UW AMath 483/583 Class Notes

Release 1.0

Randall J. LeVeque

May 22, 2016

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

Skip to... technical_topics ... applications

toc_condensed

CHAPTER

ONE

2013 VERSIONS OF SOME FILES

These pages may be referred to in Lecture Videos filmed in 2013, particularly the homeworks and project.

1.1 2013 versions of some files

These pages may be referred to in Lecture Videos filmed in 2013, particularly the homeworks and project.

1.1.1 2013 Homework

Warning: These are the 2013 homework assignment. See *Homework* for this year's assignments.

There will be 6 homeworks during the quarter with tentative due dates listed below:

- 2013_homework1: Wednesday of Week 2, April 10
- 2013_homework2: Wednesday of Week 3, April 17
- 2013_homework3: Wednesday of Week 5, May 1
- 2013_homework4: Wednesday of Week 6, May 8
- 2013_homework5: Wednesday of Week 8, May 22
- 2013_homework6: Friday of Week 9, May 29
- 2013_project: Wednesday of Week 11, June 12

1.1.2 Virtual Machine for this class [2013 version]

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. The TA's will also have the VM on memory sticks for transferring.

System requirements

The VM is around 2.1 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.

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. 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.

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 `uwhpsc`. (We would have liked to set up a VM with no password, but many things in Linux assume you have one.)

Note that you will also need this password to quit the VM.

Running programs

You can access the programs on the virtual machine through the Applications Menu (the mouse on an *X* symbol in the upper-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.

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.

Also, if you move your computer from one wireless network to another while the VM is running, it may lose connection with the internet.

If this happens, it should be sufficient to shut down the VM (with the 0/1 button on the top right corner) and then restart it. On shutdown, a script is automatically run that does the following, which in earlier iterations of the VM had to be done manually...

    $ 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.

Shutting down

When you are done using the virtual machine, you can shut it down by clicking the 0/1 button on the top-right corner of the VM. You will need the password *uwhpsc*.

Cutting and pasting

If you want to cut text from one window in the VM and paste it into another, you should be able to highlight the text and then type ctrl-c (or in a terminal window, ctrl-shift-C, since ctrl-c is the interrupt signal). To paste, type ctrl-v (or ctrl-shift-V in a terminal window).

If you want to be able to cut and paste between a window in the VM and a window on your host machine, click on Machine from the main VitualBox menu (or *Settings* in the Oracle VM VirtualBox Manager window), then click on *General* and then *Advanced*. Select *Bidirectional* from the *Shared Clipboard* menu.

Shared Folders

If you create a file on the VM that you want to move to the file system of the host machine, or vice versa, you can create a "shared folder" that is seen by both.

First create a folder (i.e. directory) on the host machine, e.g. via:

```
$ mkdir ~/uwhpsc_shared
```

This creates a new subdirectory in your home directory on the host machine.

In the VirtualBox menu click on *Devices*, then click on *Shared Folders*. Click the + button on the right side and then type in the full path to the folder you want to share under *Folder Path*, including the folder name, and then the folder name itself under *Folder name*. If you click on *Auto-mount* then this will be mounted every time you start the VM.

Then click *OK* twice.

Then, in the VM (at the linux prompt), type the following commands:

```
sharename=uwhpsc_shared   # or whatever name the folder has
sudo mkdir /mnt/$sharename
sudo chmod 777 /mnt/$sharename
sudo mount -t vboxsf -o uid=1000,gid=1000 $sharename /mnt/$sharename
```

You may need the password *uwhpsc* for the first *sudo* command.

The folder should now be found in the VM in */mnt/$sharename*. (Note *$sharename* is a variable set in the first command above.)

If auto-mounting doesn't work properly, you may need to repeat the final *sudo mount ...* command each time you start the VM.

Enabling more processors

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, open the VirtualBox *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.

Problems enabling multiple processors...

Users may encounter several problems with enabling mutliple processors. Some users may not be able to change this setting (it will be greyed out). Other users when may find no improved performance after enabling multiple processors. Still others may encounter an error such as:

```
VD: error VERR_NOT_SUPPORTED
```

All of these problems indicate that virtualization has not been enabled on your processors.

Fortunately this has an easy fix. You just have to enable virtualization in your BIOS settings.

1. To access the BIOS settings you must restart your computer and press a certain button on startup. This button will depend on the company that manufactures your computer (for example for Lenovo's it appears to be the f1 key).

2. Next you must locate a setting that will refer to either virtualization, VT, or VT-x. Again the exact specifications will depend on the computer's manufacturer, however it should be found in the Security section (or the Performance section if you are using a Dell).

3. Enable this setting, then save and exit the bios settings.After your computer reboots you should be able to start the VM using multiple processors now.

4. If your BIOS does not have any settings like this it is possible that your BIOS is set up to hide this option from you, and you may need to follow the advice here: http://mathy.vanvoorden.be/blog/2010/01/enable-vt-x-on-dell-laptop/

Note: Unfortunately some older hardware does not support virtualization, and so if these solutions don't work for you it may be that this is the case for your processors. There also may be other possible problems...so don't be afraid to ask the TAs for help!

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 Applications menu in the upper-left corner, pointing to *Settings*, then clicking *Display*. Choose a resolution from the drop-down list, then click *Apply*.

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.

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.

How This Virtual Machine was made

1. Download Ubuntu 12.04 PC (Intel x86) alternate install ISO from http://cdimage.ubuntu.com/xubuntu/releases/12.04.2/release/xubuntu-12.04.2-alternate-i386.iso
2. Create a new virtual box
3. Set the system as Ubuntu
4. Use defualt options
5. After that double click on your new virtual machine...a dropdown box should appear where you can select your ubuntu iso
6. As you are installing...at the first menu hit F4 and install a command line system

7. Let the install proceed following the instructions as given. On most options the default answer will be appropriate. When it comes time to format the hard drive, choose the manual option. Format all the free space and set it as the mount point. From the next list choose root (you dont need a swap space).

8. Install the necessary packages

```
# $UWHPSC/notes/install.sh
#
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install xfce4
sudo apt-get install jockey-gtk
sudo apt-get install xdm
sudo apt-get install ipython
sudo apt-get install python-numpy
sudo apt-get install python-scipy
sudo apt-get install python-matplotlib
sudo apt-get install python-dev
sudo apt-get install git
sudo apt-get install python-sphinx
sudo apt-get install gfortran
sudo apt-get install openmpi-bin
sudo apt-get install liblapack-dev
sudo apt-get install thunar
sudo apt-get install xfce4-terminal

# some packages not installed on the VM
# that you might want to add:

sudo apt-get install gitk               # to view git history
sudo apt-get install xxdiff             # to compare two files
sudo apt-get install python-sympy       # symbolic python
sudo apt-get install imagemagick        # so you can "display plot.png"


sudo apt-get install python-setuptools  # so easy_install is available
sudo easy_install nose                  # unit testing framework
sudo easy_install StarCluster           # to help manage clusters on AWS
```

9. To setup the login screen edit the file Xresources so that the greeting line says.:

```
xlogin*greeting: Login and Password are uwhpsc
```

10. Create the file uwhpscvm-shutdown.:

11. Save it at.:

```
/usr/local/bin/uwhpscvm-shutdown
```

12. Execute the following command command.:

```
$ sudo chmod +x /usr/local/bin/uwhpscvm-shutdown
```

13. Right click on the upper panel and select add new items and choose to add a new launcher.

14. Name the new launcher something like shutdown and in the command blank copy the following line.:

```
gksudo /usr/local/bin/uwhpscvm-shutdown
```

15. Go to preferred applications and select Thunar for file managment and the xfce4 terminal.
16. Run jockey-gtk and install guest-additions.
17. Go to Applications then Settings then screensaver and select "disable screen saver" mode
18. In the settings menu select the general settings and hit the advanced tab. Here you can set the clipboard and drag and drop to allow Host To Guest.
19. Shutdown the machine and then go to the main virtualbox screen. Click on the virtualmachine and then hit the settings button.
20. After, in the system settings click on the processor tab. This may let you allow the virtual machine to use more than one processor (depending on your computer). Choose a setting somewhere in the green section of the Processors slider.

About the VM

The class virtual machine is running XUbuntu 12.04, 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.

Further reading

[VirtualBox] [VirtualBox-documentation]

1.1.3 Amazon Web Services EC2 AMI [2013 version]

Warning: This is the 2013 version. See 2014_aws for updated instructions.

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*. You can also use the *Virtual Machine for this class [2014 Edition]*.

Another alternative is to write and run your programs "in the cloud" using Amazon Web Services (AWS) Elastic Cloud Computing (EC2). You can start up an "instance" (your own private computer, or so it appears) that is configured using an Amazon Machine Image (AMI) that has been configured with the Linux operating system and containing all the software needed for this class.

You must first sign up for an account on the AWS main page. For this you will need a credit card, but note that with an account you can get 750 hours per month of free "micro instance" usage in the free usage tier. A micro instance is a single processor (that you will probably be sharing with others) so it's not suitable for trying out parallel computing, but should be just fine for much of the programming work in this class.

You can start up more powerful instances with 2 or more processors for a cost starting at about 14.5 cents per hour (the High CPU Medium on-demand instance). See the pricing guide.

For general information and guides to getting started:

- Getting started with EC2, with tutorial to lead you through an example.
- EC2 FAQ.

- Pricing. Note: you are charged per hour for hours (or fraction thereof) that your instance is in *running* mode, regardless of whether the CPU is being used.
- High Performance Computing on AWS with instructions on starting a cluster instance.
- UW eScience information on AWS.

Launching an instance with the *uwhpsc* AMI

Quick way

Navigate your browser to https://console.aws.amazon.com/ec2/home?region=us-west-2#launchAmi=ami-b47feb84

Then you can skip the next section and proceed to aws_select_size.

Search for AMI

Going through this part may be useful if you want to see how to search for other AMI's in the future.

Once you have an AWS account, sign in to the management console and click on the EC2 tab, and then select Region US West (Oregon) from the menu at the top right of the page, next to your user name.

You should now be on the page https://console.aws.amazon.com/ec2/v2/home?region=us-west-2.

Click on the big "Launch Instance" button.

Select the "Classic Wizard" and "Continue".

On the next page you will see a list of Amazon Machine Images (AMIs) that you can select from if you want to start with a fresh VM. For this class you don't want any of these. Instead click on the "Community AMIs" tab and wait a while for a list to load. Make sure Viewing "All images" is selected from the drop-down menu.

After the list of AMIs loads, type *uwhpsc* in the search bar. Select this image.

Select size and security group

On the next page you can select what sort of instance you wish to start (larger instances cost more per hour). T1-micro is the the size you can run free (as long as you only have one running).

Click *Continue* on the next few screens through the "instance details" and eventually you get to one that asks for a key pair. If you don't already have one, create a new one and select it here.

Click *Continue* and you will get a screen to set Security Groups. Select the *quicklaunch-1* option. On the next screen click *Launch*.

Logging on to your instance

Click *Close* on the page that appears to go back to the Management Console. Click on *Instances* on the left menu and you should see a list of instance you have created, in your case only one. If the status is not yet *running* then wait until it is (click on the *Refresh* button if necessary).

Click on the instance and information about it should appear at the bottom of the screen. Scroll down until you find the *Public DNS* information

Go into the directory where your key pair is stored, in a file with a name like *rjlkey.pem* and you should be able to *ssh* into your instance using the name of the public DNS, with format like:

```
$ ssh -X -i KEYPAIR-FILE  ubuntu@DNS
```

where KEYPAIR-FILE and DNS must be replaced by the appropriate things, e.g. for the above example:

```
$ ssh -X -i rjlkey.pem ubuntu@ec2-50-19-75-229.compute-1.amazonaws.com
```

Note:

- You must include *-i keypair-file*
- You must log in as user ubuntu.
- Including -X in the ssh command allows X window forwarding, so that if you give a command that opens a new window (e.g. plotting in Python) it will appear on your local screen.
- See the section *Using ssh to connect to remote computers* for tips if you are using a Mac or Windows machine. If you use Windows, see also the Amazon notes on using *putty* found at http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/putty.html.

Once you have logged into your instance, you are on Ubuntu Linux that has software needed for this class pre-installed.

Other software is easily installed using *apt-get install*, as described in *Downloading and installing software for this class*.

Transferring files to/from your instance

You can use *scp* to transfer files between a running instance and the computer on which the ssh key is stored.

From your computer (not from the instance):

```
$ scp -i KEYPAIR-FILE FILE-TO-SEND ubuntu@DNS:REMOTE-DIRECTORY
```

where DNS is the public DNS of the instance and *REMOTE-DIRECTORY* is the path (relative to home directory) where you want the file to end up. You can leave off *:REMOTE-DIRECTORY* if you want it to end up in your home directory.

Going the other way, you can download a file from your instance to your own computer via:

```
$ scp -i KEYPAIR-FILE ubuntu@DNS:FILE-TO-GET .
```

to retrieve the file named *FILE-TO-GET* (which might include a path relative to the home directory) into the current directory.

Stopping your instance

Once you are done computing for the day, you will probably want to stop your instance so you won't be charged while it's sitting idle. You can do this by selecting the instance from the Management Console / Instances, and then select *Stop* from the *Instance Actions* menu.

You can restart it later and it will be in the same state you left it in. But note that it will probably have a new Public DNS!

Creating your own AMI

If you add additional software and want to save a disk image of your improved virtual machine (e.g. in order to launch additional images in the future to run multiple jobs at once), simply click on *Create Image (EBS AMI)* from the *Instance Actions* menu.

Viewing webpages directly from your instance

An apache webserver should already be running in your instance, but to allow people (including yourself) to view webpages you will need to adjust the security settings. Go back to the Management Console and click on *Security Groups* on the left menu. Select *quick-start-1* and then click on *Inbound*. You should see a list of ports that only lists 22 (SSH). You want to add port 80 (HTTP). Select HTTP from the drop-down menu that says *Custom TCP Rule* and type 80 for the *Port range*. Then click *Add Rule* and *Apply Rule Changes*.

Now you should be able to point your browser to *http://DNS* where *DNS* is replaced by the Public DNS name of your instance, the same as used for the *ssh* command. So for the example above, this would be

```
http://ec2-50-19-75-229.compute-1.amazonaws.com
```

The page being displayed can be found in */var/www/index.html* on your instance. Any files you want to be visible on the web should be in */var/www*, or it is sufficient to have a link from this directory to where they are located (created with the *ln -s* command in linux).

So, for example, if you do the following:

```
$ cd $HOME
$ mkdir public      # create a directory for posting things
$ chmod 755 public  # make it readable by others
$ sudo ln -s $HOME/public /var/www/public
```

then you can see the contents of your $HOME/public directory at:

```
http://ec2-50-19-75-229.compute-1.amazonaws.com/public
```

Remember to change the DNS above to the right thing for your own instance!

1.1.4 Git [2013 version]

Warning: This page is from 2013. See *Git* for this year's version.

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

Instructions for cloning the class repository

All of the materials for this class, including homeworks, sample programs, and the webpages you are now reading (or at least the *.rst* files used to create them, see *Sphinx documentation*), are in a Git repository hosted at Bitbucket, located at http://bitbucket.org/rjleveque/uwhpsc/. 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
$ git clone https://rjleveque@bitbucket.org/rjleveque/uwhpsc.git
```

Note the following:

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

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

```
$ export UWHPSC=$HOME/uwhpsc
```

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 uwhpsc
$ export UWHPSC=`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 UWHPSC
```

to make sure *UWHPSC* 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 UWHPSC=$HOME/uwhpsc
```

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 UWHPSC=full-path-to-uwhpsc
```

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

Updating your clone

The files in the class repository will change as the quarter progresses — new notes, 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 $UWHPSC
$ git fetch origin
$ git merge origin/master
```

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

The *git fetch* command instructs *git* to fetch any changes from *origin*, which points to the remote bitbucket repository that you originally cloned from. In the merge command, *origin/master* refers to the master branch in this repository (which is the only branch that exists for this particular repository). This merges any changes retrieved into the files in your current working directory.

The last two command can be combined as:

```
$ git pull origin master
```

or simply:

```
$ git pull
```

since *origin* and *master* are the defaults.

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 *git* for your own work without creating a repository on a hosted site such as Bitbucket (see newgit 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 a free private repository on Bitbucket. 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 homework1. 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:

```
$ git config --global user.name "Your Name"
$ git config --global user.email you@example.com
```

   These will be used when you commit changes. If you don't do this, you might get a warning message the first time you try to commit.

2. Go to http://bitbucket.org/ and click on "Sign up now" if you don't already have an account.
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" next to "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 *uwhpsc*, owned by user *rjleveque*. To avoid confusion, you should probably not name your repository *uwhpsc*.
6. Make sure you click on "Private" at the bottom. Also turn "Issue tracking" and "Wiki" on if you wish to use these features.
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 *MYHPSC* from inside this directory by:

```
$ export MYHPSC=`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 *MYHPSC* similar to the line added for *UWHPSC*.

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

```
$ ls
```

    But try:

```
$ ls -a
./  ../  .git/
```

    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 *.git* 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 *git*.

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:

```
$ git status -s
```

    The response should be:

```
?? testfile.txt
```

    The ?? means that this file is not under revision control. The *-s* flag results in this *short* status list. Leave it off for more information.

    To put the file under revision control, type:

```
$ git add testfile.txt
$ git status -s
A  testfile.txt
```

    The A means it has been added. However, at this point *git* is not we have not yet taken a *snapshot* of this version of the file. To do so, type:

```
$ git commit -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.

    You should get a response like:

```
[master (root-commit) 28a4da5] My first commit of a test file.
 1 file changed, 2 insertions(+)
 create mode 100644 testfile.txt
```

    We can now see the status of our directory via:

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

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

```
$ git status testfile.txt
```

You can get a list of all the commits you have made (only one so far) using:

```
$ git log

commit 28a4da5a0deb04b32a0f2fd08f78e43d6bd9e9dd
Author: Randy LeVeque <rjl@ned>
Date:   Tue Mar 5 17:44:22 2013 -0800

    My first commit of a test file.
```

The number 28a4da5a0deb04b32a0f2fd08f78e43d6bd9e9dd above is the "name" of this commit and you can always get back to the state of your files as of this commit by using this number. You don't have to remember it, you can use commands like *git log* to find it later.

Yes, this is a number... it is a 40 digit hexadecimal number, meaning it is in base 16 so in addition to 0, 1, 2, ..., 9, there are 6 more digits a, b, c, d, e, f representing 10 through 15. This number is almost certainly guaranteed to be unique among all commits you will ever do (or anyone has ever done, for that matter). It is computed based on the state of all the files in this snapshot as a SHA-1 Cryptographic hash function, called a SHA-1 Hash for short.

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:

```
$ git status -s
 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:

```
$ git diff testfile.txt
```

This will produce something like:

```
diff --git a/testfile.txt b/testfile.txt
index d80ef00..fe42584 100644
--- a/testfile.txt
+++ b/testfile.txt
@@ -1,2 +1,3 @@
 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, *git diff* would only print a few lines around any change to indicate the context.

Now let's try to commit this changed file:

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

This will fail! You should get a response like this:

```
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
#   directory)
#
#   modified:   testfile.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

git is saying that the file *testfile.txt* is modified but that no files have been **staged** for this commit.

If you are used to Mercurial, *git* has an extra level of complexity (but also flexibility): you can choose which modified files will be included in the next commit. Since we only have one file, there will not be a commit unless we add this to the **index** of files staged for the next commit:

```
$ git add testfile.txt
```

Note that the status is now:

```
$ git status -s
M  testfile.txt
```

This is different in a subtle way from what we saw before: The *M* is in the first column rather than the second, meaning it has been both modified and staged.

We can get more information if we leave off the *-s* flag:

```
$ git status

# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   testfile.txt
#
```

Now *testfile.txt* is on the index of files staged for the next commit.

Now we can do the commit:

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

[master 51918d7] added a third line to the test file
 1 file changed, 1 insertion(+)
```

Try doing *git log* now and you should see something like:

```
commit 51918d7ea4a63da6ab42b3c03f661cbc1a560815
Author: Randy LeVeque <rjl@ned>
Date:   Tue Mar 5 18:11:34 2013 -0800

    added a third line to the test file

commit 28a4da5a0deb04b32a0f2fd08f78e43d6bd9e9dd
Author: Randy LeVeque <rjl@ned>
Date:   Tue Mar 5 17:44:22 2013 -0800

    My first commit of a test file.
```

If you want to revert your working directory back to the first snapshot you could do:

```
$ git checkout 28a4da5a0de
Note: checking out '28a4da5a0de'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

HEAD is now at 28a4da5... My first commit of a test file.
```

Take a look at the file, it should be back to the state with only two lines.

Note that you don't need the full SHA-1 hash code, the first few digits are enough to uniquely identify it.

You can go back to the most recent version with:

```
$ git checkout master
Switched to branch 'master'
```

We won't discuss branches, but unless you create a new branch, the default name for your main branch is *master* and this *checkout* command just goes back to the most recent commit.

14. So far you have been using *git* 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:

    ```
    $ git status
    ```

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

    Now do:

    ```
    $ git push -u origin master
    ```

    This will prompt for your Bitbucket password and should then print something indicating that it has uploaded these two commits to your bitbucket repository.

    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.

    To push future commits to bitbucket, you should only need to do:

    ```
    $ git push
    ```

    and by default it will push your master branch (the only branch you have, probably) to *origin*, which is the shorthand name for the place you originally cloned the repository from. To see where this actually points to:

    ```
    $ git remote -v
    ```

    This lists all *remotes*. By default there is only one, the place you cloned the repository from. (Or none if you had created a new repository using *git init* rather than cloning an existing one.)

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 "Commits" 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.

If you click on the hex-string for a commit, 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!

homework1 instructs you to add some additional files to the Bitbucket repository.

Feel free to experiment further with your repository at this point.

Further reading

Next see *Bitbucket repositories: viewing changesets, issue tracking* and/or *More about Git*.

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

```
$ git help
$ git help diff  # or any other specific command name
```

Each command has lots of options!

Git references contains references to other sources of information and tutorials.

CHAPTER

TWO

COURSE MATERIALS – 2014 EDITION

2.1 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, 2014.

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 the *Bibliography and further reading* for more references and other suggested readings.

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 documentation*) and the input files are available using git (see *Instructions for cloning the class repository*).

2.1.1 License

2.2 Class Format

This class is being taught differently in 2014 than in past years.

- Students will be required to watch 3 hours of lectures a week that will be available on the web (videos recorded in Spring, 2013). These follow the slides and lecture notes from last year, but some new material will also be developed.

  Enrolled students can find the videos on the Canvas Syllabus Page or on your Canvas Calendar on the corresponding date. Click on the Lecture number link to find the videos, slides, and the associated quiz.

- There is a short quiz associated with each lecture to encourage you to watch the lectures. All the quizzes can also be found on the Canvas Quizzes Page.
- Each quiz is worth 5 points (with the lowest 3 scores dropped, out of the 28 Lecture Quizzes). The quiz for all three lectures of each calendar week must be completed by 11:00pm PDT on the following Monday.
- Slides to accompany all the lectures, along with a summary of the topics covered in each lecture, can be found at **'http://faculty.washington.edu/rjl/classes/am583s2013/slides/'_**
- For on-campus students, the class meets for a lab session T-Th, 2:30 - 3:20 in OUG 136. This is an Active Learning Classroom and class time will be used primarily for working together with other students on exercises designed to illustrate and reinforce the material. Some additional new material will also be presented. For on-campus students, attendance will be required at these sessions, and each lab will contain some work that is turned in for grading. (With the lowest two scores dropped in case you miss a lab or two.)
- Undergraduates should enroll in AMath 483 and graduate students in 583. The lectures and classroom sessions are identical, but the 583 course will have some additional and/or more advanced assignments.
- Section 583B is for online Masters degree students only and is not available to on-campus students. Students in this section will view the same video lectures as on-campus students. In addition, parts of the T-Th lab sessions will be taped and available to watch, along with the exercises tackled by students in these sessions.
- Homework and a final project will consist primarily of programming assignments. There will be 4 homeworks worth 100 points each and the project will be worth 200 points.

2.3 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.

2.3.1 Some topics

Many topics will be covered in a non-linear fashion, jumping around between topics to tie things together.

- Using the Virtual Machine
- Unix / Linux
- Version control systems
- Using Git and Bitbucket.
- Basic Python
- Ipython and the IPython notebook
- NumPy and Scipy
- Debugging Python
- Compiled vs. interpreted languages
- Introduction to Fortran 90

- Makefiles
- Computer architecture: CPU, memory access, cache hierachy, pipelining
- Optimizing Fortran
- BLAS and LAPACK routines
- Parallel computing
- OpenMP with Fortran
- MPI with Fortran
- Parallel Python
- Graphics and visualization
- I/O, Binary output
- Mixed language programming

2.4 Notes to accompany lab sessions

2.4.1 Lab 1: Tuesday April 1, 2014

Brief overview of the class structure

See:

- *Class Format*
- *Computing Options [2014 version]*
- Canvas page
- honor code

Things to try in groups:

Please work together in groups of 2 or 3.

- Log on to SageMathCloud. See smc.
- Create a new project and share it with your group members.
- Open a terminal window. If people in your group aren't comfortable with Unix, try out commands like *ls*, *pwd*, *mkdir*, *mv*, *cp*, etc. See *Unix, Linux, and OS X*.
- Create and edit a new file named *hello.py* by typing:

```
$ open hello.py
```

  at the terminal prompt *$*.

- Add one line to this file and then click Save:

```
print "Hello World!"
```

- In the terminal window, type:

```
$ python hello.py
```

In should print out Hello World!

- Edit the file to add the lines:

```
import matplotlib          # python plotting package
matplotlib.use("Agg")      # so plot commands work in this script
from pylab import *        # imports lots of things like linspace, sin, plot

x = linspace(0,1,1001)
y = sin(10 * pi * x**2)
plot(x,y)

fname = "myplot.png"
savefig(fname)
print "Saved ",fname
```

- Save the file and rerun it at the terminal prompt, then view the file:

```
$ python hello.py
$ open myplot.png
```

- Open an IPython Notebook from the New menu.
- Follow the instructions for "making a plot in IPython" from the page smc.

2.4.2 Lab 8: Thursday April 24, 2014

The code for this lab can be found in *$UWHPSC/labs/lab8*.

Lab started with a demo of using compiler flags and *gdb* to debug Fortran code, using the code in *$UWHPSC/labs/lab8/demo*.

Running it without any compiler flags gives no error by there is a *NaN* value in the results:

```
$ cd $UWHPSC/labs/lab8
$ make run

 The max value of y is   0.99973335466585400
 x(501) is    0.0000000000000000
 y(501) is                      NaN
```

Running it with compiler flags to catch floating point exceptions:

```
$ make debug
make: *** [debug] Floating point exception: 8
```

Once it's compiled with the flags specified in the Makefile for the *debug* target, the debugger *gdb* can be used to run the code and figure out where it died:

```
$ gdb ./a.out
(gdb) run

Program received signal SIGFPE, Arithmetic exception.
0x0000000000400a6d in demo () at demo.f90:12
12          y(j) = sin(x(j)) / x(j)

(gdb) p j
```

```
$1 = 501

(gdb) p x(j)
$2 = 0
```

Many commands are available in *gdb*, see for example this documentation.

Note: On the Mac with the Mavericks OS, *gdb* has been replaced by *lldb*. See http://lldb.llvm.org/index.html for more information.

The page GDB TO LLDB COMMAND MAP gives a good summary of both *gdb* and *lldb* commands and the relation between them.

Debugging compile-time errors

The code in *$UWHPSC/labs/lab8/problem1* does not compile. See if you can find and fix all the errors. See *$UWHPSC/labs/lab8/problem1b* for a corrected version (and see the *README.txt* file in that directory for some comments).

Debugging run-time errors

The code *$UWHPSC/labs/lab8/problem2/array1.f90* does not run properly. See if you can find and fix all the errors. See *$UWHPSC/labs/lab8/problem2/array1b.f90* for a corrected version and use *diff* to see the differences.

2.4.3 Lab 9: Tuesday April 29, 2014

Programming problem

Work on this in groups!

Write a Fortran program to compute the roots of a quadratic equation and determine the absolute and relative error in the roots computed. The program should do the following:

- Prompt the user and then read in the desired roots *x1true* and *x2true* (See the example below).
- Determine the coefficients *a, b, c* so that the quadratic equation $ax^2 + bx + c = 0$ has the desired roots (you can set *a=1*).
- Using *a,b,c*, compute the roots *x1* and *x2* by using the quadratic formula.
- Print out the "true" and computed values for each root along with the absolute and relative error in each.

So you should be able to do something like this:

```
$ gfortran quadratic.f90
$ ./a.out
 input x1true, x2true:
2.5, 3

Coefficients:  a =     0.100000E+01  b =    -0.550000E+01  c = 0.750000E+01

Root x1  computed:  0.250000000000000E+01  true:  0.250000000000000E+01
         absolute error:     0.000000E+00  relative error:     0.000000E+00

Root x2  computed:  0.300000000000000E+01  true:  0.300000000000000E+01
         absolute error:     0.000000E+00  relative error:     0.000000E+00
```

Don't worry too much about the formatting but you might want to print out 15 digits in the computed roots.

You might want to assume the values are entered with *x1true* <= *x2true* so you know which root from the quadratic equations goes with which original value. (And print out an informative error message otherwise.)

Test it out

Test a variety of values to see that it's working.

Once it's working on reasonable values, try the following:

- *x1 = 1e-12, x2 = 2*
- *x1 = -2, x2 = 1e-12*

In each case you should find that one root is computed accurately but the other root has a large relative error (few digits of accuracy).

Figure out why "catastrophic cancellation" is the problem.

Improvements

- Improve the code by noticing that if one root is calculated accurately, the other root can be calculated from the fact that *x1 * x2 = c*.
- Remove the assumption that *x1true* <= *x2true*.

2.4.4 Lab 10: Tuesday May 1, 2014

Programming problem

Work on this in groups!

1. The OpenMP code *$UWHPSC/labs/lab10/array_omp.f90* contains some bugs. Find the bugs and fix them so that it runs and gives output like this:

```
$ gfortran -fopenmp array_omp.f90
$ ./a.out
 nthreads =            6
 b and bt should be equal
 b=
    0.270000D+02
    0.330000D+02
    0.390000D+02
    0.450000D+02
    0.510000D+02
 bt=
    0.270000D+02
    0.330000D+02
    0.390000D+02
    0.450000D+02
    0.510000D+02
```

2. If A is an $n \times n$ matrix and x is a vector of length n, then $x^T Ax$ is a scalar, a "quadratic form" since it is the sum of terms of the form $a_{ij}x_i x_j$ that are quadratic in the elements of x .

   Write an OpenMP code to compute this for a given matrix and vector. Write out the matrix-vector multiplies as loops and use "omp parallel do" loops to compute first the vector Ax and then the inner product of this with the

vector x. Test your code using the 10×10 identity matrix for A and $x_i = i$, in which case the correct answer can be determined to be 385 from the formula

$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$.

3. There is a quiz for this lab.

2.4.5 Lab 11: Tuesday May 6, 2014

- Note that there are several example codes in the class repository that might be useful, e.g.
  - *$UWHPSC/codes/openmp*
  - *$UWHPSC/codes/lapack/random*
  - *$UWHPSC/2013/homeworks*
  - *$UWHPSC/2013/solutions*
- Discussion of random number generators. See *UWHPSC/labs/lab11*
- Questions about OpenMP?

Programming problem

Work on this in groups!

1. Write a Fortran program to do the following:
   - input *seed* and *n* from the command line
   - seed the random number generator using *init_random_seed* from the *random_util.90* module,
   - generate an array *x* of *n* random numbers
   - compute the mean of these values: the sum of all elements of *x* divided by *n*. Do this with a *do* loop.

   Since *random_number* produces numbers that should be uniformly distributed between 0 and 1, the mean should be approximately 0.5 for large *n*. It can also be shown that for a uniform distribution, the difference between the mean of a sample of *n* numbers and the true mean of the distribution should decay to zero like $1/\sqrt{n}$ as $n \longrightarrow \infty$. Do you observe this?
2. Modify your code to use OpenMP by using an *omp parallel do* loop with a suitable reduction to compute the mean.
3. There is a quiz for this lab.

2.4.6 Lab 12: Thursday May 8, 2014

Programming problem

Work on this in groups!

1. In Lab 11 you worked on a program to compute the mean of n random numbers. A sample solution can be found at *$UWHPSC/labs/lab11/mean.f90*.

   Write a Fortran program that runs over different values of *n*, and for each *n* generates a vector *x* containing *n* random numbers and then computes the mean of these. Also compute the fraction of the numbers that lie in the first quartile (the fraction of *x(i)* values that are between 0 and 0.25) and the fraction that lie in the fourth quartile

(between 0.75 and 1.0). Since the *random_number* routine returns numbers uniformly distributed between 0 and 1, we expect each of these fractions to be about 0.25.

Use OpenMP to make the loop on *i* from 1 to *n* into a parallel do loop.

Running this code should give something like this if you take as the *n* values $n = 10^k$ for $k = 2, 3, \ldots, 8$:

```
 Number of threads:           2
 input seed
12345
 seed1 for random number generator:       12345

       n          mean        quartile 1    quartile 4
       100     0.51902466     0.22000000     0.24000000
      1000     0.47476778     0.27800000     0.22500000
     10000     0.49606601     0.25670000     0.25190000
    100000     0.50121669     0.24815000     0.25130000
   1000000     0.50001034     0.24986300     0.24979800
  10000000     0.49998532     0.24994350     0.24992770
 100000000     0.49995944     0.25003764     0.24995608
```

2. If you haven't already, study the code in *$UWHPSC/codes/openmp/pisum2.f90* and make sure you understand how this coarse grain parallelism works. Discuss with your neighbors.
3. If you have time, try to follow this model to make your code that computes the mean and quartiles work in a similar manner, where you break up the different values of *n* to be tested between different threads, e.g. in the above example one thread would take the first three values of *n* and the second thread would take the final two values of *n*.
4. Discuss with your neighbors whether this is a sensible way to try to use two threads on this problem.
5. There is a quiz for this lab.
6. Sample solutions can now be found in *$UWHPSC/labs/lab12*.

2.4.7 Lab 13: Tuesday May 13, 2014

Announcements

- Homework 4 will be posted soon and due on Tuesday, May 27 at 11pm PDT.
- IPython 2.0 was released in April and has some cool new features. It is **not** available yet on SageMathCloud. See *IPython_notebook* for some references.

Demos

- IPython 2.0 with interactive widgets:

  The notebook in *$UWHPSC/labs/lab13/widgets_demo.ipynb* illustrates this. (Does not run on SMC.)

- Plots you can zoom in on in IPython notebooks: mpld3. The notebook *$UWHPSC/labs/lab13/mpld3_demo.ipynb* can be run on SMC.

Gamblers' Ruin problem

- The notebook *$UWHPSC/labs/lab13/GamblersRuin.ipynb* illustrates in Python a problem that you will tackle in Homework 4 using Fortran with OpenMP and MPI.

There is a quiz for Lab 13

2.4.8 Lab 14: Thursday May 15, 2014

Demos

- Python dictionaries: *$UWHPSC/labs/lab14/dictionary_demo.ipynb* illustrates this.
- Towers of Hanoi: *$UWHPSC/labs/lab14/Towers_of_Hanoi.ipynb* illustrates this.
- One solution to the lab problem: *$UWHPSC/labs/lab14/Towers_of_Hanoi_plots.ipynb* You can view it at http://nbviewer.ipython.org/url/faculty.washington.edu/rjl/notebooks/Towers_of_Hanoi_plots.ipynb

There is a quiz for Lab 14

2.4.9 Lab 15: Tuesday May 20, 2014

Install JSAnimation: See *Animation in Python*.

Demos

- *$UWHPSC/labs/lab15/JSAnimation_demo.ipynb*

  View at http://nbviewer.ipython.org/url/faculty.washington.edu/rjl/notebooks/JSAnimation_demo.ipynb
- *$UWHPSC/labs/lab15/WavePacket.ipynb*

  View at http://nbviewer.ipython.org/url/faculty.washington.edu/rjl/notebooks/WavePacket.ipynb
- *$UWHPSC/labs/lab15/WavePacket.py* Script version
- *$UWHPSC/labs/lab15/Towers_of_Hanoi_animation.ipynb*

  View at http://nbviewer.ipython.org/url/faculty.washington.edu/rjl/notebooks/Towers_of_Hanoi_animation.ipynb

Note: These use *$UWHPSC/labs/lab15/JSAnimation_frametools.py*.

Problem to solve

Create an animation similar to http://faculty.washington.edu/rjl/classes/am583s2014/Square.html.

Hints:

- The following matplotlib commands may be useful:

```
fill(x,y,'b')    # fill polygon specified by arrays x and y with blue
axis('scaled')    # scale x and y axes the same way
```

- Recall that to rotate a point (x,y) through angle $theta$ you can can compute

  $\hat{x} = \cos(\theta)x + \sin(\theta)y$

  $\hat{y} = -\sin(\theta)x + \cos(\theta)y$

There is no quiz for Lab 15

2.4.10 Lab 16: Thursday May 22, 2014

Problem to solve

- Adapt the program *$UWHPSC/codes/lapack/randomsys3.f90* to use a specific matrix A in place of the random matrix used in the original code. The matrix to use is the Hilbert matrix defined by

  $$a_{i,j} = \frac{1}{i+j-1}$$

  This is a notorious matrix since it is always nonsingular but is very ill-conditioned even for moderately small values of *n*.

  For more discussion of this matrix, and a formula for how the condition number grows with *n*, see this Cleve's Corner blog post.

- Note that in order to create an executable for your program, in the linking step you will need to make sure *gfortran* also links in the BLAS and LAPACK library. See the *LFLAGS* set in '$UWHPSC/codes/lapack/Makefile for the arguments you need to add to the linking step.

- Instead of using the *random_number* subroutine to generate a random *x* for checking the relative error, as is done in *$UWHPSC/codes/lapack/randomsys3.f90*, try taking *x* to be a vector of all 1's. (And as in the original code compute $b = Ax$ usint *matmul* and then solve the system to recover *x*.) Print out the computed *x* as well as computing the relative error in the 1-norm as in the original code. How well does it do? How does the accuracy relate to the condition number?

- You might want to look at the dgecon documentation.

- Try different values of *n* with your program to see if it gives the expected behavior. Note that the LAPACK function *dgecon* does not compute the exact condition number but only estimates it. Also note that the program estimates the 1-norm condition number, while the approximate formula is for the 2-norm condition number (but they grow in a similar exponential fashion).

If you have time to do more...

- Modify your code by creating a Fortran function *hilbert_condition* that returns the condition number estimate for a given value of *n*.

  Then write a main program that loops over *n* from 1 to 20, computes the estimate for each *n*, and writes a text file with two columns *n* and the estimate. These statements might be useful:

```
      open(21, file='cond.txt',status='unknown')

      do n=1,20
          cond = hilbert_condition(n)
          ! print *, "cond = ",cond
          write(21, 210) n,cond
210       format(i4,e16.6)
          enddo
```

- The text file produced should be readable by the Python script *$UWHPSC/labs/lab16/plot_cond.py*, which plots the results on a logarithmic scale, along with what the formula predicts.

- For the function version you do not need to solve a linear system, so you don't need to call *dgesv*, but you do need to compute the LU factorization of A before calling *dgecon*. The could be done by calling *dgetrf* instead of *dgesv*. You might want to look at the dgetrf documentation.

There is quiz for Lab 16

2.4.11 Lab 17: Tuesday May 27, 2014

Announcements

- Due tonight: Homework 4, lecture quizzes, lab quiz.
- Office hours today:
- In Odegaard after class, and then in Lewis 328 until 5pm,
- Online 5-6pm, look for announcement on Canvas, but may use GoToMeeting
- Final project: Part 1 will be posted soon, discussed on Thursday.

  Due Wednesday, June 11. (But don't wait to the last minute!)

Today's lab

- Go through the notebook *$UWHPSC/labs/lab17/Tridiagonal.ipynb*, also visible at http://nbviewer.ipython.org/url/faculty.washington.edu/rjl/notebooks/Tridiagonal.ipynb.
- Work in pairs on writing a Fortran program to solve the same tridiagonal linear system considered in the notebook with:

```
A =
  [[   1.  200.    0.    0.    0.]
   [  10.    2.  300.    0.    0.]
   [   0.   20.    3.  400.    0.]
   [   0.    0.   30.    4.  500.]
   [   0.    0.    0.   40.    5.]]

b =
  [ 201.  312.  423.  534.   45.]
```

- Use the LAPACK surbroutine *dgtsv*. See the documentation at http://www.netlib.no/netlib/lapack/double/dgtsv.f.
- **There is quiz for Lab 17**

2.4.12 Lab 18: Thursday May 29, 2014

- We will go through the notebook *$UWHPSC/homeworks/project/BVP.ipynb*, also visible at http://nbviewer.ipython.org/url/faculty.washington.edu/rjl/notebooks/BVP.ipynb. This outlines a recursive domain decomposition approach to solving a boundary value problem. Part 1 of the project is to convert this into Fortran with OpenMP.
- Working in pairs, copy this notebook to *BVP2.ipynb* and modify it to solve a Helmholtz equation (in one dimension) of the form:

  $$u''(x) + k^2 u(x) = -f(x)$$

  on the interval $0 < x < 1$ with specified boundary conditions.

  As an exact solution, consider the case $f(x) = 0$ in which case the general solution to $u''(x) = -k^2 u(x)$ is $u(x) = c_1 \cos(kx) + c_2 \sin(kx)$.

  The boundary value problem has a unique exact solution for any boundary values $u(0)$ and $u(1)$ provided that k is not an integer multiple of π. (Insert $x = 0$ and $x = 1$ into the general solution and determine c_1 and c_2 so that the boundary conditions are satisfied.)

You might try values such as:

```
k = 15.
u_left = 2.
u_right = -1.
```

You will need to use at least 40 or 50 grid points to get a solution that looks at all reasonable. If you make k larger, the solution will be more oscillatory and you will need even more grid points to get a reasonable approximation.

- Work through as much of the notebook as you can, adjusting things to solve the Helmholtz equation. The main objective is to work through the notebook and understand what is being done.

  Some tips:

  - Add another parameter *k* to the *solve_BVP_** functions.
  - In setting up the tridiagonal matrix in *solve_BVP_direct*, you will need to modify the diagonal terms for the difference equation that approximates the Helmholtz equation,

    $\frac{U_{i-1}-2U_i+U_{i+1}}{\Delta x^2} + k^2 U_i = -f(x_i)$

    This gives the linear system to be solved:

    $U_{i-1} + (k^2\Delta x^2 - 2)U_i + U_{i+1} = -\Delta x^2 f(x_i)$

    along with the boundary conditions.
  - If you get to the divide-and-conquer approach, you will have to modify the equation for the mismatch to take into account the modification to the linear system being solved.
- There is now a sample solution in the repository, visible at http://nbviewer.ipython.org/url/faculty.washington.edu/rjl/notebooks/BVP_helmholtz.ipynb.
- **There is quiz for Lab 18**

2.4.13 Lab 19: Tuesday June 3, 2014

- We will go through the notebook *$UWHPSC/homeworks/project/Heat_Equation.ipynb*, also visible at http://nbviewer.ipython.org/url/faculty.washington.edu/rjl/notebooks/Heat_Equation.ipynb.

  This notebook gives a brief introduction to the heat equation and two numerical methods for its solution, an explicit method and the more stable implicit Crank-Nicolson method.

Some things to try

- You might want to make a copy of this notebook before you start playing with it.
- Experiment with different initial conditions for the heat equation.
- Create an animation (in the notebook) of the numerical solution to the heat equation along with the true solution.
- The Crank-Nicolson method is *second order accurate*: the error should go to zero like $\mathcal{O}(\Delta t^2 + \Delta x^2)$ as the grid is refined. So increasing both *n* (the number of spatial points) and *nsteps* (the number of time steps) by a factor of 2 should reduce the error by a factor of 4. Test this out.
- Compute or look up the Fourier sine series for some interesting function and try this as initial conditions for the heat equation. Compare the true solution with the numerical solution (where the "true solution" might be estimated by adding up a large but finite number of terms in the Fourier series).
- Try using Sympy to compute the coefficients in the Fourier sine series.

There is quiz for Lab 19

2.4.14 Lab 20: Thursday June 5, 2014

- The directory *$UWHPSC/labs/lab20* contains a Fortran code *fourier_sum.f90* that computes terms in the Fourier sine series for the function $f(x) = e^x$ and prints them out to the file *frames.txt* along with partial sums of the series. It also creates a second file *xf.txt* that contains the values of *x* and *f(x)* on the finite grid where the terms and sum are computed.

  The Python script *animate.py* reads in these two files and produces an animation, which can be viewed at http://faculty.washington.edu/rjl/FourierSum.html.

  We will go through these codes.

- You may have to install JSAnimation on your SMC project, see *Animation in Python*.

Work on in pairs:

- The subdirectory *$UWHPSC/labs/lab20/gamblers_ruin* contains a modified version of the solution to Homework 4, part 2, and does a single random walk. The *gamblers.f90* code has been modified to print *n1, n2* each step to the file *walk.txt*.

  Produce a script *animate.py* and modify *Makefile1* so that you can do:

```
$ make movie -f Makefile1 n1=10 n2=10 seed=1234
```

  and generate a movie like the one shown at http://faculty.washington.edu/rjl/RandomWalk.html.

- Don't worry about the Makefile at first, get *animate.py* working.
- For testing, you might want to use smaller *n1* and *n2* so there are fewer *png* files to generate.

There is quiz for Lab 20

See also the codes in *$UWHPSC/labs*.

2.5 Homework

Homeworks assigned in 2013 can be found in *2013 Homework* and the code they refer to can be found in *$UWHPSC/2013/homeworks*. Some solutions to these homeworks are in the directory *$UWHPSC/2013/solutions*. These homeworks may be referenced in some of the lectures.

2.5.1 2014 Homework assignments

There will be 4 homeworks during the quarter with tentative due dates listed below:

- homework1: Thursday of Week 2, April 10
- homework2: Thursday of Week 4, April 24
- homework3: Thursday of Week 6, May 8 – homework3_solution
- homework4: Tuesday of Week 9, May 27 – homework4_solution
- project: Wednesday of Week 11, June 11 – project_hints

There will be a "final project" tentatively due on Wednesday, June 11. This will count twice as much as a homework and will be similar in spirit but longer and tying together several things from the quarter into a more interesting computing assignment.

2.6 Computing Options [2014 version]

All of the software we will use this year is open source, so in principle you can install it all on your own computer. See *Downloading and installing software for this class* for some tips on doing so.

However, there are several reasons you might want to use a different computing environment for this class:

- To avoid having to install many packages yourself,
- To make sure you have the same computing environment as fellow students and the TAs,
- To have access to a multi-core machine if your own computer is has a single processor, since much of the course material concerns parallel computing.
- To work together during lab sessions.

2.6.1 SageMathCloud

This is the recommended computing platform and what we will mostly use during the T-Th lab sessions. SageMathCloud is a freely available cloud computing resource developed by the Sage Team, led by Prof. William Stein in the UW Mathematics Department. You can easily create an account at SageMathCloud.

You should create an account using your UW email address *netid@uw.edu*. This will make it easiest for us to add you as a collaborator on projects.

All of the software needed this quarter is installed on SageMathCloud.

Use of SageMathCloud will be demonstrated during the first Lab session on Tuesday April 1, 2014.

For some tips on using it, see smc.

2.6.2 Virtual Machine

If you want to be able to compute on your own computer but don't want to try installing all the necessary software packages individually, another option is to run a *virtual machine* using the VirtualBox software. See *Virtual Machine for this class [2014 Edition]* for more information.

2.6.3 Amazon Web Services

Another possibility for cloud computing is to use Amazon Web Services. An Amazon Machine Image (a virtual machine) has been created that has all the software needed for this class. For more detail on how to launch an instance running this AMI, see *Amazon Web Services EC2 AMI [2014 version]*.

2.7 Downloading and installing software for this class

In 2014 we will use SageMathCloud for much of the computing in this class (see smc), but if you want to install software on your own computer, this page gives some hints.

Another option for computing in the cloud is to follow the instructions in the section *Amazon Web Services EC2 AMI [2014 version]*.

Rather than downloading and installing all of this software individually, you might want to consider using the *Virtual Machine for this class [2014 Edition]*, which already contains everything you need.

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* (see *Software available through apt-get*), or your system's package manager. Many Python packages can also be installed using *easy_install* (see *Software available through easy_install*).

If you must use a Windows PC, then you should download and install *[VirtualBox]* for Windows and then run the *Virtual Machine for this class [2014 Edition]* to provide a Linux environment. Some of the software we'll use is available on Windows, but we will assume you are using a Unix-like environment and learning to do so is part of the goal of this class.

If you are using a Mac and want to install the necessary software, you also need to install Xcode developer tools, which includes necessary compilers and *make*, for example.

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.

2.7.1 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
```

2.7.2 Individual packages

Python

If the version of Python on your computer is older than 2.7.0 (see above), you should upgrade.

See http://www.python.org/download/ or consider the Enthought Python Distribution (EPD) or Anaconda CE described below.

SciPy Superpack

On Mac OSX, you can often install gfortran and all the Python packages we'll need using the SciPy Superpack.

Enthought Python Distribution (EPD)

You might consider installing EPD free

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.

Anaconda CE

Anaconda is a new collection of Python tools distributed by Continuum Analytics The "community edition" Anaconda CE is free and contains most of the tools we'll be using, including IPython, NumPy, SciPy, matplotlib, and many others. The full Anaconda is also free for academic users.

IPython

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

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

NumPy and SciPy

Used for numerical computing in Python (see *Numerics in Python*). (Included in EPD, Anaconda, and the SciPy Superpack.)

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

Matplotlib

Matlab-like plotting package for 1d and 2d plots in Python. (Included in EPD, Anaconda, and the SciPy Superpack.)

See http://matplotlib.sourceforge.net/

Git

Version control system (see *Git*).

See downloads.

Sphinx

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

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/

Some instructions for installing from source on a Mac can be found at here.

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/

2.7.3 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 [2014 Edition]*, 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
Git	git
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.

The script below was used to install software on the Ubuntu VM:

```
# $UWHPSC/notes/install.sh
#
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install xfce4
sudo apt-get install jockey-gtk
sudo apt-get install xdm
sudo apt-get install ipython
sudo apt-get install python-numpy
sudo apt-get install python-scipy
sudo apt-get install python-matplotlib
sudo apt-get install python-dev
sudo apt-get install git
sudo apt-get install python-sphinx
sudo apt-get install gfortran
sudo apt-get install openmpi-bin
sudo apt-get install liblapack-dev
sudo apt-get install thunar
sudo apt-get install xfce4-terminal

# some packages not installed on the VM
# that you might want to add:

sudo apt-get install gitk               # to view git history
sudo apt-get install xxdiff             # to compare two files
sudo apt-get install python-sympy       # symbolic python
sudo apt-get install imagemagick        # so you can "display plot.png"


sudo apt-get install python-setuptools  # so easy_install is available
sudo easy_install nose                  # unit testing framework
sudo easy_install StarCluster           # to help manage clusters on AWS
```

2.7.4 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
Git	git
Sphinx	sphinx

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

2.8 Virtual Machine for this class [2014 Edition]

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 in the file uwhpsc.zip.

Note that the file is quite large (approximately 803 MB compressed), and if possible you should download it from on-campus to shorten the download time. The TA's will also have the VM on memory sticks for transferring.

2.8.1 System requirements

The VM is around 2 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.

2.8.2 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. 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.

2.8.3 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 `uwhpsc`. (We would have liked to set up a VM with no password, but many things in Linux assume you have one.)

Note that you will also need this password to quit the VM.

2.8.4 Running programs

You can access the programs on the virtual machine through the Applications Menu (the mouse on an *X* symbol in the upper-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.

2.8.5 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.

Also, if you move your computer from one wireless network to another while the VM is running, it may lose connection with the internet.

If this happens, it should be sufficient to shut down the VM (with the 0/1 button on the top right corner) and then restart it. On shutdown, a script is automatically run that does the following, which in earlier iterations of the VM had to be done manually...

```
$ 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.

2.8.6 Shutting down

When you are done using the virtual machine, you can shut it down by clicking the 0/1 button on the top-right corner of the VM. You will need the password *uwhpsc*.

2.8.7 Cutting and pasting

If you want to cut text from one window in the VM and paste it into another, you should be able to highlight the text and then type ctrl-c (or in a terminal window, ctrl-shift-C, since ctrl-c is the interrupt signal). To paste, type ctrl-v (or ctrl-shift-V in a terminal window).

If you want to be able to cut and paste between a window in the VM and a window on your host machine, click on Machine from the main VitualBox menu (or *Settings* in the Oracle VM VirtualBox Manager window), then click on *General* and then *Advanced*. Select *Bidirectional* from the *Shared Clipboard* menu.

2.8.8 Shared Folders

If you create a file on the VM that you want to move to the file system of the host machine, or vice versa, you can create a "shared folder" that is seen by both.

First create a folder (i.e. directory) on the host machine, e.g. via:

```
$ mkdir ~/uwhpsc_shared
```

This creates a new subdirectory in your home directory on the host machine.

In the VirtualBox menu click on *Devices*, then click on *Shared Folders*. Click the + button on the right side and then type in the full path to the folder you want to share under *Folder Path*, including the folder name, and then the folder name itself under *Folder name*. If you click on *Auto-mount* then this will be mounted every time you start the VM.

Then click *OK* twice.

Then, in the VM (at the linux prompt), type the following commands:

```
sharename=uwhpsc_shared   # or whatever name the folder has
sudo mkdir /mnt/$sharename
sudo chmod 777 /mnt/$sharename
sudo mount -t vboxsf -o uid=1000,gid=1000 $sharename /mnt/$sharename
```

You may need the password *uwhpsc* for the first *sudo* command.

The folder should now be found in the VM in */mnt/$sharename*. (Note *$sharename* is a variable set in the first command above.)

If auto-mounting doesn't work properly, you may need to repeat the final *sudo mount ...* command each time you start the VM.

2.8.9 Enabling more processors

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, open the VirtualBox *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.

2.8.10 Problems enabling multiple processors...

Users may encounter several problems with enabling mutliple processors. Some users may not be able to change this setting (it will be greyed out). Other users when may find no improved performance after enabling multiple processors. Still others may encounter an error such as:

```
VD: error VERR_NOT_SUPPORTED
```

All of these problems indicate that virtualization has not been enabled on your processors.

Fortunately this has an easy fix. You just have to enable virtualization in your BIOS settings.

1. To access the BIOS settings you must restart your computer and press a certain button on startup. This button will depend on the company that manufactures your computer (for example for Lenovo's it appears to be the f1 key).

2. Next you must locate a setting that will refer to either virtualization, VT, or VT-x. Again the exact specifications will depend on the computer's manufacturer, however it should be found in the Security section (or the Performance section if you are using a Dell).

3. Enable this setting, then save and exit the bios settings.After your computer reboots you should be able to start the VM using multiple processors now.

4. If your BIOS does not have any settings like this it is possible that your BIOS is set up to hide this option from you, and you may need to follow the advice here: http://mathy.vanvoorden.be/blog/2010/01/enable-vt-x-on-dell-laptop/

Note: Unfortunately some older hardware does not support virtualization, and so if these solutions don't work for you it may be that this is the case for your processors. There also may be other possible problems...so don't be afraid to ask the TAs for help!

2.8.11 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 Applications menu in the upper-left corner, pointing to *Settings*, then clicking *Display*. Choose a resolution from the drop-down list, then click *Apply*.

2.8.12 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.

2.8.13 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.

2.8.14 How This Virtual Machine was made

1. Download Ubuntu 12.04 PC (Intel x86) alternate install ISO from http://cdimage.ubuntu.com/xubuntu/releases/12.04.2/release/xubuntu-12.04.2-alternate-i386.iso
2. Create a new virtual box
3. Set the system as Ubuntu
4. Use defualt options
5. After that double click on your new virtual machine...a dropdown box should appear where you can select your ubuntu iso
6. As you are installing...at the first menu hit F4 and install a command line system
7. Let the install proceed following the instructions as given. On most options the default answer will be appropriate. When it comes time to format the hard drive, choose the manual option. Format all the free space and set it as the mount point. From the next list choose root (you dont need a swap space).
8. Install the necessary packages

```
# $UWHPSC/notes/install.sh
#
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install xfce4
sudo apt-get install jockey-gtk
sudo apt-get install xdm
sudo apt-get install ipython
sudo apt-get install python-numpy
sudo apt-get install python-scipy
sudo apt-get install python-matplotlib
sudo apt-get install python-dev
```

```
sudo apt-get install git
sudo apt-get install python-sphinx
sudo apt-get install gfortran
sudo apt-get install openmpi-bin
sudo apt-get install liblapack-dev
sudo apt-get install thunar
sudo apt-get install xfce4-terminal

# some packages not installed on the VM
# that you might want to add:

sudo apt-get install gitk               # to view git history
sudo apt-get install xxdiff             # to compare two files
sudo apt-get install python-sympy       # symbolic python
sudo apt-get install imagemagick        # so you can "display plot.png"


sudo apt-get install python-setuptools  # so easy_install is available
sudo easy_install nose                  # unit testing framework
sudo easy_install StarCluster           # to help manage clusters on AWS
```

9. To setup the login screen edit the file Xresources so that the greeting line says.:

```
xlogin*greeting: Login and Password are uwhpsc
```

10. Create the file uwhpscvm-shutdown.:

```
#!/bin/sh
#Finish up Script--does some cleanup then shuts down

#prevent network problems
rm -f /etc/udev/rules.d/70-persistent-net.rules

#shutdown
shutdown -h now
```

11. Save it at.:

```
/usr/local/bin/uwhpscvm-shutdown
```

12. Execute the following command command.:

```
$ sudo chmod +x /usr/local/bin/uwhpscvm-shutdown
```

13. Right click on the upper panel and select add new items and choose to add a new launcher.
14. Name the new launcher something like shutdown and in the command blank copy the following line.:

```
gksudo /usr/local/bin/uwhpscvm-shutdown
```

15. Go to preferred applications and select Thunar for file managment and the xfce4 terminal.
16. Run jockey-gtk and install guest-additions.
17. Go to Applications then Settings then screensaver and select “disable screen saver” mode
18. In the settings menu select the general settings and hit the advanced tab. Here you can set the clipboard and drag and drop to allow Host To Guest.

19. Shutdown the machine and then go to the main virtualbox screen. Click on the virtualmachine and then hit the settings button.
20. After, in the system settings click on the processor tab. This may let you allow the virtual machine to use more than one processor (depending on your computer). Choose a setting somewhere in the green section of the Processors slider.

2.8.15 About the VM

The class virtual machine is running XUbuntu 12.04, 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.

2.8.16 Further reading

[VirtualBox] [VirtualBox-documentation]

2.9 Amazon Web Services EC2 AMI [2014 version]

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*. You can also use the *Virtual Machine for this class [2014 Edition]*.

Another alternative is to write and run your programs "in the cloud" using Amazon Web Services (AWS) Elastic Cloud Computing (EC2). You can start up an "instance" (your own private computer, or so it appears) that is configured using an Amazon Machine Image (AMI) that has been configured with the Linux operating system and containing all the software needed for this class.

You must first sign up for an account on the AWS main page. For this you will need a credit card, but note that with an account you can get 750 hours per month of free "micro instance" usage in the free usage tier. A micro instance is a single processor (that you will probably be sharing with others) so it's not suitable for trying out parallel computing, but should be just fine for much of the programming work in this class.

You can start up more powerful instances with 2 or more processors for a cost starting at less than 3 cents per hour (the m3.large on-demand instance). See the pricing guide.

For general information and guides to getting started:

- Getting started with EC2, with tutorial to lead you through an example.
- EC2 FAQ.
- Pricing. Note: you are charged per hour for hours (or fraction thereof) that your instance is in *running* mode, regardless of whether the CPU is being used.
- High Performance Computing on AWS with instructions on starting a cluster instance.
- UW eScience information on AWS.

2.9.1 Launching an instance with the *uwhpsc* AMI

Quick way

Navigate your browser to https://console.aws.amazon.com/ec2/home?region=us-west-2#launchAmi=ami-501a7260

You should then be on a page where you see you are on Step 2 of 7 at the top of the page, "Choose instance type".

Then you can skip the next section and proceed to *Choose instance type*.

Search for AMI

Skip this section if you followed the "quick way" instructions above.

Going through this part may be useful if you want to see how to search for other AMI's in the future.

Once you have an AWS account, sign in to the management console and click on the EC2 tab, and then select Region US West (Oregon) from the menu at the top right of the page, next to your user name.

You should now be on the page https://console.aws.amazon.com/ec2/v2/home?region=us-west-2.

Click on the big "Launch Instance" button.

On the next page, you will see a list of "Quick start" Amazon Machine Images (AMIs) that you can select from if you want to start with a fresh VM. For this class you don't want any of these. Instead click on the "Community AMIs" tab and then type *uwhpsc2014* in the search bar. Select this image.

You will then be taken to Step 2, "Choose instance type".

Choose instance type

On the next page you can select what sort of instance you wish to start (larger instances cost more per hour). t1-micro is the the size you can run free (as long as you only have one running).

Click *Continue* on the next few screens through the "instance details" and eventually you get to one that asks for a key pair. If you don't already have one, create a new one and select it here.

You can now skip over steps 3-6 and jump directly to Step 7, "Review and Launch".

2.9.2 Launch instance / create key pair

When you click on "Launch", you will get a page that asks you to "Select and existing key pair or create a new one". If you don't already have a key pair, select "Create a new pair" from the menu and follow instructions. If you give the name *mykey*, for example, then this will download a file *mykey.pem*. Store this file in a directory where you can find it again, since you will need this key in order to log in to your instance once it is running.

You also need to change the permissions on this file so that is readable only by the account user. In the directory where this file lives, give the command:

```
chmod 400 mykey.pem
```

If the file is more widely readable then you will not be able to use this key to log into your instance.

2.9.3 Logging on to your instance

Click *View Instances* on the page that appears to go back to the Management Console. Click on *Instances* on the left menu and you should see a list of instance you have created, in your case only one. If the status is not yet *running* then wait until it is (click on the *Refresh* button if necessary).

Click on the instance and information about it should appear at the bottom of the screen. Scroll down until you find the *Public DNS* information

Go into the directory where your key pair is stored, in a file with a name like *mykey.pem* and you should be able to *ssh* into your instance using the name of the public DNS, with format like:

```
$ ssh -Y -i KEYPAIR-FILE  ubuntu@DNS
```

where KEYPAIR-FILE and DNS must be replaced by the appropriate things, e.g. something like this:

```
$ ssh -Y -i mykey.pem ubuntu@ec2-50-19-75-229.compute-1.amazonaws.com
```

Note:

- You must include *-i keypair-file*
- You must log in as user ubuntu.
- Including -Y in the ssh command allows X window forwarding, so that if you give a command that opens a new window (e.g. plotting in Python) it will appear on your local screen.
- See the section *Using ssh to connect to remote computers* for tips if you are using a Mac or Windows machine. If you use Windows, see also the Amazon notes on using *putty* found at http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/putty.html.

Once you have logged into your instance, you are on Ubuntu Linux that has software needed for this class pre-installed. See the file *install.sh* in the running instance to see the commands that were used to install software.

Other software is easily installed using *apt-get install*, as described in *Downloading and installing software for this class*.

2.9.4 Transferring files to/from your instance

You can use *scp* to transfer files between a running instance and the computer on which the ssh key is stored.

From your computer (not from the instance):

```
$ scp -i KEYPAIR-FILE FILE-TO-SEND ubuntu@DNS:REMOTE-DIRECTORY
```

where DNS is the public DNS of the instance and *REMOTE-DIRECTORY* is the path (relative to home directory) where you want the file to end up. You can leave off *:REMOTE-DIRECTORY* if you want it to end up in your home directory.

Going the other way, you can download a file from your instance to your own computer via:

```
$ scp -i KEYPAIR-FILE ubuntu@DNS:FILE-TO-GET .
```

to retrieve the file named *FILE-TO-GET* (which might include a path relative to the home directory) into the current directory.

2.9.5 Stopping your instance

Once you are done computing for the day, you will probably want to stop your instance so you won't be charged while it's sitting idle. You can do this by selecting the instance from the Management Console / Instances, and then select *Stop* from the *Instance Actions* menu.

You can restart it later and it will be in the same state you left it in. But note that it will probably have a new Public DNS!

2.9.6 Creating your own AMI

If you add additional software and want to save a disk image of your improved virtual machine (e.g. in order to launch additional images in the future to run multiple jobs at once), simply click on *Create Image (EBS AMI)* from the *Instance Actions* menu.

2.9.7 Viewing webpages directly from your instance

An apache webserver should already be running in your instance, but to allow people (including yourself) to view webpages you will need to adjust the security settings. Go back to the Management Console and click on *Security Groups* on the left menu. Select *launch-wizard-1* and then click on *Inbound*. Click on *+Add rule*. You should see a list of ports that only lists 22 (SSH). You want to add port 80 (HTTP). Select HTTP from the drop-down menu that says *Custom TCP Rule* and then click on *+Add rule* and *Apply Rule Change*.

Now you should be able to point your browser to *http://DNS* where *DNS* is replaced by the Public DNS name of your instance, the same as used for the *ssh* command. So for the example above, this would be

```
http://ec2-50-19-75-229.compute-1.amazonaws.com
```

The page being displayed can be found in */var/www/index.html* on your instance. Any files you want to be visible on the web should be in */var/www*, or it is sufficient to have a link from this directory to where they are located (created with the *ln -s* command in linux).

So, for example, you could do the following (this has already been done if you start with the uwhpsc2104 AMI):

```
$ cd $HOME
$ mkdir public      # create a directory for posting things
$ chmod 755 public  # make it readable by others
$ sudo ln -s $HOME/public /var/www/public
```

then you can see the contents of your $HOME/public directory at:

```
http://ec2-50-19-75-229.compute-1.amazonaws.com/public
```

Remember to change the DNS above to the right thing for your own instance!

2.10 Software Carpentry

Sofware Carpentry started out as a course at the University of Toronto taught by Greg Wilson. He has since left the university and turned this into a major effort to help researchers be more productive by teaching them basic computing skills.

Their "boot camps" are taught at many universities and other insitutions around the world, check the calendar to see if one is coming up near you.

Some sections particularly relevant to this course, with links to videos:

- The Shell
- Version Control (using Subversion – svn)
- Testing
- Python
- Systems Programming (from Python)
- Classes and Objects (object oriented programming)

- Make
- Matrix programming (linear algebra, NumPy)
- Regular expressions

CHAPTER

THREE

TECHNICAL TOPICS

3.1 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 Leopard, Lion, etc. depending on version) is essentially a flavor of Unix.

3.1.1 Unix shells

See also the Software Carpentry lectures on The Shell.

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 2.7.3 (default, Aug 28 2012, 13:37:53)
[GCC 4.2.1 Compatible Apple Clang 4.0 ((tags/Apple/clang-421.0.60))] 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 *Numerics in 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.

3.1.2 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 2.7.2 (default, Jun 20 2012, 16:23:33)
Type "copyright", "credits" or "license" for more information.

IPython 0.14.dev -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

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 [2014 Edition]*.

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

3.1.3 Further reading

See *[IPython-documentation]*

3.2 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.

3.2.1 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/uwhpsc
```

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
```

3.2.2 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]
```

3.2.3 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 myhg.

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
```

3.2.4 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
```

3.2.5 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".

3.2.6 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 *Unix top command* for some examples.

3.2.7 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.

3.2.8 sudo

A command like:

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

found in the section *Virtual Machine for this class [2014 Edition]* 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 *Software available through apt-get*, where only those with super user permission can install software on to the system.

3.2.9 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]*.

3.2.10 .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.

3.2.11 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/uwhpsc/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/uwhpsc
```

which will go to the uwhpsc 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 UWHPSC=$HOME/uwhpsc
```

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 $UWHPSC
```

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.

3.2.12 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).

3.2.13 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.

3.2.14 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.

3.2.15 Further reading

[Wikipedia-unix-utilities]

3.3 Unix `top` command

The Unix `top` command is a very useful way to see what programs are currently running on the system and how heavily they are using system resources. (The command is named "top" because it shows the top users of the system.) It is a good idea to run `top` to check for other users running large programs before running one yourself on a shared computer (such as the Applied Math servers), and you can also use it to monitor your own programs.

3.3.1 Running `top`

To run `top`, simply type:

```
$ top
```

at the command line. `top` will fill your terminal window with a real-time display of system status, with a summary area displaying general information about memory and processor usage in the first few lines, followed by a list of processes and information about them. An example display is shown below, with the process list truncated for brevity:

```
top - 14:45:34 up  6:32,  2 users,  load average: 0.78, 0.61, 0.59
Tasks: 110 total,   3 running, 106 sleeping,   0 stopped,   1 zombie
Cpu(s): 75.0%us,  0.6%sy,  0.0%ni, 24.4%id,  0.0%wa,  0.0%hi,  0.0%si,
0.0%st
Mem:    507680k total,   491268k used,    16412k free,    24560k buffers
Swap:        0k total,        0k used,        0k free,   316368k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 2342 uwhpsc    20   0  8380 1212  796 R  300  0.2   0:28.55 jacobi2d.exe
  842 root      20   0  100m  24m 9780 R    2  5.0 178:27.50 Xorg
 1051 uwhpsc    20   0 40236  11m 8904 S    1  2.3   0:01.53 xfce4-terminal
    1 root      20   0  3528 1864 1304 S    0  0.4   0:01.12 init
    2 root      20   0     0    0    0 S    0  0.0   0:00.00 kthreadd
```

3.3.2 Summary area

The summary area shows a great deal of information about the general state of the system. The main highlights are the third through fifth lines, which show CPU and memory usage.

CPU is given as percentages spent doing various tasks. The abbreviations have the following meanings:

- `us` – time spent running non-niced user processes (most of your programs will fall into this category)
- `sy` – time spent running the Linux kernel and its processes
- `ni` – time spent running niced use processes
- `id` – time spent idle
- `wa` – time spent waiting for I/O

The fourth and fifth lines show the usage of physical (i.e. actual RAM chips) and virtual memory. The first three fields are mostly self-explanatory, though on Linux the `used` memory includes disk cache. (Linux uses RAM that's not allocated by programs to cache data from the disk, which can improve the computer's performance because RAM is much faster than disk.) `buffers` gives the amount of memory used for I/O buffering, and `cached` is the amount used by the disk cache. Programs' memory allocation takes priority over buffering and caching, so the total amount of memory available is the sum `free + buffers + cached`.

3.3.3 Task area

The task area gives a sorted list of the processes currently running on your system. By default, the list is sorted in descending order of CPU usage; the example display above is sorted by memory usage instead. Many different fields can be displayed in the task area; the default fields are:

- `PID`: Process ID number. Useful for killing processes.
- `USER`: Name of the user running the process.
- `PR`: Relative priority of the process.
- `NI`: Nice value of the process. Negative nice values give a process higher priority, while positive ones give it lower priority.
- `VIRT`: Total amount of memory used by the process. This includes all code, data, and shared libraries being used. The field name comes from the fact that this includes memory that has been swapped to disk, so it measures the total virtual memory used, not necessarily how much of it is currently present in RAM.
- `RES`: Total amount of physical ("resident") memory used by the process.
- `S`: Process status. The most common values are `S` for "Sleeping" or `R` for "Running".
- `%CPU`: Percent of CPU time being used by the process. This is relative to a single CPU; in a multiprocessor system, it may be higher than 100%.
- `%MEM`: Percent of available RAM being used by the process. This does not include data that has been swapped to disk.
- `TIME+`: Total CPU time used by the process since it started. This only counts the time the process has spent using the CPU, not time spent sleeping.
- `COMMAND`: The name of the program.

3.3.4 Interactive commands

There are many useful commands you can issue to `top`; only a few of them are listed here. For more information, see the `top` manual page.

Command	Meaning
`q`	Quit
`?`	Help
`u`	Select processes belonging to a particular user. Useful on shared systems.
`k`	Kill a process
`F`	Select which field to sort by

3.3.5 Further reading

For more information, see the quick reference guide at http://www.oreillynet.com/linux/cmd/cmd.csp?path=t/top, or type `man top` in a Unix terminal window to read the manual page.

3.4 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.

On Linux or a Mac, the *ssh* command should work fine in a terminal. On Windows, you may need to install something like putty.

3.4.1 X-window forwarding

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
```

In order for X-windows forwarding to work you must be running a X-window server on your machine. If you are running on a linux machine this is generally not an issue. On a Mac you need to install the *Xcode developer tools* (which you will need anyway).

On Windows you will need something like xming. A variety of tutorials on using *putty* and *xming* together can be found by googling "putty and xming".

3.4.2 scp

To transfer files you can use *scp*, similar to the copy command *cp* but used when the source and destination are on different computers:

```
$ scp somefile username@host:somedirectory
```

which would copy *somefile* in your local directory to *somedirectory* on the remote *host*, which is an address like *homer.u.washington.edu*, for example.

Going in the other direction, you could copy a remote file to your local machine via:

```
$ scp username@host:somedirectory/somefile .
```

The last "." means "this directory". You could instead give the path to a different local directory.

You will have to type your password on the remote host each time you do this, unless you have remote ssh access set up, see for example this page.

3.5 Text editors

3.5.1 Leafpad

A simple and lightweight editor, installed on the course VM.

To use it, click on the green leaf in the bottom menu, or at the bash prompt:

```
$ leafpad <filename>
```

3.5.2 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. It can be installed on the course VM via:

```
$ sudo apt-get install gedit
```

3.5.3 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.

3.5.4 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.

3.5.5 TextWrangler

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

3.5.6 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.

3.5.7 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 via:

```
$ sudo apt-get install emacs
```

3.6 Reproducibility

Whenever you do a computation you should strive to make it reproducible, so that if you want to produce the same result again at a later time you will be able to. This is particularly true if the result is important to your research and will ultimately be used in a paper or thesis you are writing, for example.

This may take a bit more time and effort initially, but doing so routinely can save you an enormous amount of time in the future if you need to slightly modify what you did, or if you need to reconstruct what you did in order to convince yourself or others that you did it properly, or as the basis for future work in the same direction.

In addition to the potential benefits to your own productivity in the future, reproducibility of scientific results is a cornerstone of the scientific method — it should be possible for other researchers to independently verify scientific claims by repeating experiments. If it is impossible to explain exactly what you did, then it will be impossible for others to repeat your computational experiments. Unfortunately most publications in computational science fall short when it comes to reproducibility. Algorithms and data analysis techniques are usually inadequately described in publications, and readers would find it very hard to reproduce the results from scratch. An obvious solution is to make the actual computer code available. At the very least, the authors of the paper should maintain a complete copy of the code that was used to produce the results. Unfortunately even this is often not the case.

There is growing concern in many computational science communities with the lack of reproducibility, not only because of the negative effect on productivity and scientific progress, but also because the results of simulations are increasingly important in making policy decisions that can have far-reaching consequences.

Various efforts underway to develop better tools for assisting in doing computational work in a reproducible manner and to help provide better access to code and data that is used to obtain research results.

Those interested in learning more about this topic might start with the resources listed in *Reproducibility references*.

In this class we will concentrate on version control as one easy way to greatly improve the reproducibility of your own work. If you create a *git* repository for each project and are diligent about checking in versions of the code that create important research products, with suitable commit messages, then you will find it much easier in the long run to reconstruct the version used to produce any particular result.

3.6.1 Scripting vs. using a shell or GUI

Many graphics packages let you type commands in at a shell prompt and/or click buttons on a GUI to generate data or adjust plot parameters. While very convenient, this makes it hard to reproduce the resulting plot unless you remember exactly what you did. It is a very good idea to instead write a script that produces the plot and that sets all the appropriate plotting parameters in such a way that rerunning the script gives exactly the same plot. With some plotting tools such a script can be automatically generated after you've come up with the optimal plot by fiddling around with the GUI or by typing commands at the prompt. It's worth figuring out how to do this most easily for your own tools and work style. If you always create a script for each figure, and then check that it works properly and commit it to the *git* repository for the project, then you will be able to easily reproduce the figure again later.

Even if you are not concerned with allowing others to create the same plot, it is in your own best interest to make it easy for you to do so again in the future. For example, it often happens that the referees of a paper or members of a thesis committee will suggest improving a figure by plotting something differently, perhaps as simple as increasing the font size so that the labels on the axes can be read. If you have the code that produced the plot this is easy to do in a few minutes. If you don't, it may take days (or longer) to figure out again exactly how you produced that plot to begin with.

3.6.2 IPython history command

If you are using IPython and typing in commands at the prompt, the *history* command can be used to print out a list of all the commands entered, or something like:

```
In[32]:  history 19-31
```

to print out commands *In[19]* through *In[31]*.

3.7 Version Control Software

In this class we will use *git*. See the section *Git* for more information on using *git* and the repositories required for this class.

There are many other version control systems that are currently popular, such as cvs, Subversion, Mercurial, and Bazaar. 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 *Reproducibility*). 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.

3.7.1 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.

3.7.2 Distributed systems

Git, and other systems such as Mercurial and Bazaar, 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 *git* works is to use it, for example by following the instructions in Section *Git*.

3.7.3 Further reading

See the *Version control systems references* section of the bibliography.

3.8 Git

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

3.8.1 Instructions for cloning the class repository

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

Note: This repository contains all of the files used during the 2013 version of this course, some of which are referred to in the Lecture Videos. The repository also contains some new material for 2014 and will continue to be added to during this quarter. See the overview at the top of the page http://bitbucket.org/rjleveque/uwhpsc/ for an outline of the directory structure. (This overview is showing the file README.md that is in the top level directory of the repository.)

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
$ git clone https://rjleveque@bitbucket.org/rjleveque/uwhpsc.git
```

Note the following:

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

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

```
$ export UWHPSC=$HOME/uwhpsc
```

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 uwhpsc
$ export UWHPSC=`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 UWHPSC
```

to make sure *UWHPSC* 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 UWHPSC=$HOME/uwhpsc
```

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 UWHPSC=full-path-to-uwhpsc
```

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

3.8.2 Updating your clone

The files in the class repository will change as the quarter progresses — new notes, 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 $UWHPSC
$ git fetch origin
$ git merge origin/master
```

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

The *git fetch* command instructs *git* to fetch any changes from *origin*, which points to the remote bitbucket repository that you originally cloned from. In the merge command, *origin/master* refers to the master branch in this repository (which is the only branch that exists for this particular repository). This merges any changes retrieved into the files in your current working directory.

The last two command can be combined as:

```
$ git pull origin master
```

or simply:

```
$ git pull
```

since *origin* and *master* are the defaults.

3.8.3 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 *git* for your own work without creating a repository on a hosted site such as Bitbucket (see newgit 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 a free private repository on Bitbucket. 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.

(To make free open access repositories that can be viewed by anyone, GitHub is recommended, which allows an unlimited number of open repositories and is widely used for open source projects.)

To get started, follow the directions below exactly. We will work through this as part of Lab 2 (Thursday, April 3), but feel free to start earlier.

Doing this will be part of homework1. 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:

```
$ git config --global user.name "Your Name"
$ git config --global user.email you@example.com
```

   These will be used when you commit changes. If you don't do this, you might get a warning message the first time you try to commit.

2. Go to http://bitbucket.org/ and click on "Sign up now" if you don't already have an account.
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" next to "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 *uwhpsc*, owned by user *rjleveque*. To avoid confusion, you should probably not name your repository *uwhpsc*.

   You should stick to lower case letters and numbers in your repository name, e.g. *myhpsc* or *amath583* might be good choices. Upper case and special symbols such as underscore sometimes get modified by bitbucket and the repository name you try to paste into the homework submission form might not agree with what bitbucket expects.

   Don't name your repository *homework1* because you will be using the same repository for other homeworks later in the quarter.
6. Make sure you click on "Private" at the bottom. Also turn "Issue tracking" and "Wiki" on if you wish to use these features.
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 *MYHPSC* from inside this directory by:

```
$ export MYHPSC=`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 *MYHPSC* similar to the line added for *UWHPSC*.

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

```
$ ls
```

    But try:

```
$ ls -a
./  ../  .git/
```

    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 *.git* 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 *git*.

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:

```
$ git status -s
```

The response should be:

```
?? testfile.txt
```

The ?? means that this file is not under revision control. The *-s* flag results in this *short* status list. Leave it off for more information.

To put the file under revision control, type:

```
$ git add testfile.txt
$ git status -s
A  testfile.txt
```

The A means it has been added. However, at this point *git* is not we have not yet taken a *snapshot* of this version of the file. To do so, type:

```
$ git commit -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.

You should get a response like:

```
[master (root-commit) 28a4da5] My first commit of a test file.
 1 file changed, 2 insertions(+)
 create mode 100644 testfile.txt
```

We can now see the status of our directory via:

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

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

```
$ git status testfile.txt
```

You can get a list of all the commits you have made (only one so far) using:

```
$ git log

commit 28a4da5a0deb04b32a0f2fd08f78e43d6bd9e9dd
Author: Randy LeVeque <rjl@ned>
Date:   Tue Mar 5 17:44:22 2013 -0800

    My first commit of a test file.
```

The number 28a4da5a0deb04b32a0f2fd08f78e43d6bd9e9dd above is the "name" of this commit and you can always get back to the state of your files as of this commit by using this number. You don't have to remember it, you can use commands like *git log* to find it later.

Yes, this is a number... it is a 40 digit hexadecimal number, meaning it is in base 16 so in addition to 0, 1, 2, ..., 9, there are 6 more digits a, b, c, d, e, f representing 10 through 15. This number is almost certainly guaranteed to be unique among all commits you will ever do (or anyone has ever done, for that matter). It is computed based on the state of all the files in this snapshot as a SHA-1 Cryptographic hash function, called a SHA-1 Hash for short.

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:

```
$ git status -s
 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:

```
$ git diff testfile.txt
```

This will produce something like:

```
diff --git a/testfile.txt b/testfile.txt
index d80ef00..fe42584 100644
--- a/testfile.txt
+++ b/testfile.txt
@@ -1,2 +1,3 @@
 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, *git diff* would only print a few lines around any change to indicate the context.

Now let's try to commit this changed file:

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

This will fail! You should get a response like this:

```
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working
#   directory)
#
#   modified:   testfile.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

git is saying that the file *testfile.txt* is modified but that no files have been **staged** for this commit.

If you are used to Mercurial, *git* has an extra level of complexity (but also flexibility): you can choose which modified files will be included in the next commit. Since we only have one file, there will not be a commit unless we add this to the **index** of files staged for the next commit:

```
$ git add testfile.txt
```

Note that the status is now:

```
$ git status -s
M  testfile.txt
```

This is different in a subtle way from what we saw before: The *M* is in the first column rather than the second, meaning it has been both modified and staged.

We can get more information if we leave off the *-s* flag:

```
$ git status

# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   testfile.txt
#
```

Now *testfile.txt* is on the index of files staged for the next commit.

Now we can do the commit:

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

[master 51918d7] added a third line to the test file
 1 file changed, 1 insertion(+)
```

Try doing *git log* now and you should see something like:

```
commit 51918d7ea4a63da6ab42b3c03f661cbc1a560815
Author: Randy LeVeque <rjl@ned>
Date:   Tue Mar 5 18:11:34 2013 -0800

    added a third line to the test file

commit 28a4da5a0deb04b32a0f2fd08f78e43d6bd9e9dd
Author: Randy LeVeque <rjl@ned>
Date:   Tue Mar 5 17:44:22 2013 -0800

    My first commit of a test file.
```

If you want to revert your working directory back to the first snapshot you could do:

```
$ git checkout 28a4da5a0de
Note: checking out '28a4da5a0de'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

HEAD is now at 28a4da5... My first commit of a test file.
```

Take a look at the file, it should be back to the state with only two lines.

Note that you don't need the full SHA-1 hash code, the first few digits are enough to uniquely identify it.

You can go back to the most recent version with:

```
$ git checkout master
Switched to branch 'master'
```

We won't discuss branches, but unless you create a new branch, the default name for your main branch is *master* and this *checkout* command just goes back to the most recent commit.

14. So far you have been using *git* 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:

    ```
    $ git status
    ```

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

    Now do:

    ```
    $ git push -u origin master
    ```

    This will prompt for your Bitbucket password and should then print something indicating that it has uploaded these two commits to your bitbucket repository.

    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.

    To push future commits to bitbucket, you should only need to do:

    ```
    $ git push
    ```

    and by default it will push your master branch (the only branch you have, probably) to *origin*, which is the shorthand name for the place you originally cloned the repository from. To see where this actually points to:

    ```
    $ git remote -v
    ```

    This lists all *remotes*. By default there is only one, the place you cloned the repository from. (Or none if you had created a new repository using *git init* rather than cloning an existing one.)

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 "Commits" 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.

    If you click on the hex-string for a commit, 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!

homework1 instructs you to add some additional files to the Bitbucket repository.

Feel free to experiment further with your repository at this point.

3.8.4 Further reading

Next see *Bitbucket repositories: viewing changesets, issue tracking* and/or *More about Git*.

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

```
$ git help
$ git help diff  # or any other specific command name
```

Each command has lots of options!

Git references contains references to other sources of information and tutorials.

3.9 Bitbucket repositories: viewing changesets, issue tracking

The directions in Section *Git* 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.

See also Bitbucket 101.

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 such as github, for example.)

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 documentation*). 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 myhg.

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

3.10 More about Git

3.10.1 Using *git* to stay synced up on multiple computers

If you want to use your *git* repository on two or more computers, staying in sync via bitbucket should work well. To avoid having merge conflicts or missing something on one computer because you didn't push it from the other, here are some tips:

- When you finish working on one machine, make sure your directory is "clean" (using "git status") and if not, add and commit any changes.
- Use "git push" to make sure all commits are pushed to bitbucket.
- When you start working on a different machine, make sure you are up to date by doing:

```
$ git fetch origin          # fetch changes from bitbucket
$ git merge origin/master   # merge into your working directory
```

These can probably be replaced by simply doing:

```
$ git pull
```

but for more complicated merges it's recommended that you do the steps separately to have more control over what's being done, and perhaps to inspect what was fetched before merging.

If you do this in a clean directory that was pushed since you made any changes, then this merge should go fine without any conflicts.

3.11 Sphinx documentation

Sphinx is a Python-based documentation system that allows writing documentation in a simple mark-up language called ReStructuredText, which can then be converted to html or to latex files (and then to pdf or postscript). See *[sphinx]* for general information, *[sphinx-documentation]* for a complete manual, and *[sphinx-rst]* or *[rst-documentation]* for a primer on ReStructuredText. See also *[sphinx-cheatsheet]*.

Although originally designed for aiding in documentation of Python software, it is now being used for documentation of packages in other languages as well. See *[sphinx-examples]* for a list of other projects that use Sphinx.

It can also be used for things beyond software documentation. In particular, it has been used to create the class note webpages that you are now reading. It was chosen for two main reasons:

1. It is a convenient way to create a set of hyper-linked web pages on a variety of topics without having to write raw html or worry much about formatting.
2. Writing good documentation is a crucial aspect of high performance scientific computing and students in this class should learn ways to simplify this task. For this reason many homework assignments must be "submitted" in the form of Sphinx pages.

The easiest way to learn how to create Sphinx pages is to read some of the documentation and then examine Sphinx pages such as the one you are now reading to see how it is written. You can do this with these class notes or you might look at one of the other *[sphinx-examples]* to see other styles.

Note that any time you are reading a page of these class notes (or other things created with Sphinx) there is generally a menu item *Show Source* (on this page it's on the left, under the heading *This Page*) and clicking on this link shows the raw text file as originally written. *Try this now*.

It is possible to customize Sphinx so the pages look very different, as you'll see if you visit some other projects listed at *[sphinx-examples]*.

3.11.1 Using Sphinx for your own project

If you want to create your own set of Sphinx pages for some project, it is easy to get started following the instructions at *[sphinx]*, or for a quick start with a different look and feel, see *[sphinx-sampledoc]*.

3.11.2 Using Sphinx to create these webpages

If you clone the git repository for this class (see *Git*), you will find a subdirectory called *notes*, containing a number of files with the extension *.rst*, one for each webpage, containing the ReStructuredText input.

To create the html pages, first make sure you have Sphinx installed via:

```
$ which sphinx-build
```

(see *Downloading and installing software for this class*) and then type:

```
$ make html
```

This should process all the files (see *Makefiles*) and create a subdirectory of *notes* called *_build/html*. The file *_build/html/index.html* contains the main Table of Contents page. Navigate your browser to this page using *Open File* on the *File* menu of your browser.

Or you may be able to direct Firefox directly to this page via:

```
$ firefox _build/html/index.html
```

3.11.3 Making a pdf version

If you type:

```
$ make latex
```

then Sphinx will create a subdirectory *_build/latex* containing latex files. If you have latex installed, you can then do:

```
$ cd _build/latex
$ make all-pdf
```

to run latex and create a pdf version of all the class notes.

3.11.4 Further reading

See *[sphinx]* for general information, *[sphinx-documentation]* for a complete manual, and *[sphinx-rst]* or *[rst-documentation]*. See also *[sphinx-cheatsheet]*.

See *[sphinx-sampledoc]* to get started on your own project.

3.12 Binary/metric prefixes for computer size, speed, etc.

Computers are often described in terms such as megahertz, gigabytes, teraflops, etc. This section has a brief summary of the meaning of these terms.

3.12.1 Prefixes

Numbers associated with computers are often very large or small and so standard scientific prefixes are used to denote powers of 10. E.g. a kilobyte is 1000 bytes and a megabyte is a million bytes. These prefixes are listed below, where 1e3 for example means 10^3:

```
kilo  = 1e3
mega  = 1e6
giga  = 1e9
tera  = 1e12
peta  = 1e15
 exa  = 1e18
```

Note, however, that in some computer contexts (e.g. size of main memory) these prefixes refer to nearby numbers that are exactly powers of 2. However, recent standards have given these new names:

```
kibi = 2^{10} = 1024
mebi = 2^{20} = 1048576
etc.
```

So 2^{20} bytes is a *mebibyte*, abbreviated MiB. For a more detailed discussion of this (and additional prefixes) see [wikipedia].

For numbers that are much smaller than 1 a different set of prefixes are used, e.g. a millisecond is 1/1000 = 1e-3 second:

```
mille = 1e-3
micro = 1e-6
 nano = 1e-9
 pico = 1e-12
femto = 1e-15
```

3.12.2 Units of memory, storage

The amount of memory or disk space on a computer is normally measured in bytes (1 byte = 8 bits) since almost everything we want to store on a computer requires some integer number of bytes (e.g. an ASCII character can be stored in 1 byte, a standard integer in 4 bytes, see storage).

Memory on a computer is generally split up into different types of memory implemented with different technologies. There is generally a large quantity of fairly slow memory (slow to move into the processor to operate on it) and a much smaller quantity of faster memory (used for the data and programs that are actively being processed). Fast memory is much more expensive than slow memeory, consumes more power, and produces more heat.

The *hard disk* on a computer is used to store data for long periods of time and is generally slow to access (i.e. to move into the core memory for processing), but may be large, hundreds of gigabytes (hundreds of billions of bytes).

The *main memory* or *core memory* might only be 1GB or a few GB. Computers also have a smaller amount of

3.12.3 Units of speed

The speed of a processor is often measured in Hertz (cycles per second) or some multiple such as Gigahertz (billions of cycles per second). This tells how many *clock cycles* are executed each second. Each instruction that a computer can execute takes some integer number of clock cycles. Different instructions may take different numbers of clock cycles. An instruction like "add the contents of registers 1 and 2 and store the result in register 3" will typically take only 2 clock cycles. On the other hand the instruction "load x into register 1" can take a varying number of clock cycles depending on where x is stored. If x is in cache because it has been recently used, this instruction may take only a few cycles. If it is not in cache and it must be loaded from main memory, it might take 100 cycles. If the data set used by the program is so huge that it doesn't all fit in memory and x must be retrieved from the hard disk, it can be orders of magnitude slower.

So knowing how fast your computer's processor is in Hertz does not necessarily directly tell you how quickly a given program will execute. It depends on what the program is doing and also on other factors such as how fast the memory accesses are.

3.12.4 Flops

In scientific computing we frequently write programs that perform many *floating point operations* such as multiplication or addition of two floating point numbers. A floating point operation is often called a **flop**.

For many algorithms it is relatively easy to estimate how many flops are required. For example, multiplying an $n \times n$ matrix by a vector of length n requires roughly n^2 flops. So it is convenient to know roughly how many floating point operations the computer can perform in one second. In this context **flops** often stands for *floating point operations per second*. For example, a computer with a peak speed of 100 Mflops can perform up to 100 million floating point operations per second. As in the above discussion on clock speed, the actual performance on a particular problem typically depends very much on factors other than the peak speed, which is generally measured assuming that all the data needed for the floating point operations is already in cache so there is no time wasted fetching data. On a real problem there may be many more clock cycles used on memory accesses than on the actual floating point operations. Because of this, counting flops is no longer a reliable way to determine how long a program will take to execute. It's also not possible to compare the relative merits of two algorithms for the same problem simply by counting how many arithmetic operations are required. For large problems, the way that data is accessed can be at least as important.

3.13 Computer Architecture

This page is currently a very brief survey of a few issues. See the links below for more details.

Not so long ago, most computers consisted of a single central processing unit (CPU) together with some a few registers (high speed storage locations for data currently being processed) and *main memory*, typically a hard disk from which data is moved in and out of the registers as needed (a slower operation). Computer speed, at least for doing large scale scientific computations, was generally limited by the speed of the CPU. Processor speeds are often quoted in terms of *Hertz* the number of *machine cycles per second* (or these days in terms of GigaHertz, GHz, in billions of cycles per second). A single operation like adding or multiplying two numbers might take more than one cycle, how many depends on the architecture and other factors we'll get to later. For scientific codes, the most important metric was often how many floating point operations could be done per second, measured in flops (or Gigaflops, etc.). See *Flops* for more about this.

3.13.1 Moore's Law

For many years computers kept getting faster primarily because the clock cycle was reduced and so the CPU was made faster. In 1965, Gordon Moore (co-founder of Intel) predicted that the transistor density (and hence the speed) of chips would double every 18 months for the forseeable future. This is know as **Moore's law** This proved remarkably accurate for more than 40 years, see the graphs at [wikipedia-moores-law]. Note that doubling every 18 months means an increase by a factor of 4096 every 14 years.

Unfortunately, the days of just waiting for a faster computer in order to do larger calculations has come to an end. Two primary considerations are:

- The limit has nearly been reached of how densely transistors can be packed and how fast a single processor can be made.
- Even current processors can generally do computations much more quickly than sufficient quantities of data can be moved between memory and the CPU. If you are doing 1 billion meaningful multiplies per second you need to move lots of data around.

There is a hard limit imposed by the speed of light. A 2 GHz processor has a clock cycle of 0.5e-9 seconds. The speed of light is about 3e8 meters per second. So in one clock cycle information cannot possibly travel more than 0.15 meters. (A light year is a long distance but a light nanosecond is only about 1 foot.) If you're trying to move billions of bits of information each second then you have a problem.

Another major problem is power consumption. Doubling the clock speed of a processor takes roughly 8 times as much power and also produces much more heat. By contrast, doubling the number of processors only takes twice as much power.

There are ways to continue improving computing power in the future, but they must include two things:

- Increasing the number of cores or CPUs that are being used simultaneously (i.e., parallel computing)
- Using memory hierachies to improve the ability to have large amounts of data available to the CPU when it needs it.

3.13.2 Further reading

See the *Computer architecture references* section of the bibliography.

See also the slides from *[Yelick-UCB]*, *[Gropp-UIUC]* and other courses listed in the bibliography.

3.14 Storing information in binary

All information stored in a computer must somehow be encoded as a sequence of 0's and 1's, because all storage devices consist of a set of locations that can have one of two possible states. One state represents 0, the other state represents 1. For example, on a CD or DVD there are billions of locations where either small pit has been created by a laser beam (representing a 1) or no pit exists (representing a 0). An old magnetic tape (such as a audio cassette tape or VHS video tape) consisted of a sequence of locations that could be magnetized with an upward or downward polarization, representing 0 or 1.

A single storage location stores a single bit (binary digit) of information. A set of 8 bits is a byte and this is generally the smallest unit of information a computer deals with. A byte can store $2^8 = 256$ different patterns of 0's and 1's and these different patterns might represent different things, depending on the context.

3.14.1 Integers

If we want to store an integer then it makes sense to store the binary representation of the integer, and in one byte we could store any of the numbers 0 through 255, with the usual binary representation:

```
00000000 = 0
00000001 = 1
00000010 = 2
00000011 = 3
00000100 = 4
etc.
11111111 = 255
```

Of course many practical problems involve integers larger than 256, and possibly negative integers as well as positive. So in practice a single integer is generally stored using more than 1 byte. The default for most computer languages is to use 4 bytes for an integer, which is 32 bits. One bit is used to indicate whether the number is positive of negative, leaving 31 bits to represent the values from 0 to $2^{31} = 2147483648$ as well as the negatives of these values. Actually it's a bit more complicated (no pun intended) since the scheme just described allows storing both +0 and -0 and a more complicated system allows storing one more integer, and in practice the two's complement representation is used, as described at [wikipedia], which shows a table of how the numbers -128 through 127 would actually be represented in one byte.

3.14.2 Real numbers

If we are dealing with real numbers rather than integers, a more complicated system is needed for storing arbitrary numbers over a fairly wide range in a fixed number of bytes.

Note that fractions can be represented in binary using inverse powers of 2. For example, the decimal number 5.625 can be expressed as $4 + 1 + \frac{1}{2} + \frac{1}{8}$, i.e.:

```
5.625 = 1*2^2 + 0*2^1 + 1*2^0 + 1*2^{-1} + 0*2^{-2} + 1*2^{-3}
```

and hence as 101.101 in binary.

Early computers used *fixed point* notation in which it was always assumed that a certain number of bits to the right of the decimal point were stored. This does not work well for most scientific computations, however. Instead computers now store real numbers as *floating point numbers*, by storing a *mantissa* and an *exponent*.

The decimal number 5.625 could be written as 0.5625×10^1 in normalized form, with mantissa 0.5625 and exponent 1.

Similarly, the binary number 101.101 in floating point form has mantissa 0.101101 and exponent 10 (the number 2 in binary, since the mantissa must be multiplied by $2^2 = 4$ to shift the binary point by two spaces).

Most scientific computation is done using 8-byte representation of real numbers (64 bits) in which 52 bits are used for the mantissa and 11 bits for the exponent (and one for the sign). See [wikipedia] for more details. This is the standard for objects of type *float* in Python. In Fortran this is sometimes called *double precision* because 4-byte floating point numbers (*single precision*) were commonly used for non-scientific applications. 8 byte floats are generally inadequate for most scientific computing problems, but there are some problems for which higher precision (e.g. *quad precision*, 16 bytes) is required.

Before the 1980's, different computer manufacturers came up with their own conventions for how to store real numbers, often handling computer arithmetic poorly and leading to severe problems in portability of computer codes between machines. The IEEE standards have largely solved this problem. See [wikipedia] for more details.

3.14.3 Text

If we are storing text, such as the words you are now reading, the characters must also be encoded as strings of 0's and 1's. Since a single byte can represent 256 different things, text is generally encoded using one byte for each character. In the English language we need 52 different patterns to represent all the possible letters (one for each lower case letter and a distinct pattern for the corresponding upper case letter). We also need 10 patterns for the digits and a fairly large number of other patterns to represent each other symbol (e.g. punctuation symbols, dollar signs, etc.) that might be used.

A standard 8-bit encoding is UTF-8 [wikipedia]. This is an extension of the earlier standard called ASCII [wikipedia], which only used 7 bits. For encoding a wider variety of symbols and characters (such as Chinese, Arabic, etc.) there are standard encodings UTF-16 and UTF-32 using more bits for each character.

See *Punch cards* for an example of how the ASCII character are represented on an punch card, an early form of computer memory.

Obviously, in order to interpret a byte stored in the computer, such as 01001011 properly, the computer needs to know whether it represents a UTF-8 character, a 1-byte integer, or something else.

3.14.4 Colors

Another thing a string of 0's and 1's might represent is a color, for example one pixel in an image that is being displayed. Each pixel is one dot of light and a string of 0's and 1's must be used to indicate what color each pixel should be. There are various possible ways to specify a color. One that is often used is to specify an RGB triple, three integers that indicate the amount of Red, Green, and Blue in the desired color. Often each value is allowed to range from 0 (indicating none) to 255 (maximal amount). These values can all be stored in 1 byte of data, so with this system 3 bytes (24 bits) are used to store the color of a single pixel. The color red, for example, has maximal R and 0 for G and B and hence has the first byte 256 and the next two bytes 0 and 0. Here is red a few other colors in their RGB and binary representations:

```
[255,  0,  0]  = 11111111 00000000 00000000 = red
[  0,  0,  0]  = 00000000 00000000 00000000 = black
[255,255,255]  = 11111111 11111111 11111111 = white
[ 57, 39, 91]  = 00111001 00100111 01011011 = official Husky purple
[240,213,118]  = 11110000 11010101 01110110 = official Husky gold
```

Colors in html documents and elsewhere are often specified by writing out exactly what each values each of the bytes should have. Writing out the bits as above is generally awkward for humans, so instead graphics languages like Matlab or Matplotlib in Python generally allow you to specify the RGB triple in terms of fractions between 0 and 1 (divide the RGB values above by 255 to get the fractions):

```
[1.0, 0.0, 0.0]  = red
[0.0, 0.0, 0.0]  = black
[1.0, 1.0, 1.0]  = white
[ 0.22352941,  0.15294118,  0.35686275] = official Husky purple
[ 0.94117647,  0.83529412,  0.4627451 ] = official Husky gold
```

Another way is common in html documents (and also allowed in Matplotlib), where the color red is denoted by the string:

```
'#ff0000' = red
'#000000' = black
'#ffffff' = white
'#39275b' = official Husky purple
'#f0d576' = official Husky gold
```

This string is written using *hexadecimal* notation (see below). The # sign just indicates that it's a hexidecimal string and it is followed by 3 2-digit hexidecimal numbers, e.g. for red they are ff, 00, and 00.

The latter two colors can be seen in the header of this webpage.

To find the hex string for a desired color, or view the color for a given string, try the farbtastic demo. See also [wikipedia-web-colors] or this page of colors showing RGB triples and hexcodes.

3.14.5 Hexidecimal numbers

A hexidecimal number is in base 16, e.g. the hexidecimal 345 represents:

```
345 = 3*(16)**2 + 4*(16)**1 + 5
```

which is 837 in decimal notation. Each hexidecimal digit can take one of 16 values and so in addition to the digits 0, 1, ..., 9 we need symbols to represent 10, 11, 12, 13, 14, and 15. For these the letters a,b,c,d,e,f are used, so for example:

```
a4e = 10*(16)**2 + 4*(16) + 14
```

which is 2638 in decimal notation.

Hex notation is a convenient way to express binary numbers because there is a very simple way to translate between hex and binary. To convert the hex number a4e to binary, for example, just translate each hex digit separately into binary, a = 1010, 4 = 0100, e = 1110, and then catenate these together, so:

```
a4e = 101001001110
```

in binary. Conversely, to convert a binary number such as 100101101 to hex, group the bits in groups of 4 (starting at the right, adding 0's to the left if necessary) and convert each group into a hex digit:

```
1100101101 = 0011 0010 1101 = 32d in hex.
```

Returning to the hex notation for colors, we can see that '#ff0000' corresponds to 111111110000000000000000, the binary string representing pure red.

3.14.6 Machine instructions

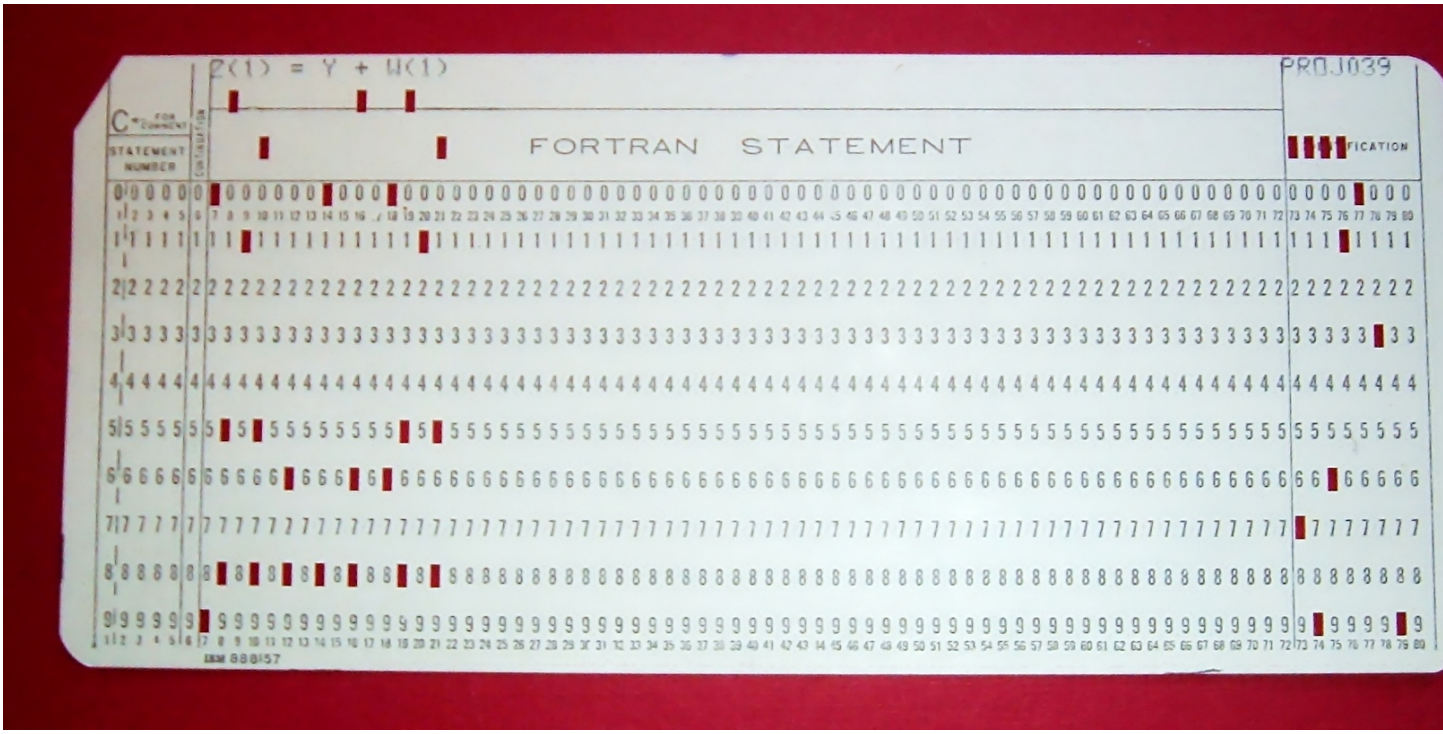
In addition to storing data (numbers, text, colors, etc.) as strings of bits, we must also store the computer instructions that make up a computer program as a string of bits. This may seem obvious now, but was actually a revolutionary idea when it was first proposed by the eminant mathematician (and one of the earliest "computer scientists") John von Neumann. The idea of a "von Neumann" machine in which the computer program is stored in the same way the data is, and can be modified at will to cause the computer to do different things, originated in 1930s and 40s through work of von Neumann, Turing, and Zuse (see [wikipedia]). Prior to this time, computers were designed to do one particular set of operations, executing one algorithm, and only the data could be easily changed. To "reprogram" the computer to do something different required physically rewiring the circuits.

Current computers store a sequence of instructions to be executed, as binary strings. These strings are different than the strings that might represent the instructions when the human-readable program is converted to ASCII to store as a text file.

See machine_code_and_assembly for more about this.

3.15 Punch cards

Once upon a time (through the 1970s) many computer programs were written on punch cards of the type shown here [image source]:

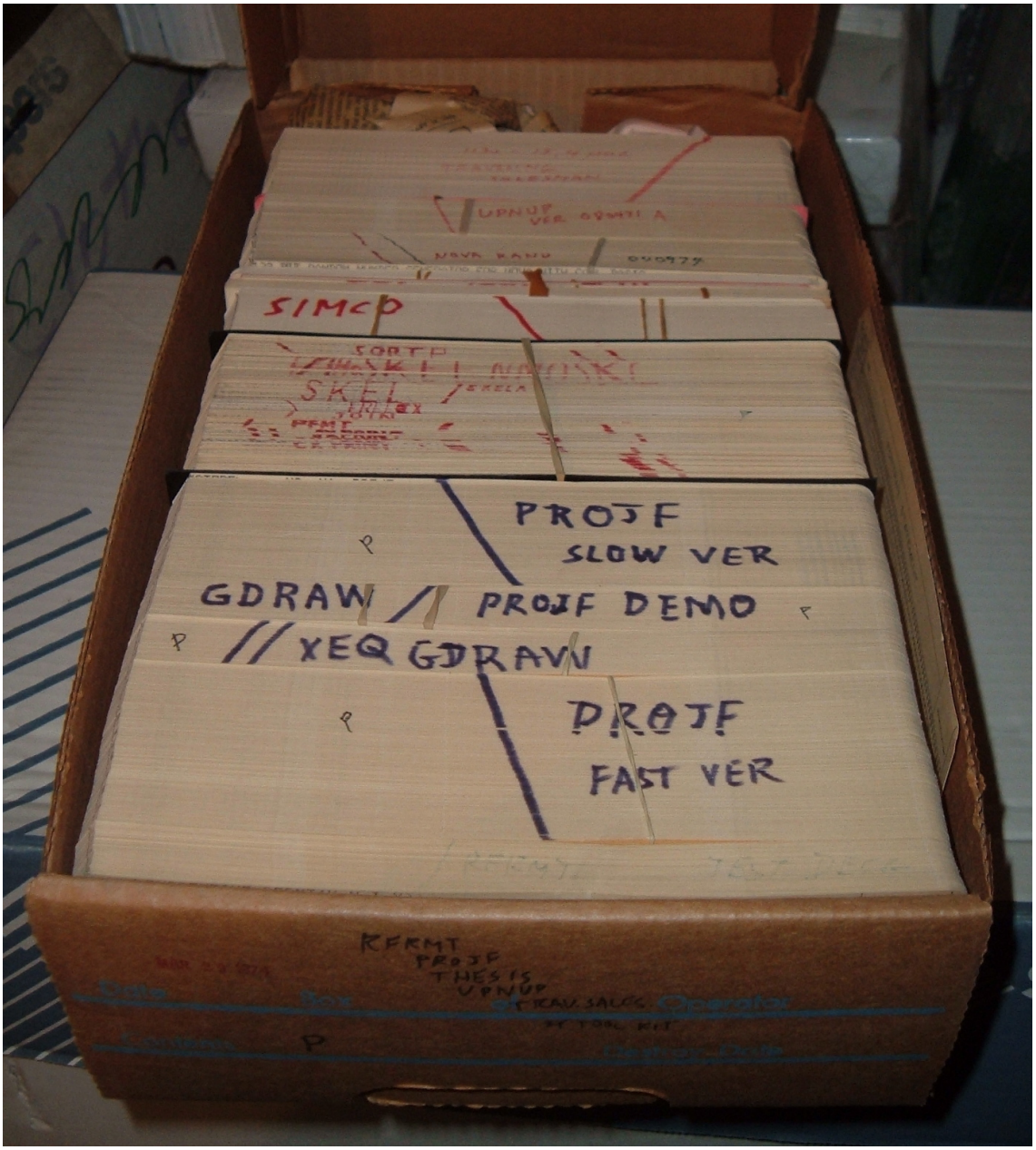


This is a form of binary memory (see *Storing information in binary*) where a specific set of locations each has a hole punched (representing 1) or not (representing 0). Each character typed on the top line is represented by the bits reading down the corresponding column, e.g. the first character Z is represented by 001000000001 since there are only two holes punched in this column.

When programs were written on cards, one card was required for each line of the program. The early conventions of the *Fortran* programming language are related to the columns on a punch card. Only the first 72 columns were used for the program statements. The last 8 columns could be used to print a card number, so that the unlucky programmer who dropped a deck of cards had some chance of reordering them properly. Many Fortran 77 compilers still ignore

any characters beyond column 72 in a line of a program, leading to bugs that can be hard to find if care is not taken, e.g. a called *xnew* might be truncated to *x* if the *x* is in column 72. If the program also uses a variable called *x*, this will not be caught by the compiler.

A program would require a *punch card deck* as shown in this photo [image source]:



Punch cards were a great step forward from *punched tape* [wikipedia] where a long strip of paper was used to store the entire program (and making a mistake required retyping more than just one card).

3.16 Python and Fortran

In this class we will use both Python and Fortran. Why these two?

- They are different types of languages that allow illustrating the difference between interperted vs. compiled languages, object oriented vs. procedural languages, etc.
- Each is very useful for certain common tasks in scientific computing. They can easily be combined to take advantage of the best of both worlds.
- Many scientific programs are written in one of these languages so you should be familiar with them.
- Both are freely available and students can set up laptops and desktops to use them beyond this class.

Learning two new languages in a quarter along with the many other topics we will be covering may seem overwhelming, but in many ways it makes sense to learn them together. By comparing features in the two languages it may help clarify what the essential concepts are, and what is truly different about the languages vs. simply different syntax or choices of convention.

We will not be writing extensive programs from scratch in this class. Instead, many homework sets will require making relatively small changes to Python of Fortran programs that have been provided as templates. This obviously isn't enough to become an expert programmer in either language, but the goal in this course is to get you started down that path.

3.16.1 Further reading

CHAPTER

FOUR

PYTHON

4.1 Python

These notes only scratch the surface of Python, with discussion of a few features of the language that are most important to getting started and to appreciating how Python can be used in computational science.

See the references below or the *Python:* section of the *Bibliography and further reading* for more detailed references.

See also the slides from lectures.

4.1.1 Interactive Python

The IPython shell is generally recommended for interactive work in Python (see http://ipython.org/documentation.html), but for most examples we'll display the >>> prompt of the standard Python shell.

Normally multiline Python statements are best written in a text file rather than typing them at the prompt, but some of the short examples below are done at the prompt. If type a line that Python recognizes as an unfinished block, it will give a line starting with three dots, like:

```
>>> if 1>2:
...     print "oops!"
... else:
...     print "this is what we expect"
...
this is what we expect
>>>
```

Once done with the full command, typing <return> alone at the ... prompt tells Python we are done and it executes the command.

4.1.2 Indentation

Most computer languages have some form of begin-end structure, or opening and closing braces, or some such thing to clearly delinieate what piece of code is in a loop, or in different parts of an if-then-else structure like what's shown above. Good programmers generally also indent their code so it is easier for a reader to see what is inside a loop, particularly if there are multiple nested loops. But in most languages this is indentation is just a matter of style and the begin-end structure of the language determines how it is actually interpreted by the computer.

In Python, indentation is everything. There are no begin-end's, only indentation. Everything that is supposed to be at one level of a loop must be indented to that level. Once the loop is done the indentation must go back out to the previous level. There are some other rules you need to learn, such as that the "else" in and if-else block like the above has to be indented exactly the same as as the "if". See if_else for more about this.

How many spaces to indent each level is a matter of style, but you must be consistent within a single code. The standard is often 4 spaces.

4.1.3 Wrapping lines

In Python normally each statement is one line, and there is no need to use separators such as the semicolon used in some languages to end a line. One the other hand you can use a semicolon to put several short statements on a single line, such as:

```
>>> x = 5; print x
5
```

It is easiest to read codes if you avoid this in most cases.

If a line of code is too long to fit on a single line, you can break it into multiple lines by putting a backslash at the end of a line:

```
>>> y = 3 + \
...     4
>>> y
7
```

4.1.4 Comments

Anything following a # in a line is ignored as a comment (unless of course the # appears in a string):

```
>>> s = "This # is part of the string"  # this is a comment
>>> s
'This # is part of the string'
```

There is another form of comment, the *docstring*, discussed below following an introduction to strings.

4.1.5 Strings

Strings are specified using either single or double quotes:

```
>>> s = 'some text'
>>> s = "some text"
```

are the same. This is useful if you want strings that themselves contain quotes of a different type.

You can also use triple double quotes, which have the advantage that they such strings can span multiple lines:

```
>>> s = """Note that a ' doesn't end
... this string and that it spans two lines"""

>>> s
"Note that a ' doesn't end\nthis string and that it spans two lines"

>>> print s
Note that a ' doesn't end
this string and that it spans two lines
```

When it prints, the carriage return at the end of the line show up as “n”. This is what is actually stored. When we “print s” it gets printed as a carriage return again.

You can put “n” in your strings as another way to break lines:

```
>>> print "This spans \n two lines"
This spans
 two lines
```

See *Python strings* for more about strings.

4.1.6 Docstrings

Often the first thing you will see in a Python script or module, or in a function or class defined in a module, is a brief description that is enclosed in triple quotes. Although ordinarily this would just be a string, in this special position it is interpreted by Python as a comment and is not part of the code. It is called the *docstring* because it is part of the documentation and some Python tools automatically use the docstring in various ways. See ipython for one example. Also the documentation formatting program Sphinx that is used to create these class notes can automatically take a Python module and create html or latex documentation for it by using the docstrings, the original purpose for which Sphinx was developed. See *Sphinx documentation* for more about this.

It's a good idea to get in the habit of putting a docstring at the top of every Python file and function you write.

4.1.7 Running Python scripts

Most Python programs are written in text files ending with the .py extension. Some of these are simple *scripts* that are just a set of Python instructions to be executed, the same things you might type at the >>> prompt but collected in a file (which makes it much easier to modify or reuse later). Such a script can be run at the Unix command line simply by typing "python" followed by the file name.

See *Python scripts and modules* for some examples. The section *Importing modules* also contains important information on how to "import" modules, and how to set the path of directories that are searched for modules when you try to import a module.

4.1.8 Python objects

Python is an object-oriented language, which just means that virtually everything you encounter in Python (variables, functions, modules, etc.) is an *object* of some *class*. There are many classes of objects built into Python and in this course we will primarily be using these pre-defined classes. For large-scale programming projects you would probably define some new classes, which is easy to do. (Maybe an example to come...)

The *type* command can be used to reveal the type of an object:

```
>>> import numpy as np
>>> type(np)
<type 'module'>

>>> type(np.pi)
<type 'float'>

>>> type(np.cos)
<type 'numpy.ufunc'>
```

We see that *np* is a module, *np.pi* is a floating point real number, and *np.cos* is of a special class that's defined in the numpy module.

The *linspace* command creates a numerical array that is also a special numpy class:

```
>>> x = np.linspace(0, 5, 6)
>>> x
array([ 0.,  1.,  2.,  3.,  4.,  5.])
>>> type(x)
<type 'numpy.ndarray'>
```

Objects of a particular class generally have certain operations that are defined on them as part of the class definition. For example, NumPy numerical arrays have a *max* method defined, which we can use on *x* in one of two ways:

```
>>> np.max(x)
5.0
>>> x.max()
5.0
```

The first way applies the method *max* defined in the *numpy* module to *x*. The second way uses the fact that *x*, by virtue of being of type *numpy.ndarray*, automatically has a *max* method which can be invoked (on itself) by calling the function *x.max()* with no argument. Which way is better depends in part on what you're doing.

Here's another example:

```
>>> L = [0, 1, 2]
>>> type(L)
<type 'list'>

>>> L.append(4)
>>> L
[0, 1, 2, 4]
```

L is a list (a standard Python class) and so has a method *append* that can be used to append an item to the end of the list.

4.1.9 Declaring variables?

In many languages, such as Fortran, you must generally declare variables before you can use them and once you've specified that *x* is a real number, say, that is the only type of things you can store in *x*, and a statement like *x = 'string'* would not be allowed.

In Python you don't declare variables, you can just type, for example:

```
>>> x = 3.4
>>> 2*x
6.7999999999999998

>>> x = 'string'
>>> 2*x
'stringstring'

>>> x = [4, 5, 6]
>>> 2*x
[4, 5, 6, 4, 5, 6]
```

Here *x* is first used for a real number, then for a character string, then for a list. Note, by the way, that multiplication behaves differently for objects of different type (which has been specified as part of the definition of each class of objects).

In Fortran if you declare *x* to be a real variable then it sets aside a particular 8 bytes of memory for *x*, enough to hold one floating point number. There's no way to store 6 characters or a list of 3 integers in these 8 bytes.

In Python it is often better to think of *x* as simply being a pointer that points to some object. When you type "x = 3.4" Python creates an object of type *float* holding one real number and points *x* to that. When you type *x = 'string'* it creates a new object of type *str* and now points *x* to that, and so on.

4.1.10 Lists

We have already seen lists in the example above.

Note that indexing in Python always starts at 0:

```
>>> L = [4,5,6]
>>> L[0]
4
>>> L[1]
5
```

Elements of a list need not all have the same type. For example, here's a list with 5 elements:

```
>>> L = [5, 2.3, 'abc', [4,'b'], np.cos]
```

Here's a way to see what each element of the list is, and its type:

```
>>> for index,value in enumerate(L):
...     print 'L[%s] is %16s     %s' % (index,value,type(value))
...
L[0] is                5     <type 'int'>
L[1] is              2.3     <type 'float'>
L[2] is              abc     <type 'str'>
L[3] is          [4, 'b']     <type 'list'>
L[4] is    <ufunc 'cos'>     <type 'numpy.ufunc'>
```

Note that *L[3]* is itself a list containing an integer and a string and that *L[4]* is a function.

One nice feature of Python is that you can also index backwards from the end: since *L[0]* is the first item, *L[-1]* is what you get going one to the left of this, and wrapping around (periodic boundary conditions in math terms):

```
>>> for index in [-1, -2, -3, -4, -5]:
...     print 'L[%s] is %16s' % (index, L[index])
...
L[-1] is    <ufunc 'cos'>
L[-2] is          [4, 'b']
L[-3] is               abc
L[-4] is               2.3
L[-5] is                 5
```

In particular, *L[-1]* always refers to the *last* item in list *L*.

4.1.11 Copying objects

One implication of the fact that variables are just pointers to objects is that two names can point to the same object, which can sometimes cause confusion. Consider this example:

```
>>> x = [4,5,6]
>>> y = x
>>> y
[4, 5, 6]

>>> y.append(9)
```

```
>>> y
[4, 5, 6, 9]
```

So far nothing too surprising. We initialized *y* to be *x* and then we appended another list element to *y*. But take a look at *x*:

```
>>> x
[4, 5, 6, 9]
```

We didn't really append 9 to *y*, we appended it to the object *y* points to, which is the same object *x* points to!

Failing to pay attention to this sort of thing can lead to programming nightmares.

What if we really want *y* to be a different object that happens to be initialized by copying *x*? We can do this by:

```
>>> x = [4,5,6]
>>> y = list(x)
>>> y
[4, 5, 6]

>>> y.append(9)
>>> y
[4, 5, 6, 9]

>>> x
[4, 5, 6]
```

This is what we want. Here *list(x)* creates a new object, that is a list, using the elements of the list *x* to initialize it, and *y* points to this new object. Changing this object doesn't change the one *x* pointed to.

You could also use the *copy* module, which works in general for any objects:

```
>>> import copy
>>> y = copy.copy(x)
```

Sometimes it is more complicated, if the list *x* itself contains other objects. See http://docs.python.org/library/copy.html for more information.

There are some objects that cannot be changed once created (*immutable objects*, as described further below). In particular, for *floats* and *integers*, you can do things like:

```
>>> x = 3.4
>>> y = x
>>> y = y+1
>>> y
4.4000000000000004

>>> x
3.3999999999999999
```

Here changing *y* did not change *x*, luckily. We don't have to explicitly make a copy of *x* for *y* in this case. If we did, writing any sort of numerical code in Python would be a nightmare.

We didn't because the command:

```
>>> y = y+1
```

above is not changing the object *y* points to, instead it is creating a new object that *y* now points to, while *x* still points to the old object.

For more about built-in data types in Python, see http://docs.python.org/release/2.5.2/ref/types.html.

4.1.12 Mutable and Immutable objects

Some objects can be changed after they have been created and others cannot be. Understanding the difference is key to understanding why the examples above concerning copying objects behave as they do.

A list is a *mutable* object. The statement:

```
$ x = [4,5,6]
```

above created an object that *x* points to, and the data held in this object can be changed without having to create a new object. The statement

$ y = x

points *y* at the same object, and since it can be changed, any change will affect the object itself and be seen whether we access it using the pointer *x* or *y*.

We can check this by:

```
>>> id(x)
1823768

>>> id(y)
1823768
```

The *id* function just returns the location in memory where the object is stored. If you do something like *x[0] = 1*, you will find that the objects' id's have not changed, they both point to the same object, but the data stored in the object has changed.

Some data types correspond to *immutable* objects that, once created, cannot be changed. Integers, floats, and strings are immutable:

```
>>> s = "This is a string"

>>> s[0]
'T'

>>> s[0] = 'b'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

You can index into a string, but you can't change a character in the string. The only way to change *s* is to redefine it as a new string (which will be stored in a **new object**):

```
>>> id(s)
1850368

>>> s = "New string"
>>> id(s)
1850128
```

What happened to the old object? It depends on whether any other variable was pointing to it. If not, as in the example above, then Python's *garbage collection* would recognize it's no longer needed and free up the memory for other uses. But if any other variable is still pointing to it, the object will still exist, e.g.

```
>>> s2 = s
>>> id(s2)                      # same object as s above
```

```
1850128

>>> s = "Yet another string"    # creates a new object
>>> id(s)                       # s now points to new object
1813104

>>> id(s2)                      # s2 still points to the old one
1850128

>>> s2
'New string'
```

4.1.13 Tuples

We have seen that lists are mutable. For some purposes we need something like a list but that is immuatable (e.g. for dictionary keys, see below). A tuple is like a list but defined with parentheses *(..)* rather than square brackets *[..]*:

```
>>> t = (4,5,6)

>>> t[0]
4

>>> t[0] = 9
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

4.1.14 Iterators

We often want to iterate over a set of things. In Python there are many ways to do this, and it often takes the form:

```
>>> for A in B:
...     # do something, probably involving the current A
```

In this construct *B* is any Python object that is *iterable*, meaning it has a built-in way (when B's class was defined) of starting with one thing in *B* and progressing through the contents of *B* in some hopefully logical order.

Lists and tuples are iterable in the obvious way: we step through it one element at a time starting at the beginning:

```
>>> for i in [3, 7, 'b']:
...     print "i is now ", i
...
i is now  3
i is now  7
i is now  b
```

4.1.15 range

In numerical work we often want to have i start at 0 and go up to some number N, stepping by one. We obviously don't want to have to construct the list [0, 1, 2, 3, ..., N] by typing all the numbers when *N* is large, so Python has a way of doing this:

```
>>> range(7)
[0, 1, 2, 3, 4, 5, 6]
```

NOTE: The last element is 6, not 7. The list has 7 elements but starts by default at 0, just as Python indexing does. This makes it convenient for doing things like:

```
>>> L = ['a', 8, 12]
>>> for i in range(len(L)):
...     print "i = ", i, "  L[i] = ", L[i]
...
i =  0   L[i] =  a
i =  1   L[i] =  8
i =  2   L[i] =  12
```

Note that *len(L)* returns the length of the list, so *range(len(L))* is always a list of all the valid indices for the list *L*.

4.1.16 enumerate

Another way to do this is:

```
>>> for i,value in enumerate(L):
...     print "i = ",i, "  L[i] = ",value
...
i =  0   L[i] =  a
i =  1   L[i] =  8
i =  2   L[i] =  12
```

range can be used with more arguments, for example if you want to start at 2 and step by 3 up to 20:

```
>>> range(2,20,3)
[2, 5, 8, 11, 14, 17]
```

Note that this doesn't go up to 20. Just like *range(7)* stops at 6, this list stops one item short of what you might expect.

NumPy has a *linspace* command that behaves like Matlab's, which is sometimes more useful in numerical work, e.g.:

```
>>> np.linspace(2,20,7)
array([  2.,   5.,   8.,  11.,  14.,  17.,  20.])
```

This returns a NumPy array with 7 equally spaced points between 2 and 20, including the endpoints. Note that the elements are floats, not integers. You could use this as an iterator too.

If you plan to iterate over a lot of values, say 1 million, it may be inefficient to generate a list object with 1 million elements using *range*. So there is another option called *xrange*, that does the iteration you want without explicitly creating and storing the list:

```
for i in xrange(1000000):
    # do something
```

does what we want.

Note that the elements in a list you're iterating on need not be numbers. For example, the sample module *myfcns* in $UWHPSC/codes/python defines two functions *f1* and *f2*. If we want to evaluate each of them at x=3., we could do:

```
>>> from myfcns import f1, f2
>>> type(f1)
<type 'function'>

>>> for f in [f1, f2]:
```

```
...     print f(3.)
...
5.0
162754.791419
```

This can be very handy if you want to perform some tests for a set of test functions.

4.1.17 Further reading

See the *Python:* section of the *Bibliography and further reading*.

In particular, see the *[Python-2.5-tutorial]* or *[Python-2.7-tutorial]* for good overviews (these two versions of Python are very similar).

There are several introductory Python pages at the *[software-carpentry]* site.

For more on basic data structures: http://docs.python.org/2/tutorial/datastructures.html

4.2 Python scripts and modules

A Python script is a collection of commands in a file designed to be executed like a program. The file can of course contain functions and import various modules, but the idea is that it will be run or executed from the command line or from within a Python interactive shell to perform a specific task. Often a script first contains a set of function definitions and then has the *main program* that might call the functions.

Consider this script, found in $UWHPSC/python/script1.py:

script1.py

```
"""
$UWHPSC/codes/python/script1.py

Sample script to print values of a function at a few points.
"""
import numpy as np

def f(x):
    """
    A quadratic function.
    """
    y = x**2 + 1.
    return y

print "     x         f(x)"
for x in np.linspace(0,4,3):
    print "%8.3f   %8.3f" % (x, f(x))

```

The *main program* starts with the print statement.

There are several ways to run a script contained in a file.

At the Unix prompt:

```
$ python script1.py
    x        f(x)
  0.000     1.000
```

```
  2.000     5.000
  4.000    17.000
```

From within Python:

```
>>> execfile("script1.py")
[same output as above]
```

From within IPython, using either *execfile* as above, or *run*:

```
In [48]: run script1.py
[same output as above]
```

Or, you can *import* the file as a module (see *Importing modules* below for more about this):

```
>>> import script1
     x         f(x)
   0.000     1.000
   2.000     5.000
   4.000    17.000
```

Note that this also gives the same output. Whenever a module is imported, any statements that are in the main body of the module are executed when it is imported. In addition, any variables or functions defined in the file are available as attributes of the module, e.g.,

```
>>> script1.f(4)
17.0

>>> script1.np
<module 'numpy' from
'/Library/Python/2.5/site-packages/numpy-1.4.0.dev7064-py2.5-macosx-10.3-fat.egg/numpy/__init__.pyc'>
```

Note there are some differences between executing the script and importing it. When it is executed as a script, it is as if the commands were typed at the command line. Hence:

```
>>> execfile('script1.py')
     x         f(x)
   0.000     1.000
   2.000     5.000
   4.000    17.000

>>> f
<function f at 0x1c0430>

>>> np
<module 'numpy' from
'/Library/Python/2.5/site-packages/numpy-1.4.0.dev7064-py2.5-macosx-10.3-fat.egg/numpy/__init__.pyc'>
```

In this case *f* and *np* are in the namespace of the interactive session as if we had defined them at the prompt.

4.2.1 Writing scripts for ease of importing

The script used above as an example contains a function *f(x)* that we might want to be able to import without necessarily running the *main program*. This can be arranged by modifying the script as follows:

script2.py

```
"""
$UWHPSC/codes/python/script2.py

Sample script to print values of a function at a few points.
The printing is only done if the file is executed as a script, not if it is
imported as a module.
"""
import numpy as np

def f(x):
    """
    A quadratic function.
    """
    y = x**2 + 1.
    return y

def print_table():
    print "      x         f(x)"
    for x in np.linspace(0,4,3):
        print "%8.3f  %8.3f" % (x, f(x))

if __name__ == "__main__":
    print_table()
```

When a file is imported or executed, an attribute *__name__* is automatically set, and has the value *__main__* only if the file is executed as a script, not if it is imported as a module. So we see the following behavior:

```
$ python script2.py
    x        f(x)
  0.000     1.000
  2.000     5.000
  4.000    17.000
```

as with *script1.py*, but:

```
>>> import script2            # does not print table

>>> script2.__name__
'script2'                     # not '__main__'

>>> script2.f(4)
17.0

>>> script2.print_table()
     x        f(x)
   0.000     1.000
   2.000     5.000
   4.000    17.000
```

4.2.2 Reloading modules

When you import a module, Python keeps track of the fact that it is imported and if it encounters another statement to import the same module will not bother to do so again (the list of modules already import is in *sys.modules*). This is convenient since loading a module can be time consuming. So if you're debugging a script using *execfile* or *run* from an IPython shell, each time you change it and then re-execute it will not reload *numpy*, for example.

Sometimes, however, you want to force reloading of a module, in particular if it has changed (e.g. when we are debugging it).

Suppose, for example, that we modify *script2.py* so that the quadratic function is changed from *y = x**2 + 1 ' to 'y = x**2 + 10*. If we make this change and then try the following (in the same Python session as above, where *script2* was already imported as a module):

```
>>> import script2

>>> script2.print_table()
     x        f(x)
   0.000     1.000
   2.000     5.000
   4.000    17.000
```

we get the same results as above, even though we changed *script2.py*.

We have to use the *reload* command to see the change we want:

```
>>> reload(script2)
<module 'script2' from 'script2.py'>

>>> script2.print_table()
     x        f(x)
   0.000    10.000
   2.000    14.000
   4.000    26.000
```

4.2.3 Command line arguments

We might want to make this script a bit fancier by adding an optional argument to the *print_table* function to print a different number of points, rather than the 3 points shown above.

The next version has this change, and also has a modified version of the main program that allows the user to specify this value *n* as a command line argument:

script3.py

```
"""
$UWHPSC/codes/python/script3.py

Modification of script2.py that allows a command line argument telling how
many points to plot in the table.

Usage example: To print table with 5 values:
   python script3 5

"""
import numpy as np

def f(x):
    """
    A quadratic function.
    """
    y = x**2 + 1.
    return y

def print_table(n=3):
    print "     x        f(x)"
    for x in np.linspace(0,4,n):
        print "%8.3f  %8.3f" % (x, f(x))

```

```
if __name__ == "__main__":
    """
    What to do if the script is executed at command line.
    Note that sys.argv is a list of the tokens typed at the command line.
    """
    import sys
    print "sys.argv is ",sys.argv
    if len(sys.argv) > 1:
        try:
            n = int(sys.argv[1])
            print_table(n)
        except:
            print "*** Error: expect an integer n as the argument"
    else:
        print_table()
```

Note that:

- The function *sys.argv* from the *sys* module returns the arguments that were present if the script is executed from the command line. It is a list of strings, with *sys.argv[0]* being the name of the script itself, *sys.argv[1]* being the next thing on the line, etc. (if there were more than one command line argument, separated by spaces).
- We use *int(sys.argv[1])* to convert the argument, if present, from a string to an integer.
- We put this conversion in a try-except block in case the user gives an invalid argument.

Sample output:

```
$ python script3.py
     x        f(x)
   0.000     1.000
   2.000     5.000
   4.000    17.000

$ python script3.py 5
     x        f(x)
   0.000     1.000
   1.000     2.000
   2.000     5.000
   3.000    10.000
   4.000    17.000

$ python script3.py 5.2
*** Error: expect an integer n as the argument
```

4.2.4 Importing modules

When Python starts up there are a certain number of basic commands defined along with the general syntax of the language, but most useful things needed for specific purposes (such as working with webpages, or solving linear systems) are in *modules* that do not load by default. Otherwise it would take forever to start up Python, loading lots of things you don't plan to use. So when you start using Python, either interactively or at the top of a script, often the first thing you do is *import* one or more modules.

A Python module is often defined simply by grouping a set of parameters and functions together in a single .py file. See *Python scripts and modules* for some examples.

Two useful modules are *os* and *sys* that help you interact with the operating system and the Python system that is running. These are standard modules that should be available with any Python implementation, so you should be able

to import them at the Python prompt:

```
>>> import os, sys
```

Each module contains many different functions and parameters which are the *methods* and *attributes* of the module. Here we will only use a couple of these. The *getcwd* method of the os module is called to return the "current working directory" (the same thing *pwd* prints in Unix), e.g.:

```
>>> os.getcwd()
/home/uwhpsc/uwhpsc/codes/python
```

Note that this function is called with no arguments, but you need the open and close parens. If you type "os.getcwd" without these, Python will instead print what type of object this function is:

```
>>> os.getcwd
<built-in function getcwd>
```

4.2.5 The Python Path

The *sys* module has an attribute *sys.path*, a variable that is set by default to the search path for modules. Whenever you perform an *import*, this is the set of directories that Python searches through looking for a file by that name (with a .py extension). If you print this, you will see a list of strings, each one of which is the full path to some directory. Sometimes the first thing in this list is the empty string, which means "the current directory", so it looks for a module in your working directory first and if it doesn't find it, searches through the other directories in order:

```
>>> print sys.path
['', '/usr/lib/python2.7', ....]
```

If you try to import a module and it doesn't find a file with this name on the path, then you will get an import error:

```
>>> import junkname
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named junkname
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named junkname
```

When new Python software such as NumPy or SciPy is installed, the installation script should modify the path appropriately so it can be found. You can also add to the path if you have your own directory that you want Python to look in, e.g.:

```
>>> sys.path.append("/home/uwhpsc/mypython")
```

will append the directory indicated to the path. To avoid having to do this each time you start Python, you can set a Unix environment variable that is used to modify the path every time Python is started. First print out the current value of this variable:

```
$ echo $PYTHONPATH
```

It will probably be blank unless you've set this before or have installed software that sets this automatically. To append the above example directory to this path:

```
$ export PYTHONPATH=$PYTHONPATH:/home/uwhpsc/mypython
```

This appends another directory to the search path already specified (if any). You can repeat this multiple times to add more directories, or put something like:

```
export PYTHONPATH=$PYTHONPATH:dir1:dir2:dir3
```

in your *.bashrc* file if there are the only 3 personal directories you always want to search.

4.2.6 Other forms of import

If all we want to use from the *os* module is *getcwd*, then another option is to do:

```
>>> from os import getcwd
>>> getcwd()
'/Users/rjl/uwhpsc/codes/python'
```

In this case we only imported one method from the module, not the whole thing. Note that now *getcwd* is called by just giving the name of the method, not *module.method*. The name *getcwd* is now in our *namespace*. If we only imported *getcwd* and tried typing "os.getcwd()" we'd get an error, since it wouldn't find *os* in our namespace.

You can rename things when you import them, which is sometimes useful if different modules contain different objects with the same name. For example, to compare how the *sqrt* function in the standard Python math module compares to the numpy version:

```
>>> from math import sqrt as sqrtm
>>> from numpy import sqrt as sqrtn

>>> sqrtm(-1.)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error

>>> sqrtn(-1.)
nan
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
```

The standard function gives an error whereas the *numpy* version returns *nan*, a special *numpy* object representing "Not a Number".

You can also import a module and give it a different name locally. This is particularly useful if you import a module with a long name, but even for *numpy* many examples you'll find on the web abbreviate this as *np* (see *Numerics in Python*):

```
>>> import numpy as np
>>> theta = np.linspace(0., 2*np.pi, 5)
>>> theta
array([ 0.        ,  1.57079633,  3.14159265,  4.71238898,  6.28318531])

>>> np.cos(theta)
array([  1.00000000e+00,   6.12323400e-17,  -1.00000000e+00, -1.83697020e-16,   1.00000000e+00])
```

If you don't like having to type the module name repeatedly you can import just the things you need into your namespace:

```
>>> from numpy import pi, linspace, cos
>>> theta = linspace(0., 2*pi, 5)
>>> theta
array([ 0.        ,  1.57079633,  3.14159265,  4.71238898,  6.28318531])
>>> cos(theta)
array([  1.00000000e+00,   6.12323400e-17,  -1.00000000e+00, -1.83697020e-16,   1.00000000e+00])
```

If you're going to be using lots of things form *numpy* you might want to import everything into your namespace:

```
>>> from numpy import *
```

Then *linspace*, *pi*, *cos*, and several hundred other things will be available without the prefix.

When writing code it is often best to not do this, however, since then it is not clear to the reader (or even to the programmer sometimes) what methods or attributes are coming from which module if several different modules are being used. (They may define methods with the same names but that do very different things, for example.)

When using IPython, it is often convenient to start it with:

```
$ ipython --pylab
```

This automatically imports everything from *numpy* into the namespace, and also all of the plotting tools from *matplotlib*.

4.2.7 Further reading

Modules section of Python documentation

4.3 Python functions

Functions are easily defined in Python using *def*, for example:

```
>>> def myfcn(x):
...     import numpy as np
...     y = np.cos(x) * np.exp(x)
...     return y
...

>>> myfcn(0.)
1.0

>>> myfcn(1.)
1.4686939399158851
```

As elsewhere in Python, there is no begin-end notation except the indentation. If you are defining a function at the command line as above, you need to input a blank line to indicate that you are done typing in the function.

4.3.1 Defining functions in modules

Except for very simple functions, you do not want to type it in at the command line in Python. Normally you want to create a text file containing your function and import the resulting module into your interactive session.

If you have a file named *myfile.py* for example that contains:

```
def myfcn(x):
    import numpy as np
    y = np.cos(x) * np.exp(x)
    return y
```

and this file is in your Python search path (see python_path), then you can do:

```
>>> from myfile import myfcn
>>> myfcn(0.)
1.0
>>> myfcn(1.)
1.4686939399158851
```

In Python a function is an object that can be manipulated like any other object.

4.3.2 Lambda functions

Some functions can be easily defined in a single line of code, and it is sometimes useful to be able to define a function "on the fly" using "lambda" notation. To define a function that returns 2*x for any input x, rather than:

```
def f(x):
    return 2*x
```

we could also define f via:

```
f = lambda x: 2*x
```

You can also define functions of more than one variable, e.g.:

```
g = lambda x,y: 2*(x+y)
```

4.3.3 Further reading

4.4 Python strings

See this List of methods applicable to strings

4.4.1 String formatting

Often you want to construct a string that incorporates the values of some variables. This can be done using the form *format % values* where *format* is a string that describes the desired format and *values* is a single value or tuple of values that go into various slots in the format.

See String Formatting Operations

This is best learned from some examples:

```
>>> x = 45.6
>>> s = "The value of x is %s"  % x
>>> s
'The value of x is 45.6'
```

The *%s* in the format string means to convert x to a string and insert into the format. It will use as few spaces as possible.

```
>>> s = "The value of x is %21.14e"  % x
>>> s
'The value of x is  4.56000000000000e+01'
```

In the case above, exponential notation is used with 14 digits to the right of the decimal point, put into a field of 21 digits total. (You need at least 7 extra characters to leave room for a possible minus sign as well as the first digit, the decimal point, and the exponent such as *e+01*.

```
>>> y = -0.324876
>>> s = "Now x is %8.3f and y is %8.3f" % (x,y)
>>> s
'Now x is   45.600 and y is   -0.325'
```

In this example, fixed notation is used instead of scientific notation, with 3 digits to the right of the decimal point, in a field 8 characters wide. Note that *y* has been rounded.

In the last example, two variables are inserted into the format string.

4.4.2 Further reading

See also:

- http://docs.python.org/tutorial/inputoutput.html
- List of methods applicable to strings
- Input and Output documentation

4.5 Numerics in Python

Python is a general programming language and is used for many purposes that have nothing to do with scientific computing or numerical methods. However, there are a number of modules that can be imported that provide a variety of numerical methods and other tools that are very useful for scientific computing.

The basic module needed for most numerical work is *NumPy*, which provides in particular the data structures needed for working with arrays of real numbers representing vectors or matrices. The module *SciPy* provides additional numerical methods and tools.

If you know Matlab, you will find that many of the things you are used to doing in that language can be done using *NumPy* and *SciPy*, although the syntax is often a bit different. Matlab users will find web page *[NumPy-for-Matlab-Users]* crucial for understanding the differences and transitioning to Python, and this page is useful for all new users. A tutorial can be found at *[NumPy-tutorial]*.

4.5.1 Vectors and Matrices

Python has lists as a built-in data type, e.g.:

```
>>> x = [1., 2., 3.]
```

defines a list that contains 3 real numbers and might be viewed as a vector. However, you cannot easily do arithmetic on such lists the way you can with vectors in Matlab, e.g. 2*x does not give [2., 4., 6.] as you might hope:

```
>>> 2*x
[1.0, 2.0, 3.0, 1.0, 2.0, 3.0]
```

instead it doubles the length of *x*, and *x+x* would give the same thing. You also cannot apply *sqrt* to a list to get a new list containing the square root of each element, for example.

Two-dimensional arrays are also a bit clumsy in Python, as they have to be specified as a list of lists, e.g. a 3x2 array with the elements 11,12 in the first row, 21,22 in the second row, 31,32 in the third row would be specified by:

```
>>> A = [[11, 12], [21, 22], [31, 32]]
```

Note that indexing always starts with 0 in Python, so we find for example that:

```
>>> A[0]
[11, 12]

>>> A[1]
[21, 22]

>>> A[1][0]
21
```

Here A[0] refers to the 0-index element of A, which is itself a list [11, 12]. and A[1][0] can be understood as the 0-index element of A[1] = [21, 22].

You cannot work with A as a matrix, for example to multiply it by a vector, except by writing code that loops over the elements explicitly.

NumPy was developed to make it easy to do the sorts of things we want to do with matrices and vectors, and more generally n-dimensional arrays of real numbers.

For example:

```
>>> import numpy as np
>>> x = np.array([1., 2., 3.])
>>> x
array([ 1.,  2.,  3.])

>>> 2*x
array([ 2.,  4.,  6.])

>>> np.sqrt(x)
array([ 1.        ,  1.41421356,  1.73205081])
```

We see that we can multiply by a scalar or take component-wise square roots.

You may find it ugly to have to start numpy command with np., as necessary here since we imported numpy as np. Instead you could do:

```
>>> from numpy import *
```

and then just use *sqrt*, for example, and you will get the NumPy version. But in these notes and many Python examples you'll see, the module is explicitly listed so it is clear where a function is coming from.

For matrices, we can convert our list of lists into a *NumPy* array as follows (we specify *dtype=float* to make sure the elements of A are stored as floating point real number even though we type them here as integers):

```
>>> A = np.array([[11, 12], [21, 22], [31, 32]], dtype=float)
>>> A
array([[ 11.,  12.],
       [ 21.,  22.],
       [ 31.,  32.]])

>>> A[0,1]
12.
```

Note that we can now index into the array as in matrix notation A[0,1] (remembering that indexing starts at 0 in Python), so this the [0,1] element of A means the first row and second column.

We can also do slicing operations, extracting a single row or column:

```
>>> A[0,:]
array([11., 12.])
```

```
>>> A[:,0]
array([11., 21., 31.])
```

Since A is a NumPy array object, there are certain methods automatically defined on A, such as the transpose:

```
>>> A.T
array([[11., 21., 31.],
       [12., 22., 32.]])
```

Seeing all the methods defined for A is easy if you use the IPython shell (see ipython), just type A. followed by the tab key (you will find there are 155 methods defined).

We can do matrix-vector or matrix-matrix multiplication using the NumPy dot function:

```
>>> np.dot(A.T, x)
array([ 146.,  152.])

>>> np.dot(A.T, A)
array([[1523., 1586.],
       [1586., 1652.]])
```

This looks somewhat less mathematical than Matlab notation A'*A, but the syntax and data structures of Matlab were designed specifically for linear algebra, whereas Python is a more general language and so doing linear algebra has to be done in this framework.

Note that elements of a *NumPy* array are always all of the same type, and generally we want *floats*, though integer arrays can also be defined. This is different than Python lists, which can contain elements with different types, e.g.:

```
>>> L = [2, 3., 'xyz', [4,5]]

>>> print type(L[0]), type(L[1]), type(L[2]), type(L[3])
<type 'int'> <type 'float'> <type 'str'> <type 'list'>
```

4.5.2 Component-wise operations

One thing to watch out for if you are used to Matlab notation: In Matlab some operations (such as sqrt, sin, cos, exp, etc) can be applied to vectors or matrices and will automatically be applied component-wise. Other operations like * and / (multiplication and division) attempt to do things in terms of linear algebra, and so in Matlab, A*B gives the matrix product and only makes sense if the number of columns of A agrees with the number of rows of B. If you want a component-wise product of A and B you must use .* instead, with a period before the *.

In NumPy, * and / are applied component-wise, like any other operation. To get a matrix-product you must use *dot*:

```
>>> A = np.array([[1,2], [3,4]])
>>> B = np.array([[5,0], [0,7]])
>>> A
array([[1, 2],
       [3, 4]])
>>> B
array([[5, 0],
       [0, 7]])

>>> A*B
array([[ 5,  0],
       [ 0, 28]])

>>> np.dot(A,B)
```

```
array([[ 5, 14],
       [15, 28]])
```

Many other linear algebra tools can be found in *NumPy*. For example, to solve a linear system Ax = b using Gaussian Elimination, we can do:

```
>>> A
array([[1, 2],
       [3, 4]])

>>> b = np.array([2,3])
>>> x = np.linalg.solve(A,b)

>>> x
array([-1. ,  1.5])
```

To find the eigenvalues and eigenvectors of A:

```
>>> evals, evecs = np.linalg.eig(A)

>>> evals
array([-0.37228132,  5.37228132])

>>> evecs
array([[-0.82456484, -0.41597356],
       [ 0.56576746, -0.90937671]])
```

Note: You may be tempted to use the variable name *lambda* for the eigenvalues of a matrix, but this isn't allowed in Python because *lambda* is a keyword of the language, see *Lambda functions*.

4.5.3 Further reading

Be sure to visit

- http://www.scipy.org/Tentative_NumPy_Tutorial
- http://www.scipy.org/NumPy_for_Matlab_Users

See also *[NumPy-pros-cons]* for more about differences with other mathematical languages.

4.6 IPython_notebook

The IPython notebook is fairly new and changing rapidly. The version originally installed in the class VM is version 0.10. To get the latest development version, which has some nicer features, do the following:

```
$ cd
$ git clone https://github.com/ipython/ipython.git
$ cd ipython
$ sudo python setup.py install
```

Then start the notebook via:

```
$ ipython notebook --pylab inline
```

in order to have the plots appear inline. If you leave off this argument, a new window will be opened for each plot.

Read more about the notebook in the documentation

See some cool examples in the IPython notebook viewer.

See also *Sage*.

4.6.1 Interactive notebooks

IPython 2.0 (released April 1, 2014) includes interactive widgets

See these Tips for installing IPython 2.0 on your own computer.

SageMathCloud does not yet have IPython 2.0 and for various technical reasons will not have it for a while.

SageMathCloud does now have mpld3, which allows zooming in on plots. For a demo, see *Lab 13: Tuesday May 13, 2014* or Jake's blog post on mpld3

In addition to *mpld3*, Jake Vanderplass has also developed:

- ipywidgets similar to the 2.0 widgets in some ways but persistent also if the "static" notebook is viewed via nbviewer. See Jake's blog post on ipywidgets.
- JSAnimation for persistent animations. This is used for example for all the animations in the Clawpack galleries. (Clawpack is an open source software package developed by Randy LeVeque and many others for solving hyperbolic partial differential equations.)

4.7 Sage

Sage is an open source collection of mathematical software with a common Python interface. The lead developer is William Stein, a number theorist in the Mathematics Department at the University of Washington.

The Sage notebook was an original model for the *IPython_notebook*.

You can try Sage online by typing into a cell at https://sagecell.sagemath.org/. You can type straight Python as well as using the enhanced capabilities of Sage.

See:

- http://interact.sagemath.org/top-rated-posts
- http://trac.sagemath.org/sage_trac/

for some online sage examples.

4.8 Plotting with Python

4.8.1 matplotlib and pylab

There are nice tools for making plots of 1d and 2d data (curves, contour plots, etc.) in the module matplotlib. Many of these plot commands are very similar to those in Matlab.

To see some of what's possible (and learn how to do it), visit the matplotlib gallery. Clicking on a figure displays the Python commands needed to create it.

The best way to get matplotlib working interactively in a standard Python shell is to do:

```
$ python
>>> import pylab
>>> pylab.interactive(True)
```

pylab includes not only *matplotlib* but also *numpy*. Then you should be able to do:

```
>>> x = pylab.linspace(-1, 1, 20)
>>> pylab.plot(x, x**2, 'o-')
```

and see a plot of a parabola appear. You should also be able to use the buttons at the bottom of the window, e.g click the magnifying glass and then use the mouse to select a rectangle in the plot to zoom in.

Alternatively, you could do:

```
>>> from pylab import *
>>> interactive(True)
```

With this approach you don't need to start every pylab function name with pylab, e.g.:

```
>>> x = linspace(-1, 1, 20)
>>> plot(x, x**2, 'o-')
```

In these notes we'll generally use module names just so it's clear where things come from.

If you use the IPython shell, you can do:

```
$ ipython --pylab

In [1]: x = linspace(-1, 1, 20)
In [2]: plot(x, x**2, 'o-')
```

The –pylab flag causes everything to be imported from pylab and set up for interactive plotting.

4.8.2 Mayavi and mlab

Mayavi is a Python plotting package designed primarily for 3d plots. See:

- Documentation
- Gallery

See *Downloading and installing software for this class* for some ways to install Mayavi.

4.8.3 VisIt

VisIt is an open source visualization package being developed at Lawrence Livermore National Laboratory. It is designed for industrial-strength visualization problems and can deal with very large distributed data sets using MPI.

There is a GUI interface and also a Python interface for scripting.

See:

- Documentation
- Gallery
- Tutorial

4.8.4 ParaView

ParaView is another open source package developed originally for work at the National Labs.

There is a GUI interface and also a Python interface for scripting.

See:

- Documentation
- Gallery

4.9 Python debugging

For some general tips on writing and debugging programs in any language, see debugging.

4.9.1 Python IDEs

An IDE (Integrated Development Environment) generally provides an editor and debugger that are linked directly to the language. See

- http://en.wikipedia.org/wiki/Integrated_development_environment.

Python has a IDE called IDLE that provides an editor that has some debugger features. You might want to explore this, see

- http://docs.python.org/library/idle.html.

Other IDEs also provided Python interfaces, such as Eclipse. See, e.g.,

- http://en.wikipedia.org/wiki/Eclipse_(software)
- http://www.vogella.de/articles/Python/article.html
- http://heather.cs.ucdavis.edu/~matloff/eclipse.html

These environments generally provide an interface to *pdb*, the Python debugger described below.

4.9.2 Reloading modules

Note that if you are debugging a Python code by running it repeatedly in an interactive shell, you need to make sure it is seeing the most recent version of the code after you make editing changes. If you run it using *execfile* (or *run* in IPython), it should find the most recent version.

If you import it as a module, then you need to make sure you do a *reload* as described at *Reloading modules*.

4.9.3 Print statements

Print statements can be added almost anywhere in a Python code to print things out to the terminal window as it goes along.

You might want to put some special symbols in debugging statements to flag them as such, which makes it easier to see what output is your debug output and also makes it easier to find them again later to remove from the code, e.g. you might use "+++" or "DEBUG".

As an example, suppose you are trying to better understand Python namespaces and the difference between local and global variables. Then this code might be useful:

```
"""
$UWHPSC/codes/python/debugdemo1a.py

Debugging demo using print statements
"""

```

```
x = 3.
y = -22.

def f(z):
    x = z+10
    print "+++ in function f: x = %s, y = %s, z = %s" % (x,y,z)
    return x

print "+++ before calling f: x = %s, y = %s" % (x,y)
y = f(x)
print "+++ after calling f: x = %s, y = %s" % (x,y)

```

Here the print function in the definition of *f(x)* is being used for debugging purposes. Executing this code gives:

```
>>> execfile("debugdemo1a.py")
+++ before calling f: x = 3.0, y = -22.0
+++ in function f: x = 13.0, y = -22.0, z = 3.0
+++ after calling f: x = 3.0, y = 13.0
```

If you are printing large amounts you might want to write to a file rather than to the terminal, see python_io.

4.9.4 pdb debugger

Inserting print statements may work best in some situations, but it is often better to use a *debugger*. The Python debugger pdb is very easy to use, often even easier than inserting print statements and well worth learning. See the pdb documentation for more information.

You can insert *breakpoints* in your code where control should pass back to the user, at which point you can query the value of any variable, or step through the program line by line from this point on. For the above example we might do this as below:

```
"""
$UWHPSC/codes/python/debugdemo1b.py

Debugging demo using pdb.
"""

x = 3.
y = -22.

def f(z):
    x = z+10
    import pdb; pdb.set_trace()
    return x

y = f(x)

print "x = ",x
print "y = ",y

```

Of course one could set multiple breakpoints with other *pdb.set_trace()* commands.

Now we get the prompt for the *pdb* shell when we hit the breakpoint:

```
>>> execfile("debugdemo1b.py")
```

```
> /Users/rjl/uwhpsc/codes/python/debugdemo1b.py(11)f()
-> return x

(Pdb) p x
13.0
(Pdb) p y
-22.0
(Pdb) p z
3.0
```

Note that *p* is short for *print*. You could also type *print x* but this would then execute the Python print command instead of the debugger command (though in this case it would print the same thing).

There are many other *pdb* commands, such as *next* to execute the next line, *continue* to continue executing until the next breakpoint, etc. See the documentation for more details.

For example, lets execute the next two statements and then print *x* and *y*:

```
(Pdb) n
--Return--
> /Users/rjl/uwhpsc/codes/python/debugdemo1b.py(11)f()->13.0
-> return x

(Pdb) n
> /Users/rjl/uwhpsc/codes/python/debugdemo1b.py(15)<module>()
-> print "x = ",x

(Pdb) p x,y
(3.0, 13.0)

(Pdb) p z
*** NameError: NameError("name 'z' is not defined",)

(Pdb) quit
>>>
```

You can also run the code as a script from the Unix prompt and again you will be put into the pdb shell when the breakpoint is reached:

```
$ python debugdemo1b.py
> /Users/rjl/uwhpsc/codes/python/debugdemo1b.py(11)f()
-> return x
(Pdb) p z
3.0
(Pdb) continue
x =  3.0
y =  13.0
```

4.9.5 Debugging after an exception occurs

Often code has bugs that cause an exception to be raised that causes the program to halt execution. Consider the following file:

If you change N to 20 it will run fine, but with *N = 40* we find:

```
>>> execfile("debugdemo2.py")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
  File "debugdemo2.py", line 14, in <module>
    w[i] = 1/eps2
ZeroDivisionError: float division
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "debugdemo2.py", line 14, in <module>
    w[i] = 1/eps2
ZeroDivisionError: float division
```

At some point *eps2* apparently has the value 0. To figure out when this happens, we could insert a *pdb.set_trace()* command in the loop and step through it until the error occurs and then look at *i*, but we can do so even more easily using a *post-mortem* analysis after it dies, using *pdb.pm()*:

```
>>> import pdb
>>> pdb.pm()
> /Users/rjl/uwhpsc/codes/python/debugdemo2.py(14)<module>()->None
-> w[i] = 1/eps2
(Pdb) p i
34
(Pdb) p eps2
0.0
(Pdb) p epsilon
5.9962169748381002e-17
```

This starts up *pdb* at exactly the point where the exception is about to occur. We see that the divide by zero happens when *i = 34* (because *epsilon* is so small that *1. + epsilon* is rounded off to *1.* in the computer, see floats).

4.9.6 Using pdb from IPython

In IPython it's even easier to do this post-mortem analysis. Just type:

```
In [50]: pdb
Automatic pdb calling has been turned ON
```

and then *pdb* will be automatically invoked if an exception occurs:

```
In [51]: run debugdemo2.py
---------------------------------------------------------------------------

ZeroDivisionError: float division
> /Users/rjl/uwhpsc/codes/python/debugdemo2.py(14)<module>()
     13     eps2 = z - 1.          # expect eps2 == epsilon?
---> 14     w[i] = 1/eps2
     15

ipdb> p i
34
ipdb> q

In [52]:
```

Type *pdb* again to turn it off.

Note: *pdb*, like *run* is a *magic function* in IPython, an extension of the language itself, type *magic* at the IPython prompt for more info.

If these commands don't work, type *%magic* and read this documentation.

4.9.7 Other pdb commands

There are a number of other commands, see the references above.

4.10 Animation in Python

matplotlib has a package *animation* that can be used directly, see for example http://matplotlib.org/examples/animation/dynamic_image2.html or this blog post.

Nicer webpage animations (within IPython notebooks or as stand-alone movies) can be created using the package *JSAnimation* created by Jake Vanderplas. For an example see this rendered example or *Lab 15: Tuesday May 20, 2014*.

4.11 Installing JSAnimation

First clone it from Github:

```
$ cd $HOME
$ git clone https://github.com/jakevdp/JSAnimation.git
$ cd JSAnimation
```

On your own laptop or the VM, you can probably install it via:

```
$ python setup.py install
```

On SageMathCloud you don't have access to the system folder where it normally installs Python packages, but you can install it for use in one project only via:

```
$ sage -python setup.py install --user
```

Then you should be able to open Python or IPython and *import JSAnimation*

4.11.1 Alternative to installing

Rather than installing it as a package, you can just add the JSAnimation directory to the Python search path. For running it from scripts in a bash shell, you might want to add this line to your Or you can just add $HOME/JSAnimation to your PYTHONPATH environment variable:

```
$ export PYTHONPATH=$PYTHONPATH:$HOME/JSAnimation
```

This appends the path to the end of whatever path is already specified in this environment variable.

As a last resort, you can also modify the path from within a Python session:

```
>>> import os, sys
>>> HOME = os.environ['HOME']
>>> JSAnimation_path = os.path.join(HOME, 'JSAnimation')
>>> sys.path.append(JSAnimation_path)
```

4.11.2 JSAnimation_frametools

For animations of complex plots, it is sometimes easier to simply create the plot for each frame as usual using *matplotlib* and then save a *.png* file for each frame. These can be created and then combined to create an animation using some tools in *JSAnimation_frametools.py*, currently found in *Lab 15: Tuesday May 20, 2014* along with some demos.

4.11.3 Matplotlib issues

JSAnimation requires a recent version of *matplotlib*. (In particular, older Ubuntu versions may not have a recent version.) If you're having problems with *matplotlib* in this context, you might want to try using the Anaconda Python distribution, or switch to smc.

CHAPTER

FIVE

FORTRAN

5.1 Fortran

5.1.1 General References:

- See the *Fortran references* section of the bibliography for links.

5.1.2 History

FORTRAN stands for *FORmula TRANslator* and was the first major *high level language* to catch on. The first compiler was written in 1954-57. Before this, programmers generally had to write programs in assembly language.

Many version followed: Fortran II, III, IV. Fortran 66 followed a set of standards formulated in 1966.

See

- http://www.ibiblio.org/pub/languages/fortran/ch1-1.html
- http://en.wikipedia.org/wiki/Fortran

for brief histories.

5.1.3 Fortran 77

The standards established in 1977 lead to Fortran 77, or f77, and many codes are still in use that follow this standard.

Fortran 77 does not have all the features of newer versions and many things are done quite differently.

One feature of f77 is that lines of code have a very rigid structure. This was required in early versions of Fortran due to the fact that computer programs were written on *Punch cards*. All statements must start in column 7 or beyond and no statement may extend beyond column 72. The first 6 columns are used for things like labels (numbers associated with particular statements). In f77 any line that starts with a 'c' in column 1 is a comment.

We will not use f77 in this class but if you need to work with Fortran in the future you may need to learn more about it because of all the *legacy codes* that still use it.

5.1.4 Fortran 90/95

Dramatically new standards were introduced with Fortran 90, and these were improved in mostly minor ways in Fortran 95. There are newer Fortran 2003 and 2008 standards but few compilers implement these fully yet. See Wikipedia page on Fortran standards for more information.

For this class we will use the Fortran 90/95 standards, which we will refer to as Fortran 90 for brevity.

5.1.5 Compilers

Unlike Python code, a Fortran program must pass through several stages before being executed. There are several different compilers that can turn Fortran code into an *executable*, as described more below.

In this class we will use *gfortran*, which is an open source compiler, part of the GNU Project. See http://gcc.gnu.org/fortran/ for more about gfortran.

There is an older compiler in this suite called *g77* which compiles Fortran 77 code, but *gfortran* can also be used for Fortran 77 code and has replaced g77.

There are several commercial compilers which are better in some ways, in particular they sometimes do better optimization and produce faster running executables. They also may have better debugging tools built in. Some popular ones are the Intel and Portland Group compilers.

5.1.6 File extensions

For the gfortran compiler, fixed format code should have the *.f* while free format code has the extension *.f90* or *.f95*. We will use *.f90*.

5.1.7 Compiling, linking, and running a Fortran code

Suppose we have a Fortran file named *demo1.f90*, for example the program below. We can not run this directly the way we did a Python script. Instead it must be converted into *object code*, a version of the code that is in a machine language specific to the type of computer. This is done by the *compiler*.

Then a *linker* must be used to convert the object code into an *executable* that can actually be executed.

This is broken into two steps because often large programs are split into many different *.f90* files. Each one can be compiled into a separate *object file*, which by default has the same name but with a *.o* extension (for example, from *demo1.f90* the compiler would produce *demo1.o*). One may also want to call on *library routines* that have already been compiled and reside in some library. The linker combines all of these into a single executable.

For more details on the process, see for example:

- http://en.wikipedia.org/wiki/Compiler
- http://en.wikipedia.org/wiki/Linker_%28computing%29

For the simplest case of a self-contained program in one file, we can combine both stages in a single *gfortran* command, e.g.

```
$ gfortran demo1.f90
```

By default this will produce an *executable* named *a.out* for obscure historical reasons (it stands for *assembler output*, see wikipedia).

To run the code you would then type:

```
$ ./a.out
```

Note we type *./a.out* to indicate that we are executing *a.out* from the current directory. There is an environment variable *PATH* that contains your *search path*, the set of directories that are searched whenever you type a command name at the Unix prompt. Often this is set so that the current directory is the first place searched, in which case you could just type *a.out* instead of *./a.out*. However, it is generally considered bad practice to include the current directory in your search path because bad things can happen if you accidentally execute a file.

If you don't like the name *a.out* you can specify an output name using the *-o* flag with the *gfortran* command. For example, if you like the Windows convention of using the extension *.exe* for executable files:

```
$ gfortran demo1.f90 -o demo1.exe
$ ./demo1.exe
```

will also run the code.

Note that if you try one of the above commands, there will be no file *demo1.o* created. By default *gfortran* removes this file once the executed is created.

Later we will see that it is often useful to split up the compile and link steps, particularly if there are several files that need to be compiled and linked. We can do this using the *-c* flag to compile without linking:

```
$ gfortran -c demo1.f90              # produces demo1.o
$ gfortran demo1.o -o demo1.exe      # produces demo1.exe
```

There are many other compiler flags that can be used, see linux man page for gfortran for a list.

5.1.8 Sample codes

The first example simply assigns some numbers to variables and then prints them out. The comments below the code explain some features.

```
1  ! $UWHPSC/codes/fortran/demo1.f90
2
3  program demo1
4
5  ! Fortran 90 program illustrating data types.
6
7  implicit none   ! to give error if a variable not declared
8  real :: x
9  real (kind=8) :: y, z
10 integer :: m
11
12 m = 3
13 print *, " "
14 print *, "M = ",M   ! note that M==m  (case insensitive)
15
16
17 print *, " "
18 print *, "x is real (kind=4)"
19 x = 1.e0 + 1.23456789e-6
20 print *, "x = ", x
21
22
23 print *, " "
24 print *, "y is real (kind=8)"
25 print *, "  but 1.e0 is real (kind=4):"
26 y = 1.e0 + 1.23456789e-6
27 print *, "y = ", y
28
29
30 print *, " "
31 print *, "z is real (kind=8)"
32 z = 1. + 1.23456789d-6
33 print *, "z = ", z
34
35 end program demo1
```

Comments:

- Exclamation points are used for comments
- The *implicit none* statement in line 7 means that any variable to be used must be explicitly declared. See fortran_implicit for more about this.
- Lines 8-10 declare four variables *x, y, z, n*. Note that *x* is declared to have type *real* which is a floating point number stored in 4 bytes, also known as *single precision*. This could have equivalently been written as:

```
    real (kind=4) :: x
```

  y and *z* are floating point numbers stored in 8 bytes (corresponding to *double precision* in older versions of Fortran). This is generally what you want to use.
- Fortran is not case-sensitive, so *M* and *m* refer to the same variable!!
- *1.23456789e-10* specifies a 4-byte real number. The 8-byte equivalent is *1.23456789d-10*, with a *d* instead of *e*. This is apparent from the output below.

Compiling and running this program produces:

```
$ gfortran demo1.f90 -o demo1.exe
$ ./demo1.exe

 M =            3

 x is real (kind=4)
 x =    1.000001

 y is real (kind=8)
   but 1.e0 is real (kind=4):
 y =    1.00000119209290

 z is real (kind=8)
 z =    1.00000123456789
```

For most of what we'll do in this class, we will use real numbers with *(kind=8)*. Be careful to specify constants using the *d* rather than *e* notation if you need to use scientific notation.

(But see *Default 8-byte real numbers* below for another approach.)

5.1.9 Intrinsic functions

There are a number of built-in functions that you can use in Fortran, for example the trig functions:

```
1  ! $UWHPSC/codes/fortran/builtinfcns.f90
2
3  program builtinfcns
4
5      implicit none
6      real (kind=8) :: pi, x, y
7
8      ! compute pi as arc-cosine of -1:
9      pi = acos(-1.d0)  ! need -1.d0 for full precision!
10
11     x = cos(pi)
12     y = sqrt(exp(log(pi)))**2
13
14     print *, "pi = ", pi
15     print *, "x = ", x
16     print *, "y = ", y
```

```
17
18 end program builtinfcns
```

This produces:

```
$ gfortran builtinfcns.f90
$ ./a.out
 pi =    3.14159265358979
 x =   -1.00000000000000
 y =    3.14159265358979
```

See http://www.nsc.liu.se/~boein/f77to90/a5.html for a good list of other intrinsic functions.

5.1.10 Default 8-byte real numbers

Note that you can declare variables to be real without appending *(kind=8)* if you compile programs with the gfortran flag *-fdefault-real-8*, e.g. if we modify the program above to:

```
1  ! $UWHPSC/codes/fortran/builtinfcns2.f90
2
3  program builtinfcns
4
5      implicit none
6      real :: pi, x, y    ! note kind is not specified
7
8      ! compute pi as arc-cosine of -1:
9      pi = acos(-1.0)
10
11     x = cos(pi)
12     y = sqrt(exp(log(pi)))**2
13
14     print *, "pi = ", pi
15     print *, "x = ", x
16     print *, "y = ", y
17
18 end program builtinfcns
```

Then:

```
$ gfortran builtinfcns2.f90
$ ./a.out
 pi =    3.141593
 x =   -1.000000
 y =    3.141593
```

gives single precision results, but we can obtain double precisions with:

```
$ gfortran -fdefault-real-8 builtinfcns2.f90
$ ./a.out
 pi =    3.14159265358979
 x =   -1.00000000000000
 y =    3.14159265358979
```

Note that if you plan to do this you might want to define a Unix alias, e.g.

```
$ alias gfort="gfortran -fdefault-real-8"
```

so you can just type:

```
$ gfort builtinfcns2.f90
$ ./a.out
 pi =    3.14159265358979
 x =   -1.00000000000000
 y =    3.14159265358979
```

Such an alias could be put in your *.bashrc file*.

We'll also see how to specify compiler flags easily in a makefile.

5.1.11 Fortran Arrays

Note that arrays are indexed starting at 1 by default, rather than 0 as in Python. Also note that components of an array are accessed using parentheses, not square brackets!

Arrays can be dimensioned and used as in the following example:

```
! $UWHPSC/codes/fortran/array1

program array1

    ! demonstrate declaring and using arrays

    implicit none
    integer, parameter :: m = 3, n=2
    real (kind=8), dimension(m,n) :: A
    real (kind=8), dimension(m) :: b
    real (kind=8), dimension(n) :: x
    integer :: i,j

    ! initialize matrix A and vector x:
    do j=1,n
        do i=1,m
            A(i,j) = i+j
            enddo
        x(j) = 1.
        enddo

    ! multiply A*x to get b:
    do i=1,m
        b(i) = 0.
        do j=1,n
            b(i) = b(i) + A(i,j)*x(j)
            enddo
        enddo

    print *, "A = "
    do i=1,m
        print *, A(i,:)    ! i'th row of A
        enddo
    print "(2d16.6)", ((A(i,j), j=1,2), i=1,3)
    print *, "x = "
    print "(d16.6)", x
    print *, "b = "
    print "(d16.6)", b

end program array1
```

Compiling and running this code gives the output:

```
A =
  2.00000000000000        3.00000000000000
  3.00000000000000        4.00000000000000
  4.00000000000000        5.00000000000000
x =
  1.00000000000000        1.00000000000000
b =
  5.00000000000000        7.00000000000000        9.00000000000000
```

Comments:

- In printing *A* we have used a *slice* operation: *A(i,:)* refers to the i'th row of *A*. In Fortran 90 there are many other array operations that can be done more easily than we have done in the loops above. We will investigate this further later.
- Here we set the values of *m,n* as integer parameters before declaring the arrays *A,x,b*. Being parameters means we can not change their values later in the program.
- It is possible to declare arrays and determine their size later, using *allocatable* arrays, which we will also see later.

There are many array operations you can do, for example:

```
! $UWHPSC/codes/fortran/vectorops.f90

program vectorops

    implicit none
    real(kind=8), dimension(3) :: x, y

    x = (/10.,20.,30./)
    y = (/100.,400.,900./)

    print *, "x = "
    print *, x

    print *, "x**2 + y = "
    print *, x**2 + y

    print *, "x*y = "
    print *, x*y

    print *, "sqrt(y) = "
    print *, sqrt(y)

    print *, "dot_product(x,y) = "
    print *, dot_product(x,y)


end program vectorops
```

produces:

```
x =
  10.0000000000000        20.0000000000000        30.0000000000000
x**2 + y =
  200.000000000000        800.000000000000        1800.00000000000
x*y =
  1000.00000000000        8000.00000000000        27000.0000000000
```

```
sqrt(y) =
  10.0000000000000        20.0000000000000        30.0000000000000
dot_product(x,y) =
  36000.0000000000
```

Note that addition, multiplication, exponentiation, and intrinsic functions such as *sqrt* all apply component-wise.

Multidimensional arrays can be manipulated in similar manner. The produce to two arrays when computed with * is always component-wise. For matrix multiplication, use *matmul*. There is also a *transpose* function:

```
! $UWHPSC/codes/fortran/arrayops.f90

program arrayops

    implicit none
    real(kind=8), dimension(3,2) :: a
    real(kind=8), dimension(2,3) :: b
    real(kind=8), dimension(3,3) :: c
    real(kind=8), dimension(2) :: x
    real(kind=8), dimension(3) :: y
    integer i

    a = reshape((/1,2,3,4,5,6/), (/3,2/))

    print *, "a = "
    do i=1,3
        print *, a(i,:)    ! i'th row
        enddo

    b = transpose(a)

    print *, "b = "
    do i=1,2
        print *, b(i,:)    ! i'th row
        enddo

    c = matmul(a,b)
    print *, "c = "
    do i=1,3
        print *, c(i,:)    ! i'th row
        enddo

    x = (/5,6/)
    y = matmul(a,x)
    print *, "x = ",x
    print *, "y = ",y

end program arrayops
```

produces:

```
a =
  1.00000000000000        4.00000000000000
  2.00000000000000        5.00000000000000
  3.00000000000000        6.00000000000000

b =
  1.00000000000000        2.00000000000000        3.00000000000000
  4.00000000000000        5.00000000000000        6.00000000000000
```

```
c =
  17.0000000000000        22.0000000000000        27.0000000000000
  22.0000000000000        29.0000000000000        36.0000000000000
  27.0000000000000        36.0000000000000        45.0000000000000

x =    5.00000000000000        6.00000000000000
y =    29.0000000000000        40.0000000000000        51.0000000000000
```

5.1.12 Loops

```
! $UWHPSC/codes/fortran/loops1.f90

program loops1

   implicit none
   integer :: i

   do i=1,3            ! prints 1,2,3
      print *, i
      enddo

   do i=5,11,2         ! prints 5,7,9,11
      print *, i
      enddo

   do i=6,2,-1         ! prints 6,5,4,3,2
      print *, i
      enddo

   i = 0
   do while (i < 5)    ! prints 0,1,2,3,4
      print *, i
      i = i+1
      enddo

end program loops1
```

The *while* statement used in the last example is considered obsolete. It is better to use a *do* loop with an *exit* statement if a condition is satisfied. The last loop could be rewritten as:

```
i = 0
do                  ! prints 0,1,2,3,4
   if (i>=5) exit
   print *, i
   i = i+1
   enddo
```

This form of the *do* is valid but is generally not a good idea. Like the while loop, this has the danger that a bug in the code may cause it to loop forever (e.g. if you typed *i = i-1* instead of *i = i+1*).

A better approach for loops of this form is to limit the number of iterations to some maximum value (chosen to be ample for your application), and then print a warning message, or take more drastic action, if this is exceeded, e.g.:

```
! $UWHPSC/codes/fortran/loops2.f90

program loops2

```

```
   implicit none
   integer :: i,j,jmax

   i = 0
   jmax = 100
   do j=1,jmax         ! prints 0,1,2,3,4
      if (i>=5) exit
      print *, i
      i = i+1
      enddo

   if (j==jmax+1) then
      print *, "Warning: jmax iterations reached."
      endif

end program loops2
```

Note: *j* is incremented *before* comparing to *jmax*.

5.1.13 if-then-else

```
! $UWHPSC/codes/fortran/ifelse1.f90

program ifelse1

    implicit none
    real(kind=8) :: x
    integer :: i

    i = 3
    if (i<2) then
        print *, "i is less than 2"
    else
        print *, "i is not less than 2"
    endif

    if (i<=2) then
        print *, "i is less or equal to 2"
    else if (i/=5) then
        print *, "i is greater than 2 but not equal to 5"
    else
        print *, "i is equal to 5"
    endif

end program ifelse1
```

Comments:

- The *else* clause is optional
- You can have optional *else if* clauses

There is also a one-line form of an *if* statement that was seen in a previous example on this page:

```
if (i>=5) exit
```

This is equivalent to:

```
if (i>=5) then
    exit
    endif
```

5.1.14 Booleans

- Compare with <, >, <=, >=, ==, /=. You can also use the older Fortran 77 style: *.lt., .gt., .le., .ge., .eq., .neq.*.
- Combine with *.and.* and *.or.*

For example:

```
((x>=1.0) .and. (x<=2.0)) .or. (x>5)
```

A boolean variable is declared with type *logical* in Fortran, as for example in the following code:

```
! $UWHPSC/codes/fortran/boolean1.f90

program boolean1

    implicit none
    integer :: i,k
    logical :: ever_zero

    ever_zero = .false.
    do i=1,10
        k = 3*i - 1
        ever_zero = (ever_zero .or. (k == 0))
        enddo

    if (ever_zero) then
        print *, "3*i - 1 takes the value 0 for some i"
    else
        print *, "3*i - 1 is never 0 for i tested"
    endif

end program boolean1
```

5.1.15 Further reading

- *Fortran subroutines and functions*
- *Fortran examples: Taylor series*
- Fortran Resources page

5.2 Useful gfortran flags

gfortran has many different command line options (also known as *flags*) that control what the compiler does and how it does it. To use these flags, simply include them on the command line when you run gfortran, e.g.:

```
$ gfortran -Wall -Wextra -c mysubroutine.f90 -o mysubroutine.o
```

If you find you use certain flags often, you can add them to an alias in your `.bashrc` file, such as:

```
alias gf="gfortran -Wall -Wextra -Wconversion -pedantic"
```

See the gfortran man page for more information. Note a "man page" is the Unix help manual documentation that is available for many Unix commands by typing, e.g.:

```
$ man gfortran
```

Warning: Different fortran compilers use different names for similar flags!

5.2.1 Output flags

These flags control what kind of output gfortran generates, and where that output goes.

- `-c`: Compile to an object file, rather than producing a standalone program. This flag is useful if your program source code is split into multiple files. The object files produced by this command can later be linked together into a complete program.
- `-o FILENAME`: Specifies the name of the output file. Without this flag, the default output file is `a.out` if compiling a complete program, or `SOURCEFILE.o` if compiling to an object file, where `SOURCEFILE.f90` is the name of the Fortran source file being compiled.

5.2.2 Warning flags

Warning flags tell gfortran to warn you about legal but potentially questionable sections of code. These sections of code may be correct, but warnings will often identify bugs before you even run your program.

- `-Wall`: Short for "warn about all," this flag tells gfortran to generate warnings about many common sources of bugs, such as having a subroutine or function with the same name as a built-in one, or passing the same variable as an `intent(in)` and an `intent(out)` argument of the same subroutine. In spite of its name, this does not turn all possible *-W* options on.
- `-Wextra`: In conjunction with `-Wall`, gives warnings about even more potential problems. In particular, `-Wextra` warns about subroutine arguments that are never used, which is almost always a bug.
- `-Wconversion`: Warns about implicit conversion. For example, if you want a double precision variable `sqrt2` to hold an accurate value for the square root of 2, you might write by accident `sqrt2 = sqrt(2.)`. Since `2.` is a single-precision value, the single-precision `sqrt` function will be used, and the value of `sqrt2` will not be as accurate as it could be. `-Wconversion` will generate a warning here, because the single-precision result of `sqrt` is implicitly converted into a double-precision value.
- `-pedantic`: Generate warnings about language features that are supported by gfortran but are not part of the official Fortran 95 standard. Useful if you want be sure your code will work with any Fortran 95 compiler.

5.2.3 Fortran dialect flags

Fortran dialect flags control how gfortran interprets your program, and whether various language extensions such as OpenMP are enabled.

- `-fopenmp`: Tells gfortran to compile using OpenMP. Without this flag, OpenMP directives in your code will be ignored.
- `-std=f95`: Enforces strict compliance with the Fortran 95 standard. This is like `-pedantic`, except it generates errors instead of warnings.

5.2.4 Debugging flags

Debugging flags tell the compiler to include information inside the compiled program that is useful in debugging, or alter the behavior of the program to help find bugs.

- `-g`: Generates extra debugging information usable by GDB. `-g3` includes even more debugging information.
- `-fbacktrace`: Specifies that if the program crashes, a backtrace should be produced if possible, showing what functions or subroutines were being called at the time of the error.
- `-fbounds-check`: Add a check that the array index is within the bounds of the array every time an array element is accessed. This substantially slows down a program using it, but is a very useful way to find bugs related to arrays; without this flag, an illegal array access will produce either a subtle error that might not become apparent until much later in the program, or will cause an immediate segmentation fault with very little information about cause of the error.
- `-ffpe-trap=zero,overflow,underflow` tells Fortran to *trap* the listed floating point errors (fpe). Having *zero* on the list means that if you divide by zero the code will die rather than setting the result to *+INFINITY* and continuing. Similarly, if *overflow* is on the list it will halt if you try to store a number larger than can be stored for the type of real number you are using because the exponent is too large.

  Trapping *underflow* will halt if you compute a number that is too small because the exponent is a very large negative number. For 8-byte floating point numbers, this happens if the number is smaller than approximate *1E-324*. If you don't trap underflows, such numbers will just be set to 0, which is generally the correct thing to do. But computing such small numbers may indicate a bug of some sort in the program, so you might want to trap them.

5.2.5 Optimization flags

Optimization options control how the compiler optimizes your code. Optimization usually makes a program faster, but this is not always true.

- `-O` *level*: Use optimizations up to and including the specified level. Higher levels usually produce faster code but take longer to compile. Levels range from `-O0` (no optimization, the default) to `-O3` (the most optimization available).

5.2.6 Further reading

This list is by no means exhaustive. A more complete list of gfortran-specific specific flags is at http://gcc.gnu.org/onlinedocs/gfortran/Invoking-GNU-Fortran.html or on the gfortran man page.

gfortran is part of the GCC family of compilers; more general information on GCC command line options is available at http://gcc.gnu.org/onlinedocs/gcc/Invoking-GCC.html, although some of this information is specific to compiling C programs rather than Fortran.

See also http://linux.die.net/man/1/gfortran.

A list of debug flags can also be found at http://www.fortran-2000.com/ArnaudRecipes/CompilerTricks.html#CompTable_fortran

5.3 Fortran subroutines and functions

5.3.1 Functions

Here's an example of the use of a function:

```
! $UWHPSC/codes/fortran/fcn1.f90

program fcn1
    implicit none
    real(kind=8) :: y,z
    real(kind=8), external :: f

    y = 2.
    z = f(y)
    print *, "z = ",z
end program fcn1

real(kind=8) function f(x)
    implicit none
    real(kind=8), intent(in) :: x
    f = x**2
end function f
```

It prints out:

```
z =     4.00000000000000
```

Comments:

- A function returns a single value. Since this function is named *f*, the value of *f* must be set in the function somewhere. You cannot use *f* on the right hand side of any expression, e.g. you cannot set $g = f$ in the function.
- f is declared *external* in the main program to let the compiler know it is a function defined elsewhere rather than a variable.
- The *intent(in)* statement tells the compiler that *x* is a variable passed into the function that will not be modified in the function.
- Here the program and function are in the same file. Later we will see how to break things up so each function or subroutine is in a separate file.

5.3.2 Modifying arguments

The input argument(s) to a function might also be modified, though this is not so common as using a subroutine as described below. But here's an example:

```
! $UWHPSC/codes/fortran/fcn2.f90

program fcn2
    implicit none
    real(kind=8) :: y,z
    real(kind=8), external :: f

    y = 2.
    print *, "Before calling f: y = ",y
    z = f(y)
    print *, "After calling f:  y = ",y
    print *, "z = ",z
end program fcn2

real(kind=8) function f(x)
    implicit none
    real(kind=8), intent(inout) :: x
    f = x**2
```

```
    x = 5.
end function f
```

This produces:

```
Before calling f: y =     2.00000000000000
After calling f:  y =     5.00000000000000
z =     4.00000000000000
```

5.3.3 The use of *intent*

You are not required to specify the intent of each argument, but there are several good reasons for doing so:

- It helps catch bugs. If you specify *intent(in)* and then this variable is changed in the function or subroutine, the compiler will give an error.
- It can help the compiler produce machine code that runs faster. For example, if it is known to the compiler that some variables will never change during execution, this fact can be used.

5.3.4 Subroutines

In Fortran, subroutines are generally used much more frequently than functions. Functions are expected to produce a single output variable and examples like the one just given where an argument is modified are considered bad programming style.

Subroutines are more flexible since they can have any number of inputs and outputs. In particular they may have not output variable. For example a subroutine might take an array as an argument and print the array to some file on disk but not return a value to the calling program.

Here is a version of the same program as *fcn1* above, that instead uses a subroutine:

```
! $UWHPSC/codes/fortran/sub1.f90

program sub1
    implicit none
    real(kind=8) :: y,z

    y = 2.
    call fsub(y,z)
    print *, "z = ",z
end program sub1

subroutine fsub(x,f)
    implicit none
    real(kind=8), intent(in) :: x
    real(kind=8), intent(out) :: f
    f = x**2
end subroutine fsub
```

Comments:

- Now we tell the compiler that x is an input variable and y is an output variable for the subroutine. The *intent* declarations are optional but sometimes help the compiler optimize code.

Here is a version that works on an array x instead of a single value:

```
! $UWHPSC/codes/fortran/sub2.f90

program sub2
    implicit none
    real(kind=8), dimension(3) :: y,z
    integer n

    y = (/2., 3., 4./)
    n = size(y)
    call fsub(y,n,z)
    print *, "z = ",z
end program sub2

subroutine fsub(x,n,f)
  ! compute f(x) = x**2 for all elements of the array x
  ! of length n.
  implicit none
  integer, intent(in) :: n
  real(kind=8), dimension(n), intent(in) :: x
  real(kind=8), dimension(n), intent(out) :: f
  f = x**2
end subroutine fsub
```

This produces:

```
4.00000000000000        9.00000000000000        16.0000000000000
```

Comments:

- The length of the array is also passed into the subroutine. You can avoid this in Fortran 90 (see the next example below), but it was unavoidable in Fortran 77 and subroutines working on arrays in Fortran are often written so that the dimensions are passed in as arguments.

5.3.5 Subroutine in a module

Here is a version that avoids passing the length of the array into the subroutine. In order for this to work some additional *interface* information must be specified. This is most easily done by including the subroutine in a *module*.

```
! $UWHPSC/codes/fortran/sub3.f90

module sub3module

contains

subroutine fsub(x,f)
  ! compute f(x) = x**2 for all elements of the array x.
  implicit none
  real(kind=8), dimension(:), intent(in) :: x
  real(kind=8), dimension(size(x)), intent(out) :: f
  f = x**2
end subroutine fsub

end module sub3module

!-----------------------------------------

program sub3
    use sub3module
```

```
    implicit none
    real(kind=8), dimension(3) :: y,z

    y = (/2., 3., 4./)
    call fsub(y,z)
    print *, "z = ",z
end program sub3
```

Comments:

- See the section *Fortran modules* for more information about modules. Note in particular that the module must be defined first and then used in the program via the *use* statement.
- The declaration of *x* in the subroutine uses *dimension(:)* to indicate that it is an array with a single index (having *rank 1*), without specifying how long the array is. If *x* was a rank 2 array indexed by *x(i,j)* then the dimension statement would be *dimension(:,:)*.
- The declaration of *f* in the subroutine uses *dimension(size(x))* to indicate that it is an array with one index and the length of the array is the same as that of *x*. In fact it would be sufficient to tell the compiler that it is an array of rank 1:

```
    real(kind=8), dimension(:), intent(out) :: f
```

  but indicating that it has the same size ax *x* is useful for someone trying to understand the code.

5.3.6 Further reading

- *Fortran*
- *Fortran examples: Taylor series*

5.4 Fortran examples: Taylor series

Here is an example code that uses the Taylor series for $exp(x)$ to estimate the value of this function at $x = 1$:

```
! $UWHPSC/codes/fortran/taylor.f90

program taylor

    implicit none
    real (kind=8) :: x, exp_true, y
    real (kind=8), external :: exptaylor
    integer :: n

    n = 20               ! number of terms to use
    x = 1.0
    exp_true = exp(x)
    y = exptaylor(x,n)   ! uses function below
    print *, "x = ",x
    print *, "exp_true  = ",exp_true
    print *, "exptaylor = ",y
    print *, "error     = ",y - exp_true

end program taylor

!========================
```

```
function exptaylor(x,n)
!==========================
    implicit none

    ! function arguments:
    real (kind=8), intent(in) :: x
    integer, intent(in) :: n
    real (kind=8) :: exptaylor

    ! local variables:
    real (kind=8) :: term, partial_sum
    integer :: j

    term = 1.
    partial_sum = term

    do j=1,n
        ! j'th term is  x**j / j!  which is the previous term times x/j:
        term = term*x/j
        ! add this term to the partial sum:
        partial_sum = partial_sum + term
        enddo
     exptaylor = partial_sum  ! this is the value returned
end function exptaylor
```

Running this code gives:

```
x =    1.00000000000000
exp_true  =    2.71828182845905
exptaylor =    2.71828174591064
error     =  -8.254840055954560E-008
```

Here's a modification where the number of terms to use is computed based on the size of the next term in the series. The function has been rewritten as a subroutine so the number of terms can be returned as well.

```
! $UWHPSC/codes/fortran/taylor_converge.f90

program taylor_converge

    implicit none
    real (kind=8) :: x, exp_true, y, relative_error
    integer :: nmax, nterms, j

    nmax = 100

    print *, "     x         true              approximate          error         nterms"
    do j = -20,20,4
       x = float(j)                      ! convert to a real
       call exptaylor(x,nmax,y,nterms)   ! defined below
       exp_true = exp(x)
       relative_error = abs(y-exp_true) / exp_true
       print '(f10.3,3d19.10,i6)', x, exp_true, y, relative_error, nterms
       enddo

end program taylor_converge

!==================================
subroutine exptaylor(x,nmax,y,nterms)
!==================================
```

```
    implicit none

    ! subroutine arguments:
    real (kind=8), intent(in) :: x
    integer, intent(in) :: nmax
    real (kind=8), intent(out) :: y
    integer, intent(out) :: nterms

    ! local variables:
    real (kind=8) :: term, partial_sum
    integer :: j

    term = 1.
    partial_sum = term

    do j=1,nmax
        ! j'th term is  x**j / j!  which is the previous term times x/j:
        term = term*x/j
        ! add this term to the partial sum:
        partial_sum = partial_sum + term
        if (abs(term) < 1.d-16*partial_sum) exit
        enddo
     nterms = j       ! number of terms used
     y = partial_sum  ! this is the value returned
end subroutine exptaylor
```

This produces:

```
   x          true              approximate          error          nterms
-20.000   0.2061153622D-08   0.5621884472D-08   0.1727542678D+01    95
-16.000   0.1125351747D-06   0.1125418051D-06   0.5891819580D-04    81
-12.000   0.6144212353D-05   0.6144213318D-05   0.1569943213D-06    66
 -8.000   0.3354626279D-03   0.3354626279D-03   0.6586251980D-11    51
 -4.000   0.1831563889D-01   0.1831563889D-01   0.1723771005D-13    34
  0.000   0.1000000000D+01   0.1000000000D+01   0.0000000000D+00     1
  4.000   0.5459815003D+02   0.5459815003D+02   0.5205617665D-15    30
  8.000   0.2980957987D+04   0.2980957987D+04   0.1525507414D-15    42
 12.000   0.1627547914D+06   0.1627547914D+06   0.3576402292D-15    51
 16.000   0.8886110521D+07   0.8886110521D+07   0.0000000000D+00    59
 20.000   0.4851651954D+09   0.4851651954D+09   0.1228543295D-15    67
```

Comments:

- Note the use of *exit* to break out of the loop.
- Note that it is getting full machine precision for positive values of x but, as expected, the accuracy suffers for negative x due to cancellation.

5.5 Array storage in Fortran

When an array is declared in Fortran, a set of storage locations in memory are set aside for the storage of all the values in the array. How many bytes of memory this requires depends on how large the array is and what data type each element has. If the array is declared *allocatable* then the declaration only determines the *rank* of the array (the number of indices it will have), and memory is not actually allocated until the *allocate* statement is encountered.

By default, arrays in Fortran are indexed starting at 1. So if you declare:

```
real(kind=8) :: x(3)
```

or equivalently:

```
real(kind=8), dimension(3) :: x
```

for example, then the elements should be referred to as *x(1), x(2),* and *x(3).*

You can also specify a different starting index, for example here are several arrays of length 3 with different starting indices:

```
real(kind=8) :: x(0:2), y(4:6), z(-2:0)
```

A statement like

x(0) = z(-1)

would then be valid.

Arrays in Fortran occupy successive memory locations starting at some address in memory, say *istart*, and increasing by some constant number for each element of the array. For example, for an array of *real (kind=8)* values the byte-address would increase by 8 for each successive element. As programmers we don't need to worry much about the actual addresses, but it is important to understand how arrays are laid out in memory, particularly if the rank of the array (number of indices) is larger than 1, as discussed below in Section *Fortran arrays of higher rank*.

5.5.1 Passing rank 1 arrays to subroutines, Fortran 77 style

Even for arrays of rank 1, it is sometimes useful to remember that to a Fortran compiler an array is often specified simply the the memory address of the first component. This helps explain the strange behavior of the following program:

```
! $UWHPSC/codes/fortran/arraypassing1.f90

program arraypassing1

    implicit none
    real(kind=8) :: x,y
    integer :: i,j

    x = 1.
    y = 2.
    i = 3
    j = 4
    call setvals(x)
    print *, "x = ",x
    print *, "y = ",y
    print *, "i = ",i
    print *, "j = ",j

end program arraypassing1

subroutine setvals(a)
    ! subroutine that sets values in an array a of length 3.
    implicit none
    real(kind=8), intent(inout) :: a(3)
    integer i
    do i = 1,3
        a(i) = 5.
```

```
        enddo
end subroutine setvals
```

which produces the output:

```
x =    5.00000000000000
y =    5.00000000000000
i =   1075052544
j =            0
```

The address of x is passed to the subroutine, which interprets it as the address of the starting point of an array of length 3. The subroutine sets the value of x to 5 and also sets the values of the next 2 memory locations (based on 8-byte real numbers) to 5. Because y, i, and j were declared after x and hence happen to occupy memory after x, these values are corrupted by the subroutine.

Note that integers are stored in 4 bytes so both i and j are covered by the single 8-bytes interpreted as *a(3)*. Storing a value as an 8-byte float and then interpreting the two halfs as integers (when i and j are printed) is likely to give nonsense.

It is also possible to make the code crash by improperly accessing memory. (This is actually be better than just producing nonsense with no warning, but figuring out *why* the code crashed may be difficult.)

If you change the dimension of a from 3 to 1000 in the subroutine above:

```
! $UWHPSC/codes/fortran/arraypassing2.f90

program arraypassing2

    implicit none
    real(kind=8) :: x,y
    integer :: i,j

    x = 1.
    y = 2.
    i = 3
    j = 4
    call setvals(x)
    print *, "x = ",x
    print *, "y = ",y
    print *, "i = ",i
    print *, "j = ",j

end program arraypassing2

subroutine setvals(a)
    ! subroutine that sets values in an array a of length 1000.
    implicit none
    real(kind=8), intent(inout) :: a(1000)
    integer i
    do i = 1,1000
        a(i) = 5.
        enddo
end subroutine setvals
```

then the code produces:

```
Segmentation fault
```

This means that the subroutine attempted to write into a memory location that it should not have access to. A small number of memory locations were reserved for data when the variables *x,y,i,j* were declared. No new memory is

allocated in the subroutine – the statement

```
real(kind=8), intent(inout) :: a(1000)
```

simply declares a *dummy argument* of rank 1. This statement could be replaced by

```
real(kind=8), intent(inout) :: a(:)
```

and the code would still compile and give the same results. The starting address of a set of storage locations is passed into the subroutine and the location of any element in the array is computed from this, whether or not it actually lies in the array as it was declared in the calling program!

The programs above are written in Fortran 77 style. In Fortran 77, every subroutine or function is compiled independently of others with no way to check that the arguments passed into a subroutine actually agree in type with the dummy arguments used in the subroutine. This is a limitation that often leads to debugging nightmares.

Luckily Fortran 90 can help reduce these problems, since it is possible to create an *interface* that provides more information about the arguments expected. Here's one simple way to do this for the code above:

```
! $UWHPSC/codes/fortran/arraypassing3.f90

program arraypassing3

    implicit none
    real(kind=8) :: x,y
    integer :: i,j

    x = 1.
    y = 2.
    i = 3
    j = 4
    call setvals(x)
    print *, "x = ",x
    print *, "y = ",y
    print *, "i = ",i
    print *, "j = ",j

contains

subroutine setvals(a)
    ! subroutine that sets values in an array a of length 3.
    implicit none
    real(kind=8), intent(inout) :: a(3)
    integer i
    do i = 1,3
        a(i) = 5.
        enddo
end subroutine setvals

end program arraypassing3
```

Trying to compile this code produces an error message:

```
$ gfortran arraypassing3.f90
arraypassing3.f90:14.17:

    call setvals(x)
                 1
Error: Type/rank mismatch in argument 'a' at (1)
```

Now at least the compiler recognizes that an array is expected rather than a single value. But it is still possible to write a code that compiles and produces nonsense by declaring *x* and *y* to be rank 1 arrays of length 1:

```
! $UWHPSC/codes/fortran/arraypassing4.f90

program arraypassing4

    implicit none
    real(kind=8), dimension(1) :: x,y
    integer :: i,j

    x(1) = 1.
    y(1) = 2.
    i = 3
    j = 4
    call setvals(x)
    print *, "x = ",x(1)
    print *, "y = ",y(1)
    print *, "i = ",i
    print *, "j = ",j

contains

subroutine setvals(a)
    ! subroutine that sets values in an array a of length 3.
    implicit none
    real(kind=8), intent(inout) :: a(3)
    integer i
    do i = 1,3
        a(i) = 5.
        enddo
end subroutine setvals

end program arraypassing4
```

The compiler sees that an object of the correct type (a rank 1 array) is being passed and does not give an error. Running the code gives

```
x =    5.00000000000000
y =    5.00000000000000
i =   1075052544
j =            0
```

You might hope that using the gfortran flag *-fbounds-check* (see *Useful gfortran flags*) would catch such bugs. Unfortunately it does not. It will catch it if you refer to *x(2)* in the main program of the code just given, where it knows the length of *x* that was declared, but not in the subroutine.

5.5.2 Fortran arrays of higher rank

Suppose we declare *A* to be a rank 2 array with 3 rows and 4 columns, which we might want to do to store a 3 by 4 matrix.

```
real(kind=8) :: A(3,4)
```

Since memory is laid out linearly (each location has a single address, not a set of indices), the compiler must decide how to map the 12 array elements to memory locations.

Unfortunately different languages use different conventions. In Fortran arrays are stored *by column* in memory, so the 12 consecutive memory locations would correspond to:

```
A(1,1)
A(2,1)
A(3,1)
A(1,2)
A(2,2)
A(3,2)
A(1,3)
A(2,3)
A(3,3)
A(1,4)
A(2,4)
A(3,4)
```

To illustrate this, consider the following program. It declares an array *A* of shape (3,4) and a rank-1 array *B* of length 12. It also uses the Fortran *equivalence* statement to tell the compiler that these two arrays should point to the *same* locations in memory. This program shows that *A(i,j)* is the same as *B(3*(j-1) + i)*:

```
1  ! $UWHPSC/codes/fortran/rank2.f90
2
3  program rank2
4
5      implicit none
6      real(kind=8) :: A(3,4), B(12)
7      equivalence (A,B)
8      integer :: i,j
9
10     A = reshape((/(10*i, i=1,12)/), (/3,4/))
11
12     do i=1,3
13         print 20, i, (A(i,j), j=1,4)
14  20     format("Row ",i1," of A contains: ", 11x, 4f6.1)
15         print 21, i, (3*(j-1)+i, j=1,4)
16  21     format("Row ",i1," is in locations",4i3)
17         print 22, (B(3*(j-1)+i), j=1,4)
18  22     format("These elements of B contain:", 4x, 4f6.1, /)
19         enddo
20
21 end program rank2
```

This program produces:

```
Row 1 of A contains:              10.0  40.0  70.0 100.0
Row 1 is in locations  1  4  7 10
These elements of B contain:      10.0  40.0  70.0 100.0

Row 2 of A contains:              20.0  50.0  80.0 110.0
Row 2 is in locations  2  5  8 11
These elements of B contain:      20.0  50.0  80.0 110.0

Row 3 of A contains:              30.0  60.0  90.0 120.0
Row 3 is in locations  3  6  9 12
These elements of B contain:      30.0  60.0  90.0 120.0
```

Note also that the *reshape* command used in Line 10 of this program takes the set of values and assigns them to elements of the array *by columns*. Actually it just puts these values into the 12 memory elements used for the matrix one after another, but because of the way arrays are interpreted, this corresponds to filling the array by columns.

Note some other things about this program:

- Lines 10, 13, 15, and 17 use an *implied do* construct, in which *i* or *j* loops over the values specified.

- Lines 14, 16, and 18 are *format statements* and these formats are used in the lines preceding them instead of the default format *. For more about formatted I/O see *Fortran Input / Output*.

In C or Python, storage is *by rows* instead, so the 12 consecutive memorylocations would correspond to:

```
A(1,1)
A(1,2)
A(1,3)
A(2,1)
etc.
```

5.5.3 Why do we care how arrays are stored?

The layout of arrays in memory is often important to know when doing high-performance computing, as we will see when we discuss cache properties.

It is also important to know this in order to understand older Fortran programs. When an array of rank 2 is passed to a subroutine, the subroutine needs to know not only that the array has rank 2, but also what the *leading dimension* of the array is, the number of rows. In older codes this value is often passed in to a subroutine along with the array. In Fortran 90 this can be avoided if there is a suitable interface, for example if the subroutine is in a module.

As we saw above, the *(i,j)* element of the 3 by 4 array *A* is in location *(3*(j-1) + i)*. The same would be true for a 3 by 5 array or a 3 by 1000 array. More generally, if the array is *nrows* by *ncols*, then the *(i,j)* element is in location *nrows*(j-1) + i* and so the value of *nrows* must be known by the compiler in order to translate the indices *(i,j)* into the propoer storage location.

5.6 Fortran modules

The general structure of a Fortran module:

```
module <MODULE-NAME>
    ! Declare variables
contains
    ! Define subroutines or functions
end module <MODULE-NAME>
```

A program or subroutine can *use* this module:

```
program <NAME>
    use <MODULE-NAME>
    ! Declare variables
    ! Executable statements
end program <NAME>
```

The line:

```
use <MODULE-NAME>
```

can be replaced by:

```
use <MODULE-NAME>, only: <LIST OF SYMBOLS>
```

to specify that only certain variables/subroutines/functions from the module should be used. Doing it this way also makes it clear exactly what symbols are coming from which module in the case where you *use* several modules.

A very simple module is:

```
! $UWHPSC/codes/fortran/multifile2/sub1m.f90

module sub1m

contains

subroutine sub1()
    print *, "In sub1"
end subroutine sub1

end module sub1m
```

and a program that uses this:

```
! $UWHPSC/codes/fortran/multifile2/main.f90

program demo
    use sub1m, only: sub1
    print *, "In main program"
    call sub1()
end program demo
```

5.6.1 Some reasons to use modules

- Can define global variables in modules to be used in several different routines.

  In Fortran 77 this had to be done with common blocks — much less elegant.

- Subroutine/function interface information is generated to aid in checking that proper arguments are passed.

  It's often best to put all subroutines and functions in modules for this reason.

- Can define new data types to be used in several routines.

5.6.2 Compiling modules

Modules must be compiled *before* any program units that *use* the module. When a module is compiled, a *.o* file is created, but also a *.mod* file is created that must be present in order to compile a unit that *uses* the module.

5.6.3 Circles module example

Here is an example of a module that defines one parameter *pi* and two functions:

```
! $UWHPSC/codes/fortran/circles/circle_mod.f90

module circle_mod

    implicit none
    real(kind=8), parameter :: pi = 3.141592653589793d0

contains

    real(kind=8) function area(r)
        real(kind=8), intent(in) :: r
        area = pi * r**2
    end function area
```

```
14
15      real(kind=8) function circumference(r)
16          real(kind=8), intent(in) :: r
17          circumference = 2.d0 * pi * r
18      end function circumference
19
20  end module circle_mod
```

This might be used as follows:

```
1   ! $UWHPSC/codes/fortran/circles/main.f90
2
3   program main
4
5       use circle_mod, only: pi, area
6       implicit none
7       real(kind=8) :: a
8
9       ! print parameter pi defined in module:
10      print *, 'pi = ', pi
11
12      ! test the area function from module:
13      a = area(2.d0)
14      print *, 'area for a circle of radius 2: ', a
15
16  end program main
```

This gives the following output:

```
pi =     3.14159265358979
area for a circle of radius 2:     12.5663706143592
```

Note: that a parameter can be defined with a specific value that will then be available to all program units using the module.

5.6.4 Module variables

It is also possible to declare *variables* that can be shared between all program units using the module. This is a way to define "global variables" that might be set in one program unit and used in another, without the need to pass the variable as a subroutine or function argument. Module variables can be defined in a module and the Fortran statement

```
save
```

is used to indicate that variables defined in the module should have values saved between one use of the module to another. You should generally specify this if you use any module variables.

Here is another version of the circles code that stores *pi* as a module variable rather than a parameter:

```
1   ! $UWHPSC/codes/fortran/circles/circle_mod.f90
2   ! Version where pi is a module variable.
3
4   module circle_mod
5
6       implicit none
7       real(kind=8) :: pi
8       save
9
10  contains
11
```

```
    real(kind=8) function area(r)
        real(kind=8), intent(in) :: r
        area = pi * r**2
    end function area

    real(kind=8) function circumference(r)
        real(kind=8), intent(in) :: r
        circumference = 2.d0 * pi * r
    end function circumference

end module circle_mod
```

In this case we also need to initialize the variable *pi* by means of a subroutine such as:

```
! $UWHPSC/codes/fortran/circles/initialize.f90

subroutine initialize()

    ! Set the value of pi used elsewhere.
    use circle_mod, only: pi
    pi = acos(-1.d0)

end subroutine initialize
```

These might be used as follows in a main program:

```
! $UWHPSC/codes/fortran/circles/main.f90

program main

    use circle_mod, only: pi, area
    implicit none
    real(kind=8) :: a

    call initialize()   ! sets pi

    ! print module variable pi:
    print *, 'pi = ', pi

    ! test the area function from module:
    a = area(2.d0)
    print *, 'area for a circle of radius 2: ', a

end program main
```

This example can be compiled and executed by going into the directory *$UWHPSC/fortran/circles2/* and typing:

```
$ gfortran circle_mod.f90 initialize.f90 main.f90 -o main.exe
$ ./main.exe
```

Or by using the Makefile in this directory:

```
$ make main.exe
$ ./main.exe
```

Here is the Makefile:

```
# $UWHPSC/codes/fortran/circles2/Makefile

OBJECTS = circle_mod.o \
```

```
          main.o \
          initialize.o

MODULES = circle_mod.mod

.PHONY: clean

output.txt: main.exe
        ./main.exe > output.txt

main.exe: $(MODULES) $(OBJECTS)
        gfortran $(OBJECTS) -o main.exe

%.o: %.f90
        gfortran -c $<

%.mod: %.f90
        gfortran -c $<

clean:
        rm -f $(OBJECTS) $(MODULES) main.exe
```

For more about Makefiles, see *Makefiles* and *Makefile references*.

5.7 Fortran Input / Output

5.7.1 Formats vs. unformatted

print or *write* statements for output and *read* statements for input can specify a *format* or can be *unformatted*.

For example,

```
print *, 'x = ', x
```

is an *unformatted* print statement that prints a character string followed by the value of a variable x. The format used to print x (e.g. the number of digits shown, the number of spaces in front) will be chosen by the compiler based on what type of variable x is.

The statements:

```
i = 4
x = 2.d0 / 3.d0
print *, 'i = ', i, ' and x = ', x
```

yield:

```
i =            4  and x =    0.666666666666667
```

The * in the print statement tells the compiler to choose the format.

To have more control over the format, a *formatted print* statement can be used. A format can be placed directly in the statement in place of the * , or can be written separately with a label, and the label number used in the *print* statement.

For example, if we wish to display the integer i in a *field* of 3 spaces and print x in scientific notation with 12 digits of the mantissa displayed, in a *field* that is 18 digits wide, we could do

```
    print 600, i, x
600 format('i = ',i3,' and x = ', e17.10)
```

This yields:

```
i =    4 and x =  0.6666666667E+00
```

The 4 is right-justified in a field of 3 characters after the 'i = ' string.

Note that if the number doesn't fit in the field, asterisks will be printed instead!

```
i = 4000
print 600, i, x
```

gives:

```
i = *** and x =  0.6666666667E+00
```

Instead of using a label and writing the format on a separate line, it can be put directly in the print statement, though this is often hard to read. The above print statement can be written as:

```
print "('i = ',i3,' and x = ', e17.10)", i, x
```

5.7.2 Writing to a file

Instead of printing directly to the terminal, we often want to write results out to a file. This can be done using the *open* statement to open a file and attach it to a particular unit number, and then use the *write* statement to write to this unit:

```
open(unit=20, file='output.txt')
write(20,*) i, x
close(20)
```

This would do an *unformatted* write to the file 'output.txt' instead of writing to the terminal. The * in the write statement can be replaced by a format, or a format label, as in the *print* statement.

There are many other optional arguments to the *open* command.

Unit numbers should generally be larger than 6. By default, unit 6 refers to the terminal for output, so

```
write(6,*) i, x
```

is the same as

```
print *, i, x
```

5.7.3 Reading input

Unformatted read:

```
print *, "Please input n... "
read *, n
```

Reading from a file:

```
open(unit=21, file="infile.txt")
read(21,*) n
close(21)
```

5.8 Fortran debugging

5.8.1 Print statements

Adding print statements to a program is a tried and true method of debugging, and the only method that many programmers use. Not because it's the best method, but it's sometimes the simplest way to examine what's going on at a particular point in a program.

Print statements can be added almost anywhere in a Fortran code to print things out to the terminal window as it goes along.

You might want to put some special symbols in debugging statements to flag them as such, which makes it easier to see what output is your debug output, and also makes it easier to find them again later to remove from the code, e.g. you might use "+++" or "DEBUG".

5.8.2 Compiling with various gfortran flags

There are a number of flags you can use when compiling your code that will make it easier to debug.

Here's a generic set of options you might try:

```
$ gfortran -g -W -Wall -fbounds-check -pedantic-errors \
      -ffpe-trap=zero,invalid,overflow,underflow  program.f90
```

See *Useful gfortran flags* or the *gfortran man page <http://linux.die.net/man/1/gfortran>* for more information. Most of these options indicate that the program should give warnings or die if certain bad things happen.

Compiling with the *-g* flag indicates that information should be generated and saved during compilation that can be used to help debug the code using a debugger such as *gdb* or *totalview*. You generally have to compile with this option to use a debugger.

5.8.3 The *gdb* debugger

This is the Gnu open source debugger for Gnu compilers such as gfortran. Unfortunately it often works very poorly for Fortran.

You can install it on the VM using:

```
$ sudo apt-get install gdb
```

And then use via, e.g.:

$ cd $UWHPSC/codes/fortran $ gfortran -g segfault1.f90 $ gdb a.out (gdb) run

.... runs for a while and then prints Program received signal EXC_BAD_ACCESS, Could not access memory. Tells what line it died in.

(gdb) p i $1 = 241 (gdb) q

This at least reveals where the error happened and allows printing the value of *i* when it died.

5.8.4 Totalview

Totalview is a commercial debugger that works quite well on Fortran codes together with various compilers, including gfortran. It also works with other languages, and for parallel computing.

See Rogue Wave Softare – totalview family.

5.9 Fortran example for Newton's method

This example shows one way to implement Newton's method for solving an equation $f(x) = 0$, i.e. for a zero or root of the function *f(x)*.

See *Newton's method for the square root* for a description of how Newton's method works.

These codes can be found in *$UWHPSC/codes/fortran/newton*.

Here is the module that implements Newton's method in the subroutine *solve*:

```
! $UWHPSC/codes/fortran/newton/newton.f90

module newton

    ! module parameters:
    implicit none
    integer, parameter :: maxiter = 20
    real(kind=8), parameter :: tol = 1.d-14

contains

subroutine solve(f, fp, x0, x, iters, debug)

    ! Estimate the zero of f(x) using Newton's method.
    ! Input:
    !   f:  the function to find a root of
    !   fp: function returning the derivative f'
    !   x0: the initial guess
    !   debug: logical, prints iterations if debug=.true.
    ! Returns:
    !   the estimate x satisfying f(x)=0 (assumes Newton converged!)
    !   the number of iterations iters

    implicit none
    real(kind=8), intent(in) :: x0
    real(kind=8), external :: f, fp
    logical, intent(in) :: debug
    real(kind=8), intent(out) :: x
    integer, intent(out) :: iters

    ! Declare any local variables:
    real(kind=8) :: deltax, fx, fxprime
    integer :: k


    ! initial guess
    x = x0

    if (debug) then
        print 11, x
 11     format('Initial guess: x = ', e22.15)
        endif

    ! Newton iteration to find a zero of f(x)

    do k=1,maxiter

        ! evaluate function and its derivative:
```

```
        fx = f(x)
        fxprime = fp(x)

        if (abs(fx) < tol) then
            exit  ! jump out of do loop
            endif

        ! compute Newton increment x:
        deltax = fx/fxprime

        ! update x:
        x = x - deltax

        if (debug) then
            print 12, k,x
 12         format('After', i3, ' iterations, x = ', e22.15)
            endif

        enddo


    if (k > maxiter) then
        ! might not have converged

        fx = f(x)
        if (abs(fx) > tol) then
            print *, '*** Warning: has not yet converged'
            endif
        endif

    ! number of iterations taken:
    iters = k-1


end subroutine solve

end module newton
```

The *solve* routine takes two functions f and *fp* as arguments. These functions must return the value $f(x)$ and $f'(x)$ respectively for any input x.

A sample test code shows how *solve* is called:

```
! $UWHPSC/codes/fortran/newton/test1.f90

program test1

    use newton, only: solve
    use functions, only: f_sqrt, fprime_sqrt

    implicit none
    real(kind=8) :: x, x0, fx
    real(kind=8) :: x0vals(3)
    integer :: iters, itest
        logical :: debug         ! set to .true. or .false.

    print *, "Test routine for computing zero of f"
    debug = .true.

```

```
    ! values to test as x0:
    x0vals = (/1.d0, 2.d0, 100.d0 /)

    do itest=1,3
        x0 = x0vals(itest)
                print *, ' '  ! blank line
        call solve(f_sqrt, fprime_sqrt, x0, x, iters, debug)

        print 11, x, iters
11      format('solver returns x = ', e22.15, ' after', i3, ' iterations')

        fx = f_sqrt(x)
        print 12, fx
12      format('the value of f(x) is ', e22.15)

        if (abs(x-2.d0) > 1d-14) then
            print 13, x
13          format('*** Unexpected result: x = ', e22.15)
            endif
        enddo

end program test1
```

This makes use of a module *functions.f90* that includes the definition of the functions used here. Since $f(x) = x^2 - 4$, we are attempting to compute the square root of 4.

```
! $UWHPSC/codes/fortran/newton/functions.f90

module functions

contains

real(kind=8) function f_sqrt(x)
    implicit none
    real(kind=8), intent(in) :: x

    f_sqrt = x**2 - 4.d0

end function f_sqrt


real(kind=8) function fprime_sqrt(x)
    implicit none
    real(kind=8), intent(in) :: x

    fprime_sqrt = 2.d0 * x

end function fprime_sqrt

end module functions
```

This test can be run via:

```
$ make test1
```

which uses the Makefile in the same directory:

```
# $UWHPSC/codes/fortran/newton/Makefile

```

```
OBJECTS = functions.o newton.o test1.o
MODULES = functions.mod newton.mod

FFLAGS = -g

.PHONY: test1 clean

test1: test1.exe
	./test1.exe

test1.exe: $(MODULES) $(OBJECTS)
	gfortran $(FFLAGS) $(OBJECTS) -o test1.exe

%.o : %.f90
	gfortran $(FFLAGS) -c  $<

%.mod: %.f90
	gfortran $(FFLAGS) -c $<

clean:
	rm -f *.o *.exe *.mod
```

CHAPTER

SIX

PARALLEL COMPUTING

6.1 OpenMP

OpenMP is discussed in slides starting with Lecture 13.

6.1.1 Sample codes

There are a few sample codes in the *$UWHPSC/codes/openmp* directory. See the *README.txt* file for instructions on compiling and executing.

Here is a very simple code, that simply evaluates a costly function at many points:

```
! $UWHPSC/codes/openmp/yeval.f90

! If this code gives a Segmentation Fault when compiled and run with -fopenmp
! Then you could try:
!   $ ulimit -s unlimited
! to increase the allowed stack size.
! This may not work on all computers.  On a Mac the best you can do is
!   $ ulimit -s hard
! Correction... you can do the following on a Mac:
!   $ gfortran -fopenmp -Wl,-stack_size,2faf2000 yeval.f90
! here 2faf2000 is a hexadecimal number slightly larger than 8e8, the
! number of bytes needed for the value of n used in this program.


program yeval

   use omp_lib
   implicit none
   integer, parameter :: n = 100000000
   integer :: i, nthreads
   real(kind=8), dimension(n) :: y
   real(kind=8) :: dx, x

   ! Specify number of threads to use:
   !$ print *, "How many threads to use? "
   !$ read *, nthreads
   !$ call omp_set_num_threads(nthreads)
   !$ print "('Using OpenMP with ',i3,' threads')", nthreads

   dx = 1.d0 / (n+1.d0)

```

```
    !$omp parallel do private(x)
    do i=1,n
       x = i*dx
       y(i) = exp(x)*cos(x)*sin(x)*sqrt(5*x+6.d0)
    enddo

    print *, "Filled vector y of length", n

end program yeval
```

Note the following:

- Lines starting with *!$* are only executed if the code is compiled and run with the flag *-fopenmp*, otherwise they are comments.
- *x* must be declared a *private* variable in the *omp parallel do* loop, so that each thread has its own version. Otherwise another thread might reset *x* between the time its assigned a value and the time this value is used to set *y(i)*.
- The loop iterator *i* is private by default, but all other varaibles are shared by default.
- If you try to run this and get a "segmentation fault", it is probably because the stack size limit is too small. You can see the limit with:

```
$ ulimit -s
```

On linux you can do:

```
$ ulimit -s unlimited
```

But on a Mac there is a hard limit and the best you can do is:

```
$ ulimit -s hard
```

If you still get a segmentation fault you will have to decrease *n* for this example.

6.1.2 Other directives

This example illustrates some directives beyond the *parallel do*:

```
! $UWHPSC/codes/openmp/demo2

program demo2

    use omp_lib
    implicit none
    integer :: i
    integer, parameter :: n = 100000
    real(kind=8), dimension(n) :: x,y,z

    ! Specify number of threads to use:
    !$ call omp_set_num_threads(2)

    !$omp parallel  ! spawn two threads
    !$omp sections  ! split up work between them

      !$omp section
      x = 1.d0   ! one thread initializes x array

```

```
      !$omp section
      y = 1.d0   ! another thread initializes y array

   !$omp end sections
   !$omp barrier   ! not needed, implied at end of sections

   !$omp single    ! only want to print once:
   print *, "Done initializing x and y"
   !$omp end single nowait  ! ok for other thread to continue

   !$omp do   ! split work between threads:
   do i=1,n
       z(i) = x(i) + y(i)
       enddo

   !$omp end parallel
   print *, "max value of z is ",maxval(z)


end program demo2
```

Notes:

- *!$omp sections* is used to split up work between threads
- There is an implicit barrier after *!$omp end sections*, so the explicit barrier here is optional.
- The print statement is only done once since it is in *!$omp single*. The *nowait* clause indicates that the other thread can proceed without waiting for this print to be executed.

6.1.3 Fine-grain vs. coarse-grain paralellism

Consider the problem of normalizing a vector by dividing each element by the 1-norm of the vector, defined by $\|x\|_1 = \sum_{i=1}^n |x_i|$.

We must first loop over all points to compute the norm. Then we must loop over all points and set $y_i = x_i/\|x\|_1$. Note that we cannot combine these two loops into a single loop!

Here is an example with *fine-grain paralellism*, where we use the OpenMP *omp parallel do* directive or the *omp do* directive within a *omp parallel* block.

```
! $UWHPSC/codes/openmp/normalize1.f90

! Example of normalizing a vector using fine-grain parallelism.

program main

    use omp_lib
    implicit none
    integer :: i, thread_num
    integer, parameter :: n = 1000

    real(kind=8), dimension(n) :: x, y
    real(kind=8) :: norm,ynorm

    integer :: nthreads

    ! Specify number of threads to use:
    nthreads = 1       ! need this value in serial mode
```

```
    !$ nthreads = 4
    !$ call omp_set_num_threads(nthreads)
    !$ print "('Using OpenMP with ',i3,' threads')", nthreads

    ! Specify number of threads to use:
    !$ call omp_set_num_threads(4)

    ! initialize x:
    !$omp parallel do
    do i=1,n
        x(i) = dble(i)  ! convert to double float
    enddo

    norm = 0.d0
    ynorm = 0.d0

    !$omp parallel private(i)

    !$omp do reduction(+ : norm)
    do i=1,n
        norm = norm + abs(x(i))
        enddo

     !$omp barrier   ! not needed (implicit)

    !$omp do reduction(+ : ynorm)
    do i=1,n
        y(i) = x(i) / norm
        ynorm = ynorm + abs(y(i))
        enddo

    !$omp end parallel

    print *, "norm of x = ",norm, "  n(n+1)/2 = ",n*(n+1)/2
    print *, 'ynorm should be 1.0:   ynorm = ', ynorm

end program main
```

Note the following:

- We initialize $x_i = i$ as a test, so $\|x\|_1 = n(n+1)/2$.
- The compiler decides how to split the loop between threads. The loop starting on line 38 might be split differently than the loop starting on line 45.
- Because of this, all threads must have access to all of memory.

Next is a version with *coarse-grain parallelism*, were we decide how to split up the array between threads and then execute the same code on each thread, but each thread will compute its own version of *istart* and *iend* for its portion of the array. With this code we are guaranteed that thread 0 always handles *x(1)*, for example, so in principle the data could be distributed. When using OpenMP on a shared memory computer this doesn't matter, but this version is more easily generalized to MPI.

```
! $UWHPSC/codes/openmp/normalize2.f90

! Example of normalizing a vector using coarse-grain parallelism.

program main

```

```
    use omp_lib
    implicit none
    integer, parameter :: n = 1000
    real(kind=8), dimension(n) :: x,y
    real(kind=8) :: norm,norm_thread,ynorm,ynorm_thread
    integer :: nthreads, points_per_thread,thread_num
    integer :: i,istart,iend

    ! Specify number of threads to use:
    nthreads = 1       ! need this value in serial mode
    !$ nthreads = 4
    !$ call omp_set_num_threads(nthreads)
    !$ print "('Using OpenMP with ',i3,' threads')", nthreads

    ! Determine how many points to handle with each thread.
    ! Note that dividing two integers and assigning to an integer will
    ! round down if the result is not an integer.
    ! This, together with the min(...) in the definition of iend below,
    ! insures that all points will get distributed to some thread.
    points_per_thread = (n + nthreads - 1) / nthreads
    print *, "points_per_thread = ",points_per_thread

    ! initialize x:
    do i=1,n
        x(i) = dble(i)  ! convert to double float
        enddo

    norm = 0.d0
    ynorm = 0.d0

    !$omp parallel private(i,norm_thread, &
    !$omp                  istart,iend,thread_num,ynorm_thread)

    thread_num = 0     ! needed in serial mode
    !$ thread_num = omp_get_thread_num()    ! unique for each thread

    ! Determine start and end index for the set of points to be
    ! handled by this thread:
    istart = thread_num * points_per_thread + 1
    iend = min((thread_num+1) * points_per_thread, n)

    !$omp critical
    print 201, thread_num, istart, iend
    !$omp end critical
201 format("Thread ",i2," will take i = ",i6," through i = ",i6)

    norm_thread = 0.d0
    do i=istart,iend
        norm_thread = norm_thread + abs(x(i))
        enddo

    ! update global norm with value from each thread:
    !$omp critical
      norm = norm + norm_thread
      print *, "norm updated to: ",norm
    !$omp end critical

    ! make sure all have updated norm before proceeding:
```

```
65      !$omp barrier
66
67      ynorm_thread = 0.d0
68      do i=istart,iend
69          y(i) = x(i) / norm
70          ynorm_thread = ynorm_thread + abs(y(i))
71          enddo
72
73      ! update global ynorm with value from each thread:
74      !$omp critical
75        ynorm = ynorm + ynorm_thread
76        print *, "ynorm updated to: ",ynorm
77      !$omp end critical
78      !$omp barrier
79
80      !$omp end parallel
81
82      print *, "norm of x = ",norm, "  n(n+1)/2 = ",n*(n+1)/2
83      print *, 'ynorm should be 1.0:   ynorm = ', ynorm
84
85  end program main
```

Note the following:

- *istart* and *iend*, the starting and ending values of *i* taken by each thread, are explicitly computed in terms of the thread number. We must be careful to handle the case when the number of threads does not evenly divide *n*.
- Various variables must be declared *private* in lines 37-38.
- *norm* must be initialized to 0 before the *omp parallel* block. Otherwise some thread might set it to 0 after another thread has already updated it by its *norm_thread*.
- The update to *norm* on line 60 must be in a *omp critical* block, so two threads don't try to update it simultaneously (data race).
- There must be an *omp barrier* on line 65 between updating *norm* by each thread and using *norm* to compute each *y(i)*. We must make sure all threads have updated *norm* or it won't have the correct value when we use it.

For comparison of fine-grain and coarse-grain parallelism on Jacobi iteration, see

- *Jacobi iteration using OpenMP with parallel do constructs*
- *Jacobi iteration using OpenMP with coarse-grain parallel block*

6.1.4 Further reading

- *OpenMP references* in bibliography

6.2 MPI

MPI stands for *Message Passing Interface* and is a standard approach for programming distributed memory machines such as clusters, supercomputers, or heterogeneous networks of computers. It can also be used on a single shared memory computer, although it is often more cumbersome to program in MPI than in OpenMP.

6.2.1 MPI implementations

A number of different implementations are available (open source and vendor supplied for specific machines). See this list, for example.

6.2.2 MPI on the class VM

The VM has open-mpi partially installed.

You will need to do the following:

```
$ sudo apt-get update
$ sudo apt-get install openmpi-dev
```

On other Ubuntu installations you will also have to do:

```
$ sudo apt-get install openmpi-bin          # Already on the VM
```

You should then be able to do the following:

```
$ cd $UWHPSC/codes/mpi
$ mpif90 test1.f90
$ mpiexec -n 4 a.out
```

and see output like:

```
Hello from Process number           1  of            4  processes
Hello from Process number           3  of            4  processes
Hello from Process number           0  of            4  processes
Hello from Process number           2  of            4  processes
```

6.2.3 Test code

The simple test code used above illustrates use of some of the basic MPI subroutines.

```
! $UWHPSC/codes/mpi/test1.f90

program test1
    use mpi
    implicit none
    integer :: ierr, numprocs, proc_num

    call mpi_init(ierr)
    call mpi_comm_size(MPI_COMM_WORLD, numprocs, ierr)
    call mpi_comm_rank(MPI_COMM_WORLD, proc_num, ierr)

    print *, 'Hello from Process number', proc_num, &
             ' of ', numprocs, ' processes'

    call mpi_finalize(ierr)

end program test1
```

6.2.4 Reduction example

The next example uses *MPI_REDUCE* to add up partial sums computed by independent processes.

```
! $UWHPSC/codes/mpi/pisum1.f90

! Computes pi using MPI.
! Compare to $UWHPSC/codes/openmp/pisum2.f90

program pisum1
    use mpi
    implicit none
    integer :: ierr, numprocs, proc_num, points_per_proc, n, &
               i, istart, iend
    real (kind=8) :: x, dx, pisum, pisum_proc, pi

    call mpi_init(ierr)
    call mpi_comm_size(MPI_COMM_WORLD, numprocs, ierr)
    call mpi_comm_rank(MPI_COMM_WORLD, proc_num, ierr)

    ! Ask the user for the number of points
    if (proc_num == 0) then
        print *, "Using ",numprocs," processors"
        print *, "Input n ... "
        read *, n
    end if

    ! Broadcast to all procs; everybody gets the value of n from proc 0
    call mpi_bcast(n, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)

    dx = 1.d0/n

    ! Determine how many points to handle with each proc
    points_per_proc = (n + numprocs - 1)/numprocs
    if (proc_num == 0) then   ! Only one proc should print to avoid clutter
        print *, "points_per_proc = ", points_per_proc
    end if

    ! Determine start and end index for this proc's points
    istart = proc_num * points_per_proc + 1
    iend = min((proc_num + 1)*points_per_proc, n)

    ! Diagnostic: tell the user which points will be handled by which proc
    print '("Process ",i2," will take i = ",i6," through i = ",i6)', &
          proc_num, istart, iend

    pisum_proc = 0.d0
    do i=istart,iend
        x = (i-0.5d0)*dx
        pisum_proc = pisum_proc + 1.d0 / (1.d0 + x**2)
        enddo

    call MPI_REDUCE(pisum_proc,pisum,1,MPI_DOUBLE_PRECISION,MPI_SUM,0, &
                        MPI_COMM_WORLD,ierr)

    if (proc_num == 0) then
        pi = 4.d0 * dx * pisum
        print *, "The approximation to pi is ",pi
        endif

    call mpi_finalize(ierr)

```

```
end program pisum1
```

6.2.5 Send-Receive example

In this example, a value is set in Process 0 and then passed to Process 1 and on to Process 2, etc. until it reaches the last process, where it is printed out.

```
! $UWHPSC/codes/mpi/copyvalue.f90
!
! Set value in Process 0 and copy this through a chain of processes
! and finally print result from Process numprocs-1.
!

program copyvalue

    use mpi

    implicit none

    integer :: i, proc_num, num_procs,ierr
    integer, dimension(MPI_STATUS_SIZE) :: status

    call MPI_INIT(ierr)
    call MPI_COMM_SIZE(MPI_COMM_WORLD, num_procs, ierr)
    call MPI_COMM_RANK(MPI_COMM_WORLD, proc_num, ierr)

    if (num_procs==1) then
        print *, "Only one process, cannot do anything"
        call MPI_FINALIZE(ierr)
        stop
        endif


    if (proc_num==0) then
        i = 55
        print '("Process ",i3," setting      i = ",i3)', proc_num, i

        call MPI_SEND(i, 1, MPI_INTEGER, 1, 21, &
                      MPI_COMM_WORLD, ierr)

      else if (proc_num < num_procs - 1) then

        call MPI_RECV(i, 1, MPI_INTEGER, proc_num-1, 21, &
                      MPI_COMM_WORLD, status, ierr)

        print '("Process ",i3," receives     i = ",i3)', proc_num, i
        print '("Process ",i3," sends        i = ",i3)', proc_num, i

        call MPI_SEND(i, 1, MPI_INTEGER, proc_num+1, 21, &
                      MPI_COMM_WORLD, ierr)


      else if (proc_num == num_procs - 1) then

        call MPI_RECV(i, 1, MPI_INTEGER, proc_num-1, 21, &
                      MPI_COMM_WORLD, status, ierr)

```

```
        print '("Process ",i3," ends up with i = ",i3)', proc_num, i
      endif

    call MPI_FINALIZE(ierr)

end program copyvalue
```

6.2.6 Master-worker examples

The next two examples illustrate using Process 0 as a *master* process to farm work out to the other processes. In both cases the 1-norm of a matrix is computed, which is the maximum over *j* of the 1-norm of the 'j'th column of the matrix.

In the first case we assume there are the same number of worker processes as columns in the matrix:

```
! $UWHPSC/codes/mpi/matrix1norm1.f90
!
! Compute 1-norm of a matrix using mpi.
! Process 0 is the master that sets things up and then sends a column
! to each worker (Processes 1, 2, ..., num_procs - 1).
!
! This version assumes there are at least as many workers as columns.

program matrix1norm1

    use mpi

    implicit none

    integer :: i,j,jj,nrows,ncols,proc_num, num_procs,ierr,nerr
    integer, dimension(MPI_STATUS_SIZE) :: status
    real(kind=8) :: colnorm
    real(kind=8), allocatable, dimension(:,:) :: a
    real(kind=8), allocatable, dimension(:) :: anorm, colvect

    call MPI_INIT(ierr)
    call MPI_COMM_SIZE(MPI_COMM_WORLD, num_procs, ierr)
    call MPI_COMM_RANK(MPI_COMM_WORLD, proc_num, ierr)

    nerr = 0
    if (proc_num==0) then
        print *, "Input nrows, ncols"
        read *, nrows, ncols
        if (ncols > num_procs-1) then
            print *, "*** Error, this version requires ncols < num_procs = ",&
                  num_procs
            nerr = 1
            endif
        allocate(a(nrows,ncols))  ! only master process 0 needs the matrix
        a = 1.d0  ! initialize to all 1's for this test
        allocate(anorm(ncols))    ! to hold norm of each column in MPI_RECV
        endif

    ! if nerr == 1 then all processes must stop:
    call MPI_BCAST(nerr, 1, MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD, ierr)

    if (nerr == 1) then
        ! Note that error message already printed by Process 0
```

```
        ! All processes must execute the MPI_FINALIZE
        ! (Could also just have "go to 99" here.)
        call MPI_FINALIZE(ierr)
        stop
        endif

    call MPI_BCAST(nrows, 1, MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD, ierr)
    call MPI_BCAST(ncols, 1, MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD, ierr)

    if (proc_num > 0) then
        allocate(colvect(nrows))    ! to hold a column vector sent from master
        endif



    ! ----------------------------------------
    ! code for Master (Processor 0):
    ! ----------------------------------------

    if (proc_num == 0) then

      do j=1,ncols
        call MPI_SEND(a(1,j), nrows, MPI_DOUBLE_PRECISION,&
                      j, j, MPI_COMM_WORLD, ierr)
        enddo

      do j=1,ncols
        call MPI_RECV(colnorm, 1, MPI_DOUBLE_PRECISION, &
                      MPI_ANY_SOURCE, MPI_ANY_TAG, &
                      MPI_COMM_WORLD, status, ierr)
        jj = status(MPI_TAG)
        anorm(jj) = colnorm
        enddo

      print *, "Finished filling anorm with values... "
      print *, anorm
      print *, "1-norm of matrix a = ", maxval(anorm)
      endif


    ! ----------------------------------------
    ! code for Workers (Processors 1, 2, ...):
    ! ----------------------------------------
    if (proc_num /= 0) then

        if (proc_num > ncols) go to 99   ! no work expected

        call MPI_RECV(colvect, nrows, MPI_DOUBLE_PRECISION,&
                      0, MPI_ANY_TAG, &
                      MPI_COMM_WORLD, status, ierr)

        j = status(MPI_TAG)    ! this is the column number
                               ! (should agree with proc_num)

        colnorm = sum(abs(colvect))

        call MPI_SEND(colnorm, 1, MPI_DOUBLE_PRECISION, &
                    0, j, MPI_COMM_WORLD, ierr)
```

```
        endif

99  continue   ! might jump to here if finished early
    call MPI_FINALIZE(ierr)

end program matrix1norm1
```

In the next case we consider the more realistic situation where there may be many more columns in the matrix than worker processes. In this case the *master* process must do more work to keep track of how which columns have already been handled and farm out work as worker processes become free.

```
! $UWHPSC/codes/mpi/matrix1norm2.f90
!
! Compute 1-norm of a matrix using mpi.
! Process 0 is the master that sets things up and then sends a column
! to each worker (Processes 1, 2, ..., num_procs - 1).
!
! This version allows more columns than workers.

program matrix1norm2

    use mpi

    implicit none

    integer :: i,j,jj,nrows,ncols,proc_num, num_procs,ierr,nerr
    integer :: numsent, sender, nextcol
    integer, dimension(MPI_STATUS_SIZE) :: status
    real(kind=8) :: colnorm
    real(kind=8), allocatable, dimension(:,:) :: a
    real(kind=8), allocatable, dimension(:) :: anorm, colvect

    logical :: debug

    debug = .true.

    call MPI_INIT(ierr)
    call MPI_COMM_SIZE(MPI_COMM_WORLD, num_procs, ierr)
    call MPI_COMM_RANK(MPI_COMM_WORLD, proc_num, ierr)

    if (proc_num==0) then
        print *, "Input nrows, ncols"
        read *, nrows, ncols
        allocate(a(nrows,ncols))  ! only master process 0 needs the matrix
        a = 1.d0  ! initialize to all 1's for this test
        allocate(anorm(ncols))    ! to hold norm of each column in MPI_RECV
        endif


    call MPI_BCAST(nrows, 1, MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD, ierr)
    call MPI_BCAST(ncols, 1, MPI_DOUBLE_PRECISION, 0, MPI_COMM_WORLD, ierr)

    if (proc_num > 0) then
        allocate(colvect(nrows))   ! to hold a column vector sent from master
        endif
```

```
45
46
47
48      ! ----------------------------------------
49      ! code for Master (Processor 0):
50      ! ----------------------------------------
51
52      if (proc_num == 0) then
53
54        numsent = 0 ! keep track of how many columns sent
55
56        ! send the first batch to get all workers working:
57        do j=1,min(num_procs-1, ncols)
58          call MPI_SEND(a(1,j), nrows, MPI_DOUBLE_PRECISION,&
59                        j, j, MPI_COMM_WORLD, ierr)
60          numsent = numsent + 1
61          enddo
62
63        ! as results come back, send out more work...
64        ! the variable sender tells who sent back a result and ready for more
65        do j=1,ncols
66          call MPI_RECV(colnorm, 1, MPI_DOUBLE_PRECISION, &
67                        MPI_ANY_SOURCE, MPI_ANY_TAG, &
68                        MPI_COMM_WORLD, status, ierr)
69          sender = status(MPI_SOURCE)
70          jj = status(MPI_TAG)
71          anorm(jj) = colnorm
72
73          if (numsent < ncols) then
74              ! still more work to do, the next column will be sent and
75              ! this index also used as the tag:
76              nextcol = numsent + 1
77              call MPI_SEND(a(1,nextcol), nrows, MPI_DOUBLE_PRECISION,&
78                            sender, nextcol, MPI_COMM_WORLD, ierr)
79              numsent = numsent + 1
80            else
81              ! send an empty message with tag=0 to indicate this worker
82              ! is done:
83              call MPI_SEND(MPI_BOTTOM, 0, MPI_DOUBLE_PRECISION,&
84                            sender, 0, MPI_COMM_WORLD, ierr)
85            endif
86
87          enddo
88
89        print *, "Finished filling anorm with values... "
90        print *, anorm
91        print *, "1-norm of matrix a = ", maxval(anorm)
92        endif
93
94
95      ! ----------------------------------------
96      ! code for Workers (Processors 1, 2, ...):
97      ! ----------------------------------------
98      if (proc_num /= 0) then
99
100         if (proc_num > ncols) go to 99   ! no work expected
101
102         do while (.true.)
```

```
            ! repeat until message with tag==0 received...

            call MPI_RECV(colvect, nrows, MPI_DOUBLE_PRECISION,&
                          0, MPI_ANY_TAG, &
                          MPI_COMM_WORLD, status, ierr)

            j = status(MPI_TAG)    ! this is the column number
                                   ! may not be proc_num in general

            if (debug) then
                print '("+++ Process ",i4,"  received message with tag ",i6)', &
                    proc_num, j
                endif

            if (j==0) go to 99    ! received "done" message

            colnorm = sum(abs(colvect))

            call MPI_SEND(colnorm, 1, MPI_DOUBLE_PRECISION, &
                        0, j, MPI_COMM_WORLD, ierr)

            enddo
        endif

99  continue   ! might jump to here if finished early
    call MPI_FINALIZE(ierr)

end program matrix1norm2
```

6.2.7 Sample codes

Some other sample codes can also be found in the *$UWHPSC/codes/mpi* directory.

- *Jacobi iteration using MPI*

See also the samples in the list below.

6.2.8 Further reading

- *MPI references* section of the bibliography lists some books.
- Argonne tutorials
- Tutorial slides by Bill Gropp
- Livermore tutorials
- Open MPI
- The MPI Standard
- Some sample codes
- LAM MPI tutorials
- Google "MPI tutorial" to find more.

- Documentation on MPI subroutines

CHAPTER

SEVEN

MISCELLANEOUS

7.1 Makefiles

The directory *$UWHPSC/codes/fortran/multifile1* contains a Fortran code *fullcode.f90* that consists of a main program and two subroutines:

```
! $UWHPSC/codes/fortran/multifile1/fullcode.f90

program demo
    print *, "In main program"
    call sub1()
    call sub2()
end program demo

subroutine sub1()
    print *, "In sub1"
end subroutine sub1

subroutine sub2()
    print *, "In sub2"
end subroutine sub2
```

To illustrate the construction of a Makefile, we first break this up into three separate files:

```
! $UWHPSC/codes/fortran/multifile1/main.f90

program demo
    print *, "In main program"
    call sub1()
    call sub2()
end program demo
```

```
! $UWHPSC/codes/fortran/multifile1/sub1.f90

subroutine sub1()
    print *, "In sub1"
end subroutine sub1
```

```
! $UWHPSC/codes/fortran/multifile1/sub2.f90

subroutine sub2()
    print *, "In sub2"
end subroutine sub2
```

The directory *$UWHPSC/codes/fortran/multifile1* contains several Makefiles that get successively more sophisticated to compile the codes in this directory.

In the first version we write out explicitly what to do for each file:

```
# $UWHPSC/codes/fortran/multifile1/Makefile

output.txt: main.exe
        ./main.exe > output.txt

main.exe: main.o sub1.o sub2.o
        gfortran main.o sub1.o sub2.o -o main.exe

main.o: main.f90
        gfortran -c main.f90
sub1.o: sub1.f90
        gfortran -c sub1.f90
sub2.o: sub2.f90
        gfortran -c sub2.f90
```

In the second version there is a general rule for creating *.o* files from *.f90* files:

```
# $UWHPSC/codes/fortran/multifile1/Makefile2

output.txt: main.exe
        ./main.exe > output.txt

main.exe: main.o sub1.o sub2.o
        gfortran main.o sub1.o sub2.o -o main.exe

%.o : %.f90
        gfortran -c $<
```

In the third version we define a macro *OBJECTS* so we only have to write out this list once, which minimizes the chance of introducing errors:

```
# $UWHPSC/codes/fortran/multifile1/Makefile3

OBJECTS = main.o sub1.o sub2.o

output.txt: main.exe
        ./main.exe > output.txt

main.exe: $(OBJECTS)
        gfortran $(OBJECTS) -o main.exe

%.o : %.f90
        gfortran -c $<
```

In the fourth version, we add a Fortran compile flag (for level 3 optimization) and an linker flag (blank in this example):

```
# $UWHPSC/codes/fortran/multifile1/Makefile4

FC = gfortran
FFLAGS = -O3
LFLAGS =
OBJECTS = main.o sub1.o sub2.o

output.txt: main.exe
        ./main.exe > output.txt
```

```

main.exe: $(OBJECTS)
	$(FC) $(LFLAGS) $(OBJECTS) -o main.exe

%.o : %.f90
	$(FC) $(FFLAGS) -c $<
```

Next we add a *phony* target *clean* that removes the files created when compiling the code in order to facilitate cleanup. It is *phony* because it does not create a file named *clean*.

```
# $UWHPSC/codes/fortran/multifile1/Makefile5

OBJECTS = main.o sub1.o sub2.o
.PHONY: clean

output.txt: main.exe
	./main.exe > output.txt

main.exe: $(OBJECTS)
	gfortran $(OBJECTS) -o main.exe

%.o : %.f90
	gfortran -c $<

clean:
	rm -f $(OBJECTS) main.exe
```

Finally we add a help message so that *make help* says something useful:

```
# $UWHPSC/codes/fortran/multifile1/Makefile6

OBJECTS = main.o sub1.o sub2.o
.PHONY: clean help

output.txt: main.exe
	./main.exe > output.txt

main.exe: $(OBJECTS)
	gfortran $(OBJECTS) -o main.exe

%.o : %.f90
	gfortran -c $<

clean:
	rm -f $(OBJECTS) main.exe

help:
	@echo "Valid targets:"
	@echo "  main.exe"
	@echo "  main.o"
	@echo "  sub1.o"
	@echo "  sub2.o"
	@echo "  clean:  removes .o and .exe files"
```

Fancier things are also possible, for example automatically detecting all the *.f90* files in the directory to construct the list of *SOURCES* and *OBJECTS*:

```
# $UWHPSC/codes/fortran/multifile1/Makefile7

```

```
SOURCES = $(wildcard *.f90)
OBJECTS = $(subst .f90,.o,$(SOURCES))

.PHONY: test

test:
        @echo "Sources are: " $(SOURCES)
        @echo "Objects are: " $(OBJECTS)
```

7.1.1 Further reading

- http://software-carpentry.org/4_0/make/
- *Makefile references* in bibliography.
- remake, a make debugger

7.2 Special functions

There are many functions that are used so frequently that they are given special names. Familiar examples are sin, cos, sqrt, exp, log, etc.

Most programming languages have build-in (intrinsic) functions with these same names that can be called to compute the value of the function for arbitrary arguments.

But "under the hood" these functions must be evaluated somehow. The basic arithmetic units of a computer are only capable of doing very basic arithmetic in hardware: addition, subtraction, multiplication, and division. Every other function must be *approximated* by applying some sequence of arithmetic operations.

There are several ways this might be done....

7.2.1 Taylor series expansions

A finite number of terms in the Taylor series expansion of a function gives a polynomial that approximates the desired function. Polynomials can be evaluated at any point using basic arithmetic.

For example, the section on *Fortran examples: Taylor series* discusses an implementation of the Taylor series approximation to the exponential function,

$$exp(x) = 1 + x + \frac{1}{2}x^2 + \frac{1}{6}x^3 + \cdots$$

If this is truncated after the 4 terms shown, for example, we obtain a polynomial of degree 3 that is easily evaluated at any point.

Some other Taylor series that can be used to approximate functions:

$$sin(x) = x - \frac{1}{3!}x^3 + \frac{1}{5!}x^5 - \frac{1}{7!}x^7 + \cdots$$
$$cos(x) = 1 - \frac{1}{2}x^2 + \frac{1}{4!}x^4 - \frac{1}{6!}x^6 + \cdots$$

7.2.2 Newton's method for the square root

One way to evaluate the square root function is to use *Newton's method*, a general procedure for estimating a *zero* of a function $f(x)$, i.e. a value x^* for which $f(x^*) = 0$.

The square root of any value $a > 0$ is a zero of the function $f(x) = x^2 - a$. (This function has two zeros, at $\pm\sqrt{a}$.

Newton's method is based on taking a current estimate x_k to x^* and (hopefully) improving it by setting

$$x_{k+1} = x_k - \delta_k$$

The increment δ_k is determined by *linearizing* the function $f(x)$ about the current esimate x_k using only the linear term in the Taylor series expansion. This linear function $G_k(x)$ is

$$G_k(x) = f(x_k) + f'(x_k)(x - x_k).$$

The next point x_{k+1} is set to be the zero of this linear function, which is trivial to find, and so

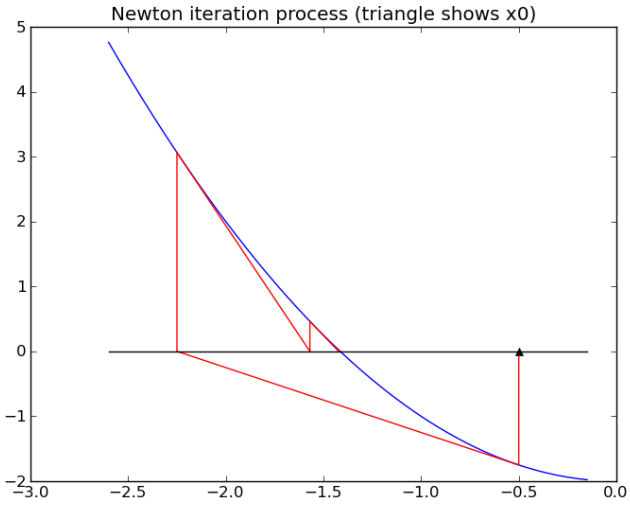
$$\delta_k = f(x_k)/f'(x_k).$$

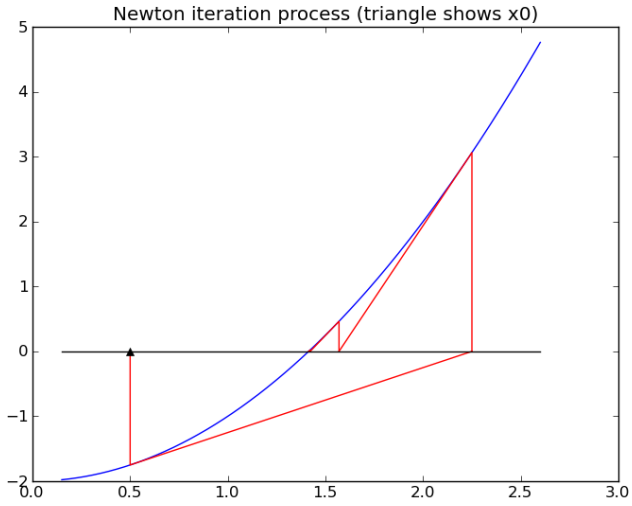
Geometrically, this means that we move along the tangent line to $f(x)$ from the point $(x_k, f(x_k))$ to the point where this line hits the x-axis, at $(x_{k+1}, 0)$. See the figures below.

There are several potential difficulties with this approach, e.g.

- To get started we require an *initial guess* x_0.
- The method may converge to a zero from some starting points but not converge at all from others, or may converge to different zeros depending on where we start.

For example, applying Newton's method to find a zero of $f(x) = x^2 - 2$ coverges to $\sqrt{2} \approx 1.414$ if we start at $x_0 = 0.5$, but convergest o $-\sqrt{2} \approx -1.414$ if we start at $x_0 = -0.5$ as illustrated in the figures below.





An advantage of Newton's method is that it is guaranteed to converge to a root provided the function $f(x)$ is smooth enough and the starting guess x_0 is sufficiently close to the zero. Moreover, in general one usually observes *quadratic convergence*. This means that once we get close to the zero, the error roughly satisfies

$$|x_{k+1} - x^*| \approx C|x_k - x^*|^2$$

The error is roughly squared in each step. In practice this means that once you have 2 correct digits in the solution, the next few steps will produce approximations with roughly 4, 8, and 16 correct digits (doubling each time), and so it rapidly converges to full machine precision.

For example, the approximations to $\sqrt{2}$ generated by Newton's method starting at $x_0 = 0.5$ are:

```
k,  x,  f(x):    1     0.500000000000000E+00    -0.175000E+01
k,  x,  f(x):    2     0.225000000000000E+01     0.306250E+01
k,  x,  f(x):    3     0.156944444444444E+01     0.463156E+00
k,  x,  f(x):    4     0.142189036381514E+01     0.217722E-01
k,  x,  f(x):    5     0.141423428594007E+01     0.586155E-04
k,  x,  f(x):    6     0.141421356252493E+01     0.429460E-09
k,  x,  f(x):    7     0.141421356237310E+01     0.444089E-15
```

The last value is correct to all digits.

There are many other methods that have been developed for finding zeros of functions, and a number of software packages that are designed to be more robust than Newton's method (less likely to fail to converge) while still converging very rapidly.

7.3 Timing code

7.3.1 Unix time command

There is a built in command *time* that can be used to time other commands. Just type *time command* at the prompt, e.g.:

```
$ time ./a.out
<output from code>

real    0m5.279s
```

```
user    0m1.915s
sys     0m0.006s
```

This executes the command *./a.out* in this case (running a Fortran executable) and then prints information about the time required to execute, where, roughly speaking, *real* is the wall-clock time, *user* is the time spent executing the user's program, and *sys* is the time spent on system tasks required by the program.

There may be a big difference between the *real* time and the sum of the other two times if the computer is simulataneously executing many other tasks and your program is only getting some of its attention.

Using *time* does not allow you to examine how much CPU time different parts of the code required, for example.

7.3.2 Fortran timing utilities

There are two Fortran intrinsic functions that are very useful.

system_clock tells the elapsed wall time between two successive calls, and might be used as follows:

```
integer(kind=8) :: tclock1, tclock2, clock_rate
real(kind=8) :: elapsed_time

call system_clock(tclock1)

<code to be timed>

call system_clock(tclock2, clock_rate)
elapsed_time = float(tclock2 - tclock1) / float(clock_rate)
```

cpu_time tells the CPU time used between two successive calls:

```
real(kind=8) :: t1, t2, elapsed_cpu_time

call cpu_time(t1)

<code to be timed>

call cpu_time(t2)
elapsed_cpu_time = t2 - t1
```

Here is an example code that uses both, and tests matrix-matrix multiply.

```
! $UWHPSC/codes/fortran/optimize/timings.f90

! Illustrate timing utilities in Fortran.
!  system_clock can be used to compute elapsed time between
!       two calls (wall time)
!  cpu_time can be used to compute CPU time used between two calls.

! Try compiling with different levels of optimization, e.g. -O3


program timings

    implicit none
    integer, parameter :: ntests = 20
    integer :: n
    real(kind=8), allocatable, dimension(:,:) :: a,b,c
    real(kind=8) :: t1, t2, elapsed_time
    integer(kind=8) :: tclock1, tclock2, clock_rate
```

```
    integer :: i,j,k,itest

    call system_clock(tclock1)

    print *, "Will multiply n by n matrices, input n: "
    read *, n

    allocate(a(n,n), b(n,n), c(n,n))

    ! fill a and b with 1's just for demo purposes:
    a = 1.d0
    b = 1.d0

    call cpu_time(t1)    ! start cpu timer
    do itest=1,ntests
        do j = 1,n
            do i = 1,n
                c(i,j) = 0.d0
                do k=1,n
                    c(i,j) = c(i,j) + a(i,k)*b(k,j)
                    enddo
                enddo
            enddo
        enddo

    call cpu_time(t2)    ! end cpu timer
    print 10, ntests, t2-t1
 10 format("Performed ",i4, " matrix multiplies: CPU time = ",f12.8, " seconds")


    call system_clock(tclock2, clock_rate)
    elapsed_time = float(tclock2 - tclock1) / float(clock_rate)
    print 11, elapsed_time
 11 format("Elapsed time = ",f12.8, " seconds")

end program timings
```

Note that the matrix-matrix product is computed 20 times over to give a better estimate of the timings.

You might want to experiment with this code and various sizes of the matrices and optimization levels. Here are a few sample results on a MacBook Pro.

First, with no optimization and 200×200 matrices:

```
$ gfortran timings.f90
$ ./a.out
 Will multiply n by n matrices, input n:
200
Performed   20 matrix multiplies: CPU time =   0.81523600 seconds
Elapsed time =   5.94083357 seconds
```

Note that the elapsed time include the time required to type in the response to the prompt for n, as well as the time required to allocate and initialize the matrices, whereas the CPU time is just for the matrix multiplication loops.

Trying a larger 400×400 case gives:

```
$ ./a.out
 Will multiply n by n matrices, input n:
400
```

```
Performed   20 matrix multiplies: CPU time =   7.33721500 seconds
Elapsed time =   9.87114525 seconds
```

Since computing the product of $n \times n$ matrices takes $O(n^3)$ flops, we expect this to take about 8 times as much CPU time as the previous test. This is roughly what we observe.

Doubling the size again gives requires much more that 8 times as long however:

```
$ ./a.out
 Will multiply n by n matrices, input n:
800
Performed   20 matrix multiplies: CPU time =  90.49682200 seconds
Elapsed time =  93.98917389 seconds
```

Note that 3 matrices that are 400×400 require 3.84 MB of memory, whereas three 800×800 matrices require 15.6 MB. Since the MacBook used for this experiment has only 6 MB of L3 cache, the data no longer fit in cache.

Compiling with optimization

If we recompile with the -O3 optimization flag, the last two experiments give these results:

```
$ gfortran -O3 timings.f90
$ ./a.out
 Will multiply n by n matrices, input n:
400
Performed   20 matrix multiplies: CPU time =   1.39002200 seconds
Elapsed time =   3.58041191 seconds
```

and

```
$ ./a.out
 Will multiply n by n matrices, input n:
800
Performed   20 matrix multiplies: CPU time =  66.39167200 seconds
Elapsed time =  68.29921722 seconds
```

Both have sped up relative to the un-optimized code, the first much more dramatically.

7.3.3 Timing OpenMP code

The code in *$UWHPSC/codes/openmp/timings.f90* shows an analogous code for matrix multiplication using OpenMP.

The code has been slightly modified so that *system_clock* is only timing the inner loops in order to illustrate that *cpu_time* now computes the sum of the CPU time of all threads.

Here's one sample result:

```
$ gfortran -fopenmp -O3 timings.f90
$ ./a.out
 Using OpenMP, how many threads?
4
 Will multiply n by n matrices, input n:
400
Performed   20 matrix multiplies: CPU time =   1.99064000 seconds
Elapsed time =   0.58711302 seconds
```

Note that the CPU time reported is nearly 2 seconds but the elapsed time is only 0.58 seconds in this case when 4 threads are being used.

The total CPU time is slightly more than the previous code that did not use OpenMP, but the wall time is considerably less.

For 800×800 matrices there is a similar speedup:

```
$ ./a.out
 Using OpenMP, how many threads?
4
 Will multiply n by n matrices, input n:
800
Performed   20 matrix multiplies: CPU time =  79.70573500 seconds
Elapsed time =  21.37633133 seconds
```

Here is the code:

```
! $UWHPSC/codes/fortran/optimize/timings1.f90

! Illustrate timing utilities in Fortran.
!  system_clock can be used to compute elapsed time between
!       two calls (wall time)
!  cpu_time can be used to compute CPU time used between two calls.

! Try compiling with different levels of optimization, e.g. -O3


program timings1

    use omp_lib

    implicit none
    integer, parameter :: ntests = 20
    integer :: n, nthreads
    real(kind=8), allocatable, dimension(:,:) :: a,b,c
    real(kind=8) :: t1, t2, elapsed_time
    integer(kind=8) :: tclock1, tclock2, clock_rate
    integer :: i,j,k,itest

    ! Specify number of threads to use:
    !$ print *, "Using OpenMP, how many threads? "
    !$ read *, nthreads
    !$ call omp_set_num_threads(nthreads)

    print *, "Will multiply n by n matrices, input n: "
    read *, n

    allocate(a(n,n), b(n,n), c(n,n))

    ! fill a and b with 1's just for demo purposes:
    a = 1.d0
    b = 1.d0

    call system_clock(tclock1)  ! start wall timer

    call cpu_time(t1)    ! start cpu timer
    do itest=1,ntests
        !$omp parallel do private(i,k)
        do j = 1,n
            do i = 1,n
                c(i,j) = 0.d0
```

```
                do k=1,n
                    c(i,j) = c(i,j) + a(i,k)*b(k,j)
                    enddo
                enddo
            enddo
        enddo

    call cpu_time(t2)    ! end cpu timer
    print 10, ntests, t2-t1
 10 format("Performed ",i4, " matrix multiplies: CPU time = ",f12.8, " seconds")


    call system_clock(tclock2, clock_rate)
    elapsed_time = float(tclock2 - tclock1) / float(clock_rate)
    print 11, elapsed_time
 11 format("Elapsed time = ",f12.8, " seconds")

end program timings1
```

7.4 Linear Algebra software

7.4.1 The BLAS

The Basic Linear Algebra Subroutines are extensively used in LAPACK, in other linear algebra packages, and elsewhere.

There are three levels of BLAS:

- Level 1: Scalar and vector operations
- Level 2: Matrix-vector operations
- Level 3: Matrix-matrix operations

For general information about BLAS see http://www.netlib.org/blas/faq.html.

Optimized versions of the BLAS are available for many computer architectures. See

- OpenBLAS
- GotoBLAS

See also:

- Automatically Tuned Linear Algebra Software (ATLAS)
- BLACS Basic Linear Algebra Communication Subprograms with message passing.
- PSBLAS – Parallel sparse BLAS.

7.4.2 LAPACK

- LAPACK
- LAPACK User's Guide
- http://en.wikipedia.org/wiki/LAPACK
- ScaLAPACK for parallel distributed memory machines

To install BLAS and LAPACK to work with gfortran, see:

- http://gcc.gnu.org/wiki/GfortranBuild

On some linux systems, including the VM for the class, you can install both BLAS and LAPACK via:

```
$ sudo apt-get install liblapack-dev
```

7.4.3 Direct methods for sparse systems

Although iterative methods are often used for sparse systems, there are also excellent software packages for direct methods (such as Gaussian elimination):

- UMFPACK
- SuperLU
- MUMPS
- Pardiso

7.4.4 Other references

- lapack_examples for some examples.
- Recent list of freely available linear algebra software

7.5 Random number generators

For some description and documentation of pseudo-random number generators, see:

- Wikipedia
- Fortran RANDOM_NUMBER subroutine
- Fortran RANDOM_SEED subroutine
- Python random
- More Python options
- OpenCL PRNG

CHAPTER

EIGHT

APPLICATIONS

8.1 Numerical methods for the Poisson problem

The steady state diffusion equation gives rise to a *Poisson problem*

$u_{xx} + u_{yy} = -f(x, y)$

where $f(x, y)$ is the source term. In the simplest case $f(x, y) = 0$ this reduces to *Laplace's equation*. This must be augmented with boundary conditions around the edge of some two-dimensional region. *Dirichlet boundary conditions* consist of specifying the solution $u(x, y)$ at all points around the boundary. *Neumann boundary conditions* consist of specifying the normal derivative (i.e. the direction derivative of the solution in the direction orthogonal to the boundary) and are used in physical situations where the if the flux of heat or the diffused quantity is known along the boundary rather than the value of the solution itself (for example an *insulated boundary* has no flux and the normal derivative is zero). We will only study Dirichlet problems, where u itself is known at boundary points. We will also concentrate on problems in a rectangular domain $a_x < x < b_x$ and $a_y < y < b_y$, in which case it is natural to discretize on a *Cartesian grid* aligned with the axes.

The Poisson problem can be discretized on a two-dimensional Cartesian grid with equal grid spacing h in the x and y directions as

$U_{i-1,j} + U_{i+1,j} + U_{i,j-1} + U_{i,j+1} - 4u_{ij} = -h^2 f(x_i, y_j)$.

This gives a coupled system of equations with $n_x n_y$ unknowns, where it is assumed that $h(n_x + 1) = b_x - a_x$ and $h(n_y + 1) = b_y - a_y$. The linear system has a very sparse coefficient matrix since each of the $n_x n_y$ rows has at most 5 nonzero entries.

If the boundary data varies smoothly around the boundary then it can be shown that solving this linear system gives an approximate solution of the partial differential equation that is $\mathcal{O}(h^2)$ accurate at each point. There are many books that contain much more about the development and analysis of such finite difference methods.

8.1.1 Iterative methods for the Poisson problem

Simple iterative methods such as Jacobi, Gauss-Siedel, and Successive Over-Relaxation (SOR) are discussed in the lectures and used as examples for implementations in OpenMP and MPI. For three implementation of Jacobi in one space dimension, see

- *Jacobi iteration using OpenMP with parallel do constructs*
- *Jacobi iteration using OpenMP with coarse-grain parallel block*
- *Jacobi iteration using MPI*

A sample implementation of Jacobi in two space dimensions can be found in *$UWHPSC/lectures/lecture1*.

8.1.2 Monte Carlo methods for the steady state diffusion equation

Solving the linear system described above would give an approximate solution to the Poisson problem at each point on the grid. Suppose we only want to approximate the solution at a single point (x_0, y_0) for some reason. Is there a way to estimate this without solving the system for all values U_{ij}? Not easily from the linear system, but there are other approaches that might be used.

We will consider a Monte Carlo approach in which a large number of *random walks* starting from the point of interest are used to estimate the solution. See *Random number generators* for a discussion of random number generators and Monte Carlo methods more generally.

We will assume there is no source term, $f(x, y) = 0$ so that we are solving Laplace's equation. The random walk solution is more complicated if there is a source term.

A random walk starting at some point (x_0, y_0) wanders randomly in the domain until it hits the boundary at some point. We do this many times over and keep track of the boundary value given for $u(x, y)$ at the point where each walk hits the boundary. It can be shown that if we do this for a large number of walks and average the results, this converges to the desired solution value $u(x_0, y_0)$. Note that we expect more walks to hit the boundary at parts of the boundary near (x_0, y_0) than at points further away, so the boundary conditions at such points will have more influence on the solution. This is intuitively what we expect for a steady state solution of a diffusion or heat conduction problem.

To implement this numerically we will consider the simplification of a *lattice random walk*, in which we put down a grid on the domain as in the finite difference discretization and allow the random walk to only go in one of 4 directions in each time step, from a point on the grid to one of its four neighbors. For isotropic diffusion as we are considering, we can define a random walk by choosing 1 of the four neighbors with equal probability in each step.

The code *$UWHPSC/codes/project/laplace_mc.py* illustrates this. Run this code with *plot_walk = True* to see plots of a few random walks on a coarse grid, or with *plot_walk = False* to report the solution after many random walks on a finer grid.

With this lattice random walk we do not expect the approximate solution to converge to the true solution of the PDE, as the number of trials increases. Instead we expect it to converge to the solution of the linear system determined by the finite difference method described above. In other words if we choose $(x_0, y_0) = (x_i, y_j)$ for some grid point (i, j) then we expect the Monte Carlo solution to converge to U_{ij} rather than to $u(x_i, y_j)$.

Why does this work? Here's one way to think about it. Suppose doing this random walk starting at (x_i, y_j) converges to some value E_{ij}, the expected value of u at the boundary hit when starting a random walk at this point. If (x_i, y_j) is one of the boundary points then $E_{ij} = U_{ij}$ since we immediately hit the boundary with zero steps, so every random walk starting at this point returns u at this point. On the other hand, if (x_i, y_j) is an interior point, then after a single step of the random walk we will be at one of the four neighbors. Continuing our original random walk from this point is equivalent to starting a new random walk at this point. So for example any random walk that first takes a step to the right from (x_i, y_j) to (x_{i+1}, y_j) has the same expected boundary value as obtained from all random walks starting at (x_{i+1}, y_j), i.e. the value $E_{i+1,j}$. But only 1/4 of the random walks starting at (x_i, y_j) go first to the right. So the expected value over all walks starting at (x_i, y_j) is expected to be the average of the expected value when starting at any of the 4 neighbors. In other words,

$$E_{ij} = \tfrac{1}{4}(E_{i-1,j} + E_{i+1,j} + E_{i,j-1} + E_{i,j+1})$$

But this means E_{ij} satisfies the same linear system of equations as U_{ij} (and also the same boundary conditions), and hence must be the same.

8.2 Jacobi iteration using OpenMP with *parallel do* constructs

The code below implements Jacobi iteration for solving the linear system arising from the steady state heat equation with a simple application of *parallel do* loops using OpenMP.

Compare to:

- *Jacobi iteration using OpenMP with coarse-grain parallel block*
- *Jacobi iteration using MPI*

The code:

```
! $UWHPSC/codes/openmp/jacobi1d_omp1.f90
!
! Jacobi iteration illustrating fine grain parallelism with OpenMP.
!
! Several omp parallel do loops are used.  Each time threads will be
! forked and the compiler will decide how to split up the loop.

program jacobi1d_omp1
    use omp_lib
    implicit none
    integer :: n, nthreads
    real(kind=8), dimension(:), allocatable :: x,u,uold,f
    real(kind=8) :: alpha, beta, dx, tol, dumax
    real(kind=8), intrinsic :: exp
    real(kind=8) :: t1,t2
    integer :: i,iter,maxiter

    ! Specify number of threads to use:
    nthreads = 2
    !$ call omp_set_num_threads(nthreads)
    !$ print "('Using OpenMP with ',i3,' threads')", nthreads

    print *, "Input n ... "
    read *, n

    ! allocate storage for boundary points too:
    allocate(x(0:n+1), u(0:n+1), uold(0:n+1), f(0:n+1))

    open(unit=20, file="heatsoln.txt", status="unknown")

    call cpu_time(t1)

    ! grid spacing:
    dx = 1.d0 / (n+1.d0)

    ! boundary conditions:
    alpha = 20.d0
    beta = 60.d0

    !$omp parallel do
    do i=0,n+1
        ! grid points:
        x(i) = i*dx
        ! source term:
        f(i) = 100.*exp(x(i))
        ! initial guess:
        u(i) = alpha + x(i)*(beta-alpha)
        enddo

    ! tolerance and max number of iterations:
    tol = 0.1 * dx**2
    print *, "Convergence tolerance: tol = ",tol
    maxiter = 100000
    print *, "Maximum number of iterations: maxiter = ",maxiter
```

```
55
56      ! Jacobi iteratation:
57
58      uold = u  ! starting values before updating
59
60      do iter=1,maxiter
61          dumax = 0.d0
62          !$omp parallel do reduction(max : dumax)
63          do i=1,n
64              u(i) = 0.5d0*(uold(i-1) + uold(i+1) + dx**2*f(i))
65              dumax = max(dumax, abs(u(i)-uold(i)))
66              enddo
67          if (mod(iter,10000)==0) then
68              print *, iter, dumax
69              endif
70          ! check for convergence:
71          if (dumax .lt. tol) exit
72
73          !$omp parallel do
74          do i=1,n
75              uold(i) = u(i)    ! for next iteration
76              enddo
77          enddo
78
79          call cpu_time(t2)
80          print '("CPU time = ",f12.8, " seconds")', t2-t1
81
82          print *, "Total number of iterations: ",iter
83
84      write(20,*) "           x                  u"
85      do i=0,n+1
86          write(20,'(2e20.10)'), x(i), u(i)
87          enddo
88
89      print *, "Solution is in heatsoln.txt"
90
91
92      close(20)
93
94  end program jacobi1d_omp1
```

8.3 Jacobi iteration using OpenMP with coarse-grain *parallel* block

The code below implements Jacobi iteration for solving the linear system arising from the steady state heat equation with a single *parallel* block. Work is split up manually between threads.

Compare to:

- *Jacobi iteration using OpenMP with parallel do constructs*
- *Jacobi iteration using MPI*

The code:

```
1  ! $UWHPSC/codes/openmp/jacobi1d_omp2.f90
2  !
3  ! Domain decomposition version of Jacobi iteration illustrating
```

```
! coarse grain parallelism with OpenMP.
!
! The grid points are split up into nthreads disjoint sets and each thread
! is assigned one set that it updates for all iterations.

program jacobi1d_omp2
    use omp_lib
    implicit none
    real(kind=8), dimension(:), allocatable :: x,u,uold,f
    real(kind=8) :: alpha, beta, dx, tol, dumax, dumax_thread
    real(kind=8), intrinsic :: exp
    real(kind=8) :: t1,t2
    integer :: n, nthreads, points_per_thread,thread_num
    integer :: i,iter,maxiter,istart,iend,nprint

    ! Specify number of threads to use:
    nthreads = 1       ! need this value in serial mode
    !$ nthreads = 2
    !$ call omp_set_num_threads(nthreads)
    !$ print "('Using OpenMP with ',i3,' threads')", nthreads

    nprint = 10000 ! print dumax every nprint iterations

    print *, "Input n ... "
    read *, n

    ! allocate storage for boundary points too:
    allocate(x(0:n+1), u(0:n+1), uold(0:n+1), f(0:n+1))

    open(unit=20, file="heatsoln.txt", status="unknown")

    call cpu_time(t1)

    ! grid spacing:
    dx = 1.d0 / (n+1.d0)

    ! boundary conditions:
    alpha = 20.d0
    beta = 60.d0

    ! tolerance and max number of iterations:
    tol = 0.1 * dx**2
    print *, "Convergence tolerance: tol = ",tol
    maxiter = 100000
    print *, "Maximum number of iterations: maxiter = ",maxiter

    ! Determine how many points to handle with each thread.
    ! Note that dividing two integers and assigning to an integer will
    ! round down if the result is not an integer.
    ! This, together with the min(...) in the definition of iend below,
    ! insures that all points will get distributed to some thread.
    points_per_thread = (n + nthreads - 1) / nthreads
    print *, "points_per_thread = ",points_per_thread


    ! Start of the parallel block...
    ! ------------------------------

```

```
    ! This is the only time threads are forked in this program:

    !$omp parallel private(thread_num, iter, istart, iend, i, dumax_thread)

    thread_num = 0      ! needed in serial mode
    !$ thread_num = omp_get_thread_num()    ! unique for each thread

    ! Determine start and end index for the set of points to be
    ! handled by this thread:
    istart = thread_num * points_per_thread + 1
    iend = min((thread_num+1) * points_per_thread, n)

    !$omp critical
    print '("Thread ",i2," will take i = ",i6," through i = ",i6)', &
          thread_num, istart, iend
    !$omp end critical

    ! Initialize:
    ! -----------

    ! each thread sets part of these arrays:
    do i=istart, iend
        ! grid points:
        x(i) = i*dx
        ! source term:
        f(i) = 100.*exp(x(i))
        ! initial guess:
        u(i) = alpha + x(i)*(beta-alpha)
        enddo

    ! boundary conditions need to be added:
    u(0) = alpha
    u(n+1) = beta

    uold = u   ! initialize, including boundary values


    ! Jacobi iteratation:
    ! -------------------


    do iter=1,maxiter

        ! initialize uold to u (note each thread does part!)
        uold(istart:iend) = u(istart:iend)

        !$omp single
        dumax = 0.d0     ! global max initialized by one thread
        !$omp end single

        ! Make sure all of uold is initialized before iterating
        !$omp barrier
        ! Make sure uold is consitent in memory:
        !$omp flush

        dumax_thread = 0.d0   ! max seen by this thread
        do i=istart,iend
            u(i) = 0.5d0*(uold(i-1) + uold(i+1) + dx**2*f(i))
```

```
            dumax_thread = max(dumax_thread, abs(u(i)-uold(i)))
            enddo

        !$omp critical
        ! update global dumax using value from this thread:
        dumax = max(dumax, dumax_thread)
        !$omp end critical

        ! make sure all threads are done with dumax:
        !$omp barrier

        !$omp single
        ! only one thread will do this print statement:
        if (mod(iter,nprint)==0) then
            print '("After ",i8," iterations, dumax = ",d16.6,/)', iter, dumax
            endif
        !$omp end single

        ! check for convergence:
        ! note that all threads have same value of dumax
        ! at this point, so they will all exit on the same iteration.

        if (dumax .lt. tol) exit

        ! need to synchronize here so no thread resets dumax = 0
        ! at start of next iteration before all have done the test above.
        !$omp barrier

        enddo

    print '("Thread number ",i2," finished after ",i9, &
            " iterations, dumax = ", e16.6)', &
          thread_num,iter,dumax

    !$omp end parallel

    call cpu_time(t2)
    print '("CPU time = ",f12.8, " seconds")', t2-t1


    ! Write solution to heatsoln.txt:
    write(20,*) "          x                  u"
    do i=0,n+1
        write(20,'(2e20.10)'), x(i), u(i)
        enddo

    print *, "Solution is in heatsoln.txt"

    close(20)

end program jacobi1d_omp2
```

8.4 Jacobi iteration using MPI

The code below implements Jacobi iteration for solving the linear system arising from the steady state heat equation using MPI. Note that in this code each process, or task, has only a portion of the arrays and must exchange boundary

data using message passing.

Compare to:

- *Jacobi iteration using OpenMP with parallel do constructs*
- *Jacobi iteration using OpenMP with coarse-grain parallel block*

The code:

```
! $UWHSPC/codes/mpi/jacobi1d_mpi.f90
!
! Domain decomposition version of Jacobi iteration illustrating
! coarse grain parallelism with MPI.
!
! The one-dimensional Poisson problem is solved, u''(x) = -f(x)
! with u(0) = alpha and u(1) = beta.
!
! The grid points are split up into ntasks disjoint sets and each task
! is assigned one set that it updates for all iterations.  The tasks
! correspond to processes.
!
! The task (or process) number is called "me" in this code for brevity
! rather than proc_num.
!
! Note that each task allocates only as much storage as needed for its
! portion of the arrays.
!
! Each iteration, boundary values at the edge of each grid must be
! exchanged with the neighbors.


program jacobi1d_mpi
    use mpi

    implicit none

    integer, parameter :: maxiter = 100000, nprint = 5000
    real (kind=8), parameter :: alpha = 20.d0, beta = 60.d0

    integer :: i, iter, istart, iend, points_per_task, itask, n
    integer :: ierr, ntasks, me, req1, req2
    integer, dimension(MPI_STATUS_SIZE) :: mpistatus
    real (kind = 8), dimension(:), allocatable :: f, u, uold
    real (kind = 8) :: x, dumax_task, dumax_global, dx, tol

    ! Initialize MPI; get total number of tasks and ID of this task
    call mpi_init(ierr)
    call mpi_comm_size(MPI_COMM_WORLD, ntasks, ierr)
    call mpi_comm_rank(MPI_COMM_WORLD, me, ierr)

    ! Ask the user for the number of points
    if (me == 0) then
        print *, "Input n ... "
        read *, n
    end if
    ! Broadcast to all tasks; everybody gets the value of n from task 0
    call mpi_bcast(n, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)

    dx = 1.d0/real(n+1, kind=8)
```

```
    tol = 0.1d0*dx**2

    ! Determine how many points to handle with each task
    points_per_task = (n + ntasks - 1)/ntasks
    if (me == 0) then   ! Only one task should print to avoid clutter
        print *, "points_per_task = ", points_per_task
    end if

    ! Determine start and end index for this task's points
    istart = me * points_per_task + 1
    iend = min((me + 1)*points_per_task, n)

    ! Diagnostic: tell the user which points will be handled by which task
    print '("Task ",i2," will take i = ",i6," through i = ",i6)', &
        me, istart, iend


    ! Initialize:
    ! -----------

    ! This makes the indices run from istart-1 to iend+1
    ! This is more or less cosmetic, but makes things easier to think about
    allocate(f(istart-1:iend+1), u(istart-1:iend+1), uold(istart-1:iend+1))

    ! Each task sets its own, independent array
    do i = istart, iend
        ! Each task is a single thread with all its variables private
        ! so re-using the scalar variable x from one loop iteration to
        ! the next does not produce a race condition.
        x = dx*real(i, kind=8)
        f(i) = 100.d0*exp(x)                ! Source term
        u(i) = alpha + x*(beta - alpha)     ! Initial guess
    end do

    ! Set boundary conditions if this task is keeping track of a boundary
    ! point
    if (me == 0)        u(istart-1) = alpha
    if (me == ntasks-1) u(iend+1)   = beta


    ! Jacobi iteratation:
    ! -------------------

    do iter = 1, maxiter
        uold = u

        ! Send endpoint values to tasks handling neighboring sections
        ! of the array.  Note that non-blocking sends are used; note
        ! also that this sends from uold, so the buffer we're sending
        ! from won't be modified while it's being sent.
        !
        ! tag=1 is used for messages sent to the left
        ! tag=2 is used for messages sent to the right

        if (me > 0) then
            ! Send left endpoint value to process to the "left"
            call mpi_isend(uold(istart), 1, MPI_DOUBLE_PRECISION, me - 1, &
                1, MPI_COMM_WORLD, req1, ierr)
```

```
        end if
        if (me < ntasks-1) then
            ! Send right endpoint value to process on the "right"
            call mpi_isend(uold(iend), 1, MPI_DOUBLE_PRECISION, me + 1, &
                2, MPI_COMM_WORLD, req2, ierr)
        end if

        ! Accept incoming endpoint values from other tasks.  Note that
        ! these are blocking receives, because we can't run the next step
        ! of the Jacobi iteration until we've received all the
        ! incoming data.

        if (me < ntasks-1) then
            ! Receive right endpoint value
            call mpi_recv(uold(iend+1), 1, MPI_DOUBLE_PRECISION, me + 1, &
                1, MPI_COMM_WORLD, mpistatus, ierr)
        end if
        if (me > 0) then
            ! Receive left endpoint value
            call mpi_recv(uold(istart-1), 1, MPI_DOUBLE_PRECISION, me - 1, &
                2, MPI_COMM_WORLD, mpistatus, ierr)
        end if

        dumax_task = 0.d0    ! Max seen by this task

        ! Apply Jacobi iteration on this task's section of the array
        do i = istart, iend
            u(i) = 0.5d0*(uold(i-1) + uold(i+1) + dx**2*f(i))
            dumax_task = max(dumax_task, abs(u(i) - uold(i)))
        end do

        ! Take global maximum of dumax values
        call mpi_allreduce(dumax_task, dumax_global, 1, MPI_DOUBLE_PRECISION, &
            MPI_MAX, MPI_COMM_WORLD, ierr)
        ! Note that this MPI_ALLREDUCE call acts as an implicit barrier,
        ! since no process can return from it until all processes
        ! have called it.  Because of this, after this call we know
        ! that all the send and receive operations initiated at the
        ! top of the loop have finished -- all the MPI_RECV calls have
        ! finished in order for each process to get here, and if the
        ! MPI_RECV calls have finished, the corresponding MPI_ISEND
        ! calls have also finished.  Thus we can safely modify uold
        ! again.

        ! Also periodically report progress to the user
        if (me == 0) then
            if (mod(iter, nprint)==0) then
                print '("After ",i8," iterations, dumax = ",d16.6,/)', &
                    iter, dumax_global
            end if
        end if

        ! All tasks now have dumax_global, and can check for convergence
        if (dumax_global < tol) exit
    end do

    print '("Task number ",i2," finished after ",i9," iterations, dumax = ",&
            e16.6)', me, iter, dumax_global
```

```
167
168
169     ! Output result:
170     ! --------------
171
172     ! Note: this only works if all processes share a file system
173     ! and can all open and write to the same file!
174
175     ! Synchronize to keep the next part from being non-deterministic
176     call mpi_barrier(MPI_COMM_WORLD, ierr)
177
178     ! Have each task output to a file in sequence, using messages to
179     ! coordinate
180
181     if (me == 0) then    ! Task 0 goes first
182         ! Open file for writing, replacing any previous version:
183         open(unit=20, file="heatsoln.txt", status="replace")
184         write(20,*) "          x                  u"
185         write(20, '(2e20.10)') 0.d0, u(0)    ! Boundary value at left end
186
187         do i = istart, iend
188             write(20, '(2e20.10)') i*dx, u(i)
189         end do
190
191         close(unit=20)
192         ! Closing the file should guarantee that all the ouput
193         ! will be written to disk.
194         ! If the file isn't closed before the next process starts writing,
195         ! output may be jumbled or missing.
196
197         ! Send go-ahead message to next task
198         ! Only the fact that the message was sent is important, not its contents
199         ! so we send the special address MPI_BOTTOM and length 0.
200         ! tag=4 is used for this message.
201
202         if (ntasks > 1) then
203             call mpi_send(MPI_BOTTOM, 0, MPI_INTEGER, 1, 4, &
204                           MPI_COMM_WORLD, ierr)
205             endif
206
207     else
208         ! Wait for go-ahead message from previous task
209         call mpi_recv(MPI_BOTTOM, 0, MPI_INTEGER, me - 1, 4, &
210                       MPI_COMM_WORLD, mpistatus, ierr)
211         ! Open file for appending; do not destroy previous contents
212         open(unit=20, file="heatsoln.txt", status="old", access="append")
213         do i = istart, iend
214             write(20, '(2e20.10)') i*dx, u(i)
215         end do
216
217         ! Boundary value at right end:
218         if (me == ntasks - 1) write(20, '(2e20.10)') 1.d0, u(iend+1)
219
220         ! Flush all pending writes to disk
221         close(unit=20)
222
223         if (me < ntasks - 1) then
224             ! Send go-ahead message to next task
```

```
            call mpi_send(MPI_BOTTOM, 0, MPI_INTEGER, me + 1, 4, &
                          MPI_COMM_WORLD, ierr)
        end if
    end if

    ! Notify the user when all tasks have finished writing
    if (me == ntasks - 1) print *, "Solution is in heatsoln.txt"

    ! Close out MPI
    call mpi_finalize(ierr)

end program jacobi1d_mpi
```

CHAPTER

NINE

REFERENCES

9.1 Bibliography and further reading

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

9.1.1 Books

9.1.2 Other courses with useful slides or webpages

9.1.3 Other Links

9.1.4 Software

See also:

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

9.1.5 Virtual machine references

9.1.6 Sphinx references

9.1.7 Python:

9.1.8 Numerical Python references

9.1.9 Unix, bash references

9.1.10 Version control systems references

9.1.11 Git references

9.1.12 Mercurial references

9.1.13 Reproducibility references

9.1.14 Fortran references

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

9.1.15 Makefile references

9.1.16 Computer architecture references

9.1.17 Floating point arithmetic

9.1.18 Languages and compilers

9.1.19 OpenMP references

9.1.20 MPI references

See also [Gropp-Lusk-Skjellum-MPI] and [Snir-Dongarra-etal-MPI]

See *MPI* for more references.

9.1.21 Exa-scale computing

More will be added, check back later

[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

[Rauber-Ruenger] T. Rauber and G. Ruenger, *Parallel Programming For Multicore and Cluster Systems*, Springer, 2010 ... book ... ebook

[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

[Snir-Dongarra-etal-MPI] M. Snir, J. Dongarra, J. S. Kowalik, S. Huss-Lederman, S. W. Otto, D. W. Walker, *MPI: The Complete Reference (2-volume set)* , MIT Press, 2000.

[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.

[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.

[Eijkhout] V. Eijkhout, *Introduction to High-Performance Scientific Computing*, [link to book and slides]

[software-carpentry] Greg Wilson, http://software-carpentry.org/ See *Software Carpentry* for links to some useful sections.

[Reynolds-class] Dan Reynolds, SMU http://dreynolds.math.smu.edu/Courses/Math6370_Spring11/.

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

[Kloekner-Berger-NYU] Andreas Kloeckner and Marsha Berger, NYU course

[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/

[CosmicProject] links to open source Python software

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

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

[sphinx] http://sphinx-doc.org

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

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

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

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

[sphinx-examples] http://sphinx-doc.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-2.7-tutorial] http://docs.python.org/tutorial/

[Python-documentation] http://docs.python.org/2/contents.html

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

[IPython-documentation] http://ipython.org/documentation.html (With lots of links to other documentation and tutorials)

[IPython-notebook] http://ipython.org/ipython-doc/dev/interactive/htmlnotebook.html

[Python-pdb] Python debugger documentation

[IPython-book] Cyrille Rossant, *Learning IPython for Interactive Computing and Data Visualization*, Packt Publishing, 2013. http://ipython.rossant.net/.

[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

[Reynolds-unix] Dan Reynolds unix page has good links.

[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(Unix_shell)

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

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

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

[git-try] Online interactive tutorial

[git-tutorials] List of 10 tutorials http://sixrevisions.com/resources/git-tutorials-beginners/

[gitref] http://gitref.org/index.html

[git-book] Git Book http://git-scm.com/book/en/Getting-Started-Git-Basics

[github-help] Github help page: http://help.github.com/

[git-parable] http://tom.preston-werner.com/2009/05/19/the-git-parable.html

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

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

[sci-code-manifesto] Science Code Manifesto

[icerm-workshop] Links from a recent workshop on the topic

[winter-school] A recent Winter School on the topic in Geilo, Norway:

[Reynolds-fortran] Dan Reynolds fortran page http://dreynolds.math.smu.edu/Courses/Math6370_Spring11/fortran.html

[Shene-fortran] C.-K. Shene's Fortran 90 tutorial http://www.cs.mtu.edu/~shene/COURSES/cs201/NOTES/fortran.html

[Dodson-fortran] Zane Dodson's Fortran 90 tutorial http://www.cisl.ucar.edu/tcg/consweb/Fortran90/F90Tutorial/tutorial.html

[fortran-tutorials] Links to a few other tutorials http://gcc.gnu.org/wiki/Fortran%2095%20tutorials%20available%20online

[advanced-fortran] Kaiser, Advanced Fortran 90 http://www.sdsc.edu/~tkaiser/f90.html

[carpentry-make] http://software-carpentry.org/4_0/make/

[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

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

[Chapman-Jost] B. Chapman, G. Jost, R. van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*, MIT Press, 2007

[openmp-RR] Section 6.3 and beyond of *[Rauber-Ruenger]*

[openmp.org] http://openmp.org/wp/resources contains pointers to many books and tutorials.

[openmp-specs] http://openmp.org/wp/openmp-specifications/ has the latest official specifications

[openmp-llnl] https://computing.llnl.gov/tutorials/openMP/ Livermore tutorials

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

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

[openmp-refcard] OpenMP in Fortran Reference card

[openmp-RR] Chapter 5 of *[Rauber-Ruenger]*

[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