1 Appendix: guidelines and good practices

1.1 Good practices: reports

1.1.1 The one rule guide to making good charts

Making good charts is child's play. However, when it actually comes to producing one by ourselves, major failures usually occur[1].

Here is a minimalist guide on good practices for drawing a good chart[2]. We will limit ourselves to the elements that are really essential to a good layout. (The advanced reader who wants to improve on the topic is encouraged to read Loren Shure's excellent "Making Pretty Graphs" posts [1])

If there is a single one idea to remember from this mini-guide, it is "MAKE LEGENDS". This applies to the three elements that make up the figure: its **title**, its **axes** and its **plot lines**.

Let's walk you through the step-by-step process of making a good chart.

We have a table of values, derived from measurements or calculations.

```
mois = [1:12];
temp_minimale_paris = [-4.3 9 5.9 9.3 7.4 15.6 14.6 16.7 13.8 13 5 0.8];
temp_minimale_papeete = [25.5 25 26.2 25.3 23.9 23.9 24.3 23 21.7 21.5 25.5 25];
```

We draw the basic graph (the one that must absolutely not shown to a third person because it is unusable!):

```
plot(mois,temp_minimale_paris)
hold on
plot(mois,temp_minimale_papeete)
hold off
```

Now we go to the essential step of this mini guide: MAKE LEGENDS ! This includes :

1. To put a title to the chart (ideally that states the goal of the chart)
2. To put a legend for all axes, ie to indicate for each axis the name of the associated quantity as well as its unit, in parenthesis
3. To put a legend for the different lines

Here are the associated command lines:

```
% Title of the graph
title('A Year of minimal temperatures on the 1st of the Month in 2017')
% Axes' names
xlabel('Month number')
ylabel('Temperature (deg. C)')
% Lines' names
legend('Paris (France)','Papeete (Fr. Polynesia)')
```

[1] which can have direct impacts on grades...
[2] which increases the probability of high grades...

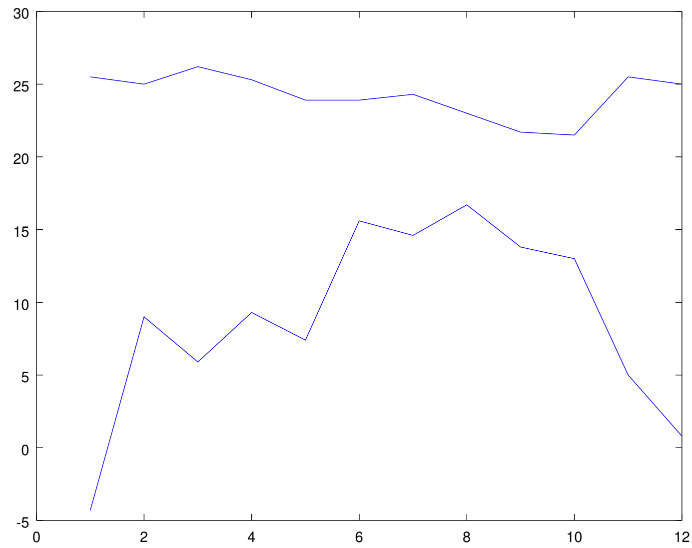

Figure 1.1: *Displayed figure at the screen: example of bad figure*

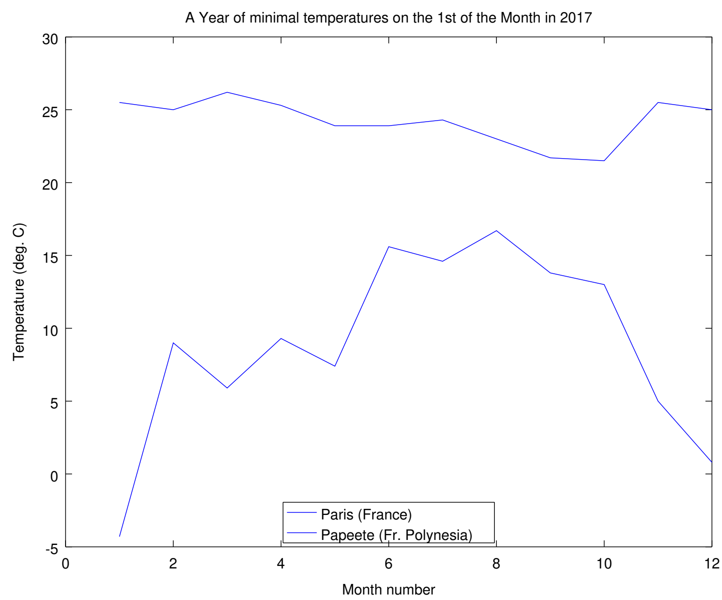

Figure 1.2: *Displayed figure at the screen: example of somewhat better figure*

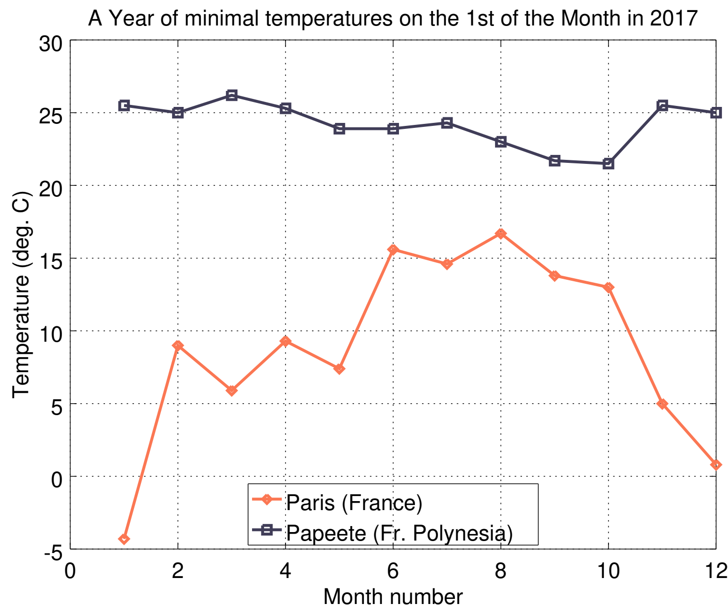

Figure 1.3: *Displayed figure at the screen: example of good figure*

Now we can go to the aesthetic part (which is strongly recommended):

```
% Display grid
grid on
% Get plot handles for each line
line_hdl=get(gca,'children')
% Change markers shape
set(line_hdl,{'marker'},{'s';'d'})
% Change color lines
set(line_hdl,{'color'},{[64,61,88]/256;[252,119,83]/256})
% Change linewidth
set(line_hdl,'linewidth',2)
% Set Fontsize of text elements
set(gca,'fontsize',15)
set(get(gca,'title'),'fontsize',15)
set(get(gca,'xlabel'),'fontsize',15)
set(get(gca,'ylabel'),'fontsize',15)
```

Comment : the last commands for changing the font size can be implemented with the following one-liner command :

```
set(findall(gcf,"-property", 'fontsize'),'fontsize',15)
```

It is advisable to configure your *startup* file `~/.octaverc` [3] so that the font size and the linewidth have the proper default values [4].

[3] https://octave.org/doc/v4.2.0/Startup-Files.html
[4] https://octave.org/doc/v4.2.0/Managing-Default-Properties.html

1.2 Good practices: programming

As with any software development project, following good practices will help ensure that your results are accurate, avoid or solve programming mistakes, and save you time in the long run. Here are some recommended guidelines, further documented in [2, 3].

1.2.1 Use a version control system

Don't share code by email or a mere folder such as our school's Owncloud service, let alone Dropbox®. You *will* get confused between versions. Git repositories will be provided. Use them as the reference version of your code.

See section 1.3 for more details and help. **The homework section ?? on page ?? details the specific operations you MUST be familiar with** for efficient collaborative work in the project.

1.2.2 Document everything, and keep it up-to-date

If you don't see the need to make your code easy to understand to other people, consider that yourself next month will be another person. You *will* waste time trying to understand what you had in mind when you wrote last month's code if you're not careful. Additionally, your teachers will evaluate your code, and will be in a better mood if they don't have to warp their mind around this undocumented optimization that you thought was so clever while pulling an all-nighter.

Comments in Octave begin with `%`. An whole line of comments usually begins with `%%`. Your text editor/development environment will indent them accordingly (see section 1.2.3). Use them to document:

- Functions: the first few lines of comments in a function are displayed when you type "`help` <*function*>" in Octave. Use this to tell the reader what your function does, both in short and in detail; what are the expected parameters and return values, what they represent and any assumption you make; and for more complex functions, some examples and non-obvious corner cases.

- Blocks of code: you don't have to add a comment to every line of code, nor to write trivialities such as:

  ```
  x = x + 1 % Add 1 to x.
  ```

  Rather, isolate meaningful blocks of code and document what the code achieves.

  ```
  %% Count total symbols sent, print status every million symbols.
  symbols_sent = symbols_sent + N;
  if (~ mod(symbols_sent, 1e6))
    fprintf('%d sent, %d errors... ', symbols_sent, errors_dfe);
  end
  ```

Finally, keep this documentation religiously up to date. You may waste a lot of time before realizing that a section of the code doesn't do what the comments say, because someone forgot to update them. Always keep them synchronized.

1.2.3 Use spacing and indentation for best legibility

Don't pack your code into the smallest possible space. Make it easy to read. If a formula doesn't fit on a single line, break it into several lines, adding "`...`" at the end of each unfinished line.

Your text editor/development environment will automatically indent code blocks to provide a visual reference of where it fits. This helps not only to read it but also to write it: if the automatic indentation level surprises you, then perhaps you forgot an `end` or a closing parenthesis somewhere.

1.2.4 Don't repeat yourself

Any piece of data or code should have one single representation in the entire program. If you copy-paste values or code, you *will* forget to update all of them when an update is required. Use functions instead, and pass them parameters. You may group related values in a `struct` and pass that as a single parameter.

1.2.5 Use meaningful variable names

It should always be obvious what a variable represents. The common-sense method is to name them `nb_errors` or `nb_symbols_sent` instead of `n_e`, `n_s`. However, in scientific computing, this may clash with the mathematician's habit of giving variables single-letter names. We recommend that single-letter variable names only be used sparingly:

- for a limited number of standard notations shared across the entire program; even then, you may want to index them (as in `X_in`, `X_out`);
- for local variables within a short function.
- Also, beware of loop counters: the traditional notation `i` clashes with Octave's usual notation for $i = \sqrt{-1}$; avoid calling your counter `i`, and/or use the explicit notation `1i` for $\sqrt{-1}$.

1.3 Good practices: version control using Git

1.3.1 Basic Git and version control concepts

This section is intended for people not already familiar with Git. For more details, you may also refer to first-year lectures on the topic [4, 5] (in French, partly based on SmartGIT, a graphical wrapper around Git); or to any of the many tutorials available online.

Version control concepts in general will be expressed in Git command-line terminology; graphical Git tools will have similar concepts (as well as most other version control software).

- A **repository** contains all your project's files, with a full revision history and references to who originated each line of code.
- A **working copy** is a repository in which you'll be editing the files. That's what you'll typically have on your hard disk: a folder containing your files, that you'll be working on, and a hidden subdirectory `.git/` containing the revision history.
- **Cloning** a repository (command: "`git clone`") means creating a working copy with the same files as another repository (usually called "`origin`"), set up to enable synchronization

between them. Typically your working copies will be cloned from your main project repository, which is hosted in the school's Gitlab server.

- A **changeset**, also called "**commit**", represents the change between different versions of your project. Git stores the history as a series of changesets.
- **Staging** files (command: "`git add`") means marking the state of your chosen files in your working copy to be part of the next changeset. Git calculates the difference between the selected files' current content and their state in the previous changeset. The previous changeset is called "`HEAD`".
- `git commit` creates a new changeset from the file contents that you selected using `git add`. (Watch out: if you have modified one of the files after `git add`, the new changeset will not contain your modifications, it uses the file contents *at the time you did* "`git add`"!) After the commit, "`HEAD`" is updated to designate this new changeset.
- `git push` uploads your new changesets to the `origin` repository. It will fail if someone else has been working on it in the meantime, therefore **you must always *pull* from the** `origin` **before pushing** (see below).
- `git fetch` downloads any new changesets from the `origin` repository, but doesn't update the files in your working copy.
- `git merge` (simplified version) resynchronizes the files in your working copy with any changesets downloaded from the `origin`. If they can be automatically merged with your local changesets (no "conflicts", see below), the new file contents are automatically staged; you can then create a new changeset using `git commit` to finalize the merge, and upload it using `git push`. (Note: you must always commit all your changes before doing `git merge`.)
- `git pull` = `git fetch` + `git merge`.
- A **conflict** is a situation where Git fails to merge the local changesets and the `origin`'s. At this point, **your files have been modified, you must fix them** before you can `git push` again (or you can abort the merge using `git merge --abort`, which restores the files to their previous state until you can deal with the problem). That situation can largely be avoided by working with branches as described in sections 1.3.2 and 1.3.3. (Conflicts may still arise when merging different branches, but the situation will be better controlled.)

Fixing conflicts: a number of tools can be used to fix merging conflicts. The most straightforward is editing the files as you would normally do in your editor, looking for sections where Git couldn't decide which version to keep, that will be marked by lines containing <<<<, ==== and >>>>.

Otherwise you can see what `git mergetool` does on your system; or maybe you're using a GUI wrapper around Git that integrates a tool to help with conflicts.

Other useful git commands:

- `git status` tells you the state of your working copy: which files have been modified, what branch it is on (see section 1.3.2), and whether there are new changesets to upload or to merge. **Always check** `git status` **before doing** `git commit`**!**

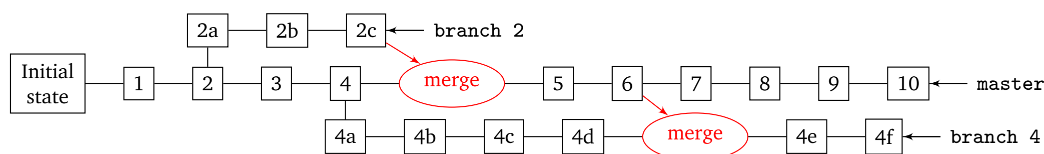

Figure 1.4: *Example Git repository history with multiple branches. Each node represents a changeset. "`master`" is the reference branch. "`branch 2`" diverges from "`master`" at changeset 2, and is later merged into master: typical of a temporary development branch. "`branch 4`" diverges from "`master`" at changeset 4, and merges* from *"`master`" at some point but continues on: typical of an ongoing development branch that sometimes reincorporates other changes from "`master`".*

- `git diff` shows the differences between the files' current state and what it was in the latest changeset (actually the changeset currently designed `HEAD`).
- `git checkout [`*commit-id*` -- ]` *file* restores a file to the state it had in `HEAD` (or in the changeset *commit-id* if specified), erasing the modifications.
- `git log [--graph]` lists the changesets (and their id!) on the current branch. The `--graph` option shows the merges if any.

If you really can't find a way out: you can always clone your main repository into another working copy, transfer your files from your broken working copy in the state you want them in, then commit in the new working copy and use it instead of the broken one. Normally you shouldn't need to do this: the Git commands described above, and many others, should allow you to recover from mistakes (see in particular [6]); but if you're not familiar with them, it can be quicker to restart from a new clone. Note: you can always recover the local versions of your files using `git log` and `git checkout`.

1.3.2 Collaborative work with Git branches

Your project's history doesn't have to be linear. All changesets are relative to a prior version, but multiple "branches" can diverge and be merged later. Figure 1.4 shows an example with 2 branches in addition to the `master` branch.

You will use your repository's `master` branch as the reference version of your code; and **use branches to experiment with code without breaking the reference version**. After experimenting, you can merge the features you developed in the experimental branches back into `master` using `git merge`, as per the workflow described in section 1.3.3.

- `git switch [-c] "`*branch name*`"` switches the working copy to the given branch.[5] Use `-c` to create the branch if it doesn't already exist.
- Git tracks branches both locally in your working copy and in your `origin` repository. Note the difference between `master` and `origin/master` (or `origin/`*any branch*) before and after `git pull`.

[5]Older Git versions used `git checkout [-b]` before the introduction of `git switch`. It can still be used, but may lead to ambiguities. **Do not give a branch the same name as a file!**

- `git merge "`*branch name*`"` merges the given branch into the current branch. (Without a branch name, as noted above, the default behavior merges `origin/`*current branch* into the current branch.) As always, you may have to solve conflicts as described above.
- `git commit` and `git push`, as noted above, create and upload new changesets *in the current branch*. You may get an error if you try to `git push` but created the branch only locally in your working copy (which is the default); the error message will tell you what command to execute to create the branch also in your remote repository (with the `--set-origin` option).

1.3.3 Recommended workflow

Once per project:

- `git clone` to get a working copy into a folder on your local computer. Go into the folder.
- `git config user.email "`*your email*`"` and `git config user.name "`*your name*`"` to identify yourself as the author of the work you're going to do. (Alternatively, you can use `git config --global` to do this step once and for all the Git repositories you will use on that computer unless otherwise indicated.)

For every session:

- Go into your working copy.
- `git switch -c develop-`*name-yyyymmdd* to create and switch to a new development branch named "`develop-`*name-etc.*" dated from today (should be a unique name).
- Work on the files in the working copy.
- For every consistent change:
  - `git status`, `git diff` to check which files have been modified and how;
  - `git add` to stage the relevant files;
  - `git commit` to create the changeset (remember to write a meaningful description);
  - (better err on the side of smaller changesets with fewer files).
- At the end, either `git push` your local branch to the main repository, if you just want your work to stay in that branch, or merge it into the `master` branch:
  - `git status`, make sure all your changes are committed;
  - `git switch master` to go back to the `master` branch;
  - `git pull`, `git log` to get and review the latest changes to `master`;
  - `git merge develop-`*name-yyyymmdd* to incorporate your work into the branch;
  - if there are conflicts, fix them, then `git status`, `git commit` to create the changeset that completes the merge;
  - (alternatively, `git merge --abort` to return the files to their state in `master` if you don't want to fix the conflicts for now;)
  - after the conflicts are fixed and the merge is complete, `git push` to send your changes to the main repository.

1.3.4 Other version control tips and tricks

- If there is an error while using Git, pay attention to the error message. Git is more informative than most software, tries to guess what you're trying to do, and gives you the necessary command.
- The command `git stash` automatically saves your un-committed modifications, in case you don't want to commit them yet, but need to switch branches or do anything that requires a clean working copy.
- When you write a commit message, the first line is known as the "subject line". It is a good practice to limit this subject line to 50 characters. In large scale project, commits should also match a given template. For example, it is a good idea to write your commits as follows :
  - the title should have the following pattern: `<file/module name>: <title>`
  - the commit should contain a brief description of what you have done in it.

  You can check the latest commits from one of Analog Devices GitHub projects in order to see examples: [7]. It is now straightforward that this kind of standard commit applies to single files; which is a much better practice than modifying tens of files at the same time.

Bibliography

[1] Loren Shure. *Making Pretty Graphs*. MathWorks. Dec. 2017. URL: https://blogs.mathworks.com/loren/2007/12/11/making-pretty-graphs/.

[2] Greg Wilson et al. "Best practices for scientific computing". In: *PLoS biology* 12.1 (2014), e1001745. DOI: 10.1371/journal.pbio.1001745.

[3] Michael Robbins. *Good Matlab Programming Practices for the Non-Programmer*. 2001. URL: http://www.mit.edu/~pwb/cssm/GMPP.pdf.

[4] Rémi Sharrock. *TP SmartGIT*. URL: https://perso.telecom-paristech.fr/bellot/CoursJava/tps/tpgit.html.

[5] Jean-Claude Dufourd. *Transparents PACT*. URL: https://perso.telecom-paristech.fr/dufourd/cours/pact/pact-gl2n.html#/git-rappel-inf103.

[6] Serdar Yegulalp. *6 Git mistakes you will make — and how to fix them*. 2020. URL: https://www.infoworld.com/article/3512975/6-git-mistakes-you-will-make-and-how-to-fix-them.html.

[7] Analog Devices. *Analog devices LibIIO GitHub webpage*. https://github.com/analogdevicesinc/libiio/commits/master. 2021.