Lab 1 - Getting Started

May 2017

Michael Benjamin, mikerb@mit.edu
Henrik Schmidt,henrik@mit.edu
Department of Mechanical Engineering
MIT, Cambridge MA 02139

主頁面：
mac起手式
http://oceanai.mit.edu/ivpman/pmwiki/pmwiki.php?n=Help.MacPorts

Getting the MacPorts
http://www.macports.org

Add the MacPorts bin directory to your shell path

port search cmake
找有沒有這個package

sudo port selfupdate
更新package list

Warning: Install them by running 'xcode-select --install'.
install Xcode

Download packages needed for our labs
一堆independence要載ＱＱ

1 Overview and Objectives

- Goal - Make sure everyone has a suitable machine for the workshop
- Goal - All students able to build and run the workshop software
- Overview of the Course Web Pages and Support
- Getting Started with the Command Line
- Downloading, Building, Running the Course software

2 Linking the Course Web Pages

We recommend putting these pages in your browser's bookmarks:

- *MOOS-IvP Documentation*: http://oceanai.mit.edu/ivpman

This lab is also online in wiki format at the above link. The wiki pages are generated from the same content (LaTeX) that generated the PDF or hard-copy version of the the labs. This is done as a matter of convenience and for later labs where there may be videos embedded in the labs.

3 Getting Started with the Command Line

Most of the exercises in this lab are given from the perspective of the command-line user. If you're not familiar with the command-line, this is a powerful mode of interfacing with your computer that, for many, *far* surpasses the efficiency afforded through a pure point-and-click GUI experience. For Mac users the command line comes by way of the Terminal application. For GNU/Linux users, you are very likely already accustomed to opening a console or shell window.

The below couple sections help absolutely new users get started a bit. Just enough so we can reach the goals of our lab. Much more information can be found on the web, and there are a few help pages related to the command line on the workshop help page:

http://oceanai.mit.edu/ivpman/help

Finally let's not overlook help already on your system. For any given GNU/Linux command, `foobar`, try one of the below to get further information.

```
$ man foobar
$ foobar --help
```

3.1 Command Line Quick Survival Basics

The Terminal, i.e., the command line environment, offers an incredibly powerful set of tools out of the box. These tools are readily extendible through the addition of countless open source additions, as well as through additions made by you, in this course and hopefully well afterwards. To get started the below list represents the bare bones basics for someone just getting start. This list

represents my own perspective, but you can find similar material by just doing a web search on "command-line basics".

Here we use the terms *Terminal* and *command line* interchangeably. This reflects a Mac user's perspective since the *Terminal* application is the most common way to interact with the command line. There are other tools, e.g., iTerm, xterm, that other Mac users find preferable. If you're a GNU/Linux user, you're more likely to refer to the *Console* instead of the terminal.

Let's get started.

3.1.1 Viewing the contents of your directory with `ls` command

The command line offers just another way of navigating through the same file system you might otherwise navigate with GUIs such as Finder. Perhaps the most commonly used command is `ls` which just displays the contents of present directory:

```
$ ls
mail   project_fango   public_html   report_a.pdf   report_b.pdf   test.txt
```

The `ls` command comes with a bunch of *command line options*. These are extra things you can type on the command line as arguments to the command being invoked. For example, in the above, six items are reported. It turns out that `mail`, `project_fango`, and `public_html` are directories, i.e., folders, containing other files or directories. This is hard to discern from the above output. But if you use a simple command line option, "-F", you'll be able to see this right away:

```
$ ls -F
mail/  project_fango/   public_html/   report_a.pdf   report_b.pdf   test.txt
```

Note all directories have the trailing "/" character. I also use `-G` to make the directories show up in color. This can be done by typing `ls -F -G`, or more simply, `ls -FG`. There are lots of useful options worth exploring when you get a chance. Just type `man ls` to see some options.

Note also that arguments provided on the command line are separated by spaces. This is why most command line users *never* name files with spaces. Note the names `report_a.pdf` above uses the underscore character instead.

3.1.2 Knowing what directory (folder) you are in with `pwd`

Using the command line, you are always "in" a particular directory. Remember, command line users tend to refer to a directory where other users might refer to a folder. To know what the "present working directory" is, simply use the "pwd" command with no arguments.

```
$ pwd
/Users/mary/lab_one
```

This is such a useful piece of information to have that many users customize their command line prompt (the dollar-sign at the left) to show the present working directory. As an exercise, google

"customize command line prompt" and see if you can set this up.

3.1.3 Changing directories with the `cd` command

Now that you know how to tell what directory you're, you might be wondering how to change directories, moving around the overall file structure. To do this we use the `cd` command. The most common argument is the full path name of the directory you want to go to, as in:

```
$ cd /Users/mary/lab_one
$ pwd
/Users/mary/lab_one
$ cd /Users/bob/projects/lab_three
$ pwd
/Users/bob/projects/lab_three
```

There are a few handy shortcuts that should be in your toolbox right away. The first is for changing to the directory right above your current directory. For example, suppose you are presently in the directory /Users/bob/projects/lab_three, and you wish to move up to the directory /Users/bob/projects. You could do it the hard way:

```
$ pwd
/Users/bob/projects/lab_three
$ cd /Users/bob/projects
$ pwd
/Users/bob/projects
```

Or you could do it the easy way, replacing the third line above with the simpler version below:

```
$ pwd
/Users/bob/projects/lab_three
$ cd ..
$ pwd
/Users/bob/projects
```

The `../` notation is an actual "directory" in every directory in your file system. This special shorthand name has been around forever. Another shorthand name is `./` which stands for the current directory. Because both directories begin with a dot, they are also known as *hidden* directories, and don't show up when you use the `ls` command. They can be seen if you use the `ls -a` option for showing "all" files and directories, including the hidden ones. Go ahead and try it now, you will see both `./` and `../` regardless of which directory you are in. Even the "root" directory, the top-most directory in the tree, contains the `../` directory. Trying to go up from the root directory will just land you back in the root directory however.

A few other handy shortcuts are below. To move up two directories at once:

```
$ cd ../../                    (go up two directories)
```

To move to straight to your home directory:

```
$ cd                             (go directly to your home directory)
$ pwd
$ /Users/bob
```

To move to a directory within your home directory:

```
$ cd ~/project            (go to a subdirectory of your home directory)
$ pwd
$ /Users/bob/project
```

To move to the last directory you were in, prior to the last invocation of `cd`:

```
$ pwd
$ /Users/bob
$ cd /Users/mary/projects/one
$ pwd
$ /Users/mary/projects/one
$ cd -                          (go to the directory prior to the last cd)
$ pwd
$ /Users/bob
```

3.2   Creating/Removing Directories and Files on the Command Line

Now that we know how to see what directory we're in with `pwd`, and see the contents of the directory with `ls` and change between directories with `cd`, we discuss how to make and remove contents of directory. A directory has two types of elements: other directories, and files. We begin first with directories.

3.2.1   Making a new directory with the `mkdir` command

Making a new directory is done with the `mkdir` command. It typically has only a single argument, the name of the new directory. For example, suppose your current directory has the following contents:

```
$ ls -F
mail/  public_html/   report_a.pdf   report_b.pdf    test.txt
```

You can make a new directory, project_fango, using `mkdir`:

```
$ mkdir project_fango
$ ls -F
mail/  project_fango/   public_html/   report_a.pdf   report_b.pdf    test.txt
```

That's it. In our example, we were in the directory where we created the new directory, but that's not necessary. You can provide a full path name of the directory you wish to create. For example:

```
$ mkdir ~/project_fango
```

will create the new directory in your home directory regardless of where you are in your file system. If the directory already exists, it will tell you so:

```
$ mkdir ~/project_fango
$ mkdir ~/project_fango
mkdir: project_fango: File exits
```

Lastly, to make a directory, you need write permissions in the directory where you're creating the new directory. By default you always have write permission in your home directory and in the new directories you make. But eventually you may encounter the issue of permissions.

3.2.2 Removing a directory with the `rmdir` command

A directory may be removed with the `rmdir` command:

```
$ rmdir project_fango
```

To use this command the directory must be empty. If you really wish to remove a directory containing a whole tree of other files and directories without having to crawl through the tree, you can do this with the `rm -rf` command discussed below.

3.2.3 Making a file with the `touch` command

There are many ways to make a new file; with an editor, as output from a program and so on. Assuming no knowledge yet of these things, the easiest way to create an empty file is with the `touch` command

```
$ touch file1 file2 file3
$ ls
file1 file2 file3
```

The above action creates three empty files. To check that they are empty it may be a good time to introduce the `-l` (for *long* format) option of the `ls` command:

```
$ touch file1 file2 file3
$ ls -l
-rw-r--r--   1 john  staff      0B Nov  3 09:51 file1.txt
-rw-r--r--   1 john  staff      0B Nov  3 09:51 file2.txt
-rw-r--r--   1 john  staff      0B Nov  3 09:51 file3.txt
```

The fifth column confirms that each new file created with `touch` has a size of zero bytes. The important thing is now we know a simple way to make a file to demonstrate the next essential command, `rm` for removing a file.

3.2.4 Removing a file with the `rm` command

Existing files may be removed with the `rm` command, which accepts one or more file names as arguments:

```
$ ls
file1 file2 file3
$ rm file3
$ ls
file1 file2
```

A note of caution. The `rm` command can be brutally powerful. It does not, by default, ask you if you're sure about what you've just requested to be removed. I highly recommend you change this default behavior. The `rm` command has a command line option `-i` that changes `rm` to ask for confirmation:

```
$ ls
file1 file2 file3
$ rm -i file3
remove file3?
```

To see how to make this interactive mode the default mode for the `rm` command, see the following help topic:

http://oceanai.mit.edu/ivpman/help/cmdline_interactive_rm_mv_cp

For now, just be really careful using the `rm` command.

3.2.5 Removing groups of files with the `rm` command

The command line offers a generic ability for matching groups of files or directories by naming a pattern. To remove all three of the files in the example above, one could instead just type:

```
$ ls
file1 file2 file3
$ rm file*
$ ls
$
```

The asterisk pattern matches any combination of characters against all file and directory names in the current directory. You could also remove, for example, all PDF files with `rm *.pdf`. Or any PDF file in any subdirectory with `rm */*.pdf`. Obviously this should be used with caution.

3.2.6   Removing directories with the `rm` command

It was discussed above that a directory may be removed with the `rmdir` command. This only works when the directory being removed is empty. To remove an entire directory, including all its subdirectories, you can use two additional command line arguments:

```
$ rm -rf project
```

The `-r` options stands for *recursive* and will recursively remove the entire tree in the named directory. The `-f` option stands for *force* and overrides any `-i` option, meaning no prompt for confirmation will be presented after the command is invoked. Extremely powerful, extremely dangerous. Use with caution.

As with `mkdir`, the `rm` command will not let you remove files or directories for which you do have write permisions. But you should certainly pause and think for a second whenever you begin a shell command with `rm -rf`.

3.3   Moving/Renaming and Copying Files on the Command Line

Moving (renaming) and copying files and directories are done on the command line using the `mv` (move) command and the `cp` (copy) command.

3.3.1   Moving and Renaming a file with the `mv` command

The jobs of moving and renaming files are both handled by the `mv` command. In the simplest case for example, `file1` can be renamed to file `file2` like this:

```
$ ls
file1
$ mv file1 file2
$ ls
file2
```

The `mv` command may also be used to move one or many files into a directory:

```
$ ls -F
file1   file2   file3   file4   file5   project/
$ mv file* project
$ ls -F
project/
$ ls project
file1   file2   file3   file4   file5
```

If you try to move a file into a directory that doesn't exist you will get an error.

```
$ ls -F
file1   project/
$ mv file1 project23/
mv: rename file1 to project23/: No such file or directory
```

By default the `mv` command will let you rename one file to another, even if the target file already exists. Most people find this a bit unsettling to allow this without prompting for confirmation of intention. If the `mv` command is used with the `-i` option, the user will indeed be prompted for confirmation if the target file already exists. For this reason most people employ an alias for the `mv` command that makes the interactive mode the default:

```
$ alias mv 'mv -i'      (if using tcsh)
$ alias mv='mv -i'      (if using bash)
```

See the help topics on aliases and shell configuration to do this. Once this is done, you can always use the `-f` (force) command line option to force the renaming of file to a target, even if the target already exists, without prompting the user for confirmation.

3.3.2 Copying files and directories with the `cp` command

Files and whole directories can be copied with the `cp` command. The simplest case of copying one file to another:

```
$ ls
file1
$ cp file1 file2
$ ls
file1 file2
```

Copying a directory with `cp` requires a couple additional arguments on the command line:

```
$ ls -F
project/
$ cp -rp project project_backup
$ ls
project/ project_backup/
```

The first argument `-r` stands for *recursive* and indicates that all subcomponents of the folder being copied should also be copied, following the tree structure recursively in each subfolder's subfolder and so on. The second argument `-p` is not required. It ensures that certain file attributes are preserved when copying. This includes the read-write permissions, and date of last modification. When copying a directory it is a good habit to always use this option.

A note of caution. If you are copying a file to a file that already exists, the `cp` does not, by default, prompt you for a confirmation of your intentions. I highly recommend you change this default behavior. See the following additional help topic:

3.4   How to Customize Your Shell Environment

The command line experience is driven by a special program called the shell. The shell utilizes a number of common commands such as `ls`, `rm`, `mkdir` and so on. It turns out, there are a number of different shells to choose from. While they are all very similar, they have important differences, slightly different syntax, and users tend to have a strong allegiance to their choice of a shell. Perhaps the two most common choices are the *bash* and *tcsh* shells. The bash shell is what is typically running out-of-the-box upon a GNU/Linux or OSX install.

An important part of using the command line involves the customization of your shell to make it support your individual work style. In this help topic we:

- Explain how you can tell which shell you are presently using
- Introduce shell configuration files
- Provide example bash and tcsh shell configuration files
- Introduce custom prompts

The bash and tcsh executables are typically found in `/bin/bash` and `/bin/tcsh`. At any time you can just launch a new shell from the command line:

```
$ /bin/bash
```

This has the effect of perhaps switching shell types if you were using another shell, and it clears the local shell history (another topic).

3.4.1   What shell am I using?

Before augmenting your shell path, you need know which shell you're running. In most cases, it is either bash or tcsh. One way to find out:

```
$ echo $0
bash                    (should return with either bash or tcsh)
```

Take note of the answer and proceed.

3.4.2   Configuration files

Regardless of which shell you use, custom configuration is done with one or two key configuration files stored in the user's home directory. When the shell is launched, the file is read, and customizations are in place. Typically each line in the configuration file uses the same format as if the user typed them all directly at the start of their session.

If you have N terminal or console windows open, then you have N separate shell sessions in operation. They all read the same configuration file(s) upon startup. If you do make a change to the

shell environment from the command line, e.g., define an alias, change an environment vaiable, it will only be in effect in the shell session where you typed it in. If you want that effect to be carried over across all future shell sessions, this needs to be reflected in the shell configuration file(s).

3.4.3 Example shell configuration files for bash users

Bash typically is configured with the `.bashrc` and `.bash_profile` files in the user's home directory. For a discussion on why both files are used, see here:

http://www.joshstaiger.org/archives/2005/07/bash_profile_vs.html

Example `.bash_profile` and `.bashrc` files can be found by pointing your web browser to the below URLs and copying and pasting what you want into your own files.

http://oceanai.mit.edu/2.680/docs/.bash_profile

http://oceanai.mit.edu/2.680/docs/.bashrc

Or just use `wget` to download copies directly into your home directory:

```
$ cd                                            // go to your home directory
$ wget http://oceanai.mit.edu/2.680/docs/.bash_profile // get the file
$ wget http://oceanai.mit.edu/2.680/docs/.bashrc       // get the file
```

3.4.4 Example shell configuration file for tcsh users

The tcsh shell typically is configured with the single `.cshrc` file in the user's home directory. An example `.cshrc` file can be found by pointing your web browser to the below URL and copying and pasting what you want into your own files.

http://oceanai.mit.edu/2.680/docs/.cshrc

Or just use `wget` to download a copy directly into your home directory:

```
$ cd                                         // go to your home directory
$ wget http://oceanai.mit.edu/2.680/docs/.cshrc   // get the file
```

3.4.5 Customize Your Shell Prompt

One customization I can't live without is the customization of my shell prompt. By convention when we indicate a command typed on the command line, we would show something like:

```
$ cd ../
```

The dollar sign character is the *shell prompt*. In practice, people commonly customize this with something like:

```
leonardo:Research/project$ cd ../
```

Everything to the left of the dollar sign is a customization. In this case `leonardo` is the name of the machine, and `Research/project` is the name of current directory. Actually the full name of the current directory, may be too long to want on your shell prompt, so only two levels are shown in the prompt. This is my personal favorite configuration, but you can choose one of your own. For me, I always want to know what machine I'm on because I'm frequently remotely logging in to other machines or robots. And I always want some indication of which directory I'm in and showing two levels is enough to do the trick.

Next we'll show how this kind of configuration can be done for both bash and tcsh. Regardless of how you configure yours, I definitely recommend taking this step. It's extremely useful in practice and will cut down common errors due to not knowing what directory or machine you're on when you type a command.

3.4.6   Customize Your Shell Prompt for bash Users

If you're using bash and you took the example `.bashrc` file provided above, then you already have customized your shell prompt. It was done with the line:

```
PS1="\H:\W\$ "
```

The
`H` will expand to the machine name, and the
`W` will expand to the last component of the present working directory. The
`$` simply puts the dollar sign at the end of prompt. A convention is to use the dollar sign for bash, and the percent sign for tcsh.

There are lots of other ways to configure your prompt to your liking, such as including the time, day of the week, and so on. A google search on "customize bash prompt" will give you all you need.

3.4.7   Customize Your Shell Prompt for tcsh Users

If you're using tcsh and you took the example `.cshrc` file provided above, then you already have customized your shell prompt. It was done with the line:

```
set  prompt="%m:%C2%% "
```

The `%m` will expand to the machine name, and the `%C2` foobar will expand to the last two components of the present working directory. The `%%` simply puts the percent sign at the end of prompt. A convention is to use the percent sign for tcsh and the dollar sign for bash.

There are lots of other ways to configure your prompt to your liking, such as including the time, day of the week, and so on. A google search on "customize tcsh prompt" will give you all you need.

3.5   Adding a Directory to Your Shell Path

The *shell path* is a list of locations your shell program knows about to look for programs to run. When you type:

```
$ pwd
```

or

```
$ du
```

The `pwd` command is located in the `/bin` directory and `du` is located in the `/usr/bin` directory. These locations are conventional locations most shells users have in their shell paths. But if you add your own code (as we will in this class) you'll need to know how to augment your shell path to tell your shell where the new programs reside.

Adding a directory to your shell path is a very common thing command line users need to do. Unfortunately the need often comes up for new users while they're still finding their feet with the command line, and before they know what a shell *environment* is, and sometimes before they even know how to use a text editor. Here we focus on the minimal steps for adding a new directory to the shell path, perhaps glossing over a few things for the sake of getting it done.

3.5.1   Make sure you know how to edit a text file

The next step involves a simple edit to a text file. If you know how to do this proceed to the next step.

If you don't know what a text file is, vs. say a Microsoft Word document which just has text, then you may not know what a *text editor* is either. Wikipedia has a good entry on this topic. I recommend giving it a skim.

The good news is that you very likely already have a text editor on your machine. On the Mac it is an application called `TextEdit`, in the `/Applications/` directory. In GNU/Linux, it is a called `gedit` in the `/usr/bin` directory. And, if for some reason those directories are not in your shell path, don't despair. You can launch these editors by typing their full path name, or double clicking on them in a GUI like Finder on the Mac.

The bad news is that we really don't want to be using editors like `gedit` or `TextEdit` in our class. And we certainly don't encourage people to launch a text editor from a GUI. We want to do this all from the command line, and want to use a more powerful text editor like `Emacs` or `vi`. But for now, do what you need to do, just so long as you can make simple edits to a text file so we can get through the next few steps below.

If you really do want to hit the pause button here and go get Emacs and learn just enough to make and save simple edits, then check out the first couple emacs help topics on the main help page.

3.5.2   Augmenting the shell path for bash users

If you're a bash user, your configuration is done in a couple files in your home directory: `.bash_profile`, and `.bashrc`. Use `ls -a` to see them since they are hidden files. If you don't have these files, I recommend downloading the example files described earlier, or on the help page:

http://oceanai.mit.edu/ivpman/help/cmdline_shell_environment

Otherwise, if you already have the `.bashrc` file, edit the file and put the following two lines at the

end of the file.

```
PATH=$PATH:/my/full/directory/
export PATH
```

3.5.3 Augmenting the shell path for tcsh users bash user不用看

If you're a tcsh user, your configuration is done in a the .cshr file in your home directory. Use `ls` with the `-a` optiona to see it since it is a hidden file. If you don't have this file, I recommend downloading the example file described earlier, or on the help page:

http://oceanai.mit.edu/ivpman/help/cmdline_shell_environment

Otherwise, if you already have the `.cshrc` file, edit the file and put the following line at the end.

```
set path = ( $path /my/full/directory )
```

3.5.4 Verifying success

To verify success, pick one of the executables in the directory you added to the shell path and use the `which` tool:

```
$ which cmake
/opt/local/bin
```

If unsuccessful, for tcsh users, try typing `rehash` on the command line to update the shell's internal cache of executables.

3.6 Augmenting the Power of the Command Line with Aliases

Aliases may be the single most powerful, and simplest tool available for command line users. They also allow you to personalize your work flow. They allow you to save time on simple things that you may do very often. For example, to change directories to the directory above the current one, you normally type:

```
$ cd ..
```

No problem, but if do this a lot, this can be simplified by making an alias:

```
$ alias cdd    'cd ..'
```

Once this is defined, I can change directories a bit more efficiently with:

新增自己的指令

15

```
$ cdd
```

3.6.1   Add aliases to your shell configuration to make them persistent

Although aliases can be defined on the fly on the command line as above, the drawback is that they do not carry over to other shell sessions in another window or at a later date after logging back in. To make more persistent aliases, add them to your shell configuration file. The above example alias can be achieved with the following line in your `.cshrc` file:

把它放到.bashrc 這樣每次開都會有

```
alias cdd 'cd ..'
```

or if you're a bash user, the same thing is accomplished in a slight different format in your `.bashrc` file:

```
alias cdd='cd ..'
```

3.6.2   Suggestions for useful aliases

Below are some suggestions offered up here partly from firm belief of best practices, and partly based on what I found to work for me over time. In the category of best practices, I recommend altering your `rm`, `mv` and `cp` shell commands to work in the interactive mode:

```
alias rm  'rm -i'
alias cp  'cp -i'
alias mv  'mv -i'
```

In my opinion, any time you're about to potentially overwrite or remove a file, a prompt should be given to the user. You can always override this by using `-f` on the command line which will override the `-i`. You might have these already if you took the class example shell configuration files. Some others:

```
alias cdd    'cd ..'
alias cddd   'cc ../..'
alias cdddd  'cd ../../..'
alias cddddd 'cd ../../../..'
```

The above should be self-explanatory. Try it. The need to change up a directory or three comes up a lot!

```
alias hemacs='/Applications/Emacs.app/Contents/MacOS/Emac -bg DarkOliveGreen -fg white'
```

This emacs alias explicity invokes the graphical emacs (if you installed it from emacsformacosx.com)

rather than the one in `/usr/bin/emacs`. It also sets my background and foreground color the way I like them.

du -d 1 -h用來看每個資料夾的大小

```
alias duh1    'du -d 1 -h'            (in OS X)
alias duh1    'du --max-depth=1 -h'   (in GNU/Linux)
```

The `du` command is super useful. It estimates the file space usage for the current directory and each subdirectory. The command line options are slightly different in GNU/Linux and OS X as shown above. By setting the depth (OSX) or max-depth (GNU/Linux) to one, this limits the depth of reporting subdirectories to just the present directory and subdirectories. Otherwise it will recursively report everything, which I find is too much. The first time you fill up your hard drive with a runaway log file, you will find this command indespensible.

alias cdls='cd ..;ls' 分號可以連續指令

```
alias ev   'awk "BEGIN{ print \!* }" '
```

I won't try to explain why this works, but it gives you a limited but handy calculator on the command line. Try `"ev 10+(3.1415 / 2)"`. (this may not work in bash as given above).

```
alias cdmi    'cd moos-ivp'
alias cdmim   'cd moos-ivp/missions'
alias cdmis   'cd moos-ivp/src'
```

In the example above the moos-ivp aliases are not what's important. The general idea is to give yourself meaningful aliases for jumping through your own file structure. Probaly 70% or more of my aliases are of this type, and may account for most of why I think using the command line is way more efficient than anything else. My "cd aliases" all begin with "cd" and usually what follows is a single letter for each directory in the path, or some heuristic along these lines. If I look over a student's shoulder midway through 2.680 and I see them typing "cd ../../project12/missions" rather than something like "cdp12m", then part of me feels like I have failed to pass on something important. Embrace this capability.

```
alias runa    'cd moos-ivp/missions/s1_alpha; pAntler alpha.moos'
alias runb    'cd moos-ivp/missions/s1_bravo; pAntler bravo.moos'
```

As before, the important thing here is not the actual mission. The key point is that an alias can include *multiple* shell commands separated by a semicolon. When you're working on a particular autonomy problem, you may find yourself launching the same mission over and over again to test little changes. With the above style alias, you can launch the mission from wherever your shell happens to be in the file system. You can even add a timewarp component to launch the mission with time warp 12, for example, with an alias like `runa12`.

3.6.3   Temporarily disabling an alias

Sometimes you may want to temporarily disable an alias like:

```
alias rm  'rm -i'
```

and return the `rm` command to the way it was before, without prompting for confirmation. You can do this by typing:

```
$ alias rm rm
. . .
```

3.6.4   Reminding yourself of previously defined aliases

If the power of aliases strikes you as it did me, you may find at some point that you have a lot of aliases defined. A quick check on my own computer just now confirms that I have over 850 aliases defined. People ask me how I can remember them all? Well, I don't. But I remember the important ones, and even the lesser used ones are still pretty useful when I come back to working on a piece of code or a paper I haven't touched in a while.

One way to remind yourself of previously defined aliases is to just look in the file where they are defined, e.g., `.cshrc` or `.bashrc`. But you know there has to be a better way. A slightly better way is to just type the following:

```
$ alias
cp      cp -i
mv     mv -i
rm        rm -i
. . .
```

This will dump all your defined aliases in alphabetical order. At least you didn't have to open an editor to get this information, but the alphabetical order may actually make it harder to find what you're looking for. A better approach is to use `grep` to find what you're looking for. For example, you may remember that you had an alias related to launching emacs, so you could do this on the command line:

```
$ alias | grep emacs
hemacs   emacs --geometry=162x83+735+0 -bg DarkOliveGreen -fg white'
nem      emacs -nw
```

This will find only only those aliases that have the word "emacs" as part of their definition. This is such a useful trick that I actually have an alias for that in my shell configuration file:

```
alias afind 'alias | grep'
```

This can now be used instead to find all aliases with the word "emacs" in the definition. The search list can be further pruned to say only show aliases that have "emacs" and "geometry" in the definition:

```
$ afind emacs | grep geometry
hemacs    emacs --geometry=162x83+735+0 -bg DarkOliveGreen -fg white'
```

來跑demo程式搜～

4 Getting, Building and Running the Workshop Software

The objective is to build and run the "Hello World" version of the marine autonomy software, the Alpha mission. This will confirm for us that:

- You have everything you need in terms of software installed on you machine.
- You have properly configured your shell path to include the locations of the software executables used in this class.

Follow the steps here until you are able to successfully run the Alpha example mission.

4.1 Downloading and Building the MOOS-IvP Software

The MOOS-IvP autonomy code may be checked out and built following the steps on this page. Before you begin, below are few steps that need to be done beforehand:

- On the Mac you will need to have installed XCode and command line tools. More info here:
  http://oceanai.mit.edu/ivpman/help/osx_get_xcode
- On the Mac you will need to have installed MacPorts. More info here:
  http://oceanai.mit.edu/ivpman/help/osx_get_macports
- In both GNU/Linux and on the Mac, you will need to have Subversion installed. More info here:
  http://oceanai.mit.edu/ivpman/help/svn_install

4.1.1 Downloading the Software

The latest software is downloaded using Subversion (svn):

貌似有人遇到問題，是要把家目錄底下的./cmake 刪了才有辦法跑好像是什麼sudo 的問題

```
$ svn co https://oceanai.mit.edu/svn/moos-ivp-aro/trunk/ moos-ivp
```

- The above command invokes svn to check out a tree, at the given URL, with the local name of "moos-ivp".
- This tree is generally available from moos-ivp.org in one of several forms. The most public forms are (1) the latest release, 15.5.1, and (2) the development trunk.
- At any point after an initial checkout, users may pull down the latest updates to the software by typing:

```
$ cd moos-ivp
$ svn update
```

update到最新版，現在是7818

19

4.1.2 Building the Course Software

After a successful download, you should have a new directory (folder) called moos-ivp. All the code is in this folder. There are README instructions in the top-level folder for each OS flavor. For GNU/Linux, the README-GNULINUX.txt file will tell you that there are a few packages to install before building:

ubuntu好方便Q Q

```
$ sudo apt-get install  g++  cmake  xterm
$ sudo apt-get install  libfltk1.3-dev  freeglut3-dev  libpng12-dev  libjpeg-dev
$ sudo apt-get install  libxft-dev  libxinerama-dev   libtiff4-dev
```

For OS X, the README-OS-X.txt file indicates the following package dependencies:

前面第一頁載過摟～

```
$ sudo port install fltk cmake tiff
```

You should also install the XQuartz software, which enables the use of X Terminal windows:

http://www.xquartz.org

Once this is completed, the course software may be built in the steps below:

```
$ cd moos-ivp
$ ./build-moos.sh
(let it build)
$ ./build-ivp.sh
(let it build)
```

4.2 Running the MOOS-IvP Software

After a successful build, there should be about 70+ new executables in the below directory. Start by confirming this.

```
$ cd  moos-ivp/bin/
$ ls
(about 70+ executables, including MOOSDB and pHelmIvP)
```

If you don't see these files in the bin directory, go back to the step of downloading and building the course software.

4.2.1 Add the new executables to your shell path

This moos-ivp/bin/ directory needs to be added to your shell path. If you know how to augment your shell path do this now and proceed to the next step. Otherwise, see the help topic:
http://oceanai.mit.edu/ivpman/help/cmdline_augment_shell_path

4.2.2 Verify that moos-ivp executables are in your shell path

After the code is built, and the shell path has been augmented, we should be able to run an example. First, a simple sanity test:

```
$ which MOOSDB
/Users/you/moos-ivp/bin/MOOSDB          確定一下裝的位置沒錯
$ which pHelmIvP
/Users/you/moos-ivp/bin/pHelmIvP
```

If either of the above `which` queries turn up blank, go back and verify that the software built (check the bin directories), and verify that your shell path has been augmented. Note: For tcsh users, you may have to type `rehash` in your shell to "remind" your shell to update its knowledge of things in its path.

4.2.3 Run the example "alpha" mission

Now launch the example "alpha" mission:

```
$ cd moos-ivp/ivp/missions/s1_alpha          執行範例程式～
$ pAntler --MOOSTimeWarp=10 alpha.moos
```

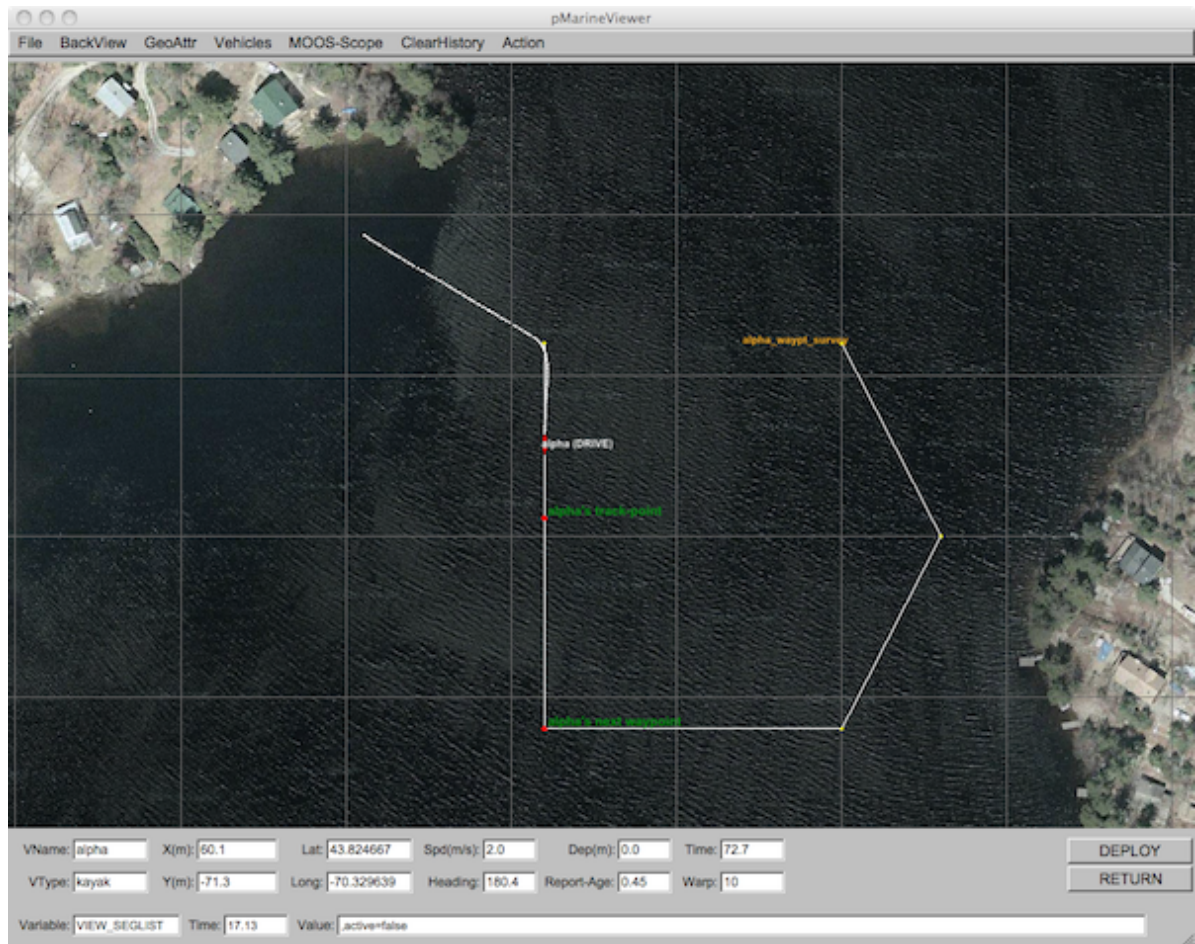When the `pMarineViewer` window comes up, hit the Deploy button. You should see something similar to this.

Figure 1: A screen shot of the Alpha example mission shortly after launch.

4.2.4   Operating Systems Supported by MOOS and IvP

The MOOS software distributed by Oxford is well supported on Linux, Windows and Mac OS X. The software distributed by MIT includes additional MOOS utility applications and the IvP Helm and related behaviors. These modules are supported on Linux and Mac OS X and the software compiles and runs on Windows but Windows support is limited.