CHAPTER NINETEEN

Epilogue: Facts Are Not Reality

THE LAST (NOT LEAST) IMPORTANT SKILL WHEN WORKING WITH DATA IS TO KEEP IN MIND THAT DATA IS ONLY part of the picture. In particular, when one is working intensely with data oneself, it is all too easy to forget that just about everyone else will have a different perspective.

When the data contradicts appearances, appearances will win. Almost always, at least. Abstract "data" will have little or no credibility when compared with direct, immediate observation. This has been one of my most common experiences. A manager observes a pile of defective items—and no amount of "data" will convince him that avoiding those defects will cost more than the defects themselves. A group of workers spends an enormous amount of effort on some task—and no amount of "data" will convince them that their efforts make no measurable difference to the quality of the product.

If something strongly *appears* to be one way, then it will be very, very difficult to challenge that appearance based on some abstract analysis—no matter how "hard" your facts may be.

And it can get ugly. If your case is watertight, so that your analysis cannot be refuted, then you may next find that your *personal* credibility or integrity is being challenged.

Never underestimate the persuasive power of appearance.

Data-driven decision making is a contradiction in terms. Making a decision means that someone must stick his or her neck out and *decide*. If we wait until the situation is clear or let "the data" dictate what we do, then there is no longer any decision involved. This also means that if things don't turn out well, then nobody will accept the blame (or the responsibility) for the outcome: after all, we did what "the data" told us to do.

It is a fine line. Gut-level decisions can be annoyingly random (this way today, that way tomorrow). They can also lead to a lack of accountability: "It was my decision to do X that led to Y!"—without a confirming look at some data, who can say?

Studying data can help us understand the situation in more detail and therefore make better-informed decisions. On the other hand, data can be misleading in subtle ways. For instance, by focusing on "data" it is easy to overlook aspects that are important but for which no data is available (including but not limited to "soft factors"). Also, keep in mind that data is always *backward* looking: there is no data available to evaluate any truly novel idea!

Looking at data can help illuminate the situation and thereby help us make better decisions. But it should not be used to absolve everyone from taking individual responsibility.

Sometimes the only reason you need is that it is the right thing to do. Some organizations feel as if you would not put out a fire in the mail room, unless you first ran a controlled experiment and developed a business case for the various alternatives. Such an environment can become frustrating and stifling; if the same approach is being applied to human factors such as creature comforts (better chairs, larger monitors) or customer service ("sales don't dip proportionally if we lower the quality of our product"), then it can start to feel toxic pretty quickly.

Don't let "data" get in the way of ethical decisions.

The most important things in life can't be measured. It is a fallacy to believe that, just because something can't be measured, it doesn't matter or doesn't even exist. And a pretty tragic fallacy at that.

APPENDIX A

Programming Environments for Scientific Computation and Data Analysis

MOST DATA ANALYSIS INVOLVES A GOOD DEAL OF DATA MANIPULATION AND NUMERICAL COMPUTATION. OF course, we use computers for these tasks, hence we also need appropriate software.

This appendix is intended to give a brief survey of several popular software systems suitable for the kind of data analysis discussed in the rest of the book. I am mostly interested in open source software, although I also mention some of the most important commercial players.

The emphasis here is on *programming environments* for scientific applications (*i.e.*, libraries or packages intended for general data manipulation and computation) because being able to operate with data easily and conveniently is a fundamental capability for all data analysis efforts. On the other hand, I do not include programs intended exclusively for graphing data: not because visualization is not important (it is), but because the choice of plotting or visualization software is less fundamental.

Software Tools

In many ways, our choice of a data manipulation environment determines what problems we can solve; it certainly determines which problems we consider to be "easy" problems. For data analysis, the hard problem that we should be grappling with is always the data and what it is trying to tell us—the mechanics of handling it should be sufficiently convenient that we don't even think about them.

Properties I look for in a tool or programming environment include:

- Low overhead or ceremony; it must be easy to get started on a new investigation.
- Facilitates iterative, interactive use.

- No arbitrary limitations (within reasonable limits).
- Scriptable—not strictly required but often nice to have.
- Stable, correct, mature; free of random defects and other annoying distractions.

Most of these items are probably not controversial. Given the investigative nature of most data analysis, the ability to support iterative, interactive use is a requirement. Scriptability and the absence of arbitrary limitations are both huge enablers. I have been in situations where the ability to generate and compare hundreds of graphs revealed obvious similarities and differences that had never been noticed before—not least because everyone else was using tools (mostly Excel) that allowed graphs to be created only one at a time. (Excel is notorious for unnecessarily limiting what can be done, and so is SQL. Putting even minimal programming abilities on top of SQL greatly expands the range of problems that can be tackled.)

In addition to these rather obvious requirements, I want to emphasize two properties that may appear less important, but are, in fact, essential for successful data analysis. First, it is very important that the tool or environment itself does not impose much overhead or "ceremony": we will be hesitant to investigate an ad hoc idea if our programming environment is awkward to use or time-consuming to start. Second, the tool must be stable and correct. Random defects that we could "work around" if we used it as a component in a larger software project are unacceptable when we use the tool by itself.

In short: whatever we use for data manipulation must not get in our way! (I consider this more important than how "sophisticated" the tool or environment might be: a dumb tool that works is better than a cutting-edge solution that does not deliver—a point that is occasionally a little bit forgotten.)

Before leaving this section, let me remind you that it is not only the size of the toolbox that matters but also our mastery of the various elements within it. Only tools we know well enough that using them feels effortless truly leverage our abilities. Balancing these opposing trends (breadth of tool selection and depth of mastery) is a constant challenge. When in doubt, I recommend you opt for depth—superficiality does not pay.

Scientific Software Is Different

It is important to realize that scientific software (for a sufficiently wide definition of "scientific") faces some unusual challenges. First of all, scientific software is *hard*. Writing high-quality scientific programs is difficult and requires rather rare and specialized skills. (We'll come back to this later.) Second, the market for scientific software is *small*, which makes it correspondingly harder for any one program or vendor to gain critical mass.

Both of these issues affect all players equally, but a third problem poses a particular challenge for open source offerings: many users of scientific software are transients. Graduate students graduate, moving on from their projects and often leaving the research

environment entirely. As a result, "abandonware" is common among open source scientific software projects. (And not just there—the long-term viability of commercial offerings is also far from assured.)

Before investing significant time and effort into mastering any one tool, it is therefore necessary to evaluate it with regard to two questions:

- Is the project of sufficiently high *quality*?
- Does the project have strong enough *momentum* and *support*?

A Catalog of Scientific Software

There are currently three main contenders for interactive, numeric programming available: Matlab (and its open source clone, Octave), R (and its commercial predecessor, S/S-Plus), and the NumPy/SciPy set of libraries for Python. Fundamentally, all three are vector and matrix packages: they treat vectors and matrices as atomic data types and allow mathematical functions to operate on them directly (addition, multiplication, application of a function to all elements in a vector or matrix). Besides this basic functionality, all three offer various other mathematical operations, such as special functions, support for function minimization, or numerical integration and nonlinear equation solving. It is important to keep in mind that all three are packages for *numerical* computations that operate with floating-point numbers. None of these three packages handles *symbolic* computations, such as the expansion of a function into its Taylor series. For this you need a symbolic math package, such as Mathematica or Maple (both commercial) or Maxima, Sage, or Axiom (all three open source). (Matlab has recently acquired the ability to perform symbolic operations as well.)

Matlab

Matlab has been around since the mid-1980s; it has a very large user base, mostly in the engineering professions but also in pure mathematics and in the machine-learning community. Rather than do all the heavy lifting itself, Matlab was conceived as a user-friendly frontend to existing high-performance numerical linear algebra libraries (LINPACK and EISPACK, which have been replaced by LAPACK). Matlab was one of the first widely used languages to treat complex data structures (such as vectors and matrices) as atomic data types, allowing the programmer to work with them as if they were scalar variables and without the need for explicit looping. (In this day and age, when object-oriented programming and operator overloading are commonly used and entirely mainstream, it is hard to imagine how revolutionary this concept seemed when it was first developed.[*]) In 2008, The MathWorks (the company that develops Matlab) acquired the

[*] I remember how blown away I personally was when I first read about such features in the programming language APL in the mid-1980s!

rights to the symbolic math package MuPAD and incorporated it into subsequent Matlab releases.

Matlab was mainly designed to be used interactively, and its programming model has serious deficiencies for larger programming projects. (There are problems with abstraction and encapsulation as well as memory management issues.) It is a commercial product but quite reasonably priced.

Matlab places particular emphasis on the quality of its numerical and floating-point algorithms and implementations.

There is an open source clone of Matlab called Octave. Octave (*http://www.gnu.org/software/octave/*) strives to be fully compatible; however, there are reports of difficulties when porting programs back and forth.

R

R is the open source clone of the S/S-Plus statistical package originally developed at Bell Labs. R (*http://www.r-project.org*) has a very large user base, mostly in the academic statistics community and a healthy tradition of user-contributed packages. The *Comprehensive R Archive Network* (CRAN) is a large central repository of user-contributed modules.

When first conceived, S was revolutionary in providing an integrated system for data analysis, including capabilities that we today associate with scripting languages (built-in support for strings, hash maps, easy file manipulations, and so on), together with extensive graphics functionality—and all that in an interactive environment! On the other hand, S was not conceived as a general-purpose programming language but is strongly geared toward statistical applications. Its programming model is quite different from current mainstream languages, which can make it surprisingly difficult for someone with a strong programming background to switch to S (or R). Finally, its primarily academic outlook makes for a sometimes awkward fit within a commercial enterprise environment.

The strongest feature of R is the large number of built-in (or user-contributed) functions for primarily *statistical* calculations. In contrast to Matlab, R is not intended as a general numerical workbench (although it can, with some limitations, be used for that purpose). Moreover—and perhaps contrary to expectations—it is not intended as a general-purpose data manipulation language, although it can serve as scripting language for text and file manipulations and similar tasks.

A serious problem when working with R is its dated programming model. It relies strongly on implicit behavior and "reasonable defaults," which leads to particularly opaque programs. Neither the language nor the libraries provide strong support for organizing information into larger structures, making it uncommonly difficult to locate pertinent information. Although it is easy to pick up isolated "tricks," it is notoriously difficult to develop a comprehensive understanding of the whole environment.

Like Matlab, R is here to stay. It has proven its worth (for 30 years!); it is mature; and it has a strong, high-caliber, and vocal user base. Unlike Matlab, it is free and open source, making it easy to get started.

Python

Python has become the scripting language of choice for scientists and scientific applications, especially in the machine-learning field and in the biological and social sciences. (Hard-core, large-scale numerical applications in physics and related fields continue to be done in C/C++ or even—*horresco referens*—in Fortran.)

The barrier to programming in Python is low, which makes it easy to start new projects. This is somewhat of a mixed blessing: on the one hand, there is an abundance of exciting Python projects out there; on the other hand, they seem to be particularly prone to the "abandonware" problem mentioned before. Also, scientists are not programmers, and it often shows (especially with regard to long-term, architectural vision and the cultivation of a strong and committed community).

In addition to a large number of smaller and more specialized projects, there have been five major attempts to provide a *comprehensive* Python library for scientific applications. It can be confusing to understand how they relate to each other, so they are summarized here:*

Numeric
: This is the original Python module for the manipulation of numeric arrays, initiated in 1995 at MIT. Superceded by NumPy.

Numarray
: An alternative implementation from the Space Telescope Science Institute (2001). Considered obsolete, replaced by NumPy.

NumPy
: The NumPy project was begun in 2005 to provide a unified framework for numerical matrix calculations. NumPy builds on (and supercedes) Numeric, and it includes the additional functionality developed by numarray.

SciPy
: Started in 2001, the SciPy project evolved out of an effort to combine several previously separate libraries for scientific computing. Builds on and includes NumPy.

ScientificPython
: An earlier (started in 1997) general-purpose library for scientific applications. In contrast to SciPy, this library tries to stay with "pure Python" implementations for better portability.

*For more information on the history and interrelations of these libraries, check out the first chapter in Travis B. Oliphant's "Guide to NumPy," which can be found on the Web.

Today, the NumPy/SciPy project has established itself as the clear winner among general-purpose libraries for scientific applications in Python, and we will take a closer look at it shortly.

A strong point in favor of Python is the convenient support it has for relatively fancy and animated graphics. The matplotlib library is the most commonly used Python library for generating standard plots, and it has a particularly close relationship with NumPy/SciPy. Besides matplotlib there are Chaco and Mayavi (for two- and three-dimensional graphics, respectively) and libraries such as PyGame and Pyglet (for animated and interactive graphics)—and, of course, many more.

Uncertainties associated with the future and adoption of Python3 affect all Python projects, but they are particularly critical for many of the scientific and graphics libraries just mentioned: to achieve higher performance, these libraries usually rely heavily on C bindings, which do not port easily to Python3. Coupled with the issue of "abandonware" discussed earlier, this poses a particular challenge for all scientific libraries based on Python at this time.

NumPy/SciPy

The NumPy/SciPy project (*http://www.scipy.org*) has become the dominant player in scientific programming for Python. NumPy provides efficient vector and matrix operations; SciPy consists of a set of higher-level functions built on top of NumPy. Together with the matplotlib graphing library and the IPython interactive shell, NumPy/SciPy provides functionality resembling Matlab. NumPy/SciPy is open source (BSD-style license) and has a large user community; it is supported and distributed by a commercial company (Enthought).

NumPy is intended to contain low-level routines for handling vectors and matrices, and SciPy is meant to contain all higher-level functionality. However, some additional functions are included in NumPy for backward compatibility, and all NumPy functions are aliased into the SciPy namespace for convenience. As a result, the distinction between NumPy and SciPy is not very clear in practice.

NumPy/SciPy can be a lot of fun. It contains a wide selection of features and is very easy to get started with. Creating graphical output is simple. Since NumPy/SciPy is built on Python, it is trivial to integrate it into other software projects. Moreover, it does not require you to learn (yet another) restricted, special-purpose language: everything is accessible from a modern, widely used scripting language.

On the other hand, NumPy/SciPy has its own share of problems. The project has a tendency to emphasize quantity over quality: the number of features is very large, but the design appears overly complicated and is often awkward to use. Edge and error cases are not always handled properly. On the scientific level, NumPy/SciPy feels amateurish. The choice of algorithms appears to reflect some well-known textbooks more than deep, practical knowledge arising from real experience.

What worries me most is that the project does not seem to be managed very well: although it has been around for nearly 10 years and has a large and active user base, it has apparently not been able to achieve and maintain a consistent level of reliability and maturity throughout. Features seem to be added haphazardly, without any long-term vision or discernible direction. Despite occasional efforts in this regard, the documentation remains patchy.

NumPy/SciPy is interesting because, among scientific and numeric projects, it probably has the lowest barrier to entry and is flexible and versatile. That makes it a convenient environment for getting started and for casual use. However, because of the overall quality issues, I would not want to rely on it for "serious" production work at this point.

What About Java?

Java is not a strong player when it comes to heavily *numerical* computations—so much so that a Java Numerics Working Group ceased operations years ago (around the year 2002) for lack of interest.

Nevertheless, a lot of production-quality machine-learning programming is done in Java, where its relatively convenient string handling (compared to C) and its widespread use for *enterprise* programming come into play. One will have to see whether these applications will over time lead to the development of high-quality numerical libraries as well.

If you want a comfortable programming environment for large (possibly distributed) systems that's relatively fast, then Java is a reasonable choice. However, Java programming has become very heavy-weight (with tools to manage your frameworks, and so on), which does not encourage ad hoc, exploratory programming. Groovy carries less programming overhead but is slow. A last issue concerns Java's traditionally weak capabilities for interactive graphics and user interfaces, especially on Linux.

Java is very strong in regard to Big Data; in particular, Hadoop—the most popular open source map/reduce implementation—is written in Java. Java is also popular for text processing and searching.

A relatively new project is Incanter (*http://incanter.org/*), which uses Clojure (a Lisp dialect running on top of the Java virtual machine) to develop an "R-like statistical computing and graphics environment." Incanter is an interesting project, but I don't feel that it has stood the test of time yet, and one will have to see how it will position itself with respect to R.

Other Players

The preceding list of programs and packages is, of course, far from complete. Among the other players, I shall briefly mention three.

SAS SAS is a classical statistics packages with strongly established uses in credit scoring and medical trials. SAS was originally developed for OS/360 mainframes, and it shows. Its command language has a distinct 1960s feel, and the whole development cycle is strongly batch oriented (neither interactive nor exploratory). SAS works best when well-defined procedures need to be repeated often and on large data sets. A unique feature of SAS is that it works well with data sets that are too large to fit into memory and therefore need to be processed on disk.

SAS, like the mainframes it used to run on, is very expensive and requires specially trained operators—it is not for the casual user. (It is not exactly fun, either. The experience has been described as comparable to "scraping down the wallpaper with your fingernails.")

SciLab SciLab is an open source project similar to Matlab. It was created by the French research institute INRIA.

GSL The GSL (Gnu Scientific Library) is a C library for classical numerical analysis: special functions, linear algebra, nonlinear equations, differential equations, the lot. The GSL was designed and implemented by a relatively small team of developers, who clearly knew what they were doing—beyond the standard textbook treatment. (This is evident from some design choices that specifically address ugly but important real-world needs.)

The API is wonderfully clear and consistent, the implementations are of high quality, and even the documentation is complete and finished. I find the GSL thoroughly enjoyable to use. (If you learned numerical analysis from *Numerical Recipes*,* this is the software that should have shipped with the book—but didn't.)

The only problem with the GSL is that it is written in C. You need to be comfortable with C programming, including memory management and function pointers, if you want to use it. Bindings to scripting languages exist, but they are not part of the core project and may not be as complete or mature as the GSL itself.

Recommendations

So, which to pick? No clear winner emerges, and every single program or environment has significant (not just superficial) drawbacks. However, here are some qualified recommendations:

- Matlab is the 800-pound gorilla of scientific software. As a commercially developed product, it also has a certain amount of "polish" that many open source alternatives

* *Numerical Recipes 3rd Edition: The Art of Scientific Computing.* William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. Cambridge University Press. 2007.

lack. If you don't have a preferred programming environment yet, *and* if you can afford it (or can make your employer pay for it), then Matlab is probably the most comprehensive, most mature, and best supported all-purpose tool. Octave is a cheap way to get started and "try before you buy."

- If you work with statisticians or have otherwise a need for formal statistical methods (tests, models), then R is a serious contender. It can also stand in as a scripting language for data manipulation if you don't already have a favorite one yet. Since it is open source software, its financial cost to you is zero, but be prepared for a significant investment of time and effort before you start feeling comfortable and proficient.
- NumPy/SciPy is particularly easy to get started with and can be a lot of fun for casual use. However, you may want to evaluate carefully whether it will meet your needs in the long run if you are planning to use it for a larger or more demanding project.
- NumPy/SciPy, together with some of its associated graphics packages, is also of interest if you have a need for fancier, possibly interactive, graphics.
- If you have a need for serious numerical analysis *and* you know C well, then the GSL is a mature, high-quality library.

I am well aware that this list of options does not cover all possibilities that may occur in practice!

Writing Your Own

Given the fragmented tool situation, it may be tempting to write your own. There is nothing wrong with that: it can be very effective to write a piece of software specifically for *your* particular problem and application domain. It is much harder to write general-purpose scientific software.

Just how much harder is generally underappreciated. When P. J. Plauger worked on his reference implementation of the standard C library,* he found that he "spent about as much time writing and debugging the functions declared in `<math.h>` as [he] did all the rest of this library combined"! Plauger then went on to state his design goals for his implementation of those functions.

This should startle you: *design goals*? Why should a reference implementation need any design goals beyond faithfully and correctly representing the standard?

The reason is that scientific and numerical routines can fail in more ways than most people expect. For such routines, correctness is not so much a binary property, as a floating-point value itself. Numerical routines have more complicated contracts than `strlen(char *)`.

* *The Standard C Library.* P. J. Plauger. Prentice Hall. 1992.

My prime example for this kind of problem is the sine function. What could possibly go wrong with it? It is analytic everywhere, strictly bounded by $[-1, 1]$, perfectly smooth, and with no weird behavior anywhere. Nonetheless, it is impossible to evaluate the sine accurately for sufficiently large values of x. The reason is that the sine sweeps out its entire range of values when x changes by as little as 2π. Today's floating-point values carry about 16 digits of precision. Once x has become so large that all of these digits are required to represent the value of x to the left of the decimal point, we are no longer able to resolve the location of x within the interval of length 2π with sufficient precision to be meaningful—hence the "value" returned by $\sin(x)$ is basically random. In practice, the quality of the results starts to degrade long before we reach this extreme regime. (More accurately the problem lies not so much in the implementation of the sine but in the inability to express its input values with the precision required for obtaining a meaningful result. This makes no difference for the present argument.)

There are two points to take away here. First, note how "correctness" is a relative quality that can degrade smoothly depending on circumstances (*i.e.*, the inputs). Second, you should register the sense of surprise that a function, which in mathematical theory is perfectly harmless, can turn nasty in the harsh reality of a computer program!

Similar examples can be found all over and are not limited to function evaluations. In particular for iterative algorithms (and almost all numerical algorithms are iterative), one needs to monitor and confirm that all intermediate values are uncorrupted—even in cases where the final result is perfectly reasonable. (This warning applies to many matrix operations, for instance.)

The punch line here is that although it is often not hard to produce an implementation that works well for a limited set of input values and in a narrow application domain, it is much more difficult to write routines that work equally well for all possible arguments. It takes a lot of experience to anticipate all possible applications and provide built-in diagnostics for likely failure modes. If at all possible, leave this work to specialists!

Further Reading

Matlab

- *Numerical Computing with MATLAB.* Cleve B. Moler. Revised reprint, SIAM. 2008.
  The literature on Matlab is vast. I mention this title because its author is Cleve Moler, the guy who started it all.

R

- *A Beginner's Guide to R.* Alain F. Zuur, Elena N. Ieno, and Erik H. W. G. Meesters. Springer. 2009.
  Probably the most elementary introduction into the mechanics of R. A useful book to get started, but it won't carry you very far. Obviously very hastily produced.

- *R in a Nutshell.* Joseph Adler. O'Reilly. 2009.
  This is the first book on R that is organized by the *task* that you want to perform. This makes it an invaluable resource in those situations where you know exactly what you want to do but can't find the appropriate commands that will tell R how to do it. The first two thirds of the book address data manipulation, programming, and graphics in general; the remainder is about statistical methods.
- *Using R for Introductory Statistics.* John Verzani. Chapman & Hall/CRC. 2004.
  This is probably my favorite introductory text on how to perform basic statistical analysis using R.

NumPy/SciPy

There is no comprehensive introduction to NumPy/SciPy currently available that takes a user's perspective. (The "Guide to NumPy" by Travis Oliphant, which can be found on the NumPy website, is too concerned with implementation issues.) Some useful bits, together with an introduction to Python and some other libraries, can be found in either of the following two books.

- *Python Scripting for Computational Science.* Hans Petter Langtangen. 3rd ed., Springer. 2009.
- *Beginning Python Visualization: Crafting Visual Transformation Scripts.* Shai Vaingast. Apress. 2009.

APPENDIX B

Results from Calculus

IN THIS APPENDIX, WE REVIEW SOME OF THE RESULTS FROM CALCULUS THAT ARE EITHER NEEDED EXPLICITLY IN the main part of the book or are conceptually sufficiently important when doing data analysis and mathematical modeling that you should at least be aware that they *exist*.

Obviously, this appendix cannot replace a class (or two) in beginning and intermediate calculus, and this is also not the intent. Instead, this appendix should serve as a reminder of things that you probably know already. More importantly, the results are presented here in a slightly different context than usual. Calculus is generally taught with an eye toward the theoretical development—it has to be, because the intent is to teach the entire body of knowledge of calculus and therefore the theoretical development is most important. However, for applications you need a different sort of tricks (based on the same fundamental techniques, of course), and it generally takes *years* of experience to make out the tricks from the theory. This appendix assumes that you have seen the theory at least once, so I am just reminding you of it, but I want to emphasize those elementary techniques that are most useful in applications of the kind explained in this book.

This appendix is also intended as somewhat of a teaser: I have included some results that are particularly interesting, noteworthy, or fascinating as an invitation for further study.

The structure of this appendix is as follows:

1. To get a head start, we first look at some common functions and their graphs.
2. Then we discuss the core concepts of calculus proper: derivative, integral, limit.
3. Next I mention a few practical tricks and techniques that are frequently useful.

4. Near the end, there is a section on notation and *very* basic concepts. *If you start feeling truly confused, check here!* (I did not want to start with that section because I'm assuming that most readers know this material already.)
5. I conclude with some pointers for further study.

A note for the mathematically fussy: this appendix quite intentionally eschews much mathematical sophistication. I know that many of the statements can be made either more general or more precise. But the way they are worded here is sufficient for my purpose, and I want to avoid the obscurity that is the by-product of presenting mathematical statements in their most general form.

Common Functions

Functions are mappings, which map a real number into another real number: $f : \mathbb{R} \mapsto \mathbb{R}$. This mapping is always unique: every input value x is mapped to exactly one result value $f(x)$. (The converse is not true: many input values may be mapped to the same result. For example, the mapping $f(x) = 0$, which maps *all* values to zero, is a valid function.)

More complicated functions are often built up as combinations of simpler functions. The most important simple functions are powers, polynomials and rational functions, and trigonometric and exponential functions.

Powers

The simplest nontrivial function is the *linear* function:

$$f(x) = ax$$

The constant factor a is the *slope*: if x increases by 1, then $f(x)$ increases by a. Figure B-1 shows linear functions with different slopes.

The next set of elementary functions are the simple powers:

$$f(x) = x^k$$

The power k can be greater or smaller than 1. The exponent can be positive or negative, and it can be an integer or a fraction. Figure B-2 shows graphs of some functions with positive integer powers, and Figure B-3 shows functions with fractional powers.

Simple powers have some important properties:

- All simple powers go through the two points $(0, 0)$ and $(1, 1)$.
- The linear function $f(x) = x$ is a simple power with $k = 1$.
- The square-root function $f(x) = \sqrt{x}$ is a simple power with $k = 1/2$.
- Integer powers ($k = 1, 2, 3, \ldots$) can be evaluated for negative x, but for fractional powers we have to be more careful.

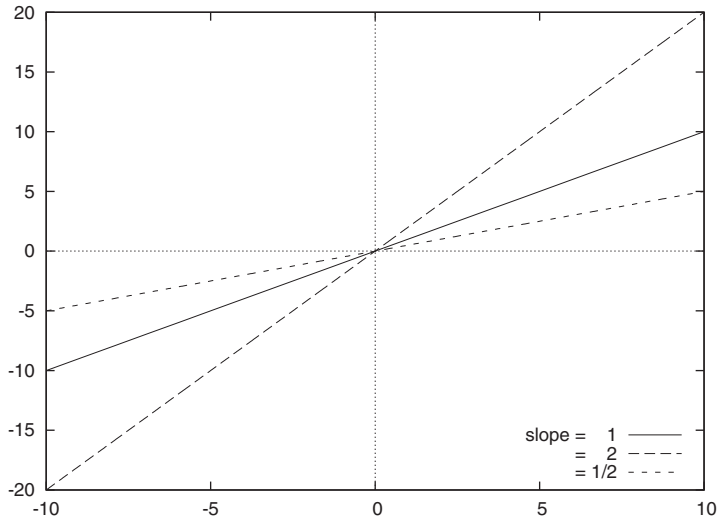

FIGURE B-1. The linear function $y = ax$.

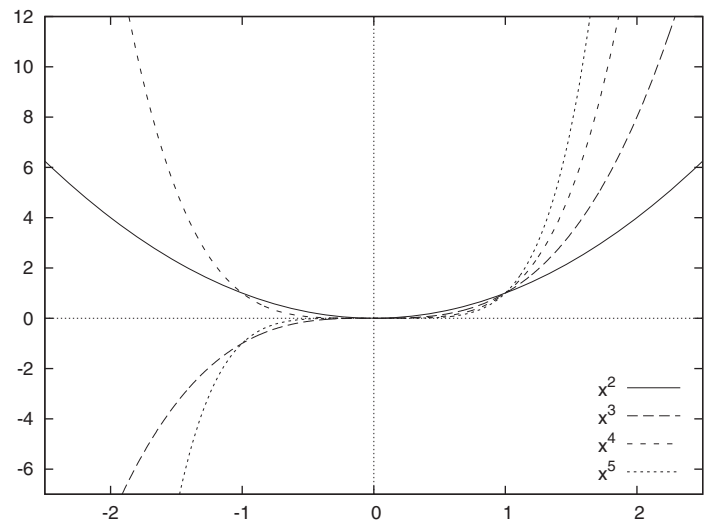

FIGURE B-2. Simple powers: $y = ax^k$.

Powers obey the following laws:

$$x^n x^m = x^{n+m}$$

$$x^n x^{-m} = \frac{x^n}{x^m}$$

$$x^0 = 1$$

$$x^1 = x$$

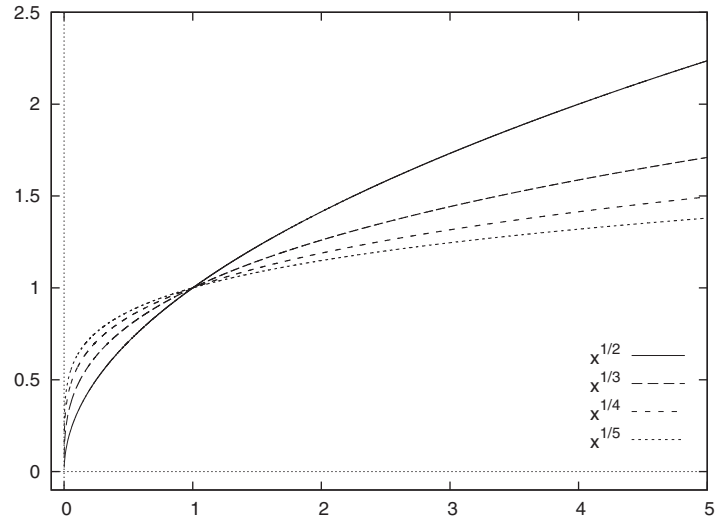

FIGURE B-3. Fractional powers: $y = a^{p/q}$.

If the exponent is negative, it turns the expression into a *fraction*:

$$x^{-n} = \frac{1}{x^n}$$

When dealing with fractions, we must always remember that the denominator must not become zero. As the denominator of a fraction approaches zero, the value of the overall expression goes to infinity. We say: the expression *diverges* and the function has a *singularity* at the position where the denominator vanishes. Figure B-4 shows graphs of functions with negative powers. Note the divergence for $x = 0$.

Polynomials and Rational Functions

Polynomials are sums of integer powers together with constant coefficients:

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0$$

Polynomials are nice because they are extremely easy to handle mathematically (after all, they are just sums of simple integer powers). Yet, more complicated functions can be approximated very well using polynomials. Polynomials therefore play an important role as approximations of more complicated functions.

All polynomials exhibit some "wiggles" and eventually diverge as x goes to plus or minus infinity (see Figure B-5). The highest power occurring in a polynomial is known as that *degree* of the polynomial.

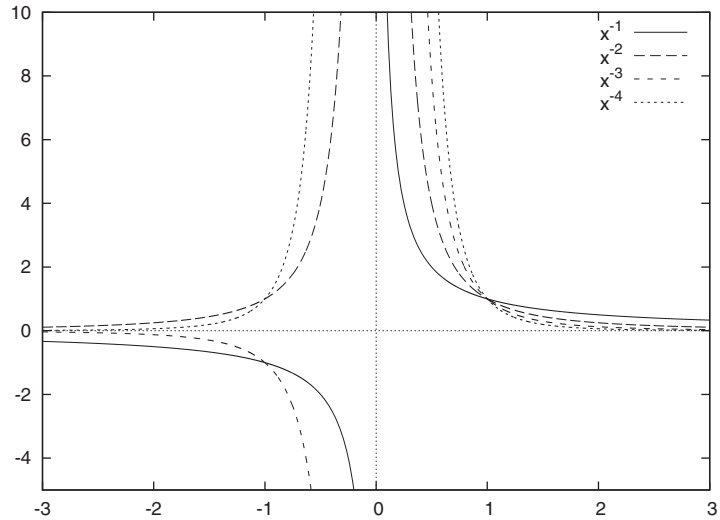

FIGURE B-4. Negative powers: $y = ax^{-k} = a/x^k$.

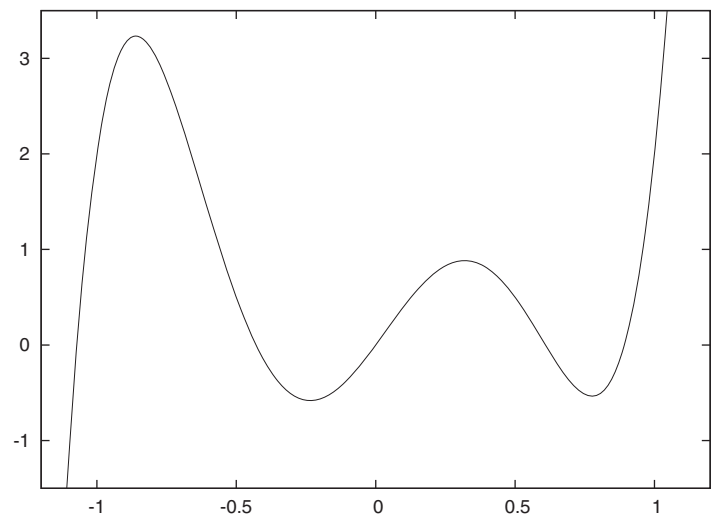

FIGURE B-5. A polynomial: $y = 16x^5 - 20x^3 + 2x^2 + 4x$.

Rational functions are fractions that have polynomials in both the numerator and the denominator:

$$r(x) = \frac{p(x)}{q(x)} = \frac{a_n x^n + a_{n-1}x^{n-1} + \cdots + a_2 x^2 + a_1 x + a_0}{b_m x^m + b_{m-1}x^{m-1} + \cdots + b_2 x^2 + b_1 x + b_0}$$

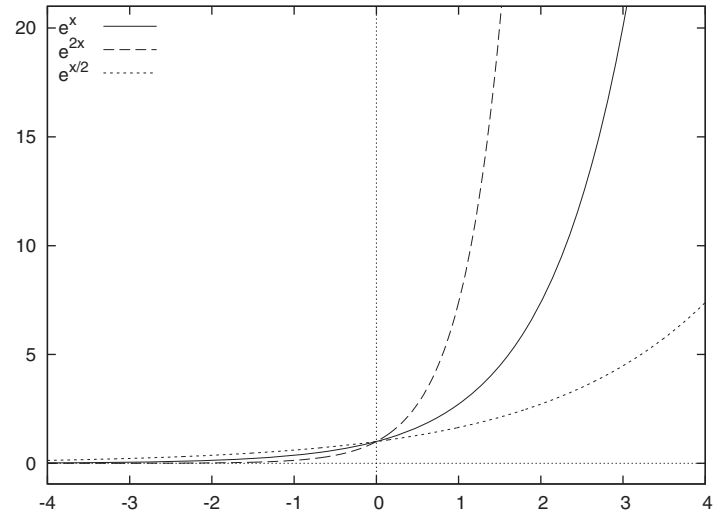

FIGURE B-6. The exponential function $y = e^x$.

Although they may appear equally harmless, rational functions are entirely more complicated beasts than polynomials. Whenever the denominator becomes zero, they blow up. The behavior as x approaches infinity depends on the relative size of the largest powers in numerator and denominator, respectively. Rational functions are *not* simple functions.

Exponential Function and Logarithm

Some functions cannot be expressed as polynomials (or as fraction of polynomials) of finite degree. Such functions are known as *transcendental functions*. For our purposes, the most important ones are the exponential function $f(x) = e^x$ (where $e = 2.718281\ldots$ is Euler's number) and its inverse, the logarithm.

A graph of the exponential function is shown in Figure B-6. For positive argument the exponential function grows *very* quickly, and for negative argument it decays equally quickly. The exponential function plays a central role in growth and decay processes.

Some properties of the exponential function follow from the rules for powers:

$$e^x\, e^y = e^{x+y}$$
$$e^{-x} = \frac{1}{e^x}$$

The logarithm is the inverse of the exponential function; in other words:

$$y = e^x \quad \Longleftrightarrow \quad \log y = x$$
$$e^{\log(x)} = x \quad \text{and} \quad \log\left(e^x\right) = x$$

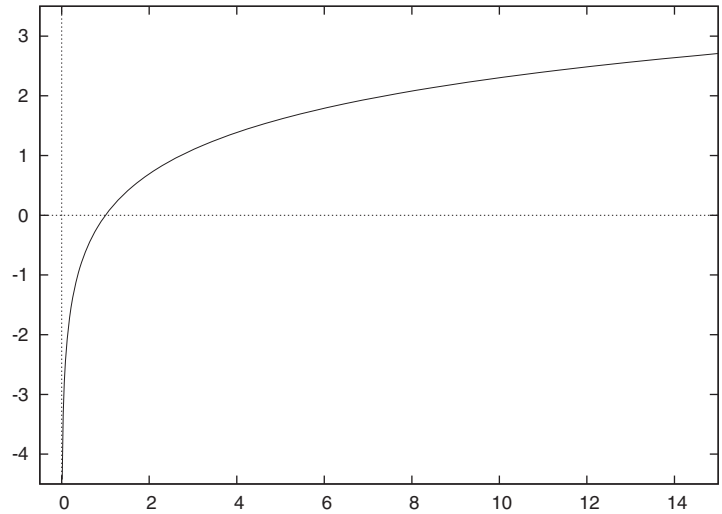

FIGURE B-7. The natural logarithm: $y = \log(x)$.

A plot of the logarithm is shown in Figure B-7. The logarithm is defined only for strictly positive values of x, and it tends to negative infinity as x approaches zero. In the opposite direction, as x becomes large the logarithm grows without bounds, but it grows almost unbelievably slowly. For $x = 2$, we have $\log 2 = 0.69\ldots$ and for $x = 10$ we find $\log 10 = 2.30\ldots$, but for $x = 1{,}000$ and $x = 10^6$ we have only $\log 1000 = 6.91\ldots$ and $\log 10^6 = 13.81\ldots$, respectively. Yet the logarithm does not have an upper bound: it keeps on growing but at an ever-decreasing rate of growth.

The logarithm has a number of basic properties:

$$\log(1) = 0$$
$$\log(x\ y) = \log x + \log y$$
$$\log(x^k) = k\ \log x$$

As you can see, logarithms turn products into sums and powers into products. In other words, logarithms "simplify" expressions. This property was (and is!) used in numerical calculations: instead of multiplying two numbers (which is complicated), you add their logarithms (which is easy—provided you have a logarithm table or a slide rule) and then exponentiate the result. This calculational scheme is still relevant today, but not for the kinds of simple products that previous generations performed using slide rules. Instead, logarithmic multiplication can be necessary when dealing with products that would generate intermediate over- or underflows even though the final result may be of reasonable size. In particular, certain kinds of combinatorial and probabilistic problems require finding the maximum of expressions such as $p^n(1-p)^k$, where $p < 1$ is a probability and n and k may be large numbers. Brute-force evaluation will underflow

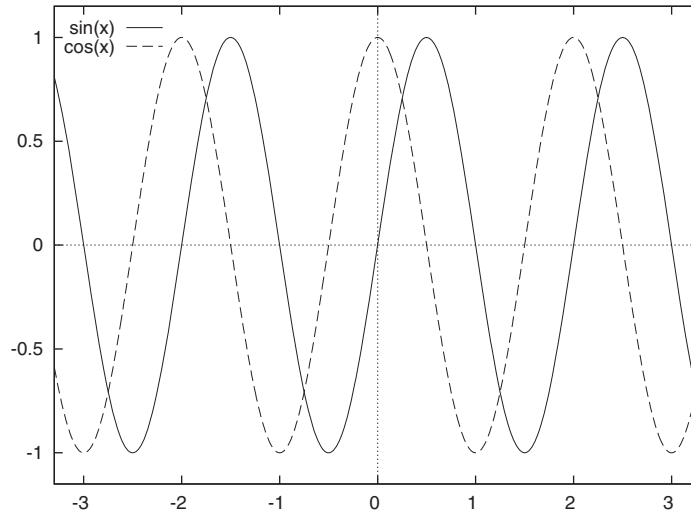

FIGURE B-8. The trigonometric functions $\sin(x)$ *and* $\cos(x)$.

even for modest values of the exponents, but taking logarithms first will result in a numerically harmless expression.

Trigonometric Functions

The trigonometric functions describe oscillations of all kinds and thus play a central role in sciences and engineering. Like the exponential function, they are transcendental functions, meaning they cannot be written down as a polynomial of finite degree.

Figure B-8 shows graphs of the two most important trigonometric functions: $\sin(x)$ and $\cos(x)$. The cosine is equal to the sine but is shifted by $\pi/2$ (90 degrees) to the left. We can see that both functions are *periodic*: they repeat themselves *exactly* after a period of length 2π. In other words, $\sin(x + 2\pi) = \sin(x)$ and $\cos(x + 2\pi) = \cos(x)$.

The length of the period is 2π, which you may recall is the *circumference of a circle with radius equal to* 1. This should make sense, because $\sin(x)$ and $\cos(x)$ repeat themselves after advancing by 2π and so does the circle: if you go around the circle once, you are back to where you started. This similarity between the trigonometric functions and the geometry of the circle is no accident, but this is not the place to explore it.

Besides their periodicity, the trigonometric functions obey a number of rules and properties ("trig identities"), only one of which is important enough to mention here:

$$\sin^2 x + \cos^2 x = 1 \quad \text{for all } x$$

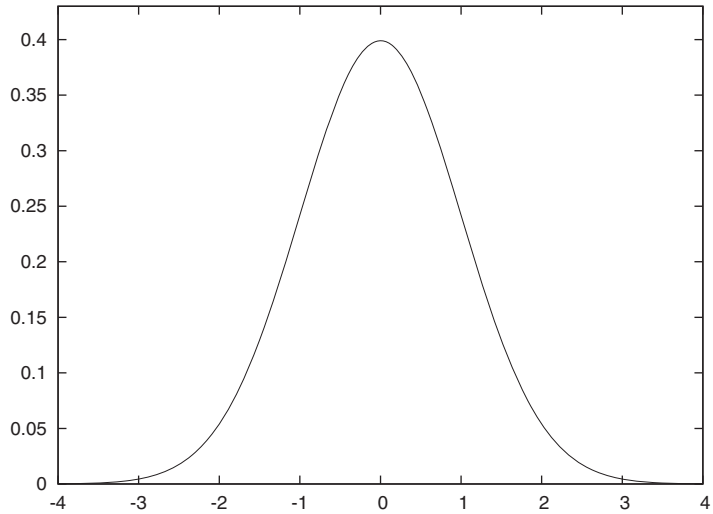

FIGURE B-9. The Gaussian: $y = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}$.

Finally, I should mention the tangent function, which is occasionally useful:

$$\tan x = \frac{\sin(x)}{\cos(x)}$$

Gaussian Function and the Normal Distribution

The Gaussian function arises frequently and in many different contexts. It is given by the formula:

$$\phi(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}$$

and its plot is shown in Figure B-9. (This is the form in which the Gaussian should be memorized, with the factor $1/2$ in the exponent and the factor $1/\sqrt{2\pi}$ up front: they ensure that the integral of the Gaussian over all x will be equal to 1.)

Two applications of the Gaussian stand out. First of all, a strong result from probability theory, the *Central Limit Theorem* states that (under rather weak assumptions) if we add many random quantities, then their sum will be distributed according to a Gaussian distribution. In particular, if we take several samples from a population and calculate the mean for each sample, then the sample means will be distributed according to a Gaussian. Because of this, the Gaussian arises *all the time* in probability theory and statistics.

It is because of this connection that the Gaussian is often identified as "the" bell curve—quite incorrectly so, since there are many bell-shaped curves, many of which have drastically different properties. In fact, there are important cases where the Central Limit

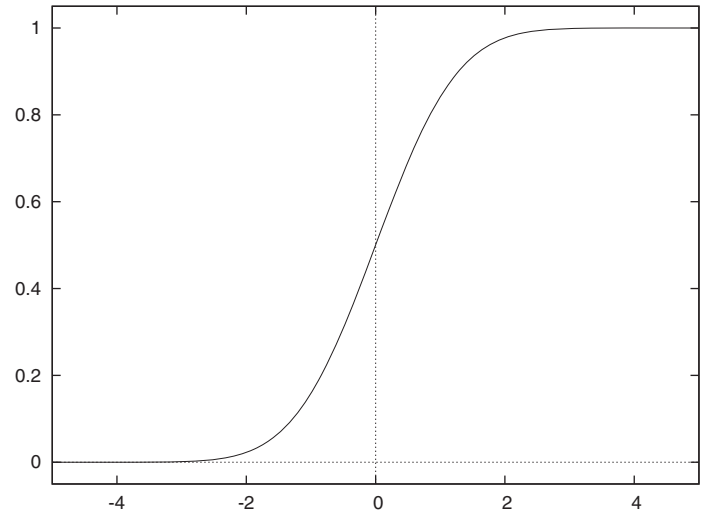

FIGURE B-10. The Gaussian distribution function.

Theorem fails, and the Gaussian is *not* a good way to describe the behavior of a random system (see the discussion of power-law distributions in Chapter 9).

The other context in which the Gaussian arises frequently is as a *kernel*—that is, as a strongly peaked and localized yet very smooth function. Although the Gaussian is greater than zero everywhere, it falls off to zero so quickly that almost the entire area underneath it is concentrated on the interval $-3 \leq x \leq 3$. It is this last property that makes the Gaussian so convenient to use as a kernel. Although the Gaussian is defined and nonzero everywhere (so that we don't need to worry about limits of integration), it can be multiplied against almost any function and integrated. The integral will retain only those values of the function near zero; values at positions far from the origin will be suppressed (smoothly) by the Gaussian.

In statistical applications, we are often interested in the area under certain parts of the curve because that will provide the answer to questions such as: "What is the probability that the point lies between -1 and 1?" The antiderivative of the Gaussian cannot be expressed in terms of elementary functions; instead it is defined through the integral directly:

$$\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{1}{2}t^2} \, \mathrm{d}t$$

This function is known as the *Normal distribution function* (see Figure B-10). As previously mentioned, the factor $1/\sqrt{2\pi}$ is a normalization constant that ensures the area under the entire curve is 1.

Given the function $\Phi(x)$, a question like the one just given can be answered easily: the area over the interval $[-1, 1]$ is simply $\Phi(1) - \Phi(-1)$.

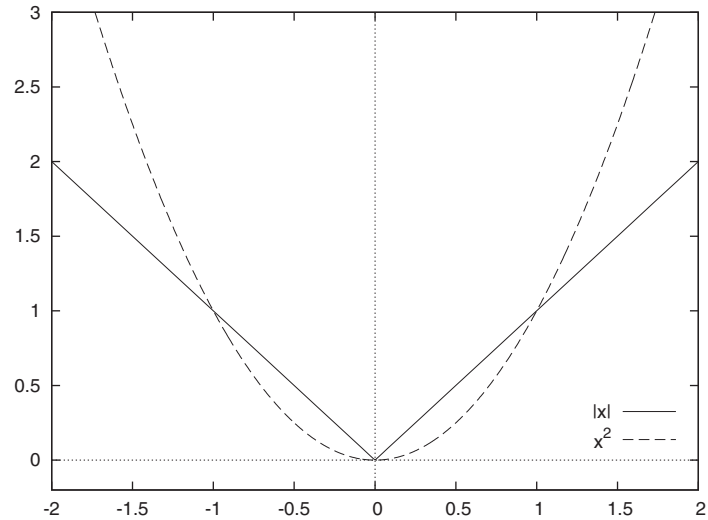

FIGURE B-11. The absolute value function $y = |x|$ and the square $y = x^2$.

Other Functions

There are some other functions that appear in applications often enough that we should be familiar with them but are a bit more exotic than the families of functions considered so far.

The *absolute value* function is defined as:

$$|a| = \begin{cases} a & \text{if } a \geq 0 \\ -a & \text{otherwise} \end{cases}$$

In other words, it is the positive ("absolute") value of its argument. From a mathematical perspective, the absolute value is hard to work with because of the need to treat the two possible cases separately and because of the kink at $x = 0$, which poses difficulties when doing analytical work. For this reason, one instead often uses the square x^2 to guarantee a positive value. The square relieves us of the need to worry about special cases explicitly, and it is smooth throughout. However, the square is relatively smaller than the absolute value for small values of x but relatively larger for large values of x. Weight functions based on the square (as in least-squares methods, for instance) therefore tend to overemphasize outliers (see Figure B-11).

Both the *hyperbolic tangent* $\tanh(x)$ (pronounced: tan-sh) and the *logistic function* are S-shaped or sigmoidal functions. The latter function is the solution to the *logistic differential equation*, hence the name. The logistic differential equation is used to model constrained

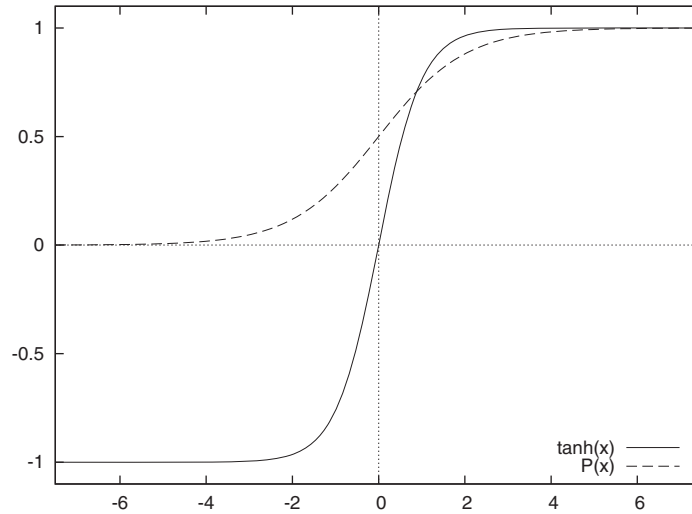

FIGURE B-12. Two sigmoid (step) functions: the hyperbolic tangent $y = \tanh(x)$ *and the logistic function* $y = 1/(1 + e^{-x})$.

growth processes such as bacteria competing for food and infection rates for contagious diseases. Both these functions are defined in terms of the exponential functions as follows:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$\mathrm{P}(x) = \frac{1}{1 + e^{-x}}$$

Both functions are smooth approximations to a step function, and they differ mostly in the range of values they assume: the $\tanh(x)$ takes on values in the interval $[-1, 1]$, whereas the logistic function takes on only positive values between 0 and 1 (see Figure B-12). It is not hard to show that the two functions can be transformed into each other; in fact, we have $\mathrm{P}(x) = (\tanh(x/2) + 1)/2$.

These two functions are each occasionally referred to as *the* sigmoid function. That is incorrect: there are infinitely many functions that smoothly interpolate a step function. But among those functions, the two discussed here have the advantage that—although everywhere smooth—they basically consist of three straight lines: very flat as x goes to plus or minus infinity and almost linear in the transition regime. The position and steepness of the transition can be changed through a standard variable transformation; for example, $\tanh((x - m)/a)$ will have a transition at m with local slope $1/a$.

The last function to consider here is the *factorial*: $n!$. The factorial is defined only for nonnegative integers, as follows:

$$0! = 1$$

$$n! = 1 \cdot 2 \cdot \cdots \cdot (n - 1) \cdot n$$

The factorial plays an important role in combinatorial problems, since it is the number of ways that n distinguishable objects can be arranged. (To see this, imagine that you have to fill n boxes with n items. To fill the first box, you have n choices. To fill the second box, you have $n-1$ choices. And so on. The total number of arrangements or *permutations* is therefore $n \cdot (n-1) \cdots 1 = n!$.)

The factorial grows *very* quickly; it grows faster even than the exponential. Because the factorial grows so quickly, it is often convenient to work with its logarithm. An important and widely used approximation for the logarithm of the factorial is *Stirling's approximation*:

$$\log n! \approx n \log(n) - n \qquad \text{for large } n$$

For the curious: it is possible to define a function that smoothly interpolates the factorial for all positive numbers (not just integers). It is known as the *Gamma function*, and it is another example (besides the Gaussian distribution function) for a function defined through an integral:

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} \, \mathrm{d}t$$

The variable t in this expression is just a "dummy" variable of integration—it does not appear in the final result. You can see that the first term in the integral grows as a power while the second falls exponentially, with the effect that the value of the integral is finite. Note that the limits of integration are fixed. The independent variable x enters the expression only as a parameter. Finally, it is easy to show that the Gamma function obeys the rule $n\,\Gamma(n) = \Gamma(n+1)$, which is the defining property of the factorial function.

We do not need the Gamma function in this book, but it is interesting as an example of how integrals can be used to define and construct new functions.

The Inverse of a Function

A function maps its argument to a result: given a value for x, we can find the corresponding value of $f(x)$. Occasionally, we want to turn this relation around and ask: given a value of $f(x)$, what is the corresponding value of x?

That's what the *inverse function* does: if $f(x)$ is some function, then its inverse $f^{-1}(x)$ is defined as the function that, when applied to $f(x)$, returns the original argument:

$$f^{-1}\left(f(x)\right) = x$$

Sometimes we can invert a function explicitly. For example, if $f(x) = x^2$, then the inverse function is the square root, because $\sqrt{x^2} = x$ (which is the definition of the inverse function). In a similar way, the logarithm is the inverse function of the exponential: $\log(e^x) = x$.

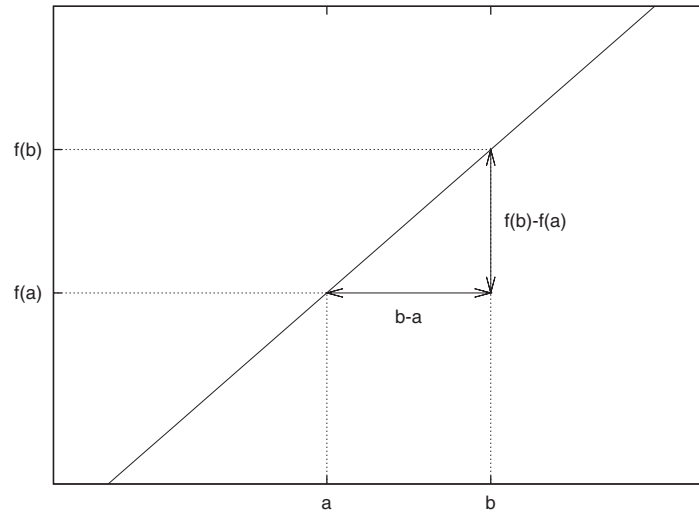

FIGURE B-13. The slope of a linear function is the ratio of the growth in the vertical direction, $f(b) - f(a)$, divided by the corresponding growth in the horizontal direction, $b - a$.

In other cases, it may not be possible to find an explicit form for the inverse function. For example, we sometimes need the inverse of the Gaussian distribution function $\Phi(x)$. However, no simple form for this function exists, so we write it symbolically as $\Phi^{-1}(x)$, which refers to the function for which $\Phi^{-1}(\Phi(x)) = x$ is true.

Calculus

Calculus proper deals with the consideration of limit processes: how does a sequence of values behave if we make infinitely many steps? The slope of a function and the area underneath a function are both defined through such limit processes (the derivative and the integral, respectively).

Calculus allows us to make statements about properties of functions and also to develop approximations.

Derivatives

We already mentioned the slope as the rate of change of a linear function. The same concept can be extended to nonlinear functions, though for such functions, the slope itself will vary from place to place. For this reason, we speak of the *local slope* of a curve at each point.

Let's examine the slope as the *rate of change* of a function in more detail, because this concept is of fundamental importance whenever we want to interpolate or approximate some data by a smooth function. Figure B-13 shows the construction used to calculate the

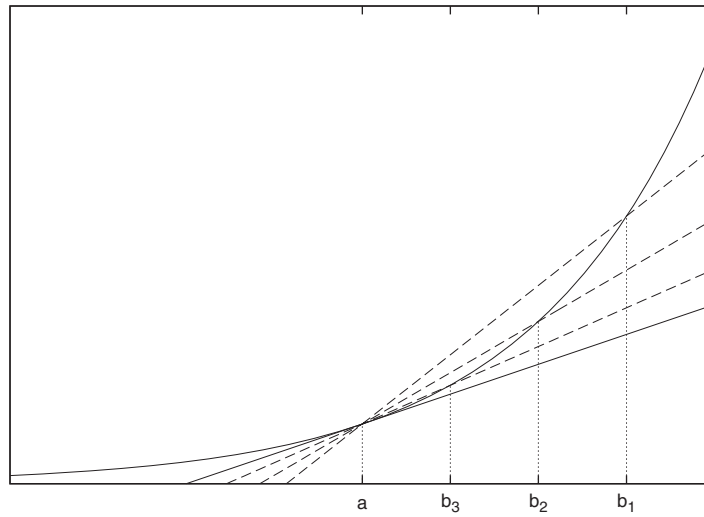

FIGURE B-14. As b_i approaches a, the slope found for these two points becomes closer and closer to the local slope at a.

slope of a linear function. As x goes from a to b, the function changes from $f(a)$ to $f(b)$. The rate of change is the ratio of the change in $f(x)$ to the change in x:

$$\text{slope} = \frac{f(b) - f(a)}{b - a}$$

Make sure that you really understand this formula!

Now, let's apply this concept to a function that is nonlinear. Because the slope of the curve varies from point to point, we cannot find the slope directly using the previous formula; however, we can use the formula to *approximate* the local slope.

Figure B-14 demonstrates the concept. We fix two points on a curve and put a straight line through them. This line has a slope, which is $\frac{f(b)-f(a)}{b-a}$. This is only an approximation to the slope at point a. But we can improve the approximation by moving the second point b closer to a. If we let b go all the way to a, we end up with the (local) slope *at* the point a exactly. This is called the *derivative*. (It is a central result of calculus that, although numerator and denominator in $\frac{f(b)-f(a)}{b-a}$ each go to zero separately in this process, the fraction itself goes to a well-defined value.)

The construction just performed was done graphically and for a single point only, but it can be carried out analytically in a fully general way. The process is sufficiently instructive that we shall study a simple example in detail—namely finding a general rule for the derivative of the function $f(x) = x^2$. It will be useful to rewrite the interval $[a, b]$ as

TABLE B-1. Derivatives and antiderivatives (integrals) for a few elementary functions.

Function	Derivative	Integral
x^n	nx^{n-1}	$\frac{1}{n+1}x^{n+1}$
e^x	e^x	e^x
$\log x$	$1/x$	$x \log x - x$
$\sin x$	$\cos x$	$-\cos x$
$\cos x$	$-\sin x$	$\sin x$

$[x, x+\epsilon]$. We can now go ahead and form the familiar ratio:

$$\begin{aligned}\frac{f(b)-f(a)}{b-a} &= \frac{f(x+\epsilon)-f(x)}{(x+\epsilon)-x} \\ &= \frac{(x+\epsilon)^2-x^2}{x+\epsilon-x} \\ &= \frac{x^2+2x\epsilon+\epsilon^2-x^2}{\epsilon} \\ &= \frac{2x\epsilon+\epsilon^2}{\epsilon} \\ &= 2x+\epsilon \\ &\to 2x \quad \text{as } \epsilon \text{ goes to zero}\end{aligned}$$

In the second step, the terms not depending on ϵ cancel each other; in the third step, we cancel an ϵ between the numerator and the denominator, which leaves an expression that is perfectly harmless as ϵ goes to zero! The (harmless) result is the sought-for derivative of the function. Notice that the result is true for *any* x, so we have obtained an expression for the derivative of x^2 that holds for all x: the derivative of x^2 is $2x$. Always. Similar rules can be set up for other functions (you may try your hand at finding the rule for x^3 or even x^k for general k). Table B-1 lists a few of the most important ones.

There are two ways to indicate the derivative. A short form uses the prime, like this: $f'(x)$ is the derivative of $f(x)$. Another form uses the *differential operator* $\frac{\mathrm{d}}{\mathrm{d}x}$, which acts on the expression to its right. Using the latter, we can write:

$$\frac{\mathrm{d}}{\mathrm{d}x}x^2 = 2x$$

Finding Minima and Maxima

When a smooth function reaches a local minimum or maximum, its slope at that point is zero. This is easy to see: as you approach a peak, you go uphill (positive slope); once over the top, you go downhill (negative slope). Hence, you must have passed a point where you were going neither uphill nor downhill—in other words, where the slope was zero. (From a mathematically rigorous point of view, this is not quite as obvious as it may seem; you may want to check for "Rolle's theorem" in a calculus text.)

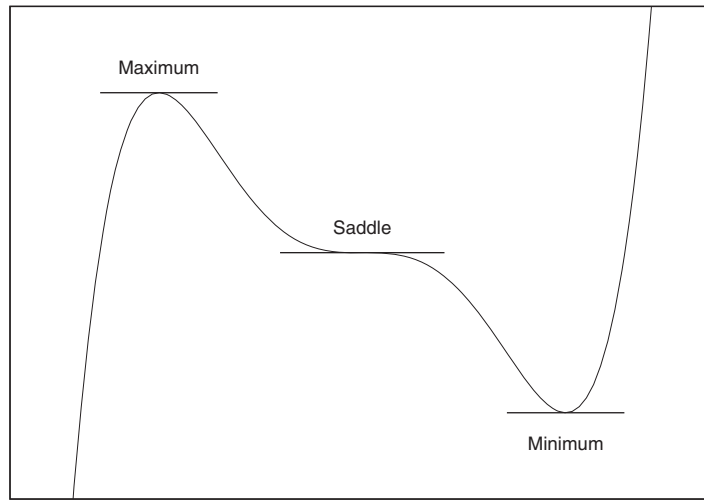

FIGURE B-15. The slope of a curve is zero when the curve reaches a maximum, a minimum, or a saddle point. Zeros in the derivative therefore indicate the occurrence of one of those special points.

The opposite is also true: if the slope (*i.e.*, the derivative) is zero somewhere, then the function has either a minimum or a maximum at that position. (There is also a third possibility: the function has a so-called saddle point there. In practice, this occurs less frequently.) Figure B-15 demonstrates all these cases.

We can therefore use derivatives to locate minima or maxima of a function. First we determine the derivative of the function, and then we find the locations where the derivative is zero (the derivative's *roots*). The roots are the locations of the extrema of the original function.

Extrema are important because they are the solution to *optimization* problems. Whenever we want to find the "best" solution in some context, we are looking for an extremum: the lowest price, the longest duration, the greatest utilization, the highest efficiency. Hence, if we have a mathematical expression for the price, duration, utilization, or efficiency, we can take its derivative with respect to its parameters, set the derivative to zero, and solve for those values of the parameters that maximize (or minimize) our objective function.

Integrals

Derivatives find the local rate of change of a curve as the limit of a sequence of better and better approximations. Integrals calculate the area underneath a curve by a similar method.

Figure B-16 demonstrates the process. We approximate the area underneath a curve by using rectangular boxes. As we make the boxes narrower, the approximation becomes more accurate. In the limit of infinitely many boxes of infinitely narrow width, we obtain the exact area under the curve.

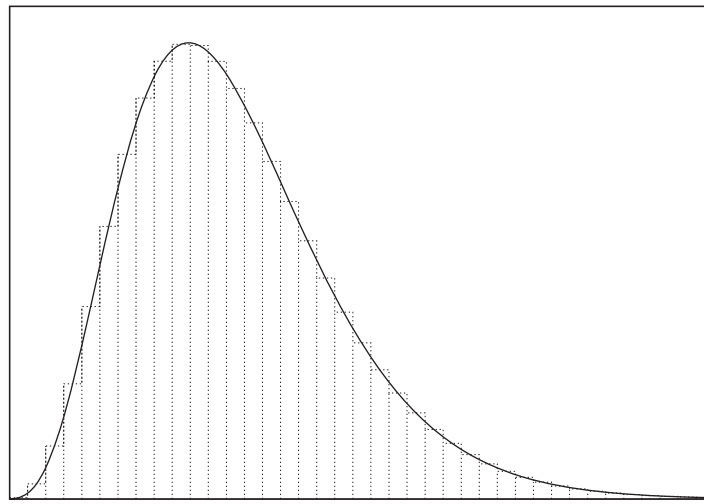

FIGURE B-16. The integral is the area under a curve. It can be approximated by filling the area under the curve with narrow rectangles and adding up their areas. The approximation improves as the width of the rectangles becomes smaller.

Integrals are conceptually very simple but analytically much more difficult than derivatives. It is always possible to find a closed-form expression for the derivative of a function. This is not so for integrals in general, but for some simple functions an expression for the integral can be found. Some examples are included in Table B-1.

Integrals are often denoted using uppercase letters, and there is a special symbol to indicate the "summing" of the area underneath a curve:

$$F(y) = \int f(x)\,\mathrm{d}x$$

We can include the limits of the domain over which we want to integrate, like this:

$$A = \int_a^b f(x)\,\mathrm{d}x$$

Notice that A is a *number*, namely the area underneath the curve between $x = a$ and $x = b$, whereas the indefinite integral (without the limits) is a *function*, which can be evaluated at any point.

Limits, Sequences, and Series

The central concept in all of calculus is the notion of a *limit*. The basic idea is as follows. We construct some process that continues indefinitely and approximates some value ever more closely as the process goes on—but without reaching the limit in any finite number of steps, no matter how many. The important insight is that, even though the limit is never reached, we can nevertheless make statements about the limiting value. The derivative (as the limit of the difference ratio) and the integral (as the limit of the sum of approximating "boxes") are examples that we have already encountered.

As simpler example, consider the numbers $1/1$, $1/2$, $1/3$, $1/4$, ... or $1/n$ in general as n goes to infinity. Clearly, the numbers approach zero ever more closely; nonetheless, for any finite n, the value of $1/n$ is always greater than zero. We call such an infinite, ordered set of numbers a *sequence*, and zero is the limit of this particular sequence.

A *series* is a sum:

$$\begin{aligned} s_n &= \sum_{i=0}^{n} a_n \\ &= a_0 + a_1 + a_2 + a_3 + \cdots + a_n \end{aligned}$$

As long as the number of terms in the series is finite, there is no problem. But once we let the number of terms go to infinity, we need to ask whether the sum still converges to a finite value. We have already seen a case where it does: we defined the integral as the value of the infinite sum of infinitely small boxes.

It may be surprising that an *infinite* sum can still add up to a *finite* value. Yet this can happen provided the terms in the sum become smaller rapidly enough. Here's an example: if you sum up 1, 0.1, 0.01, 0.001, 0.0001, ... , you can see that the sum approaches 1.1111 ... but will never be larger than 1.2. Here is a more dramatic example: I have a piece of chocolate. I break it into two equal parts and give you one. Now I repeat the process with what I have left, and so on. Obviously, we can continue like this forever because I always retain half of what I had before. However, you will never accumulate more chocolate than what I started out with!

An infinite series converges to a finite value only if the magnitude of the terms decreases sufficiently quickly. If the terms do not become smaller fast enough, the series diverges (*i.e.*, its value is infinite). An important series that does *not* converge is the *harmonic series*:

$$\sum_{k=1}^{\infty} \frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots = \infty$$

One can work out rigorous tests to determine whether or not a given series converges. For example, we can compare the terms of the series to those from a series that is known to converge: if the terms in the new series become smaller more quickly than in the converging series, then the new series will also converge.

Finding the value of an infinite sum is often tricky, but there is one example that is rather straightforward. The solution involves a trick well worth knowing. Consider the infinite *geometric series*:

$$s = \sum_{i=0}^{\infty} = 1 + q + q^2 + q^3 + \cdots \qquad \text{for } |q| < 1$$

Now, let's multiply by q and add 1:

$$\begin{aligned} qs + 1 &= q(1 + q + q^2 + q^3 + \cdots) + 1 \\ &= q + q^2 + q^3 + q^4 + \cdots + 1 \\ &= s \end{aligned}$$

To understand the last step, realize that the righthand side equals our earlier definition of s. We can now solve the resulting equation for s and obtain:

$$s = \frac{1}{1 - q}$$

This is a good trick that can be applied in similar cases: if you can express an infinite series in terms of itself, the result may be an equation that you can solve explicitly for the unknown value of the infinite series.

Power Series and Taylor Expansion

An especially important kind of series contains consecutive powers of the variable x multiplied by the constant coefficients a_i. Such series are called *power series*. The variable x can take on any value (it is a "dummy variable"), and the sum of the series is therefore a function of x:

$$s(x) = \sum_{i=0}^{n} a_i x^i$$

If n is finite, then there is only a finite number of terms in the series: in fact, the series is simply a polynomial (and, conversely, every polynomial is a finite power series). But the number of terms can also be infinite, in which case we have to ask for what values of x does the series converge. Infinite power series are of great theoretical interest because they are a (conceptually straightforward) generalization of polynomials and hence represent the "simplest" nonelementary functions.

But power series are also of the utmost *practical* importance. The reason is a remarkable result known as *Taylor's theorem*. Taylor's theorem states that any reasonably smooth function can be *expanded into a power series*. This process (and the resulting series) is known as the *Taylor expansion* of the function.

Taylor's theorem gives an explicit construction for the coefficients in the series expansion:

$$f(x) = f(0) + f'(0)x + \frac{f''(0)}{2!}x^2 + \frac{f'''(0)}{3!}x^3 + \cdots$$

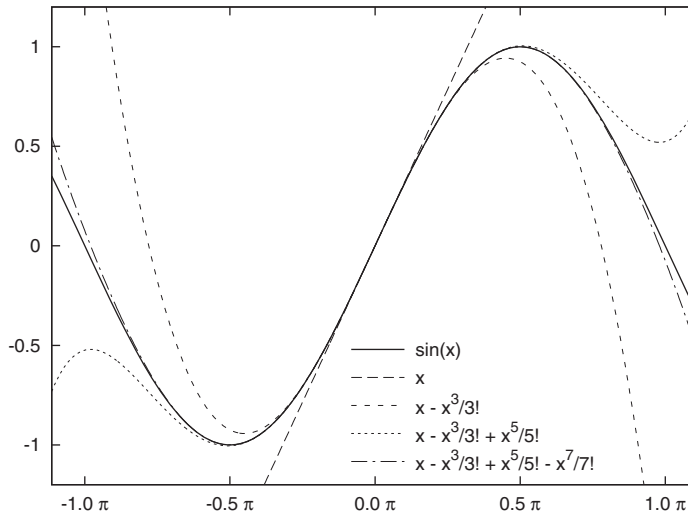

FIGURE B-17. The sine function $\sin(x)$ *and its Taylor expansions around zero, truncated after retaining different numbers of terms. If more terms are kept, the approximation is acceptable over a greater range of values.*

In other words, the coefficient of the nth term is the nth derivative (evaluated at zero) divided by $n!$. The Taylor series converges for *all* x—the factorial in the denominator grows so quickly that convergence is guaranteed no matter how large x is.

The Taylor series is an exact representation of the function on the lefthand side if we retain all (infinitely many) terms. But we can also *truncate* the series after just a few terms and so obtain a good *local approximation* of the function in question. The more terms we keep, the larger will be the range over which the approximation is good. For the sine function, Figure B-17 shows how the Taylor expansion improves as a greater number of terms is kept. Table B-2 shows the Taylor expansions for some functions we have encountered so far.

It is this last step that makes Taylor's theorem so useful from a practical point of view: it tells us that *we can approximate any smooth function locally by a polynomial*. And polynomials are always easy to work with—often much easier than the complicated functions that we started with.

One important practical point: the approximation provided by a truncated Taylor series is good only *locally*—that is, near the point around which we expand. This is because in that case x is small (*i.e.*, $x \ll 1$) and so higher powers become negligible fast. Taylor series are usually represented in a form that assumes that the expansion takes place around zero. If this is not the case, we need to remove or factor out some large quantity so that we are left with a "small parameter" in which to expand. As an example, suppose we want to obtain an approximation to e^x for values of x near 10. If we expanded in the usual fashion around zero, then we would have to sum *many* terms before the approximation becomes

TABLE B-2. The first few terms of the Taylor expansion of some important functions

Function	Taylor expansion	Comment
e^x	$1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$	all x
$\sin x$	$x - \frac{x^3}{3!} + \frac{x^5}{5!} \mp \cdots$	all x
$\cos x$	$1 - \frac{x^2}{2!} + \frac{x^4}{4!} \mp \cdots$	all x
$\log(1+x)$	$x - \frac{x^2}{2} + \frac{x^3}{3} \mp \cdots$	$-1 < x \leq 1$
$\sqrt{1+x}$	$1 + \frac{x}{2} + \frac{x^2}{8} + \frac{x^3}{16} + \cdots$	$\lvert x\rvert \leq 1$
$1/(1+x)$	$1 - x + x^2 - x^3 \pm \cdots$	$\lvert x\rvert < 1$

good (the terms grow until $10^n < n!$, which means we need to keep more than 20 terms). Instead, we proceed as follows: we write $e^x = e^{10+\delta} = e^{10}\, e^{\delta} = e^{10}\,(1 + \delta + \frac{\delta^2}{2} + \cdots)$. In other words, we set it up so that δ is small allowing us to expand e^{δ} around zero as before.

Another important point to keep in mind is that the function must be smooth at the point around which we expand: it must not have a kink or other singularity there. This is why the logarithm is usually expanded around one (not zero): recall that the logarithm diverges as x goes to zero.

Useful Tricks

The Binomial Theorem

Probably everyone has encountered the binomial formulas at some point:

$$(a+b)^2 = a^2 + 2ab + b^2$$
$$(a-b)^2 = a^2 - 2ab + b^2$$

The binomial theorem provides an extension of this result to higher powers. The theorem states that, for an arbitrary integer power n, the expansion of the lefthand side can be written as:

$$\begin{aligned}(a+b)^n &= \sum_{k=0}^{n} \binom{n}{k} a^{n-k} b^k \\ &= \binom{n}{0} a^n b^0 + \binom{n}{1} a^{n-1} b^1 + \binom{n}{2} a^{n-2} b^2 + \cdots + \binom{n}{n} a^0 b^n\end{aligned}$$

This complicated-looking expression involves the *binomial coefficients*:

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!} \qquad 0 \leq k \leq n$$

The binomial coefficients are combinatorial factors that count the number of different ways one can choose k items from a set of n items, and in fact there is a close relationship between the binomial theorem and the binomial probability distribution.

As is the case for many exact results, the greatest practical use of the binomial theorem comes from an approximate expression. Assume that $b < a$, so that $b/a < 1$. Now we can write:

$$\begin{aligned}(a+b)^n &= a^n \left(1 + \frac{b}{a}\right)^n \\ &\approx a^n \left(1 + n\frac{b}{a} + \frac{n(n-1)}{2}\left(\frac{b}{a}\right)^2 + \cdots\right)\end{aligned}$$

Here we have neglected terms involving higher powers of b/a, which are small compared to the retained terms, since $b/a < 1$ by construction (so that higher powers of b/a, which involve multiplying a small number repeatedly by itself, quickly become negligible).

In this form, the binomial theorem is frequently useful as a way to generate approximate expansions. In particular, the first-order approximation:

$$(1+x)^n \approx 1 + nx \qquad \text{for } |x| < 1$$

should be memorized.

The Linear Transformation

Here is a quick, almost trivial, trick that is useful enough to be committed to memory. Any variable can be transformed to a similar variable that takes on only values from the interval [0, 1], via the following linear transformation, where $x_{\min}$ and $x_{\max}$ are the minimum and maximum values that x can take on:

$$y = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$$

This transformation is frequently useful—for instance, if we have two quantities and would like to compare how they develop over time. If the two quantities have very different magnitudes, then we need to reduce both of them to a common range of values. The transformation just given does exactly that.

If we want the transformed quantity to *fall* whenever the original quantity goes up, we can do this by writing:

$$\bar{y} = 1 - y = 1 - \frac{x - x_{\min}}{x_{\max} - x_{\min}}$$

We don't have to shift by $x_{\min}$ and rescale by the original range $x_{\max} - x_{\min}$. Instead, we can subtract any "typical" value and divide by any "typical" measure of the range. In statistical applications, for example, it is frequently useful to subtract the mean μ and to divide by the standard deviation σ. The resulting quantity is referred to as the *z-score*:

$$z = \frac{x - \mu}{\sigma}$$

Alternatively, you might also subtract the median and divide by the inter-quartile range. The exact choice of parameters is not crucial and will depend on the specific application context. The important takeaway here is that we can normalize any variable by:

- Subtracting a typical value (shifting) and
- Dividing by the typical range (rescaling)

Dividing by Zero

Please remember that *you cannot divide by zero!* I am sure you know this—but it's surprisingly easy to forget (until the computer reminds us with a fatal "divide by zero" error).

It is instructive to understand what happens if you try to divide by zero. Take some fixed number (say, 1), and divide it by a sequence of numbers that approach zero:

$$\frac{1}{10} = 0.1$$
$$\frac{1}{5} = 0.2$$
$$\frac{1}{1} = 1.0$$
$$\frac{1}{1/5} = 5$$
$$\frac{1}{1/10} = 10$$
$$\frac{1}{0} = \ ?$$

In other words, as you divide a constant by numbers that *approach* zero, the result becomes larger and larger. Finally, if you let the divisor go to zero, the result grows beyond all bounds: it diverges. Figure B-18 shows this graphically.

What you should take away from this exercise and Figure B-18 is that you cannot replace 1/0 by something else—for instance, it is *not* a smart move to replace 1/0 by 0 "because both don't really mean anything, anyway." If you need to find a numeric value for 1/0, then it should be something like "infinity," but this is not a useful value to operate with in practical applications.

Therefore, *whenever you encounter a fraction* $\frac{a}{b}$ *of any kind, you* must *check whether the denominator can become zero and exclude these points from consideration*.

Failing to do so is one of the most common sources of error. What is worse, these errors are difficult to recover from—not just in implementations but also conceptually. A typical

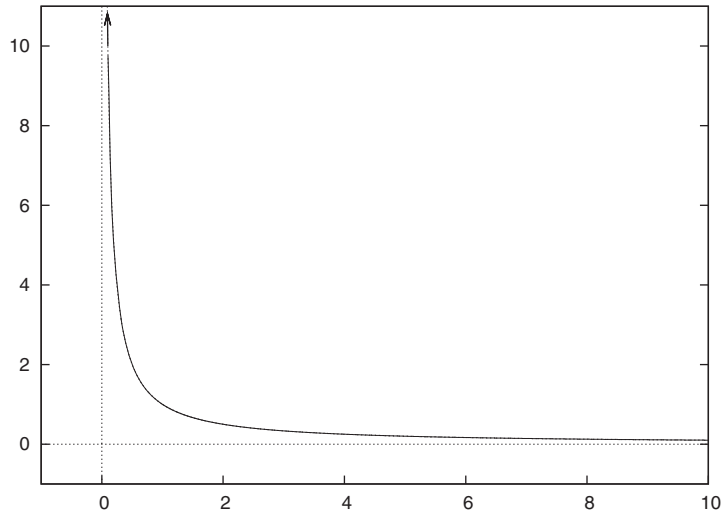

FIGURE B-18. As you divide a constant value by smaller and smaller numbers, the result is getting larger and larger. If you divide by zero, it blows up!

example involves "relative errors," where we divide the difference between the observed and the expected value by the expected value:

$$\text{relative error} = \frac{\text{observed} - \text{expected}}{\text{expected}}$$

What happens if for one day the expected value drops to zero? You are toast. There is no way to assign a meaningful value to the error in this case. (If the observed value is also zero, then you can treat this as a special case and *define* the relative error to be zero in this case, but if the observed value is not zero, then this definition is obviously inappropriate.)

These kinds of problems have an unpleasant ability to sneak up on you. A quantity such as the relative error or the defect rate (which is also a ratio: the number of defects found divided by the number of units produced) is a quantity commonly found in reports and dashboards. You don't want your entire report to crash because no units were produced for some product on this day rendering the denominator zero in one of your formulas!

There are a couple of workarounds, neither of which is perfect. In the case of the defect rate, where you can be sure that the numerator will be zero if the denominator is (because no defects can be found if no items were produced), you can add a small positive number to the denominator and thereby prevent it from ever becoming exactly zero. As long as this number is small compared to the number of items typically produced in a day, it will not significantly affect the reported defect rate, but will relieve you from having to check for the $\frac{0}{0}$ special case explicitly. In the case of calculating a relative error, you might want to replace the numerator with the average of the expected and the observed values. The advantage is that now the denominator can be zero only if the numerator is zero,

which brings us back to the suggestion for dealing with defect rates just discussed. The problem with this method is that when no events are observed but some number was expected, the relative error is reported as -2 (negative 200 percent instead of negative 100 percent); this is due to the factor $1/2$ in the denominator, which comes from calculating the average there.

So, let me say it again: whenever you are dealing with fractions, you *must* consider the case of denominators becoming zero. Either rule them out or handle them explicitly.

Notation and Basic Math

This section is not intended as a comprehensive overview of mathematical notation or as your first introduction to mathematical formulas. Rather, it should serve as a general reminder of some basic facts and to clarify some conventions used in this book. (All my conventions are pretty standard—I have been careful not to use any symbols or conventions that are not generally used and understood.)

On Reading Formulas

A mathematical formula combines different components, called *terms*, by use of operators. The most basic operators are *plus* and *minus* ($+$ and $-$) and *multiplied by* and *divided by* ($\cdot$ and $/$). Plus and minus are always written explicitly, but the multiplication operator is usually silent—in other words, if you see two terms next to each other, with nothing between them, they should be multiplied. The division operator can be written in two forms: $1/n$ or $\frac{1}{n}$, which mean exactly the same thing. The former is more convenient in text such as this; the latter is more clear for long, "display" equations. An expression such as $1/n + 1$ is ambiguous and should not be used, but if you encounter it, you should assume that it means $\frac{1}{n} + 1$ and not $1/(n + 1)$ (which is equivalent to $\frac{1}{n+1}$).

Multiplication and division have higher precedence than addition and subtraction, therefore $ab + c$ means that first you multiply a and b and then add c to the result. To change the priority, you need to use parentheses: $a(b + c)$ means that first you add b and c and then multiply the result by a. Parentheses can either be round $(\dots)$ or square $[\dots]$, but their meaning is the same.

Functions take one (or several) arguments and return a result. A function always has a *name* followed by the *arguments*. Usually the arguments are enclosed in parentheses: $f(x)$. Strictly speaking, this notation is ambiguous because an expression such as $f(a + b)$ could mean either "add a and b and then multiply by f" or "add a and b and then pass the result to the function f." However, the meaning is usually clear from the context.

(There is a slightly more advanced way to look at this. You can think of f as an operator, similar to a differential operator like $\frac{\mathrm{d}}{\mathrm{d}x}$ or an integral operator like $\int \mathrm{d}t$. This operator is now applied to the expression to the right of it. If f is a function, this means applying the

function to the argument; if the operator is a differential operator, this means taking the derivative; and if f is merely a number, then applying it simply means multiplying the term on its right by it.)

A function may take more than one argument; for example, the function $f(x, y, z)$ takes three arguments. Sometimes you may want to emphasize that not all of these arguments are equivalent: some are actual variables, whereas others are "parameters," which are kept constant while the variables change. Consider $f(x) = ax + b$. In this function, x is the variable (the quantity usually plotted along the horizontal axis) while a and b would be considered parameters. If we want to express that the function f does depend on the parameters as well as on the actual variable, we can do this by including the parameters in the list of arguments: $f(x, a, b)$. To visually separate the parameters from the actual variable (or variables), a semicolon is sometimes used: $f(x; a, b)$. There are no hard-and-fast rules for when to use a semicolon instead of a comma—it's simply a convenience that is sometimes used and other times not.

One more word on functions: several functions are regarded as "well known" in mathematics (such as sine and cosine, the exponential function, and the logarithm). The names of such well-known functions are always written in upright letters, whereas functions in general are denoted by an italic letter. (Variables are always written in italics.) For well-known functions, the parentheses around the arguments can be omitted if the argument is sufficiently simple. (This is another example of the "operator" point of view mentioned earlier.) Thus we may write $\sin(x + 1) + \log x - f(x)$ (note the upright letters for sine and logarithm, and the parentheses around the argument for the logarithm have been omitted, because it consists of only a single term). This has a different meaning than: $\sin(x + 1) + \log(x - f(x))$.

Elementary Algebra

For numbers, the following is generally true:

$$a(b + c) = ab + ac$$

This is often applied in situations like the following, where we *factor out* the a:

$$a + b = a(1 + b/a)$$

If a is much greater than b, then we have now converted the original expression $a + b$ into another expression of the form:

$$\textit{something large} \cdot (1 + \textit{something small})$$

which makes it easy to see which terms matter and which can be neglected in an approximation scheme. (The small term in the parentheses is "small" compared to the 1 in the parentheses and can therefore be treated as a perturbation.)

Quantities can be multiplied together, which gives rise to *powers*:

$$a \cdot a = a^2$$
$$a \cdot a \cdot a = a^3$$
$$\ldots$$

The raised quantity (the superscript) is also referred to as the *exponent*. In this book, superscripts always denote powers.

The three binomial formulas should be committed to memory:

$$(a+b)^2 = a^2 + 2ab + b^2$$
$$(a-b)^2 = a^2 - 2ab + b^2$$
$$(a+b)(a-b) = a^2 - b^2$$

Because the easiest things are often the most readily forgotten, let me just work out the first of these identities explicitly:

$$\begin{aligned}(a+b)^2 &= (a+b)(a+b)\\ &= a(a+b) + b(a+b)\\ &= a^2 + ab + ba + b^2\\ &= a^2 + 2ab + b^2\end{aligned}$$

where I have made use of the fact that $ab = ba$.

Working with Fractions

Let's review the basic rules for working with fractions. The expression on top is called the *numerator*, the one at the bottom is the *denominator*:

$$\frac{\text{numerator}}{\text{denominator}}$$

If you can factor out a common factor in both numerator and denominator, then this common factor can be canceled:

$$\frac{2+4x}{2+2\sin(y)} = \frac{2(1+2x)}{2(1+\sin(y)} = \frac{1+2x}{1+\sin y}$$

To add two fractions, you have to bring them onto a common denominator in an operation that is the opposite of canceling a common factor:

$$\frac{1}{a} + \frac{1}{b} = \frac{a}{ab} + \frac{b}{ab} = \frac{a+b}{ab}$$

Here is a numeric example:

$$\frac{1}{2} + \frac{1}{3} = \frac{3}{6} + \frac{2}{6} = \frac{5}{6}$$

Sets, Sequences, and Series

A *set* is a grouping of elements in no particular order. In a *sequence*, the elements occur in a fixed order, one after the other.

The individual elements of sets and sequences are usually shown with subscripts that denote the index of the element in the set or its position in the sequence (similar to indexing into an array). In this book, subscripts are used only for the purpose of indexing elements of sets or sequences in this way.

Sets are usually indicated by curly braces. The following expressions are equivalent:

$$\{x_1, x_2, x_3, \ldots, x_n\}$$
$$\{x_i \mid i = 1, \ldots, n\}$$

For brevity, it is customary to suppress the range of the index if it can be understood from context. For example, if it is clear that there are n elements in the set, I might simply write $\{x_i\}$.

One often wants to sum a finite or infinite sequence of numbers; the result is known as a *series*:

$$x_1 + x_2 + x_3 + \cdots + x_n$$

Instead of writing out the terms explicitly, it is often useful to use the sum notation:

$$\sum_{i=1}^{n} x_i = x_1 + x_2 + x_3 + \cdots + x_n$$

The meaning of the summation symbol should be clear from this example. The variable used as index (here, i) is written underneath the summation sign followed by the lower limit (here, 1). The upper limit (here, n) is written above the summation sign. As a shorthand, any one of these specifications can be omitted. For instance, if it is clear from the context that the lower limit is 1 and the upper limit is n, then I might simply write $\sum_i x_i$ or even $\sum x_i$. In the latter form, it is understood that the sum runs over the index of the summands.

It is often convenient to describe the terms to be summed over in words, rather than giving specific limits:

$$\sum_{\text{all data points}} x_i$$

Some standard transformations involving the summation notation are used fairly often. For example, one frequently needs to shift indices. The following three expressions are equal, as you can easily see by writing out explicitly the terms of the sum in each case:

$$\sum_{i=0}^{n} x_i = \sum_{i=1}^{n+1} x_{i-1} = x_0 + \sum_{i=1}^{n} x_i$$

Keep in mind that the summation notation is just a shorthand for the explicit form given at the start of this section. If you become confused, you can always write out the terms explicitly to understand what is going on.

Finally, we may take the upper limit of the sum to be infinity, in which case the sum runs over infinitely many terms. Infinite series play a fundamental role in the theoretical development of mathematics, but all series that you will encounter in applications are, of course, finite.

Special Symbols

A few mathematical symbols are either indispensable or so useful that I wouldn't do without them.

Binary relationships

There are several special symbols to describe the relationship between two expressions. Some of the most useful ones are listed in Table B-3.

TABLE B-3. Commonly used relational operators

Operator	Meaning
$= \neq$	equal to, not equal to
$< >$	less than, greater than
$\leq \geq$	less than or equal to, greater than or equal to
$\ll \gg$	much less than, much greater than
$\propto$	proportional to
$\approx$	approximately equal to
$\sim$	scales as

The last three might require a word of explanation. We say two quantities are *approximately equal* when they are equal up to a "small" error. Put differently, the difference between the two quantities must be small compared to the quantities themselves: x and $1.1x$ are approximately equal, $x \approx 1.1x$, because the difference (which is $0.1x$) is small compared to x.

One quantity is *proportional* to another if they are equal up to a constant factor that has been omitted from the expression. Often, this factor will have units associated with it. For example, when we say "time is money," what we really mean is:

$$\text{money} \propto \text{time}$$

Here the omitted constant of proportionality is the hourly rate (which is also required to fix the units: hours on the left, dollars on the right; hence hourly rate must have units of "dollars per hour" to make the equation dimensionally consistent).

We say that a quantity *scales as* some other quantity if we want to express how one quantity depends on another one in a very general way. For example, recall that the area of a circle is πr^2 (where r is the length of the radius) but that the area of a square is a^2 (where a is the length of the side of the square). We can now say that "the area *scales as* the square of the length." This is a more general statement than saying that the area is proportional to the square of the length: the latter implies that they are equal up to a constant factor, whereas the scaling behavior allows for more complicated dependencies. (In this example, the constant of proportionality depends on the *shape* of the figure, but the scaling behavior area $\sim$ length^2 is true for all symmetrical figures.)

In particular when evaluating the complexity of algorithms, there is another notation to express a very similar notion: the so-called *big O* notation. For example, the expression $\mathcal{O}(n^2)$ states that the complexity of an algorithm grows ("scales") with the square of the number of elements in the input.

Parentheses and other delimiters

Round parentheses $(\ldots)$ are used for two purposes: to group terms together (establishing precedence) and to indicate the arguments to a function:

$ab + c \neq a(b + c)$	Parentheses to establish precedence
$f(x, y) = x + y$	Parentheses to indicate function arguments

Square brackets $[\ldots]$ are mostly used to indicate an interval:

$$[a, b] \qquad \text{all } x \text{ such that } a \leq x \leq b$$

For the purpose of this book, we don't need to worry about the distinction between closed and open intervals (*i.e.*, intervals that do or don't contain their endpoints, respectively).

Very rarely I use brackets for other purposes—for example as an alternative to round parentheses to establish precedence, or indicate that a function takes another *function* as its argument, as in the expectation value: $E[f(x)]$.

Curly braces $\{\ldots\}$ always denote a set.

Miscellaneous symbols

Two particular constants are indispensable. Everybody has heard of $\pi = 3.141592\ldots$, which is the ratio of the circumference of a circle to its diameter:

$$\pi = \frac{\text{circumference}}{\text{diameter}} = 3.141592\ldots$$

Equally important is the "base of the natural logarithm" $e = 2.718281\ldots$, sometimes called Euler's number. It is defined as the value of the infinite series:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} = 2.718281\ldots$$

The function e^x obtained by raising e to the xth power has the property that its derivative also equals e^x, and it is the only function that equals its derivative (up to a multiplicative constant, to be precise).

The number e also shows up in the definition of the Gaussian function:

$$e^{-x^2}$$

(Any function that contains e raised to $-x^2$ power is called a "Gaussian"; what's crucial is that the x in the exponent is squared and enters with a negative sign. Other constants may appear also, but the $-x^2$ in the exponent is the defining property.)

Because the exponents are often complicated expressions themselves, there is an alternative notation for the exponential function that avoids superscripts and instead uses the function name exp(...). The expression $\exp(x)$ means exactly the same as e^x, and the following two expressions are equivalent, also—but the one on the right is easier to write:

$$e^{-\left(\frac{x-\mu}{\sigma}\right)^2} = \exp\left(-\left(\frac{x-\mu}{\sigma}\right)^2\right)$$

A value of infinite magnitude is indicated by a special symbol:

$$\infty \qquad \text{a value of infinite magnitude}$$

The square root sign $\sqrt{x}$ states that:

$$\text{if} \quad y = \sqrt{x} \quad \text{then} \quad y^2 = x$$

Finally, the integral sign $\int$, which always occurs together with an expression of the form $\mathrm{d}x$ (or $\mathrm{d}t$, or so), is used to denote a generalized form of summation: the expression to the right of the integral sign is to be "summed" for all values of x (or t). If explicit limits of the integration are given, they are attached to the integral sign:

$$\int_0^1 f(x)\,\mathrm{d}x$$

This means: "sum all values of $f(x)$ for x ranging from 0 to 1."

The Greek Alphabet

Greek letters are used all the time in mathematics and other sciences and should be committed to memory. (See Table B-4.)

TABLE B-4. The Greek alphabet

Lowercase	Uppercase	Name
α	A	Alpha
β	B	Beta
γ	Γ	Gamma
δ	Δ	Delta
ϵ	E	Epsilon
ζ	Z	Zeta
η	H	Eta
θ	Θ	Theta
ι	I	Iota
κ	K	Kappa
λ	Λ	Lambda
μ	M	Mu
ν	N	Nu
ξ	Ξ	Xi
o	O	Omicron
π	Π	Pi
ρ	R	Rho
σ	Σ	Sigma
τ	T	Tau
υ	Υ	Upsilon
ϕ	Φ	Phi
χ	X	Chi
ψ	Ψ	Psi
ω	Ω	Omega

Where to Go from Here

This appendix can of course only give a cartoon version of the topics mentioned, or—if you have seen this material before—at best serve as a reminder. But most of all, I hope it serves as a *teaser*: mathematics is a wonderfully rich and stimulating topic, and I would hope that in this appendix (and in the rest of this book) I have been able to convey some of its fascination—and perhaps even convinced you to dig a little deeper.

If you want to learn more, here are a couple of hints.

The first topic to explore is calculus (or real analysis). All modern mathematics starts here, and it is here that some of the most frequently used concepts (derivative, integral, Taylor expansion) are properly introduced. It is a must-have.

But if you limit your attention to calculus, you will never get over the idea that mathematics is about "calculating something." To get a sense of what math is *really* all about, you have to go beyond analysis. The next topic in a typical college syllabus is linear algebra. In linear algebra, we go beyond relatively tangible things like curves and numbers and for the first time start to consider concepts in a fully abstract way: spaces, transformations, mappings. What can we say about them *in general* without having to

appeal to any particular realization? Understanding this material requires real mental effort—you have to change the way you think. (Similarly to how you have to change the way you think if you try to learn Lisp or Haskell.) Linear algebra also provides the theoretical underpinnings of all matrix operations and hence for most frequently used numerical routines. (You can't do paper-and-pencil mathematics without calculus, and you can't do numerical mathematics without linear algebra.)

With these two subjects under your belt, you will be able to pick up pretty much any mathematical topic and make sense of it. You might then want to explore complex calculus for the elegance and beauty of its theorems, or functional analysis and Fourier theory (which blend analysis and linear algebra) because of their importance in all application-oriented areas, or take a deeper look at probability theory, with its obvious importance for anything having to do with random data.

On Math

I have observed that there are two misconceptions about mathematics that are particularly prevalent among people coming from a software or computing background. The first misconception holds that mathematics is primarily a prescriptive, calculational (not necessarily numerical) scheme and similar to an Algol-derived programming language: a pseudo-code for expressing algorithms. The other misconception views mathematics as mostly an abstract method for formal reasoning, not dissimilar to certain logic programming environments: a way to manipulate logic statements.

What both of them miss is that mathematics is not a *method* but first and foremost a body of *content* in its own right. You will never understand what mathematics is if you see it only as something you *use* to obtain certain results. Mathematics is, first and foremost, a rich and exciting story in itself.

There is an unfortunate perception among nonmathematicians (and even partially reinforced by this book) that mathematics is about "calculating things." This is not so, and it is probably the most unhelpful misconception about mathematics of all.

In fairness, this point of view is promulgated by many introductory college textbooks. In a thoroughly misguided attempt to make their subject "interesting," they try to motivate mathematical concepts with phony applications to the design of bridges and airplanes, or to calculating the probability of winning at poker. This not only obscures the beauty of the subject but also creates the incorrect impression of mathematics as a utilitarian fingering exercise and almost as a necessary evil.

Finally, I strongly recommend that you stay away from books on popular or recreational math, for two reasons. First, they tend to focus on a small set of topics that can be treated using "elementary" methods (mostly geometry and some basic number theory), and tend to omit most of the conceptually important topics. Furthermore, in their attempt to

present amusing or entertaining snippets of information, they fail to display the rich, interconnected structure of mathematical theory: all you end up with is a book of (stale) jokes.

Further Reading

Calculus

- *The Hitchhiker's Guide to Calculus.* Michael Spivak. Mathematical Association of America. 1995.
  If the material in this appendix is really new to you, then this short (120-page) booklet provides a surprisingly complete, approachable, yet mathematically respectable introduction. Highly recommended for the curious and the confused.
- *Precalculus: A Prelude to Calculus.* Sheldon Axler. Wiley. 2008.
  Axler's book covers the basics: numbers, basic algebra, inequalities, coordinate systems, and functions—including exponential, logarithmic, and trigonometric functions—but it stops short of derivatives and integrals. If you want to brush up on foundational material, this is an excellent text.
- *Calculus.* Michael Spivak. 4th ed., Publish or Perish. 2008.
  This is a comprehensive book on calculus. It concentrates exclusively on the clear development of the mathematical theory and thereby avoids the confusion that often results from an oversupply of (more or less) artificial examples. The presentation is written for the reader who is relatively new to formal mathematical reasoning, and the author does a good job motivating the peculiar arguments required by formal mathematical manipulations. Rightly popular.
- *Yet Another Introduction to Analysis.* Victor Bryant. Cambridge University Press. 1990.
  This short book is intended as a quick introduction for those readers who already possess passing familiarity with the topic and are comfortable with abstract operations.

Linear Algebra

- *Linear Algebra Done Right.* Sheldon Axler. 2nd ed., Springer. 2004.
  This is the best introduction to linear algebra that I am aware of, and it fully lives up to its grandiose title. This book treats linear algebra as abstract theory of mappings, but on a very accessible, advanced undergraduate level. Highly recommended.
- *Linear Algebra.* Klaus Jänich. Springer. 1994.
  This book employs a greater amount of abstract mathematical formalism than the previous entry, but the author tries very hard to explain and motivate all concepts. This book might therefore give a better sense of the nature of abstract algebraic arguments than Axler's streamlined presentation. The book is written for a first-year course at German universities; the style of the presentation may appear exotic to the American reader.

Complex Analysis

- *Complex Analysis.* Joseph Bak and Donald J. Newman. 2nd ed., Springer. 1996.
  This is a straightforward, and relatively short, introduction to all the standard topics of classical complex analysis.
- *Complex Variables.* Mark J. Ablowitz and Athanassios S. Fokas. 2nd ed., Cambridge University Press. 2003.
  This is a much more comprehensive and advanced book. It is split into two parts: the first part developing the theory, the second part discussing several nontrivial applications (mostly to the theory of differential equations).
- *Fourier Analysis and Its Applications.* Gerald B. Folland. American Mathematical Society. 2009.
  This is a terrific introduction to Fourier theory. The book places a strong emphasis on the solution of partial differential equations but in the course of it also develops the basics of function spaces, orthogonal polynomials, and eigenfunction expansions. The later chapters give an introduction to distributions and Green's functions. This is a very accessible book, but you will need a strong grounding in real and complex analysis, as well as some linear algebra.

Mindbenders

If you *really* want to know what math is like, pick up any one of these. You don't have to understand everything—just get the flavor of it all. None of them are "useful," all are fascinating.

- *A Primer of Analytic Number Theory.* Jeffrey Stopple. Cambridge University Press. 2003.
  This is an amazing book in every respect. The author takes one of the most advanced, obscure, and "useless" topics—namely analytic number theory—and makes it completely accessible to anyone having even minimal familiarity with calculus concepts (and even those are not strictly required). In the course of the book, the author introduces series expansions, complex numbers, and many results from calculus, finally arriving at one of the great unsolved problems in mathematics: the Riemann hypothesis. If you want to know what math *really* is, read this book!
- *The Computer As Crucible: An Introduction to Experimental Mathematics.* Jonathan Borwein and Keith Devlin. AK Peters. 2008.
  If you are coming from a programming background, you might be comfortable with this book. The idea behind "experimental mathematics" is to see whether we can use a computer to provide us with intuition about mathematical results that can later be verified through rigorous proofs. Some of the observations one encounters in the process are astounding. This book tries to maintain an elementary level of treatment.
- *Mathematics by Experiment.* Jonathan M. Borwein and David H. Bailey. 2nd ed., AK Peters. 2008.

This is a more advanced book coauthored by one of the authors of the previous entry on much the same topic.

- *A Mathematician's Lament: How School Cheats Us Out of Our Most Fascinating and Imaginative Art Form.* Paul Lockhart. Bellevue Literary Press. 2009.
  This is not a math book at all: instead it is a short essay by a mathematician (or math teacher) on *what* mathematics is and *why* and *how* it should be taught. The author's philosophy is similar to the one I've tried to present in the observations toward the end of this appendix. Read it and weep. (Then go change the world.) Versions are also available on the Web (for example, check *http://www.maa.org/devlin/devlin_03_08.html*).

APPENDIX C

Working with Data

ONE OF THE UNCOMFORTABLE (AND EASILY OVERLOOKED) TRUTHS OF WORKING WITH DATA IS THAT USUALLY only a small fraction of the time is spent on the actual "analysis." Often a far greater amount of time and effort is expended on a variety of tasks that may appear "menial" by comparison but that are absolutely critical nevertheless: obtaining the data; verifying, cleaning and possibly reformatting it; and dealing with updates, storage, and archiving. For someone new to working with data (and even, periodically, for someone not so new), it typically comes as a surprise that these preparatory tasks are not only necessary but also take up as much time as they do.

By their nature, these housekeeping and auxiliary tasks tend to be very specific: specific to the data, specific to the environment, and specific to the particular question being investigated. This implies that there is little that can be said about them in generality—it pretty much all comes down to ad hoc hackery. Of course, this absence of recognizable nontrivial techniques is one of the main reasons these activities receive as little attention as they do.

That being said, we can try to increase our awareness of such issues typically arising in practical situations.

Sources for Data

The two most common sources for data in an enterprise environment are *databases* and *logfiles*. As data sources, the two sources tend to address different needs. Databases will contain data related to the "business," whereas logfiles are a source for "operational" data: databases answer the question "what did we sell to whom?" whereas logfiles answer the question "what did we do, and when?"

Databases can be either "online transaction processing" (OLTP) or "production" databases, or "data warehouses" for long-term storage. Production databases tend to be normalized, fast, and busy. You may or may not be able to get read access to them for ad hoc queries, depending on company policy. Data warehouses tend to be denormalized, slow, and often accessed through a batch processing facility (submit your query tonight and find out tomorrow that you omitted a field you needed). Production databases tend to be owned (at least in spirit) by the application development teams. Data warehouses are invariably owned by the IT department, which implies a different culture (see also the discussion in Chapter 17). In either form, databases tend to provide a stable foundation for data needs—provided you are interested in something the company already considers part of its "business."

In contrast, logfiles are often an important source of data for new initiatives. If you want to evaluate a new business idea, chances are that the data required for your analysis will not be available in the database—not *yet*, since there has never been a reason to store it before. In such situations you may still be able to find the information you need in logfiles that are regularly produced.

One *very* important distinction is that databases and logfiles have different life cycles: making changes to the design of a database is always a slow (often, excruciatingly slow) process, but the data itself lives in the database forever (if the database is properly designed). In contrast, logfiles often contain much more information than the database, but they are usually deleted very quickly. If your organization keeps logfiles for two weeks, consider yourself lucky!

Therefore, if you want to begin a project using data contained in logfiles then you need to move *fast*: start saving all files to your desktop or another safe location immediately, *then* figure out what you want to do with them! Frequently, you will need several weeks' (or months') worth of data for a conclusive analysis, and every day that you wait can never be made up. Also keep in mind that logfiles are usually generated on production servers to which access may be heavily restricted. It is not uncommon to spend *weeks* in negotiations with network administrators if you need to move significant amounts of data off of production systems.

The same consideration applies if information is not available in the logfiles, so that existing code needs to be instrumented to support collection of the required data. In this situation, you will likely find yourself captive to preexisting release schedules and other constraints. Again: start to think about *collecting* data early.

Because databases and logfiles are so common and so directly useful sources of data in an enterprise environment, it's easy to forget that they're not the only available sources.

A separate data source that sometimes can be extremely useful is the company's finance department. Companies are required to report on various financial metrics, which means that such information *must* be available, although possibly only in a highly aggregated form (*e.g.*, quarterly) and possibly quite late. On other hand, this information is normative

and therefore reliable: after all, it's what the company is paying taxes on! (I am ignoring the possibility that the data provided by the finance department might be *wrong*, but don't get me wrong: forensic data analysis is also an interesting field of study.)

What works internally may also work with competitors. The quarterly filings that publicly listed companies are required to make can make interesting reading!

So far we have assumed that you had to find and extract the data you need from whatever sources are available; in my experience, this is by far the most common scenario. However, your data may also be handed to you—for example, if it is experimental data or if it comes from an external source. In this case, it may come in a domain-specific file format (we'll return to data formats shortly). The problem with this situation is, of course, that now you have no control over what is in the data!

Cleaning and Conditioning

Raw data, whether it was obtained from a database query or by parsing a logfile, typically needs to be cleaned or conditioned. Here are some areas that often need attention.

Missing values
: If individual attributes or entire data points are missing, we need to decide how to handle them. Should we discard the whole record, mark the information in question as missing, or backfill it in some way? Your choice will depend strongly on your specific situation and goals.

Outliers
: In general, you should be extremely careful when removing outliers—you may be removing the effect that you are looking for. *Never* should data points be removed silently. (There is a (partly apocryphal) story* that the discovery of the hole in the ozone layer over Antarctica was delayed by several years because the automated data gathering system discarded readings that it considered to be "impossibly low.")

Junk
: Data that comes over a network may contain nonprintable characters or similar junk. Such data is not only useless but can also seriously confuse downstream applications that are attempting to process the data (*e.g.*, when nonprintable characters are interpreted as control characters—many programming environments will not issue helpful diagnostics if this happens). This kind of problem frequently goes unnoticed, because such junk is typically rare and not easily noticed simply by scanning the beginning of a data set.

Formatting and normalizing
: Individual values may not be formatted in the most useful way for subsequent analysis. Examples of frequently used transformations for this purpose include: forcing upper- or

* *http://www.nas.nasa.gov/About/Education/Ozone/history.html*.

lowercase; removing blanks within strings, or replacing them with dashes; replacing timestamps with Unix Epoch seconds, the Julian day number, or a similar numerical value; replacing numeric codes with string labels, or vice versa; and so on.

Duplicate records
: Data sets often contain duplicate records that need to be recognized and removed ("de-duped"). Depending on what you consider "duplicate," this may require a nontrivial effort. (I once worked on a project that tried to recognize misspelled postal addresses and assign them to the correctly spelled one. This also is a form of de-duping.)

Merging data sets
: The need to merge data sets from different sources is arises pretty often—for instance, when the data comes from different database instances. Make sure the data is truly compatible, especially if the database instances are geographically dispersed. Differing time zones are a common trouble spot, but don't overlook things like monetary units. In addition, you may need to be aware of localization issues, such as font encodings and date formatting.*

Reading this list, you should realize that the process of *cleaning* data cannot be separated from *analyzing* it. For instance: outlier detection and evaluation require some pretty deep analysis to be reliable. On the other hand, you may need to remove outliers before you can calculate meaningful values for certain summary statistics. This is an important insight, which we will make time and again: data analysis is an *iterative* process, in which each operation is at the same time the result of a previous step and the preparation for a subsequent step.

Data files may also be defective in ways that only become apparent when subsequent analysis fails or produces nonsensical results. Some common problems are:

Clerical errors
: These are basically data entry errors: 0.01 instead of 0.001, values entered in the wrong column, all that. Because most data these days is computer generated, the classic occasional typo seems to be mostly a thing of the past. But watch out for its industrial counterpart: entire data sets that are systematically corrupted. (Once, we didn't realize that a certain string field in the database was of fixed width. As we went from entries of the form `ID1`, `ID2`, and so on to entries like `ID10`, the last character was silently truncated by the database. It took a long time before we noticed—after all, the results we got back *looked* all right.)

*Regarding time zones, I used to be a strong proponent of keeping all date/time information in Coordinated Universal Time (UTC, "Greenwich Time"), always. However, I have since learned that this is not always appropriate: for some information, such as customer behavior, it is the *local* time that matters, not the absolute time. Nevertheless, I would prefer to store such information in two parts: timestamp in UTC *and* in addition, the local time zone of the user. (Whether we can actually determine the user's time zone accurately is a different matter.)

Numerical "special" values

Missing values in a data set may be encoded using special numerical values (such as −1 or 9999). Unless these values are filtered out, they will obviously corrupt any statistical analysis. There is less of a need for special values like this when data is kept in text files (because you can indicate missing values with a marker such as `???`), but be aware that it's still an issue when you are dealing with binary files.

Crazy business rules and overloaded database fields

Bad schema design can thoroughly wreck your analysis. A pernicious problem is overloaded database fields: fields that change their meaning depending on the values of *other* fields in the database. I remember a case where the `Quantity` field in a table contained the number of items shipped—unless it was zero—in which case it signaled a discount, a promotion, or an out-of-stock situation depending on whether an entry with the same order ID existed in the `Discounts`, `Promotions`, or `BackOrders` tables—or it contained not the number of items shipped but rather the number of multi-item packages that had been shipped (if the `IsMulti` flag was set), or it contained the ID (!) of the return order associated with this line item (if some other flag was set). What made the situation so treacherous was that running a query such as `select avg(Quantity) from ...` would produce a number that *seemed* sensible even though it was, of course, complete nonsense. What's worse, most people were unaware of this situation because the data was usually accessed only through (massive) stored procedures that took all these crazy business rules into account.

Sampling

When dealing with very large data sets, we can often simplify our lives significantly by working with a *sample* instead of the full data set—provided the sample is *representative* of the whole. And therein lies the problem.

In practice, sampling often means partitioning the data on some property of the data: picking all customers whose names begin with the letter "t," for instance, or whose customer ID ends with "0"; or using the logfile from one server only (out of 10); or all transactions that occurred today. The problem is that it can be very difficult to establish a priori whether these subpopulations are at all representative of the entire population. Determining this would require an in-depth study on the *whole* population—precisely what we wanted to avoid!

Statistical lore is full of (often quite amusing) stories about the subtle biases introduced through improper sampling. Choosing all customers whose first names end in "a" will probably introduce a bias toward female customers. Surveying children for the number of siblings will overestimate the number of children per household because it excludes households without children. A long-term study of mutual funds may report overly optimistic average returns on investment because it ignores funds that have been shut

down because of poor performance ("survivorship bias"). A trailing zero may indicate a customer record that was created long ago by the previous version of the software. The server you selected for your logfile may be the "overflow" server that comes online during peak hours only. And we haven't even mentioned the problems involved with collecting data in the first place! (A phone survey is inherently biased against those who don't have a phone or don't answer it.) Furthermore, strange biases may exist that nobody is aware of. (It is not guaranteed that the network administrators will know or understand the algorithm that the load balancer uses to assign transactions to servers, particularly if the load balancer itself is "smart" and changes its logic based on traffic patterns.)

A relatively safe way to create a sample is to take the whole data set (or as large a chunk of it as possible) and randomly pick some of the records. The keyword is *randomly*: don't take every tenth record; instead, evaluate each record and retain it with a probability of 1/10. Also make sure that the data set does not contain duplicates. (For instance, to sample customers given their purchases, you must first extract the customer IDs and de-dupe them, then sample from the de-duped IDs. Sampling from the transactions alone will introduce a bias toward repeat customers.)

Sampling in this way pretty much requires that the data be available as a file. In contrast, sampling from a database is more difficult because, in general, we don't have control (or even full understanding) over how records are sorted internally. We can dump all records to file and then sample from there, but this is rather awkward and may not even be feasible for very large tables.

A good trick to enable random sampling from databases is to include an additional column, which *at the time the record is created* is filled with a random integer between (say) 0 and 99. By selecting on this column, we can extract a sample consisting of 1 percent of all records. This column can even be indexed (although the database engine may ignore the index if the result set is too large). Even when it is not possible to add such a column to the actual table, the same technique can still be used by adding a cross-reference table that contains only the primary key of the table we want to sample from and the random integer. It is critical that the the random number is assigned at the time the record is created and is never changed or updated thereafter.

Whichever approach you take, you should verify that your sampling process does lead to representative samples. (Take two independent samples and compare their properties.)

Sampling can be truly useful—even necessary. Just be very careful.

Data File Formats

When it comes to file formats for data, my recommendation is to keep it simple, even dead-simple. The simpler the file format, the greater flexibility you have in terms of the tools you can use on the data. Avoid formats that require a nontrivial parser!

My personal favorite is that old standby, the delimiter-separated text file, with one record per line and a single data set per file. (Despite the infamous difficulties with the Unix `make` utility, I nevertheless like tab-delimited files: since numbers don't contain tabs, I never need to quote or escape anything; and the tabs make it easy to visually inspect a file—easier than do commas.) In fairness, delimiter-separated text files do not work well for one-to-many relationships or other situations where each record can have a varying number of attributes. On the other hand, such situations are rare and tend to require special treatment, anyway.

One disadvantage of this format is that it does not allow you to keep information about the data ("metadata") within the file itself, except possibly the column names as first row. One solution is to use two files—one for the data and one for the metadata—and to adopt a convenient naming convention (*e.g.*, using the same basename for both files while distinguishing them by the extensions `.data` and `.names`).*

In general, I strongly recommend that you stay with text files and avoid binary files. Text files are portable (despite the annoying newline issue), robust, and self-explanatory. They also compress nicely. If you nevertheless decide to use binary files, I suggest that you use an established format (for which mature libraries exist!) instead of devising an ad hoc format of your own.

I also don't find XML very suitable as a file format for data: the ratio of markup to payload is poor which leads to unnecessarily bloated files. XML is also notoriously expensive to parse, in particular for large files. Finally, the flexibility provided by XML is rarely necessary for data sets, which typically have a very regular structure. (It may seem as if XML might be useful for metadata, but even here I disagree: the value of XML is to make data machine-readable, whereas the primary consumers of metadata are humans!)

Everything I have said so far assumes that the data files are primarily for yourself (you don't want to distribute them) and that you are willing to read in the entire file sequentially (so that you don't need to perform seeks within the file). There are file formats that allow you to bundle multiple data sets into a single file and efficiently extract parts of them (for example, check out the Hierarchical Data Format (HDF) and its variants, such as netCDF), but I have never encountered them in real life. It should not be lost on you that the statistics and machine-learning communities use delimiter-separated text almost exclusively as format for data sets on their public data repositories. (And if you need indexed lookup, you may be better off setting up a minimal standalone database for yourself: see the Workshop in Chapter 16.)

Finally, I should point out that some (scientific) disciplines have their own specialized file formats as well as the tools designed to handle them. Use them when appropriate.

*This convention is used by many data sets available from the UCI Machine Learning Repository.

The Care and Feeding of Your Data Zoo

If you work in the same environment for a while, you are likely to develop a veritable collection of different data sets. Not infrequently, it is this ready access to relevant data sets that makes you valuable to the organization (quite aside from your more celebrated skills). On the downside, *maintaining* that collection in good order requires a certain amount of effort.

My primary advice is make sure that all data sets are *self-explanatory* and *reproducible*.

To ensure that a data set is self-explanatory, you should not only include the minimal metadata with or in the file itself, but include *all* the information necessary to make sense of it. For instance, to represent a time series (*i.e.*, a data set of measurements taken over time at regular intervals), it is strictly necessary to store only the values, the starting time, and the length of the interval between data points. However, it is safer to store the corresponding timestamp with each measured value—this way, the data set still makes sense even if the metadata has been lost or garbled. Similar considerations apply more generally: I tend to be fairly generous when it comes to including information that might seem "redundant."

To keep data reproducible, you should keep track of its source *and* the cleaning and conditioning transformations. This can be tedious because so much of the latter consists of ad hoc, manual operations. I usually keep logs with my data sets to record the URLs (if the data came from the Web) or the database queries. I also capture the commands and pipelines issued at the shell prompt and keep copies of all transformation scripts. Finally, if I combine data from multiple sources into a single data set, I always retain the original data sets.

This kind of housekeeping is very important: not only to produce an audit trail (should it ever be needed) but also because data sets tend to be reused again and again and for different purposes. Being able to determine *exactly* what is in the data is crucial.

I have not found many opportunities to automate these processes; the tasks just vary too much. The one exception is the automated scheduled collection and archiving of volatile data (*e.g.*, copying logfiles to a safe location). Your needs may be different.

Finally, here are three pieces of advice on the physical handling of data files. They should be obvious but aren't necessarily.

Keep data files readily available
: Being able to run a minimal script on a file residing on a local drive to come up with an answer in seconds (compared to the 12–24 hour turnaround typical of may data warehouse installations) is a huge enabler.

Compress your data files
: I remember a group of statisticians who constantly complained about the lack of disk space and kept requesting more storage. None of them used compression or had even

heard of it. And all their data sets were kept in a textlike format that could be compressed by 90 percent! (Also keep in mind that `gzip` can read from and write to a pipe, so that the uncompressed file never needs to exist on disk.)

Have a backup strategy
: This is important especially if all of your data resides only on your local workstation. At the very least, get a second drive and mirror files to it. Of course, a remote (and, ideally, managed) storage location is much better. Keep in mind that data sets can easily become large, so you might want to sit down with your network administrators early in the process so that your storage needs can be budgeted appropriately.

Skills

I hope that I've convinced you that obtaining, preparing, and transforming data makes up a large part of day-to-day activities when working with data. To be effective in this role, I recommend you acquire and develop some skills that facilitate these aspects of your role.

For the most part, these skills come down to easy, ad hoc programming. If you come from software development, you will hardly find anything new here. But if you come from a scientific (or academic) background, you might want to broaden your expertise a little.

A special consideration is due to those who come to "data analysis" from a database-centric, SQL programming point of view. If this describes your situation, I *strongly* encourage you to pick up a language besides SQL. SQL is simply too restricted in what it can do and therefore limits the kinds of problems you will choose to tackle—whether you realize it or not! It's also a good idea to do the majority of your work "offline" so that there is less of a toll on the database (which is, after all, usually a shared resource).

Learn a scripting language
: A scripting language such as Perl, Python, or Ruby is required for easy manipulation of data files. Knowledge of a "large-scale" programming language like C/C++/Java/C# is *not* sufficient. Scripting languages eliminate the overhead ("boilerplate code") typically associated with common tasks such as input/output and file or string handling. This is important because most data transformation tasks are tiny and therefore the typical cost of overhead, relative to the overall programming task, is simply not acceptable.

  Note that R (the statistics package) can do double duty as a scripting language for these purposes.

Master regular expressions
: If you are dealing with strings (or stringlike objects, such as timestamps), then regular expressions are the solution (and an amazingly powerful solution) to problems you didn't even realize you had! You don't need to develop intimate familiarity with the whole regular expression bestiary, but working knowledge of the basics is required.

Be comfortable browsing a database

Pick a graphical database frontend* and become proficient with it. You should be able to figure out the schema of a database and the semantics of the data simply by browsing the tables and their values, requiring only minimal help.

Develop a good relationship with your system administrator and DBA

System administrators and DBAs are in the position to make your life significantly easier (by granting you access, creating accounts, saving files, providing storage, running jobs for you, ...). However, they were not hired to do that—to the contrary, they are paid to "keep the trains on time." A rogue (and possibly clueless or oblivious) data analyst, running huge batch jobs during the busiest time of the day, does *not* help with that task!

I would like to encourage you to take an interest in the situation of your system administrators: try to understand their position and the constraints they have to work under. System administrators tend to be paranoid—that's what they're paid for! Their biggest fear is that *something* will upset the system. If you can convince them that you do not pose a great risk, you will probably find them to be incredibly helpful.

(Finally, I tend to adopt the attitude that any production job by default has higher priority than the research and analysis I am working on, and therefore I better be patient.)

Work on Unix

I mean it. Unix was developed for *precisely* this kind of ad hoc programming with files and data, and it continues to provide the most liberating environment for such work.

Unix (and its variants, including Linux and Mac OS X) has some obvious technical advantages, but its most important property in the present context is that it *encourages you to devise solutions*. It does not try (or pretend) to do the job for you, but it goes out of its way to give you tools that you might find handy—without prescribing how or for what you use them. In contrast, other operating systems tend to encourage you to stay within the boundaries of certain familiar activity patterns—which does *not* encourage the development of your problem-solving abilities (or, more importantly, your problem-solving *attitudes*).

True story: I needed to send a file containing several millions of keys to a coworker. (The company did not work on Unix.) Since the file was too large to fit safely into an email message, I posted it to a web server on my desktop and sent my coworker the link. (I dutifully had provided the file with the extension `.txt`, so that he would be able to open it.) Five minutes later, he calls me back: "I can't open that"—"What do you mean?"—"Well, I click the link, but ScrapPaper [the default text editor for small text

*The SQuirreL project (*http://squirrel-sql.sourceforge.net*) is a good choice. Free, open source, and mature, it is also written in Java—which means that it can run anywhere and connect to any database for which JDBC drivers exist.

files on this particular system] dies because the file is too big." This coworker was not inept (in fact, he was quite good at his primary job), but he displayed the particular non-problem-solving attitude that develops in predefined work environments: "link, click." It did not even occur to him to think of something else to try. That's a problem!

If you want to be successful working with data, you want to work in an environment that encourages you to devise your own solutions.

You want to work on Unix.

Terminology

When working with data, there is some terminology that is frequently used.

Types of Data

We can distinguish different types of data. The most important distinction is the one between *numerical* and nonnumerical or *categorical* data.

Numerical data is the most convenient to handle because it allows us to perform arbitrary calculations. (In other words, we can calculate quantities like the mean.) Numerical data can be *continuous* (taking on all values) or *discrete* (taking on only a discrete set of values). It is often necessary to discretize or *bin* continuous data.

You will sometimes find numerical data subdivided further into *interval* and *ratio* data. Interval data is data that does not have a proper origin, whereas ratio data does. Examples of interval data (without proper origin) are calendar dates and temperatures in units of Fahrenheit or Celsius. You can subtract such data to form *intervals* (there are 7 days between 01 April 09 and 07 April 09) but you cannot form ratios: it does not make sense to say that 60 Celsius is "twice as hot" as 30 Celsius. In contrast, quantities like length or weight measurements are ratio data: 0 kilograms truly means "no mass," and 0 centimeters truly means "no length." For ratio data, it makes sense to say that a mass of 2 kilograms is "twice as heavy" as a mass of 1 kilogram.

The distinction between ratio and interval data is not very important in practice, because interval data occurs rarely (I can think of no examples other than the two just mentioned) and can always be avoided through better encoding. The data is numeric by construction, so a zero must exist; hence an encoding can be found that measures magnitudes from this origin (the Kelvin scale for temperatures does exactly that).

All nonnumerical data is categorical—in practice, you will usually find categorical data encoded as strings. Categorical data is less powerful than numerical data because there are fewer things we can do with it. Pretty much the only available operation is counting how often each value occurs.

Categorical data can be subdivided into *nominal* and *ordinal* data. The difference is that for ordinal data, a natural sort order between values exists, whereas for nominal data no such

sort order exists. An example for ordinal (sortable) data is a data set consisting of values like `Like`, `Dislike`, `Don't Care`, which have a clear sort order (namely, `Like` > `Don't Care` > `Dislike`). In contrast, the colors `Red`, `Blue`, `Green` when used to describe (say) a sweater are nominal, because there is no natural order in which to arrange these values.

Sortability is an important property because it implies that the data is "almost" numerical. If categorical data is sortable then it can be mapped to a set of numbers, which are more convenient to handle. For example, we can map `Like`, `Dislike`, `Don't Care` to the numbers 1, −1, and 0, which allows us to calculate an average value after all! However, there is no such thing as the "average color" of all sweaters that were sold.

Another property I look for determines whether data is "mixable." Can I combine arbitrary multiples of data points to construct a new data point? For data to be mixable in this way, it is not enough to be able to *combine* data points (*e.g.*, concatenating two strings) I must also be able to combine *arbitrary* multiples of all data points. If I can do this, then I can construct a *new* data point that lies, for example, "halfway" between the original ones, like so: $x/2 + y/2$. Being able to construct new data points in this way can speed up certain algorithms (see Chapter 13 for some applications).

When data is mixable it is similar to points in space, and a lot of geometric intuition can be brought to bear. (Technically, the data forms a vector space over the real numbers.)

The Data Type Depends on the Semantics

It is extremely important to realize that *the type of the data is determined by the semantics of the data*. The data type is *not* inherent in the data—it only arises from its *context*.

Postal codes are a good example: although a postal code like `98101` may *look* like a number, it does not *behave* like a number. It just does not make sense to add two postal codes together or to form the average of a bunch of postal codes! Similarly, the colors `Red`, `Yellow`, `Green` may be either nominal (if they refer to the colors of a sweater) or ordinal (if they are status indicators, in which case they obey a sort order akin to that of a traffic light).

Whether data is numerical or categorical, sortable or not, depends on its meaning. You can't just look at a data set in isolation to determine its type. You need to know what the data *means*.

Data by itself does not provide information. It is only when we take the data *together* with its context that defines its semantics that data becomes meaningful. (This point is occasionally overlooked by people with an overly formalistic disposition.)

Types of Data Sets

Data sets can be classified by the number of variables or columns they contain. Depending on the type of data set, we tend to be interested in different questions.

Univariate
: A data set containing values only for a single variable. The weights of all students in a class, for example, form a univariate data set. For univariate data sets, we usually want to know how the individual points are distributed: the shape of the distribution, whether it is symmetric, does it have outliers, and so on.

Bivariate
: A data set containing two variables. For such data sets, we are mostly interested in determining whether there is a relationship between the two quantities. If we had the heights in addition to the weights, for instance, we would ask whether there is any discernible relationship between heights and weights (*e.g.*, are taller students heavier?).

Multivariate
: If a data set contains more than two variables, then it is considered multivariate. When dealing with multivariate problems, we typically want to find a smaller group of variables that still contains most of the information about the data set.

Of course, any bivariate or multivariate data set can be *treated* as a univariate one if we consider a single variable at a time. Again, the nature of the data set is not inherent in the data but depends on how we look at it.

Further Reading

- *Problem Solving: A Statistician's Guide.* Chris Chatfield. 2nd ed., Chapman & Hall/CRC. 1995.
  This is a highly informative book about all the messy realities that are usually *not* mentioned in class: from botched experimental setups to effective communication with the public. The book is geared toward professional statisticians, and some of the technical discussion may be too advanced, but it is worthwhile for the practicality of its general advice nonetheless.
- *Unix Power Tools.* Shelley Powers, Jerry Peek, Tim O'Reilly, and Mike Loukides. 3rd ed., O'Reilly. 2002.
  The classic book on getting stuff done with Unix.
- *The Art of UNIX Programming.* Eric S. Raymond. Addison-Wesley. 2003.
  The Unix philosophy has been expounded many times before but rarely more eloquently. This is a partisan book, and one need not agree with every argument the author makes, but some of his observations on good design and desirable features in a programming environment are well worth contemplating.

Data Set Repositories

Although I assume that you have your own data sets that you would like to analyze, it's nice to have access to a wider selection of data sets—for instance, when you want to try out and learn a new method.

Several data set repositories exist on the Web. These are the ones that I have found particularly helpful.

- *The Data and Story Library at statlib.* A smaller collection of data sets, together with their motivating "stories," intended for courses in introductory statistics. (*http://lib stat.cmu.edu/DASL*)
- *Data Archive at the Journal of Statistics Education.* A large collection of often uncommonly interesting data sets. In addition to the data sets, the site provides links to the full text of the articles in which these data sets were analyzed and discussed. (*http://www.amstat.org/publications/jse*—then select "Data Archive" in the navigation bar)
- *UCI Machine Learning Repository.* A large collection of data sets, mostly suitable for classification tasks. (*http://archive.ics.uci.edu/ml/*)
- *Time Series Data Library.* An extensive collection of times series data. Unfortunately, many of the data sets are poorly documented. (*http://robjhyndman.com/TSDL/*)
- *Frequent Itemset Mining Dataset Repository.* A specialized repository with data sets for methods to find frequent item sets. (*http://fimi.cs.helsinki.fi/data/*)
- *UCINET IV Datasets.* Another specialized collection: this one includes data sets with information about social networks. (*http://vlado.fmf.uni-lj.si/pub/networks/data/Ucinet/UciData.htm*)
- *A Handbook of Small Data Sets.* David J. Hand, Fergus Daly, K. McConway, D. Lunn, and E. Ostrowski. Chapman & Hall/CRC. 1993.
  This is a rather curious resource: a book containing over 500 individual data sets (with descriptions) from all walks of life. Most of the data sets are "small," containing from a handful to a few hundred points. The data sets themselves can be found all over the Web, but only the book gives you the descriptions as well.