Short Guide to bash and NGS alignment and assembly

By Zhaoyang Jia

This guide is a bit dense, but from scratch, it will provide you a good amount of information needed to perform simple analysis. At least after this guide, you can understand enough to search up on the internet (StackOverflow and Biostars have a lot of highly credible individuals posting solutions to problems). Also, contact me if there's any question. I am happy to help!

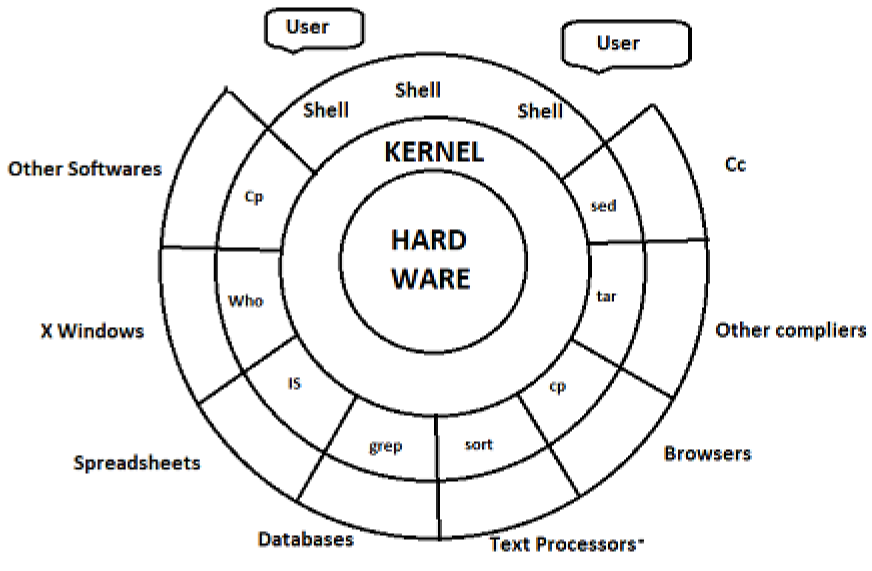
PuTTY

Most research work will provide you with server access via SSH. To establish the SSH connection, you either need to connect directly to the workplace network (wifi/ethernet), or you need to use VPN.

PuTTY is a good interface for SSH.

"Bash" Background

Bash is a "shell" scripting language, where all the shells can operate directly on top of the "kernel" of your hardware. This is more direct than your Operating System (eg. Windows, MacOS). Most servers offer Unix-based shell, which directly supports bash scripting. Learning bash scripting is essential to the usage of the server's computing power, and most of the softwares run directly from the shell scripts.



Shell-Kernel Relationship

Since the shell is operating deeper than the Operating System, the Operating System cannot catch your mistakes, thus, careless shell script may cause damage to your system. Our server is well set-uped, so your typical actions at most can break your own environment, not affecting others'. If that ever happen, the IT department can fix it for you. However, if you mess with your own machine's shell, with your administrative privaledge, you can break it.

Do not ever execute code you yet fully understand on your own machine; there are code that can irreversibly turn your computer into a brick.

Command Line

Generally, you type a line in the server, and the server will perform the task outlined by your command, and this is called the "command line". They follow the following format:

```
your_command [optional argument] <required argument>
```

- your_command: the command you want to execute
- [optional argument]: an instruction for the command, you may or may not provide it
- \<required argument>: an instruction for the command, you have to provide it
- optional argument does not always go before the required argument, but the command is always called first

For example, you want to create a directory (i.e. folder), the appropriate command's synopsis (the format) is

```
mkdir [OPTION 1] [OPTION 2] ... <directory_location>
```

`mkdir` is the command for making a directory; we will talk about it later. Let's say you are in folder_1, but you wish to create a folder_3 nested in folder_2 nested in folder_1. (i.e. ./folder_1/folder_2/folder_3, I will talk about the "path/address" format later.) The command you would use is

```
mkdir -p ./folder_1/folder_2/folder_3
```

- mkdir: command for "make directory"
- -p: optional argument for "if parent does not exist, make parent directory"
- ./folder_1/folder_2/folder_3: required argument for "directory_location"

In this case, if we do not include the "-p" optional argument, we will not be able to create our folder_3 because the parent folder folder_2 does not exist yet.

Alternatively, the preferred way to input the commandline is to create a .sh file containing all the command lines you want to do. You can execute the file using

```
sh <file_path>
```

This will act as if you were sitting in front of the computer, typing in each subsequent line after the previous line is finished executing. Using .sh file is a good practice

- It provides reporducability for your analysis, as it documents your command
- It saves time and effort. It is an automated pipeline, so you no longer need to sit in front of the computer, waiting for the previous command to finish; rather, the .sh will operate as a whole chunk, finishing the entire file of commands, without stopping in between

Notice that space in command line are used exclusively for separating different arguments, so it is a good practice to name your files without spaces (eg. use _ to replace space).

Path

"Path" is an "address" on your machine.

- ~ is the user home address, which is useful when you work on a server. Calling "~" will autofil your home address.
- / is the root address, which is the broadest location you can start on your machine

- . is the current working directory
- .. is the directory directly outside your current working directory

The path is layered via the "/" sign, so if you are in folder_3 which nested in folder_2 which nested in folder_1, which is located right in your home directory, the path would be

```
~/folder_1/folder_2/folder_3
```

This path is also called your working directory because your shell is currently "working in" this directory.

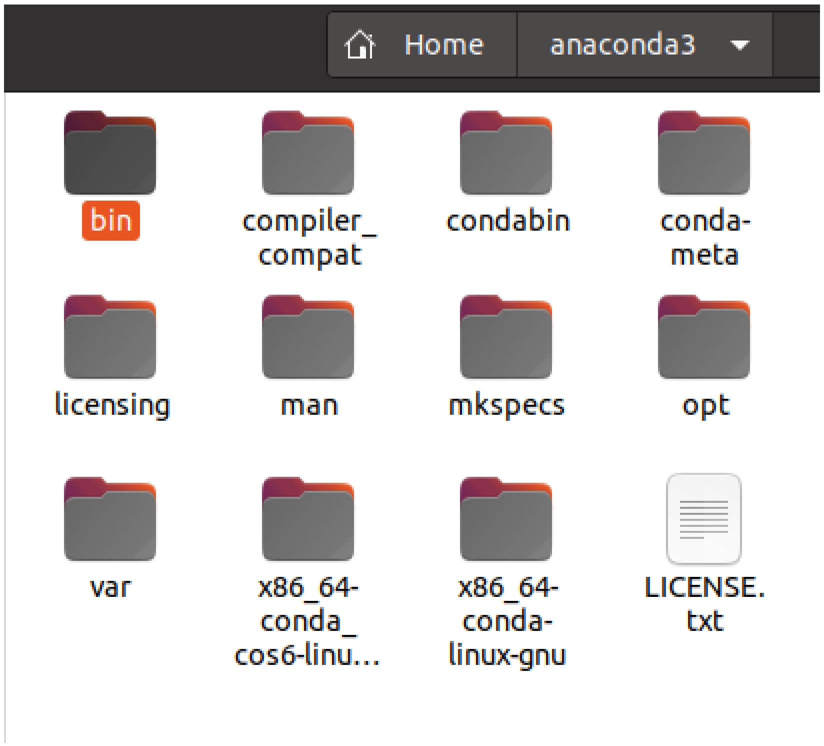
If folder_1 is in the root instead of in the home directory, the path would be

```
/folder_1/folder_2/folder_3
```

where the first / denotes root.

For example, for the image below, if we are in the "bin" directory, our path will be

```
~/Home/anaconda3/bin
```



Useful Commands

- man \<command_of_interest>: this will give you the manual of the command you are interested in learning, including its purpose, synopsis, and arguments.

For the other ones, I will not provide the synopsis. Use "man" command to find out yourself. I will occationally provide useful options.

- cd: change the current directory
- pwd: output the current directory
- mkdir: make a new directory
- ls: output the files in a directory
  - [-lh]: output in a listed format, with file size in human-readable form

- grep: search a file for a given "phrase"
    - [-c]: output the frequency of the "phrase" instead of the matching line
    - [-i]: ignore case (i.e. search all upper and lower case combinations of the "phrase")
- wc: output the number of line, word, and size of a file
    - [-l]: output only the number of lines
- mv: move a file to another location
- cp: copy a file to a location
- touch: create a file
- cat: output a file to the "standard output" (i.e. your terminal screen)
- vim: use Vim editor to open your file
    - this is like a text editor
    - the usage of Vim is a bit complicated, please refer to https://www.linux.com/training-tutorials/vim-101-beginners-guide-vim/
- scp \<source> \<target>: download or upload file, between your machine and the server
    - this will likely prompt you to enter the password for both sides, which is required for fetching and uploading the file
- sh \<file_path>: execute a .sh shell script file
- ssh: to establish connection between your machine and the server, without using PuTTY
    - you can establish connection via your own Unix shell. This is preferrable if you have a Mac or Linux system

Common Tips

- use \tab key to autofill your content (the terminal will fill in as much of the leftover content as long as it is unique)
- ctrl + c to kill an ongoing process
- ctrl + insert for copy
- shift + insert for paste
- "\" for escape, meaning the next character will be treated specially (eg. if you have a file named "file 1" with a space, you can type it as "file\ 1" and the space following the "\" will be treated as a regular space)
- again, just do not use space in file names

Special Characters

These characters are reserved for system usage; in genearl, names should not include these characters

- >:write into
    - eg. `cat file_1 > file_2` output the contents of file_1 into file_2
- >>: append onto
    - eg. if file_1 already has "de" written, and file_2 has "abc", `cat file_1 >> file_2` will result in file_2 having "abcde"
- <: input from
    - this signifies the sepecific command will take input from the following file
- *: wild-card character for "everything"
    - eg. `mv ./* folder_1` will move everything in the current folder into folder_1
    - eg. `mv ./sequence* folder_1` will move everything beginning with "sequence" in the current folder into folder_1 (sequence1, sequence2, sequence3, ...)
    - eg. mv./*.fastq folder_1 will move everything ending with ".fastq" in the current folder into folder_1

- \: will "escape" the next character, nullifying its special effect
- |: for in-line piping (ie. construct a pipeline)

In-line Piping

Creating a one-line pipe can spontaneously let a second command to take the first command's output as its input, without creating intermediate file. This saves space and can reduce messy intermediate output.

For example, `cat` outputs the content of a file, and `less` will display content on a scrolling page

```
cat file_1 | less
```

will in one step, output the file's content into a scrolling page, without the useless middle step of outputing the entire content to the screen (being unreadable).

Software for Analysis

The general pipeline for the NGS sequence analysis is as follows:

1) use FTP to download the sequencing data file\ 2) upload the data file to the server\ 3) use `fastqc` to check sequncing quality\ 4) optional: use `fastp` to perform data cleaning\ 5) use `bwa mem` to align the sequence to the reference genome\ 6) use `samtools` to convert the alignment file\ 7) use igv to visualize the alignment\ 8) use velvet to assemble your reads

1)

The FTP-download software compatible with the Scripps' database is dependent on your Operating System. The sequencing link is usually accompanied by an instruction of how to download the data. Otherwise, contact the IT department.

2)

Use `scp` command to upload the data file from local to the server. The data file should be in the form of ".fastq.gz". After uploading, use `gunzip` command to unzip the .fastq.gz sequence file in the folder you wish to work in.

```
gunzip <.fastq.gz file>
```

3)

The `fastqc` command can be used to generate an html-formated report on your sequencing quality and metadata.

```
fastqc <.fastq file>
```

4)

Note that each base of our NGS read has a base-calling confidence score, so we can target reads with low confidence to remove them or mask them with an "N" nucleotide. The `fastp` command can be used for a variety of pre-filtering and masking processes. For example, you can remove all reads that contain a certain threshold of low confident nucleotide. Or you can mask low confident nucleotide with "N", meaning they will be treated as any nucleotide in the downstream processes, preserving the rest of the information on the read, but remove the influence of the low confident portion. Please type `fastp` to see the detailed command flags.

5)

First generate a ".fa" formatted sequence file of your template, using Snapgene. Upload this sequence file to the server using `scp`. Then, you have to index the reference genome for the aligner to use.

```
bwa index <.fa reference genome>
```

This creates a .fa.fai index file, that can later be used together with the .fa genome in the alignment.

```
bwa mem <.fastq sequence file> <.fa reference genome> > <.sam output file>
```

We input our sequencing data output and reference genome for alignment. Note, the we use the path to the .fa file directly instead of the .fa.fai. The .fa file has to be in the same folder as the .fa.fai file. The result output will be a .sam file. You have to name your output file "some_name.sam".

6)

We need to convert the .sam file into its binary version for the later processes.

```
samtools sort <.sam file> -o <.bam output file>
```

This step takes in the .sam file, and the `-o` flag specifies that we wish to rename our .bam output file.

```
samtools index <.bam file>
```

The indexing step will create a .bam.bai index file, which is necessary for the visualization of the alignment.

7)

You can use igv on the server, but my experience is that using igv on your local computer is a lot easier. If you wish to use it on your local computer, use `scp` to download the .bam file, with its respective .bam.bai file. In the "genome" section of igv, select your reference genome. Then, "load file" to select the .bam file you wish to visualize.

Note, the .bam file has to be in the same folder as the .bam.bai file for igv to work.

Right clicking on the name of the alignment will allow you to chagne important display parameters such as allele frequency threshold.

8)

Velvet will use graph-based assembly method to assemble your NGS reads. The NGS reads are either single-end 100bp reads or pair-end 100bp x 2 reads (which has a fixed gap in between). The goal is to use `velveth` to generate a set of k-mers, which are sub-sequences of our indicated length out of each one of the read. For example, if your k-mer is set to 97 on a 100bp read, then, for each read, you will generate four 97bp k-mers for the assembly.

The lower the k-mer count, the less sensitive your assembly is to sequencing error, but the lower accuracy is your final assembly. Having too low of a k-mer length will result in very sort assemblies because the graph will end up having a lot of "bubbles", and these bubbles are usually randomly resolved, resulting in segmented assembly. Having too high of a k-mer length will result in us not able to assemble the graph, even when there is just one misread. For example, if you have a 99 k-mer length, the k-mer that will overlap on two sides will have

98 nucleotides of matching, and this means out of the 100bp, you have to have at least 98% accuracy, which is hard.

Generally, the higher the depth of your sequencing, the higher the k-mer value you can run with.

Please use `./velveth` to see the specific inputs they need.

Next we assemble and resolve the raph using `velvetg`. This have only a few potential parameters to work with. You can filter output by their minimum coverage and contig length.

Please use `./velvetg` to see the specific inputs they need.

My Pipeline

If you have read through the entire thing and is interested in my pipeline, I can optimize it for the server and send it to you. (My code are written to work on my own computer, so they need to be slightly modified to work on the server. However, if you are interested in using them to directly work through the entire process, I am happy to modify it and share it. I just do not want to work on it if nobody ends up using it...)

Thanks for reading.