

Introduction

As bioinformaticians, we have several platform options for running our Bioinformatics pipelines.

- Cloud platforms (e.g. AWS or GCP)
- High performance computing (HPC) servers (e.g. NSCC)
- **Remote server**
- Local workstation
- Personal laptop

Here, I document several useful tips / tools that I have used to make my working experience on our institute's remote server as pleasant as possible.



Connecting to the server with VScode

Unless you are going old-school, you will probably want to avoid using a command-line text editor for heavy bioinformatics scripting work. Personally, I use Visual Studio Code (VS Code) as my code editor, but I know others swear by Pycharm or Sublime Text so this choice seems like a matter of personal preference. What I tend to do is to keep all my bioinformatics scripts and analysis files in organised folders on the server, and then access them via remote ssh on VS Code.

To do this on VS Code on my Mac:

1. Press Command + P to open the command palette, and search for **remote-ssh: Add New SSH Host**
2. Enter the SSH Connection Command. You will need the connection port, username on the remote machine and the server IP. In the case of my University, I will also need to be connected to the University VPN to access the server:

```
ssh -p 60074 username@123.45.678.90
```

3. Then, Select an SSH configuration file to update. I add all my configurations to the default configuration file located at `~/Users/kane9530/.ssh/config`. My configuration file currently looks like this (Figure 1):  config file  Figure 1: SSH configuration file
4. Open your default configuration file and change the **Host** to a memorable name.
5. Open the command palette again and search for **remote-ssh: Connect to Host**. Click the hostname that you wish to connect to.
6. If it is your first time connecting to the remote server, VS Code will prompt you to key in the login password.
7. Voila! You are connected to your remote server! Navigate to a convenient directory of choice and start writing your scripts.

Adding an alias for establishing your SSH connection

Oftentimes, you want to ssh connect to the remote server via the command line. What can be more taxing to the mind than having to copy-paste or memorise the remote server port and IP every single time! To make my life easier, I add an **alias** to my `~/.zshrc` file (It will be a `.bashrc` file or something else, depending on your Unix shell) to create a permanent shortcut to the command. An alias is a shortcut name that references a command, filename or any other shell text.

1. To setup an alias, open your shell configuration file in an editor of choice (e.g. vim / nano/ vscode). Typically, it is a hidden file (filename starting with `.`) located in your home directory. E.g. Mine is located at `~/.zshrc`.
2. The syntax for writing an alias is as follows:

```
alias biodebian="ssh -p 60074 kane@172.12.123.45"
```

Note that there should not be any whitespace between the alias name (biodebian), the equals sign, and the string value. This is a typical "gotcha" when defining bash variables that we should remember.

1. Either reopen a new terminal or run `source ~/.zshrc` to execute the `.zshrc` file.
2. Voila! Now whenever you want to connect to your remote server (e.g. biodebian), just type **biodebian** to the command line and it will take you there.

In general, I setup aliases for commonly used complex commands.

Prettying your bash terminal

I won't lie - the default bash shell on the linux operating system doesn't look aesthetically pleasing. If you spend quite some time on the bash terminal, there may be small stirrings within your heart for a prettier interface with fancy colors, syntax highlighting and helpful display options.

You should heed that inner voice. Personally, I beautify my bash shell by installing several shell scripts from **synth-shell**. I followed this tutorial to get things working: [Linux for devices](#)

With around 5 minutes of effort, my bash shell now looks like this (Figure 2) when I login to my remote server. I get to see useful server specs like the memory usage and available storage space, and of course most importantly, everything is much prettier!

 Biodebian login interface Figure 2: Biodebian login screen

As an illustration of the practical utility of this tool, when I `cd` into a git repository, I can immediately see which branch I am working on so goodbye **git branch**! In this case (Figure 3), I am on "main". Also, hidden files are now shown by default when I run the **ls** command, instead of having to type `ls -la` each time. My fingers thank me.

 Biodebian bash prompt Figure 3: Biodebian bash prompt

Terminal multiplexing with tmux

Oftentimes, I see my colleagues working on 3 separate monitors with 3 terminals on each screen, one for each task they are working on. If that's cool with you, go for it (I once spoke to an associate professor who

really prefers having multiple terminals). I move around a lot, and do everything on my little macbook. Naturally, what I find cooler is to have everything accessible on a single terminal, rather than having to switch between 5 different terminals or having 2 tiny terminals side by side on my 14 inch macbook screen 😊

The solution to this problem is to use `tmux`. As described on the [tmux github page](#),

`tmux` is a terminal multiplexer. It lets you switch easily between several programs in one terminal, detach them (they keep running in the background) and reattach them to a different terminal.

I read the first 3 chapters of *Brian Hogan's "tmux2 productive mouse-free development"* book, which has provided me with all the commands, shortcuts and configurations that I have needed thus far to use `tmux` smoothly on a daily basis.

An example of the way I use `tmux` in my Bioinformatics work life is as follows:

- Whenever I want to run a new project, I create a new `tmux` session with a memorable name: `tmux new -s myCoolName`
- Sometimes, I forget which sessions I have available when I login to remote. In this case, I run `tmux ls`
- I may want to reattach a session that I had created from the past: `tmux attach -t myCoolName`
- Once I am done with a session, I will want to detach from it: `prefix + d`.
- If I want to remove a session permanently to clean up my `tmux` list, then I attach it and run `control + d`. In contrast to detaching the session, now the session will be completely gone and won't appear when I run `tmux ls`.
- In a typical session, I create multiple windows, one for each operation that I am performing. For example, I may have one window open for downloading the fastq files from GEO, another window for coding my script, and another running `top` to view the memory usage of the processes on my remote machine. I rename my windows as well to make the separation of tasks clear with: `prefix + ,`.
- In rare cases, within a window, I want to split it into multiple panes. Honestly, I rarely do this because I am working on a small laptop screen anyways and don't like having to squint, but I have used it in cases where I want to visualise two things side by side. For example, i may want to see all the files that I have downloaded into my directory to reference them in my script. In this case, I run `prefix + |` to have the panes side-by-side, or `prefix + _` to have them split top-to-bottom. Note that these shortcuts are different from the default `tmux` shortcuts, since I've changed my `tmux` configuration file as recommended by Brian Hogan. You can download and modify this configuration file from [hogan's config file](#).

In this way, I never, ever, ever, ever.... have to work with 10 open terminals on my laptop ever again!

Using `nohup` to run processes in the background

Sometimes, I want to quickly execute commands in the terminal and save the output and standard error logs into another file to view them after. Typically, if you have a process running, you will need to open another terminal to be able to continue writing new commands. Worse still, if you happen to shut your laptop or more commonly, experience some internet disconnection, your long-running process will get killed, which isn't fun if you have been running a read alignment program for 8 hours...

In such cases, you can use **tmux** (see above) to keep the detached session and its associated processes running in the background, but then all the standard output is sent to the terminal and not redirected to a log file.

The solution to this is to use **nohup**, which stands for **no hangup**. Its purpose is to execute a command such that it ignores the HUP (hangup) signal and therefore does not stop when the user logs out. By default, it will save the standard output to a file called **nohup.out**, but I typically want to rename it to something more sensible, especially since I may be running multiple **nohup** commands in the same directory. The command I use for this is:

```
nohup <somecommand> &> sensible_name.nohup.out &
```

This runs **somecommand** in the background and saves the standard output and standard error to a file called **sensible_name.nohup.out**.

I can also check if the command is running in the background by running **ps aux | grep kane** to check all the processes that I have running in my account.

Transferring files with RSync / http server

To transfer files from my laptop to the remote machine, I use Rsync, which is a powerful utility used for copying and synchronizing files and directories between local and remote systems.

```
rsync -avz -e "ssh -p <port-number>" \  
path/to/file username@biodebian:dir/to/folder
```

Note that biodebian is the name of my remote machine.

Regarding the Rsync options:

- **-a**: This stands for "archive" and tells rsync to preserve all file attributes, including permissions, ownership, timestamps, and symlinks. This is a useful option for copying files while preserving all their properties.
- **-v**: This stands for "verbose" and tells rsync to display detailed output about the files being copied.
- **-z**: This stands for "compress" and tells rsync to compress the data during transfer, which can reduce the amount of data transferred and speed up the transfer.

To transfer files from the remote machine to my laptop, I could use Rsync, but we set up a http server on biodebian, and so I can browse the directories and download the files directly.

```
python -m http.server
```

Final words

That's all I have to share for now. Let me know if you have other awesome tricks to level-up our remote server experience together!