Some tips for LaTeX, Matlab, and C

Jeffrey A. Fessler
EECS Department
The University of Michigan

October 9, 2003

1 Introduction

Over the years I've read (and written) many LaTeX documents, Matlab scripts, and C programs, and I've noticed that beginning users frequently do not fully take advantage of some of the powerful but sometimes more subtle features of these programming languages. This paper is a collection of tips. They are not exactly "frequently asked questions" but more like "frequently made suggestions." Take 'em or leave 'em...

This document originated in `~/fessler/l/tex/tr/tips`.

Future updated versions may become available in my technical report collection at
`http://www.eecs.umich.edu/~fessler/papers/t,tr.htm`

2 LaTeX

2.1 `newcommand`

One cannot overstate the utility of LaTeX's `\newcommand` command.

Here is a basic LaTeX example that fails to fully illustrate the benefits of `newcommands`.

```
If $\lambda$ is positive, then
\( 5 \lambda > 4 \lambda .\)
```

If λ is positive, then $5\lambda > 4\lambda$.

Liberal use of `newcommand` has several advantages:
- increases flexibility (easier to make changes to fonts and notation later)
- saves typing (saving time and tendons),
- can improve readability,
- allows one command to have multiple effects, *e.g.*, putting a term both in italics for emphasis and adding it to the index.

Here is the above example "simplified" by using `newcommand`.

```
\newcommand{\lam}{\lambda}
If $\lam$ is positive, then
\( 5 \lam > 4 \lam .\)
```

If λ is positive, then $5\lambda > 4\lambda$.

2.2 `xspace` and `ensuremath`

In the preceding example, if λ will be used frequently in text, then it would be nice not to have to type the $ signs around `\lam` each time. The `ensuremath` command has the effect of providing those $ signs when needed, so the preceding example is improved as follows.

```
\newcommand{\lam}{\ensuremath{\lambda}}
If \lam\ is positive, then
\( 5 \lam > 4 \lam .\)
```
If λ is positive, then $5\lambda > 4\lambda$.

In this example, we had to add the extra backslash after `\lam`, since otherwise the space following `lam` would have been interpreted by LaTeX as the "end of command" character, rather than as a space, and we would get "If $\lambda$is positive." The `xspace` package and command solves this problem, thereby eliminating the need for any of those extra backslashes throughout the body of the document.

The final example below is the best way of all. This must be preceded by `\usepackage{xspace}` in the document preamble.

```
\newcommand{\xmath}[1]{\ensuremath{#1}\xspace}
\newcommand{\lam}{\xmath{\lambda}}
If \lam is positive, then
\( 5 \lam > 4 \lam .\)
```
If λ is positive, then $5\lambda > 4\lambda$.

I use this `xmath` macro extensively in my preamble, and I will use it in the examples hereafter.

Since the old version of LaTeX did not have a `ensuremath` command, the original LaTeX manual [1, p. 55] suggested using `mbox` as a "trick" to provide the convenience of using the same macro in text or equations. Unfortunately, as the follow example shows, the `mbox` approach does not work properly in subscripts, whereas `ensuremath` does.

```
\newcommand{\p}{\xmath{p}}
\newcommand{\q}{\mbox{$q$}}
Here are \p and \q\ in text, but
\( \min_{\p} \neq \min_{\q} .\)
```
Here are p and q in text, but $\min_p \neq \min_{q}$.

The second edition of the LaTeX manual recommends `ensuremath` rather than `mbox`; presumably the above problem motivated the development of the `ensuremath` command.

2.3 `mathbf` vs `bm`

Bold symbols appear frequently, and are an excellent reason to use `newcommand`. Traditional LaTeX has some quirks about bold though. If you liked the above, then you would probably be disappointed by the following example.

```
\newcommand{\x}{\xmath{\mathbf{x}}}
\newcommand{\lam}{\xmath{\mathbf{\lambda}}}
\newcommand{\thet}{\mbox{\boldmath $\theta$}\xspace}
When \x, \lam, and \thet are vectors, then
\( 0 = \min_{\lam,\x,\thet} \| \x - \lam - \thet \|. \)
```
When $\mathbf{x}$, λ, and $\boldsymbol{\theta}$ are vectors, then $0 = \min_{\lambda,\mathbf{x},\boldsymbol{\theta}} \|\mathbf{x} - \lambda - \boldsymbol{\theta}\|$.

There are two annoying problems here. First, notice that λ was not made bold by the `mathbf` command; instead one needs the `boldmath` command, and who can remember the difference between `mathbf` and `boldmath`? Secondly, notice that the θ in the subscript to `min` is not of subscript size. Not very professional looking.

Fortunately, the `bm` package takes care of all of these, and saves typing too!
Just put `\usepackage{bm}` in your preamble, and the preceding example simplifies to the following.

```
\newcommand{\bmath}[1]{\ensuremath{\bm{#1}}\xspace}
\newcommand{\x}{\bmath{x}}
\newcommand{\lam}{\bmath{\lambda}}
\newcommand{\thet}{\bmath{\theta}}
When \x, \lam, and \thet are vectors, then
\( 0 = \min_{\lam,\x,\thet} \| \x - \lam - \thet \|. \)
```

When $\boldsymbol{x}$, $\boldsymbol{\lambda}$, and $\boldsymbol{\theta}$ are vectors, then $0 = \min_{\boldsymbol{\lambda},\boldsymbol{x},\boldsymbol{\theta}} \|\boldsymbol{x} - \boldsymbol{\lambda} - \boldsymbol{\theta}\|$.

Note that this handy `bmath` macro takes care of three things: math mode, bold, and the space.

2.4 Equation references

Maybe I'm just absent minded, but when writing long documents it is easy to forget equation labels. The following definition of `\ee` displays the label in the margin in the document (for drafts).

```
\newcommand{\pbox}[1]   {%
\makebox[0pt][r]{\raisebox{7mm}[0pt][0pt]{\small #1}}\ignorespaces}
\newcommand{\be} {\begin{equation}}
\newcommand{\ee}[1] {\label{#1}\end{equation}\pbox{#1}}
\newcommand{\eref}[1] {(\ref{#1})}
\be
E = mcˆ2
\ee{e,albert}
You have probably seen \eref{e,albert} before.
```

e,albert

$$E = mc^2 \tag{1}$$

You have probably seen (1) before.

Note that `\eref{}` takes less typing than `(\ref{})` and also helps make it clear in the source that this is an equation reference.

Combining `\label` and `\end{equation}` into one command ensures that I always remember to put a label for any numbered equation. (One should only number equations that are referenced in the document, or plausible to be referenced by future papers.)

Of course I don't want others to know of my forgetfulness, so for the final version I use
`\newcommand{\ee}[1]{\label{#1}\end{equation}}`

Similar tricks work for tables and figures.

A key point here is that using macros allows one to "overload" one command to have multiple effects, which can be very useful (in this case, defining a label, ending an equation, and displaying the label).

The above `ee` macro may conflict with a macro provided in one of the AMS packages. If so, you can use `renewcommand` instead of `newcommand` to force your definition of the macro. Another useful macro command is `providecommand` which only defines the macro if it is not already defined.

The `showkeys` package also shows such labels in the margins, and more! It is very very useful! (Many thanks to Markus Fenn for bringing this to my attention!)

2.5 Other nice packages

- Use `\usepackage{cite}` so that citations come out [1,2,3] instead of [1][2][3].
- Use `\usepackage{times}` to have nicer looking Times Roman fonts that convert well to PDF.
- Use `\usepackage{psfrag}` so that you can replace text fragments in included EPS figures with LaTeX stuff, using the following syntax: `\psfrag{hattheta}{$\hat{\theta}$}`, where

`hattheta` is a text string you might have used in a matlab plot axis, title, or `text` command, or an `xfig` text string.

- Use `\usepackage{color}` to add color to documents. (See `~/fessler/l/tex/talk` for examples.)
- Use `\usepackage{ifthen}` to add conditional statements.
- Use `\usepackage{amsymb}` to get special symbols like the following.

```
\newcommand{\reals}{\xmath{\mathbb{R}}}
This is the cool way to make \reals.
```

This is the cool way to make $\mathbb{R}$.

- For other examples of general macros, see `~/fessler/l/tex/def,gen`.
  For other examples of math macros, and `~/fessler/l/tex/def,math`.

2.6 PDF conversion

The *de facto* format for sharing documents has become Adobe's PDF format. (In the recent past, I have emailed postscript files to people crippled by too many years of using Windoze, and they email me back saying that they can't read the gibberish I've sent them.) The default fonts used by LaTeX are *bitmapped* fonts. These look awful when magnified in a PDF viewer, and they are also often designed for a given "resolution" (say, 300dpi) so they are suboptimal for the 600dpi and 1200dpi printers around EECS.

PDF and other modern display methods used *outline fonts*, the standard for which is Adobe's "Type 1" fonts.

To use these fonts you should do the following.

- Add `\usepackage{times}` to the start of your .tex file. This gives you Times Roman fonts instead of LaTeX's default fonts, which I think look better overall anyway.
- Convert your `.dvi` files to `.ps` files and your `.ps` to `.pdf` files using commands like the following:

```
dvips -Ppdf -G0 -K -o file.ps -t letter file.dvi

ps2pdfwr -dCompatibility=1.3 -dMaxSubsetPct=100 -dSubsetFonts=true \
-dEmbedAllFonts=true file.ps file.pdf
```

- That should be all you need! But just in case that does not work, here is the OLD way I used to do it, included for historical completeness. Now the cmfonts are included with latex, so it should not be necessary to do all this.
- Add something like the following to your `.cshrc` file.

```
setenv TEXCONFIG /usr/gnu/tex/dvips:/n/ir7/home/fessler/l/tex/font/cmtype1
setenv DVIPSHEADERS $TEXCONFIG
```

  Replace `/n/ir7/home/fessler` with whatever directory `~fessler` is from your host. My home directory moves more often than this document is updated!
  My `tex/font/cmfonts` directory has the outline descriptions of the LaTeX fonts. You need this because the `times` package only replaces the text, not the mathematics with outline fonts. (There are probably packages (`mathtime`?) (`mathptm`?) that can also replace the math.)
  These fonts came from `http://www.ams.org/tex/type1-cm-fonts.html`, if I recall correctly, or from the CTAN web site.
- Type `source .cshrc` so the new variables are set.
- Run LaTeX (and rerun it as necessary...)
- Convert your DVI file to a PS file as follows:

```
dvips -Pcmfonts -K filename.dvi
```

  This should use the outline fonts in place of any residual Computer Modern fonts (the LaTeX default) in your document.

- Run `distill` (we have it on ir2 and DCO has it on another machine - dip? quip?) to convert the PS file to a PDF file: distill filename.ps
  Or you can try `ps2pdf` on your machine, but, when I last tried it, it did make good equation fonts.
  Another option may be the shareware program `PStill` available from
  `http://www.wizards.de/~frank/pstill.html`
  That web page also has several links about LaTeX to PDF conversion.

You can also make PDF files directly from LaTeX by the command `pdflatex`, but I do not know if it automatically used outline fonts.

2.7 `bibtex` and `.bib` and `.b2` files

The cross referencing abilities of LaTeX are one of its strengths, particularly when combined with `bibtex`. A `bibtex` entry in a `.bib` file looks like the following.

```
@ARTICLE{
ramachandran:71:tdr,
author = {G N Ramachandran and A V Lakshminarayanan},
title = {Three-dimensional reconstruction from radiographs...},
journal = {Proc. Natl. Acad. Sci.},
volume = 68,
number = 9,
pages = {2236--40},
month = sep,
year = 1971
}
```

I have over 4000 `bibtex` entries in my database, and back when I reached about 400 entries I grew very weary of typing `author`, `title`, etc., and even of cutting and pasting those.

I developed my own much more concise format. The above citation is stored as follows.

```
@a ramachandran:71:tdr pnas 68 9 2236-40 sep 1971
G N Ramachandran  A V Lakshminarayanan
Three-dimensional reconstruction from radiographs...
```

There are almost no redundant characters here. The `@a` signifies a journal article.

Plain text database files are in `~/fessler/l/tex/biblio/b2`, organized by topic, each corresponding to a folder of papers.

A `Makefile` in that directory converts those files to the usual `bibtex` format, concatenating them all into `~/fessler/l/tex/biblio/master.bib`. The conversion is by the Perl script `~/fessler/l/tex/src/script,bib/b2,bib`.

Another Perl script `~/fessler/l/src/script,bib/b1,grep`. lets me extract all records containing a given keyword. This is the script that I use when trying to find a given paper.

Recommendation: put my `master.bib` in your `BIBINPUTS` path, so that you can cite any of the papers in my database *without retyping them*. On multiple occasions I have proofread papers that have had typos in the references, including to my own papers! This wastes both our time: yours in typing, and mine in proofreading. The majority of the papers you will cite are already in my database (and probably I'd like to read the new papers you find too!). To set up the path, use something like the following.

```
setenv FESS /n/ir7/home/fessler/l/tex
setenv BIBINPUTS :$FESS/biblio:$FESS/macros
```

Then to use this in your LaTeX document, simply add the following near the end.

```
\bibliographystyle{unsrt}
\bibliography{master}
```

You can also try the `IEEE.bst` bibliographystyle that is in my `macros` directory.

You are welcome to copy any of my bibliographic material to your own directory if you want your own copy. I would much rather have you copy and paste it into your own organizational style than see you retype it and get tendonitis (like me) and introduce new errors.

I am told that the commercial product `endnote` can download from various abstract databases and format references into the `bibtex` format.

2.8 `def.raw` `def.glo` files

Just like I forget my equation numbers, in long documents I can also forget my macros (`newcommands`). So, these days I just create two files, one called `def.raw` that looks like this example:

```
lam \ensuremath{\lambda}\xspace
x \ensuremath{\bm{x}}\xspace
```

(etc.) and another called `def.glo` that looks like this example:

```
y \ensuremath{\bm{y}\xspace}
a measurement vector

thet \ensuremath{\theta}\xspace
an angle in degrees
```

(etc.) Then I run the following Perl script

```
                  ~fessler/l/src/script/tex,def > def.tex
```

which converts these two files into a file `def.tex` that looks like the following

```
\newcommand{\lam} {\ensuremath{\lambda}\xspace}
\newcommand{\x} {\ensuremath{\bm{x}}\xspace}

\newcommand{\y} {\ensuremath{\bm{y}\xspace}}
\newcommand{\thet} {\ensuremath{\theta}\xspace}

\providecommand{\defitem}[3]{\ty{#1} & #2 & #3\\}
\providecommand{\defstart}{
\begin{tabular}{l|l|l}
\defitem{Macro}{Symbol}{Meaning}}
\providecommand{\defstop}{\end{tabular}}
\providecommand{\defglo}{
\defstart
\defitem{y}{\y}{a measurement vector}
\defitem{thet}{\thet}{an angle in degrees}
\defstop
}
```

This script does two things. First, it puts all the `newcommands` in, saving me from having to type them. Second, it automatically builds a command `defglo` that prints a "glossary of definitions" in my document. I'll illustrate it right here by typing `\input{def.tex}\defglo` right at the end of this sentence.

Macro	Symbol	Meaning
`y`	$\boldsymbol{y}$	a measurement vector
`thet`	θ	an angle in degrees

Admittedly I didn't start doing this until I starting writing my book (and realized that a symbol glossary would be useful). But since then I've found it useful.

3 Matlab

Ever since version 5 or so, Matlab has been an *object based* programming language. This means that operators (like +) can be *overloaded* to have generalized meanings depending on the type of object being operated on.

Here is a simple example. In conventional matlab, the expression `'path/' + 'file'` would be an error: you cannot "add" strings. But to me, a logical meaning for addition of strings would be to append them. Here's how to do it. In my `matlab/mfiles` directory, I have a subdirectory named `@char`. This name is because the class name of strings in matlab is `char`. Within this directory I have the file `plus.m` which is the following:

```
function c = plus(a,b)
c = [a b];
```

By creating this subdirectory and file, and by having that subdirectory in my matlab path, I have *overloaded* the "plus" (+) operator in matlab. When I type `name = 'path/' + 'file'`, matlab notices that the object on the left of the + is of class `char`, and it searches through its own rules of addition for strings and, finding none, it searches through my `@char` subdirectory and finding the `plus.m` file it essentially calls `name = plus('path/', 'file')` and the actually concatenation is done by the commands in the `plus.m` function. (Actually I don't know which order matlab searches for the overloaded functions.)

Ok, you might not be too impressed with the time savings in the above example, since I could have just typed `name = ['path/' 'file']` in the first place, which has the same number of keystrokes and is (almost) just as readable.

For my work, the real benefit of operator overloading is for *system matrices* and *iterative algorithms*. Consider the simple iterative algorithm

$$\boldsymbol{x}^{n+1} = \boldsymbol{x}^n + \alpha \boldsymbol{A}'(\boldsymbol{y} - \boldsymbol{A}\boldsymbol{x}^n).$$

If $\boldsymbol{A}$ is a matrix in matlab (sparse or full), then this algorithm translates very nicely into matlab as follows.

```
x = x + alpha * A' * (y - A * x);
```

You really cannot get any closer connection between the math and the program than this! But these days we are working a lot with system models that are too big to store as matrices in matlab, and instead are represented by subroutines that compute the "forward projection" action $\boldsymbol{A}\boldsymbol{x}$ and the "backprojection action $\boldsymbol{A}'\boldsymbol{z}$ for input vectors $\boldsymbol{x}$ and $\boldsymbol{z}$ respectively. The conventional way to use one of these systems in matlab would be to replace the above program as follows.

```
Ax = forward_project(x, system_arguments)
residual = y - Ax;
correction = back_project(residual, system_arguments)
x = x + alpha * correction
```

Yuch! This is displeasing for two reasons. First, the code looks a *lot* less like the mathematics. Second, you need a different version of the code for every different system model (forward/back-projector pair) that you develop.

The elegant solution is to develop matlab objects that know how to perform the actions
- `A * x` (matrix vector multiplication, operation `mtimes`)
- `A'` (`transpose`), and
- `A' * z` (`mtimes` again, with a transposed object).

Once such an object is defined, one can use *exactly* the same iterative algorithm as before. The details are too involved to be typeset. For examples, see `~/fessler/l/src/matlab/alg/systems`.

3.1 Sparse matrices

Matlab stores sparse matrices by *columns*, so column-based operations can be much faster than row-based operations. If you need many row-based operations, then it is best to first form the transpose of the sparse matrix, then access the columns of the transposed matrix (which are the rows of the original matrix). This is particularly relevant for *ordered subsets* algorithms.

4 C

Over the years, I have found that liberally using the C preprocessor `#define` command to define macros has many of the same advantages of LaTeX's `newcommand`:
- increases flexibility (easier to make global changes to routines later)
- saves typing (saving time and tendons),
- can improve readability,
- allows one command to have multiple effects, *e.g.*, both calling a subroutine and printing a debugging message during development.

These benefits are illustrated below with a memory allocation example.

4.1 Memory allocation

Most C programs written by novices have line like the following.

```
float *ptr;
ptr = (float *) calloc(n, sizeof(*ptr));
```

Sooner or later this code will crash when `n` is too large or nonpositive, and the novice will learn that it is worth the effort to put in error checking like the following.

```
float *ptr;
if ( !(ptr = (float *) calloc(n, sizeof(*ptr))) ) {
        fprintf(stderr, "could not allocate ptr at %d of %s\n",
                __LINE__, __FILE__);
        exit(-1);
}
```

There are several disadvantages of this. First, it is annoyingly long to type. Second, it is not that easy to read, since the most important parts (`ptr` and `n`) are buried in a morass of other largely superfluous characters (`sizeof` etc.). Third, exiting is better than crashing, but does not help debugging much since there is no "traceback" to the offending calling routine that presumably asked for too much memory.

To overcome some these disadvantages, consider the following macrofied code.

```
#define Mem(p,n)        { \
        if ( !((p) = (void *) calloc((unsigned) (n), sizeof(*(p)))) ) { \
                fprintf(stderr, "could not allocate ptr at %d of %s\n", \
                        __LINE__, __FILE__); \
                exit(-1); \
        } \
}

float *ptr1, *ptr2;
Mem(ptr1, n)
Mem(ptr2, 2*n)
```

This code is easier to read, and has the advantage that the `Mem` macro is easily changed to include debugging messages showing how many bytes are being allocated and where.

My own code takes this several steps further.

- I make any routine that includes memory allocation of type `bool` (aka `int`) and then use a macro `Mem0` that returns 0 (failure) rather than exiting to the calling routine. This enables an automatic traceback that simplifies debugging.
- In debugging mode (which I use all the time), my `Mem0` macro actually calls my own subroutine `io_mem_alloc` that maintains a list of all allocated memory. Similarly, I have a `Free0` macro that calls my subroutine `io_mem_free` that removes freed items from the list. Then, before my program exits, I check that all allocated memory was freed (this helps find memory leaks), and I print the maximum amount of allocated memory (this is useful for keeping track of the memory requirements of various algorithms).

If you want to take advantage of this in your own routines, do the following.

- Add `#include "def.h"` to your C programs.
- Add the following arguments to your compile flags in your `Makefile`
  `-I/n/ir7/fessler/l/src/give/code -DCountAlloc`
  (That directory contains `def.h` which defines the `Mem0` and `Free0` macros.
- Copy `~/fessler/l/src/io/univ/io,alloc.c` to your directory and edit the line `#include "defs-env.h"` to be `#include "def.h"` instead.
- Replace all `calloc`s and `malloc`s in your code with `Mem0`, and all `free`s with `Free0`, making sure the subroutines they are in are of type `bool` or `int`.
- Add the statement `PrintAlloc` somewhere (*e.g.*, near the end) of your `main` program, which will cause memory usage to be printed.

In my experience, a small investment in macrofying code when writing it more than pays off when debugging it, though it took me several years of debugging to realize this.

4.2 Other macros

I use similar (but simpler) macros for nearly all other system calls, *e.g.*, `fopen` becomes `Fopen0`, etc. See `~/fessler/l/src/defs/def,*.h` for examples. These macros both reduce typing and facilitate debugging.

References

[1] L. Lamport. *LaTeX: A document preparation system*. Addison-Wesley, New York, 1986.