Module 5

Ryo Kimura

5 Coding Tools and Documentation

5.1 Version Control Systems (VCS)

Coding is an iterative process, where we write some code, test it, find some bugs, fix it, test it again, find more bugs, etc. During this process, it is often helpful to have a record of the changes made, particularly if the code is being written collaboratively. *Version Control Systems (VCS)* are designed specifically for managing changes to source code over time. We will primarily discuss **Git** due to its widespread use in open-source software and its eponymous integration with **Github**, a website hosting many open source projects.

5.1.1 Why should I use a VCS?

Beyond just keeping track of code changes, VCS allow developers to:

- restore source code to a previous state
- manage different versions of the source code
- resolve conflicting changes to the source code
- pinpoint when certain bugs were introduced

Thus, while VCS show their usefulness most when developing code as a team, it has advantages for individual developers as well.

5.1.2 Setting up a Git Repository

If you want to set up a copy of an existing Github repository on your local machine, it's pretty simple; run `git clone <repoURL>`, where `repoURL` is the URL of the repository (e.g., https://github.com/username/reponame), then run `git config --global user.name <name>` and `git config --global user.email <email>` to set the name and email associated with your commits.

On the other hand, if you want to set up a *new* Git repository, perhaps based on a project folder with existing files, follow these steps:

1. In the command line, navigate to the directory you want to set up.
2. Run `git init` to initialize the Git repository.

3. If you have not done so already, run `git config --global user.name <name>` and `git config --global user.email <email>` to set the name and email associated with your commits for all of your Git repositories.[1]

4. Use `git add <filename>` to add files you want to track (if any).

5. Use `git commit -m"Initial commit"` to make the initial commit.

In addition, you may want to create a clone of your repository on Github. This gives you a backup of your repository in case you accidentally delete it, as well as being able to access your code over the internet. To create a clone of a local Git repository on Github, follow these steps:

1. Go to Github at https://github.com, and create an account if you don't already have one.

2. Log in to your Github account.

3. Create a new repository on Github; see https://help.github.com/en/articles/adding-an-existing-project-to-github-using-the-command-line for details.

4. In the command line, run `git remote add origin <remoteURL>`, where `remoteURL` is the URL of your repository (e.g., https://github.com/username/reponame).

5. Use `git push origin master` to push the local changes to the remote repository.

See https://help.github.com/en/articles/adding-an-existing-project-to-github-using-the-command-line and https://kbroman.org/github_tutorial/pages/init.html for details.

5.1.3 Basic Git Workflow

The general workflow when using a Git repository is as follows:

1. Edit files (e.g., `example.cpp`) using a text editor.

2. Use `git status` to see which files have been modified, and use `git diff` to check what changes were made to which files.

3. When you are ready to save your changes, use `git add` to mark the relevant files (e.g., `git add example.cpp`).

4. After marking all the files, use `git commit` to officially "commit" your changes, along with a comment describing the changes using the `-m` flag (e.g., `git commit -m "Made changes X, Y to example.cpp"`).

5. If you have a remote repository set up, run `git push` to push those changes to Github.

[1] If you want to use different names and emails for each of your repositories, run `git config` without the `--global` option.

5.1.4 Basic Resetting Commands

Broadly speaking, file changes in Git go through four "save" phases:

1. In the *edit* phase, the changes are not saved by Git in any way.
2. In the *stage* phase, the changes are saved only in the current branch of the project. This is done using the `git add` command.
3. In the *commit* phase, the changes are saved as part of the project history. This is done using the `git commit` command.
4. In the *push* phase, the changes are pushed to a remote repository (e.g., Github) so the changes are publicly accessible by others. This is done using the `git push` command.

Each of these phases has a different command for "reversing" one phase:

- (*Edit*): Git has not saved your changes in any way, so you must manually undo or use your text editor's undo function.
- (*Stage*): Use `git reset <filename>` to reverse `git add <filename>`. This removes the file from the staging phase.
- (*Commit*): Use `git reset HEAD^` to reverse `git commit`. This removes the current commit entry while keeping all file changes intact.[2]
- (*Push*): Use `git revert HEAD` followed by `git push` to reverse changes in the push phase. This creates a NEW commit that reverses the effects of the last commit, then pushes it to the remote repository.

5.1.5 Recovering Files

The simplest way to recover a file is to run `git checkout -- <filename>`, which restores a file to the state associated with the last commit. This is useful when you accidentally delete a file or when you want to "start over" from the last commit. If you want to restore a file to a state associated with an older commit, follow these steps:

1. Run `git log` to identify the commit you want to go back to, based on the commit comments. Note that each commit is associated with a unique id consisting of a string of numbers and letters (e.g., commit 157bba1d151e758a0441b4376b51469d358999c9).
2. To restore your repository to the state associated with, say commit 157bba1d15..., run a command like `git checkout 157bba1d`. Note that you do not have to specify the entire commit id, only enough that it can be uniquely identified.
3. At this point, you will get a message saying that Git is in 'detached HEAD' state. This just means that any changes you make will not be saved to the repository by default. If the "current" version of the file is what you want, copy it to a different location.
4. To restore Git to its normal state from the 'detached HEAD' state, run `git checkout master`.

See https://www.atlassian.com/git/tutorials/undoing-changes for more details on `git checkout`.

[2] If you want to reverse changes in the files as well, you can use `git reset --hard HEAD^`. Note, however, that this change is irreversible!

5.1.6 Incorporating Changes from a Remote (Github) Copy

So far we have focused on using Git in the context of a project with a single developer. However, Git is most commonly used when collaborating on a project with multiple developers. In this case, it is useful to know how to incorporate changes from other versions of the code (e.g., the one on Github which may be updated by others) with your local version of the code:[3]

1. Run `git fetch origin` to download changes from the Github repository, if any.
2. Run `git merge origin/master`[4] to merge changes into your local copy.[5]
3. At this point, the merge may fail if you have unsaved changes in your local copy. If so, run `git stash` to temporarily "stash" them away (note that `git stash` will NOT save changes to any files that Git is not tracking; in particular, if you have new files or files that are being ignored that you want to keep, use `git add <filename>` to stage them BEFORE running `git stash`), then run `git merge origin/master` again.
4. After the merge is successful, run `git stash pop` to incorporate your stashed changes to the updated code.
5. At this point, you may get a *merge conflict* if Git cannot automatically merge the two versions together. If so, resolve the conflict by looking at the affected files and making the appropriate changes (see https://www.atlassian.com/git/tutorials/using-branches/merge-conflicts for details). Once you have resolved the conflict, run `git reset` followed by `git stash drop`.

See https://longair.net/blog/2009/04/16/git-fetch-and-merge/ and https://stackoverflow.com/questions/7751555/how-to-resolve-git-stash-conflict-without-commit for more details.

5.2 Build Systems

5.2.1 What is a build system?

Build systems are an essential coding tool if you are working on a project that uses a *compiled* programming language (e.g., C++). In this case, the commands required for compilation can quickly become long and unwieldy, especially with multiple source code files. *Build Systems* automatically determine what commands need to be run with what options in what order, so the developer does not have to remember them.

5.2.2 Basic Structure of a Compilation Command

Since build systems are all about compilation, it is worth briefly discussing the basic structure of a compilation command. For example, here is a compilation command for a C++ Gurobi model:

[3] Of course, in order to do this your local copy must be configured to work with the one on Github. If you created your local copy with `git clone`, this is already done. Otherwise, see https://www.atlassian.com/git/tutorials/syncing on how to set it up.

[4] While `git merge` is a safe recommendation for small projects, for larger/older projects repeated merges can clutter the project history. For this reason, some people prefer using `git rebase` instead of `git merge` when incorporating changes from the Github repository. See https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase and https://www.atlassian.com/git/tutorials/merging-vs-rebasing for details about this alternative.

[5] The fetch and merge steps can also be achieved in a single command by running `git pull`. Here we chose to present the two commands separately for clarity, and also because this allows users to check the remote changes using `git checkout` if desired.

```
g++ -m64 -g -o example example.cpp -I/opt/gurobi810/linux64/include \
    -L/opt/gurobi810/linux64/lib -lgurobi_g++5.2 -lgurobi81 -lm
```

(Note that \ is simply an indicator that the command is continued on the following line.)

This command can be broken down into seven parts:[6]

- **Compiler** [`g++`]: command calling the C++ compiler
- **C++ flags** [`-m64 -g`]: control the compilation process in various ways; here `-m64` means to generate code for 64-bit processors, and `-g` means to keep debugging information in the final executable (so it can be analyzed by a debugger); you may also see flags to show/suppress certain warnings (e.g., `-Wall` or `-Wno-ignored-attributes`)
- **Output file** [`-o example`]: name of output file; here we are creating the final executable, but it can also be an intermediate *object file* (e.g., `example.o`)
- **Source code file(s)** [`example.cpp`]: name(s) of file(s) containing the C++ source code; here we only have one file, but in general we can have multiple files
- **Include paths** [`-I/opt/gurobi810/linux64/include`]: indicate where to look for header file(s) of libraries used in the source code; here we must specify the location for the Gurobi libraries (i.e., `/opt/gurobi810/linux64/include`) since they are not in a standard location like `/usr/include` or `/usr/local/include`
- **Library paths** [`-L/opt/gurobi810/linux64/lib`]: indicate where to look for library files used in the source code; here we must specify the location for the Gurobi libraries (i.e., `/opt/gurobi810/linux64/lib`) since they are not in a standard location like `/usr/lib` or `/usr/local/lib`
- **Library flags** [`-lgurobi_g++5.2 -lgurobi81 -lm`]: specify which libraries to link to the final executable; here we are using the Gurobi libraries `libgurobi_g++5.2.a` and `libgurobi81.so`, and the standard library `libm.so` (in `/usr/lib/x86_64-linux-gnu` on Ubuntu).

Often the include paths, library paths, and library flags are easy to forget or misspecify. It may be helpful to recognize the common compiler errors associated with each:

- `fatal error: gurobi_c++.h: No such file or directory`: This error tells us that the compiler could not find the header file `gurobi_c++.h` (we can tell it's a header file by the `.h` suffix). This is likely due to an error specifying the *include path*.
- `/usr/bin/ld: cannot find -lgurobi81`: This error tells us that the linker `ld` could not find the library file associated with `-lgurobi81`. This is likely due to an error specifying the *library path*.
- Many many MANY errors about `undefined reference to <object>`: This error tells us that the linker was unable to find the implementation for a class, function, etc. This is likely due to an error specifying the *library flags*.

See https://gist.github.com/gubatron/32f82053596c24b6bec6 for a more detailed discussion on the C++ compilation process.

[6] Here, we used a single command for compiling a final executable to concisely illustrate all the necessary parts. However, in practice it is common to separate the *compilation* step, which does not need library paths and library flags, and the *linking* step, which does not need include paths (see Section 1.5.3 of Module 1).

5.2.3 Make

Make is one of the most common build systems used for C++. The basic idea is to encode the commands that need to be run in a *Makefile*, which provides basic programming capabilities to specify compilation patterns.

While Makefiles can get very sophisticated, a basic Makefile can be created with the following knowledge:

- Comment lines start with a `#`.
- All variables in Makefiles (essentially) store strings. You can assign a value to a variable with `VARNAME = varvalue`, and you can refer to a variable value using `$(VARNAME)`.
- *Build rules* (i.e., patterns that specify how to compile things) take the following basic form:

  ```
  TARGET: DEPENDENCIES
      COMPILATION COMMAND
  ```

  For example, a simple build rule to compile `hello.cpp` (without using variables) looks like this:

  ```
  hello: hello.cpp
      g++ -g -std=c++11 -o $@ $^
  ```

  Here `$@` is a variable that evaluates to TARGET, and `$^` evaluates to DEPENDENCIES. If DEPENDENCIES consists of multiple files (e.g., `hello.cpp library.cpp`), `$^` evaluates to the whole list (e.g., `hello.cpp library.cpp`), whereas `$<` evaluates to just the first one (e.g., `hello.cpp`). Note that the indentation on the second line of a build rule MUST be created with a TAB.[7]

- Some build rules are designed not to compile code, but to group a series of commands together. These should be specified as such with `.PHONY`, to ensure they don't accidentally conflict with a file with the same name. Common examples include `all`, `clean`, and `install`.

See http://mrbook.org/blog/tutorials/make/ for details on a simple example of Makefiles. See https://spin.atomicobject.com/2016/08/26/makefile-c-projects/ and https://devhints.io/makefile for details on more sophisticated uses of Makefiles.

TODO: More complete explanation of Makefile structure

5.2.4 CMake

CMake is a meta-build system, in the sense that it is designed to generate configuration files for other build systems (e.g., Make). While Make often suffices for simple projects, you may find that CMake is easier to use for more complex projects.

Like C++, CMake has a reputation for being tricky to use correctly, but modern versions have features that significantly simplify usage. See https://www.slideshare.net/DanielPfeifer1/cmake-48475415 and https://gist.github.com/mbinna/c61dbb39bca0e4fb7d1f73b0d66a4fd1 for recommendations on how to use modern CMake effectively. The official CMake documentation at https://cmake.org/cmake/help/latest/ may also be helpful for specific details.

To use CMake, follow these steps:

[7] If you use vim with `expandtab`, you can put `.RECIPEPREFIX +=` at the beginning of the Makefile to use spaces instead (note the space after `+=`).

1. Create a configuration file called `CMakeLists.txt` which specifies all of the information necessary for compilation.

2. Create and navigate to the build directory (e.g., `mkdir Debug; cd Debug/`)

3. Run the command `cmake -DCMAKE_BUILD_TYPE=Debug` to generate the *build configuration*, which sets up the framework for compiling your program.

4. Run `cmake --build .` (notice the `.` at the end) to compile your program. If you need to re-compile your program, you just need to run

As long `CMakeLists.txt` does not change, you only need to run step 4 to re-compile your source code when it's edited. However, if `CMakeLists.txt` is changed (e.g., new source code file, new library deependency, etc.), you must run step 3 again to update the build configuration.

See the Canvas module for an example CMake configuration file for Gurobi and CPLEX (they require a little more editing to use compared to the Makefiles from Module 2).

5.3 Documentation

In a perfect world, all code would be so clearly written that documentation would be unnecessary. But in practice, if you foresee your code being used at any point in the future, it is worthwhile to spend time creating documentation. If nothing else, it will help you understand your own code when you come back to it months later.

Broadly speaking, there are two types of documentation: *code comments*, which are designed to help programmers editing the code in the future, and *docstrings*, which are designed to help users utilize the code in the context of another project. Docstrings are typically used to further generate *API documentation*, which is a formatted document separate from the code that explains its functionality.

5.3.1 Code Comments

It is worth noting that *code comments, by nature, explain code that is not clear enough to explain itself.* This leads to some people arguing that code without comments is almost never understandable, while others argue that comments should be avoided as much as possible, focusing instead on writing better code. While both positions are somewhat extreme, they hold merit in the sense that both of the following statements are widely accepted:

- Improving the code is always preferred over improving the comment, but
- purely self-documenting code is often impossible in practice.

In practice, this translates to the following general guidelines:

- If you feel the need to write a comment to explain what a piece of code is doing, *try rewriting the code first* to see if it can be written in a way that it no longer needs a comment.
- Use variable and function names that are descriptive and follow a consistent/conventional style (e.g., use `GenerateMasterProblem()` instead of `genmprob()` for a function that generates the master problem of a decomposition scheme). This often makes it easier to tell what the code is doing without additional comments.

- Comments should explain *why*, not *how*. Comments that state the obvious are unhelpful and distracting. Instead, comments should explain higher level ideas like the programmer's intent, design decisions, and the overall structure of the code.
- Write pseudocode in the comments before and while you code. This saves time and makes it easy to write good comments that summarize the code.

Alternatively we can categorize code comments into a few distinct categories (based on [1]), some good and some bad:

1. **Intent** [Good]: Describe what the code is *intended* to do and/or *why* (e.g., `Apply heuristic to every feasible solution found`). This can inform future programmers about design decisions that were made and how to best modify/extend the code.
2. **Summary** [Good]: Briefly summarize a section of code (e.g., `Add subtour elimination cuts`). This helps convey the high-level idea and the general structure.
3. **Non-coding information** [Okay]: Things like licenses, docstrings, links to online references, etc., which cannot be expressed by the code itself or required by other tools. Often included based on necessity rather than merit.
4. **Marker** [Okay]: Mark sections of code that still need to be developed (e.g., `TODO: Fix separation oracle for unconnected graph`). As long as a *consistent* convention is used, they can be helpful during code development. The final code, however, should not include any such comments.
5. **Assumptions** [Bad]: State assumptions that are being made about the code state (e.g., `N should be the length of the list`). Do NOT write these as comments; instead, use `assert` statements (e.g., `assert( N == myList.size() )`).
6. **Explanation** [Bad]: Explain a particularly tricky piece of code (e.g., `Stores state of second to last pointer address's handle`). In general, if the code is so complicated that it requires explanation, it is often (though not always) an indication that it can (and should) be improved.
7. **Repeat** [Bad]: Repeat information that is obvious to anyone with a reasonable amount of coding experience (e.g., `Initialize Cut object`). Such comments are useless, distracting, and often outdated.

See https://www.python.org/dev/peps/pep-0008/#prescriptive-naming-conventions and http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#nl-naming-and-layout-rules for "official" recommendations on naming conventions. The Google style guides may also be helpful (see sections 3.8 and 3.16 of https://google.github.io/styleguide/pyguide.html for Python, and the Naming and Comments sections of https://google.github.io/styleguide/cppguide.html for C++). Chapter 32 Self-Documenting Code of [1] may also be useful for C++.

5.3.2 Docstrings

Docstrings are less necessary if your code is not meant to be re-used, but they can still be useful to keep track of the general structure of your program. If you decide to write docstrings, it is

recommended that you write them in a way that is compatible with *documentation generation tools* that automatically generate API documentation.[8]

The most common documention tool for Python is *Sphinx* (install with `sudo apt-get install python3-sphinx` on Ubuntu). While Sphinx accepts a few docstring styles, the two main ones are the *NumPy style* and the *Google Style*. For smaller projects, the Google Style is recommended for its readability and suitability for short, simple docstrings. See https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_google.html for examples. You may also refer to the relevant section in the official Google Python Style Guide at https://github.com/google/styleguide/blob/gh-pages/pyguide.md#38-comments-and-docstrings.

For C++, *Doxygen* is a popular option (install with `sudo apt-get install doxygen` on Ubuntu). The two main docstring styles are the *Javadoc style* and *Qt style*, though they are fairly similar. See http://www.doxygen.nl/manual/docblocks.html for details on how to add documentation to source code.

TODO: Brief explanation/links on how to use Sphinx and Doxygen

5.4 Optional Content

5.4.1 Centralized VCS vs Distributed VCS

Git is not the only widely used version control system (VCS). In fact, it is a relatively new type of VCS, called a *distributed VCS*, in contrast with older types called *centralized VCS*:

- In *Centralized VCS*, there is a single (centralized) master copy of the code base that keeps track of the entire history of the code. Pieces of the code that are being worked on are typically locked, (or "checked out") so that only one developer is allowed to work on that part of the code at any one time. Access to the code base and the locking is controlled by the server. When the developer checks their code back in, the lock is released so it's available for others to check out. **CVS** and **Subversion** are two popular examples of centralized VCS.

- In *Distributed VCS*, each developer keeps a copy of the entire history of the code on their local machine. There may still be (and often there is) a master copy of the code base, but it is a purely informal designation; there is no mechanism in the VCS to mark which one is the master copy. This also means that there is no need to lock parts of the code; developers make changes in their local copy and then, once they're ready to integrate their changes into the master copy, they issue a request to the owner of the master copy to merge their changes into the master copy. **Git** and **Mercurial** are two popular examples of distributed VCS.

In general, distributed VCS has several advantages over centralized VCS (easy and fast branches, no need for network access, can look at entire history at all times) with very few drawbacks (larger space requirements, cloning can be expensive if project has long history). See https://www.atlassian.com/blog/software-teams/version-control-centralized-dvcs and https://www.teamstudio.com/blog/distributed-vs-centralized-version-control-systems-for-lotus-notes for more detailed discussions of the differences.

5.4.2 Virtual Environments (Python)

Virtual environments provide a systematic way to manage the installation of packages in Python. They can be useful when developing Python modules or when you need to use conflicting versions of modules for different projects.

[8] If you use Github and want to host a documentation page for your project on it, see https://daler.github.io/sphinxdoc-test/includeme.html for instructions.

Both Anaconda and native Python provide methods to use virtual environments:

- **Anaconda**: Use `conda create --name myenv` to create a new virtual environment, then activate it with `conda activate myenv`. See https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html for details.
- **venv**:[9] On Ubuntu, install with the command `sudo apt-get install python3-venv`. Use `python3 -m venv myenv` to create a new virtual environment. This creates a directory called `myenv` which contains a copy of the core Python installation. You can then install modules to this copy and everything will be contained within `myenv`, thus not affecting the system installation. See https://docs.python.org/3/tutorial/venv.html and https://realpython.com/python-virtual-environments-a-primer/ for details.

References

[1] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction*. 2nd ed. Microsoft Press, 2004.

[9] `venv` is only available for Python 3. If for some reason you insist on using Python 2, there is a module called `virtualenv` that can be used for Python 2 and 3, though it tends to be slower than `venv` on Python 3. You can install it via `pip` with the command `pip install virtualenv`.