Chapter 3

Typesetting with LaTeX

3.1 Elements of LaTeX

3.1.1 Preamble

Every LaTeX file has a structure like this:

```
\documentclass[12pt]{article}
\usepackage{2130}
  .........
\begin{document}
  .........
\end{document}
```

The part of the file before the line `\begin{document}` is called *preamble*, while the subsequent part is called *body* of the document.

The first line in the file tells LaTeX to use a file called `article.sty` as the main source of formatting commands. The style file contains certain default parameters that determine layout of the document, in particular:

```
\textwidth
\textheight
\topmargin
\oddsidemargin
\evensidemargin
```

The subsidiary style indicated by the argument "[12pt]" sets up the regular font size for the document to be 12pt. Line spacing and sizes of fonts described in relative terms (Large, large, small, tiny) thus become determined.

The first line may be followed by lines that import additional commands from various *packages*. The package `2130.sty`, which the second line in our document refers to, makes it

© MMIX Department of Mathematics and Statistics, Memorial University of Newfoundland

easy to format a report for this course. For example, it will default to one inch margins all around the standard size of paper. It also makes it easy to create the title page, headers and footers, using the commands

```
\headers{...}{...}{...}                \footers{...}{...}{...}
\underhead        \nounderhead         \overfoot        \nooverfoot
\underheadoverfoot
```

The package `2130.sty` also contains enhanced graphics command, which significantly extend a graphical facility provided by the standard LaTeX, as described in Section 4.4.3.

Other useful packages that you will likely need to include in your documents are `graphicx` (to enable import of EPS graphics, see Section 4.3.1) and `amssymb` (to enable mathematical symbols like $\mathbb{R}$ and various special characters, like $\varnothing$).

Besides `\usepackage` command(s), the preamble may contain your own definitions. For example, this is a convenient shorthand for an useful but long LaTeX's keyword (see p. 28):

```
\newcommand{\ds}{\displaystyle}
```

User-defined commands may have parameters, for example: `\newcommand{\pair}[2]{(#1,#2)}`. Now, the command `\pair{A12}{b34}` produces: (A12,b34).

3.1.2 Comments

The percent sign % is the commentary symbol in LaTeX. Everything that follows it is ignored till the end of line. There are two exceptions. First, the percent sign itself can be printed with command `\%` and, as a part of this command, it does not begin a commentary. Second, the percent sign within the `verbatim` environment or `\verb` command has no controlling effect.

If there is a need to disable rather long parts of a document from processing, while deleting them is undesirable, the following construction can be used:

```
\iffalse
  No matter how many lines or pages - everything here will be ignored by LaTeX
\fi
```

Finally, you may put the line `\end{document}` earlier in your document and LaTeX will stop exactly there. Everything that follows will be ignored. This is a convenient method to find and fix errors in long documents.

3.1.3 Environments

The following construction, very common in LaTeX, is called an *environment*:

```
\begin{something}
    ..........
\end{something}
```

In this case, we deal with a (nonexistent) "something"-environment. The whole document is an environment by itself.

Some common environments are: *displaymath, equation, tabular, picture, table, figure, center, thebibliography, itemize, enumerate, verbatim.*

3.1.4 Space

LaTeX regards one 'return' or 'enter', one or more 'tabs' and one or more blank spaces, as equivalent to one 'space'. An exception to this rule is presented by the spaces within the `\verb` command and the `verbatim` environment.

Paragraphs

There are two ways to start a new paragraph:

1. The usual (simplest) way is to leave a blank line in your text.
2. Use the command `\par`.

Line breaks

You may force LaTeX to start a new line within the current paragraph by means of the command \\ (double backslash). The current line ends immediately, at the same position as if the paragraph had ended. The command \\ is routinely used as a separator between lines in tables and "arrays" (multi-line formulas in math mode).

Another seemingly similar command is `\linebreak`. However it is rarely used. It causes LaTeX to stretch the current line to page width to compensate for the text that has been forced out to the next line. LaTeX may find the required stretch intolerable and deny the requested line break.

Vertical space

LaTeX does a reasonable job of inserting space and turning up new pages. Still, there are times when the user would like to assert his/her own influence.

The regular, preferred method to add vertical space between paragraphs or figures is by using one of the three commands:

`\smallskip` adds a little extra space to the regular space between lines

`\medskip` adds about twice that much

`\bigskip` adds yet about twice that much as `\medskip`.

The command `\vspace{1ex}` instructs LaTeX to push the next line down by the height of a letter "x" in the current font size. These commands ought to be given after a blank line (in particular, they should not be given within a paragraph), otherwise the layout of your page may turn out to be rather odd. The command `\vskip 1ex` has the same effect in most cases; however, its use in LaTeX documents is now deprecated.

There are several allowed measures of length, including inches (in), centimeters (cm) and millimeters (mm), and finally, points (pt), which are the preferred unit to a typesetter. (There are 72 points to an inch.) Besides these *absolute* units, the units relative to the current font size are often used: `\ex` (see above) and `\em` (see below).

The command `\vspace{...}` has no effect at the top of a page or at the bottom. Why would you want space when you are about to move to a new page? If you insist, you must use `\vspace*{...}` to force LaTeX to make space.

In the line break command inside a paragraph of text, as well as inside tables, arrays (in math mode), and parboxes, one can create space by affixing an argument to the command `\\`; for example, `\\[1ex]` leaves 1ex of additional space after the current line.

To make LaTeX to skip more space between paragraphs, as is done throughout this Manual, you may add the following command in the preamble of your document:

```
\setlength{\parskip}{\smallskipamount}
```

Finally, the commands `\pagebreak` and `\newpage` end the current page and start putting text on the new one.

Horizontal space

The command `\hspace{10pt}` skips 10 points (approx. 3.5 mm) of horizontal space. (The command `\hskip 10pt` has the same effect in most cases, but its use is discouraged.)

At the beginning of a line `\hspace{..}` will have no effect, so you must use the command `\hspace*{..}` instead.

It is recommended that you use the units 'em' (the width of a letter 'm' in the current font size) for horizontal space.

By default, paragraphs in LaTeX begin with indentation, except those immediately after the headings made with commands like `\section`. To suppress indentation, you can use the command `\noindent`.

Math space

Spacing commands like `\hspace` and `\vspace` only work in "paragraph mode" (outside math). The horizontal spacing commands in math mode are: `\; \: \, \! \quad \qquad`, which can be found in Maltby.

That's how much they measure: `\,`= || and `\:`= || and `\;`= || and `\quad`= | | and `\qquad` = | |. The remaining distance `\!`= ‖ is a tiny step *backward*!

Vertical spacing in math mode is best handled with `\\[..]`.

Centering

To center a block of text, use the *center* environment:

```
\begin{center}
 ....
\end{center}
```

Note the American spelling of "center". LaTeX is **very unforgiving** if you make a spelling error in a command.

3.1.5 Math mode

Inline math and displayed math

LaTeX can typeset in a paragraph mode or in math mode, which, in its turn can be of two kinds: *inline math* or *displayed math*. To begin and end the inline math mode, use the dollar signs: `$...$`. This places the mathematical formula as the next word in the line of text, like here: $(x+y)(x-y) = x^2 - y^2$. However, if you wish to display the formula, use

```
\begin{displaymath}
  ............
\end{displaymath}
```

A shorter delimiter for displayed math (both to begin and to end) is double dollar sign: `$$....$$`.
The displayed formula is automatically centered on its own line. (This can be artificially changed, but rarely needed.)

A displayed formula is still considered to be a part of a paragraph unless there is a blank line separating it from the paragraph, before or after. In particular, the text following the displayed math (without blank line in between) will not be indented.

A modification of the `displaymath` environment is the `equation` environment, which displays a single **numbered equation**, like this:

$$2 + 2 = 4. \tag{1}$$

LaTeX automatically numbers equations enclosed in the `equation` environment consecutively starting from (1). It does not number inline equations and equations within the `displaymath` environment, but it does number equations within the `eqnarray` environment described below.

There is a difference in style of math formulas depending whether they are printed in the inline or displayed math mode. In the displayed mode, for instance, fractions' numerators and denominators are printed in the regular font size, while in the inline mode they are printed in a reduced font size. There is a way to force LaTeX to type math in a prescribed mode: this is done by the commands `\displaystyle` and `\textstyle`. There is also a similar command `\scriptstyle` (to print in small size, like subscripts or superscripts).

3.1.6 Lists

Text (paragraph mode)

Use the 'enumerate' environment or the 'itemize' environment

```
\begin{enumerate}
  \item ......
  \item ......
     .........
  \item ......
\end{enumerate}
```

```
\begin{itemize}
 \item ......
 \item ......
    .........
 \item ......
\end{itemize}
```

to generate a numbered /bulletted list of items, respectively.

Math mode — `\eqnarray`

Here, we usually mean a set of equations. Use

```
\begin{eqnarray*}
<formula> & <connective> & <formula> \\
...
...
...
<formula> & <connective> & <formula> \\
\end{eqnarray*}
```

It is possible that some of the '`< ... >`' are empty fields. Also, the `\\` can be replaced by `\\[..]` as indicated in the section on vertical space above.

If the `*` is omitted, the equations will automatically be numbered. If a line is not to be numbered, then the command `\nonumber` must be entered somewhere in that line.

Math — `\array`

The `\array` environment is used within displayed math in the cases when an additional flexibility as compared to `\eqnarray` is required. Details can be found in Maltby.

In both, `\array` and `\eqnarray` environments, LaTeX is by default in the **inline** math mode, so you have to use the `\displaystyle` command to display fractions, sums, etc. properly. Moreover, this command only has effect between two ampersands or between an ampersand and the endline command `\\`. So you may have to issue the `\displaystyle` command repeatedly. That's why the abbreviation shown on p. 25 makes a lot of sense.

3.1.7 Advanced math typesetting

The formula

$$\frac{x^{\sin(x)}}{(\cos(x))^x} \;=\; \int_{\sqrt{x}}^{\infty} f(x)\,dx\,.$$

is obtained from

```
$$
 \frac{x^{\sin(x)}}{(\cos(x))^x} \;=\; \int_{\sqrt{x}}^\infty\,f(x)\,dx\,.
$$
```

To the mathematically literate, many math commands are intuitive. If you want a symbol and cannot remember it, try the obvious, and most times you will be correct!

You now should read Section 3.3, *An Introduction to TeX and friends* by Gavin Maltby. The definitive book is the "LaTeX User's Guide and Reference Manual", by Leslie Lamport.

3.1.8 Processing and viewing LaTeX files

Every LaTeX file must be saved with the `.tex` extension.[1] You process the file "mylab.tex" with the command

```
latex mylab
```

Note that there is no need for the extension `.tex`.

At this point, a lot of information will come across your screen. With time, much of it will even make some sense. Everything you see on the screen, and more, gets written to a "log" file. If you processed "mylab.tex", LaTeX will create "mylab.log" for you. If you are lucky and have made no errors, LaTeX will eventually stop and report that the output was written to "mylab.dvi". If you are less fortunate, LaTeX will stop at the first error and leave you hanging at a question mark on the screen. At this point, if you answer `r` (for run), LaTeX will finish processing to the best of its ability, writing all errors to "mylab.log" which one can then review in one window while correcting "mylab.tex" in another.

To view "mylab.dvi", you use a UNIX program called `xdvi` which is run like this:

```
xdvi mylab
```

Again, there is no need for the extension `.dvi`.

A `.dvi` can be converted to a `.pdf` file by the command like this:

```
dvipdf mylab
```

[1] The description below has been in this Manual since its first edition and is still valid if you are working in the UNIX command line. Many integrated environments like Kile, WinEdt, Texnic Center or Scientific Word nowadays make processing a LaTeX document more comfortable.

3.1.9 Including source code in LaTeX documents

There are a couple of ways in which you can include the source code from your programs in your LaTeX documents. One way is to use the `verbatim` environment:

```
\begin{verbatim}
    Paste a copy of your code here
\end{verbatim}
```

Remember to **break long lines by hand** so that they fit the page width; otherwise some details of your program will not be visible.

The other way to do so is to use `lgrind`, which formats program sources in a nice style. Comments are placed in Roman font, keywords in bold face, variables in italics, and strings in typewriter font. Source file line numbers appear in the right margin every 10 lines. Suppose that you have a C program in the file `sample.c`. The first step towards including the file in the document is to run it through `lgrind` to produce a file, say `sample.tex`:

```
lgrind -i -lc sample.c > sample.tex
```

This generates a file `sample.tex`, which has the pretty-printed version of `sample.c` with a lot of LaTeX commands. (Note that your program and main document should have different names!)

Now, in the declarations at the start of your main LaTeX document file, you have to be sure to include the `lgrind` package:

```
\usepackage{lgrind}
```

At the point in the document where you want to include the source code, give the command:

```
\lgrindfile{sample.tex}
```

Figure 3.1(A) presents an example of how the file would show in your document.

An **alternative method** of including source code, which does not require any intermediate processing, would be to declare a few commands near the start of your LaTeX document:

```
\newcommand{\beginverb}{\begin{verbatim}}
\newcommand{\inputfile}[1]{\input{#1}}
\newcommand{\verbatimfile}[1]{\expandafter\beginverb\inputfile{#1}}
```

and later on make use of the `verbatimfile` command:

```
{ \small
\verbatimfile{sample.c}
\end{verbatim} }
```

(Don't forget to close the brace after `\end{verbatim}`; otherwise the rest of your document will be printed in a small font size.)

The program listing printed with this method is displayed on Figure 3.1(B).

A.

```
/* This is a short C program to compute the sum of the integers
 * from 1 to 1000 and print the result.
 */
#include <stdio.h>
#define N 1000

int Sum (int maxnum);

int main ()
{
   printf ("The sum of the integers from 1 to %d is %d\n", N, Sum(N) );
   return (0);
}

int Sum (int maxnum)
{
   int i, total=0;
   for (i=1 ; i<=maxnum ; i++)
      total += i;
   return (total);
}
```

B.

```
/* This is a short C program to compute the sum of the integers
 * from 1 to 1000 and print the result.
 */
#include <stdio.h>
#define N 1000

int Sum (int maxnum);

int main ()
{
   printf ("The sum of the integers from 1 to %d is %d\n", N, Sum(N) );
   return (0);
}

int Sum (int maxnum)
{
   int i;
   int total=0;
   for (i=1 ; i<= maxnum ; i++)
      total += i;
   return (total);
}
```

Figure 3.1: Source code printed (A) using `\lgrind` and (B) using `\verbatimfile`

3.1.10 Some commands defined in `2130.sty`

Text commands

`\TODAY` produces dates in the form 22 December 2008.

`\TODAYAT` produces dates and time in the form 22 December 2008 at 14:57.

`\Cents` produces ¢.

`\INDENT` forces an indented line when `\indent` fails.

`\tildechar` produces ~

`\hatchar` produces ^

`\boxit{object}` produces

$$\boxed{object}$$

`\lbk`, short for `\linebreak`, produces a line break with horizontal justification.

`\pbk`, short for `\pagebreak`, produces a page break with vertical justification.

`\fsize{number}` is an alternative way of changing font size. This was recommended by Lamport. `\normalsize` is `\fsize{0}`. Positive integers increase and negative integers decrease from here.

For example `\fsize{1}` is equivalent to `\large`, and `\fsize{-4}` is equivalent to `\tiny`. Integers too large default to the smallest or largest fonts available.

Math commands

`\di` is short for `\displaystyle`

`\toi` produces $\rightarrow \infty$

`\dist` produces dist

`\slope` produces slope

`\LOngleftrightarrow` produces $\Longleftrightarrow$ — compare
`\Longleftrightarrow`, which produces $\Longleftrightarrow$.

`\LOngleftarrow` produces $\Longleftarrow$

`\LOngrightarrow` produces $\Longrightarrow$

`\IFF` produces $\Longleftrightarrow$ — compare `\iff`, which produces $\iff$

3.2 Formatting your Math-2130 report in LaTeX

3.2.1 Title page, footers and headers

A typical title page is shown on Figure 3.2. A LaTeX code used to produce it is as follows.

```
\begin{titlepage}
\begin{center}
\large SHAPE MANIPULATION \\
   AND MATRIX ALGEBRA
\end{center}
\vspace{6cm}

\hfill\begin{tabular}{ll}
    AM 2130 & Lab 2 \\
    Submitted by: & Unlikely Student \\
                  & \#200765432 \\
    Submitted to: & Dr.~Good Professor \\
    Feb. 18, 2009
\end{tabular}
\end{titlepage}
```

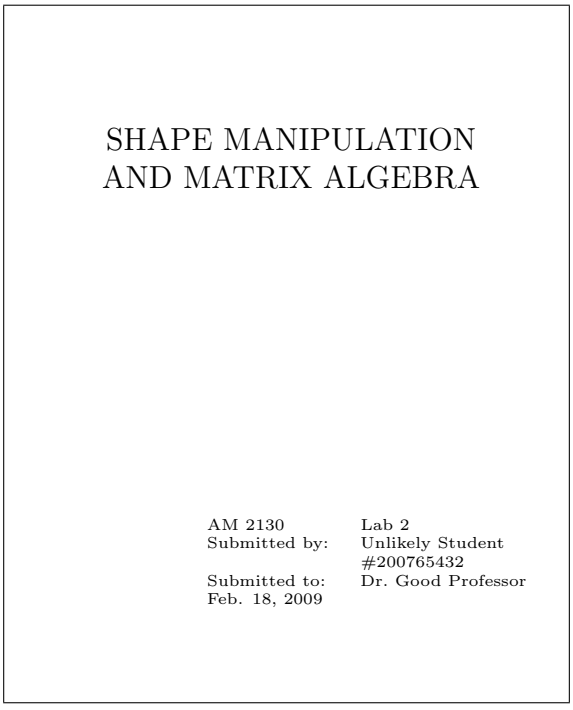

Figure 3.2: A sample title page for an AM2130 report.

Your document will look nicer with **running footers and headers**. The following few lines can be placed in the preamble or after the title page. The effect is explained in the comments.

```
\lhead{AM2130}        % appears in the header on the left
\rhead{Lab 1: Title}  % appears in the header on the right
\lfoot{Your Name}     % appears in the footer on the left
\rfoot{\thepage}      % page number, in the footer on the right
\underheadoverfoot    % dividing lines: under header and below footer
```

3.2.2 Table of contents

Assuming you use the standard LaTeX commands to structure your report ("section, "subsection), all that remains to produce the table of contents (TOC) is to insert the command

```
\tableofcontents
```

after the title page code.

You have to run LaTeX compiler **two times** to obtain a correct TOC, as the first run just creates an auxiliary file where the information about sections and subsections found is collected, but LaTeX is not able to incorporate that information into its dvi output immediately.

Some complication occurs with References and Appendix (assuming they are formatted as suggested below). The command `\thebibliography`, which generates the list of references, does not automatically yield an entry in TOC. You have to do some work by hand.

Somewhere soon after your `\thebibliography` command (perhaps, immediately after) insert the following line:

```
\addcontentsline{toc}{section}{References}
```

The heading **References** will then appear in your TOC and it will be printed in the same style as section headers. If you want to somewhat de-emphasize References in TOC, modify the above line:

```
\addcontentsline{toc}{subsection}{References}
```

Similarly, if you formatted your Appendix A using `\section*` command, you should put the corresponding TOC line immediately after it, for example:

```
\addcontentsline{toc}{section}{Appendix A: Program source code}
```

3.2.3 Abstract

> **Abstract.** The `quote` environment provides a nice format for an abstract. Type the word *Abstract* in boldface and the rest in a regular font style. Abstract should not be included in the TOC.

3.2.4 The body of report

Use the `\section{...}` command for section headers, like Introduction, Technical Details, etc. Use the `\subsection{...}` command for parts of the main sections.

Having third-tier headers (subsubsections) can hardly be justified in a typical Math-2130 report.

Acknowledgements and Appendix should not be numbered as regular sections. Use the `\section*` command, for example,

```
section*{Acknowledgments}
```

The so formatted sections are not automatically referenced in the TOC. You should include a reference to Appendix or Appendices, but probably not to Acknowledgements.

3.2.5 References

The list of references is printed using `thebibliography` environment. For example,

```
\begin{thebibliography}{4}
\bibitem{hagin} Frank G.~Hagin, A first course in differntial equations.
               Prentice-Hall, Inc.:New Jersey. 1975.
%
\bibitem{m2130} Math-2130 Course Manual. MUN, 2008.
%
\bibitem{fourier-bio} ``Jean Baptiste Joseph Fourier'' (biography).\\
\verb+http://www-history.mcs.st-andrews.ac.uk/history/Mathematicians/+
\verb+Fourier.html+ \\ (Accessed December 2, 2008.)
%
\end{thebibliography}
```

Don't forget to provide a numerical argument in `\thebibliography` line. It does not necessarily be equal to the actual number of references, it must just have the same decimal length — like in our example, 3 references and the number is 4, both are single-digit numbers. In the absence of the numerical argument LaTeX often gives a misleading information as to where the error is.

3.2.6 Appendix and program source

Use the `\section*` command for appendices. Section 3.1.9 explains how to include the source code of your program in the report. (It is not always necessary.)

A little note for those who needs to include a **Maple code** in the report: in our experience, the quickest way that does not compromise on typographical quality is simply to copy and paste the lines from Maple worksheet into the LaTeX file line by line. Maple programs (at least those in Math-2130 projects) are usually rather compact. Maple's numerical outputs can also be

copied and pasted, unless they contain exponents. In the latter case please take a trouble to type the answer properly in the LaTeX math mode, for example, 6.67×10^{-11}. If there are too many such numbers to type, think about summarizing them in a table rather than just copying.

Maple's symbolic answers, when copied into a text file, are difficult to read. Again, if the answer is important (so you are not just filling up a space), it is worth to neatly re-type in a proper mathematical format. *Important* results can hardly be too many...

A copy of your Maple worksheet submitted electronically is a definitive document and will be used by the instructor should any discrepancy between your reported input and Maple's output be revealed.

3.2.7 Floating environments: figures and tables

LaTeX has its own ideas as to where it is best to place your figures and tables made by means of the corresponding environments. In the processed document, it usually does not put them where they ought to be according to their position in the source `.tex` file. Many students, and not only, often find this frustrating.

However, LaTeX's behaviour is in most cases backed by well established rules and conventions of typography. In general, try not to fight LaTeX very hard. If it puts your figure on top of the next page rather than dropping it just on spot, think whether you can agree with that decision. Only if the displacement of the figure leads to a mess in the logic of your presentation or if a large unfilled area remains on the page, you should insist.

The way to make LaTeX to put the picture **exactly here** is to use the `figure` environment with argument `[H]`:

```
\begin{figure}[H]
```

The option `H` is not provided by standard LaTeX, you must `\usepackage{float}` in order to be able to use it. All this equally applies to the `table` environment.

3.2.8 Automatic numbering, cross-references, and citations

The `\caption` command within a `figure` or `table` environment assigns a number to the figure or table. This number can be captured by the `\label` command put right after. For example, the command `\label{xyz}` allows you to refer to your figure elsewhere in your document in the following way:

"`See Figure~\ref{xyz}` on `page~\pageref{xyz}`."

Similarly, labels can be assigned to sections, subsections (Sect. 3.2.4), equations (Sect. 3.1.5) and theorem-like environments described in Section 3.4.6.

References to sources listed under `\thebibliography` command can be done as follows:

```
Fourier's memoir ``On the Propagation of Heat in Solid Bodies'' was
read to the Paris Institute on 21 December 1807 \cite{fourier-bio}.
```

3.3 An introduction to TeX and friends

Original text by Gavin Maltby (1992); adapted by Math-2130 Instructors (1995,1998,2008)

3.3.1 An Introduction to TeX

TeX is well known to be *the* typesetting package, and a vast cult of TeX lovers has evolved. But to the beginning TeX user, or to someone wondering if they should bother changing to TeX, it is often not clear what all the fuss is about. After all, are not both WordPerfect and Microsoft Word capable of high quality output? Newcomers have often already seen what TeX is capable of (many books, journals, letters are now prepared with TeX) and so expect to find a tremendously powerful and friendly package. In fact they *do*, but that fact is well hidden in one's initial TeX experiences. In this section we describe a little of what makes TeX great, and why other packages cannot even begin to compete. Be warned that a little patience is required—TeX's virtues are rather subtle to begin with. But when the penny drops, you will wonder how you ever put up with anything different.

TeX is a typesetter, not a word-processor

TeX was designed with no limiting application in mind. It was intended to be able to prepare practically any document—from a single page all-text letter to a full blown book with huge numbers of formulae, tables, figures etc.

Conventional word processors have a fundamental limitation in that they try to "keep up" with you and "typeset" your document as you type. This means that they can only make decisions at a local level (eg, it decides where to break a line just as you type the end of the line). TeX's secret is that it waits until you have typed the *whole* document before it typesets a single thing! This means that TeX can make decisions of a global nature in order to optimise the aesthetic appeal of your document. It has been taught what looks good and what looks bad (having been given a measure of the "badness" of various possibilities) and makes choices for your document that are designed to make it "minimally bad".

But TeX's virtues run much deeper than that, which is just as well because it is possible to get satisfactory, though imperfect, results from some word processors. One of TeX's strongest points is its ability to typeset complicated formulae with ease. Not only does TeX make hundreds of special symbols easily accessible, it will lay them out for you in your formulae. It has been taught all the spacing, size, font, ... conventions that printers have decided look best in typeset formulae. Although, of course, it does not understand any mathematics it knows the grammar of mathematics—it recognises binary relations, binary operators, unary operators, etc. and has been taught how these parts should be set. It is consequently rather difficult to get an equation to look bad in TeX.

Another advantage of compiling a document after it is typed is that cross-referencing can be done. You can label and refer back to chapters, sections, tables etc. by *name* rather than

absolute number, and TEX will number and cross-reference these for you. Similarly, it will compile a table of contents, glossary, index and bibliography for you.

Essential to the spirit of TEX is that *it formats the document whilst you just take care of the content*, making for increased productivity. The cross-referencing just mentioned is just part of this. Many more labour-saving mechanisms are provided for through *class* and *style files*. These are generic descriptions of classes of documents, teaching TEX just how each class likes to be formatted. This is taught in terms of font preferences, default page sizes, placement of title, author, date, etc. For instance, a `paper` style file could teach TEX that when typesetting a theorem it should embolden the part that states the theorem number and typeset the text of the theorem statement in slanted Roman typeface (as in many journals). The typist simply provides and indication that a theorem is being stated, and then types the text of the theorem *without* bothering to choose any fonts or do any formatting—all that is done by the style file. Style files exist for all manner of document—letters, articles, papers, books, proceedings, review articles, and so on.

There are many other motivations one could cite for the superiority of TEX. But it is time that we started to get our hands dirty. The novice reader will still have no idea of what a TEX source file looks like. Indeed, why do we keep referring to it as a *source file*? The fact of the matter is that TEX is essentially a *programming language*. Just as in any compiled language (e.g., Fortran, C) one prepares a source file and submits it to the compiler which attempts to produce an object file (`.dvi` file in the TEX case). To learn TEX is to learn the command syntax of the commands that can be used in the source file.

Typical TEX interfaces

By the nature of TEX most time is spent editing the source document (before submitting it for compilation). No special interface is necessary here, you just use your favourite text editor (perhaps customising it to enhance TEXnical typing). Thus TEX user interfaces are usually small and simple, often even missing. One frequently uses TEX at command line level, just running the editor, compiler etc. as you need them. Sometimes a TEXshell program is present, which runs these for you when you choose various menu options.

Whatever the interface, there are just a few basic steps to preparing a document:

1. Choose a document style to base your document on (e.g., letter, article).
2. Glance through the material you have to type, and decide what definitions might be made to save you a lot of time. Also, decide on the overall structure of the prospective document (e.g., will the largest sectional unit be a chapter or a part?). If you are going to compose as you type, then pause a moment to think ahead and plan the structure of your document. The importance of this step cannot be overstressed, for it makes clear in *your* mind what you want from TEX.
3. Prepare your input file, specifying only the content and the logical structure (parts, sections, theorems,...) thereof and forgetting about formatting details.
4. Submit your input, or source, file to the TEX compiler for compilation of a `.dvi` file.

5. If the compiler finds anything in your source file strongly objectionable, say incorrect command syntax, then return to editing.

6. Run a *previewer* to preview your compiled document on the screen.

7. Go back to editing your document until glaring errors have been taken care of.

8. Make a printout of your compiled document, and check for those errors that you failed to notice on the screen.

Performing these steps may be effected through typing at the system prompt (bare-bones technique) or through choosing menu options in a TeXshell program. The latter will probably provide some conveniences to make your life easier.

3.3.2 A review of LaTeX

The TeX typesetting system was designed by the eminent Stanford computer scientist Donald Knuth, on commission from the American Mathematical Society. It was designed with enormous care, to be ultimately powerful and maximally flexible. The enormous success of Knuth's design is apparent from the vast number of diverse applications TeX has found. In reading the following you must keep one thing clearly in mind: *there is only TeX language, and all the other packages whose names end in the suffix -TeX simply harness its power via a whole lot of complicated macro definitions.*

TeX proper is a collection of around 300 so called *primitive* typesetting commands. These work at the very lowest level, affording enormous power. But to make this raw power manageable, some macros must be defined to tame raw TeX somewhat.

In the few years after the initial TeX release (1982), the macro package LaTeX was born. LaTeX was written for general usage. LaTeX scores high points for its enhanced command syntax. By far the majority of LaTeX users will never have to learn "raw" TeX, for they will be shielded from direct exposure by the numerous powerful macro packages.

Pre-amble

Every LaTeX document begins with a *pre-amble*. This consists of a set of commands that tells TeX how to process the document. We will explain the important parts of this in the next two sections. The first (*documentclass*) is mandatory, whereas the others are all optional.

Document classes

We have explained the concept of a document class. It remains to name a few, and indicate where they would be used. One *always* has to choose a document class when preparing a document with LaTeX.

The basic document classes in LaTeX are `article`, `letter`, `report`, and `book`. Many more are available, but these few cover the majority of straightforward applications. This is because classes are not rigid—you can impose your own parameter choices if you want. So one chooses

the class that most closely approximates the document you have in mind, and performs some minor tweaks here and there. The `article` class is used for documents that are to have the appearance of a journal or magazine article, and is the class that should be used for your reports in this course. The `report` class is usually used for larger documents than the `article` class. These classes really only differ in their choice of default page size, font, placement of title and author, sectional units, etc. and on how they format certain LaTeX constructs. You use the same LaTeX commands in each. Since the examples here will be small, we will choose to use the `article` document class.

There are a number of possible options with each document class. The syntax for choosing a document class follows. Do not worry if this leaves you with no idea of how to choose a document style, for we will soon be seeing some examples. Also, remember that an argument in square brackets is optional, and can omitted altogether (including the brackets).

`\documentclass [`*`options`*`]{`*`class`*`}`

where *class* is the main document class (eg. `report`) and the optional argument *options* is a list of document style options chosen from, for example, the following list:

`11pt`	chooses 11-point as the default font size for the document, instead of the default 10-point.
`12pt`	chooses 12-point as the default font size.
`twoside`	formats output as left and right pages, as in a book.
`twocolumn`	produces two-column magazine like output.
`titlepage`	applies to the `article` style only, causing the title and abstract to appear on a page each.

In fact there are many, many more document class options but we will not mention any more here.

Packages

To modify further the main document class, we make use of style files. They are used as in the following example: `\usepackage [`*`options`*`]{`*`package`*`}` where *package* is the modifying style file and *options* is a list of style files modifying the style file.

For example, here is a possible preamble:

```
\documentclass{article}
\usepackage{amsmath}
\usepackage{2130}
```

3.3.3 Special symbols

In the coming sections we will see that the the ten characters

`# $ % & ~ _ ^ \ { }`

are reserved for special use.

But what if we need one of these special symbols to appear in our document? The answer for seven of the symbols is to precede them by a \ character, so forming another *control symbol* (remember that \ followed by a space was also a control symbol) .

```
It is not 100\% straightforward to typeset the
characters \$ \& \% \_ \{ \}, but given the
enormous convenience of the use they are normally
reserved for this is a small price to pay.
```

produces

It is not 100% straightforward to typeset the characters $ & % _ { }, but given the enormous convenience of the use they are normally reserved for this is a small price to pay.

So what are control symbols and words?

In typing a document, we can think of ourselves as being in one of two distinct modes. We are either typing *literal text* (which will just be set into neat paragraphs for us) or we are typing text that will be *interpreted* by LaTeX as an instruction to insert a special symbol or to perform some action. Thus we are either typing material that will go straight into the document (with some beautification), or we are giving commands to LaTeX.

Some commands are implicit, in that we do not have to do anything much extra. For instance, we command LaTeX to end the present sentence by typing a period (that does not follow a capital letter). These are not so much commands as part of having to describe the logical structure of a document.

A *control word* is something of the form `\commandname`, where the command name is a word made up only of the letters a to z and A to Z. A *control symbol* consists of a \ followed by single symbol that is not a letter.

Here are some examples:

- we have met the control space symbol \␣ before,
- the commands to set symbols like % and $ are control symbols
- `\@` is a control symbol that told LaTeX that the very next period did really end the sentence,
- `\LaTeX` is a control word that tell LaTeX to insert its own name at the current point,
- `\pm` instructs that a ± be inserted,
- `\div` inserts a ÷ symbol,
- `\infty` inserts a ∞ symbol,
- `\em` makes the ensuing text *be emphasized*.

These examples show that control sequences can be used to access symbols not available from the keyboard, do some typesetting tricks like setting the word LaTeX the way it does, and change the appearance of whole chunks of text as with `\em`. We will be meeting many more of these type of control sequences.

Incidentally, underlining was the traditional way in which writers, working only with typewriters, were able to provide *emphasis*. Nowadays, underlining is poor form because it so easy to italize. With TeX, use `{\em something pretty long}` or `\emph{a word or two}` to produce *something pretty long* or *a word or two*.

Another enormously powerful class of control sequences is those that accept *arguments*. They tell LaTeX to take the parts of text you supply and do something with them—like make a fraction by setting the first argument over the second and drawing a line of the appropriate length between them. These are part of what makes LaTeX so powerful, and here are some examples.

- `\overline{words}` causes $\overline{\text{words}}$ to be overlined,
- `\frac{a+b}{c+d}` sets the given two argument as a fraction, doing most of the dirty work for us: $\frac{a+b}{c+d}$,
- `\sqrt[5]{a+b}` typesets the fifth-root of $a + b$, like this: $\sqrt[5]{a+b}$. The `5` is in square brackets instead of braces because it is an optional argument and could be ommited all together (giving the default of square root),

Mandatory arguments are given enclosed by braces, and optional arguments enclosed by square brackets. Each command knows how many arguments to expect, so you do not have to provide any indication of that.

We have actually jumped the gun a little. The above examples include examples of *mathematical* typesetting, and we have not yet seen how to tell LaTeX that it is typesetting math as opposed to some other random string of symbols that it does not understand either. We will come to mathematical typesetting in good time.

We need to dwell on a TeXnicality for a moment. How does LaTeX know where the name of a control sequence ends? Will it accept both `\pm3` and `\pm 3` in order to set ± 3, and will `\emWalrus` and `\em Walrus` both be acceptable in order to get *Walrus*? The answer is easy when you remember that a control word consists only of alphabetic characters, and a control symbol consists of exactly one nonalphabetic character.

So to determine which control sequence you typed, LaTeX does the following:

1. when a \ character is encountered, LaTeX knows that either a control symbol or a control word will follow.
2. If the \ is followed by a nonalphabetic character, then LaTeX knows that it is a control *symbol* that you have typed. It then recognises which one it was, typesets it, and goes on to read the character which follows the symbol you typed.

3. If the \ is followed by an alphabetic character, then LaTeX knows that it is a control word that you have typed. But it has to work out where the name of the control word ends and where the ensuing text takes over again. Since only alphabetic characters are allowed, LaTeX reads everything up to just before that first nonalphabetic character as the control sequence name. Since it is common to delimit the end of a control word by a space, LaTeX will *ignore* any space that follows a control word, since you want that space treated as end-of-control-word space rather than interword space.

This has one important consequence: The character in the input file immediately after a control symbol will be "seen" by LaTeX, but *any space following a control word will be discarded and never processed.* This does not affect one much if you adopt the convention of always typing a space after a control sequence name.

There is a rare circumstance in which this necessitates a little extra work and thought, which we illustrate by example:

```
If we type a control word like \LaTeX in the running text
then we must be cautious, because the string of spaces that
come after it will be discarded by the \LaTeX\ system.
```

which produces the output

> If we type a control word like LaTeXin the running text then we must be cautious, because the string of spaces that come after it will be discarded by the LaTeX system.

Accents

LaTeX provides accents for just about all occasions. They are accessed through a variety of control symbols and single-letter control worlds which accept a single argument—the letter to be accented. These control sequences are detailed in table 3.1.

`\‘{o}`	ò	(grave accent)	`\u{o}`	ŏ	(breve accent)
`\’{o}`	ó	(acute accent)	`\v{o}`	ǒ	(háček or "check")
`\^{o}`	ô	(circumflex or "hat")	`\H{o}`	ő	(long Hungarian umlaut)
`\"{o}`	ö	(umlaut or dieresis)	`\t{oo}`	o͡o	(tie-after accent)
`\~{o}`	õ	(tilde or "squiggle")	`\c{o}`	o̧	(cedilla accent)
`\={o}`	ō	(macron or "bar")	`\d{o}`	ọ	(dot-under accent)
`\.{o}`	ȯ	(dot accent)	`\b{o}`	o̲	(bar-under accent)

Table 3.1: Control sequences for accents

Thus we can produce ó by typing `\’{o}`, ă by typing `\v{a}`, and Pál Erdös by typing `P\’{a}l Erd\"{o}s`. Take special care when accenting an i or a j, for they should lose their dots when accented. Use the control words `\i` and `\j` to produce dotless versions of these letters. Thus the best way to type to type ĕxĭgent is `\u{e}x\u{\i}gent`.

3.3.4 Formatting

Font commands

We have seen a little of how to access various symbols using control sequences and we mentioned the `\em` command to emphasize text, but we did not see how to use them. We look here at commands that change the appearance of the text.

Each of the control words here is a directive rather than a control sequence that accepts an argument. This is because potential arguments consisting of text that wants to be emboldened or emphasized are very large, and it would be a nuisance to have to enclose such an argument in argument-enclosing braces.

To delimit the area of text over which one of these commands has effect (its *scope*) we make that text into what is called a *group*. Groups are used extensively in LaTeX to keep effects local to an area, rather than affecting the whole document. Apart from enhancing usability, this also in a sense protects distinct parts of a document from each other.

The LaTeX commands for changing type style are given in table 3.2, and those for changing type size are given in table 3.3. Commands for selecting fonts other than these are not discussed here.

`\rm`	Roman	`\it`	*italic*	`\sc`	Capitals
`\em`	*Emphasized*	`\sl`	*slanted*	`\tt`	`typewriter`
`\bf`	**boldface**	`\sf`	sans serif		

Table 3.2: Commands for selecting type styles

Each of the type style selection commands selects the specified style but does not change the size of font being used. The default type style is roman (you are reading a roman style font now). To change type size you issue one of the type size changing commands in table 3.3, which will select the indicated size in the currently active style.

size	default (10pt)	11pt option	12pt option
`\tiny`	5pt	6pt	6pt
`\scriptsize`	7pt	8pt	8pt
`\footnotesize`	8pt	9pt	10pt
`\small`	9pt	10pt	11pt
`\normalsize`	10pt	11pt	12pt
`\large`	12pt	12pt	14pt
`\Large`	14pt	14pt	17pt
`\LARGE`	17pt	17pt	20pt
`\huge`	20pt	20pt	25pt
`\Huge`	25pt	25pt	25pt

Table 3.3: LaTeX size-changing commands.

The point-size option referred to in table 3.3 is that specified in the `\documentclass` command issued at the beginning of the input file. Through it you select that base (or default) font for your document to be 10, 11, or 12 point Roman. If no options are specified, the default is 10-point Roman. The table shows, for instance, that if I issue a `\large` in this document for which I chose the `12pt` document class option the result will be a 14-point Roman typeface.

We mentioned that to restrict the scope of a type-changing command we will set the text to be affected off in a group. Let us look at an example of this.

```
When we want to {\em emphasize\/} some text we
use the {\tt em} command, and use grouping to
restrict the scope.  We can change font {\large sizes}
in much the same way.  We can also obtain {\it italicized},
{\bf emboldened}, {\sc Capitals} and {\sf sans serif} styles.
```

> When we want to *emphasize* some text we use the `em` command, and use grouping to restrict the scope. We can change font sizes in much the same way. We can also obtain *italicized*, **emboldened**, Capitals and sans serif styles.

Notice how clever grouping allows us to do all that without once having to use `\rm` or `\normalsize`.

One more thing slipped into that example—an italic correction `\/`. This is a very small amount of additional space that we asked to be inserted to allow for the change from sloping *emphasized* text to upright text, because the interword space has been made to look less substantial from the terminal sloping character. One has to keep an eye open for circumstances where this is necessary. See the effect of omitting an italic correction after the emphasized text earlier in this paragraph.

One might expect, by now, that LaTeX would insert an italic correction for us. But there are enough occasions when it is not wanted, and there is no good rule for LaTeX to use to decide just when to do it for us. So the italic correction is always left up to the typist.

Sentences and paragraphs

Let us create our very first LaTeX document, which will consist of just a few paragraphs.

As mentioned above, paragraph input is free-form. You type the words and separate them by spaces so that LaTeX can distinguish between words. For these purposes, pressing Return is equivalent to inserting a space—it does not indicate the end of a line, but the end of a word. You tell LaTeX that a sentence has ended by typing a period followed by a space. LaTeX ignores extra spaces; typing three or three thousand will get you no more space between the words that these spaces separate than typing just one space. Finally, you tell LaTeX that a paragraph has ended by leaving one or more blank lines. In summary: LaTeX concerns itself only with the logical concepts end-of-word, end-of-sentence, and end-of-paragraph. Sounds complicated? The example in Figure 3.3 should clear things up. Try running LaTeX on this input.

We have learned more than just how LaTeX recognises words, sentences and paragraphs. We have also seen how to specify our choice of document class and how to tell LaTeX where our document begins and ends. Any material that is to be printed must lie somewhere between the

```
\documentclass{article}
\begin{document}
Words within a sentence are ended by spaces.  One space
between words  is     equivalent      to any number. We are only
interested in separating      one    word      from        the
next, not in formatting       the space between them.
For these purposes, pressing Return
at the end of a line
and starting a new word on the next line
just serves to separate
words, not to cut a line short.
The end of a sentence is indicated by a period
followed by one or more spaces.

The end of a paragraph is indicated by leaving a blank line.
All this
means that we can type without too much regard for layout, and
the typesetter will sort things out for us.
\end{document}
```

produces the result

> Words within a sentence are ended by spaces. One space between words is equivalent to any number. We are only interested in separating one word from the next, not in formatting the space between them. For these purposes, pressing Return at the end of a line and starting a new word on the next line just serves to separate words, not to cut a line short. The end of a sentence is indicated by a period followed by one or more spaces. The end of a paragraph is indicated by leaving a blank line. All this means that we can type without too much regard for layout, and the typesetter will sort things out for us.

Figure 3.3: Some TeX input and the corresponding output

---

declaration of `\begin{document}` and that of `\end{document}`. Definitions that are to apply to the entire document can be made before the declaration of the document beginning. The specification of document class must precede all other material.

In future examples we will not explicitly display the commands that select document class and delimit the start and end of the document. But if you wish to try any of the examples, do not forget to include those commands. The `article` document class will do for most of our examples. Of course, the preceding example looks not at all like an article because it is so short and because we specified no title or author information.

Most of what you need to know to type regular text is contained in the example above. When you consider that by far the majority of any document consists of straight text, it is obvious that LaTeX makes this fabulously straightforward. LaTeX will do all the routine work of formatting, and we simply get on with the business of composing.

LaTeX does more than simply choose pleasing line breaks and provide natural spacing when setting a paragraph. Remember we said that TeX has inherited the knowledge of generations of professional printers—well part of that knowledge includes being on the look-out for *ligatures*.

These are combinations of letters within words which should be typeset as a single special symbol because they will "clash" with each if this is not done. Have a look at these words

flight, flagstaff, chaff, fixation

and compare them with these

flight, flagstaff, chaff, fixation

See the difference? In the first set I let LaTeX run as it usually does. In the second I overruled it somewhat, and stopped it from creating ligatures. Notice how the 'fl', 'ff', and 'fi' combinations are different in the two sets—in the former they form a single symbol (a ligature) and in the latter they are comprised of two disjoint symbols. There are other combinations that yield ligatures, but we do not have to bother remembering any of them because LaTeX will take care of these, too.

Notice, too, that LaTeX has been taught how to hyphenate the majority of words. It will hyphenate a word if it feels that the overall quality of the paragraph will be improved. For long words it has been taught several potential hyphenation positions.

LaTeX also goes to a lot of trouble to try to choose pleasing page breaks. It avoids "widows", which are single lines of a paragraph occurring by themselves at either the bottom of a page (where it would have to be the first line of a paragraph) or at the top of a page (where it would have to be the last). It also "vertically justifies" your page so that all pages have exactly the same height, no matter what appears on them. As testimony to the success of the pagebreaking algorithm, I have (to this point) not once chosen a page break in this document.

Punctuation

Typists have a convention whereby a single space is left after a mid-sentence comma, and two spaces are left after a sentence-ending period. How do we enforce this if LaTeX treats a string of spaces just like a single one? The answer, unsurprisingly, is that we *do not*.

```
To have a comma followed by the appropriate space,  we simply
type a comma follows by at least one space. To end a sentence
we type a period with at least one following space.     No space will
be inserted if we type a comma or period followed straight away
by something other than a space, because there are times when
we will not require any space, i.e., we do what comes naturally.
```

will produce

> To have a comma followed by the appropriate space, we simply type a comma follows by at least one space. To end a sentence we type a period with at least one following space. No space will be inserted if we type a comma or period followed straight away by something other than a space, because there are times when we will not require any space, i.e., we do what comes naturally.

LaTeX will produce suitable space after commas, periods, semi-colons and colons, exclamation marks, question marks etc. if they are followed by a space. In stretching a line to justify to the right margin, it also knows that space after a punctuation character should be

more "stretchable" than normal inter-word space and that space after a sentence-ending period should be stretched more than space after a mid-sentence comma. TeX knows the nature of punctuation if you stick to the simple rules outlined here. As we have already said, those rules tell LaTeX how to distinguish consecutive words, sentences, phrases, etc.

Actually, there is more to ending sentences than mentioned above. Since LaTeX cannot speak English, it works on the assumption that *a period followed by a space ends a sentence unless the period follows a capital letter*. This works most of the time, but can fail. To get a normal inter-word space after a period that does not end a sentence, follow the period by a *control space*—a `\␣` (a `\` character followed by a space or return). Very rarely, you will have to force a sentence to end after a period that follows a capital letter (remember that LaTeX assumes this does not end a sentence). This is done by preceding the period with a `\@` command (you can guess from the odd syntax that this is rarely needed).

It is time we saw some examples of this. After all, this is our first experience of *control symbols* (do not worry, there are many more to come).

```
We must be careful not to confuse intra-sentence periods
with periods that end a sentence, i.e.\ we must remember
that our task is to describe the sentence structure.  Periods
that the typesetter requires a little help with typically result
from abbreviations, as in etc.\ and  others.  We have to work
somewhat harder to break a sentence after a capital letter,
but that should not bother us to much if we keep up our intake
of vitamin E\@.  All this goes for other sentence-ending
punctuation characters, so I could have said vitamin E\@!
Fortunately, these are rare occurrences.
```

results in

> We must be careful not to confuse intra-sentence periods with periods that end a sentence, i.e. we must remember that our task is to describe the sentence structure. Periods that the typesetter requires a little help with typically result from abbreviations, as in etc. and others. We have to work somewhat harder to break a sentence after a capital letter, but that should not bother us to much if we keep up our intake of vitamin E. All this goes for other sentence-ending punctuation characters, so I could have said vitamin E! Fortunately, these are rare occurrences.

Quotation marks is another area where LaTeX will do some work for us. Keyboards have the characters `‘`, `’`, and `"` but we want to have access to each of ‘, ’, “, and ”. So we proceed like this:

```
‘\LaTeX’ is no conventional word-processor, and
to to get quotes, like ‘‘this’’, we type repeated
{\tt ‘} and {\tt ’} characters.  Note that modern
convention is that ‘‘punctuation comes after
the closing quote character’’.
```

which gives just what we want

> ‘LaTeX’ is no conventional word-processor, and to to get quotes, like “this”, we type repeated ‘ and ’ characters. Note that modern convention is that “punctuation comes after the closing quote character”.

Very rarely, you have three quote characters together. Merely typing those three quote characters one-after-the-other is ambiguous—how should they be grouped? You tell LaTeX how you want them grouped by inserting a very small space called a *thin space*, and invoked with `\,`.

```
``\,`Green ham' or `Eggs?'\,'' is the question.
```

gives the desired result

> "'Green ham' or 'Eggs?'" is the question.

Since we have a typesetter at our disposal, we might as well use the correct dashes where we need them. There are three types of dash: the hyphen , the en-dash, and the em-dash. A minus sign is not a dash.

Hyphens are typed as you would hope, just by typing a `-` at the point in the word that you want a hyphen. Do not forget that LaTeX takes care of hyphenation that is required to produce pretty linebreaks. You only type a hyphen when you explicitly want one to appear, as in a combination like "inter-college".

An en-dash is the correct dash to use in indicating number ranges, as in "questions 1–3". To specify an en-dash you type two consecutive dashes on the keyboard, as in `1--3`.

An em-dash is a punctuation dash, used at the end of a sentence—I tend to use them too much. To specify an em-dash you type three consecutive dashes on the keyboard, as in "`...a sentence---I tend to...`".

```
Theorems 1--3 concern the semi-completeness
of our new construct---in the case that it
satisfies the first axiom.
```

yields

> Theorems 1–3 concern the semi-completeness of our new construct—in the case that it satisfies the first axiom.

Ties

When you always remember to use *ties*, you know that you are becoming TeXnically inclined. Ties are used to prevent LaTeX from breaking lines at certain places. LaTeX will always choose line breaks that result in the most aesthetically pleasing paragraph as judged by its stringent rules. But because LaTeX does not actually understand the material it is setting so beautifully, it can make some poor choices.

A *tie* is the character `~`. It behaves as a normal interword space in all respects *except* that the line-breaking algorithm will never break a line at that point. Thus

```
Dr. Seuss should be typed as Dr.~Seuss
```

for this makes sure that LaTeX will never leave the 'Dr' at the end of one line and put the 'Seuss' at the beginning of the next.

One should try to get in to the habit of typing ties first-time, not after waiting to see if LaTeX will make a poor choice. This will allow you to make all sorts of changes to your text without ever having to go back and insert a tie at a point that has migrated to the end of a line from the middle of a line as a result of those changes.

Figure 3.4 shows some more examples of places where you should remember to place ties.

```
Lemmas 3 and~4          Chapter~10
Donald~E. Knuth         Appendix~C
width~2                 Figure~1
function~f              Theorem~2
1,~2, or~3              Lemmas 3 and~4
equals~5
```

Figure 3.4: Some places where ties are useful.

Over-ruling some of TeX's choices

We have seen that ties can be used to stop linebreaks occurring between words. But how can we stop LaTeX from hyphenating a particular word? More generally, how can we stop it from splitting any given group of characters across two lines. The answer is to make that group of characters appears as one solid *box*, through use of an `\mbox` command.

```
For instance, if we wanted to be sure that the word
{\em currentitem\/} is not split across lines
then we should type it as \mbox{\em currentitem}.
```

If for some reason we wish to break a line
in the middle of nowhere, preventing LaTeX from justifying that line to the prevailing right margin, then we use the `\newline` command. One can also use the abbreviated form `\\`.

```
We start with a short line.\newline
And  now we continue with the normal
text, remembering that where we press
Return in the input file probably will not
correspond to a line break in the final
document.  More short lines\\
are easy, too.
```

will produce the line breaks we want

We start with a short line.
And now we continue with the normal text, remembering that where we press Return in the input file probably will not correspond to a line break in the final document. More short lines
are easy, too.

A warning is in order: `\newline` must only end part of a line that is "already set". It cannot be used to add additional space between paragraphs, nor to leave space for a picture

you want to paste in. This is not to be awkward, but is just part of LaTeX holding up its end of the deal by making you have to specially request additional vertical space. This prevents unwanted extra space from entering your document.

Later we shall see how to impose our own choice of page size, paragraph indentation, etc. For now we will continue to accept those declared for us in the document class.

3.3.5 Document structure

Sectioning commands

As part of our task of describing the logical structure of the document, we must indicate to LaTeX where to start sectional units. To do this we make use of the sectioning commands: `chapter`, `section`, `subsection`, `subsubsection`.

Each sectioning command accepts a single argument—the section heading that is to be used. LaTeX will provide the section numbering (and numbering of subsections within sections, etc.) so there is no need to include any number in the argument. LaTeX will also take care of whatever spacing is required to set the new logical unit off from the others, perhaps through a little extra space and using a larger font. It will also start a new page in the case that a new chapter is begun.

It is always a good idea to *plan* the overall sectional structure of a document in advance, or at least give it a little thought. Not that it would be difficult to change your mind later (you could use the global replace feature of an editor, for instance), but so that you have a good idea of the structure that you have to describe to LaTeX.

The sectioning command that began the present sectional unit of this document was

```
\subsection{Document structure}
```

and that was all that was required to get the numbered section name and the table of contents entry.

There are occasions when you want a heading to have all the appearance of a particular sectioning command, but should not be numbered as a section in its own right or produce a table of contents entry. This can be achieved through using the **-form* of the command, as in `\section*{...}`. We will see that many LaTeX commands have such a *-form which modify their behaviour slightly.

Not only will LaTeX number your sectional units for you, it will compile a table of contents too. Just include the command `\tableofcontents` after the `\begin{document}` command and after the topmatter that should precede it.

LaTeX environments

Perhaps the most powerful and convenient concept in the LaTeX syntax is that of an *environment*. We will see most of the "heavy" typesetting problems we will encounter can be best tackled by one or other of the LaTeX environments.

Some environments are used to *display* a portion of text, i.e. to set it off from the surrounding

text by indenting it. The `center` environment (note the American spelling!) allows us to centre portions of text, while the `flushright` environment sets small portions of text flush against the right margin.

But the true power of LaTeX begins to show itself when we look at environments such as those that provide facilities for itemized or enumerated lists, complex tabular arrangements, and for taking care of figure and table positioning and captioning. What we learn here will also be applicable in typesetting some complicated mathematical arrangements in the next section.

All the environments are begun by a `\begin{`*name*`}` command and ended by an `\end{`*name*`}`, where *name* is the environment name. These commands also serve as begin-group and end-group markers (see Sect. 3.3.4), so that all commands are local to the present environment—they cannot affect text outside the environment.

We can also have environment embedded within environment within environment and so on, limited only by memory available on the computer. We must, however, be careful to check that each of these *nested* environments is indeed contained within the one just outside of it.

`quote` environment

This environment can be used to display a part of a sentence or paragraph in such a manner that the material stands out from the rest of the text. This can be used to enhance readability, or to simply emphasize something. Its syntax is simple:

```
Horace smiled and retaliated:
\begin{quote}
\em You can mock the non-WYSIWYG nature of \TeX\
all you like.  You do not understand that that is
precisely what makes \TeX\ enormously more powerful
than that lame excuse for a typesetter you use.
And I will bet that from start to finish of preparing
a document I am quicker than you are, even if you
do type at twice the speed and have the so-called
advantage of WYSIWYG.  In your case, what you see
is {\em all\/} you get!
\end{quote}
and then continued with composing his masterpiece of the
typesetting art.
```

produces the following typeset material:

Horace smiled and retaliated:

> *You can mock the non-WYSIWYG nature of TeX all you like. You do not understand that that is precisely what makes TeX enormously more powerful than that lame excuse for a typesetter you use. And I will bet that from start to finish of preparing a document I am quicker than you are, even if you do type at twice the speed and have the so-called advantage of WYSIWYG. In your case, what you see is all you get!*

and then continued with composing his masterpiece of the typesetting art.

That is a much more readable manner of presenting Horace's piece of mind than embedding it within a regular paragraph. The `quote` environment was responsible for the margins being

indented on both sides during the quote. This example has also been used to show how the commands that begin and end an environment restrict the scope of commands issued within that environment: The `\em` at the beginning of the quote did not affect the text following the quote. We have also learned here that if we use `\em` within already emphasized text, the result is roman type—and we do not require an italic correction here because the final letter of 'all' was not sloping to the right.

Although LaTeX does not care too much for how we format our source file, it is obviously a good idea to lay it out logically and readably anyway. This helps minimize errors as well as aids in finding them. For this reason I have adopted the convention of always placing the environment `\begin` and `\end` commands on lines by themselves.

`center` environment

This environment allows the centering of consecutive lines of text, new lines being indicated by a \\. If you do not separate lines with the \\ command then you will get a centred paragraph the width of the page, which will not look any different to normal. If only one line is to be centred, then no \\ is necessary.

```
The {\tt center} environment takes care of the vertical
spacing before and after it, so we do not need to leave any.
\begin{center}
If we leave no blank line after the\\
{\tt center} environment\\
then the ensuing text will not\\
be regarded as part of a new\\
paragraph, and so will not be indented.\\
\end{center}

In this case we left a blank line after the environment,
so the new text was regarded as starting a new paragraph.
```

gives the following text

> The `center` environment takes care of the vertical spacing before and after it, so we do not need to leave any.
>
> If we leave no blank line after the
> `center` environment
> then the ensuing text will not
> be regarded as part of a new
> paragraph, and so will not be indented.
>
> In this case we left a blank line after the environment, so the new text was regarded as starting a new paragraph.

`verbatim` environment

We can simulate typed text using the `verbatim` environment. The `\tt` (typewriter text) type style can be used for simulating typed words, but runs into trouble if one of the char-

acters in the simulated typed text is a specially reserved LaTeX character. For instance, `{\tt type \newline}` would not have the desired effect because LaTeX would interpret the `\newline` as an instruction to start a new line.

The `verbatim` environment allows the simulation of multiple typed lines. *Everything* within the environment is typeset in typewriter font exactly as it appears in our source file—obeying spaces and line breaks as in the source file and not recognising the existence of any special symbols.

```
\begin{verbatim}
In the verbatim environment we can type anything
we like.
So we do not need to look out for uses of %, $, & etc,
nor will control sequences like \newline have any
effect.
\end{verbatim}
```

will produce the simulated input text

```
In the verbatim environment we can type anything
we like.
So we do not  need to look out for uses of %, $, & etc,
nor will control sequences like \newline have any
effect.
```

The only thing that cannot be typed in the `verbatim` environment is the sequence `\end{verbatim}`. You might notice that I still managed to simulate that control sequence above. One can always get what you want in TeX, perhaps with a little creativity.

If we want only to simulate a few typed words, such as when I say to use `\newline` to start a new line, then the `\verb` command is used. This command has a slightly odd syntax, pressed upon it by the use for which it was intended. It cannot accept an argument, because we may want to simulate typed text that is enclosed by `{braces}`. What one does is to choose any character that is *not* in the text to be simulated, and use a pair of these characters as "argument delimiters". I usually use the `@` or `"` characters, as I rarely have any other uses for them. Thus

> use `\%` to obtain a % sign

is typed as

```
use \verb"\%" to obtain a \% sign
```

or

```
use \verb@\%@ to obtain a \% sign
```

`itemize`, `enumerate`, `description` environments

LaTeX provides three predefined list-making environment, and a "primitive" list environment for designing new list environments of your own. We shall just describe the predefined ones here.

There is delightfully little to learn in order to be able to create lists. The only new command is `\item` which indicates the beginning of a new list item (and the end of the last one if this is not the first item). This command accepts an optional argument (which means you would enclose it in square brackets) that can be used to provide an item label. If no optional argument is given, then LaTeX will provide the item label for you; in an `itemize` list it will bullet the items, in an `enumerate` list it will number the items, and in a list of `description`s the default is to have no label (which would look a bit odd, so you are expected to use the optional argument there).

Remember that `\item` is used to separate list items; it does not accept the list item as an argument.

```
\begin{itemize}
\item an item is begun with \verb@\item@
\item if we do not specify labels, then
      \LaTeX\ will bullet the items for us
\item I indent lines after the first in the
      input file, but that is just to keep things
      readable.  As always, \LaTeX\ ignores additional
      spaces.

\item a blank line between items is ignored, for
      \LaTeX\ is responsible for spacing items.
\item \LaTeX\ is in paragraph-setting mode when
      it reads the text of an item, and so will
      perform all the usual functions
\end{itemize}
```

produces the following itemized list:

- an item is begun with `\item`
- if we do not specify labels, then LaTeX will bullet the items for us
- I indent lines after the first in the input file, but that is just to keep things readable. As always, LaTeX ignores additional spaces.
- a blank line between items is ignored, for LaTeX is responsible for spacing items.
- LaTeX is in paragraph-setting mode when it reads the text of an item, and so will perform all the usual functions

Lists can also be embedded within one another, for they are just environments and we said that environments have this property. Remember that we must nest them in the correct order. We demonstrate with the following list, which also shows how to use the `enumerate` environment.

```
\noindent I still have to do the following things:
\begin{enumerate}
\item Sort out LAN accounts for people on the course
   \begin{itemize}
   \item Have new accounts created for those not already
         registered on the LAN
   \item Make sure all users have a personal directory
        on the data drive
   \item Add users to the appropriate LAN print queues
   \end{itemize}
\item Have a \TeX\ batch file added to a directory that
      is on a public search path
\item Finish typing these course notes and proof-read them
\item Photocopy and bind the finished notes
\end{enumerate}
```

will give the following list

I still have to do the following things:

1. Sort out LAN accounts for people on the course
   - Have new accounts created for those not already registered on the LAN
   - Make sure all users have a personal directory on the data drive
   - Add users to the appropriate LAN print queues
2. Have a TeX batch file added to a directory that is on a public search path
3. Finish typing these course notes and proof-read them
4. Photocopy and bind the finished notes

See how I lay the source file out in a readable fashion. This is to assist myself, not LaTeX. The `description` environment is, unsurprisingly, for making lists of descriptions.

```
\begin{description}
\item[itemize] an environment for setting itemized lists.
\item[enumerate] an environment for setting numbered lists.
\item[description] an environment for listing descriptions
(like for words in a dictionary with boldface and a nice little indentation
after the first line).
\end{description}
```

will typeset the descriptions shown in Figure 3.5. Note that the scope of the `\tt` commands used in the item labels was restricted to the labels.

itemize an environment for setting itemized lists.

enumerate an environment for setting numbered lists.

description an environment for listing descriptions (like for words in a dictionary with boldface and a nice little indendation after the first line).

Figure 3.5: The description environment.

`tabular` environment

The `tabular` environment is used to produce tables of items, particularly when the table is predominantly rectangular and when line drawing is required. LaTeX will make most decisions for us; for instance it will align everything for us without having to be told which are the longest entries in each column.

This environment is the first of many that use the TeX "tabbing character" `&`. This character is used to separate consecutive entries in a row of a table, array, etc. The end of a row is indicated in the usual manner, by using `\\`. In this way the individual cells of the table, or array, are clearly described to LaTeX, and it can analyse them to make typesetting decisions. Commands issued within a cell so defined are, again, local to that cell.

The `tabular` environment is also our first example of an *environment with arguments*. The arguments are given, in braces as usual, just after the closing brace after the environments name. In the case of `tabular` there is a single mandatory argument giving the justification of the entries in each column: `l` for left justified, `r` for right justified, and `c` for centred. There must be an entry for each column of the table, and there is no default. Let us start with a simple table.

```
\begin{tabular}{llrrl}
\bfseries{Student name} & \bfseries{Number}
       & \bfseries{Test 1} & \bfseries{Test 2} & \bfseries{Comment}\\
F. Basset    & 865432 & 78     & 85     & Pleasing\\
H. Hosepipe  & 829134 & 5      & 10     & Improving\\
I.N. Middle  & 853931 & 48     & 47     & Can make it
\end{tabular}
```

will produce the following no-frills table

Student name	**Number**	**Test 1**	**Test 2**	**Comment**
F. Basset	865432	78	85	Pleasing
H. Hosepipe	829134	5	10	Improving
I.N. Middle	853931	48	47	Can make it

Note that a `\\` was not necessary at the end of the last row. Also note that, once again, the alignment of the `&` characters was for human readability. It is conventional to set columns of numbers with right justification. The `\bf` directives apply only to the entries in which they are given.

A `|` typed in the `tabular` environment's argument causes a vertical line to be drawn at the indicated position and extending for the height of the entire table. An `\hline` given in the environment draws a horizontal line extending the width of the table to be drawn at the vertical position at which the command is given. A `\cline{`i-j`}` draws a line spanning columns i to j, at the vertical position at which the command is given. A repeated line-drawing command causes a double line to be drawn. We illustrate line drawing in tables by putting some lines into our first table. We will type this example in a somewhat expanded form, trying to make it clear why the lines appear where they do.

```
\begin{tabular}{|l|l|r|r|l|}
\hline
\bfseries{Student name} & \bfseries{Number} & \bfseries{Test 1} & \bfseries{Test 2}
      & \bfseries{Comment}\\
\hline
F. Basset        & 865432     & 78         & 85         & Pleasing\\
\hline
H. Hosepipe      & 829134     & 5          & 10         & Improving\\
\hline
I.N. Middle      & 853931     & 48         & 47         & Can make it\\
\hline
\end{tabular}
```

which will give

Student name	**Number**	**Test 1**	**Test 2**	**Comment**
F. Basset	865432	78	85	Pleasing
H. Hosepipe	829134	5	10	Improving
I.N. Middle	853931	48	47	Can make it

That way of laying out the source file makes it clear where the lines will go. As we (by now) well know, the returns that we pressed after the \\s in typing this table might as well have been spaces as far as LaTeX is concerned. Thus it is common to have the `\hline` commands following the \\s on the input lines. We will do this in future examples.

The `\multicolumn` column can be used to overrule the overall format of the table for a few columns. The syntax of this command is

`\multicolumn {`*n*`}{`*pos*`}{`*item*`}`

where *n* is the number of columns of the original format that *item* is to span, and *pos* specifies the justification of the new argument.

```
\begin{tabular}{||l|c|c|c||} \hline
\multicolumn{4}{|c|}{\LaTeX\ size changing commands}\\ \hline
Style option         & 10pt (default) & \ttfamily 11pt & \ttfamily 12pt\\ \hline
\tt footnotesize  & 8pt            & 9pt      & 10pt\\ \hline
\tt small         & 9pt            & 10pt     & 11pt\\ \hline
\tt large         & 12pt           & 12pt     & 14pt\\ \hline
\end{tabular}
```

produces the following table:

LaTeX size changing commands			
Style option	10pt (default)	`11pt`	`12pt`
`footnotesize`	8pt	9pt	10pt
`small`	9pt	10pt	11pt
`large`	12pt	12pt	14pt

`figure` and `table` environments

Figures (diagrams, pictures, etc.) and tables (perhaps created with the `tabular` environment) cannot be split across pages. So LaTeX provides a mechanism for "floating" them to a nearby place where there is room for them. This may mean that your figure or table may appear a little later in the document than its declaration in the source file might suggest. You can suggest to LaTeX that it try to place the figure or table at the present position if there is room or, failing that, at the top or bottom of the present or following page. You can also ask for it to be presented by itself on a "page of floats".

You suggest these options to LaTeX through an optional argument to the environment. One lists a combination of the letters `h`, `t`, `b` and `p` with the following meaning:

h the object should be placed *here* if there is room, so that things will appear in the same order as in the source file,

t the object can be placed at the *top* of the of a text page, but no earlier than the present page.

b the object can be placed at the *bottom* of a text page, but no earlier than the present page.

p the object should be set on a *page of floats* that consists only of tables and figures.

A combination of these indicates decreasing order of preference. The default is `tbp`.

You may also force LaTeX to place the figure or table in a desired spot by using capital `H`. To do this you must include the float package by using `\usepackage{float}` in your preamble.

LaTeX will also number a figure or table for you, supply a caption and compile a list of tables and a list of figures. Just include `\listoffigures` and `\listoftables` next to your `\tableofcontents` command at the beginning of the document. To caption a table of figure, include `\caption{`*caption text*`}` just before the `\end{table}` or `\end{figure}` command. Here is a sample source file.

```
\begin{table}[htbp]
   \begin{tabular}{lrll}
   ...
   \end{tabular}
\caption{Mark analysis}
\end{table}
```

To leave space for a figure that will inserted by some other means at a later date, we can use the `\vspace` command:

```
\begin{figure}[htbp]
\vspace{9.5cm}
\caption{An artists impression}
\end{figure}
```

3.4 Mathematical typesetting with LaTeX

Original text by Gavin Maltby (1992); adapted by Math-2130 Instructors (1995,1998)

The last section taught us a good deal of what we need to know in order to prepare quite complicated non-mathematical documents. There are still a number of useful topics that we have not covered (such as cross-referencing), but we will defer discussion of those until a later section. In the present chapter, we will learn how LaTeX typesets mathematics. It should come as no surprise that LaTeX does most of the work for us.

3.4.1 Introduction

In text-only documents we saw that our task was to describe the logical components of each sentence, paragraph, section, table, etc. When we tell LaTeX to go into *mathematical mode*, we have to describe the logical parts of a formula, matrix, operator, special symbol, etc. TeX has been taught to recognize a binary operation, a binary relation, a variable, an operator that expects limits, and so on. We just need to supply the parts that make up each of these, and TeX will take care of the rest. It will leave appropriate space around operators, italicize variables, set an operator name in roman type, leave the correct space after colons, place sub- and superscripts in the correct positions (based on what it is you are working with), choose the correct typesizes, ... the list of things it has been taught is enormous. When you want to revert to setting normal text again, you tell LaTeX to leave math mode and go back into the mode it was in (paragraphing mode).

LaTeX cannot be expected to perform these mode shifts itself, for it is not always clear just when it is mathematics that has been typed. For example, should an isolated letter `a` in the input file be regarded as a word (as in the definite article) or a mathematical variable (as in the variable a). There are no reliable rules for LaTeX to make such decisions by, so the begin-math and end-math mode switching is left entirely to you.

The symbol `$` is specially reserved (See Sect. 3.3.3) by LaTeX as the "math shift" symbol. When LaTeX starts setting a document it is in paragraphing mode, ready to set lines of the input file into paragraphs. It remains in this mode until it encounters a `$` symbol, which shifts LaTeX into mathematical mode. It now knows to be on the look-out for the components of a mathematical expression, rather than for words and paragraphs. It reads everything up to the next `$` sign in this mathematical mode, and then shifts back to paragraphing mode (i.e. the mode it was in before we took it in to math mode).

You must be careful to balance your begin-math and end-math symbols. It is often a good idea to type two `$` symbols and then move back between them and type the mathematical expression. If the math-shift symbols in a document are not matched, then LaTeX will become confused because it will be trying to set non-mathematical material as mathematics.

For those who find having the same symbol for both math-begin and math-end confusing or dangerous, there are two control symbols that perform the same operations: the control symbol `\(` is a begin-math instruction, and the control symbol `\)` is an end-math instruction. Since

it is easy to "lose" a `$` sign when typing a long formula, a math environment is provided for such occasions: you can use `\begin{math}` and `\end{math}` as the math-shift instructions. Of course, you could just decide to use `$` and take your chances.

Let us have a look at some mathematics.

```
\LaTeX\ is normally in paragraphing mode, where
it expects to find the usual paragraph material.  Including
a mathematical expression, like $2x+3y - 4= -1$, in the
paragraph text is easy.  \TeX\ has been taught to recognize
the basic elements of an expression, and typeset them appropriately,
choosing spacing, positioning, fonts, and so on.
Typing the above expression without entering math
mode produces the incorrect result: 2x+3y - 4= -1
```

will produce the following paragraph

> LaTeX is normally in paragraphing mode, where it expects to find the usual paragraph material. Including a mathematical expression, like $2x + 3y - 4 = -1$, in the paragraph text is easy. TeX has been taught to recognize the basic elements of an expression, and typeset them appropriately, choosing spacing, positioning, fonts, and so on. Typing the above expression without entering math mode produces the incorrect result: 2x+3y - 4= -1

Notice that LaTeX sets space around the binary relation = and space around the binary operators + and − on the left hand side of the equation, ignoring the spacing we typed in the input. It was also able to recognize that the -1 on the right hand side of the equation was a unary minus—negating the 1 rather than being used to indicate subtraction—and so did not put space around it. It also italicized the variables x, y, and z. However, it did not italicize the number 1.

In typing a mathematical expression we must remember to keep the following in mind:

1. All letters that are not part of an argument to some control sequence will be italicized. Arguments to control sequences will be set according to the definition of the command used. So typing `$f(x)>0 for x > 1$` will produce

   $$f(x) > 0 for x > 1$$

   instead of the expression

   $$f(x) > 0 \quad \text{for} \quad x > 1$$

   that we intended. Numerals and punctuation marks are set in normal roman type but LaTeX will take care of the spacing around punctuation symbols, as in

   ```
   $f(x,y) \geq 0$
   ```

   which produces

   $$f(x, y) \geq 0 \quad .$$

2. Even a single letter can constitute a formula, as in "the constant a". To type this you enter `$a$` in your source file. If you do not go in to math mode to type the symbol, you will get things like "the constant a".

3. Some symbols have a different meaning when typed in math mode. Not only do ordinary letters become variables, but symbols such as `-` and `+` are now interpreted as mathematical symbols. Thus in math mode `-` is no longer considered a hyphen, but as a minus sign.

4. LaTeX ignores all spaces and carriage returns when in math mode, without exception. So typing something like `the constant$ a$` will produce "the constant$a$". You should have typed `the constant $a$`. LaTeX is responsible for all spacing when in math mode, and (as in paragraphing mode) you have to specially ask to have spacing changed. Even if LaTeX does ignore all spaces when in math mode you should (as always in TeX) still employ spaces to keep your source file readable.

The above means that, at least for most material, a typist need not understand the mathematics in order to typeset it correctly. And even if one does understand the mathematics, LaTeX is there to make sure that you adhere to accepted typesetting conventions (whether you were aware of their existence or not). So one could type either

```
$f(x, y)  =  2 (x+ y)y/(5xy - 3  )$
```

or

```
$f(x,y) = 2(x+y)y / (5xy-3)$
```

and you would still get the correct result

$$f(x, y) = 2(x + y)y/(5xy - 3) \quad .$$

There are some places where this can go wrong. For instance, if we wish to speak of the x-y plane then one has to know that it is an *en-dash* that is supposed to be placed between the x and the y, not a minus sign (as `$x-y$` would produce). But typing `$x--y$` will produce $x - -y$ since both dashes are interpreted as minus signs. To avoid speaking of the $x - y$ plane or the $x - -y$ plane, we should type it as `the $x$--$y$ plane`. We are fortunate that LaTeX can recognise and cope with by far the majority of our mathematical typesetting needs.

Another thing to look out for is the use of braces in an expression. Typing

```
${x : f(x)>0}$
```

will not produce any braces. This is because, as we well know, braces are reserved for delimiting groups in the input file. Looking back to section 3.3.3, we see how it should be done:

```
$ \{ x: f(x)>0 \} $
```

Math shift commands also behave as scope delimiters, so that commands issued in an expression cannot affect anything else in a document.

3.4.2 Displaying a formula

LaTeX considers an expression `$ ... $` to be word-like in the sense that it considers it to be eligible for splitting across lines of a paragraph (but without hyphenation, of course). LaTeX assigns quite a high penalty to doing this, thus trying to avoid it (remember that LaTeX tries to minimize the "badness" of a paragraph). When there is no other way, it will split the expression at a suitable place. But there are some expressions which are just too long to fit into the running text without looking awkward. These are best "displayed" on a line by themselves. Also, some expressions are sufficiently important that they should be made to stand out. These, too, should be displayed on a line of their own.

The mechanism for displaying an expression is the *display math* mode, which is begun by typing `$$` and ended by typing the same sequence (which again means that we had better be sure to type them in pairs). Corresponding to the alternatives `\(` and `\)` that we had for the math shift character `$`, we may use `\[` and `\]` as the display-math shift sequences. One can also use the environment

```
\begin{displaymath} ... \end{displaymath}
```

which is equivalent to `$$ ... $$` and is suitable for use with long displayed expressions. If you wish LaTeX to number your equations for you you can use the environment

```
\begin{equation} ... \end{equation}
```

which is the same as the `displaymath` environment, except that an equation number will be generated.

It is poor style to have a displayed expression either begin a paragraph or be a paragraph by itself. This can be avoided if you agree to *never leave a blank line in your input file before a math display*.

We will see later how to typeset an expression that is to span multiple lines. For now, let us look at an example of displaying an expression:

```
For each $a$ for which the Lebesgue-set $L_a(f) \neq \emptyset$ we define
$$ % We could have used \begin{displaymath} here
   {\cal B}_a(f) = \{ L_{a+r}(f) : r > 0  \},
$$ % and \end{displaymath} here and these are easily seen to be completely regular.
```

which produces

> For each a for which the Lebesgue-set $L_a(f) \neq \emptyset$ we define
>
> $$\mathcal{B}_a(f) = \{L_{a+r}(f) : r > 0\},$$
>
> and these are easily seen to be completely regular.

That illustrates how to display an expression, but also shows that we have got a lot more to learn about mathematical typesetting. Before we have a look at how to arrange symbols all over the show (e.g. the subscripting above) we must learn how to access the multitude of symbols that are used in mathematical texts.

3.4.3 Using mathematical symbols

LaTeX puts all the esoteric symbols of mathematics at our fingertips. They are all referenced by name, with the naming system being perfectly logical and systematic. None of the control words that access these symbols accepts an argument, but we will soon see that some of them prepare LaTeX for something that might follow. For instance, when you ask for the symbol '$\sum$' LaTeX is warned that any sub- or superscripts that follow should be positioned appropriately as limits to a summation. In keeping with the TeX spirit, none of this requires any additional work on your part.

We will also see that some of the symbols behave differently depending on where they are used. For instance, when I ask for $\sum_{i=1}^{n} a_i$ within the running text, the limits are places differently to when I ask for that expression to be displayed:

$$\sum_{i=1}^{n} a_i \quad .$$

Again, I typed nothing different here—just asked for display math mode.

It is important to note that *almost all of the special math symbols are unavailable in ordinary paragraphing mode*. If you need to use one there, then use an in-line math expression `$...$`.

Symbols available from the keyboard

A small percentage of the available symbols can be obtained from just a single key press. They are $+ - = < > | / ( ) [ ]$ and $*$. Note that these must be typed *within math mode* to be interpreted as math symbols.

Of course, all of *a*–*z*, *A*–*Z*, the numerals $0, 1, 2, \ldots, 9$ and the punctuation characters , ; and : are available directly from the keyboard. Alphabetic letters will be assumed to be variables that are to be italicized, unless told otherwise (see Sect. 3.4.4). The numerals receive no special attention, appearing precisely as in normal paragraphing mode. The punctuation symbols are still set in standard roman type when read in math mode, but a little space is left after them so that expressions like $\{x_i : i = 1, 2, \ldots, 10\}$ look like they should. Note that this means that normal sentence punctuation should not migrate into an expression.

Greek letters

Tables 3.4 and 3.5 show the control sequences that produce the letters of the Greek alphabet. We see that a lowercase Greek letter is simply is accessed by typing the control word of the same name as the symbol, using all lowercase letters. To obtain an uppercase Greek letter, simply capitalise the *first* letter of its name.

Just as `$mistake$` produces *mistake* because the letters are interpreted as variables, so too will `$\tau \epsilon \chi$` produce the incorrectly spaced $\tau\epsilon\chi$ if you try to type greek words like this. TeX can be taught to set Greek, but this is not the way. $\tau\epsilon\chi$, incidentally, is the Greek word for "art" and it is from the initials of the Greek letters constituting this word that the name TeX was derived. TeX is "the art of typesetting".

α	`\alpha`	β	`\beta`	γ	`\gamma`	δ	`\delta`
ϵ	`\epsilon`	ε	`\varepsilon`	ζ	`\zeta`	η	`\eta`
θ	`\theta`	ϑ	`\vartheta`	ι	`\iota`	κ	`\kappa`
λ	`\lambda`	μ	`\mu`	ν	`\nu`	ξ	`\xi`
π	`\pi`	ϖ	`\varpi`	ρ	`\rho`	ϱ	`\varrho`
σ	`\sigma`	ς	`\varsigma`	τ	`\tau`	υ	`\upsilon`
ϕ	`\phi`	φ	`\varphi`	χ	`\chi`	ψ	`\psi`
ω	`\omega`						

Table 3.4: Lowercase Greek letters

Γ	`\Gamma`	Δ	`\Delta`	Θ	`\Theta`	Λ	`\Lambda`
Ξ	`\Xi`	Π	`\Pi`	Σ	`\Sigma`	Υ	`\Upsilon`
Φ	`\Phi`	Ψ	`\Psi`	Ω	`\Omega`		

Table 3.5: Uppercase Greek letters

Calligraphic uppercase letters

The letters $\mathcal{A}, \ldots, \mathcal{Z}$ are available through use of the style changing command `\cal`. This command behaves like the other style changing commands `\em`, `\it`, etc. so its scope must be delimited as with them. Thus we can type

```
... for the filter $\cal F$ we have $\varphi({\cal F}) = \cal G$.
```

to obtain

> for the filter $\mathcal{F}$ we have $\varphi(\mathcal{F}) = \mathcal{G}$.

There is no need to tabulate all the calligraphic letters, since they are all obtained by just a type style changing command. We will just list them so that we can see, for reference purposes, what they all look like. Here they are:

$$\mathcal{ABCDEFGHIJKLMNOPQRSTUVWXYZ}$$

Binary operators

LaTeX has been taught to recognise binary operators and set the appropriate space either side of one—i.e., it sets the first argument followed by a little space, then the operator followed by the same little space and finally the second argument. Table 3.6 shows the binary operators that are available via LaTeX control words (recall that the binary operators $+$, $-$, and $*$ can be typed from the keyboard). Here are some examples of their use:

Type	*To produce*
`$a+b$`	$a+b$
`$(a+b) \otimes c$`	$(a+b) \otimes c$
`$(a \vee b) \wedge c$`	$(a \vee b) \wedge c$
`$X - (A \cap B) = (X-A) \cup (X-B)$`	$X - (A \cap B) = (X - A) \cup (X - B)$

±	\pm	∩	\cap	◇	\diamond	⊕	\oplus
∓	\mp	∪	\cup	△	\bigtriangleup	⊖	\ominus
×	\times	⊎	\uplus	▽	\bigtriangledown	⊗	\otimes
÷	\div	⊓	\sqcap	◁	\triangleleft	⊘	\oslash
∗	\ast	⊔	\sqcup	▷	\triangleright	⊙	\odot
⋆	\star	∨	\vee	∧	\wedge	◯	\bigcirc
†	\dagger	\	\setminus	∐	\amalg	∘	\circ
‡	\ddagger	·	\cdot	≀	\wr	•	\bullet

Table 3.6: Binary Operation Symbols

Binary relations

LaTeX has been taught to recognize the use of binary relations, too. Table 3.7 shows those available via LaTeX control words. There are a few that you can obtain directly from the keyboard: <, >, =, and |.

To negate a symbol you can precede the control word that gives the symbol by a `\not`. Some symbols come with ready-made negations, which should be used instead of the `\not`'ing method because the slope of the negating line is just slightly changed to look more pleasing. Thus `\notin` should be used instead of `\not\in` and `\ne` should be used instead of `\not =`.

If negating a symbol produces a slash whose horizontal positioning is not to your liking, then use the math spacing characters described in section 3.4.4 to adjust it.

≤	\leq	≥	\geq	≡	\equiv	⊨	\models
≺	\prec	≻	\succ	∼	\sim	⊥	\perp
⪯	\preceq	⪰	\succeq	≃	\simeq	∣	\mid
≪	\ll	≫	\gg	≍	\asymp	∥	\parallel
⊂	\subset	⊃	\supset	≈	\approx	⋈	\bowtie
⊆	\subseteq	⊇	\supseteq	≅	\cong	⋈	\Join
⊏	\sqsubset	⊐	\sqsupset	≠	\neq	⌣	\smile
⊑	\sqsubseteq	⊒	\sqsupseteq	≐	\doteq	⌢	\frown
∈	\in	∋	\ni	∝	\propto		
⊢	\vdash	⊣	\dashv				

Table 3.7: Binary relations

Miscellaneous symbols

Table 3.8 shows a number of general-purpose symbols. Remember that these are only available in math mode. Note that `\imath` and `\jmath` should be used when you need to accent an i or a j in math mode (see Sect. 3.4.3) —you cannot use `\i` or `\j` that were available in paragraphing mode. To get a prime symbol, you can use `\prime` or you can just type `'` when in math mode, as in `$f''(x)=x$` which produces $f''(x) = x$.

The symbols $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$, $\mathbb{H}$, commonly used to denote number domains (natural, integer, rational, real, complex, and quaternion, respectively) are obtained by the commands like `\mathbb{N}`. You need to `\includepackage{amssymb}` in the preamble of your document.

$\aleph$	\aleph	$\prime$	\prime	$\forall$	\forall	∞	\infty
$\hbar$	\hbar	$\emptyset$	\emptyset	$\exists$	\exists	$\Box$	\Box
$\imath$	\imath	∇	\nabla	$\neg$	\neg	$\triangle$	\triangle
$\jmath$	\jmath	$\surd$	\surd	$\flat$	\flat	$\triangle$	\triangle
ℓ	\ell	$\top$	\top	$\natural$	\natural	$\clubsuit$	\clubsuit
$\wp$	\wp	$\bot$	\bot	$\sharp$	\sharp	$\diamondsuit$	\diamondsuit
$\Re$	\Re	$\|$	|	$\backslash$	\backslash	$\heartsuit$	\heartsuit
$\Im$	\Im	$\angle$	\angle	∂	\partial	$\spadesuit$	\spadesuit
$\mho$	\mho						

Table 3.8: Miscellaneous symbols

Arrow symbols

LaTeX has a multitude of arrow symbols, which it will set the correct space around. Note that vertical arrows can all be used as delimiters—see section 3.4.3. The available symbols are listed in table 3.9.

$\leftarrow$	\leftarrow	$\longleftarrow$	\longleftarrow	$\uparrow$	\uparrow
$\Leftarrow$	\Leftarrow	$\Longleftarrow$	\Longleftarrow	$\Uparrow$	\Uparrow
$\rightarrow$	\rightarrow	$\longrightarrow$	\longrightarrow	$\downarrow$	\downarrow
$\Rightarrow$	\Rightarrow	$\Longrightarrow$	\Longrightarrow	$\Downarrow$	\Downarrow
$\leftrightarrow$	\leftrightarrow	$\longleftrightarrow$	\longleftrightarrow	$\updownarrow$	\updownarrow
$\Leftrightarrow$	\Leftrightarrow	$\Longleftrightarrow$	\Longleftrightarrow	$\Updownarrow$	\Updownarrow
$\mapsto$	\mapsto	$\longmapsto$	\longmapsto	$\nearrow$	\nearrow
$\hookleftarrow$	\hookleftarrow	$\hookrightarrow$	\hookrightarrow	$\searrow$	\searrow
$\leftharpoonup$	\leftharpoonup	$\rightharpoonup$	\rightharpoonup	$\swarrow$	\swarrow
$\leftharpoondown$	\leftharpoondown	$\rightharpoondown$	\rightharpoondown	$\nwarrow$	\nwarrow
$\rightleftharpoons$	\rightleftharpoons	$\leadsto$	\leadsto		

Table 3.9: Arrow symbols

Expression delimiters

A pair of delimiters often enclose an expression, as in

$$\left[\begin{array}{cc} a_{11} & a_{12} \\ a_{21} & a_{22} \end{array} \right] \quad \text{and} \quad f(x) = \left\{ \begin{array}{ll} x & \text{if } x < 1 \\ x^2 & \text{if } x \geq 1 \end{array} \right. .$$

LaTeX will scale delimiters to the correct size (determined by what they enclose) for you, if you ask it to. There are times when you do not want a delimiter to be scaled, so it is left up to you to ask for scaling.

To ask that a delimiter be scaleable, you precede it by `\left` or `\right` according as it is the left or right member of the pair. Scaled delimiters must be balanced correctly. It sometimes occurs, as in the right-hand example above, that only one member of a delimiting pair is to be visible. For this purpose, use the commands `\left.` and `\right.` which will produce no visible delimiter but can be used to correctly balance the delimiters in an expression. For examples of the use of delimiters, see section 3.4.4 where we learn about arrays.

Table 3.10 shows the symbols that LaTeX will recognize as delimiters, i.e. symbols that may follow a `\left` or a `\right`. Note that you have to use `\left\{` and `\right\}` in order to get scaled braces.

(	(	)	)	↑	`\uparrow`
[	]	]	]	↓	`\downarrow`
{	`\{`	}	`\}`	↕	`\updownarrow`
⌊	`\lfloor`	⌋	`\rfloor`	⇑	`\Uparrow`
⌈	`\lceil`	⌉	`\rceil`	⇓	`\Downarrow`
⟨	`\langle`	⟩	`\rangle`	⇕	`\Updownarrow`
/	/	\	`\backslash`		
|	|	‖	`\\|`		

Table 3.10: Delimiters

Operators like $\int$ and $\sum$

These behave differently when used in display-math mode as compared with in-text math mode. When used in text, they will appear in their small form and any limits provided will be set so as to reduce the overall height of the operator, as in $\sum_{i=1}^{N} f_i$. When used in display-math mode, LaTeX will choose to use the larger form and will not try to reduce the height of the operator, as in

$$\sum_{i=1}^{N} f_i \quad .$$

Table 3.11 describes what variable-size symbols are available, showing both the small (in text) and the large (displayed) form of each. In section 3.4.4 we will learn how to place limits on these operators.

$\sum$ $\sum$	`\sum`	$\bigcap$ $\bigcap$	`\bigcap`	$\bigodot$ $\bigodot$	`\bigodot`
$\prod$ $\prod$	`\prod`	$\bigcup$ $\bigcup$	`\bigcup`	$\bigotimes$ $\bigotimes$	`\bigotimes`
$\coprod$ $\coprod$	`\coprod`	$\bigsqcup$ $\bigsqcup$	`\bigsqcup`	$\bigoplus$ $\bigoplus$	`\bigoplus`
$\int$ $\int$	`\int`	$\bigvee$ $\bigvee$	`\bigvee`	$\biguplus$ $\biguplus$	`\biguplus`
$\oint$ $\oint$	`\oint`	$\bigwedge$ $\bigwedge$	`\bigwedge`		

Table 3.11: Variable-sized symbols

Accents

The accenting commands that we learned for paragraphing mode do not apply in math mode. Consult table 3.12 to see how to accent a symbol in math mode (all the examples there accent the symbol u, but they work with any letter). Remember that i and j should lose their dots when accented, so `\imath` and `\jmath` should be used.

There also exist commands that give a "wide hat" or a "wide tilde" to their argument, `\widehat` and `\widetilde`.

$\hat{u}$	`\hat{u}`	$\acute{u}$	`\acute{u}`	$\bar{u}$	`\bar{u}`	$\dot{u}$	`\dot{u}`
$\check{u}$	`\check{u}`	$\grave{u}$	`\grave{u}`	$\vec{u}$	`\vec{u}`	$\ddot{u}$	`\ddot{u}`
$\breve{u}$	`\breve{u}`	$\tilde{u}$	`\tilde{u}`				

Table 3.12: Math accents

3.4.4 Some common mathematical structures

In this section we shall begin to learn how to manipulate all the symbols listed in section 3.4.3. Indeed, by the end of this section we will be able to typeset some quite large expressions. In the section following this we will learn how use various alignment environments that allow us to prepare material like multi-line expressions and arrays.

Subscripts and superscripts

Specifying a sub- or superscript is as easy as you would hope—you just give an indication that you want a sub- or superscript to the last expression and provide the material to be placed there, and LaTeX will position things correctly. So sub- and superscripting a single symbol, an operator, or a big array all involve the same input, and LaTeX places the material according to what the expression is that is being sub- or superscripted:

$$x^2 \quad , \quad \prod_{i=1}^{N} X_i \quad , \quad \left[\begin{array}{ccc} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{array} \right]^2 \quad .$$

To tell LaTeX that you want a single character set as a superscript to the last expression, you just type a `^` before it. The "last expression" is the preceding group or, if there is no preceding group, the single character or symbol that the `^` follows:

Type	*To produce*
`$x^2$`	x^2
`$a^b$`	a^b
`$Y^X$`	Y^X
`$\gamma^2$`	γ^2
`$(A+B)^2$`	$(A+B)^2$
`$\left[ \frac{x^2+1}{x^2 + y^2} \right]^n$`	$\left[\frac{x^2+1}{x^2+y^2}\right]^n$

Subscripts of a single character are equally easy—you just use the underscore character `_` where you used `^` for superscripting:

Type	*To produce*
`$x_2$`	x_2
`$x_i$`	x_i
`$\Gamma_1(x)$`	$\Gamma_1(x)$

Now let us see how to set a sub- or superscript that consists of more than just one character. This is no more difficult than before if we remember the following rule: *`_` and `^` set the group that follows them as a sub- and superscripts to the group that precedes the sub- and superscript symbols.* We see now now that our initial examples worked by considering a single character to be a group by itself. Here are some examples:

Type	*To produce*
`$a^2b^3$`	a^2b^3
`$2^{21}$`	2^{21}
`$2^21$`	2^21
`$a^{x+1}$`	a^{x+1}
`$a^{x^2+1}$`	a^{x^2+1}
`$(x+1)^3$`	$(x+1)^3$
`$\Gamma_{\alpha\beta\gamma}$`	$\Gamma_{\alpha\beta\gamma}$
`${}_1A_2$`	${}_1A_2$

In the very last example we see a case of setting a subscript to an empty group, which resulted in a kind of "pre-subscript". With some imagination this can be put to all sorts of uses.

In all of the above examples the sub- and superscripts were set to single-character groups. Nowhere did we group an expression before sub- or superscripting it. Even in setting the expression $(x+1)^3$, the superscript ^{3} was really only set to the character). If we had wanted to group the $(x+1)$ before setting the superscript, we would have typed `${(x+1)}^3$` which gives $(x+1)^3$, with the superscript slightly raised. One has to go to this trouble because, to most people, something like $(x^a)^b$ is just as acceptable and as readable as $(x^a)^b$. It also has the advantage of aligning the base lines in expressions such as

$$(ab)^{-2} = [(ab)^{-1}]^2 = [b^{-1}a^{-1}]^2 = b^{-1}a^{-1}b^{-1}a^{-1}$$

which looks more pleasing than if we use additional grouping to force

$$(ab)^{-2} = \left[(ab)^{-1}\right]^2 = \left[b^{-1}a^{-1}\right]^2 = b^{-1}a^{-1}b^{-1}a^{-1} \quad ,$$

and the latter has rather more braces in it that require balancing.

Here are some more examples, showing how LaTeX will set things just as we want without any further work on our part:

Type	*To produce*
`$x^{y^z}$`	x^{y^z}
`$2^{(2^2)}$`	$2^{(2^2)}$
`$2^{2^{2^{\aleph_0}}}$`	$2^{2^{2^{\aleph_0}}}$
`$\Gamma^{z_c^d}$`	$\Gamma^{z_c^d}$

We can also make use of empty groups in order to stagger sub- and superscripts to an expression, as in

```
$\Gamma_{\alpha\beta}{}^{\gamma}{}_\delta$
```

which will yield

$$\Gamma_{\alpha\beta}{}^{\gamma}{}_{\delta}$$

One can specify the sub- and superscripts to a group in any order, but it is best to be consistent. The most natural order seems to be to have subscripts first, but you may think otherwise. It is also a good idea to always include your sub- and superscripts in braces (i.e. make them a group), whether they consist of just a single character or not. This enhances readability and also helps avoid the unfortunate case where you believe that a particular control word gives a single symbol yet it really is defined in terms of several.

Primes

LaTeX provides the control word `\prime` ($\prime$) for priming symbols. Note that it is not automatically at the superscript height, so that to get f' you would have to type

```
$f^\prime$ .
```

To make lighter work of this, LaTeX will interpret a right-quote character as a prime if used in math mode. Thus we can type

```
$f'(g(x)) g'(x) h''(x)$
```

in order to get

$$f'(g(x))g'(x)h''(x) \quad .$$

Fractions

LaTeX provides the `\frac` command that accepts two arguments: the numerator and the denominator (in that order). Before we look at examples of its use, let us just note that many simple in-text fractions are often better written in the form *num/den*, as with 3/8 which can be typed as `$3/8$`. This is also often the better form for a fraction that occurs *within* some expression.

Type	*To produce*
`$\frac{x+1}{x+2}$`	$\frac{x+1}{x+2}$
`$\frac{1}{x^2+1}$`	$\frac{1}{x^2+1}$
`$\frac{1+x^2}{x^2+y^2} + x^2 y$`	$\frac{1+x^2}{x^2+y^2} + x^2 y$
`$\frac{1}{1 + \frac{x}{2}}$`	$\frac{1}{1+\frac{x}{2}}$
`$\frac{1}{1+x/2}$`	$\frac{1}{1+x/2}$

Roots

The `\sqrt` command accepts two arguments. The first, and optional, argument specifies what order of root you desire if it is anything other than the square root. The second, and mandatory, argument specifies the expression that the root sign should enclose:

Type	*To produce*
`$\sqrt{a+b}$`	$\sqrt{a+b}$
`$\sqrt[5]{a+b}$`	$\sqrt[5]{a+b}$
`$\sqrt[n]{\frac{1+x}{1+x^2}}$`	$\sqrt[n]{\frac{1+x}{1+x^2}}$
`$\frac{\sqrt{x+1}} {\sqrt[3]{x^3+1}}$`	$\frac{\sqrt{x+1}}{\sqrt[3]{x^3+1}}$

Ellipsis

Simply typing three periods in a row will not give the correct spacing of the periods if it is an ellipsis that is desired. So LaTeX provides the commands `\ldots` and `\cdots`. Centered ellipsis should be used between symbols like $+$, $-$, $*$, $\times$, and $=$. Here are some examples:

Type	*To produce*
`$a_1+ \cdots + a_n$`	$a_1 + \cdots + a_n$
`$x_1 \times x_2 \times \cdots \times x_n$`	$x_1 \times x_2 \times \cdots \times x_n$
`$v_1 = v_2 = \cdots = v_n = 0$`	$v_1 = v_2 = \cdots = v_n = 0$
`$f(x_1,\ldots,x_n) = 0$`	$f(x_1, \ldots, x_n) = 0$

Text within an expression

One can use the `\mbox` command to insert normal text into an expression. This command forces LaTeX temporarily out of math mode, so that its argument will be treated as normal

text. Its use is simple, but we must be wary on one count: remember that LaTeX ignores all space characters when in math mode; so to surround words in an expression that were placed by an `\mbox` command by space you must include the space in the `\mbox` argument.

Type	*To produce*
`$f_i(x) \leq 0 \mbox{ for } x \in I$`	$f_i(x) \leq 0$ for $x \in I$
`$\Gamma(n)=(n-1)! \mbox{ when $n$ is an integer}$`	$\Gamma(n) = (n-1)!$ when n is an integer

In section 3.4.4 we will learn of some special spacing commands that can be used in math mode. These are often very useful in positioning text within an expression, enhancing readability and logical layout.

Log-like functions

There are a number of function names and operation symbols that should be set in normal (roman) type in an expression, such as in

$$f(\theta) = \sin\theta + \log(\theta + 1) - \sinh(\theta^2 + 1)$$

and

$$\lim_{h \to 0} \frac{\sin h}{h} = 1 \quad .$$

We know that simply typing `$log\theta$` would produce the incorrect result $log\theta$ and that using `$\mbox{log}\theta$` would leave us having to insert a little extra space between the log and the θ $\log\theta$. So LaTeX provides a collection of "log-like functions" defined as control sequences. Thus `$\log\theta$` produces the perfect $\log\theta$. Table 3.13 shows various log-like functions that are available and some examples of their use. Notice how LaTeX does more than just set an operation like sup in roman type. It also knew where a subscript to that operator should go.

`\arccos`	`\cos`	`\csc`	`\exp`	`\ker`	`\limsup`	`\min`	`\sinh`
`\arcsin`	`\cosh`	`\deg`	`\gcd`	`\lg`	`\ln`	`\Pr`	`\sup`
`\arctan`	`\cot`	`\det`	`\hom`	`\lim`	`\log`	`\sec`	`\tan`
`\arg`	`\coth`	`\dim`	`\inf`	`\liminf`	`\max`	`\sin`	`\tanh`

Table 3.13: Log-like functions

Type	*To produce*
`$f(x)=\sin x + \log(x^2)$`	$f(x) = \sin x + \log(x^2)$
`$\delta = \min \{ \delta_1, \delta_2 \}$`	$\delta = \min\{\delta_1, \delta_2\}$
`$\chi(X) = \sup_{x\in X} \chi(x)$`	$\chi(X) = \sup_{x \in X} \chi(x)$
`$\lim_{n \rightarrow \infty} S_n = \gamma$`	$\lim_{n \to \infty} S_n = \gamma$

Over- and Underlining and bracing

The `\underline` command will place an unbroken line under its argument, and the `\overline` command will place an unbroken line over its argument. These two commands can also be used in normal paragraphing mode (but be careful: LaTeX will not break the line within an under- or overlined phrase, so do not go operating on large phrases).

You can place horizontal braces above or below an expression by making that expression the argument of `\overbrace` or `\underbrace`. You can place a label on an overbrace (resp. underbrace) by superscripting (resp. subscripting the group defined by the bracing command.

Type	*To produce*
`$\overline{a+bi} = a- bi$`	$\overline{a+bi} = a - bi$
`$\overline{\overline{a+bi}} = a+bi$`	$\overline{\overline{a+bi}} = a + bi$

And some examples of horizontal bracing:

```
$A^n=\overbrace{A \times A \times \ldots \times A}^{\mbox{$n$ terms}}$

$\forall x \underbrace{\exists y (y \succ x)}_{\mbox{scope of $\forall$}}$
```

will produce

$$A^n = \overbrace{A \times A \times \ldots \times A}^{n \text{ terms}}$$

and

$$\forall x \underbrace{\exists y (y \succ x)}_{\text{scope of } \forall}$$

Stacking symbols

LaTeX allows you to set one symbol above another through the `\stackrel` command. This command accepts two arguments, and sets the first centrally above the second.

Type	*To produce*
`$X \stackrel{f^*}{\rightarrow}Y$`	$X \stackrel{f^*}{\rightarrow} Y$
`$f(x) \stackrel{\triangle}{=} x^2 + 1$`	$f(x) \stackrel{\triangle}{=} x^2 + 1$

Operators; Sums, Integrals, etc.

Each of the operation symbols in table 3.11 can occur with limits. They are specified as sub- and superscripts to the operator, and LaTeX will position them appropriately. In an in-text formula they will appear in more-or-less the usual scripting positions; but in a displayed formula they will be set below and above the symbol (which will also be a little larger). The following should give you an idea of how to use them:

Type	*To produce*
`$\sum_{i=1}^{N} a_i$`	$\sum_{i=1}^{N} a_i$
`$\int_a^b f$`	$\int_a^b f$
`$\oint_{\cal C}f(x)\,dx$`	$\oint_{\mathcal{C}} f(x)\,dx$
`$\prod_{\alpha \in A} X_\alpha$`	$\prod_{\alpha \in A} X_\alpha$
`$\lim_{N\rightarrow\infty}\sum_{i=1}^{N}f(x_i)\Delta x_i$`	$\lim_{N\rightarrow\infty}\sum_{i=1}^{N} f(x_i)\Delta x_i$

We will have more to say about the use of `\,` in section 3.4.4. Let us have a look at each of those expressions when displayed:

$$\sum_{i=1}^{N} a_i\,, \qquad \int_a^b f\,, \qquad \oint_{\mathcal{C}} f(x)\,dx\,, \qquad \prod_{\alpha\in A} X_\alpha\,, \qquad \lim_{N\rightarrow\infty}\sum_{i=1}^{N} f(x_i)\Delta x_i$$

Arrays

The `array` environment is provided for typesetting arrays and array-like material. It accepts two arguments, one optional and one mandatory. The optional argument specifies the vertical alignment of the array—use `t`, `b`, or `c` to align the top, bottom, or centre of the array with the centreline of the line it occurs on (the default being `c`). The second argument is as for the `tabular` environment: a series of `l`, `r`, and `c`'s that specify the number of columns and the justification of these columns. The body of the `array` environment uses the same syntax as the `tabular` environment to specify the individual entries of the array.

For instance the input

```
$A =\left[ \begin{array}{rrr}
12 & 3 & 4\\
-2 & 1 & 0\\
3 & 7 & 9
\end{array}\right]$ ...
```

will produce the output

$$A = \left[\begin{array}{rrr} 12 & 3 & 4 \\ -2 & 1 & 0 \\ 3 & 7 & 9 \end{array}\right]$$

Note that we had to choose and supply the enclosing brackets ourselves (they are not placed for us so that we can use the `array` environment for array-like material; also, we get to choose what type of brackets we want this way). As in the `tabular` environment, the scope of a command given inside a matrix entry is restricted to that entry.

We can use ellipsis within arrays as in the following example:

$$\det A = \begin{array}{cccc} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{array}\,.$$

It has been produced by

```
$\det A = \left| \begin{array}{cccc}
a_{11} & a_{12} & \cdots & a_{1n}\\
a_{21} & a_{22} & \cdots & a_{2n}\\
\vdots & \vdots & \ddots & \vdots\\
a_{m1} & a_{m2} & \cdots & a_{mn}
\end{array} \right|.$
```

The `array` environment is often used to typeset material that is not, strictly speaking, an array:

```
$f(x) = \left\{ \begin{array}{ll}
   x & \mbox{for $x<1$}\\
 x^2 & \mbox{for $x \geq 1$}
\end{array} \right.$.
```

which will yield

$$f(x) = \begin{array}{ll} x & \text{for } x < 1 \\ x^2 & \text{for } x \geq 1 \end{array} .$$

Changes to spacing

Sometimes LaTeX needs a little help in spacing an expression, or perhaps you think that the default spacing needs adjusting. For these purposes we have the following spacing commands:

`\,`	thin space	`\:`	medium space
`\!`	negative thin space	`\;`	thick space
`\quad`	a quad of space	`\qquad`	two quads of space

The spacing commands `\,`, `\quad`, and `\qquad` can be used in paragraphing mode, too. Here are some examples of these spacing commands used to make subtle modifications to some expressions.

Type	*To produce*
`$\sqrt{2} \, x$`	$\sqrt{2}\,x$
`$\int_a^b f(x)\,dx$`	$\int_a^b f(x)\,dx$
`$\Gamma_{\!2}$`	Γ_2
`$\int_a^b \! \int_c^d f(x,y)\,dx\,dy$`	$\int_a^b\!\int_c^d f(x,y)\,dx\,dy$
`$x / \! \sin x$`	$x/\!\sin x$
`$\sqrt{\,\sin x}$`	$\sqrt{\,\sin x}$

3.4.5 Alignment

Recall that the `equation` environment can be used to display and automatically number a single-line equation (see Sect. 3.4.2). The `eqnarray` environment is used for displaying and

automatically numbering either a single expression that spreads over several lines or multiple expressions, while taking care of alignment for us. The syntax is similar to that of the `tabular` and `array` environments, except that no argument is necessary to declare the number and justification of columns. The `eqnarray*` environment does this without numbering any equations. Thus

```
\begin{eqnarray}
(a+b)(a+b) & = & a^2 + 2ab + b^2\\
(a+b)(a-b) & = & a^2 - b^2
\end{eqnarray}
```

will give

$$\begin{aligned}(a+b)(a+b) &= a^2+2ab+b^2 &(3.1)\\ (a+b)(a-b) &= a^2-b^2 &(3.2)\end{aligned}$$

See how we identify the columns so as to align the = signs. We can also leave entries empty. The following output, for instance,

$$\begin{aligned}\frac{d}{dx}\sin x &= \lim_{h\rightarrow 0}\frac{\sin(x+h)-\sin x}{h}\\ &= \lim_{h\rightarrow 0}\frac{\sin x\cos h+\cos x\sin h-\sin x}{h}\\ &= \lim_{h\rightarrow 0}\left\{\frac{\sin x(\cos h-1)}{h}+\cos x\frac{\sin h}{h}\right\}\\ &= \cos x\end{aligned}$$

has been produced from the input below:

```
\begin{eqnarray*}
\frac{d}{dx} \sin x & = & \lim_{h\rightarrow0}\frac{\sin(x+h)-\sin x}{h}\\
 & = & \lim_{h\rightarrow0}\frac{\sin x\cos h + \cos x\sin h - \sin x}{h}\\
 & = & \lim_{h\rightarrow0}\frac{\sin x(\cos h-1)}{h} + \cos x\frac{\sin h}{h}\\[8pt]
 & = & \cos x\,.
\end{eqnarray*}
```

No first column entry is required. The \\[8pt] gives extra space. Note also that `\displaystyle` command is not required in `eqnarray` environment, while it would be required if we were to produce the same result using `array` environment.

3.4.6 Theorems, Propositions, Lemmas, ...

Suppose you document contains four kinds of theorem-like structures: "theorems", "propositions", "conjectures", and "wild guesses". Then near the beginning of the document you should have something like the following:

```
\newtheorem{thm}{Theorem}
\newtheorem{prop}{Proposition}
\newtheorem{conjec}{Conjecture}
\newtheorem{wildshot}{Hypothesis} % make it sound good!
```

The first argument to `\newtheorem` defines a new theorem-like environment name of your own choosing. The second argument contains the text that you want inserted when your theorem is proclaimed:

```
\begin{thm} {\slshape $X$ is normal if, and only if, each pair of disjoint
closed sets in $X$ is completely separated.}
\end{thm}

\begin{wildshot} % remember, we chose the name 'wildshot'
The property of Moore extends to all objects of the class $\Sigma$.
\end{wildshot}
```

which will produce the following:

Theorem 1 *X is normal if, and only if, each pair of disjoint closed sets in X is completely separated.*

Hypothesis 1 *The property of Moore extends to all objects of the class Σ.*

Notice that LaTeX italicizes the theorem statement, and that you still have to shift in to math mode when you want to set symbols and expression. Typically, it is the class or style file that determines what a theorem will appear like—so do not go changing this if you are preparing for submission for publication (because the journal staff want to substitute their production style for your document class choice, and not be over-ridden by other commands).