A First Course in Scientific Computing

Symbolic, Graphic, and Numeric Modeling Using Maple, Java, Mathematica, and Fortran90

RUBIN H. LANDAU

Contributors:

Robyn Wangberg (Mathematica),
Kyle Augustson (Fortran90),
M. J. Páez, C. C. Bordeianu,
C. Barnes

PRINCETON UNIVERSITY PRESS
PRINCETON AND OXFORD

Published by Princeton University Press,
41 William Street, Princeton, New Jersey 08540
In the United Kingdom: Princeton University Press,
3 Market Place, Woodstock, Oxfordshire OX20 1SY

Library of Congress Cataloging-in-Publication Data

Landau, Rubin H.
A first course in scientific computing: symbolic, graphic, and numeric modeling using Maple, Java, Mathematica, and Fortran90 / Rubin H. Landau; contributors, R. Wangberg, K. Augustson
. . . [et al.].
p. cm.
Includes bibliographical references and index.
ISBN 0-691-12183-4 (cloth : alk. Paper)
1. Science—Data processing. I. Wangberg, R, Augustson K. II. Title
2. Q183.9.L36 2005
3. 502′.85—dc22 2004062443

British Library Cataloging-in-Publication Data is available

The publisher would like to acknowledge the author of this
volume for providing the camera-ready copy from which this book was printed.

Printed on acid-free paper. ∞

pup.princeton.edu

Printed in the United States of America

1 3 5 7 9 10 8 6 4 2

For ***Loren Landau***
and your first book

Contents

Preface

This book contains an introduction to scientific computing appropriate for all lower-division college students. Its goal is to make students comfortable using computers to do science and to provide them with tools and knowledge they can utilize throughout their college careers. Its approach is to introduce the requisite mathematics and computer science in the course of solving realistic problems. On that account care is given to indicate how each discipline uses its own language to describe the same concept, how their tools are useful to us, and how computations are often concrete examples of abstract ideas.

This is easier said than done. On the one hand, lower-division students are simultaneously learning elementary mathematics and physics, and so this may be the first place they encounter the science and mathematics used in the problems. On the other hand, in order for the tools and techniques to be useful for more than the assigned problem, we give *more* than an introduction (the original title of this book) to the computational tools. We address the first issue in our teaching by reminding the students that our focus is on having them learn the techniques in the proper context, and that any new science and mathematics they become familiar with will make it easier for them in their other courses. We address the second issue by placing an asterisk * in the title of chapters and sections containing optional materials and by reminding the students of which sections are most appropriate for the problem at hand.

This book covers some of the basics of computation, numerical analysis, and programming from a computational science point of view. We want the reader to acquire some ideas of what is possible with computers, what type of tools there are for it, and how to go about getting all the pieces to work together. After that, it is easy to use on-line help or the references to get more details. As a result, our presentation is more practical and more focused on mathematics and science than an introductory programming or computer science text, with minimal discussion of computer science theory. The book follows our own personal preference for "just enough" computer information in that it avoids going through every option for every command and instead presents realistic examples.

We follow the dictum that science and engineering students learn computing best while sitting down at a computer in a trial-and-error mode. Hence, we adopt a tutorial approach in which readers work along with us in solving a problem, learn by doing, and then work on their own version of the problem (there are also additional exercises at the end of each chapter). We use the command-line mode for the compiled languages that makes the tutorial as universal as possible, and, we believe, is better pedagogically. It then follows that this book is closer to a workbook than a reference book. Yet because one always comes back to find worked examples of commands, it should be valuable for reference as well.

A problem solving environment such as *Maple* or *Mathematica* is probably the easiest way to start scientific computing, is natural to use with trial and error, and is what we do in Part 1. Its graphical interface is friendly, it shows the user a wide spectrum of what can be done with modern computation, such as symbolic manipulations, 2-D, 3-D visualization and linear algebra, and is immediately useful in other courses and for writing technical reports. After the first week with Maple or Mathematica, students with computer fright usually feel better. These environments also demonstrate how computers can give an immediate response in beautiful mathematical notation, or fail at what should be the simplest of tasks.

Part 2 of the text is on Java (or Fortran). We believe that learning to program in a compiled language teaches more of the basics of computation, gets closer to the actual algorithms used, teaches better the importance of logic, and opens up a broader range of technical opportunities (jobs) for the students than the use of a problem solving environment. In addition, compiled languages also tend to be more powerful and flexible for numerically intensive tasks, and students naturally move to them for specialized projects. Likewise, after covering Parts 1 and 2 of the text it may make sense for a student to use environments like *Matlab*, which combine elements of both compiled languages and problem solving environments.

Many students may find that the logic and the precision of language required in programming is more challenging than anything they have ever faced before. Others find programming satisfying and a natural complement to their study of mathematics and foreign languages. We try to decrease the slope of the learning curve by starting the neophytes with sample programs to run and modify, rather than requiring them to write all their own programs from scratch. This process is more exciting, saves a great deal of time otherwise spent in frustrating debugging, and helps students learn by example.

Even though it might not be evident from all the hype about Java and Web computing, Java is actually a good language for beginning science students. It demands proper syntax, produces useful error messages, is consistent and intelligent in handling precision, goes a long way towards being computer-system independent, has Sun Microsystems providing free program-development environments [SunJ], and runs fast enough for nonindustrial purposes (its speed is increasing, as are the number of scientific subroutine libraries being developed in Java).

Part 3 of the text provides a short $\LaTeX$ survival guide. A number of colleagues have suggested the need for such materials, and since using $\LaTeX$ is quite similar to compiling a code, it does make a useful extension of the text. Even though we do not try to reveal the full power and complexity of LaTeX, we do give enough of its basic elements for the reader to write beautiful-looking scientific documents.

Depending on how many chapters and modules are used, this book contains enough materials for a one- or two-semester course. Our course has one lecture and two labs every week, with roughly one instructor for every 10 students in lab. Attending lectures and reading the materials before lab are important in acquainting the students with the general concepts behind the exercises and in providing a broad picture of what we are trying to do. The supervised lab is where the real learning occurs.

We believe that a modern student should be acquainted with several approaches to scientific computing. Notwithstanding our avowed claim that there are multiple paths leading to good scientific computing, we have had to make some choices as to what to place in the printed version of this book and what to place on the CD. The basic ideas behind scientific computing are language independent, yet the details are not. For all these reasons we have decided to cover Maple, Java, and LaTeX in the printed version of this book, but to place other languages on the CD that comes with the text.

The CD is platform independent and has been tested on Windows, Macs, and Unix. The Java and Fortran programs are pure text. The Maple and Mathematica files, however, require the respective programs to execute them. The pdf files will require Abode Acrobat, which is free. Any difficulties with the CD should be reported to rubin@physics.orst.edu. Additions and corrections to the CD are found on our Web pages (http://www.physics.orst.edu/˜rubin/IntroBook) or through Princeton University Press Web pages (http://pup.princeton.edu/titles/7916.html).

Specifically, the CD contains Java programs, LaTeX files, data files, and various supplementary materials. As indicated, it also contains Maple worksheets with essentially identical materials as in the Maple section of the text but in an interactive format that we recommend over reading the paper version. Furthermore, the CD contains essentially identical materials to the Maple tutorials as Mathematica notebooks. These can be read with Mathematica or printed out as an alternative version of the text. Likewise, the CD also contains the Java materials of the text converted over to Fortran90, as well as the appropriate Fortran programs. Though we do not recommend trying to learn two languages simultaneously, having alternative versions of the text does present some interesting teaching possibilities. Additions and corrections to the CD are found on our Web pages.

Acknowledgements

This book was developed on seven year’s worth of students in the introductory scientific computing class at Oregon State University. The course was motivated by the pioneering text by Zachary [Zach 96] and encouraged by [UCES]. The course has been taught by Albert Stetz, David McIntyre, and the author; I am proud to acknowledge their friendship and the inclusion of their materials in the text. I have been blessed with some excellent student assistants without whose efforts the course could not be taught and the materials developed; these include Matt Des Voigne, Robyn Wangberg, Kyle Augustson, Connelly Barnes, and Juan Vanegas. Valuable materials and invaluable friendships have also been contributed by Manuel Páez and Cristian Bordeianu, coauthors of our *Computational Physics* text, and Sally Haerer. They have helped make this book more than the introduction it would have been otherwise. I look forward to their continued collaborations.

Financial support for developing our degree program in computational physics and associated curricular materials comes from the National Science Foundation's Course, Curriculum, and Laboratory Improvement program directed by Duncan McBride, and the Education, Outreach and Training Thrust area of the National Partnership for Computational Infrastructure, under the leadership of Gregory Moses and Ann Redelfs. The courses and materials have benefited from formative and summative assessments by Julie Foertsch of the LEAD Center of the University of Wisconsin. This work has also benefited from formative comments by various reviewers and colleagues, in particular Jan Tobochnik and David Cook. Thanks also goes to my editor, Vickie Kearn at Princeton University Press, who has been particularly insightful, encouraging, and courageous in helping me develop this multidisciplinary text and having it turn out so well, and to Ellen Foos for her caring production of the book. And Jan Landau, always.

RHL

Chapter One

Introduction

1.1 NATURE OF SCIENTIFIC COMPUTING

Computational scientists solve tomorrow's problems with yesterday's computers; computer scientists seem to do it the other way around.

—anonymous

The goal of scientific computing is problem solving. The computer is needed for this, because real-world problems are often too difficult or complex for analytic or human solution, yet workable with the computer. When done right, the use of a computer does not replace our intellect, but rather leverages it by providing a super-calculating machine or a virtual laboratory so that we can do things that were heretofore impossible.

The mathematical modeling and problem-solving orientation of scientific computing places it in the discipline of *computational science*. In contrast, computer science, which studies computers for their own intrinsic interest, provides the underpinning for the development of the hardware and software tools that computational scientists use. As illustrated on the left of Figure 1.1, computational science is a *multidisciplinary* field that combines a traditional discipline, such as physics or finance, with computer science and mathematics, without ignoring the rigor of each. Books such as this, which employs materials from multiple fields, aim to be the central bridge in this figure, connecting and drawing together the three fields.

Studying a multidisciplinary field is challenging. Not only must you learn more than one discipline, you must also work with the separate languages and styles of the different disciplines. To illustrate, a computational scientist may be pleased with a particular solution because it is reliable, self-explanatory, and easy to run on different computers without modification. A computer scientist may view this same solution as lengthy, inelegant, and old-fashioned. Each may be right in the sense that they are making judgments based on the differing values of different disciplines.

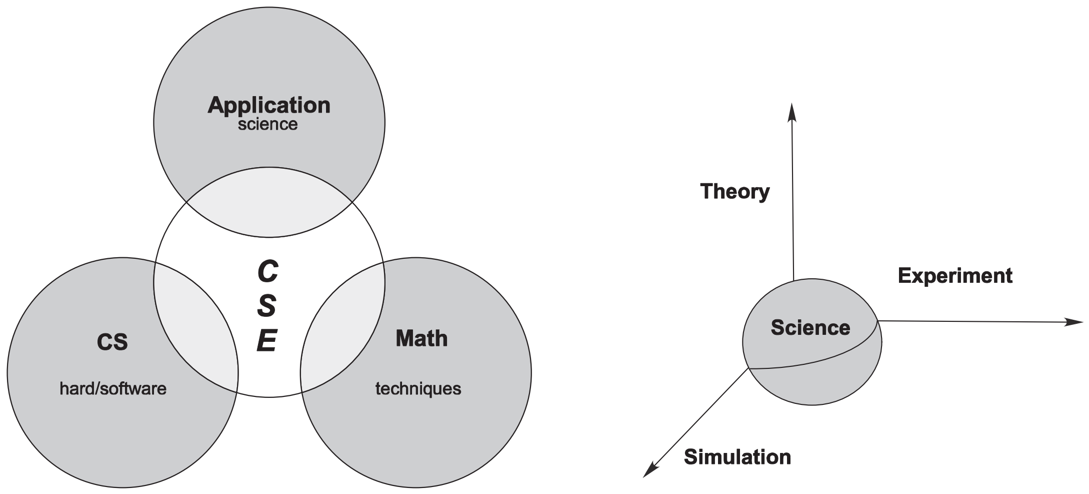


Figure 1.1 *Left:* Computational science is a multidisciplinary field that combines science with computer science and mathematics. *Right:* A new paradigm for science in which simulation plays as essential a role as does experiment and theory.

Another, possibly more fundamental, view of how computation is playing an increasingly important role in science is illustrated on the right of illustrated in Figure 1.1. This symbolizes a paradigm shift in which science's traditional foundation in theory and experiment is extended to include computer simulation. While one may argue that the future will see the CSE on the left of Figure 1.1 get absorbed into the individual disciplines, we think all would agree that the simulation on the right of Figure 1.1 will play an increasing role in science.

1.2 TALKING TO COMPUTERS

As anthropomorphic as your view of computers may be, it is good to keep in mind that a computer always does exactly as told. This means that you should not take the computer's response personally, and also that you must tell the computer exactly what you want it to do. Of course programs can be so complicated that you may not care to figure out what they will do in detail, but it is always possible in principle. Thus it follows that a basic goal of this book is to provide you with enough understanding so that you feel well enough in control, no matter how illusionary, to figure out what the computer is doing.

Before you tell the computer to obey your orders, you need to understand that life is not simple for computers. The instructions they understand are in a *basic machine language*[1] that tells the hardware to do things like move a number stored in one memory location to another location, or to do some simple, binary

[1]The "BASIC" (Beginner's All-purpose Symbolic Instruction Code) programming language should not be confused with basic machine language.

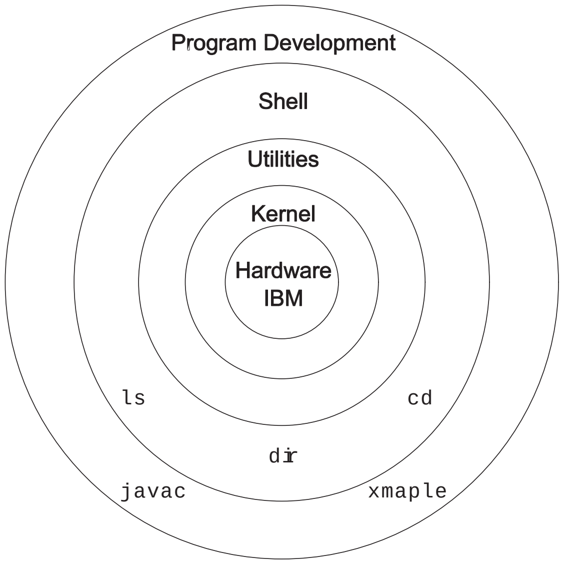


Figure 1.2 A schematic view of a computer's kernel and shells.

arithmetic. Hardly any computational scientist really talks to a computer in a language it can understand. Instead, when writing and running programs, we usually talk to the computer through a *shell* or in a *high-level language*. Eventually these commands or programs all get translated to the basic machine language.

A *shell* is a name for a command-line interpreter, that is, a place where you enter a command for the computer to obey. It is a set of medium-level commands or small programs, run by the computer. As illustrated in Figure 1.2, it is helpful to think of these shells as the outer layers of the computer's *operating system*. While every general-purpose computer has some type of shell, usually each computer has its own set of commands that constitute its shell. It is the job of the shell to run various programs, compilers, and utilities, as well as the programs of the users. There can be different types of shells on a single computer, or multiple copies of the same shell running at the same time for different users. The nucleus of the operating system is called, appropriately, the *kernel*. The user seldom interacts directly with the kernel, but the kernel interacts directly with the hardware.

The *operating system* is a group of instructions used by the computer to communicate with users and devices, to store and read data, and to execute programs. The operating system itself is a group of programs that tells the computer what to do in an elementary way. It views you, other devices, and programs as input data for it to process; in many ways it is the indispensable office manager. While all this may seem unnecessarily complicated, its purpose is to make life easier for you by letting the computer do much of the nitty-gritty work that enables you to think higher-level thoughts and communicate with the computer in something closer to your normal, everyday language. Operating systems have names such as *Unix*, *OSX*, *DOS*, and *Windows*.

In Part 1 of this book we will use a high-level *interpreted* language, either Maple or Mathematica. In Part 2 we will use a high-level *compiled* language, either Java or Fortran90. In an interpreted language the computer translates one statement at a time into basic machine instructions. In a compiled language the computer translates an entire program unit all at once. Compiled languages usually lead to faster programs, permit the use of vast libraries of subprograms, and tend to be portable. Interpreted languages appear to be more responsive, interactive, and, consequently, more “user friendly.”

When you submit a program to your computer in a compiled language, the computer uses a *compiler* to process it. The compiler is another program that treats your program as a foreign language and uses a built-in dictionary and set of rules to translate it into basic machine language. As you can imagine, the final set of instructions is quite detailed and long, especially after the compiler has made several passes through your program to translate your convoluted logic into fast code. The translated statements ultimately form an *executeable* code that runs when loaded into the computer’s memory.

1.3 INSTRUCTIONAL GUIDE

Landau’s Rules of Education: Much of the educational philosophy applied in this book is summarized by these three rules:

1. Most of education is learning what the words mean; the concepts are usually quite simple once you understand what you are being told.
2. Confusion is the first step to understanding.
3. Traumatic experiences tend to be the most educational ones.

This book has an attitude, and we hope you will develop one too! We enjoy computing and relish the increased creativity and productivity resulting from powerful computing tools. We believe that computing has so become part of the fabric of science that an introductory scientific computing course should be part of every lower-division university student’s education. Hence we deliberately mix the languages of mathematics, science, and computer science. This mix of languages is how modern scientists think about things, and since ideas must be communicated with these same words and ideas, this is how the book is written. However, we are sensitive to the confusion multiple definitions may reap and do provide a section at the end of each chapter indicating some key words and concepts, and a glossary at the end of the book defining many technical terms and jargon.

Because we aim to give an introduction to computational science, we cover some basics of numerical analysis, information about how data are stored, and the concordant limits of computation. However, we present very little discussion of hardware, computer architecture, and operating systems. That is not to say that these are not interesting and important topics, but rather that we want to get the reader busy computing and acquiring familiarity with concrete examples. In our experience, science and engineering students need this practical experience before they can appreciate the more abstract principles of computer science.

We have aimed our presentation at first- and second-year college students. We believe that the chapters and sections not marked optional with an asterisk * provide a good introductory course for them. The inclusion of the optional chapters would raise of the level of the presentation and lead to a longer course. To maintain the logical organization of materials, we have intermixed the optional chapters with the others. However, there should be nothing in the optional chapters that is required in order to understand the chapters that follow.

It is hard to keep up with the rapid changes in computer technology and with the knowledge of how to use them. That is not such a big issue with scientific computing, because it is the basic principles of mathematics, logic, science, and computing that are important, and not the details of hardware and software. Nevertheless, students and faculty often do not enjoy computing because of the frustration and helplessness they feel when computers do not work the way they should. Although we commiserate with those feelings, one of the things we try to teach is that it is not unusual to have new things not work quite right, but that if you relax and follow a trial-and-error approach, then you usually find success working around them.

Be warned, there are some exercises in this book that give the wrong answer, that lead to error messages, or that may “break” the program (but not harm the computer). Learning to cope with the limits of computation gets easier and less traumatic after you have done it a few times. We understand, however, that many instructors may not appreciate a computer’s failure that they cannot explain, and so we have assembled an *Instructor’s Survival Guide*, available upon request through the Web.

It is important that students become familiar with the material in the text before they come to lab. A lecture helps, but reading and working through the materials is essential. Even though it may not be that hard to work through the problems in lab by having instructors and fellow students prompt you with the appropriate commands to enter, there is little to be gained from that. Our goal is to make this introductory experience very much a “lab” that develops an attitude of experimentation and discovery and that nurtures rewarding feelings when a project finally computes correctly.

Many of the problems we assign require numerical solution and therefore are not covered in elementary texts. These include nonlinear oscillators, the motion of projectiles with drag, and the rotation of cubes about an arbitrary axis. Notwithstanding our prediction that other elementary courses and texts will eventually be more multidisciplinary, some readers may feel that we are requiring them to understand phenomena at a level higher than in their other courses. Our hope is that the readers will recognize that they are pioneers, will be stimulated by their new-found powers, and will help modernize their other courses.

On the technical side, we have developed the materials with Maple 6–9.5, Mathematica 4.2, and the Java Development Kit (JDK or Java 2 Standard Edition, J2SE) [SunJ]. Even though studio and workbench environments are available, we prefer the pedagogical value in using a shell to issue separate commands for compilation and execution, and then having to deal with the source and class files directly. In addition, many students are able to load JDK onto their home computers and work there as well.

We have found that *WinEdt* [WinEdt] and *TextPad* work well to edit and run source code on a Windows platform, and that *Xemacs* [Gnu] with Java tools is excellent for Unix/Linux machines. In addition, *jEdit*, the Open Source programmer's editor [jEdit], is an excellent tool that, because it itself is written in Java, runs on most any platform. Visualization is very important in computational science. It is built into Maple and Mathematica and is excellent. Part of the power of Java is that it has strong graphical capabilities built right into the language, although calling them up is somewhat involved. Consequently, we have adopted the free, open source package *PtPlot* [PtPlot] as our standard approach to plotting with Java. However, *gnuplot* [Gnuplot] and *Grace* [Grace] are also recommended as stand-alone applications. For 3-D graphics, a more specialized application, we give instructions on using *gnuplot* and refer the interested reader to *OpenDx* [DX] and *VisAd* [Visad]. These too are free, powerful, and available for Unix/linux and Windows computers.

1.4 EXERCISES TO COME BACK TO

Consider the following list of problems to which a computational approach could be applied. Indicate with an "M," "J," or "E" whether the best approach would be the use of Maple, Java, or either.

1. calculate the escape velocity from Jupiter
2. write a spreadsheet (accounting) program from scratch
3. solve problems from a calculus textbook
4. prove an algebraic identity
5. determine the time required for a sky diver to reach terminal velocity
6. write a compiler for a programming language

Chapter Four

Visualizing Data, Abstract Data Types; Electric Fields of Multipoles

4.1 WHY VISUALIZATION?

One of the most rewarding uses of computers is visualizing the results of calculations. This is done with 2-D and 3-D plots (especially with colored surfaces), with contour maps, and with animations. These types of visualization are sometimes breathtakingly beautiful and often provide deep insight into a problem by letting you see and "handle" the functions with which you are working. Visualization also assists the program debugging process, the development of physical and mathematical intuition, and the all-around enjoyment of your work. Some of the reasons for this may arise from the fact that some large fraction ($\approx$ 50%) of our brain gets involved in visual processing, and if you are able to use this extra brainpower in your scientific work, then you have extended what was otherwise possible with solely logical abilities.

Traditionally, visualization of a scientific problem was the last step in problem solving. After studying tables of numbers for hours and gaining confidence that they are right, a scientist might then go to the trouble of making a bunch of 2-D plots to examine various aspects of the data. Well, in present times computational scientists have demonstrated how much there is to be gained by going beyond 2-D plots. Now it is regular practice to use surface plots, volume rendering (dicing and slicing), and animations (movies). In this chapter we use some of these techniques within the context of visualizing the electric field around charges.

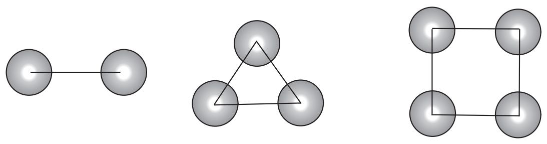


Figure 4.1 Static configurations for two, three, and four electric charges.

4.2 PROBLEM: STABLE POINTS IN ELECTRIC FIELDS

You are given the simple configurations of charges shown in Figure 4.1. The two charges are fixed on a line at coordinates (1,0), (-1,0); the three charges are fixed to the corners of an equilateral triangle at coordinates (0,1), $\sqrt{3}$ (1/2, -1/2), $-\sqrt{3}$ (1/2, -1/2); and the four charges are fixed to the corners of a square at coordinates (1,1), (1,-1), (-1,-1), (-1,1). The origin is at the center of each geometric figure. Your problem is to determine the electric potential at the point (x, y) and see if there might be some points in space at which we a free charge at rest will remain even if perturbed. For the equivalent gravitational problem, these stable points are known as Lagrange points and are the location of asteroids for the Earth-sun system.

4.3 THEORY: STABILITY CRITERIA AND POTENTIAL ENERGY

Coulomb's law tells us that if we have a charge q at the origin, then the electric field **E** (the force per unit charge) at a distance r from that charge is

$$\mathbf{E}(\mathbf{r}) = \frac{k_e\, q}{r^2}\hat{\mathbf{r}}. \tag{4.1}$$

where $\hat{\mathbf{r}}$ is a unit vector in the radial $(\mathbf{r})$ direction.[1] Here $k_e = 8.9875\, 10^9 Nm^2/C^2$ is Coulomb's constant in SI units, and the electric field **E** is directed radially away from the charge. Because **E** is a vector, the electric force field about a charge is a vector field with both magnitude and direction at each point. However, no information is lost, and it is much simpler if, instead of the electric force field **E**, we consider the electric potential field

$$V(r) = \frac{k_e\, q}{r} \quad \equiv \quad \frac{q}{r}. \tag{4.2}$$

In the second form of this equation, we have left off the electric constant k_e for simplicity; since this affects just the magnitudes of the graphs and not their shapes, it will not change the conclusions we draw. We see that $V(r)$ falls off less rapidly than **E** and is a scalar, namely, has no direction associated with it.

Our problem requires us to determine the potentials the for two- and three-charge systems shown in Figure 4.2 and then to look for stable points in these potentials. To determine the potential for two charges, we use Pythagoras's theorem to determine the distance to the charges,

$$r_1 = \sqrt{(x-a)^2 + y^2}, \qquad\qquad r_2 = \sqrt{(x+a)^2 + y^2}, \tag{4.3}$$

[1]A *vector* is a mathematical object with both magnitude and direction[Ser 00]. They are discussed further in Chapter 7.

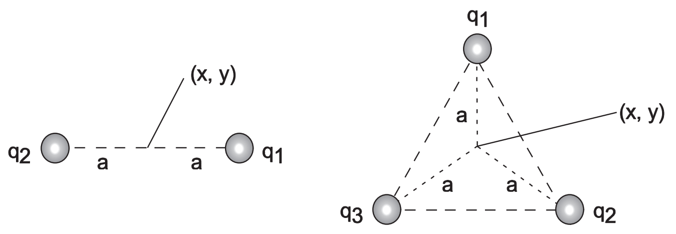


Figure 4.2 Coordinate systems for two- and three-charge configurations.

and then just add up the potentials from the individual charges:

$$V_2(x,\,y) = \frac{q_1}{\sqrt{(x-a)^2+y^2}} + \frac{q_2}{\sqrt{(x+a)^2+y^2}}. \tag{4.4}$$

For three charges at the corners of the equilateral triangle, we know the coordinates are $(0,a)$, $(a\cos\theta, -a\sin\theta)$, $(-a\cos\theta, -a\sin\theta)$, where $\theta = 30^o$. Again we use Pythagoras's theorem and add the potentials from the individual charges to obtain

$$\begin{aligned} V_3(x,\,y) =& \frac{q_1}{\sqrt{x^2+(y-a)^2}} + \frac{q_2}{\sqrt{(x-a\cos\theta)^2+(y+a\sin\theta)^2}} \\ &+ \frac{q_3}{\sqrt{(x+a\cos\theta)^2+(y+a\sin\theta)^2}}. \end{aligned} \tag{4.5}$$

These equations for the electric potentials are what we wish to visualize. To make them simpler to visualize, we set $a = 1$ and substitute for θ:

$$V_1(x,\,y) = \frac{q_1}{\sqrt{x^2+y^2}}, \tag{4.6}$$

$$V_2(x,\,y) = \frac{q_1}{\sqrt{(x-1)^2+y^2}} + \frac{q_2}{\sqrt{(x+1)^2+y^2}}, \tag{4.7}$$

$$\begin{aligned} V_3(x,\,y) =& \frac{q_1}{\sqrt{x^2+(y-1)^2}} + \frac{q_2}{\sqrt{(x-\frac{\sqrt{3}}{2})^2+(y+\frac{1}{2})^2}} \\ &+ \frac{q_3}{\sqrt{(x+\frac{\sqrt{3}}{2})^2+(y+\frac{1}{2})^2}}. \end{aligned} \tag{4.8}$$

Owing to its two-dimensional nature, a purely mathematical solution for the equilibrium points in these potentials gets complicated. Instead, we will solve it graphically and rely on our intuitive understanding of how balls roll under the action of gravity. Specifically, we know that a ball released on a surface rolls downhill, and that if the ball is placed in a concave depression, it will remain there. Because the gravitational potential near the Earth's surface is proportional to height, our description of the ball on a surface is equivalent to a description of how a particle behaves in a potential energy field. It therefore follows that charges will "roll down" the electric potential surface and will find a stable position at the concave minimum of the potential. So our problem translates into drawing pictures of the electric potential surfaces and looking for minima at the bottom of hills.

4.4 BASIC 2-D PLOTS: PLOT

Before we get to Maple's plotting commands, let us examine some general principles. First, keep in mind that the point of visualization is to make the science clearer and to better communicate your work to others. So it follows that when you produce a figure, you should look at it and think if there are some better choices of units, ranges of axes, colors, style, et cetera, that might get the message across better and provide better insight. Taking into account that we are dealing with the complexity of human perception and cognition, there may not be one definite way to do things, and some trial and error is necessary to see what looks best.

Our general recommendation for visualization is to make each figure as clear, informative, and self-explanatory as possible. This means labels for various curves and data points, a title, and labels on the axes. We know, you are thinking this is really a lot of work for a lousy assignment or report, and that you do not need all those time-consuming extras to comprehend what is going on. Yet the more often you do it, the quicker and better you get at it, and the more useful will your work be to others (and yourself in the future).

The convention when plotting is to have the independent variable, say x, along the abscissa (horizontal axis) and the dependent variable, say $y = f(x)$, along the ordinate. (Remember that your mouth spreads horizontally across when you say "abscissa" and that it puckers up vertically when you say "ordinate.") If you have trouble deciding which variable is independent, think of an experiment in which you measure the position or velocity of a ball as a function of time. Because you are free to pick the times at which you make the measurement, time is an independent variable. However, once you have chosen the time, nature picks what the position of the ball is at that particular time, so position and velocity are dependent variables.

4.4.1 Loading the plots Package

Maple excels at easily producing graphs of all sorts, and indeed, visualization is one of the most valuable aspects of Maple. Although we will discuss and give examples of a number of possible plots, Maple affords more options than we discuss, and we recommend you look at the commands listed after the `with(plots)` statement and browse the help pages to create just the graph you want. We will first make a simple plot and then embellish it with things like labels and colors.

```
> restart;      with(plots);                               # Loads plotting tools
```

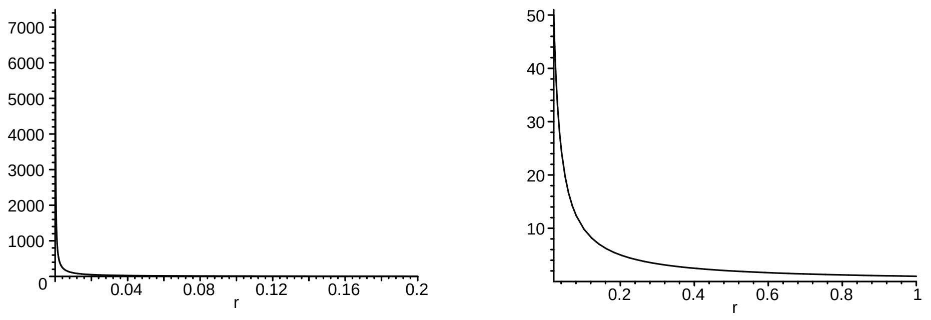
[*animate*, *animate3d*, *animatecurve*, *arrow*, *changecoords*, *complexplot*, *complexplot3d*, *conformal*, *conformal3d*, *contourplot*, *contourplot3d*, *coordplot*, *coordplot3d*, *cylinderplot*, *densityplot*, *display*, *display3d*, *fieldplot*, *fieldplot3d*, *gradplot*, *gradplot3d*, *graphplot3d*, *implicitplot*, *implicitplot3d*, *inequal*, *interactive*, *listcontplot*, *listcontplot3d*, *listdensityplot*, *listplot*, *listplot3d*, *loglogplot*, *logplot*, *matrixplot*, *odeplot*, *pareto*, *plotcompare*, *pointplot*, *pointplot3d* , *polarplot*, *polygonplot*, *polygonplot3d*, *polyhedra_supported*, *polyhedraplot*, *replot* , *rootlocus*, *semilogplot*, *setoptions*, *setoptions3d*, *spacecurve*, *sparsematrixplot*, *sphereplot*, *surfdata*, *textplot*, *textplot3d*, *tubeplot*]

We see that in response to the `with(plots)` command, Maple displays all of the plotting commands that are available with this package. (We sometimes use `with(plots):` with a colon rather than a semicolon to avoid the listing.) Now that we have the tools, let us look at the electric potential for a single charge:

```
> V := (r) -> 1/r;
```

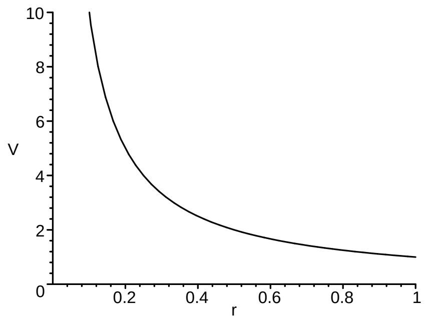
$$V := r \to \frac{1}{r}$$

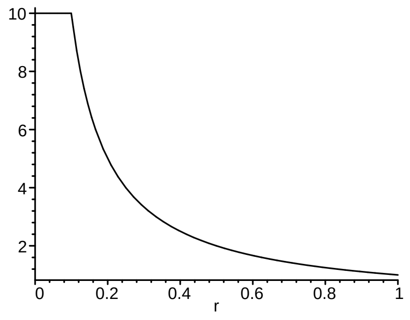
```
> plot( V(r), r = 0 ..  0.2 );                    # Plot function, range
> plot( V(r), r = 1/50 ..  1 );                   # Remove infinity
```



You will observe from the first figure that the second argument to the `plot` command gives the range of values for the abscissa (r in this case). Our interest is really for r between 0 and infinity, but this does not produce such a useful result, since we primarily see the repulsive peak at the origin. In view of that, we get a more revealing plot by not letting r get quite so close to the origin. The second plot eliminates the part of the graph with the infinity at $r = 0$, and so does not fully convey the image that the potential is infinite there. However, we tailor our plot more to our liking by giving some limits to the ordinate (also works if called the generic y):

```
> plot( V(r), r = 0 ..  1, V = 0 ..  10 );        # Limit the y range
> plot( min(V(r), 10), r = 0 ..  1 );   # Keep ordinate less than 10, another way
```





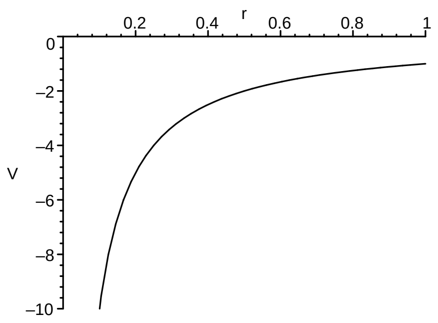
As an alternative, it is possible to tell Maple that you want to see the full dependence of the potential from $r = 0$ to ∞, `plot( V(r), r = 0..infinity )`, but then you lose some details. Try leaving off the range for the abscissa to test Maple's capabilities:

```
> plot( V(r) );
 Plotting error, empty plot
```

You see that because Maple was not given a range of r values to plot, it does not think it has anything to plot (`empty plot`).

The plot above shows the basic physics. If we view $V(r)$ as an equivalent gravitational potential, a small positive charge (mass) placed near the fixed positive charge will be repelled (roll downhill) out to infinity. There are no locations where a charge remains at rest in equilibrium. If we had fixed a negative charge at the origin, the potential would have the opposite sign:

```
> plot( -V(r), r = 0 ..  1, V = 0 ..  -10 );
```



This shows that, regardless of where we place it, our positive test charge will fall into the hole at the origin. As we have just seen by placing a minus sign in front of the first argument to the `plot` command, it is allowable to have the argument be an expression and not just a function:

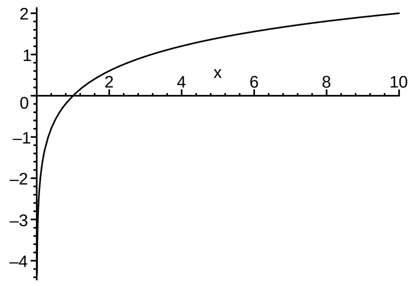
```
> plot( -5/r, r = 0 .. 5,V = 0 .. -20 );          # Plot explicit expression
```

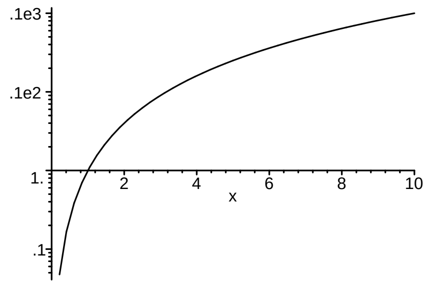
In summary, the first argument to the `plot` command is the name of the function or expression to be plotted along the ordinate, namely, the dependent variable. The second argument is the range of values for the abscissa, namely, the independent variable. The double period `..` is used to specify the range, so `-10 .. 10` means from -10 to $+10$. If the upper end of the range is a decimal value, say `.5`, then it is clearer to enter it with a leading zero as `0.5`, so that the range looks like `-10 .. 0.5`, and not the confusing (to the reader and to Maple) `-10...5`.

Before we get on to embellishing the `plot` command, let us have some fun with the pretty graph you just produced:

- Click on the graph to select it. Inspect how a box is formed around it and that there are dark little nodes at the corners and in the middle of the sides.
- Use your mouse to resize the graph by grabbing one of the nodes and dragging it with the mouse button still depressed. Monitor how when a node is selected, a little arrow appears to show you the direction in which the frame can be resized. Resizing is possible diagonally along the corners or horizontally and vertically along the edges.
- Select the graph, copy it (it is placed on the *clipboard*), and then paste it back to the worksheet so that you now have two graphs.
- Make one of your graphs wide and short and the other one tall and thin. Check how the tall one emphasizes the variation in the magnitude of $V(r)$, while the tall one emphasizes the range of r values to which $V(r)$ extends. Both are perfectly legitimate ways to view a function, with one emphasizing the singular nature near the origin and the other the long range.
- Another way to view a function, especially one that has orders of magnitude variation in value, is with a *semilog* or *log-log* plot (although you need to avoid $\log(0)$). In the execution group below, use the `log10( )` function to see how a semilog plot changes the appearance of the same $f(x)$ we have been viewing.
- Next try the explicit semilog plot function `logplot( )`, following the instruction in the comments fields below:

```
> plot( log10(x^2), x = 0 .. 10 );          # Repeat plot with log(V(r))
> logplot( x^2, x = 0 .. 10 );              # Explicit semilog plot; log(1st argument)
```



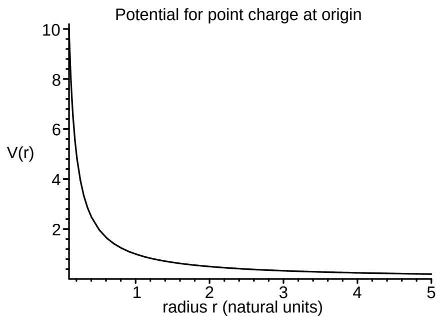


```
>                                                    # Try log-log plot here
```

4.4.2 Labels and Titles (the Plot Thickens)

Any plot worth looking at is worth explaining. This is done by placing labels along the axes and by placing a title above the curves:

```
> plot( 1/r, r = 1/10 ..  5, labels=[‘radius r (natural units)‘,‘V(r)‘],
 title = ‘Potential for a positive point charge at the origin‘ );
```



Take stock of how we just added a comma after the range and then added the options, separated by commas, to produce the labels and title. Because there are two axes, the labels field has an entry for the x axis and then the y axis. The labels and title are enclosed in back quotes in order to delimit the expressions:

```
>                                # Enter previous plot using ordinary accents
```

Modify the previous command so that the words “abscissa” and “ordinate” appear in the appropriate places and so that the actual expression being plotted appears in the title:

```
>                                          # Plot with your modified labels and title
```

You may have noticed that we have placed r along the “x-axis” and $V(r)$ along the “y” axis. In fact, there may be cases in which y is the independent variable. Thus you see why it may be better to use the words “ordinate” and “abscissa” than “x” and “y” axes.

4.5 COMPOUND (ABSTRACT) DATA TYPES: [LISTS] AND {SETS}

As we proceed with our exercises in visualization, you will see how to enter arguments to the `plot` commands using different types of parentheses. We recognize that some users may prefer just following the rules without questioning them. Nevertheless, the commands will make more sense, and will be easier to generalize, if you have some understanding of the method behind the madness. And so we now take a little excursion in which we define some terms that are frequently used in mathematics and computer science and employed by Maple commands.

We have already seen a number of ways in which Maple displays data. At times there is just a single symbol, sometimes there is a bunch of symbols separated by commas, sometimes there is a bunch of things in parentheses, and sometimes the symbols are in quotes. To illustrate, in Chapter 5 you will see that when you solve an equation that has several solutions, Maple separates the solutions with commas:

```
> solve( x^4 - 1 = 0, x );
```

$$-1,\ 1,\ I,\ -I$$

```
> solve( x^4 - 1 = 0, x )[1];                          # Solve for only 1 root
```

$$-1$$

Take note of two types of parentheses here and the different forms given for the solutions.

Abstract, compound data types: An *object* in computer science denotes a data type with multiple parts. It may also be called an *abstract* data type, or a *compound* data type. Here the word “abstract” means that there is more to something than meets the eye; namely, the data type may contain multiple parts. Many of the individual symbols or variables used in Maple can be replaced by *objects*. We will discuss objects in more depth when we study Java, which is known as an *object oriented* language.

Data types: In specifying labels for the `plot` command, we placed the x and y labels in square parentheses `[..]`. These are called “brackets.” Maple also uses standard parentheses `(..)` and curly parentheses `{..}` called “braces.” Brackets and braces are used to construct abstract data types or objects from the more elementary data types we have already seen.

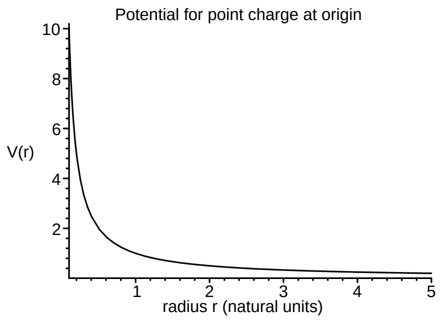
Sequence: A collection of variables (objects or data types) separated by commas is called a *sequence*. As a case in point, the arguments given to the `solve` command above, and indeed to most Maple commands, are variables separated by commas, and, hence, form sequences. While we are speaking of sequences, we may as well indicate that when we give arguments to a Maple command as a comma-separated list, the parenthesis indicate sequence. It is often convenient to let Maple form a sequence for you with the `seq` command:

```
> seq( 2*n-1, n = 1 ..  4 );
```

$$1,\ 3,\ 5,\ 7$$

List: A list is a sequence of numbers, or abstract data types, separated by commas and placed within square brackets.*The order matters for a list*, while it does not for a *set* (to be defined soon). We use a list when we issue the `plot` command:

```
> plot( 1/r, r = 1/10 ..  5, labels=['radius r (natural units)','V(r)'],
  title = 'Potential for point charge at origin' );
```



Now we try creating the plot again, this time changing the order of the list:

```
>                                              # Plot with reordered list
```

Clearly order matters here, as we would not want the labels for the abscissa and ordinates interchanged! You may enter the elements of a list by hand, as in

```
> list1 := [2,4,6,8];
```

$$list1 := [2,\ 4,\ 6,\ 8]$$

Or use Maples `seq` command to help generate the elements of the list:

```
> list2 := [seq(2*n-1, n = 1 ..  4)];
```

$$list2 := [1,\ 3,\ 5,\ 7]$$

The individual elements of a list are referenced via Maple's square bracket notation (a standard way of indicating subscripts):

```
> list2[1]; list1[1]; list2[2]; list1[2];
```

$$1, \quad 2, \quad 3, \quad 4$$

Set: A set is a well-defined collection of related objects or elements with no repeated element [Fral 76]. In contrast to a list, the order of the elements in a set does *not* matter. Sets are also used as arguments to Maple commands, but only in cases where the order of the elements does matter, for example, a set of equations to be solved. Sets are usually described by enumerating their elements, separated by commas, as a sequence within braces. To prove the point, the sets of equations and solutions used when solving simultaneous equations:

```
> solve({a + 3*b + 4*c = 41, x5*a + 6*b + 7*c = 20}, {a,b});   # Any order for set
> solve({x5*a + 6*b + 7*c = 20, a + 3*b + 4*c = 41}, {b,a});   # Any order for set
```

$$\{a = \frac{c-62}{-2+x5}, \qquad b = -\frac{-7\,c+4\,x5\,c-41\,x5+20}{3\,(-2+x5)}\}$$

Maple uses braces to denote sets:

```
> NiceSet := {0, 2, 4, 6};
```

$$NiceSet := \{0, 2, 4, 6\}$$

Seeing that lists and sets both contain comma-separated sequences within them, we emphasize that it is legal for the same element to occur more than once in a list, with the order of the elements in the list part of its definition. As an example, if we define a set with repeated elements in arbitrary order, then Maple will remove the repeats and reorder for us:

```
> MessySet := {6, 4, 0, 4, 2, 0};
```

$$MessySet := \{0, 2, 4, 6\}$$

In contrast, Maple does not change the order or elements of a list:

```
> MessyList := [6, 4, 0, 4, 2, 0];
```

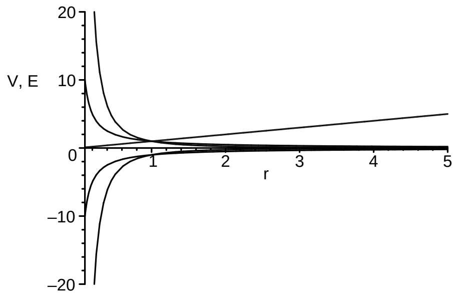
$$MessyList := [6, 4, 0, 4, 2, 0]$$

Arrays: Another data type related to vectors and matrices are arrays. We discuss them in Chapter 7.

4.5.1 Several Curves on One Plot, Sets

We have seen that the first argument to the `plot` command is the function to be plotted. As a case in point, imagine that as part of our charge problem, we want to compare, in a single plot, the r dependence of the potential and the magnitude of the electric field due to both positive and negative charges. Seeing that Maple treats the argument as an object, we substitute the set $\{\frac{1}{r}, \frac{-1}{r}, \frac{1}{r^2}, \frac{-1}{r^2}\}$ as the first argument to the `plot` command. The fact that we use a set as the object to plot rather than a list $[\frac{1}{r}, \frac{-1}{r}, \frac{1}{r^2}, \frac{-1}{r^2}]$ makes sense since order does not matter and there is no point in plotting identical functions on top of each other:

```
> plot({1/r, -1/r, 1/r^2, -1/r^2}, r = 1/10..5, y = -20..20, labels=['r','V, E']);
```



If you are reading the electronic version of this book, you will notice that Maple has chosen a different color for each of the functions. Experiment now with using a list as the first argument and noting how the colors assigned to the curves differ:

```
>                                          # Use a list [...] for function argument
```

4.5.2 Using the Figure Toolbar

The colors and line formats that Maple picks for graphs may look great on your screen but may not print out or project well (green and yellow are often barely visible). In the next subsection we discuss how to customize the colors to your preference. In this subsection we will explore some of the options using the figure toolbar.

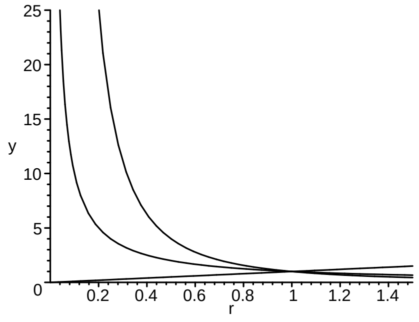
- Select the graph with your mouse (you should notice a box appearing about the graph after it is selected).
- While selected, observe that the second line of the toolbar at the top of your screen now contains icons for graphical options. Explore what each of these icons does. This is both useful and fun.
- Go to the Style pull-down menu and, under Line Width, select Broad. Observe especially the difference it makes for the yellow and green curves. Go to the Legend menu and enable Show Legend.
- Again go to the Legend pull-down, and select Edit Legend. Change the legends so they are $V(r)$, $-V(r)$, $E(r)$, $-E(r)$, and r for the appropriate curves.

- Explore how the different buttons controlling the placement of the axes work.

4.5.3 Customizing Colors and Line Types

Maple automatically chooses different color multifunction plots. You control the color of your graphs by adding the `color` option to the end of the `plot` command:

```
> plot( [1/r, 1/r^2, r], r = 0 ..  1.5, y = 0..25, color=black );
```



Taking into account that options are objects with multiple parts permitted, you enter a list (order matters) of colors to specify the colors of each curve:

```
> plot( [1/r, 1/r^2, r], r = 0..1.5, y=0..25, color = [red, blue, maroon] );
```

Likewise, you choose different styles for each curve (like dashed and solid) to help tell the curves apart, even in basic black. You do that with the `linestyle` option, which may also be a list for each curve:

linestyle=[1 (solid)**, 2** (dotted), **3** (dashed), **4** (dot-dashed) **]**

```
>                                          # Plot black with linestyles as list
>                                  # Plot default colors with linestyles as list
```

An especially effective way to distinguish different curves on the same plot without the use of color, is to draw them with different *thicknesses*. Apply this change with the `thickness = n` option, where again, a list for the curves is a legal option. The possible values for `thickness` are: `n = 0, 1, 2`, and `3`, where `0` is the default thickness.

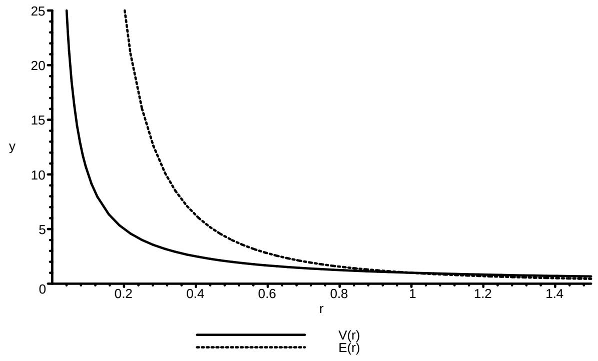
```
>                              # Replot with different thickness for each curve
```

4.5.4 Legends, Titles, and Labels

Legends explain to the reader just what is being plotted with each curve. They are invaluable and do wonders for your presentation. When presenting several curves in one plot, it is important that the viewer not only be able to tell them apart by the different color or line style used for each, but also be given information as to what the different curves represent. It is good practice to explain in the caption below a graph what each curve means, as well as in the text (or in your talk) when the graph gets referenced. However, it is also good practice to have a legend in the plot itself explaining what each curve means. Captions and your explanations may get removed, but it is a lot harder to remove a legend.

The legend option specifies a single string or a list of strings in the same order as the curves with a legend for each curve:

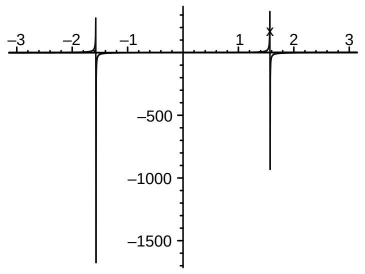
```
> plot( [1/r, 1/r^2], r = 0..1.5, y = 0..25, legend = [ ‘V(r)‘, ‘E(r)‘ ] );
```

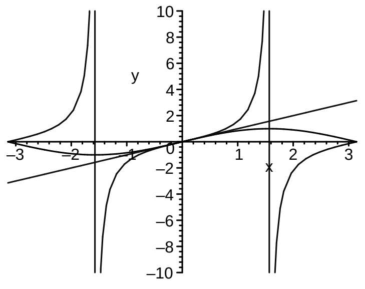


4.5.5 Other Options

As we have said, there are many ways to customize your graph, and Maple's Help pages are a good place to find out about them. Once in a while your graph may not show the features you want, because one function gets very large and Maple automatically adjusts the ordinate range to accommodate that. Here are a number of ways to limit the range of the ordinates:

```
> plot( [sin(x), tan(x),x], x = -Pi..Pi );                # tan(x) overshadows sin(x)
> plot( [sin(x), tan(x), x], x = -Pi..Pi, y = -10..10 );  # Now with y limits
```

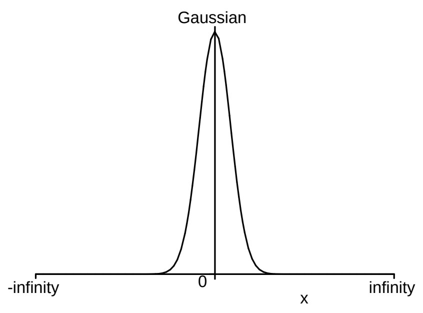


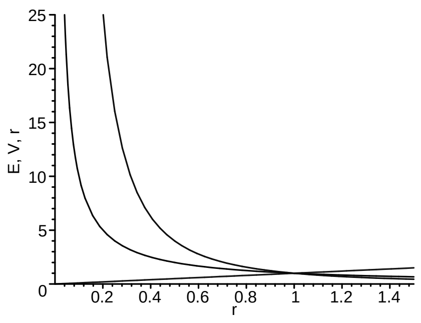


```
> plot( [sin(x), min(10,tan(x) ), x], x = -Pi..Pi );          # Limit +y values to 10
> plot( [sin(x), max(-10,tan(x)), x], x = -Pi..Pi );          # Limit -y values to -10
```

There are occasions when a function falls off slowly, and so you might want to see its behavior for values of its argument from $-\infty$ to $+\infty$. It is clear that Maple is not afraid of big numbers, yet it is nice to see that it makes this type of graph in a finite amount of space:

```
> plot( exp(-x^2), x = -infinity..infinity, title = ‘Gaussian‘ );
> plot( {1/r,1/r^2}, r = 0..infinity, title = ‘V and E of Point Charge‘ );
```





Look at the second plot and its labels along the axes, rather than horizontal. This was accomplished with the commands

```
> plot( [1/r, 1/r^2, r], r = 0..1.5, y = 0..25, labels = [‘r‘, ‘E, V, r‘] );
> plot( [1/r, 1/r^2, r], r = 0..1.5, y = 0..25, labels = [‘r‘, ‘E,V, r‘],
      labeldirections = [horizontal, vertical] );          # Rotate coordinate labels
```

4.6 3-D (SURFACE) PLOTS OF ANALYTIC FUNCTIONS

We have examined the potential field $V(r) = 1/r$ surrounding a single charge as a function of r. A 2-D plot is fine for this, since there is only one independent variable r. However, when the same potential is expressed as a function of the Cartesian coordinates x and y,

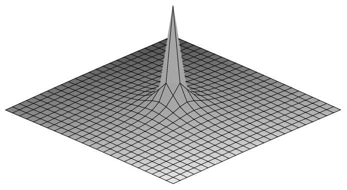
$$\mathrm{V}(x,\, y) = \frac{1}{\sqrt{x^2 + y^2}},$$

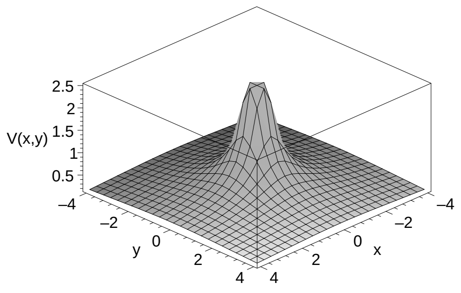
we have two independent variables, x and y, and so need a 3-D visualization. We get that by creating a world in which the z dimension (mountain height) is the value of the potential, and x and y lie on a flat plane below the mountain. Because the surface we are creating is a 3-D object, it is not possible to draw it on a flat screen, and so different techniques are used to give the impression of three dimensions to our brains. We do that by rotating the object, shading it, employing parallax, and so forth.

The command `plot3d` makes a 3-D plot and is just like our old friend `plot`:

```
> restart;    with(plots):                          # Loads tools for you to use
  V := (x,y) -> 1/sqrt( x^2 + y^2 );              # Define function of two variables
> plot3d( V(x,y), x = -4..4, y = -4..4 );                          # Basic form
```

$$V := (x,\, y) \to \frac{1}{\sqrt{x^2 + y^2}}$$





The first plot is a fairly interesting, but it primarily shows the singular nature of the potential near the origin. As seen in the second plot, we get a more useful visualization if we limit the maximum value of V to 2.5 and add labels:

```
> plot3d(min(2.5,V(x,y)), x=-4..4, y=-4..4, axes =BOXED, labels=['x','y','V(x,y)']);
```

We try to make this plot more intuitively informative by making the color red correspond to the highest values of the potential, blue smaller, and green cooler still. Color may be specified as the option `color = red` or with a number. Consequently, we try to be clever and use the actual value of the potential $V(x, y)$ as the color of the graph with `color = V(x,y)` option (yet we minimize the maximum value of the V in order to keep the singularity from confusing the color function).

Seeing as how the potential varies continuously, this means that the color will as well:

```
> plot3d( min(2.5, V(x,y)), x = -4..4, y = -4..4, color = min(2.5, V(x,y)) );
```

Even though there are many options possible as part of the `plot3d` command, it is both easier and more fun to first make a basic plot and then use Maple's graphical user interface (GUI) to modify the plot:

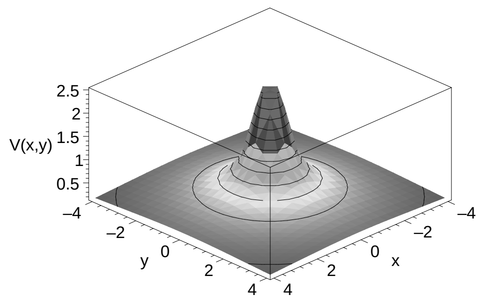
- Select this surface with your mouse (a box with filled little squares on the perimeter should appear).
- Grab the surface by depressing your left (or only) mouse button and holding it down. Now as you move your mouse, the surface will rotate in three dimensions. Make sure to move both right to left, and up and down, in order to get your brain to see the object as three-dimensional.
- With the surface still selected, notice the extra buttons that appear on the control panel. You should experiment and try to make sense of them. Remember that if you hold down a button, a message with the purpose of the button appears at the bottom of the screen. In particular, note:
  a. the different ways to draw axes,
  b. the different ways to render the surface, and
  c. how contour plots compare to the actual surface.

4.6.1 Contours and Equipotential Surfaces

To further help your mind understand that different colors mean different potential values, and that the surface is three-dimensional, we now include the `style = patchcontour` option. This adds contour lines that show different levels of the potential:

```
> V := (x,y) -> 1/sqrt( x^2 + y^2 );
> plot3d( min(2.5,V(x,y)), x = -4..4, y = -4..4, axes = BOXED,
  labels = [‘x‘, ‘y‘, ‘V(x,y)‘], color = min(2.5,V(x,y)), style = patchcontour );
```

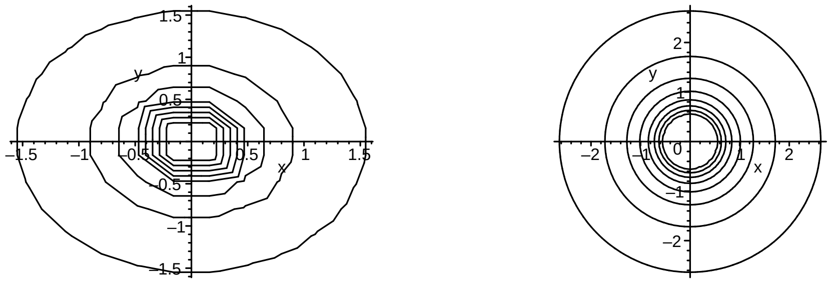
$$V := (x,\, y) \to \frac{1}{\sqrt{x^2 + y^2}}$$



In analogy to gravity, contours lines may be viewed as lines of equal elevation, which means that walking along a contour line does not change your elevation. For our electrical potential, the contours are called equipotential surfaces.

The contourplot command also supports the option of making only a 2-D contour plot of the surface (we prefer the 3-D contours to be shown soon). Because the contours are not being projected onto a curved surface or being viewed obliquely, these have the potential of being more precise:

```
> contourplot( V(x,y), x = -4..4, y = -4..4 );
```



However, we see in the left plot that the equipotential surfaces appear as ellipses, and not the circles to be expected for the symmetric case of a single charge. The reason is that the viewing screen tends to be broader than higher, and so plots are spread out that way. To get a plot with the same scales along the ordinate and abscissa, as done on the right, we add the scaling = constrained option:

```
> contourplot( V(x,y), x = -4..4, y = -4..4, scaling = constrained );
```

This produces a more symmetrical figure, but still not round. Apparently, the rapid rise of the potential near the origin is not being handled well by Maple, and so we exclude it by use of the min function: