) if we wish for the output of that command not to be displayed. Doing so is called "output suppression". Make no mistake, however, only the output is being suppressed, the command is still performed.

```
In[12]:= alice + bob;
         % + 2;
         % + 5
```

Out[12]= $7 + alice + bob$

In the above example we chose to see only the output of the final command, but we could, should we have wished, chosen differently. We may have elected to see only the output of the second command, or perhaps only the output of the first and third commands. To achieve such a thing, we would simply suppress the output of our choice with a semicolon, and leave the commands whose output we wished to see with no semicolon.

In addition to allowing the suppression of output, the use of the semicolon also allows multiple commands to co-exist on the same line. When this is done, the line of input is known as a "compound expression." Note that only the final sub-expression may omit the semicolon, and so only the final sub-expression may produce output. Note, also, that it turns out that compound expressions do not interact well with the **%** operator (see below). The interested reader should search for "compound expression" in *Mathematica*'s Documentation Center for more information.

The above examples show another thing which we have not seen before; the special operator **%**. This operator is used to refer to the value of the most recent calculation. Even if the output of the most recent calculation was suppressed with a semicolon, the **%** will still refer to the value of the computation. The **%** is a very useful tool, but be careful when using it, as the most recent calculation performed may not be the one performed on the previous input (see Exercise 3).

1.1.2 Variables

In addition to providing an unknown quantity for working with algebra, *Mathematica* variables (any string of characters that *Mathematica* doesn't know to be something else) can also have values assigned to them. There are two primary reasons to do this. The first is to give a name to an expression you want to use later; the other is to store a value that might change. In reality, these are two sides to the same coin. To assign a value to a variable, we use the assignment operator (**=**).

```
In[13]:= A = 2
```

Out[13]= 2

Note that the **A** variable is colored black instead of blue. This is because it is now a known variable, as it has a value assigned to it. Once we have assigned a value to a

variable, when we use the variable name, *Mathematica* automatically uses its value for the computation.

```
In[14]:= A
         A + 2
         A^10
```

Out[14]= 2

Out[15]= 4

Out[16]= 1024

```
In[17]:= poly = 3 x^3 + 2 x^2 + 3 x
         poly + 3 x^2 + x
```

Out[17]= $3x + 2x^2 + 3x^3$

Out[18]= $4x + 5x^2 + 3x^3$

If we were to assign a value to `x`, then the `poly` variable would then be the polynomial for that particular `x`. Alternatively, if we want to keep both the general polynomial as well as calculate its value for different values of `x`, then we may use the substitution operator (`/.`).

```
In[19]:= poly /. x -> 3
         poly /. x -> a^2
         poly
```

Out[19]= 108

Out[20]= $3a^2 + 2a^4 + 3a^6$

Out[21]= $3x + 2x^2 + 3x^3$

We can even use variable names as input to functions, and *Mathematica* still uses the value of the variable. We look at exactly what a function is in the next subsection, but for now notice that in the example below, *Mathematica* has plotted the cubic $3x^3 + 2x^2 + 3x$ to which we gave the name *poly* in an earlier example above.

```
In[22]:= Plot[poly, {x, -1, 1}]
```
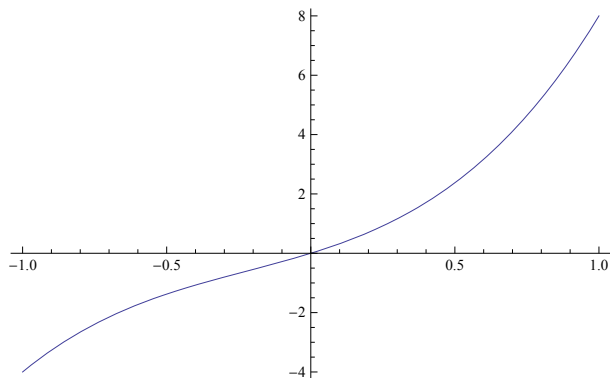
Out[22]=

We may assign any valid *Mathematica* input to a variable, and *Mathematica* will always use the value of the variable wherever we use its name. This can be very useful, if occasionally confusing. A common theme throughout the course of the book is that one must remain aware of the subtleties of the system when utilizing it. With great power comes great responsibility.

We can, if we wish, have *Mathematica* perform a calculation, and store the result of that calculation in the variable.

In[23]:= `value = 4 * 12 + 13^5`

Out[23]= 371341

So far we have used variables to give a name to something we wish to somehow utilize later on. However, as mentioned above, sometimes we want to use a variable as a storage box for values that can and will change. This is really no different from reassigning a variable name to a different expression.

In[24]:= `a = 2`
       `2 a`

Out[24]= 2

Out[25]= 4

In[26]:= `a = 4`
       `2 a`

Out[26]= 4

Out[27]= 8

Such a technique, however, lets us use a variable to store intermediate results of a calculation in progress.

In[28]:= `total = 0`

Out[28]= 0

In[29]:= `total = total + 12`

Out[29]= 12

In[30]:= `total = total + 11`

Out[30]= 23

Note here that not only are we reassigning the value of the variable named `total` but we are also using its current value to calculate its new value. The line `total = total + 12` means, in English, something along the line of "set the value of total to be its current value plus 12." What *Mathematica* actually does is evaluate the right-hand side of the definition (which in this case is `total + 12`) and assign the result of that calculation back to the variable name on the left-hand side. These need not be the same variable, for instance:

In[31]:= `subtotal1 = 12`
       `subtotal2 = 23`

Out[31]= 12

Out[32]= 23

In[33]:= `total = subtotal1 + subtotal2`

Out[33]= 35

The above code is perfectly valid, and is really just another case of the basic assignment of a variable we saw at the beginning of the subsection.

Finally, we note that there are, in fact, two assignment operators; `=` and `:=`. We have already seen the former, and we will see the latter again when we begin making our own function in Section 1.2.1. The difference between the two lies in when *Mathematica* will evaluate the expression being assigned to the variable. With the `=` operator, the expression to be assigned is evaluated first, and then assigned to the variable. With the `:=` operator, the expression being assigned is not evaluated at all before assignment to a variable, but is instead evaluated each time the variable is used. This is called a *delayed definition*. The distinction is subtle, and in many cases behaves identically, but it is worth being aware of it.

1.1.3 Functions

Having dealt with basic input and variables, we now move on to another key *Mathematica* concept, that of functions. A function, simplistically speaking, is just something that takes an input (perhaps several inputs) and produces output. In order to avoid confusion between a function input and *Mathematica* input, we will call the function inputs either arguments or parameters. We have already seen a couple of functions in our examples thus far such as **Sin** and **N**. Functions are written in the form of **Name[argument, argument, ...]** where **name** is the name of the function, and **argument, argument,** ... is a comma separated list of the function arguments. To start with we deal mostly with functions that take a single argument, and which should look very similar to the functions seen in first-year calculus.

In[34]:= **FactorInteger[1573]**

Out[34]= $\{\{11, 2\}, \{13, 1\}\}$

In[35]:= **Factor[x^4 - 2 x^3 - 13 x^2 + 14 x + 24]**

Out[35]= $(-4 + x)(-2 + x)(1 + x)(3 + x)$

In[36]:= **Simplify[(x^2 + x) / (x^3 + 2 x)]**

Out[36]= $\dfrac{1 + x}{2 + x^2}$

Note here that every one of the above functions had only a single argument. In several cases that argument was a moderately complicated *Mathematica* expression, but it was a single argument nonetheless.

A list of commonly used basic functions is included at the very beginning of this book, just after the preface. For more complicated functions, *Mathematica*'s own help files are always a good source of information. Every example thus far has been a function that comes built-in to *Mathematica*. In the next section we start creating our own functions, but for now we look at an example of a function that takes multiple arguments.

In[37]:= **Collect[x^2 * y + x * y + x^2 * z + x * z + x * y^2 * z^2, x]**

Out[37]= $x^2(y + z) + x\left(y + z + y^2 z^2\right)$

The astute reader will also recognize that the plot function we saw earlier was also a function that took multiple inputs.

1.1.4 Lists, Sets, and Sequences

In *Mathematica*, a list refers to a group of expressions separated by commas, and contained within braces ({,}). For example, the following demonstrates two *Mathematica* lists.

```
In[38]:= {1, 2, 3}
         {poly, 5, bob}
```

Out[38]= $\{1, 2, 3\}$

Out[39]= $\{3x + 2x^2 + 3x^3, 5, bob\}$

Note how, in the case of the second list, the `poly` variable was replaced by its value in the output. This was to be expected; in fact any valid *Mathematica* expression may be an element of a list. Furthermore, a list as a whole is a valid *Mathematica* expression, and may potentially be used anywhere an expression is appropriate. The astute reader might have even noticed that the `Plot` function, earlier, had a list as its second parameter. A more straightforward example is to store a list in a variable.

```
In[40]:= L1 = {a, b, c}
         L2 = {1, 2, 3}
         L3 = {i, ii, iii}
```

Out[40]= $\{a, b, c\}$

Out[41]= $\{1, 2, 3\}$

Out[42]= $\{i, ii, iii\}$

Inasmuch as a list is a valid *Mathematica* expression, it may even be an element of another list. Note, however that If we have two lists and use them as the elements of a single list, the result is a list with two elements, each of which is also a list. This is known as a nested list.

```
In[43]:= {L1, L2, L3}
```

Out[43]= $\{\{a, b, c\}, \{1, 2, 3\}, \{i, ii, iii\}\}$

This behavior should be familiar to any reader who has studied elementary set theory; it is reminiscent of sets within sets. However, while similar in some ways, a list is different to a set. We look at this more closely, as well as how to use lists as sets below.

The preceding example was a nested list of depth 2; that is we had lists within a list. There's no reason we could not nest deeper. In fact we may combine lists and other elements within a list. Below is a complicated, and contrived[1], example.

```
In[44]:= {1, 2, Sin[7], {{}, {{{}, {x, {y}}}, Sqrt[12]}, 4}}
```

Out[44]= $\left\{1, 2, \mathrm{Sin}[7], \left\{\{\}, \left\{\{\{\}, \{x, \{y\}\}\}, 2\sqrt{3}\right\}, 4\right\}\right\}$

If we did not wish to nest the lists, but instead wished to create a list whose elements were the collection of all the elements of the lists `L1`, `L2`, and `L3`, we could have achieved this using the `Join` function.

---

[1] The astute reader might well point out that most of the examples in this section are contrived. This is a fair point, so long as one does not lose sight of the fact that the point of such examples are to demonstrate language features of *Mathematica*. More natural examples will begin in Section 1.3 and will continue throughout the book.

In[45]:= `Join[L1, L2, L3]`

Out[45]= $\{a, b, c, 1, 2, 3, i, ii, iii\}$

Lists can also be a convenient way of assigning multiple variables in a single command. For instance, if we wish to assign $i = 1$, $j = 2$, and $k = 3$ then we could use

In[46]:= `{i, j, k} = {1, 2, 3}`

Out[46]= $\{1, 2, 3\}$

In[47]:= `i`
`j`
`k`

Out[47]= 1

Out[48]= 2

Out[49]= 3

A list may contain no elements at all, in which case it is an empty list.. This is again reminiscent of set theory in which the empty set is the set with no elements. An empty list in *Mathematica* is referenced by an opening and closing brace with nothing (except, perhaps, space) between them. An empty list is still a valid list in its own right, and will nest inside of other lists as we saw above, however joining an empty list to any other list doesn't change the other list all.

In[50]:= `Join[{}, {a, b, c}]`
`Join[{a, b, c}, {}]`
`Join[{a, b}, {}, {c}]`

Out[50]= $\{a, b, c\}$

Out[51]= $\{a, b, c\}$

Out[52]= $\{a, b, c\}$

Using an empty list allows a list to be built up by parts within a variable, a little like the **total** variable was in the previous subsection.

In[53]:= `S = {}`
`S = Join[S, {a, b, c}]`
`S = Join[S, {d, e}]`

Out[53]= $\{\}$

Out[54]= $\{a, b, c\}$

Out[55]= $\{a, b, c, d, e\}$

It should be understood that the elements within a list have a very definite order. There is a first element, second element, third element, and so on and so forth. Furthermore, the elements of a list need not be unique. In the following list the first, third, and fifth elements are all the same. Similarly the fourth, sixth, and seventh elements are all the same as well.

In[56]:= `{1, a, 1, b, 1, b, b}`

Out[56]= $\{1, a, 1, b, 1, b, b\}$

This is a key way in which lists differ from sets. We may, if we wish, refer directly to an element of a list by using its numeric position, which is sometimes called the "index" of the element. *Mathematica* allows this through use of the indexing operator (`[[...]]`), or alternatively through the `Part` function.

```
In[57]:= S = {a, b, c, d, e, f, g, h, i, j}
```

Out[57]= $\{a, b, c, d, e, f, g, h, i, j\}$

```
In[58]:= Part[S, 3]
         S[[3]]
```

Out[58]= c

Out[59]= c

Note that the `Part` function is more limited in its use than the `[[...]]` operator. If we want to refer to a sub-list we must use the indexing operator. Sublists may be referred to either by specifying a list of the indices we want, or by specifying a range of indices in the form `m;;n`. Note that when we ask for a sublist, the output is, itself, a list.

```
In[60]:= S[[{2, 4, 6, 8}]]
         S[[5;;8]]
         S[[;;4]]
         S[[6;;]]
```

Out[60]= $\{b, d, f, h\}$

Out[61]= $\{e, f, g, h\}$

Out[62]= $\{a, b, c, d\}$

Out[63]= $\{f, g, h, i, j\}$

The latter two commands in the above example show "half ranges" (`;;n` and `m;;`) which mean "the first n elements" and "from the mth to the last element," respectively. An alternate way of thinking of these ranges is to consider that *Mathematica* automatically inserts the beginning or the end index for the missing number as appropriate.

We have seen many similarities between lists and sets, and very few differences. The primary difference is the allowance of repeated elements in a list. Nonetheless, we may treat a list as a set, if need be. In fact, if we wish to perform set computations, we must use lists. *Mathematica* provides functions to perform the usual union, intersection, and set minus operations (`Union`, `Intersection`, and `Complement` respectively). Each of these functions will always produce a list whose elements are sorted, and with no duplicates.

If we wish to take a regular list and turn it into a list that we can think of as a set, a simple method is to use the `Union` command as follows

```
In[64]:= Union[{3, 1, 2, 2, 1}]
```

Out[64]= $\{1, 2, 3\}$

This behavior is merely a special case of the general behavior of the `Union` function, which is to take a collection of lists as function arguments, and to calculate a sorted, duplicate-free list containing all elements found in those lists. The lists that are the arguments may contain duplicates, and need not be sorted; the function takes these in its stride. In the above example we had only one argument, yet the output was precisely a sorted, duplicate-free list containing elements of that one argument. There is no limit to the number of arguments that the `Union` argument can take.

In[65]:= `Union[{1, 2, 3, 5}, {4, 2, 3}]`
         `Union[{1, 2}, {2, 3}, {3, 4}, {4, 5}, {5, 6}, {7}]`

Out[65]= $\{1, 2, 3, 4, 5\}$

Out[66]= $\{1, 2, 3, 4, 5, 6, 7\}$

The **Intersection** command works similarly; it takes a collection of lists as function arguments, and calculates a sorted, duplicate-free list containing the elements which are common to the all those lists. There is no limit to the number of arguments, and the arguments may contain duplicates, and need not be sorted. Note that if we try to compute the intersection of a single list, the output will be the same as if we had tried to compute the union of that same single list; a sorted, duplicate-free list of all the items in that one list.

In[67]:= `Intersection[{3, 1, 2, 2, 1}]`

Out[67]= $\{1, 2, 3\}$

In[68]:= `Intersection[{1, 2, 3, 5}, {4, 2, 3}]`
         `Intersection[{1, 2}, {2, 3}, {3, 4}, {4, 5}, {5, 6}, {7}]`

Out[68]= $\{2, 3\}$

Out[69]= $\{\}$

These functions work as should be expected to anybody familiar with elementary set theory, with the extra flexibility that they may be performed on arbitrary lists, not just on sorted, duplicate-free lists. This extra flexibility allows us, for practical intents and purposes, to consider any *Mathematica* list to be a set. Be aware that we may only treat finite sets in this manner; the author knows no easy way of having *Mathematica* perform computations involving infinite sets.

We come now to the **Complement** function. In elementary set theory, the complement of a set, S say, often written $\overline{S}$ is the set of all elements which are not contained in S (compared against some understood universal set, U say). We can think of this as $S \setminus U$ if we wish. This is not, however, what the **Complement** function calculates for us. Instead, as mentioned above, the **Complement** function is more of a set minus operation. It takes a collection of lists as function arguments, and calculates a sorted, duplicate-free list containing all the elements from the first list which do not appear in any of the subsequent lists. As such, if we call **Complement** with a single argument, we will see the same behavior we would see with either **Union** and **Intersection** called with the same single list. This may potentially be confusing for readers familiar with the elementary set theory described above.

In[70]:= `Complement[{3, 1, 2, 2, 1}]`

Out[70]= $\{1, 2, 3\}$

In[71]:= `Complement[{1, 2, 3, 5}, {4, 2, 3}]`
         `Complement[{1, 2}, {2, 3}, {3, 4}, {4, 5}, {5, 6}, {7}]`

Out[71]= $\{1, 5\}$

Out[72]= $\{1\}$

In many cases, we will want the elements of a list to follow a fairly predictable pattern. We may produce some such lists using the handy command, **Table**. The simplest such list is a list which has the same element repeated some number of times. To do this we use the **Table** function in the following manner:

In[73]:= `Table[x, {4}]`

Out[73]= $\{x, x, x, x\}$

In this case, `x` could have been any expression we liked. For patterns other than a single repeated expression, we will need an expression involving an unknown variable of our choosing, k in the proceeding examples, that will compute the kth element of the list. For instance, to print out the first ten squares we simply input the following:

In[74]:= `Table[k^2, {k, 1, 10}]`

Out[74]= $\{1, 4, 9, 16, 25, 36, 49, 64, 81, 100\}$

or for something a little more complicated:

In[75]:= `Table[3 * k^2 + k / 2, {k, 4, 15}]`

Out[75]= $\left\{50, \dfrac{155}{2}, 111, \dfrac{301}{2}, 196, \dfrac{495}{2}, 305, \dfrac{737}{2}, 438, \dfrac{1027}{2}, 595, \dfrac{1365}{2}\right\}$

Note the argument `{k, 4, 15}`, above. This is known as an *iterator*. It tells the `Table` function that the variable `k` will begin at 4 and will increment until it reaches 15. We will see many other functions that use iterators over the course of the book, and we even explore some of the complexities of iterators in Exercise 5. Regrettably, information on iterators is a little tricky to find. The interested user will need to search for "Some General Notations and Conventions" in the Documentation Center.

First-year calculus students that have studied sequences will be familiar with sequences written in the form $\{x_k\}_{k=a}^{\infty}$ where x_k is an expression in k. Finite sequences are often written in the form $\{x_k\}_{k=a}^{b}$. This is precisely what the `Table` command lets us compute; finite sequences of this form. For the first example we computed the sequence $\left\{k^2\right\}_{k=1}^{10}$, and in the second example, above, we computed the sequence

$$\left\{3 \cdot k^2 + \frac{k}{2}\right\}_{k=4}^{15}$$

In general, to have *Mathematica* print out the sequence $\{x_n\}_{n=a}^{b}$, where x_n is some expression involving n, we use the command `Table[x_n, {n, a, b}]`.

There are a couple of other forms of the `Table` function which can produce more complicated—usually nested—lists. We do not discuss these here, however the interested user is encouraged to look up both of these function in the help files. Additionally, there is a similar function to `Table` named `Array` that work specifically with functions instead of more general expressions. There are some technical reasons one might use `Array` instead of `Table`, none of which will concern us during the course of this book. We merely mention it here so the interested reader may find out more by reading the help files.

1.1.5 Sums and Products

Having dealt with the basics of *Mathematica*, we can now move onto some mathematics a little more like we might see at the beginning of a first-year math course. We begin by looking at how *Mathematica* can handle sums and products, both of the finite and the infinite variety. Recall that infinite sums are often called "series" when encountered in first-year calculus.

For large additions (or, indeed, for anything too complicated to use the + operator practically) *Mathematica* provides the `Sum` function. The `Sum` function works in a very similar manner to the `Table` command. The simplest usage of `Sum` is to add a bunch of terms together. For instance, to add the first 10 squares together, we input the following:

In[76]:= `Sum[k^2, {k, 1, 10}]`

Out[76]= 385

Note the similarity to the `Table` function; the syntax is identical, with the difference being that one function returns the terms in a list, while the other adds the terms together. If we happen to already have a list of the terms we wish to sum, then we may use the `Total` function to add them together.

In[77]:= `Total[{1, 4, 9, 16, 25, 36, 49, 64, 81, 100}]`

Out[77]= 385

The `Sum` function can also handle symbolic partial sums, and even series. This is achieved by simply modifying the bounds of the sum. For instance, to find an expression for the partial sum

$$\sum_{k=1}^{N} k^2$$

we would input:

In[78]:= `Sum[k^2, {k, 1, N}]`

Out[78]= $\frac{1}{6} N (1+N) (1+2N)$

For an infinite sum, we simply use `Infinity` as the upper bound. In this case, it should be clear to anybody familiar with sequences and series that the sum of squares will not converge.

In[79]:= `Sum[k^2, {k, 1, Infinity}]`

Sum::div: Sum does not converge. »

Out[79]= $\sum_{k=1}^{\infty} k^2$

Inasmuch as the sum does not converge, *Mathematica* gives us an error message that the sum does not converge, and outputs the symbolic form of the sum. For the sakes of good demonstration, let's go and compute a sum that we know does converge.

In[80]:= `Sum[1 / k^2, {k, 1, Infinity}]`

Out[80]= $\frac{\pi}{6}$

As well as sums, *Mathematica* can also handle products with the `Product` function. This function operates pretty much identically to the `Sum` function.

In[81]:= `Product[k^2, {k, 1, 10}]`

Out[81]= 13 168 189 440 000

Unfortunately, *Mathematica* provides no function to multiply the elements of a list for in a manner analogous the `Total` function. We may, however, achieve this result using the indexing operator.

In[82]:= `L = {1, 4, 9, 16, 25, 36, 49, 64, 81, 100}`
`Product[L[[k]], {k, 1, 10}]`

Out[82]= 13 168 189 440 000

We may handle symbolic partial products and infinite products in precisely the same manner as with the `Sum` function.

In[83]:= `Product[k^2, {k, 1, N}]`

Out[83]= $(N!)^2$

In[84]:= `Product[k^2, {k, 1, Infinity}]`

Product::div: Product does not converge. »

Out[84]= $\displaystyle\prod_{k=1}^{\infty} k^2$

Finally, to see an example of an infinite product that does converge we take the following calculation, known as the Wallis product.

In[85]:= `Product[4 n^2 / (4 n^2 - 1), {n, 1, Infinity}]`

Out[85]= $\dfrac{\pi}{2}$

1.1.6 Pre-, Post-, and Infix Function Notation

Functions, as we introduced them in Section 1.1.3 and have since seen them have always had the form `Name[argument, argument, ...]`. This is all well and good, however sometimes this can get cumbersome, especially if we have a few functions interacting with each other. Take for example the following computation.

In[86]:= `N[Sqrt[Sin[3 / 2]]]`

Out[86]= 0.998747

We have asked *Mathematica* to compute for us a numeric approximation of $\sqrt{\sin 3/2}$. While relatively straightforward, this is already becoming a little cumbersome to read. We may produce the same result using the "postfix" form of the function `N`.

In[87]:= `Sqrt[Sin[3 / 2]] // N`

Out[87]= 0.998747

The `// N` here means "apply the `N` function to this". Whatever appears on the left hand side of the `//` operator is considered an argument, and whatever appears on the right hand side is considered to be a function name. In other words, the postfix form of a function is `argument // Function`. Note that that this form of a function only makes sense for a function that takes a single input; it cannot work for functions of more than one input. However, this function form will work for any function that takes a single argument; nothing special needs to be done to allow a function to work this way. Postfix notation is just another way to tell *Mathematica* to use a function.

In a very similar manner, there is also a "prefix" notation. In this case the operator is `@`, and the left hand side of this operator is considered to be a function name, and the right hand side a single argument. In other words, the form is `Function @ argument`.

Everything else that was stated, above, for prefix notation, is true for postfix notation. In the case of our numeric approximation, above, if we were to use prefix notation the command would be as follows.

In[88]:= **N @ Sqrt[Sin[3 / 2]]**

Out[88]= 0.998747

Finally we have the "infix" notation. Unlike the pre- and postfix notations, infix notation applies only to functions that take exactly two arguments. The form the function takes is **argument1 ~Function~ argument2**, and is equivalent to the function in the form of **Function[argument1, argument2]**. Observe.

In[89]:= **a ~F~ b**

Out[89]= $F[a, b]$

It should hopefully be expected, in light of the both pre- and postfix notation, that any function that can take exactly two inputs may be used in infix notation. Unfortunately, our previous example is not appropriate for infix notation; it does not have any functions that take two arguments. However, we have already seen a function which lends itself very nicely to infix notation, the **Join** function.

In[90]:= **{1, 2, 3} ~Join~ {3, 4, 5}**

Out[90]= $\{1, 2, 3, 3, 4, 5\}$

We may chain these notations together, if we wish, and the results are fairly predictable. For instance both **a // F // G** and **G @ F @ a** are equivalent to **G[F[a]]**. However, the effect of chaining infix notations together is not immediately obvious. Observe the following.

In[91]:= **{1, 2, 3} ~Join~ {3, 4, 5} ~Join~ {5, 6, 7}**

Out[91]= $\{1, 2, 3, 3, 4, 5, 5, 6, 7\}$

We have produced exactly the same output, as if we had issued the command **Join[{1, 2, 3}, {3, 4, 5}, {5, 6, 7}]**. It might be tempting to think that combing infix notations like **a ~F~ b ~F~ c** would be equivalent to **F[a, b, c]**, but this is not the case. Remember that infix notation works on functions of exactly two arguments. The **Join** function can, in fact, work on any number of arguments. Nonetheless, the **Join** can certainly be called with exactly two arguments, and this is what happens when we use the infix notation with that function. To better demonstrate what happens when we chain infix functions together, observe the following.

In[92]:= **a ~F~ b ~G~ c**

Out[92]= $G[F[a, b], c]$

What we see here is a nested function. If we think about this for a minute, hopefully this makes perfect sense. This is equivalent to **(a ~F~ b) ~G~ c**. The first infix function was found, evaluated, and treated as an argument to the second infix function. The reason this worked the way it did with the **Join** function is that joining lists is transitive. Suppose we have lists A, B, and C, say. If we join A and B first, and then join the result with C we have a list containing all the elements of all three. If we were to join B and C first, then join the result to A we would still have the same result. Of course, both of these are also equivalent to just joining all three lists in one go.

```
In[93]:=  Join[Join[{1, 2, 3}, {3, 4, 5}], {5, 6, 7}]
          Join[{1, 2, 3}, Join[{3, 4, 5}, {5, 6, 7}]]
          Join[{1, 2, 3}, {3, 4, 5}, {5, 6, 7}]
```

Out[93]= $\{1, 2, 3, 3, 4, 5, 5, 6, 7\}$

Out[94]= $\{1, 2, 3, 3, 4, 5, 5, 6, 7\}$

Out[95]= $\{1, 2, 3, 3, 4, 5, 5, 6, 7\}$

It is left to the discretion of the reader as to how they use these alternate notations for functions. If used poorly, they can become just as confusing as many nested functions with the usual `F[...]` form. If used cleverly, however, these alternate forms can potentially make our input commands much easier to understand.

1.2 Putting It Together

In the previous section, we looked at inputting single commands into *Mathematica*. These may be thought of as building blocks. In this section we begin to put these building blocks together to produce more complicated calculations. We also begin to introduce some more serious mathematics in order to motivate or illustrate the particular *Mathematica* constructions with which we are dealing.

1.2.1 Creating Functions

In the previous section we have seen what a function is. In addition to the built-in functions, we may create our own. A function definition, at it's simplest, takes the form `function[arguments] := Expression`. In this case the arguments will be a comma separated sequence of variable names, each one ending with an underscore (`_`), otherwise known as a "blank."

```
In[96]:=  p[x_] := 3 x^2 + 4 x - 2
```

The above function takes a single expression as input (which it calls x), and then uses that expression to perform the calculation $3x^2 + 4x - 2$. The name of the input variable is entirely arbitrary and could be any valid *Mathematica* variable name. The blank (`_`) after the variable name is very important, however. The blank tells *Mathematica* that the x variable can be any valid mathematica expression. This is a form of pattern matching inside of *Mathematica* which we will later use to restrict the arguments of our functions to particular types of mathematical objects or *Mathematica* expressions. For the moment, however, we content ourselves with allowing any valid expression for our arguments. Fortunately, our function behaves quite sensibly with a variety of different arguments.

```
In[96]:=  p[2]
          p[4]
          p[A]
          p[{2, 4}]
```

Out[96]= 18

Out[97]= 62

Out[98]= $-2 + 4A + 3A^2$

Out[99]= $\{18, 62\}$

It is interesting to see that in the last example above that the function was applied to each element of the list we used as our argument. Many single-argument functions, when given a list as an argument, will perform their operation individually on each element of the list. The most obvious exception to this behavior will be from functions which are written to operate on lists specifically, such as the **Reverse** function (the reader should look this function up in the Documentation Center).

How about we look at something something that might be a bit more familiar. Recall from first-year calculus that

$$\sum_{k=1}^{\infty} \frac{1}{k}$$

diverges, whereas

$$\sum_{k=1}^{\infty} \frac{1}{k^2}$$

converges. This is a good reminder that the divergence test is inconclusive in the case where the sequence has a limit of 0 because $1/k \to 0$ (as $k \to \infty$) yet the series diverges but $1/k^2 \to 0$ yet the series converges.

In[100]:= `a[n_] := 1 / n ^ 2`

In[100]:= `Limit[a[N], N -> Infinity]`

Out[100]= 0

To attempt to verify the convergence of the $1/k^2$ series, we can try the ratio test with *Mathematica*, but unfortunately this test also proves inconclusive.

In[101]:= `Abs[a[N + 1] / a[N]]`
          `Limit[%, N -> Infinity]`

Out[101]= $\text{Abs}\left[\dfrac{N^2}{2 + N}\right]$

Out[102]= 1

We side-track for a moment and notice that for every N we have

$$\left|\frac{a(N+1)}{a(N)}\right| < 1$$

This is not hard to verify by hand, but we ask *Mathematica* for a verification anyway. Observe that the fraction is undefined for $N = -1$, but this isn't a concern for us as we're only interested in natural values of N. However, we may need to be a little specific in what we ask *Mathematica*.

In[103]:= `Abs[a[N + 1] / a[N]] < 1`

Out[103]= $\text{Abs}\left[\dfrac{N^2}{2 + N}\right] < 1$

The less than operator (**<**) should have produced an output of either True or False. The fact that it simply re-iterated the input tells us that *Mathematica* does not know the answer. With a bit of thought this is hopefully not too surprising. We already know

the question doesn't make any sense for $N = -1$, and the statement is false for $N < -1$. As observed above, however, we're only interested in strictly positive values of N.

In[104]:= **Refine[%, N > 0]**

Out[104]= True

The **Refine** function tells *Mathematica* to evaluate the expression just as it always would, but with some extra assumptions. These extra assumptions form a single argument to the function; multiple assumptions should be joined using logical and (**&&**) and/or logical or (**||**) operators. In the above example, we had a single assumption that that $N > 0$. Note that the **Simplify** function may also accept an assumption as its second argument. The difference between **Simplify** and **Refine** is that **Simplify** will always take pains to try to find the simplest form for output, whereas **Refine** will not. The interested reader should look up "Assumptions and Domains" in *Mathematica*'s Documentation Center.

Now, returning to the question of convergence of $1/k^2$, we have yet to verify analytically whether the series converges or diverges. At this stage it is often a good idea to postpone our desire for an analytic answer in favor of an more computational (or experimental) approach. With this in mind, recall that an infinite sum is a limit of partial sums. That is,

$$\sum_{k=1}^{\infty} a(k) = \lim_{N \to \infty} \sum_{k=1}^{N} a(k)$$

As our next attempt we attempt to see how the partial sums behave. To do this we begin by creating a function to compute the Nth partial sum of the $1/k^2$ series.

In[105]:= **par[N_] := Sum[a[k], {k, 1, N}]**

We now have **par** as a function of **N** where **N** is the number of terms to sum. Let us now see how this sum behaves with increasing values

In[106]:= **par[1]**
          par[10]
          par[50]

Out[106]= 1

Out[107]= $\dfrac{1\,968\,329}{1\,270\,080}$

Out[108]= $\dfrac{3\,121\,579\,929\,551\,692\,678\,469\,635\,660\,835\,626\,209\,661\,709}{1\,920\,815\,367\,859\,463\,099\,600\,511\,526\,151\,929\,560\,192\,000}$

Unfortunately, that wasn't particularly helpful. We now make two refinements. Firstly, we will use a **Table** instead of typing the function for each partial sum at the command prompt. Secondly, we will ask for a numerical approximation of the partial sums using the postfix notation from Section 1.1.6 for readability purposes. In addition to these refinements, we will try some more ambitious partial sums, going as high as the 100,000th in increasing powers of 10.

In[109]:= **Table[par[10^p] // N, {p, 1, 5}]**

Out[109]= $\{1.54977, 1.63498, 1.64393, 1.64483, 1.64492\}$

We should perhaps be a little careful here. We used the temporary variable p in the previous example, although prior to that we defined p to be a function. Fortunately for us, *Mathematica* understood p to be a temporary variable for the **Table** command, and

did not confuse it with the function we defined earlier. In fact, the variable p found in the arguments of the `Table` function is a completely different variable to the one we defined previously. This new p exists only temporarily, yet while it exists it supersedes any previous definition of p. This is very convenient, unless we happened to want to use the function `p` within the list we were creating, in which case we should use a different temporary variable. We would have a similar situation if we were foolish enough use N as a temporary variable, when we wanted to obtain decimal approximations with the `N`.

Getting back to our partial sums, we have computed the 10th, 100th, 1,000th, 10,000th, and 100,000th partial sums and can see some evidence that the series converges to a value somewhere in the vicinity of $1.644\ldots$. As we calculate progressively larger and larger partial sums, the value of those partial sums seems to change less and less. At this point we may as well ask *Mathematica* if it can give us an answer for the infinite sum.

In[110]:= `par[Infinity]`
          `N[%]`

Out[110]= $\dfrac{\pi^2}{6}$

Out[111]= 1.64493

Alternatively, we could have simply had the `Sum` function compute the infinite sum for us directly (without the need for the `Limit` function).

In[112]:= `N[Infinity]`

Out[112]= $\dfrac{\pi^2}{6}$

Ih either case, there we have it. It looks very much as if the series converges to $1/6\,\pi^2$, provided we trust *Mathematica*'s limit computation. To verify this analytically we would need to use the integral test, but we shall not do so here. We will see how to perform calculus with *Mathematica* in Chapter 2, but performing the integral test for this series is left as an exercise to the reader.

1.2.2 Loops

Until now if we wanted to perform something several times, we either typed it in multiple times at the command prompt, or we constructed a list. Sometimes these options aren't satisfactory. Let us revisit our example of the series $\sum(1/k^2)$. Earlier we used `Table` to print out the sequence

$$\left\{\sum_{k=1}^{10^N}\frac{1}{k^2}\right\}_{N=1}^{5}$$

which quite conveniently demonstrated the convergence of the series. The list was easy enough to read and understand. However, suppose we wanted to see more values of the sequence. Let's look at the values of the partial sums for values of N as the first 17 powers of 2. That is, $N = 2, 4, 8, 16, \ldots, 131{,}072$. We have chosen 17 as our limit because 2^17 is the first power of 2 that is larger than our previously computed largest partial sum of 100,000.

In[113]:= **`Table[par[2^p] // N, {p, 1, 17}]`**

Out[113]= $\{\, 1.25, 1.42361, 1.52742, 1.58435, 1.61417, 1.62943, 1.63715, 1.64104,$

$1.64298, 1.64396, 1.64445, 1.64469, 1.64481, 1.64487, 1.6449, 1.64492,$

$1.64493 \,\}$

That's a bit of a mess, but not completely unreadable. Now, we would like to see which values of 2^p produce which of those outputs. We can work it out by counting from the left and working out the corresponding power of 2, but it would be nicer if we could just see it. As such, we tell *Mathematica* to compute a list whose elements are of the form $par[n] == m$ where m is the value of the nth partial sum.

In[114]:= **`Table["par"[2^p] == (par[2^p] // N), {p, 1, 17}]`**

Out[114]= $\{\, par[2] == 1.25, par[4] == 1.42361, par[8] == 1.52742,$

$par[16] == 1.58435, par[32] == 1.61417, par[64] == 1.62943,$

$par[128] == 1.63715, par[256] == 1.64104, par[512] == 1.64298,$

$par[1\,024] == 1.64396, par[2\,048] == 1.64445, par[4\,096] == 1.64469,$

$par[8\,192] == 1.64481, par[16\,384] == 1.64487, par[32\,768] == 1.6449,$

$par[65\,535] == 1.64492, par[131\,072] == 1.64493 \,\}$

Note the quotes around the first **`"par"`**. These quotes tell *Mathematica* that we want to refer to the variable as a symbol only, and not to evaluate it.[2] We've been careful to only put the "par" in quotes, not the entire expression, so that the **`2^p`** expression was evaluated. The end result was, as we saw, was output of par[2], par[4], par[8], and so on without the value of the function being computed.

This evaluation suppression, as we might call it, has also allowed us to cheat a little with the **`==`** operator. Note here that the **`==`** operator is for evaluating whether expressions are equal, and is similar to the **`<`** operator we have used previously. It is important not to get this confused with the **`=`** operator, which is for assignment of variables. Ordinarily, the **`==`** operator (much like the **`<`** operator) will produce an output of "true" or "false", based on whether the expressions it operates on are equal when they are evaluated. Like the **`<`** operator and, in fact, all of these logical comparison operators (i.e., operators which compare two things and report back a "true" or "false"), the **`==`** operator will simply return the expression un-evaluated if it does not know the answer. By telling *Mathematica* not to evaluate the **`par`** function on the left hand side, we have prevented the operator from being able to determine equality, and so it simply returns the whole expression as output. The end result, from our point of view, is a very readable output with the function and its input argument on one side, and the evaluated example on the other side.

Unfortunately, even with the extra readability afforded by our cunning suppression of the evaluation of **`par`**, the list we've produced is still a mess. We might improve matters if we could somehow put each equality on its own line, instead the one big sequence we currently have. We might achieve this by simply typing out all 20 expressions one after the other, but this would be slow and tedious, and would not work well if we wanted many computations to be performed. Fortunately, *Mathematica* provides a mechanism for such repeating calculations as these, the **`Do`** function.

---

[2] In fact, this technique may be applied to any *Mathematica* expression, however it's a little simpler to think about if we restrict it to only variables.

```
In[115]:= Do[
            Print["par"[2^p] == (par[2^p] // N)],
            {p, 1, 17}
          ]
```

Out[115]= $\text{par}[2] == 1.25,$

Out[116]= $\text{par}[4] == 1.42361,$

Out[117]= $\text{par}[8] == 1.52742,$

Out[118]= $\text{par}[16] == 1.58435,$

Out[119]= $\text{par}[32] == 1.61417,$

Out[120]= $\text{par}[64] == 1.62943,$

Out[121]= $\text{par}[128] == 1.63715,$

Out[122]= $\text{par}[256] == 1.64104,$

Out[123]= $\text{par}[512] == 1.64298,$

Out[124]= $\text{par}[1\,024] == 1.64396,$

Out[125]= $\text{par}[2\,048] == 1.64445,$

Out[126]= $\text{par}[4\,096] == 1.64469,$

Out[127]= $\text{par}[8\,192] == 1.64481,$

Out[128]= $\text{par}[16\,384] == 1.64487,$

Out[129]= $\text{par}[32\,768] == 1.6449,$

Out[130]= $\text{par}[65\,535] == 1.64492,$

Out[131]= $\text{par}[131\,072] == 1.64493$

Now this is much easier to read. *Mathematica* has happily calculated the expression **"par"[2^p] == par[2^p] // N** for us, and has done so 20 times, each time with the value of **p** increased by 1. After each calculation it has output the result of the calculation just as it would have if we had entered it manually at the command prompt, although we did need to specifically ask for this output using the **Print** function. If we had not used the **Print** function, there would have been no output; it would be as if we had manually input each line with a **;** to suppress the output.

```
In[132]:= Do[
            "par"[2^p] == (par[2^p] // N),
            {p, 1, 17}
          ]
```

Notice that, in the previous two examples, we have spread the function over 4 lines; this also contributes to the easier readability. Note that the indenting is performed automatically by *Mathematica. Mathematica* quite happily accepts this as a single function. We could have, should we have wished, issued the command as a single-line command, although it would have been harder to read. Perhaps a more straightforward approach might have been the following.

```
In[133]:= Do[{2^p, par[2^p]} // Print, {p, 1, 17}]
```

Out[133]= $\{2, 1.25\}$

Out[134]= $\{4, 1.42361\}$

Out[135]= $\{8, 1.52742\}$

Out[136]= $\{16, 1.58435\}$

Out[137]= $\{32, 1.61417\}$

Out[138]= $\{64, 1.62943\}$

Out[139]= $\{128, 1.63715\}$

Out[140]= $\{256, 1.64104\}$

Out[141]= $\{512, 1.64298\}$

Out[142]= $\{1\,024, 1.64396\}$

Out[143]= $\{2\,048, 1.64445\}$

Out[144]= $\{4\,096, 1.64469\}$

Out[145]= $\{8\,192, 1.64481\}$

Out[146]= $\{16\,384, 1.64487\}$

Out[147]= $\{32\,768, 1.6449\}$

Out[148]= $\{65\,535, 1.64492\}$

Out[149]= $\{131\,072, 1.64493\}$

We have calculated some remarkably large partial sums here. It seems that we can blithely ask *Mathematica* to calculate some truly extraordinary large partial sums, and it will give us an answer almost instantaneously. If we get adventurous, we might try to find the 10^{12}th partial sum

In[150]:= `par[10^12]`

Out[150]= $\dfrac{1}{6} \left(\pi^2 - 6\,\text{Zeta}[2,\ 1\,000\,000\,000\,001] \right)$

Notice that we didn't ask for a numeric approximation this time. Previously, in Section 1.2 when we computed the partial sums symbolically *Mathematica* computed a rational number with large numerator and denominator. The larger the partial sum, the larger the numerator and denominator. However this time we have been given an answer in terms of the zeta (ζ) function, and $\pi^2/6$. Looking a bit closer, we see that the answer amounts to $\pi^2/6 - \text{Zeta}(2, 10^{12} + 1)$.

Looking up the zeta function in the Documentation Center, we find out that the zeta function is defined in terms of these infinite sums like the one we are currently exploring.

$$\zeta(p) := \sum_{k=1}^{\infty} \frac{1}{k^p}$$

and

$$\zeta(p, a) := \sum_{k=1}^{\infty} \frac{1}{(k+a)^p}$$

It seems, then, that we've been calculating partial sums of $\zeta(2)$. Furthermore, inasmuch as we know the value of our infinite sum is $\pi^2/6$, we should expect $\zeta(2, 10^{12} + 1)$ to be very small. It looks a little like we should expect the kth partial sum to be equal to $\pi^2/6 - \zeta(2, k + 1)$. We have not yet actually tried to find a general form for the kth partial sum. This is an egregious oversight which we now correct.

In[151]:= `par[k]`

Out[151]= HarmonicNumber$[k, 2]$

This is not quite what we were expecting. If we look up the **HarmonicNumber** function in the Documentation Center we will see that **HarmonicNumber[n, r]** is the *Mathematica* function to compute $H_n^{(r)}$ where

$$H_n^{(r)} = \sum_{i=1}^{n} \frac{1}{i^r}$$

In other words, these harmonic numbers are precisely the partial sums we have been computing. This isn't really telling us very much, but we have digressed somewhat, so we do not explore this avenue any further.

Getting back to the topic of large partial sums being computed quickly, we might be tempted to conclude that computer technology is just so fast nowadays that such a performance is simply to be expected. Unfortunately, this is not true. For instance, if we try to compute the 1,000,000th partial sum, we find it to be surprisingly slow.

In[152]:= **par[10^6] // N**

Out[152]= $Aborted

The author stopped the computation after approximately thirty to sixty seconds had elapsed. It turned out, upon later investigation, that the computation would only have taken approximately ten minutes. It's important to remember here that the reason we have been performing these computations has been to get a quick feel for the convergence of a system. There may very well be times when we should be happy to let a calculation run for perhaps even weeks or months if the value of the computation is sufficiently important. Computations of π to exceptionally large precision have been run that have taken months to perform. This is *not* such a case, however. It is important to remember our goals in order to ascertain how long a wait is too long.

It is interesting that the 10^{12}th partial sum computed so quickly, but the 10^6th is incredibly slow. For some reason, *Mathematica* suddenly starts giving quick answers of the form $\pi^2/6 - \zeta(2, k+1)$ once k is sufficiently large; $k \geq 10^{10}$ or so. Before this point, however, the kth partial sums take longer and longer to compute as k increases. We computed partial sums up to $k = 2^{17}$, above, mostly because partial sums for $k = 2^{18}$ and above were too slow to be practical.

If we modify our approach only slightly, we are able to calculate the millionth partial sum without the issues we saw earlier. We have already tried first computing **par[10^6]** symbolically and we did not fare well. Instead, we calculate the decimal approximation of each term, and add these decimals together using the **Sum** function.

In[153]:= **Npar[n_] := Sum[a[k] // N, {k, 1, n}]**

In[154]:= **Npar[10^6]**

Out[154]= 1.64493

The computation took approximately three seconds on the author's computer. Note that in order to use the **N** function, we could not use the variable **N** as the argument to our function. Instead we used **n**.

It might be tempting to view the above numerical computation with some contempt. Decimal approximations are just that, approximations. Worse still, the above computation is adding approximations to approximations at every step, and doing it a million times. What's more, we are using a CAS, the whole point of which is to allow the computer to perform exact symbolic calculations. Surely it would seem reasonable, even

preferable, to perform all computations symbolically, and to obtain decimal approximations from these exact mathematical constructs. This was certainly the opinion of one of the authors before the commencement of this book.

The astute reader may try to directly compute a numeric approximation of the `HarmonicNumber` function, remembering that `par[k]` is equal to `HarmonicNumber[k, 2]`. This technique works, but turns out to be about twice to three times as slow as using `Npar`.

In[155]:= `HarmonicNumber[k, 2] // N`

Out[155]= 1.64493

What we are seeing is an example of a case where obtaining decimal approximations of previous symbolic computations is simply not feasible. Sometimes we have to work purely numerically, and the reality is that numeric computations aren't as bad as all that. We should, of course, be quite aware of the fact that numerical approximation can introduce errors in our calculations, and be on the lookout for them, but this should not and does not detract from the usefulness of symbolic computation.

Now that we have computed the 10^6th partial sum, we might try something more ambitious. We try the 10^9th partial sum. Unfortunately, we find that this computation is too slow to be practical. The following computation was, out of the curiosity of the author, left overnight to run using the timing techniques introduced in Section 1.2.7 and took approximately 3,000 seconds (a little under an hour).

In[156]:= `Npar[10^9]`

Out[156]= 1.64493

Note that this long computation time is about 1000 times as long as the 10^6th numeric partial sum. This is perhaps not surprising, if we consider that the 10^9th partial sum has a thousand times as many terms. We will revisit this idea again, as well as introduce the techniques for measuring computations, below in Section 1.2.7.

We leave our exploration there, except to mention that the above example is a particular case of a slightly more general result. This result states that the *p*-series

$$\sum_{k=1}^{\infty} \frac{1}{k^p}$$

where $p \in \mathbb{R}$, converges only when $p > 1$. It is left as an exercise to the reader to explore this further.

1.2.3 Decision Structures

There are times when, as part of a computation we are performing, we need to make some sort of decision in order to proceed. To illustrate this idea, and how we implement decisions in *Mathematica* we look at the following problem.

Let us say we have a natural number n. Recall that if n can be divided by another natural number a evenly—that is, n/a is a natural number—we use the notation $a|n$ and say that a *divides* n or that a is a *divisor* of n. Furthermore, if $a|n$ then $n = ka$ for some $k \in \mathbb{N}$ and so, recalling modular arithmetic, $n \equiv 0 \mod a$.

The problem we now try to solve now with *Mathematica* is to find all the divisors of a number. To begin with, it is helpful to know that *Mathematica* can perform modular

arithmetic using the `Mod` function. Simply put, entering `Mod[a, b]` will calculate the modulus of a (modulo b).

```
In[157]:= Mod[3, 4]
          Mod[9, 7]
          Mod[10, 5]
```

Out[157]= 3

Out[158]= 2

Out[159]= 0

If we recall the infix notation of a function from Section 1.1.6, we may use it to perhaps make the input a little more intuitive. In this case `a ~Mod~ b` will compute the modulus of a (modulo b).

```
In[160]:= 3 ~Mod~ 4
          9 ~Mod~ 7
          10 ~Mod~ 5
```

Out[160]= 3

Out[161]= 2

Out[162]= 0

Returning to the question of finding the divisors of a number, we start with a straightforward approach. Given our number n, whatever it happens to be, we recognize that no number bigger than n can possibly be a divisor of n, so we check every single number a less than n and see if $a|n$. This is just the thing for which a loop would be good. We'll start small with $n = 6$

```
In[163]:= With[{n = 6},
            Do[
              Print[n / a],
              {a, 1, n}
            ]
          ]
```

Out[163]= 6

Out[164]= 3

Out[165]= 2

Out[166]= $\dfrac{3}{2}$

Out[167]= $\dfrac{6}{5}$

Out[168]= 1

The `With` function allows us to temporarily set some variables for the purpose of a computation; in this case we set $n = 6$. Setting variables in this manner means they are not set throughout the worksheet as a whole. This is very much like the temporary variables we discussed earlier in Section 1.2.1, when we were constructing lists of partial sums using the `Table` command. Doing this allows us to phrase our computations in terms of n, without having to set the value of it globally. Note, however, that these variables are constant variables; they cannot be modified at all. They are simply a

convenient name that is substituted with the appropriate value in the expressions found within the **With** function. If we wish to have modifiable temporary variables, we need to use the **Block** function instead, which otherwise works identically to **With**. We will see the use of **Block** shortly, but for now we only need n to be a temporary constant.

We can see from the output that the divisors of 6 are $6, 3, 2$, and 1. It would be nice if we could have *Mathematica* only show the divisors, and not the fractions that are clearly not divisors. To do this we have *Mathematica* make a decision using an **If** function.

Now, we need *Mathematica* to recognize which of the calculations are fractions, and which are whole numbers, but unfortunately we currently have no idea how we might do this. One possible answer is to use the modular arithmetic calculations from above, remembering that if $n/a \in \mathbb{N}$ then $n \equiv 0 \mod a$. This is something we already know how to express in *Mathematica*. In order to see if, say, 3 was a divisor of our 6 then we could issue the command

```
In[169]:= If[6 ~Mod~ 3 == 0, 6 / 3]
```

```
Out[169]= 2
```

The above code should be read as "If 6 is equal to 0 modulo 3 then calculate $6/3$," and because 6 is most certainly equivalent to 0 modulo 3, *Mathematica* has correctly gone on to calculate $6/3 = 2$. Note that if, instead of 6, we had used some other number such that was not equivalent to 0 (modulo 3) then *Mathematica* would have performed no calculation at all. Similarly, if we had picked a different number to 3 to whose modulus 6 was not equivalent to 0, *Mathematica* would have performed no calculation at all.

The first argument to the **If** function is a criterion for the expression that is the second argument to be carried out. If the criterion is met, the expression is evaluated, and if the criterion is not met, the expression is not evaluated. Note that in some cases (which we look at later), we may add an additional expression as a third argument, which will be carried out in the case that the criterion is not met. For the moment, let's look at an example with a single piece of code that does not execute

```
In[170]:= If[6 ~Mod~ 4 == 0, 6 / 3]
```

Of course, we don't want to have to type all of these in individually, so we now incorporate these decisions into our loop.

```
In[170]:= With[{n = 6},
            Do[
              If[n ~Mod~ a == 0, n / a // Print],
              {a, 1, n}
            ]
          ]
```

```
Out[170]= 6
```

```
Out[171]= 3
```

```
Out[172]= 2
```

```
Out[173]= 1
```

In fact, it is usually quite rare that we use a decision manually when using *Mathematica*, where we can make these decisions for ourselves. It is much more common that a decision would be part of a function or a loop where we do not have the luxury of being certain what values our variables contain.

The astute reader may have noticed that each of our divisors—which were all of the form n/a—were, themselves, also values of a at some point in the loop. To see this a little more clearly, we modify our loop to print both n/a and a on the same line.

```
In[174]:= With[{n = 6},
            Do[
              If[n ~Mod~ a == 0, {a, n / a} // Print],
              {a, 1, n}
            ]
          ]
```

Out[174]= $\{1, 6\}$

Out[175]= $\{2, 3\}$

Out[176]= $\{3, 2\}$

Out[177]= $\{6, 1\}$

We have, essentially, found each divisor twice. We checked every integer less than n (6 in our previous examples) to see if it was a divisor, but we need only check the numbers less than or equal to $\sqrt{n}$ (≈ 2.449 in our previous examples). We can see, by inspection above, that after our second repetition we had already found all the divisors of 6.

This is true in general because for any divisor a of a number, n say, we have a codivisor b such that $a \cdot b = n$. This is simply what it means to be a divisor. Now suppose that $a \leq \sqrt{n}$. It must, necessarily, be the case that $b \geq \sqrt{n}$, because

$$b \leq \sqrt{k} \implies ab < \sqrt{n}\sqrt{n} = n$$

which would contradict a and b being codivisors. Similarly, if $a \geq \sqrt{n}$ then $b \leq \sqrt{n}$. It follows, then, that once we've found all the divisors less than or equal to $\sqrt{n}$ then we have also found all the larger divisors in the codivisors.

We can use this fact to modify our loop and make it a little more efficient. There is a small problem here, however, because the square root of a number is not always a natural number. Luckily for us *Mathematica* iterators have no trouble with their upper bound being unnatural (in the number-theoretic sense), and will simply keep iterating until the iteration variable is larger than the upper bound.

```
In[178]:= With[{n = 6},
            Do[
              If[n ~Mod~ a == 0, {a, n / a} // Print],
              {a, 1, Sqrt[n]}
            ]
          ]
```

Out[178]= $\{1, 6\}$

Out[179]= $\{2, 3\}$

Such an improvement may not seem particularly worthwhile for the case of calculating the divisors of 6. However, for calculating the divisors of a large number, 1,000,000 say, then our modified loop would be performing only 1000 decisions, instead of 1,000,000, which is a more significant difference.

We now make one more refinement to this loop. Instead of outputting the divisors two to a line—which the author finds, frankly, distasteful—we will produce a set containing all the divisors. We do this simply by creating a temporary set which we will begin

empty, and which we will **Union** with a set containing each pair of divisors as we find them. Since we will need to modify this temporary set, we must needs use the **Block** function instead of the **With** function we have previously been using.

```
In[180]:= Block[{n = 6, divisors = {}},
          Do[
            If[n ~Mod~ a == 0, divisors = divisors ~Union~ {a, n / a}],
            {a, 1, Sqrt[n]}
          ];
          divisors
        ]
```

Out[180]= $\{1, 2, 3, 6\}$

Note that the second argument to the **Block** function consisted of a compound expression. We have already seen compound expressions on a single line of output, when we used the **;** operator to suppress output. We now see that compound expressions also allow us to use multiple commands as a single argument inside a function like **Block** or **With**. Inasmuch as only the final individual expression may omit the semicolon, the final individual sub-expression may produce output. In the above case, this expression was the expression **divisors** which has the effect of outputting the set of divisors that was computed by the **Do** function.

We now calculate the divisors of 999 (so chosen because the number of its divisors is manageable[3]), which requires only 31 iterations of our loop. Note here that our previous use of **With** and **Block** pay off, as we may copy and paste the previous loop, and need only change the single declaration of **n = 6** with our new number.

```
In[181]:= Block[{n = 999, divisors = {}},
          Do[
            If[n ~Mod~ a == 0, divisors = divisors ~Union~ {a, n / a}],
            {a, 1, Sqrt[n]}
          ];
          divisors
        ]
```

Out[181]= $\{1, 3, 9, 27, 37, 111, 333, 999\}$

It was stated above that the **Mod** function was one way to have *Mathematica* determine whether a number was natural. Another such way is to use the **Cases** function, along with pattern matching—the same pattern matching we use for arguments when defining functions. The **Cases** function will take a list, and will compute a new list containing all the elements of the old list that match some pattern. Often this pattern will describe the form of an expression, such as expressions of the form x^a, however we may also construct patterns that have criteria such as those we have used with the **If** function, above. As such, **Cases** with patterns present another form of decision making, and can be very convenient.

To use this method for our divisor computations, we first need to construct a list. Fortunately, the properties that make our initial simple approach (dividing n by every natural number smaller than or equal to it) perfect for a loop, also make it perfect for the **Table** function. We start, therefore, by computing the set $\{6/a \mid 1 \le a \le 6\}$ as a *Mathematica* list, and from that we pick only the integers.

---

[3] By "manageable", the author means "easily fits on a single line of output when typeset in this book"

```
In[182]:= With[{n = 6},
            Table[n / a, {a, 1, n}]
          ]
          Cases[%, _Integer]
```

Out[182]= $\left\{6, 3, 2, \dfrac{3}{2}, \dfrac{6}{5}, 1\right\}$

Out[183]= $\{6, 3, 2, 1\}$

Take note of the pattern in this case; it is **_Integer**. Just as when we define a function the blank means "any expression", in this case the blank followed by the key word **Integer** means "any integer." The **Cases** function computed for us a new list consisting of all the elements of the previous list which which matched the pattern; that is the elements that were integers. Inasmuch as we were only dealing with positive integers to begin with, this had the effect of giving us only natural numbers. If we wish, in general, to make sure we only pick positive integers, we should use the following instead.

```
In[184]:= Block[{n = 6, div},
            div = Table[n / a, {a, 1, n}];
            Cases[div, m_Integer /; m > 0]
          ]
```

Out[184]= $\{6, 3, 2, 1\}$

In this case the pattern **m_Integer /; m > 0** should be read as "any integer, which we shall name m, with the condition that $m > 0$." The **/;** is the operator for applying conditions to patterns. Patterns are a very flexible and useful mechanism within *Mathematica*. Unfortunately they are also very complicated. We shall use them from time to time within this book, and explain their usage as best we can, however the reader is actively encouraged to learn more through *Mathematica*'s Documentation Center. One should start with the entry named "Introduction to Patterns."

Additionally, note in the previous example that we used a single **Block** function, instead of a **With** function followed by the **Cases** externally. This was done to make the input a single expression. Previously, our command was two expressions, and while we could have made it a compound expression with the use of an **;** operator, the **%** operator does unpredictable things when used as part of a compound expression. The **Block** function allows us to avoid the use of the **%** operator in this case. Further note that we did not need to assign an initial value to the temporary **div** variable, we simply declared it and that was sufficient. We can get away with this because we assign it with the **Table** function as part of the expression inside of the **Block**.

The astute reader might hopefully be seeing how they might modify and use these ideas when creating functions using the methods we saw in Section 1.2.1. We shall look at this in detail in Section 1.2.4. Now, however, for the sakes of completeness we will re-compute the divisors of 999 using this alternate technique. It is left as an exercise to the reader to modify these list methods so as to sort the list from smallest to largest divisor (just as the output form the **Do** function produced).

```
In[185]:= Block[{n = 999, div},
            div = Table[n / a, {a, 1, n}];
            Cases[div, m_Integer /; m > 0]
          ]
```

Out[185]= $\{999, 333, 111, 37, 27, 9, 3, 1\}$

It might not surprise the reader to discover that *Mathematica* in fact has a built-in function that will calculate the divisors of a number. This function is the `Divisors` function.

In[186]:= `Divisors[6]`
          `Divisors[999]`

Out[186]= $\{1, 2, 3, 6\}$

Out[187]= $\{1, 3, 9, 27, 37, 111, 333, 999\}$

The reader may now find cause to wonder why we went through the above rigmarole of loops, decisions, lists and patterns to compute the divisors of a number, when we could have just used this `Divisors` function right from the start. This is not an unfair question, and there are two primary reasons for this approach. The immediately obvious answer is we wished to introduce the reader to *Mathematica*'s loop and decision structures, and the divisor calculation seemed a natural example that lent itself nicely to demonstrating these structures. Furthermore we have demonstrated the ability to use the more basic building blocks of *Mathematica* to perform mathematics and solve problems when we don't know a more direct way of having *Mathematica* perform the calculation. By taking this longer route, we perhaps also allow ourselves to learn a little more about the mathematics than we might have if we had just asked for an answer directly.

This is illustrative of an approach that is used repeatedly by the authors in this book. We repeatedly construct mathematics and calculations from first (or, at least, earlier) principles before introducing the *Mathematica* command that would perform the calculation directly. The reader should be sure to understand both the *Mathematica* and the mathematics involved in these constructions, but should feel free to use the "direct" methods once they have been introduced. In fact, several exercises require the use of these direct methods seemingly blindly, and it is expected that knowledge of the underlying concepts will allow the reader to confirm the answers that *Mathematica* provides, even though the reader may not have the tools to produce the answer without the aid of the computer.

We now return to our discussion of divisors and of *Mathematica* decision structures, exploring both a little further. We introduce the notion of *proper divisors* and *perfect numbers*. When considering the divisors of some number, n say, it should be clear that n is always a divisor of itself. The set of proper divisors of n are simply all of its divisors, except for n itself. So the proper divisors of 6 are $\{1, 2, 3\}$. If we add the proper divisors together, we see that $3 + 2 + 1 = 6$, and that the sum of the proper divisors of 6 is 6 itself. This is an example of a *perfect number*, which is any number n whose proper divisors sum to the value of n.

We use *Mathematica* to calculate whether some numbers are perfect, starting with 6. We need to use a decision, but this time we have two options: either the number is perfect, or it is not. We want an answer in either case, so we need to use the optional third argument to the `If` function. First we must find the proper divisors and add them.

In[188]:= `With[{n = 6},`
            `(Divisors[n] // Total) - n`
          `]`

Out[188]= 6

Note that we only wanted to sum the *proper* divisors, but the `Total` command we used will sum every divisor, so we needed to subtract out n to compensate for this unwanted addition. We can clearly see that 6 is perfect (which we already knew).

Despite the fact that we already know the answer, we will have *Mathematica* compute whether 6 is perfect, using the `If` function. We do this to demonstrate the optional third argument. The function with this optional arguments works almost identically to the way we have already seen it operate. If the condition is met, then the expression that is the second argument is evaluated, but if the condition is not met, then the expression that is the third argument is evaluated.

```
In[189]:= With[{n = 6},
            If[
              (Divisors[n] // Total) == 2 * n,
              Print[n is perfect],
              Print[n is imperfect]
            ]
          ]
```

Out[189]= 6 is perfect

Note that, in the interests of readability, we have broken the `If` function over multiple lines, with the criterion taking up a single line, as well as each of the two possible expressions to be evaluated based on that criterion. We have also simplified the expression. Instead of checking whether the sum of the divisors minus the number is equal to the number, we are simply checking whether the sum of the divisors is equal to twice the number.

We have cheated somewhat here with the message we are printing. It looks like we have printed "6 is perfect", and to a certain extend that is true. However in order for this to happen the expression we used is, in fact, a multiplication; it is the product of the number 6, and two mathematica variables `is` and `perfect`. The coloring of the variables should have given us a clue to this effect, nonetheless the end result is that the desired was output. We have also used the regular method of calling the `Print` function, instead of the postfix we have been using previously. Previously, we have had to use `Print` because if we did not, we would see no output, however it was superfluous to the computation, and as such we have used the postfix notation so that its existence did not interfere with the readability of the computation we were performing. In this case, however, the action we are performing is not a computation, but is to print a message to the screen, as such the `Print` is key, and takes centre stage, so to speak, when we read the input.

To be properly illustrative, we should see a case where the `If` function evaluates its third argument. That is, we should see a case where the criterion fails. To this end, we will try $n = 7$. Note that this also illustrates a weakness with the cheat we performed to write a message to the output; the english becomes broken due to *Mathematica* re-ordering the unknowns when it outputs the expression.

```
In[190]:= With[{n = 7},
            If[
              (Divisors[n] // Total) == 2 * n,
              Print[n is perfect],
              Print[n is imperfect]
            ]
          ]
```

Out[190]= 7 imperfect is

Let's see if there are any perfect numbers less than some upper bound N. We'll start with $N = 6$. Currently if we want to know if a number is perfect, we have a whole mess

of calculations to perform taking up seven lines; if we were to start at $n = 1$ and test up until $n = N$ we would end up with forty-two lines of code in the case of $N = 6$ and more for larger N. Such would be more than a little cumbersome even if we were to copy and paste. Now that we have seen compound expressions, it should not be a stretch to expect that the `Do` function ought to be able to loop a compound expression for us, and so it can. We therefore use a loop. Also, in light of the earlier observations with regards to printing strings, we shall compute a list containing all the perfect numbers less than N. Note that we needed only the first form of the `If` function—the one with only a single expression to be evaluated—for this computation.

```
In[191]:= Block[{N = 6, P = {}},
            Do[
              If[(Divisors[n] // Total) == 2 * n, P = P ~Union~ {n}],
              {n, 1, N}
            ];
            P
          ]
```

Out[191]= $\{6\}$

It seems, then, that 6 is the first perfect number. It turns out that there are precious few perfect numbers. We compute, below, the set of all perfect numbers less than or equal to 10,000, and see that there are only four.

```
In[192]:= Block[{N = 10 000, P = {}},
            Do[
              If[(Divisors[n] // Total) == 2 * n, P = P ~Union~ {n}],
              {n, 1, N}
            ];
            P
          ]
```

Out[192]= $\{6, 28, 496, 8128\}$

We may perform these same computations of perfect numbers less than N using lists and `Cases` if we so wish. The commands to do so are more succinct, but perhaps harder to understand at a quick glance. We use the `Range` function to first compute the list of integers from 1 to N, and then use `Cases` to choose from that list the elements which match our criterion.

```
In[193]:= With[{N = 10 000},
            Cases[Range[N], n_Integer /; (Divisors[n] // Total) == 2 * n]
          ]
```

Out[193]= $\{6, 28, 496, 8128\}$

Finally, we apply these same concepts directly to a new problem. Suppose we have a natural number, n, that is not perfect. Let m be the sum of the proper divisors of n. It is possible (but not necessarily likely) that n also happens to be equal to the sum of the proper divisors of m. If this happens, then n, m are called a pair of *amicable* numbers.

Our problem is to find all the amicable pairs where at least one of the pair is less than 10,000.

Applying some thought to the problem, we should realize that for any n there can only be one possible candidate for its amicable partner m, that candidate being the sum of the proper divisors of n. If n is a perfect number, then the candidate for its

amicable partner is itself, but remember we stipulated initially that n was not perfect, so we must make sure to somehow exclude perfect numbers in our computation.

Our approach, then, is clear. Given n, we calculate the candidate, m, directly, and then sum of the divisors of m. We then check to see if the sum of m's divisors is equal to n, and we also check that n is not perfect. The numbers n, m form an amicable pair so long as both of these checks are true. That is, n, m (with m computed as described above) is an amicable pair if and only if the sum of m's divisors is equal to n and n is not perfect. This leaves us with two criteria, which we are able to express as a single criterion for the **If** function by using the **&&** logical 'and' operator. We implement this approach using a loop similar to that used above.

```
In[194]:= Block[{N = 10 000, m},
            Do[
              m = (Divisors[n] // Total) - n
              If[
                m != n && (Divisors[m] // Total) - m == n,
                {n, m} // Print
              ],
              {n, 1, N}
            ];
          ]
```

Out[194]= $\{220, 284\}$

Out[195]= $\{284, 220\}$

Out[196]= $\{1184, 1210\}$

Out[197]= $\{1210, 1184\}$

Out[198]= $\{2620, 2924\}$

Out[199]= $\{2924, 2620\}$

Out[200]= $\{5020, 5564\}$

Out[201]= $\{5564, 5020\}$

Out[202]= $\{6232, 6368\}$

Out[203]= $\{6368, 6232\}$

To compute this with lists and cases, we first compute the list containing all candidate pairs of n, m where m is the sum of the proper divisors of m, and then apply a pattern to the **Cases** function which matches a list of 2 elements, with the elements named n and m, and choose those pairs which match our criteria, above. Due to the complexity of the **Table** function, we need to use a temporary variable L to hold the set.

```
In[204]:= Block[{N = 10 000, L},
            L = Table[{n, (Divisors[n] // Total) - n}, {n, 1, N}];
            Cases[L, {n_ , m_} /; m != n && (Divisors[m] // Total) - m == n]
          ]
```

Out[204]= $\{ \{220, 284\}, \{284, 220\}, \{1184, 1210\}, \{1210, 1184\}, \{2620, 2924\},$
          $\{2924, 2620\}, \{5020, 5564\}, \{5564, 5020\}, \{6232, 6368\}, \{6368, 6232\} \}$

Notice that, with both methods, we have found each pair twice, once starting with the smaller of the two, and once starting with the larger. We can see then that there are only five amicable pairs less than 10,000. It is left as an exercise for the reader to modify the loop so that it only prints each pair once.

1.2.4 Functions Revisited and Pattern Matching

When we created our own functions earlier, they computed only simple, single expressions. We have seen, in the previous sub-section, some computations requiring multiple lines of input, and relatively completed combinations of expressions. Having many of these as functions would be convenient. Fortunately, in order to facilitate these computations, we used special functions that enclosed them, causing the entire several lines of input to be a single expression; specifically a function call. Examples of these were `Do`, `If`, `With` and `Block`. Using these, we may create more complicated functions than we have previously. We do so now.

To start with we will create a function to compute whether or not a number is positive. We'll call this function `PerfectQ`, adopting the naming style of *Mathematica* inbuilt functions for querying properties, such as `IntegerQ` for determining whether or not its argument is an integer. These functions return either `True` or `False`, and so shall ours. Our first attempt will be to use a pattern, mimicking our pattern from the previous sub-section

```
In[205]:= PerfectQ[n_Integer /; n > 0] := If[
              Divisors[n] // Total == 2 n,
              True,
              False
           ]
```

```
In[206]:= PerfectQ[6]
          PerfectQ[10]
          PerfectQ[27]
```

Out[206]= True

Out[207]= False

Out[208]= True

Note that we have used a pattern to describe the parameters that the function may take. The pattern used is precisely the same pattern we demonstrated in Section 1.2.3; any integer which is greater than 0. If we attempt to call the function with any other argument, *Mathematica* acts as if the function has not been defined.

```
In[209]:= PerfectQ[-1]
          PerfectQ[0.5]
          PerfectQ[x]
          PerfectQ[{6, 10, 28}]
```

Out[209]= PerfectQ[−1]

Out[210]= PerfectQ[0.5]

Out[211]= PerfectQ[x]

Out[212]= PerfectQ[{6, 10, 28}]

Note that our function didn't neatly apply itself to the elements of a list, like our created functions did in Section 1.2.1. This is because a list does not match the argument pattern we gave our function. If we want this new function to apply to the elements of a list we need to use either the `Map` function, or the map operator `/@`.

```
In[213]:= Map[PerfectQ, {6, 10, 28}]
          PerfectQ /@ {6, 10, 28}
```

Out[213]= {True, False, True}

Out[214]= {True, False, True}

Unfortunately, the pattern in our **PerfectQ** function causes another small problem. Our function is supposed to report whether its argument is a perfect number, but in many cases it does nothing at all. Any argument which is not a positive integer cannot possibly be a perfect number, and so our function should return False in these cases. Observe that the **IntegerQ** function behaves precisely like this, although it also doesn't automatically apply itself to the contents of lists.

```
In[215]:=  IntegerQ[-1]
           IntegerQ[0.5]
           IntegerQ[x]
           IntegerQ[{6, 10, 28}]
```

Out[215]= True

Out[216]= False

Out[217]= False

Out[218]= False

We therefore re-write our **PerfectQ** function to be a little more flexible. This means losing the pattern, although we shall make use of it again when it is more appropriate to the desired actions of a function. Instead, we will use the **IntegerQ** function as part of our computation, as this should ensure that our function behaves similarly.

```
In[219]:=  PerfectQ[n_] := If[
             IntegerQ[n] && n > 0 && (Divisors[n] // Total) == 2 n,
             True,
             False
           ]
```

```
In[220]:=  PerfectQ[6]
           PerfectQ[10]
           PerfectQ[27]
           IntegerQ[-1]
           IntegerQ[0.5]
           IntegerQ[x]
           IntegerQ[{6, 10, 28}]
```

Out[220]= True

Out[221]= False

Out[222]= True

Out[223]= False

Out[224]= False

Out[225]= False

Out[226]= False

One advantage of having this function as a function is that it greatly simplifies our code to compute perfect numbers less than N. We may compute a list of perfect numbers in a single line. We may even safely omit the **With** or **Block** functions here, because we only ever have the upper bound in a single location, and so no extra convenience is given by specifying temporary variables.

```
In[227]:= Cases[Range[10000], n_Integer /; PerfectQ[n]]
```

Out[227]= $\{6, 28, 496, 8128\}$

We may use our **PerfectQ** function as the criterion for the **If** function too, if we like, although such code is far less elegant than the previous single-line list computation.

```
In[228]:= Block[{N = 10 000, L = {}},
            Do[
              If[IntegerQ[n], L = L ~Union~ {n}],
              {n, 1, N}
            ]
          ]
```

Out[228]= $\{6, 28, 496, 8128\}$

We now look at a slightly more complicated example. We shall create a function named **AmicableQ** which will answer True or False to the question of whether a pair of numbers are amicable. We start by creating a function which takes two arguments. If we stop to think about this for a minute before proceeding, we should realize that we have 4 conditions to meet. We need to make sure that both our arguments are integers, and then we need to check that each one is equal to the sum of the divisors of the other. That's going to be a large, and unwieldy, **If** statement. To alleviate this, we will use patterns to make sure the arguments are integers.

```
In[229]:= AmicableQ[n_Integer /; n > 0, m_Integer /; m > 0] := If[
            (Divisors[m] // Total) - m == n && (Divisors[n] // Total) - n == m,
            True,
            False
          ]
```

This function, however, is a query function; it ought to always return True or False. It is, therefore, going to have exactly the same problem we had the last time we used patterns to define a query function.

```
In[230]:= AmicableQ[x]
          AmicableQ[{220, 284}]
```

Out[230]= AmicableQ[x]

Out[231]= AmicableQ[$\{220, 284\}$]

We may avoid this by defining a function with the same name, but a different input pattern.

```
In[232]:= AmicableQ[_] := False
```

```
In[233]:= AmicableQ[220, 284]
          AmicableQ[x]
          AmicableQ[{220, 284}]
```

Out[233]= $True$

Out[234]= $False$

Out[235]= $False$

It is instructive to think of these definitions in terms of rules. We have defined a rule for **AmicableQ** in the case that it has two positive integer arguments, and we have defined a rule for **AmicableQ** in the case that it has a single argument which may be any expression. Note that in the latter case we did not even deign to name the expression, for we knew that the function must return False in that case, irrespective of the argument. We may think of this as the *default* case.

The astute reader will have noticed that our function returned False when given the argument **{220, 284}**, even though that pair of numbers is, indeed amicable. This is because **{220, 284}** is a list, and consequently a single argument to the **AmicableQ** function. The rule for single argument input is to return false. However, if we wish to use our **AmicableQ** function to help us compute amicable numbers in lists, as we did at the end of Section 1.2.3, then we need it to be able to recognize amicable pairs in a list, as well. With patterns, this is easy! We simply create a new rule that matches to a list.

```
In[236]:= AmicableQ[{n_Integer, m_Integer} /; n > 0 && m > 0] :=
            Amicable[n, m]
```

The pattern, above, should be read as "any list of exactly two integers, which we shall call n and m with the condition that both n and m are greater than 0." Note that because a list is a single argument, we have only one pattern in this example, whereas our first **AmicableQ** rule had two patterns—one for each argument. Because of this the conditions on **n** and **m** are really a single logical condition, and must be joined with the **&&** operator.

```
In[236]:= AmicableQ[{220, 284}]
```

Out[236]= True

Patterns, as we see above, allow us to describe the structure of an expression, and even to assign names to arbitrary sub-expressions. In the example above, we were able to describe a list of exactly two elements, and to give those elements individual names, as well as provide extra stipulations for them. By doing this we were able to make use of our previous rule definition to perform the actual computation. It is important to remember that these rules do not, by default, over-ride each other, unless we re-define a rule whose pattern had been previously defined. We currently have 3 rules for our **AmicableQ** function; one for a pair of positive integer arguments, one for a single argument list containing exactly two positive integers as elements, and a default rule.

We close this subsection by using this function to find, again, the amicable pairs where at least one of of the pair is less than some number N. We have already computed, in Section 1.2.3 the case of $N = 10,000$, although we computed each pair twice. What's more, the astute reader may have noticed that our previous computation was in fact a computation of all amicable pairs where one of the pair was less than $N = 10,000$. Here we will fix both problems[4]

We shall construct a function to do this, and name it **findAmicable**. The function shall take as a single argument the upper bound for the amicable pairs. Doing so will allow us to demonstrate how the **Block** function (and, by extension, the **With**) function interacts with functions we create.

Recall that in Section 1.2.3 we required a compound expression to compute the list of amicable numbers we computed. The expression consisted of two sub-expressions; one to compute an initial list of candidate pairs, and a second to choose the correct

---

[4] That only one of the pair was less than N is not a problem per-se, although it is easier to talk about pairs less than N than pairs where one of the pair is less than N, so we shall "correct" it anyway to make our discussion easier.

pairs from the candidates. It might be tempting to try and define our function this compound function directly, but unfortunately this does not work.

```
In[237]:= findAmicable[N_Integer /; N > 0] :=
          L = Table[{n, (Divisors[n] // Total) - n}, {n, 1, N}];
          Cases[L, {n_ , m_} /; m != n && (Divisors[m] // Total) - m == n]
```

```
Out[237]= {}
```

There are two problems with this approach. Our first clue that there's a problem should be the indenting; recall that *Mathematica* automatically performs the indenting for us. The fact that the second line is not indented should be a clue that it is not considered part of the previous command. Another clue is that there is an output of an empty set; it looks very much like *Mathematica* has evaluated the second expression. In fact, this is precisely what's happened. The *Mathematica* language provides no mechanism for grouping expressions within a function definition. The above example is a compound expressions, consisting of a function definition (**findAmicable[***N*_Integer **/;** *N* **> 0] := ... ;**) followed by a call to the **Cases** function. This is not what we wanted at all. We intended the input **L = Table[...];** **Cases[...]** to be a compound expression which was evaluated when the function was called.

That was all problem number one. The second problem is that the *L* variable should only be temporary, yet in the above case it is not. This is the reason for the empty output, as the **Cases** function has tried to extract elements from a variable with no value, and finding no elements which matches the pattern, promptly returns to us an empty list.

The solution to both of these problems is the same; we must use either the **Block** or **With** function. In general, the only mechanism *Mathematica* provides us for computing compound expressions as part of a function is for that compound expression to be an argument to a function. This is because a function is very clearly delimited by its opening and closing brackets ('[' and ']') as well as the commas which separate its arguments. In principle any function will do, but in practice it will mostly be the functions we have already seen (**Block**, **Cases**, **Do**, **If**, etc). In this case, we need a temporary variable, and we need to modify that temporary variable, so we must use a **Block**.

Our function, then, behaves as follows. We use a pattern to make sure that only positive integer arguments are accepted, and because we do not need to always produce an output, we will not bother with a default rule. The upper bound, *N*, is given by the function argument, so it does not need to be stipulated as a temporary variable in the **Block**. Furthermore, we may use it to initialize variables inside of the **Block**, and so we do so for the temporary list *L*. We then use the **Cases** function twice; firstly to choose only the pairs where both elements are smaller than *N* and the first element is smaller than the second, and secondly to choose which of those pairs are amicable. It is possible to compress this down to a single use of **Cases** with the **&&** operator, but we shall not do this here.

```
In[238]:= findAmicable[N_Integer /; N > 0] := Block[
          {L = Table[{n, (Divisors[n] // Total) - n}, {n, 1, N}]},
          L = Cases[L, {n_ , m_} /; n < m < N];
          Cases[L, el_ /; AmicableQ[el]]
          ]
```

```
In[239]:= findAmicable[10 000]
```

```
Out[239]= {{220, 284}, {1184, 1210}, {2620, 2924}, {5020, 5564}, {6232, 6368}}
```

Note that, because of the way we constructed our list, *L*, combined with our careful setting up of rules for the `AmicableQ` function, we did not need to be very fussy with the patterns inside of the `Cases` function. The first pattern matched any list of two elements, with no conditions on those elements, the second pattern accepted any expression. The pattern matching work was really all done by the rules of the `AmicableQ` function; we know that the elements of *L* are all lists of two positive integers, because that's how we constructed it, so we can rely on the appropriate rule in the `AmicableQ` function to take effect. The first pattern could have equally well just matched any expression, but it was more convenient to have the individual elements named, than to use the `[[...]]` operator in the condition.

To close with, we will use this new function to compute all amicable pairs less than 100,000. The following computation takes a second or two on the authors home computer.

```
In[240]:= findAmicable[10 000]
```

Out[240]= { {220, 284}, {1184, 1210}, {2620, 2924}, {5020, 5564}, {6232, 6368},
             {10744, 10856}, {12285, 14595}, {17296, 18416}, {63020, 76084},
             {66928, 66992}, {67095, 71145}, {69615, 87633}, {79750, 88730} }

1.2.5 Nesting

We have looked at loops and decisions in the previous sections. Inside a loop, or a decision, we may have any valid *Mathematica* expression, and even compound expressions. Inasmuch as loops and decisions are created with functions, which are valid expressions in their own right, it should not have greatly surprising when we saw decisions inside of a loops. It should be a great stretch, therefore, to expect that we may have a loop inside a loop, or a decision inside a decision. Doing such a thing is often called *nesting* in languages were loops and decisions are not functions, and we shall also refer to it as such in this book.

To illustrate this idea, let's look at some related notions to divisors and perfect numbers. Observe that if a number *n* is not perfect, then the sum of the proper divisors is either strictly greater, or strictly less than *n* itself. If the sum of the proper divisors is less than *n* we say the number is *deficient*, and if the sum of the proper divisors is greater than *n* then we say that *n* is *abundant*. We now have three mutually exclusive options for any natural number.

This idea lends itself nicely to a nested decision. We must first make a decision to see if the number is perfect, and if it is not then we must make a second decision as to whether it is abundant or deficient. We would implement this in *Mathematica* as follows.

```
In[241]:= classify[n_Integer /; n > 0] := With[
            {N = (Divisors[n] // Total) - n},
            If[N > n,
              abundant,
              If[N < n,
                deficient,
                perfect
              ]
```

```
            ]
          ]
```

In[242]:= `Do[{n, classify[n]} // Print, {n, 6, 12}]`

Out[242]= $\{6, \text{perfect}\}$

Out[243]= $\{7, \text{deficient}\}$

Out[244]= $\{8, \text{deficient}\}$

Out[245]= $\{9, \text{deficient}\}$

Out[246]= $\{10, \text{deficient}\}$

Out[247]= $\{11, \text{deficient}\}$

Out[248]= $\{12, \text{abundant}\}$

It is quite important here to notice the `If` within the `If` above; specifically it is the third argument, meaning that it will only be evaluated if the first criterion is false. As such, our function first checks to see if its argument is abundant, if not, then the function checks to see if the argument is deficient, and if it is neither of those, then it declares the number to be perfect. Note that we needed to use the sum of the proper divisors several times, so we stored the value in a temporary constant.

There is no necessary requirement that the innermost decision be within the third argument to the outermost decision, and furthermore we may nest potentially any number of decisions.

Note that **abundant**, **deficient**, and **perfect** are unassigned variables. We need to be a little careful here in making sure that those variables remain unassigned. We could avoid the problem by putting the variables in inverted commas, like we did in Section 1.2.2, but ultimately using variables as text messages is really a kludge. We would do better to crate decision functions, much like our previous **PerfectQ**. We did not do so because the goal of our earlier code was to demonstrate how we may nest a decision within another decision.

Let us now, on a whim, find all the abundant numbers less than or equal to 100, and collect them into a list. Note that even though the **abundant**, **deficient**, and **perfect** are unassigned, we may still use them in a comparison with the **==** operator.

In[249]:= `Cases[Range[100], n_ /; classify[n] == abundant]`

Out[249]= $\{12, 18, 20, 24, 30, 36, 40, 42, 48, 54, 56, 60, 66, 70, 72, 78, 80, 84, 88, 90, 96, 100\}$

We leave our discussion of the properties of numbers and their divisors there for now. Before we move to nested loops, we show one more example of a nested decision. This time we use a nested decision to decide in which quadrant a point in the real plan lies. For the sake of simplicity, we consider any point that lies on the x-axis to be in the upper half-plane, and any point on the y-axis to be in the right half-plane. This means, in particular, that the point $(0, 0)$ is considered to be in the first quadrant.

We construct a function that takes two arguments: x and y, as well as a function which takes a list of exactly two elements. For our decision, we observe that when $y \geq 0$ we are in either the first $(x \geq 0)$ or second $(x < 0)$ quadrant. Otherwise $y < 0$ and we are in either the third $(x < 0)$ or fourth $(x \geq 0)$ quadrant. Our nested decisions use this logic. We will return unassigned variables with this function, even though it is not the best idea. Remember our goal here is to demonstrate nesting decisions.

In[250]:= `quadrant[x_ /; -Infinity < x < Infinity,`
            `y_ /; -Infinity < y < Infinity] :=`

```
        If[y >= 0,
          If[x >= 0,
            first,
            second
          ],
          If[x < 0,
            third,
            fourth
          ]
        ]
      quadrant[{x_ , y_}] := quadrant[x , y]
```

```
In[251]:= quadrant[1, 1]
        quadrant[1, -2]
        quadrant[-4, -3]
        quadrant[{-7, 12}]
```

Out[251]= first

Out[252]= fourth

Out[253]= third

Out[254]= second

Note that in order to match any real number, we had to use the pattern of x being any expression with the condition that $-\infty < x < \infty$, and similarly for y. It is unfortunate that the pattern `x_Real` will only match decimal numbers (not integers or rationals). There is a function named `NumberQ` which might would have allowed us to use `x_ /; NumberQ[x]`, however that would have also matched complex numbers, so we did not use it.

Frustratingly, and a little surprisingly, there is no function named `RealQ`, nor `RationalQ`. If there were, we may have used the logical or operator (`||`) to construct the pattern `x_ /; IntegerQ[x] || RationalQ[x] || RealQ[x]`. On balance, however, this is ugly and inelegant, and pattern we did use is preferable to this hypothetical monstrosity.

With the question of complex numbers having cropped up, we can easily extend our quadrant function to also compute the quadrant of a complex number. We simply create a new rule that matches a single complex variable, since complex numbers are, by their very nature, two dimensional. Fortunately, complex numbers are easier to match than real numbers. We use the `Re` and `Im` functions to extract the real and imaginary parts respectively.

```
In[255]:= quadrant[z_Complex] := quadrant[Re[z], Im[z]]
```

```
In[256]:= quadrant[2 + 3 I]
```

Out[256]= first

We now move on to nested loops. As the nested decisions above should suggest, a nested loop is simply a loop inside another loop. As with decisions, we may nest any number of loops, but we concern ourselves initially with just a pair of nested loops.

If we think of a double sum $\sum_{i=1}^{N} \sum_{j=1}^{M} f(i, j)$ then

$$\sum_{i=1}^{N}\sum_{j=1}^{M} f(i,j) = \sum_{i=1}^{N}\left(\sum_{j=1}^{M} f(i,j)\right)$$

$$= \sum_{i=1}^{N}\left(f(i,1)+\cdots+f(i,M)\right)$$

$$= f(1,1)+\cdots+f(1,M)+\ \cdots\ +f(N,1)+\cdots+f(N,M)$$

The dummy variable, i in this case, assumes a set of values $1,\ldots,N$ and for each of these another sum is to be computed. For this second sum, the dummy variable j also assumes a set of values $1,\ldots,M$ and the i value stays (temporarily) fixed. The final result is the pattern we see above. A nested loop will behave in a very similar way. In fact, we may calculate a double sum with a nested loop, as we do below.

```
In[257]:= Block[{S = 0},
          Do[
            Do[
              S = S + f[i, j],
              {j, 1, 3}
            ],
            {i, 1, 3}
          ];
          S
        ]
```

Out[257]= $f[1,1]+f[1,2]+f[1,3]+f[2,1]+f[2,2]+f[2,3]+f[3,1]+f[3,2]+f[3,3]$

For each (temporarily) fixed value of i, the entirety of the innermost loop is calculated. We may achieve this same result using only the **Sum** function, but the above nicely demonstrates the behavior of a nested loop.

```
In[258]:= Sum[Sum[f[i, j], {j, 1, 3}], {i, 1, 3}]
```

Out[258]= $f[1,1]+f[1,2]+f[1,3]+f[2,1]+f[2,2]+f[2,3]+f[3,1]+f[3,2]+f[3,3]$

Note, in both cases above, that because our innermost loop (or **Sum**) is the one corresponding to the **j** variable, the parameters for this variable appear before those for the **i** variable. This is perhaps a little counter intuitive, when the **i** variable takes logical precedence. Fortunately both the **Do** and **Sum** variables offer an easier-to-use alternative. If we specify multiple iterators, it is understood we mean them to be nested. When we nest in this manner, we put the variables in the more intuitive logical order.

```
In[259]:= Block[{S = 0},
          Do[ S = S + f[i, j], {i, 1, 3}, {j, 1, 3}]
          S
        ]
```

Out[259]= $f[1,1]+f[1,2]+f[1,3]+f[2,1]+f[2,2]+f[2,3]+f[3,1]+f[3,2]+f[3,3]$

```
In[260]:= Sum[f[i, j], {i, 1, 3}], {j, 1, 3}]
```

Out[260]= $f[1,1]+f[1,2]+f[1,3]+f[2,1]+f[2,2]+f[2,3]+f[3,1]+f[3,2]+f[3,3]$

This technique of multiple iterators is the preferred method of nesting when it is available; it is quicker to type, takes up less space on the screen, is easier to read, and has the variables in the intuitive order. We may use it with other functions as well.

For instance, if we wanted to nest lists using a nested **Table** function, we may use this method. Doing so also nicely demonstrates the ordering of the variables. With our above sums *Mathematica* is helpful and sorts the output for us, so even if we had put the dummy variables in the wrong order, it wouldn't have mattered (and the astute reader may very well be quick to point out that addition is commutative, thus rending the question of order moot). Lists, however, are a different matter.

```
In[261]:= Table[{i, j}, {i, 1, 3}], {j, 1, 3}]
          Table[{i, j}, {j, 1, 3}], {i, 1, 3}]
```

Out[261]= $\{\{\{1,1\},\{1,2\},\{1,3\}\},\{\{2,1\},\{2,2\},\{2,3\}\},\{\{3,1\},\{3,2\},\{3,3\}\}\}$

Out[262]= $\{\{\{1,1\},\{2,1\},\{3,1\}\},\{\{1,2\},\{2,2\},\{3,2\}\},\{\{1,3\},\{2,3\},\{3,3\}\}\}$

Each of these commands produces a list containing three sub-lists. Each of these sub lists contains three pairs of elements. The first output is computed with the **i** variable taking precedence. That is, for each **i**, every possible value of **j** is computed, and the corresponding pairs are a sub-list. WIth the second output, however, the **j** variable takes precedence. That is, for each **j**, every possible value of **i** is computed, and the corresponding pairs are a sub-list.

The difference is perhaps subtle, and is moot in the case of sums, but this example demonstrates that not all nestings are as nicely commutative as a sum. We see that the ordering of the nested variables can make a difference. It is left as an exercise to the reader to experiment with nesting the **Table** function within itself (using the technique prior to the multiple-iterators technique) to see the difference the ordering of the iterators makes.

1.2.6 Recursive Functions

For some calculations, it is mathematically convenient to have a function use itself as part of its own calculation. Such a technique is called *recursion*. As a natural example of this we look at the Fibonacci numbers.

Recall that the Fibonacci numbers are given by the relation $f_n = f_{n-1} + f_{n-2}$. This is an example of a *recurrence relation*, which we look at in more detail in Section 1.3.3. Nonetheless, it should be clear that in order to calculate any Fibonacci number, we need to know the previous two. Each of these two numbers is, itself, a Fibonacci number, they therefore may be calculated in turn by knowing the prior numbers. In order to prevent forever looking backwards we need a starting point, or some known Fibonacci numbers, and so we also stipulate that $f_1 = f_2 = 1$.

We may implement this in *Mathematica* fairly simply with a function that uses decision and recursion. If f_1 or f_2 are asked for (these will be **fib[1]** or **fib[2]** because of the way functions and procedures work in *Mathematica*) then we return the value 1, and otherwise we will use the same procedure again to calculate the previous two Fibonacci numbers. In this way we will eventually work our way back to either f_1 or f_2.

```
In[263]:= fib[n_Integer /; n > 0] := If[
            n == 1 || n == 2,
            1,
            fib[n - 1] + fib[n - 2]
          ]
```

Note that, because the function rule states that it only applies to positive integers, we may instead have used the criterion `n <= 2` and we would have been computing the same result. Doing so would be a little less intuitive, and require a little more thinking when reading the code, whereas the logical or (`||`) operator allows us to describe criteria in much the same way as we describe the Fibonacci sequence on paper or in prose.

Let us now compute the first twenty Fibonacci numbers.

In[264]:= `Array[fib, 20]`

Out[264]= $\{1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765\}$

The `Array` function here acts as a sort of shorthand version of `Table`. The differences are that `Array` takes a function, f say, as its first argument, and a positive integer, n say, as its second argument. It computes the list $\{f(1), f(2), \ldots, f(n)\}$. We could have produced the same output with the command `Table[fib[n], {n, 1, 20}]`, however the `Array` was somewhat more succinct.

To make the function a little more intuitive to read we may, if we wish, eschew the decision completely and simply write only the recursive part of the procedure. If we do this, we also need to tell *Mathematica* directly what $fib[1]$ and $fib[2]$ are supposed to be. This is a lot more in line with the way we handle recurrence relations on paper, and so should be intuitively more familiar.

In[265]:= `fib[1] = fib[2] = 1;`
         `fib[n_Integer /; n > 0] := fib[n - 1] + fib[n - 2]`

In[266]:= `Array[fib, 20]`

Out[266]= $\{1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765\}$

We may think of this style of function definition in terms of the rules we talked about in Section 1.2.4. We have a rule for when the argument is 1, a rule for when the argument is 2 and a default-ish rule for when the argument is a positive integer. The more specific rules take precedence, and so arguments of 1 or 2 use the correct rule, even though they do technically match the pattern of being a positive integer.

Note that because we did not use a delayed assignment for `fib[1]` or `fib[2]` we suppressed the output. If we did not do this *Mathematica* would have produced an output of 1 because of the assignment, but such output makes little sense in the context in which we are working.

As it was with divisors, so it is with the Fibonacci numbers; *Mathematica* contains an inbuilt function for their direct calculation. The function is the `Fibonacci` function.

In[267]:= `Array[Fibonacci, 20]`

Out[267]= $\{1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765\}$

Using this function we do not need to write our own Fibonacci computing functions, no matter how simple they are. The reader, nonetheless, should take pains to understand the concepts in the Fibonacci computations we constructed above, for the Fibonacci numbers are not the only recurrence relation, and *Mathematica* will most certainly not always be so forthcoming with inbuilt functions for our convenience.
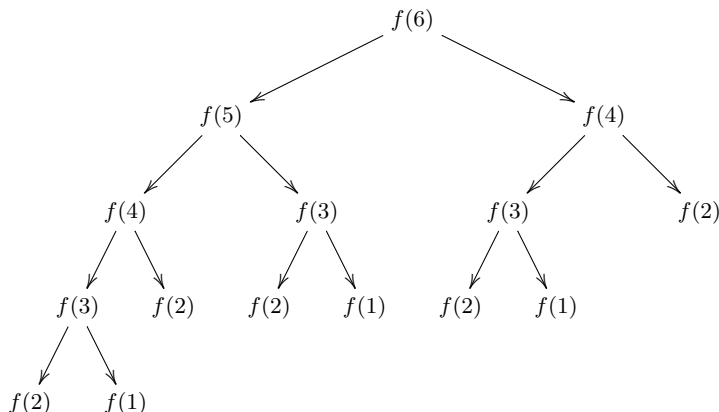
1.2.7 Computation Time

The previous section gives rise to the question of how long a computation will take to perform. It may not be obvious from the computation of the early terms of the Fi-

bonacci sequence, but the above recursive function we wrote in Section 1.2.6 is actually quite slow. The problem stems from the fact that it does not remember previous calculations. For instance, suppose we ask for `fib[5]`, the 5th Fibonacci number. *Mathematica*—acting in accord with our instructions—will first compute `fib[4]` which involves computing `fib[3]` and `fib[2]`, and computing `fib[3]` involves, in turn, computing `fib[2]` and `fib[1]`. *Mathematica* performs each of these computations, including computing `fib[2]` twice. Fortunately, we specified the value of `fib[2]` directly, so the extra computation is quick, just a simple matter of recalling the stored value. This is more easily seen with the tree diagram shown in Figure 1.1.
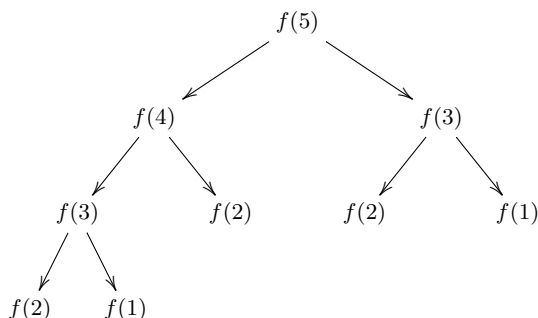
Fig. 1.1 Computation of the 5th Fibonacci number performed by the recursive [fib] function.

In total, from a single request, *Mathematica* has performed nine different computations (although five of these were simply looking up the specified initial values). If now we were to ask for `fib[6]`, then our recursive function would calculate `fib[5]` in its entirety, as well as `fib[4]` also in its entirety for a total of fifteen computations as shown in Figure 1.2. For large Fibonacci numbers, this recursive method will perform a staggering number of computations.

Fig. 1.2 Computation of the 6th Fibonacci number performed by the recursive [fib] function.

This, however, turns out to be a small issue. We may have *Mathematica* remember previous computations of a function, so that if we ask for that computation again, *Mathematica* need only look up the answer from a table of remembered values, rather

than performing the full computation. We do this, quite simply, by having an assignment as part of the function definition.

```
In[268]:= fib[1] = fib[2] = 1;
          fib[n_Integer /; n > 0] := fib[n] = fib[n - 1] + fib[n - 2]

In[269]:= Array[fib, 20]
```

Out[269]= $\{1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765\}$

When we call the function, *Mathematica* will perform an assignment. This works because the function is defined with a delayed assignment, but the function itself computes an immediate assignment. The net effect is that the first time the function is called for any particular argument, it will compute the appropriate Fibonacci number, and store the value in the same manner that we did with the initial conditions. This will greatly improve computation performance, at the expense of some extra memory usage. We show the tree diagram (Figure 1.3) for the computation of `fib[6]` with this technique. For this diagram, the left branch of any node represents the first computation. Once a node is computed for the first time, the value is remembered, and it need not be computed again.
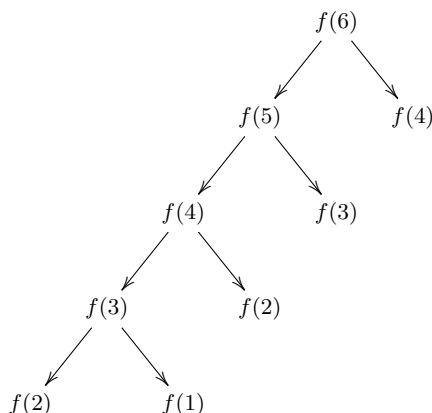
Fig. 1.3 Computations performed by the recursive [fib] function with stored results.

We may measure the time taken for a computation and thereby see the effect of this growing number of computations. *Mathematica* provides a function called `Timing` to measure computation time. The time is measured in seconds, and is a measure of CPU time as opposed to actual elapsed time. The Documentation Center entry for this function explains the specifics. We do not worry ourselves with the technical specifics too much, and just consider this measurement as internally consistent and suitable as a comparison tool.

We start by measuring the time taken to calculate single Fibonacci numbers. We've already implemented the `fib` function above as our faster (or so we claim) procedure, for Fibonacci number computation. To see the difference in speed, we need something to compare against. So we'll re-implement the slower variant, but call it `FIB` so we can test both variants together, and tell them apart.

```
In[270]:= FIB[1] = FIB[2] = 1;
          FIB[n_Integer /; n > 0] := FIB[n - ] + FIB[n - 2]
```

In[271]:= **`FIB[30] // Timing`**

Out[271]= $\{1.36608, 832\,040\}$

The output should be interpreted as a list of time and result in that order. That is it took a bit more than 1.3 seconds to compute $832\,040$ as the output for **`FIB[30]`**. Note that the execution time for a single command will always vary a little bit each time the command is executed. The important thing is that the times are always very close, "in the ballpark" if you like.

The 30th Fibonacci number was chosen because it took about a second to compute, with the slow variant, on one of the authors computers. Anything smaller was a little too quick to be useful for illustrating the concepts illustrated in this section. The reader, when trying to replicate the above, might very well find that even this number is a little too quick if they are using a computer that is faster than the author's, which is likely. Some trial and error may be needed to find a Fibonacci number that takes, approximately, a second to calculate.

Let's see how long it takes for the 31st–35th Fibonacci numbers to compute (individually).

In[272]:= **`Table[FIB[k], {k, 31, 35}]`**

Out[272]= $\{\{2.22808, 1346269\}, \{3.57574, 2178309\},$
$\{5.85781, 3524578\}, \{9.56354, 5702887\}, \{15.3638, 9227465\}\}$

That's a bit of a mess to read. We extract out only the timing information using the **`Cases`** function with a pattern that also includes a transformation rule using the **`->`** operator.

In[273]:= **`Cases[%, {time_ , val_} -> time]`**

Out[273]= $\{2.22808, 3.57574, 5.85781, 9.56354, 15.3638\}$

The pattern here should be read as any list of exactly two elements, which may each be any expression and which we will name "time" and "val" respectively, and transform this list into the expression "time." In this case then the **`Cases`** function chooses the elements that match the pattern, and constructs the list containing the transformation of these elements.

We can see now that computing **`FIB[31]`** took more than half as long again as **`FIB[30]`**, and **`FIB[32]`** took more than twice the computation time of **`FIB[30]`**. What is striking is that these times also exhibit a Fibonacci-like relationship.

$$time(\texttt{FIB[32]}) \approx time(\texttt{FIB[30]}) + time(\texttt{FIB[31]})$$
$$time(\texttt{FIB[33]}) \approx time(\texttt{FIB[31]}) + time(\texttt{FIB[32]})$$
$$\vdots$$

This is in keeping with, and explained by, our earlier observations. The computation for, say, **`FIB[32]`** would involve calculating both **`FIB[31]`** and **`FIB[30]`** in their entirety. So it stands to reason that the computation times should sum in this Fibonacci-like way. We should expect, then, that **`FIB[36]`** should take something in the vicinity of 24 seconds to compute.

In[274]:= **`FIB[36] // Timing`**

Out[274]= $\{25.4651, 14930352\}$

Let us look at the faster variant now. This variant only ever calculates a previous Fibonacci number once, so if we calculate a Fibonacci number with it, then it will calculate every previous Fibonacci number only once as part of the computation. We should expect, then, that the speed should be roughly linear with the position of the Fibonacci number to be computed. That is, if it takes, say, 1 second to calculate the nth Fibonacci number, then we would expect around 2 seconds to calculate the $(2n)$th Fibonacci number.

To begin with, we make sure to clear and re-define the old `fib` function just to make sure we don't have any previously remembered values that might effect our timings.

```
In[275]:= Clear[fib]
          fib[1] = fib[2] = 1;
          fib[n_Integer /; n > 0] := fib[n] = fib[n - 1] + fib[n - 2]
```

We can now safely perform our timings. In order to only see the timing information, we use the replace all operator (`/.`) to reply a transformation rule to the output of the `Table` function. The pattern is the same as we used above, but this time the replace-all operator modifies the output from the `Table` function. The differences between `/.` and `Cases` is that `Cases` constructs a new list consisting of only those elements in the old list that match the pattern (either the elements themselves, or a transformation of those elements), whereas the replace-all (`/.`) operator works on any expression, and leaves any parts of that expression that do not match the pattern intact. In the case below, our pattern matches every element of the list computed by the `Table` function, and so there is no practical difference between `/.` and `Table` in this case.

```
In[276]:= Table[fib[k], {k, 31, 35}] /. {time_ , val_} -> time
```
$$Out[276]= \{0.000152, 5. \times 10^{-6}, 5. \times 10^{-6}, 6. \times 10^{-6}, 6. \times 10^{-6}\}$$

The function is clearly an improvement, however, the above does not really give us very much more information than that. Let's try something bigger. However, by default *Mathematica* will only perform so many recursive calls from a function. This limit is controlled by the `$RecursionLimit` variable, and is 256 by default. This is set for safety reasons, so we do not wish to change it lightly, but we may change it temporarily using a `Block`, and this we do here. Note the semicolon suppressing the output of the `fib` function. This is the reason for the "Null" as the second element in the timing list. We have done this because the 5000th Fibonacci number is, well, huge and the point of what we're doing here is to see how long it takes to compute; a large number in the output will just get in the way.

```
In[277]:= Block[{$RecursionLimit = Infinity}, fib[5000]; // Timing]
```
$$Out[277]= \{0.023104, \text{Null}\}$$

That is exceptionally quick. Unfortunately, due of a limitation of this style of function, it is impractical to find a value that takes approximately a second to calculate. Nonetheless, we can test our earlier expectation, that f_{2n} should take approximately twice as long as f_n. In this case we expect `fib[10 000]` to take approximately 0.04 seconds.

```
In[278]:= Block[{$RecursionLimit = Infinity}, fib[10 000]; // Timing]
```
$$Out[278]= \{0.025374, \text{Null}\}$$

This is not quite what we expected. If we think a little, however, about the consequences of remembering previous computations we should actually find the above result

is not so surprising. When we calculated **fib[5000]**, above, *Mathematica* saved, and remembered, the values of the first 5000 Fibonacci numbers. So when we then went on to calculate **fib[10 000]**, the first 5000 Fibonacci numbers would not have needed to be calculated again and so only 5000 more computations (**fib[5001]**–**fib[10 000]**) needed to be performed, and so the computation times ought to have been similar, which they were.

We perform two more computations in order to better support this idea. If the above claim is correct, then we should expect that a second calculation of **fib[10 000]** should take almost no time at all, and neither should a calculation of **fib[10 001]**. If the claim is not correct, then **fib[10 000]** and **fib[10 001]** should both take similar amounts of time to calculate as the previous computation (0.025 seconds).

```
In[279]:=  fib[10 000]; // Timing
           fib[10 001]; // Timing
```
Out[279]= $\{8. \times 10^{-6}, \text{Null}\}$

Out[280]= $\{0.000018, \text{Null}\}$

It would be nice at this stage to clear and redefine the **fib** function so as to compute and time **fib[10 000]** to see if it took approximately 5 seconds as we might expect. Unfortunately, it turns out that computing more than around 7000 recursions at once causes *Mathematica* to silently fail on the author's computer[5]. This means that it is tricky to directly time the computation of **fib[10 000]**, and we do not pursue it any further here.

This technique of storing and recalling previous computations of a function is very useful, but does make execution times less precise as we have seen. It is probably most useful to think of the time taken worst case scenario (that no values have been previously calculated and stored) as a baseline, with the idea that computations will often be better than this. In the case we have been dealing with, the Fibonacci number calculator **fib** will, in the very worst case, only need to calculate each previous Fibonacci number once. This is significantly better than the earlier attempts.

We close this section with a slightly surprising result. Later, in Section 1.3.3 we look at the Fibonacci numbers again, and find a formula for calculating them. Some readers may already know of this formula. Such a formula would allow a function to perform only a single computation when computing any Fibonacci number. Contrast this with our best approach so far which involves (in the worse case) n computations when computing the nth Fibonacci number. It would seem reasonable to think that the inbuilt *Mathematica* function for Fibonacci numbers would exploit this. However, when we measure the time taken for each technique to compute the first 100,000 Fibonacci numbers we see something perhaps a little surprising.

```
In[281]:=  Clear[fib]
           fib[1] = fib[2] = 1;
           fib[n_Integer /; n > 0] := fib[n] = fib[n - 1] + fib[n - 2]
```

```
In[282]:=  Do[fib[k], {k, 1, 100 000}] // Timing
           Do[Fibonacci[k], {k, 1, 100 000}] // Timing
```
Out[282]= $\{0.739743, \text{Null}\}$

Out[283]= $\{6.52814, \text{Null}\}$

---

[5] Specifically, *Mathematica* emits a quiet beep, and seems to completely forget the definition of **fib**.

We have been very careful here to clear and redefine our functions so as to make sure our measurements are not skewed by previously performed and remembered computations. We have also been careful to apply the `Timing` function so as to time the execution of the entire loop, and not each individual sub-computation. What we see is that our `fib` function is not only impressively fast, but is also drastically quicker than the inbuilt function. Upon first seeing this, one of the authors was quite surprised, however, further thought has lessened this surprise a little. It is left as an exercise for the reader to explore this. As a starting point, one thing to notice is that the `fib` function is exceptionally well suited to sequential computation of Fibonacci numbers—reducing each subsequent computation to a single addition—but there may be other factors as well. Be careful, however; storing too many previous values at once can cause system slowdown in extreme cases.

In closing this section, we note that there are many other small and subtle complexities to the detailed structures (loops, decisions and function) than we have been able to cover. What has been covered is, however, a very good introduction and should serve—along with the relevant exercises—as an excellent and solid starting point for the readers' own computations. Be sure to examine *Mathematica*'s Documentation Center for more information.

1.3 Enough Code, Already. Show Me Some Math!

We have spent the previous two sections learning *Mathematica* code mostly for its own sake. Even when we have tackled mathematical problems, we have done so to understand or illustrate particular *Mathematica* language concepts. By now we hope to be, at the very least, passably familiar with instructing *Mathematica* to perform calculations.

The point of using *Mathematica*, however, is to allow us to perform and explore mathematics, not the other way around. So, from this section onwards we move the emphasis away from *Mathematica* itself and onto mathematical concepts and problems. Our *Mathematica* skills learned in the previous sections are used to explore the mathematics, and new *Mathematica* concepts, functions, and so on are introduced as they are needed for the problems at hand.

To begin with, we should make sure that any previous assignments are cleared, in order to avoid any unintentional conflicts. This is achieved with the following command.

```
In[284]:= ClearAll["Global`*"]
```

1.3.1 Induction

In general, the work we do in *Mathematica* does not constitute a mathematical proof. *Mathematica* is more a tool for exploring mathematics that will often lead to greater understanding and perhaps the production of a proof by the usual (non-computer-related) means. However, we may use it to perform some basic induction for us.

Recall the formula for the sum of the first n squares is

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$

Suppose we only want to add the first n even squares. It's not hard to manipulate the sum to an expression involving the above sum.

$$\sum_{k=1}^{n}(2k)^2 = \sum_{k=1}^{n} 4k^2 = 4\sum_{k=1}^{n} k^2 = \frac{2n(n+1)(2n+1)}{3}$$

and we can certainly check *Mathematica* to see if it provides the same answer.

In[285]:= `Sum[(2 k)^2, {k, 1, n}]`

Out[285]= $\dfrac{2}{3}\, n\,(1+n)\,(1+2n)$

That is all well and good, but it's not induction. How about we try the first n odd squares:

$$\sum_{k=1}^{n}(2k-1)^2$$

We could follow the same approach as above, and expand the summand into a quadratic, and apply the formulae we already know for $\sum_{k=1}^{n} k^2$, $\sum_{k=1}^{n} k$, and $\sum_{k=1}^{n} C$, respectively, and indeed a good number of first-year students would probably prefer this to induction. We do not do that this time, however. Instead we ask *Mathematica* what it thinks the answer is, and verify it using induction (also within *Mathematica*).

To begin with, we set up a function to more easily reuse the calculations. Note that, above, we were careful to use a small **n**, so as not to cause confusion with the function **N**. We have previously gotten away with using capital 'N' because we have been able to re-define its meaning thanks to **Block** and **With** functions. We did not have this luxury, above, although the computation would still have worked OK, it was a better idea to avoid the whole problem in the absence of an explicit redefine. We continue with this choice in the function, below, even though the pattern **N_** is, practically speaking, a re-definition.

In[286]:= `f[n_Integer /; n > 0] := Sum[(2 k - 1)^2, {k, 1, n}]`

Now we see what *Mathematica* thinks the function should look like for arbitrary n.

In[286]:= `expr = f[n]`

Out[286]= $f[n]$

We have been a little too smart for our own good, here. Our function has a pattern to only match positive integers for our variable **n**. This seems reasonable enough, as we are computing a finite sum, beginning with $k = 1$. Unfortunately, this rule means we cannot use our function to do any symbolic computations, which defeats the purpose of what we are doing here. We need to re-define our function, and to be less strict on the types of input. In fact, this time we will not apply any specific pattern at all.

In[287]:= `Clear[f]`
`        f[n_] := Sum[(2 k - 1)^2, {k, 1, n}]`

In[287]:= `expr = f[n]`

Out[287]= $\dfrac{1}{3}\left(-n + 4n^3\right)$

So now we have a candidate for a formula, which we have assigned the result to the variable **expr** for future reference. We can be pretty confident that it is correct because

Mathematica provided it, but it never hurts to check, especially since we don't know yet exactly how *Mathematica*'s **Sum** function handles an indefinite sum like that. This we now do. First we begin with a basis case.

In[288]:= **f[1] == expr /. n -> 1**

Out[288]= True

Note here the use of the **/.** operator. We used this operator previously with the **Cases** function, but we see here that it works perfectly well with other expressions. In this case, we used it to substitute the value 1 for the expression **n** in the expression we stored in the variable **expr**. Note that there is no pattern matching going on here, we're simply performing a replacement.

We're told that the equality is true, but we're not really any the wiser as to why. It is best to see the statement written out in its entirety before asking for an answer of true or false, but doing so is often tricky with *Mathematica*. We circumvent this here by computing a list consisting of the left hand side, and the right hand side of the equality, but note that this particular example is easy enough that we could just verify the equality by hand.

In[289]:= **{f[1], expr /. n -> 1}**

Out[289]= $\{1, 1\}$

It is clear then from the *Mathematica* output that the formula is correct for the basis case of $N = 1$.

Now we may complete the induction. Assuming that

$$\sum_{k=1}^{n} (2k-1)^2 = \frac{1}{3} \left(4n^3 - n \right)$$

we want to show that

$$\sum_{k=1}^{n+1} (2k-1)^2 = \frac{1}{3} \left(4 \cdot (n+1)^3 - (n+1) \right)$$

which we do by showing that

$$\left(\sum_{k=1}^{n} (2k-1)^2 \right) + (2 \cdot (n+1) - 1)^2 - \frac{1}{3} \left(4 \cdot (n+1)^3 - (n+1) \right) = 0$$

In[290]:= **f[n] + (2 * (n + 1) - 1)^2 - (expr /. n -> n + 1)**

Out[290]= $\frac{1}{3} \left(-n + 4n^3 \right) + (-1 + 2(1+n))^2 + \frac{1}{3} \left(1 + n - 4(1+n)^3 \right)$

In[291]:= **Simplify[%]**

Out[291]= 0

And we're done. We may not be sure how *Mathematica* handles an indefinite sum, but we can be extremely confident with its ability to do basic algebra. If we want to completely remove the question of the behavior of *Mathematica*'s **Sum** function out of the equation—as it is technically in question here—we can do the same thing with **expr** alone.

In[292]:= **expr + (2 * (n + 1) - 1)^2 == (expr /. n -> n + 1)**

Out[292]= $\frac{1}{3}\left(-n+4n^3\right)+(-1+2\left(1+n\right))^2 == \frac{1}{3}\left(-1-n+4\left(1+n\right)^3\right)$

In[293]:= `Simplify[%]`

Out[293]= True

Note that in this case we computed an equality, instead of a subtraction. The end result is that our simplify returns True, instead of returning 0. Either technique is fine, and it is often useful to know both. If *Mathematica* cannot provide a true or false answer, it might be able to perform the arithmetic. Broadly speaking, the author (due, mostly, to experiences with other languages) tends to trust *Mathematica*'s ability to perform subtraction better than it's ability to evaluate the truth of statements. As a counterpoint, however, two expressions separated by an `==` might well be easier to read and understand.

Getting back to the induction, we ought to advise caution here. It might be very tempting to attempt to do our induction using something like the following.

In[294]:= `f[N + 1] == expr /. n -> n + 1`

Out[294]= $\frac{1}{3}\left(-n+4n^3\right)+(-1+2\left(1+n\right))^2 == \frac{1}{3}\left(-1-n+4\left(1+n\right)^3\right)$

In[295]:= `% /. (lhs_ == rhs_) -> (lhs - rhs) // Simplify`

Out[295]= 0

Note the use of pattern matching and the `/.` operator to turn the equality into a subtraction. Note also the that the `Simplify` command is applied to the entire result, after the substitution is performed.

Although the above may be tempting, it is *not* induction because we have not used the assumption that $f(n) = cand$ in order to show that $f(n + 1)$ has the required form. All we have done in this case is verify that *Mathematica*'s `Sum` command produces consistent output when given arguments of n and $n+1$. We have not proved the relation using induction.

We do one more induction with *Mathematica*. This time we verify a well known one, and because we won't be explaining every step of the way, the process is much shorter. We verify the formula

$$\sum_{k=1}^{n} k^3 = \frac{n^2(n+1)^2}{4}$$

We can verify in our heads that the relation is true when $N = 1$ so we may eschew the basis step in *Mathematica*, leaving us just the inductive step itself. We will construct an equality statement, and trust *Mathematica* to give us a correct true or false answer.

In[296]:= `With[{expr = n^2 * (n + 1)^2 / 4}]`
              `expr + (n + 1)^3 == (expr /. n -> n + 1)`
            `]`
            `Simplify[%]`

Out[296]= $\frac{1}{4}\,n^2\left(1+n\right)^2+(1+n)^3 == \frac{1}{4}\left(1+n\right)^2\left(2+n\right)^2$

Out[297]= True

1.3.2 Continued Fractions

Real numbers may, as we should already be aware, be expressed by decimals that either terminate or continue (countably) infinitely. The latter category may be further partitioned into recurring and nonrecurring infinite decimal representations. Rational numbers may be written as terminating or recurring decimals, and irrational numbers have only infinite nonrecurring decimal representations.

Another way to represent real numbers is with so-called *continued fractions*. A continued fraction is a, potentially infinite, fraction of the form

$$a_0 + \cfrac{b_0}{a_1 + \cfrac{b_1}{a_2 + \cfrac{b_2}{\ddots}}} \quad \text{where } a_i, b_i \in \mathbb{Z}$$

However, for the purposes of this section, we concentrate on simple continued fractions, where the b_i are all 1

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots}}}$$

often abbreviated to just $[a_0; a_1, a_2, \ldots]$.

The procedure for calculating the continued fraction of a number is quite straightforward. Let $x \in \mathbb{R}$. We let $x_0 = x$ and separate the integer part from the fractional part. That is, we take $a_0 = \lfloor x_0 \rfloor$,

$$x = a_0 + (x_0 - a_0) = a_0 + \cfrac{1}{\left(\cfrac{1}{x_0 - a_0} \right)}$$

We now invert the fractional part for $x_1 = (x_0 - a_0)^{-1}$ and set $a_1 = \lfloor x_1 \rfloor$, yielding

$$x = a_0 + \cfrac{1}{a_1 + (x_1 - a_1)} = a_0 + \cfrac{1}{a_1 + \cfrac{1}{\left(\cfrac{1}{(x_1 - a_1)} \right)}}$$

Inverting the fractional part again we end up with $x_2 = (x_1 - a_1)^{-1}$ and repeat the process.

We explore this idea in *Mathematica* with a terminating decimal $x = 1.23456789$. We use the **Floor** function to extract the integer part The reader is encouraged to read the help documentation regarding this function. Note that the **Floor** function corresponds to $\lfloor \cdot \rfloor$ mathematically. That is $\lfloor x \rfloor$ corresponds to the *Mathematica* command **Floor[x]** (or, equivalently, **x // Floor**).

We are careful to make sure we use the rational representation of our number, rather than the decimal representation, in order to allow *Mathematica* to keep the calculations exact. *Mathematica* provides a function, **Rationalize**, to compute rational approximations of numbers, but unfortunately this function fails to produce the exact rational

that is equal to our x. It is simple enough in this case to see that $x = 123456789/10^8$, and to simply input this manually.

In[298]:= **x[0] = 123456789 / 10^8**
a[0] = x[0] // Floor

Out[298]= $\dfrac{123\,456\,789}{100\,000\,000}$

Out[299]= 1

In[300]:= **x[1] = 1 / (x[0] - a[0])**
a[1] = x[1] // Floor

Out[300]= $\dfrac{100\,000\,000}{23\,456\,789}$

Out[301]= 4

In[302]:= **x[2] = 1 / (x[1] - a[1])**
a[2] = x[2] // Floor

Out[302]= $\dfrac{23\,456\,789}{6\,172\,844}$

Out[303]= 3

Note that we are using something that looks very much like functional notation to store these values. We have used **x[0]**, **x[1]**, and **x[2]** in *Mathematica* to refer to x_0, x_1, and x_2, respectively. Similarly we have used **a[0]**, **a[1]**, and **a[2]** in *Mathematica* to refer to a_0, a_1, and a_2, respectively. If we think of these in terms of the function rules we saw above, we have a rule for arguments 0, 1, and 2, but nothing else. It is useful in this case to think of these as a single variable with an index, instead of a function. Practically speaking, either way of thinking about the notation is equivalent.

So we currently have $a_0 = 1$, $a_1 = 4$, and $a_2 = 3$ for our continued fraction. Next we need to calculate $x_3 = (x_2 - 3)^{-1}$ so we can calculate $a_3 = \lfloor x_3 \rfloor$, and so on, and so forth. Eventually, we hope, we will reach some $n \in \mathbb{N}$ where $x_n - a_n = 0$, at which point we must stop. We started off with a terminating decimal, therefore we should expect the continued fraction also to terminate. So instead of tiring our fingers typing the same two commands over and over again, we write a loop to finish the process for us.

In[304]:= **Block[{i = 2},**
** While[x[i] - a[i]] != 0,**
**   i = i + 1;**
**   x[i] = 1 / (x[i - 1] - a[i - 1]);**
**   a[i] = x[i] // Floor;**
**];**
** Table[a[k], {k, 0, i}]**
]

Out[304]= $\{1, 4, 3, 1, 3, 1, 13565, 1, 8, 10\}$

Note that we did not know ahead of time how many times we would need to perform this loop, so we could not specify an iterator for a **Do** loop. Instead we must used a **While** loop, which meant we needed to manually create and manage the temporary variable, i. Note also that we did not define either x, or a within the **Block**, and so when we used these variables, we used (and set) the non-temporary (i.e., global) variables which we were using outside of the **Block** environment. This is explored a little more in Exercise **??**.

We now have the continued fraction representation of 1.23456789 as

$$1 + \cfrac{1}{4 + \cfrac{1}{3 + \cfrac{1}{1 + \cfrac{1}{3 + \cfrac{1}{1 + \cfrac{1}{13565 + \cfrac{1}{1 + \cfrac{1}{8 + \cfrac{1}{10}}}}}}}}}$$

or, more compactly, $[1; 4, 3, 1, 3, 1, 13565, 1, 8, 10]$. For the remainder of this book we only use this more compact notation when referring to continued fractions.

We pause here and talk briefly about the mysterious 13565 which appears in the middle of this continued fraction. It perhaps seems a little incongruous sitting there in the middle of a collection of predominantly single digit numbers. Or, at least, it should look incongruous. Inasmuch as we've been working with rational numbers for the calculations in question, we can be quite confident that it is not a mistake. However, we should be aware that, in general, when we see such unexpected numbers pop up, that we may have a sign of numeric roundoff error (or some other mistake), and should be on our guard. Increasing the digit precision to see if the (apparent) anomaly persists is a good first step. Computing the number from its continued faction is another good step, and we shall do so quickly now.

```
In[305]:= 1 + 1 / (4 + 1 / (3 + 1 / (1 + 1 / (3 + 1 / (1 + 1 / (13565 + 1 / (1 + 1 / (8 + 1 / 10
          )))))))
```

$$\text{Out[305]}= \frac{123\,456\,789}{100\,000\,000}$$

We started the above computation manually, then concluded it with a loop. A tidier approach would be to have the whole thing encased within a **Block**, with all variables being temporary. Such an approach would look like the following:

```
In[306]:= Block[{n = 123456789 / 10^8, x, a, i = 0},
            x[0] = n;
            a[0] = x[0] // Floor;
            While[x[i] - a[i]] != 0,
              i = i + 1;
              x[i] = 1 / (x[i - 1] - a[i - 1]);
              a[i] = x[i] // Floor;
            ];
            Table[a[k], {k, 0, i}]
          ]
```

$$\text{Out[306]}= \{1, 4, 3, 1, 3, 1, 13565, 1, 8, 10\}$$

Note that we called the number n, instead of x; this is because we cannot use the functional notation if x is assigned a value. Note also that because we defined x and a locally to the **Block** this time, the functions within the block use the local versions of the variables, not the global ones, leaving our previously defined x and a's untouched.

Of course, because we performed exactly the same computation, they will, in this case, hold the same values, but this is coincidence. Again, we explore this phenomenon in Exercise **??**.

As the reader might very well have come to expect by now, *Mathematica* has an functions inbuilt for dealing with continued fractions. These are `ContinuedFraction` for the conversion of a real or rational number to a continued fraction, as well as `WithContinuedFraction` for computing a number from its continued fraction representation. These functions report, and accept as arguments, the list notation for continued fractions. The author knows of no simple way to have *Mathematica* output the continued fraction in its extended, rational form.[6]

> In[307]:= `ContinuedFraction[123456789 / 10^8]`

> Out[307]= $\{1, 4, 3, 1, 3, 1, 13565, 1, 8, 10\}$

> In[308]:= `WithContinuedFraction[{1, 4, 3, 1, 3, 1, 13565, 1, 8, 10}]`

> Out[308]= $\dfrac{123\,456\,789}{100\,000\,000}$

Let us explore continued fractions with some irrational numbers now. We start with the golden ratio $\phi = (1 + \sqrt{5})/2$.

> In[309]:= `With[{phi = (1 + Sqrt[5]) / 2}, ContinuedFraction[phi]]`

> Out[309]= $\{1, \{1\}\}$

Well that's interesting, we have not seen this notation before, but if we have a look inside the Documentation Center for the details on the `ContinuedFraction` function, we see that this the sub-list indicates an infinitely recurring pattern.[7] That is, the continued fraction for *phi* is $[1; 1, 1, 1, 1, \ldots]$, or perhaps $\left[1; \overline{1}\right]$ depending on our preference for notation.

As it happens, being an irrational number, the continued fraction representation of ϕ is infinite, just as its decimal representation. However, unlike its decimal representation—which exhibits no particularly discernible pattern—the continued fraction representation does indeed exhibit a clear pattern. The pattern we have seen above continues infinitely for a continued fraction representation that is all 1.

So why is this? Well, first notice that $2 < \sqrt{5} < 3$ because $4 < 5 < 9$, and so $1 < \phi < 2$. This tells us straight away that the integer part of ϕ is 1. Subtracting this yields

$$\phi - 1 = \frac{1 + \sqrt{5}}{2} - \frac{2}{2} = \frac{\sqrt{5} - 1}{2}$$

Inverting this we get

$$\frac{2}{\sqrt{5} - 1} = \frac{2}{\sqrt{5} - 1} \cdot \frac{\sqrt{5} + 1}{\sqrt{5} + 1} = \frac{2(\sqrt{5} + 1)}{4} = \frac{1 + \sqrt{5}}{2} = \phi$$

which is back where we started from. It should be clear from this then that $\phi - 1 = 1/\phi$ and that we can continue the continued fraction process begun above indefinitely providing the continued fraction $\left[1; \overline{1}\right]$.

Finally, let us look at the continued fractions of some other irrationals.

---

[6] That is, no simple way that doesn't involve writing ones own recursive function, and using the `HoldForm` function, which is outside the scope of this book. The interested reader is encouraged to pursue this themselves.

[7] One needs to extend the "More Information" drop-down to find this detail.

In[310]:= **ContinuedFraction[Sqrt[2]]**
ContinuedFraction[Sqrt[5]]
ContinuedFraction[E]

Out[310]= $\{1, \{2\}\}$

Out[311]= $\{2, \{4\}\}$

ContinuedFraction::noterms: e does not have a terminating or periodic continued fraction expansion; specify an explicit number of terms to generate. »

Out[312]= $\text{ContinuedFraction}[e]$

It seems that e doesn't have a nice pattern to its continued fraction representation, and that the **ContinuedFraction** function needs to be told explicitly how many terms to compute in this case. We'll try 20 as a large-ish round number.

In[313]:= **ContinuedFraction[E, 20]**

Out[313]= $\{2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1, 10, 1, 1, 12, 1, 1\}$

We now see that there was a nice pattern after all, but not a recurring one which could be displayed using a sub-list. We should expect the next four terms of the continued fraction to be 14, 1, 1, and 16. This is precisely what we see when we compute the extra terms.

In[314]:= **ContinuedFraction[E, 24]**

Out[314]= $\{2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1, 10, 1, 1, 12, 1, 1, 14, 1, 1, 16\}$

It should hopefully be apparent, then, that continued fractions may give more information regarding the patterns behind a real number than its decimal expansion might. When exploring an unknown number, we should look at both a decimal approximation, as well as its continued fraction. However, not all numbers have a nice continued fraction representation, as we explore in Exercise 23.

1.3.3 Recurrence Relations

We should recall from Section 1.2.6 the Fibonacci numbers, and their formulation as $f(n) = f(n-1) + f(n-2)$ where $n \in \mathbb{N}$ and $f(1) = f(2) = 1$. This is an example of a recurrence relation, a sequence where the value of each element is dependent on one or more previous elements.

The requirement that n be natural is important here, because the recurrence relation describes a *sequence* and not a function, even though we have used functional notation so far. Recurrence relations are often written using the subscript notation more usually associated with a sequence, in which case the Fibonacci numbers are defined as a sequence $\{f_n\}_{n \in \mathbb{N}}$ where $f_n = f_{n-1} + f_{n-2}$ and $f_1 = f_2 = 1$. However, due to the way *Mathematica* handles recurrence relations, we continue to use the functional notation instead of the subscript notation.

The *order* of a recurrence relation is how far back one must look in order to calculate a term. The Fibonacci numbers are therefore of order two because one needs to know the two previous terms in order to calculate any term. A recurrence relation may have other properties, but for the sake of simplicity let us say that a *constant coefficient, linear, homogeneous* recurrence relation of order k is a recurrence relation that has the form

$$a(n) = c_1 \cdot a(n-1) + \cdots + c_k \cdot a(n-k)$$

where the c_i are constants.

To begin we look at some first-order recurrence relations. In fact, we look in full generality at first order linear recurrence relations with constant coefficients. Let $a(n) = c_1 \cdot a(n-1)$. This is the general form of a first-order linear homogeneous recurrence relation. If we wish to know what the 100th term in the sequence was—provided, of course that we know both the first term $a(0)$, and the coefficient (c_1), then we would have to calculate the second term before we could calculate the third term and so on. All in all we would have to perform 99 calculations. In fact, this is exactly what we have had to do when we wrote loops and procedures to calculate the Fibonacci numbers in Sections 1.2.2 and 1.2.4.

What we would ideally like is some formula or function of n that would always calculate the nth term in the sequence. The act of finding such a formula is called *solving* the recurrence relation. In the case of first-order relations, and especially first-order linear homogeneous recurrence relations with constant coefficients, doing so is quite straightforward. Observe that

$$
\begin{aligned}
a(0) &= 1 \cdot a(0) & &= (c_1)^0 \cdot a(0) \\
a(1) &= c_1 \cdot a(0) & &= (c_1)^1 \cdot a(0) \\
a(2) &= c_1 \cdot a(1) = c_1 \cdot (c_1 \cdot a(0)) & &= (c_1)^2 \cdot a(0) \\
a(3) &= c_1 \cdot a(2) = c_1 \cdot \left((c_1)^2 \cdot a(0) \right) &= (c_1)^3 \cdot a(0)
\end{aligned}
$$

and so on. The pattern here is quite clear.

$$a(n) = (c_1)^n \cdot a(0)$$

We may easily now, if we wish, calculate the 100th element of the sequence as $(c_1)^{99} \cdot a(0)$.

A simple example of such a recurrence relation is a bank account that earns, say, 5% compound interest both calculated and paid annually. If we let $a(n)$ be the amount of money at the end of n years, then $a(0)$ is the initial deposit, and $a(n) = 1.05 \cdot a(0)$. If we start with \$5000 then we know, thanks to the analysis performed above, that after 5 years we will have $(1.05)^5 \cdot 5000 = \$6381.41$.

We can explore recurrence relations inside *Mathematica*, of course. *Mathematica* provides the `RSolve` command for solving recurrence relations. Happily, this function is not limited to only linear, homogeneous, constant coefficient recurrence relations. Unfortunately, not all recurrence relations are solvable. Let us see what *Mathematica* says about our earlier analysis.

```
In[315]:= RSolve[{a[n] == c1 * a[n - 1], a[0] == a0}, a[n], n]
```

Out[315]= $\{\{a[n] \to a0 \, c1^n\}\}$

Here we asked *Mathematica* to solve the recurrence relation $a_n = c_1 \cdot a_{n-1}$ and asked it to solve for a_n, but we did so using the mathematica functional notation. The answer we received could be used as a replacement rule for `a[n]`. As we should have hoped, *Mathematica* gave us the answer we already knew. That the answer is a list within a list suggests the possibility that `RSolve` might sometimes provide us with more than one solution.

Let us now, quickly, drop the constant coefficient condition, and see what happens. For maximum generality, we assume that a function, $f(n)$ say, multiplies the previous term in the recurrence.

```
In[316]:= RSolve[{a[n] == f[n] * a[n - 1], a[0] == a0}, a[n], n]
```

$$\text{Out[316]= } \left\{\left\{a[n] \to a0 \prod_{K[1]=1}^{-1+n} f[1 + K[1]]\right\}\right\}$$

We may confirm the result using a very similar analysis to that performed for the constant coefficient case. Note that $K[1]$, from the above computation, is a temporary summation variable. Its naming suggests that if there were more than one then they might be named $K[2]$, $K[3]$, and so on. For the purposes of our analysis, we will just call this variable k.

$$a(1) = f(1)\,a(0) \qquad\qquad = \left(\prod_{k=1}^{1} f(k)\right) a(0) = a(0) \prod_{k=0}^{1-1} f(k+1)$$

$$a(2) = f(2)\,a(1) = f(2)f(1)a(0) \quad = \left(\prod_{k=1}^{2} f(k)\right) a(0) = a(0) \prod_{k=0}^{2-1} f(k+1)$$

$$a(3) = f(3)\,a(2) = f(3)f(2)f(1)a(0) = \left(\prod_{k=1}^{3} f(k)\right) a(0) = a(0) \prod_{k=0}^{3-1} f(k+1)$$

$$\vdots$$

$$a(n) = f(n)f(n-1)\cdots f(1)\,a(0) \quad = \left(\prod_{k=1}^{n} f(k)\right) a(0) = a(0) \prod_{k=0}^{n-1} f(k+1)$$

Let us now look at second-order linear recurrence relations with constant coefficients. Performing an analysis like the ones above that we performed for first-order recurrence relations is of limited to no use with second-order relations. In short, there's too much going on to see a clear pattern. Fortunately there are some nice theorems that allow for easy solution.

Given a second order linear, homogeneous recurrence relation with constant coefficients,

$$a(n) = c_1 \cdot a(n-1) + c_2 \cdot a(n-2)$$

we construct the *characteristic polynomial* $x^2 - c_1 x - c_2$ and find the roots. Note that the recurrence may be re-written as

$$a(n) - c_1 \cdot a(n-1) + c_2 \cdot a(n-2) = 0$$

and so the transformation to the characteristic polynomial should be clearer now. Once we have the roots r_1, r_2 of the characteristic polynomial, then the recurrence relation has a formula depending on whether the roots are distinct. The general form of the solution is

$$a(n) = \begin{cases} A \cdot (r_1)^n + B \cdot (r_2)^n & \text{if } r_1 \neq r_2 \\ A \cdot (r_1)^n + n \cdot B \cdot (r_1)^n & \text{if } r_1 = r_2 \end{cases}$$

where A and B are constants. If we know some initial conditions, then we may substitute them into the general form of $a(n)$ in order to calculate exactly what the constants A and B are. What is interesting here is that the general form will always satisfy the recurrence relation no matter the choice of constants.

Let us explore this with the Fibonacci numbers. The characteristic polynomial we need to find the roots of is is $x^2 - x - 1$. Using the quadratic formula $x = (1/2a) \cdot (-b \pm$

$\sqrt{b^2 - 4ac}$) we have $x = \frac{1}{2}(1 \pm \sqrt{5})$. So $r_1 = \frac{1}{2}(1 + \sqrt{5})$, which is the golden ratio that we looked at in Section 1.3.2, and $r_2 = \frac{1}{2}(1 - \sqrt{5})$. Turning to *Mathematica* now,

In[317]:= `f[n_] := A * ((1 + Sqrt[5]) / 2) ^ n + B * ((1 - Sqrt[5]) / 2) ^ n`

In[318]:= `f[n] == f[n - 1] + f[n - 2]`

Out[318]= $\left(\frac{1}{2}\left(1 + \sqrt{5}\right)\right)^n A + \left(\frac{1}{2}\left(1 - \sqrt{5}\right)\right)^n B ==$

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

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

In[319]:= `Simplify[%]`

Out[319]= True

That's a bit of a mess, but the answer of "true" is promising. It's probably illustrative here to just look at the right hand side of the equality.

In[320]:= `f[n - 1] + f[n - 2]`

Out[320]= $\left(\frac{2}{1 + \sqrt{5}}\right)^{1-n} A + \left(\frac{2}{1 + \sqrt{5}}\right)^{2-n} A$

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

In[321]:= `Expand[%]`

Out[321]= $2^{2-n}\left(1 + \sqrt{5}\right)^{-2+n} A + 2^{1-n}\left(1 + \sqrt{5}\right)^{-1+n} A + 2^{2-n}\left(1 - \sqrt{5}\right)^{-2+n} B$

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

In[322]:= `Simplify[%]`

Out[322]= $2^{-n}\left(\left(1 + \sqrt{5}\right)^n A + \left(1 - \sqrt{5}\right)^n B\right)$

In[323]:= `Expand[%]`

Out[323]= $\left(\frac{1}{2}\left(1 + \sqrt{5}\right)\right)^n A + \left(\frac{1}{2}\left(1 - \sqrt{5}\right)\right)^n B$

We have verified that we get the formula for $f(n)$ when we simplify $f(n-1)+f(n-2)$, and that the choice of constants A and B does not matter.

In order to find the constants for the specific case of the Fibonacci numbers we could plug $f(1) = 1$ and $f(2) = 1$ into our equation above and solve for A and B. Instead of this, however, we just ask *Mathematica* to solve the recursion for us. We have already defined the variable f, above, so we make sure to use a different variable name this time.

In[324]:= `RSolve[{F[n] == F[n - 1] + F[n - 2], F[1] == F[2] == 1}, F[n], n]`

Out[324]= $\{\{F[n] \rightarrow \text{Fibonacci}[n]\}\}$

Well, thankyou Captain Obvious! That wasn't quite the answer we were perhaps hoping for, although it's perfectly correct. It seems we'll have to use the `Solve` function after all.

In[325]:= `Solve[{f[1] == f[2] == 1}]`

Out[325]= $\left\{\left\{A \to \dfrac{-5-\sqrt{5}}{5\left(1+\sqrt{5}\right)}, B \to -\dfrac{1}{\sqrt{5}}\right\}\right\}$

In[326]:= `Simplify[%]`

Out[326]= $\left\{\left\{A \to \dfrac{1}{\sqrt{5}}, B \to -\dfrac{1}{\sqrt{5}}\right\}\right\}$

and so it would seem that $A = (\sqrt{5})^{-1}$ and $B = -(\sqrt{5})^{-1}$.

1.3.4 The Sieve of Eratosthenes

It would be hard to justify this chapter as being about number theory if we didn't mention prime numbers at some stage. Here we look at the problem of listing numbers that are prime, and use a technique known to the ancient Greeks. In particular, it was a man named Eratosthenes who is responsible for this technique.

Suppose we want to find all the prime numbers less than some number, 100 say. First we write the numbers in a square (or rectangle) thusly

$$
\begin{array}{cccccccccc}
\cancel{1} & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 \\
21 & 22 & 23 & 24 & 25 & 26 & 27 & 28 & 29 & 30 \\
31 & 32 & 33 & 34 & 35 & 36 & 37 & 38 & 39 & 40 \\
41 & 42 & 43 & 44 & 45 & 46 & 47 & 48 & 49 & 50 \\
51 & 52 & 53 & 54 & 55 & 56 & 57 & 58 & 59 & 60 \\
61 & 62 & 63 & 64 & 65 & 66 & 67 & 68 & 69 & 70 \\
71 & 72 & 73 & 74 & 75 & 76 & 77 & 78 & 79 & 80 \\
81 & 82 & 83 & 84 & 85 & 86 & 87 & 88 & 89 & 90 \\
91 & 92 & 93 & 94 & 95 & 96 & 97 & 98 & 99 & 100
\end{array}
$$

We know that 1 is not prime, therefore we start by crossing it out. We now find the first uncrossed number, 2 in this case. This number is prime and so no multiple of is can be prime. So we go and cross out all the multiples of this number.

$$
\begin{array}{cccccccccc}
\cancel{1} & 2 & 3 & \cancel{4} & 5 & \cancel{6} & 7 & \cancel{8} & 9 & \cancel{10} \\
11 & \cancel{12} & 13 & \cancel{14} & 15 & \cancel{16} & 17 & \cancel{18} & 19 & \cancel{20} \\
21 & \cancel{22} & 23 & \cancel{24} & 25 & \cancel{26} & 27 & \cancel{28} & 29 & \cancel{30} \\
31 & \cancel{32} & 33 & \cancel{34} & 35 & \cancel{36} & 37 & \cancel{38} & 39 & \cancel{40} \\
41 & \cancel{42} & 43 & \cancel{44} & 45 & \cancel{46} & 47 & \cancel{48} & 49 & \cancel{50} \\
51 & \cancel{52} & 53 & \cancel{54} & 55 & \cancel{56} & 57 & \cancel{58} & 59 & \cancel{60} \\
61 & \cancel{62} & 63 & \cancel{64} & 65 & \cancel{66} & 67 & \cancel{68} & 69 & \cancel{70} \\
71 & \cancel{72} & 73 & \cancel{74} & 75 & \cancel{76} & 77 & \cancel{78} & 79 & \cancel{80} \\
81 & \cancel{82} & 83 & \cancel{84} & 85 & \cancel{86} & 87 & \cancel{88} & 89 & \cancel{90} \\
91 & \cancel{92} & 93 & \cancel{94} & 95 & \cancel{96} & 97 & \cancel{98} & 99 & \cancel{100}
\end{array}
$$

Having done that, we find the next number along (after 2, which was our previous number) which is not crossed out. In this case it's 3. This number must be prime because

it is not a multiple of any prime number less than it, of which there was only one in this case. We now cross out all multiples of 3. Note that some of the multiples of 3 are already crossed out, because they were also multiples of 2.

$$
\begin{array}{cccccccccc}
\not{1} & 2 & 3 & \not{4} & 5 & \not{6} & 7 & \not{8} & \not{9} & \not{10} \\
11 & \not{12} & 13 & \not{14} & \not{15} & \not{16} & 17 & \not{18} & 19 & \not{20} \\
\not{21} & \not{22} & 23 & \not{24} & 25 & \not{26} & \not{27} & \not{28} & 29 & \not{30} \\
31 & \not{32} & \not{33} & \not{34} & 35 & \not{36} & 37 & \not{38} & \not{39} & \not{40} \\
41 & \not{42} & 43 & \not{44} & \not{45} & \not{46} & 47 & \not{48} & 49 & \not{50} \\
\not{51} & \not{52} & 53 & \not{54} & 55 & \not{56} & \not{57} & \not{58} & 59 & \not{60} \\
61 & \not{62} & \not{63} & \not{64} & 65 & \not{66} & 67 & \not{68} & \not{69} & \not{70} \\
71 & \not{72} & 73 & \not{74} & \not{75} & \not{76} & 77 & \not{78} & 79 & \not{80} \\
\not{81} & \not{82} & 83 & \not{84} & 85 & \not{86} & \not{87} & \not{88} & 89 & \not{90} \\
91 & \not{92} & \not{93} & \not{94} & 95 & \not{96} & 97 & \not{98} & \not{99} & \not{100}
\end{array}
$$

The next number that is not crossed out is 5, which must be prime because it is not a multiple of any prime smaller than it. We repeat this process and eventually there will be no new "next" uncrossed number, in which case we will have found all primes less than or equal to 100 (our chosen upper bound for this example). This yields the following final table.

$$
\begin{array}{cccccccccc}
\not{1} & 2 & 3 & \not{4} & 5 & \not{6} & 7 & \not{8} & \not{9} & \not{10} \\
11 & \not{12} & 13 & \not{14} & \not{15} & \not{16} & 17 & \not{18} & 19 & \not{20} \\
\not{21} & \not{22} & 23 & \not{24} & \not{25} & \not{26} & \not{27} & \not{28} & 29 & \not{30} \\
31 & \not{32} & \not{33} & \not{34} & \not{35} & \not{36} & 37 & \not{38} & \not{39} & \not{40} \\
41 & \not{42} & 43 & \not{44} & \not{45} & \not{46} & 47 & \not{48} & \not{49} & \not{50} \\
\not{51} & \not{52} & 53 & \not{54} & \not{55} & \not{56} & \not{57} & \not{58} & 59 & \not{60} \\
61 & \not{62} & \not{63} & \not{64} & \not{65} & \not{66} & 67 & \not{68} & \not{69} & \not{70} \\
71 & \not{72} & 73 & \not{74} & \not{75} & \not{76} & \not{77} & \not{78} & 79 & \not{80} \\
\not{81} & \not{82} & 83 & \not{84} & \not{85} & \not{86} & \not{87} & \not{88} & 89 & \not{90} \\
\not{91} & \not{92} & \not{93} & \not{94} & \not{95} & \not{96} & 97 & \not{98} & \not{99} & \not{100}
\end{array}
$$

We can see then that our list of prime numbers less than 100 is

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

When completing the previous example—and the reader is encouraged to do so by hand—we find that once we have crossed off all the multiples of 7, then all multiples of all later primes are already crossed off. In fact, this is a phenomenon that we always see when performing the Eratosthenes' sieve for any upper bound. The special property of 7 in our particular case, above, is that it is the largest prime less than or equal to $\sqrt{100}$. In general, when using Eratosthenes sieve to find prime numbers less than or equal to some bound, n say, we may always end the process when we've found all primes less than or equal to $\sqrt{n}$.

Recall from Section 1.2.3 that when finding the divisors of a number, n say, we need only to check the numbers less than or equal to $\sqrt{n}$. In a similar vein, when performing the Eratosthenes' sieve to find all primes less than n, when we find a prime, p say, we then proceed to cross off any number for which p is a divisor. We have effectively marked all numbers less than n that have p as a divisor. We only need a single divisor to decide a number is not prime, and it follows from our observations in Section 1.2.3 that if a number has a divisor, then it has a divisor less than its square root. Finally, the simple fact that

$$a \leq b \implies \sqrt{a} \leq \sqrt{b}$$

leads us to the conclusion that once we have crossed off all multiples of all primes less than $\sqrt{n}$ then we must have found a divisor for every number less than n, as long as there was one to find in the first place. Anything not crossed out must, therefore, be prime.

In our particular example, above, once we get past 10, we have crossed off every multiple of every prime less than or equal to $\sqrt{100}$ and so we must have found at least one divisor for every number less than or equal to 100, as long as there was a divisor to find. This is exactly the observation that led us to this line of inquiry.

Now we implement this technique in *Mathematica*. We make a procedure that we will name **eratosthenes** that will return a list of all the primes less than some given number, N, which must be a positive integer.

In order to represent in *Mathematica* the "crossing out" of numbers as we performed in the sieve (above), we are going to perform set minus operations on this list with the **Complement** function. In order to do this we will first create a list P that contains the first N integers. This process is very dependent on the upper bound being a positive integer, there is no real scope here for symbolic computation, so we are particular about the pattern matching for the argument.

```
In[327]:= eratosthenes[N_Integer /; N > 1] := Block[
            {P = Range[N] ~Complement~ {1}},
            Do[
              P = P ~Complement~ Table[m * n, {m, 2, N / n}],
              {n, 2, Sqrt[N]}
            ];
            P
          ]
```

```
In[327]:= eratosthenes[100]
```

Out[327]= $\{\,2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83,$
$89, 97\,\}$

This function is relatively straightforward, if a little dense. We start with the list of the first N positive integers (**Range[N]**), and remove 1 from that list. This is done in the initialization of the list P within the **Block**. We then begin a loop with the variable n beginning at 2 (which we know is the first prime), and continuing on until it is larger than $\sqrt{N}$. For each of these values of n we construct the table of all multiples $m \cdot n$ such that $2 \leq m \leq N/n$. Note that m must begin at 2 in order to avoid removing the number itself from P, and we have chosen the upper bound of N/n because anything larger will give us multiples of n which are larger than N.

The astute reader may have noticed that the **eratosthenes** function doesn't quite follow the Eratosthenes sieve procedure we described at the beginning of the chapter. The difference is that the procedure as described removes only multiples of primes from the set of numbers, whereas the function we wrote removes multiples of every integer—primes and integers between 2 and $\sqrt{N}$, that is.

The function we wrote will still produce correct output, and it is left as an exercise to the reader to verify this. The function we wrote is quite compact, and not too troublesome to read, however it is performing more computations than it needs to, and this might well mean that for large N it is slower than it needs to be. We will refine it to be more in keeping with the sieve procedure as originally described.

```
In[328]:= eratosthenes[N_Integer /; N > 1] := Block[
            {P = Range[N] ~Complement~ {1}, p},
```

```
Do[
  p = P[[i]];
  P = P ~Complement~ Table[m * p, {m, 2, N / p}],
  {i, 1, Sqrt[N]}
];
P
]
```

In[328]:= **eratosthenes[100]**

Out[328]= $\{ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83,$
$89, 97 \}$

In this version, we have renamed the loop variable to be i, and it begins counting from 1 instead of from 2. We have also added a new temporary variable, p, which is set to the ith element of the list P. In particular, this means that p is first set to be the number 2 (i.e., the first element of P). Recall that once we remove all the multiples of 2, then the next uncrossed number must be the next prime (3), therefore once we have removed the multiples of 2 in our function, the 2nd element of the list P must be the next prime, 3. This is true for each prime, p, and so at each iteration of our loop, the ith element of the list will be the ith prime number.

In short, we have modified our function so that it only removes multiples of primes. Unfortunately, we still haven't quite precisely implemented the Eratosthenes sieve. According to our earlier description, we keep crossing out multiples of primes, until we run out of primes smaller than $\sqrt{N}$. Our function as it is currently implemented, however, crosses out multiples of the first $\sqrt{N}$ primes it finds. It should be clear that for any $M \in \mathbb{N}$ there are less than M primes. So it is hopefully clear that we are crossing out more primes than we need to.

The reader may consider this a negligible difference, and ignore it. Indeed, the reader may even consider the previous refinement unnecessary. We are certainly trading elegance and readability for efficiency. Nonetheless, we will correct this last difference here, just to have a "proper" implementation if for no other reason. The interested reader is encouraged to apply the timing techniques introduced in Section 1.2.7 to these three sieve variants to see just how much difference there is. The use of large integer arguments is recommended for this purpose.

In[329]:= **eratosthenes[N_Integer /; N > 1] := Block[**
          {P = Range[N] ~Complement~ {1}, p},
          Do[
            p = P[[i]];
            If[p > Sqrt[N], Break];
            P = P ~Complement~ Table[m * p, {m, p, N / p}],
            {i, 1, Sqrt[N]}
          ];
          P
        ]

In[329]:= **eratosthenes[100]**

Out[329]= $\{ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83,$
$89, 97 \}$

This variant is almost identical to the prior one. The difference is that we have added the line `If[p > Sqrt[N], Break]`. The **Break** function will stop the processing of a loop

(either a **Do**, **While**, or **For** function). In this case, if we find that our prime p is greater than $\sqrt{N}$ then we stop processing the **Do** loop immediately.

We have made an additional refinement, as it happens. Previously, when we constructed our list of multiples of a prime p, we would start at $2p$, and compute as high as $\lfloor (N/p) \cdot p \rfloor$. However, any multiple of p less than p^2 will have already been crossed out in previous iterations; $2p$, $4p$, etc will have been crossed out with the multiples of 2, similarly $3p$, $6p$, etc will have been crossed out with the multiples of 3, and so on and so forth. As such, when constructing these sets of multiples, we need only construct the multiples beginning at p^2, and this is what the most recent variant does.

We now use this latest variant to compute the primes less than 1000.

In[330]:= `eratosthenes[1000]`

Out[330]= $\{\, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83,$
$89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173,$
$179, 181, 191, 193, 197, 199, 2, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269,$
$271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367,$
$373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461,$
$463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557, 563, 569, 571,$
$577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659, 661,$
$673, 677, 683, 691, 701, 709, 719, 727, 733, 739, 743, 751, 757, 761, 769, 773,$
$787, 797, 809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863, 877, 881, 883,$
$887, 907, 911, 919, 929, 937, 941, 947, 953, 967, 971, 977, 983, 991, 997 \,\}$

Generating prime numbers is actually a fairly computationally intensive task. The sieve of Eratosthenes is quite interesting in its own right, and is reasonably efficient for such a simple algorithm. However, for primes larger than 10,000 it starts to get noticeably slow. Of course, as we should have well and truly come to expect by now, *Mathematica* provides inbuilt functions for finding primes, as well as testing primality. The big advantage to the inbuilt functions is that they perform all sorts of tricks to keep the execution times remarkably quick.

These functions are **Prime** which calculates the kth prime number, **NextPrime** which calculates the kth prime number after (or before if k is negative) a given number, and finally **PrimeQ** which gives a true or false answer as to whether a given number is prime.

In[331]:= `Prime[5]`
`NextPrime[5]`
`NextPrime[6]`
`NextPrime[5, 2]`
`NextPrime[5, -1]`
`PrimeQ[5]`
`PrimeQ[4]`

Out[331]= 11

Out[332]= 7

Out[333]= 7

Out[334]= 11

Out[335]= 3

Out[336]= True

Out[337]= False

The reader is encouraged to look these functions up in the Documentation Center.

1.4 Problems and Exercises

1. Enter the following expressions into *Mathematica*

   a. $\dfrac{1}{2}$

   b. e^2

   c. $x^2 + x - 1$

   d. $\dfrac{x^2 + 1}{x^2 - 1}$

   Perform the following calculations in *Mathematica*. Obtain a decimal approximation as well as simplified exact values.

   e. $\sin\left(\dfrac{3\pi}{5}\right)$

   f. $\dfrac{15!}{e^4}$

   g. $\dfrac{12! + 2^{\frac{1}{3}}}{\sin(2)^3}$

   h. $2^{2^{2^{2^2}}}$

2. We now explore *Mathematica*'s digit precision for decimal approximations. Enter the following commands into *Mathematica* and explain how the precision has been modified. Try some entries of your own if you are still unsure, or to test your explanation.

   a. `N[Pi]`

   b. `N[E]`

   c. `N[Pi, 20]`

   d. `N[E, 25]`

   e. `N[Pi, 35]`

   f. `N[E, 30]`

   Obtain numeric approximations of the following expressions, to the indicated digit precision

   g. π^2 to 10 significant figures

   h. $\sin(1)$ to 10 significant figures

   i. e^π to 15 significant figures

   j. $\log(\pi)$ to 22 significant figures

3. This exercise shows two cautionary scenarios.

   a. Some variables in *Mathematica* are protected, which means their values cannot be changed. Enter the following into *Mathematica*.
      i. `Sin = E^2`
      ii. `Log = x^2 + 3 x - 2`
      iii. `Sum = (x + 1) / (2 x - 2)`
      Why might these names be protected?
   b. We look at why caution should be taken when using the `%` operator. Follow the instructions below.
      i. Enter the following *Mathematica* commands. Make sure to keep each entry as a separate input.
         - `(x + y)^3`
         - `Expand[%]`
         - `% /. y -> 1`
      ii. Edit the first command to instead read `(x + y)^3`. Be sure to remember to hit *shift-enter* to perform the new calculation
      iii. Go back to the third calculation and press shift-enter. What happened?

4. We have seen the **Table** function used to create lists with particular patterns. We mentioned, briefly, the existence of the **Array** function. Enter the following commands, and explain how the **Array** function works. Try some entries of your own if you are still unsure, or to test your explanation. Confirm your guess with *Mathematica*'s Documentation Center.

   a. **Array[Exp, 10]**   c. **Array[Sin, 5]**
   b. **Array[Exp, 5, 0]**   d. **Array[Sin, 6,4]**

   We briefly look now at the notion of a pure function. These are functions which may be given as arguments to functions without the need to be defined beforehand. Enter the following commands, and take a guess at the format of a pure function. Verify your guess by looking up pure functions in the Documentation Center.

   e. **Array[1 / # &, 7]**   g. **Array[Subscript["a", #] &, 4, 3]**
   f. **Array[#^2 &, 7, 4]**   h. **Array[# &, 3, 3]**

5. *Mathematica*'s iterator notation (such as we have seen with the **Table** function) is capable of describing indices which increases by more than 1 at each step, or even where the index variable takes on arbitrary values entirely. Enter the following into *Mathematica* and explain what is happening. Note that the index variable can be any valid variable.

   a. **Table[i, {i, 1, 10, 3}]**
   b. **Table[a^2 - a - 1, {a, 1, 10, 2}]**
   c. **Table[2 / n, {n,{2, 3, 5, 7, 11, 12}}]**
   d. **Table[fib^2, {fib,{1, 1, 2, 3, 5, 8}}]**

   Now produce the following sequences using the **Table** function, and the ideas above.

   e. $9, 25, 49, 81$ f. $1, \dfrac{1}{2}, \dfrac{1}{4}, \dfrac{1}{7}, \dfrac{1}{11}, \dfrac{1}{16}$

   Note that inasmuch as the **Array** function does not take an iterator as an argument, these techniques can be used with the **Array** function; see Exercise 4.

6. For lists and sequences—and many other *Mathematica* constructs that use natural indices—indexing may be performed by using a negative index that begins counting from the end, instead of the beginning. To illustrate this, create in *Mathematica* a list **L = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}**, and issue the following commands.

   a. **L[[-1]]**   d. **L[[-7 ;; -5]]**
   b. **L[[-3]]**   e. **L[[-4 ;;]]**
   c. **L[[-4 ;; -2]]**   f. **L[[;; -6]]**

   Which of the following commands do you expect to fail? Enter them into *Mathematica* and see if you were correct. Explain why the failures occurred.

   g. **L[[-1 ;; -3]]**   i. **L[[4 ;; -7]]**
   h. **L[[3 ;; -3]]**   j. **L[[-7 ;; 2]]**

   Hint: Try to change the negative index into the usual positive index.

7. Enter the following commands into *Mathematica*. For the purposes of these commands **L = {x, y, z, x, y}**.

    a. `Length[{1, 2, 3, 4, 5}]`   c. `Length[L]`
    b. `Count[{1, 2, 3, 4, 5}, 3]`   d. `Count[L, x]`

Now change L to be `L = {{1, 2}, {2, 3}, {3, 4}}` and consider the following.

    e. `Length[L]`
    f. `Count[L, 3]`
    g. `Count[L, {1, 2}]`

What does it look like the `Length`, and `Count` functions are doing? Finally, redefine L to be `L = {{1, 2}, {3, 4}, 5, 2, {1, 2}}` and calculate the following. You should be able to verify the answers by eye.

    h. The number of occurrences of the element $[1, 2]$ in the list L.
    i. The number of occurrences of the element 5 in the list L.
    j. The number of occurrences of the element x in the list L.
    k. The number of occurrences of the element $[y, z]$ in the list L.

Can you get the `Count` function to count all occurrences of the number 2 in the list, regardless of how deeply embedded they are in sublists? (The answer is 3).

8. Create a list containing the first 1000 digits of π, and count the number of times each digit occurs.
   Hint: You will need to either find a function to extract digits from a number, or to write one yourself.

9. Calculate the following sums and products.

    a. $\displaystyle\sum_{i=1}^{100} \frac{1}{2^i}$ c. $\displaystyle\sum_{n=0}^{\infty} \frac{1}{n!}$

    b. $\displaystyle\prod_{n=1}^{6}\left(1 + x^{2^n}\right)$ d. $\displaystyle\prod_{k=1}^{\infty} \frac{4k^2}{4k^2 - 1}$

The following should look familiar from first-year calculus.

    e. $\displaystyle\sum_{k=0}^{\infty} \frac{x^{2n}}{(2n)!}$ f. $\displaystyle\sum_{k=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!}$

Convert the following sums to sigma notation, and then use that to implement them using the `Sum` command.

    g. $1 + 8 + 27 + \cdots + n^3$
    h. $1 + \dfrac{1}{2} + \dfrac{1}{4} + \cdots + \dfrac{1}{2^i}$
    i. $\dfrac{1}{1 \cdot 3} + \dfrac{1}{3 \cdot 5} + \dfrac{1}{5 \cdot 7} + \cdots + \dfrac{1}{(2k-1)(2k+1)}$

10. Use *Mathematica*'s `Sum` command to explore the binomial formula. Recall that the binomial theorem states that

$$(x + y)^n = \sum_{k=0}^{n} \binom{n}{k} x^{n-k} y^k, \text{ where } \binom{n}{k} = \frac{n!}{(n-k)!k!}$$

You should also use *Mathematica*'s help to find a built-in function that will calculate the binomial coefficients $\binom{n}{k}$.

a. Create a function, f say, that expands $(x+y)^n$.
b. Create another function, g say, that uses *Mathematica*'s `Sum` command to evaluate the sum in the binomial formula.
c. Choose some values for n and check that $f(n) = g(n)$.
d. Ask *Mathematica* to evaluate $f(N)$. To what does it evaluate?

Perform these tasks twice: once using *Mathematica*'s built-in function for binomial coefficients and once using the formula for the coefficients.

11. We explore the following relationship between infinite sums and infinite products.

$$\log\left(\prod_{k=a}^{\infty} f(k)\right) = \sum_{k=a}^{\infty} \log(f(k))$$

a. Find the value of the product: $\prod_{k=3}^{\infty} \cos\left(\frac{\pi}{k}\right)$
b. Why did this product need to begin at $k = 3$?
c. Find the value of the sum: $\sum_{k=3}^{\infty} \log\left(\cos\left(\frac{\pi}{k}\right)\right)$
d. Verify that the relationship for the above values.
e. Can you justify the relationship in general?

Hint: Recall that $\sum_{k=a}^{\infty} f(k) = \lim_{n\to\infty}\left(\sum_{k=a}^{n} f(k)\right)$, and similarly for products.

12. An arithmetic sequence is a sequence where each term differs from the previous term by a fixed amount. This "fixed amount" is called the *common difference* of the sequence. For example,

$$a = \{1, 2, 3, 4, \ldots\}, \quad b = \{1, 3, 5, 7, \ldots\}, \text{ and } c = \{1, 4, 7, 10, \ldots\}$$

are arithmetic sequences with a common difference of 1, 2, and 3, respectively.

a. Determine a formula for the nth element of an arbitrary arithmetic sequence starting at 1, with common difference d.
b. Write functions that will calculate the first n terms in the arithmetic sequences a, b, and c (above).
   Hint: Use part (a) in combination with a `Table` command.

Arithmetic sequences may have any first element, but those beginning at 1 generate the so-called *polygonal numbers*. The sequence a generates the triangular numbers; the nth triangular number is the sum of the first n terms in the arithmetic progression a (above). That is,

$$T := \left\{\sum_{k=1}^{N} a_k\right\}_{N=1}^{\infty}$$

Similarly the nth square number is the sum of the first n terms in the arithmetic progression b and the nth pentagonal number is the sum of the first n terms in the arithmetic sequence c.
Note: The square numbers are precisely the numbers that are perfect squares.

c. Write functions that will calculate the nth triangular, square, and pentagonal numbers.
d. Repeat (b) and (c) for the hexagonal numbers. (You will need to extrapolate which arithmetic sequence generates the hexagonal numbers).

13. If we wish to apply a function to every item in a list, *Mathematica* provides a
    function named `Map`, with a corresponding map operator, `/@`. Enter the following
    commands, and read the entry on the `Map` function in *Mathematica*'s Documentation
    Center.

    a. `Map[Exp, {1, 2, 3, 4, 5}]`   b. `Sin /@ {Pi / 2, Pi, 3 Pi / 2, 2 Pi}`

    The following use the `Block` so as to avoid permanently defining the function `f`.
    Enter the following.

    c. `Block[{f},`   d. `Block[{f},`
         `f[x_] := x^2;`   `f[x_] := x / 2;`
         `Map[f, {0, 2, 4, 6, 8}]`   `f /@ {0, 1 / 2, 1, 3 / 2, 2}`
       `]`   `]`

    Recall pure functions from Exercise 4. Enter the following.

    e. `With[{L = {2, Pi, E, Log[Pi]}},`   f. `With[{C = Table[k + I], {k, 1, 10}},`
         `Map[1 / # &, L]`   `(Abs[#] &) /@ C`
       `]`   `]`

    Create a list P that contains the first five prime numbers $[2, 3, 5, 7, 11]$. Use `Map` or
    `/@` as you choose, combined with the list P to create the following lists.

    g. $\{3, 4, 6, 8, 12\}$
    h. $\{5, 10, 26, 50, 122\}$
    i. $\{\ln(2\pi), \ln(3\pi), \ln(5\pi), \ln(7\pi), \ln(11\pi)\}$

14. Just as with the `Table` (see Exercise 5) *Mathematica* `Do` loops are capable of using
    the same iterator notation to cause their counter increase by more than 1 at each
    step, or even have the counter take on arbitrary values entirely from a list. Enter
    the following into *Mathematica* and explain what is happening.

    a. `Do[`   c. `Do[`
         `i^2 // Print,`   `2 / n // Print,`
         `{i, 1, 10, 3}`   `{n, {2, 3, 5, 7, 11, 12}}`
       `]`   `]`
    b. `Do[`   d. `Do[`
         `a^2 - a - 1 // Print,`   `fib^2 // Print,`
         `{a, 1, 10, 2}`   `{fib, {1, 1, 2, 3, 5, 8}}`
       `]`   `]`

    Loops in *Mathematica* may also continue indefinitely until some criterion is met.
    This is achieved with the `While` and `For` functions. Examine the documentation
    on the `While` and `For` functions from the Documentation Center, then enter the
    following, and explain what the loop is doing.

    e. `Block[{i = 1},`   f. `Block[{N = 1},`
         `While[i^2 < 1000,`   `While[N^3 < 10 000,`
           `i // Print;`   `{N, N^3, N^3 - N^2} // Print;`
           `i = i + 1;`   `N = N + 2;`
         `]`   `]`
       `]`   `]`

Create loops to calculate the following.

- g. Decimal approximations of e, e^6, e^{11} and so on where the power of e is less than 100.
- h. Decimal approximations of $\frac{\pi}{2}, \frac{\pi}{3}, \frac{\pi}{5}, \frac{\pi}{7}, \frac{\pi}{11}$.
- i. The Fibonacci numbers less than 1200.

Note: Recall from Section 1.3.3 that the nth Fibonacci number can be calculated using the **Fibonacci** function.

15. Write functions to perform the following. Your functions may have two or more input variables with an appropriate pattern to match. For example **f[x_ , y_]** would define a function accepting two arguments which may each be any expression, with the first one named **x** and the second one named **y**.

- a. Add all multiples of 5 and 7 less than an arbitrary number.
- b. Generate a sequence using the Fibonacci equation, but from an arbitrary pair of initial conditions.

Be sure to test your functions with small cases that you can verify by hand.

16. Use nesting to perform the following.

- a. Print out the first 5 rows of Pascal's triangle.
- b. Create a function or procedure that prints out the first N rows of Pascal's triangle

Hint: Try asking *Mathematica* what it knows about Pascal's triangle. You might need to not use punctuation. Also recall that Exercise 10 dealt with the binomial formula, which is related to Pascal's triangle.

17. Implement the numeric partial sums for the functions in Section 1.2.2 using **NSum**, as well as the method already used. Compare execution times between the two methods.

18. Write a function to compute the $n$thFibonacci number using a loop, instead of using recursion. Measure the time it takes this function to compute successive Fibonacci numbers, in a manner like that performed in Section 1.2.7. Find a Fibonacci number that takes approximately thirty seconds to compute.

19. One should be careful when using the **%** operator for creating functions. Enter the following into *Mathematica*.

```
a. Sum[k, {k, 1, n}]        b. Sum[k, {k, 1, n}]
   f[n_] := %                  f[n_] := Evaluate[%]
   "f"[2] == f[2]              "f"[2] == f[2]
```

What appears to have happened here?

20. Recall the definition of an arithmetic sequences and polygonal numbers from Exercise 12. Using the **Sum** function, find a formula for

- a. An arbitrary polygonal number, that is, the sum of the first n terms of an arbitrary arithmetic sequence beginning at 1 with common difference d
- b. The sum of the first n terms of an arbitrary arithmetic sequence beginning at a with common difference d

Use induction to prove these formulae.

21. Perform the manual calculation of the continued fraction of 1.23456789 (the one involving the **Do** loop) from Section 1.3.2 without starting with a rational number. What happens? Why do you think this is happening?

WARNING: Save your worksheet before attempting this. You will probably want to use the "stop sign" icon to cancel computations, when things seem to be going on for too long.

22. Use the **ContinuedFraction** function to change the following continued fractions into rational numbers.

    a. $\{1, 2\}$ c. $\{1, 1, 1, 2\}$
    b. $\{1, 1, 2\}$ d. $\{1, 1, 1, 1, 2\}$

    Do you notice an interesting pattern with these examples? Formulate a conjecture regarding this pattern of continued fractions and the rational numbers they are equal to, and test your conjecture.
    Hint: This behavior is related to the Fibonacci numbers and their relation to the golden ratio.

23. Have *Mathematica* calculate the continued fractions for the following numbers.

    a. $\sqrt[3]{2}$ b. $(\sqrt[5]{2})^3$ c. π

    Can you see any pattern? Be sure to try several computations, computing different amounts of quotients each time.

24. What is the solution to the general first order recurrence relation

$$a(n) = f(n)\, a(n-1)^p$$

    where $p > 1$ is a positive power?
    Test this answer by picking some simple functions $f(n)$ (polynomials are recommended) and powers of p and seeing if the first however-many terms in the sequences agree when you calculate the terms both using recursion, and the formula given by **RSolve**.

25. Find general solutions to the following recurrence relations. Verify the solutions.

    a. $a(n) = 5a(n-1) - 6a(n-2)$
    b. $s(n) = 2s(n-2)$

    Now find the solutions to the following recurrence relations with initial values. Test the solutions both by testing that the formula satisfies the recurrence, and by having *Mathematica* calculate the first 10 or 20 terms of the sequence from both the recurrence and the solution.

    c. $f(n) = f(n-1) + 2f(n-2)$ where $f(0) = 1, f(1) = -2$
    d. $a(n) = a(n-1) \cdot a(n-2)^2$ where $a(0) = 2, a(1) = 3$

26. Find a formula for the number of ways to climb a flight of steps of height n if 1 or 2 steps may be taken at a time.
    Hint: Formulate the problem as a recurrence relation.

27. Use the techniques from Exercise 1.2.7 to measure the time taken for the sieve of Eratosthenes from Section 1.3.4

    a. Measure the time taken to calculate the first $10,000, 20,000, 30,000$, and so on up to the first $90,000$ primes. Use this information to estimate the time required to calculate the first $100,000$ primes.
    b. Remove the part of the procedure that collects the primes into a list. Repeat the previous part with this modified procedure.

c. Attempt to modify the procedure to more efficiently collate the primes into a list or sequence.

Produce the same lists of primes using *Mathematica*'s `Table` function and the inbuilt prime number functions, and measure how much time these take. Were these quicker or slower than the Eratosthenes' sieve procedure?

1.5 Further Explorations

This section presents open ended problems for the interested reader. The idea is to introduce mathematics that may be explored with no real constraints. Each topic may ask specific questions, or state particular calculations to be performed, however, these should be considered to be a *beginning* to exploration, and not an exhaustive set of steps to be performed. It is expected and encouraged that one explore these topics using one's own means.

1. The field of rational numbers may be extended to incorporate irrational numbers in a way similar to the way in which the real numbers are extended to the complex numbers. If R is the solution of the equation $x^2 = 2$, then it must be that $R^2 = 2$, and we know that R must be an irrational number.
   If we include our new number R into the rationals (i.e., consider $Q' = \mathbb{Q} \cup \{R\}$) then our new set Q' will no longer be a field. In particular, what is $1 + R$? In order to have a field, we need to add every rational multiple of R as well as addition of every rational number with every multiple of R.
   We end up with what is known as a *field extension*

   $$\mathbb{Q}(R) := \{a + bR \mid a, b \in \mathbb{Q}\}$$

   with the operations

   $$a + bR + c + dR = (a + c) + (b + d)R$$
   $$(a + bR)(c + dR) = ac + (ad + bc)R + bdR^2 = (ac + 2bd) + (ad + bc)R$$

   where $a, b, c, d \in \mathbb{Q}$.
   Mathematica allows exploration of these ideas with a group of functions, including the `AlgebraicNumber` and `Root` functions. The particulars are detailed in the Documentation Center page named "Algebraic Number Fields".

2. **The** $3n + 1$ **Problem**. This problem has many other names: Collatz's problem, the Syracuse problem, Kakutani's problem, Hase's algorithm, and Ulam's problem.
   We begin with the following simple algorithm that we apply recursively, starting with an arbitrary natural number, n say.

   - If n is even then halve it.
   - If n is odd then multiply it by 3 and then add 1.

   We continue this process with each new number obtained until we end up at 1. The actual problem is does this procedure always terminate? An equivalent formulation of the problem is, if we think of the numbers produced as an infinite sequence, does the sequence always end with a repetition of the subsequence $4, 2, 1$? For example, if we start with the number 13, then we get

$$13 \to 40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$$

or, equivalently, the infinite sequence

$$\{13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, \dots\}$$

For some starting values it will take a large number of steps, and on the way very large numbers might be encountered before the sequence finally begins to drop back to 1. Such sequences are sometimes called hailstone or juggler sequences.

Implement this algorithm in *Mathematica*. See what happens when you start with 7 (you can even check this particular one without the help of *Mathematica*). Then try some other starting values. The sequence starting at 27 takes one hundred and eleven steps to reach 1. You might experiment with the rule a little: for example, what happens if you change the 3 to a 5, thus making a "$5n + 1$" rule?

You should ask yourself how (i.e., under what circumstances) the system could possibly fail to terminate. For this to happen there must be a starting value that either diverges or settles into an infinite loop (other than $4, 2, 1$).

3. A real number is said to be *normal* if its numeric expansion in any base has a normal distribution. This is to say that each single digit is equally likely to appear, as is each pair of digits, each triple of digits, and so on. There are not many known normal numbers, and none are known that were not constructed to be normal in the first place. It has been conjectured, however, that the number π is normal.

We may obtain a "feel" for whether a number may be normal by counting its digits. If we count, say, 100 digits of a number, and find that there are, give or take, 10 of each digit then it's possible the number is normal. If we find this general idea holds for 1000, and even 10,000 digits (with approximately 100 or 1000 of each digit, respectively) the hypothesis looks even stronger. We could carry on and then count each pair of digits and see if there's roughly an even number of those. We could continue in this fashion, and even perform the same sort of analysis using a different base.

Note that a number may be normally distributed in a particular base representation, but not be a normal number. Such a number, if normally distributed in base b is said to be b-normal.

Some good numbers to begin looking at for possible normality are

$$\frac{1 + \sqrt{5}}{2}, \sqrt{\frac{1 + \sqrt{5}}{2}}, \pi, e, \log 2, \sqrt[3]{2}$$

Generate 100, 1000, and 10000 decimal digits of these numbers (more, if you wish), and count the digits, and pairs of digits. If you're feeling adventurous try triples and even quadruples.

Try with another base, binary say. You'll need to ask yourself, with what probability should each binary digit occur for normal distribution. With what probability should each double (as well as triple or quadruple if you test those) occur?

There is a similar analysis that can be performed on the continued fraction of a real number. The Gauss–Kuz'min distribution states that the probability that $a_n = k$ (in the continued fraction expansion $[a_1, a_2, \dots]$ of a random real number) is

$$\text{Prob}(a_n = k) = \log_2 \left[1 - \frac{1}{(k + 1)^2} \right]$$

In other words, approximately 42% of numbers in a continued fraction expansion will be 1, approximately 17% will be 2, around 9.4% will be 3, and so on. Note that because this is a continued fraction expansion, then it is possible for numbers greater than 9 to be in the expansion; in fact any natural number may be included (you should find that the infinite sum of these probabilities is equal to 1 as you would expect). As such, the distribution also tells us that approximately 1.4% of numbers in the expansion will be 10, 0.55% will be 50, and 0.14% will be 100.

Chapter 2
Calculus

In this chapter we explore calculus with the help of *Mathematica*. Much of what is seen in this chapter should be familiar (or at least recognized) from first-year study. We aim to revise this material, as well as visualize it in ways that are, one hopes, easily accessible and in addition provide new insight even to the more capable reader. We also attempt to introduce some newer (or, at least, less familiar) material. In all cases here the aim is to use *Mathematica* to complement and improve our own calculus skills; the goal is one of a human/machine collaboration, not that of an electronic replacement for calculus skills.

2.1 Revision and Introduction

In this section we introduce the *Mathematica* commands best suited for studying and performing calculus. In addition we recall key concepts from typical first-year calculus courses. It is, however, expected that the reader is familiar with the underlying concepts and is able to perform such first-year calculus including (but not limited to) differentiation and integration of single variable functions, evaluation of limits, and curve sketching, among others. The reader is encouraged to review his or her favorite (or most readily available) calculus text.

2.1.1 Plotting

In much first-year calculus the ability to be able to visualize the functions and concepts being studied is quite valuable, but usually not readily available. Indeed in almost anything involving calculus visualization is a powerful tool. So we begin this calculus chapter by looking at how to have *Mathematica* plot functions.

Briefly in Section 1.1.2 we saw a plot of a cubic. The *Mathematica* command to plot a function is, unsurprisingly, `Plot`. The most basic use of the `Plot` function is to simply give it an expression involving a single variable, usually x but any valid *Mathematica* variable name will work just as well, and an iterator to specify the bounds of this variable.

```
In[1]:= Plot[x^2, {x, -10, 10}]
```

Out[1]=
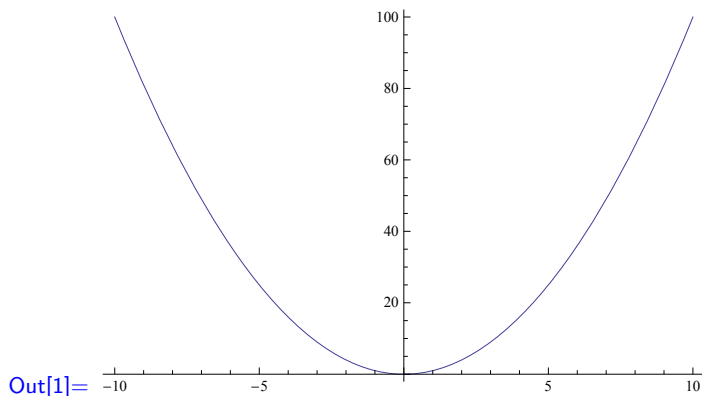
Mathematica automatically assigns the horizontal axis to be the axis for the independent variable (x in the previous example) but does not label it by default. The vertical axis is also unlabeled, but depicts—as it always does—the values of the function (or the dependent variable).

In[2]:= **Plot[Sin[t], {t, 0, 2 Pi}]**

Out[2]=
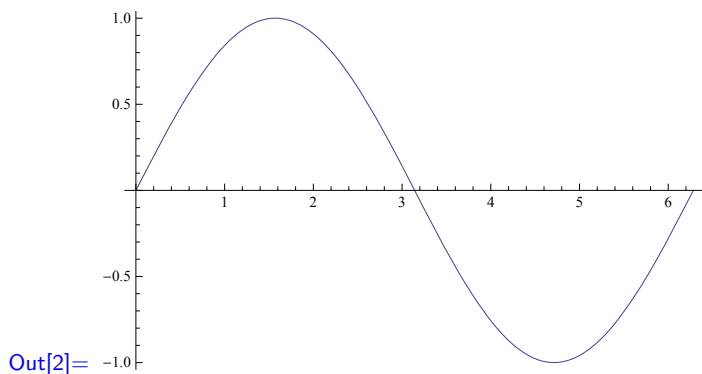
Notice the vertical axis in these two examples. The values of the vertical axis automatically adjust to suit the function we are trying to plot. For the parabola the vertical axis was between 0 and 100, corresponding to the values the parabola would have for $-10 \leq x \leq 10$, and the sine curve used vertical range $(-1, 1)$ just as we would have hoped it did. It should be interesting to note that the actual screen space taken up by the two plots is the same in both cases, showing that the vertical scale is different in both cases. In fact, the vertical scale and horizontal scale may be different even in the same plot, as is the case in both of these examples.

This automatic axis adjustment is actually a bit more clever than simply making sure that the vertical range corresponds exactly to the maximum and minimum values of the expression being plotted. For example, consider the following plot of the tan function between its asymptotes.

In[3]:= **Plot[Tan[theta], {theta, -Pi / 2, Pi / 2}]**

Out[3]=
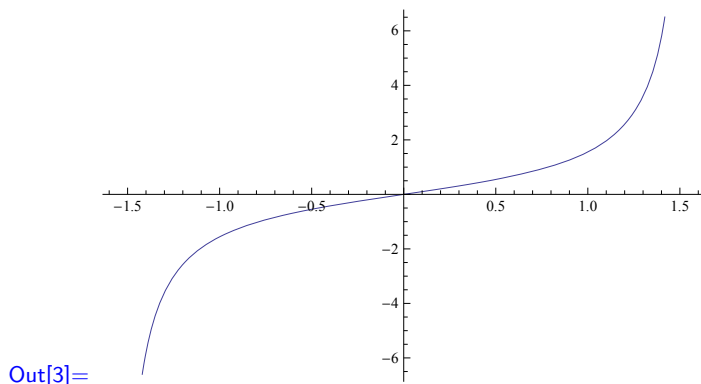
*Mathematica*has limited the vertical range to $(-6, 6)$, although we should know that $tan(\theta)$ approaches infinity as θ increases toward $\pi/2$. If we look carefully, we can see that the graph is only drawn for the horizontal range $(-1.4, 1.4)$, even though we specified that the plot should be in the range $(-\pi/2, \pi/2)$. By and large, *Mathematica*tends to be pretty smart with its plot ranges.

If, for some reason, we really did want the vertical range to correspond exactly to the maximum and minimum values of the expression being plotted, we may add `PlotRange -> Full` as an argument to the `Plot` function. This argument can be thought of as a plot option. There are many such options, and their order as function arguments does not matter, so long as they appear after the iterator.

```
In[4]:= Plot[Tan[theta], {theta, -Pi / 2, Pi / 2}, PlotRange -> Full]
```


Out[4]=
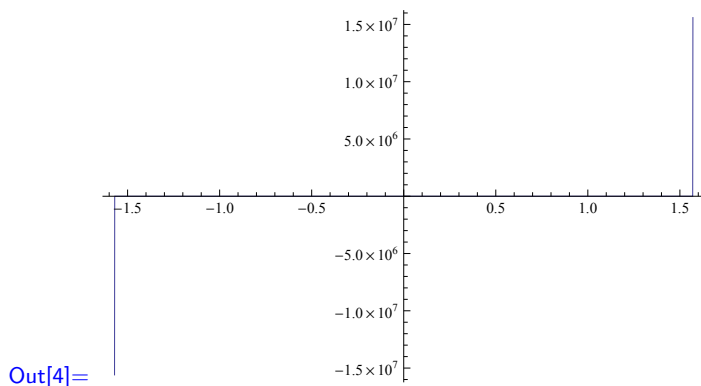
Unfortunately the plot we see certainly doesn't look like the tan function that we all know and love. Looking closely at this plot, we should notice that the scale of the vertical axis is exceptionally large, and that we have two seemingly vertical lines at the end of the graph. This is in keeping with our earlier observation of the behavior of $tan(\theta)$ as theta approaches $\pm\pi/2$.

In addition to either having the vertical range chosen automatically by *Mathematica*, or having a full vertical range—full in the sense that the range corresponds directly to the largest and smallest expression values being plotted, that is—we may also specify an exact plot range. To do this we use the `PlotRange` option again. To limit the vertical range to $\pm n$, we would use `PlotRange -> n`. To limit the vertical range to (m, n) then we would use `PlotRange -> {m, n}`.

```
In[5]:= Plot[Tan[theta], {theta, -Pi / 2, Pi / 2}, PlotRange -> 2]
        Plot[Tan[theta], {theta, -Pi / 2, Pi / 2}, PlotRange -> {-1, 2}]
```

Out[5]=

Out[6]=
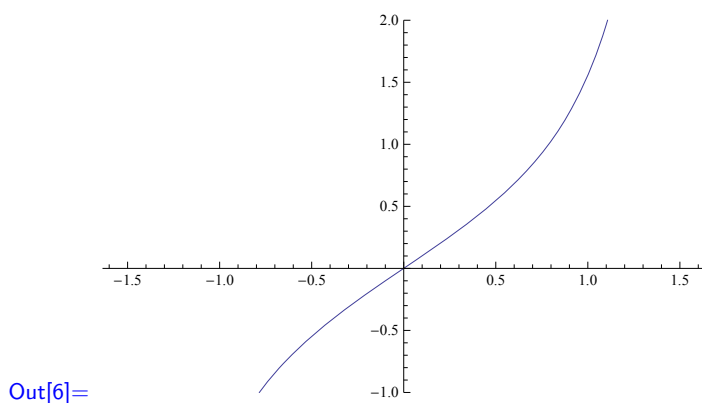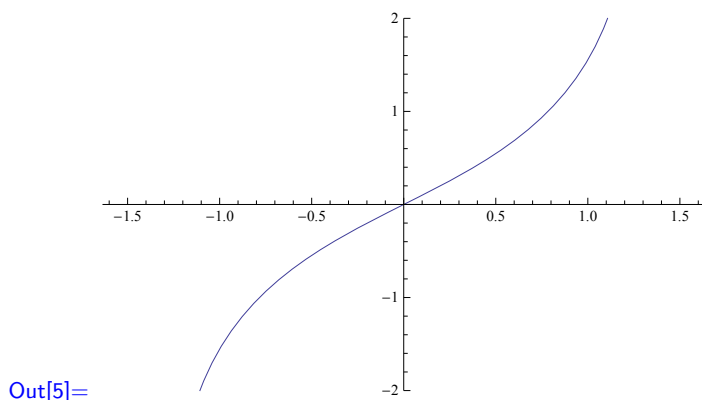
When *Mathematica* plots an expression it evaluates, or "samples" that expression at various points along the horizontal interval and fits a curve to the sampled points. We may request to see these sampling points with the **Mesh** plot option.

In[7]:= **Plot[Sin[theta], {theta, 0, 2 Pi}, Mesh -> Full]**

Out[7]=
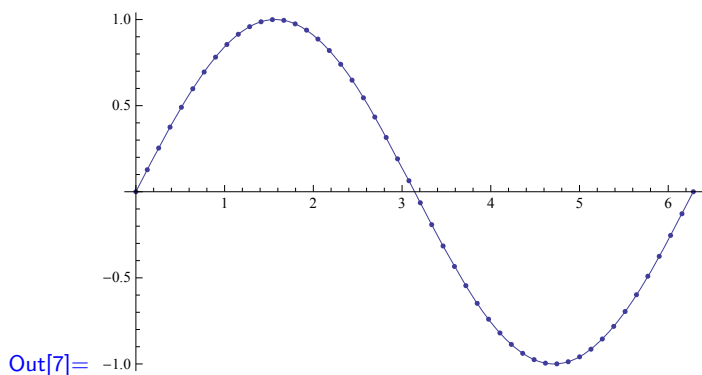
By default, *Mathematica* automatically chooses the number of sample points, but number of points may be explicitly specified with the **PlotPoints** plot option. We see approximately fifty in the above plot. Sample points are evenly spaced along the plot range, however *Mathematica* is rather clever and if it detects that the values of the expression are changing too quickly between sample points, it will sample the expression at an extra point between them to try to obtain more and better information with which to plot the function. This is known as "subdivision", and we may observe this by using the **Mesh -> All** option.

In[8]:= **Plot[Sin[theta], {theta, 0, 2 Pi}, Mesh -> All]**
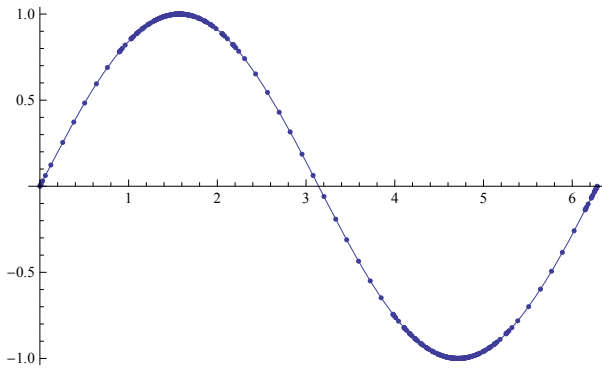
Out[8]=

The difference between **Mesh -> Full** and **Mesh -> All** is that the former shows us the initial sampling values, while the latter shows us the final sampling values, after subdivision has occurred. The number of subdivisions is, much like the number of sample points, automatically decided by *Mathematica*, but may be explicitly specified with the **MaxRecursion** plot option. In light of subdivision, one should be aware that *Mathematica* could potentially end up evaluating a function many more times than the number of sampling points requested. Mostly, however, we can just trust the *Mathematica* defaults to automatically do the sensible thing for us.

Now, armed with this knowledge, we'll have a look at the sampling values of above the tan plot with full plot range. In order to see only the mesh we specify **PlotStyle -> None**, however this also causes the mesh to not be displayed, so we also specify **MeshStyle -> Blue** to explicitly set the mesh to be blue, without changing the style of the plot.

In[9]:= **Plot[Tan[theta], {theta, -Pi / 2, Pi / 2},**
       PlotRange -> Full, Mesh -> All,
       PlotStyle -> None, MeshStyle -> Blue,
     ]
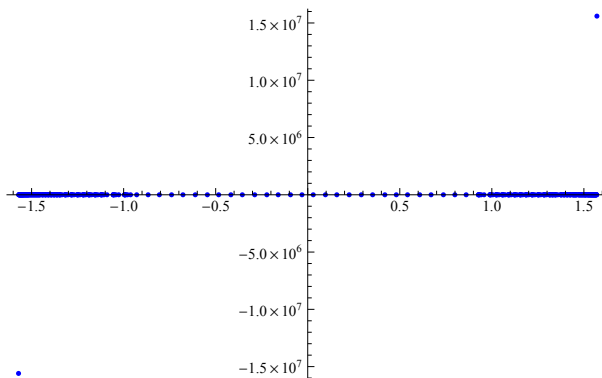
Out[9]=

We see two extreme points far to the top and bottom of the graph, with the remainder of the sampled points on or very near the *x*-axis. Of course, with the vertical range so large, the scale is such that the points seemingly on the *x*-axis could have values varying anywhere between $\pm 1{,}000{,}000$ or more and we wouldn't be able to tell the difference. This is in interesting, but not entirely surprising. We know how the function behaves near $\pm\pi/2$, and it seems that even with subdivision, the **Plot** function still couldn't draw a good picture with full range.

Without explicitly specifying the vertical range, *Mathematica* automatically produces a very sensible picture. It should be understood that explicitly requesting a full range will not always produce such a bad picture as we saw with the tan function, and may even be desirable for some plots. Modifying the plot range can be useful, and should be kept in the back of our minds when plotting functions. If ever we get a bad plot, modifying the plot range is a good first thing to try in order to get a better plot.

2.1.2 Multiple Plots

An interesting example comes to us from Borwein and Devlin [5]. Suppose we have two expressions, $y - y^2$ and $-y^2 \log(y)$, and wish to know (and eventually prove) which (if either) is always larger when $y \in (0, 1)$. A good first step would be to plot the two expressions over the unit interval, to see if the curves cross each other. However, up until now we have only plotted single expressions. Two separate plots are of limited (if any) use to us. We need a way to plot the two expressions on the same pair of axes. This is made possible in one of two ways.

The first method is far and away the simplest. *Mathematica*'s `Plot` command will happily plot a list of expressions. In place of a single expression we simply provide a list, and any parameters that modify the plots must also be provided in a list. For example, to plot the above two functions, with the first one being colored the usual dark blue, and the second colored red (so we may identify which is which) we would enter the following.

```
In[10]:= Plot[{y - y^2, -y^2 * Log[y]}, {y, 0, 1},
         PlotStyle -> {Automatic, Red}
       ]
```


Note that we specified a `PlotStyle` of `Automatic` here, whereas in the previous sub-section we used `Blue`. The astute reader might notice that the `Blue` plot style produces a slightly different color of blue to the usual plots. Specifying `Automatic` as a plot style, in this case, tells the `Plot` command to use the regular style for that expression. Note also that `PlotStyle` allows a much richer control of the look of a plot than simply changing colors, although. The reader is encouraged to explore this more in the Documentation Center.

For the second method, let us now consider a modified version of our example. This time we compare $y^2 - y^4$ and $-y^2 \log(y)$ (also from [5]). First we will make a simple observation: up until now, any valid *Mathematica* expression could be assigned to a

variable name. It should, then, be a natural question to ask whether the same can be done with the plot function. The answer is that yes it can, although the logistics of such an assignment are probably not obvious. Let's try this.

In[11]:= **plot1 = Plot[y^2 - y^4, {y, 0, 1}]**
plot2 = Plot[-y^2 * Log[y]}, {y, 0, 1}, PlotStyle -> Red]
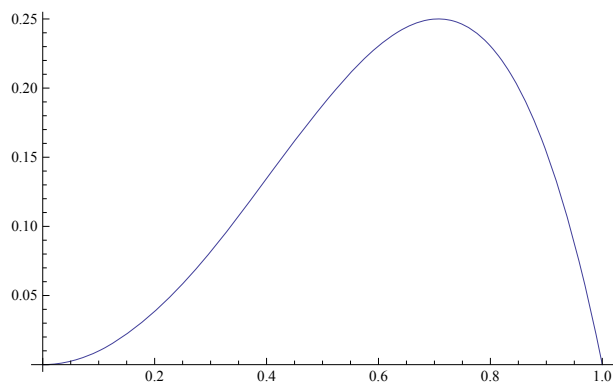
Out[11]=
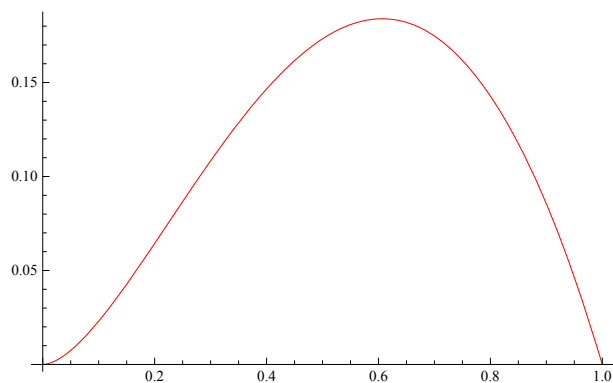
Out[12]=

Mathematica has stored the plots in the variables **plot1** and **plot2**, and has output them as well, just like it does with any assignment. Note that we may, if we wish, use a delayed assignment instead of a direct assignment. Doing so will essentially cause *Mathematica* to remember the plot command, instead of the plot image itself. In either case, if we wish to see one of the plots again, we need only to ask *Mathematica* for the contents of the variable in the usual way.

In[13]:= **plot1**
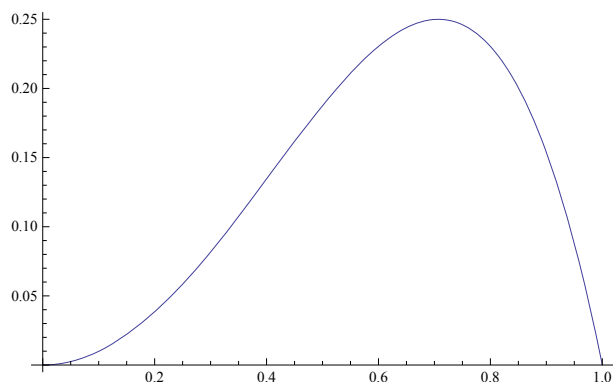
Out[13]=

This is all well and good, but it doesn't really help us show multiple plots on the same axes, at least not in any different way from the method we have already used. The key to this lies in the **Show** function, whose purpose is either to display a single stored plot using extra plot options which were not present when the plot was stored, or to display but multiple stored plots on a single set of axes. It is in this second capacity that we are currently interested. The **Show** function accepts an arbitrary number of stored plots as arguments, each plot as a separate argument, and display them all on the same axes. The **Show** function will also accept lists of stored plots as an individual arguments, and will display all the plots in all the arguments on the same axes, however the use of sets is not required.

In[14]:= **Show[plot1, plot2]**

Out[14]=
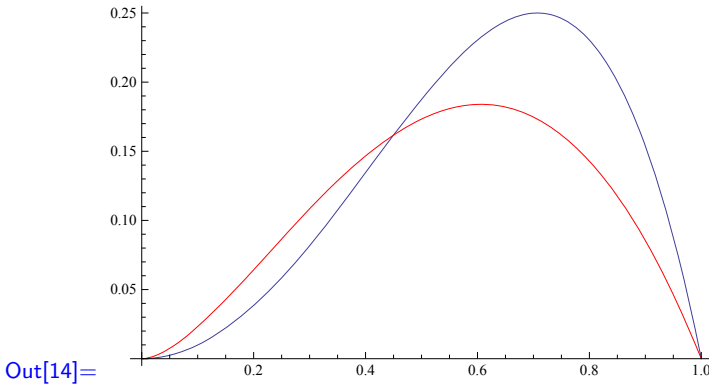
The use of **Show** for the above example above may seem unnecessarily long and complicated, when we could more easily use just a single **Plot** as we did in the first example. Such an observation is quite well founded. However, there are situations where the easier method is either impossible, or impractical to use. It is these cases where **Show** really shines.

To illustrate this utility, we show the previous example and a plot of its sample points at the same time. It does not seem possible to produce the same result with a single **Plot** command, if we want the meshes to be the same color as their corresponding plot. Trying to do so is left as an exercise to the interested reader.

In[15]:= **Block[{plot},**
         plot[1] = Plot[y^2 - y^4, {y, 0, 1}];
         plot[2] = Plot[-y^2 * Log[y]}, {y, 0, 1}, PlotStyle -> Red];
         Show[plot[1], plot[2]]
         ]

Out[15]=
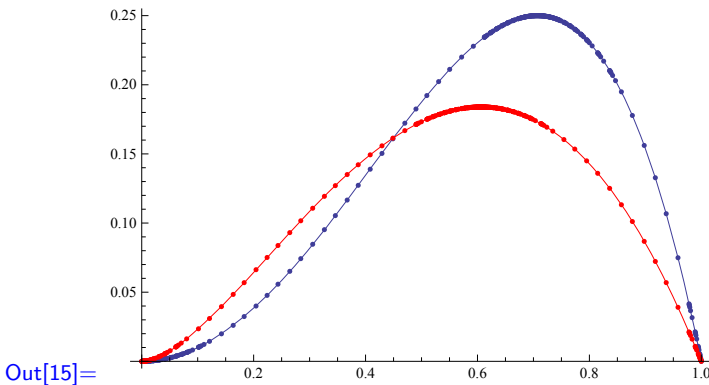
Inasmuch as the stored plots, `plot[1]` and `plot[2]` are only being plotted and stored so that we may produce the final picture, we use a `Block` and store them in temporary variables.

Storing plots in variables, along with the `Show` function, is also a very valuable tool when a plot is difficult and time consuming and we want to reuse it quickly and with certainty that it is what we wanted to draw.

2.1.3 Limits

Calculus, ultimately, all comes down to limits. So it is with limits that we begin our exploration of calculus proper. We have already seen and used very briefly in the previous chapter, the *Mathematica* `Limit` command. We look at it in more detail here, and recall quickly the math behind limits.

We may take a limit of a sequence (of the infinite variety) or of a function. The intuitive (and mathematically imprecise) notion of a limit is a value that we may approach as closely as we could ever wish, just by traveling sufficiently far along the sequence or function.

We start with sequences. Recall that the limit, L say, of some sequence

$$\{x_n\}_{n=1}^{\infty} = x_1, x_2, x_3, \cdots$$

written

$$\lim_{n \to \infty} x_n = L$$

is a number such that for every $\epsilon > 0$ we can find a natural number N so that whenever we have any other number $n \geq N$ it will be the case that $|x_n - L| \leq \epsilon$.

The sequence

$$\left\{ \frac{1}{k} \right\}_{k=1}^{\infty} = 1, \frac{1}{2}, \frac{1}{3}, \cdots$$

should be familiar and has, of course, limit 0. We ask *Mathematica* to verify this. The *Mathematica* `Limit` command.

```
In[16]:= Limit[1 / k, k -> Infinity]
```
```
Out[16]= 0
```

If we want to see something a little more mathematical, we can use the `HoldForm` function to prevent the `Limit` from being evaluated. We may combine this with the `TraditionalForm` function to tell *Mathematica* to print the limit in traditional mathematical notation, as well as the `ReleaseHold` function to tell *Mathematica* to evaluate the un-evaluated limit. To simplify this a little, we use the `With` function, with the held limit as a constant temporary variable.

```
In[17]:= With[{L = Limit[1 / k, k -> Infinity] // HoldForm},
            (L // TraditionalForm) == (L // ReleaseHold)
          ]
```
$$\text{Out[17]}= \quad \lim_{k \to \infty} \frac{1}{k} == 0$$

This is not something we will do often. The mathematical nature of the output is nice, but the effort required to produce it is, in the opinion of the author at least, not

worth persisting with. The technique gets particularly troublesome when the expressions involved begin to involve either expressions stored in variables, or nested functions. For example, producing mathematical notation output for the limit of partial sums, seen below, proved to be quite tedious. The interested reader is encouraged to look up these functions in the Documentation Center, and to experiment with them. In the interests of simplicity, however, we avoid this technique.

Returning to the matter of the converging series, if we wish a more visual clarification of the convergence, we may plot the series. We might simply plot the continuous, real-valued, function $1/x$ to see the convergence (using the fact that if the function converges, then the sequence evaluated only at integer points also converges). Instead, however, we use *Mathematica*'s `ListPlot` function and see only the points of the sequence in our visualization.

To do this we construct a list corresponding to a sub-sequence of the form $\{1/k\}_{k=1}^{n}$ for some n. The `ListPlot` function will understand that the first element of the list corresponds to the value of $k = 1$, the 2nd element to $k = 2$, and so on and so forth.

In[18]:= `Table[1 / k, {k, 1, 100}] // ListPlot`
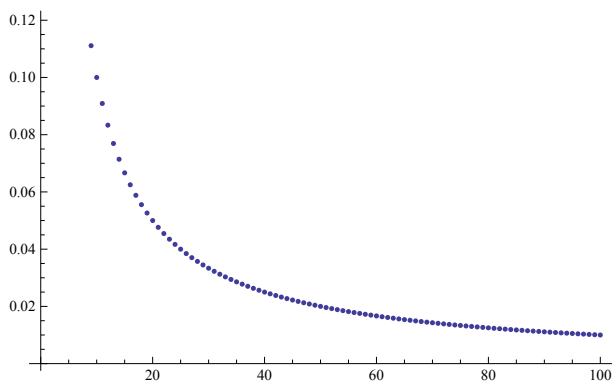
Out[18]=

The `ListPlot` function will also accept a list of 2-element lists, where the internal lists are understood to be points on the cartesian plane, and will plot those points. Using this method, we may produce the same plot by constructing the sequence of pairs `{k, 1 / k}` as follows, although the above method is perhaps easier to understand at a glance.

In[19]:= `{Table[{k, 1 / k}, {k, 1, 100}]} // ListPlot`
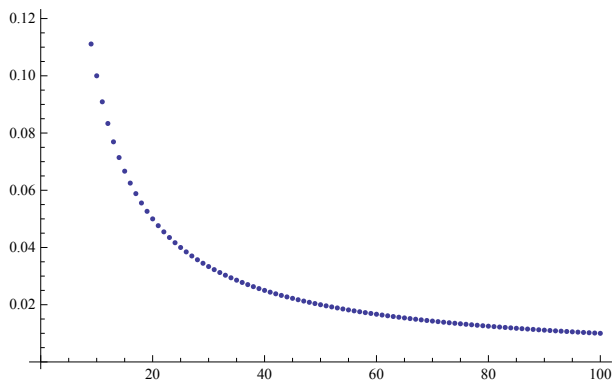
Out[19]=

The convergence is visually pretty clear. Notice that the automatic scaling we saw from the `Plot` function in Section 2.1.1 is well and truly in effect here. This is not a

bad thing in this case, as convergence is only concerned with what how the sequence behaves in the long term, not how it behaves at the beginning. We may, of course, insist on seeing all of the hundred points we computed, and doing so certainly isn't a bad idea. Note that to do this we need to add an argument to the **ListPlot** function, and doing this means that it has become a function of two arguments, and therefore not susceptible to the postfix notation. We use the regular notation instead.

In[20]:= `ListPlot[Table[1 / k, {k, 1, 100}], PlotRange -> Full]`
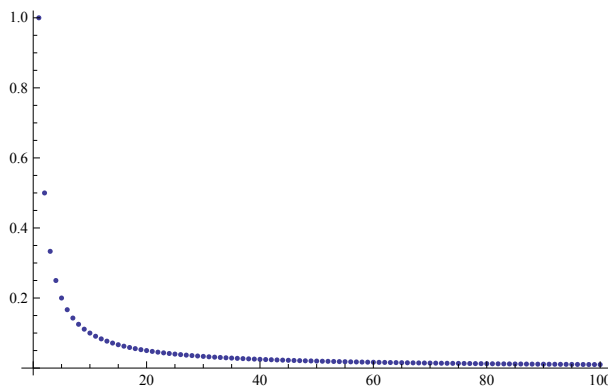
Out[20]=

It is worth stressing at this point, however, that these plots give an *indication* of convergence, not a proof of convergence. There is always the possibility that the sequence does something odd after the interval we have plotted. So we must still perform regular mathematics to verify the limits, or at the very least ask *Mathematica* to evaluate the limit.

Recall now that an infinite sum is defined to be a limit of its partial sums. Mathematically, that is,

$$\sum_{k=1}^{\infty} f(k) = \lim_{N \to \infty} \sum_{k=1}^{N} f(k)$$

We have already used *Mathematica*'s **Sum** command to calculate infinite sums for us in Section 1.1.5, but we do this again now, and demonstrate the limit property. Let us use the series $1/k^2$ again, which we know from the previous section converges to $\pi^2/6$.

In[21]:= `Sum[1 / k^2, {k, 1, n}]`
`Limit[%, n -> Infinity]`

Out[21]= HarmonicNumber[$n, 2$]

Out[22]= $\dfrac{\pi}{6}$

We have not seen the **HarmonicNumber** function before, although we may ask *Mathematica* about it by consulting the Documentation Center. We discover that **HarmonicNumber[n, r]** corresponds to the harmonic number which is denoted mathematically by $H_n^{(r)}$, where

$$H_n^{(r)} = \sum_{i=1}^{n} \frac{1}{i^r}$$

As such, *Mathematica* has just told us that our partial sum is equal to itself. This is not as insignificant as it may appear, as it also tells us that *Mathematica* knows no better closed form than the harmonic number. Of course, the limit of $\pi/6$ is exactly what we

know the sum to converge to, and this is what *Mathematica* also computes the limit of the `HarmonicNumber` function to be in this case.

Now let us look at limits of continuous functions. For this purpose we consider the function $f(x) = 1/(x^3 - 2x^2)$. Our first impulse should be to plot the function to see what it looks like. In order to plot, we need bounds to plot over, and we choose the range $(-10, 10)$, for no better reason than it is the default plotting range for another program the author uses from time to time.

In[23]:= `Plot[1 / (x^3 = 2 x^2), {x, -10, -10}]`
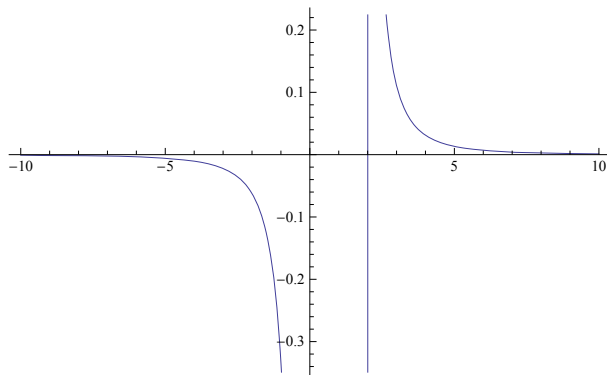
Out[23]=

This is not a very useful plot, although precisely why that is may not be readily apparent. We could attempt to use trial and error to find a good interval to plot over, but on reflection, perhaps a little elementary calculus might have indicated better bounds to begin with, so we will do that now.

First note that the denominator is equal to $x^2(x-2)$, which tells us that the function is not defined at $x = 0$ or $x = 2$ and that we should probably expect vertical asymptotes at those points. This we already see in the plot, however the function is well-defined everywhere else, and so we should see plot lines between these vertical asymptotes, which are not apparent in the above plot. It should also be clear, using the algebra of limits, that

$$\lim_{x \to a} \frac{1}{x^3 - 2x^2} = \lim_{x \to a} \frac{\frac{1}{x^3}}{1 - \frac{2}{x}} = \frac{\lim_{x \to a} \frac{1}{x^3}}{1 - \lim_{x \to a} \frac{2}{x}}$$

and so for $a = \pm\infty$ the limit will be 0.

The limits at $a = 0$ and $a = 2$ are only a little bit trickier to work out. We again look at the factored denominator $x^2(x - 2)$ and observe that

$$\lim_{x \to 0} x^2(x - 2) = 0 \text{ and } \lim_{x \to 2} x^2(x - 2) = 0$$

Furthermore we can see that x^2 will always be ≥ 0, and that for all points near $a = 0$ it is the case that $x - 2 < 0$ so the denominator near $a = 0$ must always be negative. We conclude therefore that the limit at $a = 0$ is $-\infty$. Finally observe that for $x > 2$ we have $x - 2 > 0$ and that for $x < 2$ we have $x - 2 < 0$ which tells us that $f(x) \to -\infty$ as $x \to 2^-$ and that $f(x) \to \infty$ as $x \to 2^+$

We now have plenty of information for us to choose appropriate plotting ranges. In order to put the undefined points (with the vertical asymptotes) evenly spread across the x-axis we plot across the interval $x \in (-2, 4)$. However, also we know that the function has vertical asymptotes, and that the `Plot` function, which normally handles the vertical range well, didn't do so well with the previous plot. We should, therefore, consider the y-axis range. A quick substitution of $x = 1$ into the function shows that

the midpoint between the vertical asymptotes attains the value -1, yet the above plot limited the range to, roughly, $(-0.4, 0.25)$. If we want to see the asymptotes shooting off a little ways, a reasonable guess is that $y \in (-5, 5)$ should suffice. First, however we will see if the plot behaves better now that we have more sensible bounds on x.

In[24]:= `Plot[1 / (x^3 = 2 x^2), {x, -2, 4}]`
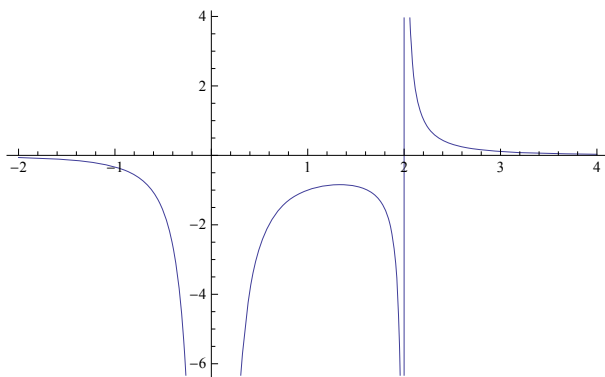
Out[24]=

Well, that's much better. Notice that the automatic vertical range of, approximately, $(-6, 4)$ is quite close to the one we guessed at, above. The discontinuity at $x = 2$ produces a vertical line in the plot, which we know shouldn't be there. Despite the fact that the Documentation Center tells us that the default behavior of certain plot options is to detect discontinuities, and to plot them properly, this line persists despite the best efforts of the author. We shall just have to live with it.

We now verify the limits with *Mathematica*. We provide, as the second argument to the `Limit` function, a list of bounds for the limit. The `Limit` function understands that this means we want to calculate the limit for each of these cases, and gives the answer back in a list.

In[25]:= `With[{p = 1 / (x^3 = 2 x^2)}`
         `Limit[p, {x -> -Infinity, x -> 0, x -> 2, x -> Infinity}]`
         `]`

Out[25]= $\{0, -\infty, \infty, 0\}$

The undefined limit at $a = 2$ should be surprising, thanks both to the plot and the calculus we performed above. The limit from below and the limit from above are not the same, and therefore the limit should not exist, yet *Mathematica* has told us that the limit is ∞. Recall that $\lim_{x \to a^-} = \lim_{x \to a^+} = L$ if and only if $\lim_{x \to a} = L$. That *Mathematica* gives us an answer for the limit as $x \to 2$ at all—at least any answer other than "undefined"—is misleading. This is a good reminder that we should not blindly trust the answers the computer gives us, and that we should keep our wits about us when using a CAS.

As it happens, *Mathematica* can actually handle single directional limits by allowing the `Limit` command to take a third input variable for this purpose, which is the `Direction` option. This may be either of `Direction -> 1` for the limit from below (increasing values of the variable) or `Direction -> -1` for the limit from above (decreasing values of the variable). Somewhat confusingly, this means that for the limit at a when $x \to a^-$, we must use `Direction -> 1`, and for the limit at a is when $x \to a^+$ we must use `Direction -> -1`.

In[26]:= `With[{p = 1 / (x^3 = 2 x^2)}`
            `Limit[p, x->2, Direction->-1], Limit[p, x->2, Direction->1]`
          `]`

Out[26]= $\{-\infty, \infty\}$

Now we see the answers that we should expect. Note that there wasn't a particularly simple way to use the **Direction** option with the nifty list technique we used, above, to produce the list of limits. As such, we simply computed the list directly, typing out the **Limit** command twice, once with each **Direction** option.

We now return to sequence limits, and a cautionary example. *Mathematica*'s **Limit** function calculates real- or complex-valued (function) limits. When we used it to calculate sequence limits above, what we were really doing was evaluating the real-valued limit at infinity of the functions in question, and using the theorem which states that if $f(x) \to L$ as $x \to \infty$ exists for a real-valued function f, then the sequence $\{f(n)\}_{n \in \mathbb{N}}$ converges to the same limit as $n \to \infty$.

However, caution is advised. This relationship between limits of functions and limits of sequences does not always work the other way around. Consider the function

$$f(x) = x \sin(Pi \cdot x) + 2 \tanh(x - 5)$$

and the corresponding sequence $\{f(n)\}_{n \in \mathbb{N}}$. If we evaluate limits or plot the function we might very well be tempted to conclude that the sequence does not converge.

In[27]:= `f[x_] := x * Sin[Pi * x] + 2 * Tanh[x - 5]`

In[28]:= `Limit[f[x], x -> Infinity]`

Out[28]= $\text{Interval}[\{-\infty, \infty\}]$

In[29]:= `Plot[f[x], {x, -10, 10}]`
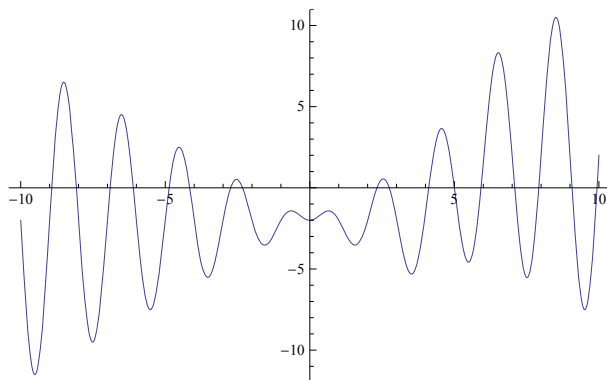
Out[29]=

We clearly see a function with no limit at infinity. However, plotting only the sequence points shows an entirely different picture. Note that we have not made **f** a local variable, but a global one. This has allowed (and continues to allow) us to reuse the expression with a minimum of typing. Because we are working with this expression interactively, we are unable to use a **Block** or a **With** function. At least, if we want the convenience of being able to choose each command based the output we see from a previous command, we can't use the those functions.

In[30]:= `Table[f[n], {n, 1, 100}] // ListPlot`

Out[30]=
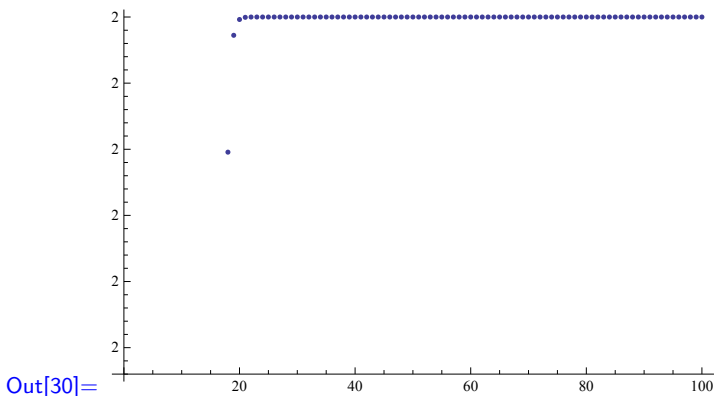
Our sequence has what looks very much like a limit. Applying our calculus knowledge, we see that $n \sin(\pi n) = 0$ for $n \in \mathbb{N}$, and so that part of the function will not affect the sequence at all. Furthermore, because $\tanh(x) \to 1$ as $x \to \infty$ it must be the case that $2 \tanh(x - 5) \to 2$ as $x \to \infty$. In short, the sequence $\{f(n)\}_{n\in\mathbb{N}}$ ought to converge to the value 2, and the plot of the sequence points exhibited precisely this behavior.

If we plot the real-valued function and the sequence on the same axes, we can see the convergence a little better. We color the continuous function light gray, so that it won't over-shadow the points of the sequence. Note that *Mathematica* puts the vertical axis at $x = 5$, not the usual $x = 0$ that we might have been expecting.

```
In[31]:= Block[{p},
           p[1] = Plot[f[n], {n, 1, 20}, PlotStyle -> LightGray]
           p[2] = ListPlot[Table[f[n], {n, 1, 20}]]
           Show[{p[1], p[2]}]
         ]
```


Out[31]=
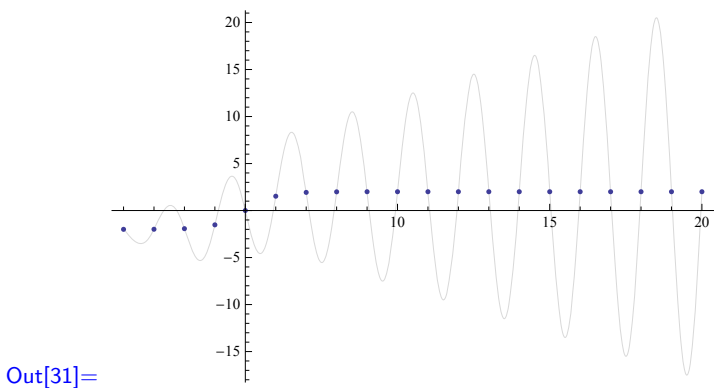
All that remains is to see if we may convince *Mathematica* to provide us with the correct answer for the sequence limit. Note that because we are evaluating a limit at infinity we are already evaluating a limit from below, and so we do not need to worry about specifying the direction of the limit. In order to tell *Mathematica* that we are only interested in the integers, we use the **Assumptions** option to the **Limit** function.

```
In[32]:= Limit[f[n], n -> Infinity, Assumptions -> n ~Element~ Integers]
```

Out[32]= 2

We have used the assumption that $n \in \mathbb{Z}$ here by using the **Element** function, and the **Integers** keyword. We might have been tempted to try the **IntegerQ** function

instead, but its behavior of always returning either true or false means that, in this case, it evaluates to false before *Mathematica* has even had a chance to try evaluating the limit. We might also have be tempted to use the **Refine** function that we saw in Section 1.2.1, however that also proves not to work. That the **Limit** function, as well as some others, provides its own mechanism for specifying assumptions ought to be an indication that this is the preferred way to go about specifying assumptions for limits.

To reiterate the key points here, the lack of a limit of a real valued function does not imply the lack of a limit of a sequence of evaluations of that function, and—more important—one must always keep one's wits about one when using a CAS (of course this is true when reading a book or taking a bus too).

2.1.4 Differentiation

Differentiation is, fundamentally, all about calculating rates of change. Recall that the derivative of a function, $f(x)$ say, at any point a is the slope of the tangent line to f at the point a. Recall, also, that the tangent at a is defined to be the line through a with slope equal to the limit of the slopes of lines drawn between a and points near a, as depicted in Figure 2.1. So it is that we come to the limit definition of the derivative
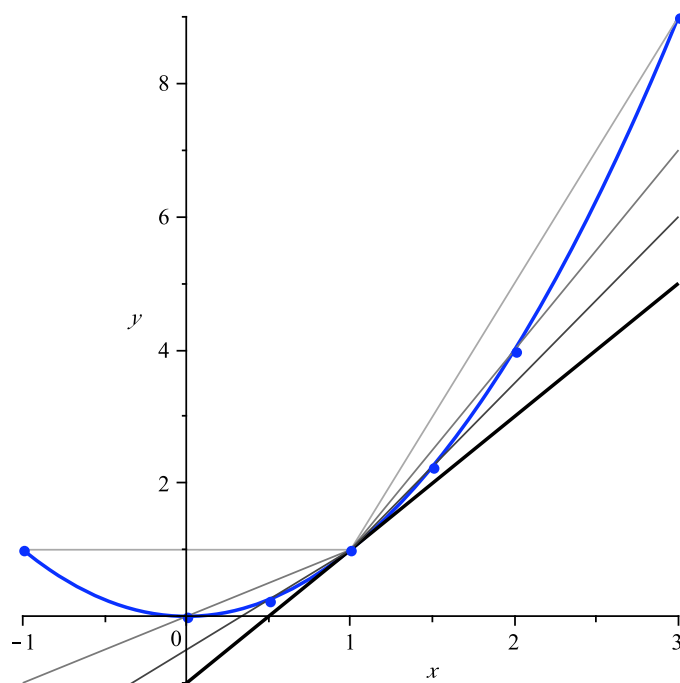
Fig. 2.1 Depiction of the convergence of lines to the tangent.

at a point a as

$$f'(a) = \lim_{x \to a} \frac{f(x) - f(a)}{x - a} = \lim_{h \to 0} \frac{f(a+h) - f(a)}{h}$$

Let us explore this a little before we introduce *Mathematica*'s native differentiation commands. We start with a parabola $f(x) = x^2$ which we know has derivative $f'(x) = 2x$ meaning that the tangent to the parabola at any point a has slope $2a$. Let's have a look at this in *Mathematica*. First, we make sure to clear our previous definitions and assignments.

```
In[33]:= ClearAll["Global`*"]
```

```
In[33]:= Block[{f, a, h},
          f[x_] := x^2;
          (f[a+h] - f[a]) / h
        ]
        Limit[%, h -> 0]
```

$$\text{Out[33]=} \quad \frac{-a^2 + (a + h)^2}{h}$$

Out[34]= $2a$

If we look a little closer at the expression in the limit, we should see that

$$(a + h)^2 - a^2 = a^2 + 2ha + h^2 - a^2 = h(2a + h)$$

and so the limit then becomes

$$\lim_{h \to 0} \frac{h(2a + h)}{h} = \lim_{h \to 0} (2a + h) = 2a$$

thus verifying both *Mathematica*'s limit calculation and our regular differentiation technique (for the parabola, at least).

Turning our eye now to trigonometric functions, we choose the sin function and the point $a = \pi/2$. This produces a limit that is a little trickier to work out on paper.

```
In[35]:= With[{a = Pi / 2},
          (Sin[a+h] - Sin[a]) / h
        ]
        Limit[%, {h -> 0}]
```

$$\text{Out[35]=} \quad \frac{-1 + \text{Cos}[h]}{h}$$

Out[36]= 0

As ever, our instinct should be to plot the function. However, try to get a good idea of the behavior near the undefined point at $h = 0$, we ask for a lot of sample points.

```
In[37]:= Plot[(Cos[h] - 1) / h, {h, -1, 1}, PlotPoints -> 1000]
```
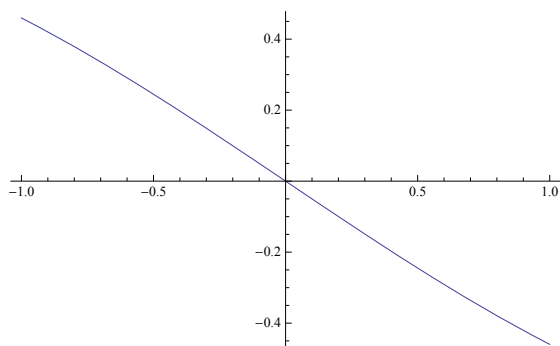

Out[37]=

Of course, in order to perform differentiation in *Mathematica* we need not perform limit calculations each and every time. The function for computing derivatives is the `D` function. It takes an expression as its first argument, and a variable name as its second argument, and computes the derivative of that expression with respect to that variable.

```
In[38]:= D[x^2, x]
         D[Sin[x], x]
         D[t^2 + t + 1, t]
         D[E^(2 var), var]
```

Out[38]= $2x$

Out[39]= $\text{Cos}[x]$

Out[40]= $1 + 2t$

Out[41]= $2e^{2\,\text{var}}$

If we wish to calculate a second derivative, we do the following.

```
In[42]:= D[t^2 + t + 1, {t, 2}]
```

Out[42]= 2

In general, for the kth derivative of some expression, **expr** say, we use the command `D[expr, {x, k}]`.

```
In[43]:= With[{p = 4 x^5 - 3 x^2 + x - 2}
           D[p, {x, k}] // Print,
           {k, 1, 6}
         ]
```

Out[43]= $1 - 6x + 20x^4$

Out[44]= $-6 + 80x^3$

Out[45]= $240x^2$

Out[46]= $480x$

Out[47]= 480

Out[48]= 0

If we have a function we want to differentiate, we may use the more familiar ´ notation, as we're used to seeing with, say f'. Doing so produces a new function which we may then give arguments for computation. For instance.

```
In[49]:= Exp´[x]
         Cos´[x]
         Sin´[Pi]
         Sin´[Pi / 2]
```

Out[49]= e^x

Out[50]= $-\text{Sin}[x]$

Out[51]= -1

Out[52]= 0

2.1.5 Integration

Integration grows out of the problem of calculating area underneath a curve, although its applications are far more wide and varied than that. Recall that a definite integral of a continuous function f between two points a and b may be approximated by a limit of rectangles.

$$\int_a^b f(x)\,dx = \lim_{n\to\infty}\left(\Delta x \sum_{k=1}^n f(a+k\Delta x)\right)\ \text{ where }\Delta x := \frac{b-a}{n}$$

In fact, the approximation can be made from rectangles coming from an arbitrary partitioning of the interval (a, b) with the heights of the rectangles being taken from arbitrary points within each of the partition elements. See [12] or any elementary calculus text for more details.

```
In[53]:= int[f_ , {a_ , b_}] := With[{delta = (b - a) / n},
           delta * Sum[f [a + k * delta], {k, 1, n}]
         ]

In[54]:= Block[{f},
           f[x_] := x^2;
           int[f, {0, 2}]
         ]
         Limit[%, n -> Infinity]
```

$$\text{Out[54]}=\ \frac{4(1+n)(1+2n)}{3n^2}$$

$$\text{Out[55]}=\ \frac{8}{3}$$

```
In[56]:= int[Sin, 0, Pi]
         Limit[%, n -> Infinity]
```

$$\text{Out[56]}=\ \frac{\pi\left(\mathrm{Csc}\left[\frac{\pi}{2n}\right]\mathrm{Sin}\left[\frac{(-1+n)\pi}{2n}\right] + \mathrm{Csc}\left[\frac{\pi}{2n}\right]\mathrm{Sin}\left[\frac{(1+n)\pi}{2n}\right]\right)}{2n}$$

$$\text{Out[57]}=\ 2$$

```
In[58]:= int[Sin, 0, 2 Pi]
         Limit[%, n -> Infinity]
```

$$\text{Out[58]}=\ \frac{\pi\left(\mathrm{Csc}\left[\frac{\pi}{n}\right]\mathrm{Sin}\left[\frac{(-2+n)\pi}{2n}\right] + \mathrm{Csc}\left[\frac{\pi}{n}\right]\mathrm{Sin}\left[\frac{(2+3n)\pi}{2n}\right]\right)}{n}$$

$$\text{Out[59]}=\ 0$$

Once this limit is established, then we are presented with the fundamental theorem of calculus which states that

$$\int_a^b f(x)\,dx = F(b) - F(a)\ \text{ where } F' = f$$

and nicely links differentiation and definite integrals. We also, by convention, denote the indefinite integral

$$\int f(x)\,dx = F(x) \Leftrightarrow F' = f$$

We may use this to check the limits found above. First we have $x^3/3$ as an antiderivative of x^2, and evaluating

$$\frac{2^3}{3} - \frac{0}{3} = \frac{8}{3}$$

verifies the integral. Similarly we have $-\cos$ as an antiderivative of $\sin$ leading us to $-\cos(\pi) - (-\cos(0)) = 1 + 1 = 2$ and $(-\cos(2\pi) - (-\cos(0))) = -1 + 1 = 0$. In fact, the final integral can be verified by the symmetric nature of the sine graph between 0 and 2π.

Again, as with differentiation, we need not perform limit calculations within *Mathematica* if we wish to calculate an integral, as there is a handy function named `Integrate`. The `Integrate` function can handle both definite and indefinite integrals and takes an expression as its first argument, and either a variable, or an iterator as its second argument. If the second parameter is an iterator, it is understood that we wish to perform a definite integral with respect to the variable, over the range specified in the iterator. In the case that the second argument is just a variable, then it is understood that we wish to perform an indefinite integral with respect to that variable.

```
In[60]:= Integrate[x, {x, 1, 3}]
         Integrate[Sin[x], x]
```

$$\text{Out[60]}= \frac{26}{3}$$

$$\text{Out[61]}= -\text{Cos}[x]$$

2.2 Univariate Calculus

2.2.1 Optimization

Suppose we wish to calculate the longest ladder that we may carry around a corner with one corridor 2 meters in width, and the other 1 meter in width. This is an example of an Optimization problem. We may use calculus (specifically, differentiation) to solve problems along these lines.

Given any angle θ we know that

$$x = \frac{1}{\cos(\theta)} \ \text{ and } \ y = \frac{2}{\sin(\theta)}$$

Therefore, the length L of a ladder that touches the corner, and the opposite walls of each corridor at an angle of θ to the corner, is given by the formula

$$L := \frac{1}{\cos(\theta)} + \frac{2}{\sin(\theta)}$$

We can see straight away that $0 < \theta < \pi/2$ and that $1/\cos(\theta) \to \infty$ as $\theta \to (\pi/2)^-$ as well as $2/\sin(\theta) \to \infty$ as $\theta \to 0^+$ which tells us that our L function will tend toward infinity at its endpoints. We also know that $L > 0$ for all values of θ, just as we would expect for the length of a ladder. We plot the function, being sure to limit the y-axis.

```
In[62]:= L[theta_] := 1 / Cos[theta] + 2 / Sin[theta]
```

```
In[63]:= Plot[L[theta], {theta, 0, Pi / 2}]
```

Out[63]=
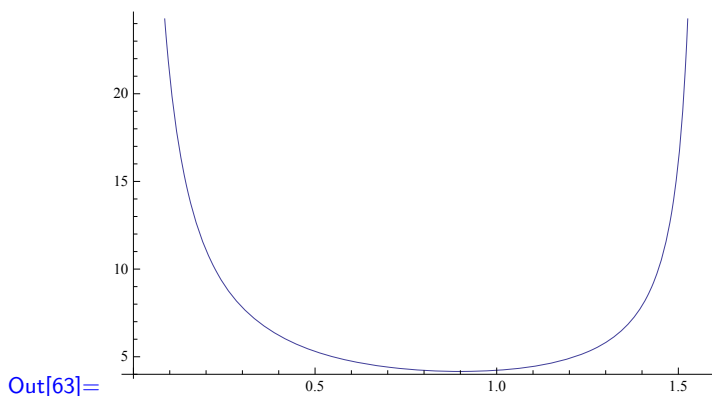
What is interesting here is that the function is concave up and has no maximum values, although it does have a minimum value. What's going on here? Well, our function L, as we know, calculates the length of a ladder at an angle of θ that touches both the far walls of the two corridors as well as the corner. Such a ladder is the longest possible ladder that can rest at that particular angle. However, if a ladder is to be carried around the corner, then it must go through all angles from $\theta = 0$ to $\theta = \pi/2$ and not become stuck. What we are, in essence, looking for is the shortest of all such longest ladders, and hence a minimum of the function L.

We can see that the minimum value lies somewhere in the range $0.5 \le \theta \le 1$. With a little trial and error, we may find the following plot and so can do a little better with these bounds. Looking at the plot (below) we can clearly see the the minimum lies to the right of the midpoint of the θ-axis. Even though it is not labelled, we know this midpoint must be $\pi/4$ because our plot is for theta between 0 and $\pi/2$. Our minimum, therefore, must lie in the range $\pi/4 \le \theta \le 1$.

In[64]:= **Plot[L[theta], {theta, 0, Pi / 2}, PlotRange -> {4, 10}]**

Out[64]=
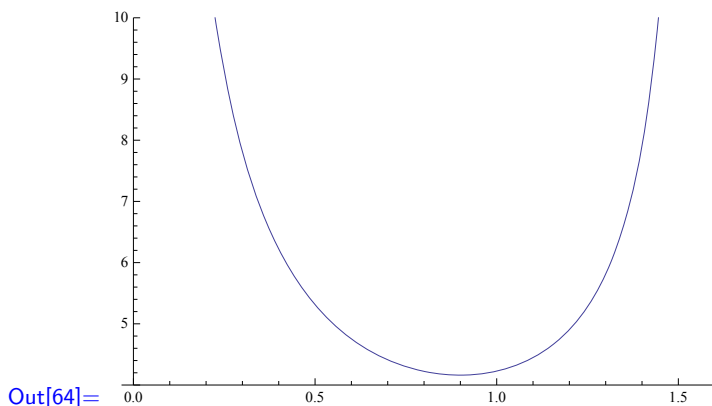
We could, if we wished "zoom in" by plotting the region $\pi/4 \le \theta \le 1$ and $4 \le L \le 5$ in the hopes of obtaining better bounds. We could even continue along these lines for some time. There is little point in doing so, at least in this case. Instead we proceed to find the minimum symbolically. The minimum is a turning point, and so will have a derivative of 0. We therefore solve $L'(\theta) = 0$.

In[65]:= **L´[theta]**
       Solve[% == 0, theta]

Out[65]= $-2 \operatorname{Cot}[\text{theta}] \operatorname{Csc}[\text{theta}] + \operatorname{Sec}[\text{theta}] \operatorname{Tan}[\text{theta}]$

Out[66]= $\Big\{ \Big\{ \text{theta} \to \text{ArcCot}\Big[\frac{1}{2^{1/3}}\Big]\Big\}, \Big\{ \text{theta} \to i\,\text{ArcCoth}\Big[\frac{1}{4}\Big(-i2^{2/3}-2^{2/3}\sqrt{3}\Big)\Big]\Big\},$

$\Big\{ \text{theta} \to -i\,\text{ArcCoth}\Big[\frac{1}{4}\Big(i2^{2/3}-2^{2/3}\sqrt{3}\Big)\Big]\Big\},$

$\Big\{ \text{theta} \to i\,\text{ArcCoth}\Big[\frac{1}{4}\Big(-i2^{2/3}+2^{2/3}\sqrt{3}\Big)\Big]\Big\},$

$\Big\{ \text{theta} \to -i\,\text{ArcCoth}\Big[\frac{1}{4}\Big(i2^{2/3}+2^{2/3}\sqrt{3}\Big)\Big]\Big\}\Big\}$

Well, that's a bit of a mess. *Mathematica* seems to have given us five solutions, four of which appear to be complex, and has even warned us that some solutions may not have been found. Nonetheless, there is clearly a real solution there, and evaluating it numerically shows it to be in the range we were expecting.

In[67]:= `guess = ArcCot[2^(-1/3)]`
        `guess // N`

Out[67]= $\text{ArcCot}\Big[\frac{1}{2^{1/3}}\Big]$

Out[68]= 0.899908

In[69]:= `L[guess]`
        `% // N`

Out[69]= $\sqrt{1+2^{2/3}}+2^{2/3}\sqrt{1+2^{2/3}}$

Out[70]= 4.16194

And so there we have it. A ladder of approximately 4.16 meters in length is the longest we may carry around the corner. We can see quite well from the plots above that this is clearly a local minimum, however, a quick second derivative test won't hurt.

In[71]:= `L´´[guess]`
        `% > 0`

Out[71]= $\sqrt{1+2^{2/3}}+2^{2/3}\sqrt{1+2^{2/3}}+2\big(1+2^{2/3}\big)^{3/2}$

Out[72]= True

In[73]:= `%% // N`

Out[73]= 12.4858

So there we are. A positive second derivative indicates that the graph is concave-up, which we already knew from our plot. Furthermore, a positive second derivative indicates a local minimum when it coincides with a critical point. Our answer of approximately 4.16 meters should be considered well and truly confirmed.

2.2.2 Integral Evaluation

Integral evaluation can, at times, be quite tricky. A tool such as *Mathematica* can indeed be an asset, however, even it may be unable to perform certain integrals symbolically. For example, suppose we ask *Mathematica* to integrate xe^{x^3} between 0 and 1.

In[74]:= `Integrate[x * E^(x^3), {x, 0, 1}]`

Out[74]= $\frac{1}{3}(-1)^{1/3}\left(-\text{Gamma}\left[\frac{2}{3}\right] + \text{Gamma}\left[\frac{2}{3}, -1\right]\right)$

Note that if the integrand were xe^{x^2}, then the task would be trifling easy by hand, however if we tried to use hand-methods to solve the above integral, we would very likely be frustrated. *Mathematica* has given us an answer involving the gamma function, which the reader may or may not find to be illuminating. Looking up `Gamma` in *Mathematica*'s Documentation Center we find that the gamma function, denoted mathematically by Γ is defined as follows

$$\Gamma(z) := \int_0^\infty t^{z-1} e^t \, dt$$
$$\Gamma(a, z) := \int_z^\infty t^{a-1} e^t \, dt$$

Again, this may or may not be illuminating. The only other thing we might try is to ask for a numerical answer

In[75]:= `N[%, 50]`

Out[75]= $0.78119703110865591510743281434829950577669739096218 + 0. \times 10^{-51} i$

That's interesting, we have a complex answer, although the complex part is infinitesimally small. Nonetheless, this shouldn't happen; a definite integral of a real-valued function should, itself, be a real number. It seems that *Mathematica* is perhaps being a little more general than we wanted it to be. We could guess that the real part of the answer, $0.78119703110865591510743281434829950577669739096218$, might be a good approximation of the integral, but this would just be a guess. Alternatively, we may ask *Mathematica* to perform the integration numerically, instead of symbolically, using the `NIntegrate` function.

In[76]:= `NIntegrate[x * E^(x^3), {x, 0, 1}, WorkingPrecision -> 50]`

Out[76]= $0.78119703110865591510743281434829950577669739096218$

It seems that the real-part of our complex answer was a good approximation of the integral after all.

We pause here to mention that, when we have a numeric value of an integral (or any other numeric value, for that matter), one approach we may take is to use online systems to try and find a probable symbolic (closed form) identity for the decimal number. Both the Inverse Symbolic Calculator[1], and Sloane's On-line Encyclopedia of Integer Sequences[2] are good resources. Unfortunately, neither one turns up anything with the above fifty digits, but it it worth remembering their existence. At least one of the integrals from Exercise 6 requires some form of inverse symbolic calculation to identify.

Moving on, suppose we wish to evaluate the integral

$$\int_0^\pi \frac{x \sin(x)}{1 + \cos^2(x)} \, dx$$

Our first attempt should be to see what *Mathematica* thinks.

---

[1] http://isc.carma.newcastle.edu.au/ at the time of writing

[2] http://oeis.org/ at the time of writing

In[77]:= `Integrate[x * Sin[x] / (1 + Cos[x]^2), {x, 0, Pi}]`

Out[77]= $\dfrac{\pi^2}{4}$

That was easy. We might, if we wish to check this, evaluate both the integral, as well as $\pi^2/4$ numerically, and subtract them to see how close they are. We'll do this to fifty digits of precision.

In[78]:= `NIntegrate[x * Sin[x] / (1 + Cos[x]^2), {x, 0, Pi},`
`    WorkingPrecision -> 50`
`]`
`% - N[Pi^2 / 4, 50]`

Out[78]= `2.4674011002723396547086227499690377838284248518102`

Out[79]= $0. \times 10^{-50}$

Of course, such a check is only interesting if the **NIntegrate** function uses a purely numerical approach to integration—using, say, some variant on a Riemann sum. It's possible, in principle, that **NIntegrate** might perform regular integration first (if it can), and produce a numerical approximation of that. Were this the case, then it's not at all surprising that the two values, above, coincide. Looking at the Documentation Center we see—under the More Information dropdown—that **NIntegrate** has a number of possible rules it may use, which include Riemann and trapezoidal rules. Furthermore, we see that the **N** function, when applied to the regular **Integrate** function will go and call **NIntegrate** function if the integral cannot be performed symbolically. This suggests somewhat strongly that *Mathematica*'s numerical integration does behave differently to its symbolic integration.

To explore this integral a little more, we may apply the following fact. It can be shown that

$$\int_0^\pi x f(\sin(x)) \, dx = \frac{\pi}{2} \int_0^\pi f(\sin(x)) \, dx$$

and it should be pretty clear that

$$\frac{\sin(x)}{1 + \cos^2(x)} = \frac{\sin(x)}{2 - \sin^2(x)} = f(sin(x)) \ \text{ where } f = \frac{x}{2 - x^2}$$

which leaves us with the slightly simpler integral

$$\int_0^\pi \frac{x \sin(x)}{1 + \cos^2(x)} \, dx = \frac{\pi}{2} \int_0^\pi \frac{\sin(x)}{1 + \cos^2(x)} \, dx$$

We should expect *Mathematica* to give the same answer of $\pi^2/4$ for this integral.

In[80]:= `Pi / 2 * Integrate[Sin[x] / (1 + Cos[x]^2), {x, 0, Pi}]`

Out[80]= $\dfrac{\pi^2}{4}$

Let us perform one more integral. This time we evaluate

$$\int_0^\infty \frac{x^2}{\sqrt{e^x - 1}} \, dx$$

As a first attempt, we see what *Mathematica* makes of the integral.

In[81]:= `Integrate[x^2/Sqrt[E^x-1], {x, 0, Infinity}]`

Out[81]= $\dfrac{\pi^3}{3} + \pi \operatorname{Log}[4]^2$

Having a closer look at the integral, such as we might do before we tried a hand calculation, we see a possible substitution we might perform. Looking at the denominator, $\sqrt{e^x - 1}$, a substitution of $x = \log(u)$ will change the denominator to $\sqrt{u - 1}$ thus eliminating the exponent term. Performing this substitution, we see that

$$\frac{dx}{du} = \frac{1}{u} \Rightarrow dx = \frac{du}{u}$$

It is also the case that $\log(0) = 1$ and $\log(x) \to \infty$ as $x \to \infty$ so that

$$\int_0^\infty \frac{x^2}{\sqrt{e^x - 1}}\, dx = \int_1^\infty \frac{\log(u)^2}{u\sqrt{u - 1}}\, du$$

Turning again to *Mathematica* with this new integral we see, as we should hope, the same answer as for the original integral.

In[82]:= `Integrate[Log[u]^2/(u*Sqrt[u-1]), {u, 1, Infinity}]`

Out[82]= $\dfrac{\pi^3}{3} + \pi \operatorname{Log}[4]^2$

This integral gives rise to an illustrative phenomenon if we evaluate it numerically. Specifically, evaluating it to ten digits gives a value of 16.37297602. Using the Inverse Symbolic Calculator with this value gives a probable identity as $16 + 1/\sqrt{6^{1/4} + 10^{3/4}}$. A competing product to *Mathematica* identifies[3] the same decimal value as $\sqrt{3} \cdot \zeta(5)^9 \cdot 9/2^{2/5}$. Both of these identifications differ from the answer provided by *Mathematica* above. It turns out that if we obtain even one more digit of precision, then neither system can identify the number anymore. This should shatter any confidence we had in either of those two values, and serves as a good reminder that inverse symbolic computation, while immensely useful, does not provide certain results.

We should take pains to stress that *Mathematica* is not a substitution for poor calculus skills (or, at least, it is a poor one). It is a good idea to check answers—especially answers from any inverse symbolic computation—from a number of avenues before accepting them. Human/machine collaboration is the name of the game here. *Mathematica* can be wonderful for performing tedious calculations quickly, and is remarkably adept at performing integral calculus, but a correct substitution, or other mathematical insight on our part might mean the difference between successfully obtaining a symbolic answer, or not.

2.2.3 Differential Equations

Differential equations are equations that relate a function to its derivatives. A solution to a differential equation is a function that has the required relationship with its derivatives. The simplest differential equation is $y' = y$ which has the solution $y = Ce^x$ (where C is an arbitrary constant). This should be nothing new to anybody who has studied first-year calculus.

---

[3] Identifies by means of a similar inverse symbolic calculation employed by the Inverse Symbolic Calculator

A first-order linear differential equation can always be re-written to have the form

$$y' + P(x)\, y = Q(x)$$

and can be solved by use of an *integrating factor*

$$I(x) := e^{\int P(x)\, dx}$$

which has the property that

$$\int \Big(I(x)\, y' + I(x)\, P(x)\, y \Big)\, dx = I(x)\, y$$

It is fairly simple to verify the claimed property of the integrating factor by hand. We check it in *Mathematica*. Note that it is understood, above, that y is a function of x. It is important to make this explicit when dealing with *Mathematica*.

```
In[83]:= Block[{P, I, y, x},
            I[x_] := E^Integrate[P[x], x];
            Integrate[I[x] * (y'[x] + P[x] * y[x]), x] == I[x] * y[x]
         ]
```

Out[83]= $\displaystyle\int e^{\int P[x]\, dx}\, (P[x]y[x] + y'[x])\ dx == e^{\int P[x]\, dx}\, y[x]$

```
In[84]:= Simplify[%]
```

Out[84]= $\displaystyle\int e^{\int P[x]\, dx}\, (P[x]y[x] + y'[x])\ dx == e^{\int P[x]\, dx}\, y[x]$

This is a relatively easy integral to do by hand, and the reader is actively encouraged to do so. Unfortunately, *Mathematica* seems to be struggling with it somewhat. Instead, we try the equivalent formulation, that

$$\frac{d}{dx}\, (I(x)\, y) = I(x) \cdot (y' + P(x)\, y)$$

which we know is true because of the fundamental theorem of calculus.

```
In[85]:= Block[{P, I, y, x},
            I[x_] := E^Integrate[P[x], x];
            D[Ix] * y[x] == I[x] * (y'[x] + P[x] * y[x])
         ]
```

Out[85]= $e^{\int P[x]\, dx}\, P[x]\, y[x] + e^{\int P[x]\, dx}\, y'[x] == e^{\int P[x]\, dx}\, (P[x]\, y[x] + y'[x])$

```
In[86]:= Simplify[%]
```

Out[86]= True

With this identity available to us, we may solve the equation by multiplying the left-hand and right-hand sides by the integrating factor, integrating both sides, and solving for y. The solution, therefore, to the differential equation is

$$y' + P(x)\, y = Q(x) \implies \int I(x)(y' + P(x)\, y)\, dx = \int I(x)Q(x)\, dx$$

$$\implies I(x)\, y = \int I(x)Q(x)\, dx$$

and so the solution is

$$y = \frac{\int I(x)\,Q(x)\,dx}{I(x)}$$

Let us now consider the linear differential equation $xy' + y = 3x^3$, which may be rewritten as $y' + x^{-1}y = 3x^2$; then we may use *Mathematica*

```
In[87]:= Block[{P, Q, I, x, C},
            P[x_] := 1 / x;
            Q[x_] := 3 x^2;
            I[x_] := E^Integrate[P[x], x];
            (Integrate[I[x] * Q[x], x] + C) / I[x]
         ]
```

$$\text{Out[87]=} \quad \frac{C + \frac{3x^4}{4}}{x}$$

Note, however, that for a general solution we needed to add manually the constant of integration when calculating the answer, because *Mathematica*'s **Integrate** function does not include this constant.

It is a good idea to check the solution by substituting it into the differential equation. In this case, we expect that if we calculate $y' + x^{-1}y$ we should get $3x^2$.

```
In[88]:= D[%, x] + % / x
```

$$\text{Out[88]=} \quad 3x^2$$

We may further cross-check this using *Mathematica*'s inbuilt differential equation solving function **DSolve**.

```
In[89]:= DSolve[y´[x] + y[x] / x == 3 x^2, y[x], x]
```

$$\text{Out[89]=} \quad \left\{\left\{y[x] \to \frac{3x^3}{4} + \frac{C[1]}{x}\right\}\right\}$$

Note that, in a manner similar to the **RSolve** function, we had to specify the differential equation itself, the function for which we were solving, and the independent variable, in that order.

Moving on now to second-order differential equations. A *second-order linear differential equation* is a differential equation of the form

$$P(x)\,y'' + Q(x)\,y' + R(x)\,y = G(x)$$

and is furthermore said to be *homogeneous* if $G(x) = 0$. Finally, if the functions $P(x)$, $Q(x)$, and $R(x)$ are constant functions then the differential equation is said to have *constant coefficients*; however, if the differential equation is still to be second-order then $P(x) \neq 0$.

Homogeneous second-order linear differential equations with constant coefficients may be solved in a manner almost identical to that used to solve homogeneous second-order linear recurrence relations with constant coefficients, which we looked at in Section 1.3.3.

Given the equation $ay'' + by' + c = 0$ we construct the *characteristic equation* $at^2 + bt + c = 0$ and solve for t. There are only three possibilities for the roots r_1, r_2 of the equation. The general formula is as follows.

$$y(x) = \begin{cases} Ae^{r_1 x} + Be^{r_2 x} & r_1 \neq r_2 \\ Ae^{r_1 x} + Bxe^{r_2 x} & r_1 = r_2 \\ e^{\alpha x}\left(A\sin(\beta x) + B\cos(\beta x)\right) & r_1, r_2 = \alpha \pm i\beta \end{cases}$$

where A and B are arbitrary constants.

Let's look at some examples. We start with a differential equation that has the same characteristic equation as the Fibonacci numbers, $y'' - y' - y = 0$. Because this has characteristic equation $t^2 - t - 1$, we know from Section 1.3.2 that the roots are $t = \frac{1}{2}(1 \pm \sqrt{5})$. As such, the solution to the differential equation should be

$$y = Ae^{x\cdot(1+\sqrt{5})/2} + Be^{x\cdot(1-\sqrt{5})/2}$$

We check this in *Mathematica*.

```
In[90]:= Block[{A, B, x, y},
            y[x_] := A * E^(x * (1 + Sqrt[5])/2) + B * E^(x * (1 - Sqrt[5])/2);
            y''[x] - y'[x] - y[x] ]
```

$$\text{Out[90]} = -Be^{\frac{1}{2}\left(1-\sqrt{5}\right)x} - \frac{1}{2}\left(1-\sqrt{5}\right)Be^{\frac{1}{2}\left(1-\sqrt{5}\right)x} + \frac{1}{4}\left(1-\sqrt{5}\right)^2 Be^{\frac{1}{2}\left(1-\sqrt{5}\right)x} - Ae^{\frac{1}{2}\left(1+\sqrt{5}\right)x} - \frac{1}{2}\left(1+\sqrt{5}\right)Ae^{\frac{1}{2}\left(1+\sqrt{5}\right)x} + \frac{1}{4}\left(1+\sqrt{5}\right)^2 Ae^{\frac{1}{2}\left(1+\sqrt{5}\right)x}$$

```
In[91]:= Simplify[%]
```

$$\text{Out[91]} = 0$$

And asking *Mathematica* for the solution directly also gives the expected result (which is always nice).

```
In[92]:= DSolve[y''[x] - y'[x] - y[x] == 0, y[x], x]
```

$$\text{Out[92]} = \left\{\left\{y[x] \to e^{\left(\frac{1}{2}-\frac{\sqrt{5}}{2}\right)x}C[1] + e^{\left(\frac{1}{2}+\frac{\sqrt{5}}{2}\right)x}C[2]\right\}\right\}$$

In the case of nonhomogeneous second-order linear differential equations with constant coefficients the solution is obtained by first calculating the solution to the equivalent homogeneous differential equation (essentially just pretending that $G(x) = 0$) and adding to that a *particular* solution.

To be more mathematically rigorous here, given the equation

$$ay'' + by' + cy = G(x)$$

the general solution is

$$y = y_p + y_c$$

where y_p is some (any) solution to the equation, and y_c is the solution to the complementary homogeneous equation

$$ay'' + by' + cy = 0$$

which is henceforth referred to only as the *complementary equation*.

Ordinarily in first-year courses the particular solution is obtained by a guess-and-check approach. A more systematic method named *variation of parameters* is described in [12], but tends to be trickier and slower to implement in practice, at least for the sorts of problems studied in first-year calculus.

We eschew the entire business, and go straight to the "just ask *Mathematica*" method. We extend our previous example to a nonhomogeneous problem where $G(x) = \sin x$.

In[93]:= `DSolve[y''[x] - y'[x] - y[x] == 0, y[x], x]`

Out[93]= $\left\{ \left\{ y[x] \rightarrow e^{\left(\frac{1}{2} - \frac{\sqrt{5}}{2}\right)x} C[1] + e^{\left(\frac{1}{2} + \frac{\sqrt{5}}{2}\right)x} C[2] - \frac{4(\text{Cos}[x] - 2\,\text{Sin}[x])}{(-5 + \sqrt{5})(5 + \sqrt{5})} \right\} \right\}$

We can clearly see our homogeneous solution there, and an extra bit at the end which is, presumably, a particular solution. We have a closer look at that. Firstly, it is in need of some simplification. A quick computation in our heads should tell us that the denominator will simplify to -20. As such, the expression can be simplified to something along the lines of $\frac{1}{5}(\cos x - 2\sin x)$.

In[94]:= `Simplify[-4 * (Cos[x] - 2 Sin[x]) / ((-5 + Sqrt[5]) * (5 + Sqrt[5]))]`

Out[94]= $\frac{1}{5}(\text{Cos}[x] - 2\sin[x])$

If this expression is a solution to the differential equation, as we suspect it to be, then substituting the expression into the differential equation should produce an outcome of $\sin x$.

In[95]:= `Block[{y, x},`
           `y[x_] := (Cos[x] - 2 * Sin[x]) / 5;`
           `y''[x] - y'[x] - y[x]`
         `]`

Out[95]= $\frac{1}{5}(2\,\text{Cos}[x] + \text{Sin}[x]) + \frac{2}{5}(-\text{Cos}[x] + 2\,\text{Sin}[x])$

In[96]:= `Simplify[%]`

Out[96]= $\text{Sin}[x]$

So it is clear that $\frac{1}{5}(\cos x - 2\sin x)$ is a solution to the original nonhomogeneous problem. It should be clear, using the fact that

$$\frac{d}{dx}(f + g) = \frac{d}{dx}(f) + \frac{d}{dx}(g)$$

that the entire expression returned by *Mathematica*'s `DSolve` function is also a solution to the equation as well.

One might well ask why, when we have a perfectly good and easy-to-write solution such as $\frac{1}{5}(\cos x - 2\sin x)$ would we ever want to bother with adding in the extra mess of the solution to the complementary equation as well. The answer is that $y_c + y_p$ is the *general* solution to the equation. That is to say that any and every solution to the equation can be written in that form. The proof of this fact is quite elementary, and can be found in [12] if desired. Certainly, the particular solution, above, was the same as the general solution with the constants $C[1] = C[2] = 0$.

2.2.4 Parametric Equations, Alternative co-ordinates, and Other Esoteric Plotting Fun

Recall that when a graph is plotted, what we are seeing is a graphical representation of pairs of points that satisfy some relationship. The parabola, for example, is the collection of all points (x, y) in $\mathbb{R} \times \mathbb{R}$ where $y = x^2$.

We may also plot functions from parametric equations, where a parameter, t say, varies, and the points in the plot are of the form $(x, y) = (x(t), y(t))$. There is not necessarily a relationship between the x and y co-ordinates in a parametric equation, apart from the fact that they share the same t-value. Of course, any function $f(x)$ may be turned into a parametric equation $(x, y) = (t, f(t))$.

If we wish to plot a parametric equation, then *Mathematica*'s **ParametricPlot** function will allow us to do so. The **ParametricPlot** function accepts a two-element list as its first argument. This elements of list is interpreted as $x(t)$ and $y(t)$ respectively where t is any valid *Mathematica* variable name, and $x(t)$ and $y(t)$ are arbitrary expressions involving that variable. Note that *Mathematica* will automatically scale the horizontal and vertical axes to fit the plot, based on the values of $x(t)$ and $y(t)$ as t varies through its range.

For example, to plot our parabola using parametric equations, we would input the following.

In[97]:= **ParametricPlot[{t, t^2}, {t, -10, 10}]**

Out[97]=
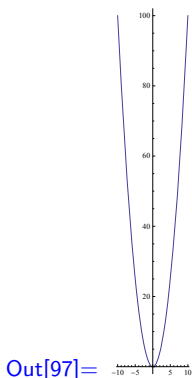
Note that the **ParametricPlot** function has produced a picture which is tall and thin. It appears as if the function has used the same scale for both horizontal and vertical axes, and has chosen the dimensions of the image accordingly. We can change this behavior with the **AspectRatio** option. The aspect ratio is the ratio of the height and width of an image[4]. Looking at the Documentation Center, we find that the default **AspectRatio** for the **Plot** function, is **1/GoldenRatio**, so we use that value.

In[98]:= **ParametricPlot[{t, t^2}, {t, -10, 10},**
          AspectRatio -> 1/GoldenRatio
          ]

---

[4] Most modern television sets have an aspect ratio of 16:9. Note that a television aspect ratio is width:height, whereas *Mathematica*'s **AspectRatio** option expects a ratio of height:width

Out[98]=
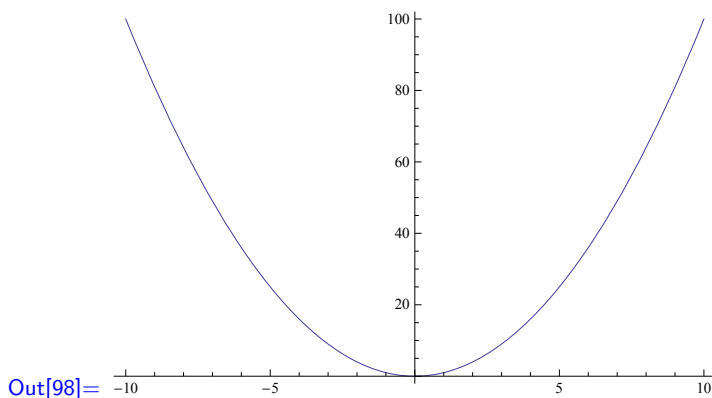
Let's look at something that parametric equations allow us to plot, that the regular **Plot** command cannot do. To this end recall the parametric equations of a circle centered at an arbitrary point, (x_0, y_0) say, are

$$(x, y) = (x_0, y_0) + (r \cos \theta, r \sin \theta) = (x_0 + r \cos \theta, y_0 + r \sin \theta)$$

This is easiest to see by treating these parametric equations as vector equations, and recalling that the parametric equations of a circle centered at the origin are $(x, y) = (r \cos \theta, r \sin \theta)$.

We plot a circle, centered at $(1, 2)$, with radius 3.

In[99]:= **ParametricPlot[{1 + 3 Cos[t], 2 + 3 Sin[t]}, {t, 0, 2 Pi}]**

Out[99]=
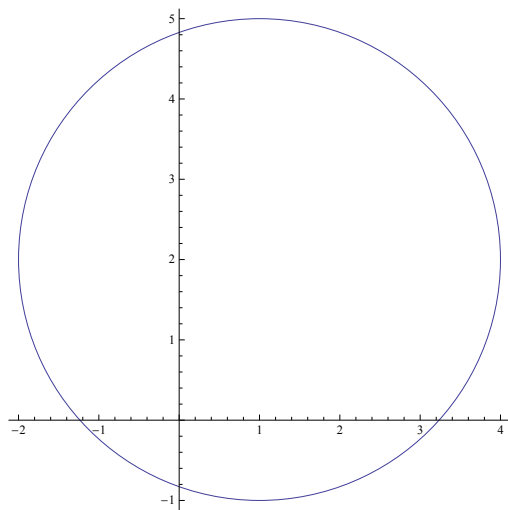
Inasmuch as we knew both the center and the radius of the circle, we should have expected the horizontal range to be $(-2, 4)$ and the vertical range to be $(-1, 5)$, which was precisely what we saw. However, we should point out at that the axis ranges are independent of the parameter range, and all three may be modified separately. Ordinarily, *Mathematica* will calculate the axis ranges based on the values the parametric equation takes over the parameter range. In order to demonstrate this independence of axis and parameter ranges, we plot the same circle, but this time specifically instruct *Mathematica* to plot for the range $-3 \le x \le 5$ and $-2 \le y \le 6$.

In[100]:= **ParametricPlot[{1 + 3 Cos[t], 2 + 3 Sin[t]}, {t, 0, 2 Pi},**
        PlotRange->{{-3, 5}, {-2, 6}}
    ]
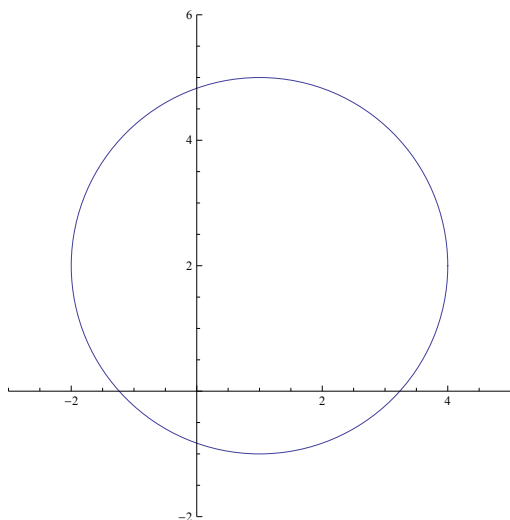
Out[100]=

We can clearly see the same circle circle as before, happily plotted within the larger ranges as defined in the **PlotRange** option. Note that in this case we specified both the horizontal range as well as the vertical range, whereas previously we used **PlotRange** to only specify a vertical range. If the range we specify is a list containing two lists each of which has two elements, it is understood that the first two-element list is the horizontal range, and the second two-element list is the vertical range.

In a lot of (if not most) cases separately modifying the horizontal, vertical, and parameter ranges will not be necessary. Nonetheless, it is worth being aware of the fact that they may be independently specified.

The circle example demonstrates a very useful feature of parametric plots, which is that they may be used to plot curves that are not the result of functions. The circle is clearly not a function, as it violates the vertical line test. Recall that a function always associates a single value in the range with each single value in the domain. This is not the case with a circle; we cannot assign a function $y = f(x)$ that will produce all the points in a circle. We can, of course, simply plot $y = \pm\sqrt{1 - x^2}$ and display them together, but doing this is often neither easy nor even possible.

To further illustrate this advantage of parametric equations, we plot a spiral using parametric equations. We use the parametric equations $(x, y) = (t \cos t, t \sin t)$. This differs from the circle in that the radius is no longer fixed. If we think of the points (x, y) as vectors, then each point on the curve will be a vector of length t and angle t. As t increases, then the angle will cycle, but the length will continue increasing. We plot four full revolutions of this spiral.

In[101]:= **ParametricPlot[{t * Cos[t], t * Sin[t]}, {t, 0, 8 Pi}]**
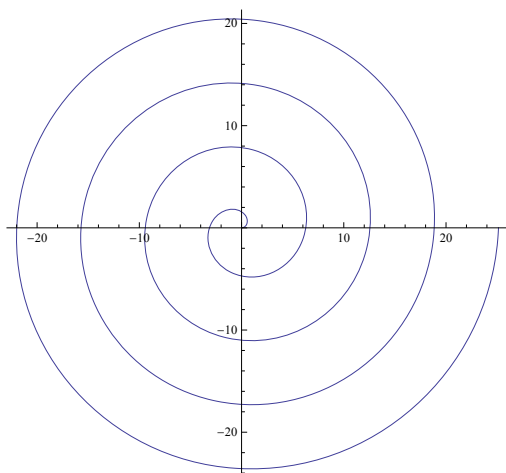
Out[101]=

Returning to our circle plots, the reader may well recall that although there is no explicit function for a circle, there most certainly is an equation that gives an implicit function for the circle. That equation is, of course, $(x-x_0)^2+(y-y_0)^2 = r^2$ where (x_0, y_0) is the center of the circle and r is the radius. One may well wonder if *Mathematica* can plot such implicit equations. The answer is that yes it can, thanks to the `ContourPlot` function.

In[102]:= `ContourPlot[(x - 1)^2 + (y - 2)^2 == 9, {x, -2, 4}, {y, -1, 5}]`
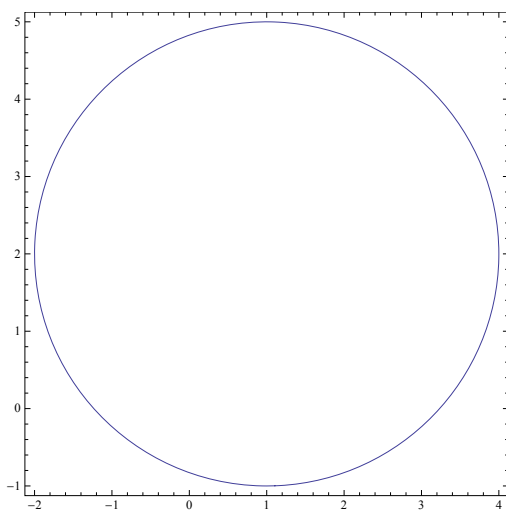
Out[102]=

Ordinarily, the `ContourPlot` function is used for plotting contours of surfaces that results from functions of two variables (see Section **??**). However, it is also the go-to function for plotting implicit functions as well, and supersedes an older function for implicit function plotting.

So far we have considered only Cartesian co-ordinates (of the form (x, y) in relation to two axes) when plotting. Another common co-ordinate system is the so-called *polar* co-ordinates, which are co-ordinates of the form (r, θ) where θ is the angle, and r is the distance traveled in that direction. The polar co-ordinates $(\sqrt{2}, \frac{1}{4}\pi)$, for example, correspond to the Cartesian co-ordinates $(1, 1)$. We may freely convert between polar and Cartesian co-ordinates with the identities $r = \sqrt{x^2 + y^2}$, $x = r\cos\theta$, and $y = r\sin\theta$. These identities may easily be confirmed by drawing up a trigonometric triangle.

Mathematica will happily plot polar equations (i.e., equations given in terms of polar co-ordinates) for us using the `PolarPlot` function. We may plot on regular Cartesian pair of axes, or on a special background more suited to the polar co-ordinates. The latter requires some rather particular options to be specified. Polar plots expect an expression for r to be a function of θ (just as regular plots expect an expression for y as a function of x). This should be unsurprising, given that polar equations are usually written as $r = r(\theta)$.

Let us start with a circle, as it is quite simple. A circle contains points that are equidistant from its center, hence r is the constant radius, and θ may take any value. So our polar equation is simply $r = C$ where C is a constant. Such a circle, however, will always be centered at the origin. Plotting, in polar co-ordinates, a circle not centered at the origin is trickier. See Exercise 9b.

Let us look at the spiral again. The spiral construction with parametric equations $(x, y) = (t \cos t, t \sin t)$ was constructed in a way that is very amenable to a polar equation. Recall that our t parameter varied as both an angle and a distance. This sounds very much like a polar equation. It should come as no surprise then that the polar equation of the spiral is simply $r = \theta$. To plot this, we simply pass the parameter *coords=polar* to the `Plot` function.

In[103]:= `PolarPlot[theta, {theta, 0, 8 Pi}]`
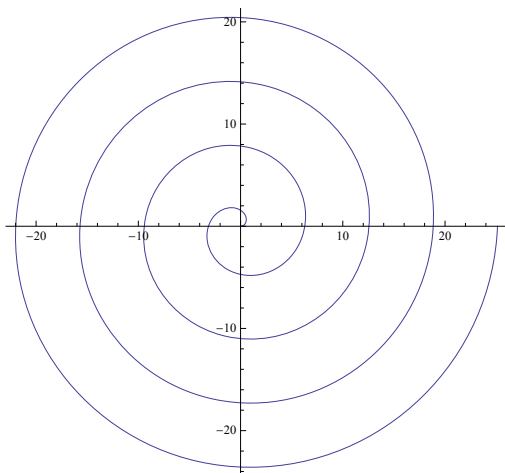
Out[103]=

Unfortunately, here, the axes are drawn and labelled as they would be for the cartesian co-ordinate system; we have an x-axis and a y-axis. This is not particularly helpful for identifying which polar co-ordinates correspond to any given point on the curve. If we want to avoid this problem, and correctly label the r and θ parameters of the polar coordinate system, we specify the `PolarAxes` and `PolarGridLines` options, setting them both to `Automatic`. Doing so will show the r-values as circles centred at the origin, and the θ values as radial line. These markings allow us to much more easily read the coordinates directly from the plot in much the same way we might read (x, y) value pairs from a plot on the plane.

In[104]:= `PolarPlot[theta, {theta, 0, 8 Pi},`
            `PolarAxes -> Automatic,`
             `PolarGridLines->Automatic`
           `]`
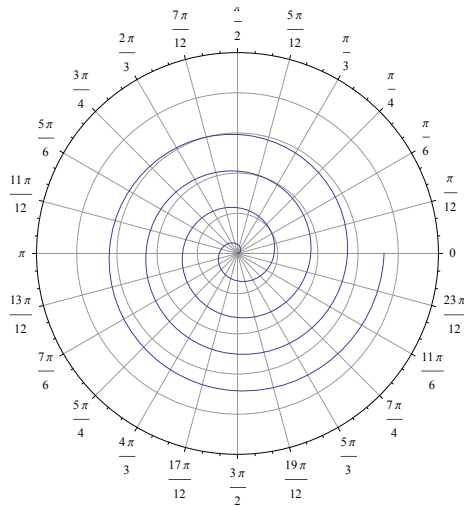
Out[104]=

For a more interesting and complicated example, we now look at a more complicated polar plot; $r = \sin(8\theta/5)$. This is clearly a cyclic equation, however the precise nature of the cycling needs some thought. It is clear that r will cycle every time θ changes through $5\pi/4$ radians. In order to cycle fully through all possible pairs of (r, θ) we need to find integers a, b such that $a \cdot 2\pi = b \cdot 5\pi/4$ and the smallest such value for a is $a = 5$, corresponding to $b = 8$. We therefore plot this function for $0 \leq \theta \leq 10\pi$.

In[105]:= **PolarPlot[Sin[8 * theta / 5], {theta, 0, 10 Pi}]**
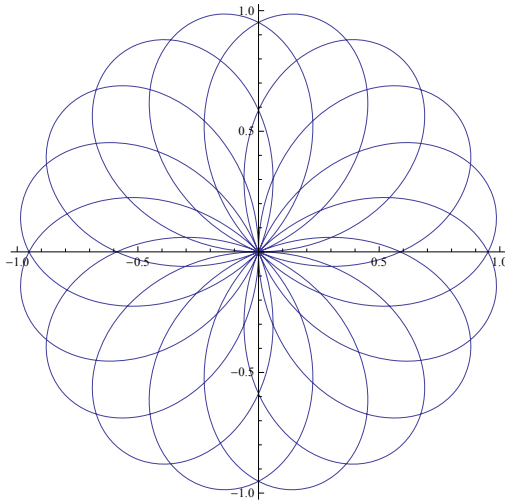
Out[105]=

Finally, we look at plotting ranges of inequalities. If we have an inequality, and wish to plot it, we use the **RegionPlot** function. We look at a couple of simple examples to close this section. The general approach is similar to that used for the **ContourPlot** function, and is basically as one would expect; we replace the equation to be plotted with the inequality. For instance, if we wish to see the inequality $4x - 2y < 3$, we would try the following.

In[106]:= **RegionPlot[4 x - 2 y < 3, {x, -1, 1}, {y, -1, 1}]**

Out[106]=
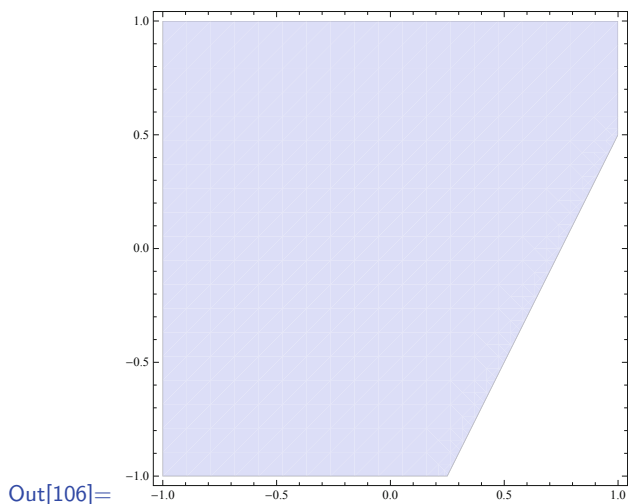
To close off this section, we plot our familiar radius-3 circle, centered at $(1, 2)$ with a red interior, and a thick black boundary.

```
In[107]:= RegionPlot[(x - 1)^2 + (y - 2)^2 <= 9, {x, -2, 4}, {y, -1, 5},
          PlotStyle -> Red, BoundaryStyle -> Thick
        ]
```

Out[107]=
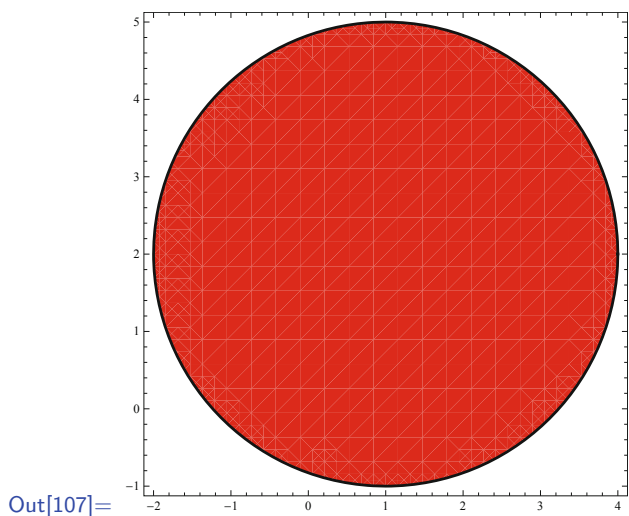
2.3 Multivariate Calculus

2.3.1 Three-Dimensional Plotting

The **Plot** function, which is for two-dimensional plots, has a counterpart named **Plot3D** which is rather unsurprisingly for three-dimensional plots. Ordinarily, **Plot3D** will plot a function $z = f(x, y)$. To plot, for example, the paraboloid $z = x^2 + y^2$ we simply input the following.

```
In[108]:= Plot3D[x^2 + y^2, {x, -1, 1}, {y, -1, 1}]
```
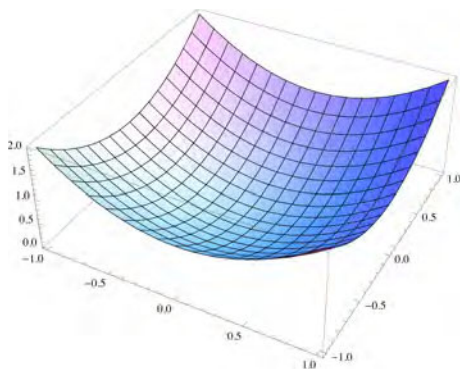
Out[108]=

Note that we needed two iterators for the `Plot3D`function—one for each of the variables in the two-variable expression we were plotting. Also be aware that 3-D plots default to having the z-axis pointing "up" (which is to say up with respect to the screen), unlike 2-D plots which have the y-axis pointing up. An advantage to these plots is that they may be rotated by dragging them with the mouse, which allows a better appreciation of the overall three-dimensional shape, even though we ultimately only have a two-dimensional projection on our computer screen. Of course, such rotation will almost always change which, if any, axis is pointing up with respect to the screen.

Just as with two-dimensional plots, we may parameterize surfaces and may plot them with the `ParametricPlot3D` function. In much the same way that our `ParametricPlot` accepted a two-element list for its first argument, `ParametricPlot3D` takes a three-element list as its first argument. The three-element list is interpreted to be the parametric expressions for points (x, y, z).

```
In[109]:= ParametricPlot3D[{v * Cos[u], v * Sin[u], v},
          {u, 0, 2 Pi}, {v, -1, 1}
          ]
```
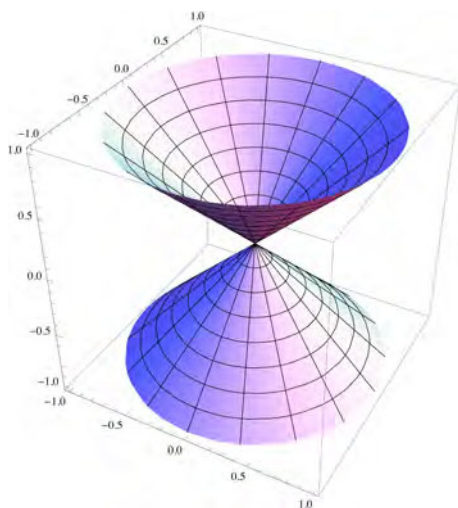

Out[109]=

Three-dimensional plots are usually—although not always—parameterized with two parameters. The trivial parameterization, for a function $f(x, y)$ is simply $(u, v, f(u, v))$, and should look quite similar to the trivial parameterization of a function of one variable.

We look quickly at two alternative co-ordinate systems for three-dimensional surfaces. These are *cylindrical* and *spherical* co-ordinates.

Cylindrical co-ordinates are an extension of polar co-ordinates. Any point in 3-space may be located by specifying an angle on the xy-plane, and a distance from the origin at that angle (which is identical to polar co-ordinates), and finally by a height above (or below) the xy-plane. As such a point in cylindrical co-ordinates is of the form (r, θ, z).

In order to plot surfaces given in terms of cylindrical coordinates, we must use the `RevolutionPlot3D` function. This function is designed for drawing surfaces of revolution (see Section 2.3.2), however we may also use it for cylindrical coordinate plotting. The cone (which we plotted above with parametric cartesian equations of $(v \cos u, v \sin u, v)$) may be expressed, in cylindrical coordinates, by the function $z = r$.

In[110]:= `RevolutionPlot3D[r, {r, -1, 1}]`
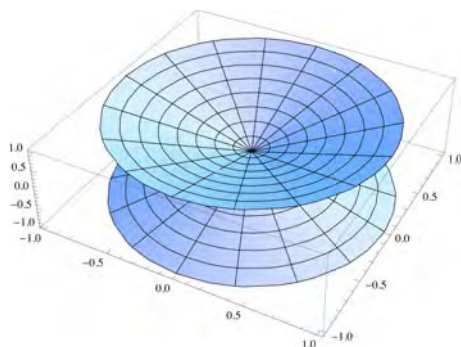
Out[110]=

Note here that we did not need to specify an iterator for θ. This is due to two factors acting in concert. Firstly, our expression did not involve θ at all. Secondly, `RevolutionPlot3D` will, by default, apply the rotation angle through a range of 0 to 2π. We may specify a different rotation angle by providing a second iterator, which will be understood to be the angle, regardless of which name we give it.

In[111]:= `RevolutionPlot3D[r, {r, -1, 1}, {bob, 0, 3 Pi / 2}]`
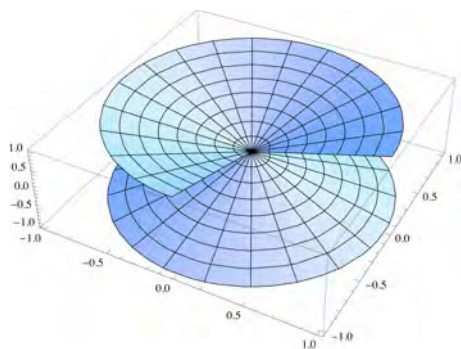
Out[111]=

Spherical co-ordinates are similar to cylindrical, differing only in the final co-ordinate. As with cylindrical co-ordinates, we have an r, θ pair that locates a point on the xy-plane. We then rotate that point *vertically* by an angle $\phi \in [0..\pi]$ where the angle is measured against the z-axis, with 0 pointing upwards, and π pointing downward. Thus a point in 3-space in spherical co-ordinates has the form (r, θ, ϕ). Our r co-ordinate in this system will always be the distance of a point from the origin (note that this is not the case with cylindrical co-ordinates). As the name might suggest, this co-ordinate system is very well suited to locating points on a sphere. Indeed, anybody familiar with longitude and latitude on maps of earth will have seen this concept before.

Unlike cylindrical coordinates, there is a function specifically for expressions given in terms of spherical co-ordinates named **SphericalPlot3D**. This function expects its first argument to be an expression for r to be as a function of θ and ϕ. For example, a sphere of radius 1 has the simple cylindrical expression $r = 1$.

In[112]:= `SphericalPlot3D[1, {theta, 0, 2 Pi}, {phi, 0, Pi}]`
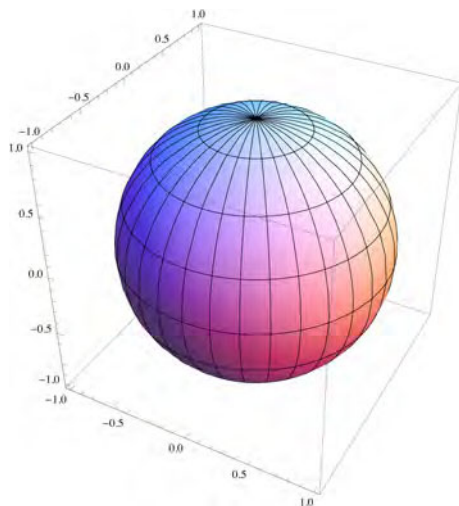
Out[112]=

We may also describe the sphere in terms of implicit functions. As such we may plot it using the **ContourPlot3D** function, which is a three-dimensional analogue of the **ContourPlot** function from Section 2.2.4. As one might expect, the three-dimensional version takes an expression in three variables, as well as three iterators.

In[113]:= `ContourPlot3D[x^2 + y^2 + z^2 == 1,`
         `{x, -1, 1}, {y, -1, 1}, {z, -1, 1}`
         `]`
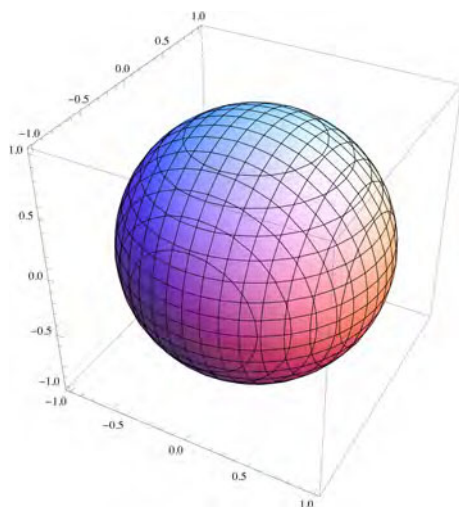
Out[113]=

Note that if we want to perform parametric plots of functions in either cylindrical or spherical coordinates, we will need to convert them back to cartesian coordinate parametric equations, and use the **ParametricPlot3D**. In fact, this is essentially what

SphericalPlot3D already does. The interested reader is encouraged to check out *Mathematica*'s Documentation Center for more information on any of the plotting functions, described above.

2.3.2 Surfaces and Volumes of Rotation

In general, calculating volumes is usually a job for iterated integrals (see Section 2.3.4). However, volumes of solids of revolution may be calculated with a single integral. A solid of revolution is produced by rotating a curve in the plane about a line. Usually, but not always, one of the axes is chosen. As an example, let us consider the sine curve between 0 and π.

In[114]:= `Plot[Sin[x], {x, 0, Pi}, AspectRatio -> Automatic]`
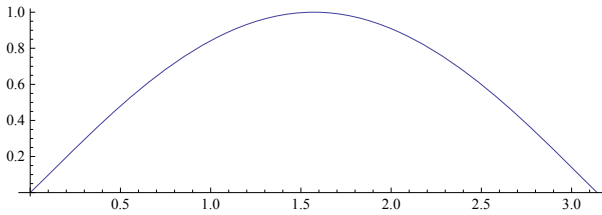
Out[114]=

If we rotate the above sine curve about the x-axis we can imagine a sort of symmetrical teardrop shape. In fact, we can do better than imagine. We can have *Mathematica* draw us a picture. We have already seen the **RevolutionPlot3D** function, and noted that it is for plotting surfaces of revolution. However, this function produces surfaces by rotating a curve around the vertical axis, whereas here we are rotating around the horizontal axis. Instead, it is simplest to use a parametric equation.

To this end, notice that at any point $x \in [0, \pi]$, our rotated surface will have a cross-section, parallel to the yz-plane which is a circle or radius $\sin(x)$. We may, therefore, parameterize the surface of the rotated surface by

$$[x, y, z] = [t, \sin(t)\sin(\theta), \sin(t)\cos(\theta)]$$

where $t \in [0, \pi]$ and $\theta \in [0, 2\pi)$. This we may now plot.

In[115]:= `ParametricPlot3d[{t, Sin[t] * Sin[theta], Sin[t] * Cos[theta]},`
          `{t, 0, Pi}, {theta, 0, 2 Pi}`
          `]`
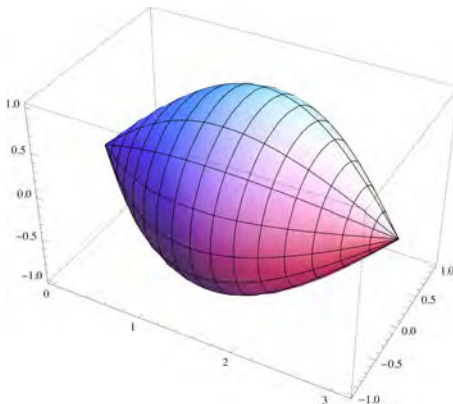
Out[115]=

The parameterization of this surface gives us a hint as to how to use integration to calculate the volume. We think of the volume as an infinite number of infinitely small disks (otherwise known as "circles"), and add up the area of each circle over an interval, the interval being $[0..\pi]$ in this case. The accumulated area is the volume. This is, incidentally, identical to a Riemann sum where we (more or less) add up the height of an infinite number of infinitely small boxes (otherwise known as "lines") and accumulate these heights over an interval to obtain an area. The disks we are calculating are perpendicular to the axis of rotation.

We know that the area of any particular disk is πr^2, and because our radius is $\sin(x)$, each disk in our particular example will have area $A(x) = \pi \sin(x)^2$. So the area, remembering we have only rotated the portion of the sine curve between 0 and π, is

$$\int_0^\pi A(x)\, dx = \int_0^\pi \pi \sin(x)^2\, dx = \pi \int_0^\pi \sin(x)^2\, dx$$

This we may now calculate in *Mathematica* or manually, as we see fit.

In[116]:= `Pi * Integrate[Sin[x]^2, {x, 0, Pi}]`

Out[116]= $\dfrac{\pi^2}{2}$

In general, the volume of a solid produced by rotating a function $f(x)$ inside an interval $[a, b]$ around the x-axis is given by the integral

$$\pi \int_a^b f(x)^2\, dx$$

If we wish to rotate a function $f(x)$ around the y-axis, then we need to rewrite the function as a function of y instead. Note that this is more complicated than just replacing every x in the function with a y. For example, let us rotate the parabola $y = x^2$ around the y-axis for $x \in [0, 2]$.

We start by plotting what we want to see. First, however, remember that with a three-dimensional plot it is the z-axis which is pointing upwards, but for a two-dimensional plot it is the y-axis that is pointing upwards. This is easily fixed by just renaming the y-axis to be the z-axis instead, giving us our "new" function of $z = x^2$, being rotated around the z-axis. For the remainder of this section, and any time we discuss volumes of revolution, we consider the 2-D plots to be in the xz-plane.

We are rotating a curve that sits in the xy-plane around the z-axis. *Mathematica* provides a function specifically for plotting such things, called `RevolutionPlot3D`. We simply give the function the same expression we would use to plot the curve in two dimensions, and it takes care of the rest.

In[117]:= `RevolutionPlot3D[r^2, {r, 0, 2}]`
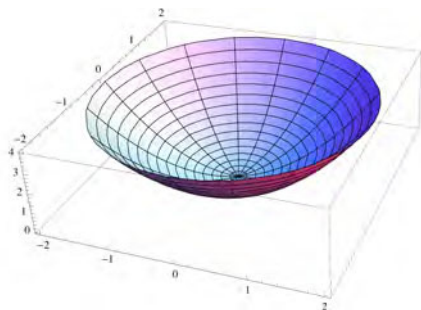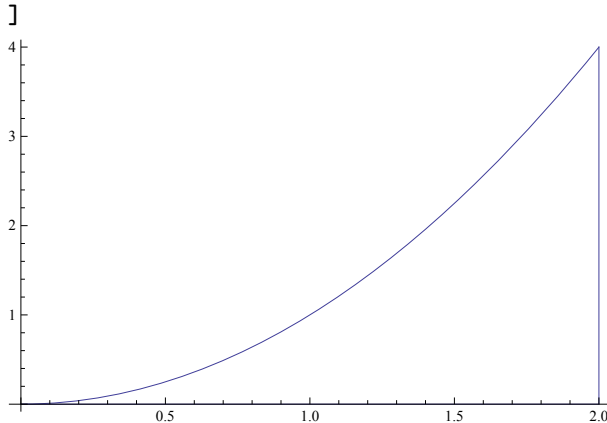
Out[117]=

Straight away we have an issue (although it is not immediately obvious). It is not clear from this plot whether we mean the volume between the paraboloid and the z-axis, or the volume "below" the paraboloid, down to the xy-plane. In fact we meant the latter, which is troublesome to visualize on that plot.

We remedy matters here. It is useful to be explicit about precisely which area we are rotating around the z-axis. In this case, we wish to take the volume under the curve $z = x^2$, but above the x-axis, and between the values $x = 0$ and $x = 2$. By plotting these bounds in two dimensions we should make it clearer exactly which area will be rotated.

```
In[118]:= Block[{p, x, z},
            P[1] = ParametricPlot[{x, x^2}, {x, 0, 2}]
            P[2] = ParametricPlot[{x, 0}, {x, 0, 2}]
            P[3] = ParametricPlot[{2, x^2}, {x, 0, 4}]
            Show[P[1], P[2], P[3]]
          ]
```

Out[118]=

We can't see the line that forms the lower bound, because the x-axis is obscuring it, nonetheless it is quite clear now exactly which area is being rotated. We plot the volume of revolution now, and again—just as with the 2-D plot above—we include the bounds as well. The line where $x = 2$ becomes a cylinder when rotated, and the line at the bottom of the plot becomes a disk. We are also cunning and leave a part of the solid open, so that we can see inside for a better impression of the solid in question.

```
In[119]:= RevolutionPlot3D[{{x, x^2}, {x, 0}, {2, x^2}}
            {x, 0, 2}, {theta, 0, 5 Pi / 3}
          ]
```

Out[119]=
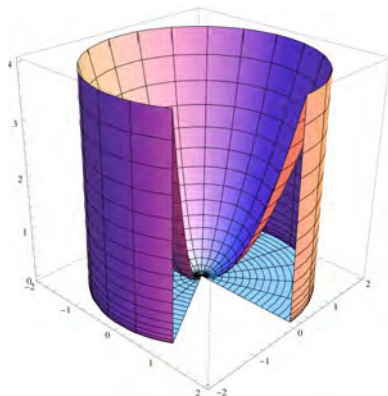
Note that instead of a list of expressions, we have instead used two-dimensional parametric equations in the **RevolutionPlot3D** command, above. The function has no trouble with this, and simply takes the two-dimensional curve described by these parametric equations, and rotates them around the vertical axis. Be careful here, these are cartesian parametric equations rotated around an axis. They are not cylindrical parametric equations. Unfortunately, the **RevolutionPlot3D** function will not plot cylindrical parametric equations.

Now in order to compute the volume using our integral, above, we need to be rotating around the same axis as the independent variable of our function. Here, however, we are rotating a function of x around the z-axis (still using the z-axis as the upward pointing one). We need to rewrite our function as $x = f(z)$. Rearranging $z = x^2$ in this manner produces $x = \sqrt{z}$. We also notice that $x = 0 \implies z = 0$ and $x = 2 \implies z = 4$, so we may equivalently consider our rotation as rotating the function $x = \sqrt{z}$ for $z \in [0, 4]$ around the z-axis. However, it is the area of the function above $x = \sqrt{z}$ that we are rotating around the z-axis. The upper bound for this function is the line $x = 2$.

In[120]:= `Plot[{Sqrt[z], 2}, {z, 0, 4}]`
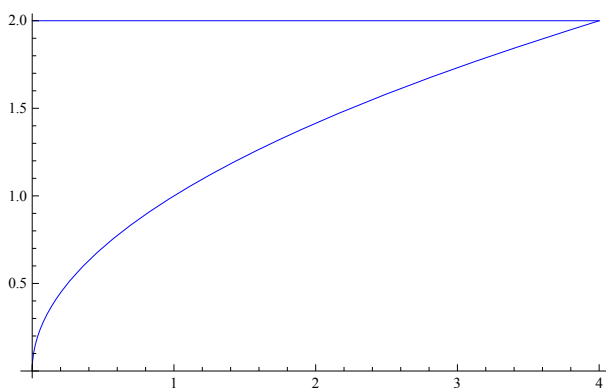
Out[120]=

Keep in mind that we are now rotating around the horizontal axis. So, what we actually want is the area between the two curves $x = 2$ and $x = \sqrt{z}$ for $0 \le z \le 4$ to be rotated. That's easy. Observe that $x = 2$ is always larger than $x = \sqrt{z}$ on the interval in question. For the area itself, we could happily calculate the integral of the larger minus the integral of the smaller. We may take the same approach with the volume integral. If we calculate the volume of the cylinder we obtain by rotating $x = 2$ around the z-axis, and subtract from that the volume of the paraboloid (obtained by rotating the volume under the function $x = \sqrt{z}$) then we have calculated the precise area we wanted. This is demonstrated in Figure 2.2.

In short, we wish to calculate

$$\pi \int_0^4 2^2 \, dz - \pi \int_0^4 \sqrt{z}^2 \, dz = \pi \int_0^4 4 - z \, dz$$

In[121]:= `Pi * Integrate[4 - z, {z, 0, 4}]`

Out[121]= 8π

Let us return now to our sine function, plotted between $x = 0$ and $x = \pi$. We now rotate this around the z-axis (treating z as the axis pointing "up" again, as we did with the last example). We start as we have each other time, with plotting the surface to see with what we're dealing.
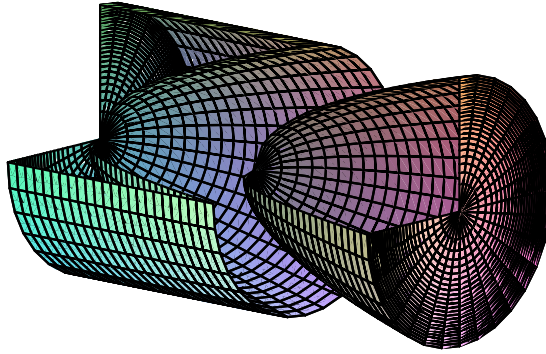
Fig. 2.2 Cylinder with paraboloid removed.

```
In[122]:= RevolutionPlot3D[{{x, Sin[x]}, {x, 0}},
            {x, 0, Pi}, {theta, 0, 5 Pi / 3}
          ]
```
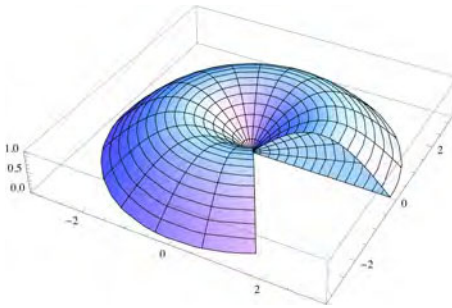

Out[122]=

Note the parametric equations used in the above plot. We also used parametric equations when plotting the paraboloid volume earlier. The astute reader may realize that we might have avoided parametric equations, since both $z = sin(x)$ and $z = 0$ are regular functions. We might have tried the following.

```
In[123]:= RevolutionPlot3D[{Sin[x], 0},
            {x, 0, Pi}, {theta, 0, 5 Pi / 3}
          ]
```
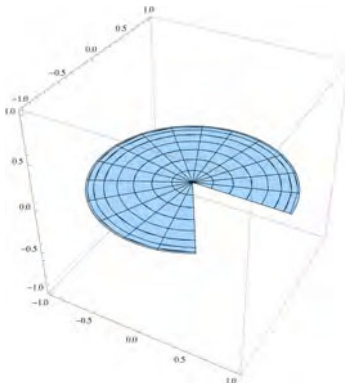

Out[123]=

All we got was the disc, and not the sine curve. This is because the expression `{Sin[x], 0}` was interpreted as a single cartesian parametric equation. The x-coordinate is equal to $\sin x$, and the y-coordinate is equal to 0, with the parameter x varying from 0 to π. In other words, we have described a line segment on the x-axis between $x = 0$ and $x = 1$. When rotated around the vertical axis by the `RevolutionPlot3D` function, we have a disc. In order to avoid this problem, we instead, above, used parametric equations to describe both curves we wished to rotate, and there was no confusion on the part of *Mathematica* with regards to what we wished drawn.

Back to the volume we wish to calculate, we have a problem inasmuch as there's no particularly easy or, at least, obvious way to rewrite $z = \sin(x)$ as a function of z. The z-interval is clearly $[0, 1]$, however, for any given value of z there are two values of x. This is easily seen with a plot.

In[124]:= `Plot[{Sin[x], 0.9}, {x, 0, Pi}, AspectRatio -> Automatic]`
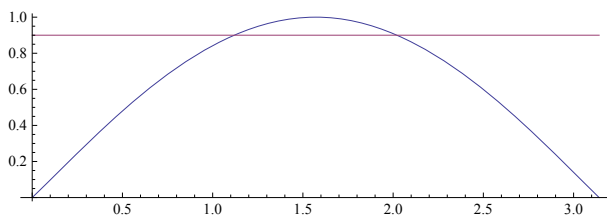
Out[124]=

The line $z = 0.9$ clearly cuts the sine function in two places.

Now, we may try and express the solid of rotation as an area between two functions, $f(z), g(z)$ and calculate the integral accordingly. However finding these two functions is tedious and time consuming[5]. Instead, we use a slightly different integral to calculate the area. Our previous method used disks that were perpendicular to the axis of rotation. This time we use a method known as "shells". To do this, we approximate the area as infinitely many cylinders, centered at the origin, and with radius x and height $\sin(x)$. The surface area of each cylinder is the circumference multiplied by the height. As such, the area function is $A(x) = 2\pi x \sin(x)$, and our volume may now be calculated as

$$\int_0^\pi A(x)\,dx = \int_0^\pi 2\pi x \sin(x)\,dx = 2\pi \int_0^\pi x \sin(x)\,dx$$

In[125]:= `2 * Pi * Integrate[x * Sin[x], {x, 0, Pi}]`

Out[125]= $2\pi^2$

And, in general, if we rotate a function $y = f(x)$ between $x = a$ and $x = b$ around the y-axis then the volume of the solid of revolution is given by the integral

$$2\pi \int_a^b x f(x)\,dx$$

And we always have the possibility of changing between these two integrals, by rewriting the function and interchanging the dependent and independent variables if the integration in one method proves too troublesome.

We may check our paraboloid volume calculation using this method.

_____

[5] The astute reader may notice the symmetrical nature of the sin function, and might try computing the volume over the interval $0 \le x \le \pi/2$, and doubling it. This is an admirable approach, but does not lead to us introducing the shells method integral.

```
In[126]:= 2 * Pi * Integrate[x * x^2, {x, 0, 2}]
```
Out[126]= $8\,\pi$

2.3.3 Partial and Directional Derivatives

Recall that for a function of two or more variables, the derivatives are taken with respect to one of the variables at a time. These are known as *partial derivatives*. There are several ways to denote partial derivatives, of which we use the following two. Let $f : \mathbb{R}^2 \to \mathbb{R}$. Then the first partial derivative of f with respect to x is denoted by

$$f_x \quad \text{or} \quad \frac{\partial f}{\partial x}$$

and the first partial derivative of f with respect to y is denoted by

$$f_y \quad \text{or} \quad \frac{\partial f}{\partial y}$$

Both of these are functions ($\mathbb{R}^2 \to \mathbb{R}$) in their own right.

Note that, if we do not have a name for our function, we may write the function in brackets after the ∂ notation. For example,

$$\frac{\partial}{\partial x} \left(\frac{x^2 + y^2}{x^2 - y^2} \right)$$

would be the first partial derivative with respect to x of the rational polynomial $(x^2 + y^2)/(x^2 - y^2)$.

If we take the derivative of one of these derivatives, then we again may take a partial derivative. By doing so we obtain a *second partial derivative*. The variable with respect to which the second derivative is taken may be different from that with respect to which the first derivative was taken (because a first partial derivative is still a valid function in its own right). The four possibilities for a second partial derivative are as follows.

$$f_{x,x} \quad \text{or} \quad \frac{\partial^2 f}{\partial x^2} \qquad\qquad\qquad f_{x,y} \quad \text{or} \quad \frac{\partial^2 f}{\partial y \partial x}$$

$$f_{y,x} \quad \text{or} \quad \frac{\partial^2 f}{\partial x \partial y} \qquad\qquad\qquad f_{y,y} \quad \text{or} \quad \frac{\partial^2 f}{\partial y^2}$$

In general if we have a function that is an nth partial derivative, which we then take yet another partial derivative of, then we have the following scenario.

$$(f_{x_1,\ldots,x_n})_{x_{n+1}} = f_{x_1,\ldots,x_{n+1}} \quad \text{or} \quad \frac{\partial}{\partial x_{n+1}} \left(\frac{\partial^n}{\partial x_n \cdots \partial x_1} \right) = \frac{\partial^{n+1}}{\partial x_{n+1} \cdots \partial x_1}$$

Note that the above also demonstrates why the ∂ notation has the variables written backwards.

Like standard derivatives, partial derivatives are defined in terms of the limit of the slopes of a series of lines between the point at which we wish to calculate the derivative, and another point near to it, as follows.

$$f_x(x, y) := \lim_{h \to 0} \frac{f(x + h, y) - f(x, y)}{h} \ \text{ or } \ f_y(x, y) := \lim_{h \to 0} \frac{f(x, y + h) - f(x, y)}{h}$$

Mathematica may perform partial derivatives. Indeed, the `D` function we have already used for regular differentiation will happily perform partial differentiation. In fact, if we look it up in the Documentation Center we are told that the very purpose of `D` is to compute partial derivatives. Note that in the case of a single variable function, there is no distinction between a partial derivative, and a For a first partial derivative, we tell `Diff` with respect to which variable we want to take the derivative.

```
In[127]:= With[{p = Sum[x^i * y^j, {i, 0, 2}, {j, 0, 2}]},
          p // Print;
          D[p, x] // Print;
          D[p, y] // Print;
          ]
```

Out[127]= $1 + x + x^2 + y + xy + x^2y + y^2 + xy^2 + x^2y^2$

Out[128]= $1 + 2x + y + 2xy + y^2 + 2xy^2$

Out[129]= $1 + x + x^2 + 2y + 2xy + 2x^2y$

And for second and later partial derivatives, we simply list the derivatives in the order they are to be taken. For example, $\partial^2 f / \partial x^2$ is calculated with `D[f[x], x, x]` which tells *Mathematica* to take the first derivative with respect to x and then the second derivative with respect to x. Mixed partial derivatives are achieved by specifying the variable in which we wish to derive with respect to as each argument. The order of the arguments is the order of the differentiation. Observe.

```
In[130]:= With[{p = Sum[x^i * y^j, {i, 0, 2}, {j, 0, 2}]},
          D[p, {x, 2}] // Print;
          D[p, x, y] // Print;
          D[p, y, x] // Print;
          D[p, {y, 2}] // Print;
          ]
```

Out[130]= $2 + 2y + 2y^2$

Out[131]= $1 + 2x + 2y + 4xy$

Out[132]= $1 + 2x + 2y + 4xy$

Out[133]= $2 + 2x + 2x^2$

Note that in Section 2.1.4, we performed kth derivatives of one variable with the input `D[expr, x, k]`. We may do this for multivariate expressions as well, however this input form will only work for successive differentiation with respect to only one of the variables. That is, it will only work for $\partial^k / \partial x^k$, $\partial^k / \partial y^k$, $\partial^k / \partial z^k$, etc. To put it yet another way, we cannot express mixed partial derivatives this way.

```
In[134]:= With[{p = Sum[x^i * y^j, {i, 0, 2}, {j, 0, 2}]},
          D[p, {x, 2}] // Print;
          D[p, {y, 2}] // Print;
```

Out[134]= $2 + 2y + 2y^2$

Out[135]= $2 + 2x + 2x^2$

There is no ´ notation for computing partial derivatives of functions, like we had for regular derivatives. However, the ´ notation is shorthand for the **Derivative** function. That is, **f´[x]** is shorthand for **Derivative[f][x]**. For a function of multiple variables, to compute the first partial derivative with respect to the first variable, we use the command **Derivative[1, 0]**. To compute the first partial derivative with respect to the second variable, we use the command **Derivative[0, 1]**. For example,

```
In[136]:= Block[{f, x, y}
            f[x_ , y_] := x^2 - y^2 + x * y;
            f[x, y] // Print;
            Derivative[1, 0][f][x, y] // Print;
            Derivative[0, 1][f][x, y] // Print;
          ]
```

Out[136]= $x^2 + xy - y^2$

Out[137]= $2x + y$

Out[138]= $x - 2y$

Note that **Derivative[f]**, which is equivalent to **f´** remember, is, itself, a function which will take a single argument and will output the derivative of **f** evaluated at that argument. This allows us to make sense of the input **Derivative[f][x]** or, equivalently, **f´[x]**. Similarly **Derivative[0, 1][f]** is also a function which will take arguments; in this case, we should expect it to take two. This allows us to make sense of the input **Derivative[0, 1][x,y]**. There's no reason these arguments have to be unknown variables. For instance, the first partial derivative with respect to x of the polynomial in the example, above, is $2x + y$. This partial derivative, evaluated at the point $(1, 2)$ is equal to 4. To compute this using the **Derivative** function, we may do the following.

```
In[139]:= Block[{f, x, y}
            f[x_ , y_] := x^2 - y^2 + x * y;
            Derivative[1, 0][f][1, 2]
          ]
```

Out[139]= 4

In this case, the following might be simpler, however.

```
In[140]:= D[x^2 - y^2 + x * y, x] /. {x -> 1, y -> 2}
```

Out[140]= 4

Returning to the **Derivative** function, the numbers within the first set of brackets indicate how many times to take the derivative with respect to each variable. For example, **Derivative[1, 0][f]** indicates that we wish to derive with respect to the first variable once, and the second variable not at all. To compute second partial derivatives, then, we simply use larger numbers. The following examples may be quickly checked by hand.

```
In[141]:= Block[{f, x, y}
            f[x_ , y_] := x^2 - y^2 + x * y;
            Derivative[1, 0][f][1, 2]
          ]
```

Out[141]= 4

The above commands correspond to $f_{x,x}$, $f_{x,y}$, and $f_{y,y}$ respectively. Note that this method gives us no way of determining the order in which the partial derivatives are given. That is, we cannot specify whether to derive with respect to x first, then y, or the other way around.

Recall that for a function with continuous second partial derivatives, then the second partial derivatives, $f_{y,x}$ and $f_{x,y}$ will be equal. This is Clairaut's theorem (see [12]). More precisely:

Theorem 1 (Clairaut's). *Let f be a function defined on a disk $D \subset \mathbb{R}^2$, such that all the second partial derivatives are continuous on D. Then $f_{y,x}(a,b) = f_{x,y}(a,b)$ for every $(a,b) \in D$.*

It follows, of course, that if the function f is defined for all of $\mathbb{R}^2$ and its partial derivatives are continuous on all of $\mathbb{R}^2$ then it will certainly be the case that $f_{y,x} = f_{x,y}$.

We have seen an example of this above, with the second partial derivatives of our polynomial p. A quick check of our first partial derivatives of our function f shows that the second partial derivatives $f_{x,y}$ and $f_{y,x}$ are clearly equal. Let's look at another example.

```
In[142]:= Block[{exp, x, y}
           exp = Sin[x^2 + y^2]
           D[exp, x, y]
           D[exp, y, x]
         ]
```

$$\text{Out[142]= } -4xy \operatorname{Sin}\left[x^2 + y^2\right]$$

$$\text{Out[143]= } -4xy \operatorname{Sin}\left[x^2 + y^2\right]$$

This even extends to higher partial derivatives.

```
In[144]:= Block[{exp, x, y}
           exp = Sin[x^2 + y^2]
           D[exp, x, x, y]
           D[exp, x, y, x]
           D[exp, y, x, x]
         ]
```

$$\text{Out[144]= } -8x^2 y \operatorname{Cos}\left[x^2 + y^2\right] - 4y \operatorname{Sin}\left[x^2 + y^2\right]$$

$$\text{Out[145]= } -8x^2 y \operatorname{Cos}\left[x^2 + y^2\right] - 4y \operatorname{Sin}\left[x^2 + y^2\right]$$

$$\text{Out[146]= } -8x^2 y \operatorname{Cos}\left[x^2 + y^2\right] - 4y \operatorname{Sin}\left[x^2 + y^2\right]$$

For an example where Clairaut's theorem does not hold, see Exercise 13.

The partial derivatives, just as regular derivatives, allow calculation of a line that is tangent to a surface, $f(x,y)$ say. However, there are potentially many such tangent lines in a three-dimensional space. As such, the partial derivatives f_x and f_y give the slope of a tangent line parallel to the x- or y-axes, respectively.

Lines in three-dimensional space are tricky, but may be described by a parametric equation. In the case of our directional derivatives, we know a point on the line, the direction each line is traveling, and the rate at which the height of each line changes (the slope). In short, we have all the information we need to plot the tangent lines.

For a surface $f(x,y)$ we may use the following parameterization of the tangent lines.

$$\begin{aligned}
\left(x_0 + u,\ y_0,\ f(x_0, y_0) + u \cdot f_x(x_0, y_0)\right) &\quad \text{in the } x \text{ direction} \\
\left(x_0,\ y_0 + v,\ f(x_0, y_0) + v \cdot f_y(x_0, y_0)\right) &\quad \text{in the } y \text{ direction}
\end{aligned}$$

and, of course, the obvious parameterization of the surface itself as $[u, v, f(u, v)]$.

We may (and, indeed, do) explore this in *Mathematica*.

```
In[147]:= Block[{tlx, tly, p, f, u, v, x0 = 0, y0 = 1}
        f[x_, y_] := x^2 + y^2;
        tlx = {x0 + u, y0, f[x0, y0] + u * Derivative[1, 0][f][x0, y0]};
        tly = {x0, y0 + v, f[x0, y0] + v * Derivative[0, 1][f][x0, y0]};
        p = {u, v, f[u, v]};
        ParametricPlot3D[{p, tlx, tly}, {u, -1, 1}, {v, -1, 1},
          BoxRatios -> {1, 1, 1}
        ]
      ]
```
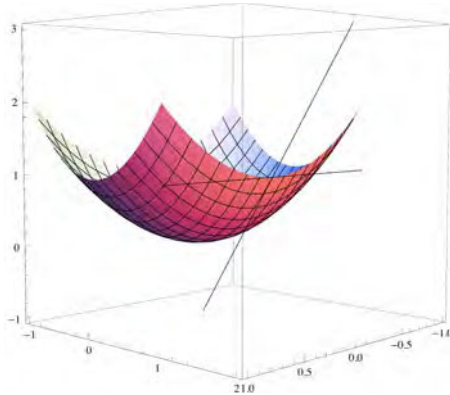
Out[147]=

Note that the author needed to manually rotate the plot in order to see its "best angle". Also note the **BoxRatios** option which was used to force the plot volume to be scaled to a cube. The reader is encouraged to look up this option in the Documentation Center for more information.

The directional derivatives allow us to calculate the *tangent plane* to a surface in 3-D. Given a function, $f(x, y)$ say, with continuous derivatives the equation of the tangent plane to the surface of f at a point (x_0, y_0, z_0) is

$$z - z_0 = \frac{\partial f}{\partial x}(x_0, y_0) \cdot (x - x_0) + \frac{\partial f}{\partial y}(x_0, y_0) \cdot (y - y_0)$$

which may be rewritten as

$$z = \frac{\partial f}{\partial x}(x_0, y_0) \cdot (x - x_0) + \frac{\partial f}{\partial y}(x_0, y_0) \cdot (y - y_0) + z_0$$
$$= \frac{\partial f}{\partial x}(x_0, y_0) \cdot (x - x_0) + \frac{\partial f}{\partial y}(x_0, y_0) \cdot (y - y_0) + f(x_0, y_0)$$

because if (x_0, y_0, z_0) is a point on the surface of $f(x, y)$, then it must be the case that $z_0 = f(x_0, y_0)$.

Turning to *Mathematica* now. First we'll look at the generic tangent plane equation for the paraboloid.

```
In[148]:= Block[{f, x0 = "a", y0 = "b"},
        f[x_, y_] := x^2 + y^2;
        f[x0, y0] + Derivative[1, 0][f][x0, y0] * (x - x0) +
```

```
            Derivative[0, 1][f][x0, y0] * (y - y0)
        ]
```

Out[148]= $a^2 + b^2 + 2a(-a + x) + 2b(-b + y)$

Well, that's all well and good, but it's about time we produced some plots.

```
In[149]:= Block[{f, tp, x0 = 0, y0 = 0},
          f[x_ , y_] := x^2 + y^2;
          tp[x_ , y_] := f[x0, y0] + Derivative[1, 0][f][x0, y0] * (x - x0)
             + Derivative[0, 1][f][x0, y0] * (y - y0);
          Plot3D[{f[x, y], tp[x, y]}, {x, -1, 1}, {y, -1, 1}]
        ]
```
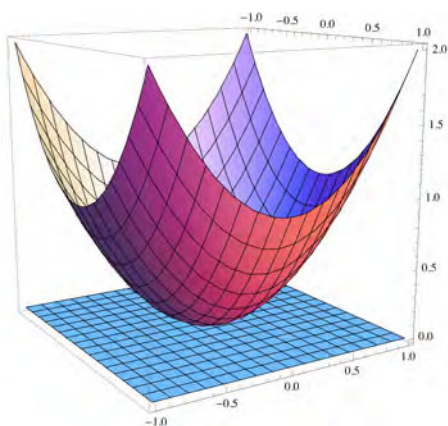
Out[149]=


```
In[150]:= Block[{f, tp, x0 = 1, y0 = 2},
          f[x_ , y_] := x^2 + y^2;
          tp[x_ , y_] := f[x0, y0] + Derivative[1, 0][f][x0, y0] * (x - x0)
             + Derivative[0, 1][f][x0, y0] * (y - y0);
          Plot3D[{f[x, y], tp[x, y]}, {x, -2, 2}, {y, -2, 2}]
        ]
```
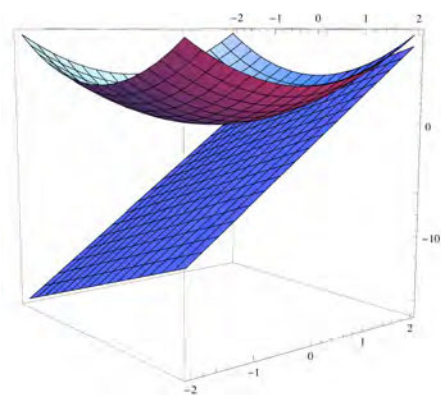
Out[150]=

We may, as we have seen, calculate tangent lines (and so instantaneous rate of change) at any point in either the x- or y-directions as we wish. Furthermore, we may use this information to calculate the tangent plane at the same point. It should follow, therefore, that we ought to be able to calculate the slope of a tangent line in any direction, and also that it should lie on the tangent plane we have already calculated.

To do this, we need a vector of unit length pointing in the direction we wish to calculate the slope. Call this vector $u = (a, b)$. Then we may calculate the directional derivative in the direction of u at any point (x, y)

$$D_u f(x, y) := f_x(x, y) \cdot a + f_y(x, y) \cdot b$$

We show that this vector is on the tangent plane. If we ignore the z-axis for the moment, we can parameterize the line from a point (x_0, y_0) in the direction of u as $[x_0 + t \cdot a, y_0 + t \cdot b]$ (think of the vector equation $(x_0, y_0) + t \cdot (a, b)$). Getting back to thinking three-dimensionally, we know that this line is climbing at the rate of $D_u f(x_0, y_0)$ in the z axis for each unit moved in the direction of u, which is supposed to be of unit length anyway.

This leads us to the parameterization of the line as

$$\left(x_0 + t \cdot a, \ y_0 + t \cdot b, \ f(x_0, y_0) + t \cdot D_u f(x_0, y_0) \right)$$

and so if a point on this line is on the plane, which we should recall has equation

$$z = f(x_0, y_0) + f_x(x_0, y_0) \cdot (x - x_0) + f_y(x_0, y_0) \cdot (y - y_0) \tag{2.1}$$

then it must be the case that the point satisfies equation 2.1. Therefore, if the entire line as parameterized above lies on the plane, then every point on it satisfies equation 2.1. So if we substitute $x = x_0 + t \cdot a$ and $y = y_0 + t \cdot b$ into 2.1 we should get $z = f(x_0, y_0) + t \cdot D_u f(x_0, y_0)$.

Over to *Mathematica* now.

```
In[151]:= Block[{f, tp, du, x0, y0, x, y, t, a, b},
            tp = f[x0, y0] + Derivative[1, 0][f][x0, y0] * (x - x0) +
              Derivative[0, 1][f][x0, y0] * (y - y0);
            du = Derivative[1, 0][f][x0, y0] * a +
              Derivative[0, 1][f][x0, y0] * b
            f[x0, y0] + t * du ==
              (tp /. {x -> x0 + t * a, y -> x0 + t * b})
          ]
```

$$\text{Out[151]= } f[x0, y0] + t\left(v f^{(0,1)}[x0, y0] + u f^{(1,0)}[x0, y0] \right) ==$$
$$f[x0, y0] + tv f^{(0,1)}[x0, y0] + tu f^{(1,0)}[x0, y0]$$

```
In[152]:= Simplify[%]
```

Out[152]= True

And there we have it. We could probably have done without the final simplification, inasmuch as it was clear that the two calculated expressions were the same, but it never hurts to check and we didn't have to go out of our way to do so. In any event, it is clear that the directional derivative in the direction of a unit vector u will give us the slope of a tangent line that lies on the tangent plane and travels parallel to the direction of u.

2.3.4 Double Integrals

Just as the space under a curve, but above the x-axis, may be calculated as an area via integration, so too can the space under a surface (i.e., a function $z = f(x, y)$), and above the x-y plane, may be calculated as a volume using integration.

The same basic approach applies. Given a range for x and y we have a rectangle that is a subsection of the xy-plane. We partition the x and y ranges, resulting in the area under the surface being partitioned into rectangles. We then create rectangular prisms by choosing a point inside every rectangle, and setting the height of that rectangular prism to be the height of the function evaluated at that point. This should sound awfully similar to the method of approximating the area under a curve with rectangles.

If we let A_{ij} be the area of the (i, j)th sub-rectangle (i.e., the rectangle that is at the intersection of the ith x-partition and jth y-partition), and (x^*_{ij}, y^*_{ij}) be a point inside that sub-rectangle, then the area of the resulting rectangular prism is $f(x^*_{ij}, y^*_{ij})A_{ij}$. The volume underneath the surface (and above the xy-plane) may be approximated by adding up the volumes of all the rectangular prisms.
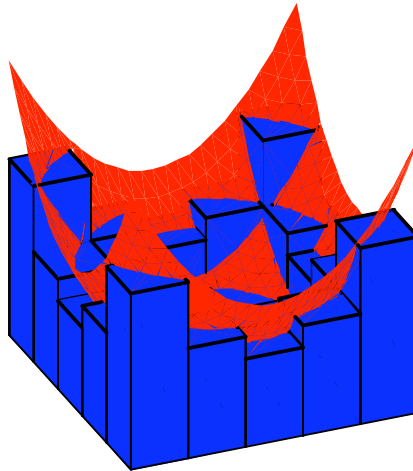
Fig. 2.3 Approximation of volume under a surface by rectangular prisms.

Figure 2.3 demonstrates this technique by using midpoints as the sample points (x^*_{ij}, y^*_{ij}). Random points work just as well, as it happens (and, incidentally, also work for Riemann sums of single variable functions).

We obtain the volume under the curve by taking progressively more and more rectangular prisms (which are, in turn, smaller and smaller). As such the volume under the surface may be approximated by the double sum

$$\sum_{i=1}^{n} \sum_{j=1}^{m} f(x^*_{ij}, y^*_{ij})A_i j$$

and the volume as the limit of this sum as both n and m approach ∞.

$$V = \lim_{n,m \to \infty} \sum_{i=1}^{n} \sum_{j=1}^{m} f(x^*_{ij}, y^*_{ij})A_i j$$

The question now is, how do we turn this into an integral? The key lies in recognizing that the two sums (above) may be taken to infinity independently. In integration terms, fix y as a constant, and calculate the integral of the function as if it were just a function of the single variable x. The result will be a function of y that tells us the *area* under the surface for any given y value. Integrating this function with respect to y will give us the volume. This is, in fact, very similar to our volumes of revolution from Section 2.3.2. If we think of this as accumulation, we accumulate an infinite amount of areas to obtain a volume, just as we did with the disks or cylinders of the solids of revolution.

To make this more rigorous, let $f(x, y)$ be a function of two variables. We wish to find the volume under the surface of f for $x \in [x_1, x_2]$ and $y \in [y_1, y_2]$. In other words we want to integrate over the rectangle $[x_1, x_2] \times [y_1, y_2]$. Furthermore, suppose that $f(x, y)$ is *continuous* over that rectangle. The integral $\int_{x_1}^{x_2} f(x, y) \, dx$ is used to denote integrating with respect to x while holding y fixed. Similarly $\int_{y_1}^{y_2} f(x, y) \, dy$ denotes integrating with respect to y while holding x to be fixed. These are called *partial integrals* (compare to partial derivatives).

Then the volume can be calculated as

$$V = \int_{y_1}^{y_2} \left(\int_{x_1}^{x_2} f(x, y) \, dx \right) dy$$

which, for simplicity's sake is usually just written without the bracketing, as it is understood that the innermost integral needs to be performed first, before the outermost integral may be performed

$$V = \int_{y_1}^{y_2} \int_{x_1}^{x_2} f(x, y) \, dx \, dy$$

Let's look at this in *Mathematica* using our favorite paraboloid, over the rectangle $[-2, 2] \times [-2, 2]$.

```
In[153]:= Integrate[x^2+y^2, {x, -2, 2}] Integrate[%, {y, -2, 2}]
```

$$\text{Out[153]= } \frac{16}{3} + 4y^2$$

$$\text{Out[154]= } \frac{128}{3}$$

We see from the above that the first integration did indeed leave us with a function of y. We also see that, apparently, the volume is $128/3$ cubic units. We should hope, given the double sum definition, and the basic idea that we are calculating a volume, that if the order of integration is reversed, we should obtain the same answer. As it happens, we do indeed.

```
In[155]:= Integrate[x^2+y^2, {y, -2, 2}] Integrate[%, {x, -2, 2}]
```

$$\text{Out[155]= } \frac{16}{3} + 4x^2$$

$$\text{Out[156]= } \frac{128}{3}$$

A guarantee that the double integral will always give the same answer, no matter the order the integrals are performed in, is given by Fubini's theorem. The guarantee is dependent on the function f being continuous over the rectangle in question. In fact, it is even true sometimes when f is not continuous over the rectangle, but we do not concern ourselves with the generalization.

Mathematica is often capable, it should come as no surprise to find, of handling double (even multiple) integrals without the need to manually perform each single integral. This is achieved by simply having multiple iterator arguments, much like we did for the `Sum` and `Table` functions back in Section 1.2.5.

```
In[157]:= Integrate[x^2+y^2, {x, -2, 2}, {y, -2, 2}]
          Integrate[x^2+y^2, {y, -2, 2}, {x, -2, 2}]
```
$$Out[157]= \frac{128}{3}$$
$$Out[158]= \frac{128}{3}$$

The reader may have noticed that the volume we get for this paraboloid is somewhat larger than the 8π cubic units of the paraboloid we obtained from revolving $z = x^2$ around the z-axis. Observe.

```
In[159]:= {128/3, 8 Pi} // N
```
$$Out[159]= \{42.6667, 25.1327\}$$

This is explained by remembering that the volume of revolution was calculated with a circular base, whereas our paraboloid has a square base. The extra area under the corners (where, incidentally, the paraboloid realizes its largest values) explains the discrepancy.

To see this, we can adopt a couple of approaches. The first (and least satisfying) is to plot both surfaces together and see if we can see one completely contained in the other.

```
In[160]:= Block[{P},
            P[1] = Plot3D[x^2+y^2, {x, -2, 2}, {y, -2, 2},
              PlotStyle -> Blue
            ];
            p[2] = RevolutionPlot3D[r^2, {r, 0, 2}, PlotStyle -> Red];
            Show[P[1], P[2]]
          ]
```
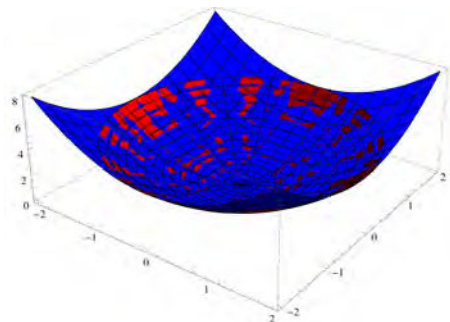

Out[160]=

We can see where the revolved paraboloid stops, and the other continues, but it's messy to say the least. A more compelling way of convincing ourselves would be to make a new function that is $x^2 + y^2$ as long as $\sqrt{x^2 + y^2} \leq 2$ and 0 otherwise. That is, we want

$$F(x, y) = \begin{cases} x^2 + y^2 & \text{if } \sqrt{x^2 + y^2} \leq 2 \\ 0 & \text{otherwise} \end{cases}$$

Integrating this function then should only give us the area of the paraboloid under the circle of radius 2 centered at the origin, and thus our 8π volume. To achieve this we use the **Piecewise** function.

In[161]:= **Piecewise[{{x^2 + y^2, Sqrt[x^2 + y^2 <= 2}}, 0]]**
          Integrate[%, {x, -2, 2}, {y, -2, 2}]

Out[161]= $\begin{cases} x^2 + y^2 & \sqrt{x^2 + y^2} \leq 2 \\ 0 & \text{True} \end{cases}$

Out[162]= 8π

Readers familiar with iterated integrals should be able to verify this using integration techniques for type I or type II regions (see [12]). We content ourselves with the above.

Let us return to the idea of partial integration. We should expect the notions of integrals as antiderivatives to extend to partial integrals and partial differentiation. We explore this in *Mathematica*.

In[163]:= **Integrate[x^2 + y^2, x]**
          D[%, x]

Out[163]= $\dfrac{x^3}{3} + xy^2$

Out[164]= $x^2 + y^2$

In[165]:= **Integrate[x^2 + y^2, y]**
          D[%, y]

Out[165]= $x^2 y + \dfrac{y^3}{3}$

Out[166]= $x^2 + y^2$

That certainly looks promising. Partial integration with respect to x or y is undone by partial differentiation with respect to x or y as appropriate. Also interesting is that if we follow the usual substitution done by hand in integration, we can probably obtain the definite integrals *Mathematica* calculated for us earlier.

In[167]:= **With[{int = x^2 + y^2, x},**
          (int /. x -> 2) - (int /. x -> -2)
          ]

Out[167]= $\dfrac{16}{3} + 4y^2$

In[168]:= **With[{int = x^2 + y^2, y},**
          (int /. y -> 2) - (int /. y -> -2)
          ]

Out[168]= $\dfrac{16}{3} + 4x^2$

That is, we have shown that

$$\left[\frac{1}{3} x^3 + y^2 x \right]_{x=-2}^{x=2} = \frac{16}{3} + 4y^2 \text{ and } \left[x^2 y + \frac{1}{3} y^3 \right]_{y=-2}^{y=2} = 4x^2 + \frac{16}{3}$$

which is all as it should be, but is nonetheless nice to have verified.

Finally, we try a truly general function, and see if *Mathematica*'s partial differentiation and partial integration still undo each other.

In[169]:= `D[Integrate[f[x, y], x], x]`
          `Integrate[D[f[x, y], x], x]`

Out[169]= $f[x, y]$

Out[170]= $f[x, y]$

In[171]:= `D[Integrate[f[x, y], y], y]`
          `Integrate[D[f[x, y], y], y]`

Out[171]= $f[x, y]$

Out[172]= $f[x, y]$

Mathematica certainly seems to think that partial differentiation and partial integration are inverse procedures.

We have only just scraped the surface here with double integrals in particular, and multivariate calculus in general. The interested reader is encouraged to read [12] and/or any other good calculus texts. Exploration and confirmation of such material should be possible with the tools used and discussed in this section.

2.4 Exercises

The exercises for this, and subsequent, chapters are less numerous and slightly longer in duration when compared to the exercises from Chapter 1. An effort has been made to keep the amount of work each question requires roughly the same for each question, and also for each question to be more or less self-contained. However, no guarantees to this effect are made.

1. Plot the following functions. Make sure that, where possible, you plot all important information to the plot; turning points, zeroes, and so on.

    a. $x^5 - 7x^4 - 162x^3 + 878x^2 + 3937x - 15015$
    b. $\dfrac{\sin x}{x}$
    c. $\dfrac{\cos x - 1}{x}$
    d. $x^5 - 3x^4 + x^2 - x - 5$
    e. $\sin(x)^2 + \cos(x)^2$

2. Consider the expressions $p(y) = -y^3 \log(y)$ and $q(y) = y \cdot (\log(y))^2$, with $0 \le y \le 2$.

    a. For what ranges of y is $p(y) < q(y)$?
    b. For what ranges of y is $q(y) < p(y)$?
    c. How do these expressions behave when $y > 2$? Is one always larger than the other? Can you be sure?

3. Evaluate the limits of the following functions for $x = \pm\infty$ as well as any undefined points. Produce appropriate plots to demonstrate these limits.

    a. $\tanh x$

    b. $\dfrac{x^2 + 1}{x^2 - 1}$

    c. $\sin \dfrac{1}{x}$

    d. $\cos \dfrac{1}{x}$

4. Find all critical points, maxima and minima, and inflection points for the function $y = x^4 + x$.

5. A cone may be constructed by cutting a sector out of a circle, and then joining the two straight lines CA and CB (from the diagram to the right) which are created by the removal of arc. If the circle has radius R, then find a formula for the maximum volume that such a cone may have.
   Hint: Try finding the circumference of the circle at the top of the cone.
   Hint2: You will probably need to tell *Mathematica* some assumptions about your variables.
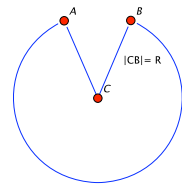
6. Evaluate the following integrals. In each case the answer is a combination (i.e., sums or products) of constants such as e, $\sqrt{2}$, $\sqrt{3}$, π, $\zeta(3)$, $\log 2$, and γ (Euler's gamma constant). At least one even involves $\log \pi$. If *Mathematica* cannot calculate the integral directly, evaluate it numerically and try to identify the result.

    a. $\displaystyle\int_0^{\frac{\pi}{2}} \frac{x^2}{\sin^2 x}\, dx$

    b. $\displaystyle\int_0^{\frac{\pi}{2}} \frac{x^4}{\sin^4 x}\, dx$

    c. $\displaystyle\int_0^{\frac{\pi}{2}} \frac{\arcsin\left(\frac{\sqrt{2}}{2} \sin x\right) \sin x}{\sqrt{4 - 2\sin^2 x}}\, dx$

    d. $\displaystyle\int_0^{\infty} \frac{\log x}{\cosh^2 x}\, dx$

The following two integrals arise from mathematical physics, but neither had a known closed form as of 2009. This may have changed.

e. $\displaystyle\int_0^1 \frac{\log\left(\sqrt{3+y^2}+1\right) - \log\left(\sqrt{3+y^2}-1\right)}{1+y^2}\,\mathrm{d}y$

f. $\displaystyle\int_3^4 \frac{\operatorname{arcsec}(x)}{\sqrt{x^2-4x-3}}\,\mathrm{d}x$

7. Solve the following linear differential equations, verify the solution, and plot it for some values of the constant.

   a. $xy' + y = x\cos x$ (for $x > 0$)   b. $y' + (\cos x)y = \cos x$

   How do the solution curves change as the constant changes?

8. The differential equation

   $$x^2 y'' + xy' + (x^2 - \alpha^2)y = 0$$

   is known as the Bessel equation. The solutions to this equation give rise to the so-called *Bessel functions of the first and second kind*, $J_\alpha(x)$ and $Y_\alpha(x)$, respectively.

   a. Solve the Bessel equation for the special cases of $\alpha = \frac{1}{2}$ and $\alpha = \frac{3}{2}$. Verify the solutions.
   b. Solve the Bessel equation for the general case. Verify the solution.
   c. Plot the Bessel functions J_α and Y_α for some values of α of your choosing.

   The modified Bessel functions of the first and second kind—$I_\alpha(x)$ and $K_\alpha(x)$, respectively—are solutions of the modified Bessel equation,

   $$x^2 y'' + xy' - (x^2 + \alpha^2)y = 0$$

   d. Solve the modified Bessel equation for the general case. Verify the solution.
   e. Plot the modified Bessel functions I_α and K_α for some values of α of your choosing.

9. a. Plot Pac-Man[6] on a set of Cartesian axes. You need only produce the basic outline.
      Hint: Try using multiple polar co-ordinate plots and the **Show** function.
   b. Find a polar equation for the circle of radius 3 with center $(1, 2)$. Plot the circle using this equation.

10. a. Find the volume obtained by rotating the area between by the curve $z = x^2$ and the z-axis for $x \in [0, 1]$ around the z-axis. Notice anything interesting?
    b. Plot and calculate the volumes obtained by rotating the following areas around the z-axis. In all cases the curve is $z = \log x$ with $x \in [0, 1]$.
       i. The area underneath the curve (between the curve and the x-axis)
       ii. The area between the curve and the z-axis.
       Check your answers by calculating both the disks and the shells method.

11. Find and classify the critical points of the following functions of two variables. Recall that critical points occur where $f_x(a, b) = f_y(a, b) = 0$. A critical point may be a maximum, a minimum, or a saddle point.

---

[6] Pac-Man is a video game character from the early 1980s. A quick Internet search should be all that is needed in order to know what the plot must look like.

a. $f(x,y) := 3x^2y + y^3 - 3x^2 - 3y^2 + 2.$ b. $f(x,y) := xye^{-x^2-y^2}$

12. Verify that the following functions are solutions to the partial differential equation $f_x + f_y = \sin(x) + \cos(y)$. A solution to a partial differential equation is a function whose partial derivatives satisfy the equation (compare to an ordinary differential equation).

a. $\sin(y) - \cos(x) + C \cdot (y - x)$
b. $\sin(y) - \cos(x) + (y - x)^n$

c. $\sin(y) - \cos(x) + \sqrt{y - x}$
d. $\sin(y) - \cos(x) + e^{C \cdot (y-x)^n}$

What do you think the general solution might be?

13. Let f be the function defined below.

$$f(x,y) := \begin{cases} \dfrac{xy(x^2 - y^2)}{x^2 + y^2} & \text{if } (x,y) \neq (0,0) \\ 0 & \text{if } (x,y) = (0,0) \end{cases}$$

a. Verify—visually or otherwise—that f is continuous at the origin.
b. Show that $f_{xy}(0,0) \neq f_{yx}(0,0)$ and so Clairaut's theorem does not apply to this function.

Note: You need to use the limit definition on all functions to establish the value of their partial derivatives at the point $(0,0)$.

14. Plot the following, and calculate their area using iterated integrals.

a. The area underneath $z = x^5y^3e^{xy}$ for $0 \le x \le 1$ and $0 \le y \le 1$
b. The area between $z = e^{-x^2}\cos(x^2 + y^2)$ and $z = 2 - x^2 - y^2$ for $|x| \le 1$ and $|y| \le 1$

The following iterated integrals are volumes underneath surfaces $(z = f(x,y))$ for points (x,y) that do not lie inside a rectangle. Calculate the integrals. Can you work out the bounding curves for the points (x,y)?

c. $\displaystyle \int_1^2 \int_0^y x^2y^2 \, dx \, dy$

d. $\displaystyle \int_0^2 \int_0^{\frac{x^2}{2}} \frac{x}{\sqrt{1 + x^2 + y^2}} \, dy \, dx$

2.5 Further Explorations

1. Finding limits.

a. Let $a_0 = 0, a_1 = \frac{1}{2}$ and define

$$a_{n+1} := \frac{1 + a_n + a_{n-1}^3}{3}$$

Determine the limit as $n \to \infty$, and find out what happens when $a_1 = a$ is allowed to vary.
b. Let $a_1 = 1$ and define

$$a_{n+1} := \frac{3 + 2a_n}{3 + a_n}$$

Determine the limit and find out what happens when $a_1 = a$ is allowed to vary.

The above two limits are easy enough to find and (depending on what you know) to prove.

c. Let $a_1 \geq 1$ be given and determine the limit of the iteration

$$a_{n+1} := a_n - \frac{a_n}{\sqrt{1 + a_n^2}} + \sin(\theta)$$

for arbitrary θ.

2. A *(strict) mean* $M(a, b)$ is a continuous function of two positive numbers that calculates a number c lying between a and b (strictly between them as long as $a \neq b$). The arithmetic and geometric means are clearly such objects.

A mean iteration takes two means, M and N say, and iterates by setting $a_0 = a$, $b_0 = b$ and

$$a_{n+1} := M(a_n, b_n), \quad b_{n+1} := N(a_n, b_n)$$

The limit of such a strict mean iteration always exists and is denoted by $M \otimes N(a, b)$. Identify the limits of the mean iteration defined by the means in the following.

a. $M(a, b) := \dfrac{a + b}{2}$, $N(a, b) := \dfrac{2ab}{a + b}$

b. $M(a, b) := \dfrac{a + \sqrt{ab}}{2}$, $N(a, b) := \dfrac{b + \sqrt{ab}}{2}$

3. Define

$$\mathrm{sinc}(x) := \frac{\sin x}{x}$$

and explore the following integrals. Calculate them for all (natural) values of N from 1 to 8 at least (more if you wish), and measure the time each calculation takes to perform. Can you work out what is going on?

a. $\displaystyle \int_0^\infty \prod_{n=0}^N \mathrm{sinc}\left(\frac{x}{2n + 1}\right) dx$

b. $\displaystyle \int_0^\infty \prod_{n=0}^N \mathrm{sinc}\left(\frac{x}{3n + 2}\right) dx$

Chapter 3
Linear Algebra

3.1 Introduction and Review

In this section we introduce *Mathematica*'s linear algebra capabilities, and examine some of the basic linear algebra that should already be familiar (or, at least, have been seen before now). We presume that the reader is proficient, at least, in Gaussian and Gauss–Jordan elimination of matrices.

3.1.1 Vectors and Matrices in Mathematica

Before we can start to explore much linear algebra in *Mathematica*, we must first know how to create the basic building blocks required. As such, this section is essentially devoted solely to *Mathematica* syntax and semantics for the basic building blocks of linear algebra; matrices and vectors. This is unfortunate, but necessary, and we return to more predominant mathematical endeavors as quickly as possible.

As previous stated, linear algebra more or less boils down to vectors and matrices (although the distinction is not all that clear and students of second-year or later linear algebra should see that matrices may, themselves, be the vectors of a vector space). *Mathematica* handles both vectors and matrices, using lists.

To declare a vector, we simply declare a list that has no sublists as elements.

```
In[1]:= {1, 2, 3, 4}
```
Out[1]= $\{1, 2, 3, 4\}$

Mathematica makes no effective differentiation between a column or a row vector. However, if we wish to see our vectors displayed as a row vector we may use the **MatrixForm** function. Be aware that once we do this, *Mathematica* no longer considers the output to be a vector (or matrix, if we are working with matrices), and so the **MatrixForm** function should only be used for "prettying-up" output.

```
In[2]:= {a, b, c}
```
Out[2]= $\begin{pmatrix} a \\ b \\ c \end{pmatrix}$

Inasmuch as we use lists to represent vectors in *Mathematica*, we may use the `Table` or `Array` commands to produce vectors whose elements have some sort of predictable pattern.

In[3]:= `Table[3 i, {i, 1, 5}]`
`Array[v, 4]`

Out[3]= $\{3, 6, 9, 12, 15\}$

Out[4]= $\{v[1], v[2], v[3], v[4]\}$

Matrices may be declared in a similar fashion. We declare a matrix as a list of vectors, where the vectors are the row vectors of the matrix. If we wish the output to be displayed in the usual matrix form, we again use the `MatrixForm` function.

In[5]:= `{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}`
`% // MatrixForm`

Out[5]= $\{\{1, 2, 3\}, \{4, 5, 6\}, \{7, 8, 9\}\}$

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

If we want to declare the matrix in terms of its column vectors, instead of its row vectors, then we are a little stuck. We dodge this limitation in *Mathematica*'s list representation by using the `Transpose` function to compute the transpose of a matrix. The *transpose* of a matrix, A say, is the matrix created when column vectors are changed to be row vectors, or—equivalently—vice versa, and is usually denoted A^T.

In[7]:= `{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}} // Transpose`
`% // MatrixForm`

Out[7]= $\{\{1, 4, 7\}, \{2, 5, 8\}, \{3, 6, 9\}\}$

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

Just as with vectors, we may also use the `Table` and `Array` functions for declaration of matrices. If we give two iterators to the `Table` function, then it is understood that the first iterator refers to the outer-most list, and the second iterator to the inner lists. For the `Array` function, we simply give a list of the dimensions of the matrix as the second argument.

In[9]:= `Table[10 i + j, {i, 1, 4}, {j, 1, 4}]`
`% // MatrixForm`
`Array[A, {3, 3}] // MatrixForm`

Out[9]= $\{\{11, 12, 13, 14\}, \{21, 22, 23, 24\}, \{31, 32, 33, 34\}, \{41, 42, 43, 44\}\}$

Out[10]= $\begin{pmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \\ 41 & 42 & 43 & 44 \end{pmatrix}$

Out[11]= $\begin{pmatrix} A[1, 1] & A[1, 2] & A[1, 3] \\ A[2, 1] & A[2, 2] & A[2, 3] \\ A[3, 1] & A[3, 2] & A[3, 3] \end{pmatrix}$

Now that we know how to produce vectors and matrices in *Mathematica*, we should expect to be able to perform the usual arithmetic with them. Matrices and vectors may both be added and subtracted with the usual *Mathematica* operators for those purposes. They may be scaled using the multiplication and division operators (note that division by a is the same as scaling the vector by a^{-1}). Similarly, matrices must be of the same size to be able to added.

```
In[12]:= {u1, u2} + {v1, v2}
         a * {u1, u2}
```

Out[12]= $\{u1 + v1, u2 + v2\}$

Out[13]= $\{au1, au2\}$

```
In[14]:= {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}} + {{a, b, c}, {d, e, f}, {g, h, i}}
         % // MatrixForm
         a * {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}
         % // MatrixForm
```

Out[14]= $\{\{1 + a, 2 + b, 3 + c\}, \{4 + d, 5 + e, 6 + f\}, \{7 + g, 8 + h, 9 + i\}\}$

Out[15]= $\begin{pmatrix} 1 + a & 2 + b & 3 + c \\ 4 + d & 5 + e & 6 + f \\ 7 + g & 8 + h & 9 + i \end{pmatrix}$

Out[16]= $\{\{a, 2a, 3a\}, \{4a, 5a, 6a\}, \{7a, 8a, 9a\}\}$

Out[17]= $\begin{pmatrix} a & 2a & 3a \\ 4a & 5a & 6a \\ 7a & 8a & 9a \end{pmatrix}$

Matrices and vectors may also be operated on together together using the . (dot) operator. In the case of vectors, this will compute the dot product, and in the case of vectors it will be matrix multiplication. In the case of a matrix and a vector, it will be the usual operation of a matrix begin applied to a vector; that is, matrix multiplication as if the vector is a matrix with a single column. Note that the dot operator is a "full stop" (a period).

```
In[18]:= {1, 2, 3} . {a, b, c}
         {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}} . {{a, b, c}, {d, e, f}, {g, h, i}}
          // MatrixForm
         {{a, b, c}, {d, e, f}, {g, h, i}} . {1, 2, 3}
```

Out[18]= $u1 + 2\,u2 + 3\,u3$

Out[19]= $\begin{pmatrix} a + 2d + 3g & b + 2e + 3h & c + 2f + 3i \\ 4a + 5d + 6g & 4b + 5e + 6h & 4c + 5f + 6i \\ 7a + 8d + 9g & 7b + 8e + 9h & 7c + 8f + 9i \end{pmatrix}$

Out[20]= $\{a + 2b + 3c, d + 2e + 3f, g + 2h + 3i\}$

The matrix multiplication is handy to see symbolically. We use the the fact that after the use of **MatrixForm**, *Mathematica* no longer recognizes the output as a list, and thus as a matrix or vector. This prevents operations from being carried out, and thus allows us to view the matrix equations.

```
In[21]:= With[{M = {{a, b, c}, {d, e, f}, {g, h, i}}, v = {1, 2, 3}},
          (M // MatrixForm) . (v // MatrixForm) == (M.v // MatrixForm)
          ]
```

Out[21]= $\begin{pmatrix} a\ b\ c \\ d\ e\ f \\ g\ h\ i \end{pmatrix} . \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} == \begin{pmatrix} a + 2b + 3c \\ d + 2e + 3f \\ g + 2h + 3i \end{pmatrix}$

Note that if the vector is on the left, the multiplication is performed as if the vector is a matrix with a single row. *Mathematica* does not, remember, distinguish between row and column vectors, but nonetheless applies the correct vector/matrix multiplication as is appropriate. Observe.

```
In[22]:= With[{M = {{a, b, c}, {d, e, f}, {g, h, i}}, v = {1, 2, 3}},
           v . (M // MatrixForm) == (v.M // MatrixForm)
         ]
```

Out[22]= $\{1, 2, 3\} . \begin{pmatrix} a\ b\ c \\ d\ e\ f \\ g\ h\ i \end{pmatrix} == \begin{pmatrix} a + 2d + 3g \\ b + 2e + 3h \\ c + 2f + 3i \end{pmatrix}$

We pause to mention here that the **Dot** function is equivalent to the **.** operator. That is **u . v** is the same as **Dot[u, v]**. Similarly **u . v . w** is the same as **Dot[u, v, w]** and so on and so forth. We shall not make any use of the **Dot** function explicitly in this book, preferring instead the **.** operator notation.

Finally, before we move on to something more mathematical, we demonstrate look at powers and inverses of matrices. Unfortunately, the usual **^** operator for powers or inverses won't work with matrices. Instead, the operator is applied to each element of the list.

```
In[23]:= {{1, 2}, {3, 4}} . {{1, 2}, {3, 4}} // MatrixForm
         {{1, 2}, {3, 4}}^2 // MatrixForm
```

Out[23]= $\begin{pmatrix} 7\ 10 \\ 15\ 22 \end{pmatrix}$

Out[24]= $\begin{pmatrix} 1\ 4 \\ 9\ 16 \end{pmatrix}$

Calculating powers of a matrix is easy enough with the **.** operator for squares or cubes, but if we wish to do large powers, it is better to use the **MatrixPower** function. This function accepts a matrix as its first argument, and an integer as its second. The function simply computes the matrix to the power of the second argument.

```
In[25]:= MatrixPower[{{1, 2}, {3, 4}}, 2] // MatrixForm
         MatrixPower[{{1, 2}, {3, 4}}, 10] // MatrixForm
```

Out[25]= $\begin{pmatrix} 1\ 4 \\ 9\ 16 \end{pmatrix}$

Out[26]= $\begin{pmatrix} 4\,783\,807 & 6\,972\,050 \\ 10\,458\,075 & 15\,241\,882 \end{pmatrix}$

Inasmuch as the **^** operator does not work with matrices, we can hardly expect to be able to use it to compute matrix inverses using the usual **^-1** notation. Instead, we must use the **Inverse** function to compute the inverse of a matrix (if it exists). Recall that for a square matrix A the inverse matrix is the unique matrix (usually written A^{-1}) with the property that $A^{-1}A = AA^{-1} = I$.

```
In[27]:= {{1, 2}, {3, 4}} // Inverse // MatrixForm
```

Out[27]= $\begin{pmatrix} -2 & 1 \\ \frac{3}{2} & -\frac{1}{2} \end{pmatrix}$

```
In[28]:= Block[{A, Ai},
           A = {{1, 2}, {3, 4}};
           Ai = A // Inverse;
           {(A . Ai) // MatrixForm, (Ai . A) // MatrixForm}
         ]
```

Out[28]= $\left\{ \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \right\}$

Further discussion on functions relating to vectors, matrices, and *Mathematica* related approaches to linear algebra in general takes place as and when they are needed in the course of the mathematics that follows. The reader is encouraged to supplement the information presented above by referring to the Documentation Center. A good start is to read the pages on "Vectors and Matrices" as well as "Linear Algebra".

3.1.2 Simultaneous Linear Equations

One of the first parts of linear algebra a student sees is the method of solving systems of linear equations using matrices and Gauss–Jordan elimination. We re-examine this topic now.

Let's start with a simple example. We want to find values for x and y that satisfy both the equations $x + y = 2$ and $2x + y = 3$ simultaneously. This we should be able to do quickly in our heads, or on paper. If we were to use *Mathematica* we would probably use the **Solve** command.

```
In[29]:= Solve[{x + y == 2, 2 x + y == 3}]
```

Out[29]= $\{\{x \to 1, y \to 1\}\}$

However, we can also attack this problem using our tools from linear algebra. Let's start by constructing a vector where the elements inside the vector are the equations we are trying to simultaneously solve

$$\begin{bmatrix} x + y = 2 \\ 2x + y = 3 \end{bmatrix}$$

This can be thought of as

$$\begin{bmatrix} x + y \\ 2x + y \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

which in turn can be thought of as

$$\begin{bmatrix} 1 & 1 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

which is an equation of the form $Ax = b$ where A is a matrix, and x and b are matrices.

The solution to this problem is to construct the *augmented matrix*

$$\begin{bmatrix} 1 & 1 & 2 \\ 2 & 1 & 3 \end{bmatrix}$$

and to perform *Gaussian elimination* in order that the matrix be in *row echelon form*, or *Gauss–Jordan elimination* in order that the matrix be in *reduced row echelon form*.

Reduced row echelon is the easier form to see the solutions directly, but is usually a little tricky (or, at least fiddly) to produce by hand. Fortunately, we do not have such problems when using a CAS, and *Mathematica* includes a function that will calculate the reduced row echelon form of a matrix named **RowReduce**.

```
In[30]:= With[{A = {{1, 1, 2}, {2, 1, 3}}},
          RowReduce[A] // MatrixForm
        ]
```

$$\text{Out[30]}= \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

We can read the answer that $x = 1$ and $y = 1$ directly from this matrix, remembering that it is an augmented matrix representing the vector equation

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

which is equivalent to our original equations (due to the fact that elementary row operations, as performed in Gauss–Jordan elimination do not change the solution of a system of equations).

In general, for a system of m simultaneous linear equations in n unknowns, we construct the $m \times n$ coefficient matrix, A say, and the vector of constants (the right-hand sides of the equations) b. The system may then be considered as the vector equation

$$Ax = b$$

where x is the vector of the unknown values. We then calculate the solution by reducing the augmented matrix $[A \,|\, b]$ into reduced row echelon form. (Note here that $[A \,|\, b]$ is the matrix A with b added in as a final column vector). This should not be anything new to any student who has studied first-year university mathematics.

In the case of systems of equations with two or three unknowns, there is a clear geometric interpretation of these systems of linear equations. The equation $ax + by = c$ describes a unique line in the Euclidean plane. This line can be thought of as the collection of all points (x, y) in the plane that satisfy the equation $ax + by = c$. When evaluating two such equations simultaneously, we are looking for the collection of all points (x, y) that satisfy both equations at the same time, or—geometrically—the collection of all points that lie on both lines (the intersection of the two lines).

With the plane, there are not very many different things that can happen. Either the equations all describe the same line, in which case we should see infinitely many solutions, or the lines will meet at a point somewhere, in which case there will be one solution, or the lines are parallel in which case there will be no solutions.

Let's have a look at the example from above, $x + y = 2$ and $2x + y = 3$. We already know that the only solution to this system of equations is $x = 1, y = 1$. When we plot these lines we should expect to see them crossing at the point $(1, 1)$. Note that in order to plot these lines, we needed to manipulate the equations into the equivalent forms of $y = 2 - x$ and $y = 3 - 2x$.

```
In[31]:= Plot[{2 - x, 3 - 2 x}, {x, -1, 2}, PlotStyle -> {Automatic, Red}]
```
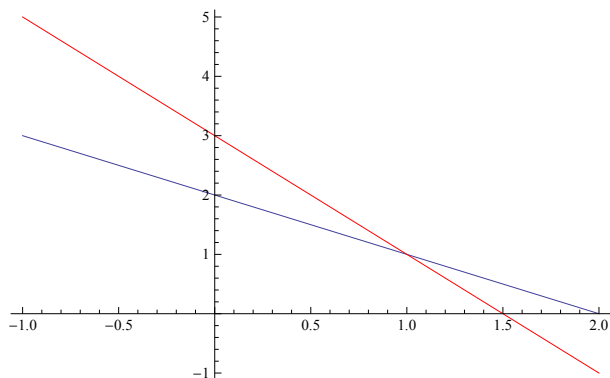
Out[31]=

When more than two lines are introduced, the chances that there is any point at which they all intersect becomes smaller (it is quite likely that any two lines will intersect; the point here is that a solution to the simultaneous equations must be a point where all the lines intersect at the same point). For instance, if we extend the system above to also include the equations $-3x + y = 4$ and $-x + y = -2$, plotting all four lines shows clearly that there is no common intersection point for the four lines

```
In[32]:= Plot[{2 - x, 3 - 2 x, 4 + 3 x, x - 2}, {x, -5, 5},
          PlotStyle -> {Blue, Blue, Red, Black}
        ]
```
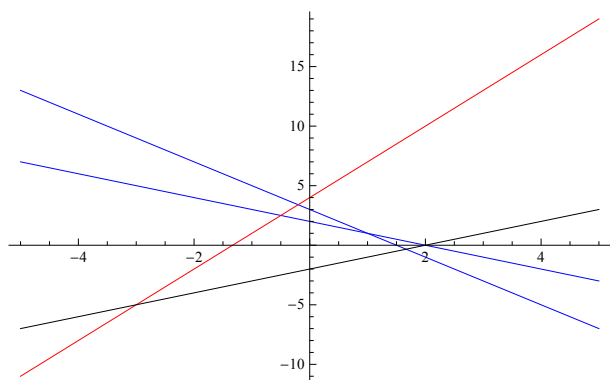

Out[32]=

And the linear algebra produces the expected result.

```
In[33]:= M = {{1, 1}, {2, 1}, {-3, 1}, {-1, 1}};
        b = {2, 3, 4, -2};
        {"M" == (M // MatrixForm), "b" == (b // MatrixForm)}
```

$$\text{Out[33]= } \left\{ M == \begin{pmatrix} 1 & 1 \\ 2 & 1 \\ -3 & 1 \\ -1 & 1 \end{pmatrix}, b == \begin{pmatrix} 2 \\ 3 \\ 4 \\ -2 \end{pmatrix} \right\}$$

Note here that we have set the `M` and `b` variables globally. We expect to be using them over multiple input commands, and so did not have the luxury of hiding them away within a `Block` or a `With` function.

Continuing on, let's have a look at the matrix equation.

```
In[34]:= (M . {x, y} // MatrixForm) == (b // MatrixForm)
```

Out[34]= $\begin{pmatrix} x + y \\ 2x + y \\ -3x + y \\ -x + y \end{pmatrix} == \begin{pmatrix} 2 \\ 3 \\ 4 \\ -2 \end{pmatrix}$

That's all well and good, we can see the four equations, and it's a simple task to check that they're the same equations that we plotted, above. All that remains is to row-reduce the augmented matrix.

```
In[35]:= With[{Aug = Join[M, Partition[b, 1], 2]},
         RowReduce[Aug] // MatrixForm
       ]
```

Out[35]= $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$

We can read directly from this reduced row echelon form augmented matrix in that there is no solution. In particular, the second row from the bottom which should be read $0x + 0y = 1$ indicates the lack of any solution.

Note the use of both the **Join** and **Partition** functions. The **Join** function, previously used to join elements of lists together, can also be used to join the elements of sub-lists together. In the above case, the final argument of 2 tells **Join** that we wish to join the second level of lists (that is lists as direct elements of the outer-most list) together. The **Partition** function has the effect, in this case, of changing the list **{2, 3, 4, -2}** into the list **{{2}, {3}, {4}, {-2}}**, thus ensuring that it has second level lists for the **Join** function to use. Observe.

```
In[36]:= M
         Partition[b, 1]
         Join[%%, %, 2]
```

Out[36]= $\{\{1, 1\}, \{2, 1\}, \{-3, 1\}, \{-1, 1\}\}$

Out[37]= $\{\{2\}, \{3\}, \{4\}, \{-2\}\}$

Out[38]= $\{\{1, 1, 2\}, \{2, 1, 3\}, \{-3, 1, 4\}, \{-1, 1, -2\}\}$

As it happens, *Mathematica* has another function for solving linear systems which has the advantage of giving us the solution directly, without having to construct and read from the augmented matrix. That function is **LinearSolve** function, and takes as its arguments a matrix and a vector, in that order. If we suppose that the matrix is M and the vector b, then **LinearSolve** effectively solves the vector vector equation $M \cdot x = b$ for x.

```
In[39]:= LinearSolve[M, b]
```

LinearSolve::nosol: Linear equation encountered that has no solution. »

Out[39]= $\text{LinearSolve}[\{\{1, 1\}, \{2, 1\}, \{-3, 1\}, \{-1, 1\}\}, \{2, 3, 4, -2\}]$

It doesn't really come much clearer than that error message, so long as we know that an inconsistent system of linear equations has no solution. Even if we didn't know this, it should be quite clear form the context.

We quickly look again at the first system of linear equations from the beginning of this subsection, and solve it with the **LinearSolve** function. Recall that the solution

was $x = y = 1$, so we should expect the vector (x, y) solution to the vector equation to be $(1, 1)$.

In[40]:= **LinearSolve[{{1, 1}, {2, 1}}, {2, 3}]**

Out[40]= $\{1, 1\}$

Let's now have a look at this in three dimensions. Recall that in real 3-space, the equation $ax + by + cz = d$ describes a plane. The intersection of any three or more different planes will either be a line, a point, or nonexistent. In the case of only two distinct planes then the intersection can only be a line or nonexistent. We start with the following set of equations.

$$x + y + z = 3$$
$$-x - y + z = -2$$
$$2x + y + z = 0$$

Our aim here is to see how the planes intersect (if they do at all). To do this most effectively we need to know if they intersect, and if so then where. So we first solve the system of equations. We solve these equations using the **LinearSolve** function.

In[41]:= **With[{M = {{1, 1, 1}, {-1, -1, 1}, {2, 1, 1}}, b = {3, -2, 0}},**
       LinearSolve[M, b]
    ]

Out[41]= $\left\{ -3, \dfrac{11}{2}, \dfrac{1}{2} \right\}$

And so we now see that there is but a single point, $[-3, \frac{11}{2}, \frac{1}{2}]$ at which the three planes intersect. We can now plot the planes in such a way as to see this intersection clearly. In order to have the intersection point close to the middle of the plot (so as, we hope, to see the most of the interrelations of the planes) we plot over the x-range $(-4, -2)$ and the y-range $(4, 6)$.

In[42]:= **Plot3D[{3 - x - y, -2 + x + y, -2 x - y}, {x, -4, -2}, {y, 4, 6}**
       PlotStyle -> {Blue, Red, White}
    ]
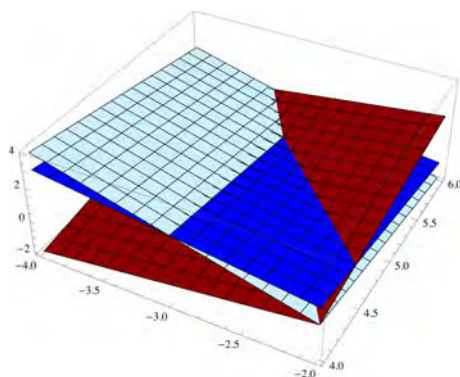
Out[42]=

To close with, we clean up after ourselves, unsetting everything we have set in the subsection.

In[43]:= **ClearAll["Global`*"]**

3.1.3 Elementary Row Operations

We now look more closely at the elementary row (and, equivalently, column) operations. There are three basic operations that may be performed:

- Swap two rows.
- Multiply a row by a constant.
- Add a row to another row.

It is expected that none of this is new to the reader.

What may be new, however, is that it is possible to construct matrices that perform these row operations when they are multiplied (on the left) with another matrix. It is this idea that we explore. It is important to note at this point that there is no practical purpose to performing row operations with matrix multiplication and, indeed, directly performing the operations is quicker and easier. However, the mere fact that we can do this allows us to prove some facts about invertible matrices.

To construct these row-operation matrices, called *elementary matrices*, we simply perform a row operation manually on the identity matrix. The resulting matrix will then perform that same row operation when it is multiplied with another matrix. We demonstrate this for the case of 2×2 matrices.

First we must write some functions to perform the elementary row operations for us, because *Mathematica* does not have any such functions itself. The functions will take two arguments, the first one being a matrix, and the second being a list of two numbers, $\{i, \ j\}$ say. In the case of swapping rows, we swap the rows i and j, in the case of adding rows, we add row i to row j, and in the case of multiplying a row by a constant we multiply row i by j. We take some pains in pattern matching to make sure the arguments are of the appropriate type.

```
In[44]:= RowSwp[
           M_ /; MatrixQ[M],
           {r1_Integer, r1_Integer} /; (r1 > 0 && r2 > 0)
         ] :=
         Block[{N = M},
           N[[r1]] = M[[r2]];
           N[[r2]] = M[[r1]];
           N
         ]

In[45]:= RowMul[
           M_ /; MatrixQ[M],
           {r_Integer, k_} /; (r > 0)
         ] :=
         Block[{N = M},
           N[[r]] = k * M[[r]];
           N
         ]

In[46]:= RowAdd[
           M_ /; MatrixQ[M],
           {r1_Integer, r1_Integer} /; (r1 > 0 && r2 > 0)
         ] :=
         Block[{N = M},
           N[[r1]] = N[[r1]] + M[[r2]];
```

```
        N
    ]
```

Note that we have placed no restriction on the variable, `k`, used for multiplying rows of a matrix in the `RowMul` function. This is to allow us to multiply matrix rows by arbitrary variables, or even expressions if we wish to later. Also note that while we have taken pains with the pattern matching, we have not put any upper bounds on the rows we may select. It is entirely possible with these functions to try and swap rows 30 and 147, say, of a 2×2 Matrix. If we try to do this, however, we are met with *Mathematica* errors, so we consider the function to be good enough for our purposes.

```
In[47]:= RowSwp[{{1, 2}, {3, 4}}, {30, 147}]
```

Part::partw: "Part 147 of 1,2,3,4 does not exist." »

Set::partw: "Part 30 of 1,2,3,4 does not exist. " »

Part::partw: "Part 30 of 1,2,3,4 does not exist. " »

Set::partw: "Part 147 of 1,2,3,4 does not exist." »

Out[47]= $\{\{1, 2\}, \{3, 4\}\}$

We observe now what happens when we now take the 2×2 identity matrix (obtained using the `IdentityMatrix` function), and swap its rows. Specifically, we observe what happens when the resulting matrix is multiplied against an arbitrary matrix.

```
In[48]:= With[{G = {{a, b}, {c, d}}, S = RowSwp[IdentityMatrix[2], {1, 2}]},
            S // MatrixForm // Print;
            S.G // MatrixForm // Print;
            G.S // MatrixForm // Print;
        ]
```

Out[48]= $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$

Out[49]= $\begin{pmatrix} c & d \\ a & b \end{pmatrix}$

Out[50]= $\begin{pmatrix} b & a \\ d & c \end{pmatrix}$

When we multiply our matrix S on the left against an arbitrary matrix, we can see that it swaps the rows as claimed. This behavior can be verified on paper, and doing so should demonstrate just why the matrix does this. Furthermore, when we multiply on the right by our matrix, it performs the swap as a column operation. This may also be confirmed on paper.

Continuing on in this manner, we now observe the behavior matrices for multiplying a given row by a constant.

```
In[51]:= With[{G = {{a, b}, {c, d}}, S = RowMul[IdentityMatrix[2], {1, k}]},
            S // MatrixForm // Print;
            S.G // MatrixForm // Print;
            G.S // MatrixForm // Print;
        ]
```

Out[51]= $\begin{pmatrix} k & 0 \\ 0 & 1 \end{pmatrix}$

Out[52]= $\begin{pmatrix} ak & bk \\ c & d \end{pmatrix}$

Out[53]= $\begin{pmatrix} ak & b \\ ck & d \end{pmatrix}$

```
In[54]:= With[{G = {{a, b}, {c, d}}, S = RowMul[IdentityMatrix[2], {2, k}]},
          S // MatrixForm // Print;
          S.G // MatrixForm // Print;
          G.S // MatrixForm // Print;
        ]
```

Out[54]= $\begin{pmatrix} 1 & 0 \\ 0 & k \end{pmatrix}$

Out[55]= $\begin{pmatrix} a & b \\ ck & dk \end{pmatrix}$

Out[56]= $\begin{pmatrix} a & bk \\ c & dk \end{pmatrix}$

Sure enough these two matrices behave just as we should have expected. Multiplying on the left against an arbitrary matrix performs the row operation, and multiplying on the right performs the equivalent column operation.

Finally, we observe the behavior of the matrices for adding a row to another row. Unsurprisingly, we see the same behavior as we did with the other matrices.

```
In[57]:= With[{G = {{a, b}, {c, d}}, S = RowAdd[IdentityMatrix[2], {1, 2}]},
          S // MatrixForm // Print;
          S . G // MatrixForm // Print;
          G . S // MatrixForm // Print;
        ]
```

Out[57]= $\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$

Out[58]= $\begin{pmatrix} a+c & b+d \\ c & d \end{pmatrix}$

Out[59]= $\begin{pmatrix} a & a+b \\ c & c+d \end{pmatrix}$

```
In[60]:= With[{G = {{a, b}, {c, d}}, S = RowAdd[IdentityMatrix[2], {2, 1}]},
          S // MatrixForm // Print;
          S . G // MatrixForm // Print;
          G . S // MatrixForm // Print;
        ]
```

Out[60]= $\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$

Out[61]= $\begin{pmatrix} a & b \\ a+c & b+d \end{pmatrix}$

Out[62]= $\begin{pmatrix} a+b & b \\ c+d & d \end{pmatrix}$

It should be mentioned at this point that these elementary matrices need not be multiplied with only square matrices. The elementary matrices themselves will always

be square (because they are obtained by performing a row operation on an identity matrix, which is always square). However, just as row operations may be performed on any size matrix, so will the elementary matrices perform their appropriate row operation on any size matrix. The only stipulation is the usual one for matrix multiplication, which is that when multiplying matrices A and B together, A must have the same number of columns as B has rows.

What this means for our elementary matrices is that an $n \times n$ elementary matrix will perform its row operation when multiplying on the left any matrix with exactly n rows. Similarly, when multiplying on the right said matrix will perform column operations on any matrix with n columns

```mathematica
In[63]:= With[
           {
             G = {{a, b, c}, {d, e, f}},
             S = RowSwp[IdentityMatrix[2], {1, 2}]},
             M = RowMul[IdentityMatrix[2], {1, k}]},
             A = RowAdd[IdentityMatrix[2], {1, 2}]},
           },
           G // MatrixForm // Print;
           S . G // MatrixForm // Print;
           M . G // MatrixForm // Print;
           A . G // MatrixForm // Print;
           G . S // MatrixForm // Print;
         ]
```

$$\text{Out[63]=} \quad \begin{pmatrix} a\ b\ c \\ d\ e\ f \end{pmatrix}$$

$$\text{Out[64]=} \quad \begin{pmatrix} d\ e\ f \\ a\ b\ c \end{pmatrix}$$

$$\text{Out[65]=} \quad \begin{pmatrix} ak\ bk\ ck \\ d\ \ e\ \ f \end{pmatrix}$$

$$\text{Out[66]=} \quad \begin{pmatrix} a+d\ b+e\ c+f \\ d\ \ \ \ e\ \ \ \ f \end{pmatrix}$$

Dot::dotsh: "Tensors {{a,b,c},{d,e,f}} and {{0,1},{1,0}} have incompatible shapes." »

$$\text{Out[67]=} \quad \{\{a, b, c\}, \{d, e, f\}\} . \{\{0, 1\}, \{1, 0\}\}$$

Notice that the final thing we tried was to multiply on the right instead of on the left, and *Mathematica* issued an error to tell us that the matrices could not be multiplied in that order, just as we already knew they couldn't.

```mathematica
In[68]:= With[
           {
             G = {{a, b, c}, {d, e, f}} // Transpose,
             S = RowSwp[IdentityMatrix[2], {1, 2}]},
             M = RowMul[IdentityMatrix[2], {1, k}]},
             A = RowAdd[IdentityMatrix[2], {1, 2}]},
           },
           G // MatrixForm // Print;
           G . S // MatrixForm // Print;
           G . M // MatrixForm // Print;
           G . A // MatrixForm // Print;
```

```
         S . G // MatrixForm // Print;
      ]
```

Out[68]= $\begin{pmatrix} a\ d \\ b\ e \\ c\ f \end{pmatrix}$

Out[69]= $\begin{pmatrix} d\ a \\ e\ b \\ f\ c \end{pmatrix}$

Out[70]= $\begin{pmatrix} ak\ d \\ bk\ e \\ ck\ f \end{pmatrix}$

Out[71]= $\begin{pmatrix} a\ a+d \\ b\ b+e \\ c\ c+f \end{pmatrix}$

Dot::dotsh: "Tensors {{0,1},{1,0}} and {{a,d},{b,e},{c,f}} have incompatible shapes." »

Out[72]= $\{\{0, 1\}, \{1, 0\}\}.\{\{a, d\}, \{b, e\}, \{c, f\}\}$

We now put all this together with a real (pun intended) matrix. In order to allow us to properly use the **S**, **M**, and **A** matrices more robustly, we create functions that allow us to choose the particular rows we wish to operate on, and multiples we wish to multiply with.

```
In[73]:= S[r1_ , r2_] := Block[{Id = IdentityMatrix[2]},
           RowSwp[Id, {r1, r2}]
         ]
         M[r1_ , r2_] := Block[{Id = IdentityMatrix[2]},
           RowMul[Id, {r1, r2}]
         ]
         A[r1_ , r2_] := Block[{Id = IdentityMatrix[2]},
           RowAdd[Id, {r1, r2}]
         ]
```

We add one more function to make our lives easier. Notice that and **RowAdd[r1, r2]** and consequently **A[r1, r2]** add row **r2** to row **r1**. Suppose now that we multiply row **r2** by some value, k say, before adding it to row **r1**. Suppose further that we multiply row **r2** by $1/k$ immediately after having added it to row **r2**. If this were to happen, we would have effectively added k lots of row **r2** to row **r1**.

We produce a new rule for the **A** function which performs precisely this combination operations. The rule will take a third argument which will be the value to multiply by. In other words, **A[r1, r2, k]** will add **k** times row **r2** to row **r1**. Remember that matrix multiplication is associative (that is $(A.B).C = A.(B.C)$), and that the elementary matrices must be multiplied on the left in order to perform row operations.

```
In[74]:= A[r1_ , r2_ , k_] := Block[{Id = IdentityMatrix[2]},
           RowMul[Id, {r2, 1 / k}] . RowAdd[Id, {r1, r2}].RowMul[Id, {r2, k}]
         ]
```

```
In[75]:= A[1, 2, k] . IdentityMatrix[2] // MatrixForm
```

Out[75]= $\begin{pmatrix} 1\ k \\ 0\ 1 \end{pmatrix}$

Now we perform row operation—via multiplication with elementary matrices—to reduce our matrix R into reduced row echelon form. Note that we use the `MatrixForm` function to show our outputs in the more mathematical matrix format, and this does not cause our matrix arithmetic to stop working, as it has done in the past. This is entirely due to the use of the `%` operator, which seems to somehow forget the format of the output on the previous line.[1]

In[76]:= `R = {{1, 2}, {3, 4}} // Transpose`

Out[76]= $\{\{1,3\},\{2,4\}\}$

In[77]:= `% // MatrixForm`

Out[77]= $\begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix}$

In[78]:= `A[2, 1, -2].% // MatrixForm`

Out[78]= $\begin{pmatrix} 1 & 3 \\ 0 & -2 \end{pmatrix}$

In[79]:= `M[2, -1 / -2].% // MatrixForm`

Out[79]= $\begin{pmatrix} 1 & 3 \\ 0 & 1 \end{pmatrix}$

In[80]:= `A[1, 2, -3] . % // MatrixForm`

Out[80]= $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$

We should note here that we have inadvertently found an inverse matrix for R. That inverse is the product of the three elementary matrices we used in the correct order. Because we were multiplying on the left the entire way, the correct order for multiplication is to start with the final elementary matrix, and work our way backwards.

In[81]:= `A[1, 2, -3] . M[2, -1 / -2] . A[2, 1, -2] // MatrixForm`
       `{%.R // MatrixForm, R.% // MatrixForm}`

Out[81]= $\begin{pmatrix} -2 & \frac{3}{2} \\ 1 & -\frac{1}{2} \end{pmatrix}$

Out[82]= $\left\{ \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \right\}$

Sure enough, if we ask *Mathematica* directly for the inverse of our matrix R then it gives us the matrix we just calculated.

In[83]:= `R // Inverse // MatrixForm`

Out[83]= $\begin{pmatrix} -2 & \frac{3}{2} \\ 1 & -\frac{1}{2} \end{pmatrix}$

This is no coincidence. It is necessarily the case that an invertible matrix will always become the identity matrix when changed into reduced row echelon form. Furthermore any invertible matrix can always be created by multiplying some sequence of elementary matrices together.

---

[1] The interested reader is encouraged to explore this difference by looking at the difference between, say, `{{1,2},{3,4}} // MatrixForm` followed by `% . %` and `M = {{1, 2}, {3, 4}} // MatrixForm` followed by `M . M`

Let's look at a three-dimensional example, and consequently the elementary matrices. First we should reassign the `S`, `M`, and `A` functions we used for the various flavors of elementary matrices earlier, so that they use the three-dimensional identity matrix instead of the two-dimensional one. We will only use the three-argument form of `A`, as we can always perform the earlier function by using a multiple of 1.

```
In[84]:= S[r1_ , r2_] := Block[{Id = IdentityMatrix[3]},
           RowSwp[Id, {r1, r2}]
         ]
         M[r1_ , r2_] := Block[{Id = IdentityMatrix[3]},
           RowMul[Id, {r1, r2}]
         ]
         A[r1_ , r2_ , k_] := Block[{Id = IdentityMatrix[3]},
           RowMul[Id, {r2, 1 / k}].RowAdd[Id, {r1, r2}].RowMul[Id, {r2, k}]
         ]
```

Let us have a look at our fresh new elementary matrices now. Note the use of the `/@` operator, which causes a function to be applied to all elements of a list, much the same as the `Map` function.

```
In[85]:= MatrixForm /@ {S[1, 2], S[1, 3], S[2, 3]}
         MatrixForm /@ {S[1, k], S[2, k], S[3, k]}
         MatrixForm /@ {A[1, 2, k], A[1, 3, k], A[2, 1, k], A[2, 3, k],
           A[3, 1, k], A[3, 2, k]}
```

$$\text{Out[85]}= \left\{ \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \right\}$$

$$\text{Out[86]}= \left\{ \begin{pmatrix} k & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 \\ 0 & k & 0 \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & k \end{pmatrix} \right\}$$

$$\text{Out[87]}= \left\{ \begin{pmatrix} 1 & k & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & k \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 \\ k & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & k \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ k & 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & k & 1 \end{pmatrix} \right\}$$

We now perform row reduction again on a real 3×3 matrix, just as we did above for our 2×2 matrix R. In fact, we even reuse the name. However, we proceed a little more quickly, and take multiple steps at a time, where it is obvious to do so.

```
In[88]:= R = {{1, 4 -7,}, {-2, 5, 8, }, {3, -6, 9}}
```

$$\text{Out[88]}= \{\{1, 4, -7\}, \{-2, 5, 8\}, \{3, -6, 9\}\}$$

```
In[89]:= % // MatrixForm
```

$$\text{Out[89]}= \begin{pmatrix} 1 & 4 & -7 \\ -2 & 5 & 8 \\ 3 & -6 & 9 \end{pmatrix}$$

```
In[90]:= A[2, 1, 2] . A[3, 1, -3] . % // MatrixForm
```

$$\text{Out[90]}= \begin{pmatrix} 1 & 4 & -7 \\ 0 & 13 & -6 \\ 0 & -18 & 30 \end{pmatrix}$$

```
In[91]:= M[2, -1 / 13] . % // MatrixForm
```

Out[91]= $\begin{pmatrix} 1 & 4 & -7 \\ 0 & 1 & -\frac{6}{13} \\ 0 & -18 & 30 \end{pmatrix}$

In[92]:= `A[1, 2, -4] . A[3, 2, 18].% // MatrixForm`

Out[92]= $\begin{pmatrix} 1 & 0 & -\frac{67}{13} \\ 0 & 1 & -\frac{6}{13} \\ 0 & 0 & \frac{282}{13} \end{pmatrix}$

In[93]:= `M[2, 13 / 282] . % // MatrixForm`

Out[93]= $\begin{pmatrix} 1 & 0 & -\frac{67}{13} \\ 0 & 1 & -\frac{6}{13} \\ 0 & 0 & 1 \end{pmatrix}$

In[94]:= `A[1, 3, 67 / 13] . A[2, 3, 6 / 13] . % // MatrixForm`

Out[94]= $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

And so it is that we may replay this sequence of moves to find the inverse of R.

In[95]:= `A[1, 3, 67/13] . A[2, 3, 6/13] . M[3, 13/282] . A[1, 2, -4] . A[3, 2, 18] .`
`M[2, 1/13] . A[2, 1, 2] . A[3, 1, -3] // MatrixForm`
`{%.R // MatrixForm, R.% // MatrixForm}`

Out[95]= $\begin{pmatrix} \frac{31}{94} & \frac{1}{47} & \frac{67}{282} \\ \frac{7}{47} & \frac{5}{47} & \frac{1}{47} \\ -\frac{1}{94} & \frac{3}{47} & \frac{13}{282} \end{pmatrix}$

Out[96]= $\left\{ \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \right\}$

For simplicity's sake, let's call this sequence of elementary matrices

$$E_1, E_2, E_3, E_4, E_5, E_6, E_7, E_8$$

(because it's easier to write and type). We know that

$$E_1 E_2 E_3 E_4 E_5 E_6 E_7 E_8 R = I \text{ or equivalently } E_1 E_2 E_3 E_4 E_5 E_6 E_7 E_8 = R^{-1}$$

Now, observe that every elementary matrix is invertible. This is a direct consequence of the fact that the row operations themselves are invertible processes. We can, very simply, undo any row operation we perform as follows:

Operation	**Inverse**
Swap rows a and b	Swap rows a and b
Multiply a row a by a constant k	Multiply a row a by the constant $\frac{1}{k}$
Add k times row a to row b	Add $-k$ times row a to row b

Note that the final inverse process implies that both of our two variants of the row add operation are invertible, inasmuch as adding a row to another row is equivalent to adding 1 times the one row to the other row.

We know, therefore, that all of the E_i matrices are invertible, and we already knew that the R matrix are invertible. Recalling that $(AB)^{-1} = B^{-1}A^{-1}$, then it must be the case that

$$R = E_8^{-1} E_7^{-1} E_6^{-1} E_5^{-1} E_4^{-1} E_3^{-1} E_2^{-1} E_1^{-1}$$

Checking this in *Mathematica* for our above example:

In[97]:= **A[3, 1, 3] . A[2, 1, -2] . M[2, 13] . A[3, 2, -18] . A[1, 2, 4] .**
         M[3, 282/13] . A[2, 3, -6/13] . A[1, 3, -67/13] // MatrixForm
         % == R

Out[97]= $\begin{pmatrix} 1 & 4 & -7 \\ -2 & 5 & 8 \\ 3 & -6 & 9 \end{pmatrix}$

Out[98]= True

In fact, this will always happen for an invertible matrix. We have the following theorem.

Theorem 2. *Let A be a square matrix. The following are equivalent.*

1. A is invertible.
2. The linear system $Ax = 0$ has only the trivial solution (i.e., $x = 0$).
3. The reduced row echelon form of A is the identity matrix.
4. A may be expressed as a product of elementary matrices.

Proof. Observe that if A is invertible, then

$$Ax = 0 \implies A^{-1}(Ax) = A^{-1}0 \implies (A^{-1}A)x = 0 \implies x = 0$$

showing that the x vector must be the zero vector.

If $x = 0$ is the only solution to the linear system $Ax = 0$ then the augmented matrix $[A|0]$ must have reduced row echelon form $[I|0]$ (corresponding to the solution $x = 0$). And so the reduced row echelon form of A is I.

If the I is the reduced row echelon form of A then it must be the case that there is some set of elementary matrices such that

$$E_1 E_2 \cdots E_i A = I \implies A = E_i^{-1} \cdots E_2^{-1} E_1^{-1}$$

showing that A is expressible as the product of elementary matrices.

And finally if A is expressible as the product of elementary matrices, then A must be invertible, inasmuch as the elementary matrices are all invertible, so

$$A^{-1} = E_1 E_2 \cdots E_i$$

$\square$

It is precisely this theorem that allows us to find the inverse of an invertible matrix A by using row operations on the augmented matrix $[A|I]$ where I is the identity matrix. The sequence of row operations which transforms a matrix, M say, into the identity matrix is precisely the same sequence of row operations which turns the identity matrix into the inverse of M. Furthermore, the theorem guarantees that this happens when and only when the matrix is invertible. By constructing the augmented matrix, we are effectively performing the row operations on both the matrix M and the identity matrix simultaneously.

In[99]:= **Join[R, IdentityMatrix[3], 2] // MatrixForm**
         RowReduce[%] // MatrixForm

Out[99]= $\begin{pmatrix} 1 & 4 & -7 & 1 & 0 & 0 \\ -2 & 5 & 8 & 0 & 1 & 0 \\ 3 & -6 & 9 & 0 & 0 & 1 \end{pmatrix}$

Out[100]= $\begin{pmatrix} 1 & 0 & 0 & \frac{31}{94} & \frac{1}{47} & \frac{67}{282} \\ 0 & 1 & 0 & \frac{7}{47} & \frac{5}{47} & \frac{1}{47} \\ 0 & 0 & 1 & -\frac{1}{94} & \frac{3}{47} & \frac{13}{282} \end{pmatrix}$

Remember, however, that in practice there's no point to performing the row operations with the elementary matrices, as directly performing row operations on a matrix is far faster (both for a computer and for a human). However, the existence of the elementary matrices allows us to prove the above theorem, whose utility is quite significant indeed.

3.2 Vector Spaces

3.2.1 Vector Spaces

Until now we have been using vectors without really saying what they are. However, before we can do much more with linear algebra, we need to define exactly what a vector, or rather a *vector space* is.

If V is a set of objects, and $\mathbb{F}$ is a field, then we call V a vector space—or, more accurately, a vector space over $\mathbb{F}$—if the elements of V and $\mathbb{F}$ interact as described below. Note that fields for our purposes are nearly always real numbers $\mathbb{R}$ and sometimes complex numbers $\mathbb{C}$, although other fields do exist and may be used in place of $\mathbb{F}$. The elements of V are, not surprisingly, the vectors, and the elements of $\mathbb{F}$ are referred to as *scalars*.

For V to be a vector space (over the field $\mathbb{F}$, remember), then there must be an addition operator on V (i.e., a way of adding any two vectors in such a way as to always result in a vector) and a scalar multiplication operation (i.e., a way of multiplying any scalar and any vector in such a way as to always result in a vector). In addition, the following axioms must hold for $u, v, w \in V$ and $a, b \in \mathbb{F}$

1. $u + v = v + u$ (Commutativity of vector addition).
2. $(u + v) + w = u + (v + w)$ (Associativity of vector addition).
3. There is an element $\mathbf{0} \in V$ such that $\mathbf{0} + u = u$ (Additive identity, or *zero vector*).
4. There is an element $-v \in V$ such that $v + (-v) = 0$ (Additive inverse).

Anybody who has studied abstract algebra should recognize that the above four axioms show that V must be an Abelian group. Continuing on:

5. $a(u + v) = au + av$ (Distributive property).
6. $(a + b)u = au + bu$ (Distributive property).
7. $(ab)u = a(bu)$ (Associativity of scalar multiplication).
8. $1v = v$ where $1 \in \mathbb{F}$ is the multiplicative identity.

Scalar multiplication and vector addition interact in a very similar fashion to the way the more familiar addition and multiplication of real or complex numbers interact.

We should already be familiar with some vector spaces, even if we have never thought of them in such terms. The points of the Cartesian plane can be thought of as vectors, and indeed we have used *Mathematica* to manipulate or produce them earlier. To be

specific, the set V in this case is the set $\mathbb{R}^2$ or $\mathbb{R} \times \mathbb{R}$ which consists of ordered pairs of real numbers ($\mathbb{R}^2 := \{(x, y) \mid x \in \mathbb{R} \text{ and } y \in \mathbb{R}\}$) and the field is the real numbers $\mathbb{R}$. It is straightforward to check that these two sets satisfy all the axioms above.

In a similar fashion we can see that the points in three-dimensional real space, or $\mathbb{R}^3$ also form a vector space over the real numbers. In fact, any n-dimensional real space $\mathbb{R}^n := \{(x_1, \ldots, x_n) \mid x_i \in \mathbb{R}\}$ forms a vector space over the real numbers, with vector addition and scalar multiplication working as it does for $\mathbb{R}^2$ and $\mathbb{R}^3$.

We can extend this idea a little further by replacing the real numbers with complex numbers and end up with $\mathbb{C}^n := \{(z_1, \ldots, z_n) \mid z_i \in \mathbb{C}\}$ with vector addition and scalar multiplication working in precisely the same way as with the real case (excepting, of course, that we perform complex multiplication). Let's have a quick look at that in *Mathematica*.

```
In[101]:= With[{u = {1 + 2 I, 2 + 3 I, 3 + 4 I}, v = {5 + 6 I, 6 + 7 I, 7 + 8 I}},
           u + v // Print;
           (9 + 10 I) * u // Print;
          ]
```

Out[101]= $\{6 + 8i, 8 + 10i, 10 + 12i\}$

Out[102]= $\{-11 + 28i, -12 + 47i, -13 + 66i\}$

We may even construct vector spaces of polynomials of fixed degree, or matrices of a fixed size. It is left as an exercise for the reader to verify the vector space axioms for both of these cases. We denote the space of $m \times n$ matrices as $M_{m,n}(\mathbb{F})$, and the space of degree n polynomials as $P_n(\mathbb{F})$ (both over the field $\mathbb{F}$).

We now demonstrate a useful correlation between the vector space $P_n(\mathbb{F})$ and the vector space $\mathbb{F}^{n+1}$. Let $p_0 + p_1 x + \cdots + p_n x^n \in P_n(\mathbb{F})$ (so $p_i \in \mathbb{F}$). We take the vector $(p_0, p_1, \ldots, p_n) \in \mathbb{F}^{n+1}$, and consider it as being equivalent. We can freely change between these forms; given the vector we can easily construct the equivalent polynomial, and given the polynomial we can easily construct the vector.

```
In[103]:= With[
            {
              p = p0 + p1 * x + p2 * x^2 + p3 * x^3 + p4 * x^4,
              q = q0 + q1 * x + q2 * x^2 + q3 * x^3 + q4 * x^4,
              u = {p0, p1, p2, p3, p4},
              v = {q0, q1, q2, q3, q4}
            },
            k * p // Expand // Print;
            k * p // Expand // Print;
            Collect[p + q, x] // Print;
            u + v // Print;
          ]
```

Out[103]= $k\,p0 + k\,p1\,x + k\,p2\,x^2 + k\,p3\,x^3 + k\,p4\,x^4$

Out[104]= $\{k\,p0, k\,p1, k\,p2, k\,p3, k\,p4\}$

Out[105]= $p0 + q0 + (p1 + q1)x + (p2 + q2)x^2 + (p3 + q3)x^3 + (p4 + q4)x^4$

Out[106]= $\{p0 + q0, p1 + q1, p2 + q2, p3 + q3, p4 + q4\}$

We have demonstrated that vector addition and polynomial addition produce equivalent results, as do vector and polynomial scaling. Note that for the above example we used $n = 4$, however this idea works for any n (which should hopefully be clear,

given what we know about vectors and polynomials). We have, in effect, shown that polynomials can be thought of as $n + 1$ vectors, and that the two vector spaces can be interchangeably used.

This is all a special case of a result we look at a little later; that any finite-dimensional vector space with dimension n can be thought of as $\mathbb{F}^n$. This is of key use to us with *Mathematica*.

Incidentally, converting between forms in *Mathematica* is quite straightforward. We simply declare two functions, one which takes a polynomial as an argument to produce a vector, and one which takes a vector as an argument and produces a polynomial.

```
In[107]:= f[p_ /; PolynomialQ[p, x] && Exponent[p, x] <= 4] :=
            Table[Coefficient[p, x, n], {n, 0, 4}]
          f[{p0_, p1_, p2_, p3_, p4_}] :=
            p0_ + p1_ * x + p2_ * x^2 + p3_ * x^3 + p4_ * x^4

In[108]:= f[1 + x + 3 x^3]
          f[{0, 4, 0, 5, 0}]

Out[108]= {1, 1, 0, 3, 0}

Out[109]= 4x + 5x^3
```

We have used the **PolynomialQ** function to test for an argument being a polynomial, and the **Exponent** function to find the degree of the polynomial. To construct the vector, we have used the **Coefficient** function which will return the coefficient in the first argument of the unknown given in the second argument to the power of the value of the third argument. That is, **Coefficient[p, x, n]** will return the coefficient of x^n in the polynomial p.

Note that we have specifically written both functions for the case of $n = 4$. This allows us to use polynomials of lower degree and still get the correct sized vector. However, this comes at the expense of having to write a new function if we wish to convert between different vector spaces. In practice there are many vector spaces we might wish to convert between, and so we are likely going to have to write specific conversion functions between the spaces we are using, so doing so here does not really hamper us much. We will see and use this technique again several times before the end of the chapter.

3.2.2 Linear Combinations

Let V be a vector space, and let $v_1, v_2, \ldots, v_k \in V$. A *linear combination* of these vectors is a new vector of the form $v = a_1 v_1 + a_2 v_2 + \cdots + a_k v_k$ where the a_i are scalars (i.e., $a_i \in \mathbb{F}$). That $v \in V$ follows directly from the axioms of a vector space.

We may wish to consider the set of all possible linear combinations of a set of vectors, which we call the *span* of those vectors, for example, $\mathrm{span}(\{v_1, v_2, \ldots, v_k\})$ which is equal to $\{a_2 v_2 + \cdots + a_k v_k : a_i \in \mathbb{F}\}$.

Let $S \subset V$ be a nonempty and finite set of vectors. A natural question arises as to how we may tell whether a vector v is a linear combination of the vectors in S. Equivalently this same question may be phrased as whether a vector is in the span of S. For example, is the vector $(7, 8, 9)$ a linear combination of the vectors $(1, 2, 3)$ and $(4, 5, 6)$?

For simplicity, we consider only the case of the vector spaces $\mathbb{F}^n$. Think of what a linear combination would mean in this case. If v is a linear combination of vectors in S then

$$a_1 \begin{bmatrix} v_{1_1} \\ \vdots \\ v_{1_n} \end{bmatrix} + a_2 \begin{bmatrix} v_{2_1} \\ \vdots \\ v_{2_n} \end{bmatrix} + \cdots + a_k \begin{bmatrix} v_{k_1} \\ \vdots \\ v_{k_n} \end{bmatrix} = v = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix}$$

which is equivalent to

$$\begin{bmatrix} v_{1_1} & v_{2_1} & \cdots & v_{k_1} \\ \vdots & \vdots & \ddots & \vdots \\ v_{1_n} & v_{2_n} & \cdots & v_{k_n} \end{bmatrix} \begin{bmatrix} a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix}$$

which, in turn, is a system of linear equations, just as we dealt with in Section 3.1.2. If the linear system has at least one solution, then the vector v is a linear combination of the vectors in S. Indeed if there are many solutions, then there are many ways to represent v as such a linear combination, but this does not detract in any way from the fact that it is a linear combination. If, however, there is no solution to the system, then v cannot be written as a linear combination of the vectors in S.

This will be true for any set of vectors $v_1, \ldots, v_k$, as long as our vector space is $\mathbb{F}^n$. We know how to solve such systems, so we can now easily answer our example question from above.

```
In[110]:= LinearSolve[{{1, 2, 3}, {4, 5, 6}} // Transpose, {7, 8, 9}]
```
Out[110]= $\{-1, 2\}$

It would seem that there is a linear combination here, with coefficients -1 and 2. We can, of course, check this easily.

```
In[111]:= -1 * {1, 2, 3} + 2 * {4, 5, 6}
```
Out[111]= $\{7, 8, 9\}$

This gives us a new way to look at simultaneous linear equations as well. Recall from Section 3.1.2 our first (and simplest) example $x + y = 2$ and $2x + y = 3$ which had solution $x = 1, y = 1$. We constructed from this the linear system

$$\begin{bmatrix} 1 & 1 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$$

which we may now, equivalently, consider as finding $(2, 3)$ as a linear combination of $(1, 2)$ and $(1, 1)$. The solution $1, 1$ clearly works for this interpretation.

```
In[112]:= ({{1, 2}, {1, 1}} // Transpose).{1, 1}
          1 * {1, 2} + 1 * {1, 1}
```
Out[112]= $\{2, 3\}$

Out[113]= $\{2, 3\}$

We may use this same linear system technique with polynomials, as well, because we established a correspondence between polynomials and n-tuples (meaning vectors in $\mathbb{F}^n$). Let us then ascertain whether $-4x^4 + x^3 + 5x^2 + 3x - 13$ is a linear combination of $x^4 + 3x^3 + 7x^2 - 6$, $2x^4 + 4x^2 - 5x + 2$ and $5x^3 - 3x^2 + 12x - 5$. To do this, we need to consider them as the vectors $(-13, 3, 5, 1, -4)$, $(-6, 0, 7, 3, 1)$, $(2, -5, 4, 0, 2)$, and

$(-5, 12, -3, 5, 0)$, which we can do in *Mathematica* with the use of the **f** function, we defined at the end of Section 3.2.1.[2]

```
In[114]:=  p1 = x^4 + 3 x^3 + 7 x^2 - 6
           p2 = 1 x^4 + 4 x^2 - 5 x + 2
           p3 = 5 x^3 - 3 x^2 + 12 x - 5
           q = 4 x^4 + x^3 + 5 x^2 + 3 x - 13
```

Out[114]= $-6 + 7x^2 + 3x^3 + x^4$

Out[115]= $2 - 5x + 4x^2 + 2x^4$

Out[116]= $-5 + 12x - 3x^2 + 5x^3$

Out[117]= $-13 + 3x + 5x^2 + x^3 - 4x^4$

```
In[118]:=  M = {f[p1], f[p2] f[p3]} // Transpose
           v = f[q]
```

Out[118]= $\{\{-6, 2, -5\}, \{0, -5, 12\}, \{7, 4, -3\}, \{3, 0, 5\}, \{1, 2, 0\}\}$

Out[119]= $\{-13, 3, 5, 1, -4\}$

Note that we have declared these variables globally, as we wish to use them several times before we have been finished. Unfortunately, it is a little difficult to see our polynomial coefficients in that matrix when it is in list form, but were we to use the **MatrixForm** function, then the **M** variable would not be recognized as a matrix by *Mathematica*'s matrix manipulation functions. We should be confident that our conversion function, **f**, will have produced the correct vectors. Nonetheless, it will be illustrative to view the matrix/vector problem we are solving, and we may double-check the matrix while we're at it.

```
In[120]:=  (M // MatrixForm).({x1, x2, x3} // MatrixForm) == (v // MatrixForm)
```

Out[120]= $\begin{pmatrix} -6 & 2 & -5 \\ 0 & -5 & 12 \\ 7 & 4 & -3 \\ 3 & 0 & 5 \\ 1 & 2 & 0 \end{pmatrix} . \begin{pmatrix} x1 \\ x2 \\ x3 \end{pmatrix} == \begin{pmatrix} -13 \\ 3 \\ 5 \\ 1 \\ -4 \end{pmatrix}$

We should be able to see our polynomial coefficients in there. Remember that $p_0 + p_1\, x + p_2\, x^2 + p_3\, x^3 + p_4\, x^4$ corresponds to the vector vector $(p_0, p_1, p_2, p_3, p_4)$, and that we have transposed these vectors. As such, the vectors are columns in the matrix, where the top-most elements are the constant, and the bottom-most elements are the x^4 coefficients.

We may now solve the problem

```
In[121]:=  LinearSolve[M, v]
```

Out[121]= $\{2, -3, -1\}$

and check the solution

```
In[122]:=  2 * p1 - 3 * p2 - p3
```

Out[122]= $5 - 12x + 3x^2 - 5x^3 + 2\left(-6 + 7x^2 + 3x^3 + x^4\right) - 3\left(2 - 5x + 4x^2 + 2x^4\right)$

---

[2] If the reader is not following the book sequentially, then that function may need to be manually defined now, so that the following commands will work. The reader in this position should find the definition from Section 3.2.1 and apply it to their *Mathematica* notebook file before proceeding.

In[123]:= **Expand[%]**

Out[123]= $-13 + 3x + 5x^2 + x^3 - 4x^4$

and there we have it.

To close with, we note that we have defined a number of global variables, and so we should make sure we clean up after ourselves now.

In[124]:= **ClearAll["Global`*"]**

3.2.3 Linear Independence

Along with the question of whether a vector may be expressed as a linear combination of other vectors comes a related but different question. If we have a nonempty and finite set $S \subset V$, can we express one of them (any one) as a linear combination of the others? If we can, then we say that the set S is *linearly dependent*. Conversely, if we cannot express any of the vectors in S as a linear combination of the others, then we say the set is *linearly independent*.

This is different from the question of whether an arbitrary vector $v \in V$ is in the span of S. The difference here is that previously v could have been any vector at all, whereas now we are asking whether vectors in the subset S are linear combinations of other vectors in the same subset which is a more specific question. Another way to think about this idea is whether we can remove vectors from S and still have the same span.

Another, equivalent, formulation of linear (in)dependence is to say that a finite set $\{v_1, \ldots, v_n\}$ of vectors is linearly independent if and only if the only solution to $a_1v_1 + a_2v_2 + \cdots + a_nv_n = 0$ is the trivial solution $a_1 = a_2 = \cdots = a_n = 0$. To see that this is an equivalent notion, suppose that $v_1 \neq 0$ can be expressed as a linear combination of the other vectors,

$$v_1 = a_2v_2 + \cdots + a_nv_n \implies 0 = a_2v_2 + \cdots + a_nv_n - v_1$$

showing that the zero vector can be obtained as a nontrivial linear combination of the vectors. If v_2 or any other v_i can be expressed as a linear combination of the other vectors, then we may similarly construct the zero-vector as a non-trivial linear combination. Conversely, now suppose that

$$a_1v_1 + a_2v_2 + \cdots + a_nv_n = 0 \text{ where } a_1 \neq 0$$

Then

$$v_1 = \left(-\frac{a_2}{a_1}\right)v_2 + \cdots + \left(-\frac{a_n}{a_1}\right)v_n$$

showing that v_1 can be written as a linear combination of the other vectors. If v_2 or any other v_i is nonzero, then we can similarly show that it can be written as a linear combination of the other vectors.

We have now shown that the two notions of linear independence (and, thus, linear dependence) are equivalent.

This, in turn, allows us to link these linear independence notions to systems of linear equations and matrix invertibility. If we wish to see if a set, S, of vectors is linearly independent, we need to see if the zero vector can be expressed as a linear combination

of the vectors in S. This, as we have seen, can be accomplished by solving a linear system.

For example, we have a look at the vectors $(-1, 2, 3, 4)$, $(1, -2, 3, 4)$, $(1, 2, -3, 4)$, and $(1, 2, 3, -4)$, and see if they are a linearly independent set. We do this by solving the linear system

$$\begin{bmatrix} -1 & 1 & 1 & 1 \\ 2 & -2 & 2 & 2 \\ 3 & 3 & -3 & 3 \\ 4 & 4 & 4 & -4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

and seeing whether the solution is the zero vector, or not.

```
In[125]:= v = {{-1, 2, 3, 4}, {1, -2, 3, 4}, {1, 2, -3, 4}, {1, 2, 3, -4}}
```

Out[125]= $\{\{-1, 2, 3, 4\}, \{1, -2, 3, 4\}, \{1, 2, -3, 4\}, \{1, 2, 3, -4\}\}$

```
In[126]:= With[{A = v // Transpose},
          A // MatrixForm // Print;
          LinearSolve[A, {0, 0, 0, 0}] // Print;
        ]
```

Out[126]= $\begin{pmatrix} -1 & 1 & 1 & 1 \\ 2 & -2 & 2 & 2 \\ 3 & 3 & -3 & 3 \\ 4 & 4 & 4 & -4 \end{pmatrix}$

Out[127]= $\{0, 0, 0, 0\}$

The solution is the trivial solution, and so it would seem that the the vectors are linearly independent. However, the **LinearSolve** function is not guaranteed to give us all solutions; it will just give us *a* solution. However, the trivial solution is always a possible solution. so we aren't really any the wiser here.

Fortunately, *Mathematica* provides a function for specifically answering the question of when $Mx = 0$ that we are asking here. The function in question is named **NullSpace**. The *null space* of a matrix, M say, is the set of all vectors with the property that $Mv = 0$. The **NullSpace** function computes a set of vectors which span the null space.

```
In[128]:= NullSpace[v // Transpose]
```

Out[128]= $\{\}$

We are presented with the empty set here, which indicates that the null space is trivial. That is the only vector in the null space is the zero vector, which is always in the null space. *Mathematica* omits it for simplicity.

If we add a new vector into the mix, $(1, 2, 3, 4)$, we can ask the same question of linear independence about the new set.

```
In[129]:= v ~ Join ~ {1, 2, 3, 4}
```

Out[129]= $\{\{-1, 2, 3, 4\}, \{1, -2, 3, 4\}, \{1, 2, -3, 4\}, \{1, 2, 3, -4\}, \{1, 2, 3, 4\}\}$

```
In[130]:= With[{A = v // Transpose},
          A // MatrixForm // Print;
          LinearSolve[A, {0, 0, 0, 0}] // Print;
          NullSpace[A] // Print;
        ]
```

Out[130]= $\begin{pmatrix} -1 & 1 & 1 & 1 & 1 \\ 2 & -2 & 2 & 2 & 2 \\ 3 & 3 & -3 & 3 & 3 \\ 4 & 4 & 4 & -4 & 4 \end{pmatrix}$

Out[131]= $\{0,0,0,0,0\}$

Out[132]= $\{\{-1,-1,-1,-1,2\}\}$

This time we have a linearly independent set. Note that the **LinearSolve** function still returned the zero vector as a solution to the linear system. Clearly **NullSpace** is a much better choice for ascertaining whether vectors are linearly dependent or independent. We can read straight from the solution that $-v_1 - v_2 - v_3 - v_4 + 2v_5 = 0$ or, equivalently, $2v_5 = v_1 + v_2 + v_3 + v_4$. However, recall that is is the span of the vectors returned by **NullSpace** that forms the null space of the matrix. As such we should expect that $M \cdot t(-1,-1,-1,-1,2) = 0$, or in other words that $2tv_5 = tv_1 + tv_2 + tv_3 + tv_4$.

```
In[133]:= (v // Transpose) . (t * {-1, -1, -1, -1, 2})
          t * v[[1]] + t * v[[2]] + t * v[[3]] + t * v[[4]]
```

Out[133]= $\{0,0,0,0\}$

Out[134]= $\{2t, 4t, 6t, 8t\}$

When looking at the question of linear dependence as a linear system problem, in the case of $\mathbb{F}^n$, if we have more than n vectors, then we have a linear system with an infinite number of solutions. This linear system would correspond to a set of simultaneous equations with more unknowns than equations. As such, any set of more than n vectors from the vector space $\mathbb{F}^n$ must be linearly dependent.

Furthermore, if we have exactly n vectors then we have a square matrix, and so we know by the theorem in Section 3.1.3 that if the only solution to $Ax = 0$ is the trivial solution, then the matrix is invertible, and vice-versa. Inasmuch as the column vectors of a matrix are linearly independent if and only if the only solution to $Ax = 0$ is the trivial solution, we now have a new equivalence to add to our theorem. That is, the column vectors of a square matrix being linearly independent is equivalent to the matrix being invertible, and hence is equivalent to all the other things which we already knew were equivalent to a matrix being invertible.

We now look at this same phenomenon with polynomials and their correspondence to n-tuples which we have been using. We start by using the same polynomials from Section 3.2.2, as we already know the answer to the linear system. Recall that $2p_1 - 3p_2 - p_3 = q$ or, in other words $2p_1 + 3p_2 - p_3 - q = 0$ and so the set $\{p_1, p_2, p_3, q\}$ is linearly dependent.

```
In[135]:= With[
            {
              p1 = x^4 + 3 x^3 + 7 x^2 - 6,
              p2 = 1 x^4 + 4 x^2 - 5 x + 2,
              p3 = 5 x^3 - 3 x^2 + 12 x - 5,
              q = 4 x^4 + x^3 + 5 x^2 + 3 x - 13
            },
            2 * p1 - 3 * p2 - p3 - q
          ]
```

Out[135]= $18 - 15x - 2x^2 - 6x^3 + 4x^4 + 2\left(-6 + 7x^2 + 3x^3 + x^4\right) - 3\left(2 - 5x + 4x^2 + 2x^4\right)$

```
In[136]:= Expand[%]
```

Out[136]= 0

Let's try some larger polynomials. We use $90x^5 + 80x^4 + 19x^3 + 88x^2 - 82x - 70$, $41x^5 + 91x^4 + 29x^3 + 70x^2 - 32x - 1$ and $52x^5 - 13x^4 + 82x^3 + 72x^2 + 42x + 18$. Inasmuch as we're using now using degree 5 polynomials, we will need to change our conversion function to handle the larger polynomials and vectors. Also, we cleared all of our previous definitions at the end of the last subsection.

```
In[137]:= f[p_ /; PolynomialQ[p, x] && Exponent[p, x] <= 5] :=
          Table[Coefficient[p, x, n], {n, 0, 5}]
          f[{p0_, p1_, p2_, p3_, p4_, p5_}] :=
          p0_ + p1_ * x + p2_ * x^2 + p3_ * x^3 + p4_ * x^4 + p5_ * x^5
```

We are going to do some exploration of these concepts, and so we are going to want to use the polynomials several times. As such, we continue to declare the variables globally. It will be simpler if we keep all the polynomials in a list, so this we do.

```
In[138]:= p = {
          90 x^5 + 80 x^4 + 19 x^3 + 88 x^2 - 82 x - 70,
          41 x^5 + 91 x^4 + 29 x^3 + 70 x^2 - 32 x - 1,
          52 x^5 - 13 x^4 + 82 x^3 + 72 x^2 + 42 x + 18
          }
```

Out[138]= $\{ -70 - 82x + 88x^2 + 19x^3 + 80x^4 + 90x^5,$

$-1 - 32x + 70x^2 + 29x^3 + 91x^4 + 41x^5,$

$18 + 42x + 72x^2 + 82x^3 - 13x^4 + 52x^5 \}$

```
In[139]:= M := (f /@ p) // Transpose
```

Note that we have been clever here, and have used a delayed definition for our matrix M. By the use of the map operator we turn our list of polynomials into a list of vectors, and hence a matrix. Of course, this new matrix will have the converted vectors as rows when we want them as columns, so we transpose the matrix, just as we have done with our previous matrices. The delayed definition means that our matrix, M, will always use the current contents of the list p at the time of use. The reason for this will become apparent shortly. For now, we should have a look at our matrix, and see if the vectors, and hence the polynomials, are linearly dependent.

```
In[140]:= M // MatrixForm
          NullSpace[M]
```

Out[140]= $\begin{pmatrix} -70 & -1 & 18 \\ -82 & -32 & 42 \\ 88 & 70 & 72 \\ 19 & 29 & 82 \\ 80 & 91 & -13 \\ 90 & 41 & 52 \end{pmatrix}$

Out[141]= $\{\}$

We can see that our polynomials are linearly independent. We now add two more polynomials to this list, $-19\,x^5 - 68\,x^4 - 89\,x^3 + 66\,x + 77$ and $-80\,x^5 - 19\,x^4 - 62\,x^3 + 81\,x^2 + 22\,x + 50$.

In[142]:= `p = p ~Join~ {`
          `-19 x^5 - 68 x^4 - 89 x^3 + 66 x + 77`
          `-80 x^5 - 19 x^4 - 62 x^3 + 81 x^2 + 22 x + 50`
          `}`

Out[142]= $\{ -70 - 82x + 88x^2 + 19x^3 + 80x^4 + 90x^5,$

$-1 - 32x + 70x^2 + 29x^3 + 91x^4 + 41x^5,$

$18 + 42x + 72x^2 + 82x^3 - 13x^4 + 52x^5, 77 + 66x - 89x^3 - 68x^4 - 19x^5,$

$50 + 22x + 81x^2 - 62x^3 - 19x^4 - 80x^5 \}$

The delayed definition of `M` now begins to work for us. We have modified `p`, and so when next we use `M`, it will use the new contents of `p`.

In[143]:= `M // MatrixForm`
          `NullSpace[M]`

Out[143]= $\begin{pmatrix} -70 & -1 & 18 & 77 & 50 \\ -82 & -32 & 42 & 66 & 22 \\ 88 & 70 & 72 & 0 & 81 \\ 19 & 29 & 82 & -89 & -62 \\ 80 & 91 & -13 & -68 & -19 \\ 90 & 41 & 52 & -19 & -80 \end{pmatrix}$

Out[144]= $\{\}$

Our collection of vectors is still linearly independent. We add one more polynomial $162\,x^5 - 73\,x^4 + 45\,x^3 + 9\,x^2 + 36\,x - 24$. Note that this brings us to six vectors, each with six elements, and so our matrix `M` will be square. As such, we may check the linear dependence of the column vectors using any of the equivalences of an invertible matrix, as well as with the methods we have been using previously.

In[145]:= `p = p ~Join~ {162 x^5 - 73 x^4 + 45 x^3 + 9 x^2 + 36 x - 24}`

Out[145]= $\{ -70 - 82x + 88x^2 + 19x^3 + 80x^4 + 90x^5,$

$-1 - 32x + 70x^2 + 29x^3 + 91x^4 + 41x^5,$

$18 + 42x + 72x^2 + 82x^3 - 13x^4 + 52x^5, 77 + 66x - 89x^3 - 68x^4 - 19x^5,$

$50 + 22x + 81x^2 - 62x^3 - 19x^4 - 80x^5,$

$-24 + 36x + 9x^2 + 45x^3 - 73x^4 + 162x^5 \}$

In[146]:= `M // MatrixForm`
          `Det[M]`

Out[146]= $\begin{pmatrix} -70 & -1 & 18 & 77 & 50 & -24 \\ -82 & -32 & 42 & 66 & 22 & 36 \\ 88 & 70 & 72 & 0 & 81 & 9 \\ 19 & 29 & 82 & -89 & -62 & 45 \\ 80 & 91 & -13 & -68 & -19 & -73 \\ 90 & 41 & 52 & -19 & -80 & 162 \end{pmatrix}$

Out[147]= 0

We have used the `Det` function to compute the determinant of our matrix as 0, which tells us that the matrix is not invertible, which in turn tells us that the column vectors are linearly dependent. This, in turn, tells us that we have a linearly dependent set of polynomials. We go ahead and compute the null space anyway.

In[148]:= **NullSpace[M]**

Out[148]= $\{\{-1, 1, -1, -1, 1, 1\}\}$

We may read directly from this solution that $-t \cdot p_1 + t \cdot p_2 - t \cdot p_3 - t \cdot p_4 + t \cdot p_5 = -t \cdot p_6$.

In[149]:= **-t * p[[1]] + t * p[[2]] - t * p[[3]] - t * p[[4]] + t * p[[5]]**

Out[149]= $t \left(50 + 22x + 81x^2 - 62x^3 - 19x^4 - 80x^5\right) -$
$\qquad t \left(77 + 66x - 89x^3 - 68x^4 - 19x^5\right) +$
$\qquad t \left(-1 - 32x + 70x^2 + 29x^3 + 91x^4 + 41x^5\right) -$
$\qquad t \left(18 + 42x + 72x^2 + 82x^3 - 13x^4 + 52x^5\right) -$
$\qquad t \left(-70 - 82x + 88x^2 + 19x^3 + 80x^4 + 90x^5\right)$

In[150]:= **Expand[%]**

Out[150]= $24t - 36tx - 9tx^2 - 45tx^3 + 73tx^4 - 162tx^5$

In[151]:= **% == -t * p[[6]]**

Out[151]= $24t - 36tx - 9tx^2 - 45tx^3 + 73tx^4 - 162tx^5 ==$
$\qquad - t \left(-24 + 36x + 9x^2 + 45x^3 - 73x^4 + 162x^5\right)$

In[152]:= **Simplify[%]**

Out[152]= True

As ever when we have made global definitions, it is prudent to clean up after ourselves.

In[153]:= **ClearAll["Global`*"]**

3.2.4 Basis and Dimension

Having discussed linear combinations and linear independence, we now come to the notion of the *basis* for a vector space. If we have a subset $S \subset V$ of vectors, then it is possible that the span of S (the collection of all linear combinations of the vectors) may indeed be the entire vector space V. If, in addition, S is linearly independent, then we say that S is a *basis* of V.

Because a basis, B say, must be linearly independent, then no vector in B can be made from a linear combination of the other vectors, and so if we remove a basis vector, the span of the resultant set will no longer be all of V. Furthermore, if we add any vector, $v \in V$ to B, then the new set will no longer be linearly independent (because v will be able to be written as a linear combination of the other basis vectors). As such we can think of a basis as the smallest set of vectors that span a given vector space.

We should be familiar with some standard bases. For example, $\mathbb{F}^3$ has basis $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$. In fact $\mathbb{F}^n$ has a standard basis of n vectors

$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \dots, \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$$

An interesting result, which we do not prove, is that any basis for a given vector space will always have the same number of elements. This gives rise to the notion of *dimension* of a vector space. The dimension of a vector space V is the number of elements in any basis for the space.

If we look to polynomial spaces again, we may see that the space of all degree-n polynomials has a basis consisting of the $n+1$ elements $\{1, x, x^2, \ldots, x^n\}$ or, alternatively, $\{x^k : 0 \leq k \leq n \text{ and } k \in \mathbb{Z}\}$. In fact, what we have been doing with our correspondence between polynomials and n-tuples has been to establish a correspondence between their basis vectors. More correctly, we have established a correspondence between degree-n polynomials and $(n+1)$-tuples.

In fact, this notion extends to any finite-dimensional vector space, and allows us to treat any n-dimensional vector space as being $\mathbb{F}^n$. This allows us to use *Mathematica* vectors to perform calculations with any finite-dimensional vector space, just as we have previously been doing with polynomial spaces. This result is nontrivial, and should be proved in any good linear algebra text. We do not prove it here.

The standard bases are not the only basis for any given space; in fact there are many. In fact, if V is an n-dimensional vector space, then any linearly independent subset of V with exactly n elements will be a basis for V. We explore this a little now.

We choose $(1, 2)$ and $(-1, 3)$ as a linearly independent set of vectors in $\mathbb{R}^2$. That these vectors are linearly independent should be clear, but we shall verify this with *Mathematica* regardless.

```
In[154]:= v = {{1, 2}, {-1, 3}}
          NullSpace[v // Transpose]
```

Out[154]= $\{\{1, 2\}, \{-1, 3\}\}$

Out[155]= $\{\}$

Our claim is that any vector (a, b) in $\mathbb{R}^2$ may be expressed as a linear combination of our basis vectors. If we ask *Mathematica* for an arbitrary linear combination directly, we do not get an entirely satisfactory answer.

```
In[156]:= c1 * v[[1]] + c2 * v[[2]]
```

Out[156]= $\{a - b, 2a + 3b\}$

However, we are asking a question about when (or, in this case, if) a vector is a linear combination of other vectors. We know that this problem can be solved with a linear system. Attacking the problem in this way gives a better answer.

```
In[157]:= LinearSolve[v // Transpose, {a, b}]
```

Out[157]= $\left\{\dfrac{3a}{5} + \dfrac{b}{5}, \dfrac{1}{5}(-2a + b)\right\}$

```
In[158]:= %[[1]] * v[[1]] + %[[2]] * v[[2]]
```

Out[158]= $\left\{\dfrac{3a}{5} + \dfrac{1}{5}(2a - b) + \dfrac{b}{5}, 2\left(\dfrac{3a}{5} + \dfrac{b}{5}\right) + \dfrac{3}{5}(-2a + b)\right\}$

```
In[159]:= Simplify[%]
```

Out[159]= $\{a, b\}$

In fact, we may use this technique to verify the claim that any n linearly independent vectors of an n-dimensional vector space will form a basis for the space. We do this by treating the problem as the equivalent question of whether an arbitrary vector may

be constructed as a linear combination of the basis vectors, which in turn may be thought of as solving the vector equation $Ax = b$. In this case, A is the square matrix constructed with the columns being the column vectors of the (alleged) basis vectors. If the basis vectors are linearly independent, then the matrix is invertible, and so the solution $Ax = b$ has a unique solution for every b.

Let's look at this with some polynomials now.

```
In[160]:= p = {x^3, x^3 + 1, x^3 + x + 1, x^3 + x + x^2 + 1}
```

Out[160]= $\left\{x^3,\ 1 + x^3,\ 1 + x + x^3,\ 1 + x + x^2 + x^3\right\}$

We have defined 4 linearly independent degree-3 polynomials. We convert these into equivalent 4-tuples, and verify the linear independence.

```
In[161]:= f[p_ /; PolynomialQ[p, x] && Exponent[p, x] <= 3] :=
            Table[Coefficient[p, x, n], {n, 0, 3}]
          f[{p0_, p1_, p2_, p3_}] :=
            p0 + p1 * x + p2 * x^2 + p3 * x^3
```

```
In[162]:= M := (f /@ p) // Transpose
          M // MatrixForm
          NullSpace[M]
```

Out[162]= $\begin{pmatrix} 0\ 1\ 1\ 1 \\ 0\ 0\ 1\ 1 \\ 0\ 0\ 0\ 1 \\ 1\ 1\ 1\ 1 \end{pmatrix}$

Out[163]= $\{\}$

Now all that remains to be seen is whether a general polynomial $dx^3 + cx^2 + bx + a$ can be constructed as a linear combination of the polynomials in our list `p`. We answer this question in $\mathbb{F}^4$ using our equivalencies, and verify the answer in $P_3(\mathbb{F})$.

```
In[164]:= LinearSolve[M, {a, b, c, d}]
```

Out[164]= $\{-a + d,\ a - b,\ b - c,\ c\}$

```
In[165]:= Sum[%[[i]] * p[[i]], {i, 1, Length[p]}]
```

Out[165]= $(-a + d)x^3 + (a - b)\left(1 + x^3\right) + (b - c)\left(1 + x + x^3\right) + c\left(1 + x + x^2 + x^3\right)$

```
In[166]:= Expand[%]
```

Out[166]= $a + bx + cx^2 + dx^3$

And so, not only can we see that the polynomials form a basis for $P_3(\mathbb{F})$, but we even have a formula for the coefficients the basis vectors will have for any given polynomial.

As always, we clean up our global definitions.

```
In[167]:= ClearAll["Global`*"]
```

3.3 Linear Transformations

3.3.1 Introduction to Linear Transformations

We now look at functions between vector spaces. Such functions are sometimes called *transformations*. We say a function (or transformation) is *linear* if it preserves linear

combinations. That is if we have vector spaces U, V, and a function $f : V \to V$, we say that f is linear if

$$f(c_1 v_1 + c_1 v_1) = c_1 f(v_1) + c_2 f(v_2) \text{ for each } c_1, c_2 \in \mathbb{F} \text{ and } v_1, v_2 \in V$$

or, in other words if we put a linear combination of vectors into the function, what we get out is the same linear combination, but of the images of our original vectors.

Let's look at two simple examples of functions from $\mathbb{R}^2 \to \mathbb{R}^2$.

$$f : \mathbb{R}^2 \to \mathbb{R}^2 \qquad\qquad g : \mathbb{R}^2 \to \mathbb{R}^2$$

$$\begin{bmatrix} a \\ b \end{bmatrix} \mapsto \begin{bmatrix} 2a + b \\ b - a \end{bmatrix} \qquad\qquad \begin{bmatrix} a \\ b \end{bmatrix} \mapsto \begin{bmatrix} a^2 \\ b \end{bmatrix}$$

We'll begin with f.

```
In[168]:= Block[{f, u = {u1, u2}, v = {v1, v2}, c1, c2},
          f[{a_, b_}] := {2 a + b, b - a}
          f[c1 * u + c2 * v] == c1 * f[u] + c2 * f[v]
        ]
```

Out[168]= $\{c1\,u2 + 2(c1\,u1 + c2\,v1) + c2\,v2,\ -c1\,u1 + c1\,u2 - c2\,v1 + c2\,v2\} ==$
$\{c1(2u1 + u2) + c2(2v1 + v2),\ c1(-u1 + u2) + c2(-v1 + v2)\}$

```
In[169]:= Simplify[%]
```

Out[169]= True

We see that f is linear. Now for g.

```
In[170]:= Block[{g, u = {u1, u2}, v = {v1, v2}, c1, c2},
          g[{a_, b_}] := {a^2, b}
          g[c1 * u + c2 * v] == c1 * g[u] + c2 * g[v]
        ]
```

Out[170]= $\left\{(c1\,u1 + c2\,v1)^2,\ c1\,u2 + c2\,v2\right\} == \left\{c1\,u1^2 + c2\,v1^2,\ c1\,u2 + c2\,v2\right\}$

```
In[171]:= Simplify[%]
```

Out[171]= $\left\{-c1\,u1^2 - c2\,v1^2 + (c1\,u1 + c2\,v1)^2,\ 0\right\} == \{0,\ 0\}$

We see that because, in general, $-c1\,u1^2 - c2\,v1^2 + (c1\,u1 + c2\,v1)^2 \neq 0$ it is the case that g is not linear.

3.3.2 Linear Transformations as Matrices

We know that every vector in a given vector space may be written as a linear combination of basis vectors. We also know that a linear transformation preserves linear combination. It follows then that once we know how a linear transformation modifies the basis vectors of a vector space, we know how it will modify any vector in the space.

We now extend our equivalence between vector spaces and n-tuples a little. If $v_1, \ldots, v_n$ is a basis for an n-dimensional vector space, then we may write an arbitrary vector $u = c_1 v_1 + \cdots + c_n v_n$ simply as the n-tuple $(c_1, \ldots, c_n)$. This is precisely what we have been doing previously with polynomials and matrices. However, we may use this idea to write the same vector in $\mathbb{F}^n$ in many different ways.

Let us recall an example from Section 3.2.4, where we showed that the vectors $(1, 2)$ and $(-1, 3)$ formed a basis for $\mathbb{R}^2$. Of course, these vectors are already written in terms of the standard basis $(1, 0)$ and $(0, 1)$ in that $(1, 2) = 1 \cdot (1, 0) + 2 \cdot (0, 1)$ and $(-1, 3) = -1 \cdot (1, 0) + 3 \cdot (0, 1)$. We already see our principle in action. However, things can become confusing, as we are not altogether accustomed to thinking of (a, b) as a shorthand notation for these linear combinations, and worse still we are about to be comparing different bases. So we now give these bases names. Let S (for "standard") be the basis $\{(1, 0), (0, 1)\}$ and let B (for, simply, "basis") be our alternate basis $\{(1, 2), (-1, 3)\}$.

At this stage, it is quite important that we be aware of just which basis we are referring to at any one time. We do this by denoting what basis a vector is in respect to by using a subscript. That is, for some basis $A = \{a_1, \ldots, a_n\}$, the vector $(u_1, \ldots, u_n)_A$ denotes the vector that is equal to $u_1 a_1 + \cdots + u_n a_m$. If we do not specify a basis in this way, and it is not abundantly clear from the context which we mean, then we mean the standard basis.

Returning to our example, let us take the vector $v = (5, 15)$. To be more clear, we mean $(5, 15)_S$. We know from our example in Section 3.2.4 that we can write this vector as a linear combination of our basis B with the coefficients $c_1 = \frac{3}{5} \cdot 5 + \frac{1}{5} \cdot 15 = 6$ and $c_2 = -\frac{2}{5} \cdot 5 + \frac{1}{5} \cdot 15 = 1$ so that $(5, 15) = 6 \cdot (1, 2) + 1 \cdot (-1, 3)$. All of this is with respect to the usual basis, remember, and so should feel familiar. However, because of this, we could equally well write the vector with respect to the alternate basis B and say that $v = (6, 1)_B$. There is no question that we refer to the same actual vector within our vector space. The reader who is familiar with representation of natural numbers in different bases should see a similarity with this idea.

So now, when we see a vector written as an n-tuple we should think of this as the list of coefficients of some basis, even if that basis is simply the standard basis. In particular the vector $(c, 0, \ldots, 0)$ represents the vector which is the first basis vector scaled by a constant c, $(0, c, 0, \ldots, 0)$ as the second basis vector scaled by a constant c, and so on.

Observe, now, the effect matrix multiplication has on these vectors. We think of a matrix as being a collection of column vectors.

```
In[172]:= With[{M = {{u1, u2}, {v1, v2}} // Transpose},
            M // MatrixForm // Print;
            M . {c, 0} // Print;
            M . {0, d} // Print;
          ]
```

$$\text{Out[172]}= \begin{pmatrix} u1 & v1 \\ u2 & v2 \end{pmatrix}$$

$$\text{Out[173]}= \{c\,u1,\ c\,u2\}$$

$$\text{Out[174]}= \{d\,v1,\ d\,v2\}$$

Observe that due to the nature of matrix multiplication, multiplying M on the right by the vector $(c, 0)$ will yield the column vector (cu_1, cu_2). Similarly multiplying M on the right by the vector $(0, d)$ will yield the column vector (dv_1, dv_2). It should be clear that this will hold for any $n \times m$ matrix and n-tuple. Furthermore multiplying M on the right by the vector (c, d) will yield the vector $(cu_1 + dv_1, cu_2 + dv_2) = c(u_1, u_2) + d(v_1, v_2)$.

```
In[175]:= With[{M = {{u1, u2}, {v1, v2}} // Transpose},
            M.{c, d} // Print;
            c * {u1, u2} + d * {v1, v2} // Print
          ]
```

$$\text{Out[175]}= \{c\,u1 + d\,v1,\ c\,u2 + d\,v2\}$$

Out[176]= $\{c\,u1 + d\,v1,\ c\,u2 + d\,v2\}$

Again, because of the way matrix multiplication is performed, this idea extends to any $n \times m$ matrix and n-tuple.

The point to all this is to demonstrate that matrix multiplication coincides with a linear combination of the column vectors of the matrix. It can be fairly easily checked that matrix multiplication itself is a linear transformation. What is less immediately obvious (but is strongly suggested by the above observations) is that every linear transformation between finite-dimensional vector spaces can be represented by matrix multiplication, for a suitable matrix.

We put this information together now. We know that a vector is just a shorthand way of writing a linear combination of basis vectors. We also know that linear transformations preserve linear combinations. Finally we know multiplying a matrix on the left by a vector creates a linear combination of the column vectors of that matrix. So, if we calculate the image of the basis vectors under the transformation, and then construct the matrix whose column vectors are those image vectors (in the correct order), then multiplying that matrix on the right by any vector (written with respect to the basis) will be exactly the same as applying the linear transformation to the vector directly.

For example, using our linear function f from Section 3.3.1,

```
In[177]:= Block[{f, M},
            f[{a_, b_}] := {2 a + b, b - a};
            M = {f[{1, 0}], f[{0, 1}]} // Transpose
            M // MatrixForm // Print;
            f[{a, b}] // Print
            M . {a, b} // Print
          ]
```

Out[177]= $\begin{pmatrix} 2 & 1 \\ -1 & 1 \end{pmatrix}$

Out[178]= $\{2a + b,\ -a + b\}$

Out[179]= $\{2a + b,\ -a + b\}$

We use this concept now to rotate some plots in $\mathbb{R}^2$. Specifically, to rotate them around the origin (as rotation around an arbitrary point is not, in general, a linear transformation). We need only know what happens to the standard basis vectors $(1, 0), (0, 1)$ under the rotation. It is simpler to use polar co-ordinates, however, be warned that polar co-ordinates (r, θ) most emphatically are not a linear combination of basis vectors. They refer to a distance from the origin and an angle from the x-axis. To avoid confusion, we use the polar form of complex numbers $e^{i\theta}$ when referring to points in polar form.

As complex numbers, our basis vectors are simply $1 = e^{i0}$ and $e^{i\frac{\pi}{2}}$. Rotating anti-clockwise by an angle of θ we have that

$$\begin{array}{rcccl} (1, 0) = & 1 & \mapsto & e^{i\theta} & = (\cos(\theta), \sin(\theta)) \\ (0, 1) = & e^{i\frac{\pi}{2}} & \mapsto & e^{i(\frac{\pi}{2} + \theta)} & = \left(\cos\left(\frac{\pi}{2} + \theta\right), \sin\left(\frac{\pi}{2} + \theta\right)\right) \end{array}$$

and we can now construct our matrix, which we construct as a function of θ so that we may reuse it for different angles.

```
In[180]:= R[th_] := Transpose[
            {{Cos[th], Sin[th]}, {Cos[Pi / 2 + th], Sin[Pi / 2 + th]}}
          ]
```

In[181]:= **R[t] // MatrixForm**

Out[181]= $\begin{pmatrix} \mathrm{Cos}[t] & -\mathrm{Sin}[t] \\ \mathrm{Sin}[t] & \mathrm{Cos}[t] \end{pmatrix}$

The observant reader will notice that *Mathematica* has simplified our second column vector for us. These verifications can easily be verified ($\cos(\pi/2 + \theta) = -\sin(\theta)$ and $\sin(\pi/2 + \theta) = \cos theta$). This is the standard form for a rotation matrix in $\mathbb{R}^2$.

Let us first rotate a parabola through an angle of $\frac{1}{3}\pi$. In order for this linear function to be applied, the matrix must be multiplied with a vector, so we need to use the vector equation (or parameterized equation) for the parabola. That is, $(x, y) = (t, t^2)$.

In[182]:= **R[Pi / 3] . {t, t^2}**
       ParametricPlot[%, {t, -2, 2}]

Out[182]= $\left\{ \dfrac{t}{2} - \dfrac{\sqrt{3}t^2}{2}, \dfrac{\sqrt{3}t}{2} + \dfrac{t^2}{2} \right\}$
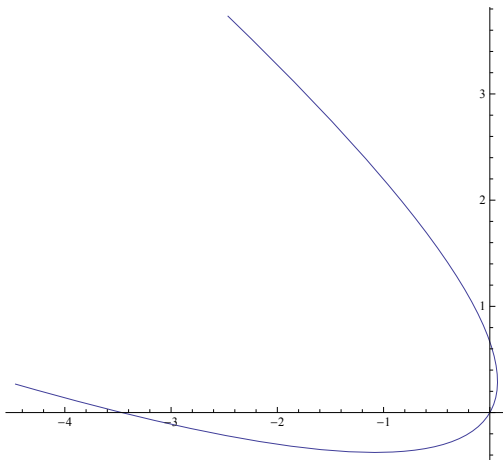
Out[183]=

And now we rotate a sine curve clockwise by $\frac{1}{4}\pi$.

In[184]:= **ParametricPlot[R[-Pi / 4] . {t, Sin[t]}, {t, -2 Pi, 2 Pi}]**
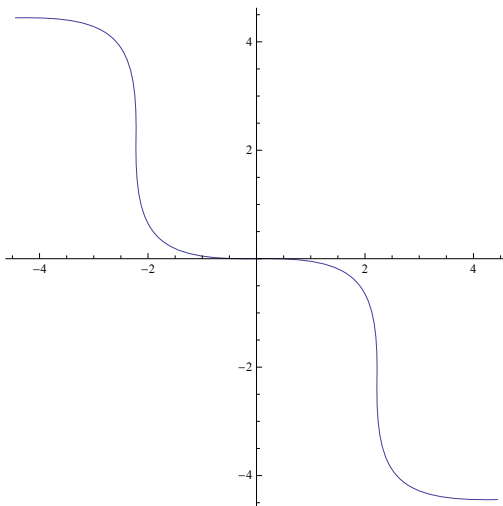
Out[184]=

Note that inasmuch as vectors are implements by lists, the vector form of the parameterization is automatically in a suitable form to use directly in the **ParametricPlot** function. Note, also, that by choosing $-2 \leq t \leq 2$ we have rotated the parabola that would normally lie above the region $-2 \leq x \leq 2$.

Finally we make an observation. Because every linear transformation can be represented as matrix multiplication, it should then be clear that a linear transformation may only produce linear combinations of the components of a vector. In other words, given an arbitrary vector $(v_1, \ldots, v_n)$ we should never see $v_i^k, v_i v_j$ or anything similar as components of the resultant vector. We can only see linear combinations of the v_i, as a result of matrix multiplication. Looking back at our examples from Section 3.3.1 it should be quite clear now that $g : (a, b) \mapsto (a^2, b)$ could not possibly be a linear mapping, whereas $f : (a, b) \mapsto (2a + b, b - a)$ clearly is.

3.3.3 Eigenvectors and Eigenvalues

If we have a linear transformation, $T : V \to V$ say, operating on a vector space V, then it might be the case that there is a vector $v \in V$ that doesn't get moved at all by T, and is only scaled. That is, $T(v) = \lambda v$. We call such a vector an *eigenvector*, and λ the *eigenvalue* corresponding to the eigenvector. Because 0 always remains unchanged by a linear transformation, we do not consider it to be an eigenvector. Note that if $\lambda = 1$ then v is not only an eigenvector, but it is a fixed point of the mapping (as it remains unchanged by the transformation).

The question now is how do we find these eigenvectors? First, instead of thinking of $T(v)$ as the image of v under the transformation T, we consider T to be a matrix (which we can do in light of the previous section). So we are looking for solutions to the vector equation $Tv = \lambda v$, which we may equivalently write $Tv = \lambda Iv$, where I is the identity matrix. Then

$$Tv = \lambda Iv \;\Rightarrow\; Tv - \lambda Iv = 0 \;\Rightarrow\; (T - \lambda I)v = 0$$

and we now have a vector equation corresponding to a linear system of the sort we have dealt with earlier in Sections 3.1.2 and 3.2.

We know that 0 is not an eigenvector, but will be a solution to the linear system, so we need the linear system to have a nontrivial solution. We know from our theorem that this will not happen if the determinant of the matrix is nonzero (i.e., if the matrix is invertible), so we are looking for values of λ where the determinant of $T - \lambda I$ is zero.

Let us have a look at our linear transformation f from Section 3.3.1, remembering that it has the matrix

$$\begin{bmatrix} 2 & 1 \\ -1 & 1 \end{bmatrix}$$

In[185]:= **T = {{-2, 1}, {-1, 1}}**

Out[185]= $\{\{2, 1\}, \{-1, 1\}\}$

In order to get the λ symbol in *Mathematica*, we use the input command **\[Lambda]**.

In[186]:= **T - \[Lambda] * IdentityMatrix[2] // MatrixForm**

Out[186]= $\begin{pmatrix} 2 - \lambda & 1 \\ -1 & 1 - \lambda \end{pmatrix}$

The resultant matrix $T - \lambda I$ is simply the matrix T with λ subtracted from each entry on the diagonal. This always happens no matter the size of the matrix. When we calculate the determinant of this (or any other) matrix, we end up with a polynomial in λ.

In[187]:= `T - \[Lambda] * IdentityMatrix[2] // Det`

Out[187]= $3 - 3\lambda + \lambda^2$

This polynomial is called the *characteristic polynomial* of T. *Mathematica* can calculate this directly using the `CharacteristicPolynomial` function. To use this function we also need to tell it the name we want to use for the variable of the polynomial.

In[188]:= `CharacteristicPolynomial[T, \[Lambda]]`

Out[188]= $3 - 3\lambda + \lambda^2$

We may now easily find solutions for λ. We know that we know the only possible eigenvalues are the values of λ that solve $\chi(\lambda) = 0$ where χ is the characteristic polynomial. Once we find the values for λ (of which there can only be finitely many, for they are the roots of a polynomial) then we may substitute them into our linear system and solve for v in order to find the corresponding eigenvectors.

In[189]:= `Solve[CharacteristicPolynomial[T, \[Lambda]] == 0]`

Out[189]= $\left\{ \left\{ \lambda \to \frac{1}{2} \left(3 - i\sqrt{3} \right) \right\}, \left\{ \lambda \to \frac{1}{2} \left(3 + i\sqrt{3} \right) \right\} \right\}$

We can see straight away that our linear transformation f has no eigenvectors, because the possible eigenvalues are all complex, and there is no possible way our integer-valued matrix multiplied by any real-valued vector could produce a complex scaling of that vector.

Let us look at another example—one that actually has eigenvectors this time—and in three dimensions.

In[190]:= `T = {{-1, 2, 0}, {-6, 6, 0}, {4, -2, 1}}`
           `T // MatrixForm`

Out[190]= $\{\{-1, 2, 0\}, \{-6, 6, 0\}, \{4, -2, 1\}\}$

Out[191]= $\begin{pmatrix} -1 & 2 & 0 \\ -6 & 6 & 0 \\ 4 & -2 & 1 \end{pmatrix}$

In[192]:= `CharacteristicPolynomial[T, \[Lambda]]`
           `Factor[%]`

Out[192]= $6 - 11\lambda + 6\lambda^2 - \lambda^3$

Out[193]= $-(-3 + \lambda)(-2 + \lambda)(-1 + \lambda)$

We can see straight away that $\lambda \in \{1, 2, 3\}$ are solutions to the characteristic polynomial, and so are eigenvalues of the matrix T. This is great, but we still need to know what the corresponding eigenvectors are. All we know at the moment is that, for our linear map T, there is a vector which stays the same (after T is applied), there is another vector which becomes twice as large, and yet another vector which becomes three times as large. But we have no idea which vectors they may be.[3]

---

[3] The astute reader who is familiar with matrix operations should be able to make a good guess, from the matrix of T alone, as to which vector remains unchanged.

In order to find the vectors, we return to our vector equation $(T - \lambda I)v = 0$. However, we now know the only values of λ that could possibly satisfy this equation, and if we substitute these values into the above equation (one by one) we have three separate equations of the form $Mv = 0$ where $M = (T - \lambda I)$. These are, of course, linear systems that we should be quite familiar and adept with by now. So we simply solve these linear systems.

Note that the **Solve** command returns a list of substitution rules. Using a list of rules with the substitution operator will produce a list of expressions; one for each of the rules. We use this here, and call the list of matrices **L**.

```
In[194]:= Solve[CharacteristicPolynomial[T, \[Lambda]] == 0]
          L = T - \[Lambda] * IdentityMatrix[3] /. %
          MatrixForm /@ L
```

Out[194]= $\{\{\lambda \to 1\}, \{\lambda \to 2\}, \{\lambda \to 3\}\}$

Out[195]= $\{\{\{-2, 2, 0\}, \{-6, 5, 0\}, \{4, -2, 0\}\}, \{\{-3, 2, 0\}, \{-6, 4, 0\}, \{4, -2, -1\}\},$
$\{\{-4, 2, 0\}, \{-6, 3, 0\}, \{4, -2, -2\}\}\}$

Out[196]= $\left\{ \begin{pmatrix} -2 & 2 & 0 \\ -6 & 5 & 0 \\ 4 & -2 & 0 \end{pmatrix}, \begin{pmatrix} -3 & 2 & 0 \\ -6 & 4 & 0 \\ 4 & -2 & -1 \end{pmatrix}, \begin{pmatrix} -4 & 2 & 0 \\ -6 & 3 & 0 \\ 4 & -2 & -2 \end{pmatrix} \right\}$

The list of eigenvalues was in the order of 1, 2, then 3, and so the vectors correspond to the eigenvalues in that same order. We may quickly verify this if we are unsure. In order to find the eigenvectors corresponding to our eigenvalues, we need to know for which vectors, v say, that $Mv = 0$ for each matrix M in our list, **L**. To do this we need to apply the **NullSpace** function to each of the matrices.

```
In[197]:= NullSpace /@ L
```

Out[197]= $\{\{\{0, 0, 1\}\}, \{\{2, 3, 2\}\}, \{\{1, 2, 0\}\}\}$

And we have our answer, although we should think about it for a second. If we have an eigenvector v, then $Tv = \lambda v$ for some λ. But because T is linear, then $T(kv) = kTv = k\lambda v$ for any $k \in \mathbb{F}$ in our field of scalars. So any scale of v is also an eigenvector corresponding to the eigenvector λ. What we usually do here is pick a vector to represent all of the possible vectors. Inasmuch as the **NullSpace** function provides a basis for the null space of the matrix, those basis vectors would seem to be the obvious choices. As such, we declare that $(0, 0, 1), (2, 3, 2)$ and $(1, 2, 0)$ are the eigenvectors corresponding to the eigenvectors $1, 2,$ and 3, respectively. To demonstrate this, we check this answer in *Mathematica*.

```
In[198]:= T . (t * 0, 0, 1)
          T . (t * 2, 3, 2)
          T . (t * 1, 2, 0)
```

Out[198]= $\{0, 0, t\}$

Out[199]= $\{4t, 6t, 4t\}$

Out[200]= $\{3t, 6t, 0\}$

We explore one more example. In the previous example we had a 3×3 matrix, and we ended up with three distinct eigenvalues, and three distinct single-dimensional families of eigenvectors, each one corresponding to a particular eigenvalue. This does not always

happen in general.[4] Furthermore, there is no requirement that eigenvalues be nonzero (unlike eigen*vectors*). The example we now look at demonstrates both of these points.

In[201]:= `T = {{2, -5, 6, 0}, {4, -19, 24, 0}, {3, -15, 19, 0}, {1, -29, 38, 2}}`
         `T // MatrixForm`

Out[201]= $\{\{2, -5, 6, 0\}, \{4, -19, 24, 0\}, \{3, -15, 19, 0\}, \{1, -29, 38, 2\}\}$

Out[202]= $\begin{pmatrix} 2 & -5 & 6 & 0 \\ 4 & -19 & 24 & 0 \\ 3 & -15 & 19 & 0 \\ 1 & -29 & 38 & 2 \end{pmatrix}$

In[203]:= `CharacteristicPolynomial[T, \[Lambda]]`
         `Factor[%]`

Out[203]= $-2\lambda + 5\lambda^2 - 4\lambda^3 + \lambda^4$

Out[204]= $(-2 + \lambda)(-1 + \lambda)^2\lambda$

This time we have $0, 1$, and 2 as the only eigenvalues, however, notice that in this case 1 is a repeated root of the characteristic polynomial, and so we also consider it to be a repeated eigenvalue of the linear operator (or, equivalently, matrix) T. We now find the eigenvectors.

In[205]:= `Solve[CharacteristicPolynomial[T, \[Lambda]] == 0]`
         `L = T - \[Lambda] * IdentityMatrix[4] /. %`
         `MatrixForm /@ L`

Out[205]= $\{\{\lambda \to 0\}, \{\lambda \to 1\}, \{\lambda \to 1\}, \{\lambda \to 2\}\}$

Out[206]= $\{\{\{2, -5, 6, 0\}, \{4, -19, 24, 0\}, \{3, -15, 19, 0\}, \{1, -29, 38, 2\}\},$
         $\{\{1, -5, 6, 0\}, \{4, -20, 24, 0\}, \{3, -15, 18, 0\}, \{1, -29, 38, 1\}\},$
         $\{\{1, -5, 6, 0\}, \{4, -20, 24, 0\}, \{3, -15, 18, 0\}, \{1, -29, 38, 1\}\},$
         $\{\{0, -5, 6, 0\}, \{4, -21, 24, 0\}, \{3, -15, 17, 0\}, \{1, -29, 38, 0\}\}\}$

Out[207]= $\left\{ \begin{pmatrix} 2 & -5 & 6 & 0 \\ 4 & -19 & 24 & 0 \\ 3 & -15 & 19 & 0 \\ 1 & -29 & 38 & 2 \end{pmatrix}, \begin{pmatrix} 1 & -5 & 6 & 0 \\ 4 & -20 & 24 & 0 \\ 3 & -15 & 18 & 0 \\ 1 & -29 & 38 & 1 \end{pmatrix}, \begin{pmatrix} 1 & -5 & 6 & 0 \\ 4 & -20 & 24 & 0 \\ 3 & -15 & 18 & 0 \\ 1 & -29 & 38 & 1 \end{pmatrix}, \begin{pmatrix} 0 & -5 & 6 & 0 \\ 4 & -21 & 24 & 0 \\ 3 & -15 & 17 & 0 \\ 1 & -29 & 38 & 0 \end{pmatrix} \right\}$

Note that the `Solve` function produced a duplicate rule for the duplicate solution of $\lambda = 1$, and so we also have a duplicate matrix in our list. We quickly fix this with the `DeleteDuplicates` function before proceeding.

In[208]:= `L = DeleteDuplicates[L]`
         `MatrixForm /@ L`

Out[208]= $\{\{\{2, -5, 6, 0\}, \{4, -19, 24, 0\}, \{3, -15, 19, 0\}, \{1, -29, 38, 2\}\},$
         $\{\{1, -5, 6, 0\}, \{4, -20, 24, 0\}, \{3, -15, 18, 0\}, \{1, -29, 38, 1\}\},$
         $\{\{0, -5, 6, 0\}, \{4, -21, 24, 0\}, \{3, -15, 17, 0\}, \{1, -29, 38, 0\}\}\}$

Out[209]= $\left\{ \begin{pmatrix} 2 & -5 & 6 & 0 \\ 4 & -19 & 24 & 0 \\ 3 & -15 & 19 & 0 \\ 1 & -29 & 38 & 2 \end{pmatrix}, \begin{pmatrix} 1 & -5 & 6 & 0 \\ 4 & -20 & 24 & 0 \\ 3 & -15 & 18 & 0 \\ 1 & -29 & 38 & 1 \end{pmatrix}, \begin{pmatrix} 0 & -5 & 6 & 0 \\ 4 & -21 & 24 & 0 \\ 3 & -15 & 17 & 0 \\ 1 & -29 & 38 & 0 \end{pmatrix} \right\}$

---

[4] Indeed, we have already seen an example with no eigenvalues or eigenvectors.

In[210]:= **NullSpace /@ L**

Out[210]= $\{\{\{2, 8, 6, 1\}\}, \{\{5, 1, 0, 24\}, \{2, 4, 3, 0\}\}, \{\{0, 0, 0, 1\}\}\}$

Now, we have $(2, 8, 6, 1)$ as "the" eigenvector corresponding to eigenvalue 0 as well as $(0, 0, 0, 1)$ as "the" eigenvector corresponding to eigenvalue 2. Of course, any scale multiple of these vectors is also a corresponding eigenvector. In particular, this means that any scale of the vector $(2, 8, 6, 1)$ will be turned into the zero vector by our transformation T. Eigenvalue 1 is more interesting, but we verify the easy ones first.

In[211]:= **T . (t.{2, 8, 6, 1})**
         T . (t.{0, 0, 0, 1})

Out[211]= $\{0, 0, 0, 0\}$

Out[212]= $\{0, 0, 0, 2t\}$

Now, to our repeated eigenvalue, 1: note that *Mathematica* has given us a list of two vectors, and so the null space of the matrix $T - 2I$ is spanned by two vectors, and so is a two-dimensional space. We could think of the two eigenvectors, $(5, 1, 0, 24)$ and $(2, 4, 3, 0)$ as being two separate eigenvectors of T, both corresponding to the eigenvalue 1 (and, of course, any scale of each vector) as follows.

In[213]:= **T . (t.{5, 1, 0, 24})**
         T . (t.{2, 4, 3, 0})

Out[213]= $\{5t, t, 0, 24t\}$

Out[214]= $\{2t, 4t, 3t, 0\}$

However, this is really only half the story. The two vectors together span the null space of $T - 2I$, and so any linear combination of these two vectors ought to be left unchanged by our transformation T.

In[215]:= **t . {5, 1, 0, 24} + s . {2, 4, 3, 0}**
         T . %

Out[215]= $\{2s + 5t, 4s + t, 3s, 24t\}$

Out[216]= $\{18s - 5(4s + t) + 2(2s + 5t), 72s - 19(4s + t) + 4(2s + 5t),$
         $57s - 15(4s + t) + 3(2s + 5t), 116s + 53t - 29(4s + t)\}$

In[217]:= **Simplify[%]**

Out[217]= $\{2s + 5t, 4s + t, 3s, 24t\}$

Observe, firstly, that each family of eigenvectors is, in fact, a vector subspace of the space which the matrix acts on; verification is left as an exercise for the reader. These spaces are called *eigenspaces*.

Observe, secondly, that in this, and the previous, example the dimensions of the eigenspaces all add up to the dimension of the whole space. In the most recent case we had eigenspaces of degree 2, 1, and 1 which add together to give us 4 which was the dimension of the vector space upon which the matrix acted. This is not always the case, but we explore the cases in which it happens in the next section.

3.3.4 Diagonalization

We motivate this section with an observation regarding a previous example. Let us look at our 3×3 matrix from the previous section,

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

and its eigenvectors, $(0, 0, 1), (1, 3/2, 1)$ and $(1, 2, 0)$ which correspond to eigenvalues $1, 2$ and 3 respectively. In order to avoid messy fractions, we use the vector $(2, 3, 2)$ instead of $(1, 3/2, 1)$ as they are from the same eigenspace.

In[218]:= **T = {{-1, 2, 0}, {-6, 6, 0}, {4, -2, 1}}**
        e = {{0, 0, 1}, {2, 3, 2}, {1, 2, 0}}

Out[218]= $\{\{-1, 2, 0\}, \{-6, 6, 0\}, \{4, -2, 1\}\}$

Out[219]= $\{\{0, 0, 1\}, \{2, 3, 2\}, \{1, 2, 0\}\}$

The observation we make is simple. These eigenvectors are linearly independent, and because there are three of them, they form an alternate basis for $\mathbb{R}^3$. This is an observation that we can easily check in *Mathematica* thanks to our work back in Section 3.2.

In[220]:= **NullSpace[e // Transpose]**

Out[220]= $\{\}$

Now, inasmuch as our vectors e_1, e_2, and e_3 form a basis (which we call B), then we can write any vector $v \in \mathbb{R}^3$ as a linear combination of these new basis vectors. We use *Mathematica* to find an explicit formula for this new combination, by solving the appropriate linear system (just as we did in Section 3.2.4).

In[221]:= **LinearSolve[e // Transpose, {x, y, x}]**

Out[221]= $\{-4x + 2y + z, 2x - y, -3x + 2y\}$

It is left as an exercise to the reader to check this. We now set up a function, which we name **ChgBasis**, to perform this basis conversion for us.

In[222]:= **ChgBasis[{x_, y_, z_}] := LinearSolve[e // Transpose, {x, y, x}]**

Let's see what the vector $(1, 1, 1)$ becomes in terms of the basis B.

In[222]:= **ChgBasis[{1, 1, 1}]**

Out[222]= $\{-1, 1, -1\}$

It is quite elementary to verify that $(-1, 1, -1)_B = -1 \cdot (0, 0, 1) + 1 \cdot (2, 3, 2) - 1 \cdot (1, 2, 0) = (1, 1, 1)$.

The question that may occur now, is what will our linear operator T do to vectors written in terms of our new basis? Clearly T has not changed, and neither have the vectors themselves, just how we write them. However, because we are writing the vectors slightly differently now, our old matrix for T will probably not be appropriate anymore, as it was expecting to be multiplied by a vector that contained the coefficients of the standard basis, and not our new basis.

If we think about this for a bit, we can apply what we already know about linear transformations to obtain an answer. We know that, because T is linear, then $T(\alpha_1 v_1 + \alpha_2 v_2) = \alpha_1 T(v_1) + \alpha_2 T(v_2)$. We apply this to an arbitrary vector v written in terms of our basis B; that is, $v = \alpha_1 e_1 + \alpha_2 e_2 + \alpha_3 e_3$. When we do this we get that

$$
\begin{aligned}
T(v) &= T(\alpha_1 e_1 + \alpha_2 e_2 + \alpha_3 e_3) \\
&= \alpha_1 T(e_1) + \alpha_2 T(e_2) + \alpha_3 T(e_3) \\
&= \alpha_1 \cdot 1 \cdot e_1 + \alpha_2 \cdot 2 \cdot e_2 + \alpha_3 \cdot 3 \cdot e_3
\end{aligned}
$$

because, in this case e_1, e_2, and e_3 are eigenvectors corresponding to eigenvalues 1, 2, and 3, respectively. To put this more succinctly

$$T\left((\alpha_1, \alpha_2, \alpha_3)_B\right) = (\alpha_1, 2\alpha_2, 3\alpha_3)_B \tag{3.1}$$

which can be achieved by multiplication (on the left) by the matrix

$$
\begin{bmatrix}
1 & 0 & 0 \\
0 & 2 & 0 \\
0 & 0 & 3
\end{bmatrix}
$$

We demonstrate this for an arbitrary three-dimensional vector, $v = (x, y, z)$. First, we show what the transformation does to the vector directly. We call this new vector w.

```
In[223]:= v = {x, y, z}
          w = T . v
```

Out[223]= $\{x, y, z\}$

Out[224]= $\{-x + 2y, -6x + 6y, 4x - 2y + z\}$

Recall from earlier that v when written as a vector with respect to the basis B is

$$v = (-4x + 2y + z, 2x - y, -3x + 2y)_B \tag{3.2}$$

Now, from Equation (3.1), we should expect $T(v)$ to be equal to

$$T(v) = (1 \cdot (-4x + 2y + z), 2 \cdot (2x - y), 3 \cdot (-3x + 2y))_B = (-4x + 2y + z, 4x - 2y, -9x + 6y)_B$$

which is, as it happens, exactly what happens.

```
In[225]:= ChgBasis[w]
```

Out[225]= $\{-4x + 2y + z, 4x - 2y, -3(3x - 2y)\}$

Let us look a little more closely at the change of basis. One should notice that it is a linear transformation, as it is the solution of a linear system of equations. As such we should be able to represent it as matrix multiplication. We construct the matrix as we did in Section 3.3.2, by seeing what the change of basis does to the standard basis vectors. We call the matrix P.

```
In[226]:= P = (ChgBasis /@ {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}}) // Transpose
          P // MatrixForm
```

Out[226]= $\{\{-4, 2, 1\}, \{2, -1, 0\}, \{-3, 2, 0\}\}$

Out[227]= $\begin{pmatrix} -4 & 2 & 1 \\ 2 & -1 & 0 \\ -3 & 2 & 0 \end{pmatrix}$

The form of this matrix should not be surprising when we compare it to the elements of the of an arbitrary vector written with regards to the basis B, as we did in Equation (3.2). We see, in our matrix, a column of 4, 2, and -3 which are precisely the coefficients of x, another column of 2, -1, and 2 which are the coefficients of y, and 1, 0, and 0 which are the coefficients of z. In other words, we see the matrix which, when multiplied on the left against (x, y, z) will produce exactly the vector in Equation (3.2).

With this information we may now calculate Tv, written with respect to the basis B, with only simple matrix multiplication. We do this using the matrix P, and the diagonal matrix of eigenvalues, above, which we name Di. We would prefer to call this diagonal matrix simply D, but recall that *Mathematica* uses `D` for differentiation, and it is a reserved name. We use *Mathematica*'s `DiagonalMatrix` function to more easily create the diagonal matrix. We see that we compute precisely the same vector we computed with `ChgBasis[w]`.

```
In[228]:= Di = DiagonalMatrix[{1, 2, 3}]},
          Di // MatrixForm
          Di . P . v
```

Out[228]= $\{\{1, 0, 0\}, \{0, 2, 0\}, \{0, 0, 3\}\}$

Out[229]= $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix}$

Out[230]= $\{-4x + 2y + z, 4x - 2y, -9x + 6y\}$

It should follow that if we can change our basis from the standard basis to the basis B, then we should be able to freely change back. That is, the change of basis should be an invertible function. This is reinforced when we remember that the coefficients of the standard basis are unique to each unique vector, as are the coefficients of the vectors in our basis B. What we have here is an isomorphism, which really ought to be invertible. We should expect, then, that the matrix P has an inverse, and that multiplying by this inverse should undo the basis change, as clearly $P^{-1}P = I$.

```
In[231]:= (P // Inverse).%
```

Out[231]= $\{-9x + 2(4x - 2y) + 6y, 3(4x - 2y) + 2(-9x + 6y), -4x + 2(4x - 2y) + 2y + z\}$

```
In[232]:= Simplify[%]
```

Out[232]= $\{-x + 2y, -6x + 6y, 4x - 2y + z\}$

This is the same vector we computed as Tv, above, and named `w`.

```
In[233]:= % == w
```

Out[233]= True

So now we have two matrix representations for our linear operator T. One is just the matrix we started with, and the other is the product of three matrices, $P^{-1} Di P$, one of which is diagonal. In fact, we should expect that $T = P^{-1} Di P$.

```
In[234]:= {(T // MatrixForm), (P // Inverse) . Di . P // MatrixForm}
          %[[1]] == %[[2]]
```

Out[234]= $\left\{ \begin{pmatrix} -1 & 2 & 0 \\ -6 & 6 & 0 \\ 4 & -2 & 1 \end{pmatrix}, \begin{pmatrix} -1 & 2 & 0 \\ -6 & 6 & 0 \\ 4 & -2 & 1 \end{pmatrix} \right\}$

Out[235]= True

So now, one might ask, just what is P^{-1}?

In[236]:= **P // Inverse // MatrixForm**

Out[236]= $\begin{pmatrix} 0 & 2 & 1 \\ 0 & 3 & 2 \\ 1 & 2 & 0 \end{pmatrix}$

It is precisely the matrix consisting of our eigenvectors, in the exact order we used them back when we solved the linear system for the **ChgBasis** function we wrote.

In[237]:= **e // Transpose // MatrixForm**

Out[237]= $\begin{pmatrix} 0 & 2 & 1 \\ 0 & 3 & 2 \\ 1 & 2 & 0 \end{pmatrix}$

This is an example of a *diagonalizable* matrix. That is, an $n \times n$ matrix, M say, which may be written as a product of matrices $P\,D\,P^{-1}$ where D is a diagonal matrix. Alternately, we might say that $P^{-1}\,M\,P$ is a diagonal matrix, because

$$M = P\,D\,P^{-1} \implies P^{-1}\,M = D\,P^{-1} \implies P^{-1}\,M\,P = D$$

If we rename P^{-1} to be P in the previous example, then we see that this is satisfied.

In[238]:= **P = P // Inverse**
        P . Di . (P // Inverse) // MatrixForm
        (P // Inverse) . T . P // MatrixForm

Out[238]= $\{\{0, 2, 1\}, \{0, 3, 2\}, \{1, 2, 0\}\}$

Out[239]= $\begin{pmatrix} -1 & 2 & 0 \\ -6 & 6 & 0 \\ 4 & -2 & 1 \end{pmatrix}$

Out[240]= $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{pmatrix}$

Let us look now at the next example from the previous section (Section 3.3.3).

In[241]:= **T = {{2, -5, 6, 0}, {4, -19, 24, 0}, {3, -15, 19, 0}, {1, -29, 38, 2}}**
        T // MatrixForm

Out[241]= $\{\{2, -5, 6, 0\}, \{4, -19, 24, 0\}, \{3, -15, 19, 0\}, \{1, -29, 38, 2\}\}$

Out[242]= $\begin{pmatrix} 2 & -5 & 6 & 0 \\ 4 & -19 & 24 & 0 \\ 3 & -15 & 19 & 0 \\ 1 & -29 & 38 & 2 \end{pmatrix}$

We should remember that this example had eigenvalues $0, 1, 2$ and that the eigenspace associated with eigenvalue 1 was two-dimensional. Specifically, we had eigenvector $(2, 8, 6, 1)$ corresponding to eigenvalue 0, eigenvectors $(5, 1, 0, 24)$ and $(2, 4, 3, 0)$ formed

a basis for the eigenspace corresponding to eigenvalue 1, and eigenvector $(0, 0, 0, 1)$ corresponded to eigenvalue 2.

It is prudent to talk about the multiplicity of an eigenvalue as well as the dimension of the eigenspace. The multiplicity of the eigenvalue, is simply its multiplicity in as a root of the characteristic polynomial. In this example the characteristic polynomial is $x(x - 2)(x - 1)^2$, and so $x = 0$ and $x = 2$ are single roots, whereas $x = 1$ is a double root. Hence the eigenvalues 0 and 2 had multiplicity 1, and eigenvalue 1 had multiplicity 2. Similarly the dimensions of the eigenspaces are the same as the multiplicity of the corresponding eigenvalue.

We ask *Mathematica* to calculate the eigenvectors and eigenvalues directly using the **Eigenvectors** and **Eigenvalues** functions.

```
In[243]:= vec = Eigenvectors[T]
          val = Eigenvalues[T]
```

Out[243]= $\{\{0, 0, 0, 1\}, \{5, 1, 0, 24\}, \{2, 4, 3, 0\}, \{2, 8, 6, 1\}\}$

Out[244]= $\{2, 1, 1, 0\}$

We can now produce our diagonal matrix, and see if we can diagonalize the original matrix.

```
In[245]:= Di = DiagonalMatrix[val]
          P = vec // Transpose
          {P // MatrixForm, Di // MatrixForm}
```

Out[245]= $\{\{2, 0, 0, 0\}, \{0, 1, 0, 0\}, \{0, 0, 1, 0\}, \{0, 0, 0, 0\}\}$

Out[246]= $\{\{0, 5, 2, 2\}, \{0, 1, 4, 8\}, \{0, 0, 3, 6\}, \{1, 24, 0, 1\}\}$

Out[247]= $\left\{ \begin{pmatrix} 0 & 5 & 2 & 2 \\ 0 & 1 & 4 & 8 \\ 0 & 0 & 3 & 6 \\ 1 & 24 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \right\}$

We could, at this point, simply calculate $P\ Di\ P^{-1}$ and see if the matrix we get is equal to T, however, it is probably prudent and illustrative to check that the eigenvectors we have are all linearly independent (and thus form a basis for $\mathbb{R}^4$). Fortunately, we already have the eigenvectors handily arranged into the matrix named P, so establishing the linear independence is precisely the same as verifying that the vector equation $Px = 0$ has only the trivial solution.

```
In[248]:= NullSpace[P]
```

Out[248]= $\{\}$

Thus we see that these vectors form the required basis, and so we can be quite sure that diagonalization will work. We see that indeed it does.

```
In[249]:= P . Di . (P // Inverse) // MatrixForm
```

Out[249]= $\begin{pmatrix} 2 & -5 & 6 & 0 \\ 4 & -19 & 24 & 0 \\ 3 & -15 & 19 & 0 \\ 1 & -29 & 38 & 2 \end{pmatrix}$

In both of the above examples, our matrix has had exactly as many linearly independent eigenvectors in number as the dimension of the vector space upon which it acts.

That is, we had a 3×3 matrix with 3 linearly independent eigenvectors, and also a 4×4 matrix with 4 linearly independent eigenvectors. The process of diagonalization that we have used in both of these examples is one of using these eigenvectors as a basis for the underlying space. With this basis we find that we may apply the function simply by multiplying the coefficients of the basis vectors by fixed values (and hence the diagonal matrix), and then change back to the regular basis.

It should be clear then that as long as we have n linearly independent eigenvectors for an $n \times n$ matrix, M say, we can always follow this procedure, and we will thus always have a diagonalizable matrix. It turns out that this must always happen and that, in fact, an equivalent definition of an $n \times n$ matrix being diagonalizable is that it has exactly n linearly independent eigenvectors. To see that this is equivalent, we must see that a diagonalizable matrix M will always have exactly n linearly independent eigenvectors. We do not prove this here, but the proof is quite elementary and can be found in any good linear algebra text.

Why is diagonalization desirable? Well, diagonalization has applications in the solving of differential equations, as well as recurrence relations, and more besides. However, a more down to earth reason is that it can make taking powers of the matrix much simpler.

Without diagonalization, to compute T^n (which is just $TT \cdots T$ where there are n multiplications) we would have to compute n matrix multiplications. In the case where n is large, such a computation might be prohibitively time consuming, even for a computer.

In the case that T is diagonalizable, then we can write $T = PDP^{-1}$, and so

$$T^n = TT \cdots T = PDP^{-1}PDP^{-1} \cdots PDP^{-1} = PD^nP^{-1}$$

where each P^{-1} is canceled out by multiplication by P in the middle of the expression, leaving only a single P on the left and P^{-1} on the right, and n Ds all multiplied together in the middle forming D^n.

Now, the power of a diagonal matrix is very simple to take. One simply raises each entry on the diagonal to the power n. The reader is encouraged to experiment with some simple 2×2 or 3×3 examples of diagonal matrices to see why. For this it is actually more illustrative to perform the calculations by hand in order to see the pattern of multiplication.

Clearly this is a potentially very great decrease in time taken to find large powers of a matrix. Imagine trying to find T^{100} the long way, compared to just three matrix multiplications, plus some work to find eigenvectors and eigenvalues. This is not just a speedup of hand calculations, either. Matrix multiplication on computers can be similarly sped up this way, and it is quite likely that *Mathematica* itself uses such techniques (and likely more sophisticated ones as well).

We look at one more example of a diagonalizable matrix. This time, we do so in complex space. We take this example from the previous section as well.

In[250]:= `T = {{2, 1}, {-1, 1}}`
`T // MatrixForm`

Out[250]= $\{\{2, 1\}, \{-1, 1\}\}$

Out[251]= $\begin{pmatrix} 2 & 1 \\ -1 & 1 \end{pmatrix}$

We should remember, from our earlier computation, that this example had no real eigenvalues, but it did have complex eigenvalues. If we think about this matrix as a

linear transformation from $\mathbb{C}^2 \to \mathbb{C}^2$, then the roots of the characteristic equation, and thus the eigenvalues are $\frac{1}{2}(3 + i\sqrt{3})$ and $\frac{1}{2}(3 - i\sqrt{3})$.

In[252]:= `Di = DiagonalMatrix[Eigenvalues[T]]`
`P = Eigenvectors[T] // Transpose`

Out[252]= $\left\{ \left\{ \frac{1}{2}\left(3 + i\sqrt{3} \right), \, 0 \right\}, \, \left\{ 0, \, \frac{1}{2}\left(3 - i\sqrt{3} \right) \right\} \right\}$

Out[253]= $\left\{ \left\{ 1 + \frac{1}{2}\left(-3 - i\sqrt{3} \right), \, 1 + \frac{1}{2}\left(-3 + i\sqrt{3} \right) \right\}, \, \{1, \, 1\} \right\}$

We see that the vectors are linearly independent by finding the determinant of the matrix P, remembering that a nonzero determinant is equivalent to the expression $Tx = 0$ having only the trivial solution, and thus the column vectors being linearly independent. We could just as easily compute the null space of P, if we wish, but we'll use the determinant for a bit of variety.

In[254]:= `P // Det`

Out[254]= $-i\sqrt{3}$

We have exactly two linearly independent, complex eigenvectors, which must therefore form a basis for $\mathbb{C}^2$. We can diagonalize the matrix.

In[255]:= `P . Di . (P // Inverse) // Simplify // MatrixForm`

Out[255]= $\begin{pmatrix} 2 & 1 \\ -1 & 1 \end{pmatrix}$

And there we have it. Note that we need to simplify the expression. The result of the computation $P. Di .P^{-1}$ on its own is complicated, and messy, and too troublesome to print.

Finally we look at an example of a matrix that is not diagonal. We have not proven that an $n \times n$ matrix must have n linearly independent eigenvectors for it to be diagonalizable, although we have referred the reader to where such a proof may be found. We take this as read, however, and show a matrix with too few eigenvectors.

In[256]:= `T = {{6, -9}, {4, -6}}`
`T // MatrixForm`

Out[256]= $\{\{6, -9\}, \{4, -6\}\}$

Out[257]= $\begin{pmatrix} 6 & -9 \\ 4 & -6 \end{pmatrix}$

In[258]:= `vec = Eigenvectors[T]`
`val = Eigenvalues[T]`

Out[258]= $\{\{3, 2\}, \{0, 0\}\}$

Out[259]= $\{0, 0\}$

Interpreting the output, we have a repeated eigenvalue of 0, and two eigenvectors, $(3, 2)$ and $(0, 0)$. However, we know the zero vector cannot be an eigenvector. Furthermore, taking linear combinations of these vectors yields only a one-dimensional eigenspace. Let's look at this a little closer to see what's going on.

In[260]:= `CharacteristicPolynomial[T, \[Lambda]]`

Out[260]= λ^2

Well that's an easy one to solve. Clearly $\lambda = 0$ is the only eigenvalue with multiplicity of 2. We will now manually calculate the eigenvectors. Recall that we need to find the null space of the matrix $(T - \lambda I)$, which in this case collapses to T thanks to the eigenvalues of 0.

In[261]:= **NullSpace[T]**

Out[261]= $\{\{3, 2\}\}$

And here we see only a single vector, and hence only a one dimensional eigenspace. Because the multiplicity of the eigenvalue is greater than the dimension of the eigenspace we say that the eigenspace is *deficient*. This is sufficient to render our matrix T as not being invertible.

3.4 Exercises

1. a. Create the following vectors and matrices using the angle bracket $\langle \rangle$ notation.

   i. $\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$
   ii. (x, y, z, w)
   iii. $\begin{bmatrix} 1 & -2 & 7 \\ 8 & 4 & -5 \\ 7 & 9 & 2 \end{bmatrix}$

   Create the matrix twice, once using the column vectors and then using the row vectors.

   b. Create the following vectors using the **Table** function.
      i. $(\pi, \pi, \pi, \pi, \pi)$
      ii. $(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)$
      iii. $u = (u_1, \ldots, u_8)$ where $u_i = i^i$.
      iv. $v = (v_1, \ldots, v_{10})$ where v_i is the ith Fibonacci number.
      Create the following matrices using the **Table** function

   v. $\begin{bmatrix} e^2 & e^2 & e^2 & e^2 \\ e^2 & e^2 & e^2 & e^2 \\ e^2 & e^2 & e^2 & e^2 \\ e^2 & e^2 & e^2 & e^2 \end{bmatrix}$
   vi. $\begin{bmatrix} m_{1,1} & m_{1,2} & \cdots & m_{1,10} \\ m_{2,1} & m_{2,2} & \cdots & m_{2,10} \\ \vdots & \vdots & \ddots & \vdots \\ m_{5,1} & m_{5,2} & \cdots & m_{5,10} \end{bmatrix}$

   c. Create functions to produce the following general matrices.
      i. An $n \times n$ matrix $A = [a_{i,j}]$ where $a_{i,j} = \binom{n}{i} + \binom{n}{j}$ and $\binom{k}{m}$ are the binomial coefficients.
      ii. An $n \times m$ matrix $B = [b_{i,j}]$ where $b_{i,j} = i^3 + j^2$.
      iii. An $n \times m$ matrix $C = [c_{i,j}]$ where $c_{i,j} = f(i, j)$ for an arbitrary 2-variable function f.

2. a. Calculate the following matrix products

   i. $\begin{bmatrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \\ 4 & 8 & 12 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \end{bmatrix}$
   ii. $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}$

   b. Recall that the dot product between two vectors allows the calculation of the angle between the vectors with the formula

   $$u.v = |u| \cdot |v| \cdot \cos \theta$$

   Use this formula to find the angle between the following vectors. Also, find an in-built function for calculating the angle between vectors, and use it to verify your calculations.

   i. $(1, 0, 1)$ and $(1, 1, 0)$
   ii. $(2, 2, 2, 2)$ and $(3, 0, 3, 3)$
   iii. $(1, 2, 3, 4, 5, 6)$ and $(6, 5, 4, 3, 2, 1)$
   iv. $(1, 2, 3)$ and $(4, 5, 6)$

Note: For a vector v, the value $|v|$ may be calculated in *Mathematica* with the **Norm** command.

c. Recall that the vector cross-product is an operation that can only be performed on three-dimensional vectors, and that it calculates a new vector perpendicular to the two vectors used in its calculation.

Find a *Mathematica* command to perform cross-products, or write your own, and use it to calculate the cross-product of the following vectors. Verify that the cross-product is, indeed, perpendicular to the two vectors.

i. $(1, 0, 0)$ and $(0, 0, 1)$ ii. $(1, -2, 3)$ and $(3, 2, -1)$

3. a. Plot the following systems of equations, and attempt to identify from the plot whether the system is solvable. Solve the system using linear algebraic techniques. Plot the solution space if there is more than one solution.

i. $2x - 3y = -2$
     $2x + \ y = 1$
     $3x + 2y = 1$

ii. $-x + y = 16$
     $2x + y = -17$
     $6x + 2y = -56$

iii. $2x + z = 1$
       $-3x + z = 3$
       $2y + z = 4$
       $-2y + z = 2$

iv. $x + y + \ z = 3$
      $-x + y + 3z = 3$
      $4x + y - 2z = 3$

b. Solve the following linear systems and verify the solution. Express the solutions as vector equations.

i. $4x + 3y + 2z + \ w = 1$
     $x + 2y + 3z + 4w = 2$

ii. $2x + \ y + \ z + \ w = 1$
     $x + 2y + \ z + \ w = 2$
     $x + \ y + 2z + \ w = 3$
     $x + \ y + \ z + 2w = 4$

4. Let A_n be the $n \times n$ matrix where

$$a_{i,j} = \begin{cases} 2 \text{ if } i = j \\ 1 \text{ otherwise} \end{cases}$$

and let $b = (b_1, \ldots, b_n)$ be the n-vector where $b_i = i$.

Solve the linear system $A_n x = b$ for all values of n up to 10. Form a hypothesis about the solution for general n, and test that hypothesis for $n = 100$, and any other values of n that you choose.

5. Use row reduction to attempt to find inverses for the following matrices. If they are invertible, then give the inverse matrix, otherwise explain why they are not invertible.

a. $\begin{bmatrix} -90 & 30 & -4 \\ -54 & 57 & 17 \\ -27 & -69 & -40 \end{bmatrix}$

b. $\begin{bmatrix} 40 & 71 & 40 & 72 \\ -51 & -44 & 83 & -26 \\ -21 & 79 & 41 & -39 \\ -5 & 17 & -76 & -58 \end{bmatrix}$

$$
\text{c.} \quad \begin{bmatrix} 22 & -3 & -56 & -18 & -72 \\ 29 & 7 & 80 & -94 & -55 \\ -84 & -62 & -42 & -26 & 89 \\ 64 & -84 & -12 & 23 & 59 \\ 7 & -65 & -30 & 27 & 60 \end{bmatrix}
$$

6. Which of the following matrices will always give a unique solution to the vector equation $Ax = b$, which are invertible, and which will become the identity matrix in reduced row echelon form?

$$
\text{a.} \quad \begin{bmatrix} 15 & 33 & 0 & -97 & 47 & 48 \\ -70 & 0 & 0 & -84 & 66 & 65 \\ 0 & -19 & -65 & 13 & 0 & 0 \\ 0 & -61 & 65 & 0 & -55 & -78 \\ -38 & 0 & 0 & 64 & -42 & 0 \\ -53 & 0 & -41 & 0 & -9 & 0 \end{bmatrix}
\qquad
\text{c.} \quad \begin{bmatrix} 76 & 66 & 98 & -37 & 82 & -153 \\ 93 & -18 & -89 & -10 & -88 & 76 \\ 56 & -18 & -50 & 87 & -85 & -26 \\ -58 & 79 & 92 & 46 & 34 & -35 \\ 68 & 10 & -89 & -16 & 1 & 46 \\ 40 & -72 & 32 & 78 & -26 & -196 \end{bmatrix}
$$

$$
\text{b.} \quad \begin{bmatrix} -62 & 32 & 0 & 61 & 31 & -124 \\ 0 & -94 & 0 & 93 & -1 & 0 \\ -74 & 0 & 51 & 0 & -23 & -46 \\ -86 & -25 & 39 & 0 & -72 & -94 \\ -36 & -20 & 0 & 0 & -56 & -72 \\ 0 & -69 & 0 & -26 & -95 & 0 \end{bmatrix}
\qquad
\text{d.} \quad \begin{bmatrix} 25 & -16 & -38 & 57 & -32 & 99 \\ 94 & -9 & -18 & 27 & -74 & 29 \\ 12 & -50 & 87 & -93 & -4 & 44 \\ -2 & -22 & 33 & -76 & 27 & 92 \\ 50 & 45 & -98 & -72 & 8 & -31 \\ 10 & -81 & -77 & -2 & 69 & 67 \end{bmatrix}
$$

7. For the matrices in Exercise 6 that were invertible, find a sequence of row operations that will produce the matrix when performed on the identity matrix. (In other words, find the expression of the matrix as a product of elementary matrices).

8. a. For the following, state whether the first vector is a linear combination of the other vectors.

   i. $(1, 2), (-1, 4), (2, -3)$

   ii. $(1, 2, 3), (-1, 2, 3), (1, -2, 3)$

   iii. $(-1, 2, 3), (1, 2, -3), (1, 10, -3)$

   iv. $(-1, 2, 3, -4), (3, -2, -1, 5), (7, -2, 3, 7), (1, 2, 5, -3)$

   b. For the following, state whether the first polynomial is a linear combination of the other polynomials.

   i. $71\,x^4 - 136\,x^3 + 142\,x^2 + 264\,x + 265$

   $$-31\,x^4 - 54\,x^3 + 88\,x + 31$$

   $$-82\,x^4 - 13\,x^3 - 71\,x^2 - 86$$

   ii. $67\,x^5 - 31\,x^4 + 92\,x^3 + 44\,x^2 + 29\,x + 99$

   $$69\,x^5 + 8\,x^4 + 27\,x^3 - 4\,x^2 - 74\,x - 32$$

   $$-2\,x^5 - 72\,x^4 - 76\,x^3 - 93\,x^2 + 27\,x + 57$$

   $$-77\,x^5 - 98\,x^4 + 33\,x^3 + 87\,x^2 - 18\,x - 38$$

9. Which of the sets of vectors and polynomials from Exercise 8 are linearly independent, and which are linearly dependent?

10. The matrix space $M_2(\mathbb{R})$ has a standard basis:

$$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$$

And so has dimension 4.

a. Extend this notion to find a standard basis for $M_n(\mathbb{R})$. What is the dimension of $M_n(\mathbb{R})$? Use this to establish a correspondence between matrices in $M_n(\mathbb{R})$ and $\mathbb{R}^m$ for suitable m.

b. For the following sets of matrices calculate whether the first is a linear combination of the others. Which of these sets are linearly independent, and which are linearly dependent?

i. $\begin{bmatrix} -50 & 45 \\ -22 & -81 \end{bmatrix}, \begin{bmatrix} 50 & -16 \\ 10 & -9 \end{bmatrix}, \begin{bmatrix} 25 & 12 \\ 94 & -2 \end{bmatrix}, \begin{bmatrix} 31 & -80 \\ -50 & 43 \end{bmatrix}$

ii. $\begin{bmatrix} -36 & 1 \\ -99 & 53 \end{bmatrix}, \begin{bmatrix} -61 & 77 \\ -48 & 9 \end{bmatrix}, \begin{bmatrix} 24 & 86 \\ 65 & 20 \end{bmatrix}, \begin{bmatrix} -25 & 76 \\ 51 & -44 \end{bmatrix}$

iii. $\begin{bmatrix} -67 & 16 & 60 \\ 22 & 9 & -95 \\ 14 & 99 & -20 \end{bmatrix}, \begin{bmatrix} 82 & 18 & -62 \\ 72 & -59 & -33 \\ 42 & 12 & -68 \end{bmatrix}, \begin{bmatrix} -70 & 29 & -1 \\ 41 & 70 & 52 \\ 91 & -32 & -13 \end{bmatrix}, \begin{bmatrix} -14 & 21 & 19 \\ 60 & 90 & 88 \\ -35 & 80 & -82 \end{bmatrix}$

11. A vector in $\mathbb{R}^3$ may be rotated around any of the three axes. Rotation around the z-axis is equivalent to rotating in the xy plane. Similarly rotating around the y-axis is equivalent to rotating in the xz plane and rotating around the x-axis is equivalent to rotating in the yz plane. We call these rotations R_{xy}, R_{xz} and R_{zy}, respectively.

a. Convince yourself that each of these rotations is a linear transformation on $\mathbb{R}^3$.

b. Construct rotation matrices for R_{xy}, R_{xz}, and R_{zy} for a rotation by an arbitrary angle θ.

c. Show that any one of these rotations can be realized as a composite transformation using only the other two and their inverses.

d. How would you rotate a vector around an arbitrary line in $\mathbb{R}^3$?

e. Plot a cube with a face pointing in the direction of the vector $(1, 1, 1)$.

12. Find the eigenvectors and eigenvalues of the following matrices.

a. $\begin{bmatrix} -300 & 296 & -36 & 24 & 0 \\ -309 & 305 & -36 & 24 & 0 \\ -365 & 356 & -40 & 27 & 0 \\ -699 & 680 & -92 & 61 & 0 \\ 328 & -320 & 44 & -24 & 4 \end{bmatrix}$

b. $\begin{bmatrix} 1277 & -336 & -1668 & 1572 & 288 \\ 744 & -187 & -972 & 900 & 168 \\ 1634 & -432 & -2138 & 2021 & 370 \\ 548 & -144 & -718 & 679 & 124 \\ 1722 & -456 & -2259 & 2133 & 395 \end{bmatrix}$

13. We may generalise the notion of square roots to matrices by defining the *square root* of a (square) matrix, M say, as a matrix A such that $A \cdot A = M$.
Using this definition of a matrix square root:

a. Find the square root of the following matrices.

$$
\text{i. } \begin{bmatrix} 16 & 0 & 0 & 0 \\ 0 & 49 & 0 & 0 \\ 0 & 0 & 64 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\qquad
\text{ii. } \begin{bmatrix} 36 & 0 & 0 & 0 & 0 & 0 \\ 0 & 16 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 49 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 81 \end{bmatrix}
$$

What is the square root of an arbitrary diagonal matrix? (Justify your answer)

b. Find the square root of the following diagonalizable matrices from Sections 3.3.3 and 3.3.4.

$$
\text{i. } \begin{bmatrix} -1 & 2 & 0 \\ -6 & 6 & 0 \\ 4 & -2 & 1 \end{bmatrix}
\qquad
\text{ii. } \begin{bmatrix} 2 & -5 & 6 & 0 \\ 4 & -19 & 24 & 0 \\ 33 & -15 & 19 & 0 \\ 17 & -29 & 38 & 2 \end{bmatrix}
$$

Hint: Use the fact that these matrices are diagonalizable, and the properties of powers of diagonalizable matrices.

14. Diagonalize the following matrices.

$$
\text{a. } \begin{bmatrix} 9 & -45 & 9 & -9 \\ 0 & 0 & 0 & 0 \\ 2 & -10 & 2 & -2 \\ 3 & -15 & 3 & -3 \end{bmatrix}
\qquad
\text{b. } \begin{bmatrix} 4 & 10 & -16 & 44 \\ -3 & 15 & -9 & 24 \\ -8 & 16 & 2 & -4 \\ -2 & 4 & -1 & 5 \end{bmatrix}
\qquad
\text{c. } \begin{bmatrix} 18 & -8 & 0 & -16 \\ 18 & -7 & 0 & -18 \\ 0 & 0 & 2 & 0 \\ 8 & -4 & 0 & -6 \end{bmatrix}
$$

$$
\text{d. } \begin{bmatrix} -300 & 296 & -36 & 24 & 0 \\ -309 & 305 & -36 & 24 & 0 \\ -365 & 356 & -40 & 27 & 0 \\ -699 & 680 & -92 & 61 & 0 \\ 328 & -320 & 44 & -24 & 4 \end{bmatrix}
$$

3.5 Further Explorations

1. A *positive* matrix $A = (a_{i,j})$ is a matrix over $\mathbb{R}$ where $a_{i,j} > 0$ for every i and j. In other words it is a real matrix with all positive entries. The Perron–Frobenius theorem states that such a matrix has a unique, largest, real eigenvalue, and a corresponding eigenvalue with all positive entries.
More technically stated:

Theorem 3 (Perron–Frobenius). *Let $A = (a_{i,j})$ be an $n \times n$ positive matrix. Then the following hold.*

> a. There is a unique eigenvalue $r \in \mathbb{R}$ with the property that for every other eigenvalue λ it is the case that $|\lambda| < r$.
> b. The eigenvalue r is a simple root of the characteristic polynomial, and so is a degree 1 eigenvalue.
> c. There is an eigenvector $v = (v_1, \ldots, v_n)$ corresponding to the eigenvalue r has the property that $v_i > 0$ for every $1 \le i \le n$.
> d. The eigenvector v (above) is the only eigenvector with nonnegative entries.

The eigenvalue r is sometimes called the Perron root *or the* Perron–Frobenius eigenvalue.

Be aware that the eigenvalues λ in 1a could potentially be complex, in which case the absolute value is the complex modulus. Similarly, as a consequence of 1d any other eigenvector of A (i.e., an eigenvector corresponding to a different eigenvalue) must have either a negative entry, or a complex one.

2. Recall recurrence relations from Section 1.3.3. The solution to a first order recurrence relation is quite straightforward. We may use a similarly straightforward approach to systems of recurrence relations (often called *difference equations*). Suppose we have n recurrence relations $a_1(k), \ldots, a_n(k)$ which are interlinked in some way. That is,

$$a_1(k) = \lambda_{1_1} a_1(k-1) + \lambda_{1_2} a_2(k-1) + \cdots + \lambda_{1_n} a_n(k-1)$$
$$a_2(k) = \lambda_{2_1} a_1(k-1) + \lambda_{2_2} a_2(k-1) + \cdots + \lambda_{2_n} a_n(k-1)$$
$$\vdots$$
$$a_n(k) = \lambda_{n_1} a_1(k-1) + \lambda_{n_2} a_2(k-1) + \cdots + \lambda_{n_n} a_n(k-1)$$

We may decompose this into something very reminiscent of a linear system. Let

$$A := \begin{bmatrix} \lambda_{1_1} & \lambda_{1_2} & \ldots & \lambda_{1_n} \\ \lambda_{2_1} & \lambda_{2_2} & \ldots & \lambda_{2_n} \\ \vdots & \vdots & \ddots & \vdots \\ \lambda_{n_1} & \lambda_{n_2} & \ldots & \lambda_{n_n} \end{bmatrix} \text{ and } a(k) := \begin{bmatrix} a_1(k) \\ a_2(k) \\ \vdots \\ a_n(k) \end{bmatrix}$$

then

$$a(k) = A \cdot a(k-1)$$

and by the same argument we used in Section 1.3.3 we can see that

$$a(k) = A^k \cdot a(0)$$

How would you ascertain the long-term behavior of such a system?

In the special case that the elements of every row of A add to 1, we have a representation of a time-homogeneous Markov chain. This is not actually a difference equation, however. In this case we are representing some system with n states that between which it may transition. Each state is represented by a row and a column. The element $a_{i,j}$ (being the element in row i and column j) is the probability that the system will move to the state represented by column j if it is currently in the state represented by row i.

The similarity to difference equations is in taking higher powers of the matrix A in order to obtain a solution. If we have an n-vector, v_0 say, whose elements sum to 1, we can consider it to be a probability distribution of the states. That is, we consider

the element v_i as being the probability that the state is in the state represented by row i, then the vector $v_1 = A \cdot v$ is the probability distribution of the system after a single transition, and the vector $v_k = A^k \cdot v$ is the probability distribution of the system after k transitions.

How would you ascertain the long term behavior of such a system?

3. Recall differential equations from Section 2.2.3. We may have interrelated differential equations in a similar manner to our difference equations above. Such equations are sometimes called *coupled* differential equations. Suppose we have the following system of differential equations.

$$y_1'(t) = \lambda_{1_1} y_1(t) + \lambda_{1_2} y_2(t) + \cdots + \lambda_{1_n} y_n(t)$$
$$y_2'(t) = \lambda_{2_1} y_1(t) + \lambda_{2_2} y_2(t) + \cdots + \lambda_{2_n} y_n(t)$$
$$\vdots$$
$$y_n'(t) = \lambda_{n_1} y_1(t) + \lambda_{n_2} y_2(t) + \cdots + \lambda_{n_n} y_n(t)$$

We construct a linear system. Let

$$A := \begin{bmatrix} \lambda_{1_1} & \lambda_{1_2} & \cdots & \lambda_{1_n} \\ \lambda_{2_1} & \lambda_{2_2} & \cdots & \lambda_{2_n} \\ \vdots & \vdots & \ddots & \vdots \\ \lambda_{n_1} & \lambda_{n_2} & \cdots & \lambda_{n_n} \end{bmatrix}, \quad f(t) := \begin{bmatrix} f_1(t) \\ f_2(t) \\ \vdots \\ f_n(t) \end{bmatrix} \text{ and } f'(t) := \begin{bmatrix} f_1'(t) \\ f_2'(t) \\ \vdots \\ f_n'(t) \end{bmatrix}$$

Then our system of differential equations can be written as

$$f'(t) = A \cdot f(t)$$

If $\mu_1, \ldots, \mu_n$ are distinct eigenvalues of A with corresponding eigenvectors $v_1, \ldots, v_n$ then

$$f(t) = \sum_{i=1}^{n} c_i e^{\mu_i t} v_i$$

is the general form of the solution to where c_i are arbitrary constants. You can easily show that $e^{\mu t} v$ is a solution, and it's an easy step from there to see that a linear combination of solutions must also be a solution.

For second-order differential equations, $ay'' + by' + cy = 0$, we introduce a new function x such that $x = y'$ and we now have the following system of equations

$$y' = x$$
$$x'(t) = -\frac{b}{a}x - \frac{c}{a}y$$

which we can now evaluate using the matrix method above. Doing so will verify the characteristic polynomial method.

Extend this method to deal with higher-degree differential equations, and systems of coupled higher-degree differential equations. How might you cope with inhomogeneous cases?

Chapter 4
Visualization and Geometry: A Postscript

We conclude with a brief chapter on visualization and geometry. We will look at a number of tools which can be used either to enhance the visuals we have already been producing during the book (with `Plot` and the like), as well as some more general functions for producing images. We will finish up with a very quick look at some "interactive geometry" using the software known as *Cinderella*.

4.1 Useful Visualization Tools

Mathematica contains some useful tools for visualization that we discuss briefly here.

4.1.1 Interactive Mathematica and Demonstrations

Mathematica comes with some tools to produce interactive elements within a *Mathematica* notebook. They are perhaps a little cumbersome to use, and are, sadly, outside the scope of this book to detail. They are, nonetheless, deserving of attention, and the reader is encouraged to look further into them. The Documentation Center pages named: "Interactive Manipulation" as well as "Introduction to Manipulate" and "Introduction to Dynamic" to begin with. The Documentation Center entry "Build an Interactive Application" is also worth looking at.

An extensive library of interactive *Mathematica* demonstrations can be found online at http://demonstrations.wolfram.com/, as well as a free player for the demonstration files available at http://www.wolfram.com/cdf-player/. Anybody with a *Mathematica* license, may create and distribute their own demonstrations. Readers from an educational background might be particularly interested in this. The Documentation Center page "Create a Demonstration" should be a good place to start for the interested reader. The player will also display and print mathematica notebook files, with all inputs and outputs that were present at the time the file was saved, but will not allow them to be modified in any way.

4.1.2 Animation

For plots with a single parameter, the **Animate** function may be used as a "field expedient" method of producing a quick interactive *Mathematica* element. It's also useful for creating animations.

For example, to see the pointwise convergence of the bell-curve like function e^{-kx^2} as $k \to \infty$, we could issue the following command.

In[1]:= `Animate[`
    `Plot[E^(-k*x^2), {x, -5, 5}, PlotRange -> Full],`
    `{k, 1, 10}`
    `]`

Out[1]=
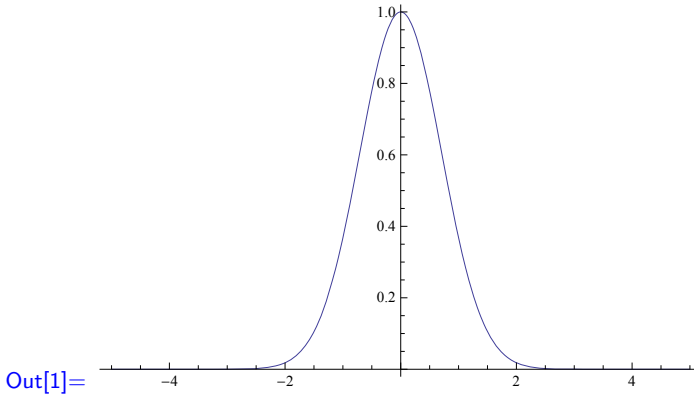
Note that the actual notebook will have a frame around the graphic, with a slidebar for controlling the value of `k`, as well as play, pause, and other controls for the animation.

We may use this technique to better see the change of parameter of a parametric equation. We revisit an example from Section 2.2.4 to demonstrate this.

In[2]:= `Animate[`
    `PolarPlot[Sin[8*theta/5], {theta, 0, k}, PlotRange -> 1]`
    `{k, 0.01, 10 Pi}`
    `]`

Out[2]=
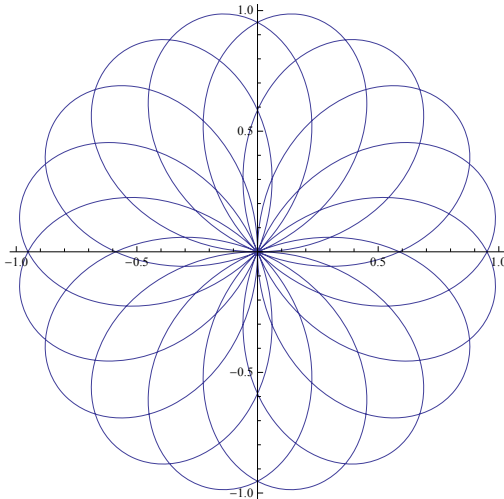
The reader should refer to the Documentation Center for more information.

4.1.3 Text and Labeling

Many of the plots we produce may be enhanced with the addition of text to the plot to explain or label. Up until now the only labeling our plots have had have been numeric values, and tickmarks to indicate sub-values. We show here a variety of labeling methods; the *Mathematica* plotting functions have a number of built-in labeling options, and the reader is encouraged to reference the Documentation Center regarding for more details, or to find further options not mentioned here.

First, we look at labeling the axes. The *Mathematica* plotting functions, by default, will not label axes, but there is an option we may set if we wish it to happen. This option is the **AxesLabel** option. Usually when we plot a function, we are plotting an expression with a well-defined unknown (or unknowns), and so the vertical axis corresponds to an easily identified variable. The vertical axis, however, often does not correspond to an easily identifiable variable, although convention is often to consider it the "y" axis. By default, if we use **AxesLabel -> Automatic** the **Plot** command will label the vertical axis with the obvious variable, and will not label the vertical axis. (The **Plot3d** function behaves analogously by labeling the obvious two horizontal axes, but not the vertical axis).

In[3]:= **Plot[Sin[\[Theta]], {\[Theta], 0, 2 Pi}]**
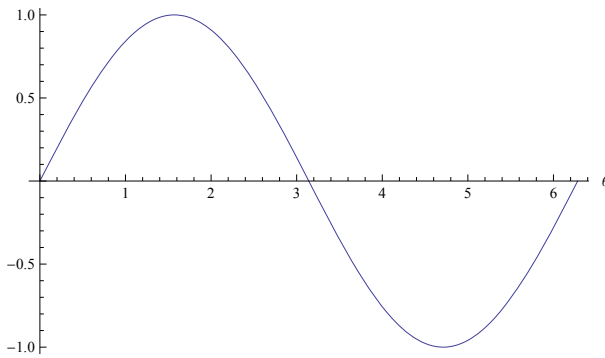
We see that **Plot** has labelled the horizontal axis with a θ to the right of the axis. If we wish, we may choose our own labels by giving a list to the **AxesLabel** option. That list will be interpreted as the horizontal, and the vertical axis labels, in that order.

In[4]:= **Plot[Sin[\[Theta]], {\[Theta], 0, 2 Pi},**
        AxisLabel -> {\[Theta], Sin[\[Theta]]}
    ]
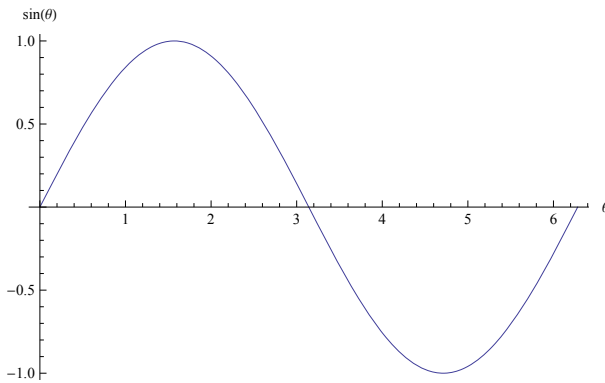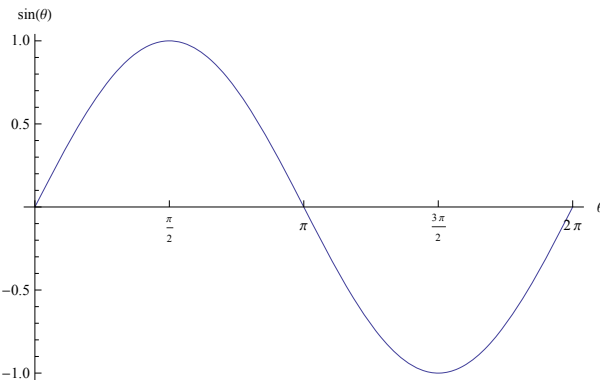
Another option we may change, with regards to marking and labeling of plots, is to change which values are marked along the axes. In our example above, *Mathematica* has helpfully marked out 1, 2, etc all the way up to 6, and then a bit for us on the horizontal axis, with smaller ticks to indicate increments of 0.2. However, we are plotting the sin function, and so markings of $\pi/2$ would be more appropriate.

The markings on any axis are called *tickmarks*, and are modified using the `Ticks` option. The simplest version of this is to specify either the `Automatic` or `None` keywords for the `Ticks` option. Both of these options do as we should expect; to provide the behavior we see by default and to completely remove tickmarks respectively. We do not demonstrate these here.

If we do not specify either of the above key words, we may give a list to the `Ticks` option. This list will be interpreted as a list of tickmark specifications, one for each axis. There are several forms the tickmark specifications can take, but we will only concern ourselves with one in particular; a list of the values to be marked. Note that this specification is, itself, a list that will be an element of the list of axis specifications.

```
In[5]:= Plot[Sin[\[Theta]], {\[Theta], 0, 2 Pi},
          AxisLabel -> {\[Theta], Sin[\[Theta]]},
          Ticks -> {{0, Pi / 2, Pi, 3 Pi / 2, 2 Pi}}
        ]
```


In our case we wanted the tickmarks on the horizontal axis to be shown at regular intervals of $\frac{1}{2}\pi$ in order to have the critical points of the sine curve suitably marked. We did not need, nor wish, for the vertical axis to be marked any differently from usual, and so we simply omitted a specification for that axis, and it was left as default.

For the sakes of demonstration, let us produce a plot of the sin function, marking off all multiples of $\pi/4$, as well as the corresponding values of sin at those values. Note that there are only five distinct values of sin for those values of θ.

```
In[6]:= With[{values = Table[k * Pi / 4, {k, 0, 8}]},
          Plot[Sin[\[Theta]], {\[Theta], 0, 2 Pi},
          Ticks -> {values, Sin /@ values}]
        ]
```
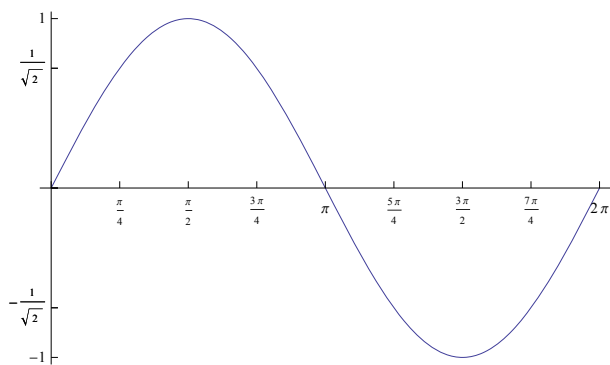
Out[6]=

We may, if we wish, attach a label to an entire plot. The **PlotLabel** plot option
allows us to do this.

```
In[7]:= With[{values = Table[k * Pi / 4, {k, 0, 8}]},
          Plot[Sin[\[Theta]], {\[Theta], 0, 2 Pi},
            Ticks -> {values, Sin /@ values},
            PlotLabel -> "A plot of the sine function"
          ]
        ]
```
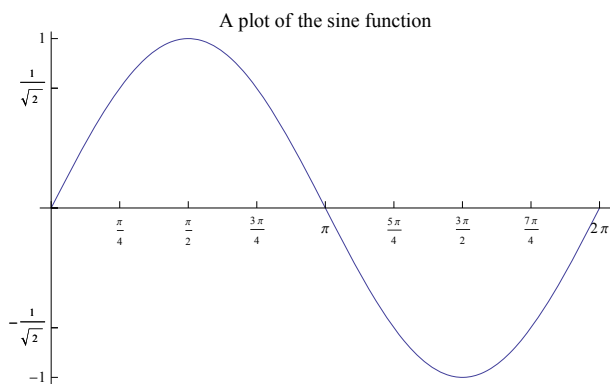

Out[7]=

We pause to mention that we may label any *Mathematica* output (not just plots) with
the use of the **Labeled** function, however we will not look at that function here. The
interested reader is encouraged to look the function up in the Documentation Center.

Getting back to plotting, we may, if we wish, place arbitrary text anywhere on a plot
using the **Text** and **Graphics** functions. The **Graphics** function is a generic function for
producing 2-D graphics, and serves to collect together functions which produce graphics
primitives, such as **Text**, **Circle**, **Polygon**, etc. We will look at some more of these in
Section 4.1.4, but for now we'll look at **Text** specifically.

As an example, let's look at the cubic $x^3 + x^2 - 5x - 5$, which we call p. Through
simple analysis (which we leave as an exercise to the reader) we discover that p has
zeroes at $x = 1$ and $x = \pm\sqrt{5}$. We also discover that p has turning points at $(1, -8)$
and $(-5/3, 40/27)$ which are a local minimum and maximum, respectively. We intend
to plot this cubic, and label these interesting points.

To begin with, we plot the curve, with as many markings as we can manage. We use
the **Ticks** option to mark the zeroes, and we use a new option, **Epilog**, to add two
points onto the plot at the critical points.

```
In[8]:= p = x^3 + x^2 - 5 x - 5
        p1 = Plot[p, {x, -3.25, 2.75},
```

```
            PlotLabel -> y == p,
            AxesLabel -> {x, y},
            Ticks -> {{-Sqrt[5], -1, Sqrt[5]}},
            Epilog -> {
              PointSize[0.015], Point[{-5/3, 40/27}], Point[{1, -8}]
            }
          ]
```

Out[8]= $-5 - 5x + x^2 + x^3$

$$y = x^3 + x^2 - 5x - 5$$
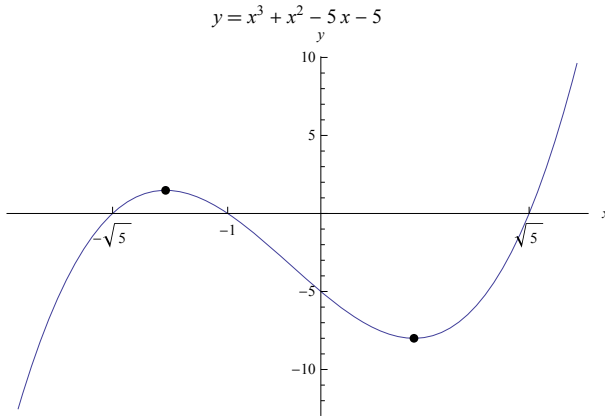
Out[9]=

Now we mark the critical points. It should be noted here that the **Graphics** function produces a plot, just as do any of our other plotting functions. If we want to put the text near the plot of our curve, we need to use the **Show** function. It is for this reason that we assigned our first plot to the variable **pl**, above.

We want the label for the local maximum to sit immediately above the point in question, and the label for the local minimum to sit immediately below the point. We use the **Text** function to achieve this, although it produces a lot of whitespace here.

```
In[10]:= cp = Graphics[
           {
             Text[{-5/3, 40/27}, {-5/3, 40/27},{0, -1.5}]
             Text[{1, -8}, {1, -8}, {0, 1.5}]
           }
         ]
```

$\{-\frac{5}{3}, \frac{40}{27}\}$

Out[10]= $\{1, -8\}$

The **Text** function takes the form **Text[*expr, coord, offset*]**, where *expr* is the expression whose text is to be plotted, *coord* is a list describing the coordinates at which to plot the expression, and *offset* is an offset from that position. In this case, we have printed the coordinates, at their location, but offset slightly up, or down, so that they're not directly on top of the plot curve, or point. Note that the **Text** function needs to be use inside the **Graphics** function.

The offset describes not just how far away from the specified coordinates to put the text, but also how to align the text. In our case **{0, -1.5}** specifies that the expression is to be centered 1.5 units above the specified coordinates. Similarly, **{0, -1.5}** specifies that the expression is to be centered 1.5 units below the specified coordinates. Specifying **{a, 0}** or **{-a, 0}** will align the right or left edge of the expression at the specified coordinates, respectively, with an appropriate shift by *a* units. Diagonal alignment is possible with a non-zero entry for each element of the list.

All that remains now, is to look at the final plot.

In[11]:= **Show[pl, cp]**

Out[11]=
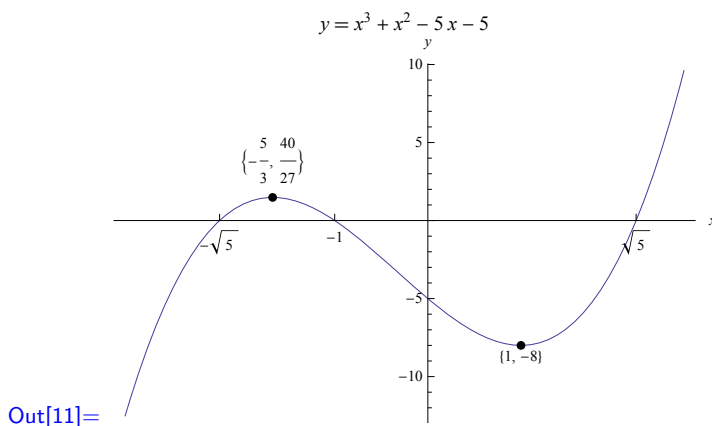
As always, we should clean up after ourselves, and clear our global definitions.

In[12]:= **ClearAll["Global`*"]**

4.1.4 Polygons, Polyhedra, and so on

In addition to plotting functions (implicit or otherwise), there may be other things we wish to visualize. We have already seen the **Graphics** function in Section 4.1.3, when we used it to add text to plots. We look at some more uses of this function, as well as some other potentially useful visualization functions.

We start by visualizing basic polygons and polyhedra. The **Polygon** function, which must be used in conjunction with the **Graphics** function, takes a list of points which it then uses to create a polygon.

In[12]:= **Graphics[{Blue, Polygon[{{0, 0}, {3, 1}, {2, 1}}]}]**
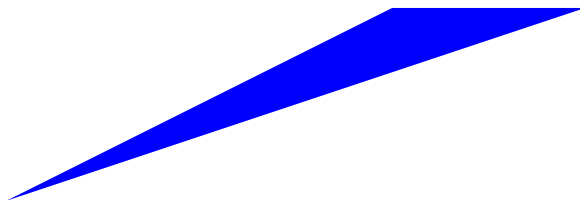
With a judicious use of the **Table** command we can create a regular 11-gon. We find eleven equidistant points on the unit circle. These points have the form $(\cos(2k\pi/11), \sin(2k\pi/11))$ with $1 \le k \le 11$. Note the **Blue** directive, which appears as a list element before the **Polygon** function.

```
In[13]:= With[
            {V = Table[{Cos[2 * Pi * k / 11], Sin[2 * Pi * k / 11]}, {k, 1, 11}]},
            Graphics[{Blue, Polygon[V]}]
         ]
```
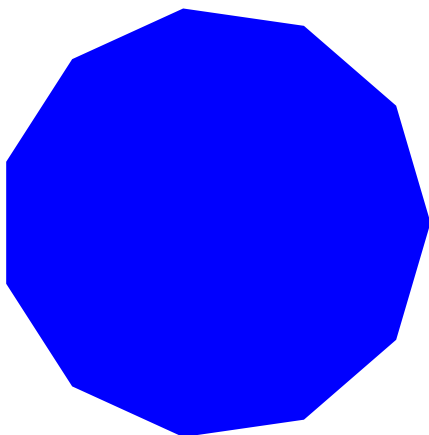

Out[13]=

The order of the vertices is important here. The **Polygon** function draws the boundary of the polygon by drawing lines between the vertices in the order they appear (1st to 2nd then 2nd to 3rd and so on until the nth and then finally from the nth to the 1st). If the boundary lines intersect, then the interior changes appropriately.

```
In[14]:= Graphics[{Blue, Polygon[{{0, 0}, {1, 0}, {0, 1}, {1, 1}}]}]
```
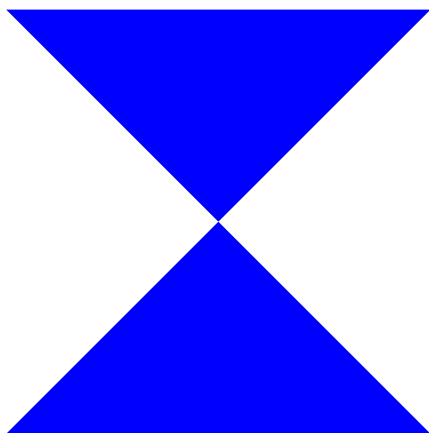

Out[14]=

Note here that there is also a three-dimensional variant of **Graphics** called, unsurprisingly, **Graphics3D**. We will not look at that here, and the interested reader is, as always, encouraged to look it up in the Documentation Center.

We will, however, look at some three-dimensional visualization. Specifically, we will visualize the regular polyhedra. The function in question is the **PolyhedronData**, and takes a string as an argument, describing the polyhedron we wish to see.

In[15]:= **PolyhedronData["Icosahedron"]**
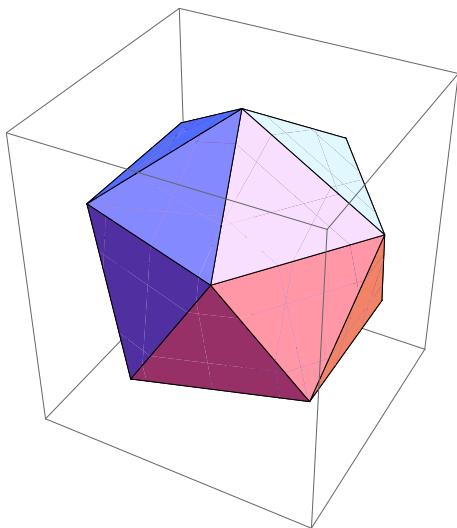
Out[15]=

In fact, the **PolyhedronData** function is much more complicated than just a function to plot regular polyhedra. The reader is encouraged to look into this themselves, but we will show one other use of this function. We will use it to see the two dimensional net for the icosahedron.

In[16]:= **PolyhedronData["Icosahedron", "NetImage"]**
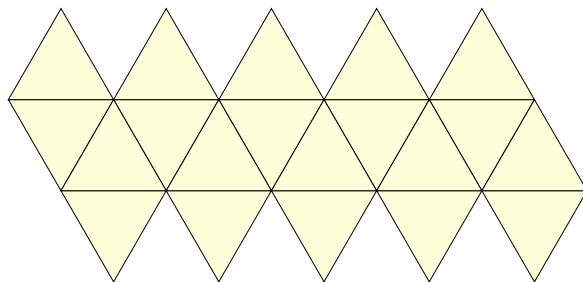
Out[16]=

We leave the discussion of polygons, polyhedra and so on there. *Mathematica* has a great number of functions for graphic visualization, and we could dedicate an entire book if we wanted to. The reader should be capable of finding more information through the Documentation Center.

4.2 Geometry and Geometric Constructions

Mathematica, somewhat surprisingly, does not come with a geometry package capable of performing geometric computations.[1] We have seen the `Graphics` function, which is capable of drawing geometric shapes, but not, apparently, performing geometric constructions of the sort we will look at in this section. As such, we will shift our attentions from *Mathematica* to the interactive geometry package *Cinderella*.

Interactive geometry packages are wonderful tools for learning geometry; the points and lines created may be moved about the screen freely, and the software computes the changes in real time. This allows us to see a very large number of different cases very quickly, and even of easily realizing truths which are not so plain with a pen-and-paper construction. Unfortunately, the dynamic nature of such software is difficult to convey in the strictly static nature of a book such as this one. We will persist nonetheless, but the reader should be aware that we are not even really scratching the surface in this section. Exploration is highly encouraged, and we hope that this small taste wets the readers appetite.

Cinderella is available from http://www.cinderella.de/. At the time of writing, a basic version of the software was available for download and use free of charge, with an advanced version available for use under a paid license. All constructions found in this book are possible with the basic version. A similar, open source program named *GeoGebra* is available from http://www.geogebra.org/.

We do not attempt to explain the operation of the *Cinderella* software here; we leave that up to the reader. Instead, we present the logic of the constructions themselves, along with the images produced by *Cinderella*, and trust the reader to be able to follow the reasoning.

4.2.1 Constructing a Circle Given Three Points

Given three points, we may find a unique circle that passes through all three of those points. This fact is related to the fact that the perpendicular bisector of any two chords on a circle will always intersect at the center of that circle.

Let us start with three points.

---
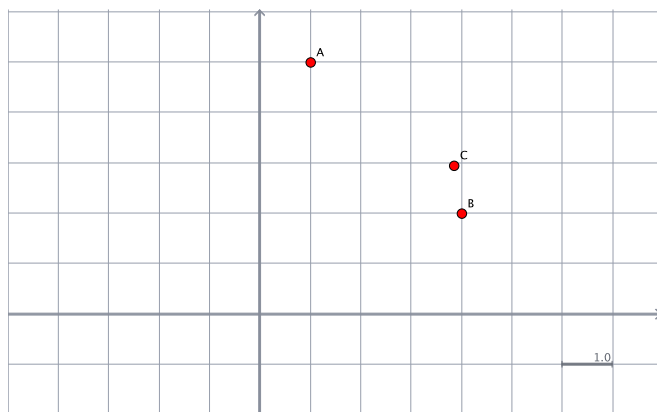
[1] At the very least, the author was unable to identify one.

These could, in principle, quite happily be any three points. However, the author has taken pains to choose three particular points that lie on a familiar circle.

$$A = (1, 5), B = (4, 2) \text{ and } C = \left(1 + \frac{9\sqrt{10}}{10}, 2 + \frac{3\sqrt{10}}{10}\right)$$

the reader should use any three points they so choose.

The way in which we construct the circle is to suppose that our three points lie on the surface of some circle. If this were the case, then we could create up to three chords of that circle by drawing line segments between the three points.
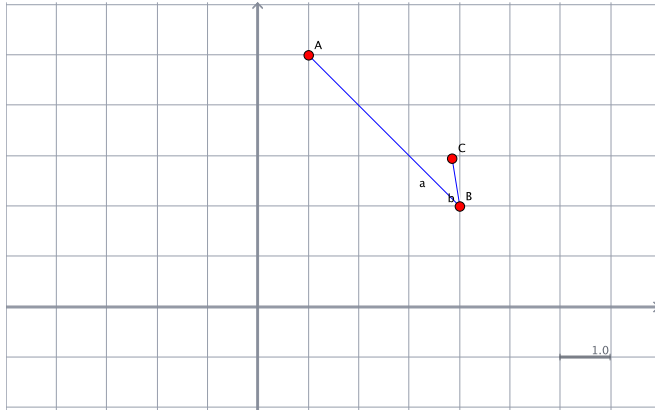
In order to find the center of our hypothetical circle, we need only two of these chords. For simplicity's sake, we chose the chords AB and BC. Observe, however, that whichever two chords we chose would always have had a point in common. This is critical. We now construct the perpendicular bisector of each of these chords. First we need to find the midpoints of the two line segments.
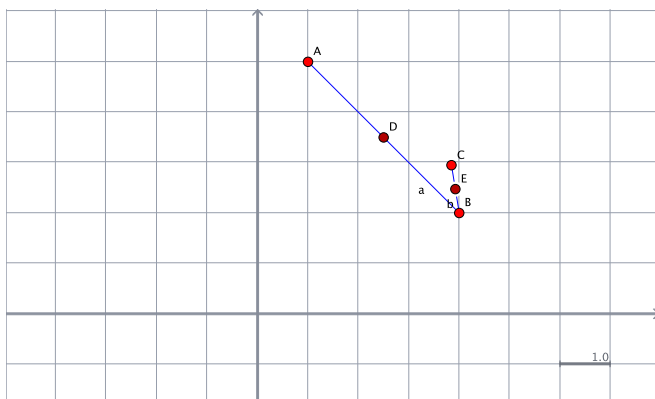
The perpendicular bisector of the chord a is the line perpendicular to a that passes through the midpoint of a, which is the point D. Similarly, the perpendicular bisector of the chord b is the line perpendicular to it that passes through its midpoint, E. We construct these lines now.
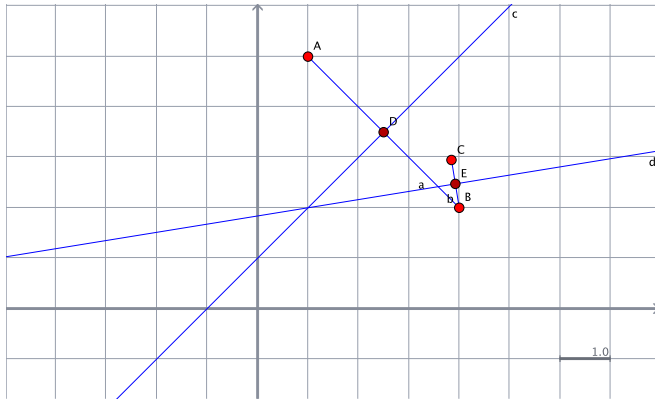
So now we have an intersection of two perpendicular bisectors drawn from chords on a hypothetical circle. How do we know this circle even exists? One way would be to try to draw the circle whose center is at the intersection of the two bisectors and using one of the points as a radius. If we did this then maybe that circle would also pass through the other three points in which case we would have our desired circle. Doing this, however, would not guarantee that the same construction would work for a different three points.

So instead, let's think about these bisector lines for a bit. If we take any point, P say, on the bisector of, say, AB, then we know that point is equidistant from A and B. If we imagine the isosceles triangle PAB we should be convinced of this fact. In particular, then, the point of intersection between our two bisectors—let's call it F—is equidistant to A and B.
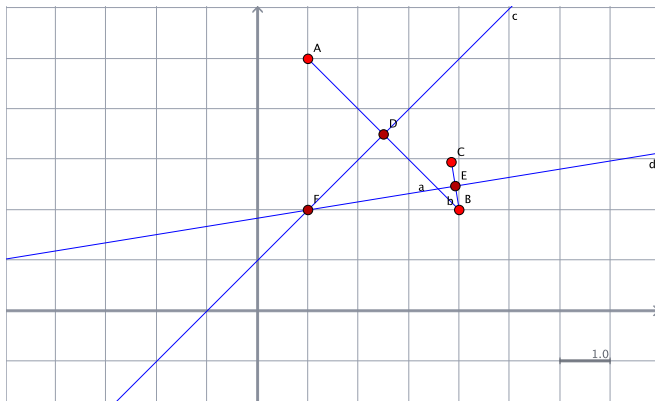
By a similar argument, applied to the chord BC, the intersection point F is also equidistant from the points B and C. So it must be the case that all three of our points are equidistant from F, which is the same as saying that they lie on a circle whose center is F and whose radius is the distance between the points and F.

We now know that not only is our hypothetical circle actually a real circle, but that it makes no difference which three points we choose. This is because as we observed above, there must always be a common point that our two chords share. The proof that this circle is unique is left as an exercise for the reader. All that remains is to draw the circle.
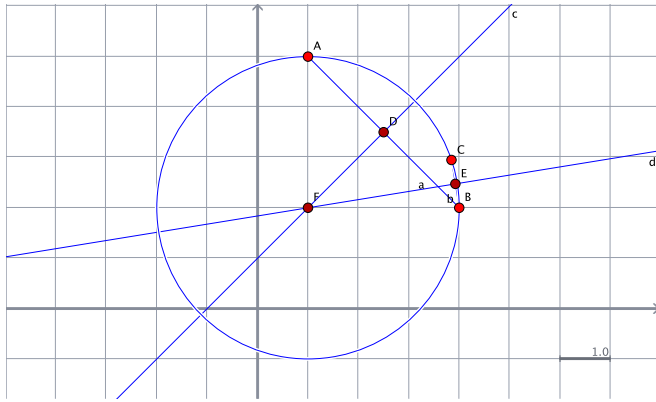
The reader should recognise the circle of radius 3, centered at the point $(1, 2)$ that we plotted several times in Section 2.2.4.

As is usually the case with these things, *Cinderella* as an in-built tool which will calculate the unique circle from three arbitrary points. We might simply have used thistool for computing a circle given three points, and it would have produced the correct circle for us. If we had done so, however, we would not have led us to the proof of correctness of the above technique. By taking the longer route, we have led ourselves to a greater understanding

Nonetheless, doing so allows us to see our circle without the extra construction lines and points.
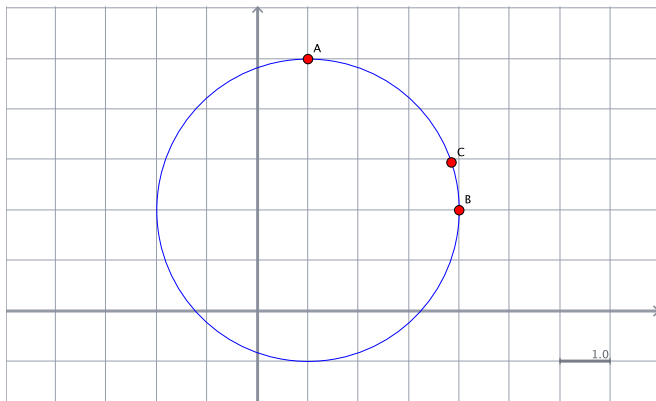
4.2.2 Constructing the Orthocenter of a Triangle

We perform one more construction. We start with a triangle, the points of which have been chosen arbitrarily for no particular reason other than they fit fairly neatly within the a golden rectangle, and thus will look good on the page. In other words, the choice was made with an eye to the constraints of publishing, not of geometry. As with the previous example, the choice of points is not important; all that is important is that they form a triangle.
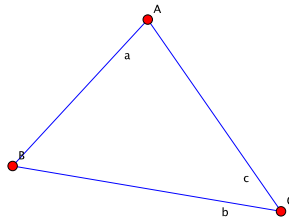
To construct the orthocenter of the triangle, we draw the line perpendicular to each side and passing though that side's opposite corner. In this case, we take the lines perpendicular to *a* passing though *C*, the line perpendicular to *b* passing though *A*, and the line perpendicular to *c* passing through *B*.
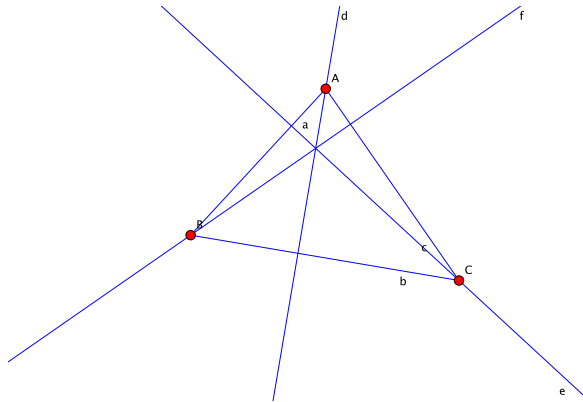
These lines, appear to intersect at a single point. In fact, performing this construction for any given triangle will always produce a single intersection point. We call this point the orthocenter. We give no proof that the lines always intersect at a single point, nor does our construction shed any light on why this might be. What we have done is to produce but a single example.

A single example, whether on paper or on screen, should not be very compelling, however the interactive nature of *Cinderella* allows us to see many more configurations of the three points, very quickly. If we drag points around the screen we will see a dynamically changing triangle, with the perpendicular lines always, apparently at least, meeting at a single point. This should be at least a little more compelling than a single image.

If we drag points around for long enough, we may even discover a configuration that looks something like the following, with the so called "center" lying outside the triangle. Now we're beginning to see both the complexities of the geometric construction we're performing, as well as the benefits of interactive geometry. This is something that we might sink our teeth into and get to the bottom of.
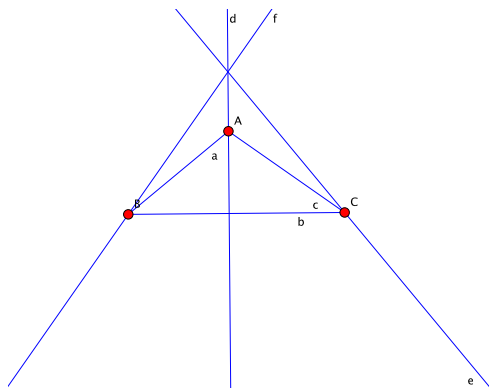
If we look closely, we see that the line f is acutally perpendicular to the line c and passes through the point B, just as it's supposed to. Similarly the line e is perpendicular to the line a and passes through the point C. The problem lies in the fact that the triangle ABC has, in this specific case, an obtuse angle. Verifying this fact is left as an exercise to the reader.

We leave the discussion of geometry there. Take note that the interactive nature of *Cinderella* has allowed us to quickly find and see degenerate configurations that likely would not have been immediately obvious with more static methods. The reader is, as always, encouraged to experiment on their own. Have fun!

Appendix A
Sample Quizzes

A.1 Number Theory

Short Answer Section

The following questions are worth 1 point each. Answer the questions in your *Mathematica* notebook file.

1. What is e^{13} evaluated to 23 significant figures?
2. Factorize $x^{12} + 3\,x^{10} - 23\,x^8 - 51\,x^6 + 94\,x^4 + 120\,x^2$.
3. Convert $\dfrac{e^{2x} - 1}{xe^{2x} + x}$ into an expression involving trig functions.
4. What is the partial fraction decomposition of the rational polynomial.

$$\frac{x^7 + 4\,x^6 + 25\,x^4 - 20\,x^3 + 53\,x^2 - 42\,x + 33}{x^8 - 2\,x^7 + 7\,x^6 - 12\,x^5 + 18\,x^4 - 24\,x^3 + 20\,x^2 - 16\,x + 8}$$

5. What are the first 20 terms of the sequence $\left\{ \dfrac{1}{k(k+1)} \right\}_{k=1}^{\infty}$?
6. Evaluate $\displaystyle\sum_{k=1}^{\infty} \frac{1}{k(k+1)}$.
7. Evaluate $\displaystyle\prod_{k=1}^{\infty} \left(1 - \frac{1}{2x^2} \right)$.
8. What is the 100,000,000th prime number?

Long Answer Section

The following questions are worth 3 points each. Points are given for working. Answer the questions in your *Mathematica* notebook file.

9. Let $a_n = 2n - 1$ and $s_n = n^2$ and define the sequences

$$A := \{a_n\}_{n=1}^{\infty} \quad S := \{s_n\}_{n=1}^{\infty}$$

It should be clear then that $A = \{1, 3, 5, \dots\}$ and $S = \{1, 4, 9, 16, \dots\}$.

a. Calculate the first 20 terms of the sequence

$$\{s_{n+1} - s_n\}_{n=1}^{\infty}$$

   What is $s_{n+1} - s_n$?

b. Calculate the first 20 terms of the sequence

$$\left\{ \sum_{k=1}^{n} a_k \right\}_{n=1}^{\infty}$$

   What is $\sum_{k=1}^{n} a_k$?

10. Recall that $\sum k^{-1}$ diverges. It may be shown that $\sum k^{-(1+\epsilon)}$ converges for any $\epsilon > 0$. For this question we let $\epsilon = \frac{1}{100}$

a. Evaluate the series $\sum_{k=1}^{\infty} 1/k^{\frac{101}{100}}$, and obtain a decimal approximation.

b. Calculate decimal approximations of the partial sums

$$\sum_{k=1}^{N} \frac{1}{k^{101/100}} \text{ for } N = 10, 100, 1000, 10000, 100000$$

   and measure how much time each takes to calculate.

   Notice that this series converges very slowly.

11. Let $\{f_n\}$ be the Fibonacci-like sequence defined by

$$f_n := f_{n-1} + f_{n-2} \quad f_1 = -2, f_2 = 3$$

a. Write a function to calculate the terms of this sequence.

b. What are the first 10 terms of this sequence?

c. What is the largest number in this sequence less than 1,000,000, and what is its index?

12. Let s be the first-order nonlinear recurrence relation defined by

$$s_n = n \, s_{n-1}^2$$

a. Let $s_0 = C$ and calculate the first 5 or so terms of the recurrence.

b. Solve the recurrence.

c. Verify the solution for at least 20 terms of the sequence, and in general if you can.

A.2 Calculus

Short Answer Section

The following questions are worth 1 point each. Answer the questions in your *Mathematica* notebook file.

1. What is the limit of $\dfrac{x + \sin x}{\pi\, x}$ at $x \to \infty$?

2. Find $\lim\limits_{x \to 0^+} \dfrac{-\cosh x}{x}$.

3. What is the derivative of $\dfrac{\cos x}{x}$?

4. What is the slope of the tangent to the curve $y = \dfrac{\cos x}{x}$ at $x = \frac{1}{3}\pi$?

5. Evaluate $\displaystyle\int_0^1 \log x \, dx$.

6. Find a function whose derivative is $\tanh x$.

7. Find the first partial derivatives of $z = x^2 - y^2$.

8. How many critical points does $z = x^2 - y^2$ have, and what kind of critical points are they?

Long Answer Section

The following questions are worth 3 points each. Points are given for working. Answer the questions in your *Mathematica* notebook file.

9. A length of wire 10 meters long is cut in two. One of the pieces is bent into a square, the other into an equilateral triangle. Let x be the length of wire that is bent into the square (meaning that $10 - x$ is the length of wire bent into the triangle). Let A_s be the area of the square and let A_t be the area of the triangle.

    a. Define A to be the formula for the total area of the two shapes (i.e., $A :=$ $A_s + A_t$). Plot A.
    b. How much wire should be used for the square to maximize the total area?
    c. How much wire should be used for the square to minimize the total area?

10. The Airy functions $\mathrm{Ai}(z), \mathrm{Bi}(z)$ are the two independent solutions to the differential equation

$$y'' - z\,y = 0 \tag{A.1}$$

    a. Solve the differential equation (A.1), and verify the solution.
    b. Plot the Airy functions together on the same axes. Make sure to show good detail of what the functions are doing.
    c. Find and plot a third solution, other than $y = \mathrm{Ai}(z)$ and $y = \mathrm{Bi}(z)$, to equation (A.1).

11. Use solids of revolution to verify the following volumes.

    a. The volume of a sphere with radius r ($4/3\pi r^3$)
    b. The volume of a cone with height h and radius r ($1/3\pi r^2 h$)

12. Consider the surface $z = \sin(x)\cos(y)$.

    a. Plot the surface z.
    b. Find a general formula, or formulae, for the critical points.
    c. Which critical points are maxima, which are minima, and which are saddle points?

A.3 Linear Algebra

The following questions are worth 1 point each. Answer the questions in your *Mathematica* notebook file.

1. Calculate the vector $\pi \cdot (8, 1, 5, 1, 9) + e \cdot (1, 2, 5, 6, 6)$.

2. Calculate the matrix product $\begin{bmatrix} 3 & 1 & 6 \\ 0 & 0 & 7 \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 0 & 0 \\ 5 & 4 \end{bmatrix}$.

3. Calculate the dot product of the vectors $(3, 3, 2, 3, 3, 3)$ and $(3, 2, 3, 3, 2, 1)$.

4. Find the angle between the vectors $(3, 3, 2, 3, 3, 3)$ and $(3, 2, 3, 3, 2, 1)$.

5. Find a vector perpendicular to the vectors $(5, 5, 3)$ and $(5, 5, 5)$.

6. Create the 9×10 matrix M whose entries $m_{i,j} = 17\, i\, j$.

7. Find the elementary matrix that will add k multiplied by row 7 to row 9 of a 10×10 matrix.

8. How many solutions are there to the vector equation $M \cdot x = 0$ where

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

Long Answer Section

The following questions are worth 3 points each. Points are given for working. Answer the questions in your *Mathematica* notebook file.

9. This question refers to the following simultaneous equations.

$$y + 3z = 2$$
$$8x + 2y + 2z = 3$$
$$3x + 3y + 5z = 2$$

   a. Solve the simultaneous equations. How many solutions are there?
   b. Plot the three surfaces in such a way that clearly shows the solution.

10. This question refers to the following three matrices

$$\begin{bmatrix} 1\,1\,7\,2 \\ 2\,2\,0\,9 \\ 7\,7\,2\,5 \\ 0\,0\,1\,5 \end{bmatrix}, \begin{bmatrix} 0\,4\,1\,4 \\ 0\,2\,1\,1 \\ 3\,3\,4\,4 \\ 6\,4\,9\,5 \end{bmatrix}, \begin{bmatrix} 0\,1\,1\,1 \\ 6\,1\,0\,0 \\ 2\,2\,6\,6 \\ 9\,4\,4\,4 \end{bmatrix}$$

   a. Which of the matrices may be expressed as a product of elementary matrices?
   b. For the matrices that may be expressed as a product of elementary matrices, find the sequence of elementary matrices whose product is that matrix. (Equivalently, you may find the sequence of row operations performed on the identity matrix.)

11. Let A be the matrix below, and let $p, q \in \mathbb{R}$.

$$A := \begin{bmatrix} p & q & 1-p-q \\ 1-p-q & p & q \\ q & 1-p-q & p \end{bmatrix}$$

   a. Create A in *Mathematica* as a function of p and q.
   b. By examining various numerical cases where $p > 0$, $q > 0$ and $1-p-q > 0$, conjecture the behavior of the matrix A^n as $n \to \infty$.

12. This question refers to the following set of matrices

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

   a. Do the matrices form a basis for $M_3(\mathbb{R})$? Justify your answer.
   b. Find the coefficients of a linear combination of these matrices for an arbitrary 3×3 matrix.

References

1. ANTON, H., AND RORRES, C. *Elementary Linear Algebra*, 7th ed. John Wiley and Sons Inc, Brisbane, 1994.
2. BAILEY, D., BORWEIN, J., CALKIN, N., GIRGENSOHN, R., LUKE, R., AND MOLL, V. *Experimental Mathematics in Action*, 1st ed. AK Peters, Wellesley, MA, 2007.
3. BORWEIN, J., AND BAILEY, D. *Mathematics by Experiment: Plausible Reasoning in the 21st Century*, 2nd ed. AK Peters, Wellesley, MA, 2008.
4. BORWEIN, J., BAILEY, D., AND GIRGENSOHN, R. *Experimentation in Mathematics: Computational Paths to Discovery*, 1st ed. AK Peters, Natick, MA, 2004.
5. BORWEIN, J., AND DEVLIN, K. *The Computer as Crucible: An Introduction to Experimental Mathematics*. AK Peters, Wellesley, MA, 2009.
6. GANDER, W., AND HREBK, J. *Solving Problems in Scientific Computing Using Maple and MATLAB*, 4th ed. Springer, New York, 2008.
7. GARVAN, F. *The Maple 5 Primer Rel 4*. Prentice Hall, Englewood Cliffs, NJ, 1997.
8. GARVAN, F. *The Maple Book*. Chapman and Hall/CRC, Boca Raton, FL, 2001.
9. HECK, A. *Introduction to Maple*, 3rd ed. Springer, New York, 2003.
10. KLIMEK, G., AND KLIMEK, M. *Discovering Curves and Surfaces with Maple*. Springer, New York, 1997.
11. ROVENSKI, V. Y. *Geometry of Curves and Surfaces with MAPLE*. Springer, New York, 2000.
12. STEWART, J. *Calculus*, 6th ed. Brooks/Cole, 2008.
13. TROTT, M. *The Mathematica Guidebooks*, 3rd ed. Springer, New York, 2004–2006.
14. WAGON, S. *Mathematica in Action*, 2nd ed. Springer, New York, 1999.

Index

3n + 1 Problem, 74
* (operator), 2
-> (operator), 46
. (operator), 141, 142
/. (operator), 4, 47, 51, 52, 176
// (operator), 13
/; (operator), 28
/@ (operator), 33, 71, 154, 165
:= (operator), 6
; (operator), 3, 20, 27, 28
< (operator), 16, 19
= (operator), 3, 6, 19
== (operator), 19, 39, 52
@ (operator), 13
[[...]] (operator), 9, 38
% (operator), 3, 28, 67, 72, 153
&& (operator), 17, 32, 36, 37
´ (operator), 124
^ (operator), 142
_ (operator), 15
Logical OR (operator), 17, 40, 43

abundant number, 38, 39
AlgebraicNumber (function), 74
amicable numbers, 31, 32, 35–38
amicable pair, 31, 32, 35–38
Animate (function), 196
argument, 6
arithmetic sequence, 70, 72
Array (function), 11, 43, 68, 140
AspectRatio (keyword), 106
Assumptions (keyword), 91
assumptions, specifying, 16–17, 91–92
augmented matrix, 143, 144, 146, 156
Automatic (keyword), 82, 198
AxesLabel (keyword), 197

Bessel equation, 135
    modified, 135
Bessel function, 135
Bessel functions of the first and second kind,
    135
    modified, 135

Block (function), 25, 27, 28, 33, 34, 36, 37, 47,
    50, 54, 55, 63, 71, 85, 90, 145
Blue (keyword), 82, 202
BoxRatios (keyword), 126
Break (function), 64

Cases (function), 27, 28, 31, 32, 37, 38, 46, 47,
    51
characteristic equation, 103, 104
characteristic polynomial, 59, 175, 177, 183,
    192, 193
CharacteristicPolynomial (function), 175
Circle (function), 199
Clairaut's theorem, 125
    failure of, 136
co-ordinates
    cylindrical, 113
    polar, 109
    polar, conversion to Cartesian, 109
    spherical, 114
Coefficient (function), 159
Collatz's conjecture, 74
command separation, 3
command termination, 3
Complement (function), 9, 10, 63
compound expression, 3, 27
computation time, 23, 43
    measuring, 45
constant variable, 24
continued fraction, 53–57, 73
ContinuedFraction (function), 56, 57, 73
ContourPlot (function), 109, 111, 115
ContourPlot3D (function), 115
conversion between $P_n(\mathbb{F})$ and $\mathbb{F}^n$, 158, 160,
    164, 168
Count (function), 69
cylindrical co-ordinates, 113

D (function), 94, 123, 181
deficient number, 38
definition, delayed, 6
definition,delayed, 165
delayed definition, 6, 165
DeleteDuplicates (function), 177