Tutorial 1: Fundamentals

Ryo Kimura

December 7, 2020

1 Fundamentals

In this module, we will briefly review the fundamental practical issues of coding at a high level. Depending on how much coding experience you have, some of the material in this module may seem obvious; in that case, feel free to skip over them. On the other hand, if you feel like you are lost and need more specific instructions, refer to the links in each section or ask a fellow classmate for help. The goal is to ensure everyone has a solid foundation from which we can start discussing the more specific details of performing computational experiments.

1.1 Programming Environment

In order to develop code, you need a programming environment from which you can write code and run it. Outside of code specific to certain applications and interpreters (e.g., MATLAB, Python), **most academic code is developed on Linux**, which is a free and open source operating system that is especially suited to programming. For this reason, **it is HIGHLY RECOMMENDED that you use a Linux system as your primary programming environment**.

However, for Windows users it can be inconvenient to have a completely separate operating system just for programming. Fortunately, there are several ways of creating a virtual Linux system within a Windows host system.

1.1.1 Windows Subsystem for Linux (WSL)

The easiest way is to utilize an optional builtin feature of Windows 10: the Windows Subsystem for Linux (WSL).[1] To set up the WSL, follow these steps:

1. Go to **Control Panel** > **Programs and Features** > **Turn Windows features on or off**, check **Windows Subsystem for Linux**, and hit **OK**

   (Alternatively, open PowerShell as an Adminstrator and run `Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Windows-Subsystem-Linux`)

2. Restart your computer when prompted

3. Download and install a distro (e.g., **Ubuntu**) from Windows Store

[1] Starting from Windows 10 Version 1903 Build 18362, there are two versions of WSL: WSL1 and WSL2. While WSL2 provides major performance benefits over WSL1, it is known to cause issues with Gurobi's academic license validation. (See https://support.gurobi.com/hc/en-us/articles/360013194572-What-system-changes-can-invalidate-a-license-file- for details.) Therefore, it is recommended to use WSL1 for this course.

4. Launch distro app; create a new user account and password by following prompts

After doing this, you will have an application which, when launched, gives you command line access to a virtual Linux system.[2] See https://docs.microsoft.com/en-us/windows/wsl/install-win10 for details.

1.1.2 Updating Packages

At this point, it is recommended that you bring your Linux system up-to-date with the latest packages using the distro's *package manager*. For example, for Ubuntu/Debian this can be done with the following commands:

```
sudo apt update
sudo apt full-upgrade
sudo apt autoremove
```

The first command updates the local package database, the second command upgrades all packages, and the third command removes any packages that are no longer needed. See your distro's documentation for details.

1.1.3 Accessing Windows from within WSL

The Windows system can be accessed from within WSL via the `/mnt` directory. For example, `/mnt/c` in Linux corresponds to `C:\` in Windows, `/mnt/d/Users/alice` in Linux corresponds to `D:\Users\alice` in Windows, and so on. Using this, you can easily transfer files between the two systems (see the `cp` and `mv` commands in section 1.3.1).[3]

1.1.4 Alternatives to WSL

If you do not want to use WSL, Docker for Windows is a popular alternative that provides an isolated, platform-independent "container" for running applications. However, it is a little more difficult to set up, and some software (e.g., Gurobi) cannot be run in a container without specific licenses/configurations. See https://docs.docker.com/docker-for-windows/install/ for more details.

Another alternative is to use a *virtual machine*, which allows you to simulate a virtual computer that you can then install an operating system on. Two applications that can do this are VMware Workstation Pro[4] (available through CMU software) and Virtualbox (free and open source). While these applications will allow you more flexibility (in particular, you can have a GUI desktop as opposed to just a command line), they are somewhat slower and use more memory compared to the previous options. See https://www.cmu.edu/computing/software/all/vmware/index.html and https://www.virtualbox.org for more details.

[2] You can also use GUI applications (e.g., web browsers and image viewers) by installing an X Client (e.g., VcXsrv) on your Windows system; in this case, you need to add `export DISPLAY=:0` to your `.bashrc` and make sure the X client is running on your Windows system.

[3] While it is technically possible to access the Linux system from Windows, it is **highly discouraged** due to how easily modifying Linux files from Windows can break the Linux system. However, if you absolutely must know, the Linux files are typically stored in the User's AppData directory.

[4] Note, there are two versions of VMware Workstation: Pro (sometimes just called Workstation) and Workstation Player (somestimes just called Player). Workstation Pro is preferred since it allows you to run multiple sessions and has seamless host/guest file sharing and networking.

1.2 SSH/SFTP

1.2.1 SSH

Depending on the project, you may need to access and work on a remote computer (as opposed to a local computer like your laptop). In this case you will need to use an *SSH client*, which allows you to securely access a remote computer via a command line interface through the internet.[5] While not all remote computers are servers, in this section we will use the term *server* to refer to any remote computer that can be accessed via SSH.

Before connecting to a server via SSH, you need to have a user account on the server you are trying to access. This is set up by whoever has admin access on that server (e.g., your advisor or your course instructor). To connect to the server via SSH, you need the *host name* of the server, the *user name* of your account, and your *password*.

1.2.2 Potential Connection Issues

The first time you try to connect to the server, you may see a message that looks like this:

```
The authenticity of host 'sample.ssh.com' can't be established.
ECDSA key fingerprint is 04:48:30:31:b0:f3:5a:9b:01:9d:b3:17:38:e2:b1:0c.
Are you sure you want to continue connecting (yes/no)?
```

This message is perfectly normal, and it's just asking you to double check that the host you are trying to connect to is legitimate (for security reasons). If you were given an ECDSA key fingerprint, check that they match before continuing. Otherwise, type/select `yes` to accept the authenticity of the remote host, and you will see the following message:

```
Warning: Permanently added 'sample.ssh.com' (DSA) to the list of known hosts.
```

This just means that the server has been added to your list of known hosts, and SSH will not prompt you again to verify its authenticity.

On a different note, you may find that you can access the server when you are at CMU but not when you are at home. This is probably because the server is configured to only allow access from computers that are connected through CMU's network. If you want to access this server when you are not on CMU's network, you can use a *VPN client* to remotely "log in" to CMU's network, allowing you to access the server. One option is *Cisco AnyConnect Secure Mobility Client* (available through CMU software if not already installed). To use, connect to `vpn.cmu.edu`, select the **General Resources VPN**, and type in your andrewid and password when prompted. If you see the application icon in the notification section with a lock on it, it means you have successfully connected via VPN and you should be able to access the server from home.

1.2.3 Using SSH via Linux Command Line

The 'simplest' way to access a server through SSH is via the Linux command line, which you can do with the command `ssh [username]@[hostname]`. You will then enter your server password, and you will be connected. See https://www.ssh.com/ssh/command for details.

[5] While rarely necessary, it is possible to use GUI applications through SSH by utilizing *SSH tunneling*. X-Win 32 (available through CMU software if not already installed) is one terminal client that can be configured to do so.

1.2.4 Using SSH via Tectia - SSH Terminal

A more convenient way is to use an SSH client, e.g., *Tectia - SSH Terminal* (available through CMU software). After installation, you can press "Quick Connect" to connect to SSH as we did above. You can also set up a *profile*, which will save the username and hostname so you don't have to type it every time. To create a new profile follows these steps:

1. Go to **Profiles** > **Add Profile...**
2. If not already showing, click on the **Connection** tab
3. In **Profile name**, enter the name of the profile
4. In **Host name**, enter the hostname of the server
5. Select **Specify user name** in the **User Name** section, and enter your username

From now on, you can select this profile *Profiles* and you will only be prompted to enter your password. See https://www.cmu.edu/computing/software/all/tectia/index.html for details.

1.2.5 SFTP

You may also want to transfer files between the server and your local computer. For this you will need to use an *SFTP client*, which is separate from an SSH client.

1.2.6 Using SFTP via Linux Command Line

To run SFTP via the Linux command line, use the command `sftp [username]@[hostname]`. You will then be dropped into a special terminal window designed for remote file transfers. You can use `ls` and `cd` to navigate the remote directory, and `lls` and `lcd` to navigate the local directory (see next section for a brief description of the `ls` and `cd` commands). To transfer files, use the following two commands:

- `get FILE`: transfer `FILE` from server to local computer
- `put FILE`: transfer `FILE` from local computer to server

You can also use the `help` command to get a brief description of the various commands you can use. To exit back to the normal command line, use `exit` or `quit`. See https://www.ssh.com/ssh/sftp/ for details.

1.2.7 Using SFTP via Tectia Secure File Transfer

Tectia also comes with an SFTP client called *Tectia Secure File Transfer*. To use, simple log in to the server via the SSH client, then click on the new "New File Transfer Window" icon (a folder with an arrow on it) to open the SFTP window. You can then drag and drop files between your local computer (on the left side) and the server (on the right side). See https://www.cmu.edu/computing/software/all/tectia/index.html for details.

1.2.8 Alternatives

If you don't want to use Tectia, PuTTY is a well-established free and open source alternative. See https://www.putty.org for details.

1.3 Bash Command Line Shell

It is often useful to be able to run things directly from the *command line*, which is a purely text-based interface that allows you to run various commands/programs. Programs that provide this interface are called *shells*; while there are several command line shells in use, the one that is used by default on most Linux systems is the **Bash shell**. Like other shells, the Bash shell is a program that runs other programs/commands, in addition to providing various facilities to make this task easier (e.g., keyboard shortcuts, command history, command completion, variables, looping constructs).

1.3.1 Commands

While becoming comfortable with the Bash shell takes time, the vast majority of everyday tasks can be accomplished with just a few commands. The following is a list of the most common commands which are frequently used and a simple description of how to use them. We will follow man-pages convention and indicate optional arguments via `[optional_argument]`.

- `ls [DIRECTORY]`: display contents of `DIRECTORY` or the current directory if unspecified
- `mkdir DIRECTORY`: create a directory with name `DIRECTORY`
- `cd [DIRECTORY]`: move to `DIRECTORY` or the user's home directory if unspecified
- `cp [-r] ORIGIN DESTINATION`: copy `ORIGIN` to `DESTINATION`; use the `-r` option to copy recursively (e.g., when copying directories)
- `rm [-r] FILE`: delete `FILE`; note that deletion on Linux commandline is PERMANENT, so be careful when using `rm`!; use `-r` to delete recursively (by default you cannot use `rm` on directories)[6]
- `mv ORIGIN DESTINATION`: move `ORIGIN` to `DESTINATION`, essentially equivalent to `cp ORIGIN DESTINATION; rm ORIGIN`
- `nano FILE`: edit `FILE` using the Nano text editor; all the keyboard shortcuts you need are listed at the bottom (for example, `^X` stands for **Ctrl** + **x** and `M-/` stands for **Alt** + /)
- `man COMMAND`: display the manual pages for `COMMAND`; you can scroll through it using the arrow keys, search using `/`, and exit by pressing `q`

In addition, here are some commands that will be useful in Module 2 when we install Gurobi and CPLEX to Linux:

- `sudo COMMAND`: run `COMMAND` as an administrator
- `tar -xf TARGZ`: extract a `TARGZ` archive of the form `*.tar.gz`
- `chmod MODE FILE`: set the permissions of `FILE` to `MODE`; in general, use a `MODE` of 666 for normal files, 755 for directories, and 777 for executable files[7]

[6] By default, `rm` prompts you when you try to delete an unwritable file (e.g., a file that does not exist); if desired, you can use the `-f` option to disable such prompts.

[7] These recommendations are specifically for WSL, as a virtual Linux system on a Windows host. More common recommendations for `MODE` are 644 for normal files and 755 for directories and executables.

1.3.2 Bash Shell Features

In addition, the most important features of Bash for everyday use are:

- **Command Completion**: If you type part of a filename/command, you can press TAB and Bash will finish it for you if the name is unambiguous (if not, you can press TAB again and Bash will display a list of potential options). This is extremely useful when you are working with files and/or commands with names that are long or difficult to remember.
- **Command History**: Whenever you use the command line, Bash internally keeps a list of commands you have issued so far, and you can use the up/down arrows to search through them. This is extremely useful when you need to use a very long/complex command multiple times.
- **Environment Variables**: Bash has a basic programming language that includes variables and simple looping constructs. In particular, Bash can be configured to store information in **environment variables** whose values are accessible from the command line. This is useful for storing configuration information that is used throughout the command line (e.g., where CPLEX/Gurobi are installed). You can configure your Bash shell (e.g., setting environment variables) by editing the `.bashrc` file in your home directory (typically `/home/<username>/`).

See https://en.wikibooks.org/wiki/Bash_Shell_Scripting for details.

1.3.3 Keyboard Shortcuts

Finally, here are some of the most useful keyboard shortcuts in Bash:

- Ctrl + a : go to the start of the command line
- Ctrl + e : go to the end of the command line
- Ctrl + r : search history backwards
- Ctrl + g : escape from history searching mode
- Ctrl + c : terminate running job

See https://github.com/fliptheweb/bash-shortcuts-cheat-sheet for more.

1.3.4 Tutorials

Bash has many more features, but for space considerations we will not cover them here. See https://linuxsurvival.com for a free online tutorial-style introduction to Bash.

1.4 Python

1.4.1 Overview

Python is a programming language that is often used to quickly develop scripts and to link different applications together. It is known for its simple, straightforward syntax and an impressive collection of modules for additional features.

As of 2021, the most commonly used version of Python is Python 3. While some older projects still use Python 2, it is **highly recommended** to use Python 3 unless you have a specific reason preventing you from doing so.[8] That said, you may find that on some systems, running `python` will call Python 2 by default rather than Python 3; in such cases you can usually run Python 3 with the command `python3`.

Python has many packages that enable you to use code written by other people. If you are not using a Python distribution (like Anaconda) that has its own installation command, it is **highly recommended** to install packages with the `pip`/`pip3` command. (In particular, **do not** use `easy_install` or run `setup.py` directly unless absolutely necessary.) For some systems, you may also need to modify environment variables in order for Python to find your installed modules.

Python has one of the better official documentation pages among widely used languages. In particular, make sure you are familiar with the standard functions and standard libraries; Python is explicitly designed so that "There should be one–and preferably only one—obvious way to do it," so many common programming tasks have a dedicated module in the standard library. The following links are especially recommended if you are not particularly familiar with Python:

- **Documentation**
  - https://docs.python.org/3/: Official Python 3 documentation
  - https://docs.python.org/2/: Official Python 2 documentation
  - https://docs.python-guide.org: The Hitchhiker's Guide to Python, good for best practices and recommended libraries
- **Key Language Concepts**
  - *Name Binding*: https://mathieularose.com/python-variables/
  - *Pass-by-Object-Reference*: https://robertheaton.com/2014/02/09/pythons-pass-by-object-reference-as-explained-by-philip-k-dick/
  - *List Comprehension*: https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions
  - *Mutable vs Immutable Objects*: https://towardsdatascience.com/https-towardsdatascience-com-python-basics-mutable-vs-immutable-objects-829a0cb1530a

1.4.2 Installation

While many Linux machines already have Python installed on them, if you need to install Python yourself, there are three main options to consider:

- **Dist Package**: The **recommended** method for installing Python on a Linux machine is to use an official package provided by your distro. Different distros (e.g., Ubuntu, openSUSE, Arch) have different command line (e.g., Apt, Zypper, Pacman) and graphical (e.g., Synaptic, YaST, Pamac) package managers, so find out how to install Python for your distro. Since installing Python packages correctly can often be confusing, it is **highly recommended** that you also install pip, which is the official package installer for Python.

  Note that in many cases the packages for Python 3 will be called `python3` and `pip3` whereas the packages for Python 2 will simply be called `python` and `pip`. See https://docs.python-

[8] Note that Python 2 has been officially **deprecated** since January 1st, 2020, and has not received any updates since October 16, 2000.

guide.org/starting/install3/linux/ for details on how to install Python 3 on Ubuntu, and https://packaging.python.org/guides/installing-using-linux-tools/ for details on installing `pip` for different Linux distros.

- **Anaconda**: An alternative method is to use Anaconda, which is the standard Python distribution bundled together with additional utilities for managing Python packages. This largely simplifies the package installation process and the management of virtual environments. You can download Anaconda from https://www.anaconda.com/distribution/.
- **Official Website**: A third option is to download Python directly from the official website at https://www.python.org/downloads/. This may be useful if your (obscure) Linux distribution does not provide a Python package or if you want maximum control over the Python installation process. However, for most users the previous two options will be sufficient.

1.4.3 Running Python Code

Python is an *interpreted language*, meaning that its source code is converted into machine instructions by an *interpreter* (e.g., the *python* program) as the program is running. This means that Python applications (e.g., `example.py`) are executed either by running it via an interpreter:

```
python3 example.py
```

or by running it directly (assuming it is executable, see the `chmod` command in 1.3.1)[9]:

```
./example.py
```

1.5 C++

1.5.1 Overview

C++ is the most popular programming language used for high-performance applications. It is notable for providing relatively low-level access to hardware/memory and as one of the most feature-rich languages in common use.

Like Python, the C++ language has several "standards" (i.e., versions of C++) which have been developed over the years. The most commonly used standards are C++98 and C++11, though the most recent actively developed standard is C++20. **For this course, we will be using the C++11 standard**, due to its widespread availability on modern architectures and its updated standard library which mitigates many of the shortcomings of C++98.

Some people have the impression that C++ is difficult to use safely and correctly, due to its lax approach to memory management and vast array of features. Fortunately, compilers have gotten better at error detection and developers have established best practices that makes modern C++ programming signficantly more structured than it used to. The following links are especially recommended if you are not particularly familiar with C++:

- **Documentation**
  - https://isocpp.org/faq - unified Super-FAQ by the Standard C++ Foundation; good for best practices and gotchas

[9] The `./` is needed since Bash does not execute commands from the current directory by default.

  - https://en.cppreference.com - comprehensive reference on C++ standards; covers C, C++98, C++11, C++14, C++17, and C++20 (preliminary)
  - http://www.cplusplus.com/reference - covers C++98 and C++11, somewhat outdated/incorrect but often comes up in searches

- **Key Language Concepts**
  - *Dynamic Memory Allocation*: https://www.learncpp.com/cpp-tutorial/69-dynamic-memory-allocation-with-new-and-delete/
  - *References*: https://www.ntu.edu.sg/home/ehchua/programming/cpp/cp4_PointerReference.html
  - *Template Classes*: https://www.learncpp.com/cpp-tutorial/133-template-classes/
  - *C++ Arrays*: https://shendrick.net/Coding%20Tips/2015/03/15/cpparrayvsvector.html

1.5.2 Installation

In order to build C++ applications you need a *C++ compiler*. While many C++ compilers exist, the software that will be used extensively in this course (i.e., CPLEX and Gurobi) **only** supports the GNU C++ Compiler (GCC) on Linux systems.

Fortunately, GCC is installed by default on many Linux systems. If not, you should use your distro's package manager to install it (similar to Python). Note that the **recommended** method is to install a *development tools* package, which bundles together GCC with several other packages you need to compile C++ code, rather than just the GCC package. For example, for Ubuntu this package is called `build-essentials`, for openSUSE it is `devel_C_C++`, and for Arch it is `base-devel`. See https://help.ubuntu.com/community/InstallingCompilers and https://tanmaync.wordpress.com/2018/01/18/install-c-c-development-environment-linux/ for details.

1.5.3 Running C++ Code

C++ is a *compiled language*, meaning that its source code is converted into a *binary* that can be run directly by the machine. Since the machine does not need to use an interpreter to run the binary, it is able to make more efficient use of the system resources, which generally results in higher performance. However, it also means that C++ applications must first be *compiled* into an object file, and then *linked* with libraries on the system, before they can be executed. For example, in order to execute `example.cpp`, you need to run the following three commands:

```
g++ -c -o example.o example.cpp
g++ -o example example.o
example
```

For simple applications, the compiling and linking step can be done in a single command:

```
g++ -o example example.cpp
```

However, most C++ applications consist of many different files and use various external libraries. In these cases, the commands for compiling and linking are significantly more complicated. Fortunately, there are tools that make this process much easier (e.g., Make/CMake); we will cover them in detail in Module 5 when we talk about Coding Tools.

1.6 Optional Content

1.6.1 More Linux Commands

- `passwd`: set user passwords
- `wget`, `curl`: download things from the internet
- `top`: interactively display running processes
- `ps`: display user's currently running processes
- `kill`: kill processes; you can kill all stopped jobs with `kill $(jobs -ps)`
- `df`: display hard drive usage
- `du`: display file sizes
- `zip`, `unzip`: compress and extrat zip archives
- `grep`: search for words using regular expressions; more complex scripting tools include `awk` and `sed`
- `w3m`, `lynx`, `links`, `elinks`: browse webpages from the command line
- `cat`, `head`, `tail`: print the contents (`cat`)/the first few lines (`head`)/the last few lines (`tail`) of a file
- `pwd`: print working directory (sometimes this is useful)
- `find`: search for files fitting a pattern

1.6.2 More Bash Keyboard Shortcuts

- Ctrl+z: stop current job; useful when Ctrl+c does not work (see `kill` command in 1.6.1)
- Alt+b: move backward one word (or go to start of word the cursor is currently on)
- Alt+f: move forward one word (or go to end of word the cursor is currently on)
- Ctrl+x+x: toggle between start of command line and current cursor position
- Ctrl+k: delete from cursor to the end of the command line
- Ctrl+u: delete from cursor to start of the command line
- Ctrl+w: delete from cursor to start of word (i.e., delete backwards one word)
- Alt+d: delete to end of word starting at cursor (i.e., delete forward one word)
- Ctrl+y: paste word or text that was cut using one of the deletion shortcuts (such as the one above) after the cursor

1.6.3 Optional: Text Editors and IDEs

Since coding involves a lot of text editing, it is worth investing some thought into what you want to use as your primary editor. Depending on your programming environment and your preferences, there are three broad classes of editors to consider:

- **Command Line Text Editors**: if your programming environment does not have graphical capabilities, these are your only options. While they take some time to learn, you will always have access to them since every machine has a command line. Some recommendations are:
  - **Nano**: installed on most Linux systems, easy to use, limited functionality
  - **Vim**: installed on many Linux systems, extensive functionality, steeper learning curve
  - **Emacs**: common Vim alternative, highly extensible, awkward keyboard shortcuts
- **Graphical Text Editors**: these are general purpose text editors with nice GUIs. All of them are highly extensible and come packed with features that make truly powerful and all-purpose. Some recommendations are:
  - **Notepad++** (Windows only): the Notepad you "love" with more features
  - **GVim**: when you want to use Vim outside the command line
  - **Atom**: free and open source, originally designed for coding web apps
  - **Sublime**: consistently tops best text editors list, but official version is expensive (trial version is free but will nag you)
- **IDEs**: these are text editors designed specifically for code development. Most allow you to edit, compile, debug, run tests, and more all from the same application and easily manage projects with dozens of files. Some recommendations are:
  - **Visual Studio Code**: extremely popular cross-platform IDE from Microsoft
  - **Code::Blocks**: simple lightweight C++ IDE
  - **PyCharm**: beautiful Python IDE from JetBrains, free community version

1.6.4 Programming Environment on Windows

As mentioned in section 1.1, it is highly recommended that you use a Linux system as your primary programming environment even if you use Windows for other computing tasks. However, in situations where you need access to these tools in Windows, here are some recommendations:

- For Python, the **recommended** method is to use Anaconda, due to its virtual environment features which can help organize different programming environments with conflicting software requirements. However, some tools (e.g., the `minted` LaTeX package) require access to Python from a non-qualified terminal, in which case you either need to add Anaconda to the system's PATH environment variable, or download Python directly from the official website.
- For C++, there are two major options for C++ compilers:

- **Microsoft Visual C++ (MSVC)** is the native C++ compiler for Windows and the **only** one that is supported by Gurobi and CPLEX (on the Windows system). In this case, it is **recommended** that you install MSVC as part of **Visual Studio IDE**, which is a C++ IDE specifically designed to work with MSVC on Windows.[10] It is extremely well-supported, feature-rich, and easy to set up. In addition, the free Community version has virtually all of the relevant features of the Professional version. However, the software is huge (6.48GB) and can become quite slow for large projects. Furthermore, MSVC is typically not available on Linux systems, and you may need to modify your code if you want to compile it with GCC. See https://visualstudio.microsoft.com/ for details.[11]
- **Mingw-w64**, which is a framework for using GCC on Windows, is a very good alternative to MSVC. It is up-to-date, mature, significantly smaller than MSVC (733MB vs 4.7GB), and supported by many popular C++ IDEs. In addition, the compiler retains the ability to create native Window binaries, requiring only a handful of small DLLs. However, it is not always supported by software packages (e.g., Gurobi, CPLEX), project configuration is a bit more complicated, and the official documentation is quite out-of-date. See https://mingw-w64.org/doku.php for details (it is **recommended** to use the Sourceforge links rather than the MSYS2 links).[12]

[10] Note that Visual Studio IDE is different from *Visual Studio Code*, which is a cross-platform IDE that does not come bundled with MSVC.

[11] While it is possible to install the Microsoft Visual C++ compiler on its own (i.e., *Build Tools for Visual Studio*, link is at the bottom of page under Build Tools), there is not much benefit in doing so since the compiler alone is still quite large (4.7GB) and there are few applications that provide better integration than Visual Studio IDE.

[12] An alternative to installing Mingw-w64 on its own is to install it as part of *MSYS2*, which is a POSIX compatibility layer for building GNU software on Windows. However, with MSYS2 you gain a dependency on the msys2 DLL, the extra compatibility layer is typically not needed, configuration is more complicated, and WSL provides similar functionality while also being significantly easier to set up.