Essential command line tools in HPC environments

Shao-Ching Huang

2020-08-13

Ask questions using google docs (URL in zoom chat box)
Survey of future topics: https://forms.gle/BBCD96RcVVTpbsfg7
Save the zoom chat room for announcements

Somewhat subjective

text terminal-based

vim (text editor)
git (version control)
tmux (terminal on steroids :))
...

Essential command line tools in HPC environments

(POSIX-conformant) Your lab's servers, your laptop/desktop computer, UCLA Hoffman2 cluster, cloud computing services (e.g. Amazon AWS, Google cloud), etc

POSIX tools: https://pubs.opengroup.org/onlinepubs/9699919799/utilities/contents.html

Terminology

POSIX (Portable Operating System Interface) or POSIX-conformant

Think: MacOS, Linux, Cygwin (or the like) on Windows

Command line interface

The text interface of your computer. Think: "the terminal" or "the shell"

GUI: Graphical user interface

TUI: Textual user interface

HPC: High performance computing (also think: computational and data sciences)

GUIs are good and cool, but ...

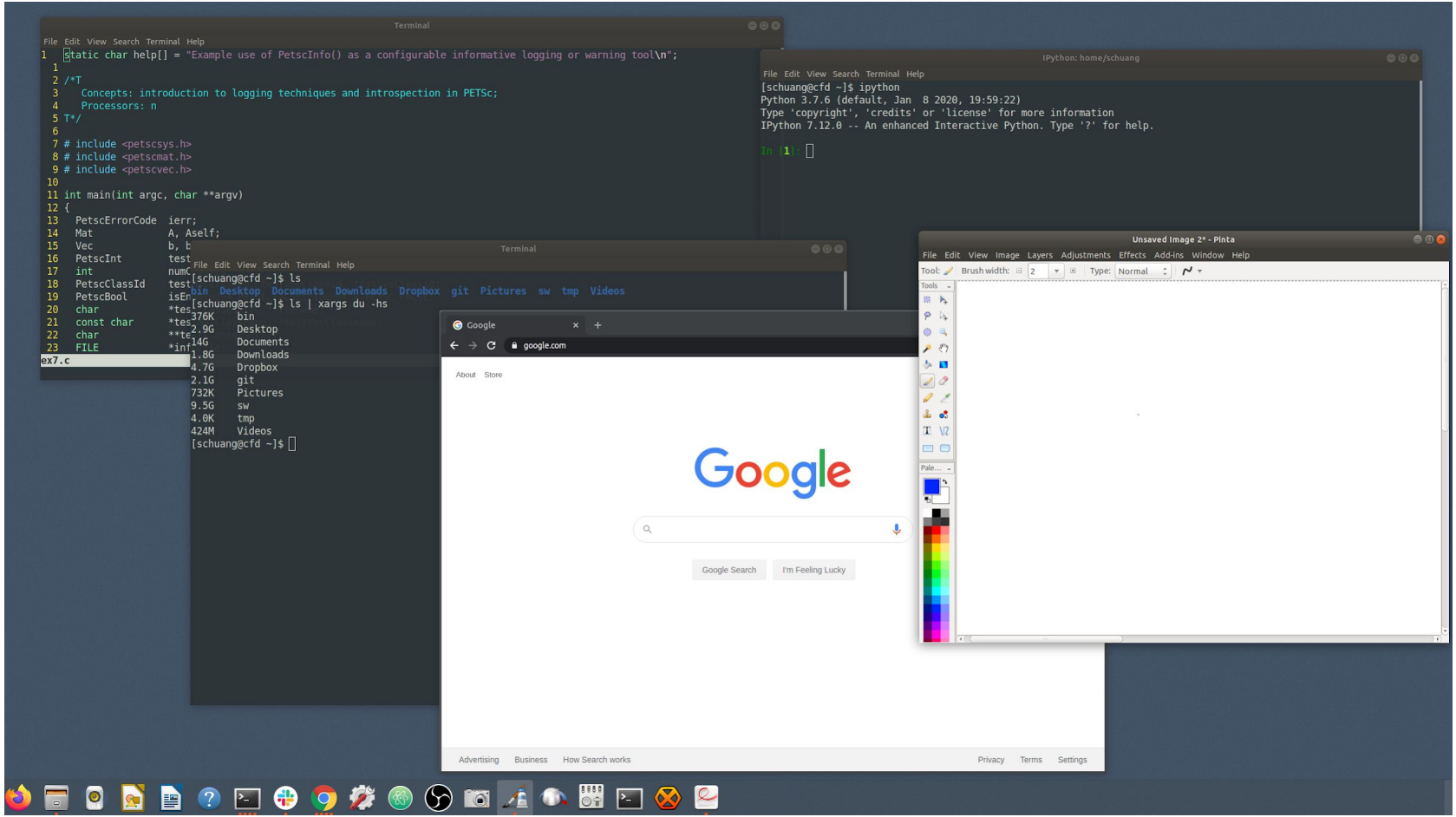
- We now live in a GUI world
  - smart phones
  - tablets
  - GUI-based applications
- GUI can be restrictive
  - What if there is no menu button to do exactly what you want?
  - simple things can become complex
- This is more so in the HPC environment
  - Programming-oriented tasks
  - Overhead of remote connection

source: apple.com

Q: How do I get a list of files that I have created?

Sometimes you have to use the GUIs, but in many cases that TUIs are more flexible and highly efficient.

GUI (with embedded TUI)



Some sources of human inefficiency

(In the context of programming-oriented tasks)

- Have to use the mouse
  - Travel time between the keyboard and the mouse
  - Shoulder/arm strain/injuries

Some GUI editors encourage you to do this

- Use only 1 or 2 fingers, or just the mouse

Using all fingers are much more efficient, like playing the piano.

image source: wikipedia.org

The big picture

- ssh (for access, file transfer, etc)
- text editors
- shell programming
- version control (git)
- other command line tools
- advanced terminal tool (tmux)

To generate your "product" (text/data files) efficiently

The techniques discussed here are applicable to any HPC environments, such as UCLA Hoffman2 cluster, your lab's computer servers, your laptop/desktop computers, cloud-based servers (e.g. Amazon AWS and Google cloud).

Of course, if GUI is more useful in certain situations, use it!

Talk overview

- ssh
  - configuration (~/.ssh/config) and use cases
- bash shell environment
  - configuration (~/.bashrc) and use cases
- vim text editor
  - configuration (~/.vimrc) and use cases
- tmux
  - configuration (~/.tmux.conf) and use cases
- git version control
  - configuration (~/.gitconfig) and use cases
- Putting things together (demo)

A word for Microsoft Windows users

Microsoft Windows is not POSIX-conformant, but you can simulate it by considering one of the following approaches:

- Windows 10 Linux subsystem
- Cygwin (https://www.cygwin.com/)
- Mobaxterm (https://mobaxterm.mobatek.net/)
  - not exactly POSIX-conformant, but very easy to install to get started

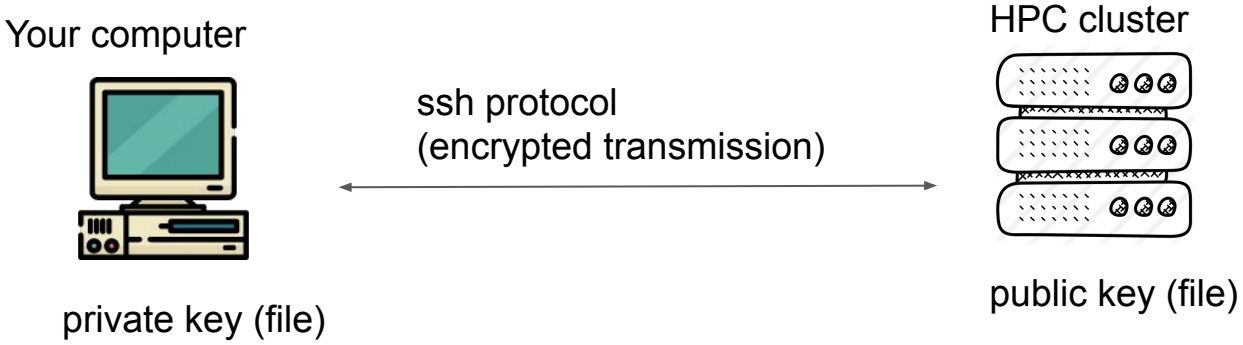
ssh: secure shell

ssh: secure shell

- A secure data transmission protocol
  - The transmitted data is encrypted
- A client-server model
  - Your laptop/desktop computer is the client
  - The remote HPC cluster/server is the server
- In practice, this is the tool used to "log in" onto a remote server/HPC cluster
- Commands:
  - ssh, sftp, ssh-keygen and more
- For example, Hoffman2 cluster (and many other servers) use ssh as the login mechanism

ssh key pair authentication

ssh has multiple authentication methods: password, key pair, etc.



You can use the key-pair mechanism to log in, or running commands remotely "without logging in".

Setting up ssh key pair

- Command: `ssh-keygen`
  - Produces two files: a private key and a public key
- The private key stays on your computer
  - Under directory ~/.ssh
- The public key is copied to the remote server
  - Appended to ~/.ssh/authorized_keys
- The priv/pub keys work as a pair (must match)


```
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDkPGrgxI6sHbBTIY3DhWPhnp
3ggei/njvQ8rRxyhZ68fFtWtUBwNrWIIHELNCQviV5RPiC03EcAXIcBgUuSWvZ
8/XAwcjW5e35YydxNruVPcS[illegible]BR11p61Xnr/acQcGwfozXnF
0kmTUDf1Qhsw/LVxca7vbNi[illegible]FqoMydUx1FjSulAFJ/9OEqO
GBhYAlVfV6u1r1pnXuQb6UtOrXJFZtxXHjLy3Pci/HRNQtVuEHJUHIvusx2pki
O8MLEu2L/cb1T632FFHhARUa08eUJiWeREp2yVCWBot7NC7pAAbfeC5uAuNaJl
oc2v6Fnt key_pair_test
```


public key

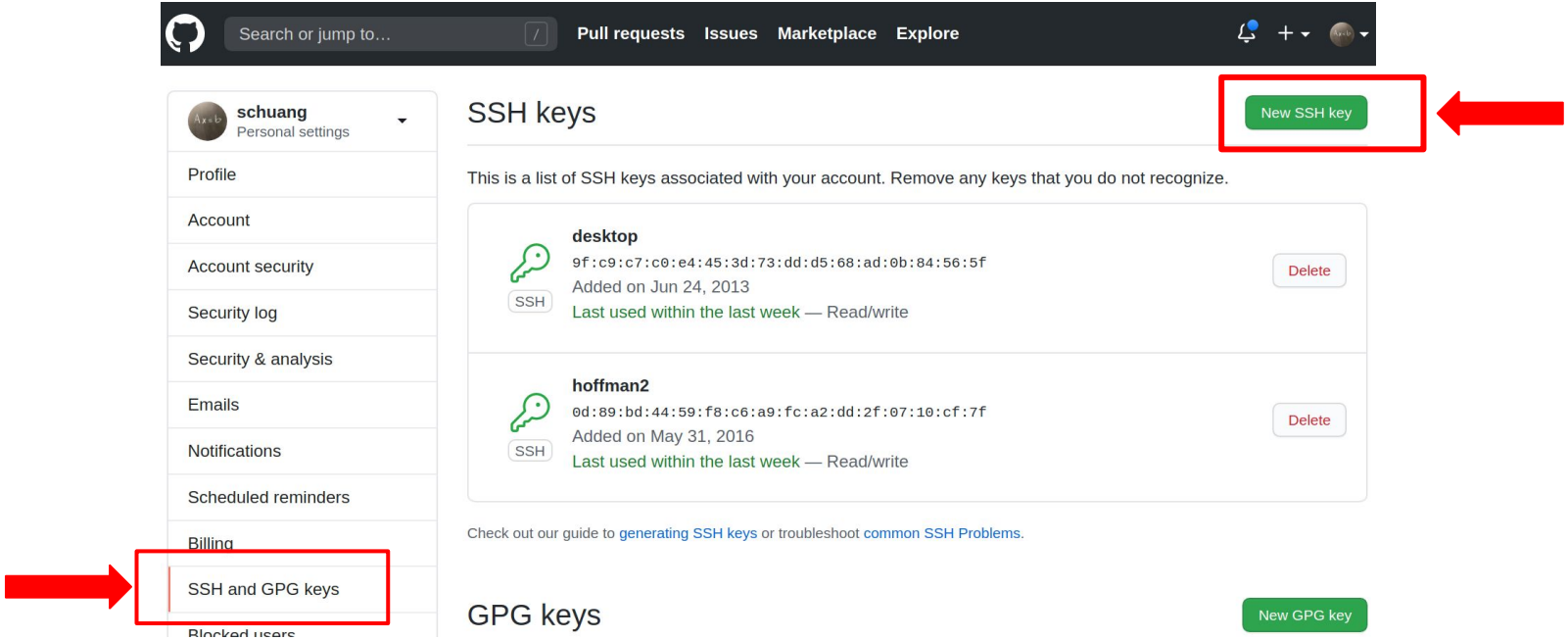
comment (arbitrary texts)


```
-----BEGIN RSA PRIVATE KEY-----
MIIEowIBAAKCAQEAnKFXF5y9eQgrpRJB0Fl33RKxuKPCKzCt2IISA0ljjB0PdUZa
s0nT1hmglFU9ilhyCAJRw8ueujseK3SVxrI2YLjPesIHA3QM8QZ1/bVSu3mZH4oa
NPxlN1QHEBwf5zMv0x/xkoGQ2MKQA88rdOxCuCL710uqpK2RmjN3gMyjAHKeXsIR
C5s+4QERfwPNU2rURIuFlEzrJoCDi5cPeG4wkNzeJ6nbuaKFh3b6RscIyrnD7gf0
aQAokrjkQWn9ZRuzNGJ4SldkTCjXRKDzFlY+4xc9IXtVIcfuwwg0V8TdlXZlal/Y
c4UXqbigJNhRXcLhK+d1zc9vLCX5JqHkc8fkIQIBIwKCAQEAk64QSXaGwpKsztaz
FOakNtccmCVfTUsnkZ81jhH+xe+DmnWIt6wCPuT2fTptOUwTvmiWPEOkSTBs5yTH
u1eSWzHoMeoj3q8/ZujVphFcoiIxSZ93y47jFujTdZbcQGN2PBbN00b2SLdx17Sz
8eYTBWLQMWSZk/QbkWO50TX4xebvjX1DxcrLaIm8JDv5l8lOd8r+h1UuSEt0rn4x
nCMpMycmhSeovfgw74sBRq[illegible]AAXjdAy5pjg4FH/4x12
ovUxnluyl6/zTORwEI0P/C[illegible]H5tuJ6N5ngucSCMl+gd
al/u5wKBgQDQHMg7I2+w0D[illegible]dO3Zq9k7QdgFttXgFW1
ttPdjNGYrzANMdhYc5ElQxTLGANqG3B7B+LRwN3YoMFyDQ5wnd7rrL8xf8t1RAOD
0EUL+6o5LImU+IgkQJmXlnTHXYWAw35jD1Zsp20KqG3PMKDmCJWOIwKBgQDAq+lX
uUy2VFKFyF/FZd30I5kOQ43ZVbXGSDW+mrDGuygXJn2qd5MnjMIbX340iadbrCJF
phcTBInPX9RNqsJlArJZR1OF5Ea6q4qIhdBcpak28UqYAJQTH0TRlnaCyGzGxDJe
bGJoGykMJ9yhC83qtdlbYH4UOkgQ98zAMK2O6wKBgQDKKpaf2UftlxU46Z/Vj8CJ
UpL3v+cOnZkdugDeC/rXEiRr+PJVAFNCztxw0fAt7ALK/TiQcEPbDfbxLUUlPzoR
HZqCnhlWGIFnemXMfBphSLm7Dm3foSf8Y+tNeCGz4h9AQeNWaqPVFdCHJ6ZCkgW/
ULpa68kDCgRFnPQSqUDTNwKBgQC1qWb68IooQOAXvOyVi+6HGj/aP7GhB6tjLiQL
e+iA3Fj/4nZ8J5lfzduWJtYUR0YF+hGvYhXBeU68Ncgr+MXqNb4ZqaaFjhbNQqcw
Q6cyyBSS44DYdZLtdUDi3lJ7T0H9PKSE6dkgVB9jO4bg/IBSU6+t7UOsq/rOK3fo
aGkgWQKBgH32MnJfBzxPB5tRyLOwSH8Ngt32cJLoNM+KL2L6HCOdXdpL7OlLhfTv
hzA4CpwlGaFELogyVQXMQ08Drq0gZJHRgFUSJizxOXBeiM0x5XYzE2PnnFHobBsE
cew1gXoyKUpZvIyVeWtyVaAMuYb3lbo40PRDcC55jdCgUziRnxk9
-----END RSA PRIVATE KEY-----
```


private key

ssh key pairs are not only for server logins

github.com, gitlab.com and other hosting services use it too:



Screenshot from https://github.com/settings/keys

Managing multiple key pairs with ssh config file

- Different servers may use different key pairs
- ~/.ssh/config defines which server to use which private key file

```
Host github.com
   IdentityFile ~/.ssh/id_rsa_github
   User git

Host gitlab.idre.ucla.edu
   IdentityFile ~/.ssh/id_rsa_gitlab
   User git

Host login1
   Hostname login1.hoffman2.idre.ucla.edu
   User xxx
   IdentityFile ~/.ssh/id_rsa_test
   ForwardX11 yes
   IdentitiesOnly yes
   ForwardX11Trusted yes
```

The server must have a matching public key file (placed in the correct location)

Use your actual user name on the **server**

The private key file

Other options for the connection

Demo: using `ssh-keygen`

Other technical details (e.g. file permission) are found at:

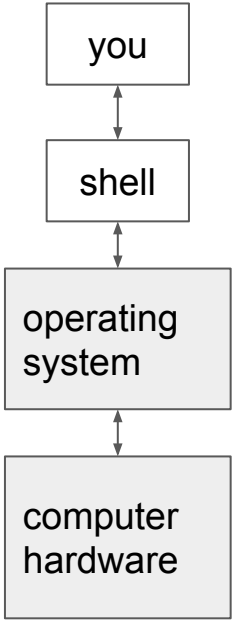
https://github.com/schuang/essential-command-line-tools-tutorial/tree/master/ssh

The shell environment

The components of a shell

- Shell prompt
  - Can be customized to display certain information
- A programming environment
  - Define shell variables
  - Run the shell commands (the shell's programming language)
  - Execute commands or executable files on the file system
- The shell initialization file(s) control the default behavior
  - e.g. ~/.bashrc etc.
- Infinite flexibility
  - Multiple commands can work together via shell piping
  - Can compose arbitrary workflows by executing a series of commands/executables

A Bash shell running inside a terminal



Shell variables

- One can define variables in the shell, just like in a programming language
- Some important shell variables, aka **environment variables**, are important in determining the shell's behavior
- Scenario:
  - Executable `foo-test` depends on the shared library `~/demo/foo-lib/libfoo.so`, which is at a non-standard location
  - Errors occur when the operating system cannot find the shared library
  - One can fix this by adjusting the shell variables `PATH` and `LD_LIBRARY_PATH`

```
$ foo-test
foo-test: error while loading shared libraries:
libfoo.so: cannot open shared object file: No such file
or directory
```

Demo: shell environment

Environment variables influencing program execution:

`PATH`	finding executable files
`LD_LIBRARY_PATH`	finding shared libraries

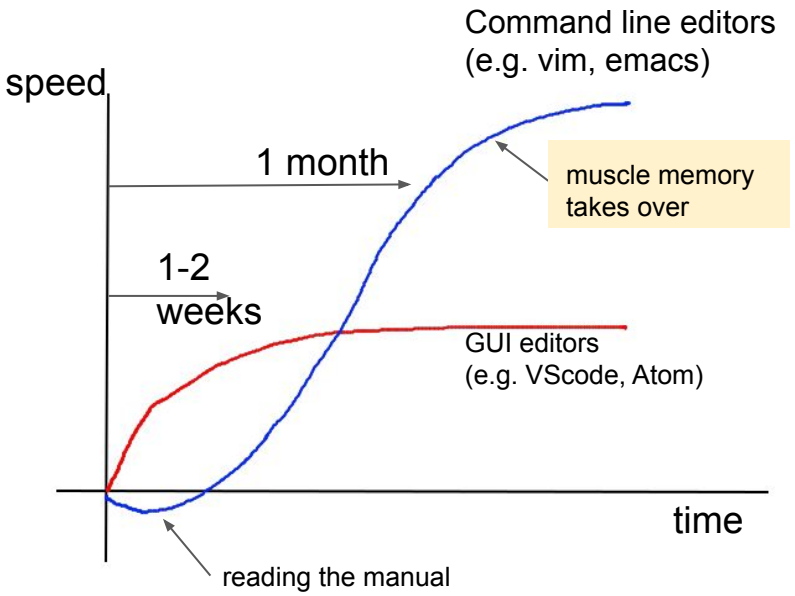
...

vim: text editor

Not word processor (e.g. Microsoft Word)

Text editors

- We spend a lot of time editing texts
  - code, scripts, text files
  - writing structured notes/reports
  - must be version controlled!
- If you invest time to become good at it, you will save a lot of time in the years to come



A few weeks of getting used to vs. time saved over the years of your graduate school or career

Open-source text editors

For example, on Hoffman2 cluster:

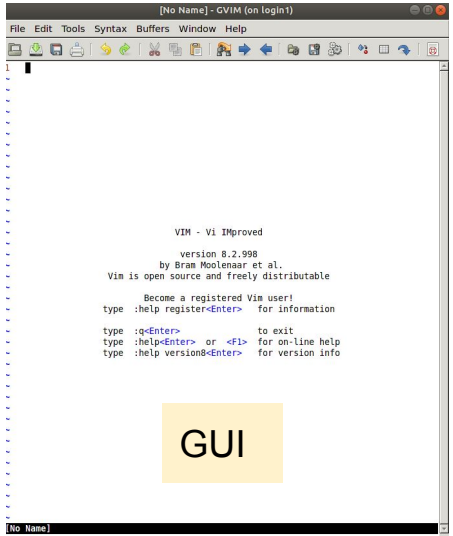
- nano
  - Simple to use
- emacs
  - Very powerful
- vim
  - Very powerful

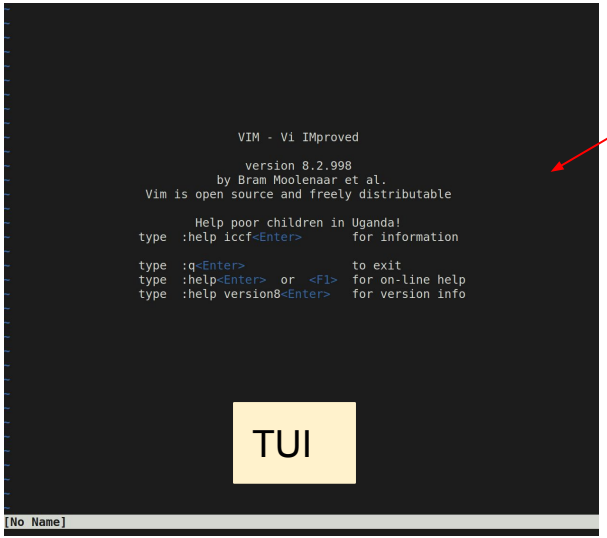
Also have GUI versions, but we will focus on the TUI versions

Note: You can install these (free) powerful text editors on your laptop/desktop computers. You don't need to learn different editors in different environments.

vim (text editor)

- Powerful, particularly good for programming-oriented tasks
- Online tutor: `vimtutor`

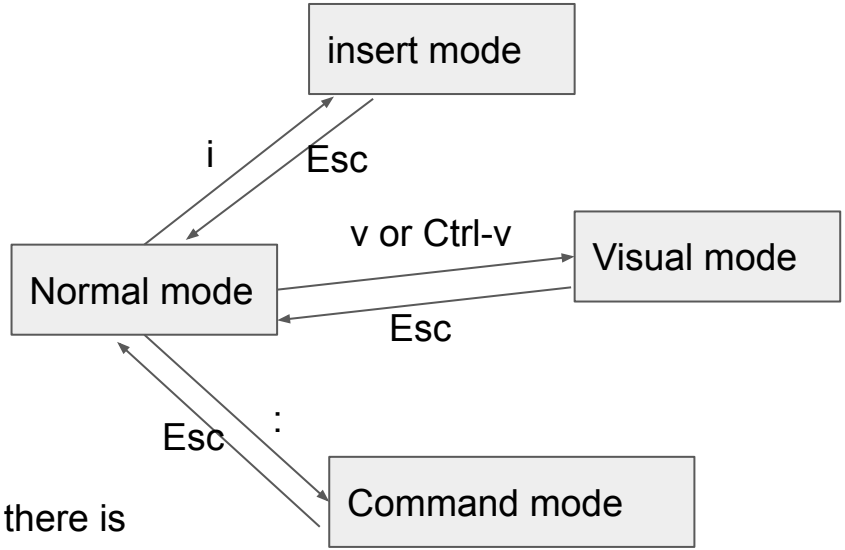




Today's focus

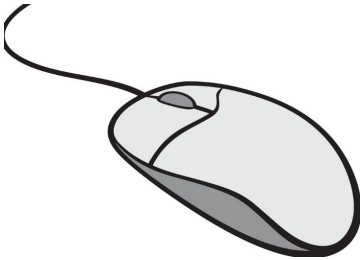
vim has multiple modes

- Normal mode: navigating the files, moving around
- Insert mode: enter texts
- Visual mode: for selection
  - line-oriented
  - block-oriented
- Command line mode
- There are several other modes



This is different from most GUI editors where there is only one mode.

vim uses h,j,k,l keys to move around



All fingers on the main keyboard, reducing time to repeatedly travel to the arrow keys and the mouse.

There are many other key combinations to more quickly move around.

Demo:

vim modes and basic editing

Demo:

browse large code base using vim

"tags" are pre-generated by ctags and loaded into vim.

Press `Ctrl-]` to go to a target (push into the tag stack)
Press `Ctrl-t` to return to the previous location (pop the tag stack)

git: version control

git: version control

- Reproducible computational and data sciences
  - Code vs data
  - Must have exact revision control in order to track changes, errors, discrepancies, etc.
  - Should be in everyone's workflow
- Free hosting services available
  - github.com, gitlab.com, and others
- Powerful command line interface
- It is not as intimidating (and mysterious) as it looks if the basics are understood

See also IDRE class: Version control using Git
https://idre.ucla.edu/calendar-event/version-control-with-git-3

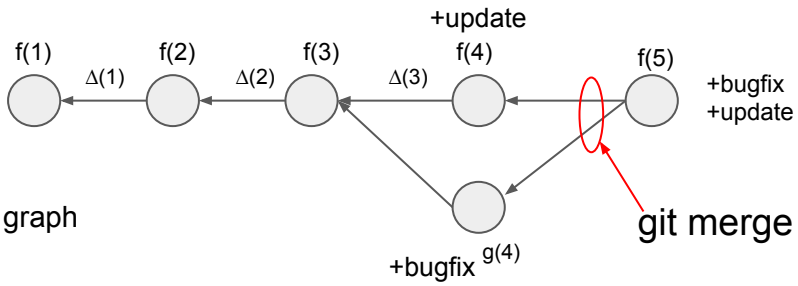
git: basic structure

- tree: folders (or directories)
- blog: files
- file system state ("snapshot")
- commit
  - the difference between two states f(n) and f(n+1) is $\Delta(n)=f(n+1)-f(n)$
  - other metadata (author, date, etc)
- objects are identified by hashes (unique identifiers)

```
.
├── .git
│   ├── branches
│   ├── COMMIT_EDITMSG
│   ├── config
│   ├── description
│   ├── HEAD
│   ├── hooks
│   ├── index
│   ├── info
│   ├── logs
│   ├── objects
│   └── refs
├── hello.py
└── README.md
```

git stores the commits in .git directory (`git log`).

Think: snapshots are constructed from the commits (`git checkout`).



A directional graph

Using vim for git merge conflict resolution

Consider the repository:

```
* commit d9f2cb3 (HEAD -> master, origin/master)
| Author: Shao-Ching Huang <sch@ucla.edu>
| Date:   Tue Aug 11 21:02:00 2020 -0700
|
|     rename func1 to f1 (master)
|
| * commit daa1271 (origin/branch1, branch1)
|/  Author: Shao-Ching Huang <sch@ucla.edu>
|   Date:   Tue Aug 11 21:01:24 2020 -0700
|
|       rename func1 to func_1 (branch1)
|
* commit bf680b7
  Author: Shao-Ching Huang <sch@ucla.edu>
  Date:   Tue Aug 11 20:59:24 2020 -0700

      first commit (master)
```

File system states

master

```
def f1():
    print('func1')

def func2():
    print('func2')

def func3():
    print('func3')

def main():
    f1()
    func2()
    func3()

if __name__ == '__main__':
    main()
```

branch1

```
def func_1():
    print('func1')

def func2():
    print('func2')

def func3():
    print('func3')

def main():
    func_1()
    func2()
    func3()

if __name__ == '__main__':
    main()
```

potential merge conflict !

Using vim for git merge conflict resolution

```
def f1():

    print('func1')

def func2():
    print('func2')

def func3():
    print('func3')

def main():
    f1()
```

LOCAL

./hello_LOCAL_18519.py

```
def func1():

    print('func1')

def func2():
    print('func2')

def func3():
    print('func3')

def main():
    func1()
```

BASE

./hello_BASE_18519.py

```
def func_1():

    print('func1')

def func2():
    print('func2')

def func3():
    print('func3')

def main():
    func_1()
```

REMOTE

./hello_REMOTE_18519.py

```
<<<<<<< HEAD
def f1():
||||||| merged common ancestors
def func1():
=======
def func_1():
>>>>>>> branch1
    print('func1')

def func2():
    print('func2')

def func3():
    print('func3')

def main():
<<<<<<< HEAD
    f1()
||||||| merged common ancestors
    func1()
=======
hello.py
"hello.py" 28L, 370C
```

MERGED RESULT

LOCAL

```
* commit d9f2cb3 (HEAD -> master, origin/master)
| Author: Shao-Ching Huang <sch@ucla.edu>
| Date:   Tue Aug 11 21:02:00 2020 -0700
|
|     rename func1 to f1 (master)
|
| * commit daa1271 (origin/branch1, branch1)
|/  Author: Shao-Ching Huang <sch@ucla.edu>
|   Date:   Tue Aug 11 21:01:24 2020 -0700
|
|       rename func1 to func_1 (branch1)
|
* commit bf680b7
  Author: Shao-Ching Huang <sch@ucla.edu>
  Date:   Tue Aug 11 20:59:24 2020 -0700

      first commit (master)
```

REMOTE

BASE

The window after running:

```
$ git mergetool
```

Pick and choose

```
:diffg R
:diffg L
:diffg B
```

`]c` and `[c` to move between changes

Demo:

Resolving git merge conflict using vim

Using vim for interactive git rebase

Consider the branches

```
* e6c1e51 (origin/feature1, feature1) hello3: update 4 (feature1)
* 5d2c675 hello3: update 3 (feature1)
* cd931e6 hello3: update 2 (feature1)
* bfe0bae hello3: update 1 (feature1)
* 38d1e5a add hello3 (feature1)
| * fa26317 (HEAD -> master, origin/master) hello2: update 2 (master)
| * 7115495 hello2: update 1
| * dcc5f08 add hello2.txt (master)
|/
* a58b577 hello: update 1 (master)
* dcdf817 add hello.txt
```

From a58b577, feature1 branch is created, followed by multiple commits. At the same time, master continues to evolve.

Direct merge results in:

```
*   b0293b4 (HEAD -> master) Merge branch 'feature1'
|\
| * e6c1e51 (origin/feature1, feature1) hello3: update 4 (feature1)
| * 5d2c675 hello3: update 3 (feature1)
| * cd931e6 hello3: update 2 (feature1)
| * bfe0bae hello3: update 1 (feature1)
| * 38d1e5a add hello3 (feature1)
* | fa26317 (origin/master, origin/HEAD) hello2: update 2 (master)
* | 7115495 hello2: update 1
* | dcc5f08 add hello2.txt (master)
|/
* a58b577 hello: update 1 (master)
* dcdf817 add hello.txt
```

What if I want to "collapse" these commits into one (squash), and put them on top of the master branch (rebase) to have a cleaner commit history?

Demo preview

Direct merge

```
*   b0293b4 (HEAD -> master) Merge branch 'feature1'
|\
| * e6c1e51 (origin/feature1, feature1) hello3: update 4 (feature1)
| * 5d2c675 hello3: update 3 (feature1)
| * cd931e6 hello3: update 2 (feature1)
| * bfe0bae hello3: update 1 (feature1)
| * 38d1e5a add hello3 (feature1)
* | fa26317 (origin/master, origin/HEAD) hello2: update 2 (master)
* | 7115495 hello2: update 1
* | dcc5f08 add hello2.txt (master)
|/
* a58b577 hello: update 1 (master)
* dcdf817 add hello.txt
```

Rebase and squash

```
* dab8c0b (HEAD -> feature1) This is a combined commit message
* fa26317 (origin/master, origin/HEAD, master) hello2: update 2 (master)
* 7115495 hello2: update 1
* dcc5f08 add hello2.txt (master)
| * e6c1e51 (origin/feature1) hello3: update 4 (feature1)
| * 5d2c675 hello3: update 3 (feature1)
| * cd931e6 hello3: update 2 (feature1)
| * bfe0bae hello3: update 1 (feature1)
| * 38d1e5a add hello3 (feature1)
|/
* a58b577 hello: update 1 (master)
* dcdf817 add hello.txt
```

To be deleted

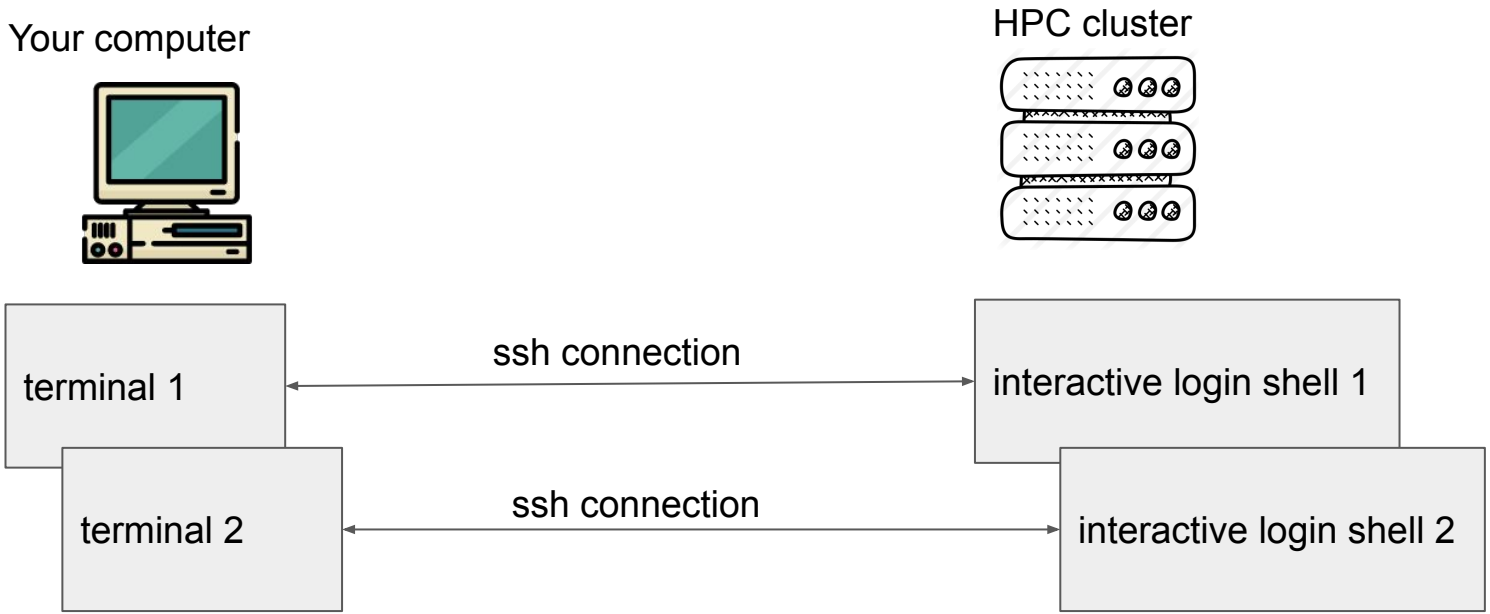
feature1's parent is a58b577

feature1's parent is now fa26317

Demo: git rebase with vim

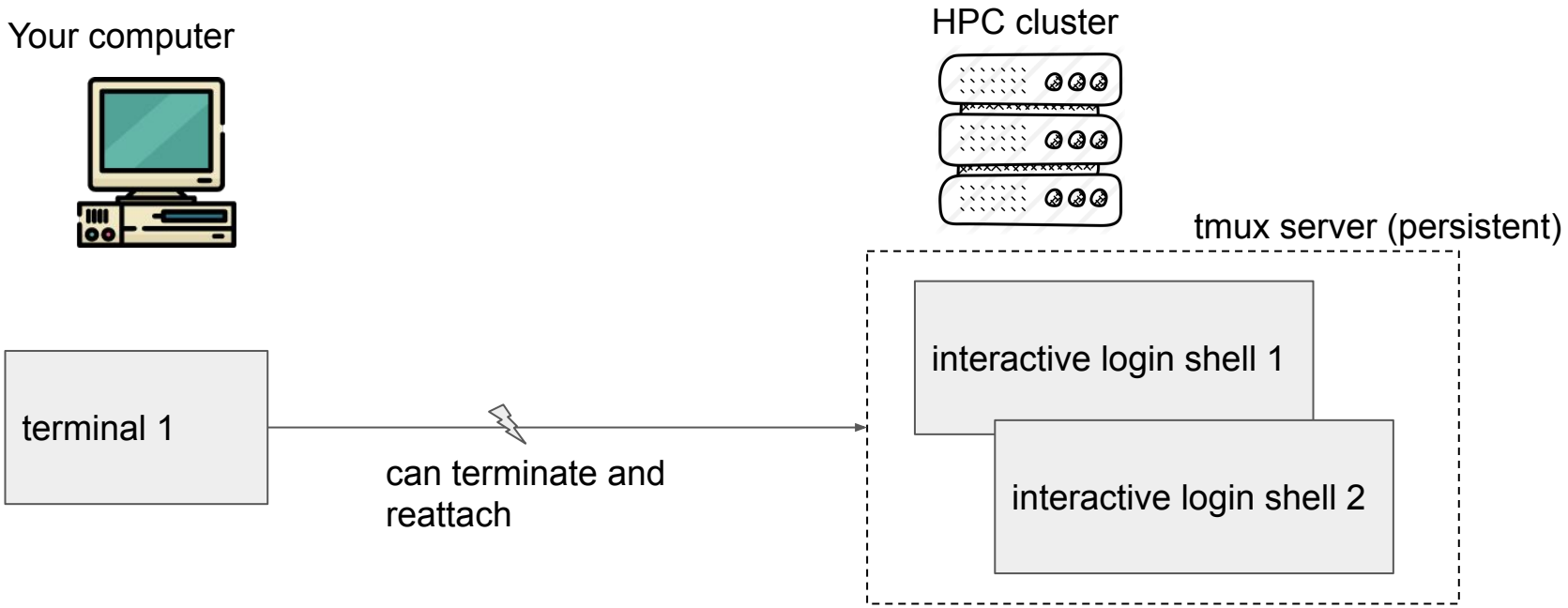
tmux: terminal multiplexer

Standard ssh connection



If the connection is interrupted, or local computer shut down, all sessions are gone.

ssh connection with tmux



tmux

- On Hoffman2 cluster:

```
$ module avail tmux
$ module load tmux
```

- Configuration file
  - ~/.tmux.conf

```
sch@login1:~
$ uptime
 00:36:40 up 237 days, 10:29, 69 users,  load average: 0.22, 0.60, 0.56
sch@login1:~
$ tmux ls
login1: 6 windows (created Wed Aug 12 11:32:44 2020) (attached)
test: 2 windows (created Thu Aug 13 00:35:10 2020) (attached)
sch@login1:~
$
```

```
sch@login1:~/demo
$
```

```
sch@login1:~
$
```

```
[test] 1:split-demo* 2:bash-                    "login1" 00:40 13-Aug-20
```

split panes

session

multiple tabs (windows)

Demo: tmux

- basic operations
- configuration file
- detach and reattach

Summary

- Command line tools have many advantages over GUI tools for programming-oriented tasks
  - Precise, fast, robust and lightweight
  - Daily computational and data analytical workflows
- Tools can work together to accomplish complex tasks easily
- Fully utilize the strength of the keyboard (and all fingers!) to achieve speed
- Once mastered, more time for your research
- Same interface for years to come, and you only get faster and better
- No need to learn different GUI tools
- We have only scratched the surface of some of these tools
- Of course, GUIs are still useful for certain tasks

Thank you!

Please fill out and submit the survey 🙏
(Link in zoom chat box)