Useful *Mathematica* Commands A Summary for MATH 166

Mathematics Department,
Iowa State University
Fall 1996

Introduction
This document describes the *Mathematica* commands and options that you may find useful for your assignments in this class.
The commands covered are listed in the Table of Contents.
If you are using the HTML document, and you have registered properly, you will be given a username and password, which will allow you to use the document interactively. For details, see The HTML Document.

1 Table of Contents

Table of Contents:

- FindRoot
- GridLines
- Infinite Series
- Integrate
- Limit
- ListPlot
- Lists of Data
- N (Numeric value)
- NDSolve
- NIntegrate
- NSolve
- NSum
- ParametricPlot
- Plot
- Plot and D Don't Mix
- PlotLabel
- PlotRange
- Polar Grids
- PolarPlot
- Prime
- Print
- RGBColor
- Sampling Plot Coordinates
- Sequences
- Series
- Show
- Simple Functions
- Simplify
- Solve
- Special Constants
- Special Functions

- Sum
- SurfaceOfRevolution
- Table
- TableForm
- Together

2 The Mathematica Notebooks

One version of this document is available in the format of *Mathematica* notebooks. The explanatory part appears in text cells, while the *Mathematica* commands being discussed appear in command cells.
These commands may be easily executed simply by clicking in a particular command cell and pressing the **Enter** key, or holding down the **Shift** key and pressing **Return**.
The output from such a command will appear just below the command. You can easily modify any of the commands, or type in new commands that you wish to try. You may print all or a portion of the document, using the *Mathematica* **Print** command from the **File** menu at the top of the window.

2.1 Accessing the Notebooks

The *Mathematica* version of this document is available as several notebook files in the Mathematics Department Computer Lab, which is open to all ISU students.
To access the notebooks, you first need to "discard" any old connection to the Mathematics Delta Server, if someone else accessed it before you. Look along the right margin of the Macintosh Desktop for a little object labeled "Delta Server". If you find it, place your cursor on its icon, press and hold the mouse button, and drag the cursor (and the Server) into the trash can. Release the mouse button to discard the old connection to the server.
Secondly, you need to make a new connection to the Delta Server, which recognizes that you are entitled to access, and are in Math 166. Go to the "Apple" menu in the upper left corner of the screen, and select **Chooser**. Now click on **AppleShare**. AppleShare should show that you are in the "MathLab" zone. Now click on the **Delta Server**. The Macintosh will request authentication. The instructor for your calculus class will have told you what to type here. Once you have been authenticated, a new symbol should appear on the desktop, representing the Delta Server.
Now, to get one of the files, open the Delta Server (that is, click twice on its icon). Open the **Classes** folder, then **Math 166**, and finally **Labs**. The files **commands1**, **commands21** and **commands32** should be visible now. You should copy each file to the local hard disk, by clicking once on its icon and dragging it to a free area of the desktop.
To examine the contents of one of these files, click twice on one of them. This will cause *Mathematica* to begin running, and displaying the information in the file.

3 The HTML Document

One version of this document has been marked up using HTML, and can be accessed by an Internet broswer, such as NetScape. The document is broken down into subpages devoted to a particular command.
Through tools developed by the (Undergraduate Computational Engineering and Sciences Project) at Ames Laboratory, a person browsing through the HTML document can actually execute the *Mathematica* commands discussed, and see the numeric or graphic output in a small window below the main browser window.

3.1 Executing *Mathematica* in HTML

If you are authorized for the remote server, then to execute a *Mathematica* command, click on the colorful arrow icon that is just to the left of the command. The first time you do this, the following text should appear in the smaller box below the main one:

```
How do you prefer to run Mathematica:

  SUBMIT when you're done:

    O I have both hamlet and Mathematica on my machine.
    O I'll cut-n-paste the commands from my browser to Mathematica.
    O Use a remote server (requires a proper username and password).
```

Click on **Use a remote server** (that is, click on the little open diamond **O** in front of those words). The open diamond should change to a filled diamond.
Then click on **SUBMIT**. The remote computer will ask you for a username and password. Once you are authenticated, the results of your *Mathematica* commands will appear in the same box below the main one. You will not be able to enter new commands of your own, however.

4 Advanced Functions

The simplest user-defined functions are the "one-liners", where the quantity of interest can be computed by a single formula. Such formulas are discussed in Simple Functions**Simple Functions** ()Simple Functions.
In some cases, you may find it impossible to define the function's value in a single simple formula. Instead, you may need to carry out several steps of computation, using temporary variables.
You may want several input values, and you may want the user to group some of those input values in curly brackets.
Here is an example of a user-defined function, which computes the left Riemann sum approximation to the integral of a function **f[x]** from **a** to **b**, using **n** intervals:

```
Clear[f, leftRsum, x]
```

```
leftRsum[ f_, {x_, a_, b_, n_} ] :=
  Block[ {approx, delx, i},
    delx = (b-a)/n ;
    approx = N[ Sum[ (f /. x->a+delx*i )*delx, {i,0,n-1}] ]
  ]
```

The first line defines the name of the function to be **leftRsum**, with 5 arguments. The last four arguments must be enclosed in curly brackets.
The **Block** command marks the beginning of a group of commands that will carry out the procedure named **leftRsum**.
The statement **{approx, delx, i}** defines *local variables*, that is, names for variables which will be set and used within the procedure, but which are not function arguments, and which should not alter the values of variables with the same name, defined outside the procedure.
The first assignment command has a semicolon after it to suppress output. The value returned by the function will be the last value computed, namely, the value of **approx**..
The assignment of **approx** carries out the Riemann approximation, and causes the function **leftRsum** to return this value.
As an example of how such a routine might be used, consider the following commands:

```
f[x_] := x^2
```

```
leftRsum[ f[x], {x, 0,  10, 5} ]
```

For other examples of this kind of function, see the discussion of Cube Roots**Cube Roots** ()Cube Roots or Polar Grids**Polar Grids** ()Polar Grids.

5 Animation

Warning: It is not possible to try out the animation feature in the HTML document!
If you draw a sequence of plots, you can have *Mathematica* animate the plots, that is, display them one after another, as though they were individual frames of a movie.
To make an animation, you must first generate a sequence of plots, using any of the usual graphics commands, such as ListPlot**ListPlot** ()ListPlot, ParametricPlot**ParametricPlot** ()ParametricPlot or Plot**Plot** ()Plot. You should try to make sure that each "frame" or plot has the same shape (what *Mathematica* calls the AspectRatio**AspectRatio** ()AspectRatio), and uses the same **x** intervals, and the same **y** intervals (which you can control with the option PlotRange**PlotRange** ()PlotRange).
Here is a very simple set of frames that show how the shape of the function $y = a * x^3 + x + 1$ will change as the coefficient a is varied.

```
Clear[a,f,p0,p1,p2,p3,p4,p5,x]
```

```
f[x_]:=a*x^3+x+1
```

```
a=0;
p0=Plot[f[x], {x,-2,2}, PlotRange->{-2,4}]
```

```
a=1;
p1=Plot[f[x], {x,-2,2}, PlotRange->{-2,4}]
```

```
a=2;
p2=Plot[f[x], {x,-2,2}, PlotRange->{-2,4}]
```

```
a=3;
p3=Plot[f[x], {x,-2,2}, PlotRange->{-2,4}]
```

```
a=4;
p4=Plot[f[x], {x,-2,2}, PlotRange->{-2,4}]
```

```
a=5;
p5=Plot[f[x], {x,-2,2}, PlotRange->{-2,4}]
```

If you are not using a notebook interface, it is possible that you can animate these plots using the command:

```
ShowAnimation[{p0,p1,p2,p3,p4,p5}]
```

In the notebook interface, we "select" the frames to be animated - that is, we place the cursor near the bracket on the right margin of the first plot and click once. Now hold down the shift key, and click in the right hand brackets of all the other plots you want to include in the animation.
Now go to the **Graph** menu and select **Animate Selected Graphics**. This should cause a brief animation of your plots to be displayed. It will probably be over too quickly for you to enjoy. In that case, go back to the **Graph** menu and select **Animation**. This will give you a menu that will allow you to specify how many frames per second should be displayed, and whether the animation should loop forward only, or forward and back again. A little experimentation should make a pleasing display.

6 Apart

A polynomial fraction $\frac{p(x)}{q(x)}$ is said to be *improper* if the degree of $p(x)$ is greater than or equal to that of $q(x)$.
The **Apart** command can be used to replace a polynomial fraction by a "whole" part, and a new fraction with the same denominator, but a "proper" numerator. This is similar to changing the "improper" fraction 17/5 to 3 + 2/5.
The **Apart** command, applied to a polynomial fraction, is essentially carrying out the method of synthetic division. Synthetic division is a useful simplification, and is *mandatory* before applying the method of Partial Fractions to an improper polynomial fraction.

```
Clear[x]
```

```
Apart[ (x^3 + 2*x^2 - x + 4) / (x^2 + x + 1) ]
```

In the next example, we can see that *Mathematica* is not only dividing out the improper part of the polynomial fraction, but is also factoring the denominator and applying the method of Partial Fractions:

```
Apart[ (x^5+4x^4+2x^3-7x^2+4x+1)
     / ( (x^2+1) * (x+3)^2 ) ]
```

The command Together**Together** ()Together reverses the operation of **Apart**.

7 AspectRatio

AspectRatio is an option that may be specified in the graphics commands ListPlot**ListPlot** ()ListPlot, ParametricPlot**ParametricPlot** ()ParametricPlot, Plot**Plot** ()Plot, and Show**Show** ()Show.
Suppose we want to plot a quarter circle and a 45 degree ray. We could execute the following *Mathematica* commands:

```
Clear[f, g, x]
```

```
f[x_]:=Sqrt[1-x^2]
```

```
g[x_]:=x
```

```
Plot[ { f[x], g[x]}, {x,0,1}]
```

Unfortunately, the curve doesn't look like part of a circle, nor is the ray at 45 degrees, because *Mathematica* has chosen different **x** and **y** scales. We can force these scales to correspond to the relative lengths signified by the actual coordinates by specifying the **AspectRatio -> Automatic** option:

```
f[x_]:=Sqrt[1-x^2]
```

```
g[x_]:=x
```

```
Plot[ { f[x], g[x]}, {x,0,1} , AspectRatio->Automatic]
```

You may also specify a numeric value for the **AspectRatio**. For instance, setting **AspectRatio -> 2** makes a new plot that is twice as high as the actual data would justify.

8 Assignments

The easiest way to give a variable a value uses a single equal sign, as in **x = 1**. This causes *Mathematica* to find the value of the right hand side of the equals sign, and assign that value to the variable named on the left hand side.

```
Clear[x, y]
```

```
x = 2;
```

```
y = 10*x+1
```

When a single equals sign is used in an assignment, then the variable on the left hand side is given a value immediately. If the quantities on the right hand side are later changed, this will not affect the left hand side. For instance, let's set **x** to 1, and set **y** to be two times **x**, using a single equals sign.

```
x = 1;
```

```
y = 2*x;
```

```
x
```

```
y
```

Now let's change **x** to 10. Does **y** change to 20, or stay at 2?

```
x = 10;
```

```
x
```

```
y
```

If you want to specify that **y** should change whenever **x** changes, then you need to replace the single equals sign by the combination "colon-equals".

```
x = 1;
```

```
y := 2*x;
```

```
x
```

```
y
```

Now, when we change **x** to 10, the value of **y** will change as well:

```
x = 10;
```

```
x
```

```
y
```

The double equals sign is used when describing a mathematical equation, rather than an assignment. Thus, to ask the command Solve**Solve** ()Solve to find the values of **x** that make a certain equation true, we write

Solve[4x+2y==3z-1, x].

The double equals sign tells **Mathematica** that we are not trying to assign a value to the quantity **4x+2y**, whereas a single equals sign would confuse it.
Often you don't want to assign a single value, but to set up a functional relationship. To see how to do this, refer to Simple Functions**Simple Functions** ()Simple Functions.
To see how to assign values to a vector or matrix, refer to Lists of Data**Lists of Data** ()Lists of Data.

9 AxesLabel

AxesLabel is an option that may be specified in the graphics commands ListPlot**ListPlot** ()ListPlot, ParametricPlot**ParametricPlot** ()ParametricPlot, Plot**Plot** ()Plot, and Show**Show** ()Show.
AxesLabel allows you to label the **x** and **y** axes of your plot.

```
Clear[t]
```

```
Plot[ 1-Exp[-t], {t,0,1},
  AxesLabel->{"Time", "Radiation"} ]
```

10 AxesOrigin

AxesOrigin is an option that may be specified in the graphics commands ListPlot**ListPlot** ()ListPlot, ParametricPlot**ParametricPlot** ()ParametricPlot, Plot**Plot** ()Plot, and Show**Show** ()Show.
Sometimes, *Mathematica* will not include the origin in your plot:

```
Clear[x]
```

```
Plot[Sin[x]+4, {x,1,10}]
```

If you want the point (0,0) included, with both axes going through it (or at least pointing towards it) then you might use the **AxesOrigin** option:.

```
Plot[Sin[x]+4, {x,1,10}, AxesOrigin->{0,0},
  PlotLabel->"Same plot, with AxesOrigin command" ]
```

Of course, a sneaky way to force the **x** axis to show up is simply to include the line **y=0** in your graph. The funny thing is, this graph is a lot nicer than the previous one!

```
Plot[ { Sin[x]+4, 0}, {x,1,10} ]
```

11 Clear

The **Clear** command tells *Mathematica* to "forget" anything you may have said about a variable or formula, allowing you to redefine it or use it in a new way. This is important, because the commands D (differentiate)**D** ()D, Integrate**Integrate** ()Integrate, and Plot**Plot** ()Plot will not work properly if the "x" argument has been previously assigned a numerical value. Here is a sample **Clear** command:

```
Clear[x]
```

Here is a case where **Clear** is necessary. Because **x** is set to 1, the**D** command cannot be carried out until a **Clear** command "frees up" **x**:

```
x=1
```

```
D[Sin[x],x]
```

```
Clear[x]
```

```
D[Sin[x], x]
```

In this document, we use **Clear** as the first command in every example, so that variables left over from other examples won't affect the current example.

12 Comments

When you are using *Mathematica* to solve a homework problem, it can be important to explain what you're doing, or to "annotate" your results. There are three simple things you can do.
First, you can use text cells. A text cell is a part of a *Mathematica* notebook that is set aside for text, rather than commands. To make a text cell, you simply start a new cell, and then select the cell in the usual way, by clicking in the little cell marker in the right hand margin. Then, go to the **Style** menu, then to the **Cell Style** submenu, and choose **Text**. From now on, anything you type in this particular cell will be considered text. Mathematica will print it in plain style rather than boldface, and won't try to execute your sentences as commands.
The second thing you can do is simpler, though less pretty. You can cause *Mathematica* to ignore a piece of text by marking it as a "comment". You begin a comment with the symbols **(*** and end with ***)**. Thus, *Mathematica* won't be confused by the following commands:

```
Clear[area, r]
```

```
r=4
```

```
(* This command computes the area of the circle! *)
area = Pi * r^2
```

The third thing you can do to comment your commands is simply to use short strings of text, marked off by double quotes, to add labels or explanations to the numbers you compute. In this case, the quoted string will actually be printed out right next to the number you compute.

```
area = N[Pi * r^2] " is the area of the circle."
```

This is not always a great solution. For one thing, you can't do any more calculations with **area**, because its value is **not** a number; it's a number followed by a string. For another, *Mathematica* has many rules for rearranging results that may hurt your intended output. For instance:

```
area = Pi * r^2 " is the area of the circle."
```

13 Cube Roots

We know that negative numbers don't have square roots - at least not in the real number system, so we are not surprised that *Mathematica* will not plot the **Sqrt** function over a negative range, and returns a complex value for the square root of -3.
However, we do expect that negative numbers have a cube root, and so it is very surprising to see that *Mathematica* has a strange idea about this:

```
N[ (-1)^(1/3) ]
```

And because *Mathematica* returns complex values for such cube roots, you can't plot the cube root function the way you might like:

```
Plot[ x^(1/3), {x,-2,2}]
```

However, it is fairly easy to set up a function that returns the cube root we expect:

```
Clear[cuberoot]
```

```
cuberoot[ x_ ] :=
  Block[ {value},
    value = Sign[x] * (Abs[x])^(1/3)
  ]
```

This function allows us to compute the cube roots we expect:

```
cuberoot[-1]
```

and to plot the whole function:

```
Plot[cuberoot[x], {x,-2,2} ]
```

The same problem occurs for all odd roots of negative numbers, and can be corrected similarly.

14 D (Differentiation)

The command **D[f[x], x]** will differentiate the formula or expression **f[x]** with respect to the variable **x**.

```
Clear[f, x]
```

```
D[ x^2+2*x+1, x]
```

```
f[x_]:=Cos[x]+x
```

```
D[f[x],x]
```

You can also compute higher order derivatives:

```
D[ x^2+2*x+1, {x,2} ]
```

The **D** command will not work properly if the (symbolic) variable of differentiation has been assigned a numeric value. In such cases, you must first use the command Clear**Clear** ()Clear, to "free up" the variable.

```
x=1
```

```
D[Cos[x],x]
```

```
Clear[x]
```

```
D[Cos[x],x]
```

If you have written your expression as a function of a variable, then you may also compute the derivative using an apostrophe or "prime", while the second derivative uses two primes:

```
f[x_]:=Cos[x]+x
```

```
f'[x]
```

```
f''[x]
```

You can use **f'[x]** in the commands Plot**Plot** ()Plot or Solve**Solve** ()Solve.

```
f[x_]:=Sin[x]
```

```
Plot[ f'[x], {x, 0, Pi} ]
```

You may be interested in seeing the example Plot and D Don't Mix**Plot and D Don't Mix** ()Plot and D Don't Mix.
Table of Contents**Table of Contents** ()Table of Contents.

15 DisplayFunction

Sometimes, you don't immediately want to see the output of a graphics command such as ListPlot**ListPlot** ()ListPlot, ParametricPlot**ParametricPlot** ()ParametricPlot, or Plot**Plot** ()Plot. Instead, you plan to set up several plots, and then show them together with the command Show**Show** ()Show.
You can easily hide the preliminary plots, and only show the composite one, by using the **DisplayFunction** option:

```
Clear[plot1, plot2, x]
```

```
plot1 = Plot[ Sin[x], {x,0,Pi/2},
  DisplayFunction->Identity ]
```

```
plot2 = Plot[ 1, {x, Pi/2, 3.0},
  DisplayFunction->Identity]
```

```
Show[ plot1, plot2,
  DisplayFunction->\$DisplayFunction ]
```

16 Do

The **Do** command allows you to make a loop that repeats one or more operations. The format of the **Do** command is

```
Do [
  { command1,
    command2,
    ...
    last command } ,
{ counter, start-value, end-value, increment} ]
```

For instance, the following **Do** loop calculates the square of the odd numbers between 1 and 9 and then prints out the number and its square.
Warning: this command won't produce any output in the HTML document!

```
Clear[n, nsquare]
```

```
Do [
  { nsquare = n^2,
    Print ["n=", n, "   n squared= ",nsquare] },
{n, 1, 9, 2} ] }
```

In this **Do** loop, the commands to be repeated are grouped inside the curly brackets. (If there is only one command, you don't need the brackets). After the commands comes an iterator, similar to what you've seen for the command Table**Table** ()Table. This tells *Mathematica*

- the counter variable (**n** in our case);
- the starting value of the counter (1);
- the ending value of the counter (9);
- how the counter variable increases on each step (add 2 to the previous value).

The **Do** command can carry out a simple method of approximating the solution of a differential equation. (Of course, the **DSolve** command is better, if you can use it.) Let **h**, **v**, and **a** represent the height, velocity, and acceleration of a ball under the influence of gravity.
Set the initial values at time 0:

```
Clear[a, dt, h, v]
```

```
h[0]=200;
```

```
v[0]=50;
```

```
a[0]=-32;
```

```
dt=0.5;
```

And now use a **Do** statement to compute new values, using the approximation:

$$h(t+dt) = h(t) + dt * v(t) \quad (1)$$
$$v(t+dt) = v(t) + dt * a(t) \quad (2)$$
$$a(t+dt) = -32 \quad (3)$$

```
Do[
  { h[i]=h[i-1]+dt*v[i-1],
    v[i]=v[i-1]+dt*a[i-1],
    a[i]=-32
  },
    {i,1,20}
  ]
```

We can make a plot to see what we have computed:

```
hvals=Table[{i*dt, h[i]}, {i,0,20}];
```

```
ListPlot[hvals]
```

Table of Contents**Table of Contents** ()Table of Contents.

17 DSolve

The **DSolve** command solves a differential equation, using exact techniques. If this command cannot solve your equation, you may need to look at the command NDSolve**NDSolve** ()NDSolve instead.
DSolve requires you to specify the following information:

- the differential equations and the initial conditions;
- the names of the dependent variables;
- the names of the independent variables.

For a simple example, let's solve

$$\frac{dy}{dx} = \frac{2y}{x} \tag{4}$$

with initial condition:

$$y(1) = 1 \tag{5}$$

which has the solution $y = x^2$.
Notice that when you specify the differential equation and initial condition, you must use the double equal sign.

```
Clear[x,y]
```

```
solution = DSolve[ { y'[x]==2*y[x]/x,  y[1]==1 }, y[x], x]
```

If you named the output of the **DSolve** command, then you can plot it, or use it in other formulas, as long as you use the "replacement operator" symbolized by "/.":

```
Plot[ y[x] /. solution, {x, 1, 5} ]
```

Table of Contents**Table of Contents** ()Table of Contents.

18 Expand

The **Expand** command is most useful when you want to force *Mathematica* to show you an expression in polynomial form, which is currently written as a set of factors, or as a polynomial raised to a power.

```
Clear[x]
```

```
Expand[ (x+1) * (x-3)^2 ]
```

The command Factor**Factor** ()Factor can undo the **Expand** command.
Table of Contents**Table of Contents** ()Table of Contents.

19 Factor

The **Factor** command takes a polynomial expression and tries to write it as a product of factors. This is similar to replacing 12 by 2 * 2 * 3.

```
Clear[formula, x]
```

```
formula=4*x^5 + 8*x^4 - 3*x^3 - 9*x^2
```

```
Factor[formula]
```

The command Expand**Expand** ()Expand can undo the **Factor** command.
Table of Contents**Table of Contents** ()Table of Contents.

20 FindRoot

The **FindRoot** command tries to find one value for a variable that will make an equation true. The command uses approximate techniques, so the answer is always a number, and it is not exact. The command requires that you supply a starting point, or guess, for the root, and the command may fail if the starting point is not good enough.
When you specify the equation to be solved, you must use the double equal sign.

```
Clear[x]
```

```
FindRoot[ x^2 == 7, {x, -2.0} ]
```

FindRoot can find roots for problems that the commands NSolve**NSolve** ()NSolve and Solve**Solve** ()Solve can't handle. Don't forget, though, that **FindRoot** can only find at most one root when it is called, no matter how many roots there actually are. For this next problem, there are roots at 2 and 4, but we'll only hear about one of them.

```
FindRoot[ 2^x == x^2, {x, 3} ]
```

Warning: If the variable you are solving for already has been assigned a value through an equals sign, then the output of **FindRoot** will be garbled.

```
x = 17;
```

```
FindRoot[ x^2 == 4, {x, 3} ]
```

In this case, you should use the command Clear**Clear** ()Clear just before the **FindRoot** command, to "free up" the variable **x**.
Table of Contents**Table of Contents** ()Table of Contents.

21 GridLines

The **GridLines** option to the commands ListPlot**ListPlot** ()ListPlot, ParametricPlot**ParametricPlot** ()ParametricPlot and Plot**Plot** ()Plot allows you to request that grid lines be included in your plot. The option **GridLines -> Automatic** has *Mathematica* choose the placement of the lines:

```
Plot[ Sin[x^3], {x,-2,2}, GridLines->Automatic]
```

You may also choose the location of grid lines yourself, by specifying a list of **x** coordinates and a list of **y** coordinates. You may want to add the option AspectRatio**AspectRatio** ()AspectRatio so that the units of measurement are the same in both directions.

```
Plot[ ArcTan[x], {x,-2,3},
  GridLines->{ {-2, -1.5, -1, -0.5, 0, 0.5, 1, 1.5, 2},
               {-1, -0.5, 0, 0.5, 1} },
  AspectRatio->Automatic]
```

To see how to define grid lines for a polar plot, see the discussion of Polar Grids**Polar Grids** ()Polar Grids.
Table of Contents**Table of Contents** ()Table of Contents.

22 Infinite Series

An infinite series is an expression that represents the sum of an infinite sequence of quantities. There is a standard mathematical notation for infinite series. For instance, to represent the infinite sum S of the

quantities 1, $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, ..., we would write

$$S = \sum_{i=1}^{\infty} \frac{1}{2^i}. \tag{6}$$

Using limits, it may be possible to assign a value to an infinite series. We do this by considering the behavior of *partial sums* in the limit. For instance, let us write S_n to represent the sum of the first n terms of the series. That is,

$$S_n = \sum_{i=1}^{n} \frac{1}{2^i}. \tag{7}$$

Certainly, for any n, we can compute this quantity. The first few values of S_n are 1, 1.5, 1.75, 1.875, ... and we might guess (correctly) that the value of S_n never reaches 2, but can be made as close to 2 as we want by choosing n large enough. In other words,

$$\lim_{n->\infty} S_n = 2. \tag{8}$$

In some cases, the limit is infinite, and we say that the series *diverges* to infinity. In other cases, there is no limit at all, and the series cannot be assigned any value.
Therefore, you might think that one way to analyze a series with *Mathematica* is to define the sequence**sequence** ()Sequences of partial sums, and then apply the command Limit**Limit** ()Limit. For our simple series above, here is how we could do this:

```
Clear[n, partial]
```

```
partial[n_] := Sum[ 1/2^i, {i,1,n} ]
```

```
Limit[ partial[n], n->Infinity}
```

Such an answer is useless. *Mathematica* is telling us it doesn't know how to handle this extremely simple problem. We can try another approach: simply use the **Sum** command with an infinite upper limit:

```
Sum[ (1/2)^i, {i,1,Infinity} ]
```

This seems to be getting us nowhere! Actually, however, *Mathematica* has computed the answer, but we have to force it out with the command N**N** ()N!

```
N[ Sum[ (1/2)^i, {i,1,Infinity} ] ]
```

Another useful tool is the *Ratio Test*, which considers the limit of the absolute value of the ratio of the *n+1*-st and *n*-th terms. If this limit is less than 1, then the infinite series has a (finite) limit. If the limit is greater than 1, then the infinite series diverges, while if the limit is exactly 1, the test is inconclusive. We can make this test easily:

```
term[n_] := 1/2^n
```

```
Limit[ term[n+1] / term[n], n->Infinity}
```

You may also want to refer to the discussion of Sequences**Sequences** ()Sequences.
Table of Contents**Table of Contents** ()Table of Contents.

23 Integrate

The **Integrate** command can be used to compute the indefinite integral of a function.

```
Clear[c,x]
```

```
Integrate[ x^2, x]
```

Notice that *Mathematica* does not tack on a symbolic constant like **C** to the answer. If you want to impress your instructor, you can do it yourself:

```
Integrate[ Sin[x], x] + c
```

The **Integrate** command can also be used to compute the definite integral. In this case, you must enclose the variable of integration and the limits of integration in curly brackets.

```
Integrate[ x^2, {x,0,3} ]
```

In some cases, the **Integrate** command can compute answers in which the lower limit is minus Infinity or the upper limit is Infinity**Infinity**, ()Special Constants:

```
Integrate[ 1/x^3, {x,1,Infinity} ]
```

Integrate can easily handle definite integrals involving symbolic limits, which is useful when considering an improper integral. (Of course, *Mathematica* can handle improper integrals easily, but we may want to see the steps ourselves.) For instance, here is one way we could compute the value of the improper integral of $\frac{1}{x^2}$ from 1 to Infinity:

```
Integrate[ 1/x^2, {x,1,b}]
```

```
Limit[ 1 - 1/b, b->Infinity}
```

If *Mathematica* does not know an exact antiderivative of the function, then it will simply reprint the input you typed. If you don't think *Mathematica* is going to give you a useful answer, and you need a definite integral (not an indefinite one!) with numeric (not symbolic!) limits, then you can try using the command NIntegrate**NIntegrate** ()NIntegrate instead.
Table of Contents**Table of Contents** ()Table of Contents.

24 Limit

The **Limit** command is used to compute the limit of an dependendent variable or expression as some independent variable approaches a special value.

```
Clear[f, h, x]
```

```
f[x_]:=x^3
```

```
Limit[ (f[x+h]-f[x])/h, h->0]
```

Mathematica can also seek limits as a variable goes to positive or negative Infinity**Infinity**, ()Special Constants:

```
Limit[ (7*x+1)/x, x->Infinity]
```

By the way, *Mathematica* can handle the commands Integrate**Integrate** ()Integrate and Sum**Sum** ()Sum with an upper limit of Infinity**Infinity** ()Special Constants. The result should be the same as applying the appropriate **Limit** command when the upper limit is a finite symbolic value.
Table of Contents**Table of Contents** ()Table of Contents.

25 ListPlot

The **ListPlot** command allows you to make a plot out of pairs of data values. In the simplest case, you simply have a single list of data which we'll call **y**:

```
Clear[i,pairs,xvals,y,yvals]
```

```
y = { 1.1, 2.4, 2.2, 2.7, 5, 1.0, 4 }
```

```
ListPlot[ y, PlotJoined->True]
```

Suppose, however, that you want to plot pairs of values, that is, you have both the **x** and **y** coordinates for the points. Then you first need to to join the two lists into a single table, using the command Table**Table** ()Table:

```
xvals = {0, 1, 3, 2}
```

```
yvals = {4, 1, 5, 4}
```

```
pairs = Table[ { xvals[[i]], yvals[[i]] }, {i, 1, 4} ]
```

```
ListPlot[ pairs, PlotJoined->True ]
```

You may prefer to set up your data in a single command:

```
pairs = {  {0,4}, {1,1}, {3,5}, {2,4} }
```

You can assign the output of the **ListPlot** command a name, and then use the command Show**Show** ()Show to redisplay it.
There are many graphics options available, including AspectRatio**Aspect Ratio** ()AspectRatio, AxesLabel**AxesLabel** ()AxesLabel, AxesOrigin**AxesOrigin** ()AxesOrigin, GridLines**GridLines** ()GridLines, PlotLabel**PlotLabel** ()PlotLabel, PlotRange**PlotRange** ()PlotRange, and RGBColor**RGBColor** ()RGBColor. Table of Contents**Table of Contents** ()Table of Contents.

26 Lists of Data

In many cases, you will want to set up or reference data in a list. In *Mathematica*, the simplest kind of list corresponds to a "vector", and is simply a list of number or symbolic values, separated by commas and enclosed in curly brackets:

```
Clear[chart,datax,v]
```

```
v = { 1, Sin[Pi/5] }
```

Individual entries of a list may then be accessed by using double square brackets. You can use the command ListPlot**ListPlot** ()ListPlot to plot such data.

```
datax = { 1, 2, 5, 8, 14, 12, 6, 3 }
```

```
datax[[3]]
```

What about a two dimensional array or matrix? *Mathematica* thinks of such quantities as simply a list of lists. Thus, a two dimensional array is again a list of quantities separated by commas and enclosed in curly brackets. What makes it special is that the quantities themselves are lists, namely, the rows of the matrix. If we call our list **chart**, here is how we might set it up:

```
chart = { {11, 12, 13, 14},
          {21, 22, 23, 24},
          {31, 32, 33, 34} }
```

and we can refer to row 2 of **chart** as:

```
chart[[2]]
```

or to the entry in row 2, column 3 as:

```
chart[[2,3]]
```

However, there is not an easy way to reference a column of data. The best way uses the command Table**Table** ()Table.
Table of Contents**Table of Contents** ()Table of Contents.

27 N (numeric value)

The **N** command is used to turn a symbolic or exact number or result into a decimal value.

```
Clear[x]
```

```
N[Pi]
```

```
Integrate[Sin[x],{x,0,1}]
```

```
N[Integrate[Sin[x],{x,0,1}]]
```

You can request more decimal places of accuracy:

```
N[Sqrt[2],20]
```

```
N[Pi,100]
```

The output of most *Mathematica* commands is one or more numbers. You can simply "wrap" the **N** command around such a command, to convert the results to a decimal value:

```
N[ Solve[ x^3-3*x^2-5*x+15 == 0, x ] ]
```

Table of Contents**Table of Contents** ()Table of Contents.

28 NDSolve

The **NDSolve** command is similar to the command DSolve**DSolve** ()DSolve in that it is used to solve one or more differential equations with initial conditions. However, it is different in several respects.
NDSolve does not try to compute an exact, symbolic answer. Instead, it uses special methods to approximate the answer.
Therefore, **NDSolve** can be applied to all the problems **DSolve** can handle, but can also solve problems for which no exact solution can be found.
However, because of the way it handles the problem, **NDSolve** does not find a formula for the exact answer. Instead, it computes an "Interpolating Function". The interpolating function is *Mathematica's* estimate for the behavior of the function. When you call **NDSolve**, you have to specify a name to be used for the interpolating function.
The format of the **NDSolve** command is:

```
name = NDSolve[  { Differential equations, initial conditions},
                                y[x], {x, xmin, xmax} ]
```

For example, to solve the problem

$$\frac{dy}{dx} = (4 - 2x)y \tag{9}$$

with initial condition:

$$y(0) = \frac{1}{6} \tag{10}$$

over the interval

$$0 \leq x \leq 5. \tag{11}$$

we would issue the following *Mathematica* commands:

```
Clear[solution, table1, x, y]
```

```
solution =
  NDSolve[ { y'[x]==(4-2*x)*y[x], y[0]==1/6}, y[x], {x, 0, 5} ]
```

In order to actually get numbers out of our answer, we have to use the **/.** substitution operator. For instance, if our function **y** was defined by a formula in the usual way, we would get the value of **y** at 1.0 by typing **y[1]**. Now, our function is not a formula, but an interpolation function whose properties are defined in the variable called **solution**. Therefore, we have to use the following command to evaluate **y[1.0]**:

```
y[1.0] /. solution
```

Similarly, wherever we would normally use a formula like **y[x]**, we instead must use the expression **y[x] /. solution**. Here, for instance, we make a table of the solution:

```
table1 =
  Table[ {x, y[x] /. solution }, {x, 0, 5, 0.5} ];
```

```
TableForm[ table1, TableHeadings->{None, {"x","y[x]"} } ]
```

Warning: An interpolating function may only be evaluated over a specific range. For our example, we only asked for an approximate solution over the range 0 <= x <= 5. *Mathematica* will refuse to compute a value of the interpolating function outside this range:

```
y[6] /. solution
```

Table of Contents**Table of Contents** ()Table of Contents.

29 NIntegrate

The **NIntegrate** command approximates the definite integral of a function between two limits. Instead of trying to find the indefinite integral, and then using the Fundamental Theorem of Calculus, **NIntegrate** uses ideas based on Riemann sums to get an approximate value of the integral of many functions for which the command Integrate**Integrate** ()Integrate cannot get the exact value.

```
Clear[x]
```

```
NIntegrate[Sin[x^3],{x,0,1}]
```

Warning: **NIntegrate** requires finite numerical values for the upper and lower limits of the integral. You definitely can not use **NIntegrate** for integrals with an upper or lower lower bound of Infinity**Infinity** (Special Constants.

Table of Contents**Table of Contents** ()Table of Contents.

30 NSolve

The **NSolve** command tries to find all the roots of an algebraic equation, using approximate methods. The equation to be solved is specified using a double equal sign. The second argument is the name of the variable to be solved for.

```
Clear[x]
```

```
NSolve[ x^2 == 7, x]
```

Here's how **NSolve** fails on a simple equation whose roots are 2 and 4. Notice that **NSolve** can't solve this problem any better than **Solve** can, because this is not an algebraic equation involving polynomials. The term 2^x is considered "transcendental". However, **NSolve** should be able to find the roots of most polynomial equations, whereas **Solve** will fail for most polynomials of degree 5 or greater.

```
NSolve[ x^2 == 2^x, x]
```

You may also wish to refer to the commands Solve**Solve** ()Solve or FindRoot**FindRoot** ()FindRoot.
Table of Contents**Table of Contents** ()Table of Contents.

31 NSum

NSum numerically estimates the value of an infinite sum.
To estimate the sum of the terms $\frac{1}{2^n}$, the command would be

```
Clear[n]
```

```
NSum[ 1/(2^n), {n,1,Infinity} ]
```

Since **NSum** only uses numerical techniques, it cannot sum symbolic quantities, and the answers it produces may be slightly or greatly inaccurate. For instance, the actual sum of the terms $\frac{1}{n}$ is infinite, but here is what **NSum** tells us:

```
NSum[ 1/n, {n,1,Infinity} ]
```

If you are summing symbolic quantities, or need an exact answer, you should refer to the command Sum**Sum** ()Sum.
Table of Contents**Table of Contents** ()Table of Contents.

32 ParametricPlot

Most plots are of functions that can be expressed in the form **y=f(x)**. But there are some objects we'd like to draw that don't fit this pattern. In particular, the points on a circle are exactly the points which satisfy the relationship $x^2 + y^2 = r^2$, but this equation can't be reduced to the form **y=f(x)** without leaving out some points. The expression

$$y = \sqrt{r^2 - x^2} \tag{12}$$

only produces the top half of the circle, for instance.
In such a case, it is often possible to express both the **x** and **y** coordinates of the graph in terms of a helping variable, called a parameter. For instance, the **x** and **y** coordinates of a circle of radius **r** can be expressed in terms of a parameter **t** as:

```
Clear[r, t, x, y]
```

```
r=2;
```

```
x[t_]:=r*Cos[t]
```

```
y[t_]:=r*Sin[t]
```

For such a situation, *Mathematica* supplies the **ParametricPlot** command, which is similar to the command Plot**Plot** ()Plot, except that it requires formulas for both **x** and **y** as functions of the parameter:

```
ParametricPlot[ {x[t], y[t]}, {t, 0, 2*Pi},
  AspectRatio->Automatic]
```

There are many graphics options available, including AspectRatio**Aspect Ratio** ()AspectRatio, AxesLabel**AxesLabel** ()AxesLabel, AxesOrigin**AxesOrigin** ()AxesOrigin, GridLines**GridLines** ()GridLines, PlotLabel**PlotLabel** ()PlotLabel, PlotRange**PlotRange** ()PlotRange, and RGBColor**RGBColor** ()RGBColor.
The command PolarPlot**PolarPlot** ()PolarPlot is defined in terms of the **ParametricPlot** command.
Table of Contents**Table of Contents** ()Table of Contents.

33 Plot

The most common **Plot** command plots one function on a given **x** domain, letting *Mathematica* decide how to choose the **y** range of the plot.

```
Clear[f,g,x]
```

```
Plot[ Sin[x], {x,-Pi,Pi} ]
```

To plot two graphs at the same time, you can simply include both formulas in curly brackets. To make things clearer, we will give the formulas names first:

```
f[x_]:=Sqrt[1-x^2]
```

```
g[x_]:=x
```

```
Plot[ { f[x], g[x] }, {x,0,1} ]
```

You can also use the command Show**Show** ()Show to accomplish this task.
Plots can take up a lot of memory. If you can't save your notebook to a floppy disk because it's too large, try deleting some of your pictures. As long as you leave the actual **Plot** commands in your file, you can easily get your pictures back the next time you run *Mathematica*.
You may be interested in the example Plot and D Don't Mix**Plot and D Don't Mix** ()Plot and D Don't Mix.
There are many graphics options available, including AspectRatio**Aspect Ratio** ()AspectRatio, AxesLabel**AxesLabel** ()AxesLabel, AxesOrigin**AxesOrigin** ()AxesOrigin, GridLines**GridLines** ()GridLines, PlotLabel**PlotLabel** ()PlotLabel, PlotRange**PlotRange** ()PlotRange, and RGBColor**RGBColor** ()RGBColor.
Table of Contents**Table of Contents** ()Table of Contents.

34 Plot and D Don't Mix

If you have defined a function **f**, and you want to plot its first derivative, then the proper way to do this is to use the "prime" or "single quote" symbol in the command Plot**Plot** ()Plot:

```
Clear[f, x]
```

```
f[x_]:=Sin[x]
```

```
Plot[ f'[x], {x,0,Pi} ]
```

You cannot use **D[f[x],x]** in the **Plot** command, even though it means the same thing (to us) as **f'[x]**! This is because the **Plot** command would see the variable **x** used twice, once as a symbolic argument, and once as the range of plotting. So the following command won't work:

```
Plot[ D[f[x],x], {x,0,Pi} ]
```

You may want to refer to the commands D **D** ()D, or Plot**Plot** ()Plot.
Table of Contents**Table of Contents** ()Table of Contents.

35 PlotLabel

PlotLabel is an option that may be specified in the graphics commands ListPlot**ListPlot** ()ListPlot, ParametricPlot**ParametricPlot** ()ParametricPlot, Plot**Plot** ()Plot, and Show**Show** ()Show.
The **PlotLabel** option prints a title on your plot. The title should be enclosed in double quotes:

```
Clear[x]
```

```
Plot[ Sin[x]*Sin[2*x], {x,0,10},
  PlotLabel->"Product of two Sine functions" ]
```

If you have several plots, each with a plot label, and you show them together with the **Show** command, only one label will show up. If the **Show** command specifies a **PlotLabel** option, then that is the plot label that will appear.
Table of Contents**Table of Contents** ()Table of Contents.

36 PlotRange

PlotRange is an option that may be specified in the graphics commands ListPlot**ListPlot** ()ListPlot, ParametricPlot**ParametricPlot** ()ParametricPlot, Plot**Plot** ()Plot, and Show**Show** ()Show.
You may often find that you want to change the range that *Mathematica* chooses for the graph. This is easy to do with the **PlotRange** option. It can be useful when *Mathematica* decides to "crop" your graph, omitting very high or very low values. In that case, you can use the option

```
PlotRange->All
```

to force the display of all data. In other cases, you may want to do some "cropping" yourself. You might want to focus on where a graph crosses the **x** axis, for instance:

```
Clear[f, x]
```

```
f[x_]:=(x+1) * (x-2)^2 * x
```

```
Plot[ f[x], {x,-3,3}, PlotRange->{-1,1} ]
```

Table of Contents**Table of Contents** ()Table of Contents.

37 Polar Grids

When drawing a function using polar coordinates, it may be convenient to show a set of polar grid lines, that is, a set of regularly spaced circles and rays.
The following *Mathematica* function **circles** can be used to draw **ncirc** circles around the origin, with radius **rmin** to **rmax**:

```
Clear[circles]
```

```
circles[rmin_,rmax_,ncirc_]:=
  Block[{circplot},
    circplot=Graphics[
      Table[
        Circle[{0,0},(rmin*(ncirc-n)+rmax*(n-1))/(ncirc-1)],
          {n,1,ncirc}
      ]
    ]
  ]
```

To use the function, choose values for the arguments and call **circles**. There won't be any output, except for a note from *Mathematica* saying "Graphics". You should save the output of the command as a variable.

```
rmin=1;
rmax=2;
ncirc=11;
p1=circles[rmin,rmax,ncirc]
```

The grid can be displayed at any time using the command Show**Show** ()Show.

```
Show[p1, AspectRatio->Automatic]
```

The function **rays** can be used to draw **nrays** rays from the origin, extending from radius **rmin** to **rmax**:

```
Clear[rays]
```

```
rays[rmin_,rmax_,nrays_]:=
  Block[{rayplot},
    rayplot=Graphics[
      Table[
        Line[{{rmin*Cos[n],rmin*Sin[n]},
              {rmax*Cos[n],rmax*Sin[n]}}],
          {n,0,2Pi-2*Pi/nrays, 2*Pi/nrays}
      ]
    ]
  ]
```

As before, the output of **rays** should be saved in a variable:

```
rmin=0.5;
rmax=2;
nrays=10;
p2=rays[rmin,rmax,nrays]
```

Now prepare a graph of your function in polar coordinates, using the command ParametricPlot**ParametricPlot** ()ParametricPlot:

```
f[t_]:=2*Cos[3*t]
```

```
p3=ParametricPlot[ {f[t]*Cos[t], f[t]*Sin[t]}, {t,0,2*Pi},
  AspectRatio->Automatic, PlotStyle->RGBColor[0,0,1] ]
```

To show your function along with the circles and rays that form the grid, use the command Show**Show** ()Show. Don't forget the option AspectRatio**AspectRatio** ()AspectRatio so that your plot doesn't come out squashed:

```
Show[p1,p2,p3,AspectRatio->Automatic]
```

You may also want to refer to the discussion of the option GridLines**GridLines** ()GridLines, used for (x,y) graphics.
Table of Contents**Table of Contents** ()Table of Contents.

38 PolarPlot

The **PolarPlot** command produces a plot using a formula of the form

$$r = f(\theta). \tag{13}$$

Here, we are using polar coordinates, (r, θ) to represent points in the plane. The value **r** represents the *radius* or distance of a point from the origin, and θ or *theta* represents the angle between the **x**-axis and the line from the origin to the point.
PolarPlot is not a built-in *Mathematica* command, so you have to request that it be read in. One method of doing this uses the **Needs** command. Beware! You must type in the double quotes and the backward quotes exactly as they appear here!

```
Clear[f,r,t]
```

```
Needs["Graphics`Graphics`"]
```

The command << can be used instead of **Needs**. However, every time it is invoked, the << command reads in a new copy of the requested command, and does not wipe out the old copies. Using << twice for the same command can cause memory or logic problems for *Mathematica*.

```
(* Don't enter this command if "Needs" worked! *)
```

```
<<Graphics`Graphics`
```

Your third option is simply to define the **PolarPlot** command yourself:

```
(* Don't enter this command if you entered "Needs" *)
(* or "<<" !                                       *)
```

```
PolarPlot[ r_, {t_, tmin_, tmax_}] :=
  ParametricPlot[ {r*Cos[t], r*Sin[t]}, {t,tmin,tmax},
    AspectRatio->Automatic]
```

PolarPlot expects a formula for the radius **r** in terms of the angle θ.

```
r[theta_] := 2*theta
```

```
PolarPlot[ r[theta], {theta,0,20*Pi} ]
```

If *Mathematica* did not draw a plot in response to the above command, then you have not loaded **PolarPlot** successfully.
Because of the way **PolarPlot** is defined in terms of ParametricPlot**ParametricPlot** ()ParametricPlot, you can't add any graphics options, such as RGBColor**RGBColor** ()RGBColor:

```
PolarPlot[ Cos[3*t], {t,0,Pi}, PlotStyle->RGBColor[1,0,0]}]
```

So if you really want a flexible way of doing polar plots, you will probably want to use the fourth option, which is not to use the clumsy, inflexible **PolarPlot** command at all, but instead, use the built-in command ParametricPlot**ParametricPlot** ()ParametricPlot:

```
r[theta_]:= Cos[3*theta]
```

```
ParametricPlot[ {r[t]*Cos[t], r[t]*Sin[t]},
  {t,0,Pi}, AspectRatio->Automatic,
    PlotStyle->{RGBColor[1,0,0]}, Axes->None]
```

Table of Contents**Table of Contents** ()Table of Contents.

39 Prime

The command **Prime[n]** produces the **n**-th prime number. This command will operate fairly quickly for **n** as large as 100,000,000.

```
Clear[n]
```

Here's the first prime number:

```
Prime[1]
```

and here's one way to print out the first 10 primes:
Warning: This command won't work properly in the HTML version.

```
Do[ {
  Print ["n=", n, " Prime[n]= ",Prime[n] ] },
  {n, 1, 10} ]
```

Table of Contents**Table of Contents** ()Table of Contents.

40 Print

If you want to see the value of a variable, you can simply type its name. For more complicated output, the **Print** command can help you display output lines that include values and text.
Warning: The output of this command in the HTML document will have lost all its spaces.

```
Print[ "The value of Pi to 20 places is ",N[Pi,20] ]
```

For another example of the **Print** command, see the Do**Do** ()Do command.
Table of Contents**Table of Contents** ()Table of Contents.

41 RGBColor

RGBColor is an option that may be specified in the graphics commands ListPlot**ListPlot** ()ListPlot, ParametricPlot**ParametricPlot** ()ParametricPlot, Plot**Plot** ()Plot, and Show**Show** ()Show.
If you are drawing two curves on the same plot, you may want to draw them in different colors, using the **RGBColor** option. The three numbers you specify for each line are the relative amounts of red, green and blue to be used, between 0 and 1.

```
Clear[f, g, x]
```

```
f[x_]:=Sqrt[1-x^2]
```

```
g[x_]:=x
```

```
Plot[ {f[x],g[x]}, {x,0,1},
  PlotStyle->{ RGBColor[1.000, 0.000, 0.000],
  RGBColor[0.000, 1.000, 0.000] } ]
```

Table of Contents**Table of Contents** ()Table of Contents.

42 Sampling Plot Coordinates

In some cases, you might want *Mathematica* to report to you the coordinates of specific locations in a plot that it has displayed. This is easy to do if you follow these steps:

- Move the cursor so that it lies within the plot. The cursor should look like a circle with a plus sign inside it.
- Now click once. The cursor should change, to look like four arrows.
- Now hold down the command key. On the Macintosh, this is the key with the "apple" and "cloverleaf", while on the PC it is the **CTRL** or **Control** key. Move the cursor around in the plot. You should notice that the current coordinates of the cursor show up in the lower left corner of your *Mathematica* window.

Now you can copy the coordinates of one or more of the points on the graph into a data array. To do so:

- Move the cursor around the plot, and click on points whose coordinates you want.
- Release the command key.
- Go to the **Edit** menu and select **Copy**.
- Move the cursor to a text or graphics cell, and click.
- Go to the **Edit** menu and select **Paste**.

You will get pairs of coordinates in a *Mathematica* array format. You can assign this data to a matrix or table, and then manipulate it any way you like.
Table of Contents**Table of Contents** ()Table of Contents.

43 Sequences

We can think of a sequence as an endless list of numbers, whose values are determined by some rule. Each number in the sequence is called a **term** of the sequence. We think of the terms as being numbered, starting with term 1 or sometimes with term 0.

```
Clear{f,fib,n,values}
```

Some sequences can be defined by writing a formula for the **n**-th term. In such a case, any particular term of the sequence can be found simply by evaluating the formula. For instance, if the **n**-th term is $\frac{2n+1}{n}$, then we can symbolize the general term with a formula, and find the value of, say, the 10-th term by evaluating the function there:

```
f[n_] := (2*n+1) / n
```

```
f[10]
```

We can use *Mathematica* commands like **Table** or even **Plot**:

```
values = Table[ {n, f[n]}, {n,1,5} ];
```

```
TableForm[values]
```

```
Plot[ {0, f[n]}, {n,1,5} ]
```

It's usually easy for *Mathematica* to check the limit of the sequence, or even to differentiate it, when using L'Hopital's rule.

```
Limit[ f[n], n->Infinity]
```

In some cases, however, we don't know a formula for each term. Instead, we may know a rule for making the next value of the sequence from the previous values. A sequence defined in this way is called a *recursive sequence*. For instance, the Fibonacci sequence is defined as follows: the first two terms are 0 and 1, and each following term is the sum of the two previous ones, so the sequence begins 0, 1, 1, 2, 3, 5, 8, 11, ... How can we get *Mathematica* to handle this case? We have to use a special form of the function definition, in which we specify the starting valus, and the rule for getting the next one:

```
fib[0] = 0;
```

```
fib[1] = 1;
```

```
fib[n_] := fib[n] = fib[n-2] + fib[n-1]
```

Notice that we counted the first term as number 0, rather than 1. That's up to us. Notice also the strange form of the functional definition. The expression **fib[n]** is not actually necessary; it is there to make the computations faster. It tells *Mathematica* that, whenever it computes a specific value of the Fibonacci sequence, it should "remember" it, so that it can be used in other computations.
The other thing to note about this sort of definition is that we can't use a **Plot** command to display the function, since it will only be defined for integer values. Instead, we must use the command Table**Table**, ()Table to gather up the values we want, and then display them with the command ListPlot**ListPlot**, ()ListPlot.

```
values = Table[ {n, fib[n]}, {n,0,5} ];
```

```
ListPlot[ values, PlotJoined->True]
```

Another problem with recursive sequences is that we can't use the **Limit** command with them. If you actually entered the following (disabled) *Mathematica* command, you might wait several minutes, with no response, before having to go up to the **Action** menu and selecting **Abort Calculation** to halt it:

```
Limit[ fib[n], n->Infinity]
```

One common source of sequences is in the study of Infinite Series**Infinite Series**, ()Infinite Series. We try to understand such quantities by considering the limit of the sequence of partial sums.
Table of Contents**Table of Contents** ()Table of Contents.

44 Series

The **Series** command produces the beginning terms of a power series for a given function. For instance, let us suppose we want the series for **ln(x)** around the point **x=1**, up to the term involving fourth powers:

```
Clear[approx,f,result,t,x]
```

```
f[x_]:= Log[x]
```

```
Series[ f[x], {x, 1, 4} ]
```

The error term $O(x-1)^5$ is an expression that tells us how our approximate curve will differ from the true curve as we move away from 1. Now we're just interested in the series function itself, without this error term. We could simply type it in again, but we don't have to. The **Normal** command exists just to chop off the error term. For future use, we will also name this quantity:

```
result = Normal[ Series[ f[x], {x, 1, 4} ]  ]
```

Note that we could probably have used the simpler command **Normal**[%].
There are many interesting things to do with a series approximation, but we will need to have *Mathematica* treat it as a function first. This is surprisingly difficult to do. We'd like to think of it as a function of x, but unfortunately, the symbol x was used to define the function in the first place. We just have to figure out a way to make a new version of the function with a different symbolic variable:

```
approx[t_] := result /. x-> t
```

This command essentially says that once you've gotten the power series for **ln(x)**, replace x by t in the formula. This is only so that we can define the function properly. Once that's done, we can use *any* variable name as an argument of **approx**. Once we have set up or function, we can plot it or make tables.

```
data=Table[ {x, f[x], approx[x], f[x]-approx[x]},{x,0.5,5,0.5} ];
```

```
TableForm[data]
```

A plot of **ln(x)** versus the approximation shows good agreement near the point x=1, but the error increases rapidly after x=2.

```
Plot[ {f[x], approx[x]}, {x, 0, 3},
  PlotStyle->{RGBColor[1,0,0], RGBColor[0,0,1]}]
```

Table of Contents**Table of Contents** ()Table of Contents.

45 Show

For various reasons, you may want to prepare two plots separately, and then show them together on the same grid. One reason would be if you are defining a function using two formulas, each one good over a specific interval. You can graph each piece with the command Plot**Plot** ()Plot, being sure to "name" each plot, and then use the **Show** command to put them together.

```
Clear[plot1, plot2, x]
```

```
plot1 = Plot[ Sin[x], {x,0,Pi/2}, PlotLabel->"The Sine piece" ]
```

```
plot2 = Plot[ 1, {x, Pi/2, 3.0}, PlotLabel->"The One piece"]
```

```
Show[ plot1, plot2 ]
```

If the plots are labeled, then the **Show** command uses the first label it encounters. If this is a problem, just use the option PlotLabel**PlotLabel** ()PlotLabel to make a more appropriate label:

```
Show[ plot1, plot2, PlotLabel->"The composite plot"]
```

There are many graphics options available, including AspectRatio**AspectRatio** ()AspectRatio, AxesLabel**AxesLabel** ()AxesLabel, AxesOrigin**AxesOrigin** ()AxesOrigin, PlotLabel**PlotLabel** ()PlotLabel, PlotRange**PlotRange** ()PlotRange, and RGBColor**RGBColor** ()RGBColor.
Table of Contents**Table of Contents** ()Table of Contents.

46 Simple Functions

Many *Mathematica* commands, such as D (differentiate)**D** ()D, Integrate**Integrate** ()Integrate, and Plot**Plot** ()Plot, expect to receive a function to work on. The variable that the function depends on is very important, and it's best to define a function of a variable in a way that allows *Mathematica* to understand what's going on:

```
Clear[f, x]
```

```
f[x_] := 10*x^2+1
```

The underline is only used on the left hand side, after the name of the independent variable, and you must also be sure to use the "colon-equals" sign as part of the function definition. If you want to pass the function to a *Mathematica* command, you specify it as **f[x]**.

```
Clear[f, x]
```

```
f[x_] := 10*x^2+1
```

```
D[ f[x], x]
```

And you can easily evaluate the function by replacing its argument by a number.

```
Clear[f, x]
```

```
f[x_] := 10*x^2+1
```

```
f[3]
```

Be careful, though. If you plan to use the expression **f[x]** in a *Mathematica* command, then **x** should not have been assigned a particular value. You should not issue a command like **x=1**, for instance, because then *Mathematica* will become confused.

```
Clear[f, x]
```

```
x=3
```

```
f[x_] := 10*x^2+1
```

```
f[x]
```

```
Integrate[ f[x], x]
```

On the one hand, *Mathematica* thinks **x** is the number 1, and on the other hand, it will think **x** is an argument of a function, and can have any value. If you think you're having a problem like this, use the command Clear**Clear** ()Clear to get rid of it!

```
Clear[f, x]
```

```
x=3
```

```
Clear[x]
```

```
f[x_] := 10*x^2+1
```

```
Integrate[ f[x], x]
```

To see how to make a function that carries out several steps, uses iteration, or uses temporary variables, refer to Advanced Functions**Advanced Functions** ()Advanced Functions.
Table of Contents**Table of Contents** ()Table of Contents.

47 Simplify

Mathematica can simplify many polynomial or trigonometric expressions, but usually you must explicitly request it to do so, using the **Simplify** command. For instance, the polynomial fraction

$$\frac{(x^2 - 2x - 3)}{(x + 1)}$$

can be simplified:

```
Clear[x]
```

```
(x^2-2*x-3)/(x+1)
```

```
Simplify[ (x^2-2*x-3)/(x+1) ]
```

Another time when **Simplify** is useful is when verifying that one function is the inverse of another:

```
Clear[f, g, x]
```

```
f[x_] := (x+2)/x
```

```
g[x_] :=  2/(x-1)
```

```
Simplify[ g[ f[x] ] ]
```

```
Simplify[ f[ g[x] ] ]
```

You might also be interested in the commands Apart**Apart** ()Apart, Expand**Expand** ()Expand, Factor**Factor** ()Factor, or the Together**Together** ()Together.
Table of Contents**Table of Contents** ()Table of Contents.

48 Solve

The **Solve** command can be used to seek values of a variable which make an algebraic equation exactly true. The command will try to find all possible solutions. When you describe the equation to the command, you must use two equal signs! In some cases, **Solve** will return an answer in symbolic form, since it is computing exact values.

```
Clear[x, y]
```

```
Solve[ x^2 == 7, x]
```

In some cases, you may want to use the **N** command to immediately convert the output of **Solve** to decimal values:

```
N[Solve[x^2 == 7, x] ]
```

Solve can handle equations in which extra variables appear, even if those variables have not been assigned a value.

```
Clear[x, y]
```

```
Solve[ 2*x+y == 1, x]
```

Solve can solve several equations in several unknowns:

```
Solve[ {2 x + 3 y == 8,
        x +   y == 3}, {x, y} ]
```

Solve can handle some equations that include trigonometric functions:

```
Clear[x]
```

```
Solve[ y == 12 + 0.6*Sin[(t-80)/3], t]
```

Solve should be able to treat any polynomial equation of order 4 or less. But higher order equations can make **Solve** fail. For instance, if **Solve** fails, and returns a message that begins with the words **ToRules**, it's telling you that it can't find the exact solution, but does have a numerical estimate, which you can get by typing **N[%]**:

```
Clear[x]
```

```
Solve[  x^5+2*x+1==0, x]
```

```
N[%]
```

And here is a transcendental equation that **Solve** cannot handle. (The problem is the term 2^x:

```
Clear[x]
```

```
Solve[  2^x==x^2, x]
```

If **Solve** can't handle your equation, you may need to try the command NSolve**NSolve** ()NSolve, which uses approximate techniques and seeks all roots, or FindRoot**FindRoot** ()FindRoot, which uses approximate techniques and seeks just one root.
Table of Contents**Table of Contents** ()Table of Contents.

49 Special Constants

There are several special constants that *Mathematica* names:

```
Clear[x]
```

- **Degree**, the conversion factor from degrees to radians;

```
N[Sin[20 Degree]]
```

- **E**, the base of the natural logarithm system;

```
D[ E^(x^2), x]
```

- **GoldenRatio**, the "golden ratio", $\frac{1+\sqrt{5}}{2}$;

```
Solve[ x / 1 == 1 / (x-1), x]
```

- **I**, the square root of minus 1;

```
Solve[ x^2 + 2x + 2 == 0, x]
```

- **Infinity**, useful in the command Limit**Limit** ()Limit, and as a limit of on the commands Integrate**Integrate** ()Integrate and Sum**Sum** ()Sum,;

```
Integrate[ 1/x^2, {x,1,Infinity} ]
```

- **Pi**, the ratio of the circumference of a circle to its diameter.

```
Sin[ Pi/2 ]
```

Table of Contents**Table of Contents** ()Table of Contents.

50 Special Functions

Mathematica has many mathematical functions, including:

- **Abs[x]**, the absolute value;
- **ArcCos[x]**, the inverse cosine function;
- **ArcSin[x]**, the inverse sine function;
- **ArcTan[x]**, the inverse tangent function;
- **Cos[x]** and **Cosh[x]**, the cosine and hyperbolic cosine;
- **Exp[x]**, the exponential function, **e** raised to the power **x**;
- **Factorial[x]**, or $x!$, the factorial function, $1 \cdot 2 \cdot 3 \cdot ... \cdot (x-1) \cdot (x)$, for x a nonnegative integer;
- **Gamma[x]**, the gamma function, defined for all x except nonpositive integers, with **Gamma[x]=(x-1)!** if x is a positive integer;
- **Log[x]**, the natural logarithm;
- **Log[x,b]**, the logarithm of **x**, base **b**;
- **Prime[x]**, the n-th prime number;
- **Sign[x]** returns -1, 0, or +1, when x is negative, zero, or positive;

- **Sin**[**x**] and **Sinh**[**x**], the sine and hyperbolic sine;
- **Sqrt**[**x**], the square root;
- **Tan**[**x**] and **Tanh**[**x**], the tangent and hyperbolic tangent.

Table of Contents**Table of Contents** ()Table of Contents.

51 Sum

The **Sum** command computes the sum of a set of numbers. One way to specify the set of numbers is as a formula:

```
Clear[a, b, delx, f, i, n, x]
```

```
Sum[ 2*i-1, {i, 1, 10} ]
```

You can also specify an increment, which tells how the index variable is increased from the lower limit to the upper limit. For instance, you could also add up the odd numbers between 1 and 19 by typing

```
Sum[ i, {i, 1, 19, 2}]
```

The counter variable does not have to be an integer. To do a left hand Riemann sum, for instance, you could try a formula like this:

```
f[x_]:=x+1
```

```
a=0;
```

```
b=2;
```

```
n=8;
```

```
delx=(b-a)/n;
```

```
N[ Sum[ f[x]*delx, {x, a, b-delx, delx} ] ]
```

The upper limit of the **Sum** command can be Infinity**Infinity** ()Special Constants.

```
Sum[ 1/2^n, {n,1,Infinity} ]
```

Sometimes, the **Sum** command may compute a useful answer for an infinite sum, but does not report it to you. When this happens, you may be able to retrieve a numerical result by using the command N**N** ()N:

```
N[ Sum[ 1/2^n, {n,1,Infinity} ] ]
```

If the **Sum** command does not perform satisfactorily, and you can accept a numerical approximation, you may be interested in the command NSum**NSum** ()NSum.
Table of Contents**Table of Contents** ()Table of Contents.

52 SurfaceOfRevolution

Warning: The "Needs" and "<<" commands won't work properly in the HTML version, and so the SurfaceOfRevolution command can't be loaded.
The **SurfaceOfRevolution** command produces a plot of the 3D surface of revolution generated by revolving a given curve **z=f(x)** about the vertical or **z** axis.
SurfaceOfRevolution is not a built-in command, so you have to tell *Mathematica* to read it in. The safest method uses the **Needs** command. Beware! You must type in the double quotes and backward quotes exactly as they appear here!

```
Clear[t,x]
```

```
Needs["Graphics`SurfaceOfRevolution`"]
```

The command << can be used instead of **Needs**. However, every time it is invoked, the << command reads in a new copy of the requested command, and does not wipe out old copies. Using << twice for the same command can cause memory or logic problems for *Mathematica*.

```
<<Graphics`SurfaceOfRevolution`
```

The simplest version of the **SurfaceOfRevolution** command assumes that we have a function **z=f(x)** which we wish to rotate about the **z** axis. If this is all we want, we have only to specify the domain of the function in order to get our plot.

```
SurfaceOfRevolution[ Sin[x], {x, 0, Pi}]
```

If *Mathematica* did not draw a plot in response to your command, then you probably have not loaded the **SurfaceOfRevolution** command properly.
Although we ususally want to rotate the curve through a full revolution, you can control this by specifying a third argument, of the form { **theta, theta_min, theta_max** }:

```
SurfaceOfRevolution[ Sin[x], {x, 0, Pi}, {t, 0, 2 Pi/3} ]
```

But suppose you want to rotate the sine function about the **x** axis, getting a cigar shape? This can be accomplished by specifying the value of the option **RevolutionAxis**. By default, this option has the value {0,0,1}, specifying revolution about the **z** axis. You never want to rotate about the **y** axis in this setup. To rotate about the **x** axis, specify an axis of { 1,0,0 }.

```
SurfaceOfRevolution[ Sin[x], {x, 0, Pi}, RotationAxis->{1,0,0} ]
```

Warnings:

- Plots can take up a lot of memory. If you can't save your notebook to a floppy disk because it's too large, try deleting some of your pictures. As long as you leave the actual **Plot** commands in your file, you can easily get your pictures back the next time you run *Mathematica*.
- The online documentation for **SurfaceOfRevolution** is just horrible, and there isn't any documentation at all for this command in Wolfram's big black book.

Table of Contents**Table of Contents** ()Table of Contents.

53 Table

The **Table** command allows you to set up and store a table of data. Usually, you want to use a semicolon at the end of the **Table** command to suppress its output. The command TableForm**TableForm** ()TableForm prints out the table in a neat format, with optional row and column labels:

```
Clear[col2, fibonacci, i, row1, row2, row5, squares]
```

```
squares = Table[ {i, i^2}, {i,1,10}]
```

```
TableForm[squares,
  TableHeadings->{None, {"Number", "Square"} } ]
```

A table is not just something to be printed out. It is also an array of data, usually two dimensional. Any single entry of the table can be accessed by specifying its row and column numbers, within double square brackets:

```
squares[[7,2]]
```

Similarly, any row of the table can be accessed by specifying just the row number. The result will be a list:

```
row5 = squares[[5]]
```

Getting a column of the table into a list is a little trickier. We'll actually use the **Table** command to do it!

```
col2 = Table[ squares[[i,2]], {i,1,9} ]
```

Also, you can build a table, one row at a time, from a collection of lists:

```
row1 = {1, 2, 3, 4, 5, 6}
```

```
row2 = {1, 1, 2, 3, 5, 8}
```

```
fibonacci = {row1, row2}
```

```
TableForm[fibonacci,
  TableHeadings->{{"N", "Fibonacci Number"}, None} ]
```

Table of Contents**Table of Contents** ()Table of Contents.

54 TableForm

The **TableForm** command prints a table in a neat format, by rows and columns. The table will usually have been set up by the command Table**Table** ()Table.
In the simplest use, we immediately follow a **Table** command by a **TableForm** command, whose argument is simply %:

```
Clear[i, matrix]
```

```
Table[ {i, i^2, i^3}, {i,1,10}]
```

```
TableForm[%]
```

If the data to be printed has been named, then that name should be specified in the **TableForm** command:

```
matrix = { {11, 12, 13}, {21, 22, 23}, {31, 32, 33} }
```

```
TableForm[matrix]
```

The rows and columns may be labeled by specifying the **TableHeadings** option. The form of the option is:

```
TableHeadings->{
  {"row 1 label",    "row 2 label",    ..., "last row label"},
  {"column 1 label", "column 2 label", ..., "last column label"} }
```

If there are to be no row labels, then the entire list of row labels is replaced by the word **None**; the same holds if there will be no column labels:

```
TableForm[ matrix, TableHeadings->{
  {"Row 1", "Second Row", "Last One"},
  {"Column 1", "Column 2", "Column 3"} } ]
```

```
TableForm[ matrix, TableHeadings->{
  None, {"Column 1", "Column 2", "Column 3"} } ]
```

```
TableForm[ matrix, TableHeadings->{
  {"Row 1", "Second Row", "Last One"}, None } ]
```

Warning: If your column labels are too long to fit on one line, then **TableForm** will produce ugly output, by just continuing the labels (and the data beneath them) onto the next line.
Table of Contents**Table of Contents** ()Table of Contents.

55 Together

The **Together** command can be used to take several polynomials and polynomial fractions and join them into a single polynomial fraction with a common denominator. This is similar to the operation of converting $2/3 + 1/4 + 3$ into $47/12$.

```
Clear[stuff, x]
```

```
stuff = x^2 + 1 + (x-3) / (x^2 -3*x + 4)
```

```
Together[stuff]
```

The command Apart**Apart** ()Apart can undo the **Together** command.