MAPLE ANIMATION

MAPLE ANIMATION

JOHN F. PUTZ

CHAPMAN & HALL/CRC

A CRC Press Company
Boca Raton London New York Washington, D.C.

Library of Congress Cataloging-in-Publication Data

Putz, John F.
Maple animation / John F. Putz
p. cm.
Includes bibliographical references and index.
ISBN 1-58488-378-2 (alk. paper)
1. Maple (Computer file) 2. Algebra—Data processing. 3. Computer animation. I. Title.

QA155.7.E4P88 2003
512′.0285′6696—dc21 2002041776

International Standard Book Number 1-58488-378-2
Library of Congress Card Number 2002041776
Printed in the United States of America 1 2 3 4 5 6 7 8 9 0
Printed on acid-free paper

To my wife, Melinda,
and to my sons, Kevin and David

Preface

On the first day of class, my physics professor stood behind a device mounted on the lab table at the front of the room. He placed a large, shiny ball bearing at each end of a spring-loaded plunger, explaining that when he released the trigger, one of the bearings would be propelled horizontally to our right, while the other would drop straight toward the floor. The vertical component of the acceleration given to both objects would be the same, zero, so gravity alone would be responsible for their vertical positions, and equally so. He released the trigger. One bearing sailed in a broad half-parabola toward the classroom door; the other dropped straight downward. When they hit the old wooden floor, there was a solitary klunk.

Thirty-five years later, I still have the image that my professor gave me to associate with the concept he was teaching: the motion of an object can be understood by isolating, then combining, the influences on it in single directions. In mathematics, our objects are abstractions, not often lending themselves to such demonstrations. But we do frequently talk about motion, too: a point travels along a curve toward another, determining various secant lines as it moves; a horizontal plane descends through a surface, cutting it in varying level curves. What we do, though, is draw a static figure, describe the motion, and hope that our students can imagine its effects. It is not always the weaker ones who cannot.

The capability of computer algebra systems to produce animations provides a means for teachers of mathematics to produce demonstrations of our own. Using animation, we can show our students the motion we see in our mind's eye. We can give our students a vivid moving image to tie to a concept.

Even when a particular idea does not inherently involve motion, a moving picture can give students something concrete to associate with it. In linear algebra, for example, when we teach linear transformations in $\mathbb{R}^2$, we usually show a square at the origin and, next to it, its image, as shown in Figure 0.1. But if our students are really to understand the geometry of linear transformations, then they should understand this in the richer context of $\mathbb{R}^3$. Although we could just show a three-dimensional object, then have it snap to its image under the transformation, this would not demonstrate effectively *how* it becomes transformed. A better way is to cause the object to "morph" gradually into its image. This way, students can see the space transforming, not just the space transformed.

In this book, I will show the structures that I use to create animations. Each example will be a demonstration that can be used directly in the classroom.

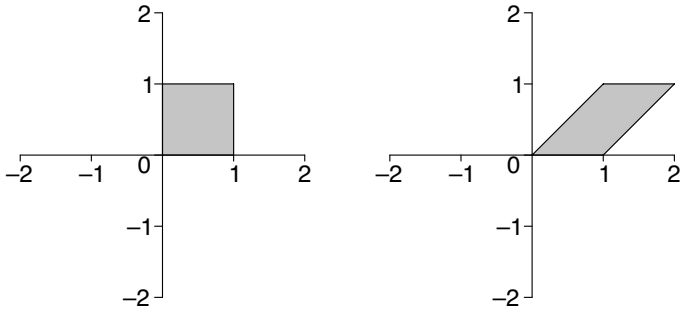


FIGURE 0.1: A square and its image under a linear transformation

Most of the examples are useful for calculus, especially multivariable calculus, because it is in that subject that I have found animation to be the most illuminating.

I know, from experience, that animated demonstrations do work. For me, one of the most difficult ideas to convey is the geometry of the directional derivative and gradient vector. I decided to try to write an animation showing a surface and gradient vector together with all of the following objects rotating about a point: unit vector, vertical plane determined by the vector and point, trace of the surface in the plane, and tangent to the trace. I could not quite make the animation work properly before it was time to teach the topic, so I taught directional derivatives in the old way, using chalkboard and prepared transparency. Later that day, I found a way to make the animation work, and I used it as a classroom demonstration on the following day. After class, two people thanked me for writing it. "We didn't understand before," they said, "but we do now."

I cannot imagine, now, teaching without using this wonderful technology, so significantly has it enhanced my teaching. Particularly for three-dimensional ideas, I really don't know what I ever did without it.

One other thing. I believe animations enhance the students' enjoyment of a course. Using the computer instead of the chalkboard or overhead projector helps to add variety. Moreover, the moving images themselves are simply appealing and, well, fun. I hope you will agree.

Acknowledgement

I thank my son, David, for his careful proofreading of the manuscript and for his thoughtful and valuable suggestions.

John F. Putz

How to use this book

As you read, work with a computer at hand. You should have Maple® 7 or Maple 8.[1] All of the examples and demonstrations will work in either version, but Maple 8 has enhanced features related to animation, and examples of those are included in this book, too. If you are new to Maple, or if you consider yourself a novice, begin by typing the commands in the examples into an open worksheet and watching Maple do its work. Sometimes you will not get the syntax right, and this is useful because it will help you to learn correct syntax, and because you will learn from watching what Maple does when you get it wrong. (If you type flawlessly, make some mistakes on purpose.) When my students are new to Maple, they often want to enter "sinx" instead of "sin(x)." Maple dutifully treats *sinx* as a variable name just as x, y, or t. Making this mistake helps them to realize that, since the sine is a function, it needs an argument, as any other function does. If you enter the code in the examples and experiment—sometimes a little, sometimes a lot—you will soon become confidently familiar with Maple and its syntax checker.

When you feel that you have arrived at that point, or if you are already there, the accompanying CD is for you. It has all the Maple examples and demonstrations in this book. The Maple code on the CD is organized in sections with the same titles as those in the book. These sections may be expanded or collapsed. To open (expand) a section, click on the plus sign in the box next to its title. To close (collapse) it, click on the minus sign in the box. Within each section, the Maple code is further organized into execution groups, identified by an enclosing square bracket to the left, that match the groupings of the examples in the text. Moving the cursor to any point in the group and pressing the *enter* key will execute all the commands in the group.

All the Maple commands and other Maple code in this book will appear in `typewriter font` so that code is distinguishable from other text. The appearance of Maple code in the book, then, will reflect that in a Maple worksheet, unless you have changed the default font for some reason.

I encourage you to begin writing animations of your own, however simple, as soon as you think you can. A good time to do that would be after you have read Chapter 4, Simple Animations. This will help you to start thinking about how you could use animation in your own work. The purpose of this book is to give you the means to implement your own creative ideas.

[1]Maple is a registered trademark of Waterloo Maple Inc., 57 Erb Street West, Waterloo, Ontario N2L 6C2, Canada.

Contents

Chapter 1

Getting Started

In this brief chapter, you will learn some Maple fundamentals. These include Maple's command line, arithmetic, and built-in functions. You will learn how to define new functions and how to assign names to Maple objects for storage. We will be limiting ourselves, in this and the next two chapters, to the features that are pertinent to our purpose, on the assumption that the reader will want to begin writing animations as soon as possible. For those who would like a less focused introduction to Maple, the excellent help facility provides one. You will learn how to access that, too.

Start the Maple application program now. On most systems, this is done by just double-clicking the Maple icon, as usual for starting application software. A blank worksheet will appear, ready for your input.

1.1 The basic command line

Maple's default prompt is the `>` sign, which indicates that Maple is waiting for input. Type commands directly after the prompt, then press the *enter* key. For example, the expression $23 \cdot 85 + 14/43$ is evaluated by the Maple command

```
> 23*85 + 14/43;
```

which generates the output

$$\frac{63310}{43}$$

Notice that the command (or statement) terminator in Maple is a semicolon. This signals the end of the input line. The semicolon and the *enter* key are easy to forget at first. If you forget the semicolon, you will get an error message warning of a premature end to the input. If you find yourself staring expectantly at the screen while nothing is happening, you probably forgot to press *enter*. The other command terminator is the colon, which suppresses output. If we had used a colon to end the statement, Maple would not have shown us the answer. There are times when we will want to do that, but this is not one of them.

Exponentiation is denoted by the `^` operator, as in

```
> 5^(1/2);
```

$$\sqrt{5}$$

and

```
> 2^50;
```

$$1125899906842624$$

If you need to edit a command line, just make the changes and press the *enter* key. Try that now. Move the cursor (insertion point) into the line that you just typed and change the 50 to 100 so that you have `2^100` now instead of `2^50`. Press *enter* to execute the new command. Also try `2^(2-7)`.

If you want to execute or re-execute any command, whether you are editing it or not, just move the cursor to a point within the statement and press *enter*. You can execute all the commands in an entire worksheet by choosing that option under **Execute** in the **Edit** menu.

1.2 A few words about Maple arithmetic

Notice that, in the examples, Maple used exact arithmetic; the answers are exact real numbers. Because each value entered was in integer form (no decimal points), the assumption is that you would like to see exact arithmetic where possible. This is consistent with the design goal to make Maple act as a mathematician might. If you'd rather see answers in floating-point (decimal) form, this is easily arranged. One way is to use a floating-point value somewhere in the expression. For example, you could enter 14.0 instead of 14 in

```
> 23*85 + 14.0/43;
```

$$1955.325581$$

or 2.0 instead of 2 in

```
> 5^(1/2.0);
```

$$2.236067977$$

and

```
> 2.0^50;
```

$$0.1125899907\,10^{16}$$

Another way to do this—and one that makes the instructions explicit—is to use Maple's `evalf` function. This will cause Maple to evaluate the result in floating-point form.

```
> evalf( 23*85 + 14/43 );
> evalf( 5^(1/2) );
> evalf( 2^50 );
```

$$1955.325581$$
$$2.236067977$$
$$0.1125899907\,10^{16}$$

The constant π is denoted `Pi` in Maple. Note the upper-case *P* and lower-case *i*. Expressions involving π can be converted to floating-point form using `evalf`, too. Notice the differences between the following:

```
> Pi/4;
> evalf( Pi/4 );
> Pi/4.0;
```

$$\frac{\pi}{4}$$
$$0.7853981635$$
$$0.2500000000$$

1.3 Comments

When you want to write a note to yourself, you wouldn't want Maple to try to process it as input. What you need in this case is a *comment*. Comments are indicated by a `#` sign. Maple ignores everything from the `#` to the end of the same line. (Although a particularly nasty or insulting comment might be difficult to ignore, even for Maple.) For example,

```
> 2^(89-1)*(2^89-1);   # Tenth perfect number.  R.E. Powers,
  1911.
```

$$191561942608236107294793378084303638130997321548169216$$

You can make the entire line a comment if you like.

```
> # You can make the entire line a comment if you like.
```

1.4 Assigning names to results

Often, it is useful to assign a result to a *name* so that it can be used later. A name consists of alphanumeric characters and underscores. So *s*, *parabola*, *Inverse_of_f*, *SecondDerivative*, and *Surface1* are all names. Names cannot begin with a number, so *2nd_derivative* is not a name. Maple is case-sensitive, so *f* and *F* are two different names. Avoid beginning a name with an underscore because Maple uses such names for its own purposes. If you inadvertently use a reserved word, such as *evalf*, as a name, Maple will tell you that you can't do it.

The assignment of a value, or anything else, to a name is accomplished using the `:=` operator. For example, we can assign a general quadratic equation to the name `y` with the command

```
> y := a*x^2 + b*x + c;
```

$$y := a\,x^2 + b\,x + c$$

We can then assign values to the coefficients `a`, `b`, and `c`, which are, themselves, names:

```
> a := 2;
> b := 3;
> c := 4;
```

$$a := 2$$
$$b := 3$$
$$c := 4$$

To see what `y` contains now, we just enter that name followed, as usual, by a semicolon,

```
> y;
```

$$2\,x^2 + 3\,x + 4$$

and we see that Maple has made the substitutions. The name `y` can be used in other expressions as well. For example,

```
> 5*y + 1;
```

$$10\,x^2 + 15\,x + 21$$

Names can hold all sorts of things. They can store constants, expressions, equations, inequalities, matrices, sequences, sets, and other objects. We will often use a name to store a plot or a sequence of them.

1.5 Built-in functions

The trigonometric functions and their inverses, the exponential and natural logarithmic functions, the hyperbolic trigonometric functions and their inverses, and practically any other function you are likely to want are all built in. For example,

```
> sqrt(256*x);
> abs(4*x);
```

$$16\sqrt{x}$$
$$4\,|x|$$

The parentheses are necessary. They delimit the argument of the function. Maple will accept the standard notation for n!, however. For example,

```
> 5!;
```

$$120$$

but this is just shorthand for `factorial(5)`.

Some examples of trigonometric functions and their inverses are

```
> sin(Pi/4);
> arcsin( sqrt(3)/2 );
> evalf( arcsin( sqrt(3)/2 ) );
```

$$\frac{\sqrt{2}}{2}$$

$$\frac{\pi}{3}$$
$$1.047197551$$

and

```
> tan(Pi/6);
> arctan(infinity);
```

$$\frac{\sqrt{3}}{3}$$

$$\frac{\pi}{2}$$

Examples of the exponential function and natural logarithm are

```
> exp(2*x);
> ln(3*x^2+1);
```

$$\mathbf{e}^{(2\,x)}$$

$$\ln(3\,\mathrm{x}^2+1)$$

The real number `e` is not stored in Maple as a constant, so the exponential function e^x is denoted `exp(x)` and not `e^x`. Of course, that means that `e` is readily available as

```
> exp(1);
> evalf(exp(1));
```

$$\mathbf{e}$$

$$2.718281828$$

Incidentally, although we will not need complex values in this book, Maple uses `I` (note the upper-case) to denote the complex number `i` such that $\mathtt{i}^2 = -1$. For example,

```
> (5+I)*(2+3*I);
```

$$7+17\,\mathtt{I}$$

and—you'll like this—Euler's relationship between the five principal constants of mathematics, 0, 1, π, `e`, and `i`:

```
> exp(Pi*I) + 1;
```

$$0$$

Information on these and other functions that are built into Maple is available by typing `?inifcn` at the Maple prompt. This will take you to a pertinent page in Maple's help facility. To get back from the help page to your worksheet, either close the help window or choose your worksheet under the **Window** menu.

1.6 Defining functions

Maple accepts the mapping notation for defining a *function*. The mapping arrow, `->`, is formed from a hyphen and a greater-than sign. For example, we can define the functions given by $\mathtt{f}(\mathtt{x}) = \mathtt{x}\cos\mathtt{x}$ and $\mathtt{g}(\mathtt{x}) = \mathtt{x}^2+1$ as the mappings $\mathtt{x} \mapsto \mathtt{x}\cos\mathtt{x}$ and $\mathtt{x} \mapsto \mathtt{x}^2+1$ as follows:

```
> f := x -> x*cos(x);
> g := x -> x^2 + 1;
```

$$\mathtt{f} := \mathtt{x} \rightarrow \mathtt{x}\cos(\mathtt{x})$$
$$\mathtt{g} := \mathtt{x} \rightarrow \mathtt{x}^2 + 1$$

which establishes `f` and `g` as names for mappings that send `x` to $\mathtt{x}\cos\mathtt{x}$ and to $\mathtt{x}^2 + 1$, respectively. We can then use familiar notation for evaluating these functions at specific points, forming composite functions, and performing other algebraic manipulations. For example,

```
> f(Pi/3);
> f(Pi)*g(2);
> f(g(x));
> g(f(x));
> g(f(Pi/6));
```

$$\frac{\pi}{6}$$
$$-5$$
$$(\mathtt{x}^2 + 1)\cos(\mathtt{x}^2 + 1)$$
$$\mathtt{x}^2\cos(\mathtt{x})^2 + 1$$
$$\frac{\pi^2}{48} + 1$$

Functions of several variables are defined just as naturally. For example,

```
> f := (x,y) -> x^3 + y^2;
```

$$\mathtt{f} := (\mathtt{x},\mathtt{y}) \rightarrow \mathtt{x}^3 + \mathtt{y}^2$$

defines `f` as a mapping that sends each ordered pair `(x,y)` to the value $\mathtt{x}^3 + \mathtt{y}^2$. Then we can use the usual notation to evaluate a function at a point:

```
> f(2,3);
> f(3,2);
```

$$17$$
$$31$$

1.7 Getting help and taking the tour

Maple has an elaborate, well-organized help facility. You have already accessed it if you entered `?inifcn` at the Maple prompt as suggested above. You can, of course, also get to it by using the **Help** menu. For example, choose **Topic Search...** and enter *evalf* for the topic. Then highlight *evalf* in the list of matching topics and click **OK**.

A convenient way to use the help facility is to move the cursor into the command about which you would like to have more information and then choose **Help on "*command*"** (or the choice may be **Help on Context**) from the **Help** menu. Try this by moving the cursor into `exp` where you have typed it into the worksheet. You can place the cursor between letters or highlight the whole function name, `exp`. Then choose **Help on "exp"** (or **Help on Context**) from the **Help** menu.

The **Introduction** in the **Help** menu is well worth a look. There is a **New User's Tour**, which can be enlightening even if you don't fit the description. It even includes help on **Help** under the **Using Help** option.

In this book, we will have occasion to use only a small subset of Maple's vast capabilities. Here is a sampling of some of Maple's other talents:

```
> factor( x^5+8*x^3-3*x^4-24*x^2+12*x-36 );
```

$$(x-3)\,(x^2+2)\,(x^2+6)$$

```
> expand( (3*x-5)^4 );
```

$$81\,x^4-540\,x^3+1350\,x^2-1500\,x+625$$

```
> limit( sin(x)/x, x=0 );
```

$$1$$

```
> limit( (x^4+5)/(x+2), x=-infinity );
```

$$-\infty$$

```
> solve( x^3-3*x^2-5*x+15=0 );  # Find exact solutions,
  if possible.
```

$$3, \sqrt{5}, -\sqrt{5}$$

```
> fsolve( exp(2*x)-3*cos(x)=0, x=0..2 );  # Find an
  approximate solution in the interval [0,2].
```

$$0.4874035006$$

```
> sum( 1/i^4, i=1..infinity );
```

$$\frac{4}{90}$$

```
> int( x^2*sin(x), x=0..Pi/2 );  # Evaluate the definite
  integral.
```

$$-2$$

```
> int( x^2*sin(x), x );  # Evaluate the indefinite
  integral.
```

$$-x^2\cos(x) + 2\cos(x) + 2\,x\sin(x)$$

1.8 Saving, quitting, and returning to a saved worksheet

To save a worksheet or to exit (or quit) Maple, just make those selections under the **File** menu. To reopen a saved worksheet, just double-click on its icon. Alternatively, you can start Maple and open the worksheet from the **File** menu.

Whenever you reopen a saved worksheet, although it appears to be in the state in which you left it, this is not really the case. All the output is there, including graphs and working animations. However, none of the assignments that you might have made to names are active and none of the packages called by `with` statements (described in Section 2.3) are loaded. You'll need to re-execute any of those statements that you would like to have in effect again. As mentioned in Section 1.1, if you want to re-execute all the statements in a worksheet, a convenient way to do that is to choose that option under the **Edit** menu.

On the other hand, you may have a worksheet that is in good working order, and you just want to use the output in it—typically graphs and animations. The output will be ready to use. In this case, there is no need to re-execute any of the statements.

Chapter 2

The Plot

Shirley Kolmer of Saint Louis University, whose many talents included dry wit, once said to our graduate class in number theory, "Primes tend to be odd." Similarly, animations tend to be plots. In this chapter, you will learn to plot functions of one and two variables, curves and surfaces represented by parametric equations, and individual points. You will also learn many optional features by which you can produce a high-quality plot that suits your needs and conforms to your preferences. These options will be your artist's brushes.

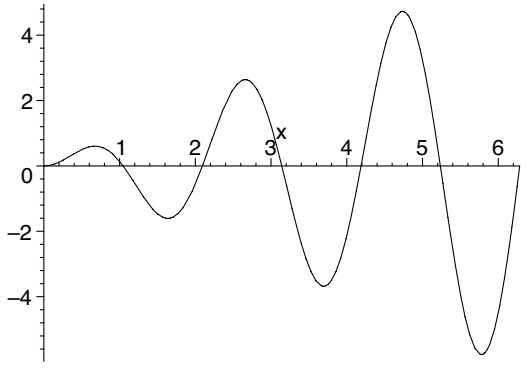
2.1 The basics

In its simplest form, a plotting procedure needs a specification for a function and a domain. A function of a single variable may be plotted using the two-dimensional `plot` procedure, which has the form

`plot( f(x), x=a..b,` *options* `)`

where the options will be described below. For example, the function $\mathtt{f(x)} = \mathtt{x}\sin 3\mathtt{x}$ can be plotted on the interval $[\,0,2\,]$ by entering

```
> plot( x*sin(3*x), x=0..2*Pi );
```



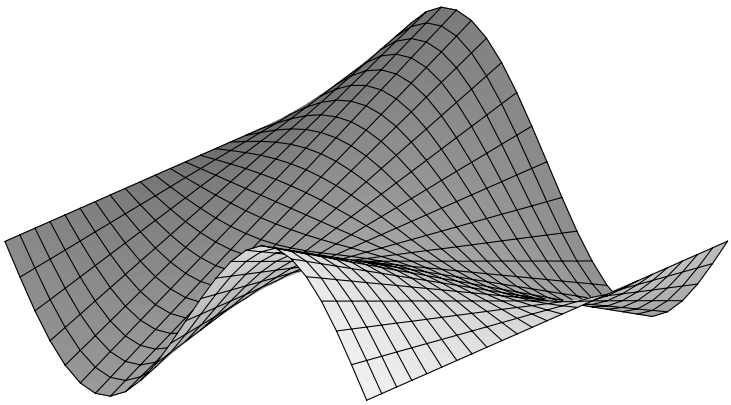
Be sure to capitalize *Pi*.

A function of two variables may be plotted with the three-dimensional `plot3d` procedure, which has the form

`plot3d( f(x,y), x=a..b, y=c..d,` *options* `)`

For example, we may plot the function `f(x,y)` = `x sin y` on the rectangular region $[-8,8] \times [-\ ,\ ]$ using

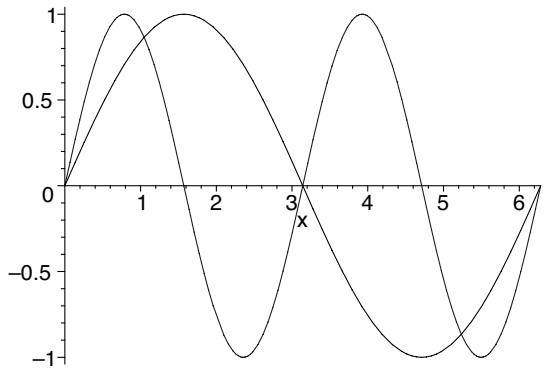
```
> plot3d( x*sin(y), x=-8..8, y=-Pi..Pi );
```



By clicking and dragging with the mouse, you can rotate a three-dimensional plot such as this one so that you can see it from any viewpoint. Try it.

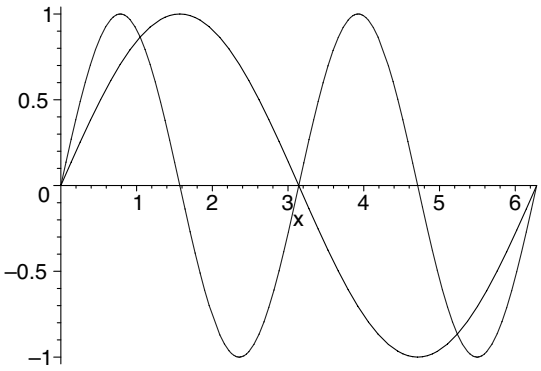
To plot two or more functions on the same axes, in the two-dimensional case you can use either *list* or *set* notation. A *list* is an ordered n-tuple enclosed in square brackets; a *set* is an unordered n-tuple enclosed in curly brackets. For example, sin x and sin 2x can be plotted together using either

```
> plot( [sin(x), sin(2*x)], x=0..2*Pi );
```



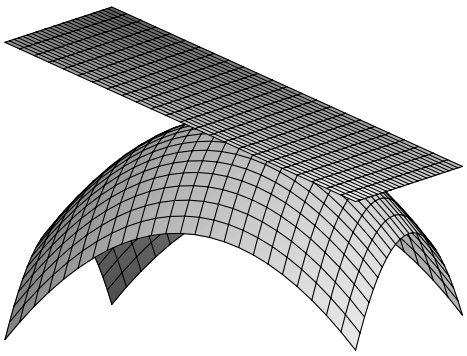
or

```
> plot( {sin(x), sin(2*x)}, x=0..2*Pi );
```



In the three-dimensional case, use only set notation:

```
> plot3d( {4-x^2-2*y^2, 6-4*y}, x=-4..4, y=-3..3 );
```



2.2 Parametric forms

There are three objects defined by parametric equations to consider: curves in two dimensions, curves in three dimensions, and surfaces in three dimensions. Here, we will discuss curves in two dimensions and surfaces in three. Chapter 9 is about curves in three dimensions.

A parametric form is represented in Maple as a list. In two dimensions, the syntax is

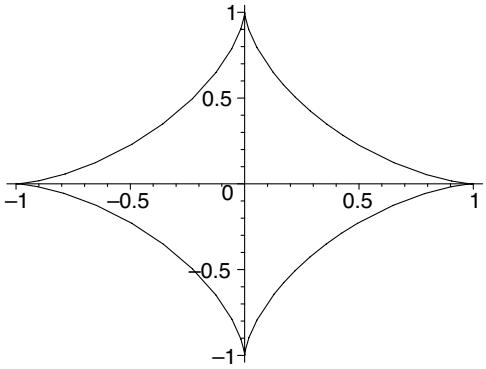
`plot( [f(t), g(t), t=a..b]`, *options* `)`

For example, the curve whose parametric equations are

$$x = \cos^3$$
$$y = \sin^3$$

may be plotted with

```
> plot( [cos(theta)^3, sin(theta)^3, theta=-Pi..Pi] );
```



The syntax for surfaces in three dimensions is

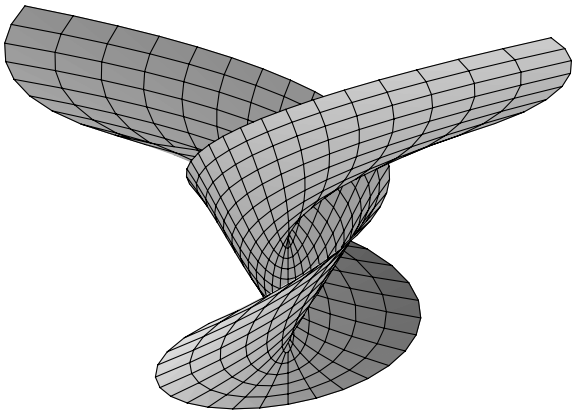
`plot3d( [f(s,t), g(s,t), h(s,t)], s=a..b, t=c..d,` *options* `)`

For example, the surface defined parametrically by

$$\begin{aligned} x &= 2t - 3s^2 \sin t \\ y &= st \\ z &= 2s - 3\cos t \end{aligned}$$

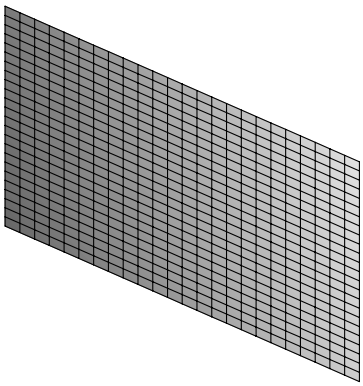
may be plotted using

```
> plot3d( [2*t-3*s^2*sin(t), s*t, 2*s-3*cos(t)], s=-2..2,
  t=-2..2 );
```



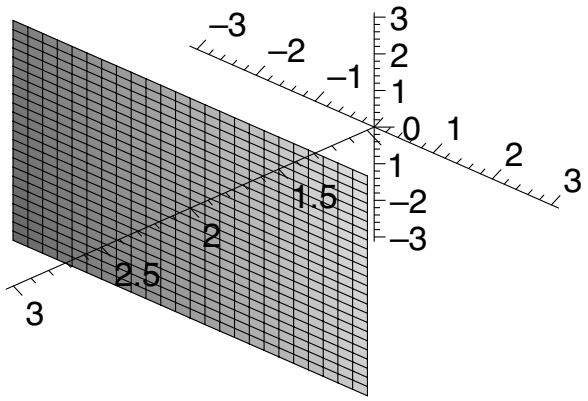
Notice where the domains—expressions of the form `a..b` are called *ranges* in Maple—for the parameters are given: inside the list in the two-dimensional case, but outside in the three-dimensional case. Two-parameter forms are useful for defining surfaces in three dimensions that, as the previous example, are not functions of `x` and `y`. In particular, we can use them to plot vertical planes. For example, the plane $x = 2$ may be plotted using

```
> plot3d( [2, s, t], s=-3..3, t=-3..3 );
```



As you will have noticed, Maple does not automatically include axes in three-dimensional plots, and that makes it difficult to get our bearings and, in the present case, to verify that the plane we have just plotted is the plane it is supposed to be. One way to add axes is to include `axes=normal` in the call to the `plot3d` procedure:

```
> plot3d( [2, s, t], s=-3..3, t=-3..3, axes=normal );
```

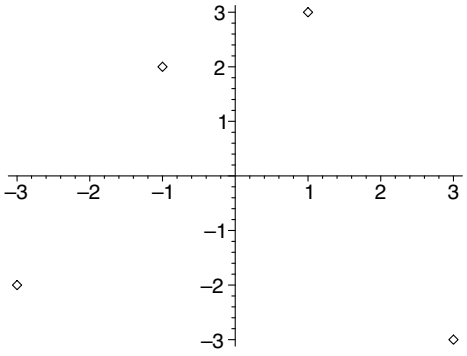


Other options for adding axes are described below in Sections 2.5 and 2.12.

2.3 Plotting points and using the `plots` package

An individual point is represented as a list. So, in two dimensions, the point `(a,b)` is denoted `[a,b]`. To plot several points, group them as in

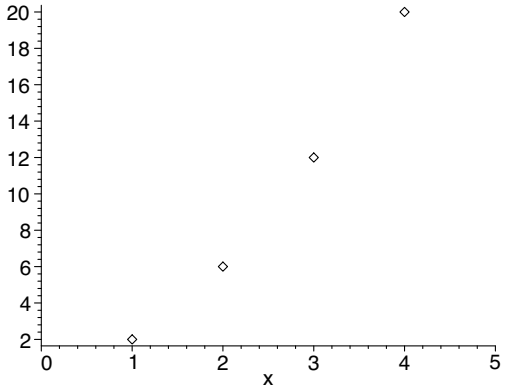
```
> plot( {[-1,2],[1,3],[-3,-2],[3,-3]}, style=point );
```



and

```
> f := x -> x^2 + x;
> plot( {[1,f(1)],[2,f(2)],[3,f(3)],[4,f(4)]}, x=0..5,
  style=point );
```

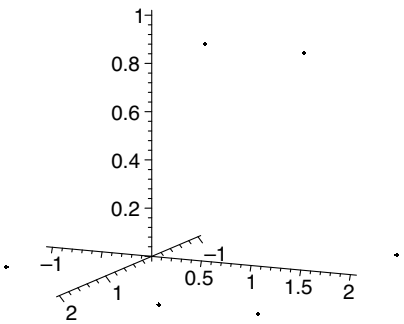
$$\mathtt{f} := \mathtt{x} \rightarrow \mathtt{x}^2 + \mathtt{x}$$



where `style=point` causes the points to be plotted without any line segments connecting them. (`plot`'s natural instinct is to connect points with segments.) The specification of the domain for `x` is optional.

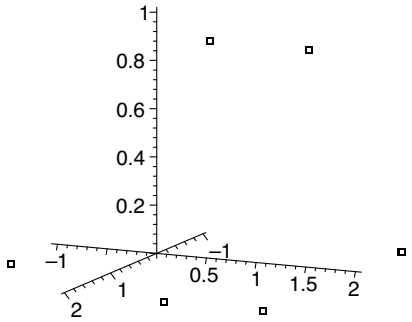
In three dimensions, the point `(a,b,c)` is denoted `[a,b,c]`. For example,

```
> plot3d( {[1,-1,0],[-1,2,0],[2,2,0],[2,1,0],[1,1,1],
  [1,2,1]}, x=-1..2, y=-1..2, style=point, axes=normal );
```



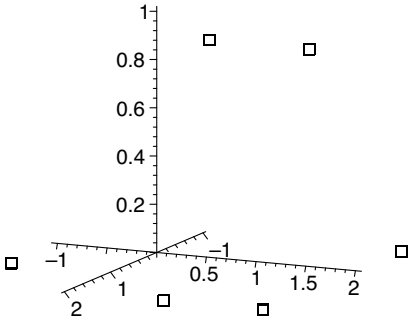
where, this time, the domain specifications are required. If the default representations of the points are too tiny to be seen easily, it will help to specify the symbol to use when plotting them. For example,

```
> plot3d( {[1,-1,0],[-1,2,0],[2,2,0],[2,1,0],[1,1,1],
  [1,2,1]}, x=-1..2, y=-1..2, style=point, symbol=box,
  axes=normal );
```



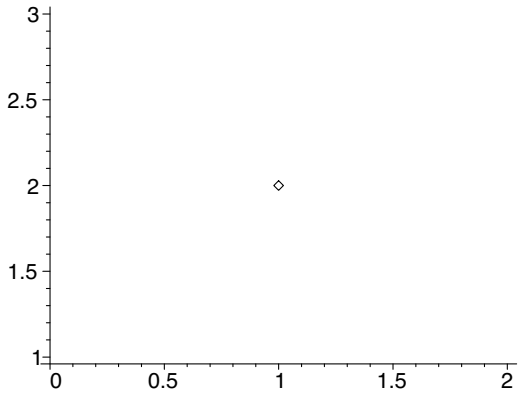
Other options for the plotting symbol, in both two and three dimensions, are `circle`, `cross`, `point`, or `diamond`, the default symbol depending on the plotting device. You can also specify a size, in units of points, with the `symbolsize` option. This option does not affect the symbol `point`, however. The default size, in both two and three dimensions, is 10 points.

```
> plot3d( {[1,-1,0],[-1,2,0],[2,2,0],[2,1,0],[1,1,1],
  [1,2,1]}, x=-1..2, y=-1..2, style=point, symbol=box,
  symbolsize=18, axes=normal );
```



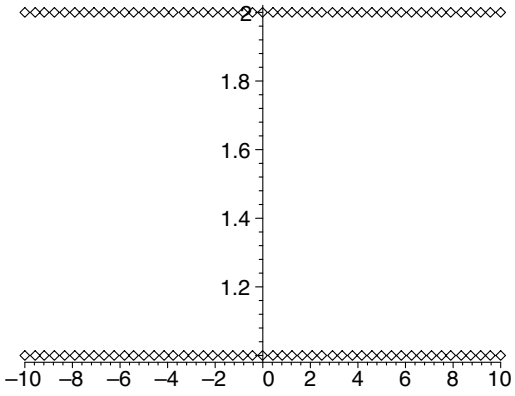
Use set notation (or, alternatively in two dimensions, list notation) to group the points, even if you are only plotting a single point. Otherwise, Maple will think you want a list of two constant functions to be plotted, and you will get two horizontal lines of points. You want, for example,

```
> plot( {[1,2]}, style=point );
```



and not

```
> plot( [1,2], style=point );
```



(Actually, you can get by without the set brackets when plotting a single point in three dimensions with `plot3d` because Maple will think you want to plot a parametric form whose components are constant functions, but it's a good idea to use the brackets anyway.)

There is a procedure, `pointplot`, that eliminates the need for the `style=point` specification. To use this procedure, though, we will need to summon the `plots` package by issuing the command

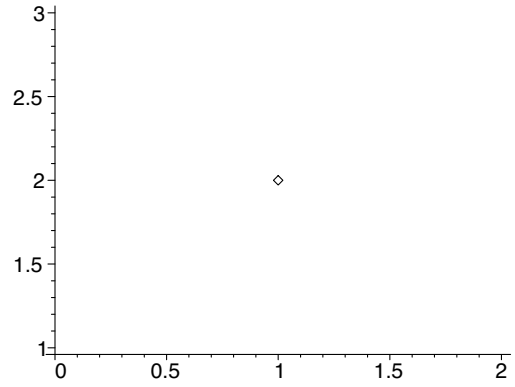
```
> with( plots );
Warning, the name changecoords has been redefined
```

[*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* ,

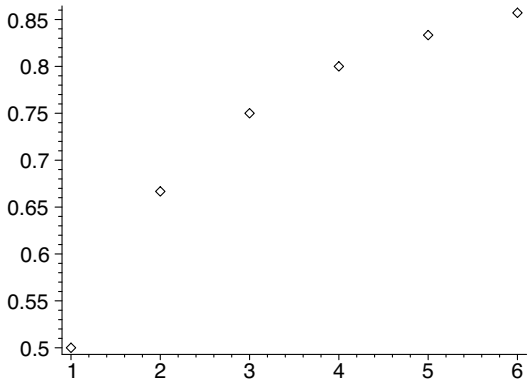
replot , *rootlocus* , *semilogplot* , *setoptions* , *setoptions3d* , *spacecurve* , *sparsematrixplot* , *sphereplot* , *surfdata* , *textplot* , *textplot3d* , *tubeplot*]

The `pointplot` procedure is listed together with the others included in the `plots` package. The warning is notification that one of the procedures available before the `with` statement was executed has been redefined using the definition of that procedure as it exists within the `plots` package. Although it is useful sometimes to see this list, you will usually want to suppress the output of the `with` statement by using a colon instead of a semicolon as the statement terminator. The `with( plots )` statement needs to be executed only once per session for all the procedures in the `plots` package to be available for that session. We're going to include it in each example, however. That way, small pieces of example code will stand alone. When you refer to them later, you won't need to be concerned that you are missing something from the larger context in which the examples were given. Examples using `pointplot` are

```
> with( plots ):
> pointplot( {[1,2]} );
> f := n -> n/(n+1);
> pointplot( {[1,f(1)],[2,f(2)],[3,f(3)],[4,f(4)],[5,f(5)],
  [6,f(6)]} );
```

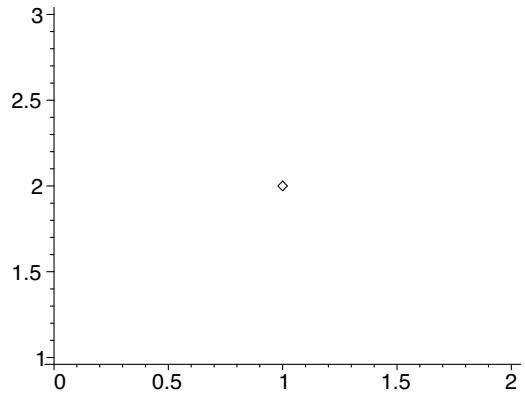


$$f := n \rightarrow \frac{n}{n+1}$$

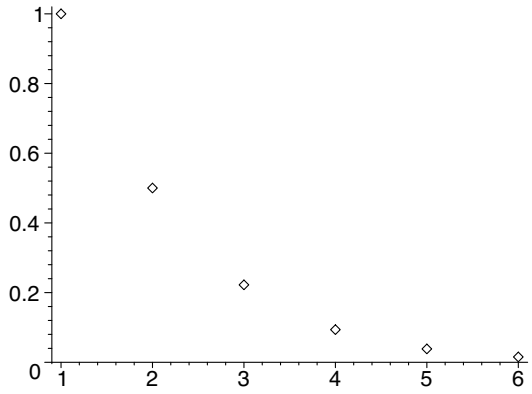


The `pointplot` procedure accepts either a set or list of points. It will also accept a flat list with an even number of elements, which will be paired to form points. For example,

```
> with( plots ):
> pointplot( [1,2] );
> f := n -> n!/n^n;
> pointplot( [1,f(1), 2,f(2), 3,f(3), 4,f(4), 5,f(5),
  6,f(6)] );
```

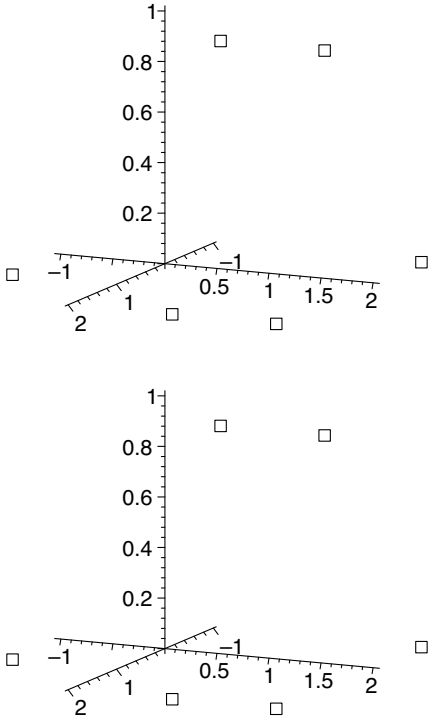


$$\mathrm{f} := \mathrm{n} \to \frac{\mathrm{n!}}{\mathrm{n}^n}$$



There is a procedure `pointplot3d`, also within the `plots` package, for plotting points in three dimensions. It eliminates the need for both `style=point` and domain specifications. The `pointplot3d` procedure accepts a set or list of points, or a flat list whose length is a multiple of three. For example,

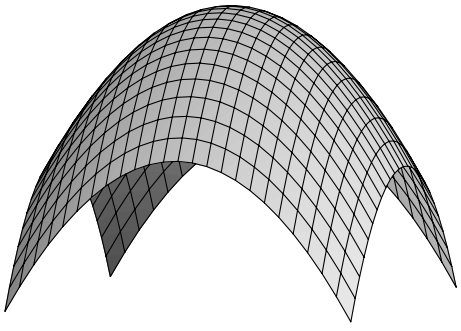
```
> with( plots ):
> pointplot3d( {[1,-1,0],[-1,2,0],[2,2,0],[2,1,0],[1,1,1],
  [1,2,1]}, symbol=box, symbolsize=18, axes=normal );
> pointplot3d( [1,-1,0, -1,2,0, 2,2,0, 2,1,0, 1,1,1, 1,2,1],
  symbol=box, symbolsize=18, axes=normal );
```

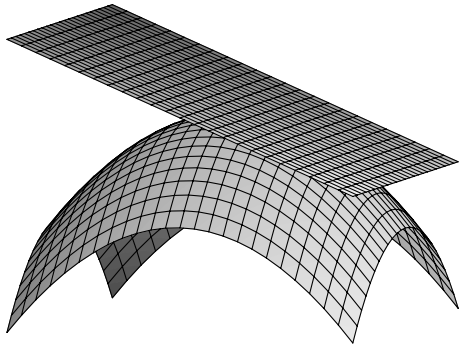


2.4 Storing and displaying plots

Often, it is useful to assign a plot to a name so that it can be used later. This is accomplished with the assignment statement, as we have done for other objects. Typically, plots are stored in names, then displayed later with the `display` procedure, which is among the other procedures in the `plots` package. An example of this is

```
> with( plots ):
> Surface := plot3d( 4-x^2-2*y^2, x=-4..4, y=-3..3 ):
> TangentPlane := plot3d( 6-4*y, x=-4..4, y=-3..3 ):
> display( Surface );
> display( Surface, TangentPlane );
```





Notice the colons at the ends of the lines whose output we don't need to see.

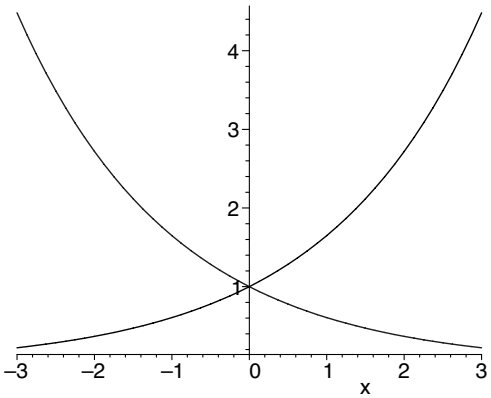
2.5 The plot thickens

The examples above that included the specifications `axes=normal`, `style=point`, `symbol=box`, and `symbolsize=18` took advantage of some of the *options* available in plotting. You can certainly improve the appearance of your plots and tailor them to your purpose by exercising the available options. For example, since the two-dimensional `plot` procedure, by default, displays axes, but the three-dimensional `plot3d` procedure does not, we've usually included the option `axes=normal` in `plot3d` statements somewhere after the specifications for the domain. Other choices for the axes are `boxed`, `framed`, or `none`. The two-dimensional `plot` procedure has the same choices for the axes, but it defaults to `axes=normal`.

Lines and curves appearing in the plot output can be made heavier using the `thickness` option. This is useful if you are using a projection system in a large room where some people may be sitting far away from the screen. For example,

```
> f := x -> exp(x/2);
> plot( {f(x), f(-x)}, x=-3..3, thickness=3 );
```

$$\mathrm{f} := \mathrm{x} \rightarrow \mathbf{e}^{(1/2\,x)}$$

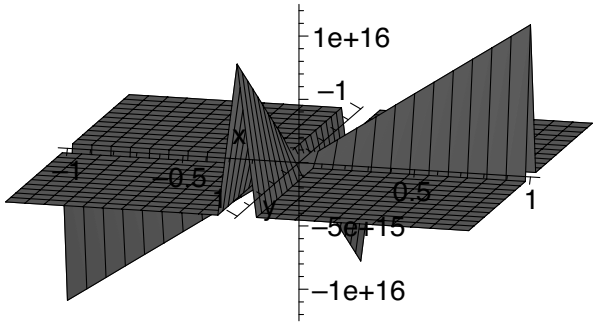


(Recall that the exponential function is entered as `exp(x)` and not as `e^x`.) From thinnest to thickest, the choices for `thickness` are 0,1,...,15. The default is 0.

2.6 Smoothing plots

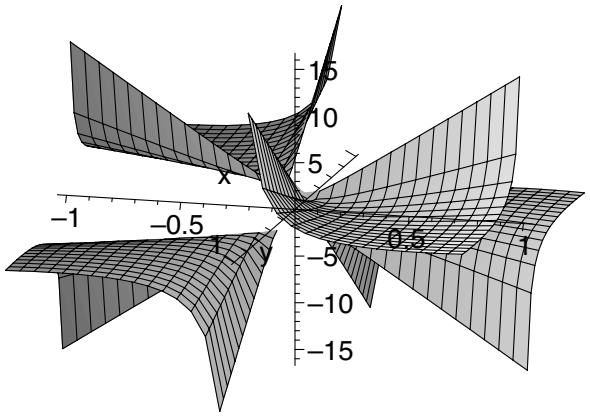
Sometimes a three-dimensional plot can look a little chunky or ragged, particularly around discontinuities. Maple does its work by sampling points from the domain, so you can improve the appearance by instructing Maple to take a larger sample. The default number of points is 625, a 25-by-25 grid. There are two ways: with `numpoints` and with `grid`. Notice the difference between

```
> plot3d( (x^2+y^2)/sin(x*y), x=-1..1, y=-1..1,
  axes=normal );
```



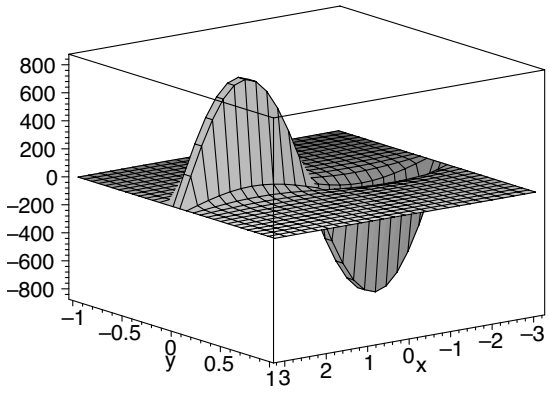
and

```
> plot3d( (x^2+y^2)/sin(x*y), x=-1..1, y=-1..1, axes=normal,
  numpoints=1000 );
```



When `numpoints` is set to `n`, Maple uses $\sqrt{\texttt{n}}$ points in each of the two domain intervals ($[-1,1]$ for both `x` and `y` in the example above). For more control over this, you can use `grid`. For example,

```
> plot3d(sin(x)/y^2, x=-Pi..Pi, y=-1..1, axes=boxed,
  grid=[25,30] );
```



instructs Maple to use 25 points in the interval of x-values and 30 in the interval of y-values.

Since animations can be very large, space (memory) limitations can become a problem, in which case it is helpful to specify a number of points fewer than the default. If possible, though, choose at least the default number. A smooth, graceful surface is an appealing thing to behold.

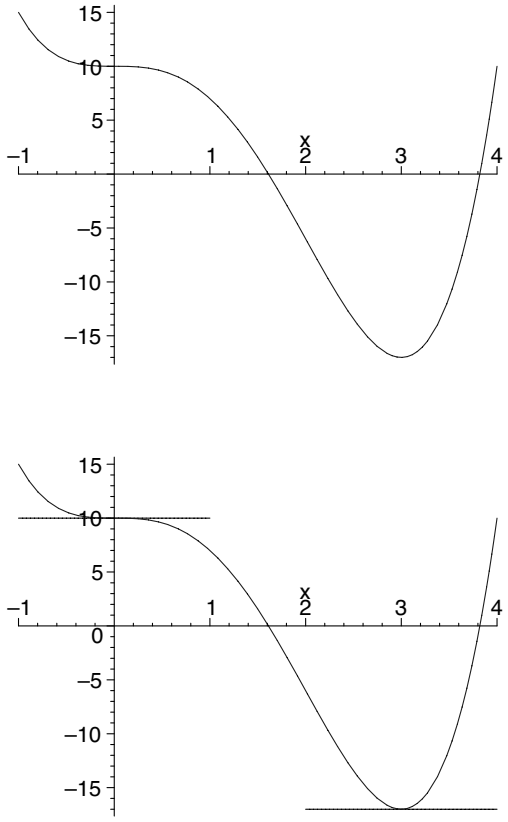
The `numpoints` option is available for two-dimensional plots as well, and defaults to 50. The `plot` procedure of two dimensions, however, is smart enough to sample more points wherever a curve changes quickly. This is called *adaptive* plotting. If, for some reason, you would like to turn it off, specify `adaptive=false` in your `plot` statement.

2.7 Color

Another option is `color`. Predefined colors are: `aquamarine`, `black`, `blue`, `navy`, `coral`, `cyan`, `brown`, `gold`, `green`, `gray` (or `grey`), `khaki`, `magenta`, `maroon`, `orange`, `pink`, `plum`, `red`, `sienna`, `tan`, `turquoise`, `violet`, `wheat`, `white`, or `yellow`. For example,

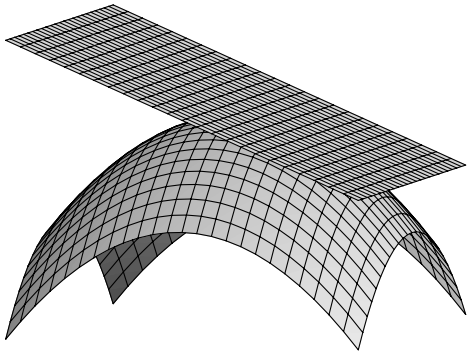
```
> with( plots ):
> f := x -> x^4 - 4*x^3 + 10;
> Curve := plot( f(x), x=-1..4, color=red ):
> HorizontalTan1 := plot( f(0), x=-1..1, color=blue ):
> HorizontalTan2 := plot( f(3), x=2..4, color=blue ):
> display( Curve );
> display( Curve, HorizontalTan1, HorizontalTan2 );
```

$$\mathrm{f} := \mathrm{x} \rightarrow \mathrm{x}^4 - 4\,\mathrm{x}^3 + 10$$



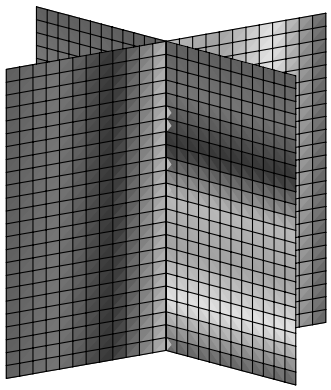
It is helpful to use the semicolon at the end of function definitions, even though you don't really need to see the output. It allows Maple to echo the function in mathematical notation (pretty printing) so that you can check that you have entered it correctly. An example of using color in a three-dimensional plot is

```
> with( plots ):
> Surface := plot3d( 4-x^2-2*y^2, x=-4..4, y=-3..3 ):
> TangentPlane := plot3d( 6-4*y, x=-4..4, y=-3..3,
  color=wheat ):
> display( Surface, TangentPlane );
```



Three-dimensional plots allow the use of variable colors. You might, for example, want to vary the color of a surface according to its z-value. (This can also be done using the `shading` option, `shading=z`, which we discuss next.) A more sophisticated use of this capability is to demonstrate, perhaps to linear algebra students, that two planes really can intersect in a single point in four dimensions. Let the first three coordinates `x`, `y`, and `z` of the point `(x,y,z,w)` be the usual three-dimensional coordinates, and let the fourth coordinate `w` be represented by color. Then the output of

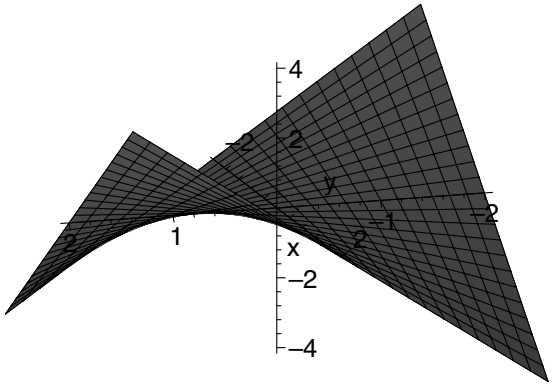
```
> with( plots ):
> Plane1 := plot3d( [x,0,z], x=-2..2, z=-2..2, color=x ):
> Plane2 := plot3d( [0,y,z], y=-2..2, z=-2..2, color=z ):
> display( Plane1, Plane2 );
```



shows the two planes sharing just one point in four dimensions, because only one point on each plane has the same color (`w`-coordinate) as well as the same `x`-, `y`-, and `z`-coordinates. For more information on the `color` option, particularly on how to define your own colors, enter `?plot,color` or `?plot3d,colorfunc` at the Maple prompt.

Three-dimensional plots allow a `shading` option. Although this is largely a matter a personal preference, it can be used to make three-dimensional plots a little easier to interpret. For example, in

```
> plot3d( x*y, x=-2..2, y=-2..2, axes=normal, shading=z );
```

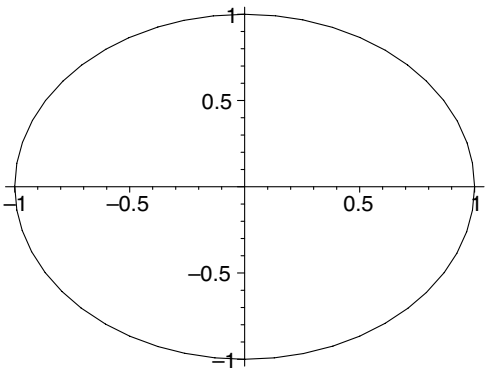


the `shading=z` option renders the surface so that the higher you are, the redder you are, and, as in life, the lower you are, the bluer you are. Other options for `shading` are `xyz`, `xy`, `zgrayscale`, `zhue`, or `none`. Experiment a little.

2.8 Scaling

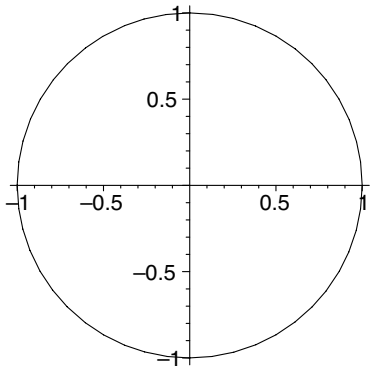
Maple will scale a plot to fit the plot window, adjusting the horizontal dimension independently of the vertical dimension. This is sometimes undesirable, but, if so, it is easily prevented by using `scaling=constrained`. For example,

```
> plot( [cos(theta), sin(theta), theta=0..2*Pi] );
```



produces an ellipse in a non-square window, but

```
> plot( [cos(theta), sin(theta), theta=0..2*Pi],
  scaling=constrained );
```



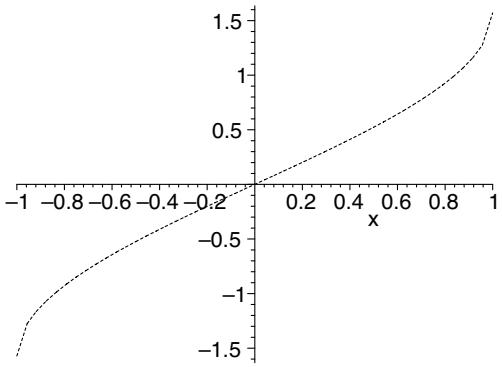
accurately produces a circle. The default plot window is square in Maple 8, so this particular plot should look fine, with or without constrained scaling. In Maple 7, though, the default plot window is not a square, so you may want to use constrained scaling or just reshape the window. To change the shape, or size, of the plot window, click on the plot to select it, then click and drag one of the small, black squares at the corners and edges of the window.

The other choice for the `scaling` option is `unconstrained`, which is default. Both two- and three-dimensional plots offer the `scaling` option.

2.9 Plotting with style

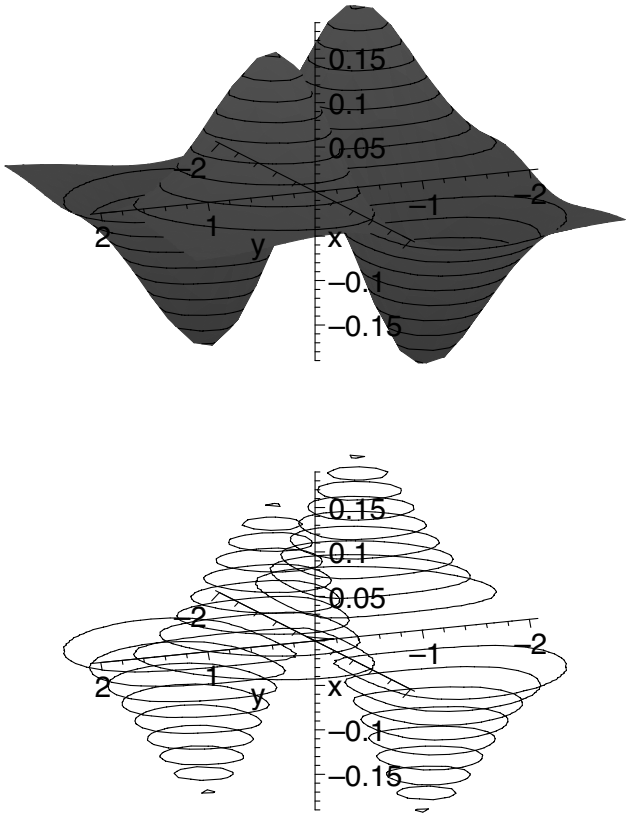
We have already encountered the `style` option; we used it when we wanted to plot individual points (`style=point`). The choices for the two-dimensional `plot` procedure are: `point`, `line`, `patch`, or `patchnogrid`. Default is `line`. The `patch` and `patchnogrid` options are used for plotting polygons, which we will do in Sections 8.5 and 10.1. With the `linestyle` option (also available for three-dimensional plots), you can specify `SOLID`, `DOT`, `DASH`, or `DASHDOT` (upper-case required) or 1, 2, 3, or 4, respectively. For example,

```
> plot( arcsin(x), x=-1..1, linestyle=DOT );
```



For the three-dimensional `plot3d` procedure, the choices for `style` are: `point`, `hidden`, `patch`, `wireframe`, `contour`, `patchnogrid`, `patchcontour`, or `line`. Default is `patch`. The choices `patchcontour` and `contour` provide a handy way to show level curves of a surface:

```
> plot3d( x*y*exp(-x^2-y^2), x=-2..2, y=-2..2, axes=normal,
  shading=z, style=patchcontour );
> plot3d( x*y*exp(-x^2-y^2), x=-2..2, y=-2..2, axes=normal,
  shading=z, style=contour );
```

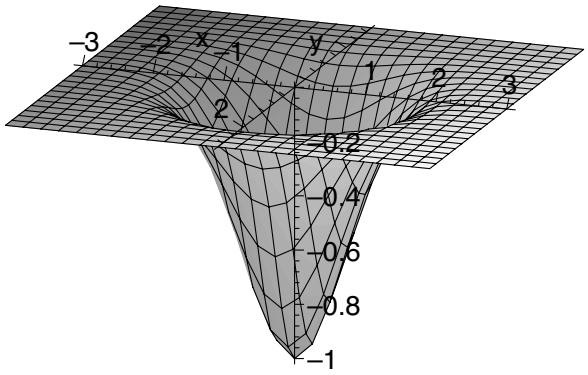


You can produce an instant contour plot of the surface by rotating either plot (by clicking and dragging with the mouse) so that the viewer is looking straight down the z-axis, seeing the projections of the level curves onto the xy-plane.

2.10 Adjusting your point of view

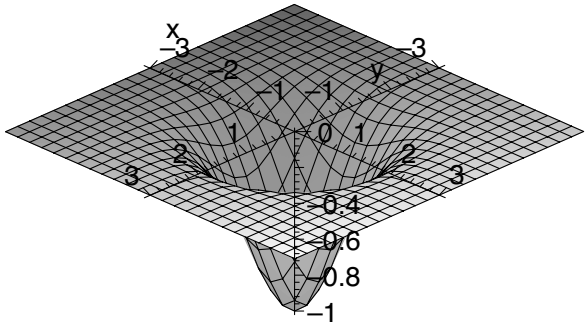
Although a three-dimensional plot can be rotated so that it can be seen from any viewpoint, the viewpoint can also be prescribed. The option that accomplishes this is `orientation=[ , ]`, where and are the angles, in degrees, of spherical coordinates. That is, is the angle in the xy-plane measured counterclockwise from the positive end of the x-axis, and is the angle measured from the positive end of the z-axis downward (toward the negative end of the z-axis). Default is `orientation=[45,45]`. For example, compare

```
> plot3d( -1/exp(x^2+y^2), x=-3..3, y=-3..3, axes=normal,
  orientation=[20,65] );
```



with the default

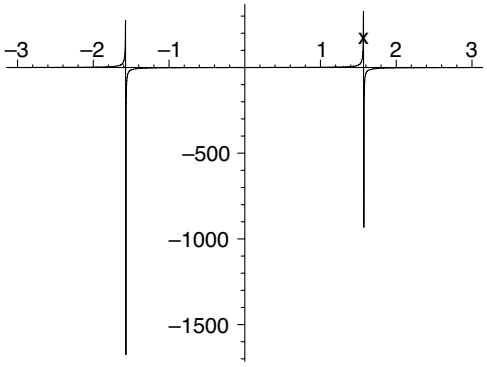
```
> plot3d( -1/exp(x^2+y^2), x=-3..3, y=-3..3, axes=normal );
```



2.11 A limited view

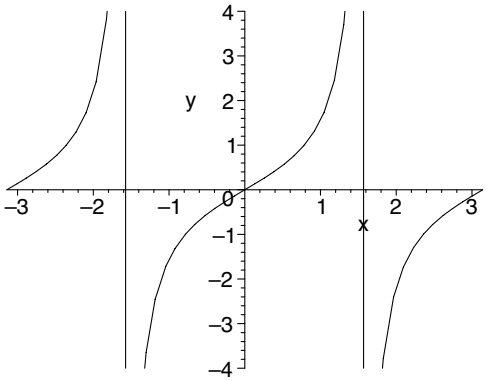
The magnitude of a function's value at sampled points can be so large as to obscure the salient features of its graph. For example,

```
> plot( tan(x), x=-Pi..Pi );
```



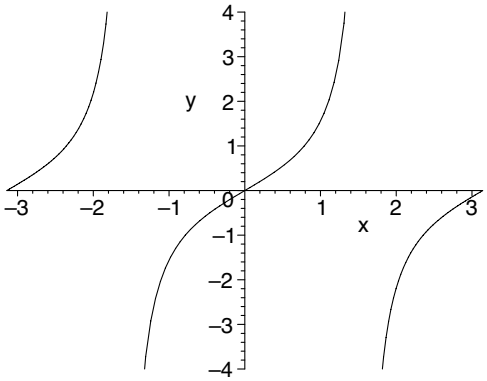
This plot can be improved by specifying a range

```
> plot( tan(x), x=-Pi..Pi, y=-4..4 );
```



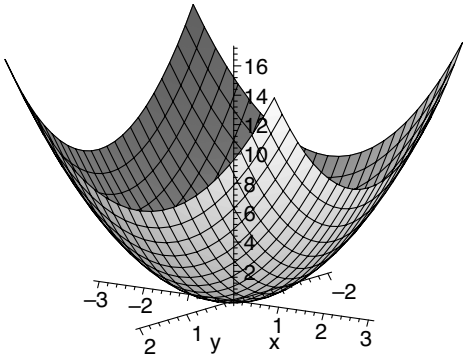
and further improved by warning Maple about the discontinuities

```
> plot( tan(x), x=-Pi..Pi, y=-4..4, discont=true );
```



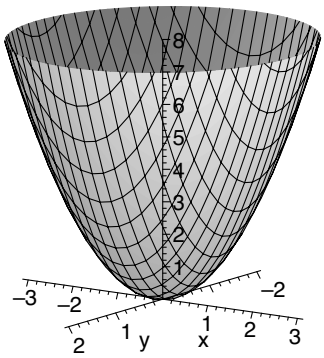
For three-dimensional plots, the plot can be limited via the `view` option, which takes either the form `view=`*zmin*`..`*zmax* or `view=[`*xmin*`..`*xmax*`,` *ymin*`..`*ymax*`,`*zmin*`..`*zmax*`]`. For example, a plot of an elliptic paraboloid

```
> plot3d( 2*x^2+y^2, x=-2..2, y=-3..3, axes=normal );
```



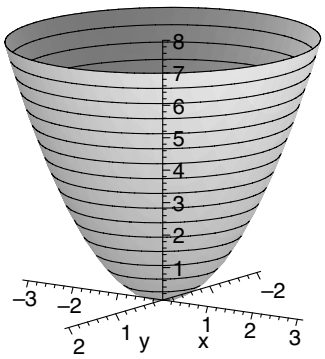
can be "improved" with

```
> plot3d( 2*x^2+y^2, x=-2..2, y=-3..3, axes=normal,
  view=0..8 );
```



Whether this is an improvement is a matter of opinion. The second plot does convey that the horizontal cross-sections are ellipses without losing the parabolic vertical cross-sections. If our goal is to emphasize only the horizontal cross-sections, we can use the `style` option:

```
> plot3d( 2*x^2+y^2, x=-2..2, y=-3..3, axes=normal,
  style=patchcontour, view=0..8 );
```

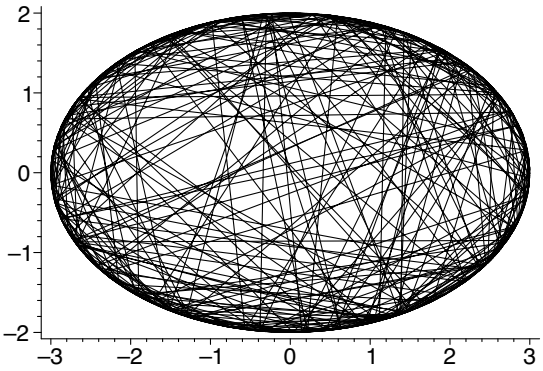


For two-dimensional plots, Maple will accept the similar `view=`*ymin*`..`*ymax* or `view=[`*xmin*`..`*xmax*`,`*ymin*`..`*ymax*`]` options.

2.12 Tailoring the axes

As mentioned in Section 2.5, the choices for the `axes` option are `normal`, `boxed`, `framed`, or `none`, with two-dimensional plots defaulting to `axes=normal` and three-dimensional ones defaulting to `axes=none`. For example,

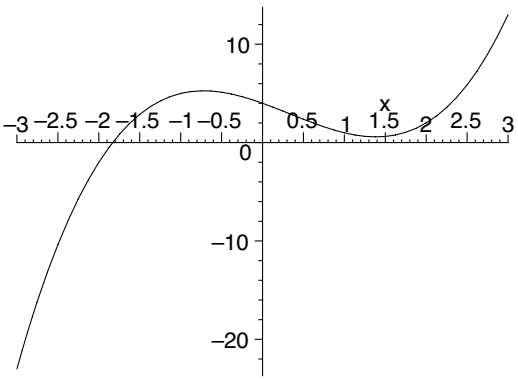
```
> plot( [3*cos(1/t), 2*sin(1/t), t=-0.1..0.1], axes=framed,
  scaling=constrained );
```



Maple accepts `box` and `frame`, instead of `boxed` and `framed`.

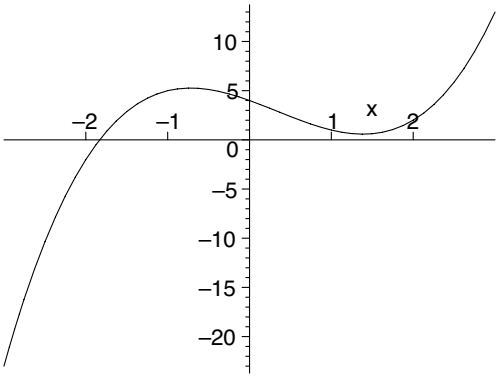
The `tickmarks=[`*xticks*`,`*yticks*`]` option will cause Maple to use at least *xticks* marks along the horizontal axis and at least *yticks* along the vertical axis. In three dimensions, the option `tickmarks=[`*xticks*`,`*yticks*`,`*zticks*`]` behaves similarly. For example,

```
> plot( x^3-x^2-3*x+4, x=-3..3, tickmarks=[10,4] );
```



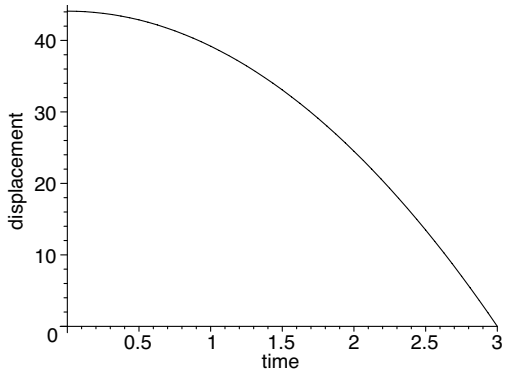
For two-dimensional plots, if you want to specify a minimum number of marks along only one axis, you can use the `xtickmarks=`*xticks* or the `ytickmarks=` *yticks* option. These options also accept a specific list of values. For example,

```
> plot( x^3-x^2-3*x+4, x=-3..3, xtickmarks=[-2,-1,0,1,2] );
```



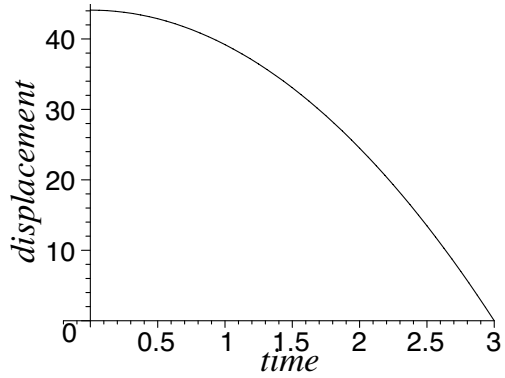
By default, Maple will label the axes using the same names as the variables in the function being plotted. You can override this using the options `labels=[`*string1*`,`*string2*`]` and `labels=[`*string1*`,`*string2*`,`*string3*`]` for two- and three-dimensional plots, respectively. As implied, each entry in the list should be a *string*, which consists of any characters enclosed in double quotes. (There is a limit to the length of a string, but it's huge.) By default, the labels print horizontally. You can change that with the `labeldirections=[`d_1`,`d_2`]` option, where d_1 and d_2 are either `vertical` or `horizontal`. For example,

```
> plot( -4.9*t^2+44.1, t=0..3,
  labels=["time","displacement"],
  labeldirections=[horizontal,vertical] );
```



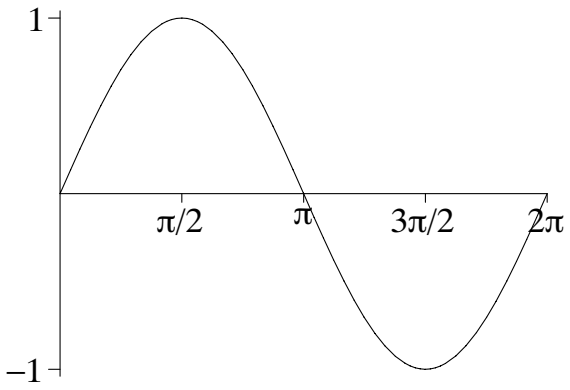
The font used for the axis labels can be controlled using `labelfont`, and the font for the values at the tick marks can be specified using `axesfont` or, more generally, using `font`, which controls all text in the plot. All three of these specifications are in the form of a list, `[`*family*`,`*style*`,`*size*`]`, where *family* is either `TIMES`, `COURIER`, `HELVETICA`, or `SYMBOL` (upper-case required). For the family `TIMES`, *style* can be `ROMAN`, `BOLD`, `ITALIC`, or `BOLDITALIC`. For the `COURIER` and `HELVETICA` families, *style* may be `BOLD`, `OBLIQUE`, `BOLDOBLIQUE`, or omitted altogether. The `SYMBOL` family does not allow a choice of style. The *size* units are points. An example is

```
> plot( -4.9*t^2 + 44.1, t=0..3,
  labels=["time","displacement"],
  labeldirections=[horizontal,vertical],
  labelfont=[TIMES,ITALIC,18], axesfont=[HELVETICA,14] );
```



I gave a talk once in Baltimore, and someone in the audience asked me whether I knew how to get the scale along the x-axis to be measured in units of π so that, for example, a sine function would have a maximum at a point labeled $\pi/2$. I didn't. But I do now:

```
> plot( sin(x), x=0..2*Pi, xtickmarks=evalf([Pi/2="p/2",
  Pi="p", 3*Pi/2="3p/2", 2*Pi="2p"]), ytickmarks=[-1,1],
  axesfont=[SYMBOL,16], labels=["","",""] );
```



The `evalf` function causes values in its argument to be evaluated as floating-point numbers, thus permitting Maple to find them on the axis. The letter p in `SYMBOL` font is the Greek letter π.

2.13 Toward leaner code

The `display` procedure has the same options as `plot` and `plot3d`. If you are storing plots using the same option, and then displaying them together later, you can move that option into the `display` statement. Having specified the option in one place instead of several, you will have streamlined your code. For example, instead of

```
> with( plots ):
> Plane1 := plot3d( 2*x-3*y, x=-1..1, y=-1..1, color=red,
  style=patchnogrid, axes=normal ):
> Plane2 := plot3d( 5*x+2*y, x=-1..1, y=-1..1, color=blue,
  style=patchnogrid, axes=normal ):
> Plane3 := plot3d( -1, x=-1..1, y=-1..1, color=green,
  style=patchnogrid, axes=normal ):
> display( Plane1, Plane2, Plane3 );
```

you could use

```
> with( plots ):
> Plane1 := plot3d( 2*x-3*y, x=-1..1, y=-1..1, color=red ):
> Plane2 := plot3d( 5*x+2*y, x=-1..1, y=-1..1, color=blue ):
> Plane3 := plot3d( -1, x=-1..1, y=-1..1, color=green ):
> display( Plane1, Plane2, Plane3, style=patchnogrid,
  axes=normal );
```

If you have certain options that you know you would like to use for all the plots in `display` statements in a worksheet, you can state them up front, at the beginning of the worksheet, using `setoptions` for two-dimensional plots and `setoptions3d` for three-dimensional ones.[1] These procedures are included in the `plots` package. Returning to the previous example,

```
> with( plots ):
> setoptions3d( style=patchnogrid, axes=normal ):
> Plane1 := plot3d( 2*x-3*y, x=-1..1, y=-1..1, color=red ):
> Plane2 := plot3d( 5*x+2*y, x=-1..1, y=-1..1, color=blue ):
> Plane3 := plot3d( -1, x=-1..1, y=-1..1, color=green ):
> display( Plane1, Plane2, Plane3 );
```

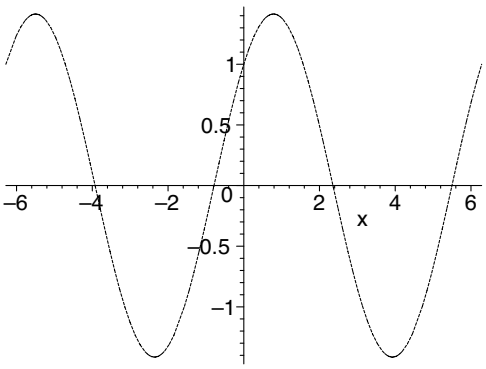
Better, yes?

2.14 Context-sensitive menus and context bars

Several of the options such as `shading`, `style`, and `axes` can be changed after the plot is rendered. Just click on the plot output and use the menus at the top of the worksheet. These are called *context-sensitive* menus because they depend on the type of output. Clicking on a two-dimensional plot and clicking on a three-dimensional plot, for example, will bring forth two different sets of menus. These menus are also available by clicking the right mouse button or, if you have only one button, by option-clicking (holding down the *option* key while clicking). Try clicking in the plot,

```
> plot( sin(x) + cos(x), x=-2*Pi..2*Pi, linestyle=DASH );
```

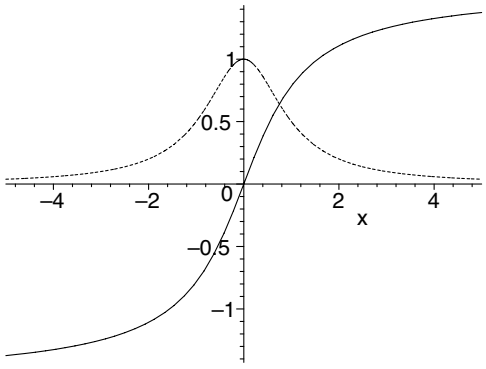
[1] The intention was to have `setoptions` and `setoptions3d` control the options for plots directly output by `plot` and `plot3d` statements, respectively, as well as for plots output by `display` statements. This works for most options, but, due to a bug present in both Maple 7 and 8, not for all. To ensure that all the options specified in `setoptions` or `setoptions3d` are the options actually used in the plot output, just `display` the plot.



and changing line style to `SOLID`. This option is under **Line Style** in the **Style** menu.

In Maple 8, when a plot is shown using `display`, the options that are written directly into the plotting statement (such as `plot` or `plot3d`) are protected from change. The context-sensitive menus, then, cannot alter such "hard-coded" options. For example, in the plot of

```
> with( plots ):
> Curve1 := plot( arctan(x), x=-5..5 ):
> Curve2 := plot( 1/(1+x^2), x=-5..5, linestyle=DOT ):
> display( Curve1, Curve2 );
```



the line style of *Curve1* can be changed, while that of *Curve2* cannot. Options that are written into the `display` statement, instead of the plotting statement, may be changed with the context-sensitive menus.

In the **View** menu, select **Context Bar** if it isn't already checked. This makes various buttons appear at the top of the window whenever you click on a plot. They offer another way to makes changes to the options after the plot is rendered.

2.15 Further details

In this chapter, we have seen most of the options for plotting. The details of these and others are available from Maple's help facility. Just type `?plot,options` at the Maple prompt to learn more about the options for two-dimensional plotting, and type `?plot3d,options` for details about three-dimensional plot options. For full details on the procedures `plot` and `plot3d` themselves, enter `?plot` or `?plot3d` at the Maple prompt. For help on `display`, enter `?plots,display`.

If Maple is new to you, your time would be well spent experimenting some now. Try graphing a few functions from a calculus book. Plot some functions of a single variable and some of two variables. Try graphing a surface and a plane tangent to it at some point. Bounce around Maple's help facility for a while.

Chapter 3

Non-Cartesian Coordinates and Quadric Surfaces

The choice of coordinate system can be the difference between mathematics awkwardly or elegantly expressed. It can also be the difference between surfaces crudely or finely rendered. In this chapter, we will discuss polar, cylindrical, and spherical coordinates, and techniques for making high-quality plots of quadric surfaces. We will also discuss a way to get a quick, although usually lower-quality, plot of a surface.

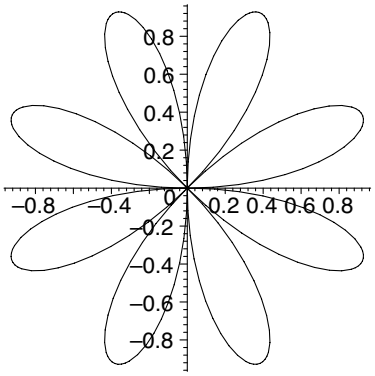
3.1 Polar coordinates

Among the options for plot statements is the `coords` option, which provides a way to plot in coordinate systems other than the default Cartesian (or rectangular) system. When the option `coords=polar` is chosen, Maple will plot points of the form (r, θ), where $|r|$ is the distance from the origin and θ is the angle measured counterclockwise from the positive end of the x-axis. Maple expects to see a function r in terms of θ and a domain for θ. The syntax is

`plot(` $r(\theta)$`, `θ`=`$\alpha..\beta$`, coords=polar,` *other options* `)`

For example,

```
> plot( sin(4*theta), theta=0..2*Pi, coords=polar,
  scaling=constrained );
```



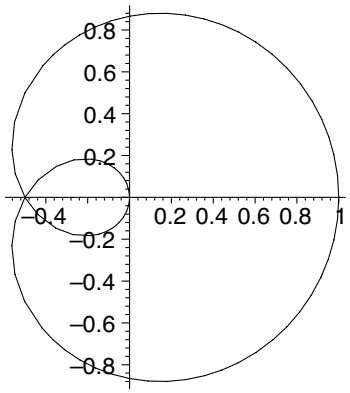
Recall from Section 2.8 that the option `scaling=constrained` prevents Maple from scaling the vertical and horizontal dimensions independently.

For parametric forms, the syntax is a list

`plot( [`$r(t)$`,` $\theta(t)$`,` t`=`a`..`b`], coords=polar,` *other options* `)`

as in

```
> plot( [cos(t), 3*t, t=0..Pi], coords=polar,
  scaling=constrained );
```



Another way to plot in polar coordinates is with the procedure `polarplot`, which is available in the `plots` package. It eliminates the need for the `coords=polar` option. The syntax is

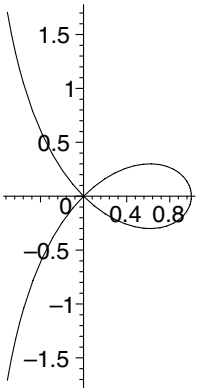
`polarplot(` $r(\theta)$`,` θ`=`α`..`β`,` *options* `)`

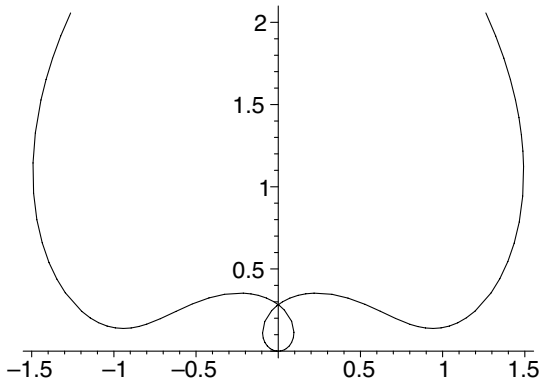
or, for parametric forms,

`polarplot( [`$r(t)$`,` $\theta(t)$`,` t`=`a`..`b`],` *options* `)`

For example,

```
> with( plots ):
> polarplot( cos(2*theta)*sec(theta), theta=-3*Pi/8..3*Pi/8,
  scaling=constrained );
> polarplot( [cot(theta), 3*sin(2*theta),
  theta=Pi/8..7*Pi/8], scaling=constrained );
```





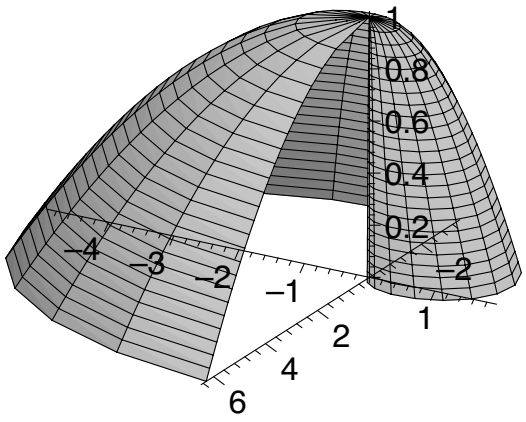
3.2 Cylindrical coordinates

With the option `coords=cylindrical`, Maple plots points in cylindrical coordinates (r, θ, z). These coordinates are particularly useful for generating smooth plots of quadric surfaces. Maple wants r as a function of θ and z, so the syntax is

`plot3d(` $r(\theta, z)$`,` θ=$\alpha..\beta$`,` z=$a..b$`, coords=cylindrical,` *other options* `)`

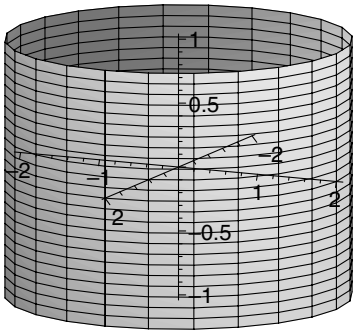
For example,

```
> plot3d( theta*sqrt(1-z), theta=0..2*Pi, z=0..1,
  coords=cylindrical, axes=normal );
```



A cylinder with axis the z-axis is, of course, easily plotted:

```
> plot3d( 2, theta=0..2*Pi, z=-1..1, coords=cylindrical,
  axes=normal );
```

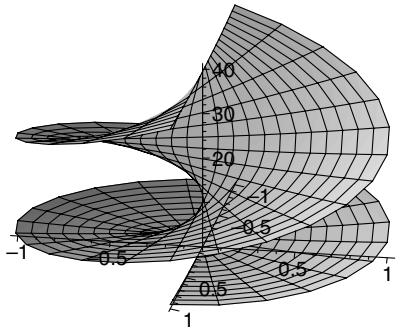


The parametric form is a list

`plot3d( [`$r(s,t)$`, `$\theta(s,t)$`, `$z(s,t)$`], `s`=`$a..b$`, `t`=`$c..d$`, coords=cylindrical,`
other options `)`

For example,

```
> plot3d( [s, t, s^2+t^2], s=-1..1, t=0..2*Pi,
  coords=cylindrical, axes=normal );
```

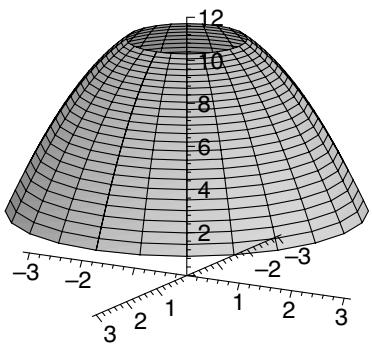


If you prefer to think of z as a function of r and θ (instead of r as a function of θ and z), parametric form provides a handy way to do that. Just use the form

`plot3d( [`r`, `θ`, `$z(r,\theta)$`], `r`=`$a..b$`, `θ`=`$\alpha..\beta$`, coords=cylindrical,`
other options `)`

For example,

```
> plot3d( [r, theta, 12-r^2*(cos(theta)^2+sin(theta)^2)],
  r=1..3, theta=0..2*Pi, coords=cylindrical, view=0..12,
  axes=normal );
```



An alternative method of plotting in cylindrical coordinates is to use the `cylinderplot` procedure from the `plots` package. It makes the `coords=cylindrical` option unnecessary. The syntax is

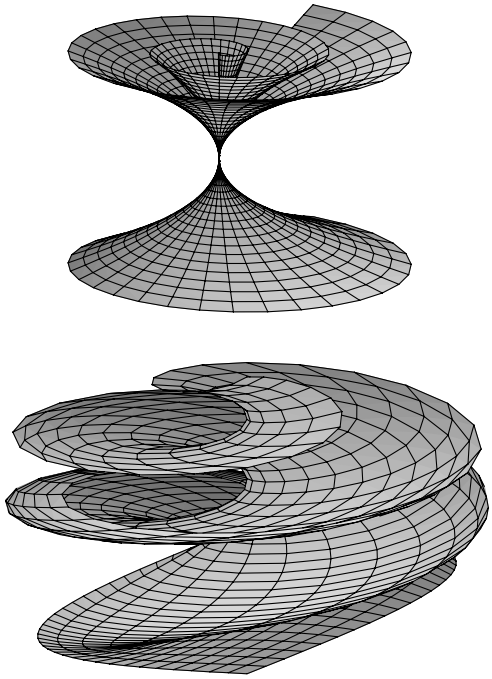
`cylinderplot(` $r(\theta,z)$`,` θ`=`$\alpha..\beta$`,` z`=`$a..b$`,` *options* `)`

or, for parametric forms,

`cylinderplot( [`$r(s,t)$`,` $\theta(s,t)$`,` $z(s,t)$`],` s`=`$a..b$`,` t`=`$c..d$`,` *options* `)`

For example,

```
> with( plots ):
> cylinderplot( z^2*theta, theta=0..5*Pi, z=-1..1,
  grid=[80,40] );
> cylinderplot( [sin(s-t), s*t, s+t], s=0..Pi, t=0..Pi,
  numpoints=1200 );
```



where we have used the `grid` and `numpoints` options (Section 2.6) to create smoother surfaces. In the first `cylinderplot` statement, `grid=[80,40]` causes Maple to use 80 values for θ and 40 for z.

3.3 Spherical coordinates and others

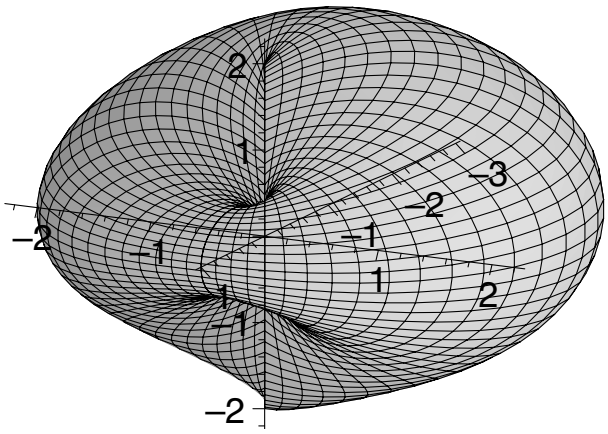
The option `coords=spherical` invokes spherical coordinates (ρ,θ,ϕ) where $|\rho|$ is the distance of the point from the origin (ρ can be negative), θ is the

same angle as in polar coordinates, and ϕ is the angle that a segment from the origin to the point makes with the positive end of the z-axis. Maple expects ρ to be expressed as a function of θ and ϕ, so the syntax is

`plot3d(` $\rho(\theta,\phi)$`,` θ`=`α`..`$\beta`, `$\phi$`=`$\gamma$`..`$\delta`, coords=spherical,` *other options* `)`

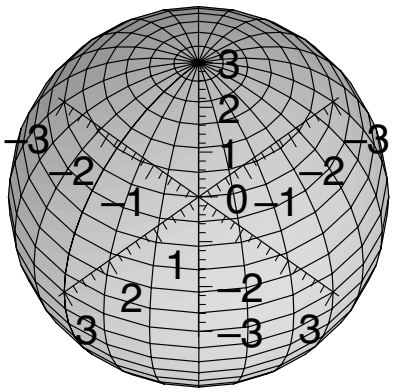
as in

```
> plot3d( 1+sin(phi)-cos(theta), theta=0..2*Pi, phi=0..Pi,
  coords=spherical, grid=[50,40], axes=normal );
```



A sphere centered at the origin can, of course, be plotted very simply in spherical coordinates:

```
> plot3d( 3, theta=0..2*Pi, phi=0..Pi, coords=spherical,
  scaling=constrained, axes=normal );
```

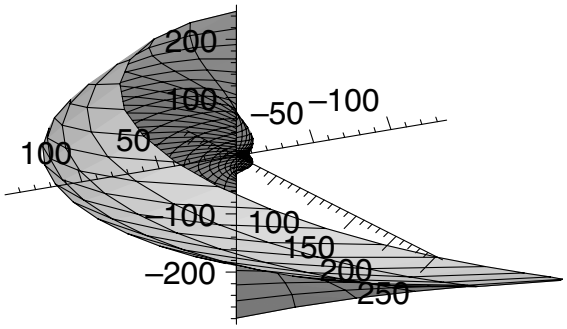


The parametric form is a list

`plot3d( [`$\rho(s,t)$`,` $\theta(s,t)$`,` $\phi(s,t)$`], `s`=`a`..`b`, `t`=`c`..`d`, coords=spherical,`
other options `)`

such as

```
> plot3d( [s^3+t^3, s+t, t], s=0..2*Pi, t=0..Pi,
  coords=spherical, axes=normal );
```



The alternative method of plotting in spherical coordinates is `sphereplot` within the `plots` package:

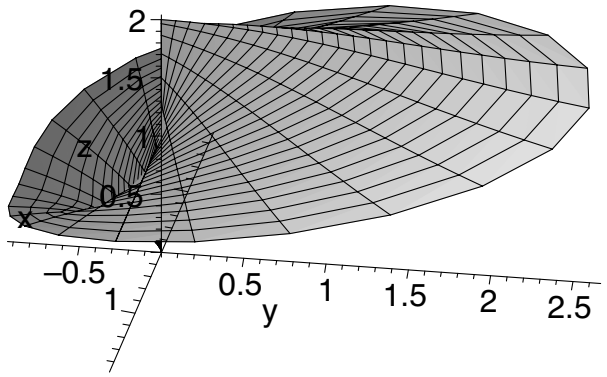
`sphereplot(` $\rho(\theta,\phi)$`,` θ=$\alpha..\beta$`,` ϕ=$\gamma..\delta$`,` *options* `)`

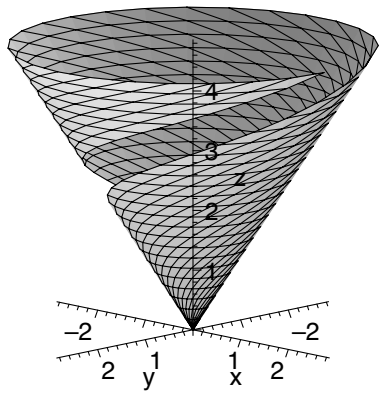
or

`sphereplot( [`$\rho(s,t)$`,` $\theta(s,t)$`,` $\phi(s,t)$`],` s=$a..b$`,` t=$c..d$`,` *options* `)`

For example,

```
> with( plots ):
> sphereplot( sin(theta)+sec(phi), theta=0..2*Pi,
  phi=0..Pi/3, axes=normal );
> sphereplot( [s, Pi+s+t, Pi/6], s=0..7*Pi/4, t=0..7*Pi/4,
  shading=xy, axes=normal );
```





Maple supports an impressive variety of coordinate systems. To see the list, enter `?coords` at the Maple prompt. It is likely that you will find what you are looking for there. If not, you can create your own coordinate system. For information on how to do that, enter `?addcoords` at the prompt.

3.4 Quadrics quickly

We will consider six types of quadric surfaces: elliptic and hyperbolic paraboloids, elliptic cones, ellipsoids (including spheres), and hyperboloids of one and of two sheets. Maple's ability to plot a surface in three dimensions is a significant advantage in helping students to understand these analogues of conic sections, and the capability to rotate the plot in real time is especially helpful.

We can use Maple's `implicitplot3d` procedure, which is in the `plots` package, to get graphs of quadric surfaces without much investment of time or thought. This procedure accepts an equation (or an expression), samples a default $10 \times 10 \times 10$ grid of points, then uses a numerical algorithm based upon triangulation into tetrahedra to interpolate a surface. The options for `implicitplot3d` are the same as those for `plot3d`, except that `gridstyle`, which can be either `rectangular` or `triangular` in a `plot3d` plot, is fixed as `triangular` by `implicitplot3d`.

Let's plot the following quadric surfaces:

$$z = \frac{x^2}{4} + \frac{y^2}{9} \qquad \text{an elliptic paraboloid}$$

$$z = \frac{x^2}{4} - \frac{y^2}{9} \qquad \text{a hyperbolic paraboloid}$$

$$\frac{x^2}{4} + \frac{y^2}{9} - z^2 = 0 \qquad \text{an elliptic cone}$$

$$\frac{x^2}{4} + \frac{y^2}{9} + z^2 = 1 \qquad \text{an ellipsoid}$$

$$\frac{x^2}{4} + \frac{y^2}{9} - z^2 = 1 \qquad \text{a hyperboloid of one sheet}$$

$$-\frac{x^2}{4} - \frac{y^2}{9} + z^2 = 1 \qquad \text{a hyperboloid of two sheets}$$

Although it is not necessary to name and store each surface's defining equation before plotting, it is useful to do that and let Maple echo the input so that we can check that we have entered it correctly:

```
> Elliptic_paraboloid := z = (x^2)/4 + (y^2)/9;
> Hyperbolic_paraboloid := z = (x^2)/4 - (y^2)/9;
```

```
> Elliptic_cone := (x^2)/4 + (y^2)/9 - z^2 = 0;
> Ellipsoid := (x^2)/4 + (y^2)/9 + z^2 = 1;
> Hyperboloid_of_1_sheet := (x^2)/4 + (y^2)/9 - z^2 = 1;
> Hyperboloid_of_2_sheets := -(x^2)/4 - (y^2)/9 + z^2 = 1;
```

$$Elliptic\_paraboloid := z = \frac{x^2}{4} + \frac{y^2}{9}$$

$$Hyperbolic\_paraboloid := z = \frac{x^2}{4} - \frac{y^2}{9}$$

$$Elliptic\_cone := \frac{x^2}{4} + \frac{y^2}{9} - z^2 = 0$$

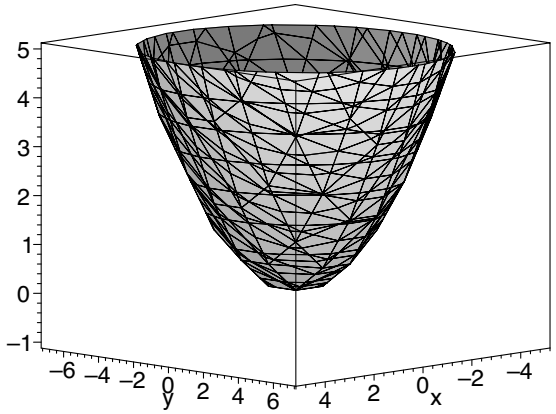
$$Ellipsoid := \frac{x^2}{4} + \frac{y^2}{9} + z^2 = 1$$

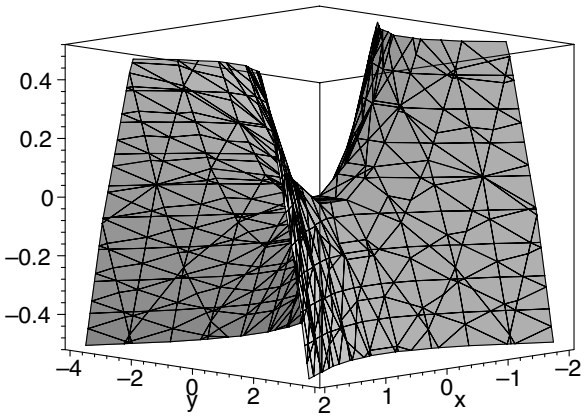
$$Hyperboloid\_of\_1\_sheet := \frac{x^2}{4} + \frac{y^2}{9} - z^2 = 1$$

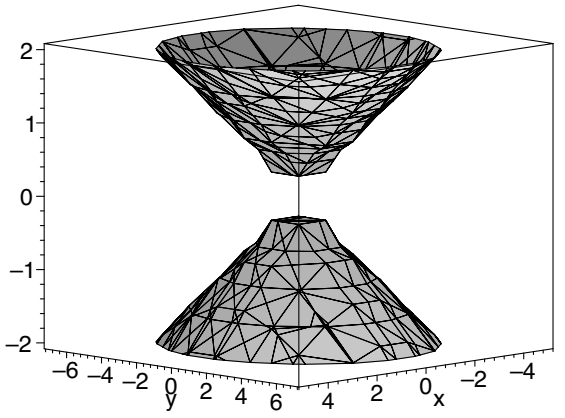
$$Hyperboloid\_of\_2\_sheets := -\frac{x^2}{4} - \frac{y^2}{9} + z^2 = 1$$

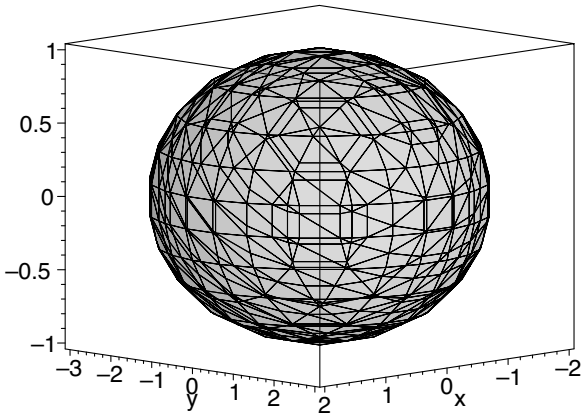
We can then plot these using `implicitplot3d`:

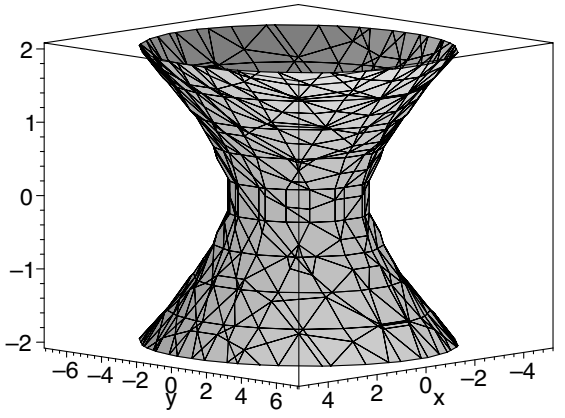
```
> with(plots):
> implicitplot3d( Elliptic_paraboloid, x=-5..5, y=-7..7,
  z=-1..5, axes=boxed );
> implicitplot3d( Hyperbolic_paraboloid, x=-2..2, y=-4..4,
  z=-1/2..1/2, axes=boxed );
> implicitplot3d( Elliptic_cone, x=-5..5, y=-7..7, z=-2..2,
  axes=boxed );
> implicitplot3d( Ellipsoid, x=-2..2, y=-3..3, z=-1..1,
  axes=boxed );
> implicitplot3d( Hyperboloid_of_1_sheet, x=-5..5, y=-7..7,
  z=-2..2, axes=boxed );
> implicitplot3d( Hyperboloid_of_2_sheets, x=-4..4, y=-6..6,
  z=-2..2, axes=boxed );
```

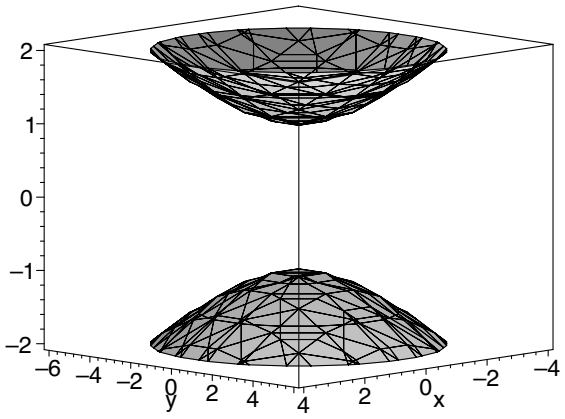



0.4
0.2
0
−0.2
−0.4
−4
−2
0
2
y
2
1
0
x
−1
−2


2
1
0
−1
−2
−6
−4
−2
0
2
4
6
y
4
2
0
x
−2
−4

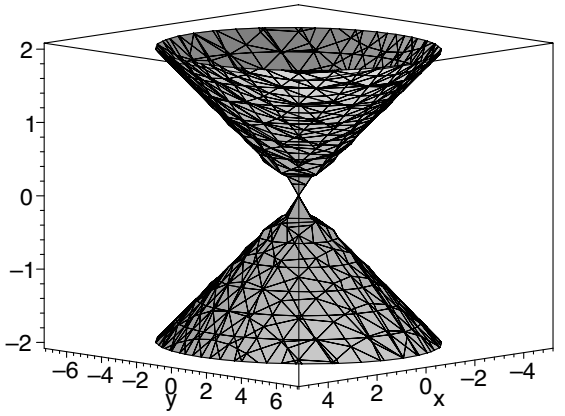

1
0.5
0
−0.5
−1
−3
−2
−1
0
1
2
y
2
1
0
x
−1
−2


2
1
0
−1
−2
−6
−4
−2
0
2
4
6
y
4
2
0
x
−2
−4



These aren't bad, except for the elliptic cone, which we can improve by using the `grid` option:

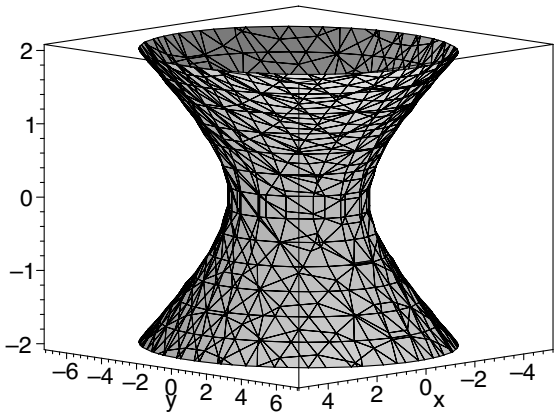
```
> with( plots ):
> implicitplot3d( Elliptic_cone, x=-5..5, y=-7..7, z=-2..2,
  grid=[15,15,15], axes=boxed );
```



Here, Maple uses 15 equally-spaced points from each of the intervals that we specified: $[-5, 5]$, $[-7, 7]$, and $[-2, 2]$. The choice of an odd number of points, together with the symmetry of these intervals about the origin, means that Maple will sample the point $(0, 0, 0)$, which is the vertex of the cone. Since the vertex was not one of the points sampled when we used the default grid $[10, 10, 10]$, our first plot didn't include it.

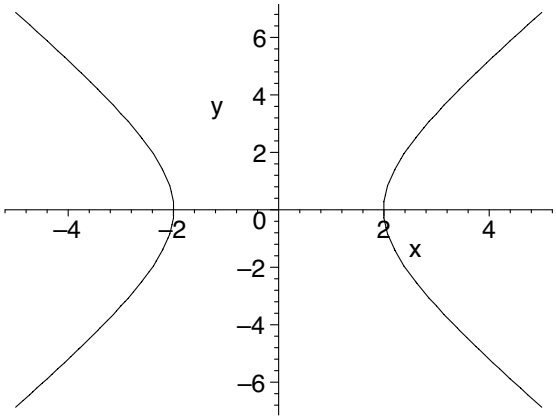
The smoothness of any of these plots can be improved by ordering Maple to use a finer mesh than default. This can be done either by using the `grid` option as above or by setting the `numpoints` option to a number greater than 1000. For example,

```
> with( plots ):
> implicitplot3d( Hyperboloid_of_1_sheet, x=-5..5, y=-7..7,
  z=-2..2, numpoints=1500, axes=boxed );
```



Incidentally, there is also an `implicitplot` procedure for plotting curves represented by equations in two variables. For example, the hyperbola $x^2/4 - y^2/9 = 1$ can be plotted with

```
> with( plots ):
> implicitplot( x^2/4 - y^2/9 = 1, x=-5..5, y=-7..7 );
```

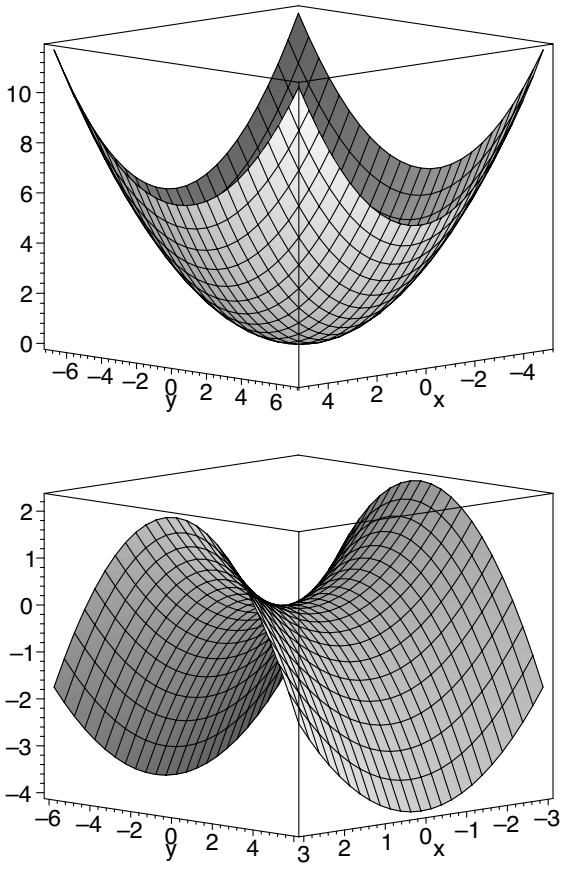


The plots generated by `implicitplot3d` provide at least a rough representation of surfaces. Students can use this method to plot these surfaces without investing a great deal of time in learning the intricacies of Maple. We will want higher-quality plots, however, when we begin writing animations that depend on them. Let's investigate how we might do that.

3.5 Paraboloids

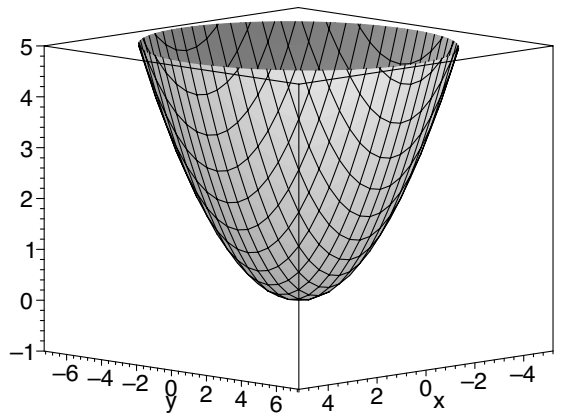
Since elliptic and hyperbolic paraboloids (that have the form of the above examples) are functions of x and y, we can use the `plot3d` procedure to plot them. Redoing our examples of the elliptic paraboloid $z = x^2/4 + y^2/9$ and the hyperbolic paraboloid $z = x^2/4 - y^2/9$ in this way, we have

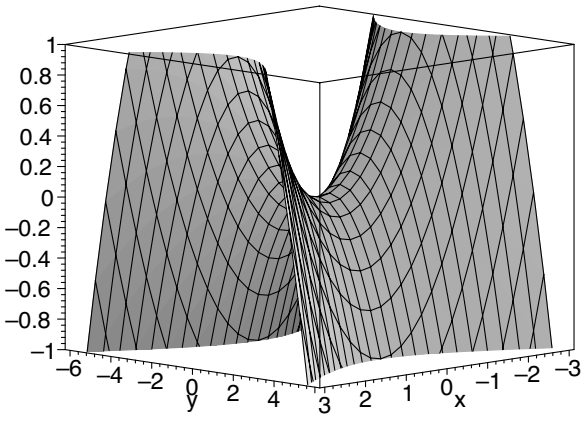
```
> plot3d( x^2/4 + y^2/9, x=-5..5, y=-7..7, axes=boxed );
> plot3d( x^2/4 - y^2/9, x=-3..3, y=-6..6, axes=boxed );
```



Although accurate, these plots don't have the appearance of the plots usually presented in textbooks. The reason is that we are seeing too much z. We can remedy that by using the `view` option:

```
> plot3d( x^2/4 + y^2/9, x=-5..5, y=-7..7, view=-1..5,
  axes=boxed );
> plot3d( x^2/4 - y^2/9, x=-3..3, y=-6..6, view=-1..1,
  axes=boxed );
```





3.6 Elliptic cones

To get a high-quality plot of an elliptic cone, we will use cylindrical coordinates. To plot the elliptic cone

$$\frac{x^2}{4} + \frac{y^2}{9} - z^2 = 0$$

we convert from Cartesian to cylindrical coordinates

$$\frac{r^2 \cos^2\theta}{4} + \frac{r^2 \sin^2\theta}{9} - z^2 = 0$$

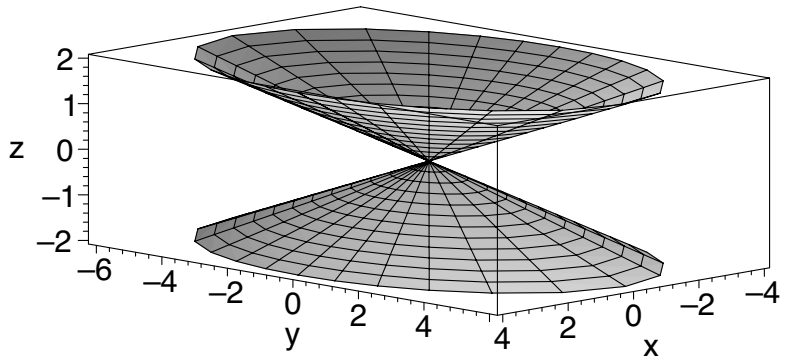
Then, since Maple requires r as a function of θ and z, we solve for r

$$r = \frac{6z}{\sqrt{9\cos^2\theta + 4\sin^2\theta}}$$

and plot using `cylinderplot` from the `plots` package:

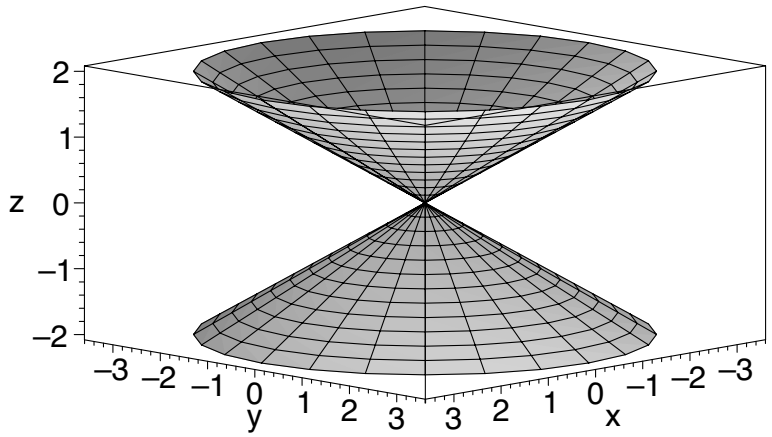
```
> with( plots ):
> r := (theta,z) -> 6*z/sqrt(9*cos(theta)^2+4*sin(theta)^2);
> cylinderplot( r(theta,z), theta=0..2*Pi, z=-2..2,
  scaling=constrained, axes=boxed );
```

$$r := (\theta,\, z) \to \frac{6\,z}{\sqrt{9\cos(\theta)^2 + 4\sin(\theta)^2}}$$



A circular cone with axis the z-axis has a simple form in spherical coordinates: $\phi = c$, where c is a constant in $(0, \pi)$. As ρ is not a function of θ and ϕ in this case, we use parametric form. For example,

```
> with( plots ):
> sphereplot( [rho, theta, Pi/3], rho=-4..4, theta=0..2*Pi,
  scaling=constrained, axes=boxed );
```



3.7 Ellipsoids

We will use cylindrical coordinates for ellipsoids, too. Converting the ellipsoid

$$\frac{x^2}{4} + \frac{y^2}{9} + z^2 = 1$$

to cylindrical coordinates,

$$\frac{r^2 \cos^2 \theta}{4} + \frac{r^2 \sin^2 \theta}{9} + z^2 = 1$$

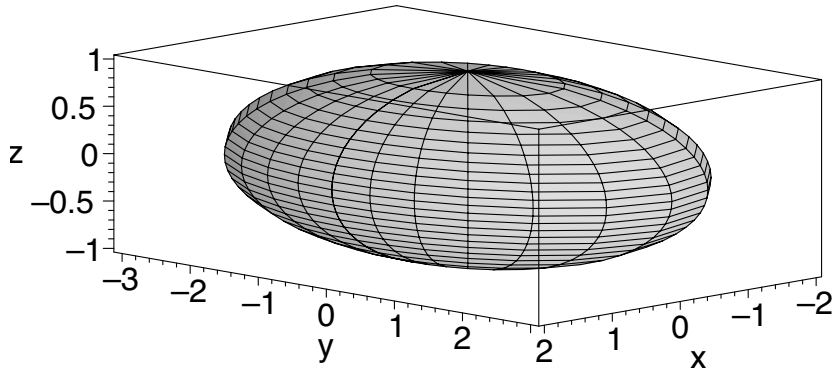
then solving for r, we have

$$r = 6\sqrt{\frac{1-z^2}{9\cos^2\theta + 4\sin^2\theta}}$$

We plot this using `cylinderplot`:

```
> with( plots ):
> r := (theta,z) ->
  6*sqrt((1-z^2)/(9*cos(theta)^2+4*sin(theta)^2));
> cylinderplot( r(theta,z), theta=0..2*Pi, z=-1..1,
  scaling=constrained, axes=boxed );
```

$$r := (\theta,\, z) \to 6\sqrt{\frac{1-z^2}{9\cos(\theta)^2 + 4\sin(\theta)^2}}$$



The spherical coordinate system would seem a natural choice for an ellipsoid; let's try it. Converting

$$\frac{x^2}{4} + \frac{y^2}{9} + z^2 = 1$$

to spherical coordinates

$$\frac{\rho^2\sin^2\phi\cos^2\theta}{4} + \frac{\rho^2\sin^2\phi\sin^2\theta}{9} + \rho^2\cos^2\phi = 1$$

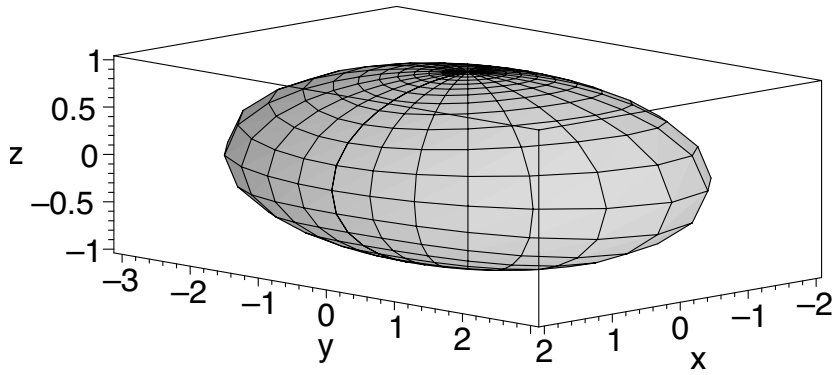
and solving for ρ, we have

$$\rho = \frac{6}{\sqrt{\sin^2\phi(9\cos^2\theta + 4\sin^2\theta) + 36\cos^2\phi}}$$

We then plot this using `sphereplot`:

```
> with( plots ):
> rho := (theta,phi) -> 6/sqrt(sin(phi)^2*(9*cos(theta)^2+
  4*sin(theta)^2)+36*cos(phi)^2);
> sphereplot( rho(theta,phi), theta=0..2*Pi, phi=0..Pi,
  scaling=constrained, axes=boxed );
```

$$\rho := (\theta,\, \phi) \to \frac{6}{\sqrt{\sin(\phi)^2\,(9\cos(\theta)^2 + 4\sin(\theta)^2) + 36\cos(\phi)^2}}$$



There is no improvement over the already good plot from `cylinderplot`. Since the function ρ is marginally more complex than r, we will opt for cylindrical coordinates.

3.8 Hyperboloids

Converting the hyperboloid of one sheet

$$\frac{x^2}{4} + \frac{y^2}{9} - z^2 = 1$$

to cylindrical coordinates

$$\frac{r^2\cos^2\theta}{4} + \frac{r^2\sin^2\theta}{9} - z^2 = 1$$

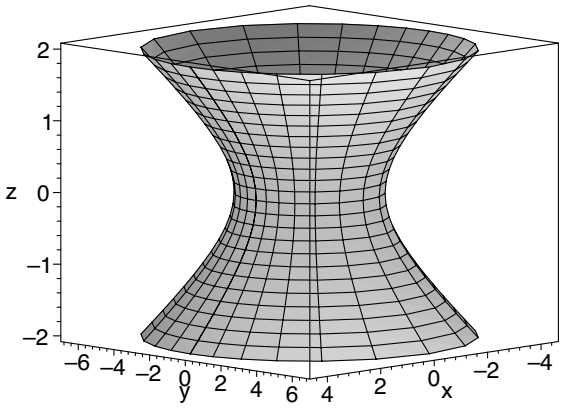
and solving for r, we have

$$r = 6\sqrt{\frac{z^2+1}{9\cos^2\theta + 4\sin^2\theta}}$$

We then plot using `cylinderplot`:

```
> with( plots ):
> r := (theta,z) ->
  6*sqrt((z^2+1)/(9*(cos(theta))^2+4*(sin(theta))^2));
> cylinderplot( r(theta,z), theta=0..2*Pi, z=-2..2,
  axes=boxed );
```

$$r := (\theta,\, z) \to 6\sqrt{\frac{z^2+1}{9\cos(\theta)^2 + 4\sin(\theta)^2}}$$



Doing the same for the hyperboloid of two sheets

$$-\frac{x^2}{4}-\frac{y^2}{9}+z^2=1$$

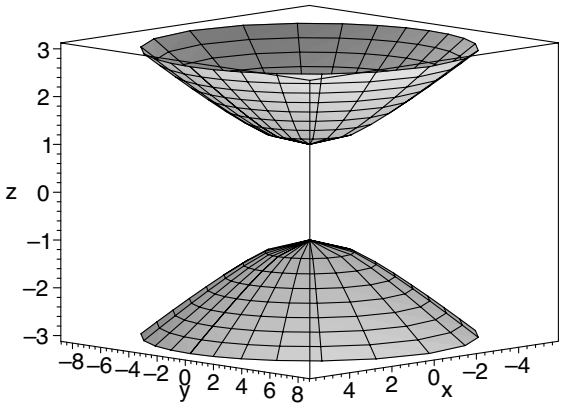
yields

$$r=6\sqrt{\frac{z^2-1}{9\cos^2\theta+4\sin^2\theta}}$$

and

```
> with( plots ):
> r := (theta,z) ->
  6*sqrt((z^2-1)/(9*cos(theta)^2+4*sin(theta)^2));
> cylinderplot( r(theta,z), theta=0..2*Pi, z=-3..3,
  axes=boxed );
```

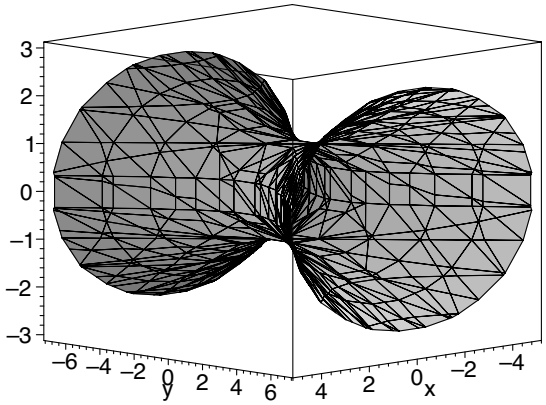
$$r:=(\theta,\,z)\rightarrow 6\sqrt{\frac{z^2-1}{9\cos(\theta)^2+4\sin(\theta)^2}}$$

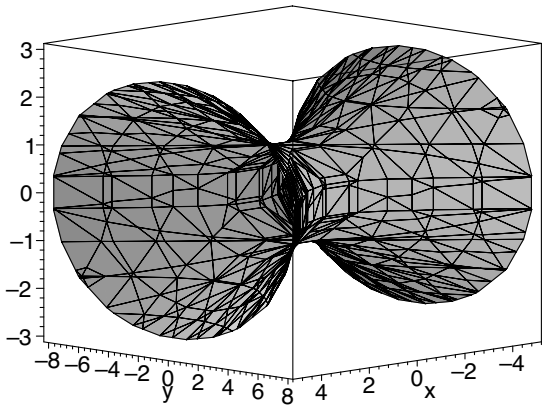


3.9 Quadric surfaces with axes other than the z-axis

For simplicity, our examples have had as their axes the z-axis. For demonstrating the salient features of quadric surfaces, these are fine. You may, however, want to plot quadric surfaces with axis the x-axis or the y-axis. A simple way to do that is to use `implicitplot3d` as above. For example,

```
> with( plots ):
> implicitplot3d( (x^2)/4-(y^2)/9+z^2=1, x=-5..5, y=-7..7,
  z=-3..3, axes=boxed );
> implicitplot3d( -(x^2)/4+(y^2)/9+z^2=1, x=-5..5, y=-8..8,
  z=-3..3, axes=boxed );
```



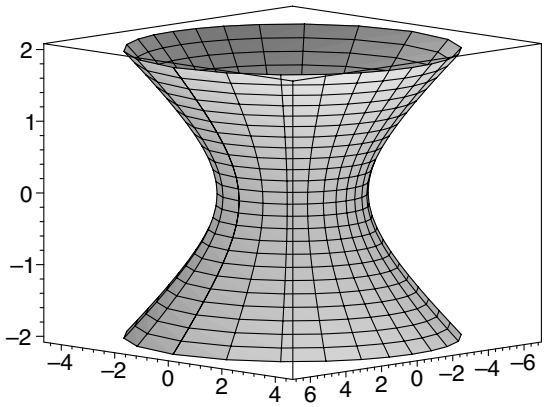


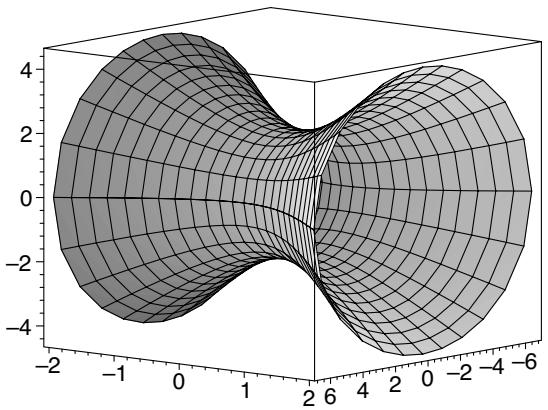
A second way—and one that produces higher-quality results—is to create a general function r in terms of θ and t, where t will be replaced by x, y, or z, then use Cartesian coordinates in parametric form. For example,

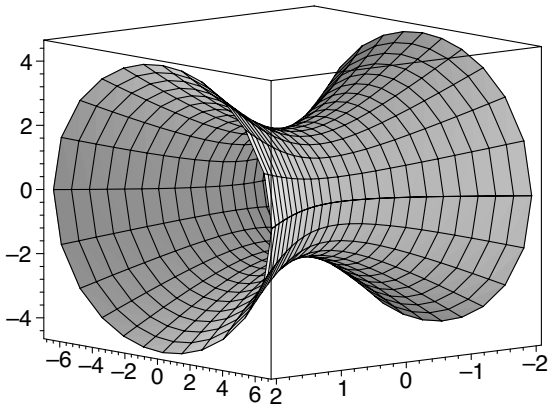
```
> with( plots ):
> r := (theta,t) ->
  6*sqrt((t^2+1)/(4*(cos(theta))^2+9*(sin(theta))^2));
> plot3d( [r(theta,z)*cos(theta), r(theta,z)*sin(theta), z],
  theta=0..2*Pi, z=-2..2, axes=boxed );
```

```
> plot3d( [r(theta,y)*cos(theta), y, r(theta,y)*sin(theta)],
  theta=0..2*Pi, y=-2..2, axes=boxed );
> plot3d( [x, r(theta,x)*cos(theta), r(theta,x)*sin(theta)],
  theta=0..2*Pi, x=-2..2, axes=boxed );
```

$$r := (\theta,\, t) \to 6\sqrt{\frac{t^2+1}{4\cos(\theta)^2+9\sin(\theta)^2}}$$

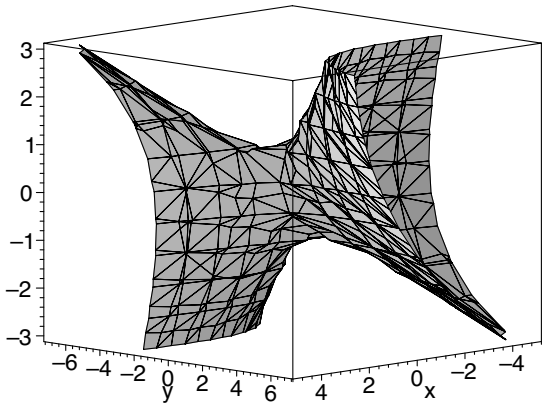






For a quadric surface whose axis is oblique, the easiest way is just to use `implicitplot3d`

```
> with( plots ):
> implicitplot3d( (x^2)/4-(y^2)/9+x*y-y*z+z^2=1, x=-5..5,
  y=-7..7, z=-3..3, axes=boxed );
```



but the plot does not hold much appeal. Another way is to rotate one of the superior plots that we produced using `cylinderplot` above. Still another is to use a linear transformation. In Chapter 10, we will consider two procedures, `rotate` and `transform`, that do that, as well as some other procedures in the `plottools` package.

Chapter 4

Simple Animations

We will make our first animations in this chapter, using three built-in procedures. These convenient procedures are useful when the object you want to animate is a function of one or two variables. We will adopt a general outline for animation worksheets and use it to create many animated classroom demonstrations. One animation illustrates the idea of a tangent line to a curve at a point by showing various secant lines passing through a common point and approaching, or not approaching, a limiting line. Another one demonstrates the squeeze theorem. It draws a curve as it is being forced toward a limit by two bounding functions. Four more are useful for investigating the shapes of hyperboloids and paraboloids by passing a plane through them and studying the cross-sections. Another demonstrates the concepts of level curves and contour plots.

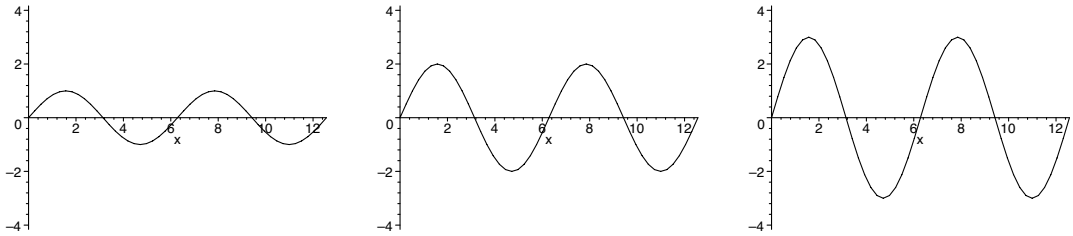
4.1 Animating a function of a single variable

The procedure `animate`, within the `plots` package, provides a quick way to produce an animation of a function of a single variable. This procedure creates frames, 16 by default, and displays them in sequence. The first argument of `animate` is a function, possibly in parametric form, of two variables. One of these, say x, is the independent variable of the function you are animating; the other, say t, is the frame variable. It is the varying values of the frame variable that make the individual frames of the animation differ from each other. The syntax is

`animate(` $F(x,t)$`,` x`=`a`..`b`,` t`=`p`..`q`,` options `)`

For example, we can plot a sine function on the interval $[0, 4\pi]$ with amplitude increasing from 1 to 4 with the command

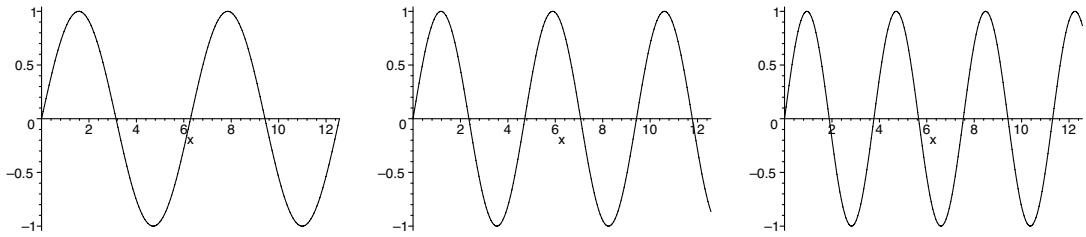
```
> with( plots ):
> animate( t*sin(x), x=0..4*Pi, t=1..4 );
```



In the **View** menu, select **Context Bar** if it isn't already checked. If you click on the plot, various buttons will appear in the context bar (Section 2.14) that allow you to: stop or play the animation, step through one frame at a time, play backward or forward, play back slower or faster, and run for just one cycle or loop continuously. Try it. These options are also available under the **Animation** menu and by clicking the right mouse button (or option-clicking if you have only one button) on the plot. If your computer is a quick one, you may find that the animations play so rapidly that you'll want to slow them down a little.

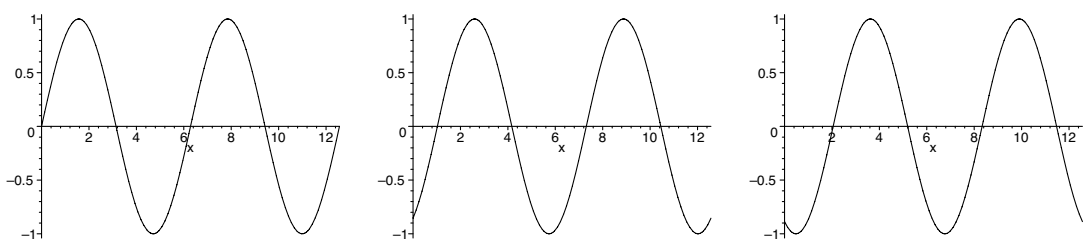
All the options for the `plot` procedure are available in `animate`. For example,

```
> with( plots ):
> animate( sin(t*x), x=0..4*Pi, t=1..2, numpoints=200,
  color=blue );
```



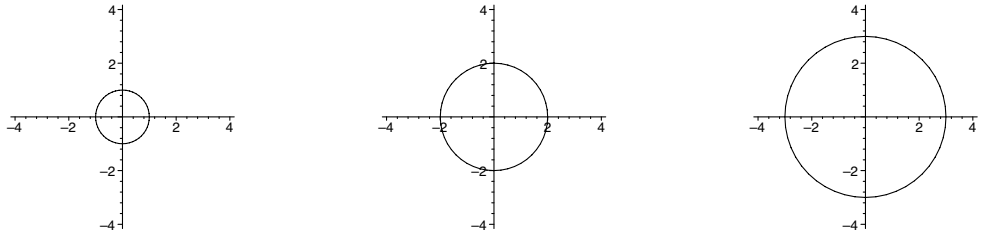
produces smoother plots by way of the `numpoints` option. An additional option, `frames`, allows you to smooth out the animation itself by ordering more than the default 16 frames. For example,

```
> with( plots ):
> animate( sin(x-t), x=0..4*Pi, t=0..2*Pi, frames=40,
  numpoints=100 );
```



The function to be animated may be in parametric form. For example,

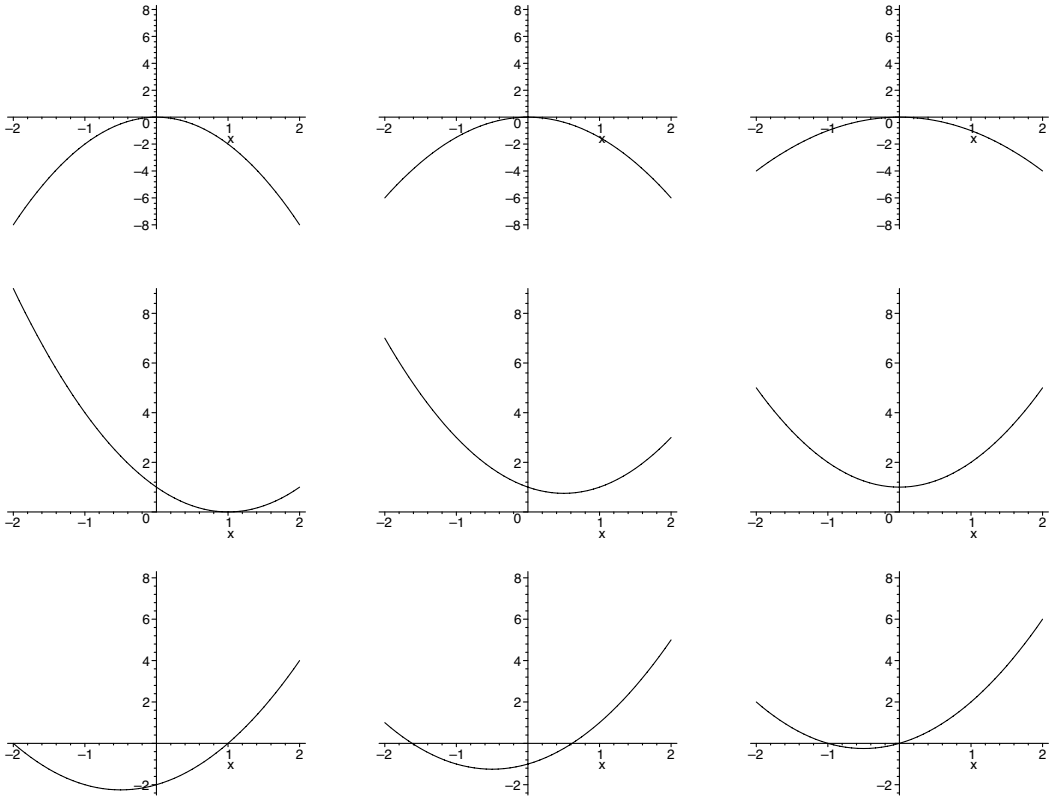
```
> with( plots ):
> animate( [r*cos(theta), r*sin(theta), theta=0..2*Pi],
  r=1..4, scaling=constrained );
```



Recall that the domain for the parameter in two-dimensional parametric plots is given within the list, so theta is the parameter in this plot, and r is the frame variable.

Here are some animations that might be used in a pre-calculus class to investigate the behavior of the graph of $f(x) = ax^2 + bx + c$ as a, b, or c varies.

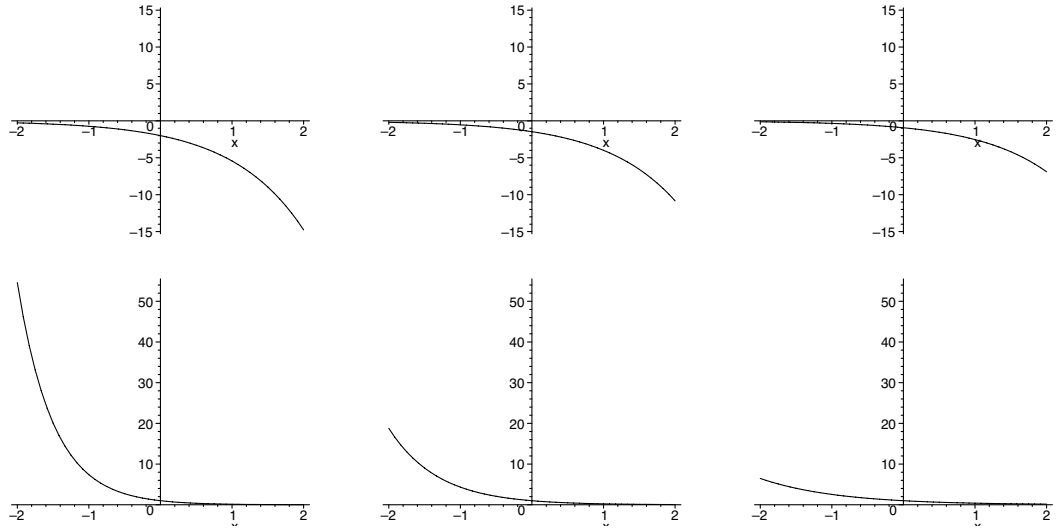
```
> with( plots ):
> animate( a*x^2, x=-2..2, a=-2..2 );
> animate( x^2+b*x+1, x=-2..2, b=-2..2 );
> animate( x^2+x+c, x=-2..2, c=-2..2 );
```



The behavior as b varies will probably not be obvious to the students. It would make a good exercise for them to try this and explain why it happens.

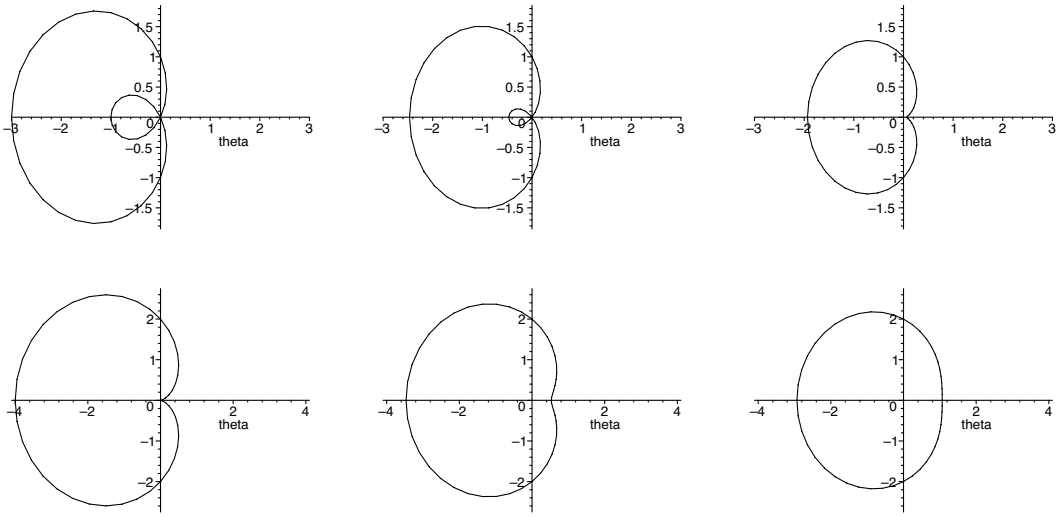
Here are animations useful for studying exponential functions,

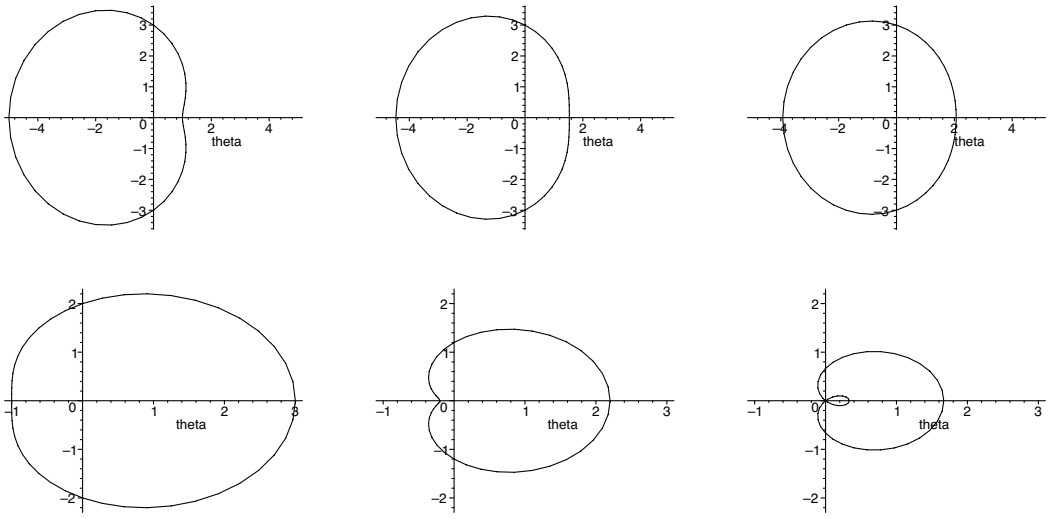
```
> with( plots ):
> animate( A*exp(x), x=-2..2, A=-2..2 );
> animate( exp(B*x), x=-2..2, B=-2..2 );
```



and a few for studying polar functions.

```
> with( plots ):
> animate( 1+a*cos(theta), theta=0..2*Pi, a=-2..2,
  coords=polar );
> animate( 2+a*cos(theta), theta=0..2*Pi, a=-2..2,
  coords=polar );
> animate( 3+a*cos(theta), theta=0..2*Pi, a=-2..2,
  coords=polar );
> animate( a+cos(theta), theta=0..2*Pi, a=-2..2,
  coords=polar );
```

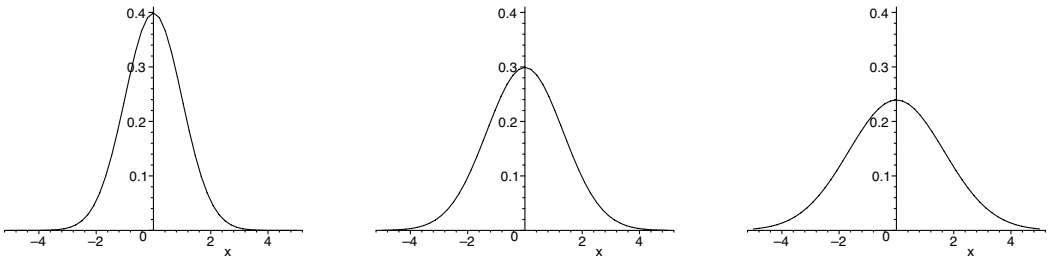




The following animation would be useful in a statistics course for demonstrating what happens to a distribution as the standard deviation increases.

```
> with( plots ):
> G := (x,sigma) ->
  1/(sigma*sqrt(2*Pi))*exp(-x^2/(2*sigma^2));
> animate( G(x,sigma), x=-5..5, sigma=1..2 );
```

$$G := (x,\, \sigma) \rightarrow \frac{\mathbf{e}^{(-1/2\,\frac{x^2}{\sigma^2})}}{\sigma\,\sqrt{2\,\pi}}$$



4.2 Outline of an animation worksheet

The demonstration worksheets in this book have the following general form. It is one way to organize things.

```
> restart:
> with( packages such as plots ):
> setoptions( options ) or setoptions3d( options ):
> BackgroundPlot := optional plot structure that is not animated:
```

```
> Element_1 := ... :
> Element_2 := ... :
      ⋮
> Element_k := ... :
> display( BackgroundPlot, Element_1, Element_2,..., Element_k,
  options );
```

Animated elements created using built-in procedures, such as `animate`, or loop structures, which are discussed in later chapters.

The `restart` command clears variable names and Maple's own internal memory. Whenever Maple begins doing something untoward, it often helps to issue a `restart` command and then re-execute the statements in the worksheet that you need. It could be that you have previously assigned a constant value to a name that you are now trying to use as a variable. Maple won't like that. A `restart` will clear all the names. To restore variable status to just one name, say x, type `x := 'x';` at the Maple prompt. It is good general practice to begin a worksheet with a `restart`. Hereafter, we will begin each section of code on the CD with one.

What `restart` won't do is return memory to the operating system. If memory limits are the problem, you'll need to quit Maple altogether, then restart. Whenever Maple is doing something that you can't account for, first remove the output from the worksheet (under the **Edit** menu), save the worksheet, exit, and then restart Maple. (There's nothing like a new beginning.) If you find yourself having to do this often, it will probably help to increase the memory allocated to Maple.

Generally, worksheets will have a `with` statement near the beginning to summon packages such as `plots`, which contains `animate` and `display` among other things. Recall (Section 2.13) that `setoptions` and `setoptions3d` provide a way to state, up front, any plotting options that you would like to be in effect for all the displays of plots in the worksheet. This saves entering them in every `display` statement. A background plot is a structure—perhaps a curve or a surface—that doesn't move in the animation. The animated elements, possibly many, are created and named, then everything is displayed in one or more `display` statements that can contain options local to them. Next is an example.

4.3 Demonstrations: Secant lines and tangent lines

Let's create animations to demonstrate the idea that a tangent line to a curve at a point is the limiting line of secant lines through the point. To do a thorough job, we will want to show what happens when a curve does have a tangent line at a point and also when it doesn't. We will make one animation that shows the secants from both sides nearing the same line, the tangent line,

and another that shows a case in which the limiting lines from the left and right are different.

4.3.1 Secant lines at a point approaching a tangent line

We begin with

```
> restart:
> with( plots ):
> setoptions( thickness=2, axes=boxed, labels=["","""] ):
```

Choosing a thickness greater than the default 0 improves readability of the plots when they are projected onto a screen.

Next, we define a function to use as an example and choose a point a at which to construct the secant lines. We also store plots of the function and point.

```
> f := x -> x^3 + 8;
> a := 2:
> Curve := plot( f(x), x=a-2..a+2, color=black ):
> FixedPt := pointplot( [a,f(a)], symbol=circle,
  symbolsize=14, color=red ):
```

$$f := x \rightarrow x^3 + 8$$

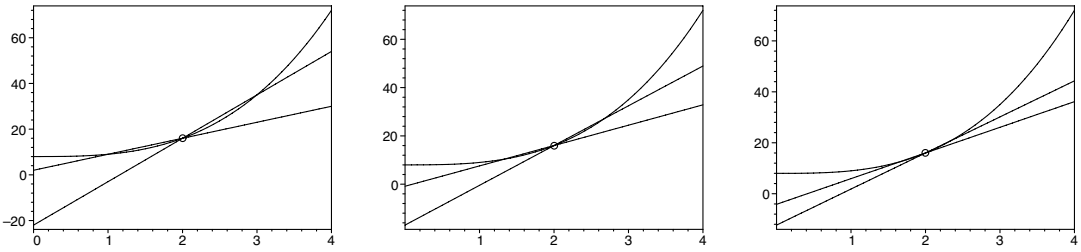
These plots constitute the background plot. The `symbol=circle` option might not be necessary; recall from Section 2.3 that the default plotting symbol for points is device-specific. The choice of a symbol size greater than the default 10 makes the point show up better on a projected image.

We now set up a little function to use as h, the value we would like to approach zero. This is necessary because Maple wants the domain of `animate`'s frame variable to be of the form $a..b$, where $a < b$. It wouldn't do, then, to give the frame variable's domain as, say, `t=1..0.01`. We also create the animated elements: a set of secant lines on the right and another set on the left.

```
> h := t -> 1 - t:
> RightSecants := animate( (f(a+h(t))-f(a))/h(t)*(x-a) +
  f(a), x=a-2..a+2, t=0..0.99, color=blue ):
> LeftSecants := animate( (f(a-h(t))-f(a))/(-h(t))*(x-a) +
  f(a), x=a-2..a+2, t=0..0.99, color=blue ):
```

Finally, we display the results so that we can compare them and decide whether the limiting line is the same from both sides.

```
> display( Curve, FixedPt, RightSecants, LeftSecants );
```

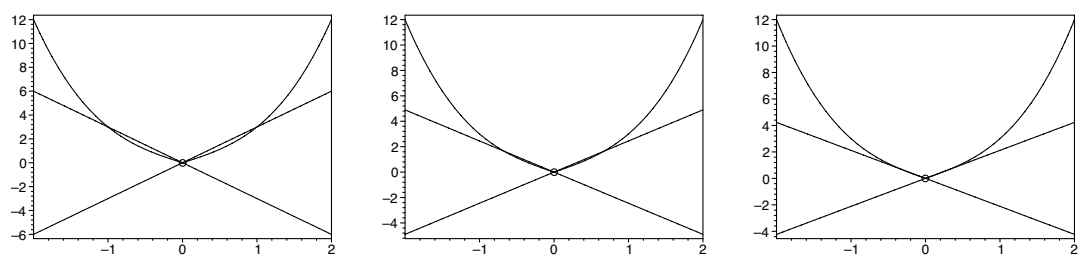


4.3.2 Secant lines at a corner point

We can also use this animation to demonstrate to students that there are continuous functions with points at which no tangent line is defined at all, because the secant lines on the left and right do not approach the same limiting line. One such example is $f(x) = |x^3 + 2x|$ at 0. To demonstrate this, we would need to change only the function f, but we will make another small change. Because the domain `a-2..a+2` appears several times in the code, it would be convenient to give it a name. This would make it easy to change the domain without having to find every place in the code that the domain appears.

```
> restart:
> with( plots ):
> setoptions( thickness=2, axes=boxed, labels=["","",""] ):
> f := x -> abs( x^3 + 2*x );
> a := 0:
> Domain := a-2..a+2:
> Curve := plot( f(x), x=Domain, color=black ):
> FixedPt := pointplot( [a,f(a)], symbol=circle,
  symbolsize=14, color=red ):
> h := t -> 1 - t:
> RightSecants := animate( (f(a+h(t))-f(a))/h(t)*(x-a) +
  f(a), x=Domain, t=0..0.99, color=blue ):
> LeftSecants := animate( (f(a-h(t))-f(a))/(-h(t))*(x-a) +
  f(a), x=Domain, t=0..0.99, color=blue ):
> display( Curve, FixedPt, RightSecants, LeftSecants );
```

$$f := x \rightarrow \left| x^3 + 2\,x \right|$$



4.4 Using animated demonstrations in the classroom

When you use an animation for classroom demonstration, you will probably want to begin with the animation paused so that you can talk some about the first frame and tell the students what to watch for. Then step through a few frames, one at a time, discussing individual ones with the class so the students will understand what is happening when the animation runs.

If the plot is three-dimensional, it helps to rotate it so the students can see it from several points of view. This helps them to get oriented and simply to understand better what they are seeing. Some gradual rotation also seems to enhance the illusion of three-dimensionality. Also, some of the plot's features may at first be hidden. My experience is that two things will cause some students to lose the orientation of a three-dimensional plot: one is rotating it too quickly; the other is rotating it all the way around. Generally, then, I will rotate it slowly from side to side, 30 or 40 degrees each way.

To ensure that everyone can see the plot clearly, set the **Zoom Factor** under the **View** menu to be at least 150%. The buttons with the magnifying glass on them adjust the zoom factor, too. You can also enlarge the plot window (by clicking on the plot, then clicking and dragging one of the black squares at the corners and edges) to fill the screen. If you are in a very large room, you might want to enlarge the plot window beyond the size of the screen. This has the effect of cropping out the white space and maximizing the size of the plot itself.

4.5 Watching a curve being drawn

Maple has a special-purpose procedure that animates the production of a curve, as if it were being drawn on a chalkboard. We might attempt to create this ourselves using `animate` with something such as

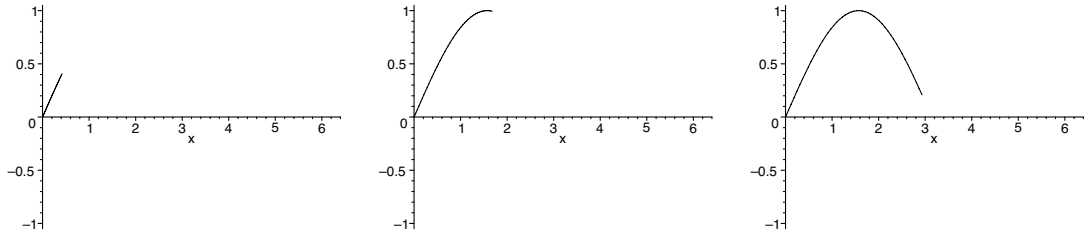
```
> with( plots ):
> animate( sin(x), x=0..t, t=0.1..2*Pi );
```

but Maple won't accept the variable endpoint of the domain. The procedure that is designed to do this is `animatecurve` in the `plots` package. It will animate the drawing of a curve in two dimensions. The syntax is

`animatecurve(` $f(x)$`,` x`=`a`..`b`,` options `)`

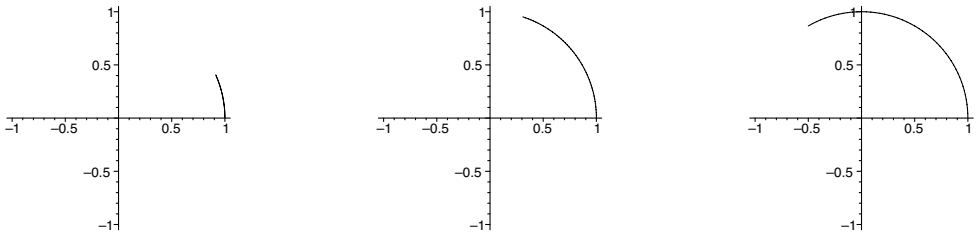
and the options are the same as those of `animate`, including the `frames` option, which defaults to 16. For example,

```
> with( plots ):
> animatecurve( sin(x), x=0..2*Pi, color=green );
```



The `animatecurve` procedure will also accept functions in parametric form, such as

```
> with( plots ):
> animatecurve( [cos(theta), sin(theta), theta=0..2*Pi],
  scaling=constrained );
```



4.6 Demonstration: The squeeze theorem

Let's create an animated demonstration of the squeeze theorem: if $f(x) \leq g(x) \leq h(x)$ for all x in some open interval containing a (except, possibly, a itself) and if $\lim_{x\to a} f(x) = L = \lim_{x\to a} h(x)$, then $\lim_{x\to a} g(x) = L$ as well. We will take as our example, $\lim_{x\to 0} x^2 \sin(1/x)$. Since $-1 \leq \sin(1/x) \leq 1$, we have $-x^2 \leq x^2 \sin(1/x) \leq x^2$, and we want to see $x^2 \sin(1/x)$ being squeezed to 0 by $-x^2$ and x^2 as $x \to 0$. First we define the function g and plot its bounds, the background plot.

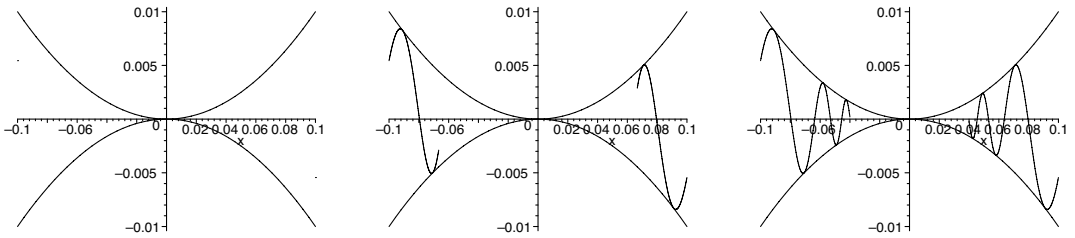
```
> restart:
> with( plots ):
> setoptions( thickness = 2 ):
> g := x -> x^2*sin(1/x):
> Bounds := plot( {-x^2, x^2}, x=-0.1..0.1, color=red ):
```

Next, we create the animated elements. Since we are demonstrating a two-sided limit, it would be ideal if the curve could be produced as approaching 0 both from the left and from the right, as we might draw it on a chalkboard. To accomplish this, we choose parametric form and plot $[\,x, g(x)\,]$ for the approach from the left, and $[\,-x, g(-x)\,]$ for the approach from the right. Although parametric form is not necessary for the left-side approach, it is for the right-side approach. This is because we would like to draw the curve backwards—that is, from right to left—and Maple will not accept a domain such as `x=0.1..0`. We use the domain `x=-0.1..0` for both, and Maple is happy since $-0.1 < 0$.

```
> LeftSide := animatecurve( [x, g(x), x=-0.1..0],
  numpoints=200, color=blue ):
> RightSide := animatecurve( [-x, g(-x), x=-0.1..0],
  numpoints=200, color=blue ):
```

Finally, we display the background plot and the animated elements.

```
> display( Bounds, LeftSide, RightSide );
```



The example we have used is a typical one, but it is easy to change to your favorite. One way to use the animation is to show it immediately after stating the theorem, so that the students are clear about what the squeeze theorem says, before working through any examples. Another good way is to show it after doing the analysis of this particular limit (or your favorite) on the blackboard so that the demonstration verifies the analysis. Either way, the animation gives the students a lasting mental image of the squeeze theorem.

4.7 Animating a function of two variables

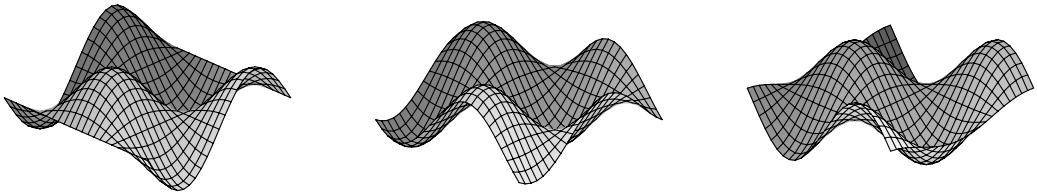
The `animate3d` procedure, within the `plots` package and similar to the `animate` procedure, is useful for producing an animation of a function of two variables. This procedure creates 8 frames, by default, and displays them in sequence. The first argument of `animate3d` is a function, possibly in parametric form, of three variables. Two of the variables, say x and y, are the

independent variables of the function to be animated. The other, say t, is the frame variable, its varying values causing the frames to differ from each other. The syntax is

`animate3d(` $F(x,y,t)$`,` x`=`$a..b$`,` y`=`$c..d$`,` t`=`$p..q$`,` `options )`

For example,

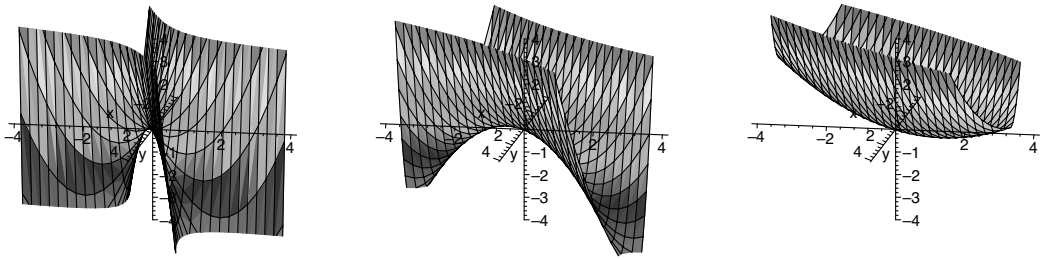
```
> with( plots ):
> animate3d( sin(x-t)*cos(y-t), x=0..2*Pi, y=0..2*Pi,
  t=0..Pi );
```



which plots $f(x,y) = \sin x \cos y$ on the square region $[0, 2\pi] \times [0, 2\pi]$ with a phase shift increasing from 0 to π.

The options are the same as those for `plot3d`, with the addition of the `frames` option, by which you can specify a number of frames other than the default 8 for the animation. For example,

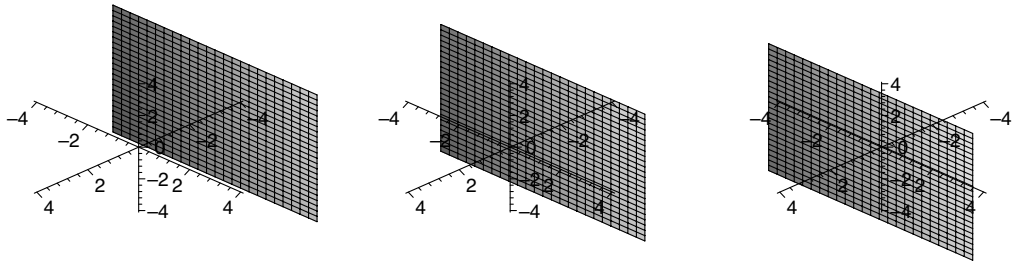
```
> with( plots ):
> animate3d( x^2 + t*x*y + y^2, x=-4..4, y=-4..4, t=-4..4,
  frames=20, numpoints=900, shading=zhue, view=-4..4,
  axes=normal, orientation=[10,70] );
```



which shows what happens to a paraboloid as the coefficient of the cross-product term xy is varied.

The function that you are animating can be in parametric form. For example, the following is an animation of a vertical plane moving along the x-axis.

```
> with( plots ):
> animate3d( [t, y, z], y=-4..4, z=-4..4, t=-3..3,
  axes=normal );
```



Recall that the domains for the parameters, in this case y and z, are given outside the brackets for three-dimensional parametric forms. Here, t is again the frame variable.

4.8 Demonstrations: Hyperboloids

The key to understanding why quadric surfaces have the shapes they do is an analysis of their traces (cross-sections) in various planes parallel or orthogonal to the axis of the surface. To keep things simple, we will restrict ourselves to surfaces with axis the z-axis, so the planes we will be interested in will be parallel to the coordinate planes. We will carry out the analysis, then create animations that illustrate the geometry and confirm our analysis.

4.8.1 Hyperboloid of one sheet

We begin with $x^2/4 + y^2/9 - z^2 = 1$, a hyperboloid of one sheet. To find its traces in vertical planes parallel to the yz-plane, we set $x = k$, where k is a constant. Rearranging, we have

$$\frac{4y^2}{9(4-k^2)} - \frac{4z^2}{4-k^2} = 1 \quad \text{if } k \neq \pm 2$$

$$z = \pm\frac{1}{3}y \quad \text{if } k = \pm 2$$

a family of hyperbolas together with their asymptotes. The branches are oriented to the left and right of the z-axis when $|k| < 2$ and oriented above and below the y-axis when $|k| > 2$. Similarly, the traces in planes parallel to the xz-plane form a family of hyperbolas. For the traces in horizontal planes, we set $z = k$ and rearrange

$$\frac{x^2}{4(1+k^2)} + \frac{y^2}{9(1+k^2)} = 1$$

producing ellipses for all values of k.

Let's create an animation to demonstrate this geometry. We would like to see a plane moving through the surface and slicing it in the various traces.

First, we need to create the background plot, the hyperboloid. To produce a high-quality plot, we will change to cylindrical coordinates, express r as a function of θ and z, and use `cylinderplot` as we did in Section 3.8.

```
> restart:
> with( plots ):
> setoptions3d( axes=boxed, orientation=[30,75],
  labels=["x","y",""], labelfont=[TIMES,BOLDITALIC,24],
  axesfont=[HELVETICA,18] ):
> r := (theta,z) ->
  6*sqrt((z^2+1)/(9*(cos(theta))^2+4*(sin(theta))^2));
> Hyperboloid1 := cylinderplot( r(theta,z), theta=0..2*Pi,
  z=-4..4 ):
```

$$r := (\theta,\, z) \to 6\sqrt{\frac{z^2+1}{9\cos(\theta)^2+4\sin(\theta)^2}}$$

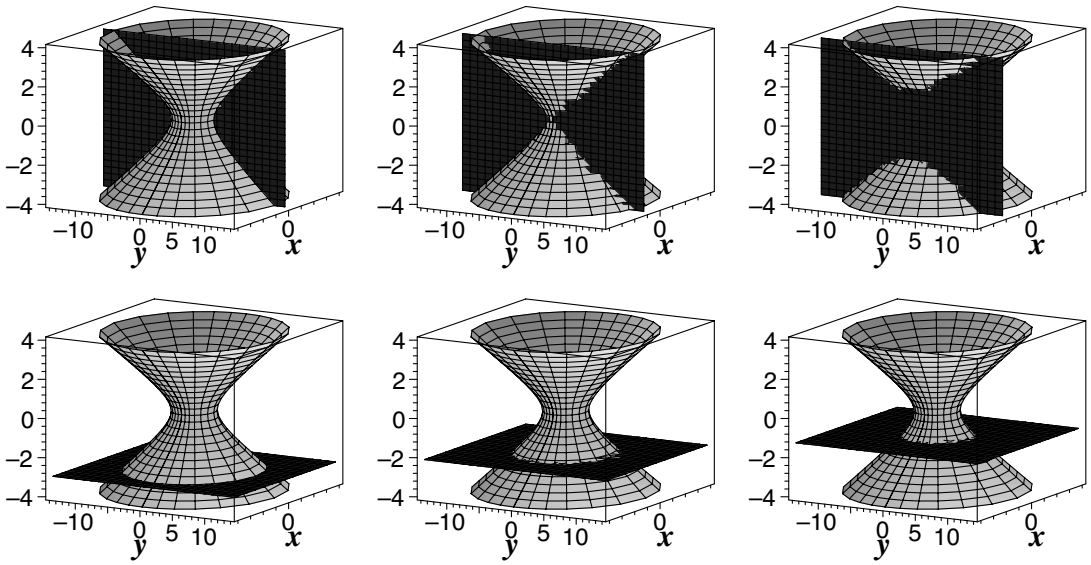
Here, we have also specified the options we prefer for this demonstration. In particular, the font is large enough to be read at a distance from a projection screen.

Next, we create the animated elements, the vertical and horizontal planes, using parametric form for the vertical planes because they are not functions of x and y.

```
> VerticalPlanes := animate3d( [k,y,z], y=-14..14, z=-4..4,
  k=0..7, color=blue ):
> HorizontalPlanes := animate3d( k, x=-8..8, y=-14..14,
  k=-3..3, color=blue ):
```

Finally, we display the results.

```
> display( Hyperboloid1, VerticalPlanes );
> display( Hyperboloid1, HorizontalPlanes );
```



As predicted, the vertical planes intersect the hyperboloid in a family of hyperbolas with the branches switching orientation, and the horizontal planes intersect in ellipses. The choice to use the default number of frames, 8, was a judicious one. With 8 frames and `k=0..7`, the k-values will be $0, 1, 2, \ldots, 7$. The frames will, therefore, include $k = 2$, and the asymptotes of the family of hyperbolas will be among the traces.

4.8.2 Hyperboloid of two sheets

We turn now to the hyperboloid of two sheets $-x^2/4 - y^2/9 + z^2 = 1$. To find its traces in planes parallel to the yz-plane, we set $x = k$ and rearrange

$$-\frac{4y^2}{9(4+k^2)} + \frac{4z^2}{4+k^2} = 1$$

These are hyperbolas whose branches are oriented above and below the y-axis for all k. The traces in planes parallel to the xz-plane are hyperbolas with branches similarly oriented. Setting $z = k$ to find the traces in horizontal planes, we have

$$\frac{x^2}{4(k^2-1)} + \frac{y^2}{9(k^2-1)} = 1 \quad \text{if } k \neq \pm 1$$

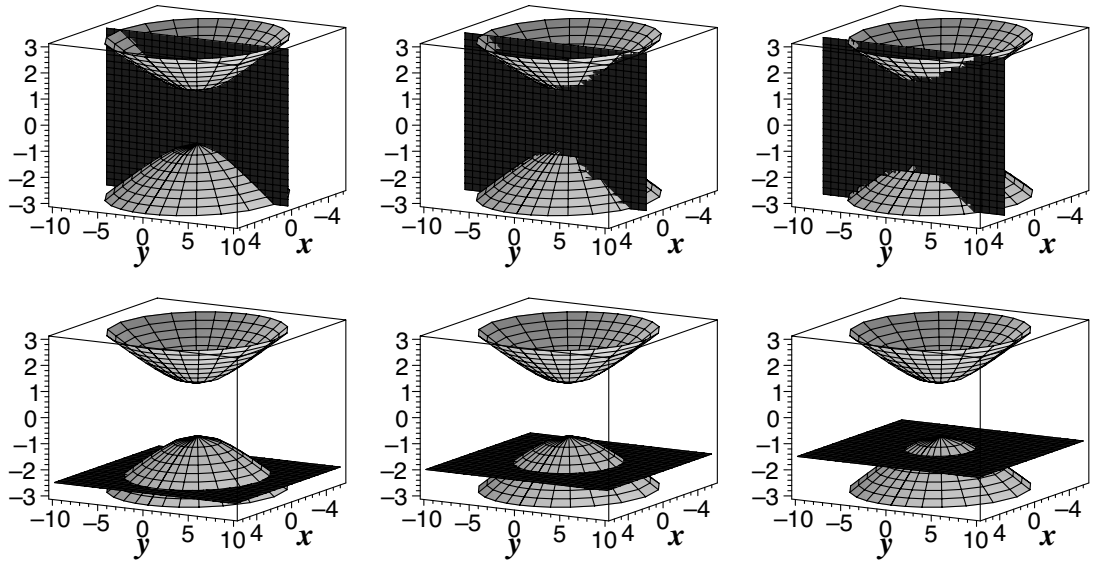
$$\text{the point } (0,0) \quad \text{if } k = \pm 1$$

Therefore, the trace is an ellipse when $|k| > 1$, a point when $|k| = 1$, and there is no trace at all when $|k| < 1$.

We proceed in a similar way to that above, using the plot of the hyperboloid of two sheets that we developed in Section 3.8.

```
> restart:
> with( plots ):
> setoptions3d( axes=boxed, orientation=[30,75],
  labels=["x","y",""], labelfont=[TIMES,BOLDITALIC,24],
  axesfont=[HELVETICA,18] ):
> r := (theta,z) ->
  6*sqrt((z^2-1)/(9*cos(theta)^2+4*sin(theta)^2));
> Hyperboloid2 := cylinderplot( r(theta,z), theta=0..2*Pi,
  z=-3..3 ):
> VerticalPlanes := animate3d( [k,y,z], y=-10..10, z=-3..3,
  k=0..5, color=blue ):
> HorizontalPlanes := animate3d( k, x=-6..6, y=-10..10,
  k=-5/2..5/2, frames=11, color=blue ):
> display( Hyperboloid2, VerticalPlanes );
> display( Hyperboloid2, HorizontalPlanes );
```

$$r := (\theta,\, z) \to 6\sqrt{\frac{z^2-1}{9\cos(\theta)^2 + 4\sin(\theta)^2}}$$



Again, the number of frames for the horizontal planes was chosen with the special cases in mind. With 11 frames and `k=-5/2..5/2`, the k-values will be $-5/2, -2, -3/2, \ldots, 5/2$, which will include $k = \pm 1$.

My own preference is to use these demonstrations just as presented here, first going through the analysis presented above with the students. Together, we predict what the traces in various planes are going to be; then we watch the demonstrations to verify that. So that the students can see how the surface and plane are oriented, begin with the animation paused and rotate the plot from side to side. Then step through the animation one frame at a time and point out the varying traces where the planes meet the surface. Then run the animation once or twice.

4.9 Demonstrations: Paraboloids

We now create animations that illustrate the geometry of two other quadric surfaces: elliptic and hyperbolic paraboloids. As with hyperboloids, we will restrict ourselves to surfaces with axis the z-axis, and we will consider planes that are parallel to the coordinate planes.

4.9.1 Elliptic paraboloid

First, we consider the elliptic paraboloid $z = x^2/4 + y^2/9$. Its traces in vertical planes $x = k$ are

$$z = \frac{y^2}{9} + \frac{k^2}{4}$$

which are parabolas, all concave upward. Similarly, the traces in vertical planes $y = k$ are concave-upward parabolas. The surface's traces in horizontal

planes $z = k$ are

$$\frac{x^2}{4k} + \frac{y^2}{9k} = 1 \qquad \text{if } k \neq 0$$

$$\text{the point } (0,0) \qquad \text{if } k = 0$$

ellipses or a point when $k \geq 0$, and no trace when $k < 0$.

To demonstrate this geometry, again we seek an animation that shows a plane moving through the surface, cutting it in varying traces as it moves. First, we create the background plot, the elliptic paraboloid. In Section 3.5, we found that we could get a good plot of a paraboloid by using `plot3d`. We will improve the plot's smoothness a little by increasing the number of points sampled beyond the default number of 625.

```
> restart:
> with( plots ):
> setoptions3d( axes=boxed, view=-1..5, orientation=[60,60],
  labels=["x","y",""], labelfont=[TIMES,BOLDITALIC,24],
  axesfont=[HELVETICA,18] ):
> EllipticParaboloid := plot3d( x^2/4 + y^2/9, x=-5..5,
  y=-7..7, numpoints=900, shading=zhue ):
```

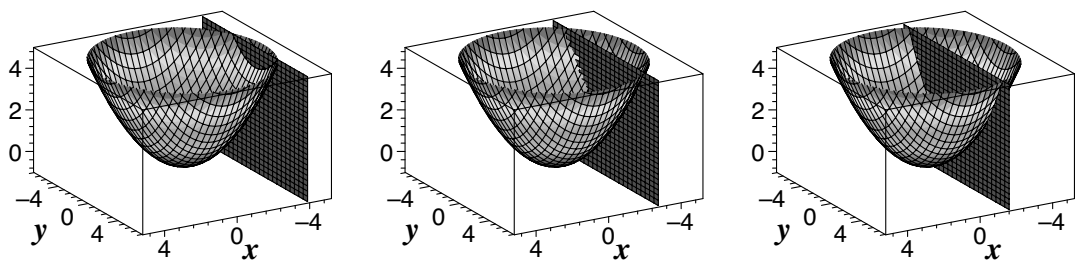
Next, we will create vertical and horizontal planes

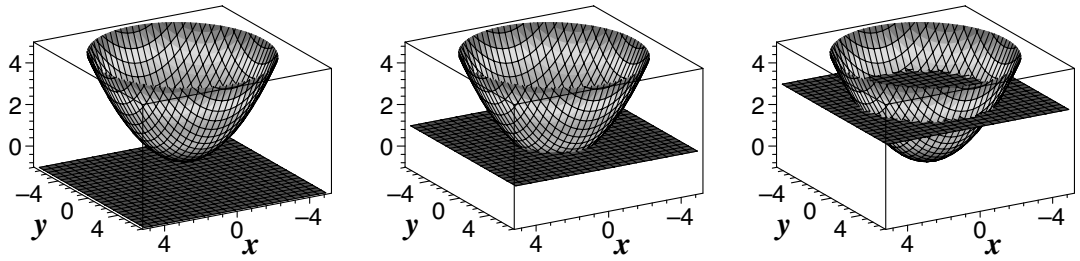
```
> VerticalPlanes := animate3d( [k,y,z], y=-7..7, z=-1..5,
  k=-4..4, color=red ):
> HorizontalPlanes := animate3d( k, x=-5..5, y=-7..7,
  k=-1..4, frames=11, color=red ):
```

where, again, the number of frames for the horizontal planes was chosen so that the plane $z = 0$, which intersects the surface in a single point, would be among those shown.

Last, we display the results.

```
> display( EllipticParaboloid, VerticalPlanes );
> display( EllipticParaboloid, HorizontalPlanes );
```





4.9.2 Hyperbolic paraboloid

Consider now the hyperbolic paraboloid $z = x^2/4 - y^2/9$. Its traces in vertical planes $x = k$ are

$$z = -\frac{y^2}{9} + \frac{k^2}{4}$$

parabolas that are concave downward. The traces in vertical planes $y = k$ are also parabolas, but concave upward. The surface's traces in horizontal planes $z = k$ are

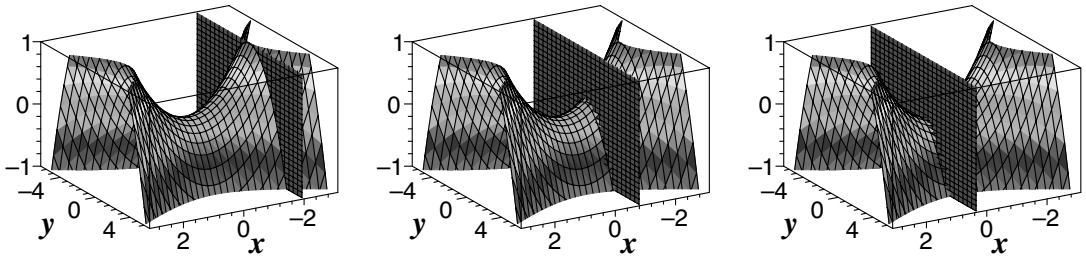
$$\frac{x^2}{4k} - \frac{y^2}{9k} = 1 \quad \text{if } k \neq 0$$

$$y = \pm\frac{3}{2}x \qquad \text{if } k = 0$$

a family of hyperbolas together with their asymptotes. The branches are oriented to the left and right of the y-axis when $k > 0$, and above and below the x-axis when $k < 0$.

We can make an animation of vertical and horizontal planes moving through this surface with a few changes to the animation we made for the elliptic paraboloid. Again, we will arrange the number of frames and the domain for k so that one of the horizontal planes shown will cut the surface in the asymptotes of the family of hyperbolas.

```
> restart:
> with( plots ):
> setoptions3d( axes=boxed, view=-1..1, orientation=[60,60],
  labels=["x","y",""], labelfont=[TIMES,BOLDITALIC,24],
  axesfont=[HELVETICA,18] ):
> HyperbolicParaboloid := plot3d( x^2/4-y^2/9, x=-3..3,
  y=-6..6, numpoints=900, shading=zhue ):
> VerticalPlanes := animate3d( [k,y,z], y=-6..6, z=-1..1,
  k=-2..2, color=red ):
> HorizontalPlanes := animate3d( k, x=-3..3, y=-6..6,
  k=-3/4..3/4, frames=9, color=red ):
> display( HyperbolicParaboloid, VerticalPlanes );
> display( HyperbolicParaboloid, HorizontalPlanes );
```



The capability of Maple to produce such a good plot, and one that can be rotated so that it can be viewed from any angle, is especially helpful for this surface. It is, I think, the most difficult to draw of all the quadric surfaces.

My own general method, on any given day, is to prove the theorem or do the analysis, and then show the examples. I have a colleague who does things the other way around, and very effectively. He shows a few examples to motivate the theorem, then proves the general result. Although I haven't used these demonstrations this way myself, I think it would work well to show the animations first, pausing them several times and asking students to identify the traces in vertical and horizontal planes, and then prove that the traces are what they appear to be.

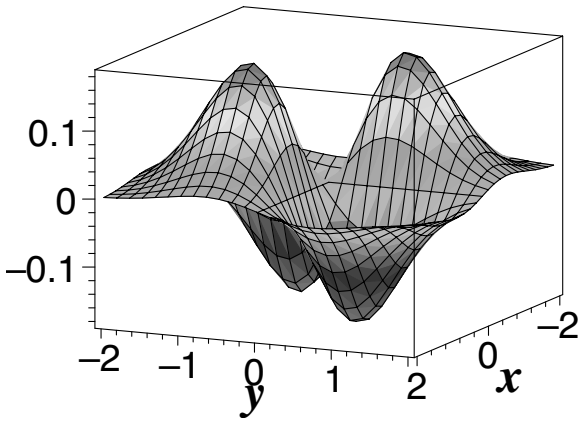
4.10 Demonstration: Level curves and contour plots

The early stages of a study of functions of two real variables typically involve analyzing the level curves $f(x, y) = k$ for various constants k. This amounts to slicing the surface with horizontal planes $z = k$. We seek a demonstration, similar to those we have made for quadric surfaces, that shows a horizontal plane moving along the z-axis and intersecting the surface in varying level curves. We will also plot the surface with the level curves shown in position.

As our example, we will use the function $f(x, y) = -xye^{-x^2-y^2}$. First, we store and display the surface.

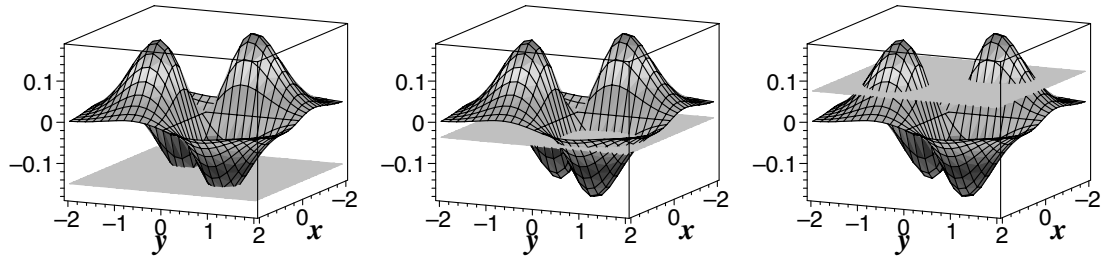
```
> restart:
> with( plots ):
> setoptions3d( labels=["x","y",""],
  labelfont=[TIMES,BOLDITALIC,24], axesfont=[HELVETICA,18] ):
> f := (x,y) -> -x*y*exp(-x^2-y^2);
> Surface := plot3d( f(x,y), x=-2..2, y=-2..2, style=patch,
  shading=zhue, axes=boxed, orientation=[25,75] ):
> display( Surface );
```

$$f := (x,\, y) \rightarrow -x\, y\, \mathbf{e}^{(-x^2-y^2)}$$



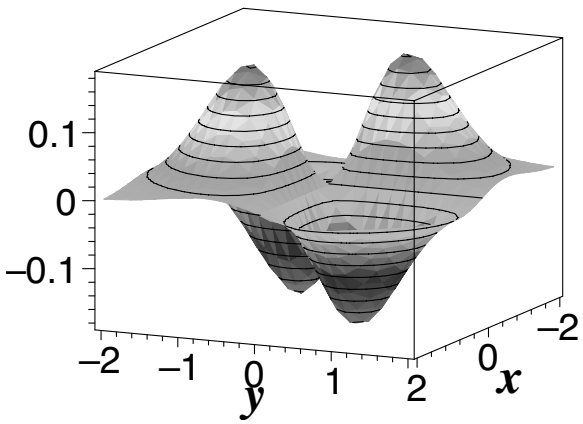
We generate the horizontal planes, just as we have in the quadric surface demonstrations, and display the background plot and the animated element.

```
> HorizontalPlanes := animate3d( k, x=-2..2, y=-2..2,
  k=-0.15..0.15, frames=9, style=patchnogrid, color=gray ):
> display( Surface, HorizontalPlanes );
```



To show the surface with its level curves in place, we can use the `style` option.

```
> display( plot3d( f(x,y), x=-2..2, y=-2..2,
  style=patchcontour, shading=zhue, axes=boxed,
  orientation=[25,75] ) );
```

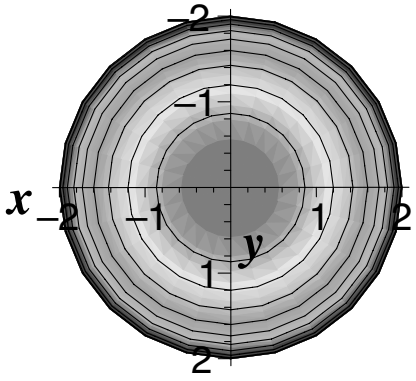


Here, we have used `display` so that the options we chose with `setoptions3d` will be in effect. (See the footnote in Section 2.13.)

Here are some suggestions for using this demonstration. Shortly after defining functions of two variables and their surfaces, explain to the students that one way to understand the shape of a surface is to slice it with horizontal planes, yielding the level curves. We can then sketch the curves in the plane, but we need to imagine them stacked in space to visualize the surface. First, show the plot of the example surface and rotate it a little. Then show the animation a few times so the students have the image in mind of the plane moving up the z-axis. It helps to rotate the plot forward a little more so that the positive end of the z-axis comes toward you, before you run the animation each time. This way, the point of view becomes more from above the xy-plane than from the side of it, and the intersections of the planes with the surface come more directly into view. Then show the third plot with the level curves in place. Rotate this plot until the point of view is on the positive end of the z-axis, and you are looking straight down onto the xy-plane. At this point, we see the projections of the level curves onto the xy-plane. Tell the students that this is the kind of diagram they'll be drawing when you ask them to sketch a contour plot.

Finally, let's plot a different surface oriented so that its level curves are projected onto the xy-plane, showing the contour plot.

```
> plot3d( sqrt(4-z^2), theta=0..2*Pi, z=0..2,
  coords=cylindrical, scaling=constrained,
  style=patchcontour, axes=normal, shading=zhue,
  orientation=[0,0] );
```



When you show this plot of concentric circles in the plane, tell the students that these are level curves projected onto the xy-plane—a contour plot—and you want them to guess the shape of the surface when the contours are stacked in space. When the students make contour plots, they label the level curves with the corresponding values of k. In this plot, instead of the labels, we have color. The blue end of the spectrum represents the lower values of k; the red end, the higher values of k. Most of the students, although not all, can guess that the surface is either a hemisphere or something resembling one. To verify

that, rotate the plot so that the point of view is the usual one, somewhere in the first octant.

Now is a good time for you to begin writing some animations of your own. Think of some topic that you are teaching that might be enhanced by animation or some point that you could demonstrate using animation. Even if it is just playful experimentation with an idea related to a course, it would probably be beneficial to share that with students. Since such experimentation reflects the kind of activity that sometimes leads mathematicians to new results, it is a fine behavior to model. And it can be contagious.

Chapter 5

Building and Displaying a Frame Sequence

The convenient `animate`, `animatecurve`, and `animate3d` procedures of Chapter 4 are designed for animating a function of one or two variables. Not all animations fall into that category. In this chapter, we will meet the built-in procedure `seq`, which generates sequences, and use it in conjunction with the `display` procedure to create animations. The method we will develop is more generally applicable. It can be used when the object we want to animate is generated by another Maple procedure. We will also encounter the `student` and `Student[Calculus1]` packages, and create some more demonstrations for classroom use. One shows rectangles that approximate an area under a curve becoming more region-filling as their number increases. Another demonstrates the concept of level surfaces, extending to four dimensions the demonstration of level curves in Section 4.10. Two more show the paths of projectiles, and another shows a cycloid being generated by a chosen point on a rolling circle.

5.1 Sequences

There are three related structures in Maple: a *sequence*, a *list*, and a *set*. We have already encountered lists and sets in Section 2.1, but we will redefine them here for comparison. A *sequence* is an ordered n-tuple. A *list* is an ordered n-tuple enclosed in square brackets. A *set* is an unordered n-tuple enclosed in curly brackets. Basically, a sequence is a list without the square brackets. For example,

`3,7,Pi,4,x,19`	is a sequence
`[3,7,Pi,4,x,19]`	is a list
`{3,7,Pi,4,x,19}`	is a set

The sets `{12,5,8,5}`, `{12,5,8}`, and `{5,8,12}` are the same, but `[12,5,8,5]`, `[12,5,8]`, and `[5,8,12]` are three different lists, and `12,5,8,5` and `12,5,8` and `5,8,12` are three different sequences. The sequence without any elements is denoted `NULL`. The empty list is `[ ]`; the empty set is `{ }`.

Elements may be appended to sequences using the comma (an operator). For example, we can create a sequence S using

```
> S := 4, 9, 7;
```

$$S := 4,\ 9,\ 7$$

then append elements 13 and 40 to S using

```
> S := S, 13, 40;
```

$$S := 4,\ 9,\ 7,\ 13,\ 40$$

The i^{th} element of the sequence S can be accessed with the selection operation, $S[\,i\,]$. For example,

```
> S[1];
> S[3];
> S[5];
```

$$4$$
$$7$$
$$40$$

5.2 The `student` and `Student[Calculus1]` packages

Maple has some useful routines, particularly for calculus, in the `student` package. For example, the `showtangent` procedure, as it sounds as though it might, plots a function and its tangent line at a point. The `simpson` procedure approximates a definite integral using Simpson's one-third rule. Maple 8 also includes the upgraded `Student[Calculus1]` package offering similar procedures such as `Tangent` and `ApproximateInt`, but with much more functionality. Several of the procedures in the `Student[Calculus1]` package will even output an animation. For those readers who are using Maple 8, we will include examples where appropriate in this and subsequent chapters. For more information now, type `?student` or `?Student[Calculus1]` at the Maple prompt. Like the `plots` package, the `student` and `Student[Calculus1]` packages are not automatically loaded when you start Maple. To use them, you'll need to include a `with` statement.

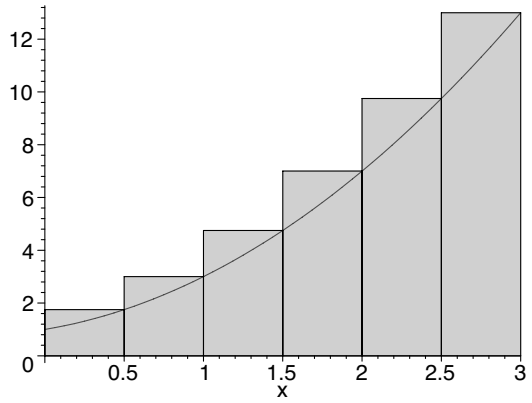
The `rightbox` procedure in the `student` package illustrates one type of rectangular approximation of a definite integral. The `rightbox` procedure plots a function f on an interval $[\,a, b\,]$ and a specified number n of rectangles (default is 4) built upon a regular partition of $[\,a, b\,]$ into n subintervals. It

uses, as the height of each rectangle, the value of f at the right-hand endpoint of each subinterval. The plot is exactly the kind of diagram you draw when you introduce the idea of areas under curves and Riemann sums. The syntax is

`rightbox(` $f(x)$`,` x`=`a`..`b`,` n`,` *options* `)`

where the options are the same as for `plot` and also include a `shading=`*color* option for the fill color of the rectangles. For example,

```
> with( student ):
> rightbox( x^2+x+1, x=0..3, 6 );
```



For evaluation of the function at left-hand endpoints or midpoints of the subintervals, there are `leftbox` and `middlebox` procedures. The `student` package also contains associated `rightsum`, `leftsum`, and `middlesum` functions that give the sum, in sigma notation, of the areas of the rectangles in each case. For example,

```
> with( student ):
> S := rightsum( x^2+x+1, x=0..3, 6 );
> value( S );
```

$$S := \frac{1}{2}\left(\sum_{i=1}^{6}\left(\frac{1}{4}\,i^2+\frac{1}{2}\,i+1\right)\right)$$

$$\frac{157}{8}$$

where the `value` function forces evaluation of the "inert" expression S.

5.3 Displaying a sequence of frames

Animation is fundamentally a process of creating a series of individual images, consecutive images differing only slightly from each other, and showing

them one after another fairly quickly so as to foster the illusion of motion. In Maple terms, the series of images is a sequence of plots. To show them, we will use the `display` procedure, and to cause them to be shown one after another, we will use the option `insequence=true`.

For example, we might initialize a short sequence of `rightbox` plots with

```
> with( student ):
> Rectangles := rightbox( x^2+x+1, x=0..3, 6 ):
```

then append a `rightbox` plot that uses 9 rectangles instead of 6,

```
> Rectangles := Rectangles, rightbox( x^2+x+1, x=0..3, 9 ):
```

then append to the existing sequence a `rightbox` plot that uses 12 rectangles,

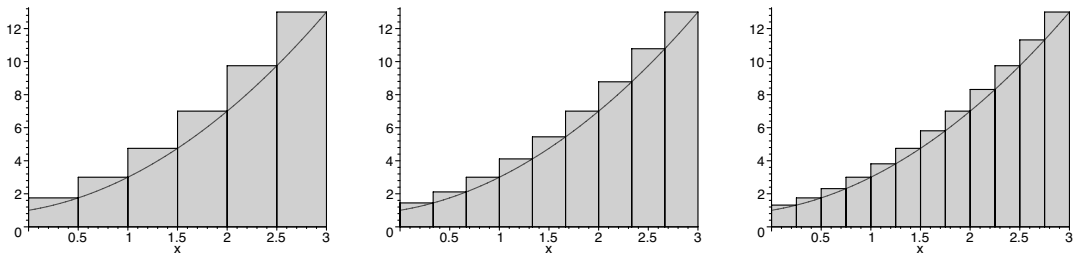
```
> Rectangles := Rectangles, rightbox( x^2+x+1, x=0..3, 12 ):
```

and then append yet another with 15 rectangles,

```
> Rectangles := Rectangles, rightbox( x^2+x+1, x=0..3, 15 ):
```

So we now have a sequence of four `rightbox` plots. We can make a very short animation from this sequence with each `rightbox` plot as a frame. We just need to display the plots using the `display` procedure with the option `insequence=true`

```
> with( plots ):
> display( Rectangles, insequence=true );
```



which displays the frames, one after the other.

We could, of course, have created this short sequence with one single statement

```
> Rectangles := rightbox( x^2+x+1, x=0..3, 6 ),
  rightbox( x^2+x+1, x=0..3, 9 ),
  rightbox( x^2+x+1, x=0..3, 12 ),
  rightbox( x^2+x+1, x=0..3, 15 ):
```

but the method of building a sequence by appending new frames to an existing sequence as we have, then displaying them in order, is a general one that we will use in more elaborate animations in later chapters. This is, in fact, what the `animate`, `animatecurve`, and `animate3d` procedures of Chapter 4 did for us automatically. They created a sequence of frames, then displayed them in order.

5.4 Building sequences with `seq`

The `seq` procedure is an automatic sequence builder. We just specify the elements in terms of an index and provide a domain for the index. One syntax is

`seq(` $f(i)$`,` i=p`..`q `)`

which, loosely speaking, creates a sequence $f(p), f(p+1), f(p+2), \ldots, f(q)$. But that description isn't quite accurate. What actually happens is that Maple initializes a sequence to `NULL`, sets $i = p$ and, if $i \leq q$, appends $f(i)$, then increments i by 1 and repeats, continuing to append $f(i)$ until the test $i \leq q$ fails, at which point $f(i)$ is not appended. For example,

```
> seq( 2*i+1, i=0..6 );
```

1, 3, 5, 7, 9, 11, 13

and

```
> seq( j, j=3.4..8.3 );
```

3.4, 4.4, 5.4, 6.4, 7.4

It isn't necessary that the domain for the index be an arithmetic sequence. For example,

```
> Fibonacci := 1, 1, 2, 3, 5, 8, 13, 21:
> seq( i^3-i^2, i=Fibonacci );
```

0, 0, 4, 18, 100, 448, 2028, 8820

The domain can even be a string. For example,

```
> seq( i, i="Any string" );
```

"A", "n", "y", " ", "s", "t", "r", "i", "n", "g"

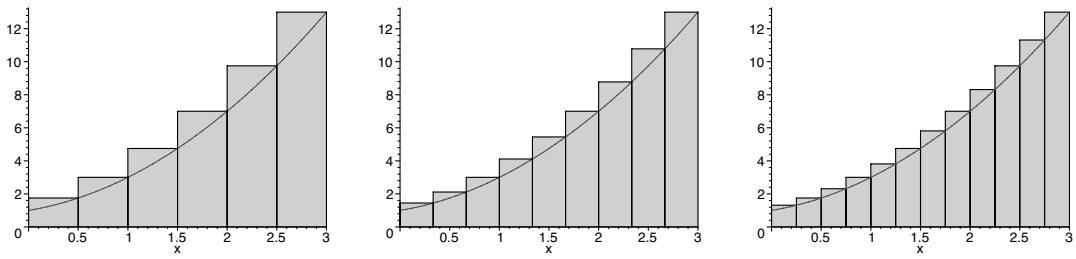
For more information, type `?seq` at the Maple prompt.

Although sometimes we will want to build such alphanumeric sequences with `seq`, for the purposes of creating animations, ours will usually be sequences of plots. We can, for example, use `seq` to create a sequence of `rightbox` plots. Returning to our example above, we could have created our short sequence of `rightbox` plots with

```
> with( student ):
> Rectangles := seq( rightbox( x^2+x+1, x=0..3, 3*i ),
  i=2..5 ):
```

and then displayed them with

```
> with( plots ):
> display( Rectangles, insequence=true );
```



This is a technique worth remembering: Create a sequence of frames using `seq`, then display them using `display` with the option `insequence=true`.

5.5 Demonstrations: Rectangular approximation of the definite integral

We now apply our technique to demonstrate the method of approximation by rectangles of the area under a curve. We wish to show that, as their number increases, the rectangles become more region-filling. In the first demonstration, we will use `seq` to build a sequence of `rightbox` plots, and we'll use enough rectangles to make it convincing. The second demonstration is for Maple 8 users; it relies on a procedure from the `Student[Calculus1]` package.

5.5.1 Using `seq` and `rightbox`

We start by calling the necessary packages and selecting a font large enough to be seen from a distance. We also choose to suppress axis labels.

```
> restart:
> with( student ):
> with( plots ):
> setoptions( labels=["",""], axesfont=[HELVETICA,18] ):
```

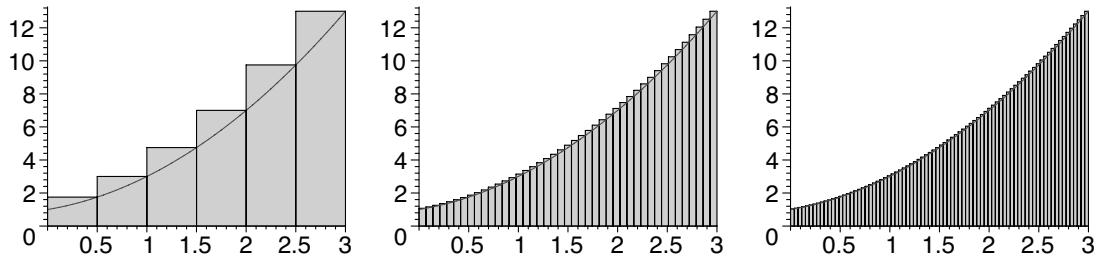
We choose an example function and domain.

```
> f := x -> x^2 + x + 1;
> a := 0;
> b := 3;
```

$$f := x \to x^2 + x + 1$$
$$a := 0$$
$$b := 3$$

Instead of writing two separate statements as we did above—one to create and store the sequence of frames, and another to display it—we can fold them together into a single command.

```
> display( seq( rightbox( f(x), x=a..b, NumRects ),
  NumRects=6..80 ), insequence=true );
```



Its form reflects its function: It `displays` a `sequence` of `rightboxes`.

This demonstration is fairly effective. It looks convincing that the sum of the areas of the rectangles converges to the area under the curve. It makes it seem reasonable, then, to define the area under the curve as the limit of this sum, or one like it. I do an example on the blackboard, finding the limit of the sum, then show the demonstration, changing the function and domain to match the example that I've worked. Then I repeat the process with a different example. Using the computer provides a welcome break from the lengthy algebraic manipulations, and the demonstration helps students visualize what the mathematics is actually accomplishing. Later, in Section 7.6, we will improve this demonstration by including in the plot the running sum of the areas of the rectangles.

5.5.2 The `RiemannSum` procedure of Maple 8

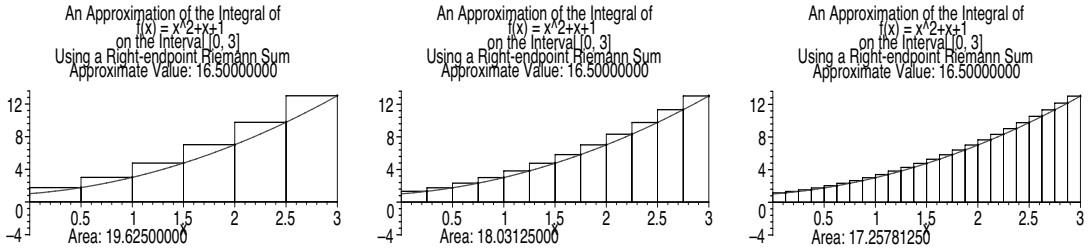
In Maple 8, the procedure `RiemannSum` in the `Student[Calculus1]` package can produce an animation much like this one. The syntax is

`RiemannSum(` $f(x)$`,` x`=`a`..`b`,` *options* `)`

Among the options, of which there are many, is `output=animation`. To match our previous animation, we will set two of the other options to `method=right` (default is `method=midpoint`) and `partition=6` (default is 10). Although the partition need not be regular, by default `RiemannSum` doubles the number of subintervals in a regular partition for each of five frames after the first one.

```
> restart:
> with( Student[Calculus1] ):
> f := x -> x^2 + x + 1;
> a := 0;
> b := 3;
> RiemannSum( f(x), x=a..b, method=right, partition=6,
  output=animation );
```

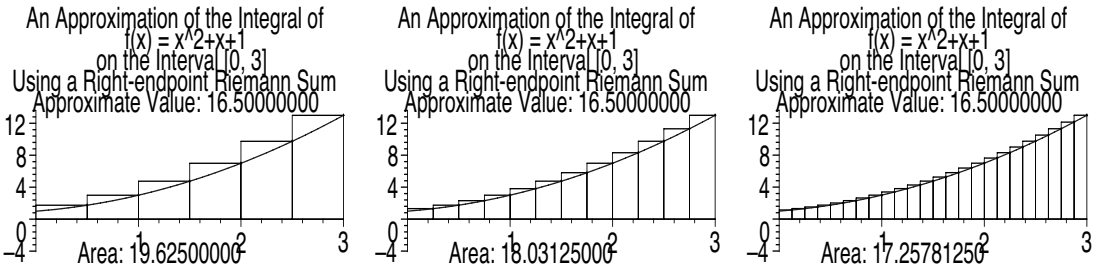
$$f := x \rightarrow x^2 + x + 1$$
$$a := 0$$
$$b := 3$$



By using some of the other options, you can make the plot more suitable for projecting onto a screen in the classroom. Using `functionoptions`, for example, you can adjust such things as the thickness of the curve and the font used for the values at the tick marks.

```
> restart:
> with( Student[Calculus1] ):
> f := x -> x^2 + x + 1;
> a := 0;
> b := 3;
> RiemannSum( f(x), x=a..b, method=right, partition=6,
  output=animation, functionoptions=[thickness=2,
  labels=["",""], axesfont=[HELVETICA,18]],
  font=[HELVETICA,18], titlefont=[HELVETICA,18] );
```

$$f := x \rightarrow x^2 + x + 1$$
$$a := 0$$
$$b := 3$$



The separate `titlefont` and `font` specifications are apparently necessary to enlarge both the title and the value for the area of the rectangles printed below the plot.

Another option is `iterations`, by which you can set the number of frames in the animation to something other than the default 6. With `boxoptions`, you can adjust the appearance of the boxes using `plot` options, such as `color` and `linestyle`. The `subpartition` option allows you to subdivide just one subinterval. The choice `subpartition=width` subdivides only the widest subinterval; `subpartition=area` subdivides only the subinterval of greatest area. The default is `subpartition=all`. The manner in which the subintervals are divided can be controlled by the `refinement` option. The default is `refinement=halve`, which divides each subinterval in half. The other choices for `refinement` are `random` or a value c, with $0 < c < 1$, which divides each subinterval $[x_i, x_{i+1}]$ at the point $x_i + c\,(x_{i+1} - x_i)$. For the complete details, type `?RiemannSum` at the Maple prompt.

5.6 Demonstration: Level surfaces

Alicia Boole Stott (1860–1940), George Boole's daughter, who was without formal education in mathematics, could visualize objects in four-dimensional space, correctly identifying three-dimensional sections of the six regular polytopes that are the analogs of the five Platonic solids. [3] For the rest of us, there is computer graphics. In Section 4.10, we used graphic output effectively to analyze three-dimensional surfaces by slicing them with planes and stacking the resulting level curves in three-dimensional space. We should be able to study four-dimensional surfaces, then, by slicing them with hyperplanes. What we cannot do is stack the resulting level surfaces in four-space.

On the eve of a new millennium, but otherwise an ordinary night in the two-dimensional world of Edwin Abbott's *Flatland* [1], an extraordinary visitor startles the protagonist, A Square. The visitor, a sphere, passes through A Square's planar world, Flatland. As the sphere enters and then leaves Flatland, A Square sees a point appear out of nowhere, then a growing circle, then a shrinking circle, then a point, then nothing. A Square sees the sphere as a circle changing over time, but it isn't. It is all of those circles at once, assembled in a three-dimensional world that he cannot perceive. Analogously, if a hypersphere were to enter our three-dimensional space, we would see a point, then a growing sphere, then a shrinking sphere, then a point, and then nothing. We would see the hypersphere as a sphere changing over time, but it would not be that. It would be all of those spheres at once, put together in a four-dimensional world that we cannot perceive.

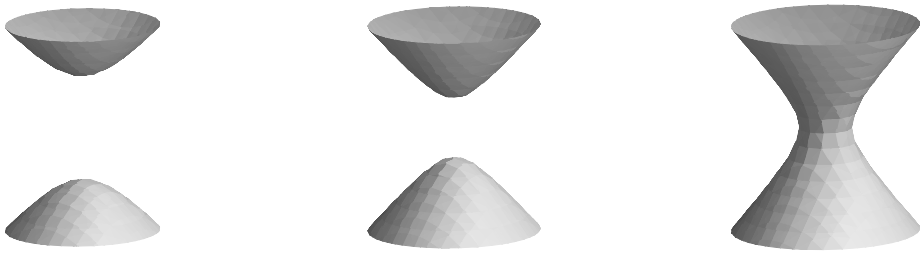
For four-dimensional surfaces, then, our own limitations are analogous to A Square's. We can perceive the level surfaces as A Square could perceive the level curves. We now make an animated demonstration for level surfaces, similar to the one we constructed in Section 4.10 for level curves. This time,

however, we won't be plotting the analog of the moving horizontal plane. That analog is a hyperplane, and we will not be able to see it, because we will be in it. We will try to think of our own three-dimensional space as a hyperplane moving along the w-axis through four-space. What we will see will look like a surface changing over time. But it is, instead, all of those surfaces assembled in four-space.

As our example, we will use the function $f(x, y, z) = x^2/4 + y^2/6 - z^2/8$ and consider its level surfaces $f(x, y, z) = k$ for various constants k. The `implicitplot3d` procedure works well enough in this case. We will create a sequence of level surfaces for values of k from $-1/2$ to $1/2$. We'll do that by using `seq` with index $i = -5, -4, \ldots, 5$ and setting $f(x, y, z) = i/10$.

```
> restart:
> with( plots ):
> f := (x^2)/4 + (y^2)/6 - (z^2)/8;
> LevelSurfs := seq( implicitplot3d( f=i/10, x=-3..3,
  y=-4..4, z=-3..3, grid=[21,21,15] ), i=-5..5 ):
> display( LevelSurfs, insequence=true, style=patchnogrid,
  orientation=[35,75], lightmodel=light2 );
```

$$f := \frac{x^2}{4} + \frac{y^2}{6} - \frac{z^2}{8}$$



Setting the `grid` option to $[21, 21, 15]$ yields fairly smooth surfaces, and the odd number of divisions in each direction, together with the fact that the sampling region is symmetric about the origin, causes the origin to be one of the points sampled. Since one of the level surfaces is a cone with vertex at the origin, this improves the appearance of that plot. The `lightmodel` option introduces some shadows as from a light source. We will pursue this technique somewhat further in Section 8.2.

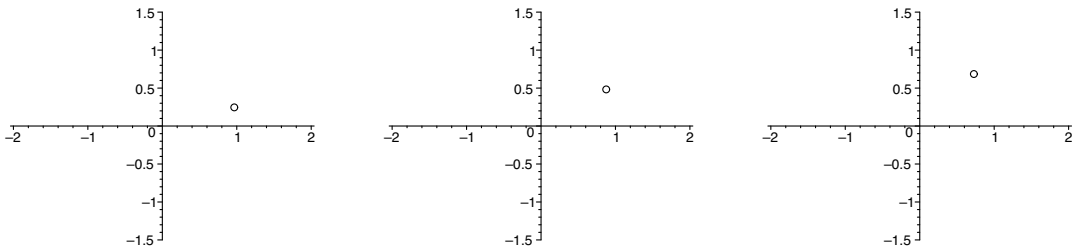
A good time to use this demonstration is directly after the one in Section 4.10 on level curves. It needs to be motivated, though. When I use it, I explain that we will be reasoning by analogy, and I set things up by talking about spheres passing through planes and hyperspheres passing through hyperplanes as above in this section. I play the animation a few times and ask the students to try to imagine that they are in a hyperplane moving through a four-dimensional object. Generally, they are quite intrigued by this idea. You might point out that the slice sequence is analogous to the one we saw in

Section 4.9.2 where we sliced a hyperbolic paraboloid with horizontal planes. In that case, we saw a family of hyperbolas together with their asymptotes, the two asymptotes being a degenerate hyperbola. In the present case, we are seeing a family of hyperboloids together with a cone, a degenerate hyperboloid.

5.7 Moving points

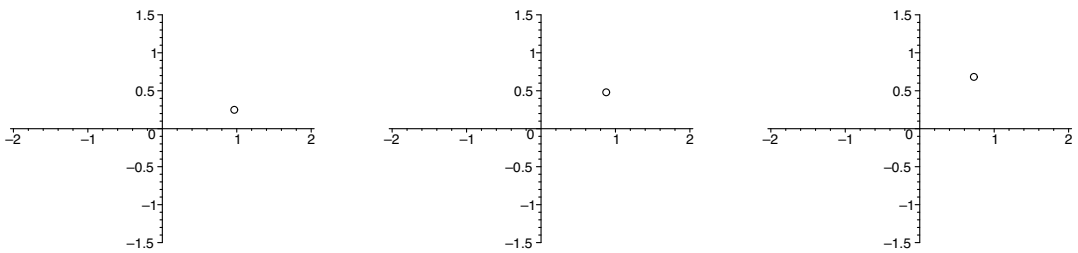
We can also use `seq` and `display` to create a moving point. One way to do this is to create the frames using `pointplot` or `pointplot3d`. Another way is to use `plot` or `plot3d` together with `style=point`. Recall from Section 2.3 that the point (a, b) is denoted as the list `[`a`,`b`]`. Recall, too, that if you choose to use the `plot` statement, set brackets are necessary even if you are only plotting a single point. So you plot a set of lists. (Alternatively, in two dimensions, you can plot a list of lists.) For example,

```
> with( plots ):
> MovingPoint := seq( pointplot( [cos(2*Pi/50*t),
  sin(2*Pi/50*t)], symbol=circle ), t=0..50 ):
> display( MovingPoint, insequence=true );
```



or

```
> with( plots ):
> MovingPoint := seq( plot( {[cos(2*Pi/50*t),
  sin(2*Pi/50*t)]}, style=point, symbol=circle ), t=0..50 ):
> display( MovingPoint, insequence=true );
```



5.8 Demonstrations: Projectiles

We create animations now that show the behavior of objects moving under the influence of gravity. We assume the typical ideal conditions, which ignore such things as air resistance and the curvature of the earth.

5.8.1 Path of a single projectile

Suppose that a projectile is fired from ground level with a force whose horizontal component is 8 m/s and whose vertical component is 98 m/s. We will create an animation that shows the path and the object moving along it.

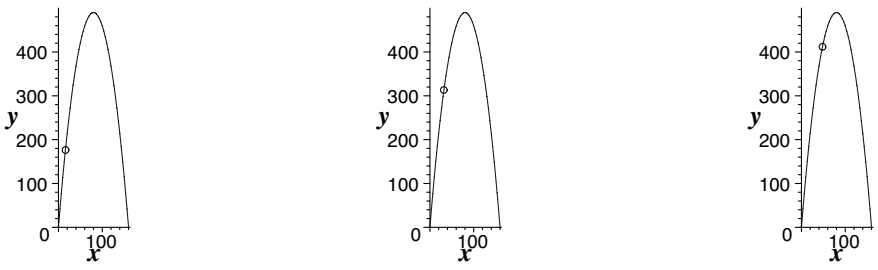
First, we choose some options, then create two component functions, $x(t)$ and $y(t)$, and plot the path on the time interval $[\,0, 20\,]$ for which the projectile remains in the air.

```
> restart:
> with( plots ):
> setoptions( labels=["x","y"],
  labelfont=[TIMES,BOLDITALIC,20], axesfont=[HELVETICA,14],
  scaling=constrained, symbol=circle, symbolsize=18 ):
> x := t -> 8*t;
> y := t -> -4.9*t^2+98*t;
> a := 0;
> b := 20;
> Curve := plot( [x(t), y(t), t=a..b], color=blue ):
```

$$x := t \to 8\,t$$
$$y := t \to -4.9\,t^2 + 98\,t$$
$$a := 0$$
$$b := 20$$

We create a moving point to traverse this path, then display the point and path.

```
> Projectile := display( seq( pointplot( [x(t),y(t)],
  color=red ), t=a..b ), insequence=true ):
> display( Curve, Projectile );
```



Notice that two `display` statements are used here. The first one establishes that the points are to be displayed in sequence. The second one displays *Curve* and *Projectile* together. This way, only the moving points are displayed in sequence, and the curve serves as the background plot. If we had left out the first `display` statement, defining *Projectile* as

```
> Projectile := seq( pointplot( [x(t),y(t)], color=red ),
  t=a..b ):
```

then the first frame would have been the curve, the second frame would have been the first point, and the next 20 frames would have been the rest of the points. The curve would have disappeared after the first frame; it would not have been a background plot.

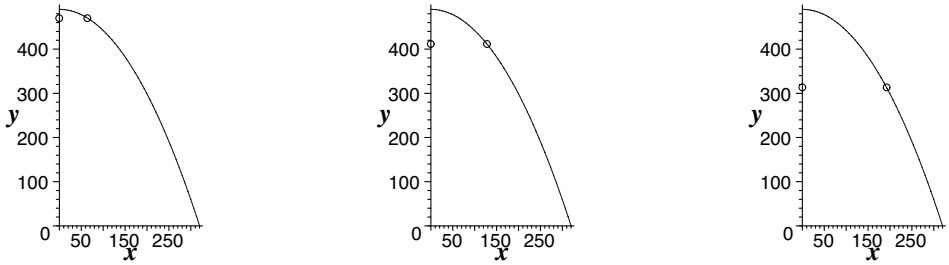
On some computer screens, the symbol for the point sometimes disappears when the plot is selected by clicking on it. You can probably make it reappear by resizing the plot window slightly. If not, try stepping through the animation one frame at a time. Failing that, try this: save the worksheet in its current state, quit Maple, and then reopen the saved worksheet.

5.8.2 Comparison of a dropped object and a propelled object

Now we create an animation to demonstrate the behavior of two objects, one that is propelled horizontally and another that is simply dropped from the same height. Because the vertical components of their initial velocities are the same (zero), gravity alone will determine their positions above the ground, and they will land simultaneously. In our example, both objects start from a height of 490 m. One object is propelled horizontally with initial velocity 32 m/s.

```
> restart:
> with( plots ):
> setoptions( labels=["x","y"],
  labelfont=[TIMES,BOLDITALIC,20], axesfont=[HELVETICA,14],
  scaling=constrained, symbol=circle, symbolsize=18 ):
> x := t -> 32*t;
> y := t -> -4.9*t^2 + 490;
> a := 0;
> b := 10;
> Curve := plot( [x(t), y(t), t=a..b], color=blue ):
> PropelledObject := display( seq( pointplot( [x(t),y(t)],
  color=red ), t=a..b ), insequence=true ):
> DroppedObject := display( seq( pointplot( [0,y(t)],
  color=red ), t=a..b ), insequence=true ):
> display( Curve, PropelledObject, DroppedObject );
```

$$x := t \to 32\,t$$
$$y := t \to -4.9\,t^2 + 490$$
$$a := 0$$
$$b := 10$$



These examples of parametric equations are simple. They make good first examples. I develop the equations on the chalkboard with help from students, show the animations, then move on to the following example.

5.9 Demonstration: Cycloid

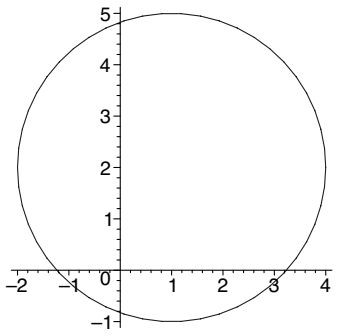
A more interesting example of a curve defined parametrically is a cycloid, the path that a point on the rim of a rolling wheel follows. In a demonstration, we'd like to see a circle rolling down the x-axis with a chosen point on the circle tracing out the curve as this happens.

For the circle, although we could create our own, the `plottools` package contains a special-purpose procedure, `circle`. Its syntax is

`circle( [`c_1`,`c_2`],` *radius*`,` *options* `)`

where (c_1, c_2) is the center, *radius* defaults to 1, and the available options are the same as those for the `plot` statement. We use `display` to plot the circle. Like the `plots` and `student` packages, the `plottools` package needs to be called using a `with` statement. For example,

```
> with( plottools ):
> with( plots ):
> display( circle( [1,2], 3, color=green ),
  scaling=constrained );
```



We begin the demonstration by calling the packages we need and arranging some details such as labels, fonts, and sizes. We also specify the plot symbol, and its size, for the points.

```
> restart:
> with( plots ):
> with( plottools ):
> setoptions( labels=["x","y"],
  labelfont=[TIMES,BOLDITALIC,20], axesfont=[HELVETICA,14],
  scaling=constrained, thickness=2, symbol=circle,
  symbolsize=18 ):
```

Next, we set up parametric equations for the cycloid, choose a radius for the generating circle, and choose a domain $[0, b]$ for θ, the angle of revolution. Two complete revolutions of the circle should be enough to demonstrate the behavior.

```
> x := theta -> r*(theta-sin(theta));
> y := theta -> r*(1 - cos(theta));
> r := 3:
> b := 4*Pi:
```

$$x := \theta \to r\,(\theta - \sin(\theta))$$
$$y := \theta \to r\,(1 - \cos(\theta))$$

We choose a number of frames N beyond the first frame, then compute an angle increment *DeltaTheta*. Each frame will differ from the last by this increment.

```
> N := 30:
> DeltaTheta := b/N:
```

We now store the plots of $N + 1$ points from $(x(0), y(0))$ to $(x(b), y(b))$ in increments of *DeltaTheta*.

```
> Point := display( seq( pointplot( [x(DeltaTheta*i),
  y(DeltaTheta*i)], color=red ), i=0..N ), insequence=true ):
```

For the cycloid, the `animatecurve` procedure (Section 4.5) will be just the thing. It will trace out the curve, showing the trail of the moving point. We use `animatecurve` with $N + 1$ frames.

```
> Cycloid := animatecurve( [x(theta), y(theta), theta=0..b],
  frames=N+1, color=blue ):
```

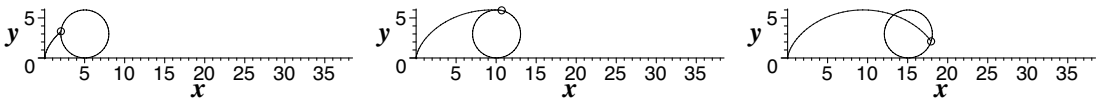
We need to determine the coordinates of the center of the generating circle. Its y-coordinate is always the radius r. As the circle lays the points of its circumference onto the x-axis, the center moves horizontally exactly

the length of the corresponding arc of the circle. So the x-coordinate of the center is just the length $r\theta$ of this arc. In any given frame i, the arc so far subtended is $r \cdot DeltaTheta \cdot i$. Therefore, the center of the generating circle is $(r \cdot DeltaTheta \cdot i, r)$.

```
> RollingCircle := display( seq( circle([r*DeltaTheta*i,r],
  r), i=0..N), insequence=true ):
```

Finally, we display the elements of the animation.

```
> display( Point, Cycloid, RollingCircle );
```



Chapter 6

Loops and Derivatives

The `animate`, `animatecurve`, `animate3d`, and `seq` commands are convenient, but as the animations we write become more elaborate, we will sometimes want more control over how the frames are produced. A loop allows that. Also, when the structure of each frame is an intricate one, although we may be able to use `seq` cleverly to create the frames, a loop can clarify the code. We will use loops to create a sequence of frames, then use `display` to display them in order. In this chapter, we will also see how to use Maple to find derivatives for functions of one and of several variables. We will encounter a convenient line-drawing procedure from the `plottools` package, and we'll create animations for generating solids and surfaces of revolution and for demonstrating Newton's method.

6.1 The `for` loop

The `for` loop provides a structure for repeatedly executing a group of statements. In one of its forms, control of the repetition is implemented by a variable that acts as a counter. The general syntax of this form of the `for` loop is

`for` i `from` m `by` j `to` n `do`
statement 1`;`
statement 2`;`
$\vdots$
statement k
`end do`

Assuming that j is positive, the loop counter i is initialized to m and, if $i \leq n$, then the statements, 1 through k, in the body of the loop are executed. Then i is incremented by j and, if $i \leq n$, the statements in the body of the loop are executed again. This continues until the test $i \leq n$ fails, at which point the body of the loop is not entered, and repetition stops. Any evaluation that may be required to determine n is done only once at the beginning of the loop, and not after each iteration of the loop. The default increment is 1, so the

`by` j specification can be omitted when $j = 1$, as is often the case. Actually, m defaults to 1 as well, so the `from` m specification could also be omitted in this case. If j is negative, then i counts down from m to n, and the test for entry into the loop each time is $i \geq n$. Here are some examples.

```
> for i from 5 to 10 do
>   i
>   end do;
```

$$5$$
$$6$$
$$7$$
$$8$$
$$9$$
$$10$$

```
> for i from 5 by 2 to 10 do
>   i
>   end do;
```

$$5$$
$$7$$
$$9$$

```
> for i from 10 by -2 to 5 do
>   i
>   end do;
```

$$10$$
$$8$$
$$6$$

Frequently, the repetitive nature of a loop is used to accumulate things—values or objects. For example,

```
> SumSquares := 0:
> SumCubes := 0:
> for i from 1 to 5 do
>   SumSquares := SumSquares + i^2;
>   SumCubes := SumCubes + i^3
>   end do;
```

$$SumSquares := 1$$
$$SumCubes := 1$$
$$SumSquares := 5$$
$$SumCubes := 9$$
$$SumSquares := 14$$

$$SumCubes := 36$$
$$SumSquares := 30$$
$$SumCubes := 100$$
$$SumSquares := 55$$
$$SumCubes := 225$$

which sums the squares and sums the cubes of the integers from 1 to 5. Before loops such as this, initialize the accumulators. In this case, *SumSquares* and *SumCubes* are both initialized to 0. On the first pass through the loop, Maple sets $i = 1$, adds 1^2 to the current value of *SumSquares* (0), then stores the result in *SumSquares*, replacing the old value of *SumSquares*. Similarly, Maple adds 1^3 to the current value of *SumCubes* (0) and replaces *SumCubes* with this value. The second time through the loop, Maple sets $i = 2$, adds 2^2 to the current value of *SumSquares* (which is 1) and 2^3 to the current value of *SumCubes* (which is also 1), then replaces *SumSquares* and *SumCubes*, respectively, with the results.

If you would like to suppress the output of the iterations of the loop, this can be done, as usual, by ending the `for` statement with a colon. We will redo the last example that way, adding two statements after the loop so that we can verify the results.

```
> SumSquares := 0:
> SumCubes := 0:
> for i from 1 to 5 do
>   SumSquares := SumSquares + i^2;
>   SumCubes := SumCubes + i^3
>   end do:
> SumSquares;
> SumCubes;
```

$$55$$
$$225$$

In like manner, we can use loops to generate sequences. For example,

```
> S := NULL:
> for i from 1 to 5 do
>   S := S, i^2
>   end do;
```

$$S := 1$$
$$S := 1, 4$$
$$S := 1, 4, 9$$
$$S := 1, 4, 9, 16$$
$$S := 1, 4, 9, 16, 25$$

Here, S is initialized to the empty sequence `NULL`, and terms are appended using the comma operator, as in Section 5.1.

Actually, the `seq` procedure of Section 5.4 is a convenient, single-purpose form of the `for` loop. The `seq` statement

```
> S := seq( i^2, i=1..5 );
```

$$S := 1, 4, 9, 16, 25$$

generates the same sequence as the previous `for` loop. In addition to convenience, however, `seq` is the more efficient structure. The cost of `seq` is linear in the sequence length; the `for` loop's cost is quadratic. So, when efficiency is paramount, `seq` is a friend.

Another form of the `for` loop implements control over the repetition by executing the body of the loop for each value of the control variable i in T, where T is a sequence, list, or set. Any evaluation that may be required to determine T is done only once at the beginning of the loop, and not after each iteration of the loop. In general, the syntax is

`for` i `in` T `do`
  statement 1`;`
  statement 2`;`
  $\vdots$
  statement k
  `end do`

Some examples are

```
> T := 1,1,2,3:
> for i in T do
>   i^2
>   end do;
```

1
1
4
9

```
> T := [1,1,2,3]:
> for i in T do
>   i^2
>   end do;
```

1
1
4
9

```
> T := {1,1,2,3}:
> for i in T do
>   i^2
>   end do;
```

$$1$$
$$4$$
$$9$$

Notice that the first and second examples generate the same output of four values, but the third example produces only three values. This is because the set $\{1, 1, 2, 3\}$ is the same as the set $\{1, 2, 3\}$.

6.2 The `while` loop

The `while` loop offers a means to execute repeatedly a group of statements as long as some condition continues to hold. The syntax is

`while` *condition* `do`
  statement 1`;`
  statement 2`;`
  $\vdots$
  statement k
  `end do`

If *condition* is true, the statements in the body of the loop are executed. Then *condition* is tested again and, if it is true, the body of the loop is executed again. This continues until *condition* is false, in which case the loop is not entered, and repetition stops. For example,

```
> F1 := 1:
> F2 := 1:
> Fibonacci := F1, F2:
> while F2 < 50 do
>   Fnext := F1 + F2;
>   Fibonacci := Fibonacci, Fnext;
>   F1 := F2;
>   F2 := Fnext
>   end do:
> Fibonacci;
```

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55$$

which generates a Fibonacci sequence until its last term is greater than or equal to 50.

Maple allows the group of statements in the body of the loop to be empty. It also permits a kind of hybrid `for...while` loop. Examples are

```
> for i from 1 while i^3<=10! do
>   end do;
> i;
```

$$154$$

which finds the first integer whose cube exceeds 10 factorial, and

```
> SumOddSquares := 0:
> for i from 1 by 2 while SumOddSquares <= 100 do
>   SumOddSquares := SumOddSquares + i^2
>   end do;
```

$$\begin{aligned} SumOddSquares &:= 1 \\ SumOddSquares &:= 10 \\ SumOddSquares &:= 35 \\ SumOddSquares &:= 84 \\ SumOddSquares &:= 165 \end{aligned}$$

which sums the squares of odd positive integers until the sum exceeds 100. For more details on loops, type `?do` at the Maple prompt.

Loops are broadly applicable structures. In this book, our interest in them is their use in creating sequences of plot structures to be displayed as frames in an animation. Loops are useful when the structure of the sequence is complex enough that the convenient `seq` procedure is not a clear favorite. The demonstrations in this chapter are examples of that. First, though, we need a few more tools.

6.3 Derivatives

There is a distinction between *functions* and *expressions*, and they are differentiated differently. In

```
> f := x -> x^5 + x^2 + 4;
> g := x^4 + 3*x^3;
```

$$\begin{aligned} f &:= x \to x^5 + x^2 + 4 \\ g &:= x^4 + 3\,x^3 \end{aligned}$$

f is a function—notice the mapping arrow—and g is an expression. Functions are differentiated using the `D` operator, which yields another function. Both functions and expressions can be differentiated using the `diff` procedure, but the result is an expression. Note the differences in the results of

```
> D(f);
> diff(f(x),x);
> diff(g,x);
```

$$x \to 5\,x^4 + 2\,x$$
$$5\,x^4 + 2\,x$$
$$4\,x^3 + 9\,x^2$$

the last two statements meaning, "Differentiate $f(x)$ with respect to x" and "Differentiate g with respect to x."

Because the `D` operator yields a function, we will generally find it more useful than `diff` because the derivative may then be evaluated at a point using notation similar to standard mathematical notation. For example, the statements

```
> f(1);
> D(f)(1);
```

$$6$$
$$7$$

evaluate f and its derivative at 1. The counterpart for expressions uses the `eval` procedure:

```
> eval(g,x=1);
> eval(diff(g,x),x=1);
```

$$4$$
$$13$$

Higher derivatives are denoted `D@@`k. The effect is to apply the `D` operator k times. For example,

```
> f := x-> a*x^4 + b*x^3 + c*x^2 + d*x + e;
> D(f);
> (D@@2)(f);
> (D@@3)(f);
> (D@@4)(f);
```

$$f := x \to a\,x^4 + b\,x^3 + c\,x^2 + d\,x + e$$
$$x \to 4\,a\,x^3 + 3\,b\,x^2 + 2\,c\,x + d$$
$$x \to 12\,a\,x^2 + 6\,b\,x + 2\,c$$
$$x \to 24\,a\,x + 6\,b$$
$$x \to 24\,a$$

Notice the parentheses, which are necessary, around `D@@`k.

For partial derivatives of functions of several variables, the notation is `D[`k`](`f`)`, similar to the mathematical notation $D_k f$ for the partial derivative of f with respect to the k^{th} variable. For example,

```
> f := (x,y,z) -> x^3*y^2 + y^3*z^2 + x*y*z;
> D[1](f);
> D[2](f);
> D[3](f);
```

$$f := (x,\, y,\, z) \to x^3\, y^2 + y^3\, z^2 + x\, y\, z$$
$$(x,\, y,\, z) \to 3\, x^2\, y^2 + y\, z$$
$$(x,\, y,\, z) \to 2\, x^3\, y + 3\, y^2\, z^2 + x\, z$$
$$(x,\, y,\, z) \to 2\, y^3\, z + x\, y$$

computes the partial derivatives $\partial f/\partial x$, $\partial f/\partial y$, and $\partial f/\partial z$, respectively, of the function $f(x, y, z) = x^3y^2 + y^3z^2 + xyz$. Higher partial derivatives are specified by multiple selectors (subscripts). For example,

```
> D[1,1](f);
> D[2,2](f);
> D[2,1](f);
> D[3,2,1](f);
```

$$(x,\, y,\, z) \to 6\, x\, y^2$$
$$(x,\, y,\, z) \to 2\, x^3 + 6\, y\, z^2$$
$$(x,\, y,\, z) \to 6\, x^2\, y + z$$
$$1$$

are $\partial^2 f/\partial x^2$, $\partial^2 f/\partial y^2$, $\partial^2 f/\partial y\, \partial x$, and $\partial^3 f/\partial z\, \partial y\, \partial x$, respectively.

6.4 The `line` procedure

The `plottools` package contains a convenient line-drawing procedure that will be useful in the next demonstration. The `line` procedure creates a line segment between two points in either two or three dimensions. It has the form

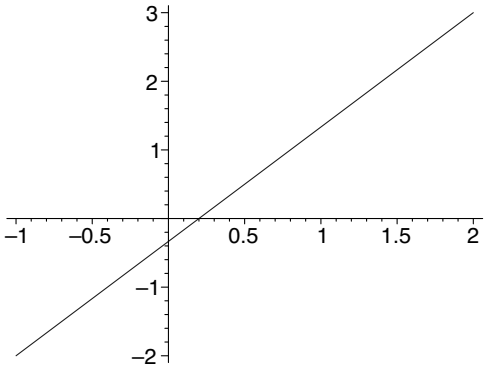
`line( [`x_1`,`y_1`], [`x_2`,`y_2`],` *options* `)`

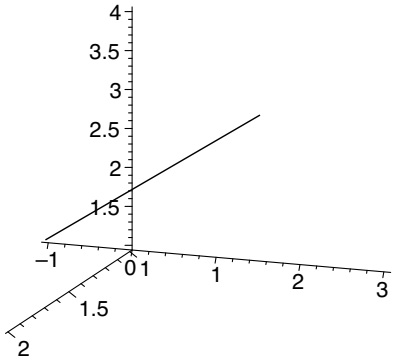
or

`line( [`x_1`,`y_1`,`z_1`], [`x_2`,`y_2`,`z_2`],` *options* `)`

The structure can then be plotted using `display`. All the `plot` or `plot3d` options, such as `color` and `thickness`, are available. For example,

```
> with( plottools ):
> L1 := line( [-1,-2], [2,3], color=magenta ):
> L2 := line( [1,-1,1], [2,3,4], color=blue, thickness=3 ):
> with( plots ):
> display( L1 );
> display( L2, axes=normal, orientation=[20,70] );
```





6.5 Demonstrations: Newton's method

We will now put together several of the procedures discussed above to create an animated demonstration of Newton's method (or the Newton-Raphson method) for approximating zeros of functions. Then, for Maple 8 users, we will construct a similar demonstration that uses the `Student[Calculus1]` package, and, by modifying it, experiment some with Newton's method.

6.5.1 Using a `for` loop

We seek an animation that shows a starting value as a point on the x-axis, as shown in Figure 6.1, constructs a vertical line segment from that point to

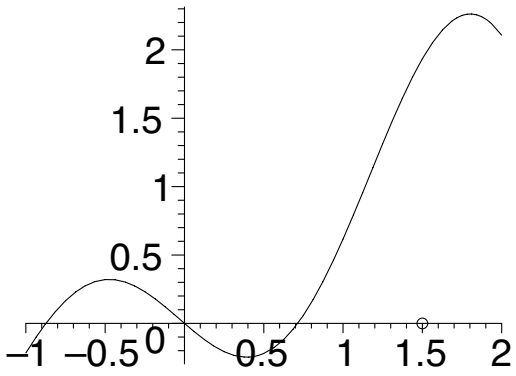


FIGURE 6.1: Newton's method, starting value

the graph of the function, as shown in Figure 6.2, then shows the tangent line

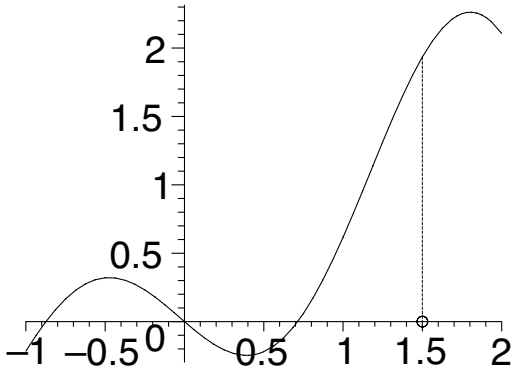


FIGURE 6.2: Newton's method, vertical line segment from starting value to graph

to the graph at the point on the curve and the tangent's point of intersection with the x-axis, as shown in Figure 6.3. Using this point of intersection in place of the starting value, we want to repeat the process.

We begin by calling the `plots` package so that we can use `display`, and the `plottools` package so that we can use `line`, and by establishing some preferences.

```
> restart:
> with( plots ):
> with( plottools ):
> setoptions( thickness=2, symbol=circle, symbolsize=16,
  axesfont=[HELVETICA,18], labels=["","] ):
```

Next, we choose a function to use as an example, some limits for its domain,

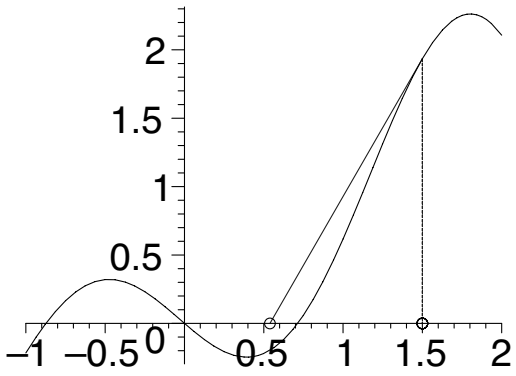


FIGURE 6.3: Newton's method, starting value, vertical segment, and tangent line with its x-intercept

a starting value, and a number of iterations.

```
> f := x -> 1/5*x^2 - x*cos(2*x);
> a := -1:
> b := 2:
> StartingValue := 1.5;
> NumIterations := 3:
```

$$f := x \to \frac{1}{5}\,x^2 - x\cos(2\,x)$$
$$StartingValue := 1.5$$

Now we plot the curve and create two functions. The first, *TangentLine*, is a function of two variables that creates a plot structure of a line from the point on the curve above the old estimate to the point where the tangent line to the curve meets the x-axis. This intersection point is the new estimate of the zero. The second function, *VertLine*, creates a vertical line from the current estimate x on the x-axis to the curve.

```
> Curve := plot( f(x), x=a..b, color=green ):
> TangentLine := (xOld,xNew) -> line( [xOld,f(xOld)],
  [xNew,0], color=blue ):
> VertLine := x -> line( [x,f(x)], [x,0], color=blue,
  linestyle=DASH ):
```

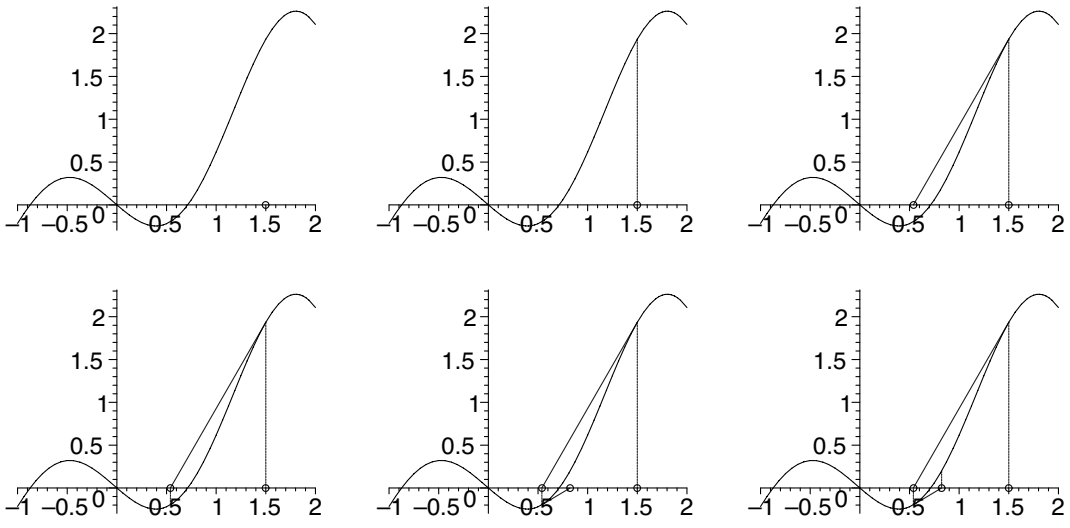
The heart of the code is a `for` loop. We initialize the frame sequence with a plot of the starting value on the x-axis, and we initialize *xOld* to be that starting value. In the body of the loop, we compute *xNew* by Newton's method, plot and name the point on the x-axis at the new estimate, and append two structures to the frame sequence. The first displays the frame sequence as it currently exists together with the vertical line to the curve. The second also displays the entire (new) frame sequence together with the

tangent line and its intersection with the x-axis. This inclusion of the entire frame sequence in each new frame means that, as each line or point is plotted, it will remain in the plot—it will be *sustained*. This is a more effective way to demonstrate the geometry of Newton's method than allowing the elements to disappear after they are plotted. At the end of the `for` loop, we set *xOld* equal to the new estimate to prepare for the next iteration.

```
> FrameSequence := pointplot( [StartingValue,0],
  color=red ):
> xOld := StartingValue:
> for i from 1 to NumIterations do
>   xNew := xOld - f(xOld)/D(f)(xOld);
>   Point := pointplot( [xNew,0], color=red );
>   FrameSequence := FrameSequence,
    display( FrameSequence, VertLine(xOld) );
>   FrameSequence := FrameSequence,
    display( FrameSequence, TangentLine(xOld,xNew), Point );
>   xOld := xNew
>   end do:
```

Finally, we specify that the frames are to be displayed in sequence, then display them together with the background plot, *Curve*.

```
> Frames := display( FrameSequence, insequence=true ):
> display( Curve, Frames );
```



One good way to use this demonstration is to develop the mathematics as you step through the first few frames. I start by explaining that we seek a method to arrive at approximate solutions to equations that we have no means to solve exactly. This amounts to finding a zero of a function, and our approach will be to improve on an initial estimate. I tell the students that, in our examples, the first estimate will be a poor one because Newton's method

works so well that they wouldn't be able to see what was happening very well if it were a good one. I show the first frame of the animation, which includes only the curve and the initial estimate, and I point out the zeros whose values we are trying to find. Then I show the second frame, which sends a line from the estimate to the curve, and I explain that we'll use the tangent line at that point as a rough approximation of the function, hoping that the point where the tangent line crosses the x-axis won't be far from the point where the curve itself crosses. Then I show the third frame, which includes the tangent line and the new estimate. At this point, I develop the formula for Newton's method on the blackboard. After this, I step through the next two frames, which show, first the line constructed from the current estimate to the curve, and then the tangent line to the new estimate. Then I step through the last two frames, which show the final estimate. After this, I run the animation straight through once or twice so the students can see the method at work.

Newton's method is amazing, really. It converges incredibly quickly. Of course, you can foil it if you try, but Newton's method is fairly robust. I find it quite rewarding to share this beautiful, three-and-half-century-old method with my students, who are only a little younger than Newton was when he invented it. I know of no better way to demonstrate Newton's method than this.

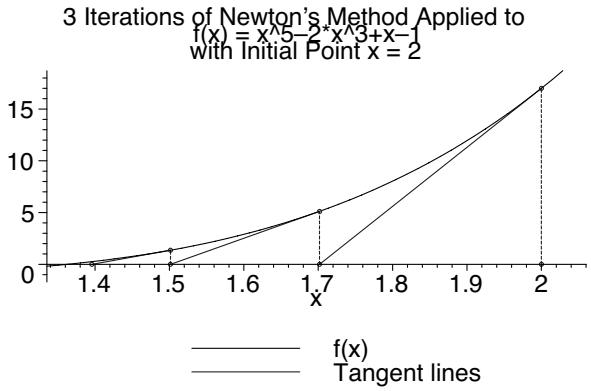
6.5.2 The `NewtonsMethod` procedure of Maple 8

The `Student[Calculus1]` package of Maple 8 contains `NewtonsMethod`, a procedure that applies Newton's method a specified number of times (5 by default), given a function and starting value. The syntax is

`NewtonsMethod(` $f(x)$`,` x`=`*StartingValue*`,` *options* `)`

The option `output=sequence` produces a sequence that includes the starting value and the successive approximations. The default `output=value` returns just the final approximation. The option `output=plot` returns a plot that shows the same kinds of things as in the previous demonstration. For example,

```
> with( Student[Calculus1] ):
> NewtonsMethod( x^5 - 2*x^3 + x - 1, x=2, iterations=3,
  output=plot );
```



where we have used the `iterations` option to specify that Newton's method be applied 3 times.

Since this plot contains most of the same objects as our previous demonstration, we can make a similar animation letting the `NewtonsMethod` procedure generate the objects for us. The principal difference is that `NewtonsMethod` combines into one frame the lines that we have called *VertLine* and *TangentLine* and plotted separately. We begin by loading the packages we need and choosing some options.

```
> restart:
> with( Student[Calculus1] ):
> with( plots ):
> setoptions( thickness=2, labels=["",""],
  axesfont=[HELVETICA,16] ):
```

For comparison, we will use the same example as in the previous demonstration.

```
> f := x -> 1/5*x^2 - x*cos(2*x);
> a := -1:
> b := 2:
> StartingValue := 1.5;
```

$$f := x \to \frac{1}{5}x^2 - x\cos(2\,x)$$
$$StartingValue := 1.5$$

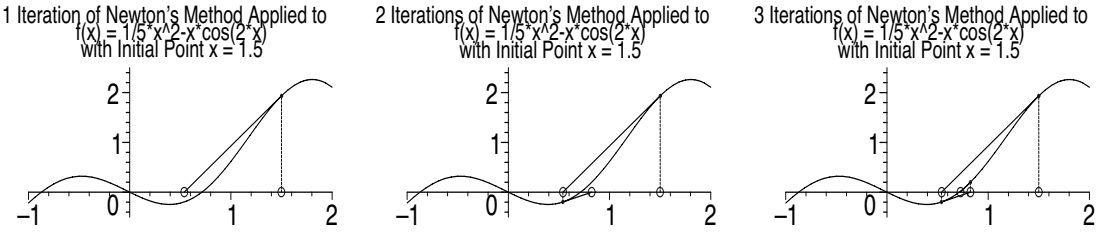
Next, we create a function N which will accept an argument i and generate a plot that shows the geometry when Newton's method is applied i times.

```
> N := i -> NewtonsMethod( f(x), x=StartingValue,
  iterations=i, output=plot, pointoptions=[symbolsize=16],
  view=[a..b,-.5..2.5], titlefont=[HELVETICA,14] ):
```

Here, we have used the `pointoptions`, `view`, and `titlefont` options to render the plot suitable for screen projection.

Finally, we display a sequence of these plots.

```
> display( seq( N(i), i=1..3 ), insequence=true );
```



6.5.3 Maple 8 demonstrations: Experimenting with Newton's method

The ability of the `NewtonsMethod` procedure to generate efficiently a plot that illustrates the geometry of Newton's method offers some interesting possibilities. For example, how does the geometry change as the starting value varies? We create an animation now that demonstrates this behavior. We will use as an example the function $f(x) = x^3 + 3x^2 + x - 1$, and we'll apply Newton's method 5 times for 25 different starting values (after the first one) from -3 to 1.

```
> restart:
> with( Student[Calculus1] ):
> with( plots ):
> setoptions( thickness=2, labels=["",""],
  axesfont=[HELVETICA,16] ):
> f := x -> x^3 + 3*x^2 + x - 1;
> a := -4:
> b := 2:
> NumIterations := 5:
> NumStartValues := 25:
> LowStartValue := -3:
> HighStartValue := 1:
```

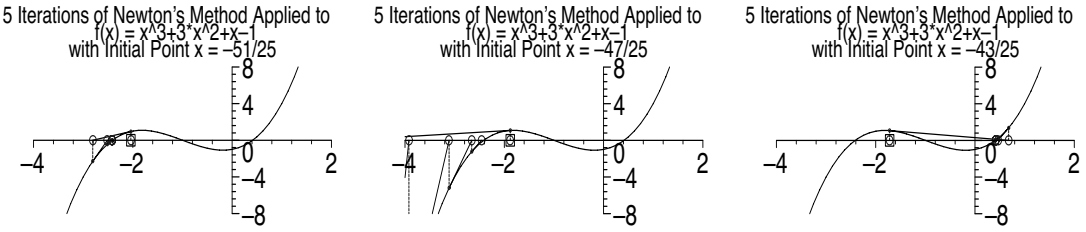
$$f := x \to x^3 + 3\,x^2 + x - 1$$

We define two functions: *StartingValue* will be used to create the various starting values from *LowStartValue* to *HighStartValue*. *N* will generate a plot showing Newton's method applied *NumIterations* times using starting value *StartingValue*(i).

```
> StartingValue := i -> LowStartValue +
  (HighStartValue-LowStartValue)/NumStartValues*i:
> N := i -> NewtonsMethod( f(x), x=StartingValue(i),
  iterations=NumIterations, output=plot,
  pointoptions=[symbolsize=16],  view=[a..b,-8..8],
  titlefont=[HELVETICA,14] ):
```

To make the starting point distinctive in the plot, we will plot it as a red box on the x-axis. We create a sequence of such plots and another sequence of the `NewtonsMethod` plots generated by N, then display them both.

```
> StartingPoint := display( seq( pointplot(
  [StartingValue(i),0], color=red, symbol=box,
  symbolsize=20 ), i=0..NumStartValues ), insequence=true ):
> NewtMeth := display( seq( N(i), i=0..NumStartValues ),
  insequence=true ):
> display( StartingPoint, NewtMeth );
```



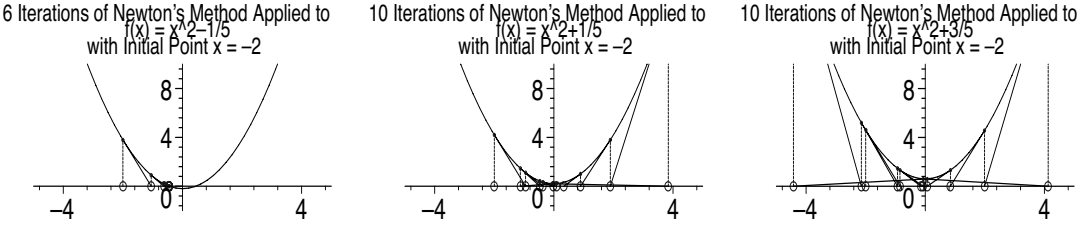
It would also be interesting to watch the behavior of Newton's method as it is applied to the function $f(x) = x^2 + c$, when c takes on various values, particularly positive ones.[1] We begin by choosing some options and creating a function f that permits us to vary c conveniently.

```
> restart:
> with( Student[Calculus1] ):
> with( plots ):
> setoptions( thickness=2, labels=["",""],
  axesfont=[HELVETICA,16] ):
> f := c -> x^2 + c;
```

$$f := c \to x^2 + c$$

We adapt the function N to apply Newton's method 10 times to $x^2 + c$ for varying values of c, using -2 as a starting value. Then we display a sequence of plots of Newton's method, generated by the function N.

```
> N := i -> NewtonsMethod( f(i/10), x=-2, iterations=10,
  output=plot, pointoptions=[symbolsize=16],
  view=[-5..5,-2..10], titlefont=[HELVETICA,14] ):
> display( seq( N(i), i=-10..20 ), insequence=true );
```



Letting i vary from -10 to 20 in $N(i)$ creates plots of Newton's method applied to $x^2 + c$ for values of c from -1 to 2. When c takes on positive values, Newton's method exhibits chaotic behavior.

[1] Thanks to Robert Devaney of Boston University for showing this example to me.

6.6 Demonstrations: Solids of revolution

Another topic from calculus, longing to be animated, is that of solids of revolution. We will create animations that show vividly what we can only describe in class: a solid being generated as a planar region revolves about an axis.

6.6.1 Revolving a region about the vertical axis

We begin with solids formed by revolution about the vertical axis. The first frame should show the region in the plane so that we can study its shape before it begins to move, then we will show it sweeping out the solid as it revolves. We will use as our example the region bounded by $y = -(x-1)^2 + x$, $y = \sqrt[3]{x} - 3$, $x = 1/2$, and $x = 3$. We begin by defining the two functions and the constants a and b.

```
> restart:
> with( plots ):
> setoptions3d( axes=normal, tickmarks=[0,5,5],
  axesfont=[HELVETICA,18], shading=zhue ):
> f := x -> -(x-1)^2 + x; # f is the upper bounding function
> g := x -> x^(1/3) - 3;  # g is the lower bounding function
> a := 1/2:
> b := 3:
```

$$f := x \to -(x-1)^2 + x$$
$$g := x \to x^{(1/3)} - 3$$

Recall (Section 1.3) that the `#` character allows a comment; Maple ignores everything after it on the same command line.

Next, we initialize the frame sequence with a frame that contains a plot of the region to be revolved. Since we will be revolving about the vertical axis and plotting in three dimensions, we will let the z-axis play the role of the axis of revolution. This arrangement seems perfect for cylindrical coordinates (r, θ, z), so we'll use `cylinderplot` (in parametric form) from the `plots` package.

```
> FrameSeq := cylinderplot( [r, Pi/2, z], r=a..b,
  z=g(r)..f(r), style=patchnogrid, labels=["","",""],
  color=blue, numpoints=100 ):
> NumFrames := 12:
```

In setting $\theta = \pi/2$, we have chosen the y-axis of $\mathbb{R}^3$ to play the role of the horizontal axis of $\mathbb{R}^2$. It is because the y- and z-axes will be representing in $\mathbb{R}^3$ the horizontal and vertical axes, respectively, of $\mathbb{R}^2$ that we used no tick

marks on the x-axis in the `setoptions3d` statement. This helps to make the other two axes more prominent in the plot. We have also chosen to use no axis labels; they sometimes don't appear where we would like them to be. Using 12 frames seems to make the animation smooth enough in this case.

We use a `for` loop to create the other frames. First, we need to compute an angle T, which will be the cylindrical coordinate θ of the region as it revolves about the z-axis. We will have a frame every $\Delta\theta = 2\pi/NumFrames$ radians, so the angle T will be $\pi/2$ (the region's initial position) decremented by a multiple of $\Delta\theta$. The choice of subtracting, rather than adding, multiples of $\Delta\theta$ means that the region will appear to revolve toward, instead of away from, the viewer. We then plot and store the parts that will constitute a frame, each depending on T. We will need the region in its new position. As shown in

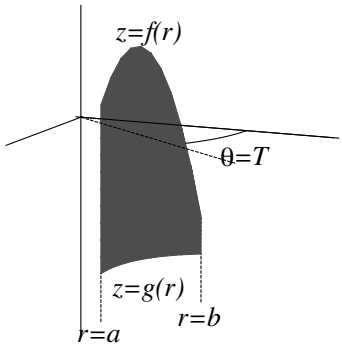


FIGURE 6.4: Region revolved through $\theta = T$ radians about the z-axis of $\mathbb{R}^3$

Figure 6.4, the region spans r from a to b and z from $g(r)$ to $f(r)$, but θ is constant at T. We will also need the top, bottom, inside, and outside of the partially formed solid. As shown in Figure 6.5, the top and bottom also span r from a to b with z determined by $f(r)$ and $g(r)$, respectively, and they span θ from $\pi/2$ to T. For the inside, r is constant at a, z runs from $g(a)$ to $f(a)$, and θ runs from $\pi/2$ to T. The outside is similar. Having generated the parts of a frame, we display them together and append the frame to the frame sequence.

```
> for i from 1 to NumFrames do
>   T := Pi/2 - 2*Pi*i/NumFrames;
>   Region := cylinderplot( [r, T, z], r=a..b, z=g(r)..f(r),
    style=patchnogrid, color=blue, numpoints=100 );
>   Top := cylinderplot( [r, theta, f(r)], r=a..b,
    theta=Pi/2..T, numpoints=50*i );
>   Bottom := cylinderplot( [r, theta, g(r)], r=a..b,
    theta=Pi/2..T, numpoints=50*i );
```

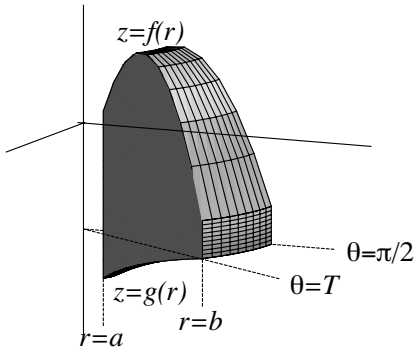


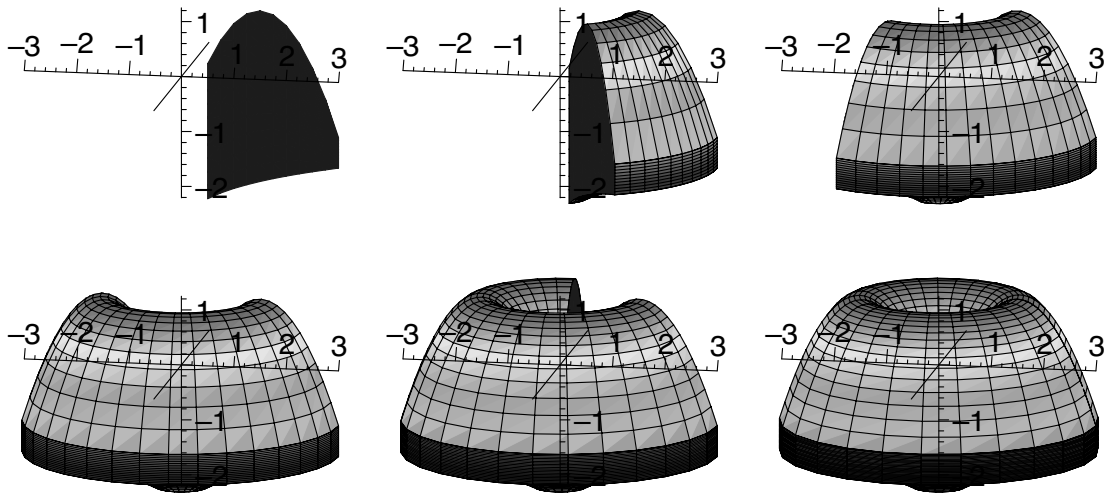
FIGURE 6.5: Partially formed solid of revolution about the z-axis of $\mathbb{R}^3$

```
>   Inside := cylinderplot( [a, theta, z], z=g(a)..f(a),
    theta=Pi/2..T, numpoints=50*i );
>   Outside := cylinderplot( [b, theta, z], z=g(b)..f(b),
    theta=Pi/2..T, numpoints=50*i );
>   FrameSeq := FrameSeq, display( Region, Top, Bottom,
    Inside, Outside )
>   end do:
```

The `numpoints=50*i` option was chosen in the interest of economy. The default value of 625 is larger than necessary when the partially formed solid is in its early stages. Letting `numpoints` vary directly as i allows the number of points used in the plot to grow as the solid forms.

Finally, we display the frames in sequence, with an orientation that seems to work well.

```
> display( FrameSeq, insequence=true, orientation=[10,70] );
```



6.6.2 Revolving a region about the horizontal axis

We turn now to solids generated by revolving a region about the horizontal axis. As an example, we will use the region bounded by $y = -(x-2)^2 + 3$, $y = x/3$, $x = 1$, and $x = 3$. We begin in the usual way.

```
> restart:
> with( plots ):
> setoptions3d( axes=normal, tickmarks=[0,5,5],
  axesfont=[HELVETICA,18], shading=zhue ):
> f := x -> -(x-2)^2 + 3; # f is the upper bounding function
> g := x -> x/3;          # g is the lower bounding function
> a := 1:
> b := 3:
```

$$f := x \to -(x-2)^2 + 3$$
$$g := x \to \frac{1}{3}\,x$$

This time, we will use the y-axis of $\mathbb{R}^3$ as the axis of revolution. The usual cylindrical coordinates, then, are not so well-suited as they were when the axis of revolution was z. We do have an analogous situation, however, so our method should work, with some adaptations. One way to do this is to create our own cylindrical coordinate system with the roles of the z-axis and y-axis interchanged. That is,

$$\begin{aligned} x &= r\cos\theta \\ y &= y \\ z &= r\sin\theta \end{aligned}$$

so that r and θ are just polar coordinates for the xz-plane.

Again, we need to initialize the frame sequence and choose a number of frames.

```
> FrameSeq := plot3d( [0, y, r], y=a..b, r=g(y)..f(y),
  style=patchnogrid, labels=["","",""], color=blue,
  numpoints=100 ):
> NumFrames := 12:
```

And again, we need to plot and store the components of a frame, this time the left and right sides, and the inside and outside. The components are created in an analogous way to those of the solid of revolution about the vertical axis. For example, the region (see Figure 6.6) spans y from a to b and r from $g(y)$ to $f(y)$, but θ is constant at T.

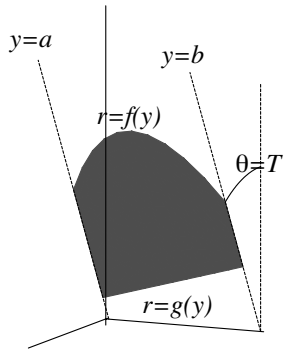
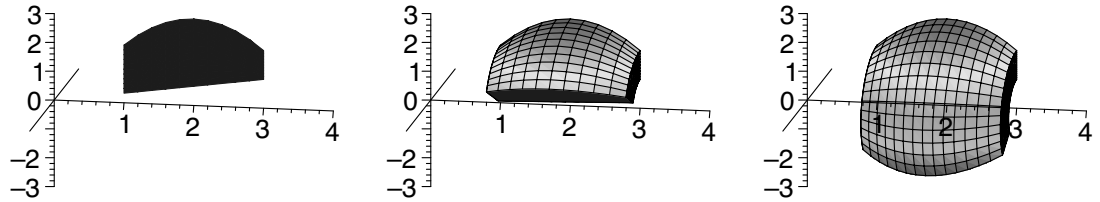


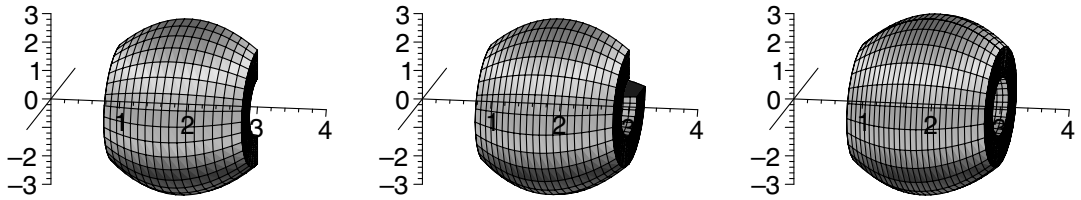
FIGURE 6.6: Region revolved through $\theta = T$ radians about the y-axis of $\mathbb{R}^3$

```
> for i from 1 to NumFrames do
>   T := Pi/2 - 2*Pi*i/NumFrames;
>   Region := plot3d( [r*cos(T), y, r*sin(T)], y=a..b,
    r=g(y)..f(y), style=patchnogrid, color=blue,
    numpoints=100 );
>   LeftSide := plot3d( [r*cos(theta), a, r*sin(theta)],
    r=g(a)..f(a), theta=Pi/2..T, numpoints=50*i );
>   RightSide := plot3d( [r*cos(theta), b, r*sin(theta)],
    r=g(b)..f(b), theta=Pi/2..T, numpoints=50*i );
>   Inside := plot3d( [g(y)*cos(theta), y, g(y)*sin(theta)],
    y=a..b, theta=Pi/2..T, numpoints=50*i );
>   Outside := plot3d( [f(y)*cos(theta), y,
    f(y)*sin(theta)], y=a..b, theta=Pi/2..T,
    numpoints=50*i );
>   FrameSeq := FrameSeq, display( Region, LeftSide,
    RightSide, Inside, Outside )
>   end do:
```

Finally, we display the results.

```
> Max := maximize( f(x), x=a..b ):
> display( FrameSeq, insequence=true, orientation=[10,70],
  view=[-Max..Max,(a-1)..(b+1),-Max..Max] );
```





The `maximize` procedure computes the maximum value of f on $[a, b]$. The specification of the `view` option, using this value, improves the appearance of the plot some. It's useful to know that `maximize` is available. There is also a `minimize`.

I use these demonstrations when I first introduce the idea of solids of revolution to ensure that everyone has a clear mental picture of the type of object we are going to be studying. I also use them to illustrate examples that I work in class, changing f, g, a, and b in the worksheet to conform to the given information in the example. When students ask about assigned problems, it is useful to have these demonstrations on hand. Again changing the particulars, f, g, a, and b, in the worksheet to match the details of the problem, I use the first frame to check that we have the region sketched correctly. Then I run the animation to confirm, or not, that the solid has the shape the students thought it would.

6.7 Demonstrations: Surfaces of revolution

By modifying the demonstrations of Section 6.6 that show a revolving planar region generating a solid, we can create demonstrations that illustrate a revolving planar curve generating a surface. The ideas are the same, but the frames are less elaborate because their components are fewer. To keep things simple, we will restrict ourselves to curves where y is a function of x, as these will be sufficient to illustrate the geometry of surfaces of revolution.

6.7.1 Revolving a curve about the vertical axis

First, we adapt the demonstration of a solid of revolution about the vertical axis (Section 6.6.1) to illustrate instead a surface of revolution. We begin the same way. As an example, we will use the curve defined by $f(x) = (1/2)\ln x$ on $[1, e]$.

```
> restart:
> with( plots ):
> setoptions3d( axes=normal, tickmarks=[0,5,5],
  axesfont=[HELVETICA,18], shading=zhue ):
```

```
> f := x -> ln(x)/2;
> a := 1:
> b := exp(1):
```

$$f := x \to \frac{1}{2}\ln(x)$$

Recall (Section 1.5) that e is represented in Maple as `exp(1)`.

We initialize the frame sequence with a plot of the curve in the yz-plane. Although there are other ways to represent it, we can make a minor change to the code in the original demonstration and think of it as the trace in the plane $\theta = \pi/2$ of the surface $z = f(r, \theta)$ for $r \in [a, b]$ in cylindrical coordinates.

```
> FrameSeq := cylinderplot( [r, Pi/2, z], r=a..b,
  z=f(r)..f(r), thickness=2, labels=["","",""],
  numpoints=200 ):
> NumFrames := 12:
```

In place of the `style=patchnogrid` option, we have used `thickness=2` to make the curve show up well, and we've increased `numpoints` to 200 to make it smooth.

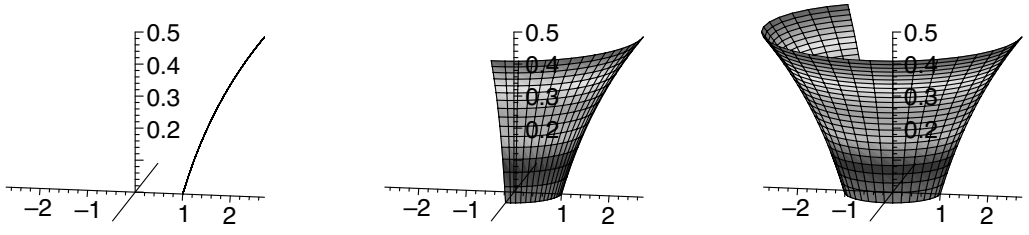
We won't need many of the elements of the loop body from the solid of revolution demonstration. We still need to compute the angle T of revolution, but this time we only need the plot that we called *Top*. We append it to the frame sequence.

```
> for i from 1 to NumFrames do
>   T := Pi/2 - 2*Pi*i/NumFrames;
>   FrameSeq := FrameSeq, cylinderplot( [r, theta, f(r)],
    r=a..b, theta=Pi/2..T, numpoints=70*i )
>   end do:
```

Here, we have increased `numpoints` to $70i$ for a smoother surface.

Finally, we display the frames in order.

```
> display( FrameSeq, insequence=true, orientation=[10,70] );
```



6.7.2 Revolving a curve about the horizontal axis

We readily adapt the demonstration of a solid of revolution about the horizontal axis (Section 6.6.2) for a surface of revolution. For our example function, we use $f(x) = x^3$ on $[0, 2]$.

```
> restart:
> with( plots ):
> setoptions3d( axes=normal, tickmarks=[0,5,5],
  axesfont=[HELVETICA,18], shading=zhue ):
> f := x -> x^3;
> a := 0:
> b := 2:
```

$$f := x \to x^3$$

Again, we initialize the frame sequence with a plot of the curve in the yz-plane. Using the same coordinate system as in Section 6.6.2, we regard the curve as the trace of the surface in the plane $x = 0$.

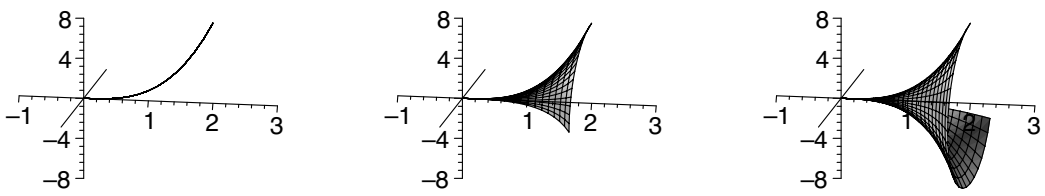
```
> FrameSeq := plot3d( [0, y, r], y=a..b, r=f(y)..f(y),
  thickness=2, labels=["","",""], numpoints=200 ):
> NumFrames := 12:
```

Of the statements in the loop body, we need the one that computes the angle T of revolution, and we need one to append the plot that we called *Outside* to the frame sequence.

```
> for i from 1 to NumFrames do
>   T := Pi/2 - 2*Pi*i/NumFrames;
>   FrameSeq := FrameSeq,
    plot3d( [f(y)*cos(theta), y, f(y)*sin(theta)], y=a..b,
    theta=Pi/2..T, numpoints=70*i )
>   end do:
```

Last, we display the results, again using `maximize`.

```
> Max := maximize( f(x), x=a..b ):
> display( FrameSeq, insequence=true, orientation=[10,70],
  view=[-Max..Max,(a-1)..(b+1),-Max..Max] );
```



Chapter 7

Adding Text to Animations

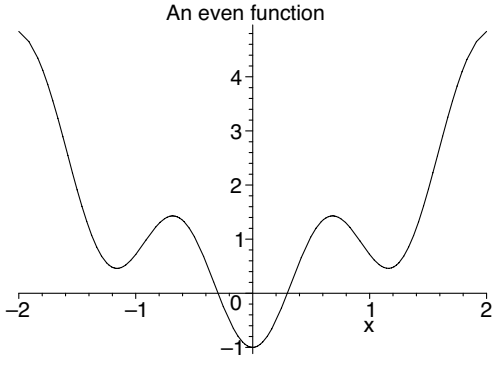
Some of our demonstrations may have seemed somewhat incomplete for their lack of labeling. It is often important to include notation in plots. In fact, sometimes it is the connection between the notation and the geometry that really constitutes the essence of the idea you are demonstrating. For example, think of the concept of choosing two values, a and $a+h$, evaluating a function at both a and $a+h$, then letting $h \to 0$. You want to demonstrate that what this accomplishes geometrically is the movement of one point along the x-axis toward another and, at the same time, the movement of an associated point along a curve toward another. You are really explaining the notation; you need to have it in the plot.

In this chapter, you will learn how to include text in plots, how to make it move, and how to include the results of computations. We will create an animation to show how Taylor polynomials of increasing degree can approximate a function increasingly well, then experiment by moving the center. We will also improve the demonstrations we made in Section 4.3 of secant and tangent lines and in Section 5.5.1 of rectangles approximating a definite integral.

7.1 Titles

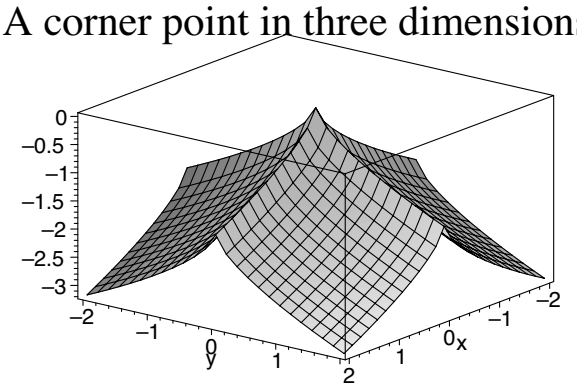
Both two-dimensional and three-dimensional plots may be given titles using the `title="`*text*`"` option, where `"`*text*`"` is a string. Recall from Section 2.12 that a *string* consists of any characters within double quotes. For example,

```
> plot( x^2 - cos(5*x), x=-2..2, title="An even function" );
```



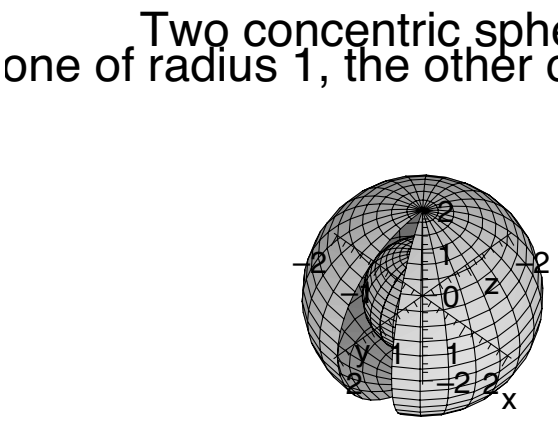
A title's font may be specified by using the `titlefont` option. The specification has the same syntax as that of the other font options, `font`, `axesfont`, and `labelfont` (see Section 2.12). For example,

```
> plot3d( -(x^2)^(1/3) - (y^2)^(1/3), x=-2..2, y=-2..2,
  axes=boxed, title="A corner point in three dimensions",
  titlefont=[TIMES,ROMAN,24] );
```



If you would like the title to appear on two or more lines, just include the characters `\n` in the string for each line break. For example,

```
> with( plots ):
> S1 := sphereplot( 1, theta=0..2*Pi, phi=0..Pi ):
> S2 := sphereplot( 2, theta=Pi/6..2*Pi, phi=0..Pi ):
> display( S1, S2, scaling=constrained, axes=normal,
  title="Two concentric spheres:\none of radius 1, the other
  of radius 2", titlefont=[HELVETICA,20] );
```



Another way to force a line break is to include a *shift-enter* or *shift-return* in the string.

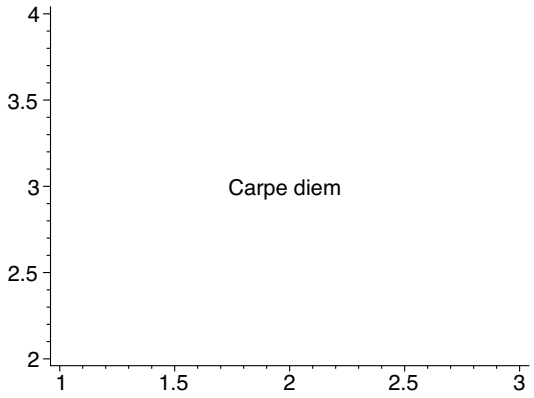
7.2 The `textplot` and `textplot3d` procedures

For two-dimensional plots, the procedure `textplot` within the `plots` package is designed to, well, plot text. The information about what text to plot and where to put it is passed to the procedure as a list. The syntax is

`textplot( [`*x*`,`*y*`,"`*text*`"], `*options*` )`

where the first two components are the coordinates of the position, and the third is the string to be plotted. For example,

```
> with( plots ):
> textplot( [2,3,"Carpe diem"] );
```

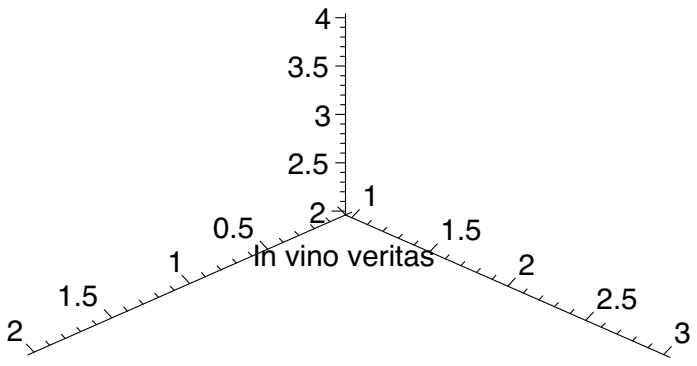


For three-dimensional plots, the procedure is `textplot3d`, and the syntax is, predictably,

`textplot3d( [`*x*`,`*y*`,`*z*`,"`*text*`"], `*options*` )`

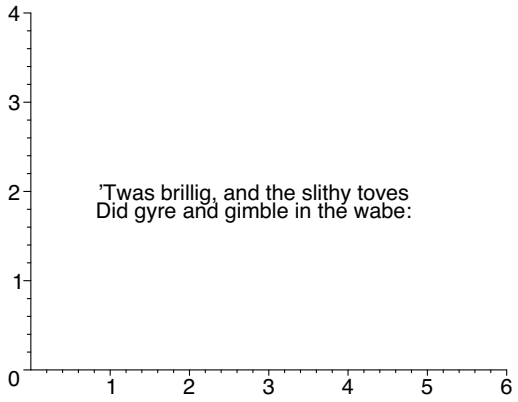
For example,

```
> with( plots ):
> textplot3d( [1,2,3,"In vino veritas"], axes=normal );
```



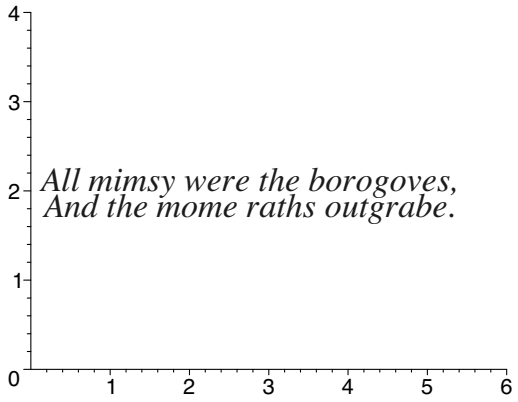
The plot structure resulting from a call to `textplot` or `textplot3d` can be stored and displayed later. For example, [2]

```
> with( plots ):
> T1 := textplot( [3,2,
  "'Twas brillig, and the slithy toves"] ):
> T2 := textplot( [3,1.8,
  "Did gyre and gimble in the wabe:"] ):
> display( T1, T2, view=[0..6,0..4] );
```



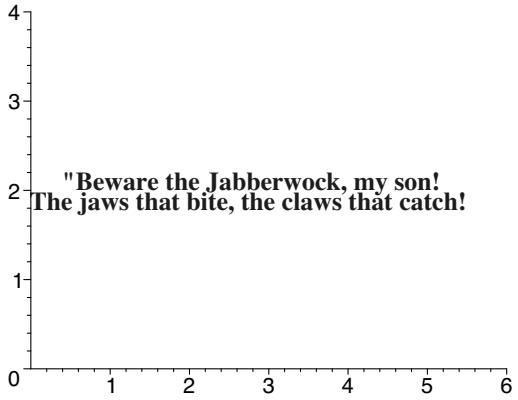
Both `textplot` and `textplot3d` accept the same options as `plot` and `plot3d`, respectively. An additional option, `align`, allows positioning the text relative to the point at which it is plotted. The choices are `ABOVE`, `BELOW`, `LEFT`, and `RIGHT`, where upper-case is required. Default is centered at the point. For example,

```
> with( plots ):
> T1 := textplot( [3,2,"All mimsy were the borogoves,"],
  align=ABOVE, color=blue, font=[TIMES,ITALIC,16] ):
> T2 := textplot( [3,2,"And the mome raths outgrabe."],
  align=BELOW, color=blue, font=[TIMES,ITALIC,16] ):
> display( T1, T2, view=[0..6,0..4] );
```



Incidentally, because strings are delimited with double quotes, some special provision is necessary if you want a double quote to be part of the string itself. One accommodation for this is a backslash (\) before the double quote. For example,

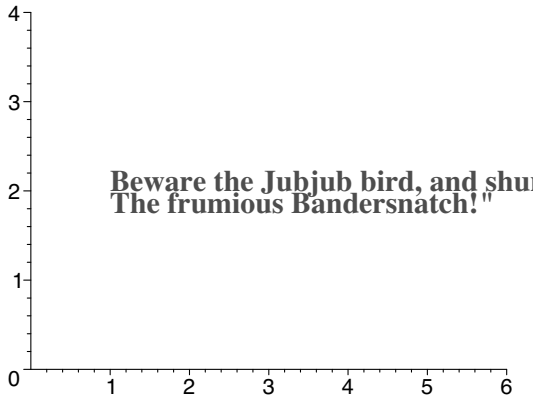
```
> with( plots ):
> T1 := textplot( [3,2,"\"Beware the Jabberwock, my son!"],
  align=ABOVE, color=blue, font=[TIMES,BOLD,14] ):
> T2 := textplot( [3,2,
  "The jaws that bite, the claws that catch!"], align=BELOW,
  color=blue, font=[TIMES,BOLD,14] ):
> display( T1, T2, view=[0..6,0..4] );
```



It also works to double the double quote.

The `align` option can be a set, allowing you to position the text, say, both above and to the right. For example,

```
> with( plots ):
> T1 := textplot( [1,2,"Beware the Jubjub bird, and shun"],
  align={ABOVE,RIGHT}, color=red, font=[TIMES,BOLD,14] ):
> T2 := textplot( [1,2,"The frumious Bandersnatch!\""],
  align={BELOW,RIGHT}, color=red, font=[TIMES,BOLD,14] ):
> display( T1, T2, view=[0..6,0..4] );
```

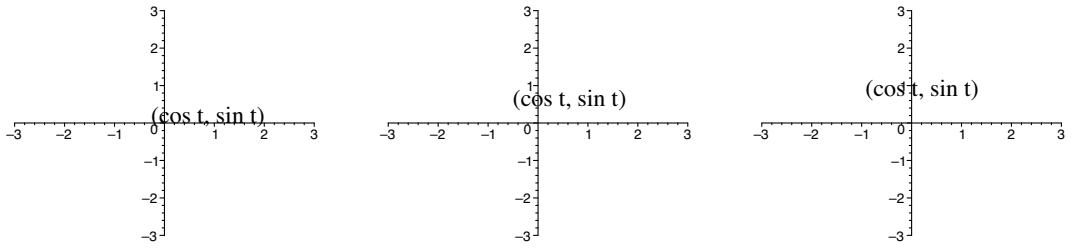


Notice that `RIGHT` means that the text will be plotted to the right of the point, not right-justified at the point.

7.3 Making text move

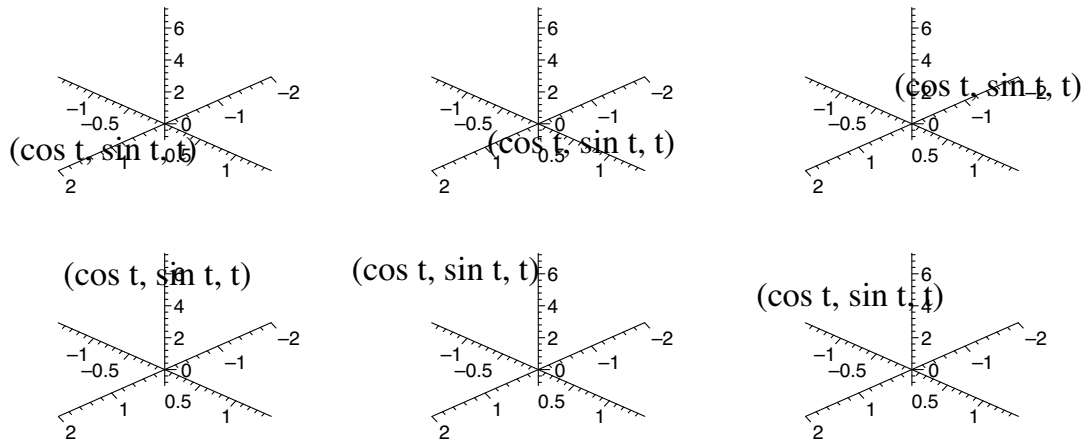
The text produced by `textplot` or `textplot3d` can be treated as any other plot structure. We can, therefore, create a sequence of `textplot`s or `textplot3d`s, then display them in sequence. For example, we can move text around a circle with

```
> with( plots ):
> MovingText := seq( textplot( [cos(2*Pi/50*i),
  sin(2*Pi/50*i),"(cos t, sin t)"], font=[TIMES,ROMAN,14] ),
  i=0..50 ):
> display( MovingText, insequence=true,
  view=[-3..3,-3..3] );
```



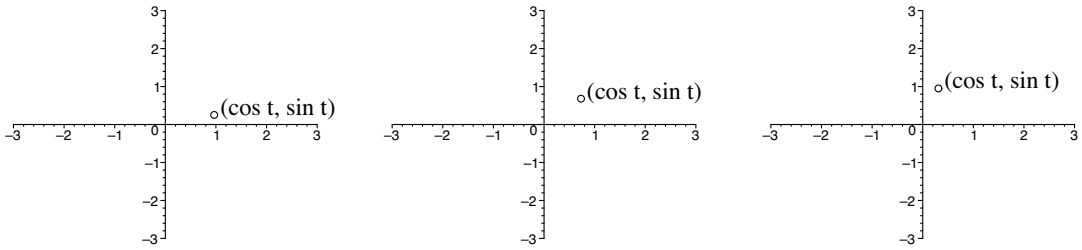
or along a helix with

```
> with( plots ):
> MovingText := seq( textplot3d( [cos(2*Pi/50*i),
  sin(2*Pi/50*i),2*Pi/50*i,"(cos t, sin t, t)"],
  font=[TIMES,ROMAN,14] ), i=0..50 ):
> display( MovingText, insequence=true, axes=normal );
```



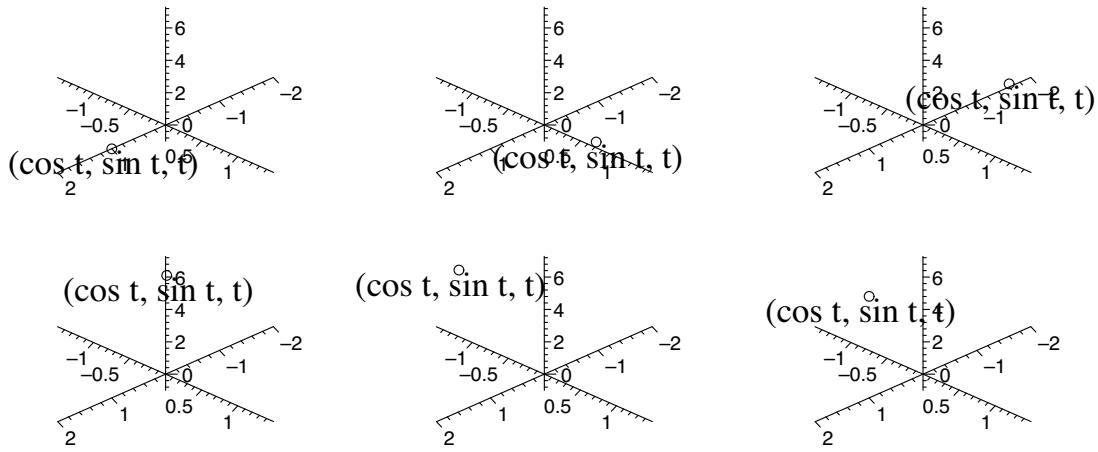
Often, we will want to attach text to a point as a label. We will need to move the label out of the way of the point symbol, though, by using `align`. If the point symbol and the label still collide, we can include in the label a leading or trailing space or two. For example,

```
> with( plots ):
> MovingPoint := display( seq( pointplot( [cos(2*Pi/50*i),
  sin(2*Pi/50*i)], symbol=circle, symbolsize=14 ),
  i=0..50 ), insequence=true ):
> MovingLabel := display( seq( textplot( [cos(2*Pi/50*i),
  sin(2*Pi/50*i)," (cos t, sin t)"], align={ABOVE,RIGHT},
  font=[TIMES,ROMAN,14] ), i=0..50 ), insequence=true ):
> display( MovingPoint, MovingLabel, view=[-3..3,-3..3] );
```



and

```
> with( plots ):
> MovingPoint := display( seq( pointplot3d( [cos(2*Pi/50*i),
  sin(2*Pi/50*i),2*Pi/50*i], symbol=circle, symbolsize=14 ),
  i=0..50 ), insequence=true ):
> MovingLabel := display( seq( textplot3d( [cos(2*Pi/50*i),
  sin(2*Pi/50*i),2*Pi/50*i,"(cos t, sin t, t)"],
  align={BELOW}, font=[TIMES,ROMAN,14] ), i=0..50 ),
  insequence=true ):
> display( MovingPoint, MovingLabel, axes=normal );
```



7.4 Demonstrations: Secant lines and tangent lines with labels

We return now to the demonstrations we made in Section 4.3 of secant lines and tangent lines and improve them. We will include labels, this time: a point labeled $a+h$ moving along the x-axis toward a point labeled a; a point labeled $(a+h, f(a+h))$ moving along the curve toward a point labeled $(a, f(a))$.

7.4.1 Secant lines at a point approaching a tangent line

We start in a similar way but expand the domain to accommodate the labels. If the axes move a bit when this animation runs, widening the plot window (by clicking and dragging one of the squares at the vertical sides of the window) will probably prevent this distraction. If not, the domain in the `plot` statement can be further expanded or the view can be expanded using the `view` option (Section 2.11).

```
> restart:
> with( plots ):
> setoptions( thickness=2, tickmarks=[0,0], labels=["",""],
  font=[TIMES,ITALIC,18], symbol=circle, symbolsize=14 ):
> f := x -> x^3 + 8;
> a := 2:
> Curve := plot( f(x), x=a-3.5..a+3.5, y=-10..50,
  color=black ):
```

$$f := x \to x^3 + 8$$

We now plot and store the fixed points $(a, 0)$ and $(a, f(a))$ together with their labels. We make a set to use for the secants that approach from the right and another for the secants that approach from the left. The differences being in alignment and in the inclusion of either leading or trailing spaces, both of which improve the appearance and prevent the labels of the fixed points from colliding with those of the moving points.

```
> RightFixedPts :=
  pointplot( {[a,0],[a,f(a)]}, color=red ),
  textplot( [a,0,"a  "], align={BELOW,LEFT} ),
  textplot( [a,f(a),"(a, f(a))  "], align={ABOVE,LEFT} ):
> LeftFixedPts :=
  pointplot( {[a,0],[a,f(a)]}, color=red ),
  textplot( [a,0,"  a"], align={BELOW,RIGHT} ),
  textplot( [a,f(a),"  (a, f(a))"], align={ABOVE,RIGHT} ):
```

Each of these is a sequence of three elements: the plot of the two points, the label for the point $(a, 0)$, and the label for the point $(a, f(a))$. We don't want these elements displayed in sequence, and they won't be. When we display them below, we will let the `insequence` option default to `false` so that the three elements of the sequence will be displayed together.

Next, we choose a number of frames for the animation and create a decreasing function h. We generate the two sets of secant lines in much the same manner as before.

```
> NumFrames := 20:
> h := i -> 1 - 0.99*i/NumFrames:
> RightSecants := animate( (f(a+h(i))-f(a))/h(i)*(x-a) +
  f(a), x=a-2..a+2, i=1..NumFrames, frames=NumFrames,
  color=blue ):
> LeftSecants := animate( (f(a-h(i))-f(a))/(-h(i))*(x-a) +
  f(a), x=a-2..a+2, i=1..NumFrames, frames=NumFrames,
  color=blue ):
```

To create the two sets of moving points and their labels, we have several elements to generate. We plot the points $(a + h, 0)$ and $(a + h, f(a + h))$ with a single call to `pointplot`. We plot the labels for these points with two separate calls to `textplot`. These elements need to be displayed together (with `insequence` defaulting to `false`); they constitute one frame. A frame, then, has the form

```
display( pointplot( point on the x-axis, point on the curve ),
         textplot( label for the x-axis point ),
         textplot( label for the curve point ) )                (*)
```

We need to create a sequence of these frames and display them in order. So the structure for the moving points has the form

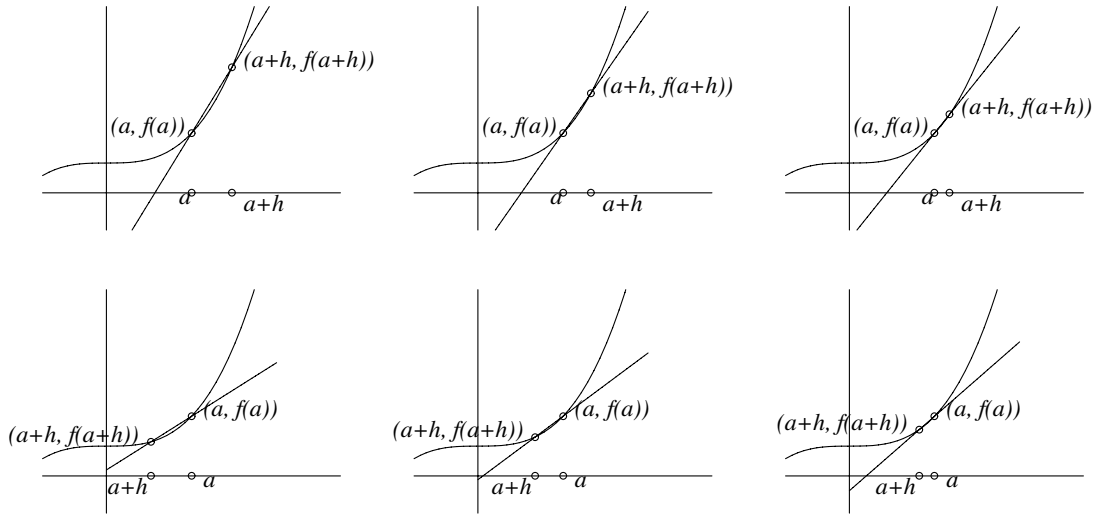
```
display( seq( frame_i, i=1..NumFrames ), insequence=true )
```

where *frame_i* has the form $(*)$.

```
> RightMovingPts := display( seq ( display(
  pointplot( {[a+h(i),0],[a+h(i),f(a+h(i))]}, color=red ),
  textplot( [a+h(i),0,"  a+h"], align={BELOW,RIGHT} ),
  textplot( [a+h(i),f(a+h(i)),"  (a+h, f(a+h))"],
  align={ABOVE,RIGHT} ) ), i=1..NumFrames ),
  insequence=true ):
> LeftMovingPts := display( seq ( display(
  pointplot( {[a-h(i),0],[a-h(i),f(a-h(i))]}, color=red ),
  textplot( [a-h(i),0,"a+h  "], align={BELOW,LEFT} ),
  textplot( [a-h(i),f(a-h(i)),"(a+h, f(a+h))  "],
  align={ABOVE,LEFT} ) ), i=1..NumFrames ),
  insequence=true ):
```

Finally, we display the background plot (the curve and the fixed points) and the animated elements (the moving points and the secants).

```
> display( Curve, RightFixedPts, RightMovingPts,
  RightSecants );
> display( Curve, LeftFixedPts, LeftMovingPts,
  LeftSecants );
```



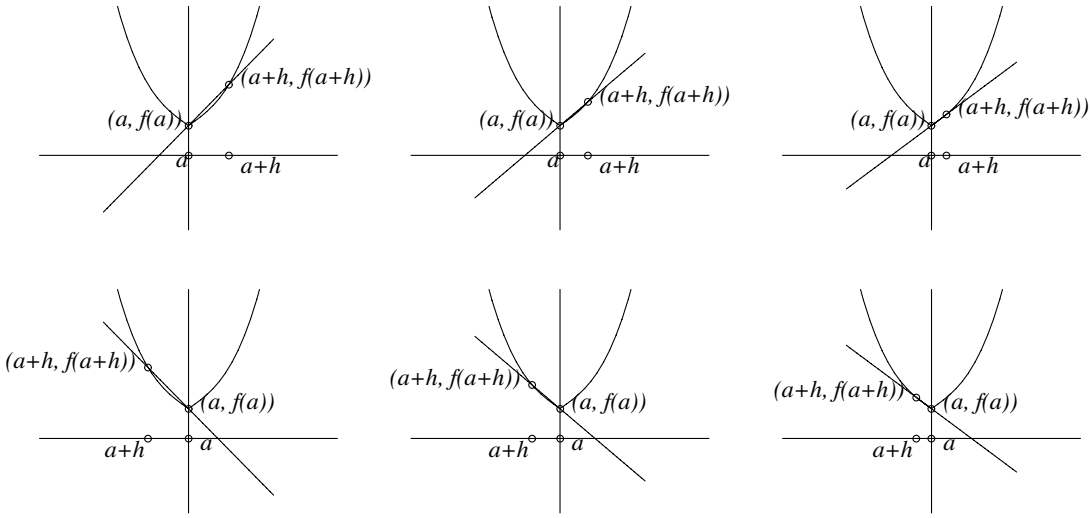
7.4.2 Secant lines at a corner point

So that we will also have on hand a demonstration, with notation, that a continuous function can fail to have a tangent line at a given point, we will change our example to $f(x) = |x^3 + 2x| + 2$ at $a = 0$.

```
> restart:
> with( plots ):
> setoptions( thickness=2, tickmarks=[0,0], labels=["","],
  font=[TIMES,ITALIC,18], symbol=circle, symbolsize=14 ):
> f := x -> abs( x^3 + 2*x ) + 2;
> a := 0:
> Curve := plot( f(x), x=a-3.5..a+3.5, y=-5..10,
  color=black ):
> RightFixedPts :=
  pointplot( {[a,0],[a,f(a)]}, color=red ),
  textplot( [a,0,"a  "], align={BELOW,LEFT} ),
  textplot( [a,f(a),"(a, f(a))  "], align={ABOVE,LEFT} ):
> LeftFixedPts :=
  pointplot( {[a,0],[a,f(a)]}, color=red ),
  textplot( [a,0,"  a"], align={BELOW,RIGHT} ),
  textplot( [a,f(a),"  (a, f(a))"], align={ABOVE,RIGHT} ):
```

```
> NumFrames := 20:
> h := i -> 1 - 0.99*i/NumFrames:
> RightSecants := animate( (f(a+h(i))-f(a))/h(i)*(x-a) +
  f(a), x=a-2..a+2, i=1..NumFrames, frames=NumFrames,
  color=blue ):
> LeftSecants := animate( (f(a-h(i))-f(a))/(-h(i))*(x-a) +
  f(a), x=a-2..a+2, i=1..NumFrames, frames=NumFrames,
  color=blue ):
> RightMovingPts := display( seq ( display(
  pointplot( {[a+h(i),0],[a+h(i),f(a+h(i))]}, color=red ),
  textplot( [a+h(i),0,"  a+h"], align={BELOW,RIGHT} ),
  textplot( [a+h(i),f(a+h(i)),"  (a+h,f(a+h))"],
  align={ABOVE,RIGHT} ) ), i=1..NumFrames ),
  insequence=true ):
> LeftMovingPts := display( seq ( display(
  pointplot( {[a-h(i),0],[a-h(i),f(a-h(i))]}, color=red ),
  textplot( [a-h(i),0,"a+h  "], align={BELOW,LEFT} ),
  textplot( [a-h(i),f(a-h(i)),"(a+h, f(a+h))  "],
  align={ABOVE,LEFT} ) ), i=1..NumFrames ),
  insequence=true ):
> display( Curve, RightFixedPts, RightMovingPts,
  RightSecants );
> display( Curve, LeftFixedPts, LeftMovingPts,
  LeftSecants );
```

$$f := x \to \left|x^3 + 2\,x\right| + 2$$



I use these demonstrations very early in first-term calculus when I discuss the distinction between a tangent line and a line that intersects a curve in a single point. I show both of these animations to illustrate the difference. Soon after, when I define the derivative, I use them again to illustrate geometrically what $\lim_{h\to 0}(f(x+h)-f(x))/h$ accomplishes analytically.

7.4.3 The `NewtonQuotient` procedure of Maple 8

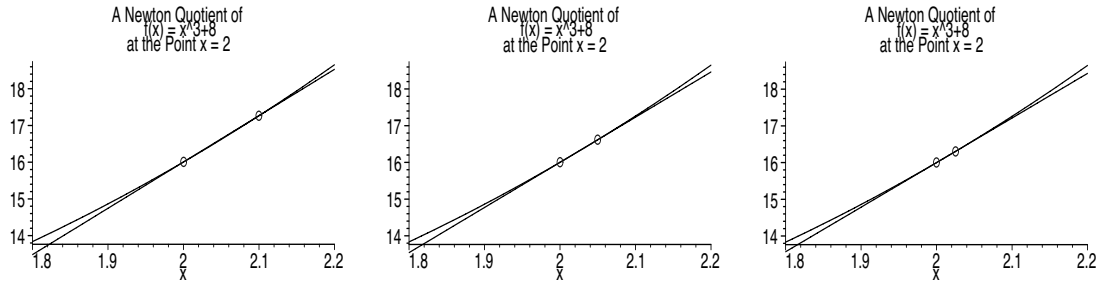
In the `Student[Calculus1]` package of Maple 8 is `NewtonQuotient`, a procedure that can produce an animation of secant lines. The animation does not include the notation, though, so, in this sense, it is more like the demonstrations of Section 4.3 than of this section. The syntax that yields an animation is

`NewtonQuotient(` $f(x)$`,` x`=`a`, output=animation,` *other options* `)`

showing secant lines for the function f at the point a. For example,

```
> with( Student[Calculus1] ):
> f := x -> x^3 + 8;
> a := 2:
> NewtonQuotient( f(x), x=a, output=animation );
```

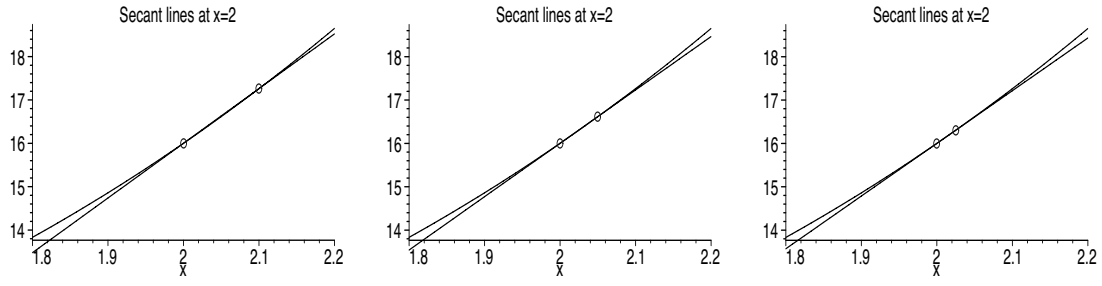
$$f := x \to x^3 + 8$$



You will probably want to change the title to reflect the content more accurately. For example,

```
> with( Student[Calculus1] ):
> f := x -> x^3 + 8;
> a := 2:
> NewtonQuotient( f(x), x=a, output=animation,
  title="Secant lines at x=2" );
```

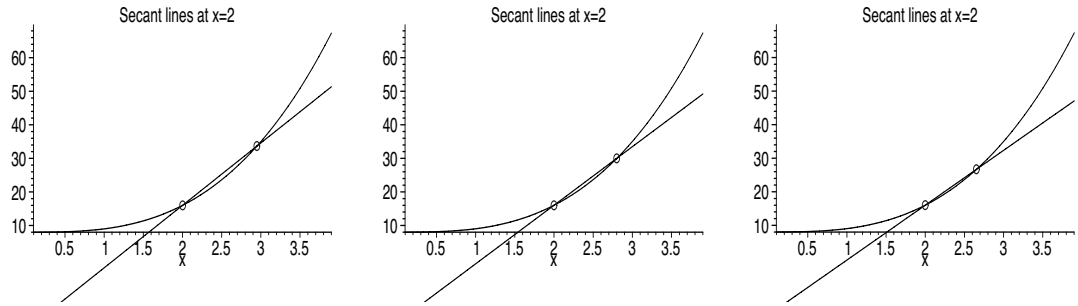
$$f := x \to x^3 + 8$$



You might have noticed that the moving point in this animation decelerates toward the fixed point. This is because, by default, the distance h (which has initial value 0.1) between these points is halved for each frame. A constant rate of approach can be arranged, if you prefer, by specifying that h take on the values in a list. For example,

```
> with( Student[Calculus1] ):
> f := x -> x^3 + 8;
> a := 2:
> NumFrames := 20:
> hList := [ seq( 1-0.99*i/NumFrames, i=1..NumFrames ) ]:
> NewtonQuotient( f(x), x=a, output=animation, h=hList,
  title="Secant lines at x=2" );
```

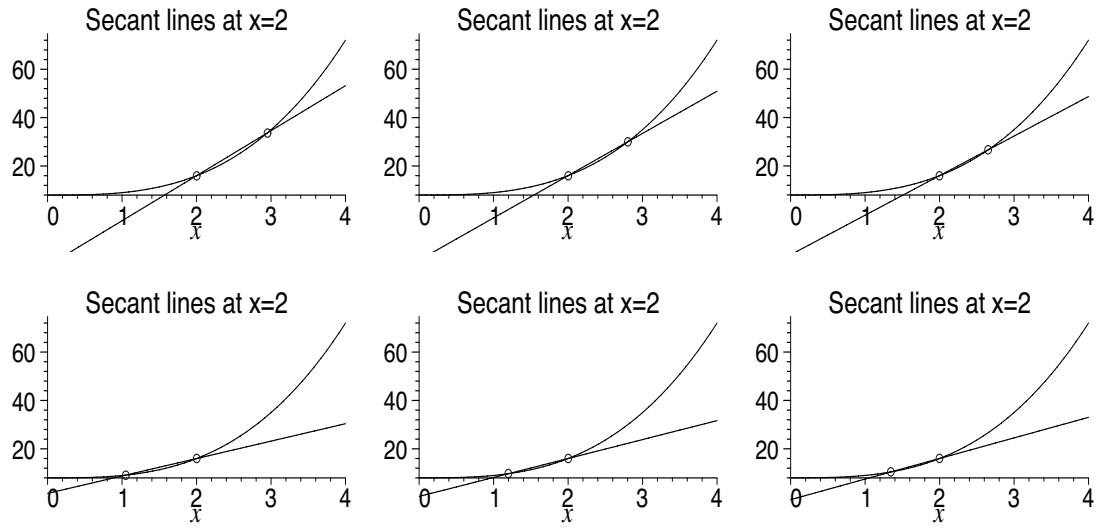
$$f := x \rightarrow x^3 + 8$$



So, we can use `NewtonQuotient` to make a demonstration similar to the unlabeled ones that we made in Section 4.3. We will create a list of positive h-values for the right-hand secant lines and a list of negative h-values for the left-hand ones. We will also increase the sizes of the objects and the fonts to improve the readability at a distance.

```
> restart:
> with( Student[Calculus1] ):
> f := x -> x^3 + 8;
> a := 2:
> NumFrames := 20:
> hListRight := [ seq( 1-0.99*i/NumFrames,
  i=1..NumFrames ) ]:
> NewtonQuotient( f(x), x=a, a-2..a+2, output=animation,
  h=hListRight, thickness=2, pointoptions=[symbolsize=14],
  axesfont=[HELVETICA,16], labelfont=[TIMES,ITALIC,18],
  title="Secant lines at x=2", titlefont=[HELVETICA,18] );
> hListLeft := [ seq( -1+0.99*i/NumFrames,
  i=1..NumFrames ) ]:
> NewtonQuotient( f(x), x=a, a-2..a+2, output=animation,
  h=hListLeft, thickness=2, pointoptions=[symbolsize=14],
  axesfont=[HELVETICA,16], labelfont=[TIMES,ITALIC,18],
  title="Secant lines at x=2", titlefont=[HELVETICA,18] );
```

$$f := x \to x^3 + 8$$



Here, we have used `a-2..a+2` to specify the domain, and `pointoptions`, one of the other options of the `NewtonQuotient` procedure, to increase the size of the plot symbol for the points.

As usual, you can access a full description of this procedure by typing `?NewtonQuotient` at the Maple prompt.

7.5 Including computed values in text

The procedure `sprintf` will convert a numerical value to a string. The syntax, as we will use `sprintf`, is

`sprintf( "`*format*`",` v_1`,` v_2`,...,` v_k `)`

where "*format*" is itself a string that specifies the format of the output string, and where $v_1, v_2, \ldots, v_k$ are the values to be converted. The `sprintf` procedure provides a great deal of control over format, and we will include some, but not all, of the details here. If this isn't enough for you, type `?sprintf` at the Maple prompt for the complete story. If, on the other hand, you just want to convert a numerical value to a string and do not particularly care precisely how it will be formatted, use the `%g` format described below.

The `%d` format is used for converting integer values to strings. For example,

```
> sprintf( "%d", 65 );
```

"65"

The format string can contain other characters. For example,

```
> sprintf( "the global maximum is %d", 10 );
> sprintf( "%d is the global maximum", 10 );
> sprintf( "(%d,%d) is a critical point", 3, 5 );
```

"the global maximum is 10"
"10 is the global maximum"
"(3,5) is a critical point"

If you try to convert a non-integer value using `%d`, you will get an error message, rather than a truncated or rounded value. There are, however, `trunc` and `round` procedures for doing that. For example,

```
> sprintf( "truncated = %d", trunc(25/7) );
> sprintf( "rounded = %d", round(25/7) );
```

"truncated = 3"
"rounded = 4"

The `%e` format is scientific notation for floating-point values. For example,

```
> sprintf( "e format is %e square meters", 69560345/7 );
```

"e format is 9.937192e+06 square meters"

The `%f` format is for fixed-point values. The default number of decimal places is 6. For example,

```
> sprintf( "f format is %f cubic meters", 245/11 );
```

"f format is 22.272727 cubic meters"

The `%g` format basically hands over control to Maple. Either `%d`, `%e`, or `%f` format is used, whichever is appropriate. For example,

```
> sprintf( "g format is %g", 12 );
> sprintf( "g format is %g", 8/3 );
> sprintf( "g format is %g", 34456*123.7 );
```

"g format is 12 "
"g format is 2.666667"
"g format is 4.262207e+06"

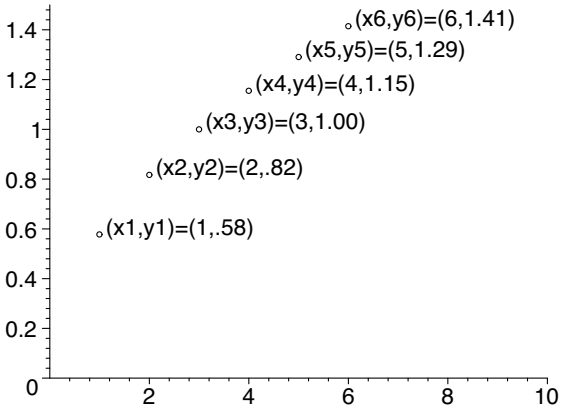
Any of these formats can be tailored further by including a specification of the form w, $.p$, or $w.p$ between the `%` sign and the code letter `d`, `e`, `f`, or `g`. Here, w (width) specifies the minimum number of characters to appear in the entire string, and p dictates the number of decimal places to the right of the decimal point. For example,

```
> sprintf( "%12d", 12345 );
> sprintf( "%12.3e", 12345 );
> sprintf( "%12.3f", 12345 );
> sprintf( "%12f", 12345 );
> sprintf( "%.3f", 12345 );
> sprintf( "%12.3g", 12345 );
```

```
"       12345"
"   1.235e+04"
"   12345.000"
"12345.000000"
"12345.000"
"   1.235e+04"
```

To include computed numerical values in labels, we combine the capability of `sprintf` to convert numerical values to strings with `textplot`'s or `textplot3d`'s ability to plot strings. For example,

```
> with( plots ):
> f := x -> sqrt( x/3 ):
> Points := seq( pointplot( [i,f(i)], symbol=circle,
  symbolsize=14, color=black ), i=1..6 ):
> Labels := seq( textplot(
  [i,f(i),sprintf( " (x%d,y%d)=(%d,%.2f)", i, i, i, f(i) )],
  align={ABOVE,RIGHT}, color=blue ), i=1..6 ):
> display( Points, Labels, view=[0..10,0..1.5] );
```



7.6 Demonstration: Rectangular approximation of the definite integral with annotation

We return now to the demonstration in Section 5.5.1 of the approximation of the definite integral and make some improvements. As a minor one, we

will add a title. We will also use `rightsum` to compute the sum of the areas of the rectangles in the approximation. We would like to display this area in the center of the plot, and, so that the y-axis doesn't interfere with it, we will use boxed axes. For the sake of variety, we'll also change the function and domain to $f(x) = 4 - x^2$ on $[-2, 2]$.

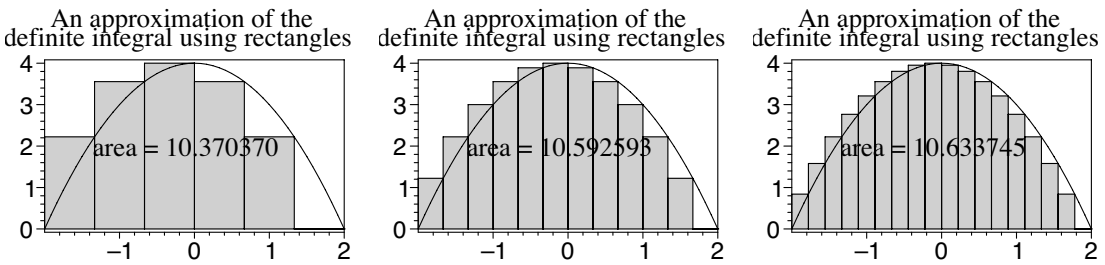
```
> restart:
> with( student ):
> with( plots ):
> setoptions( labels=["",""], axesfont=[HELVETICA,18],
  font=[TIMES,ROMAN,20], axes=boxed, title="An approximation
  of the\ndefinite integral using rectangles",
  titlefont=[TIMES,ROMAN,20] ):
> f := x -> 4 - x^2:
> a := -2:
> b := 2:
```

Now we find the center of the plot so that we can place the text for the area there. We first use `minimize` and `maximize` to find *f_min* and *f_max*. Then we compute a short sequence, *MidGraph*, of the coordinates of the center of the graph to use as the position in `textplot`.

```
> f_min := minimize( f(x), x=a..b ):
> f_max := maximize( f(x), x=a..b ):
> MidGraph := (a+b)/2, (f_min+f_max)/2:
```

Next, we generate and store the displays of two sequences. The first, *Rectangles*, is the sequence of `rightbox`s as before. The second, *Area*, is a sequence of `textplot`s. In the call to `textplot`, we use *MidGraph* as the position of the text, `rightsum` to compute the sum of the areas of the rectangles, and `sprintf` to convert the sum to a string. Finally, we display the two elements of the animation.

```
> Rectangles := display( seq( rightbox( f(x), x=a..b,
  NumRects ), NumRects=6..80 ), insequence=true ):
> Area := display( seq( textplot( [MidGraph,sprintf(
  "area = %f", rightsum( f(x), x=a..b, NumRects ) )] ),
  NumRects=6..80 ), insequence=true ):
> display( Rectangles, Area );
```



7.7 Constructing Taylor polynomials

We will discuss two methods for constructing Taylor polynomials. The first, which is valid in either Maple 7 or Maple 8, involves converting a partial sum of a Taylor series to a polynomial. The second is a convenient procedure built into Maple 8.

7.7.1 Taylor series and the `convert` procedure

The `taylor` procedure produces a truncated Taylor series including a term indicating the order of the remainder. The call

$$\texttt{taylor(}\ f(x)\texttt{, }x\texttt{=}a\texttt{, }n+1\ \texttt{)}$$

generates $\sum_{i=0}^{n}(f^{(i)}(a)/i!)(x-a)^i$ plus a term of order $n+1$. For example,

```
> taylor( 1/x, x=1, 5 );
> taylor( exp(x), x=0, 6 );
> taylor( sin(x), x=0, 8 );
> taylor( cos(x), x=0, 8 );
```

$$1-(x-1)+(x-1)^2-(x-1)^3+(x-1)^4+\mathrm{O}((x-1)^5)$$

$$1+x+\frac{1}{2}x^2+\frac{1}{6}x^3+\frac{1}{24}x^4+\frac{1}{120}x^5+\mathrm{O}(x^6)$$

$$x-\frac{1}{6}x^3+\frac{1}{120}x^5-\frac{1}{5040}x^7+\mathrm{O}(x^8)$$

$$1-\frac{1}{2}x^2+\frac{1}{24}x^4-\frac{1}{720}x^6+\mathrm{O}(x^8)$$

To get a Taylor polynomial, we need to remove the last term, and, to do that, we will use the `convert` procedure.

The `convert` procedure, which converts expressions from one form to another, is worth a look. It will do simple conversions such as

```
> convert( 7*Pi/6, degrees );
> convert( 120*degrees, radians );
```

$$210\ \mathit{degrees}$$

$$\frac{2\,\pi}{3}$$

but also such things as conversion to continued fractions (`confrac`)

```
> convert( exp(x), confrac, x );
```

$$1+\cfrac{x}{1+\cfrac{x}{-2+\cfrac{x}{-3+\cfrac{x}{2+\cfrac{x}{5}}}}}$$

and partial fraction decomposition (`parfrac`)

```
> R := (6*x^3 - 34*x^2 - 5*x - 29)/
  (3*x^4 - 13*x^3 + 13*x^2 - 39*x + 12 );
> convert( R, parfrac, x );
```

$$R := \frac{6\,x^3-34\,x^2-5\,x-29}{3\,x^4-13\,x^3+13\,x^2-39\,x+12}$$
$$-\frac{1}{x-4}+\frac{1+2\,x}{x^2+3}+\frac{3}{3\,x-1}$$

Type `?convert` at the Maple prompt for more details.

To create a Taylor polynomial, we need to convert a truncated series to a polynomial (`polynom`). Returning to the examples above, we have

```
> convert( taylor( 1/x, x=1, 5 ), polynom );
> convert( taylor( exp(x), x=0, 6 ), polynom );
> convert( taylor( sin(x), x=0, 8 ), polynom );
> convert( taylor( cos(x), x=0, 8 ), polynom );
```

$$2-x+(x-1)^2-(x-1)^3+(x-1)^4$$
$$1+x+\frac{1}{2}\,x^2+\frac{1}{6}\,x^3+\frac{1}{24}\,x^4+\frac{1}{120}\,x^5$$
$$x-\frac{1}{6}\,x^3+\frac{1}{120}\,x^5-\frac{1}{5040}\,x^7$$
$$1-\frac{1}{2}\,x^2+\frac{1}{24}\,x^4-\frac{1}{720}\,x^6$$

7.7.2 The `TaylorApproximation` procedure of Maple 8

Maple 8's `TaylorApproximation` procedure, in the `Student[Calculus1]` package, can produce a Taylor polynomial directly. The syntax is

`TaylorApproximation(` $f(x)$`,` x`=`a`, order=`n`,` *other options* `)`

which generates a Taylor polynomial of degree at most n about a for $f(x)$. Redoing the previous examples, we have

```
> with( Student[Calculus1] ):
> TaylorApproximation( 1/x, x=1, order=4 );
> TaylorApproximation( exp(x), x=0, order=5 );
> TaylorApproximation( sin(x), x=0, order=7 );
> TaylorApproximation( cos(x), x=0, order=7 );
```

$$5 - 10\,x + 10\,x^2 - 5\,x^3 + x^4$$

$$1 + x + \frac{1}{2}\,x^2 + \frac{1}{6}\,x^3 + \frac{1}{24}\,x^4 + \frac{1}{120}\,x^5$$

$$x - \frac{1}{6}\,x^3 + \frac{1}{120}\,x^5 - \frac{1}{5040}\,x^7$$

$$1 - \frac{1}{2}\,x^2 + \frac{1}{24}\,x^4 - \frac{1}{720}\,x^6$$

7.8 Demonstrations: Taylor polynomials

We now put together some of the above techniques to create a demonstration of the power of Taylor polynomials to approximate other functions near a point. The animation we have in mind will show the function to be approximated and, successively, Taylor polynomials of increasing degree. It will include annotation showing the degree of the approximating polynomial. We implement this in two ways. The first uses the `taylor` and `convert` procedures. The second, for Maple 8, uses the `TaylorApproximation` procedure.

7.8.1 Taylor polynomials of varying degree

First, we choose a function f to use as an example and a point a about which to create approximating polynomials. We also select a domain and range for the plot; a short sequence, *TextPoint*, to use for the point at which to display the degree; and the maximum degree, *MaxDegree*, for the Taylor polynomials.

```
> restart:
> with( plots ):
> setoptions( labels=["",""], font=[TIMES,ROMAN,24],
  thickness=2, axesfont=[HELVETICA,18] ):
> f := x -> sin(x):
> a := 0:
> Domain := -8..8:
> Range := -2..2:
> TextPoint := 2,1.5:
> MaxDegree := 20:
```

Next, we create the background plot of f. We also define a function T_i that creates a Taylor polynomial of degree at most i, and we store a display of a sequence of plots of T_i for values of i from 1 to *MaxDegree*.

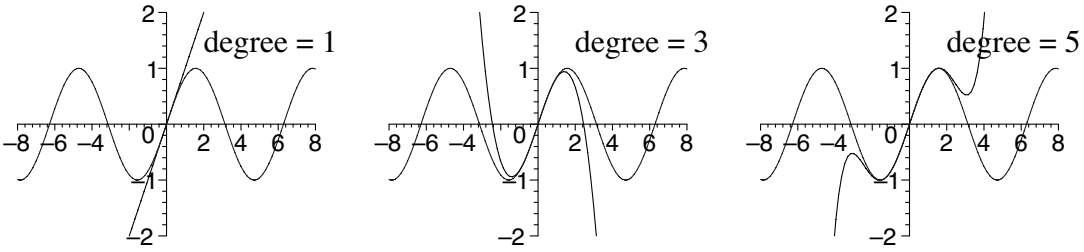
```
> fPlot := plot( f(x), x=Domain ):
> T := i -> convert( taylor( f(x), x=a, i+1 ), polynom ):
> TaylorPoly := display( seq( plot( T(i), x=Domain,
  numpoints=100, color=blue ), i=1..MaxDegree ),
  insequence=true ):
```

We now generate a sequence of `textplot`s to display the degree of the Taylor polynomial plotted in the frame. To do this, we use the procedure `degree`, which simply evaluates the degree of its argument, the polynomial T_i. So that this value is useable by `textplot`, we convert it to a string using `sprintf`.

```
> PolyDegree := display( seq( textplot(
  [TextPoint,sprintf( "degree = %d", degree(T(i)) )],
  align=RIGHT ), i=1..MaxDegree ), insequence=true ):
```

Finally, we display the background plot together with the animated elements, restricting the view using the selected range.

```
> display( fPlot, TaylorPoly, PolyDegree, view=Range );
```



I use this demonstration after I have developed the formula for the coefficients in a Taylor polynomial. I find that it is best used by stepping through the frames, one at a time. I explain that the first frame shows a simple approximating function, a linear one, that shares a point and first derivative with the given function. (The students will probably have seen this before in the context of linear approximations of functions.) Stepping to the next frame, nothing changes; this is a good opportunity to explain why, for the sine, pairs of successive Taylor polynomials are the same. Stepping to the next frame, I explain that this cubic polynomial not only shares a point with the given function, but also agrees at its first, second, and third derivatives. I proceed in this way, stepping one frame at a time. It is entertaining, however, to let the animation run.

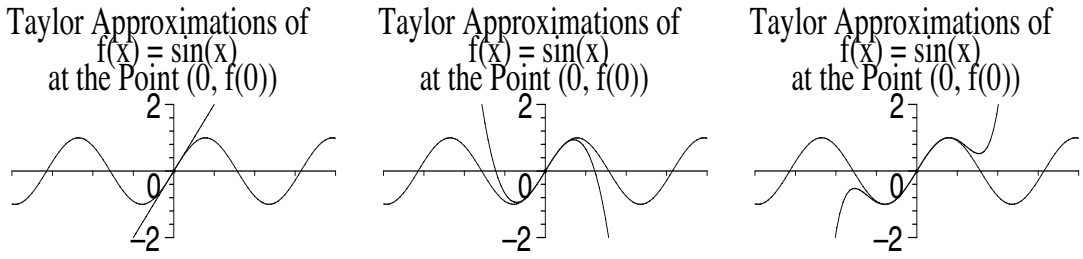
After this, it is easy to change the function, the value a, and possibly the domain and range to try some other examples. This is a good time to

encourage student participation by asking for suggestions, entering them into the worksheet, and then watching Taylor polynomials at work. I make sure that one of the examples we try is $\ln x$ at $a = 1$ (with a range of about $[-7, 7]$), and I ask the students to explain why the Taylor polynomials do such a poor job of approximating $\ln x$ beyond 2.

7.8.2 Maple 8 alternative using `TaylorApproximation`

Among the options for the `TaylorApproximation` procedure is `output`. The choices are `animation`, `plot`, and the default `polynomial`. The `output=animation` option can produce an animation very much like the one that we created above; we just specify a range for the `order` option. The difference is that the animation does not display the degree of the Taylor polynomial. For example,

```
> with( Student[Calculus1] ):
> TaylorApproximation( sin(x), x=0, order=1..20,
  output=animation, thickness=2, view=[-8..8,-2..2],
  labels=["",""], axesfont=[HELVETICA,18],
  titlefont=[TIMES,ROMAN,24] );
```



Here, we have also used the other options to adjust font size and curve thickness for a readable projected image.

This is certainly a quick and easy way to create an animation to demonstrate the concept. With a little more effort, we can include a display of the order of the truncated Taylor series (as opposed to the degree of the Taylor polynomial) used in the approximation. We begin in much the same way as our demonstration above.

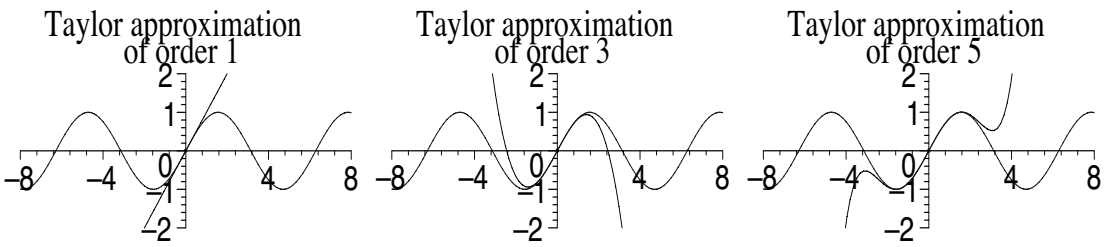
```
> restart:
> with( Student[Calculus1] ):
> with( plots ):
> f := x -> sin(x):
> a := 0:
> Domain := -8..8:
> Range := -2..2:
> MaxDegree := 20:
```

The heart of the code is a function T_i that generates a single plot (`output=plot`) of a Taylor approximation of order i. To accomplish the display of the order, we will include it in the title. We use the `title` option, which expects a string, together with `sprintf`, which produces one.

```
> T := i -> TaylorApproximation( sin(x), x=0, order=i,
  output=plot, thickness=2, view=[Domain,Range],
  labels=["",""], axesfont=[HELVETICA,18],
  title=sprintf( "Taylor approximation\nof order %d", i ),
  titlefont=[TIMES,ROMAN,24] ):
```

Finally, we display a sequence of these plots for i from 1 to *MaxDegree*.

```
> display( seq( T(i), i=1..MaxDegree ), insequence=true );
```



7.9 Demonstrations: Experimenting with Taylor polynomials

In the previous demonstrations, we fixed the value a at which we centered the Taylor polynomials and varied the degree. It would be interesting to see what happens if, instead, we were to fix the degree and vary a. Again, we will implement this in two ways: using `taylor` and `convert`, and, for Maple 8, using `TaylorApproximation`.

7.9.1 Taylor polynomials with varying center

We begin by setting the same options.

```
> restart:
> with( plots ):
> setoptions( labels=["",""], font=[TIMES,ROMAN,24],
  thickness=2, axesfont=[HELVETICA,18] ):
```

We use the sine function again as our example and fix the maximum degree of the Taylor polynomial at 5. We choose a lowest and highest value for a,

set the domain to include those values plus some on each side, and choose a number of frames and a suitable point at which to position text to display the a-value.

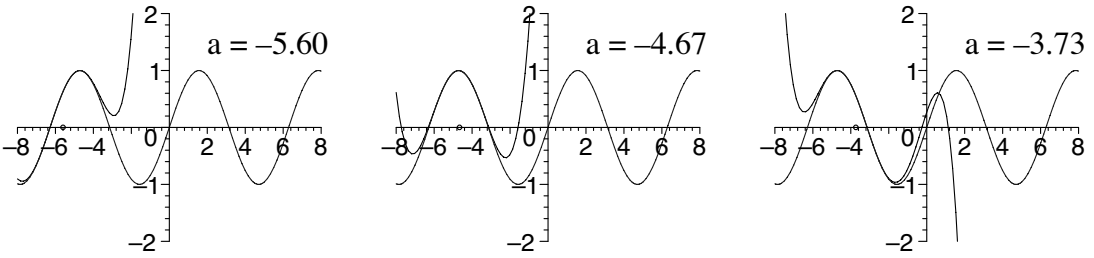
```
> f := x -> sin(x):
> MaxDegree := 5:
> aLow := -7:
> aHigh := 7:
> Domain := aLow-1..aHigh+1:
> Range := -2..2:
> NumFrames := 30:
> TextPoint := 2,1.5:
```

Next, we plot the example function and create a function $a(i)$ to compute values of a from $aLow$ to $aHigh$ in increments of $(aHigh - aLow)/NumFrames$. To make it clear in each frame exactly where the Taylor polynomial is centered, we plot a point on the x-axis and display the a-value.

```
> fPlot := plot( f(x), x=Domain ):
> a := i -> aLow + i*(aHigh-aLow)/NumFrames:
> aPlot := display( seq( pointplot( [a(i),0], symbol=circle,
  symbolsize=20, color=blue ), i=0..NumFrames ),
  insequence=true ):
> aValue := display( seq( textplot(
  [TextPoint,sprintf( "a = %.2f", a(i) )], align=RIGHT ),
  i=0..NumFrames ), insequence=true ):
```

We create a function T to compute the Taylor polynomials, each depending, this time, on the varying value of a. We then create a sequence of plots of these polynomials and display them.

```
> T := a -> convert( taylor( f(x), x=a, MaxDegree+1 ),
  polynom ):
> TaylorPoly := display( seq( plot( T(a(i)), x=Domain,
  numpoints=100, color=blue ), i=0..NumFrames ),
  insequence=true ):
> display( fPlot, aPlot, TaylorPoly, aValue, view=Range );
```



7.9.2 Maple 8 alternative using `TaylorApproximation`

The `TaylorApproximation` procedure includes by default some of the elements that we created above. The value of a is in the title, and a plot of the point $(a, f(a))$ is automatic, as is the function being approximated.

The preliminaries are the same.

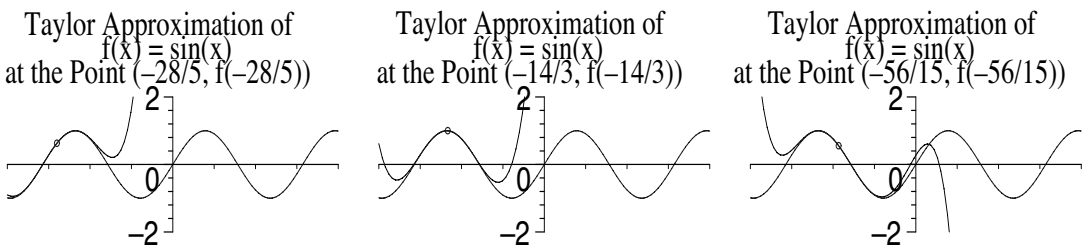
```
> restart:
> with( Student[Calculus1] ):
> with( plots ):
> setoptions( labels=["",""], thickness=2,
  axesfont=[HELVETICA,18] ):
> f := x -> sin(x):
> MaxDegree := 5:
> aLow := -7:
> aHigh := 7:
> Domain := aLow-1..aHigh+1:
> Range := -2..2:
> NumFrames := 30:
```

We use the same function to compute the a-value. This time, the function T will apply the procedure `TaylorApproximation` at the computed value of a.

```
> a := i -> aLow + i*(aHigh-aLow)/NumFrames:
> T := a -> TaylorApproximation( sin(x), x=a,
  order=MaxDegree, output=plot,
  pointoptions=[symbolsize=20], view=[Domain,Range],
  titlefont=[TIMES,ROMAN,24] ):
```

So $T(a(i))$ is a call to `TaylorApproximation` to produce a Taylor polynomial about the point $aLow$ plus i increments of $(aHigh - aLow)/NumFrames$. Finally, we display a sequence of these frames.

```
> display( seq( T(a(i)), i=0..NumFrames ),
  insequence=true );
```



Chapter 8

Plotting Vectors

Near the close of the nineteenth century, Lord Kelvin, the great British physicist, wrote that vectors have "never been of the slightest use to any creature." [4, p. 772] In this chapter, we will give them a wee look nonetheless. You will learn how to use Maple to compute dot products and cross products, to represent vectors as directed segments, and to enhance the illusion of three-dimensionality for plots of vectors in space. As usual, we will create some new animated demonstrations: one for teaching the concept of the cross product vector, one for velocity and acceleration vectors in two dimensions, and another for illustrating the central idea in the development of equations of lines in space.

8.1 The two `arrow` procedures

There are two procedures in Maple, both called `arrow`, useful for plotting a directed-segment representation of a vector, one in the `plots` package and another in `plottools`. The `plots` version is the newer and more convenient. It has fewer required arguments, and it plots three-dimensional vectors, by default, as a cylinder with a cone on the end. Although the `plottools` version does offer a `cylindrical_arrow` option to do that, it plots vectors, by default, as planar arrows even in three dimensions. The disadvantage is that, if you need to rotate a vector in three dimensions, as is likely, it will disappear whenever your point of view happens to be edge-on to the plane containing the arrow. We are going to opt here for the `plots` version of `arrow`.

This brings up a point worth considering. Suppose you want to use procedures from both `plots` and `plottools` in the same worksheet. If you load both packages, then which `arrow` will be in effect, the `plots` or the `plottools` version? The answer is the last one called. A `with(` *package* `)` statement redefines any procedures in *package* that are already available in the worksheet. (Maple always warns you that the redefinition has occurred.) So

```
> with( plottools ):
> with( plots ):
```

would make the `plots` version of `arrow` the active one. As a programming practice, however, this is not ideal. It makes the code less transparent. A better way is to load only those procedures that you need by qualifying the `with` statement. This is done by specifying, after the package name, which of the procedures you want to load. For example,

```
> with( plots, arrow, display ):
> with( plottools, line ):
```

loads only the `arrow` and `display` procedures from the `plots` package, and only the `line` procedure from `plottools`.

Another way to avoid the potential confusion of conflicting procedure definitions is to use longer procedure names, which include the package name, each time the procedure is used. For example, `plots[display](`*arguments*`)` or `plots[arrow](`*arguments*`)`. These longer names for the procedures tell Maple where to find them, so no `with` statement is needed at all.

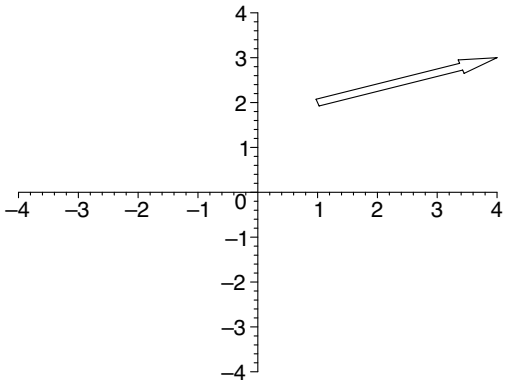
8.2 The `arrow` procedure of the `plots` package

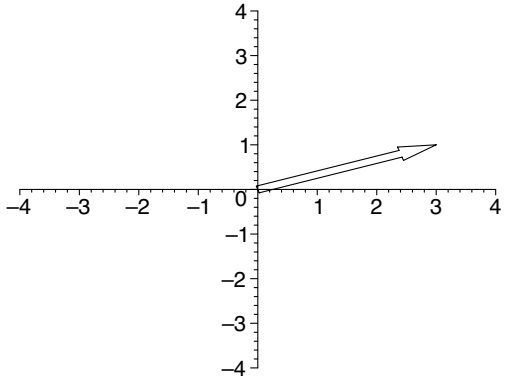
One form of the `arrow` procedure is

`arrow(` *initial point*`,` *vector*`,` *options* `)`

which plots a directed-segment representation of *vector* beginning at *initial point*. Both *initial point* and *vector* can be in the form of a point (`[`*a*`,`*b*`]` or `[`*a*`,`*b*`,`*c*`]`), which is a list, or a vector (`<`*a*`,`*b*`>` or `<`*a*`,`*b*`,`*c*`>`), which is a column vector. If *initial point* is omitted, it defaults to the origin. For the `arrow` options, see Section 8.4. A simple and very plain example is

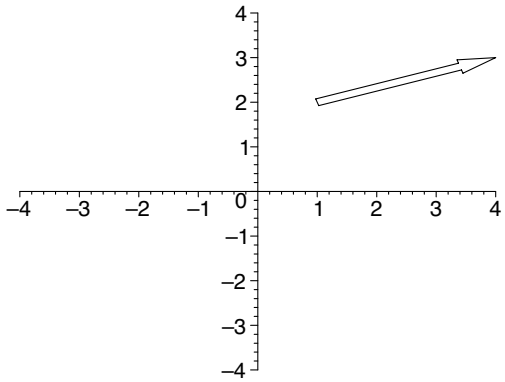
```
> with( plots ):
> arrow( [1,2], <3,1>, view=[-4..4,-4..4] );
> arrow( <3,1>, view=[-4..4,-4..4] );
```

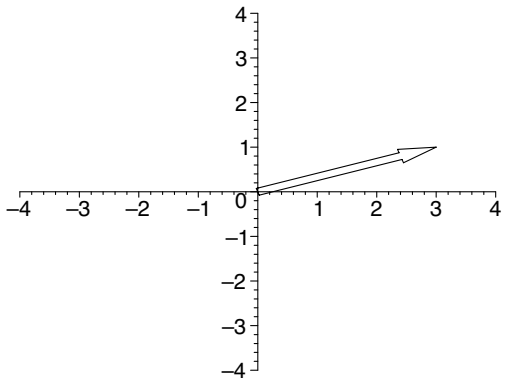




which plots the vector $\langle\, 3, 1 \,\rangle$, first with initial point $(1, 2)$, then with initial point $(0, 0)$. Using point notation for $\langle\, 3, 1 \,\rangle$ instead, we could have written

```
> with( plots ):
> arrow( [1,2], [3,1], view=[-4..4,-4..4] );
> arrow( [3,1], view=[-4..4,-4..4] );
```

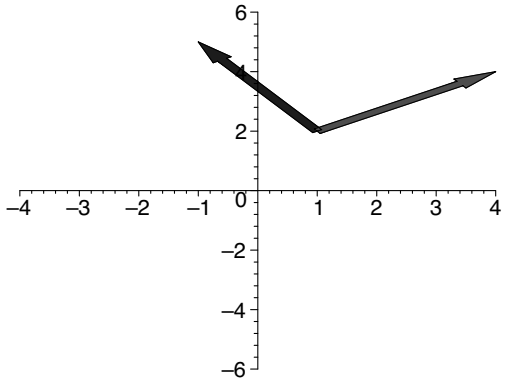




which accomplishes the same thing.

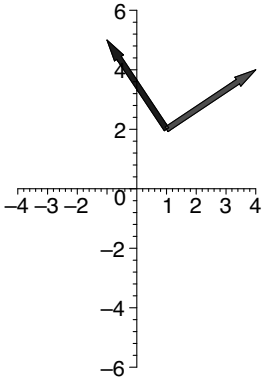
Typically, we will want to store representations of vectors for display later, such as

```
> with( plots ):
> v1 := arrow( [1,2],<3,2>, color=red ):
> v2 := arrow( [1,2],<-2,3>, color=blue ):
> display( v1, v2, view=[-4..4,-6..6] );
```



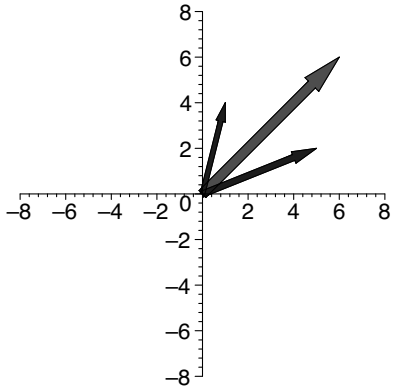
These vectors certainly don't look orthogonal, although they may look fine on your screen. If not, you can correct that by using constrained scaling:

```
> display( v1, v2, scaling=constrained,
  view=[-4..4,-6..6] );
```



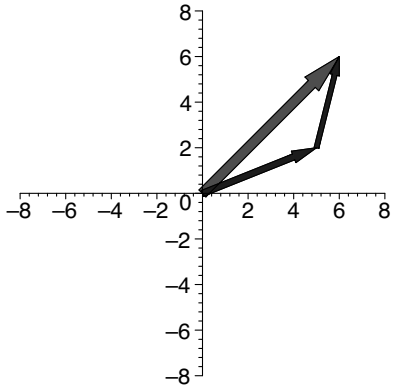
One way that we can use `arrow` is to illustrate the geometry of the sum of two vectors, by the parallelogram method,

```
> with( plots ):
> v1 := arrow( <5,2>, color=blue ):
> v2 := arrow( [5,2], <1,4>, color=blue ):
> s := arrow( <5,2> + <1,4>, color=red ):
> display( v1, v2, s, scaling=constrained,
  view=[-8..8,-8..8] );
```



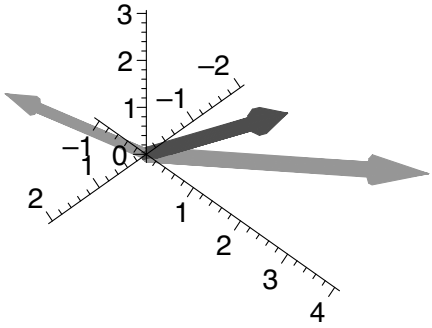
or the tip-to-tail method,

```
> with( plots ):
> v1 := arrow( <5,2>, color=blue ):
> v2 := arrow( <5,2>, <1,4>, color=blue ):
> s := arrow( <5,2> + <1,4>, color=red ):
> display( v1, v2, s, scaling=constrained,
  view=[-8..8,-8..8] );
```



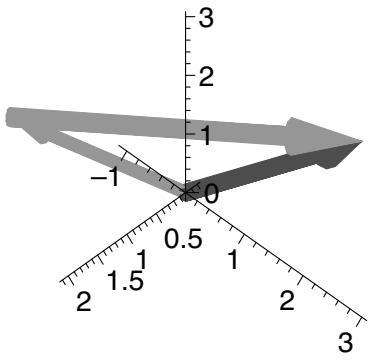
Three-dimensional vectors can be plotted in a similar way. The parallelogram method for vector sums in space is illustrated by

```
> with( plots ):
> v1 := arrow( <2,-1,2>, color=green ):
> v2 := arrow( <-2,4,1>, color=green ):
> s := arrow( <2,-1,2> + <-2,4,1>, color=red ):
> display( v1, v2, s, axes=normal, scaling=constrained );
```



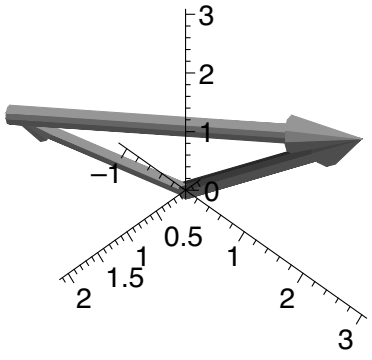
and the tip-to-tail method by

```
> with( plots ):
> v1 := arrow( <2,-1,2>, color=green ):
> v2 := arrow( [2,-1,2], <-2,4,1>, color=green ):
> s := arrow( <2,-1,2> + <-2,4,1>, color=red ):
> display( v1, v2, s, axes=normal, scaling=constrained );
```



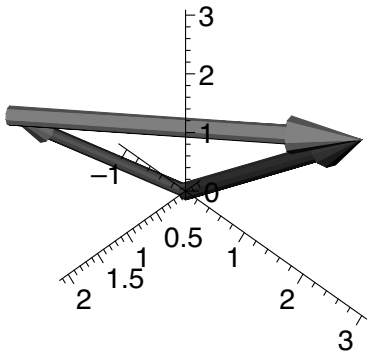
To give arrows in space a more three-dimensional appearance, we can add some shading, as if from a light source, using the `lightmodel` option. Notice how much more rounded and substantial the arrows appear in

```
> display( v1, v2, s, axes=normal, scaling=constrained,
  lightmodel=light3 );
```



The choices for `lightmodel` are `light1`, `light2`, `light3`, and `light4`. You can experiment with these by using the **Color** menu. (Click in the plot to select it, and the `plot3d` menus will appear.) It is also possible to design your own model using the option `light=[`ϕ`,`θ`,`r`,`g`,`b`]`. This option simulates a light source from the direction given by θ and ϕ, the angles (measured in degrees) of spherical coordinates, but notice that they are listed in reverse order. The values r, g, and b are the red, green, and blue intensities, each in the interval $[\,0,1\,]$. For example,

```
> display( v1, v2, s, axes=normal, scaling=constrained,
  light=[80,-10,.9,.9,.9] );
```



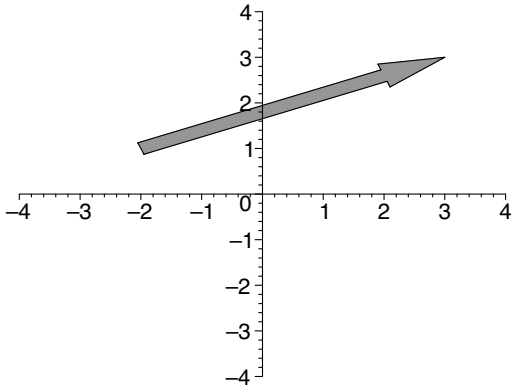
which puts the light source roughly over your left shoulder.

Another way to use the `arrow` procedure is to create a directed segment from one point to another by specifying the initial and terminal points. The general form, which employs the `difference` option, is

`arrow(` *initial point*`,` *terminal point*`, difference=true,`
other options `)`

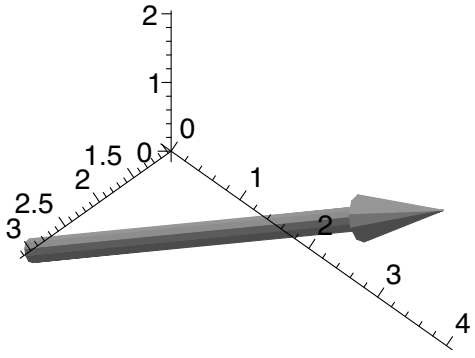
For example,

```
> with( plots ):
> arrow( [-2,1], [3,3], difference=true, color=green,
  view=[-4..4,-4..4] );
```



and

```
> with( plots ):
> arrow( [3,0,0], [1,4,2], difference=true, axes=normal,
  scaling=constrained, color=green, lightmodel=light3 );
```



Default is `difference=false`.

8.3 Dot product and cross product

Dot product (inner product or scalar product) is readily available by way of the "." operator. For example,

```
> a := <-2,3,1>;
> b := <1,-1,2>;
> a.b;
```

$$a := \begin{bmatrix} -2 \\ 3 \\ 1 \end{bmatrix}$$

$$b := \begin{bmatrix} 1 \\ -1 \\ 2 \end{bmatrix}$$

$$-3$$

For the cross product (vector product), we will need the `LinearAlgebra` package, which contains the `CrossProduct` procedure. For example,

```
> with( LinearAlgebra ):
> a := <1,2,3>;
> b := <2,1,4>;
> c := CrossProduct(<1,2,3>,<2,1,4>);
```

$$a := \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

$$b := \begin{bmatrix} 2 \\ 1 \\ 4 \end{bmatrix}$$

$$c := \begin{bmatrix} 5 \\ 2 \\ -3 \end{bmatrix}$$

The `LinearAlgebra` package also has a `DotProduct` procedure, so we could have found $\mathbf{a} \cdot \mathbf{b}$ using

```
> with( LinearAlgebra ):
> a := <-2,3,1>;
> b := <1,-1,2>;
> DotProduct(a,b);
```

$$a := \begin{bmatrix} -2 \\ 3 \\ 1 \end{bmatrix}$$

$$b := \begin{bmatrix} 1 \\ -1 \\ 2 \end{bmatrix}$$

$$-3$$

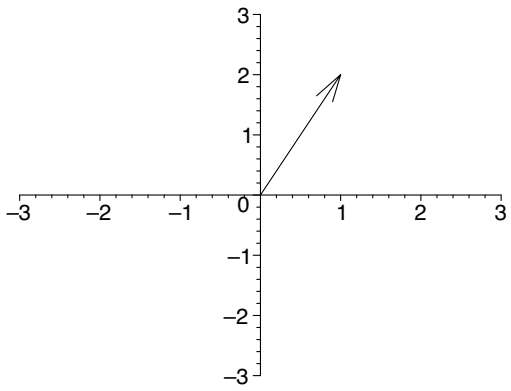
The `LinearAlgebra` package, as the name suggests, is the place to find Maple's matrix procedures, of which we will have occasion to use only a few. There is also an older, less efficient, `linalg` package. For a look, enter `?LinearAlgebra` or `?linalg` at the Maple prompt. Maple 8 has a new `VectorCalculus` package that supports a wide range of coordinate systems and allows adding your own. Enter `?VectorCalculus` for details.

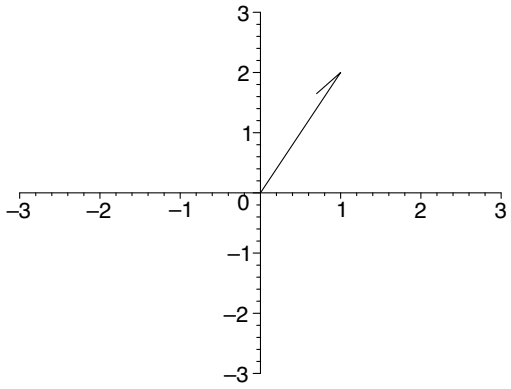
8.4 The `arrow` options

In addition to the `difference` option and the pertinent plotting options, the `arrow` procedure of the `plots` package offers some of its own. There are six: `shape`, `length`, `width`, `head_width`, `head_length`, and `plane`.

The `shape` option specifies the appearance of the arrow. We have already seen two of the choices, `double_arrow` and `cylindrical_arrow`, since these are the defaults for two and three dimensions, respectively. The other two, `arrow` and `harpoon`, are illustrated by

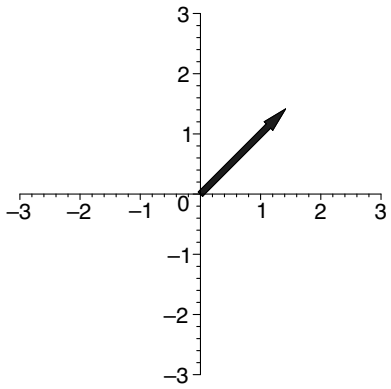
```
> with( plots ):
> arrow( <1,2>, shape=arrow, thickness=2,
  view=[-3..3,-3..3] );
> arrow( <1,2>, shape=harpoon, thickness=2,
  view=[-3..3,-3..3] );
```





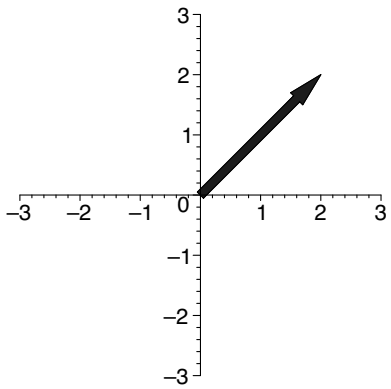
The `length` option can be used either to specify the length of the arrow directly or to specify a value by which the arrow is to be scaled. For example, the specification `length=4` creates an arrow of length 4, but the specification `length=[4,relative=true]` creates an arrow multiplied by the scalar 4. The default length is `length=[1,relative=true]`. For example,

```
> with( plots ):
> arrow( <1,1>, length=2, color=blue, scaling=constrained,
  view=[-3..3,-3..3] );
```



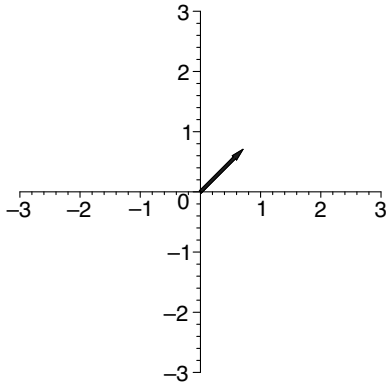
plots an arrow of length 2 in the direction of the vector $\langle 1, 1 \rangle$, and

```
> with( plots ):
> arrow( <1,1>, length=[2,relative=true], color=blue,
  scaling=constrained, view=[-3..3,-3..3] );
```



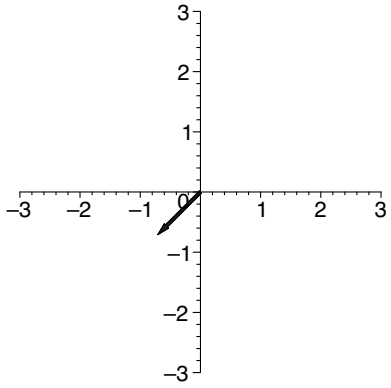
plots an arrow twice as long as the vector $\langle 1, 1 \rangle$. That is, it plots $2\langle 1, 1 \rangle$. A unit vector in the same direction as $\langle 1, 1 \rangle$ may be plotted using

```
> with( plots ):
> arrow( <1,1>, length=1, color=blue, scaling=constrained,
  view=[-3..3,-3..3] );
```



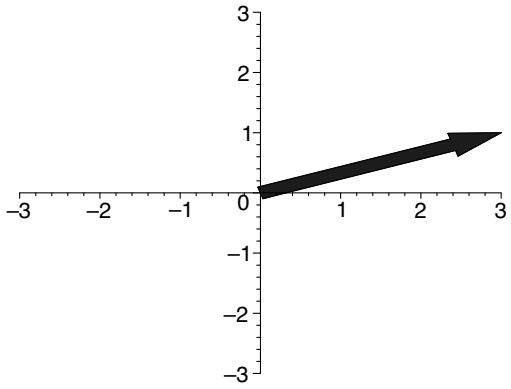
A unit vector in the opposite direction may be plotted with

```
> with( plots ):
> arrow( <1,1>, length=-1, color=blue, scaling=constrained,
  view=[-3..3,-3..3] );
```



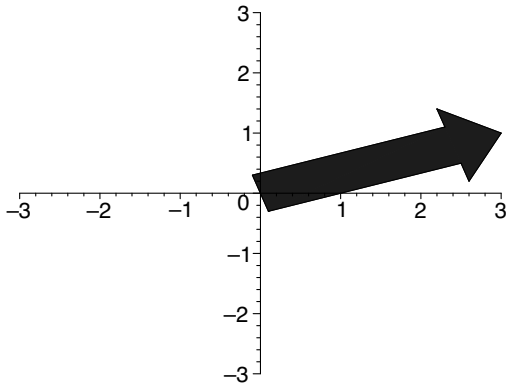
The `width` option can be used either to specify the width of the arrow directly or the ratio of width to length. The specification `width=.1` creates an arrow of width .1, but `width=[.1,relative=true]` creates an arrow whose width is one tenth of its length. The default is `width=[.05,relative=true]`. For example,

```
> with( plots ):
> arrow( <3,1>, width=.2, color=blue, view=[-3..3,-3..3] );
```



plots an arrow of width .2, but

```
> with( plots ):
> arrow( <3,1>, width=[.2,relative=true], color=blue,
  view=[-3..3,-3..3] );
```



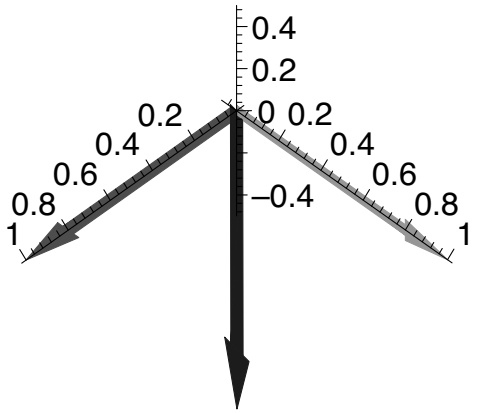
plots an arrow whose width is two tenths of its length.

The `head_width` option is used to specify either the width of the head of the arrow directly or the ratio of the width of the head to the width of the body of the arrow. For example, `head_width=.2` creates an arrow with a head of width .2, and `head_width=[3,relative=true]` creates an arrow whose head is three times as wide as the body of the arrow. Default is `head_width=[2,relative=true]`. Similarly, `head_length=.4` creates an arrow with a head of length .4, and `head_length=[.4,relative=true]` creates an arrow whose head is four tenths of the total length of the arrow. Default is `head_length=[.2,relative=true]`.

Whenever the choice specified for the `shape` option is `double_arrow`, `harpoon`, or `arrow`, the arrow created is planar. If you are rotating such an arrow in three dimensions, then, it will disappear whenever your viewpoint happens to be in the same plane as the arrow. The simplest way to prevent this is not to choose any shape in the first place, and just let this option default to `cylindrical_arrow`. If your preference for a planar arrow is unshakable, however, you can specify the plane that contains the arrow by using the `plane` option. The command `arrow(`P`, a, plane=b)` plots an

arrow with initial point P in the direction of the vector $\mathbf{a}$ and in the plane containing the point P that is parallel to $\mathbf{a}$ and $\mathbf{a} \times \mathbf{b}$. If $\mathbf{a}$ and $\mathbf{b}$ are parallel, the option is just ignored. Try, for example,

```
> with( plots ):
> v1 := arrow( <1,0,0>, shape=double_arrow, plane=<0,0,1>,
  color=red ):
> v2 := arrow( <1,1,0>, shape=double_arrow, plane=<-1,1,1>,
  color=blue ):
> v3 := arrow( <0,1,0>, shape=double_arrow, plane=<1,0,0>,
  color=green ):
> display( v1, v2, v3, axes=normal, scaling=constrained,
  view=-0.5..0.5 );
```



where the point P defaults to the origin. Rotate this plot so you can see how the planes containing these arrows are oriented. The vector $\mathbf{b}$ may be expressed in point form `[`b_1`,`b_2`,`b_3`]`, instead of vector form `<`b_1`,`b_2`,`b_3`>`.

It is good to have so much control over the appearance. Typically, however, I choose a width that seems appropriate, use it for all the arrows in the plot, and let the other options default. Generally, this looks better than having arrows with varying width. Moreover, it is the length of an arrow (representing the magnitude of a vector) that is the more important thing to focus on in the plot. In animations, this length is often changing. If the width is changing, too—as, by default, it would—it is a distraction from the central idea.

8.5 Demonstration: The cross product vector

The cross product vector is fundamental to the geometry of three dimensions, so it is important that students have a clear mental picture of its geometrical properties. Topics such as equations of planes and lines of intersection of two planes, and the distance between a point and a plane go more smoothly

when I have spent a few extra minutes sharing with the students some useful Maple plots. Three plots will be helpful.

The first will show two vectors **a** and **b** and their cross product $\mathbf{a} \times \mathbf{b}$ so that we can verify that $\mathbf{a} \times \mathbf{b}$ looks orthogonal to both **a** and **b**. We load the packages we need, choose two example vectors, and compute their cross product.

```
> restart:
> with( plots ):
> with( LinearAlgebra ):
> a := <1,2,3>;
> b := <2,-1,2>;
> c := CrossProduct(a,b);
```

$$a := \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

$$b := \begin{bmatrix} 2 \\ -1 \\ 2 \end{bmatrix}$$

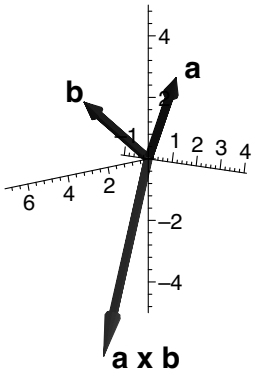
$$c := \begin{bmatrix} 7 \\ 4 \\ -5 \end{bmatrix}$$

Next, we create representations of these vectors using `arrow` and some labels for them using `textplot3d`.

```
> a_arrow := display( arrow( a, width=.25, color=blue ),
  textplot3d( [a[1],a[2],a[3]," a"], color=black,
  font=[HELVETICA,BOLD,14], align={ABOVE,RIGHT} ) ):
> b_arrow := display( arrow( b, width=.25, color=blue ),
  textplot3d( [b[1],b[2],b[3],"b "], color=black,
  font=[HELVETICA,BOLD,14], align={ABOVE,LEFT} ) ):
> c_arrow := display( arrow( c, width=.25, color=red ),
  textplot3d( [c[1],c[2],c[3]," a x b"], color=black,
  font=[HELVETICA,BOLD,14], align=RIGHT ) ):
```

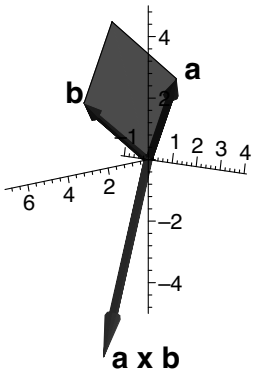
The choice of a width of .25 is the result of some experimentation to find a value that works well for this particular plot. In creating the labels, we have used `a[`i`]`, `b[`i`]`, and `c[`i`]` to access the components of the vectors, then located the labels at the terminal points of the vectors. Now, we display these structures.

```
> display( a_arrow, b_arrow, c_arrow, axes=normal,
  scaling=constrained, lightmodel=light3, view=-5..5 );
```



The second plot will show, in addition to $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{a} \times \mathbf{b}$, the parallelogram whose area has the same magnitude as $\mathbf{a} \times \mathbf{b}$. To plot the parallelogram, we will use the procedure `polygonplot3d` from the `plots` package, which expects a list of the vertices, that is, a list of lists. The vertices will be the origin together with the terminal points of the position vectors $\mathbf{a}$, $\mathbf{a} + \mathbf{b}$, and $\mathbf{b}$. The terminal points are just the components of those vectors, but, to be useable by `polygonplot3d`, we need to convert the vectors to lists. To do that, we will use the `convert` procedure (Section 7.7.1).

```
> P := polygonplot3d( [[0,0,0], convert(a,list),
  convert(a+b,list), convert(b,list)], color=green ):
> display( a_arrow, b_arrow, c_arrow, P, axes=normal,
  scaling=constrained, lightmodel=light3, view=-5..5 );
```



It would be instructive to see what happens to $\mathbf{a} \times \mathbf{b}$ and the parallelogram as the angle between $\mathbf{a}$ and $\mathbf{b}$ varies. We will fix $\mathbf{a}$ and let $\mathbf{b}$ rotate about the common initial point.

First, we choose a vector $\mathbf{a}$ and create a label for it.

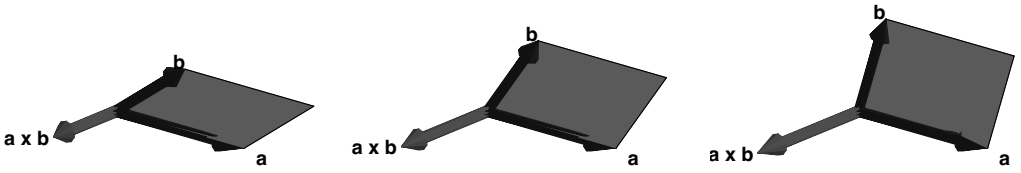
```
> a := <0,3/2,0>:
> a_arrow := display( arrow( a, width=.1, color=blue ),
  textplot3d( [a[1],a[2]+.2,a[3],"a"], color=black,
  font=[HELVETICA,BOLD,14], align=BELOW ) ):
```

We will build the frame sequence with a loop. We rotate $\mathbf{b}$ in the yz-plane by setting $\mathbf{b} = \langle\, 0, \cos(\pi i/10), \sin(\pi i/10) \,\rangle$ for $i = 1, 2, \ldots, 20$. In the body of the loop, for each i we create the vector $\mathbf{b}$ and its label, the vector $\mathbf{c} = \mathbf{a} \times \mathbf{b}$ and its label, and the corresponding parallelogram, then append a display of these to the existing frame sequence.

```
> FrameSequence := NULL:
> for i from 1 to 20 do
>   b := <0,cos(Pi/10*i),sin(Pi/10*i)>:
>   b_arrow := display( arrow( b, width=.1, color=blue ),
    textplot3d( [b[1],b[2],b[3],"b"], color=black,
    font=[HELVETICA,BOLD,14], align={ABOVE,LEFT} ) ):
>   c := CrossProduct(a,b);
>   c_arrow := display( arrow( c, width=.1, color=red ),
    textplot3d( [c[1],c[2],c[3]," a x b"], color=black,
    font=[HELVETICA,BOLD,14], align=LEFT ) ):
>   P := polygonplot3d( [[0,0,0], convert(a,list),
    convert(a+b,list), convert(b,list)], color=green ):
>   FrameSequence := FrameSequence,
    display( b_arrow, c_arrow, P )
>   end do:
```

Finally, we store a display of the frames in sequence, then display this together with the fixed vector $\mathbf{a}$, which is the background plot.

```
> Frames := display( FrameSequence, insequence=true ):
> display( a_arrow, Frames, scaling=constrained,
  lightmodel=light4, orientation=[40,70] );
```



Before I use this demonstration, I establish that $\mathbf{a} \times \mathbf{b}$ is orthogonal to both $\mathbf{a}$ and $\mathbf{b}$ and that its direction obeys the right-hand rule. (If the fingers of your right hand curl in the direction from $\mathbf{a}$ toward $\mathbf{b}$, your thumb points in the direction of $\mathbf{a} \times \mathbf{b}$.) Then I use the first plot, which shows only the vectors. I rotate this a little so we can check that $\mathbf{a} \times \mathbf{b}$ looks orthogonal to both $\mathbf{a}$ and $\mathbf{b}$. Then I prove that $|\mathbf{a} \times \mathbf{b}| = |\mathbf{a}||\mathbf{b}| \sin\theta$, where θ is the angle between $\mathbf{a}$ and $\mathbf{b}$. So, in addition, the length of the cross product of $\mathbf{a}$ and $\mathbf{b}$ is the area of the parallelogram determined by $\mathbf{a}$ and $\mathbf{b}$. Then I show the second plot, which includes this parallelogram. Next, we discuss that $\mathbf{a}$ and $\mathbf{b}$ are parallel exactly when their cross product is the zero vector. After this, I tell the

students that it would be useful to watch what happens, as the angle between **a** and **b** changes, to the parallelogram and the cross product vector. I step the animation one frame at a time. We check that $\mathbf{a} \times \mathbf{b}$ obeys the right-hand rule and that it lengthens or shortens as the parallelogram's area increases or decreases. When **a** and **b** are opposite in direction, I point out that it makes sense that the parallelogram and cross product have disappeared. When the cross product's direction is away from the viewer and into the screen, I point out that it is, in fact, obeying the right-hand rule. Finally, when **a** and **b** have the same direction, the parallelogram and cross product have vanished once again.

8.6 Demonstration: Velocity and acceleration vectors in two dimensions

An important application of vectors is their use to represent velocity and acceleration. To demonstrate this idea, it would be ideal to see how the velocity and acceleration vectors behave, and how they interact, as a point moves along a curved path. We will create an animation to do that. We will make three plots: one to show the point as it moves along the curve, another to show the changing velocity vector, and a third to show both the velocity and the acceleration vectors. This demonstration will be two-dimensional; we will create a three-dimensional version in the next chapter.

We begin by calling the necessary package. We choose constrained scaling so that the lengths of the vectors will not be distorted, and select a suitable view.

```
> restart:
> with( plots ):
> setoptions( scaling=constrained,
  view=[-0.85..0.85,-1.1..0.7] ):
```

As the path for the point, we will use the position function $\langle\, f(t), g(t) \,\rangle = \langle\, \sin 3t \cos t, \sin 3t \sin t \,\rangle$ on $[\, \pi/4, 3\pi/4 \,]$. We need to create vector-valued functions for the position, velocity, and acceleration. Recall from Section 6.3 that useful notations for the first and second derivatives of a function f are `D(`f`)` and `(D@@2)(`f`)`, respectively.

```
> f := t -> sin(3*t)*cos(t);
> g := t -> sin(3*t)*sin(t);
> alpha := Pi/4;
> beta := 3*Pi/4;
> Position := t -> <f(t), g(t)>:
> Velocity := t -> <D(f)(t), D(g)(t)>:
> Acceleration := t -> <(D@@2)(f)(t), (D@@2)(g)(t)>:
```

$$f := t \to \sin(3\,t)\cos(t)$$
$$g := t \to \sin(3\,t)\sin(t)$$
$$\alpha := \frac{\pi}{4}$$
$$\beta := \frac{3\,\pi}{4}$$

Next, we plot and store the curve, which is the background plot.

```
> Curve := plot( [f(t), g(t), t=alpha..beta],  style=line,
  color=blue, thickness=2 ):
```

We now choose a number of frames (beyond the first one) *NumFrames* and create a function for the parameter t that we can use to compute values from α to β in increments of $(\beta - \alpha)/NumFrames$.

```
> NumFrames := 30:
> t := i -> alpha + i*(beta-alpha)/NumFrames:
> Scale := .15:
```

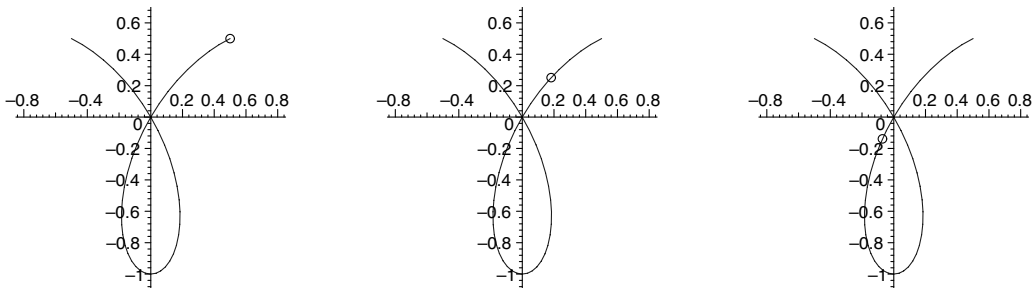
The factor *Scale* is used to scale down the vectors so that they are not so large in relation to the curve. This artifice, which improves the appearance of the plot, amounts to a change in units, so nothing is lost mathematically.

Next, we generate a moving point, as we did in Section 5.7, and sequences of arrows for the velocity and acceleration vectors.

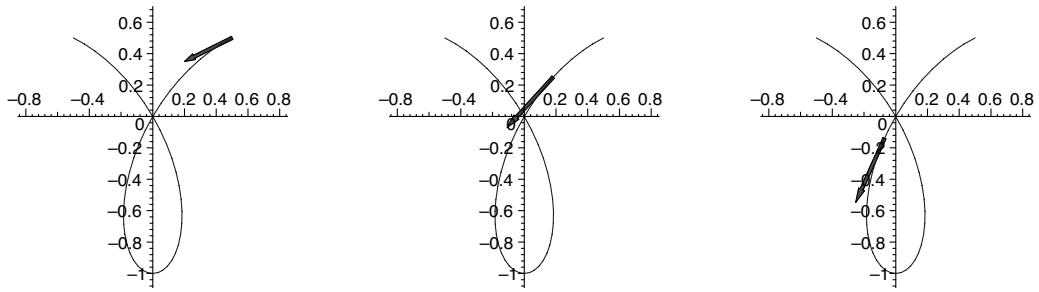
```
> Points := display( seq( pointplot( [f(t(i)), g(t(i))],
  style=point, symbol=circle, symbolsize=17, color=black ),
  i=0..NumFrames ), insequence=true ):
> VelocityVectors := display( seq( arrow( Position(t(i)),
  Scale*Velocity(t(i)), width=.02, color=red ),
  i=0..NumFrames ), insequence=true ):
> AccelerationVectors := display( seq( arrow(
  Position(t(i)), Scale*Acceleration(t(i)), width=.02,
  color=green ), i=0..NumFrames ), insequence=true ):
```

Finally, we display the three animations.

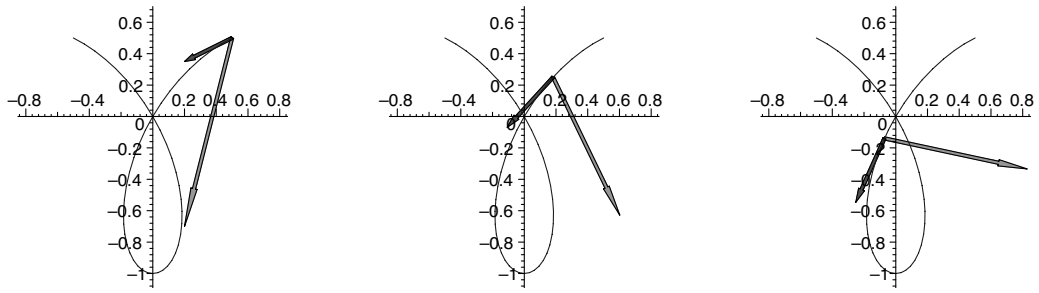
```
> display( Curve, Points );
```



```
> display( Curve, VelocityVectors );
```



```
> display( Curve, VelocityVectors, AccelerationVectors );
```



When I use this demonstration, first I show just the point moving along the curve. I click the button in the context bar to make the animation cycle continuously and let the students watch the point move for a while. I want them to become familiar with its motion. As they watch, I call their attention to the fact that the point slows down as it approaches the tight turn at the bottom and speeds up as it comes out of the turn, as a driver might do when negotiating such a curve.

Then I show the plot that includes the velocity vector, making it play continuously, too. I point out two things. The vector is always pointing in the direction of the inertial path, that is, the path the particle would follow if the forces that are currently holding it on the curve were to cease doing so. The other thing is that the velocity vector shortens as the particle slows down, as it should since speed is the magnitude of velocity, and lengthens as the particle speeds up.

Then I show the plot that contains both the velocity and the acceleration vectors. Here, I start with the animation paused and point out, again, two things. The acceleration vector acts to push the particle off its inertial (straight-line) path, always, therefore, acting toward the inside of the turn. I step the animation through a few frames so we can watch that happen. Then I point out that, whenever the vector projection of the acceleration onto the velocity is in the same direction as the velocity, the particle is speeding up, and whenever the vector projection of acceleration onto velocity is in the opposite direction to the velocity, the particle slows down. Finally, I let the animation run continuously so the students can watch the interaction of the two vectors as the point moves along the curve.

8.7 Demonstration: Lines in space

A development of an equation of a line in space, whether we are aiming for parametric form, symmetric form, or vector form, will begin with vectors. A fixed point and a direction are sufficient to determine a line. Let P be the fixed point whose position is given by the vector $\mathbf{r}_0$, and let the direction be given by the vector $\mathbf{v}$. Then every point on the line has position vector $\mathbf{r}(t) = \mathbf{r}_0 + t\mathbf{v}$ for some scalar t (and the converse). To convince students of that, we seek a demonstration that shows the three vectors, $\mathbf{r}$, $\mathbf{r}_0$, and $t\mathbf{v}$, as t varies over real values. We want students to see the terminal point of $\mathbf{r}$ tracing out points along the line.

The `line` procedure (Section 6.4) from the `plottools` package will be useful here. As usual, we will also want some procedures from the `plots` package. To avoid confusing the different `arrow` procedures in these two packages, we will call just the procedures we need from each.

```
> restart;
> with( plots, arrow, display, textplot3d ):
> with( plottools, line ):
> setoptions3d( scaling=constrained ):
```

As our example, we will use $\mathbf{r}_0 = \langle 1, 2, 3 \rangle$ and $\mathbf{v} = \langle 2, 3, 2 \rangle$. In this demonstration, we take advantage of the option to use point notation instead of vector notation for $\mathbf{r}_0$ and $\mathbf{v}$. This way, our notation will do double duty. It will serve to represent both a point and a position vector. We can then use $\mathbf{r}_0$ and $\mathbf{v}$ in procedures that expect their arguments to be points.

```
> r0 := [1,2,3]:
> v := [2,3,2]:
> r := t -> r0 + t*v:
```

Next, we create an arrow to represent $\mathbf{r}_0$ and a label for it, then display them together as *r0Vector*. To position the label near the middle of the arrow, we divide each component of $\mathbf{r}_0$ by 2. The i^{th} component of $\mathbf{r}_0$ is denoted `r0[`i`]`. In this way, we create a sequence *V1LabelPos* of three values to use as the position of the label in `textplot3d` (Section 7.2).

```
> V1 := arrow( r0, color=green, width=.2 ):
> V1LabelPos := r0[1]/2, r0[2]/2, r0[3]/2:
> V1Label := textplot3d( [V1LabelPos, "r0"], color=green,
  font=[TIMES,BOLD,18], align=LEFT ):
> r0Vector := display( V1, V1Label ):
```

Similarly, for $\mathbf{v}$ we create an object *vVector*. The midpoint of the directed segment from $P(1, 2, 3)$ representing $\mathbf{v}$ has position vector $\mathbf{r}(1/2)$.

```
> V2 := arrow( r0, v, color=red, width=.2 ):
> V2MidPt := r(1/2):
> V2LabelPos := V2MidPt[1], V2MidPt[2], V2MidPt[3]:
> V2Label := textplot3d( [V2LabelPos, "v"], color=red,
  font=[TIMES,BOLD,18], align={ABOVE,LEFT} ):
> vVector := display( V2, V2Label ):
```

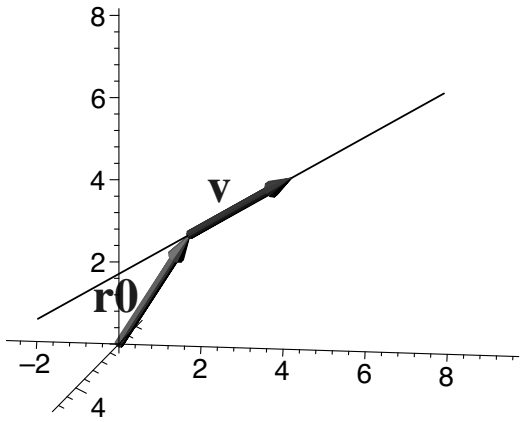
To plot the line, we choose two points and use the `line` procedure. The points whose position vectors are $\mathbf{r}(-1.5)$ and $\mathbf{r}(2.5)$ work well in this case.

```
> L := line( r(-1.5), r(2.5), color=black, thickness=3 ):
```

Here, we have taken advantage of our representation of $\mathbf{r}$ as a list (point form). We can hand $\mathbf{r}(-1.5)$ and $\mathbf{r}(2.5)$ directly to the `line` procedure, which expects lists.

We then display the line and the two vectors that determine it.

```
> display( r0Vector, vVector, L, axes=normal,
  orientation=[10,75], light=[60,-30,.9,.9,.9] );
```



Some tweaking may be in order. If you are using Maple 8, this plot probably looks fine. In Maple 7, it probably doesn't. If the labels interfere with the arrows, you can adjust *V1LabelPos* leftward and *V2LabelPos* upward. On my screen, this works in Maple 7:

```
> V1LabelPos := r0[1]/2, r0[2]/2-1, r0[3]/2:
> V2LabelPos := V2MidPt[1], V2MidPt[2], V2MidPt[3]+.4:
```

Opting for a loop structure, we will first need to initialize a sequence S in which to keep the frames. To do a thorough job, we will show the vector $t\mathbf{v}$ for both positive and negative values of t. Starting at 1, we will let t increase in increments of .2 to 2, then decrease in decrements of .2 to -1. To accomplish that, we just create two sequences and append one to the other.

```
> S := NULL:
> ParameterValues := seq( 1+.2*i, i=0..5 ),
  seq( 2-.2*i, i=0..15 ):
```

Inside the loop, we form an arrow, *tvVector*, for the vector $t\mathbf{v}$ and a label for it. The label is in two parts: *tLabel* for the scalar t, which should be italic, and *vLabel* for the vector $\mathbf{v}$, which we will make bold as is often done for vectors. We take advantage of the opportunity offered by the two separate parts to color them differently. We also create an arrow *SumVector* for the vector sum $\mathbf{r}(t) = \mathbf{r}_0 + t\mathbf{v}$, then display all these elements as a single frame and append it to the frame sequence S.

```
> for t in ParameterValues do
>   tvVector := arrow( r0, t*v, width=.2, color=red );
>   tvMidPt := r(t/2);
>   tvLabelPos := tvMidPt[1], tvMidPt[2], tvMidPt[3];
>   tLabel := textplot3d( [tvLabelPos, "t  "], color=black,
    font=[TIMES,ITALIC,18], align={ABOVE,LEFT} );
>   vLabel := textplot3d( [tvLabelPos, " v"], color=red,
    font=[TIMES,BOLD,18], align={ABOVE,LEFT} );
>   SumVector := arrow( r(t), width=.2, color=blue );
>   S := S, display( SumVector, tvVector, tLabel, vLabel )
>   end do:
```

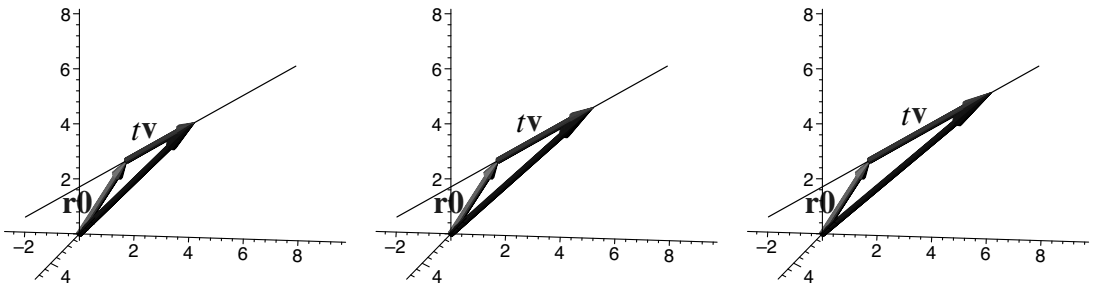
Again, the position of the label *tvLabelPos* may need an upward adjustment, particularly in Maple 7.

Finally, we store a display of the frames in sequence,

```
> Sum_and_tvVectors := display( S, insequence=true ):
```

then display the background plot and the animated elements.

```
> display( r0Vector, Sum_and_tvVectors, L, axes=normal,
  orientation=[10,75], light=[60,-30,.9,.9,.9] );
```



After defining the vector form of the equation of a line, and before I use this demonstration, I often show an example in two dimensions on the blackboard, drawing the position vector, the direction vector, and the line. When I use this demonstration, I show the first plot—the one with no animated elements—and rotate it. This allows the students to look for a moment at the geometrical objects. Then I play the animation, letting it loop, and explain that the scalar t is varying over values between 1 and -2, and as this happens, the terminal point of the (blue) position vector $\mathbf{r}(t)$ traces out the line.

Chapter 9

Plotting Space Curves

A significant benefit of a computer algebra system such as Maple is its ability to plot a curve in three dimensions. The capability to rotate the plot in real time is a remarkably effective aid to visualizing the curve's shape. In this chapter, we will discuss two ways to represent space curves. One is the standard curved-line representation. The other, a tube, significantly improves the three-dimensionality of the representation. We will create three new demonstrations. One shows the relationship between a vector-valued function and a space curve. Another is an elaborate one for demonstrating the directional derivative and the gradient vector. The third demonstrates in three dimensions the ideas of velocity and acceleration, showing the changing velocity and acceleration vectors as a point moves along a curved path in space.

9.1 The `spacecurve` procedure

To plot a curve in three dimensions, we can use the `spacecurve` procedure of the `plots` package. One form is

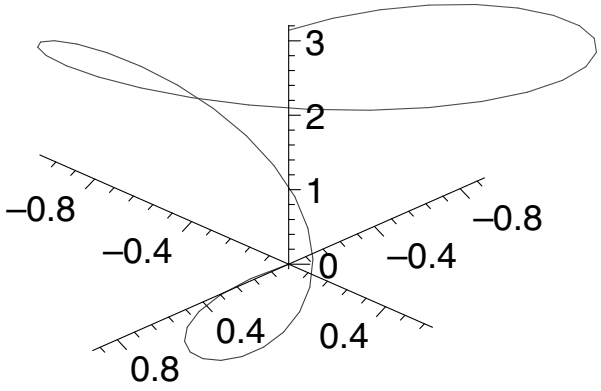
`spacecurve( [`$f(t)$`,`$g(t)$`,`$h(t)$`] ,` t`=`a`..`b`,` *options* `)`

where f, g, and h are the component functions of the curve. The options are the same ones as for `plot3d`, except for the `grid` option, which is not applicable here. For example, to plot the space curve whose parametric equations are

$$x = \sin 3t \cos t$$
$$y = \sin 3t \sin t$$
$$z = t$$

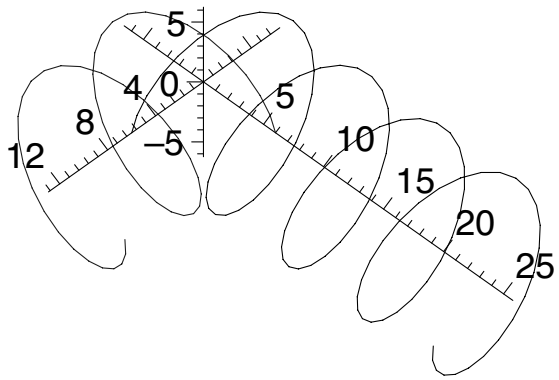
we may use

```
> with( plots ):
> spacecurve( [sin(3*t)*cos(t), sin(3*t)*sin(t), t],
  t=0..Pi, axes=normal, shading=z );
```



We have given the domain for the parameter outside the list, which is consistent with the other three-dimensional parametric form, a surface defined using two parameters (see Section 2.2). Syntax does not require this, however. We may include the parameter's domain within the list instead. This is useful whenever we have more than one space curve to plot—`spacecurve` accepts a set of lists—and the curves have different domains. We can also include any options local to a particular curve inside the list with that curve's component functions. For example,

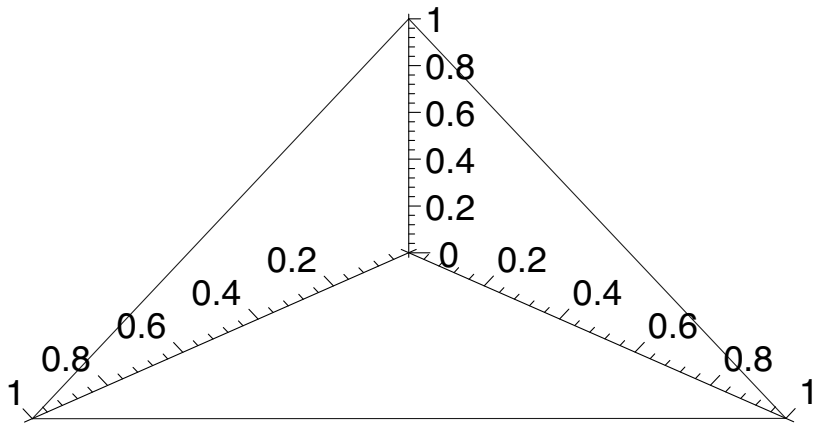
```
> with( plots ):
> spacecurve( {[t,6*cos(t),6*sin(t),t=0..4*Pi,color=red],
  [6*cos(t),t,6*sin(t),t=0..8*Pi,color=blue,numpoints=100]},
  thickness=2, axes=normal, scaling=constrained );
```



The `numpoints` option defaults to 50 as it does for curves in two dimensions.

The list of component functions can be replaced by a list of points. In this case, `spacecurve` connects successive points with line segments. For example,

```
> with( plots ):
> spacecurve( [[0,0,1], [0,1,0], [1,0,0], [0,0,1]],
  axes=normal, color=green );
```



9.2 Demonstration: Curves in space

One way to help students understand what a vector-valued function is, geometrically, is to connect the function to a space curve. Think of placing all the vectors in the range of the function in standard position (initial point at the origin) and collecting all the terminal points of the vectors. That collection is a space curve. Having read those two sentences, you may have a picture in mind. It might even be moving. Let's make a short animation that will give that mental image to students, too. It should show a vector swinging about the origin, sweeping out a space curve.

We will use the vector-valued function $\mathbf{r}(t) = \langle \sin 3t \cos t, \sin 3t \sin t, t \rangle$ for our example.

```
> restart:
> with( plots ):
> setoptions3d( thickness=3, axes=normal,
  scaling=constrained, orientation=[10,60],
  lightmodel=light2, view=[-3..3,-3..3,-1..7] ):
> r := t -> [sin(3*t)*cos(t), sin(3*t)*sin(t), t];
```

$$r := t \to [\sin(3\,t)\cos(t),\ \sin(3\,t)\sin(t),\ t]$$

As we did in Section 8.7, and for similar reasons, we have chosen to represent **r** in point form so that **r** is a list. Since the `spacecurve` procedure expects a list, we can use **r** as a parametric form of the curve in a call to `spacecurve`. Since `arrow` will accept a list, **r** will also serve as the terminal point of the arrows we use to represent the vectors. We also add some shading using `lightmodel` to strengthen the illusion of three-dimensionality.

We will use a loop to generate the sequence of frames. We begin by initializing it and choosing a number of frames.

```
> NFrames := 20:
> FrameSeq := NULL:
```

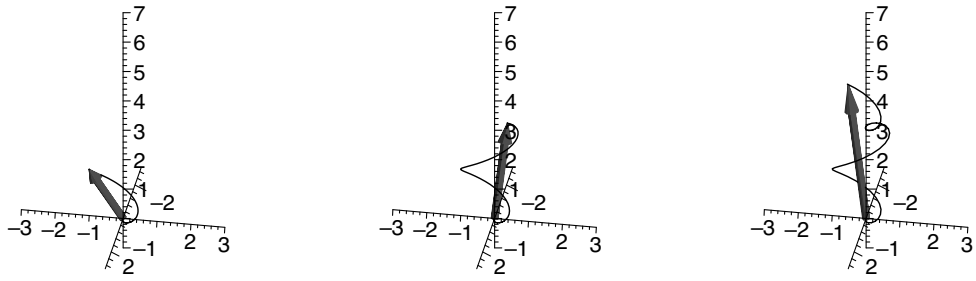
In the body of the loop, we want to plot a vector and a portion of the space curve from, say, $t = 0$ to the tip of the vector.

```
> for i from 0 to NFrames do
>   rVector := arrow( r(2*Pi*i/NFrames), color=red,
    width=.2 ):
>   Curve := spacecurve( r(t), t=0..2*Pi*i/NFrames+0.01,
    color=black ):
>   FrameSeq := FrameSeq, display( Curve, rVector )
>   end do:
```

So we generate, for each i, an arrow in standard position to represent the vector $\mathbf{r}(2\pi i/NFrames)$ and the associated space curve from $t = 0$ to (near) $t = 2\pi i/NFrames$. We have added a small value, 0.01, to the upper limit of the domain of t because `spacecurve` wants the domain for the parameter to be nonempty. We then append a display of these two objects to the frame sequence. We omitted the initial point in the `arrow` procedure, letting it default to the origin, because we want the vectors to be in standard position.

There is no background plot, so we finish by just displaying the frame sequence.

```
> display( FrameSeq, insequence=true );
```



When I use this demonstration, I first explain that a space curve is the set of all the terminal points of vectors (represented by directed segments in standard position) in the range of a vector-valued function, then I show this animation. After seeing the animation once, everyone knows the relationship between a vector function and a space curve. No lengthy discussion is needed. I usually pause and ask students to guess what would be the projection of this space curve onto the xy-plane. It isn't easy. I have tried this with audiences of mathematicians. (I didn't show them the defining equation of the space curve, of course.) Most are surprised, when I rotate the plot so that the view is straight down the z-axis, to see a three-leaved rose take shape.

9.3 Demonstration: Directional derivative and gradient vector

These are the concepts that prompted me to learn to create animations. I used to take my students outside and position them at various points on a hillside, having them point in the direction of the gradient vector and extend their arms more or less in accordance with the magnitude of the gradient at the particular spot where they stood. It was marginally effective, but subject to the weather (in Michigan). For directional derivatives, I could draw what I wanted to illustrate reasonably well on a blackboard, but I believed that the best way to help students understand the directional derivative is to show them what happens to it as the direction actually changes. And for that, I needed animation.

In this elaborate demonstration, we will create all the elements we might want to have on hand to demonstrate what a directional derivative is and how it relates to the gradient vector. Then we can choose which ones we prefer to use. A list is:

- a surface $z = f(x, y)$
- a point $P(a, b)$ in the xy-plane
- a unit vector $\mathbf{u}$, for the changing direction, represented as an arrow from P in the xy-plane
- the vertical plane through P and parallel to $\mathbf{u}$
- the trace of the surface in the vertical plane
- the point $(a, b, f(a, b))$ on the trace
- the line tangent to the trace at $(a, b, f(a, b))$ and within the vertical plane
- the slope (the directional derivative) of the tangent line in the direction $\mathbf{u}$
- the gradient vector

To demonstrate what happens as direction changes, we want $\mathbf{u}$ to rotate about P so we can observe the effect on the other objects.

We start by choosing a suitable view and some labels and fonts. We will use `lightmodel` to add some shading and give the appearance of depth.

```
> restart:
> with( plots ):
> with( plottools, line ):
> setoptions3d( axes=boxed, view=[-2..2,-2..2,-0.15..3],
  font=[TIMES,ROMAN,24], labels=[x,y,""],
  labelfont=[TIMES,BOLDITALIC,24], axesfont=[HELVETICA,18],
  lightmodel=light4, scaling=constrained ):
```

We have loaded just the `line` procedure from the `plottools` package to avoid redefining `arrow`, as discussed in Section 8.1.

We choose the number of frames we will be using and, with that, calculate the increment $\Delta\theta$ for the angle that $\mathbf{u}$ makes with the x-axis.

```
> NumFrames := 16:
> DeltaTheta := 2*Pi/NumFrames:
```

As our example, we will use the function $f(x,y) = 3\sin x \sin y/xy$ and the point $P(1,1)$. We store a plot of the surface as well as a plot of the point $(1,1,0)$, about which the unit vector will rotate, and the point $(1,1,f(1,1))$, through which the tangent line will pass.

```
> f := (x,y) -> 3*sin(x)*sin(y)/(x*y);
> Surface := plot3d( f(x,y), x=-2..2, y=-2..2,
  style=patchnogrid, shading=zhue, orientation=[20,75] ):
> Points := pointplot3d( {[1,1,0], [1,1,f(1,1)]},
  color=black, symbol=circle ):
```

$$f := (x,\, y) \rightarrow \frac{3\sin(x)\sin(y)}{y\,x}$$

The unit vectors have the form $\langle \cos(i\,\Delta\theta), \sin(i\,\Delta\theta) \rangle$, which we will represent in three dimensions as a sequence of arrows with initial point $(1,1,0)$ in the direction $\mathbf{u}_i = \langle \cos(i\,\Delta\theta), \sin(i\,\Delta\theta), 0 \rangle$ for $i = 0, 1, 2, \ldots, \mathit{NumFrames} - 1$. We first form two component functions, $u_1(i) = \cos(i\,\Delta\theta)$ and $u_2(i) = \sin(i\,\Delta\theta)$, that we can use again later.

```
> u1 := i -> cos(i*DeltaTheta):
> u2 := i -> sin(i*DeltaTheta):
> u := i -> arrow( [1,1,0], <u1(i),u2(i),0>, width=.1,
  color=red ):
> UnitVectors := display( seq( u(i), i=0..NumFrames-1 ),
  insequence=true ):
```

We define the plane P_i through $(1,1,0)$ and parallel to $\mathbf{u}_i$ parametrically by $[\,1 + r\,u_1(i), 1 + r\,u_2(i), z\,]$, where we will use $r \in [\,-1,1\,]$ and $z \in [\,0,3\,]$.

```
> P := i -> [1+r*u1(i), 1+r*u2(i), z]:
> VerticalPlanes := animate3d( P(i), r=-1..1, z=0..3,
  i=0..NumFrames-1, frames=NumFrames, grid=[7,10],
  color=gray, style=wireframe ):
```

The choices `grid=[7,10]` and `style=wireframe` render the planes substantial enough to be clearly visible, but transparent enough that they do not hide other features. (To eliminate the internal grid lines, use `grid=[2,2]`.)

The points on a trace are exactly the points on the surface that are also on some vertical plane. So the trace T_i of the surface in the vertical plane P_i is the space curve defined parametrically by $[\,x, y, f(x,y)\,]$, where $x = 1 + r\,u_1(i)$ and $y = 1 + r\,u_2(i)$. We will use $r \in [\,-1,1\,]$ again.

```
> T := i -> [1+r*u1(i), 1+r*u2(i), f(1+r*u1(i),1+r*u2(i))]:
> Traces := display( seq( spacecurve( T(i), r=-1..1),
  i=0..NumFrames-1 ), color=black, thickness=2,
  insequence=true ):
```

The gradient vector at $P(1,1)$ is $\nabla f = \langle \frac{\partial}{\partial x} f(1,1), \frac{\partial}{\partial y} f(1,1) \rangle$, which we will represent in three dimensions as an arrow with initial point $(1,1,0)$ in the direction $\langle \frac{\partial}{\partial x} f(1,1), \frac{\partial}{\partial y} f(1,1), 0 \rangle$. First, we form ∇f, denoting it *Del_f*, then access its components, ∇f_1 and ∇f_2, using *Del_f*[1] and *Del_f*[2]. Recall from Section 6.3 that $\partial f/\partial x$ and $\partial f/\partial y$ are denoted `D[1](f)` and `D[2](f)`, respectively.

```
> Del_f := <D[1](f)(1,1),D[2](f)(1,1)>:
> GradientVector := arrow( [1,1,0], <Del_f[1],Del_f[2],0>,
  width=.1, color=green ):
```

The tangent to the trace T_i passes through the point $(1,1,f(1,1))$ and has direction $\langle u_1(i), u_2(i), \nabla f(1,1) \cdot \langle u_1(i), u_2(i) \rangle \rangle$. So the tangent line has the parametric form $[1 + r\,u_1(i), 1 + r\,u_2(i), f(1,1) + r\,\nabla f(1,1) \cdot \langle u_1(i), u_2(i) \rangle]$. We will just find two points P_1 and P_2 on this line, corresponding to $r = -1$ and $r = 1$, respectively, then we will use the `line` procedure of `plottools` to generate the tangent lines. Recall from Section 8.3 that the operator "." can be used to denote the dot product.

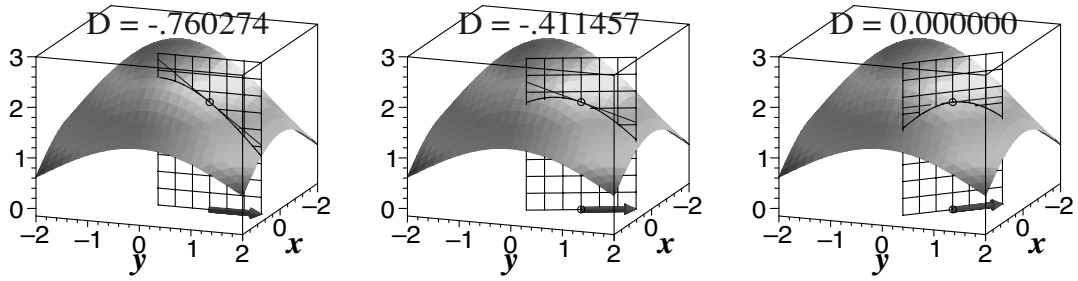
```
> P1 := i ->
  [1-u1(i), 1-u2(i), f(1,1)-Del_f.<u1(i),u2(i)>]:
> P2 := i ->
  [1+u1(i), 1+u2(i), f(1,1)+Del_f.<u1(i),u2(i)>]:
> TanLine := i -> line( P1(i), P2(i), thickness=2,
  color=blue ):
> Tangents := display( seq( TanLine(i), i=0..NumFrames-1 ),
  insequence=true ):
```

We next compute the directional derivative $\nabla f(1,1) \cdot \mathbf{u}_i$ so that we can print its value. We convert it to a string using `sprintf` as described in Section 7.5, choose a suitable location, and print it using `textplot3d`.

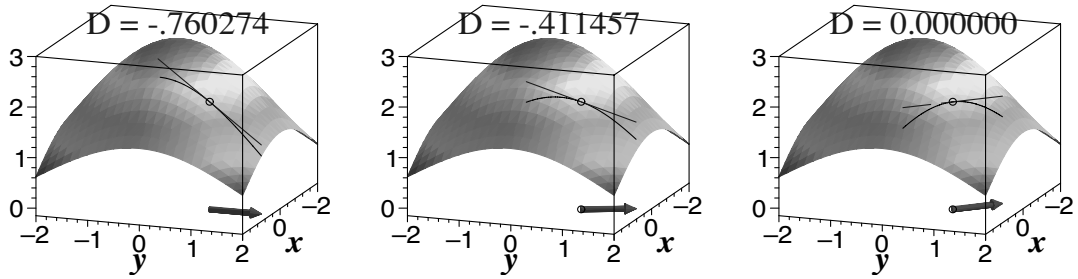
```
> DirectDeriv := i -> Del_f.<u1(i),u2(i)>:
> DerivativeValues := display( seq( textplot3d(
  [0,-2,3,sprintf( "D = %f", DirectDeriv(i) )], align=RIGHT,
  color=blue ), i=0..NumFrames-1 ), insequence=true ):
```

Finally, we display everything in three separate plots so that we can use them as suggested below.

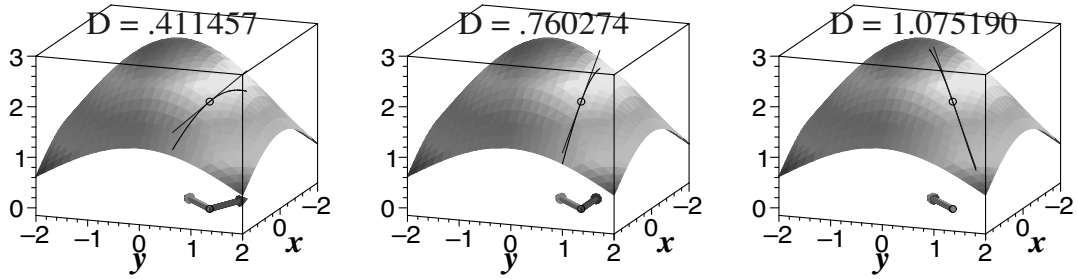
```
> display( Surface, Points, VerticalPlanes, UnitVectors,
  Traces, Tangents, DerivativeValues );
```




```
> display( Surface, Points, UnitVectors, Traces, Tangents,
  DerivativeValues );
```




```
> display( Surface, Points, UnitVectors, Traces, Tangents,
  DerivativeValues, GradientVector );
```



I have found this demonstration to be useful enough that I use it several times. I think it is best used early. First, I define the directional derivative as a limit, prove that $D_{\langle u_1,u_2\rangle} f(x,y) = \frac{\partial}{\partial x} f(x,y)\, u_1 + \frac{\partial}{\partial y} f(x,y)\, u_2$, work an example, then show the demonstration immediately. What I want the students to understand from the beginning is the geometry so that they will have an image to tie to the concept.

I start by showing the first plot with animation stopped. I rotate it to help students get oriented. Then I point out all the objects in the plot: the surface, the unit vector at the point P and the vertical plane they determine, the trace of the surface in this plane, and the tangent line to the trace. I start with a frame in which the directional derivative is nonzero—say negative—and explain that, of the two possible directions along the tangent line, the unit vector specifies one, and the line is, in fact, downward-sloping in this direction. If the unit vector $\mathbf{u}$ were in the opposite direction, then the directional derivative would be the opposite of its present value. Then I step through a few frames, pointing out the changing direction of $\mathbf{u}$ and its effect on the vertical

plane, trace, tangent line, and slope. I stop when **u** has direction opposite its starting direction and verify that the directional derivative is the negative of its starting value. I also stop when **u** has the direction of the positive end of the x-axis and note that the directional derivative there is the same as the partial derivative with respect to x. Similarly, for the partial with respect to y.

There is a good deal of information in this plot, embodied by the several objects in it, and, although all of it is useful at some point, some people find it a little too much to deal with at one time. It helps to take something out. I recommend removing the vertical plane, as in the second `display` statement. I play the animation slowly and let the students watch it for a while.

The next class, after the students have practiced finding directional derivatives, I use the demonstration a second time. First, I define the gradient vector ∇f and prove that the directional derivative is a maximum when **u** has the same direction as ∇f and that this maximum value is $|\nabla f|$. The third `display` statement in the demonstration includes ∇f (in green). I step through this animation and point out that the directional derivative does reach its maximum value when **u** is in the direction of ∇f. I stop the animation at the point when **u** and ∇f have the same direction and explain that, since the directional derivative is a little more than 1, ∇f should be a little longer than **u**. If you rotate the plot a bit, you can see that ∇f does peek out a little beyond **u**.

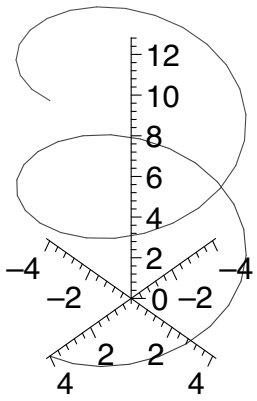
This animation is useful for demonstrating some other properties, too. The directional derivative is the scalar projection of ∇f onto **u**. Step the animation to the point when the directional derivative is about 3/4, then turn the plot upside down, rotating it so that your point of view is directly beneath the xy-plane, that is, from the negative end of the z-axis. From this viewpoint, you can see how ∇f and **u** are related to each other without the distortion of perspective. Now think of projecting ∇f (green) onto **u** (red), and this projection will, indeed, be about 3/4 of the length of **u**. Now when ∇f and **u** are orthogonal, the scalar projection, and therefore the directional derivative, should be zero. Without rotating the plot yet, step through a few frames until the two vectors are orthogonal, then rotate the plot back to near its original orientation so that you can see the displayed value of the directional derivative, which is, of course, zero.

9.4 The `tubeplot` procedure

Clearly, `spacecurve` is a useful procedure in many plotting situations. Whenever the exact shape of the curve is central, however, we can convey this information more effectively with a tube. The procedure `tubeplot` of

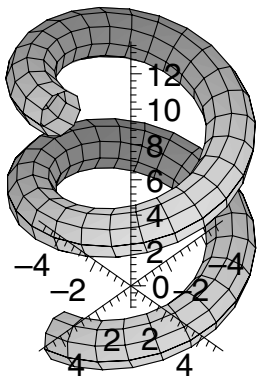
the `plots` package does this. The syntax is similar to that for `spacecurve`. Compare

```
> with( plots ):
> spacecurve( [4*cos(t),4*sin(t),t], t=0..4*Pi, axes=normal,
  scaling=constrained );
```



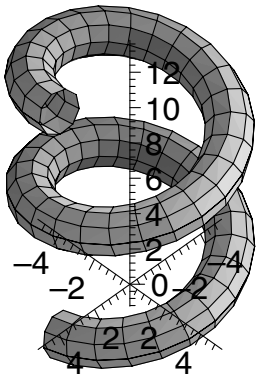
with

```
> with( plots ):
> tubeplot( [4*cos(t),4*sin(t),t], t=0..4*Pi, axes=normal,
  scaling=constrained );
```



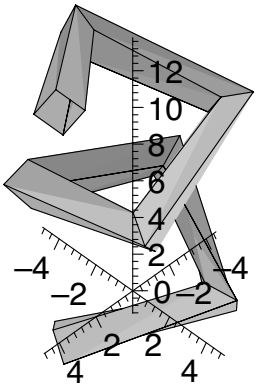
Notice how much more three-dimensional the curve appears when plotted as a tube. At points where the curve passes in front of itself, it is clear which portion of the curve is nearer and which farther, and that is especially helpful for curves less familiar than a helix. To strengthen further the appearance of three-dimensionality, we can add some shading using `lightmodel`.

```
> with( plots ):
> tubeplot( [4*cos(t),4*sin(t),t], t=0..4*Pi, axes=normal,
  scaling=constrained, lightmodel=light4 );
```



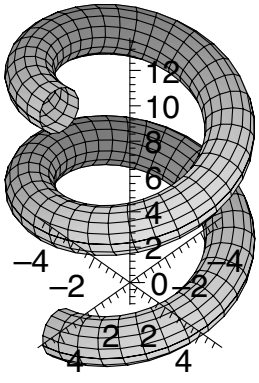
The `tubeplot` procedure offers its own options in addition to the ones for `spacecurve`. The option `tubepoints=`n will create a tube whose cross-section is a regular polygon with $n-1$ sides. Default is $n = 10$. The `numpoints` option, as it applies to `tubeplot`, specifies the number of such polygons to be used along the length of the tube. Default is 50. Compare

```
> with( plots ):
> tubeplot( [4*cos(t),4*sin(t),t], t=0..4*Pi, tubepoints=5,
  numpoints=8, axes=normal, scaling=constrained );
```



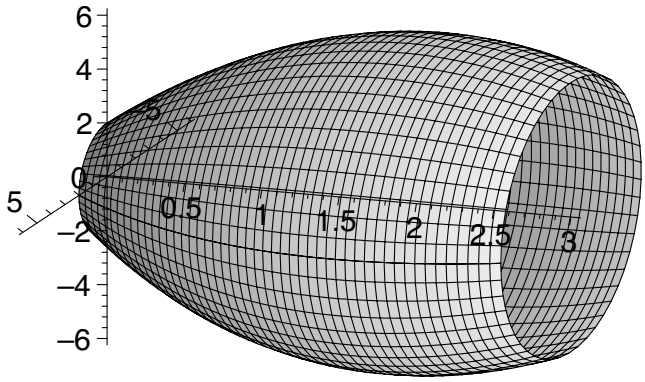
with

```
> with( plots ):
> tubeplot( [4*cos(t),4*sin(t),t], t=0..4*Pi, tubepoints=15,
  numpoints=70, axes=normal, scaling=constrained );
```



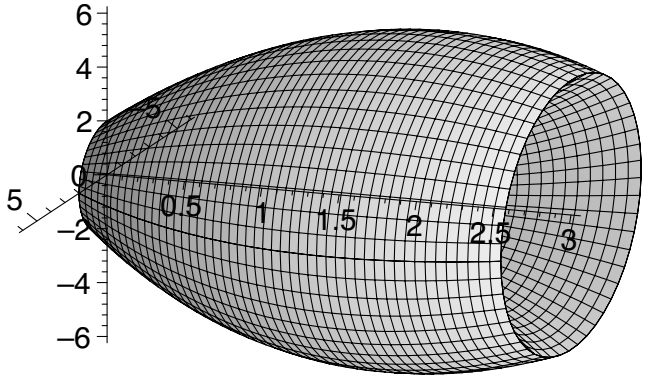
The `radius` option can be used to specify a radius for the tube other than the default value of 1. It is a value that you will usually want to adjust to an appropriate size for the particular curve and scale in your plot. The radius can also be variable, and we can use that fact to get an acceptable plot quickly of a surface of revolution:

```
> with( plots ):
> f := x -> -x^2 + 4*x + 2:
> a := 0:
> b := 3:
> tubeplot( [0,t,0], t=a..b, radius=f(t), tubepoints=50,
  axes=normal );
```



The `tubeplot` procedure will accept a set of lists, parametrically defining a set of space curves. Any options local to a particular curve can be specified within its list. For example, we can get a quick plot of the solid generated by revolving about the x-axis the region bounded by $y = -x^2 + 4x + 2$ and $y = x^2 - 2x + 2$ with

```
> with( plots ):
> f := x -> -x^2 + 4*x + 2:
> g := x -> x^2 - 2*x + 2:
> a := 0:
> b := 3:
> tubeplot( {[0,t,0, radius=f(t)], [0,t,0, radius=g(t)]},
  t=a..b, tubepoints=50, axes=normal );
```



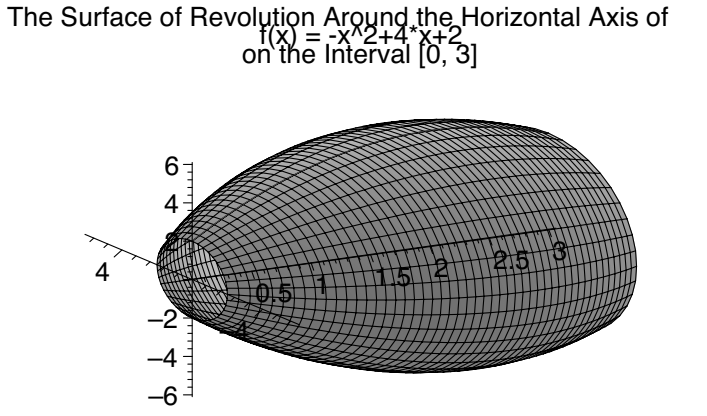
To appreciate this graceful plot, which we made with so little effort, rotate it a little so that you can see inside it. If you value terse code, you might like knowing that we could have produced this solid with the single statement,

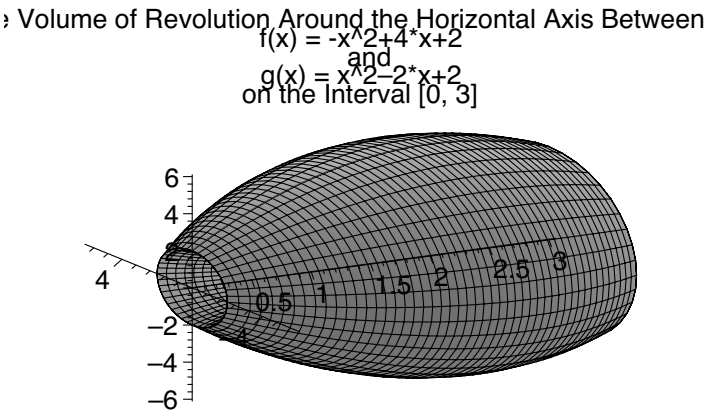
```
> plots[tubeplot]( {[0,t,0, radius=-t^2 + 4*t + 2],
  [0,t,0, radius=t^2 - 2*t + 2]}, t=0..3, tubepoints=50,
  axes=normal );
```

which employs the long form of the procedure name (Section 8.1). My own preference is for code that reflects mathematical notation. I hope that makes it more readable by others. It makes it more readable by me when I return to it later.

Incidentally, the `Student[Calculus1]` package of Maple 8 has procedures `SurfaceOfRevolution` and `VolumeOfRevolution`. Using these, code that produces equivalent plots to our two examples is

```
> restart:
> with( Student[Calculus1] ):
> f := x -> -x^2 + 4*x + 2:
> g := x -> x^2 - 2*x + 2:
> a := 0:
> b := 3:
> SurfaceOfRevolution( f(x), x=a..b, output=plot );
> VolumeOfRevolution( f(x), g(x), x=a..b, output=plot );
```





9.5 Demonstration: Velocity and acceleration vectors in three dimensions

To gain a full understanding of velocity and acceleration vectors, a student should see them at work in three dimensions. With this demonstration, we finish what we began in Section 8.6. We create an animation to demonstrate the behavior and interaction of velocity and acceleration as a point moves along a curved path in space. Maple's power is particularly helpful in this case. The excellent moving image is illuminating, and I can tell that the demonstration helps my students. After I made it and watched it for a while, I am sure that I understood the behavior of these vectors better myself.

For our example, we will use the position function $\langle f(t), g(t), h(t) \rangle = \langle \sin 3t \cos t, \sin 3t \sin t, 5/2 \sin^2 t \rangle$ on $[\pi/4, 5\pi/4]$. We will proceed in much the same way as we did in creating the two-dimensional version of this demonstration in Section 8.6. We begin in the usual way, then form vector-valued functions for position, velocity, and acceleration.

```
> restart:
> with( plots ):
> setoptions3d( scaling=constrained,
  view=[-3..3,-3..3,-0.5..3] ):
> f := t -> sin(3*t)*cos(t);
> g := t -> sin(3*t)*sin(t);
> h := t -> 2.5*sin(t)^2;
> alpha := Pi/4;
> beta := 5*Pi/4;
> Position := t ->  <f(t), g(t), h(t)>:
> Velocity := t ->  <D(f)(t), D(g)(t), D(h)(t)>:
> Acceleration := t ->
  <(D@@2)(f)(t), (D@@2)(g)(t), (D@@2)(h)(t)>:
```

$$f := t \to \sin(3\,t)\cos(t)$$

$$g := t \to \sin(3\,t)\sin(t)$$

$$h := t \to 2.5\sin(t)^2$$

$$\alpha := \frac{\pi}{4}$$

$$\beta := \frac{5\,\pi}{4}$$

Next, we need a plot of the curve to serve as the background plot. We add shading, using `lightmodel`.

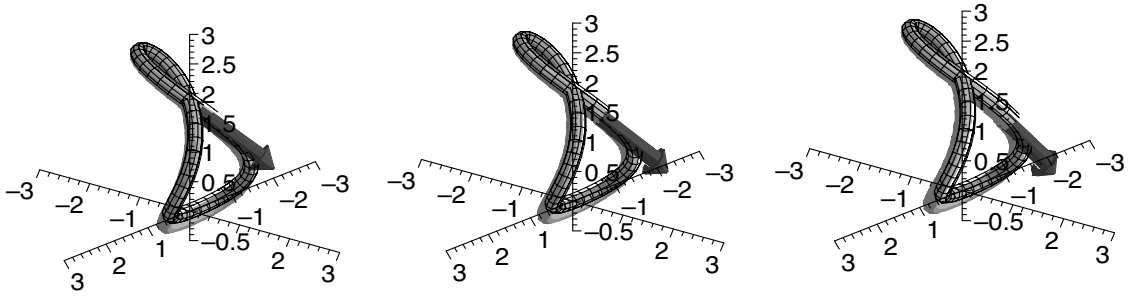
```
> Curve := tubeplot( [f(t), g(t), h(t)], t=alpha..beta,
  radius=.12, axes=normal, color=cyan, thickness=1,
  orientation=[60, 50], lightmodel=light4 ):
```

We choose a number of frames and a scale factor, and create the function t for the parameter. Then we generate arrows to represent the velocity and acceleration vectors. This animation is consumptive. In the interest of efficiency, we have not plotted the moving point this time. It will be clear enough where it is since the arrows do emanate from it. If it takes very long or uses too much of your computer's available memory to generate this plot, *NumFrames* can, of course, always be reduced.

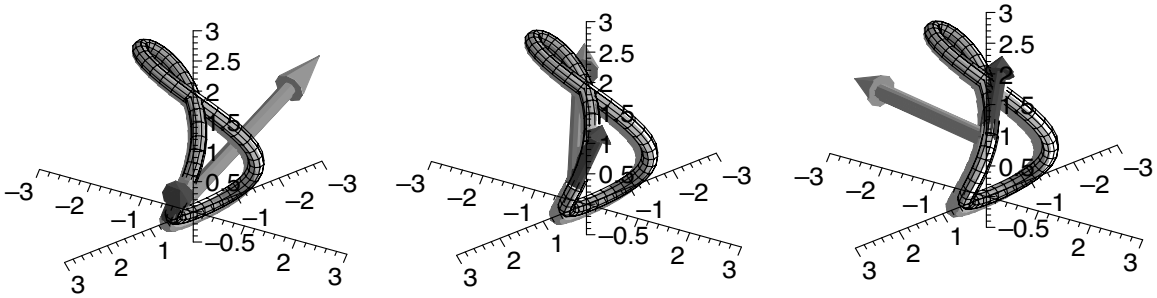
```
> NumFrames := 30:
> t := i -> alpha + i*(beta-alpha)/NumFrames:
> Scale := .4:
> VelocityVectors := display( seq( arrow(
  Position(t(i)), Scale*Velocity(t(i)), width=.24,
  color=red ), i=1..NumFrames ), insequence=true ):
> AccelerationVectors := display( seq( arrow(
  Position(t(i)), Scale*Acceleration(t(i)), width=.24,
  color=green ), i=1..NumFrames ), insequence=true ):
```

Finally, we generate two plots.

```
> display( Curve, VelocityVectors );
```



```
> display( Curve, VelocityVectors, AccelerationVectors );
```



I use this demonstration directly after the two-dimensional one that we made in Section 8.6, and I emphasize the same kinds of things. The particle slows down as it approaches the tight turn—this time, at the top. As this happens, the vector projection of acceleration onto velocity becomes opposite to the velocity, and the velocity vector shortens. The acceleration vector acts to push the particle off a straight-line path, always acting toward the inside of the turn. I let the animation that includes both velocity and acceleration

cycle continuously and rotate it gently from side to side as it runs, allowing the students to observe the interaction of the vectors from several points of view.

Chapter 10

Transformations and Morphing

In this chapter, we will develop tools to demonstrate linear transformations by gradually transforming the space. The `plottools` package contains some procedures that we can use to do this. We will look at those, as well as some other procedures within `plottools` that are generally useful tools. We will see how to implement a matrix transformation in Maple and discuss a technique for morphing. Then we will develop demonstrations for linear transformations in two and three dimensions.

10.1 The `plottools` package

The `plottools` package contains several procedures that are useful in creating animations. Two of them, `circle` and `line`, are procedures that we have already used in demonstrations. For completeness, we reproduce them here. The `circle` procedure has the form

`circle(` `[`c_1`,`c_2`]`, *radius*, *options* `)`

where (c_1, c_2) is the center, *radius* defaults to 1, and the options are the same as those for the `plot` statement. The `line` procedure has the forms

`line(` `[`x_1`,`y_1`]`, `[`x_2`,`y_2`]`, *options* `)`

and

`line(` `[`x_1`,`y_1`,`z_1`]`, `[`x_2`,`y_2`,`z_2`]`, *options* `)`

which plot the line segment between the points (x_1, y_1) and (x_2, y_2) and the points (x_1, y_1, z_1) and (x_2, y_2, z_2), respectively. The options are the same as for `plot` or `plot3d`. The structures produced by `circle` and `line` can be plotted using `display`.

With the `polygon` procedure, we can create a polygonal plot structure in two or three dimensions, then display it using `display`. The essential argument is just a list of the vertices. The syntax is

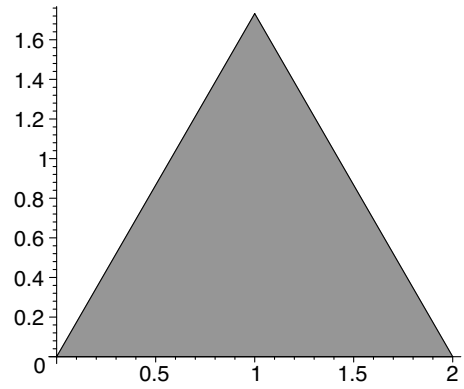
`polygon(` `[[`x_1`,`y_1`]`, `[`x_2`,`y_2`]`,..., `[`x_n`,`y_n`]]`, *options* `)`

or

`polygon( [[`x_1`,`y_1`,`z_1`], [`x_2`,`y_2`,`z_2`],...,[`x_n`,`y_n`,`z_n`]], `*options*` )`

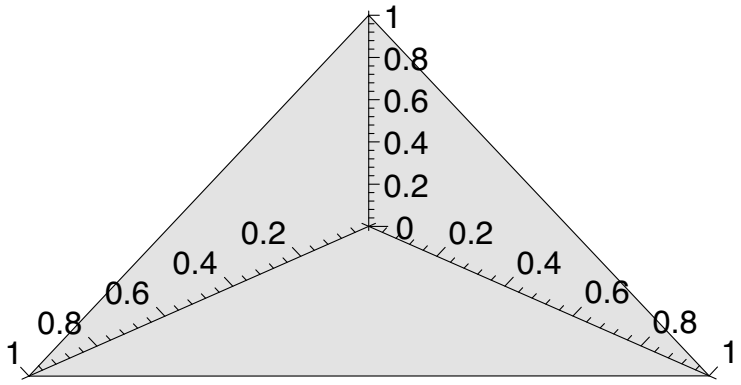
The usual plotting options are available. For example,

```
> with( plottools ):
> with( plots ):
> display( polygon( [[0,0], [2,0], [1,sqrt(3)]],
  color=green ), scaling=constrained );
```



and

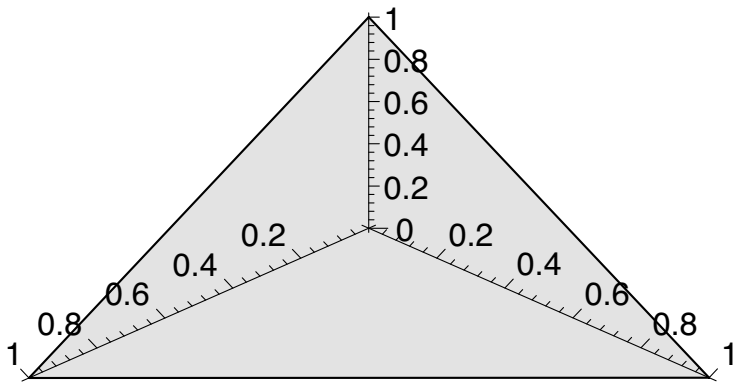
```
> with( plottools ):
> with( plots ):
> display( polygon( [[1,0,0], [0,1,0], [0,0,1]],
  color=yellow ), axes=normal );
```



The `style` option offers choices such as `patch` and `patchnogrid`. You can specify these settings in the `polygon` or the `display` statement, and, in Maple 7, you can change them after the polygon is displayed by clicking on the plot to select it and using the **Style** menu. In Maple 8, options specified in the `polygon` statement are protected from change. (See "hard-coded" options in

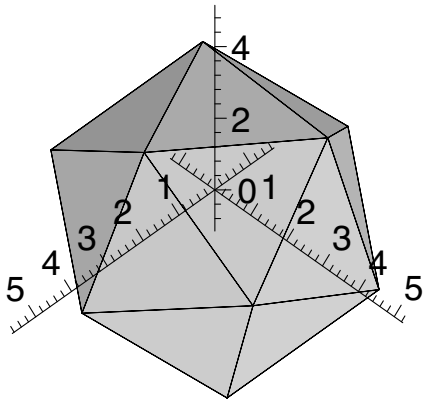
Section 2.14.) Also, you can emphasize the edges by choosing a larger value for the thickness under the same menu or by using the `thickness` option. For example,

```
> with( plottools ):
> with( plots ):
> display( polygon( [[1,0,0], [0,1,0], [0,0,1]],
  color=yellow, thickness=3 ), axes=normal );
```



The `plottools` package also has some useful built-in routines for plotting polyhedra. For example,

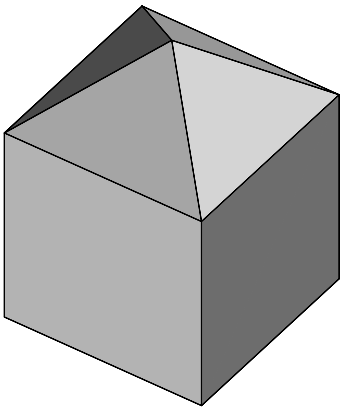
```
> with( plottools ):
> Icos := icosahedron( [2,2,2], 3 ):
> with( plots ):
> display( Icos, scaling=constrained, axes=normal );
```



plots an icosahedron centered at the point $(2, 2, 2)$ and scaled by a factor of 3. You can also build your own polyhedron from polygons. For example,

```
> with( plots ):
> with( plottools ):
> P1 := [0,0,0]:
```

```
> P2 := [1,0,0]:
> P3 := [1,1,0]:
> P4 := [0,1,0]:
> P5 := [0,0,1]:
> P6 := [1,0,1]:
> P7 := [1,1,1]:
> P8 := [0,1,1]:
> P9 := [.5,.5,1.4]:
> face1 := polygon( [P1,P2,P3,P4], color=navy ):
> face2 := polygon( [P1,P2,P6,P5], color=turquoise ):
> face3 := polygon( [P2,P3,P7,P6], color=aquamarine ):
> face4 := polygon( [P3,P4,P8,P7], color=sienna ):
> face5 := polygon( [P4,P1,P5,P8], color=gray ):
> face6 := polygon( [P5,P6,P9], color=maroon ):
> face7 := polygon( [P6,P7,P9], color=tan ):
> face8 := polygon( [P7,P8,P9], color=wheat ):
> face9 := polygon( [P8,P5,P9], color=khaki ):
> display( face1, face2, face3, face4, face5, face6, face7,
  face8, face9, scaling=constrained );
```



where we have created polygonal faces from the points $P_1, P_2, \ldots, P_9$.

Two other procedures, `rotate` and `transform`, are pertinent to our topic. We will discuss them separately. For the details on other `plottools` procedures, type `?plottools` at the Maple prompt.

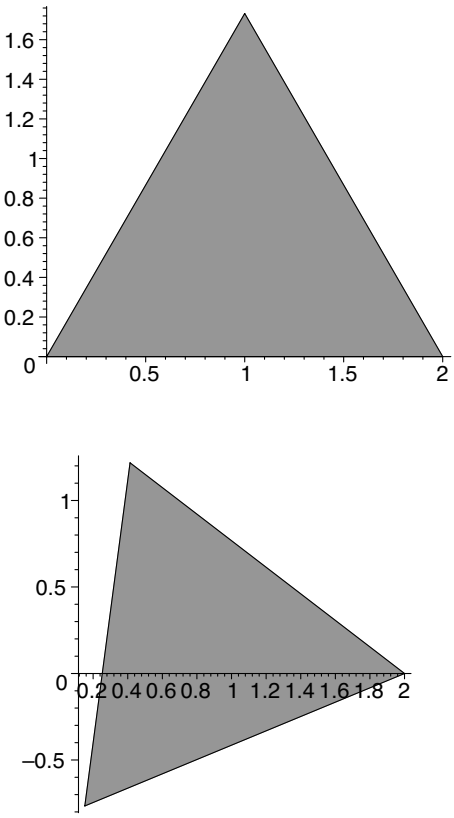
10.2 The `rotate` procedure

For rotations in two or three dimensions, the `plottools` package contains a handy procedure. In its two-dimensional form, the `rotate` procedure rotates an object counterclockwise about a point. The syntax is

`rotate(` P`,` α`, [`x_0`,`y_0`] )`

which rotates the structure P counterclockwise α radians about the point (x_0, y_0). For example,

```
> with( plottools ):
> with( plots ):
> T := polygon( [[0,0], [2,0], [1,sqrt(3)]], color=green ):
> display( T, scaling=constrained );
> display( rotate( T, Pi/8, [2,0] ), scaling=constrained );
```

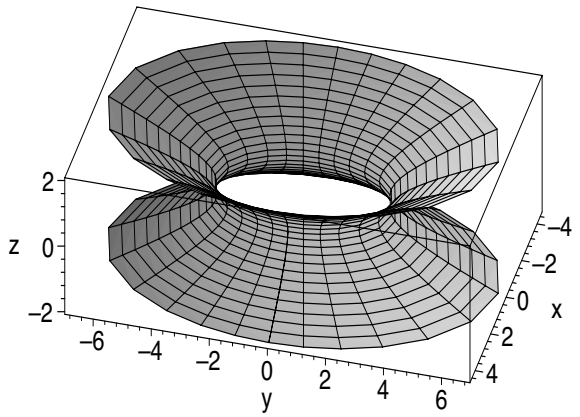


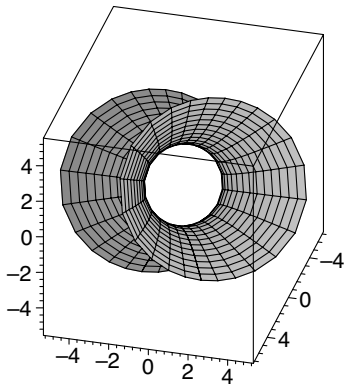
The `rotate` procedure has two forms in three dimensions. The form

`rotate(` P`,` α`,` β`,` γ `)`

rotates the object P through α radians about the x-axis, β radians about the y-axis, and γ radians about the z-axis. The left-hand rule determines the direction of the rotation. (If your left thumb points in the positive direction of the axis, your fingers curl in the direction of the rotation.) As an example, we can use this procedure to get a high-quality plot of a quadric surface with an oblique axis. We created superior plots of quadric surfaces with axis the z-axis in Sections 3.5, 3.6, 3.7, and 3.8, so we can just rotate one of those. We will rotate the hyperboloid of one sheet that we created in Section 3.8 using `cylinderplot`.

```
> with( plots ):
> with( plottools ):
> r := (theta,z) ->
  6*sqrt((z^2+1)/(9*(cos(theta))^2+4*(sin(theta))^2)):
> Hyperboloid := cylinderplot( r(theta,z), theta=0..2*Pi,
  z=-2..2, axes=boxed, scaling=constrained ):
> Hyperboloid;
> rotate( Hyperboloid, Pi/3, -Pi/3, 0 );
```



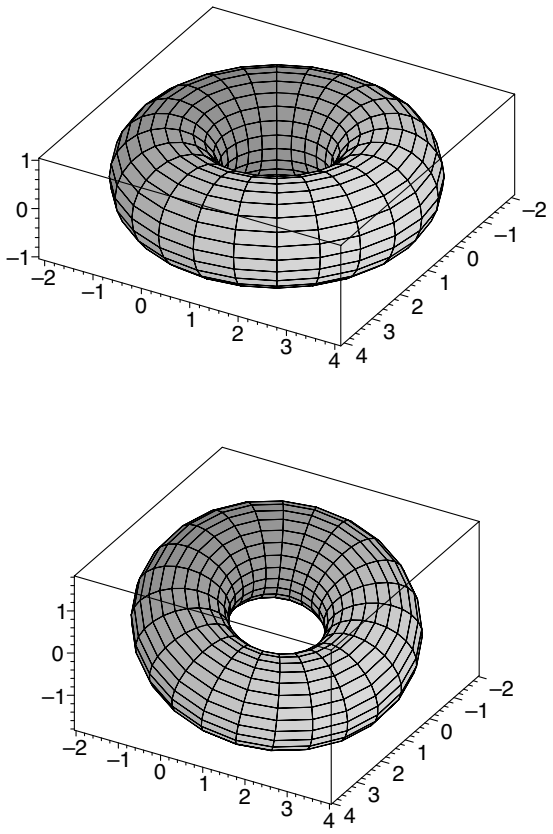


The other form is

`rotate(` P`,` α`, [[`x_1`,`y_1`,`z_1`],[`x_2`,`y_2`,`z_2`]] )`

which rotates the structure P through α radians about the line determined by the points (x_1, y_1, z_1) and (x_2, y_2, z_2). The positive direction (for determining the left-hand rule) along this line is from (x_2, y_2, z_2) toward (x_1, y_1, z_1). For example,

```
> with( plots ):
> with( plottools ):
> T := display( torus( [1,1,0] ), scaling=constrained,
  axes=boxed ):
> T;
> rotate( T, Pi/8, [[2,0,0],[0,2,0]] );
```



which plots a torus centered at the point $(1, 1, 0)$, then plots it rotated $\pi/8$ radians about the line through the points $(0, 2, 0)$ and $(2, 0, 0)$.

10.3 The `transform` procedure

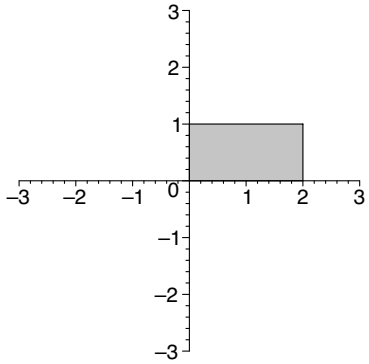
The `transform` procedure is a general one that applies, to all the points of a plot structure, a transformation defined by a mapping from $\mathbb{R}^m$ to $\mathbb{R}^n$, where m and n are either 2 or 3. The syntax of the mapping is to send each ordered pair or triple to a list of length two or three. So the procedure takes the form

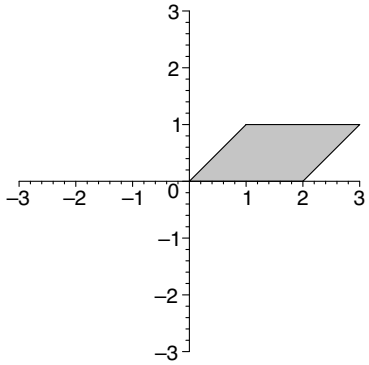
`transform(` *ordered pair or triple* `->` *list of two or three* `)`

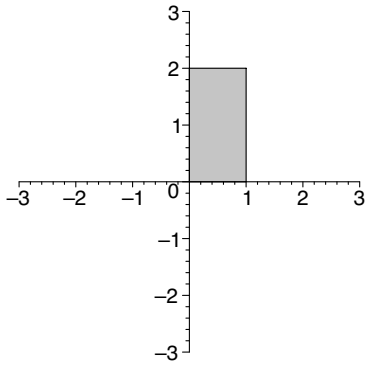
For example, the shear that sends each point (x, y) in the plane to the point $(x+y, y)$ can be represented by `transform( (x,y)->[x+y,y] )`. The reflection in the line $y = x$ can be written `transform( (x,y)->[y,x] )`. To illustrate the effects, we will plot a rectangle and its image under those transformations.

```
> with( plots ):
> with( plottools ):
> setoptions( view=[-3..3,-3..3], scaling=constrained ):
```

```
> rect := polygon( [[0,0], [2,0], [2,1], [0,1]],
  color=plum ):
> Shear := transform( (x,y) -> [x+y,y] ):
> Refl := transform( (x,y) -> [y,x] ):
> display( rect );
> display( Shear(rect) );
> display( Refl(rect) );
```

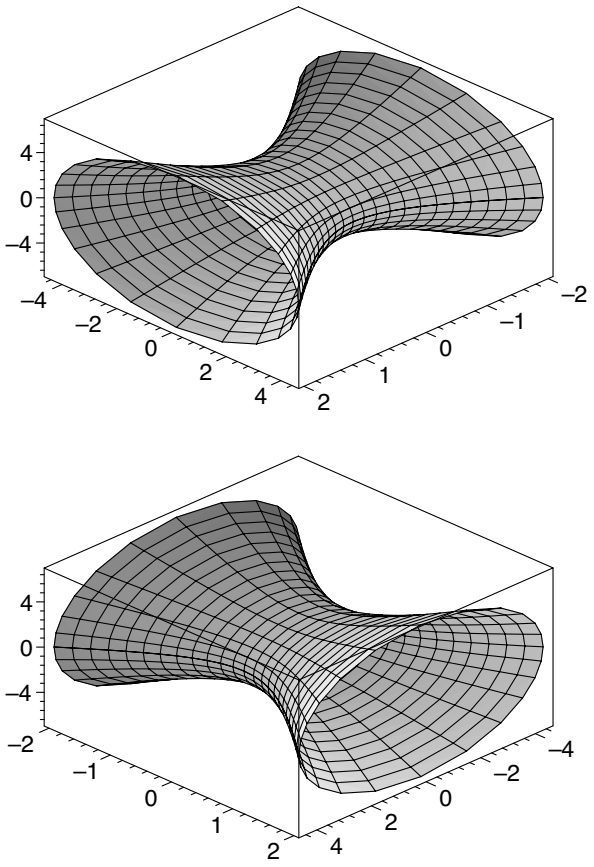






We can also use `transform` to reorient a three-dimensional plot by exchanging the axes. As an example, we will again use the plot from Section 3.8 of the hyperboloid of one sheet with axis the z-axis and use `transform` to create plots whose axes are the x- and y-axes.

```
> with( plots ):
> with( plottools ):
> r := (theta,z) ->
  6*sqrt((z^2+1)/(9*(cos(theta))^2+4*(sin(theta))^2)):
> Hyperboloid := cylinderplot( r(theta,z), theta=0..2*Pi,
  z=-2..2, axes=boxed ):
> T1 := transform( (x,y,z) -> [z,x,y] ):
> T2 := transform( (x,y,z) -> [x,z,y] ):
> T1(Hyperboloid);
> T2(Hyperboloid);
```



10.4 Matrix transformations

Although the transformation defined by `transform` need not be a linear one, whenever it is, we can express the transformation in the standard way—by using a matrix. A convenient way to create an $m \times n$ matrix in Maple is by columns:

$$\texttt{<}\,\texttt{<}a_{11}\texttt{,}a_{21}\texttt{,}\ldots\texttt{,}a_{m1}\texttt{>}\;\texttt{|}\;\texttt{<}a_{12}\texttt{,}a_{22}\texttt{,}\ldots\texttt{,}a_{m2}\texttt{>}\;\texttt{|}\;\cdots\;\texttt{|}\;\texttt{<}a_{1n}\texttt{,}a_{2n}\texttt{,}\ldots\texttt{,}a_{mn}\texttt{>}\,\texttt{>}$$

(Recall from Section 8.2 that <x_1,x_2,...,x_m> is a column vector.) For example,

```
> M := < <1,2> | <3,4> | <5,6> >;
```

$$M := \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

We can accomplish multiplication of a matrix and a vector using the "." operator, which we used in Section 8.3 for the dot product of two vectors. For example,

```
> A := < <1,2> | <3,4> >;
> X := <x,y>;
> A.X;
```

$$A := \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$X := \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\begin{bmatrix} x + 3\,y \\ 2\,x + 4\,y \end{bmatrix}$$

The "." operator is a talented one, accepting several types of arguments. For information, type `?.` at the Maple prompt.

We can effect the shear and reflection in the previous section, then, by using two matrices S and R, respectively, as follows:

```
> S := < <1,0> | <1,1> >;
> S.<x,y>;
```

$$S := \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} x + y \\ y \end{bmatrix}$$

and

```
> R := < <0,1> | <1,0> >;
> R.<x,y>;
```

$$R := \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} y \\ x \end{bmatrix}$$

The result, as expected, is a column vector. The `transform` procedure wants the result (that is, the image under the transformation) to be a list. So, whenever we want to use matrix notation for a linear transformation, we need to convert the result to a list, so as not to disappoint `transform`. This is easily done with the `convert` procedure.

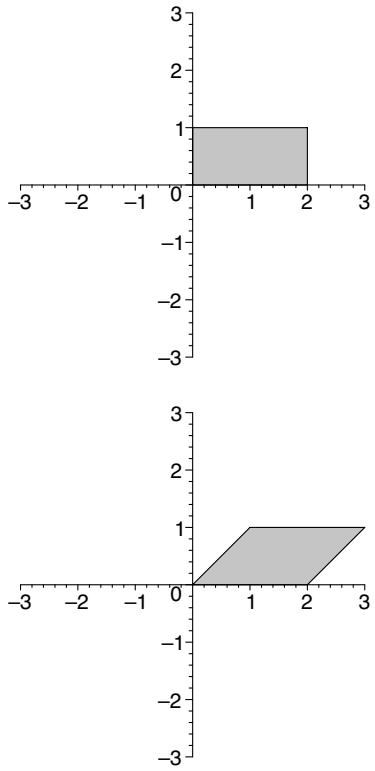
```
> convert( S.<x,y>, list );
> convert( R.<x,y>, list );
```

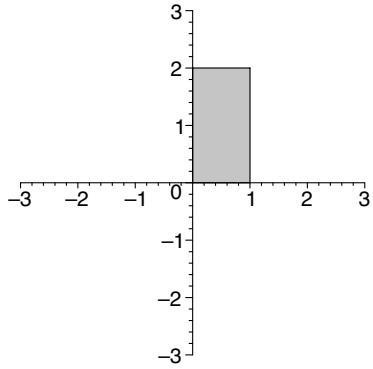
$$[x+y,\, y]$$

$$[y,\, x]$$

Using matrix notation, then, we could have written the linear transformations in the previous section as

```
> with( plots ):
> with( plottools ):
> setoptions( view=[-3..3,-3..3], scaling=constrained ):
> rect := polygon( [[0,0], [2,0], [2,1], [0,1]],
  color=plum ):
> S := < <1,0> | <1,1> >:
> Shear := transform( (x,y) -> convert( S.<x,y>, list ) ):
> R := < <0,1> | <1,0> >:
> Refl := transform( (x,y) -> convert( R.<x,y>, list ) ):
> display( rect );
> display( Shear(rect) );
> display( Refl(rect) );
```





10.5 Morphing

To *morph* one plot into another is to display a sequence of frames that depict a gradual transition from one plot to the other. We can think of this as a mapping, and we can accomplish it by moving each point in the first plot along the line segment that connects the point to its image in the second plot. In two dimensions, then, to send the point (x_1, y_1) to the point (x_2, y_2), we can plot points $((1-k)\,x_1 + k x_2, (1-k)\,y_1 + k y_2)$ for values of k from 0 to 1. This is the essence of morphing. We have a mixture of (x_1, y_1) and (x_2, y_2). We start out with all (x_1, y_1) and no (x_2, y_2) in the mix. As k increases from 0 to 1, we get less (x_1, y_1) and more (x_2, y_2), until we end with no (x_1, y_1) and all (x_2, y_2).

To illustrate the method, we will morph the curve (the evolute of an ellipse) defined parametrically by

$$\begin{aligned} x_1(t) &= 2\cos^3 t \\ y_1(t) &= 3\sin^3 t, \quad 0 \le t \le 2\pi \end{aligned}$$

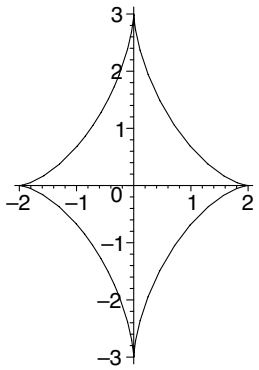
into the curve (a nephroid) defined by

$$\begin{aligned} x_2(t) &= 3\cos t - \cos 3t \\ y_2(t) &= 3\sin t - \sin 3t, \quad 0 \le t \le 2\pi \end{aligned}$$

The mapping, here, is just $(x_1(t), y_1(t)) \mapsto (x_2(t), y_2(t))$. Our first plot is

```
> x1 := t -> 2*cos(t)^3;
> y1 := t -> 3*sin(t)^3;
> plot( [x1(t),y1(t), t=0..2*Pi], scaling=constrained );
```

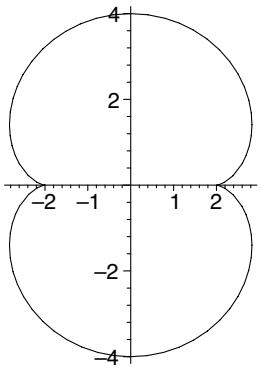
$$\begin{aligned} x1 &:= t \to 2\cos(t)^3 \\ y1 &:= t \to 3\sin(t)^3 \end{aligned}$$



and the second plot is

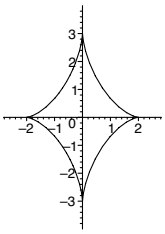
```
> x2 := t -> 3*cos(t)-cos(3*t);
> y2 := t -> 3*sin(t)-sin(3*t);
> plot( [x2(t),y2(t), t=0..2*Pi], scaling=constrained );
```

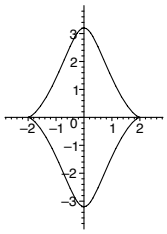
$$x2 := t \to 3\cos(t) - \cos(3\,t)$$
$$y2 := t \to 3\sin(t) - \sin(3\,t)$$

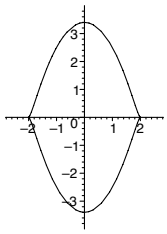


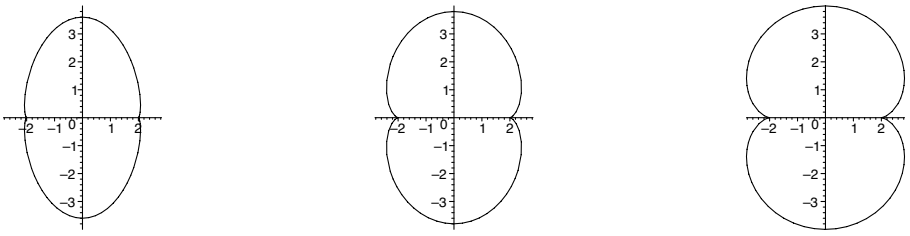
To accomplish the morphing, we can use `animate`.

```
> with( plots ):
> animate( [(1-k)*x1(t)+k*x2(t),(1-k)*y1(t)+k*y2(t),
  t=0..2*Pi], k=0..1, scaling=constrained );
```









10.6 Linear transformations

The study of linear transformations offers an opportunity to add geometric fullness to the algebraic beauty of linear algebra. Typically, we illustrate linear transformations only in $\mathbb{R}^2$, and do so by showing a square at the origin and, next to it, its image, as shown in Figure 10.1. But if our students are

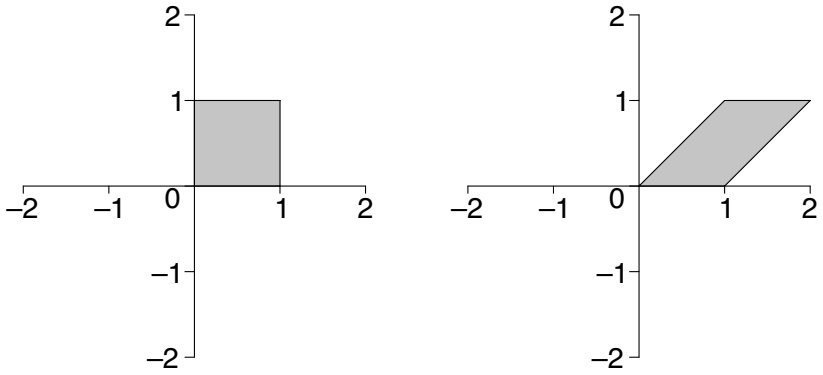


FIGURE 10.1: Square and its image under a linear transformation

really to understand the geometry of linear transformations, then they should understand this in the richer context of $\mathbb{R}^3$. We will create demonstrations to illustrate linear transformations in both $\mathbb{R}^2$ and $\mathbb{R}^3$. Although we could just show an object, then have it snap to its image under the transformation, this would not demonstrate effectively *how* it becomes transformed. A better way is to cause the object to morph into its image. This way, students can see the space transforming, not just the space transformed. Moreover, they can follow each basis vector as it undergoes the transformation.

10.6.1 Demonstrations: Linear transformations of $\mathbb{R}^2$

In this demonstration, we will show an object undergoing a linear transformation of $\mathbb{R}^2$ defined by the matrix $\mathbf{A}$. To do this, we will combine the `transform` procedure with morphing. We begin by calling the packages that we need, choosing some options, and creating an object to transform.

```
> restart:
> with( plots ):
> with( plottools ):
> setoptions( style=patch, view=[-3..3,-3..3],
  scaling=constrained, axesfont=[HELVETICA,18] ):
> rect := polygon( [[0,0], [2,0], [2,1], [0,1]],
  color=plum ):
```

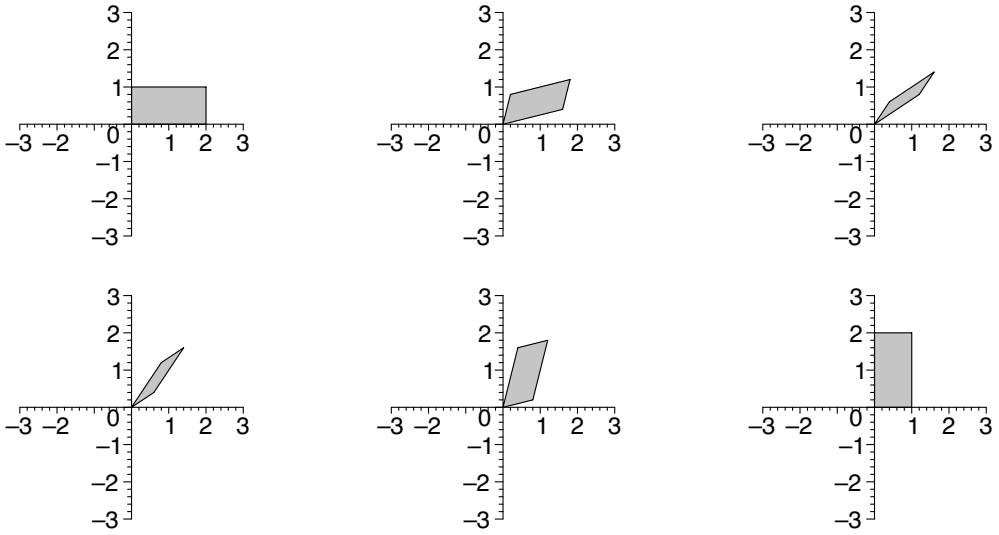
Next, we choose a number of frames (beyond the first) N and create two functions. The function F performs the morphing. It varies the mix of $[x, y]^T$ and $\mathbf{A}[x, y]^T$ as k ranges from 0 to N. The function L carries out the transformation.

```
> N := 20:
> F := (x,y) -> (1-k/N)*<x,y> + k/N*(A.<x,y>):
> L := transform( (x,y) -> convert( F(x,y), list ) ):
```

We choose an example matrix to represent the linear transformation, in this case the reflection in the line $y = x$, then create and display a sequence of frames.

```
> A := < <0,1> | <1,0> >;
> Frames := seq( L(rect), k = 0..N ):
> display( Frames, insequence=true );
```

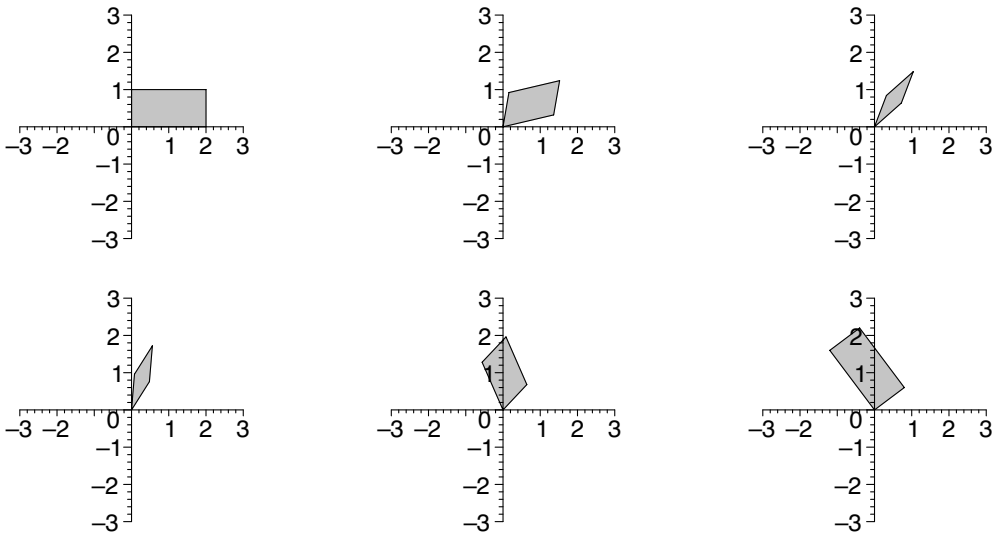
$$A := \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$



Let's generalize this example. We create a matrix for reflection in the line $y = mx$ so that we can easily change the slope m to experiment a little with reflections through various lines.

```
> A := 1/(1+m^2) * < <1-m^2,2*m> | <2*m,m^2-1> >;
> m := 2:
> Frames := seq( L(rect), k = 0..N ):
> display( Frames, insequence=true );
```

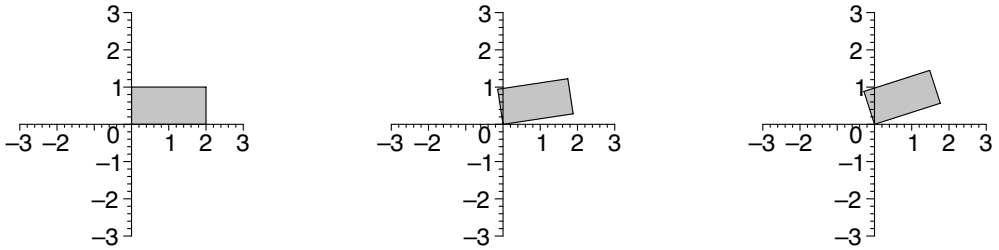
$$A := \frac{\begin{bmatrix} 1-m^2 & 2\,m \\ 2\,m & m^2-1 \end{bmatrix}}{1+m^2}$$

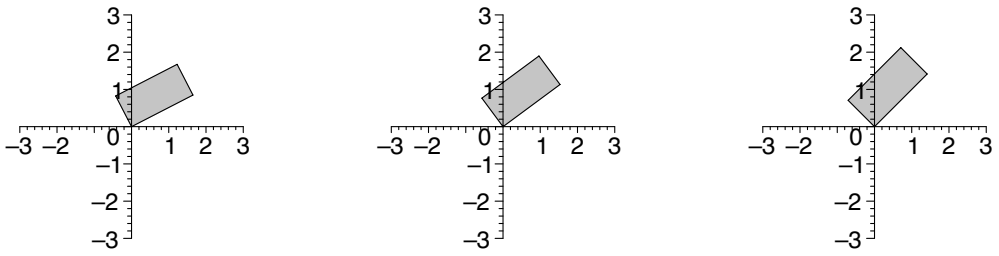


Finally, let's change the linear transformation to a rotation through θ about the origin. We just change the matrix **A**.

```
> A :=
  < <cos(theta),sin(theta)> | <-sin(theta),cos(theta)> >;
> theta := Pi/4:
> Frames := seq( L(rect), k = 0..N ):
> display( Frames, insequence=true );
```

$$A := \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$





10.6.2 Demonstrations: Linear transformations of $\mathbb{R}^3$

Let's adapt our previous demonstration for use in three dimensions. This time, we will want to label the axes to make the orientation of the plot clearer. Labels for the x- and y-axes will be enough.

```
> restart:
> with( plots ):
> with( plottools ):
> setoptions3d( style=patch, axes=normal,
  scaling=constrained, labels=["x","y",""],
  labelfont=[TIMES,BOLDITALIC,24],
  axesfont=[HELVETICA,18] ):
```

We need an object to transform. The obelisk that we created in Section 10.1 above will work well.

```
> P1 := [0,0,0]:
> P2 := [1,0,0]:
> P3 := [1,1,0]:
> P4 := [0,1,0]:
> P5 := [0,0,1]:
> P6 := [1,0,1]:
> P7 := [1,1,1]:
> P8 := [0,1,1]:
> P9 := [.5,.5,1.4]:
> face1 := polygon( [P1,P2,P3,P4], color=navy ):
> face2 := polygon( [P1,P2,P6,P5], color=turquoise ):
> face3 := polygon( [P2,P3,P7,P6], color=aquamarine ):
> face4 := polygon( [P3,P4,P8,P7], color=sienna ):
> face5 := polygon( [P4,P1,P5,P8], color=gray ):
> face6 := polygon( [P5,P6,P9], color=maroon ):
> face7 := polygon( [P6,P7,P9], color=tan ):
> face8 := polygon( [P7,P8,P9], color=wheat ):
> face9 := polygon( [P8,P5,P9], color=khaki ):
> obelisk := display( face1, face2, face3, face4, face5,
  face6, face7, face8, face9 ):
```

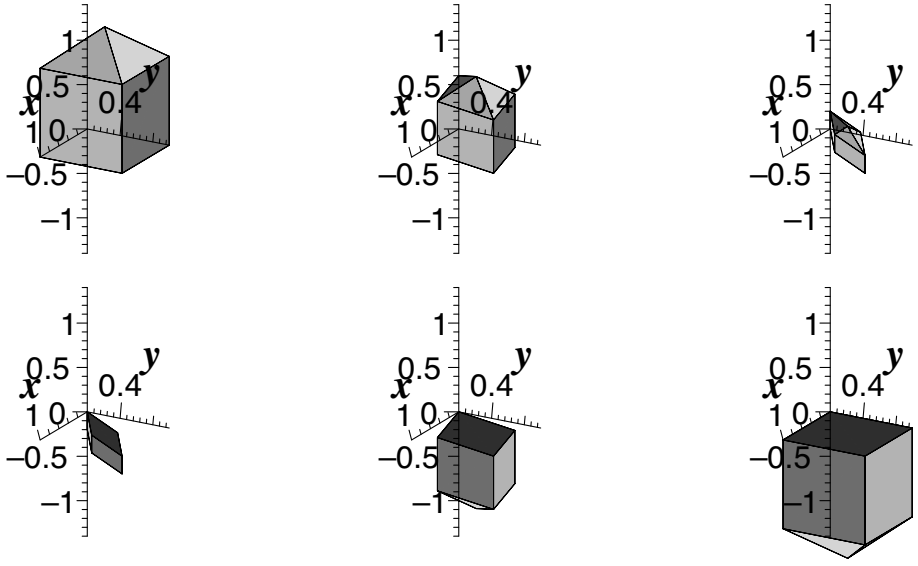
We readily adapt the functions F and L for three dimensions. We will keep the number of frames the same.

```
> N := 20:
> F := (x,y,z) -> (1-k/N)*<x,y,z> + k/N*(A.<x,y,z>):
> L := transform( (x,y,z) -> convert( F(x,y,z), list ) ):
```

We choose a linear transformation to use as an example. This one is the composition of a reflection in the xy-plane and a reflection in the vertical plane $y = x$.

```
> A := < <0,1,0> | <1,0,0> | <0,0,-1> >;
> Frames := seq( L(obelisk), k = 0..N ):
> display( Frames, insequence=true );
```

$$A := \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$



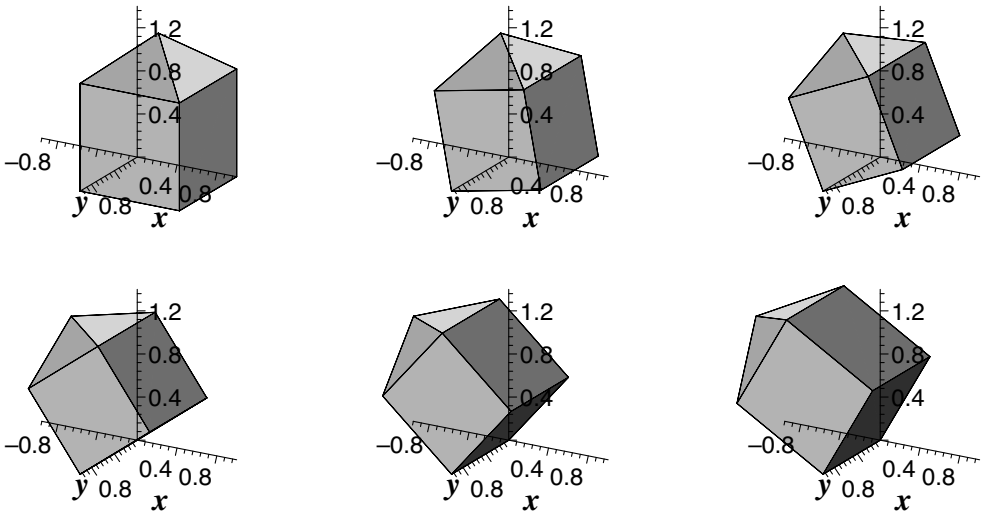
For demonstrating other linear transformations, it will be useful to have a few standard ones on hand. These are the matrices for rotations through θ (according to the right-hand rule) about the coordinate axes. We will set them as functions of θ so that, for example, $RotX(\theta)$ will be the matrix representing the rotation through θ about the x-axis.

```
> RotX := theta -> < <1,0,0> | <0,cos(theta),sin(theta)> |
  <0,-sin(theta),cos(theta)> >:
> RotY := theta -> < <cos(theta),0,-sin(theta)> | <0,1,0> |
  <sin(theta),0,cos(theta)> >:
> RotZ := theta -> < <cos(theta),sin(theta),0> |
  <-sin(theta),cos(theta),0> | <0,0,1> >:
```

We can use them to demonstrate, for example, a rotation through $\pi/3$ about the x-axis:

```
> A := RotX(Pi/3);
> Frames := seq( L(obelisk), k = 0..N ):
> display( Frames, insequence=true );
```

$$A := \begin{bmatrix} 1 & 0 & 0 \\ 0 & \frac{1}{2} & -\frac{\sqrt{3}}{2} \\ 0 & \frac{\sqrt{3}}{2} & \frac{1}{2} \end{bmatrix}$$

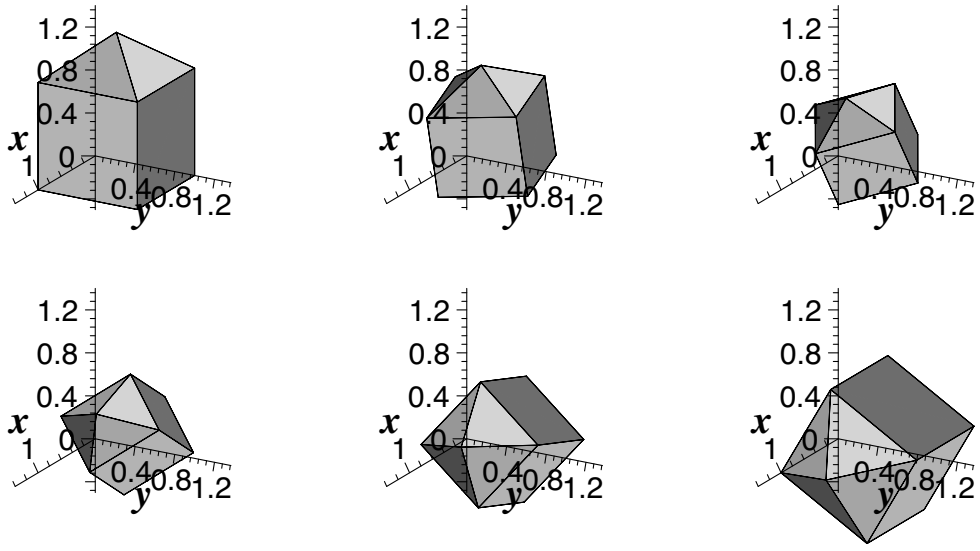


(The axis labels land in places that are less than ideal. This must be a difficult problem.)

A product of these matrices will, of course, represent the composition of the corresponding linear transformations. For example, the composition of a rotation through $\pi/2$ about the y-axis and a rotation through $\pi/3$ about the x-axis is

```
> A := RotX(Pi/3).RotY(Pi/2);
> Frames := seq( L(obelisk), k = 0..N ):
> display( Frames, insequence=true );
```

$$A := \begin{bmatrix} 0 & 0 & 1 \\ \frac{\sqrt{3}}{2} & \frac{1}{2} & 0 \\ \frac{-1}{2} & \frac{\sqrt{3}}{2} & 0 \end{bmatrix}$$



These demonstrations give students something concrete to tie to the concept of linear transformation. I think it is important and motivating for students to have these mental images from the start, so I use the demonstrations early. Just after I have defined linear transformations and shown that every matrix transformation is linear, I use the demonstration for $\mathbb{R}^2$. I show two or three examples—usually reflections in lines and rotations about the origin. I run the animations several times and let the students get the geometry well in mind.

After we have developed more theory, which is soon, I use the demonstration for $\mathbb{R}^3$. As background, the students need to understand that the columns of the transformation matrix are the images of the standard basis vectors. I use the demonstrations to show some examples, usually some rotations about axes and then compositions of them. I point out that the standard basis vectors are represented as the edges of the obelisk that lie along the coordinate axes. I suggest to students that they focus on those edges as they watch the space transforming. After that, I let the students decide what should become of each standard basis vector. I enter these into the worksheet as the columns of $\mathbf{A}$, and we watch as each basis vector morphs to its image. It is a powerful demonstration that linear transformations can be created at will in this way. It is also useful to point out that, since every basis vector must go somewhere, every linear transformation must have a transformation matrix.

Bibliography

[1] Abbott, E.A., with a new introduction by Banchoff, T., *Flatland: A Romance of Many Dimensions*, Princeton University Press, Princeton, 1991.

[2] Carroll, L., "Jabberwocky," in *The Complete Works of Lewis Carroll*, illustrated by Tenniel, J., introduction by Woollcott, A., Modern Library, New York, c. 1970, 153–155.

[3] Coxeter, H.S.M., Alicia Boole Stott, in *Women of Mathematics: A Biobibliographic Sourcebook*, Grinstein, L.S. and Campbell, P.J., Eds., Greenwood Press, New York, 1987, 220–224.

[4] Kline, M., *Mathematical Thought from Ancient to Modern Times*, Princeton University Press, New York, 1972.

[5] Monagan, M.B. et al., *Maple 7 Programming Guide*, Waterloo Maple Inc., Waterloo, 2001.

[6] Monagan, M.B. et al., *Maple 8 Advanced Programming Guide*, Waterloo Maple Inc., Waterloo, 2002.

[7] Monagan, M.B. et al., *Maple 8 Introductory Programming Guide*, Waterloo Maple Inc., Waterloo, 2002.

[8] Waterloo Maple, based in part on the work of Char, B.W., *Maple 7 Learning Guide*, Waterloo Maple Inc., Waterloo, 2001.

[9] Waterloo Maple, based in part on the work of Char, B.W., *Maple 8 Learning Guide*, Waterloo Maple Inc., Waterloo, 2002.