Physics 115/242
Introduction to *Mathematica*.

Peter Young

May 13, 2013

Contents

1 Introduction

1.1 Starting *Mathematica*

This is a basic introduction to *Mathematica*. Since *Mathematica* is a very powerful program with many features only a small fraction of its capabilities will be discussed here. For a very thorough account see the book by Wolfram[1] *Mathematica*'s creator. For more readable summaries of *Mathematica*'s features see the other books in the list at the end; I found that Chapter 2 of the book by Tam[2] is particularly helpful, see also the book by Kaufmann[3]. Appendix A of Kinzel and Reents[4] illustrates the use of a large number of *Mathematica* commands.

Unlike conventional computer languages *Mathematica* can do symbolic manipulation, has a huge number of built-in numerical functions, can do numerical calculations to arbitrary precision, and has powerful plotting routines which interface directly with the results of calculations. *Mathematica* can also be used for programming in styles which are quite different from those of C and fortran. Here we will only discuss programming in *Mathematica* quite briefly.

There are two interfaces to *Mathematica*: (i) the command line interface, and (ii) the notebook interface. The same commands are given in both, but the notebook interface has some additional features. In this course, the special features of the notebook interface will not be essential.

If one starts *Mathematica* in Windows or on a Mac by clicking on an icon one goes straight into the notebook interface in which a *separate* window appears into which *Mathematica* commands are typed. One can go back to an old command, edit it, and then rerun it; convenient if one makes a typo in a long command. The output looks nice, and both input and output can produce Greek letters and other special symbols. To execute a command in Windows, you need to hold down the Shift key as well as pressing Enter (just pressing Enter allows you to continue entering your command on the next line but does not execute it). On a Mac, execute a command by typing Shift-Return or Enter (probably on the numeric keypad).

If one is running *Mathematica* under linux, one can either use the command line interface or the notebook interface.

1. To start the command line interface give the command "math" at the prompt from within an X-window. One then types commands into the *same* window. The output is not as pretty as with the notebook interface, one cannot produce Greek letters, and one cannot go back and rerun a previous command. On the other hand, one still has all the power of *Mathematica* at ones disposal. To execute a command simply hit the Enter key.

2. To start the notebook interface give the command "mathematica". A separate window is then fired up into which one types the commands. A command is executed by typing Shift-Enter. If you are running *Mathematica* remotely on another machine the notebook interface will not give good results unless the *Mathematica* fonts are present on the local machine.

The examples in this document are obtained from the command line interface. The document has been prepared over a period of time using different versions of *Mathematica*, so the precise form of the output may be different for you and depend on the version that you are using. Here is an example of a very simple session under linux, in which the command line interface is started up, the indefinite integral of $\ln x$ is evaluated, and *Mathematica* is exited by typing the command `"Quit"`.

```
 ~ => math
Mathematica{} 4.0 for Linux
Copyright 1988-1999 Wolfram Research, Inc.
 -- Motif graphics initialized --

In[1]:= Integrate[ Log[x], x ]

Out[1]= -x + x Log[x]

In[2]:= Quit
 ~ =>
```

Please notice three important things about this example:

1. You will see that lines are alternately input lines beginning with `In` and output lines beginning with `Out`. In the first line *Mathematica* entered `In[1]:=` and waited for the user to type input, in this case to integrate the natural log of x. In the next line, beginning with `Out[1]=`, *Mathematica* gives the result of the command.

2. *Mathematica* commands begin with a capital letter. The most common mistake of a beginner is to type in the name of a function, tan say, without the first letter being capitalized, i.e. the correct name of this function is `Tan`. It is conventional to use lower case letters for functions or variables that the user defines in order to differentiate them from *Mathematica* functions.

3. Arguments to commands are enclosed in *rectangular* brackets, $[\cdots]$. This enables *Mathematica* to differentiate between an argument of a function, on the one hand, from an expression in brackets used for grouping, e.g. `x (2 + 3x)`, on the other. In the latter case, *round* brackets are used. *Mathematica* needs to be more precise in its notation that conventional notation where much is implicitly understood from the context. A nice feature of this is that if you want to multiply a variable `x` by 5, say, you can just type `5x` in *Mathematica* just as you would by hand, rather than `5*x`, which you would have to do in most other computer languages. Note that `5*x` will also work. You can also multiply two variables `x` and `y` without an explicit multiplication symbol provided you leave a space between then, i.e. `x y` (if there is no space, i.e. `xy`, *Mathematica* will think you are referring to a *single* variable called `xy`). A variable name cannot start with a number, e.g.`2x`, because *Mathematica* would interpret this as 2 times x.

Note that in the above example, which used the command line interface, the version number was written out. You can also get this information by typing `$Version`. There are a number of other commands giving global system information which are described by Wolfram[1].

You will find that *Mathematica* gives warning messages about possible spelling errors if you use two variable names which are quite similar. These are annoying but can be switched off with the command `Off[General::spell1]`.

1.2 A Simple *Mathematica* Session

Mathematica understands the usual operators `+, -, *, /,` and ^ for exponentiation. It also understands basic constants such as π, (type `Pi`), e, (type `E`), ∞, (type `Infinity`), and $\sqrt{-1}$, (type `I`). It also knows about a large number of Mathematical functions such as `Exp[x], Sin[x], Cos[x], ArcSin[x], Log`[x] for the *natural* log,

`Factorial[n]` or more simply `n!`, and many more. See the references for more details. *Mathematica* know about the values of these functions for certain values of the arguments.

The following session illustrates some of these features:

```
In[1]:= x = 7

Out[1]= 7

In[2]:= y = 19

Out[2]= 19

In[3]:= x y

Out[3]= 133

In[4]:= (x + 3) y

Out[4]= 190

In[5]:= 2^x

Out[5]= 128

In[6]:= Exp[I Pi]

Out[6]= -1
```

1.3 On-Line Help

On-line help on *Mathematica* functions and other objects can be obtained by typing a question mark at the start of a line followed by the name of the object, e.g. to obtain information on `Pi`

```
In[1]:= ?Pi
Pi is the constant pi, with numerical value approximately equal to 3.14159.
```

More information is obtained with a double question mark, e.g. `??Plot` gives a list of the (many) options to the plot command as well as the syntax of the command. You can also use wild card characters, such as `*` which matches any sequence of alphanumeric characters. For example `?Plot*` give a list of commands which begin with `Plot`.

In the notebook interface there is a *huge* amount of information available from the help menu, including all of Wolfram's book[1] on line.

1.4 Referencing Previous Commands

The result from the previous command can be referenced by the percent sign `%`, the second previous result by two percent signs, `%%`, and so on, and the result from output line n by `%n`, e.g.

```
In[1]:= 17

Out[1]= 17

In[2]:= % + 100

Out[2]= 117
```

```
In[3]:= 100 * %%

Out[3]= 1700

In[4]:= 100 * %1

Out[4]= 1700
```

1.5 Clearing Variables

Mathematica remembers variables and functions that you have defined until the end of the session. One of the most common mistakes is to use the same name for a variable that has previously been defined. This can lead to unexpected results. To clear a previous definition of the variable `x` use `Clear[x]`. (Often the command `Remove` is used which is more powerful. As discussed in e.g. Tam[2] p. 10–12, `Clear` removes just removes the definition of the symbol, while `Remove` also removes the symbol from the list of user defined symbols.)

To remove *all* definitions created by the user, type the hieroglyphics `Clear["Global`*"]`. It is useful to understand what this means. "Global" refers to the set of all variables that you have defined (called a "context" in *Mathematica*). Think of it as like a directory in a computer filesystem. The backquote "`" is then like the slash (in Unix) which takes you to a subdirectory, and the star "*" is the usual wildcard character standing for any number of any characters. Since `Global`*` is not a *Mathematica* variable or function (more precisely is not a *Mathematica* "symbol") is has to be put in quotes (" ··· "). `"Global`*"` therefore stands for all the symbols that you have defined. `Clear["Global`*"]` then removes the definitions of all these symbols.

I recommend you always put `Clear["Global`*"]` as the first line of a notebook. This means that, it you have already run the notebook but have make some changes and want to run it again, you can do so without current values of the variables causing problems. To rerun the notebook, go to the `Kernel` menu, select `Evaluation` and then select `Evaluate Notebook`.

Consider the following example in which `x` is defined to be 10 and then we try to integrate x^2 with respect to `x`. An error is given because of the previous definition of x, and *Mathematica* returns the expression unevaluated. However, after `Clear[x]` the integration can be done.

```
In[1]:= x = 10

Out[1]= 10

In[2]:= Integrate[x^2, x]

Integrate::ilim: Invalid integration variable or limit(s) in 10 .

Out[2]= Integrate[100, 10]

In[3]:= Clear[x]

In[4]:= Integrate[x^2, x]

         3
        x
Out[4]= --
        3
```

1.6 Ending a Command with a Semicolon

If you end a line with a semicolon, ";", the output is not printed. This might be useful if the line in question is a complicated intermediate stage in a calculation, not of interest in its own right but which is going to be used in subsequent steps. The semicolon can also be used to put several commands on the same line. The output from

the last command is printed (assuming that there is no semicolon after that) but not from the other commands on the line. Here is a simple example:

```
In[1]:= x = 10; y = x^4

Out[1]= 10000
```

1.7 Iterators, Table, Do, Example of Fibonacci Numbers

Many commands in *Mathematica* use "iterators"which perform a task a certain number of times. One example is `Table[expr, iterator]` which generates a list of `expr`. A list is a set of variables enclosed in curly brackets. For example:

```
In[1]:= Table[ 1/2^n, {n, 4} ]

         1  1  1  1
Out[1]= {-, -, -, --}
         2  4  8  16

In[2]:= Table[ 1/2^n, {n, 0, 4} ]

            1  1  1  1
Out[2]= {1, -, -, -, --}
            2  4  8  16

In[3]:= Table [ 1/2^n, {n, 0, 8, 3} ]

            1  1
Out[3]= {1, -, --}
            8  64
```

where `{n, 4}` increments `n` from 1 (by default) to 4, `{n, 0, 4}` increases `n` from 0 to 4 in (by default) unit intervals, and `{n, 0, 8, 3}` increases `n` from 0 to 8 in increments of 3. In addition `Table[expr, {n}]` executes `expr n` times, e.g.

```
In[4]:= Table[RandomReal[], {4}]

Out[4]= {0.0375749, 0.081942, 0.510687, 0.401433}
```

where `RandomReal[]` generates a random number between 0 and 1.

Another command which uses iterators is `Do[expr, iterator]` which evaluates `expr` as many times as is indicated by the iterator. The output from the commands generated by `Do` is *not* printed by default. Here is an example which uses `Do` to generate Fibonacci numbers , a_n. These are defined by $a_1 = 1, a_2 = 1$ and $a_{n+1} = a_n + a_{n-1}$ for $n = 2, 3, \cdots$ etc., so $a_3 = a_2 + a_1 = 2, a_4 = a_3 + a_2 = 3$, and so on. These can be calculated in *Mathematica* as follows:

```
In[5]:= n = 200;

In[6]:= a[1] = 1; a[2] = 1; Do [ a[k] = a[k-1] + a[k-2], {k, 3, n} ]; a[n]

Out[6]= 280571172992510140037611932413038677189525
```

Note that the values of `a[k]` generated by the `Do` command are not printed. The output of `a[n]` is due to the statement after the last semicolon in the `In[6]` line. Note also that *Mathematica* has no difficulty in dealing with the very large number that is generated. If we want to have the intermediate values `a[k]` printed as we go along we have to include a `Print[expr]` command, i.e.

```
In[7]:= n = 10;

In[8]:= a[1] = 1; a[2] = 1;Print[a[1]];Print[a[2]];\
         Do [ a[k] = a[k-1] + a[k-2]; Print[a[k]], {k, 3, n} ]
1
1
2
3
5
8
13
21
34
55
```

Note the \ at the end of the `In[8]:=` line so that the command will run on to the second line (this is not necessary in the notebook interface).

1.8 Lists

A list is a set of variables enclosed in curly brackets. Lists are important in *Mathematica* and there are many commands to manipulate them. Here we just note a few. If `a, b` and `c` are lists then `Join[a, b, c, ...]` creates a single list out of them and `Union[a, b, c, ...]` creates a sorted list with duplicated elements removed. `Join` and `Union` can take any number of arguments, each of which should be list. Also `Intersection[a, b, c, ...]` gives a sorted list of elements which are common to *all* the lists. For example;

```
In[1]:= a = {1, 2, 3, 4}; b = {1, 4, 9, 16};

In[2]:= d = Join[a, b]

Out[2]= {1, 2, 3, 4, 1, 4, 9, 16}

In[3]:= e = Union[a, b]

Out[3]= {1, 2, 3, 4, 9, 16}

In[4]:= e = Intersection[a, b]

Out[4]= {1, 4}
```

`Flatten[list]` removes inner brackets from nested lists, e.g.

```
In[1]:= a = {1, 2, 3, 4}; b = {1, 4, 9, 16};

In[4]:= c = {b, 100}

Out[4]= {{1, 4, 9, 16}, 100}

In[5]:= f = {a, b, c}

Out[5]= {{1, 2, 3, 4}, {1, 4, 9, 16}, {{1, 4, 9, 16}, 100}}

In[6]:= Flatten[%]

Out[6]= {1, 2, 3, 4, 1, 4, 9, 16, 1, 4, 9, 16, 100}
```

The n-th element of list `mylist` is given by `mylist[[n]]` (note the double square parentheses), e.g.

```
In[1]:= mylist = Table[n^2, {n, 1, 5}]

Out[1]= {1, 4, 9, 16, 25}

In[2]:= mylist[[3]]

Out[2]= 9
```

Mathematica has many test functions which are designed to select items from a list. For example `NumericQ[expr]` yields `True` if `expr` is a numerical expression and `False` otherwise, and `PrimeQ[expr]` detects whether `expr` is a prime number See the references for a description of others. Items of a list that satisfy the condition can then be extracted by `Select`. The following produces a list of the first 12 integers and then extracts those that are prime:

```
In[3]:= Table[n, {n, 1, 12}]

Out[3]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}

In[4]:= Select[%, PrimeQ]

Out[4]= {2, 3, 5, 7, 11}
```

There are many other functions for obtaining elements of list, see the books. Here we will just mention two of the them. `Drop[list, n]` removes the first n elements of `list`, while `Drop[list, {m, n}]` removes the m-th through the n-th elements of `list`. `Take[list, n]` gives the first n elements of `list` while `Take[list, {m, n}]` gives a list comprising the m-th through n-th elements of `list`. As an example

```
In[1]:= mylist = Table[n, {n, 10}]

Out[1]= {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

In[2]:= Drop[mylist, {3, 7}]

Out[2]= {1, 2, 8, 9, 10}

In[3]:= Take[mylist,{3, 7}]

Out[3]= {3, 4, 5, 6, 7}
```

1.9 Defining Functions

There are many functions in *Mathematica* but it is often very useful to be able to define functions oneself. This is illustrated by the following example, which defines the function $f(x) = x + x^2$.

```
In[37]:= f[x_] := x + x^2

In[38]:= f[3]

Out[38]= 12

In[39]:= f[t]
                2
Out[39]= t + t
```

Note two things about the line `In[37]:`. First of all the underscore "_" after the `x`, which is called a "blank", stands for "any expression", so when the function is called, the argument can be named anything, not necessarily

x. Secondly, the use of :=, rather than =, which means that the assignment is delayed until the function is called. In the present example it wouldn't have made any difference if we had used =, the "immediate" assignment operator, but it would if the function depended on a parameter. Look at the following example:

```
In[40]:= k = 10

Out[40]= 10

In[41]:= f[x_] = x + k x^2

                   2
Out[41]= x + 10 x

In[42]:= g[x_] := x + k x^2

In[43]:= g[2]

Out[43]= 42

In[44]:= k = 3

Out[44]= 3

In[45]:= g[2]

Out[45]= 14
```

In the line `In[41], f[x]` is defined with the current value of `k` hard-wired in. If later on we change `k` to some other value `f` will not change. However, `g[x]` is defined with delayed assignment so the value of `k` to be put into the function is the current value *when the function is called*, as lines 43-45 show. Usually this is what we want, so I recommend that you get into the habit of defining functions with delayed assignment.

If the function requires more than one command, the beginning and end of the function must be marked by round brackets, $(\cdots)$, e.g. we can make a function to generate Fibonacci numbers :

```
In[1]:= fib[n_] := (a[1] = 1; a[2] = 1; Do [a[k] = a[k-1] + a[k-2], \
        {k, 3, n} ]; a[n])

In[2]:= fib[10]

Out[2]= 55
```

Note that there is no output following the delayed assignment in the first input line.

One potential problem with user defined functions is that local variables, defined within the function, might have the same name as a global variable and overwrite its value. To prevent this one can define the function to be a `Module`, in which one specifies that certain variables are to be local to the module. See Sec. 5.1 and the references, e.g. Tam[2] p. 238.

We saw above the use of the underscore _ () as a "blank" in the definition of a function. Because of this problems arise if you put an underscore in a variable or function name. Hence, **do not put an underscore in function or variable names**. Similarly, it is not recommended to put a subscript or superscript in a variable name, see e.g. Ref. [6], p. 3. The problem is that Mathematica doesn't regard a variable x_1, for example, as completely independent of variable x. So, if you have a variable x_1 and set $x = 7$, say, then x_1 becomes 7_1 which is nonsense. Hence, it is better to define a list of variables as $\{x1, x2, x3, \cdots\}$ rather than $\{x_1, x_2, x_3, \cdots\}$.

A final way of defining a function in *Mathematica* is to create a "pure function". We will not use these much here except briefly in the case of differential equations, see Sec. 3.9. The command is `Function[x, body]` where the variable is `x` and `body` describes the function, e.g.

```
In[3]:= f1 = Function[x, Cos[x] + Sin[x]];
```

This can be more concisely (but not more readably!) be written as

```
In[4]:= f1 = Cos[#] + Sin[#]&;
```

where the argument of the function is not specified but is represented by `#`, and the ampersand `&` is used to terminate the function. If there is more than one argument they are represented by `#1, #2` etc. One can determine values of the function for given values of the argument, and also obtain the derivative, for example, as another pure function:

```
In[5]:= f1[Pi/4]

Out[5]= Sqrt[2]

In[6]:= f1'

Out[6]= Cos[#1] - Sin[#1] &
```

Starting with version 5 of *Mathematica* it is sometimes necessary to specify that the function will only be evaluated if the argument if numerical (rather than symbolic) if it is be subsequently used in numerical routines like `FindRoot` or `NDSolve` discussed in Sec. 2 below. This is done by adding the hieroglyphics `?NumericQ` to the argument of the function when it is defined (we mentioned `NumericQ` in the previous subsection). An example is

```
In[1]:= fun[x_?NumericQ] := x^2

In[2]:= fun[3]

Out[2]= 9

In[3]:= fun[k]

Out[3]= fun[k]
```

which shows that the function is evaluated when its argument is 3 but not when its argument is k. (In my view, this requirement is a retrograde step; it makes *Mathematica* less intuitive and easy to use.)

1.10 Listable Functions and Map

Most *Mathematica* functions act separately on each element of a list,

```
In[10]:= a = {1, 2, 3}

Out[10]= {1, 2, 3}

In[11]:= b = a^3

Out[11]= {1, 8, 27}
```

They are said to be listable functions . However, some functions defined by the user are not listable. To make sure that you act with the function on each element of the list use the command `Map[f, list]` which applies the command `f` to each element of `list`, e.g.

```
In[32]:= a = {1, 2, 3, 4}

Out[32]= {1, 2, 3, 4}

In[33]:= g[x_] := 2
```

```
In[34]:= g[a]

Out[34]= 2

In[35]:= Map [g, a]

Out[35]= {2, 2, 2, 2}
```

Note, though, that many user defined functions do act on each element of a list. In the example below, we take the square of the list a:

```
In[33]:= f[x_] := x^2

In[34]:= f[a]

Out[34]= {1, 4, 9, 16, 25}
```

2 Numerical Capabilities

2.1 Basic Numerical Calculations

Mathematica can be used to do numerical calculations line a calculator, e.g. it can evaluate

$$\sqrt{\frac{(6.9 \times 10^{39})(7.4 \times 10^{-49})}{3.69762 \times 10^{11}}}$$

```
In[5]:= Sqrt[ (6.9 * 10^39) (7.4 * 10^-49) / (3.69762 * 10^11) ]

                   -10
Out[5]= 1.17511 10
```

2.2 Exact Arithmetic

Mathematica treats integers and rational fractions as exact. When we give it exact values as input it tries to return an exact answer:

```
In[6]:= 1/12 - 7/9 + 31/36

        1
Out[6]= -
        6

In[7]:= Sqrt[4]

Out[7]= 2

In[8]:= Sqrt[5]

Out[8]= Sqrt[5]
```

Note that since there is no exact value for $\sqrt{5}$ *Mathematica* leaves it unevaluated. If an argument of a function is a numerical value with a decimal point, *Mathematica* assumes that it is an approximate value and returns a numerical value, e.g.

```
In[9]:= Sqrt[5.]

Out[9]= 2.23607
```

2.3 Precision

If we want a numerical value we use the important command `N[x]` which returns the numerical value of `x`. By default, decimal numbers are stored to 16 digits precision , but higher precision can be specified using `N[x, k]` which evaluates `x` to `k` significant digits, e.g.

```
In[8]:= Sqrt[5]

Out[8]= Sqrt[5]

In[9]:= N[%8]

Out[9]= 2.23607

In[10]:= N[%8, 100]

Out[10]= 2.2360679774997896964091736687312762354406183596115257242708972454105209256378048994144144083787822275
```

```
In[11]:= Precision[Sqrt[5]]

Out[11]= \[Infinity]

In[12]:= Precision[%9]

Out[12]= 16
```

We have used the command `Precision[x]` to show the precision that *Mathematica* has for the variable `x`. The last line shows that although *Mathematica* only prints numerical results with the default precision to 6 digits, the number is actually stored to 16 digits of precision. To display all the digits that *Mathematica* knows about use the command `InputForm` i.e.

```
In[13]:= InputForm[%9]

Out[13]//InputForm= 2.23606797749979
```

If a number is given in decimal form, *Mathematica* automatically assumes that it is default precision and so some function of that number cannot be obtained with greater precision. For example, in

```
In[3]:= N[Log[2.79], 50]

Out[3]= 1.02604
```

we only get the log to default precision even though we asked for 50 decimal places, because 2.79 was assumed to have only default precision. If we want to indicate that a decimal number has greater than default precision we add a backquote " ` " at the end of it followed by the number of decimal places. For example, with

```
In[4]:= N[Log[2.79`50], 50]

Out[4]= 1.0260415958332742606836718890186802045427170688685
```

we now get $\log 2.79$ correct to 50 places.

If a precision greater than the default is specified, *Mathematica* keeps track of the actual precision and only prints the number of digits that it believes to be correct. In the following example, the precision is reduced by subtracting two numbers which are nearly equal.

```
In[1]:= x = N[1 - 1/10^14, 18]

Out[1]= 0.99999999999999000

In[2]:= y = 1 - x

                -14
Out[2]= 1.000 10

In[3]:= Precision[y]

Out[3]= 4
```

We see that *Mathematica* correctly recognizes that there are only 4 digits of precision. However, if we work with default precision, in the interests of efficiency *Mathematica* does not keep track of how many digits of precision there actually are, but just assumes it is the default, e.g.

```
In[18]:= N[1 - 1/10^14]

Out[18]= 1.

In[19]:= 1 - %

                   -15
Out[19]= 9.99201 10

In[20]:= Precision[%]

Out[20]= 16
```

In fact the last result only has two digits of precision.

Mathematica knows about many functions, and give numerical values of them to arbitrarily high precision anywhere in the complex plane. For example, the Bessel function $J_n(x)$ is called `BesselJ[n, x]`. Suppose we want to evaluate $J_3(1+2i)$ to 100 digits. This is easily accomplished:

```
In[1]:= N[BesselJ[3, 1 + 2I], 100]

Out[1]= -0.28103966684576790767179865444025092304035621980956282410110023340\

>       3

>      0.04501788320344931842181451329193804 +

>     0.01717506200339023212714254881178061301156905191342715771433420653675\

>       65 I

>       06012469307750659043622266432 0
```

It would be very challenging to obtain this result from a C or fortran program.

2.4 Vectors and Matrices

In *Mathematica* vectors are represented by lists, e.g. a vector

$$\begin{pmatrix} a \\ b \\ c \end{pmatrix}$$

would be represented by the list {a, b, c}. A matrix is represented by a nested list, e.g.

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

is represented by { {a b c}, {d e f}, {g h i} }.

Mathematica will output the matrices in rows and columns (rather than as nested lists) by giving the command MatrixForm[mat], where mat is the matrix. In the example below we accomplish the same thing by appending //MatrixForm at the *end* of the line. This adding a command as an "afterthought" is an example of a "postfix" way of giving a command. Many *Mathematica* commands can given in a postfix form. For example Sqrt[5] //N gives the same result as N[Sqrt[5]]. In many cases the postfix form is clearest because it separates the two commands.

Matrix multiplication is indicated by putting a dot between the two matrices, e.g.:

```
In[1]:= m1 = {{0, 1}, {1, 0}};

In[2]:= %1 //MatrixForm

Out[2]//MatrixForm= 0   1
                    1   0

In[3]:=  m2 = {{0, -1}, {1, 0}};

In[4]:= %3 //MatrixForm

Out[4]//MatrixForm= 0    -1

                    1    0

In[5]:= mprod = m1.m2;

In[6]:= %5 //MatrixForm

Out[6]//MatrixForm= 1    0

                    0    -1
```

Many other functions are also available for manipulating matrices, such as determining the inverse, the determinant , eigenvalues and eigenvectors . The following example takes the matrix

$$\begin{pmatrix} 3 & 4 \\ 4 & -3 \end{pmatrix}$$

and finds that its determinant is -25, its inverse is

$$\begin{pmatrix} 3/25 & 4/25 \\ 4/25 & -3/25 \end{pmatrix}$$

and that its eigenvalues are ± 5 with corresponding (unnormalized) eigenvectors $(-1, 2)$ and $(2, 1)$:

```
In[1]:= m = { {3, 4}, {4, -3} }

Out[1]= {{3, 4}, {4, -3}}

In[2]:= Det[m]
```

```
Out[2]= -25

In[3]:= Inverse[m]

          3   4     4     3
Out[3]= {{--, --}, {--, -(--)}}
          25  25    25    25

In[4]:= Eigensystem[m]

Out[4]= {{-5, 5}, {{-1, 2}, {2, 1}}}
```

If the elements of the matrix are given exactly, *Mathematica* will try to obtain analytical results for eigenvalues and eigenvectors. This takes a lot of time if the matrix is large, and you probably then want a numerical result. To do this, tell *Mathematica* to use the numerical values of the matrix, e.g. to find the numerical eigenvalues of a matrix `mat` the command is `Eigenvalues[ N[mat] ]`.

2.5 Numerical Solution of Polynomial Equations; Transformation Rules

Numerical solutions of polynomial equations are obtained from `NSolve`, e.g.

```
In[8]:=  NSolve [ x^3 + 5x^2 + 5x + 1 == 0 ]

Out[8]= {{x -> -3.73205}, {x -> -1.}, {x -> -0.267949}}
```

Note two things about this. Firstly an equation is written *lhs == rhs* with a double equals sign. The operator "=" is for an assignment and the operator "==" is for equations. One of the most common mistakes of beginners is to use a single "=" for equations.

Secondly *Mathematica* gives the solution as a *rule* in the form `x -> sol` which is read as " x goes to sol" or "x is replaced by sol". These "transformation rules" occur in conjunction with the "replacement operator" "`/.`" . To make a replacement, we write

```
expr /.  variable -> value
```

which is read as "evaluate `expr` given that `variable` is replaced by `value`". Note that there is no space between `-` and `>` or between the `/` and the `.` . Transformation rules are very frequently used in *Mathematica* so you *must* be familiar with them. Here is a simple example

```
In[9]:= 2w -79 /. w -> 100

Out[9]= 121
```

We can therefore extract a list of the roots of the cubic equation from the transformation rule in `Out[8]` that `NSolve` as follows,

```
In[10]:= x /. %8

Out[10]= {-3.73205, -1., -0.267949}
```

If several replacements are to be made, one puts them in a list, e.g.

```
In[11]:= x^2 + 2 y /. {x -> 1, y -> 2}

Out[11]= 5
```

Just as one can give a delayed assignment see Sec. 1.9, rather than an immediate assignment, one can give a delayed transformation rather than an immediate transformation rule by using `expr /.  lhs :> rhs` rather than `expr /.  lhs -> rhs`, see Tam[2] p. 68. (There is no space between `:` and `>`.)

2.6 Numerical Solution of General Equations and finding minima

To find numerical solutions of non-polynomial equations use the command `FindRoot[lhs == rhs , {x, x0}` , where `x0` is an initial guess for `x` to be used by the algorithm. FindRoot is an example of a *Mathematica* command whose name is made by combining two or more words. It is a general rule that the first letter of each of the words is capitalized. Whereas `NSolve` finds all the solutions of a polynomial, `FindRoot` will at most give one root. If the equation has more than one root, which one is obtained depends on the choice of the initial guess. Here is an example of `FindRoot`:

```
In[9]:= FindRoot[ Sin[x] == 0.5 , {x, 1} ]

Out[9]= {x -> 0.523599}
```

If *Mathematica* cannot determine the derivative of the function then you must give two values as starting points, i.e. `FindRoot[lhs == rhs , {x, x0, x1}]`.

It is possible to find a root, i.e. a zero, of an expression `f[x]` without writing it explicitly as an equation as follows: `FindRoot[f[x], {x, x0}]`.

By default, `FindRoot` uses machine precision, which has about 16 digits of accuracy. This can be increased with the option `WorkingPrecision`, e.g.

```
In[1]:= FindRoot[x == Cos[x], {x, 0.5}, WorkingPrecision -> 50]

Out[1]= {x -> 0.739085133215160641655312087673873404013411758900 76}
```

Let's verify that this precision was indeed obtained:

```
In[2]:= x - Cos[x] /. %1

                -49
Out[2]= 0. 10
```

It is also possible to locate minima of a function with the command `FindMinimum[f[x], {x, x0}]`, where `x0` is the starting point of the search, if one can compute analytically the derivative of $f(x)$. If the derivative cannot be obtained analytically, one needs to give two starting values, i.e. `FindMinimum[f[x], {x, x0, x1}]`.

2.7 Numerical Integration

Mathematica can perform numerical integration. The function `NIntegrate[f, {x, xmin, xmax}]` will numerically evaluate the integral

$$\int_{x_{min}}^{x_{max}} f(x)\,dx.$$

As an example we evaluate

$$\int_0^\infty \exp(-\sqrt{x+x^2})\,dx$$

```
In[10]:= NIntegrate[ Exp[ -Sqrt[x + x^2] ], {x, 0, Infinity} ]

Out[10]= 0.696669
```

2.8 Numerical Solution of ODEs

Ordinary differential equations (ODEs) can be integrated numerically using `NDSolve[ eqns, y, {x, xmin, xmax} ]`. Here `eqns` is a set of equations (arranged as a list), namely the differential equation and the boundary conditions, the equation is then solved for `y` with independent variable `x` in the range from `xmin` to `xmax`. Note that the boundary conditions , as well as the ODE, must be given as "equals" (with "==") rather than an assignment. As an example, consider the damped simple harmonic oscillator

$$\frac{d^2x}{dt^2} + \gamma\frac{dx}{dt} + \omega_0^2 x = 0.$$

We set $\omega_0^2 = 9s^{-2}, \gamma = 0.5s^{-1}, x(0) = 1$ m, and $x'(0) = 0$. We numerically determine $x(t)$ in the range 0 to 12 and assign the result to a variable `sol`.

```
In[14]:= sol =  NDSolve [ {x''[t] + 0.5 x'[t] + 9 x[t] == 0 , \
                x'[0] == 0, x[0] == 1}, x, {t, 0, 12} ]

Out[14]= {{x -> InterpolatingFunction[{{0., 12.}}, <>]}}
```

Note that derivatives are conveniently represented by apostrophes. The result is an interpolating function from which *Mathematica* can determine the value of x at a given value of t. For example, to determine $x(2)$,

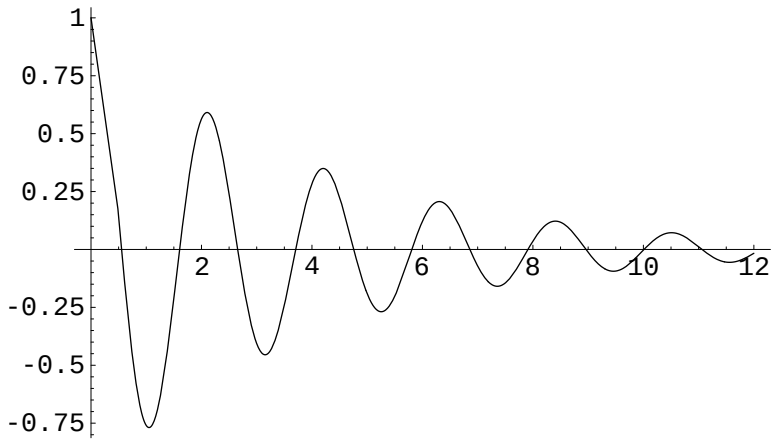
```
In[15]:= x[2] /. sol

Out[15]= {0.563522}
```

Note that there is still a curly bracket in the output because *Mathematica* puts two curly brackets round the transformation rule in the `Out[14]=` line above (making it a list) in order to deal with equations which have several solutions. One can get rid of this by picking out the first item in the list (even though in this case there's only 1):

```
In[16]:= x[2] /. sol[[1]]

Out[16]= 0.563522
```

Probably the most useful thing that one can do with the interpolating function is to plot it. We will discuss the `Plot` function in more detail in Sec. 4, but here we just use it to plot the solution to our differential equation.

```
In[17]:= Plot[x[t] /. sol, {t, 0, 12}]

Out[17]=
```



We see the expected damped oscillations.

Derivatives of the solution can also be plotted, e.g.

```
In[18]:= Plot[x'[t] /. sol, {t, 0, 12}]
```

(The resulting graph is not shown here.)

In "`In[14]`" above we have determined `sol` using an immediate assignment which means that the equation is immediately solved and stored in `sol`. If we refer to `sol` later, the equation is not solved again, but the required information is obtained from the already existing solution. Usually this is what you want. However, sometimes the equation may depend on a parameter and we want to re-solve the equation many times for different parameter values. We shall see some examples of this in the course. In this case one gives a delayed assignment, i.e. `sol :=`

`NDSolve` ···, which means that *each* time `sol` is referenced the equation is solved from scratch using the current parameter value. Note that this is definitely *not* not a good idea if you simply want to plot the solution. It is very wasteful to solve the equation from scratch for each data point to be plotted.

It is often more convenient to express the answer as a function rather than in the form above in which `sol` is a transformation rule. We can do this, for example, by giving a second command

```
In[19]:= ans[t_] := x[t] /. sol[[1]]
```

We can then get the solution, or even derivatives of the solution, by treating `ans[t]` as a function, e.g.

```
In[21]:= ans[1]

Out[21]= -0.759954

In[22]:= ans'[1]

Out[22]= -0.355062
```

Note that in `In[19]` just above we used a delayed assignment. This is convenient because `ans[t]` will still work even if the differential equation is been solved again with new parameters, to get a new transformation rule `sol`. It is also useful to express the *analytical* solution of differential equations, obtained from `DSolve` discussed in Sec. 3.9 below, as a function rather than a transformation rule.

2.9 Kepler Problem

`NDSolve` can also solve *sets* of differential equations. As an example we consider the problem of the orbit of a planet round the sun, which we know is an ellipse. From Newton's laws of motion and gravitation the equations of motion are

$$m\frac{d^2\vec{r}}{dt^2} = -GMm\frac{\hat{r}}{r^2},$$

where m is the mass of the planet, M the mass of the sun, G is the gravitational constant, $\vec{r}$ is the vector from the sun to the planet, and $\hat{r}$ is a unit vector in the direction of $\vec{r}$. Taking the orbit to be in the $x-y$ plane, we can write this vector equation as two ordinary differential equations

$$\begin{aligned}\frac{d^2x}{dt^2} + \frac{GMx}{(x^2+y^2)^{3/2}} &= 0\\ \frac{d^2y}{dt^2} + \frac{GMy}{(x^2+y^2)^{3/2}} &= 0\end{aligned}$$

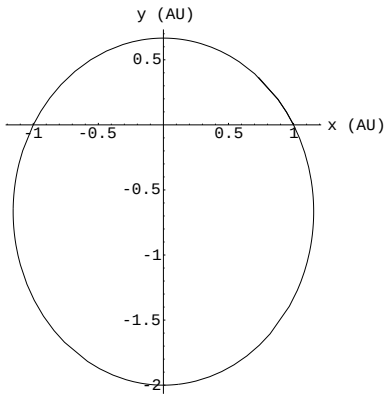
In a system of units where the unit of length is the astronomical unit (AU), the distance of the semi-major axis of the (elliptical) orbit of the earth, and the unit of time is one year, then it turns out that $GM = 4\pi^2$, see Tam[2] and the reference therein. We solve these equations starting from $t = 0$ assuming that $x(0) = 1, y(0) = 0, x'(0) = -\pi, y'(0) = 2\pi$. We integrate up to $t = 1.6$, which turns out to be enough to see one complete orbit (this could be determined by trial and error, or, in this case, from the analytical solution).

`NDSolve` finds a numerical solution as follows:

```
In[1]:= orbit = NDSolve [{x''[t] + (4 Pi^2) x[t]/(x[t]^2 + y[t]^2)^(3/2) == 0,
         y''[t] + (4 Pi^2) y[t]/(x[t]^2 + y[t]^2)^(3/2) == 0,
    x[0] == 1, y[0] == 0, x'[0] == -Pi, y'[0] == 2 Pi},
    {x, y}, {t, 0, 1.6} ]

Out[1]= {{x -> InterpolatingFunction[{{0., 1.6}}, <>],

>     y -> InterpolatingFunction[{{0., 1.6}}, <>]}}
```

Now we plot the path of the planet (the plotting routine `ParametricPlot` will be explained in Sec. 4.5):

```
In[2]:= ParametricPlot[ Evaluate [{x[t], y[t]} /. orbit],
{t, 0, 1.6},
AxesLabel -> {"x (AU)", " y (AU)" } ]
Out[2]=
```



One sees the expected elliptical orbit. Notice that this is much easier than computing the orbit from a C or fortran program using one of the routines for integrating ODEs (such as fourth order Runge-Kutta), producing a set of (x, y) values, and then invoking a separate graphics program to plot the results. Here we got everything with two commands within a single program.

3 Symbolic Capabilities

3.1 Algebraic Expressions

Mathematica has many functions for manipulating algebraic expressions . Here is an example which illustrates two of them, `Expand` and `Factor` (see the Refs. for information on additional functions)

```
In[1]:= Expand[ (x-1)^2 (2x+7) (5x - 3) ]

                          2      3       4
Out[1]= -21 + 71 x - 69 x  + 9 x  + 10 x

In[2]:= Expand[ (x-1)^4 (5x - 3)^3 (3x^2 + 4x - 5) ]

                              2          3          4          5         6
Out[2]= 135 - 1323 x + 5526 x  - 12694 x  + 17076 x  - 12764 x  + 3514 x  +

           7         8        9
>    1830 x  - 1675 x  + 375 x

In[3]:= Factor [%2/%1]

                2           2                2
        (-1 + x)  (-3 + 5 x)  (-5 + 4 x + 3 x )
Out[3]= ----------------------------------------
                      7 + 2 x
```

3.2 Simplifying Expressions

`Simplify[expr]` is a useful general purpose for simplifying expressions , though sometimes *Mathematica*'s definition of simplest may be different from yours. It is sometimes tricky to find which *Mathematica* function will simplify an expression in the way you want. Here is an example of `Simplify`

```
In[1]:= Simplify[Cos[x]^2  - Sin[x]^2]

Out[1]= Cos[2 x]
```

It may be useful to specify additional conditions as a second argument to `Simplify` to get the maximum simplification. For example, $\sqrt{x^2}$ is not always equal to x, and *Mathematica* knows this:

```
In[2]:=  Simplify[Sqrt[x^2]]

                2
Out[2]= Sqrt[x ]
```

However, if you specify that $x > 0$, then $\sqrt{x^2}$ *is* always equal to x:

```
In[3]:= Simplify[Sqrt[x^2], x > 0]

Out[3]= x
```

You can also specify that x is a real number (or integer etc.) using the `Element` function[6] and get the correct result:

```
In[21]:= Simplify[Sqrt[x^2], Element[x, Reals]]

Out[21]= Abs[x]
```

(Note: if you are using the graphical interface this can be done more elegantly writing the condition as $x \in$ Reals where $\in$ can be got from the Basic Input palette. If you want to specify that x is an an integer the command is `Element[x, Integers]`.)

`FullSimplify[expr]` is a very powerful function which applies a wide range of transformations involving both elementary functions, like polynomials, and special functions. As an example one can prove the following identity involving Bessel functions, $J_n(z)$;

$$J_{-n}(z)J_{n-1}(z) + J_{-n+1}(z)J_n(z) = \frac{2\sin(n\pi)}{\pi z}.$$

`Simplify` does not know about this but `FullSimplify` does:

```
In[4]:= FullSimplify[ BesselJ[-n, z] BesselJ[n-1, z] + \
        BesselJ[-n+1, z] BesselJ[n, z] ]

        2 Sin[n Pi]
Out[4]= -----------
           Pi z
```

There are a number of functions for regrouping expressions. For example, as noted in Sec. 3.1, `Factor` breaks up an expression into its factors and `Expand` multiplies factors and expands them out. The function `ExpandAll` further expands subexpressions while `ExpandNumerator` and `ExpandDenominator`, as their names suggest, just act on the numerator and denominator of the expression.

Sometimes it is difficult to get *Mathematica* to rearrange expressions as you would like. Consider the following example in which you want to get the factors over a common denominator:

```
In[3]:= f = 1/(x^2 - 16) - ((x+4)/(x^2 - 3x - 4))

           1           4 + x
Out[3]= -------- - -------------
              2                2
        -16 + x    -4 - 3 x + x
```

```
In[4]:= Simplify[%]

            4 + x           1
Out[4]= ------------ + --------
                   2          2
        4 + 3 x - x    -16 + x
```

`Simplify` evidently doesn't consider that putting the terms over a common denominator is simpler than the original expression. However, `Together` will do what we want:

```
In[5]:= Together[f]

                            2
              -15 - 7 x - x
Out[5]= ------------------------
        (-4 + x) (1 + x) (4 + x)
```

`ComplexExpand` expands out expressions, including trigonometric expressions, assuming the variables are real:

```
In[6]:= ComplexExpand[Sin[x + I y]]

Out[6]= Cosh[y] Sin[x] + I Cos[x] Sinh[y]
```

Apart from `ComplexExpand`, `Simplify` and `FullSimplify`, the other functions for transforming algebraic expressions do not change trigonometric expressions, so *Mathematica* provides several functions which act on trigonometric expressions. Two of the most useful are `TrigExpand`, which expands trigonometric expressions out, and `TrigReduce` which applies multiple angle identities to simplify the expression:

```
In[7]:= TrigExpand[Cos[10x]]

               10             8       2             6       4
Out[7]= Cos[x]   - 45 Cos[x]  Sin[x]  + 210 Cos[x]  Sin[x]  -

                4       6             2       8         10
>    210 Cos[x]  Sin[x]  + 45 Cos[x]  Sin[x]  - Sin[x]

In[8]:= TrigReduce[%]

Out[8]= Cos[10 x]
```

Sometimes `TrigToExp`, which writes trigonometric functions in terms of exponentials, and its inverse, `ExpToTrig`, are also useful.

Frequently it is an art to get *Mathematica* to put an expression in the form which *you* think is the simplest.

3.3 Solving Polynomial Equations

`Solve[lhs == rhs, var]` attempts to solve exactly an equation for the variable `var`. Usually `eqn` has to be a polynomial equation in order that `Solve` will find the exact solution, but some other equations can also be solved. Consider the equation

$$x^2 - 3x - 10 = 0$$

```
In[5]:= sol = Solve [ x^2 - 3x - 10 == 0, x]

Out[5]= {{x -> -2}, {x -> 5}}
```

The solution is again in the form of a list of transformation rules. To obtain just a list of solutions apply the replacement operator /.

```
In[6]:= x /. sol

Out[6]= {-2, 5}
```

If you want just one of the solutions, you have to extract the appropriate element of the list, e.g. to get the second root in the above example,

```
In[7]:= x /. sol[[2]]

Out[7]= 5
```

`Solve` can also solve sets of simultaneous equations. For example, consider the system of equations

$$\begin{aligned} x + y &= 10 \\ 3x - 5y &= 38 \end{aligned}$$

```
In[7]:= Solve [ {x + y == 10, 3x - 5y == 38}, {x, y} ]

Out[7]= {{x -> 11, y -> -1}}
```

`Solve` gives *generic* solutions, which may not be correct in all cases. For example, for the equation $ax = b$ the generic solution $x = b/a$ is not valid for the special case $a = 0$. The command `Reduce` will take special cases into account. Compare

```
In[8]:= Solve [a x == b, x]

                b
Out[8]= {{x -> -}}
                a
```

with

```
In[9]:= Reduce [a x == b, x]

                                    b
Out[9]= a == 0 && b == 0 || x == - && a != 0
                                    a
```

where the logical *or* is written `||`, the logical *and* as `&&`, and the logical *unequal* as `!=` or $\neq$.

3.4 Differentiation

Mathematica can take the derivative of functions with the command `D[f, x]` which give the derivative of `f[x]` with respect to `x`, e.g. we can differentiate the function:

$$\frac{\cos 4x}{1 - \sin 4x}$$

```
In[3]:= D[Cos[4x] / (1 - Sin[4x]), x]

                     2
          4 Cos[4 x]        4 Sin[4 x]
Out[3]= --------------- - ------------
                      2   1 - Sin[4 x]
        (1 - Sin[4 x])

In[4]:= Simplify[%]

             -4
Out[4]= -------------
        -1 + Sin[4 x]
```

which also illustrates the use of `Simplify`. `D[f, {x, n}]` gives $\partial^n f/\partial x^n$, and `D[f, x1, x2, ... ]` gives

$$\frac{\partial}{\partial x_1}\frac{\partial}{\partial x_2}\cdots f,$$

the multiple partial derivative. Similarly `D[f, {x, n}, {y, m}, ··· ]` gives the result of differentiating $f(x)$ n times by x, m times by y etc.

Derivatives of functions of a single variable can be conveniently performed by putting an apostrophe (') after the name of the function, e.g.

```
In[5]:= f[x_] = Log[Sin[x]]

Out[5]= Log[Sin[x]]

In[6]:= f'[x]

Out[6]= Cot[x]
```

This notation is often used when solving differential equations, see Secs. 2.8 and 3.9.

3.5 Integration

Mathematica can also do indefinite and definite integrals. `Integrate[f, x]` gives the indefinite integral of `f` with respect to `x`. For example we can get

$$\int \frac{x^4}{a^2+x^2}dx$$

```
In[1]:= Integrate[x^4/(a^2 + x^2), x]

                   3
          2      x     3        x
Out[1]= -(a  x) + -- + a  ArcTan[-]
                   3             a

In[2]:= D[%, x]

                       2
          2    2     a
Out[2]= -a  + x  + ------
                        2
                       x
                   1 + --
                        2
                       a

In[3]:= Simplify[%]

           4
          x
Out[3]= -------
         2    2
        a  + x
```

in which we have checked that differentiating the result gives back the integrand (at least after we used `Simplify`).

The definite integral

$$\int_{x_{min}}^{x_{max}} f\,dx$$

is done by `Integrate[f, {x, xmin, xmax}]`, e.g.

$$\int_1^2 \frac{x^4 \log(x(x+3))}{\sqrt{x}}\, dx$$

```
In[4]:= Integrate[ x^4 Log[x(x+3)]/ Sqrt[x], {x, 1, 2} ]

        92728   88612 Sqrt[2]                                          2
Out[4]= ----- - ------------- - 6 Sqrt[3] Pi + 36 Sqrt[3] ArcTan[Sqrt[-]] -
        2835        2835                                               3

     4 Log[2]   32 Sqrt[2] Log[10]
>    -------- + ------------------
        9               9
```

This is the sort of calculation that would be tedious to do by hand but which *Mathematica* excels at.

Sometimes *Mathematica* needs to be told that parameters lie in a certain range to get a simple answer. For example

$$\int_0^\pi \frac{dx}{a + b\cos x} = \frac{\pi}{\sqrt{a^2 - b^2}}\,,$$

assuming that $a > b > 0$. If we just ask *Mathematica* to do the integral we get a messy answer:

```
In[1]:= Integrate[1/(a + b Cos[x]), {x, 0, Pi}]

Out[1]= If[Re[a] > Re[b] && Re[a + b] > 0 ||

>     Re[a] < Re[b] && Re[a + b] < 0 || Im[a] != Im[b] || Im[a + b] != 0,

                        2
              Sign[a - b]                2    2
     Pi Sqrt[-------------] Sqrt[Sign[-a  + b ]]
                  2    2
              Sign[a  - b ]
>    ----------------------------------------,
                   2    2
              Sqrt[-a  + b ] Sign[a - b]

                    1
>    Integrate[------------, {x, 0, Pi},
               a + b Cos[x]

>     Assumptions ->

>      !(Re[a] > Re[b] && Re[a + b] > 0 || Re[a] < Re[b] && Re[a + b] < 0 ||

>        Im[a] != Im[b] || Im[a + b] != 0)]]
```

However, if we simplify the result using `Simplify` with the required condition $a > b > 0$ as the second argument we get the expected result:

```
In[2]:= Simplify[Integrate[1/(a + b Cos[x]), {x, 0, Pi}], a > b > 0]

             Pi
Out[2]= -------------
              2    2
        Sqrt[a  - b ]
```

This can also be accomplished in a more elegant manner by using the postfix form of `Simplify`, i.e. adding it at the *end* of the line as follows:

```
In[3]:= Integrate[1/(a + b Cos[x]), {x, 0, Pi}] // Simplify[#, a > b > 0]&

              Pi
Out[3]= -------------
                2    2
        Sqrt[a  - b ]
```

Note that `#` refers to the result of the previous command, `Integrate` in this case, and the `Simplify` command is terminated by the `&` symbol, as with pure functions discussed in Sec. 1.9.

The same result can be obtained from by using the function `Assuming` as follows:

```
In[1]:= Assuming[ a > b > 0, Integrate[1/(a + b Cos[x]), {x, 0, Pi}]]

              Pi
Out[1]= -------------
                2    2
        Sqrt[a  - b ]
```

3.6 Sums

Mathematica can evaluate sums with `Sum[f, {i, imin, imax}]`, e.g.

$$1 + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \frac{1}{5^2}$$

```
In[5]:= Sum[ 1/n^2, {n, 1, 5}]

        5269
Out[5]= ----
        3600
```

Symbolic (also called indefinite) sums, where the upper limit is a parameter rather than a number, can also often be done, e.g.

```
In[6]:= Sum[k^3, {k, 1, n}]

         2        2
        n  (1 + n)
Out[6]= -----------
             4

In[7]:= Sum[1/k^2, {k, 1, Infinity}]

          2
        Pi
Out[7]= ---
         6
```

3.7 Series Expansions

Mathematica can work out series expansions of complicated functions to very high order. The command is `Series[f, {x, x0, n}]` which expands f in powers of $x - x_0$ up to n-th order, e.g. to get the first 20 terms in the expansion of $\exp(\sin(x))$ about $x = 0$:

```
In[8]:= Series[Exp[Sin[x]], {x, 0, 20}]

                 2    4    5    6     7       8     9          10    11
                x    x    x    x     x    31 x     x     2951 x     x
Out[8]= 1 + x + -- - -- - -- - --- + -- + ----- + ---- - -------- - ---- +
                2    8    15   240   90   5760    5670   3628800    3150

          12          13          14           15               16
     181 x      2417 x     58913 x       5699 x       52635599 x
>    -------- + -------- + ---------- - ---------- - ------------ -
     14515200   48648600   4151347200   2554051500   20922789888000

              17                 18           19                 20
      19993 x       1126610929 x       3631 x         27069353 x           21
>    ----------- + ---------------- + ----------- + ---------------- + O[x]
     43418875500   6402373705728000   34735100400   3283268567040000
```

To manipulate the series further it is often necessary to convert it to normal form, i.e. to remove the $O[x]^n$ at the end. This is done with the command `Normal[expr]`, e.g.

```
In[9]:= Series [ Sqrt[1 + x], {x, 0, 4} ]

                 2    3      4
            x   x    x    5 x        5
Out[9]= 1 + - - -- + -- - ---- + O[x]
            2   8    16   128

In[10]:= ser = Normal[%]

                  2    3      4
             x   x    x    5 x
Out[10]= 1 + - - -- + -- - ----
             2   8    16   128

In[11]:= N[{ser, Sqrt[1+x]} /. x -> 0.3]

Out[11]= {1.14012, 1.14018}
```

where we converted the series to normal form and compared the value of the series with the original function for $x = 0.3$. Note again the use of the replacement operator, `/. x -> 0.3`. It is often more convenient to use replacements rather than assigning a value to x because the replacement only affects that command, whereas the assignment will persist until x is redefined or the definition removed with `Clear[x]` or `Remove[x]`. Trouble can arise with later commands which expect x to be a variable if the assignment is not removed, as discussed in Sec. 1.5.

3.8 Limits

Limits can also be determined by *Mathematica*, e.g.

```
In[13]:= Limit[(Sin[x] - Tan[x])/x^3, x -> 0]

           1
Out[13]= -(-)
           2

In[14]:= Limit [ Sqrt[n^2 + n] - n, n -> Infinity]
```

```
         1
Out[14]= -
         2
```

Note the use of the rule `"->"`, with no space in between the two characters, in the implementation of `Limit`.

3.9 Analytical Solution of ODEs

In Sec. 2.8 we discussed the numerical solution of ordinary differential equations using `NDSolve`. Here we describe the *Mathematica* command, `DSolve` for solving ODEs, analytically, if possible. The usage is `DSolve[eqns, y[x], x]` where `eqns` is a list of the ODE and boundary conditions, and the equation is solved for `y[x]` with `x` the independent variable. Consider

$$\frac{d^2y}{dx^2} + 8\frac{dy}{dx} + 16y = 0$$

with $y = 2$ and $dy/dx = 1$ when $x = 0$:

```
In[15]:= DSolve [ {y''[x] + 8y'[x] + 16y[x] == 0, \
                y[0] == 2, y'[0] == 1}, y[x], x]

                    2 + 9 x
Out[15]= {{y[x] -> -------}}
                     4 x
                    E
```

The solution decays exponentially at large x. Note that the auxiliary equation has a repeated root, -4, which explains why there is an xe^{-4x} term as well as a e^{-4x} term.

This way of solving the differential equation, as a transformation rule for `y[x]` (the second argument of the `DSolve` command), can be used to plot the function, see Sec. 3.10, but has some disadvantages since it doesn't match anything other than `y[x]`. So, for example you can't use it to get the value of `y` for some numerical value of `x`; `y[2]` just returns `y[2]`. Also you can't plot, or obtain values for, derivatives. These difficulties can be remedied by solving the equation as a transformation rule for `y` rather than `y[x]`. The only change is to replace `y[x]` by `y` in the second argument of the `DSolve` command, (this was also used in the the `NDSolve` function in Sec. 2.8), i.e.

```
In[16]:= DSolve [ {y''[x] + 8y'[x] + 16y[x] == 0, \
                y[0] == 2, y'[0] == 1}, y, x]

                               2 + 9 x
Out[16]= {{y -> Function[{x}, -------]}}
                                4 x
                               E
```

The price you pay is the that the result is given in terms of a pure function, see Sec. 1.9, for which the notation is rather messy. However, one can then obtain the solution at specified values of `x` , e.g.

```
In[17]:= y[2] /. %16[[1]]

         20
Out[17]= --
          8
         E
```

and also obtain derivatives. Here we use the derivatives to verify analytically that the solution does satisfy the differential equation:

```
In[18]:=  y''[x] + 8y'[x] + 16y[x] == 0 /. %16[[1]] //Simplify

Out[18]= True
```

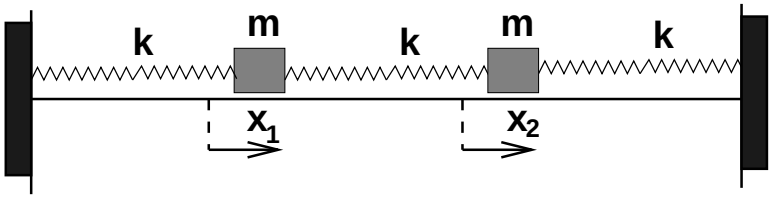
As discussed for numerical solutions of ODEs in Sec. 2 above, it is generally more convenient to write the answer as a function rather than a transformation rule. This can be done as follows:

```
In[19]:= ans[x_] = y[x] /. DSolve[{y''[x] + 8y'[x] + 16y[x] == 0, \
                y[0] == 2, y'[0] == 1}, y, x] [[1]]

        2 + 9 x
Out[20]= -------
         4 x
        E
```

One can then treat `ans[x]` as a normal function, and determine its value for different values of x, differentiate it, etc. In my view, this is the best form in which to obtain the analytical solution of an ODE.

3.10 Problem of Coupled Oscillators

`DSolve` can also solve sets of differential equations and we now use it to solve a textbook example of coupled oscillators. Consider two masses, m, sliding on a smooth surface coupled by springs with force constant k as shown. The two outer springs are also connected to fixed walls. At time $t = 0$, the mass 2 (on the right) is displaced by an amount a while mass 1 is fixed at its equilibrium position. The two masses are then released. Find an expression for the displacements of the masses at later times. Normally this would be done by finding the normal modes of oscillation, an eigenvalue problem, and then determining how much of each normal mode is excited by the initial conditions. Here we will see *Mathematica* get the solution directly without explicitly asking it to determine the normal modes.



The force on mass 1 is $-kx_1 - k(x_1 - x_2)$ and that on mass 2 is $-kx_2 - k(x_2 - x_1)$. Hence the equations of motion are

$$\begin{aligned} m\frac{d^2x_1}{dt^2} + 2kx_1 - kx_2 &= 0 \\ m\frac{d^2x_2}{dt^2} + 2kx_2 - kx_1 &= 0. \end{aligned}$$

The boundary conditions at $t = 0$ are

$$\frac{dx_1}{dt} = \frac{dx_2}{dt} = x_1 = 0, \qquad x_2 = a.$$

Defining $\omega^2 = k/m$, and using `w`, rather than ω, in the command line interface to *Mathematica* (in the notebook interface you can use Greek letters and have the output look much prettier in other ways too), we have

```
In[18]:= DSolve[ { x1''[t] + 2 w^2 x1[t] - w^2 x2[t] == 0, \
                   x2''[t] + 2 w^2 x2[t] - w^2 x1[t] == 0, \
                   x1'[0] == 0, x2'[0] == 0, x1[0] == 0,   \
                   x2[0] == a}, {x1[t], x2[t]}, t]

                   a Cos[t w]   a Cos[Sqrt[3] t w]
```

```
Out[18]= {{x1[t] -> ---------- - ------------------,
                        2                2

                a Cos[t w]    a Cos[Sqrt[3] t w]
>      x2[t] -> ---------- + ------------------}}
                    2                2
```

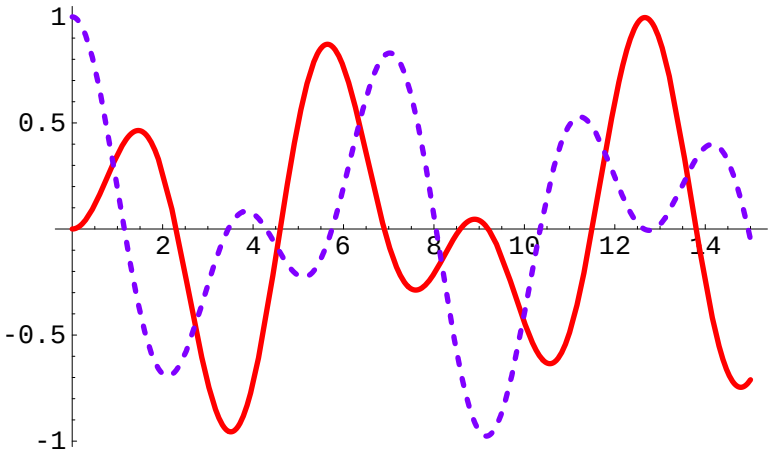
(This is the output from version 5.2; in earlier versions the result from `DSolve` was much more messy and needed to be simplified.)

One sees that there are two normal mode frequencies, at ω ($\equiv \sqrt{k/m}$) and $\sqrt{3}\omega$. It is useful to plot the results. We will first set $a = \omega = 1$,

```
In[19]:= % /. {w-> 1, a-> 1}

                     Cos[t]   Cos[Sqrt[3] t]            Cos[t]   Cos[Sqrt[3] t]
Out[19]= {{x1[t] -> ------ - --------------, x2[t] -> ------ + --------------}}
                      2            2                     2            2
```

and then plot the results (the plot options will be described in Sec. 4.2)

```
In[20]:= Plot[ Evaluate [ {x1[t], x2[t]} /. %20], {t, 0, 15}, PlotStyle -> { \
        {AbsoluteThickness[2], Hue[0]} , \
{AbsoluteThickness[2], Hue[0.75], Dashing[{0.01, 0.02} ] } } ]

Out[20]=
```



The motion is a combination of oscillations at the two normal mode frequencies.

3.11 Relational and Logical Operators

We have previously seen the importance of differentiating between assignment, `"="`, and equals, `"=="`. Let us ask what is the result if we simply give a *Mathematica* command `"lhs == rhs"`, e.g.:

```
In[4]:= y = 36; x = 6;

In[5]:= x^2 == y

Out[5]= True
```

We see that the statement `x^2 == y` is logical statement to which *Mathematica*, correctly, gives the result `True`. If the two sides had not been equal the result would have been `False`. One can even assign a variable to this true or false result and thus have both `"=="` and `"="` in the same command:

```
In[6]:= p = y == x

Out[6]= False

In[7]:= p

Out[7]= False
```

As usual, the variable `p` can be reused in subsequent expressions.

The symbol `"=="` is an example of a relational operator. Others include `"!="`, not equal, `">"`, greater than, `"<"`, less than, and so on:

```
In[8]:= 9 * 6 > 55

Out[8]= False
```

Mathematica also recognizes logical operators such as `"&&"` for "and" (the result is true only if both expressions are true), and `"||"` for "or" (the result is true if at least one of the expressions is true). This notation will be familiar to C programmers.

```
In[9]:= 4 > 6 || 9 <= 12

Out[9]= True

In[10]:= 4 > 6 && 9 <= 12

Out[10]= False
```

One can also indicate to *Mathematica* a conditional statement, i.e. to do something provided that a condition is fulfilled. The symbol for "provided that" is `/;`. For example, to define a function $f(x)$ which is 1 if $|x| < 1$ and -1 otherwise the commands are:

```
In[7]:= f[x_] := 1 /; Abs[x] < 1

In[8]:= f[x_] := -1 /; Abs[x] > 1
```

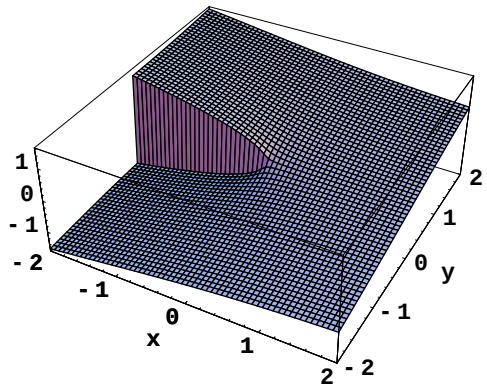
3.12 Multivalued Functions

Beginners are often surprised that *Mathematica* does not simplify $\sqrt{x^2}$ to x. Even if one applies the commands `Simplify` and even `FullSimplify`, *Mathematica* refuses to return the result x. The reason is quite simple, namely the expected result is *not* always true.

The square root is an example of a multivalued function, namely one with more than one possible result. In this case there are two possible values. However, when we calculate a function with a given argument, we expect a single answer, and this is what *Mathematica* gives. In fact, giving a single result is part of the *definition* of a function. *Mathematica* makes a choice of which solution to take depending on the value of x in the complex plane. We say that there are two "branches" of the square root function, and *Mathematica* returns the value of one of them called the "principal branch". For example, if x is real and positive then the principle branch is the positive root. Hence if $x = 2$ we have $\sqrt{x^2} = x = 2$, but if $x = -2$, then $\sqrt{x^2} = \sqrt{4} = 2$ which is *minus* x.

You should refer to a text on functions of a complex variables for a detailed analysis, but, in brief, there is a "branch cut" which terminates at $z = 0$ (we now use the symbol z rather than x to indicate that we are dealing with a complex number) and for which *Mathematica* follows the common choice of running along the negative real axis. Here we show a 3-d plot of the imaginary part of $\sqrt{z}$ (where $z = x + iy$) in the complex plane in a region which includes the origin:

```
In[1]:= Plot3D[ Im [Sqrt [x + I y] ], {x, -2, 2}, {y, -2, 2}, \\
    PlotPoints -> 60, AxesLabel -> {"x", "y", ""}]

Out[21]=
```



You clearly see the discontinuity along the negative real axis. With this choice of the branch cut, it is not difficult to show that

$$\begin{aligned} \sqrt{z^2} &= z, \qquad \text{if Re(z)} > 0 \\ \sqrt{z^2} &= -z, \qquad \text{if Re(z)} < 0 \end{aligned}$$

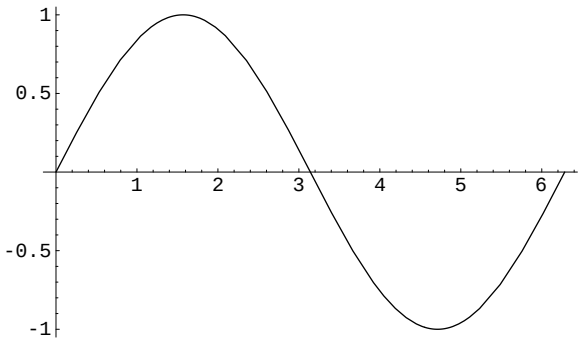
4 Plotting

One of the most powerful features of *Mathematica* is its plotting routines.

4.1 Two-Dimensional Plots

The basic plotting command is `Plot[f, {x, xmin, xmax}, opts]`, which plots $f(x)$ from x_{min} to x_{max} with options specified by `opts`. It `opts` is omitted, the default options are used. Some options will be discussed in the next section, but here we just use the default options. For example, to plot $\sin(x)$ from 0 to 2π:

```
In[2]:= Plot[ Sin[x], {x, 0, 2 Pi} ]

Out[2]=
```



Now lets do a less trivial example from physics. The normalized eigenfunctions of the hydrogen atom are

$$\psi_{nlm}(r, \theta, \phi) = R_{nl}(r) Y_{lm}(\theta, \phi),$$

where R_{nl} is the radial function and Y_{lm} is a spherical harmonic. Here we focus on the radial function which can be expressed as

$$R_{nl}(r) = \frac{1}{a_0^{3/2}} \frac{2}{n^2} \sqrt{\frac{(n-l-1)!}{[(n+l)!]^3}} \; F_{nl}\left(\frac{2r}{na_0}\right)$$

where

$$F_{nl}(x) = x^l e^{-x/2} L_{n-l-1}^{2l+1}(x)$$

where a_0 is the Bohr radius and $L_p^q(x)$ are associated Laguerre polynomials defined in *Mathematica* to be `(p+q)! LaguerreL[p, q, x]`. Hence we can define a *Mathematica* function to compute the radial wave functions as follows. Setting the Bohr radius to unity (so r will really represent r/a_0) we first of all define `f` by

```
In[14]:= f[n_, l_, x_] := x^l Exp[-x/2] (n+l)! \
                          LaguerreL[n-l-1, 2l+1, x]
```

in terms of which the function `radialFunction` can be expressed as

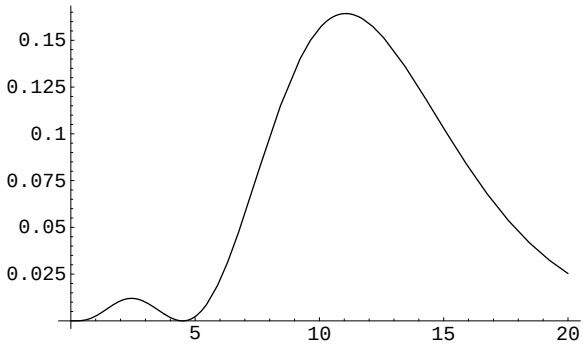
```
In[15]:= radialFunction[n_, l_, r_] := (2/n^2) Sqrt[(n-l-1)! / (n+l)!^3] \
                                       f[n, l, 2r/n]
```

To see that this works we determine the ground state wavefunction and check that it is correctly normalized, i.e. the integral of the radial probability density, $r^2 R_{nl}(r)^2\, dr$ is unity:

```
In[16]:= radialFunction[1, 0, r]

         2
Out[16]= --
          r
         E

In[17]:= Integrate[%16^2 r^2, {r, 0, Infinity}]

Out[17]= 1
```

Now let us plot the radial probability density, $r^2 R_{nl}(r)^2$ for one of these wavefunctions. We take $n = 3, l = 1$:

```
In[19]:= Plot[ r^2 radialFunction[3, 1, r]^2 , {r, 0, 20}]
```



4.2 Plot Options

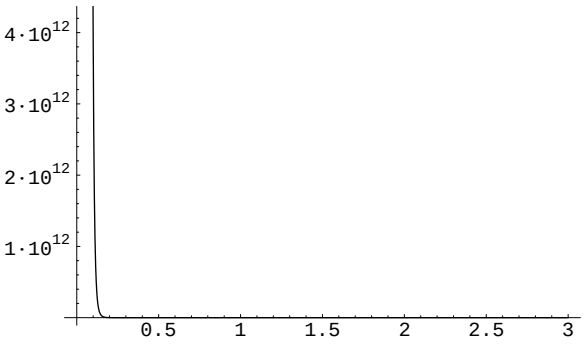
There are many plotting options. A list of these can be obtained by giving the command `Options[Plot]` or `??Plot`. A good discussion of them is given in Tam[2], p. 278-281.

Suppose we want to plot the Lennard-Jones potential

$$u(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right]$$

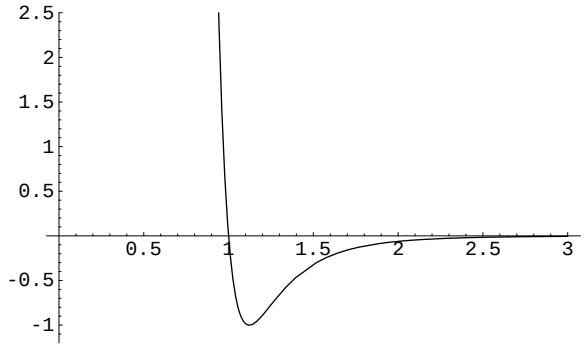
where r is the distance between the atoms, σ is a length which determines the location of the minimum of the potential, and ϵ is an energy scale. It is easy to see that the minimum is at $r = 2^{1/6}\sigma$ and $u = -\epsilon$. If we just try to plot the function (with $\epsilon = \sigma = 1$)

```
In[19]:= Plot [ 4 ( 1/r^12 - 1/r^6 ), {r, 0.001, 3} ]
```



the graph does not show us the minimum, its most important feature. The option `PlotRange -> {ymin, ymax}` specifies the range of y values that will be plotted, i.e.

```
In[20]:= Plot [ 4 ( 1/r^12 - 1/r^6 ), {r, 0.001, 3}, PlotRange -> {-1.2, 2.5} ]
```



This shows the expected minimum, and makes clear that it is asymmetric, with a very steep curve inside the minimum and a shallow curve outside the minimum.

The option `PlotRange -> {{xmin, xmax}, {ymin, ymax}}` specifies the range of x values as well as y values for the plot.

A very useful command is `Show` which can be used to redraw a plot but with different options. For example, the command

```
In[21]:= Show [ %19, PlotRange -> {-1.2, 2.5} ]
```

produces the same figure as that above.

In order to specify the style of the line, e.g. continuous or dashed, colored or black, use the option `PlotStyle`. We give more information on it in the next section where we discuss multiple plots.

It is generally useful to label the axes. This is done with the option `AxesLabel -> {xlabel, ylabel}` , where `xlabel` and `ylabel` are the labels. It is best to put the label in double quotes `"..."` to avoid the possibility that *Mathematica* will replace the label by the value of a variable of the same name. We already saw an example in the plot of the elliptical orbit in Sec. 2.9.

The default options can be changed with the `SetOptions` command as discussed in Sec. 6.3. Here is an example which makes the lines and axes thicker for the `Plot` command:

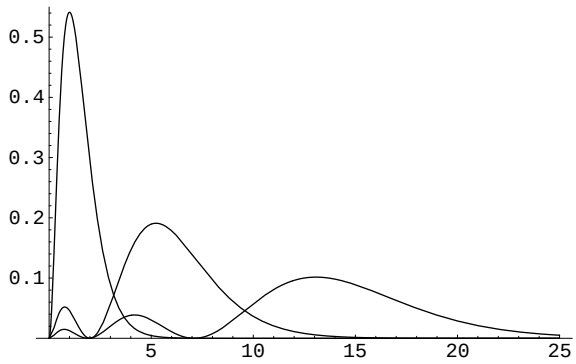
```
SetOptions[Plot, AxesStyle -> AbsoluteThickness[2], \
PlotStyle -> AbsoluteThickness[2] ]
```

The option `PlotStyle -> AbsoluteThickness[2]` will be explained in the next section.

4.3 Multiple Plots and More on Plot Options

Multiple plots can also be done by grouping the functions together into a list, i.e. `Plot[ {f1, f2, ...}, {x, xmin, xmax} ]`, or, if `Evaluate` is needed, `Plot[ Evaluate [ {f1, f2, ...} ], {x, xmin, xmax} ]`. For example, let us plot the first three radial probability densities of the hydrogen atom, discussed in the last section, with $l = 0$,

```
In[20]:= Plot[ Evaluate [ Table [ r^2 radialFunction[n, 0, r]^2, {n, 1, 3} ]],\
                 {r, 0, 25}, PlotRange->{0,0.55}]
```



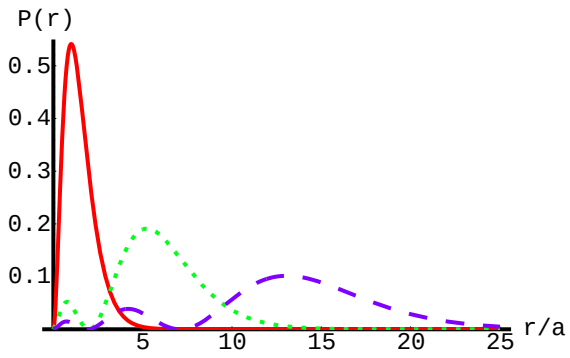
The figure is a bit hard to understand because all the lines are the same. We can assign a different style to each line with the option `PlotStyle -> {{styles for curve 1},{styles for curve 2}, ...}`. Some styles are:

- `AbsoluteThickness[r]`: The argument, an integer, specifies the thickness of the curve; the default is 1.
- `Thickness[r]`: The argument specifies the thickness of the curves as a fraction of the width of the plot.
- `RGBColor[r, g, b]`: The three arguments are the components of red, green and blue, each between 0 and 1. Hence white is `RGBColor[1, 1, 1]`, black is `RGBColor[0, 0, 0]`, red is `RGBColor[1, 0, 0]`, green is `RGBColor[0, 1, 0]`, and so on.
- `Dashing[{r1, r2, ... }]`: The arguments, $r1, r2, \cdots$, are the lengths of successive drawn and undrawn segments.
- `Hue[h]`: The color has "hue" `h` between 0 and 1, where 0 is red, 3/4 is blue and then the color goes back to 1 as hue varies between 3/4 and 1.

See Tam[2] p. 274-276 for more details.

As an example we replot the radial probability distributions of the hydrogen atom using different styles for the lines:

```
In[41]: Plot[ Evaluate [ Table [ r^2 radialFunction[n, 0, r]^2 , {n, 1, 3} ]],\
    {r, 0, 25}, PlotRange->{0,0.55}, PlotStyle -> {  \
    {AbsoluteThickness[2], Hue[0]}, \
    {AbsoluteThickness[2], Hue[0.35], Dashing[{0.01, 0.02}] }, \
    {AbsoluteThickness[2], Hue[0.75], Dashing [{0.04, 0.04} ]} }, \
AxesLabel -> {" r/a ", " P(r) "} ]
```



The original figure is in color but the colors do not appear in a black and white reproduction.

It would be even better to put text on the figure indicating which values of n correspond to each line. This can be done with the option:

```
Epilog -> Text["expr", {x, y}]
```

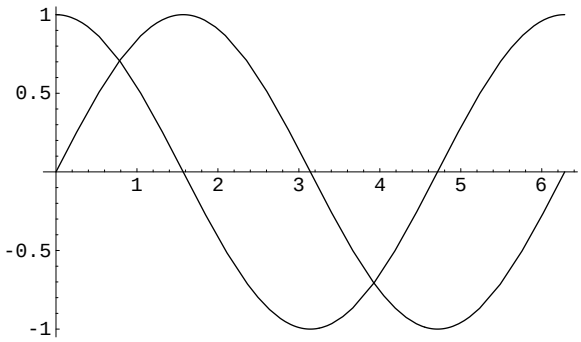
(see e.g. Tam[2] p. 417 and p. 271) which puts the text "expr" on the graph centered at point (x, y). Similarly `Epilog -> {Text["expr1", {x1, y1}], Text["expr2", {x2, y2}], ... }` puts a list of several items of text at different locations.

The command `Show` can also be used to combine plots. Suppose we create plots of two functions using two separate calls to `Plot`, e.g.

```
In[1]:= p1 = Plot[Sin[x], {x, 0, 2Pi}];

In[2]:= p2 = Plot[Cos[x], {x, 0, 2Pi}];
```

(The resulting graphics are not shown). We can then plot both $\sin(x)$ and $\cos(x)$ on the same plot with

```
In[3]:= Show[p1, p2]
```



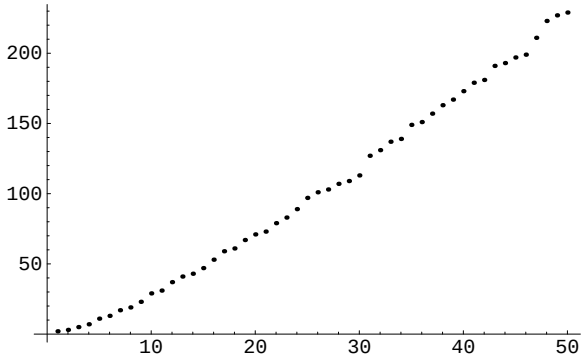
4.4 Data Plots; Example of a Random Walk

Instead of a continuous function, discrete sets of data can also be plotted using `ListPlot[{ {x1, y1}, {x2, y2}, ... }]`. As an example, consider the function `Prime[n]` which returns the n-th prime number. We can form a list of primes as follows:

```
In[4]:= Table [ {n, Prime[n]}, {n, 1, 10} ]

Out[4]= {{1, 2}, {2, 3}, {3, 5}, {4, 7}, {5, 11}, {6, 13}, {7, 17}, {8, 19},

>    {9, 23}, {10, 29}}
```

Now let us plot the first 50 prime numbers:

```
In[5]:= ListPlot [ Table [ {n, Prime[n]}, {n, 1, 50} ] ];
```



Sometimes we have the x and y coordinates as separate lists. These can be put in the form required for `ListPlot` (i.e. list of $\{x, y\}$ pairs) in a simple manner by using the function `Transpose` which interchanges the rows and columns of a matrix, e.g.

```
In[1]:= x = Table[n, {n, 1, 8}]

Out[1]= {1, 2, 3, 4, 5, 6, 7, 8}

In[2]:=  y = x^2

Out[2]= {1, 4, 9, 16, 25, 36, 49, 64}

In[3]:= data = Transpose[{x, y}]

Out[3]= {{1, 1}, {2, 4}, {3, 9}, {4, 16}, {5, 25}, {6, 36}, {7, 49}, {8, 64}}
```

so one can plot the points using `ListPlot[Transpose[x, y]]`.

As a more interesting example of a `ListPlot` we will show a random walk in one dimension. Random walks are used in many problems in physics, such as Brownian motion and fluctuations in the stock market. Starting at the origin, a particle can move one unit either to the right or to the left with equal probability at each time step. Thus at time $t = 1$ the particle has equal probability to be at $x = 1$ or -1, and at $t = 2$ the particle can be at $x = 2, 0$ or -2 with probabilities $1/4, 1/2$ and $1/4$ respectively. After time t the particle has typically wandered a distance of order $\sqrt{t}$ from the origin. We use the *Mathematica* routine `Random[]`, which generates a random number uniformly in the interval from zero to one, and then convert this to a random number which is 1 or -1 with equal probability with the following function:

```
In[1]:= r := If [ Random[] > 0.5, 1, -1 ]
```

Note that it is crucial to give a delayed assignment here. With an immediate assignment, *Mathematica* would assign to the variable `r` a *fixed* value of either -1 or 1. What we want is for the `Random` function to be called *each time* the value of r is needed. This is accomplished by the delayed assignment. The above command illustrates the use of the `If[cond, x1, x2]` command in *Mathematica*. If the condition `cond` is satisfied then `x1` is returned, otherwise `x2` is returned. Lets check that our function `r` works by generating a list of numbers

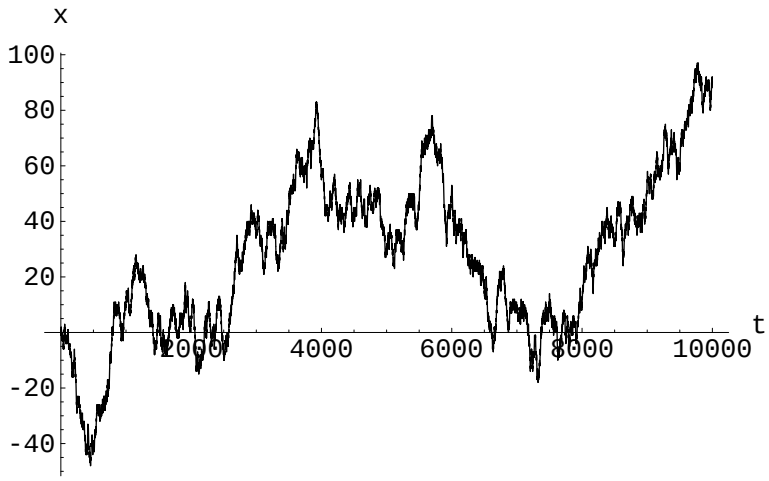
```
In[2]:= Table[r, {10}]

Out[2]= {-1, -1, -1, 1, -1, -1, -1, 1, -1, -1}
```

Notice the use of the iterator `{10}`, with just a single integer, which repeats the operation 10 times, as discussed in Sec. 1.7. We now define a function, `rwalk[n]` which will generate and plot a random walk of `n` steps:

```
In[3]:= rwalk[n_] := ( x[0] = 0; Do [x[i] = x[i-1] + r, {i, 1, n}]; \
ListPlot[Table[{i, x[i]}, {i, 0, n}], \
Joined -> True, AxesLabel -> {"t", "x"} ]; )
```

Note the option `Joined -> True` which plots lines between the points rather than the points themselves. We now generate and plot a random walk of 10000 steps:

```
In[4]:= rwalk[10000];
```

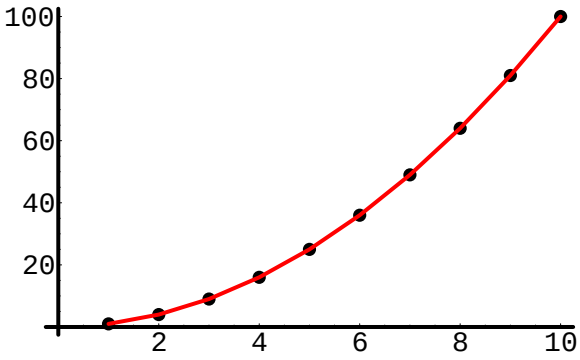


Each time `rwalk` is called, *Mathematica* generates a different set of random numbers so each random walk will be different. Notice how easy it is with *Mathematica* to get a good visual impression of what a random walk looks like.

To get *Mathematica* to plot *both* the points and the lines connecting them in a `ListPlot` plot the set of points twice, once with `Joined -> True`, and the other time with `Joined -> False`. This can all be done in one command by putting the two sets of points as a list, and the two values for the `Joined` option as a list, as follows:

```
In[28]:= points = Table[{n, n^2}, {n, 1, 10}];
In[29]:= ListPlot[{points, points}, Joined -> {True, False}]
```

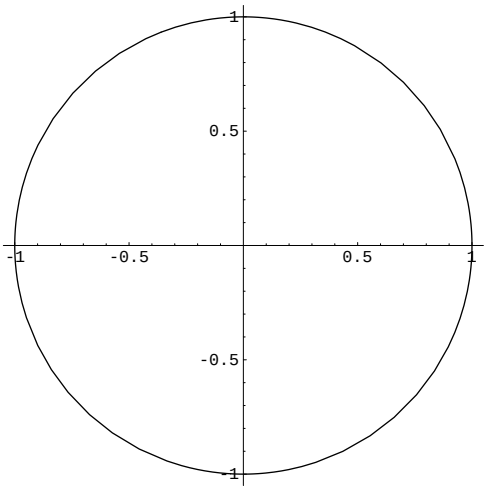
resulting graph is shown below.



4.5 Parametric Plots

The command `ParametricPlot[{fx, fy}, {t, tmin, tmax} ]` makes a parametric plot with x and y coordinates $f_x(t)$ and $f_y(t)$ which are functions of a parameter t. For example the following plot will produce a circle

```
In[6]:= ParametricPlot[ {Cos[t], Sin[t]}, {t, 0, 2 Pi}];
```



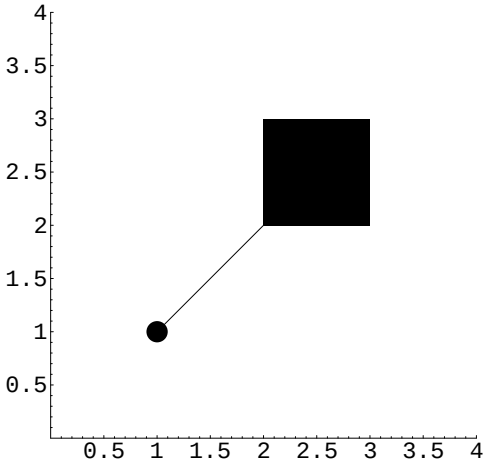
4.6 Graphics Commands

We have so far met some of the standard plotting routines such as `Plot` and `ListPlot`. Sometimes we need to have more detailed control over objects which are plotted, which is accomplished by the command `Graphics[elements, options]`. The first argument, `elements`, is a list comprising basic objects to be drawn, and "directives" for how to draw them. Examples of graphics objects are `Point[{x, y}]` which draws a point at (x, y), `Line{x1, y1}, {x2, y2}, ...` which draws a line through the points $(x1, y1), (x2, y2), \ldots$, and `Rectangle[{xmin, ymin}, {xmax, ymax}]` which draws a filled rectangle with the opposite corners specified and edges parallel to the axes. An example of a graphics directive is `PointSize[r]` which specifies the radius of the point in units of the size of the graph. Graphics directives apply to all *subsequent* graphics objects in the list `elements`. There may also be subsequent arguments to `Graphics` which are options describing how the graphics objects will be rendered. `Graphics` does not itself display the graph. This is subsequently done with `Show[graphics, options]` where graphics refers to the output from the `Graphics` command, and options gives additional options for controlling how the graph is rendered.

We will not go into (the many) details of `Graphics` but refer to the books, such as see Tam[2], for a discussion of `Graphics` commands and options. We will just content ourselves with the following simple example:

```
In[23]:= grph = Graphics[ {PointSize[0.05], Point[{1, 1}], Line[{{1, 1}, {2,
2}}], Rectangle[{2, 2}, {3, 3}] } ];

In[24]:=  Show[grph, Axes -> True, PlotRange -> {{0, 4}, {0, 4}}, AspectRatio
-> Automatic]
```

which draws a point of a specific radius, a line, and a filled rectangle:

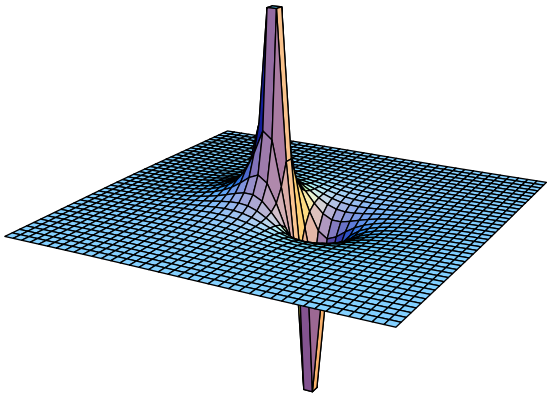


The plotting routines we met earlier, such as `Plot` and `ListPlot`, first create the objects to be displayed using `Graphics`, and then display them. To see this, type `//InputForm` after the `Plot` or `ListPlot` command. The output will then be the graphics object. (The command `InputForm` displays the full form of the command that *Mathematica* will act on rather than the shorter `Standard Form` which you usually type.) `Plot` and `ListPlot` are therefore a simpler way to plot results, without needing to understand the details of `Graphics`. There are, however, some occasions when `Graphics` is needed.

4.7 Three Dimensional Plots

An impressive feature of *Mathematica* is the ease with which three-dimensional graphics can be produced. Since we will not use much 3-D graphics in the course, I just give an example here and refer to the references, such as Tam[2] p. 301-308, for more details. The following command plots the potential, in two dimensions, due to an electric dipole made up of a charge of $+1$ at $(-1, 0)$ and a charge of -1 at $(1, 0)$. Some options have been given to make a better image. The original is in color.

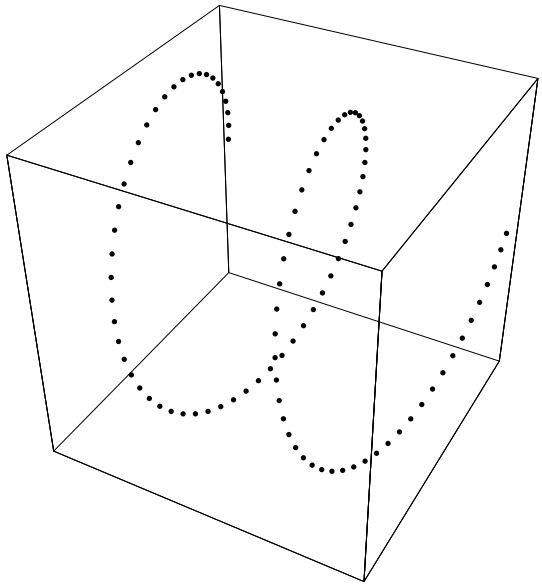
```
In[8]:= Plot3D[ 1/Sqrt[(x+1)^2 + y^2] - 1/Sqrt[(x-1)^2 + y^2], \
{x, -10, 10}, {y, -10, 10}, PlotRange -> {-2, 2}, PlotPoints -> 40, \
Boxed-> False, Axes -> False, BoxRatios -> {1, 1, 1},  \
  ViewPoint -> {1.3, -2.4, 1}]
```



Starting with Version 6, one can rotate the orientation of a 3-D plot in real time by dragging the mouse over the figure. This is very helpful.

Just as the command `Graphics`, discussed briefly in Sec. 4.6 gives the ability to create and plot "graphics elements" in two-dimensional plots, there is an analogous command, `Graphics3D`, which plots graphics elements in three-dimensions. The reader is referred to Tam[2] for details. Some commands are `Point[{x, y, z}]`, which draws a point with coordinates (x, y, z), and `Line{x1, y1, z1}, {x2, y2, z2}, ...` which draws a line through the points $(x1, y1, z1), (x2, y2, z2), \ldots$. The following example produces a list of 100 points, called `spiral` (each with three coordinates), in the form of a spiral, and then plots them.

```
In[1]:= m = 50;

In[2]:= spiral = Table[{n/m, Cos[2Pi n/m], Sin[ 2 Pi n / m]}, {n, 2m} ];

In[3]:= Show[Graphics3D[Map[Point, spiral] ] ]
```



This last command is an example of a functional program, see Sec. 5.2 below, consisting of a single line of nested commands. Note the use of `Map` which applies the command `Point` to each of the 100 triplets of numbers in the list `spiral`.

4.8 Animation

It is possible to animate a sequence of graphics in the notebook interface of *Mathematica*. This can provide amusing demonstrations. See the references for more details.

4.9 Saving and Printing Figures

In the notebook interface, figures can be saved and/or printed out by choosing an appropriate item in the menus. Explore the various menus to find out how to do this.

In the command line interface the command `Display["file", graphics, "EPS"]` will cause the graphics to be saved in encapsulated postscript format file named `file`, which can then later be printed.

5 Programming in *Mathematica*

5.1 Procedural Programming; Modules

Different styles of programming are possible using Mathematica. Procedural programming, discussed in this subsection, is quite similar to programming in traditional languages such as C or fortran. Standard constructs such as `Do, If, Switch,` and `For` are allowed[1, 2]. Sets of commands to perform some task are conveniently grouped as a function which can be called with appropriate argument(s). Different commands in the function are separated by semicolons. As an example, consider the problem in Sec. 1.7 of printing out the first $n+1$ Fibonacci numbers. This can be written as a function as follows:

```
In[1]:= firstfib[n_] := ( a[1] = 1; a[2] = 1;Print[a[0]];Print[a[1]];\
         Do [ a[k] = a[k-1] + a[k-2]; Print[a[k]], {k, 3, n} ])
```

We can then call the function for any value of the argument n:

```
In[2]:= firstfib[10]
1
1
2
```

```
3
5
8
13
21
34
55
```

You will see that the function involves new variables k and a. This might cause problems if k or a had already been defined. Hence, if the function needs new variables (in addition to its arguments), it is better if these are *local* to that function. This is accomplished by making the function a `Module` in which the local variables are declared, in a list, at the start, and Mathematica then keeps them separate from any global variables of the same name. We can promote our function to be a module as follows:

```
In[3]:= Remove[firstfib]

In[4]:= firstfib[n_] := Module[ {k, a},  a[0] = 1; a[1] = 1; \
         Print[a[0]]; Print[a[1]]; \
         Do [ a[k] = a[k-1] + a[k-2]; Print[a[k]], {k, 2, n} ] ]

In[5]:= firstfib[4]
1
1
2
3
5
```

We declared k and a to be local variables by putting them in a list, `{k, a}`, as the first argument of the module.

We will use modules quite extensively in the course. For more details see the books especially Tam[2] Secs. 3.4 and 3.6.1.

5.2 Functional Programming

A functional program consists of one line made up of many nested functions. As an example, suppose we want to determine the number of primes between n and $n + m$. The following function will do the job:

```
In[1]:= nofprimes[n_, m_] := Length[ Select[ Table[k, {k, n, n+m}], PrimeQ]]
```

Execution starts with the innermost function, `Table` which generates a list of integers between n and $n + m$. The result is then passed to `Select[... , PrimeQ]` which extracts those elements of the list which are prime. Finally the list of primes is passed to the function `Length` which gives the number of elements in the list. We can then apply this function to get the number of primes between 1000 and 3000:

```
In[2]:= nofprimes[1000, 2000]

Out[2]= 262
```

We will come across several examples of functional programming during the course. For more information see e.g. Tam[2] Sec. 3.6.2.

5.3 Rule-Based Programming

A rule-based program consists of a set of user-defined rules. An example of a rule-based program, which evaluates commutators in quantum mechanics is given in Tam[2] Sec. 3.6.3. *Mathematica* can be very powerful when used in this way, but this style of programming is not relevant for the course and so will not be discussed here.

6 Input and Output from Files

6.1 Basic I/O

It is sometimes convenient to store a set of *Mathematica* commands in a file and read then in. Here is a trivial example. The following commands are stored in a file "runmath.m" (it is conventional to give the ending ".m" to files of *Mathematica* commands)

```
sinseries = Series[ Sin[x], {x, 0, 10} ]
Normal[sinseries]
```

These can be read in and executed in a *Mathematica* session by the command `<< runmath.m` (or, equivalently, `Get[runmath.m]`):

```
In[1]:= << runmath.m

             3    5      7       9
            x    x      x       x
Out[1]= x - -- + --- - ---- + ------
            6    120   5040   362880
```

Note that only the result of the last line of the file is printed, although all the lines are executed. It is as if the whole file were on one line with the different commands separated by a semi-colon.

Under linux one can also fire up the command line interface such that "runmath.m" is read in and executed right away:

```
 ~/courses/115 => math < runmath.m
Mathematica 3.0 for Students: Linux Version
Copyright 1988-97 Wolfram Research, Inc.
 -- Motif graphics initialized --

In[1]:=
             3    5      7       9
            x    x      x       x          11
Out[1]= x - -- + --- - ---- + ------ + O[x]
            6    120   5040   362880

In[2]:=
             3    5      7       9
            x    x      x       x
Out[2]= x - -- + --- - ---- + ------
            6    120   5040   362880

In[3]:=
 ~/courses/115 =>
```

In this case, the results of all the commands are outputted and *Mathematica* then exits.

A long *Mathematica* computation can be run in the background. Under linux this is done in the usual way:

```
 ~/courses/115 => math < runmath.m > outmath &
[1] 12097
 ~/courses/115 =>
```

in which the commands from runmath.m are read in and the output goes to the file "outmath".

One can also read in data from files produced by other programs, C or fortran say. Typically one would like to read in columns of data from the file into a (nested) list in *Mathematica* which could then be plotted by `ListPlot`, for example, see Sec. 4.4. This is performed by the command `Import`, which takes two arguments, the first is the name of the file (in double quotes) and the second is the desired *Mathematica* format, in this case `Table`, also in double quotes. Here is an example:

```
In[1]:= !!fortran.dat
   1 1
   2 4
   3 9
   4 16

In[2]:= Import["fortran.dat", "Table"]

Out[2]:= {{1, 1}, {2, 4}, {3, 9}, {4, 16}}
```

The first line, `!!fortran.dat`, displays the contents of the file "fortran.dat", but does not read it in. The second line, with `Import[...]`, reads in the file, creating a separate list (of two elements in this case) for each line of the imported file. The final result is a list of these lists, as required by `ListPlot`.

One could accomplish the same thing more elegantly by *executing* the C or fortran program that produces the data directly from within *Mathematica*, and having the output stored as a *Mathematica* list. In this way one does not have to create a separate data file (fortran.dat in the above example). To accomplish this, replace the name of the data file in the above `Import` command by the name of the executable file preceded by an exclamation point, i.e. if the name of the executable is "gendata" the command is

```
In[3]:= Import["!gendata", "Table"]

Out[3]:= {{1, 1}, {2, 4}, {3, 9}, {4, 16}}
```

This places the output from the program "gendata" as input to *Mathematica*. If, as in our example, the data consists of two columns, then it could be plotted using the `ListPlot` command, see Sec. 4.4, i.e.

```
In[3]:= ListPlot [ %3 ]
```

Hence, *Mathematica* might be a convenient way for you to plot data produced by a C or fortran program.

Within a *Mathematica* session one can also send output from a command to a file by putting `>> filename` at the end of a command, e.g.

```
In[4]:= << runmath.m

             3    5      7       9
            x    x      x       x
Out[4]= x - -- + --- - ---- + ------
            6    120   5040   362880

In[5]:= % >> outmath2
```

produces

```
x - x^3/6 + x^5/120 - x^7/5040 + x^9/362880
```

in file "outmath2". If one then redirects another line of output to the same file the previous contents are overwritten. To *append* to the end of the file (on a new line) use `>>> filename`.

6.2 Saving Work

If you are using the notebook interface you can save the entire notebook to a file (it is customary to give it a name ending in ".nb") and later restart a new *Mathematica* session by reading in this file. With Windows or a Mac interface double click on the icon of the notebook. Under linux, type "mathematica work.nb" if the notebook is called "work.nb". In a notebook one can combine together, text, mathematical calculations and graphics. See the references and on-line help for more details. With hindsight it would probably have been better to produce this document as a notebook rather than in LaTeX, but it is now a lot of work to make the change.

In the command line interface, one can save definitions in a file with the command `Save["fname", def1, def2, ... ]`, see e.g. Gray[5], p.67. As an example, suppose that we have defined `f1, f2` and `f3`. These can be saved into the file "savedefs" by the command `Save["savedefs", f1, f2, f3]`. If one clears these definitions with `Remove[f1, f2, f3]`, they can be read back from the file with the command `<< savedefs` command discussed above. Here is an example,

```
In[1]:= f1 = x; f2 = x^2; f3 = x + f2;

In[2]:= ?f3
Global`f3

f3 = x + x^2

In[3]:= Save["savedefs", f1, f2, f3]

In[4]:= Remove[f1, f2, f3]

In[5]:= ?f3
Global`f3
```

The last line shows that *Mathematica* no longer knows about `f3`. However, if we read in the definitions from the file "savedefs", then the definition is recovered

```
In[6]:= << savedefs

             2
Out[6]= x + x

In[7]:= ?f3
Global`f3

f3 = x + x^2
```

6.3 Setting Options

Mathematica has many options for its commands. The default options can be changed for any future commands in the same session with the command `SetOptions`, see e.g. Tam[2] p. 256. For example, I find that lines, the points and the axes are too small the plots, and I like to have color. To change the defaults for `ListPlot` *within a session* one could execute the command:

```
In[1]:= SetOptions[ListPlot,                    \
    AxesStyle -> {AbsoluteThickness[3]},        \
    Prolog-> AbsolutePointSize[6],              \
    PlotStyle -> {AbsoluteThickness [3], Hue[0]}
```

A related example of the use of the `SetOptions` command is given in Sec. 4.2.

One might also wish to change the default options permanently, so that they will be available automatically in a new *Mathematica* session. Some of the default settings can be changed permanently from within a notebook session by going to the *Preferences...* menu (under linux this is the last item in the *Edit Menu*). For example the default font can be made bigger.

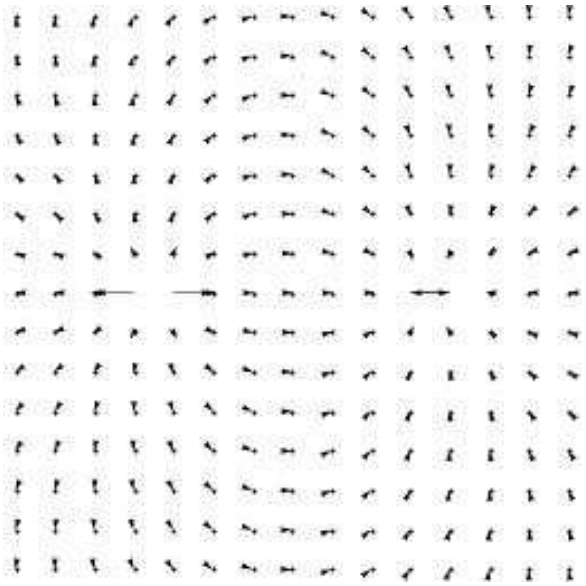
Surprisingly, as far as I can tell it does not seem possible to permanently change default settings for line widths and colors of plots from within *Mathematica*; one has to add the `SetOptions` commands by hand to a file which is read in automatically when *Mathematica* starts up. Under linux this file is "$HOME/.*Mathematica*/Kernel/init.m", where "$HOME" is your home directory.

6.4 Loading Packages

Although there are many functions built into *Mathematica* which are immediately available, standard *Mathematica* distributions come with additional capabilities which are not loaded by default (presumably because this would take too much memory) but which can be loaded upon request. These additional function are contained in what are known as "packages". To load a package the command is `Needs["packagename"]` where "packagename" is the name of the package. (Note that it is better to use `Needs["packagename"]` than `<< packagename`, see e.g. Tam[2] p. 10.)

For example, if we have the expression for the scalar potential due to some charges, and we want a plot of the direction of the electric field (the gradient) we need the command `GradientFieldPlot` which is in the package `VectorFieldPlots‘`. Below I load the package and show the direction of the electric field of a dipole made up of charges at $(1, 0)$ and $(-1, 0)$:

```
In[1]:= Needs["VectorFieldPlots‘"]

In[2]:= GradientFieldPlot[1/Sqrt[(x - 1)^2 + y^2] - 1/Sqrt[(x + 1)^2 +
y^2],{x, -2, 2}, {y, -2, 2}]
```



A problem will arise if you try to use a symbol for which a package is needed before loading the package. *Mathematica* will naturally give an error message. You might think that you could just load the package with the `Needs` command and then execute the command. The problem is that *Mathematica* then "knows" about two "symbols" with the same name, the one you "defined" with your unsuccessful attempt to execute it, and the one in the library package. (You can get a list of symbols that you have defined with the command `?Global‘*`.) The symbol you used is said to be "shadowing" the one in the package, and *Mathematica* will attempt to use yours (rather than the one in the package) in the absence of any instruction to the contrary. Hence, before you can use the symbols in the package, you have to first remove your own invocation of the symbol with the command `Remove[name]` where `name` is the the name of the symbol to be removed. If you need to remove several symbols the command is `Remove[name1, name2, ···]`.

For example, in the previous example, where we we used the symbol `GradientFieldPlot`, if we try to use it before loading the appropriate package

```
In[1]:= GradientFieldPlot[
 1/Sqrt[(x - 1)^2 + y^2] - 1/Sqrt[(x + 1)^2 + y^2], {x, -2,
  2}, {y, -2, 2}]
```

then nothing happens. *Mathematica* doesn't know about the `GradientFieldPlot` command so the ouptut is just a repeat of the command. If we then load in the package `VectorFieldPlots‘`,

```
In[2]:= Needs["VectorFieldPlots`"]

GradientFieldPlot::shdw:
   Symbol GradientFieldPlot appears in multiple contexts
    {VectorFieldPlots`, Global`}; definitions in context
VectorFieldPlots`
     may shadow or be shadowed by other definitions.
```

we get a warning about multiple defintions of `GradientFieldPlot`. Giving the `GradientFieldPlot` command doesn't work because *Mathematica* now has two symbols with the name `GradientFieldPlot`, the one we used and the one in the package. However, if we remove our symbol by the following command,

```
In[3]:= Remove[Global`GradientFieldPlot]
```

then all is well, and repeating the `GradientFieldPlot` command above, does generate the plot. Not removing a previous unsuccessful attempt to use a "symbol" when the symbol requires a package to be loaded, is one of the most common causes of grief among beginning users of *Mathematica*.

References

[1] *The Mathematica Book* by S. Wolfram, Cambridge University Press. A huge volume written by the creator of *Mathematica*.

[2] *A Physicists Guide to Mathematica* by P. T. Tam, Academic Press. A good introduction with some examples from physics.

[3] *A Crash Course in Mathematica* by S. Kaufmann, Birkhäuser. A concise introduction.

[4] *Physics by Computer* by W. Kinzel and G. Reents, Springer. A good source of ideas for numerical problems to work on. Requires a high degree of sophistication and independence from the student.

[5] *Mastering Mathematica* by John W. Gray, Academic Press.

[6] *Mathematica for Physics* by R. L. Zimmermann and F. I. Olness, Addison Wesley.

[7] *Mathematica for Scientists and Engineers* by R. Gass, Prentice Hall.

[8] *Mathematica for Calculus–Based Physics* by M. de Jong, Addison-Wesley.

Index