UW AMath 483/583 Class Notes

Release 1.0

Randall J. LeVeque

March 31, 2011

CONTENTS

See the Class Webpage for information on instructor, TA, office hours, etc.

Skip to... *Technical Topics ... Applications*

Condensed Table of Contents without subsections

CHAPTER

ONE

COURSE MATERIALS

1.1 About these notes – important disclaimers

These note on high performance scientific computing are being developed for the course Applied Mathematics 483/583 at the University of Washington, Spring Quarter, 2011.

They are very much a work in progress. Many pages are not yet here and the ones that are will mostly be modified and supplemented as the quarter progresses.

It is not intended to be a complete textbook on the subject, by any means. The goal is to get the student started with a few key concepts and techniques and then encourage further reading elsewhere. So it is a collection of brief introductions to various important topics with pointers to books, websites, and other references for more details. See in particular the sections *resources* and *Bibliography and further reading* for useful pointers.

There are many pointers to Wikipedia pages sprinkled through the notes and in the bibliography, simply because these pages often give a good overview of issues without getting into too much detail. They are not necessarily definitive sources of accurate information.

These notes are mostly written in Sphinx (see *sphinx*) and the input files are available using Mercurial (see *Instructions for cloning the class repository*).

1.2 Overview and syllabus

This course will cover a large number of topics in a somewhat superficial manner. The main goals are to:

- Introduce a number of concepts related to machine architecture, programming languages, etc. that necessary to understand if one plans to write computer programs beyond simple exercises, where it is important that they run efficiently.
- Introduce a variety of software tools that are useful to programmers working in scientific computing.
- Gain a bit of hands on experience with these tools so that at the end of the quarter students will be able to continue working with them and be in a good position to learn more on their own from available resources.

1.2.1 Outline and schedule

Warning: This is a tentative outline that is subject to change.

Week 1:

- Overview and introduction of some main ideas.
- Using the Virtual Machine.
- Unix / Linux
- Version control systems
- Using Mercurial and Bitbucket.

Week 2:

- Reproducible research
- Compiled vs. interpreted languages
- Fortran 90
- Makefiles

Week 3:

- Binary, floating point numbers, memory
- BLAS, LAPACK
- Iterative methods for linear systems

Week 4:

- Computer architecture: CPU, memory access, cache hierachy, pipelining
- Parallel computing
- OpenMP

Week 5:

- OpenMP

Week 6:

- MPI

Week 7:

- MPI

Week 8:

- Other parallel languages
- GPUs

Week 9:

- Graphics and visualization

Week 10:

1.3 Slides from lectures

Slides are not included in the repository in order to keep the size down. Below are links to pdf files of the slides for each lecture for which they are available.

In each case 3 versions are provided:

- nup1 means one slide per page, as projected in lecture,
- nup3 has 3 slides per page with room for notes, portrait mode
- nup4 has 4 slides per page, landscape mode

Week 1

- Lecture 1: March 28, 2011 ... nup1 ... nup3 ... nup4
  - Overview of class, large-scale computational science, Virtual Machine
- Lecture 2: March 30, 2011 ... nup1 ... nup3 ... nup4
  - Version control

1.4 Homework Assignments

Contents:

1.4.1 Homework 1

Due Wednesday, April 6, 2011, by 11:00pm PDT.

The goals of this homework are to:

- Make sure you are familiar with basic Unix commands (see *Unix, Linux, and OS X*) and an editor, (see *Text editors*),
- Start using Mercurial (*Mercurial (hg)*) to download course materials,
- Create your own hg repository to keep your coursework and post homeworks to be graded.

Before tackling this homework, you should read some of the class notes and links they point to. In particular, the following sections are relevant:

- *Unix, Linux, and OS X*
- *Downloading and installing software for this class*
- *Text editors*
- *Mercurial (hg)*
- *Bitbucket repositories: viewing changesets, issue tracking*
- *Bibliography and further reading*

You do not need to "turn in" anything for this homework, but by doing it you will create files in your repository that we can view to grade.

1. First, if you have not already completed the survey on your background and computing needs, please do so now. You can find it here.
2. Make sure you have access to Mercurial on the computer you plan to use (see *Downloading and installing software for this class*). Read the section *Mercurial (hg)* and the documentation linked from there in order to get a sense of how it works.
3. Clone the class repository following the *Instructions for cloning the class repository*.

   Make sure you have set the environment variable *CLASSHG* since this is used below.

4. Set up your own personal repository on Bitbucket, by carefully following all of the instructions at *Creating your own Bitbucket repository*. By following these instructions you will also create a clone of the repository and add a file to it called *testfile.txt*.

   Make sure you have set the environment variable *MYHG* since this is used below.

5. In the clone of your repository, create a subdirectory *homeworks* and within this directory a subdirectory *homework1*, via the following commands:

   ```
   $ cd $MYHG
   $ mkdir homeworks
   $ cd homeworks
   $ mkdir homework1
   ```

   You should now be able to *cd* into this directory:

   ```
   $ cd homework1
   ```

   or later you can get there from anywhere via:

   ```
   $ cd $MYHG/homeworks/homework1
   ```

6. Copy a file from the class repository to your own repository by:

   ```
   $ cp $CLASSHG/homeworks/homework1/testpython.py $MYHG/homeworks/homework1
   ```

   This should create a file *testpython.py* in the directory *$MYHG/homeworks/homework1*. Read the instructions in the file and do as instructed. As part of this you will *add* two files using *hg*, *commit* this change, and *push* them to your Bitbucket repository so that we can view them.

   Note: If you have done this and also followed the instructions at *Creating your own Bitbucket repository* and in the file testpython.py, you should have 3 files in your repository:

   - $MYHG/testfile.txt
   - $MYHG/homeworks/homework1/testpython.py
   - $MYHG/homeworks/homework1/testoutput.txt

   These files should also be visible from your bitbucket webpage, by clicking on the "Source" tab, see *Bitbucket repositories: viewing changesets, issue tracking*.

   Note: After running the test script, you will also have a file *test.f90* and perhaps *a.out* in your directory $MYHG. Do **not** add these to your repository with *hg add*.

7. You created a private repository, so you will have to give us permission to view or clone it. Do so by clicking on the "Admin" tab at the top of your Bitbucket account page and then use the "Add reader" tool. You should add both *rjleveque* and *grady.lemoine* (the TA).

8. Finally, let us know where your Bitbucket repository is so that we can clone it and/or view your source files online in order to grade it. If you are registered in the class, you should be able access the class Catalyst webpage. Go to that page and follow instructions to send us this information.

1.5 Downloading and installing software for this class

Rather than downloading and installing this software, you might want to consider using the *Virtual Machine for this class*, which already contains everything you need.

Registered students can also get an account on a Linux machine in the Applied Mathematics Department at UW to do your work if desired, a machine with all this software already installed. Registered class members can visit the class Catalyst webpage for information on how to obtain an account. Then see *Using ssh to connect to remote computers* for information on how to use these machines.

It is assumed that you are on a Unix-like machine (e.g Linux or Mac OS X). For some flavors of Unix it is easy to download and install some of the required packages using apt-get, or your system's package manager; many Python packages can also be installed using easy_install. Notes about this are provided below.

If you insist on using a Windows machine (not recommended for high-performance scientific computing in general), then you will need to either download and install [VirtualBox] for Windows and then run the *Virtual Machine for this class* to provide a Linux environment, or else use *cygwin*.

Some of this software may already be available on your machine. The *which* command in Unix will tell you if a command is found on your *search path*, e.g.:

```
$ which python
/usr/bin/python
```

tells me that when I type the python command it runs the program located in the file listed. Often executables are stored in directories named *bin*, which is short for *binary*, since they are often binary machine code files.

If *which* doesn't print anything, or says something like:

```
$ which xyz
/usr/bin/which: no xyz in (/usr/bin:/usr/local/bin)
```

then the command cannot be found on the *search path*. So either the software is not installed or it has been installed in a directory that isn't searched by the shell program (see *Shells*) when it tries to interpret your command. See *PATH and other search paths* for more information.

1.5.1 Versions

Often there is more than one version of software packages in use. Newer versions may have more features than older versions and perhaps even behave differently with respect to common features. For some of what we do it will be important to have a version that is sufficiently current.

For example, Python has changed dramatically in recent years. Everything we need (I think!) for this class can be found in Version 2.X.Y for any $X \geq 4$.

Major changes were made to Python in going to Python 3.0, which has not been broadly adopted by the community yet (because much code would have to be rewritten). In this class we are *not* using Python 3.X. (See [Python-3.0-tutorial] for more information.)

To determine what version of software you have installed, often the command can be issued with the `--version` flag, e.g.:

```
$ python --version
Python 2.5.4
```

1.5.2 Individual packages

Python

If the version of Python on your computer is older than 2.4.0 (see above), you should upgrade.

See http://www.python.org/download/ or consider the EPD described below.

Enthought Python Distribution (EPD)

You might consider installing the EPD (free for academic users, see http://code.enthought.com). This includes a recent version of Python 2.X as well as many of the other Python packages listed below (IPython, NumPy, SciPy, matplotlib, mayavi).

EPD works well on Windows machines too.

IPython

The IPython shell is much nicer to use than the standard Python shell (see *Shells* and *ipython*). (Included in EPD.)

See http://ipython.scipy.org/moin/

NumPy and SciPy

Used for numerical computing in Python (see *numerical_python*). (Included in EPD.)

See http://www.scipy.org/Installing_SciPy

Matplotlib

Matlab-like plotting package for 1d and 2d plots in Python. (Included in EPD.)

See http://matplotlib.sourceforge.net/

Mercurial

Version control system (see *Mercurial (hg)*).

See http://mercurial.selenic.com/

Sphinx

Documentation system used to create these class notes pages (see *sphinx*).

See http://sphinx.pocoo.org/

gfortran

GNU fortran compiler (see *fortran*).

You may already have this installed, try:

```
$ which gfortran
```

See http://gcc.gnu.org/wiki/GFortran

OpenMP

Included with gfortran (see *openmp*).

Open MPI

Message Passing Interface software for parallel computing (see *mpi*).

See http://www.open-mpi.org/

LAPack

Linear Algebra Package, a standard library of highly optimized linear algebra subroutines. LAPack depends on the BLAS (Basic Linear Algebra Subroutines); it is distributed with a reference BLAS implementation, but more highly optimized BLAS are available for most systems.

See http://www.netlib.org/lapack/

1.5.3 Software available through *apt-get*

On a recent Debian or Ubuntu Linux system, most of the software for this class can be installed through *apt-get*. To install, type the command:

```
$ sudo apt-get install PACKAGE
```

where the appropriate PACKAGE to install comes from the list below.

NOTE: You will only be able to do this on your own machine, the VM described at *Virtual Machine for this class*, or a computer on which you have super user privileges to install software in the sytsem files. (See *sudo*)

You can also install these packages using a graphical package manager such as Synaptic instead of *apt-get*. If you are able to install all of these packages, you do not need to install the Enthought Python Distribution.

Software	Package
Python	python
IPython	ipython
NumPy	python-numpy
SciPy	python-scipy
Matplotlib	python-matplotlib
Python development files	python-dev
Mercurial	mercurial
Sphinx	python-sphinx
gfortran	gfortran
OpenMPI libraries	libopenmpi-dev
OpenMPI executables	openmpi-bin
LAPack	liblapack-dev

Many of these packages depend on other packages; answer "yes" when *apt-get* asks you if you want to download them. Some of them, such as Python, are probably already installed on your system, in which case *apt-get* will tell you that they are already installed and do nothing.

1.5.4 Software available through *easy_install*

easy_install is a Python utility that can automatically download and install many Python packages. It is part of the Python *setuptools* package, available from http://pypi.python.org/pypi/setuptools, and requires Python to already be installed on your system. Once this package is installed, you can install Python packages on a Unix system by typing:

```
$ sudo easy_install PACKAGE
```

where the PACKAGE to install comes from the list below. Note that these packages are redundant with the ones available from *apt-get*; use *apt-get* if it's available.

Software	Package
IPython	IPython[kernel,security]
NumPy	numpy
SciPy	scipy
Matplotlib	matplotlib
Mayavi	mayavi
Mercurial	mercurial
Sphinx	sphinx

If these packages fail to build, you may need to install the Python headers.

1.6 Virtual Machine for this class

We are using a wide variety of software in this class, much of which is probably not found on your computer. It is all open source software (see *licences*) and links/instructions can be found in the section *Downloading and installing software for this class*.

An alternative, which many will find more convenient, is to download and install the [VirtualBox] software and then download a Virtual Machine (VM) that has been built specifically for this course. VirtualBox will run this machine, which will emulate a specific version of Linux that already has installed all of the software packages that will be used in this course.

You can find the VM on the class webpage. Note that the file is quite large (approximately 750 MB compressed), and if possible you should download it from on-campus to shorten the download time.

1.6.1 System requirements

The VM is around 1.9 GB in size, uncompressed, and the virtual disk image may expand to up to 8 GB, depending on how much data you store in the VM. Make sure you have enough free space available before installing. You can set how much RAM is available to the VM when configuring it, but it is recommended that you give it at least 512 MB; since your computer must host your own operating system at the same time, it is recommended that you have at least 1 GB of total RAM.

1.6.2 Setting up the VM in VirtualBox

Once you have downloaded and uncompressed the virtual machine disk image from the class web site, you can set it up in VirtualBox, by doing the following:

1. Start VirtualBox
2. Click the *New* button near the upper-left corner
3. Click *Next* at the starting page

4. Enter a name for the VM (put in whatever you like); for *OS Type*, select "Linux", and for *Version*, select "Ubuntu". Click *Next*.
5. Enter the amount of memory to give the VM, in megabytes. Give it as much as you can spare; 512 MB is the recommended minimum. Click *Next*.
6. Click *Use existing hard disk*, then click the folder icon next to the disk list. In the Virtual Media Manager that appears, click *Add*, then select the virtual machine disk image you downloaded from the class web site. Ignore the message about the recommended size of the boot disk, and leave the box labeled "Boot Hard Disk (Primary Master)" checked. Once you have selected the disk image, click *Next*.
7. Review the summary VirtualBox gives you, then click *Finish*. Your new virtual machine should appear on the left side of the VirtualBox window.

Optionally, if you have a reasonably new computer with a multi-core processor and want to be able to run parallel programs across multiple cores, you can tell VirtualBox to allow the VM to use additional cores. To do this, click the VM on the left side of the VirtualBox window, then click *Settings*. Under *System*, click the *Processor* tab, then use the slider to set the number of processors the VM will see. Note that some older multi-core processors do not support the necessary extensions for this, and on these machines you will only be able to run the VM on a single core.

1.6.3 Starting the VM

Once you have configured the VM in VirtualBox, you can start it by double-clicking it in the list of VM's on your system. The virtual machine will take a little time to start up; as it does, VirtualBox will display a few messages explaining about mouse pointer and keyboard capturing, which you should read.

After the VM has finished booting, it will present you with a login screen; the login and password are both `amath583`. (We would have liked to set up a VM with no password, but many things in Linux assume you have one.)

1.6.4 Running programs

You can access the programs on the virtual machine through the main menu (the mouse on an *X* symbol in the lower-left corner of the screen), or by clicking the quick-launch icons next to the menu button. By default, you will have quick-launch icons for a command prompt window (also known as a *terminal window*), a text editor, and a web browser. After logging in for the first time, you should start the web browser to make sure your network connection is working.

1.6.5 Fixing networking issues

When a Linux VM is moved to a new computer, it sometimes doesn't realize that the previous computer's network adaptor is no longer available. If you find yourself unable to connect to the Internet, open a terminal window and type the following command:

```
$ sudo rm /etc/udev/rules.d/70-persistent-net.rules
```

This will remove the incorrect settings; Linux should then autodetect and correctly configure the network interface it boots. To reboot the VM, click the door icon in the bottom-right of the screen, then click *Restart*. If this does not work, contact the TA.

1.6.6 Changing guest resolution/VM window size

See Also:

The section *Guest Additions*, which makes this easier.

It's possible that the size of the VM's window may be too large for your display; resizing it in the normal way will result in not all of the VM desktop being displayed, which may not be the ideal way to work. Alternately, if you are working on a high-resolution display, you may want to *increase* the size of the VM's desktop to take advantage of it. In either case, you can change the VM's display size by going to the main menu in the lower-left corner, pointing to *Settings*, then clicking *Display*. Choose a resolution from the drop-down list, then click *Apply*.

1.6.7 Setting the host key

See Also:

The section *Guest Additions*, which makes this easier.

When you click on the VM window, it will capture your mouse and future mouse actions will apply to the windows in the VM. To uncapture the mouse you need to hit some control key, called the *host key*. It should give you a message about this. If it says the host key is Right Control, for example, that means the Control key on the right side of your keyboard (it does *not* mean to click the right mouse button).

On some systems, the host key that transfers input focus between the VM and the host operating system may be a key that you want to use in the VM for other purposes. To fix this, you can change the host key in VirtualBox. In the main VirtualBox window (not the VM's window; in fact, the VM doesn't need to be running to do this), go to the *File* menu, then click *Settings*. Under *Input*, click the box marked "Host Key", then press the key you want to use.

1.6.8 Shutting down

When you are done using the virtual machine, you can shut it down by clicking the door icon in the bottom-right of the screen, then clicking *Shut down*.

1.6.9 About the VM

The class virtual machine is running XUbuntu 10.10, a variant of Ubuntu Linux (http://www.ubuntu.com), which itself is an offshoot of Debian GNU/Linux (http://www.debian.org). XUbuntu is a stripped-down, simplified version of Ubuntu suitable for running on smaller systems (or virtual machines); it runs the *xfce4* desktop environment.

1.6.10 Guest Additions

While we have installed the VirtualBox guest additions on the class VM, the guest additions sometimes stop working when the VM is moved to a different computer, so you may need to reinstall them. Do the following so that the VM will automatically capture and uncapture your mouse depending on whether you click in the VM window or outside it, and to make it easier to resize the VM window to fit your display.

1. Boot the VM, and log in.
2. In the VirtualBox menu bar on your host system, select Devices –> Install Guest Additions... (Note: click on the window for the class VM itself to get this menu, not on the main "Sun VirtualBox" window.)
3. A CD drive should appear on the VM's desktop, along with a popup window. (If it doesn't, see the additional instructions below.) Select "Allow Auto-Run" in the popup window. Then enter the password you use to log in.
4. The Guest Additions will begin to install, and a window will appear, displaying the progress of the installation. When the installation is done, the window will tell you to press 'Enter' to close it.
5. Right-click the CD drive on the desktop, and select 'Eject'.
6. Restart the VM.

If step 3 doesn't work the first time, you might need to:

Alternative Step 3:

1. Reboot the VM.
2. Mount the CD image by right-clicking the CD drive icon, and clicking 'Mount'.
3. Double click the CD image to open it.
4. Double click 'autorun.sh'.
5. Enter the VM password to install.

Thanks go to Grady Lemoine and Jonathan Claridge for setting up the VM.

1.6.11 Further reading

[VirtualBox] [VirtualBox-documentation]

1.7 Bibliography and further reading

See Also:

Links from 2009 seminar

Many other pages in these notes have links not listed below. These are some references that are partiuclarly useful or are cited often elsewhere.

1.7.1 Books

1.7.2 Other courses with useful slides or webpages

1.7.3 Other Links

Software

See Also:

Downloading and installing software for this class for links to software download pages.

Virtual machine:

Sphinx:

Python:

Numerical Python

Unix, bash:

Mercurial and other version control systems:

Fortran

Many tutorials and references are available online. Search for "fortran 90 tutorial" or "fortran 95 tutorial" to find many others.

Makefiles:

Computer architecture

Floating point arithmetic

Languages and compilers

OpenMP:

Exa-scale computing

More will be added, check back later

CHAPTER

TWO

TECHNICAL TOPICS

2.1 Shells

A shell is a program that allows you to interact with the computer's operating system or some software package by typing in commands. The shell interprets (parses) the commands and typically immediately performs some action as a result. Sometimes a shell is called a *command line interface* (CLI), as opposed to a *graphical user interface* (GUI), which generally is more point-and-click.

On a Windows system, most people use the point-and-click approach, though it is also possible to open a window in command-line mode for its DOS operating system. Note that DOS is different from Unix, and we will *not* be using DOS. Using *cygwin* is one way to get a unix-like environment on Windows, but if have a Windows PC, we recommend that you use one of the other options listed in *Downloading and installing software for this class*.

On a Unix or Linux computer, people normally use a shell in a "terminal window" to interact with the computer, although most flavors of Linux also have point-and-click interfaces depending on what "Window manager" is being used.

On a Mac there is also the option of using a Unix shell in a terminal window (go to Applications –> Untilities –> Terminal to open a terminal). The Mac OS X operating system (also known as Tiger, Leopard, Snow Leopard, etc. depending on version) is essentially a flavor of Unix.

2.1.1 Unix shells

When a terminal opens, it shows a *prompt* to indicate that it is waiting for input. In these notes a single $ will generally be used to indicate a Unix prompt, though your system might give something different. Often the name of the computer appears in the prompt. (See *Setting the prompt* for information on how you can change the Unix prompt to your liking.)

Type a command at the prompt and hit return, and in general you should get some response followed by a new prompt. For example:

```
$ pwd
/Users/rjl/
$
```

In Unix the *pwd* command means "print working directory", and the result is the full path to the directory you are currently working in. (Directories are called "folders" on windows systems.) The output above shows that on my computer at the top level there is a directory named */Users* that has a subdirectory for each distinct user of the computer. The directory */Users/rjl* is where Randy LeVeque's files are stored, and within this we are several levels down.

To see what files are in the current working directory, the *ls* (list) command can be used:

```
$ ls
```

For more about Unix commands, see the section *Unix, Linux, and OS X*.

There are actually several different shells that have been developed for Unix, which have somewhat different command names and capabilities. Basic commands like *pwd* and *ls* (and many others) are the same for any Unix shell, but they more complicated things may differ.

In this class, we will assume you are using the bash shell (see *The bash shell*). See *Unix, Linux, and OS X* for more Unix commands.

Matlab shell

If you have used Matlab before, you are familiar with the Matlab shell, which uses the prompt >>. If you use the GUI version of Matlab then this shell is running in the "Command window". You can also run Matlab from the command line in Unix, resulting in the Matlab prompt simply showing up in your terminal window. To start it this way, use the *-nojvm* option:

```
$ matlab -nojvm
>>
```

Python shell

We will use Python extensively in this class. For more information see the section *python*.

Most Unix (Linux, OSX) computers have Python available by default, invoked by:

```
$ python
Python 2.5.4 (r254:67916, Jul  7 2009, 23:51:24)
[GCC 4.2.1 (Apple Inc. build 5646)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

This prints out some information about the version of Python and then gives the standard Python prompt, >>>. At this point you are in the Python shell and any commands you type will be interpreted by this shell rather than the Unix shell. You can now type Python commands, e.g.:

```
>>> x = 3+4
>>> x
7
>>> x+2
9
>>> 4/3
1
```

The last line might be cause for concern, since 4/3 is not 1. For more about this, see *numerical_python*. The problem is that since 4 and 3 are both integers, Python gives an integer result. To get a better result, express 4 and 3 as real numbers (called *float*s in Python) by adding decimal points:

```
>>> 4./3.
1.3333333333333333
```

The standard Python shell is very basic; you can type in Python commands and it will interpret them, but it doesn't do much else.

2.1.2 IPython shell

A much better shell for Python is the *IPython shell*, which has extensive documentation at [IPython-documentation].

Note that IPython has a different sort of prompt:

```
$ ipython

Python 2.5.4 (r254:67916, Jul  7 2009, 23:51:24)
Type "copyright", "credits" or "license" for more information.

IPython 0.9.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ?object also works, ?? prints more.

In [1]: x = 4./3.

In [2]: x
Out[2]: 1.3333333333333333

In [3]:
```

The prompt has the form *In [n]* and any output is preceeded by by *Out [n]*. IPython stores all the inputs and outputs in an array of strings, allowing you to later reuse expressions.

For more about some handy features of this shell, see *ipython*.

The IPython shell also is programmed to recognize many commands that are not Python commands, making it easier to do many things. For example, IPython recognizes *pwd*, *ls* and various other Unix commands, e.g. to print out the working directory you are in while in IPython, just do:

```
In [3]: pwd
```

Note that IPython is not installed by default on most computers, you will have to download it and install yourself (see [IPython-documentation]). It is installed on the *Virtual Machine for this class*.

If you get hooked on the IPython shell, you can even use it as a Unix shell, see documentation.

2.1.3 Further reading

[IPython-documentation], [IPython-links]

2.2 Unix, Linux, and OS X

A brief introduction to Unix shells appears in the section *Shells*. Please read that first before continuing here.

There are many Unix commands and most of them have many optional arguments (generally specified by adding something like -x after the command name, where x is some letter). Only a few important commands are reviewed here. See the references (e.g. [Wikipedia-unix-utilities]) for links with many more details.

2.2.1 pwd and cd

The command name *pwd* stands for "print working directory" and tells you the full path to the directory you are currently working in, e.g.:

```
$ pwd
/Users/rjl/uwamath583
```

To change directories, use the *cd* command, either relative to the current directory, or as an absolute path (starting with a "/", like the output of the above pwd command). To go up one level:

```
$ cd ..
$ pwd
/Users/rjl
```

2.2.2 ls

ls is used to list the contents of the current working directory. As with many commands, *ls* can take both *arguments* and *options*. An option tells the shell more about what you want the command to do, and is preceded by a dash, e.g.:

```
$ ls -l
```

The *-l* option tells *ls* to give a long listing that contains additional information about each file, such as how large it is, who owns it, when it was last modified, etc. The first 10 mysterious characters tell who has permission to read, write, or execute the file, see [Wikipedia].

Commands often also take *arguments*, just like a function takes an argument and the function value depends on the argument supplied. In the case of *ls*, you can specify a list of files to apply *ls* to. For example, if we only want to list the information about a specific file:

```
$ ls -l fname
```

You can also use the *wildcard* * character to match more than one file:

```
$ ls *.x
```

If you type

    $ ls -F

then directories will show up with a trailing / and executable files with a trailing asterisk, useful in distinguishing these from ordinary files.

When you type *ls* with no arguments it generally shows most files and subdirectories, but may not show them all. By default it does not show files or directories that start with a period (dot). These are "hidden" files such as *.bashrc* described in Section *.bashrc file*.

To list these hidden files use:

```
$ ls -aF
```

Note that this will also list two directories *./* and *../* These appear in every directory and always refer to the current directory and the parent directory up one level. The latter is frequently used to move up one level in the directory structure via:

```
$ cd ..
```

For more about *ls*, try:

```
$ man ls
```

Note that this invokes the *man* command (manual pages) with the argument *ls*, and causes Unix to print out some user manual information about *ls*.

If you try this, you will probably get one page of information with a ':' at the bottom. At this point you are in a different shell, one designed to let you scroll or search through a long file within a terminal. The ':' is the prompt for this shell. The commands you can type at this point are different than those in the Unix shell. The most useful are:

```
: q  [to quit out of this shell and return to Unix]
: <SPACE>   [tap the Spacebar to display the next screenfull]
: b  [go back to the previous screenfull]
```

2.2.3 more, less, cat, head, tail

The same technique to paging through a long file can be applied to your own files using the *less* command (an improvement over the original *more* command of Unix), e.g.:

```
$ less filename
```

will display the first screenfull of the file and give the : prompt.

The *cat* command prints the entire file rather than a page at a time. *cat* stands for "catenate" and *cat* can also be used to combine multiple files into a single file:

```
$ cat file1 file2 file3 > bigfile
```

The contents of all three files, in the order given, will now be in *bigfile*. If you leave off the "> bigfile" then the results go to the screen instead of to a new file, so "cat file1" just prints file1 on the screen. If you leave off file names before ">" it takes input from the screen until you type <ctrl>-d, as used in the example at *Creating your own Bitbucket repository*.

Sometimes you want to just see the first 10 lines or the last 5 lines of a file, for example. Try:

```
$ head -10 filename
$ tail -5 filename
```

2.2.4 removing, moving, copying files

If you want to get rid of a file named *filename*, use *rm*:

```
$ rm -i filename
remove filename?
```

The -i flags forces *rm* to ask before deleting, a good precaution. Many systems are set up so this is the default, possibly by including the following line in the *.bashrc file*:

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

If you want to force removal without asking (useful if you're removing a bunch of files at once and you're sure about what you're doing), use the -f flag.

To rename a file or move to a different place (e.g. a different directory):

```
$ mv oldfile newfile
```

each can be a full or relative path to a location outside the current working directory.

To copy a file use *cp*:

```
$ cp oldfile newfile
```

The original *oldfile* still exists. To copy an entire directory structure recursively (copying all files in it and any subdirectories the same way), use "cp -r":

```
$ cp -r olddir newdir
```

2.2.5 background and foreground jobs

When you run a program that will take a long time to execute, you might want to run it in *background* so that you can continue to use the Unix command line to do other things while it runs. For example, suppose *fortrancode.exe* is a Fortran executable in your current directory that is going to run for a long time. You can do:

```
$ ./fortrancode.exe &
[1] 15442
```

if you now hit return you should get the Unix prompt back and can continue working.

The ./ before the command in the example above is to tell Unix to run the executable in this directory (see *paths*), and the & at the end of the line tells it to run in background. The "[1] 15442" means that it is background job number 1 run from this shell and that it has the *processor id* 15442.

If you want to find out what jobs you have running in background and their pid's, try:

```
$ jobs -l
[1]+ 15443 Running                 ./fortrancode.exe &
```

You can bring the job back to the foreground with:

```
$ fg %1
```

Now you won't get a Unix prompt back until the job finishes (or you put it back into background as described below). The %1 refers to job 1. In this example *fg* alone would suffice since there's only one job running, but more generally you may have several in background.

To put a job that is foreground into background, you can often type <ctrl>-z, which will pause the job and give you the prompt back:

```
^Z
[1]+  Stopped                 ./fortrancode.exe
$
```

Note that the job is not running in background now, it is stopped. To get it running again in background, type:

```
$ bg %1
```

Or you could get it running in foreground with "fg %1".

2.2.6 nice and top

If you are running a code that will run for a long time you might want to make sure it doesn't slow down other things you are doing. You can do this with the *nice* command, e.g.:

```
$ nice -n 19 ./fortrancode.exe &
```

gives the job lowest priority (nice values between 1 and 19 can be used) so it won't hog the CPU if you're also trying to edit a file at the same time, for example.

You can change the priority of a job running in background with *renice*, e.g.:

```
$ renice -n 19 15443
```

where the last number is the process id.

Another useful command is *top*. This will fill your window with a page of information about the jobs running on your computer that are using the most resources currently. See *top* for some examples.

2.2.7 killing jobs

Sometimes you need to kill a job that's running, perhaps because you realize it's going to run for too long, or you gave it or the wrong input data. Or you may be running a program like the IPython shell and it freezes up on you with no way to get control back. (This sometimes happens when plotting when you give the *pylab.show()* command, for example.)

Many programs can be killed with <ctrl>-c. For this to work the job must be running in the foreground, so you might need to first give the *fg* command.

Sometimes this doesn't work, like when IPython freezes. Then try stopping it with <ctrl>-z (which should work), find out its PID, and use the *kill* command:

```
$ jobs -l
[1]+ 15841 Suspended               ipython

$ kill 15841
```

Hit return again you with luck you will see:

```
$
[1]+ Terminated              ipython
$
```

If not, more drastic action is needed with the -9 flag:

```
$ kill -9 15841
```

This almost always kills a process. Be careful what you kill.

2.2.8 sudo

A command like:

```
$ sudo rm 70-persistent-net.rules
```

found in the section *Virtual Machine for this class* means to do the remove command as super user. You will be prompted for your password at this point.

You cannot do this unless you are registered on a list of super users. You can do this on the VM because the *amath583* account has sudo privileges. The reason this is needed is that the file being removed here is a system file that ordinary users are not allowed to modify or delete.

Another example is seen at *apt-get*, where only those with super user permission can install software on to the system.

2.2.9 The bash shell

There are several popular shells for Unix. The examples given in these notes assume the bash shell is used. If you think your shell is different, you can probably just type:

```
$ bash
```

which will start a new bash shell and give you the bash prompt.

For more information on bash, see for example [Bash-Beginners-Guide], [gnu-bash], [Wikipedia-bash].

2.2.10 .bashrc file

Everytime you start a new bash shell, e.g. by the command above, or when you first log in or open a new window (assuming bash is the default), a file named ".bashrc" in your home directory is executed as a bash script. You can place in this file anything you want to have executed on startup, such as exporting environment variables, setting paths, defining aliases, setting your prompt the way you like it, etc. See below for more about these things.

2.2.11 Environment variables

The command *printenv* will print out any environment variables you have set, e.g.:

```
$ printenv
USER=rjl
HOME=/Users/rjl
PWD=/Users/rjl/uwamath583/sphinx
FC=gfortran
PYTHONPATH=/Users/rjl/claw4/trunk/python:/Applications/visit1.11.2/src/lib:
PATH=/opt/local/bin:/opt/local/sbin:/Users/rjl/bin
etc.
```

You can also print just one variable by, e.g.:

```
$ printenv HOME
/Users/rjl
```

or:

```
$ echo $HOME
/Users/rjl
```

The latter form has $HOME instead of HOME because we are actually *using* the variable in an echo command rather than just printing its value. This particular variable is useful for things like

    $ cd $HOME/uwamath583

which will go to the uwamath583 subdirectory of your home directory no matter where you start.

As part of Homework 1 you are instructed to define a new environment variable to make this even easier, for example by:

```
$ export CLASSHG=$HOME/uwamath583
```

Note there are no spaces around the =. This defines a new environment variable and *exports* it, so that it can be used by other programs you might run from this shell (not so important for our purposes, but sometimes necessary).

You can now just do:

```
$ cd $CLASSHG
```

to go to this directory.

Note that I have set an environment variable FC as:

```
$ printenv FC
gfortran
```

This environment variable is used in some Makefiles (see *makefiles*) to determine which Fortran compiler to use in compiling Fortran codes.

2.2.12 PATH and other search paths

Whenever you type a command at the Unix prompt, the shell looks for a program to run. This is true of built-in commands and also new commands you might define or programs that have been installed. To figure out where to look for such programs, the shell searches through the directories specified by the PATH variable (see *Environment variables* above). This might look something like:

```
$ printenv PATH
PATH=/usr/local/bin:/usr/bin:/Users/rjl/bin
```

This gives a list of directories to search through, in order, separated by ":". The PATH is usually longer than this, but in the above example there are 3 directories on the path. The first two are general system-wide repositories and the last one is my own *bin* directory (bin stands for binary since the executables are often binary files, though often the bin directory also contains shell scripts or other programs in text).

2.2.13 which

The *which* command is useful for finding out the full path to the code that is actually being executed when you type a command, e.g.:

```
$ which gfortran
/usr/bin/gfortran

$ which f77
$
```

In the latter case it found no program called f77 in the search path, either because it is not installed or because the proper diretory is not on the PATH.

Some programs require their own path to be set if it needs to search for input files. For example, you can set MATLABPATH or PYTHONPATH to be a list of directories (separated by ":") to search for .m files when you execute a command in Matlab, or for .py files when you import a module in Python.

2.2.14 Setting the prompt

If you don't like the prompt bash is using you can change it by changing the environment variable PS1, e.g.:

```
$ PS1='myprompt* '
myprompt*
```

This is now your prompt. There are various special characters you can use to specify things in your prompts, for example:

```
$ PS1='[\W] \h% '
[sphinx] aspen%
```

tells me that I'm currently in a directory named sphinx on a computer named aspen. This is handy to keep track of where you are, and what machine the shell is running on if you might be using ssh to connect to remote machines in some windows.

Once you find something you like, you can put this command in your .bashrc file.

2.2.15 Further reading

[Wikipedia-unix-utilities]

2.3 Using ssh to connect to remote computers

Some computers allow you to remotely log and start a Unix shell running using ssh (secure shell). To do so you generally type something like:

```
$ ssh username@host
```

where username is your account name on the machine you are trying to connect to and host is the host name (registered students can see the class Catalyst webpage for instructions on reaching Applied Math machines).

If you plan on running a program remotely that might pop up its own X-window, e.g. when doing plotting in Python or Matlab, you should use:

```
$ ssh -X username@host
```

2.3.1 sftp

To transfer files you can use sftp (secure file transfer protocol), e.g.:

```
$ sftp username@host
password:
sftp> cd somedirectory
sftp> get somefile
sftp> quit
```

which would copy somedirectory/somefile on the remote machine to your local directory. You will have to type your password before getting the sftp prompt.

2.4 Text editors

2.4.1 gedit

`gedit` is a simple, easy-to-use text editor with support for syntax highlighting for a variety of programming languages, including Python and Fortran. It is installed by default on most GNOME-based Linux systems, such as Ubuntu, and is included in the *Virtual Machine for this class*.

2.4.2 NEdit

Another easy-to-use editor, available from http://www.nedit.org/. NEdit is available for almost all Unix systems, including Linux and Mac OS X; OS X support requires an X Windows server.

2.4.3 XCode

XCode is a free integrated development environment available for Mac OS X from Apple at http://developer.apple.com/technologies/tools/xcode.html. It should be more than adequate for the needs of this class.

2.4.4 TextWrangler

TextWrangler is another free programmer's text editor for Mac OS X, available at http://www.barebones.com/products/TextWrangler/.

2.4.5 vi or vim

`vi` ("Visual Interface") / `vim` ("`vi` iMproved") is a fast, powerful text editor. Its interface is extremely different from modern editors, and can be difficult to get used to, but `vi` can offer substantially higher productivity for an experienced user. It is available for all operating systems; see http://www.vim.org/ for downloads and documentation. A command line version of `vi` is already installed in the class VM.

2.4.6 emacs

`emacs` ("Editing Macros") is another powerful text editor. Its interface may be slightly easier to get used to than that of `vi`, but it is still extremely different from modern editors, and is also extremely different from `vi`. It offers similar productivity benefits to `vi`. See http://www.gnu.org/software/emacs/ for downloads and documentation. On a Debian or Ubuntu Linux system, such as the class VM, you can install it by typing `sudo apt-get install emacs` at the command line.

2.4.7 Further reading

2.5 Version Control Software

In this class we will use Mercurial. See the section *Mercurial (hg)* for more information on using hg and the repositories required for this class.

There are many other version control systems that are currently popular, such as cvs, Subversion, Bazaar, and GIT. See [wikipedia-revision-control-software] for a much longer list with links. See [wikipedia-revision-control] for a general discussion of such systems.

Version control systems were originally developed to aid in the development of large software projects with many authors working on inter-related pieces. The basic idea is that you want to work on a file (one piece of the code), you check it out of a repository, make changes, and then check it back in when you're satisfied. The repository keeps track of all changes (and who made them) and can restore any previous version of a single file or of the state of the whole project. It does not keep a full copy of every file ever checked in, it keeps track of differences (*diff*s) between versions, so if you check in a version that only has one line changed from the previous version, only the characters that actually changed are kept track of.

It sounds like a hassle to be checking files in and out, but there are a number of advantages to this system that make version control an extremely useful tool even for use with you own projects if you are the only one working on something. Once you get comfortable with it you may wonder how you ever lived without it.

Advantages include:

- You can revert to a previous version of a file if you decide the changes you made are incorrect. You can also easily compare different versions to see what changes you made, e.g. where a bug was introduced.
- If you use a computer program and some set of data to produce some results for a publication, you can check in exactly the code and data used. If you later want to modify the code or data to produce new results, as generally happens with computer programs, you still have access to the first version without having to archive a full copy of all files for every experiment you do. Working in this manner is crucial if you want to be able to later reproduce earlier results, as if often necessary if you need to tweak the plots for to some journal's specifications or if a reader of your paper wants to know exactly what parameter choices you made to get a certain set of results. This is an important aspect of doing *reproducible research*, as should be required in science. (See Section *repro_research*). If nothing else you can save yourself hours of headaches down the road trying to figure out how you got your own results.
- If you work on more than one machine, e.g. a desktop and laptop, version control systems are one way to keep your projects synched up between machines.

2.5.1 Client-server systems

The original version control systems all used a client-server model, in which there is one computer that contains **the repository** and everyone else checks code into and out of that repository.

Systems such as CVS and Subversion (svn) have this form. An important feature of these systems is that only the repository has the full history of all changes made.

There is a software-carpentry webpage on version control that gives a brief overview of client-server systems.

2.5.2 Distributed systems

Mercurial, and other systems such as Bazaar and GIT, use a distributed system in which there is not necessarily a "master repository". Any working copy contains the full history of changes made to this copy.

The best way to get a feel for how Mercurial works is to use it, for example by following the instructions in Section *Mercurial (hg)*.

2.6 Mercurial (hg)

See *Version Control Software* and the links there for a more general discussion of the concepts.

2.6.1 Instructions for cloning the class repository

All of the materials for this class, including slides, sample programs, and the webpages you are now reading (or at least the *.rst* files used to create them, see *sphinx*), are in a Mercurial repository hosted at Bitbucket, located at http://bitbucket.org/rjleveque/uwamath583s11/. In addition to viewing the files via the link above, you can also view changesets, issues, etc. (see *Bitbucket repositories: viewing changesets, issue tracking*).

To obtain a copy, simply move to the directory where you want your copy to reside (assumed to be your home directory below) and then *clone* the repository:

```
$ cd
$ hg clone http://bitbucket.org/rjleveque/uwamath583s11/
```

Note the following:

- It is assumed you have Mercurial (hg) installed, see *Downloading and installing software for this class*.
- The clone statement will download the entire repository as a new subdirectory called *uwamath583s11*, residing in your home directory. If you want *uwamath583s11* to reside elsewhere, you should first *cd* to that directory.

It will be useful to set a Unix environment variable (see *Environment variables*) called *CLASSHG* to refer to the directory you have just created. Assuming you are using the bash shell (see *The bash shell*), and that you cloned uwamath583s11 into your home directory, you can do this via:

```
$ export CLASSHG=$HOME/uwamath583s11
```

This uses the standard environment variable *HOME*, which is the full path to your home directory.

If you put it somewhere else, you can instead do:

```
$ cd uwamath583s11
$ export CLASSHG=`pwd`
```

The syntax *`pwd`* means to run the *pwd* command (print working directory) and insert the output of this command into the export command.

Type:

```
$ printenv CLASSHG
```

to make sure *CLASSHG* is set properly. This should print the full path to the new directory.

If you log out and log in again later, you will find that this environment variable is no longer set. Or if you set it in one terminal window, it will not be set in others. To have it set automatically every time a new bash shell is created (e.g. whenever a new terminal window is opened), add a line of the form:

```
export CLASSHG=$HOME/uwamath583s11
```

to your *.bashrc* file. (See *.bashrc file*). This assumes it is in your home directory. If not, you will have to add a line of the form:

```
export CLASSHG=full-path-to-uwamath583s11
```

where the full path is what was returned by the *printenv* statement above.

2.6.2 Updating your clone

The files in the class repository will change as the quarter progresses — new notes, slides, sample programs, and homeworks will be added. In order to bring these changes over to your cloned copy, all you need to do is:

```
$ cd $CLASSHG
$ hg pull
$ hg update
```

Of course this assumes that *CLASSHG* has been properly set, see above.

The last two command can be combined as:

```
$ hg pull -u
```

2.6.3 Creating your own Bitbucket repository

In addition to using the class repository, students in AMath 483/583 are also required to create their own repository on Bitbucket. It is possible to use Mercurial for your own work without creating a repository on a hosted site such as Bitbucket (see *newhg* below), but there are several reasons for this requirement:

- You should learn how to use Bitbucket for more than just pulling changes.
- You will use this repository to "submit" your solutions to homeworks. You will give the instructor and TA permission to clone your repository so that we can grade the homework (others will not be able to clone or view it unless you also give them permission).
- It is recommended that after the class ends you continue to use your repository as a way to back up your important work on another computer (with all the benefits of version control too!). At that point, of course, you can change the permissions so the instructor and TA no longer have access.

Below are the instructions for creating your own repository. Note that this should be a *private repository* so nobody can view or clone it unless you grant permission.

Anyone can create one free private repository on Bitbucket. More than one costs money. If you already have a private repository that you don't want to give us access to, contact us and we can arrange something else. (Note that you can also create an unlimited number of public repositories free at Bitbucket, which you might want to do for open source software projects, or for classes like this one.)

Follow these directions exactly. Doing so is part of *Homework 1*. We will clone your repository and check that *testfile.txt* has been created and modified as directed below.

1. On the machine you're working on, go to your home directory ("cd $HOME" on Unix) and create a file named .hgrc (with a dot at the front) containing:

   ```
   [ui]
   username = Your Name <yournetid@uw.edu>
   verbose = True
   ```

   This will be used when you commit changes. This is a configuration file used by hg, similar to the *.bashrc file* used by bash. If you don't do this, you might get a message like:

   ```
   abort: no username supplied (see "hg help config")
   ```

   the first time you try to commit.

2. Go to http://bitbucket.org/ and click on "Sign up now"
3. Fill in the form, make sure you remember your username and password.
4. You should then be taken to your account. Click on "+ Create new" next to "Your repositories".
5. You should now see a form where you can specify the name of a repository and a description. The repository name need not be the same as your user name (a single user might have several repositories). For example, the class repository is named *uwamath583s11*, owned by user *rjleveque*. To avoid confusion, you should probably not name your repository *uwamath583s11*.
6. Make sure you click on "Private" at the bottom. Also keep "Issue tracking" and "Wiki" clicked on.
7. Click on "Create repository".
8. You should now see a page with instructions on how to *clone* your (currently empty) repository. In a Unix window, *cd* to the directory where you want your cloned copy to reside, and perform the clone by typing in the clone command shown. This will create a new directory with the same name as the repository.
9. You should now be able to *cd* into the directory this created.
10. To keep track of where this directory is and get to it easily in the future, create an environment variable *MYHG* from inside this directory by:

    ```
    $ export MYHG=`pwd`
    ```

    See the discussion above in section *Instructions for cloning the class repository* for what this does. You will also probably want to add a line to your *.bashrc* file to define MYHG similar to the line added for *CLASSHG*.

11. The directory you are now in will appear empty if you simply do:

    ```
    $ ls
    ```

    But try:

    ```
    $ ls -a
    ./  ../  .hg/
    ```

    the *-a* option causes *ls* to list files starting with a dot, which are normally suppressed. See *ls* for a discussion of *./* and *../*. The directory *.hg* is the directory that stores all the information about the contents of this directory and a complete history of every file and every change ever committed. You shouldn't touch or modify the files in this directory, they are used by Mercurial.

12. Add a new file to your directory:

```
$ cat > testfile.txt
This is a new file
with only two lines so far.
^D
```

The Unix *cat* command simply redirects everything you type on the following lines into a file called *testfile.txt*. This goes on until you type a <ctrl>-d (the 4th line in the example above). After typing <ctrl>-d you should get the Unix prompt back. Alternatively, you could create the file testfile.txt using your favorite text editor (see *Text editors*).

13. Type:

```
$ hg status
```

The response should be:

```
? testfile.txt
```

The ? means that this file is not under revision control. To put it under revision control, type:

```
$ hg add testfile.txt
$ hg status
A testfile.txt
```

The A means it has been added. However, at this point Mercurial is not yet keeping track of this version or any changes to this file. We can tell Mercurial to track this version by:

```
$ hg commit testfile.txt -m "My first commit of a test file."
```

The string following the -m is a comment about this commit that may help you in general remember why you committed new or changed files.

If you left off the file name *testfile.txt* then be default hg will commit all changes made in this directory since the last commit. In the current example there is only one change, the addition of this file, so it would have the same behavior either way.

We can now see the status of our directory via:

```
$ hg status --all
C testfile.txt
```

Now the C means this file is *clean* (agrees with the version being tracked). If you leave off the *–all* flag, by default *hg status* does not list the clean files and only shows files that differ from the lastest version being tracked. This is generally what you want if you have thousands of files in your repository and only a few have been changed.

Alternatively, you can check the status of a single file with:

```
$ hg status testfile.txt
```

Now let's modify this file:

```
$ cat >> testfile.txt
Adding a third line
^D
```

Here the >> tells *cat* that we want to add on to the end of an existing file rather than creating a new one. (Or you can edit the file with your favorite editor and add this third line.)

Now try the following:

```
$ hg status
M testfile.txt
```

The M indicates this file has been modified relative to the most recent version that was committed.

To see what changes have been made, try:

```
$ hg diff testfile.txt
```

This will produce something like:

```
diff -r 6e3383561936 testfile.txt
--- a/testfile.txt      Tue Mar 02 23:15:38 2010 -0800
+++ b/testfile.txt      Tue Mar 02 23:23:57 2010 -0800
@@ -1,2 +1,3 @@ This is a new file
 This is a new file
 with only two lines so far
+Adding a third line
```

The + in front of the last line shows that it was added. The two lines before it are printed to show the context. If the file were longer, *hg diff* would only print a few lines around any change to indicate the context.

The first line printed out above shows the diff command that's actually being executed: it is doing a diff between this file and the one stored in the revision that Mercurial has given the name 6e3383561936. Internally it creates long hexidecimal names like this (see *hex*) in order to keep things straight if changes made by several different people are merged. There's a simpler revision number you can discover by typing:

```
$ hg tip
changeset:   0:6e3383561936
tag:         tip
user:        rjl@spinach
date:        Tue Mar 02 23:15:38 2010 -0800
summary:     My first commit of a test file.
```

This shows that the *tip* (the most recent version of all files being tracked) is changeset 0.

Now let's commit this changed file:

```
$ hg commit -m "added a third line to the test file"
```

After this, "hg tip" should indicate the changeset of the tip is 1 (along with a long hex string).

14. So far you have been using Mercurial to keep track of changes in your own directory, on your computer. None of these changes have been seen by Bitbucket, so if someone else cloned your repository from there, they would not see *testfile.txt*.

    Now let's *push* these changes back to the Bitbucket repository:

    First do:

    ```
    $ hg status
    ```

to make sure there are no changes that have not been committed. This should print nothing.

Now do:

```
$ hg push
```

This will prompt for your Bitbucket password and should then print something like:

```
searching for changes
adding changesets
adding manifests
adding file changes
added 2 changesets with 2 changes to 1 files
```

Not only has it copied the 1 file over, it has added both changesets, so the entire history of your commits is now stored in the repository. If someone else clones the repository, they get the entire commit history and could revert to any previous version, for example (see *Reverting to a previous version*).

15. Check that the file is in your Bitbucket repository: Go back to that web page for your repository and click on the "Source" tab at the top. It should display the files in your repository and show *testfile.txt*.

    Now click on the "Changesets" tab at the top. It should show that you made two commits and display the comments you added with the *-m* flag with each commit.

    Look on the right side, where it says "commit 1: hex-string". The hex-string is the internal identifier of this commit. If you click on the hex-string it will show the *change set* for this commit. What you should see is the file in its final state, with three lines. The third line should be highlighted in green, indicating that this line was added in this changeset. A line highlighted in red would indicate a line deleted in this changeset. (See also *Bitbucket repositories: viewing changesets, issue tracking*.)

This is enough for now!

Homework 1 instructs you to add some additional files to the Bitbucket repository.

Feel free to experiment further with your repository at this point. See *More commands in hg* for some more examples of how hg can be used.

2.6.4 Further reading

Next see *Bitbucket repositories: viewing changesets, issue tracking*.

Remember that you can get help with Mercurial commands by typing, e.g.:

```
$ hg help
$ hg diff help
```

See [Bitbucket-hg-intro] for an overview or [hgbook] for complete details, [hg-faq] for an FAQ, and [hg-hgrc] for more about configuration files.

The site <*http://mercurial.selenic.com/wiki/*> has many other useful things to read. In particular, to help get started, see:

- '**QuickStart** <http://mercurial.selenic.com/wiki/QuickStart>'_
- '**Tutorial** <http://mercurial.selenic.com/wiki/Tutorial>'_
- '**ChangeSetComments** <http://mercurial.selenic.com/wiki/ChangeSetComments>'_
- '**UnderstandingMercurial** <http://mercurial.selenic.com/wiki/UnderstandingMercurial>'_
- '**RepositoryNaming** <http://mercurial.selenic.com/wiki/RepositoryNaming>'_

2.7 More commands in hg

2.7.1 Reverting to a previous version

This is a continuation of the example in *Creating your own Bitbucket repository*.

1. Now try the following:

```
$ rm -f testfile.txt
$ hg status
! testfile.txt
```

The first line removes this file (the *-f* flag forces this without prompting the user to make sure that's what's intended, since removing files can be dangerous).

The "hg status" command now shows ! next to the file name, indicating that a file under version control has disappeared.

Ordinarily if you remove a file in Unix it's gone. Period. But with version control, we can easily recover from such a blunder:

```
$ hg revert testfile.txt
$ hg status --all
C testfile.txt
```

The "hg revert" has restored this file using the most recent version committed. You can also use "hg revert" if you make some changes to a file and then decide they were a bad idea.

Not only can you revert to the most recent version (in the *tip*), you can revert to any previous version that was committed.

Try this:

```
$ hg revert -r 0 testfile.txt
$ hg status
M testfile.txt

$ hg diff testfile.txt
diff -r 11d71622c220 testfile.txt
--- a/testfile.txt      Tue Mar 02 23:33:52 2010 -0800
+++ b/testfile.txt      Wed Mar 03 00:01:29 2010 -0800
@@ -1,3 +1,2 @@ This is a new file
 This is a new file
 with only two lines so far
-Adding a third line
```

Here we have reverted to the version in changeset 0 (the first time we committed, when the file only had two lines). Now "hg status" shows that it is modified (relative to the *tip*, which has three lines) and "hg diff" shows the change relative to the *tip*: the third line was removed.

2. Revert back to the tip:

```
$ hg revert --no-backup testfile.txt
```

The file should now have 3 lines again. If the *–no-backup* flag is omitted, Mercurial will create a file *testfile.txt.orig* with the modified version, just in case you regret your *revert*. (Not all revision control systems have this feature!)

2.7.2 Further reading

2.8 Bitbucket repositories: viewing changesets, issue tracking

The directions in Section *Mercurial (hg)* explain how to use the class Bitbucket repository to download these class notes and other resources used in the class, and also how to set up your own repository there.

In addition to providing a hosting site for repositories to allow them to be easily shared, Bitbucket provides other web-based resources for working with a repository. (So do other sites based on Mercurial or other version control software, such as github, for example, a repository based on GIT).

To get a feel for what's possible, take a look at one of the major software projects hosted on bitbucket, for example http://bitbucket.org/birkenfeld/sphinx/ which is the repository for the Sphinx software used for these class notes pages (see *sphinx*). You will see that software is being actively developed.

If you click on the "Source" tab at the top of the page you can browse through the source code repository in its current state.

If you click on the "Changesets" tab you can see all the changes ever committed, with the message that was written following the -m flag when the commit was made. If you click on one of these messages, it will show all the changes in the form of the lines of the files changed, highlighted in green for things added or red for things deleted.

If you click on the "Issues" tab, you will see the issue-tracking page. If someone notices a bug that should be fixed, or thinks of an improvement that should be made, a new issue can be created (called a "ticket" in some systems).

If you want to try creating a ticket, **don't** do it on the Sphinx page, the developers won't appreciate it. Instead try doing it on your own bitbucket repository that you set up following *Creating your own Bitbucket repository*.

You might also want to look at the bitbucket page for this class repository, at http://bitbucket.org/rjleveque/uwamath583s11/ to keep track of changes made to notes or homework assignments.

If you find major errors in the notes, feel free to use the issue tracker to report it. But please don't point out incomplete sections (there are many!) or suggest other topics to be added at this point.

2.8.1 Further reading

2.9 Mercurial examples

For some simple examples elsewhere, see:

- http://hginit.com/
- Other references at *Mercurial and other version control systems:*

2.9.1 Examples for Lecture 3

This example shows how one might create and use a stand-alone repository for one project, showing how you might use Mercurial for a single directory with no cloning, no bitbucket repository, etc.

At the end we'll also look at how it can be cloned.

We'll go through this example with more detail in Lecture 3.

Make a new directory:

```
$ cd
$ mkdir myproject
```

Confirm it's empty:

```
$ cd myproject
/Users/rjl/myproject
$ ls -a
./  ../
```

Copy some files here so we'll have something to start with:

```
$ cp $CLASSHG/codes/python/myfcns*.py .
$ ls
myfcns.py   myfcns2.py
```

We're not yet using hg:

```
$ hg status
abort: There is no Mercurial repository here (.hg not found)!
```

2.9.2 hg init

Let's create a repository for this directory:

```
$ hg init
$ ls -a
./      ../     .hg/        myfcns.py   myfcns2.py
```

No files have been committed yet:

```
$ hg status
? myfcns.py
? myfcns2.py
```

Let's add everything in this directory (and recursively in any subdirectories, if there were any):

```
$ hg add
adding myfcns.py
adding myfcns2.py

$ hg status
A myfcns.py
A myfcns2.py
```

Now commit them:

```
$ hg commit -m "Initial commit of files copied from $CLASSHG"
myfcns.py
myfcns2.py
committed changeset 0:2a3c30a21b10

$ hg status
```

The last command prints nothing since the directory is "clean"

To force printing status of each file (C = clean):

```
$ hg status -A
C myfcns.py
C myfcns2.py
```

Now edit *myfcns.py* so the function *f2* is *x**4* instead of the exponential... [not shown here]

Now:

```
$ hg diff
diff -r 2a3c30a21b10 myfcns.py
--- a/myfcns.py Thu Apr 01 18:45:59 2010 -0700
+++ b/myfcns.py Thu Apr 01 18:56:44 2010 -0700
@@ -12,8 +12,9 @@

 def f2(x):
     """
-    An exponential function.
+    A power function.
     """
     from numpy import exp
-    y = exp(4.*x)
+    y = x**4
     return y
+
```

Commit this change:

```
$ hg status
M myfcns.py

$ hg commit -m "Changed f2 to be a power function"
myfcns.py
committed changeset 1:77711e71d739
```

Now let's change the power from 4 to 5 [editing not shown]:

```
$ hg diff
diff -r 77711e71d739 myfcns.py
--- a/myfcns.py Thu Apr 01 18:58:00 2010 -0700
+++ b/myfcns.py Thu Apr 01 18:59:21 2010 -0700
@@ -15,6 +15,6 @@
     A power function.
     """
     from numpy import exp
-    y = x**4
+    y = x**5
     return y
```

The file is modified in the working copy, but not yet in the repository since we haven't committed.

2.9.3 hg revert

Suppose after testing we decide this was a bad idea and want to revert back to what we had before changing the working copy:

```
$ hg status
M myfcns.py

$ hg revert myfcns.py
saving current version of myfcns.py as myfcns.py.orig
reverting myfcns.py
```

Now myfcns.py is back to the power function with exponent 4. Note that hg has also saved a copy of the working file before the revert as *myfcns.py.orig*, just in case you regret the revert. This file is not under version control:

```
$ hg status
? myfcns.py.orig
```

We could add it, but probably we don't want to, not with this name anyway. Instead we'll probably eventually delete it.

2.9.4 hgignore

You can tell hg to ignore certain files when it's reporting on the status of the repository. This is useful for files of the form **.orig*, for example, and also for files that are produced as the result of compiling or running a code, that you generally don't want to place under version control. For example, we'll see later that a python file like *myfcns.py* can lead to a file *myfcns.pyc* (compiled version) in some cases, so we might want hg to ignore all files of the form **.pyc*. Also, compiling a fortran program in *fname.f90* produces an *object code* file *fname.o* that we also won't want to keep under version control.

You can print the status ignoring certain files with the *-X* flag:

```
$ hg status -X *.orig
```

But for file types you always want to ignore, it's easier to create a file *.hgignore* to do this once and for all. This file should be in the same directory as *.hg* and contain, for example:

```
syntax: glob
*.orig
*.pyc
*.o
```

to ignore the types of files just described. *glob* refers to pattern matching.

See also:

- http://en.wikipedia.org/wiki/Glob_%28programming%29.
- http://www.selenic.com/mercurial/hgignore.5.html

Now *hg status* will ignore the **.orig* file, but note that we've created a new file that we probably do want to put under version control (so if we ever clone this repository it will go along with it):

```
$ hg status
? .hgignore

$ hg add .hgignore
```

```
adding .hgignore

$ hg commit -m "added .hgignore file to ignore some files"
.hgignore
committed changeset 2:0367a3426fc9
```

2.9.5 hg diff

You can use *hg diff* to print the diffs between a file in your working directory and the one at the tip of the repository, or using the *-r* flag between two different revisions, as in the example on the Lecture 2 slides. You can also use xxdiff as done there.

2.9.6 hg serve

If you want a web-page view similar to what bitbucket provides, you can even run your own web server specific to this repository. In a different window (since this will print out stuff to the window and make it hard to work in it), do:

```
$ hg serve
```

and then point your web browser to *http://localhost:8000/*. You should see a log of this directory. Try clicking on one of the commit messages to see the changeset.

To kill the server, type <ctrl>-c in the window where you started it.

If port 8000 is already in use (e.g. you want to serve multiple repositories at once), you can also specify a different port, e.g.

```
$ hg serve -p 8001
```

2.9.7 Cloning this repository

Suppose you want to try out a new idea that will require many changes and you don't want to disturb this directory, which you're still using for other purposes. (For our simple example, maybe we want to replace the power function with the sine function.)

Then you could make a clone:

```
$ cd ..
$ hg clone myproject myproject-branch-for-sine
updating to branch default
resolving manifests
getting .hgignore
getting myfcns.py
getting myfcns2.py
3 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

Now we can go into this version and make the desired change. The new clone is an independent repository that might go off in a different direction, or it might eventually be merged back into the original one.

2.9.8 Removing .hg

Suppose you decide you no longer want a directory to be under version control. For example, you are done debugging and know you will never again want to refer to the past history. Then you could do:

```
$ rm -r .hg
```

The *-r* flag means recursively, so the directory *.hg* will be removed along with all files in it (and all subdirectories, and files in them, etc.). If *rm* is set to ask you for confirmation before removing each file, you can *force* it to remove everything without asking by:

```
$ rm -rf .hg
```

Warning: If this is your only copy of the repository (i.e. there are no clones) and you do this, all your revision history is gone forever! Make very sure you are in the right directory first!

This is all you need to do to remove version control, you're left with an ordinary directory.

2.9.9 Making a copy of a directory

Perhaps you want to keep using version control in *myproject*, but you want to make a copy of the directory that does not have the *.hg* subdirectory, for example to turn into a tar file [wikipedia-tar] that you can send to others or post on a website. To do this, simply clone the repository and then go into the clone and remove the *.hg* directory, leaving a clean copy:

```
$ cd
$ clone myproject myproject-copy
$ cd myproject-copy
$ rm -rf .hg
```

Note: Cloning the repository only copies the files committed, not all the files in the working directory that have not been added, or any local modifications not committed.

If you want a copy of **all** files in the working directory, in their current state, you could instead do:

```
$ cd
$ cp -r myproject myproject-copy
$ cd myproject-copy
$ rm -rf .hg
```

Warning: Think twice before any *rm -rf* operation!

2.10 Bibliography and further reading

See Also:

Links from 2009 seminar

Many other pages in these notes have links not listed below. These are some references that are partiuclarly useful or are cited often elsewhere.

2.10.1 Books

2.10.2 Other courses with useful slides or webpages

2.10.3 Other Links

Software

See Also:

Downloading and installing software for this class for links to software download pages.

Virtual machine:

Sphinx:

Python:

Numerical Python

Unix, bash:

Mercurial and other version control systems:

Fortran

Many tutorials and references are available online. Search for "fortran 90 tutorial" or "fortran 95 tutorial" to find many others.

Makefiles:

Computer architecture

Floating point arithmetic

Languages and compilers

OpenMP:

Exa-scale computing

More will be added, check back later

CHAPTER

THREE

APPLICATIONS

BIBLIOGRAPHY

[Lin-Snyder] C. Lin and L. Snyder, *Principles of Parallel Programming*, 2008.

[Scott-Clark-Bagheri] L. R. Scott, T. Clark, B. Bagheri, *Scientific Parallel Computing*, Princeton University Press, 2005.

[McCormack-scientific-fortran] D. McCormack, *Scientific Software Development in Fortran*, Lulu Press, ... ebook ... paperback **(A copy is on reserve in the Engineering Library)**

[Chandra-etal-openmp] R. Chandra, L. Dagum, et. al., *Parallel Programming in OpenMP*, Academic Press, 2001.

[Gropp-Lusk-Skjellum-MPI] W. Gropp, E. Lusk, A. Skjellum, *Using MPI*, Second Edition, MIT Press, 1999. Google books

[Dive-into-Python] M. Pilgram, *Dive Into Python*, http://www.diveintopython.org/.

[Python] G. van Rossum, *An Introduction to Python*, http://www.network-theory.co.uk/docs/pytut/index.html

[Langtangen-scripting] H. P. Langangen, *Python Scripting for Computational Science*, 3rd edition, Springer, 2008. book and scripts ... lots of slides

[Langtangen-Primer] H. P. Langtangen, *A Primer on Scientific Programming with Python*, Springer 2009 What's the difference from the previous one?

[Goedecker-Hoisie-optimization] S. Goedecker and A. Hoisie, *Performance Optimization of Numerically intensive Codes*, SIAM 2001. **(A copy is on reserve in the Engineering Library)**

[Matloff-Salzman-debugging] N. Matloff and P. J. Salzman, *The Art of Debugging with GDB, DDD, and Eclipse*, no starch press, San Francisco, 2008.

[Overton-IEEE] M. Overton, *Numerical Computing with IEEE Floating Point Arithmetic*, SIAM, 2001.

[software-carpentry] Greg Wilson, http://software-carpentry.org/.

[Snyder-UW-CSE524] Larry Snyder, UW CSE 524, Parallel Algorithms

[Gropp-UIUC] William Gropp UIUC Topics in HPC

[Yelick-UCB] Kathy Yelick, Berkeley course on parallel computing

[Demmel-Simon-UCB] Jim Demmel and Horst Simon, Berkeley course on parallel computing

[Berger-Bindel-NYU] Marsha Berger and David Bindel, NYU course

[LLNL-HPC] Livermore HPC tutorials

[NERSC-tutorials] NERSC tutorials

[HPC-University] http://www.hpcuniv.org/resources/list/

[CosmicProject] links to open source software

[VirtualBox] http://www.virtualbox.org/

[VirtualBox-documentation] http://www.virtualbox.org/wiki/Documentation

[sphinx] http://sphinx.pocoo.org

[sphinx-documentation] http://sphinx.pocoo.org/contents.html

[sphinx-rst] http://sphinx.pocoo.org/rest.html

[rst-documentation] http://docutils.sourceforge.net/rst.html

[sphinx-cheatsheet] http://matplotlib.sourceforge.net/sampledoc/cheatsheet.html

[sphinx-examples] http://sphinx.pocoo.org/examples.html

[sphinx-sampledoc] http://matplotlib.sourceforge.net/sampledoc/index.html

[Python-2.5-tutorial] http://www.python.org/doc/2.5.2/tut/tut.html

[Python-2.6-tutorial] http://docs.python.org/tutorial/

[Python-3.0-tutorial] http://docs.python.org/dev/3.0/tutorial/ (we are *not* using Python 3.0 in this class!)

[IPython-documentation] http://ipython.scipy.org/doc/stable/html/

[IPython-links] http://ipython.scipy.org/moin/Documentation

[Python-pdb] **'Python debugger documentation <http://docs.python.org/library/pdb.html'_**

[IPython-pdb] Using pdb from IPython

[NumPy-tutorial] http://www.scipy.org/Tentative_NumPy_Tutorial

[NumPy-reference] http://docs.scipy.org/doc/numpy/reference/

[NumPy-SciPy-docs] http://docs.scipy.org/doc/

[NumPy-for-Matlab-Users] http://www.scipy.org/NumPy_for_Matlab_Users

[NumPy-pros-cons] http://www.scipy.org/NumPyProConPage

[Numerical-Python-links] http://wiki.python.org/moin/NumericAndScientific

[Wikipedia-unix-utilities] http://en.wikipedia.org/wiki/List_of_Unix_utilities

[Bash-Beginners-Guide] http://tldp.org/LDP/Bash-Beginners-Guide/html/index.html

[gnu-bash] http://www.gnu.org/software/bash/bash.html

[Wikipedia-bash] http://en.wikipedia.org/wiki/Bash

[wikipedia-tar] http://en.wikipedia.org/wiki/Tar_%28file_format%29

[hgbook] http://hgbook.red-bean.com/

[Bitbucket-hg-intro] http://bitbucket.org/help/GettingStartedWithMercurial

[hg-faq] http://mercurial.selenic.com/wiki/FAQ

[hg-hgrc] http://www.selenic.com/mercurial/hgrc.5.html

[wikipedia-revision-control] http://en.wikipedia.org/wiki/Revision_control

[wikipedia-revision-control-software] http://en.wikipedia.org/wiki/List_of_revision_control_software

[Shene-fortran] C.-K. Shene's Fortran 90 tutorial

[Dodson-fortran] Zane Dodson's Fortran 90 tutorial

[fortran-tutorials] Links to a few other tutorials

[carpentry-make] http://software-carpentry.org/build.html

[gnu-make] http://www.gnu.org/software/make/manual/make.html

[make-tutorial] http://mrbook.org/tutorials/make/

[Wikipedia-make] http://en.wikipedia.org/wiki/Make_%28software%29

[wikipedia-computer-architecture] http://en.wikipedia.org/wiki/Computer_architecture

[wikipedia-memory-hierachy] http://en.wikipedia.org/wiki/Memory_hierarchy

[wikipedia-moores-law] http://en.wikipedia.org/wiki/Moore%27s_law.

[Arnold-disasters] Doug Arnold's descriptions of some disasters due to bad numerical computing, http://www.ima.umn.edu/~arnold/disasters/

[wikipedia-machine-code] http://en.wikipedia.org/wiki/Machine_code

[wikipedia-assembly] http://en.wikipedia.org/wiki/Assembly_language

[wikipedia-compilers] http://en.wikipedia.org/wiki/Compilers

[openmp.org] http://openmp.org/wp/

[openmp-gfortran] http://gcc.gnu.org/onlinedocs/gfortran/OpenMP.html

[openmp-gfortran2] http://sites.google.com/site/gfortransite/

[openmp-api3.0] http://www.openmp.org/mp-documents/spec30.pdf

[openmp-refcard] OpenMP in Fortran Reference card

[openmp-fort90-examples] http://people.sc.fsu.edu/~burkardt/f_src/open_mp/open_mp.html

[exascale-doe] Modeling and Simulation at the Exascale for Energy and the Environment, DOE Town Hall Meetings Report

[exascale-sc08] http://www.lbl.gov/CS/html/SC08ExascalePowerWorkshop/index.html

[Lin-Snyder] C. Lin and L. Snyder, *Principles of Parallel Programming*, 2008.

[Scott-Clark-Bagheri] L. R. Scott, T. Clark, B. Bagheri, *Scientific Parallel Computing*, Princeton University Press, 2005.

[McCormack-scientific-fortran] D. McCormack, *Scientific Software Development in Fortran*, Lulu Press, ... ebook ... paperback **(A copy is on reserve in the Engineering Library)**

[Chandra-etal-openmp] R. Chandra, L. Dagum, et. al., *Parallel Programming in OpenMP*, Academic Press, 2001.

[Gropp-Lusk-Skjellum-MPI] W. Gropp, E. Lusk, A. Skjellum, *Using MPI*, Second Edition, MIT Press, 1999. Google books

[Dive-into-Python] M. Pilgram, *Dive Into Python*, http://www.diveintopython.org/.

[Python] G. van Rossum, *An Introduction to Python*, http://www.network-theory.co.uk/docs/pytut/index.html

[Langtangen-scripting] H. P. Langangen, *Python Scripting for Computational Science*, 3rd edition, Springer, 2008. book and scripts ... lots of slides

[Langtangen-Primer] H. P. Langtangen, *A Primer on Scientific Programming with Python*, Springer 2009 What's the difference from the previous one?

[Goedecker-Hoisie-optimization] S. Goedecker and A. Hoisie, *Performance Optimization of Numerically intensive Codes*, SIAM 2001. **(A copy is on reserve in the Engineering Library)**

[Matloff-Salzman-debugging] N. Matloff and P. J. Salzman, *The Art of Debugging with GDB, DDD, and Eclipse*, no starch press, San Francisco, 2008.

[Overton-IEEE] M. Overton, *Numerical Computing with IEEE Floating Point Arithmetic*, SIAM, 2001.

[software-carpentry] Greg Wilson, http://software-carpentry.org/.

[Snyder-UW-CSE524] Larry Snyder, UW CSE 524, Parallel Algorithms

[Gropp-UIUC] William Gropp UIUC Topics in HPC

[Yelick-UCB] Kathy Yelick, Berkeley course on parallel computing

[Demmel-Simon-UCB] Jim Demmel and Horst Simon, Berkeley course on parallel computing

[Berger-Bindel-NYU] Marsha Berger and David Bindel, NYU course

[LLNL-HPC] Livermore HPC tutorials

[NERSC-tutorials] NERSC tutorials

[HPC-University] http://www.hpcuniv.org/resources/list/

[CosmicProject] links to open source software

[VirtualBox] http://www.virtualbox.org/

[VirtualBox-documentation] http://www.virtualbox.org/wiki/Documentation

[sphinx] http://sphinx.pocoo.org

[sphinx-documentation] http://sphinx.pocoo.org/contents.html

[sphinx-rst] http://sphinx.pocoo.org/rest.html

[rst-documentation] http://docutils.sourceforge.net/rst.html

[sphinx-cheatsheet] http://matplotlib.sourceforge.net/sampledoc/cheatsheet.html

[sphinx-examples] http://sphinx.pocoo.org/examples.html

[sphinx-sampledoc] http://matplotlib.sourceforge.net/sampledoc/index.html

[Python-2.5-tutorial] http://www.python.org/doc/2.5.2/tut/tut.html

[Python-2.6-tutorial] http://docs.python.org/tutorial/

[Python-3.0-tutorial] http://docs.python.org/dev/3.0/tutorial/ (we are *not* using Python 3.0 in this class!)

[IPython-documentation] http://ipython.scipy.org/doc/stable/html/

[IPython-links] http://ipython.scipy.org/moin/Documentation

[Python-pdb] **'Python debugger documentation <http://docs.python.org/library/pdb.html'_**

[IPython-pdb] Using pdb from IPython

[NumPy-tutorial] http://www.scipy.org/Tentative_NumPy_Tutorial

[NumPy-reference] http://docs.scipy.org/doc/numpy/reference/

[NumPy-SciPy-docs] http://docs.scipy.org/doc/

[NumPy-for-Matlab-Users] http://www.scipy.org/NumPy_for_Matlab_Users

[NumPy-pros-cons] http://www.scipy.org/NumPyProConPage

[Numerical-Python-links] http://wiki.python.org/moin/NumericAndScientific

[Wikipedia-unix-utilities] http://en.wikipedia.org/wiki/List_of_Unix_utilities

[Bash-Beginners-Guide] http://tldp.org/LDP/Bash-Beginners-Guide/html/index.html

[gnu-bash] http://www.gnu.org/software/bash/bash.html

[Wikipedia-bash] http://en.wikipedia.org/wiki/Bash

[wikipedia-tar] http://en.wikipedia.org/wiki/Tar_%28file_format%29

[hgbook] http://hgbook.red-bean.com/

[Bitbucket-hg-intro] http://bitbucket.org/help/GettingStartedWithMercurial

[hg-faq] http://mercurial.selenic.com/wiki/FAQ

[hg-hgrc] http://www.selenic.com/mercurial/hgrc.5.html

[wikipedia-revision-control] http://en.wikipedia.org/wiki/Revision_control

[wikipedia-revision-control-software] http://en.wikipedia.org/wiki/List_of_revision_control_software

[Shene-fortran] C.-K. Shene's Fortran 90 tutorial

[Dodson-fortran] Zane Dodson's Fortran 90 tutorial

[fortran-tutorials] Links to a few other tutorials

[carpentry-make] http://software-carpentry.org/build.html

[gnu-make] http://www.gnu.org/software/make/manual/make.html

[make-tutorial] http://mrbook.org/tutorials/make/

[Wikipedia-make] http://en.wikipedia.org/wiki/Make_%28software%29

[wikipedia-computer-architecture] http://en.wikipedia.org/wiki/Computer_architecture

[wikipedia-memory-hierachy] http://en.wikipedia.org/wiki/Memory_hierarchy

[wikipedia-moores-law] http://en.wikipedia.org/wiki/Moore%27s_law.

[Arnold-disasters] Doug Arnold's descriptions of some disasters due to bad numerical computing, http://www.ima.umn.edu/~arnold/disasters/

[wikipedia-machine-code] http://en.wikipedia.org/wiki/Machine_code

[wikipedia-assembly] http://en.wikipedia.org/wiki/Assembly_language

[wikipedia-compilers] http://en.wikipedia.org/wiki/Compilers

[openmp.org] http://openmp.org/wp/

[openmp-gfortran] http://gcc.gnu.org/onlinedocs/gfortran/OpenMP.html

[openmp-gfortran2] http://sites.google.com/site/gfortransite/

[openmp-api3.0] http://www.openmp.org/mp-documents/spec30.pdf

[openmp-refcard] OpenMP in Fortran Reference card

[openmp-fort90-examples] http://people.sc.fsu.edu/~burkardt/f_src/open_mp/open_mp.html

[exascale-doe] Modeling and Simulation at the Exascale for Energy and the Environment, DOE Town Hall Meetings Report

[exascale-sc08] http://www.lbl.gov/CS/html/SC08ExascalePowerWorkshop/index.html