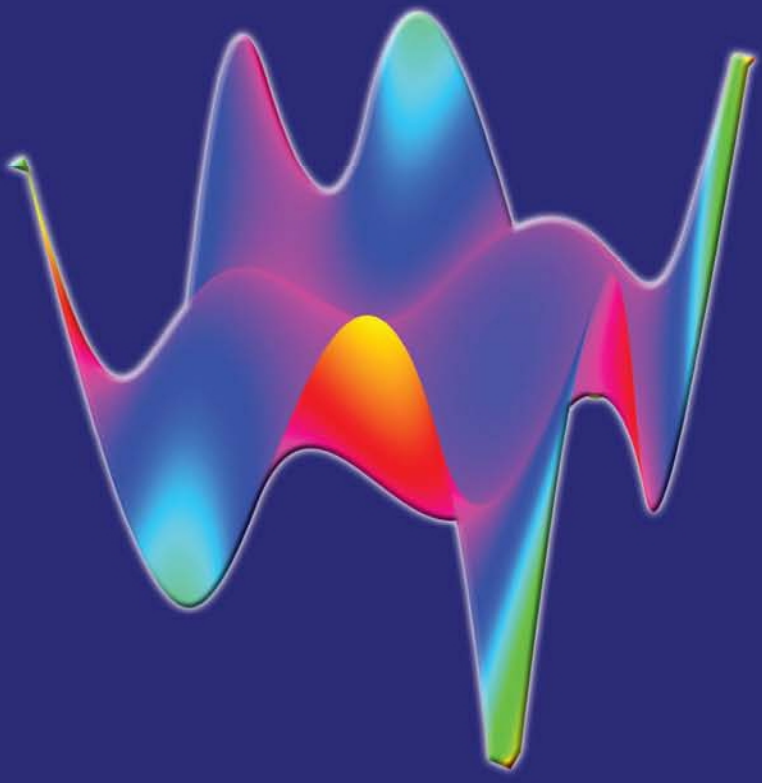
A Beginner's Guide to Mathematica



David McMahon
Daniel M. Topa

Chapman & Hall/CRC
Taylor & Francis Group

A Beginner's Guide to Mathematica

A Beginner's Guide to Mathematica

David McMahon
Daniel M. Topa

Chapman & Hall/CRC
Taylor & Francis Group
Boca Raton London New York

Cover image: This polynomial is an extension of the Legendre polynomial defined on a square instead of a one-dimensional interval. The color scheme is determined by the value of the Laplacian of the polynomial.

Published in 2006 by
Chapman & Hall/CRC
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

© 2006 by Taylor & Francis Group, LLC
Chapman & Hall/CRC is an imprint of Taylor & Francis Group

No claim to original U.S. Government works
Printed in the United States of America on acid-free paper
10 9 8 7 6 5 4 3 2 1

International Standard Book Number-10: 1-58488-467-3 (Softcover)
International Standard Book Number-13: 978-1-58488-467-5 (Softcover)
Library of Congress Card Number 2005051949

This book contains information obtained from authentic and highly regarded sources. Reprinted material is quoted with permission, and sources are indicated. A wide variety of references are listed. Reasonable efforts have been made to publish reliable data and information, but the author and the publisher cannot assume responsibility for the validity of all materials or for the consequences of their use.

No part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, please access www.copyright.com (http://www.copyright.com/) or contact the Copyright Clearance Center, Inc. (CCC) 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. CCC is a not-for-profit organization that provides licenses and registration for a variety of users. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

Trademark Notice: Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

Library of Congress Cataloging-in-Publication Data

McMahon, David.
A beginners guide to Mathematica / David McMahon, Dan Topa.
p. cm.
Includes bibliographical references and index.
ISBN 1-58488-467-3 (alk. paper)
1. Mathematica (Computer file) 2. Mathematics--Data processing. I. Topa, Dan. II. Title.

QA76.95.M44 2005
510'.285'536--dc22 2005051949

informa
Taylor & Francis Group
is the Academic Division of Informa plc.

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

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

Contents

Preface **vii**

Chapter 1
Introduction and survey **1**
1.1 Why *Mathematica*? 4
1.2 Notebooks 5
1.3 Entering data 17
1.4 Data structures 22
1.5 Programming 53
1.6 Standard add-on packages 65
1.7 Miscellaneous packages 67
1.8 Palettes 78
1.9 Other resources 91
1.10 In conclusion 96

Chapter 2
Computation examples **99**
2.1 The quadratic equation 99
2.2 Singular matrices and inversion 118
2.3 Linear regression 128
2.4 An inverse problem 180

Chapter 3
Graphics examples **209**
3.1 Graphics primitives 214
3.2 Plotting in two dimensions 263
3.3 Pictionary of 2D graphic types 280
3.4 Plotting in three dimensions 281
3.5 Rotation through parity states 299

Chapter 4
Ordinary differential equations **303**
4.1 Defining, entering and solving differential equations 303

Chapter 5
Transforms **369**
5.1 Properties of linear integral transforms 370
5.2 The Laplace transform 370
5.3 The Fourier transform 398
5.4 The z-transform 417

Chapter 6
Integration **423**
6.1 Basic integrals: polynomials and rational functions 424
6.2 Multivariate expressions 430
6.3 Definite integration 435
6.4 Integrals involving the Dirac delta function 438
6.5 Using the Integrate command 440
6.6 Monte Carlo integration 442

Chapter 7
Special functions **451**
7.1 The Gamma function 451
7.2 The Bessel functions 465
7.3 The Riemann zeta function 478
7.4 Working with Legendre and other polynomials 487
7.5 Spherical harmonics 495

Appendix 1 **497**

Appendix 2 **599**

References **693**

Index **695**

Preface

Mathematica is an incredibly powerful and useful environment for symbolic computing, numerical computing and data visualization. To us, it is the flagship product that gave the personal computing revolution its biggest boost. Some may counter that spreadsheet programs or databases were more significant. For sure these products have great relevance. But in those cases the application migrated from mainframe computers to personal computers (PCs) and hence became more accessible. Considering computing languages like FORTRAN we were so vexed by the low compiler qualities for the PCs that we found them of little utility. *Mathematica*, on the other hand, gave people a whole new power that had heretofore been unrealized. It extended capabilities far beyond mainframes. Certainly other applications attempted to claim the scientific computing market such as Theorist and Math Cad. But our experiences with them made us feel we could do the symbolic computations and integrations faster without the computer.

Admittedly, in prerelease 3.0 the user interface was not very appealing. Yet it was impossible to argue that you were working with a brand new set of capabilities. The sophistication of the symbolic capabilities has gone unmatched for decades. We can think of no truer testament to the breakthrough that *Mathematica* represents than this. There have been a few feigned attempts at duplication, but nothing has come very close. The architecture of *Mathematica* was ahead of its time. When you look at the transition from FORTRAN 77 to C you see a true modernization of the computing language. After mastering C, one looks at FORTRAN as a bit confining. For instance, C was capable of making a list of arbitrarily sized lists and FORTRAN was not. *Mathematica* came and delivered the best part of FORTRAN — the rigorous, robust computation — with the wonderful, modern features of C. On top of that we got stun-

ning symbolic capabilities and superlative graphical capabilities. The only significant downside we are faced with is that unlike IDL, we cannot compile our *Mathematica* code into machine language files. However, it would be irresponsible to characterize such a miraculous tool as incomplete. We think even critics of *Mathematica* would have to agree either that it is the gold standard for PC software or that it has at least greatly elevated the standards. Its customers are served very well, the releases are virtually bug-free, the documentation is excellent and available on paper too! The online help is state-of-the-art and the Wolfram website is a door to a vast amount of examples and tutorials.

One particularly frustrating problem we have with other PC applications is how often the paradigms shift. New releases mean legacy code is now broken. This really frustrates someone who basked in the interrelease years between FORTRAN 77 and FORTRAN 90. Over the years one thing that has impressed us with *Mathematica* is the stability of the design. To be sure, a program with such far-reaching scope and power will have changes, but they are so few as to border on the inconsequential. Perhaps because of a brilliant design, or perhaps because of good stewardship, *Mathematica* has been a friendly monolith over the years.

One goal of this volume is to start new users on a voyage of discovery and take previous users a bit farther along. The subject matter is so rich that it cannot be done justice in a single volume. We have tried to sift through a lot of the material and to introduce you to those items that are the most useful to practical applications.

To be sure, we will document cases of bugs, but these bugs are mostly in the front end. Wolfram Research seems to guard its computational kernel most jealously. We suppose that because there are so few bugs it is easy to delineate them. Given their appearance we feel some discussion is warranted because they could easily bedevil users. We hope that this volume — accompanied by the copious Wolfram material — will help you see how helpful *Mathematica* can be in everyday use.

Another goal of this book is to help you learn the aspects of *Mathematica* needed for many practical applications. The publisher has chartered us to come up with a distinct approach that separates this book from so many other good ones. So we place a high value on notebook and file system organization, cross-platform capabilities, data reading and writing. We also want to help you write notebooks that are easy to read and to debug. We work with new users, and a great deal of their trouble comes from lack of discipline. *Mathematica* has a wonderful flexibility; it is not a rigid program. Some users view the absence of rigidity as a problem. They seem to yearn for single path to master. We want to show that there are often multiple ways to solve a problem and you should choose the one most comfortable for you. The publisher has tasked us to include the error messages users are likely to encounter and to assist in their interpretation.

We have been working with numerous associates over the years helping them to understand and to learn *Mathematica.* In some sense this book documents what we have learned from our talented colleagues as they endeavoured to conquer *Mathematica*. Perhaps our deep appreciation for *Mathematica* stems from our reliance upon it. We are certain that this application can be a boon to most people.

Some of our peers have been quite vocal and challenged us to write a book that will help them solve significant problems in their laboratories and universities. The attempt to do so has been quite daunting. One complaint we heard over and over is that many books help you solve problems that they don't need *Mathematica* for. They can look at a wave equation and tell you what the answer is; they don't need *Mathematica* for that. We wanted a readable book and we felt strongly that to be useful we would need to keep it brief. We didn't want to present a compendium of arcane problems and esoteric solutions. But we did want to present a hierarchy. We want to start with problems that are simple and present multiple solution paths. Solutions ranging from basic to elegant to gradually introduce the user to the *Mathematica* toolkit. And then we wanted to introduce challenging problems and eventually cutting-edge problems.

We want to build an intuition and that takes time and repetition. Fortunately we are dealing with an extremely well-designed and documented product and ultimately the diligent will be rewarded. We also want this book to be useful to people of different skill levels. We do not pretend that you will master the application after one reading. We count on readers to revisit sections of interest and sections that were challenging, so that they can continue to refine their skills.

To write this book we used *Mathematica* 5.1 for the Macintosh and for the PC. We feel quite strongly that the cross-platform capabilities of *Mathematica* enhance its value. In general if you are using 5.0 or even 4.x, we would expect that almost all of the code fragments presented here will run in your environment.

Finally, the publisher will set up a website where you can download the code fragments you see in this book. We encourage you to download these scripts and run them on your own machine to experiment and to reinforce the lessons. Also, you will be able to see the figures in full color. The URL is http://www.crcpress.com/.

C H A P T E R 1

Introduction and survey

We use *Mathematica* every day and have helped many other users to learn *Mathematica*. We have listened to advanced users and heard their comments and protestations. This book is a product that we have crafted to address the issues that we have frequently confronted. As we mentioned in the preface, *Mathematica* is too broad to conquer quickly. But you can rapidly learn what you need to become functional and to begin exploring.

We have written a book that will take multiple readings. We have tried to present tiered solutions: solutions you can grasp immediately, solutions that require a bit more experience but simplify the problem, and elegant solutions showing the core of *Mathematica*'s power. On first exposure one should be content to glean the rudiments. As you revisit the material it will become clear, and hopefully you can progress to the more powerful *Mathematica* structures.

We start out with familiar mathematics to reduce confusion. We show different ways to solve the same problem. Hopefully as you reread sections, you will become more familiar with the more sophisticated formulations. We also show common user mistakes and error messages. The point of this volume is not to show that the authors can solve problems with *Mathematica*, it is instead to show the reader that he can solve problems with *Mathematica* and a little help.

1.0.1 Augmenting Wolfram material

Before starting with this book, you need to take a few minutes to go through the *Mathematica* tutorials we list here and to explore the on-line help (available though the F1 key on Windows platforms or *help* on the Macintosh).

First, the later versions of *Mathematica* (4.2 and beyond) come with quick tutorials which are of high quality and will help quickly introduce you to features and capabilities of *Mathematica.* In version 5 and later, the tutorial is accessed through the menu bar under the Help command. We cannot think of a better introduction than this tutorial. You can see the commands and output together. You can copy the commands from the tutorial into your own notebook and experiment with them. At the end of the tutorial you are presented with a host of hyperlinks to take you to topics for further scrutiny. We encourage users to view the tutorial and we will assume that you have done so before reading this book. We intend to elaborate topics outside of the tutorial.

Second, you can use the vast resources at the Wolfram web portal www.wolfram.com. You can find all manner of materials to help you master *Mathematica:* presentations, papers, notebooks, etc. The sophistication ranges from elementary to advanced. In the last few years Wolfram Research has put a great deal of effort into their Internet sites and you should periodically check them. Three superb Internet tutorials for new users can be found at:

http://documents.wolfram.com/v5/Tour/

http://documents.wolfram.com/v5/TheMathematicaBook/APracticalIntroductionToMathematica/

http://www.wolfram.com/products/mathematica/tour/

These materials should be appreciated for their brevity, their accuracy, and their utility. The goal of this book is to expand beyond these tutorials. In other words, we are operating under the assumption that you have reviewed the on-line tutorial and the Internet tutorials.

Mathematica includes a book called *Getting Started*. It is a superlative way to introduce users to *Mathematica*. If you can find this book, read it first and then turn to the Help Browser and Internet materials.

The on-line help is superlative and you should make use of it early and often. You will find a clear and succinct summary for each command and usually several examples. These examples are invaluable to mastering *Mathematica* for they show the basics and often very sophisticated examples. To access the on-line help, double click on a command word to highlight it and then press the F1 (or *help*) key. This will launch the Help Browser. The references are thoroughly cross-referenced and hyperlinked. For example, if you are manipulating the structure of a list and you launch the help browser on the command **Take** you see a clear and concise statement of what the command does. As you read down the bullet list, you will find hyperlinks to the relevant sections in *The Mathematica Book* (sections 1.8.4 and 2.4.2). There is a Back button to return you to this page if you wander off using the hyperlinks. You will also notice a list of commands related to this one. In this instance we find **Part**, **Drop**, **StringTake**, **Select**, **Cases**, **Partition** and **PadLeft**. Typically these commands have similar or opposite functionality.

Every user who has access to *Mathematica* has access to the on-line help, the tutorial, and (if Internet connected) the websites. Some users may have other, more expensive options. You should also consider subscribing to the *Mathematica Journal* to learn cutting-edge techniques and the latest applications. They may seem overwhelming at first, but eventually you will appreciate the beauty and power of *Mathe-*

matica. Also, you should consider upgrading your *Mathematica* subscription to the Premier package which supplies you with all the current releases and provides for telephone support from the staff at Wolfram Research.

Special characters make things easier to read.

1.0.2 Using this book

We believe very much in reinforcement. By reinforcement we mean repetition and repetition in different contexts. We do think it realistic that you can read the excellent on-line help and master *Mathematica*. We think that material and complexity should be introduced gradually and that the contexts should be varied. Most of all, we believe the reader needs to be exposed to realistic problems that show the subtleties and nuances of Mathematica. The reason we have titled this book *Practical Mathematica* is because we want to prepare the reader to use *Mathematica* on his own in the laboratory, the office or at home. This means acquainting you with common problems, errors and their messages and practical tools like data I/O. We have specifically chosen exercises that require attention to details. We want to convince the readers that they can use *Mathematica* to successfully solve problems on their own.

To reinforce the importance of the on-line help we will often refer to relevant sections for further reading. Such entries will be preceded by the *Mathematica* spikey ✵.

Also due to the fantastically rich mathematical concepts used it is invaluable to have a single reference which spans the panorama of mathematics used. The second edition of *The CRC Concise Encyclopedia of Mathematics* is a priceless reference to keep nearby. It does a good job of unifying notation amongst hundreds of sources and relies heavily upon *Mathematica* for either demonstration or comparison. The on-line help is excellent and the *Encyclopedia* augments these explanations admirably. The second edition has a much stronger tie-in with the *Mathematica*. References to *The CRC Encyclopedia of Mathematics* will be annotated with 💡.

The World Wide Web hosts a variety of useful information pertaining to *Mathematica*. Most of the information is available through the Wolfram Research website. Any references to the web will be marked with 🗺.

Finally, we note a curious silence over a few of the bugs in *Mathematica*. To be sure the functionality is quite robust, but there are a few minor glitches with the front end. However, these minor problems may be enough to completely derail the new user and as such we show how to diagnose, how to correct, and how to avoid. In this same category of advice we will include comments to guide users past some common errors new users make. Such comments will be marked with ⚠.

✵ These statements refer to entries in the Help Browser.

💡 The statements refer to entries in *The CRC Concise Encyclopedia of Mathematics*.

🗺 These statements refer to material on the World Wide Web.

⚠ These statements refer to common pitfalls of the new user as well as *Mathematica* bugs. This is information you will typically find nowhere else.

1.1 Why *Mathematica*?

A common question we ask our colleagues is why they chose a particular product. Oftentimes the answer has to do with legacy code. The organization has accumulated a great deal of coding in one particular application. The cases of interest are when the laboratory or office has no legacy code and may have users familiar with packages such as Matlab and IDL. What drove them to select the product that they did?

We appreciate that there are other valuable products out there that will serve the users admirably and we think they serve their market segment in good stead. If you say you want to produce machine language executables from your code, then IDL is a great choice. If you want a Matlab package that is exactly tailored to your job, then Matlab is an excellent choice. If you want to do problems in the Feynman calculus, relativistic tensor calculus, nonlinear differential equations, number theory or plow through the integrals in Gradshteyn and Ryzhik then *Mathematica* is an excellent choice.

At one point we too were trying to find the best application to serve our needs. We explored a variety of packages and found many, many useful features across all of the products. We would like to clear up some of the *Mathematica* myths that seem to be immortal.

1. *Mathematica* is hard to learn.

This is frustrating to hear. From what we can discern, it seems that other packages with far-reduced capabilities are "quick" to learn. But as we will show, the rudiments of *Mathematica* can be grasped in as brief a period. Of course with *Mathematica* there still remains a significant tool kit to be explored. For example, say you want to program a linear regression using *Mathematica* and application X. We will show that the basics of *Mathematica* can be conquered in a comparable period of time. Except with application X there is not much more to assimilate and with *Mathematica* you have only scratched the surface. So your choice could be to use application X until you are faced with more daunting problems and then start all over again with *Mathematica*, or you can start and grow with *Mathematica*. In summary, *Mathematica* has a comparable learning rate. Mastering the complete application will take far more time due to the extensive capabilities of this tool kit.

In this book we attempt to show a tiered approach to problem solving. We show how to use elementary techniques in a transparent fashion and then introduce refinements. We think you will see in almost all cases the more advanced features — which may seem foreign — greatly simplify coding.

2. *Mathematica* is slower than other similar applications.

This is remarkable because in many cases the opposite is true: *Mathematica* frequently has a significant speed advantage over other applications. Whenever possible *Mathematica* will do calculations in arbitrary precision by using a rational number representation and symbolic methods. Two comments: this can give you exact answers free of numerical errors — a real benefit — and it can be extremely slow. But

the other packages don't have an arbitrary precision mode, so in this sense *Mathematica* is infinitely faster. Now if you want to run *Mathematica* in a double precision mode (by using the numeric operator **N[]** or simply typing 3. in lieu of 3), then *Mathematica* runs with comparable speed and precision.

In general, we are very impressed by the careful attention Wolfram Research gives to its releases. Perhaps because of the program's enormous scope we should expect an occasional glitch. We encountered a rather severe problem when transitioning from 4.0 to 4.2. An entire class of singular value decomposition (SVD) problems would no longer evaluate. Out of thousands of notebooks this was the only problem discovered. Unfortunately, it did cause quite a dilemma for us.

We are not criticizing Wolfram Research. They usually do a superb job with upgrades. The more usual case is for an application upgrade to be tantamount to a bug swap. Old bugs are exchanged for new bugs. Also, there are annoying changes in command paradigms and syntax that require extensive modification to interface codes or documentation.

So in regard to stability, Wolfram Research provides a shining example of how to make an application grow and evolve. Perhaps we are showing our age and our fondness for the Pax FORTRANA, the 13 years between FORTRAN 77 and FORTRAN 90. We are 10 years into the second period of stability ushered in by FORTRAN 95.



If possible, do not delete old versions of *Mathematica* when you upgrade. There is a minute chance that the upgrade may contain a bug that may impact previously developed calculations.

1.2 Notebooks

At this point we consider that you have gone through the tutorials and are familiar with basic operations. After working with *Mathematica* for awhile, you will probably find yourself using multiple notebooks at once and moving notebooks between different machines, maybe even different platforms. It is always important to ensure that your notebooks execute linearly, that is, from top to bottom. Requiring commands to be executed out of order is a very poor coding practice. Also, you should ensure that your notebook can run independently and does not require unreferenced notebooks to be open. Perhaps you are familiar with using modules; they are a great way to give notebooks a clean, readable appearance. They are an example of a referenced file since they are loaded by reference. One should use modules and one must avoid unreferenced data. For example, suppose on one notebook you do a long computation for some variable *q*. This variable is needed by yet another notebook, yet there is no referencing; the second notebook uses *q* as a global variable. This is a terrible programming practice that many users run into at first. The target notebook should be able to run on its own. You can test this by launching *Mathematica* and immediately running the target notebook. If the run is successful, you have avoided unreferenced data.

1.2.1 A seed notebook

To help us with these housekeeping details, we maintain a read-only notebook called seed.nb. As the name implies this is a file that all notebooks will grow from. To create a new notebook we open **seed.nb** and save it under a different name. We are then ready to begin coding. Before we start however we will discuss the overhead of the template file seed.nb

The first command is an important one because this cleans out the memory. This will ensure that we are not relying upon unreferenced data, a very important consideration. It also precludes a problem new users often create: using the same variable names in different notebooks. For example, if we have two notebooks open and they both use a variable *x*, the value of *x* is the value from the most recent execution. This can create an extremely difficult debugging environment. In fact, when we assist new users who present us with problematic notebooks, we first run the **Clear** command to prevent any ambiguities from manifesting. At first glance, this command has a simple syntax. For example, to clear the variables **x** and **name** you would write:

```
Clear[x, name]
```

Let's look at some examples of clearing variables. First, we will show how to clear numeric values.

```
x = 2;              (* assign integer value of 2 *)
Print["x = ", x];   (* interrogate *)
x =.;               (* clear x value *)
Print["x = ", x];   (* interrogate *)
```

```
x = 2
```

```
x = x
```

The last statement simply tells us that **x** is a variable with no assigned value. As you might suspect, we can clear string values in the same manner.

```
x = "bcc";            (* assign integer value of 2 *)
Print["x = ", x];     (* interrogate *)
Clear[x]              (* clear x value *)
Print["x = ", x];     (* interrogate *)
```

```
x = bcc
```

```
x = x
```

This kind of generality makes programming in *Mathematica* much easier. One command operates across all data types. Another way to clear a variable is to use the **Set** statement (=) and the period as shown here.

```
x = 2;                (* assign integer value of 2 *)
Print["x = ", x];     (* interrogate *)
x = .;                (* clear x value *)
Print["x = ", x];     (* interrogate *)
```

```
x = 2
```

```
x = x
```

Notice that *Mathematica* has told us that *x* has no value and is simply a variable.

This syntax is straightforward. The issue though is that we want to be able to clear all of our variables without having to shut down the kernel and relaunch. That syntax is a bit more cumbersome. We admit that the first line in your first notebook is a bit esoteric for it is the command that clears all global variables.

```
(* clear all variable names and assignments *)
Clear["Global`*"];
```

⚠ Clear["Global`*"] is one of the most useful commands available. Use it to ensure that your notebooks don't call for unreferenced data.

These next commands have to do with file system organization. You should plan for the day when *Mathematica* is outputting many types of graphics and is creating data files to record the results of the computations. We give our strongest warning against mixing notebooks with the installation files for *Mathematica*. Put them some-

where else; a little organization up front allows for a graceful expansion later. In the seed notebook we define a variable **dirnb** which points to a highest level of the notebook file system structure. All file system references are then relative to **dirnb**. If you work on different computers and platforms, a good idea is to maintain duplicate files structures. If you work on different platforms, you would only then need to change the definition of **dirnb**.

The subdirectories of interest are **dirPack** which is a repository for all modules (or packages) common to all notebooks. In general, the computation modules are kept in the project or topic folder. This precludes a packages directory with dozens or hundreds of modules. The home directory, dirHome, points to a specific project or topic folder. Clearly, we are advocating a horizontal file structure over a vertical one. Again, as your notebooks proliferate we feel this is the best structure. Every folder containing an executable notebook must have two folders: data and graphics. The data folder contains input files and output files. Since the contents of a notebook are easily changed, it is a good idea to archive results in data files. Also keep in mind the time value of some of your data: if your computer must run overnight or over the weekend, your data has a great time value since it cannot be easily regenerated. It is a good idea to archive the output into data files. Finally, the graphics folder will contain the graphics output. We use a module which automatically outputs graphics in three formats: EPS, BMP and JPEG or GIFF. The extended PostScript (EPS) files are for use in printed matter like journal articles. The bitmap (BMP) files are for use in screen display or projection. The Joint Project Experts Group (JPEG) or Graphical Interchange File Format (GIFF) files are for viewing and data management. Since these files can be viewed by any web browser, they can be viewed quickly allowing the user to manage his files. If you wish, you may modify the graphics output module to segregate the graphics files and place them into subfolders such as EPS.

```
(* relative file structure *)
(* directory of Mathematica files *)
dirnb = "/Volumes/gluon/nb/";
(* packages added by the user *)
dirPack = dirnb <> "packages/";
(* directory for this project *)
dirHome = dirnb <> "books/out/Practical Mathematica/04 io/";
(* subdirectory for data I/O *)
dirData = dirHome <> "data/";
(* subdirectory for graphics files *)
dirGraph = dirHome <> "graphics/";
```

Notice that this presumes the existence of the samples/ subdirectories data/ and graphics/. Users of even modest familiarity will be reading and creating data files by the drove. Also, we'll see the ease and necessity of outputting graphics files in multiple

formats. For these reasons it is best to create subdirectories separate from your notebooks.

The final statement in this block, **Null**, is simply a placement. As the name implies no action is taken. If we did not have that command, then the line above it would be broken with a carriage return and the comment string would become the final line.

An important advantage of this directory structure is how it facilitates moving machines between different platforms. As long as you maintain mirror images of the file systems on your different platforms, jumping platforms is easy. For example, this case is obviously written for a Windows environment. When we move this file over to the Macintosh, we change the first command to read:

```
(* directory of Mathematica files *)
dirnb = "/Volumes/gluon/nb/";
```

Next we provide common settings. For example, we provide a list of colors such as cred (color red) for RGBColor[1,0,0] which makes the notebooks more readable and uniform. The stamp module has a few goodies like the ID stamp shown as output. This is a valuable string which will help you sort your notebooks and is particularly useful in a multi-machine multi-platform environment. Also, it keeps track of which *Mathematica* version you used. The string is also used to tag data files to identify the source. We cannot overstate the importance of being able to associate a specific notebook with a data file. This way you can track versions and verify that your data was generated using a corrected algorithm. It is also invaluable to be able to associate a graphic image with a notebook. The graphics output routine that we mentioned above not only creates multiple format images, it also maintains a list of these images so that you may trace the image to the source notebook. Finally, this section gives the graphical outputs a common look. We pick a single font and image size for all the graphics using the **$TextStyle** command. We also store some commonly used settings for plot routines to reduce the amount of typing that we must do and to ensure that our plots have a uniform appearance. The last command, **Null**, is a placeholder that prevents the comment from the line above jumping to a new line beneath the command.

When you are moving notebooks between home and office and lab and different facilities and there are different platforms and different versions of *Mathematica* it can be quite a chore keeping your output straight. Also when we write a data file, we want to be able to trace it to a source notebook. This leads us to write a module called stamp which records this information:

```
(* time and date stamp *)
stamp
```

```
prepared by dantopa on quark using v 5.0 on 3/1/05 at 22:48:25 in /
  Volumes/gluon/nb/books/out/Practical Mathematica /04 io/import xls 01.nb
```

Finally, we include a statement at the end of the notebook which will save the notebook. This is advantageous if you are doing an extended run. In the Windows environment you can open a notebook and execute all the commands, including the terminal save command which will record all of your data. We call your attention to cases where you are generating megabytes of graphical data in the front end. In this case an automatic save may not be beneficial. We urge users to store graphical output in one or a few of the data formats mentioned above and keep your notebooks free of graphics when practicable.

When you put it all together, the seed notebook looks like this

```
(* clear all variable names and assignments *)
Clear["Global`*"];
```

```
(* relative file structure *)
(* directory of Mathematica files *)
dirnb = "/Volumes/gluon/nb/";
(* packages added by the user *)
dirPack = dirnb <> "packages/";
(* directory for this project *)
dirHome = dirnb <> "books/out/Practical Mathematica/04 io/";
(* subdirectory for data I/O *)
dirData = dirHome <> "data/";
(* subdirectory for graphics files *)
dirGraph = dirHome <> "graphics/";
```

```
(* common variable definitions *)
Get["common.m", Path → dirPack];
(* common variable definitions and settings *)
Get["CRC common.m", Path → dirPack];
(* time and date stamp *)
stamp
```

```
prepared by dantopa on quark using v 5.0 on 3/1/05 at 22:48:25 in /
  Volumes/gluon/nb/books/out/Practical Mathematica/04 io/import xls 01.nb
```

```
(* your commands go here *)
```

```
(* save notebook *)
NotebookSave[SelectedNotebook[]];
```

We close with the observation that the stamp string displayed above is available at run time and is an excellent way to tag data files, especially on processes shared by several machines. Such a process is described below.

1.2.2 Distributed computing

Here we present a toy model for using several machines to complete a task. In our example, we used half a dozen machines to generate high-density plots of two-dimensional functions we were generating. We have one machine which is the controller. The controller has two special files: the task assignment file and the *Mathematica* plot module. The task assignment problem can be an ASCII file. It is simply the number of the next plot to evaluate (the plot were numbered linearly).

Distribution is simple. In our case, all servant machines had mapped their B: drive to the controller directory. In this way every servant is running the exact same code. This is imperative. Different code on different machines creates headaches. When the servant machine is ready, it first gets a task assignment. For example, if plot 15 is the next plot to do out of 100 plots, then the task assignment file contains two lines: the value 15 and the value 100. To get an assignment, a servant machine opens the file, reads 15, checks that this is less than or equal to the termination value 100, writes 16 and closes the file. The plot module is loaded from the controller machine which ensures that the plots are identical.

Distributed processing is a great way to use computing power normally unused overnight or during the weekends. As a courtesy to those affected, they should be reminded to reboot their machines since *Mathematica* may have saturated their physical memory and may push applications to virtual memory on the hard drive which is much slower.

Note that Wolfram Research has a parallel processing add-on package to turn multiple CPUs into a true parallel processing computer. You can read about it at www.wolfram.com/products/applications/parallel/.

1.2.3 Minor bugs in the front end

There are some minor bugs in the front end that have persisted through a few releases. Fortunately, they do not affect the quality of the kernel computations. However, one class of error will prevent the kernel from seeing your code.

Leading spaces

At times the front end will introduce leading spaces. Suppose, for example, that you have created and saved the directory structure above. It executes fine and when you close the notebook, the code block looks exactly as it does above. But when you open it, spaces have been prepended. When you execute the code block, you will see

```
(* relative file structure *)
(* directory of Mathematica files *)
dirnb = "/Volumes/gluon/nb/";
 (* packages added by the user *)
 dirPack = dirnb <> "packages/";
 (* directory for this project *)
 dirHome = dirnb <> "books/out/Practical Mathematica/04 io/";
 (* subdirectory for data I/O *)
 dirData = dirHome <> "data/";
 (* subdirectory for graphics files *)
 dirGraph = dirHome <> "graphics/";
```

```
Null[4]
```

The Null[4] tells us that we have evaluated two empty lines. To fix this malady, delete just the first extraneous blank — the blank on the line with the dirPack assignment and reevaluate the code block. The problem is gone until you launch the notebook again.

⚠ The front end may inject extraneous leading spaces on some lines when the notebook is launched. Deleting these spaces and resaving the file will not prevent the spaces from reappearing at launch time.

This problem doesn't typically affect the functioning of your notebook. But the sudden appearance of the Null command often confuses new users. There are some cases where the Null can cause considerable confusion.

```
(* module to generate the squared spherical harmonic function *)
pgen[l_Integer, m_Integer] := Module[{g},
    g = SphericalHarmonicY[l, m, θ, φ];
    g2 = Simplify[g Conjugate[g], θ ∈ Reals ∧ φ ∈ Reals];
   ];
 (* specify the eigenstate *)
 {l, m} = {5, 3};
 (* call the generation module *)
 pgen[l, m]
 (* display the squared function *)
 g2
```

$$\frac{385\,\text{Null}^3\,(7+9\,\text{Cos}[2\,\theta])^2\,\text{Sin}[\theta]^6}{4096\,\pi}$$

It turns out that the Null comes from the module definition. However, this time there is no discernible difference between the snippet that produced the Null (above) and the snippet that functions as desired (below). To fix the problem we went into the module, deleted the comment and then immediately pasted it back into the same position. This restored normal operation.

```
(* module to generate the squared spherical harmonic function *)
pgen[l_Integer, m_Integer] := Module[{g},
    g = SphericalHarmonicY[l, m, θ, φ];
    g2 = Simplify[g Conjugate[g], θ ∈ Reals ∧ φ ∈ Reals];
   ];
 (* specify the eigenstate *)
 {l, m} = {5, 3};
 (* call the generation module *)
 pgen[l, m]
 (* display the squared function *)
 g2
```

$$\frac{385\,(7+9\,\text{Cos}[2\,\theta])^2\,\text{Sin}[\theta]^6}{4096\,\pi}$$

⚠ The *Mathematica* front end may react unfavorably to comments and cause extraneous Nulls to appear. Deleting the comments and repasting them will restore normal operation for the remainder of the session.

Extraneous characters

From time to time you may find a case where extraneous characters are injected around your embedded statements. For example, we saved this snippet of code.

```
(* number of rectangles to create *)
nrec = 8;
(* create a sequence of functions *)
lst = Table[d[x, i], {i, nrec}];
(* plot all rectangles *)
g1 = Plot[Evaluate[lst], {x, -1, 1}, Axes → False, Frame → True];
```

and when we launch *Mathematica* and open the notebook, we find this.

```
(* number of rectangles to create *)
nrec = 8;
 \(□(* create a sequence of functions *)□\)
 lst = Table[d[x, i], {i, nrec}];
 (* plot all rectangles *)
 g1 = Plot[Evaluate[lst], {x, -1, 1}, Axes → False, Frame → True];
```

extraneous characters padding this line.

Delete these characters and continue working.

⚠ The front end may pad a comment line with extraneous characters when the notebook is launched. Deleting these spaces and resaving the file may not prevent the spaces from reappearing at launch time.

Blinding the kernel

There are times when embedded comments will blind the kernel. By this we mean that the comments prevent the kernel from executing subsequent comments in a cell. The problem manifests as follows. You develop code and debug it, content that it works fine. You save and close your notebook. When you reopen the notebook, you have a problem with a cell. You discover that lines below a certain comment are not being passed to the kernel.

This fix is simple. You delete the offending comment and execute the cell. You can then paste the comment back into the original place and the cell will continue to operate normally for the rest of the session. When you reopen the notebook, you will probably have the same problem with the same commands failing to execute.

For example, we have a notebook with a long cell beginning with this code.

```
(* build the interaction matrix *)
α = Table[μ = p[[i]][[1]]; ν = p[[i]][[2]];
   τ[μ, ν] + τ[μ, ν + 1] + τ[μ + 1, ν + 1] + τ[μ + 1, ν], {i, 1, m}];
(* tag the neighbors *)
A = DiagonalMatrix[α]; Clear[α]
```

When we open the book, we discover that nothing below the comment (* tag the neighbors *) has executed. We delete that comment line, execute the cell, and paste the comment back into the notebook. For the remainder of the session this cell executes normally.

⚠ At times the kernel may be blinded by comments embedded in cells. You can delete the comment and execute the cell normally.

1.2.4 Notebooks and crashes

In this section we would like to discuss crashes and other events that cause you to lose control of you process. In the following convention we will use the nb file extension for all *Mathematica* files. Although Macintosh files do not read the file extension and therefore do not require it, the file extension is a good idea because it simplifies jumping platforms.

1. *Mathematica* crash. Sometimes *Mathematica* will crash while you are performing some process like calculating page breaks or writing notebooks with a large amount of graphics. Under earlier versions this would at times cause the file to corrupt and make it extremely difficult. Almost all of the crashes in later files are recoverable. If your file becomes corrupted, you can go through the menu bar to File > Open Special to bypass the malady. A concept that we use is to spawn a new notebook periodically. For example, if you had a notebook to calculate Catalan's constant, we would do a Save As operation to create catalan 01.nb, catalan 02.nb, etc.
2. Loss of kernel control. In some cases the kernel will separate from the front end and continue to process indefinitely. The interrupt and abort commands under Kernel on the menu bar have no effect. In this case your only redress is to close *Mathematica*. At times even this will slay the errant process and that must be done manually through the Windows Task Manager or the *Mathematica* Force Quit option. You should monitor your virtual memory use, particularly in the Windows environment. Memory leaks — episodes where closing *Mathematica* do not liberate all the virtual memory it was using — can cause your machine to bog down or crash. In these cases you should restart your computer.
3. Loss of front end control. If you accidently try to view an enormous matrix, display an incredibly long result, or create a monster graphic, your application will

appear to stall. It is just the front end working busily for you. But if completion of the task will take days, weeks, or years, waiting may not be an option. Your only recourse is to close *Mathematica* and to relaunch. When you get back into your notebook, divert your output to a file. Also as in the second case, check for memory leaks and restart if warranted.

4. *Mathematica* has very strong capabilities and can easily generate problems that gobble up gigabytes of memory. We have noticed on Windows machines that saturating the memory can lead to erratic behavior. Other applications may exhibit unusual symptoms and *Mathematica* may act strangely and give the wrong answer. Although quite rare, the user should be aware. The prescribed cure is to reboot. Although many people do not believe us, we have demonstrated problems with virtual memory in the Windows environment. We had notebooks that would not work when using virtual memory. As soon as we boosted physical memory the notebooks could work. Yes, we know the official story is that virtual and physical memory have the same functionality (albeit different speeds), but the empirical data clearly shows that physical memory is best. In a Windows environment *Mathematica* will take less than 2 GB, so if you are wondering how much RAM to install make sure that *Mathematica* will have its 2 GB.

1.3 Entering data

Are you a keyboard person or a mouse person? Either way your preferences are satisfied with *Mathematica*. Mouse users can pick and choose from the well-organized palettes available from the menu bar. These palettes will be presented next. Keyboard users will be happy to find a rather extensive set of keyboard shortcuts. These shortcuts are nice and we show an example below.

1.3.1 Standard palettes

Mathematica has many useful palettes to simplify our lives. To access the palettes, go through the menu bar File > Palettes and you will see the following list of nine palettes:

OpenAuthorTools	BasicInput	CreateSlideShow
AlgebraicManipulation	BasicTypesetting	InternationalCharacters
BasicCalculations	CompleteCharacters	NotebookLauncher

Unfortunately, not all of these palettes display their titles. For this reason, and to acquaint the reader with their functional capability we will show all the palettes below.

Many of the palettes have an active legend. At the bottom of the frame there is an info window which displays the keyboard shortcuts for the command you are entering. For example, in the BasicTypesetting palette we are ready to use the cursor to select the Greek letter y. In the legend we see that the keyboard shortcuts for this character are ESC ps ESC.

These palettes are "sticky" in the sense that their state is recorded when *Mathematica* closes and the environment is recreated when *Mathematica* is launched. In other words if you have three palettes open when you close *Mathematica*, the same three, and only these three will be opened when *Mathematica* is launched. The demo palettes listed below are not sticky and they need to be opened each time you use them.

1.3.2 Keyboard shortcuts

Over time people tend to gravitate toward the keyboard shortcuts because they are so much faster. We will show two examples in this section. Also, for the remainder of the chapter we will display the keyboard shortcuts that we used to create the *Mathematica* code. At the end of the chapter we point you to an extensive list of the keyboard shortcuts in the Help Browser.

The keys listed are pressed sequentially unless they are joined by a plus sign. For example, CTRL+SPACE means that you should press and hold the CTRL key and then press the SPACE bar. A sequence like scV means to sequentially press the keys s, c, and V.

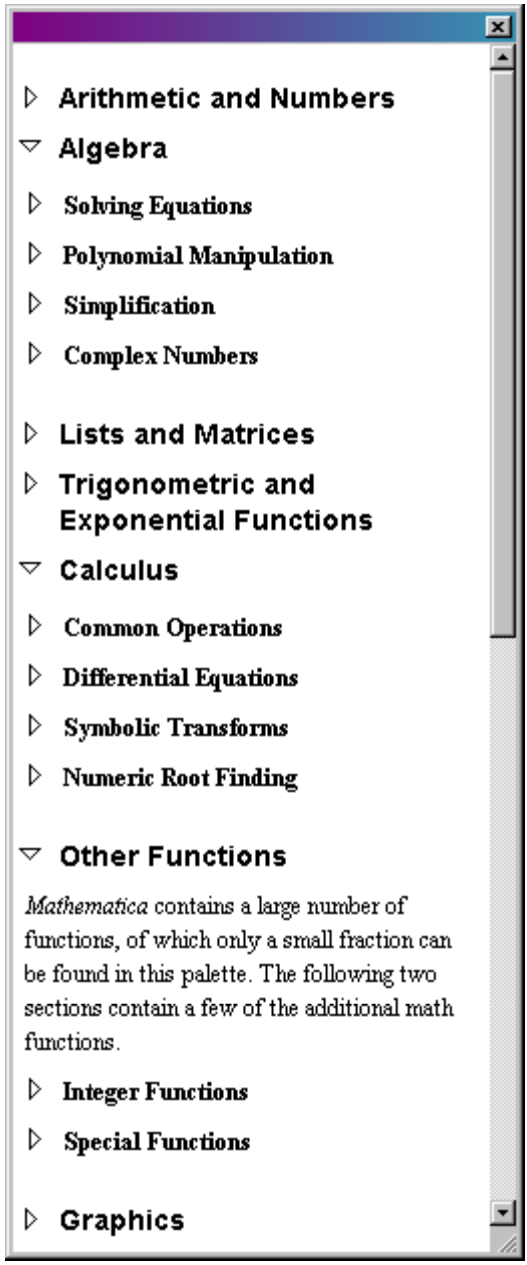
$$\rho = \sqrt{x^2 + y^2}$$

Keyboard sequence:
ESC r ESC = CTRL+2 x CTRL+6 2 CTRL+SPACE + y CTRL+6 2 CTRL+SPACE CTRL+SPACE

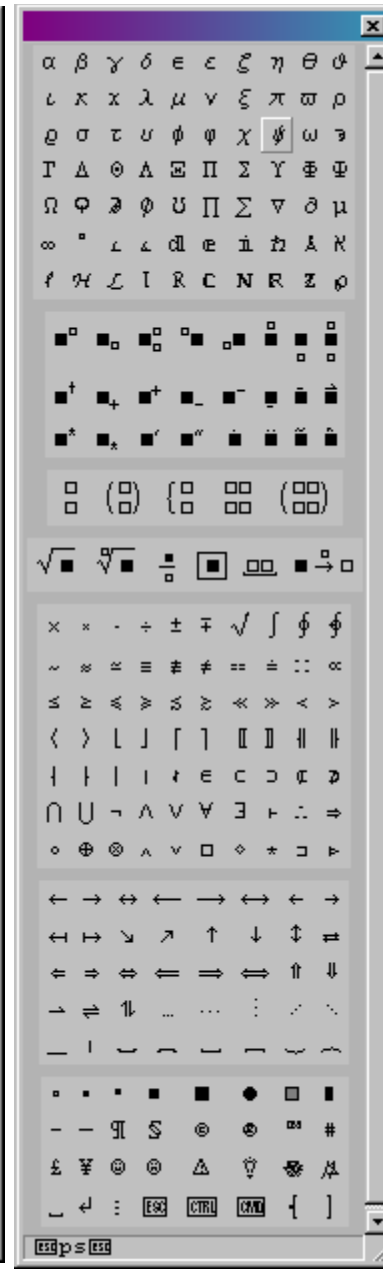
Note the repeated use of CTRL+SPACE at the end. The first sequence pulls the cursor down from the exponent and the second sequence moves the cursor outside of the radical.

$$v = \frac{4}{3} \pi r^3$$

Keyboard sequence:
ESC scV ESC = 4 CTRL+/ 3 CTRL+SPACE + ESC p ESC r CTRL+6 3



BasicCalculation palette



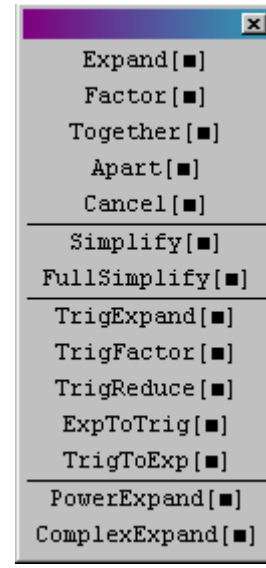
BasicTypsetting palette
(active legend)



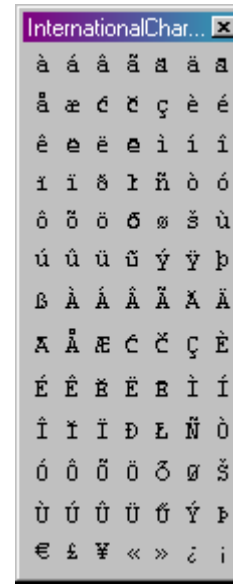
CompleteCharacters palette
(active legend)



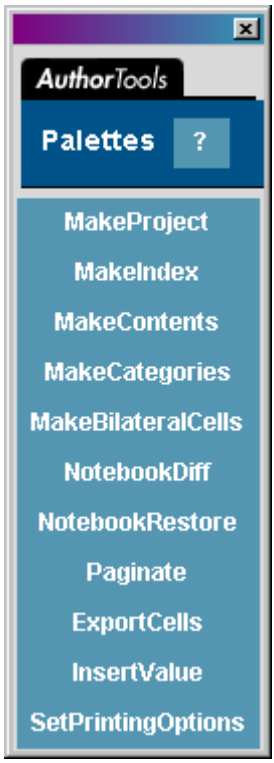
BasicInput palette
(active legend)



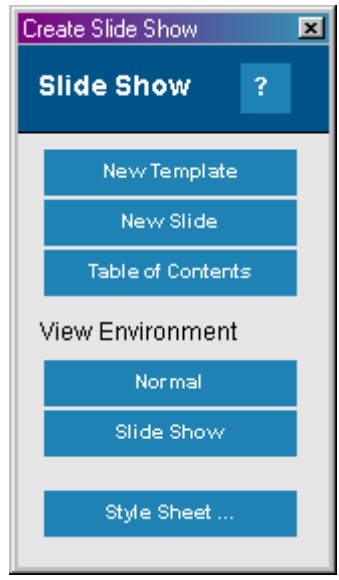
AlgebraicManipulation
palette



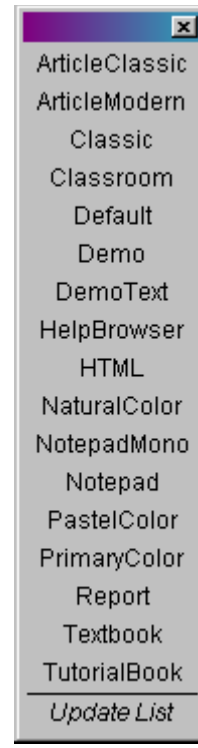
InternationalCharacters
palette



OpenAuthorTools palette



CreateSlideShow palette



NotebookLauncher palette

Note that ESC scV ESC is the shortcut to enter a scroll V.

Once again we call your attention to the high quality of support documentation on the Wolfram web sites and we encourage you to use it.

Getting Started > Working with Notebooks > Entering Mathematical Notation

Front End > Keyboard Shortcuts > Microsoft Windows

Front End > Keyboard Shortcuts > Macintosh

Front End > Keyboard Shortcuts > X

The documentation is also available on the Internet at the URL shown below. From here on, information on the Internet that duplicates the Help Browser will not be referenced. We will call attention only to Internet materials not available in the Help Browser.

http://documents.wolfram.com/mathematica/FrontEnd/KeyboardShortcuts/

1.4 Data structures

Many of us are familiar with the representation of numbers in modern computer languages. We expect to find integers of different sizes (say 4 or 8 bytes), real numbers in single and double precision, and complex numbers composed of pairs of real numbers and strings. *Mathematica* gives us a far richer set of tools:

1. Integers of arbitrary size
2. Rational numbers of arbitrary size
3. Symbols such a p
4. Real numbers of double precision
5. Complex numbers and quaternions
6. Lists

Our experience from learning *Mathematica* and from aiding others to learn *Mathematica* has shown us that it is difficult to look at the list of data structures and realize how very different *Mathematica* is. Perhaps this is because we have used the conventional data structures to explore our world for so long that we are now accustomed to formulating problems in terms of the tools we are familiar with.

For example, the arbitrary sized integers and rational number representation allow us to compute some answers to arbitrary precision. In other words, we can get as much precision as we want. It becomes a question of computer time. This is a great boon. Perhaps some users will think of arbitrary precision and an extension of double precision — but it is far more than that as we shall see. We are now able to do computations far closer to singularities. Matrix inversions can now dance within a razors edge of singularity. It truly empowers us for we are now no longer limited by the computer.

Also, the symbolic capability allows us to do incredibly long calculations and recover exact answers or answers of arbitrary precision. This is a most welcome gift. Consider the case of using *Mathematica* to develop an algorithm. Typically we test the algorithm by looking for a difference from zero in some special cases. In FORTRAN we get an answer like machine noise and must decide whether or not is it truly machine noise or some algorithmic imprecision.

Do not think that because *Mathematica* allows completely arbitrary list structures that it is list-friendly. *Mathematica* is the ultimate list-processing platform and is designed around this concept. So many of our examples will show that *Mathematica* was designed around the list concept. When you see the performance improvements from using lists, you will realize *Mathematica*'s true pedigree.

1.4.1 Representations

There are many ways to represent data with *Mathematica*. We shall explore some of the most popular types using the sample list shown here that allows you to see how different types of data are represented.

```
(* sample list containing symbols,
 a list, an integer, a real, and an operation *)
lst = {a, π, x^2 + y^2, Sin[2.2], 8, {2, 3, 5}}

{a, π, x^2 + y^2, 0.808496, 8, {2, 3, 5}}
```

If you want to write data to an ASCII file, your two best options are FortranForm and CForm. The list outputs reals to full double precision and can handle lists.

```
FortranForm[lst]

List(a,Pi,x**2 + y**2,0.8084964038195901,8,List(2,3,5))
```

```
CForm[lst]

List(a,Pi,Power(x,2) + Power(y,2),0.8084964038195901,8,List(2,3,5))
```

Notice the encoding of the quadratic sum. You are able to output your computations from *Mathematica* directly to FORTRAN or C code. The problem is that operations like Determinant[], FindMinimum[], and Inverse[] do not generate code. As we go to press, we understand that the package MathCode remedies much of these problems.

http://www.wolfram.com/products/applications/mathcode/

```
TeXForm[lst]

\{ a,\pi ,x^2 + y^2,0.808496,8,\{ 2,3,5\} \}
```

```
StandardForm[lst]

{a, π, x^2 + y^2, 0.808496, 8, {2, 3, 5}}
```

```
TraditionalForm[lst]
```

$$\{a, \pi, x^2 + y^2, 0.808496, 8, \{2, 3, 5\}\}$$

```
InputForm[lst]
```

```
{a, Pi, x^2 + y^2, 0.8084964038195901, 8, {2, 3, 5}}
```

```
OutputForm[lst]
```

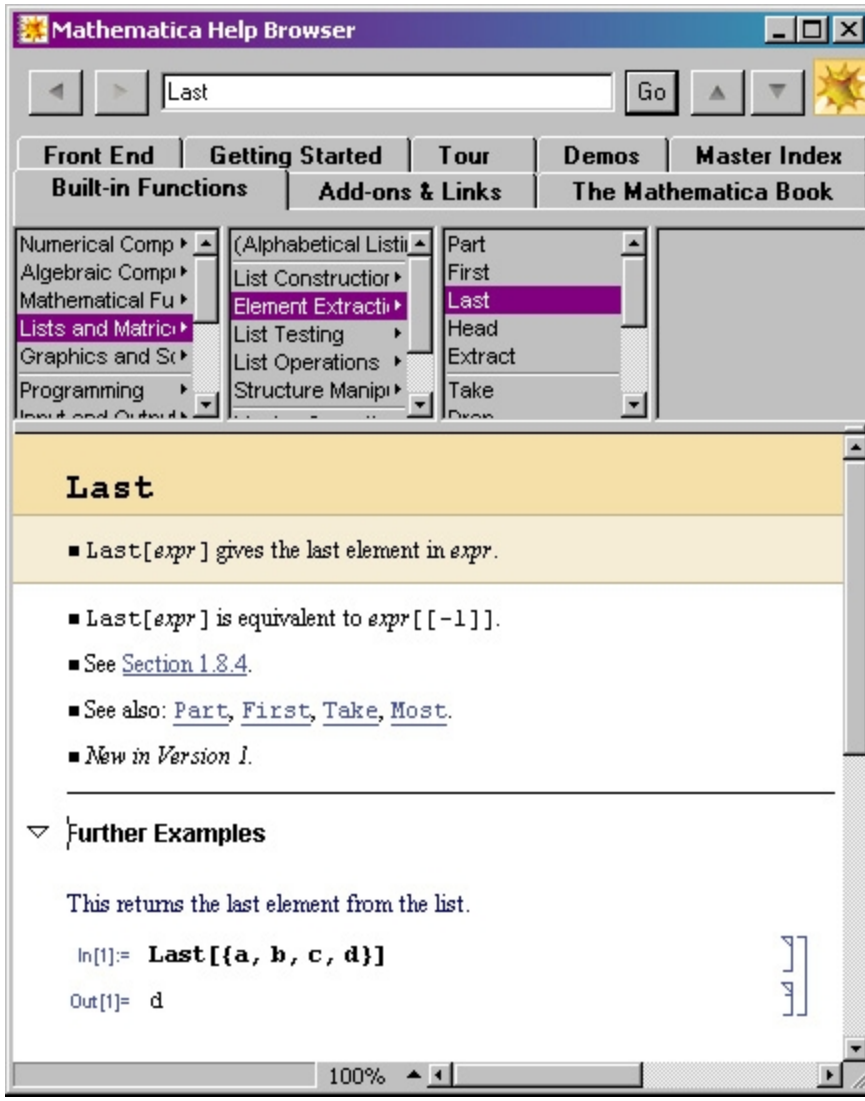
```
         2    2
{a, Pi, x  + y , 0.808496, 8, {2, 3, 5}}
```

1.4.2 Queries

A wonderful feature of the interactive *Mathematica* environment is being able to query our computations in real times. The user should immediately etch into memory two very important commands: Head[] and ?.

The Head of a number describes its type. The question mark operator shows us all definitions of the symbol. This is most helpful for new users who have accidently created multiple definitions of a variable or module. We will frequently show how to call these commands and interpret their meaning because we think this should be the first step in constructing larger code blocks and debugging.

1.4.3 Entering data



You will be happy to learn that the front end is interpreting your commands as you type them and is providing visual and aural clues to call your attention to mistakes.

As an example, let's look at a command line from an example in chapter 3. When we need to enter complicated structures like this one, we often do it in phases since looking for stray punctuation can be tedious. Consider the full command shown here.

```
lines = Table[{Thickness[0.01 - n 0.002],
   Line[{{-1, 0}, {-1/(2 n), 0}, {0, n}, {1/(2 n), 0}, {1, 0}}]}, {n, nmax}]
```

We would first enter a shell like this.

```
lines = Table[{Thickness[], Line[{}]}, {n, nmax}]
```

This form is certainly more readable and easier to debug and to read. It tells us that we are creating a list of graphics primitives called lines. There are nmax elements in this list, and we will be varying the thickness of each line in the list.

We only need to stir in the thickness specification

```
0.01 - n 0.002
```

and the list of points describing the line.

```
{-1, 0}, {-1/(2 n), 0}, {0, n}, {1/(2 n), 0}, {1, 0}
```

Should you ignore this advice and begin typing, you will be aided by the Front End. Look at these examples which show how *Mathematica* reacts to different syntax errors.

```
lines = Table[{Thickness[0.01 - n 0.002],
    Line[{{-1, 0}, {-1/(2 n), 0}, {0, n}, {1/(2 n), 0}, {1, 0}}], {n, nmax}]
```

we have omitted the brace to close the graphics primitive.

As you become more familiar with the common typos, you will become more adept at interpreting the highlighted brackets, braces and parentheses. In this case, there was some ambiguity over where to place the need brace.

```
lines = Table[{Thickness[0.01 - n 0.002],
    Line[{{-1, 0}, {-1/(2 n), 0}, {0, n}, {1/(2 n), 0}, {1, 0}}]}, {n, nmax}]]
```

we added an unneeded bracket.

This next example is a bit more elaborate. Look how many delimiters are lit up.

```
lines = Table[{Thickness[0.01 - n 0.002],
      Line[{{-1, 0}, {-1/(2 n), 0}, {0, n, {1/(2 n), 0}, {1, 0}}]}, {n, nmax}]
```

we are missing a brace

The final example is the simplest.

```
lines = Table[{Thickness[0.01 - n 0.002],
    Line[{{-1, 0}, {-1/(2 n), 0}, {0, n}, {1/(2 n), 0}, {1, 0}}]}, {n, nmax]
```

we need a brace to close the iterator

1.4.4 *Mathematica* loves large numbers

After that rich defense of learning from examples, we are ironically going to take a quick diversion to talk about how *Mathematica* stores numbers. The internal representation used in *Mathematica* is quite clever and provides the application with a great deal of power. Of course numbers can be stored as integers or double precision objects, something familiar to all of us who have written computer programs. But there is also a rational number mode where rational numbers are represented as the ratios or two integers p and q. The integers p and q can be arbitrarily large. Also, some quantities such as π and $\sqrt{2}$ are treated as symbols.

Due to recent breakthroughs (** footnote www.mersenne.org/primes.htm), a good example to use is the Mersenne primes, M_n. These are prime numbers which follow the patterns

$$M_n = 2^n - 1 \tag{1.1}$$

where n is also a prime.

On May 15, 2004 Josh Findlay reported his discovery of the 41st Mersenne prime which has the form

$$2^{24,036,583} - 1 \tag{1.2}$$

Mathematica is very content to generate this number. We will use the TimeUsed[] command to keep track of how much CPU time is used to compute this number.

```
dsum = 0;
Do[
  dsum = dsum + digs[[i]];
  , {i, 10}];
Print["total digits = ", dsum];
```

```
total digits = 7235733
```

The number mersenne was computed in 0.04 seconds — less time than it takes to blink an eye.

You may have noticed that we did not display this monsterous number. It is very long and the display time is far greater than the generation time. This is why we did not use the Timing[] function in *Mathematica*. Representing this number in the front end took 59 seconds and 2240 pages. This makes the notebook extremely slow and unwieldy to use. As you will see, the front end is not an efficient place to place some results like extremely large numbers or detailed graphics. *Mathematica* is very happy to work with them in memory. For example, we can count the number of times each digit appears by:

```
digs = DigitCount[mersenne]

{723188, 722754, 722181, 723758,
 724196, 723856, 724543, 723551, 725093, 722613}
```

For example, the digit zero occurs 723,188 times. How many digits are there altogether? To find out we can sum this list. Our first attempt will be a basic do-loop.

```
dsum = 0;
Do[
  dsum = dsum + digs[[i]];
  , {i, 10}];
Print["total digits = ", dsum];
```

```
total digits = 7235733
```

The do-loop is probably familiar to many of you. Some counter is incremented and tasks are performed:

```
Do[
  command1;
  command2;
  , {i, 10}];
```

Of course, you could use the syntax Do[task1,task2,{i,10}] but this quickly becomes hard to read and debug. So we encourage our readers to use the vertical structure shown first.

By default, the counter increments by one. So writing {j,-3,18) would yield integer values for j varying from -3 to 18. If you want the counter to decrement instead of increment you must specify a step size. The obvious decrement of -1 is not assumed. For example, the syntax {counter, 20, 10} will not work because we are decrementing within a specified step size. The syntax {counter, 20, 10, -1} will work. We are not constrained to integers. We could use {x, 0, 1, 0.0025} for example.

However, we are now in a regime where *accumulation* errors may appear. This would be true for any computing language; it's just that *Mathematica* with its arbitrary precision provides a stage upon which such errors are more easily viewed.

For example, consider the syntax {x, 0, 1000, 0.0025}.

```
dsum = 0;
Do[
  dsum += digs[[i]];
  , {i, 10}];
Print["total digits = ", dsum];
```

```
total digits = 7235733
```

```
Print["total digits = ", Plus @@ dsum];
```

```
total digits = 7235733
```

The Mathematica Book > Mathematica Reference Guide > Some General Notations and Conventions > Mathematical Functions (A.3.10)

Mersenne Prime

```
(* clear any assignments to x *)
Clear[x];
(* diagnostic print before entering Block protection *)
Print["entering Block x = ", x];
(* entering Block *)
Block[{x},
  (* assign a value inside the loop *)
  x = 3.14;
  (* diagnostic print showing local x value *)
  Print["inside block x = ", x];
 ];
(* diagnostic print after leaving Block *)
Print["exiting Block x = ", x];
```

```
entering Block x = x
```

```
inside block x = 3.14
```

```
exiting Block x = x
```

Table[,{i,100},{j,100}}]
Do[Table[,{i,100}}]

We will also demonstrate the Block command which allows us to declare local variables. This is particularly useful if say, for example, you have a routine which requires a variable x. If x assumes a constant value, it is no longer a variable. However, we can use the Block command to create a probe. We can assign x a numerical value inside of the Block and outside of the Block x is still a variable.

First, let us demonstrate the behavior of the Block function. We cannot state how important it is for users to perform these type of exercises as they learn *Mathematica*.

1.4.5 Lists

Lists are a bedrock concept in *Mathematica* and experience suggests that new users don't immediately think in terms of lists. But all throughout this book, you will be bombarded with list formulations. Do not worry if they seem foreign — we also show you how to get by without them.

Generating lists

Let's look at a few of the ways that we are able to generate lists. The first command generates a range of numbers.

```
x = Range[5]

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

Here we create a table of areas for circles with radii 1, 2, 3, and 4.

```
Table[π r^2, {r, 4}] // N

{3.14159, 12.5664, 28.2743, 50.2655}
```

We use the numeric postfix **(//N)** to avoid getting a symbolic answer.

```
Table[π r^2, {r, 4}]

{π, 4 π, 9 π, 16 π}
```

Mathematica also can create lists of characters with this rather easy syntax.

```
x = CharacterRange["A", "G"]

{A, B, C, D, E, F, G}
```

Of course the numeric range can be easily controlled. If we don't supply a starting point *Mathematica* will start at one. Otherwise we are free to specify the starting and stopping points.

```
a = Range[-313, -301]

{-313, -312, -311, -310, -309, -308,
 -307, -306, -305, -304, -303, -302, -301}
```

The natural extension would be to allow us to alter the increment size. Be careful because in a do-loop we are forced to state a negative decrement.

```
Range[-313, -301, 0.73]

{-313, -312.27, -311.54, -310.81, -310.08,
 -309.35, -308.62, -307.89, -307.16, -306.43, -305.7,
 -304.97, -304.24, -303.51, -302.78, -302.05, -301.32}
```

We will talk about polymorphism throughout this book and we want to call your attention to the example we have just presented. We have used the **Range** command with three different syntaxes.

```
1. Range[end]
2. Range[character start, character end]
3. Range[start, end]
4. Range[start, end, increment]
```

All forms create a range, it is just a different kind of range. But regardless of what we wanted, it all starts with the **Range** command. This is rather nice feature of *Mathematica*. In time you will learn to savor this generality.

Concepts of use

One of the most seductive properties of *Mathematica* is its ability to have lists of lists. The lists can have different sizes and compositions. Consider the rudimentary example below.

```
b = Range[Range[4]]

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

Consider a case like the one of the spherical harmonics which we will see in section 99. At order n, there are $2n + 1$ polynomials. Suppose that in some measurement you got an integer for each term. If you were constrained to a rectangular array, your data could look like this

```
⎛2  0   0   0  0 0  0   0  0  0  0  0  0  0  0⎞
⎜1  5  -1   0  0 0  0   0  0  0  0  0  0  0  0⎟
⎜5  3  -5   2  5 0  0   0  0  0  0  0  0  0  0⎟
⎜3 -4  -3  -1  1 1  4   0  0  0  0  0  0  0  0⎟
⎜0  4   4  -4  1 1  1  -3 -3  0  0  0  0  0  0⎟
⎜0 -5  -1  -1  4 5  5  -2 -3  4  3  0  0  0  0⎟
⎜5 -5   1  -4  4 5 -3  -1  1  2 -3  3  2  0  0⎟
⎝3  0   5  -2  2 4 -4  -3 -3  4  2  2 -1  4 -2⎠
```

In programming and visualizing this form is suboptimal. We are wasting a lot of memory and the shape doesn't correspond to our problem. Here is how *Mathematica* prefers to save the states:

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

which can be recast in the form:

```
% // MatrixForm

⎛                          {2}                          ⎞
⎜                      {1, 5, -1}                       ⎟
⎜                  {5, 3, -5, 2, 5}                     ⎟
⎜              {3, -4, -3, -1, 1, 1, 4}                 ⎟
⎜          {0, 4, 4, -4, 1, 1, 1, -3, -3}               ⎟
⎜       {0, -5, -1, -1, 4, 5, 5, -2, -3, 4, 3}          ⎟
⎜    {5, -5, 1, -4, 4, 5, -3, -1, 1, 2, -3, 3, 2}       ⎟
⎝{3, 0, 5, -2, 2, 4, -4, -3, -3, 4, 2, 2, -1, 4, -2}    ⎠
```

which has the same geometry as the problem: a single term, three terms, etc.

Let's look at the example of the circle polynomials of Nobel laureate Fritz Zernike. [** Born and Wolf]. Although this is a specific case, this example applies to all polynomials with rational coefficients.

Although the circle polynomials as they are called are defined over the unit disk, one typically makes with a CCD array. So we are forced to abandon the polar coordinates of the disk and use the Cartesian coordinates of the imaging system. For this reason, we will present the Cartesian form of the first few terms.

$$Z_{00}(x,\ y) = 1$$

$$Z_{10}(x, y) = x$$

$$Z_{11}(x, y) = y$$

$$Z_{20}(x, y) = 2xy$$

$$Z_{21}(x, y) = 2x^2 + 2y^2 - 1$$

$$Z_{22}(x, y) = -x^2 + y^2$$

Storing the polynomials as polynomials is quite wasteful and makes them extremely hard to manipulate. But if we can store them as vectors, we have achieved significant advantages.

Look the sequence of monomials:

$$B = (\{1\}, \{x, y\}, \{x^2, xy, y^2\}, \ldots \tag{1.3}$$

as a basis for a vector space. The Zernike polynomial coefficients can be collected to form a coefficient vector, *a*. Look at the coefficient vectors for the polynomials listed above.

$$a_{00} = \{\{1\}\}$$

$$a_{10} = \{\{0\}, \{1, 0\}\}$$

$$a_{11} = \{\{0\}, \{0, 1\}\}$$

$$a_{20} = \{\{0\}, \{0, 0\}, \{0, 2, 0\}\}$$

$$a_{21} = \{\{-1\}, \{0, 0\}, \{2, 0, 2\}\}$$

$$a_{22} = \{\{0\}, \{0, 0\}, \{-1, 0, 1\}\}$$

In this scheme we can recreate the polynomials by taking the dot product of the amplitude vector and the basis states. For example

$$Z_{20}(x, y) = a_{20} \cdot B = \{\{0\}, \{0, 0\}, \{0, 2, 0\}\} \cdot \{1\}, \{x, y\}, \{x^2, xy, y^2\} = 2xy \tag{1.4}$$

When manipulating the Zernike polynomials in the computer this vector form is singularly helpful. As mentioned before, this applies to every polynomial with rational coefficients allowing to go from a Zernike representation to, say, a Chebysheff representation via affine transformations.

Now we can turn our attention to the structure of lists. Consider a concrete example.

```
zernike = {{{1}}, {{0, 1, 0}, {0, 0, 1}},
   {{0, 0, 0, 0, 2, 0}, {-1, 0, 0, 2, 0, 2}, {0, 0, 0, -1, 0, 1}}};
```

The Dimensions command counts the number of elements.

```
Dimensions[zernike]

{3}
```

This tells us we have three lists. To get the size of the lists we can Map the Dimensions operator across the list zernike. We don't want to be sidetracked here, so we will explain the mapping operation later. For now accept that we probe the structure of zernike with this command.

```
Dimensions /@ zernike

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

These results tell us that the three elements in zernike are

1. a list with one element
2. a list with three elements
3. a list with six elements

The hierarchy is quite useful. For example, we can get all the coefficient vectors for order *j* by taking the (*j* - 1)th part of Zernike. There are the coefficient vectors for the entire second order.

```
zernike[[3]]

{{0, 0, 0, 0, 2, 0}, {-1, 0, 0, 2, 0, 2}, {0, 0, 0, -1, 0, 1}}
```

We can conjure up the term corresponding to a specific polynomial.

```
zernike[[3]][[1]]

{0, 0, 0, 0, 2, 0}
```

This is the term we used in equation 1.4. We can isolate the coefficient multiplying the cross term *xy*.

```
zernike[[3]][[1]][[5]]

2
```

Manipulating lists

Mathematica has been designed around the concept of lists and you will find a natural comfort using the application to manipulate lists. Look how simple it is to subtract one list from another.

```
x = Range[5];
y = Range[11, 15];
x - y

{-10, -10, -10, -10, -10}
```

We can easily create a table of the first six odd numbers:

```
x = Table[2 n + 1, {n, 0, 5}]

{1, 3, 5, 7, 9, 11}
```

or the first six prime numbers:

```
Prime[Range[6]]

{2, 3, 5, 7, 11, 13}
```

There is even a facility to generate lists using functions:

```
z = Array[Sin, 3]

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

Mathematica is faithfully holding these symbolic forms until we request numerical evaluation which we can do in prefix:

```
N[z]

{0.841471, 0.909297, 0.14112}
```

or postfix form:

```
z // N

{0.841471, 0.909297, 0.14112}
```

Of course, we are only shown the first six digits of an answer unless we request more.

```
N[z, 20]

{0.84147098480789650665,
 0.90929742682568169540, 0.14112000805986722210}
```

All *Mathematica* functions operate on lists seamlessly:

```
Sin[Range[4]]

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

These lists can have any structure and dimension:

```
Sin[Range[Range[4]]]

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

Notice that the answer has the same structure as the argument. We can force numerical evaluation at any time.

```
% // N

{{0.841471}, {0.841471, 0.909297}, {0.841471, 0.909297, 0.14112},
 {0.841471, 0.909297, 0.14112, -0.756802}}
```

We can use the command Flatten to force all the numbers into a linear order. You can see that Flatten will remove all the internal braces.

```
Flatten[%]

{0.841471, 0.841471, 0.909297, 0.841471, 0.909297,
 0.14112, 0.841471, 0.909297, 0.14112, -0.756802}
```

All *Mathematica* operators are content to operate upon lists.

```
% // N

{0.841471, 0.909297, 0.14112, -0.756802}
```

The lists can have any form of content.

```
Sin[{1, π, 3.1415926, Dan}]

{Sin[1], 0, 5.35898×10^-8, Sin[Dan]}
```

The Head command is an extremely useful command for the beginner. It specifically identifies the data type. This is of vital importance because you will discover that *Mathematica* is extremely disciplined about how it handles different data types. This may surprise the new user who has just seen evidence that we can apply virtually any operator to any data type, but as we go along you will see the precision with which *Mathematica* distinguishes data types.

```
Head[1]

Integer
```

Notice that π is not a number — it is a *symbol* representing a number. This subtlety is required for an arbitrary precision computation engine.

```
Head[π]

Symbol
```

When we take the **Head** of the numerical value of π, we see that we have a **Real** number.

```
Head[3.1415926]

Real
```

We see that the variable `Dan` is a symbol now.

```
Head[Dan]

Symbol
```

Again for arbitrary precision, we see that *Mathematica* is keeping this as a functional value, not a number.

```
Head[Sin[1]]

Sin
```

Mathematica looks at this statement and recognizes the sine function has produced an integer. Since integers are an exact data type in *Mathematica*, this expression is evaluated as -1.

```
Head[Sin[π]]

Integer
```

Let's explore how these data types help *Mathematica* maintain arbitrary precision. Consider the statement below where we know the answer from basic algebra.

```
(* algebraic expression with obvious answer *)
10^24 - Sin[1] - 10^24

-Sin[1]
```

However, in a double precision world we will not get this answer. We will use the numeric operator N to show us what a double precision computation of this expression would yield:

```
(* double precision destroys the obvious relationship *)
N[10^24 - Sin[1]] - N[10^24]

0.
```

This is a nuance and we want to clarify the concept. Consider the example below. You could not form this number using a computer language based on IEEE double precision variables.

```
z = 10^24 + Sin[1]

1000000000000000000000000 + Sin[1]
```

However, because *Mathematica* can handle arbitrarily large integers and symbols, we can compute this answer to arbitrary precision.

```
N[z, 50]

1.0000000000000000000000000841470984807896506652502 3 × 10^24
```

Over time you will see that this capability exposes us to entirely new vistas.

```
Clear[x];
f1 = Exp[x Range[4]]
f2 = Sin[Range[4] x]
```

$$\{e^{x}, e^{2x}, e^{3x}, e^{4x}\}$$

```
{Sin[x], Sin[2 x], Sin[3 x], Sin[4 x]}
```

```
(* multiplication *)
f1 f2
```

$$\{e^{x}\,\text{Sin}[x], e^{2x}\,\text{Sin}[2x], e^{3x}\,\text{Sin}[3x], e^{4x}\,\text{Sin}[4x]\}$$

```
(* dot product *)
f1.f2
```

$$e^{x}\,\text{Sin}[x] + e^{2x}\,\text{Sin}[2x] + e^{3x}\,\text{Sin}[3x] + e^{4x}\,\text{Sin}[4x]$$

```
?x
```

Global`x
x = {1, 2, 3, 4, 5}

```
SeedRandom[1];
x = Table[2 Random[], {2}, {4}, {3}]
```

```
{{{1.33739, 1.6624, 1.56361}, {0.249268, 1.86907, 1.2005},
  {1.51671, 1.93818, 0.251398}, {1.15127, 1.5101, 1.64288}},
 {{1.2092, 0.388929, 1.31461}, {1.11912, 1.67346, 0.321073},
  {1.70887, 1.89087, 1.87436}, {0.521834, 1.45458, 0.558263}}}
```

```
SeedRandom[1];
x = Table[2 Random[], {2}, {4}, {3}]
```

```
{{{1.33739, 1.6624, 1.56361}, {0.249268, 1.86907, 1.2005},
  {1.51671, 1.93818, 0.251398}, {1.15127, 1.5101, 1.64288}},
 {{1.2092, 0.388929, 1.31461}, {1.11912, 1.67346, 0.321073},
  {1.70887, 1.89087, 1.87436}, {0.521834, 1.45458, 0.558263}}}
```

```
√x
```

```
{{{1.15645, 1.28934}, {1.25045, 0.499268}, {1.36714, 1.09568}},
 {{1.23155, 1.39218}, {0.501396, 1.07297}, {1.22886, 1.28175}},
 {{1.09964, 0.623642}, {1.14657, 1.05788}, {1.29362, 0.566633}}}
```

```
matrix = Table[{r, c}, {r, 3}, {c, 3}]
```

```
{{{1, 1}, {1, 2}, {1, 3}},
 {{2, 1}, {2, 2}, {2, 3}}, {{3, 1}, {3, 2}, {3, 3}}}
```

```
matrix // MatrixForm
```

$$\begin{pmatrix} \binom{1}{1} & \binom{1}{2} & \binom{1}{3} \\ \binom{2}{1} & \binom{2}{2} & \binom{2}{3} \\ \binom{3}{1} & \binom{3}{2} & \binom{3}{3} \end{pmatrix}$$

```
matrix = Table[ToString[{r, c}], {r, 3}, {c, 3}];
matrix // MatrixForm
```

```
⎛ {1, 1}  {1, 2}  {1, 3} ⎞
⎜ {2, 1}  {2, 2}  {2, 3} ⎟
⎝ {3, 1}  {3, 2}  {3, 3} ⎠
```

There is even a facility to generate lists using functions:

array[sin]

Note that these values are held as symbols; they are not evaluated. To do that we need to use the *numeric* operator N in prefix or postfix notation.

If we want to see the first 20 digits, we can use the prefix form as shown here:
N[z,20]

Functions in *Mathematica* are polymorphic — they perform the same function for different inputs. Consider the sine functioning on this list.

Let's correlate the outputs to the inputs. The output Sin[1] is a symbol right now. We could force numeric evaluation, but a secret to arbitrary precision is to evaluate all symbols at the end of a calculation. Sin[pi] gives the expected exact answer of 0, not o., a zero with a period signalling a zero to 16 or 17 digits. We see that the sign of p to single precision gives us a zero to single precision. Finally the Sin[Dan] is being held as just that until we can specify how to interpret Dan.

Notice what happens when we omit the brackets around the list. *Mathematica* will repeat the input as a signal that it doesn't know how to interpret these instructions.

As always we encourage the reader to experiment and probe with the tools at hand. Let's examine the pedigrees of the inputs to the sine function.
Head

We see an Integer, a Symbol, a Real and another Symbol. The first Symbol is p and can be evaluated by *Mathematica* at the end of the computation. The second symbol Dan will be carried along too just like pi until the end. Hopefully by that time the user will have supplied some definition.

Now let's look at the output types. The **Head** command tells us what data type *Mathematica* "sees" when it encounters an expression like this. This reveals a little bit about how *Mathematica* works. Let's discuss these results out of order. Head[Sin[p]] is an Integer. This means that *Mathematica* has immediately evaluated this special case to an integer zero. The Head[Sin[3.1415926]] is a real, telling us that *Mathemat-*

ica has substituted the single precision zero answer. Head[Sin[1]] is a Symbol telling us that *Mathematica* will hold the symbolic value until tasked by the numeric operator. Similarly, the Head query tells us that Sin[Dan]] is a symbol and there will be no attempt at evaluation until by the N[].

Let's build up some functions using lists. We'll build a list f1 which will be Exp[n x] and a list f2 which will be Cos[n x]. This is the height of simplicity.
f1
f2
Now let's manipulate these lists. Notice the difference between the multiplication and the dot product.

You may think the difference between a multiplication and a dot product is manifest, but confusing these quantities is not uncommon for new users. Note that the list argument is not restricted to one-dimensional lists.

1.4.6 The square root operator sweeps through them all.

We call your attention to the **SeedRandom[]** command. This very important command allows you to start a particular string of random numbers. This is very important when debugging your code and verifying your results. For example, say you have a computationally intensive code that suddenly yields an unexpected answer. Is it an algorithm problem or a data problem? If you have seeded your random number generator, then you have the ability to trace through the code.

The subtlety missed is that while you want random numbers to simulate real measurements, you want your computer code to be *exactly* repeatable. Consider that you have verified and validated your code. Later circumstances compel you to make minor changes which should not affect the operation of the code. How could you prove that the code is unchanged if you are unable to reproduce your previous answers?

As you might imagine, matrices are particularly simple to represent with lists. Let's create a matrix that shows us the row and column address for each entry.

flat matrix

This is how *Mathematica* "sees" the matrix. But we want it in a more familiar form.

vector

What we see is a list of vectors since the {r,c} list is actually a vector itself presented in proper form. However, to force the form into a {row, column} form, we will use the **ToString[]** operator as shown here.

⚠ At small dimensions *Mathematica* automatically handles vectors and covectors.

List gymnastics

One of the great joys about working with *Mathematica* is the tool kit for list operations. We can natively perform all the important operations that we learned in linear algebra. Experience shows that the list gymnastics are often the most unfamiliar exercises in *Mathematica*. We'll work through a few examples to clarify this. If you are still uncertain about how to use lists, keep reading because we will show how to do most operations without the list tools.

Let's begin with two short sample lists which we will call *x* and *y*.

```
(* List gymnastics *)
x = Table[2 n + 1, {n, 0, 4}]
y = CharacterRange["α", "ε"]

{1, 3, 5, 7, 9}
```

```
{α, β, γ, δ, ε}
```

Very often we will need to take these two lists and merge them into a single list of points $\{x, y\}$. A do-loop can do this for us easily enough. Schematically the process looks like this.

```
count the number of terms (nx);
create a container to hold the {x, y} data (z);
 loop through the data and create an {x, y} pair;
 add the {x, y} pair to the container z;
display container contents;
```

A *Mathematica* implementation of this process is this.

```
(* count the number of elements *)
nx = Length[x];
(* empty container to hold {x, y} pairs *)
z = {};
(* loop through the elements of the x and y lists *)
Do[
  (* add the {x, y} pair to the container *)
  AppendTo[z, {x[[i]], y[[i]]}];
  , {i, nx}];
(* show container contents *)
z

{{1, α}, {3, β}, {5, γ}, {7, δ}, {9, ε}}
```

A better way to construct this process is to use the **Table** command. This involves replacing the code:

```
(* empty container to hold {x, y} pairs *)
z = {};
(* loop through the elements of the x and y lists *)
Do[
  (* add the {x, y} pair to the container *)
  AppendTo[z, {x[[i]], y[[i]]}];
  , {i, nx}];
```

with the more succinct:

```
(* construct the table of {x, y} pairs *)
z = Table[{x[[i]], y[[i]]}, {i, nx}];
```

This streamlines and clarifies the process.

```
(* count the number of elements *)
nx = Length[x];
(* construct the table of {x, y} pairs *)
z = Table[{x[[i]], y[[i]]}, {i, nx}];
(* show container contents *)
z

{{1, α}, {3, β}, {5, γ}, {7, δ}, {9, ε}}
```

By way of demonstration we want to show yet another way to fold the data together. We are not recommending this method; we present it to teach the user about a different set of tools.

These tools will be discussed a few more times and for now we simply want to expose the reader to their existence.

The first thing we'll do is multiply a scalar times a vector. How does this process work in *Mathematica*? We encourage the reader to write simple fragments of code whenever in doubt.

```
a {1, 1}

{a, a}
```

```
a {1, 0}

{a, 0}
```

```
a {0, 1}

{0, a}
```

We can use this knowledge then to take a scalar and make a vector. So for the *x* list we just multiply every member of the list by the vector {1,0}.

Again, we could rely upon the do-loop.

```
(* count the number of elements *)
nx = Length[x];
(* empty container to hold {x, 0} pairs *)
x2 = {};
(* loop through the elements of the x list *)
Do[
  (* project out an x vector *)
  AppendTo[x2, x[[i]] {1, 0}];
  , {i, nx}];
(* show container contents *)
x2

{{1, 0}, {3, 0}, {5, 0}, {7, 0}, {9, 0}}
```

However, there is a more compact way using a pure function. Since the notion of a pure function is completely foreign to many users, we will go through a few explicit cases in this book. This is an important tool and you will benefit from being able to use it.

We will write a simple function and watch its action on the lists *x* and *y*. When you look at the function, you'll see a pound sign #, an ampersand &, and the symbols /@. You can always select these symbols and then launch the Help Browser for an explanation; the Browser is quite complete.

The pound sign # is the slot operator and it is a wildcard for the data in the list. The pound sign marks where the data will come in. The ampersand & is the suffix for the function. The /@ is shorthand for the Map command. This will map the function across every element in the list.

```
(* demo pure function *)
fcn = Print["Grabbing ", #] &;
```

What we expect from this formulation is to see the fcn grab each element in order and print its value. Let's apply this function to the *x* list first.

```
(* test on the x list *)
fcn /@x;
```

```
Grabbing  1
Grabbing  3
Grabbing  5
Grabbing  7
Grabbing  9
```

Now the *y* list.

```
(* test on the y list *)
fcn /@y;
```

```
Grabbing  α
Grabbing  β
Grabbing  γ
Grabbing  δ
Grabbing  ε
```

Hopefully at this point you have a grasp of how to apply a pure function to a list. If not, don't worry because we will be addressing this important concept repeatedly.

Let's take all that we have discussed and put it together. We will write two functions: one to inflate *x* values and one to inflate *y* values. We will then be able to add the inflated lists. Here is the composition and operation of the first function.

```
(* inflation method *)
(* create a 0 element in the y slot *)
x2 = (#{1, 0}) &/@x

{{1, 0}, {3, 0}, {5, 0}, {7, 0}, {9, 0}}
```

The *x* values are now embedded in a two-dimensional list. This the first part of an address.

```
(* add a 0 element in the x slot *)
y2 = (#{0, 1}) &/@ y

{{0, α}, {0, β}, {0, γ}, {0, δ}, {0, ε}}
```

Now the *y* values are embedded allowing us to add the lists.

```
(* sum the components *)
z = x2 + y2

{{1, α}, {3, β}, {5, γ}, {7, δ}, {9, ε}}
```

This is the answer that we want, but the purpose of this last exercise was to acclimate the reader a little more to functions, slot operators, and mapping. The proper way to solve the problem is incredibly simple.

The first thing that we will do is create a list using *x* and *y*. We do this in the simplest fashion: with two braces and a comma.

```
(* create a single list *)
zt = {x, y}

{{1, 3, 5, 7, 9}, {α, β, γ, δ, ε}}
```

We could simply state the answer, but we want to show you how to resolve problems, not encourage rote memorization. Let's examine our new list *zt* in **MatrixForm** to visualize its shape.

```
(* examine the matrix form *)
zt // MatrixForm

(1  3  5  7  9)
(α  β  γ  δ  ε)
```

Our problem would be solved if the first row was the first column and the second row was the second column. In other words, if we just had the transpose.

```
(* we just need the transpose *)
z = Transpose[zt]

{{1, α}, {3, β}, {5, γ}, {7, δ}, {9, ε}}
```

That was remarkably easy. To encourage list-based thinking by the reader we will show the solution in `MatrixForm`. To reinforce the elegant simplicity, we will show all the steps needed to take a list of x values and a list of y values and create a list of $\{x, y\}$ points.

```
(* merging separate lists in a list of points *)
Transpose[{x, y}]

{{1, α}, {3, β}, {5, γ}, {7, δ}, {9, ε}}
```

Generating constant vectors

We can count on *Mathematica* to do the right thing. Consider the interactions of vectors and scalars. When we add a scalar to a vector, *Mathematica* adds the scalar to every term.

```
(* vector + scalar *)
y + 1

{1 + α, 1 + β, 1 + γ, 1 + δ, 1 + ε}
```

Multiplication by a scalar gives the expected result.

```
(* scalar times vector *)
2 y

{2 α, 2 β, 2 γ, 2 δ, 2 ε}
```

We view *Mathematica*'s generality as a blessing. At times though it can hinder the new user because for example *Mathematica* is content to add numbers and string and symbols and may not send a error message when there is a problem. This requires a little more attention to programming.

For instance, we have seen codes where the product of vectors was used in lieu of the dot product. They are quite different as you would expect.

```
(* multiplication of vectors *)
{a, b} {c, d}

{a c, b d}
```

```
(* dot product of two vectors *)
{a, b} .{c, d}

a c + b d
```

The relationship that seems most confusing however is this one.

```
(* this fools some users *)
{a, b}
{c, d}

{a/c, b/d}
```

We hope that these rules are clear. However, if you are debugging your code and are unsure, just do what we did: compose simple lists and manipulate them and see whether you find the expected results.

So what happens when we subtract an *n*-dimensional vector from itself? We get an *n*-dimensional zero.

```
(* the zero identity *)
x - x

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

This identity is essential if you want to have true vector arithmetic. It is also a quick way to generate zero vectors if you need them for accumulators. Another popular use comes from least squares problems when you need a one vector of the proper dimension as we shall see in chapter 2.

```
(* creating a vector of 1s *)
x - x + 1

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

⚠ You can use simple vector arithmetic to create constant vectors.

1.5 Programming

Mathematica has a unique and powerful programming language associated with it. Many comment on the similarity with the program languages C++ and Scheme. We think that the interactive environment is an excellent one to learn in because you can view variable states, display intermediate steps, and perform all manner of diagnostic and probe sequences.

Throughout the volume we will stress learning by doing. Sit down at a computer and perform small, quick experiments to teach yourself how the programming language works. We are far, far away from the day when you had to punch computer cards, read them in, and then wait for the line printer to output your job. This is a new paradigm; you have an excellent Help Browser and an interactive environment and you should feel compelled to investigate.

A goal for us is to teach the reader how to perform little tests needed to help sort out functionality and proper syntax. We aren't trying to show you how to solve a problem per se, but trying to show you how you should approach problems in *Mathematica*.

✵ Built-in Functions > Programming

1.5.1 Flow control

There are a rich variety of flow control commands in *Mathematica*. Some are quite simple, some are nuanced. Our tour will be quite brief and will cover the most basic tools needed to get started.

✵ Built-in Functions > Programming > Flow Control

The Do loop

We think that the do-loop is one of the most basic commands and hope that all users can grasp its functioning directly.

Here is a prototype do-loop that will have ten steps as `i` ranges from 1 to 10.

```
(* this is a prototype Do loop *)
Do[
  (* your commands go here *)
  , {i, 10}];
```

The counter syntax is fairly simply. You can start at zero.

```
(* i can start at 0 *)
Do[
  , {i, 0, 10}];
```

Or start at a negative number.

```
(* i can start at a negative number *)
Do[
  , {i, -20, 10}];
```

You can use any numeric variable for the counter, the maximum value and the increment.

```
(* we are not restricted to integers *)
Do[
  , {i, 0.32565, 11.4567, 0.998}];
```

We can step backwards with a decrement. But there is some fine print. For while an omitted increment causes *Mathematica* to use unity, an omitted decrement of -1 will *not* be assumed. The loop will not execute. Perhaps this sounds like minutiae to you. The point we sound often is not to strain your memory, but to investigate. Try this sample loop.

```
(* we can decrement... *)
Do[
 Print["inside the Do loop"];
 , {i, 0, -2}]
```

There was no output statement so we never made it into the loop. Once we explicitly show the decrement the loop will work as shown here.

```
(* ...as long as we specify a decrement *)
Do[
 Print["inside the Do loop"];
 ,{i,0,-2,-1}]
```

```
inside the Do loop
```

```
inside the Do loop
```

```
inside the Do loop
```

We can nest our do-loops — that is, place one inside another.

```
(* we can nest Do loops *)
Do[
  Do[
    Do[
      (* your commands go here *)
      ,{i,n}];
    (* or here *)
    ,{j,n}];
  (* or here *)
  ,{k,n}];
```

This is the more general form for nesting `Do` loops. You can nest as many levels as you wish. If your process allows it, you may be able to use a more compact statement such as this.

```
(* multiple counter syntax *)
Do[
  (* your commands go here *)
  ,{i,n},{j,n},{k,n}];
```

Something about the latter form tends to confuse the new user. Whereas in the former instance the index incrementing in clear, it is not so clear here. In the former case what we expect is that k increments from 1 to n, and then j increments. After k has incremented n^2 times, j has incremented n times and i finally increments. In other words, k is changing the fastest and i is changing the slowest.

When some people see the counters together on one line, they do not always realize which counter is going faster. Here is how someone could sort this out for themselves. Just count the steps and print out the indices.

```
(* sample Do loop to show how multiple counters are used *)
n = 2;    (* controls loop: total iterations = n^3 *)
m = 1;   (* counter *)
(* out indices are incremented first *)
Do[
 (* we can use lists in output statements *)
 Print[m++, ". indices are {i,j,k} = ", {i, j, k}];
 (* indices spin like a odometer: rightmost moves fastest *)
 , {i, n}, {j, n}, {k, n}];
```

```
1. indices  are {i,j,k} = {1, 1, 1}
2. indices  are {i,j,k} = {1, 1, 2}
3. indices  are {i,j,k} = {1, 2, 1}
4. indices  are {i,j,k} = {1, 2, 2}
5. indices  are {i,j,k} = {2, 1, 1}
6. indices  are {i,j,k} = {2, 1, 2}
7. indices  are {i,j,k} = {2, 2, 1}
8. indices  are {i,j,k} = {2, 2, 2}
```

The inquisitive reader may be wondering if one method offers a time advantage over the other. Well, we could state the answer, but we are not writing a collection of facts for memorization. We are writing a manual on how to explore and use *Mathemtica*.

We will compose do-loop in both formats and have them perform a simple task so that we can ensure both loops have performed the same amount of work.

Our next step will be to turn on a timer.

```
<< Utilities`ShowTime`;  (* turn on autotiming *)
```

Now we will compose and run the competing loops.

```
(* single Do loop with 3 counters *)
n = 100;    (* controls loop: total iterations = n^3 *)
p = 0;       (* clear summation value *)
(* single Do loop to control three indices *)
Do[
 p += π;
 , {i, n}, {j, n}, {k, n}];
(* control to insure all
  loops are performing the same operations *)
p
```

```
0. Second
```

```
10.595  Second
```

```
0. Second
```

```
1000000 π
```

The more explicit method is seen on the next page. The significant results are these.

First, both methods successfully return $10^6\pi$. This assures us we successfully coded both loops. (Well, actually it says we similarly coded both loops.) The first method to 10.595 seconds and the second method to 10.465 seconds. This is basically a dead heat. If you run these loops yourself several times, you will see timing differences within each example that suggest 0.13 second difference is not meaningful.

An extension of the multiple counter scenario is to have one counter depend upon another. We'll run into this case later on and for now we will study the basic functioning of the nested do-loop. The prototype form is this.

```
(* index j depends upon index i *)
Do[
 , {i, 0, n}, {j, 0, i}]
```

Are you able to write on a piece of paper, in the proper order, what the values of *i* and *j* will be? Can you predict how many steps there will be as a function of *n*? The first question is one you should be able to answer. If not, you should experiment much more with do-loops before leaving this section. This is a bedrock concept for the rest of this book.

```
(* 3 Do loops for 3 counters *)
n = 100;   (* controls loop: total iterations = n^3 *)
p = 0;      (* clear summation value *)
(* 3 Do loops for 3 indices *)
Do[
 Do[
   Do[
     p += π;
     , {i, n}];
   , {j, n}];
 , {k, n}];
(* control to insure all
  loops are performing the same operations *)
p
```

```
0. Second
```

```
10.465 Second
```

```
0. Second
```

```
1000000 π
```

The trial do-loop is seen atop the next page. We have introduced other elements to.

First, you noticed that we have counted the number of steps with the variable **steps**. Perhaps you know that the sum of the integers from 1 to *n* is:

$$\sum_{i=1}^{n} i = \frac{1}{2}n(n-1) \tag{1.5}$$

Arithmetic Series (formula 3)

In general we find that formulae such as this are a great error source. The starting index may get changed as in this case from one to zero and the summation value is overlooked, or you mistyped the formula, or maybe you never knew the formula. Regardless, your CPU is doing over a billion operations a second and is more than

happy to count them up for you. When *Mathematica* sees the command **step++** , it returns the value of **step** and then adds one to it. If you wished to increment before querying **step**, then you would type **++step**.

Also, we call your attention to the fact that we can put output variables.

```
(* index j depends upon index i *)
step = 1;     (* count the steps *)
n = 2;        (* limiting value for i *)
Do[
 (* diagnostic print *)
 Print[step++, ". {i,j} = ", {i, j}];
 , {i, 0, n}, {j, 0, i}]
```

```
1. {i,j} = {0, 0}
```

```
2. {i,j} = {1, 0}
```

```
3. {i,j} = {1, 1}
```

```
4. {i,j} = {2, 0}
```

```
5. {i,j} = {2, 1}
```

```
6. {i,j} = {2, 2}
```

A successful execution is easy to recognize, but we want to expose the nascent user to error messages most likely to appear. For this case, we will swap the order of the indices.

```
(* index j depends upon index i *)
step = 1;     (* count the steps *)
n = 2;        (* limiting value for i *)
Do[
 (* diagnostic print *)
 Print[step++, ". {i,j} = ", {i, j}];
 , {j, 0, i}, {i, 0, n}]
```

```
⚠ Do::iterb : Iterator {j, 0, i} does not have appropriate bounds . More...

Do[Print[step++, . {i,j} = , {i, j}];, {j, 0, i}, {i, 0, n}]
```

What the error message is saying is that it cannot set up a loop for *j* to go from 0 to *i* because *i* has no value. We know what we meant but *Mathematica* can only search inside the counter (to the left) and not outside (to the right).

As always, *Mathematica* repeats commands it cannot execute. This is a handy way to differentiate between a bad algorithm and a bad command.

We close the section on do-loops with an observation by our colleague P. Riera. Oftentimes people use do-loops to step through some space and make measurements. Say you are moving in small steps δ. You have two choices for keeping track of the measurement position: **x+=** δ and **x = i** δ. Do not accumulate! The first form is a nursery for machine noise as we show here.

```
(* don't accumulate! *)
δ = 0.31415926; (* typical increment amount *)
sum = 0;          (* accumulator *)
Do[
 product = i δ;  (* use this form *)
 sum += δ;       (* not this form *)
 , {i, 10^5}]
Print["the error = ", product - sum];
```

```
the error = -3.92174 ×10^-8
```

After only 10^5 steps we are already down to single precision.

⚠ Do not accumulate in your loops. This is an efficient way to erode precision.

If this is your first exposure to numerical computing, this may surprise you. It is due to the fact that we are trying to represent base 10 numbers in base 2 (binary). To demonstrate this, we can make the problem go away when our increment has an exact binary representation.

```
(* special cases where accumulation works: integers and 2^-n *)
δ = 0.125;        (* 2^-n will work *)
sum = 0;          (* accumulator *)
Do[
 product = i δ;  (* preferred form *)
 sum += δ;       (* acceptable form *)
 , {i, 10^5}];
Print["the error = ", product - sum];
```

```
the error = 0.
```

Don't flirt with the special cases. Make it a practice to avoid accumulation.

Built-in Functions > Programming > Flow Control > Do

The For loop

While we believe the do-loop is the workhorse for flow control, we do want to expose readers to the **For** loop. We show here an equivalent formulation for the triple index do-loops we presented.

```
(* For loop replicating the Do loops *)
n = 100;    (* controls loop: total iterations = n^3 *)
p = 0;       (* clear summation value *)
For[i = 1, i ≤ n,
 For[j = 1, j ≤ n,
  For[k = 1, k ≤ n,
   p += π
   , k++]
  , j++]
 , i++]
(* control to insure all
  loops are performing the same operations *)
p
```

```
0. Second
```

```
13.59 Second
```

```
0. Second
```

```
1000000 π
```

We see that this form is appreciably slower (about 3 seconds) and is a bit harder to read. But if you are comfortable with this syntax, you should use it.

Built-in Functions > Programming > Flow Control > For

While

The **While** command has some interesting uses and we encourage readers to ponder ways to integrate this command into their programs. The syntax is the height of simplicity.

```
While[test,
  (* commands *)
 ];
```

An excellent case for this problem is the random walk. Our variant of the problem is this. We start at the origin and take a series of unit steps, each in a random direction. We stop when we have reached some predetermined distance from the origin.

Say that we want to keep going until we have traveled the distance 15. Our basic structure would look like this.

```
(* the random walk process *)
While[distance ≤ 15,
  increment step counter;
  create a random angle;
  take the step;
  compute the distance;
 ];
```

We are trying to encourage users to reduce the processes to separate and simple statements that can be tested before putting the whole statement together. Far too often have we been asked to unravel some Gordian knot of code suffering from a host of maladies, all of which would have been exposed had the pieces been tested independently.

```
(* the random walk problem *)
(* start at the origin *)
(* steps are unit size in a random direction *)
d = -∞;           (* total distance traveled: controls While *)
k = 0;            (* count the steps *)
SeedRandom[1];   (* always seed for repeatability *)
p = {0, 0};       (* position vector *)
(* keep walking until we are 15 units away *)
While[d ≤ 15,
 k++;           (* increment step counter *)
 θ = Random[Real, {0, 2 π}];  (* direction in radians *)
 p += {Cos[θ], Sin[θ]};       (* take a step *)
 d = √(Plus @@ p^2);             (* distance from origin *)
];
(* output number of steps and how far we went *)
Print[k, " iterations to go a distance ", d];
```

```
93 iterations  to go a distance  15.0538
```

In the interest of expediency we present the **While** loop after assembling tested components. We call your attention to the following features.

First, we were careful to give *d* a numeric value before we started. One thing we love about *Mathematica* are the values $\pm\infty$. They make comparisons so easy and they prevent annoying bugs. We have seen users use what they consider to be a large number to control feedback loops but they have found out the hard way that there are legitimate values in excess of their bound.

⚠ Take advantage of the infinities in *Mathematica* $(\pm\infty)$ to ensure that your comparison functions as desired.

When we replace the statement

```
d = -∞;           (* total distance traveled: controls While *)
```

with the statement:

```
Clear[d];       (* simulates forgetting to assign d a value *)
```

this simulates what would happen if you forgot to assign a value to the test variable. This produces the output:

```
0 iterations  to go a distance  d
```

which, we know from experience, means that the While loop was never entered. What happened? Of course we could tell you, but we want to show you how to decipher this on your own. Your first recourse may be an If test similar to the one employed by the While loop.

```
(* probe statement to test the control variable *)
If[d ≤ 15, Print["d ≤ 15"], Print["d > 15"], Print["huh?"]];
```

```
huh ?
```

For better or for worse we have replicated the failed numerical comparison. Our next step would be to probe d. We need to see why it failed the test. Our commands are ? and Head[].

```
Head[d]
?d

Symbol
```

```
Global`d
```

We see that *d* is a symbol with no value. *Mathematica* has a very sophisticated classification and what it is saying is that it cannot tell whether *d* is less than or equal to 15 because it is a symbol without a value. This may seem abstract, so let's be a bit more definite with an example where we compare a string to a number.

```
(* probe statement to test the control variable *)
d = "I am a string";
If[d ≤ 15, Print["d ≤ 15"], Print["d > 15"], Print["huh?"]];
```

```
huh ?
```

Mathematica is saying that when we try to compare a string to a number we cannot say either True or False. We simply cannot make the comparison.

‹ Make sure that your test variable has a value when you first enter the **While** loop.

The final thing we call your attention to is something that we will state often. Whenever you use a random series of numbers in your code, seed the random number generator with SeedRandom[]. This may seem counterintuitive, but remember, the random number generator simulates random conditions. You don't want your algorithm to behave randomly, you want your data to behave randomly. If you have not seeded your sequence, your validation may be impossible. For example, suppose you changed your code in a way that should not have altered the computations. How would you show that your code behaves the same? If you have seeded your data, you can run the same sequences again and look for the same results.

Another reason is that many times we have watched users look at their output and get unexpected or "impossible" answers. How can you know if the problem is your code or if your problem is the data or if the problem is your understanding of the algorithm? If you have seeded your data you can repeat the sequence.

Also, we are very pleased to note that different platforms will supply the same sequence of random numbers if you seed with a definite value. That means that you can continue testing and development on different platforms.

‹ Make sure you always seed the random number generator with a definite value.

With these points said, we will close our discussion on flow control. We do want to state that we will be visiting the random walk problem again in this chapter to develop a graphical output form.

Ÿ Built-in Functions > Programming > Flow Control > Which

1.6 Standard add-on packages

Mathematica offers a large number of standard add-on packages to extend its capabilities. This is a buffet style structure — you can take whatever you want. You should take a few minutes to examine this list. Many users ignore this list and end up attempting to duplicate the function found in these packages.

In addition to providing a comprehensive environment for calculations and a programming language, *Mathematica* is also a system for representing and presenting scientific and technical knowledge. Certain packages are included with *Mathematica* to provide easy access to commonly used scientific data, such as the value of physical constants and conversion factors for various systems of units

So how do we access packages?

Package category	Packages included		
Algebra	AlgebraicInequalities	PolynomialExtendedQCD	ReIm
	FiniteFields	PolynomialPowerMod	RootIsolation
	Horner	Quaternions	SymmetricPolynomials
	InequalitySolve		
Algebra ReIm	AlgebraicInequalities	PolynomialExtendedQCD	
Calculus	DSolveIntegrals	VariationalMethods	Integration
	FourierTransform	VectorAnalysis	Pade
DiscreteMath	CombinatorialFunctions	ComputationalGeometry	RSolve
	Combinatorica	DiscreteStep	Tree
Geometry	Polytopes	Rotations	
Graphics	Animation	Graphics3D	PlotField3D
	ArgColors	ImplicitPlot	Polyhedra
	Arrow	InequalityGraphics	Shapes
	Colors	Legend	Spline
	ComplexMap	MultipleListPlot	SurfaceOfRevolution
	ContourPlot3D	ParametricPlot3D	ThreeScript
	FilledPlot	PlotField	Common
	Graphics		
LinearAlgebra	FourierTrig	Orthogonalization	Tridiagonal
	MatrixManipulation		
Miscellaneous	Audio	StandardAtmosphere	Geodesy
	BlackBodyRadiation	Music	Units
	Calendar	PhysicalConstants	WorldData
	ChemicalElements	RealOnly	WorldNames
	CityData	ResonanceAbsorptionLines	WorldPlot
NumberTheory	AlgebraicNumberFields	NumberTheoryFunctions	Rationalize
	ContinuedFractions	PrimeQ	Recognize
	FactorIntegerECM	PrimitiveElement	SiegelTheta
	NumberTheoryFunctions		
NumericalMath	Approximations	IntervalRoots	NSeries

	BesselZeros	ListIntegrate	Newton-Cotes
	Butcher	Microscope	OrderStar
	CauchyPrincipalValue	NIntegrateInterpolatingFunct	PolynomialFit
	ComputerArithmetic	NLimit	SplineFit
	GaussianQuadrature	NResidue	TrigFit
	InterpolateRoot		
Statistics	ANOVA	DiscreteDistributions	MultinormalDistribution
	ConfidenceIntervals	HypothesisTests	NonlinearFit
	ContinuousDistributions	LinearRegression	NormalDistribution
	DataManipulation	MultiDescriptiveStatistics	StatisticsPlots
	DataSmoothing	MultiDiscreteDistributions	Common
	DescriptiveStatistics		
Utilities	BinaryFiles	MemoryConserve	ShowTime
	FilterOptions	Package	

✵ Add-ons & Links > Standard Packages

An important point to remember is that this list is growing. If you have a Premier subscription, then you will automatically receive the distributed packages. Other packages are available through the *Mathematica* Information Center at library.wolfram.com.

Because of the diversity of the packages in the Miscellaneous category, we would like to spend some time to familiarize the reader with the sundry functions.

1.7 Miscellaneous packages

Most users are surprised at the depth and scope of the add-on packages to *Mathematica*. For example, it is not uncommon for people to write code to perform vector operations even though the Calculus‘VectorAnalysis package has a rather formidable tool kit. Hopefully the list above will prevent someone from writing code for routines they already have.

The catchall category, Miscellaneous, has some intriguing packages and we want to take some time to scan through some of them. Again the hope is that this will preclude readers from trying to replicate this work themselves. We will separate the packages with the command needed to load them.

1.7.1 Miscellaneous'Audio

```
(* load the audio package *)
<< Miscellaneous`Audio`
```

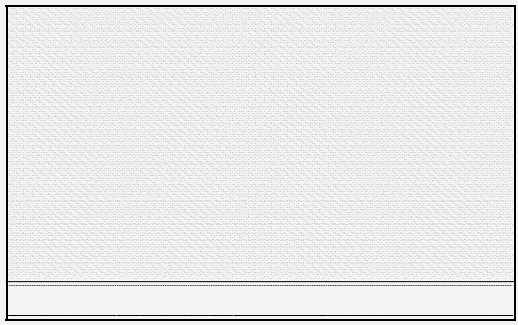
The Audio package provides the user with a tool kit to create standard and custom wave forms. The command below will create a sawtooth form with a fundamental frequency of 700 Hz lasting for 3 seconds. Notice that the command creates a sound object, but does not generate the sound.

```
noise = Waveform[Sawtooth, 700, 3]

- Sound -
```

To hear this rather annoying sound we use the **Show** command as shown here. You may have expected a command like **Hear**, but keeping in the spirit of generality, **Mathematica** has a rather universal **Show** command.

```
s1 = Show[noise];
```



The Help Browser has some excellent background material which we show below. Also, there are some very interesting sound demos in the Demos, including the tones for the touch tone phones.

- Add-on & Links > Standard Packages > Miscellaneous > Audio
- Add-on & Links > Standard Packages > Miscellaneous > Music
- The Mathematica Book > A Practical Introduction to Mathematica > Graphics and Sound > Sound
- Demos > Sound Gallery > Touch Tones

We have calculated peak wavelength above and we can check it somewhat crudely on the chart. Select the graphic object above by clicking on it. Your cursor will then take on the appearance of four arrows. Then hold down the Alt key (⌘ or command key on the Macintosh) and you will be able to read the coordinates in the lower left-hand corner of the notebook window. The result will be in the coordinates of the plot. This is an extremely useful feature for all two-dimensional plots that many users seem to miss.

In this particular case the measurement seems quite crude. We placed the cursor on the spot where the dotted line intercepted the wavelength axis and the coordinate as 8.06×10^{-7}. However, we understand that cursor movement is quantized by the pixels, so we shifted the cursor one pixel to the right and the crosshairs were clearly beside the dotted line and the value was 8.18×10^{-7}, a noticeable discrepancy from the value of 8.19×10^{-7}.

✵ Add-on & Links > Standard Packages > Miscellaneous > BlackBodyRadiation

⚠ Be somewhat wary of using the cursor to measure graphics coordinates. There seem to be some small errors.

1.7.2 Miscellaneous‘Calendar‘

```
(* H B.: a unififed treatmen of basic calendar operations *)
<< Miscellaneous`Calendar`
```

It is somewhat surprising to see how complete a tool kit Wolfram provides for calendrical calculations. However, we must surmise that users doing financial calculations and users doing celestial mechanics are quite pleased with the capabilities. For example, we can find out what day of the week it was on July 4, 1776;

```
DayOfWeek[{1776, 7, 4}]

Thursday
```

On May 25, 1961 President Kennedy addressed a joint session of Congress and promised to send a man to the moon and return him safely. We can count the number of days between this speech and Neil Armstrong's famous moonwalk easily

```
DaysBetween[{1961, 5, 25}, {1970, 7, 20}]

3343
```

The Mars Polar Lander was launched on January 1, 1999. Its voyage lasted 334 days. What day was it due to land on Mars?

```
DaysPlus[{1999, 1, 3}, 334]

{1999, 12, 3}
```

There are tools to convert between Gregorian and Julian calendars as well as the Islamic calendar. You can also compute special days.

In 325 A.D. the Roman emperor Constantine convened the Council of Nicea and established that Easter will be the first Sunday that occurs after the first full moon on or after the vernal equinox. However, this mandated an ecclesiastical full moon and equinox which differ from their astronomical counterparts used today. The ecclesiastical computations don't account for the full complexity of lunar motion which is defined as the 14th day of tabular lunation. The net effect is to constrain Easter to fall between March 22 and April 25. These Gregorian tables are used by the Roman Catholic Church and the Protestants. The Eastern or Orthodox Christian Church uses the Julian tables.

It may sound confusing, but it is all sorted out in *Mathematica*. Also, the Naval Observatory gives the day of Easter close attention. You can check the listing at the URL below.

http://aa.usno.navy.mil/faq/docs/easter.html

We can generate our own table easily.

```
(* check http://aa.usno.navy.mil/faq/docs/easter.html *)
es = Table[EasterSunday[d], {d, 1980, 2024}]

{{1980, 4, 6}, {1981, 4, 19}, {1982, 4, 11}, {1983, 4, 3}, {1984, 4, 22},
 {1985, 4, 7}, {1986, 3, 30}, {1987, 4, 19}, {1988, 4, 3}, {1989, 3, 26},
 {1990, 4, 15}, {1991, 3, 31}, {1992, 4, 19}, {1993, 4, 11}, {1994, 4, 3},
 {1995, 4, 16}, {1996, 4, 7}, {1997, 3, 30}, {1998, 4, 12}, {1999, 4, 4},
 {2000, 4, 23}, {2001, 4, 15}, {2002, 3, 31}, {2003, 4, 20}, {2004, 4, 11},
 {2005, 3, 27}, {2006, 4, 16}, {2007, 4, 8}, {2008, 3, 23}, {2009, 4, 12},
 {2010, 4, 4}, {2011, 4, 24}, {2012, 4, 8}, {2013, 3, 31}, {2014, 4, 20},
 {2015, 4, 5}, {2016, 3, 27}, {2017, 4, 16}, {2018, 4, 1}, {2019, 4, 21},
 {2020, 4, 12}, {2021, 4, 4}, {2022, 4, 17}, {2023, 4, 9}, {2024, 3, 31}}
```

In this form comparison with the website is quite tedious. This gives us a good reason to demonstrate the benefits of `TableForm` as you see below.

```
(* easier to compare to the Navy table *)
TableForm[Table[EasterSunday[d], {d, 1980, 2024}]]
```

```
1980    4    6
1981    4    19
1982    4    11
1983    4    3
1984    4    22
1985    4    7
1986    3    30
1987    4    19
1988    4    3
1989    3    26
1990    4    15
1991    3    31
1992    4    19
1993    4    11
1994    4    3
1995    4    16
1996    4    7
1997    3    30
1998    4    12
1999    4    4
2000    4    23
2001    4    15
2002    3    31
2003    4    20
2004    4    11
2005    3    27
2006    4    16
2007    4    8
2008    3    23
2009    4    12
2010    4    4
2011    4    24
2012    4    8
2013    3    31
2014    4    20
2015    4    5
2016    3    27
2017    4    16
2018    4    1
2019    4    21
2020    4    12
2021    4    4
2022    4    17
2023    4    9
2024    3    31
```

Add-on & Links > Standard Packages > Miscellaneous > Calendar

1.7.3 Miscellaneous‘ChemicalElements‘

```
(* a database of chemical properties *)
<< Miscellaneous`ChemicalElements`
```

Here we find the chemical elements and their properties in a form more amenable to programming. We can query the atomic weight using either the full element name

```
(* you can use the full element name *)
AtomicWeight[Gadolinium]

157.25
```

or the abbreviation:

```
(* you can use the abbreviation *)
AtomicWeight[Gd]

157.25
```

As we will do in the section on palettes, we can find the boiling point in degrees Kelvin.

```
(* assign a name to the boiling point temperature *)
bp = BoilingPoint[Gd]

3539. Kelvin
```

Let's suppose that we want the value but not the units. In other words, we want the pure number 3539. How would we isolate that? Our first step is to look at the dimension of the number:

```
(* we see a number with units and wonder how this is assembled *)
Dimensions[bp]

{2}
```

We see that we have two items, a value and a unit.

```
(* let's look at the two components separately *)
bp[[1]]
bp[[2]]

3539.
```

```
Kelvin
```

We can use the **Part** command in shorthand as shown above or in command form

```
(* command form of bp[[1]] *)
Part[bp, 1]

3539.
```

We can use a zero index with the Part command to find the Head of the expression.

```
(* we can use Part to get the Head of the expression *)
Part[bp, 0]

Times
```

As always, it is good practice to check our results.

```
(* verification *)
Head[bp]

Times
```

This leads us to wonder what is the exact form of a number with dimensions?

```
(* what is the structure of bp? *)
FullForm[bp]

Times[3539.`, Kelvin]
```

So given a list we can always **Take** and element or range of elements. An example is shown here.

```
(* we can take an element from the list *)
Take[bp, 1]

3539.
```

Equivalently we can **Extract** elements from the list.

```
(* we can extract an element from the list *)
Extract[bp, 1]

3539.
```

Yet another syntax would be to **Drop** elements from the list.

```
(* we can shave off (drop) parts of the list *)
(* here the negative index means that we count from the end *)
Drop[bp, -1]

3539.
```

Here we used a negative index to count from the end, not the beginning.

Ÿ Built-in Functions > Lists and Matrices > Element Extraction

1.7.4 Miscellaneous'BlackBodyRadiation'

```
(* basic calculations for black bodies *)
<< Miscellaneous`BlackBodyRadiation`
```

Plank's postulation of quantized oscillators successfully explained the energy spectrum of blackbody radiation and its variation with temperature. The Black Body Radiation tool kit computes properties and displays spectral distributions for black bodies whose temperatures are given in Kelvin. Consider a blackbody at the temperature of boiling gadolinium shown below.

```
(* substitution to extract boiling point temperature *)
T = BoilingPoint /. gd

3539. Kelvin
```

The wavelength at the peak of the spectral distribution is:

```
PeakWavelength[T]

8.19441 × 10^-7 Meter
```

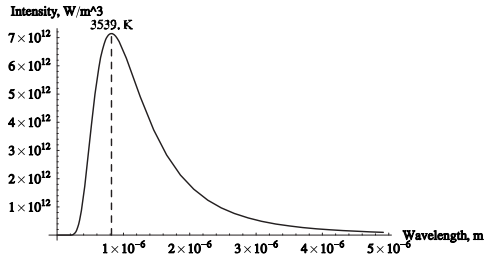
In a moment we will actually plot this distribution and use *Mathematica* cursor tools to measure the peak. First though we will compute the total power emitted

```
TotalPower[T]

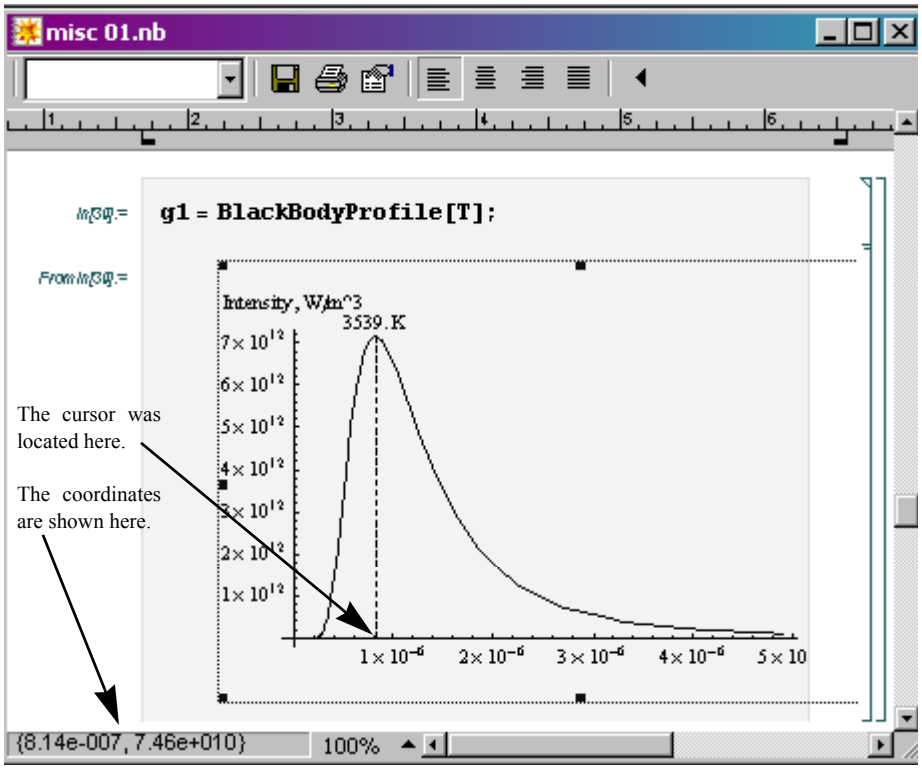
8.89479 × 10^6 Watt / Meter^2
```

Routines in this package will automatically generate the spectral distribution.

```
g1 = BlackBodyProfile[T, ImageSize → 5 72];
```



Recently Wolfram Research has installed tools that allow you to position the cursor on figures and read the coordinates of the points. We show an example of this process below. Unfortunately, the Print Screen process has removed the cursor.



```
(* load the package of common physical constants *)
<< Miscellaneous`PhysicalConstants`
```

This package has a nice assortment of physical constants in SI units. To save you the trouble of determining how current they are, we show their values.

```
AccelerationDueToGravity       9.80665 Meter/Second^2
AgeOfUniverse                  4.7×10^17 Second
AvogadroConstant               6.02214×10^23 /Mole
BohrRadius                     5.29177×10^-11 Meter
BoltzmannConstant              1.38065×10^-23 Joule/Mole
ClassicalElectronRadius        2.81794×10^-15 Meter
CosmicBackgroundTemperature    2.726 Kelvin
DeuteronMagneticMoment         4.33073×10^-27 Joule/Tesla
DeuteronMass                   3.34358×10^-27 Kilogram
EarthMass                      5.9742×10^24 Kilogram
EarthRadius                    6378140 Meter
ElectronCharge                 1.60218×10^-19 Coulomb
ElectronComptonWavelength      2.42631×10^-12 Meter
ElectronGFactor                -2.00232
ElectronMagneticMoment         -9.28476×10^-24 Joule/Tesla
ElectronMass                   9.10938×10^-31 Kilogram
FaradayConstant                96485.3 Coulomb/Mole
FineStructureConstant          0.00729735
GalacticUnit                   2.6×10^20 Meter
GravitationalConstant          6.673×10^-11     Meter^2    Newton/
                                 Kilogram^2
HubbleConstant                 3.2×10^-18 /Second
IcePoint                       273.15 Kelvin
MagneticFluxQuantum            2.06783×10^-15 Kilogram
MolarGasConstant               8.31447 Joule/Kelvin Mole
MolarVolume                    0.022414 Meter^2/Mole
MuonGFactor                    -2.00233
MuonMagneticMoment             -4.4905×10^-26 Joule/Tesla
MuonMass                       1.88353×10^-28 Kilogram
NeutronComptonWavelength       1.31959×10^-12 Meter
NeutronMagneticMoment          -9.66236×10^-27 Joule/Tesla
NeutronMass                    1.67493×10^-27 Kilogram
PlanckConstant                 6.62607×10^-34Joule Second
PlanckConstantReduced          1.05457×10^-34Joule Second
PlanckMass                     2.1767×10^-8 Kilogram
ProtonComptonWavelength        1.32141×10^-15 Meter
ProtonMagneticMoment           1.41061×10^-26 Joule/Tesla
ProtonMass                     1.67262×10^-27 Kilogram
QuantizedHallConductance       0.0000387405 Ampere/Volt
```

RydbergConstant	1.09737×10^{7} /Meter
SackurTetrodeConstant	-1.1517
SolarConstant	1373. Watt/Meter^2
SolarLuminosity	3.86×1026 Watt
SolarRadius	6.9599×10^{8} Meter
SolarSchwarzschildRadius	2953.25 Meter
SpeedOfLight	299792458 Meter/Second
SpeedOfSound	340.292 Meter/Second
StefanConstant	5.6704×10^{-8} Watt/Kelvin^4 Meter^2
ThomsonCrossSection	6.65246×10^{-29} Meter
VacuumPermeability	π Second Volt/(2,500,000 Ampere Meter)
VacuumPermittivity	8.85419×10^{-12}Ampere Second/Meter Volt
WeakMixingAngle	0.2224

A greatly expanded package is available from a Wolfram website as 'StandardPhysicalConstants'. We show the web address below.

http://library.wolfram.com/infocenter/Conferences/4993/

1.8 Palettes

Palettes are a great way to extend the functionality of *Mathematica*. We will not spend a lot of time on this but we will refer to some innovative examples available to you.

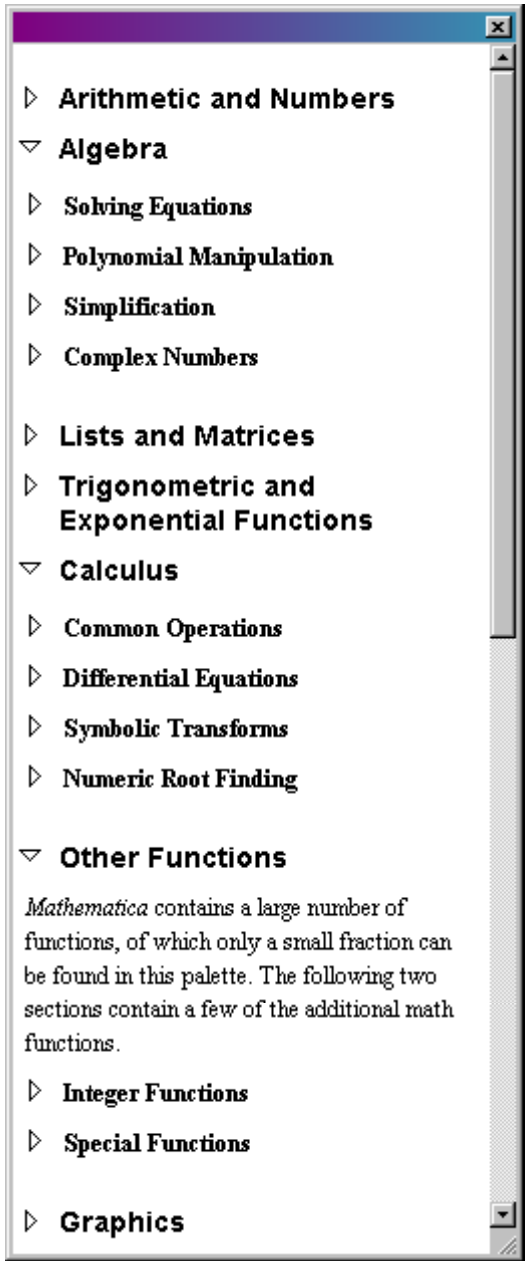
1.8.1 Standard palettes

Mathematica has many useful palettes to simplify our lives. To access the palettes, go through the menu bar File > Palettes and you will see the following list of nine palettes:

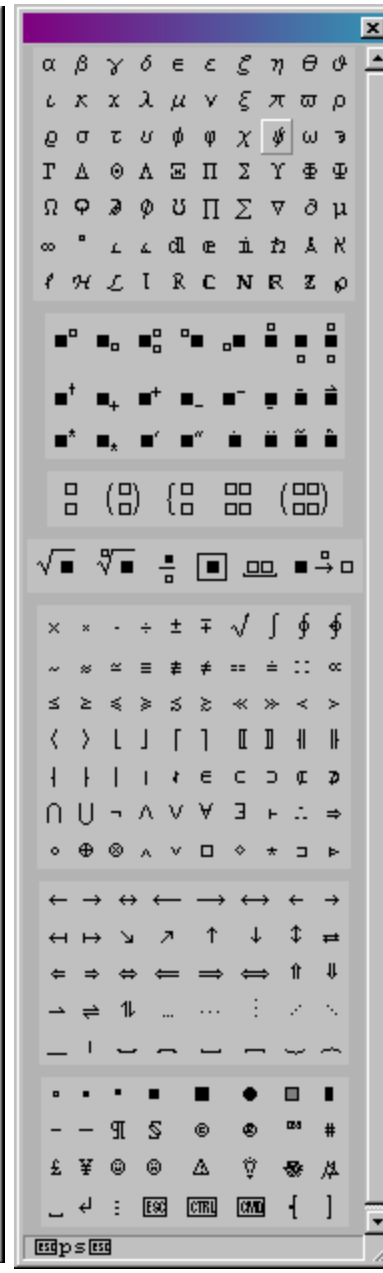
OpenAuthorTools	BasicInput	CreateSlideShow
AlgebraicManipulation	BasicTypesetting	InternationalCharacters
BasicCalculations	CompleteCharacters	NotebookLauncher

Unfortunately, not all of these palettes display their titles. For this reason, and to acquaint the reader with their functional capability we will show all the palettes below.

These palettes are "sticky" in the sense that their state is recorded when *Mathematica* closes and the environment is recreated when *Mathematica* is launched. In other words if you have three palettes open when you close *Mathematica*, the same three, and only these three will be opened when *Mathematica* is launched. The demo palettes listed below are not sticky and they need to be opened each time you use them.



BasicCalculation palette



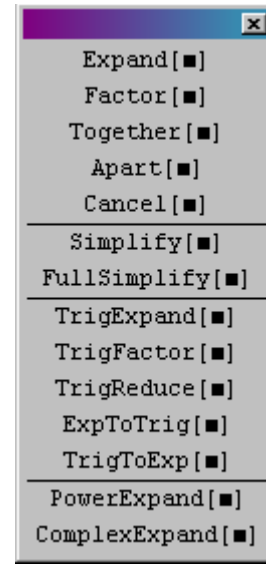
BasicTypsetting palette
(active legend)



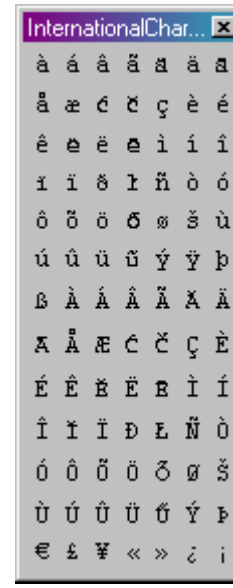
CompleteCharacters palette
(active legend)



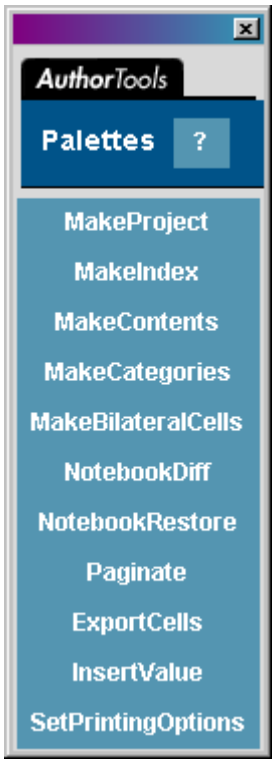
BasicInput palette
(active legend)



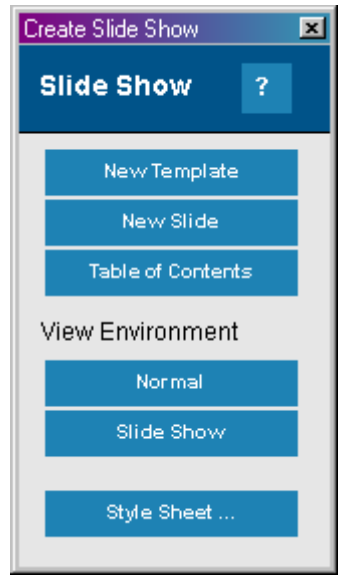
AlgebraicManipulation
palette



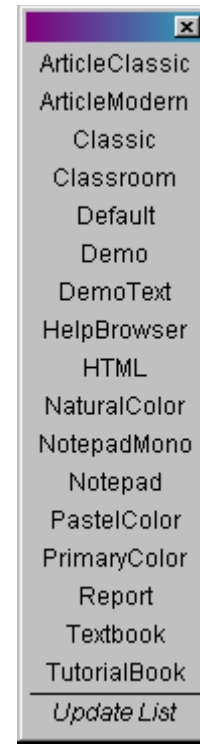
InternationalCharacters
palette



OpenAuthorTools palette



CreateSlideShow palette



NotebookLauncher palette

Palettes are a great way to extend the functionality of *Mathematica*. The only limitation is your imagination. We will discuss a few of the examples included the Help Browser demos.

Basic Input
Basic Calculations
Algebraic Manipulation
Color Palette
International Characters
Polyhedon Explorer
Periodic Table
Physical Constants
Notebook Launcher
Selection Mover
Demo Maker
Basic Typesetting
Complete Characters

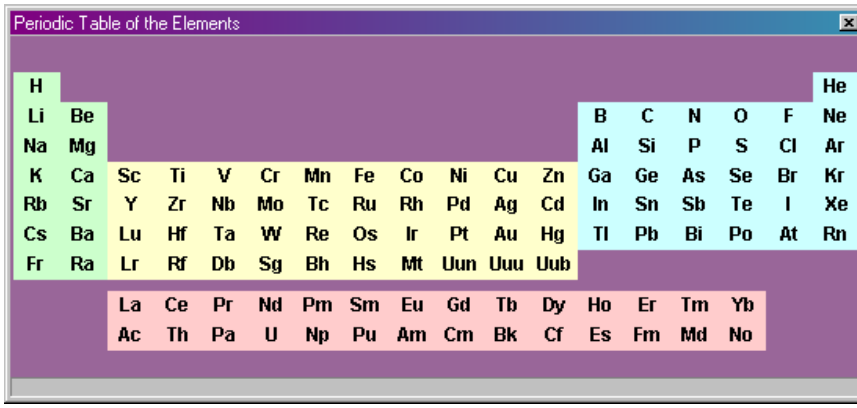
Demos > Palettes

1.8.2 Demo palettes

Periodic table of the elements

One of the most interesting palettes is for the periodic table of elements. To access this palette, go through the Help Browser, Demos > Palettes and select Periodic Table. The palette looks something like this.

How do we harvest the information on this palette? Suppose, for example, we wanted to capture the properties of the rare earth metal gadolinium. We could first type a null assignment like this.




```
(* set up an assignment *)
gd=;
```

Then we would click between the equals sign and the colon. This will plant the cursor at the insertion point and we will be able to see the cursor blink. Then we slide over to the palette and double click on element 64, gadolinium. Notice that when you mouse over an element, the name and number are displayed on the bottom of the palette. (Gd is in the penultimate row, eighth from the left.)

When you double click on an element, a handy table is created in the target notebook. When you assign this information to an object, here named `gd`, you have this information at your fingertips.

```
gd =
```

Gadolinium (Gd)	64
AtomicNumber: 64	
AtomicWeight: 157.25	
StableIsotopes: {154, 155, 156, 157, 158, 160}	
ElectronConfiguration $1s^2\ 2s^2\ 2p^6\ 3s^2\ 3p^6\ 3d^{10}$ $4s^2\ 4p^6\ 4d^{10}\ 4f^7\ 5s^2\ 5p^6\ 5d^1\ 6s^2$	
Density: 7900.4 kg m^{-3}	
ThermalConductivity: 10.6 W K^{-1} m^{-1}	
MeltingPoint: 1586 K	
HeatofFusion: 15.5×10^3 J mol^{-1}	
BoilingPoint: 3539 K	
HeatofVaporization: 301×10^3 J mol^{-1}	

;

```
In[4]:= (* set up an assignment *)
```

```
gd =
```

Gadolinium (Gd)	64
Atomic Number : 64	
Atomic Weight : 157.25	
Stable Isotopes : {154, 155, 156, 157, 158, 160}	
Electron Configuration $1s^2\ 2s^2\ 2p^6\ 3s^2\ 3p^6\ 3d^{10}$ $4s^2\ 4p^6\ 4d^{10}\ 4f^7\ 5s^2\ 5p^6\ 5d^1\ 6s^2$	
Density : 7900.4 kg m^{-3}	
Thermal Conductivity : 10.6 W K^{-1} m^{-1}	
Melting Point : 1586 K	
Heat of Fusion : 15.5×10^3 J mol^{-1}	
Boiling Point : 3539 K	
Heat of Vaporization : 301×10^3 J mol^{-1}	

;

All the vital information is available as substitution rules. Since this seems to be a concept that requires reinforcement, we will show how to harvest the data at a few places throughout the book. First, we show what the substitution rules look like.

```
gd
```

$$\left\{\text{Abbreviation} \to \text{Gd},\ \text{AtomicNumber} \to 64,\ \text{AtomicWeight} \to 157.25,\right.$$
$$\text{StableIsotopes} \to \{154, 155, 156, 157, 158, 160\},$$
$$\text{ElectronConfiguration} \to \{\{2\}, \{2, 6\}, \{2, 6, 10\}, \{2, 6, 10, 7\}, \{2, 6, 1\}, \{2\}\},$$
$$\text{MeltingPoint} \to 1586.\,\text{Kelvin},\ \text{BoilingPoint} \to 3539.\,\text{Kelvin},$$
$$\text{HeatOfFusion} \to \frac{15.5\,\text{Joule}\,\text{Kilo}}{\text{Mole}},\ \text{HeatOfVaporization} \to \frac{301.\,\text{Joule}\,\text{Kilo}}{\text{Mole}},$$
$$\left.\text{Density} \to \frac{7900.4\,\text{Kilogram}}{\text{Meter}^3},\ \text{ThermalConductivity} \to \frac{10.6\,\text{Watt}}{\text{Kelvin}\,\text{Meter}}\right\}$$

Suppose we want to extract the boiling point. We know from the substitution rule above:

```
BoilingPoint → 3539. Kelvin
```

that a variable called **`BoilingPoint`** will be replaced with the list **`{3539., Kelvin}`**. We just need to apply the rule using **`/.`** the **`ReplaceAll`** operator as shown here.

```
(* harvest the boiling point using shorthand *)
BoilingPoint /. gd

3539. Kelvin
```

If you are not comfortable with the shorthand yet, you can use an explicit form:

```
(* harvest the boiling point using command notation *)
ReplaceAll[BoilingPoint, gd]

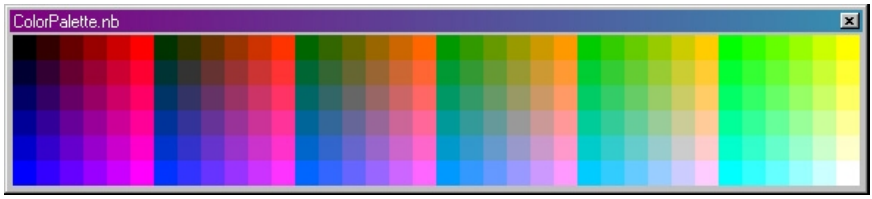
3539. Kelvin
```

You should use whichever convention you find easiest to work with.

We close with the observation that one of the standard add-on packages under **`Miscellaneous`**, **`ChemicalElements`**, also contains the same information.

- Demos > Palettes > Periodic Table
  Add-on & Links > Standard Packages > Miscellaneous > Chemical Elements

RGB color selector
Another very useful palette helps us select RGB colors visually. The palette, part of which is shown here, allows us to click on a color and get the RGB directives. The palette is quite large and we show only the left-most portion here.



Suppose we want to use a particular purple. We might assign this color the variable name **cpur**. As before, we enter an assignment shell.

```
(* prepare a statement for color command injection *)
cpur =;
```

As before, we plant the cursor between the equals sign and the colon. Then we move the cursor over to the palette and click on the square that is third from the left and third from the bottom. We find the color assignment done for us.

```
(* pick one of the purples *)
cpur = RGBColor[0.4, 0, 0.6];
```

This palette is quite useful since many of us who use *Mathematica* are mathematically oriented but not overly familiar with the RGB color scheme.

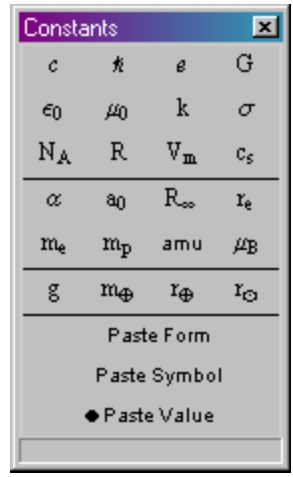
Also Wolfram Research has imbued *Mathematica* with significant color capabilities and interested readers can check out the following items in the Help Browser.

- Demos > Palettes > Color Palette
  Add-ons & Links > Standard Packages > Graphics > Colors
  Built-in Functions > Graphics and Sound > Graphics Primitives > GrayLevel
  Built-in Functions > Graphics and Sound > Graphics Primitives > RGBColor
  Built-in Functions > Graphics and Sound > Graphics Primitives > Hue

Built-in Functions > Graphics and Sound > Graphics Primitives > CMYKColor

Fundamental Constants

The fundamental constants warrant their own palette and Miscellaneous package. The palette takes the form



Foreshadowing events to come, we are going to check and see which packages are loaded right now.

```
$Packages
```

```
{Global`, System`}
```

We have a clean slate. Let's explore the three different paste options using the velocity of light. **`Paste Form`** inserts the symbol.

```
(* Paste Form *)
c
```

$$\frac{299792458\ \text{Meter}}{\text{Second}}$$

We see that by using the palette we have loaded a few packages.

```
$Packages

{Miscellaneous`Units`, Miscellaneous`SIUnits`,
 Miscellaneous`PhysicalConstants`, Global`, System`}
```

Paste Symbol shows use of the variable name from the add-on package:

```
(* Paste Symbol *)
SpeedOfLight

299792458 Meter
     Second
```

Paste Value gives us the magnitude and units.

```
(* Paste Value *)
299792458 Meter/Second

299792458 Meter
     Second
```

The speed of light is defined in terms of even more fundamental units, the vacuum *permittivity* and the vacuum *permeability.*

$$c = (\varepsilon_0 \mu_0)^{-\frac{1}{2}} \tag{1.1}$$

The *Mathematica* environment makes such comparisons facile. The least desirable way is to inspect the two values. As we shall see, this inspection will fail to find a small discrepancy. Pressing forth, we use **Paste Form** to input:

```
(* check the values against the fundamental definition *)
c = (ϵ0 μ0)^-1/2
```

$$\frac{2.99792 \times 10^8}{\sqrt{\frac{\text{Second}^2}{\text{Meter}^2}}}$$

They appear to be equal, but are they exactly equal as required? The best way to answer this question is to take the difference of the two quantities and look for a zero. However, this example is a special case showing a subtle problem. So before we difference the two forms, we'll reluctantly show a way to visually compare the two numbers.

```
(* let's check all the digits *)
FortranForm[%]

2.997924580105029e8/Sqrt(Second**2/Meter**2)
```

```
(* let's check all the digits *)
FortranForm[c]

(299792458*Meter)/Second
```

So we uncover a problem beyond the 9th decimal. Let's quantify the difference.

```
(* quantify the difference *)
δ = c - c
```

$$\frac{299792458\,\text{Meter}}{\text{Second}} - \frac{2.99792 \times 10^8}{\sqrt{\frac{\text{Second}^2}{\text{Meter}^2}}}$$

When we look at the expression, we see that the unit on both sides of the minus sign are equivalent. The natural response is to hope that **Simplify** will resolve the units problem.

```
(* try to get the units to cancel *)
Simplify[%]
```

$$\frac{\text{Meter}\left(2.99792\times10^{8}\,\text{Second}-2.99792\times10^{8}\,\text{Meter}\sqrt{\frac{\text{Second}^2}{\text{Meter}^2}}\right)}{\text{Second}^2}$$

That didn't work and we reflexively try **FullSimplify**.

```
(* maybe FullSimplify will work *)
FullSimplify[%%]
```

$$\frac{\text{Meter}\left(2.99792\times10^{8}\,\text{Second}-2.99792\times10^{8}\,\text{Meter}\sqrt{\frac{\text{Second}^2}{\text{Meter}^2}}\right)}{\text{Second}^2}$$

When all else fails, try the Help Browser. When we go to the browser entry for

Add-ons & Links > Miscellaneous > Units

(there is no help for SIUnits) we find the **Convert** command and try it immediately.

```
(* look in the Help Browser for the Units package *)
(* and discover the Convert command *)
Convert[δ, Meter / Second]
```

$$-\frac{0.0105029\,\text{Meter}}{\text{Second}}$$

The difference is now quantified and we will now check the values for the constants in the composite formula.

```
(* check the value for μ0 *)
VacuumPermeability
```

$$\frac{\pi \text{ Second Volt}}{2500000 \text{ Ampere Meter}}$$

```
(* check the value for ε0 *)
VacuumPermittivity
```

$$\frac{8.85419 \times 10^{-12} \text{ Ampere Second}}{\text{Meter Volt}}$$

Turning to [13], we see that the velocity of light *in vacua* is defined to be an exact quantity at 299,792,458 m/s. So the source of the problem is not with the speed of light. Also, the vacuum permeability (here magnetic constant) is defined exactly as:

$$\mu_0 = 4\pi \times 10^{-7} \text{NA}^{-2} \quad .$$

We notice the speed of light and vacuum permeability as defined in *Mathematica* are exact. The speed of light is an integer and the vacuum permeability is defined symbolically using π. This focuses our attention upon the vacuum permittivity (electric constant) which is defined now in terms of the previous two fundamental constants.

$$\varepsilon_0 = (\mu_0 c^2)^{-1} \tag{1.6}$$

The value of ε_0 then should be by definition.

```
(* adjusted vacuum permittivity *)
def = 1/(VacuumPermeability SpeedOfLight^2)
```

$$\frac{625000 \text{ Ampere Second}}{22468879468420441 \text{ Meter } \pi \text{ Volt}}$$

which works out to:

```
FortranForm[def]

(625000*Ampere*Second)/(2.246887946842044e16*Meter*Pi*Volt)
```

It would be nice to force a numerical valuation to remove the symbol π and to collect the units.

```
(* the correct definition *)
FortranForm[def // N]

(8.85418781762039e-12*Ampere*Second)/(Meter*Volt)
```

We are able to contrast this with the value stored in the **PhysicalConstants** package.

```
(* Mathematica internal value *)
FortranForm[ VacuumPermittivity ]

(8.854187817e-12*Ampere*Second)/(Meter*Volt)
```

Add-ons & Links > Standard Packages > Miscellaneous > PhysicalConstants

http://library.wolfram.com/infocenter/MathSource/5511/ (a much larger set of units)

1.9 Other resources

1.9.1 The *Mathematica* Journal

This quarterly journal is a useful collection of articles and quick answers that address common *Mathematica* questions and present cutting-edge applications.

Articles
There are some very interesting articles that teach us a little science and a lot of *Mathematica*. They make for very enjoyable reading. But there are articles which are quite useful. For instance, there was a 51 page article entitled “What’s New in *Mathematica* 5?”. This article, written by the *Mathematica* 5 Product Team at Wolfram Research presents a detailed discussion and examples of the new features and capabilities of *Mathematica* 5. The same issue also had a 40 page article by Wolfram Research staff member Lars Homuth called “*Mathematica* 4.2: A Technical Review.”

Demystifying Rules 8:4, Nancy Blackman

In and Out
The column edited by Paul Abbot has several short answers to reader questions.

Q: Normally, when creating a recursive function, I use the name of the function to perform the recursive call.

Q: I am computing the singular values of a matrix consisting of physical data. If I have some estimate of the error in each matrix element, is there a simple way to compute the error in the singular values?

Q: How can I compute the limit of as through integer values?

Q: Is there a way to produce 2D text for inclusion within 3D graphics?

Q: How can I play a sound to alert me regarding my program's progress during a long looping procedure?

Given that it is a matrix that varies with time, define its Moore-Penrose pseudoinverse. How can I find a simplified expression for in terms of?

Is it possible to have *Mathematica* interpret space *not* as multiplication but instead as noncommutative multiplication?

Q: Given a numeric data set, how does one go about finding a reasonable fitting function when you have no underlying theoretical equation to work with?

Q: Is there any way to separate input and output? I want to input code in one window and show the results in another.

Tricks of the trade
Construction of Matrix Differential Operators
Solving integral equations
Trott's corner

1.9.2 Other *Mathematica* books

The are dozens of quality *Mathematica* books in print and more are added every month.

1.9.3 Boneyard

There are some very important features to note in this module we will address individually. The first thing to note is that the comments are all outside of the module. Comments inside the module can cause erratic behavior. This is one of the few quirks in *Mathematica*, but it can be a vexing one. With internal comments you may have a situation where a module suddenly ceases to perform. You use it one day and on the next the module will not execute even though you have made no changes to the module whatsoever. The maladies of comments internal to module fall into two basic categories:

1. The module will not execute. The observed cure is to remove the internal comments. The problem is that this is not repeatable; the manifestation is sporadic.

2. The module declares that some valid line has improper syntax. A practical solution is to delete a carriage return and re-insert it. Unfortunately when you have thousands of line of code the message is often hard to associate with the specific line. Removal of internal comments also clears this up.

Developing long modules without comments is onerous for sure, but you may end up feeling that debugging a sporadic problem is a bigger burden. Now in the examples that follow we have included internal comments because we want the coding to be as clear as possible. But we often had problems with some of these modules and on the website we have the purged versions which we encourage you to harvest as needed.

The next thing we notice is the form of the arguments: the accompanying underscores *e.g.* `a_`. The underscore structure is used to specify a `Head` or variable type. We could restrict **a** to be an integer with the assignment `a_Integer`. Similarly, we could write `a_Real` or `a_Complex`. If we don't specify a type, that is if we simply use **a_**, then there is no restriction up the variable **a**. We'll do some experiments with headers in a moment.

But first we will discuss the `SetDelayed` command embodied in the symbol `:=`. The Set command that we are all familiar with, `=`, does an immediate assignment to the variable `qSolve`. With the delayed statement the assignment is not evaluated until the `qSolve` is actually used. Later on after we have learned a bit more you will see example where `SetDelayed` is absolutely essential. For now, let us simply state that it is the proper form to use.

Now for some simple experiments. Let's just look at calling different variable types. The only action we'll perform is to print the value of the variable passed. First we create the module

```
printIntegers[x_Integer] := Module[{},
   Print["x is an ", Head[x], " and has the value ", x];
  ];
```

and then we call it:

```
x = 2;
printIntegers[x]
```

```
x is a Integer  and has the value  2
```

Now we'll force *x* to be a real number by appending a period.

```
x = 3.;
Head[x]
printIntegers[x]

Real
```

```
printIntegers[3.]
```

Mathematica has just told us there was no action taken. It simply repeats the call. No action was taken. This is your clue that the module was not called. The three most common reasons that modules fail to execute are these:

1. The module has not yet been defined. Either the definition has been cleared or the definition has been overwritten.

2. The argument lists do not match. In the case immediately preceding we tried to pass a real as an argument and there is no definition of printIntegers that accepts a real number.

3. You have comments embedded in you module.

4. The name has multiple definitions and a non-functioning version takes precedence over your version.

This is an important topic and we shall dwell on it a bit longer. First we call your attention to something that comes as a surprise to many: Real numbers are considered distinct from Integer numbers in *Mathematica*. Here the distinction between real and integer should not be pondered in the mathematical sense, but instead in the sense of computer languages.

Let's return to the failure cases listed above and show how they could be debugged. In the first case let's call a non-existent module:

```
ghost[1, 2]

ghost[1, 2]
```

Our first step is to see what *Mathematica* thinks the module ghost is:

```
?ghost
```

```
Global`ghost
```

We see that there is no definition yet and nothing for the kernel to do. Another common malady is to make a mistake in typing the module. Some people like to turn off the spelling checker. This can cause the novice some heartache. Watch what happens where we accidently type Reals in lieu of `Real`:

```
ghost[x_Reals, y_Reals] := √(x^2 + y^2) ;
ghost[2., 3.]

ghost[2., 3.]
```

```
?ghost
```

```
Global`ghost

ghost [x_Reals , y_Reals ] := √(x^2 + y^2)
```

Why allow this? Why allow Reals? It is to allow us users to define our own types. This is a feature, not a bug.

A failure mechanism that can be quite vexing is when internal comments cause your modules to fail. The symptoms are perplexing. Modules that worked yesterday do not work today even though they have not been modified. An unstated assumption of debugging is a trust in the compiler or interpreter, so typically your investigations will delve elsewhere. But for this reason we strongly encourage users to avoid comments inside of modules. It is tempting to add them back later and you will probably be able to do so and you may think the malady has disappeared. Be fearful for the problem will probably return.

The final primary failure mechanism, precedence problems, will be explored later. We will continue with refining our example of the quadratic equation.

While it is easy to find the roots, in general we want these numbers so that we might use them in subsequent computations. Or perhaps we want the root closest to some value or we want the only real value. We will begin with finding the root closest to an input value.

```
(* find the root closest to x0 *)
constraint1[a_, b_, c_, x0_] := Module[{d, d1, d2},
  d = b^2 - 4 a c;
  x1 = (-b + Sqrt[d])/(2 a); x2 = (-b - Sqrt[d])/(2 a);
  d1 = x0 - x1; d2 = x0 - x2;
 ];
```

A list can help keep track of things much easier. A list is a collection of objects; in totality it can be a very abstract concept and we will revisit this topic often. Here we will let *r* be our list of roots. We could create the list *r* by executing:

```
(* find the root closest to x0 *)
constraint2[a_, b_, c_, x0_] := Module[{d, d1, d2},
  d = b^2 - 4 a c;
  r = (-b + Sqrt[d] {1, -1})/(2 a);
  d1 = x0 - x1; d2 = x0 - x2;
 ];
```

```
constraint2[1, 1, -1, 0];
r
```

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

1.10 In conclusion

This has been a brief introduction and survey for sure. We hope the reader now knows that

1. Wolfram Research has made some very helpful material to introduce *Mathematica* to new users.
2. The Wolfram web portal www.wolfram.com is a valuable resource for tutorials, examples, applications and news and should be checked periodically.

3. *Mathematica* is much more than a set of symbolic, numeric and graphic engines. There are many diverse packages to perform specialized tasks.

4. *Mathematica* has many other capabilities, such as the ability to make palettes, demos, and slide shows. Don't let your imagination limit how you use this application.

5. We are ready to begin learning through example.

C H A P T E R 2

Computation examples

In this chapter we will see how to use *Mathematica* to solve some computational problems. Before we dive into anything too complex, we will introduce the reader to different functions and solution methods using some straightforward examples. We begin by studying the solutions to a *quadratic equation*.

Demos > Palettes > Color Palette

2.1 The quadratic equation

Consider a second order polynomial known as a quadratic equation:

$$ax^2 + bx + c = 0 \tag{2.1}$$

Solutions to this equation are given by the familiar *quadratic formula:*

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{2.2}$$

Let's see how we can use *Mathematica* to find the roots of a quadratic equation using this formula.

Quadratic Equation

Given the coefficients of a quadratic equation, find solutions given by the quadratic formula

The first thing that we note is that there are two solutions indicated by the plus or minus symbol, ±. The most basic approach would be a sequence like:

```
(* define the coefficients a, b, and c in a + bx + c x^2 *)
a = 1; b = 2; c = 3;
(* define the two roots *)
x1 = (-b + Sqrt[b^2 - 4 a c])/(2 a)   (* positive root *)
x2 = (-b - Sqrt[b^2 - 4 a c])/(2 a)   (* negative root *)
```

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

$$\frac{1}{2}\left(-2-2\,i\,\sqrt{2}\right)$$

This certainly works, and we are happy to see that *Mathematica* is perfectly content to return complex numbers. But such unlabeled output is a bad habit to fall into. We should include some explanatory text:

```
x1 = (-b + Sqrt[b^2 - 4 a c])/(2 a); (* positive root *)
x2 = (-b - Sqrt[b^2 - 4 a c])/(2 a); (* negative root *)
Print["positive root = x1 = ", x1];
Print["negative root = x2 = ", x2];
```

positive root = x1 = $\frac{1}{2}\left(-2+2\,i\,\sqrt{2}\right)$

negative root = x2 = $\frac{1}{2}\left(-2-2\,i\,\sqrt{2}\right)$

The output now not only identifies the roots, but also tells us that these values are in the data variables `x1` and `x2` which can be used in subsequent computations. Naturally we would like to turn these instructions into an object like a subroutine which we

could use whenever we encounter a quadratic polynomial. We will use the **Module** structure which has the syntax:

```
(* syntax for a Module *)
Module[{local variable list},
  instruction;
  instruction;
 ];
```

The braces inside the module surround the local variable list. Over time, you will appreciate that your module variables can be local. As local variables, they will have no values outside of the module. Consider this example.

We have defined a module to solve for the two roots of a quadratic equation. In the module we call the roots `x1` and `x2`. External to the module, however, we define a global variable `x1`. Once we enter the module and *Mathematica* sees a request for a local variable `x1`, the global value will be ignored. So we will see that we enter the module with `x1` having a global value; then inside the loop `x1` will have a local value. When we exit the module, the global value of `x1` is restored.

```
(* define a global variable *)
x1 = 1;
(* diagnostic print *)
Print["the value of x1 outside of the module = ", x1];
(* Module that solve for the two roots of a quadratic equation *)
Module[{discriminant, x1, x2},
  discriminant = b^2 - 4 a c;
  (* information print *)
  Print["inside; discriminant = ", discriminant];
  x1 = (-b + Sqrt[discriminant])/(2 a);
  x2 = (-b1 Sqrt[discriminant])/(2 a);
  (* diagnostic print *)Print[
   "the value of x1 inside of the module = ", x1];
 ];
(* diagnostic prints *)
Print["the value of x1 outside of the module = ", x1];
Print["outside: discriminant = ", discriminant];
```

```
the value of x1 outside of the module = 1
inside ; discriminant = -8
the value of x1 inside of the module = 1/2 (-2 + 2 i Sqrt[2])
the value of x1 outside of the module = 1
outside : discriminant = discriminant
```

So we see that a module can maintain its own local variable list. The method used here is pedagogical. In practice we would want to maintain the values of the roots and assign the module a name so that we could call it when needed. Such a module to solve the quadratic equation might look like this.

```
(* module to solve the quadratic equation *)
(* input variables: the coefficients a, b, c *)
(* in the polynomial y(x) = a + b x + c x^2 *)
(* output variables: the two roots x1 and x2 *)
(* local variable: discriminant d (the rotational invariant) *)
qSolve[a_, b_, c_] := Module[{d},
   d = b^2 - 4 a c;          (* discriminant *)
   x1 = (-b + Sqrt[d])/(2 a);        (* positive root *)
   x2 = (-b - Sqrt[d])/(2 a);        (* negative root *)
  ];
```

Testing the code we find the following:

```
(* test the root finder *)
qSolve[1, 1, -1];
(* interogate *)
x1
x2
```

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

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

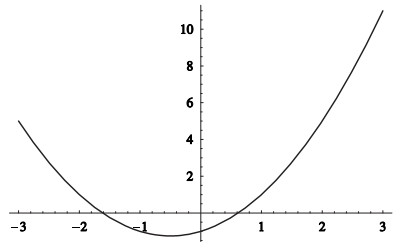
We have an answer, so the *Mathematica* coding is adequate. Do we have the right answer? Our first cross-check will be to see whether we have recovered the input coefficients.

```
(* verify we recover the coefficients *)
Simplify[(x - x1) (x - x2)]
```

$$-1+x+x^2$$

We can quickly create a plot to verify that we have in fact a quadratic polynomial.

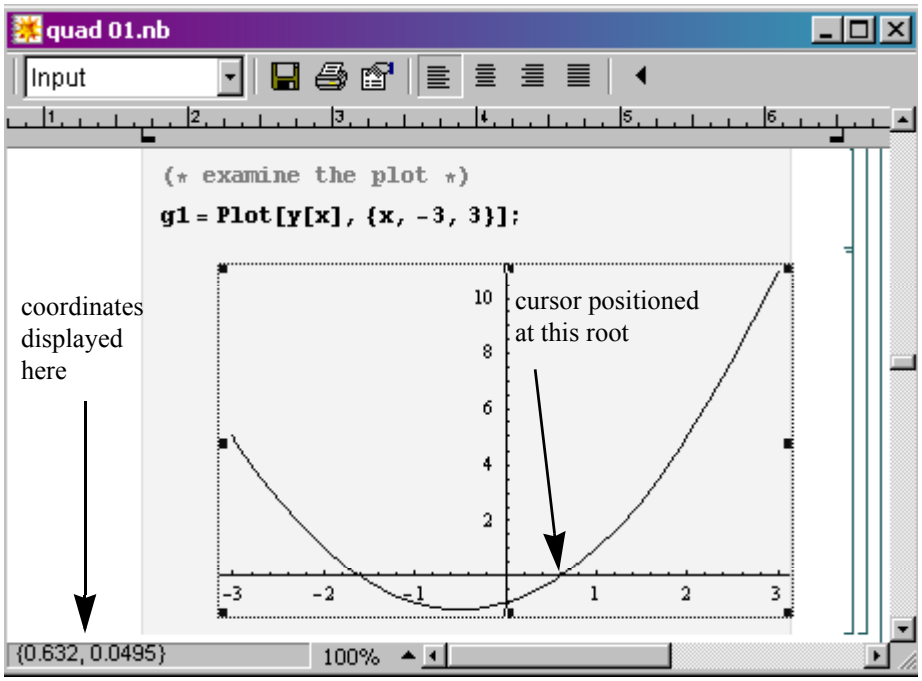
```
(* build the function *)
y[x_] := (x - x1) (x - x2)
(* examine the plot *)
g1 = Plot[y[x], {x, -3, 3}];
```



In our analysis, if possible exact comparisons are preferred. However, approximate checks are acceptable if they are not the final step in our analysis. In this vein, we present a *Mathematica* feature that allows us to make measurements directly on it plots.

Select the plot graphic by clicking on it. You'll see a bounding box with nine handles like the one in the picture below. Move your mouse inside of the box and hold down the CTRL key (⌘ or command on the Macintosh). Your cursor will turn into a set of crosshairs and the location of the center will be displayed in the lower left-hand corner of you notebook.

The graduations on the plot are abrupt. For example, in this case the x values jumped from 0.61 to 0.632 when the cursor jumped pixels. That was as close as we could come to the expected value 0.618. (You'll notice that the `Print Screen` command removes the cursor.)



So with the graph we see we are reasonably close. We can use `FindRoot` to show us whether or not we have found the true root. The syntax we will use is basic.

```
(* syntax used in this example *)
FindRoot[f[x] == 0, {x, xguess}];
```

Here `xguess` is some number near the root we are searching for. The plot above helps us to get these values. We find the negative root:

```
(* we can quickly use find root *)
(* negative root *)
FindRoot[y[x] == 0, {x, -2}]

{x → -1.61803}
```

and the positive root.

```
(* positive root *)
FindRoot[y[x] == 0, {x, 1}]

{x → 0.618034}
```

A somewhat crude test is to check the first six digits.

```
(* check the numeric value of the roots *)
{x1, x2} // N

{0.618034, -1.61803}
```

But we prefer a precise test, checking all digits.

```
(* check all digits in the answer *)
x1 - x /. root2
x2 - x /. root1

0.
```

```
0.
```

An important lesson to remember is that when you are looking for an equality, check for a zero; don't try to verify the digits yourself.

Symbolic solution

In the last example, while we looked up the solution for the quadratic equation, the full power of *Mathematica* can be exploited by using its symbolic computation engine.

Let's see how to solve the problem symbolically. The first step is to define a quadratic polynomial. Obviously we don't want to waste time naming coefficients. We want a labeling scheme that allows us to deal with the coefficients as if they were a list.

```
(* generic quadratic function *)
y[x_] := a[[1]] + a[[2]] x + a[[3]] x^2
```

New users would probably select a list form. A format like this allows us to immediately recognize that the *n*th term would be:

```
a[[n]] x^(n-1)
```

The problem is that the we cannot have an empty list. If we reference the third term in a list `a`, then there must be a list called `a` with at least three terms. Watch what happens when we violate this rule.

```
(* find the roots *)
Solve[y[x] == 0, x]
```

```
Part::partd : Part specification a[[1]] is longer than depth of object . More...

Part::partd : Part specification a[[2]] is longer than depth of object . More...

Part::partd : Part specification a[[3]] is longer than depth of object . More...

General::stop : Further output of
    Part::partd will be suppressed during this calculation . More...
```

$$\left\{\left\{x \to \frac{-a[[2]] - \sqrt{a[[2]]^2 - 4\,a[[1]]\,a[[3]]}}{2\,a[[3]]}\right\}, \left\{x \to \frac{-a[[2]] + \sqrt{a[[2]]^2 - 4\,a[[1]]\,a[[3]]}}{2\,a[[3]]}\right\}\right\}$$

We did get an answer, but *Mathematica* is unhappy. The error message

```
Part specification a[[1]] is longer than depth of object
```

tells us that we are looking for the first part in a list called `a`. There is no list called `a`. So even though we have an answer, the error is nudging us to use a different method.

This compels us to use a different syntax: `a[i]`. This stands for a dummy list of objects called `a`. The size need not be specified and we can use zero for an index. Now when we define the polynomial:

```
(* generic quadratic function *)
(* notice the use of a[i] *)
y[x_] := a[0] + a[1] x + a[2] x^2
```

we are able to solve for the roots without complaint.

```
(* find the roots *)
soln = Solve[y[x] == 0, x]
```

$$\left\{\left\{x \to \frac{-a[1] - \sqrt{a[1]^2 - 4\,a[0]\,a[2]}}{2\,a[2]}\right\}, \left\{x \to \frac{-a[1] + \sqrt{a[1]^2 - 4\,a[0]\,a[2]}}{2\,a[2]}\right\}\right\}$$

Again we see our two roots in a list. But this is a 2D list (i.e. a list indexed by two parameters). New users can be confused by this. Let's look at the components.

```
soln[[1]]
```

$$\left\{x \to \frac{-a[1] - \sqrt{a[1]^2 - 4\,a[0]\,a[2]}}{2\,a[2]}\right\}$$

You can clearly see the root, but it is inside braces. This tells us that we have another level. In fact, we can display the first element in `soln[[1]]` which is a list itself.

```
%[[1]]
```

$$x \to \frac{-a[1] - \sqrt{a[1]^2 - 4\,a[0]\,a[2]}}{2\,a[2]}$$

This is what we want: a naked substitution rule. To emphasize the structure of `soln`, we show the direct access method to this root.

```
soln[[1]][[1]]
```

$$x \to \frac{-a[1] - \sqrt{a[1]^2 - 4\,a[0]\,a[2]}}{2\,a[2]}$$

The universal tool to reduce the dimensions of a list is `Flatten`. The form we will use will flatten any structure down to a simple list. So a 2×2 matrix would become a list of four elements; a $3 \times 3 \times 3$ tensor would become a list of 27 elements. In our case, `rts` will become a list with two elements.

```
(* Flatten the subsitution rules *)
(* remove one set of braces; makes addressing easier *)
rts = Flatten[soln]
```

$$\left\{x \to \frac{-a[1] - \sqrt{a[1]^2 - 4\,a[0]\,a[2]}}{2\,a[2]},\ x \to \frac{-a[1] + \sqrt{a[1]^2 - 4\,a[0]\,a[2]}}{2\,a[2]}\right\}$$

With the dimension fixed, we want to convert the roots from rules to values. This leaves us with the familiar positive and negative roots.

```
(* convert substitution rules into roots *)
roots = {x /. rts[[1]], x /. rts[[2]]}
```

$$\left\{\frac{-a[1] - \sqrt{a[1]^2 - 4\,a[0]\,a[2]}}{2\,a[2]},\ \frac{-a[1] + \sqrt{a[1]^2 - 4\,a[0]\,a[2]}}{2\,a[2]}\right\}$$

The reason to find this form was to allow us to symbolically solve for the roots of a quadratic polynomial and then compute the roots. We will posit some vector of coefficients `b`.

```
(* let's inject these b values into the roots *)
(* corresponds to y[x] = 1 + x - x^2 *)
b = {1, 1, -1};
```

Digression: What is the zeroth element of a list? We cannot store data in this element because *Mathematica* stores the Head there.

```
(* the zeroth element is the Head *)
b[[0]]

List
```

Returning to the problem at hand, how do we put the `b` values in place of the `a`'s? We could show the method directly, but it's more instructive to show the readers how they might develop the solution.

Let's see how to swap just one element. We pick the second set and try to write a rule.

```
(* prototype command *)
roots /. a[1] → b[[2]]

{(-1 - Sqrt[1 - 4 a[0] a[2]])/(2 a[2]), (-1 + Sqrt[1 - 4 a[0] a[2]])/(2 a[2])}
```

Bolstered by our success, we will now have *Mathematica* create a list of rules for us.

```
(* assemble all the rules *)
rules = Table[a[i - 1] → b[[i]], {i, 3}]

{a[0] → 1, a[1] → 1, a[2] → -1}
```

We can apply these roots altogether:

```
(* apply all the rules *)
newroots = roots /. rules

{1/2 (1 + Sqrt[5]), 1/2 (1 - Sqrt[5])}
```

recovering the familiar result from the previous section.

We close this section with a look at polynomials through order five. Notice that *Mathematica* is content to solve even the most difficult of systems. This is why it is a good idea to use smaller systems when developing code. This generates a great deal of output and produces some powerful results.

```
(* find roots for the first few polynomials *)
Do[
 (* coefficient vector *)
 A = Table[a[i], {i, 0, n}];
 (* powers vector *)
 X = Table[x^i, {i, 0, n}];
 (* assemble the polynomial *)
 y[x_] := A.X;
 (* solve for the roots *)
 soln = Solve[y[x] == 0, x];
 (* output results *)
 Print[""];
 Print["roots for polynomial of order ", n];
 Print[Flatten[soln]];
 , {n, 5}];
```

```
roots for polynomial of order 1
```

$$\left\{x \to -\frac{a[0]}{a[1]}\right\}$$

```
roots for polynomial of order 2
```

$$\left\{x \to \frac{-a[1] - \sqrt{a[1]^2 - 4\,a[0]\,a[2]}}{2\,a[2]},\ x \to \frac{-a[1] + \sqrt{a[1]^2 - 4\,a[0]\,a[2]}}{2\,a[2]}\right\}$$

```
roots for polynomial of order 3
```

$$\left\{x \to -\frac{a[2]}{3\,a[3]} - \left(2^{1/3}\left(-a[2]^2 + 3\,a[1]\,a[3]\right)\right) \Big/ \left(3\,a[3]\left(-2\,a[2]^3 + 9\,a[1]\,a[2]\,a[3] - 27\,a[0]\,a[3]^2 + \sqrt{4\left(-a[2]^2 + 3\,a[1]\,a[3]\right)^3 + \left(-2\,a[2]^3 + 9\,a[1]\,a[2]\,a[3] - 27\,a[0]\,a[3]^2\right)^2}\right)^{1/3}\right) + \frac{1}{3\,2^{1/3}\,a[3]}\left(\left(-2\,a[2]^3 + 9\,a[1]\,a[2]\,a[3] - 27\,a[0]\,a[3]^2 + \sqrt{4\left(-a[2]^2 + 3\,a[1]\,a[3]\right)^3 + \left(-2\,a[2]^3 + 9\,a[1]\,a[2]\,a[3] - 27\,a[0]\,a[3]^2\right)^2}\right)^{1/3}\right),\right.$$

$$x \to -\frac{a[2]}{3\,a[3]} + \left(\left(1 + i\sqrt{3}\right)\left(-a[2]^2 + 3\,a[1]\,a[3]\right)\right) \Big/ \left(3\,2^{2/3}\,a[3]\left(-2\,a[2]^3 + 9\,a[1]\,a[2]\,a[3] - 27\,a[0]\,a[3]^2 + \sqrt{4\left(-a[2]^2 + 3\,a[1]\,a[3]\right)^3 + \left(-2\,a[2]^3 + 9\,a[1]\,a[2]\,a[3] - 27\,a[0]\,a[3]^2\right)^2}\right)^{1/3}\right) - \frac{1}{6\,2^{1/3}\,a[3]}\left(\left(1 - i\sqrt{3}\right)\left(-2\,a[2]^3 + 9\,a[1]\,a[2]\,a[3] - 27\,a[0]\,a[3]^2 + \sqrt{4\left(-a[2]^2 + 3\,a[1]\,a[3]\right)^3 + \left(-2\,a[2]^3 + 9\,a[1]\,a[2]\,a[3] - 27\,a[0]\,a[3]^2\right)^2}\right)^{1/3}\right),$$

$$x \to -\frac{a[2]}{3\,a[3]} + \left(\left(1 - i\sqrt{3}\right)\left(-a[2]^2 + 3\,a[1]\,a[3]\right)\right) \Big/ \left(3\,2^{2/3}\,a[3]\left(-2\,a[2]^3 + 9\,a[1]\,a[2]\,a[3] - 27\,a[0]\,a[3]^2 + \sqrt{4\left(-a[2]^2 + 3\,a[1]\,a[3]\right)^3 + \left(-2\,a[2]^3 + 9\,a[1]\,a[2]\,a[3] - 27\,a[0]\,a[3]^2\right)^2}\right)^{1/3}\right) - \frac{1}{6\,2^{1/3}\,a[3]}\left(\left(1 + i\sqrt{3}\right)\left(-2\,a[2]^3 + 9\,a[1]\,a[2]\,a[3] - 27\,a[0]\,a[3]^2 + \sqrt{4\left(-a[2]^2 + 3\,a[1]\,a[3]\right)^3 + \left(-2\,a[2]^3 + 9\,a[1]\,a[2]\,a[3] - 27\,a[0]\,a[3]^2\right)^2}\right)^{1/3}\right)\right\}$$

$$
\begin{aligned}
&\{x \to \\
&-\frac{a[3]}{4\,a[4]} - \frac{1}{2}\sqrt{\left\{\frac{a[3]^2}{4\,a[4]^2} - \frac{2\,a[2]}{3\,a[4]} + \left(2^{1/3}\left(a[2]^2 - 3\,a[1]\,a[3] + 12\,a[0]\,a[4]\right)\right)\Big/\left(3\,a[4]\right.\right.} \\
&\qquad \left(2\,a[2]^3 - 9\,a[1]\,a[2]\,a[3] + 27\,a[0]\,a[3]^2 + 27\,a[1]^2\,a[4] - 72\,a[0]\,a[2]\,a[4] + \right. \\
&\qquad\quad \sqrt{\left(-4\left(a[2]^2 - 3\,a[1]\,a[3] + 12\,a[0]\,a[4]\right)^3 + \left(2\,a[2]^3 - 9\,a[1]\,a[2]\,a[3] + \right.\right.} \\
&\qquad\qquad \left.\left.\left.27\,a[0]\,a[3]^2 + 27\,a[1]^2\,a[4] - 72\,a[0]\,a[2]\,a[4]\right)^2\right)\right)^{1/3}\Big) + \\
&\qquad \frac{1}{3\,2^{1/3}\,a[4]}\left(\left(2\,a[2]^3 - 9\,a[1]\,a[2]\,a[3] + 27\,a[0]\,a[3]^2 + 27\,a[1]^2\,a[4] - \right.\right. \\
&\qquad\quad 72\,a[0]\,a[2]\,a[4] + \\
&\qquad\quad \sqrt{\left(-4\left(a[2]^2 - 3\,a[1]\,a[3] + 12\,a[0]\,a[4]\right)^3 + \left(2\,a[2]^3 - 9\,a[1]\,a[2]\,a[3] + \right.} \\
&\qquad\qquad \left.\left.\left.\left.27\,a[0]\,a[3]^2 + 27\,a[1]^2\,a[4] - 72\,a[0]\,a[2]\,a[4]\right)^2\right)\right)^{1/3}\right)\Big\} - \\
&\frac{1}{2}\sqrt{\left\{\frac{a[3]^2}{2\,a[4]^2} - \frac{4\,a[2]}{3\,a[4]} - \left(2^{1/3}\left(a[2]^2 - 3\,a[1]\,a[3] + 12\,a[0]\,a[4]\right)\right)\Big/\right.} \\
&\qquad \left(3\,a[4]\left(2\,a[2]^3 - 9\,a[1]\,a[2]\,a[3] + 27\,a[0]\,a[3]^2 + 27\,a[1]^2\,a[4] - 72\,a[0]\,a[2]\right.\right. \\
&\qquad\quad a[4] + \sqrt{\left(-4\left(a[2]^2 - 3\,a[1]\,a[3] + 12\,a[0]\,a[4]\right)^3 + \left(2\,a[2]^3 - 9\,a[1]\,a[2]\right.} \\
&\qquad\qquad \left.\left.\left.a[3] + 27\,a[0]\,a[3]^2 + 27\,a[1]^2\,a[4] - 72\,a[0]\,a[2]\,a[4]\right)^2\right)\right)^{1/3}\Big) - \\
&\qquad \frac{1}{3\,2^{1/3}\,a[4]}\left(\left(2\,a[2]^3 - 9\,a[1]\,a[2]\,a[3] + 27\,a[0]\,a[3]^2 + 27\,a[1]^2\,a[4] - \right.\right. \\
&\qquad\quad 72\,a[0]\,a[2]\,a[4] + \sqrt{\left(-4\left(a[2]^2 - 3\,a[1]\,a[3] + 12\,a[0]\,a[4]\right)^3 + \right.} \\
&\qquad\qquad \left.\left(2\,a[2]^3 - 9\,a[1]\,a[2]\,a[3] + 27\,a[0]\,a[3]^2 + 27\,a[1]^2\,a[4] - \right.\right. \\
&\qquad\qquad\quad \left.\left.\left.72\,a[0]\,a[2]\,a[4]\right)^2\right)\right)^{1/3}\Big) - \left\{-\frac{a[3]^3}{a[4]^3} + \frac{4\,a[2]\,a[3]}{a[4]^2} - \frac{8\,a[1]}{a[4]}\right\}\Big/ \\
&\qquad \left\{4\sqrt{\left\{\frac{a[3]^2}{4\,a[4]^2} - \frac{2\,a[2]}{3\,a[4]} + \left(2^{1/3}\left(a[2]^2 - 3\,a[1]\,a[3] + 12\,a[0]\,a[4]\right)\right)\Big/\right.} \right. \\
&\qquad\quad \left(3\,a[4]\left(2\,a[2]^3 - 9\,a[1]\,a[2]\,a[3] + 27\,a[0]\,a[3]^2 + 27\,a[1]^2\,a[4] - \right.\right. \\
&\qquad\qquad 72\,a[0]\,a[2]\,a[4] + \sqrt{\left(-4\left(a[2]^2 - 3\,a[1]\,a[3] + 12\,a[0]\,a[4]\right)^3 + \right.} \\
&\qquad\qquad\quad \left(2\,a[2]^3 - 9\,a[1]\,a[2]\,a[3] + 27\,a[0]\,a[3]^2 + 27\,a[1]^2\,a[4] - \right. \\
&\qquad\qquad\qquad \left.\left.\left.\left.72\,a[0]\,a[2]\,a[4]\right)^2\right)\right)^{1/3}\Big) + \frac{1}{3\,2^{1/3}\,a[4]} \\
&\qquad\quad \left(\left(2\,a[2]^3 - 9\,a[1]\,a[2]\,a[3] + 27\,a[0]\,a[3]^2 + 27\,a[1]^2\,a[4] - \right.\right. \\
&\qquad\qquad 72\,a[0]\,a[2]\,a[4] + \sqrt{\left(-4\left(a[2]^2 - 3\,a[1]\,a[3] + 12\,a[0]\,a[4]\right)^3 + \right.} \\
&\qquad\qquad\quad \left(2\,a[2]^3 - 9\,a[1]\,a[2]\,a[3] + 27\,a[0]\,a[3]^2 + \right. \\
&\qquad\qquad\qquad \left.\left.\left.\left.\left.\left.27\,a[1]^2\,a[4] - 72\,a[0]\,a[2]\,a[4]\right)^2\right)\right)^{1/3}\Big)\right\}\Big\}\Big\},
\end{aligned}
$$

```
x →
 -a[3]/(4 a[4]) - 1/2 Sqrt[a[3]^2/(4 a[4]^2) - (2 a[2])/(3 a[4]) + (2^(1/3) (a[2]^2 - 3 a[1] a[3] + 12 a[0] a[4]))/
      (3 a[4] (2 a[2]^3 - 9 a[1] a[2] a[3] + 27 a[0] a[3]^2 + 27 a[1]^2 a[4] - 72 a[0] a[2] a[4] +
          Sqrt[(-4 (a[2]^2 - 3 a[1] a[3] + 12 a[0] a[4])^3 + (2 a[2]^3 - 9 a[1] a[2] a[3] +
              27 a[0] a[3]^2 + 27 a[1]^2 a[4] - 72 a[0] a[2] a[4])^2)]^(1/3)) +
    1/(3 2^(1/3) a[4]) ((2 a[2]^3 - 9 a[1] a[2] a[3] + 27 a[0] a[3]^2 + 27 a[1]^2 a[4] -
        72 a[0] a[2] a[4] +
        Sqrt[(-4 (a[2]^2 - 3 a[1] a[3] + 12 a[0] a[4])^3 + (2 a[2]^3 - 9 a[1] a[2] a[3] +
            27 a[0] a[3]^2 + 27 a[1]^2 a[4] - 72 a[0] a[2] a[4])^2)]^(1/3))) +
 1/2 Sqrt[a[3]^2/(2 a[4]^2) - (4 a[2])/(3 a[4]) - (2^(1/3) (a[2]^2 - 3 a[1] a[3] + 12 a[0] a[4]))/
      (3 a[4] (2 a[2]^3 - 9 a[1] a[2] a[3] + 27 a[0] a[3]^2 + 27 a[1]^2 a[4] - 72 a[0] a[2]
          a[4] + Sqrt[(-4 (a[2]^2 - 3 a[1] a[3] + 12 a[0] a[4])^3 + (2 a[2]^3 - 9 a[1] a[2]
              a[3] + 27 a[0] a[3]^2 + 27 a[1]^2 a[4] - 72 a[0] a[2] a[4])^2)]^(1/3)) -
    1/(3 2^(1/3) a[4]) ((2 a[2]^3 - 9 a[1] a[2] a[3] + 27 a[0] a[3]^2 + 27 a[1]^2 a[4] -
        72 a[0] a[2] a[4] + Sqrt[(-4 (a[2]^2 - 3 a[1] a[3] + 12 a[0] a[4])^3 +
            (2 a[2]^3 - 9 a[1] a[2] a[3] + 27 a[0] a[3]^2 + 27 a[1]^2 a[4] -
              72 a[0] a[2] a[4])^2)]^(1/3)) - (-(a[3]^3/a[4]^3) + (4 a[2] a[3])/a[4]^2 - (8 a[1])/a[4])/
     (4 Sqrt[a[3]^2/(4 a[4]^2) - (2 a[2])/(3 a[4]) + (2^(1/3) (a[2]^2 - 3 a[1] a[3] + 12 a[0] a[4]))/
          (3 a[4] (2 a[2]^3 - 9 a[1] a[2] a[3] + 27 a[0] a[3]^2 + 27 a[1]^2 a[4] -
              72 a[0] a[2] a[4] + Sqrt[(-4 (a[2]^2 - 3 a[1] a[3] + 12 a[0] a[4])^3 +
                  (2 a[2]^3 - 9 a[1] a[2] a[3] + 27 a[0] a[3]^2 +
                    27 a[1]^2 a[4] - 72 a[0] a[2] a[4])^2)]^(1/3)) +
        1/(3 2^(1/3) a[4]) ((2 a[2]^3 - 9 a[1] a[2] a[3] + 27 a[0] a[3]^2 + 27 a[1]^2
            a[4] - 72 a[0] a[2] a[4] + Sqrt[(-4 (a[2]^2 - 3 a[1] a[3] + 12 a[0] a[4])^3 +
              (2 a[2]^3 - 9 a[1] a[2] a[3] + 27 a[0] a[3]^2 +
                27 a[1]^2 a[4] - 72 a[0] a[2] a[4])^2)]^(1/3)))]),
```

$$
\begin{aligned}
x \to\ & -\frac{a[3]}{4\,a[4]} + \frac{1}{2}\sqrt{\left\{\frac{a[3]^2}{4\,a[4]^2} - \frac{2\,a[2]}{3\,a[4]} + \left(2^{1/3}\left(a[2]^2 - 3\,a[1]\,a[3] + 12\,a[0]\,a[4]\right)\right)\Big/\left(3\,a[4]\right.\right.} \\
& \left(2\,a[2]^3 - 9\,a[1]\,a[2]\,a[3] + 27\,a[0]\,a[3]^2 + 27\,a[1]^2\,a[4] - 72\,a[0]\,a[2]\,a[4] + \right. \\
& \sqrt{\left(-4\left(a[2]^2 - 3\,a[1]\,a[3] + 12\,a[0]\,a[4]\right)^3 + \left(2\,a[2]^3 - 9\,a[1]\,a[2]\,a[3] + \right.\right.} \\
& \left.\left.\left.\left.\left.27\,a[0]\,a[3]^2 + 27\,a[1]^2\,a[4] - 72\,a[0]\,a[2]\,a[4]\right)^2\right)\right)^{1/3}\right) + \\
& \frac{1}{3\,2^{1/3}\,a[4]}\left(\left(2\,a[2]^3 - 9\,a[1]\,a[2]\,a[3] + 27\,a[0]\,a[3]^2 + 27\,a[1]^2\,a[4] - \right.\right. \\
& 72\,a[0]\,a[2]\,a[4] + \\
& \sqrt{\left(-4\left(a[2]^2 - 3\,a[1]\,a[3] + 12\,a[0]\,a[4]\right)^3 + \left(2\,a[2]^3 - 9\,a[1]\,a[2]\,a[3] + \right.\right.} \\
& \left.\left.\left.\left.\left.\left.27\,a[0]\,a[3]^2 + 27\,a[1]^2\,a[4] - 72\,a[0]\,a[2]\,a[4]\right)^2\right)\right)^{1/3}\right)\right\} - \\
& \frac{1}{2}\sqrt{\left\{\frac{a[3]^2}{2\,a[4]^2} - \frac{4\,a[2]}{3\,a[4]} - \left(2^{1/3}\left(a[2]^2 - 3\,a[1]\,a[3] + 12\,a[0]\,a[4]\right)\right)\Big/\right.} \\
& \left(3\,a[4]\left(2\,a[2]^3 - 9\,a[1]\,a[2]\,a[3] + 27\,a[0]\,a[3]^2 + 27\,a[1]^2\,a[4] - 72\,a[0]\,a[2]\right.\right. \\
& a[4] + \sqrt{\left(-4\left(a[2]^2 - 3\,a[1]\,a[3] + 12\,a[0]\,a[4]\right)^3 + \left(2\,a[2]^3 - 9\,a[1]\,a[2]\right.} \\
& \left.\left.\left.\left.\left.a[3] + 27\,a[0]\,a[3]^2 + 27\,a[1]^2\,a[4] - 72\,a[0]\,a[2]\,a[4]\right)^2\right)\right)^{1/3}\right) - \\
& \frac{1}{3\,2^{1/3}\,a[4]}\left(\left(2\,a[2]^3 - 9\,a[1]\,a[2]\,a[3] + 27\,a[0]\,a[3]^2 + 27\,a[1]^2\,a[4] - \right.\right. \\
& 72\,a[0]\,a[2]\,a[4] + \sqrt{\left(-4\left(a[2]^2 - 3\,a[1]\,a[3] + 12\,a[0]\,a[4]\right)^3 + \right.} \\
& \left(2\,a[2]^3 - 9\,a[1]\,a[2]\,a[3] + 27\,a[0]\,a[3]^2 + 27\,a[1]^2\,a[4] - \right. \\
& \left.\left.\left.\left.72\,a[0]\,a[2]\,a[4]\right)^2\right)\right)^{1/3}\right) + \left\{-\frac{a[3]^3}{a[4]^3} + \frac{4\,a[2]\,a[3]}{a[4]^2} - \frac{8\,a[1]}{a[4]}\right\}\Big/ \\
& \left\{4\sqrt{\left\{\frac{a[3]^2}{4\,a[4]^2} - \frac{2\,a[2]}{3\,a[4]} + \left(2^{1/3}\left(a[2]^2 - 3\,a[1]\,a[3] + 12\,a[0]\,a[4]\right)\right)\Big/\right.}\right. \\
& \left(3\,a[4]\left(2\,a[2]^3 - 9\,a[1]\,a[2]\,a[3] + 27\,a[0]\,a[3]^2 + 27\,a[1]^2\,a[4] - \right.\right. \\
& 72\,a[0]\,a[2]\,a[4] + \sqrt{\left(-4\left(a[2]^2 - 3\,a[1]\,a[3] + 12\,a[0]\,a[4]\right)^3 + \right.} \\
& \left(2\,a[2]^3 - 9\,a[1]\,a[2]\,a[3] + 27\,a[0]\,a[3]^2 + \right. \\
& \left.\left.\left.\left.27\,a[1]^2\,a[4] - 72\,a[0]\,a[2]\,a[4]\right)^2\right)\right)^{1/3}\right) + \\
& \frac{1}{3\,2^{1/3}\,a[4]}\left(\left(2\,a[2]^3 - 9\,a[1]\,a[2]\,a[3] + 27\,a[0]\,a[3]^2 + 27\,a[1]^2\right.\right. \\
& a[4] - 72\,a[0]\,a[2]\,a[4] + \sqrt{\left(-4\left(a[2]^2 - 3\,a[1]\,a[3] + 12\,a[0]\,a[4]\right)^3 + \right.} \\
& \left(2\,a[2]^3 - 9\,a[1]\,a[2]\,a[3] + 27\,a[0]\,a[3]^2 + \right. \\
& \left.\left.\left.\left.\left.\left.\left.27\,a[1]^2\,a[4] - 72\,a[0]\,a[2]\,a[4]\right)^2\right)\right)^{1/3}\right)\right\}\right\}\Bigg\},
\end{aligned}
$$

$$
\begin{aligned}
x \to\ & -\frac{a[3]}{4\,a[4]} + \frac{1}{2}\sqrt{\left\{\frac{a[3]^2}{4\,a[4]^2} - \frac{2\,a[2]}{3\,a[4]} + \left(2^{1/3}\left(a[2]^2 - 3\,a[1]\,a[3] + 12\,a[0]\,a[4]\right)\right)\Big/\left(3\,a[4]\right.\right.} \\
& \left(2\,a[2]^3 - 9\,a[1]\,a[2]\,a[3] + 27\,a[0]\,a[3]^2 + 27\,a[1]^2\,a[4] - 72\,a[0]\,a[2]\,a[4] + \right. \\
& \sqrt{\left(-4\left(a[2]^2 - 3\,a[1]\,a[3] + 12\,a[0]\,a[4]\right)^3 + \left(2\,a[2]^3 - 9\,a[1]\,a[2]\,a[3] + \right.\right.} \\
& \left.\left.\left.\left.27\,a[0]\,a[3]^2 + 27\,a[1]^2\,a[4] - 72\,a[0]\,a[2]\,a[4]\right)^2\right)\right)^{1/3}\Big) + \\
& \frac{1}{3\,2^{1/3}\,a[4]}\left(\left(2\,a[2]^3 - 9\,a[1]\,a[2]\,a[3] + 27\,a[0]\,a[3]^2 + 27\,a[1]^2\,a[4] - \right.\right. \\
& 72\,a[0]\,a[2]\,a[4] + \\
& \sqrt{\left(-4\left(a[2]^2 - 3\,a[1]\,a[3] + 12\,a[0]\,a[4]\right)^3 + \left(2\,a[2]^3 - 9\,a[1]\,a[2]\,a[3] + \right.\right.} \\
& \left.\left.\left.\left.\left.27\,a[0]\,a[3]^2 + 27\,a[1]^2\,a[4] - 72\,a[0]\,a[2]\,a[4]\right)^2\right)\right)^{1/3}\Big)\Big\} + \\
& \frac{1}{2}\sqrt{\left\{\frac{a[3]^2}{2\,a[4]^2} - \frac{4\,a[2]}{3\,a[4]} - \left(2^{1/3}\left(a[2]^2 - 3\,a[1]\,a[3] + 12\,a[0]\,a[4]\right)\right)\Big/\right.} \\
& \left(3\,a[4]\,\left(2\,a[2]^3 - 9\,a[1]\,a[2]\,a[3] + 27\,a[0]\,a[3]^2 + 27\,a[1]^2\,a[4] - 72\,a[0]\,a[2]\right.\right. \\
& a[4] + \sqrt{\left(-4\left(a[2]^2 - 3\,a[1]\,a[3] + 12\,a[0]\,a[4]\right)^3 + \left(2\,a[2]^3 - 9\,a[1]\,a[2]\right.} \\
& \left.\left.\left.\left.a[3] + 27\,a[0]\,a[3]^2 + 27\,a[1]^2\,a[4] - 72\,a[0]\,a[2]\,a[4]\right)^2\right)\right)^{1/3}\Big) - \\
& \frac{1}{3\,2^{1/3}\,a[4]}\left(\left(2\,a[2]^3 - 9\,a[1]\,a[2]\,a[3] + 27\,a[0]\,a[3]^2 + 27\,a[1]^2\,a[4] - \right.\right. \\
& 72\,a[0]\,a[2]\,a[4] + \sqrt{\left(-4\left(a[2]^2 - 3\,a[1]\,a[3] + 12\,a[0]\,a[4]\right)^3 + \right.} \\
& \left(2\,a[2]^3 - 9\,a[1]\,a[2]\,a[3] + 27\,a[0]\,a[3]^2 + 27\,a[1]^2\,a[4] - \right. \\
& \left.\left.\left.\left.72\,a[0]\,a[2]\,a[4]\right)^2\right)\right)^{1/3}\Big) + \left\{-\frac{a[3]^3}{a[4]^3} + \frac{4\,a[2]\,a[3]}{a[4]^2} - \frac{8\,a[1]}{a[4]}\right\}\Big/ \\
& \left\{4\sqrt{\left\{\frac{a[3]^2}{4\,a[4]^2} - \frac{2\,a[2]}{3\,a[4]} + \left(2^{1/3}\left(a[2]^2 - 3\,a[1]\,a[3] + 12\,a[0]\,a[4]\right)\right)\Big/\right.}\right. \\
& \left(3\,a[4]\,\left(2\,a[2]^3 - 9\,a[1]\,a[2]\,a[3] + 27\,a[0]\,a[3]^2 + 27\,a[1]^2\,a[4] - \right.\right. \\
& 72\,a[0]\,a[2]\,a[4] + \sqrt{\left(-4\left(a[2]^2 - 3\,a[1]\,a[3] + 12\,a[0]\,a[4]\right)^3 + \right.} \\
& \left(2\,a[2]^3 - 9\,a[1]\,a[2]\,a[3] + 27\,a[0]\,a[3]^2 + 27\,a[1]^2\,a[4] - \right. \\
& \left.\left.\left.\left.72\,a[0]\,a[2]\,a[4]\right)^2\right)\right)^{1/3}\Big) + \frac{1}{3\,2^{1/3}\,a[4]} \\
& \left(\left(2\,a[2]^3 - 9\,a[1]\,a[2]\,a[3] + 27\,a[0]\,a[3]^2 + 27\,a[1]^2\,a[4] - \right.\right. \\
& 72\,a[0]\,a[2]\,a[4] + \sqrt{\left(-4\left(a[2]^2 - 3\,a[1]\,a[3] + 12\,a[0]\,a[4]\right)^3 + \right.} \\
& \left(2\,a[2]^3 - 9\,a[1]\,a[2]\,a[3] + 27\,a[0]\,a[3]^2 + \right. \\
& \left.\left.\left.\left.\left.\left.\left.27\,a[1]^2\,a[4] - 72\,a[0]\,a[2]\,a[4]\right)^2\right)\right)^{1/3}\Big)\Big\}\Big\}\Big\}\Big\}\Big\}\Big\}
\end{aligned}
$$

```
roots for polynomial of order 5
```

$$
\begin{aligned}
& \{x \to \text{Root}\left[a[0] + a[1]\,\#1 + a[2]\,\#1^2 + a[3]\,\#1^3 + a[4]\,\#1^4 + a[5]\,\#1^5\ \&,\ 1\right], \\
& \ x \to \text{Root}\left[a[0] + a[1]\,\#1 + a[2]\,\#1^2 + a[3]\,\#1^3 + a[4]\,\#1^4 + a[5]\,\#1^5\ \&,\ 2\right], \\
& \ x \to \text{Root}\left[a[0] + a[1]\,\#1 + a[2]\,\#1^2 + a[3]\,\#1^3 + a[4]\,\#1^4 + a[5]\,\#1^5\ \&,\ 3\right], \\
& \ x \to \text{Root}\left[a[0] + a[1]\,\#1 + a[2]\,\#1^2 + a[3]\,\#1^3 + a[4]\,\#1^4 + a[5]\,\#1^5\ \&,\ 4\right], \\
& \ x \to \text{Root}\left[a[0] + a[1]\,\#1 + a[2]\,\#1^2 + a[3]\,\#1^3 + a[4]\,\#1^4 + a[5]\,\#1^5\ \&,\ 5\right]\}
\end{aligned}
$$

The reader should consult the Help Browser to learn about the syntax of the `Root` command.

Built-in Functions > Polynomial Functions > Root

Cubic Equation> Quartic Equation> Quintic Equation

Advanced topic: arbitrary orders

In this section we are going to show an advanced use for lists. Typically we prefer to show a do-loop first. However since this is an advanced topic, we invite the novice reader to skip this section and return to it later.

Let's think of this problem in terms of vectors. The coefficient vector a can be written as:

$$a = \{a_0,\ a_1,\ \ldots,\ a_n\} \tag{2.3}$$

and the powers of x can be written in a vector s which describes the space we are in

$$s = \{1,\ x,\ \ldots,\ x^n\}. \tag{2.4}$$

Contrast this with the basis vectors that are more familiar

$$\hat{x},\ \hat{y},\ \hat{z}) \tag{2.5}$$

In the new basis the generic nth degree polynomial $P^n(x)$ becomes a dot product.

$$P^n(x) = a \cdot s \tag{2.6}$$

For example, we can create our quadratic equation this way.

```
(* a simpler way *)
(* highest power in x *)
n = 2;
(* coefficient vector *)
A = Table[a[i], {i, n + 1}];
(* powers vector *)
X = Table[x^i, {i, 0, n}];
(* assemble the polynomial *)
y[x_] := A.X;
(* solve for the roots *)
soln = Solve[y[x] == 0, x];
```

Examine the pieces.

```
(* let's look at the pieces *)
A
X
A.X
soln
```

```
{a[0], a[1], a[2]}
```

```
{1, x, x^2}
```

```
a[0] + x a[1] + x^2 a[2]
```

$$\left\{\left\{x \to \frac{-a[1] - \sqrt{a[1]^2 - 4\,a[0]\,a[2]}}{2\,a[2]}\right\}, \left\{x \to \frac{-a[1] + \sqrt{a[1]^2 - 4\,a[0]\,a[2]}}{2\,a[2]}\right\}\right\}$$

We see the two vectors from equations 2.3 and 2.4. We recognize the polynomial from the dot product and we see the now-customary roots.

This vector formulation is extremely handy in two environments. The first is when you have arbitrary sizes of polynomials. As we'll show next, we can easily use this recipe to generate higher orders immediately. The other arena where this is a blessing is dealing with special functions. Instead of representing them as polynomials we can write them as vectors. This makes manipulation and computation quite facile.

2.2 Singular matrices and inversion

In many linear problems, *singular* matrices can be encountered. We will consider an example using a least squares fit which generates the linear equation:

$$Ax = B \tag{2.7}$$

where A in an $n \times n$ matrix describing the measurement system, x is an $n \times 1$ vector of the fit parameters and B is an $n \times 1$ vector containing the measurements. In general, such a linear system would admit the solution:

$$x = A^{-1}B \tag{2.8}$$

However, in many physically relevant problems such as this the matrix A is singular and we cannot find the inverse A^{-1} such that:

$$A^{-1}A = I \tag{2.9}$$

where I is the identity matrix. But this will not stop us from solving for parameter vector x.

We show below a singular matrix from a least squares fit problem. The matrix describes the physical layout of the system with only six elements. This 2D system has been reduced to a linear order. The positive numbers tell us how many immediate neighbors each site has; the negative numbers mark the position of the neighbor in the linear order. Matrices of this type are always singular.

A matrix A is singular if there is some vector a which rotates the A into a null space.(**NR) That is if :

$$A \cdot a = 0 \tag{2.10}$$

Clearly any constant vector satisfies equation 2.10 because every row and every column sums to zero by definition of the problem. Hence, our interaction matrix A is singular.

```
A // MatrixForm
```

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

But suppose we were remiss in this observation. What happens when we try to invert A? We get an explicit error message and the matrix values are returned.

```
Inverse[A]

⚠ Inverse ::sing : Matrix {{1, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0},
     <<10>>, {0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 1}} is singular . More...

Inverse[{{1, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0},
  {0, 2, 0, 0, -1, 0, -1, 0, 0, 0, 0, 0},
  {0, 0, 2, 0, 0, -1, 0, -1, 0, 0, 0, 0},
  {0, 0, 0, 1, 0, 0, -1, 0, 0, 0, 0, 0},
  {0, -1, 0, 0, 2, 0, 0, 0, 0, -1, 0, 0},
  {-1, 0, -1, 0, 0, 4, 0, 0, -1, 0, -1, 0},
  {0, -1, 0, -1, 0, 0, 4, 0, 0, -1, 0, -1},
  {0, 0, -1, 0, 0, 0, 0, 2, 0, 0, -1, 0},
  {0, 0, 0, 0, 0, -1, 0, 0, 1, 0, 0, 0},
  {0, 0, 0, 0, -1, 0, -1, 0, 0, 2, 0, 0},
  {0, 0, 0, 0, 0, -1, 0, -1, 0, 0, 2, 0},
  {0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 1}}]
```

This error message is succinct and to the point. The matrix is singular and we turn to singular value decomposition or the Moore-Penrose generalized matrix inverse sometimes called the pseudoinverse. You may be unsure which one to use. But this is one of the many joys of *Mathematica*: we can quickly compare both methods.

The pseudoinverse of a matrix m is denoted by $m^{(-1)}$ in the *Mathematica* help and as m^{+} in *The CRC Concise Encyclopedia of Mathematics*. This is one of the few instances where the notation is not shared. The concept of the pseudoinverse covers both singular matrices and matrices that are not square and has the properties that:

$$mm^{+}m = m \tag{2.11}$$

$$m^{+}mm^{+} = m^{+} \tag{2.12}$$

$$(mm^{+})^{T} = mm^{+} \tag{2.13}$$

$$(m^{+}m)^{T} = m^{+}m \tag{2.14}$$

If the inverse of exists, then:

$$m^{+} = (m^{T}m)^{-1}m^{T} \tag{2.15}$$

Moore-Penrose Generalized Matrix Inverse

The Help Browser notes that the pseudoinverse in the matrix m^{+} which minimizes:

$$\sum_{i,\,j} (mm^{+} - I)^{2}. \tag{2.16}$$

PseudoInverse

The command to compute the pseudoinverse in *Mathematica* is `PseudoInverse` as shown here. We also want to expose the reader to the appearance of the pseudoinverse matrices so we show the result, too.

```
(* compute and display the pseudoinverse *)
piA = PseudoInverse[A];
piA // MatrixForm
```

$$\begin{pmatrix}
\frac{31}{36} & 0 & -\frac{2}{9} & 0 & 0 & \frac{1}{36} & 0 & -\frac{11}{36} & -\frac{5}{36} & 0 & -\frac{2}{9} & 0 \\
0 & \frac{4}{9} & 0 & -\frac{2}{9} & \frac{1}{9} & 0 & -\frac{1}{18} & 0 & 0 & -\frac{1}{18} & 0 & -\frac{2}{9} \\
-\frac{2}{9} & 0 & \frac{4}{9} & 0 & 0 & -\frac{1}{18} & 0 & \frac{1}{9} & -\frac{2}{9} & 0 & -\frac{1}{18} & 0 \\
0 & -\frac{2}{9} & 0 & \frac{31}{36} & -\frac{11}{36} & 0 & \frac{1}{36} & 0 & 0 & -\frac{2}{9} & 0 & -\frac{5}{36} \\
0 & \frac{1}{9} & 0 & -\frac{11}{36} & \frac{19}{36} & 0 & -\frac{5}{36} & 0 & 0 & \frac{1}{9} & 0 & -\frac{11}{36} \\
\frac{1}{36} & 0 & -\frac{1}{18} & 0 & 0 & \frac{7}{36} & 0 & -\frac{5}{36} & \frac{1}{36} & 0 & -\frac{1}{18} & 0 \\
0 & -\frac{1}{18} & 0 & \frac{1}{36} & -\frac{5}{36} & 0 & \frac{7}{36} & 0 & 0 & -\frac{1}{18} & 0 & \frac{1}{36} \\
-\frac{11}{36} & 0 & \frac{1}{9} & 0 & 0 & -\frac{5}{36} & 0 & \frac{19}{36} & -\frac{11}{36} & 0 & \frac{1}{9} & 0 \\
-\frac{5}{36} & 0 & -\frac{2}{9} & 0 & 0 & \frac{1}{36} & 0 & -\frac{11}{36} & \frac{31}{36} & 0 & -\frac{2}{9} & 0 \\
0 & -\frac{1}{18} & 0 & -\frac{2}{9} & \frac{1}{9} & 0 & -\frac{1}{18} & 0 & 0 & \frac{4}{9} & 0 & -\frac{2}{9} \\
-\frac{2}{9} & 0 & -\frac{1}{18} & 0 & 0 & -\frac{1}{18} & 0 & \frac{1}{9} & -\frac{2}{9} & 0 & \frac{4}{9} & 0 \\
0 & -\frac{2}{9} & 0 & -\frac{5}{36} & -\frac{11}{36} & 0 & \frac{1}{36} & 0 & 0 & -\frac{2}{9} & 0 & \frac{31}{36}
\end{pmatrix}$$

The inquisitive reader may be curious as to how closely the product of the matrix and its pseudoinverse resembles the identity matrix. We show this product and then we see a tidier form gained by multiplying by an integer.

```
(* how closely does the
  product A.piA match the identity matrix? *)
A.piA // MatrixForm
```

$$\begin{pmatrix} \frac{5}{6} & 0 & -\frac{1}{6} & 0 & 0 & -\frac{1}{6} & 0 & -\frac{1}{6} & -\frac{1}{6} & 0 & -\frac{1}{6} & 0 \\ 0 & \frac{5}{6} & 0 & -\frac{1}{6} & -\frac{1}{6} & 0 & -\frac{1}{6} & 0 & 0 & -\frac{1}{6} & 0 & -\frac{1}{6} \\ -\frac{1}{6} & 0 & \frac{5}{6} & 0 & 0 & -\frac{1}{6} & 0 & -\frac{1}{6} & -\frac{1}{6} & 0 & -\frac{1}{6} & 0 \\ 0 & -\frac{1}{6} & 0 & \frac{5}{6} & -\frac{1}{6} & 0 & -\frac{1}{6} & 0 & 0 & -\frac{1}{6} & 0 & -\frac{1}{6} \\ 0 & -\frac{1}{6} & 0 & -\frac{1}{6} & \frac{5}{6} & 0 & -\frac{1}{6} & 0 & 0 & -\frac{1}{6} & 0 & -\frac{1}{6} \\ -\frac{1}{6} & 0 & -\frac{1}{6} & 0 & 0 & \frac{5}{6} & 0 & -\frac{1}{6} & -\frac{1}{6} & 0 & -\frac{1}{6} & 0 \\ 0 & -\frac{1}{6} & 0 & -\frac{1}{6} & -\frac{1}{6} & 0 & \frac{5}{6} & 0 & 0 & -\frac{1}{6} & 0 & -\frac{1}{6} \\ -\frac{1}{6} & 0 & -\frac{1}{6} & 0 & 0 & -\frac{1}{6} & 0 & \frac{5}{6} & -\frac{1}{6} & 0 & -\frac{1}{6} & 0 \\ -\frac{1}{6} & 0 & -\frac{1}{6} & 0 & 0 & -\frac{1}{6} & 0 & -\frac{1}{6} & \frac{5}{6} & 0 & -\frac{1}{6} & 0 \\ 0 & -\frac{1}{6} & 0 & -\frac{1}{6} & -\frac{1}{6} & 0 & -\frac{1}{6} & 0 & 0 & \frac{5}{6} & 0 & -\frac{1}{6} \\ -\frac{1}{6} & 0 & -\frac{1}{6} & 0 & 0 & -\frac{1}{6} & 0 & -\frac{1}{6} & -\frac{1}{6} & 0 & \frac{5}{6} & 0 \\ 0 & -\frac{1}{6} & 0 & -\frac{1}{6} & -\frac{1}{6} & 0 & -\frac{1}{6} & 0 & 0 & -\frac{1}{6} & 0 & \frac{5}{6} \end{pmatrix}$$

If we multiply by six, the pattern is a bit more obvious.

$$\frac{1}{6}\begin{pmatrix} 5 & 0 & -1 & 0 & 0 & -1 & 0 & -1 & -1 & 0 & -1 & 0 \\ 0 & 5 & 0 & -1 & -1 & 0 & -1 & 0 & 0 & -1 & 0 & -1 \\ -1 & 0 & 5 & 0 & 0 & -1 & 0 & -1 & -1 & 0 & -1 & 0 \\ 0 & -1 & 0 & 5 & -1 & 0 & -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & 0 & -1 & 5 & 0 & -1 & 0 & 0 & -1 & 0 & -1 \\ -1 & 0 & -1 & 0 & 0 & 5 & 0 & -1 & -1 & 0 & -1 & 0 \\ 0 & -1 & 0 & -1 & -1 & 0 & 5 & 0 & 0 & -1 & 0 & -1 \\ -1 & 0 & -1 & 0 & 0 & -1 & 0 & 5 & -1 & 0 & -1 & 0 \\ -1 & 0 & -1 & 0 & 0 & -1 & 0 & -1 & 5 & 0 & -1 & 0 \\ 0 & -1 & 0 & -1 & -1 & 0 & -1 & 0 & 0 & 5 & 0 & -1 \\ -1 & 0 & -1 & 0 & 0 & -1 & 0 & -1 & -1 & 0 & 5 & 0 \\ 0 & -1 & 0 & -1 & -1 & 0 & -1 & 0 & 0 & -1 & 0 & 5 \end{pmatrix}$$

The pseudoinverse is a great tool that returns an exact result. The problem is that the calculation is so long that we are excluded from studying many physically relevant systems. In optics applications, for example, it is common to encounter systems with sizes like 48×48 and 64×64. In the following table, we show some computa-

tional times for small systems, indicating that real physical applications can be problematic.

System size	Computation time, s
2×2	0.11
2×3	0.361
3×2	0.43
3×3	91.192
3×4	1218
4×3	1416
4×4	31,891
5×5	Kernel crash

No more memory available.
Mathematica kernel has shut down.
Try quitting other applications and then retry.

In short, in the time it takes to run a singular value decomposition on a 64×64 system, the pseudoinverse has completed only a 5×5 system. This suggests to us that when dealing with most applications, the singular values decomposition is a better choice. However, there is now a problem. In version 5.0 the command used to perform the SVD, `SingularValues Decomposition`, was declared obsolete. Given the magnitude of the impending disaster, we want to show the exact wording used to explain this predicament.

> `SingularValues` has been superseded by `SingularValueList` and `SingularValueDecomposition`. `SingularValueDecomposition` uses a different and more complete definition.

Built-in Functions > New in Version 5.0 > Obsolete in Version 5 > SingularValues.

Unfortunately, *The CRC Concise Encyclopedia of Mathematics* also references the obsolete command.

Singular Value Decomposition

Though the future of this vital tool seems uncertain, we shall press forth with the demonstration.

Let's pass A to the routine `SingularValues`. The attempt fails.

```
(* decompose A *)
{u, md, v} = SingularValues[A];
```

```
SingularValues::svdf :
  SingularValues has received a matrix with infinite precision . More...

Set::shape :
  Lists {u, md, v} and SingularValues [{{1, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0},
      <<10>>, {0, 0, 0, 0, <<4>>, 0, 0, 0, 1}}] are not the same shape . More...
```

The comment displayed tells us that A has *infinite precision*. This means that A is composed of integers. We can easily remedy this problem by instead passing `N[A]`.

```
(* decompose N[A] *)
{u, md, v} = SingularValues[N[A]];
(* assemble inverse surrogate *)
Ainv = Chop[Transpose[v].DiagonalMatrix[1/md].u];
(* look at the SVD result *)
Ainv // MatrixForm
```

0.861111	0	-0.222222	0	0	0.0277778	0	-0.305556	-0.138889	0	-0.222222	0
0	0.444444	0	-0.222222	0.111111	0	-0.0555556	0	0	-0.0555556	0	-0.222222
-0.222222	0	0.444444	0	0	-0.0555556	0	0.111111	-0.222222	0	-0.0555556	0
0	-0.222222	0	0.861111	-0.305556	0	0.0277778	0	0	-0.222222	0	-0.138889
0	0.111111	0	-0.305556	0.527778	0	-0.138889	0	0	0.111111	0	-0.305556
0.0277778	0	-0.0555556	0	0	0.194444	0	-0.138889	0.0277778	0	-0.0555556	0
0	-0.0555556	0	0.0277778	-0.138889	0	0.194444	0	0	-0.0555556	0	0.0277778
-0.305556	0	0.111111	0	0	-0.138889	0	0.527778	-0.305556	0	0.111111	0
-0.138889	0	-0.222222	0	0	0.0277778	0	-0.305556	0.861111	0	-0.222222	0
0	-0.0555556	0	-0.222222	0.111111	0	-0.0555556	0	0	0.444444	0	-0.222222
-0.222222	0	-0.0555556	0	0	-0.0555556	0	0.111111	-0.222222	0	0.444444	0
0	-0.222222	0	-0.138889	-0.305556	0	0.0277778	0	0	-0.222222	0	0.861111

This form worked. What we have just done is to decompose the matrix A into three separate matrices:

$$A = U\Sigma V^T \tag{2.17}$$

where U and V have special properties and Σ is a diagonal matrix composed of the *singular values*. The surrogate inverse is then composed using this recipe:

$$A^{(-1)} = V\frac{1}{\Sigma}U^T \tag{2.18}$$

In our *Mathematica* notebook we call the matrix resulting from equation 2.18 `Ainv`. The obvious question is how does `Ainv` compare to `piA`? When we difference the two matrices, we see this:

```
(* compare the Moore-Penrose pseudoinverse *)
(* to the SVD solution *)
piA - Ainv
```

```
{{2.22045×10^-16, 0, -1.66533×10^-16, 0, 0, -3.81639×10^-17,
  0, -5.55112×10^-17, 2.498×10^-16, 0, -2.77556×10^-17, 0},
 {0, 0., 0, 5.55112×10^-17, 6.93889×10^-17, 0, -1.73472×10^-16, 0, 0,
  1.04083×10^-16, 0, -5.55112×10^-17}, {0., 0, 0., 0, 0, 6.93889×10^-18,
  0, 4.16334×10^-17, -2.77556×10^-17, 0, -3.46945×10^-17, 0},
 {0, -1.66533×10^-16, 0, 2.22045×10^-16, -5.55112×10^-16, 0,
  5.20417×10^-17, 0, 0, -2.22045×10^-16, 0, 4.996×10^-16},
 {0, 1.94289×10^-16, 0, -2.22045×10^-16, 2.22045×10^-16, 0,
  -5.55112×10^-17, 0, 0, 4.16334×10^-17, 0, -4.44089×10^-16},
 {-1.31839×10^-16, 0, 4.85723×10^-17, 0, 0, -1.11022×10^-16,
  0, 8.32667×10^-17, -4.85723×10^-17, 0, 1.04083×10^-16, 0},
 {0, -5.55112×10^-17, 0, 8.32667×10^-17, 0., 0, 5.55112×10^-17, 0,
  0, -2.77556×10^-17, 0, 4.51028×10^-17}, {1.11022×10^-16, 0, 0., 0,
  0, 2.77556×10^-17, 0, 0., -5.55112×10^-17, 0, -1.38778×10^-17, 0},
 {-1.11022×10^-16, 0, 2.77556×10^-17, 0, 0, 7.28584×10^-17, 0,
  -1.66533×10^-16, -1.11022×10^-16, 0, 2.77556×10^-17, 0},
 {0, 6.93889×10^-18, 0, -8.32667×10^-17, 2.63678×10^-16, 0,
  -4.85723×10^-17, 0, 0, -5.55112×10^-17, 0, -1.66533×10^-16},
 {0., 0, -4.16334×10^-17, 0, 0, 3.46945×10^-17, 0, 0., 0., 0, 0., 0},
 {0, 0., 0, 4.44089×10^-16, -2.22045×10^-16,
  0, 1.70003×10^-16, 0, 0, 0., 0, 2.22045×10^-16}}
```

This is a bit tedious to sift through, so we will rely on the *Mathematica* command `Chop` to force the small numbers to zero. The result is far more pleasing.

```
(* clear away the machine noise *)
Chop[%] // MatrixForm
```

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

We see that the Moore-Penrose pseudo-inverse is an algorithm that gives exact answers equivalent to the obsolete `SingularValues`. A serious downside is that the staggering time penalty of the pseudoinverse drives us to use the obsolete command. Unfortunately, this command is slated to disappear shortly and the replacement command `SingularValuesDecomposition` does not yield the result we need.

⚠ Wolfram Research is planning to abandon SingularValues, an important tool for solving singular linear systems.

Symbolic exploration

We want to demonstrate how easy it is to check the identities from *The CRC Concise Encyclopedia of Mathematics*, equations 2.11-2.14. We'll use the matrix `A` and its pseudoinverse `piA` from the previous example. Notice how the use of `Equal` test (==) gives us a wonderful result.

$$m m^{+} m = m \tag{2.11}$$

```
A.piA.A == A
```

```
True
```

$$m^{+} m m^{+} = m^{+} \tag{2.12}$$

```
piA.A.piA == piA
```

```
True
```

$$(mm^{+})^{T} = mm^{+} \tag{2.13}$$

```
Transpose[A.piA] == A.piA
```

```
True
```

$$(m^{+}m)^{T} = m^{+}m \tag{2.14}$$

```
Transpose[piA.A] == piA.A
```

```
True
```

Equation 2.16 tells us that the pseudoinverse minimizes the squared difference between the product of the matrix and its pseudoinverse and the identity matrix. In this case the difference is 2.

```
diff = (A.piA - IdentityMatrix[Length[A]])^2;
diff // MatrixForm
Plus @@ Plus @@ diff
```

$$\begin{pmatrix}
\frac{1}{36} & 0 & \frac{1}{36} & 0 & 0 & \frac{1}{36} & 0 & \frac{1}{36} & \frac{1}{36} & 0 & \frac{1}{36} & 0 \\
0 & \frac{1}{36} & 0 & \frac{1}{36} & \frac{1}{36} & 0 & \frac{1}{36} & 0 & 0 & \frac{1}{36} & 0 & \frac{1}{36} \\
\frac{1}{36} & 0 & \frac{1}{36} & 0 & 0 & \frac{1}{36} & 0 & \frac{1}{36} & \frac{1}{36} & 0 & \frac{1}{36} & 0 \\
0 & \frac{1}{36} & 0 & \frac{1}{36} & \frac{1}{36} & 0 & \frac{1}{36} & 0 & 0 & \frac{1}{36} & 0 & \frac{1}{36} \\
0 & \frac{1}{36} & 0 & \frac{1}{36} & \frac{1}{36} & 0 & \frac{1}{36} & 0 & 0 & \frac{1}{36} & 0 & \frac{1}{36} \\
\frac{1}{36} & 0 & \frac{1}{36} & 0 & 0 & \frac{1}{36} & 0 & \frac{1}{36} & \frac{1}{36} & 0 & \frac{1}{36} & 0 \\
0 & \frac{1}{36} & 0 & \frac{1}{36} & \frac{1}{36} & 0 & \frac{1}{36} & 0 & 0 & \frac{1}{36} & 0 & \frac{1}{36} \\
\frac{1}{36} & 0 & \frac{1}{36} & 0 & 0 & \frac{1}{36} & 0 & \frac{1}{36} & \frac{1}{36} & 0 & \frac{1}{36} & 0 \\
\frac{1}{36} & 0 & \frac{1}{36} & 0 & 0 & \frac{1}{36} & 0 & \frac{1}{36} & \frac{1}{36} & 0 & \frac{1}{36} & 0 \\
0 & \frac{1}{36} & 0 & \frac{1}{36} & \frac{1}{36} & 0 & \frac{1}{36} & 0 & 0 & \frac{1}{36} & 0 & \frac{1}{36} \\
\frac{1}{36} & 0 & \frac{1}{36} & 0 & 0 & \frac{1}{36} & 0 & \frac{1}{36} & \frac{1}{36} & 0 & \frac{1}{36} & 0 \\
0 & \frac{1}{36} & 0 & \frac{1}{36} & \frac{1}{36} & 0 & \frac{1}{36} & 0 & 0 & \frac{1}{36} & 0 & \frac{1}{36}
\end{pmatrix}$$

```
2
```

2.3 Linear regression

Now let's move onto to the archetypical example linear regression, a least squares fit to a straight line. There is a standard package, referenced below, that will do linear regression for us. But the point of this section is not to perform the regression, but to learn *Mathematica* by seeing how it is employed to solve familiar problems.

2.3.1 Derivation

Consider a set on *n* measurements of the form *(x, y)*. The goal of a linear regression is to find the slope *m* and intercept *b* for line which minimizes a *merit function*. In general, the merit function for a least squares fit is of the form:

$$\chi^2 = \sum_{i=1}^{n} \frac{(measurement_i - prediction_i)^2}{\sigma_i^2} \tag{2.19}$$

Here the prediction will be $(m \pm \sigma_m,\ b \pm \sigma_b)$. For the case of equal measurement uncertainties at each point we can write the merit function χ^2 as

$$\chi^2 = \sum_{i=1}^{n} (y_i - m x_i - b)^2 \tag{2.20}$$

Canonical minimization finds the zeros of the derivatives in parameter space:

$$\partial_m \chi^2 = 0 \text{ and } \partial_b \chi^2 = 0 \tag{2.21}$$

These simultaneous equations lead to the direct solution

$$m \pm \sigma_m = \Delta^{-1} \left(n \sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i \pm s \sqrt{\Delta n} \right) \tag{2.22}$$

and

$$b \pm \sigma_b = \Delta^{-1} \left(\sum_{i=1}^{n} x_i^2 \sum_{i=1}^{n} y_i - \sum_{i=1}^{n} x_i \sum_{i=1}^{n} x_i y_i \pm s \sqrt{\Delta \sum_{i=1}^{n} x_i} \right) \tag{2.23}$$

where the determinant Δ is computed as

$$\Delta = n \sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i \right) \tag{2.24}$$

and the square of the estimated variance is:

$$s^2 = \frac{\sum_{i=1}^{n} (y_i - mx_i - b)^2}{n-2}. \tag{2.25}$$

An effective quality metric is the correlation index:

$$r_c^2 = 1 - \frac{\sum_{i=1}^{n} (y_i - mx_i - b)^2}{\left(\sum_{i=1}^{n} y_i \right)^2}. \tag{2.26}$$

Correlation Index, Least Squares Fitting

2.3.2 Basic implementation

We begin with a list of measurements.

```
(* data from a linear regression simulator *)
data = {{0, 0.002164}, {0.1, 0.184447}, {0.2, 0.386971},
   {0.3, 0.582324}, {0.4, 0.80787}, {0.5, 1.061785}, {0.6, 1.231706},
   {0.7, 1.430628}, {0.8, 1.613865}, {0.9, 1.80023}, {1, 2.007923}};
```

We need to write modules to perform the regression and output the results. The functionality will be split like this

Read in the data Build the accumulators	Compute fit parameters and quality measures	Print the results
Accumulate	*Analyze*	*Present*

Table 2.1: Dividing the function for a linear regression.

Dividing the modules up like this makes the code much easier to read and to debug and allows you to swap components between other routines.

The first module operates on a list `data` in memory:

```
(* linear regression accumulation module *)
(* operates on global data list *)
Module[{x, y},
  n = Length[data];          (* count the data points *)
  (* clear the accumulators *)
  {X, XX, Y, YX, χ2} = {0, 0, 0, 0, 0};
  (* begin loop over the n data points to build accumulators *)
  Do[
   (* extract x and y *)
   x = data[[i]][[1]]; y = data[[i]][[2]];
   (* begin accumulating *)
   X += x;    (* sum of the x terms *)
   XX += x^2; (* sum of the squares *)
   Y += y;    (* sum of the y terms *)
   YX += y x; (* sum of product *)
   (* end loop over the n data points *)
   , {i, n}];
 ];
```

To make the code easier to read and to understand we extract the quantities *x* and *y* from the list `data`. Then we loop through the data set and augment the accumulators in what we hope is an explicit manner.

The outputs of this module are the accumulators `X`, `XX`, `Y`, `YX` and the number of points n. These are passed to the analysis module.

```
(* linear regression analysis module *)
(* operates on global list of accumulators *)
Module[{},
  (* fit parameters *)
  Δ = n XX - X^2;                    (* determinant *)
  m = Δ^-1 (n YX - Y X);             (* slope *)
  b = Δ^-1 (XX Y - X YX);            (* intercept *)
  (* statistical properties *)
  (* begin loop over n data points to sum regression errors *)
  Do[
   (* extract x and y *)
   x = data[[i]][[1]]; y = data[[i]][[2]];
   (* fit error at point i *)
   χ2 += (y - m x - b)^2;
   (* end loop over the n data points *)
   , {i, n}];
  s2 = χ2 / (n - 2);                 (* estimated sample variance *)
  (* fit quality parameters *)
  σm = Sqrt[(s2 / Δ) n];             (* slope error *)
  σb = Sqrt[(s2 / Δ) X];             (* intercept error *)
  rc2 = 1 - χ2 / Y^2;                (* correlation index *)
 ];
```

Now we have completed the computations and are ready to print the data.

```
(* output *)
Module[{},
  Print["results for the linear regression of ", n, " points:"];
  Print[""];
  Print["m = ", m, " ± ", σm];
  Print["b = ", b, " ± ", σb];
  Print[""];
  Print["correlation index = ", rc2];
 ];
```

```
results for the linear regression of 11 points :

m = 2.02664 ± 0.0223098

b = -0.0033285 ± 0.0157754

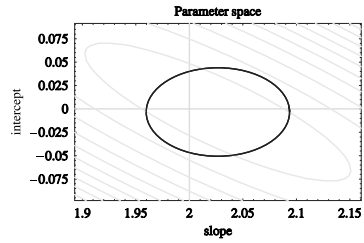
correlation index = 0.99996
```

One of the most common problems seen when a least squares fit is not done properly is that there is no computation for the error in the fit parameters. This tells us how well the data matched the model and how well we know the slope and intercept.

We must always show the data against the fit. This allows for a qualitative evaluation. We have seen far too many cases where people debugged valid code. This is because they have not viewed their data and they have assumed the program must be wrong. Also, we must plot the solution in parameter space when possible. After viewing this plot you will instantly accept its utility. The final part is to plot the residuals and see whether they are uncorrelated.

The next chapter presents the basics of *Mathematica*'s prodigious graphics routines and we will defer detailed explanation of the plots till then. For now we will simply present them.

First, we show the parameter space plot which reveals the accuracy and precision of the plot.



In this plot we are no longer in a physical space where x and y were measured. Instead, this is a *parameter space* plot. The parameter space is the abstract space where the problem was minimized. The horizontal axis represents slope and the vertical axis represents intercept. We can see the isoχ^2 contours as a faint background and it appears that the solution point is at the minimum. The crosshairs mark the "true solution." By this we mean the values of slope and intercept that we put into the module that generated the data points.

The distance between the intersection of the crosshairs (the ideal result) and the single point (the measured result) represents our accuracy. This tells us how close we came to the true answer. The ellipse centered about the single point is an error ellipse. This ellipse has radii $\{\sigma_m, \sigma_b\}$ and it represents the precision with which the result is known.

The last two plots are shown together.

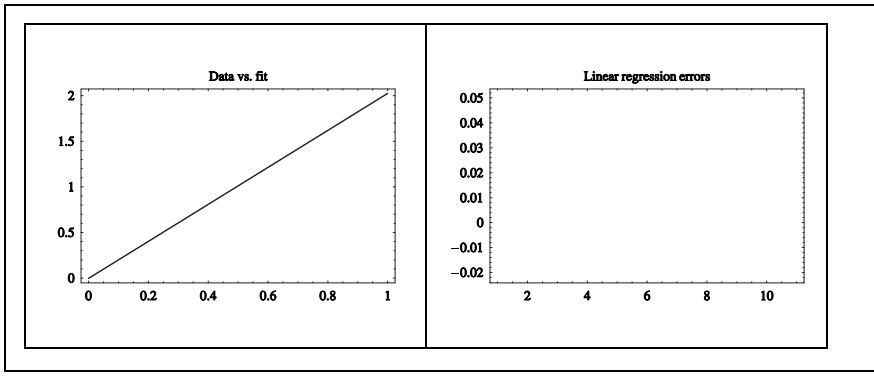


Table 2.2: Plots from the linear regression. The plot on the left displays the data superimposed over the solution curve. The plot on the right shows the regression errors at each point. Ideal residuals are uncorrelated.

2.3.3 List-based tools

At this point we see that *Mathematica* has a basic yet powerful tool kit which is quite intuitive. If this is your initial introduction, then we suggest that you spend some time to familiarize yourself with the basics as you become comfortable in the interactive environment. As you work with *Mathematica* you will interact with the on-line help

and hopefully begin to broaden your knowledge and increase your understanding of the language. You will learn how the interpreter helps you to make sure that you have closed all your parentheses, how to interact with the operating system, how to read the error messages, etc.

Advanced users or users familiar with programming languages like *Scheme* may wish to immediately begin with the more sophisticated features. We will use the linear regression example to explore vector operations. We begin some List exercises.

Consider the data set above in the variable data. How would we take this list and generate a list of x values and a list of y values? A simple approach is to consider a single element, say $\{0,0.002164\}$. How would we project out the x value? If we are in a basis with unit vectors $\hat{x}$ and y, we can use the dot product with the basis vectors to project out the amplitudes. *Mathematica*lly the operation can be expressed as:

$$(a\hat{x} + b\hat{y}) \cdot \hat{x} = a \tag{2.27}$$

where the basis vectors are orthonormal. That is,

$$\hat{x} \cdot \hat{x} = 0, \hat{y} \cdot \hat{y} = 0, \text{ and } x \cdot y = 0 \tag{2.28}$$

We can isolate the x and y values with this simple prescription based on equation 2.28.

```
(* define orthonormal basis vectors *)
x̂ = {1, 0};
ŷ = {0, 1};
(* project out x component *)
x = {0, 0.002164}.x̂
(* project out y component *)
y = {0, 0.002164}.ŷ

0.
```

```
0.002164
```

We can see how to project out terms for a single measurement, but how would we do it for the entire list? The answer is very simple: apply the projection operator to the entire list.

```
(* project out the x components *)
x = data.x̂
(* project out the y components *)
y = data.ŷ

{{0, 0.002164}, {0.1, 0.184447}, {0.2, 0.386971},
  {0.3, 0.582324}, {0.4, 0.80787}, {0.5, 1.06179}, {0.6, 1.23171},
  {0.7, 1.43063}, {0.8, 1.61387}, {0.9, 1.80023}, {1, 2.00792}}.0̂.
```

```
{{0, 0.002164}, {0.1, 0.184447}, {0.2, 0.386971}, {0.3, 0.582324},
  {0.4, 0.80787}, {0.5, 1.06179}, {0.6, 1.23171}, {0.7, 1.43063},
  {0.8, 1.61387}, {0.9, 1.80023}, {1, 2.00792}}.0.00̂2164
```

Unfortunately, we have a major problem here. The dot product failed and as a diagnostic *Mathematica* has demonstrated the operation it was trying to perform. The gist of the problem is that the vector x has been overwritten; it is no longer $\{1, 0\}$. Instead it has taken on the components of the x vector that we found previously. The fundamental problem is that x and $\hat{x}$ are not separate variables. A trivial fix is to use the script character set as shown here.

```
(* scalable way to build orthonormal basis vectors *)
{x, y} = IdentityMatrix[2];
(* project out all x components *)
x = data.x
(* project out all y components *)
y = data.y

{0., 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.}
```

```
{0.002164, 0.184447, 0.386971, 0.582324, 0.80787,
 1.06179, 1.23171, 1.43063, 1.61387, 1.80023, 2.00792}
```

We should explore the statement assigning the basis vectors their values. We could have written:

```
(* manual construction of orthonormal basis vectors *)
x = {1, 0};
y = {0, 1};
```

But we realize the values on the right-hand side form an identity matrix. In the spirit of foreshadowing we choose to use the identity matrix since this will scale with the dimension of the problem. More on this later. By now it should be clear that the list assignment:

$$\{\text{lista}_{[1]}, \text{lista}_{[2]}, \text{lista}_{[3]}, \ldots\} = \{\text{listb}_{[1]}, \text{listb}_{[2]}, \text{listb}_{[3]}, \ldots\}$$

is shorthand for:

$$\begin{aligned} \text{lista}_{[1]} &= \text{listb}_{[1]} \\ \text{lista}_{[2]} &= \text{listb}_{[2]} \\ \text{lista}_{[3]} &= \text{listb}_{[3]} \\ &\vdots \end{aligned}$$

We hope that you can appreciate the ease and elegance of these vector operations. Another exercise we can consider is to take the two lists we have above and blend them into the original form of data. This is a wise thing to do because if you have made a series of *y* measurements at a series of *x* values, these two sets should be combined into one object, a data object. Divorcing the *x* and *y* values and maintaining separate lists are common mistakes of the novitiate and should be avoided. They can lead to confusion and data corruption. The only time you may wish to consider divorcing the lists is if you have made a large number of measurements with one *x* set. Here it would be wasteful to store the *x* values millions of times and in this case, you would want to use the first value of your *y* list to indicate which *x* set is applicable.

We will take these two lists and create a measurement object which we will call **series1**. We have covered this basic method in the last chapter, so we hope this concrete example is quite clear. Observe how simple this is.

```
(* merge the x list and the y list *)
series1 = {x, y}

{{0., 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.},
 {0.002164, 0.184447, 0.386971, 0.582324, 0.80787,
  1.06179, 1.23171, 1.43063, 1.61387, 1.80023, 2.00792}}
```

When we do, we realize that we can transpose the matrix:

```
(* a transpose will give us the form we need *)
MatrixForm[series1 = Transpose[series1]]
```

$$\begin{pmatrix} 0. & 0.002164 \\ 0.1 & 0.184447 \\ 0.2 & 0.386971 \\ 0.3 & 0.582324 \\ 0.4 & 0.80787 \\ 0.5 & 1.06179 \\ 0.6 & 1.23171 \\ 0.7 & 1.43063 \\ 0.8 & 1.61387 \\ 0.9 & 1.80023 \\ 1. & 2.00792 \end{pmatrix}$$

and recover a measurement object:

```
(* the form we started with *)
series1
```

```
{{0., 0.002164}, {0.1, 0.184447}, {0.2, 0.386971},
 {0.3, 0.582324}, {0.4, 0.80787}, {0.5, 1.06179}, {0.6, 1.23171},
 {0.7, 1.43063}, {0.8, 1.61387}, {0.9, 1.80023}, {1., 2.00792}}
```

2.3.4 A more elegant implementation

Before we return to the linear regression problem we want to explore the vector operations a bit more. We begin with the case of multiple measurements shown here. We have performed five sets of measurements shown below. We have varied the shadings to simplify the content of the results object.

Let's go back to the point where we separate the *x* and *y* vectors.

```
(* scalable way to build orthonormal basis vectors *)
{x, y} = IdentityMatrix[2];
(* project out all x components *)
x = data.x
(* project out all y components *)
y = data.y

{0., 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.}
```

```
{0.002164, 0.184447, 0.386971, 0.582324, 0.80787,
 1.06179, 1.23171, 1.43063, 1.61387, 1.80023, 2.00792}
```

We can use *Mathematica*'s impressive assortment of commands to prepare our sums without writing a `Do` loop.

```
(* vector form for summations *)
(* notice we still are measuring the size of the list *)
n = Length[data];   (* count the number of data points *)
X = Plus @@ x;      (* sum the x values *)
XX = Plus @@ x^2;   (* sum the x^2 values *)
Y = Plus @@ y;      (* sum the y values *)
YX = x.y;           (* sum the x y values *)
```

One thing is certain: you learn by doing a lot faster than you learn by reading. We encourage you to experiment with the code fragments on your own. Try to write quick diagnostics when in doubt. You can study the function of the `MapAll` operator (`@@`) with a few lines of code.

After you read the Help Browser entries, you should experiment for yourself and see how the commands work. The first step is to build a small list.

```
(* small list for experimentation *)
r = Range[5]

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

We expect the `Plus` operator to sum this list, so we will use the `Sum` command to first calculate the sum.

```
(* sum all the elements *)
Sum[r[[i]], {i, 5}]

15
```

Now we can compare this result to mapping the `Plus` operator across the list.

```
(* test mapping procedure *)
Plus @@ r

15
```

Another presentation which explicitly shows the action uses characters in lieu of numbers.

```
(* use variables this time *)
r = CharacterRange["a", "e"]

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

Mathematica is perfectly happy to add symbols.

```
(* sum the symbols *)
Sum[r[[i]], {i, 5}]

a + b + c + d + e
```

The internal representation is revealing for it shows us that *Mathematica* is using a `Plus` operator as the foundation for the sum.

```
(* examine how Mathematica describes this internally *)
FullForm[%]

Plus["a", "b", "c", "d", "e"]
```

Now we can compare this result to mapping the `Plus` operator across the list.

```
(* compare to the sum *)
Plus @@ r

a + b + c + d + e
```

We see that the results for the `Sum` and the `Plus` operators are identical because *Mathematica* resolves them into the same internal representation.

```
(* the internal forms are identical *)
FullForm[%]

Plus["a", "b", "c", "d", "e"]
```

Hopefully, this is all quite clear, and the extension to summing χ^2 is transparent. But since we have encountered users confused as to whether we are computing the sum of the squares or the square of the sums, we will show how to resolve this question. Look at the values for the list `r` and the list r^2.

```
(* remember how easy it is to manipulate lists *)
r
r^2

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

```
{a^2, b^2, c^2, d^2, e^2}
```

Mapping the `Plus` operator across the list of squared elements provides us with the sum of squared elements.

```
(* we can easily sum the squares *)
Plus @@ r^2

a^2 + b^2 + c^2 + d^2 + e^2
```

Mapping the `Plus` operator across the list of squared elements provides us with the sum of squared elements.

Notice that we can compute the sum of the squares with the dot product.

```
(* we can use the dot product too *)
r.r
```

$$a^2 + b^2 + c^2 + d^2 + e^2$$

Those were different ways to compute the sum of the squares. The square of the sum can be computed using:

```
(* this the square of the sum, not the sum of the squares *)
(Plus @@ r)^2
```

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

Again the point of these exercises is not so much to resolve questions about mapping but to demonstrate to the reader how to explore these issues.

Having clarified the methods of summation we are ready to compute our results. We are able to use the same code from the module.

```
(* this code is from the module above *)
Δ = n XX - X^2;                    (* determinant *)
m = Δ^-1 (n YX - Y X);             (* slope *)
b = Δ^-1 (XX Y - X YX);            (* intercept *)
```

Can we eliminate the do-loop used to compute the residuals? Yes and we will use *Mathematica* to show us how. We expand a symbolic form of the merit function being careful not to use the variables *m* and *b* since they have numerical values from the computation above.

```
(* let Mathematica do this multiplication *)
Expand[(yi - M xi - B)^2]
```

$$B^2 + 2\,B\,M\,xi + M^2\,xi^2 - 2\,B\,yi - 2\,M\,xi\,yi + yi^2$$

See how we did not use an actual list element, but surrogates like `yi`. This `i` will mark the terms for summation. We need the result from summing this quantity of all n points. That sum becomes:

```
(* equivalent formulation *)
Sum[B^2 + 2 B M x[[i]] + M^2 x[[i]]^2 - 2 B y[[i]] - 2 M x[[i]] y[[i]] + y[[i]]^2, {i, 1, n}]
```

One thing that seems to consistently confuse people is the evaluation of the sum of a constant like B^2. They don't seem to connect the pieces. The formula below shows that when we sum k copies of B^2, we get $k\,B^2$.

```
(* a simple test *)
Sum[B^2, {i, 1, k}]
```

```
B^2 k
```

With this form we can eliminate a second sweep through the data to compute χ^2.

We can pull all this together and write some purely list-based operations. We have gone a step farther in exploiting the vector nature of the data. Notice that the new accumulator below does not even need to isolate the x and y components.

```
(* accumulation using vectors *)
accum[a_List] := Module[{m},
   (* number of data points *)
   n = Length[data];
   (* sum the x and y components *)
   {X, Y} = Plus @@ data;
   (* composite quantities *)
   m = Transpose[data].data;
   {XX, YY, YX} = {m[[1]][[1]], m[[2]][[2]], m[[1]][[2]]};
   (* collect the outputs *)
   registers = {n, X, Y, XX, YX, YY};
  ];
```

The analysis module has been tightened a little bit. The parameter errors are computed in list form. The χ^2 has been changed to be summation free.

```
(* linear regression analysis engine *)
(* argument pattern: {n, X, Y, XX, YX, YY} *)
analyze[a_List] := Module[{Δ},
   Δ = a[[1]] a[[4]] - a[[2]]^2;                    (* determinant *)
   m = Δ^-1 (a[[1]] a[[5]] - a[[3]] a[[2]]);         (* slope *)
   b = Δ^-1 (a[[4]] a[[3]] - a[[2]] a[[5]]);         (* intercept *)
   (* rapid χ^2 *)
   chi2 = a[[1]] b^2 + 2 m b a[[2]] + m^2 a[[4]] - 2 b a[[3]] - 2 m a[[5]] + a[[6]];
   (* estimated sample variance *)
   s2 = chi2/(a[[1]] - 2);
   (* parameter errors *)
   {σm, σb} = Sqrt[s2/Δ] Sqrt[{a[[1]], a[[2]]}];
   (* correlation index *)
   rc2 = 1 - chi2/a[[3]]^2;
  ];
```

The publication module is basically the same.

```
publish := Module[{},
   Print["results for the linear regression of ", n, " points:"];
   Print[""];
   Print["m = ", m, " ± ", σm];
   Print["b = ", b, " ± ", σb];
   Print[""];
   Print["correlation index = ", rc2];
  ];
```

Notice that there are no brackets after the module name `publish`. To do the analysis we call the modules sequentially. In actual use, you would probably want a cover module that would call the three of them.

```
(* linear regression *)
accum[data];
analyze[registers];
publish
```

```
results for the linear regression of 11 points :

m = 2.02664 ± 0.0223098

b = -0.0033285 ± 0.0157754

correlation index = 0.99996
```

2.3.5 An important cross-check

By way of completeness, we show how *Mathematica* let's us check our answer easily. We define the merit function using equation 2.20.

```
(* define the merit function *)
f[m_, b_] := Sum_{i=1}^{n} (y[[i]] - m x[[i]] - b)^2
```

We pass this function to `FindMinimum` and ask it to minimize the function with respect to the parameters *m* and *b*. An initial guess for these parameters is required, so we start both parameters at one.

```
(* minimize the merit function *)
(* finds linear regression parameters m and b *)
FindMinimum[f[m, b], {{m, 1}, {b, 1}}]

{0.00492752, {m → 2.02664, b → -0.0033285}}
```

This is a quick and painless way to check part of the answer. You'll notice that the very important error terms are missing. The best way to use `FindMinimum` is at the

beginning to check the quality of the data set. If you have problems with your data, you want to resolve them before writing and debugging your code.

Add-ons & Links > Standard Packages > LinearRegression

2.3.6 Extending the example

In this section we'll see how easy it is to extend concepts naturally with the list-based tools. Consider now a set of data from five measurement series. We have manually varied the shading between measurement sets to help the reader visualize the data more easily.

```
(* the result of five measurement series *)
results =
 {{{0., -0.017676}, {0.2, 0.40787}, {0.4, 0.861785}, {0.6, 1.231706},
   {0.8, 1.630628}, {1., 2.013865}}, {{0, 0.00023}, {0.2, 0.407923},
   {0.4, 0.79291}, {0.6, 1.225163}, {0.8, 1.647544}, {1., 2.018279}},
  {{0., -0.001209}, {0.2, 0.417067}, {0.4, 0.803969},
   {0.6, 1.236133}, {0.8, 1.615108}, {1., 2.022991}},
  {{0., 0.006654}, {0.2, 0.419374}, {0.4, 0.774871},
   {0.6, 1.222758}, {0.8, 1.558579}, {1., 2.032099}},
  {{0., 0.023023}, {0.2, 0.426533}, {0.4, 0.829841},
   {0.6, 1.24384}, {0.8, 1.621614}, {1., 2.02398}}};
```

What happens when we use the *x* projection operator?

```
(* we can easily project out the x components *)
results.x

{{0., 0.2, 0.4, 0.6, 0.8, 1.},
 {0., 0.2, 0.4, 0.6, 0.8, 1.}, {0., 0.2, 0.4, 0.6, 0.8, 1.},
 {0., 0.2, 0.4, 0.6, 0.8, 1.}, {0., 0.2, 0.4, 0.6, 0.8, 1.}}
```

We see that we have projected out the *x* values for all five measurements. This is equivalent to the do-loop:

```
(* dimensions are *)
(* {number of measurements, *)
(*  number of data points in each measurement, *)
(*  dimension of the measurement space} *)
n = Dimensions[results];
xlists = {};          (* x values for each measurement *)
Do[                    (* loop over measurements *)
 x = {};
 (* container for x values for the ith measurement *)
 Do[
  (* loop over the data points in each measurement *)
  AppendTo[x, results[[i]][[j]][[1]]];
  , {j, n[[2]]}];
 (* loop over the data points in each measurement *)
 AppendTo[xlists, x];
 , {i, n[[1]]}];          (* loop over measurements *)
(* show the lists of x values *)
xlists
```

```
Syntax::sntxf :
  "AppendTo [x, <<7>> <<1>> [[j]][[1]]]; " cannot be followed by ", {j, n[[2]]}". More...

xlists = {}; Do[x = {}; Do[AppendTo [x, results [[i]][[j]][[1]]];
    , {j, n[[2]]}]; AppendTo [xlists , x];, {i, n[[1]]}]; xlists
```

Unfortunately, out of nowhere an error has occurred. We appear to have some invalid syntax. However, we know this is not the case for the program worked flawlessly when we last saved it. The problem is caused by an embedded comment. When we read the error message, we see that it points to the first `Do` loop which used `j` for a counter. So we turn our attention to the last comment before that.

The cure for this malady seems counterintuitive: simply cut the comment out and then immediately paste it back in. While this might strike the reader as tedious, this will restore normal operation. The good news is that there is no problem with the code and the fix is trivial. The bad news is that this is fairly common. We have the problem in several of the notebooks used to write this book.

⚠ Beware of embedded comments. They can cause error messages that incorrectly imply you have problems with your code.

This kind of behavior can absolutely vex new users. For this reason we caution the user to avoid in-line comments and internal comments. Comments are most safely relegated to their own lines outside of the functional code as shown below.

```
(* comments like this outside *)
(* of operational statements are benign *)
```

You'll notice that we lighten the comments. This is not required and is simply a personal preference that allows the code to stand out more. It is also possible to use colors in your text and comments.

The dot product formulation is far simpler to read and to implement. Some may counter that learning the vector notation takes more time. However, the cumbersome nature of the do-loop actually impedes progress and dissuades one from attempting more complex problems. In the end, the pain required to learn the new paradigm will make using the program far easier.

Suppose, for example, you wanted a list of the form {{*x* values at first data point}, {*x* values at second data point},...}. We could create a do-loop like we did above, or we could simply write:

```
(* at what x value was y measured at? *)
Transpose[results].x

{{0., 0., 0., 0., 0.}, {0.2, 0.2, 0.2, 0.2, 0.2},
 {0.4, 0.4, 0.4, 0.4, 0.4}, {0.6, 0.6, 0.6, 0.6, 0.6},
 {0.8, 0.8, 0.8, 0.8, 0.8}, {1., 1., 1., 1., 1.}}
```

We can now compare the *y* values at each data point.

```
(* compare the results of measurements at identical points *)
yvalues = Transpose[results].y

{{-0.017676, 0.00023, -0.001209, 0.006654, 0.023023},
 {0.40787, 0.407923, 0.417067, 0.419374, 0.426533},
 {0.861785, 0.79291, 0.803969, 0.774871, 0.829841},
 {1.23171, 1.22516, 1.23613, 1.22276, 1.24384},
 {1.63063, 1.64754, 1.61511, 1.55858, 1.62161},
 {2.01387, 2.01828, 2.02299, 2.0321, 2.02398}}
```

A data analyst would look at these measurements at the same position (the same x value) and immediately want to see the average and standard deviation. We will explore the hierarchy of ways to compute these quantities.

```
(* module to find the average *)
(* and standard deviation of a list *)
(* of values, a *)
```

```
average[a_List] := Module[{n, X, X2},
  n = Length[a];                (* measure the list *)
  {X, X2} = {0, 0};             (* accumulators *)
  Do[                           (* loop over data points *)
   X += a[[i]];
   X2 += a[[i]]^2;
   , {i, n}];                   (* loop over data points *)
  X /= n; X2 /= n;              (* compute averages *)
  {α, σ} = {X, √(X2 - X^2)};    (* average and standard deviation *)
  Print["average = ", α, " ± ", σ];
 ];
```

At this point we can apply this module to the data, looking at the y values of the first data point in each measurement.

```
(* call the module to compute the average of the yvalues *)
average[yvalues[[1]]]
```

```
average = 0.0022044 ± 0.0131453
```

How would we evaluate the average for every data point? Here is a simplistic way.

```
(* fragment to compute the average *)
(* at each data point for each *)
(* measurement series *)
n = Length[yvalues];
Do[
 average[yvalues[[i]]]
 , {i, n}]
```

```
average  = 0.0022044  ± 0.0131453
average  = 0.415753  ± 0.00713438
average  = 0.812675  ± 0.0303383
average  = 1.23192  ± 0.00760895
average  = 1.61469  ± 0.0300997
average  = 2.02224  ± 0.00610936
```

Notice that in this fragment we use *n* to count the number of measurements (5). Inside the module average *n* is used to count the number of data points in each measurement (6). Because we made *n* a local variable, in average it does not conflict with the global *n*.

As usual, there are many different ways to do the same thing in *Mathematica*. Let's look at a vector operation to perform the same operation. We will use the **Apply operator** `@@` that we saw previously. This operator applies a function across an array. This time we will study how functions are applied across arrays. Look at the example below

```
(* insure that these variables have no values *)
Clear[a, b, c, d, e, f];
(* the first step in exploring the effect of MapAll (@@) *)
Plus @@ {a, b, c}

a + b + c
```

If you are confused by the shorthand notation, we show the command form too.

```
(* long form for Plus@@{a,b,c} *)
Apply[Plus, {a, b, c}]

a + b + c
```

The inquiring reader will wonder how this operation acts on multidimensional lists. As always, the example tells the story.

```
Plus @@ {{a, b, c}, {d, e, f}}

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

Remember that in `MatrixForm` the operand looks like:

```
{{a, b, c}, {d, e, f}} // MatrixForm

(a b c)
(d e f)
```

So we see that the `Apply` operator is summing columns. But what if we want it to sum rows? Then we just operate on the transpose.

```
(* sum rows, not columns *)
Plus @@ Transpose[{{a, b, c}, {d, e, f}}]

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

Let's end this topic with a look at ways to sum the entire array. When we apply the `Plus` operator to the array, we get the expected:

```
(* sum the columns *)
Plus @@ {{a, b, c}, {d, e, f}}

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

This array can then be summed using the `Apply` operator:

```
Plus @@ %

a + b + c + d + e + f
```

We could also have written:

```
(* sum all elements in a 2D array *)
Plus @@ Plus @@ {{a, b, c}, {d, e, f}}

a + b + c + d + e + f
```

The drawback with this method is that an *n*-dimensional array has *n* - 1 occurrences and who wants to write all of that? After all, a great beauty of *Mathematica* code is how closely it resembles the mathematical equations that we write on paper. For example, if we write *a*·*b* we understand that *a* and *b* can have any common dimension. In many computing languages we would have to construct subroutines for every dimension the problem might have. Modern languages allow us to conceal these separate operations with polymorphism, but we still are spending our time writing computer code to handle the different cases. One quick remedy is to remove the structure from the contents with the `Flatten` command:

```
(* removes all the braces except two *)
Flatten[{{a, b, c}, {d, e, f}}]

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

There are no longer any rows or columns, just the data. We only need a single application of `Plus@@` now:

```
Plus @@ %

a + b + c + d + e + f
```

To close this discussion we show how to take our flattened array and recreate the matrix using the `Partition` command.

```
Partition[{a, b, c, d, e, f}, 3]
```

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

At last we return to the averaging exercise. The following steps should be clear:

```
(* we want to work with the transpose matrix *)
tyvalues = Transpose[yvalues];
(* show the results in matrix form *)
(* columns now correspond to successive *)
(* measurements at the same points *)
tyvalues // MatrixForm
(* count the number of measurements at this point *)
n = Length[tyvalues];
(* sum the common measurements *)
X = Plus @@ Transpose[yvalues] / n
(* sum the squares of the common measurements *)
X2 = Plus @@ Transpose[yvalues^2] / n
```

```
⎛ -0.017676   0.40787   0.861785  1.23171  1.63063  2.01387 ⎞
⎜  0.00023    0.407923  0.79291   1.22516  1.64754  2.01828 ⎟
⎜ -0.001209   0.417067  0.803969  1.23613  1.61511  2.02299 ⎟
⎜  0.006654   0.419374  0.774871  1.22276  1.55858   2.0321 ⎟
⎝  0.023023   0.426533  0.829841  1.24384  1.62161  2.02398 ⎠
```

```
{0.0022044, 0.415753, 0.812675, 1.23192, 1.61469, 2.02224}
```

```
{0.000177658, 0.172902, 0.661361, 1.51768, 2.60814, 4.0895}
```

It makes sense now to take the transpose of the two output vectors to again cluster the data measurement series instead of measurement points.

```
averages = Transpose[{X, X2}]

{{0.0022044, 0.000177658}, {0.415753, 0.172902}, {0.812675, 0.661361},
 {1.23192, 1.51768}, {1.61469, 2.60814}, {2.02224, 4.0895}}
```

Now we need to operate on the second number in each pair to convert it from the average of the squares to the standard deviation. In other words, we want some process that will take a data pair and perform this operation.

$$\text{second number} \to \sqrt{\text{second number} - (\text{first number})^2} \tag{2.29}$$

One way to implement such an operation is to define a function like this.

```
(* compute the standard deviation given
  the average and the average of the squares *)
sdev[a_List] := {a[[1]], Sqrt[a[[2]] - a[[1]]^2]};
```

We could then apply this across the data pairs to yield the final result.

```
sdev /@ averages

{{0.0022044, 0.0131453}, {0.415753, 0.00713438},
 {0.812675, 0.0303383}, {1.23192, 0.00760895},
 {1.61469, 0.0300997}, {2.02224, 0.00610936}}
```

Yet another approach would be to write a function using the slot operator `#`. To illustrate how the slot operator works we'll compose a short function `fcn` and apply it to the argument `[a, b, c]`.

```
(* demonstrate the slot operator *)
fcn =
  (Print["slot 1 = ", #1, "; slot 2 = ", #2, "; slot 3 = ", #3]) &;
fcn[a, b, c];
```

```
slot 1 = a; slot 2 = b; slot 3 = c
```

Of course, the slot operator can reference inside of lists.

```
(* examine operation of slot operator on arrays *)
fcn =
   (Print["first element = ", #[[1]], "; second element = ", #[[2]]]) &;
 (* create an array *)
 array = {{a, b}, {c, d}};
 (* examine MatrixForm *)
 array // MatrixForm
 (* apply the function *)
 fcn[{{a, b}, {c, d}}];
```

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

```
first element = {a, b}; second element = {c, d}
```

We can reference into lists at any level:

```
(* grab specific subelements *)
fcn = (Print["first sub element = ",
     #[[1]][[1]], "; second sub element = ", #[[1]][[2]]]) &;
fcn[{{a, b}, {c, d}}];
```

```
first sub element = a; second sub element = b
```

Buoyed by this knowledge we can write a new standard deviation function using the slot operator:

```
(* compact form of standard deviationfunction *)
sd = ({#[[1]], Sqrt[#[[2]] - #[[1]]^2]}) &;
```

and apply it to the list `averages` to obtain the final result:

```
(* compute the standard deviations *)
sd /@ averages

{{0.0022044, 0.0131453}, {0.415753, 0.00713438},
 {0.812675, 0.0303383}, {1.23192, 0.00760895},
 {1.61469, 0.0300997}, {2.02224, 0.00610936}}
```

Of course, the advantage of having repeated measurements comes from analyzing the aggregate data. How do we take the five data sets and put them in the proper format? We will use one `Flatten` statement.

```
(* how do we run all the data at once? *)
data = Flatten[results, 1]

{{0., -0.017676}, {0.2, 0.40787}, {0.4, 0.861785},
 {0.6, 1.23171}, {0.8, 1.63063}, {1., 2.01387}, {0, 0.00023},
 {0.2, 0.407923}, {0.4, 0.79291}, {0.6, 1.22516}, {0.8, 1.64754},
 {1., 2.01828}, {0., -0.001209}, {0.2, 0.417067}, {0.4, 0.803969},
 {0.6, 1.23613}, {0.8, 1.61511}, {1., 2.02299}, {0., 0.006654},
 {0.2, 0.419374}, {0.4, 0.774871}, {0.6, 1.22276},
 {0.8, 1.55858}, {1., 2.0321}, {0., 0.023023}, {0.2, 0.426533},
 {0.4, 0.829841}, {0.6, 1.24384}, {0.8, 1.62161}, {1., 2.02398}}
```

All the previous routines for analysis and plotting will operate on this list. The analysis results are these.

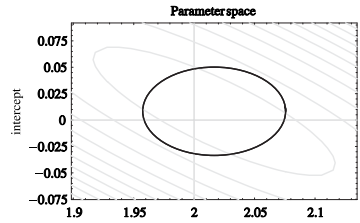
```
results for the linear regression of 30 points :

m = 2.01661 ± 0.0591095

b = 0.00827742 ± 0.0417968

correlation index = 0.999987
```

Here is the plot showing the result in parameter space.



Finally, we conclude the analysis presentation with a look at the solution line and the residuals.

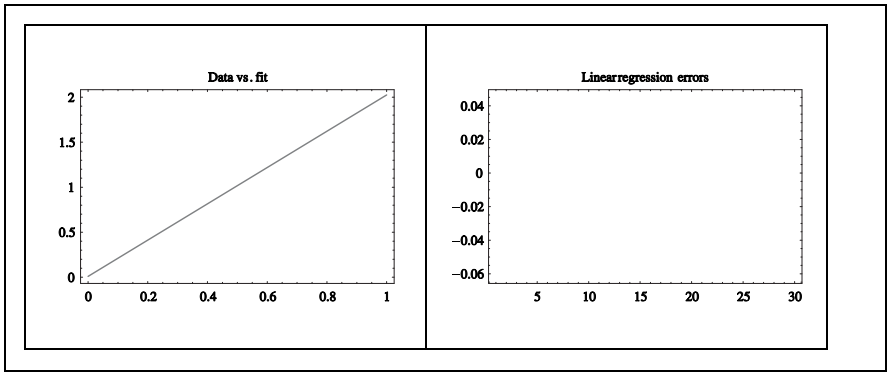


Table 2.3: Plots from the linear regression of five data sets. The plot on the left displays the data superimposed over the solution curve. The plot on the right shows the regression errors at each point. Ideal residuals are uncorrelated.

Perhaps at this point the reader is expecting us to demonstrate how to bring *Mathematica*'s impressive symbolic powers to bear on the mathematics demonstrated in equations 2.21-2.25. Certainly these problems are intricate. However, we will use *Mathematica* to perform an even better trick: we will demonstrate a way to set up and solve polynomials least squares fits that does not require us to solve the equations. These solutions are most easily visualized.

The Mathematica Book > Principles of Mathematica > Functional Operators > Pure Function

Built-in Functions > Programming > Functional Programming > Function (&)

Function

2.3.7 Higher powers

Mathematica really shines when it comes to vector operations which comprise the preferred method for handling higher dimensions problems. We'll start with a simple

problem handled explicitly and from that we'll generalize the order of the polynomial and the dimension of the fit.

Readers with a background in linear algebra know that we can think of polynomials as comprising a vector space. We'll use an example of a quadratic polynomial in a 2D space. If the polynomial is:

$$P^2(x,\ y) = a_{00} + a_{10}x + a_{01}y + a_{20}x^2 + a_{02}y^2 + a_{11}xy \tag{2.30}$$

we can write the components as:

$$= (a_{00},\ a_{10},\ a_{01},\ a_{20},\ a_{02},\ a_{11} \tag{2.31}$$

We call a the amplitude vector. We define a space vector s by:

$$= (1,\ x,\ y,\ x^2,\ xy,\ y^2 \tag{2.32}$$

and define the polynomial to be:

$$^{2}(x,\ y) = a \cdot\ s \tag{2.33}$$

Look closely at the space vector. We can define a power vector p which holds the powers for the terms in the space vector as

$$: ((0,\ 0),\ ((1,\ 0),\ (0,\ 1)),\ ((2,\ 0),\ (1,\ 1),\ (0,\ 2 \tag{2.34}$$

Notice that the indexing on the a vector matches the values of the p vector. So we know for example that the coefficient a_{ij} multiplies the term $x^i y^j$.

This is pleasant, but when we do the matrices, we will be forced to use a linear ordering scheme. Think of the result of flattening p. We would find that $p_1 = (0,\ 0)$, $p_2 = (1,\ 0)$ and so on until $p_6 = (0,\ 2)$.

We define the merit function as

$$\chi^2 = \sum_{i=1}^{n} (P_i - a \cdot\ s_i)^2 \tag{2.35}$$

where s_i represents the space vector evaluated with the values $\{x_i, y_i\}$. We minimize this function in the canonical fashion by simultaneously setting all derivatives with respect to the fit parameters equal to zero.

Consider the derivative with respect to a_μ where we use Greek letters to denote the position in the vectors a, s, and p. Roman letters are used to mark data points.

$$\partial_{a_\mu}\chi^2 = \sum_{i=1}^{n} (P_i - a \cdot\ s_i)s_\mu = 0. \tag{2.36}$$

This generates a series of ω equations linear in the fitting parameters. Here ω represents the length of the a, s, and p vectors. The solution to the linear system is the a vector which minimizes equation 2.35.

The linear system that we will set up is:

$$Aa = B \tag{2.37}$$

We want to show the current case specifically and then show how to handle the general case.

Proceeding, our six linear equations have the forms:

$$a_{00}\sum_{i=1}^{n}1+a_{10}\sum_{i=1}^{n}x_i+a_{01}\sum_{i=1}^{n}y_i+a_{20}\sum_{i=1}^{n}x_i^2+a_{11}\sum_{i=1}^{n}x_iy_i+a_{02}\sum_{i=1}^{n}y_i^2=\sum_{i=1}^{n}P_i$$

$$a_{00}\sum_{i=1}^{n}x_i+a_{10}\sum_{i=1}^{n}x_i^2+a_{01}\sum_{i=1}^{n}x_iy_i+a_{20}\sum_{i=1}^{n}x_i^3+a_{11}\sum_{i=1}^{n}x_i^2y_i+a_{02}\sum_{i=1}^{n}x_iy_i^2=\sum_{i=1}^{n}P_ix_i$$

$$a_{00}\sum_{i=1}^{n}y_i+a_{10}\sum_{i=1}^{n}x_iy_i+a_{01}\sum_{i=1}^{n}y_i^2+a_{20}\sum_{i=1}^{n}x_i^2y_i+a_{11}\sum_{i=1}^{n}x_iy_i^2+a_{02}\sum_{i=1}^{n}y_i^3=\sum_{i=1}^{n}P_iy_i$$

$$a_{00}\sum_{i=1}^{n}x_i^2+a_{10}\sum_{i=1}^{n}x_i^3+a_{01}\sum_{i=1}^{n}x_i^2y_i+a_{20}\sum_{i=1}^{n}x_i^4+a_{11}\sum_{i=1}^{n}x_i^3y_i+a_{02}\sum_{i=1}^{n}x_i^2y_i^2=\sum_{i=1}^{n}P_ix_i^2$$

$$a_{00}\sum_{i=1}^{n}x_iy_i+a_{10}\sum_{i=1}^{n}x_i^2y_i+a_{01}\sum_{i=1}^{n}x_iy_i^2+a_{20}\sum_{i=1}^{n}x_i^3y_i^2+a_{11}\sum_{i=1}^{n}x_i^2y_i^2+a_{02}\sum_{i=1}^{n}x_iy_i^3$$

$$=\sum_{i=1}^{n}P_ix_iy_i$$

$$a_{00}\sum_{i=1}^{n}y_i^2+a_{10}\sum_{i=1}^{n}x_iy_i^2+a_{01}\sum_{i=1}^{n}y_i^3+a_{20}\sum_{i=1}^{n}x_i^2y_i^2+a_{11}\sum_{i=1}^{n}x_iy_i^3+a_{02}\sum_{i=1}^{n}y_i^4=\sum_{i=1}^{n}P_iy_i^2$$

Consider the first equation closely and see how the subsequent equations relate to it. You will see that the μth equation is formed by multiplying the first equation by s_μ. This is, of course, the prescription in equation 2.36 where we have moved the measured values, the P_i terms, to the right-hand side of the equation.

This problem represents a very simple system: a quadratic equation in two dimensions. But already any attempt to manipulate this and larger systems seems to invite ennui. Clearly, any desire to solve more realistic systems compels one to explore other avenues of solution. We will discuss more elegant resolutions in later sections.

Continuing with the solution, these equations form the linear system:

$$\begin{bmatrix} n & \sum x_i & \sum y_i & \sum x_i^2 & \sum x_i y_i & \sum y_i^2 \\ \sum x_i & \sum x_i^2 & \sum x_i y_i & \sum x_i^3 & \sum x_i^2 y_i & \sum x_i y_i^2 \\ \sum y_i & \sum x_i y_i & \sum y_i^2 & \sum x_i^2 y_i & \sum x_i y_i^2 & \sum y_i^3 \\ \sum x_i & \sum x_i^3 & \sum x_i^2 y_i & \sum x_i^4 & \sum x_i^3 y_i & \sum x_i^2 y_i^2 \\ \sum x_i y_i & \sum x_i^2 y_i & \sum x_i y_i^2 & \sum x_i^3 y_i & \sum x_i^2 y_i^2 & \sum x_i y_i^3 \\ \sum y_i^2 & \sum x_i y_i^2 & \sum y_i^3 & \sum x_i^2 y_i^2 & \sum x_i y_i^3 & \sum y_i^4 \end{bmatrix} \begin{bmatrix} a_{00} \\ a_{10} \\ a_{01} \\ a_{20} \\ a_{11} \\ a_{02} \end{bmatrix} = \begin{bmatrix} \sum P_i \\ \sum P_i x_i \\ \sum P_i y_i \\ \sum P_i x_i^2 \\ \sum P_i x_i y_i \\ \sum P_i y_i^2 \end{bmatrix} \tag{2.38}$$

represented as:

$$A \cdot a = B \tag{2.37}$$

We apologize to readers who find this level of detail tedious and unnecessary, but experience has proven over and over that this kind of definition is needed to connect people to a process that many find overly abstract.

Two observations should come from looking at equation 2.38. Look at the A matrix on the left-hand side. The term in the μth column and νth row is:

$$A_{\mu\nu} = \sum x_i^{p_{\mu_1} + p_{\nu_1}} y_i^{p_{\mu_2} + p_{\nu_2}} \tag{2.39}$$

also, the term in the μth column of the B matrix is

$$B_\mu = \sum P_i x_i^{p_{\mu_1}} y_i^{p_{\mu_2}} \tag{2.40}$$

From equation 2.39 it is now clear that the A matrix must be symmetric. This is expressed by the arithmetic fact that

$$p_\mu + p_\nu = p_\nu + p_\mu \tag{2.41}$$

Before moving forward the reader must be convinced that the properties in equations 2.39-2.41 are reflected in equation 2.38.

With the A and B components in hand, the amplitude vector, the solution, is

$$a = A^{-1} \cdot B \tag{2.42}$$

Bypassing the algebra

By using the p vector we have shown how to generalize these polynomial regression problems. We no longer need to even bother with the derivatives and the linear equations. We can boost the degree of fit d and increase the dimension of the space D. We have a prescription in equations 2.39 and 2.40 that tells us how to build the entire linear system.

Now we will use *Mathematica*'s vector nature to further simplify matters; we will not use the `Length` command for any computation. Notice how a fit of degree d needs to sum the spatial variables through order $2d$. Think of q as the list of $\{x, y\}$

points where the measurement was made. As we have shown, it is a trivial matter to separate the list into component lists X and Y. If we maintain this list structure, we see that we need lists like $\{X, X^2, X^3, \ldots\}$ and $\{Y, Y^2, Y^3, \ldots\}$. We can use `Plus@@` to easily build the needed sums on these lists, and we can use the dot product to build the mixed sums like $\sum x_i y_i$.

Consider the construction of the list:

$$X = \{X^0,\ X^1,\ X^2,\ \ldots,\ X^{2d}\}. \tag{2.43}$$

The first component is the list of x values to the zeroth power. This requires some reflection because we must allow for measurements at the origin which means that we will have zeros in our lists and 0^0 is an ambiguous quantity which can be either zero or one. We show the two limiting conditions below.

```
(* drive the exponent to 0 *)
Limit[2^x, x → 0]

1
```

```
(* drive the argument to 0 *)
Limit[x^2, x → 0]

0
```

In this case we will interpret 0^0 as unity. This means that X^0 will be a list of n ones. The good news is that we can then build successive powers of X by simply multiplying by x. A demonstration of building the **X** list should clarify the process. First, we identify a rather typical set of measurement locations on a grid.

```
(* generate a set of measurement locations on a grid *)
addr = Table[{i, j}, {i, 0, 1, 1/4}, {j, 0, 1, 1/4}]
```

$$\left\{\left\{\{0,0\},\left\{0,\frac{1}{4}\right\},\left\{0,\frac{1}{2}\right\},\left\{0,\frac{3}{4}\right\},\{0,1\}\right\},\right.$$
$$\left\{\left\{\frac{1}{4},0\right\},\left\{\frac{1}{4},\frac{1}{4}\right\},\left\{\frac{1}{4},\frac{1}{2}\right\},\left\{\frac{1}{4},\frac{3}{4}\right\},\left\{\frac{1}{4},1\right\}\right\},$$
$$\left\{\left\{\frac{1}{2},0\right\},\left\{\frac{1}{2},\frac{1}{4}\right\},\left\{\frac{1}{2},\frac{1}{2}\right\},\left\{\frac{1}{2},\frac{3}{4}\right\},\left\{\frac{1}{2},1\right\}\right\},$$
$$\left\{\left\{\frac{3}{4},0\right\},\left\{\frac{3}{4},\frac{1}{4}\right\},\left\{\frac{3}{4},\frac{1}{2}\right\},\left\{\frac{3}{4},\frac{3}{4}\right\},\left\{\frac{3}{4},1\right\}\right\},$$
$$\left.\left\{\{1,0\},\left\{1,\frac{1}{4}\right\},\left\{1,\frac{1}{2}\right\},\left\{1,\frac{3}{4}\right\},\{1,1\}\right\}\right\}$$

This has more structure than we want. If you look, each address has an $\{i, j\}$ address. We want a simple list of points which is addressed linearly.

```
(* we need a simple list of points *)
addr = Flatten[addr, 1]
```

$$\left\{\{0,0\},\left\{0,\frac{1}{4}\right\},\left\{0,\frac{1}{2}\right\},\left\{0,\frac{3}{4}\right\},\{0,1\},\left\{\frac{1}{4},0\right\},\right.$$
$$\left\{\frac{1}{4},\frac{1}{4}\right\},\left\{\frac{1}{4},\frac{1}{2}\right\},\left\{\frac{1}{4},\frac{3}{4}\right\},\left\{\frac{1}{4},1\right\},\left\{\frac{1}{2},0\right\},\left\{\frac{1}{2},\frac{1}{4}\right\},$$
$$\left\{\frac{1}{2},\frac{1}{2}\right\},\left\{\frac{1}{2},\frac{3}{4}\right\},\left\{\frac{1}{2},1\right\},\left\{\frac{3}{4},0\right\},\left\{\frac{3}{4},\frac{1}{4}\right\},\left\{\frac{3}{4},\frac{1}{2}\right\},$$
$$\left.\left\{\frac{3}{4},\frac{3}{4}\right\},\left\{\frac{3}{4},1\right\},\{1,0\},\left\{1,\frac{1}{4}\right\},\left\{1,\frac{1}{2}\right\},\left\{1,\frac{3}{4}\right\},\{1,1\}\right\}$$

We will now compute the value of the polynomial at each of these addresses. Let's create the overhead to describe our 2D space.

```
(* degree of the problem: quadratic *)
d = 2;
(* number of terms we are dealing with *)
t = 1/2 (d + 1) (d + 2);
```

Next comes the table of powers for the x and y components of each monomial in the space vector. When we create the vector, it will have a superfluous level of internal structure that we need to remove. This is how we created the vector shown in equation 2.34.

```
(* vector of powers *)
Table[{i - j, j}, {i, 0, d}, {j, 0, i}]
p = Flatten[%, 1]

{{{0, 0}}, {{1, 0}, {0, 1}}, {{2, 0}, {1, 1}, {0, 2}}}
```

```
{{0, 0}, {1, 0}, {0, 1}, {2, 0}, {1, 1}, {0, 2}}
```

With these exponents we can build the space vector as shown in equation 2.32.

```
(* space vector *)
s = Table[x^p[[i]][[1]] y^p[[i]][[2]], {i, t}]

{1, x, y, x^2, x y, y^2}
```

We are ready to specify an amplitude vector a. If this code is confusing, just type the list in yourself. We wanted to create a list of amplitudes that we could easily associate with each order. For example, the amplitude for order zero is one; for order one, it is two, etc.

```
(* amplitude vector which defines polynomial *)
a = (#[[1]] + #[[2]] + 1) &/@ p

{1, 2, 2, 3, 3, 3}
```

When we follow the prescription of equation 2.33, we get the desired result shown in equation 2.30.

```
(* build the polynomial *)
f = a.s

1 + 2 x + 3 x^2 + 2 y + 3 x y + 3 y^2
```

Our next action is to construct a list of measurements. This represents the type of data we might take in the laboratory. The most basic strategy is to use the `Table` command.

```
(* count the measurement locations *)
nd = Length[addr];
(* build a list of measurements in the form {x, y, f(x, y)} *)
data = Table[
  {addr[[i]][[1]], addr[[i]][[2]], f /. x → addr[[i]][[1]] /. y → addr[[i]][[2]]}
  , {i, nd}]
```

$$\left\{\{0, 0, 1\}, \left\{0, \frac{1}{4}, \frac{27}{16}\right\}, \left\{0, \frac{1}{2}, \frac{11}{4}\right\}, \left\{0, \frac{3}{4}, \frac{67}{16}\right\}, \{0, 1, 6\},\right.$$
$$\left\{\frac{1}{4}, 0, \frac{27}{16}\right\}, \left\{\frac{1}{4}, \frac{1}{4}, \frac{41}{16}\right\}, \left\{\frac{1}{4}, \frac{1}{2}, \frac{61}{16}\right\}, \left\{\frac{1}{4}, \frac{3}{4}, \frac{87}{16}\right\},$$
$$\left\{\frac{1}{4}, 1, \frac{119}{16}\right\}, \left\{\frac{1}{2}, 0, \frac{11}{4}\right\}, \left\{\frac{1}{2}, \frac{1}{4}, \frac{61}{16}\right\}, \left\{\frac{1}{2}, \frac{1}{2}, \frac{21}{4}\right\},$$
$$\left\{\frac{1}{2}, \frac{3}{4}, \frac{113}{16}\right\}, \left\{\frac{1}{2}, 1, \frac{37}{4}\right\}, \left\{\frac{3}{4}, 0, \frac{67}{16}\right\}, \left\{\frac{3}{4}, \frac{1}{4}, \frac{87}{16}\right\},$$
$$\left\{\frac{3}{4}, \frac{1}{2}, \frac{113}{16}\right\}, \left\{\frac{3}{4}, \frac{3}{4}, \frac{145}{16}\right\}, \left\{\frac{3}{4}, 1, \frac{183}{16}\right\}, \{1, 0, 6\},$$
$$\left.\left\{1, \frac{1}{4}, \frac{119}{16}\right\}, \left\{1, \frac{1}{2}, \frac{37}{4}\right\}, \left\{1, \frac{3}{4}, \frac{183}{16}\right\}, \{1, 1, 14\}\right\}$$

However, as evidenced by the **Length** command, we are not exploiting *Mathematica*'s vector powers. Here is a form that does, using a pure function.

```
(* build the measurements in the form {x, y, f(x, y)} *)
data = ({#[[1]], #[[2]], f /. x → #[[1]] /. y → #[[2]]}) & /@ addr
```

$$\left\{\{0, 0, 1\}, \left\{0, \frac{1}{4}, \frac{27}{16}\right\}, \left\{0, \frac{1}{2}, \frac{11}{4}\right\}, \left\{0, \frac{3}{4}, \frac{67}{16}\right\}, \{0, 1, 6\},\right.$$
$$\left\{\frac{1}{4}, 0, \frac{27}{16}\right\}, \left\{\frac{1}{4}, \frac{1}{4}, \frac{41}{16}\right\}, \left\{\frac{1}{4}, \frac{1}{2}, \frac{61}{16}\right\}, \left\{\frac{1}{4}, \frac{3}{4}, \frac{87}{16}\right\},$$
$$\left\{\frac{1}{4}, 1, \frac{119}{16}\right\}, \left\{\frac{1}{2}, 0, \frac{11}{4}\right\}, \left\{\frac{1}{2}, \frac{1}{4}, \frac{61}{16}\right\}, \left\{\frac{1}{2}, \frac{1}{2}, \frac{21}{4}\right\},$$
$$\left\{\frac{1}{2}, \frac{3}{4}, \frac{113}{16}\right\}, \left\{\frac{1}{2}, 1, \frac{37}{4}\right\}, \left\{\frac{3}{4}, 0, \frac{67}{16}\right\}, \left\{\frac{3}{4}, \frac{1}{4}, \frac{87}{16}\right\},$$
$$\left\{\frac{3}{4}, \frac{1}{2}, \frac{113}{16}\right\}, \left\{\frac{3}{4}, \frac{3}{4}, \frac{145}{16}\right\}, \left\{\frac{3}{4}, 1, \frac{183}{16}\right\}, \{1, 0, 6\},$$
$$\left.\left\{1, \frac{1}{4}, \frac{119}{16}\right\}, \left\{1, \frac{1}{2}, \frac{37}{4}\right\}, \left\{1, \frac{3}{4}, \frac{183}{16}\right\}, \{1, 1, 14\}\right\}$$

With data in hand we are ready to begin the computation of the amplitudes. Our first step is to resolve the components of the measurement object. When you look at the following dot products, the identity matrix should be manifest.

```
(* isolate measurements by component *)
x = data.{1, 0, 0}
y = data.{0, 1, 0}
F = data.{0, 0, 1}
```

$$\left\{0, 0, 0, 0, 0, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{3}{4}, \frac{3}{4}, \frac{3}{4}, \frac{3}{4}, \frac{3}{4}, 1, 1, 1, 1, 1\right\}$$

$$\left\{0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1, 0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1, 0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1, 0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1, 0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1\right\}$$

$$\left\{1, \frac{27}{16}, \frac{11}{4}, \frac{67}{16}, 6, \frac{27}{16}, \frac{41}{16}, \frac{61}{16}, \frac{87}{16}, \frac{119}{16}, \frac{11}{4}, \frac{61}{16}, \frac{21}{4}, \frac{113}{16}, \frac{37}{4}, \frac{67}{16}, \frac{87}{16}, \frac{113}{16}, \frac{145}{16}, \frac{183}{16}, 6, \frac{119}{16}, \frac{37}{4}, \frac{183}{16}, 14\right\}$$

We will use these lists to build the computation quantities symbolized in list 2.43. The strategy for all components is the same. For clarity, we will choose the specific example of *X*. We lay the foundation first.

```
(* we exploit vector arithmetic to build the foundation x^0 *)
X = {x - x + 1}
```

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

The next term is gained by multiplying the last term in *X* by *x*. This rule is true not only for this term, but also for all subsequent terms.

```
(* this is the next term in the series *)
Last[X] x
```

$$\left\{0, 0, 0, 0, 0, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{3}{4}, \frac{3}{4}, \frac{3}{4}, \frac{3}{4}, \frac{3}{4}, 1, 1, 1, 1, 1\right\}$$

The new term is appended to the *X* list.

```
(* we append the latest term *)
AppendTo[X, %]
```

$$\left\{\{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1\}, \left\{0, 0, 0, 0, 0, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{3}{4}, \frac{3}{4}, \frac{3}{4}, \frac{3}{4}, \frac{3}{4}, 1, 1, 1, 1, 1\right\}\right\}$$

This recipe is easy. In this fashion we never manually compute things like x^8. Instead we gradually build up $x \cdot x \cdot x \cdot x \cdot x \cdot x$, quite an efficient use of CPU time. Here is what the loops are to compute *X* and *Y*:

```
(* build up power vectors *)
Clear[X];
(* vector of 1s *)
X = {x - x + 1};
(* loop through x powers *)
Do[
  (* new power *)
  new = Last[X] x;
  (* add to list *)
  AppendTo[X, new];
  , {i, 2 d}];
Clear[Y];
(* vector of 1s *)
Y = {y - y + 1};
(* loop through y powers *)
Do[
  (* new power *)
  new = Last[Y] y;
  (* add to list *)
  AppendTo[Y, new];
  , {i, 2 d}];
```

Of course we should always check our results to monitor quality and to visualize the process, particularly when the concept is new. The reader should ensure that the power structure is evident.

```
(* check the results in MatrixForm *)
X // MatrixForm
```

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & \frac{3}{4} & \frac{3}{4} & \frac{3}{4} \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{16} & \frac{1}{16} & \frac{1}{16} & \frac{1}{16} & \frac{1}{16} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{9}{16} & \frac{9}{16} & \frac{9}{16} \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{64} & \frac{1}{64} & \frac{1}{64} & \frac{1}{64} & \frac{1}{64} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{1}{8} & \frac{27}{64} & \frac{27}{64} & \frac{27}{64} \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{256} & \frac{1}{256} & \frac{1}{256} & \frac{1}{256} & \frac{1}{256} & \frac{1}{16} & \frac{1}{16} & \frac{1}{16} & \frac{1}{16} & \frac{1}{16} & \frac{81}{256} & \frac{81}{256} & \frac{81}{256} \end{pmatrix}$$

A little bit is off the screen, but we can view the matrix in its entirety in *Mathematica*.

Proceeding we can build A using the prescription in equation 2.39.

```
(* build the A matrix *)
A = Table[k = p[[i]] + p[[j]]; X[[k[[1]]+1]] . Y[[k[[2]]+1]], {i, t}, {j, t}];
```

Notice that because *Mathematica* forces us to use one-based arrays (i.e., the first index is always one), we can't have a zero-based array. This forces us to shift the index *k* up by one. Similar action is required for the *B* vector.

```
(* build the B vector *)
B = Table[k = p[[i]]; Plus @@ (F X[[k[[1]]+1]] Y[[k[[2]]+1]]), {i, t}];
```

You'll notice in the case of the *A* matrix there is no multiplication or summation. We express the matrix elements as a dot product. This is an elegant way to view the problem. To reinforce this point, we will consider a specific case.

```
(* pick some components *)
X[[2]]
Y[[3]]
```

$$\left\{0, 0, 0, 0, 0, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{3}{4}, \frac{3}{4}, \frac{3}{4}, \frac{3}{4}, \frac{3}{4}, 1, 1, 1, 1, 1\right\}$$

$$\left\{0, \frac{1}{16}, \frac{1}{4}, \frac{9}{16}, 1, 0, \frac{1}{16}, \frac{1}{4}, \frac{9}{16}, 1, 0, \frac{1}{16}, \frac{1}{4}, \frac{9}{16}, 1, 0, \frac{1}{16}, \frac{1}{4}, \frac{9}{16}, 1, 0, \frac{1}{16}, \frac{1}{4}, \frac{9}{16}, 1\right\}$$

The dot product is trivial to write.

```
(* compute the dot product *)
X[[2]] . Y[[3]]
```

$$\frac{75}{16}$$

Compare this formulation to the equivalent `Do` loop.

```
(* count the terms *)
n = Length[X[[2]]];
(* clear summation variable *)
term = 0;
(* loop through the terms *)
Do[
  (* sum the product *)
  term += X[[2]][[i]] Y[[3]][[i]]
  , {i, n}];
(* output the result *)
term
```

$$\frac{75}{16}$$

We encourage our readers to set a personal goal of eventually learning how to use the vector tools.

Returning to the calculation, we always find it helpful to look at the terms in equation 2.38.

```
(* notice the symmetry *)
A // MatrixForm
```

$$\begin{pmatrix} 25 & \frac{25}{2} & \frac{25}{2} & \frac{75}{8} & \frac{25}{4} & \frac{75}{8} \\ \frac{25}{2} & \frac{75}{8} & \frac{25}{4} & \frac{125}{16} & \frac{75}{16} & \frac{75}{16} \\ \frac{25}{2} & \frac{25}{4} & \frac{75}{8} & \frac{75}{16} & \frac{75}{16} & \frac{125}{16} \\ \frac{75}{8} & \frac{125}{16} & \frac{75}{16} & \frac{885}{128} & \frac{125}{32} & \frac{225}{64} \\ \frac{25}{4} & \frac{75}{16} & \frac{75}{16} & \frac{125}{32} & \frac{225}{64} & \frac{125}{32} \\ \frac{75}{8} & \frac{75}{16} & \frac{125}{16} & \frac{225}{64} & \frac{125}{32} & \frac{885}{128} \end{pmatrix}$$

This matrix comes in two flavors. The number of data points is either in the upper left-hand corner as shown here or in the lower right-hand corner. Notice that in true vector form, we never measured *n*; it comes naturally from the vector formulation.

The B vector shows the effect of the bottom terms being multiplied by powers of *x* and *y*. These terms are smaller. This is the ideal case. One should always endeavor to reduce the magnitude of the problem to improve precision.

```
(* same height as A *)
B // MatrixForm
```

$$\begin{pmatrix} 150 \\ \frac{1525}{16} \\ \frac{1525}{16} \\ \frac{9905}{128} \\ \frac{3775}{64} \\ \frac{9905}{128} \end{pmatrix}$$

We need to invert A to solve the linear system.

```
(* compute the inverse of the A matrix *)
Ainv = Inverse[A];
Ainv // MatrixForm
```

$$\begin{pmatrix} \frac{83}{175} & -\frac{164}{175} & -\frac{164}{175} & \frac{16}{35} & \frac{16}{25} & \frac{16}{35} \\ -\frac{164}{175} & \frac{808}{175} & \frac{16}{25} & -\frac{128}{35} & -\frac{32}{25} & 0 \\ -\frac{164}{175} & \frac{16}{25} & \frac{808}{175} & 0 & -\frac{32}{25} & -\frac{128}{35} \\ \frac{16}{35} & -\frac{128}{35} & 0 & \frac{128}{35} & 0 & 0 \\ \frac{16}{25} & -\frac{32}{25} & -\frac{32}{25} & 0 & \frac{64}{25} & 0 \\ \frac{16}{35} & 0 & -\frac{128}{35} & 0 & 0 & \frac{128}{35} \end{pmatrix}$$

One multiplication solves the problem.

```
(* we recover the amplitude vector *)
a = Ainv.B

{1, 2, 2, 3, 3, 3}
```

This is an exact answer, something extremely useful to those who carefully validate their codes.

Let's look at a numeric version of this computation. The only change we have to make is to replace this line

```
addr = Table[{i, j}, {i, 0, 1, 1/4}, {j, 0, 1, 1/4}]
```

with this line:

```
addr = Table[{i, j}, {i, 0, 1, 0.25}, {j, 0, 1, 0.25}]
```

Mathematica will see the decimal points and convert everything to numerical values. The final answer is:

```
(* we recover the amplitude vector *)
a = Ainv.B

{1., 2., 2., 3., 3., 3.}
```

We can't tell much from this form. We only know that the first six digits are the same. This form is more revealing.

```
(* difference between input amplitudes an numeric amplitudes *)
a - a

{2.22045×10^-15, -1.64313×10^-14, -1.64313×10^-14, 0., 0., 0.}
```

If you only had these results, how would you go about proving that they are machine noise and not caused by some subtle error in the algorithm? Typically, the worst part of a calculation isn't getting the right answer; it's verifying that the answer is right. By using *Mathematica* intelligently we can alleviate a good part of the burden.

Precomputation

An important point often overlooked is the ability to precompute part of the linear system. If the measurements are always made at the same points, then the matrix A will not change. A simply encodes information about the measurement geometry. And if the measurement geometry doesn't change, then neither does the A. This allows us to compute the inverse matrix for each geometry *a priori*.

So, if we are doing a series of measurements at the same points, we compute A^{-1} once and store it. For each new measurement you build B and compute $A^{-1} \cdot B$.

2.3.8 Higher dimensions

The next step is to extend this method to spaces of higher dimension *D*. There is one small detail which we must attend: the dot product turns into a vector multiplication. Also, experience shows that many people are a little confused by how to formulate the *p* vector in higher dimension. Finally, we will speed the process up a bit by using recursion.

We shall address the generalizations as we encounter them. First, we will pose a 3-space polynomial for study.

$$f(x,\ y,\ z) = x + 2y + 3z + 4x^2 + 5xy + 6y^2 + 7yz + 8z^2 \tag{2.44}$$

This has the amplitude vector:

```
(* amplitude vector for the polynomial *)
a = {0, 1, 2, 3, 4, 5, 0, 6, 7, 8};
```

We are still looking at a quadratic form, just in a space of higher dimension.

```
(* define order *)
d = 2;
```

How does one generalize the vector of exponents *p*? Quite easily.

```
(* define the vector of exponents *)
p = Flatten[Table[{i - j, j - k, k}, {i, 0, d}, {j, 0, i}, {k, 0, j}], 2]
(* measure the dimension of the components *)
t = Length[p];

{{0, 0, 0}, {1, 0, 0}, {0, 1, 0}, {0, 0, 1}, {2, 0, 0},
 {1, 1, 0}, {1, 0, 1}, {0, 2, 0}, {0, 1, 1}, {0, 0, 2}}
```

For a space of *D* dimensions the components of the *p* vector will each have *D* terms as shown here for $D = 3$.

We need to build vectors of the components. Posit a list data which is a list of measurements in the form $\{x, y, z, f(x, y, z)\}$. We can extract component lists as before using:

$$\begin{bmatrix} x \\ y \\ z \\ f \end{bmatrix} = data \cdot I \tag{2.45}$$

where I is the identity matrix of appropriate dimension. This has the *Mathematica* manifestation of:

```
(* isolate component vectors *)
x = data.{1, 0, 0, 0};
y = data.{0, 1, 0, 0};
z = data.{0, 0, 1, 0};
f = data.{0, 0, 0, 1};
```

Here we would like to improve on the do-loops which built the X and Y vectors. We wrote two different loops for a single process applied to two different data sets. This is poor programming practice. We should encode the single procedure in such a form that it can be applied to variables easily. In other words, we don't want to hard-code or process.

We turn once again to a pure function.

```
(* takes the last list in #1 and multiplies it by #2 *)
(* we will apply this function to arguments like [X,x] *)
build = (Last[#1] #2) &;
```

Here is how the command is used.

```
(* build the ones vector *)
X = {x - x + 1};
(* build the rest of the terms *)
Do[
 AppendTo[X, build[X, x]];
 , {i, 2 d}]
```

This is correct, but we still have to manually apply our data to it. This may not be a burden at $D = 3$, but if you are a chemist or a physicist working in dozens or hundreds or thousands of dimensions, this approach is out of the question.

We won't handle the arbitrarily large spaces because we think a specific example is more helpful to the novitiate. Hopefully, advanced users will be able to modify this code as needed.

First, we need to specify our space.

```
(* define order *)
d = 2;
(* define dimension *)
𝒟 = 3;
(* accumulators to feed to the LSF *)
big = Table[1, {𝒟}];
(* measurement positions *)
little = {x, y, z};
```

Now we have a module to do the buildup for us.

```
(* build the X, Y, and Z lists *)
(* i specifies the dimension, here x, y, or z *)
(* d specifies the degree *)
assemble[i_Integer, d_Integer] := Module[{X, x},
   x = little[[i]];
   (* build the list of ones *)
   X = {x - x + 1};
   (* build up the rest of the list *)
   Do[
    (* multiply last member by x to increase power *)
    AppendTo[X, build[X, x]];
    , {j, 2 d}];
   (* store in the proper place *)
   big = ReplacePart[big, X, i];
  ];
```

This structure can now be used to sweep through the entire space, one dimension at a time.

```
(* sweep through the dimensions *)
Do[
 (* assemble accumulator for dimension i *)
 assemble[i, d];
 , {i, 𝒟}]
```

Now we need to again generalize the method we used in the last section. As you can see, our computation was accomplished by a dot product.

```
A = Table[k = p[[i]] + p[[j]]; X[[k[[1]]+1]] . Y[[k[[2]]+1]], {i, t}, {j, t}];
```

Yet now we have three terms X, Y, and Z. There is no dot product for three vectors, the analog of the vector triple product. The scalar triple product involves a dot product and a cross product. Clearly, this part needs to be fixed.



Dot Product, Vector Triple Product, Scalar Triple Product

What we asked the dot product to do was multiply the first components, then multiply the second components, etc. Then we wanted these products summed. We need to same action, except for an arbitrary number of components. Once again, *Mathematica* has the perfect tool. We simply need to multiply the lists and then sum the resultant lists. It is a difficult process to describe with words, but the mathematical formulation is unambiguous. We are going to replace the form that only works in two dimensions:

```
X[[k[[1]]+1]] . Y[[k[[2]]+1]]
```

with:

```
Plus @@ (X[[i+1]] Y[[j+1]] Z[[k+1]])
```

a form that scales to arbitrary dimension. We are not done yet.

The observant reader may have looked at the formulation in the previous section and noted somewhat of an inefficiency. Here is the offending line.

```
A = Table[k = p[[i]] + p[[j]]; X[[k[[1]]+1]] . Y[[k[[2]]+1]], {i, t}, {j, t}];
```

The problem is that while we are using *Mathematica*'s highly optimized vector tool, the dot product, we are still recomputing redundant terms. For example, the matrix is

symmetric, yet we just go ahead and recompute A_{ji} even though we have the equivalent A_{ij} in hand.

Even worse are some cases like $\sum x_i^2 y_i$ which appear four times, and each time it is computed anew. Certainly, the effect is imperceptible for the data set that we presented. But if we have large numbers of dimensions and large numbers of measurements, it behooves us to use *Mathematica*'s tools that help us prevent such waste.

This is a case that clearly calls for recursion. When we formulate the problem recursively, *Mathematica* remembers all previous results. So if a new result is called for, *Mathematica* checks for the value and recomputes it only if needed. We will build a recursive function called `elem` to build the elements of the A matrix. To make the function recursive, we will just combine the `SetDelayed` (`:=`) operator and the `Set` (`=`) operator like so.

```
(* a recursive function to build the elements of the A matrix *)
elem[i_Integer, j_Integer, k_Integer] :=
 elem[i, j, k] = Plus @@ (X[[i+1]] Y[[j+1]] Z[[k+1]])
```

Let's dissect this syntax. This is a basic function defined with the `SetDelayed` operator.

```
(* a typical function *)
elem[i_Integer, j_Integer, k_Integer] := Plus @@ (X[[i+1]] Y[[j+1]] Z[[k+1]])
```

This is a set operation which we are all intimately familiar with. The value of the computation on the right-hand side of the equality is assigned to the variable on the left-hand side. Again, the wording is awkward and the formulation is simple.

```
(* a typical statement *)
elem[i, j, k] = Plus @@ (X[[i+1]] Y[[j+1]] Z[[k+1]])
```

When we combine these two statements, we form a function that stores all its values. So if we call `elem[1,2,3]`. *Mathematica* will first check to see whether `elem[1,2,3]` is a stored value. If not, it will use the functional definition to compute it.

If this seems a bit nebulous, do not worry. Recursion is an important topic and we will be reinforcing this lesson.



Recursion

We think looking at the before and after values for this function should make the process clear. We need to move the results from the `big` list to the *X*, *Y*, and *Z* lists. This is simply to make the matrix element computation more concrete. Then we will define the recursive function `elem`.

```
(* move the items from the list big *)
{X, Y, Z} = big;
(* define a recursive function to avoid redudant computation *)
elem[i_Integer, j_Integer, k_Integer] :=
  elem[i, j, k] = Plus @@ (X[[i+1]] Y[[j+1]] Z[[k+1]]);
```

At this point the function maintains a simple definition.

```
?elem
```

```
Global`elem
```

```
elem[i_Integer, j_Integer, k_Integer] :=
 elem[i, j, k] = Plus @@ (X[[i + 1]] Y[[j + 1]] Z[[k + 1]])
```

Then we proceed to build the *A* matrix.

```
(* build the A matrix *)
A = Table[n = p[[i]] + p[[j]]; elem[n[[1]], n[[2]], n[[3]]], {i, t}, {j, t}];
```

Now `elem` has the value shown here. The stored values are evident and they are first.

```
?elem
```

```
Global`elem
```

$$\text{elem}[0, 0, 4] = \frac{159347853}{80000}$$

$$\text{elem}[0, 3, 1] = \frac{194481}{160}$$

$$\text{elem}[3, 0, 1] = \frac{194481}{160}$$

$$\text{elem}[2, 1, 1] = \frac{126567}{160}$$

$$\text{elem}[2, 0, 1] = \frac{126567}{80}$$

$$\text{elem}[1, 1, 1] = \frac{9261}{8}$$

$$\text{elem}[0, 2, 1] = \frac{126567}{80}$$

$$\text{elem}[1, 0, 1] = \frac{9261}{4}$$

$$\text{elem}[1, 0, 3] = \frac{194481}{160}$$

$$\text{elem}[0, 1, 1] = \frac{9261}{4}$$

$$\text{elem}[0, 1, 3] = \frac{194481}{160}$$

$$\text{elem}[4, 0, 0] = \frac{159347853}{80000}$$

$$\text{elem}[0, 0, 1] = \frac{9261}{2}$$

$$\text{elem}[3, 1, 0] = \frac{194481}{160}$$

$$\text{elem}[0, 0, 3] = \frac{194481}{80}$$

$$\text{elem}[2, 2, 0] = \frac{1729749}{1600}$$

$$\text{elem}[3, 0, 0] = \frac{194481}{80}$$

$$\text{elem}[1, 3, 0] = \frac{194481}{160}$$

$$\text{elem}[2, 1, 0] = \frac{126567}{80}$$

$$\text{elem}[0, 4, 0] = \frac{159347853}{80000}$$

$$\text{elem}[1, 2, 0] = \frac{126567}{80}$$

```
elem[2, 0, 0] = 126567/40

elem[2, 0, 2] = 1729749/1600

elem[0, 3, 0] = 194481/80

elem[1, 1, 0] = 9261/4

elem[1, 1, 2] = 126567/160

elem[0, 2, 0] = 126567/40

elem[1, 0, 0] = 9261/2

elem[0, 2, 2] = 1729749/1600

elem[1, 0, 2] = 126567/80

elem[0, 1, 0] = 9261/2

elem[0, 1, 2] = 126567/80

elem[0, 0, 0] = 9261

elem[0, 0, 2] = 126567/40

elem[1, 2, 1] = 126567/160

elem[i_Integer , j_Integer , k_Integer ] :=
 elem[i, j, k] = Plus @@ (X[[i + 1]] Y[[j + 1]] Z[[k + 1]])
```

The last detail is the *B* vector. Here no change is required; we already have used the list multiplication tools. Here is the code for the previous case:

```
(* build the B vector *)
B = Table[k = p[[i]]; Plus @@ (F X[[k[[1]] + 1]] Y[[k[[2]] + 1]]), {i, t}];
```

and for the current case.

```
(* build the B vector *)
B = Table[k = p[[i]]; Plus @@ (f X[[k[[1]]+1]] Y[[k[[2]]+1]] Z[[k[[3]]+1]]), {i, t}];
```

Again we want to show you the joy of exact results. We ran this computation in double precision by swapping the line:

```
(* sample size *)
δ = 1/20;
```

with:

```
(* sample size *)
δ = 0.05;
```

For these 9,261 points we measured the amplitudes:

```
(* solution vector *)
a = Ainv.B

{8.53281×10^-12, 1., 2., 3., 4., 5., 3.37952×10^-12, 6., 7., 8.}
```

and the full errors are:

```
(* difference between input amplitudes an numeric amplitudes *)
a - a

{-8.53281×10^-12, 2.07983×10^-11, 1.62292×10^-11,
 2.37144×10^-12, -1.36744×10^-11, -8.78408×10^-12,
 -3.37952×10^-12, -8.89244×10^-12, -4.14957×10^-12, 1.51257×10^-12}
```

This unequivocally proves that these numbers are the result of machine noise in a grueling task which *Mathematica* has obviated with exact computation.

In closing we want to again remind the reader that the A matrix describes the measurement geometry. This allows one to precompute A and hence A^{-1} for every geometry used.

Error propagation
As we have stated emphatically, computation of the errors associated with the amplitudes is absolutely essential. However, this is a book to help one learn about *Mathematica* and so we are forced to exclude these intricate calculations. Fortunately for us, we have been able to teach *Mathematica* how to compute them for us. If you need to see how to compute these terms, you can e-mail the author.[1]

2.4 An inverse problem

This is a very interesting case taken from a laboratory problem. We will see how *Mathematica* can put incredible power at your fingertips and on rare occasions, taunt us. This was a problem originally solved in version 4.2. But when switched over to version 5.0, the notebook no longer calculated as desired. We want to address this and also use the example to demonstrate real world problem solving. This is a good teaching exercise because this is a real problem of significance and it exemplifies the types of issues you are likely to encounter.

2.4.1 Measuring curvature

Consider an ideal point source of light with a wavefront emitted at a time t_0. The source will produce a spherical wavefront with a radius r increasing at a constant rate in the same way ripples in a pond spread after you drop in a stone. A very important and fundamental measurement involves determining the radius of the wavefront from the measurement of relative height variation over a small patch.

We will consider one measurement type called a *modal fit*. This method relies on describing the wavefront using a series of basis functions. We will consider the basis set of minimal complexity, the Taylor monomials:

$$, \; x, \; y, \; x^2, \; xy, \; y^2, \; . \tag{2.46}$$

The computational problem is immediately apparent. We are attempting to use rational functions to describe a surface with irrational form.

$$z = \sqrt{r^2 - x^2 - y^2} \tag{2.47}$$

For centuries a simple formula has been used. One resolves the measured patch into a quadratic function. Without loss of generality, we can discuss this problem in one dimension since the sphere is rotationally invariant. If we call A the amplitude of the quadratic term, the curvature can be expressed as

$$\kappa = 4A \, . \tag{2.48}$$

---

1 dantopa@gmail.com

Certainly paraboloids do not resemble spheres. So how well does equation 2.48 do in describing the relationship between the measured paraboloid and the ideal sphere?

Let's step back for a moment and talk about the issue of curvature. How does one compute the curvature κ at the point *p* of some curve? We call up the formula we learned in first semester calculus.

$$\kappa = \frac{y''}{(1+y'^2)^{3/2}} \tag{2.49}$$



Curvature

We can apply this formula to the curve of the upper half of a circle.

$$y(x) = \sqrt{r^2 - x^2} \tag{2.50}$$

We will see that *Mathematica* can at times exhibit vexatious behavior. We start by creating a curvature operator.

```
(* define a curvature operator *)
κ[f_] = f''[x] / (1 + f'[x]^2)^(3/2);
```

Then we define a function using equation 2.50.

```
(* define a curve *)
f[x_] = √(r^2 - x^2);
```

Notice that we are not computing the curvature at a point. We are going to return a function that will describe the curvature at all points.

```
(* compute the curvature *)
sphere = κ[f]
```

$$\frac{-\frac{x^2}{(r^2-x^2)^{3/2}} - \frac{1}{\sqrt{r^2-x^2}}}{\left(1 + \frac{x^2}{r^2-x^2}\right)^{3/2}}$$

Some simplification is in order.

```
(* basic simplification *)
Simplify[sphere]
```

$$-\frac{1}{\sqrt{\frac{r^2}{r^2-x^2}}\ \sqrt{r^2-x^2}}$$

Clearly, we need to try `FullSimplify`.

```
(* more simplification *)
FullSimplify[sphere]
```

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

We are stymied that *Mathematica* has factored $r^2 - x^2$ in one case, but not the other. Let's backtrack and consider simplification with a simple example.

```
(* test case *)
Simplify[Sqrt[x^2]]
```

$$\sqrt{x^2}$$

This may not be what you expected, but it is correct. After all, x may be complex. Specifying that x is real helps.

```
(* specify x is real *)
Simplify[Sqrt[x^2], x ∈ Reals]
```

```
Abs[x]
```

This still may not be what you expected, but it is correct because x may be negative. When we limit x to be non-negative we get:

```
(* specify x is real and greater than zero *)
Simplify[√x^2 , x ∈ Reals ∧ x ≥ 0]

x
```

There is another tool we can use called `Refine`. In this case, it functions in the same manner as `Simplify`.

```
(* Refine test case *)
Refine[√x^2 ]

√x^2
```

Here it has similar limitations.

```
(* specify x is real and greater than zero *)
Refine[√x^2 , x ∈ Reals ∧ x ≥ 0]

x
```

So we go back to the original problem and hope that by restricting *r* and *x* we can simplify the result.

```
(* try restricting the arguments *)
FullSimplify[sphere, r ∈ Reals ∧ x ∈ Reals ∧ r > 0 ∧ x > 0]

-(√((r - x) (r + x)) √(1/(r^2 - x^2)))/r
```

Such is not the case. We will then try `Refine`.

```
(* maybe Refine will work *)
Refine[sphere, r ∈ Reals ∧ x ∈ Reals ∧ r > 0 ∧ x > 0]
```

$$\frac{-\frac{x^2}{(r^2-x^2)^{3/2}}-\frac{1}{\sqrt{r^2-x^2}}}{\left(1+\frac{x^2}{r^2-x^2}\right)^{3/2}}$$

After considerable effort we finally managed to sidestep the problem with this equivalent definition for the curvature.

```
(* allows for simplification *)
κ[f_] = Sqrt[f''[x]^2 / (1 + f'[x]^2)^3];
```

You'll notice that we squared the numerator and brought it under the radical. The curvature function takes the initial form:

```
(* curvature function *)
κ[f]
```

$$\sqrt{\frac{\left(-\frac{x^2}{(r^2-x^2)^{3/2}}-\frac{1}{\sqrt{r^2-x^2}}\right)^2}{\left(1+\frac{x^2}{r^2-x^2}\right)^3}}$$

This allows for simplification.

```
(* basic simplification works *)
Simplify[κ[f], r ∈ Reals ∧ r > 0]
```

$$\frac{1}{r}$$

This leads us to the spectacular result that a sphere is a surface of constant curvature. Also, we see that the curvature is the reciprocal of the radius. This convention

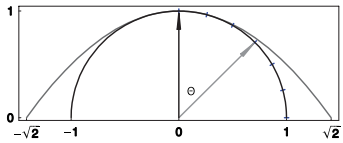
allows us to say that planes have no curvature. Pressing forth, how can we use a quadratic function to describe the curvature.

Remembering back to freshman calculus when we learned about curvature; we were told about the osculating circle. When we computed the curvature at some point *p*, the osculating circle touched the curve at that one point. The radius of the osculating circle was the reciprocal of the computed curvature.



Osculating Circle

Instead of looking at the apex of the parabola and asking about the radius of the osculating circle there, we should think of the wavefront as an osculating circle and ask what value *A* for the amplitude of the quadratic monomial is needed to match the parabola to the circle. Consider that now the circle is fixed and we are varying the parabola. Figure 2.1 helps visualize the problem and define key parameters.



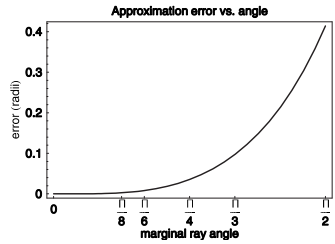


Figure 2.1: The geometry of the circle and its parabolic approximation. The figure on the left shows a slice of the hemispherical wave (black) and the parabola (gray) which matches the curvature at the apex. The black ray extending from the origin is the chief ray; the gray ray represents the marginal ray. These two rays define the marginal ray angle θ which describes how much of the circle we are looking at. The figure on the right shows the difference between the circle and the parabola measured along the normal rays. The blue tick marks on the left identify specific marginal ray angles which correspond to the blue dots on the right.

What we find is that the formula 2.48 is exact only at the apex of the parabola. When you look at the curve on the left, you may at first consider the match to be quite good. But in optical experiments one can measure deviations of the order of fractions of the wavelength of the light used in the measurement. The figure on the right shows what the error is between the ideal surface and the measured surface.

Again, we must pay attention to units. As an example, consider a sphere with radius 9 mm being examined over an angle of π⁄ 16 (roughly 11º). At the edge of the measurement, the discrepancy is 0.0016 mm. For a 635 nm laser this corresponds to over 2.5 waves, an enormous difference.

How can we use a modal expansion to measure a sphere over some appreciable angle θ?

The same second order expansion that was used to find the amplitude *A* in equation can be employed in a different fashion to yield an exact result. We will use two amplitudes this time: the amplitude for the constant term, a_{00} and the amplitude for the *xy* cross term a_{11}.

Examine figure 2.2 below which shows the ratio of the amplitudes for the constant terms (a_{00}) and the $x\,y$ term (a_{11}). The dependent variable is a dimensionless quantity r defined as:

$$r = \sin\theta\,. \tag{2.51}$$

By looking at figure 2.2 we see that r varies from zero at the apex to unity for a full 90° sample.

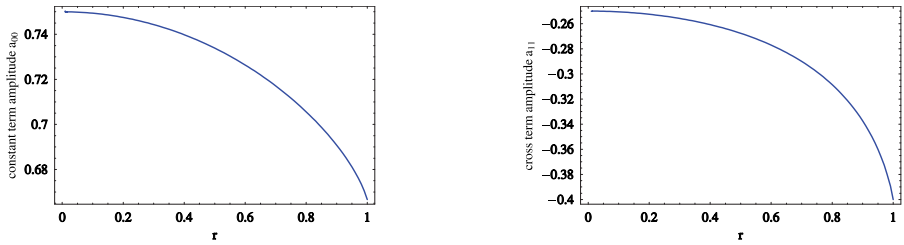


Figure 2.2: An absolute measurement of the radius for the sphere requires knowledge from two different amplitudes. The two lowest nonzero amplitudes are a_{00} and a_{11}. The variation of these terms with respect to r is shown here.

These quantities take the functional forms:

$$a_{00} = \frac{2(20r^4 - 27r^2 + 12 + (1-r^2)^{3/2}(-2r^4 + 9r^2 - 12))}{15r^6} \tag{2.52}$$

$$a_{11} = \frac{2(-5r^2 + 4 + \sqrt{1-r^2}(r^4 + 3r^2 - 4))}{5r^6} \tag{2.53}$$

As alluded to earlier, the ratio of these two amplitudes can tell us exactly what the curvature is over a large r.

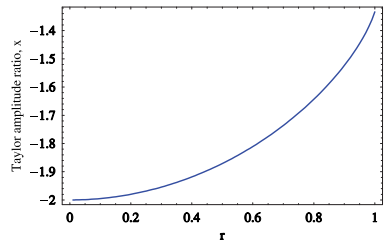


Figure 2.3: The ratio of the constant amplitude over the cross amplitude can tell us what the value of r is.

The problem now is that we have the ratio ξ as a function of r, $\xi(r)$. We need to invert this to find r as a function of ξ since ξ is what we measured. For this task, we turn to *Mathematica*. We start with computing the ratio. We found that while v. 4.2 could fully simplify this equation, version 5.0 could not. This can be particularly problematic.

```
(* from version 4.2 *)
ξ42 = FullSimplify[ (8 (-3 + 5 r^2) + 4 Sqrt[1 - r^2] (6 - 7 r^2 + r^4)) / (15 r^4) /
    (4 (4 - 5 r^2 + Sqrt[1 - r^2] (-4 + 3 r^2 + r^4)) / (5 r^6)) ]

(15 - 15 r^2 - 5 r^4 + r^6 + 15 (1 - r^2)^(3/2)) / (3 (-5 + 5 r^2 + r^4))
```

Considerable effort was made to coax the proper answer from version 5.0, but to no avail.

```
(* from version 5.0 *)
ξ50 = FullSimplify[ (8 (-3 + 5 r^2) + 4 Sqrt[1 - r^2] (6 - 7 r^2 + r^4)) / (15 r^4) /
    (4 (4 - 5 r^2 + Sqrt[1 - r^2] (-4 + 3 r^2 + r^4)) / (5 r^6)) ]

(r^2 (8 (-3 + 5 r^2) + 4 Sqrt[1 - r^2] (6 - 7 r^2 + r^4))) / (12 (4 - 5 r^2 + Sqrt[1 - r^2] (-4 + 3 r^2 + r^4)))
```

At least it is trivial to show that the two answers are equivalent.

```
(* check that the 4.2 answer is the same as the 5.0 answer *)
diff = ξ42 - ξ50

0
```

The form of the amplitude ratio that we will use is the version 4.2 form:

$$\xi(r) = \frac{r^6 - 5r^4 - 15r^2 + 15 + 15(1-r^2)^{3/2}}{3(r^4 + 5r^2 - 5)}. \tag{2.54}$$

```
(* from a LSF to describe the hemisphere *)
ξ = (15 - 15 r^2 - 5 r^4 + r^6 + 15 (1 - r^2)^(3/2)) / (3 (-5 + 5 r^2 + r^4));
```

Before we perform the analytic inversion, let's exploit the `ImplicitPlot` command to look at the inverse function.

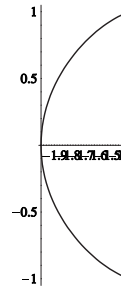
```
(* load the implicit plot package *)
<< Graphics`ImplicitPlot`
(* plot r(Ξ) *)
g1 =
  ImplicitPlot[Ξ == (15 - 15 r^2 - 5 r^4 + r^6 + 15 (1 - r^2)^(3/2)) / (3 (-5 + 5 r^2 + r^4)), {Ξ, -2, -4/3}];
```

```
Solve::verif : Potential solution {r → -<<69>>} (possibly discarded by verifier ) should be checked by hand. May require use of limits. More...

Solve::verif : Potential solution {r → -<<69>> i} (possibly discarded by verifier ) should be checked by hand. May require use of limits. More...

Solve::verif : Potential solution {r → <<69>> i} (possibly discarded by verifier ) should be checked by hand. May require use of limits. More...

General::stop : Further output of Solve::verif will be suppressed during this calculation. More...
```



This method does not work and now we rely exclusively on the analytic inverse.

Mathematica inverts this function in an instant.

```
(* invert the equation ξ[r] and find r[ξ] *)
soln = Solve[f == ξ, r]; // Timing
(* count the number of solutions *)
ns = Length[soln]
```

```
{0.181 Second, Null}
```

```
8
```

We see there are eight solutions for us to evaluate, each quite lengthy. We'll show the first solution here; the other seven solutions are comparable in length.

```
(* examine the first solution in the series *)
First[soln]
```

$$\left\{r \to -\sqrt{\left(\frac{15}{4}+\frac{3\,f}{2}-\frac{1}{2}\sqrt{\left(25+10\,f+3\,f^2+\frac{9}{4}\,(5+2\,f)^2-3\,(25+10\,f+3\,f^2)+\left(3\,2^{1/3}\,(400+740\,f+595\,f^2+150\,f^3+9\,f^4)\right)\Big/\left(442800+1279800\,f+1601775\,f^2+962820\,f^3+296460\,f^4+36450\,f^5+1458\,f^6+\sqrt{(9447840000+97627680000\,f+307448460000\,f^2+479595978000\,f^3+414147542625\,f^4+193975965000\,f^5+42249559500\,f^6+2391484500\,f^7)}\right)^{1/3}+\frac{1}{3\,2^{1/3}}\left(\left(442800+1279800\,f+1601775\,f^2+962820\,f^3+296460\,f^4+36450\,f^5+1458\,f^6+\sqrt{(9447840000+97627680000\,f+307448460000\,f^2+479595978000\,f^3+414147542625\,f^4+193975965000\,f^5+42249559500\,f^6+2391484500\,f^7)}\right)^{1/3}\right)\right)}-\frac{1}{2}\sqrt{\left(-25-10\,f-3\,f^2+\frac{9}{2}\,(5+2\,f)^2-3\,(25+10\,f+3\,f^2)-\left(3\,2^{1/3}\,(400+740\,f+595\,f^2+150\,f^3+9\,f^4)\right)\Big/\left(442800+1279800\,f+1601775\,f^2+962820\,f^3+296460\,f^4+36450\,f^5+1458\,f^6+\sqrt{(9447840000+97627680000\,f+307448460000\,f^2+479595978000\,f^3+414147542625\,f^4+193975965000\,f^5+42249559500\,f^6+2391484500\,f^7)}\right)^{1/3}-\frac{1}{3\,2^{1/3}}\left(\left(442800+1279800\,f+1601775\,f^2+962820\,f^3+296460\,f^4+36450\,f^5+1458\,f^6+\sqrt{(9447840000+97627680000\,f+307448460000\,f^2+479595978000\,f^3+414147542625\,f^4+193975965000\,f^5+42249559500\,f^6+2391484500\,f^7)}\right)^{1/3}\right)-\left(27\,(5+2\,f)^3-360\,(-1+2\,f+f^2)-36\,(5+2\,f)\,(25+10\,f+3\,f^2)\right)\Big/\left(4\sqrt{\left(25+10\,f+3\,f^2+\frac{9}{4}\,(5+2\,f)^2-3\,(25+10\,f+3\,f^2)+\left(3\,2^{1/3}\,(400+740\,f+595\,f^2+150\,f^3+9\,f^4)\right)\Big/\left(442800+1279800\,f+1601775\,f^2+962820\,f^3+296460\,f^4+36450\,f^5+1458\,f^6+\sqrt{(9447840000+97627680000\,f+307448460000\,f^2+479595978000\,f^3+414147542625\,f^4+193975965000\,f^5+42249559500\,f^6+2391484500\,f^7)}\right)^{1/3}+\frac{1}{3\,2^{1/3}}\left(\left(442800+1279800\,f+1601775\,f^2+962820\,f^3+296460\,f^4+36450\,f^5+1458\,f^6+\sqrt{(9447840000+97627680000\,f+307448460000\,f^2+479595978000\,f^3+414147542625\,f^4+193975965000\,f^5+42249559500\,f^6+2391484500\,f^7)}\right)^{1/3}\right)\right)}\right)\right)\right)\right)\right)\right\}$$

As you can see, we are overwhelmed by the voluminous output. How are we to decide which of these eight solutions is the one that we need? We will develop the evaluation process using just the first element from the solution. First, you may want to simplify the expressions since we know the range ξ is a real number We will see that simplification at this point does not seem to help. But first examine the simplification.

```
(* simplify this solution *)
ssoln = Simplify[First[soln], f ∈ Reals]
```

$$\left\{r \to -\frac{1}{2}\right.$$

$$\sqrt{\left(15 + 6\,f - \sqrt{\left(100 + 40\,f + 12\,f^2 + 9\,(5 + 2\,f)^2 - 12\,(25 + 10\,f + 3\,f^2) + \left(4\,2^{1/3}\right.\right.\right.}$$

$$\left.\left.\left.(400 + 740\,f + 595\,f^2 + 150\,f^3 + 9\,f^4)\right) \Big/ \left(16400 + 47400\,f + 59325\right.\right.\right.$$

$$f^2 + 35660\,f^3 + 10980\,f^4 + 1350\,f^5 + 54\,f^6 + 15\sqrt{15}$$

$$\left.\left.\left.\sqrt{(4 + 3\,f)^3\,(60 + 485\,f + 760\,f^2 + 492\,f^3 + 36\,f^4)}\right)^{1/3} + 2\,2^{2/3}\right.\right.$$

$$\left.\left(16400 + 47400\,f + 59325\,f^2 + 35660\,f^3 + 10980\,f^4 + 1350\,f^5 + 54\,f^6 + \right.\right.$$

$$\left.\left.15\sqrt{15}\,\sqrt{(4 + 3\,f)^3\,(60 + 485\,f + 760\,f^2 + 492\,f^3 + 36\,f^4)}\right)^{1/3}\right) -$$

$$\sqrt{2}\,\sqrt{\left(-50 - 20\,f - 6\,f^2 + 9\,(5 + 2\,f)^2 - 6\,(25 + 10\,f + 3\,f^2) - \right.}$$

$$\left.\left(2\,2^{1/3}\,(400 + 740\,f + 595\,f^2 + 150\,f^3 + 9\,f^4)\right) \Big/ \right.$$

$$\left.\left(16400 + 47400\,f + 59325\,f^2 + 35660\,f^3 + 10980\,f^4 + 1350\,f^5 + 54\,f^6 + \right.\right.$$

$$\left.\left.15\sqrt{15}\,\sqrt{(4 + 3\,f)^3\,(60 + 485\,f + 760\,f^2 + 492\,f^3 + 36\,f^4)}\right)^{1/3} -\right.$$

$$2^{2/3}\left(16400 + 47400\,f + 59325\,f^2 + 35660\,f^3 + 10980\right.$$

$$\left.f^4 + 1350\,f^5 + 54\,f^6 + 15\sqrt{15}\right.$$

$$\left.\left.\sqrt{(4 + 3\,f)^3\,(60 + 485\,f + 760\,f^2 + 492\,f^3 + 36\,f^4)}\right)^{1/3} +\right.$$

$$\left.(45\,(17 + 6\,f)) \Big/ \left(\sqrt{\left(100 + 40\,f + 12\,f^2 + 9\,(5 + 2\,f)^2 - 12\right.\right.\right.$$

$$\left.\left.\left.(25 + 10\,f + 3\,f^2) + \left(4\,2^{1/3}\,(400 + 740\,f + 595\,f^2 + \right.\right.\right.\right.$$

$$\left.\left.\left.\left.150\,f^3 + 9\,f^4)\right) \Big/ \left(16400 + 47400\,f + 59325\,f^2 + \right.\right.\right.\right.$$

$$\left.\left.\left.\left.35660\,f^3 + 10980\,f^4 + 1350\,f^5 + 54\,f^6 + 15\sqrt{15}\right.\right.\right.\right.$$

$$\left.\left.\left.\left.\sqrt{(4 + 3\,f)^3\,(60 + 485\,f + 760\,f^2 + 492\,f^3 + 36\,f^4)}\right)^{1/3} +\right.\right.\right.$$

$$\left.\left.\left.2\,2^{2/3}\left(16400 + 47400\,f + 59325\,f^2 + 35660\,f^3 + \right.\right.\right.\right.$$

$$\left.\left.\left.\left.10980\,f^4 + 1350\,f^5 + 54\,f^6 + 15\sqrt{15}\right.\right.\right.\right.$$

$$\left.\left.\left.\left.\sqrt{(4 + 3\,f)^3\,(60 + 485\,f + 760\,f^2 + 492\,f^3 + 36\,f^4)}\right)^{1/3}\right)\right)\right)$$

$$\left.\left.\right)\right\}$$

How will we use these substitution rules? The basic strategy is this. We will sample equation 2.54 uniformly as r varies from zero to unit and develop a list of points of the form $\{r, \xi(r)\}$. Then we will feed these points to the inverse function and look for agreement.

```
(* sample this curve to get points to check the inverse *)
pts = Table[{r, ξ}, {r, 0, 1, 0.1}]

{{0, -2}, {0.1, -1.99499}, {0.2, -1.9799}, {0.3, -1.95451},
 {0.4, -1.91837}, {0.5, -1.87082}, {0.6, -1.81069},
 {0.7, -1.73606}, {0.8, -1.64321}, {0.9, -1.52345}, {1., -1.33333}}
```

We need to extract the list or r values and ξ values separately. We will feed the x values into the inverse function and see if it returns the ξ values.

```
(* points to feed to the inverse solution *)
(* the solution that we need will return the r values above *)
x = pts.{1, 0};
y = pts.{0, 1}
ny = Length[y]

{-2., -1.99499, -1.9799, -1.95451, -1.91837, -1.87082,
 -1.81069, -1.73606, -1.64321, -1.52345, -1.33333}
```

```
11
```

Since we are dealing with such a long and complex solution, we really don't want to be tinkering with the entire list. So let's take one piece of it and develop our methodology. The first piece is as good as any at this point.

```
(* grab one of the solutions for testing *)
test = First[soln];
```

As usual, we will show the process first as a `Do` loop. But before showing the entire loop, we want to present the key steps. This will help you to visualize the machinations of the do-loop. As you recall, the variable `test` is a giant substitution rule of this form:

```
(* schematic representation of test *)
subrule = {r → function[f]}
```

We will give *f* a numerical value from the list of *y*'s.

```
(* schematic representation of test after substituting for f *)
subrule = {r → number}
```

To convert from a substitution rule to a number we simply apply the substitution rule.

```
(* apply the subrule to end up with the number *)
X = r /. subrule
```

This is the process. Let's watch this process in action. We pick one of the *y* values and substitute for *f*.

```
(* convert from a function of f to a numerical value *)
R = test /. f → y[[1]]

{r → 0. - 4.02331 × 10^-7 i}
```

Apply the substitution rule to extract the numerical value.

```
(* now apply the substitution rule to r to create a number *)
x = r /. R

0. - 4.02331 × 10^-7 i
```

These two steps immediately above are the engine in the do-loop that will be collecting the values for us.

```
(* container to hold the r values we compute *)
X = {};
(* loop through the sample points *)
Do[
  (* feed y values to the inverse *)
  (* to see if it generates the proper x *)
  R = test /. f → y[[i]];
  (* substitute to compute r values; add to list *)
  AppendTo[X, r /. R]
  , {i, ny}];
(* this should match the x list *)
X
```

```
{0. - 4.02331 × 10^-7 i, -0.1, -0.2, -0.3, -0.4,
 -0.5, -0.6, -0.7, -0.8, -0.9, -1. + 1.97686 × 10^-7 i}
```

Following established procedure, we show how to implement this process using a better tool: the `Table` command.

```
(* feed in the y values *)
R = Table[test /. f → y[[i]], {i, ny}]
```

```
{{r → 0. - 4.02331 × 10^-7 i}, {r → -0.1}, {r → -0.2},
 {r → -0.3}, {r → -0.4}, {r → -0.5}, {r → -0.6}, {r → -0.7},
 {r → -0.8}, {r → -0.9}, {r → -1. + 1.97686 × 10^-7 i}}
```

We will apply these rules together to get the X list.

```
(* make the substitution rules into values *)
X = r /. R
```

```
{0. - 4.02331 × 10^-7 i, -0.1, -0.2, -0.3, -0.4,
 -0.5, -0.6, -0.7, -0.8, -0.9, -1. + 1.97686 × 10^-7 i}
```

The advanced reader may be wondering why we used two separate commands. After all we could accomplish the process using:

```
(* ultracompact notation *)
X = r /. Table[test /. f → y[[i]], {i, ny}]

{0. - 4.02331 × 10^-7 i, -0.1, -0.2, -0.3, -0.4,
 -0.5, -0.6, -0.7, -0.8, -0.9, -1. + 1.97686 × 10^-7 i}
```

Certainly, one must agree that this form has limited pedagogic value. But we present it for the mid-level users who wish to improve their *Mathematica* skills.

Now we ask that you pay close attention. We are going to show why the arbitrary precision is such a blessing and why simplification can be such a chore.

Look at our output. First of all, we see complex numbers that are small. While they may be close to a single precision zero value, they appear to have substantial values in a double precision calculation. Also, when we examine the real values, we see significant errors. You should always view your results at full precision. This is because the screen display only shows six digits. In a double precision calculation you are able to hide all manner of errors past the sixth decimal.

⚠ Always check the full precision of your output. Remember, the screen display defaults to six characters.

```
(* look at these errors! *)
FortranForm[X]

        List((0.,-4.023313522338867e-7),-0.09999999999483113,
     -
-0.19999999999911372,-0.2999999999979378,-0.3999999999990377,
     -  -0.4999999999996558,-0.6000000000007402,-0.7000000000008683,
     -  -0.7999999999997527,-0.9000000000004127,
     -  (-1.0000000000000162,1.9768624248238478e-7))
```

Are these errors in the algorithm or are they errors in computation? We certainly cannot tell by inspection and hunches don't count. If we had a double precision tool we would be doomed to some serious evaluation to establish the fidelity of the algorithm. This process can most charitably be described as tedious and more accurately characterized as ghastly. But we have *Mathematica* and we can exploit infinite precision here.

Why did *Mathematica* give us numerical values anyway? The answer is because we feed numeric values into the computation. It all boils down to the increment on the iterator. We used a numeric value 0.1.

```
(* sample this curve to get points to check the inverse *)
pts = Table[{r, ξ}, {r, 0, 1, 0.1}]

{{0, -2}, {0.1, -1.99499}, {0.2, -1.9799}, {0.3, -1.95451},
 {0.4, -1.91837}, {0.5, -1.87082}, {0.6, -1.81069},
 {0.7, -1.73606}, {0.8, -1.64321}, {0.9, -1.52345}, {1., -1.33333}}
```

If we use a rational value, we get exact answers.

```
(* sample this curve to get points to check the inverse *)
pts = Table[{r, ξ}, {r, 0, 1, 1/10}]
```

$$\left\{\{0,-2\},\left\{\frac{1}{10},-\frac{10000\left(\frac{14849501}{1000000}+\frac{891\sqrt{11}}{200}\right)}{148497}\right\},\right.$$

$$\left\{\frac{1}{5},-\frac{625\left(\frac{224876}{15625}+\frac{144\sqrt{6}}{25}\right)}{8997}\right\},\left\{\frac{3}{10},-\frac{10000\left(\frac{13610229}{1000000}+\frac{273\sqrt{91}}{200}\right)}{136257}\right\},$$

$$\left\{\frac{2}{5},-\frac{625\left(\frac{194939}{15625}+\frac{63\sqrt{21}}{25}\right)}{7827}\right\},\left\{\frac{1}{2},-\frac{16}{177}\left(\frac{701}{64}+\frac{45\sqrt{3}}{8}\right)\right\},$$

$$\left\{\frac{3}{5},-\frac{4572}{2525}\right\},\left\{\frac{7}{10},-\frac{10000\left(\frac{6567149}{1000000}+\frac{153\sqrt{51}}{200}\right)}{69297}\right\},$$

$$\left.\left\{\frac{4}{5},-\frac{9736}{5925}\right\},\left\{\frac{9}{10},-\frac{10000\left(\frac{100941}{1000000}+\frac{57\sqrt{19}}{200}\right)}{8817}\right\},\left\{1,-\frac{4}{3}\right\}\right\}$$

We'll show this process in more detail with the solution. But first, we must find the solution.

```
(* start the CPU clock *)
t0 = TimeUsed[];
(* holds the results in a list *)
norms = {};
(* sweep through all 8 solutions *)
Do[
  (* this is a lengthy process so a little feedback is nice *)
  Print["testing root ", i];
  (* grab a solution and simplify it *)
  solni = Simplify[soln[[i]]];
  (* the X values will mirror
    the x values when we find the solution *)
  X = r /. Table[Simplify[solni /. f → y[[i]]], {i, ny}];
  (* compute the norm of the difference *)
  quality = Plus @@ Simplify[X - x]^2;
  (* store the norm *)
  AppendTo[norms, quality];
  (* announce the solution we are looking for *)
  If[quality == 0, Print["Solution = root ", i]];
  , {i, ns}];
(* measure the time *)
time = TimeUsed[] - t0;
```

As you can see from the statement marked with the arrowed line the loop will announce all roots that meet the solution criteria. That criteria being a zero norm for the difference vector between input and output. When we run this loop, however, we are surprised: no solution is found.

```
testing root 1
```

```
testing root 2
```

```
⚠ N::meprec : Internal precision limit
    $MaxExtraPrecision = 50.` reached while evaluating 1/4 (3 - Sqrt[<<1>>] -
      Sqrt[-254 - 488 <<1>>/(<<1>> <<1>>) - 4 <<1>> <<1>> + 450/Sqrt[-127 + <<1>>/<<1>> + 4 <<1>> <<1>>]] +
      <<8>> + <<1>>^2. More...
```

```
testing root 3
```

```
testing root 4
```

```
testing root 5
```

```
testing root 6
```

```
testing root 7
```

```
testing root 8
```

With curiosity piqued, we check results to see what values are there.

```
(* check the results *)
norms // N

{15.4, -2.77001 × 10^-14, 24.824, 3.42412, 42.5182 - 89.9 i,
 -2.53428 - 55.4531 i, 42.5182 + 89.9 i, -2.53428 + 55.4531 i}
```

We see large real numbers, large complex numbers, and the second entry which is quite small, but not zero. Then we notice the blue error message telling us that we exceeded the limit for extra precision. So while the second solution is the most likely candidate, we are very uncomfortable with the nonzero answer. A close inspection of the difference vector is warranted.

We can recycle the code from the `Do` loop quickly to capture the difference vector.

```
(* grab the second solution *)
i = 2;
solni = Simplify[soln[[i]]];
(* the X values will mirror
  the x values when we find the solution *)
X = r /. Table[Simplify[solni /. f → y[[i]], {i, ny}];
(* let's examine what should be a vector of zeros *)
diff = X - x;
```

The difference vector is too large to display here. We see that only one term — the final term — is actually zero. We begin our exploration with a peek at the first element.

```
(* look at the first element *)
First[X]
```

$$\frac{1}{2}\sqrt{\left(3-\sqrt{-127+\frac{488\,2^{2/3}}{\left(2389+225\sqrt{41}\right)^{1/3}}+4\left(4778+450\sqrt{41}\right)^{1/3}}-\right.}$$

$$\sqrt{\left(-254-\frac{488\,2^{2/3}}{\left(2389+225\sqrt{41}\right)^{1/3}}-4\left(4778+450\sqrt{41}\right)^{1/3}+\right.}$$

$$\left.\left.\frac{450}{\sqrt{-127+\frac{488\,2^{2/3}}{\left(2389+225\sqrt{41}\right)^{1/3}}+4\left(4778+450\sqrt{41}\right)^{1/3}}}\right)\right)$$

We suspect that this simplifies to zero. So we apply `Simplify` only to find:

```
(* try Simplify *)
Simplify[%]
```

$$\frac{1}{2}\sqrt{\left|3-\sqrt{-127+\frac{488\,2^{2/3}}{\left(2389+225\sqrt{41}\right)^{1/3}}+4\left(4778+450\sqrt{41}\right)^{1/3}}-\right.}$$

$$\sqrt{\left|-254-\frac{488\,2^{2/3}}{\left(2389+225\sqrt{41}\right)^{1/3}}-4\left(4778+450\sqrt{41}\right)^{1/3}+\right.}$$

$$\left.\left.\frac{450}{\sqrt{-127+\frac{488\,2^{2/3}}{\left(2389+225\sqrt{41}\right)^{1/3}}+4\left(4778+450\sqrt{41}\right)^{1/3}}}\right|\right|$$

that there is no change. We then try `FullSimplify` with success.

```
(* try FullSimplify *)
FullSimplify[%%]

0
```

Bolstered we move to the next term which proves resilient against both `Simplify` and `FullSimplify`.

```
(* let's look at the next element *)
X[[2]]
```

```
1/2 √(15 + (-14849501 - 4455000 √11)/2474950 - 1/2474950 (√((3
(-22388251749385790137716041903497414624275743496315549545 9
-
133922765856805223525427994546567249839170471567963760000
√11 - 55897232586290689940725500000000
√(247495 (508721731160524385776588082404608337271423 +
685329089078341085686902401210301392550000 √11)) +
34924207146378704054514015086872673362498 69
(34924207146378704054514015086872673362498 69 -
2746237856765404425055916999840737791300 00 √11 +
1091542270945500000 √(2722445
(508721731160524385776588082404608337271423 +
685329089078341085686902401210301392550000
√11))^(1/3) -
27462378567654044250559169998407377913000 0 √11
(34924207146378704054514015086872673362498 69 -
2746237856765404425055916999840737791300 00 √11 +
1091542270945500000 √(2722445
(508721731160524385776588082404608337271423 +
685329089078341085686902401210301392550000
√11))^(1/3) + 213344226579150143763744673 21
(34924207146378704054514015086872673362498 69 -
2746237856765404425055916999840737791300 00 √11 +
1091542270945500000 √(2722445
(508721731160524385776588082404608337271423 +
685329089078341085686902401210301392550000
√11))^(2/3) - 153826868570706344150478000 0 √11
(34924207146378704054514015086872673362498 69 -
2746237856765404425055916999840737791300 00 √11 +
1091542270945500000 √(2722445
(508721731160524385776588082404608337271423 +
685329089078341085686902401210301392550000
√11))^(2/3) + 1091542270945500000
√(2722445 (508721731160524385776588082404608337271423 +
685329089078341085686902401210301392550000
√11)) (-104373373040111 +
(34924207146378704054514015086872673362498 69 -
```

$$\left(34924207146378704054514015086872673362498 69 - 27462378567654044250559169998407377913000 0\sqrt{11} + 1091542270945500000 \sqrt{\left(2722445\left(5087217311605243857765880824046083372714 23 + 6853290890783410856869024012103013925500 0\sqrt{11}\right)\right)}^{1/3}\right)\Big/ \left(34924207146378704054514015086872673362498 69 - 27462378567654044250559169998407377913000 0\sqrt{11} + 1091542270945500000 \sqrt{\left(2722445\left(5087217311605243857765880824046083372714 23 + 6853290890783410856869024012103013925500 0\sqrt{11}\right)\right)}\right) - \frac{1}{2474950}\left(\sqrt{\left(\frac{1}{6}\left(-12923643035700006 + 529253946540000\sqrt{11} + 12\left(22274749 - 4455000\sqrt{11}\right)^2 + 98998000\left(14849501 + 4455000\sqrt{11}\right) - 2\left(14849501 + 4455000\sqrt{11}\right)^2 + \left(18\left(-21334422657915014376374467321 + 15382686857070634415047800 00\sqrt{11}\right)\right)\Big/ \left(34924207146378704054514015086872673362498 69 - 27462378567654044250559169998407377913000 0\sqrt{11} + 1091542270945500000 \sqrt{\left(2722445\left(5087217311605243857765880824046083372714 23 + 6853290890783410856869024012103013925500 0\sqrt{11}\right)\right)}^{1/3} - 18\left(34924207146378704054514015086872673362498 69 - 27462378567654044250559169998407377913000 0\sqrt{11} + 1091542270945500000 \sqrt{\left(2722445\left(5087217311605243857765880824046083372714 23 + 6853290890783410856869024012103013925500 0\sqrt{11}\right)\right)}^{1/3} - \left(9923111554050000\left(-3024961 + 495000\sqrt{11}\right)\right)\Big/ \left(\sqrt{\left(\left(-22388251749385790137716041903497414624275743496315549\right.\right.\right.

```
(34924207146378704054514015086872673362498 69 -
      27462378567654044250559169998407377913 0000 √11 +
      10915422709455000 00
       √(2722445 (50872173116052438577658808240460833727142 3 +
            68532908907834108568690240121030139255000 √11 ))) ^ (1/3) -
   27462378567654044250559169998407377913 0000 √11
    (34924207146378704054514015086872673362498 69 -
       27462378567654044250559169998407377913 0000 √11 +
       10915422709455000 00
        √(2722445 (50872173116052438577658808240460833727142 3 +
             68532908907834108568690240121030139255000 √11 ))) ^ (1/3) +
   21334422657915014376374467321
    (34924207146378704054514015086872673362498 69 -
       27462378567654044250559169998407377913 0000 √11 +
       10915422709455000 00
        √(2722445 (50872173116052438577658808240460833727142 3 +
             68532908907834108568690240121030139255000 √11 ))) ^ (2/3) -
   15382686857070634415047800 00 √11
    (34924207146378704054514015086872673362498 69 -
       27462378567654044250559169998407377913 0000 √11 +
       10915422709455000 00
        √(2722445 (50872173116052438577658808240460833727142 3 +
             68532908907834108568690240121030139255000 √11 ))) ^ (2/3) +
   10915422709455000 00 √(2722445
       (50872173116052438577658808240460833727142 3 +
         68532908907834108568690240121030139255000 √11 ))
    (-10437337304011 1 + (34924207146378704054514015086872673362498 69 -
         27462378567654044250559169998407377913 0000 √11 +
         10915422709455000 00
          √(2722445 (50872173116052438577658808240460833727142 3 +
               68532908907834108568690240121030139255000
                √11 ))) ^ (1/3)) ) /
(10477262143913611216354204526061802008749607 -
   82387135702962132751677509995222133739 0000
    √11 +
   32746268128365000 00
    √(2722445 (50872173116052438577658808240460833727142 3 +
       68532908907834108568690240121030139255000 √11 )))))))))
```

What is the point of showing all this? This is to show that *Mathematica* can take you to new places outside the reach of other packages. The measurement of curvature is one of optics oldest problems and is today one of its most fundamental tasks. For example, the exercise of measuring your eyes for glasses and contacts is about finding the right curvature for the spherical and cylindrical terms. By knowing the curvature of light from distant stars, radio telescopes are able to watch the North American continental plate bulge and sag by inches with changes in atmospheric pressure.

So the problem is fundamental and *Mathematica* has opened a new doorway. We have eight other terms of comparable complexity to deal with, and in practical applications, you'll need to be able to manipulate such terms. We will see not only that they require special treatment, but also that they will drive you to change some practices. This much output can drive the size of the file up quickly and can quickly saturate your virtual memory. When you have saturated your virtual memory, we urge you, when possible, to leave *Mathematica* and reboot. Your platform will slow to a crawl if the virtual memory is saturated.

Also, exercises like this will make clear distinctions between physical and virtual memory. You expect an enormous speed difference. But we found that some notebooks would not run under virtual memory but worked fine with physical memory. Our IT person swore that this was impossible, but the demonstration showed otherwise.

We also want to make the point that some of these results which may take minutes or hours to compute should be stored as output in modules. Suppose we did a calculation which takes 8 hours and outputs 100 pages. Obviously, you don't have time to compute this anew each day and the best thing is to store the output. Here are the steps to take to preserve the answer. First, create the output. (We will use the factorial function as a surrogate here.)

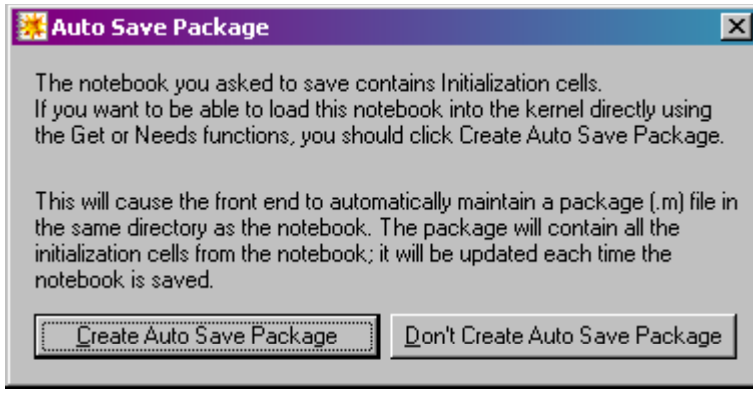
```
(* surrogate for a long duration computation *)
100!

933262154439441526816992388562667004907159682643816214685929638952 1
 759999322991560894146397615651828625369792082722375825118521091686 1
 40000000000000000000000
```

Next, we will open a new notebook which we call results.nb. We will paste this output into the input field of a variable called `out`. You will notice that the face of the print goes from regular to bold.

```
(* paste the output into the input of another cell *)
out =
  93326215443944152681699238856266700490715968264381621468592963895 2\
   17599993229915608941463976156518286253697920827223758251185210916\
   86400000000000000000000000;
```

We need to turn this into an evaluation cell. We select the cell and go through the menu bar Cell > Cell Properties > Initialization Cell. When we save the notebook into the packages directory (`dirPack`) as `long result.nb`, we will be prompted to create an autosave package. Click the left button (the default) to create the *.m file which *Mathematica* needs.



Finally, we need to load this module. In the notebooks where you need this result you can add this line to the header.

```
Get["long result.m", Path → dirPack];
```

Every time you open your other notebooks, this statement will execute. *Mathematica* will go to the *.m file. Then it will automatically load the variable `out`.

So, manipulation of large files is an integral part of the *Mathematica* curriculum.

Now the question is how do we resolve these enormous equations? `FullSimplify` ran on one term for over 12 hours without finishing. Earlier we saw an error message like this.

```
N::meprec : Internal precision limit $MaxExtraPrecision = 50.` reached while evaluating
```

When we click on this error, we get some advice on adjusting the numerical precision. The entry suggests using a `Block` function:

```
(* set the extra precision at 10 M digits *)
prec = 10^7;
(* count the number of zeros found *)
k = 0;
Block[{$MaxExtraPrecision = prec},
  (* the If test will force evaluation of the term *)
   result = (If[# == 0, k++; #, #]) &/@ diff;
 ];
(* output the number of zeros found *)
Print[k, " zeros found"];
```

Even a million digits were not enough. We promptly tried 10 million only to have the kernel crash after some hours.

(Notice that the Block statement prevents the precision from being adjusted on all computations.)

Built-in Functions > Warning Messages > Numerical Computation > Numerical Evaluation > General::meprec

We then tried to FullSimplify this term and aborted the calculation after 12 CPU hours. After all, there are nine terms to evaluate and we are looking at over 120 CPU hours to resolve this problem.

We then went inside the expression and tried all manners of substitutions and tried to extract pieces that would simplify, all without success. So we are forced finally to admit defeat after resolving only the two end points exactly and take comfort from the numerical and graphical evaluation. While we prefer an exact evaluation, in this case there was no other inverse solution which even came close. The original function and its inverse are plotted in table 2.4.

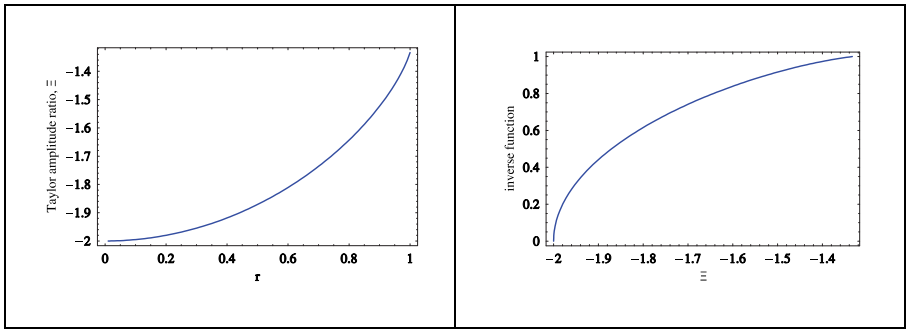


Table 2.4: The ratio of the Taylor expansion amplitudes and the inverse function. The function is on the left, the inverse on the right. The different aspect ratios complicate comparison.

We hope this exercise was instructive. We believe it was an ideal platform for us to display the practical use of *Mathematica*. We saw that you may take relatively simple examples and generate incredibly long equations and we saw how to manipulate the results to extract the information that we need. We also saw that there is not always a happy ending. Even though you have high confidence that a solution exists, the complexity may be more than *Mathematica* can handle.

C H A P T E R 3

Graphics examples

Mathematica is the preeminent symbolic computation package whose utility is greatly enhanced by an extensive set of visualization tools suitable for scientific visualization. There is a basic set of graphics routines which will quickly allow the user to plot functions and data easily in several formats. *Mathematica* presents a tool kit of graphics primitives enabling users to significantly extend graphics capabilities beyond the basic routines. We will see how easy it is to create animation sequences which considerably augment interpretation as we will see. Because of the power and facility of the graphics routines, we will devote a significant portion of our time to explaining them.

The online help for the mathematical functions is superlative. The help for the graphics functions is more Spartan; however, version 5 has significant improvements in this area. With the online help and this book, the user should be able to create complicated graphics objects. We will explicitly demonstrate capabilities that are not shown by example in the help files. For example, we will show how to create surface color maps and how to remove mesh lines from parametric curves.

In tribute, the burgeoning types of output *Mathematica* makes available, we close this chapter with a pictionary which allows the reader to find the output type he likes by inspection.

3.0.1 Color options

The graphics tools are complemented by professional-strength output utilities. The user can choose the color representations schemes of red, green, and blue (RGB) or cyan, magenta, yellow, and black (CMYK). And of course a monochrome pattern varying from black to white is available. In a moment we will start with the graphics

primitives and show how the various color palettes can be used. But first we want to discuss the color schemes.

Of course, the gray level (monochrome) maps are best when the image will be presented in monochrome; like a black and white document, you will get better results by rendering the graphic in gray level. Using a black and white copier to reproduce a color figure needlessly reduces the quality of the image. The distinction between RGB and CMYK is a distinction between the output methods. The RGB palette is designed for display on monitors; the actual pixels in the screen are either red, white or blue. However, most printing processes use a CMYK palette reflecting the four inks that are used. Translating between the two palettes has some ambiguity because the color spaces have different sizes.

Mathematica also presents a hue scheme which allows for variation of hue, saturation, and brightness. Much of the time we will simply want to vary colors and in this case there is a mechanism for a variation in hue only. We will treat the hue methods as a shortcut enabling us to quickly and simply color our figures.

In the common file loaded with the seed notebooks we have a list of predefined colors using a rustic shorthand. This precludes us having to type out the full color name and having to remember the combinations.

```
(* a sampling of predefined colors in the common module *)
{cred, cblu, cgrn, cwht, cblk, cpur}

{RGBColor[1, 0, 0], RGBColor[0, 0, 1], RGBColor[0, 1, 0],
 RGBColor[1, 1, 1], RGBColor[0, 0, 0], RGBColor[0.4, 0, 0.8]}
```

Demos > Palettes > Color Palette

Add-ons & Links > Standard Packages > Graphics > Colors

Built-in Functions > Graphics and Sound > Graphics Primitives > GrayLevel

Built-in Functions > Graphics and Sound > Graphics Primitives > RGBColor

Built-in Functions > Graphics and Sound > Graphics Primitives > Hue

Built-in Functions > Graphics and Sound > Graphics Primitives > CMYKColor

3.0.2 Output file types

After we have created the graphics we will want to store them. *Mathematica* 5 allows for a wide variety of outputs: a complete list is given in the help for the **Export** command. We will discuss three formats which are quite useful regardless of whether the host platform is using Linux, Mac OS or Windows. The three formats are EPS (enhanced PostScript), BMP (bitmap) and JPEG (joint project experts group). (We note that GIF, graphics interchange file format, can be used interchangeably with JPEG in the following discussion). The PostScript files are in a vector form, and these can scale to different sizes without compromising resolution. So the size of the graphic does not have much of an effect on the file size. In our examples we will

be very specific about the graphic size but is to allow the user to perform uniform scaling when the images are inserted into documents. When we create a PostScript image, we will include a TIFF (***) preview. The PostScript figure will be rendered when the document is printed, and the TIFF preview gives a rough representation on the screen. In general, a computer may not have a utility that allows the user to view the PostScript file. In these cases the TIFF preview is essential. (EPS files can be opened by applications like PhotoShop and Illustrator.)

The other formats yield best results when they are rendered in the desired size, or some factor of 2^n larger $n \in Integers$. The bitmap format works best for screen displays, for example, when using PowerPoint. The bitmap file is not as large as the PostScript file, but it is still of considerable size. The bitmap files shrink gracefully only when the reduction is some power of 2. So reducing by 50% will result in a faithful rendition, but reduction by 65% will distort details and text.

The final format to consider is JPEG (or GIFF). These files are very small and can be viewed in a web browser. Because of the small size, we are able to view dozens of images at a time, which is very handy for sorting through a collection of hundreds of images. The formats are well matched for a screen resolution of 72 dots per inch (dpi). Naturally when printed at 1200 dpi or 2400 dpi, the quality will be awful: details and sharp edges will be lost. For example, text may be unreadable. So these formats are primarily to display on web pages and aid in controlling your image library.

Table 3.1 below summarizes the three graphics formats of interest and their properties as discussed.

	EPS	BMP	JPG
File size	Very large	Large	Small
Resolution	Excellent	Marginal	Marginal
Application	Print	Display	View
Viewed with	-	Paint	Browser

Table 3.1: Graphics output formats.

A useful trick is to use a single routine to simultaneously render images in all three formats. We include such a prototype here. This can be very handy when it precludes the need to replicate graphics settings and reproduce a figure.

Built-in Functions > Graphics and Sound > Import and Export > Display (will refer to Export)

Built-in Functions > Graphics and Sound > Import and Export > Export

Built-in Functions > Graphics and Sound > Import and Export > Import

3.0.3 Displaying graphics

Some of the common variables that load when we execute a notebook control the display of graphics. The *Mathematica* command DisplayFunction is well documented

and the command is simple to use. We have found though that these surrogates are even easier to use.

The surrogates come in two flavors: a rule and an option. This is because some graphics commands accept the rule and some do not. We will clarify this through example. First, we will show the variables in rule form.

```
(* option form to turn display on *)
don
(* option form to turn display off *)
doff

DisplayFunction → (Display[$Display, #1] &)
```

```
DisplayFunction → Identity
```

This is the preferred usage. However, at times, we are limited to these.

```
(* option value to turn display on *)
von
(* option value to turn display off *)
voff

Display[$Display, #1] &
```

```
Identity
```

One word on entering these values. To turn the display on we actually enter `$DisplayFunction`, *not* `($Display, #1] &)`. That is how the entry is interpreted. You can see this clearly below.

```
(* what you enter and what Mathematica substitutes *)
$DisplayFunction

Display[$Display, #1] &
```

Let's demonstrate the two different environments. The `Plot` command is restrictive and will not accept a rule.

```
(* trying to use rule form *)
Plot[Sin[x], {x, -π, π}, doff];
```

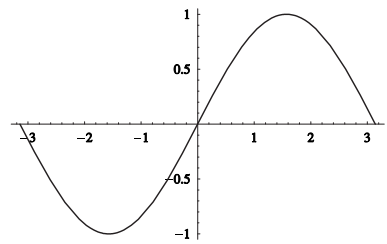
```
⚠ Plot::nonopt : Options expected (instead of doff) beyond position 2 in Plot[Sin[x], {x, -π, π}, doff]. An option must be a rule or a list of rules. More...
```

But when we switch to the option form, all is well.

```
(* switch to option value *)
g1 = Plot[Sin[x], {x, -π, π}, DisplayFunction → voff];
```

The plot is not displayed as requested. The plot was generated and we can view it with `Show`.

```
(* display the sin plot *)
Show[g1, don];
```



Let's look at a plot command that does accept rules. To do this, we will create a list of ten random $\{x,y\}$ pairs and use `ListPlot` to display them.

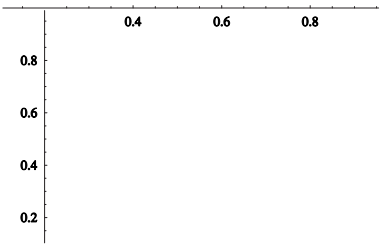
```
(* quick list of ten points {x, y} *)
SeedRandom[1];
a = Table[{Random[], Random[]}, {10}];
```

Now we shall plot them.

```
(* plot the list of points *)
g1 = ListPlot[a, doff];
```

The lack of output tells us that we have been successful. As always though, we should verify the result.

```
(* view the plot *)
Show[g1, don];
```



Built-in Functions > Graphics and Sound > Advanced Options > DisplayFunction

3.1 Graphics primitives

We will begin our exploration with the graphics primitives. As we go along we will also learn a bit more about the mathematical capabilities of *Mathematica*. The online help now lists the collection of all graphics primitives together, greatly expediting the learning process. We will concentrate on members from the subset

Point	Rectangle	Circle	Raster
Line	Cuboid	Disk	RasterArray

While we are exploring these primitives, we hope to give the reader a view from the bottom up. The goal is to use these primitives to also teach you the basics about plots: aspect ratios, object properties, etc.

Built-in Functions > Graphics and Sound > Graphics Primitives >

3.1.1 Examples using graphics primitives: the Dirac delta distribution

We are all familiar with functions. They are maps that relate an x value to a y value. Typically functions have derivatives and we can use them to compute areas via integration. But consider a generalization of this concept: we are unable to specify the

value of the function, but we may be able to specify its derivative or its enclosed area. Such a function would be called a generalized function, or in the earlier literature, an ideal function.

The noted physicist P.A.M. Dirac posited the general function that bears his name — the *Dirac delta function*. He developed this linear functional to perform theoretical calculations in quantum mechanics. It has some interesting properties that are summarized in [19]. For our display purposes, we will define the delta function as the limit of a sequence having unit area. The distribution is zero everywhere except at the origin where the value is not bounded.

Many people like to dispel the ambiguity of a generalized function by showing a sequence that will reach the delta function in the limit. We will show two such sequences here.

Delta function, Distribution (Generalized Function)

Isosceles triangles

We begin using isosceles triangles. As shown in [**], the area A under the triangle is given by

$$A = bh \tag{3.1}$$

where b is the base width and h is the height. Using integer values n for the height at the origin, we see that the base will extend from $-1/2n$ to $1/2n$. We will use this to enter points in on the **Line** command which has the syntax:

```
(* syntax for the Line primitive *)
Line[{pt[[1]], pt[[2]], ..., pt[[n]]}]
```

So to create a `Graphics` object for an arbitrary height we would use

```
(* points defining an isosceles triangle of height n, width 1/n *)
pts = {{-1, 0}, {-1/(2n), 0}, {0, n}, {1/(2n), 0}, {1, 0}};
(* generate a graphics primitive *)
g1 = Graphics[Line[pts]];
```

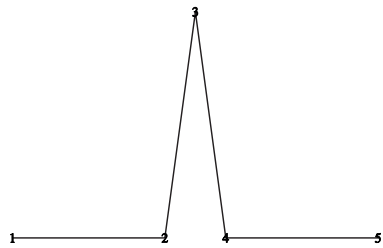
Notice that we have formed a list of points `pts` external to the `Line` primitive. This makes the code more explicit and more readable. *Mathematica* allows users to concoct lines of arbitrary length but we urge users to avoid the temptation to conglomerate everything.

A common problem that new users have is that they assemble large chunks or code and create something that won't work. We encourage a more deliberate

approach where the pieces are tested individually. In this light we want to test our line command to see if it performs as expected.

```
(* use the block structure to vary n locally and *)
(* to test primary plot command *)
Block[{n = 3},
  (* points defining an isosceles triangle of height n,
   width 1/n *)
  pts = {{-1, 0}, {-1/(2 n), 0}, {0, n}, {1/(2 n), 0}, {1, 0}};
  (* generate a graphics primitive *)
  g1 = Graphics[Line[pts]];
  (* render graphic on screen *)
  Show[g1];
  (* diagnostic since we haven't plotted a scale *)
  Print["connect the points ", pts];
 ];
```

```
connect the points {{-1, 0}, {-1/6, 0}, {0, 3}, {1/6, 0}, {1, 0}}
```



The witch's hat looks fine and the points are correct. While this exercise was simple, we want to use it to demonstrate a good method to debug more complicated figures.

The structure of *Mathematica* allows us to quickly label the points. This can be a lifesaver when you are using two different addressing schemes. A standard {x, y} address is measured from the lower left-hand corner and both variables increase as you move up and right. Matrices, however, have their origin in the upper left-hand corner and the row index increases as you move down; the column index increases as you move to the right.

Visualizing and ordering your points can make a tremendous difference. While we do this, we will learn a little bit about the flexibility *Mathematica* provides us. Our

first approach will be the ever-reliable `Do` loop. First, we present the basic syntax for the `Text` command.

```
(* prototype Text command *)
Text[text, address]
```

The basic command to label a point with the number `i` and the location `pts1` is:

```
(* i is the Do loop counter, pts are the point list from above *)
Text[i, pts[[i]]]
```

We will use this basic structure to build a table of graphics objects that labels the points on the line.

```
(* count the points *)
np = Length[pts];
(* label the points *)
(* create a table of Graphics objects *)
gtxt = Table[Graphics[Text[i, pts[[i]]]], {i, np}];
(* view the labeled points *)
g2 = Show[gtxt];
g3 = Show[g1, gtxt];
```

The output from this code is shown in table 3.2. Notice how we have used the `Show` command to combine graphics objects.

```
(* combining graphics objects *)
combinedgraphics = Show[graphics1, graphics2, ...];
```

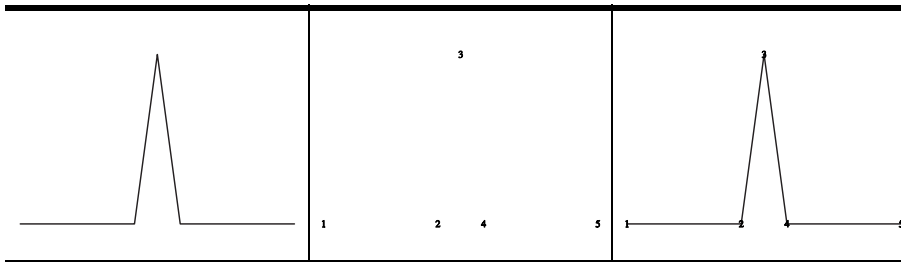


Table 3.2: The assembly sequence for the line with labeled points.

. Built-in Functions > Graphics and Sound > Graphics Primitives > Text

In general, we consider the `Table` command to be comparable to the **Do** command in terms of ease of use. Therefore, we will not show how to use both structures after this example. In this case, we will show how to employ a do-loop.

Our first step is to measure the length of the point list to find out how many lines we are drawing. The function of the do-loop should be transparent.

```
(* count the points *)
np = Length[pts];
(* preserve the original object g1; manipulate gtxt *)
gtxt = g1;
(* loop through the points *)
Do[
  (* label the points *)
  gtxt = Show[gtxt, Graphics[Text[i, pts[[i]]]]];
  , {i, np}];
(* view the labeled points *)
Show[gtxt];
```

This sequence produces the exact same figure as the example based on the `Table` command. This method certainly works, and you should use this structure as long as you are comfortable. However, *Mathematica* has a great faculty for lists. Because of this, we want to emphasize how to use the lists.

Let's examine the functioning of the `Table` command a bit more closely. We start by showing the output to help the reader visualize the operation. In this case, we are building a table of graphics objects.

```
Table[Graphics[Text[i, pts[[i]]]], {i, np}]

{- Graphics -, - Graphics -, - Graphics -, - Graphics -, - Graphics -}
```

The next option is to move the `Table` command inside the `Graphics` command and create a table of `Text` objects as shown below. We can do this because **Graphics**, like almost all *Mathematica* commands, works as well on lists as it does on single arguments.

```
Table[Text[i, pts[[i]]], {i, np}]

{Text[1, {-1, 0}], Text[2, {-1/6, 0}],
 Text[3, {0, 3}], Text[4, {1/6, 0}], Text[5, {1, 0}]}
```

This code produces the exact same figure as before.

A natural consideration would be to consider putting the `Table` command inside of the `Text` command. This will fail.

```
Text[Table[i, pts[[i]], {i, np}]]

⚠ Table::itform : Argument pts[[i]] at position
    2 does not have the correct form for an iterator . More...

Text[Table[i, pts[[i]], {i, np}]]
```

```
Text[Table[{i, pts[[i]]}, {i, np}]]

Text[
 {{1, {-1, 0}}, {2, {-1/6, 0}}, {3, {0, 3}}, {4, {1/6, 0}}, {5, {1, 0}}}]
```

```
Graphics[Text[Table[i, pts[[i]], {i, np}]]]
```

```
Table::itform : Argument pts[[i]] at position
    2 does not have the correct form for an iterator . More...
```

```
- Graphics -
```

```
Show[%]
```

```
Text::argbu :
  Text called with 1 argument ; between 2 and 5 arguments are expected . More...
```

The graphics primitives don't accept list arguments. This is probably because they are so general in their argument types that it would be difficult to distinguish a short list from a command.

As you will see, this does not mean that we cannot map them to a list. We are simply excluded from the list syntax. We can use a pure function which is applied to a list where the list is:

```
(* feeder list for a function to create Text directives *)
tlst = Table[{i, pts[[i]]}, {i, np}]
```

$$\left\{\{1, \{-1, 0\}\}, \left\{2, \left\{-\frac{1}{10}, 0\right\}\right\}, \{3, \{0, 5\}\}, \left\{4, \left\{\frac{1}{10}, 0\right\}\right\}, \{5, \{1, 0\}\}\right\}$$

Let's write a simple function that allows us to sort out the inputs and outputs. We'll call this experimental function `fcn`.

```
(* diagnostic function to show what pieces we are grabbing *)
fcn = (Print["first element = ", #[[1]], "; second element = ", #[[2]]) &;
```

When applied to `tlst` we get:

```
(* Map the function over the list *)
fcn /@ tlst;
```

```
first element = 1; second element = {-1, 0}
```

```
first element = 2; second element = {-1/6, 0}
```

```
first element = 3; second element = {0, 3}
```

```
first element = 4; second element = {1/6, 0}
```

```
first element = 5; second element = {1, 0}
```

We see clearly how the function moved through all five points in the list. Also we see that the slot operator #1 grabbed the integer part and #2 grabbed the list representing the address. This example may seem simplistic, but we strongly encourage our readers to use such diagnostics as they are using new commands or bringing multiple commands together.

It should be straightforward to rewrite **fcn** to fit our needs:

.

```
(* production function to produce Text directives *)
fcn = (Text[#[[1]], #[[2]]]) &;
```

When this function is mapped over the points list, we get this:

```
(* create a list of graphics firectives *)
fcn /@ tlst
```

```
{Text[1, {-1, 0}], Text[2, {-1/6, 0}],
 Text[3, {0, 3}], Text[4, {1/6, 0}], Text[5, {1, 0}]}
```

We can assign this to a variable and convert it to a graphics object with the `Show` command.

```
(* view the numbers marking the points *)
gtxt = Show[Graphics[%]];
```

The inquisitive reader may be asking: why not simply `Map Text` across list?

```
(* trying to Map Text across tlst *)
Text /@ tlst
Show[Graphics[%]]
```

$$\left\{\text{Text}[\{1, \{-1, 0\}\}], \text{Text}\left[\left\{2, \left\{-\frac{1}{6}, 0\right\}\right\}\right],\right.$$
$$\left.\text{Text}[\{3, \{0, 3\}\}], \text{Text}\left[\left\{4, \left\{\frac{1}{6}, 0\right\}\right\}\right], \text{Text}[\{5, \{1, 0\}\}]\right\}$$

```
Text::argbu : Text called with 1 argument ; between 2 and 5 arguments are expected . More...
```

The short answer, of course, is that it will not work. The problem has to do with the braces inside the brackets. `Text` is looking for two arguments and it is getting a single argument — a list. We must be careful because *Mathematica* treats the scalar *a* differently than it would the one-item list {*a*}.

Built-in Functions > Programming > Functional Programming > Function (&)

DIagnostics aside, we will pull the pieces together to build a Module that will draw the first few terms in the sequence of isosceles triangles that represents the Dirac delta function.

Let's create a do-loop to create a series of triangles. First, we'll specify the range of *n* as

$$0 < n \leq nmax \quad . \tag{3.2}$$

Next, we'll create a shell plot. This will be an empty plot in the format that we want. The lines that we draw will then be combined with this shell. The height of the plot area must be greater than `nmax`. The widest triangle ($n = 1$) stretches from -1/2 to 1/2, so a domain of [-1, 1] will be adequate.

```
(* draw an empty graph as a canvas for the triangles *)
bottom = -0.1; top = nmax + 1 / 3;
g0 = Plot[bottom, {x, -1, 1},
   PlotRange → {bottom, top}, Frame → True, Axes → False];
```

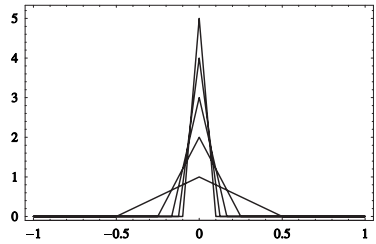
Notice that the bottom of the plot is not zero. We do this so that a line along $y = 0$ does not overwrite the plot frame and disappear.

The next chore is to write a function to create the point list for each line. All three of the subsequent methods will use this function to create the separate lines.

```
(* function to generate points defining isosceles triangle *)
pts[n_Integer] := {{-1, 0}, {-1/(2n), 0}, {0, n}, {1/(2n), 0}, {1, 0}};
```

Hopefully, the following listing is clear for you. If there is some ambiguity, take advantage of the interactive environment and test the pieces and commands individually.

```
(* Do loop implementation *)
(* empty container holding the separate lines *)
lines = {};
(* loop through nmax lines *)
Do[
  (* create a single line *)
  g2 = Graphics[Line[pts[n]]];
  (* add this line to the container *)
  AppendTo[lines, g2];
  , {n, nmax}];
(* show the final result *)
g3 = Show[g0, lines];
```



With the do-loop aside, we would like to showcase the *Mathematica* tools for working with lists. The notation is compact, so we will elaborate. We know that `pts[i]` makes a list of points for the *i*th line. We create a list object `allpts` which contains the points for all `nmax` lines. Then we mapped the `Line` command across `allpts` to create a list of `Line` objects ready for conversion into a `Graphics` object. We use the `Show` command to pull in the background and to convert the `Line` objects into `Graphics` objects.

```
(* create a list of the points for all nmax lines *)
allpts = Table[pts[n], {n, nmax}];
(* turn each set of points into a line *)
Line /@ allpts;
(* merge the pieces with Show *)
g1 = Show[g0, Graphics[%]];
```

This produces the exact same figure as above.

This notation allows for extremely compact code writing. See how we can fold everything into a line of modest length.

```
(* we can create a super command *)
g1 = Show[g0, Graphics[Line /@ Table[pts[n], {n, nmax}]]];
```

We do not encourage this type of coding practice until the reader is quite comfortable with *Mathematica*.

For those of you who looked at the do-loop longingly, we would like to motivate you to adapt the list-based commands. To that end, we will show explicitly how these terms are working. We think you'll agree that the process is simple; the only new thing is the notation.

Creating a list of all the lines means creating a list where each element is the list of points for one line.

```
(* create a list of the points for all nmax lines *)
allpts = Table[pts[n], {n, nmax}]
```

$$\left\{\left\{\{-1, 0\}, \left\{-\frac{1}{2}, 0\right\}, \{0, 1\}, \left\{\frac{1}{2}, 0\right\}, \{1, 0\}\right\},\right.$$
$$\left\{\{-1, 0\}, \left\{-\frac{1}{4}, 0\right\}, \{0, 2\}, \left\{\frac{1}{4}, 0\right\}, \{1, 0\}\right\},$$
$$\left\{\{-1, 0\}, \left\{-\frac{1}{6}, 0\right\}, \{0, 3\}, \left\{\frac{1}{6}, 0\right\}, \{1, 0\}\right\},$$
$$\left\{\{-1, 0\}, \left\{-\frac{1}{8}, 0\right\}, \{0, 4\}, \left\{\frac{1}{8}, 0\right\}, \{1, 0\}\right\},$$
$$\left.\left\{\{-1, 0\}, \left\{-\frac{1}{10}, 0\right\}, \{0, 5\}, \left\{\frac{1}{10}, 0\right\}, \{1, 0\}\right\}\right\}$$

The equivalent do-loop for is shown below.

```
(* create an empty list *)
allpts = {};
 (* loop through all of the lines *)
 Do[
  (* add the points for line n to the master list *)
  AppendTo[allpts, pts[n]];
  , {n, nmax}]
 (* show the results *)
 allpts
```

$$\left\{\left\{\{-1, 0\}, \left\{-\frac{1}{2}, 0\right\}, \{0, 1\}, \left\{\frac{1}{2}, 0\right\}, \{1, 0\}\right\},\right.$$
$$\left\{\{-1, 0\}, \left\{-\frac{1}{4}, 0\right\}, \{0, 2\}, \left\{\frac{1}{4}, 0\right\}, \{1, 0\}\right\},$$
$$\left\{\{-1, 0\}, \left\{-\frac{1}{6}, 0\right\}, \{0, 3\}, \left\{\frac{1}{6}, 0\right\}, \{1, 0\}\right\},$$
$$\left\{\{-1, 0\}, \left\{-\frac{1}{8}, 0\right\}, \{0, 4\}, \left\{\frac{1}{8}, 0\right\}, \{1, 0\}\right\},$$
$$\left.\left\{\{-1, 0\}, \left\{-\frac{1}{10}, 0\right\}, \{0, 5\}, \left\{\frac{1}{10}, 0\right\}, \{1, 0\}\right\}\right\}$$

You must agree that the `Table` command has a simple syntax and has the advantage of creating the container for us.

The next step involves mapping the `Line` operator across each element in the list of lines. This simply means that we take each element from the list `allpts` and `Map`s the `Line` operator to it.

```
(* turn each set of points into a line *)
Line /@ allpts
```

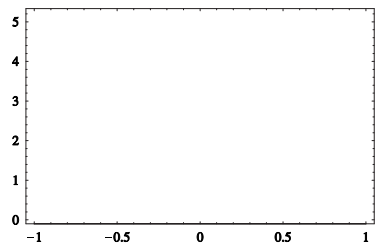
$$\left\{\text{Line}\left[\left\{\{-1, 0\}, \left\{-\frac{1}{2}, 0\right\}, \{0, 1\}, \left\{\frac{1}{2}, 0\right\}, \{1, 0\}\right\}\right],\right.$$
$$\text{Line}\left[\left\{\{-1, 0\}, \left\{-\frac{1}{4}, 0\right\}, \{0, 2\}, \left\{\frac{1}{4}, 0\right\}, \{1, 0\}\right\}\right],$$
$$\text{Line}\left[\left\{\{-1, 0\}, \left\{-\frac{1}{6}, 0\right\}, \{0, 3\}, \left\{\frac{1}{6}, 0\right\}, \{1, 0\}\right\}\right],$$
$$\text{Line}\left[\left\{\{-1, 0\}, \left\{-\frac{1}{8}, 0\right\}, \{0, 4\}, \left\{\frac{1}{8}, 0\right\}, \{1, 0\}\right\}\right],$$
$$\left.\text{Line}\left[\left\{\{-1, 0\}, \left\{-\frac{1}{10}, 0\right\}, \{0, 5\}, \left\{\frac{1}{10}, 0\right\}, \{1, 0\}\right\}\right]\right\}$$

If the growing list of *Mathematica* commands and syntax structures is taxing your memory and you have forgotten what it means to `Map` a function to an argument, just remembers that when we apply the sine function to a list of numbers we get a list of sine values. You are quite comfortable with the concept; it is just the appellation that seems new.

The astute reader may ask if we could also create this sequence using the `ListPlot` command. This will involve a bit of manipulation, so we would like to discuss this approach.

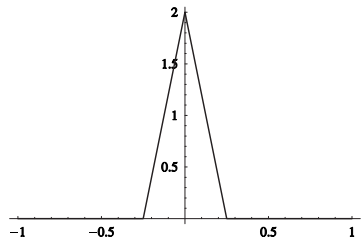
We will discuss `ListPlot` more in a later sections, but for now we are content with the fact that it will plot a list of points, which is exactly what we have. We can `Map ListPlot` to the entire points list `allpts` which is still in memory.

```
(* Apply ListPlot across all the points *)
g4 = ListPlot /@ allpts
```



This isn't quite what we need since the points are unconnected. After checking the Help Browser we see that we can modify `ListPlot` to join the plot points. We test this syntax like so.

```
(* select a specific case for testing *)
n = 2;
(* use ListPlot to display and to connect these points *)
g4 = ListPlot[pts[n], PlotJoined→ True];
```



The attentive reader is not alarmed by the presence of the *x* and *y* axes because he remembers that we can load the graphic `g0` first and our final graphic will inherit the property:

```
Axes→ False
```

The question now is how does one map `ListPlot` with options changed? We again will create a function and the function will be the `ListPlot` command with the options that we need.

```
(* function to apply across allpts to changr ListPlot options *)
fcn = (ListPlot[#, PlotJoined→ True, doff]) &;
```

Creating the plot is trivial now.

```
(* create all the triangles *)
g4 = fcn /@ allpts;
```

This produces the exact same plot as the other two methods.

To complete the discussion we bring up an important topic that causes new users some consternation. We have just drawn five lines on one plot. How can we distin-

guish them? You, the reader, may wish to use different colors, or dashing or some other method. We will show how to vary the line thickness. We are not suggesting that the thickness variation is a good way to distinguish lines, but the method serves as a template for other schemes the reader may have.

The application syntax is shown here.

```
(* Thickness operates on
  primitives inside of the Graphics command *)
Graphics[{Thickness[th], Line[pts]}];
```

Going back to the example using the `Line` operator we see that we need to make a change where we have:

```
(* turn each set of points into a line *)
Line /@ allpts;
```

Of course if we wanted to apply a single option to all the `Line`s, we could simply modify the `Graphics` command at the end.

```
(* merge the pieces with Show *)
g1 = Show[g0, Graphics[%]];
```

But we need to have a different directive for each line. This is the same type of problem that we just solved for the `ListPlot` problem. We need to write a function so that we can pass the options. First, we will decide what directive to apply and then show the function we will use to map the directive across the lines.

After a brief experimentation we have decided a good choice will be:

```
Thickness[0.0085 - n 0.001]
```

where `n` is the number of the element in `allpts`. An attempt using this syntax will fail.

```
(* attempting to alter line thickness *)
fcn = (Graphics[{Thickness[0.0085 - n 0.001], Line[#]}]) &;
```

This is because we are not inside a loop with a counter `n`. The resourceful reader may try set `n = 0` before calling the function and replacing `n` with `n++`. This will not work because there will be no incrementing during the mapping.

Fortunately, we see that the basic entity here, the list of points for the isosceles triangle, contains an `n`.

```
(* points defining an isosceles triangle of height n, width 1/n *)
pts = {{-1, 0}, {-1/(2 n), 0}, {0, n}, {1/(2 n), 0}, {1, 0}};
```

We turn our attention to the third element.

```
(* the third element has a naked n *)
pts[[3]]

{0, n}
```

The third element is a list. We want the second element from this list.

```
(* the second element of the list above is n *)
%[[2]]

n
```

So the syntax we will use is:

```
(* we want the third element, and from that the second element *)
pts[[3]][[2]]

n
```

The function we will use takes the form:

```
(* alter line thickness according to n *)
fcn = (Graphics[{Thickness[0.0085 - #[[3]][[2]] 0.001], Line[#]}]) &;
```

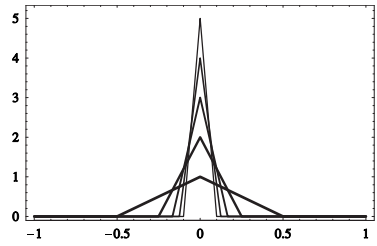
Again we caution the new user to start with a template command like this:

```
(* function template *)
fcn = (Graphics[{Thickness[], Line[]}]) &;
```

to balance delimiters and avoid punctuation purgatory.

When we bring the pieces together we see this.

```
(* create a list of the points for all nmax lines *)
allpts = Table[pts[n], {n, nmax}];
(* create a Graphics directive with differing thicknesses *)
fcn = (Graphics[{Thickness[0.0085 - #[[3]][[2]] 0.001], Line[#]}]) &;
(* turn each set of points into a line *)
fcn /@ allpts;
(* merge the pieces with Show *)
g5 = Show[g0, %];
```



Here the list-based commands allow for extremely compact notation.

```
(* ultra compact notation *)
g5 = (Graphics[{Thickness[0.0085 - #[[3]][[2]] 0.001], Line[#]}]) & /@
   Table[pts[n], {n, nmax}];
(* merge the pieces with Show *)
g5 = Show[g0, g5];
```

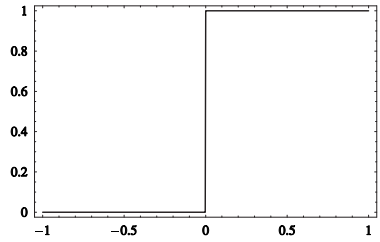
In closing we have looked at three different paradigms to create the delta function sequence with isosceles triangles. One should not be confused by this multiplicity. If you are overwhelmed, pick the paradigm that you are most comfortable with and move forward.

Tophats

Let us take a brief detour out of graphics primitives to look at yet another sequence for representing the delta function. We can construct a sequence similar to the one above using rectangles of unit area. If the rectangle height is n, the width must be $1/n$. This may sound like a job for the `Rectangle` command, but the problem there is that a rectangle is not a function: it is dual valued. Instead we will build a tophat function using the `UnitStep` function.

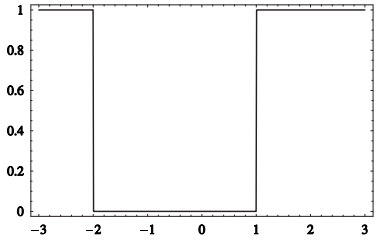
Of course, we could show how to use the `UnitStep` function directly, but a major thrust of this book is to teach the reader how to explore on his own. So if a professor told you that you could build a tophat using the unit step function, what would you do? The first task would be to consult the Help Browser and then plot the function.

```
g0 = Plot[UnitStep[x], {x, -1, 1}, Axes → False, Frame → True];
```



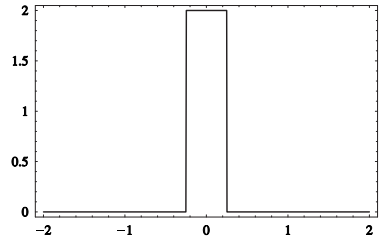
The Help Browser has an example showing how to construct crenellations. This shows us the basic pattern that we will use. Following their example we construct a more advanced function with the `UnitStep` command.

```
(* exploring the unit step *)
g1 =
  Plot[UnitStep[(x - 1) (x + 2)], {x, -3, 3}, Axes → False, Frame → True];
```



We show that we have created a negative tophat. We need to flip it over and fix the boundaries and amplitude.

```
(* building a tophat function *)
(* amplitude is n *)
(* width is 1/n; [-1/(2n), 1/(2n) *)
n = 2;
g2 = Plot[n UnitStep[-(x - 1/(2n)) (x + 1/(2n))],
   {x, -2, 2}, Axes → False, Frame → True];
```

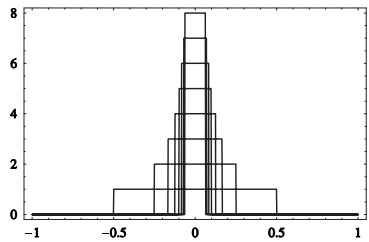


Now that we have identified the core process for building a tophat we can encapsulate the functionality.

```
(* define a top hat *)
(* height = n; width = 1/n *)
d[x_, n_Integer] := n UnitStep[-(x - 1/(2n)) (x + 1/(2n))]
```

In this function, *x* is a plot variable and we will use an integer sequence to supply values for *n*. The code below will create the first eight rectangles in the sequence.

```
(* number of rectangles to create *)
nrec = 8;
(* create a sequence of functions *)
lst = Table[d[x, i], {i, nrec}];
(* plot all rectangles *)
g1 = Plot[Evaluate[lst], {x, -1, 1}, Axes → False, Frame → True];
```



This is certainly an interesting sequence and we now want to show how to make the older (wider) figures disappear as we build up the sequence. The best choice for this is to use the `GrayLevel` command.

Mathematica provides you with a helpful chart of the gray levels and new users will be surprised to find it with the Color Graphics. More seasoned veterans realize this is because `GrayLevel` is a coloring option.

Demos > Graphics Gallery > Color Charts > GrayLevel

Since our tophats are in a table, we will construct a table of GrayLevel directives to map to the tophats. Again we will do this in steps. Too many times we have been asked to debug huge conglomerations of undebugged fragments. We often find it more expedient to start fresh from the beginning than to unravel kludge code.

For our static example here, we will not want the gray level to fade out completely, so there is a minimum fade we will tolerate. The term `grf` marks this value as 0.9. Just to keep things parametric we also define a maximum darkness of `gri` being 0 (completely black). After fixing the scale we will decide how many steps we will allow across the scale and from this compute the fundamental change Δ in `GrayLevel` space.

With these values in hand we are able to build a table for `GrayLevel` directives which we will show here.

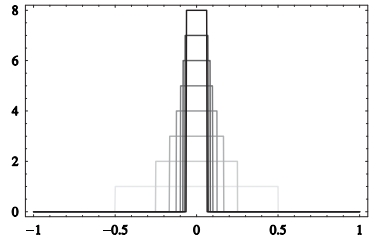
```
(* range of GrayLevel variation allowed *)
gri = 0.9; grf = 0;
(* unit step in gray colorspace *)
Δ = (grf - gri)/(nrec - 1);
(* GrayLevel directives for plot *)
stlst = Table[GrayLevel[gri + (i - 1) Δ], {i, nrec}]

{GrayLevel[0.9], GrayLevel[0.771429], GrayLevel[0.642857],
 GrayLevel[0.514286], GrayLevel[0.385714], GrayLevel[0.257143],
 GrayLevel[0.128571], GrayLevel[1.11022 × 10^-16]}
```

Look at how trivial it is to map this table. We just needed to append:

```
PlotStyle → stlst
```

```
g2 = Plot[Evaluate[lst], {x, -1, 1},
   Axes → False, Frame → True, PlotStyle → stlst];
```



If you are comfortable with the `Table` command, you could fall back on the reliable `Do` loop. You can replace

```
stlst = Table[GrayLevel[gri + (i - 1) Δ], {i, nrec}]
```

with the equivalent

```
(* create an empty list to store GrayLevel directives *)
stlst = {};
 (* sweep through color
   scale creating equally spaced directives *)
 Do[
   AppendTo[stlst, GrayLevel[gri + (i - 1) Δ]];
   , {i, nrec}];
 stlst

{GrayLevel[0.9], GrayLevel[0.771429], GrayLevel[0.642857],
 GrayLevel[0.514286], GrayLevel[0.385714], GrayLevel[0.257143],
 GrayLevel[0.128571], GrayLevel[1.11022 × 10^-16]}
```

3.1.2 Examples using graphics primitives: Lattice field theory

In an influential 1983 article [15], Claudio Rebbi brought the mysteries of lattice gauge field theory to a popular audience. His ambitious undertaking managed to explain how a quantum field theory formulated for a discrete world was able to explain the fundamental phenomenon of quark confinement.

As the quark model of hadron structure gained credence in the 1960's and 1970's, considerable experimental efforts were undertaken to find or create isolated quarks. The belief was that if all baryons were composed of quarks, we should be able to find or isolate naked quarks. A broad range of determined efforts failed to find any evidence for isolated quarks.

Perhaps the theory of the color force, quantum chromodynamics (QCD) was a victim of the success of the theory of the electromagnetic force (QED) which is the most precise theory ever formulated. The important distinction between the two quantum field theories of QCD and QED is that QED is perturbative and this allows calculations to approach finite values because successive terms in an approximation are considerably smaller.

http://scienceworld.wolfram.com/physics/QuantumElectrodynamics.html

http://scienceworld.wolfram.com/physics/GyromagneticRatio.html

QCD is non-perturbative and will not yield to the theoretic methods of its predecessor QED. Hence, the development of the lattice, a discrete version of the continuous world that we live in. A basic lattice may have dimensions of $x \times y \times z \times t$ of $10 \times 10 \times 10 \times 10$. Time and space are no longer continuous: they exist only at the lattice vertices and fields can exist only on the links connecting two vertices.

This all sounds a bit abstract and is difficult for many people to visualize. But Rebbi's article contains what we call Rebbi diagrams which give this process a simple form that fosters intuition.

Let us proceed to create a Rebbi diagram for QED. We start with some fundamental parameters. We will study a two-dimensional slice of 5×4 vertices $\{\xi, \eta\}$. We want the vertices to have some spatial extent since points are so hard to see, so each one will be represented by a small disk of radius 0.075 (ρ) with a light gray shading (`vgry`).

```
(* define key parameters *)
{ξ, η} = {5, 4};              (* spatial dimensions *)
ρ = 0.075;                    (* vertex radius *)
vgry = GrayLevel[0.5];        (* vertex shading *)
r = 2ρ;                       (* charge radius *)
I2 = IdentityMatrix[2];       (* basis vectors *)
```

As you can see, construction of the graphics primitives is a simple task. (For this display we used a much smaller 3×2 grid.)

```
(* table of graphics primitives *)
verts = Table[{vgry, Disk[{μ, ν}, ρ]}, {μ, ξ}, {ν, η}]
```

```
{{{GrayLevel[0.5], Disk[{1, 1}, 0.075]},
  {GrayLevel[0.5], Disk[{1, 2}, 0.075]}},
 {{GrayLevel[0.5], Disk[{2, 1}, 0.075]},
  {GrayLevel[0.5], Disk[{2, 2}, 0.075]}},
 {{GrayLevel[0.5], Disk[{3, 1}, 0.075]},
  {GrayLevel[0.5], Disk[{3, 2}, 0.075]}}}
```

Now that we have commands for the vertices, we will process them through the `Graphics` command.

```
(* render the image *)
pv = Show[Graphics[verts], AspectRatio → η/ξ];
```



Next, we will add some charges.

```
(* create a list of charges *)
(* {{x,y}, ±} *)
clist = {{{3, 3}, "+"}, {{4, 3}, "-"}}; (* + at {3,3}, - at {4,3} *)
(* count the number of charges *)
nc = Length[clist];
```

As always, we are counting the number of charges rather than entering `nc = 2`. It is difficult to estimate how many bugs would be eliminated if this simple procedure were followed. So many times programmers make changes and forget to update variables like `nc`. Let the computer do what it does so well: counting.

3.1.3 Examples using graphics primitives: Gray scales

While there is a Legend package available (as shown in 2D graphics pictionary, plots 34 and 68), you may find the tool a tad rustic. Regardless, it is instructive to see how to assemble one on your own.

This endeavor is simple enough and we shall construct a simplistic color scale using the gray levels. During this process we will discover an important problem and present its solution.

Again we caution the reader about the problems of stringing together untested code fragments. A more deliberate approach is almost always faster. Too many users want to learn to swim by jumping into the deep end of the pool. In this case we will see that a graduated approach allows us to quickly identify a potential problem.

For this problem we will use the graphics primitive `Rectangle`. We will use the syntax:

```
Rectangle[{xmin, ymin}, {xmax, ymax}, graphics]
```

where `{xmin, ymin}` and `{xmax, ymax}` are the coordinates of the lower left-hand corner and the upper right-hand corner, respectively. The final argument represents the shading directives we will map.

Our first decision is how many graduations do we want in our scale? It would be nice to give the scale a smooth look with imperceptible graduations, so we will choose a large number. However, we will use a small number in development to accelerate the process.

```
(* number of shades *)
ns = 3; Δ = 1 / ns;
```

Next, we will build a list of the corners needed for `Rectangle`. The `Table` command is the ideal choice. Note that `w` is the width which we have not supplied yet.

```
(* list of lower left-hand and upper right-hand corners *)
lst = Table[{{0, i Δ}, {w, (i + 1) Δ}}, {i, 0, ns - 1}]
```

$$\left\{\left\{\{0,\ 0\},\ \left\{w,\ \frac{1}{3}\right\}\right\},\ \left\{\left\{0,\ \frac{1}{3}\right\},\ \left\{w,\ \frac{2}{3}\right\}\right\},\ \left\{\left\{0,\ \frac{2}{3}\right\},\ \{w,\ 1\}\right\}\right\}$$

If you are uncomfortable with the `Table` command, you can use this `Do` loop. You should study the two methods used together here to see how they work.

```
(* empty list to hold the bounding corners *)
lst = {};
(* sweep through the graduations *)
Do[
  (* append {llhc, urhc} *)
  AppendTo[lst, {{0, i Δ}, {w, (i + 1) Δ}}];
  , {i, 0, ns - 1}];
(* display the list of bounding corners *)
lst
```

$$\left\{\left\{\{0,\ 0\},\ \left\{w,\ \frac{1}{3}\right\}\right\},\ \left\{\left\{0,\ \frac{1}{3}\right\},\ \left\{w,\ \frac{2}{3}\right\}\right\},\ \left\{\left\{0,\ \frac{2}{3}\right\},\ \{w,\ 1\}\right\}\right\}$$

On a minor stylistic point we want to state that the `Do` loop iterator could have been over the range `1` to `ns + 1`. Readers should use whichever convention they are most comfortable with.

```
(* alternate Do loop running from 1 to ns + 1 *)
(* use the option most comfortable to you *)
Do[
  AppendTo[lst, {{0, (i - 1) Δ}, {w, i Δ}}];
  , {i, ns + 1}];
```

Now that we have specified the location and size of the rectangles we need to supply a shade. Note that we are not going to supply a graphics directive. There is no need to. When we sweep through `lst` to convert the points into `Rectangle`s, we will also convert the shades into `GrayLevel` commands. What shade will each box have? It will be shaded with `GrayLevel[iΔ]`. Therefore, we need to include the quantity `iΔ` with each pair of corners.

Now we need to turn our numerical corners list into a list of graphics. First, we will just isolate the components and state how they will be used. This should make the subsequent function definition more clear.

```
(* count the number of boxes *)
nbox = Length[lst];
(* sweep through all the elements in lst *)
Do[
  (* grab the GrayLevel shade *)
  shade = lst[[i]][[1]];
  (* grab the lower left-hand cornder *)
  llhc = lst[[i]][[2]][[1]];
  (* grab the upper right-hand corner *)
  urhc = lst[[i]][[2]][[2]];
  (* explanatory print *)
  Print[i, ".  Rectangle[",
   {llhc, urhc}, "] will be GrayLevel[", shade, "]"];
  , {i, nbox}];
```

```
1.  Rectangle [{0, 0}] will be GrayLevel [0]
```

```
2.  Rectangle [{0, 1/3}] will be GrayLevel [1/3]
```

```
3.  Rectangle [{0, 2/3}] will be GrayLevel [2/3]
```

Let's replicate this functionality in a function. Our first step is to write a shell for the command since we are going to end up with a lot of punctuation.

```
fcn = (Print[]) &;
```

Then we write out the `Print` statement on a separate line and see that we have balanced the quotes and braces.

```
Print["Rectangle[", {#[[2]], #[[3]]}, "] will be GrayLevel[", #[[1]], "]"];
```

Now we paste this `Print` command inside of `fcn`.

```
(* verify that we are grabbing the pieces in the proper order *)
fcn = (Print["Rectangle[", {#[[2]], #[[3]]},
      "] will be GrayLevel[", #[[1]], "]"];) &;
```

This process may seem unnecessary, but we find it a useful way to avoid Punctuation Purgatory.

⚠ Don't be reluctant to construct commands in steps that have been checked by the Front End interpreter for proper punctuation.

As always then we will `Map` this function across some list.

```
(* Map the function across the list *)
fcn /@ lst;
```

```
Rectangle [{{0, 0}, {w, 1/3}}] will be GrayLevel [0]
```

```
Rectangle [{{0, 1/3}, {w, 2/3}}] will be GrayLevel [1/3]
```

```
Rectangle [{{0, 2/3}, {w, 1}}] will be GrayLevel [2/3]
```

This seems to be in order. But with *Mathematica,* we are easily able to test this. We select a width parameter `w` and generate some boxes.

By proceeding in this fashion we have exposed the inner workings of `fcn` and made its operation transparent. Now we know that we are grabbing the pieces in the proper manner and can now replace with the functional form that we need.

```
(* w is the width of the scale *)
(* use a very skinny scale that does not dominate the graphic *)
w = 0.1;
```

We will start with a function template again to allow the front end to help us balance out delimiters.

```
(* template to balance delimiters *)
fcn = (Graphics[{GrayLevel[], Rectangle[]}]) &;
```

We should reflect upon this syntax. It says that we are going to produce a graphics object. This graphics object will be given a `GrayLevel` shading and will be a `Rectangle`.

The full form of the function is shown here.

```
(* new function to generate graphics objects *)
fcn = (Graphics[{GrayLevel[#[[1]]], Rectangle[#[[2]], #[[3]]]}]) &;
```

We apply this function straightaway to the list `lst`.

```
(* Map fcn across the list to generate Graphics objects *)
g00 = fcn /@ lst

{- Graphics -, - Graphics -, - Graphics -}
```

Pleased to see a list of `Graphics` objects, we will `Show` them to see how they look.

```
(* what we have built so far *)
g01 = Show[g00];
```



The number of rectangles is good. The shading is good. At first glance, we are surprised to see such a chubby graphic. After all, we specified a height of unity and a width of 0.1. But we remember that *Mathematica* defaults the aspect ratio to the golden ratio. We can remedy this with an explicit setting for the aspect ratio.

Golden Ratio

```
(* there are three shades of gray *)
(* eash shaded region is one third the total height *)
g01 = Show[g00, AspectRatio → 1 / w, ImageSize → 0.25 72];
```



This looks good and we call attention to the fact that this figure has a size of a quarter of an inch (0.25 72). With this high aspect ratio graphic that means the height will be 10 times larger — 2.5 inches. If we had accepted the default size of 4 inches the graphic would have been 40 inches tall on our screen.

⚠ Be careful setting the size on high aspect ratio graphics.

How many graduations do we need? Well, we can quickly prepare graphics with increasingly fine resolution. It is best to use trial prints to see which level of resolution that you need.

5 divisions	25 divisions	50 divisions	100 divisions	200 divisions
				

Table 3.3: Gray scales with varying levels of graduation.

Our final step is to label the bar. This will be a bit tricky. Let's proceed as always with chewable bite sizes. The first task will be to simply add ticks in advance of numbers. We will use just three ticks: bottom, middle, and top. Of course, the number of ticks will be parametric so the user can adjust that whenever he wishes.

```
(* number of ticks; distance between ticks *)
nt = 3; Δ = 1 / (nt - 1);
(* tick marks half width *)
δ = w / 8;
```

We will use the graphics primitive `Line` to draw the tick marks. The ticks marks will be drawn at a height `h` and have a width `w` being drawn from `w - δ` to `w + δ`, hence the reason for δ being a half-width. We are going to encounter our first problem and it is easier to see against the simplest scale.

```
(* preserve the base image so we don't have to regenerate it *)
g01 = g00;
(* loop over all the ticks *)
Do[
  (* tick height *)
  h = (j - 1) Δ;
  (* draw tick *)
  gl = Graphics[Line[{{w - δ, h}, {w + δ, h}}]];
  (* place tick on graphics *)
  g01 = Show[g01, gl, doff];
  , {j, nt}];
(* use handle g02 to store correct aspect ratio version *)
g02 = Show[g01, AspectRatio → 1 / w, ImageSize → 0.25 72, don];
```



We present this graphic in a larger size to show an issue with the tick marks whose finite thickness extends beyond the border of the graphic. This is because, as stated, the lines have finite width. We need to outline the bar and the discontinuity will vanish.

```
(* the are four corners to the bounding box *)
(* but we need to repeat the first point to close the figure *)
verts = {{0, 0}, {w, 0}, {w, 1}, {0, 1}, {0, 0}};
(* draw the line around the shaded bar *)
fix = Graphics[Line[verts]];
```

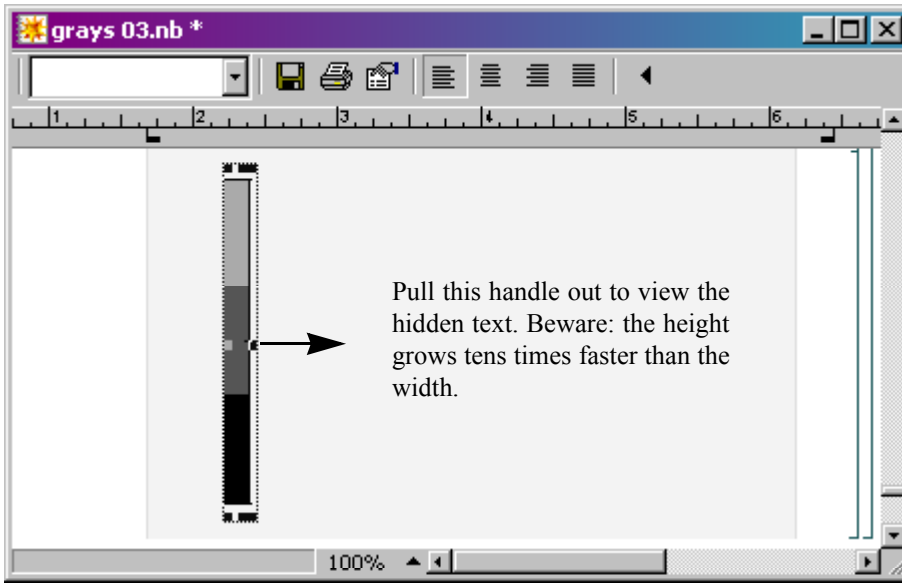
Now we blend the bar and the box and ticks into a single graphics object.

```
(* create a shaded bar ready for text *)
g03 = Show[g02, fix];
```



So far, so good. We will attempt to finish the job by labeling the ticks, herewith {0, 1/2, 1}. This will require some effort. Reusing the same do-loop we used to draw the tick marks, we will add the numbers.

```
(* preserve the base image so we don't have to regenerate it *)
g04 = g03;
(* loop over all the ticks *)
Do[
  (* tick height *)
  h = (j - 1) Δ;
  (* add text *)
  gt = Graphics[Text[h, {w + 2 δ, h}, {-1, 0}]];
  (* place tick and text on graphics *)
  g01 = Show[g01, gt, doff];
  , {j, nt}];
(* use handle g02 to store correct aspect ratio version *)
g04 = Show[g04, AspectRatio → 1 / w, ImageSize → 0.25 72, don];
```



⚠ Adding text to a graphic will not increase the graphic size. You must somehow expand the figure if you plan to add text.

3.1.4 Examples using graphics primitives: disks

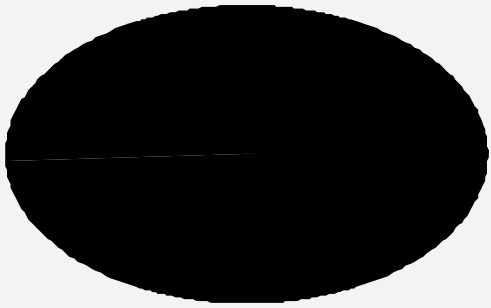
First, we make a comment on the precision of the *Mathematica* syntax. A circle is a one dimensional object that lives in a two-dimensional space. We say one dimensional in the usual context: the locus of points is mapped using a single parameter, in this case an angle. The disk is solid and all points can be mapped using the two parameters of radial distance and angle. More simplistically, a circle is like a hoop and a disk is like a coin. Of course in general usage, the word circle is often used where the word disk should be. With that resolved, we will now draw a disk.

The basic syntax is `Disk[{x,y},r]` where the italicized characters are the input values for the origin and radius. Let's draw a unit disk.

```
(* your basic disk *)
Graphics[Disk[{0, 0}, 1]];
```

So where is the disk? It has been rendered, but not displayed; it is in memory. The tag `-Graphics-` tells us that a graphics object has been created. To view a graphics object, we use the `Show` command and the `%` operator to point to the previous result.

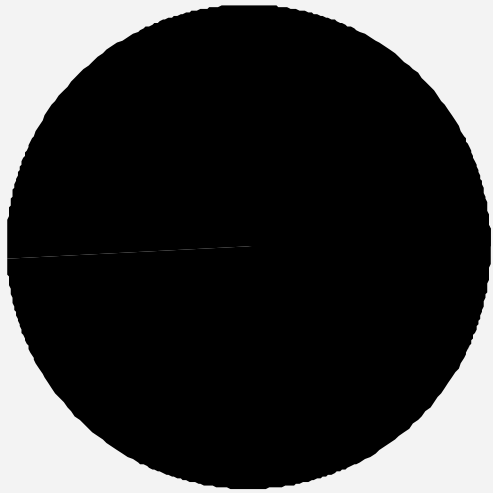
```
Show[%]
```



```
- Graphics -
```

This certainly doesn't look like a circular object; the aspect ratio is wrong. The aspect ratio for a graphic is the ratio of the height to the width. As explained in the help for the option **AspectRatio**, the default setting for the aspect ratio is Golden Ratio $\phi = (1 + \sqrt{5})/2$. In general, there is evidence of considerable ingenuity in the presentation of graphics in *Mathematica*. There are some excellent algorithms to decide how to label axes, for instance. But while these routines are quite good, in general, we will occasionally need to make adjustments. In this case we will simply let *Mathematica* determine a better aspect ratio by setting the option **AspectRatioÆAutomatic**. Also, at this time we will assign a pointer to the graphics object by naming it **g01** for "graphics object 01". Finally, we will suppress the tag by using a semicolon suffix.

```
(* assign a container for the disk *)
g01 = Graphics[Disk[{0, 0}, 1]];
Show[g01, AspectRatio → Automatic];
```



You will notice that the output tag is suppressed. We now have a graphic object in `g01` which we can modify, combine with other graphics objects, or write to a file. Let's consider creating an EPS file. There are two command options available, Export and `Display`. For now we will chose the `Display` command which has the basic syntax `Display[filename, handle, format]`. This is a little different format from what you will see in the help file; the syntax we are using is a bit more specific and we hope simpler.

```
Display[dirGraph <> "disk.eps", g01, "EPSI"];
```

This creates an EPS file named `disk.eps` in the directory `dirGraph`. The <> operator is used to concatenate two string variables. If you don't specify a directory, your graphic will be created in the default directory.

```
Display["disk.eps", g01, "EPSI"];
```

This creates an EPS file named `disk.eps` in the current directory. (To see which directory you are pointed to use the *Mathematica* command `Directory[]`.) We discourage this process because mixing notebooks, graphics files, and data files is a poor practice.

We could have changed the aspect ratio when we generated the graphic. That is we could have used:

```
g01 = Graphics[Disk[{0, 0}, 1], AspectRatio → Automatic];
Show[%];
```

and produced the exact same image on the screen. However, we note a significant difference. In the initial case, the graphics object we generated was stretched after creation by changing the aspect ratio. The output file for this graphic contains the elliptic object we first generated. This means that when the object is embedded in a text-processing application, it must be scaled in that application. This is undesirable; we should create graphics that are ready for immediate use. We should embed the command for the desired aspect ratio in the command which generates the graphics.

So why present an example of something to avoid? It is to make the point that we have a rich tool kit available to us which allows us to manipulate graphics objects after they have been created. If we are working with extremely intricate objects, we may save time by not rerendering them and using the `Show` command to do the manipulation.

The Mathematica Book > Principles of Mathematica > The Structure of Graphics and Sound > Advanced Topic: Low Level Graphics Rendering

3.1.5 Examples using graphics primitives: color wheels

Let's return to the discussion of color palettes and show how the RGB and CMYK palettes combine. We will start by overlapping three circles, each shaded with a primary color from the RGB palette. For visual appeal, we want the origins of the three circles to form an isosceles triangle. So by definition the offset angle will be $\pi/3$ and we need only to choose the size of the offset. We arbitrarily chose a value of 125% of the circle radius. We will choose one circle to be the anchor and define the other two circle locations relative to the anchor. A natural selection would be to put the anchor circle origin at $\{0, 0\}$ and assign unit radius.

The other two circles are then offset from the anchor origin by a distance of $\left\{\pm\sin\frac{\theta}{2},\ \cos\frac{\theta}{2}\right\}$. We will attempt to draw these circles.

```
(* define the circles *)
ρ = 1;              (* circle radii *)
l = 1.25;           (* offset length *)
o = {0, 0};         (* anchor circle origin *)
θ = π / 3;          (* offset angle *)
```

We have now specified the origins for all three circles. Even better, we have expressed them in terms of an offset from the anchor circle. These parameters specify the system completely and all other information will be derived from this set of numbers.

Down below, we take the above numbers and use them to locate the circles in space and to define the circles algebraically.

```
(* offset vector *)
v = l {Sin[θ / 2], Cos[θ / 2]};
(* locate the circle centers c and count them *)
c = {o, o + {-1, 1} v, o + {1, 1} v};
nc = Length[c];
(* define the circles *)
radii = Table[(x - c[[i]][[1]])^2 + (y - c[[i]][[2]])^2, {i, nc}];
(* generate color directives in RGB space *)
r = {1, 0, 0}; g = {0, 1, 0}; b = {0, 0, 1};
(* generate color directives in CMYK space *)
c = {1, 0, 0, 0}; m = {0, 1, 0, 0}; y = {0, 0, 1, 0};
(* aspect ratio is height / width *)
aspct = AspectRatio → (2 ρ + Cos[θ / 2] l) / (2 ρ + l);
```

Let's look at equations which will be used to define the circles.

```
(* let's look at the algebraic expressions *)
radii
```

```
{x^2 + y^2, (0.625 + x)^2 + (-1.08253 + y)^2, (-0.625 + x)^2 + (-1.08253 + y)^2}
```

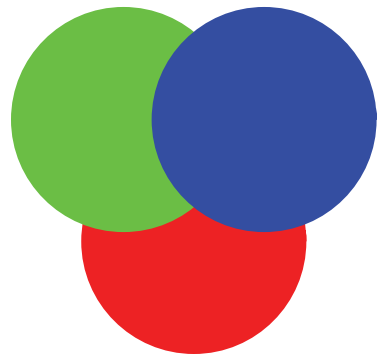
For example, let's take the case of the second element and show how this is used to define a lamina.

```
(* generate a lamina *)
eq1 = radii[[2]] ≤ ρ

(0.625 + x)^2 + (-1.08253 + y)^2 ≤ 1
```

For now we are able to draw the circles because they have an origin and a radius. The algebraic forms will be needed to determine overlap regions.

```
(* draw the circles *)
(* create a table of graphics primitives *)
prim =
   Table[{RGBColor[r[[i]], g[[i]], b[[i]]], Disk[c[[i]], ρ]}, {i, nc}];
 (* convert primitives to graphics *)
 gc = Graphics[prim];
 (* show and stretch *)
 gx = Show[gc, aspct];
```

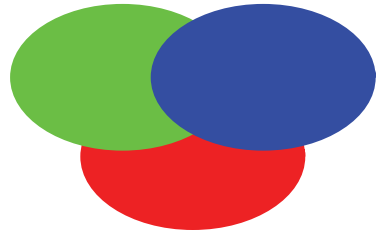


When working problems like this one it is always wise to be explicit. Let's number the circles so that we can print the graphic to have in front of us for debugging.

We see after producing the plot that the numbers were almost too small to read, so we went to the Help Browser to read about `Text` and that referred us to an example in `ListPlot` (under Further Examples) which showed us the format we needed.

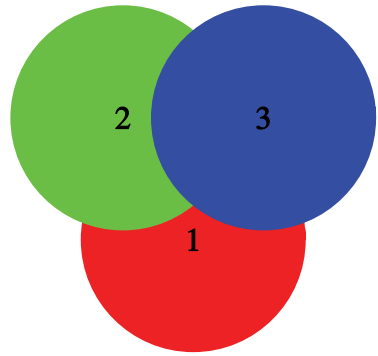
- Built-in Functions > Graphics and Sound > Graphics Primitives > Text
- Built-in Functions > Graphics and Sound > 2D Plots > ListPlot

```
(* create a table of text primitives to number the circles *)
txt = Table[Text[i, c[[i]], TextStyle → {FontSize → 24}], {i, nc}];
(* convert primitives to graphics *)
gt01 = Graphics[txt];
(* draw disks, add numbers *)
g00 = Show[gt01, gx];
```



Our aspect ratio is wrong even though we clearly specified it in the command `gx = Show[gc,aspct]`. The problem is the order that we combine the graphics. The first graphic we pull in sets the properties for the composite object. You can say that the properties for the composite object are inherited from the first object. The correction for this is quite simple: we just have to load the graphics object `gx` first. This will also bring the numbers to the top of the graphic.

```
(* draw disks, add numbers; aspect ratio is inherited *)
g00 = Show[gx, gt01];
```



You could have specified the aspect ratio in the graphics object for the text characters, `gt`. We prefer to keep the aspect ratio in the assignment `gx` which is the source of the aspect information.

Notice how we created a list of graphics objects; the circles were created in a list. First, we drew three circles in a simple fashion. Then we numbered the circles with a list of text graphics objects. We used the simplest form of the `Text` command: `Text[value, location]`. Now that we have the diagram, we can see how to shade the overlap regions.

Mathematica will not combine colors for us; this is a simple operation we must do ourselves. Look at what happens when we draw the disks. Each disk can be considered opaque; you can tell which order they were drawn by looking at the exposed colors.

So the graphics engine will not combine colors for us, but it allows us to deal with color algebraically. This will be extremely helpful when we want to color our 3D plots. In this case, we note that there are four overlap regions: three "petals" and a central spot. The three petals are defined by the following unions:

add colors 1 and 2	$1 \cap 2 \cap \bar{3}$	intersect 1 and 2, exclude 3
add colors 1 and 3	$1 \cap 3 \cap \bar{2}$	intersect 1 and 3, exclude 2
add colors 2 and 3	$2 \cap 3 \cap \bar{1}$	intersect 2 and 3, exclude 1

The central spot is the intersection of all three sections, $1 \cap 2 \cap 3$, and combines all three colors. Notice that we are not defining colors in the overlap regions; we are adding colors. This methodology will allow us to easily switch over to another color scheme like CMYK.

First, we need to define the overlap regions. While *Mathematica* is well suited to solve for these laminae algebraically, it allows an easier method using the inequalities package. The template for the command we will use is:

```
InequalityPlot[And[condtion1, condition2, condition3] ,
 {x}, {y}, Fills → RGBColor[r, g, b]]
```

We see that we generate some logical conditions for the `Inequality` plot. These will closely match the intersections listed above. Fortunately, we do not need to specify ranges for *x* and *y* so we can syntax **{x}** and **{y}**. Finally, the option **Fills** defines what color the lamina will be. This is where we will add colors in accordance with logical conditions. Our first step is to load **InequalityGraphics** package and then create each overlap region.

```
(* load the package for the inequality plots *)
<< Graphics`InequalityGraphics`;
```

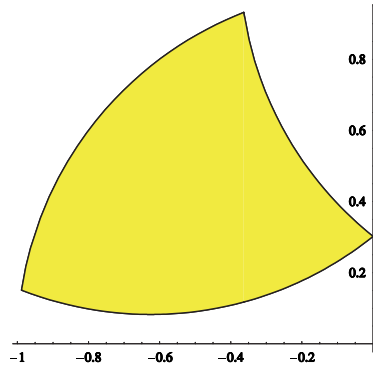
Keeping in *Mathematica*'s spirit of generality, we will write a module that looks at the three "petals" and determines what color they should be.

```
(* prepares an overlap zone *)
(* i ⋂ j ⋂ k̄ *)
prep[i_Integer, j_Integer, k_Integer] := Module[{},
  (* algebraic description of overlap area *)
  ineq = And[ radii[[i]] ≤ ρ, radii[[j]] ≤ ρ, radii[[k]] ≥ ρ];
  (* fill color is the sum of the RGB colors for i and j *)
  fill = {Fills → RGBColor[r[[i]] + r[[j]], g[[i]] + g[[j]], b[[i]] + b[[j]], doff};
  (* create the inequality plot *)
  g0 = InequalityPlot[ineq, {x}, {y}, fill];
  (* add latest graphic to container *)
  AppendTo[laminae, g0];
 ];
```

The premise of this routine is that we look at the intersection of two regions and exclude the third. This is a very basic routine. When we call the routine, we will cycle through the three circles to call this routine properly. The fill color is determined by adding the colors of the two overlap zones. With the petal shape and location determined by the algebraic equations and the color determined by summing the RGB directives, we are ready to call the `InequalityPlot` routine. As we create a shaded petal, we add it to the container lamina. Later on we see that we will be able to use this list object as if it were a single graphic and this will relieve us the burden of having to specify the names of the objects.

We never think it is a good idea to write a bunch of code and then see whether it works. You should test as you go and exploit one of the great features of the interactive *Mathematica* environment. In this vein, we will generate a test petal.

```
(* test run *)
(* clear the graphics container *)
laminae = {};
(* intersection of circles 1 and 2, exclusion of 3 *)
prep[1, 2, 3];
(* view result *)
g0 = Show[laminae, don];
```



Apparently, red + blue = yellow. One beauty of this method is that we don't have to know anything about the adding colors; we let *Mathematica* handle that for us. Also, we are happy with the shape and location so we are eager to proceed.

```
(* region where all three circles overlap *)
all = And[ radii[[1]] ≤ ρ, radii[[2]] ≤ ρ, radii[[3]] ≤ ρ];
(* fill color is the sum of RGB colors for each disk *)
fill = Fills → RGBColor[∑_{i=1}^{nc} r[[i]], ∑_{i=1}^{nc} g[[i]], ∑_{i=1}^{nc} b[[i]]];
(* graphics container is seeded with all *)
laminae = {InequalityPlot[all , {x}, {y}, fill, doff]};
(* sweep through all three remaining overlap regions *)
prep[1, 2, 3];
prep[2, 3, 1];
prep[3, 1, 2];
```

The first task is to handle the central region where all three disks intersect. This is the only case where this happens, and so we handle it explicitly. Notice how easy it is to read the state beginning with `all =`. We are looking for the intersection of all three disks. We don't need to specify them because they are in the list `radii`. Next, we compute the color of this region by combining the RGB directives of all three regions. With this information, we are able to ask `InequalityPlot` to create the lamina for us and this lamina will be the first element of `laminae`. Notice too how we use a simple cyclic permutation to pass the circle lists to the `prep` command.

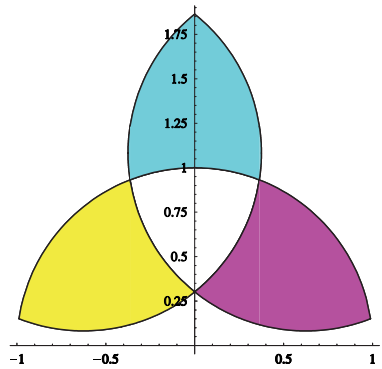
It is prudent to examine the intermediate products before moving forward. For the sake of familiarization we query the contents of laminae. We are looking for the central region and the three petals.

```
(* what is in the graphics container? Four graphics *)
laminae

{-Graphics-, -Graphics-, -Graphics-, -Graphics-}
```

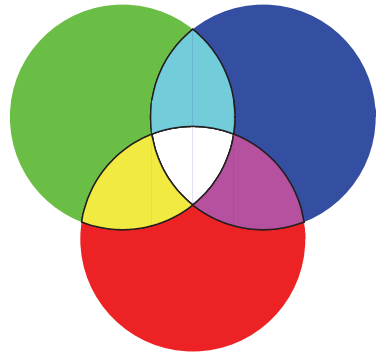
We find four `Graphics` objects which is reassuring. Let's view them.

```
(* what do all four overlap regions look like?  *)
g0 = Show[laminae, don];
```



The anxious reader may be concerned by the axes displayed here since we did not bother to suppress them with `Axes->False`. This was deliberate. We want the reader to think consciously about how the final graphic will inherit its properties. We plan to make sure the foundation graphic (the first graphic in the `Show` list) does have the property `Axes>False`.

```
(* bring all the pieces together *)
(* insure loading order preserved *)
(* proper aspect ratio and lask of axes *)
gf = Show[gx, laminae];
```



This simple chart shows us that red + green = yellow, green + blue = cyan, red + blue = magenta, and red + blue + green = white.

In the spirit of generality, let's ask ourselves how to create the color wheels for the CMYK colors. We want to recreate the algorithm so that it can use either color. We begin by looking at the simple color relationships from above in terms of vectors.

red + green = yellow	{1,0,0} + {0,1,0} = {1,1,0} = yellow
green + blue = cyan	{0,1,0} + {0,0,1} = {0,1,1} = cyan
red + blue = magenta	{1,0,0} + {0,0,1} = {1,0,1} = magenta

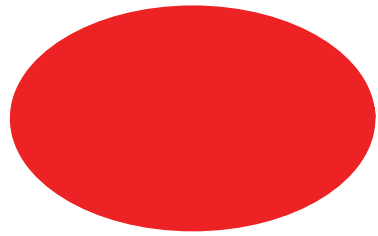
Certainly in these cases, the addition is straightforward. But we need a color addition scheme that can add colors from either species: RGB or CMYK.

To do this we will need to dig into the *Mathematica* tool bag. If we are going to represent colors like vectors as above, we need to be able to use these lists as arguments for the color directives. Let's define the colors as vectors and then use some of them for examples.

```
(* generate color directives in RGB space *)
r = {1, 0, 0}; g = {0, 1, 0}; b = {0, 0, 1};
(* generate color directives in CMYK space *)
c = {1, 0, 0, 0}; m = {0, 1, 0, 0}; y = {0, 0, 1, 0};
```

We will draw a disk and color it red. Notice the syntax on the RGBColor command.

```
(* the RGB vector is not a list *)
g00 = Disk[{0, 0}, 1];   (* disk of radius 1, centered at origin *)
g01 = Graphics[{RGBColor[1, 0, 0], g00}];  (* color the disk *)
Show[g01];
```



Watch what happened when we pass the vector *r* to `RGBColor`. We get a succinct error message and the disk is not colored.

```
(* pass the vector r *)
g00 = Disk[{0, 0}, 1];   (* disk of radius 1, centered at origin *)
g01 = Graphics[{RGBColor[r], g00}];  (* color the disk *)
Show[g01];

⚠ RGBColor ::argr :
  RGBColor called with 1 argument ; 3 arguments are expected . More...
```

What we are being told is this: we are using the wrong syntax.

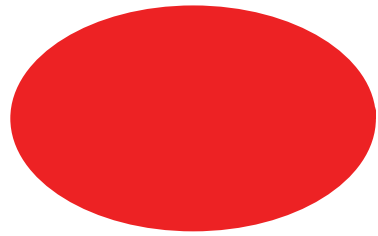
`RGBColor[1, 0, 0]`	good
`RGBColor[{1, 0, 0}]`	bad

How do we tell *Mathematica* to interpret the list as the arguments? We use the `Sequence` command which will splice the list into a sequence of arguments using the `Apply` operator `@@`. More examples with `Sequence` will be shown later. For now, we will see that:

`RGBColor[Sequence@@{1, 0, 0}]` ⟹ `RGBColor[1, 0, 0]`

This simple statement will allow us to pass color vectors to both the RGB and CMYK directives.

```
(* use Sequence to splice the vector r into RGBColor arguments *)
g00 = Disk[{0, 0}, 1];  (* disk of radius 1, centered at origin *)
g01 = Graphics[{RGBColor[Sequence @@ r], g00}];  (* color the disk *)
Show[g01];
```



The next step in our generalization process is to create a command that can look at a list and turn it into either an RGB or CMYK directive based upon the length. We wrote a function to make a directive from a list.

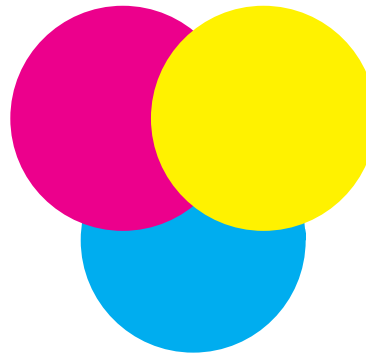
```
(* make a color directive from a list *)
makedir[a_List] := Module[{n},
   (* determine the length of the color vector *)
   n = Length[a];
   (* RGB lists have 3 terms; CMYK lists have 4 terms *)
   If[n == 3, Return[RGBColor[Sequence @@ a]],
    Return[CMYKColor[Sequence @@ a]]]
   (* Return the color directive so makedir is like a function *)
  ];
```

Let's go back to the command that drew the three colored circles and show how `makedir` has liberated us from specifying the color species. First, we define the colors for the circles.

```
(* define colors for CMYK space *)
color = {c, m, y};
```

Next, we will apply the colors in this list to the three circles.

```
(* draw the circles *)
(* create a table of graphics primitives using makedir *)
prim =
  Table[{makedir[color[[i]]], Disk[c[[i]], ρ]}, {i, nc}];
(* convert primitives to graphics *)
gc = Graphics[prim];
(* show and stretch *)
gx = Show[gc, aspct];
```



The next component is a module to add colors. This is quite simple because of the vector form of the colors. The module `addcolor` will take a list, add the vectors, and return the appropriate color directive. A key element of this process is creating a zero vector to act as an accumulator. We covered this in the first chapter, and we will emphasize it here again.

The point is to create a zero vector of length three or length four. This is trivial in *Mathematica*: we simply subtract a vector from itself. Look at these examples.

```
r - r = {1, 0, 0} - {1, 0, 0} = {0, 0, 0}
c - c = {1, 0, 0, 0} - {1, 0, 0, 0} = {0, 0, 0, 0}
```

This is needed because we will use a do-loop to add the color values and we need to start with a zero of proper dimension, three or four.

```
(* module to add a list of colors in the same color space *)
(* and return a color directive *)
addcolor[a_List] := Module[{n, m, scolor},
   (* how many different regions will be combined? *)
   n = Length[a];
   (* create a zero vector for a summation variable *)
   scolor = a[[1]] - a[[1]];  (* vector 0 *)
   (* loop through the regions *)
   Do[
    (* add the color for region i *)
    scolor += a[[i]];
    , {i, n}];
   (* determine RGB or CMYK space *)
   m = Length[scolor];
   (* form the color directive *)
   coldir = If[m == 3,
     RGBColor[Sequence @@ scolor], CMYKColor[Sequence @@ scolor]];
   (* return the color directive *)
   Return[coldir];
  ];
```

As we have said often and will continue to say, it is better to take advantage of *Mathematica*'s extensive list tools. For example, we can replace the code:

```
(* how many different regions will be combined? *)
n = Length[a];
(* create a zero vector for a summation variable *)
scolor = a[[1]] - a[[1]];  (* vector 0 *)
(* loop through the regions *)
Do[
 (* add the color for region i *)
 scolor += a[[i]];
 , {i, n}];
```

with

```
scolor = Plus @@ a;
```

But as always, we encourage the reader to use the paradigm that is the easiest for him.

The module **prep** needs updated too. Specifically the lines:

```
(* fill color is the sum of the RGB colors for i and j *)
fill = {Fills → RGBColor[r[[i]] + r[[j]], g[[i]] + g[[j]], b[[i]] + b[[j]]], doff};
```

must be replaced with:

```
(* fill color is the sum of the RGB colors for i and j *)
fill = {Fills → addcolor[{color[[i]], color[[j]]}], doff};
```

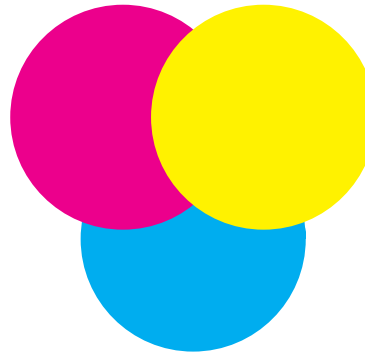
Finally, in the code that orchestrates the assembly process of the disks and the laminae, we must replace the line:

```
(* fill color is the sum of RGB colors for each disk *)
fill = Fills → RGBColor[∑_{i=1}^{nc} r[[i]], ∑_{i=1}^{nc} g[[i]], ∑_{i=1}^{nc} b[[i]]];
```

with:

```
(* fill color is the sum of RGB colors for each disk *)
fill = Fills → makedir[Plus @@ color];
```

With these changes we can draw either the RGB or CMYK color wheels simply by changing the contents of the list `color`. When we look at the CMYK wheels, we see:



In *Mathematica* we find the relationships cyan + magenta = blue, cyan + yellow = red, and yellow + cyan = green. We also see that the RGB colors are additive since they sum to white, and the CMYK colors are subtractive because they sum to black.

3.1.6 Lessons learned from the primitives

We started with the graphics primitives because we felt such a simplified environment would foster a feeling of relaxation when dealing with *Mathematica*'s graphics tools. We have learned how to combine graphics objects, how to manipulate color and shading, how to suppress displays, how to adjust aspect ratios and how to use a few more commands like `Sequence` and `Apply`.

Our goal is to make the user eager to use the more general tools like `Plot`. Already the reader should have a feeling of comfort when learning the more general commands.

3.2 Plotting in two dimensions

The plotting capabilities of *Mathematica* are prodigious and we receive a great deal of attention in this book. We will start with the **Plot** command since it is the most intuitive. As mentioned before, one should always check the *Mathematica* book for a clear and succinct summary of the command. In the case of the `Plot` command, there are some particularly subtle demonstrations.

3.2.1 The Plot command

The `Plot` command is an excellent construct that has a good balance between simplicity and flexibility. As always, your first recourse to investigate a command should be the Help Browser.

```
Options[Plot]

{AspectRatio → 1/GoldenRatio, Axes → Automatic, AxesLabel → None,
 AxesOrigin → Automatic, AxesStyle → Automatic, Background → Automatic,
 ColorOutput → Automatic, Compiled → True, DefaultColor → Automatic,
 DefaultFont :→ $DefaultFont, DisplayFunction :→ $DisplayFunction,
 Epilog → {}, FormatType :→ $FormatType, Frame → False, FrameLabel → None,
 FrameStyle → Automatic, FrameTicks → Automatic, GridLines → None,
 ImageSize → Automatic, MaxBend → 10., PlotDivision → 30.,
 PlotLabel → None, PlotPoints → 25, PlotRange → Automatic,
 PlotRegion → Automatic, PlotStyle → Automatic, Prolog → {},
 RotateLabel → True, TextStyle :→ $TextStyle, Ticks → Automatic}
```

This represents a very powerful tool kit. We will detail the basic settings you are likely to need.

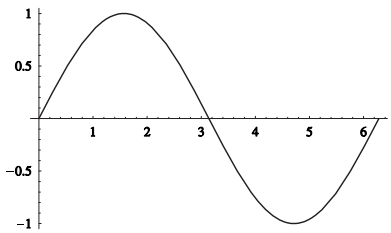
`AspectRatio`: this is the height divided by the width. For example, the sin*x* plot is 2 units tall and 2π units wide.
`Axes`: `True` (default) to display axes, **`False`** to suppress them.
`DisplayFunction`: can suppress the display of a graphic. Very helpful if you are creating complex graphics.
`Frame`: **`True`** to display a frame, `False` (default) to suppress a frame.
`FrameLabel`: when **`Frame`** is **`True`** this option controls the display of the labels for all four edges of the frame.
`FrameTicks`: controls labeling of the ticks on the frame.
`ImageSize`: controls the size of the graphic, measured in print points. Using *x* 72 will create a graphic with a width of *x* inches.
`PlotLabel`: labels the plot if `Frame` is `False`.
`PlotPoints`: controls the sampling density. Rapidly varying functions will demand a higher value for `PlotPoints`.
`PlotRange`: *Mathematica* has a very intelligent method of scaling plots, but at times you may want to override it.
`Ticks`: controls labeling of the ticks on the axes.

Again, the hypertext help is extremely helpful, especially for the basic commands above. We always prefer to print the Help Browser entries so that we may annotate them and refer to them while entering command syntax.

Let's start by plotting a complete period of the function sin*x*.

```
Plot[Sin[x], {x, 0, 2 π}]

- Graphics -
```



We see a very direct syntax that is easy to use. When the mouse button is clicked while the cursor is over the figure, *Mathematica* will display a bounding box with drag handles. This allows for an immediate resize of the figure. Later on we will recommend that plots sizes be standardized by using the `ImageSize` option. The next thing that we notice is that the output is a `-Graphics-` object. When are you more comfortable with the front end, you may append a semicolon to the plot command and suppress this output. We make two important observations about the graphics objects. First, they consume a great deal of memory and if you save the notebook with the graphics rendered, the file size will be much larger. Second, we should assign a handle to this object to allow us to use the rich graphics manipulation routines of *Mathematica*.

So a more typical plot command may look like:

```
g01 = Plot[Sin[x], {x, 0, 2 π}];
```

The output tag is suppressed and we now have a graphics object `p1` which we can store in a variety of graphics formats of further manipulation with *Mathematica*. But before we combine this with other graphics objects, we should examine several options for the `Plot` command. The *Mathematica* command `Option` can be used to list all available options for each command. When we enter `Options[Plot],` we will see the following.

Before showing some examples, we need to stress a subtle point. All commands are interpreted from left to right. Subsequent settings of the same option are ignored. The rules for suboptions are simple. By suboption, we mean an option like `FrameTicks` for the option `Frame`. The first rule is that suboptions do not activate the parent option. So setting the `FrameTicks` will not activate the `Frame`. Second, the suboptions can be set at any time; that is, they may be set either before or after the parent option is set within the command line.

3.2.2 List plots

Mathematica comes with a very effective offering of list plots: they correspond to all the continuum plot features.

Random walk

Let's revisit the random walk problem from chapter 1; this is a great example for the `ListPlot`. We will modify the `While` loop to capture the steps in a list we shall call `q`. Here is the modified loop.

```
(* the random walk problem *)
(* start at the origin *)
(* steps are unit size in a random direction *)
d = -∞;            (* total distance traveled: controls While *)
max = 5;           (* threshold distance to stop walking *)
k = 0;             (* count the steps *)
SeedRandom[1];     (* always seed for repeatability *)
p = {0, 0};        (* position vector *)
q = {p};           (* container for the steps *)
(* keep walking until we are 15 units away *)
While[d ≤ max,
 k++;                          (* increment step counter *)
 θ = Random[Real, {0, 2 π}];   (* direction in radians *)
 s = {Cos[θ], Sin[θ]};         (* this is the step *)
 p += s;                       (* take the step *)
 AppendTo[q, p];               (* save the step *)
 d = √(Plus @@ p^2);           (* distance from origin *)
];
(* output number of steps and how far we went *)
Print[k, " iterations to go a distance ", d];
```

```
11 iterations  to go a distance  5.2976
```

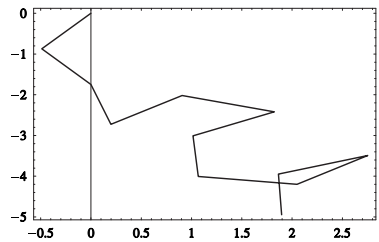
The gray sidebars on the left mark the modifications needed to get the loop to store the history of its steps which we show below.

```
(* a complete record of all the steps *)
q

{{0, 0}, {-0.488935, -0.87232}, {-0.000589038, -1.74497},
 {0.197935, -2.72507}, {0.906666, -2.01959}, {1.82326, -2.4194},
 {1.01518, -3.00847}, {1.06765, -4.00709}, {2.04884, -4.20009},
 {2.75284, -3.48988}, {1.86365, -3.94743}, {1.89539, -4.94693}}
```

Our first task will be to draw the steps. `ListPlot` is an ideal command for this when we realize that we can connect the plot points.

```
(* ListPlot the steps *)
g1 = ListPlot[q, PlotJoined → True, Frame → True];
```

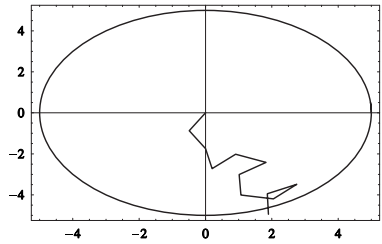


Pay close attention to how we draw the circle.

```
(* threshold circle *)
g2 = Circle[{0, 0}, max];
```

It is not a graphic yet until converted from a primitive using the `Graphics` command which we shall do when we combine the objects using `Show`.

```
(* combine the graphics *)
g3 = Show[g1, Graphics[g2]];
```

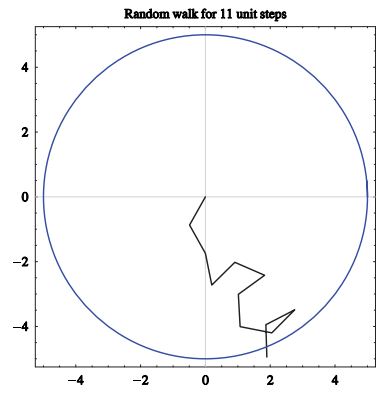


That was really quite simple. When we look at this plot, some improvements immediately spring to mind. First, we should lighten the axes. They are a nice reference, but they draw the eye away from the trajectory.

```
(* lighten the axes *)
g1 = ListPlot[q, PlotJoined → True,
   Frame → True, AxesStyle → GrayLevel[0.8]];
```

Next, we'll give the threshold circle a blue color to separate it more. Also, we will build a descriptive title telling how many steps were taken. Observe how we folded the step counter *k* into the title so that the plot label is automatic. Too often have we seen people manually label charts and then change the data making the title incorrect. Let *Mathematica* take care of that for us. Finally, we want an aspect ratio that makes the circle look round and *Mathematica* is happy to do that also.

```
(* add a descriptive title and use the proper aspect ratio *)
g4 = Show[g1, Graphics[{cblu, g2}], AspectRatio → Automatic,
   PlotLabel → "Random walk for " <> ToString[k] <> " unit steps"];
```



Plotting imaginary roots
Consider the equation:

$$x^8 + 1 = 0 \tag{3.1}$$

which we realize will have eight imaginary roots. How might we find these roots and the plot them?

The solution is easy with *Mathematica*'s `Solve`. First, we define the equation.

```
(* equation with complex roots *)
eq = x^8 + 1 == 0

1 + x^8 == 0
```

Then we `Solve` it.

```
(* find the solution *)
soln = Solve[eq, x]

{{x → -(-1)^1/8}, {x → (-1)^1/8}, {x → -(-1)^3/8}, {x → (-1)^3/8},
 {x → -(-1)^5/8}, {x → (-1)^5/8}, {x → -(-1)^7/8}, {x → (-1)^7/8}}
```

To be sure, each of these roots is complex and we can make the complex form manifest with `ComplexExpand`.

```
(* express the roots in complex form *)
ComplexExpand[(-1)^1/8]

Cos[π/8] + i Sin[π/8]
```

So how do we apply this command to all of the roots? Let's try a substitution rule that we will apply to the substitution rules for the roots. In other words, we want to cast a substitution rule to apply to all the substitution rules like $x \longrightarrow (-1)^{3/8}$.

The naive attempt fails.

```
(* first attempt at substitution rule *)
subrule = {(x_ → y_) → (x → ComplexExpand[y])}

{(x_ → y_) → x → y}
```

We need to use the `RuleDelayed` operator. This will hold the evaluation of the right-hand side until after the rule is applied. This fixes the substitution rule `subrule`.

Built-in Functions > Programming > Rule Application > Rule Delayed (`:>`)

```
(* substitution rule to operate on the roots *)
(* swap x → root with x → ComplexExpand[root] *)
subrule = {(x_ → y_) :> (x → ComplexExpand[y])}

{(x_ → y_) :> x → ComplexExpand[y]}
```

Now we can apply `subrule` to `soln` which is itself a collection of substitution rules.

```
(* view roots in complex form *)
csoln = soln /. subrule
```

$$\left\{\left\{x \to -\text{Cos}\left[\frac{\pi}{8}\right] - i\,\text{Sin}\left[\frac{\pi}{8}\right]\right\}, \left\{x \to \text{Cos}\left[\frac{\pi}{8}\right] + i\,\text{Sin}\left[\frac{\pi}{8}\right]\right\},\right.$$
$$\left\{x \to -i\,\text{Cos}\left[\frac{\pi}{8}\right] - \text{Sin}\left[\frac{\pi}{8}\right]\right\}, \left\{x \to i\,\text{Cos}\left[\frac{\pi}{8}\right] + \text{Sin}\left[\frac{\pi}{8}\right]\right\},$$
$$\left\{x \to -i\,\text{Cos}\left[\frac{\pi}{8}\right] + \text{Sin}\left[\frac{\pi}{8}\right]\right\}, \left\{x \to i\,\text{Cos}\left[\frac{\pi}{8}\right] - \text{Sin}\left[\frac{\pi}{8}\right]\right\},$$
$$\left.\left\{x \to \text{Cos}\left[\frac{\pi}{8}\right] - i\,\text{Sin}\left[\frac{\pi}{8}\right]\right\}, \left\{x \to -\text{Cos}\left[\frac{\pi}{8}\right] + i\,\text{Sin}\left[\frac{\pi}{8}\right]\right\}\right\}$$

At last, all the roots can be viewed. We need to take these roots and turn them into plottable coordinates.

Built-in Functions > Algebraic Computation > Equation Solving > Solve > Further Examples > Polynomial Equations in One Variable

We will treat these roots just as we did in school and plot them in the imaginary plane with coordinates `{Re[z], Im[z]}`. Three paths will be shown to collect the plot data that we need.

The most basic way to recover these points is with a `Do` loop. If that is what you are comfortable with, we encourage you to keep using it. Over time, you should accli-

mate to the more powerful tools. When we use a do-loop to build a list, we are using this basic structure.

```
(* Do loop schematic for building a list *)
n = Length[list];  (* count the iterations *)
container = {};     (* empty container to hold list *)
Do[                (* loop through all n cases *)
 calculations     (* compute results for the list *)
   AppendTo[container, results]; (* add results to the list *)
 , {i, n}];
```

Now let's fill out the template.

```
(* how many roots did we find? *)
nr = Length[csoln];
(* empty container for the plot points *)
pts = {};
(* loop through all the roots *)
Do[
  (* grab a specific root *)
  z = x /. csoln[[i]];
  (* compute the plot coordinates *)
  plotpt = {Re[z], Im[z]};
  (* collect the real and imaginary components *)
  AppendTo[pts, plotpt];
  , {i, nr}];
(* review data *)
pts
```

$$\left\{\left\{-\text{Cos}\left[\frac{\pi}{8}\right], -\text{Sin}\left[\frac{\pi}{8}\right]\right\}, \left\{\text{Cos}\left[\frac{\pi}{8}\right], \text{Sin}\left[\frac{\pi}{8}\right]\right\},\right.$$
$$\left\{-\text{Sin}\left[\frac{\pi}{8}\right], -\text{Cos}\left[\frac{\pi}{8}\right]\right\}, \left\{\text{Sin}\left[\frac{\pi}{8}\right], \text{Cos}\left[\frac{\pi}{8}\right]\right\}, \left\{\text{Sin}\left[\frac{\pi}{8}\right], -\text{Cos}\left[\frac{\pi}{8}\right]\right\},$$
$$\left.\left\{-\text{Sin}\left[\frac{\pi}{8}\right], \text{Cos}\left[\frac{\pi}{8}\right]\right\}, \left\{\text{Cos}\left[\frac{\pi}{8}\right], -\text{Sin}\left[\frac{\pi}{8}\right]\right\}, \left\{-\text{Cos}\left[\frac{\pi}{8}\right], \text{Sin}\left[\frac{\pi}{8}\right]\right\}\right\}$$

If you are brand-new to *Mathematica*, then you should study these pieces carefully. When in doubt, take advantage of the interactive environment and study the individual commands you are using. Let's start with the root extraction substitution rule.

```
(* display the substitution rule *)
csoln[[1]]
(* sample root extraction *)
z = x /. csoln[[1]]
```

$$\left\{x \to -\text{Cos}\left[\frac{\pi}{8}\right] - i\,\text{Sin}\left[\frac{\pi}{8}\right]\right\}$$

$$-\text{Cos}\left[\frac{\pi}{8}\right] - i\,\text{Sin}\left[\frac{\pi}{8}\right]$$

This is the core process and if you are not comfortable with what is happening, you should grab the *Mathematica* code and experiment all the while consulting with the Help Browser.

How is this root converted into plottable coordinates? Using the `Real` and `Imaginary` operators.

```
(* plot coordinates *)
plotpt = {Re[z], Im[z]}
```

$$\left\{-\text{Cos}\left[\frac{\pi}{8}\right], -\text{Sin}\left[\frac{\pi}{8}\right]\right\}$$

The next step is to add this list of plot points. We choose the `AppendTo` because it is more compact than `Append`.

```
(* sample list building *)
AppendTo[pts, plotpt]
```

$$\left\{\left\{-\text{Cos}\left[\frac{\pi}{8}\right], \text{Sin}\left[\frac{\pi}{8}\right]\right\}\right\}$$

As we alluded to, we could use the `Append` command. Notice that this requires the use of the `Set` command — an equals sign.

```
(* another method using Append in lieu of AppendTo *)
pts = Append[pts, plotpt]
```

$$\left\{\left\{-\text{Cos}\left[\frac{\pi}{8}\right], \text{Sin}\left[\frac{\pi}{8}\right]\right\}\right\}$$

With either syntax we see that we have added a list of two elements to the list of plot points.

A better way to build the list of plot points is to use the `Table` command which automatically constructs a list for us, relieving us from the need to create an empty list and having to manually add each element. Notice how compact this fragment is.

```
(* how many roots did we find? *)
nr = Length[csoln];
(* build the list of plot points *)
pts = Table[z = x /. csoln[[i]]; {Re[z], Im[z]}, {i, nr}]
```

$$\left\{\left\{-\text{Cos}\left[\frac{\pi}{8}\right], -\text{Sin}\left[\frac{\pi}{8}\right]\right\}, \left\{\text{Cos}\left[\frac{\pi}{8}\right], \text{Sin}\left[\frac{\pi}{8}\right]\right\},\right.$$
$$\left\{-\text{Sin}\left[\frac{\pi}{8}\right], -\text{Cos}\left[\frac{\pi}{8}\right]\right\}, \left\{\text{Sin}\left[\frac{\pi}{8}\right], \text{Cos}\left[\frac{\pi}{8}\right]\right\}, \left\{\text{Sin}\left[\frac{\pi}{8}\right], -\text{Cos}\left[\frac{\pi}{8}\right]\right\},$$
$$\left.\left\{-\text{Sin}\left[\frac{\pi}{8}\right], \text{Cos}\left[\frac{\pi}{8}\right]\right\}, \left\{\text{Cos}\left[\frac{\pi}{8}\right], -\text{Sin}\left[\frac{\pi}{8}\right]\right\}, \left\{-\text{Cos}\left[\frac{\pi}{8}\right], \text{Sin}\left[\frac{\pi}{8}\right]\right\}\right\}$$

Notice that in the `Table` command we are generating the complex number *z*. This *z* is not displayed because the substitution rule is terminated with a colon. So we have *z* in memory for the **Real** and **Imaginary** commands to process. This is a very nice syntax.

But we look at the Length command and we can tell that we are not thinking in vector terms yet. After all, when you compute a dot product, do you first have to specify the size of the vectors? Of course not; you realize that the answer will have the same dimension as the input vectors. Let us work towards a vector paradigm that liberates us of this elephantine measuring.

We turn to the pure function. When we map a pure function across an array, it processes every element. Instead of saying "perform this process n times" we merely say "perform this process on every member". Here is the candidate function to compute the address of a plot point.

```
(* pure function to convert roots into real plot values *)
addr = (z = x /. #; {Re[z], Im[z]}) &;
```

The slot operator `#` marks the position that the array elements will come through. Compare this function to the `Table` command.

We can apply this function to every element in the list using the `Map` command (`/@`).

```
(* generate the plot values *)
pts = addr /@ csoln
```

$$\left\{\left\{-\text{Cos}\left[\frac{\pi}{8}\right], -\text{Sin}\left[\frac{\pi}{8}\right]\right\}, \left\{\text{Cos}\left[\frac{\pi}{8}\right], \text{Sin}\left[\frac{\pi}{8}\right]\right\},\right.$$
$$\left\{-\text{Sin}\left[\frac{\pi}{8}\right], -\text{Cos}\left[\frac{\pi}{8}\right]\right\}, \left\{\text{Sin}\left[\frac{\pi}{8}\right], \text{Cos}\left[\frac{\pi}{8}\right]\right\}, \left\{\text{Sin}\left[\frac{\pi}{8}\right], -\text{Cos}\left[\frac{\pi}{8}\right]\right\},$$
$$\left.\left\{-\text{Sin}\left[\frac{\pi}{8}\right], \text{Cos}\left[\frac{\pi}{8}\right]\right\}, \left\{\text{Cos}\left[\frac{\pi}{8}\right], -\text{Sin}\left[\frac{\pi}{8}\right]\right\}, \left\{-\text{Cos}\left[\frac{\pi}{8}\right], \text{Sin}\left[\frac{\pi}{8}\right]\right\}\right\}$$

We understand that many users need to see concepts like pure functions and mapping in a few different contexts before they can assimilate them. So do not worry if this seems a bit nebulous. We will revisit these important concepts several more times.

In closing, we address the concerns of the serious reader who is wondering why we call the pure function method the most compact since it and the `Table` method required two separate lines of code. The truth is that we can collapse everything into one line.

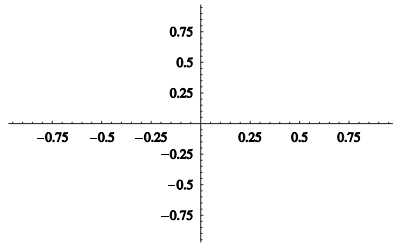
```
(* ultra compact form *)
pts = (z = x /. #; {Re[z], Im[z]}) & /@ csoln
```

$$\left\{\left\{-\text{Cos}\left[\frac{\pi}{8}\right], -\text{Sin}\left[\frac{\pi}{8}\right]\right\}, \left\{\text{Cos}\left[\frac{\pi}{8}\right], \text{Sin}\left[\frac{\pi}{8}\right]\right\},\right.$$
$$\left\{-\text{Sin}\left[\frac{\pi}{8}\right], -\text{Cos}\left[\frac{\pi}{8}\right]\right\}, \left\{\text{Sin}\left[\frac{\pi}{8}\right], \text{Cos}\left[\frac{\pi}{8}\right]\right\}, \left\{\text{Sin}\left[\frac{\pi}{8}\right], -\text{Cos}\left[\frac{\pi}{8}\right]\right\},$$
$$\left.\left\{-\text{Sin}\left[\frac{\pi}{8}\right], \text{Cos}\left[\frac{\pi}{8}\right]\right\}, \left\{\text{Cos}\left[\frac{\pi}{8}\right], -\text{Sin}\left[\frac{\pi}{8}\right]\right\}, \left\{-\text{Cos}\left[\frac{\pi}{8}\right], \text{Sin}\left[\frac{\pi}{8}\right]\right\}\right\}$$

But we cannot stress enough how important it is for people learning *Mathematica* to avoid such high-density coding and use simpler fragments more amenable to debugging.

Now that we have the plot points we will use `ListPlot` to display them.

```
(* plot these points *)
g1 = ListPlot[pts];
```

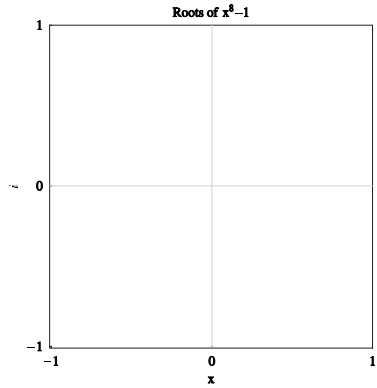


The point of this chapter is to familiarize you with the graphics tools available to you. We have made several changes to this plot. They are:

```
(* plot options for the refined plot *)
popts = {PlotRange → {{-1.01, 1}, {-1.01, 1}},
   AxesStyle → GrayLevel[0.8],
   AspectRatio → Automatic,
   PlotStyle → {cblu, PointSize[0.015]},
   Frame → True,
   FrameTicks → {{-1, 0, 1}, {-1, 0, 1}, None, None},
   FrameLabel → {"x", "i", "Roots of x^8-1", ""}};
```

Notice how clean and readable our plotting command is now.

```
(* plot these points *)
g2 = ListPlot[pts, popts];
```



We will discuss the plot options individually and we encourage the reader to get the code from the website and experiment by removing and changing parameters. That is the fastest way to learn.

Here are the plot options that we chose.

`PlotRange`: both axes from {-1.01,1}. If you use {-1,1} you will lose the labelling on the points -1 for both axes.

`AxesStyle`: we want to mute the axes so it does not draw attention away from the symmetric presentation of the points.

`AspectRatio`: Abandon the `GoldenRule` for `Automatic`.

`PlotStyle`: we made the points blue and a little bigger than default so they would be more apparent.

`Frame`: turn on the frame.

`FrameTicks`: simplify the ticks to reduce visual clutter.

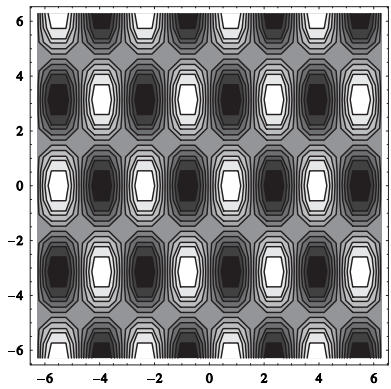
`FrameLabel`: add descriptive labels to the axes and add a title.

3.2.3 Contour plots

Contour plots are extremely useful plots that are too often ignored because people want the eye candy of a 3D plot.

Let's start immediately with a basic contour plot. We are deliberately using an asymmetric function because later on you will see a transpose of this function when you are not expecting it.

```
(* basic asymmetric contour plot *)
g0 = ContourPlot[Sin[2 x] Cos[y], {x, -2 π, 2 π}, {y, -2 π, 2 π}];
```

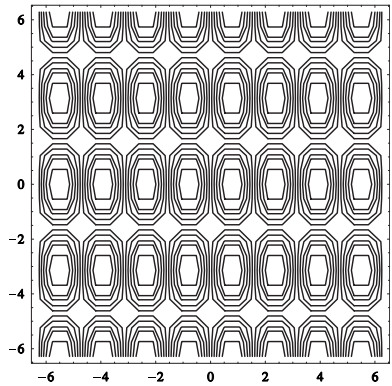


We think the `GrayLevel` shading is the best because you can readily tell that white is high and black is low. The hue color scheme is a poor choice here because of the periodic boundary conditions. Red is high and it is also low.

Whenever possible, we like to turn of the shading and focus on the contours.

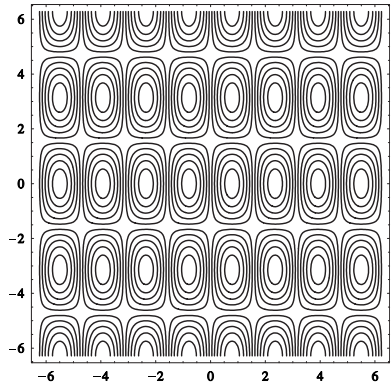
⚠ Don't use ColorFunction Hue with ContourPlots

```
(* often these graphs are easier to interpret without the shading *)
g1 = Show[g0, ContourShading → False];
```



We notice that the contours are not smooth, so we need to increase the number of sampling points.

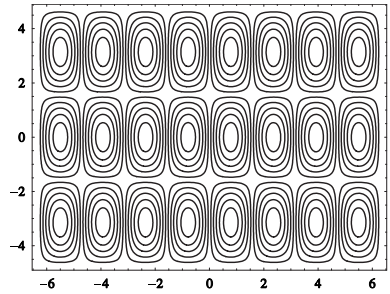
```
(* increasing the sample density smooths the contours *)
g2 = ContourPlot[Sin[2 x] Cos[y], {x, -2 π, 2 π},
    {y, -2 π, 2 π}, ContourShading → False, PlotPoints → 200];
```



The contours look much nicer, but remember that your generation time and output file size increase as the square of `PlotPoints`.

Until new users are comfortable with the two different address paradigms in *Mathematica*, we encourage you to stick with asymmetric plots and domains. This way when you encounter an unexpected transpose, you will notice it immediately.

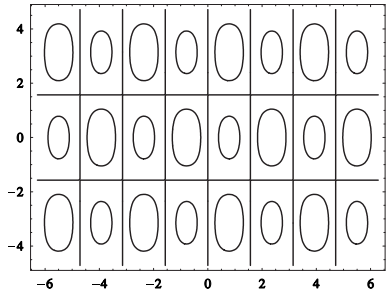
```
(* new users should work with asymmetric plot domains *)
g2a = ContourPlot[Sin[2 x] Cos[y], {x, -2 π, 2 π}, {y, -3/2 π, 3/2 π},
   ContourShading → False, PlotPoints → 200, AspectRatio → 3/4];
```



Notice here that the aspect ratio is determined by the plot domains.

Moving through the `ContourPlot` tutorial, we discuss the first thing you may want to control: the contour values. Instead of accepting the *Mathematica* default values, you can supply your own values. But we will see it is not as straightforward as you may expect. The syntax is simple enough; you pass a list to the setting `Contours` as shown here.

```
(* user-specified contours *)
g3 = ContourPlot[Sin[2 x] Cos[y], {x, -2 π, 2 π},
   {y, -3/2 π, 3/2 π}, ContourShading → False, PlotPoints → 200,
   Contours → {-1/2, 0, √2/2}, AspectRatio → 3/4];
```



We call attention to the contours that are horizontal and vertical lines. Such right angles are rare in contour plots, in general, but they are a property of many separable functions.

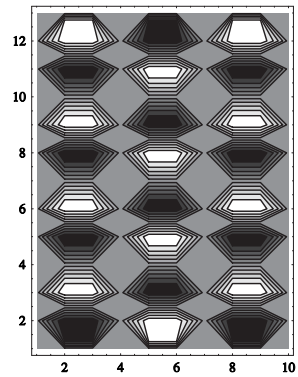
At this point we would like to shift to the discrete version of this command: `ListContourPlot`. As you would expect, instead of acting on continuous functions, this command operates on a list of points. The fine print states that the list must be rectangular and is basically a list of function values at a grid of measurement points.

In the example below we use the function from above and sample it in increments of δ. We deliberately choose a small number like this to show how you can easily have problems with aliasing.

```
(* create a discrete list of function values *)
δ = π/3;
(* notice the aliasing *)
lst = Table[Sin[2 x] Cos[y], {x, -2 π, 2 π, δ}, {y, -3/2 π, 3/2 π, δ}];
```

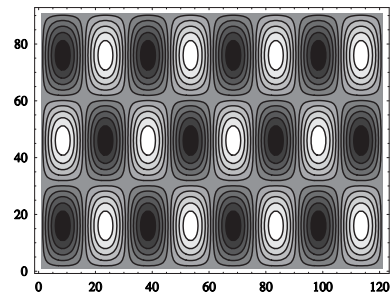
With the list of function values `lst` in hand, we are ready to plot the figure.

```
g4 = ListContourPlot[lst, AspectRatio → 4/3];
```



Here is the sudden transpose we warned you about.

```
(* notice the discrete list is *)
(* transposed compared to the continuous case *)
(* increase the sample density by an order of magnitude *)
δ = π/30;
lst = Table[Sin[2 x] Cos[y], {x, -2 π, 2 π, δ}, {y, -3/2 π, 3/2 π, δ}];
g5 = ListContourPlot[Transpose[lst], AspectRatio → 3/4];
```



3.3 Pictionary of 2D graphic types

Mathematica is an absolute treasure chest of graphical tools. The breadth and depth are significant. We do however have one minor issue with the presentation since we have on several occasions been searching vigorously through the Help Browser and

The Mathematica Book looking for a specific format. It would be helpful to collect all the plot types and locate them together. That is exactly what we have done here. We have brought together all the 2D and 3D graphics types to allow you to quickly find the format you are looking for or to simply familiarize yourself with the tools of *Mathematica*.

We will show the plots first and following those will be the specific commands to produce each image. We have tried to present the many different ways you may employ the graphics package and so similar plots may have very different command syntax.

3.4 Plotting in three dimensions

Mathematica has a powerful and diverse tool kit for three-dimensional graphics. The Help Browser entry shows a smattering of what is possible with the tool kit.

✵ Built-in Functions > Graphics and Sound > Graphics Primitives > Graphics3D

3.4.1 Plot3D

```
No more memory available.
Mathematica kernel has shut down.
Try quitting other applications and then retry.
```

✵ Demos > Graphics > 3D Graphics

The tool kits for the three-dimensional plots are a little richer and artistry is a bit more important.

First of all, when doing a series of plots or comparing plots, it is important to use `SphericalRegion→True`. This will make sure that the unit vectors have the same size in both plots.

Second, to orient your figures, use the 3D VewPoint Selector under the Input entry on your menu bar. This is far more convenient than varying numbers in a list.

⚠ Be prudent and set `SphericalRegion→True`

⚠ To orient your figures use the menu bar entry Input > 3D VewPoint Selector

3.4.2 The sampling function

We begin with a simple example using the sampling function, also known as the sinc (pronounced “sink”) function which is defined as

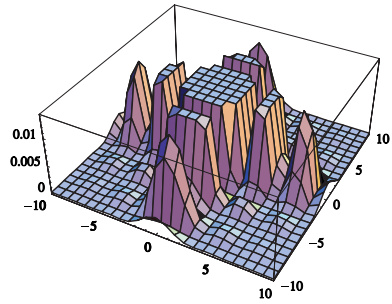
$$\mathrm{sinc}(x) \equiv \frac{\sin(x)}{x} \tag{3.2}$$

When a plane wave irradiates a rectangular aperture the spatial irradiance distribution in the Fourier plane has the form $sinc^2(x)sinc^2(y)$. Let’s plot the profile. The

units on the x and y axes will be in units where the aperture size is unity. Note the trademark simplicity of the plot syntax.

```
sinc[x_] := Sin[x]/x;
Plot3D[sinc[x]^2 sinc[y]^2, {x, -10, 10}, {y, -10, 10}];
```

We will study the resultant graph in detail.



Notice that *Mathematica* has greatly truncated the height scale in order to fully resolve the fine structure at the base of the peak. Notice too that we automatically get a bounding box and that the default aspect ratio, described by `BoxRatios` is:

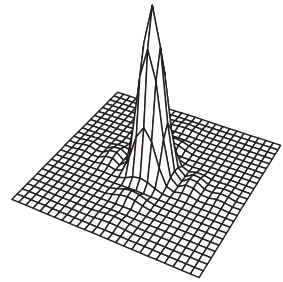
```
Options[%, BoxRatios]

{BoxRatios → {1, 1, 0.4}}
```

Let's change the aspect ratios and a few other parameters. Observe that there is no need to rerender the graphic to make these changes. Suppose we are preparing this graphic for a technical journal and we need to render a crisp black and white image. We will then turn off the shading and boost the sampling density. We will also turn off the axes because the vertical units are arbitrary and the x, y units are scaled to the aperture. Also, we will turn on the `SphericalRegion` bounding feature.

```
gg2 = Show[gg1, Shading → False, PlotRange → All, BoxRatios → {1, 1, 1},
   Boxed → False, Axes → False, SphericalRegion → True];
```

Our graph now appears as:



You will notice that the figure seems to be off-center and closer to the bottom. This is because of the `SphericalRegion` bounds. One advantage of this option is that all of your common figures will import into documents with exactly the same size.

We would like to boost the sampling density and this will require the figure to be rendered again. We will try a `PlotPoints` value of {65, 65}.

```
gg3 =
  Plot3D[sinc[x]^2 sinc[y]^2, {x, -10, 10}, {y, -10, 10}, Shading → False,
   PlotRange → All, BoxRatios → {1, 1, 1}, Boxed → False,
   Axes → False, PlotPoints → {65, 65}, SphericalRegion → True];
```

Disaster ensues. We show an abridged list of the errors.

```
⚠ Power :: infy : Infinite expression 1/0. encountered . More...
```

```
⚠ ∞ :: indet : Indeterminate expression 0. ComplexInfinity encountered . More...
```

```
⚠ General :: stop :
  Further output of Power :: infy will be suppressed during this calculation . More...
```

```
⚠ Plot3D ::plnc :
   sinc [x]^2 sinc [y]^2 is neither a machine -size real number at {x, y}={0., -10.}
     nor a list of a real number and a valid color directive . More...
```

```
⚠ General ::stop : Further output of
     Plot3D ::plnc will be suppressed during this calculation . More...
```

```
⚠ Plot3D ::gval : Function value Indeterminate
     at grid point xi = 1, yi = 33 is not a real number . More...
```

```
⚠ General ::stop : Further output of
     Plot3D ::gval will be suppressed during this calculation . More...
```

Fortunately for us, *Mathematica* will only repeat an error message three times per command and then it will suppress further instances. We will show the plot generated in a moment. For now, we call your attention to the error messages.

Look at the message `ΔPlot#d::plnc:`(fourth entry). It tells us that at the address `{x, y} = {0, 10}` the plot function had an indeterminate value. This is a very helpful piece of information that allows us to begin probing directly. Let's reproduce this error.

```
sinc[0] sinc[-10]

⚠ Power ::infy : Infinite expression 1/0 encountered . More...

⚠ ∞::indet : Indeterminate expression 0 ComplexInfinity encountered . More...

Indeterminate
```

There is a problem with the product of these two numbers. We will look at them individually.

```
sinc[0]
sinc[-10]
```

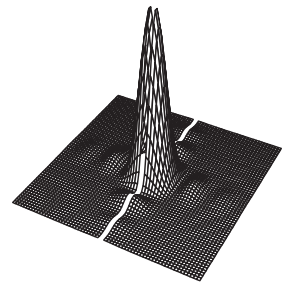
```
⚠ Power :: infy : Infinite expression 1/0 encountered . More...

⚠ ∞ :: indet : Indeterminate expression 0 ComplexInfinity encountered . More...

Indeterminate
```

```
Sin[10]
  10
```

The problem is that we did not follow the prescription in [19]. We need to fix the definition of the function at zero. The graph makes this fairly clear. The plot routine chooses *x* and *y* sample values of exactly zero.



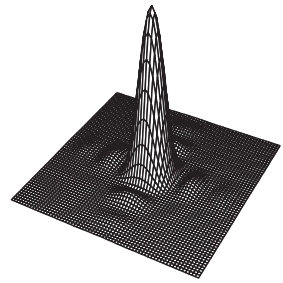
We'll cover limits in chapter 5. For now, we think it should be fairly clear how this command operates.

```
Limit[sinc[x], x → 0]

1
```

With a corrected definition we are done.

```
gg4 =
  Plot3D[sinc[x]^2 sinc[y]^2, {x, -10, 10}, {y, -10, 10}, Shading → False,
   PlotRange → All, BoxRatios → {1, 1, 1}, Boxed → False,
   Axes → False, PlotPoints → {65, 65}, SphericalRegion → True];
```



Sinc function

The switch from Plot3D

Let's return to another example with the now-familiar Zernike polynomials.

$$Z_{8,\,4}(r,\,\theta) = 70r^8 - 140r^6 + 90r^4 - 20r^2 + 1 \qquad (3.3)$$

For years, these decades were plotted on a rectangular domain looking like the proverbial round peg forced into the square hole. For demonstration purposes, we will reproduce the result here.

The functional definition for the Zernike polynomials we will use is:

```
(* Zernike polynomial Z 84 which is rotationally invariant *)
g[r_] := 70 r^8 - 140 r^6 + 90 r^4 - 20 r^2 + 1;
```

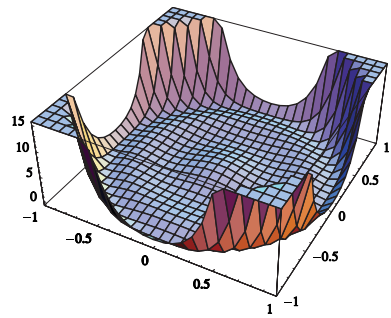
However, we need to put this into Cartesian coordinates. *Mathematica* will do this nicely for us.

```
(* convert to cartesian coordinates *)
h = Simplify[g[r] /. r → √(x^2 + y^2)]
```

$$1 - 20\,(x^2 + y^2) + 90\,(x^2 + y^2)^2 - 140\,(x^2 + y^2)^3 + 70\,(x^2 + y^2)^4$$

Accepting the defaults we produce this frightful rendition.

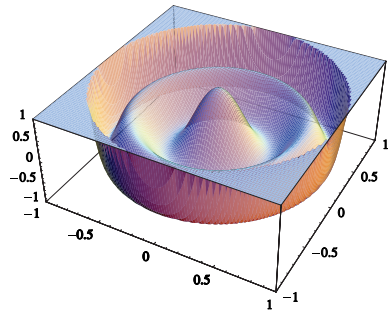
```
(* first attempt to plot Z84 *)
g1 = Plot3D[h, {x, -1, 1}, {y, -1, 1}];
```



From experience we know that the polynomials are bound on the range [-1, 1]. This figure has the scale dominated by height values outside of the unit circle. So we need to input the range manually. Also, we want to increase the resolution which will require us to suppress the mesh.

```
(* boost resolution, suppress mesh,fix range *)
g1 = Plot3D[h, {x, -1, 1}, {y, -1, 1},
   Mesh → False, PlotPoints → 100, PlotRange → {-1, 1}];
```

The result, while not pleasing, is not as horrific.



We could try to plot this function on a rectangular domain using Plot3D and a masking function which will suppress any part of the function outside of the unit disk. A sample masking function is given here.

```
(* masking or aperture function *)
mask[x_, y_] := Null      /;  x^2 + y^2 > 1;
mask[x_, y_] := 1         /;  x^2 + y^2 ≤ 1;
```

We can pull the conditionals inside of the argument list as shown below. The choice is a matter of style, not function.

```
(* masking or aperture function *)
(* conditionals inside the argument list *)
mask[x_, y_ /; x^2 + y^2 > 1;] := Null
mask[x_, y_ /; x^2 + y^2 ≤ 1;] := 1
```

Pressing forth we will use the masking function and see what kind of relief we are afforded.

```
(* use a masking function to exclude data outside the unit disk *)
ppts = 100;
g2 = Plot3D[h mask[x, y], {x, -1, 1}, {y, -1, 1},
   Mesh → False, PlotPoints → ppts, PlotRange → {-1, 1}];
```

which produces the subsequent plot. We will talk about the errors generated during the screen rendering and the EPS file rendering. What the messages are telling us is

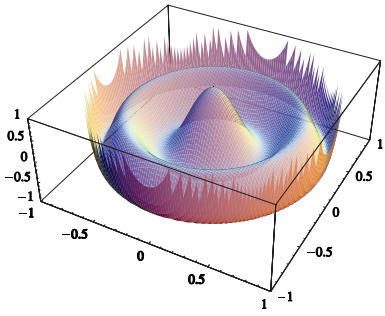
that we are asking *Mathematica* to represent `Null`s, which are not numbers. The first error message comes from the first point it samples. This quantity is `Null Null`. This message is repeated for the default three times and then suppressed.

```
Plot3D::plnc :
  hmask[x, y] is neither a machine-size real number at {x, y}={-1., -1.}
    nor a list of a real number and a valid color directive . More...
```

The next series of messages come when *Mathematica* is told to plot a number times the `Null`, still unplottable. This message too is repeated three times and then suppressed.

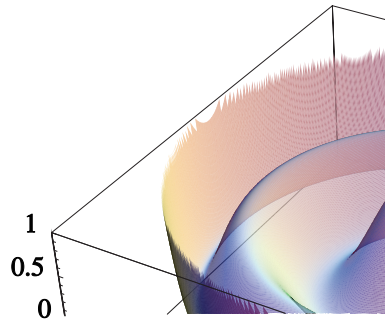
```
Plot3D::gval : Function value 121.457 Null
    at grid point xi = 1, yi = 3 is not a real number . More...
```

Eventually we will see this plot.



This exercise has two messages. One is about the difficulty in plotting circular objects in rectangular domains. The other is that we can use `Null` to control our graphics.

This new image is unsatisfactory. Our last hope is to boost the resolution dramatically. When we set `PlotPoints` to 500, we get this:



We have paid a grievous price to get to this point. Look at the time and file size penalties shown below in table 3.4. The generation time is short, but the time to render into an EPS file is several minutes. Also the graphic size is so huge that we are in danger of creating a document that cannot be printed.

PlotPoints	time, s	file size, KB
40	0.06	148
100	0.95	685
500	21.17	14,839

Table 3.4: Creation times and file sizes for plots of different resolutions. Notice the time and file sizes increase as the square of PlotPoints.

ListPlot3D

Before we turn to nonrectangular plot strategies, we will examine discrete forms of plotting. The first is ListPlot3D which takes an ordered list of height values and plots them on a 2D array.

We will keep using the same sample function from the previous section and generate a list of points. The sample spacing is $d = 0.2$ units telling us that we will have 11 sample points. This is far less than the default value of 25 points used in Plot3D.

```
(* space between samples *)
δ = 0.2;
(* generate an ordered list of function values *)
(* notice that we are varying x and y yet plotting with r *)
(* a benefit of rotational invariance *)
pts =
  Table[r = √(x^2 + y^2); If[r ≤ 1, g[r], Null], {x, -1, 1, δ}, {y, -1, 1, δ}];
```

The plot syntax is elementary.

```
(* plot the list of discrete points *)
(* axes refer to position in list, not position in space *)
g3 = ListPlot3D[pts];
```

We ran different resolutions to show what would happen and we present the results in table 3.5. For the plots at higher density we have suppressed the surface mesh. As with all list-type plots we lose information about the coordinate system the samples were made in. In this case we varied the coordinates x and y inside the domain [-1, 1] and created a list of points devoid of spatial location information. For example, in the rightmost figure the point at the x = 750, y = 500 means this is the height value from the 750th row and the 500th column.

As before if you wish to preserve the spatial information of the plot positions, we will need to turn to a scatter-type plot.

ScatterPlot3D

Before we turn to non-rectangular plot strategies, we will examine discrete forms of plotting. The first is `ListPlot3D` which takes an ordered list of height values and plots them on a 2D array.

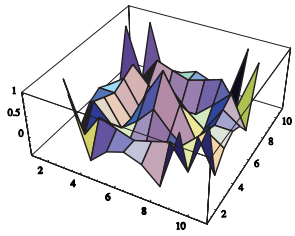
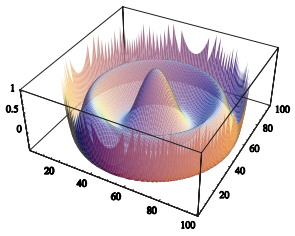
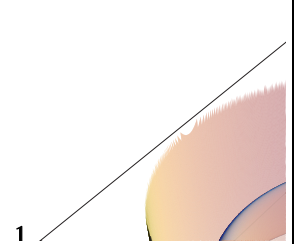
		
δ = 0.2 (11 points)	δ = 0.02 (101 points)	δ = 0.002 (1001 points)

Table 3.5: `ListPlot3D`, the discrete version of `Plot3D`. Here are plots of the Zernike polynomial from the previous section. The parameter d is the gap between samples in the *x*-*y* plane. The number in parentheses next to it is the number of sample points along an axis. Notice that the *x*-*y* axes in these plots are no longer referring to a physical space. Instead they simple refer to a position in the data array.

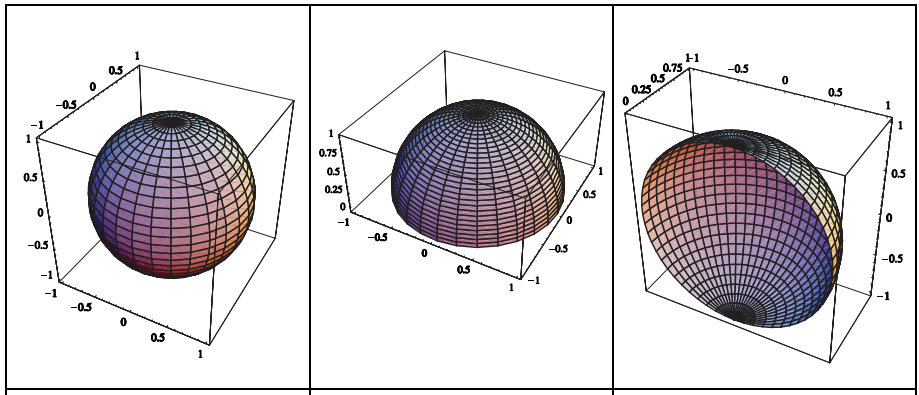
3.4.3 SphericalPlot3D

```
Options[SphericalPlot3D]

{AmbientLight → GrayLevel[0.], AspectRatio → Automatic,
 Axes → True, AxesEdge → Automatic, AxesLabel → None,
 AxesStyle → Automatic, Background → Automatic,
 Boxed → True, BoxRatios → Automatic, BoxStyle → Automatic,
 ColorOutput → Automatic, Compiled → True, DefaultColor → Automatic,
 DefaultFont :→ $DefaultFont, DisplayFunction :→ $DisplayFunction,
 Epilog → {}, FaceGrids → None, FormatType :→ $FormatType,
 ImageSize → Automatic, Lighting → True, LightSources →
  {{{1., 0., 1.}, RGBColor[1, 0, 0]}, {{1., 1., 1.}, RGBColor[0, 1, 0]},
   {{0., 1., 1.}, RGBColor[0, 0, 1]}}, Plot3Matrix → Automatic,
 PlotLabel → None, PlotPoints → Automatic, PlotRange → Automatic,
 PlotRegion → Automatic, PolygonIntersections → True, Prolog → {},
 RenderAll → True, Shading → True, SphericalRegion → False,
 TextStyle :→ $TextStyle, Ticks → Automatic, ViewCenter → Automatic,
 ViewPoint → {1.3, -2.4, 2.}, ViewVertical → {0., 0., 1.}}
```

Each of these entries has a description within the Help Browser and over time you can assimilate the nuances. We want to demonstrate that you can begin plotting immedi-

ately. We do want to pass on two tips for those of you who will plan to scour the Help Browser later.



$0 \le \theta \le \pi,\ 0 \le \varphi \le 2\pi$	$0 \le \theta \le \frac{\pi}{2}$	$0 \le \varphi \le \pi$

Table 3.6: `SphericalPlot3D`, the discrete version of `Plot3D`. Here are plots of the Zernike polynomial from the previous section. The parameter *d* is the gap between samples in the *x*-*y* plane. The number in parentheses next to it is the number of sample points along an axis. Notice that the *x*-*y* axes in these plots are no longer referring to a physical space. Instead they simply refer to a position in the data array.

Spherical harmonics

We will start with the single example of plotting the spherical harmonic functions and then of the more typical manipulations a reader may make when publishing graphics. The spherical harmonic functions $Y_l^m(\theta,\ \phi)$, as shown in a multitude of books on quantum mechanics, are eigenfunctions of the angular momentum operators L^2 and L_z in spherical polar coordinates. The square of the spherical harmonic $\left|Y_l^m(\theta,\ \phi)\right|^2$ is interpreted as the probability density for the particles angular coordinates. We can look at the integral

$$\int_{\phi_1}^{\phi_2}\int_{\theta_1}^{\theta_2} \left|Y_l^m(\theta,\ \phi)\right|^2 (d\theta)d\phi \tag{3.4}$$

to tell us the probability of finding the particle between the bounds

$$\theta_1 \le \theta \le \theta_2 \ \text{ and } \ \phi_1 \le \phi \le \phi_2\,. \tag{3.5}$$

As you might expect, *Mathematica* makes these functions available to us and these are called `SphericalHarmonicY[l, m, q, f]`. Since it is so easy, let's examine the first few polynomials in this series.

```
(* loop through l values [0, 3] *)
Do[
  (* loop through m values [-l, l] *)
  Do[
    Print[{l, m}, ": ", SphericalHarmonicY[l, m, θ, φ]];
    , {m, -l, l}];
  , {l, 0, 2}];
```

$$\{0, 0\}: \frac{1}{2\sqrt{\pi}}$$

$$\{1, -1\}: \frac{1}{2} e^{-i\varphi} \sqrt{\frac{3}{2\pi}} \text{Sin}[\theta]$$

$$\{1, 0\}: \frac{1}{2} \sqrt{\frac{3}{\pi}} \text{Cos}[\theta]$$

$$\{1, 1\}: -\frac{1}{2} e^{i\varphi} \sqrt{\frac{3}{2\pi}} \text{Sin}[\theta]$$

$$\{2, -2\}: \frac{1}{4} e^{-2i\varphi} \sqrt{\frac{15}{2\pi}} \text{Sin}[\theta]^2$$

$$\{2, -1\}: \frac{1}{2} e^{-i\varphi} \sqrt{\frac{15}{2\pi}} \text{Cos}[\theta] \text{Sin}[\theta]$$

$$\{2, 0\}: \frac{1}{4} \sqrt{\frac{5}{\pi}} \left(-1 + 3 \text{Cos}[\theta]^2\right)$$

$$\{2, 1\}: -\frac{1}{2} e^{i\varphi} \sqrt{\frac{15}{2\pi}} \text{Cos}[\theta] \text{Sin}[\theta]$$

$$\{2, 2\}: \frac{1}{4} e^{2i\varphi} \sqrt{\frac{15}{2\pi}} \text{Sin}[\theta]^2$$

do loop

paste from sh 5 3 01

results

Let's pick the specific example computing the square of the magnitude of $Y_5^3(\theta, \phi)$. As we can see, the function is complex, so we will use the identity:

$$|z|^2 = z\bar{z} \tag{3.6}$$

where $\bar{z}$ is the complex conjugate of z. When we perform this process we find:

```
{l, m} = {5, 3};
g2 = SphericalHarmonicY[l, m, θ, φ]
  Conjugate[SphericalHarmonicY[l, m, θ, φ]]
```

$$\frac{1}{1024\,\pi}\left(385\, e^{3\,i\,\varphi - 3\,i\,\text{Conjugate}[\varphi]}\,(-1 + 9\,\text{Conjugate}[\text{Cos}[\theta]]^2)\right.$$
$$\left.\text{Conjugate}[\text{Sin}[\theta]]^3\,(-1 + 9\,\text{Cos}[\theta]^2)\,\text{Sin}[\theta]^3\right)$$

Mathematica has returned the most general form which tells us that it allows for the possibility that the angles θ and φ may be complex. We will use `Simplify` with the assumption that these angles are real.

```
Simplify[%, θ ∈ Reals ∧ φ ∈ Reals]
```

$$\frac{385\,(7 + 9\,\text{Cos}[2\,\theta])^2\,\text{Sin}[\theta]^6}{4096\,\pi}$$

Now we are ready to write a module to generate the square of the eigenfunctions.

```
(* module to generate the squared spherical harmonic function *)
pgen[l_Integer, m_Integer] := Module[{g},
    g = SphericalHarmonicY[l, m, θ, φ];
    g2 = Simplify[g Conjugate[g], θ ∈ Reals ∧ φ ∈ Reals];
   ];
 (* specify the eigenstate *)
 {l, m} = {5, 3};
 (* call the generation module *)
 pgen[l, m]
 (* display the squared function *)
 g2
```

$$\frac{385\,(7+9\,\text{Cos}[2\,\theta])^2\,\text{Sin}[\theta]^6}{4096\,\pi}$$

With the computation in hand we are now ready to plot the functions. First, we will accept all the default values.

```
g00 = SphericalPlot3D[g2, {θ, 0, π}, {φ, 0, 2 π}];
```

Next we suppress the labeling of the axis since we have the luxury of only being concerned with qualitative information.

```
g01 = Show[g00, Axes → False];
```

Notice that we do not have to rerender the graphic. We can use the `Show` command to display the existing graphic with the new option. For detail-intensive figures this can be a tremendous time savings.

What remains behind in the bounding box we will now dispose of.

```
g02 = Show[g01, Boxed → False];
```

It is always prudent to bundle your plot options. Assigning them a variable name not only helps keep your plot command free from clutter but also allows you to easily standardize your figures. We will bundle our options here in the variable `opts`.

Since we have made all the simple changes, we now want to increase the sampling density to make the figure look smoother.

```
opts = {Axes → False, Boxed → False, PlotPoints → {150, 50}};
g03 = SphericalPlot3D[g2, {θ, 0, π}, {φ, 0, 2 π}, opts];
```

At this higher sampling density the mesh lines are running together. They need to be removed using the `EdgeForm[]` command.

```
g04 = SphericalPlot3D[{g2, EdgeForm[]}, {θ, 0, π}, {φ, 0, 2 π}, opts];
```

At this juncture we can talk about shading and illumination. Experience tells us that the quickest way to understand the effects of illumination and shading are to look at examples. As you might suspect, the Help Browser has some very brief and informative examples showing variation of these parameters. We will use a light source from one of these examples.

```
ls = LightSources →
  {{{0, 0, 1}, RGBColor[0, 0, 1]}, {{1, 0, 0.4}, RGBColor[1, 0, 0]},
   {{0, 1, 0.4}, RGBColor[0, 1, 0]}};
```

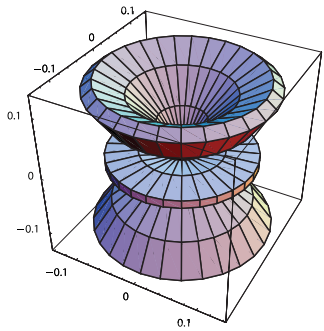
Notice how easy it is to see and manipulate the light sources when they are separated like this. See how compact the plot command can be.

```
g05 =
  SphericalPlot3D[{g2, EdgeForm[]}, {θ, 0, π}, {φ, 0, 2 π}, opts, ls];
```

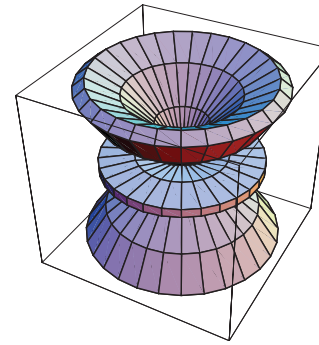
Demos > Graphic Gallery > Color Charts > Light Source Variation
Demos > Graphic Gallery > Color Charts > Surface Color Variation

Spherical Harmonic

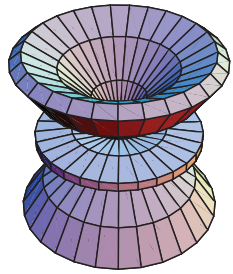
For example, see McGervey, J.D., *Introduction to Modern Physics*, Academic Press, 1971.



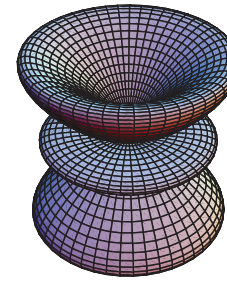
1. g0



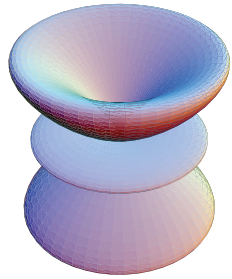
2. g1



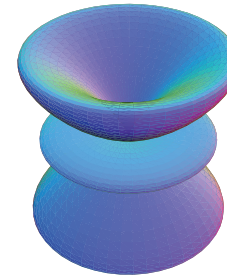
3. g2



4. g3



5. g4



6. g5

Table 3.7: A typical evolution for a 3D plot.

3.5 Rotation through parity states

We would like to use this section as an opportunity to discuss a little-known property of the Zernike polynomials. After you are done here, you'll see how this applies to more common special functions like the polynomials of Legendre.

We are going to discuss the parity properties of these polynomials. In physics, we classify the functions according to a parity operator P which acts upon a function f. If the function f has a definite state of parity then we can write:

$$f(\pm x) = P[f(x)] \tag{3.7}$$

and P has a multiplicative action of either 1 or -1. In other words, if:

$$f(-x) = f(x) \tag{3.8}$$

the function is even, and if:

$$f(-x) = -f(x) \tag{3.9}$$

the function is odd. Look at the two graphs below.

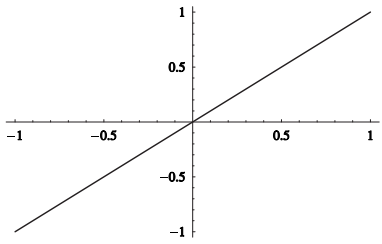
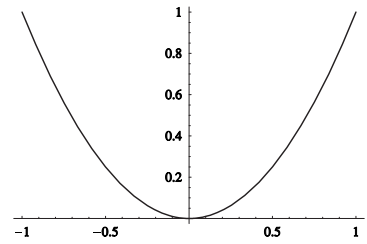
Odd function: *f(-x) = -f(x)*	Even function: *f(-x) = f(x)*
	

Table 3.8: Two possible types of definite parity for functions.

Of course, any function *g(x)* can be resolved into two parity components, even $g^{+}(x)$ and odd $g^{-}(x)$, using this prescription.

$$g^{+}(x) = \frac{1}{2}[g(x)+g(-x)],\ g^{-}(x) = \frac{1}{2}[g(x)-g(x)] \tag{3.10}$$

To bolster this concept, let's have *Mathematica* do an example for us. We'll use the trial function $g(x) = x + x^2$.

```
(* a function with indefinite parity *)
g[x_] := x + x^2;
```

Now we define the operators (modules) to extract the even and odd components of a given function.

```
(* even parity component *)
even[f_] := 1/2 Simplify[f[x] + f[-x]];
```

```
(* odd parity component *)
odd[f_] := 1/2 Simplify[f[x] - f[-x]];
```

Hopefully, we are able to compute the answers already. Here is what *Mathematica* says.

```
(* pull out the even terms *)
even[g]
```

$$x^2$$

```
(* pull out the odd terms *)
odd[g]
```

$$x$$

As always we perform some form of check. Here we will ensure that the sum of the even and odd components restores the original components.

```
(* we should recover the original function *)
odd[g] + even[g]
```

$$x + x^2$$



Zernike Polynomials

Parity

While *The CRC Concise Encyclopedia of Mathematics* gives a surprisingly thorough list of properties for the Zernike polynomials, it does not mention that the polynomials are prescribed in complex space. Zernike defined his polynomials as:

$$R_n^m(r)e^{\pm jm\theta} \tag{3.11}$$

where $R_n^m(r)$ is clearly a radial equation. The Zernike polynomials that you see in tables are the functions $Z_n^{\pm m}(r,\ \theta)$ where

$$Z_n^m(r,\ \theta) \pm Z_n^{-m}(r,\ \theta) = R_n^m(r)e^{\pm jm\theta} \tag{3.12}$$

We see that the Euler identity

$$e^{ix} = \cos x + i\sin x \tag{3.13}$$

resolves the complex exponential into functions of even (cosine) and odd (sine) parity. In this vein, we would expect then a complex exponential to be resolved into two distinct parity states. In fact this is the case here.

$$Z_n^m(r,\ \theta) = R_n^m(r)\cos(m\theta), Z_n^{-m}(r,\ \theta) = R_n^m(r)\sin(m\theta) \tag{3.14}$$

Lacking this insight, most people fail to think of the Zernike polynomials as paired functions of opposite parity. (Note: the rotationally invariant polynomials $Z_n^0(r,\ \theta)$ do not have a parity partner. This should be clear by the end of this section.)

Polynomial	Polar form	Cartesian form
Z_{41}	$(4r^4 - 3r^2)\sin 2\theta$	$8x^3y + 8xy^3 - 6xy$
Z_{43}	$(4r^4 - 3r^2)\cos 2\theta$	$4y^4 - 4x^4 - 3y^2 + 3x^2$

Table 3.9: A closer look at two different parity states. One would not expect these functions to be definite parity states of the same function.

C H A P T E R 4

Ordinary differential equations

An equation involving one or more derivatives is known as a differential equation. There are two basic tools that can be used to solve differential equations using *Mathematica*, the `DSolve` and `NDSolve` commands. `DSolve` can be used to arrive at symbolic solutions, while we can use `NDSolve` to find numerical solutions to differential equations. In this chapter we will explore the use of `DSolve`.

4.1 Defining, entering and solving differential equations

Let's begin by considering a very simple differential equation. Let *y(x)* be a function such that its derivative with respect to *x* is a constant:

$$\frac{dy}{dx} = A \tag{4.1}$$

Multiplying both sides by *dx* we have:

$$dy = A\,dx \tag{4.2}$$

We can then integrate both sides:

$$dy = A\int d. \tag{4.3}$$

Giving the solution:

$$y = Ax + C \tag{4.4}$$

To solve a problem like this using *Mathematica,* we use the `DSolve` command. First, let's ask *Mathematica* to tell us something about `DSolve` by typing `?DSolve`:

```
?DSolve
```

```
DSolve [eqn, y, x] solves a differential equation for the function
   y, with independent variable x. DSolve [{eqn1, eqn2, ... }, {y1,
   y2, ... }, x] solves a list of differential equations . DSolve [eqn,
   y, {x1, x2, ... }] solves a partial differential equation . More...
```

Now let's use `DSolve` to obtain the solution we just found analytically. First, we define the equation. Equations that will be used with the `DSolve` command need to have two equal signs. Writing $\frac{dy}{dx}$ as `D[y[x], x]`, we have:

```
(* simple differential equation *)
eqn = D[y[x], x] == A

y'[x] == A
```

Next, we use this expression as input to the `DSolve` command. To get a solution to the equation, we need to tell *Mathematica* three things. The first is the equation we want to solve, which we have just defined using the variable `eqn`. Next, we need to tell *Mathematica* the function we are solving for. In this case, we are solving for `y[x]`. Finally, we have to specify the independent variable, *x*. These pieces of data are passed to `DSolve` and separated by commas. Let's store the solution returned by `DSolve` in a variable called `result`:

```
(* solve the differential equation *)
result = DSolve[eqn, y[x], x]

{{y[x] → A x + C[1]}}
```

Mathematica correctly reports the result including the constant of integration, which serves as the initial condition. But, as things currently stand, the form of the solution

is not of much use. For example, suppose we wanted to display y(7). If we simply type this into *Mathematica*, it just repeats what we typed:

```
y[7]

y[7]
```

To see how to extract the solution, notice that the solution is enclosed in double braces. This tells us that the **DSolve** command returns the solutions it finds in a list. In this case, with a simple first order differential equation, there is only one solution. We can use the list manipulation techniques we learned in chapter 1 to extract the contents of this list. For this example, let's define a new function of *x* and extract the solution the way we would extract an element of an ordinary list. This is done by writing `listname[[i,j,...]]`, and in this case, the list name is simply the name we have given to the returned solution, result. So we write:

```
(* extract the functional dependence *)
f[x_] := result[[1,1,2]]

Ax + C[1]
```

Now we have the solution in a format that we can use. Suppose that we are told that $f(0) = 2$ and $A = 7$. We can type these values into *Mathematica* and then obtain a plot of the function $f(x)$, say for $0 \le x \le 10$:

```
(* load values for the parameters *)
C[1] = 2;
A = 7;
g1 = Plot[f[x], {x, 0, 10}];

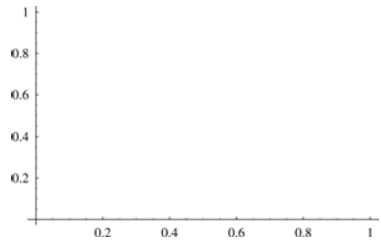
⚠ Set::write : Tag C in C[1] is Protected . More...

⚠ Plot::plnr :
  f[x] is not a machine-size real number at x = 4.1666666666666667`*^-7 . More...

⚠ Plot::plnr :
  f[x] is not a machine-size real number at x = 0.40566991572915795` . More...

⚠ Plot::plnr :
  f[x] is not a machine-size real number at x = 0.8480879985937368` . More...

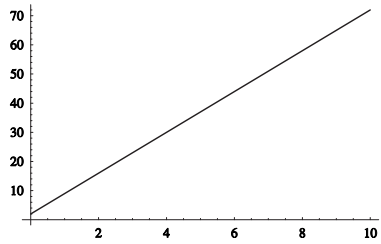
⚠ General::stop :
  Further output of Plot::plnr will be suppressed during this calculation . More...
```



Unfortunately, this doesn't seem to have worked. It turns out that *Mathematica* allows us to specify initial conditions using the `DSolve` command. This can be done by enclosing the equation you want to solve together with the initial condition in braces. The two must be separated by a comma. So let's try again, this time telling `DSolve` that `y[0] = 2`:

```
(* supply an initial value *)
result = DSolve[{eqn, y[0] == 2}, y[x], x]
(* load the solution into the function g *)
g[x_] := result[[1,1,2]]
(* specify the parameter A *)
A = 7;
(* view the plot *)
g1 = Plot[g[x], {x, 0, 10}];

{{y[x] → 2 + 7 x}}
```



This time *Mathematica* has given us exactly what we wanted. After some exercises, let's move on and try a more realistic example.

Exercises
Use *Mathematica* to obtain general solutions to the following equations:

4.1 $\frac{\partial x}{\partial t} - 6x = 0$

4.2 $\frac{\partial y}{\partial x} = \frac{x^2}{2x+3} y$

4.3 $\frac{\partial x}{\partial t} = x \ln(t), \; x(0) = 1$

4.1.1 Example: Newton's law of convective cooling

A metal bar is heated to 350°F. In 20 seconds, the temperature of the bar has dropped to 275°. Find the amount of time required for the bar to drop to a temperature of 80°. The temperature of the surrounding environment is 70°.

Let's quickly review Newton's law of cooling, or introduce it for readers not familiar with this equation. Newton's law of cooling relates the temperature change of a heated object to the temperature of the surrounding environment in the following

way. If we call the temperature of the object *T(t)* and the temperature of the surrounding environment *A*, we find that:

$$\frac{\partial T}{\partial t} = -k(T - A) \tag{4.5}$$

Note that we are assuming that the temperature of the environment is a constant. If we call the initial temperature *B*, we can obtain a solution to this simple differential equation using the same techniques discussed in the previous section. First, we use `DSolve` to find the solution:

```
(* using DSolve on convective (vs. radiative) cooling *)
result = DSolve[{∂t T[t] == -k (T[t] - A), T[0] == B}, T[t], t]

{{T[t] → e^-kt (-A + B + A e^kt)}}
```

This time we've placed the equation directly into `DSolve` without first defining a variable to store it. Remember, use two equal signs in defining the equation:

```
∂t T[t] == -k (T[t] - A)
```

Also recall that the equation and the initial condition are enclosed in braces `{}` and separated by a comma:

```
{∂t T[t] == -k (T[t] - A), T[0] == B}
```

Finally, we again told *Mathematica* two more pieces of information. The first is the function we are solving for, which in this case is *T(t)*. The second is the independent variable, which is *t*. These have been placed after the definition of the equation and the initial condition and separated by commas.

The solution is once again returned in the form of a list. The first item of business is to extract that solution from the list so we can use it. Note that as the problem currently stands, there are really two variables in the problem, the time *t* and the constant of proportionality *k*. We will be able to use the problem data to solve for k, but with two variables, we need to define the solution as a function of t *and* k. Following the same procedure we used above, we write:

```
(* load the solution into a function *)
h[t_, k_] = result⟦1,1,2⟧

e^-kt (-A + B + A e^kt)
```

Now that we have the solution in a form that we can use, let's put in some specific values for the constants. First consider the value of *f(t)* at $t = 0$:

```
(* intial value for T *)
h[0, k]

B
```

We see that *B* gives us the initial temperature of the object. In the statement of the problem, we are told that the bar is initially at 350°. So we need to set $B = 350$. What about the other constant in the solution, *A*? *A* represents the temperature of the surrounding environment, which is 70°. To find the constant of proportionality *k*, we use the given data that the temperature of the bar is 275° after 20 seconds. We do this by setting *f*(20) = 275 and using the `Solve` command:

```
(* specify parameters *)
A = 70; B = 350;
 (* solve for proportionality constant *)
 k0 = Solve[h[20, k] == 275, k] // N

⚠ Solve::ifun :
  Inverse functions are being used by Solve, so some solutions may not
    be found; use Reduce for complete solution information. More...

{{k → 0.015589}}
```

Again, we have the solution in a list. We extract it using the same notation that we have used to extract the solutions to the differential equations. We use the Greek symbol κ to store the proportionality constant just found:

```
(* load the numeric value into κ *)
κ = k0[[1,1,2]]

0.015589
```

Now we can see what *h(t,*κ*)* looks like for the given initial condition, environmental temperature, and proportionality constant:

```
(* solution function *)
h[t, κ]
```

$$e^{-0.015589\,t}\,(280 + 70\,e^{0.015589\,t})$$

The final piece of the problem is to find the time at which the iron bar has reached a temperature of 80°. We again use `Solve`, this time solving for t using the value of κ we just found:

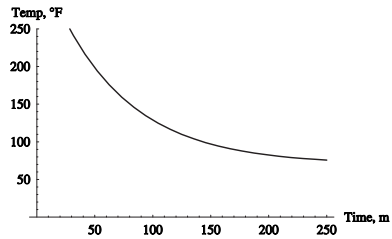
```
(* solve for proportionality constant *)
t0 = Solve[h[t, κ] == 80, t] // N
```

```
Solve::ifun : Inverse functions are being used by Solve, so some solutions may not be found; use Reduce for complete solution information. More...

{{t → 213.754}}
```

We have found that it takes about 213 seconds for the bar to reach the desired temperature. Given the constant of proportionality, we can plot the temperature of the bar as a function of time. Here we plot it from $t = 0$ to $t = 250$ seconds.

```
(* plot the solution *)
g1 = Plot[h[t, κ], {t, 0, 250},
   AxesLabel → {"Time, m", "Temp, °F"}, PlotRange → {0, 250}];
```



So far we simply entered statements to set the constants to the values we desired. However, sometimes it is desirable to only do this temporarily. We may, for example, wish to plot the function for a specific given set of constants, but retain the abstract

form of the function for later use. We can do this by using the `Block` command. Let's ask *Mathematica* for some details about `Block`:

```
? Block
```

```
Block [{x, y, ... }, expr ] specifies that expr is to be
   evaluated with local values for the symbols x, y, ... . Block [{x =
   x0, ... }, expr ] defines initial local values for x, ... . More...
```

So we see that `Block` allows us to specify temporary or local values for the constants. Outside of the **Block**, the constants will revert back to their original values. To see how a `Block` could be used, consider the following equation:

$$\frac{\partial f}{\partial t} + t^2 f = 0 \tag{4.6}$$

The solution of this equation is:

```
ans = DSolve[{∂t f[t] + t^2 f[t] == 0, f[0] == γ}, f[t], t]
```

```
{{f[t] → e^(-t^3/3) γ}}
```

There is one undetermined constant, the initial condition γ. To see how the `Block` command can be used, we consider the case where $f(0) = 5$, and plot the resulting solution.

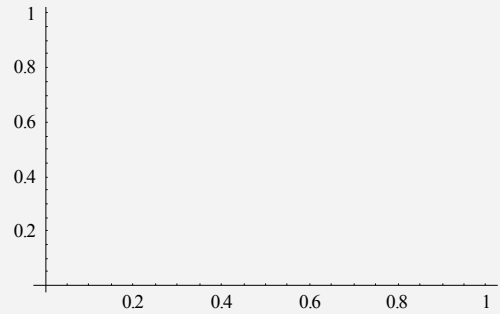
```
(* poor discipline *)
Block[{γ = 5},
  r[t] = ans[[1, 1, 2]]
    Plot[Evaluate[r[t]], {t, 0, 3}]
 ];
```

```
Plot::plnr :
  5 e^(-t^3/3) (-Graphics-) is not a machine-size real number at t = 1.25`*^-7 . More...

Plot::plnr : 5 e^(-t^3/3) (-Graphics-) is not a
    machine-size real number at t = 0.12170097471874736` . More...

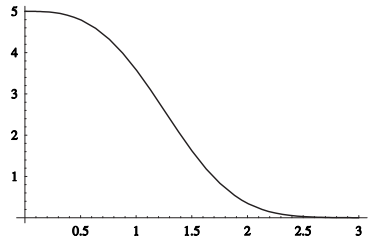
Plot::plnr : 5 e^(-t^3/3) (-Graphics-) is not a
    machine-size real number at t = 0.2544263995781211` . More...

General::stop :
  Further output of Plot::plnr will be suppressed during this calculation . More...
```



What happened? The problem is we did not include semicolons after each statement in the `Block`. Appending the semicolons will allow the `Block` to execute correctly:

```
(* use semicolons at the end of each line *)
Block[{γ = 5},
  r[t] = ans[[1,1,2]];
  g1 = Plot[Evaluate[r[t]], {t, 0, 3}]
 ];
```



4.1.2 Example: radioactive decay

Consider a 2 gram amount of a certain radioactive material. After 4 days, the material has lost 15% of its active mass. Radioactive materials decay at a rate that is proportional to the amount present in the sample. Find an expression that gives the amount of material present in the sample at time t, find the mass of the material after 1 day has passed, and find the material's half-life.

Radioactive decay is governed by the simple equation:

$$\frac{dy}{dt} = -ky \tag{4.7}$$

We use `DSolve` to find a solution to this equation, calling the initial mass M:

```
(* radiocative decay law, bad syntax *)
dsol1 = DSolve[{y'[t] == -k y[t], y[0] = M}, y[t], t]

DSolve::deqn : Equation or list of equations expected
    instead of M in the first argument {y'[t] == -k y[t], M}. More...

DSolve::deqn : Equation or list of equations expected
    instead of M in the first argument {y'[t] == -k y[t], M}. More...

DSolve[{y'[t] == -k y[t], M}, y[t], t]
```

The source of this error is the fact that we only used a single equal sign when specifying the initial condition, M. Lets try again with the correct syntax:

```
(* radiocative decay law *)
dsol1 = DSolve[{y'[t] == -k y[t], y[0] == M}, y[t], t]

{{y[t] → e^(-k t) M}}
```

Again, we can extract the solution of the equation by writing `dsol1[[1,1,2]]` and set the initial mass M to 2 grams:

```
(* specify an active mass *)
M = 2;
(* harvest the solution *)
dres1[t_, k_] = dsol1[[1,1,2]]
```

$$2\, e^{-k\,t}$$

This error message will cause problems later. The problem is the name we have assigned the function, `dres1`. Lets try calling it `h` instead:

```
(* specify an active mass *)
M = 2;
(* harvest the solution *)
h[t_, k_] = dsol1[[1,1,2]]
```

$$2\, e^{-k\,t}$$

We are told that after 4 days, the sample has lost 15% of its mass. This information allows us to find the decay constant κ, by using the `Solve` command:

```
(* call the remaining mass the reduced mass μ *)
μ = (1 - 0.15) M;        (* 15 % of active mass is gone *)
Print["After 4 days the active mass is ", μ, " g"];
(* find the decay constant *)
k0 = Solve[h[4, k] == μ, k] // N
(* load numerical value into κ *)
κ = k0[[1,1,2]]
```

```
After 4 days the active mass is 1.7 g
```

```
⚠ Solve::ifun : Inverse functions are being used by Solve, so some solutions may not be found; use Reduce for complete solution information. More...

{{k → 0.0406297}}
```

```
0.0406297
```

With the decay constant in hand, we can find the amount of the substance present at any time t. We substitute $t = 1$ day and the value of κ found above for the decay constant to report the result:

```
Print["After 1 day the active mass is ", h[1, κ], " g"];
```

```
After 1 day the active mass is 1.92037 g
```

In a decay problem it can be useful to display the amount of the active substance remaining after a given number of days in a table. Let's say we want to display the amount of the substance in grams on each day for 20 days.

```
(* table showing the active mass at the end of each day *)
T = Table[h[i, κ], {i, 20}]
```

```
{1.92037, 1.84391, 1.77049, 1.7, 1.63231, 1.56732, 1.50492,
 1.445, 1.38747, 1.33222, 1.27918, 1.22825, 1.17935, 1.13239,
 1.0873, 1.04401, 1.00244, 0.962532, 0.924208, 0.887411}
```

The current format of the table isn't very professional. The `TableForm` command allows us to put the table in a presentable format. First, we change the table so that it prints the day and amount of the substance remaining on that day together as a pair:

```
(* new table: active mass at the end of each day and the day *)
T = Table[{i, h[i, κ]}, {i, 20}]
(* more presentable form *)
TableForm[Prepend[T, {"t, days", " active mass, g"}]]
```

```
{{1, 1.92037}, {2, 1.84391}, {3, 1.77049}, {4, 1.7},
 {5, 1.63231}, {6, 1.56732}, {7, 1.50492}, {8, 1.445},
 {9, 1.38747}, {10, 1.33222}, {11, 1.27918}, {12, 1.22825},
 {13, 1.17935}, {14, 1.13239}, {15, 1.0873}, {16, 1.04401},
 {17, 1.00244}, {18, 0.962532}, {19, 0.924208}, {20, 0.887411}}
```

```
t, days      active mass, g
1            1.92037
2            1.84391
3            1.77049
4            1.7
5            1.63231
6            1.56732
7            1.50492
8            1.445
9            1.38747
10           1.33222
11           1.27918
12           1.22825
13           1.17935
14           1.13239
15           1.0873
16           1.04401
17           1.00244
18           0.962532
19           0.924208
20           0.887411
```

4.1.3 Higher order differential equations

Mathematica can also be used to solve differential equations involving second order and higher derivatives. As a simple example, consider:

$$\frac{d^2 f}{dx^2} + f(x) = 0 \tag{4.8}$$

The solution to this equation is found in the same way as the solution to a first order equation by using `DSolve`:

```
(* second order equations are no problem *)
DSolve[f''[x] + f[x] == 0, f[x], x]

{{f[x] → C[1] Cos[x] + C[2] Sin[x]}}
```

Second order equations, which will have two undetermined constants, require that the initial condition be specified for both the function and its derivative. The initial condition for the derivative of the function can be added immediately after the initial condition for the function, usually enclosed in curly braces { } with the equation. For our example, let's suppose that *f(0)* = 1 and *f''(0)* = 1. We can enter the equation in the following way:

```
(* second order equation with initial conditions *)
sol1 = DSolve[{f''[x] + f[x] == 0, f[0] == 1, f'[0] == 1}, f[x], x]

{{f[x] → Cos[x] + Sin[x]}}
```

The solution is extracted in exactly the same way as we have been doing with first order equations. Here we try to plot, but get an error.

```
(* extract the solution to the 2nd order ODE and plot *)
f1[x] = sol1[[1,1,2]]
 (* try to plot this result *)
 g1 = Plot[f1[x], {x, 0, 2 π}];

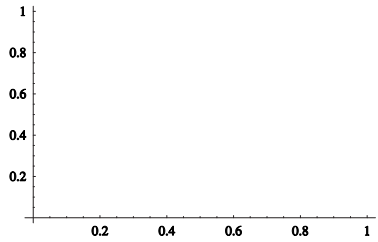
Cos[x] + Sin[x]
```

```
⚠ Plot::plnr : f1[x] is not a machine-size real number at x = 2.617993877991494`*^-7 . More...

⚠ Plot::plnr : f1[x] is not a machine-size real number at x = 0.25488992540742256` . More...

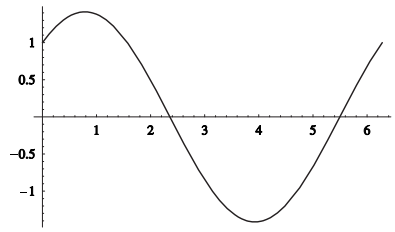
⚠ Plot::plnr : f1[x] is not a machine-size real number at x = 0.5328694051959509` . More...

⚠ General::stop : Further output of Plot::plnr will be suppressed during this calculation . More...
```



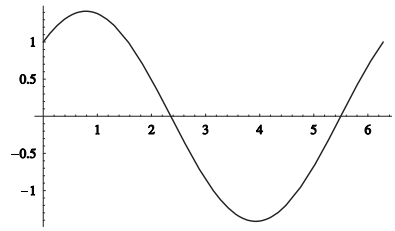
One way to get around the error is to use `Evaluate`:

```
(* force evaluation *)
g1 = Plot[Evaluate[f1[x]], {x, 0, 2 π}];
```



Another way is to use the fact that the solution is stated as a substitution rule.

```
(* a more direct approach: use the substitution rule *)
g1 = Plot[f[x] /. Flatten[sol1], {x, 0, 2 π}];
```



where we have used `Flatten` to remove a superfluous level of structure.

Example: Using different coefficients
Find the solution of:

$$\frac{dy}{dx} + \pi y = \cos x \,. \tag{4.9}$$

Plot the result setting the arbitrary constant C = 0, 2, and 4.

First, let's solve the equation. Since it is a second order equation, there will be one undetermined constant.

```
(* solve the second order equation *)
soln = DSolve[y'[x] + π y[x] == Cos[x], y[x], x];
soln[[1,1,2]]
```

$$e^{-\pi x} C[1] + \frac{\pi \text{Cos}[x] + \text{Sin}[x]}{1 + \pi^2}$$

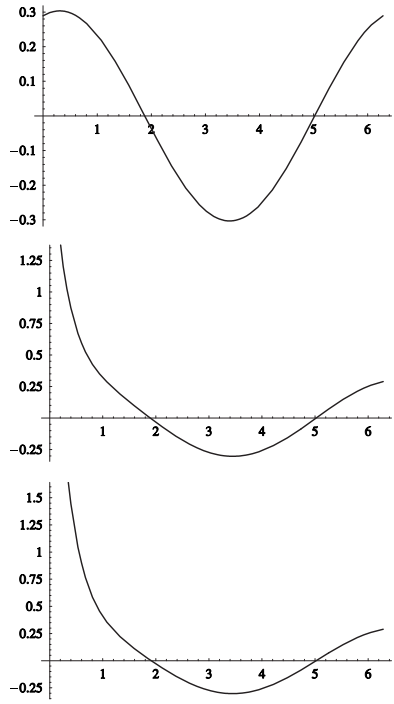
To evaluate the solution for different values of the constant, we can use the `ReplaceAll` command (/.). For example, to set the constant to C = -4, we write:

```
(* substitute for C *)
soln[[1,1,2]] /. C[1] → -4
```

$$-4\, e^{-\pi x} + \frac{\pi \text{Cos}[x] + \text{Sin}[x]}{1 + \pi^2}$$

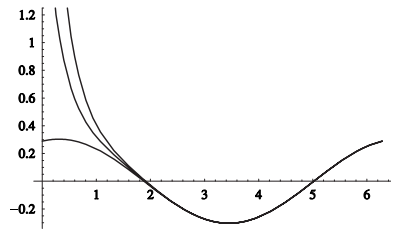
We can create separate plots for each value of the constant by using a `While` loop:

```
(* control variable *)
i = 0;
(* begin loop *)
While[i < 5,
  (* substitute for C and plot the solution *)
  g1 = Plot[soln[[1,1,2]] /. C[1] → i, {x, 0, 2 π}];
  (* increment control variable *)
  i = i + 2;
 ];
```



To plot the results together on the same graphic, we create a table and then plot the table. Be sure to use `Evaluate` when plotting with the `Table` command.

```
(* create a table of solutions *)
T = Table[soln[[1,1,2]] /. C[1] → i, {i, 0, 4, 2}];
(* plot the solutions *)
g1 = Plot[Evaluate[T], {x, 0, 2 π}];
```



Often when solving differential equations, the equation at hand is nonhomogeneous. In these cases, it is sometimes desired to obtain the homogeneous and particular solutions. As an example, consider the equation:

$$u'' + 5u = te^{-t} \tag{4.10}$$

To find the homogeneous equation, we set it equal to zero. The homogeneous equation in this case is:

$$u'' + 5u = 0 \tag{4.11}$$

One way to separate out the two solutions is to find the solution to the homogeneous equation and subtract that from the general solution to obtain the particular solution. Let's try that in this case.

```
(* general solution *)
ugen = DSolve[u''[t] + 5u[t] == te^-t, u[t], t];
```

```
(* homogenous solution *)
uhom = DSolve[u''[t] + 5u[t] == 0, u[t], t];
```

```
(* load these solutions into functions *)
UG[t_] := ugen[[1,1,2]];
UH[t_] := uhom[[1,1,2]];
```

```
(* particular solution = general solution - homogenous solution *)
UP[t_] := UG[t] - UH[t];
```

```
(* show results *)
Print["general solution: ", UG[t]];
Print["homogenous solution: ", UH[t]];
Print["particular solution: ", UP[t]];
```

```
general solution :
C[1] Cos[√5 t] + C[2] Sin[√5 t] + (te^-t (Cos[√5 t]^2 + Sin[√5 t]^2)) / (5 + Log[te]^2)

homogenous solution : C[1] Cos[√5 t] + C[2] Sin[√5 t]

particular solution : (te^-t (Cos[√5 t]^2 + Sin[√5 t]^2)) / (5 + Log[te]^2)
```

The form of the particular solution can obviously be simplified since:

$$\sin^2 x + \cos^2 x = 1 \tag{4.12}$$

We can then check to see that it does give us back te^{-t} and plot the solution.

```
(* simplify the particular solution *)
UP1[t_] = Simplify[UP[t]]
```

$$\frac{te^{-t}}{5 + \text{Log}[te]^2}$$

A descriptive print and a plot will help clarify the problem and solution.

```
(* print a statement of solution *)
Print["d^2 u_p/dt^2 + 5u_p = ", Simplify[∂_{t,t} UP1[t] + 5 UP1[t]]];
```

$$\frac{d^2 u_p}{dt^2} + 5u_p = te^{-t}$$

We are having trouble with the plot.

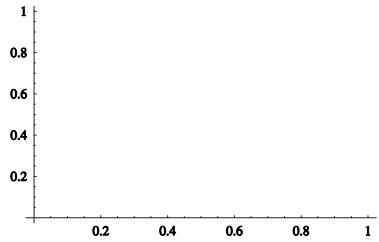
```
(* plot the particular solution *)
g1 = Plot[UP1[t], {t, 0, 1}];
```

```
⚠ Plot::plnr : UP1[t] is not a machine-size real number at t = 4.166666666666666`*^-8 . More...

⚠ Plot::plnr : UP1[t] is not a machine-size real number at t = 0.04056699157291579` . More...

⚠ Plot::plnr : UP1[t] is not a machine-size real number at t = 0.08480879985937367` . More...

⚠ General::stop : Further output of Plot::plnr will be suppressed during this calculation . More...
```



What is the problem? The error message is quite helpful. It says that we are not returning numerical values from our function. So we will test a case.

```
(* what is the problem? *)
UP1[0]
```

$$\frac{1}{5 + \text{Log[te]}^2}$$

The value of 0 was clearly not substituted for *t*. After all, we see *t* explicitly. The problem is that we made a typo early on. Instead of typing the desired:

```
t e^-t
```

we instead missed the space and create a variable `te`

```
te^-t
```

The good news is that the fix is trivial because we used the `SetDelay` operator, `:=`. Once we fix the source statement, the other definitions will perform the assignments when they are called, and this is after we have fixed things.

```
(* we need to fix a typo here *)
ugen = DSolve[u''[t] + 5 u[t] == t e^-t, u[t], t];
```

With the patch made, we immediately find correct answers throughout the calculation chain.

```
(* show results *)
Print["general solution: ", UG[t]];
Print["homogenous solution: ", UH[t]];
Print["particular solution: ", UP[t]];
```

general solution :

$$C[1]\,\text{Cos}\left[\sqrt{5}\,t\right] + C[2]\,\text{Sin}\left[\sqrt{5}\,t\right] + \frac{1}{18}\,e^{-t}\,(1+3t)\left(\text{Cos}\left[\sqrt{5}\,t\right]^2 + \text{Sin}\left[\sqrt{5}\,t\right]^2\right)$$

homogenous solution : $C[1]\,\text{Cos}\left[\sqrt{5}\,t\right] + C[2]\,\text{Sin}\left[\sqrt{5}\,t\right]$

particular solution : $\frac{1}{18}\,e^{-t}\,(1+3t)\left(\text{Cos}\left[\sqrt{5}\,t\right]^2 + \text{Sin}\left[\sqrt{5}\,t\right]^2\right)$

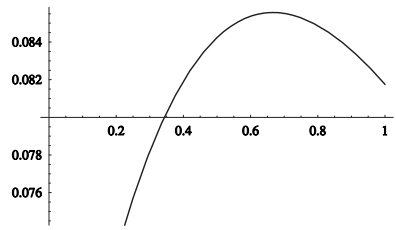
With a little polish we will be ready to plot the particular solution.

```
(* since we used the SetDelayed command our equations update *)
(* automatically and there is no need to retype or re-evaluate *)
UP1[t_] = Simplify[UP[t]]
```

$$\frac{1}{18}\,e^{-t}\,(1+3t)$$

We are ready for plotting.

```
(* plot the particular solution *)
g1 = Plot[UP1[t], {t, 0, 1}];
```



Exercises:
Solve the following differential equation:

$$\frac{dy}{dt} + 8y = \sin t \tag{4.13}$$

for three different cases:

4.4 $y(\pi) = 0$

4.5 $y(\pi) = -1$

4.6 $y(\pi) = 2$

Electrical circuit problem

Consider an inductor resistor (L-R) circuit driven by an AC source. The equation describing the time behavior of the current i(t) is:

$$R\, i(t) + L\frac{di}{dt} = \varepsilon \sin(\omega t) \tag{4.14}$$

Solve this equation for R = 100 ohms, L = 10 H, and ε = 10. Plot the results on separate graphs for ω = 100, 500, 1000 Hz. Assume that $i(0) = 0$.

First, we solve the differential equation with the given initial condition:

```
(* equation for an R-L circuit *)
s1 = DSolve[{R i[t] + L i'[t] == ϵ Sin[ω t], i[0] == 0}, i[t], t]
```

$$\left\{\left\{i[t] \to -\frac{e^{-10\,t}\,(-\omega + e^{10\,t}\,\omega\,\text{Cos}[t\,\omega] - 10\,e^{10\,t}\,\text{Sin}[t\,\omega])}{100 + \omega^2}\right\}\right\}$$

Next, we define the constants specified in the problem and define the function `cur[t]` that we will use to store the solution:

```
(* specify parameters *)
ϵ = 10; R = 100; L = 10;
(* load the solution into a function *)
cur[t_] = s1[[1,1,2]]
```

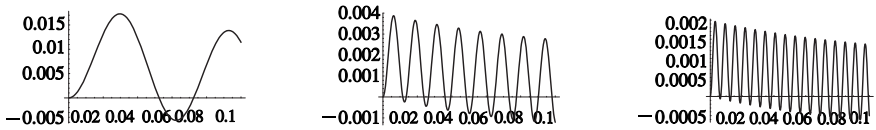
$$-\frac{e^{-10\,t}\,(-\omega + e^{10\,t}\,\omega\,\text{Cos}[t\,\omega] - 10\,e^{10\,t}\,\text{Sin}[t\,\omega])}{100 + \omega^2}$$

To plot the solution for three given frequencies, we will form a `GraphicsArray` object using the `ReplaceAll` operator `/.` to specify the frequency ω in each case. Each plot can be set up in the usual way with the additional requirement that we add the replacement operator to the input slot where we specify the function to plot. The plots are created in this manner:

```
(* plot for ω = 100 *)
plota = Plot[cur[t] /. ω → 100, {t, 0, 0.1}, DisplayFunction → voff];
(* plot for ω = 500 *)
plotb = Plot[cur[t] /. ω → 500, {t, 0, 0.1}, DisplayFunction → voff];
(* plot for ω = 1000 *)
plotc = Plot[cur[t] /. ω → 1000, {t, 0, 0.1}, DisplayFunction → voff];
```

Next, we set up our graphics array. Each plot name we have assigned is entered into the graphics array by enclosing it in curly braces. To display the plots, we use the `Show` command.

```
(* create the graphics array *)
ga = GraphicsArray[{plota, plotb, plotc}];
(* show the graphics array *)
g1 = Show[ga, ImageSize → 5 72];
```

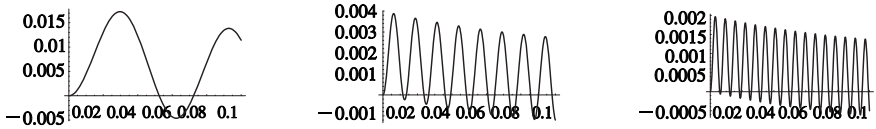


An equivalent solution for this problem is to create the graphics objects in the list rather than placing them in the list at the end. This would be preferable if you had a large number of ωvalues to process.

```
(* specify the ω values in a list and measure the list *)
Ω = {100, 500, 1000}; nΩ = Length[Ω];
(* create a list of graphics objects *)
plots = Table[Plot[cur[t] /. ω → Ω[[i]],
    {t, 0, 0.1}, DisplayFunction → voff], {i, nΩ}];
```

The plots are now packaged for the `GraphicsArray` command.

```
(* view the list as a graphics array *)
g1 = Show[GraphicsArray[plots], ImageSize→ 5 72];
```



Second order equation ay'' + by' + cy = F

As an example of this class of equation, consider the following problem:

$$y''(t) + 2y'(t) - y(t) = \cos 4t \tag{4.15}$$

with the initial conditions $y(0) = 1$ and $y'(0) = 0$.

We obtain a solution and plot for $0 \le t \le$. The solution returned by `DSolve` is quite messy:

```
sol1 = DSolve[
  {y''[t] + 2y'[t] - y[t] == Cos[4 t], y[0] == 1, y'[0] == 0}, y[t], t]
```

```
{{y[t] → (ℯ^(-Sqrt[2] t)
      (-740 ℯ^(Sqrt[2] t+(-1-Sqrt[2]) t) + 338 Sqrt[2] ℯ^(Sqrt[2] t+(-1-Sqrt[2]) t) - 740 ℯ^(Sqrt[2] t+(-1+Sqrt[2]) t) -
        338 Sqrt[2] ℯ^(Sqrt[2] t+(-1+Sqrt[2]) t) + 34 ℯ^(t+2 Sqrt[2] t+(-1-Sqrt[2]) t) Cos[4 t] +
        15 Sqrt[2] ℯ^(t+2 Sqrt[2] t+(-1-Sqrt[2]) t) Cos[4 t] + 34 ℯ^(t+(-1+Sqrt[2]) t) Cos[4 t] -
        15 Sqrt[2] ℯ^(t+(-1+Sqrt[2]) t) Cos[4 t] - 16 ℯ^(t+2 Sqrt[2] t+(-1-Sqrt[2]) t) Sin[4 t] +
        76 Sqrt[2] ℯ^(t+2 Sqrt[2] t+(-1-Sqrt[2]) t) Sin[4 t] - 16 ℯ^(t+(-1+Sqrt[2]) t) Sin[4 t] -
        76 Sqrt[2] ℯ^(t+(-1+Sqrt[2]) t) Sin[4 t]) / (4 (-19 + 2 Sqrt[2]) (19 + 2 Sqrt[2])))}}
```

Let's examine our simplification options.

```
(* first we try basic simplification *)
Simplify[sol1[[1,1,2]]]
```

$$\frac{1}{1412}\left(e^{-\sqrt{2}\,t}\left(-68\,e^{\sqrt{2}\,t}\,\text{Cos}[4\,t]+e^{-t}\left(740-338\sqrt{2}+\left(740+338\sqrt{2}\right)e^{2\sqrt{2}\,t}+32\,e^{t+\sqrt{2}\,t}\,\text{Sin}[4\,t]\right)\right)\right)$$

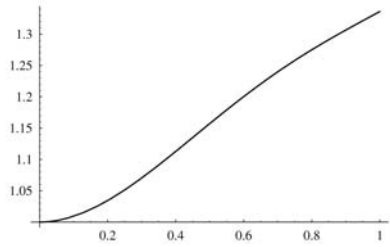
```
(* now we try full simplification *)
FullSimplify[sol1[[1,1,2]]]
```

$$\frac{1}{706}\,e^{-\left(1+\sqrt{2}\right)t}\left(370-169\sqrt{2}+\left(370+169\sqrt{2}\right)e^{2\sqrt{2}\,t}-2\,e^{t+\sqrt{2}\,t}\left(17\,\text{Cos}[4\,t]-8\,\text{Sin}[4\,t]\right)\right)$$

```
(* we select the latter simplification *)
s[t_] = %;
```

Now we can plot the result.

```
(* plot the result *)
g1 = Plot[s[t], {t, 0, 1}];
```



Mass in a gravitational field

A 2 kg mass is thrown upward with an initial velocity of 100 m/s. Find out how long it takes to hit the ground and plot the position and velocity as functions of time up to the time the object hits the ground. Find the magnitude of the velocity at impact.

This problem gives us a chance to solve a second order equation. From Newton's laws, the equation of motion is:

$$m\frac{d^2x}{dt^2} = -mg \tag{4.16}$$

As the mass terms cancel, the equation can be set up as follows:

```
(* acceleration due to gravity *)
g = 9.81;
(* F = ma *)
eq1 = x''[t] + g == 0;
```

Next, we define the initial condition, which states that the velocity is 100 m/s, and solve the equation.

```
(* initial velocity of the ball *)
v0 = 100;
(* solve the differential equation *)
s1 = DSolve[{eq1, x[0] == 0, x'[0] == v0}, x[t], t]

{{x[t] → 100 t - 4.905 t^2}}
```

The specification of the problem requires us to obtain the velocity as a function of time. Velocity *v(t)* is defined as the first derivative of the position:

$$v(t) = \frac{dx}{dt} \tag{4.17}$$

First, we extract the solution and then create a new function for the velocity by taking its derivative. Notice that we use the underscore character in the function definition to ensure that we are defining functions:

```
(* initial velocity of the ball *)
v0 = 100;
(* solve the differential equation *)
s1 = DSolve[{eq1, x[0] == 0, x'[0] == v0}, x[t], t]

{{x[t] → 100 t - 4.905 t^2}}
```

The object is at ground level when the position $x = 0$. To find the time when this occurs, we need to solve $100t - 4.905t^2 = 0$. Since this is a quadratic, there will be two solutions:

```
(* final time: ball lands on the ground *)
tf = Solve[pos[t] == 0, t]

{{t → 0.}, {t → 20.3874}}
```

The first solution, $t = 0$, naturally tells us that at the initial time the object was at ground level. The second time is more interesting and tells us that the object falls back to ground in about 20 seconds. We can extract these two solutions and store them in numerical variables which we call t_1 and t_2. From what we've seen so far about extracting solutions, we make a guess as to how to obtain the first solution:

```
(* harvest the first number, the starting time *)
t1 = tf[[1,1,2]]

0.
```

How do we obtain the second solution? A guess might be `tf[1,2,2]]`:

```
(* we want to harvest the second number *)
t2 = tf[[1,2,2]]

⚠ Part::partw : Part 2 of {t → 0.} does not exist . More…

{{t → 0.}, {t → 20.3874}}[[1, 2, 2]]
```

Unfortunately this just gives us an error telling us we have referenced a nonexistent part, sort of like trying to take the elevator to the tenth floor of a an eight-story building. In fact the correct syntax is given as follows:

```
(* this is the proper syntax *)
t2 = tf[[2,1,2]]

20.3874
```

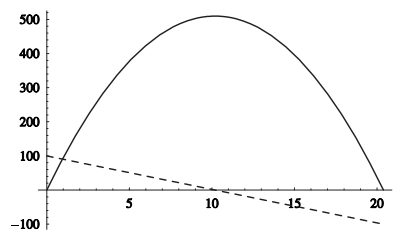
Now we have all the information we need to finish the problem. First, let's report the magnitude of the velocity when the object hits the ground. To report the magnitude, we use the `Abs` function to report the absolute value (the velocity, which will be in a direction such that it points toward the ground, will actually be negative).

```
(* state the result *)
Print["Velocity when ball hits the ground = ", vel[t2], " m/sec"]
```

```
Velocity when ball hits the ground = -100. m/sec
```

Finally, we create a plot of the position and velocity. We will include both on the same graph, so we will distinguish between the curves by using the `PlotStyle` option. We will plot the velocity with a dashed line:

```
(* plot the position and the velocity *)
g1 = Plot[{pos[t], vel[t]}, {t, t1, t2},
  PlotStyle → {GrayLevel[0], Dashing[{0.01}]}];
```



Another mass in a gravitational field: a parachute jumper with air resistance
A 150 pound man jumps out of an airplane at 10,000 feet and immediately opens his parachute. How long does it take him to reach the ground? Consider the problem with air resistance proportional to the velocity squared.

Let *x(t)* represent the altitude of the man above ground as a function of time. The equation we need to solve is a modified form of equation 4.16:

$$m\frac{dx^2}{dt^2} + \alpha\left(\frac{dx}{dt}\right) = -mg \tag{4.18}$$

where m is the mass of the parachutist, g is the acceleration due to gravity, and α is a constant of proportionality. Let us do some dimensional analysis for α. On the right-hand side, we have a mass times an acceleration, so the units are:

$$mg = [kg]\left[\frac{m}{s^2}\right] = mass\frac{length}{time^2} \tag{4.19}$$

The air resistance term must have these same units. We have α multiplied by velocity squared, which gives us:

$$\alpha\left[\frac{m^2}{s^2}\right] = \alpha\left[\frac{length^2}{time^2}\right] \tag{4.20}$$

Comparison with the above leads to the conclusion that:

$$\alpha = \left[\frac{mass}{length}\right] \tag{4.21}$$

We can divide through the equation by the mass and consider the ratio $\beta = \frac{\alpha}{m}$. For this example, we take $m = 0.2$.

First, we define some constants. We call the initial altitude of the parachutist h_0:

```
(* constant for the parachutist problem *)
g = 9.81;    (* acceleration due to gravity, m/s^2 *)
m = 150;     (* parachutist mass, kg *)
β = 0.09;    (* air resistance factor, s^-1 *)
h0 = 10000;   (* jump altitude, m *)
```

One important note is to remember when entering constants, do not insert commas. In other words, it would be incorrect to define $h_0 = 10,\ 000$. Instead we enter $h_0 = 10000$.

With the appropriate constant definitions completed, we use `DSolve` to obtain a solution of the equation. Let's restate the equation here:

$$m\frac{dx^2}{dt^2} + \alpha\left(\frac{dx}{dt}\right) = -mg \tag{4.18}$$

Following the previous example, we use prime notation to indicate the derivatives. The first and second derivatives can be correctly entered into *Mathematica* with the following notation:

```
x'[t]     (* first derivative *)
x''[t]    (* second derivative *)
```

Remember, in `DSolve`, we must use two consecutive equal signs (==) to specify the equation. So the correct syntax for the entire equation is:

```
(* equation 6.18 *)
x''[t] + β x'[t] == -g

β x'[t] + x''[t] == 0
```

Remember, always make sure that there is a space between individually defined symbols. For example, for the first derivative term, make sure to include a space after the β character. Otherwise, this would confuse *Mathematica* and cause the program to arrive at erroneous results. Examine the differences below.

```
(* correct and incorrect syntaxes *)
x''[t] + β x'[t] == 0
x''[t] + βx'[t] == 0
```

Now that we understand the correct format for the equation, we can enter it into the program. As we have seen this is done by placing the equation inside curly braces, separated by commas from the initial conditions. As this is a second order equation, there will be two initial conditions, one for the function *h(t)* and one for the first derivative *h'(t)*. We assume that the plane is flying at a constant height when the parachutist jumps out of the aircraft, and so we set *h'(0)* = 0. If we ask *Mathematica* to store the solutions in a list called `sol`, the correct syntax to enter all the data as given is:

```
(* differential equation and initial conditions *)
sol = DSolve[{x''[t] + β x'[t] == -g, x[0] == h0, x'[0] == 0}, x[t], t]

{{x[t] →
   2.71828^(-0.09 t) (-1211.11 + 11211.1 2.71828^(0.09 t) - 109. 2.71828^(0.09 t) t)}}
```

You may consider this form more heuristic.

```
(* specify the differential equation *)
diffeq = x''[t] + β x'[t] == -g;
(* specify the initial conditions *)
system = {diffeq, x[0] == h0, x'[0] == 0};
(* solve the system of equations *)
sol1 = DSolve[system, x[t], t]

{{x[t] →
   2.71828^-0.09 t (-1211.11 + 11211.1 2.71828^0.09 t - 109. 2.71828^0.09 t t)}}
```

Let's do two things with this information. The specification of the problem asked us to find out how long it will take the parachutist to reach the ground. We can find this out by solving for $x = 0$. Finally, lets make a plot of *x(t)*.

First, we need to get the solution out and assign it to a function that we can use. This is done in the standard way:

```
(* load the solution into a function *)
A[t_] = soln[[1,1,2]]

2.71828^-0.09 t (-1211.11 + 11211.1 2.71828^0.09 t - 109. 2.71828^0.09 t t)
```

Another way to load the function is longer, but it specifically shows us that we are harvesting a solution for `x[t]`.

```
(* another way to load the function *)
A[t_] = x[t] /. Flatten[soln]

2.71828^-0.09 t (-1211.11 + 11211.1 2.71828^0.09 t - 109. 2.71828^0.09 t t)
```

Notice that the arrow `x[t]` ⟶ is no longer included in the representation of the function. This means that we now have a function that we can use to do things like plot the results. Now we can ask *Mathematica* to give us the jumper's altitude at any given time such as when the parachute is first opened:

```
(* initial altitude *)
A[0]

10000.
```

To find the time that the parachutist will reach the ground, which is located at height $h = 0$, we can use `Solve` to find a solution for `A[t] = 0`. We store the result in a variable called `tfinal`:

```
(* solve for the time when the jumper lands *)
tfinal = Solve[A[t] == 0, t]

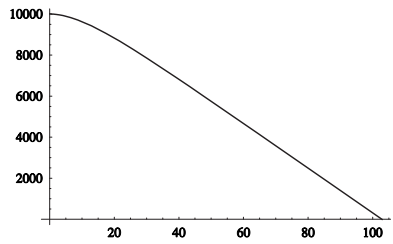
⚠ InverseFunction ::ifun : Inverse functions are
    being used. Values may be lost for multivalued inverses. More...

⚠ Solve::ifun :
  Inverse functions are being used by Solve, so some solutions may not
    be found; use Reduce for complete solution information. More...

{{t → -27.3459}, {t → 102.853}}
```

As we found in the previous example, the result is returned as a list. The first time listed is negative, and so is discarded as physically meaningless. Knowing that the time to reach ground is about 100 seconds, we can use `A[t]` to create a plot of height as a function of time in seconds:

```
(* plot the altitude from the time the chute opens until landing *)
g1 = Plot[A[t], {t, 0, 103}];
```



Suppose that we wanted to access the solutions returned by `Solve` directly. How can we do it? `Solve` has returned a nested list with two entries. Seeing that the entry of interest is located in the second position, we might try this:

```
(* trying to extract the total time *)
total_time = tfinal[[1,1,2]]

-27.3459
```

Changing the index used allows us to access the other solution, the total time required to reach the ground:

Exercise: **A mass-spring system**

4.7 Use *Mathematica* to obtain a solution to the following problem. Find the displacement *y(t)* for a mass *m* hanging from a spring of length *l* is described by the following equation:

$$m\frac{d^2y}{dt^2} + k\,y(t) = 0 \tag{4.22}$$

The constant *k* is determined from the relation $k = \frac{mg}{\Delta l}$, where Δl is the extension of the spring when attaching the mass *m* to its bottom. The initial position is given as *y(0)* = 5 cm and we take the initial velocity $\frac{dy}{dt}$ at time *t* = 0 to be $\frac{dy}{dt} = 0$. Use SI units where g = 9.81 m/s^2.

4.1.4 The Dirac delta function

For our next few examples, we consider differential equations involving the Dirac delta function. Suppose that we have:

$$\frac{df}{dx} + \sin x = \delta(x) \tag{4.23}$$

where *δ(x)* is the Dirac delta function we saw in chapter 3. To enter the Dirac delta, we simply type `DiracDelta[x]`. The equation is easily solved:

```
(* equation 6.23 *)
dsol = DSolve[∂x f[x] + Sin[x] == DiracDelta[x], f[x], x]

{{f[x] → C[1] + Cos[x] + UnitStep[x]}}
```

```
(* fix the arbitrary constant at 2 *)
sol = dsol⟦1,1,2⟧ /. C[1] → 2

2 + Cos[x] + UnitStep[x]
```

Now suppose that we want to evaluate the solution at a particular value of x, say $x = 0$. If we try to write `sol[0]`, we get nonsensical output:

```
sol[0]

(2 + Cos[x] + UnitStep[x])[0]
```

The correct way to extract a functional evaluation is to apply the `ReplaceAll` operator:

```
sol /. x→ 0

4
```

Delta Function

Example: Solve y"(t) - 2y(t) = δ(x-3)
We begin by defining the function:

```
(* define the differential equation *)
deq1 = y''[t] - 2 y[t] == DiracDelta[t - 3];
```

Next, we use **DSolve** to get the solution:

```
sol1 = DSolve[deq1, y[t], t]

{{y[t] → ℯ^(√2 t) C[1] + ℯ^(-√2 t) C[2] +
    (ℯ^(-3√2 - √2 t) (-ℯ^(6√2) + ℯ^(2√2 t)) UnitStep[-3 + t]) / (2√2)}}
```

Now we need to get the solution into a form that we can use to generate the plot. Up until now we have been using the notation `sol[[1,1,2]]`. Alternatively, we can use the icon found in the `BasicInput` palette which you saw in chapter 1. To do this correctly, we need to add three levels. This is done by clicking on the icon three times, without moving your cursor. When this is done correctly, your screen should look like this:

```
(* the Part command *)
□[[□]][[□]][[□]]
```

For the current example, we create a function g:

```
(* fix the free parameters, load the solution into a function *)
g[t] = sol1[[1]][[1]][[2]] /. C[1] → 0 /. C[2] → 1
```

$$e^{-\sqrt{2}\,t} + \frac{e^{-3\sqrt{2}-\sqrt{2}\,t}\left(-e^{6\sqrt{2}} + e^{2\sqrt{2}\,t}\right)\text{UnitStep}[-3+t]}{2\sqrt{2}}$$

We didn't specify any initial conditions in the problem, but let's suppose that `C[1] = 0` and `C[2] = 1`. These constants can be specified after the solution to the differential equation has been obtained by using the `ReplaceAll` operator (`/.`).

```
(* fix the free parameters, load the solution into a function *)
g[t] = sol1[[1]][[1]][[2]] /. C[1] → 0 /. C[2] → 1
```

$$e^{-\sqrt{2}\,t} + \frac{e^{-3\sqrt{2}-\sqrt{2}\,t}\left(-e^{6\sqrt{2}} + e^{2\sqrt{2}\,t}\right)\text{UnitStep}[-3+t]}{2\sqrt{2}}$$

Now we can plot the solution.

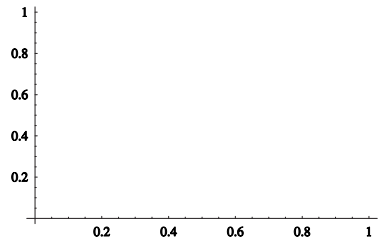
```
(* plot the specified solution *)
g1 = Plot[g[t], {t, 0, 10}];

⚠ Plot::plnr :
  g[t] is not a machine-size real number at t = 4.1666666666666667`*^-7 . More...

⚠ Plot::plnr :
  g[t] is not a machine-size real number at t = 0.40566991572915795` . More...

⚠ Plot::plnr :
  g[t] is not a machine-size real number at t = 0.8480879985937368` . More...

⚠ General::stop :
  Further output of Plot::plnr will be suppressed during this calculation . More...
```



The plot did not work. We look at the error message and see that we are not generating numbers so we try an experiment.

```
(* check the function *)
g[0]

g[0]
```

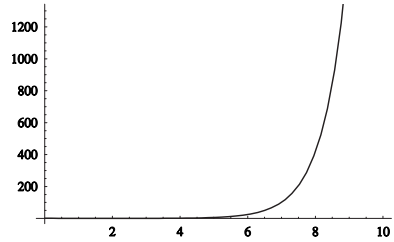
Will `Evaluate` help us again?

```
(* Evaluate to the rescue *)
Evaluate[g[t] /. t→ 0]

1
```

So we need to wrap our function with an `Evaluate` wrapper.

```
(* plot the specified solution *)
g1 = Plot[Evaluate[g[t]], {t, 0, 10}];
```



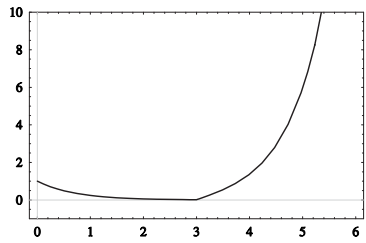
Another way to get around the problem that we had with the plot is to define `g(t)` as a function, by including the underscore character after *t*, that is, write `g[t_]` instead of `g[t]`. We illustrate this here:

```
(* fix the free parameters, load the solution into a function *)
g[t_] = sol1[[1]][[1]][[2]] /. C[1] → 0 /. C[2] → 1
```

$$e^{-\sqrt{2}\,t} + \frac{e^{-3\sqrt{2}-\sqrt{2}\,t}\left(-e^{6\sqrt{2}} + e^{2\sqrt{2}\,t}\right)\text{UnitStep}[-3+t]}{2\sqrt{2}}$$

By way of demonstration, we will use this to zoom in on the behavior of function near $t = 0$.

```
(* plot the specified solution *)
g1 = Plot[g[t], {t, 0, 6}, Frame → True, Axes → True,
   AxesStyle → GrayLevel[0.8], PlotRange → {-1, 10}];
```



Exercise: The Dirac delta function in a differential equation

4.8 Solve the following equation:

$$y''(x) + 9y(x) = \delta\left(x - \frac{1}{2}\right) \tag{4.24}$$

with the initial conditions $y(0) = 0$ and $y'(0) = 0$. Plot the solution for $0 \leq x \leq \pi$.

Example: Boundary value problems

The student is often confronted by boundary value problems, or problems that involve conditions on the solution or the derivative at two or more points. In this section we will try several examples, beginning with the following simple problem. Find *u(x)* given that:

$$u''(x) + u(x) = 0 \tag{4.25}$$

where $u(0) = u(\pi) = 0$.

To solve the equation, we simply add the second boundary condition to `DSolve`:

```
(* solve the differential equation with boundary values *)
soln = DSolve[{u''[x] + u[x] == 0, u[0] == 0, u[π] == 0}, u[x], x]

{{u[x] → C[2] Sin[x]}}
```

The constant in this equation is completely arbitrary.

Example: Heat conduction in a cylinder

In this example we take our first look at Laplace's equation. Here we consider heat conduction through a cylinder such that there is no variation of temperature in either the *z* or angular directions. Letting *T(r)* represent the temperature which varies with radius, in cylindrical coordinates, Laplace's equation for *T* is given by (noting that there is no (z, ϕ) dependence:

$$\nabla^2 T = \frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial T}{\partial r}\right) = 0 \tag{4.26}$$

Suppose that the cylinder has inner radius *a* and outer radius *b*, and that it is subject to the boundary conditions $T(a) = A$ and $T(b) = B$.

Find the solution for *T(r)* and plot this solution for the following scenario:

inner radius	$a = 2$ m
outer radius	$b = 4$ m
inner temperature	$A = 50$°K
outer temperature	$B = 100$°K

We begin by obtaining the solution with `DSolve`:

```
(* solve the differential equation with boundary values *)
soln = DSolve[{1/r ∂r (r ∂r T[r]) == 0, T[a] == A, T[b] == B}, T[r], r]
```

$$\left\{\left\{T[r] \to \frac{B\,\text{Log}[a] - A\,\text{Log}[b] + A\,\text{Log}[r] - B\,\text{Log}[r]}{\text{Log}[a] - \text{Log}[b]}\right\}\right\}$$

The result is a doubly nested list. As a tip, you can remove one level of the list by applying the `Flatten` command. In that case we would have written:

```
(* remove one set of braces *)
soln = soln // Flatten
```

$$\left\{T[r] \to \frac{B\,\text{Log}[a] - A\,\text{Log}[b] + A\,\text{Log}[r] - B\,\text{Log}[r]}{\text{Log}[a] - \text{Log}[b]}\right\}$$

or equivalently:

```
(* another way to remove one set of braces *)
soln = Flatten[soln]
```

$$\left\{T[r] \to \frac{B\,\text{Log}[a] - A\,\text{Log}[b] + A\,\text{Log}[r] - B\,\text{Log}[r]}{\text{Log}[a] - \text{Log}[b]}\right\}$$

Notice that there is only one set of braces in the answer returned, while before the answer was returned with two sets of braces. We can create a function, which we will call `H[r]`, to access this solution in the following way:

```
(* load the solution into a function *)
H[r_] = soln[[1]]
```

$$T[r] \to \frac{-200\,\text{Log}[4] + 200\,\text{Log}[r]}{\text{Log}[2] - \text{Log}[4]}$$

Now we set the parameters a, b, A, and B to the values specified in the problem description and plot.

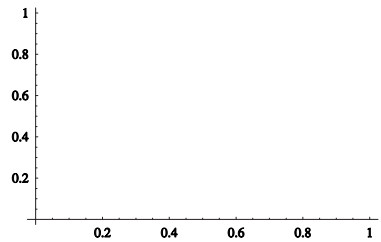
```
(* set the parameter values *)
a = 2; b = 4; A = 200; B = 100;
(* plot the solution function *)
g1 = Plot[H[r], {r, 2, 4}];
```

```
⚠ Plot::plnr :
  H[r] is not a machine-size real number at r = 2.000000083333333` . More...

⚠ Plot::plnr :
  H[r] is not a machine-size real number at r = 2.0811339831458313` . More...

⚠ Plot::plnr :
  H[r] is not a machine-size real number at r = 2.1696175997187472` . More...

⚠ General::stop :
  Further output of Plot::plnr will be suppressed during this calculation . More...
```



Why didn't this work? We look at the error message and then check the function.

```
(* check the function *)
H[2]
```

$$\left\{T[2] \to \frac{200\,\text{Log}[2] - 200\,\text{Log}[4]}{\text{Log}[2] - \text{Log}[4]}\right\}$$

While we can see a number on the right-hand side, we do not have a number. We have a rule.

```
(* check the function *)
H[2]
```

$$\left\{T[2] \to \frac{200\,\text{Log}[2] - 200\,\text{Log}[4]}{\text{Log}[2] - \text{Log}[4]}\right\}$$

We have not correctly accessed the solution. This time we will use the `Part` icon from the `BasicInput` palette as before. First, we click the icon once to place it in the notebook:

```
(* input from the BasicInput palette *)
(* one click *)
□[[□]]
```

We select the icon a second time, allowing us to access two levels:

```
(* select the field you just entered and click again *)
□[[□]][[□]]
```

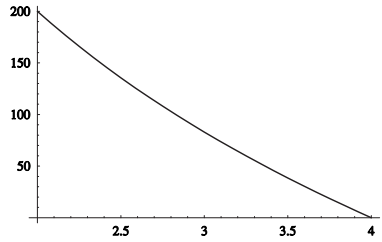
Now we can use this to create a function in *Mathematica*:

```
(* load the proper part from soln *)
H[r_] = soln[[1]][[2]]
```

$$\frac{-200\,\text{Log}[4] + 200\,\text{Log}[r]}{\text{Log}[2] - \text{Log}[4]}$$

Now that we have `H[r]` in hand, we can generate the plot:

```
(* plot the solution function *)
g1 = Plot[H[r], {r, 2, 4}];
```



Example: Critically damped system
Next we consider the following damped mass-spring system:

$$y'' + 3y' + 7y = 0 \text{ where } \frac{dy}{dt} = a(t) \tag{4.27}$$

with two sets of initial conditions. The first set is $y(0) = 1$, $a(0) = y'(0) = 2$. The second set is $y(0) = 1$, $a(0) = y'(0) = 3$. Let's make a phase plot for the system.

We begin by using `DSolve` to obtain a solution for the first set of initial conditions:

```
(* damped mass-spring system *)
diffeq = y''[t] + 3y'[t] + 7y[t] == 0;
(* 1st set of initial conditions *)
mssol1 = DSolve[{diffeq, y[0] == 1, y'[0] == 2}, y[t], t]
```

$$\left\{\left\{y[t] \to \frac{1}{19} e^{-3t/2}\left(19\,\text{Cos}\left[\frac{\sqrt{19}\,t}{2}\right] + 7\sqrt{19}\,\text{Sin}\left[\frac{\sqrt{19}\,t}{2}\right]\right)\right\}\right\}$$

Now we extract the solution, which we call `sp1(t)`:

```
(* load the first solution into a function *)
sp1[t_] = mssol1[[1]][[1]][[2]]
```

$$\frac{1}{19} e^{-3t/2}\left(19\,\text{Cos}\left[\frac{\sqrt{19}\,t}{2}\right] + 7\sqrt{19}\,\text{Sin}\left[\frac{\sqrt{19}\,t}{2}\right]\right)$$

Next, we obtain the solution based on the second set of initial conditions:

```
(* 1st set of initial conditions *)
mssol2 = DSolve[{diffeq, y[0] == 0, y'[0] == 3}, y[t], t]
```

$$\left\{\left\{y[t] \to \frac{6\, e^{-3t/2} \sin\left[\frac{\sqrt{19}\, t}{2}\right]}{\sqrt{19}}\right\}\right\}$$

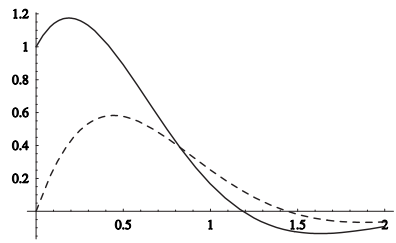
and harvest the solution.

```
(* load the second solution into a function *)
sp2[t_] = mssol2[[1]][[1]][[2]]
```

$$\frac{6\, e^{-3t/2} \sin\left[\frac{\sqrt{19}\, t}{2}\right]}{\sqrt{19}}$$

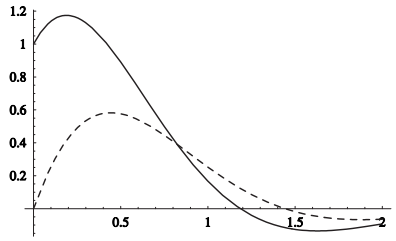
Let's plot these two functions to examine their behavior. We include both functions on the same graph.

```
(* plot these two solutions on the same graph *)
g1 = Plot[{sp1[t], sp2[t]}, {t, 0, 2},
   PlotStyle → {Dashing[{0.01, 0}], Dashing[{0.01, 0.01}]}];
```



Notice how easy it easy to use the graphics primitive text to label these curves.

```
(* label the curves *)
gt1 = Graphics[Text["ic 1", {0.6, 1}]];
gt2 = Graphics[Text["ic 2", {0.2, 0.6}]];
g2 = Show[g1, gt1, gt2];
```



Example: Systems of equations and phase plots

A frequent task that is encountered when solving differential equations is the construction of a phase plot. In this section we will demonstrate a way to do this in *Mathematica* by solving the following set of equations:

$$x' = -x \tag{4.28}$$

$$y' = 4x + y \tag{4.29}$$

We begin in the usual way by generating the solution with `DSolve`. To include multiple equations, you put the equations inside one set of curly braces as before, and separate with commas. So we would write the current set of equations that we are interested in the following manner:

```
(* combine the systems *)
{x'[t] == -x[t], y'[t] == 4 x[t] + y[t]}
```

Perhaps you will find this form easier to read:

```
(* first system *)
sys1 = x'[t] == -x[t];
(* second system *)
sys2 = y'[t] == 4 x[t] + y[t];
(* specify the functions *)
fcns = {x[t], y[t]};
```

Admittedly, we lose real estate, but we gain clarity. You must also tell *Mathematica* that you are solving for two functions; this is also done by enclosing them inside curly braces, separated by commas and assigning them to the list `fcns`.

The solution statement takes this form:

```
(* solve the coupled systems *)
ms1 = DSolve[{sys1, sys2}, fcns, t]

{{x[t] → e^-t C[1], y[t] → 2 e^-t (-1 + e^2t) C[1] + e^t C[2]}}
```

Mathematica has provided us with two solutions inside a nested list. As before, the first solution is indexed by `[1,1,2]` and the second solution is indexed by `[1,2,2]`.

```
(* harvest the solutions *)
x1[t_] = ms1[[1]][[1]][[2]]
y1[t_] = ms1[[1]][[2]][[2]]

e^-t C[1]
```

```
2 e^-t (-1 + e^2t) C[1] + e^t C[2]
```

As expected from the definition of the problem, we have two undetermined constants. It is often desirable to examine a phase plot of the solution for different values of the constants. We will do that here examining the constants over the range {-2, -1, 0, 1, 2}. It is helpful to store the results in a table, so we will generate a table and use the `ReplaceAll` operator /. to substitute the appropriate values for the constants:

```
(* families of solutions *)
ftable =
 Table[{x1[t], y1[t]} /. {C[1] → i, C[2] → j}, {i, -1, 1}, {j, -1, 1}]

{{{-e^-t, -e^t - 2 e^-t (-1 + e^2t)},
  {-e^-t, -2 e^-t (-1 + e^2t)}, {-e^-t, e^t - 2 e^-t (-1 + e^2t)}},
 {{0, -e^t}, {0, 0}, {0, e^t}}, {{e^-t, -e^t + 2 e^-t (-1 + e^2t)},
  {e^-t, 2 e^-t (-1 + e^2t)}, {e^-t, e^t + 2 e^-t (-1 + e^2t)}}}
```

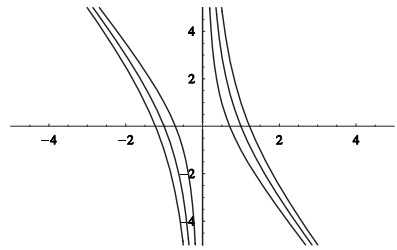
Immediately we notice an extra level of braces that need to be removed since the list of functions needs to be linear, (i.e. indexed by a single parameter. `Flatten` does the job once more).

```
(* remove one level of braces *)
fftable = Flatten[ftable, 1]
```

$$\{\{-e^{-t}, -e^{t} - 2e^{-t}(-1+e^{2t})\}, \{-e^{-t}, -2e^{-t}(-1+e^{2t})\}, \{-e^{-t}, e^{t} - 2e^{-t}(-1+e^{2t})\}, \{0, -e^{t}\}, \{0, 0\}, \{0, e^{t}\}, \{e^{-t}, -e^{t} + 2e^{-t}(-1+e^{2t})\}, \{e^{-t}, 2e^{-t}(-1+e^{2t})\}, \{e^{-t}, e^{t} + 2e^{-t}(-1+e^{2t})\}\}$$

A parametric plot can now be created showing the integral curves of the solution.

```
(* parametric plot of the flattened ftable *)
g1 = parplot = ParametricPlot[Evaluate[fftable],
    {t, -2, 2}, PlotRange → {{-5, 5}, {-5, 5}}];
```



In most cases it is desired to show the vector field associated with the integral curves of a given solution. To do this you will need the `PlotField` graphics package that you saw in chapter 3. It can be loaded using the following statement:

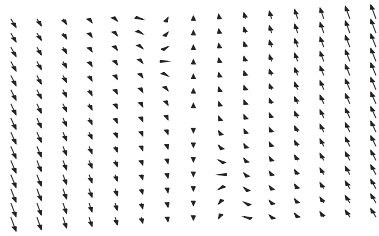
```
(* load the package to plot vector fields *)
<< Graphics`PlotField`
```

A minor but important fact to notice when loading packages with this type of statement is that the tick marks used here are the apostrophe character found in the upper left of most keyboards (`'`), and not a single quote (`‘`).

The goal is to plot the functions against the derivatives telling us where the function is headed. The vector field that we will be plotting is the derivative which is resolved into the components $\{x', y'\}$. These derivatives are given in equation 4.28.

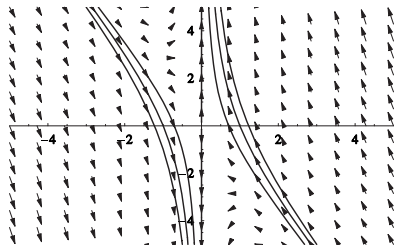
Here is what the vector plot looks like.

```
(* plot the derivative field *)
g2 = PlotVectorField[{-x, 4 x + y}, {x, -5, 5}, {y, -5, 5}];
```



The final step is to combine the two plots which can be done with the `Show` command.

```
(* combine the two plots *)
g3 = Show[g1, g2];
```



Exercise: Phase plots

4.9 Find the solution of $y'' + 9y' + 2y = 0$ and create a vector field and parametric plot.

Finding critical points

Consider the problem defined by these two equations:

$$\frac{dx}{dt} = x(2x - 7y + 22), \text{ and } \frac{dy}{dt} = y(x + 9y - 2) \tag{4.30}$$

what are the equilibrium points of the system? To find these stationary points we need to solve:

$$x(2x-7y+22) = 0 \text{, and } y(x+9y-2) = 0 . \tag{4.31}$$

```
(* derivative of the first system *)
eq1 = x (2 x - 7 y + 22);
(* derivative of the second system *)
eq2 = y (x + 9 y - 2);
```

This can be done simultaneously by using `Solve`, enclosing both equations inside curly braces. We begin with two equation definitions:

Stationary Point

We find these equilibrium points by invoking `Solve`, setting each equation equal to zero:

```
(* find the stationary points *)
spoints = Solve[{eq1 == 0, eq2 == 0}, {x, y}]
```

$$\left\{\{x \to -11,\ y \to 0\},\ \left\{x \to -\frac{184}{25},\ y \to \frac{26}{25}\right\},\ \{x \to 0,\ y \to 0\},\ \left\{x \to 0,\ y \to \frac{2}{9}\right\}\right\}$$

Now how do we make use of the equilibrium points? We can access the points using the same notation that we have been using to extract solutions of differential equations. Let's extract the first point and label it by `(x1, y1)`:

```
x1 = spoints[[1]][[1]][[2]]

-11
```

To get the value of y in the first pair, we extract `[1,2,2]`:

```
y1 = spoints[[1]][[2]][[2]]

0
```

A similar procedure can be used to access the other points. The notation works as follows: the first index selects the address of an equilibrium point from the solution

list. The second index specifies either x or y. The final index marks the object on either side of the arrow in the substitution rule.

Let's go through this process explicitly. We'll approach this like we are peeling an onion. We'll remove a layer, remove another layer, and then remove the final layer using the `Out` operator `%`.

```
(* first level index marks a stationary point *)
spoints[[2]]
```

$$\left\{x \to -\frac{184}{25}, y \to \frac{26}{25}\right\}$$

```
(* within the stationary point index marks *)
(* either the x or y substitution rule *)
%[[1]]
```

$$x \to -\frac{184}{25}$$

```
(* within this substitution rule index points *)
(* to values on either side of the arrow → *)
%[[2]]
```

$$-\frac{184}{25}$$

List manipulation is a topic that one needs to master to be comfortable with *Mathematica*. We have multiple tools to do the same thing. Let's look at five other different ways to get the x value we have shown above.

```
(* five different ways to access data from lists *)
{i, j, k} = {2, 1, 2};   (* note we bundled indices into a list *)
Part[spoints, i, j, k]   (* explicit command *)
spoints[[i,j,k]]          (* easiest to read *)
spoints[[i]][[j]][[k]]    (* levels in list reflected in index level *)
spoints[[i, j, k]]     (* if you don't like the palettes *)
spoints[[i]][[j]][[k]] (* an eye test for C programmers *)
```

$$-\frac{184}{25}$$

$$-\frac{184}{25}$$

$$-\frac{184}{25}$$

$$-\frac{184}{25}$$

$$-\frac{184}{25}$$

As we have seen in chapters 1 and 2, *Mathematica* can quickly generate enormous numbers and lists and navigating these full screens demands more than using Page Up and Page Down. You should spend time practicing accessing lists so you are comfortable with being able to grab or find the exact part that you want.

As we have said many times before, do not try to commit these to memory. Pick the one that you are most comfortable with and use that. The one thing you should take away from this list is the `Part` command. That's where you need to look in the Help Browser for more information.

Part

Critical Point

Exercises

Find the critical points for the following equations.

4.10 $\frac{dx}{dt} = x - 8y + 2$ and $\frac{dy}{dt} = 5x + y - 9$

4.11 $\frac{dx}{dt} = 2x + 2y - 5$ and $\frac{dy}{dt} = x + y + 1$

Example: The Lotka-Volterra equations
In this example we will find the critical points of the following system and evaluate their stability.

$$x' = x - 2xy\,,\; y' = -y + 3xy \tag{4.32}$$

We begin by defining two functions `f1` and `f2` that are given in terms of the definitions of `x'` and `y'`:

```
(* from x' *)
f1 = x - 2 x y;
(* from y' *)
f2 = -y + 3 x y;
```

Notice that we chose the `Set` command (=) over the `SetDelayed` (:=). This is because these values of f define the problem and hence will not change during evaluation.

Next, we use `Solve` to find the critical points of the system. Since we are solving a system of equations and not a system of differential equations, we use `Solve` instead of `DSolve`.

```
(* solve for the critical points *)
cpts = Solve[{f1 == 0, f2 == 0}, {x, y}]
```

$$\left\{\{x \to 0,\ y \to 0\},\ \left\{x \to \frac{1}{3},\ y \to \frac{1}{2}\right\}\right\}$$

We have two critical points. Let's investigate the stability of each in turn and create phase plots. The procedure used is to solve A = ?, where A is a matrix given by:

$$A = \begin{bmatrix} \partial_x f1 & \partial_y f1 \\ \partial_x f2 & \partial_y f2 \end{bmatrix} = \begin{bmatrix} A1 & A2 \\ A3 & A4 \end{bmatrix} \tag{4.33}$$

where we have used the labels $A1$ through $A4$ for later convenience. Although we have already shown some details of matrix manipulation, we want to show how to compute this answer outside of matrix formalism for readers not yet comfortable with linear algebra. Since the system is small, this should not cause too much difficulty. At the end we'll show how to use linear algebra to solve this problem.

First, we compute each of the derivatives found in the matrix:

```
(* build the matrix elements *)
A1 = ∂x f1
A2 = ∂y f1
A3 = ∂x f2
A4 = ∂y f2

1 - 2 y
```

```
-2 x
```

```
3 y
```

```
-1 + 3 x
```

Next, we will need the critical points. There are only two, so lets extract each of them individually as was done in the previous section. We label the first critical point by `(x1, y1)` and the second critical point by `(x2, y2)`.

```
(* harvest the critical points *)
x1 = cpts[[1,1,2]]
y1 = cpts[[1,2,2]]
x2 = cpts[[2,1,2]]
y2 = cpts[[2,2,2]]

0
```

```
0
```

$$\frac{1}{3}$$

$$\frac{1}{2}$$

To gain an understanding of the system, we can study a linearized form of the equations. We seek solutions *u(t)* and *v(t)* that satisfy the given equations at each of the critical points:

$$\frac{du}{dt} = A1u(t) + A2v(t) \tag{4.34}$$

$$\frac{dv}{dt} = A3u(t) + A4v(t)\,. \tag{4.35}$$

where $A1$ through $A4$ are the components of the matrix defined above. First, let's define new quantities `{z1, z2, z3, z4}` that are defined in terms of `{A1, A2, A3, A4}` at the critical points. This can be done easily by using the `ReplaceAll` operator:

```
(* evaluate the A matrix at the critical point {x1,y1} *)
z1 = A1 /. {x → x1, y → y1}
z2 = A2 /. {x → x1, y → y1}
z3 = A3 /. {x → x1, y → y1}
z4 = A4 /. {x → x1, y → y1}

1
```

```
0
```

```
0
```

```
-1
```

Next, we define the equations with the `SetDelayed` operator:

```
(* linearized equations *)
ueqn := ∂t u[t] == z1 u[t] + z2 v[t];
veqn := ∂t v[t] == z3 u[t] + z4 v[t];
```

This is done so that when we pick a new critical point, we don't have to solve the differential equations anew. The same reasoning applies to the `DSolve` command:

```
(* solve the Lotka-Voltera system *)
lvsols := DSolve[{ueqn, veqn}, {u[t], v[t]}, t]
```

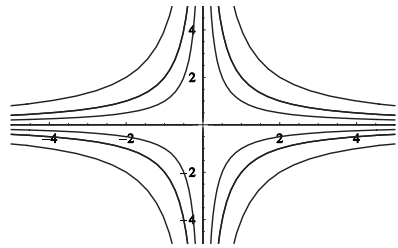
To examine stability, we make a phase plot. First let's make a table of solutions for various values of the parameters `C[1]` and `C[2]`.

```
(* families of solutions *)
ftable :=
  Table[{u1[t], u2[t]} /. {C[1] → i, C[2] → j}, {i, -2, 2}, {j, -2, 2}];
```

```
(* remove one level of braces *)
fftable := Flatten[ftable, 1]
```

All we have to do now is to make a parametric plot in the same way we did in the previous section:

```
(* plot the family of solutions *)
g2 = ParametricPlot[Evaluate[fftable],
   {t, -2, 2}, PlotRange → {{-5, 5}, {-5, 5}}];
```



The result obtained is a familiar one. The critical point (0,0) is a saddle point, which is unstable. We now duplicate the procedure for the second critical point, which has been stored in the variables `(x2,y2)`.

```
(* evaluate the A matrix at the critical point {x2,y2} *)
z1 = A1 /. {x → x2, y → y2}
z2 = A2 /. {x → x2, y → y2}
z3 = A3 /. {x → x2, y → y2}
z4 = A4 /. {x → x2, y → y2}

0
```

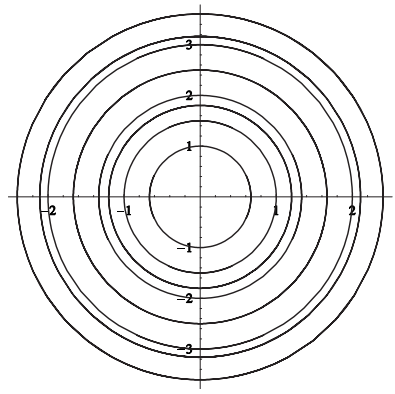
$$-\frac{2}{3}$$

$$\frac{3}{2}$$

```
0
```

The second critical point has given us a set of circular orbits.

```
(* plot the family of solutions *)
g4 = ParametricPlot[Evaluate[fftable], {t, -2, 2}, AspectRatio → 1];
```



Notice that we did not have to go back and solve the differential equation again. We used the `SetDelay` commands so the expressions were interrogated when we called `fftable`. In other words, the evaluation was delayed until call time.

Schematically, here is what happened:

Request for `fftable`: no value stored, compute from `ftable`.
Request for `fftable`: no value stored, compute from `u1` and `u2`
Request for `u1 and u2`: no values stored, compute from `lvsols`
Request for `lvsols`: no value stored, compute from `ueqn and veqn`
Request for `ueqn and veqn`: no value stored, using `z1` through `z4`

Critical Point, Lotka-Voltera Equations

Now let's see how we would approach this problem using the vector tool kit in *Mathematica*. This formulation will allow us to handle problems of arbitrary dimensions and coupled systems of arbitrary size.

We proceed as before by defining the derivative functions.

```
(* from x' *)
f1 = x - 2 x y;
(* from y' *)
f2 = -y + 3 x y;
```

Except now we are going to combine these entries into a list. The size of this list is the size of our system.

```
(* create a list and measure its size *)
f = {f1, f2};
(* measures the size of the system *)
nsys = Length[f];
```

Now we need to specify our coordinate system. We will call the number of spatial dimensions `ndim`.

```
(* define the coordinate system and measure its size *)
var = {x, y};
(* measures the dimension of the space *)
ndim = Length[var];
```

We are now able to have *Mathematica* construct the *A* matrix for us using the recipe in equation 4.33.

```
(* create the A matrix *)
A = Table[D[f[[j]], var[[i]]], {j, nsys}, {i, ndim}];
A // MatrixForm
```

We can now solve for all the critical points at once.

```
(* solve for the critical points *)
cpts = Solve[f == 0, {x, y}]
```

$$\left\{\{x \to 0,\ y \to 0\},\ \left\{x \to \frac{1}{3},\ y \to \frac{1}{2}\right\}\right\}$$

What we really need is not the substitution rules above, but the addresses.

```
(* grab the addresses of the critical points *)
addr = {x, y} /. cpts
nc = Length[addr];
```

$$\left\{\{0,\ 0\},\ \left\{\frac{1}{3},\ \frac{1}{2}\right\}\right\}$$

We are able to evaluate the *A* matrix for every set of critical points.

```
(* create the A matrix *)
A = Table[D[f[[j]], var[[i]]], {j, nsys}, {i, ndim}];
A // MatrixForm
```

$$\begin{pmatrix} 1-2y & -2x \\ 3y & -1+3x \end{pmatrix}$$

Now we need to make preparations for `DSolve` to solve the system of equations 4.34 and 4.35, the first of which is to describe the solution functions.

```
(* define the function list *)
fcns = {u[t], v[t]};
```

What happens now is that we need to express the **DSolve** arguments in terms of the lists that we have created. The index gymnastics can be a bit tedious, but when you see the indices in action it becomes more clear.

Let's build the first set of coupled equations. Our goal is to produce an analog to these two statements:

```
(* first set of critical points *)
DSolve[{u'[t] == u[t], v'[t] == -v[t]}, fcns, t]

{{u[t] → ℯ^t C[1], v[t] → ℯ^-t C[2]}}
```

```
(* second set of critical points *)
DSolve[{u'[t] == -2/3 v[t], v'[t] == 3/2 u[t]}, fcns, t]

{{u[t] → C[1] Cos[t] - 2/3 C[2] Sin[t],
  v[t] → C[2] Cos[t] + 3/2 C[1] Sin[t]}}
```

The pieces that we have are:

```
(* left-hand side of the coupled equations *)
∂t fcns

{u'[t], v'[t]}
```

and *A* and *z* which we have already seen. We show z again to emphasize that we are setting up a system whose solutions will have the same length as *z*.

```
z

{{{1, 0}, {0, -1}}, {{0, -2/3}, {3/2, 0}}}
```

Formally, we will be needing the dot product.

```
(* we can use the dot product *)
z.fcns
```

$$\left\{\{u[t], -v[t]\}, \left\{-\frac{2\,v[t]}{3}, \frac{3\,u[t]}{2}\right\}\right\}$$

By looking at these pieces and the desired solution we can see that we need to construct this syntax.

```
(* solve the problem for the first value of z *)
Table[(∂t fcns)[[i]] == (z[[1]].fcns)[[i]], {i, 2}]
```

$$\{u'[t] == u[t],\ v'[t] == -v[t]\}$$

Hopefully, the pattern is clear.

```
(* now we see the pattern *)
Table[(∂t fcns)[[i]] == (z[[2]].fcns)[[i]], {i, 2}]
```

$$\left\{u'[t] == -\frac{2\,v[t]}{3},\ v'[t] == \frac{3\,u[t]}{2}\right\}$$

The next step in our development is to use the list tools to construct the equations.

```
(* this structure builds the coupled equations *)
Table[
 Table[
  (∂t fcns)[[i]] == (z[[j]].fcns)[[i]]
  , {i, ndim}]
 , {j, nsys}]
```

$$\left\{\{u'[t] == u[t],\ v'[t] == -v[t]\},\ \left\{u'[t] == -\frac{2\,v[t]}{3},\ v'[t] == \frac{3\,u[t]}{2}\right\}\right\}$$

A little bit of care must be taken when we add the `DSolve` command.

```
(* this structure solves the coupled equations *)
solns = Table[
  DSolve[
   Table[
    (∂t fcns)[[i]] == (z[[j]].fcns)[[i]]
    , {i, ndim}]
   , fcns, t]
  , {j, nsys}]
```

$$\left\{\{\{u[t] \to e^{t} C[1],\ v[t] \to e^{-t} C[2]\}\},\ \left\{\left\{u[t] \to C[1]\ \text{Cos}[t] - \frac{2}{3} C[2]\ \text{Sin}[t],\ v[t] \to C[2]\ \text{Cos}[t] + \frac{3}{2} C[1]\ \text{Sin}[t]\right\}\right\}\right\}$$

We need to collapse this list.

```
(* remove one layer of braces *)
fsolns = Flatten[solns, 1];
```

To finish the problem we will add a descriptive output.

```
(* descriptive print *)
Table[
  Print[i, ". critical point = ",
    addr[[i]], "; solution = ", fsolns[[i]];
  , {i, nsys}];
```

```
1. critical point = {0, 0}; solution = {u[t] → e^t C[1], v[t] → e^-t C[2]}
```

$$2.\ \text{critical point} = \left\{\frac{1}{3}, \frac{1}{2}\right\};\ \text{solution} = \left\{u[t] \to C[1]\ \text{Cos}[t] - \frac{2}{3} C[2]\ \text{Sin}[t],\ v[t] \to C[2]\ \text{Cos}[t] + \frac{3}{2} C[1]\ \text{Sin}[t]\right\}$$

Most of the problem that people have with lists is that they don't take the systematic approach shown above. Instead they try to write everything at once like this.

```
(* the compact solution *)
solns =
 Table[DSolve[Table[(∂t fcns)[[i]] == (z[[j]].fcns)[[i]], {i, ndim}], fcns, t]
   Flatten[, {j, nsys}], 1]
```

Clearly, the better approach is to solve the problem incrementally as done earlier.

The concerned reader may wonder how this approach would work if the problem was in, say, six dimensions. Would we be able to build the solution so easily? The answer is of course not. You would take a two-dimensional problem and generalize it like we have done here.

Exercise:

4.12 Investigate the critical points and create phase plots for the following Lotka-Volterra system:

$$x' = x - xy \tag{4.36}$$

$$y' = -y + xy \tag{4.37}$$

Phase plane portrait

As another example that will illustrate the creation of a phase plane portrait, consider the system:

$$x' = -2x + 3y \tag{4.38}$$

$$y' = -2x - 3y \tag{4.39}$$

We load these equations into the list `f` as before.

```
(* bundle the equations *)
f = {-2 x[t] + 3 y[t], -2 x[t] - 3 y[t]}

{-2 x[t] + 3 y[t], -2 x[t] - 3 y[t]}
```

Next, update the solutions list.

```
(* define the solution functions *)
fcns = {x[t], y[t]};
```

This step creates the equations to feed into `DSolve`.

```
(* this is the coupled system to solve *)
sys := Table[(∂t fcns)[[i]] == f[[i]], {i, 2}]
```

This solves the coupled system. Note the economical expression is very readable.

```
(* solve the coupled equations *)
st1 := DSolve[sys, fcns, t]
```

Here is a way to harvest the solutions without worrying about indices.

```
(* harvest the solutions without indexing *)
x1[t_] = fcns[[1]] /. st1;
y1[t_] = fcns[[2]] /. st1;
```

A descriptive summary is always helpful. Notice here that we have replaced a `Table` with a `Do` loop.

```
(* descriptive print *)
Print["the Lotka-Volterra system:"];
Do[
  Print[∂t fcns[[i]], " = ", f[[i]]];
  , {i, 2}];
Print["has the solutions:"];
Print["x[t] = ", x1[t]];
Print["y[t] = ", y1[t]];
```

the Lotka-Volterra system :

$$x'[t] = -2\,x[t] + 3\,y[t]$$

$$y'[t] = -2\,x[t] - 3\,y[t]$$

has the solutions :

$$x[t] = \left\{\frac{6\,e^{-5t/2}\,C[2]\,\text{Sin}\left[\frac{\sqrt{23}\,t}{2}\right]}{\sqrt{23}} + \frac{1}{23}\,e^{-5t/2}\,C[1]\left(23\,\text{Cos}\left[\frac{\sqrt{23}\,t}{2}\right] + \sqrt{23}\,\text{Sin}\left[\frac{\sqrt{23}\,t}{2}\right]\right)\right\}$$

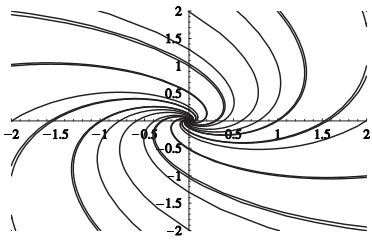
$$y[t] = \left\{-\frac{4\,e^{-5t/2}\,C[1]\,\text{Sin}\left[\frac{\sqrt{23}\,t}{2}\right]}{\sqrt{23}} + \frac{1}{23}\,e^{-5t/2}\,C[2]\left(23\,\text{Cos}\left[\frac{\sqrt{23}\,t}{2}\right] - \sqrt{23}\,\text{Sin}\left[\frac{\sqrt{23}\,t}{2}\right]\right)\right\}$$

We can create a phase portrait in the same way that we did in the previous example. First create a few solution families:

```
(* families of solutions *)
ftable =
  Table[{x1[t], y1[t]} /. {C[1] → i, C[2] → j}, {i, -2, 2}, {j, -2, 2}];
(* remove one level of braces *)
fftable = Flatten[ftable, 1];
```

Then plot them.

```
(* parametric plot of the flattened ftable *)
g1 = ParametricPlot[Evaluate[fftable],
   {t, -2, 2}, PlotRange → {{-2, 2}, {-2, 2}}];
```



C H A P T E R 5

Transforms

As we see in *The CRC Concise Encyclopedia of Mathematics*, a general integral transform $g(s)$ of the function *f(t)* is given by:

$$g(\alpha) = \int_a^b K(s, t)f(t)dt \tag{5.1}$$

where $K(s, t)$ is called the kernel of the transform. Table 5.1 below shows the most popular transforms and their kernels. In general, these transforms are used to solve problems for which a more conventional approach is difficult or impossible. The transform takes the problem into a space where the solution is relatively easy. Then a reverse transform brings us to the solution to the original problem. We will show several examples of this process using differential equations.

When we get to the Fourier transform, we will start with the continuum transform and then look at discrete cases of the Fourier transform. From here, we will look at another discrete transform, the *Z*-transform. First, we want to discuss two important properties.

Name	Transform	Kernel
Laplace	$\int_0^\infty e^{-st} f(t)\,dt$	e^{-st}
Fourier	$\int_{-\infty}^\infty e^{ist} f(t)\,dt$	e^{ist}
Hankel	$\int_0^\infty tJ_n(st) f(t)\,dt$	$tJ_n(st)$
Mellin	$\int_0^\infty t^{s-1} f(t)\,dt$	t^{s-1}

Table 5.1: The most common integral transforms.

5.1 Properties of linear integral transforms

An important property of these transforms is their linearity which can be stated succinctly as:

$$\int_t^{} K(s,\ t)(c_1 f_1(t) + c_2 f_2(t))dt \ = \ c_1 \int_a^b K(s,\ t) f_1(t)dt + c_2 \int_a^b K(s,\ t) f_2(t \qquad (5.2)$$

If we describe the transform with an operator Λ , then the transform is written as

$$g(s) \ = \ \Lambda\,(f) \qquad (5.3)$$

and the linearity can be expressed in a more readable form as

$$\Lambda\,(c_1 f_1 + c_2 f_2) \ = \ \Lambda\,(c_1 f_1) + \Lambda\,(c_2 f_2) \ = \ c_1 \Lambda\,(f_1) + c_2 \Lambda\,(f_2 \qquad (5.4)$$

We rely on the existence of an inverse operator Λ^{-1} so that we can say

$$f(t) \ = \ \Lambda^{-1}(g(s)) \qquad (5.5)$$

Armed with these two basic properties, we are ready to turn our attention to some examples using the Heaviside calculus: transforming differential equations into algebraic equations

In this chapter, we will examine the techniques necessary to compute and use transforms with *Mathematica*. We begin with the Laplace transform.

Integral Transforms, Heaviside Calculus

5.2 The Laplace transform

The Laplace transform is a vital tool in electrical engineering, allowing many differential equations that arise in circuit analysis to be solved using algebraic methods.

5.2.1 Computation of Laplace transforms

The Laplace transform is defined by:

$$F(s) = \int_0^\infty e^{-st} f(t)dt \tag{5.6}$$

In *Mathematica*, we can obtain the Laplace transform of any function by simply typing `LaplaceTransform`. To find out more about the `LaplaceTransform` command, we type:

```
?LaplaceTransform
```

```
LaplaceTransform [expr , t, s] gives the Laplace transform of
   expr . LaplaceTransform [expr , {t1, t2, ... }, {s1, s2, ... }]
   gives the multidimensional Laplace transform of expr . More...
```

Laplace Transform

We begin by computing the Laplace transform of some elementary functions. We designate the Laplace transform of *f* symbolically by *L[f]*. For example, it is known that:

$$L[t^n] = \frac{n!}{s^{n+1}} \tag{5.7}$$

where *n*! is the factorial function, given by $n! = n(n\text{-}1)(n\text{-}2)....(2)(1)$. Note that the factorial can be written in terms of the *Gamma* function:

$$n! = \Gamma(n+1) \tag{5.8}$$

Factorial, Gamma

Factorial, Gamma Function

We can show this result using *Mathematica*:

```
LaplaceTransform[t^n, t, s]
```

```
s^(-1-n) Gamma[1 + n]
```

Here we show some more standard examples by finding the Laplace transforms of the functions $\cos(\omega t)$, $cosh(2t)$, and e^{6x}:

```
LaplaceTransform[Cos[ω t], t, s]
```

$$\frac{s}{s^2 + \omega^2}$$

```
LaplaceTransform[Cosh[2 t], t, s]
```

$$\frac{s}{-4 + s^2}$$

```
LaplaceTransform[ℯ^(6x), x, s]
```

$$\frac{1}{-6 + s}$$

As mentioned above, the Laplace transform is a linear operator. This implies:

$$L[af(t) + bg(t)] = aL[f(t)] + bL[g(t)] = aF(s) + bG(s) \tag{5.9}$$

where the functions with capital letters represent the transform of the functions written in lowercase.

We can show this linearity with *Mathematica* by demonstrating that the transform of the sum is equal to the sum of the transforms. For example:

```
(* demonstrating the linearity *)
lhs = LaplaceTransform[5 Cos[8 t] + 3 ℯ^(2t), t, s]
```

$$\frac{3}{-2 + s} + \frac{5\,s}{64 + s^2}$$

```
(* the transform of the sum equals the sum of the transforms *)
rhs1 = LaplaceTransform[5 Cos[8 t], t, s]
rhs2 = LaplaceTransform[3 ℯ^(2t), t, s]
```

$$\frac{5\,s}{64 + s^2}$$

$$\frac{3}{-2 + s}$$

As mentioned before, we don't put much faith in the "optical inspection technique." Let *Mathematica* do the checking for us.

```
(* verify that this is a linear operator *)
lhs == rhs1 + rhs2

True
```

Here is a more symbolic representation:

```
(* pick two functions arbitrarily *)
fcn1 = 5 Cos[8 t];
fcn2 = 3 e^(2 t);
```

Does the transform of the sum equal the sum of the transforms?

```
(* a symbolic representation *)
LaplaceTransform[fcn1 + fcn2, t, s] ==
 LaplaceTransform[fcn1, t, s] + LaplaceTransform[fcn2, t, s]

True
```

To put the Laplace transform to use in a practical setting, it will be necessary to also calculate the inverse Laplace transform. This is done in *Mathematica* by typing `InverseLaplaceTransform`:

```
? InverseLaplaceTransform
```

```
InverseLaplaceTransform [expr , s, t] gives the inverse Laplace transform of
   expr . InverseLaplaceTransform [expr , {s1, s2, ... }, {t1, t2, ... }]
   gives the multidimensional inverse Laplace transform of expr . More...
```

We demonstrate with a few examples:

```
InverseLaplaceTransform[2/(s + 9), s, t]
```

$$2\,e^{-9t}$$

```
InverseLaplaceTransform[4/(s^2 + 6 s), s, t]
```

$$4\left(\frac{1}{6} - \frac{e^{-6t}}{6}\right)$$

```
InverseLaplaceTransform[3!/s^4, s, t]
```

$$t^3$$

Example: Inverse Laplace transforms
Find and plot the function *f(t)* that corresponds to the Laplace transform given by:

$$F(s) = \frac{2}{2} - \frac{1}{(s+7)(s^2+6s-3)} + \frac{1}{s^2+9} \tag{5.10}$$

```
(* define an inverse Laplace transform *)
g[s_] = 2/s - 1/((s + 7) (s^2 + 6 s - 3)) + s/(s^2 + 9);
(* find the original function *)
f[t_] = InverseLaplaceTransform[g[s], s, t]
```

$$2 - \frac{e^{-7t}}{4} + \frac{1}{24} e^{-\left(3+2\sqrt{3}\right)t}\left(3 + 2\sqrt{3} + \left(3 - 2\sqrt{3}\right) e^{4\sqrt{3}\,t}\right) + \text{Cos}[3\,t]$$

Can this be simplified?

```
(* explore simplification options *)
Simplify[f[t]]
```

$$2-\frac{e^{-7t}}{4}+\frac{1}{24}e^{-\left(3+2\sqrt{3}\right)t}\left(3+2\sqrt{3}+\left(3-2\sqrt{3}\right)e^{4\sqrt{3}\,t}\right)+\text{Cos}[3\,t]$$

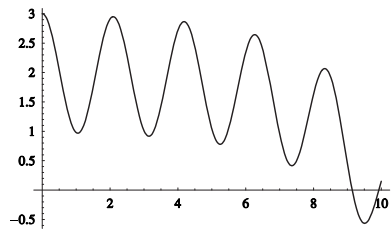
Maybe we will have better luck with `FullSimplify`:

```
(* FullSimplify reduces the expression *)
sf[t_] = FullSimplify[f[t]]
```

$$2-\frac{e^{-7t}}{4}+\text{Cos}[3\,t]+\frac{1}{12}e^{-3t}\left(3\,\text{Cosh}\left[2\sqrt{3}\,t\right]-2\sqrt{3}\,\text{Sinh}\left[2\sqrt{3}\,t\right]\right)$$

Plot the simplified function:

```
(* plot the source function *)
g1 = Plot[sf[t], {t, 0, 10}];
```



Example: Solution of an ODE

Solve the following differential equation using Laplace transforms and compare with the solution from `DSolve`:

$$y'(t)-y(t)\;=\;\cos 5t\,,\;y'(0)\;=\;0 \tag{5.11}$$

First, we set up the differential equation and find the solution with `DSolve`:

```
(* define the differential equation *)
diffeqn = y'[t] - y[t] == Cos[5 t];
(* solve the differential equation *)
soln = DSolve[{eqn, y[0] == 0}, y[t], t]
```

$$\left\{\left\{y[t] \to \frac{1}{26}\left(e^{t} - \text{Cos}[5\,t] + 5\,\text{Sin}[5\,t]\right)\right\}\right\}$$

Now let's take the Laplace transform of the equation:

```
(* transform the expression *)
lap1 = LaplaceTransform[y'[t] - y[t] - Cos[5 t], t, s]
```

$$-\frac{s}{25 + s^2} - \text{LaplaceTransform}[y[t],\, t,\, s] + s\,\text{LaplaceTransform}[y[t],\, t,\, s] - y[0]$$

This procedure has turned a differential equation in t into an algebraic equation in s. All we need to do now is solve for `LaplaceTransform[y[t],t,s]` and invert the result.

```
(* solve the algebraic equation *)
soln2 = Solve[lap1 == 0, LaplaceTransform[y[t], t, s]]
```

$$\left\{\left\{\text{LaplaceTransform}[y[t],\, t,\, s] \to \frac{s + 25\,y[0] + s^2\,y[0]}{(-1 + s)\,(25 + s^2)}\right\}\right\}$$

Here we extract the solution subject to the given initial condition:

```
(* harvest the solution *)
F[s_] = soln2[[1,1,2]] /. y[0] → 0
```

$$\frac{s}{(-1 + s)\,(25 + s^2)}$$

Now we use `InverseLaplaceTransform` to write the solution in terms of t.

```
(* reverse transform brings us back to solution space *)
soln3[t_] = InverseLaplaceTransform[F[s], s, t]
```

$$\frac{1}{26}\left(e^{t} - \text{Cos}[5\,t] + 5\,\text{Sin}[5\,t]\right)$$

Comparison with the solution obtained with `DSolve` shows that they are identical, as expected. To have *Mathematica* do the work for us, we grab the `DSolve` solution:

```
(* grab the solution from DSolve *)
soln1[t_] = y[t] /. Flatten[soln]
```

$$\frac{1}{26}\left(e^{t} - \text{Cos}[5\,t] + 5\,\text{Sin}[5\,t]\right)$$

and compare it with the solution we just found. They agree, as expected.

```
(* DSolve soln =?= LaplaceTransform soln *)
soln1[t] == soln3[t]

True
```

Often it is important to write a Laplace transform in terms of a partial fraction decomposition. This is helpful for the inversion process, especially when the inversion is done by hand. An expression can be split into partial fractions using the *Mathematica* command `Apart`.

Using the previous example, we find that:

```
(* partial fraction decomposition *)
Apart[F[s]]
```

$$\frac{1}{26\,(-1+s)} + \frac{25 - s}{26\,(25 + s^2)}$$

Partial Fraction Decomposition

Further simplification can be obtained by using `FactorTerms`. If we apply `FactorTerms` and `Apart` together on *F(s)*, we get an expression whose elements enjoy a one-to-one correspondence with the elements in the equation in *t*:

```
FactorTerms[%]
```

$$\frac{1}{26}\left(\frac{1}{-1+s}+\frac{25}{25+s^2}-\frac{s}{25+s^2}\right)$$

To see the one-to-one correspondence, note the following inverse transforms:

```
InverseLaplaceTransform[1/(s-1), s, t]
```

$$e^t$$

```
InverseLaplaceTransform[25/(25+s^2), s, t]
```

```
5 Sin[5 t]
```

```
InverseLaplaceTransform[s/(25+s^2), s, t]
```

```
Cos[5 t]
```

This is a great opportunity to explore the list features of *Mathematica*. We begin by bundling our *s* space components into a list.

```
(* list formulation *)
fcn = {1/(s-1), 25/(25+s^2), s/(25+s^2)};
```

Naturally, *Mathematica* is content to apply the inverse Laplace transform to a list.

```
(* take the inverse transform of all the functions in the list *)
soln = InverseLaplaceTransform[fcn, s, t]

{e^t, 5 Sin[5 t], Cos[5 t]}
```

Now we are ready to begin working with the composite functions. We will form the composite function from the *s* space components:

```
(* build the composite function for the list *)
cfcn = Plus @@ fcn

  1          25          s
------ + ---------- + ----------
-1 + s    25 + s^2    25 + s^2
```

and the composite solution using the inverse transform.

```
(* build composite solution from sum of ILT of the components *)
csoln = Plus @@ InverseLaplaceTransform[fcn, s, t]

e^t + Cos[5 t] + 5 Sin[5 t]
```

We see that the sum of the inverse transforms is the inverse transform of the sums.

```
(* a statement of linearity *)
Plus @@ InverseLaplaceTransform[fcn, s, t] ==
 InverseLaplaceTransform[Plus @@ fcn, s, t]

True
```

Example: Solving a circuit problem

One of the most useful applications of the Laplace transform is in the algebraic solution of a differential equation. Let's reconsider the electrical circuit problem worked in the last chapter. The governing equation is:

$$R\, i(t) + L\frac{di}{dt} = \varepsilon \sin(\omega t)$$

We will study the case where $R = 100$ ohms, $L = 10$ H, and $\epsilon = 10$. We also set *i(0)* = 0. First, let's define the constants and the equation:

```
(* assign all the variables at once *)
{R, L, ϵ} = {100, 10, 10};
```

```
(* governing equation *)
eqn1 = R i[t] + L ∂t i[t] - ϵ Sin[ω t]

100 i[t] - 10 Sin[t ω] + 10 i'[t]
```

Next, we compute the Laplace transform of the equation:

```
(* transform of the circuit equation *)
lt1 = LaplaceTransform[eqn1, t, s]

100 LaplaceTransform[i[t], t, s] +
 10 (-i[0] + s LaplaceTransform[i[t], t, s]) - 10 √ω^2 Sign[ω] / (s^2 + ω^2)
```

To find the solution *i(t)*, we need to compute the inverse Laplace transform. First, we need to obtain the Laplace transform of *i* in terms of the other variables:

```
(* find Laplace transform of i as a function of the other terms *)
soln1 = Solve[lt1 == 0, LaplaceTransform[i[t], t, s]]

{{LaplaceTransform[i[t], t, s] → (s^2 i[0] + ω^2 i[0] + √ω^2 Sign[ω]) / ((10 + s) (s^2 + ω^2))}}
```

Now lets define a new function in terms of this result:

```
(* harvest the solution as the inverse transform for the current *)
InvCur[s] = soln1[[1,1,2]]
```

$$\frac{s^2\, i[0] + \omega^2\, i[0] + \sqrt{\omega^2}\ \text{Sign}[\omega]}{(10 + s)\ (s^2 + \omega^2)}$$

To find the current, we compute the inverse transform of `InvCur`:

```
(* solve for the current as a function of time *)
Cur[t_] = InverseLaplaceTransform[InvCur[s], s, t]
```

$$\frac{1}{\omega\ (100 + \omega^2)}\left(e^{-10\,t}\left(\omega\ (100 + \omega^2)\ i[0] - \sqrt{\omega^2}\ \text{Sign}[\omega]\ (-\omega + e^{10\,t}\,\omega\,\text{Cos}[t\,\omega] - 10\,e^{10\,t}\,\text{Sin}[t\,\omega])\right)\right)$$

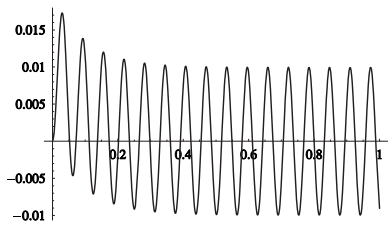
Now we apply the initial condition *i(0)* = 0 and let ω⟶100 Hz.

```
(* specify a solution for plotting *)
pCur[t_] = Cur[t] /. {i[0] → 0, ω → 100}
```

$$-\frac{e^{-10\,t}\ (-100 + 100\,e^{10\,t}\,\text{Cos}[100\,t] - 10\,e^{10\,t}\,\text{Sin}[100\,t])}{10100}$$

With the solution function defined, we can create a plot.

```
(* plot the specific solution *)
g1 = Plot[pCur[t], {t, 0, 1}];
```



Example: Coupled circuits
Two coupled L-R circuits are described by the following two equations:

$$L_1\frac{di_1}{dt} + M\frac{di_2}{dt} + R_1 i_1 = \cos(5t)H(t) \tag{5.12}$$

$$L_2\frac{di_2}{dt} + M\frac{di_i}{dt} + R_2 i_2 = 0 \tag{5.13}$$

where *H(t)* is the Heaviside step function, implemented in *Mathematica* as `UnitStep`. Also, M is the mutual inductance, L_1 and L_1 are the inductance in circuit 1 and 2, and R_1 and R_2 are the resistance in circuit 1 and 2. The initial conditions are known to be $i_1(0) = i_2(0) = 0$. Solve for the currents $i_1(0)$ and $i_2(0)$ and plot them.



Heaviside Step Function

First, we define our constants:

```
(* define the constants *)
{L1, L2, M, R1, R2} = {10, 5, 6, 100, 90};
```

Next, we define a function to represent the source

```
(* voltage as a function of time *)
V[t_] = Cos[5 t] UnitStep[t];
```

and the equations in the problem.

```
(* set up the coupled differential equations *)
diffeq1 = L1 ∂t i1[t] + M ∂t i2[t] + R1 i1[t] == V[t];
diffeq2 = L2 ∂t i2[t] + M ∂t i1[t] + R2 i2[t] == 0;
```

Finally, we encode the initial conditions.

```
(* express initial conditions as a list of substitution rules *)
icrules = {i1[0] → 0, i2[0] → 0};
```

Now we compute the Laplace transforms of these quantities, subject to the initial conditions:

```
(* Laplace transform of the differential equations *)
lt1 = LaplaceTransform[diffeq1, t, s] /. icrules
lt2 = LaplaceTransform[diffeq2, t, s] /. icrules
```

$$100\,\text{LaplaceTransform}[i1[t], t, s] + 10\,s\,\text{LaplaceTransform}[i1[t], t, s] + 6\,s\,\text{LaplaceTransform}[i2[t], t, s] == \frac{s}{25 + s^2}$$

$$6\,s\,\text{LaplaceTransform}[i1[t], t, s] + 90\,\text{LaplaceTransform}[i2[t], t, s] + 5\,s\,\text{LaplaceTransform}[i2[t], t, s] == 0$$

We have a system of equations with two unknowns, the Laplace transforms of the currents. To arrive at a solution to this system, we will use Cramer's rule which tells us how to solve a system of equations like this:

$$ax + by = c \tag{5.14}$$

$$dx + ey = f \tag{5.15}$$

To compute the solution, first we form the determinant of the matrix formed by the coefficients in the equations on the left:

$$\Delta = Det\begin{bmatrix} a & b \\ d & e \end{bmatrix} \tag{5.16}$$

Then the solutions for x and y are given by:

$$x = \frac{Det\begin{bmatrix} c & b \\ f & e \end{bmatrix}}{\Delta}, x = \frac{Det\begin{bmatrix} a & c \\ d & f \end{bmatrix}}{\Delta} \tag{5.17}$$

Cramer's Rule

Returning to our example, first we form the determinant Δ of the system:

```
(* matrix for Cramer's rule *)
α = ( 100 + 10 s    6 s
        6 s      90 + 5 s );
 Δ = Det[α]
```

$$9000 + 1400\,s + 14\,s^2$$

Let us now label the Laplace transforms of the current by `I1[s]` and `I2[s]`.

Let's form the matrices for each that will go in the numerators of equations 5.17 as specified Cramer's rule:

```
(* matrices for the Cramer formula *)
B1[s] = ( V1[s]      6 s
          0      90 + 5 s );
B2[s] = ( 100 + 10 s  V1[s]
            6 s         0 );
```

We are now able to pose solutions for the currents.

```
(* solutions from Cramer's rule *)
I1[s] = Det[B1[s]] / Δ
I2[s] = Det[B2[s]] / Δ
```

$$\frac{\frac{90\,s}{25+s^2}+\frac{5\,s^2}{25+s^2}}{9000+1400\,s+14\,s^2}$$

$$-\frac{6\,s^2}{(25+s^2)\,(9000+1400\,s+14\,s^2)}$$

These solutions can now be inverted to get each of the currents as a function of time:

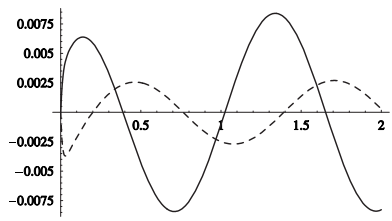
```
(* solve for the currents as a function of time *)
curr1[t_] = InverseLaplaceTransform[I1[s], s, t]
curr2[t_] = InverseLaplaceTransform[I2[s], s, t]
```

$$\frac{1}{45071390}\left(e^{-\frac{10}{7}\left(35+\sqrt{910}\right)t}\left(-173537+773\sqrt{910}-\left(173537+773\sqrt{910}\right)e^{20\sqrt{\frac{130}{7}}t}\right)+91\left(3814\,\text{Cos}[5\,t]+1655\,\text{Sin}[5\,t]\right)\right)$$

$$-\frac{1}{45071390}\left(3\left(-e^{-\frac{10}{7}\left(35+\sqrt{910}\right)t}\left(12740+2047\sqrt{910}+\left(12740-2047\sqrt{910}\right)e^{20\sqrt{\frac{130}{7}}t}\right)+182\left(140\,\text{Cos}[5\,t]-173\,\text{Sin}[5\,t]\right)\right)\right)$$

A plot is much more helpful here.

```
(* plot the currents as a function of time *)
Plot[{curr1[t], curr2[t]}, {t, 0, 2},
  PlotStyle→ {Dashing[{0.01, 0}], Dashing[{0.01, 0.01}]}];
```



Now we shall go back and repeat this process, this time taking advantage of *Mathematica*'s vector tools.

Our thinking shifts here. Instead of thinking of two circuits, we need to think of a list of circuits. Although we have but two here, we won't see a tremendous gain from the vector treatment. On the other hand, the reduced number makes this a great teaching tool.

```
(* define the constants as lists *)
L = {10, 5};    (* inductance *)
R = {100, 90}; (* resistance *)
M = 6;          (* mutual inductance *)
```

We put items in a list and let *Mathematica* handle the operations across the list. In the case above we had constants to assign. Here we will bundle our current variables together for solution.

```
(* define a vector current which describes both system *)
i[t_] = {i1[t], i2[t]};
```

The initial conditions are now expressed as a substitution rule in this form.

```
(* express initial conditions as a list of substitution rules *)
icrules = {i1[0] → 0, i2[0] → 0};
```

The applied voltage is still the same:

```
(* voltage as a function of time *)
V[t_] = Cos[5 t] UnitStep[t];
```

except that know we want to group the voltages applied to both circuits.

```
(* package for Cramer's rule *)
rhs[t_] := {V[t], 0};
(* Laplace transform of the applied voltages *)
nrhs = LaplaceTransform[rhs[t], t, s]
```

$$\left\{\frac{s}{25 + s^2}, 0\right\}$$

This has all been very straightforward. Now, how do we generalize the differential equations? Let's study one equation.

```
(* prototype differential equation *)
eq1 = L1 ∂t i1[t] + M ∂t i2[t] + R1 i1[t] == V[t];
```

This is the first differential equation and we have underlined all the terms that use elements that would be the first in their list: the first current, the first resistance, the first inductance, and the first applied voltage. The term that is boxed is the only term that breaks the pattern of using first entries. It uses the second current in the list.

We are going to build the differential equations in a table with an index μ. We see that the underlined terms will have the same index μ as the table index. The current term will have another index which we will call ν.

What is ν? Typically in these problems of multiple dimensions, ν is the next term. We treat the indices as though they are cyclic. So when μ = 1, ν = 2. When μ = 2, the next coordinate will be 1 because we cycled through the list.

Sometimes these generalization problems become more clear in higher dimensions. Suppose we had three equations. Let's show the values of the three indices as we loop through the cases.

1. {μ, ν, κ} = {1, 2, 3}
2. {μ, ν, κ} = {2, 3, 1}
3. {μ, ν, κ} = {3, 1, 2}

The easiest way to represent such cyclical boundary conditions is to double the list. That is, define:

$$\eta = \{1, 2, 3, 1, 2, 3 \tag{5.18}$$

Then when μ has an arbitrary value, the other indices are always $\nu = \mu + 1$ and $\kappa = \nu + 1$. We see now how quickly this approach generalizes to arbitrary dimension. However, in three dimensions or more we no longer use indices like μ and ν. Instead we use m_μ and m_ν.

Also in two dimensions we will not set up a list like 5.18. We will use the simple fact that

$$\nu = 3 - \mu \tag{5.19}$$

Here is what the generalized differential equation looks like. Compared to the prototype equation the generalization is straightforward.

```
(* prototype differential equation *)
L1 ∂t i1[t] + M∂t i2[t] + R1 i1[t] == V[t];
(* generalized differential equation *)
L⟦μ⟧ ∂t i[t]⟦μ⟧ + M∂t i[t]⟦ν⟧ + R⟦μ⟧ i[t]⟦μ⟧ == rhs[t]⟦ν⟧
```

So now instead of building equations for circuit 1 and circuit 2, we follow a generic prescription and build as many equations as needed. As stated earlier, we won't see a great convenience with only two cases; but if you had three or several cases, then you would need to use the list-based approach.

Here is what the loop looks like.

```
(* build the coupled differential equations in a table *)
eqns = Table[
  ν = 3 - μ;
  L⟦μ⟧ ∂t i[t]⟦μ⟧ + M∂t i[t]⟦ν⟧ + R⟦μ⟧ i[t]⟦μ⟧ == rhs[t]⟦ν⟧
  , {μ, 2}]

{100 i1[t] + 10 i1'[t] + 6 i2'[t] == 0,
 90 i2[t] + 6 i1'[t] + 5 i2'[t] == Cos[5 t] UnitStep[t]}
```

Mathematica will find the Laplace transform of all items in the list.

```
(* apply the Laplace transform to the entire system *)
lt = LaplaceTransform[eqns, t, s] /. icrules

{100 LaplaceTransform[i1[t], t, s] + 10 s LaplaceTransform[i1[t], t, s] +
   6 s LaplaceTransform[i2[t], t, s] == 0,
 6 s LaplaceTransform[i1[t], t, s] + 90 LaplaceTransform[i2[t], t, s] +
   5 s LaplaceTransform[i2[t], t, s] == s/(25 + s^2)}
```

Now comes the most challenging part: isolating the terms for Cramer's rule. Certainly while you are learning *Mathematica,* it is a good practice to copy and paste to move output into input. But in general, we want to get the human out of the loop: we are so slow and so error prone.

This is what we have:

```
(* this is what we have *)
{100 LaplaceTransform[i1[t], t, s] + 10 s LaplaceTransform[i1[t], t, s] +
   6 s LaplaceTransform[i2[t], t, s] == 0,
 6 s LaplaceTransform[i1[t], t, s] + 90 LaplaceTransform[i2[t], t, s] +
   5 s LaplaceTransform[i2[t], t, s] == s/(25 + s^2)}
```

and we need to take this and create a system like this

```
(* this is the system we want to solve *)
(100 + 10 s) LPi1 +        6 s LPi2 = 0
        6 s LPi1 + (90 + 5 s) LPi2 = s/(25 + s^2)
```

This will be an interesting task that will teach us a little about the internal workings of *Mathematica*.

The initial step is to isolate the terms on the left-hand side of the equations. We already have the right-hand side values in the list `nrhs`. We know from previous experience that the function `First` will do the trick. Look at the example action below.

```
(* demonstrating how First gives us the lhs of the equation *)
First[
 100 LaplaceTransform[i1[t], t, s] + 10 s LaplaceTransform[i1[t], t, s] +
   6 s LaplaceTransform[i2[t], t, s] == 0]

100 LaplaceTransform[i1[t], t, s] +
 10 s LaplaceTransform[i1[t], t, s] + 6 s LaplaceTransform[i2[t], t, s]
```

We took one of the output equations and applied `First`. This gives us the left-hand side of the equation. Now we need to `Apply First` across the equations list. If we simply try `First[lt],` we will get the first differential equation. Instead we want the first part of each equation, so we need to enter `First/@lt`.

```
(* grab the lhs of the equalities from the transform *)
eqns = First /@ lt

{100 LaplaceTransform[i1[t], t, s] + 10 s LaplaceTransform[i1[t], t, s] +
  6 s LaplaceTransform[i2[t], t, s], 6 s LaplaceTransform[i1[t], t, s] +
  90 LaplaceTransform[i2[t], t, s] + 5 s LaplaceTransform[i2[t], t, s]}
```

Next, we need to convert these sums into a list. This is easy to do once you see how *Mathematica* handles these quantities. Let's work with a small sum.

```
(* an example of converting a sum into a list *)
test = a + b + c

a + b + c
```

How does *Mathematica* represent this expression?

```
(* we see a Plus operator applied to the element *)
FullForm[test]

Plus[a, b, c]
```

We see the `Plus` operator applied to the argument list. This implies that we can just swap the operator.

```
(* replace the Plus command with List *)
test /. Plus → List

{a, b, c}
```

The `FullForm` highlights the change.

```
(* compare the two FullForms *)
FullForm[%]

List[a, b, c]
```

We are now ready to operate on the extracts from the Laplace transforms.

```
(* convert the sums into a list *)
neqns = eqns /. Plus → List

{{100 LaplaceTransform[i1[t], t, s], 10 s LaplaceTransform[i1[t], t, s],
   6 s LaplaceTransform[i2[t], t, s]},
 {6 s LaplaceTransform[i1[t], t, s], 90 LaplaceTransform[i2[t], t, s],
   5 s LaplaceTransform[i2[t], t, s]}}
```

The ensuing challenge is to remove the Laplace transform terms. As usual, we encourage the reader to investigate things in the interactive environment. We will experiment with a single term.

```
(* use copy and paste to isolate a term for study *)
xx = 100 LaplaceTransform[i1[t], t, s]

100 LaplaceTransform[i1[t], t, s]
```

The `FullForm` is most helpful.

```
(* we need to see FullForm to understand the process *)
FullForm[xx]

Times[100, LaplaceTransform[i1[t], t, s]]
```

We need to know what kind of object the transform term is so that we may address it.

The `Head` operator will reveal the type.

```
(* what kind of object are we dealing with? *)
Head[LaplaceTransform[i1[t], t, s]]

LaplaceTransform
```

For comparison we show a more familiar example.

```
(* the Head specifies the data type *)
Head[3]

Integer
```

The point that we need to demonstrate is that the `Head` of a term is always the 0th element. You may be surprised to see that the integer 3 has a 0th element.

```
(* the Head is also the 0 th list element *)
3[[0]]

Integer
```

Besides being very interesting, the discussion on the 0th element is highly relevant. For when we look for the transform terms, we will see the 0th element listed.

```
(* where is the transform located? *)
spot = Position[xx, LaplaceTransform]

{{2, 0}}
```

Since we don't want to replace the `Head`, we only want to replace the term and we need to ignore the 0 reference in the position list.

```
(* 100 × LaplaceTransform → 100 × 1 *)
ReplacePart[xx, 1, 2]

100
```

If we use the full position `{2,0}`, we replace the `Head` with undesirable results.

```
(* this replaces only the Head, not the command *)
ReplacePart[xx, 1, {2, 0}]

100 1[i1[t], t, s]
```

We want, for example, to go from an expression that is 100*LaplaceTransform to a the expression 100*1 which is of course 100.

With the details resolved, we will find all the locations of transform.

```
(* mark the position of all transforms *)
lst = Position[eqns, LaplaceTransform]

{{1, 1, 2, 0}, {1, 2, 3, 0}, {1, 3, 3, 0},
 {2, 1, 3, 0}, {2, 2, 2, 0}, {2, 3, 3, 0}}
```

We need to sweep through this list and drop the 0 terms. We can use the **Drop** command. Specifying a negative location number tells *Mathematica* to count from the end, not the beginning.

```
(* purge the trailing 0 *)
Drop[{1, 1, 2, 0}, -1]

{1, 1, 2}
```

We can easily bundle this action into a pure function and act on the entire list.

```
(* discard the terminating 0 s *)
lsta = (Drop[#, -1]) & /@ lst
```

We know what we are looking for: Laplace transforms. We know where they are: the positions are in the list `lsta`. Now we need to replace the transforms with the number one. We work on the details on a smaller fragment.

```
(* prototype command *)
ReplacePart[neqns, 1, {1, 1, 2}]

{{100, 10 s LaplaceTransform[i1[t], t, s],
  6 s LaplaceTransform[i2[t], t, s]},
 {6 s LaplaceTransform[i1[t], t, s], 90 LaplaceTransform[i2[t], t, s],
  5 s LaplaceTransform[i2[t], t, s]}}
```

We need to place this action into a pure function.

```
(* functional form of the prototype command *)
fcn = (neqns = ReplacePart[neqns, 1, #]) &;
```

This function can be applied to the Laplace transform of the differential equations, providing us at last with a list of coefficients.

```
(* apply the function across the list *)
fcn /@ lsta;
(* view the result *)
neqns

{{100, 10 s, 6 s}, {6 s, 90, 5 s}}
```

We aren't quite ready to tackle Cramer's rule. The data that we have is an array that is 2×3 and we need a 2×2 array.

```
(* for comparison to the desired form *)
neqns // MatrixForm

⎛100  10 s  6 s⎞
⎝ 6 s   90   5 s⎠
```

We need this form:

```
(* desired form collects the results for each current *)
{{a, b}, {d, e}} = {{100 + 10 s, 6 s}, {6 s, 90 + 5 s}}
```

The most expedient thing to do at this point is to simply build the coefficients to pass into the module we will build to implement Cramer's rule. We will rely heavily upon the dot product with unit vectors to project out the value that we need.

```
(* first term: isolate components *)
a = First[neqns.{1, 1, 0}]

100 + 10 s
```

The second term comes with a superfluous element:

```
(* second term comes with superfluous element *)
b = neqns.{0, 0, 1}

{6 s, 5 s}
```

which we will discard.

```
(* isolate the element that we need: the coupling term *)
b = First[b]

6 s
```

The next term is easy because it is the redundant term: the coupling common to both currents.

```
(* coupling term M is the same for both circuits *)
d = b;
```

The last term is harvested is a way similar to the way we solved for the first term *a*.

```
(* compare to the process for a *)
e = Total[Last[neqns.{1, 1, 0}]];
```

With the terms in hand we will put Cramer's rule into a module.

```
(* Cramer's rule *)
cramer[A_List, B_List] := Module[{det, a, b, c, d, e, f},
  (* set up problem to resemble equations 7.14 - 7.17 *)
  (a b; d e) = A;
  {c, f} = Flatten[B];
  (* solution *)
  det = Det[A];
  {x, y} = Simplify[det^-1 {Det[(c b; f e)], Det[(a c; d f)]}];
 ];
```

We have kept the same {x, y} notation for the output as used in equation 5.17.

Finally, after all these machinations, we are ready to use the Heaviside calculus.

```
(* use's Cramer's rule to solve this system *)
cramer[(a b; d e), nrhs]
{x, y}
```

$$\left\{\frac{s\,(15+s)}{2\,(25+s^2)\,(750+s\,(125+2\,s))},\ -\frac{s^2}{2\,(25+s^2)\,(750+s\,(125+2\,s))}\right\}$$

Our coupled differential equations have been reduced to algebraic expressions. Notice the slight difference between this output and the output we initially saw. This is because we have built the `Simplify` command into the module `cramer`. This will be why the solution function looks slightly different also.

We are now ready to take the inverse transforms to recover the solution.

```
(* use the inverse transform to find the currents *)
current[t_] := InverseLaplaceTransform[{x, y}, s, t]
```

Here is the functional form.

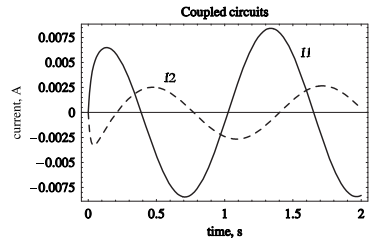
```
(* these terms look different because of simplification *)
current[t]
```

$$\left\{\frac{1}{2169860}\left(-e^{-\frac{5}{4}\left(25+\sqrt{385}\right)t}\left(8393-19\sqrt{385}+\left(8393+19\sqrt{385}\right)e^{\frac{5\sqrt{385}\,t}{2}}\right)+154\,(109\,\text{Cos}[5\,t]+47\,\text{Sin}[5\,t])\right),\right.$$

$$\left.\frac{1}{2169860}\left(e^{-\frac{5}{4}\left(25+\sqrt{385}\right)t}\left(1925+461\sqrt{385}+\left(1925-461\sqrt{385}\right)e^{\frac{5\sqrt{385}\,t}{2}}\right)-154\,(25\,\text{Cos}[5\,t]-28\,\text{Sin}[5\,t])\right)\right\}$$

We will plot this result to demonstrate that the two solutions are equivalent.

```
(* plot parameters *)
frmlbl = {"time, s", "current, A", "Coupled circuits", ""};
pstyle = {Dashing[{0.01, 0}], Dashing[{0.01, 0.01}]};
(* plot command *)
g1 = Plot[Evaluate[current[t]], {t, 0, 2}, PlotStyle → pstyle,
   Frame → True, FrameLabel → frmlbl, DisplayFunction → voff];
(* coordinates used by the plot *)
t1 = Text["I1", {1.6, 0.006}];
t2 = Text["I2", {0.6, 0.0035}];
(* label the curves *)
gt = Graphics[{t1, t2}];
(* combine text and graphics *)
g2 = Show[g1, gt, don];
```



Exercises

5.1 Compute the Laplace transforms of the following functions. Verify the result by using the `InverseLaplaceTransform` command:

a) $f(t) = \dfrac{\sqrt{t}}{\cos t}$

b) $f(t) = e^{-7t} + \sin(2t)$

c) $g(x) = x^2 e^{6x}$

5.2 Use Laplace transform methods to find a solution of the following two equations:

a) $x'(t) + x(t) = te^{-t} + \cosh 5t$

b) $y''(t) + 8y'(t) - 2y(t) = 4\cosh 2t\,,\ y(0) = 1\,,\ y'(0) = 0$

5.3 A mass-spring system with a resistive force proportional to the velocity is described by the equation:

$$mx''(t) + ax'(t) + kx(t) = F(t) \tag{5.20}$$

Solve this equation using Laplace transforms for $m = 2$ kg, $a = 0.1$, $k = 1$, $x(0) = x'(0) = 0$ for the following driving forces:

a) $F(t) = \cos t$

b) $F(t) = e^t \sin t$

5.4 Find the partial fraction decomposition of $\dfrac{1}{(s+3)(s^2+7s+2)}$.

5.3 The Fourier transform

The next transform we will examine is the Fourier transform. In *Mathematica*, there are four commands that are of interest:

`FourierTransform` : returns the Fourier transform of a function
`FourierCosTransform` : returns the Fourier cosine transform of a function
`FourierSinTransfrom` : returns the Fourier sin transform of a function

`Fourier` : gives the discrete Fourier transform of a list of complex numbers

The Fourier transform is written as:

$$F(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{i\omega x} f(x) dx \tag{5.21}$$

while the inverse is defined by:

$$f(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-i\omega x} F(\omega) dx \tag{5.22}$$

5.3.1 The Fourier transform in *Mathematica*

For help on computing the Fourier transform in *Mathematica*, we type `?Fourier-Transform`:

```
? FourierTransform
```

```
FourierTransform [expr , t, ω] gives the symbolic Fourier transform
   of expr . FourierTransform [expr , {t1, t2, ... }, {ω1, ω2, ... }]
   gives the multidimensional Fourier transform of expr . More...
```

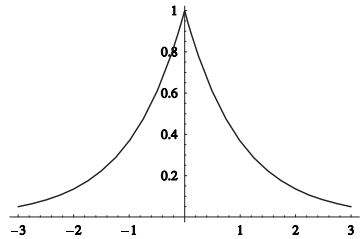
For our first example we consider the function

$$f(t) = e^{-Abs(t)} \tag{5.23}$$

First, we will define and plot the function in *Mathematica*.

```
(* equivalent notation: e^-Abs[t] *)
f1[t_] := Exp[-Abs[t]];
```

```
(* plot the function near the origin *)
g1 = Plot[f1[t], {t, -3, 3}];
```



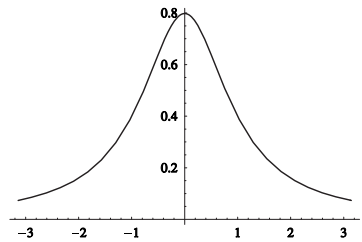
Now let's compute the Fourier transform.

```
(* compute the Fourier transform *)
g[ω_] = FourierTransform[f1[t], t, ω]
```

$$\frac{\sqrt{\frac{2}{\pi}}}{1+\omega^2}$$

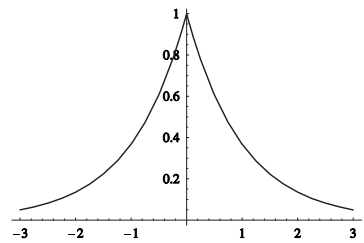
We can plot the Fourier transform against the frequency ω

```
(* look at the transform in frequency space *)
g2 = Plot[g[ω], {ω, -π, π}];
```



Next, we revisit the sinc function, an important function in communications theory. The definition is given in equation 3.2. First, here is a plot.

```
(* plot the function over several periods *)
g1 = Plot[sinc[t], {t, -8 π, 8 π}, PlotRange → All];
```



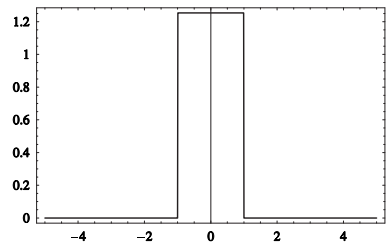
Then we have the Fourier transform.

```
(* compute the Fourier transform of the sinc function *)
g[ω_] = FourierTransform[sinc[t], t, ω]
```

$$\frac{1}{2}\sqrt{\frac{\pi}{2}}\ (\text{Sign}[1-\omega]+\text{Sign}[1+\omega])$$

We will plot the transform in a frame since much of the function is zero-valued.

```
(* plot the transform with a frame *)
g1 = Plot[g[ω], {ω, -5, 5}, Frame → True];
```



In section 3.1.4 we used the `UnitStep` function to build tophat functions to represent a sequence of functions approaching the *Dirac delta* function. We can also use the Sign function. You can see from the previous plot how to build a tophat function with the Sign function, also known as the *signum* function.

```
?Sign
```

```
Sign[x] gives -1, 0 or 1 depending on whether x is negative, zero, or positive.
More...
```

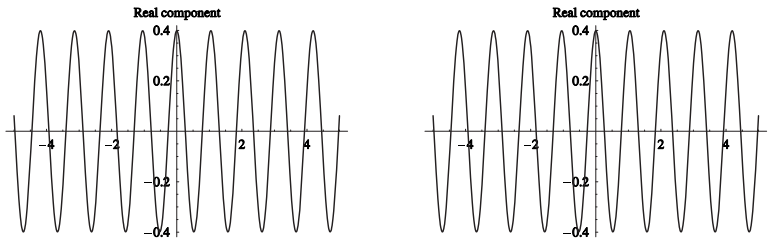


Sign

Here are some commonly used Fourier transforms. The first is the shifted Dirac delta function.

```
(* shifted Dirac delta *)
FourierTransform[DiracDelta[t - 6], t, ω]
```

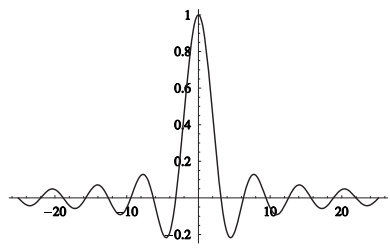
$$\frac{e^{6 i \omega}}{\sqrt{2 \pi}}$$



The Fourier transform of a Gaussian function is another Gaussian function.

```
(* Gaussian *)
FourierTransform[e^(-t^2/4), t, ω]
```

$$\sqrt{2}\, e^{-\omega^2}$$



The `FourierParameters` option

There are many conventions for the definition of the Fourier transform that involve constants multiplying the defining integral. *Mathematica* allows you to select the convention you would like to use by specifying an option called `FourierParameters`. The `FourierParameters` {a, b} are used in the following way:

Fourier Transform, equations 9-10.

Fourier Transform

$$F(\omega) = \sqrt{\frac{|b|}{(2\pi)^{1-a}}} \int_{-\infty}^{\infty} Exp(ib\omega t) f(t)\, dt \tag{5.24}$$

$$f(t) = \sqrt{\frac{|b|}{(2\pi)^{1+a}}} \int_{-\infty}^{\infty} Exp(-ib\omega t) F(\omega)\, d\omega \tag{5.25}$$

By default, *Mathematica* uses $\{a, b\} \rightarrow \{0, 1\}$. You can change the convention used by specifying the `FourierParameters` option as an additional argument in the `FourierTransform` command by typing `FourierParameters⟶{a,b}`. For example, some engineering texts define:

$$g(t) = \int_{-\infty}^{\infty} Exp(-i2\pi f) F(\omega)\, d\omega \tag{5.26}$$

$$G(t) = \int_{-\infty}^{\infty} Exp(i2\pi f) f(t)\, dt \tag{5.27}$$

In this case, comparison with the above shows that {a, b} = {0, 2π}. Therefore we would add the argument `FourierParameters⟶{0, 2π}` to the `Fourier transform` command. To see an example of how to compute a Fourier Transform of a signal using this convention, we again compute the transform of $e^{-Abs(t)}$:

```
(* demonstrating how to change
  the parameters in different conventions *)
FourierTransform[Exp[-Abs[t]], t, f, FourierParameters → {0, 2 π}]
```

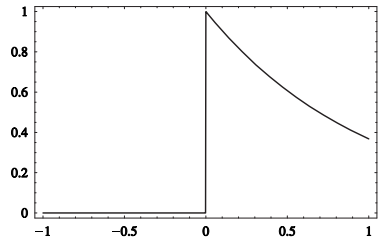
$$\frac{2}{1 + 4\, f^2\, \pi^2}$$

Now that we've seen how to compute the Fourier transform, we can now explore how to compute the inverse. This is easily done by typing `InverseFourierTransform`. For example, we compute the Fourier transform of an exponential pulse. A decaying exponential pulse starts at $f(t) = 1$ for $t = 0$ and decays exponentially as t increases. We can define compose this as:

```
(* decaying exponential *)
f4[t_] = e^-t UnitStep[t];
```

The plot shows the simple behavior.

```
(* plot the decaying exponential *)
g1 = Plot[f4[t], {t, -1, 1}, Frame → True, Axes → False];
```



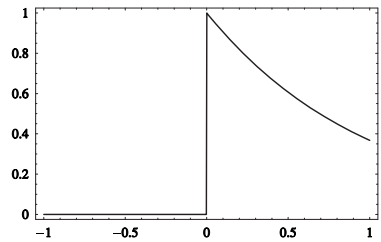
To check, we compute the inverse Fourier transform:

```
(* take the inverse transform *)
x[t_] =
 InverseFourierTransform[g4[f], f, t, FourierParameters → {0, 2 π}]

e^-t UnitStep[2 π t]
```

The result doesn't quite look right, but when we plot, we see that it's the same:

```
(* plot the function *)
g1 = Plot[x[t], {t, -1, 1}, Frame → True, Axes → False];
```



5.3.2 Applications of the Fourier transform

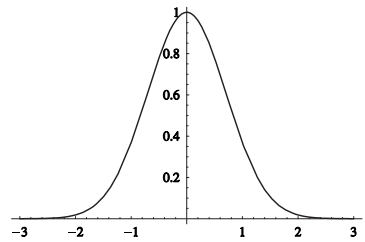
The Fourier transform is used extensively in science and engineering. Even a dedicated volume could not do the topic justice. Our hope here is to provide a few examples as a laboratory for the reader to experiment in.

Example: Signal modulation

As an example application of the Fourier transform, we consider a model of amplitude modulation and filtering. For a message signal *m(t)* amplitude modulation is implemented by multiplication with a carrier wave *c(t)*. As an example, we consider a message signal *m(t)* that is a Gaussian pulse centered at the origin:

```
(* sample message wave *)
m1[t_] = Exp[-t^2];
```

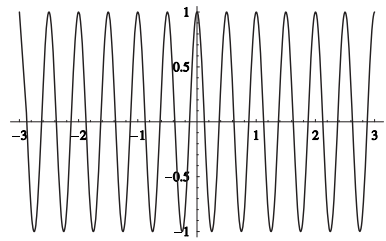
```
(* plot the message wave *)
g1 = Plot[m1[t], {t, -3, 3}];
```



For the carrier wave, we choose a cosine function:

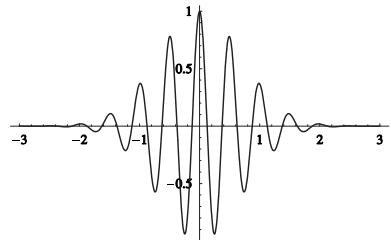
```
(* sample carrier wave *)
c1[t_] = Cos[4 π t];
```

```
(* plot the carrier wave *)
g1 = Plot[c1[t], {t, -3, 3}];
```



The modulated signal is formed by simple multiplication:

```
(* the modulated signal *)
x[t_] = c1[t] m1[t];
```



To demodulate the signal, we begin by computing the Fourier transform:

```
(* Fourier transform of the modulated signal *)
fx[ω_] = FourierTransform[x[t], t, ω]
```

$$\frac{\text{Cosh}\left[\frac{1}{4}(-4\pi+\omega)^2\right]}{2\sqrt{2}}+\frac{\text{Cosh}\left[\frac{1}{4}(4\pi+\omega)^2\right]}{2\sqrt{2}}-\frac{\text{Sinh}\left[\frac{1}{4}(-4\pi+\omega)^2\right]}{2\sqrt{2}}-\frac{\text{Sinh}\left[\frac{1}{4}(4\pi+\omega)^2\right]}{2\sqrt{2}}$$

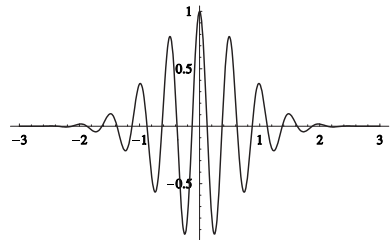
This form looks ripe for simplification.

```
(* the simplified transform *)
fx[ω] = Simplify[fx[ω]]
```

$$e^{-\omega^2}\,\text{Cos}[4\pi\omega]$$

Let's see how the amplitude spectrum of the modulated function looks:

```
(* plot the amplitude spectrum *)
g2 = Plot[fx[ω], {ω, -3, 3}];
```



Often, a signal must be passed through a filter. For example a low pass filter will only allow frequencies that fall within a certain range about the origin in the frequency domain. To construct a low pass filter that lets only frequencies $|\omega| \leq 10$, we can use the `Sign` function:

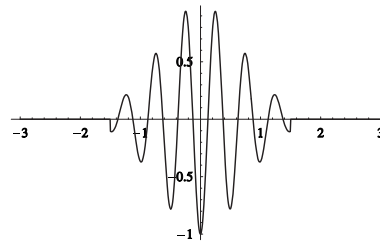
```
(* filter frequencies |ω| ≤ 10 *)
filter[ω_] = 1/2 (Sign[ω - 1.5] - Sign[ω + 1.5]);
```

We will call the filtered function pass, since this is the part of the signal which passes through.

```
(* pass is the filtered function *)
pass[ω_] = filter[ω] fx[ω];
```

Notice how closely the Fourier transform resembles the input function.

```
(* plot the filtered signal in frequency space *)
g1 = Plot[pass[ω], {ω, -3, 3}, PlotRange → All];
```



We could invert this function to obtain the original signal, unfortunately *Mathematica* has problems with this.

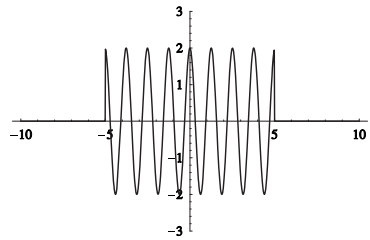
```
(* we are unable to invert the composite function *)
new = InverseFourierTransform[ filter[ω] fx[ω], ω, t]

InverseFourier
 Transform[ 1/2 e^(-ω^2) Cos[4 π ω] (Sign[-1.5 + ω] - Sign[1.5 + ω]), ω, t]
```

Example: An RF pulse

Find the Fourier transform of a radio frequency (RF) pulse with amplitude $A = 2$ and frequency $2\pi f = 5$, that exists from $-5 \le t \le 5$. Plot it, and plot the amplitude spectrum.

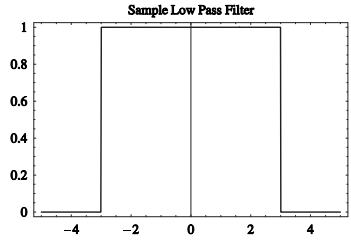
The RF pulse is a sinusoidal wave defined only over a short period of time, as shown in the following plot:



Aside: Representing a filter in *Mathematica*

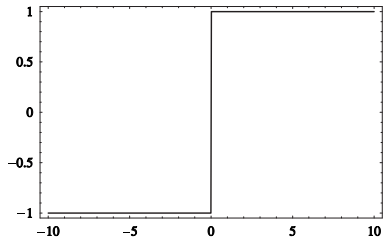
A low pass filter selects frequencies over a given range $-a \le \omega \le a$. Such a selection function can be represented as a step centered at the origin. For example, for $a = 3$, we can construct a low pass filter by using unit step functions as in section 3.1.4:

```
g1 = Plot[UnitStep[t + 3] - UnitStep[t - 3], {t, -5, 5},
   PlotLabel → "Sample Low Pass Filter", Frame → True];
```



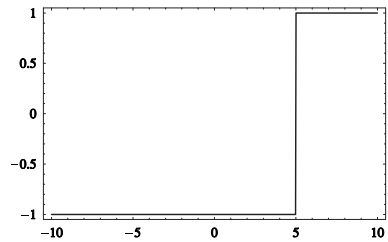
Another way to construct this function in *Mathematica* is to use `Sign` to create a rectangular pulse, which we can then multiply by the appropriate trigonometric function. To see how this works, we plot `Sign[t]`:

```
(* the sign function *)
g1 = Plot[Sign[t], {t, -10, 10}, Axes → False, Frame → True];
```



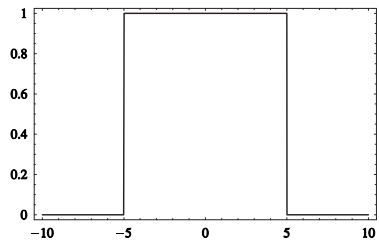
By changing the argument to `t = -5`, we can shift the function five units to the right:

```
(* the shifted sign function *)
g1 = Plot[Sign[t - 5], {t, -10, 10}, Axes → False, Frame → True];
```



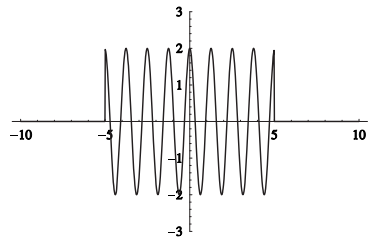
Following a different path from section 3.1.4, we can create the desired tophat function.

```
(* the tophat function *)
g1 = Plot[1/2 (Sign[t + 5] - Sign[t - 5]),
   {t, -10, 10}, Axes → False, Frame → True];
```



With our rectangular pulse in hand, we can define the appropriate RF pulse and plot. Since the problem specifies that the amplitude of the sinusoidal function is $A = 2$, it will cancel the $\frac{1}{2}$ in the definition of the rectangular function. We use a sine function for our pulse:

```
(* define a pulse *)
f3[t_] = Cos[5 t] (Sign[5 - t] + Sign[5 + t]);
(* plot the pulse *)
g1 = Plot[f3[t], {t, -10, 10}, PlotRange → {-3, 3}];
```



The Fourier transform of this pulse is:

```
(* the Fourier transform of the pulse *)
g[ω_] = FourierTransform[f3[t], t, ω]
```

$$\frac{2\sqrt{\frac{2}{\pi}}\ (-5\,\text{Cos}[5\,\omega]\,\text{Sin}[25] + \omega\,\text{Cos}[25]\,\text{Sin}[5\,\omega])}{-25 + \omega^2}$$

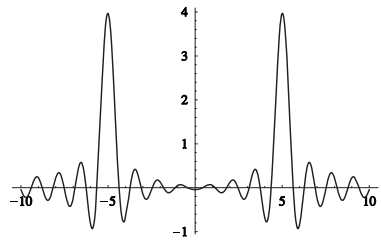
Sometimes it is more useful to examine the frequency domain function in exponential form. We can do this with the `TrigToExp` function:

```
(* look at the frequency domain in exponential form *)
TrigToExp[g[ω]]
```

$$\frac{1}{-25+\omega^2}\left(2\sqrt{\frac{2}{\pi}}\right.$$
$$\left.\left(-\frac{5}{4}\,i\,\left(e^{-25\,i}-e^{25\,i}\right)\,\left(e^{-5\,i\,\omega}+e^{5\,i\,\omega}\right)+\frac{1}{4}\,i\,\left(e^{-25\,i}+e^{25\,i}\right)\,\left(e^{-5\,i\,\omega}-e^{5\,i\,\omega}\right)\,\omega\right)\right)$$

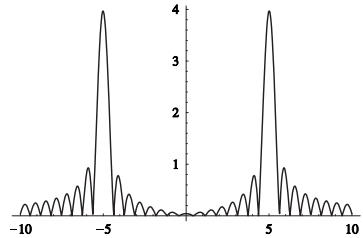
Now let's plot the Fourier transform:

```
(* plot the Fourier transform of the pulse *)
g1 = Plot[g[ω], {ω, -10, 10}, PlotRange → All];
```



To plot the amplitude spectrum, we plot the magnitude of the function:

```
(* plot the amplitude spectrum of the pulse *)
g2 = Plot[Abs[g[ω]], {ω, -10, 10}, PlotRange → All];
```



Exercise

Use the `Sign` function to create a rectangular pulse of unit height that exists for $-\frac{1}{2} \leq t \leq \frac{1}{2}$. Show that the Fourier transform is given by a sinc function and plot the amplitude spectrum.

5.3.3 Parseval's theorem

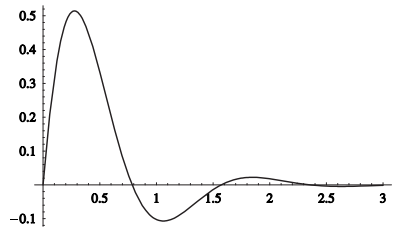
Parseval's theorem tells us that:

$$\int_{-\infty}^{\infty} |F(\omega)|^2 d\omega = \int_{-\infty}^{\infty} |f(t)|^2 dt \tag{5.28}$$

Parseval's Theorem

One application of Parseval is in that it allows us to determine the energy of a signal in either the time or frequency domain, whichever is convenient. As a simple example consider the following signal:

```
(* sample signal *)
i1[t_] = e^-2 t Sin[4 t] UnitStep[t];
```



How much energy does this signal contain? The answer is given by the integral:

$$\int_{-\infty}^{\infty} |f(t)|^2 dt \tag{5.29}$$

We compute that here and find the total energy to be:

```
(* computing the total energy *)
E1 = ∫_0^∞ i1[t]^2 dt
```

```
Integrate::gener : Unable to check convergence . More...
```

$$\frac{1}{10}$$

Note that we integrated from 0 because the unit step function cuts off any contribution over negative *t*. Now we apply Parseval's theorem. First, we take the Fourier transform, and then compute the absolute modulus squared. Then we integrate over all frequencies. Notice that the answers are identical:

```
(* an easier integral *)
E2 = ∫_-∞^∞ g2[ω] dω

1/10
```

Exercises

5.5 Find the Fourier transform of $f(t) = \frac{1}{2} + \frac{2}{\pi}\cos \pi t$. Plot the amplitude spectrum.

5.6 Using Fourier transform and inverse Fourier transform verify that `Sign(t)` and $\frac{1}{i\omega}$ are Fourier transform pairs.

5.7 Apply Parseval's theorem to show the total energy in $i(t) = 6\ Exp(-5Abs(t))$ is $\frac{18}{5}$. Show that half of the energy is contained in the frequency range $-5 \leq \omega \leq 5$.

`FourierCos` and `FourierSin` transforms

In addition to the ordinary Fourier transform, *Mathematica* can compute the Fourier cosine and Fourier sine transforms. In analog to the Euler formula:

$$e^{ix} = \cos x + i \sin x \tag{5.30}$$

the full complex Fourier transform can be resolved into real and imaginary parts. The real part is the Fourier cosine transform and the imaginary part is the Fourier sine transform.

Euler Formula, Fourier Cosine Transform, Fourier Sine Transform

In *Mathematica* these transforms are called using a similar syntax, for example:

```
(* define a function *)
f[t_] = t^4 - 4;
```

```
(* trigonometric transforms *)
FourierCosTransform[f[t], t, w]
FourierSinTransform[f[t], t, w]
```

$$-4\sqrt{2\pi}\ \text{DiracDelta}[w] + \sqrt{2\pi}\ \text{DiracDelta}^{(4)}[w]$$

$$-\frac{4\sqrt{\frac{2}{\pi}}\ (-6+w^4)}{w^5}$$

5.3.4 Finding discrete Fourier transforms

The discrete Fourier transform of a list u of n numbers is called ν_s and is given by

$$\nu_s = \frac{1}{\sqrt{n}} \sum_{r=1}^{n} u_r Exp\left(\frac{2\pi i}{n}\right)(r-1)(s-1) \tag{5.31}$$

To calculate discrete Fourier transforms in *Mathematica*, we use the `Fourier` command.

```
? Fourier
```

```
Fourier [list ] finds the discrete
   Fourier transform of a list of complex numbers . More...
```

As the description indicates, the Fourier command is applied to a set of numbers stored in a list. let's create a list of data by sampling with the sinc function. To do this, we'll need to upgrade our definition of the sinc function.

```
(* upgrade sinc function *)
sinc[x_] := Sin[x]/x  /; x ≠ 0;
sinc[x_] := 0         /; x == 0;
```

Next, we will create a list of sample points by reading the function values along this curve in very small increments.

```
(* create a list of points with the sinc function *)
lst = Table[sinc[x], {x, -8 π, 8 π, π/100}];
```

We compute the discrete Fourier transform of this data as follows:

```
(* discrete Fourier transform *)
flst = Fourier[lst];
```

It would be nice to plot this data in the complex plane. To facilitate this, we will write a quick function to convert complex numbers into a format for plotting as we did in section 3.2.2.

```
(* turns a complex number z into {Re[z], Im[z]} *)
fcn = ({Re[#], Im[#]}) &;
```

We just apply this function to the data to create a list which we can `ListPlot`.

```
(* create a list of plot points *)
plst = fcn /@ flst;
```

It is interesting to compare the continuous and discrete transformations together and we have done this is table 5.2 below. This table makes the point that the process is very different in the continuum than it is from the discrete case.

To demonstrate the normalization of the discrete transform we apply the inverse operation:

```
(* the inverse recovers the original data *)
ilst = InverseFourier[flst];
```

and compare it to the original data set. We will not even bother to print the list for inspection. Instead, we sum the squares of the errors.

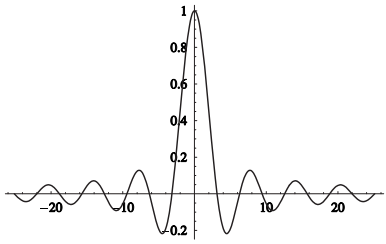
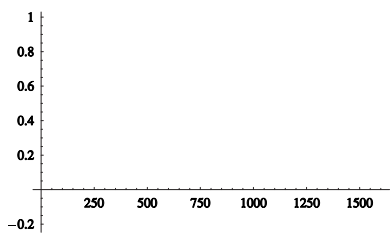
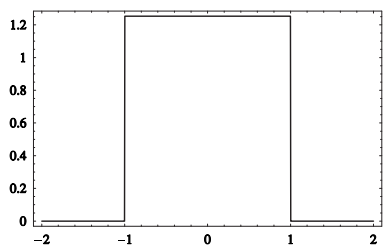
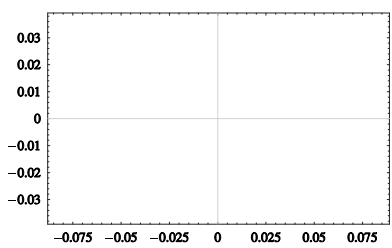
Continuous	Discrete
	
	

Table 5.2: Comparing the continuous and discrete Fourier transforms. The top row shows the original data. The bottom row shows the results of the transforms.

```
(* the inverse recovers the original data *)
diff = (lst - ilst);
Total[diff^2]

2.35299×10^-29
```

To get some idea of the error at each point, we compute the RMS of the errors.

```
(* count the number of points *)
nl = Length[lst];
(* compute the root mean square of the error *)
rmserr = Sqrt[Total[diff^2]/nl]
```

$$1.21231 \times 10^{-16}$$

The last thing to check is the extremal values to see if all our errors were localized.

```
Max[diff]
Min[diff]
```

$$6.66134 \times 10^{-16}$$

$$-6.66134 \times 10^{-16}$$

5.4 The z-transform

The z-transform is defined for a function *f(n)* by the series

$$Z[f(n)] = \sum_{n=0}^{\infty} f(n) z^{-n}. \tag{5.32}$$

The z-transform finds its application primarily in electrical engineering and in the frequency responses of circuits. To calculate a *z*-transform in *Mathematica* we use the `ZTransform` command. We can learn about this command by using the Help Browser:

```
?ZTransform
```

```
ZTransform [expr , n, z] gives the Z transform of expr . More...
```

Z-Transform

Here are some examples of z-transforms and their inverses.

```
ZTransform[1, n, z]
```

$$\frac{z}{-1+z}$$

```
InverseZTransform[%, z, n]
```

$$1$$

```
ZTransform[0.8^n UnitStep[n], n, z]
```

$$\frac{z}{-0.8+z}$$

```
InverseZTransform[%, z, n]
```

$$0.8^n$$

```
ZTransform[Cos[n], n, z]
```

$$-\frac{z\left(-1-e^{2i}+2e^{i}z\right)}{2\left(z+e^{2i}z-e^{i}\left(1+z^2\right)\right)}$$

```
InverseZTransform[%, z, n]
```

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

```
ExpToTrig[%]
```

$$\frac{1}{2}\left(\text{Cos}[n]-i\,\text{Sin}[n]\right)\left(1+\text{Cos}[2n]+i\,\text{Sin}[2n]\right)$$

Here is an interesting inverse transformation.

```
InverseZTransform[1, z, n]

KroneckerDelta[n]
```

To find the initial value of a signal represented by the z-transform, we take the limit as $z \to \infty$. For example, we will show how to find the z-transform of $4 - 2^n(n+1)$ and its initial.

```
(* sample signal *)
signal = 4 - 2^n (n + 1);
```

```
(* Z-transform of the signal *)
a1[z_] = ZTransform[signal, n, z]

-2z/(1-2z)^2 + 4z/(-1+z) - 2z/(-1+2z)
```

```
(* simplify the transform *)
Simplify[a1[z]]

4z(1-3z+3z^2)/((1-2z)^2(-1+z))
```

Now we can find the initial value of the signal from the transform.

```
(* find the initial value of the signal *)
Limit[a1[z], z → ∞]

3
```

This agrees with the initial value found from the signal definition.

```
(* check against the definition of the signal *)
signal /. n → 0

3
```

The final value of a function is found by multiplying the *z*-transform by $z-1$ and letting $z \to 1$. This formula is valid as long as the limit of *f(n)* as $n \to \infty$ exists. For example, consider this signal:

```
(* sample signal *)
signal = 2^-n - 6;
```

The *z*-transform is:

```
(* Z-transform of the signal *)
h[z_] = ZTransform[signal, n, z]

-(6 z)/(-1 + z) + (2 z)/(-1 + 2 z)
```

Again we find the initial value from both the signal and the transform.

```
(* find the initial value of the signal *)
Limit[h[z], z → ∞]

-5
```

```
(* check against the definition of the signal *)
signal /. n → 0

-5
```

Now we will find the limiting, or final, value. Given the *z*-transform *h(z)* we take the limit of $h(z)(z-1)$ as $z \to 1$.

```
(* this will give us the final value in the limit *)
final = Simplify[h[z] (z - 1)]
```

$$\frac{2\,(2-5\,z)\,z}{-1+2\,z}$$

The limit will give us the final value:

```
(* final value for the signal *)
Limit[final, z → 1]
```

```
-6
```

which agrees with the signal:

```
(* check against the definition of the signal *)
signal /. n → ∞
```

```
-6
```

Exercise

5.8 Find the z-transform of $3^{-n} - \cos(2n)$. Compute the initial and final values.

CHAPTER 6

Integration

Integration is one of the most important and frequently used tools of calculus, finding its way into many applications of science and engineering. Unfortunately, it's often the case where only the simplest of expressions are easy to integrate by hand. In the past this could mean that if you came across an expression that could not be integrated using the elementary methods and tricks taught in calculus, you would have to dig out some heavy book and search endless tables to find the answer. If you were lucky some 19th century mathematician had done the hard work for you and you would find what you were looking for. Luckily for us, *Mathematica* shines when computing integrals of all kinds, and can be used to quickly determine the answer whether the problem at hand involves a definite or indefinite integral. Answers to indefinite integrals are reported symbolically in functional form just like you would write them on a piece of paper. Integrals can be computed using any coordinate system you desire. So instead of taking a trip to the library or bookstore to leaf through a heavy table of integrals, you can simply type in the integral you are trying to compute and let *Mathematica* do the hard work. In this chapter we will consider how to use *Mathematica* to integrate several well-known functions. This will be done in two ways. First we investigate the use of symbolic input that lets you enter the functions you want to inte-

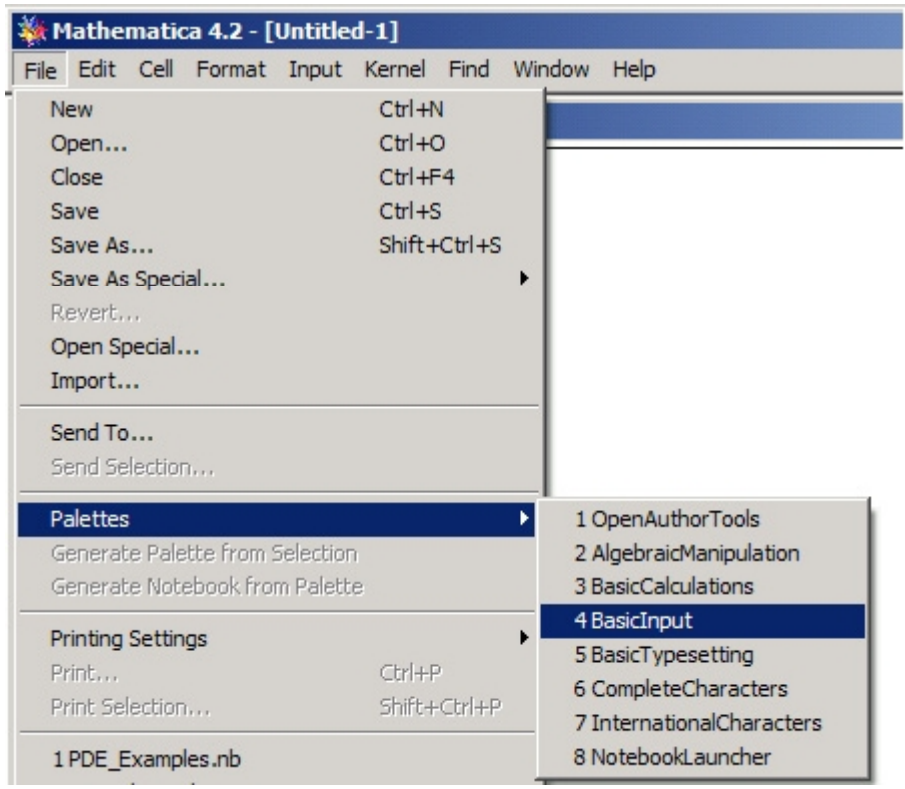
grate the way you would by writing the formulas down on a piece of paper. Next, we will examine the use of the `Integrate` command, where integrals are input the way you might expect to enter data into a computer program. Here the trade-off is giving up some of the familiarity for a bit of power.

6.1 Basic integrals: polynomials and rational functions

We begin by considering the simplest of integrals, computing the antiderivative of $f(x) = x$. Of course, integrating such a simple expression is no doubt trivial. However by looking at this case we can quickly get a feel on how to enter integrals into the software. To get started, first make sure that the `BasicInput` palette is open. If you cannot find this palette on your screen, execute these steps:

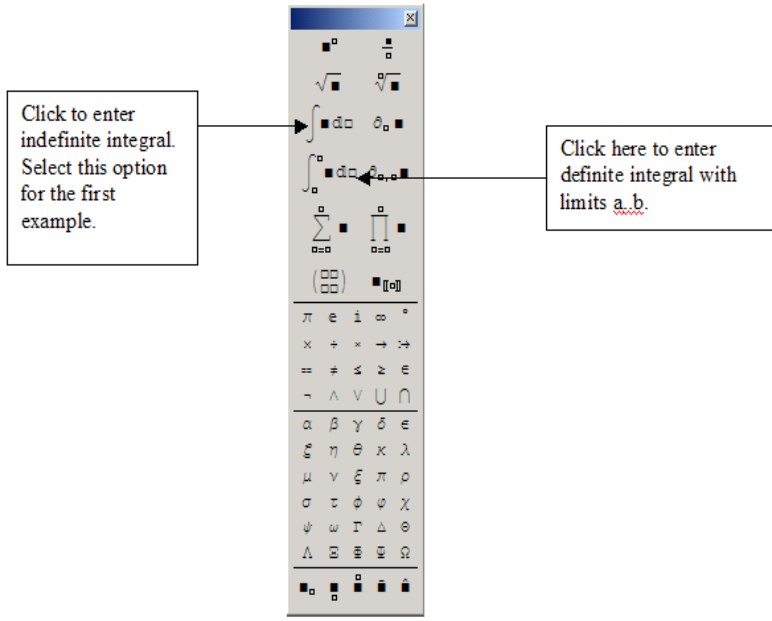
1) Click the `File` pull-down menu
2) Select `Palettes`
3) Click on selection 4, `Basic Input`.

If you have executed these steps correctly, your menu will appear as shown here:

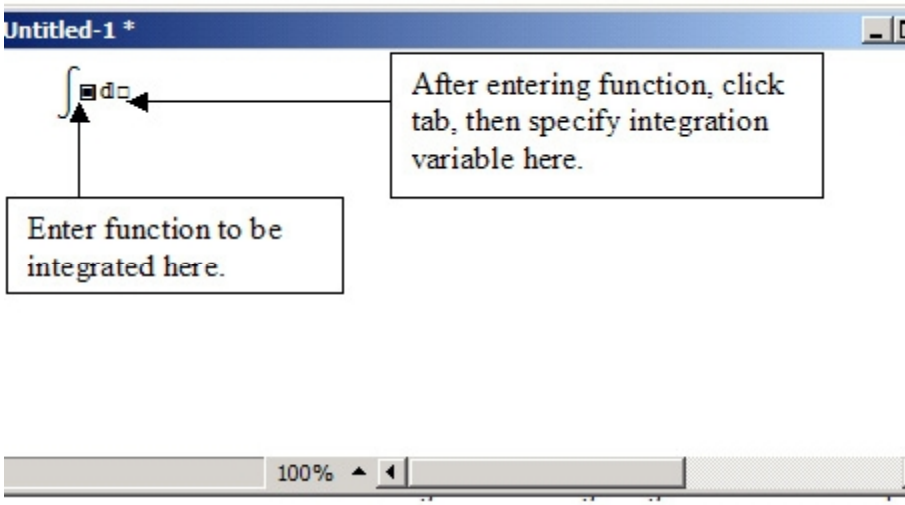


The `BasicInput` palette will open on the right end of your screen. For our present purposes, there are two icons that are of importance, and you will quickly recognize them from your experience with calculus.

The first of these is an icon for calculating antiderivatives or indefinite integrals. Just below this you will notice an icon for entering a definite integral, or an integral that is computed between limits *a* and *b*.These icons are shown below.



These icons demonstrate one of the nicest features of *Mathematica*, which allow us to enter input in a way that closely resembles the way we would write the formula on a piece of paper. But the software will be doing all the hard work for us, so we can avoid those famous headaches that come with doing integral calculus. To begin, click on the indefinite integral icon. *Mathematica* will add an integral to the input screen with the cursor placed in a position where you can type the integrand. The screen should look like this:



Begin by typing an *x* for our integrand. Next, we have to tell *Mathematica* what the integration variable is. To do this, press the Tab key. This will move the cursor to the right where you can simply type another x. To display the result, follow these steps:

1) Click the *Kernel* pull down menu.
2) Select *Evaluation.*
3) Click *Evaluate Notebook.*

After a quick computation *Mathematica* will display the result we all know from calculus, $\frac{1}{2}x^2$ .Your screen should look like this:

```
(* a simple integral *)
∫ x dx

x^2
---
2
```

This first example wasn't very complicated. In fact, most of us could recite the answer automatically. To see the value of using *Mathematica*, let's gradually build up to integrating more complicated functions. The first we will consider are functions involving various powers of x. *Mathematica* can handle such functions completely

symbolically or numerically. For example, we can compute the integral of x^r without specifying the value of r. If we enter this expression:

```
∫ x^r dx
```

$$\frac{x^{1+r}}{1+r}$$

We can also include constants in our integrals. Suppose, for example, we wanted to calculate the integral of cx^2 , where c is a constant:

```
∫ cx^2 dx
```

$$cx^2\,x$$

This probably isn't what you expected. So what happened? We confused the software by entering our data incorrectly. *Mathematica* isn't sure whether we are considering x as a single variable, or we really want c to be a multiplicative constant. If you run into this problem, simply add a space between the c and x^2 in your integrand:

```
∫ c x^2 dx
```

$$\frac{c\,x^3}{3}$$

Another, less pretty way to ensure *Mathematica* understands what you mean is to use the multiplication symbol * :

```
∫ c * x^2 dx
```

$$\frac{c\,x^3}{3}$$

The next step is to integrate more complicated expressions like square roots or rational functions. For example, suppose we wish to integrate the following expression:

```
f[x_] = √(x^2 - 2 x^3);
int = ∫ f[x] dx
```

$$\frac{(-1 - x + 6x^2)\sqrt{x^2 - 2x^3}}{15x}$$

Can the quadratic term be factored?

```
Simplify[int]
FullSimplify[int]
```

$$\frac{(-1 - x + 6x^2)\sqrt{x^2 - 2x^3}}{15x}$$

$$\frac{(-1 + 2x)(1 + 3x)\sqrt{x^2 - 2x^3}}{15x}$$

Remember that the semicolon tells *Mathematica* to suppress any output of the expression just entered. If you want to use the result of an integral later, you can define a function in terms of that integral. This can be done by simply entering an expression like this:

```
f[x_] := ∫ x^3 dx
```

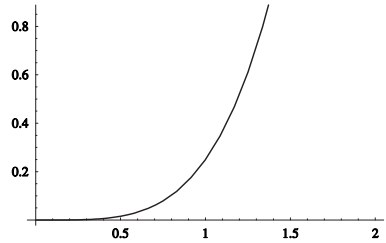
$$\frac{x^4}{4}$$

Later, we might want to do different things with `f[x]`. As a simple example, we can then include *f* in a new expression and calculate the derivative, or plot the function:

```
g[x_] = ∂x f[x]
```

$$x^3$$

```
g1 = Plot[f[x], {x, 0, 2}];
```



Sometimes the expressions you enter can give the software a headache. For instance, if we try to evaluate the following:

```
r[x] = ∫ x/5 + x^2 dx
```

```
Syntax :: sntxb : Expression cannot begin with "∫ x/5 + x^2 dx". More...
```

```
r[x] = ∫ x/5 + x^2 dx
```

To avoid this kind of problem, we can enclose the integrand in parentheses:

```
r[x_] = ∫ (x/5 + x^2) dx
```

$$\frac{x^2}{10} + \frac{x^3}{3}$$

Notice that since we have defined the integral as a new function `r[x]`, we can use the `Simplify` command we met in chapter 1 to put both terms over a common denominator:

```
Simplify[r[x]]
```

$$\frac{1}{30} x^2 (3 + 10\,x)$$

Example

Find $\int \sqrt{\frac{1}{2x^2+11}}\,dx$ and plot the result over the domain $x = \{0,1\}$.

The steps required to solve this problem using *Mathematica* are the following:

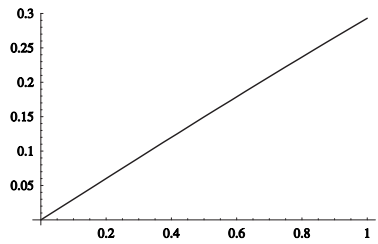
1) Type $f[x] = \sqrt{2x^2+11}$ and then click on the integral icon of the `BasicInput` palette.
2) Next click on the fraction icon, also found on the `BasicInput` palette.
3) Enter a "1" for the numerator, and press the Tab key to move the cursor to the denominator and enter f [x].
4) Press Tab and enter `x` for the integration variable. The input should look like this:

```
f[x_] = √(2 x^2 + 11);
g[x_] = ∫ 1/f[x] dx
```

$$\frac{\text{ArcSinh}\left[\sqrt{\frac{2}{11}}\,x\right]}{\sqrt{2}}$$

Now tell *Mathematica* to plot by typing the following command:

```
g1 = Plot[g[x], {x, 0, 1}];
```



6.2 Multivariate expressions

We now move on to consider the integration of expressions in more than one dimension. For our first example, we consider the integration of a function *f(x,y)* given by the following:

```
(* define a multivariate polynomial *)
2 x y + y^2 x - x^3;
```

Our natural inclination is to simply enter this into the program the way we might write it on a piece of paper:

```
(* integrate the polynomial *)
∫∫ 2 x y + y^2 x - x^3 dx dy
```

```
⚠ Integrate::nodiffd : ∫∫ 2 xy cannot be interpreted. Integrals are entered in the form ∫ f dx, where d is entered as ESC dd ESC. More...

⚠ ∫∫ 2 x y + y^2 x - x^3 dx dy
```

It appears that the software has had trouble interpreting this expression. There are two ways to get around this problem. The first is to enclose the expression to be integrated in parentheses:

```
∫∫ (2 x y + y^2 x - x^3) dx dy
```

$$-\frac{x^4 y}{4} + \frac{x^2 y^2}{2} + \frac{x^2 y^3}{6}$$

Another way to avoid this error is by defining the function ahead of time, in the same way that we did with the previous example when integrating the square root of the function. If instead we first enter this statement:

```
(* define a multivariate function *)
f[x_, y_] = 2 x y + y^2 x - x^3
```

$$-x^3 + 2 x y + x y^2$$

```
dint = ∫∫ f[x, y] dx dy
```

$$-\frac{x^4 y}{4} + \frac{x^2 y^2}{2} + \frac{x^2 y^3}{6}$$

Simplification would be nice.

```
FullSimplify[dint]
FullSimplify[dint]
```

$$\frac{1}{12} x^2 y (-3 x^2 + 2 y (3 + y))$$

$$\frac{1}{12} x^2 y (-3 x^2 + 2 y (3 + y))$$

To enter a multidimensional integral, use these steps. First, add an integral to your notebook by clicking on the `Integration` icon found on the `BasicInput` palette. Your notebook should look like this:

```
(* palette entry *)
∫□ d□
```

Now we need to repeat the process to tell the program this is a multivariate integral. Click where you normally would to type in the function you want to integrate and then click again on the integration icon. The screen should look like the following:

```
(* double palette entry *)
∫∫□ d□ d□
```

Now enter the expression you wish to evaluate, either by enclosing it in parentheses or by defining a function to integrate. If you define a function, you can enter the expression for any two variables you want. For instance, we could define the following function:

```
(* sample function *)
g[x_, y_] = x^2 - 2 y x^3;
```

Recall that an expression like this simply defines *x* and *y* as placeholders. We can now use this as a general function definition and if for some reason we want to use other variables in the expression, we can do so. Suppose, for example, we wish to use variables *s* and *t.* To do this we enter the following:

```
(* integrate the function *)
dint = ∫∫ g[s, t] ⅆs ⅆt
```

$$\frac{s^3 t}{3} - \frac{s^4 t^2}{4}$$

What kind of simplification can we use?

```
Simplify[dint]
FullSimplify[dint]
```

$$\frac{1}{12} s^3 t (4 - 3 s t)$$

$$\frac{1}{12} s^3 t (4 - 3 s t)$$

We can extend the process to compute triple integrals or for string theory fans, integrals in any number of dimensions. For example, *Mathematica* quickly calculates the integral:

```
(* problematic syntax *)
∫∫∫∫ u v + u^2 x - 2 x y + u^4 x^2 v du dv dx dy
```

```
⚠ Integrate::nodiffd : ∫∫∫∫ u v cannot be interpreted. Integrals are entered in the
    form ∫ f dx, where d is entered as ESC dd ESC. More...

⚠ ∫∫∫∫ u v + u^2 x - 2 x y + u^4 x^2 v du dv dx dy
```

Again, recall we can fix this problem by enclosing the integrand in parentheses:

```
(* parentheses are the cure *)
∫∫∫∫ (u v + u^2 x - 2 x y + u^4 x^2 v) du dv dx dy
```

$$\frac{1}{60}\, u\, v\, x\, y\, (15\, u\, v + 10\, u^2\, x + 2\, u^4\, v\, x^2 - 30\, x\, y)$$

Example
One of our favorite exercises in calculus is integration by parts. Use *Mathematica* to integrate the function $e^x \sin x$.

To integrate the sin function, we will use the built-in *Mathematica* function `Sin[]` that we met in chapter 1. When you add the integral to your notebook, it should look like this:

```
∫ e^x Sin[x] dx
```

$$\frac{1}{2}\, e^x\, (-\mathrm{Cos}[x] + \mathrm{Sin}[x])$$

Notice, however, that *Mathematica* does not include the integration constant. Keep this in mind when performing indefinite integrals.

6.3 Definite integration

Up to this point, we have seen how to enter some basic indefinite integrals into *Mathematica*. But as we all know, most of the time integrals are going to be calculated over some range of data. To compute such integrals, from now on we will use the definite integral icon found on the basic input palette:

```
(* palette entry for definite integration *)
∫□□ □ dl □
```

Notice that this symbol allows you to specify the limits of integration. When you click this icon and the integral is added to your notebook, the cursor will be located at the lower limit of the integral. To see how this works, consider integration of the trigonometric function $\sec(x)^4$ between 0 and π/4.

```
(* palette entry for definite integration *)
∫□□ □ dl □
```

The first step is to select the definite integral icon from the *Basic Input* palette. You will find the cursor located at the lower limit, so you can type a 0 there. Press tab and the cursor will move to the upper limit of the integral. Enter a π */4.* When you press the Tab key, the cursor will move to the integrand. Our notebook looks like this:

$$\int_0^{\frac{\pi}{4}} \square \, dl \square$$

Use the icon to enter a variable or function to a power that we saw in the previous example. Then type in `Sec[x]`. Press the tab key and type 4. Finally, press tab and enter `x` as the variable to integrate. Your notebook input should look like this:

$$\int_0^{\frac{\pi}{4}} \text{Sec}[x]^4 \, dx$$

$$\frac{4}{3}$$

When integrating expressions involving built-in function definitions used by *Mathematica*, be sure to space your input correctly in the same way we did when integrating a function multiplied by a constant. For example, suppose you want to compute this integral:

```
∫ xSinh[x] dx

General::spell1 : Possible spelling error : new
    symbol name "xSinh" is similar to existing symbol "Sinh". More…

∫ xSinh[x] dx
```

To remedy this situation, we simply insert a space before `Sinh`:

```
∫ x Sinh[x] dx

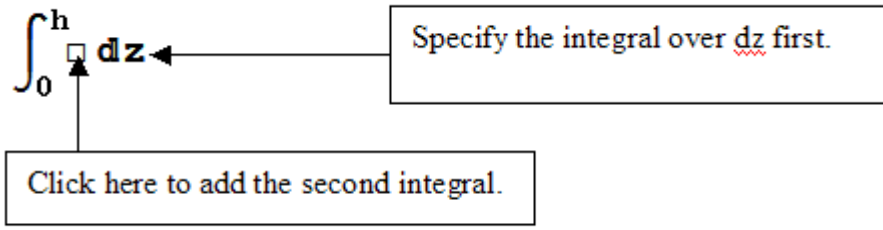
x Cosh[x] - Sinh[x]
```

We can also evaluate definite integrals in higher dimensions and in different coordinate systems. For example, we can use *Mathematica* to calculate the volume of a cylinder, integrating using cylindrical coordinates:

```
(* cylinder volume *)
∫0h ∫02π ∫0ρ r dr dϕ dz

h π ρ^2
```

When entering multidimensional integrals like this one, be sure to follow the correct order when specifying the variable to be integrated. Notice that the integral sign on the outer left side matches the `dz` on the far right, and then the variables work inward. The best thing to do is to specify the variable of integration at each step before adding the next integral. For example, suppose we wanted to compute the integral of zr^2 in cylindrical coordinates. In this case we would have:



After entering the second integral we have:

$$\int_0^h \int_0^{2\pi} \square \, d\phi \, dz$$

Then we add the integration over *r*:

$$\int_0^h \int_0^{2\pi} \int_0^{\rho} \square \, dr \, d\phi \, dz$$

We can finally specify the integrand. Remember we need to include an extra *r* for the volume element in cylindrical coordinates:

$$\int_0^h \int_0^{2\pi} \int_0^{\rho} z r^3 \, dr \, d\phi \, dz$$

$$\frac{1}{4} h^2 \pi \rho^4$$

As a user, you have to be careful about entering the data in the correct way. *Mathematica* cannot tell you that you have specified the integration variables out of order, and will calculate erroneous results. For instance, if we mistakenly switched `dz` and `df`:

```
∫_0^h ∫_0^(2 π) ∫_0^ρ z r^3 dr dz dϕ

1/2 h π^2 ρ^4
```

Mathematica reports a result which is in error.

Example
Use *Mathematica* to compute the volume of a sphere of radius *a* in spherical coordinates. As we saw above, *Mathematica* can compute integrals using any coordinate system. Spherical polar coordinates are used often, and computing integrals with them is no problem. We will refer to **d□** as the differential. To solve this problem, we use the following steps:

1) Click the definite integral icon. We will use the first integral to integrate over the polar angle. Set the range to run from 0 to π. Click on the differential and enter. Click on the integrand input marker.
2) Now return to the `BasicInput` palette. Again click the definite integral icon.The second integral will be used for the azimuthal angle. Enter the range of integration as 0 to 2 π. Click on the differential and enter. Click to place the cursor on the integrand.
3) Once more, click the definite integral icon to add the third integral. Now set the limits of integration to run between 0 and *a*. Click on the differential and specify *r*.
4) For the last integrand, enter the Jacobian for spherical coordinates, $r^2 \sin\theta$.
5) Following the order in which we entered the limits on integration, the final result should look like the integral shown below:

```
∫_0^π ∫_0^(2 π) ∫_0^a r^2 Sin[θ] dr dϕ dθ

4 a^3 π
-------
   3
```

6.4 Integrals involving the Dirac delta function

One of the biggest advantages of using *Mathematica* is that we have a multitude of built-in functionality at our fingertips. Later on we will see how to use many of the famous special functions of mathematics in various situations. Here, we will briefly examine how the *Dirac delta* function can be used in integrals. The Dirac delta function is used with the following syntax in *Mathematica*:

```
(* calling the Dirac delta function *)
DiracDelta[x]

DiracDelta[x]
```

This *Mathematica* command can be included inside integrals to use the famous sampling property of the delta function:

$$\int_0^4 x^2 \, \text{DiracDelta}[x-3] \, dx$$

```
9
```

To use the Dirac delta with more than one dimension, we can either enter individual Dirac functions for each integration variable:

$$\int_0^{12} \int_0^{14} \left(x^2+y\right) \text{DiracDelta}[x-2] \, \text{DiracDelta}[y-3] \, dx \, dy$$

```
7
```

Or we can use commas to separate each variable in a multidimensional Dirac function:

$$\int_0^{12} \int_0^{14} \left(x^2+y\right) \text{DiracDelta}[x-2, y-3] \, dx \, dy$$

```
7
```

Either way, the result is 7 for this particular expression. The built-in Dirac delta function has all of the behavior we would expect. For instance, if the argument is outside of the range of the integral, the result will be zero:

```
(* delta function is inside the domain *)
∫_0^(2 π) Cos[x] DiracDelta[x - π] dx

-1
```

```
(* delta function is outside the domain *)
∫_0^(2 π) Cos[x] DiracDelta[x - 9] dx

0
```

6.5 Using the Integrate command

In this chapter we have been doing all of our integrals using symbolic input that lets us view the input in much the same way we would do our math by hand on a piece of paper. This is one of the nicest features of the recent versions of *Mathematica*. But for those who want more direct access, it is possible to use the `Integrate` command instead. `Integrate` can be used to compute both definite and indefinite integrals, in one or more dimensions.The syntax for computing an indefinite integral is:

```
(* basic syntax for the Integrate command *)
Integrate[f[x], x]
```

For example, if we want to compute the integral of the tangent function, we can use the following commands:

```
(* define a function *)
f[x_] := Tan[x];
(* integrate the function *)
Integrate[f[x], x]

-Log[Cos[x]]
```

To integrate in more than one dimension, simply input your variables in a comma separated list. For example:

```
Integrate[xy^2, x, y]
```

$$\frac{x^2 y^3}{6}$$

While this input produces:

```
Integrate[Cos[x] + Sin[y], x, y, z]

-x z Cos[y] + y z Sin[x]
```

```
Integrate[xy^2, {x, -4, 4}, {y, 0, 2}]
```

Follow the same input conventions used with the graphical integrals. Here we need to make sure there is a space, *, or enclose one of the expressions in parentheses so that the software can correctly interpret all the variables.

To calculate a definite integral, enclose each integration variable in curly braces along with the integration limits. The general syntax is:

```
(* basic syntax for using the Integrate command over a domain *)
Integrate[f, {x, min, max}]
```

For example, to calculate $\int_0^2 xy^2 dx$ we write:

```
Integrate[x^2, {x, 0, 2}]

8/3
```

Exercises

6.1 Show that $\int \frac{1}{\sqrt{1-x^2}}dx$ is the inverse sine function.

6.2 Define a function in terms of the following integral. Compute the integral and plot the result for x ranging over [-1,1]. Then calculate the derivative of the function to make sure you get the integrand back.

$$\int \frac{1}{(5x+14)^5}dx \tag{6.1}$$

6.3 Show that the integral of $\sum_{x=1}^{\infty} x^{-2}$ between $x = \{0,2\}$ is $\frac{\pi^2}{3}$.

6.4 Use *Mathematica* to find the volume of a right circular cone. Integrate in cylindrical coordinates.

6.6 Monte Carlo integration

Mathematica is a very potent tool for symbolic integration and Wolfram states that it is able to compute 95% of the integrals in the classic volume by Gradshteyn and Ryzhik [9]. This is very impressive. Even more impressive, *Mathematica* is also an extremely powerful tool for numerical integration.

Monte Carlo Integration, Monte Carlo Method, Buffon Needle Problem, Buffon-Laplace Needle Problem

Note: Unfortunately the Metropolis method, an important aspect of Monte Carlo integration refers to the nonexistent entry for Simmulated Annealing in [19].

We simply cannot do justice to the integration capabilities built into *Mathematica*. But we would at least like to discuss one topic, Monte Carlo integration. The gist of this technique is that you randomly sample data points and count the ones that are inside the integration domain. Our example should clarify the simplicity of the technique.

Unfortunately Monte Carlo integration is a low-precision technique, so we do not wish to use it in general. However, with the low precision comes a faster execution, so it is a good technique for some problems. Where do you need Monte Carlo integration? If you have a problem that falls into one of these categories:

1. Vast number of integration variables
   Example: quantum theories
2. Highly oscillatory integrals
   Example: $\sin(x^{-2})$ near the origin
3. Cases where the integration domain is difficult to formulate
   Example: a three-dimensional torus

Here is an extremely simple exercise into the topic. We have a square inscribed with a circle and we are throwing darts at the square. All of the darts land within the square and the darts are distributed randomly and uniformly. By simply counting the darts inside and outside of the circle we are able to estimate the ratio of the areas of the two figures. If we consider a circle with origin r centered at the origin, we can write:

$$\frac{\text{umber inside the circle}}{\text{total number}} = \frac{\int_0^r \int_0^{2\pi} \rho \, d\theta \, d\rho}{\int_{-r}^{r} \int_{-r}^{r} dx dy} = \frac{\pi r^2}{4r^2} = \frac{1}{4} \tag{6.2}$$

in the continuum. For our discrete measurement we will count the number of darts that are in the circle (`in`) and the number outside of the circle (`out`). We will use the ratio:

$$\text{area ratio} = \frac{in}{in + out} \tag{6.3}$$

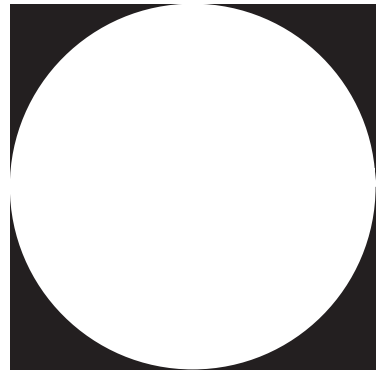
So the integration is reduced to a counting exercise. Notice too that the area ratio is what is computed, not an area. To get an area we would multiply the area of the square by the ratio above.

These are the details. We will build the target first. For convenience we'll collocate the origins of the circle and the coordinate system. Also, the radius π will be one. We'll draw a black square and overlay that with the white circle.

```
(* origin and radius for unit circle *)
o = {0, 0}; ρ = 1;
(* draw the square *)
g01 = Graphics[Rectangle[{-ρ, -ρ}, {ρ, ρ}]];
(* draw the circle *)
g02 = Graphics[{GrayLevel[1], Disk[o, ρ]}];
```

We combine these objects to form a target so that the circle is on top.

```
(* draw the target *)
gtarget = Show[g01, g02, AspectRatio → Automatic];
```



How do we simulate the impact points for the darts? We will need a continuous, uniform distribution. When we want an impact point, we execute `DistRU[-ρ,ρ]`. As always, we have seeded our random number generator. This will allow readers to duplicate the example exactly.

```
(* load up the statistics package *)
Needs["Statistics`ContinuousDistributions`"];
(* choose the uniform distribution *)
DistRU[mn_, mx_] := Random[UniformDistribution[mn, mx]];
(* always seed your rng for reproducibility *)
SeedRandom[1];
```

We are almost ready to begin the simulation. There are some overhead tasks to establish. We need to mark the CPU time so that we are able to measure the time used by this method. To measure the accuracy, we will define a variable with the continuum answer $\pi/4$. We need to specify the number of darts we want to throw and arrange for counters and collectors, counters for the analysis and collectors for the display. Our overhead tasks look like this.

```
(* start the clock *)
t0 = TimeUsed[];
(* analytic computation of ratio (n → ∞): for accuracy *)
ar = π / 4;
(* number of darts *)
n = 1000;
(* counter variables *)
{in, out} = {0, 0};
(* buckets for dart impact points *)
ptsin = {}; ptsout = {};
```

We are now ready to begin throwing the darts. The loop is rather simple. We generate an address pair `{x, y}`; this is the dart impact point. Then we measure to see if the point is inside the circle. Once we know that, we increment the proper counter and add the point to the proper list.

```
(* loop throw the dart throws *)
Do[
  (* throwing darts *)
  {x, y} = {DistRU[-1, 1], DistRU[-1, 1]};
  (* register dart impact *)
  If[x^2 + y^2 ≤ 1,
   in += 1; AppendTo[ptsin, {x, y}],
   out += 1; AppendTo[ptsout, {x, y}];
   , "Error! Null result"];
  , {i, n}];
```

The analysis of the data is almost anticlimactic. We just need to compute the ratio shown in equation 6.3. For comparison, we show the exact result and then quantify the accuracy.

```
(* percentage of darts landing within the circle *)
ratio = in / (in + out);
(* print the results *)
Print[" measured ratio = ", ratio // N];
Print[" analytic ratio = ", ar // N];
Print[" accuracy error = ", (ar - ratio) / ar 100 // N, "%"];
Print["time for ", n, " darts = ", TimeUsed[] - t0, " s"];
```

```
 measured  ratio  = 0.767

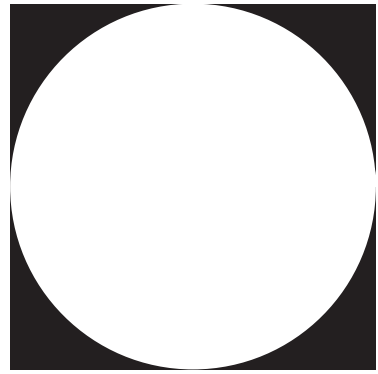
 analytic  ratio  = 0.785398

 accuracy  error  = 2.34253 %

1000  darts : 0.07  sec
```

We would be negligent if we did not take advantage of the visualization tool kit to see what the data look like.

```
(* draw the dartboard -
  always a good idea to visulize your results *)
(* plot the impact points inside of the circle *)
g01 = ListPlot[ptsin, PlotStyle → GrayLevel[0.1], doff];
(* plot the impact points outside of the circle *)
g02 = ListPlot[ptsout, PlotStyle → GrayLevel[0.9], doff];
(* superimpose the impact points on the target *)
g03 = Show[gtarget, g01, g02, don];
```



When you look at this figure, keep in mind that the process of measuring the area of the circle was reduced to counting the number of darts that landed inside and the total number of darts thrown.

Let's call this the explicit method. If you are new to *Mathematica,* this is easy to follow. User's should strive to use the full power of *Mathematica* and the next example exploits functions to simplify the coding. Specifically, we will replace the loop where we throw darts one at a time:

```
(* throwing a dart *)
{x, y} = {DistRU[-1, 1], DistRU[-1, 1]};
```

with a command that throws all 1000 at once:

```
(* throw all the darts at once *)
pts = Table[{DistRU[-1, 1], DistRU[-1, 1]}, {n}];
```

In the former method we had a do-loop that gave us the address of impact point for each dart. As we generated them, we were able to sort them. In the latter method, the impact points are bundled together and handed to us. How do we sort them?

By this juncture the reader probably suspects we will use a pure function. Indeed we will. With the do-loop structure abandoned, the pure function will do our sorting. Compare this sorting procedure to the previous sort. The process is the same; we just have a different syntax.

```
(* define scoring rules: inside the circle or outside *)
score = (
    If[#[[1]]^2 + #[[2]]^2 ≤ 1,
     in += 1; AppendTo[ptsin, {x, y}],
     out += 1; AppendTo[ptsout, {x, y}],
     "Error! Null result"
    ]
   ) &;
```

What's happening is that an {x, y} address is grabbed and checked to see whether it is within the circle. If it is, the `in` counter is incremented and the point is added to the list of points inside the circle, `ptsin`. If outside of the circle, the counter `out` is incremented and the point is added to the list of points outside of the circle, `ptsout`. The third condition should never be reached. A point is either inside or on the circle or outside of it. However, good programming discipline drives us to include the `Null` case condition as a useful debugging tool.

To use this function `score`, we simply `Apply` the function across the points list as shown here:

```
(* score the impacts *)
score /@ pts;
```

This new formulation produces the same results in about a third less time.

One practice that is far too often ignored is measuring the accuracy of the computation. We cringe when people take two uncertain tools and compare their results trying to make statements about accuracy. One of the great advantages of modeling physical systems is that you are able to measure the accuracy of the computation. We will demonstrate using this case.

For this problem we know that the circle covers $\pi/4$ of the area of a square. That is the exact answer that we are seeking. As you can see in the printed summary, we do provide a quantified accuracy number. But we are curious about how the number of points improves the accuracy.

To study the simulations march toward the continuum solution, we analyzed large numbers of points and we probed the results periodically. The results are charted in table 6.1 below. For the first run represented by the top row, we used a million darts and we interrogated the system every 1,000 throws to see what the area value was. The left-hand side of the first row is the best presentation for we did not specify `PlotRange` and we relied upon *Mathematica* to crop the plot for us. In the second graph, on the right, we forced the `PlotRange` to `All` and lost some detail.

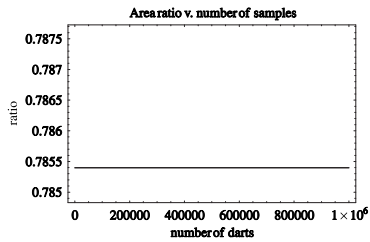
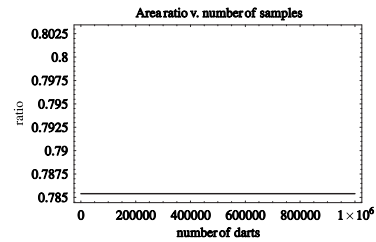
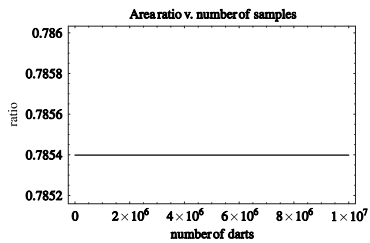
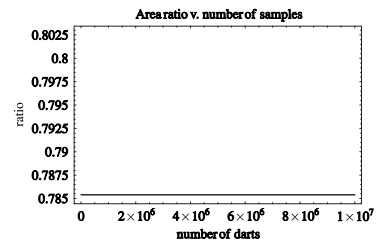
default plot range	PlotRange ⟶ All
	
1000 runs of 1000 points	1000 runs of 1000 points
	
10,000 runs of 1000 points	10,000 runs of 1000 points

Table 6.1: Using Monte Carlo integration to measure the area of a circle. Notice the difference the PlotRange setting makes. Also notice the lack of precision improvement with more data points

We find an expected behavior. As we increase the number of samples the accuracy improves: the points march toward the line representing the exact answer.

Then we considered the same string of numbers but this time going up to 10 million samples. These data are shown on the bottom row of table 6.1. These data show that the additional measurements are wasted because they do not translate into an better computation.

C H A P T E R 7

Special functions

Bessel functions, Gamma functions, and other named mathematical objects frequently find their way into many applications in mathematical physics and engineering. Fortunately, *Mathematica* is well suited to handling special functions. In this chapter we will introduce the reader to the use of a few special functions within the *Mathematica* environment. We begin by considering the Gamma or Factorial function.

7.1 The Gamma function

The factorial function n! is given by the following definition:

$$n! = n \cdot (n-1) \cdot (n-2) \ldots 2 \cdot 1 \tag{7.1}$$

We assume most readers are familiar with this concept. In *Mathematica*, the factorial of any integer can be computed using the `Factorial` function. For example, we can construct a table that lists the factorial of the integers from 1 to 20:

```
Table[Factorial[i], {i, 1, 20}] // TableForm
```

```
1
2
6
24
120
720
5040
40320
362880
3628800
39916800
479001600
6227020800
87178291200
1307674368000
20922789888000
355687428096000
6402373705728000
121645100408832000
2432902008176640000
```

It turns out that the factorial can be generalized by introducing the Gamma function. This is typically written as a function of z, $\Gamma(z)$, and it also gives us the factorial of any integer. In fact we have:

$$\Gamma(z) = (n-1)! \tag{7.2}$$

In *Mathematica*, we can compute the value of the Gamma function for any z by writing `Gamma[z]`. Here we build the previous table using `Gamma` instead of factorial:

```
Table[Gamma[i], {i, 1, 20}] // TableForm
```

```
1
1
2
6
24
120
720
5040
40320
362880
3628800
39916800
479001600
6227020800
87178291200
1307674368000
20922789888000
355687428096000
6402373705728000
121645100408832000
```

As we stated earlier, however, the Gamma function allows us to generalize the Gamma function to any *z*. For example we can compute the Gamma function for real numbers, even if they are negative:

```
Print["Γ(3/2) = ", Gamma[3/2]];
Print["Γ(5.6) = ", Gamma[5.6]];
Print["Γ(-3.4) = ", Gamma[-3.4]];
Print["Γ(-2) = ", Gamma[-2]];
```

$$\Gamma\left(\frac{3}{2}\right) = \frac{\sqrt{\pi}}{2}$$

Γ(5.6) = 61.5539

Γ(-3.4) = 0.325891

Γ(-2) = ComplexInfinity

It can also be used with complex numbers:

```
Gamma[2. + 7. i]

-0.000648833 + 0.000444865 i
```

Note, however, that the values specified as the real and imaginary parts must be entered as reals. If you type the input as 2 + 7*i*, *Mathematica* is not able to calculate the result:

```
Gamma[2 + 7 i]

-0.000648833 + 0.000444865 i
```

The Gamma function is defined via the Euler integral, which is written as:

$$\Gamma(z) = \int_0^\infty e^{-t} t^{z-1} dt \tag{7.3}$$

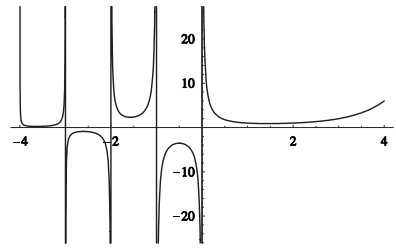
As we so often see *Mathematica* reflects mathematics:

```
(* comparing the command to the integral *)
Gamma[5/2] == ∫_0^∞ e^-t t^(5/2-1) dt

True
```

It's easy to plot the Gamma function in *Mathematica.* For example:

```
(* function with boosted resolution to capture tight curves *)
g1 = Plot[Gamma[x], {x, -4, 4}, PlotPoints → 200];
```



Example
The function:

$$f(x) = \frac{\pi}{\sin(\pi x)} \tag{7.4}$$

can be written in terms of Gamma functions. We can do this by considering $\Gamma(x)$ in combination with $\Gamma(1\text{-}x)$. The defining function is:

```
(* natural expression *)
f[x_] = π / Sin[π x];
(* equivalent formulation using Γ *)
g[x_] = Gamma[x] Gamma[1 - x];
```

This allows us to strictly test the equivalence of the answers.

```
(* compare numerical values *)
f[1.6] == g[1.6]

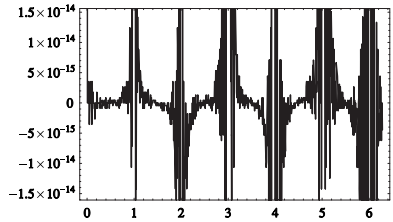
True
```

Now let's look at a graphical comparison. Here we form a difference function. If the two formulations are identical, the difference will be zero at all points.

```
(* define the difference function *)
diff[x_] := g[x] - f[x]
```

The plot is quite interesting:

```
(* plot the difference between the two formulations *)
g3 = Plot[diff[x], {x, 0, 2 π}];
```



What we are seeing here is double precision machine noise. As we see, *Mathematica* accounts for that in the comparison:

```
(* what is the difference? *)
f[1.6] -g[1.6]

4.44089×10^-16
```

There are several other functions of interest that are related to the Gamma function. One of these is the Euler Beta function, $B(x,y)$. It is related to the Gamma function via the relationship:

$$B(x,\ y) = \frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)} \tag{7.5}$$

In *Mathematica*, you can obtain values of this function by entering `Beta[a,b]`. Here we verify the relationship between the Gamma and Beta functions:

```
a = Gamma[3] Gamma[4] / Gamma[7]
b = Beta[3, 4]

1/60
```

```
1/60
```

We see that the answers are expressed in rational numbers, so this equivalence is exact.

Exercise

Products of even or odd numbers are represented by the double factorial function. For example:

$$(2n+1)!! = (2n+1)\cdot\ (2n-1)..3\cdot\ 1 \tag{7.6}$$

$$(2n)!! = 2n\cdot\ (2n-2)..4\cdot\ 2 \tag{7.7}$$

7.1 Using the `Factorial2` function in *Mathematica* show that:

$$\int_0^\infty x^{14} e^{-x^2} dx = \frac{13!!}{2^8}\sqrt{\pi} \tag{7.8}$$

One interesting application of the Gamma function is that it can be used to calculate the surface area and volume of a sphere in any number of dimensions. If we call the number of dimensions *n*, the surface area of a sphere is found from the formula:

$$V_n = R\frac{2\pi^{n/2}}{\Gamma(n/2)} \tag{7.9}$$

where *R* is the radius of the sphere. Let's compute a few examples:

```
(* 3 dimensions, surface area of a sphere *)
R^2 (2 π^(3/2))/Gamma[3/2]
```

$$4\,\pi\,R^2$$

```
(* 2 dimensions, circumference of a circle *)
R (2 π)/Gamma[1]
```

$$2\,\pi\,R$$

```
(* 10 dimensions *)
R^9 (2 π^5)/Gamma[5]
```

$$\frac{\pi^5\,R^9}{12}$$

Now to find the volume of a sphere in *n* dimensions, we use the formula:

```
(* volume of a sphere in n dimensions *)
Vol[n_Integer, R_] := R^2 π^(n/2) / Gamma[1 + n/2]
```

The trial case is well-known.

```
(* volume of a sphere in familiar 3 dimensions *)
Vol[3, R]
```

$$\frac{4\,\pi\,R^2}{3}$$

Example: Polynomial approximation to a sphere

The following example is borrowed from an optics application as was discussed in section 2.4. We consider a Taylor expansion in two dimensions for a hemisphere. The expansion through order *d* is:

$$\sum_{i=0}^{d}\sum_{j=0}^{i} a_{i-j,\,j}\,x^{i-j}y^{j} \tag{7.10}$$

Our goal is to find the amplitudes *a* which best describe the hemisphere:

$$\psi(x,\,y) = \sqrt{1-x^2-y^2}\,. \tag{7.11}$$

The least squares fit requires us to evaluate integrals of the form:

$$\int_D \left(\sum_{\mu=0}^{d}\sum_{\nu=0}^{\mu} a_{\mu-\nu,\,\nu}\,x^{\mu-\nu}y^{\nu}\right) x^{\xi}y^{\eta}dx^2 = 0 \tag{7.12}$$

where the domain *D* is the unit disk; D_2 and the exponents ξ and η are integers. An interesting aspect about this problem is that the general case is easier to solve than the specific case. We proceed with generalization directly.

Let *p* be a vector containing the exponents of the monomials in the Taylor expansion. As an example, the vector for a second order expansion is:

$$\text{: ((0, 0), (1, 0), (0, 1), (2, 0), (1, 1), (0,} \tag{7.13}$$

The integral in equation (7.14) represents the matrix *A* in the linear equations $Aa = B$ where a is the solution: the amplitude vector we are solving for. Using the *p* vector the matrix elements are expressed simply as:

$$A_{ij} = \int_D x^{p_{i_1}+p_{j_1}} y^{p_{i_2}+p_{j_2}} dx^2 \tag{7.14}$$

As you might imagine, these integrals require some time; a great deal of time in fact. Let's look at the direct attack and compare that to using special functions. First, we need to specify the system we are in.

```
(* define the problem *)
d = 3;  (* degree of expansion *)
(* vector of exponents *)
p = Flatten[Table[{i - j, j}, {i, 0, d}, {j, 0, i}], 1];
(* number of terms in the expansion *)
n = Dimensions[p][[1]];
Print[
  "the powers for the xy monomials through degree ", d, " are ", p];
```

```
the powers for the xy monomials through degree 3 are
 {{0, 0}, {1, 0}, {0, 1}, {2, 0}, {1, 1}, {0, 2}, {3, 0}, {2, 1}, {1, 2}, {0, 3}}
```

Now we will follow the recipe in equation 7.14.

```
(* do the integrals explicitly *)
t0 = TimeUsed[];
A1 = Table[Simplify[
      ∫_-r^r ∫_-√(r^2-x^2)^√(r^2-x^2) x^P[[i]][[1]]+P[[j]][[1]] y^P[[i]][[2]]+P[[j]][[2]] dy dx, r > 0]
    , {i, n}, {j, n}];
t1 = TimeUsed[];
Print["Total time to execute ", t1 - t0 // N, " s"]
A1 // MatrixForm
```

```
Total time to execute 23.934 s
```

$$\begin{pmatrix}
\pi r^2 & 0 & 0 & \frac{\pi r^4}{4} & 0 & \frac{\pi r^4}{4} & 0 & 0 & 0 & 0 \\
0 & \frac{\pi r^4}{4} & 0 & 0 & 0 & 0 & \frac{\pi r^6}{8} & 0 & \frac{\pi r^6}{24} & 0 \\
0 & 0 & \frac{\pi r^4}{4} & 0 & 0 & 0 & 0 & \frac{\pi r^6}{24} & 0 & \frac{\pi r^6}{8} \\
\frac{\pi r^4}{4} & 0 & 0 & \frac{\pi r^6}{8} & 0 & \frac{\pi r^6}{24} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \frac{\pi r^6}{24} & 0 & 0 & 0 & 0 & 0 \\
\frac{\pi r^4}{4} & 0 & 0 & \frac{\pi r^6}{24} & 0 & \frac{\pi r^6}{8} & 0 & 0 & 0 & 0 \\
0 & \frac{\pi r^6}{8} & 0 & 0 & 0 & 0 & \frac{5\pi r^8}{64} & 0 & \frac{\pi r^8}{64} & 0 \\
0 & 0 & \frac{\pi r^6}{24} & 0 & 0 & 0 & 0 & \frac{\pi r^8}{64} & 0 & \frac{\pi r^8}{64} \\
0 & \frac{\pi r^6}{24} & 0 & 0 & 0 & 0 & \frac{\pi r^8}{64} & 0 & \frac{\pi r^8}{64} & 0 \\
0 & 0 & \frac{\pi r^6}{8} & 0 & 0 & 0 & 0 & \frac{\pi r^8}{64} & 0 & \frac{5\pi r^8}{64}
\end{pmatrix}$$

The point of this exercise is to demonstrate why the direct approach is undesirable. Instead, let's take advantage of *Mathematica*'s symbolic tool kit and evaluate the result analytically.

```
(* first attempt *)
∫_-r^r ∫_-√(r^2-x^2)^√(r^2-x^2) x^n y^m dy dx

If[r > 0 && Re[m] > -3 && Re[n] > -1,
 ((1 + (-1)^m) (1 + (-1)^n) r^(2+m+n) Gamma[(1+m)/2] Gamma[(1+n)/2])/(4 Gamma[1/2 (4 + m + n)]),
 Integrate[(x^n (r^2 - x^2)^((1+m)/2))/(1 + m) + ((-1)^m x^n (r^2 - x^2)^((1+m)/2))/(1 + m),
  {x, -r, r}, Assumptions → ! (r > 0 && Re[m] > -3 && Re[n] > -1)]]
```

After a little time pondering the parity of the problem, the solution can be extracted. The germane parity considerations are these. We are integrating over two symmetric domains. We may, for example, consider the *y* domain to be from $\{-\alpha, \alpha\}$. The only thing that will survive this integration is an even function. (Recall equations 3.8 and 3.9.) Also, the *x* domain is similarly even and will filter out even functions or the even components of functions. The perplexed reader is encouraged to use the interactive environment to investigate.

What you will eventually come to is that

$$\int_D x^n y^m dy dx = \frac{4}{m+1} \frac{r^{2+n+m}\Gamma\left(\frac{n+1}{2}\right)\Gamma\left(\frac{m+3}{2}\right)}{2\Gamma\left(\frac{1}{2}(4+n+m)\right)} \tag{7.15}$$

and the only nonzero answers are when the powers of *x* and *y*, *n* and *m*, are both even.

So we are looking at a function definition like this:

```
(* basic formula *)
integral[n, m] = 4/(m + 1) (r^(2+m+n) Gamma[(3+m)/2] Gamma[(1+n)/2])/(2 Gamma[1/2 (4 + m + n)])
```

to load the *A* matrix. Except that we want to do so more efficiently. If we look at the *A* matrix, it is obviously symmetric since

$$p_{i_k} + p_{j_k} = p_{j_k} + p_{i_k}. \tag{7.16}$$

What is less apparent but still true is that besides this symmetry, many terms are repeated. We are loathe to waste computation time.

Fortunately, the sophisticated design of *Mathematica* allows us to write a function that guarantees that each unique matrix element will be computed once and only once. The basic syntax is

```
(* even-even parity cases *)
integral[n_?EvenQ, m_?EvenQ] := integral[n, m] =. . .;
(* all other cases *)
integral[n_Integer, m_Integer] := integral[n, m] = 0;
```

where the ellipsis represents the formula in equation 7.15. Notice that we have combined the `SetDelay` operator (`:=`) with the `Set` operator (`=`). This will cause *Mathematica* to first inspect the functional definition and see if the requested problem has been solved yet. If the value has been computed, the kernel returns that value. If the value has not been computed, then the kernel will evaluate the function in the place of the ellipsis. Note too that we have bypassed evaluation of the terms with improper parity and have forced them to zero. The following example should clarify the process.

First, we load the function definition:

```
(* even-even parity cases *)
integral[n_?EvenQ, m_?EvenQ] :=
  integral[n, m] = 4/(m+1) r^(2+m+n) Gamma[(3+m)/2] Gamma[(1+n)/2] / (2 Gamma[1/2 (4+m+n)]);
(* all other cases *)
integral[n_Integer, m_Integer] := integral[n, m] = 0;
```

Let's check the definition.

```
?integral
```

```
Global`integral

integral [n_?EvenQ , m_?EvenQ ] := integral [n, m] = 4 (r^(2+m+n) Gamma[(3+m)/2] Gamma[(1+n)/2]) / ((m+1) (2 Gamma[1/2 (4+m+n)]))

integral [n_Integer , m_Integer ] := integral [n, m] = 0
```

Now let's compute the *A* matrix.

```
(* use recursive functions *)
t0 = TimeUsed[];
A2 = Table[n = p[[i]][[1]] + p[[j]][[1]]; m = p[[i]][[2]] + p[[j]][[2]];
   integral[n, m]
   , {i, n}, {j, n}];
t1 = TimeUsed[];
Print["Time required using Gamma functions ", t1 - t0]
A2 // MatrixForm
```

```
Time required using Gamma functions 0.01
```

$$\begin{pmatrix} \pi r^2 & 0 & 0 & \frac{\pi r^4}{4} & 0 & \frac{\pi r^4}{4} & 0 & 0 & 0 & 0 \\ 0 & \frac{\pi r^4}{4} & 0 & 0 & 0 & 0 & \frac{\pi r^6}{8} & 0 & \frac{\pi r^6}{24} & 0 \\ 0 & 0 & \frac{\pi r^4}{4} & 0 & 0 & 0 & 0 & \frac{\pi r^6}{24} & 0 & \frac{\pi r^6}{8} \\ \frac{\pi r^4}{4} & 0 & 0 & \frac{\pi r^6}{8} & 0 & \frac{\pi r^6}{24} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{\pi r^6}{24} & 0 & 0 & 0 & 0 & 0 \\ \frac{\pi r^4}{4} & 0 & 0 & \frac{\pi r^6}{24} & 0 & \frac{\pi r^6}{8} & 0 & 0 & 0 & 0 \\ 0 & \frac{\pi r^6}{8} & 0 & 0 & 0 & 0 & \frac{5\pi r^8}{64} & 0 & \frac{\pi r^8}{64} & 0 \\ 0 & 0 & \frac{\pi r^6}{24} & 0 & 0 & 0 & 0 & \frac{\pi r^8}{64} & 0 & \frac{\pi r^8}{64} \\ 0 & \frac{\pi r^6}{24} & 0 & 0 & 0 & 0 & \frac{\pi r^8}{64} & 0 & \frac{\pi r^8}{64} & 0 \\ 0 & 0 & \frac{\pi r^6}{8} & 0 & 0 & 0 & 0 & \frac{\pi r^8}{64} & 0 & \frac{5\pi r^8}{64} \end{pmatrix}$$

Compare the times: the first method took 23.93 seconds; this method took 0.01 seconds. Are the answers the same?

```
(* compare the two results *)
A1 == A2

True
```

So the answer is the same, and it is a much faster way. Let's look at how recursion prevented redundant computations. Look at the function integral now:

```
?integral
```

Global`integral

integral [3, 2] = 0

integral [2, 0] = $\frac{\pi r^4}{4}$

integral [2, 2] = $\frac{\pi r^6}{24}$

integral [1, 0] = 0

integral [2, 4] = $\frac{\pi r^8}{64}$

integral [1, 2] = 0

integral [0, 0] = πr^2

integral [1, 4] = 0

integral [0, 2] = $\frac{\pi r^4}{4}$

integral [0, 4] = $\frac{\pi r^6}{8}$

integral [0, 6] = $\frac{5 \pi r^8}{64}$

integral [5, 1] = 0

integral [4, 1] = 0

integral [3, 1] = 0

integral [3, 3] = 0

integral [2, 1] = 0

integral [2, 3] = 0

integral [1, 1] = 0

integral [1, 3] = 0

integral [6, 0] = $\frac{5 \pi r^8}{64}$

integral [5, 0] = 0

integral [4, 0] = $\frac{\pi r^6}{8}$

integral [0, 1] = 0

integral [1, 5] = 0

```
integral [4, 2] = π r^8 / 64

integral [0, 3] = 0

integral [3, 0] = 0

integral [0, 5] = 0

integral [n_?EvenQ , m_?EvenQ ] := integral [n, m] = 4 (r^(2+m+n) Gamma[(3+m)/2] Gamma[(1+n)/2]) / ((m+1) (2 Gamma[1/2 (4+m+n)]))

integral [n_Integer , m_Integer ] := integral [n, m] = 0
```

As a value is computed, it is stored at the top of the definition and these already computed cases are what the kernel looks for first.

The clever reader is wondering why we haven't taken notice of the symmetry relation

$$integral[n, m] = integral[m, n]. \tag{7.17}$$

We leave it as an exercise.

7.2 The Bessel functions

A frequently encountered beast that often shows up in applications of electromagnetism is the Bessel function. There are actually several "kinds" of Bessel functions and we refer the reader to *The CRC Concise Encyclopedia of Mathematics*.



Bessel Differential Equation, Bessel Function of the First Kind, Bessel Function of the Second Kind, Hankel Function.

We begin with $J_\nu(x)$, which *Mathematica* refers to as the `BesselJ` function. This function represents the solutions to Bessel's differential equation

$$x^2y'' + xy' + (x^2 - \nu^2)y = 0 \tag{7.18}$$

which are not singular at the origin.

In *Mathematica* we define this equation in the following manner:

```
(* Bessel's differential equation *)
bdiffeq = x^2 y''[x] + x y'[x] + (x^2 - ν^2) y[x] == 0;
```

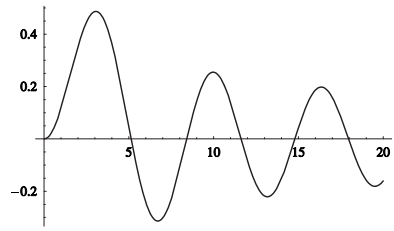
When we apply `DSolve` to the equation, we get:

```
(* solve Bessel's differential equation *)
DSolve[bdiffeq, y[x], x]

{{y[x] → BesselJ[ν, x] C[1] + BesselY[ν, x] C[2]}}
```

a superposition of Bessel functions of the first and second kind. Plots of the Bessel functions of the first kind show that they are oscillating, but nonperiodic:

```
(* oscillatory, but not periodic *)
g1 = Plot[BesselJ[2, x], {x, 0, 20}];
```



Let's examine the first few together. We will plot the odd parity functions, and then the even parity functions. The plots are quite interesting. (We only show the code to produce the odd parity functions. The even parity case is a trivial change.)

```
(* examine the first few odd parity Bessel functions *)
btable = Table[BesselJ[i, x], {i, 1, 5, 2}];
(* darkest curves are earliest *)
gtable = Table[GrayLevel[2 (i - 1) / 10], {i, 1, 5, 2}];
(* display the curves *)
g1 = Plot[Evaluate[btable], {x, -15, 15}, PlotStyle → gtable];
```

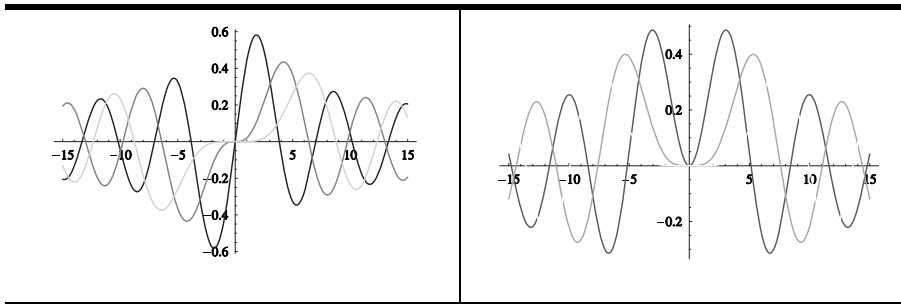


Table 7.1: The first few odd and even parity Bessel functions of the first kind.

Near the origin the `BesselJ` function is related to the `Gamma` function. The limiting form is given by:

$$J_\nu(x) \to \frac{1}{\Gamma(\nu+1)}\left(\frac{x}{2}\right)^\nu \tag{7.19}$$

For example, we have the limiting form of $J_2(x)$:

```
(* Bessel function *)
f1[x_] = BesselJ[2, x];
(* approximation valid near origin *)
f2[x_] = 1/Gamma[3] (x/2)^2;
```

Let's use *Mathematica* to compare these two forms: `f1` the exact, and `f2` the approximate.

```
(* Bessel function *)
f1[x_] = BesselJ[2, x];
(* approximation valid near origin *)
f2[x_] = 1/Gamma[3] (x/2)^2;
```

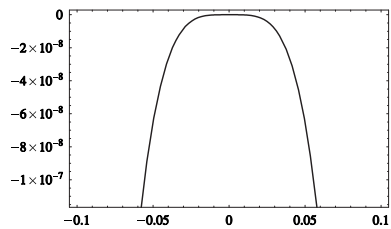
As usual we will not compare the two numbers; we will check the difference between them.

```
(* value near origin *)
x = 0.025;
(* difference between exact and approximate forms *)
diff = f1[x] - f2[x];
Print["at x = ", x, " exact - approximate = ",
  diff, " (", (f1[x] - f2[x])/f1[x] 100, " %)"];
```

```
at x = 0.025  exact - approximate  = -4.06893 ×10^-9 (-0.0052085 %)
```

Similarly, for the plot, the two curves are not distinguishable, so we will plot the difference between them:

```
(* the difference plot is more helpful *)
g2 = Plot[f1[x] - f2[x], {x, -0.1, 0.1}, Axes → False, Frame → True];
```



The derivatives of Bessel functions satisfy several recursive relations, which can be verified using *Mathematica*. For example:

```
(* we can create recursion relationships *)
∂x BesselJ[n, x]
```

$$\frac{1}{2}\,(\text{BesselJ}[-1+n,\,x]-\text{BesselJ}[1+n,\,x])$$

Example: Simple model of a nuclear reactor

An important quantity is the study of a reactor design in the neutron flux given in neutrons per square centimeter per second. In this example we consider a simple model of a reactor given by an infinite cylinder. The flux ϕ obeys the homogeneous reactor equation:

$$\frac{\partial^2 \phi}{\partial r^2} + \frac{1}{r}\frac{\partial \phi}{\partial r} + B^2 \phi = 0 \tag{7.20}$$

B is a constant, known as the *Buckling constant*. Its meaning won't concern us here, so we will just note that it is a constant we will find using the boundary conditions of the problem. This equation is known as the reactor equation. Solve the equation and plot the solutions. We begin by defining the equation and solving with `DSolve`:

```
(* the reactor equation for neutron flux *)
f[r_] := ϕ''[r] + 1/r ϕ'[r] + B^2 ϕ[r] == 0
```

```
(* solve the reactor equation *)
soln = DSolve[f[r], ϕ[r], r]

{{ϕ[r] → BesselJ[0, B r] C[1] + BesselY[0, B r] C[2]}}
```

The solution is given in terms of two Bessel functions. Since this is a second order equation, there are two undetermined constants. We will use some physical intuition and examine the limiting behavior of the Bessel functions. Let's compute the limits as $r \to 0$ and $r \to \infty$ before we plot the functions. First, we will put the equations in a list.

```
(* put the functions of interest in a list *)
fcns = {BesselJ[0, r], BesselY[0, r]};
```

Then we will examine behavior near the origin:

```
(* r → 0 limits *)
Print["limit as r→0 for ", fcns, ":"];
Limit[#, r → 0] & /@ fcns
```

```
limit as r→0 for {BesselJ [0, r], BesselY [0, r]}:

{1, -∞}
```

and finally as the radius increases without bound

```
(* r → ∞ limits *)
Print["limit as r→∞ for ", fcns, ":"];
Limit[#, r → ∞] & /@ fcns
```

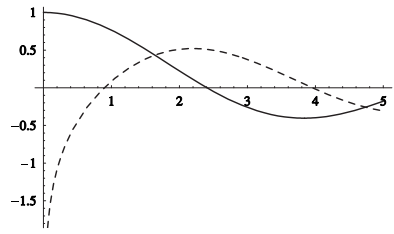
```
limit  as  r→∞ for  {BesselJ [0, r], BesselY [0, r]}:
```

```
{0, 0}
```

Both functions go to zero as $r \to \infty$, which is physically reasonable. We expect that as you move far from the reactor the neutron flux will drop to zero. However, we notice that the `BesselY` function in not finite at the origin. This is not physically reasonable, because the neutron flux must remain finite at the core. Let's plot the two functions, using a dashed line for the `BesselY` function:

```
(* dashing conditions *)
dash = {Dashing[{0.01, 0}], Dashing[{0.01, 0.01}]};
```

```
(* comparing BesselJ and BesselY *)
g1 = Plot[{BesselJ[0, r], BesselY[0, r]}, {r, 0, 5}, PlotStyle → dash];
```



The plot clearly shows that the `BesselY` function is not physically reasonable, and therefore cannot represent the neutron flux. Therefore the constant `C[2]` in the solution must be set to zero. We extract the solution into a function variable, applying the condition that `C[2]` is zero.

```
(* harvest the solution from DSolve *)
g[r_] = soln[[1,1,2]] /. C[2] → 0

BesselJ[0, B r] C[1]
```

To determine the Buckling constant B we apply a boundary condition which is based on the zeros of the Bessel function. Suppose that we want the flux to vanish at the outer radius a of the reactor. Then we have:

$$\phi(a) = C[1]J_0(Br) \tag{7.21}$$

By determining the zeros of the Bessel function, we can find B in terms of the radius a of the reactor. We will investigate several methods to find the zeros of the Bessel function. To find the first zero, lets expand the Bessel function in a series, taking only a few terms:

```
(* looking for zeros of the Bessel functions *)
order = 8;      (* define expansion order *)
(* Taylor expansion *)
series1 = Series[BesselJ[0, r], {r, 0, order}]
```

$$1 - \frac{r^2}{4} + \frac{r^4}{64} - \frac{r^6}{2304} + \frac{r^8}{147456} + O[r]^9$$

We can solve this eighth order polynomial for $r = 0$. First, we need to eliminate the big O terms on the end using the `Normal` command.

```
(* trim off the remaining error estimate *)
Normal[series1]
```

$$1 - \frac{r^2}{4} + \frac{r^4}{64} - \frac{r^6}{2304} + \frac{r^8}{147456}$$

Now we ask for the roots.

```
(* roots of BesselJ[0,r] *)
root1 = Solve[Normal[series1] == 0, r] // N

{{r → -4.65672}, {r → -2.40564}, {r → 2.40564}, {r → 4.65672},
 {r → -5.12554 - 2.8297 i}, {r → 5.12554 - 2.8297 i},
 {r → -5.12554 + 2.8297 i}, {r → 5.12554 + 2.8297 i}}
```

There are several roots here, but only the real roots are significant, and furthermore only positive roots have any meaning. We can find the roots of interest by looping through each case:

```
(* length of roots list *)
n = Length[root1];
(* object holding the numeric roots *)
froot = {};
Do[
  (* grab the numeric value from the rule *)
  x = r /. root1[[i]];
  (* test skips complex,negative and 0 *)
  If[NonNegative[x], AppendTo[froot, x]];
  , {i, n}];
Print["roots of interest: ", froot];
```

```
roots of interest : {2.40564 , 4.65672 }
```

Another way to accomplish this task is to write a function that sweeps through the roots and extracts nonnegative real values. We call this function `sweep`:

```
(* function that sweeps for real, positive roots *)
sweep = (x = r /. #; If[NonNegative[x], AppendTo[froot, x]]) &;
(* object holding the numeric roots *)
froot = {};
(* apply the function to the list of rules *)
sweep /@ root1;
Print["roots of interest: ", froot];
```

```
roots of interest : {2.40564 , 4.65672 }
```

But as is so often the case, we find the exact tools we need are available to us. First, we create a list of numerical values, not substitution rules. Notice how we can take care of the entire list as easily as we would convert one element in the list.

```
(* a more refined way *)
(* build a list of the numeric values for the roots *)
nroot = r /. root1

{-4.65672, -2.40564, 2.40564, 4.65672, -5.12554 - 2.8297 i,
 5.12554 - 2.8297 i, -5.12554 + 2.8297 i, 5.12554 + 2.8297 i}
```

The `Select` command will do the rest of the work for us.

```
(* grab the nonnegative values (this will also exclude complex) *)
froot = Select[nroot, NonNegative]

{2.40564, 4.65672}
```

There is, however, an even better way to accomplish this task when working with Bessel functions. We can use the `BesselJZeros` function that is built into *Mathematica*. Let's find the roots one more time. The root of interest is the smallest root, so after finding the roots we will apply the `Min` function to the list to extract it. This approach works like this:

```
(* we'll see that the package is far faster and more precise *)
<< NumericalMath`BesselZeros`
roots2 = BesselJZeros[0, 5]

{2.40483, 5.52008, 8.65373, 11.7915, 14.9309}
```

The `Min` function returns the root of interest.

```
(* pick the smallest root *)
Rn = Min[roots2]

2.40483
```

The constant B is then determined by:

$$B = \frac{R_n}{a} \tag{7.22}$$

where a is the outer radius of the reactor.

```
a = 2;  (* 2 m radius *)
g[r_] = g[r] /. B → Rn/2

BesselJ[0, 1.20241 r] C[1]
```

There is one final constant left in the problem, `C[1]`. We don't want to dive too far into nuclear engineering here, but we will simply note that this constant can be found in relation to the power P of the reactor via the relation:

$$P = \alpha C[1] \int_0^a \phi dr \tag{7.23}$$

where a is a collection of constants that don't concern us here. This involves integrating the Bessel function. For the case at hand we find that:

```
int = ∫_0^a g[r] dr

1.22279 C[1]
```

and `C[1]` can then be determined from:

```
Solve[P == α int, C[1]]

{{C[1] → 0.817801 P/α}}
```

Example: TE$_z$ modes in a cylindrical cavity

Consider an electromagnetic waveguide with a cylindrical shape. Given a vector potential function F_z which satisfies the wave equation, for frequency ω find the electric and magnetic fields and calculate the impedance for the waveguide.

We are concerned in this case with transverse-electric modes for a wave traveling in the z-direction. This means that the electric field will be zero in the direction of z. The function:

$$F_z = F_z(r, \phi, z)$$

must satisfy the wave equation:

$$\nabla^2 F_z + \beta^2 F_z = 0 \tag{7.24}$$

where the Laplacian is computed in cylindrical coordinates and β is a constant. For simplicity, we also take $\mu = \varepsilon = 1$. If we consider outward traveling waves only, then the function F_z is given by:

$$F_z(r, \phi, z) = A_{mn} H_m^2(\beta r) \cos(m\phi) \sin\left(\frac{n\pi}{z} h\right) \tag{7.25}$$

These are called Hankel functions of the second kind. They can be written in terms of Bessel functions that we have already met via the relation:

$$H_m(\beta r) = J_m(\beta r) - i Y_m(\beta r) \tag{7.26}$$

Hankel functions are not defined in *Mathematica*, so we will construct a function called H_2 to represent it. Let's suppose that we are considering the (1,0) mode only and for convenience set $A_{10} = 1$, $\beta = 1$ and $h = 1$.

```
(* construct the Hankel function *)
H2[B_, r_, n_] := (BesselJ[n, B r] - i BesselY[n, B r]);
```

The force is then given by:

```
(* the force *)
Fz[r_, ϕ_, Θ_] := H2[1, r, 1] Cos[ϕ] Sin[π z];
```

Now we can calculate the elements of the electric and magnetic forces. We start with the electric forces.

```
(* electric field components *)
Er[r_, ϕ_, z_] := -1/r ∂ϕ Fz[r, ϕ, z];
Eϕ[r_, ϕ_, z_] := ∂r Fz[r, ϕ, z];
Ez[r_, ϕ_, z_] := 0;
```

and follow up with the magnetic forces:

```
(* magnetic field components *)
Hr[r_, ϕ_, z_] := FullSimplify[-(i/ω) ∂z ∂r Fz[r, ϕ, z]];
Hϕ[r_, ϕ_, z_] := FullSimplify[-(i/(ω r)) ∂z ∂ϕ Fz[r, ϕ, z]];
Hz[r_, ϕ_, z_] :=
  FullSimplify[-(i/ω) (∂z,z - (i/ω) ∂z ∂r Fz[r, ϕ, z] + β^2 - (i/ω) ∂z ∂r Fz[r, ϕ, z])];
```

Since this is a bit cumbersome, we will generate some useful `Print` statements.

```
(* descriptive output *)
Print["electric field components:"];
Print["Er = ", Er[r, ϕ, z]];
Print["Eϕ = ", Eϕ[r, ϕ, z]];
Print["Ez = ", Ez[r, ϕ, z]];
```

```
electric field components :

Er = (BesselJ[1, r] - i BesselY[1, r]) Sin[π z] Sin[ϕ] / r

Eϕ = ((1/2) (BesselJ[0, r] - BesselJ[2, r]) - (1/2) i (BesselY[0, r] - BesselY[2, r]))
     Cos[ϕ] Sin[π z]

Ez = 0
```

```
(* descriptive output *)
Print["magnetic field components:"];
Print["H_r = ", H_r[r, φ, z]];
Print["H_φ = ", H_φ[r, φ, z]];
Print["H_z = ", H_z[r, φ, z]];
```

```
magnetic field components :
```

$$H_r = \frac{1}{2\omega}\left(\pi\left(-i\,\text{BesselJ}[0,r] + i\,\text{BesselJ}[2,r] - \text{BesselY}[0,r] + \text{BesselY}[2,r]\right)\cos[\pi z]\cos[\phi]\right)$$

$$H_\phi = \frac{\pi\left(i\,\text{BesselJ}[1,r] + \text{BesselY}[1,r]\right)\cos[\pi z]\sin[\phi]}{r\,\omega}$$

$$H_z = \frac{1}{2\omega^2}\left(-2i\beta^2\omega + \pi\left(-\text{BesselJ}[0,r] + \text{BesselJ}[2,r] + i\left(\text{BesselY}[0,r] - \text{BesselY}[2,r]\right)\right)\cos[\pi z]\cos[\phi]\right)$$

Notice that *Mathematica* has applied the familiar relationships between derivatives of the Bessel functions to obtain the form of the electric field. Specifically:

```
∂_r BesselJ[1, r]
```

$$\frac{1}{2}\left(\text{BesselJ}[0,r] - \text{BesselJ}[2,r]\right)$$

Finally, we determine the functional form of the impedance. The impedance in this case is given by the electric field in the ϕ direction divided by the magnetic field in the z-direction:

```
(* impedance *)
imped = Eϕ[r, ϕ, z] / Hz[r, ϕ, z] // FullSimplify

(ω^2 (i BesselJ[0, r] - i BesselJ[2, r] + BesselY[0, r] - BesselY[2, r])
   Cos[ϕ] Sin[π z]) / (2 β^2 ω +
   π (-i BesselJ[0, r] + i BesselJ[2, r] - BesselY[0, r] + BesselY[2, r])
    Cos[π z] Cos[ϕ])
```

Exercises

7.2 Consider the use of other types of Bessel functions in *Mathematica*, specifically the `BesselK` and `BesselI` functions. Compute the derivatives of functions of different orders and see if you can determine recursion relations that exist for these derivatives. Examine plots of the functions.

7.3 Consider the case of TM waves in a cylindrical resonant cavity. The electric and magnetic fields are represented in terms of a separable function

$$\psi(r,\ \phi) = \Psi(r)e^{im\phi} \tag{7.27}$$

where $m = 0, 1, 2, \ldots$ The function obeys a two-dimensional wave equation given by:

$$\frac{\partial^2\psi}{\partial r^2} + \frac{1}{r}\frac{\partial\psi}{\partial r} + \left(\alpha^2 - \frac{m^2}{r^2}\right)\psi = 0 \tag{7.28}$$

Solve this equation and investigate various cases of m.

7.3 The Riemann zeta function

The Riemann zeta function is defined by the series:

$$\zeta(t) = \sum_{j=1}^{\infty} j^{-r}. \tag{7.29}$$

It has wide application in many areas of physics. In *Mathematica*, we can use `Zeta[r]` to calculate $\zeta(r)$. The `Zeta` function is defined over the complex plane, and is usually written as $\zeta(s)$ (instead of using the conventional variable z to represent a complex number). For a real variable $x > 1$, this function is closely related to the `Gamma` function, and can be defined with the integral:

$$\zeta(x) = \frac{1}{\Gamma(x)}\int_0^x \frac{t^{x-1}}{e^t - 1}dt \tag{7.30}$$

Let's verify this in *Mathematica.* We use the `Integrate` command so that we can specify that the real part of $x > 1$:

```
(* Zeta surrogate *)
f[x_] =
  1/Gamma[x] Integrate[t^(x-1)/(e^t - 1), {t, 0, ∞}, Assumptions → Re[x] > 1]

PolyLog[x, 1]
```

Pick a sample value:

```
(* check this function against the intrinsic Zeta function *)
f[4]
Zeta[4]
```

$$\frac{\pi^4}{90}$$

$$\frac{\pi^4}{90}$$

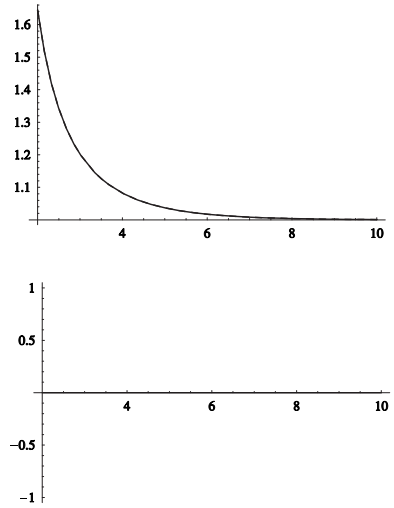
As usual, let *Mathematica* handle the comparison, no matter how obvious it seems.

```
(* are these values the same? *)
%% == %

True
```

When we plot these functions, we see that the curves are indistinguishable and that the difference is exactly zero over the plot domain:

```
(* plot these two functions *)
g1 = Plot[{f[x], Zeta[x]}, {x, 2, 10}, PlotStyle → dash];
g2 = Plot[f[x] - Zeta[x], {x, 2, 10}];
```



$\zeta(s)$ plays an interesting role in many series. For example, consider:

```
4 (1 - 2 Sum[4^(-2 i) Zeta[2 i], {i, 1, Infinity}]) // Simplify
```

```
π
```

Another interesting series result is:

```
γ = 1/2 + Sum[(Zeta[i] - 1)/2^i, {i, 2, Infinity}]
```

```
1/2 + 1/2 (-1 + Log[4])
```

Example: The Fermi-Distribution

We now consider the distribution of particles in a gas consisting of fermions. Fermions are particles like electrons and protons that obey the Pauli exclusion principle. This basically says that no two fermions can exist in the same identical energy state. Here we will investigate the occupation number of the energy states, which is described by a probability distribution known as the Fermi distribution. If we call this distribution $f(\varepsilon)$ where ε is the energy, then:

$$f(\varepsilon) = \frac{1}{e^{\beta(\varepsilon-\mu)}+1} \tag{7.31}$$

where the inverse temperature:

$$\beta = \frac{1}{kT}, \tag{7.32}$$

k is Boltzmann's constant, T is the temperature, and μ is a quantity known as the chemical potential. If you are not familiar with these concepts don't worry; we will be focusing on manipulating this expression with *Mathematica*. First we examine a plot of this distribution. We define the constants:

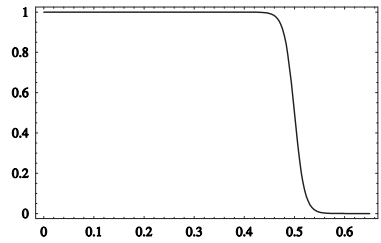
```
(* chemical potential, Boltzmann's constant *)
{μ, β} = {0.5, 100};
```

and the Fermi distribution:

```
(* energy distribution *)
f[ϵ_] := 1/(Exp[β (ϵ - μ)] + 1)
```

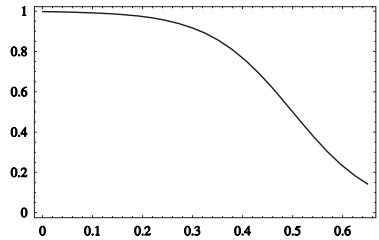
The chemical potential μ will determine where the plot goes to zero. First, we consider a low temperature case.

```
(* energy distribution plot *)
g1 = Plot[f[ϵ], {ϵ, 0, 0.65}, Frame → True];
```



Notice that near $\varepsilon = 0.5$, the value of m, the function drops off steeply to zero after having remained constant for a range of energies. Now we consider a high temperature case. As the temperature increases, the drop off becomes more gradual and the range over which the occupation number is constant shrinks.

```
(* high temperature case *)
β = 12;
(* energy distribution plot *)
g1 = Plot[f[ϵ], {ϵ, 0, 0.65}, Frame → True];
```



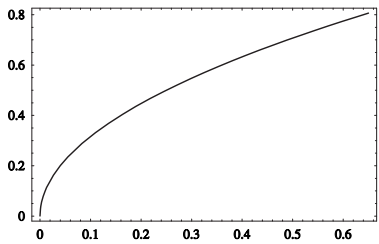
To obtain the actual distribution of particles in the gas, we need to know the distribution of energy states. This distribution is given the label *g(ε)*. Two common forms of this function are:

$$g(\varepsilon) = \sqrt{\varepsilon}\text{, and } g(\varepsilon) = \varepsilon^2 \tag{7.33}$$

Let's examine the first case. Let's plot *g* over the same range of energy:

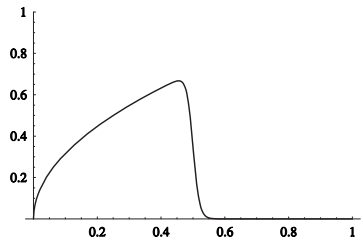
```
(* distribution of energy states *)
g[ϵ_] := √ϵ;
```

```
g1 = Plot[g[ϵ], {ϵ, 0, 0.65}, Frame → True];
```

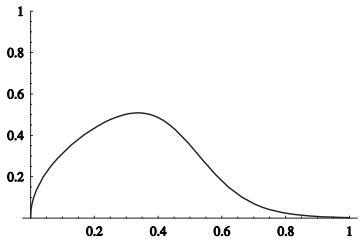


The actual distribution of particles is found by forming the product of these two quantities. We compare the low and high temperature cases:

```
(* low temperature *)
β = 100;
g1 = Plot[g[ϵ] f[ϵ], {ϵ, 0, 1}, PlotRange → {0, 1}];
```




```
(* high temperature *)
β = 12;
g2 = Plot[g[ϵ] f[ϵ], {ϵ, 0, 1}, PlotRange → {0, 1}];
```

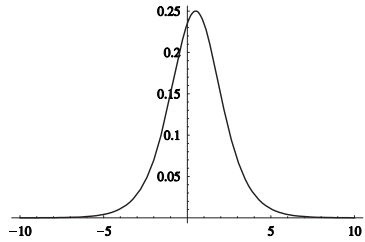


Now consider the derivative of the occupation number function, which will tell us how occupation number changes with energy and examine a plot (recall $\mu = 0.5$ for our example):

```
Clear[β];
(* derivative of the occupation number function *)
h[ϵ_] = ∂ϵ f[ϵ]
```

$$-\frac{e^{\beta(-0.5+\epsilon)}\,\beta}{(1+e^{\beta(-0.5+\epsilon)})^2}$$

```
β = 1;
Plot[-h[ϵ], {ϵ, -10, 10}];
```



When working with probability, it is often of interest to compute the moments of a given distribution. Let's calculate a few of these for *h(ε)*. For the rest of the example we set $x = \varepsilon - \mu$ and $\beta = 1$.

```
(* calculate some moments *)
f[x_] := 1 / (Exp[x] + 1);
h[x_] := ∂x f[x];
```

```
(* print the moments *)
Table[Print["the ", i, "th moment = ", ∫_-∞^∞ x^i h[x] dx], {i, 0, 4}];
```

the 0th moment = −1
the 1th moment = 0
the 2th moment = $-\frac{\pi^2}{3}$
the 3th moment = 0
the 4th moment = $-\frac{7\pi^4}{15}$

We see that odd moments seem to be zero. Even moments are taking on powers of π. Let's look at the parity states of these moments. We will rely upon equations 3.8 and 3.9 to guide us. First, we examine *h*:

```
h[x]
```

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

The parity of this function is not immediately obvious. So to resolve this and the higher moments, let's construct a module which will take a function and separate it into even and odd parity components.

```
(* resovle functions into even and odd parity components *)
parity[f_] := Module[{},
   (* resolve the two parity states *)
   even = 1/2 (f[x] + f[-x]);
   odd = 1/2 (f[x] - f[-x]);
   (* print results *)
   Print["parity analysis for the function ", f[x]];
   Print["even portion = ", Simplify[even]];
   Print["odd portion = ", Simplify[odd]];
  ];
```

Our first test is:

```
g[x_] := -e^x/(1 + e^x)^2;
```

```
parity[g]
```

parity analysis for the function $-\frac{e^x}{(1+e^x)^2}$

even portion $= -\frac{e^x}{(1+e^x)^2}$

odd portion $= 0$

This is good news. We have all even functions. The multiplier function x^n has the same parity as the power n. We just need these simple rules for multiplying functions:

$$even \times even = even \tag{7.34}$$

$$even \times odd = odd \tag{7.35}$$

$$odd \times odd = even \tag{7.36}$$

For this application we only need the first two cases since at least one function is always even.

Since we are integrating over a symmetric domain, the integral of an odd function will be zero. This leaves us to consider the even cases.

Backtracking a little bit, the results computed by *Mathematica* should hint that the moments of the distribution can be represented by Riemann zeta functions. In fact, there is a result which states that we can write these integrals in terms of zeta functions:

$$\int_0^\infty \frac{x^n e^x}{(e^x+1)^2} dx = n!(1-2^{1-n})\zeta(n) \tag{7.37}$$

In the current example the nth moment (n even) of the derivative of the Fermi distribution is given by:

$$\int_0^\infty x^n \frac{df}{dx}(dx) = -2n!(1-2^{1-n})\zeta(n) \tag{7.38}$$

Let's verify this with *Mathematica*.

```
(* define a function to compute the moments *)
moments[n_Integer] := 0 /; OddQ[n]
moments[n_Integer] := -2 n ! (1 - 2^(1-n)) Zeta[n] /; EvenQ[n]
```

Now for the comparison:

```
(* compare the answers *)
(* compute moments both ways and compare them with == *)
compare =
   Table[{a, b} = {∫_-∞^∞ x^j h[x] dx, moments[j]}; {a, b, a == b}, {j, 10}];
 (* print a table *)
 TableForm[Prepend[compare, {"integral", "formula", "equivalent?"}]]
```

integral	formula	equivalent?
0	0	True
$-\frac{\pi^2}{3}$	$-\frac{\pi^2}{3}$	True
0	0	True
$-\frac{7\pi^4}{15}$	$-\frac{7\pi^4}{15}$	True
0	0	True
$-\frac{31\pi^6}{21}$	$-\frac{31\pi^6}{21}$	True
0	0	True
$-\frac{127\pi^8}{15}$	$-\frac{127\pi^8}{15}$	True
0	0	True
$-\frac{2555\pi^{10}}{33}$	$-\frac{2555\pi^{10}}{33}$	True

7.4 Working with Legendre and other polynomials

Consider Laplace's equation which gives the potential V for a charge free region of space. In spherical coordinates, assuming azimuthal symmetry, V then satisfies:

$$\frac{\partial}{\partial r}\left(r^2\frac{\partial V}{\partial r}\right)+\frac{1}{\sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta\frac{\partial V}{\partial\theta}\right)=0 \tag{7.39}$$

This equation can be solved by separation of variables, assuming that $V(r,\ \theta)=f(r)g(\theta)$. Without exploring the details here, which can be found in any text on partial differential equations, we find that the equations f and g must satisfy the following, with each equation in $(r,\ \theta)$ separately equal to a constant that we call k:

$$\frac{1}{f}\frac{d}{dr}\left(r^2\frac{df}{dr}\right)=k \tag{7.40}$$

$$\frac{1}{g\sin\theta}\frac{d}{d\theta}\left(\sin\theta\frac{dg}{d\theta}\right)=-k \tag{7.41}$$

It is the second of these equations, the equation in θ, that is of interest here. Solutions of this equation are given in terms of the Legendre polynomials. If we let $k=m(m+1)$, then the solutions are:

$$g(\theta) = P_m(\cos\theta) \tag{7.42}$$

where P_m is the Legendre polynomial. In *Mathematica*, the Legendre polynomials are implemented by the function `LegendreP[n,x]`. Legendre polynomials are indexed from $n = 0$. Here we construct a table of the first ten Legendre polynomials:

```
(* the first few Legendre formulas *)
Table[LegendreP[n, x], {n, 0, 9}] // TableForm
```

$$\begin{array}{l}
1 \\
x \\
-\frac{1}{2} + \frac{3x^2}{2} \\
-\frac{3x}{2} + \frac{5x^3}{2} \\
\frac{3}{8} - \frac{15x^2}{4} + \frac{35x^4}{8} \\
\frac{15x}{8} - \frac{35x^3}{4} + \frac{63x^5}{8} \\
-\frac{5}{16} + \frac{105x^2}{16} - \frac{315x^4}{16} + \frac{231x^6}{16} \\
-\frac{35x}{16} + \frac{315x^3}{16} - \frac{693x^5}{16} + \frac{429x^7}{16} \\
\frac{35}{128} - \frac{315x^2}{32} + \frac{3465x^4}{64} - \frac{3003x^6}{32} + \frac{6435x^8}{128} \\
\frac{315x}{128} - \frac{1155x^3}{32} + \frac{9009x^5}{64} - \frac{6435x^7}{32} + \frac{12155x^9}{128}
\end{array}$$

The graphs are interesting. We begin by separating the parity states.

```
(* separate the Legendre polynomials according to parity *)
eL = Table[LegendreP[n, x], {n, 0, 8, 2}];
oL = Table[LegendreP[n, x], {n, 1, 9, 2}];
```

As before, we will use shading differences to distinguish the polynomials. We will build a list named `shades` of `GrayLevel` settings for this purpose.

```
(* shade the terms: create GrayLevel values  *)
shades = Table[2 (i - 1) / 10, {i, 5}]
```

$$\left\{0, \frac{1}{5}, \frac{2}{5}, \frac{3}{5}, \frac{4}{5}\right\}$$

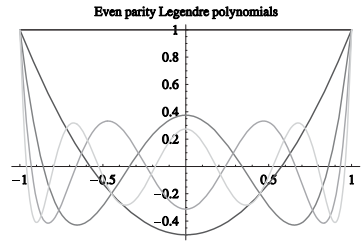
Let's see how this works before we bury it in our plot command.

```
(* test run *)
GrayLevel /@ shades
```

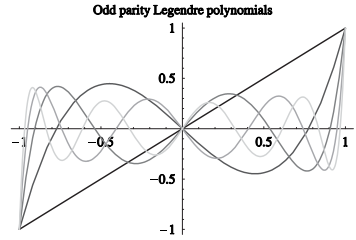
$$\left\{\text{GrayLevel}[0], \text{GrayLevel}\left[\frac{1}{5}\right], \text{GrayLevel}\left[\frac{2}{5}\right], \text{GrayLevel}\left[\frac{3}{5}\right], \text{GrayLevel}\left[\frac{4}{5}\right]\right\}$$

Now we are ready for the plots.

```
(* plot the even parity terms *)
plbl = "Even parity Legendre polynomials";
ge = Plot[Evaluate[eL], {x, -1, 1},
  PlotLabel → plbl, PlotStyle → GrayLevel /@ shades];
```




```
(* plot the odd parity terms *)
plbl = "Odd parity Legendre polynomials";
go = Plot[Evaluate[oL], {x, -1, 1},
  PlotLabel → plbl, PlotStyle → GrayLevel /@ shades];
```



Returning to the original problem, in the case of electrostatics, a general solution of the potential V is found to be:

$$V(r,\ \theta) = \sum_{n=0}^{\infty}\left(A_n r^n + \frac{B_n}{r^{n+1}}\right)P_m(\cos\theta) \tag{7.43}$$

Here A_n and B_n are constants that are determined from the boundary conditions of the problem. We now consider an example.

Example: A hollow sphere

The potential on the surface of a hollow sphere of radius R = 0.4 m is found to be $\cos\left(\frac{\theta}{2}\right)^3$. What is the potential inside the sphere?

Examining the general series solution above, we note that as $r \to 0$, if the constants B_n are not zero, the potential will go to infinity. This is a nonphysical result: we require that the potential remain finite. Therefore, we can immediately set $B_n = 0$ for all n. We are then left with:

$$V(r,\ \theta) = \sum_{n=0}^{\infty} A_n r^n P_m(\cos\theta) \tag{7.44}$$

At $r = R$ (0.4 m), we have the condition that

$$V = \cos\left(\frac{\theta}{2}\right)^3. \tag{7.45}$$

Therefore, we know that

$$\sum_{n=0}^{\infty} A_n r^n P_m(\cos\theta) = \cos\left(\frac{\theta}{2}\right)^3 \tag{7.46}$$

To solve for the A_n, we use the fact that the Legendre polynomials form a complete set of orthogonal functions. We can show this by integrating them in *Mathematica*:

```
(* check the orthogonality relationship *)
ortho[n_Integer, m_Integer] :=
  ∫_0^π LegendreP[n, Cos[θ]] LegendreP[m, Cos[θ]] Sin[θ] dθ
```

```
(* tbest visualized in a matrix *)
mat = Table[ortho[i, j], {i, 0, 3}, {j, 0, 3}];
mat // MatrixForm
```

$$\begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & \frac{2}{3} & 0 & 0 \\ 0 & 0 & \frac{2}{5} & 0 \\ 0 & 0 & 0 & \frac{2}{7} \end{pmatrix}$$

Given the importance of the concept of orthogonality in special functions, we would be wise to ponder it a bit more. Let's look at another one to evaluate the orthogonality relationship.

```
(* compute a few orthogonality integrals *)
results = Table[{i, j, ortho[i, j]}, {i, 0, 3}, {j, 0, 3}]
```

$$\left\{\{\{0, 0, 2\}, \{0, 1, 0\}, \{0, 2, 0\}, \{0, 3, 0\}\},\right.$$
$$\left.\{\{1, 0, 0\}, \left\{1, 1, \frac{2}{3}\right\}, \{1, 2, 0\}, \{1, 3, 0\}\right\},$$
$$\left\{\{2, 0, 0\}, \{2, 1, 0\}, \left\{2, 2, \frac{2}{5}\right\}, \{2, 3, 0\}\right\},$$
$$\left.\left\{\{3, 0, 0\}, \{3, 1, 0\}, \{3, 2, 0\}, \left\{3, 3, \frac{2}{7}\right\}\right\}\right\}$$

If you encountered such a list, how would you determine whether there was an orthogonal function involved? First, we `Flatten` the list to remove unneeded substructure.

```
fresults = Flatten[results, 1]
```

$$\left\{\{0, 0, 2\}, \{0, 1, 0\}, \{0, 2, 0\}, \{0, 3, 0\}, \{1, 0, 0\},\right.$$
$$\left\{1, 1, \frac{2}{3}\right\}, \{1, 2, 0\}, \{1, 3, 0\}, \{2, 0, 0\}, \{2, 1, 0\},$$
$$\left.\left\{2, 2, \frac{2}{5}\right\}, \{2, 3, 0\}, \{3, 0, 0\}, \{3, 1, 0\}, \{3, 2, 0\}, \left\{3, 3, \frac{2}{7}\right\}\right\}$$

We will compose an inspector function to go through the list and tell us whether the datum supports orthogonality ("good") or does not support orthogonality ("bad").

```
(* inspector function *)
fcn = ({l, r, i} = #;
    If[l == r, If[i ≠ 0, "good", "bad"], If[i ≠ 0, "bad", "good"]]) &;
```

When we apply this function to `fresults` we see:

```
(* results *)
fcn /@ fresults

{good, good, good, good, good, good, good,
 good, good, good, good, good, good, good, good, good}
```

Returning to the computation, we find that in general, the integral

$$\int_0^{\pi} P_m(\cos\theta) P_n(\cos\theta) d\theta = \frac{2}{2m+1}\delta_{mn} \tag{7.47}$$

Using this property we can solve for each of the A_n constants by using the orthogonality relationship to project out individual components. In other words, we can solve for the constants by multiplying each side of the equation above by $P_m(\cos\theta)\sin\theta$ and integrating. This allows us to specify each constant as:

$$A_n = \frac{2n+1}{2R^n}\int_0^{\pi} \cos\left(\frac{\theta}{2}\right)^3 P_m(\cos\theta)\sin\theta d\theta \tag{7.48}$$

Mathematica computes the first constant as about 1.2:

```
(* find the constant A *)
R = 0.4;
μ = ∫_0^π LegendreP[1, Cos[θ]] LegendreP[1, Cos[θ]] Sin[θ] dθ;
A1 = 1/μ 1/R ∫_0^π Cos[x/2]^3 LegendreP[1, Cos[x]] Sin[x] dx

1.28571
```

To make this more general, we can define a function to calculate these terms. For example, let's define μ as a function of *n* that will let us calculate the inner product of any order of Legendre polynomials, where *n* = *m*.

```
(* generalize the calculation *)
Clear[μ];
μ[n_Integer] := ∫_0^π LegendreP[n, Cos[θ]] LegendreP[n, Cos[θ]] Sin[θ] dθ
```

Now let's test it by computing a few values:

```
(* check functioning *)
Table[Print["n = ", n, ", μ = ", μ[n]], {n, 0, 5}];
```

n = 0, μ = 2
n = 1, μ = $\frac{2}{3}$
n = 2, μ = $\frac{2}{5}$
n = 3, μ = $\frac{2}{7}$
n = 4, μ = $\frac{2}{9}$
n = 5, μ = $\frac{2}{11}$

Now we can define a function `A[n,R]` that will calculate the coefficient for any values of *n* and *R*:

```
(* general function for A *)
A[n_Integer, R_] :=
  1/μ[n] 1/R ∫_0^π Cos[x/2]^3 LegendreP[1, Cos[x]] Sin[x] dx;
```

Let’s check some sample cases.

```
(* sample cases *)
A[1, 0.4]
```

```
1.28571
```

```
A[2, 0.4]
```

```
2.14286
```

With this tool in hand, we can form a series to represent the potential.

```
(* use the first 6 terms to form the potential *)
R = 0.4; m = 5;
```

```
(* potential function *)
V[r_, θ_] := Sum[A[n, R] r^n LegendreP[n, Cos[θ]], {n, 0, m}]
```

This evaluates to:

```
(* show current value *)
V[r, θ]
```

$$0.428571 + 1.28571\, r \cos[\theta] + 2.14286\, r^2 \left(-\frac{1}{2} + \frac{3\cos[\theta]^2}{2}\right) + 3.\, r^3 \left(-\frac{3\cos[\theta]}{2} + \frac{5\cos[\theta]^3}{2}\right) + 3.85714\, r^4 \left(\frac{3}{8} - \frac{15\cos[\theta]^2}{4} + \frac{35\cos[\theta]^4}{8}\right) + 4.71429\, r^5 \left(\frac{15\cos[\theta]}{8} - \frac{35\cos[\theta]^3}{4} + \frac{63\cos[\theta]^5}{8}\right) + 5.57143\, r^6 \left(-\frac{5}{16} + \frac{105\cos[\theta]^2}{16} - \frac{315\cos[\theta]^4}{16} + \frac{231\cos[\theta]^6}{16}\right)$$

7.5 Spherical harmonics

Spherical harmonics are functions of the angular variables (θ, ϕ) that are related to the Legendre polynomials. In *Mathematica*, we can calculate the spherical harmonics by calling on the `SphericalHarmonicY[l, m, θ, ϕ]` function, which gives $Y_{lm}(\theta, \phi)$. For example, here are few functions:

```
(* more spherical harmonics *)
SphericalHarmonicY[0, 0, θ, ϕ]
SphericalHarmonicY[1, 0, θ, ϕ]
SphericalHarmonicY[3, -2, θ, ϕ]
```

$$\frac{1}{2\sqrt{\pi}}$$

$$\frac{1}{2}\sqrt{\frac{3}{\pi}}\,\text{Cos}[\theta]$$

$$\frac{1}{4}\,e^{-2i\phi}\sqrt{\frac{105}{2\pi}}\,\text{Cos}[\theta]\,\text{Sin}[\theta]^2$$

The spherical harmonics are often represented in Cartesian coordinates. This can be done by noting the relationship between Cartesian and spherical coordinates, namely

Using $r = \sqrt{x^2 + y^2}$, we can define the following functions in *Mathematica* to help us write the spherical harmonics in cartesian coordinates:

One application of the spherical harmonics is in the description of angular momentum in quantum mechanics. There exist operators, known as the raising and lowering operators, that can change the value of *m* by 1. We won't worry about how such operators are derived in quantum mechanics but simply describe how they could be implemented simply in *Mathematica*. Here we define two functions, LPlus and LMinus, where LPlus takes $Y_{l,m} \to Y_{l,m+1}$ and LMinus takes $Y_{l,m} \to Y_{l,m-1}$ modulo a constant. Here h is Planck's constant:

Exercise

Modify the LPlus and LMinus operators so that LPlus $Y_{l,l} = 0$ and LMinus $Y_{l,-l} = 0$. Now consider the operator L^2 such that $L^2 Y_{l,m} = h^2 l\,(l+1)$, and L_z such that $L_z Y_{l,m} = mh Y_{l,m}$. If

$L^2 = (L_x)^2 + (L_y)^2 + (L_z)^2$, and LPlus = L_x + äL_y, LMinus = L_x - äL_y, construct functions in *Mathematica* to implement L^2 and L_z, where L^2 is constructed out of LPlus, LMinus, and L_z.

A P P E N D I X 1

A1.1 Pictionary of 2D graphic types

Mathematica is an absolute treasure chest of graphics tools. The breadth and depth is significant. We do, however, have one minor issue with the presentation since we have on several occasions been searching vigorously through the Help Browser and *The Mathematica Book* looking for a specific format. It would be helpful to collect all the plot types and locate them together. That is exactly what we have done here. We have brought together all of the 2D and 3D graphics types to allow you to quickly find the format you are looking for or to simply familiarize yourself with the tools of *Mathematica*.

We will show the plots first and following those will be the specific commands to produce each image. We have tried to present the many different ways you may employ the graphics package and so similar plots may have very different command syntax.

This section was designed as a reference, so we have tried to make each plot description self-contained. This leads to a lot of duplication; you will see several definitions of the sinc function. We have also tried to be generous with our *Mathematica* comments so that more advanced users could just look at the code fragments to understand the process.

One point we must stress is that a package only needs to be loaded once and it must be loaded before you call the desired commands. Do not duplicate the package loads. We show a load with every fragment in order to make the description self-contained. We don't want users thinking they need to read previous entries in order to understand a plot.

We want to stress the flexibility of *Mathematica* and so many times we will perform similar processes using different methods. Don't be confused by this: simply choose the method you are most comfortable with and use that.

The only time that we need data from a previous plot comes when we look at the raster arrays starting with plot 35.. Because the data generation was so long, we just refer to the results in plots 37., 38., and 39..

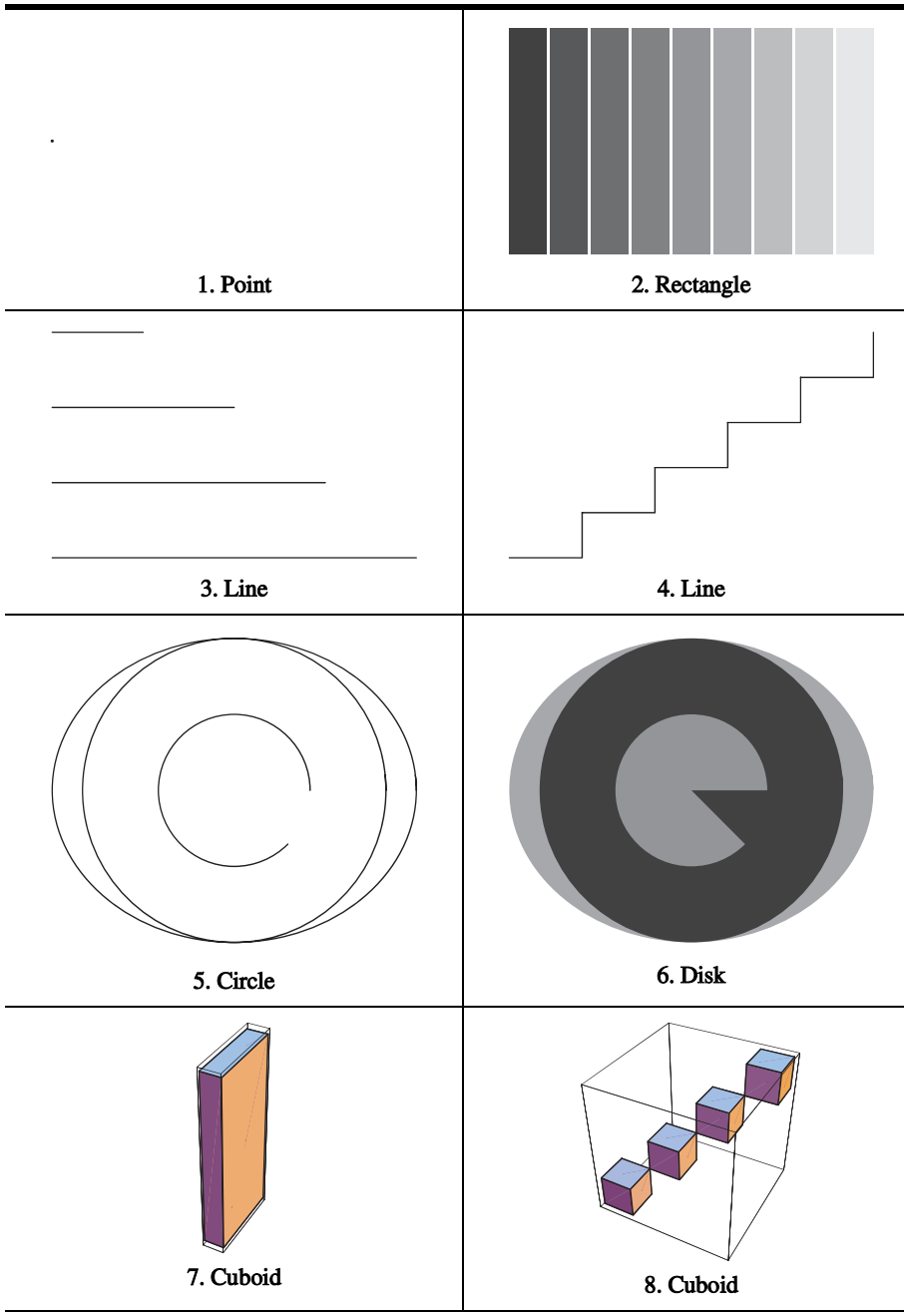
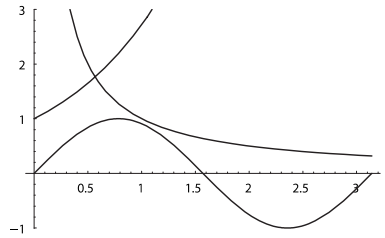
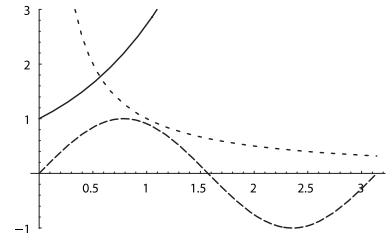


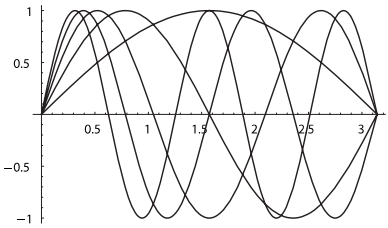
Table A1.1: Graphics primitives. The commands give the user a wide range of control. The `Raster` graphics primitive will be presented with the `DensityPlot` commands. Note the `Cuboid` commands create Graphics3D objects.



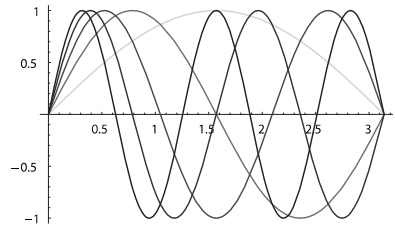
9. Plot



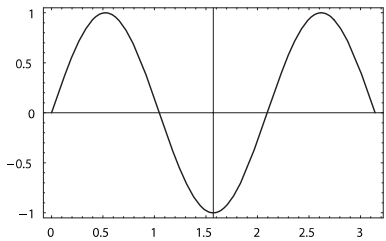
10. Plot



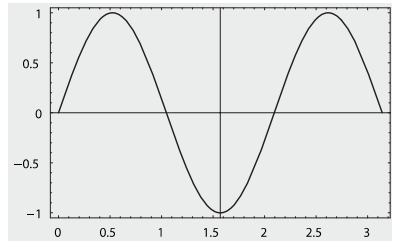
11. Plot



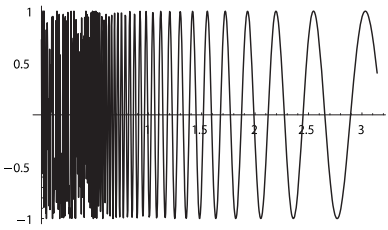
12. Plot



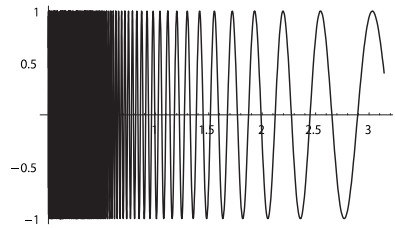
13. Plot



14. Plot

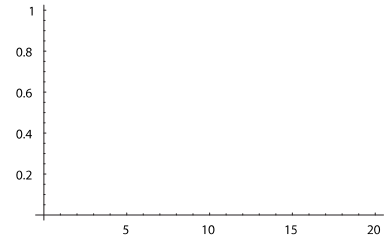


15. Plot

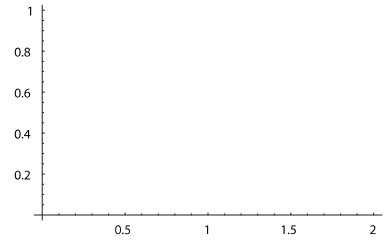


16. Plot

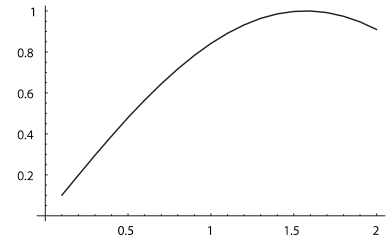
Table A1.2: Exploring the basics of `Plot`. The top four graphs show different ways to plot multiple curves. The bottom four graphs display different strategies to refine the appearance of a plot.



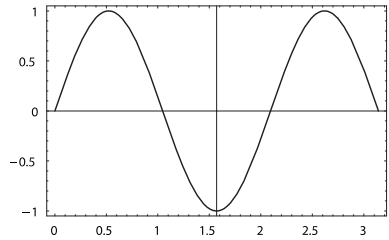
17. ListPlot



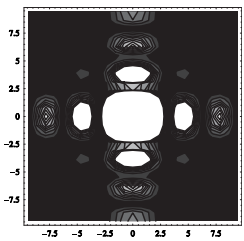
18. ListPlot



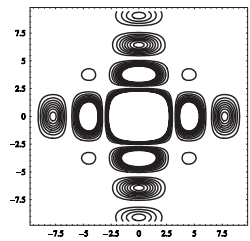
19. ListPlot



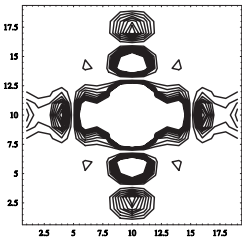
20. ListPlot



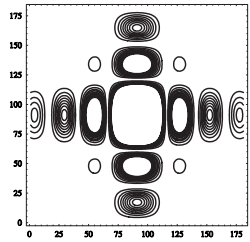
21. ContourPlot



22. ContourPlot

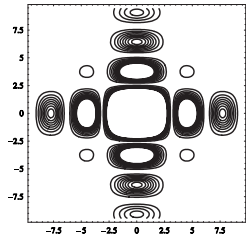


23. ListContourPlot

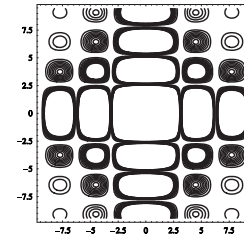


24. ListContourPlot

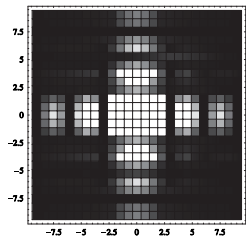
Table A1.3: Looking at `ListPlot`, `ContourPlot` and `ListContourPlot`. The top four figures all show `ListPlot`s. Plot 20. shows a `Plot` overlaid on a `ListPlot`. There are two formats for the points in `ListPlot`. Figures 17. and 18. show the differences; look at the *x* axes. The bottom four graphics compare `ContourPlot` to `ListContourPlot`. Note the lower figures are transposed.



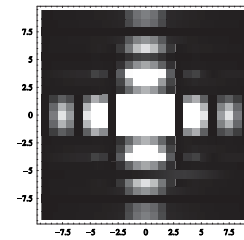
25. ContourPlot



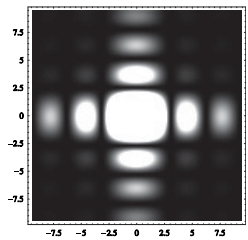
26. ContourPlot



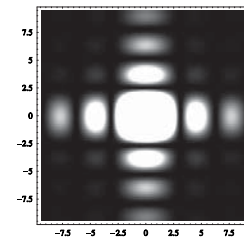
27. DensityPlot



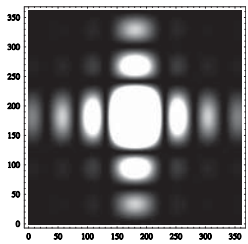
28. DensityPlot



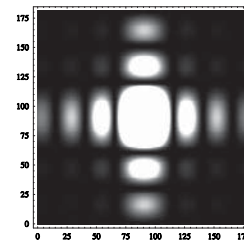
29. DensityPlot



30. DensityPlot



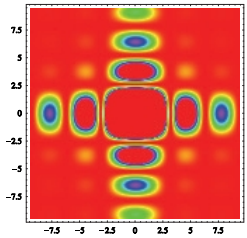
31. ListDensityPlot



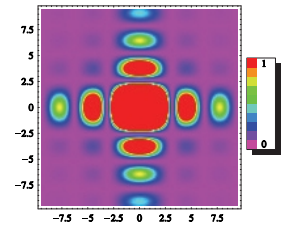
32. ListDensityPlot

Table A1.4: `ContourPlot`, `DensityPlot` and `ListDensityPlot`. Very often these two-dimensional representations are more informative than a three-dimensional view. Notice the transposition of the `ListDensityPlot`s. Compare these images with the `Raster` plots on the next page.

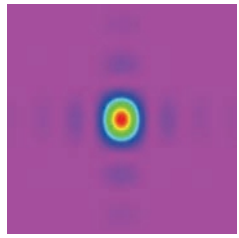
rq



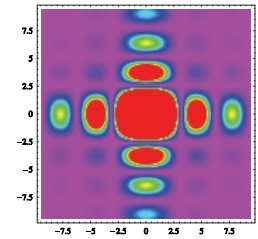
33. DensityPlot



34. DensityPlot with Legend



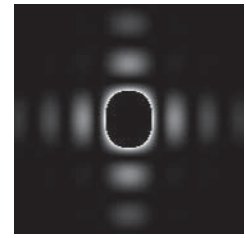
35. Graphics[Raster[ ]]



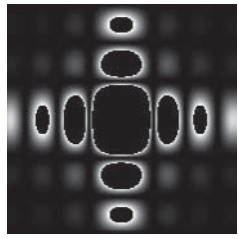
36. DensityPlot



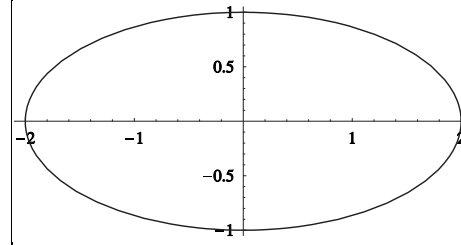
37. Raster



38. Raster

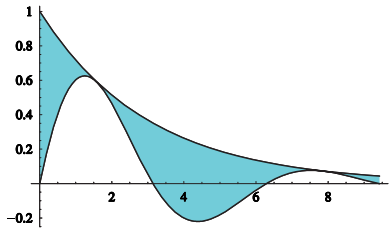


39. Raster

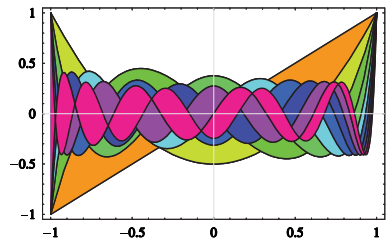


40. ParametricPlot

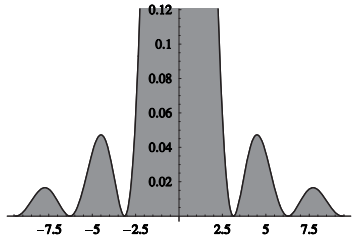
Table A1.5: `DensityPlot`s with color, `Raster` plots and a `ParametricPlot`. Plots 35. and 36. share the same color map. We use a shading function to block out the high features of the `Raster` graphics allowing the small features to bloom. We close with a `ParametricPlot`.



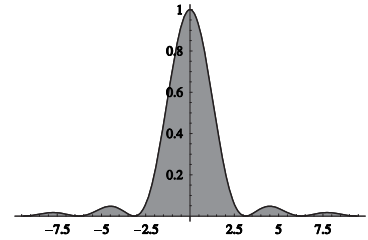
41. FilledPlot



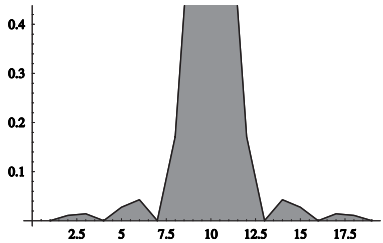
42. FilledPlot



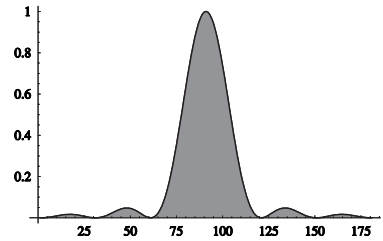
43. FilledPlot



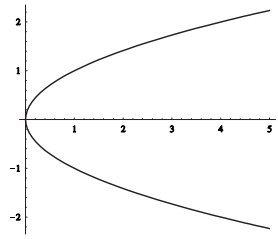
44. FilledPlot



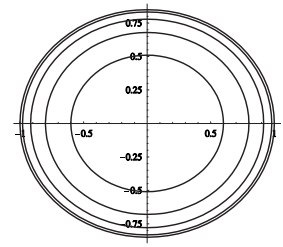
45. ListFilledPlot



46. ListFilledPlot

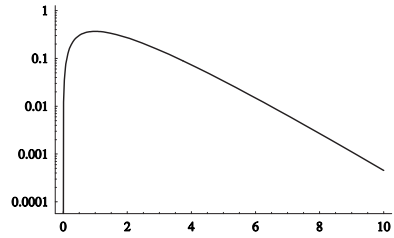


47. ImplicitPlot

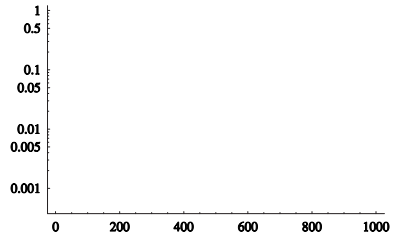


48. ImplicitPlot

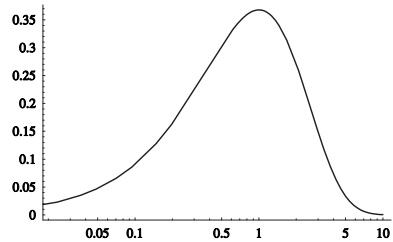
Table A1.6: Exploring the `Graphics` add-on package. We see here examples from the `Graphics`‘`FilledPlot`‘and `Graphics`‘`ImplicitPlot`‘ packages.



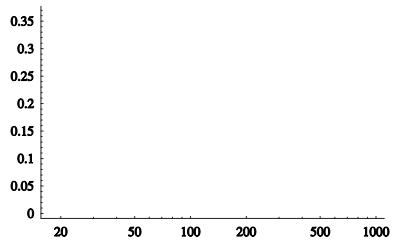
49. LogPlot



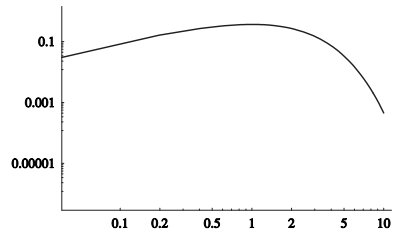
50. LogListPlot



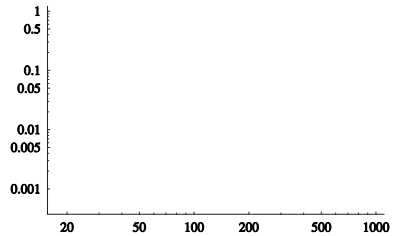
51. LogLinearPlot



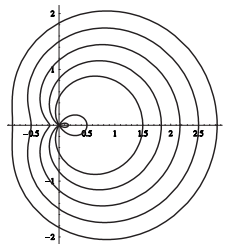
52. LogLinearListPlot



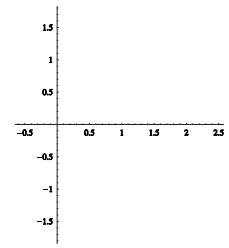
53. LogLogPlot



54. LogLogListPlot

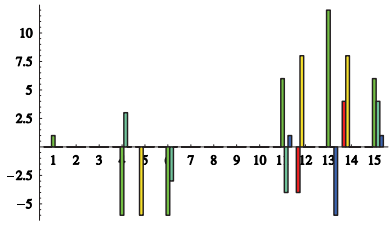


55. PolarPlot

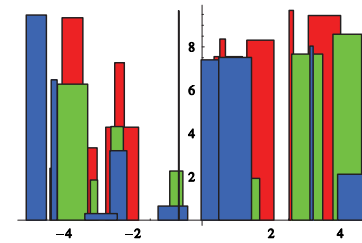


56. PolarListPlot

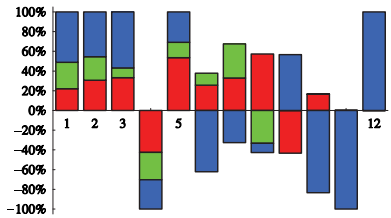
Table A1.7: A survey of the `Graphics`'`Graphics` tools.



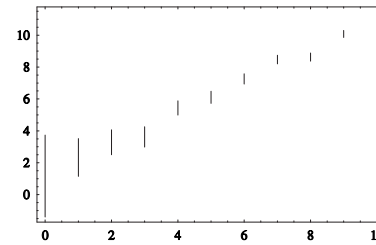
57. Graphics‘BarChart



58. GeneralizedBarChart



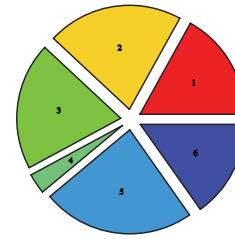
59. PercentileBarChart



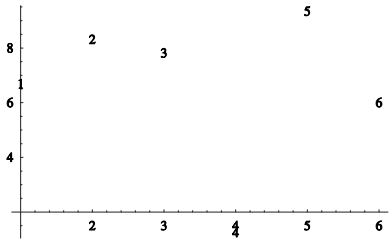
60. ErrorListPlot



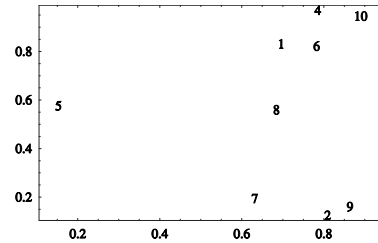
61. Graphics‘PieChart



62. Graphics‘PieChart

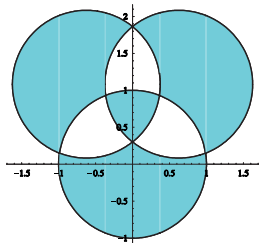


63. Graphics‘TextListPlot

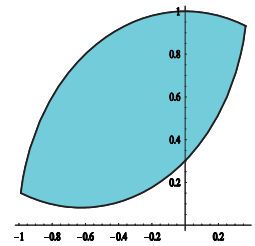


64. Graphics‘LabeledListPlot

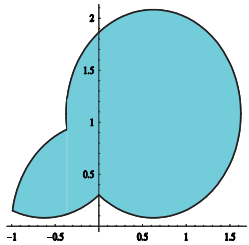
Table A1.8: More from the `Graphics` package. We see here examples from the `Graphics‘ InequalityPlot‘`, `Graphics‘ InequalityPlot`, and `Graphics‘MultipleListPlot‘` packages.



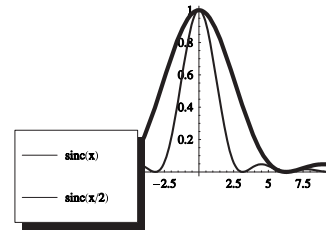
65. InequalityPlot



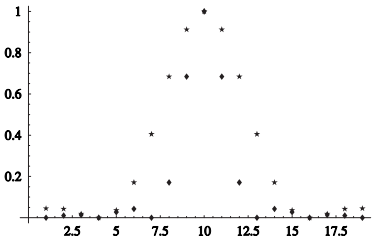
66. InequalityPlot



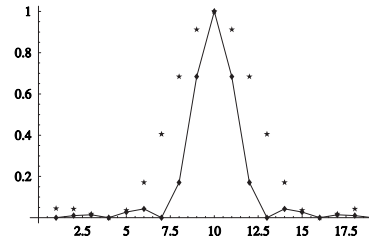
67. InequalityPlot



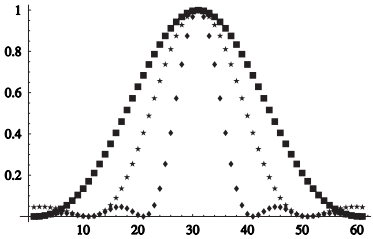
68. Plot with Legend



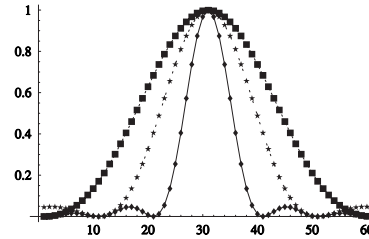
69. MultipleListPlot



70. MultipleListPlot

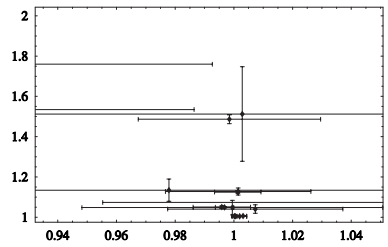


71. MultipleListPlot

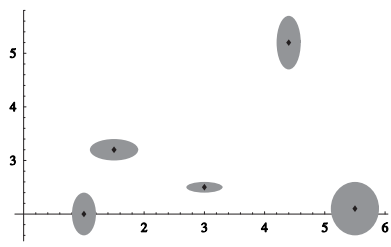


72. MultipleListPlot

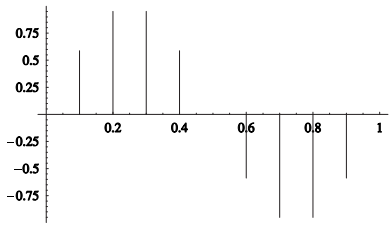
Table A1.9: More from the `Graphics` package. We see here examples from the `Graphics' InequalityPlot`, `Graphics' InequalityPlot`, and `Graphics`MultipleListPlot` packages.



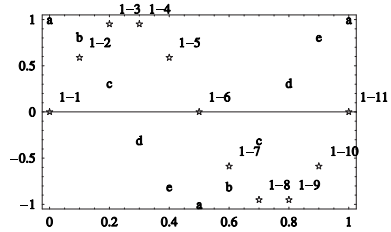
73. MultipleListPlot



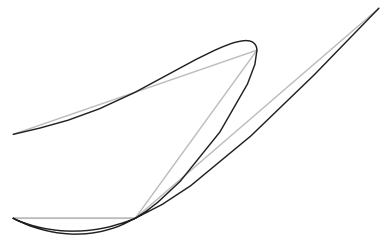
74. MultipleListPlot



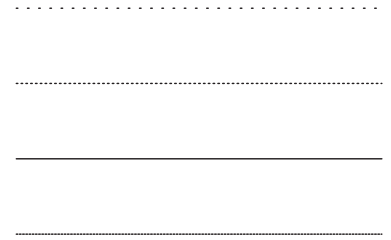
75. MultipleListPlot



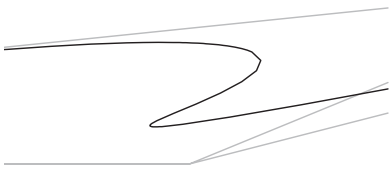
76. MultipleListPlot



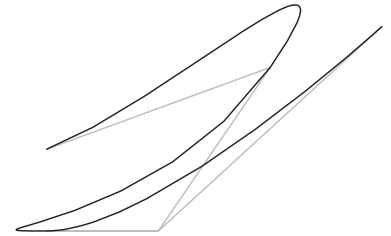
77. Graphics[Spline[ ]]



78. Examples of dashing

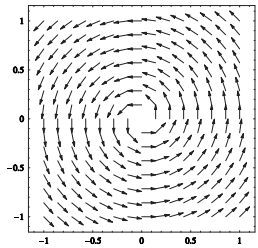


79. Graphics[Spline[ ]]

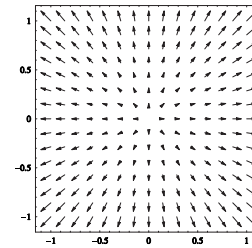


80. Graphics[Spline[ ]]

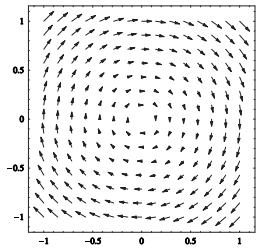
Table A1.10: Still more plots from the `Graphics` package. We see here more examples from `Graphics`MultipleListPlot`` packages. We end our look at the `Graphics`` package with examples from `Spline` which is used like a graphics primitive.



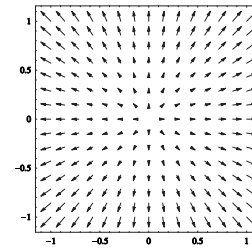
81. PlotField`PlotVectorField



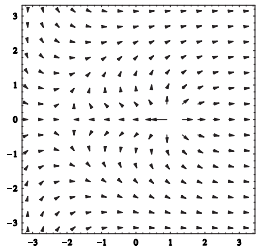
82. PlotField`PlotVectorField



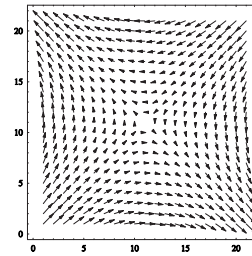
83. PlotField`PlotHamiltonianField



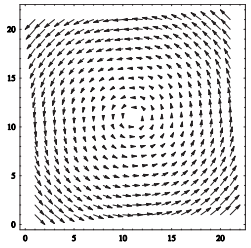
84. PlotField`PlotGradientField



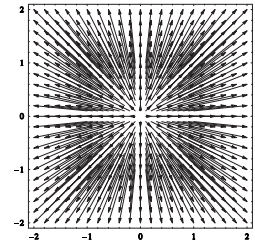
85. PlotField`PlotPolyaField



86. PlotField`ListPlotVectorField

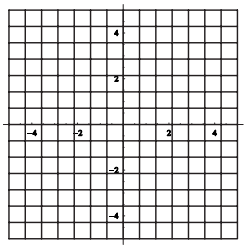


87. PlotField`ListPlotVectorField

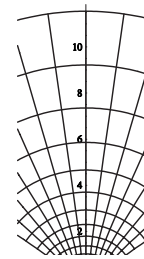


88. PlotField`ListPlotVectorField

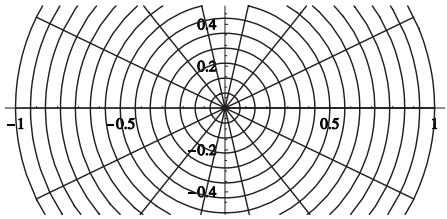
Table A1.11: Final 2D plots from the `Graphics` package. The `PlotField` package can produce some very interesting plots of vector fields. This package quickly reveals the relationship between conservative potentials and their forces.



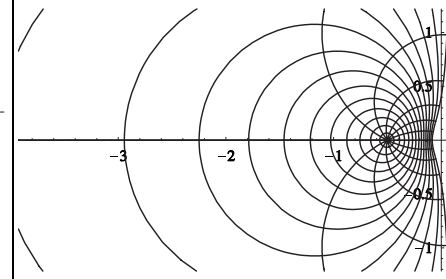
89. ComplexMap‘CartesianMap



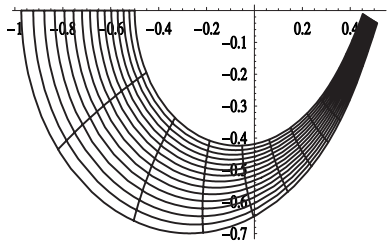
90. ComplexMap‘CartesianMap



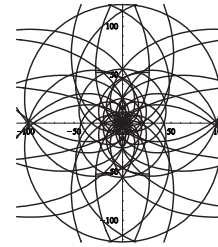
91. ComplexMap‘PolarMap



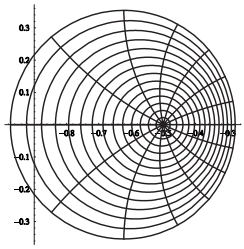
92. ComplexMap‘PolarMap



93. ComplexMap‘CartesianMap



94. ComplexMap‘PolarMap



95. ComplexMap‘PolarMap



96. ComplexMap‘CartesianMap

Table A1.12: Final 2D plots from the `Graphics` package. The `PlotField` package can produce some very interesting plots of vector fields. This package quickly reveals the relationship between conservative potentials and their forces.

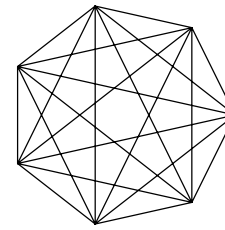
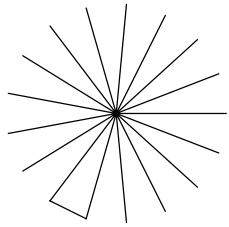
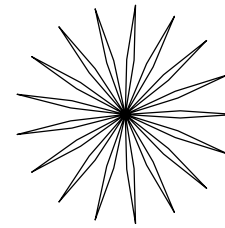
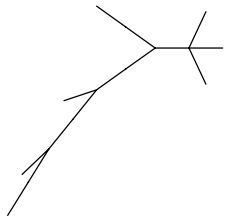
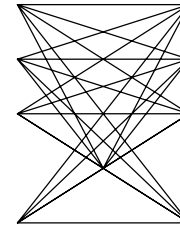
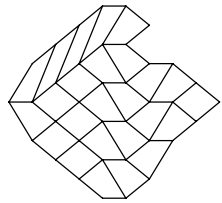
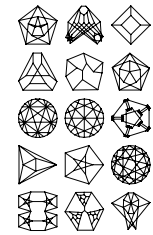
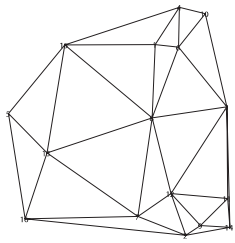
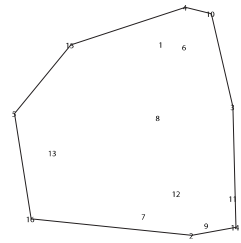
	
97. Combinatorica`FerrersDiagram	98. Combinatorica`ShowGraph
	
99. Combinatorica`ShowGraph	100. Combinatorica`ShowGraph
	
101. Combinatorica`ShowGraph	102. Combinatorica`ShowGraph
	
103. Combinatorica`ShowGraph	104. Combinatorica`ShowGraphArray

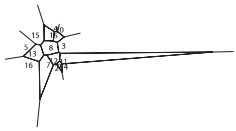
Table A1.13: Samplings from `DiscreteMath`Combinatorica`.



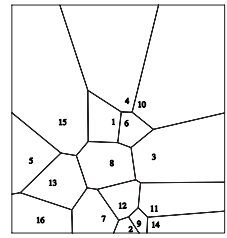
105. ComputationalGeometry‘PlanarGraphPlot



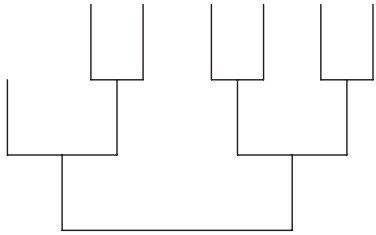
106. ComputationalGeometry‘PlanarGraphPlot



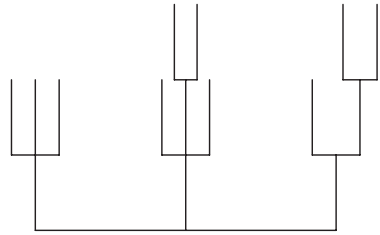
107. ComputationalGeometry‘DiagramPlot



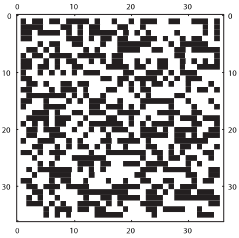
108. ComputationalGeometry‘TriangularSurfacePlot



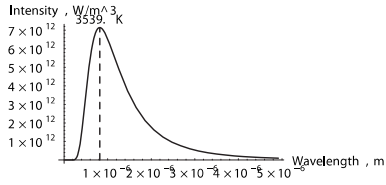
109. DiscreteMath‘TreePlot



110. DiscreteMath‘ExprPlot

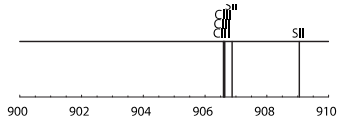


111. MatrixManipulation‘MatrixPlot

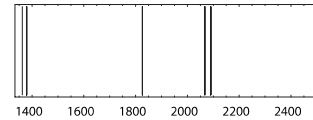


112. BlackBodyRadiation‘BlackBodyProfile

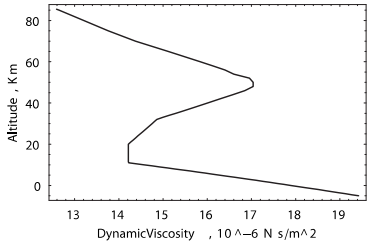
Table A1.14: Samplings from `DiscreteMath‘ComputationalGeometry`, `DiscreteMath‘Tree`.



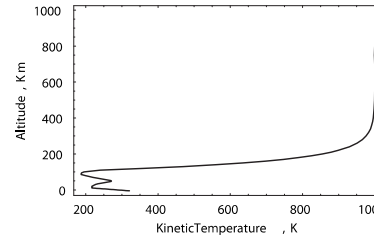
113. ResonanceAbsoprtionLines'WavelengthAbsorptionMap



114. ResonanceAbsoprtionLines'ElementAbsoprtionMap



115. StandardAtmosphere'AtmosphericPlot



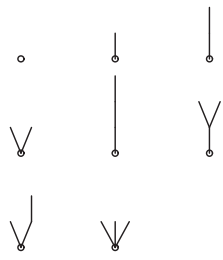
116. StandardAtmosphere'AtmosphericPlot



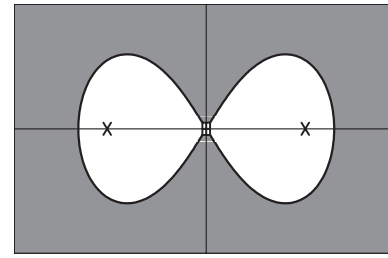
117. WorldPlot'WorldPlot



118. WorldPlot'WorldPlot



119. Butcher'ButcherPlot

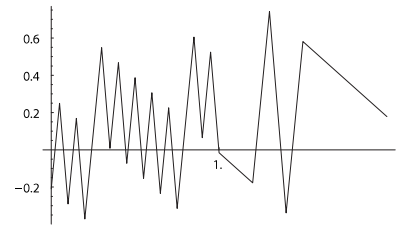


120. OrderStar'OrderStar

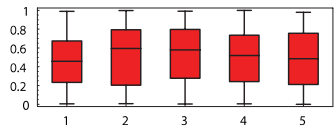
Table A1.15: Samplings from `DiscreteMath'ComputationalGeometry`, `DiscreteMath'Tree`.



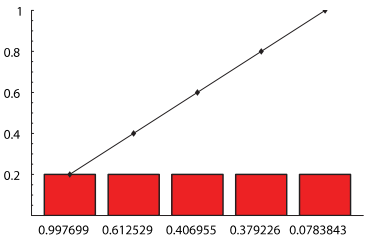
121. Microscope‘Microscope



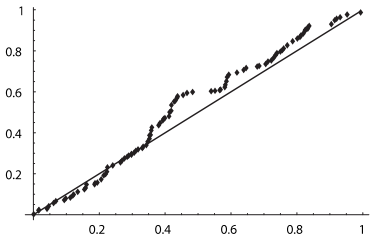
122. Microscope‘MicroscopicError



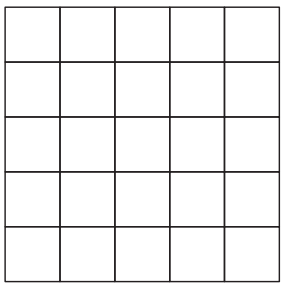
123. StatisticsPlots‘BoxWhiskerPlot



124. StatisticsPlots‘BoxWhiskerPlot



125. StatisticsPlots‘QuantilePlot



126. StatisticsPlots‘PairwiseScatterPlot

Table A1.16: Samplings from `DiscreteMath‘ComputationalGeometry`, `DiscreteMath‘Tree`

A1.2 Graphics primitives

We begin our survey with the graphics primitives. This is a good laboratory to start with. The commands are very simple and they introduce the user to basic concepts like the coordinate system and dimension of the plot. Also we learn about shading, coloring and concepts like aspect ratio.

Plot 1.. We can easily plot a list of points. First, we create a list of points by sampling a function.

```
(* create a list of points *)
pts = Table[{x, Sin[x]}, {x, 0, 2 π, π/12}];
```

Next, we create a list of graphics primitives by applying `Point` across the list of points.

```
(* create a graphics primitive *)
prim = Point /@ pts;
```

You can see what the list is like by examining the first element.

```
(* you can see the format by examining the first element *)
First[prim]

Point[{0, 0}]
```

We need to convert these commands into a `Graphics` object and display it with the `Show` command.

```
(* convert primitive into a graphics object and display *)
g1 = Show[Graphics[prim]];
```

Plot 2.. Notice how we avoided a white rectangle (`GrayLevel[1]`) since it would be invisible.

```
(* spacing between blocks: 10 % of unit block width *)
δ = 0.1;
(* number of blocks *)
nr = 9;
(* create a table of graphics primitives *)
recs = Table[Rectangle[l = i (1 + δ); r = l + 1; {l, 0}, {r, 1}], {i, 0, nr}];
(* generate shading options *)
shades = Table[{GrayLevel[i/10], recs[[i]]}, {i, nr}];
(* convert primitive into a graphics object and display *)
g1 = Show[Graphics /@ shades];
```

Plot 3.. We can draw disconnected lines…

```
(* create a list of end points *)
pts = Table[{{0, i}, {5 - i, i}}, {i, 4}];
(* create a graphics primitive *)
prim = Line /@ pts;
(* convert primitive into a graphics object and display *)
g1 = Show[Graphics[prim]];
```

Plot 4.. Or connected lines.

```
(* offset starting point *)
p = {-1, -1};
(* generate a table of stair steps *)
pts = Table[p += {1, 1}; {p, p + {1, 0}, p + {1, 1}}, {i, 0, 4}];
(* linearize list *)
Flatten[pts, 1];
(* create a graphics primitive *)
prim = Line /@ pts;
(* convert primitive into a graphics object and display *)
g1 = Show[Graphics[prim]];
```

Plot 5.. There is a command for circles…

```
(* unit radius, centered on origin *)
cir1 = Circle[{0, 0}, 1];
(* unit radius, centered on origin, partial arc *)
cir2 = Circle[{0, 0}, 1/2, {0, 7π/4}];
(* ellipse, centered on origin *)
cir3 = Circle[{0, 0}, {1.2, 1}];
(* convert primitives into a graphics object and display *)
g1 = Show[Graphics /@ {cir1, cir2, cir3}, AspectRatio → Automatic];
```

Plot 6.. and a separate one for disks.

```
(* ellipse, centered on origin *)
disk1 = {GrayLevel[0.1], Disk[{0, 0}, 1]};
(* ellipse, centered on origin *)
disk2 = {GrayLevel[0.5], Disk[{0, 0}, 1/2, {0, 7π/4}]};
(* unit radius, centered on origin *)
disk3 = {GrayLevel[0.6], Disk[{0, 0}, {1.2, 1}]};
(* convert primitives into a graphics object and display *)
g1 = Show[Graphics /@ {%, %%%, %%}, AspectRatio → Automatic];
```

Plot 7.. This is the monolith from 2001: A Space Odyssey. The aspect ratios were given by the squares of the first three integers.

```
(* always a good idea to have the origin defined as a variable *)
o = {0, 0, 0};
(* create a graphics primitive *)
prim = Cuboid[o, {1, 4, 9}];
(* convert primitive to 3D graphics object and display *)
g1 = Show[Graphics3D[prim]];
```

Plot 8.. Notice that these are `Graphics3D` object.

```
(* always a good idea to have the origin defined as a variable *)
o = {0, 0, 0};
(* create a graphics primitive *)
cubes = Table[o + i {1, 1, 1}, {i, 4}];
(* convert primitives to 3 D graphics object and display *)
g1 = Show[Graphics3D[Cuboid /@ cubes]];
```

A1.3 Kernel commands

We now will explore the plot commands available to users in the standard kernel. These commands are available in the *Mathematica* kernel and do not require us to load any packages to access them. (The exception is plot 34. where you must load a package to create the plot legend.)

Plot 9.. We begin a look at the work horse command, `Plot`.

```
(* plot three different functions *)
g1 = Plot[{Sin[2 x], Exp[x], 1/x}, {x, 0, π}, PlotRange → {-1.01, 3}];
```

Plot 10.. We can identify the curves by dashing.

```
(* vary dashing to distinguish curves *)
dashes = {{0.15, 0.05}, {1, 0}, {0.05, 0.15}};
(* plot three different curves *)
g1 = Plot[{Sin[2 x], Exp[x], 1/x}, {x, 0, π},
   PlotRange → {-1.01, 3}, PlotStyle → Dashing /@ dashes];
```

Plot 11.. We can plot lists of functions.

```
(* maximum number of functions in the table *)
nmax = 5;
(* create a table of functions *)
f[x_] := Table[Sin[n x], {n, nmax}];
(* notice the use of Evaluate *)
g1 = Plot[Evaluate[f[x]], {x, 0, π}];
```

Plot 12.. We can also identify curves through shading.

```
(* maximum number of functions in the table *)
nmax = 5;
(* create a table of functions *)
f[x_] := Table[Sin[n x], {n, nmax}];
(* vary the shades of the different curves *)
g1 = Plot[Evaluate[f[x]], {x, 0, π},
   PlotStyle → Table[GrayLevel[1/n - 1/nmax], {n, nmax}]];
```

Plot 13.. Here we translate the origin of the plot.

```
(* shift the origin away from {0, 0} *)
g13 = Plot[Sin[3 x], {x, 0, π}, AxesOrigin → {π/2, 0}, Frame → True];
```

Plot 14.. There are two ways to create this plot. The easiest is via `Show`, or we can recreate the plot.

```
(* change the background color without rerendering *)
g2 = Show[g1, Background → GrayLevel[0.92]];
```

```
(* render the plot with a different background color *)
g1 = Plot[Sin[3 x], {x, 0, π}, AxesOrigin → {π/2, 0},
   Frame → True, Background → GrayLevel[0.92]];
```

Plot 15.. Here the default plot sample density struggles with this wildly oscillating function. Notice that the maximum and minimum values near the origin no longer appear to be ±1, which we know they should be.

```
(* default sampling density, 25 *)
g1 = Plot[Sin[100/x], {x, 0, π}];
```

Plot 16.. Boosting the sample density by a factor of 20 fixes the amplitude problem.

```
(* change sample density from 25 to 500 *)
g1 = Plot[Sin[100/x], {x, 0, π}, PlotPoints → 500];
```

Plot 17.. We begin our examination of `ListPlot`. First, we need to create some lists using two different syntaxes. Study them closely. The first format is simply a list of values. The first value will be plotted at position 1, the second at position 2, etc. The *x* axis is now a counter and no longer has physical relevance.

```
(* create two different types of lists *)
(* be careful: they look the same *)
(* list1 is a list of y[x[[i]]] *)
list1 = Table[Sin[i/10], {i, 20}];
(* list2 is a list of {x[[i]], y[x[[i]]]} *)
list2 = Table[{i/10, Sin[i/10]}, {i, 20}];
```

Plot 18.. The second is a list of *x* and *y* values together and the *x* axis now has physical dimension.

```
(* horizontal axis counts the points *)
g1 = ListPlot[list1];
```

Plot 19.. You may wish to connect the points to make a form more distinct.

```
(* this is the previous plor with the dots connected *)
g1 = ListPlot[list2, PlotJoined→ True];
```

Plot 20.. It's easy to combine `Graphics` objects.

```
(* combine plots 5 and 10 *)
g1 = Show[g13, g18];
```

Plot 21.. For the next several plots, we will be using an asymmetric function built from the sampling function, also called the sinc function. We start with the default contour plot.



Sinc Function

```
(* define the sampling function *)
sinc[x_ /; x ≠ 0] := Sin[x] / x;
sinc[x_ /; x == 0] := 1;
```

```
(* default ContourPlot *)
(* the slight asymmetry (y will have more lobes) *)
(* allows us to verify the x (across) and y (up) axes *)
g1 = ContourPlot[sinc[x]^2 sinc[1.2 y]^2, {x, -3 π, 3 π}, {y, -3 π, 3 π}];
```

Plot 22.. Turning off the shading can be done without regenerating the plot.

```
(* there is no need to rerender the graphic *)
g2 = Show[g1, ContourShading→ False];
```

Plot 23.. To explore the list version of this command, `ListContourPlot`, we will need to create a list of points. These points will be evaluated and then contours will be drawn. The first list will be rather coarse, consisting of just 19 points.

```
(* coarse sample size *)
δ = π / 3;
(* create a list of sample points for ListContourPlot *)
pts = Table[sinc[x]^2 sinc[1.2 y]^2, {x, -3 π, 3 π, δ}, {y, -3 π, 3 π, δ}];
(* count the points *)
Length[pts]
```

```
19
```

```
(* ListContourPlot, coarse resolution *)
g1 = ListContourPlot[pts, ContourShading → False];
```

Plot 24.. We increase the resolution substantially, by almost an order of magnitude.

```
(* fine sample size *)
δ = π / 30;
(* create a list of sample points *)
pts =
  Table[sinc[x]^2 sinc[1.2 y]^2, {x, -3 π, 3 π, δ}, {y, -3 π, 3 π, δ}];
(* count the points *)
Length[pts]
```

```
181
```

```
(* ListContourPlot, fine resolution *)
g1 = ListContourPlot[pts, ContourShading → False];
```

Plot 25.. We begin a series of improvements to the basic contour plot. The first being resolution.

```
(* a refined ContourPlot: improved resolution *)
g1 = ContourPlot[sinc[x]^2 sinc[1.2 y]^2, {x, -3 π, 3 π},
   {y, -3 π, 3 π}, ContourShading → False, PlotPoints → 150];
```

Plot 26.. By limiting the `PlotRange`, we can zoom in on interesting features.

```
(* improved resolution, zoom in by limiting PlotRange *)
g1 = ContourPlot[sinc[x]^2 sinc[1.2 y]^2, {x, -3 π, 3 π}, {y, -3 π, 3 π},
   ContourShading → False, PlotRange → {0, 0.001}, PlotPoints → 150];
```

Plot 27.. Compare the `ContourPlot`s with the `DensityPlot`s.

```
(* default DensityPlot *)
g1 = DensityPlot[sinc[x]^2 sinc[1.2 y]^2, {x, -3 π, 3 π}, {y, -3 π, 3 π}];
```

Plot 28.. Turning of the mesh can be very helpful.

```
(* default DensityPlot without the mesh *)
g1 = DensityPlot[sinc[x]^2 sinc[1.2 y]^2,
   {x, -3 π, 3 π}, {y, -3 π, 3 π}, Mesh → False];
```

Plot 29.. Eliminating the mesh is essential at higher plot density.

```
(* refined DensityPlot: turn off the mesh *)
g1 = DensityPlot[sinc[x]^2 sinc[1.2 y]^2,
   {x, -3 π, 3 π}, {y, -3 π, 3 π}, Mesh → False, PlotPoints → 150];
```

Plot 30.. If we increase the resolution enough, we can make the effects of discretization disappear.

```
(* no discritization *)
g1 = DensityPlot[sinc[x]^2 sinc[1.2 y]^2,
   {x, -3 π, 3 π}, {y, -3 π, 3 π}, Mesh → False, PlotPoints → 250];
```

Plot 31.. The next few images we made with the list version, `ListDensityPlot`. This required us to create two lists: a coarse sampling list and a fine sampling list. First, we give the coarse list.

```
(* coarse sample size *)
δ = π / 15;
(* create a list of sample points *)
pts =
  Table[sinc[x]^2 sinc[1.2 y]^2, {x, -3 π, 3 π, δ}, {y, -3 π, 3 π, δ}];
(* count the points *)
Length[pts]
```

```
91
```

```
(* density plot, coarse data *)
g1 = ListDensityPlot[pts, Mesh → False];
```

Plot 32.. Now we give the fine list.

```
(* fine sample size *)
δ = π / 60;
(* create a list of sample points *)
pts =
  Table[sinc[x]^2 sinc[1.2 y]^2, {x, -3 π, 3 π, δ}, {y, -3 π, 3 π, δ}];
(* count the points *)
Length[pts]
```

```
361
```

```
(* density plot, coarse data *)
g1 = ListDensityPlot[pts, Mesh → False];
```

Plot 33.. We begin a quick look at color scales. The `Hue` scale is convenient, but it wraps. That means that red is high and red is low, highly undesirable.

```
(* the trouble with Hue: it wraps *)
(* i.e. red is high and red is low *)
g1 = DensityPlot[sinc[x]^2 sinc[1.2 y]^2, {x, -3 π, 3 π}, {y, -3 π, 3 π},
   Mesh → False, ColorFunction → Hue, PlotPoints → 150];
```

Plot 34.. This is a color scale based upon `Hue` which is called `hot`. It only uses part of the `Hue` map and this precludes the wrapping problem.

```
(* we'll define a color map with Hue that does not wrap *)
(* red is hot (high) *)
hot[x_] := Hue[(1 - x) 6.3/7.7];
```

This plot shows the intrinsic tool *Mathematica* provides for adding legends. As you can see, the legend is added to the completed plot.

In order to optimize the arrangement on the page, this explanatory sequence is out of order. Hence, we are grabbing the plot from figure 36. and adding the legend to that.

```
(* add a color legend to the following density plot *)
(* getting the ShowLegend command *)
<< Graphics`Legend`;
(* color map for the legend is a pure function *)
g2 = ShowLegend[g1, {Hue[# 6.6/7.7] &, 10, " 1",
   " 0", LegendPosition → {1.05, -0.3}}, PlotPoints → 150];
```

Plot 35.. We could use the `Raster` command just like the `DensityPlot` command, but we wanted to try something different. We wanted the raster regions (each rectangle) to represent a pixel on a *charge-coupled device* (CCD) camera. What does this mean? Instead of treating the raster region like a point sample we instead treat it as a tiny integrator giving it a finite extent. The bounds of integration corresponding to the physical edge of the light-collecting region. More precisely, the pixel measures the total amount of light received which has the effect of averaging the light over the collection region. Remember, we just know the total number of photons collected over the region, we do not know their distribution. In essence, we really have the average value of the distribution function over the collection region.

In one dimension, consider a distribution function *f(x)*. What we collect over the domain {a,b} in one dimension is:

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

and of course, the average value of the function is

$$\overline{f(x)} = (b-a)^{-1}\int_a^b f(x)dx \tag{A1.1}$$

While the measured quantity is the integral of *f(x)*, the usable quantity is the average of *f(x)*. So if you are told that the integral has a certain value, that information cannot be used until you know the size of the domain.

While we will continue to work with same basic function, now we need an integrated form. We show the simplest definition of the sinc function here so that we may integrate analytically.

```
(* the irradiance profile is separable,
 so we only define one dimension *)
sinc[x_] := Sin[x]/x;
(* think of the raster cell as an integrator *)
Integrate[sinc[x]^2, {x, a, b}, Assumptions → {a ∈ Reals, b ∈ Reals}]

If[b == 0 || b > 0 && a ≥ 0 || b < 0 && a ≤ 0,
 Sin[a]^2/a - Sin[b]^2/b - SinIntegral[2 a] + SinIntegral[2 b],
 Integrate[Sin[x]^2/x^2, {x, a, b},
  Assumptions → a ∈ Reals && b ∈ Reals && (a > 0 && b < 0 || a < 0 && b > 0)]]
```

(If you are evaluating symbolic manipulation packages, a good test is this integra1.) The basic integral is:

$$\int \text{sinc}^2(x)dx = Si(2x) - x\,\text{sinc}^2(x) \tag{A1.2}$$

where *Si(x)* is the sine integral function.



Sine Integral

There is a scale factor to account for (the 1.2 in sinc(1.2y)), but we wanted to show this form to establish a more clear functional relationship. Since the irradiance profile is separable, we only need to worry about a one-dimensional representation because we can multiply the results together as shown here.

```
(* integrate with the scale factor *)
Integrate[sinc[w x]^2, {x, a, b}, Assumptions → {a, b} ∈ Reals]
```

$$\frac{\frac{\text{Sin[a w]}^2}{a} - \frac{\text{Sin[b w]}^2}{b} - w\,\text{SinIntegral[2 a w]} + w\,\text{SinIntegral[2 b w]}}{w^2}$$

So we see that:

$$\int \text{sinc}^2(wx)dx = \frac{Si(2wx)}{w} - x\,\text{sinc}^2(wx) \qquad \text{(A1.3)}$$

Notice the suppression of conditions in the second `Integrate` command when we introduced the scale parameter *w*.

By looking at the two `Integrate` commands we see the basic integration and a reminder to check for the special cases where the boundary is zero. However, if we use the full form of the `sinc` function, this problem is taken care of for us.

```
(* full form of definition includes the value at x = 0 *)
sinc[x_] := Sin[x]/x /; x ≠ 0
sinc[x_] := 1        /; x == 0
```

Now we are ready to define the indefinite integral of the square of this function.

```
(* idefinite integral of the square of the sinc function *)
IIsinc2[w_, x_] := w^-1 SinIntegral[2 w x] - x sinc[w x]^2
IIsinc2[x_] := IIsinc2[1, x]
```

With the indefinite integrals we can build the results for definite integrals.

```
(* general case for definite integration *)
IDsinc2[w_, a_, b_] := IIsinc2[w, b] - IIsinc2[w, a]
```

At last we are ready to define a function `pixel` which integrates the light over the domains `{a,b}` and `{c,d}` with scale factors `{w1,w2}`.

```
(* mathematical representation of light collecting pixels *)
pixel[w1_, a_, b_, w2_, c_, d_] := IDsinc2[w1, a, b] IDsinc2[w2, c, d]
```

To create the CCD array, we call `pixel` and pass the integration domains and the scale factors of 1 and 1.2.

```
(* pixel size is δ×δ *)
δ = π / 30;
(* irradiate the pixels in the CCD array *)
ccd = Table[pixel[1, x, x + δ, 12/10, y, y + δ],
   {x, -3 π, 3 π - δ, δ}, {y, -3 π, 3 π - δ, δ}];
```

Some attention is required because we used the scale factor 12/10 in lieu of 1.2, *Mathematica* has generated an enormous table of expressions which looks like this one.

```
(* Mathematica has returned exact results *)
ccd[[1]][[1]]
```

$$\left(\frac{30\,\text{Sin}\left[\frac{89\pi}{30}\right]^2}{89\pi} - \text{SinIntegral}\left[\frac{89\pi}{15}\right] + \text{SinIntegral}[6\pi]\right)$$

$$\left(-\frac{25\left(5+\sqrt{5}\right)}{864\pi} + \frac{125\,\text{Sin}\left[\frac{89\pi}{25}\right]^2}{534\pi} - \right.$$

$$\left.\frac{5}{6}\,\text{SinIntegral}\left[\frac{178\pi}{25}\right] + \frac{5}{6}\,\text{SinIntegral}\left[\frac{36\pi}{5}\right]\right)$$

Manipulation will be much faster if we convert to numeric values at this point. Also we need to scale by δ^{-2} to finish the calculation of the average as shown in equation (A1.1).

```
(* the numeric array is much faster to manipulate *)
ccdn = ccd / δ^2 // N;
```

A useful and helpful test is to inspect the maximum value and ensure that it is close to one. Physically, we expect the maximum value of the irradiance to be unity, but because the pixel averages over some small area near the peak, we expect an answer less than one. As we increase the sample density, this maximum value will get closer to unity.

```
(* ideally the maximum is 1; with finer *)
(* discretization the maximum comes closer to 1 *)
Max[ccdn]

0.997032
```

The same color map will be used.

```
(* we'll define a color map with Hue that does not wrap *)
(* red is hot (high) *)
hot[x_] := Hue[(1 - x) 6.3/7.7];
```

We create a graphics object g1 from the raster array and the graphics object g2 shows the CCD array with the proper aspect ratio.

```
(* view the simulated CCD output *)
g1 = Graphics[Raster[ccdn, ColorFunction → hot]];
g2 = Show[g1, AspectRatio → Automatic];
```

Plot 36.. Here we generate the basic `DensityPlot` that was used with the `Legend` in figure 34.. We use the color map that does not wrap:

```
(* we'll define a color map with Hue that does not wrap *)
(* red is hot (high) *)
hot[x_] := Hue[(1 - x) 6.3/7.7];
```

and the basic plot command is:

```
(* the new color scale is an improvement *)
g1 = DensityPlot[sinc[x]^2 sinc[1.2 y]^2, {x, -3 π, 3 π}, {y, -3 π, 3 π},
   Mesh → False, ColorFunction → hot, PlotPoints → 150];
```

Plot 37.. We are back to the `Raster` primitive. We will use the same functions to irradiate the CCD, but we will explore black and white shading options. The large central peak completely dominates the graphic and the default shading shows us this large peak and only hints that there may be side lobes. One standard practice to correct for this would be to apply a threshold in a gain function.

For example, suppose the threshold was 0.1. What we would want then is for the `GrayLevel` shading to vary from 0 to 1 and the function varies from 0 to 0.1. Of course we could use a `ColorFunction` to handle this for us, but we want to demonstrate the gain. The ensuing two plots use modified `ColorFunction`s.

The first step is to sweep through the pixels and replace any value over 0.1 with the threshold value 0.1. Then when we divide by the threshold value, we will end up with pixel values that vary from 0 to 1 and we can use the default `ColorFunction`. Here is the process in application.

```
(* apply a threshold *)
thresh = 0.1;
(* define a gain function *)
gain = (If[# ≥ thresh, thresh, #]) &;
```

Now we have some details to sort out. We need to apply the gain to the entire array. One way to do this is to linearize (`Flatten`) the array, apply the gain, and `Partition` the result back into a matrix.

```
(* we will apply gain to a linear array *)
fccdn = Flatten[ccdn];
(* apply the gain function to threshold the data *)
tccd = gain /@ fccdn;
(* since this array is square the linear size is √total size *)
n = √Dimensions[tccd] ;
(* repack into 2 D form; values ≥ thresh are now 1 *)
ccdn1 = Partition[tccd / thresh, n[[1]]];
```

We are now ready to generate the graphics.

```
(* create a graphics object using the Raster primitive *)
g1 = Graphics[Raster[ccdn1]];
(* display the graphic *)
g2 = Show[g1, AspectRatio → Automatic];
```

Plot 38.. Now we will use a modified `ColorFunction` to accentuate the side lobes. This `ColorFunction` will also black out an pixels above the threshold. This makes the side lobes seem more apparent.

```
(* set a threshold *)
thresh = 0.1;
(* define a ColorFunction *)
shade = (GrayLevel[If[# < thresh, # / thresh, 0]]) &;
```

We could always use one command to generate the graphic, but two steps help the user follow the process better.

```
(* create a graphics object using the Raster primitive *)
g1 = Graphics[Raster[ccdn1, ColorFunction → shade]];
(* display the graphic *)
g2 = Show[g1, AspectRatio → Automatic];
```

Plot 39.. We use a much lower threshold going from 0.1 in the previous graph to 0.005 here. The side lobes are quite apparent.

```
(* set a much lower threshold *)
thresh = 0.005;
```

We use the same plot commands.

```
(* create a graphics object using the Raster primitive *)
g1 = Graphics[Raster[ccdn1, ColorFunction → shade]];
(* display the graphic *)
g2 = Show[g1, AspectRatio → Automatic];
```

Plot 40.. We use `ParametricPlot` to draw an ellipse. This is not something we could do with a single command in `Plot` since the command `Plot` works with functions.

```
(* parametric plot of an ellipse *)
g1 = ParametricPlot[
   {2 Cos[t], Sin[t]}, {t, 0, 2 π}, AspectRatio → Automatic];
```

A1.4 Add-on package: graphics

There are a vast number of packages within the add-on package `Graphics`. In the examples we will show how to load the specific package that you need. Another strategy is to simply load all of the packages at once like this:

```
(* load all the packages *)
<< Graphics`
```

The disadvantage of this method is that you greatly increase the amount of named variables which increases chances for a shadowing conflict.

A1.4.1 Graphics'FilledPlot'

The first subpackage we will look at includes the commands to create filled plots. These can quickly show complicated laminae or highlight differences between functions.

Plot 41.. We begin a detailed survey of the add-on package `Graphics`. We will first examine the `FilledPlot` package. First, we load the package.

```
(* load the package *)
<< Graphics`FilledPlot`
```

For demonstration purposes, we look at a damped sine wave. *Mathematica* will color in the gap between the sine wave and the damping function.

```
(* damping function *)
f[x_] := Exp[-x/3]
(* view the test function and the damping function *)
g1 = FilledPlot[{f[x] Sin[x], f[x]}, {x, 0, 3 π}, PlotRange → All];
```

Plot 42.. What happens when we have several functions?

```
(* load the package *)
<< Graphics`FilledPlot`
```

```
(* build a table of the first few Legendre functions *)
fcns = Table[LegendreP[i, x], {i, 8}];
(* color in the areas between curves *)
g1 = FilledPlot[Evaluate[fcns], {x, -1, 1}, PlotRange → All];
```

Plot 43.. We turn our attention to filled plots of a single function which will subsequently be compared to discretely sampled versions of the same curve. As always, the first step is to load the package.

```
(* load the package *)
<< Graphics`FilledPlot`
```

Next, we define the curve of interest.

```
(* full form of definition includes the value at x = 0 *)
sinc[x_] := Sin[x]/x /; x ≠ 0
sinc[x_] := 1        /; x == 0
```

When we plot this curve, we see that *Mathematica* has applied a threshold in order to allow the smaller features to be more visible.

```
(* plot the sinc^2 function: note automatic thresholding *)
g1 = FilledPlot[sinc[x]^2, {x, -3 π, 3 π}];
```

Plot 44.. We will now redo the previous figure forcing the plot range to show the entire function. We will also add a frame and suppress the axes to improve the appearance. Observe that we do not have to rerender the curve.

```
(* we do note need to rerender the plot *)
g2 = Show[g1, PlotRange → All, Frame → True, Axes → False];
```

Plot 45.. The next two plots are discrete versions of the previous two plots. This plot is a coarsely sampled sinc^2 function. The first step is to load the appropriate package.

```
(* load the package *)
<< Graphics`FilledPlot`
```

We define the function of interest.

```
(* full form of definition includes the value at x = 0 *)
sinc[x_] := Sin[x] / x /; x ≠ 0
sinc[x_] := 1 /; x == 0
```

Now we sample the function and put these values into a list called `pts`.

```
(* select a coarse sample density *)
δ = π / 3;
(* create a list of sampled points *)
pts = Table[sinc[x]^2, {x, -3 π, 3 π, δ}];
```

When we plot the function, notice that the discrete version also is thresholded to make the smaller features more apparent. Also, notice the change in the values along the *x* axis.

```
(* select a coarse sample density *)
δ = π / 3;
(* create a list of sampled points *)
pts = Table[sinc[x]^2, {x, -3 π, 3 π, δ}];
```

Plot 46.. This plot will be a more finely sampled version of the previous plot and we will also add a frame and suppress the axes in order to facilitate comparison to plot 44.. We begin by loading the package:

```
(* load the package *)
<< Graphics`FilledPlot`
```

We define the function we are sampling.

```
(* full form of definition includes the value at x = 0 *)
sinc[x_] := Sin[x] / x  /; x ≠ 0
sinc[x_] := 1          /; x == 0
```

The list of sampled values is called `pts`.

```
(* select a fine sample density *)
δ = π / 30;
(* create a list of sampled points *)
pts = Table[sinc[x]^2, {x, -3 π, 3 π, δ}];
```

Compare this to plot 44.. Notice the dramatic change in the *x* values.

```
(* plot the list: notice the change in x-axis values *)
g1 = FilledListPlot[pts, Frame → True, Axes → False];
```

A1.4.2 Graphics'ImplicitPlot'

The implicit plot functions are real time-savers. They allow you to plot curves that are not functions and allow for instant plotting of inverse functions without having to define them.

Plot 47.. Implicit plots are a valuable feature that alleviates the task of inverting functions. It also allows us to plot curves that are not functions. We will first load the package:

```
(* load the package *)
<< Graphics`ImplicitPlot`
```

and then plot a curve which is not a function:

```
(* alleviates the task of solving for the inverse *)
g1 = ImplicitPlot[y^2 == x, {x, -1, 5}];
```

Plot 48.. Now we will plot a family of curves. We will look at an ellipsoid defined by

$$\frac{x^2}{a^2}+\frac{y^2}{b^2}+\frac{z^2}{c^2} = 1 \qquad \text{(A1.4)}$$

This is of course a three-dimensionsal surface and we have a two-dimensional tool. So what we will do is plot slices of the ellipsoid and these slices will act like the contours in a topological map. The first step is to load the needed package.

```
(* load the package *)
<< Graphics`ImplicitPlot`
```

Then we define the general ellipsoid.

```
(* general ellipsoid: we are not constrained to functions *)
eq1 = (x/a)^2 + (y/b)^2 + (z/c)^2 == 1
```

$$\frac{x^2}{a^2}+\frac{y^2}{b^2}+\frac{z^2}{c^2} == 1$$

For the plot we are forced to select a specific ellipsoid.

```
(* specific ellipsoid *)
{a, b, c} = {1, 0.85, 0.5};
(* create the contours *)
fcns = Table[eq1, {z, 0, c, 0.1}]
```

Note that we are able to plot the curves without having to use `Evaluate`.

```
(* plot slices of the ellipsoid *)
g1 = ImplicitPlot[fcns, {x, -a, a}];
```

Ellipsoid

A1.4.3 Graphics‘Graphics‘

The doubly named graphics package is a catchall of many useful routines. The logarithmic plots are in here. The polar plots are in here. Business-like routines for bar charts, pies charts and text listed charts are in here. The most overlooked category is the error charts.

Plot 49.. The next six plots demonstrate *Mathematica*’s three basic logarithmic plots in both continuous and discrete form. We start with the basic logarithmic curve. First, we load the package.

```
(* load the package *)
<< Graphics`Graphics`
```

Then we specify a function and an upper limit.

```
(* upper bound *)
ω = 10;
(* define a function for plotting *)
fcn[x_] := x Exp[-x];
```

Now we plot the function with a logarithmic scale (*x* linear, *y* logarithmic).

```
(* plot the function on a logarithmic scale *)
g1 = LogPlot[fcn[x], {x, 0, ω}];
```

Plot 50.. Here we will create a discrete version of the previous plot. First we load the package.

```
(* load the package *)
<< Graphics`Graphics`
```

Then we specify the same function and upper limit.

```
(* upper bound *)
ω = 10;
(* define a function for plotting *)
fcn[x_] := x Exp[-x];
```

Now we will discretely sample the function and store the values in the list `pts`.

```
(* sample the function discretely *)
pts = Table[fcn[x], {x, 0, ω, 1/10}];
```

Plotting is simple. Notice the change in the *x* axis values.

```
(* plot the function on a logarithmic scale *)
g1 = LogListPlot[pts];
```

Plot 51.. The last two graphs had a linear *x* scale and a logarithmic *y* scale. In these next two plots we will switch the scale. The *x* scale will be logarithmic and the *y* scale will be linear.

We load the package:

```
(* load the package *)
<< Graphics`Graphics`
```

and specify the same function as in the preceding two graphs.

```
(* upper bound *)
ω = 10;
(* define a function for plotting *)
fcn[x_] := x Exp[-x];
```

The format we want is the `LogLinearPlot` (*x* logarithmic, *y* linear).

```
(* plot the function on a log-linear scale *)
g1 = LogLinearPlot[fcn[x], {x, 0, ω}];
```

Plot 52.. Next is the discrete version of the same graph. As before we will sample the function and collect the values in a list called `pts`. We begin by loading the package.

```
(* load the package *)
<< Graphics`Graphics`
```

We define the same function and upper bound as the other graphs in this series.

```
(* upper bound *)
ω = 10;
(* define a function for plotting *)
fcn[x_] := x Exp[-x];
```

The function is sampled discretely

```
(* sample the function discretely *)
pts = Table[fcn[x], {x, 0, ω, 1/10}];
```

Notice the change in the *x* axes values between this plot and plot 51..

```
(* plot the function on a log-linear scale *)
g1 = LogLinearListPlot[pts];
```

Plot 53.. The final format has logarithmic scales for both axes. We proceed as usual by loading the package:

```
(* load the package *)
<< Graphics`Graphics`
```

and specify the same function as in the preceding four graphs.

```
(* upper bound *)
ω = 10;
(* define a function for plotting *)
fcn[x_] := x Exp[-x];
```

The format we want is the `LogLogPlot` (*x* logarithmic, *y* logarithmic).

```
(* plot the function on a log-log scale *)
g1 = LogLogPlot[fcn[x], {x, 0, ω}];
```

Plot 54.. To prepare the discretely sampled version of the previous plot we first load the package.

```
(* load the package *)
<< Graphics`Graphics`
```

We specify the same function and domain as plots 49.-53..

```
(* upper bound *)
ω = 10;
(* define a function for plotting *)
fcn[x_] := x Exp[-x];
```

Next, we collect sample values in the list `pts`.

```
(* sample the function discretely *)
pts = Table[fcn[x], {x, 0, ω, 1/10}];
```

The plot command has the expected syntax where the word `List` is inserted before `Plot`.

```
(* plot the function on a log-linear scale *)
g1 = LogLogListPlot[pts];
```

Plot 55.. The next two plots show different versions of the Limaçon of Pascal (not the famous Blaise Pascal whose name appears so frequently in mathematics; instead his father Étienne).

The general form for the limaçon is:

$$r = b + a\cos\theta\,. \tag{A1.5}$$

Special cases for this curve are these:

1. $b \geq 2a$ convex limaçon
2. $2a > b > a$ dimpled limaçon
3. $b = a$ cardioid
4. $b < a$ limaçon with an inner loop
5. $b = a/2$ trisectrix (limaçon with an inner loop)

We will prepare a list of functions that has examples of every case listed above. To do this we first load the package.

```
(* load the package *)
<< Graphics`Graphics`
```

We set $a = 1$. The parameter b will vary from 1/2 to 2 in increments of 1/4. This will create cases of each type of limaçon mentioned above. Specifically the cases will be:

1. $b = 1/2$ trisectrix (limaçon with an inner loop)
2. $b = 3/4$ limaçon with an inner loop
3. $b = 1$ cardioid
4. $b = 5/4$ dimpled limaçon
5. $b = 3/2$ dimpled limaçon

6. $b = 7/4$	dimpled limaçon
7. $b = 2$	convex limaçon

```
(* table of limacones *)
fcns = Table[b + Cos[θ], {b, 1/2, 2, 1/4}];
```

We will create an interesting graphic by plotting all these sections together.

```
(* plot the family of curves *)
g1 = PolarPlot[Evaluate[fcns], {θ, 0, 2 π}];
```

Limaçon

Plot 56.. The `PolarPlot` has a discrete version: `PolarListPlot`. We will plot a discretely sampled trisectrix. We load the package first.

```
(* load the package *)
<< Graphics`Graphics`
```

The values of $\{a, b\} = \left\{1, \frac{1}{2}\right\}$ specify the trisectrix. We discretely sample this curve.

```
(* create a trisectrix *)
b = 1/2;
(* discretely sample the curve *)
pts = Table[b + Cos[θ], {θ, 0, 2 π, π/100}];
```

The plot syntax is simple. Notice this `List` plot command does not change the values in either axes.

```
(* plot the trisectrix points *)
g1 = PolarListPlot[pts];
```

Plot 57.. *Mathematica* has some very flexible commands for creating bar charts which we will explore. As always, we start by loading the package.

```
(* load the package *)
<< Graphics`Graphics`
```

For a subject to plot, we will return to the Zernike polynomials that we saw in section 1.4.5. We will consider the following polynomials:

$$Z_{4,\,0}(x_1,\,x_2) = 4x_1x_2^3 - 4x_1^3x_2 \tag{A1.6}$$

$$Z_{4,\,1}(x_1,\,x_2) = 8x_1^3x_2 + 8x_1x_2^3 - 6x_1x_2 \tag{A1.7}$$

$$Z_{4,\,2}(x_1,\,x_2) = 6x_1^4 + 12x_1^2x_2^2 + 6x_2^4 - 6x_1^2 - 6x_2^2 + 1 \tag{A1.8}$$

$$Z_{4,\,3}(x_1,\,x_2) = -4x_1^4 + 4x_2^4 + 3x_1^2 - 3x_2^2 \tag{A1.9}$$

$$Z_{4,\,4}(x_1,\,x_2) = x_1^4 - 6x_1^2x_2^2 + x_2^4 \tag{A1.10}$$

In our codes we do not represent these polynomials as functions. Instead they are vectors of coefficients in a Cartesian basis space. The Cartesian basis that we are using is the set monomials

$$x_1,\,x_2),\,(x_1^2,\,x_1x_2,\,x_2^2),\,(x_1^3,\,x_1^2x_2,\,x_1x_2^2,\,x_2^3),\,(x_1^4,\,x_1^3x_2,\,x_1^2x_2^2,\,x_1x_2^3,\,. \tag{A1.11}$$

The relationship between the polynomials and the coefficient vectors c_{nm} is the following:

$$4x_1x_2^3 - 4x_1^3x_2 \Longrightarrow$$

$$= ((0),\,(0,\,0),\,(0,\,0,\,0),\,(0,\,0,\,0,\,0),\,(0,\,-4,\,0,\,4, \tag{A1.12}$$

$$8x_1^3x_2 + 8x_1x_2^3 - 6x_1x_2 \Longrightarrow$$

$$= ((0),\,(0,\,0),\,(0,\,-6,\,0),\,(0,\,0,\,0,\,0),\,(0,\,8,\,0,\,8, \tag{A1.13}$$

$$6x_1^4 + 12x_1^2x_2^2 + 6x_2^4 - 6x_1^2 - 6x_2^2 + 1 \Longrightarrow$$

$$= ((1),\,(0,\,0),\,(-6,\,0,\,-6),\,(0,\,0,\,0,\,0),\,(6,\,0,\,12,\,0, \tag{A1.14}$$

$$-4x_1^4 + 4x_2^4 + 3x_1^2 - 3x_2^2 \Longrightarrow$$

$$= ((0),\,(0,\,0),\,(3,\,0,\,-3),\,(0,\,0,\,0,\,0),\,(-4,\,0,\,0,\,0, \tag{A1.15}$$

$$x_1^4 - 6x_1^2x_2^2 + x_2^4 \Longrightarrow$$

$$= ((0),\ (0,\ 0),\ (0,\ 0,\ 0),\ (0,\ 0,\ 0,\ 0),\ (1,\ 0,\ -6,\ 0, \tag{A1.16}$$

So in *Mathematica* the quantity we store is the coefficient vectors.

```
(* 4th order Zernike coefficients for Cartesian coordinates *)
c = {{0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -4, 0, 4, 0},
   {0, 0, 0, 0, -6, 0, 0, 0, 0, 0, 0, 8, 0, 8, 0},
   {1, 0, 0, -6, 0, -6, 0, 0, 0, 0, 6, 0, 12, 0, 6},
   {0, 0, 0, 3, 0, -3, 0, 0, 0, 0, -4, 0, 0, 0, 4},
   {0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, -6, 0, 1}};
```

We can use a bar chart to represent these coefficient vectors. However, the `BarChart` command does not work with lists. So we cannot plot *c* immediately.

```
(* doesn't take lists! this syntax will not work *)
g1 = BarChart[c];
```

Two choices are apparent. The simplest method would be to specify the list members.

```
(* specify list members *)
g1 = BarChart[c[[1]], c[[2]], c[[3]], c[[4]], c[[5]]];
```

The more elegant solution is to use the `Sequence` command to splice in the list members. Both methods produce the result in plot 57..

```
(* the Sequence command extracts the elements from the list *)
g1 = BarChart[Sequence @@ c];
```

Plot 58.. There is a very general bar chart tool that allows the user to specify not only the heights, but also the widths and locations. We will create three data lists corresponding to the red, blue and green shaded bars in the plot. Each data set will have ten points in the format {position, height, width}.

First, we load the package.

```
(* load the package *)
<< Graphics`Graphics`
```

Since we are going to generate random data for plotting, it is best to seed the random number generator. This will allow the user to reproduce these results exactly.

```
(* seed the random number sequence for repeatability *)
SeedRandom[1];
```

The `GeneralizedBarChart` command plots a list of points in the form {position, height, width} and interprets them as describing a single bar on the chart. We will create expressions to generate the parameters individually and then bundle them into a point.

```
(* common use definitions *)
pos := 10 (Random[] - 1/2);    (* position *)
hgt := 5 (Random[] - 1/2);     (* height *)
wid := Random[];               (* width *)
datum := {pos, hgt, wid};      (* single point *)
```

For clarity, we will create three different data sets, each with a unique number of bars. The names of the sets are tied to the color *Mathematica* will use to display them.

```
(* data name indicates display color *)
(* data point format = position, height, width *)
datared = Table[datum, {10}];
datablue = Table[datum, {9}];
datagreen = Table[datum, {8}];
```

This command also will not accept lists. So as in the previous case we may either specify the list members individually

```
(* plot the three data sets *)
g1 = GeneralizedBarChart[datared, datablue, datagreen];
```

or we may use the `Sequence` command.

```
(* how would you plot a list of lists? *)
alldata = {datared, datablue, datagreen};
```

```
(* plot the three data sets *)
g1 = GeneralizedBarChart[Sequence @@ alldata];
```

Both commands produce the plot shown. We encourage users to acquaint themselves with the `Sequence` command.

Plot 59.. There are two similar commands, `StackedBarChart` and `PercentileBarChart` that have a similar usage and output. We will use an example of a `PercentileBarChart`. After we load the package we will create the data.

```
(* load the package *)
<< Graphics`Graphics`
```

We start by creating three lists, each with a different number of elements. Since we will use random data, we need to seed the sequence.

```
(* seed the random number sequence for repeatability *)
SeedRandom[1];
```

We also will ensure that the random sequences vary in amplitude and mean.

```
(* vary amplitudes and mean *)
lsta := 10 (Random[] - 1/3);
lstb := 8 (Random[] - 1/2);
lstc := 11 (Random[] - 1/2);
```

```
(* create lists of differing sizes *)
lstA = Table[lsta, {11}];
lstB = Table[lsta, {9}];
lstC = Table[lstc, {12}];
```

We are now ready to plot the data.

```
g1 = PercentileBarChart[lstA, lstB, lstC];
```

You'll notice that this plot command will not accept a list of lists as its argument. If you wanted to plot such a list, you would use the **Sequence** command.

```
(* how would you plot a list of lists? *)
alldata = {lstA, lstB, lstC};
```

```
(* plot the three data sets *)
g1 = PercentileBarChart[Sequence @@ alldata];
```

This produces the exact same plot.

Plot 60.. The `ErrorListPlot` is our nominee for the most valuable overlooked feature in *Mathematica*. This is a wonderful command that seems absent from the collective consciousness of the user community.

Our first step is to load the package.

```
(* load the package *)
<< Graphics`Graphics`
```

We will generate a curve with random noise and since we will be calling random numbers we need to seed the generator.

```
(* seed the random number sequence for repeatability *)
SeedRandom[1];
```

The format for the points passed to the plot command is

```
plot point format : {x, y, σy}
```

where σ_y is the error in the y measurement. In a real measurement this error would be computed or estimated. For the sake of expediency, in this case we have created a somewhat arbitrary function to generate the errors.

```
(* y value = 1 + x + noise term *)
(* point format is {x, y, error in y *)
pts = Table[y = 1 + x + Random[] - 1/2; {x, y, 3 (x^2 + y^2)^-1/2}, {x, 0, 10}];
```

The plot syntax is quite simple.

```
(* plot the points with the error terms *)
g1 = ErrorListPlot[pts, Axes → False, Frame → True];
```

Plot 61.. The next two plots show basics examples of the intrinsic tool for making pie charts. The initial step is to load the package.

```
(* load the package *)
<< Graphics`Graphics`
```

Since we are going to use random data to create table values, we need to seed the random number generator.

```
(* seed the random number sequence for repeatability *)
SeedRandom[1];
```

Now we can create a short list of plot values.

```
(* create some numbers for the pie chart *)
lst = Table[Random[], {6}];
```

Here too the plot syntax is straightforward.

```
(* display the pie chart *)
g1 = PieChart[lst];
```

Plot 62.. We return to the same data and chart in the previous plot. The tool `PieChart` allows us to accent an element by separating it from the pie. This will draw attention to the specific datum. Here we will show how to separate all the pieces.

First, we load the package.

```
(* load the package *)
<< Graphics`Graphics`
```

As before we will seed the random number generator.

```
(* seed the random number sequence for repeatability *)
SeedRandom[1];
```

and create a short data list.

```
(* create some numbers for the pie chart *)
lst = Table[Random[], {6}];
```

The option to explode all the pieces is straightforward.

```
(* display the pie chart *)
g1 = PieChart[lst, PieExploded → All];
```

Plot 63.. The next two charts show options for labeling data points in much the same manner as we did in section 3.**. In this first example we will plot a list of random points but instead of plotting dots for the points we will plot the address of the point in the list. The addresses here run from one to six.

We begin with the package load.

```
(* load the package *)
<< Graphics`Graphics`
```

As always, we seed the random number generator.

```
(* seed the random number sequence for repeatability *)
SeedRandom[1];
```

Next, we create a list of six points in the form `{x,y}`.

```
(* create some random points for the plot *)
lst = Table[{Random[], Random[]}, {6}];
```

The `TextListPlot` command will automatically plot the point number in lieu of a dot.

```
(* display the plot *)
g1 = TextListPlot[lst, Axes → False, Frame → True];
```

Plot 64.. If you feel the presentation in the previous plot is deficient because it does not show any dots, then you should use instead `LabeledListPlot`. This will produce both the dots and the text labeling.

We begin with the package load.

```
(* load the package *)
<< Graphics`Graphics`
```

Of course, we need to seed the random number generator before we request any random numbers.

```
(* seed the random number sequence for repeatability *)
SeedRandom[1];
```

We generate the same table of points as in plot 63..

```
(* create some numbers for the pie chart *)
lst = Table[{Random[], Random[]}, {6}];
```

The plot command here follows the same syntax as the previous plot.

```
(* display the pie chart *)
g1 = LabeledListPlot[lst, Axes → False, Frame → True];
```

A1.4.4 Graphics'InequalityGraphics'

This tool kit allows us to handle inequalities, which extends us beyond the equalities that we have been using so far. This package was used extensively to prepare the color wheels we created in section 3.1.5. We will in fact use the same three circles from that example and show how to create different laminae using logical operators.

Plot 65.. We will study the important `InequalityGraphics` package with three more plots. Section 3.1.5 also used this package and so curious readers may go there for more examples.

Being able to plot inequalities is certainly a great strength. We feel however, that the greatest leverage from this package comes from being able to solve multiple systems not constrained to functions. Also, the ability to create domains with logical functions is a boon. We will demonstrate these features through example using the same three circles from section 3.1.5.

First, we must load the package.

```
(* load the package *)
<< Graphics`InequalityGraphics`
```

We have manually extracted the circles from the example.

```
(* these are the circle from section 2.** *)
circles = {x^2 + y^2, (5/8 + x)^2 + (-5 Sqrt[3]/8 + y)^2, (-5/8 + x)^2 + (-5 Sqrt[3]/8 + y)^2};
```

Of course, these do not yet describe the disks until we turn them into inequalities.

```
(* generate equations for the lamina *)
lam = (# ≤ 1) & /@ circles
```

$$\left\{x^2 + y^2 \le 1,\ \left(\frac{5}{8} + x\right)^2 + \left(-\frac{5\sqrt{3}}{8} + y\right)^2 \le 1,\ \left(-\frac{5}{8} + x\right)^2 + \left(-\frac{5\sqrt{3}}{8} + y\right)^2 \le 1\right\}$$

These overlapping circles are made of three types of regions:

1. naked disk: involves just one disk
2. two disk overlap: involves two disks
3. three disk overlap: involves all three disks.

We will use logical operators to look at combinations of these regions.

First we apply the `XOR` operator to these three domains. This is a very interesting condition. We see from the Help Browser that the XOR operator will be `True` if an odd number of conditions are `True` and the rest are `False`.

When will these conditions be satisfied? In two places: the conditions 1 and three from the list above. These are the cases that have an odd number of `True`s.

We see that the `XOR` command cannot operate on a list.

```
(* Xor will not operate on a list *)
g1 = InequalityPlot[ Xor[lam[[1]], lam[[2]], lam[[3]]], {x}, {y} ];
```

Of course, we could use the special character for `XOR` generated by ESC xor ESC

```
(* Xor implemented with special characters *)
g1 = InequalityPlot[ lam[[1]] ⊻ lam[[2]] ⊻ lam[[3]], {x}, {y} ];
```

We encourage user's to learn how to apply the `Sequence` command.

```
(* sequencing is better idea than listing the elements *)
g1 = InequalityPlot[ Xor[ Sequence @@ lam], {x}, {y} ];
```

Plot 66.. We will revisit the previous plot and this time apply the `Or` function instead of the `Xor` function. We proceed with the same steps to set up the problem.

```
(* load the package *)
<< Graphics`InequalityGraphics`
```

```
(* these are the circle from section 3.1 .8 *)
circles = {x^2 + y^2, (5/8 + x)^2 + (-5 Sqrt[3]/8 + y)^2, (-5/8 + x)^2 + (-5 Sqrt[3]/8 + y)^2};
```

```
(* generate equations for the lamina *)
lam = (# ≤ 1) & /@ circles
```

$$\left\{x^2 + y^2 \le 1,\ \left(\frac{5}{8} + x\right)^2 + \left(-\frac{5\sqrt{3}}{8} + y\right)^2 \le 1,\ \left(-\frac{5}{8} + x\right)^2 + \left(-\frac{5\sqrt{3}}{8} + y\right)^2 \le 1\right\}$$

This time we use the logical `Or` function. We notice that the `Sequence` command will not work here.

```
(* sequencing will not work *)
g1 = InequalityPlot[Or[lam[[1]], lam[[2]], lam[[3]]], {x}, {y} ];
```

The special character representation is:

```
(* sequencing will not work *)
g1 = InequalityPlot[Or[lam[[1]], lam[[2]], lam[[3]]], {x}, {y}];
```

Plot 67.. This is the final entry in our three-plot look at the `InequalityGraphics` command. The previous two plots used the `Xor` and `Or` logical commands and here will use the `And` operator. This will select the third region in the list presented with plot 65.. The initial steps are the same for all three plots.

```
(* load the package *)
<< Graphics`InequalityGraphics`
```

```
(* these are the circle from section 3.1 .8 *)
circles = {x^2 + y^2, (5/8 + x)^2 + (-5 Sqrt[3]/8 + y)^2, (-5/8 + x)^2 + (-5 Sqrt[3]/8 + y)^2};
```

```
(* generate equations for the lamina *)
lam = (# ≤ 1) & /@ circles
```

$$\left\{x^2 + y^2 \le 1,\ \left(\frac{5}{8} + x\right)^2 + \left(-\frac{5\sqrt{3}}{8} + y\right)^2 \le 1,\ \left(-\frac{5}{8} + x\right)^2 + \left(-\frac{5\sqrt{3}}{8} + y\right)^2 \le 1\right\}$$

The `And` operator also does not allow sequencing.

```
(* sequencing will not work *)
g1 = InequalityPlot[ And[lam[[1]], lam[[2]], lam[[3]]], {x}, {y}];
```

The special character presentation looks like this.

```
(* equivalent formulation for this plot *)
g1 = InequalityPlot[ lam[[1]] ⋀ lam[[2]] ⋀ lam[[3]], {x}, {y}];
```

Plot 68.. We feel the `Legend` package deserves one more look in addition to plot 34.. We first load the package:

```
(* load the package *)
<< Graphics`Legend`
```

and define the sinc function.

```
(* define the sampling function *)
sinc[x_ /; x ≠ 0] := Sin[x] / x ;
sinc[x_ /; x == 0] := 1;
```

In order to simplify the **plot** command, we will use variables for two plot options. The first option supplies us with different line thicknesses to separate our curves. The second option is the text for the legend.

```
(* plot parameter: plot style *)
pst = {Thickness[0.0075], Thickness[0.015]};
(* plot parameter: legend text *)
plg = {"sinc[x]^2", "sinc[x/2]^2"};
```

We call your attention to one problem: in the legend, the lines have the same thickness.

```
(* line thicknesses not reflected in the legend *)
g1 = Plot[{sinc[x]^2, sinc[x/2]^2}, {x, -3 π, 3 π},
   PlotRange → All, PlotStyle → pst, PlotLegend → plg];
```

A1.4.5 Graphics‘MultipleListPlot‘

Here we will briefly discuss how to combine and view different discrete data sets. We did not touch on the topic very much, but you have a wide selection of plot symbols as well as the capability to create your own symbols. Also in plot 78. we show how to use defined variables to create dashed lines.

Plot 69.. The next eight plots will demonstrate different aspects of the command `MultipleListPlot`. Pay attention to how different lists automatically get different symbols and note the different options for connecting the symbols.

The first step is to load the package.

```
(* load the package *)
<< Graphics`MultipleListPlot`
```

We will be sampling the sinc function.

```
(* define the sampling function *)
sinc[x_ /; x ≠ 0] := Sin[x] / x;
sinc[x_ /; x == 0] := 1;
```

We will create two different lists by discretely sampling two different functions.

```
(* sample size *)
δ = π / 3;
(* first list of points *)
lst1 = Table[sinc[x]^2, {x, -3 π, 3 π, δ}];
(* second list of points *)
lst2 = Table[sinc[x / 2]^2, {x, -3 π, 3 π, δ}];
```

Plotting these two lists is simple.

```
(* note the different symbols for the two curves *)
g1 = MultipleListPlot[lst1, lst2];
```

Plot 70.. We will use the same data set as in the previous example and this time we will show how to connect the symbols on a selected set. The initial steps are the same.

```
(* load the package *)
<< Graphics`MultipleListPlot`
```

```
(* define the sampling function *)
sinc[x_ /; x ≠ 0] := Sin[x] / x ;
sinc[x_ /; x == 0] := 1;
```

```
(* sample size *)
δ = π / 3 ;
(* first list of points *)
lst1 = Table[sinc[x]^2, {x, -3 π, 3 π, δ}];
(* second list of points *)
lst2 = Table[sinc[x / 2]^2, {x, -3 π, 3 π, δ}];
```

We use a list of logical variables to dictate which set of points is connected.

```
(* note the different symbols for the two curves *)
g1 = MultipleListPlot[lst1, lst2, PlotJoined → {True, False}];
```

Plot 71.. We will proceed as before in the two previous plots, then add a third list of sample points.

```
(* load the package *)
<< Graphics`MultipleListPlot`
```

```
(* define the sampling function *)
sinc[x_ /; x ≠ 0] := Sin[x] / x ;
sinc[x_ /; x == 0] := 1;
```

This time we add a third list of points.

```
(* sample size *)
δ = π/3;
(* first list of points *)
lst1 = Table[sinc[x]^2, {x, -3π, 3π, δ}];
(* second list of points *)
lst2 = Table[sinc[x/2]^2, {x, -3π, 3π, δ}];
(* third list of points *)
lst3 = Table[sinc[x/3]^2, {x, -3π, 3π, δ}];
```

The additional curve is easy to add.

```
(* note the different symbols for the three curves *)
g1 = MultipleListPlot[lst1, lst2, lst3];
```

Plot 72.. Our final look at the basic `MultipleListPlot` takes the previous example and makes two changes. First, the creation of the data sets is simplified. Second, we use one command to specify that all lists should have the points joined.

We begin with the same initial steps.

```
(* load the package *)
<< Graphics`MultipleListPlot`
```

```
(* define the sampling function *)
sinc[x_ /; x ≠ 0] := Sin[x]/x;
sinc[x_ /; x == 0] := 1;
```

Rather than specifying the lists individually, we create a list of lists.

```
(* sample size *)
δ = π/3;
(* a more efficient way to generate the lists *)
lst = Table[Table[sinc[x/i]^2, {x, -3π, 3π, δ}], {i, 3}];
```

Next, we plot the list and specify that each sublist has the consecutive elements connected.

```
(* note the different symbols for the two curves *)
g1 = MultipleListPlot[lst, PlotJoined→True];
```

Plot 73.. The next two plots show different strategies for displaying data with the concomitant errors.

For the first series we will perform a series of linear regressions. We will vary the quality of the regression using two mechanisms: the sample density and the simulated noise. The input parameters for the fit were {slope, intercept} = {1,2}.

We begin by loading the package.

```
(* load the package *)
<<Graphics`MultipleListPlot`
```

Next, we define the parameters for our fit. We vary the sample density from low to high and we vary the noise from high to low. This will give us a mix of results. We expect to see fit results of distinctly different accuracy and precision.

```
(* create a set of sample densities *)
δ = {0.25, 0.1, 0.01, 0.001};
(* count the number of cases to consider *)
nd = Length[δ];
(* create a set of different levels of noise *)
noise = {1, 0.25, 0.1, 0.01};
(* count the number of cases to consider *)
nn = Length[noise];
```

We take a linear regression module from chapter 2.

```
(* linear regression routine based on section 2.3.2 *)
lr[a_List] := Module[{n, x, y, X, XX, Y, YX, χ, γ},
  (* scalable way to build orthonormal basis vectors *)
  {χ, γ} = IdentityMatrix[2];
  x = a.χ;                (* project out all x components *)
  y = a.γ;                (* project out all y components *)
  n = Length[a];          (* count the number of data points *)
  X = Total[x];           (* sum the x values *)
  XX = Total[x^2];        (* sum the x^2 values *)
  Y = Total[y];           (* sum the y values *)
  YY = Total[y^2];        (* sum the y values *)
  YX = x.y;               (* sum the x y values *)
  Δ = n XX - X^2;         (* determinant *)
  m = Δ^-1 (n YX - Y X);  (* slope *)
  b = Δ^-1 (XX Y - X YX); (* intercept *)
  (* merit function: sum of the squared errors *)
  χ2 = n b^2 + 2 b m X + m^2 XX - 2 b Y - 2 m YX + YY;
  (* estimated sample variance *)
  s2 = χ2/(n - 2);
  (* fit quality parameters *)
  σm = Sqrt[s2/Δ n];              (* slope error *)
  σb = Sqrt[s2/Δ X];              (* intercept error *)
  rc2 = 1 - χ2/Y^2;               (* correlation index *)
  Return[{{m, b}, ErrorBar[σm, σb]}];
 ];
```

Then we seed our random number generator and create the sixteen different raw data sets to feed to the linear regression module.

```
(* seed for repeatability *)
SeedRandom[1];
(* results for multiple lr *)
data = Table[
   Table[Table[{x, 2 + x + noise[[j]] Random[]}, {x, 0, 1, δ[[i]]}], {i, nd}],
   {j, nn}];
```

However, in this form the data needs to be restructured. The sixteen cases are packed into a four by four array and we want a linear array with a single index. This is a simple use of the `Flatten` command.

After flattening the array, we apply the linear regression module to it.

```
(* we need to put the data in a linear order for analysis *)
fdata = Flatten[data, 1];
(* create the plot points by analyzing the 16 data sets *)
pts = lr /@ fdata;
```

The plot is easy to create.

```
(* plot the results with the error *)
g1 = MultipleListPlot[pts, Frame → True,
   FrameLabel → {"slope", "itercept", "Regression results", ""}];
```

Plot 74.. For this plot we relied exclusively upon material in the Help Browser. First we load the package.

```
(* load the package *)
<< Graphics`MultipleListPlot`
```

Then we go into the Help Browser and execute `In[11]` which defines `myfunc`. Then we prepend `g1=` to `In[12]` which is the `MultipleListPlot` command. When we execute these two steps, the variable `g1` will contain the graphic.

```
(* Add-ons & Links >
  Standard Packages > Graphics > MultipleListPlot *)
Show[g1];
```

Add-ons & Links > Standard Packages > Graphics > MultipleListPlot

Plot 75.. This use of the `MultipleListPlot` may seem counterintuitive. For instead of passing multiple lists, we will pass a single list. This duplicates the example in the Help Browser showing a typical stem plot, a format useful in signal processing.

The first step is to load the package.

```
(* load the package *)
<< Graphics`MultipleListPlot`
```

Next, we create a table of plot points. Here we followed the example in Help Browser and used the sine function.

```
(* create a list of points *)
lst = Table[{x, Sin[x]} // N, {x, 0, 2 π, π/6}];
```

The plot command is quite simple.

```
(* produce a stem plot *)
g1 = MultipleListPlot[lst, SymbolShape → Stem];
```

Plot 76.. In this final `MultipleListPlot` we wanted to accentuate different ways to identify points through symbols and labels.

Our first step is to load the package.

```
(* load the package *)
<< Graphics`MultipleListPlot`
```

Following the example in the Help Browser we discretely sample two different curves.

```
(* create two lists of points *)
lst1 = Table[{x, Sin[x]} // N, {x, 0, 2 π, π/6}];
lst2 = Table[{x, Cos[x]} // N, {x, 0, 2 π, π/6}];
```

These are the data to be plotted. Now we will specify plot symbols and labels.

```
(* symbol shape information *)
syms = {PlotSymbol[Diamond, Filled→ False], Label};
(* symbol label information *)
syml = {Automatic, CharacterRange["a", "e"]};
```

These options are invoked in the plot command.

```
(* plot command *)
g1 = MultipleListPlot[lst1, lst2, SymbolShape→ syms,
   SymbolLabel→ syml, Frame→ True, PlotRange→ All];
```

A1.4.6 Graphics‘Spline‘

In this section we will use one data set to demonstrate the three splines available in *Mathematica*: cubic, Bezier and composite Bezier. The attentive reader will notice that the spline routines behave like a graphics primitive in that they need a `Graphics` wrapper to create a visual output. Mathematical details are found in the Help Browser entry from `NumericalMath‘SplineFit‘`.

Plot 78. is not part of this sequence; it is positioned here so that it would appear at the end of the `MultipleListPlot` package.

Plot 77.. We wanted to take a common list of points and show three different splines through them. Plots 77., 79. and 80. all use the same data. The splines will be very different.

First we load the package.

```
(* load the package *)
<< Graphics`Spline`
```

Then we create an arbitrary list of points. For comparison, we connect these points with a gray line and the spline fit will overlay this.

```
(* arbitrary points *)
pts = {{0, 0}, {1, 1}, {2, 2}, {1, -2}, {0, -2}, {1, -2}, {3, 3}};
(* arbitrary points *)
gpts = Graphics[{GrayLevel[0.5], Line[pts]}];
```

We fit the data set with a cubic spline. Notice that `Spline` is not a graphics command. It functions like a graphics primitive.

```
(* cubic spline *)
g1 = Show[gpts, Graphics[Spline[pts, Cubic]]];
```



Spline, Cubic Spline

Plot 78.. Due to clustering constraints, the desire to keep like figures together, we end up with seemingly misplaced cases like this one. Here we want to show the user different options available for using dashed lines.

We will use the **ListPlot** command to draw four separated lines of the same length. Each of these lines will have a different dashing pattern. The first step is to build a table of these endpoints.

```
(* table of line endpoints *)
endpts = Table[{{0, i}, {1, i}}, {i, 4}];
```

Next we build a table of dashing options.

```
(* line dashing options *)
dsh = {Dashing[{Dot}], Dashing[{Dot, Dash}],
   Dashing[{Dot, Dash, Dot, LongDash}], Dashing[{Dot, Dot, Dash}]};
```

Now we need to combine these two lists into a single list of points of the form {endpoints, style}. This will allow us to create a function to build the plots. We will use the process described in section 1.4.5 as well as section 2.3.3.

```
(* create a list of the format {endpt, style} *)
(* first combine the two lists *)
lst = {endpts, dsh};
(* the transpose creates the needed for (cf. List gymnastics ) *)
nlst = Transpose[lst];
```

With the list in hand, we can now compose a function to create the `ListPlot`s in the desired format.

```
(* this function sweeps through the list and creates a ListPlot *)
fcn = (ListPlot[#[[1]], Axes → False,
     PlotJoined → True, PlotStyle → #[[2]], doff]) &;
```

We can simply `Apply` this function to `nlst` to create our plots. Notice that the display has been turned off.

Now we can display this list of graphics objects being careful to turn the display back on.

```
(* show the results *)
g2 = Show[g1, don];
```

Plot 79.. The cubic spline and the Bezier spline are two of the most used spline fits. The previous plot showed an example of the cubic fit; this plot shows an example of the Bezier fit.

The first step is to load the package.

```
(* load the package *)
<< Graphics`Spline`
```

Then we create an arbitrary list of points. For comparison, we connect these points with a gray line and the spline fit will overlay this.

```
(* arbitrary points *)
pts = {{0, 0}, {1, 1}, {2, 2}, {1, -2}, {0, -2}, {1, -2}, {3, 3}};
(* arbitrary points *)
gpts = Graphics[{GrayLevel[0.5], Line[pts]}];
```

We fit the data set with a Bezier spline. Notice that `Spline` is not a graphics command. It functions like a graphics primitive.

```
(* cubic spline *)
g1 = Show[gpts, Graphics[Spline[pts, Bezier]]];
```



Bézier Spline

Plot 80.. Our final spline is the composite Bezier. The mathematical details of these spline fits can be found in `NumericalMath`SplineFit``.

We proceed by loading the package.

```
(* load the package *)
<< Graphics`Spline`
```

Then we create an arbitrary list of points. For comparison, we connect these points with a gray line and the spline fit will overlay this.

```
(* arbitrary points *)
pts = {{0, 0}, {1, 1}, {2, 2}, {1, -2}, {0, -2}, {1, -2}, {3, 3}};
(* arbitrary points *)
gpts = Graphics[{GrayLevel[0.5], Line[pts]}];
```

We fit the data set with a composite Bezier spline. Notice that `Spline` is not a graphics command. It functions like a graphics primitive.

```
(* cubic spline *)
g1 = Show[gpts, Graphics[Spline[pts, CompositeBezier]]];
```

A1.4.7 Graphics‘PlotField‘

This is a rich toolkit for plotting vector fields. Also, we can input scalar fields and `Mathematica` will perform vector operations like taking the gradient and plotting the vector output. The visualization of these fields is one of the most rewarding uses of the two-dimensional plot routines.

Plot 81.. We begin a entire page dedicated to the `Graphics‘PlotField‘` package. This is package is interesting and fun to use. We will start with the basic commands first.

To begin, we load the package.

```
(* load the package *)
<< Graphics`PlotField`
```

Then we define a vector-valued force field.

```
(* vector force field *)
f1 = {-y, x} / Sqrt[x^2 + y^2] ;
```

Plotting is easy.

```
(* plot the field values *)
g1 = PlotVectorField[f1, {x, -1, 1}, {y, -1, 1}, Frame → True];
```

Plot 82.. We have just plotted a nonconservative force and we would like to examine a conservative force. We load the package:

```
(* load the package *)
<< Graphics`PlotField`
```

and define the force.

```
(* vector force field *)
f2 = {x, y};
```

We plot the field values:

```
(* plot the field values *)
g1 = PlotVectorField[f2, {x, -1, 1}, {y, -1, 1}, Frame → True];
```

and ask if this is a conservative force. Equivalently, can we integrate the force to find a potential?

```
(* is this a conservative force? *)
(* if the potential field is given as ... *)
ϕ = (x^2 + y^2)/2
```

$$\frac{1}{2}\left(x^2 + y^2\right)$$

We will test this potential:

```
(* the force would be ... *)
F = {∂x ϕ, ∂y ϕ}
```

```
{x, y}
```

We ask *Mathematica* to do the comparison for us.

```
(* is this a conservative force? *)
F === f2
```

```
True
```



Conservative Field

Plot 83.. The command `PlotHamiltonianField` is named descriptively for it plots the Hamiltonian vector field of a scalar function. We load the package:

```
(* load the package *)
<< Graphics`PlotField`
```

define the usual scalar potential:

```
(* scalar potential *)
ϕ = (x^2 + y^2)/2 ;
```

and plot the Hamiltonian vector field.

```
(* plot the Hamiltonian vector field *)
g1 = PlotHamiltonianField[ϕ, {x, -1, 1}, {y, -1, 1}, Frame → True];
```

Since we are in the plane, this plot will be orthogonal to the gradient field, the subject of the next plot

Plot 84.. There is a command that accepts scalar potentials and plots the force, the gradient field: `PlotGradientField`.

We start with the loading of the package.

```
(* load the package *)
<< Graphics`PlotField`
```

We stick with the same scalar potential.

```
(* scalar potential *)
ϕ = (x^2 + y^2)/2 ;
```

We ask *Mathematica* to form the derivatives and plot them for us in one step.

```
(* plot the gradient vector field *)
g1 = PlotGradientField[ϕ, {x, -1, 1}, {y, -1, 1}, Frame → True];
```

Careful inspection of this plot and its predecessor can convince that in the plane, these two vector fields are orthogonal.

Plot 85.. Now we will demonstrate how to plot a Polya field. Since the argument accepts complex functions, we will use the Riemann zeta function.

First we load the package.

```
(* load the package *)
<< Graphics`PlotField`
```

There isn't anything else to do except to plot.

```
(* plot the Polya field of the Riemann Zeta function *)
p5 =
  PlotPolyaField[Zeta[x + i y], {x, -π, π}, {y, -π, π}, Frame → True];
```



Zeta Function

Plot 86.. The last three plots will deal with the discrete versions of the `PlotVector` command.

First, we load the package.

```
(* load the package *)
<< Graphics`PlotField`
```

Now we are ready to discuss the specifics of our problem. We start by defining a sample density, δ. The sample points will be separated by the distance δ.

```
(* discrete sample density *)
δ = 0.1;
```

Next, we define and discretely sample a vector field.

```
(* define a vector field *)
F[x_, y_] := {x, -y};
(* sample the vector field *)
vec = Table[F[x, y], {x, -1, 1, δ}, {y, -1, 1, δ}];
```

When we plot the vector field notice that the axes now count points: they do not correspond to the physical space of measurements.

```
(* plot the discrete vector field *)
(* axes marks are not physical units: they just count points *)
g1 = ListPlotVectorField[vec, Frame → True];
```

Plot 87.. For the next plot, we will simply change the sign on one of the arguments of the vector field.

But first we need to load the package.

```
(* load the package *)
<< Graphics`PlotField`
```

We want to keep the sample density the same.

```
(* discrete sample density *)
δ = 0.1;
```

Notice that we have only changed the sign of *y*.

```
(* define a vector field *)
F[x_, y_] := {x, y};
(* sample the vector field *)
vec = Table[F[x, y], {x, -1, 1, δ}, {y, -1, 1, δ}];
```

The plot is very different. But we have the same scaling malady: the axes count points; they do not correspond to the space of measurements.

```
(* plot the discrete vector field *)
(* axes marks are not physical units: they just count points *)
g1 = ListPlotVectorField[vec, Frame → True];
```

Plot 88.. For the final list plot, we will create the data in a different format that allows us to display the data in the same space the measurements were made in.

First, we load the package.

```
(* load the package *)
<< Graphics`PlotField`
```

We keep the same sample density.

```
(* discrete sample density *)
δ = 0.1;
```

Now we define a vector function and this time when we sample, we record the sample address {x, y} and save it in the data array.

```
(* define a vector field *)
F[x_, y_] := {x, y};
(* sample the vector field *)
(* point format is {{x,y}, F{x,y}} *)
vec = Table[{{x, y}, F[x, y]}, {x, -1, 1, δ}, {y, -1, 1, δ}];
(* collapse points into linear array *)
fvec = Flatten[vec, 1];
```

Now when we plot, our axes correspond to the measurement space.

```
(* plot the discrete vector field *)
(* axes marks are now physical units *)
g1 = ListPlotVectorField[fvec, Frame → True];
```

A1.4.8 Graphics‘ComplexMap‘

As explained so well in the Help Browser, any attempt to plot a complex valued function *f* of a complex variable *z* requires four dimensions: two for the input and two for the output.

If you have had a course in complex variables, then you remember how we studied mapping sets of lines in the complex plane. This package does these mappings for us.

Plot 89.. Now we start a full page of plots dedicated to the `ComplexMap` package. We begin with a display of the Cartesian coordinate system. Below this plot is a display of the polar coordinate system.

First, however, we need to load the package.

```
(* load the package *)
<< Graphics`ComplexMap`
```

Then we plot the coordinate system using the `Identity` operator.

```
(* the Cartesian coordinate system *)
g1 = CartesianMap[Identity, {-5, 5}, {-5, 5}];
```

This is an excellent reminder of the domain used here. The twin domains {-5, 5} here specify $\{x_{min}, x_{max}\}$ and $\{y_{min}, y_{max}\}$.

Plot 90.. This plot shows what the sine function looks like in a Cartesian map. After loading the package:

```
(* load the package *)
<< Graphics`ComplexMap`
```

we can plot the function immediately.

```
(* the Cartesian coordinate system *)
g1 = CartesianMap[Sin, {-1, 1}, {0, π}];
```

The domains here are for the polar variables *r* and θ. The first domain specifies that $-1 \leq r \leq 1$ and $0 \leq \theta \leq \pi$.

Plot 91.. Compare this plot coordinate presentation to the one above it, plot 89.. To create this plot we first load the package:

```
(* load the package *)
<< Graphics`ComplexMap`
```

Next, display the coordinate system using the `Identity` operator.

```
(* the polar coordinate system *)
g1 = PolarMap[Identity, {0, 1}, {0, 2 π}];
```

Plot 92.. Naturally since we are have the capability to plot complex functions we would want to look at the Riemann zeta function. But first, we need to load the package.

```
(* load the package *)
<< Graphics`ComplexMap`
```

Then we specify the function and a domain. Notice that the function `Zeta` is specified without arguments.

```
(* the Riemann Zeta function *)
g1 = PolarMap[Zeta, {0, 1}, {0, 2 π}];
```

Plot 93.. This first look at the zeta function has merely whetted the appetite. We now want to look at this function in a Cartesian map.

We load the package.

```
(* load the package *)
<< Graphics`ComplexMap`
```

Then issue a simple plot command. Notice the restricted domain. We will see this domain again in plot 95.. The profoundly different results will be a graphic reminder that the similar domains are for different coordinates.

```
(* the Riemann Zeta function *)
g1 = CartesianMap[Zeta, {0, 1/3}, {0, 2 π}];
```

Plot 94.. Let's look at the sine function in both coordinate systems. We'll start with the polar plot. Compare this with the Cartesian presentation in the plot below it, plot 96..

We begin with the package load.

```
(* load the package *)
<< Graphics`ComplexMap`
```

Now we plot the sine function. We will be careful to use the same domain with the polar coordinates. As noted earlier, we will get a profoundly different set of lines because the similar domains are for differing coordinates.

```
(* the sine function *)
g1 = PolarMap[Sin, {0, 11}, {0, 2 π}];
```

Plot 95.. We return to the zeta function first shown in plots 92. and 93..

We load the package first.

```
(* load the package *)
<< Graphics`ComplexMap`
```

Then we plot with the restricted domain.

```
(* the Riemann Zeta function *)
g1 = PolarMap[Zeta, {0, 1/3}, {0, 2 π}];
```

Plot 96.. Our final entry in the `ComplexMap` series is another look at the sine function in Cartesian coordinates.

We start by loading the package.

```
(* load the package *)
<< Graphics`ComplexMap`
```

Then we plot in Cartesian coordinates. We used the same domain in plot 94., except now the domain is for different coordinates. The plots are very different.

```
(* the sine function *)
g1 = CartesianMap[Sin, {0, 11}, {0, 2 π}];
```

A1.5 DiscreteMath‘

We now begin a survey of some very powerful visualization tools for discrete math. The tools are quite diverse and we will discuss three subpackages for combinatorics, computational geometry, and tree.

A1.5.1 DiscreteMath‘Combinatorica‘

The visualization tools for the combinatorics are extremely useful in trying to learn the subject. The best documentation available is in the Help Browser, and we refer the readers there. The dual entries correspond to the redundant ways to get to the `Combinatorica` package in the Help Browser.

Plot 97.. We now begin a page dedicated to the package `Combinatorica` package and its tools for building graphs. Our first example is a Ferrers diagram (notice the absence of an apostrophe). This type of figure is sometimes referred to as a Young diagram.

First, we load the package.

```
(* load the package *)
<< DiscreteMath`Combinatorica`
```

The plot syntax is elementary:

```
(* aka a Young diagram; dots represent partitions *)
g1 = FerrersDiagram[RandomPartition[500]];
```

Add-ons & Links > Standard Packages > DiscreteMath > Combinatorica

Add-ons & Links > Combinatorica

Ferrers Diagram

Plot 98.. The next seven plots will all deal with graphs in one fashion or another. Our first task is to load the package.

```
(* load the package *)
<< DiscreteMath`Combinatorica`
```

Here we show the complete graph on seven vertices, K_7.

```
(* K7 *)
g1 = ShowGraph[CompleteGraph[7]];
```

Add-ons & Links > Standard Packages > DiscreteMath > Combinatorica

Add-ons & Links > Combinatorica

Graph

Plot 99.. We see in the Help Browser that "A star is a tree with one vertex of degree n-1. Adding any new edge to a star produces a cycle of length 3."

To show an example, we first load the package.

```
(* load the package *)
<< DiscreteMath`Combinatorica`
```

We will show a star with 18 vertices and we will connect point 1 to point 3.

```
(* show a star with 18 points; connect points 1 and 3 *)
g1 = ShowGraph[AddEdge[Star[18], {1, 3}]];
```

Add-ons & Links > Standard Packages > DiscreteMath > Combinatorica

Add-ons & Links > Combinatorica

Graph

Plot 100.. What happens when we combine two copies of each edge of a star? We will perform this combination using the previous plot as a test case.

First we load the package.

```
(* load the package *)
<< DiscreteMath`Combinatorica`
```

Combining the copies is easy.

```
(* combine two copies of the edge
  of the star shown in the previous plot *)
g1 = ShowGraph[GraphSum[Star[18], Star[18]]];
```

Add-ons & Links > Standard Packages > DiscreteMath > Combinatorica

Add-ons & Links > Combinatorica

Graph, Tree

Plot 101.. You can read about trees in the Help Browser entry listed below and in the *CRC Concise Encyclopedia of Mathematics* entry for Tree. Here we will create a random tree

```
(* load the package *)
<< DiscreteMath`Combinatorica`
```

If you wish to duplicate this result, you will need to duplicate the seeding of the random number generator.

```
(* seed the random number sequence for repeatability *)
SeedRandom[1];
```

The display syntax is simple.

```
(* create a random tree with 15 nodes *)
g1 = ShowGraph[RandomTree[15]];
```

Add-ons & Links > Standard Packages > DiscreteMath > Combinatorica

Add-ons & Links > Combinatorica

Graph, Tree

Plot 102.. We next demonstrate how *Mathematica* allows you to contract graphs. First, we load the package.

```
(* load the package *)
<< DiscreteMath`Combinatorica`
```

Then we show the contraction.

```
g1 = ShowGraph[Contract[CompleteGraph[5, 5], {2, 7}]];
```

Add-ons & Links > Standard Packages > DiscreteMath > Combinatorica

Add-ons & Links > Combinatorica

Graph

Plot 103.. Different embeddings can reveal different aspects of a graph's structure. We will show one embedding and we encourage the inquisitive reader to experiment with the specified graph.

First, we load the package.

```
(* load the package *)
<< DiscreteMath`Combinatorica`
```

Somewhat arbitrarily we select the 30th embedding.

```
(* show the 30 th embedding *)
g1 = ShowGraph[RankedEmbedding[GridGraph[6, 6], {30}]];
```

Add-ons & Links > Standard Packages > DiscreteMath > Combinatorica

Add-ons & Links > Combinatorica

Graph, Embedding

Plot 104.. For the final plot in the `DiscreteMath'Combinatorica'` package, we show a feature that allows the user to view several interesting graphs. As explained in the Help Browser, this options allows one to see several related graphs.

First we load the package.

```
(* load the package *)
<< DiscreteMath`Combinatorica`
```

Then we plot the graphs in one fell swoop.

```
(* show all the related plots *)
g1 = ShowGraphArray[Partition[FiniteGraphs, 3, 5]];
```

Add-ons & Links > Standard Packages > DiscreteMath > Combinatorica

Add-ons & Links > Combinatorica

Graph

A1.5.2 DiscreteMath'ComputationalGeometry'

This is a potent and well-documented package. Also *The CRC Concise Encyclopedia of Mathematics* has many useful entries to augment the Help Browser.

Plot 105.. The next four plots give us a feel for the tools available in the `ComputationalGeometry` package within the `DiscreteMath` package.

We start by loading the package.

```
(* load the package *)
<< DiscreteMath`ComputationalGeometry`
```

Now we need a list of random points which will serve as the basis for triangulation. Of course we will seed the random number generator to allow the reader to exactly duplicate these results.

```
(* seed the random number sequence for repeatability *)
SeedRandom[1];
(* create a 2 D array or random data *)
data2D = Flatten[Table[10 {Random[], Random[]}, {4}, {4}], 1]

{{6.68693, 8.312}, {7.81807, 1.24634},
 {9.34537, 6.00252}, {7.58355, 9.69089},
 {1.25699, 5.75636}, {7.55052, 8.21438}, {6.04601, 1.94465},
 {6.57307, 5.59559}, {8.36731, 1.60536}, {8.54436, 9.45436},
 {9.37182, 2.60917}, {7.27288, 2.79132}, {2.68489, 4.29717},
 {9.4548, 1.54498}, {3.33952, 8.29465}, {1.87126, 1.85409}}
```

We will display the Delaunay triangulation of these points.

```
(* plot the Delauney triangulation of these points *)
g1 = PlanarGraphPlot[data2D];
```



Triangulation, Delaunay Triangulation

Plot 106.. A very important process is finding the convex hull, the list of points forming a bounding perimeter for the data set.

We start by loading the package.

```
(* load the package *)
<< DiscreteMath`ComputationalGeometry`
```

We will use the same data set as the previous example.

```
(* seed the random number sequence for repeatability *)
SeedRandom[1];
(* create a 2 D array or random data *)
data2D = Flatten[Table[10 {Random[], Random[]}, {4}, {4}], 1]

{{6.68693, 8.312}, {7.81807, 1.24634},
 {9.34537, 6.00252}, {7.58355, 9.69089},
 {1.25699, 5.75636}, {7.55052, 8.21438}, {6.04601, 1.94465},
 {6.57307, 5.59559}, {8.36731, 1.60536}, {8.54436, 9.45436},
 {9.37182, 2.60917}, {7.27288, 2.79132}, {2.68489, 4.29717},
 {9.4548, 1.54498}, {3.33952, 8.29465}, {1.87126, 1.85409}}
```

Next, we find the convex hull.

```
(* find the convex hull *)
chull = ConvexHull[data2D];
```

We want to plot the convex hull against the data points.

```
(* plot the convex hull and the data points *)
g1 = PlanarGraphPlot[data2D, chull];
```



Convex Hull

Plot 107.. *Mathematica* allows us to plot a diagram from a vertex list. First we load the package.

```
(* load the package *)
<< DiscreteMath`ComputationalGeometry`
```

Now we will use the same data set as in the previous two examples. Thanks to the benefit of seeding, we can recreate this string of random numbers simply by using the same seed.

```
(* seed the random number sequence for repeatability *)
SeedRandom[1];
(* create a 2 D array or random data *)
data2D = Flatten[Table[10 {Random[], Random[]}, {4}, {4}], 1]

{{6.68693, 8.312}, {7.81807, 1.24634},
 {9.34537, 6.00252}, {7.58355, 9.69089},
 {1.25699, 5.75636}, {7.55052, 8.21438}, {6.04601, 1.94465},
 {6.57307, 5.59559}, {8.36731, 1.60536}, {8.54436, 9.45436},
 {9.37182, 2.60917}, {7.27288, 2.79132}, {2.68489, 4.29717},
 {9.4548, 1.54498}, {3.33952, 8.29465}, {1.87126, 1.85409}}
```

Plotting the diagram is easy.

```
(* plot the diagram from the vertex list *)
g1 = DiagramPlot[data2D];
```

Plot 108.. This next example requires a bit more work than the others in this group. We want to plot a bounded diagram of the data set that we have been using, `data2D`. *Mathematica* will need to compute the convex hull, the bounded diagram vertices, and the bounded diagram adjacency list. We will need to compute the coordinates of a rectangular bounding box; a simple task with the vector tools available to us.

We begin by loading the package.

```
(* load the package *)
<< DiscreteMath`ComputationalGeometry`
```

We will use the same data set as the previous three examples.

```
(* seed the random number sequence for repeatability *)
SeedRandom[1];
(* create a 2 D array or random data *)
data2D = Flatten[Table[10 {Random[], Random[]}, {4}, {4}], 1]

{{6.68693, 8.312}, {7.81807, 1.24634},
 {9.34537, 6.00252}, {7.58355, 9.69089},
 {1.25699, 5.75636}, {7.55052, 8.21438}, {6.04601, 1.94465},
 {6.57307, 5.59559}, {8.36731, 1.60536}, {8.54436, 9.45436},
 {9.37182, 2.60917}, {7.27288, 2.79132}, {2.68489, 4.29717},
 {9.4548, 1.54498}, {3.33952, 8.29465}, {1.87126, 1.85409}}
```

We find the convex hull as we did in plot 106..

```
(* find the convex hull *)
chull = ConvexHull[data2D];
```

We compute the Delaunay triangulation displayed in plot 105..

```
(* find the Delaunay triangulation *)
delval = DelaunayTriangulation[data2D];
```

Now we need to find the bounding coordinates for `data2D`. This entails separating out the *x* and *y* coordinates into individual lists and applying the commands `Min` and `Max`.

```
(* find the bounding coordinates on the data set data2D *)
(* isolate the x and y values *)
x = data2D.{1, 0};
y = data2D.{0, 1};
(* find the min and max values for each list *)
{left, right} = {Floor[Min[x]], Ceiling[Max[x]]};
{bottom, top} = {Floor[Min[y]], Ceiling[Max[y]]};
```

We can now assemble a list of the bounding coordinates.

```
(* coordinates of the rectanglar region bounding data2D *)
b1 = {{left, bottom}, {right, bottom}, {right, top}, {left, top}};
```

We now rely upon *Mathematica* to compute the bounded diagram vertices and the bounded diagram adjacency list.

```
(* bounded diagram vertices, bounded diagram adjacency list *)
{diagvert1, diagval1} = BoundedDiagram[b1, data2D, delval, chull];
```

Now we can plot the results.

```
(* plot the bounded diagram of data2D *)
g1 = DiagramPlot[data2D, diagvert1, diagval1];
```



Adjacency List, Vertex (Graph), Voronoi Diagram

A1.5.3 DiscreteMath‘Tree‘

It is difficult to imagine learning about trees and graphs without a visualization tool such as this one. Compare the arguments to the plots. The arguments are arcane, while the plots are almost intuitive.

Plot 109.. We want to show to quick examples of the tools available for tree plots in the package `DiscreteMath‘Tree‘`.

First, we load the package.

```
(* load the package *)
<< DiscreteMath`Tree`
```

Then we will create a simple list using the range command and use this list to make a tree.

```
(* create a simple list and make a tree from this *)
tree = MakeTree[Range[6]]

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

Now we can plot the tree.

```
(* plot the tree *)
g1 = TreePlot[tree];
```



Tree

Plot 110.. In this example we will view an expression as a tree. We will just create a randomly complicated expression for study.

We first load the package.

```
(* load the package *)
<< DiscreteMath`Tree`
```

The we create a tree for the input expression.

```
(* view the expression as a tree *)
g1 = ExprPlot[f[g[x, y, z], g[x, h[x, y], z], g[x, j[x, y]]]];
```



Tree

A1.6 LinearAlgebra'MatrixManipulation'

This is an extremely useful, and often overlooked, standard package. We will only see a single example here because there is but one plot command within it. We hope that users will rely upon this tool to visualize large matrices.

Plot 111.. There is a handy tool for viewing sparse matrices and large matrices. The MatrixPlot command presents an array. Zero elements are shown in white; nonzero elements are shown in black.

To see an example, we first load the package.

```
(* load the package *)
<< LinearAlgebra`MatrixManipulation`
```

Then we create a large random matrix for display. Of course, we need to seed the random number sequence.

```
(* seed the random number sequence for repeatability *)
SeedRandom[1];
```

Now we are ready to view this large matrix.

```
(* white cells = 0; black cells ≠ 0 *)
g1 = MatrixPlot[Table[Random[Integer], {35}, {35}]];
```

A1.7 Miscellaneous

The rich and diverse packages within the Miscellaneous add-on package are reflected in the varied and diverse plots here. Since many subpackages have but one or two plots we will not bother to separate them topically in a subsection of text.

Plot 112.. In the first chapter we found the boiling point of gadolinium was 3539°K. What does the intensity spectrum look like for a blackbody at this temperature?

To find out we first must load the package.

```
(* load the package *)
<< Miscellaneous`BlackBodyRadiation`;
```

The plot is generated for us.

```
(* blackbody spectrum for molten gadolinium *)
g1 = BlackBodyProfile[3539 Kelvin];
```

Plot 113.. The ResonanceAbsorptionLines package has very interesting capabilities which we will sample with two plots. For the first plot, we will present an absorption spectrum for the elements over a specified range of wavelengths.

First we load the package.

```
(* load the package *)
<< Miscellaneous`ResonanceAbsorptionLines`;
```

Then we plot the absorption lines over the range 900-910 angstroms.

```
(* display the absorption map between 900 and 910 Angstroms *)
g1 = WavelengthAbsorptionMap[900 Angstrom, 910 Angstrom];
```

Plot 114.. We can also plot absorption spectra for each element.

Of course, we first need to load the package.

```
(* load the package *)
<< Miscellaneous`ResonanceAbsorptionLines`;
```

In this example we will look at boron.

```
(* display the absorption map for Boron *)
g1 = ElementAbsorptionMap[Boron];
```

Plot 115.. There is a handy package that describes the standard atmosphere. We will show two examples here.

First, we load the package.

```
(* load the package *)
<< Miscellaneous`StandardAtmosphere`;
```

Then we plot the dynamic viscosity as a function of altitude.

```
(* plot dynamic viscosity as a function of altitude *)
g1 = AtmosphericPlot[DynamicViscosity];
```

Plot 116.. The previous plot showed the dynamic viscosity of the atmosphere as a function of altitude. Let's also look at the kinetic temperature.

First, we load the package.

```
(* load the package *)
<< Miscellaneous`StandardAtmosphere`;
```

Then we request a plot of the kinetic temperature.

```
(* plot kinetic temperature as a function of altitude *)
g1 = AtmosphericPlot[KineticTemperature];
```

Plot 117.. The WorldPlot package is an extremely interesting package and we encourage the reader to explore this in the Help Browser. We shall limit ourselves to two examples.

First, we load the package.

```
(* load the package *)
<< Miscellaneous`WorldPlot`;
```

We decide to focus on Oceania. We want to use `RandomGrays` and after brief experimentation we decide to change our random number seed to create a shading scheme we like.

```
(* seed the random number sequence for repeatability *)
(* we changed the seed to get a shading we liked *)
SeedRandom[2];
```

Before we plot this region of the globe, we call your attention to an unpleasant detail. The output from these routines is not a `-Graphics-` object, but is instead a `-WorldGraphics-` object. The need to have a different graphics object is not clear to

us and the problem is that you cannot use the standard tools `Export` and `Display` to send the results to a file.

Certainly, the Help Browser sets a high standard for technical documentation, but this is a minor blemish. The Help Browser specifies two remedies; unfortunately, it does not specify them together. You have to do a lot of reading to secure this basic fact. An example would be most appreciated.

The two choices that you have to remedy this are:

1. Set the option `WorldToGraphics⟶True`
2. Convert the to a standard graphics object using the `Graphics` command.

Look at Oceania and generate a `-Graphics-` object in lieu of the default `-WorldGraphics-` object.

```
(* take a look at Oceania *)
g1 = WorldPlot[{Oceania, RandomGrays}, WorldToGraphics → True];
```

If you use the default settings, you will create a `-WorldGraphics-` object.

```
(* another way to convert to Graphics *)
g2 = WorldPlot[{Oceania, RandomGrays}, doff];
Head[g2]

WorldGraphics
```

The conversion is easy.

```
(* convert from WorldGraphics to Graphics *)
g3 = Graphics[g2]

- Graphics -
```

Plot 118.. For the next plot we wanted to show a specific world view.
First we need to load the package.

```
(* load the package *)
<< Miscellaneous`WorldPlot`;
```

Since we are using `RandomGrays` we need to seed the random number sequence. We choose a seed that produces a favorable shading pattern.

```
(* seed the random number sequence for repeatability *)
(* we changed the seed to get a shading we liked *)
SeedRandom[2];
```

To keep the plot command clean we will use variables to encode our choice for some options.

```
(* plot parameters *)
(* WorldRotation setting: rotation applied before projection *)
wrot = {90, 0, 105};
(* WorldRange setting: {{minlat, maxlat}, {minlong, maxlong}} *)
wran = {{0, 90}, {-180, 180}};
```

Here is the plot command.

```
(* show the plot *)
g1 = WorldPlot[{World, RandomGrays},
   WorldRotation → wrot, WorldRange → wran,
   WorldProjection → LambertAzimuthal, WorldToGraphics → True];
```

See the previous example (117.) for comments on outputting `-WorldGraphics-` output to files.

A1.8 NumericalMath‘

This is a rich assortment of utilities to perform some numerical work and to evaluate numerical results. We will constrain ourselves here to the plotting routines.

A1.8.1 NumericalMath‘Butcher‘

Extending Runge-Kutta fits to higher methods can be extremely difficult. The package is designed to relieve much of the computational burden. Interested readers should be sure to check the *MathSource* web address at the end of this section.

Plot 119.. The `NumericalMath'Butcher'` package is designed to help formulate higher-order Runge-Kutta methods. Much more specific information is found in the Help Browser and the references at the end of this example.

First, we load the package.

```
(* load the package *)
<< NumericalMath`Butcher`;
```

Then we create a list of the trees for a Runge-Kutta method of order four. This list is partitioned by order.

```
(* create a list partitioned by order of the trees for any Runge-
  Kutta method of order 4 *)
g1 = ButcherTrees[4]

{{f}, {f[f]}, {f[f[f]], f[f^2]},
 {f[f[f[f]]], f[f[f^2]], f[f f[f]], f[f^3]}}
```

It is helpful to view these trees as a plot.

```
(* plot these trees *)
g2 = ButcherPlot[g1];
```

This is one of the rare times that *The CRC Concise Encyclopedia of Mathematics* does not have a direct entry for a topic. While the Help Browser explanation is very good, there are some useful augmentations of this esoteric topic, the first of which is in Wolfram's Mathsource.

- *Runge-Kutta Order Conditions Package*: http://library.wolfram.com/infocenter/MathSource/1524/
- *Symbolic Derivation of Runge-Kutta Conditions*: http://www.lacim.uqam.ca/~plouffe/OEISarchive/SDRKOCfi.pdf

A1.8.2 NumericalMath'OrderStar'

Plot 120.. In numerical work we sometimes benefit from being able to describe a function with a rational function approximation. This package performs Padé approximations. The package `NumericalMath`Approximations`` has functions that perform minimax and general rational approximations.

We begin by loading the package.

```
(* load the package *)
<< NumericalMath`OrderStar`
```

We now specify a function, an expansion point and an order and generate the approximation function.

```
(* Pade approximation of the tangent function *)
(* centered at 0 of degree {1,0} *)
approx = Pade[Tan[z], {z, 0, 1, 0}]

z
```

Plot the convergence region.

```
(* draw the region where Abs[approx/Tan[z]] < 1 *)
g1 = OrderStar[approx, Tan[z]];
```

Padé Approximant

A very helpful explanation and *Mathematica* demonstration can be found at http://math.fullerton.edu/mathews/n2003/Web/PadeApproximationMod/Pade-ApproximationMod.html

A1.8.3 NumericalMath'Microscope'

Plot 121.. While arbitrary precision arithmetic is an enormous boon, it is unique to *Mathematica*. Because of this ability, *Mathematica* is able to quantify the errors in fixed precision computations.

Let's load the package and do some basic computations.

```
(* load the package *)
<< NumericalMath`Microscope`;
```

As explained in the Help Browser, numbers represented in a computer are a discrete set due to their binary representation in memory. This means that there will be gaps in the representation of non-integer numbers. The machine error is the difference between the computed number and its representation on the host machine.

For example, consider the function e^x at $x = 1$. We can compute the error with a simple command.

```
(* compute the error in compute e^x at x = 1 *)
g1 = MachineError[Exp[x], x → 1]

-0.325531 Ulps
```

The answer is measured in terms of **Ulps**: unit in the last place. This is just one digit in the least significant place. On this machine an **Ulp** is

```
(* compute the unit in the last place on this machine *)
Ulp[1]

1.11022 × 10^-16
```

Fortunately we are given the ability to plot these errors over an extended range.

```
(* plot e^x near 1 using machine arithmetic *)
g1 = Microscope[Exp[x], {x, 1}];
```

Plot 122.. A more revealing version of the information in the previous plot is to look not at the computed value of the function, but at the error in the representation.

The first step is to load the package.

```
(* load the package *)
<< NumericalMath`Microscope`;
```

Now we are able to display the error in **Ulps** as a function of x.

```
(* plot the error in evaluating e^x near x = 1 *)
g1 = MicroscopicError[Exp[x], {x, 1, 20}];
```

A1.9 Statistics‘StatisticsPlots‘

We end our whirlwind tour with a set of four plots from the package `Statistics‘StatisticPlots‘`.

Plot 123.. The first example is the box-and-whisker plot.
First we load the package.

```
(* load the package *)
<< Statistics`StatisticsPlots`;
```

Since we are using random data, we need to seed the random number generator.

```
(* seed the random number sequence for repeatability *)
SeedRandom[1];
```

We are now able to create a reproducible sequence of random numbers.

```
(* build a large random data set with 5 columns *)
data = Table[Random[], {100}, {5}];
```

This random array is the argument for the `BoxWhiskerPlot` command. Each column is summarized by one figure in the plot.

```
(* create the box-and-whisker plot *)
g1 = BoxWhiskerPlot[data];
```

Box-and-Whisker Plot

Add-ons & Links > Standard Packages > Statistics > Statistics Plots

Plot 124.. Next we will show an example of a quality control plot using the Pareto format.
First, we open the package.

```
(* load the package *)
<< Statistics`StatisticsPlots`;
```

Since we will be using random numbers, we will seed them for repeatability.

```
(* seed the random number sequence for repeatability *)
SeedRandom[1];
```

We will now create some random data.

```
(* create a table of random data *)
data = Table[Random[], {5}]

{0.600252, 0.758355, 0.969089, 0.125699, 0.575636}
```

As stated in the Help Browser the Pareto format combines a bar chart showing percentages of each category with a line plot showing the cumulative percentage.

```
(* present the Pareto quality control plot *)
g1 = ParetoPlot[data];
```

Add-ons & Links > Standard Packages > Statistics > Statistics Plots

Plot 125.. Here we will present a sample quantile-quantile plot using random data.

First, we load the package.

```
(* load the package *)
<< Statistics`StatisticsPlots`;
```

We seed the random number sequence so that readers may exactly duplicate this result.

```
(* seed the random number sequence for repeatability *)
SeedRandom[1];
```

Next, we create a list from two random data sets.

```
(* create a table of random data *)
data = {Table[Random[], {100}], Table[Random[], {300}]};
```

Then we are able to plot the data.

```
(* create a qunatile, quantile plot from data *)
g1 = QuantilePlot[data[[1]], data[[2]]];
```

Add-ons & Links > Standard Packages > Statistics > Statistics Plots

Quantile

Plot 126.. The final example from this package is a pairs scatter plot from multivariate data.

We load the package.

```
(* load the package *)
<< Statistics`StatisticsPlots`;
```

Then we seed the random number generator.

```
(* seed the random number sequence for repeatability *)
SeedRandom[1];
```

We create five sets of data of 100 points.

```
(* same data set as in plot 123 *)
data = Table[Random[], {100}, {5}];
```

The plot syntax is quite simple.

```
g1 = PairwiseScatterPlot[data];
```

Add-ons & Links > Standard Packages > Statistics > Statistics Plots

APPENDIX 2

A2.1 Pictionary of 3D graphic types

We hope that readers find the pictionary format a convenient way to explore the 3D graphics routines in *Mathematica*. The new user can be overwhelmed by the number of different options available and we hope it will be convenient to look for a format on these pages.

As we point out in the ensuing *Mathematica* code, there are some things that might confuse users. The `ParametricPlot3D` command is in the kernel, which means that it is available without loading any packages. However, there are versions of the same command that run only after you load `GraphicsParametricPlot3D`'.

Also, you will see commands for `SphericalPlot3D` and `Cylindrical-Plot3D`. While these commands can produce distinctly different kinds of plots, they can also produce the same kinds of plot.

We have designed this section so that you can flip through the plots and see which format you want to learn about. Then you can turn to the *Mathematica* code that produced that example. In general we tried to make the *Mathematica* code self-contained so that you could just run a few commands to create your own reproduction. In some cases this was not possible and we used code from other examples. However, these fragments are clearly marked and will explicitly tell you what is needed and where it is located.

Since we want the code fragments to be independent, we have duplicated some code: most notably the package loads. You only need to load a package once during your *Mathematica* session. It will then be available until you close the application.

We have tried to use the space as economically as possible. This means that in general we used the more powerful tools available and we also used differing approaches for the same procedure. So do not be alarmed if you see that same process performed different ways.

We have gone to great lengths to ensure that these examples are useful and interesting. We have included references to other works as well as to this book. We have

commented on the work heavily and we hope you are able to learn from these examples.

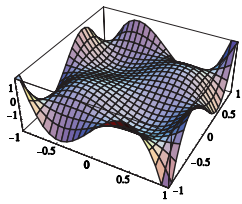
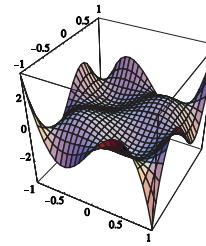
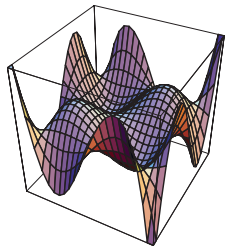
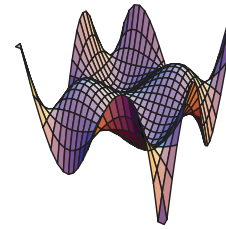
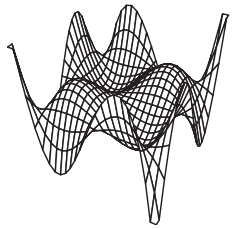
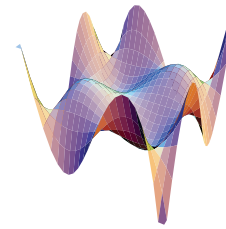
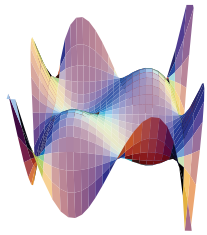
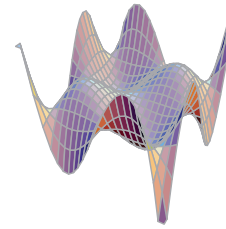
	
1. Plot3D default	**2. BoxRatios ⟶ {1,1,1}**
	
3. Axes ⟶ False	**4. Boxed ⟶ False**
	
5. Shading ⟶ False	**6. Mesh ⟶ False**
	
7. SphericalRegion ⟶ True	**8. MeshStyle ⟶ GrayLevel[0.6]**

Table A2.1: The most basic adjustments with the **Plot3D** command. For this series only plot 1 was created by plotting a function. All other plots were operations on the graphics object in plot 1.

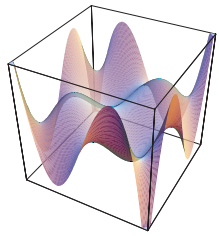
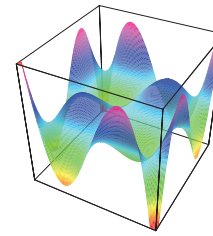
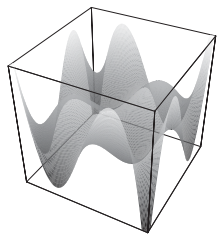
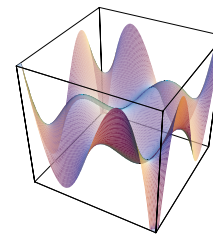
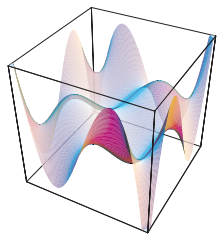
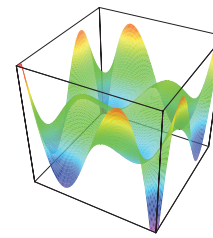
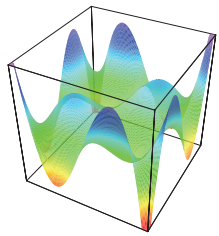
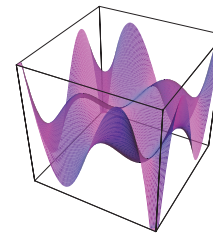
 **9. startingplot**	 **10. ColorFunction⟶Hue**
 **11. ColorFunction⟶GrayLevel**	 **12. ColorOutput⟶RGBColor**
 **13. ColorOutput⟶CMYKColor**	 **14. ColorFunction⟶redhot**
 **15. ColorFunction⟶redcold**	 **16. LightSources⟶lsrc**

Table A2.2: Once more we start with a single rendering, this time in higher resolution. We then demonstrate some basic options for coloring, starting with the same graphic object.

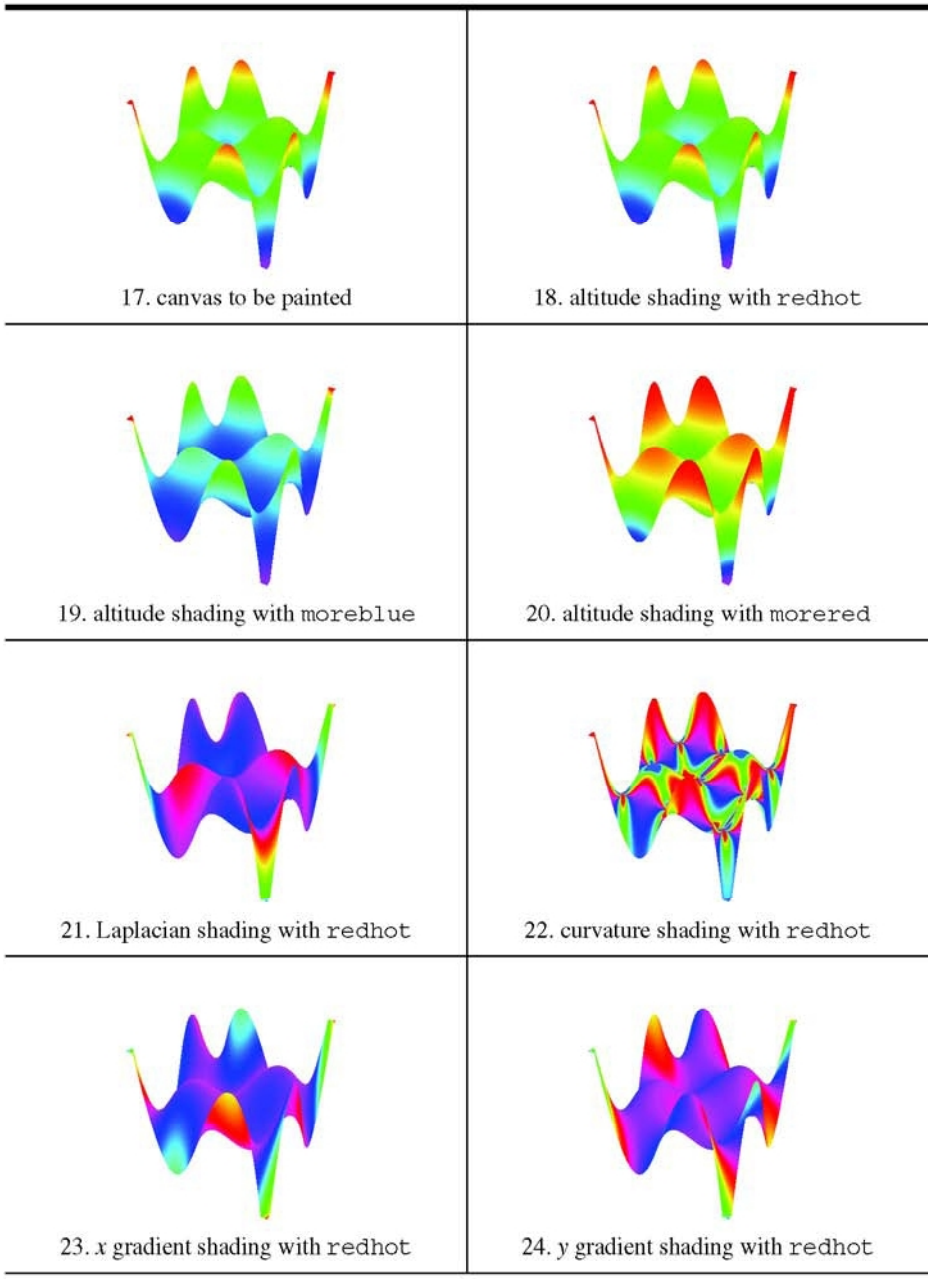


Table A2.3: Interesting shading options. We started with the figure in plot 17 and then used different shading schemes. Figures 18-20 use altitude shading with different color schemes. The remaining figures shade according to values for the Laplacian, the curvature and the gradient.

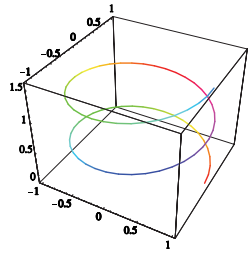
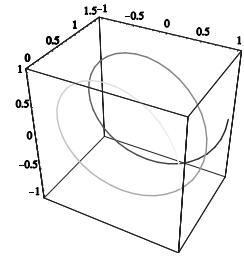
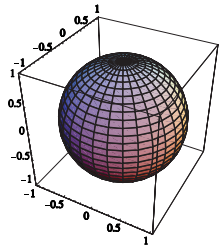
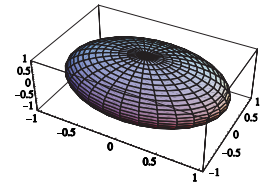
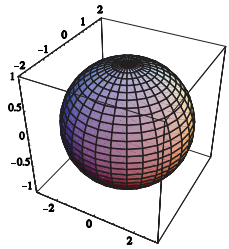
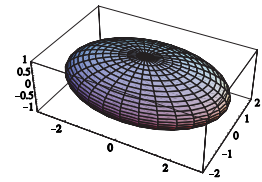
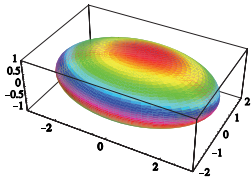
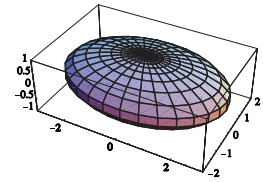
	
25. ParametricPlot3D, helix	**26. ParametricPlot3D, helix**
	
27. ParametricPlot3D, sphere	**28. ParametricPlot3D, sphere**
	
29. ParametricPlot3D, ellipsoid	**30. ParametricPlot3D, ellipsoid**
	
31. ParametricPlot3D, ellipsoid	**32. ParametricPlot3D, ellipsoid**

Table A2.4: The `ParametricPlot3D` package gives the user a valuable tool kit. As seen on this page, we can plot curves or surfaces and we have different ways to shade any figure. Notice however that we can make spheres look like ellipsoids and ellipsoids look like sphere. Careful attention to the axes establishes the pedigree.

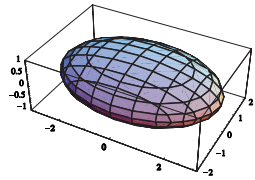
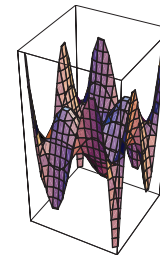
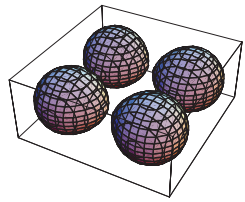
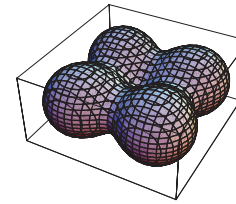
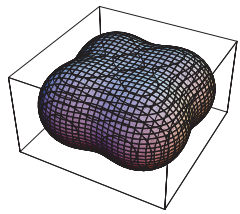
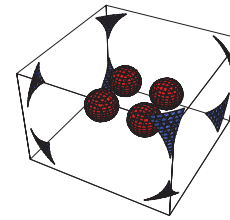
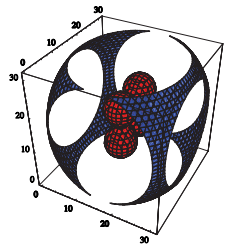
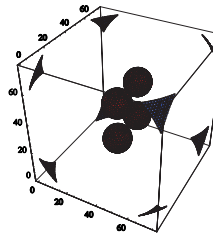
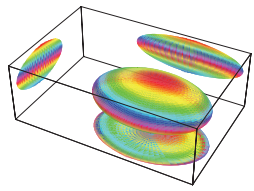
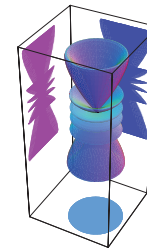
	
33. Graphics'ContourPlot3D'	**34. Graphics'ContourPlot3D'**
	
35. Graphics'ContourPlot3D'	**36. Graphics'ContourPlot3D'**
	
37. Graphics'ContourPlot3D'	**38. Graphics'ContourPlot3D'**
	
39. Graphics'ListContourPlot3D'	**40. Graphics'ListContourPlot3D'**

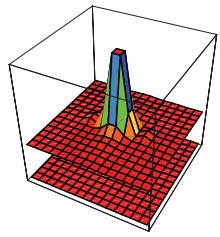
Table A2.5: Here we show some uses for the 3D contour plots that might seem novel. The first two images show how we can skip parameterization and create surfaces using the `ContourPlot3D`. The final six plots are of the same configuration of four positive charges.



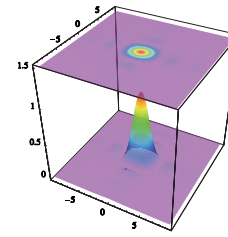
41. Shadow



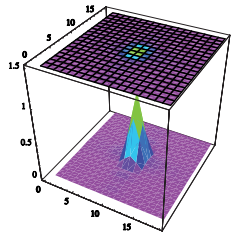
42. Shadow



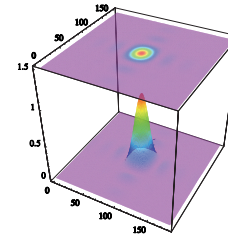
43. ShadowPlot3D



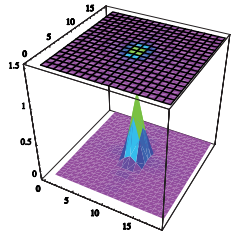
44. ShadowPlot3D



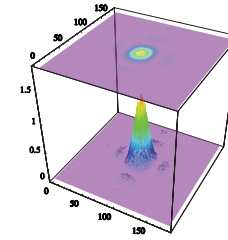
45. ListShadowPlot3D



46. ListShadowPlot3D

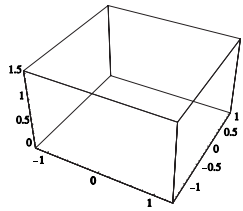


47. ListShadowPlot3D

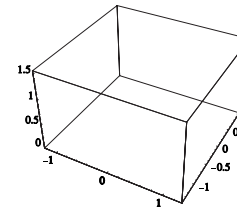


48. ListShadowPlot3D

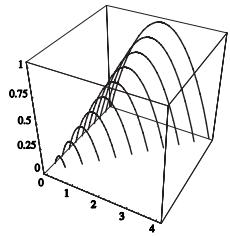
Table A2.6: The wise reader will play close attention to the different shadow capabilities in `Graphics`‘`Graphics3D`’. These can provide helpful perspectives in visually 3D shapes. A related command in this tool kit is `Project` which can project shadows in a completely arbitrary way. See the next table for an example.



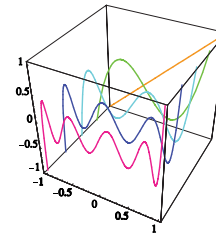
49. ScatterPlot3D



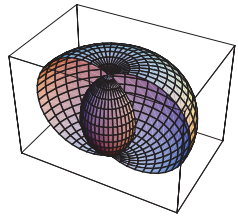
50. ScatterPlot3D



51. StackGraphics



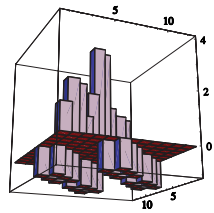
52. StackGraphics



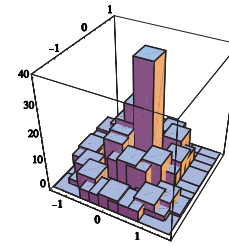
53. ListSurfacePlot3D



54. Project



55. BarChart3D



56. Histogram3D

Table A2.7: Closing our examination of the tools in `Graphics`'`Graphics3D`'.

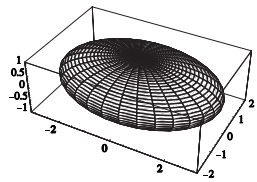
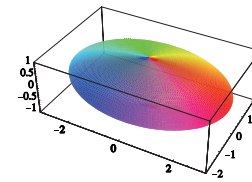
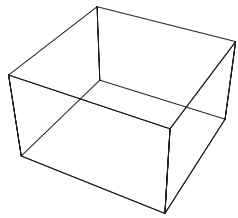
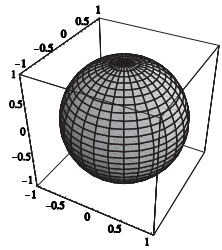
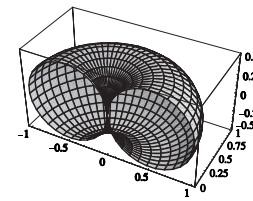
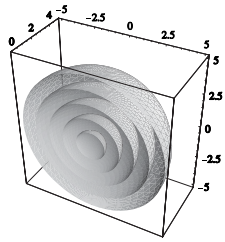
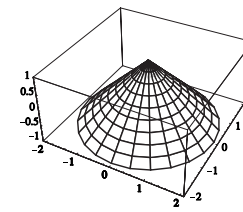
	
57. ParametricPlot3D	**58. ParametricPlot3D**
	
59. PointParametricPlot3D	**60. PointParametricPlot3D**
	
61. SphericalPlot3D	**62. SphericalPlot3D**
	
63. SphericalPlot3D	**64. CylindricalPlot3D**

Table A2.8: The `ParametricPlot3D` package gives the user a valuable tool kit. As seen on this page, we can plot curves or surfaces and we have a different ways to shade any figure.

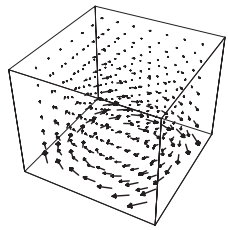
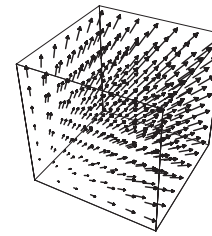
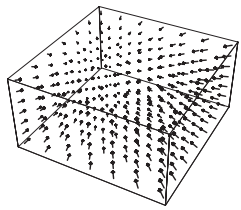
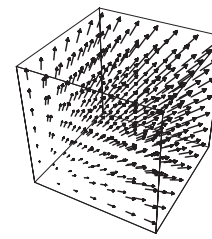
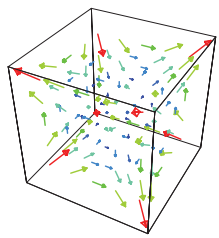
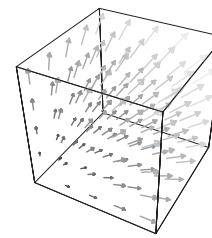
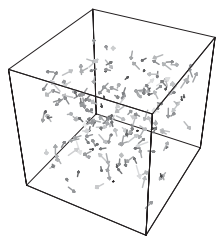
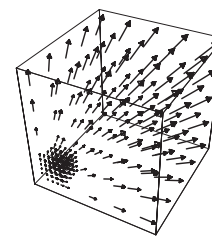
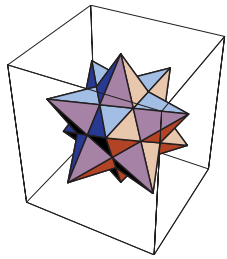
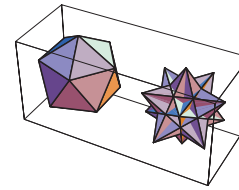
 **65. PlotVectorField3D**	 **66. PlotVectorField3D**
 **67. PlotGradientField3D**	 **68. PlotGradientField3D**
 **69. PlotGradientField3D**	 **70. PlotGradientField3D**
 **71. ListPlotVectorField3D**	 **72. ListPlotVectorField3D**

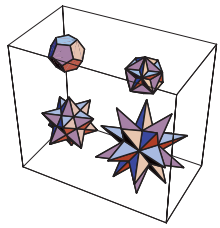
Table A2.9: Samples of the tools available in `Graphics‘PlotVectorField3D‘`. The figures in the right-hand column represent the same thing; there are just different paths to take. While these commands are invaluable when working with vector fields, their utility is somewhat mitigated by limited coloring options.



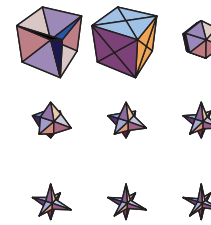
73. Polyhedron



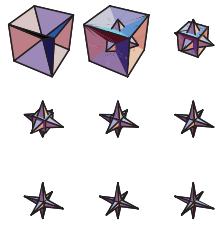
74. Polyhedron



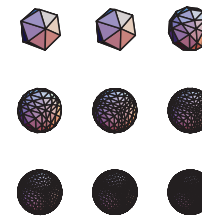
75. Polyhedron



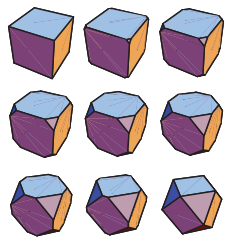
76. Polyhedron, positive stellation



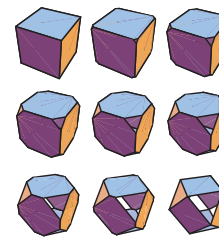
77. Polyhedron, negative stellation



78. Polyhedron, tessellation projection



79. Polyhedron, truncation



80. Polyhedron, open truncation

Table A2.10: A survey of the `Graphic`Polyhedra`` package. Not only are several different types of solids defined in the package, but you can also perform operations demonstrated here like tessellation, truncation, and geodesation.

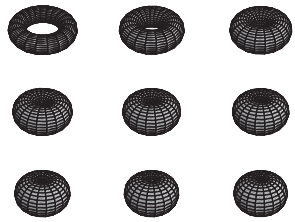
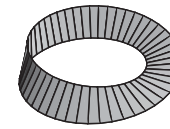
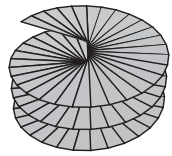
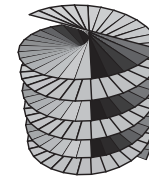
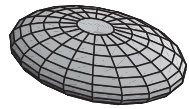
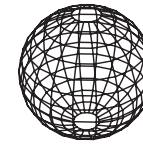
 **81. Graphics3D[Torus[...]]**	 **82. Graphics3D[MoebiusStrip[...]]**
 **83. Graphics3D[Helix[...]]**	 **84. Graphics3D[DoubleHelix[...]]**
 85. AffineShape	 **86. WireFrame**
 **87. SurfaceOfRevolution**	 **88. SurfaceOfRevolution**

Table A2.11: A look at some of the interesting shapes available in the `Shapes` package. Also, the final two figures show two different views of Gabriel's horn created with the *SurfaceOfRevolution* package.

A2.2 Kernel commands

We begin our survey with the 3D graphics routines available in the kernel. Because these commands are in the kernel, there is no need to load a package before we can use them.

Plot 1: To study the 3D plot routines, we created a series of orthogonal functions over the unit square by applying the Gram-Schmidt orthogonalization process to the series $1,\ x,\ y,\ x^2,\ xy,\ y^2,\ \ldots$ Out of this series of functions we have selected:

```
(* define a 2D function *)
f[x_, y_] := 7/8 (9 x y - 15 x^3 y - 15 x y^3 + 25 x^3 y^3)
```

We present the default plot.

```
(* default plot *)
g1 = Plot3D[f[x, y], {x, -1, 1}, {y, -1, 1}];
```



Gram-Schmidt Orthonormalization

Plot 2: Here we give the figure more realistic proportioning and extend the plot range to show the sharply pointed extrema.

```
(* isotropic space *)
g2 = Show[g1, BoxRatios → {1, 1, 1}, PlotRange → All];
```

Plot 3. Now we suppress the axes. Notice the numbering is on the axes; the frame still is present.

```
(* suppress the axes *)
g3 = Show[g2, Axes → False];
```

Plot 4. Here we turn off the frame which is controlled through the `Boxed` parameter.

```
(* remove the box frame *)
g4 = Show[g3, Boxed → False];
```

Plot 5. Turn of the shading and only show the mesh. This is good for black and white publication.

```
(* turn off the shading *)
g5 = Show[g4, Shading → False];
```

Plot 6. Let's go back to plot 4 and turn off the mesh.

```
(* turn off the mesh *)
g6 = Show[g4, Mesh → False];
```

Plot 7. This is how you change the viewpoint. If you want all your figures to appear to be the same size, then you need to use `SphericalRegion⟶True` as done here.

```
(* change the viewpoint; enforce spherical region bounding *)
g7 =
  Show[g6, ViewPoint → {0.070, -2.400, 2.000}, SphericalRegion → True];
```

Plot 8. Instead of turning the mesh off, you may wish to mute the appearance.

```
(* change mesh style *)
g8 = Show[g4, MeshStyle → GrayLevel[0.6]];
```

Plot 9. We begin a look at plot options in higher resolution. We begin with the same function we used on the previous page.

```
(* define a 2D function *)
f[x_, y_] := 7/8 (9 x y - 15 x^3 y - 15 x y^3 + 25 x^3 y^3)
```

We have brought over our favorite options from the preceding page.

```
(* starting high resolution plot *)
ga =
  Plot3D[f[x, y], {x, -1, 1}, {y, -1, 1}, PlotPoints → 150, Axes → False,
   BoxRatios → {1, 1, 1}, Mesh → False, SphericalRegion → True];
```

Plot 10. The `Hue` coloring option is convenient since it can take a single parameter, but it has a fatal flaw in that it wraps. That is, red is high and red is low.

```
(* note color wrapping: red is high, red is low *)
gb = Show[ga, ColorFunction → Hue];
```

Plot 11. `GrayLevel` is a good option for black and white printing.

```
(* note color wrapping: black and white version *)
gc = Show[ga, ColorFunction → GrayLevel];
```

Plot 12. For color output there are two basic choices: RGB and CMYK. Use `RGBColor` if your plots are for display on a monitor or screen. Since most laser printers use CMYK toners, any printed output should be rendered in `CMYKColor`.

This is the familiar (and default) `RGBColor`.

```
(* note color wrapping: red is high, red is low *)
gd = Show[ga, ColorOutput → RGBColor];
```

Plot 13. This is the `CMYKColor` option.

```
(* print processes often use CMYK colors *)
ge = Show[ga, ColorOutput → CMYKColor];
```

Plot 14. While hue does suffer from the malady of wrapping, it can be remedied easily. Here we will only use 80% of the scale. This will make red hot (high) and blue cold (low). Here is a simple function definition

```
(* hue color map that doesn't wrap *)
Clear[redhot];
redhot[x_] := Hue[(1 - x) 0.8];
```

Here is an equivalent form as a pure function.

```
(* alternative definition as a pure function *)
Clear[redhot];
redhot := Hue[(1 - #) 0.8] &;
```

This is what plot 9 looks like with altitude shading using the `redhot` color scheme.

```
(* note color wrapping: red is high, red is low *)
gf = Show[ga, ColorFunction → redhot];
```



Gram-Schmidt Orthonormalization

Plot 15. It is a simple matter to reverse this color scale and make red be the low color.

```
(* red is now cold *)
redcold := Hue[0.8 #] &;
```

This becomes the new color function.

```
(* note color wrapping: red is high, red is low *)
gg = Show[ga, ColorFunction → redcold];
```

Plot 16. Finally, we want to simply point out that you can vary the position and the color of the three virtual lights used to illuminate graphics objects. We will show one example here and we encourage the reader to look at the cited material at the end of this chapter for a more thorough presentation.

First, we define three different colors for the lights.

```
(* define some colors *)
cpur = RGBColor[0.4, 0, 0.8];
cblu = RGBColor[0, 0, 1];
cred = RGBColor[1, 0, 0];
```

We plug these colors into the light source list. We do not change the positions of the lights.

```
(* change the lighting colors *)
lsrc = {{{1, 0, 1}, cpur}, {{1, 1, 1}, cred}, {{0, 1, 1}, cblu}}

{{{1, 0, 1}, RGBColor[0.4, 0, 0.8]},
 {{1, 1, 1}, RGBColor[1, 0, 0]}, {{0, 1, 1}, RGBColor[0, 0, 1]}}
```

Notice that we do not have to redraw the figure. We can change the illumination using the `Show` command.

```
(* note color wrapping: red is high, red is low *)
gh = Show[ga, LightSources → lsrc];
```

Demos > Graphics Galleries > Color Charts > Light Source Variations

Demos > Graphics Galleries > Color Charts > Surface Color Variations

Plot 17. For the plots on this page we will exploit the `ListPlot3D` command. While this may seem a trivial extension of the `Plot3D` command, we will discover that it offers an easy way to color a diagram using a parameter other than the plot value.

We will start with the same 2D function that we have been using so far.

```
(* define a 2D function *)
f[x_, y_] := 7/8 (9 x y - 15 x^3 y - 15 x y^3 + 25 x^3 y^3)
```

Except that now we wish to sample the function over regular intervals to form a grid of plot values. The list `lst` will contain the function values for plotting for all figures on the page.

```
(* distance between samples *)
δ = 1/200;
(* sample at fine resolution *)
lst = Table[f[x, y], {x, -1, 1, δ}, {y, -1, 1, δ}];
```

We will use the variable `params` to contain the common plot parameters which we want to use for the plot page. The variable allows us to ensure that all plots have the same appearance and it simplifies the plot command syntax.

```
(* bundle the plot parameters *)
params = {Mesh → False, Axes → False, Boxed → False,
   BoxRatios → {1, 1, 1}, SphericalRegion → True};
(* use this plot as a canvas *)
g1 = ListPlot3D[lst, params];
```

Plot 18. One of the easiest chores is to shade the figure by altitude. This means that the plot value is what is used to determine the color. The slot operator # in the function redhot is where the plot value will come in. If we were plotting $z = f(x, y)$, then the slot operator will be where the z values are passed in.

```
(* hue color map that doesn't wrap *)
redhot = Hue[(1 - #) 0.8] &;
```

Notice that we do not need to redraw the figure with `ListPlot3D`. We can simply call the graphic and apply the new color scheme.

```
(* shade by altitude *)
(* g1 is the graphics object shown in plot 17 *)
g2 = Show[g1, ColorFunction → redhot];
```

Plot 19. The next two plots will experiment with variations of the previous color schemes. What we wanted to do was to reduce the amount of green near the zero values of the function to help us see the smaller features better.

To do this we are just take the square root of the function value that we pass to the `Hue` command. The `Norm` command produces the desired answer when we take the square root of a negative number. This slows the transition from blue to green

```
(* use more blues *)
moreblue = Hue[Norm[√(1 - #)] 0.8] &;
```

There is no need to render the figure again; we can recolor the figure in the `Show` command.

```
(* shade by altitude *)
(* g1 is the graphics object shown in plot 17 *)
g2 = Show[g1, ColorFunction → moreblue, don];
```

Plot 20. Next, we will boost the red colors by squaring the function value passed to the hue command. This will accelerate the transition to red colors.

```
(* use more reds *)
morered = Hue[(1 - #)^2 0.8] &;
```

When you look at this color scheme you will see why we call it `morered`.

```
(* shade by altitude *)
(* g1 is the graphics object shown in plot 17 *)
g2 = Show[g1, ColorFunction → morered, don];
```

Plot 21. Now we want to shift paradigms and create some shadings that depend on parameters other than the altitude. As we create the plot values list we can now create a list of shading directives to pass to `ListPlot3D`. The new syntax is

```
(* new paradigm *)
g1 = ListPlot3D[plot values, shading commands]
```

This is entirely different. Before we only had access to the z or altitude values. Now we have access to the two parameters: x and y or θ and ϕ. This allows to compute a value for, say, the Laplacian. And that is precisely what we shall do in this plot.

We will stick with the same basic function.

```
(* define the same 2 D function as in plot 17 *)
f[x_, y_] := 7/8 (9 x y - 15 x^3 y - 15 x y^3 + 25 x^3 y^3)
```

And we shall sample the function as before. In addition, we will need to keep a list of the sample addresses in `addr`. Tending to the details, you will see that we measure the two-dimensional array. This is because we need to flatten the arrays to simplify our functions. The size of the square array, `na`, will be used to fold or `Partition` the linear arrays back into two-dimensonsal form.

```
(* distance between samples, same as in plot 17 *)
δ = 1/200;
(* sample addresses *)
addr = Table[{x, y}, {x, -1, 1, δ}, {y, -1, 1, δ}];
(* count the sample points *)
na = Length[addr];
(* flatten to apply pure functions to a linear list *)
faddr = Flatten[addr, 1];
```

The linear list `faddr` is the flattened address list. We will now create the list of function values in a linear list `lst` since we acted on the linear list `faddr`. We will `Partition` the function values into the square array `plst`.

```
(* sample the function values for plotting *)
lst = f[#[[1]], #[[2]]] & /@ faddr;
(* create a 2D plot list for ListPlot3D *)
plst = Partition[lst, na];
```

Now we are ready to compute the Laplacian for this function. We start with the analytic computation and use this result to define the function `lap`.

```
(* formulate the Laplacian function *)
lap[x_, y_] = Simplify[D[f[x, y], {x, 2}] + D[f[x, y], {y, 2}]]
```

$$\frac{105}{4}\, x\, y\, (-6 + 5\, x^2 + 5\, y^2)$$

We will apply the function `lap` to the flattened address list and create a linear list of Laplacian values `lval`.

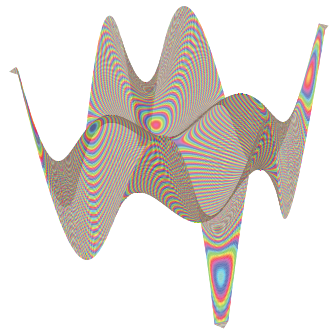
```
(* these are the values passed to the shading routine *)
(* compute the Laplacian at the sample points *)
lval = lap[#[[1]], #[[2]]] & /@ faddr;
```

However, these values are not quite ready to pass to a color function. Look at the range of the function.

```
(* we need to normalize the color scale *)
{Max[lval], Min[lval]}
```

```
{105, -105}
```

If we passed the existing values to the color function, we would end up with this monstrosity. We see that the color scale is cycled through over 100 times.



This drives us to normalize the scale.

```
(* shading directives for the Laplacian values, linear form *)
(* same color map as plot 18 *)
shdlap = redhot /@ lval/Max[lval] ;
```

The list of `shdlap` contains the shading directives generated by passing the normalized plot values to the function `redhot`. After we `Partition` this list into a square array:

```
(* fold the directives into a list with the sams shape as plst *)
pshdlap = Partition[shdlap, na];
```

we are ready to plot the figure with the shading determined by the value of the Laplacian.

```
(* params is defined in plot 17 *)
(* plot the figure and shade using the Laplacian *)
g1 = ListPlot3D[plst, pshdlap, params];
```



Laplacian

Plot 22. The next figure will be more dramatic. We will shade the figure according to the value of the curvature. As we see in formula (9) in the cited reference, the curvature κ of a surface described by:

$$g(x,\ y)\ =\ 0 \tag{A2.1}$$

is given by

$$\kappa\ =\ \frac{g_{xx}f_y^2 - 2g_{xy}g_xg_y + g_{yy}g_x^2}{(g_x^2 + g_y^2)^{3/2}} \tag{A2.2}$$

where the subscripts denote differentiation.

We will, of course, ask *Mathematica* to compute the derivatives for us, starting with the x derivatives.

```
(* compute the basic derivative functions *)
(* x derivatives *)
gx = ∂x f[x, y]
gxx = ∂x,x f[x, y]
```

$$\frac{7}{8}\ (9\,y - 45\,x^2\,y - 15\,y^3 + 75\,x^2\,y^3)$$

```
7/8 (-90 x y + 150 x y^3)
```

Next are the y derivatives.

```
(* y derivatives *)
gy = ∂y f[x, y]
gyy = ∂y,y f[x, y]
```

$$\frac{7}{8}\ (9\,x - 15\,x^3 - 45\,x\,y^2 + 75\,x^3\,y^2)$$

```
7/8 (-90 x y + 150 x^3 y)
```

Finally, we have the mixed derivative.

```
(* mixed derivatives *)
gxy = ∂x,y f[x, y]
```

$$\frac{7}{8}\,(9 - 45\,x^2 - 45\,y^2 + 225\,x^2\,y^2)$$

We can combine these results to create a curvature function.

```
(* compute the curvature for a 2D implicit curve *)
κ[x_, y_] = FullSimplify[ (gxx gy^2 - 2 gxy gx gy + gyy gx^2) / (gx^2 + gy^2)^(3/2) ]
```

$$-(2\,x\,(-3 + 5\,x^2)\,y\,(-3 + 5\,y^2)\,(3 - 15\,y^2 + 50\,y^4 - 5\,x^2\,(3 + 25\,y^4) + 25\,x^4\,(2 - 5\,y^2 + 25\,y^4))) \Big/ \left(25\,x^6\,(1 - 5\,y^2)^2 + y^2\,(3 - 5\,y^2)^2 + x^2\,(9 - 180\,y^2 + 525\,y^4 - 250\,y^6) + 5\,x^4\,(-6 + 5\,y^2\,(21 - 60\,y^2 + 25\,y^4))\right)^{3/2}$$

When we apply the curvature function to the address list, we encounter some trouble.

```
(* compute the curvature at the sample points *)
kval = κ[#[[1]], #[[2]]] & /@ faddr // N;
```

```
⚠ Power::infy : Infinite expression 1/0^(3/2) encountered . More...

⚠ ∞::indet : Indeterminate expression 0 ComplexInfinity encountered . More...
```

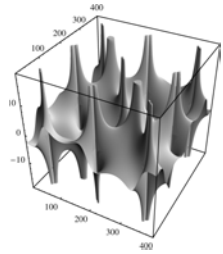
For now we will simply force the nonnumerical values to be zero. This is certainly not rigorous; it is merely expeditious.

```
(* define a function that replaces non-
  numeric values with numeric values *)
sweep = If[NumericQ[#], #, 0] &;
```

We apply this pure function to the curvature values list and create a new curvature values list `nkval`.

```
(* create a list of numeric values *)
nkval = sweep /@ kval;
```

If you were to plot this values list, you would see



and perhaps you would share our conclusion to threshold the data between -10 and 10. The values after applying the threshold are in the list `tkval`.

```
(* threshold value *)
τ = 10;
(* threshold function *)
fcn = (If[# ≥ τ, τ, If[# ≤ -τ, -τ, #]]) &;
(* thresholded data *)
tkval = fcn /@ nkval / τ ;
```

Now we are ready to apply the function redhot to the normalized, thresholded values list.

```
(* shading directives for the curvature values, linear form *)
(* same color map as plot 18 *)
shdcur = redhot /@ tkval / Max[tkval] ;
```

This linear list needs to be partitioned into a square array.

```
(* fold the directives into a list with the same shape as plst *)
pshdcur = Partition[shdcur, na];
```

At last we are able to plot. Notice that we are using the list of function values `plst` from plot 21 and the plot parameters from plot 17.

```
(* plst computed for plot 21 *)
(* use plot parameters from plot 17 *)
(* plot the figure and shade using
  the x component of the divergence *)
g1 = ListPlot3D[plst, pshdcur, params];
```



Curvature

Plot 23. In the next two plots we will shade the figure according to the value of the gradient. In general for a two-dimensional function in Cartesian space the gradient is given by

$$\nabla f(x,\ y) = (f_x,\ f_y) \tag{A2.3}$$

This is a vector quantity and we will use two plots. The first plot will be shaded according to the x value of the gradient and the second plot will be shaded using the y value of the gradient.

Both graphs will use the `redhot` color map. The first step is to compute the x component of the gradient and for that we will turn to *Mathematica*.

```
(* compute the x component of the gradient function *)
gradx[x_, y_] = Simplify[∂x f[x, y]]
```

$$\frac{21}{8}\,(-1+5x^2)\,y\,(-3+5y^2)$$

We will apply this function to the address list and create a list of values of the x gradient at each address, `dval`.

```
(* compute the x component of the gradient at the sample points *)
gval = gradx[#[[1]], #[[2]]] & /@ faddr;
```

We will need to normalize this list so that the color function will create reasonable results.

```
(* we need to normalize the color scale *)
{Max[gval], Min[gval]}

{21, -21}
```

The maximum values are the same magnitude. This makes it easier to normalize the data. The list `shddiv` is the list of color directives based upon the value of the *x* derivative.

```
(* shading directives for the x component of the gradient values,
 linear form *)
(* same color map as plot 18 *)
shdgrd = redhot /@ gval/Max[gval] ;
```

As before, this linear list needs to be partitioned into a square array.

```
(* fold the directives into a list with the same shape as plst *)
pshdgrd = Partition[shdgrd, na];
```

We will not plot the figure. Notice that we are using the list of function values `plst` from plot 21 and the plot parameters from plot 17.

```
(* plst computed for plot 21 *)
(* use plot parameters from plot 17 *)
(* plot the figure and shade using
  the x component of the gradient *)
g1 = ListPlot3D[plst, pshdgrd, params];
```



Gradient

Plot 24. This is a complement to the preceding plot. We will shade this figure using the *y* value of the gradient and the same `redhot` color map.

We create a function that gives us the *y* value of the gradient.

```
(* compute the y component of the gradient function *)
grady[x_, y_] = Simplify[∂y f[x, y]]
```

$$\frac{21}{8}\, x\, (-3 + 5\, x^2)\, (-1 + 5\, y^2)$$

We apply this function to the address list.

```
(* compute the y component of the gradient at the sample points *)
gval = grady[#[[1]], #[[2]]] & /@ faddr;
```

We need to normalize the values, so we need to find the extrema.

```
(* we need to normalize the color scale *)
{Max[gval], Min[gval]}
```

```
{21, -21}
```

The normalized values are passed to the function `redhot`.

```
(* shading directives for the y component of the gradient values,
 linear form *)
(* same color map as plot 15 *)
shdgrd = redhot /@ gval/Max[gval] ;
```

After partitioning the data:

```
(* fold the directives into a list with the same shape as plst *)
pshdgrd = Partition[shdgrd, na];
```

we are ready to plot the figure with the shading.

```
(* plst computed for plot 21 *)
(* use plot parameters from plot 17 *)
(* plot the figure and shade using
  the y component of the gradient *)
g1 = ListPlot3D[plst, pshdgrd, params];
```

Gradient

Plot 25. This page is devoted to parametric plots in three dimensions. We will see that `ParametericPlot3D` can plot either a curve or a surface. The first two figures are curves.

This is a simple helix. Rendering the curve in black leads to a confusing drawing so we applied a simple coloration scheme. While we have faulted `Hue` for color wrapping problems, in this case we do not care. The natural variation allows the eye to follow the curve as sketched.

```
(* plot a helix *)
g1 = ParametricPlot3D[{Cos[8 t], Sin[8 t], t, Hue[t]}, {t, 0, π/2}];
```

Helix

Plot 26. Notice how we have simply changed the order of the arguments to generate this helix. We also wanted to demonstrate a `GrayLevel` shading scheme. We have normalized the function values so that the shading value varies from one (white) and the start and gradually goes to zero (black) at the terminus. We hope you agree this aids the eye in following the curve.

```
(* plot a helix *)
g1 = ParametricPlot3D[
   {Cos[8 t], t, Sin[8 t], GrayLevel[1 - 2 t/π]}, {t, 0, π/2}];
```

Helix

Plot 27. We will begin a look at parametric plots using the spherical coordinates. As you will see in the citation at the end of this section, the relationship between the Cartesian parameters x, y, and z and the spherical polar parameters r, θ, and ϕ is according to equations 4-6:

$$x = r\cos\theta\sin\phi \tag{A2.4}$$

$$y = r\sin\theta\sin\phi \tag{A2.5}$$

$$z = r\cos\phi \tag{A2.6}$$

On this page we shall deal with spheres and their generalizations, the ellipsoids.

```
(* plot a sphere *)
g1 = ParametricPlot3D[{Cos[θ] Sin[ϕ], Sin[θ] Sin[ϕ], Cos[ϕ]},
   {θ, 0, 2 π}, {ϕ, 0, π}, SphericalRegion → True];
```



Spherical Coordinates

Plot 28. Now looking ahead to the following examples, we want to show how to make a sphere look like an ellipsoid by adjusting the `BoxRatios`.

```
(* parameters for the ensuing ellipsoids *)
{a, b, c} = {3, 2, 1};
```

Notice that we are not drawing a sphere: we are taken a sphere that is already drawn and stretching the box it is in.

```
(* use BoxRatios to make the sphere look like the ellipsoid *)
g2 = Show[g1, BoxRatios → {a, b, c}];
```

Plot 29. Now we will plot an ellipsoid but stretch the bounding box so that the figure looks like a sphere. In the reference cited below, we learn that an ellipsoid is a quadratic surface defined as

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1\,. \tag{A2.7}$$

In spherical coordinates the definition becomes

$$\frac{r^2\cos^2\theta\sin^2\phi}{a^2} + \frac{r^2\sin^2\theta\sin^2\phi}{b^2} + \frac{r^2\cos^2\phi}{c^2} = 1\,. \tag{A2.8}$$

The parametric equations are then

$$x = r\cos\theta\sin\phi \tag{A2.9}$$

$$y = r\sin\theta\sin\phi \tag{A2.10}$$

$$z = r\cos\phi \tag{A2.11}$$

where

$$0 \le \theta \le 2\pi \quad \text{and} \quad 0 \le \phi \le \pi \quad . \tag{A2.12}$$

These equations are passed directly to the plot command.

```
(* specify an ellipsoid *)
{a, b, c} = {3, 2, 1};
(* plot an ellipsoid, change BoxRatios *)
g1 = ParametricPlot3D[{a Cos[θ] Sin[ϕ], b Sin[θ] Sin[ϕ], c Cos[ϕ]},
   {θ, 0, 2 π}, {ϕ, 0, π}, SphericalRegion → True, BoxRatios → {1, 1, 1}];
```



Ellipsoid

Plot 30. Now we will plot the ellipsoid with the default box ratios and we will find the ellipsoid that we expected.

```
(* specify an ellipsoid *)
{a, b, c} = {3, 2, 1};
(* plot an ellipsoid *)
g1 = ParametricPlot3D[{a Cos[θ] Sin[ϕ], b Sin[θ] Sin[ϕ], c Cos[ϕ]},
   {θ, 0, 2 π}, {ϕ, 0, π}, SphericalRegion → True];
```

When we look at plots 27-30, we see two sphere and two ellipsoids. The arrangement of the figures shows that we can make spheres look like ellipsoids and ellipsoids look like spheres. How can we tell them apart? Just look at the axis.

Look at figure 28, the squashed sphere. The x axis varies from {-1,1) as do the y and z axes. Yet these axes of the same dimension have different lengths on the plot, a sure sign of distortion.

Similarly, when you look at figure 29 you will see that the plot lengths of the axes are the same yet the dimensions are very different: x varies from {-3,3}, y from {-2,2} and z from {-1,1}. This figure too has been distorted.



Ellipsoid

Plot 31. The ability to shade a figure using the basic parameters is extremely useful. Here we will show how to color our ellipsoid according to the divergence. In the case of a three-dimensional Cartesian space the divergence of a scalar function ϕ is given in equation.

$$\nabla\phi = \phi_x + \phi_y + \phi_z \tag{A2.13}$$

which, in *Mathematica*, takes the form

```
(* divergence operator for 3D Cartesian space *)
div3c[ϕ_] := D[ϕ, x] + D[ϕ, y] + D[ϕ, z];
```

Our scalar function comes from the functional definition of the ellipsoid in Cartesian coordinates as described in reference 7. We apply the divergence function in *Mathematica*.

```
(* compute the divergence function for the ellipse *)
fcn = div3c[x^2/a^2 + y^2/b^2 + z^2/c^2 - 1]
```

$$\frac{2\,x}{a^2} + \frac{2\,y}{b^2} + \frac{2\,z}{c^2}$$

Now we will convert this function to spherical coordinates.

```
(* convert the divergence function over to spherical coordinates *)
sfcn[θ_, ϕ_] :=
 FullSimplify[fcn /. x → Cos[θ] Sin[ϕ] /. y → Sin[θ] Sin[ϕ] /. z → Cos[ϕ]]
```

Out of curiosity we examine this functional form.

```
(* what does this function look like? *)
sfcn[θ, ϕ]
```

$$2\left(\frac{\text{Cos}[\phi]}{c^2} + \left(\frac{\text{Cos}[\theta]}{a^2} + \frac{\text{Sin}[\theta]}{b^2}\right)\text{Sin}[\phi]\right)$$

As in the other examples (37-40), we will need to normalize the source function so that it is restricted to the domain (-1,1). To do that, we need to find the extrema of this function and that requires us to specify an ellipsoid.

```
(* specify an ellipsoid *)
{a, b, c} = {3, 2, 1};
```

```
(* find the extrema to normalize the divergence function *)
FindMaximum[sfcn[θ, ϕ], {θ, 0}, {ϕ, 0}]
FindMinimum[sfcn[θ, ϕ], {θ, 0}, {ϕ, 0}]

{2.0735, {θ → 1.15257, ϕ → 0.267045}}
```

```
{-2.0735, {θ → 1.15257, ϕ → -9.15773}}
```

We capture the normalization factor in the variable `normfac`.

```
(* harvest the normalization *)
normfac = %%[[1]]

2.0735
```

The function we will be using to generate the scaled inputs for the shading directive is

```
(* normalized divergence function *)
ndiv[θ_, ϕ_] := normfac^-1 (2 (Cos[ϕ]/c^2 + (Cos[θ]/a^2 + Sin[θ]/b^2) Sin[ϕ]))
```

This is the same redhot color map that we used on the plots of the previous page.

```
(* hue color map that doesn't wrap *)
redhot = Hue[(1 - #) 0.8] &;
```

The syntax used for coloring surface graphics is not documented very well in the Help Browser. We will be using:

```
{SurfaceColor[redhot[ndiv[θ, ϕ]]], EdgeForm[]}
```

which colors the surface and turns off the mesh with the `EdgeForm[]` call.

You will also have noticed by now that we cannot pass plot parameters in a variable which leads to the rather cluttered call

```
(* plot an ellipsoid, shade by support function *)
g1 = ParametricPlot3D[{a Cos[θ] Sin[ϕ], b Sin[θ] Sin[ϕ],
    c Cos[ϕ], {SurfaceColor[redhot[ndiv[θ, ϕ]]], EdgeForm[]}},
   {θ, 0, 2 π}, {ϕ, 0, π}, SphericalRegion → True,
   PlotPoints → {50, 50}, AmbientLight → GrayLevel[0.45]];
```

Divergence, Ellipsoid

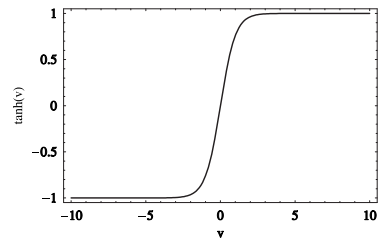
Plot 32. Closing our look at the `ParametricPlot3D` command, we will turn to the Mercator parameterization of the ellipsoid using trigonometric and hyperbolic functions. According to the reference, equations 15-17, the Mercator parameterization of the ellipsoid is given by

$$x(u,\ v) = a\,\mathrm{sech}v\cos u \tag{A2.14}$$

$$y(u,\ v) = b\,\mathrm{sech}v\sin u \tag{A2.15}$$

$$z(u,\ v) = c\tanh v \tag{A2.16}$$

You will find that the top of the figure is hard to close.You will see that there is a small opening in the top of the ellipsoid. This is due to the asymptotic behavior of the hyperbolic tangent which we plot below.



The curve approaches the asymptotic values of ±1almost reluctantly. It appears the figure in the reference used the domain {-1,1} for *v*, leaving the small hole in the top. We used the *v* domain {-5,5} which closed the hole, but at the default setting of `PlotPoints` the edge is less rounded.

```
(* specify an ellipsoid *)
{a, b, c} = {3, 2, 1};
(* plot an ellipsoid using the Mercator parametrization *)
g1 = ParametricPlot3D[{a Sech[v] Cos[u], b Sech[v] Sin[u], c Tanh[v]},
   {v, -5, 5}, {u, 0, 2 π}, SphericalRegion → True];
```

Ellipsoid

A2.3 Add-on package: Graphics

As we will see, there currently is a single add-on package that has commands to generate three-dimensional graphics objects. This certainly simplifies our classification efforts.

A2.3.1 Graphics'ContourPlot3D'

The three-dimensional extension to the contour plotting capability may seem straightforward. However, we will see that there is some subtlety involved since it is usually difficult to view more than one three-dimensional contour. On the other hand, this is not a great limitation; in fact shall see that we can render shapes without knowing the parametric representation by plotting a zero contour value.

Plot 33. We will show how to plot an ellipsoid without having to know the parameterization. The key is to realize that the equation for an ellipsoid in Cartesian coordinates,

$$\frac{x^2}{a^2}+\frac{y^2}{b^2}+\frac{z^2}{c^2} = 1 \qquad \text{A2.7}$$

can be written as $f(x, y, z) = 0$ allowing us to plot the ellipsoid surface as the zero contour value.

We start by loading the package.

```
(* load the package *)
<< Graphics`ContourPlot3D`
```

Now we specify the same ellipsoid that we have been using

```
(* specify an ellipsoid *)
{a, b, c} = {3, 2, 1};
(* plot the 0 contour: this is the surface *)
g1 = ContourPlot3D[x^2/a^2 + y^2/b^2 + z^2/c^2 - 1, {x, -a, a},
   {y, -b, b}, {z, -c, c}, Axes → True, SphericalRegion → True];
```

The alert reader probably suspects that we do not have to create a plot function such that $f(x, y, z) = 0$; we should be able to use the equation more directly. And in fact we can. We can exactly duplicate this chart by plotting the contour of value one using

```
(* specify an ellipsoid *)
{a, b, c} = {3, 2, 1};
(* plot the 0 contour: this is the surface *)
g1 = ContourPlot3D[x^2/a^2 + y^2/b^2 + z^2/c^2, {x, -a, a}, {y, -b, b},
   {z, -c, c}, Axes → True, SphericalRegion → True, Contours → {1}];
```

⚠ You must pass a list to the `Contours` option. Remember, a list with one element is different from an integer.

💡 Ellipsoid

Plot 34 This feature is so valuable that we want to demonstrate it one more time. Here we will use the function that was plotted in table A2.5.

Of course we need to load the package first.

```
(* load the package *)
<< Graphics`ContourPlot3D`
```

Then we can define the function.

```
(* define a 2D function *)
f[x_, y_] := 7/8 (9 x y - 15 x^3 y - 15 x y^3 + 25 x^3 y^3)
```

The only subtlety here is going from a 2D function of x and y to a 3D function of x, y, and z. However, this is cured by casting a plot function of the form $f(x, y, z) = 0$. So if we write the equality:

$$z(x, y) = f(x, y) \tag{A2.17}$$

then the plot function:

$$z(x, y) - f(x, y) = 0 \tag{A2.18}$$

is apparent.

```
g1 = ContourPlot3D[z - f[x, y],
   {x, -1, 1}, {y, -1, 1}, {z, -2, 2}, PlotPoints → {5, 5}];
```

Plot 35. For the remainder of the plots in the table, we will be looking at four positive charges placed at the vertices of the unit square in the plane $z = 0$. We will probe the high-field and weak-field equipotentials and will even be able to simultaneously display equipotentials from both regimes.

We proceed with the obligatory package load.

```
(* load the package *)
<< Graphics`ContourPlot3D`
```

The we locate the positive charges at the vertices of the unit square.

```
(* place 4 postive charges on the corners of the unit square *)
p = {{0, 0, 0}, {1, 0, 0}, {1, 1, 0}, {0, 1, 0}};
np = Length[p];
```

Now we can create a function to compute the field value at any location due to the four charges.

```
(* function to compute the field values *)
ϕ = Sum[1 / ((x - p[[i]][[1]])^2 + (y - p[[i]][[2]])^2 + (z - p[[i]][[3]])^2), {i, np}];
```

First, we will look at a high-field value of 10 which is found only in the immediate region of the charges.

```
(* plot the surface where the field value is 10 *)
g1 = ContourPlot3D[ϕ, {x, -1, 2}, {y, -1, 2}, {z, -1, 1},
   Contours → {10}, PlotPoints → {7, 7}, SphericalRegion → True];
```

The observant reader will realize that the equipotential surfaces are not ideal spheres. We can see a nipple-like structure point to nearest neighbors.

Plot 36. Proceeding from the previous plot, we now want to find a transition surface: this is the surface with the smallest field value that unites all four charges.

Of course, we need to load the package first.

```
(* load the package *)
<< Graphics`ContourPlot3D`
```

As before, we locate the four positive charges:

```
(* place 4 postive charges on the corners of the unit square *)
p = {{0, 0, 0}, {1, 0, 0}, {1, 1, 0}, {0, 1, 0}};
np = Length[p];
```

and write a function to compute the value of the field at any point.

```
(* function to compute the field values *)
ϕ = Sum[1 / ((x - p[[i]][[1]])^2 + (y - p[[i]][[2]])^2 + (z - p[[i]][[3]])^2), {i, np}];
```

We are looking for the equipotential with the smallest value that covers all four charges. This value is:

```
(* calibration point where the fields merge *)
ϕ /. x → 1/2 /. y → 1/2 /. z → 0

8
```

So we plot the contour for a field strength of eight.

```
(* plot the surface where the field value is 8 *)
g1 = ContourPlot3D[ϕ, {x, -1, 2}, {y, -1, 2}, {z, -1, 1},
   Contours → {8}, PlotPoints → {7, 7}, SphericalRegion → True];
```

Plot 37. We continue to explore even weaker field values which correspond to longer distances. We will see that the equipotential surface begins to mask the structure of the charge distribution.

As usual, we start by loading the package.

```
(* load the package *)
<< Graphics`ContourPlot3D`
```

Then we position the four charges at the vertices of the unit square.

```
(* place 4 postive charges on the corners of the unit square *)
p = {{0, 0, 0}, {1, 0, 0}, {1, 1, 0}, {0, 1, 0}};
np = Length[p];
```

Then we cast a function that can compute the field strength at any point.

```
(* function to compute the field values *)
ϕ = Sum[1/((x - p[[i]][[1]])^2 + (y - p[[i]][[2]])^2 + (z - p[[i]][[3]])^2), {i, np}];
```

The contour for field value five is created like this.

```
(* plot the surface where the field value is 5 *)
g1 = ContourPlot3D[ϕ, {x, -1, 2}, {y, -1, 2}, {z, -1, 1},
   Contours → {5}, PlotPoints → {7, 7}, SphericalRegion → True];
```

Plot 38. This is that last of four plots looking at four charges in the continuum. We will be able to plot two equipotential surfaces here if we choose them judiciously.

We begin by loading the package.

```
(* load the package *)
<< Graphics`ContourPlot3D`
```

We place the positive charges as in the previous three plots:

```
(* place 4 postive charges on the corners of the unit square *)
p = {{0, 0, 0}, {1, 0, 0}, {1, 1, 0}, {0, 1, 0}};
np = Length[p];
```

and we create the same equation that describes the field strength at any point.

```
(* function to compute the field values *)
ϕ = Sum[1 / ((x - p[[i]][[1]])^2 + (y - p[[i]][[2]])^2 + (z - p[[i]][[3]])^2), {i, np}];
```

For this example, we will show two equipotential surfaces and we will want to color them to increase contrast. We established some color definitions in the header when we loaded the notebook `CRC common.m`. The colors we will use are these.

```
(* color definitions from CRC common.m *)
cblu
cred

RGBColor[0, 0, 1]
```

```
RGBColor[1, 0, 0]
```

The field values that we will plot are 1 and 12. The low field value one is not contained by the bounding box and we will see only fragments of it. The high field value of 12 will take the guise of spheres around each of the charges.

```
(* plot the surface where the field values are 1 and 12 *)
g1 = ContourPlot3D[ϕ, {x, -1, 2}, {y, -1, 2}, {z, -1, 1},
   Contours → {1, 12}, ContourStyle → {{cblu}, {cred}},
   Lighting → False, PlotPoints → {7, 7}, SphericalRegion → True];
```

Plot 39. For the final two plots in the table we will look at the discrete form of the three-dimensional contour plotting package `ListContourPlot3D`. As usual, when we switch to a list plot, the axes no longer measure physical distances or units; instead they count points. We will also see the orientation change.

To start, we must load the package.

```
(* load the package *)
<< Graphics`ContourPlot3D`
```

We will use the same charge distribution:

```
(* place 4 postive charges on the corners of the unit square *)
p = {{0, 0, 0}, {1, 0, 0}, {1, 1, 0}, {0, 1, 0}};
np = Length[p];
```

and potential function as in the preceding four plots:

```
(* function to compute the field values *)
ϕ = Sum[1 / ((x - p[[i]][[1]])^2 + (y - p[[i]][[2]])^2 + (z - p[[i]][[3]])^2), {i, np}];
```

The difference comes when we discretely sample the field by creating a table of field values. Here we will use a coarser sample than in the following plot:

```
(* distance between sample points *)
δ = 1/10;
(* sample the field values *)
fieldval = Table[ϕ, {x, -1, 2, δ}, {y, -1, 2, δ}, {z, -1.5, 1.5, δ}];
```

Mathematica will examine these field values and draw the equipotentials for use.

```
(* plot two different contours and shade them *)
g1 = ListContourPlot3D[fieldval,
   Contours → {1, 9}, MeshRange → {{-1, 1}, {-1, 1}},
   Lighting → False, ContourStyle → {{cblu}, {cred}},
   SphericalRegion → True, Axes → True];
```

Plot 40. Now we want to increase the resolution from the previous plot because we want to look at higher values for the field.

We start by loading the package.

```
(* load the package *)
<< Graphics`ContourPlot3D`
```

All plots in this table has the same charge distribution

```
(* place 4 postive charges on the corners of the unit square *)
p = {{0, 0, 0}, {1, 0, 0}, {1, 1, 0}, {0, 1, 0}};
np = Length[p];
```

and therefore the same function describing the field values at arbitrary points.

```
(* function to compute the field values *)
ϕ = Sum[1/((x - p[[i]][[1]])^2 + (y - p[[i]][[2]])^2 + (z - p[[i]][[3]])^2), {i, np}];
```

Here is where we will improve the resolution by shrinking the distance between sample points.

```
(* distance between sample points *)
δ = 1/25;
(* sample the field values *)
fieldval = Table[ϕ, {x, -1, 2, δ}, {y, -1, 2, δ}, {z, -1.5, 1.5, δ}];
```

This 3D array is passed directly to the plot routine. Notice the change in orientation between this plot and plot 38.

```
(* plot two different contours and shade them *)
g1 = ListContourPlot3D[fieldval,
   Contours → {0.75, 12}, MeshRange → {{-1, 1}, {-1, 1}},
   Lighting → False, ContourStyle → {{cblu}, {cred}},
   SphericalRegion → True, Axes → True];
```

A2.4 Graphics‘Graphics3D‘

This package is another catchall with various and diverse functions. We will explore these functions individually.

Plot 41. As we mentioned in chapter 3, three-dimensional graphics are at times hard to read. To mitigate this problem, *Mathematica* provides a `Shadow` routine that will project shadows onto the walls helping us to more quickly visualize the shape. Unfortunately, we will see that our careful shading is corrupted in the shadows.

We start by loading the package.

```
(* load the package *)
<< Graphics`Graphics3D`
```

If you are trying to reproduce this figure you will first have to regenerate the figure displayed in plot 31. We will start with this figure and then use the `Shadow` routine to cast shadows from it.

```
(* g1 is the graphics object from plot 31 *)
(* project shaded shadows onto the walls *)
g2 =
  Shadow[g1, AmbientLight → GrayLevel[0.6], SphericalRegion → True];
```

⚠ The `Shadow` routine does not always project shading schemes correctly.

Plot 42. We do not wish to discourage the reader because of the problems with projecting the shading. This is a valuable tool and we want to show an example that works as desired.

We need to load the package first.

```
(* load the package *)
<< Graphics`Graphics3D`
```

We are going to recreate the spherical harmonic we studied in chapter 3. We will use the same module to create the plot function from the square of the spherical harmonic. Except this time, because we have the shadow projection, we can study a more foliated function since we will be able to see how the sheaves are shaped.

```
(* module to generate the squared spherical harmonic function *)
pgen[l_Integer, m_Integer] := Module[{g},
    g = SphericalHarmonicY[l, m, θ, φ];
    sh = Simplify[g Conjugate[g], θ ∈ Reals ∧ φ ∈ Reals];
   ];
 (* specify the eigenstate *)
 {l, m} = {7, 2};
 (* call the generation module *)
 pgen[l, m]
 (* display the squared function *)
 sh
```

$$\frac{315\,\text{Cos}[\theta]^2\,(15 - 110\,\text{Cos}[\theta]^2 + 143\,\text{Cos}[\theta]^4)^2\,\text{Sin}[\theta]^4}{4096\,\pi}$$

As before, we will change the light source to something we find a bit more appealing. We will bundle all these options into a single list `opts` allowing us to maintain a very simple plot syntax.

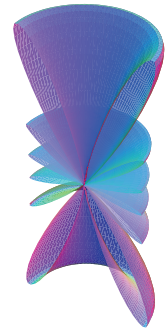
```
(* define a different light source to enliven the figure *)
ls = LightSources →
   {{{0, 0, 1}, RGBColor[0, 0, 1]}, {{1, 0, 0.4}, RGBColor[1, 0, 0]},
    {{0, 1, 0.4}, RGBColor[0, 1, 0]}};
(* collect our plot parameters in a list *)
opts = {Axes → False, Boxed → False,
   PlotPoints → {150, 50}, Mesh → False, SphericalRegion → True, ls};
(* display the square of the spherical harmonic function *)
g1 = SphericalPlot3D[{sh, EdgeForm[]}, {θ, 0, π}, {φ, 0, 2 π}, opts];
```

We will use this plot, graphics object `g1`, in the `Shadow` plot command. We will also use the same light source to use the same coloration.

```
(* project shadows onto the walls *)
g2 = Shadow[g1, ls, SphericalRegion → True];
```

The inquisitive reader may wish to compare how the shadows reveal structure to a cutaway shown here, created by restricting the domain of ϕ.

```
(* cutaway plot shows the structure of the sheaves *)
g1 = SphericalPlot3D[{sh, EdgeForm[]}, {θ, 0, π}, {φ, 0, π}, opts];
```



Plot 43. A related plot command is `ShadowPlot3D`. This will render the figure in three-dimensions and project a contour-like shadow plot either above or below the bounding box. We'll show the default plot first; then we'll adjust the plot parameters to improve the appearance.

But first, we need to load the package.

```
(* load the package *)
<< Graphics`Graphics3D`
```

Then we define the sinc function which we have been relying upon quite often.

```
(* define the sampling function *)
sinc[x_ /; x ≠ 0] := Sin[x] / x ;
sinc[x_ /; x == 0] := 1;
```

We'll use the same plot function as numerous other plots in this book.

```
(* plot function *)
fcn[x_, y_] := sinc[x]^2 sinc[1.2 y]^2
```

This is the default plot. It is a bit coarse and we can't see the shadow.

```
(* default plot *)
g1 = ShadowPlot3D[fcn[x, y],
   {x, -3 π, 3 π}, {y, -3 π, 3 π}, SphericalRegion → True];
```

Plot 44. Let's improve the appearance of the previous plot. Let's keep everything the exact same until we get to the plot command. In lieu of the old plot command we'll substitute the color map that doesn't wrap:

```
(* the color function that does not wrap *)
redhot := Hue[(1 - #) 0.8] &;
```

and a new plot command, rich with settings.

```
(* a more pleasing plot *)
g1 =
  ShadowPlot3D[fcn[x, y], {x, -3 π, 3 π}, {y, -3 π, 3 π}, PlotPoints → 200,
   SurfaceMesh → False, ShadowMesh → False, ShadowPosition → 1,
   Axes → True, ColorFunction → redhot, SphericalRegion → True];
```

We have increased the resolution by increasing `PlotPoints`. This drives us to turn the two meshes off: the surface mesh on the three-dimensional figure and the shadow mesh. We will use `ShadowPosition` to move the shadow to the top of the figure. Notice that we have to turn `Axes` on. In most plots, `Axes` defaults to `True`. The color map is the same we have used dozens of times by now.

Plot 45. The next four charts will use the discrete form of the `ShadowPlot3D` command. This may seem to be overzealous, but we want to make two points. First, the discrete command works the same as the continuum command with the usual caveat that the discrete axes now count points instead of marking some physical distance. Second, this command is an extremely powerful data visualization tool.

To begin, we load the package.

```
(* load the package *)
<< Graphics`Graphics3D`
```

Then we define the sinc function.

```
(* define the sampling function *)
sinc[x_ /; x ≠ 0] := Sin[x] / x ;
sinc[x_ /; x == 0] := 1;
```

We use the sinc function to build a plot function.

```
(* plot function *)
fcn[x_, y_] := sinc[x]^2 sinc[1.2 y]^2
```

Now we need to create a rectangular table of function values. We have done this many times before in the text.

```
(* coarse sample size *)
δ = π / 3 ;
(* discrete sample values *)
z = Table[fcn[x, y], {x, -3 π, 3 π, δ}, {y, -3 π, 3 π, δ}] // N;
```

Before we plot the height values in the array `z`, we want to use the color map that doesn't wrap. We rely upon the same color map we have used dozens of times by now.

```
(* the color function that does not wrap *)
redhot := Hue[(1 - #) 0.8] &;
```

Now we are ready to plot the data. We will find the display to be rather coarse.

```
(* shadow plot the coarse data *)
g1 = ListShadowPlot3D[z, SurfaceMesh → False,
   ShadowMesh → True, ShadowPosition → 1, Axes → True,
   ColorFunction → redhot, SphericalRegion → True];
```

Plot 46. We want to repeat the previous exercise, except this time at a much higher resolution.

We first load the package.

```
(* load the package *)
<< Graphics`Graphics3D`
```

Then we define the sinc function

```
(* define the sampling function *)
sinc[x_ /; x ≠ 0] := Sin[x] / x ;
sinc[x_ /; x == 0] := 1;
```

which is used to build the plot function.

```
(* plot function *)
fcn[x_, y_] := sinc[x]^2 sinc[1.2 y]^2
```

We will discretely sample the plot function with a sample density that is 10 times higher than the previous example. (This will generate 100 times more points.)

```
(* fine sample size *)
δ = π / 30 ;
(* discrete sample values *)
z = Table[fcn[x, y], {x, -3 π, 3 π, δ}, {y, -3 π, 3 π, δ}] // N;
```

We will, of course, use the same color map.

```
(* the color function that does not wrap *)
redhot := Hue[(1 - #) 0.8] &;
```

Finally, we present the plot. Compare this plot to plot 44.

```
(* shadow plot the discrete points *)
g1 = ListShadowPlot3D[z, SurfaceMesh → False,
   ShadowMesh → False, ShadowPosition → 1, Axes → True,
   ColorFunction → redhot, SphericalRegion → True];
```

Plot 47. We have demonstrated the expected result that we can make the discrete plots look like the continuum plots by simply driving the sample density higher. There is the usual tell-tale trace, however, of axes with different scales and a transposition.

One story not yet told is what a great tool this command is for visualizing data. Real data have finite errors associated with them. We will demonstrate how strongly the signal survives in a high noise environment.

We start with the obligatory package load.

```
(* load the package *)
<< Graphics`Graphics3D`
```

Once more we define the sinc function.

```
(* define the sampling function *)
sinc[x_ /; x ≠ 0] := Sin[x] / x;
sinc[x_ /; x == 0] := 1;
```

The sinc function is used to create the plot function `fcn`.

```
(* plot function *)
fcn[x_, y_] := sinc[x]^2 sinc[1.2 y]^2
```

This time when we sample the data, we will introduce some random noise. In this case, the noise will be 10% of the signal. Note that we have seeded the random number sequence. This will allow you to exactly replicate these results.

```
(* fine sample size *)
δ = π/3;
(* seed for repeatability *)
SeedRandom[1];
(* noise *)
ϵ = 0.1;
(* discrete sample values with noise: measurements *)
nz = Table[fcn[x, y] (1 + ϵ RandomReal[{-1, 1}]),
    {x, -3π, 3π, δ}, {y, -3π, 3π, δ}] // N;
```

The array `nz` contains the noisy data, the measurement surrogate. We will also create the ideal signal for comparison later.

```
(* ideal function values *)
z = Table[fcn[x, y], {x, -3π, 3π, δ}, {y, -3π, 3π, δ}] // N;
```

This is the common color map for all of these plots.

```
(* the color function that does not wrap *)
redhot := Hue[(1 - #) 0.8] &;
```

Now we are ready to plot the noisy data.

```
(* shadow plot the discrete points *)
g1 = ListShadowPlot3D[nz, SurfaceMesh → False,
   ShadowMesh → True, ShadowPosition → 1, Axes → True,
   ColorFunction → redhot, SphericalRegion → True];
```

These data are virtually indistinguishable from the pure signal in plot 45. Let's quantify the differences now.

We start by picking the middle row. Picking a single row allows us to compare data in one dimension. What we want is a difference vector which computes the difference between the signal and the measurement.

```
(* difference between ideal and measured: this is the noise *)
diff = z[[10]] - nz[[10]]

{-0.000239859, 0.000125101, 0.0000505524, 0.00107762,
 3.25637×10^-35, -0.00164128, -0.0014477, 0.00184026, 0.0404135,
 -0.00479161, 0.0565682, -0.000267032, 0.00221527, 0.000612281,
 2.17626×10^-35, 0.00108673, 0.000443597, 0.000175955, 0.0000283661}
```

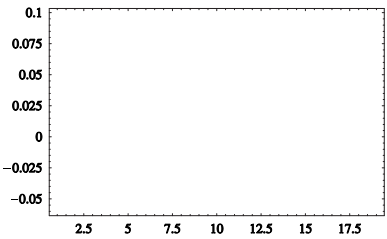
These numbers are a bit hard to decipher, so we will compute them to a relative percentage of the signal. Notice how *Mathematica*'s vector division makes this easy.

```
(* relative error *)
err = diff / z[[10]]

{-0.0339194, 0.036595, 0.0113219, 0.0677292,
 0.0214309, -0.0458469, -0.059553, 0.033645, 0.070556,
 -0.00479161, 0.0987596, -0.00488208, 0.091128, 0.0171032,
 0.0143224, 0.0683016, 0.0993497, 0.0514713, 0.00401136}
```

These percentages are more meaningful, and a plot will really tell the story.

```
(* plot the relative errors *)
g1 = ListPlot[err, PlotStyle → cblu,
   PlotRange → All, Axes → False, Frame → True];
```



We are looking at some rather large errors of up to 10% in the data, yet the plot representations between ideal and measured are virtually identical.

Plot 48. We want to repeat the previous exercise and this time we will more than double the error and see if this erodes our ability to see the side lobes.

As before, we first load the package.

```
(* load the package *)
<< Graphics`Graphics3D`
```

Our plot function is based upon the sinc function.

```
(* define the sampling function *)
sinc[x_ /; x ≠ 0] := Sin[x] / x ;
sinc[x_ /; x == 0] := 1;
```

The plot function is separable in Cartesian coordinates.

```
(* plot function *)
fcn[x_, y_] := sinc[x]^2 sinc[1.2 y]^2
```

As in the previous example, we will discretely sample the plot function and add a random amount of noise. The noisy data simulating a measurement is in the list `nz`.

```
(* fine sample size *)
δ = π / 30 ;
(* seed for repeatability *)
SeedRandom[1];
(* noise *)
ϵ = 0.25;
(* discrete sample values with noise *)
nz = Table[fcn[x, y] (1 + ϵ Random[Real, {-1, 1}]),
    {x, -3 π, 3 π, δ}, {y, -3 π, 3 π, δ}] // N;
```

When we want to measure the noise in each point, we will need to know the exact value of the input function.

```
(* ideal function values *)
z = Table[fcn[x, y], {x, -3 π, 3 π, δ}, {y, -3 π, 3 π, δ}] // N;
```

We continue to use the `redhot` color map.

```
(* the color function that does not wrap *)
redhot := Hue[(1 - #) 0.8] &;
```

When we plot the data, we see that the plot is virtually indistinguishable from the ideal data displayed in plot 46.

```
(* shadow plot the discrete points *)
g1 = ListShadowPlot3D[nz, SurfaceMesh → False,
   ShadowMesh → False, ShadowPosition → 1, Axes → True,
   ColorFunction → redhot, SphericalRegion → True];
```

Once again we want to precisely measure the amount of noise present in the signal and we will use the middle row as an example. First, we compute the difference between the ideal value and the surrogate measurement.

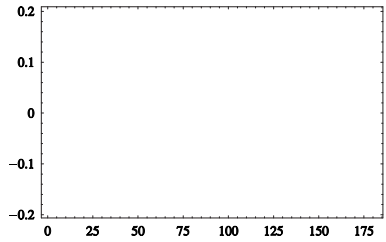
```
(* difference between ideal and measured: this is the noise *)
diff = z⟦10⟧ - nz⟦10⟧;
```

Then we convert these errors in to percentages.

```
(* relative error *)
err = diff / z⟦10⟧;
```

When we plot these values, we see some breath-stealing errors over 20%. Yet the plot remains faithful to the signal.

```
(* plot the relative errors *)
g1 = ListPlot[err, PlotStyle → cblu,
   PlotRange → All, Axes → False, Frame → True];
```



Plot 49. The next two plots will examine the `ListScatterPlot3D` command which is very useful. We will show a basic plot and then a strategy for shading the individual points.

We begin with a package load of course.

```
(* load the package *)
<< Graphics`Graphics3D`
```

These points represent a descent trajectory where the speed and circle radius are gradually reduced.

```
(* helical descent and slowing *)
pts = Table[{t Cos[8 t], t Sin[8 t], t}, {t, 0, 1.5, 0.005}];
```

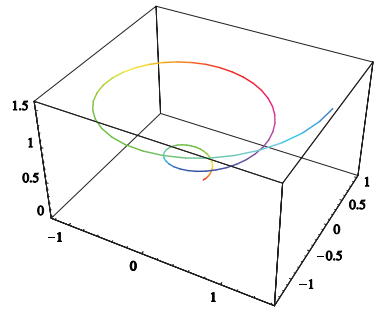
The plot syntax is quite simple. We are using the `SphericalRegion` option to force our plots to be the same size.

```
(* plot the points *)
g1 = ScatterPlot3D[pts, SphericalRegion → True];
```

Plot 50. This is easy to use, but we are limited in our coloring options. The problem is that the `PlotStyle` option applies to all of the points. In other words, we can

change the color for all of them. We would really like to have a discrete version of the plot generated here.

```
(* helical descent and slowing with color variation *)
g1 = ParametricPlot3D[{t Cos[8 t], t Sin[8 t], t, Hue[t]},
   {t, 0, π/2}, SphericalRegion → True];
```



So how can we get around this syntax limitation? The answer is simple. We create a series of plots, each plot having a single point. When we plot a single point, we can use the `PlotStyle` to provide a unique color directive for that point. This is all trivial with *Mathematica*'s list-based architecture.

As usual, we first load the package.

```
(* load the package *)
<< Graphics`Graphics3D`
```

Now we will create a table of plots. Each plot shall have but one point. We have to be careful here to use the single point, which is `ScatterPlot3D[{pt}]`. Also, since we don't want to see a series of hundreds of single point plots, we will suppress the display.

```
(* create a series of plots with just one point *)
(* syntax for plotting a single point: ScatterPlot3D[{pt}] *)
plots = Table[ScatterPlot3D[{{t Cos[8 t], t Sin[8 t], t}},
    PlotStyle → Hue[t], doff], {t, 0, 1.5, 0.005}];
```

We now have a table of `Graphics3D` objects. The `Show` command will automatically unite them into a single plot. We have to be careful to turn the display on since the graphics objects all have the display turned off.

```
(* combine all the plots *)
g1 = Show[plots, don, SphericalRegion → True];
```

Plot 51. These next two plots use the `StackGraphics` command to combine two-dimensional graphics. This is an interesting alternative to putting all of the curves on one plot.

For this example, we will plot the trajectories of a cannonball shot out of a cannon at a 45° to ensure maximum range. We will vary the energy of the propellant in two Joule increments.

The fine print is as follows: all of the propellant energy is converted into kinetic energy. There is no atmosphere and Earth is not spinning. These corrections would have a negligible effect upon the result and a deleterious effect on the transparency of the calculation.

The basic strategy is to build a table of plots where each plot is a single trajectory. Notice that in this format we could apply an arbitrary shading.

```
(* MKS units: cannonball is 1 kg *)
{g, m} = {9.8, 1};
b = 1;
(* create a series of plots for vacuum trajectories *)
(* energies vary from 2 to 20 J in 2 J increments *)
plots = Table[a = -g m / (2 ε);
   Plot[a x^2 + b x, {x, 0, -b/a}, Frame → True, DisplayFunction → voff]
   , {ε, 2, 20, 2}];
```

Now we are ready to present the data. First, we load the package.

```
(* load the package *)
<< Graphics`Graphics3D`
```

The plot syntax is quite simple.

```
(* show all the trajectories together *)
g1 = Show[StackGraphics[plots], SphericalRegion → True];
```

Plot 52. Let's revisit the same parity plots of the Legendre plots we showed in section A1.4. Recall how we combined all of those curves onto a single `Plot`. Here we will use the `StackGraphics` option to combine them.

We will create a table of plots and each plot displays a single polynomial. Notice how this allows us to color the individual curves.

```
(* odd-parity Legendre functions *)
plotso = Table[
   Plot[LegendreP[n, x], {x, -1, 1},
    PlotStyle → Hue[n/10], DisplayFunction → voff]
   , {n, 9, 1, -2}];
```

Then we load the package:

```
(* load the package *)
<< Graphics`Graphics3D`
```

and display them with `StackGraphics`.

```
(* look at the odd-parity Legendre functions *)
g1 = Show[StackGraphics[plotso], SphericalRegion → True];
```

Plot 53. For this plot we were a little aggressive. We wanted to show how the `ListSurfacePlot3D` command works and we chose a spindle torus for the test. As we see from our ever ready reference (equations 2-4), the general parameterization for a torus is given by:

$$x = (c + a\cos v)\cos u \tag{A2.19}$$

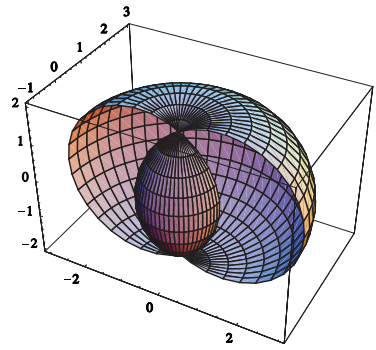
$$y = (c + a\cos v)\sin u \tag{A2.20}$$

$$x = a\sin v \tag{A2.21}$$

where the parameters u and v are bound by the domain $[0, 2\pi)$. (We have corrected an obvious typo in the reference equation 3.)

When $c < a$, we have a spindle torus. Let's show a cutaway view of one by restricting the domain of the parameter u.

```
(* spindle torus *)
{a, c} = {2, 1};
(* plot the torus *)
g1 = ParametricPlot3D[
   {(c + a Cos[v]) Cos[u], (c + a Cos[v]) Sin[u], a Sin[v]},
   {u, 0, π}, {v, 0, 2 π}, SphericalRegion → True];
```



Certainly, this looks like a stressful case for a routine which will use the points to create vertices in a polygon mesh. After all, we have two distinct surfaces here. Let's see how well we do.

We will create the list of points first. This requires us encode equations 19-21 above.

```
(* parameterization *)
x := (c + a Cos[v]) Cos[u]
y := (c + a Cos[v]) Sin[u]
z := a Sin[v]
```

Now we need to specify the same torus.

```
(* spindle torus *)
{a, c} = {2, 1};
```

We are now able to build a list of sample points.

```
(* sample spacing *)
δ = π/20;
(* sample the torus *)
pts = Table[{x, y, z}
   , {u, 0, π, δ}, {v, 0, 2π, δ}];
```

We are now ready to plot, so we need to load the package.

```
(* load the package *)
<< Graphics`Graphics3D`
```

The plot syntax is quite simple.

```
(* load the package *)
<< Graphics`Graphics3D`
```

The results are amazing. Compare the plot immediately above with plot 53.



Torus

Watch out for the typo in equation 3, p. 2,991 in the previous reference.

Plot 54. The **Project** command allows us to take our graphics to the carnival House of Mirrors and change (distort) the figures in an arbitrary manner.

First, we load the package.

```
(* load the package *)
<< Graphics`Graphics3D`
```

We will use the figure to generate for plot 41.

```
(* g1 is the graphics object g1 from plot 42 *)
```

For this demonstration, we will change the basis vectors for the projection plane.

```
(* we will change the basis vectors of the projection plane *)
g2 = Show[Project[g1, {{1/2, 1/2, 0}, {0, 0, 1}}, {0, 1, 1}]];
```

Plot 55. The ability to create three-dimensional bar charts can be handy. Here we want to show how to use this command to visualize matrices.

We begin by loading the package.

```
(* load the package *)
<< Graphics`Graphics3D`
```

The we grab the vexatious matrix we worked with in section 2.2. We will use the `BarChart3D` command to display this matrix. Some change in the viewpoint is needed.

```
(* this is the matrix we started with in section 2.2 *)

A = {{1, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0},
     {0, 2, 0, 0, -1, 0, -1, 0, 0, 0, 0, 0},
     {0, 0, 2, 0, 0, -1, 0, -1, 0, 0, 0, 0},
     {0, 0, 0, 1, 0, 0, -1, 0, 0, 0, 0, 0},
     {0, -1, 0, 0, 2, 0, 0, 0, 0, -1, 0, 0},
     {-1, 0, -1, 0, 0, 4, 0, 0, -1, 0, -1, 0},
     {0, -1, 0, -1, 0, 0, 4, 0, 0, -1, 0, -1},
     {0, 0, -1, 0, 0, 0, 0, 2, 0, 0, -1, 0},
     {0, 0, 0, 0, 0, -1, 0, 0, 1, 0, 0, 0},
     {0, 0, 0, 0, -1, 0, -1, 0, 0, 2, 0, 0},
     {0, 0, 0, 0, 0, -1, 0, -1, 0, 0, 2, 0},
     {0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 1}};
```

In order to be able to see the negative terms, we have rotated the image.

```
(* present the matrix as a bar chart *)
(* rotate the image so the negatives are visible *)
g1 = BarChart3D[A,
   ViewPoint → {-1.579, -3.559, -1.211}, SphericalRegion → True];
```

Plot 56. Complementing the three-dimensional bar chart is a three-dimensional histogram. Here too we will use this command in an different manner. Let's go back to the points shown in plot 49. We will make a histogram of the x and y values for the points in that list.

This drives us to some brief list gymnastics. We need to go from a list in the format $\{x, y, z\}$ to a list in the form $\{x, y\}$. This is fairly simple. We can project out the x values into a list using the dot product. A similar operation works to create a y list. These lists are then merged and the resultant list transposed giving us the data and the format that we needed. (This process was detailed at the end of section 2.2.)

First, we create the points used in plot 49.

```
(* helical descent and slowing *)
pts = Table[{t Cos[8 t], t Sin[8 t], t}, {t, 0, 1.5, 0.005}];
```

Clearly we could have manufactured the list in the proper format, but we wanted to answer the question of how to analyze the data used in that plot. Proceeding, we create our $\{x, y\}$ list.

```
(* extract the {x,y} coordinates *)
(* x coordinates *)
x = pts.{1, 0, 0};
(* y coordinates *)
y = pts.{0, 1, 0};
(* merge the x and y lists *)
z = {x, y};
(* reshape the list *)
data = Transpose[z];
```

This list is summarily plotted.

```
(* histogram of {x,y} positions for the points in plot 49 *)
g1 = Histogram3D[data, SphericalRegion → True];
```

A2.5 Graphics`ParametricPlot3D`

We believe that there is a nomenclature conflict. There is a `ParametricPlot3D` command in the kernel which means it is available as soon as *Mathematica* is launched. Then there is a package `Graphics`ParametricPlot3D`. Certainly the former is a package and the latter is a command, so they are different types of things. It is just that the types of plots they produce are so similar. We can only hope that one day Wolfram Research can change one of the names.

Of course we can expect that the Help Browser has a succinct explanation:

> This package extends `ParametricPlot3D` by providing an alternative to the `PlotPoints` option where the sampling may be specified by giving a step size in each coordinate.

Pressing forth, we shall examine the different kinds of plots this package can produce.

Plot 57. Remember the problematic Mercator parameterization in plot 32? Specifically the problem was that by the time we closed the hole in the top of the figure the ellipsoid was no longer smooth, having taken an angular appearance. This is precisely the type of reason that you would use the extended form of the `ParametricPlot3D` command.

We first load the package.

```
(* load the package *)
<< Graphics`ParametricPlot3D`
```

Then as before, we specify an ellipse and plot the figure using the Mercator projection. This time however you will note the subtle change in the plot syntax: we have specified the increments in the plot parameters. We have drawn boxes around these increments.

```
(* specify an ellipsoid *)
{a, b, c} = {3, 2, 1};
(* plot an ellipsoid using the Mercator parametrization *)
g1 = ParametricPlot3D[
   {a Sech[v] Cos[u], b Sech[v] Sin[u], c Tanh[v]}, {v, -5, 5, 1/10},
   {u, 0, 2 π, π/15}, Shading → False, SphericalRegion → True];
```

Compare this plot to 32.

Plot 58. One oversight in the documentation for these commands was that there were no examples where the edge lines were removed and the figure was colored. This is the problem that made us wish we could search the entire *Mathematica Book* to find example.

Here we will demonstrate how to turn off the edge lines and apply a color scheme. We have drawn a box around the list that makes these changes.

Of course, to duplicate this plot, you need to load the package

```
(* load the package *)
<< Graphics`ParametricPlot3D`
```

You'll notice that we essentially recycled the code from the previous example, making only the change that we have outlined.

```
(* specify an ellipsoid *)
{a, b, c} = {3, 2, 1};
(* plot an ellipsoid using the Mercator parametrization *)
g1 = ParametricPlot3D[{a Sech[v] Cos[u], b Sech[v] Sin[u],
    c Tanh[v], {Hue[u/(2 π)], EdgeForm[]}}, {v, -5, 5, 1/10},
   {u, 0, 2 π, π/100}, Lighting → False, SphericalRegion → True];
```

Plot 59. We will now present two examples using `PointParametricPlot3D`, the discrete version of the continuum command we have explored. We will replicate the braking trajectory presented in plot 49.

First, we load the package

```
(* load the package *)
<< Graphics`ParametricPlot3D`
```

Then we pass the parametric equations and a sampling recipe.

```
(* plot the helical slowing descent *)
g1 = PointParametricPlot3D[{t Cos[8 t], t Sin[8 t], t},
   {t, 0, 3/2, 5/1000}, SphericalRegion → True];
```

The plots are the same. Compare the code with the commands used in plot 49:

```
(* helical descent and slowing *)
pts = Table[{t Cos[8 t], t Sin[8 t], t}, {t, 0, 1.5, 0.005}];
```

```
(* plot the points *)
g1 = ScatterPlot3D[pts, SphericalRegion → True];
```

Plot 60. In this plot we will return to the horn torus; this is the case where the two characteristic radii are the same.

We start by loading the package.

```
(* load the package *)
<< Graphics`ParametricPlot3D`
```

Then following the parameterization in the cited reference we plot the sample points.

```
(* horn torus *)
{a, c} = {1, 1};
(* plot the torus *)
g1 = PointParametricPlot3D[{(c + a Cos[v]) Cos[u],
    (c + a Cos[v]) Sin[u], a Sin[v]}, {u, 0, 2 π, π/16},
   {v, 0, π, π/20}, Boxed → False, SphericalRegion → True];
```

Torus

Plot 61. The next three examples will demonstrate the `SphericalPlot3D` command.

We start by loading the package.

```
(* load the package *)
<< Graphics`ParametricPlot3D`
```

Notice how trivial it is to plot a sphere: we plot the constant radius.

```
(* plot the unit sphere *)
g1 = SphericalPlot3D[1, {θ, 0, π}, {ϕ, 0, 2 π},
   ColorOutput → GrayLevel, SphericalRegion → True];
```

Typically, the plot function (here the constant one) is some function of the parameters θ and ϕ. We will show more general cases next.

Plot 62. The horn torus from plot 60 is back. We will create this version using `SphericalPlot3D`.

First, we load the package.

```
(* load the package *)
<< Graphics`ParametricPlot3D`
```

Then we plot the function. We truncated the domain of ϕ to provide the reader with a cutaway view.

```
(* the horn torus *)
g1 = SphericalPlot3D[Sin[θ], {θ, 0, π}, {ϕ, 0, π},
   ColorOutput → GrayLevel, SphericalRegion → True];
```

Notice that the horn tours is created by two different parameterizations: $\sin\theta$ in `SphericalPlot3D` and equations A2.19-21 with $a = c$ in `ParametricPlot3D`.

Plot 63. In our final look at `SphericalPlot3D` we will show how to combine three-dimensional graphics objects.

First we load the package.

```
(* load the package *)
<< Graphics`ParametricPlot3D`
```

Then we will build a table of `Graphics3D` objects. Notice that we have turned the display off while building the table as we do not want our screen to fill with so many images.

```
(* create a series of concentric spheres in cutaway *)
g1 = Table[
   SphericalPlot3D[{r, EdgeForm[]}, {θ, 0, π}, {ϕ, 0, π}, ColorOutput →
     GrayLevel, SphericalRegion → True, DisplayFunction → voff]
   , {r, 1, 5, 1}];
```

We will use the `Show` command to combine the graphics and bring them to life. Since these figures all had the display turned off the final graphics object `g2` will also have that property, forcing us to turn the display on in the last step.

```
(* display all the shells *)
g2 = Show[g1, don];
```

Plot 64. We will close this subsection by creating a cone with `CylindricalPlot3D`. In these coordinates, the cone has a simple description. For this example we choose $1 - r$.

First, of course, we must load the package.

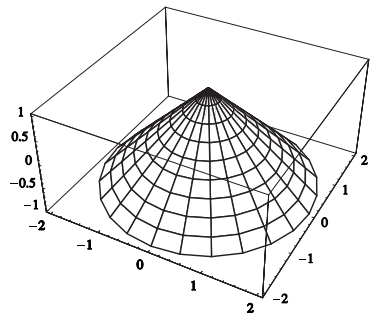
```
(* load the package *)
<< Graphics`ParametricPlot3D`
```

Look at the simplicity of the plot syntax. (Ignore the optional parameter settings.) This is a very simple description.

```
(* plot a cone *)
g1 = CylindricalPlot3D[1 - r, {r, 0, 2}, {ϕ, 0, 2 π},
   PlotPoints → {10, 25}, Shading → False, SphericalRegion → True];
```

Observe we can create an identical figure with `ParametricPlot3D`:

```
(* specify the height and the radius *)
h = 2; r = 2;
(* plot the cone *)
g1 = ParametricPlot3D[
   {(h - z)/h r Cos[θ], (h - z)/h r Sin[θ], z - 1}, {z, 0, h}, {θ, 0, 2 π},
   PlotPoints → {10, 25}, Shading → False, SphericalRegion → True];
```



The moral of this story is that the proper choice of coordinate systems can greatly simplify plotting.

A2.5.1 Graphics`PlotField3D`

This package is the natural extension of the two-dimensional plot commands we saw in Appendix 1, plots 81-88. We will see that the visualization in three-dimensions is a bit more complicated.

Plot 65. As you might expect, this command will plot vector fields of three dimensions. In our first example however, we will force all the vectors to lie within planes of constant z.

First we load the package.

```
(* load the package *)
<< Graphics`PlotField3D`
```

The we define the vector field.

```
(* define a force with planar orientation *)
f2 = {y, -x, 0} / z^2 ;
```

Then we plot the field.

```
(* plot the field vectors *)
g1 = PlotVectorField3D[f2, {x, -1, 1}, {y, -1, 1},
   {z, 1, 3}, VectorHeads → True, SphericalRegion → True];
```

Compare this figure to plot 81 which used this vector field.

```
(* vector force field *)
f1 = {-y, x} / Sqrt[x^2 + y^2] ;
```

Plot 66. Here we will extend the potential used in Appendix 1, plot 82 to three dimensions.

Before we start we will load the package.

```
(* load the package *)
<< Graphics`PlotField3D`
```

Here is the obvious extension to the scalar potential.

```
(* define a potential function *)
ϕ = x^2 + y^2 + z^2;
```

In the first appendix, we had two dimensions and we just built a gradient function manually. In this case, we would like to build a gradient function which handles arbitrary dimensions. This is useful not only for higher dimensions, but also for extracts of higher dimensions.

The basic process is to define a list of your dimensions here called ζ. As usual we will count these dimensions rather than hard-coding the number. This is one less thing to debug if we change the dimensions. Now our gradient becomes a list object assembled with the `Table` command.

```
(* list the spatial variables *)
ζ = {x, y, z};
(* count the dimensions *)
nζ = Length[ζ];
(* build a gradient function *)
field = Table[D[ϕ, ζ[[i]]], {i, nζ}];
```

In this case we build the scalar field into the gradient definition `field`. The plot syntax is then

```
(* plot the field vectors *)
g1 = PlotVectorField3D[field, {x, 0, 1}, {y, 0, 1},
   {z, 0, 1}, VectorHeads → True, SphericalRegion → True];
```

Plot 67. For the next case we want to present a more interesting potential.

But first we load the package.

```
(* load the package *)
<< Graphics`PlotField3D`
```

We will just add some terms to the potential used in the previous plot.

```
(* define a more complicated force *)
ϕ = x^2 + y^2 + z^2 - 2 ((x + 1/2)^2 + (y - 1/2)^2 + z^2);
```

Here we will ask *Mathematica* to compute the gradient for us.

```
(* plot the field vectors *)
g1 = PlotGradientField3D[ϕ, {x, -1, 1}, {y, -1, 1},
   {z, 0, 1}, VectorHeads → True, SphericalRegion → True];
```

Plot 68. Now we want to return to the potential in plot 66. In that plot we computed the gradient and plotted the vectors. Now we will ask *Mathematica* to compute the gradient for us. Compare this plot to the one above it.

Loading the package is our first step.

```
(* load the package *)
<< Graphics`PlotField3D`
```

Then we rely upon the plot command to do the work for us.

```
(* plot the gradient vectors *)
g1 = PlotGradientField3D[ϕ, {x, 0, 1}, {y, 0, 1},
   {z, 0, 1}, VectorHeads → True, SphericalRegion → True];
```

Plot 69. In this example we will show a separable potential and we will do some shading.

We begin by loading the package.

```
(* load the package *)
<< Graphics`PlotField3D`
```

We will use the `redhot` color map as in so many previous examples.

```
(* color map as a pure function *)
redhot := Hue[(1 - #) 0.8] &;
```

The scalar potential is separable.

```
(* define a force with planar orientation *)
ϕ = x y z;
```

We can now plot the color shaded gradient field.

```
(* plot the field vectors *)
g1 = PlotGradientField3D[ϕ,
   {x, -1, 1}, {y, -1, 1}, {z, -1, 1}, VectorHeads → True,
   ColorFunction → redhot, PlotPoints → 5, SphericalRegion → True];
```

Plot 70. Let's return to the potential used in plot 66. This time we will adjust the plot domains and vary the gray shading.

We load the package.

```
(* load the package *)
<< Graphics`PlotField3D`
```

Then we define the scalar potential.

```
(* define a potential function *)
ϕ = x^2 + y^2 + z^2;
```

We leave computation of the gradient to *Mathematica*.

```
(* plot the field vectors *)
g1 = PlotGradientField3D[ϕ, {x, 0, 1},
   {y, 0, 1}, {z, 0, 1}, PlotPoints → 5, VectorHeads → True,
   ColorFunction → GrayLevel, SphericalRegion → True];
```

Plot 71. We will present two examples of the discrete version of `PlotVectorField3D`. The first plot will explore a gas of noninteracting particles.

We start by loading the package.

```
(* load the package *)
<< Graphics`PlotField3D`
```

Now we create a random gas. Here we use 200 particles. Notice how we have avoided a syntax like:

```
(* elephantine formulation *)
vel = Table[{Random[Real, 0.075 {-1, 1}],
    Random[Real, 0.075 {-1, 1}], Random[Real, 0.075 {-1, 1}]}, {i, np}];
```

Also, notice that we are using a `GrayLevel` shading.

```
(* define a force with planar orientation *)
(* set the number of particles in our gas *)
np = 200;
(* seeding allows reader to verify the results *)
SeedRandom[1];
(* position function *)
x := Random[]
(* velocity function *)
v := Random[Real, 0.075 {-1, 1}]
(* particle positions *)
pos = Table[{x, x, x}, {i, np}];
(* particle velocities *)
vel = Table[{v, v, v}, {i, np}];
```

The curious reader may wonder why we created data in this format since we cannot use these lists directly in the plot command. We think that your code should match the physics of the problem. So we separate out the velocities which are temperature dependent.

We do this because it is so simple to put the data into the desired format.

```
(* simple to put into needed format *)
data = Transpose[{pos, vel}];
(* plot the field vectors *)
g1 = ListPlotVectorField3D[data, ColorFunction → GrayLevel,
   VectorHeads → True, SphericalRegion → True];
```

Plot 72. We return to the potential of problems 66, 68, and 70. This plot will be a discrete version. If you look at the plot figures, you will notice that these four plots form the right-hand column of the page.

We load the package.

```
(* load the package *)
<< Graphics`PlotField3D`
```

We define the same potential.

```
(* define a potential function *)
ϕ = x^2 + y^2 + z^2;
```

We steal the gradient function from plot 66.

```
(* list the spatial variables *)
ξ = {x, y, z};
(* count the dimensions *)
nξ = Length[ξ];
(* build a gradient function *)
field = Table[D[ϕ, ξ[[i]]], {i, nξ}];
```

Then we build a table of sample points. Here the samples are 0.25 units apart. The list of sample values is `data1`.

```
(* global samples: look at entire domain *)
(* distance between samples *)
δ = 0.25;
(* build a list in the format {pos, field} *)
pts = Table[
   {{x, y, z}, field}
   , {x, 0, 1, δ}, {y, 0, 1, δ}, {z, 0, 1, δ}];
(* put the data in a linear array *)
data1 = Flatten[pts, 2];
```

Instead of plotting these points we want to zoom in on a specific region. We will sample this region at higher density and then combine the high and low resolution data for a single plot.

```
(* local samples: zoom in on a region of interest *)
(* distance between samples *)
δ = 0.05;
(* build a list in the format {pos, field} *)
pts = Table[
   {{x, y, z}, field}
   , {x, 0, 0.2, δ}, {y, 0, 0.2, δ}, {z, 0, 0.2, δ}];
(* put the data in a linear array *)
data2 = Flatten[pts, 2];
```

The data at finer resolution is `data2` and we need to join that to `data1`.

```
(* combine the two data lists *)
data = Join[data1, data2];
```

We can now plot both sets at once.

```
(* plot the field vectors *)
g1 = ListPlotVectorField3D[data, VectorHeads → True,
   ScaleFactor → 0.25, SphericalRegion → True];
```

A2.5.2 Graphics`Polyhedra`

While these package may seem esoteric to many users, their output is probably the most recognizable symbol of *Mathematica*: the *Mathematica* spikey. This will be the first of eight samples drawn from this package.

The ten solids covered in this package are these:

Tetrahedron	Cube
Octahedron	Dodecahedron
Icosahedron	Hexahedron
GreatDodecahedron	SmallStellatedDodecahedron
GreatStellatedDodecahedron	GreatIcosahedron



Platonic Solid, Uniform Polyhedron

Plot 73. Let's draw the *Mathematica* spikey.

We start by loading the package.

```
(* load the package *)
<< Graphics`Polyhedra`
```

This step is not necessary, but it is convenient. We collected all the recognized solids into a single list. This enables us to do things like loop through the solids and it keeps us from having to continually type these long names.

```
(* put all the shapes into a list *)
lst = {Cube, Tetrahedron, Octahedron, Icosahedron,
   GreatDodecahedron, GreatStellatedDodecahedron, Dodecahedron,
   Hexahedron, SmallStellatedDodecahedron, GreatIcosahedron};
(* count the special shapes *)
nl = Length[lst];
```

Don't forget that a negative list index counts from the end backwards.

```
(* show the spikey *)
lst[[-2]]

SmallStellatedDodecahedron
```

Here we will separate the steps to clarify the process. The `Polyhedron` command creates a graphics object, not as plot.

```
(* create a graphics object *)
g1 = Polyhedron[lst[[-2]]];
(* Show graphics object *)
g2 = Show[g1, Axes → True];
```

We could have combined the steps in this manner.

```
(* equivalent formulation 1 *)
g2 = Show[Polyhedron[lst[[-2]]], Axes → True];
```

And we could have skipped the `lst` reference and typed in the name.

```
(* equivalent formulation 2 *)
g2 = Show[Polyhedron[SmallStellatedDodecahedron], Axes → True];
```

Plot 74. What is the difference between an icosahedron and a great icosahedron? We load the package first.

```
(* load the package *)
<< Graphics`Polyhedra`
```

Again, the list is not necessary; it is just nice.

```
(* put all the shapes into a list *)
lst = {Cube, Tetrahedron, Octahedron, Icosahedron,
   GreatDodecahedron, GreatStellatedDodecahedron, Dodecahedron,
   Hexahedron, SmallStellatedDodecahedron, GreatIcosahedron};
(* count the special shapes *)
nl = Length[lst];
```

Here we show positive (forward) and negative (backward) referencing.

```
(* select two icosahedra *)
lst[[4]]
lst[[-1]]

Icosahedron
```

```
GreatIcosahedron
```

We'll create two graphics objects and combine them with show.

```
(* create graphics objects *)
g1 = Polyhedron[lst[[4]], {0, 0, 0}];
g2 = Polyhedron[lst[[-1]], {3, 0, 0}];
(* Show graphics object *)
gg = Show[g1, g2, SphericalRegion → True];
```

We could have used a list to hold the graphics objects.

```
(* equivalent yet more cumbersome formulation *)
g1 = {Polyhedron[lst[[4]], {0, 0, 0}], Polyhedron[lst[[-1]], {3, 0, 0}]};
gg = Show[g1, SphericalRegion → True];
```

Here is the gumbo version: everything is thrown into one pot (or command).

```
(* equivalent and even more cumbersome formulation *)
gg = Show[{Polyhedron[lst[[4]], {0, 0, 0}],
    Polyhedron[lst[[-1]], {3, 0, 0}]}, SphericalRegion → True];
```

When you try to read this command you'll see why we eschew such compact code writing.

```
(* equivalent and really cumbersome formulation *)
gg = Show[{Polyhedron[Icosahedron, {0, 0, 0}],
    Polyhedron[GreatIcosahedron, {3, 0, 0}]}, SphericalRegion → True];
```



Icosahedron

Plot 75. We notice that we have four flavors of dodecahedron. It would be nice to view them all together in one plot.

We load the package to start.

```
(* load the package *)
<< Graphics`Polyhedra`
```

We show the list of recognized solids one last time.

```
(* put all the shapes into a list *)
lst = {Cube, Tetrahedron, Octahedron, Icosahedron,
   GreatDodecahedron, GreatStellatedDodecahedron, Dodecahedron,
   Hexahedron, SmallStellatedDodecahedron, GreatIcosahedron};
(* count the special shapes *)
nl = Length[lst];
```

These are the four dodecahedra.

```
(* compare the different flavors of dodecahedron *)
lst[[7]]
lst[[5]]
lst[[9]]
lst[[6]]
```

```
Dodecahedron
```

```
GreatDodecahedron
```

```
SmallStellatedDodecahedron
```

```
GreatStellatedDodecahedron
```

We follow the previous prescription by creating a graphics object for each dodecahedron. We use `Show` to combine them.

```
(* dodecahedra *)
g1 = Polyhedron[lst[[7]], {0, 0, λ}];
g2 = Polyhedron[lst[[5]], {λ, 0, λ}];
g3 = Polyhedron[lst[[9]], {0, 0, 0}];
g4 = Polyhedron[lst[[6]], {λ, 0, 0}];
(* display them all *)
gg = Show[g1, g2, g3, g4, SphericalRegion → True];
```

The engaged reader may ask if there isn't some way to use *Mathematica*'s list-based tools. Here is one such implementation.

```
(* equivalent formulation *)
(* select the polyhedra from the list *)
elem = {7, 5, 9, 6};
(* count the elements *)
ne = Length[elem];
(* spacing parameter *)
λ = 5;
(* positions for each figure *)
where = {{0, 0, λ}, {λ, 0, λ}, {0, 0, 0}, {λ, 0, 0}};
(* equivalent formulation *)
ge = Table[
   Polyhedron[lst[[elem[[i]]]], where[[i]]]
   , {i, ne}];
(* display them all *)
gg = Show[ge, SphericalRegion → True];
```



Dodecahedron

Plot 76. The next two plots explore stellation. The first shows positive stellation, the second negative stellation. However, we discovered a bug in the package and we will see that stellation and translation don't mix. This precludes us from having stellated graphics anywhere but at the origin. To circumvent this problem we will use the `GraphicsArray` command.



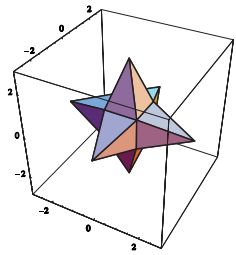
Stellated objects must be plotted at the origin.

We load the package to begin.

```
(* load the package *)
<< Graphics`Polyhedra`
```

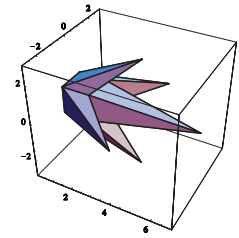
First we will show a stellated cube, then we will show what happens when we translate that cube.

```
(* source object *)
g0 = Polyhedron[Cube];
(* this is a stellated cube *)
g1 = Show[Stellate[g0, 4], Axes → True, SphericalRegion → True];
```



Look what happens when we translate the stellated figure.

```
(* create the cube in the desired position *)
g1 = Polyhedron[Cube, {1, 0, 0}];
(* stellate the cube *)
g2 = Show[Stellate[g1, 4], Axes → True, SphericalRegion → True];
```



It appears as though the cube has translated but the points of stellation have not. Because of this we cannot combine stellated figures into one plot object like `gg` in the previous plot. This is not a great hardship; we point it out so that the new user would not look for a syntax error.

The plan is as follows. We create a graphics object `g0` which is the basic cube. We will stellate this object in a table.

```
(* source object *)
g0 = Polyhedron[Cube, Boxed→ False];
(* create a table of cubes with increasing stellation *)
g1 = Table[
   Stellate[g0, i]
   , {i, 0, 8}];
```

We have to repack the array. We have a linear array and we want a two-dimensional square array. The `Partition` command handles this nicely.

```
(* go from a 1 × 9 array to a 3 × 3 array *)
g2 = Partition[g1, 3];
(* display this array *)
g3 = Show[GraphicsArray[g2], SphericalRegion→ True];
```



Stellation

Plot 77. The previous plot displayed the cases of positive stellation. We now wish to explore negative stellation. We refer the reader to the caution sounded in the last example which showed us that we can't move a stellated object from the origin.

As always, we first load the package.

```
(* load the package *)
<< Graphics`Polyhedra`
```

We create a base graphic object `g0` for the cube. Next, we apply negative stellation indices to the base object.

```
(* source object *)
g0 = Polyhedron[Cube, Boxed→ False];
(* create a table of cubes with increasing stellation *)
g1 = Table[
   Stellate[g0, -i]
   , {i, 0, 8}];
```

Then we fold the linear array in a square matrix and display the graphics array.

```
(* go from a 1 × 9 array to a 3 × 3 array *)
g2 = Partition[g1, 3];
(* display this array *)
g3 = Show[GraphicsArray[g2], SphericalRegion → True];
```

Stellation

Plot 78. The next command is the `Geodesate` command. As described in the Help Browser this command displays the projection of the order *n* regular tessellation of each face of the polyhedron onto the circumscribed sphere.

We start by loading the package.

```
(* load the package *)
<< Graphics`Polyhedra`
```

We will start with a basic cube. With this basic object, we will create a table of graphics object created by a tesselation of order *i*.

```
(* source object *)
g0 = Polyhedron[Cube, Boxed → False];
(* n regular tessalation of each face projected onto a sphere *)
g1 = Table[
   Geodesate[g0, i]
   , {i, 9}];
```

We will continue using the **GraphicsArray** command to display the images.

```
(* go from a 1 × 9 array to a 3 × 3 array *)
g2 = Partition[g1, 3];
(* display this array *)
g3 = Show[GraphicsArray[g2], SphericalRegion → True];
```

Geodesic Dome

Plot 79. The final two plots in this series deal with truncation. The first series will present the default truncation, the second plot presents figures using the open truncation.

We load the package.

```
(* load the package *)
<< Graphics`Polyhedra`
```

We will stick with the established pattern. We will create a graphics object `g0` which is the basic cube. We will then use the `Table` command to create a list of graphics objects, each representing a successively higher degree of truncation. The highest truncation parameter is one-half.

```
(* source object *)
g0 = Polyhedron[Cube, Boxed→ False];
(* truncate the source object *)
g1 = Table[
   Truncate[g0, i]
   , {i, 0, 0.5, 0.0625}];
```

We use `GraphicsArray` to display the results.

```
(* go from a 1 × 9 array to a 3 × 3 array *)
g2 = Partition[g1, 3];
(* display this array *)
g3 = Show[GraphicsArray[g2], SphericalRegion→ True];
```

Plot 80. In the previous plot we looked at truncation of the cube. Now we will look at open truncation of the cube. The difference between truncation and open truncation is quite clear once you compare the figures.

Load the package.

```
(* load the package *)
<< Graphics`Polyhedra`
```

Create the base graphics object `g0` which will be passed to the `OpenTruncate` command.

```
(* source object *)
g0 = Polyhedron[Cube, Boxed→ False];
(* truncate the source object *)
g1 = Table[
   OpenTruncate[g0, i]
   , {i, 0, 0.5, 0.0625}];
```

We rely upon `GraphicsArray` to display the figures.

```
(* go from a 1 × 9 array to a 3 × 3 array *)
g2 = Partition[g1, 3];
(* display this array *)
g3 = Show[GraphicsArray[g2], SphericalRegion→ True];
```



Truncated Cube

A2.5.3 Graphics`Shapes`

Mathematica has a package to draw and manipulate these six basic shapes.

Cylinder	Sphere
Cone	Helix
Torus	Double Helix

We will plot some of these figures and demonstrate the manipulation commands.

Plot 81. In the previous plot we looked at truncation of the cube. Now we will look at open truncation of the cube. The difference between truncation and open truncation is quite clear once you compare the figures.

Load the package.

```
(* load the package *)
<< Graphics`Shapes`
```

This plot command allows us to collect our plot parameters into a list which makes the plot commands much easier to read. This also helps us to standardize the appearance of our plots.

```
(* common plot parameters *)
param =
  {Boxed → False, ColorOutput → GrayLevel, SphericalRegion → True};
```

In the cited reference we see there are three different types of torus. The most common form looks like a donut and is called a ring torus. If we shrink the donut hole to zero we have a horn torus. And if we keep shrinking the internal radius we get a spindle torus.

We will build a table of tori and you will be able to see the transition from ring through horn to spindle. We will build a table of graphics objects.

```
(* create a table of graphics, each graphic a torus *)
g1 = Table[
   r = (i + 1)/3;
   (* torus: radii = {2, r} *)
   Graphics3D[Torus[2, r, 20, 30], param]
   , {i, 9}];
(* pack into 2 D array *)
g2 = Partition[g1, 3];
```

We display these graphics objects using the `GraphicsArray` command.

```
(* display tori in transition *)
g3 = Show[GraphicsArray[g2], SphericalRegion → True];
```

The horn torus is in the middle of the figure: second row, second column. The table starts in the upper left-hand corner and goes across and then down. Every torus before the horn torus is a ring torus; every figure after the horn torus is a spindle torus.



Torus

Plot 82. We can quickly plot a Möbius strip.

Load the package first.

```
(* load the package *)
<< Graphics`Shapes`
```

If you have already defined your plot parameters, you may skip this step.

```
(* common plot parameters *)
param = {Boxed → False, ColorOutput → GrayLevel};
```

Then specify the parameters of the Möbius strip.

```
(* create a Moebius strip graphics object *)
g1 = Graphics3D[MoebiusStrip[2, 0.5, 50], param];
(* show the Moebius strip *)
g2 = Show[g1, SphericalRegion → True];
```



Möbius Strip

Plot 83. Next we will look at the `Helix`. This is another of the rare cases where the *CRC Concise Encyclopedia of Mathematics* differs from *Mathematica*. The *CRC Concise Encyclopedia of Mathematics* distinguishes between the wire like helix and the helicoid which is a form of surface. A helix is addressed by one parameter and a helicoid is addressed by two. *Mathematica* calls both objects a helix. Intrigued readers are directed to the entries for Helix and Helicoid.

Load the package.

```
(* load the package *)
<< Graphics`Shapes`
```

We have consolidated our favorite plot parameter settings.

```
(* common plot parameters *)
param =
  {Boxed → False, ColorOutput → GrayLevel, SphericalRegion → True};
```

We create the `Graphics3D` object and then use `Show` to display it. These steps can be combined.

```
(* create a helix graphics object *)
g1 = Graphics3D[Helix[2, 1, 3, 30], param];
(* show the helix *)
g2 = Show[g1];
```



Helicoid

Plot 84. This is an example of a double helix. We refer the reader to the discussion in the previous entry for the helix.

Load the package.

```
(* load the package *)
<< Graphics`Shapes`
```

If you have already defined these parameters you do not need to repeat this step.

```
(* common plot parameters *)
param =
  {Boxed → False, ColorOutput → GrayLevel, SphericalRegion → True};
```

Again we have used two steps in the interest of clarity. These commands can be merged into a single line.

```
(* create a double helix graphics object *)
g1 = Graphics3D[DoubleHelix[1, 1, 3, 30], param];
(* show the double helix *)
g2 = Show[g1];
```



Helicoid

Plot 85. Now we would like to demonstrate the image manipulation tools in this package. The `AffineShape` command will be applied to a sphere.

Load the package to begin.

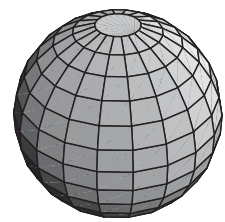
```
(* load the package *)
<< Graphics`Shapes`
```

If you have already defined your plot parameters you do not need to repeat this step.

```
(* common plot parameters *)
param =
  {Boxed → False, ColorOutput → GrayLevel, SphericalRegion → True};
```

We want to show you what kind of a sphere we get out of this command.

```
(* draw the default sphere *)
g1 = Graphics3D[Sphere[], param];
(* show the sphere *)
g2 = Show[g1];
```



Notice the opening in the top of the sphere. This is the source of the opening in plot 85; it is not an artifact of the `AffineShape` command.

Now we will apply the `AffineShape` command and feed it the parameters for the ellipsoids we saw in plot 30 and beyond.

```
(* squish the sphere into an ellipse *)
g2 = AffineShape[g1, {3, 2, 1}];
(* show the ellipsoid *)
g3 = Show[g2];
```



Affine Transformation

Plot 86. In our final plot we will show how to apply the `WireFrame` command. As expected, this will reduce a plot to a series of polygons without shading or color.

Load the package.

```
(* load the package *)
<< Graphics`Shapes`
```

Here are the plot parameters that we will use.

```
(* common plot parameters *)
param = {Boxed → False, ColorOutput → GrayLevel};
```

We will use the same sphere from the previous example. Notice that we apply the `WireFrame` command to the `Graphics3D` object.

```
(* draw the default sphere *)
g1 = Graphics3D[Sphere[], param];
(* show the sphere *)
g2 = Show[WireFrame[g1], SphericalRegion → True];
```

A2.5.4 Graphics`SurfaceOfRevolution`

The package for studying surfaces of revolution is quite simple conceptually. You specify a curve and a revolution axis. We will look at Gabriel's Horn, a special case generated by rotating the curve $y(x) = 1/x$ about the x axis ($x \geq 1$). As pointed out in the reference this curve has the unusual property that it has finite volume and infinite surface area. The analog is that you can fill it with paint, but there can never be enough paint to cover the outside.

Plot 87. For our first encounter we will present a cutaway view.

Load the package.

```
(* load the package *)
<< Graphics`SurfaceOfRevolution`
```

Collect the plot parameters.

```
(* plot parameters *)
param = {Boxed → False, ColorOutput → GrayLevel, BoxRatios → {1, 1, 1},
   Shading → False, Axes → False, SphericalRegion → True};
```

Plot the cutaway.

```
(* Gabriel's horn *)
g1 = SurfaceOfRevolution[1/x, {x, 1, 10}, {θ, 0, 3/2 π}, param];
```

Gabriel's Horn

Plot 88. Here we will present a more typical view of this figure.
Load the package first.

```
(* load the package *)
<< Graphics`SurfaceOfRevolution`
```

Here are the plot parameters

```
(* plot parameters *)
param =
  {ViewVertical → {1, 0, 0}, Ticks → {Automatic, Automatic, {-1, 0, 1}},
   Shading → False, Boxed → False, Axes → False,
   PlotPoints → {20, 15}, SphericalRegion → True};
```

This greatly simplifies the plot command.

```
(* plot Gabriel's horn *)
g1 = SurfaceOfRevolution[x^-1, {x, 0.1, 1}, param];
```

References

1. Abell, M. and Braselton, J., *Mathematica by Example*, 2e, Academic Press, San Diego, 1997.
2. Arfken, G. and Weber, H., *Mathematical Methods for Physicists*, 5e, Harcourt Academic Press, New York, 2000.
3. Bevington, P.R. and Robinson, D.K., *Data Reduction and Error Analysis for the Physical Sciences*, 2e, McGraw-Hill, New York, 1992.
4. Cap, F., *Mathematical Methods in Physics and Engineering with Mathematica*, CRC Press, Boca Raton, FL, 2003.
5. Folland, G.B., *Introduction to Partial Differential Equations*, 2e, Princeton, 1995.
6. Garcia, A.L., *Numerical Methods for Physicists*, 2e, Prentice Hall, 2000.
7. Ganzha, V. and Evgenii V., *Numerical Solutions for Partial Differential Equations: Problem Solving Using Mathematica*, CRC Press, Boca Raton, FL, 1996.
8. Glynn, J. and Gray, T., *The Beginners Guide to Mathematica Version 4*, Cambridge University Press, New York, 2000.
9. Gradshteyn, I.S. and Ryzhik, I.M., *Tables of Integrals, Series and Products*, 6e, Academic Press, 2000.
10. Griffiths, D., *Introduction to Electrodynamics*, Prentice Hall, 1998.
11. Magnusson, P. et al., *Transmission Lines and Wave Propagation*, 4e, CRC Press, Boca Raton, FL, 2001.
12. McGervey, J.D., *Introduction to Modern Physics*, Academic Press, 1971.
13. Mohr, P.J. and Taylor, B.N., *The Fundamental Physical Constants*, Physics Today, **56**:8, p. BG6, August 2003.
14. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P., *Numerical Recipes in FORTRAN*, 2e, Prentice Hall, 2000.
15. Rebbi, C., *The Lattice Theory of Quark Confinement*, Scientific American, p. 54, 1983.

16. Rebbi, C., The lattice theory of quark confinement, in *Particles and Forces at the Heart of the Matter*, eds. Carrigan, Jr., R.A. and Trower, P., p. 36, Freman, 1990.
17. Sanchez, D., Allen, R. and Kyner, W., *Differential Equations*, 2e, Addison-Wesley, New York, 1988.
18. Stowe, K., *Introduction to Statistical Mechanics and Thermodynamics*, John Wiley & Sons, New York, 1984.
19. Weisstein, E., *CRC Concise Encyclopedia of Mathematics*, 2e, CRC Press, Boca Raton, FL, 2004.
20. Wolfram, S., *The Mathematica Book,* 4e, Cambridge University Press, 1999.

Index

A

Abbot, Paul, 92
AccelerationDueToGravity, 77
Accum Module, 142
Accumulation errors, 29, 60
Addcolor Module, 260–262
AddEdge command, 577
Add-on packages, *see also specific package*
  DiscreteMath', 576–586
  Graphics', *503, 503–509,* 531–576, 634–642
  Graphics'ContourPlot3D', 634–642
  Graphics'Graphics3D', 643–662
  Graphics'Legend', 524
  Graphics'ParametricPlot3D', 662–691
  Graphics'PlotField3D', 668–675
  Graphics'Polyhedra', 675–685
  Graphics'Shapes', 685–690
  Graphics'SurfaceOfRevolution', 690–691
  LinearAlgebra'MatrixManipulation', 586–587
  Miscellaneous'Audio', 68–69
  Miscellaneous'BlackBodyRadiation', 75–78, 587
  Miscellaneous'Calendar', 69–72
  Miscellaneous'ChemicalElements', 72–74
  Miscellaneous'ResonanceAbsorptionLines', 588
  Miscellaneous'StandardAtmosphere', 588–589
  Miscellaneous'WorldPlot', 589–591
  NumericalMath', 591–594
  parallel processing, 12
  standard, 65–67
  Statistics'StatisticsPlot', 595–597
Adjacency List, 585
Affine Shape, *611*
AffineShape command, 688–690
Affine Transformation, 690
Affine transformations, 34–36
AgeOfUniverse, 77
Air resistance, 331–336
Algebra, linear, *see* LinearAlgebra'MatrixManipulation' add-on package
Algebra bypass, 159–170
AlgebraicManipulation palette, *20, 80*
Algebra package, *66*
Algebra Relm package, *66*
Altitude
  shading by, *603,* 617–618
  viscosity, 588–589
AmbientLight option, 633, 643
Ampersand sign (&), 48
Amplitude spectrum, 406, 411
Analyze Module, 143
AND operator, 254–255, 554
Apart command, 377–378
AppendTo command
  Bessel functions, 472

bypassing algebra, 165–166
color wheels, 254
cross-check, 146
curvature measurements, 195, 198
Dirac delta function, 223, 234
gray scales, 238
higher dimensions, 172–173
imaginary roots, 271–273
lists, 46
Monte Carlo Integration, 445
random walk, 266
Apply operator
coupled circuits, 389
cross-check, 149–151
Graphics'Spline', 565
Monte Carlo Integration, 448
Arbitrary orders, 117–118
Arithmetic Series (formula 3), 58
Armstrong, Neil, 69
Array function, 36, 43
Articles, resources, 91–92
Aspect ratio
color wheels, 252–253
gray scales, 241–242
AspectRatio option
basics, 264
color wheels, 250
contour plots, 278
disks, 247–249
graphics primitives, 516
gray scales, 242
imaginary roots, 275–276
kernel commands, 528, 530–531
Lattice field theory, 236
ListContourPlot, 280
Lotka-Volterra equations, 358
Monte Carlo Integration, 443
random walk, 268
tick marks, 243, 245
Assemble, higher dimensions, 173–174
Assumptions option, 479, 525–526
Asymmetric function, 520
Atmosphere, 588–589
AtmosphericPlot command, 589
AtomicWeight command, 72
Audio, 68
Average Module, 148
AvogardroConstant, 77
AxesLabel option, 310
Axes option
basics, 264
Bessel functions, 468
color wheels, 256
Dirac delta function, 231–232, 340
extraneous characters, 15
Fourier Transform, 404
Graphics'ContourPlot3D', 635, 641
Graphics'FilledPlot', 533, 535
Graphics'Graphics', 548, 550
Graphics'Graphics3D', 644, 646, 649–651, 653–654
Graphics'PlotField', 571–572
Graphics'Polyhedra', 676–677, 681
Graphics'SurfaceOfRevolution', 691
kernel commands, 612, 614, 617
pictionary, *601*
radio frequency pulse, 409
sampling function, 282–283, 286
spherical harmonics, 296–297
AxesOrigin option, 518
AxesStyle option
Dirac delta function, 340
imaginary roots, 275–276
random walk, 268
Axes suppression, 612

B

Background option, 518
Backward referencing, 677
BarChart command, 544
BarChart3D, pictionary, *607*
BarChart3D command, 660–661
BasicCalculation palette, *19, 79*
BasicInput palette
basics, *20, 80*
definite integration, 435, 438
Dirac delta function, 338
multivariate expressions, 432
polynomials and rational functions, 424–425
BasicTypesetting palette, *19, 79*
Bessel Differential Equation, 465
Bessel Function of the First Kind, 465
Bessel Function of the Second Kind, 465

Bessel functions, 465–478, *467*
Beta function, 456
Bézier Spline, 563, 565–566
Bitmap (BMP) files, 8, 210–211
BlackBodyProfile command, 75, 587
Blackbody radiation, 69, *see also* Miscellaneous'BlackBodyRadiation'
BlackBodyRadiation'BlackBodyProfile, *511*
Blackman, Nancy, 92
Blinding the kernel, 15–16
Block command
  differential equations, 311–312
  Dirac delta function, 216
  large numbers, 30
Block function, enormous equations, 207
BMP, *see* Bitmap (BMP) files
BohrRadius, 77
BoilingPoint command
  Miscellaneous'BlackBodyRadiation', 75
  Miscellaneous'ChemicalElements', 72
  periodic table of the elements, 84
Boltzmann constant, 481
BoltzmannConstant command, 77
Boneyard, resources, 92–96
Born and Wolf studies, 33
Boundary values, Dirac delta function, 341
Bounded diagram, 583–585
BoundedDiagram command, 585
Box-and-Whisker Plot, 595
Boxed option
  Graphics'Graphics3D', 644
  Graphics'Polyhedra', 682–685
  Graphics'Shapes', 686
  Graphics'SurfaceOfRevolution', 691
  helicoids, 689–690
  helix, 687–688
  kernel commands, 612–613, 617
  Möbius Strip, 687
  pictionary, *601*
  PointParametricPlot3D, 665
  sampling function, 282–283, 286
  spherical harmonics, 296
BoxRatios option
  Graphics'SurfaceOfRevolution', 691
  kernel commands, 614, 617, 629–630
  pictionary, *601*
  sampling function, 282–283, 286
BoxWhiskerPlot command, 595
Braking trajectory, 654, 663–665
Buckling constant, 469
Buffon-Laplace Needle Problem, 442
Buffon Needle Problem, 442
Bugs, 12–15, *see also* Cautions; Debugging code
Butcher'ButcherPlot, pictionary, *512*
ButcherPlot command, 592
ButcherTrees command, 592

C

Calculus package, *66*
Calendar, Help Browser resource, 72
Cartesian coordinates
  basics, 33
  Graphics'Graphics3D', 652
  sampling function, 286–287
  spherical harmonics, 495
CartesianMap command, 573, 575–576
Cartesian space, kernel commands, 625–628
Cautions
  accumulation, loops, 60
  characters, extraneous, 15
  characters, number of, 196
  Clear["Global'*"], 7
  code problems, 15–16, 146
  ColorFunction Hue, 277
  comments, 15–16, 146
  common typos, 26
  comparison functions, 63
  constant vectors, 53
  ContourPlots, 277
  Contours option, 635
  covectors, 44
  cursor, graphic coordinates, 69
  elements, 635
  embedded comments, 15–16, 146
  error in equation, 659
  figure orientation, 281
  graphic coordinates, measuring, 69
  graphics, size, 242, 246
  high aspect ratio graphics, 242
  Hue, 277
  icon, 3
  infinities, 63
  integers, 635

kernel blinding, 16
leading spaces, 13
lists, 635
loops, accumulation, 60
Metropolis, 442
Monte Carlo integration, 442
Nulls, extraneous, 15
orientation of figures, 281
output precision, 196
precision erosion, 60
punctuation, 240
screen display default, 196
shading schemes, 643
Shadow routine, 643
Simulated Annealing, 442
singular elements, 635
SingularValues, 126
small dimensions, 44
SphericalRegion, 281
stellated objects, 680
text, 246
typos, common, 26
unreferenced data, 3
vector arithmetic, 53
vectors, 44
Which function, 65
CCD, *see* Charge-coupled devices (CCD)
C code, representations, 23
Ceiling, 584
CForm command, 23
CharacterRange command
generating lists, 31
Graphics'MultipleListPlot, 563
linear regression, 139
lists, 45
Characters, 15, 196
Charge-coupled devices (CCD), 33, 524, 527–530
Chebysheff representation, 34
Chemical elements, 85
Chop command, 124–126
Circle command
Graphics'InequalityGraphics', 552–554
graphics primitives, *498,* 515–516
random walk, 267
Circle polynomials, 33
Circuits
higher order differential equations, 325–327
Laplace transforms, 379–398
ClassicalElectronRadius, 77
Clear command
bypassing algebra, 166
cross-check, 149
kernel commands, 615
large numbers, 30
Legendre polynomials, 493
manipulating lists, 41
Riemann zeta function, 483
seed notebooks, 6–7
While loop, 63
Clear["Global'*"] command, 7, 11
CMYK color
basics, 209–210
color wheels, 257–259
Help Browser resource, 86, 210
kernel commands, 614–615
pictionary, *602*
sum, 263
CMYKColor command, 261, 614–615
Code problems, 15–16, 146
Coefficients, 318–324
ColorFunction option
cautions, 277
Graphics'Graphics3D', 646, 648–650, 653
Graphics'PlotField3D', 671–672
kernel commands, 523, 528–531, 614–616, 618, 620–621, 626
ListPlotVectorField3D, 673
pictionary, *602*
PlotVectorField3D, 673
Coloring surface graphics, 632
Color options, graphics primitives, 209–210
ColorOutput option
Graphics'Shapes', 686
Graphics'SurfaceOfRevolution', 691
helicoids, 689–690
helix, 687–688
Möbius Strip, 687
PointParametricPlot3D, 665
SphericalPlot3D, 666
Color palette, 85, 99, 210
Colors, 85, 210
Color scales, 523–524
Color scheme variations, 618
Color wheels, 249–263
Combinatorica, 577–579

Combinatorica'FerrersDiagram, *510*
Combinatorica'ShowGraph, *510*
Combinatorica'ShowGraphArray, *510*
Combining three-dimensional graphics objects, 666
Comments, cautions, 15–16, 146
Common settings, 9, 11
Compare function, 487
Comparison functions, 63
CompleteCharacters palette, *20, 80*
CompleteGraph command, 577, 579
ComplexExpand command, 269–270
ComplexMap, *see* Graphics'ComplexMap'
Complexmap'CartesianMap, *509*
ComplexMap'PolarMap, *509*
Composite Bézier command, 566
Composite Bézier splines, 563
Computation, Laplace transforms, 371–398
ComputationalGeometry'DiagramPlot, *511*
ComputationalGeometry'PlanarGraphPlot, *511*
ComputationalGeometry'TriangularSurfacePlot, *511*
Computation examples
  algebra bypass, 159–170
  arbitrary orders, 117–118
  basics, 99
  cross-check, 144–145
  curvature measurement, 180–208, *185–186, 208*
  derivation, 128–129
  error propagation, 180
  higher dimensions, 171–180
  higher powers, 156–170
  inverse problem, 180–208
  linear regression, 128–180
  list-based tools, 133–137
  precomputation, 170
  quadratic equation, 99–118
  singular matrices and inversion, 118–127
  symbolic explorations, 126–127
  symbolic solution, 106–117
Conjugate command
  Graphics'Graphics3D', 644
  leading spaces, 14
  spherical harmonics, 295–296
Connected lines, 515
Connecting points, 519–520
Conservative Field, 568
Conservative forces, 567–568
Constantine (Roman emperor), 70
Constants, palettes, 86–91
Constant vectors, 51–53
Constraint Module, 96
ContourPlot command, *500–501,* 520–522
ContourPlot3D command, 635–636, 638
ContourPlots, cautions, 277
Contour plots, 2D plotting, 276–280
ContourShading option, 277–278, 520–522
Contours option
  cautions, 635
  contour plots, 278–279
  Graphics'ContourPlot3D', 635, 637–638, 640–642
ContourStyle option, 640–642
Contract command, 579
Convective cooling, 307–313
Convert command, 89
Convex hull, 581–582, 584
Coordinate systems, choice, 666–667
Corners, rectangles, 237–239
Correlation Index, 129
Cos[] function
  Bessel functions, 475
  color wheels, 250
  contour plots, 278
  Dirac delta function, 440
  Fourier Transform, 405
  Graphics'Graphics3D', 655
  Graphics'MultipleListPlot, 563
  Integrate command, 441
  kernel commands, 531, 629–631, 634
  Laplace Transform, 372
  Legendre polynomials, 490, 492–494
  radio frequency pulse, 410
  z-transform, 418
Cosh function, 372–373
CosmicBackgroundTemperature, 77
Coupled circuits, 382–398
Covectors, cautions, 44
Cramer Module, 396
Cramer's Rule
  coupled circuits, 389, 395–396
  Laplace transforms, 383–384
  The CRC Encyclopedia of Mathematics, 383
Crashes, notebooks, 16–17
CRC common.m notebook, 639

The CRC Encyclopedia of Mathematics
Adjacency List, 585
Affine Transformation, 690
Arithmetic Series (formula 3), 58
Bessel Differential Equation, 465
Bessel Function of the First Kind, 465
Bessel Function of the Second Kind, 465
Bézier Spline, 566
Box-and-Whisker Plot, 595
Buffon-Laplace Needle Problem, 442
Buffon Needle Problem, 442
Conservative Field, 568
Convex Hull, 582
Correlation Index, 129
Cramer's Rule, 383
Critical Point, 353, 359
Cubic Spline, 564
Curvature, 181, 625
Delaunay Triangulation, 581
Delta Function, 215, 337
Distribution (Generalized Function), 215
Divergence, 633
Dodecahedron, 680
Dot Product, 174
Ellipsoid, 536, 630, 633–635
Embedding, 580
Euler Formula, 413
Factorial, 371
Ferrers Diagram, 577
Fourier Cosine Transform, 413
Fourier Sine Transform, 413
Fourier Transform, 402–403
Function, 156
Gabriel's Horn, 691
Gamma Function, 371
Geodesic Dome, 683
Golden Ratio, 242
Gradient, 626, 628
Gram-Schmidt Orthonormalization, 612, 615
Graph, 577–580
Hankel Function, 465
Heaviside Calculus, 370
Heaviside Step Function, 382
Helicoid, 687–688
Helix, 628, 687
high-order Runge-Kutta methods, 592
icon, 3
Icosahedron, 678
Integral Transforms, 370
Laplace Transform, 371
Laplacian, 621
Least Squares Fitting, 129
Limaçon, 542
Lotka-Volterra Equations, 359
Mersenne Prime, 29
Möbius Strip, 687
Monte Carlo Integration, 442
Monte Carlo Method, 442
Moore-Penrose Generalized Matrix Inverse, 120, 126
Osculating Circle, 185
Padé Approximant, 593
Parity, 300
Parseval's Theorem, 412
Partial Fraction Decomposition, 377
Platonic Solid, 675
Pseudoinverse, 120
Quadratic Equation, 690
Quantile, 597
Quintic Equation, 117
Recursion, 176
Runge-Kutta methods, higher-order, 592
Scalar Triple Product, 174
Sign, 401
Sinc Function, 286, 520
Sine Integral, 525
Singular Value Decomposition, 123
Spherical Coordinates, 629
Spherical Harmonic, 297
Spline, 564
Stationary Point, 351
Stellation, 682–683
Torus, 659, 665, 686
Tree, 578–579, 586
Triangulation, 581
Truncated Cube, 685
Uniform Polyhedron, 675
Vector Triple Product, 174
Vertex (Graph), 585
Voronoi Diagram, 585
Zernike Polynomials, 300–301
Zeta Function, 570
Z-transform, 417
CreateSlideShow palette, *21, 81*
Creating plots, 518
Critically damped system, 345–347
Critical points, 350–359

Cross-check, 144–145
Cross product, 174
Cube, 676–677, 679, 681–685
Cubic spline, 563–566
Cuboid command, *498,* 516
Cursor, graphic coordinates, 69
Curvature function, 621–625
Curvature measurement, 180–208, *185–186, 208*
Curvature shading, *603*
Curves identification, 518
Cylinders, 341–345
Cylindrical cavity, 474–478
Cylindrical coordinates, 437
CylindricalPlot3D, 599, *608*
CylindricalPlot3D command, 666–667

D

Damped mass-spring system, 345–347
Darts, throwing, 443–449
Dashing, pictionary, *507*
Dashing command
  Bessel functions, 470
  coupled circuits, 385, 397
  damped mass-spring system, 346
  Graphics'Spline', 564
  kernel commands, 517
Data
  entering, 17, 25–27
  keyboard shortcuts, 18–21
  standard palettes, 17–18, *19–21*
Data structures
  basics, 22
  concepts of lists, 32–36
  constant vectors, 51–53
  entering data, 25–27
  generating lists, 30–32
  large numbers, 27–30
  lists, 30–51
  manipulating lists, 36–44
  queries, 24
  representations, 22–24
  square root operator, 44–53
Datestamp, *see* Stamp
DayOfWeek command, 69
DaysBetween command, 69
DaysPlus command, 70
Debugging code, 52, 144–145
Defining differential equations, 303–367
Delaunay Triangulation, 581, 584
Delta Function, 215, 337
Demo palettes, 82–91
Density, kernel commands, 519
DensityPlot command
  kernel commands, 522, 524–529
  pictionary, *501–502*
Derivation, 128–129
Determinant[] operations, 23
DeutreronMagneticMoment, 77
DeutreronMass, 77
DiagonalMatrix command, 16, 124
Diagram color, 616–617
DiagramPlot command, 583, 585
Differential equations
  air resistance, 331–336
  basics, 303
  boundary values, 341
  coefficients, 318–324
  convective cooling, 307–313
  critically damped system, 345–347
  critical points, 350–353
  cylinders, 341–345
  defining, 303–367
  different coefficients, 318–324
  Dirac delta function, 336–367
  electrical circuits, 325–327
  entering, 303–367
  gravitational field, 328–336
  heat conduction, 341–345
  higher order, 316–336
  Lotka-Volterra equations, 354–364
  mass, 328–336
  mass-spring system, 336
  Newton's law, convective cooling, 307–313
  parachute jumper exercise, 331–336
  phase plane portrait, 364–367
  phase plots, 347–350
  radioactive decay, 313–316
  second order equations, 327–328
  solving, 303–367
  systems of equations, 347–350
Digits, number shown, 37
Dimensions command

basics, 35
Gamma function, 459
kernel commands, 530
Miscellaneous'ChemicalElements', 72
Dirac, P.A.M., 215
Dirac delta distribution, 214–234
Dirac delta function
boundary values, 341
critically damped system, 345–347
critical points, 350–353
cylinders, 341–345
differential equations, 336–367
Fourier Transform, 401–402
heat conduction, 341–345
integration, 438–440
Lotka-Volterra equations, 354–364
phase plane portrait, 364–367
phase plots, 347–350
systems of equations, 347–350
DirData, 8, 11, 13
Directory structure, 7–9, 11, 249
DirGraph
directory structure, 8
disks, 248
leading spaces, 13
notebooks, 11
DirHome, 8, 13
DirPack
curvature measurements, 206
directory structure, 8
leading spaces, 13
notebooks, 11
Disconnected lines, 515
Discrete Fourier transforms, 414–417, *416*
DiscreteMath'Combinatorica, *510*
DiscreteMath'Combinatorica', 576–580
DiscreteMath'ComputationalGeometry, *511–513*
DiscreteMath'ComputationalGeometry', 580–585
DiscreteMath'ExprPlot, *511*
DiscreteMath package, *66*
DiscreteMath'Tree, *511–513*
DiscreteMath'Tree', 585–586
DiscreteMath'TreePlot, *511*
Discrete versions
Fourier transforms, 414–417, *416*
ListContourPlot, 279–280
ListPlot3D, 290–291, *292*
ListPlotVectorField3D, 672–675
LogLinearListPlot command, 539
LogLogListPlot command, 540–541
PointParametricPlot3D, 663–665
Discretization, 522
Discriminant function, 102
Disk command
color wheels, 251, 258–260
Lattice field theory, 236
Monte Carlo Integration, 443
Disks, graphics primitives, 246–249, *498,* 516
Display, graphics primitives, 211–214
Display command, 248
Display (export), 211
$DisplayFunction command, 212
DisplayFunction option
basics, 264
coupled circuits, 397
displaying graphics, 211
electrical circuit problem, 326
Graphics'Graphics3D', 657
Help Browser resource, 214
SphericalPlot3D, 666
Distributed computing, 12
Distribution (Generalized Function), 215
DistRU function, 444, 447
Divergence, 633
Do command, 218
Dodecahedron, 676–680
Do loop
Bessel functions, 472
bypassing algebra, 166, 168
color wheels, 260–261
cross-check, 146
curvature measurements, 194–195, 198–199
Dirac delta function, 218, 222–225, 234
generating lists, 31–32
gray scales, 238–239
Help Browser resource, 61
higher dimensions, 172–174
imaginary roots, 270–271
large numbers, 28–29
linear regression, 130–131, 141
lists, 46–48
Monte Carlo Integration, 445
phase plane portrait, 365–366
programming, 53–61

spherical harmonics, 294
tick marks, 243, 245
Dot product, 167, 174
Double helix, *611,* 688
Double helix command, 688
Double precision, 179
Drop command, 74, 393
DSolve command, *see also* Differential Equations
Bessel functions, 465–466, 469
Laplace Transform, 375–376
DynamicViscoity command, 589

E

EarthMass, 77
Easter table, 70
EdgeForm command
Graphics'Graphics3D', 644–645
Graphics'ParametricPlot3D', 663
kernel commands, 632–633
spherical harmonics, 297
SphericalPlot3D, 666
Electrical circuits, 325–327, 379–381
ElectronCharge, 77
ElectronComptonWavelength, 77
ElectronFactor, 77
ElectronMagneticMoment, 77
ElectronMass, 77
ElementAbsorptionMap command, 588
Element extraction, 74
Elements, *82,* 635
Elem recursive function, 175–178
Ellipsoids
helicoids, 689–690
kernel commands, 629–633
pictionary, *604*
The CRC Encyclopedia of Mathematics, 536, 630, 633–635
Embedded comments, 15–16, 146
Embedding, 579–580
Enormous equations, 206–208
Entering differential equations, 303–367
EPS, *see* Extended PostScript (EPS) files
Equation of motion, 329
Equations, *see* Differential equations
Equations, enormous, 206–208
Equipotential surfaces, 637–639
ErrorBar command, 560
ErrorListPlot, *505*
ErrorListPlot command, 547–548
Errors
computation examples, 180
discrete Fourier transforms, 416–417
equation, caution, 659
formulae, 58–60
linear regression, 180
relative, 653–654
sampling function, 284
singular matrices and inversion, 120
spelling checker, 95
Euler Beta function, 456
Euler formula, 413
Evaluate command
Bessel functions, 466
coupled circuits, 397
differential equations, 312
Dirac delta function, 232, 234, 339–340
extraneous characters, 15
Graphics'FilledPlot', 532
higher order differential equations, 318, 320
kernel commands, 518
Legendre polynomials, 489
Lotka-Volterra equations, 357–358
phase plane portrait, 367
systems of equations and phase plots, 349
Expand command, 141Even function, 300, 485–486
Exp command
Fourier Transform, 399, 403, 405
Graphics'FilledPlot', 532
Graphics'Graphics', 537–540
Microscope'Microscope, 594
Riemann zeta function, 481
Exploding pie charts, 550
Export, 211
Export command, 210, 248
ExprPlot command, 586
ExpToTrig command, 418
Extended PostScript (EPS) files, 8, 210–211, 248, 288–289
Extract command, 74
Extraneous characters, 15

F

Factorial function
curvature measurements, 205
Gamma function, 451
Help Browser resource, 371
The CRC Encyclopedia of Mathematics, 371
Factorial2 function, 457
FactorTerms command, 378
FaradayConstant, 77
Fermi-Distribution, 480–487
Ferrers Diagram command, 577
Figure orientation, cautions, 281
File system organization, 7–9, 11, 249
FilledPlot, *503*, *see also* Graphics'FilledPlot'
FilledPlot command, 532–533
Fills command, 253, 255, 262
Filter representation, 407–411
Findlay, Josh, 27
FindMaximum command, 632
FindMinimum command, 23, 144, 632
FindRoot command, 105–106
Fine list, kernel commands, 523
FiniteGraphs command, 580
First function
coupled circuits, 389–390, 395
curvature measurements, 189, 192–193, 200
graphics primitives, 514
Flat matrix, 44
Flatten command
bypassing algebra, 161–162
cross-check, 151, 155
DiscreteMath'ComputationalGeometry', 581–584
Gamma function, 459
Graphics'MultipleListPlot, 561
Graphics'PlotField', 572
graphics primitives, 515
heat conduction, cylinder, 342
higher dimensions, 171
higher order differential equations, 318
inverse Laplace transform, 377
kernel commands, 529–530, 619
Legendre polynomials, 491
ListPlotVectorField3D, 674–675
Lotka-Volterra equations, 363–364
manipulating lists, 38
mass in gravitational field, 334
phase plane portrait, 366
quadratic equation, 109, 111
systems of equations and phase plots, 349
Floor command, 584
Flow control, 53–65
Fonts, common settings, 9
For loop, 61
FORTRAN, 22–23
FortranForm command
curvature measurements, 196
fundamental constants, 88, 91
representations, 23
Forward referencing, 677
Fourier command, 399
Fourier Cosine Transform, 413
FourierCosTransform command, 398
FourierCos transforms, 413–414
FourierParameters option, 402–404
Fourier Sine Transform, 413
FourierSinTransform command, 398
FourierSin transforms, 413–414
Fourier Transform, 402–403
FourierTransform command, 398–404
Fourier transforms
basics, 398–399
discrete Fourier transforms, 414–417, *416*
filter representation, 408–411
FourierCos transforms, 413–414
FourierParameters option, 402–404
FourierSin transforms, 413–414
Parseval's theorem, 412–414
radio frequency pulse, 408
signal modulation, 405–408
FrameLabel option
basics, 264
coupled circuits, 397
Graphics'MultipleListPlot, 561
imaginary roots, 275–276
radio frequency pulse, 408
Frame option
Bessel functions, 468
coupled circuits, 397
Dirac delta function, 231–232, 340
extraneous characters, 15
Fourier Transform, 401, 404
Graphics'FilledPlot', 533, 535

Graphics'Graphics', 548, 550
Graphics'Graphics3D', 651, 654, 657
Graphics'PlotField', 567–572
imaginary roots, 275–276
kernel commands, 518
Plot command, 265
radio frequency pulse, 409
random walk, 267–268
Riemann zeta function, 481–482
FrameTicks option, 264–265, 275–276
FullForm command
coupled circuits, 390–391
linear regression, 139–140
Miscellaneous'ChemicalElements', 74
FullSimplify command
Bessel functions, 476, 478
curvature measurements, 182–183, 187, 201
enormous equations, 206
fundamental constants, 89
Inverse Laplace Transform, 375
kernel commands, 623, 631
multivariate expressions, 432–433
polynomials and rational functions, 428
second order equation, 328
Function (&), 156, 222
Functions
mathematical, 29
polymorphism, 43
The CRC Encyclopedia of Mathematics, 156

G

Gabriel's Horn, *611,* 691
Gain function, kernel commands, 529–530
GalacticUnit, 77
Gamma function
basics, 451–465
Bessel functions, 467
Help Browser resource, 371
Laplace Transform, 371
Riemann zeta function, 479
The CRC Encyclopedia of Mathematics, 371
Gas, noninteracting particles, 672–673
Gaussian function, 402
GeneralizedBarChart, *505*
GeneralizedBarChart command, 544–546
General::meprec, 207
Geodesate command, 683
Geodesic Dome, 683
Geometry package, *66*
Get command, 206
Ghost module, 94–95
GIFF, *see* Graphical Interchange File Format (GIFF)
Glitches, *see* Bugs; Cautions
Global variables, clearing, 7, 11
Golden Ratio, 241–242, 247
Gradient, 626, 628
Gradient function, 625
Gradient shading, *603,* 625
Gradshteyn and Ryzhik studies, 4, 442
Gram-Schmidt orthogonalization, 612
Gram-Schmidt Orthonormalization, 612, 615
Graph, 577–580
Graphical Interchange File Format (GIFF), 8, 210
Graphic coordinates, measuring, 69
Graphics
basics, 209
color options, 209–210
contour plots, 276–280
harmonics, spherical, 293–297, *298*
imaginary roots, plotting, 269–276
ListPlot3D, 290–291, *292*
list plots, 266–276
output file types, 210–211, *211*
parity states, rotation, *299,* 299–301, *301*
pictionary, 280–281
Plot command, 263–265
Plot3D, 281, 286–290, *290*
random walk, 266–268
rotation, parity states, *299,* 299–301, *301*
sampling function, 281–292
ScatterPlot3D, 291–292
SphericalPlot3D, 292–298, *293,* 599
three-dimensional plotting, 281–298
two-dimensional graphic types, pictionary, 280–281
Graphics, size, 242, 246
GraphicsArray command
electrical circuit problem, 326–327
Graphics'Polyhedra', 680–685

Graphics'Shapes', 686
GraphicsArray object, 325
Graphics'BarChart, *505*
Graphics command
color wheels, 251–252, 258–260
coupled circuits, 397
damped mass-spring system, 347
Dirac delta function, 215–217, 219, 223, 228–230
disks, 246, 248–249
graphics primitives, 514–516
Graphics'Spline', 564, 566
gray scales, 240
kernel commands, 528, 530
Lattice field theory, 236
Miscellaneous'WorldPlot', 590
Monte Carlo Integration, 443
tick marks, 243–245
Graphics'ComplexMap', 573–576
Graphics'ContourPlot3D', *605*
Graphics'ContourPlot3D' package, 634–642
Graphics3D, 281
Graphics3D command
graphics primitives, 516–517
Graphics'Shapes', 686
helicoid, 689
helix, 688
Möbius Strip, 687
Graphics3D objects
graphics primitives, 516–517
helicoids, 690
helix, 688
Graphics3D[Torus], *611*
Graphics'FilledPlot', *503,* 532–535
Graphics'Graphics', 537–551
Graphics'Graphics3D' package, 643–662
Graphics'Graphics tools, *504*
Graphics'ImplicitPlot'
basics, 535–536
curvature measurements, 188
pictionary, *503*
Graphics'InequalityGraphics', 253, 551–555
Graphics'InequalityPlot', *505–506*
Graphics'LabeledListPlot, *505*
Graphics'Legend', 524, 554–555, *see also* Legend
Graphics'ListContourPlot3D', *605*
Graphics'MultipleListPlot', *505–507, 555–563*
-Graphics- objects
disks, 246–247
Miscellaneous'WorldPlot', 590
Plot command, 265
Graphics objects, kernel commands, 520
Graphics package, *66*
Graphics'ParametricPlot3D' package, 662–691
Graphics'PieChart, *505*
Graphics'PlotField', 349, 567–572, *see also* PlotField
Graphics'PlotField3D' package, 668–675
Graphics'Polyhedra' package, 675–685
Graphics primitives
basics, 514–517
circles, 515–516
color wheels, 249–263
connected lines, 515
Dirac delta distribution, 214–234
disconnected lines, 515
disks, 246–249, 516
Graphics3D objects, 516–517
gray scales, 237–246, *242,* 514–515
Help Browser resource, 214
isosceles triangles, 215–230, *218*
lattice field theory, 235–237
lessons learned, 263
monolith, 516
pictionary, 280–281, *498*
points, 514
three-dimensional plotting, 281–298
tophats, 231–234
two-dimensional plotting, 263–280
Graphics rendering, 249
Graphics'Shapes' package, 685–690
Graphics'Spline', 563–566, *see also* Spline
Graphics[Spline[]], *507*
Graphics'SurfaceOfRevolution' package, 690–691
Graphics'TextListPlot, *505*
GraphSum command, 578
GravitationalConstant, 77
Gravitational field, 328–336
GrayLevel command
Bessel functions, 466
contour plots, 276
Dirac delta function, 233–234
graphics primitives, 514, 516
Graphics'Spline', 564, 566

kernel commands, 518, 530, 614, 628
Lattice field theory, 235
Legendre polynomials, 488–489
Monte Carlo Integration, 443
PlotVectorField3D, 672
GrayLevel option
kernel commands, 529, 614, 628
pictionary, *602*
SphericalPlot3D, 666
Gray levels, 85, 210, 233
Gray scales, graphics primitives, 237–246, *242,* 514–515
GreatDodecahedron, 676–677, 679
GreatIcosahedron, 676–679
GreatStellatedDodecahedron, 676–677, 679
GridGraph command, 579
Gyromagnetic ratio, 235

H

Hankel Function, 465, 475
Head command
basics, 94
coupled circuits, 391–393
manipulating lists, 38–40, 43
Miscellaneous'ChemicalElements', 73
Miscellaneous'WorldPlot', 590
queries, 24
While loop, 64
Heat conduction, 341–345
Heaviside Calculus, 370
Heaviside Step Function, 382
Helicoid, 687–689
Helix
Graphics'Shapes', 687–688
kernel commands, 628
pictionary, *604, 611*
The CRC Encyclopedia of Mathematics, 628, 687
Help Browser
audio, 68
basics, 2
blackbody radiation, 69
calendar, 72
chemical elements, 85
CMYK color, 86, 210
color palette, 85, 99, 210
colors, 85, 210
Combinatorica, 577–579
display (export), 211
DisplayFunction, 214
do loop, 61
element extraction, 74
export, 211
factorial function, 371
flow control, 53
Function (&), 156, 222
functions, mathematical, 29
Gamma function, 371
General::meprec, 207
Graphics3D, 281
graphics primitives, 214
graphics rendering, 249
gray levels, 85, 210, 233
hue, 85, 210
icon, 3
import, 211
keyboard shortcuts, 21
light source variation, 297, 616
linear regression, 145
ListPlot, 251
for loop, 61
low level graphics rendering, 249
Macintosh, 21
mathematical functions, 29
mathematical notation, entering, 21
Microsoft Windows, 21
MultipleListPlot, 562
music, 68
NumericalMath'SplineFit', 563
older *Mathematica* versions, 5
palettes, 81
Part command, 353
periodic table, 85
physical constants, 91
polynomial equations, 270
programming, 53
pseudoinverse, 120–121
Pure Function, 156
Quintic equation, 117
rendering, graphics, 249
RGB color, 85, 210
Root command, 117
Rule Delayed operator, 270
SingularValues, 123

sound, 68
standard packages, 67
statistics plots, 595–597
surface color variation, 297, 616
text, 218, 251
3D Graphics, 281
touch tones, 68
units, 89
variables, 270
Hexahedron, 676–677, 679
High aspect ratio graphics, 242
Higher dimensions, 171–180
Higher order differential equations
air resistance, 331–336
basics, 316–318
coefficients, 318–324
electrical circuits, 325–327
gravitational field, 328–336
mass, 328–336
mass-spring system, 336
parachute jumper exercise, 331–336
radioactive decay, 313–316
second order equations, 327–328
Higher powers, 156–170
High-order Runge-Kutta methods, 592
Histogram3D, *607*
Histogram3D command, 661–662
Homuth, Lars, 91
Horn torus, 686, *see also* Torus
Hot function, kernel commands, 524
HubbleConstant, 77
Hue option
cautions, 277
Graphics'Graphics3D', 646–647, 650, 655
Graphics'ParametricPlot3D', 663
Graphics'PlotField3D', 671
Help Browser resource, 85, 210
kernel commands, 524, 528–529, 614–615, 617–618
pictionary, *602*

I

IcePoint, 77
Icons (in book), 3
Icosahedron, 676–679
Identity command, 573–574
IdentityMatrix command
Graphics'MultipleListPlot, 560
Lattice field theory, 235
linear regression, 138
list-based tools, 135
singular matrices and inversion, 127
If statement
Bessel functions, 472
curvature measurements, 198
kernel commands, 624
Monte Carlo Integration, 448
While loop, 64
Illumination, kernel commands, 616
Image size, common settings, 9
ImageSize option
basics, 264
electrical circuit problem, 326
gray scales, 242
Miscellaneous'BlackBodyRadiation', 75
tick marks, 243, 245
Imaginary command, 273
Imaginary operator, 272
Imaginary roots, plotting, 269–276
ImplicitPlot, *503*
ImplicitPlot command, 188, 535–536, *see also* Graphics'ImplicitPlot'
Import, 211
In and Out column, 92
InequalityGraphics, *see* Graphics'InequalityGraphics'
InequalityPlot, *506*
InequalityPlot command
color wheels, 253–255
Graphics'InequalityGraphics', 552–554
Infinities, cautions, 63
Inflating values, 49–51
InputForm command, 24
Inputs, sorting, 220
Integer command, 587
Integers, 14, 635
Integer variables
color wheels, 254
Gamma function, 458, 462
Graphics'Graphics3D', 644
higher dimensions, 173, 175–176
Legendre polynomials, 490, 493
Riemann zeta function, 486

Integral function, 461–465
Integral Transforms, 370
Integrate command
  integration, 440–442
  kernel commands, 525–526
  Riemann zeta function, 479
Integration
  basics, 423–424
  defined, 435–438
  Dirac delta function, 438–440
  Integrate command, 440–442
  Monte Carlo integration, 442–449, *449*
  multivariate expressions, 430–434
  polynomials, 424–430
  rational functions, 424–430
Integration icon, 432
InternationalCharacters palette, *20, 80*
Internet tutorials, 2, *see also* World Wide Web
InverseFourier operation, 415
InverseFourierTransform option, 404, 408
Inverse functions, curvature measurements, 188–189
InverseLaplaceTransform command
  circuit problem, 381
  coupled circuits, 385, 397
  Laplace Transform, 373–375
  ODE solution, 376–377
Inverse[] operations
  bypassing algebra, 169
  representations, 23
  singular matrices and inversion, 120
Inverse problem, 180–208
InverseZTransform, 418–419
Inversion, *see* Singular matrices and inversion
Isosceles triangles, 215–230, *218*

J

Join command, 675
Joint Project Experts Group (JPEG) files, 8, 210–211
JPEG, *see* Joint Project Experts Group (JPEG) files

K

Kennedy, John (former President), 69
Kernel, blinding, 15–16
Kernel commands, 2D plot types
  basics, 517
  charge-coupled devices, 524, 527–530
  ColorFunction, 529–531
  color scales, 523–524
  connecting points, 519–520
  ContourPlot, 522
  creating plots, 518
  curves identification, 518
  density, 519
  DensityPlot, 522, 524–529
  discretization, 522
  fine list, 523
  Flatten, 529–530
  Graphics objects, 520
  GrayLevel option, 529
  hot, 524
  Hue option, 524
  Integrate command, 526
  Legend, 528
  ListContourPlot, 520–521
  ListDensityPlot, 522–523
  ListPlot, 519
  mesh, 522
  origin of plot, 518
  Parametric Plot, 531
  Partition, 529–530
  Plot command, 517, 531
  plot improvements, 521
  PlotRange, 522
  plotting lists of functions, 517–518
  Raster command, 524, 529
  resolution, 521–522
  sampling function, 520
  shading, 518, 520
  sinc function, 520, 525–526
Kernel commands, 3D plot types
  altitude, shading by, 617–618
  axes suppression, 612
  Boxed parameter, 612–613
  BoxRatios, 629
  CMYK, 614–615
  color function, 620–621, 626
  coloring surface graphics, 632

color scheme variations, 618
curvatures and curvature function, 621–625
diagram color, 616–617
EdgeForm, 632
ellipsoids, 629–633
gradient, shading, 625
Gram-Schmidt orthogonalization, 612
GrayLevel option, 614, 628
helix, 628
Hue option, 614–615
illumination, 616
lap function, 620
ListPlot3D command, 616–617, 619
Mercator parameterization, 633
mesh, 613, 632
morered, 618
Norm command, 618
normfac variable, 632
orthogonal functions, 612
ParametricPlot3D, 628, 633–634
params variable, 617
Partition, 619–621
PlotPoints, 633–634
position of lights, 616
proportioning, 612
redcold function, 615–616
redhot function, 615, 621, 625–627, 632
resolution, 613–614
RGBColor, 614
scalar function, 631
shading, 613, 617–622, 625–628
Show command, 616
spherical coordinates, 628–629, 631
SphericalRegion, 613
viewpoint, 613
Keyboard shortcuts, 18–21
KineticTtemperature command, 589
KroneckerDelta, 419

L

LabeledListPlot command, 550–551
Labeling points, 216–217
Lamina, 251–263
Lap function, 620
Laplace's equation, 487
Laplace transforms
basics, 370
circuits, 379–398
computation, 371–398
coupled circuits, 382–398
The CRC Encyclopedia of Mathematics, 371
electrical circuits, 379–381
inverse Laplace transforms, 374–375
linear integral transform properties, 370
ODE solution, 375–379
transforms, 370–398
Laplacian, 621
Laplacian shading, *603*
Large numbers, data structures, 27–30
Last function, 165–166, 172
Lattice field theory, 235–237
Leading spaces, 12–15
Learning curve, 4
Least square problems, 52
Least Squares Fitting, 129
Legend, *see also* Graphics'Legend'
kernel commands, 528
plot with, pictionary, *506*
Legend package, 237
LegendPosition option, 524
Legendre command, 532, 657
Legendre polynomials, 487–494
Length command
Bessel functions, 472
bypassing algebra, 159, 163, 168
color wheels, 250, 259, 261
cross-check, 149, 152
curvature measurements, 189, 193
Dirac delta function, 217–218
discrete Fourier transforms, 417
electrical circuit problem, 326
Graphics'ContourPlot3D', 636–641
Graphics'MultipleListPlot, 559–560
Graphics'PlotField3D', 669
Graphics'Polyhedra', 680
gray scales, 239
higher dimensions, 171
imaginary roots, 271, 273
kernel commands, 521, 523, 619
Lattice field theory, 236
linear regression, 130, 142
ListPlotVectorField3D, 674

lists, 46–48
Lotka-Volterra equations, 359–360
Lessons learned, graphics primitives, 263
Lighting option, 640–642, 663
LightSources option
Graphics'Graphics3D', 644
kernel commands, 616
spherical harmonics, 297
Light source variation, 297, 616
Limaçon, 541–542
Limit command
Bessel functions, 469–470
bypassing algebra, 160
sampling function, 285
z-transform, 419–421
LinearAlgebra'MatrixManipulation' add-on package, 586–587
LinearAlgebra package, *66*
Linear integral transform properties, 370
Linear regression
algebra bypass, 159–170
cross-check, 144–145
derivation, 128–129
error propagation, 180
higher dimensions, 171–180
higher powers, 156–170
list-based tools, 133–137
precomputation, 170
Line command
Dirac delta function, 215, 224
entering data, 26
graphics primitives, 515
tick marks, 243
Line operator, 226, 228, 244
Lines, graphics primitives, *498*
List-based tools, 133–137
ListContourPlot command
contour plots, 279–280
kernel commands, 520–521
pictionary, *500*
ListContourPlot3D package, 640
ListDensityPlot command, *501,* 522–523
ListFilledPlot, *503*
ListPlot command
color wheels, 251
Dirac delta function, 226–227
discrete Fourier transforms, 415
displaying graphics, 213–214
Graphics'Graphics3D', 651, 654
Graphics'Spline', 564
Help Browser resource, 251
imaginary roots, 274–275
kernel commands, 519–520
Monte Carlo Integration, 446
pictionary, *500*
ListPlot3D command
kernel commands, 616–621, 626
three-dimensional plotting, 290–291, *292*
List plots, 2D plotting, 266–276
ListPlotVectorField command, 571–572
ListPlotVectorField3D, *609*
Lists
basics, 30
cautions, 635
concepts, 32–36
data structures, 22
generating, 30–32
manipulating, 36–44
merging, 45–46
operations, 45–51
usage, 32–36
ListScatterPlot3D command, 654–656
ListShadowPlot3D, *606*
ListShadowPlot3D command, 648–650, 653
ListSurfacePlot3D, *607*
ListSurfacePlot3D command, 657–659
LogLinearListPlot, *504*
LogLinearListPlot command, 539
LogLinearPlot, *504*
LogLinearPlot command, 538–539
LogListPlot, *504*
LogListPlot command, 538
LogLogListPlot, *504*
LogLogListPlot command, 540–541
LogLogPlot, *504*
LogLogPlot command, 540
LogPlot, *504*
LogPlot command, 537
Loops, accumulation, 60
Lotka-Volterra Equations, 354–364
Low level graphics rendering, 249
Lr Module, 560
Lsrc option, *602*

M

MachineError command, 594
Machine noise, 22, 179
Macintosh computer, 21
MagneticFluxQuantum, 77
Makedir Module, 259
MakeTree command, 586
Manipulation, lists, 36–44
Map command
  Dirac delta function, 226
  gray scales, 240
  imaginary roots, 274
  shorthand, 48
Masking function, 288
Mass, higher order differential equations, 328–336
Mass-spring system, 336
Mat function, 491
MathCode, 23
Mathematica
  add-on packages, 65–78
  augmentation, Wolfram material, 1–3
  basics, 1, 4–5, 96–97
  boneyard, 92–96
  control loss, 16–17
  crashes, 16
  data structures, 22–53
  front end control loss, 16
  function polymorphism, 43
  kernel control loss, 16
  learning curve, 4
  Mathematica Journal, 91–92
  memory usage, 17
  notebooks, 5–17
  older versions, 5
  packages included, *66–67*
  palettes, 78–91
  previous versions, 5
  programming, 53–65
  resources, 91–96
  speed, 4–5
  upgrading, 5
Mathematica Journal, 2, 91–92
Mathematical functions, 29
Mathematical notation, entering, 21
Matrices, *see* LinearAlgebra'MatrixManipulation' add-on package
  basics, 33MatrixForm command
  bypassing algebra, 166, 168–169
  coupled circuits, 394
  cross-check, 150, 152, 154
  Gamma function, 460, 463
  Legendre polynomials, 491
  list-based tools, 137
  lists, 50
  Lotka-Volterra equations, 360
  manipulating lists, 42–43
  singular matrices and inversion, 119, 121–122, 124, 126–127
MatrixManipulation'MatrixPlot, *511*
MatrixPlot command, 586–587
Max function
  DiscreteMath'ComputationalGeometry', 584
  kernel commands, 528, 620, 626–627
  random walk, 266
McGervey, J.D., 297
Memory, curvature measurements, 205
Mercator parameterization, 633, 662–663
Mersenne Prime, 27–29
Mesh option
  Graphics'Graphics3D', 644
  kernel commands, 522–523, 529, 613, 617, 632
  masking function, 288
  pictionary, *601*
  sampling function, 287
MeshRange option, 641–642
MeshStyle option, *601,* 613
Metropolis, 442
Microscope command, 594
Microscope'Microscope, *513*
Microscope'MicroscopeError, *513*
MicroscopicError command, 594
Microsoft Windows, 21
Min function
  Bessel functions, 473
  DiscreteMath'ComputationalGeometry', 584
  kernel commands, 620, 626–627
Miscellaneous'Audio', 68–69
Miscellaneous'BlackBodyRadiation', 75–78, 587
Miscellaneous'Calendar', 69–72
Miscellaneous'ChemicalElements', 72–74, 84

Miscellaneous package, graphic primitives, *66*
Miscellaneous'PhysicalConstants', 77–78
Miscellaneous'ResonanceAbsorptionLines', 588
Miscellaneous'StandardAtmosphere', 588–589
Miscellaneous'WorldPlot', 589–591
Möbius Strip, *611,* 686–687
Modal fit, 180
Modules
  accum, 142
  analyze, 143
  average, 148
  color wheels, 259, 261
  constraint, 96
  cramer, 396
  cross-check, 148
  ghost, 94–95
  Graphics'MultipleListPlot, 560
  higher dimensions, 173
  leading spaces, 14
  linear regression, 130–132, 142–143
  makedir, 259
  nonexistent, 94–95
  parity, 485
  publish, 143
  quadratic equation, 101–103
  regression, 129–132
  syntax, 101–102
MolarGasConstant, 77
MolarVolume, 77
Moments function, 486–487
Monolith, 516
Monomials, 180
Monte Carlo integration, 442–449, *449*
Monte Carlo Method, 442
Moore-Penrose Generalized Matrix Inverse, 120, 126
Moreblue option, *603,* 618
Morered option, *603,* 618
Multidimensional integrals, 432
MultipleListPlot, *506–507,* 562, *see also* Graphics'MultipleListPlot'
MultipleListPlot command, 556–558, 561–562
Multivariate expressions, 430–434
MuonGFactor, 77
MuonMagneticMoment, 77
MuonMass, 77
Music, 68
Myths, 4–5

N

Needs command, 444
Negative decrement, 31–32
Negative index, 74, 676
Negative referencing, 677
Nesting, 55–56, 342
NeutronComptonWavelength, 77
NeutronMagneticMoment, 77
NeutronMass, 77
Newton's law, convective cooling, 307–313
Noise environment, 649–651
Nonconservative forces, 567
Non-existant modules, 94–95
Normal command, 471–472
Norm command, 618
Normfac variable, 632
NotebookLauncher palette, *21, 81*
Notebooks
  basics, 5
  bugs, front end, 12–16
  crashes, 16–17
  curvature measurements, 205
  distributed computing, 12
  extraneous characters, 15
  kernel, blinding, 15–16
  leading spaces, 12–15
  seed, 6–11
NotebookSave command, 11
Nuclear reactor, Bessel functions, 468–474
Null
  common settings, 9
  directory structure, 9
  extraneous, cautions, 15
  leading spaces, 13–16
  ListPlot3D, 291
  masking function, 288–289
  ScatterPlot3D, 291
NumberTheory package, *66*
NumericalMath'Approximations, 592–593
NumericalMath'BesselZeros', 473
NumericalMath'Butcher', 591–592

NumericalMath'Microscope', 593–594
NumericalMath'OrderStar', 592–593
NumericalMath package, *66–67*
NumericalMath'SplineFit', 563, 566

O

Oceania, 589–590
Octahedron, 676–677, 679
Odd function, 300, 485–486
ODE solution, 375–379
On-line Help Browser, *see* Help Browser
On-line Web sites, *see* World Wide Web
OpenAuthorTools palette, *21, 81*
Open truncation, *610,* 684–686
% operator, 246–247
Options
  Plot command, 264
  SphericalPlot3D, 292
OrderStar command, 593
OrderStar'OrderStar', *512*
Orientation of figures, 281
Origin of plot, 518
OR operator, 553–554
Orthogonal functions, 490–491, 612
Osculating Circle, 185
Out operator, 352
OutputForm command, 24
Outputs
  file types, 210–211, *211*
  precision, cautions, 196
  sorting, Dirac delta function, 220
  storing, curvature measurements, 205–206

P

$Packages, 86–87
Packages, currently loaded, 86
Padé Approximant, 593
Padé approximation, 592–593
PairwiseScatterPlot command, 597
Palettes, *see also specific palette*
  constants, 86–91
  Demo palettes, 82–91
  Help Browser resource, 81
  periodic table of the elements, *82,* 82–85
  RGB color selector, 85–86
  standard, 17–18, *19–21,* 78–81, *79–81*
  stickiness, 18, 78
Parachute jumper exercise, 331–336
Parallel processing add-on package, 12
Parameter space, 132–133
ParametricPlot command
  kernel commands, 531
  Lotka-Volterra equations, 357–358
  phase plane portrait, 367
  pictionary, *502*
  systems of equations and phase plots, 349
ParametricPlot3D, 599, 628, 633–634
ParametricPlot3D command
  Graphics'Graphics3D', 655, 658
  Graphics'ParametricPlot3D', 663
  kernel commands, 628–630, 633–634
Params variable, 617
Pareto format, 595
Parity, 300
Parity Module, 485
Parity states, rotation, *299,* 299–301, *301*
Parseval's Theorem, 412–414
Part command, 73, 353
Partial Fraction Decomposition, 377
Partition command
  cross-check, 151–152
  DiscreteMath'Combinatorica', 580
  Graphics'Polyhedra', 682–685
  Graphics'Shapes', 686
  kernel commands, 529–530, 619–621, 625–627
Pascal, Blaise, 541
Pascal, Étienne, 541
PasteForm, 87–88
Paste Symbol, 87
Paste Value, 87
PeakWavelength command, 75
PercentileBarChart, *505*
PercentileBarChart command, 546–547
% operator, 246–247
Periodic table of the elements, *82,* 82–85
Phase plane portrait, 364–367
Phase plots, 347–350
Physical constants, 78, 91
Physical memory, 205

Pictionary, 2D plot types
basics, 497
ContourPlot, *500–501*
DensityPlot, *501–502*
DiscreteMath'Combinatorica, *510*
DiscreteMath'ComputationalGeometry, *511–513*
DiscreteMath'Tree, *511–513*
Graphics add-on package, *503, 503–509*
Graphics'FilledPlot', *503*
Graphics'Graphics tools, *504*
Graphics'ImplicitPlot', *503*
Graphics'InequalityPlot', *505–506*
Graphics'MultipleListPlot', *505, 505–507*
graphics primitives, 280–281, *498*
ListContourPlot, *500*
ListDensityPlot, *501*
ListPlot, *500*
ParametricPlot, *502*
Plot, *499*
PlotField, *508–509*
Raster, *502*
Spline, *507*
Pictionary, 3D plot types
basics, 599, *601–611*
coloring options, *602*
ContourPlot3D, *605*
Graphics'Grahics3D', *606–607*
Graphics'PlotVectorField3D', *609*
Graphics'Polyhedra' package, *610*
ParametricPlot3D package, *604, 608*
Plot3D, *601*
Project command, *606*
resolution, *602*
shading options, *603*
Shapes package, *611*
PieChart command, 548–549
PieExploded option, 550
Pitfalls, *see* Cautions
PlanarGraphPlot command, 581–582
PlanckConstant, 77
PlanckConstantReduced, 77
PlanckMass, 77
Planck's constant, 495
Platonic Solid, 675
Plot command
Bessel functions, 466, 468, 470
circuit problem, 381
coupled circuits, 385, 397
damped mass-spring system, 346
differential equations, 306–307, 310, 312
Dirac delta function, 223, 231–232, 234, 339–340
displaying graphics, 212–213
electrical circuit problem, 326
extraneous characters, 15
Fourier Transform, 399–401, 404–407
Gamma function, 454–455
Graphics'Graphics3D', 657
Graphics'InequalityGraphics', 555
heat conduction, cylinder, 343–344
higher order differential equations, 317–320, 322, 324
Inverse Laplace Transform, 375
kernel commands, 517–519, 531
Legendre polynomials, 489
mass in gravitational field, 331, 335
pictionary, *499*
polynomials and rational functions, 429–430
quadratic equation, 104
radio frequency pulse, 408–411
Riemann zeta function, 479, 481–484
second order equation, 328
two-dimensional plotting, 263–265
Plot3D command
kernel commands, 612, 614
sampling function, 282–283, 286
three-dimensional plotting, 281, 286–290, *290*
PlotField, *508–509*, *see also* Graphics'PlotField'
PlotField'ListPlotVectorField, *508*
PlotField'PlotGradientField, *508*
PlotField'PlotHamiltonianField, *508*
PlotField'PlotPolyaField, *508*
PlotField'PlotVectorField, *508*
PlotGradientField, *609*
PlotGradientField command, 569–570
PlotGradientField3D command, 670–672
PlotHamiltonianField command, 569
Plot improvements, 521
PlotJoined option, 267, 557, 565
PlotLabel option
basics, 264
Legendre polynomials, 489
radio frequency pulse, 408
random walk, 268

PlotLegend option, 555
Plotpoint format command, 548
PlotPoints option
  basics, 264
  contour plots, 277–278
  CylindricalPlot3D, 667
  Gamma function, 454
  Graphics'ContourPlot3D', 636–638, 640
  Graphics'Graphics3D', 644, 646
  Graphics'PlotField3D', 671–672
  Graphics'SurfaceOfRevolution', 691
  kernel commands, 519, 521–524, 529, 614, 633–634
  masking function, 288
  resolution, 289–290
  sampling function, 283–284, 286–287
  spherical harmonics, 297
PlotRange option
  basics, 264
  differential equations, 310
  Dirac delta function, 223, 340
  Fourier Transform, 400, 407
  Graphics'FilledPlot', 532–533
  Graphics'Graphics3D', 651, 654
  Graphics'InequalityGraphics', 555
  Graphics'MultipleListPlot, 563
  imaginary roots, 275–276
  kernel commands, 517, 522
  Lotka-Volterra equations, 357
  masking function, 288
  Monte Carlo Integration, 448
  phase plane portrait, 367
  radio frequency pulse, 410–411
  Riemann zeta function, 483
  sampling function, 282–283, 286–287
  systems of equations and phase plots, 349
PlotStyle option
  Bessel functions, 466, 470
  coupled circuits, 385, 397
  damped mass-spring system, 346
  Dirac delta function, 234
  Graphics'Graphics3D', 654–655, 657
  Graphics'Spline', 565
  imaginary roots, 275–276
  kernel commands, 517–518
  Legendre polynomials, 489
  mass in gravitational field, 331
  Monte Carlo Integration, 446
  Riemann zeta function, 479
PlotSymbol command, 563
Plotting
  function lists, 517–518
  graphics primitives, 263–298
PlotVector command, 570–572
PlotVectorField command, 350, 567–568
PlotVectorField3D, *609,* 672–675
PlotVectorField3D command, 668–669
Plot with legend, *506*
Plus operator
  bypassing algebra, 167
  color wheels, 262
  cross-check, 149–152
  curvature measurements, 198
  higher dimensions, 174–175
  inverse Laplace transform, 379
  linear regression, 138–141
  singular matrices and inversion, 127
Point function, 514
PointParametricPlot3D, *608,* 663–665
Points
  graphics primitives, *498*
  labeling, 216–217
  plotting, 514
PolarListPlot, *504*
PolarListPlot command, 542
PolarMap command, 574–575
PolarPlot, *504*
PolarPlot command, 542
Polya fields, 570
Polyhedron, *610*
Polyhedron command, 676–685
Polymorphism, 32
Polynomials
  basics, 33–36
  Gamma function, 458–465
  Help Browser resource, 270
  integration, 424–430
  Legendre type, 487–494
Position command, 392–393
Position of lights, 616
Positive referencing, 677
Positive stellation, *610*
Potentials, 670–671, 673–675
Pound sign (#), 48
Precision
  data structures, 22
  erosion, cautions, 60
  recursion, 179

reducing magnitude, problems, 168
Precomputation, 170
Prepend command, 487
Prep module, 262
Prime command, 36
Print command
Bessel functions, 468–470, 472, 476–477
cross-check, 148, 153–154
curvature measurements, 198
differential equations, 314–315
Dirac delta function, 216, 220
Do loop, 54–56, 59–60
enormous equations, 207
Gamma function, 453, 459–460, 463
gray scales, 239–240
higher order differential equations, 321, 324
large numbers, 28–30
Legendre polynomials, 493
linear regression, 132
Lotka-Volterra equations, 363
mass in gravitational field, 331
Monte Carlo Integration, 446
phase plane portrait, 366
pure function, 48
quadratic equation, 100, 102, 111
random walk, 266
Riemann zeta function, 484–485
seed notebooks, 6–7
spherical harmonics, 294
While loop, 63–64
PrintIntegers command, 94
Print Screen command, 104
Programming
basics, 53
Do loop, 53–61
flow control, 53–65
Help Browser resource, 53
For loop, 61
While loop, 61–65
Project, *607*
Project command, 659
Proportioning, 612
ProtonComptonWavelength, 77
ProtonMagneticMoment, 77
ProtonMass, 77
Pseudoinverse, 120–121
PseudoInverse command, 121
Publish Module, 143
Punctuation, 239–240
Pure functions
bypassing algebra, 163
Dirac delta function, 220–221
Help Browser resource, 156
higher dimensions, 172
imaginary roots, 273–274
lists, 48

Q

QSolve variable, 93, 103
Quadratic equation
arbitrary orders, 117–118
basics, 99–106
symbolic solution, 106–117
The CRC Encyclopedia of Mathematics, 690
Quantile, 597
QuantilePlot command, 597
QuantizedHallConductance, 77
Quantum electrodynamics, 235
Queries, 24
Quintic equation, 117

R

Radioactive decay, 313–316
Radio frequency pulse, 408
Random command
DiscreteMath'ComputationalGeometry', 581–584
Graphics'Graphics', 545, 548, 550
Graphics'Graphics3D', 650, 652
Graphics'MultipleListPlot, 561
LinearAlgebra'MatrixManipulation', 587
manipulating lists, 41–42
Monte Carlo Integration, 444
PlotVectorField3D, 672–673
random walk, 266
Statistics'StatisticsPlot', 595–597
While loop, 63
RandomGrays command, 590–591
RandomPartition command, 577

RandomTree command, 579
Random walk, 266–268
Range command
  DiscreteMath'Tree', 586
  generating lists, 31–32
  linear regression, 138
  manipulating lists, 36
RankedEmbedding command, 579
Raster, pictionary, *502*
Raster command, 524, 528–530
Rational functions, 424–430
Real command, 273
Real operator, 272
Reals
  curvature measurements, 183–184, 192
  Graphics'Graphics3D', 644
  leading spaces, 14
  spelling checker, 95
  spherical harmonics, 295–296
Rebbi, Claudio, 235
Rebooting computer, 12
Rectangle command, 237–241, 443
Rectangles, graphics primitives, *498*
Recursion, 176
Recursive function, 175–178
Redcold option, *602,* 615–616
Redhot option
  Graphics'Graphics3D', 646–650, 653
  Graphics'PlotField3D', 671
  kernel commands, 615–618, 621, 624–627, 632–633
  pictionary, *602–603*
Refine command, 183–184
Relative errors, 653–654
Rendering, graphics, 249, 288
ReplaceAll operator
  Dirac delta function, 337–338
  electrical circuit problem, 325
  Lotka-Volterra equations, 356
  periodic table of the elements, 84
  systems of equations and phase plots, 348
ReplacePart command
  coupled circuits, 392, 394
  higher dimensions, 173
Representations, data structures, 22–24
Resolution, kernel commands, 521–522, 613–614
Resonance absorption lines, 588
ResonanceAbsorptionLines'ElementAbsorptionMap, *512*
ResonanceAbsorptionLines'WavelengthAbsorptionMap, *512*
Resources
  articles, 91–92
  boneyard, 92–96
  Mathematica Journal, 91–92
  In and Out column, 92
  Tricks of the Trade column, 92
Result variable, 304–305, 307–308
Return command, 259, 560
RGB color
  basics, 209–210
  Help Browser resource, 85, 210
  palettes, 85–86
  sum, 263
RGBColor command
  color wheels, 257–259, 261
  Graphics'Graphics3D', 644
  kernel commands, 614–616
  pictionary, *602*
  spherical harmonics, 297
Riemann zeta function, 485
  basics, 478–487
  Graphics'ComplexMap', 574–575
  Graphics'PlotField', 570
Riera, P., 60
Ring torus, 686, *see also* Torus
RMS, errors, 416–417
Root command, 117
Roots, quadratic equation, 106–111
Rotation, parity states, *299,* 299–301, *301*
Rule Delayed operator, 270
Runge-Kutta methods, 592
RydbergConstant, 77
Ryzhik, Gradshteyn and, studies, 4, 442

S

SackurTetrodeConstant, 77
Sampling function, 281–292, 520
Save statement, 10–11
Scalar function, kernel commands, 631
Scalar Triple Product, 174
ScaleFactor option, 675

ScatterPlot3D, 291–292, *607*
ScatterPlot3D command, 654–655, 664
Score function, 448
Screen display default, 196
Sec[] function, 435
Second order equations, 327–328
Seed notebooks, 6–11
SeedRandom[] command
  DiscreteMath'Combinatorica', 578
  DiscreteMath'ComputationalGeometry', 581–584
  displaying graphics, 213
  Graphics'Graphics', 545–546, 548–551
  Graphics'Graphics3D', 650, 652
  Graphics'MultipleListPlot, 560–561
  LinearAlgebra'MatrixManipulation', 587
  manipulating lists, 41–42
  Miscellaneous'WorldPlot', 589–590
  Monte Carlo Integration, 444
  PlotVectorField3D, 673
  random walk, 266
  square root operator, 44
  Statistics'StatisticsPlot', 595–597
  While loop, 63, 65
Select command, 473
Separable potential, 670–671
Sequence command
  color wheels, 258–259, 261
  Graphics'Graphics', 544, 546–547
  Graphics'InequalityGraphics', 553
Series command, 471
Set command
  basics, 93
  higher dimensions, 175
  imaginary roots, 272
  Lotka-Volterra equations, 354
  seed notebooks, 7
SetDelayed command
  basics, 93
  higher dimensions, 175
  Lotka-Volterra equations, 356, 358
Shading function
  curvature, *603*
  kernel commands, 518–520, 530, 613, 617–622, 625–628
  Legendre polynomials, 488–489
Shading option
  altitude, 617–618
  cautions, 643
  CylindricalPlot3D, 667
  Graphics'ParametricPlot3D', 663
  Graphics'SurfaceOfRevolution', 691
  gray scales, 239
  kernel commands, 613
  pictionary, *601*
  sampling function, 282–283, 286
Shadow, pictionary, *606*
Shadow command, 643–644
ShadowMesh option, 646, 648–650, 653
ShadowPlot3D, *606*
ShadowPlot3D command, 645–646
ShadowPosition option, 646, 648–650, 653
Shadow routine, 643
Shell plot, 222–223
Show command
  color wheels, 251–252, 254, 256, 258, 260
  contour plots, 277
  coupled circuits, 397
  damped mass-spring system, 347
  Dirac delta function, 216–218, 220–224, 228, 230
  disks, 246–249
  displaying graphics, 213–214
  electrical circuit problem, 326
  Graphics'FilledPlot', 533
  Graphics'Graphics3D', 656–657, 660
  Graphics'MultipleListPlot, 562
  Graphics'Polyhedra', 676–682, 684–685
  graphics primitives, 514–516
  Graphics'Shapes', 686
  Graphics'Spline', 565–566
  gray scales, 241–242
  helicoids, 689–690
  helix, 688
  kernel commands, 517, 520, 528, 530, 612–616, 618, 629
  Lattice field theory, 236
  Miscellaneous'Audio', 68
  Möbius Strip, 687
  Monte Carlo Integration, 443, 446
  random walk, 267–268
  sampling function, 282
  spherical harmonics, 296
  SphericalPlot3D, 666
  systems of equations and phase plots, 350
  tick marks, 243–245
ShowGraphArray command, 580

ShowGraph command, 577–579
ShowLegend command, 524
Signal function, 419–420
Signal modulation, transforms, 405–408
Sign function, 401, 409–410
Simplify command
  coupled circuits, 396
  curvature measurements, 182–184, 192, 198, 200–201
  Fourier Transform, 406
  fundamental constants, 88–89
  Gamma function, 460
  Graphics'Graphics3D', 644
  higher order differential equations, 322, 324
  Inverse Laplace Transform, 375
  kernel commands, 625, 627
  leading spaces, 14
  multivariate expressions, 433
  polynomials and rational functions, 428–429
  quadratic equation, 103
  Riemann zeta function, 480, 485
  rotation, parity states, 300
  sampling function, 287
  second order equation, 328
  spherical harmonics, 295–296
  z-transform, 419, 421
Simulated Annealing, 442
Sinc function
  discrete Fourier transforms, 414–415
  Fourier Transform, 400
  Graphics'FilledPlot', 533–535
  Graphics'Graphics3D', 645–649, 652
  Graphics'InequalityGraphics', 555
  Graphics'MultipleListPlot, 556–558
  kernel commands, 520–523, 525–526, 529
  sampling function, 281–282, 284–285
  The CRC Encyclopedia of Mathematics, 286, 520
Sine Integral, 525
Sin[] function
  color wheels, 250
  contour plots, 278
  Dirac delta function, 336
  Gamma function, 455
  Graphics'ComplexMap', 573
  Graphics'FilledPlot', 532, 535
  Graphics'Graphics3D', 646, 649, 655
  Graphics'MultipleListPlot, 556–558, 563
  Integrate command, 441
  kernel commands, 517–520, 525, 531, 629–631, 634
  Legendre polynomials, 490, 492–493
  manipulating lists, 36, 40
  multivariate expressions, 434
  radio frequency pulse, 412
Singular elements, cautions, 635
Singular matrices and inversion, 118–127
Singular Value Decomposition, 123
SingularValues, 123, 126
SingularValues command, 124
SingularValuesDecomposition command, 126
Sinh[] function, 436
Slot operators, 48, 617
Small dimensions, 44
SmallStellatedDodecahedron, 676–677, 679
SolarConstant, 77
SolarLuminosity, 77
SolarRadius, 77
SolarSchwarzschildRadius, 77
Solve command
  arbitrary orders, 117
  Bessel functions, 472, 474
  critical points, 351–352
  curvature measurements, 189
  differential equations, 309–310, 314
  imaginary roots, 269
  Laplace Transform, 376, 380
  Lotka-Volterra equations, 354–358, 360
  mass in gravitational field, 330, 335
  quadratic equation, 107–108, 111
Solving differential equations, 303–367
Sound, 68
Spaces, leading, 12–15
Special functions
  basics, 451
  cylindrical cavity, 474–478Bessel functions, 465–478, *467*
  Fermi-Distribution, 480–487
  Gamma function, 451–465
  harmonics, spherical, 495
  Legendre polynomials, 487–494
  nuclear reactor, 468–474
  polynomials, 458–465, 487–494
  Riemann zeta function, 478–487

spheres, 458–465, 490–494
TE_2 modes, 474–478
Speed, software, 4–5
SpeedOfLight, 77, 87, 90
SpeedOfSound, 77
Spelling checker, 95
Sphere
curvature measurements, 181–182
Gamma function, 458–465
Legendre polynomials, 490–494
pictionary, *604*
Sphere command, 689–690
Spherical coordinates, 628–629, 631
Spherical Harmonic, 297
SphericalHarmonicY, 14, 294–296, 644
SphericalPlot3D, 292–298, *293, 608*
SphericalPlot3D command, 644–645, 665–666
SphericalRegion option
cautions, 281
CylindricalPlot3D, 667
Graphics'ContourPlot3D', 635, 637–638, 640–642
Graphics'Graphics3D', 643–644, 646, 648–650, 653, 655–658, 661
Graphics'ParametricPlot3D', 663
Graphics'PlotField3D', 668–672
Graphics'Polyhedra', 678–685
Graphics'Shapes', 686
Graphics'SurfaceOfRevolution', 691
helicoids, 689–690
helix, 687–688
kernel commands, 613–614, 617, 629–630, 633–634
ListPlotVectorField3D, 673, 675
Möbius Strip, 687
pictionary, *601*
PointParametricPlot3D, 664–665
sampling function, 282–283, 286
SphericalPlot3D, 666
Spindle torus, 658–659, 686
Spline, *507,* 564, *see also* Graphics'Spline'
Square root operator, 44–53
StackedBarChart command, 546
StackGraphics, *607*
StackGraphics command, 656–657
Stamp, 9–11
StandardAtmosphere'AtmosphericPlot, *512*
StandardForm command, 23
Standard packages, 67
Standard palettes, 17–18, *19–21,* 78–81, *79–81*, *see also* Palettes
Stationary Point, 351
Statistics'ContinuousDistributions', 444
Statistics package, *67*
Statistics plots, 595–597
StatisticsPlots'BoxWhiskerPlot, *513*
StatisticsPlots'PairwiseScatterPlot, *513*
StatisticsPlots'QuantilePlot, *513*
Statistics'StatisticsPlot', 595–597
StefanConstant, 77
Stellate command, 682
Stellated objects, 680
Stellation, 680–683
Step++ command, 59
Subrule, 194, 270
Substitution rules, 108–109, 193–194
Sum command, 138–139, 636–641
Sum of squares, 141
SurfaceColor command, 632–633
Surface color variation, 297, 616
SurfaceMesh option, 646, 648–650, 653
Sweep function, 472, 623–624
Symbolic explorations, 126–127
Symbolic solution, 106–117
SymbolLabel option, 563
SymbolShape option, 562–563
Symbols (in book), 3
Systems of equations, 347–350

T

Table command
arbitrary orders, 117
Bessel functions, 466
blinding the kernel, 16
bypassing algebra, 161–163, 167, 170
color wheels, 250–252, 260
coupled circuits, 388
curvature measurements, 193, 195–198, 200
differential equations, 315–316
Dirac delta function, 217–220, 224–225, 230, 232–234
discrete Fourier transforms, 415

DiscreteMath'ComputationalGeometry', 581–584
displaying graphics, 213
electrical circuit problem, 326
entering data, 26
extraneous characters, 15
Gamma function, 452–453, 460, 463
generating lists, 31
Graphics'ContourPlot3D', 641–642
Graphics'FilledPlot', 532, 534–535
Graphics'Graphics', 538–542, 545, 548–551
Graphics'Graphics3D', 647–657, 659, 661
Graphics'ImplicitPlot', 536
Graphics'MultipleListPlot, 556–559, 561–563
Graphics'PlotField', 571–572
Graphics'PlotField3D', 669
Graphics'Polyhedra', 680, 682–685
graphics primitives, 514–515, 517
gray scales, 237–238
higher dimensions, 173–174, 176
higher order differential equations, 320
imaginary roots, 273–274
kernel commands, 518–523, 527, 617, 619
Lattice field theory, 236
Legendre polynomials, 488, 491, 493
LinearAlgebra'MatrixManipulation', 587
ListContourPlot, 280
ListPlot3D, 291
ListPlotVectorField3D, 674–675
lists, 45–47
Lotka-Volterra equations, 357, 360, 362–364
manipulating lists, 36, 41–43
Monte Carlo Integration, 447
phase plane portrait, 365–366
PlotVectorField3D, 672–673
quadratic equation, 110–111
Riemann zeta function, 484, 487
SphericalPlot3D, 666
Statistics'StatisticsPlot', 595–597
systems of equations and phase plots, 348
TableForm command
differential equations, 315–316
Gamma function, 452–453
Legendre polynomials, 488
Miscellaneous'Calendar', 70
Riemann zeta function, 487
Take command, 74
Tan function, 440
Taylor expansion, 458
Taylor monomials, 180
TE_2 modes, 474–478
Tesselation projection, *610*
Test function, 194–196
Tetrahedron, 676–677, 679
TeXForm command, 23
Text, 218, 246, 251
Text command
color wheels, 251–253
coupled circuits, 397
Dirac delta function, 217, 219–222
tick marks, 245
TextListPlot command, 550
TextStyle command, 252
$TextStyle command, 9
The CRC Encyclopedia of Mathematics
Adjacency List, 585
Affine Transformation, 690
Arithmetic Series (formula 3), 58
Bessel Differential Equation, 465
Bessel Function of the First Kind, 465
Bessel Function of the Second Kind, 465
Bézier Spline, 566
Box-and-Whisker Plot, 595
Buffon-Laplace Needle Problem, 442
Buffon Needle Problem, 442
Conservative Field, 568
Convex Hull, 582
Correlation Index, 129
Cramer's Rule, 383
Critical Point, 353, 359
Cubic Spline, 564
Curvature, 181, 625
Delaunay Triangulation, 581
Delta Function, 215, 337
Distribution (Generalized Function), 215
Divergence, 633
Dodecahedron, 680
Dot Product, 174
Ellipsoid, 536, 630, 633–635
Embedding, 580
Euler Formula, 413
Factorial, 371
Ferrers Diagram, 577

Fourier Cosine Transform, 413
Fourier Sine Transform, 413
Fourier Transform, 402–403
Function, 156
Gabriel's Horn, 691
Gamma Function, 371
Geodesic Dome, 683
Golden Ratio, 242
Gradient, 626, 628
Gram-Schmidt Orthonormalization, 612, 615
Graph, 577–580
Hankel Function, 465
Heaviside Calculus, 370
Heaviside Step Function, 382
Helicoid, 687–688
Helix, 628, 687
high-order Runge-Kutta methods, 592
icon, 3
Icosahedron, 678
Integral Transforms, 370
Laplacian, 621
Least Squares Fitting, 129
Limaçon, 542
Lotka-Volterra Equations, 359
Mersenne Prime, 29
Möbius Strip, 687
Monte Carlo Integration, 442
Monte Carlo Method, 442
Moore-Penrose Generalized Matrix Inverse, 120, 126
Osculating Circle, 185
Padé Approximant, 593
Parity, 300
Parseval's Theorem, 412
Partial Fraction Decomposition, 377
Platonic Solid, 675
Pseudoinverse, 120
Quadratic Equation, 690
Quantile, 597
Quintic Equation, 117
Recursion, 176
Runge-Kutta methods, higher-order, 592
Scalar Triple Product, 174
Sign, 401
Sinc Function, 286, 520
Sine Integral, 525
Singular Value Decomposition, 123
Spherical Coordinates, 629
Spherical Harmonic, 297
Spline, 564
Stationary Point, 351
Stellation, 682–683
Torus, 659, 665, 686
Tree, 578–579, 586
Triangulation, 581
Truncated Cube, 685
Uniform Polyhedron, 675
Vector Triple Product, 174
Vertex (Graph), 585
Voronoi Diagram, 585
Zernike Polynomials, 300–301
Zeta Function, 570
Z-transform, 417
Thickness, 26, 228–230, 555
ThomsonCrossSection, 77
3D Graphics, 281
Three-dimensional plotting
basics, 281
graphics primitives, 281–298
harmonics, spherical, 293–297, *298*
ListPlot3D, 290–291, *292*
Plot3D, 281, 286–290, *290*
sampling function, 281–292
ScatterPlot3D, 291–292
SphericalPlot3D, 292–298, *293*
Thresh function, 529–531
Tick marks, 242–245
Ticks option, 264, 691
TIFF files, 211
Timer, Do loops, 56–57
Timestamp, *see* Stamp
TimeUsed[] command
curvature measurements, 198
Gamma function, 460, 463
large numbers, 28
Monte Carlo Integration, 445–446
Timing[] function, 28, 189
Tophats
Dirac delta function, 233
graphics primitives, 231–234
radio frequency pulse, 409
Torus
Graphics'Graphics3D', 658–659
Graphics'Shapes', 686
pictionary, *611*
SphericalPlot3D, 665–666

The CRC Encyclopedia of Mathematics, 659, 665, 686
Torus command, 686
ToString[] operator, 43–44, 268
Total command, 416, 560
TotalPower command, 75
Touch tones, 68
TraditionalForm command, 24
Transforms
  basics, 369, *370*
  circuits, 379–397
  coupled circuits, 382–397
  discrete Fourier transforms, 414–417, *416*
  electrical circuits, 379–381
  filter representation, 408–411
  FourierCos transforms, 413–414
  FourierParameters option, 402–404
  FourierSin transforms, 413–414
  Fourier transforms, 398–417
  inverse Laplace transforms, 374–375
  Laplace transforms, 370–398
  linear integral transform properties, 370
  ODE solution, 375–379
  Parseval's theorem, 412–414
  radio frequency pulse, 408
  signal modulation, 405–408
  z-transform, 417–421
Transpose, lists, 50–51
Transpose command
  cross-check, 147, 150, 152–153
  Graphics'Graphics3D', 661
  Graphics'Spline', 565
  linear regression, 142
  ListContourPlot, 280
  ListPlotVectorField3D, 673
  singular matrices and inversion, 124, 127
Tree, 578–579, 586
TreePlot command, 586
Triangles, series, 222–223
Triangulation, 581
Tricks of the Trade column, 92
TrigToExp function, 410–411
Truncated Cube, 685
Truncation, 684–685, *see also* Open truncation
Two-dimensional graphic types, pictionary, 280–281
Two-dimensional plotting
  basics, 263
  contour plots, 276–280
  graphics primitives, 263–280
  imaginary roots, plotting, 269–276
  list plots, 266–276
  Plot command, 263–265
  random walk, 266–268
2001: A Space Odyssey, 516
Typos, common, 26

U

Ulp, *see* Units in last place (Ulp)
Underscore structure, 93
Uniform Polyhedron, 675
Units, 89, 91
Units in last place (Ulp), 594
UnitStep function
  coupled circuits, 386
  Dirac delta function, 231–232
  Fourier Transform, 403
  radio frequency pulse, 408, 412
  z-transform, 418
Unit steps, random walk, 268
Unreferenced data, 3
Upgrading *Mathematica* versions, 5
URLs, *see* World Wide Web
User warnings, *see* Bugs; Cautions
Utilities package, *67*
Utilities'ShowTime', 56–57

V

VacuumPermeability, 77, 87–91
VacuumPermittivity, 77, 87–91
Variables
  Help Browser resource, 270
  Miscellaneous'WorldPlot', 591
  underscore structure, 93
Vector arithmetic, 53
VectorHeads option, 668–673, 675
Vectors, 44, 52
Vector Triple Product, 174
Velocity, 328–336

Vertex (Graph), 585
Vertex list, 582–583
Viewpoint, kernel commands, 613
ViewPoint option, 613, 661
ViewVertical option, 691
Virtual memory, 205
Viscosity, function of altitude, 588–589
Voronoi Diagram, 585

W

Warnings, *see* Bugs; Cautions
Waveform command, 68
WavelengthAbsorptionMap command, 588
WeakMixingAngle, 77
Web, *see* World Wide Web
Which function, 65
While loop
  higher order differential equations, 319
  programming, 61–65
  random walk, 266
Wildcards, lists, 48
WireFrame, *611*
WireFrame command, 690
Wolf, Born and, studies, 33
Wolfram Research, 1–3, 592, *see also* World Wide Web
-WorldGraphics- object, 589–590
WorldPlot, *see* Miscellaneous'WorldPlot'
WorldPlot command, 590
WorldPlot'WorldPlot, *512*
WorldProjection option, 591
WorldRange option, 591
WorldRotation option, 591
WorldToGraphics option, 590–591
World Wide Web
  Easter table, 70
  gyromagnetic ratio, 235
  icon, 3
  keyboard shortcut, 21
  MathCode, 23
  packages included, 67
  Padé approximation, 593
  parallel processing add-in package, 12
  physical constants, 78
  quantum electrodynamics, 235
  Runge-Kutta methods, 592
  tutorials, 2
  units, 91

X

X gradient shading, *603*
XOR operator, 552–553

Y

Y gradient shading, *603*

Z

Zernike Polynomials
  basics, 33–36
  sampling function, 286–287
  ScatterPlot3D, 292
  SphericalPlot3D, 293
  The CRC Encyclopedia of Mathematics, 300–301
Zernike studies, 33, 301
Zeta function, 570, *see also* Riemann zeta function
Z-transform, 417–421