Some tips and tricks for LaTeX

Y. Zwols (yz2198@columbia.edu)

February 27, 2012

Contents

1 A little bit on the internals of LaTeX **2**

1.1 Counters and lengths . . . 2

1.2 Commands . . . 4

1.2.1 Replacing existing LaTeX commands . . . 7

1.3 Creating environments . . . 7

2 Useful tools for 'debugging' your LaTeX document **9**

2.1 The `draft` option . . . 9

2.2 Checking cross references with `refcheck` . . . 9

3 Using multiple files **10**

3.1 Creating a style file (a.k.a. package) . . . 10

3.1.1 The layout of a .sty file . . . 10

3.1.2 Making sure LaTeX can find your .sty file . . . 11

3.2 Separating documents into several files . . . 11

3.3 Generating tables with Matlab or C(++) . . . 11

4 Layout **14**

4.1 Page headers and footers using `fancyhdr` . . . 14

4.1.1 Using the fancy style for the first page of chapters too . . . 14

4.2 Indentation and paragraph skip . . . 14

4.3 Choosing fonts . . . 15

4.4 Minipages . . . 15

4.5 Changing section headings using `sectsty` . . . 15

4.6 Customizing enumerations using `enumerate` 15
4.7 Table of contents 15
4.8 Nice chapter headings using `quotchap` 16
4.9 Nice fractions using `nicefrac` 16
4.10 Including source code (C++/Matlab/...) 16
4.11 Activating hyperlinks in PDF files 16
4.12 Scaling boxes with `scalebox` 16

5 Graphics **17**
5.1 Inkscape 17
5.2 TikZ 17

6 Examples **20**
6.1 Creating nice looking boxes 20
6.2 Putting parenthesis around references to equations 21

7 Getting help **22**
7.1 Package manuals 22

1

A little bit on the internals of LaTeX

Although you may not always realize it, LaTeX is an actual (admittedly quite ugly) programming language, just like C and Matlab. This means that it supports all sorts of constructs for making life easier when you are editing papers or a thesis. In this section, we will go over some useful techniques for the not-so-beginner LaTeX user.

1.1 Counters and lengths

While the LaTeX compiler compiles a document, it keeps track of a number of variables. LaTeX has two useful types of variables: counters and lengths. These variables have names, just like in any other programming language. Like the names suggest, counters are used to count, and lengths are used to measure lengths.

Counters Common counters include `page` (the current page number), `section` (the number of the current section), `subsection` (the number of the current subsection), etc. At any point in your document code, these variables are available to you. You can modify counters by using commands like `\stepcounter`, `setcounter`. You can also create your own counters by using the command `\newcounter`. Counters are mostly useful when defining your own commands and/or environments. See Section **??** for an example.

At times, you may want to turn a counter into text. This can be done in several ways. Say you have a counter `mycounter`. You can turn it into a numerical value using `\arabic{mycounter}`, into a roman numeral using `\roman{mycounter}`, a letter `\alph{mycounter}` (not to be confused with `\alpha`, which is the greek letter α), etc. The following example illustrates this:

Example 1.1.1: Counters

```
\newcounter{mycounter}
\setcounter{mycounter}{1}
\arabic{mycounter}, \alph{mycounter}, \Alph{mycounter},
\roman{mycounter}, \Roman{mycounter}

\stepcounter{mycounter}
\arabic{mycounter}, \alph{mycounter}, \Alph{mycounter},
\roman{mycounter}, \Roman{mycounter}
```

This has the following output:

1, a, A, i, I
2, b, B, ii, II

Lengths Common lengths include `parindent`, `parskip` (see **??**), and many more. You can change the values of length variables using the commands `\setlength`, `\addtolength`. Lengths are always specified by a number and a unit. Common units are in (inch), pt (point), cm (centimeter). Lengths are useful in many places. For example, if you want to make the width of tge columns in a table uniform, you can either type the lengths multiple times, or use a length. For example:

Example 1.1.2: Length variables

```
\newlength\mycolwidth
\setlength\mycolwidth{0.5in}

\begin{tabular}{|p{\mycolwidth}|p{\mycolwidth}|p{\mycolwidth}|}
\hline
A & B & C\\
D & E & F \\
\hline
\end{tabular}
```

This code has the following output:

A	B	C
D	E	F

You can do some arithmetic with length variables as well. For example, you may want to make the columns of your table about 1/3 of the page width, excluding margins. The length variable that measures this width is called `textwidth`. The following example illustrates the arithmetic:

Example 1.1.3: Length variable arithmetic

```
\newlength\mycolwidth
\setlength\mycolwidth{0.3\textwidth}

\begin{tabular}{|p{\mycolwidth}|p{\mycolwidth}|p{\mycolwidth}|}
\hline
A & B & C\\
D & E & F \\
\hline
\end{tabular}
```

This code has the following output:

A D	B E	C F

(Notice that I used `0.3\textwidth` and not `0.33\textwidth`. This is because the column width specified in the `p{...}` argument in the tabular environment measures the width of the column without the space between the columns. So the actual width of the column will exceed 3×`colwidth`.

1.2 Commands

Everyone is familiar with commands in LaTeX, e.g. the commands `\section`, `\textbf`. I think that knowing a little bit about the way these commands work internally in LaTeX improves your programming and debugging skills considerably.

The way LaTeX deals with commands is quite easy: every command, except for a set of 'primitive commands', has a definition. For example, the definition of the `\section` command contains all sorts of commands for ending the current line, setting up the correct fonts and font sizes, keeping track of the current section number, outputting the current section number and it titles, and starting a new paragraph. The primitive commands include `\setcounter`, `\addtolength`, etc. Internally, whenever the LaTeX compiler sees a non-primitive command, it replaces the command by its definition. So whenever you type `\section{Introduction}`, this command is expanded 'under water' to a much longer command. Thus, commands can be thought of as 'subroutines'. Many commands are part of the LaTeX distribution, but you can also create your own commands.

For example, in many documents, I find myself using the same commands over and over again. For example, I use the symbol $\mathbb{R}$ for the set of real numbers many times. Instead of writing `\mathbb{R}`, I prefer to keep it short. To achieve this goal, I always define at the beginning of every document a new command named `\dR` (actually I don't exactly do that – see Section **??**). Now, whenever I want to use the symbol $\mathbb{R}$, I just write `\dR`. The following example shows how to do this:

Example 1.2.1: Simple commands

```
\newcommand\dR{\mathbb{R}}
\newcommand\cP{\mathcal{P}}
...
Let $x\in \dR$. Let $\cP$ be the set of all perfect graphs.  ...
```

When this code is fed to the LaTeX compiler, whenever it comes accross the command `\dR`, it is replaced immediately by the code `\mathbbR`, which in turn is replaced by whatever the definition of `\mathbb` is (this probably involves selecting a different font, but we don't particularly care about that right now). Notice that the outer curly braces in the first line of the example are NOT part of the definition of `\dR`. Only the command inside the outer curly braces count. This is sometimes important as the following example shows. In this example, there are two commands. `\myname` changes the font to italic and then displays my name. The second command, `\mynamebetter` does exactly the same, but it has an extra set of curly braces. This makes sure that after writing my name, the font is reset to what it was before.

Example 1.2.2: Curly braces in command definitions

```
\newcommand\myname{\it Yori Zwols}
\newcommand\mynamebetter{{\it Yori Zwols}}

\normalfont\myname~is my name.

\normalfont\mynamebetter~is my name.
```

This has the following output:

> *Yori Zwols is my name.*
> *Yori Zwols* is my name.

The reason is that this code gets expanded as follows:

```
\normalfont\it Yori Zwols~is my name.

\normalfont{\it Yori Zwols}~is my name.
```

As said, one can think of these commands as subroutines (or functions) in your document, which suggests that it is possible to add arguments to a command. This is indeed possible and very useful. For example, I don't like writing `\{1, 2, \hdots, n\}` or `\{1, 2, \hdots, n\}` many times for typesetting the sets $\{1, 2, \ldots, n\}$ and $\{1, 2, \ldots, k\}$. Therefore, I always define a command named `\enum`, which stands for enumeration. The following example illustrates this:

Example 1.2.3: Commands with arguments

```
\newcommand\enum[1]{\ensuremath{\{1, 2, \hdots, #1\}}}
Let $i\in \enum{n}$ and $j\in \enum{k}$. The set \enum{k} is ...

\newcommand\enumb[2]{\ensuremath{\{#1, \hdots, #2\}}}
Let $q\in \enumb{5}{n}$.

\newcommand\advenum[2][1]{\ensuremath{\{#1, \hdots, #2\}}}
Let $i\in \advenum{n}$ and let $q \in \advenum[5]{n}$.
```

This code has the following output:

Let $i \in \{1, 2, \ldots, n\}$ and $j \in \{1, 2, \ldots, k\}$. The set $\{1, 2, \ldots, k\}$ is ...
Let $q \in \{5, \ldots, n\}$.
Let $i \in \{1, \ldots, n\}$ and let $q \in \{5, \ldots, n\}$.

The lines with command definitions of this example may seem a bit cryptic. In the first line, we are defining a new command called `\enum`. The part '[1]' means that the command will take exaclty one argument. The `ensuremath` part makes sure that whenever we use the `\enum` command, the part inside the `ensuremath` command is typeset in mathematics mode (notice that in the last line, we use the `\enum` command both in math mode and in normal text mode). Finally, the '#1' shows the LaTeX compiler where to insert the text that is given as an argument to the `\enum` command. The second command we define, `\enumb`, shows how to define more than one argument. The third commandm `\advenum` shows how to use optional arguments. The '[2]' means that the command takes in principle two arguments. However, the next '[1]' says that the default value of the first argument is '1'. Now, we can use `\advenum` in two ways: either we can specify one argument, and let the lower bound of the enumeration be 1, or we can specify two arguments, but we have to specify the first argument with square brackets.

This example also illustrates another good point: I sometimes decide not to use $\{1, 2, \ldots, n\}$ for these standard sets of integers, but instead I decided to use the notation $[n]$. By changing the first line of **??**, one can easily change the way any enumeration is typeset throughout the document, so no need to edit every single occurrence:

Example 1.2.4

```
\newcommand\enum[1]{\ensuremath{[#1]}}
...
Let $i\in \enum{n}$ and $j\in \enum{k}$. The set \enum{k} is ...
```

1.2.1 Replacing existing LaTeX commands

In the definitions in the previous section, we always created new commands. Sometimes, you want to redefine an already existing command. This can be done by using the `\renewcommand` instead. For example, I redefine the `\Re` and `\Im` (for the real and imaginary part of a complex number) commands as follows:

Example 1.2.5

```
\renewcommand{\Re}{\mathop{\rm Re}}
\renewcommand{\Im}{\mathop{\rm Im}}
```

A problem occurs when you want to redefine a command, but still use the old version of the command inside the new definition. If you do this in the straightforward way (i.e. redefine the command and refer to the same command), you will end up with a circular definition and your LaTeX document will not compile. The proper way to tackle this, is to make a copy of the original command. As an example, the following code shows how make all citations boldface.

Example 1.2.6

```
\let\oldcite=\cite
\renewcommand\cite[1]{{\normalfont\textbf{\oldcite{#1}}}}
```

The code works as follows: we first make a copy of the original command `\cite` and call this copy `\oldcite`. Then, we redefine the `\cite` command, and use the copy `\oldcite` to cite.

1.3 Creating environments

Another useful construction that you are familiar with is environments. Environments are like commands, but they have the advantage that you don't have to put the argument within curly braces. Instead, environments start with a `\begin{...}` command and end with a `\end{...}` command. Like commands, environments are expanded internally by the LaTeX compiler.

Creating your own environments is as easy as creating new commands. For example, I sometimes have equations that I don't want to number, but I want to give them a text label. I call these 'named equations'.

Example 1.3.1

```
\newenvironment{namedeq}[1]%
    {   \renewcommand{\theequation}{#1}%
        \begin{equation}%
    }%
    {   \end{equation}%
        \renewcommand{\theequation}{\arabic{equation}}%
        \addtocounter{equation}{-1}%
    }
...
\begin{namedeq}{LP1} \label{linearprogram}
\min c^Tx \mbox{ subject to } Ax \leq b, x\geq 0
\end{namedeq}
\eqnref{linearprogram} is a generic linear program.
```

This code gets expanded as follows:

```
\newenvironment{namedeq}[1]%
    {   %
    }%
    {
    }
...
\renewcommand{\theequation}{LP1}
\begin{equation} \label{linearprogram}
\min c^Tx \mbox{ subject to } Ax \leq b, x\geq 0
\end{equation}%
\renewcommand{\theequation}{\arabic{equation}}%
\addtocounter{equation}{-1}

\eqnref{linearprogram} is a generic linear program.
```

This has the following output:

$$\min c^T x \text{ subject to } Ax \leq b, x \geq 0 \tag{LP1}$$

(**??**) is a generic linear program.

Notice the use of counters in this example: the counter `equation` keeps track of the number of the current equation. The command `\begin{equation}` automatically increases the `equation` counter by one, but since we decided to give the equation a name instead of a number, there is no need to increase the counter. Hence the command `\addtocounter{equation}{-1}`, which decreases the counter by one again.

2

Useful tools for ‘debugging’ your LaTeX document

2.1 The `draft` option

By specifying the `draft` option ...

2.2 Checking cross references with `refcheck`

3

Using multiple files

When creating large documents, for example a Ph.D. thesis, it is tempting to put all the LaTeX code into one big file. Although this will compile completely fine, it becomes hard to manipulate such large files. The section deals with ways to divide a large document into separate files.

3.1 Creating a style file (a.k.a. package)

I tend to use the same definitions over and over again throughout different documents. Therefore, instead of copying and pasting these definitions into every new file, I have one personal package which I called `yzdefs`. Now, whenever I create a new LaTeX file, I just type `\usepackage{yzdefs}` and I get all my standard definitions and packages for free. In this section, we will go through the process of creating a package.

3.1.1 The layout of a .sty file

A .sty file is not much different from a .tex file. The main difference is that it does not include any commands that directly output text. It should only contains definitions. Also, you never compile a .sty file directly. Rather, you use the `\usepackage` command to load it into a .tex file. The following example illustrates how to create you own package, which I have named `mypackage.sty`:

Example 3.1.1: mypackage.sty, an example package

```
\NeedsTeXFormat{LaTeX2e}[1995/12/01]
\ProvidesPackage{mypackage}[My personal LaTeX definitions]

\RequirePackage{times}

\newcommand\dR{\mathbb{R}}
\newcommand\cP{\mathcal{P}}
\newcommand\enum[1]{\ensuremath{\{1, 2, \hdots, #1\}}}
```

The first two lines of this example form the header of the package. The important part is that the name of the package is correct in the second line, or you will get warnings when you compile your document. The command `\RequirePackage` is the package-equivalent of `\usepackage`. In this case, we load the Times New Roman fonts as default fonts. The last three lines define commands as before.

3.1.2 Making sure LaTeX can find your .sty file

In order to be able to use your newly created package, you have to make sure that the .sty file is available to the LaTeX compiler. You can do this by copying the .sty file to the directory in which you have a document that requires the package, but the problem is that you will have to make many copies of the same file. Hence, if you want to change the .sty file, you'll have to update all copies. A nicer way to make sure that LaTeX can find your package, is by installing it into your 'local texmf directory'. This directory can be found either in `c:\tex\localtexmf` (for MiKTeX on Windows), or `~/texmf` (for Linux or MacOS). To install it properly, you should create a new subdirectory in the `tex\latex` directory and put your package in that new subdirectory. For example, I put my `yzdefs` package in the directory `c:\tex\localtexmf\tex\latex\yzdefs` directory on Windows, and in the `~/texmf/tex/latex/yzdefs` directory on Linux. The name of the directory does not necessarily have to be the same as the name of your package.

3.2 Separating documents into several files

3.3 Generating tables with Matlab or C(++)

Now that we know how to include files into our main document, this opens a useful technique: automatically generating tables. The following two examples show how to automatically generate a LaTeX table from Matlab and C.

Example 3.3.1: Matlab code to generate a LaTeX table

```
% open file for writing text to
f = fopen('table1.tex', 'wt');

% write beginning of table
fprintf(f, '\\begin{tabular}{|c|rrr|}\n');
fprintf(f, '\\hline\n');
fprintf(f, '\\textbf{Sample} & 1 & 2 & 3\\\\\n');
fprintf(f, '\\hline\n');

% write contents of table
for i = 1:10
    fprintf(f, '%d & %0.2f & %0.2f & %0.2f \\\\\n', i, ...
        rand, rand, rand);
end

% write end of table
fprintf(f, '\\hline\n');
fprintf(f, '\\end{tabular}\n');

% close file
fclose(f);
```

Example 3.3.2: C code to generate a LaTeX table

```
#include <stdio.h>

double randu()
{
  return (rand() % 1000) / 1000.0;
}

int main()
{
  int i;

  // open file for writing text to
  FILE* f = fopen("table1.tex", "wt");

  // write beginning of table
  fprintf(f, "\\begin{tabular}{|c|rrr|}\n");
  fprintf(f, "\\hline\n");
  fprintf(f, "\\textbf{Sample} & 1 & 2 & 3\\\\\n");
  fprintf(f, "\\hline\n");

  // write contents of table
  for (i = 1; i <= 10; i++)
    fprintf(f, "%d & %0.2f & %0.2f & %0.2f \\\\\n",
                i, randu(), randu(), randu());

  // write end of table
  fprintf(f, "\\hline\n");
  fprintf(f, "\\end{tabular}\n");

  // close file
  fclose(f);
}
```

4

Layout

4.1 Page headers and footers using `fancyhdr`

4.1.1 Using the fancy style for the first page of chapters too

The `fancyhdr` package sets up fancy headers for all pages, except for the first page of every chapter. There is a simple remedy to fix this:

```
\fancypagestyle{plain}
```

4.2 Indentation and paragraph skip

There are two length variables that I tend to change at the beginning of any document, namely `parindent` and `parskip`. The length variable `parindent` represents the amount of indentation for every paragraph. The variable `parskip` represents the amount of space between paragraphs. The values of these variables can be changes as described in Section **??** For example:

Example 4.2.1: Setting `parindent` and `parskip`

```
\setlength\parindent{0pt}\setlength\parskip{16pt}
The purpose of this example is to illustrate how to use {\tt parindent} and
{\tt parskip}. This is a paragraph without indentation.

As you can see, to make up for it, there is space between paragraphs. \medskip

\hrule\medskip

\setlength\parindent{16pt}\setlength\parskip{0pt}
The purpose of this example is to illustrate how to use {\tt parindent} and
{\tt parskip}. This is a paragraph with indentation.

Clearly, now it is not necessary to set the {\tt parskip}
variable to a high value.
```

This code has the following output:

The purpose of this example is to illustrate how to use `parindent` and `parskip`. This is a paragraph with no indentation.

As you can see, to make up for it, there is space between paragraphs.

The purpose of this example is to illustrate how to use `parindent` and `parskip`. This is a paragraph with indentation.

Clearly, now it is not necessary to set the `parskip` variable to a high value.

4.3 Choosing fonts

4.4 Minipages

4.5 Changing section headings using `sectsty`

4.6 Customizing enumerations using `enumerate`

4.7 Table of contents

```
\setcounter{tocdepth}{2}
\setcounter{secnumdepth}{5}
```

4.8 Nice chapter headings using `quotchap`

4.9 Nice fractions using `nicefrac`

4.10 Including source code (C++/Matlab/...)

4.11 Activating hyperlinks in PDF files

Example 4.11.1: PDF hyperlinks

```
\definecolor{linkcolor}{rgb}{0, 0, 0}

\hypersetup{
    unicode=true,           % non-Latin characters in AcrobatâĂŹs bookmarks
    pdftoolbar=true,        % show AcrobatâĂŹs toolbar?
    pdfmenubar=true,        % show AcrobatâĂŹs menu?
    plainpages,
    pdffitwindow=false,     % window fit to page when opened
    pdfstartview={FitH},    % fits the width of the page to the window
    pdftitle={TITLE},       % title (appears in Acrobat's title bar)
    pdfauthor={AUTHOR},     % author
    colorlinks=true,
    linkcolor=linkcolor,
    citecolor=linkcolor,
    filecolor=linkcolor,
    urlcolor=linkcolor
}
```

4.12 Scaling boxes with `scalebox`

5 Graphics

Adding graphics to a LaTeX document is not always straightforward. In this section, we will talk about two different ways of adding high-quality graphics to your document. By 'high-quality' I mean the following. There are two fundamentally different types of images: 'bitmap' images and 'vector' images. Bitmap images are basically stored as a matrix in which the entries correspond to the color of each pixel in your image. Bitmap images are very useful for photographs, but not so much for diagrams. In contrast, vector graphics are stored by the 'objects' contained in them. For example, it would contain entries like 'draw a line here ..' and 'draw a circle there'. The main advantage of vector graphics is that they remaing good-looking when scaled. This is not true for bitmap images: when you zoom in on a bitmap, the pixels will become visible and this looks quite ugly. Therefore, for diagrams, it is better to use vector graphics.

In this section we will first talk about a way of making vector graphics the 'what-you-see-is-what-you-get' way, using a program called Inkscape. The second part of the section talks about a programmer's way of creating graphics, using PGF and TikZ.

5.1 Inkscape

5.2 TikZ

Installing PGF and TikZ

Creating a TiKZ figure TikZ figures are usually surrounded by the `\begin{tikzpicture}` and `\end{tikzpicture}` commands. Between these commands the TikZ language is in effect. The

TikZ language is relatively straightforward. For example:

Example 5.2.1: My first TikZ code

```
\begin{tikzpicture}
\draw (-1.5,0) -- (1.5,0);
\draw (0,-1.5) -- (0,1.5);
\draw (0,0) circle (1);
\end{tikzpicture}
```

This has the following output:

In this example, we give three command to TikZ: each command consists of the `\draw` command, followed by instructions on what to draw, and ended by a semicolon. For example, in each of the first two lines, we issue a command to draw a line between two points. In the third line, we draw a circle with center $(0, 0)$ and radius 1.

We can also change properties (i.e. line width, and rounding) of the objects to be drawn by adding some arguments to the `\draw` command. For example:

Example 5.2.2: Rounded corners

```
\begin{tikzpicture}
\draw[thick,rounded corners=8pt] (0,0) -- (0,2) -- (1,3.25)
-- (2,2) -- (2,0) -- (0,2) -- (2,2) -- (0,0) -- (2,0);
\end{tikzpicture}
```

This has the following output:

The possibilities with TikZ are practically endless, and it takes a 560-page manual to go through all of it. See:

http://www.ctan.org/tex-archive/graphics/pgf/base/doc/generic/pgf/pgfmanual.pdf.

6 Examples

6.1 Creating nice looking boxes

Next, let us look at how I created the example boxes in this document:

Example 6.1.1: Example boxes

```
\tikzstyle{nicebox}=[draw=gray!100, fill=blue!10, very thick,
    rounded corners, rectangle, inner sep=4pt, inner ysep=16pt]
\tikzstyle{niceboxtitle}=[draw=gray!100, fill=white, text=black,
    rounded corners, very thick, rectangle]
\newcommand\nicebox[2]{
   {\centering
   \begin{tikzpicture}
      \node [nicebox](box){
         \begin{minipage}{0.95\textwidth}\centering
         \begin{minipage}{0.95\textwidth}
                        #2
         \end{minipage}\end{minipage}};
      \node[niceboxtitle, right=10pt] at (box.north west)
         {\small\textbf{#1}};
   \end{tikzpicture}\par}
}
```

This example creates a new command named `\nicebox`. It can be used for example as follows: `\nicebox{Title}{This is the content of my box}`, which results in the following box:

Title

This is the content of my box

6.2 Putting parenthesis around references to equations

A useful command I like to define is `eqnref`, which makes just a reference to an equation, but puts the parentheses around the equation number:

Example 6.2.1

```
\newcommand\eqnref[1]{(\ref{#1})}
```

7 Getting help

Getting help on LaTeX is relatively easy due to the fact that it has a huge userbase. The most important thing to remember is the following: whenever you encounter a problem or difficulty, with high probability some other person on the internet has encountered it. Is there something you want to implement in your document but you can't figure out how? Again, almost surely someone else has thought about the same problem and thought of a solution. And if you're lucky, there is even a package available for doing whatever you want to do. Google is the answer to almost all your (LaTeX) questions.

7.1 Package manuals

Besides particular problems, you may want to find out how to work with a specific package. Packages may have many options to customize its behavior. A good way to find the manual of a package, say `quotchap`, is to just Google for the name of the package and the word 'package'. So for example, to get the manual for `quotchap`, I Google for 'quotchap package'. The first result I get today is the user manual in PDF format.

Most package manuals have approximately the same structure. They start with an introduction describing what the package does. Then it contains documentation on what options the package has, and (hopefully) some examples on how to use it. Finally, there is usually an Implementation section, which describes the internal workings of the package.

I think that reading the manual is generally not necessary: just try to pick what you need. For example, I usually ignore the implementation part, unless I really have to. Sections on options and examples tend to be the most useful.