TEXing in Emacs

Marcin Borkowski

Abstract

In this paper I describe how I use GNU Emacs to work with LaTeX. It is not a comprehensive survey of what can be done, but rather a subjective story about my personal usage.

In 2017, I gave a presentation [1] during the joint GUST/TUG conference at Bachotek. I talked about my experiences typesetting a journal (*Wiadomości Matematyczne*, a journal of the Polish Mathematical Society), and how I utilized LaTeX and GNU Emacs in my workflow. After submitting my paper to the proceedings issue of *TUGboat*, Karl Berry asked me whether I'd like to prepare a paper about using Emacs with LaTeX.

Well, I jumped at the proposal. I am a great fan of Emacs, and I've been using it for nearly two decades now. So, here is my Emacs/TeX story.

I decided to divide this tale in four parts. The zeroth one is a very brief explanation of how I got where I am with respect to Emacs. The first one is a very short introduction to the main concepts and terminology of Emacs. Then, I talk about various Emacs packages I use in my day-to-day (La)TeX work. Finally, for the brave souls who would like to go deeper into the Emacs rabbit hole, I present a few example snippets I wrote to make Emacs suit my personal needs.

0 The beginnings

GNU Emacs is an ancient piece of software, started by the famous Richard M. Stallman, and used right through today. (Interestingly, one of the reasons Stallman started the GNU project, which GNU Emacs soon became part of, was ethical rather than technical. He has a very distinct set of moral beliefs, one of which is that everyone should have certain freedoms with respect to the software they use, and sticks to them *without compromise*. Disclaimer: while I share some, but not all of his views, I still admire his perseverance and his strong conviction about the objectivity of moral principles — even if he gets some of these wrong.)

When I started using Emacs (note: I will not say "GNU Emacs" each time; nowadays, there are essentially no other Emacsen, since the last "competitor", XEmacs, seems to have died a few years ago), I needed just a text editor for TeX (I was using plain TeX at the time). When I switched from DOS and MS Windows 3.11 to a GNU/Linux system, I heard that there are two editors, and had to choose one of them. I used a simple criterion: Emacs had a nice tutorial, and Vim apparently did not (at that time).

I wince at the very thought I might have chosen wrong!

And so it went. I started with reading the manual [8]. As a student, I had *a lot* of free time on my hands, so I basically read most of it. (I still recommend that to people who want to use Emacs seriously.) I noticed that Emacs had a nice TeX mode built-in, but also remembered from one of the BachoTeXs that other people had put together something called AUCTeX, which was a TeX-mode on steroids.

In the previous paragraph, I mentioned *modes*. In order to understand what an Emacs mode is, let me explain what this whole Emacs thing is about.

1 Basics of Emacs

It is actually easy to understand Emacs. (Do not confuse "understand" with "master", by the way.) There are many ways of explaining the phenomenon of a piece of software which has existed for about four decades and is still relevant today. A popular view (and a subject of jokes) is that Emacs is an operating system disguised as an editor. This is surprisingly close to the truth, but this metaphor does not help with talking about how Emacs interacts with (La)TeX. Another known aphorism is that Emacs is not a text editor, but a DIY kit for creating *your* own editor, suited to your needs. This is also true, and I will come back to it later. However, I think that in order to explain Emacs, I should describe the core concepts which make it what it is. For those, I choose three basic Emacs notions of *buffers*, *commands* and *keybindings*, and a fourth one which, well, *binds* them together: modes.

Just as in Unix "everything is a file", in Emacs "everything is a buffer". Interestingly, an Emacs buffer is an entity which is quite close to a Unix file. It is identified by name, and it consists of characters. (There's more to it than that, but let's keep things simple.) If you visit a file in Emacs (this is what most other editors call "opening" a file), a buffer with a name corresponding to the name of the file is created and filled with the characters read from that file. From now on, Emacs does not care about a physical file on the disk (or somewhere else — Emacs can also open files located "in the cloud", i.e., on other people's computers), at least not until we want to save the buffer to the file again.

In most (La)TeX editors (and most other applications, for that matter), we have things such as dialog windows, non-editable text areas with the compilation log, file-selecting widgets, etc. Not in Emacs:

here, all these are buffers (some of them read-only, of course). Here is an example: if you press `C-x d RET` (in Emacs parlance, this means pressing `Control-X`, then `D` and then `Enter`), you will see a Dired *buffer*, which contains a listing of files in the current directory, much like the output of the Unix `ls` command. It is not normally editable, but you can use various keybindings to move *point* (i.e., the cursor), and perform various actions on the listed files, like visiting them (`f` or `RET`), copying them somewhere else (`C`, i.e., `Shift-C`), diffing them with other files (`=`), etc.

Let us now talk about *commands*. They are pieces of code (usually in Emacs Lisp, or Elisp, the language the majority of Emacs is written in) which perform various tasks you would expect from a text editor (and more). For instance, there is a command called `find-file`, which asks the user for a file name and visits it in an Emacs buffer. Another, called `save-buffer`, saves the contents of the current buffer to a file. Yet another, called `pong`, is an implementation of the old arcade classic.

It is important to understand that basically *any* action you perform in Emacs is a result of running some command. If you press some key (or key combination), Emacs checks the *binding* of that key, which says what command is *bound* to that key, and runs that command. This works even for very basic things, like the command `forward-char` (usually bound to `C-f` and `<right>`, that is, the right-arrow key), or keys with printable characters, which are usually bound to `self-insert-command`. A command need not be bound to a key — you can also call it by its name (for instance, the `pong` command is usually run by `M-x pong`, which means pressing `Alt-X`, then typing `pong` — the name of the command — then pressing `Enter`).

The last piece of this puzzle are *modes*, which tell Emacs the bindings of keys to commands in any particular buffer. (Each buffer has its associated mode, so switching buffers changes keybindings dynamically. Of course, there are also *global* bindings, which work everywhere, like `C-x C-c`, which exits Emacs.) For instance, in TEX mode, `C-c C-c` is bound to a command which does the "next logical thing", like compiling the file, running BIBTEX or launching a PDF viewer. The same key combination, `C-c C-c`, sends a message in a mode designed to write emails.

Finally, any introduction like this one would be incomplete without mentioning the self-documenting nature of Emacs. If you are in a buffer with some Emacs Lisp code, you can press `C-h f` when the point is on a function name, or `C-h v` when it is on a variable name, and immediately see the docstring for that function or variable, or even jump to the place in the source where it is defined. This is extremely useful, and there are specialized Emacs packages which streamline this even further (like showing the docstring in a tool-tip-like fashion, or finding all places where a function is called). Also, when you are in the middle of a function expression, Emacs shows the list of parameters of the current function at the bottom, with the parameter the point is on set in boldface. Finally, there is the whole set of *apropos* commands, which show all functions or variables matching the given regex — and "matching" can mean matching the name, the docstring, or even the value in case of variables. At any time, you can press `C-h C-h` to see the extensive list of Emacs help subsystems.

2 Emacs packages and features for TEXnicians

In this section, I am going to describe a few well-known (at least in the Emacs world) packages and features Emacs has to offer for people dealing with (LA)TEX. Note that this is not a comprehensive list — these are just the ones I happen to use.

2.1 AUCTEX

Although Emacs has a TEX mode built-in (and there are people using it), it is rather bare-bones. Happily, there is AUCTEX, a well-known package which is used by the majority of Emacsers who need to do stuff in TEX and friends.

Of course, AUCTEX has all the things you would expect, like auto-completing macro and environment names (it even knows the syntax of many LATEX commands and asks for the parameters, and parses `\newcommand`s to insert the proper number of braces after user-defined macros), commands to compile the file we are editing or syntax coloring (called "font-lock" in the Emacs world). It also has, however, some features which I believe are unique to it (although I admit that I have not used other editors extensively). AUCTEX has a very good manual (as do many Emacs packages — documenting things for users is emphasized a lot in the Emacs world), so let me just mention my personal favorite feature. You can select a portion of text and press `C-c C-r`. AUCTEX then writes a temporary file consisting of the preamble, the selection and `\end{document}`, compiles it and prepares the next viewing command to show this PDF instead of the whole file. Imagine a Beamer presentation with more than a thousand pages and a lot of drawings, which compiles for a few minutes in its entirety (true story!), and you can see what a life-saver this can be.

There is a lot more to AUCTEX than this. Let me mention LATEX-Math mode, which makes the key combinations starting with a backtick (`) insert many mathematical commands, so that you can get `\alpha` by pressing `a or `\emptyset` by pressing `0. Another feature of Emacs, which AUCTEX utilizes, is `prettify-symbols-mode`, which displays things like `\alpha` or `\int` using Unicode characters, which renders the source of math formulas much easier to read. (While at that, let me mention that despite its age, Emacs has excellent Unicode support, but can also handle some other, nowadays less popular encodings.) Yet another thing AUCTEX leverages is Emacs' `compilation-mode`, which parses the log buffer and allows you to, for instance, jump to the next or previous error. Basically, you have all you would expect from a good TEX editor. (One caveat is that support for plain TEX and ConTEXt is rather rudimentary.)

There is also a package called RefTEX, which is part of AUCTEX, whose goal is to help with all the references. One of its coolest features is showing the bibliographic info about a reference when the point is on the `\cite` macro. Unfortunately, RefTEX does not work with AMSrefs, and hence I do not use it personally (yet).

2.2 pdf-tools

There has been a PDF viewer in Emacs for a long time, but it never worked too well: under the hood, it just converted PDFs to bitmaps using Ghostscript and displayed them. Not very impressive, and very slow. Recently, however, another PDF viewer for Emacs was written, called pdf-tools. It is a wonderful piece of software, and even though it has its quirks (it does not have "spread" or "continuous" modes, for instance), it almost completely replaced Evince for me. It supports the SyncTEX extensions, thus allowing for jumping between the source and the PDF with very good accuracy, and allows for incremental search in a PDF (also with regexen). It also supports an Emacs concept called *occur*, which asks the user for a regex and displays a list of lines in the buffer matching that regex, allowing to jump immediately to any of these lines. You can also make Emacs watch for changes in the PDF and refresh it automatically, or issue a refresh when the compilation is finished. Perhaps the killer feature of pdf-tools, however, is its support for PDF annotations: it is possible to view all annotations without moving your hands from the keyboard.

This is also a good place to mention one of the many ways Emacs is flexible. In many places in the Emacs source code there are so-called *hooks*. They are variables, usually empty by default, which contain functions to be run at various moments. The pdf-tools package has a hook which contains functions run each time an annotation is shown. You can, for instance, add a function which checks whether the annotation is written in LATEX syntax, and if yes, call LATEX and then `dvipng` to display a picture of the typeset formula of something instead of the text of the annotation. In fact, a function doing exactly this is provided as an example, so I can actually format my PDF annotations in LATEX and pdf-tools just displays them correctly! There are about 150 hooks in stock Emacs, and many packages add their own — my Emacs has more than 700.

2.3 Other useful Emacs features

What makes Emacs an even better (LA)TEX editor are its features as a general text editor, applied to the particular case of (LA)TEX. I have already mentioned `compilation-mode` and `prettify-symbols-mode`; there is also a spell-checker (delegating its job to external tools, but the integration is seamless) and `auto-fill-mode`, which automatically wraps long lines using hard newlines (which is an abomination to many and a no-brainer for others). Emacs, however, is used by many people to edit texts in human languages (as opposed to computer programs), and has really good support for that task. For instance, while many editors have support for movement by words, Emacs has also movement by sentences. Another feature which is a real time-saver is the series of `transpose` commands: for instance, `transpose-chars` swaps the two characters on both sides of the point. There is a whole chapter in the Emacs manual describing commands for dealing with text in human languages.

Another Emacs tool which is used by many people is Yasnippet. It is a very useful tool for creating and inserting *snippets*, i.e., templates with placeholders for variable fragments. It is very easy to define one's own snippets. (Personally, I do not use Yasnippet with AUCTEX very often, since the latter can insert a lot of things for me, but I have snippets for, e.g., preambles of some documents.)

Last but not least, let me mention the Avy package. It solves the problem of quickly navigating to any place on the screen without using any pointing device (many Emacs users have an aversion to rodents and prefer using their keyboards as much as possible). The classical Emacs way has always been *isearch*, or *incremental search*: you can press `C-s` and start to type, and the point moves to the nearest occurrence of the typed character sequence while you

enter more and more characters (and Emacs highlights all occurrences thereof). Avy, which is based on ace-jump (a previous implementation of the same idea, which in turn was based on the *EasyMotion* Vim plugin), implements a simple but powerful concept. You can invoke the `avy-goto-char` command (many people bind it to some convenient key), then press a character key and all instances of this character on the screen become highlighted with a letter or a combination of letters. Pressing the letter (or a sequence of letters) moves the point to the respective location. It is an extremely fast and very convenient way of navigating, and also has many variants (like only selecting letters at the beginning of words or jumping to beginnings of lines).

2.4 Org-mode and LaTeX

In recent years, a new invention took the Emacs world by storm: Org-mode. Originally a note-taking mode on steroids, it quickly gained more and more features and is now a full-fledged application written on top of Emacs. (What saves it from the usual symptoms of *featuritis* is one of its design goals: advanced features should never get in the way if you do not want to use them.) It is difficult to describe Org-mode, since it combines a notebook, a literate programming environment (capable even of chaining pieces written in different languages!), a todo-list, a spreadsheet, a time-tracking tool and a few other things. What is interesting for TeX users is that Org-mode (which defines a markup syntax, quite similar to Markdown) has something called an *exporter*, which can save the document as a LaTeX article (or book), a Beamer presentation, an HTML page or even a LibreOffice document (and a few other, more obscure formats). Seasoned LaTeX users might scoff at such an idea and claim that one does not get the full power of LaTeX — and they would be right to some extent. However, for many people Org-mode syntax is much friendlier than LaTeX's, and most scientists do not use advanced TeX features anyway. (Also, you can embed arbitrary LaTeX stuff in an Org-mode document.) There is one place, however, where Org-mode is clearly superior to LaTeX: tables. Table editing in LaTeX is far from pleasant (although Emacs can help with that with automatic alignment of the table source on ampersand characters), and table sources are not very legible. On the other hand, see figure 1 for a table written in Org-mode and what Org-mode made with it when asked to export to LaTeX. Note that Org-mode has very good support for ASCII table editing — it takes care for making columns wide enough to accommodate the widest entry, for instance. Also, please note the last line, which has two

```
| Product | Price | Quantity | Amount |
|---------+-------+----------+--------|
| Bread   |  4.50 |        1 |   4.50 |
| Apples  |  2.40 |        4 |   9.60 |
| Tea     |  6.99 |        2 |  13.98 |
|---------+-------+----------+--------|
|         |       |    Total |  28.08 |
#+TBLFM: $4=$-1*$-2;f2::@5$4=vsum(@I..@II);f2

\begin{center}
\begin{tabular}{lrrr}
Product & Price & Quantity & Amount\\
\hline
Bread & 4.50 & 1 & 4.50\\
Apples & 2.40 & 4 & 9.60\\
Tea & 6.99 & 2 & 13.98\\
\hline
 &  & Total & 28.08\\
\end{tabular}
\end{center}
```

Figure 1: Org-mode table and the result of LaTeX exporting

formulæ for totaling. While an Org-mode spreadsheet does not feature automatic recalculation (it has to be triggered by issuing a special command), it is capable of performing quite advanced calculations — Org-mode utilizes Calc, a scientific calculator written in Elisp, for that.

3 Emacs customization

One of the most prominent features of Emacs is its flexibility. (In the first sentence of the Emacs manual it is called "the extensible, customizable, self-documenting real-time display editor".) There are literally thousands of options even in stock Emacs (without any packages loaded, Emacs has more than 2 000 variables; my Emacs has almost 15 000), and most packages add their own share. All these options can be set up using a nice and discoverable interface or manually, by editing an initialization file. For instance, AUCTeX by default asks the user whether to save the (La)TeX file before compilation. One can set, however, the `TeX-save-query` variable to nil (which is Elisp for "false"), and from then on the saving would be automatic.

Where Emacs really shines, though, is not in its *customizability*, but in its *extensibility*. Emacs has a small core written in C, but the rest is written in Elisp. In a properly configured Emacs, the source code for any command is a few keystrokes away, and you can modify its behavior within seconds. Of course, this requires knowledge of Emacs Lisp — but it is not a difficult language, and you can learn the basics within a few afternoons. There is an excellent

book [2], which is an introduction to Elisp for non-programmers. There is also the book [3], which is kind of a next step, although parts of it seem to be pretty outdated.

For the rest of this article I am thus going to talk about how you can mold Emacs to fit your needs. It is a selection of snippets of code which show the customizability and extensibility of Emacs.

I cannot resist to mention here that while writing this very paper I was advised against spending more time on coding one's own little extensions to Emacs, the argument being that few people actually do it. While it is probably true that only a minority of Emacs users learn enough Elisp to do it, I beg to differ — I think many more people *could* benefit from doing exactly this if only they knew how. There is this lovely quotation from one of rms' speeches [9], however, which comments on the issue:

> Multics Emacs proved to be a great success — programming new editing commands was so convenient that even the secretaries in his office started learning how to use it. They used a manual someone had written which showed how to extend Emacs, but didn't say it was a programming [task]. So the secretaries, who believed they couldn't do programming, weren't scared off. They read the manual, discovered they could do useful things and they learned to program.

Please consider this section as a kind of teaser which might hook you into Elisp programming.

3.1 Support for tildes

As a Polish TEX user, I tend to put a lot of tildes (hard spaces) in my files. (A Polish tradition is not to break a line after a one-letter word, and we have a few one-letter prepositions and conjunctions.) Of course, TEX has a solution to that: tildes (called "ties" in *The TEXbook*). Typing them manually is rather tedious, though. Happily, Emacs has you covered: there is a (built-in) package called "tildify", which replaces spaces after one-letter words (or in other places, since it is configurable). It is not even restricted to TEX — it can insert ` ` in HTML files, for example. It can do it dynamically while you type or after the fact on some portion of the file.

This is, however, not enough. I find myself editing other people's files on a regular basis, and oftentimes I need to insert a tie where a space was. In a regular editor this means navigating to the right place, deleting the space and inserting the tie. At one point I asked myself a question: how often do I need to have a tie next to a space? The answer is: *never*. Thus I decided to bind the tilde to a command which

```
(defun smart-tie ()
  "Delete any whitespace character(s),
then insert a tilde."
  (interactive)
  (delete-horizontal-space)
  (insert "~"))

(eval-after-load 'tex
  '(define-key TeX-mode-map "~" 'smart-tie))
```

Figure 2: The `smart-tie` Elisp source code

```
(add-hook 'TeX-mode-hook
  (lambda ()
    (font-lock-add-keywords nil
     '(("~" . 'font-latex-sedate-face)))))
```

Figure 3: A snippet making ties gray

starts by deleting any whitespace around point and only then inserting a tie (figure 2).

Emacs commands are just Elisp functions (defined with `defun`), which contain an `(interactive)` call at the beginning. (Notice that before that, there is a docstring. While not mandatory, it is a good practice to include it in all Elisp functions.) Notice that one of the reasons coding simple Emacs commands like this is, well, simple, is that you can just write down the things you press in order to make a similar edit manually, then check what commands are run by these key-presses, and just make a function out of them. It is even possible to have Emacs do it for you — you can record a so-called *keyboard macro*, performing some editing functions by hand, and let a suitable command generate an Elisp function mimicking your actions.

(I am not going to explain Elisp syntax in this article. I think much of it is pretty self-explanatory for anyone into programming, and for those new to it, the book [2] is a nice general introduction to both programming and Emacs Lisp.)

One last thing connected with ties is legibility. While I appreciate the fact that a (LA)TEX source file is plain text and hence I can see exactly where a hard space is, lots of them make the text less readable. I would prefer if they were gray instead of black (I use a dark-on-light default color theme). This is not a problem for Emacs. I put the snippet from figure 3 in my init file, and from now on all my tildes are displayed in gray. Here you can see a hook in action. `TeX-mode-hook` contains a list of functions called when turning any TEX-like mode on. We add to the hook an anonymous function (introduced with the `lambda` macro) which adds the `~` character to the "keywords" recognized by the font-locking machinery.

```
(defun smart-self-insert-punct (count)
  "If COUNT=1 and the point is after
a space, insert the relevant character
before any spaces."
  (interactive "p")
  (if (and (= count 1)
           (eq (char-before) ?\s))
      (save-excursion
        (skip-chars-backward " ")
        (self-insert-command 1))
  (self-insert-command count)))

(eval-after-load
  'tex
  '(define-key TeX-mode-map
              ","
              'smart-self-insert-punct))
```

Figure 4: The `smart-self-insert-punct` Elisp code

3.2 Smart commas

A common mistake is to forget a comma where it is needed; a copyeditor has to insert a lot of these. Since many navigation commands land the point at the beginning of some word, I always had to press the left arrow and then insert a comma. And then it struck me that I virtually *never* need a comma *after* a space between words, so why not automate this? And thus I wrote a short command called `smart-self-insert-punct` (see figure 4), which detects whether the point is after a space, and if yes, backs up first before entering the character used to issue the command.

This code is more or less self-explanatory (at least when you get accustomed to the Lisp prefix notation — for instance, to check for equality of two numbers `a` and `b`, you write `(= a b)`), but two things are probably worth mentioning. First of all, the `(interactive "p")` part performs some tricks so that `count` is one unless the user presses something like `C-u` ⟨*number*⟩ before issuing the above command. This is called a *prefix argument* and serves as a repeat count for many commands. Then, we have the very useful `save-excursion` form, which remembers the position of the point, performs the code given and returns the point to its previous position. (You usually do not expect the point to jump around when Emacs does something, and Emacs can do a lot of things — like spell-checking, for instance — even without the user doing anything.)

3.3 Converting `\cites`

As an editor of *Wiadomości Matematyczne* I often receive a paper with lots of citations done wrong. Many times the author says something like

```
see papers~\cite{A} and~\cite[p.~12]{B}.
```

```
(defun skip-cite-at-point ()
  "Move point to the end of the \\cite
at point."
  (when (looking-at "\\\\cite")
    (forward-char 5)
    (cond ((= (char-after) ?\[)
           (forward-sexp 2))
          ((= (char-after) ?\{)
           (forward-sexp)
           (when (and (not (eobp))
                      (= (char-after) ?*))
             (forward-char)
             (forward-sexp)))
          (t (error
              "Malformed \\cite")))))

(defun cites-to-citelist ()
  "Convert region to a \\citelist command.
All \\cite's are preserved and things
between them deleted.  This command will
be fooled by things like \"\\\\cite\"."
  (interactive)
  (if (use-region-p)
      (let
          ((end (copy-marker (region-end))))
        (goto-char (region-beginning))
        (insert "\\citelist{")
        (while (< (point) end)
          (skip-cite-at-point)
          (delete-region
           (point)
           (if (search-forward "\\cite" end t)
               ((progn )
                (backward-char 5)
                (point))
             end)))
        (insert "}"))
    (message "Region not active")))
```

Figure 5: The `cites-to-citelist` command

Since we use AMSrefs, this should be converted to something along the lines of

```
see papers~\citelist{%
  \cite{A}\cite{B}*{p.~12}}.
```

Hence I wrote another simple Elisp command, called `cites-to-citelist` (see figure 5), which performs the conversion for me. (It does not perform the whole job, i.e., it leaves the optional argument in brackets. This is not a huge problem, since I have another command to convert it to the AMSrefs syntax.) These commands are actually more complicated. I will not explain them in full, but let me highlight a few key points. (If you are interested in learning the details, you can use [4] and/or install Emacs and look at the docstrings of all the functions called in

the above code.) Again, let me emphasize that writing a `skip-cite-at-point` function is easier than it might seem, since it mimics the operations you (as a user) would perform to move the point past the `\cite` LaTeX macro: first, you check whether you are actually on it, then move by five characters, then move forward past the part(s) enclosed in brackets/braces. Also, in the `cites-to-citelist` function, we utilize the *region*, which is the Emacs term for the selection.

4 Conclusion

As you could hopefully see, Emacs works extremely well as a (LA)TeX editor. There are three reasons for that. First and foremost, it is an excellent general-purpose editor, with a simple TeX mode included. Secondly, there is the AUCTeX package, which is a robust tool, still under active development, and numerous other packages, like RefTeX, pdf-tools and Org-mode, which make the experience even better. The third reason is that Emacs truly delivers on its promise to be *extensible, customizable, self-documenting*, and automating repetitive tasks is fairly easy. If you are currently using TeXworks or even Vim (or any other TeX editor — there are so many of them), do yourself a favor and try out Emacs. You might stay in it for your whole life!

If you want to learn more about Emacs, you can install it and start with the built-in tutorial and proceed to at least skimming the manual. There is also a reference card included in the distribution, and others available on the Internet. A very good source of tips for using (though not programming) Emacs is Mickey Petersen's book *Mastering Emacs* [6]. A good source of useful information is *Planet Emacsen* [7], an Emacs blog aggregator. You can ask all sorts of Emacs-related questions on the official mailing list [5]. If you want to start your own adventure with Elisp, definitely start with Robert Chassell's *An Introduction to Programming in Emacs Lisp* [2]. Finally, let me mention a (now dormant) project of mine of writing a modern sequel to Chassell's book, which I hope to revive this year; I will surely post updates to it on my blog at `http://mbork.pl/Content_AND_Presentation`.

Acknowledgments

I would like to thank my friends from the `gust-l` and `help-gnu-emacs` mailing lists for their valuable input and many suggestions, and I am indebted to the editors for their excellent proofreading job.

Bibliography

[1] Marcin Borkowski, *Ten years of work in Wiadomości Matematyczne — an adventure with LaTeX and Emacs hacking*, TUGboat **38** (2017), no. 2, 255–263. `https://tug.org/TUGboat/tb38-2/tb119borkowski.pdf`.

[2] Robert J. Chassell, *An Introduction to Programming in Emacs Lisp*, 3rd ed., GNU Press, 2006. Bundled with Emacs source code and available in the Emacs Info documentation system. `https://www.gnu.org/software/emacs/manual/eintr`.

[3] Bob Glickstein, *Writing GNU Emacs Extensions*, 1st ed., O'Reilly Media, 1997.

[4] Bil Lewis et al., *GNU Emacs Lisp Reference Manual*. Bundled with Emacs source code and available in the Emacs Info documentation system. `https://www.gnu.org/software/emacs/manual/elisp`.

[5] *help-gnu-emacs*: *Users list for the GNU Emacs text editor*, `https://lists.gnu.org/mailman/listinfo/help-gnu-emacs`.

[6] Mickey Petersen, *Mastering Emacs*, v2, 2016. Available at `https://www.masteringemacs.org/book`.

[7] *Planet Emacsen*, `http://planet.emacsen.org/`.

[8] Richard M. Stallman et al., *GNU Emacs Manual*. Bundled with Emacs source code and available in the Emacs Info documentation system. `https://www.gnu.org/software/emacs/manual/emacs`.

[9] Richard M. Stallman, *My Lisp Experiences and the Development of GNU Emacs* (2002), available at `https://www.gnu.org/gnu/rms-lisp.html`.

◇ Marcin Borkowski
Faculty of Mathematics
and Computer Science
Adam Mickiewicz University
ul. Umultowska 87
61-614 Poznań, Poland
`mbork (at) amu dot edu dot pl`
`http://mbork.pl`

Editor's note: As it happens, I (Karl), like Marcin, work in Emacs, but my environment is set up completely differently from his. After comparing notes, Marcin and I thought it might be interesting to briefly describe mine as well, as an example of Emacs's extreme customizability and extensibility. All my changes are done at the Elisp level.

Some 35 years ago when I started using Emacs, my basic idea is to eradicate editing modes altogether. No `tex-mode`, no `c-mode`, etc. Keystrokes mean the same thing no matter what's being edited. I eliminate all fontification and colorization. Those are just distractions for me; I want to focus on the text.

I've also rebound nearly every key, and created hundreds of new bindings and many simple functions, so that I can do more things with less effort. For instance: save all buffers and run (what's normally) `M-x compile` with one keystroke. I typically do this dozens of times a day (I use `make` for essentially all building, e.g., running TeX). I read mail inside Emacs, use shell buffers (inside Emacs) for working locally, ssh buffers (inside Emacs) for working remotely, besides logging in remotely ... to run Emacs.

I primarily still use Emacs 21.[34], in terminal mode (not X mode), because (a) the Unicode support in new releases is painful for me when editing *TUGboat* papers in other encodings (autorecognition of encodings doesn't always work), or which use characters not in my favorite font. Just give me the bytes! (b) The changing of interfaces at every level, with no easy way back to previous behavior, that the Emacs developers have engaged in is too time-consuming for me to keep up with, especially when there is no significant benefit to the new versions in my environment. ◇