ASTR 575

Me

Fall 2015

Contents

`something` that looks red?
`something else` that’s also red.

1 Course Overview

Uquity of computing in field, and also as skill for other fields.

Philosophy: need to have understanding and flexibility, computing skills are lifelong learning skills. Note that the way that scientists have tended to program may be rather different from how computer programmers in other sectors work!

Operational details: course material (including language discussion), course resources, homework, notes and note-taking

575 and 535

git branches, how to fork and send pull requests. Learn more about git command in general.

2 Introduction to computing hardware and NMSU/Astronomy computing

2.1 Hardware

Computer components and functions:

- CPU(s): characterized by the rate at which they can do operations (*flops* is a standard, to be distinguished from *clock speed* in Hz/MHz/GHz).
- memory: volatile storage (but relatively fast), typically Gbytes per machine.
- disk(s)

  internal permament storage, typically TBy per disk (example: Serial AT Attachment (SATA)).
  external USB, eSATA, Firewire
- network: communication between machines
- monitor/keyboard
- power
- bus

Understand and be able to articulate the different parts of computer hardware and their basic functions.

2.2 Bits, bytes, and whatever

- Bit: single binary unit of information (0/1).
- Byte: 8-bits, usually the smallest unit used to represent something. How many different quantities can a byte represent?

Data types/sizes:

- character (ASCII 1 byte, or more for other character encodings)
- short integer (2 bytes)
- integer (4 bytes)
- long integer (4/8 bytes)
- float (4/8 bytes)
- double (8/16 bytes)

How large a range of data values in a given data type?

Know what bits and bytes are. Know the basic variable types and be able to give their numerical ranges.

File/data sizes:

Question: APOGEE image cubes are 2048x2048x47 reads: how much memory?

Question: MULITDARK simulations have 2048x2048x2048 particles with full phase space information: how much memory?

Physical memory and swap space: code with memory in mind

Be able to calculate how much memory will be used for a specified amount of data.

Integer representation: unsigned and signed integers (e.g., two's complement)

Floating point representation: IEEE, note precision issues

binary vs character representation

architectures and byte-swapping (big-endian vs little-endian)

2.3 Computing software

2.3.1 Operating systems

Operating system: controls basic interface between user and hardware

- Windows
- Unix (various "flavors", originally BSD vs AT&T)
- Mac OSX (flavor of Unix)
- Linux (flavor of Unix)
- Others: VMS, ChromeOS, Android, ISO

OS must evolve as new hardware is developed, leading to versions, with implications for stability/support/security. The main core of the operating system is called the **kernel**, which provides a programming interface for programs.

Linux implementations (distros): include basic Linux kernel plus add-on packages, usually with a **package manager**, e.g. yum, apt-get.

RedHat / Fedora / CentOS, Debian / Ubuntu, SUSE, Gentoo, others

versions accommodate both OS modifications and package development

2.3.2 Operating system interfaces

Operating system generally provides kernel and a core set of commands. On top of this, there may be command interpreters and/or graphical interfaces to commands.

Command-line interfaces:

UNIX uses "shells" that, in addition to kernel command interface, allow scripting operations, variables, input/output redirection, etc. Common shells: csh/tcsh, bash, ksh, each allow for different "style"

Graphical interfaces: "window" system:

Operating system provides hardware interface; most Unix machines use protocol called X11, Mac OS X uses quartz (but X11 is available) On top of this, distributions provide window manager / desktop enviroment, which is a graphical use interface (GUI) to the operating system Common desktop enviroments: KDE, GNOME

Understand and be able to articulate the functions/differences of operating systems, shells, and windowing systems.

2.3.3 Applications

A large amount of software has been developed to work under a given operating system, and these are available as packages: examples are editors, compilers, etc. (do a `rpm -q -a` on one of the Linux machine to see what is installed). In fact, the shells are installable applications, so the distinction between core software and applications is not totally clear.

2.4 Network communication

Various software allows for various actions across the network:

- ssh: allows for secure login (encrypted communication)
- NFS (network file system): allows network disk access (at network speeds)
- NIS (network information system): allows for sharing of common information (e.g., login information, automount services, mail aliases, etc.)
- CUPS (common Unix printing system) allows for sharing of printers on the network
- any software can communicate over the network using the general Unix concept of sockets, which allow for intermachine communication

Remember, network communication is slower than disk communication, and affects other who are using the network.

2.5 NMSU/Astronomy (academic)

Cluster of computers running Linux (CentOS) on Astronomy building subnet

- Some individual user machines, some department machines
- CentOS adopted because of security/stability/support issues
- Some faculty/students have Apple machines (Mac, Macbook) which run MacOSX, another Unix-like variety
- Some faculty/students have laptops with some flavor of Linux installed (usually Ubuntu?)

Servers (most located in room 116A, apart from backup devices in computer center)

- astronomy: NIS master, web server, email server, printer server
- astrodisk: disk server (/astro: users, httpd, local, ftp, also catalogs, aips, redhat)
- astrobackup, astrobackup2: disk backup machines

Compute nodes

- public machines: hyades, praesepe, virgo
- "private" machines: seismo, solarstorm, milkyway
- typically have 16-64 processors, additional memory

Desktop machines

- (usually) single processors, 4-8 Gby memory
- Student machines: 1 TB disk, split into two user partitions: /{machine} and /{machine}-data, with backup implication : files on /{machine} are backed up (daily for small files, with some history retained, weekly for all files, but with no history retained), files on /{machine}-data are not backed up!

Disk sharing is accomplished through NFS. The NIS master has a table of all mounted partitions, and makes them accessible via /home/partitionname. However, the machine serving the partition needs to give permission. By default, server partitions are shared to all machines, and main partition on client machines are shared to the servers, but client partitions are not shared with other clients except by request. Local partitions are also available via the NFS interface, with no penalty, so it is a good idea to use the /home/partitionname for generic disk access.

Note that the default home directories for students (/home/users/name) is located on the astronomy server. As a result, if you work in or under your home directory when logged into your machine, you are using the network whenever you read/write from disk. It is generally better to use the local disk if you are doing any significant I/O (note distinction between /partitionname/ and /partitionname-data/)

Server shares software that is not installable as packages, so using such software invokes network traffic. Since the server disk is generally checked whenever you type a command, you may see slow response at your computer if the network is being excessively taxed.

Understand enough about the NMSU Astronomy setup to recognize what pices of hardware (CPU, disk, memory, network) you are using depending on what machine you are logged into, and where you are reading/writing from/to on disk.

2.6 Linux resource usage

Determining CPU, disk, and memory capability: /proc/cpuinfo, /proc/meminfo, df, various grapical interface tools (Mac OS X: system_profiler)

(note man command to get information on any command!)

machine types, etc.: uname -a

Determining CPU, disk, and memory usage: top, ps, w, du

3 Working in a Unix environment

See tabular summary at unix.html

3.1 Linux help

- http://freeengineer.org/learnUNIXin10minutes.html
- http://www.tutorialspoint.com/unix/unix-useful-commands.htm

3.2 Basics: directories

File systems: disks (physical units) and partitions (logical units).

df Display free disk space
rm -r Recursively remove directories
rm -i "Are you sure?" Might alias `rm` to this.
cd - Go back to directory you were in before the previous `cd` command.

Separating file name from directory name: basename and dirname.

```
/dev/sda3
/dev/sda5
/acrux - root partition
```

3.3 Basics: files

3.3.1 Displaying files

`ls`

-a Include directory names that begin with '.'
-l Permissions, # hardlinks, owner, group, size (bytes), date, filename
-F Display '/' after directory names
-r Reverse order
-t Sort by modification time
-R Recursive `ls` (all sub-directories)

3.3.2 Creating files

touch
echo output redirection
mv
cp

3.3.3 permissions

chmod
chown
umask

Unix permissions cover three types of users:

- user, or owner (u)
- group (g)
- other (o)

Each of these can have read, write, execute permission, expressed in three bits, in order rwx. You can change permissions using characters, e.g.:

chmod o +w Give `write` permissions to `other`
chmod g -x Remove `execute` permissions from `group`

or via a permissions mask.

```
755 (rwxr-xr-x)
644 (rw-r-r-)
000 no permission for anyone
101 group and other can execute and read (1) but not write (0)
755 - 111, 101, 101  permission for user to do everything, group and
                     other can't write.
> umask 022 Can put this in .cshrc, second '2' - 010...?
            All permission to owner, everything except write for group
            and other...
```

If a file is an executable (program), you can run it simply by typing its name; either need absolute path, or just filename if file is in the current PATH environment (see below).

3.3.4 advanced file types

Links come in two types:

- **symbolic (soft):** A symbolic link points to another file, and can work across file systems: the link is distinct from the destination file, and if the destination file is removed the link will be broken.
- **hard:** A hard link works only within a file system, and provides an independent link to the same file: if either source or destination is deleted, the file still remains with the linked name.

¿ ln -s {source} {destination}	Create symbolic link
¿ ln {source} {destination}	Create hard link

Pipes allow for interprocess communication through a pseudo-file. They are created using: `mkfifo`

inside a single machine; program reads from file on disk; make interface to program on webpage?; submit write input to file.; Inside program: open file to read from: pipe temporary .pipefile; program and website can send information to each other.;

3.3.5 file types

- executable
- directories
- regular files
- links

standard file extensions and file naming conventions, e.g. https://www.openoffice.org/dev_docs/source/file_extensions.html; https://kb.iu.edu/d/affo

Directory names: `my\ file`... Don't use spaces! "Royal pain in the butt!"

stdin, stdout and stderr: input and output redirection (note shell dependence)

piping commands: you can direct the output of one command into the input of another using the — character, e.g. ls — more pattern matching: wildcards (globbing):

***** All strings
? Single character
[**abc**] Matches any one, not necessarily in that order
regular expressions ?

Re-naming in bulk! Append `.old` to extension, or change the extension:

```
foreach x (*.txt)
    mv $x $x.old
    mv $x $x:r.dat
```

Is this a script? Or typed at command line?

3.3.6 locating files

`find` "descends the directory tree" beginning at each pathname and locates files that meet the specified conditions. Searches all subdirectories by default, but not upward from where you are unless you specify an absolute pathname. Note that pathname must be specified, even for current directory.

Basic syntax: `cl> find` *`pathname conditions`*

-name *pattern*	Find files whose names match *pattern*
-iname *pattern*	Ignore case
-type d	directory
-type l	symbolic link
-type f	plain file
-mtime +n—-n—n	Find files that were last modified more than n, less than n, or exactly n days ago.
-exec [command] {}	Run the command on the files found. When the command runs, the argument {} substitues the current file. Follow the entire sequence with an escaped semicolon. In some shells, the braces may need to be escaped as well.
-size n[c]	File containing n blocks, or if c is specified, n characters (c is optional, brackets not included in actual command).
-user *user*	Files belonging to user (name or ID)
¿ find . -maxdepth *n*	Specifiy how many directories deep you want to search. $n = 1$ is current directory only (`find` is recursive by default).

Examples:

¿ find . -name a.txt	Match `a.txt` exactly, not simply contain it.
¿ find . -name *.tex	
¿ find . -type f -maxdepth 1 -exec mv dir1	Move all files to `dir1`.

Note that on linux, need to escape special characters, e.g. `\*.tex`.

`locate` uses pre-built system database of files. Search database(s) of filenames and print matches. Matches include all files that contain pattern unless pattern includes metacharacters, in which case locate requires an exact match. *, ?, [, and] are treated specially; / and . are not. Searches are conducted against a database of system contents that is updated periodically. To update the database, use the `updatedb` command.

Basic syntax: `cl> locate [options]` *`pattern`*

-d *path*, –database=*path*	Searches databases in *`path`*, which must be a colon-separated list.
-h, –help	Print a help message and then exit.
–version	Print version information and then exit…apparently you still need to specify a pattern to get the version information…?

3.3.7 Display file contents

cat
more
less
od Bytes level
head -n file n = number of lines
tail -3 file Last 3 lines
head -15 — tail -1 Last line of first 15 lines

You can use " to put output of a command into another command. Must be a typo here.

Be comfortable…

3.4 Unix useful file manipulation tools

Unlike the previous commands (`find`, etc.), the following commands involve the file *contents*, not the files themselves.

3.4.1 `grep`

Search for specified pattern within files and print the results (the contents of each line) to standard output.

Syntax: `> grep [options] pattern [files]`

Options:

-i Case insensitive
-v Search for all lines that do *not* have specified pattern.
-r Search directories recursively (current directory only by default)
-c Number of occurrances
-n Print line number in front
-l List only the names of the files with matches, not every single line. Scanning per file stops on the first match.

Examples:

ls -l — grep -v .fits Everything *except* fits.
¿ grep *string* *.txt Look for *string* in all .txt files.
grep -c /bin/tcsh /etc/passwd List the number of users who use `tcsh`
grep -w not haiku.txt Print every line that contains 'not' to screen.
grep -l '^#include' /usr/include/* list header files that have at least one #include directive.
¿ history — grep head List all commands that used head

3.4.2 `sed`

Command line editor, line-by-line processing using regular expressions.

sed '/{pattern}/d' {filename} remove lines that match specified pattern
sed 's/{old}/{new}/' {filename} search and replace

¿ sed '/holtz/d' filename Delete the pattern 'holtz'
¿ sed 's/hotz/holtz/g' filename Correct misspelled words

3.4.3 `tr`

line-by-line processing with simple replacement. Mostly useful to translating special characters, e.g., Windows files ending with ¡CR¿ ¡LF¿ to Unix files.

3.4.4 `sort`

Sort lines of text files. Numerical and then alphabetical by default (i.e. no options specified), using the characters in the first column.

Syntax: `> sort [options] {file}`

-key=column e.g. `sort -key=2 {filename}` will sort alphabetically by column 2 (or `> sort -k2`?).

-n numerical sort (i.e. use whole numbers, not just the number in the *nth* column).
¿ sort . . . — tail -1 Find out what the biggest/smallest number is.

3.4.5 paste/join

Pasting files together, merge n files based on line-matching.

3.4.6 awk

Provides a command-line programming interface, convenient for simple operations (but can handle more complex ones). Pattern matching: searches line by line for those that match the specified pattern, then performs the associated actions.

Syntax: `> awk ‘condition {action}’ filename`

No pattern = does action on every line
No action = prints to screen every line that satisfies the pattern.

```
cl> awk ‘/search pattern 1 / {actions}
         /search pattern 2 / {actions}’ file
         [| output for another command]
```

```
cl> awk ‘{print NR, $0}’
cl> ls -l | awk ‘{print NR, $2}’
cl> awk ‘{print > ‘‘line- ’’ NR%2}’ filename
cl> awk ‘NF != 15 {print NR}’ filename
```

where `NR` = Record Number (line number), `NF` = Number of Field in current record (line). Can think of NF as the column number? `$NF` = last field, and `$#` = column number (`$0` = all columns). By default, the columns are separated by a white space. For the line `12 me 849`, ‘12’ is in column 1, ‘me’ is in column 2, and ‘849’ is in column 3.

```
cl> awk ‘$2 > 99’ filename
cl> awk ‘$2 < 10 {print $4, $5, $6}’ filename
```

Arithmetic based on columns:

```
cl> awk ‘{print $2, $3, $2*$3}’ filename
cl> awk ‘$1 > x {print $2, $3, 5*$4, 6*$3}’ filename
cl> awk ‘NR%2 == 0’ filename
    0
    1
    0
    1
cl> awk ‘NR%2 != 0’ filename
    1
    0
    1
    0
```

Last two examples have a condition, but no action, so the output is printed to screen. The condition is that each line whose value of `NR%2` either is (or isn't) equal to 0 is printed. So every other line, starting with the second in example 1 and the first in example 2. Original file:

```
        0
        1
        1
        0
        0
        1
        1
        0
        0
```

I think.

Use awk on variables:

```
    cl> echo $var | awk '{printf("set %s = %s.new\n)",$1,$1}'
```

Put awk commands in a separate file and use this to execute! Syntax:

`cl> awk [options] {script} [var=value] {files}`

```
    cl> awk -f awkfile {filename}
    cl> awk -t cmd.awk {filename}
```

Example of `awk` programming:

```
    BEGIN{
       TOT3 = 0
      }
       TOT3 = TOT3 + $3
      {
      }
    END{
       PRINT TOT3
     }
```

http://www.thegeekstuff.com/2010/01/awk-introduction-tutorial-7-awk-print-examples/

3.4.7 `diff`

Show differences between files. Syntax

`diff [options] [directory options] file1 file2`

-wb Ignore whitespace (linux)
-B Ignore whitespace (mac)

3.4.8 `wc`

Counts lines, words, characters.

wc -l ¡filename¿ prints the line count (note that if the last line does not have `\n`, it will not be counted).
wc -c ¡filename¿ prints the byte count
wc -m ¡filename¿ prints the character count
wc -L ¡filename¿ prints the length of longest line (GNU extension)
wc -w ¡filename¿ prints the word count

Be familiar with Linux commands: wc, grep, sort, diff, and awk. Be able to use awk to (at least) do simple arithmetic manipulation of files.

3.5 Archiving and compression tools

3.5.1 tar

Bundles multiple files into a single archive

tar -cvf {archivename} {file}—{directory} Put files into archive.
tar -xvf {archivename} Extract
gzip compresses files losslessly, creates .gz files
compress another compression, creates .Z files
xzip yet another compression, creates .xz files
fpack tool for compressing FITS files

3.6 Unix system information, resources and usage

```
Determining CPU, disk, and memory capability: /proc/cpuinfo,
/proc/meminfo, df, various grapical interface tools (Mac OS X:
system_profiler)
```

(note man command to get information on any command!)

machine types, etc.: uname -a

Determining CPU, disk, and memory usage: top, ps, w, du

w, whoami

hostname

Understand. . .

3.7 Unix job control

ps: show current processes

top: show processes sorted by resource usage, updates in real-time

kill: sends signals to processes multiple signals available

CTRL-Z: sends a stop signal to running process

CTRL-

: sends a kill signal to running process foreground/background: using signals (note signal trapping in code)

cron jobs: allow for job to run on a regular schedule, through use of a crontab (crontab -e to edit).

Understand...

3.8 Intermachine communication

ssh: used for secure remote login, but also can be used to execute commands on remote host (e.g., ssh hyades w)

scp: copy files over ssh connected

sftp: open a session with a remote host to enable transfer of one or multiple files/directories

ftp: older, less secure, method for file transfer, but still often used for anonymous ftp, where external users can access a restricted area to grab files, or even to transfer in, if that is enabled. On our cluster, we run an FTP server on astronomy.nmsu.edu: the reserved area is under /home/ftp/pub for outgoing file, /home/ftp/incoming for incoming files.

rsync: used to ”sync” files/directories, i.e., transfer only files that differ between systems. Can run locally or between machines using ssh protocol (-e ssh)

Globus?

ssh-keys: provides an alternative to password authentication. Instead of sending a password to a remote machine, a key pair is generated, with a public and private key. The public key is initially transferred to the desired remote server; on subseqeuent connection requests, the server returns the public key to get a match with private key, and if so, the connection is established. Usually, the matching of the keys requires a passphrase as well. One advantage is better security. Another is convenience, as it is possible to register your private key with an ”ssh-agent” so that the passphrase is entered only once for a session, allowing remote ssh logins for the rest of the session without needing a password. Commands:

```
ssh-keygen
ssh-agent
ssh-add.

Keys are created in /.ssh/; public keys are appended to /.ssh/authorized_keys
```

Understand how...

3.9 Unix environment

Unix allows for ”environment” variables that are visible to all shells (as opposed to shell variables that are local to a given shell). These are often used for general configuration, and are very useful in the context of software setup and package/data management. This is especially true when you may be setting up an environment on multiple machines, where root directory names differ.

setenv, printenv: commands to set and show environment variables in csh/tcsh.

In bash, set environment variables using: export var=value

some common/useful environment variables:

```
EDITOR/VISUAL
DISPLAY
SSH_ASKPASS
SHELL
PATH
```

Understand. . .

3.10 Editors

An editor is a basic tool used for nearly all computing tasks, so it is worthwhile to have a strong command of the editor that you choose to use.

Given today's networked environment, it is very likely that you will need at some point to edit files on remote machines. Working with editors that open graphical windows can become a significant challenge when working over the network, and for this reason, I would strongly discourage them.

The historical editor associated with Unix is vi/vim, and this is still in widespread use. Another extremely widespread editor is emacs, which can be used within a terminal window using the -nw option. Another terminal based window that is sometimes installed is nano.

With whatever editor you use, you should be able to

efficiently, move the cursor within files. Goto specific line numbers Search for text strings Search for text string and replace with alternate string delete line or blocks of lines with single command be able to efficiently move blocks of text around, e.g. copy and paste

Be extrememly comfortable. . .

3.11 Unix shells and shell scripting

startup: .cshrc, .bashrc, with useful customization commands:

alias definitions environment varibles set path for searching for executables using PATH environment variable (printenv PATH to see what path is) which command: find full directory path of a given command (or find out whether the command is in the path)

history command: lists past N commands

command completion (TAB-completion) and recall (!N executes command N; !x executes last command that begins with x.

When to use shell scripts? Often most convenient for operations have to do with files and simple file modification. Not usually a good choice for numerical work!

running command files: source vs starting new shell (#!), file permissions (a file must have executable permission for it to be identified in the path as a command).

scripting:

variables: reference using $varnamecommandlinearguments$:0 is first word (the command), $1thefirstargument, etc.;$* refers to all arguments (after command), which is very useful, e.g., in conjunction with for loops (below) to loop over all command line arguments conditionals:

```
== (equal)
!= (not equal)
```

```
&& (and)
|| (or)
```

looping

see http://astronomy.nmsu.edu/holtz/a575/unix.html#shell for syntax of shell commands in bash and tcsh.

see PDF presentation from Utah CHPC on shell scripts

csh:

```
set
if / then / else
often used with file inquiry conditionals (e.g. -e tests for
existence)
foreach /end
while / end
```

3.11.1 Example

```
bsyn &
set bsynjob = $!
set tmax = 300
set runtime = ‘ps -q $bsynjob -o cputime | tail -1 | awk -F: ’{print ($1*3600)+($2*60)+$3}’‘
while ( $runtime < $tmax )
  sleep 2
  set runtime = ‘ps -q $bsynjob -o cputime | tail -1 | awk -F:
  ’{print ($1*3600)+($2*60)+$3}’‘
  if ( ‘ps -p $bsynjob -o comm=‘ == "" ) then
    echo process done, exiting!
    exit
  endif
end
echo expired, killing job
kill $bsynjob
```

Understand. . .

3.12 Remote & virtual desktops

e.g., VNC: vncserver and vncviewer.

4 Presentation/communication

4.1 Text processing: LaTeX

Basic LaTeX:

`\documentclass{class}` article, letter, book, others (see AASTeX below, which provides aastex and emulateapj)

Preamble:

- package setup: `\usepackage{package}` graphicx, amsmath, hyperref, color
- command aliases, e.g., `\newcommand{\mh}[0]{[M/H]}`

math mode: reference `$` and `$$`, superscript, subscript, greek letters, math symbols, etc. (can be used in iPython notebooks)

various environments: itemize, enumerate, figure, table, tabular, center, equation, e.g.: Figures with `\usepackage{graphicx}`, `/includegraphics`

```
\begin{figure}
    \includegraphics[width=0.5 \textwidth]{cfig.pdf}
    \caption{caption here}
    \label{fig:cfig} (see below)
    \end{figure}
```

Refer to Figure [*] in the text to automatically get the correct number!

```
\begin{table}
    \begin{tabular}{llcr}
        obj1 & data1 & text1 & number1 \\
        obj2 & data2 & text2 & number2 \\
    \end{tabular}
    \label{tab:data}
\end{table}
```

Refer to Table [*] in the text to automatically get the correct number.

AASTeX: documentclass for astronomical publications: aastex, emulateapj, deluxetable environment for tables (including multi-page).

labels: to allow for arbitrary moving/addition/deletion of figures, tables, sections, etc., do not build in numbering into the text. Instead use the `\label{labelname}` command to label each entity, and automatically reference the correct numerical identification using `\ref{labelname}`. Note that using these will require two passes through latex, to register the labels correctly. Note that, in the figure environment, the label command must come after the caption command.

References: use bibitems. Create the reference once, give it an identifier, and reference it in the text by the identifier using `\citep, \citet, \citealt` (the natbib package provides an extension beyond the standard LaTeX commands). You can create the bibitem manually (`\bibitem{identifier} reference`), or, better yet, use BiBTeX reference information (available from ADS) and automatically create the bibitems.

- manual bibitem:

  ```
  \begin{thebibliography}
      \bibitem
      ...
  \end{thebibliography}
  ```

- bibtex: create `reffile.bib` file with bibtex entries, use `\bibliography{reffile}` in latex file, then

  ```
  latex file
  ```

```
        bibtex file
        latex file
        latex file
```

to put it all together. See discussion on ADS; for astronomical journal reference style, you may want to use the apj bibliography style, using the apj.bst file.

Refer to an article inline (with year in parenthesis) using `\citet{id}`. Refer to an article where author and year are both in parenthees using `\citep{id}`.

`\bibliography{ref}` at end of document.

"compiling" LateX / pdflatex. Can use a makefile to simplify. Using bibtex, the sequence is:

1. prepare a `doc.tex` file with the document and a `ref.bib` file with a bunch of BibTeX entries (e.g., from ADS)
2. from command line, run

```
        pdflatex doc
        bibtex doc
        pdflatex doc
        pdflatex doc
```

latex editors: kile. On line resources for sharing/editing/viewing LaTeX: sharelatex, authorea, overleaf, etc.

Spell checking: aspell, hunspell. ALWAYS SPELL CHECK YOUR DOCUMENTS!

Be able to easily create a complete LaTeX file that includes sectioning, tables, figures, and a bibliogrphy, using labels for cross referencing figures, tables, and sections, and the family of cite commands for references. Understand how to turn the LaTeX file into a PDF file, including the sequence needed to include references from a bibtex file, and section cross-referencing.

4.1.1 Introduction to makefiles

Introduction to makefiles (for example, see the GNU documentations or someone else's explanations).

Standard rules. Standard targets: install, objs, etc.

4.2 HTML

http://www.w3schools.com/tags/
http://www.w3schools.com/css/
http://www.ironspider.ca/index.htm
http://www.advancedhtml.co.uk

Structure of an HTML document:

`<command></command> structure, <HTML><BODY [bgcolor=]>,` Basic HTML text processing: Like LaTeX, text will appear continuously in a browser regardless of how it is entered in the HTML source file, unless there are command directives. The width of the text will adapt to the width of the browser. Various text directives:

```
    <Hn> Headings in different font sizes (1 is largest) </Hn>
    <P> Force a line break with vertical paragraph space
    <BR> Force a line break without extra vertical space
    &nbsp; Force small horizonal space
    <UL>/<OL> Unordered list / ordered (numbered) list
        (end with </UL> or </OL>)
            <LI>  individual list element
```

Font commands

```
    <I> italic </I>
    <B> bold </B>
    <FONT [color=]> general font command
```

HTML links:

```
    <A HREF=address> active text </A> : sets up a link
    <A NAME=label> : sets a lable in a HTML file, can be directly skipped
        to using address\#label format
```

HTML tables:

```
    <TABLE [BORDER=]>: start a table
    <TR> : start a new row
    <TD [COLSPAN=]> : start a new column
    </TABLE> : end table
```

HTML figures:

```
    <IMG SRC={file} [WIDTH=] [HEIGHT=]>
```

Force size using `WIDTH=` and/or `HEIGHT=`, usually only one of these, as this will automatically preserve aspect ratio. If at all possible, use file format that most browsers know how to display inline: GIF, JPG, PNG

comments in HTML files: `<!-- comment -->`

All of these can be combined, e.g., a table of figure thumbnails with links to full size figures:

```
<TABLE BORDER=2>
<TR> <TD> <A HREF=fig11.jpg> <IMG SRC=fig11.jpg WIDTH=200> </A>
<TR> <TD> <A HREF=fig12.jpg> <IMG SRC=fig12.jpg WIDTH=200> </A>
<TR> <TD> <A HREF=fig21.jpg> <IMG SRC=fig21.jpg WIDTH=200> </A>
<TR> <TD> <A HREF=fig22.jpg> <IMG SRC=fig22.jpg WIDTH=200> </A>
</TABLE>
```

Frames, CSS, and more.

Understand how to write a basic HTML page, including lists, figures, tables, and hyperlinks.

A web server is a network program that runs on a server machine, and interprets/delivers web pages as requested. Standard web requests are http://{machinename/address}/{file—directory}. One the web server, the {file—directory} is interpreted as a relative path to some document root directory. If the requested address is a directory, the web server will look for a file named index.html in that directory and, if it exists, will use the contents of that file; if it doesn't exist, it will show the directory listing of the directory.

Web server at NMSU uses /home/httpd/html as the document root, and all users have a directory /home/httpd/html/{username}, so web requests to http://astronomy.nmsu.edu/{username} will look for files in this directory. Note that this directory is located on the disk server, so it is not a great location to put large files; if you want to provide access to large files, or even whole directories, located on a disk on your machine, consider the use of symbolic links.

4.3 Collaboration

email/listservers : value as email archive/ risk of junk mail. Be aware of reply-all. Listservers as archives. Listserver options: digest delivery.

File size and format (e.g., for plots): deliver files so as to make it as easy as possible for audience to read/view/digest them!

web pages: very useful as hierarchical source of information, with figures

wiki : centralized set of user-editable web pages, with many other features.

iPython notebooks.

5 Programming

5.1 Overarching concepts

- Computing/software is lifelong learning
- do as little as possible "interactively" so it is repeatable.
- less is more (to a point); don't repeat work, make the tool better and reusable.
- all software should be written as if someone else will use it: document and share.

5.2 Philosophy

From Quora, response to a question about value of formal training in programming vs. picking it up on your own:

Upvoted by Alagunarayanan Narayanan, Software Engineer "I was a self-taught programmer for about 4 years, before taking a few courses as extras in my EE degree. I am now pursuing a MSc in CS. So my answer is mostly observations on how I do things differently now compared to how I did them before.

Architecting projects is a big thing. Given a list of things the software is required to do, how would you lay out your program so that it

1. Does what it's supposed to do
2. Is maintainable
3. Is easily understandable for other people (and yourself in a few months)
4. Is easily extensible if the requirements change (as they always do) Uses the most suitable design patterns to make the code intuitive

Usually self-taught programmers do 1., but don't pay nearly as much attention, or only pay lip service to the other points. I thought I knew about all those other points, but as I found out later, I really had no idea.

Another big difference is how much trial and error. I did a lot more trial and error back in the days, because I didn't have good understanding about how "the whole stack" works. I used a lot of things without bothering to truly understand how they work under the hood.

Nowadays I almost don't do any trial and error at all. I think through everything, and most things I actually start coding actually work on the first try (not counting typos, etc). I know exactly how memory management works all the way from kernel to malloc. I know how schedulers work (having written a few for a course), and I know what the synchronization mechanisms are, the pros and cons of each, the usual patterns in which they are used, as well as how they are actually implemented on the instruction level. As a self-taught programmer, I knew how to use mutexes and that's about it. As it turned out, I actually (poorly) reinvented a few of the other standard mechanisms.

Self-taught programmers are also usually not used to reading a lot of code written by other people, which is a very important skill when working in teams.

Knowledge of the existing algorithms is another one.

When a non-trivial problem is encountered, a bad programmer dives head first into coding a solution. A better programmer looks for solutions, and tries them out. A good programmer looks for solutions, analyzes them for time and space complexity as well as other constraints, and implements the most likely one.

Most self-taught programmers start coding too early.

And like you said, things like AI and ML. Most self-taught programmers never learn those things, because usually they only learn things they need, and if you don't know AI and ML, they won't seem like possible solutions to your problems, and you'll never think about learning them. It's a chicken and eggs problem. Self-taught programmers often don't know what they don't know. One nice thing about doing a degree is that it almost forcefully introduces you to everything, so by the time you are done, at least you know what you don't know."

5.3 Languages

Compiled vs non-compiled. Distinction perhaps less clear now than it was in the past. But key point is that, for certain applications, you may want to consider execution time.

Open-source vs proprietary

Languages: commonly used in astronomy: Python, C, C++, Fortran, IDL, MATLAB. Many other languages exist: C++, C#, Java, Javscript, etc. etc.

Fortran: historical language of choice of scientific computing in mid/late 20th century. Many codes developed that are still in existence/development (e.g., Anatoly's N-body and Hydro codes, synthetic spectra generation MOOG and TURBOSPEC, stellar evolution MESA, Chris' codes, others...). In addition, many routines coded for fast operation available, e.g. LAPACK. Various versions of fortran: F66, F77, F90/F95

C: foundational language for computer science. Some astronomical routines: SAO WCSTOOLS for astronomical coordinate system routines, HSTPHOT/DOLPHOT for HST photometry, SLALIB for more astronomical coordinate routines, others....

Python: major push in current astronomical software development. astropy

IDL: historical very close connection to astronomy, with much development at LASP and Goddard Space Flight Center in the public domain. Marketed through RSI, then ITT, now Exelis, as a licensed software product. Originally written in C? Major astronomical software infrastructure exists in IDL, e.g. for planetary and solar astrophysics (perhaps because of NASA / GSFC). Astronomy users library. Major component of SDSS software, although modern evolution away from it.

5.3.1 Getting started

see programming reference table.

5.3.2 Getting started with Fortran

Basic program structure:

1. program statement
2. variable declarations
3. program statements
4. end

Basic language:

- f77 starts in column 7, f95 can start in column 1
- Comments: '!' (f95) 'C' in column 1 (f77)
- line continuation: '&' (f95 and f77)

Compiling and running programs: compiling, linking, makefile, BINDIR, path

5.3.3 Getting started with C

Basic program structure:

1. #include files
2. int main(){
3. variable declarations
4. program statements
5. }

Comments: '//' or embed between '/*' and '*/'.

Compiling and running programs: compiling, linking, makefile, BINDIR, path

5.3.4 Introduction to makefiles

https://www.gnu.org/software/make/manual/html_node/index.html#Top, http://www.rsmas.miami.edu/personal/miskandarani/Courses/MSC321/make.pdf

Explicit commands, variable filenames, rules.

Defining rules, e.g. Latex into PDF, etc Standard rules.

Standard targets: install, objs, etc.

5.3.5 Getting started with Python

Using command interpreter:

- python
- ipython features

Using programs: running interactively, running as a script, running as an executable, using %run with ipython

Environment variables:

executables searched for in PATH
imports searched for in PYTHONPATH, as well as system-installed libraries.

To edit PYTHONPATH:

```
export PYTHONPATH=${PYTHONPATH}:/path/to/directory/
```

```
>>> import sys
>>> sys.path.append(‘‘path/to/Modules’’)
>>> print sys.path
```

In order to have Python see the modules inside each subdirectory, add a blank file called `__init__.py` to each subdirectory (with two underscores on each side).

To automatically import the usual, edit the file:

```
cl> vi ~/.ipython/profile_default/startup/00startup.ipy
  import numpy as np
  import matplotlib.pyplot as plt
  ... etc.
```

This will automatically be executed when `ipython` is started. The numbers in front of the file name specify the order of preference in which they are executed, e.g.

```
00first.ipy
50middle.ipy
99last.ipy
```

Comments: hash (#).

Python useful references: Python tutorial, python4astronomers

5.3.6 Getting started with IDL

Using command interpreter.

Using programs: run a script using `IDL> @scriptname`, run a program using `IDL> .run programname` or `cl> idl -e`

IDL_PATH

Comments: semicolon (;)

5.3.7 Simple makefile

Simple makefile (note that indents must be TABs):

```
hello_f:
    f95 -c hello_f.f95
    f95 -o hello_f hello_o

hello_c:
    cc  -c hello_c.c
    cc  -o hello_c hello_c.o

run:
    hello_f
    hello_c
    hello.py
    idl -e ‘‘.run hello.py’’
```

Understand how to write basic programs in Fortran, C, Python, and IDL. KNow how to write and use a basic Makefile for compiling and linking Fortran and C programs.

5.4 Basic Programming

See programming reference table for syntactical details in each language.

5.4.1 Program construction

basic program structure, comments and good commenting practice, line continuation

5.4.2 Variables

variables and memory, declaration and initialization. Basic variable types: integer, float, string. Determining variable type in Python (type) and IDL (size,/type). Type conversion arrays and array operation: order of elements in multidimensional array (column-major vs row-major). Array as a continuous stretch in memory. pointers: addresses vs values dynamic memory allocation: allocate and malloc multitype collections: structures (derived data types). Arrays of structures. Python lists, tuples, dictionaries, and structured arrays

Understand the basic variable types and difference between languages that require variables to be declared and those in which the type is determined dynamically

.

5.4.3 Operators

Mathematical, string, bitmasks and logical operators, vector operators

Understand basic operators and how to use them. Understand bitmasks and logical operators. Understand how vector operators can be used in some languages.

5.4.4 Control statements

Conditional: if/then/else, control loops: for and while.

Understand and be very familiar with how to use control statements in a programming language (Python recommended), and how to at least recognize and understand the funtionality of such statements in other languages.

5.4.5 Input and output (I/O)

* I/O : formatted output. * I/O : simple input. Reading unknown length files in Fortran and C * I/O : higher level routines: Python astropy.io.ascii, numpy.loadtxt (but note this only reads a single data type into an array, rather than into a structured array, as astropy.io.ascii does). IDL read_ascii * Binary data: note issues of N-dimensional array unwrapping, byte order Fortran: OPEN(lun,file,FORM='binary'), READ/WRITE without format statements C: fopen(), fread(), fwrite() Python: open files in binary mode (rb, wb), struct.pack and struct.unpack to convert to binary data IDL: READ_BINARY * FITS files image data: headers and data. N-dimensional images. Standard header cards. WCS. FITS tables Extensions/HDUs Routines: Fortran/C cfitsio, Python astropy.io, IDL Users library FITS routines, e.g., mrdfits/mwrfits * note Python astropy.table unified I/O for multiple file types, including ASCII and FITS! * Databases as alternative to data files

Python4astronomers primer on reading and writing files

Understand and be very familiar with how to read and write files in a programming language (python recommended) and how to at least recognize and understand the functionality of such functions in other languages.

5.4.6 Program organization: subroutines and functions

Utility of functions: improve readability of code, minimize/eliminate code repetition: more compact code and easier to change, use same functions in multiple programs

Using libraries in Fortran, C, Python, IDL:

In many cases, you may have functions that you wish to use with multiple main programs. In this case, rather than including the same code in multiple program files, you want to keep the code in separate "utility" files, where these files contain only functions/subroutines, but not main programs.

Python: modules. from and import statements. Location of files: system installation and PYTHONPATH environment variable.

```
import name
import name.subdir
import name.subdir as subdir
from name import subdir
from name import *
```

Suggestion: collect all of your Python routines under a python directory and include this in your PYTHONPATH, so you can find all of your Python routines in one place (no need for "I know I did this somewhere, but can't find where. . . ")

Examples of some libraries:

- Python: standard libraries, including os (operating system routines), add-on libraries, e.g., astropy, numpy scipy (wide range of scientific/numerical techniques), and many others.

Understand . . .

Understand . . .

5.4.7 Objects and object-oriented programming

traditional programs tend to think of variables and functions that act on variables. Object oriented program tends to think of objects that can have associated attributes, but also actions that may depend on the attributes.

Compare dictionary (or structured array) with a class Objects: attributes and methods. Python: everything is an object! E.g., strings, files as objects. Inspecting attributes and methods in Python using iPython: object.¡tab¿, object.function?. Regular Python: dir() function. Defining objects with CLASS, e.g.,

```
h0 = 72.

class gal :
    def __init__(self) :
        self.name = 'test'
        self.ra = 120.
        self.dec = 40.
        self.vrad = 3000.

    def dist(self):
        return self.vrad / h0
```

Note that object inheritance is possible, example: isochrone as a collection of stars

Understand . . .

5.5 Error handling and code testing

error handling: good code will trap errors and report the source rather than rely on the error message when the code crashes. Note Python try/except statements.

testing code: find a case where you know the solution and make sure your code gets the right answer! Think of challenging cases that you can test, or at least test for behavior in the expected direction.

5.6 Debugging code

Python: pdb

5.7 Coding practices

style, e.g. PEP8 guidelines, e.g., white space, indentation rules, ... Make your program readable. Consider white space judiciously, recognizing tradeoff between separating program blocks and increasing length of code. Modular code is generally easier to read and digest. comments: every program must be commmented! Include an overview comment at the top and clarification comments throughout the code as needed. Don't need to comment what is apparent from reading the code if the code is straightforward. Separate comment statements are usually preferable to "end-of-line" comments. documentation and documentation tools: Sphinx, doxygen (examples) Avoid hard-wired directories: consider use of environment variables to specify root directories Avoid hard-wired constants, file lengths version control and package management: tagging in version control software; modules: handles environment variable setup, dependencies, etc. Example. . .

Understand . . .

5.8 More advanced coding

Command line arguments Signal trapping sockets Parallel programming cross-language programming

5.9 Case study/review

Consider isochrone reading code examples (isochrones.py) Example of astropy FITS I/O.

6 Plotting

Interactive plotting vs. subroutine plotting.

TOPCAT

Python matplotlib and IDL

6.1 Basic Plots

Line graphs: colors, line styles, limits, labels, multiple data sets on a single plot

Point graphs: colors, point types, limits, labels

Hardcopy

6.1.1 Python

See matplotlib usage Parts of a plot.

tutorial

plot commands

State machine interface:

Line plots:

```
import matplotlib.pyplot as plt #(already done with ipython --matplotlib)
plt.plot(y)
plt.plot(x,y)
plt.plot(x,y,color='red'|'green'|'blue'|'cyan'|'magenta'|'yellow'|'black'|....)
plt.plot(x,y,linestyle='-'|'--'|'-.'|':',linewidth=1)
plt.plot(x,y,drawstyle='steps-mid')  # "histogram" style connect
plt.plot?  # for more options
```

Plot points:

```
plt.plot(x,y,marker='o'|'.'|...)
plt.plot(x,y,'r-'|'go'|'b.'|....)
```

For all sorts of point/marker types, see markers.

Error bars:

```
plt.errorbar(x,y,xerr=xerr,yerr=yerr)
plt.errorbar?  # for more options
```

where xerr and yerr can be single values to be used for all points, or arrays of values for separate error bar lengths for each point.

Limits and labels:

```
plt.xlim([xmin,xmax])
plt.ylim([ymin,ymax])
plt.xlabel(label)
plt.ylabel(label)
plt.text(label[0],label[1],label[2])
```

Hardcopy (note that if you want to make hardcopy plots without having a plotting window opened, don't use ipython, just use python.):

```
plt.savefig(file)
```

Alternatively, there is a pyplot object interface, which is claimed to offer more flexibility (e.g., can have multiple plots open):

```
fig = plt.figure()
ax = fig.add_subplot(1,1,1)
ax.plot(x, y)
ax.set_xlim([xmin,xmax])  # note variant on state method command
ax.set_ylim([xmin,xmax])  # note variant on state method command
plt.set_xlabel(label)     # note variant on state method command
plt.set_ylabel(label)     # note variant on state method command
plt.draw()                # required in object mode to draw plot
plt.show()                # needed to show figure in non-interactive
mode
plt.savefig(file)
```

Many data sets in astronomy are "multidimensional", and it can be powerful to display additional information on point graphs using color and point size:

```
scat=ax.scatter(x,y,c=z,vmin=vmin,vmax=vmax,s=size,cmap=cmap,linewidth=linewidth)
```

where z and size can be arrays of colors (or values to be scaled using vmin and vmax) and point sizes. If you are mapping a range of z values, the vmin= and vmax= keywords give the minimum and maximum data values; all of your data will be scaled into an 8-bit range between these two values. These will then be displayed in "color" depending on the color map that is specified; a color map determines what colors are used for each of the scaled data values. The default matplotlib color map is 'jet', which scales things from blue to red, but *many* different colormaps are available, and it is possible to define your own. Note that there is fair amount of literature and opinion on good choices of color map (see, e.g. here), and that most believe that the default 'jet' colormap is a poor choice.

Exercise: plot isochrone color coded by age, with radius coded by point size

Multipanel plots:

```
plt.subplot(2,1,1)
plt.plot(x,y)
plt.subplot(2,1,2)
plt.plot(x2,y2)
plt.subplots_adjust([hspace=x],[wspace=x],[sharex=True])
# need to control axis labels?
plt.xticks([])
#alternatively, using axis objects
fit=plt.figure()
```

```
ax=fig.add_subplot(2,1,1)
ax.plot(x,y)
# or to get a grid of axes objects in a single command:
f, ax=plt.subplots(ny,nx)
ax[iy][ix].plot(x,y)
ax[iy][ix].set_xticks([])
ax[iy][ix].set_visible(False)
```

A useful Python command for "packing" the plots onto the page:

```
plt.tight_layout()
```

(also available through the subwindows button on the matplotlib window).

Exercise: plot temperature vs log g, absolute mag on two plots, sharing x axis.

For even more flexibility, see gridspec.

Histograms: binning, limits, labels. Python hist.

```
plt.hist(data,[bins=n|bins=binarray])
```

Understand. . .

6.2 "Image" plots

It is often the case that you have 3D information, i.e., a variable as a function of two other variables. A common example is an image, which is a array of intensities at a grid of different (x,y) pixel locations. But there are many other possibilities as well, e.g., the value of some function across a 2D parameter space.

3D data/images can be displayed in multiple ways. One such way is to encode the variable as a color or intensity, i.e., visualize the data as an image. In Python, this is done using the imshow command. When you display an image, you are displaying data as a function of each (x, y) pixel position. It is likely that these pixel position represent some physical quantity (e.g., position on the sky, value of some independent variables, etc.), so you may want to associate this physical quantity with the pixel positions; this can be done at the plotting level using the extent= keyword in imshow.

Note that imshow by default may attempt to "smooth" your image; this behavior can be avoid by using the interpolation='none' keyword.

```
# generate "unit" grids, note useful numpy.mgrid
y, x = np.mgrid[0:100,0:100]
# alternatively, generate grids with fixed number of points between two values
y, x = np.mgrid[0:5:200j, 3500:5500:200j]
plt.imshow(y,[vmin=],[vmax=],[extent=ymin:ymax,xmin:xmax],[interpolation='none'])
plt.colorbar
```

The "unit" grids can be used to generate functions of two variables, e.g.

```
r2=x**2+y**2
plt.imshow(r2,[vmin=],[vmax=],extent=[ymin:ymax,xmin:xmax])
```

Exercise: display vmacro relation across HR diagram. Add isochrone points. Add observed data points.

An alternative way to display 3D images is via a countour plot, e.g. matplotlibcontour plots:

```
cs=plt.contour(x,y,z)
plt.clabel(cs)
```

Be able to generate a simple image plot from a 2D array of data values.

6.3 Other plots

6.3.1 IDL examples

6.4 Event handling

It can be very useful for a program to be to interact with a plot, i.e., for the program to take some action based on input from the plot. This can be accomplished by setting up an "event handler," which provides code that responds to events such as mouse click, mouse motion, key press, etc.

See matplotlib event handling

For example, in matplotlib windows, a basic event hander is set up that displays the position of the cursor in the plot window. But this can be extended, e.g.:

```
fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(x,y)

#define an event handler to work with key presses
def onpress(event):
    print 'key=%s, x=%d, y=%d, xdata=%f, ydata=%f'%(
        event.key, event.x, event.y, event.xdata, event.ydata)

cid = fig.canvas.mpl_connect('key_press_event', onpress)
```

This prints to screen, but what if you want to get information back to a calling routine? Example: say you are fitting a relation to a data set, and recognize that outliers are throwing the fit off: you want to identify these based on a residual plot. Set up blocking and return of data, e.g.:

```
def onpress(event):
    ''' the event handler '''
    # define variables as global so that they can be returned
    global button, xdata, ydata
    print 'key=%s, x=%d, y=%d, xdata=%f, ydata=%f'%(
        event.key, event.x, event.y, event.xdata, event.ydata)
    # load the global variables
    button = event.key
    xdata = event.xdata
    ydata = event.ydata
    # once we have an event, stop the blocking
    fig.canvas.stop_event_loop()
```

```
def mark() :
    ''' blocking routine to wait for keypress event and return data'''s
    # start the blocking, wait indefinitely (-1) for an event
    fig.canvas.start_event_loop(-1)
    # once we get the event, return the event variables
    return button, xdata, ydata

# call the blocking routine
    button, xdata, ydata = mark()
```

Generate random sequence (numpy.random.normal) with an outlier, and identify it from a plot.

7 Algorithms

7.1 Writing a Program

Simple suggestions:

1. Make sure you fully understand the question/problem before starting to write the code. Outline the methodology you will use in words, diagrams, or figures, before getting caught up in the syntax of coding.
2. Generate the code in pieces. Consider writing out all the comments before the actual coding. Include some debugging statements, considering the possibility of building in a "verbosity" level from the start. Consider building in error trapping from the start.
3. Test code. Use it in an unexpected way to see what happens. Do your best to find what is wrong with it. Try to break it.
4. Clean up code. Consolidate repeated code, find a more efficient or transparent way to write things.
5. Fully comment code. Document the overall strategy and points of the code that are not immediately understandable from the code itself.

7.2 Speed and scaling

7.3 Lists and list matching

(sorting, finding items, matching lists, etc.)

7.4 Random number generation

(see NR chapter 7).

A random distribution of numbers is useful for simulating data sets to test analysis routines. Research your random number generator if what you're doing is important since crappy ones exist.

Computer generated random numbers are repeatable. They generally start with a *seed number* (usually the current clock time) if the user doesn't specify one. Users should record whichever seed they use in case repeatability is desired.

Lowest level random number generators give *uniform deviates*, random numbers that lie within a specified range. (usually 0 to 1 for floats, or 0 to $2^{32} - 1$ or $2^{64} - 1$ for integers).

Python stuff:

```
>>> random.random
>>> random.seed
>>> numpy.random.random
>>> numby.random.seed
```

Transformation Method: To generate random numbers for some other distribution, such as a Gaussian, Poisson, luminosity function, or mass function: consider a cumulative distribution of the desired function (whose values range between zero and one). Generate a uniform random deviate between 0 and 1; these are your y-values. Solve your function for x in terms of y, and calculate all the values of x that correspond to your y-values (the random numbers). This does require that you can integrate your function, then invert the integral. (see NR, figure 7.3.1)

In-class exercise: generate random deviates for a “triangular” distribution: $p(x) \propto x$. What does the constant of proportionality need to be in order to make the integral equal to 1 (aka: a probability distribution)?

$$\begin{aligned}\int p(x)dx &= \int Cxdx \\ &= C\int xdx \\ C &= 2\end{aligned}$$

Use the relation to generate random deviates, and plot them with a histogram.

$$\begin{aligned}y &= 2x \\ F &= \int y \ \mathrm{d}x = x^2 \\ x(F) &= \sqrt{(F)}\end{aligned}$$

Rejection Method: If you can’t integrate and invert your function: choose a function that you *can* integrate and invert that is always higher than your desired distribution. This is the comparison function, $c(x)$. As before, choose a random deviate and find the corresponding x values. Calculate the (x) value of both the desired function and comparison function. Choose a uniform deviate between 0 and $c(x)$. If it is larger than $f(x)$, reject your value and start again. This requires two random deviates for each attempt. The number of attempts before you get a deviate within your desired distribution depends on how close your comparison function is to your desired function (see NR figure 7.3.2).

Example demonstration: truncated Gaussian distribution. Generate random deviates with rejection method.

Understand how to use and implement the transformation method for getting deviates from any function that is integrable and invertible.

See NR for several ways to generate deviates in desired functions.

Gaussian distribution: Applies to many physical quantities, such as the Maxwellian speed distribution, and in general as a reasonable approximation for a large mean.

```
>>> numpy.random.normal
```

$$P(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$$

Poisson distribution: Used for counting statistics.

```
>>> numpy.random.poisson
```

$$P(x, \mu) = \frac{\mu^x e^{-\mu}}{x!}$$

In class exercise: simulate some data, e.g. a linear relation (x=1-10, y=x) with a Gaussian (mean=0, sigma=varius) scatter, and plot it. Alternate: CMD from isochrones with Poisson scatter in colors and magnitudes.

Know how to use canned functions for generating uniform deviates, Gaussian deviates, and Poisson deviates.

7.5 Interpolation

(see NR chapter 3)

Interpolation can be used if you have a tabulated or measured set of data, and want to estimate the values at intermediate locations in your data set, such as inverting a function (e.g. random deviates as above, wavelength calibration, etc.), re-sampling data (images or spectra), etc.

- **Linear interpolation:** This is the simplest interpolation. To find the value $f(x)$ at some intermediate location x between two points, x_i and x_{i+1}:

$$f(x) = Ay_i + By_{i+1}$$
$$A = \frac{x_{i+1} - x}{x_{i+1} - x_i}$$
$$B = \frac{x - x_i}{x_{i+1} - x_i}$$

- **Polynomial interpolation:** (NR 3.2) The *Lagrange formula* gives the value of a polynomial of degree k going through $k + 1$ points at some arbitrary position, x, as a function of the tabulated data values:

$$L(x) = \sum_{j=0}^{k} y_j \ell_j(x)$$
$$\ell_j(x) = \prod_{0 \le m \le k, m \ne j} \frac{x - x_m}{x_j - x_m}$$

  Note that this provides the values of the interpolating polynomial at any location without providing the coefficients of the polynomial itself. The coefficients are determined such that the polynomial fit goes exactly through the tabulated data points (the measured data).

Be aware that higher order is not necessarily better. A classic example:

$$f(x) = \frac{1}{(1 + 25x^2)}$$

Higher order is bad here because…?

There are also issues with even orders because this leads to using more tabulated points on one side of the desired value than the other (since even order means odd number of data points). At high order, the polynomial can go wildly off, especially near data edges.

Another problem with piecewise polynomial interpolation, even at low order, is that the input data values that are used to interpolate to a desired location change as you cross each data point. This leads to abrupt changes in derivatives (infinite second derivative) of the interpolated function. For some applications, this can cause problems, for example, if you are trying to fit an observed data set to some tabulated series of models and are using derivatives to get the best fit.

Spline interpolation: This issue can be overcome using spline interpolation, which gives interpolated values that go through the data points but also provides *continuous second derivatives.* Doing so, however, requires the use of non-local interpolation, i.e. the interpolating function includes values from all tabulated data points. Generally, people use cubic fits to the second derivatives, leading to *cubic spline interpolation.* To do this, specify boundary conditions at the ends of the data range. Usually, a *natural spline* is adopted, with zero second derivatives at the ends, but it is also possible to specify a pair of first derivatives at the ends.

Cubic splines are probably the most common form of interpolation, though not necessarily the best in all circumstances.

Python implementation: `scipy.interpolate.interp1d` calculates interpolating coefficients for linear and spline interpolation, and can then be called to interpolate to desired position(s):

```
>>> from scipy import interpolate
>>> intfunc = interpolate.interp1d(xdata,ydata,
        kind='linear'|'slinear'|'quadratic'|'cubic'|order)
>>> intfunc(x)  # returns interpolated value(s) at x
```

Exercise:

1. Define some function, $f(x)$
   `def f(x):` ...Your function here...
2. For some given domain, plot your function at high sampling, e.g.:

   ```
   # get a high sampled array of independent variable
   x = numpy.linspace(xmin,xmax,1000)
   # plot the relation as a smooth line
   plt.plot(x,f(c))
   ```

3. For some given domain, sample your function with N points (evenly spaced, or maybe random), e.g.:

   ```
   np.linspace(xmin,xmax,n) # provides n xdata points in your domain
   plt.plot(xdata,func(data),'o') # plot data points as points
   ```

4. Determine the interpolating polynomial for linear and cubic spline, and overplot the interpolated function.

   ```
   from scipy import interpolate
   # get the interpolating function (you need to choose which type)
   intfunc = interpolate.interp1d(xdata,func(xdata),
       kind = 'linear'|'slinear'|'quadratic'|'cubic'|order)
   # plot interpolated data at high sampling
   plt.plot(x,intfunc(x))
   ```

Understand what is meant by piecewise polynomial interpolation and spline interpolation and under what circumstances it might be better to use the latter.

7.6 Fourier analysis basics and sinc interpolation

(see NR chapter 12)

Consider a function in two representations: values in "physical" space (e.g., time or location) vs. values in "frequency" space (e.g., temporal frequency or wavenumber). The two are related by Fourier transforms:

$$H(f) = \int h(x) e^{-2\pi i f x}\, \mathrm{d}x$$
$$h(x) = \int H(f) e^{-2\pi i f x}\, \mathrm{d}f$$

Note that different implementations use different sign conventions, for example: *angular* frequency ($\omega = 2\pi f$).

The physical interpretation is that a function can be decomposed into the sum of a series of sine waves, with different amplitudes and phases at each wavelength/frequency. *Because we have amplitude and phase, the Fourier transform is, in general, a complex function.*

Fourier transforms can be determined for discrete (as opposed to continuous) streams of data using the *discrete* Fourier transform, which replaces the integrals with sums. Algorithms have been developed to compute the discrete Fourier transform quickly by means of the *Fast Fourier Transform (FFT)*; generally, this is done for data sets that are padded to a length of a power of 2 (???). However, the FFT algorithm requires equally spaced data points. For unequally spaced points, the full discrete Fourier transform is required.

Python implementations: `numpy.fft`, `scipy.fftpack`

```
import matplotlib.pyplot as plt
import numpy as np

# generate sine wave
x = np.linspace(0, 10000., 8192)
y = np.sin(100*x)
plt.plot(x,y)

# Do a fast fourier transform
f = np.fft.fft(y)

# plot amplitude, try to get frequencies right...
plt.plot(np.abs(f)) \\
plt.plot(np.fft.fftfreq(8192), np.abs(f))
plt.plot(np.fft.fftfreq(8192,100./8192), np.abs(f))
```

Two particular operations involving pairs of functions are of general interest: convolution and cross-correlation.

- **Convolution**

  $$g(x) * h(x) = \int g(x')h(x - x')\mathrm{d}x'$$

  Usually, convolution is seen in the context of *smoothing*, where $h(x)$ is a normalized function (with a sum of unity), centered on zero. Convolution is the process of running this function across an *input function* to produce a smoothed version. Note that the process of convolution is computationally expensive; at each point in a data series, you have to loop over all of the points (or at least those that contribute to the convolution integrat [integrand? integral?]). In some cases, it can be advantageous to consider convolution in the Fourier domain because of the *convolution theorem*, which states that convolving two functions in physical space is equivalent to multiplying the transforms of the functions in Fourier space. Multiplication of two functions is computationally faster than convolution).

  `numpy.convolve`

- **Cross-correlation**

  $$g(x) \star h(x) = \int g(x')h(x + x')\mathrm{d}x'$$

  Cross-correlation is actually the same as convolution if h is a symmetric function, but is usually used quite differently. It is generally considered a function of the *lag*, x. Multiply two functions, calculate the sum, then shift one of the functions and do it again. Then look at the sums as a function of the shift. For two similar functions, the cross-correlation will be a maximum when the two functions "line up", so this is useful for determining shifts between two functions, e.g. spatial shifts of images, or spectral shifts from velocity (but in velocity space, i.e. log(wavelength), not linear wavelength space). Cross-correlation can also be computed in the Fourier domain. It is equivalent to multiplying the Fourier transform of one function by the complex conjugate of the Fourier transform of the other. Note that the height and width of the cross-correlation function has information about the degree to which the two functions are similar. A specific case of cross-correlation is the *autocorrelation function*, which is the cross-correlation of a function with itself.

Understand the basics of Fourier transforms. Understand what convolution and cross-correlation are, and understand the convolution theorem.

Back to interpolation... See Bracewell chapter 10 for sinc interpolation and the sampling theorem.

When we sample a function at finite locations, we are multiplying by a *sampling function*, a discrete set of impulses spaced by Δx. In the Fourier domain, this is *convolving* by the Fourier transform of the sampling function, a discrete set of frequencies spaced by $1/\Delta x$.

An important and useful result for interpolation is called the *sampling theorem* which arises in the case of a *band-limited function*, a function that has no power above some critical frequency known as the *Nyquist frequency*. In this case, if the function is sampled at sufficient frequency, (twice the Nyquist frequency), it is possible to recover the entire function. Figure 10.3 from Bracewell shows the idea graphically.

What to do:

1. Fourier transform the sampled function.
2. Multiply by a box function.
3. Fourier transform back.

Alternatively, convolve the sampled function with the Fourier transform of a box function, which is the sinc function:

$$\mathrm{sinc}(x) = \frac{\sin(x)}{x}$$

This leads to *sinc interpolation*, which is the ideal interpolation in the case of a band-limited function. As mentioned at the beginning of this section (out of place), this is one application of Fouier decomposition, which is useful when considering a series of data points.

Understand the sampling theorem, and be able to explain it by reference to how the Fourier transform of a band-limited function that is sufficiently sampled compares to the transform of one that is undersampled.

Interpolation and Uncertainties:

Generally all interpolation schemes can be considered as a sum of data values multiplied by a set of interpolation coefficients at each point:

$$y(x) = \sum_{i=0}^{n} y_i P_i(x)$$

So far, we have been considering interpolation in perfect data. But if there are uncertainties on the data points, there will be uncertainties in the interpolated values. These uncertainties can be derived by error propagation (if the errors on the individual data points are uncorrelated):

$$\sigma(y(x))^2 = \sum \sigma(y(x_i))^2 P_i(x)^2$$

However, the interpolated errors are now themselves correlated, so using them properly requires the *full covariance matrix.*

$$\sigma_x^2 = \sigma_u^2 \left(\frac{\partial x}{\partial u}\right)^2 + \sigma_v^2 \left(\frac{\partial x}{\partial v}\right)^2 + 2\sigma_{uv}^2 \frac{\partial x}{\partial u}\frac{\partial x}{\partial v}$$

$$\sigma_{uv}^2 = \lim_{n\to\infty} \frac{1}{N} \sum (u_i - < u >)(v_i - < v >)$$

In the presence of noise (uncertainties), it is often advantageous to *model* the data by fitting an underlying function to a tabulated set of data points with some merit function criteria for determing what function provides the best fit. The fit can then be used to predict values at arbitrary location.

Be aware of mixing and matching interpolation and fitting: doing fits on data sets with uncorrelated errors is substantially easier than doing them on data sets with correlated errors.

7.7 Differentiation and integration

- Differentiation: finite differences
- Integration: simple rules, extended rules, reaching a desired accuracy.
- Monte Carlo integration

7.8 Differential equations

- Runge-Kutta
- boundary value problems.

8 Fitting

8.1 Overview: frequentism vs Bayesian

Given a set of observations/data, one often wants to summarize and get at underlying physics by fitting some sort of model to the data. The model might be an empirical model or it might be motivated by some underlying theory. In many cases, the model is parametric: there is some number of parameters that specify a particular model out of a given class.

The general scheme for doing this is to define some merit function that is used to determine the quality of a particular model fit, and choose the parameters that provide the best match, e.g. a minimum deviation between model and data, or a maximum probability that a given set of parameters matches the data.

It is also important to understand how reliable the derived parameters are. The extent to which the model is consistent with your understanding of uncertainties on the data is an indication of how good the model fit actually is.

There are two different "schools" about model fitting: Bayesian and frequentism.

Frequentism (also called the classical approach). Consider how *frequently* a data set might be observed given some underlying model. $P(D|M)$ – *probability of observing a data set given a model*. The model that produces the observed data most frequently is viewed as the correct underlying model, and as such, gives the best parameters, along with some estimate of their uncertainty.

Bayesian $P(M|D)$ – *probability that a model is correct given a data set*. It allows for the possibility that external information may prefer one model over another, and this is incorporated into the analysis as a *prior*. It considers the probability of different models, and hence, the probability distribution functions of parameters. Examples of priors: fitting a Hess diagram with a combination of SSPs, with external constraints on allowed ages; fitting UTR data for low count rates in the presence of readout noise.

The frequentist paradigm has been mostly used in astronomy up until fairly recently, but the Bayesian paradigm has become increasingly widespread. In many cases they can give the same result, but with somewhat different interpretation. In some cases, results can differ. The basic underpinning of Bayesian analysis comes from **Bayes theorem** of probabilities:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

where $P(A|B)$ is the conditional probability, i.e. the probability of A given that B has occurred. Imagine there is some joint probability distribution function of two variables, $p(x, y)$ (see ML 3.2; this could be a distribution of stars as a function of effective temperature and luminosity). Think about slices in each direction to get to the probability at a given point, and we have:

$$p(x, y) = p(x|y)p(y) = p(y|x)p(x)$$

This gives:

$$p(x|y) = \frac{p(y|x)p(x)}{p(y)}$$

which is Bayes theorem. Bayes theorem also relates the conditional probability distribution to the *marginal* probability distribution:

$$p(x) = \int p(x|y)p(y)\ \mathrm{d}y$$
$$p(y) = \int p(y|x)p(x)\ \mathrm{d}y$$
$$p(x|y) = \frac{p(y|x)p(x)}{\int p(y|x)p(x)\ \mathrm{d}x}$$

This is all fairly standard statistics, but the Bayesian paradigm extends this to the idea that the quantity A or B can also represent a hypothesis, or model, i.e. the relative probability of different models.

The context of Bayesian analysis involves the probability of models (instead of data), and we have:

$$P(M|D) = \frac{P(D|M)P(M)}{P(D)}$$

In this case, P(D) is a normalization constant, P(M) is the prior on the model, which, in the absense of any additional information, is equivalent for all models: a *noninformative prior.* (However, a noninformative prior can itself be a prior; a uniform prior is not necessarily invarient to a change in variables. For example, a uniform prior in the logarithm of a variable is not uniform in the variable iteself). In the noninformative case, the Bayesian result is the same as the frequentist maximum likelihood. However, in the Bayesian analysis we'd want to calculate the full probability distribution function for the model parameter, μ. More on that later (insert section reference here!)

In practice, frequentist analysis yields parameters and their uncertainties, while Bayesian analysis yields probability distribution functions of parameters. The latter is often more computationally intensive to calculate. Bayesian analysis includes explicit priors.

For some more detailed discussion, see http://arxiv.org/abs/1411.5018

Understand the basic conceptual differences between a frequentist and a Bayesian analysis. Know Bayes' theorem. Understand the practical differences.

Starting with a frequentist analysis (which often provides a component of the Bayesian analysis): given a set of data and some model (with parameters), consider the probability that the data will be observed if the model is correct, and choose the set of parameters that maximizes this probability.

For example, when fitting a straight line through the data, the best fit is determined by the method of *least squares.* This comes from the consideration that observed data points have some measurement uncertainty, σ (for now, all points have homoschedastic (equal) uncertainties that are distributed according to a Gaussian). The questions to ask are:

1. What is the probability is of observing a given data value from a known Gaussian distribution?
2. What is the probability of observing a series of two independently drawn data values?

Suppose we are fitting N data points (x_i, y_i), where $i = 0, \ldots, N-1$ to a model that has M adjustable parameters a_j, where $j = 0, \ldots, M-1$. The model predicts a functional relationship between the measured independent and dependent variables:

$$y(x) = y(x|a_0 \ldots a_{M-1})$$

where the notation indicates dependence on the parameters explicitly on the right-hand side, following the vertical bar.

So given some model $y(x_i|a_j)$, the probability of observing a series of data points is:

$$P(D|M) \propto \prod_{i=0}^{N-1} \left\{ \exp\left[-\frac{1}{2}\left[\frac{y_i - y(x_i|a_j)}{\sigma}\right]^2 \right] \Delta y \right\}$$

where a_j is the jth parameter of the model, and Δy is some small range in y.

Maximizing the probability is the same thing as maximizing the *logarithm* of the probability, which is the same thing as *minimizing* the *negative* of the (natural?) logarithm ($-\log x = \log x^{-1} = \log(1/x)$). In other words, minimize:

$$-\log P(D|M) = \sum_{i=0}^{N-1} \left\{ \frac{1}{2}\left[\frac{y_i - y(x_i)}{\sigma}\right]^2 \right\} - N\log(\Delta y) + const$$

If σ is the same for all points, then this is equivalent to minimizing

$$\sum_{i=1}^{N-1} \left[y_i - y\left(x_i|a_j\right)\right]^2$$

which is the least squares fit.

Understand how least squares minimization can be derived from a maximum likelihood consideration in the case of normally-distributed uncertainties.

Consider a simple application, where we have multiple measurements of some quantity, so the model is $y(x_i) = \mu$, and we want to determine the most probable value of μ. To minimize $-\log P(D|M)$, take the derivative and set it equal to zero. This gives:

$$\frac{\mathrm{d}(-\log P(D|M))}{\mathrm{d}\mu} = 2\sum_{i=0}^{N-1}(y_i - \mu) = 0$$

$$\mu = \sum \frac{y_i}{N}$$

which should look familiar (μ = mean). To calculate the uncertainty on the mean, use error propagation:

$$\sigma(\mu)^2 = \sigma_{y_0}^2\left(\frac{\partial\mu}{\partial y_0}\right)^2 + \sigma_{y_1}^2\left(\frac{\partial\mu}{\partial y_1}\right)^2 + \ldots$$

$$\sigma_\mu^2 = \sum\frac{\sigma^2}{N^2} = \frac{\sigma^2}{N}$$

$$\sigma_\mu = \frac{\sigma}{\sqrt{N}}$$

Side note: Error propagation

$$x = f(u, v)$$

$$\sigma_x^2 = \sigma_u^2\left(\frac{\partial x}{\partial u}\right)^2 + \sigma_v^2\left(\frac{\partial x}{\partial v}\right)^2$$

For *heteroschedastic* (unequal) uncertainties on the data points:

$$-\log P(D|M) = \sum_{i=0}^{N-1}\left[\frac{(y_i - y(x_i))^2}{\sigma_i^2}\right] + const \equiv \chi^2$$

Here, we are minimizing a quantity called χ^2. An important thing about χ^2 is that the probability of a given value of χ^2 given N data points and M parameters can be calculated *analytically*. This is called the χ^2 *distribution for* $\nu = N - M$ *degrees of freedom.* (See scipy.stats.chi2. The cumulative density function (cdf), for example, needs to be in some range, like 0.05 0.95).

The quality of a fit can be qualitatively judged by the *reduced* χ^2, or χ^2 per degree of freedom:

$$\chi_\nu^2 = \frac{\chi^2}{N-M} = \frac{\chi^2}{\nu}$$

which is expected to have a value near unity. However, it is the probability of χ^2 that should be calculated, not χ^2 itself since the spread in χ_ν^2 depends on ν (the standard deviation is $\sqrt{2\nu}$).

It is important to recognize that this analysis depends on the assumption that the uncertainties are distributed according to a normal (Gaussian) distribution.

χ^2 can be used as a method for checking uncertainties, or even determining them (at the expense of being able to say whether your model is a good representation of the data). See ML 4.1.

Know specifically what χ^2 is and how to use it. Know what reduced χ^2 is, and understand degrees of freedom.

As with linear least squares, to minimize χ^2, take the derivative with respect to the parameters, and set it equal to zero. For our simple model of measuring a *single quantity*, we have:

$$\frac{\mathrm{d}(-\log P(D|M))}{\mathrm{d}\mu} = 2\sum\frac{(y_i-\mu)}{\sigma_i^2} = 0$$

$$\sum\frac{y_i}{\sigma_i^2} - \mu\sum\frac{1}{\sigma_i^2} = 0$$

$$\mu = \frac{\sum\frac{y_i}{\sigma_i^2}}{\sum\frac{1}{\sigma_i^2}}$$

i.e., a weighted mean. Again, you can use error propagation to get (work not shown):

$$\sigma_\mu = \left(\sum\frac{1}{\sigma_i^2}\right)^{-1/2}$$

A more complicated example: fitting a straight line to a set of data. Here, the model is

$$y(x_i) = a + bx_i$$

and χ^2 is:

$$\chi^2 = \sum_{i=0}^{N}\left[\frac{[y_i-(a+bx_i)]^2}{\sigma_i^2}\right]$$

$$= \sum_{i=0}^{N}\left[\frac{[y_i-a-bx_i]^2}{\sigma_i^2}\right]$$

Again, we want to minimize χ^2, so we take the derivatives with respect to each of the parameters and set them to zero:

$$\frac{\partial\chi^2}{\partial a} = -2\sum\frac{(y_i - a - bx_i)}{\sigma_i^2} = 0$$
$$\frac{\partial\chi^2}{\partial b} = -2\sum\frac{x_i(y_i - a - bx_i)}{\sigma_i^2} = 0$$

Separating out the sums, we have:

$$\sum\frac{y_i}{\sigma_i^2} - a\sum\frac{1}{\sigma_i^2} - b\sum\frac{x_i}{\sigma_i^2} = 0$$

$$\sum\frac{x_iy_i}{\sigma_i^2} - a\sum\frac{x_i}{\sigma_i^2} - b\sum\frac{x_i^2}{\sigma_i^2} = 0$$

or

$$aS + bS_x = S_y$$
$$aS_x + bS_{xx} = S_{xy}$$

where the various S are a shorthand for the sums:

$$S = \sum\frac{1}{\sigma_i^2}$$

$$S_x = \sum\frac{x_i}{\sigma_i^2}$$

$$S_{xy} = \sum\frac{x_iy_i}{\sigma_i^2}$$

$$S_{xx} = \sum\frac{x_iy_i}{\sigma_i^2}$$

$$S_y = \sum\frac{y_i}{\sigma_i^2}$$

which is a set of two equations with two unknowns. Solving, you get:

$$a = \frac{S_{xx}S_y - S_xS_{xy}}{SS_{xx} - S_x^2}$$

$$b = \frac{SS_{xy} - S_xS_y}{SS_{xx} - S_x^2}$$

Know how to derive the least squares solution for a fit to a straight line.

We also want the uncertainties in the parameters. Again, use propagation of errors to get (work not shown):

$$\sigma_a^2 = \frac{S_{xx}}{SS_{xx} - S_x^2}$$

$$\sigma_b^2 = \frac{S}{SS_{xx} - S_x^2}$$

Finally, we will want to calculate the probability of getting χ^2 for our fit, in an effort to understand

1. if our uncertainties are not properly calculated and normally distributed
2. if our model is a poor model

Let's do it. Simulate a data set, fit a line, and output parameters, parameter uncertainties, χ^2, and probability of χ^2.

8.2 General linear fits

We can generalize the least squares idea to any model that is some linear combination of terms that are a function of the independent variable, e.g.

$$y = a_0 + a_1 f_1(x) + a_2 f_2(x) + a_3 f_3(x) + \dots$$

such a model is called a *linear* model because it is linear in the *parameters*, a_i but not necessarily linear in the independent variable, x. The model could be a polynomial of arbitrary order, but could also include trigonometric functions, etc. We write the model in simple form:

$$y(x) = \sum_{k=0}^{M-1} a_k X_k(x)$$

where there are M parameters, N data points, and $N > M$. The χ^2 merit function (???) can be written as:

$$\chi^2 = \sum_{i=0}^{N-1} \left[\frac{(y_i - \sum_{k=0}^{M-1} a_k X_k(x_i))^2}{\sigma_i^2} \right]$$

Minimizing χ^2 leads to the set of M equations:

$$0 = \sum_{i=1}^{N-1} \frac{1}{\sigma_i^2} \left[y_i - \sum_{j=0}^{M-1} a_j X_j(x_i) \right] X_k(x_i)$$

where $k = 0, \dots, M-1$.

Separating the terms and interchanging the order of the sums gives:

$$\sum_{i=0}^{N-1} \frac{y_i X_k(x_i)}{\sigma_i^2} = \sum_{j=0}^{M-1} \sum_{i=0}^{N-1} \frac{a_j X_j(x_i) X_k(x_i)}{\sigma_i^2}$$

Define:

$$\alpha_{jk} = \sum_{i=0}^{N-1} \frac{X_j(x_i) X_k(x_i)}{\sigma_i^2}$$

$$\beta_j = \sum_{i=0}^{N-1} \frac{X_j(x_i) y_i}{\sigma_i^2}$$

then we have the set of equations:

$$\alpha_{jk} \alpha_j = \beta_k$$

for $k = 0, \dots, M-1$.

Know the equations for a general linear least squares problem, and how they are derived.

Sometimes these equations are cast in terms of the *design matrix*, A, which consists of N measurements of M terms:

$$A_{ij} = \frac{X_j(x_i)}{\sigma_i}$$

with N rows and M columns. Along with the definition:

$$b_i = \frac{y_i}{\sigma_i}$$

we have:

$$\alpha = A^T \cdot A$$
$$\beta = A^T \cdot b$$

where the dots are for the matrix operation that produces the sums. This is just another notation for the same thing, introduced here in case you run across this language or formalism.

For a given data set, α and β can be calculated either by doing the sums or by setting up the disign matrix and using matrix arithmetic. Then solve the set of equations for α_k.

Note that this formulation applies to problems with multiple independent variables, e.g., fitting a surface to a set of points; simply treat x as a vector of data, and the formulation is exactly the same.

8.3 Solving linear equations

This is just a linear algebra problem, and there are well-developed techniques (see NR chapter 2). For the most simple case, invert the matrix α to get

$$\alpha_k = \alpha^{-1}_{jk}\beta_k$$

A simple algorithm for inverting a matrix is called a *Gauss-Jordan elimination.* In particular, NR recommends the use of singular value decomposition for solving all but the simplest least squares problems; this is especially important if your problem is nearly singular, i.e. where two or more of the equations may not be totally independent of each other: fully singular problems should be recognized and redefined, but it is possible to have non-singular problems encounter singularity under some conditions depending on how the data are sampled. See NR 15.4.2 and chapter 2 (in these notes?).

As an aside, note that it is possible to solve a linear set of equations significantly faster than inverting the matrix through various types of matrix decomposition, e.g., LU decomposition and Cholesky decomposition (see NR Chapter 2). There may be linear algebra problems that you encounter that only require the solution of the equations and not the inverse matrix. However, for the least squares problem, we often do want the inverse matrix $C = \alpha^{-1}$, because its elements have meaning. Propagation of errors gives:

$$\sigma^2(a_k) = C_{jj}$$

i.e., the diagonal elements of the inverse matrix. The off-diagonal elements give the covariances between the parameters, and the inverse matrix, C, is called the *covariance matrix.*

A Python implementation of matrix inversion and linear algebra in general can be found in scipy.linalg. Here is the summed matrix implementation for a straight line (but the functional form only comes in through the derivative function.):

```
def deriv(x) :
    # if x is numpy array, then we can return vectors of derivatives
    try :
        return [np.ones(len(x)),x]
    except :
        return [1.,x]

# data points in (x,y)
# loop over parameters, sums over data points (y) are done with vector arithmetic
for k in np.arange(npar) :
    beta[k] = np.sum(deriv(x)[k]*y/sigma**2)
```

```
        for j in np.arange(npar) :
            alpha[k,j] = np.sum(deriv(x)[k]*deriv(x)[j]/sigma**2)
    c=np.linalg.inv(alpha)
    print np.dot(c,beta)
```

Here is the design matrix approach to the sums:

```
    A = np.vander(x, 2)  # Take a look at the documentation to see what this function does.
    ATA = np.dot(A.T, A / yerr[:, None]**2)
    w = np.linalg.solve(ATA, np.dot(A.T, y / yerr**2))
    V = np.linalg.inv(ATA)
```

Linear least squares is also implemented in `astropy` in the modeling module using the LinearLSQFitter class (which uses `numpy.linalg`); `astropy` has a number of standard models to use, or you can provide your own custom model.

```
    from astropy.modeling import models, fitting
    fit_p = fitting.LinearLSQLSQFitter()
    p_init = models.Polynomial1D(degree=degree)
    pfit = fit_p(p_init, x, data)
```

`astropy` has a number of common models, but you can also define your own model using models.custom_model, by supplying a function that returns values, and a function that returns derivatives with respect to the parameters.

Be able to computationally implement a linear least squares solution to a problem.

Note that for problems with large numbers of parameters, the linear algebra can become very computationally expensive. In many cases, however, the matrix that represents these problems may only be sparsely populated, i.e. so-called sparse matrices (see Numerical Recipes 2.7, Figure 2.7.1).

Example: spectral extraction

In such cases, there are methods that allow the equations to be solved significantly more efficiently.

However, not all fits are linear in the parameters. Which of the following are not?

- $y = a_0 x + a_1 e^{-x}$
- $y = a_0(1 + a_1 x)$
- $y = a_0 \sin(x - a_1)$
- $y = a_0 \sin^2 x$
- $y = a_0 e^{(-(x-a_1)^2/a_2^2)}$

8.4 Nonlinear fits

In a linear fit, the χ^2 surface is parabolic in parameter space, with a single minimum, and linear least squares can be used to determine the location of the minimum. In the non-linear case, however, the χ^2 surface can be considerably more complex, and there is a (good) possibility that there are multiple minima, so one needs to be concerned about finding the global minimum and not just a local minimum. Because of the complexity of the surface, the best fit is found by an iterative approach.

A conceptually simple approach would be a *grid search*, where one simply tries all combinations of parameters and finds the one with the lowest χ^2. Obviously, this is extremely computationally inefficient, especially for problems with more than a few parameters. One is also forced to decide on a step size in the grid, although you might imagine a successively refined grid as you proceed. But in general, this method is not recommended, apart from occaisionally trying it to try to ensure that your more efficient solution is not landing in a local minimum.

Better approaches attempt to use the χ^2 values to find the minimum more efficiently. They can generally be split into two classes: those that use the derivative of the function and those that don't. If you can calculate derivatives of χ^2 with respect to your parameters, then this provides information about how far you can move in parameter space towards the minimum. If you can't calculate derivates, you can evaluate χ^2 at several different locations, and use these values to try to work your way towards the minimum.

With derivatives, the approach has a fairly close parallel to the linear least squares problem. Around the final minimum, the χ^2 surface can be approximated as a parabola, and it is possible to correct the solution to the minimum solution if one can arrive at a set of parameters near to the final minimum. This is acheived via the set of equations:

$$\sum_{i=0}^{M-1} \alpha_{kl} \delta a_1 = \beta_k$$

where

$$\beta_k = -\frac{1}{2}\frac{\partial \chi^2}{\partial a_k} = \sum_{i=0}^{N-1} \frac{(y_i - y(x_i|a))}{\sigma_i^2} \frac{\partial (y(x_i|a))}{\partial a_k}$$

$$\alpha_{kl} = \frac{1}{2}\frac{\partial^2 \chi^2}{\partial a_k \partial a_l} = \sum_{i=0}^{N-1} \left[\frac{1}{\sigma_i^2} \frac{\partial y(x_i|a)}{\partial a_k} \frac{\partial y(x_i|a)}{\partial a_l} - (y_i - y(x_i|a)) \frac{\partial^2 y(x_i|a)}{\partial a_k \partial a_l} \right]$$

The matrix α_{kl} is known as the *curvature matrix*. In most cases, it is advisable to drop the second derivative term in this matrix (see NR 15.5.1 for a partial explanation). The standard approach uses:

$$\alpha_{kl} = \sum_{i=0}^{N-1} \left[\frac{1}{\sigma_i^2} \frac{\partial y(x_i|a)}{\partial a_k} \frac{\partial y(x_i|a)}{\partial a_l} \right]$$

To implement this, you choose a starting guess of parameters, solve for the corrections δa_l to be applied to a current set of parameters, and iterates until one arrives at a solution that does not change significantly.

Far from the solution, this method can be significantly off in providing a good correction, and, in fact, can even move parameters away from the correct solution. In this case, it may be advisable to simply head in the the steepest downhill direction in χ^2 space, which is known as the method of steepest descent. Note that while this sounds like a reasonable thing to do in all cases, it can be very inefficient in finding the final solution (see, e.g. NR Figure 10.8.1).

A common algorithm switches between the method of steepest descent and the parabolic approximation and is known as the Levenberg-Marquardt method. This is done by using a modified curvature matrix:

$$\sum_{i=0}^{M-1} \alpha'_{kl} \delta a_1 = \beta_k$$

where

$$\alpha'_{jj} = \alpha_{jj}(1+\lambda)$$
$$\alpha'_{jk} = \alpha_{jk} \text{ (for } j \neq k)$$

When λ is large, this gives a *steepest descent* method. When λ is small, this gives a *parabolic* method. This leads to the following recipe:

1. Choose a starting guess of parameter vector (a) and calculate $\chi^2(a)$.
2. Calculate the correction using model λ, e.g. $\lambda = 0.001$ and evaluate χ^2 at the new point.
3. If $\chi^2(a + \delta a) > \chi^2(a)$, increase λ and try again.
4. If $\chi^2(a + \delta a) < \chi^2(a)$, decrease λ, update a, and start the next step.
5. Stop when the convergence criteria are met.
6. Invert the curvature matrix to get parameter uncertainties.

Two key issues in nonlinear least squares is finding a good starting guess and a good convergence criterion. You may want to consider multiple starting guesses to verify that you're not converging to a local minimum. For convergence, you can look at the amplitude of the change in parameters or the amplitude in the change of χ^2.

This method is implemented in `astropy` in the modeling module using the LevMarLSQFitter which is used identically to the linear least squares fitter described above.

There are certainly other approaches to non-linear least squares, but this provides an introduction.

8.4.1 A nonlinear fitter without derivatives

A common minimization method that does not use derivatives is the "downhill simple" or *Nelder-Mead* algorithm. For this technique, χ^2 is initially evaluated at $M + 1$ points (where M is the number of parameters). This makes an M-dimensional figure, called a *simplex*. The largest value is found, and a trial evalaution is made at a value reflected through the volume of the simplex. If this is smaller than the next-highest value of the original simplex, a value twice the distance is chosen and tested; if not, then a value half the distance is tested, until a better value is found. This point replaces the initial point, and the simplex has moved a step; if a better value can't be found, the entire simplex is contracted around the best point, and the process tries again. The process in then repeated until a convergence criterion is met. See NR Figure 10.5.1 and section 10.5. The simplex works its way down the χ^2 surface, expanding when it can take larger steps, and contracting when it needs to take smaller steps. Because of this behavior, it is also known as the "amoeba" algorithm.

Python/scipy implementation of Nelder-Mead

Python/scipy summary of optimization routines in general

Understand the concepts of how a nonlinear least squares fit is performed, and the issues with global vs local minima, starting guesses, and convergence criteria. Know how to computationally imiplement a nonlinear least squares solution.

8.5 Parameter uncertainties and confidence limits

Fitting provides us with a set of best-fit parameters, but, because of uncertainties and limited number of data points, these will not necessarily be the true parameters. One generally wants to understand how different the derived parameters might be from the true ones.

The covariance matrix gives information about the uncertainties on the derived parameters. In the case of well-understood uncertainties that are strictly distributed according to a Gaussian, these can be used to provide *confidence levels* on your parameters; see NR 15.6.5.

However, if the uncertainties are not well-understood, numerical techniques can be used to derive parameter uncertainties.

A straightforward technique if you have a good understanding of your *measurement* uncertainties is the *Monte Carlo simulation*. In this case, you simulate your data set multiple times, derive parameters for each simulated data set, and look at the range of fit parameters as compared with the input parameters. To be completely representative of your uncertainties, you would need to draw the simulated data set from the true distribution, but you don't know what that is (it's what you're trying to derive). So we take the best-fit from the actual data as representative of the true distribution, and hope that the *difference* between the derived parameters from the simulated sets and the input parameters is representative of the difference between the actual data set and the true parameters.

If you don't have a solid understanding of your *data* uncertainties, then *Monte Carlo* will not give an accurate representation of your parameter uncertainties. In this case, you can use multiple samples of your own data to get some estimate of the parameter uncertainties. A common technique is the *bootstrap* technique, where, if you have N data points, you make multiple simulations using the same data, drawing N data points from the original set **with replacement** (i.e. the same data point can be drawn more than once), derive parameters from multiple simulations, and look at the distribution of these parameters.

However, you may need to be careful about the interpretation of the confidence intervals determined by any of these techniques, which are based on a frequentist interpretation of the data. For these calculations, note that confidence intervals will change for different data sets. The frequentist interpretation is that the true value of the parameter will fall within the confidence levels at the frequency specified by your confidence interval. If you happen to have taken an unusual (*in*frequent) data set, the true parameters may not fall within the confidence levels derived from this data set.

Understand how uncertainties can be estimated from least-squares fitting via the covariance matrix, Monte Carlo simulation, and the bootstrap method, and how these techniques work

.

8.6 Bayesian analysis

Review material above, also see astroML, NR and http://arxiv.org/abs/1411.5018

As discussed above and in additional references, Bayesian analysis has several significant differences from the frequentist/likelihood approach. First, Bayesian analysis calculates the probability of model parameters given data, rather than the probability of data given a model.

$$P(M|D) = \frac{P(D|M)P(M)}{P(D)}$$

Practically, this allows for the possibility of specifying an explicit prior on the model parameters. The Bayesian analysis may give the same result as the frequentist analysis for some choices of the prior, but it makes the prior explicit, rather than hidden.

Second, the Bayesian analysis produces a joint probability distribution function of the model parameters. The frequentist analysis produces a most probable set of parameters and a set of confidence intervals.

The Bayesian analysis is computationally more challenging because you actually have to compute the probability for different model parameters. This is essentially performing a grid search of the full parameter space, which can be computationally intractable for large numbers of parameters. Fortunately, computational techniques have been developed to search through the full parameter space concentrating only on the regions with non-negligible probability; as discussed below, Markov Chain Monte Carlo is foremost among such techniques.

8.6.1 The prior

Priors in Bayesian analysis reflect external knowledge about parameters that exist independent of the data being analyzed. If there is no external knowledge, then you want to use a *noninformative* prior. Often, this is just a statement that all parameter values are equally likely. In many cases, this makes a Bayesian analysis equivalent to a frequentist analysis. However, beware that model parameters could be expressed with a transformation of variables, and a flat distribution in one variable may not be a flat distribution under a variable transformation, and could lead to different results. Also, a prior that appears to be unbiased may not be so, e.g., consider the fitting parameters for a straight line, in particular the slope.

8.6.2 Marginalization

For many problems, only some of the parameters are of interest, while others fall into the category of so-called *nuisance* parameters, which may be important for specifying an accurate model (which is fundamentally important), even though they may not be of interest for the scientific question you are trying to answer. In such cases, Bayesian analysis uses the concept of *marginalization*, where one integrates over the dimensions of the nuisance parameters to provide probability distribution functions of only the parameter(s) of interest.

Marginalization is also used to give probability distribution functions of individual parameters, i.e. without regard to the values of other parameters. However, one needs to be aware that parameter values may be correlated with each other, and marginalization hides such correlations.

Marginalization can also be used to derive the probability distribution of some desired quantity given measurements of other quantities if there is some relation between the measured and desired quantity. An example is in determining ages of stars given a set of observations, e.g. of spectroscopic parameters and/or apparent magnitudes. Here, stellar evolution gives predictions for observed quantities as a function of age, mass, and metallicity; in conjunction with an initial mass function, you get predictions for number of stars for each set of parameters. Given some priors on age, mass, and/or metallicity, you could compute the probability distribution of a given quantity by marginalizing over all other parameters. Observational constraints would modify this probability distribution.

Understand what marginalization means, and why and when it can be used

.

8.6.3 Markov Chain Monte Carlo (MCMC)

Calculating marginal probability distributions is basically a big integration problem. If the problem has many parameters, the multi-dimensional integral can be very intensive to calculate. One technique for multi-dimensional integration is **Monte Carlo** integration. Choose a (large) number of points at random within some specified volume (limits in multiple dimensions), sample the value of your function at these points and estimate the integral as

$$\int f \mathrm{d}V = V < f >$$

where

$$< f >= \frac{1}{N} \sum f(x_i)$$

We could use this to calculate marginal probability distribution functions, but it is likely to be very inefficient if the probability is small over most of the volume being integrated over. Also, for most Bayesian problems, we do not have a proper probability density function because of an unknown normalizing constant; all we have is the relative probability at different points in parameter space.

To overcome these problems, we would instead like to place points in a volume proportional to the probability distribution at that point; we can then calculate the integral by summing up the number of points. This is achieved by a **Markov Chain Monte Carlo** analysis. Here, unlike Monte Carlo, the points we choose are not statistically independent, but are chosen such that they sample the (unnormalized) probability distribution function in proportion to its value. This is acheived by setting up a Markov Chain, a process where the value of a sampled point depends only on the value of the previous point. To get the Markov Chain to sample the (unnormalized) PDF, the transistion probability has to satisfy:

$$\pi(x_1)p(x_2|x_1) = \pi(x_2)p(x_1|x_2)$$

where p is a transition probability to go from one point to another.

To do the integral, sum the number of points in the chain at the different parameter values.

There are multiple schemes to generate such a transition probability. A common one is called the *Metropolis-Hastings* algorithm: from a given point, generate a candidate step from some proposal distribution, $q(x2|x1)$, that is broad enough to move around in the probability distribution $q(x2|x1)$ in steps that are not too large or too small. If the candidate step falls at a larger value of the probability function, accept it, and start again. If it falls at a smaller value, accept it with probability proportional to

$$\pi(x_c)q(x_1|x_{2c})\pi(x_1)q(x_{2c}|x_1)$$

For a symmetric Gaussian proposal definition, note that the q values cancel out.

Note also that you need to choose a starting point. If this is far from the maximum of the probability distribution, it may take some time to get to the equilibrium distribution. This leads to the requirement of a *burn-in* period for MCMC algorithms.

In practice, you have to be careful about choosing an appropriate proposal distribution, and an appropriate burn-in period.

Implementation: see emcee and pymc, among others. Also this.

Note that `emcee` uses an algorithm that is different from Metropolis-Hastings, that is designed to function more efficiently in a case where the PDF may be very narrow, leading to very low acceptance probabilities with M-H. Partly this is accomplished by using a number of *walkers*, i.e. multiple different Markov chains, but these are not totally independent: knowledge from other walkers are used (somehow) to help explore the parameter space more efficiently. Note that since the walkers are not independent, the chain from a single walker does not sample the PDF proportionally; the combined chain from all of the walkers is what needs to be used.

Understand what Markov Chain Monte Carlo analysis accomplishes, and at least qualitatively how it works. Be able to implement an MCMC analysis of a problem.