ams209_fall_2016 Documentation

Release 1.1

Dongwook Lee

September 23, 2016

CONTENTS

1 Announcements on Office Hours **3**

2 Contents: **5**

2.1 Disclaimers 5
2.2 Course Mission 5
2.3 About the Website 5
2.4 License 5
2.5 Syllabus 6
2.6 Preliminaries 6
2.7 Lecture Notes 6

3 Webpages of class members **29**

Welcome to AMS 209 Foundations of Scientific Computing at AMS UCSC!

This is a class webpage where you can find the classnotes for the course. The contents herein are going to be continuously updated throughout the quarter.

CHAPTER

ONE

ANNOUNCEMENTS ON OFFICE HOURS

1. With Dongwook (dlee79 _at_ ucsc _dot_ edu)
    - TBA

CHAPTER

TWO

CONTENTS:

2.1 Disclaimers

These lecture notes on Foundations of Scientific Computing are going to be progressively under development for the course AMS 209 at the Department of Applied Mathematics of the University of California Santa Cruz, Fall Quarter, 2016.

The most parts of the contents of the materials are in progress and they are continuosly getting updated and modified throughout the entire quarter.

The materials are intended to serve as a good intelletual guidance for the course, and it is never meant to be neither perfect nor best.

Please keep in mind that the topics that are covered in this class are very widely spreadout in many different areas. Therefore the course materials provided here do not intend to be most updated and accurate. Rather, the purpose of this classnote is to provide some levels of broader aspects in those various topics that play fundamental roles in scientific computing.

2.2 Course Mission

This course is not a CS/CE course where you would study various programming languages, as well as software engineerings, and hardware architectures, etc. in depth from theoretical aspects. If you're interested in such topics you're in the wrong place.

Please note that the primary goal of the course is to focus on *how* to use scientific tools to successfully conduct your researches in modern sciences from practical perspectives.

2.3 About the Website

These online class materials are writen in Sphinx.

2.4 License

2.5 Syllabus

Please note that these schedules are tentative and they may be modified if needed.

Week 1: Introduction to Unix/Linux basics including basic tools for programming – editors, compilers, libraries, Makefiles, config files, ssh/scp/sftp, version control, code publication, etc.

Week 2-3: Introduction to basic algorithm development and program structures (e.g., data types, data structures, IF, DO, and WHILE constructs, functions, subroutines, arrays, modules, etc.) common to many languages but using Fortran (90 and above) as the primary example language.

Week 4: More advanced programming in Fortran (Fortran 90 or above), e.g. modular programming, dynamic array allocation, user typed structures, I/O, debugging, etc.

Week 5: Introduction to object-oriented programming concepts through compare-and-contrast of C/C++ and Fortran.

Week 6-7: Introduction to flexible interpreted languages for interfaces using Python programming as the example.

Week 8: Basics of computer architecture (chip architecture, cache, network infrastructure, file systems, etc) with a view to understanding code optimization, bottlenecks, debugging, etc.

Week 9: Data analysis and visualization: Introduction to basic analysis and visualization tools; good practices for running codes in production mode.

Week 10: Introduction to good software engineering practices; code validation and verification.

2.6 Preliminaries

Q1. What is your name and what is your major?

Q2. Please tell me about your research interests.

Q3. List any courses you took in the past which required for you to use computers, including word processings.

Q4. What do you want to get the most out of it from this class? What is your motivation in taking this class?

Q5. What do you think scientific computing is about? Why do you think people study that?

Q6. What kinds of computing resources and softwares have you used?

Q7. What kind of computer(s) do you own and what do you do with them?

Q8. Do you have any computing exprience using Unix/Linux, or Mac OS X?

Q9. Are you familiar with LaTex?

Q10. Have you heard about version control, for instance, svn or git?

Q11. Have you been involved in any group project in the past?

Q12. How passionate are you in scientific computing? Please let me know if you have any comments and thoughts.

2.7 Lecture Notes

2.7.1 Chapter 1. Operation systems, Version controls, Remote access

Motivations and Needs for Scientific Computing

Let's begin our first class with a couple of interesting scenarios.

Scenario 1

Consider you're a chief scientist in a big aerospace research lab. See Figure 2.1. You're given a mission to develop a new aerospace plane that can reach at hypersonic speed (> Mach 5) within minutes after taking off. Its powerful supersonic combustion ramjets continue to propel the aircraft even faster to reach to a velocity near 26,000 ft/s (or 7.92 km/s, or Mach 25.4 in air at high altitudes, or a speed of NY to LA in 10 min), which is simply a low Earth orbital speed. This is the concept of transatmospheric vehicle the subject of study in several countries during the 1980s and 1990s. When designing such extreme hypersonic vehicles, it is very important to understand full three-dimensional flow filed over the vehicle with great accuracy and reliability. Unfortunately, ground test facilities – wind tunnels – do not exist in all the flight regimes around such hypersonic flight. We neither have no wind tunnels that can simultaneously simulate the higher Mach numbers and high flow field temperatures to be encountered by transatmospheric vehicles.

Fig. 2.1: Figure 1. DARPA's Falcon HTV-2 unmanned aircraft can max out at a speed of about 16,700 miles per hour – Mach 22, NY to LA in 12.

Scenario 2

Consider you're a theoretical astrophysicist who tries to understand core collapse supernova explosions. See Figure 2.2. The theory tells us that very massive starts can undergo core collapse when the core fail to sustain against its own gravity due to unstable behavior of nuclear fusion. We simply cannot find any ground facilities that allow us to conduct any laboratory experiments in such highly extreme energetic astrophysical conditions. It is also true that in many astrophysical circumstances, both temporal and spatial scales are too huge to be operated in laboratory environments.

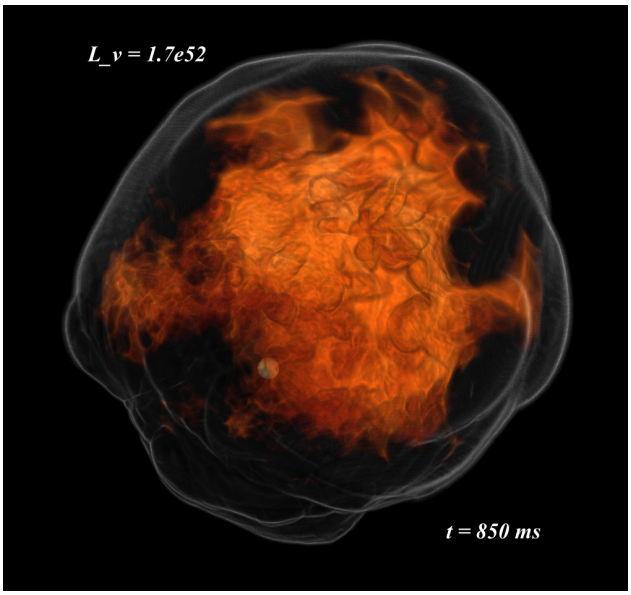

Fig. 2.2: Figure 2. FLASH simulations of neutrino-driven core-collapse supernova explosions. Sean Couch (2013), ApJ, 775, 35.

Scenario 3

Consider you a golf ball manufacturer. See Figure 2.3. Your goal is to understand flow behaviors over a flying golf ball in order to make a better golf ball design (and become a millionaire!) Although you've already collected a wide range of the laboratory experimental data on a set of golf ball shapes (i.e., surface dimple design), you realize that it is very hard to analyze the data and understand them because the data are all nonlinearly coupled and can't be isolated easily. To keep your study in a better organized way, you wish to perform a set of parameter studies by controlling flow properties one by one so that you can also make reliable flow prediction for a new golf ball design.

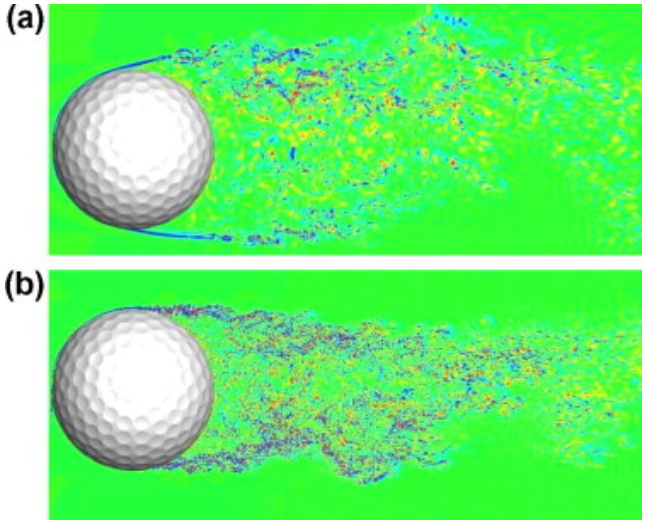

Fig. 2.3: Figure 3. Contours of azimuthal velocity over a golf ball: (a) Re = 2.5 x 10^4; (b) Re = 1.1 x 10^5. C. E. Smith et al. (2010), Int. J. Heat and Fluid Flow, 31, 262-273.

As briefly hinted above, in practice there are various levels of difficulties encountered in real experimental setups.

When performing the above mentioned research work, computational fluid dynamics (CFD) therefore can be the major player that leads you to success because you obtain mathematical controls in numerical simulations. Let us take an example how numerical experiment via CFD can elucidate physical aspects of a real flow field. Consider the subsonic compressible flow over an airfoil. We are interested in answering the differences between laminar and turbulent flow over the airfoil for Re = 10^5. For the computer program (assuming the computer algorithm is already well established, validated and verified!), this is a straightforward matter – it is just a problem of making one run with the turbulence model switched off (for the laminar setup), another run with the turbulence model switched on (for the turbulent flow), followed by a comparison study of the two simulation results. In this way one can mimic Mother Nature with simple knobs in the computer program – something you cannot achieve quite readily (if at all) in the wind tunnel. Without doubt, however, in order to achieve such success using CFD, you'd better to know what you do exactly when it comes to numerical modeling.

CFD as a Scientific Tool

We are now ready to define what CFD is. CFD is a scientific tool, similar to experimental tools, used to gain greater physical insights into problems of interest. It is a study of the numerical solving of PDEs on a discretized system that, given the available computer resources, best approximates the real geometry and fluid flow phenomena of interests. CFD constitutes a new "third approach" in studying and developing the whole discipline of fluid dynamics. A brief history on fluid dynamics says that the foundations for *experimental* fluid dynamics began in 17th century in England and France. In the 18th and 19th centuries in Europe, there was the gradual development of *theoretical* fluid dynamics. These two branches – experiment and theory – of fluid dynamics have been the mainstreams throughout most of the twentieth century. However, with the advent of the high speed computer with the development of solid numerical studies, solving physical models using computer simulations has revolutionized the way we study and practice fluid dynamics today – the approach of CFD. As sketched in Fig. 4, CFD plays a truly important role in modern physics as an equal partner with theory and experiment, in that it helps bringing deeper physical insights in theory as well as help better desiging experimental setups. See Figure 2.4.

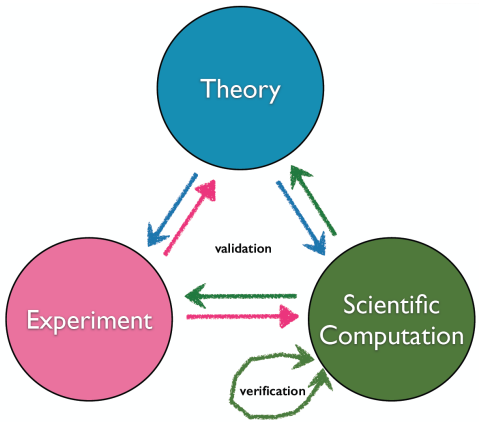

Fig. 2.4: Figure 4. Three healthy cyclic relationship in fluid dynamics.

The real-world applications of CFD are to those problems that do *not* have known analytical solutions; rather, CFD is a scientific vehicle for solving flow problems that cannot be solved in any other way. In this reason – the fact that we use CFD to tackle to solve those *unknown* systems – we are strongly encouraged to learn thorough aspects in *all* three essential areas of study: (i) numerical theories, (ii) fluid dynamics, and (iii) computer programing skills.

Items for the Class

Hardware (hardly free)

- A computer: laptop or desktop. No tablet PCs. Better if you already have a linux-based one or Mac. For a Windows-based PC, you need to install some packages to use a linux-based operating system.

Softwares (mostly free)

Essential software (required, free):

- Linux/Unix or Mac OSX with Xcode (no Windows).
- Compilers for Fortran 90, C/C++ (e.g., gfortran, gcc)
- Debuggers (gdb , lldb, etc.)
- Python and its scientific libraries (e.g., Anaconda, NumPy and SciPy, Matplotlib, etc.)
- Version control system, e.g., Git or SVN
- Text editors (GNU emacs, aquamacs, vi, vim, etc.)
- Sphinx
- Latex packages (texshop, LaTeXiT)

Extra software (optional, free or commercial):

- Matlab or GNU Octave
- software package manager (Yum, Homebrew, MacPorts, etc.)
- GNU plot, IDL
- symbolic math using sage
- even more for high performance computing on Mac

Others (absolutely free)

- Most importantly, your passion and energy to learn new things

Programming Languages, Platforms, Operating Systems

Scientific languages

One of the crucial components of the entire course work is to write computer programming codes for homework sets, programming assignments, and a final coding project.

A choice for a reference language is going to be Fortran 90 in this course. Fortran 90 (or higher) has been one of the most widely used programming languages in high performance computing communities for over half a centry. It is easy, compatible, and has been chosen for various benchmark tests that run on large scale computing architectures, or supercomputers.

The main goal in this course is to learn practical experiences in different programming languages as well, such as C and python, and learn how to use them for of scientific computing.

You should be able to submit your course assignments by successfully implementing required numerical algorithms in those languages.

Computing platforms

In order to conduct scientific programming it is required that you have an access to a Linux/Unix computing platform, where you can excersise such series of programming studies.

Basically, there are several options to bring a Linux/Unix computing system to your daily scientific adventures. Please see also *Installing Linux on Your PC*.

- if your machine is either a Linux or Mac machine, use your own machine to run your code locally.
- if your machine is a Windows PC, you can remotely access to a Linux computer using an X-forwarding terminal such as PuTTy over the network. Putty is one of the SSH clients on Windows allowing you to work on a remote Linux computers.
- if you prefer to run programming locally rather remotely (e.g., limited internet access at your place), you can either install Cygwin which brings you functionality similar to a Linux distribution on Windows. Note that Cygwin is *not* a way to run native Linux apps on Windows.
- if you rather wish to have a native Linux setup on your Windows PC, you can run a pure linux environment using a free virtualization software called virtualbox which is quite excellent. This also allows file sharing between your host operating system (e.g., Windows) and the virtual operating system (e.g., Linux).

Remark You can also learn useful tips not only from google searches but also from youtube these days. So please use those visual resources as well as readable resources.

Remark And, don't forget one thing. If you need help, please don't be shy, hesitating to ask around good people. And I am one of them, hopefully.

Computing Resources in AMS Even though you might have your own computing resources over the course (your own Linux or Mac), we choose the default computing platform to be the Linux Grape cluster from the AMS department. You can see the description of Grape.

If you are not an SOE student, please come to see the instructor to get an account on Grape. Basic Fortran and C/C++ compilers (e.g., GNU gfortran), some other necessary libraries, and softwares (gnuplot, matlab, idl, etc) are available on Grape. If you find any specific software needs to be installed, please let the instructor know your request.

Since it uses a Linux operating system (Grape runs on a Linux operating system called *Rocks*), and it is a remote cluster, in order to remotely access to Grape, you will need to

- learn how to use Unix/Linux command lines (or simply commands) (see *Basic Unix/Linux Commands*)
- make sure you have remote access to it and to its text editors (emacs, xemacs, vi, vim, etc.).

As mentioned, this should be trivial if you are using either a Linux or Mac machines. If you are a Windows PC user, you can install PuTTy (enough for accessing Grape remotely) or Cygwin (allowing remote access capability as well as Linux functionality).

If you haven't had any chance to work on Linux/Unix type operating systems, please make sure you first familiarize yourself with basic Linux/Unix commands (This is very crucial. See *Basic Unix/Linux Commands*). It is your prime responsibility to learn about working on Linux/Unix environment as quickly as possible – you should trust that you can do it!

The instructor is happy to help you, however, is bound to be limited to provide you with detailed technical supports at all levels.

Remark If you prefer to use your own laptop/desktop to program, it is your responsibility to install Fortran and all the libraries we will be using. You should be able to find a free Fortran 90 compiler (e.g., GNU gfortran) for most

platforms. The most basic one will be sufficient for this class, but please be aware that the quality of a compiler has a lot to do with the speed of execution of the program.

Remark In order to obtain an account on the Grape Linux cluster machine in AMS, please go visit the link and request an account for youself for the course:

- <https://accounts.soe.ucsc.edu/accounts/register>

You need to specify me as your sponsoring faculty member.

Scientific Computings on Linux/Unix, Mac OS X

Most projects of scientific computing heavily rely on developers' interactive programming and software handlings across various components of softwares. Engineers, scientists and researchers in scientific computing are often assumed to carry out programming practices using different combinations of command lines on a Unix-based operating system (OS).

Shell Most common examples of such OS are Unix, Linux and Mac OS X, where you perform computing jobs by typing in your command lines on a terminal, called a shell.

The shell provides users with a user interface by which users can access the computer's operating system and/or interact with target software packages in order to carry out computational jobs.

If your experience with computers has only been in using Windows PC, with which you interact by clicking some icon apps to open and conduct jobs, this will be an absolutely new opportunity to use computers in a very different way.

Although a Windows system has its kernel OS, called DOS, where users can possibly run command-line jobs, we will *not* use DOS (like every wise soul wouldn't) for scientific computing.

Installing Linux on Your PC There are a couple of different ways to run Unix/Linux-system on your Windows PC. Here are my recommentations (you will need to do one of them).

If you are considering an updating your laptop anyway, please do:

- get a Mac, and download and install Xcode on it. This can be easily done in a terminal window (go to Applications –> Untilities –> Terminal to open a terminal) by typing in (also see how-to-1):

```
$ xcode-select --install
```

- get a PC, erase the Windows OS, and install a Linux OS (e.g., Ubuntu) as a sole operating system.

Otherwise, if you want to keep your old Windows PC but want to do some cool scientific computings, please do:

- install virtualbox (easy and best),
- install Cygwin (altenatively good and efficient),
- configure dual-boot to run both Windows and Linux,
- access an external Linux machine over the network using PuTTy (least recommended)
- More to read here

If you are one of the computer grus, you might as well come up with a crazy combination, such as (I saw few people who did this):

- get a Mac, erase Mac OS X, and install Linux on it (you wonder why you would want to do this?).

Remark Further details on the Linux installing instructions on PCs can be easily found online, e.g., how-to-2. You will also find online tutorials on how-to instructions on Youtube as well. Please don't forget to use these great resources, and of course, you're always welcome to ask me or TA.

Basic Unix/Linux Commands In this section we give a quick overview on some of the basic Linux commands. There are few rules in using command lines in Linux. Several important rules are

- Commands are case-sensitive.
- Make sure you always logout yourself by typing `exit` when you're done.
- The Linux command lines enables you to create complex functions by combining built in command lines together. This capability gives you countless ways to make your commands work in various different ways.

Here you're introduced to learn very basic Linux command lines. For more comprehensive studies on various options associated with Linux commands, you can use to display a manual page using the `man` command followed by your search commands. For instance if you want to learn more about `cp` command

```
$ man cp
```

1. Here are some basic commands for managing files:
   - list directory contents:

```
$ ls
```

   - `ls` in long format:

```
$ ls -l
```

   - `ls` all entries of contents:

```
$ ls -a
```

   - rename (or move) `filename1` to `filemane2`:

```
$ mv filename1 filename2
```

   - copy `filename1` to `filename2`:

```
$ cp filename1 filename2
```

   - remove a file named `filename`:

```
$ rm filename
```

   - display the contents of a file named `filename` as much as will fit on your screen:

```
$ more filename
```

   - similar to `more` with the extended navigation capability allowing both forward and backward navigations:

```
$ less filename
```

   - print the entire `filename` rather than a page at a time:

```
$ cat filename
```

   - view multiple files' contents and direct the output to save into a new file `output` using standard output > (`STDOUT`):

```
$ cat filename1 filename2 > output
```

   - tells you number of lines, words and characters in `filename`:

```
$ wc filename
```

- creating an empty file named `filename` (multiple filenames after `touch` command will create multiple empty files):

```
$ touch filename
```

- find a file named `filename` in the current directory `.`:

```
$ find . -name filename
```

- find a file named `filename` in the entire file system `/`:

```
$ find / -name filename
```

- find a file named `filename` only under your personal directory `~/`:

```
$ find ~/ -name filename
```

- find a search keyword `ams209` among all files with extension `.tex`:

```
$ grep "ams209" *.tex
```

2. Here are some basic commands for managing directories:

- create a new directory called `dirname`:

```
$ mkdir dirname
```

- change directory, meaning you go to a directory called `dirname`:

```
$ cd dirname
```

- tells you where you currently are in the directory tree:

```
$ pwd
```

- remove empty directory: it will fail if the directory is not empty:

```
$ rmdir dirname
```

- remove non-empty directory with all the contents therein:

```
$ rmdir -f dirname
```

- remove non-empty directory as well as all of the sub contents therein:

```
$ rmdir -rf dirname
```

3. Here are some commands for 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.

   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:

```
$ ps
18917 ttys000    0:00.19 -bash
21181 ttys000    0:00.00 /bin/bash /Users/dongwook/anaconda/bin/python.app /Users/dongwook/anacc
21182 ttys000    0:00.19 /Users/dongwook/anaconda/python.app/Contents/MacOS/python /Users/dongwc
```

```
18921 ttys001    0:00.01 -bash
18925 ttys002    0:00.02 -bash
20647 ttys003    0:00.01 -bash
20656 ttys004    0:00.01 -bash
21171 ttys005    0:00.01 -bash

$ kill 21181
```

Hit return again you will see:

```
$
In [1]: Terminated: 15
$
```

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

```
$ kill -9 21181
```

This almost always kills a process. Be careful what you kill. Also try to see more options in using `kill` command by typing `man kill`.

4. Setting up environment varialbes (case sensitive!):

   - Environment variables: There are a couple of choices for Unix/Linux shells. One can categorize them into two, *Bourn shell* and *C shell*, where in each category there are a number of variant shells. In this class we use bash shell as our default choice.

     Under any circumstances where your default shell might not the `bash` shell, you can initiate `bash` by typing:

     ```
     $ bash
     ```

     in a terminal. This will start the bash prompt.

     In Unix/Linux there are variables called *environment variables* which define various properties that are important in the system. They include various variables, paths and shortcuts which can be set by the system, users including you, or by the shells, or even by some of the programs that are installed or used interactively with some other programs.

     The following list includes several important environment variables that users often encounter (note that they are all capitalized):

Variables	Description
DISPLAY	Contains the identifier for the display that X11 programs should
HOME	Indicates the home directory of the current user. the default arg built-in command, that is to say, typing 'cd' will jump to HOME f
LD_LIBRARY_PATH	On many Unix systems with a dynamic linker, contains a colon-sepa directories that the dynamic linker should search for shared obje building a process image after exec, before searching in any othe
PATH	Indicates search path for commands. It is a colon-separated list directories in which the shell looks for commands.
PWD	Indicates the current working directory as set by the cd command.
USER	Current user name(s)

Please see more in article-bash.

- Customizing enviromental variables in `.bashrc` or `.bash_profile`:

  Such variables are exported to the system everytime you start a new bash shell, e.g., opening a new terminal (in case bash is your default shell), or logining in to the system, or typing in `bash`. In these cases, a file named `.bashrc` under your home directory is automatically executed. This means that you can always customize your own enviromental variables settings by modifying `.bashrc` file. What you can do by modifying the file includes:

  - any custom excution of program on startup
  - exporting environmental variables
  - setting paths
  - defining aliaes
  - customizing your prompt, etc.

  Here is an example of `.bashrc`:

```
#-----------------------------------
# Print current user names
#-----------------------------------
u="$USER"
echo "user name $u"

#-----------------------------------
# Export some paths
#-----------------------------------
export ams209svn="$HOME/Repos/ucsc/soe/teaching/2015-2016/Fall/AMS209/lectureNote"
export ams209git="$HOME/Repos/ucsc/soe/teaching/2015-2016/Fall/AMS209/ams209_git/lecture_not
export SVN_EDITOR="emacs -nw"
export IDL_DIR="/usr/local/itt/idl"
export PAPER_DIR="/Users/dongwook/Repos/ucsc/mongchi/DOCS/ucsc/research/papers"
export PATH="/usr/local/Cellar/colordiff/1.0.13/bin:/usr/local/bin:$PATH"
export PATH="/usr/local/Cellar/valgrind/3.8.1/bin:$PATH"

.
.
.
.
.
.

#-----------------------------------
# Aliases
#-----------------------------------
# Bash
export LSCOLORS=gxBxhxDxfxhxhxhxhxcxcx # dark background
alias lls='ls -laghFG'
alias clean='rm *~'

# Commom Mac programs
alias reload='source ~/.bash_profile'
alias sublime='/Applications/Sublime\ Text.app/Contents/SharedSupport/bin/subl'
alias text='open -a TextEdit'
alias pre='open -a Preview'
alias grepp='grep -in'
alias sshy='ssh -Y'
```

There is another way to achieve the same using a different file called `.bash_profile`. You can put all of the above in `.bash_profile` instead of `.bashrc`, and customize your settings as you wish. One may ask then what is the difference between the two files. The answer for the Unix/Linux system is that `.bashrc` is executed for interactive non-login shells (e.g., opening a new terminal), whereas `.bash_profile` is executed for login shells.

In the above `.bashrc` example, you as a system admin are going to see a list of the current user names logged in to the machine you just logged in, everytime when you open a new shell terminal. Usually you want this information only once when you login to the machine and to keep prompting this information on every new terminal would be unnecessary. To avoid this you can instead add such monitoring/diagnostic tools in `.bash_profile` which will only be executed when logins.

This difference doesn't exist in Mac OS X as an exception though and `.bash_profile` is invoked for each new terminal window instead of `.bashrc` on Mac OS X operating systems.

In general, you don't want to maintain the two separate files differently for login and non-login shells, especially, you want to set `PATH` properly in both shells. A good way of consolidating the two files into one can be done by sourcing `.bashrc` from `.bash_profile`. A good example of `.bash_profile` then begins with:

```
#-----------------------------------
# Source global definitions (if any)
#-----------------------------------

if [ -f ~/.bashrc ]; then
   source ~/.bashrc
fi

#-----------------------------------
# Print current user names
#-----------------------------------
u="$USER"
echo "user name $u"
```

With this you can put all the paths, custom aliases and common settings only in `.bashrc`.

For more information on the bash shells and enviroment variables please read the two articles and a youtube tutorial:

- article: bash 1
- article: bash 2
- youtube: variables

- The `which` command is useful for finding out the full path to the code that is actually being executed when you type a command. For example, if you have successfully installed *gfortran* and *python* on your computer, you should be able to see:

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

$ which python
/Users/dongwook/anaconda/bin/python

$ 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.

Version Control System – Managing Your Projects

Note This part of the lecture note has been partially extracted and modified from Prof. Randy LeVeque's class website on HPC.

In this class we will use `git` for

- homework submission,
- code project submission,
- final coding project submission,
- electronic file transfers needed for the course work between you and the instructor.

See the below 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.

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` 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

- 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. 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.

Two Types of Version Control Systems: SVN vs. Git

Client-server systems (e.g., CVS, SVN) 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.

Please see articles on comparison between svn and git:

- article 1

- article 2

both of which give brief overviews on two different client-server systems.

Distributed systems (e.g., Git) 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 the next section.

Remark Please also go watch the following Youtube video tutorials on git:

- video 1
- video 2
- video 3

Git for the Class using Bitbucket

Instructions for cloning the class git repository **Note** This part of the lecture note has been extracted from Prof. Randy LeVeque's class website on Git and has been modified sligtly.

All of the materials for this class, including homework assignments, sample programs, and lecture note (html and pdf) are controled in a Git repository hosted at Bitbucket, located at ams 209 git.

In addition to viewing the class materials and associated files via the link above, you can also view changesets, issues, and update histories, etc. as well. To obtain a copy of the class git repo, simply create one directory where you want your copy to reside, say, `ams209` in your home directory, move to the directory, and then `clone` the repository as follows:

```
$ mkdir ams209
$ cd ams209
$ git clone https://bitbucket.org/dongwook159/ams209-fall-2015.git ./
```

If you fail to clone the repo with the following message:

```
$ fatal: Authentication failed
```

then this means that you haven't been invited to join as a team member to have an access to the course repo. In this case, please send me your email (preferably your ucsc email, rather than your personal email) so that I can send you out an invitation. You are going to use the same email when you create your own Bitbucket account for your own later (see *Creating your own Bitbucket repository*).

There is no (white) space in the above git command line. At this point, it is assumed you have git installed on your OS. Otherwise, go visit download:git. The clone statement will download the entire contents of the class repository as a new subdirectory called `ams209`.

Keep your cloned git repo updated/synced with the course repo The files in the class repository remotely hosted in the Bitbucket website will continuously get changed and updated as the quarter progresses with new notes, sample programs, and homework sets, etc. In order to bring these changes over to your cloned copy, all you need to do is

```
$ cd ams209
$ git fetch origin
$ git merge origin/master
```

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
```

because `origin` and `master` are the defaults.

Creating your own Bitbucket repository In addition to using the class repository, you 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, 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.

Remark 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.)

Remark Please take a look at an article comparing Bitbucket and GitHub

Getting used to your own local git repo 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 *ams209-fall-2015*, owned by user *dongwook159*. To avoid confusion, you should probably not name your repository *ams209-fall-2015*.

You should stick to lower case letters and numbers in your repository name, e.g. *ams209-ucsc* or *ams209-scicomp* 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. The directory you are now in will appear empty if you simply do:

```
$ ls
```

But it will look slightly different if you try:

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

the *-a* option causes *ls* to list files starting with a dot, which are normally suppressed. See *Basic Unix/Linux Commands* 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*.

11. 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 *Items for the Class*).

12. To see *status* of your folder, 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 31cb6ed] 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 31cb6ed38310eed36f47d3d3aed769e03da540c9
Author: dongwook159 <dlee79@ucsc.edu>
Date:   Fri Sep 25 00:04:14 2015 -0700

My first commit of a test file.
```

The number 31cb6ed38310eed36f47d3d3aed769e03da540c9 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.

Modifying a file

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 271bd14e5b8d68840e7e6481ad7e99e5708e50e7
Author: dongwook159 <dlee79@ucsc.edu>
Date:   Fri Sep 25 00:02:34 2015 -0700

       added a third line to the test file

       commit 0c20925f98b5d76d0b973d25fdc78fd43941792e
       Author: dongwook159 <dlee79@ucsc.edu>
       Date:   Fri Sep 25 00:01:25 2015 -0700

       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  31cb6ed383
Note: checking out '31cb6ed383'.

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 31cb6ed383... 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.

1. 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.)

2. 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.

Rolling back to a previous state Let's take a look at the case where you do not like your last change you made to your repo, and you want to revert your repo status back to a previous state, say,

- `commit 1b82c21688befa80560807247594d73768d64f0a` (the current unsatisfied revision) –> `commit c27d1bdf0098efe59aa25f809a719ce4fa910fef` (the previous revision you wish to roll back to)

In this case, there are two ways to roll back your repo to the previous state.

Firstly, if you do:

```
$ git reset --hard c27d1bdf0098
```

it will revert **both** *the local code* and *the local history* back to the previous state. This might look ok but it would fail if you wished to push your reverted repo to the remote public repo especially when there is someone else in your team who already has the new history from the state `commit 1b82c21688befa80560807247594d73768d64f0a`.

Instead, if you do:

```
$ git reset --soft c27d1bdf0098
```

it will only revert your local files back to the previous state, leaving your history unchanged. In this case, you can successfully push your changes to the public repo without causing any conflicts in histories among your project team members.

In case you want to recover files that are deleted locally, you can do:

```
$ git ls-files -d | xagrs git checkout --
```

Similarly, to recover modified files back to the previous states:

```
$ git ls-files -m | xagrs git checkout --
```

See more examples at https://git-scm.com/docs/git-ls-files. In some cases, you may wish to forget about all your local changes and want git to overwrite the entire local files. In general, if you have some changes in your local files that git sees as potential conflicts, *git pull* will not allow you to bring in the most recent updates committed to the git by others. Git will give you errors such as:

```
$ error: Your local changes to the following files would be
overwritten by merge:
```

or:

```
$ error: The following untracked working tree files would be overwritten by merge:
```

In this case if you don't mind overwritting your local changes with whatever available in the git, you can do the following:

```
$ git fetch --all
$ git reset --hard origin/master
```

or you can combine the two in a single line command using *&&*:

```
$ git fetch --all && git reset --hard origin/master
```

Again, with this command, all of your local changes will be lost with or without *–hard* option, and therefore any local commits that haven't been pushed will be lost. So, you do this if you know what you're doing and trust the recent updates by pulling from the git repo.

Summary The commands we discussed so far will give you a good start with git. As you're getting used to use git you will learn that only a handful git commands are needed in many cases. This is in particular true unlesss you work on the project with many other project members over the network. In our class it will primarily be yourself only who will keep checking in and out changes to and from your central repo hosted in Bitbucket. Another frequent usage will be to sync your local repo with the course repo on a regular basis.

In this simple project enviroment, you will most likely need to use the following commands:

```
$ git status
$ git add
$ git commit
$ git push
$ git pull
```

Accessing the Network Resources

Login via `ssh`

Basic syntax SSH, or *secure shell* is one of the most common way to access remote Unix/Linux servers over the network by allowing users to logon to the servers with a secure protocol.

In this section we take a look at how to logon to the Grape AMS Lunux server remotely.

- The first step is to open a terminal.
- In the terminal, type in:

```
$ ssh yourID@grape.soe.ucsc.edu
```

where `yourID` is your SOE login ID. You are going to be asked to type in your SOE password for a successful login.

- You can also include `-X` or `-Y` after `ssh` in order to allow `X11 Forwarding` to view a remote system's graphical user interface (GUI) getting forwarded on to your local system:

```
$ ssh -Y yourID@grape.soe.ucsc.edu
```

- If your login is successful you should see something like the following on your terminal:

```
Last login: Mon Apr 27 17:52:34 2015 from mongchi.soe.ucsc.edu

Computer technical support requests can be submitted via the web at
     https://itrequest.ucsc.edu

or by e-mailing
     help@ucsc.edu

*********************************************************

Online documentation for the grape cluster can be found at
     http://grape.soe.ucsc.edu

Grape cluster is using the infiniband fabric.

*********************************************************

Some Torque commands

qsub        --> Submits a job (create a shell script, then run qsub shellscript)
                use the -q option to specify which queue to use
qdel        --> Delete a job
qstat       --> see the status of jobs in the queue
qstat -Q    --> List of usable queues
pbsnodes    --> List status of all compute nodes


*********************************************************

There are currently 4 queues on Grape

orig   -  compute-0-0-compute-0-4  PowerEdge 1950
          Intel(R) Xeon(R) CPU 2.33GHz 15G MEM

new    -  compute-0-5-compute-11  Dell PowerEdge R610
          Intel(R) Xeon(R) CPU 2.40GHz 15G MEM

newest  - compute-0-12-compute-0-19  PowerEdge R420
          Intel(R) Xeon(R) CPU 2.30GHz 32G MEM

default - includes all of the nodes.  This is the
          default queue.

*********************************************************
```

Login into SSH with Keys One can set up *key-based authentification* once which can be very useful to log in to a remote machine *without typing in* password in every login. To see more details, please read this article:ssh-key, and this youtube:ssh-key.

File transfer via `scp`

There are many cases where you want to transfer files from host machine A to host machine B. `scp` is a command for *secured copy* that allows you data transfer and provide the same authentication and same level of security as `ssh`.

- In the following examples we assume I am transfering `fileA` that resides in the directory `/Users/dongwook/Documents/` in your local machine, to my `HOME` directory `/soe/dongwook/ams209/` on Grape:

```
$ cd /Users/dongwook/Documents/
$ scp fileA dongwook@grape.soe.ucsc.edu:~/ams209/
```

  The last line can be also replaced with the command with a full explicit path:

```
$ scp fileA dongwook@grape.soe.ucsc.edu:/soe/dongwook/ams209/
```

- In case if I just want to transfer `fileA` to my `HOME` directory (which is `/soe/dongwook` instead of `/soe/dongwook/ams209/`), it can simply be:

```
$ scp fileA dongwook@grape.soe.ucsc.edu:
```

- If I would want to transfer multiple files, `fileA`, `fileB`, etc., I just list all of them after `scp`:

```
$ scp fileA file B dongwook@grape.soe.ucsc.edu:
```

- If I want to transfer `fileA` to Grape in a different name, `fileC` in `HOME` directory:

```
$ scp fileA file B dongwook@grape.soe.ucsc.edu:~/fileC
```

File transfer via `sftp`

`sftp` command is another similar protocol as `scp` for file transfer, but also can be used to allow more interactive commands such as generating new directories, deleting and moving files as well. For more details, please see article-sftp.

Syncing files via `rsync`

`rsync` is not a secured file transfers (or syncs) between the two remote computers. However, it provides a consistent way to maintain files in a two different locations. Please see more on wiki-rsync.

CHAPTER

THREE

WEBPAGES OF CLASS MEMBERS

- Prof. Dongwook Lee, Instructor, AMS
- Arevalo, Daniel, AMS
- Bardales, Alexander, EE
- Biswas, Sudipto, SciCAM
- Ding, Zhehao, CMPE
- Dixit, Akhil, CMPS
- Ganjallzadeh, Vahid, EE
- Gonzalez Torres, Bernardo, CMPS
- Gulla, Roy, SciCAM
- Hancock, Alexander, CMPS
- Hou, Jiaql, EE
- Lam, Tuwin, EE
- Li, Yanzhong, CMPE
- Liu, Suhan, SciCAM
- Malik, Osman, CMPE
- Mazhari, Arash, CMPE
- Milenska, Lillya, SciCAM
- Nasab, Sara, AMS
- Pansodtee, Pattawong, CMPE
- Richardson, Jennie, BME
- Vargas, Brian, SciCAM