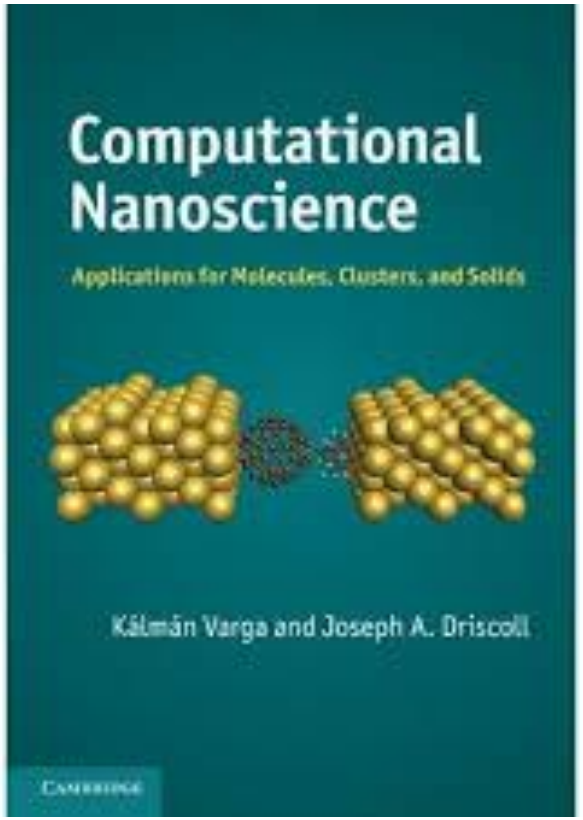
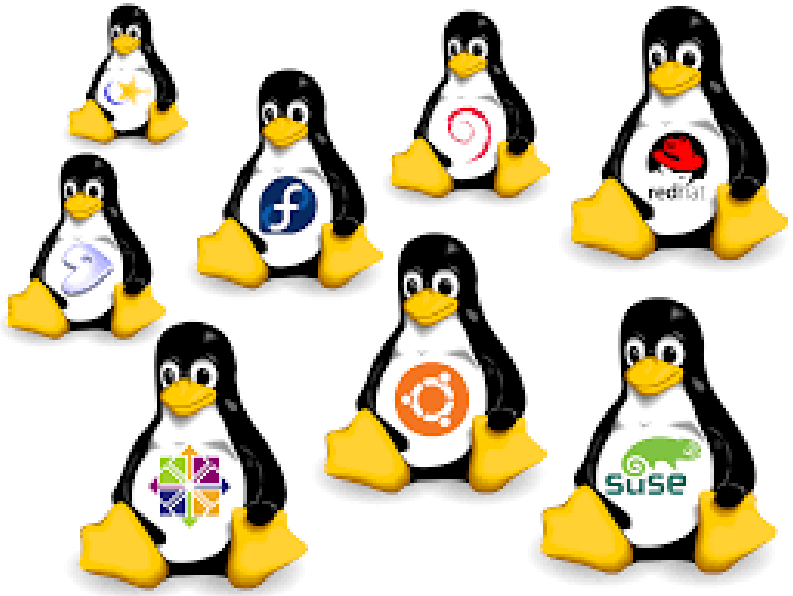
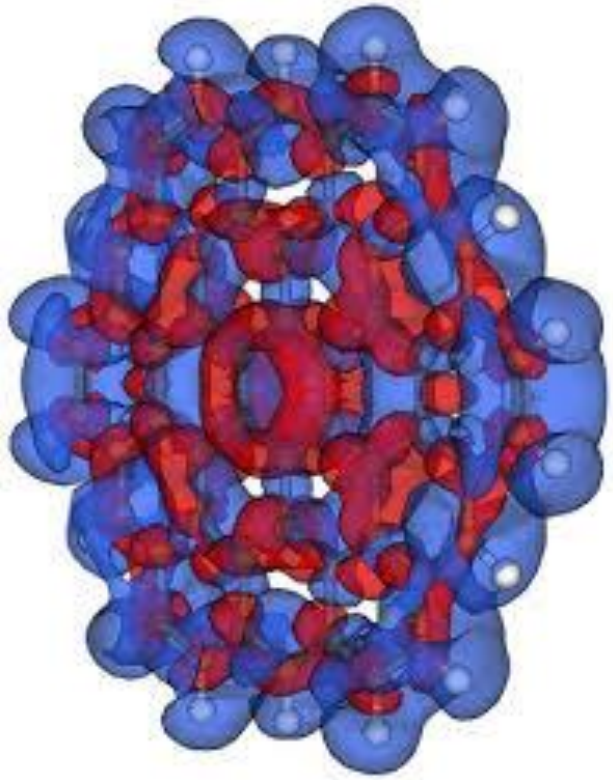
Introduction to Linux and Scientific Computing with the Varga Group



Cody Covington





Linux

Linux is an operating system, like Mac OS or Windows

- Store files, run programs
- Can run from a graphical user interface or from a text window (called a terminal)
- Need username and password to log in
- Linux is case sensitive
- Files organized into a directory structure

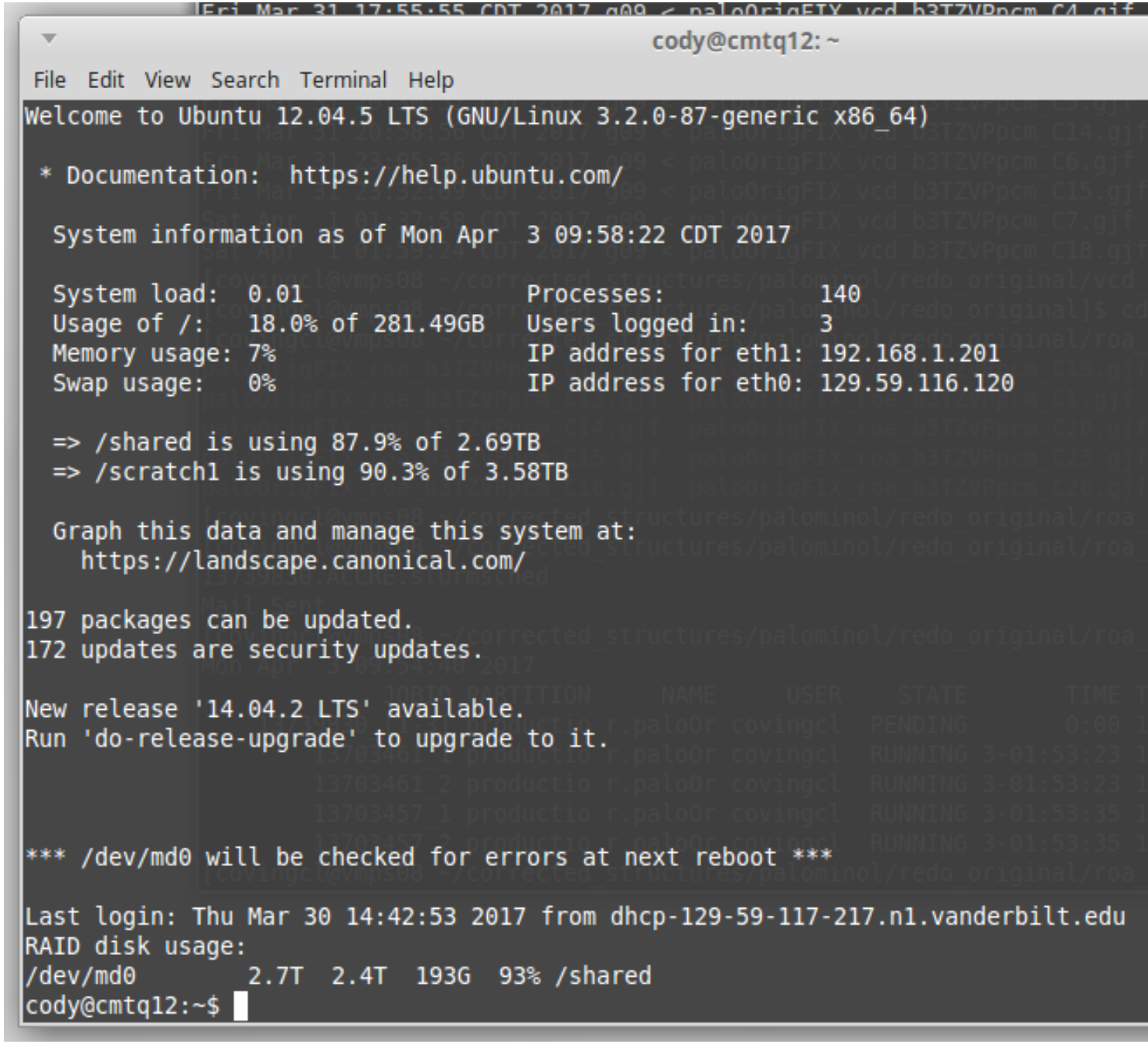
Why Linux?

- Command line interface (CLI) is more powerful and flexible than a graphical user interface (GUI)
- Remote access is much faster
- Many (all?) computing clusters use Linux
- More compiler options.
- Many other reasons…

Using the Terminal

Shell is a user program or it's environment provided for user interaction. The default shell for GNU OS is **BASH**.

- Type commands
  - Press enter to execute
- To cancel press Ctrl+C
- To exit type "**exit**"
- Press the up/down arrows to review previous commands
- **Pasting things into the terminal may not work because of special characters that may not be seen!**



First SSH into the server Kalman-all-series1

- Try these commands on your laptop, or ssh into one of our linux computers.

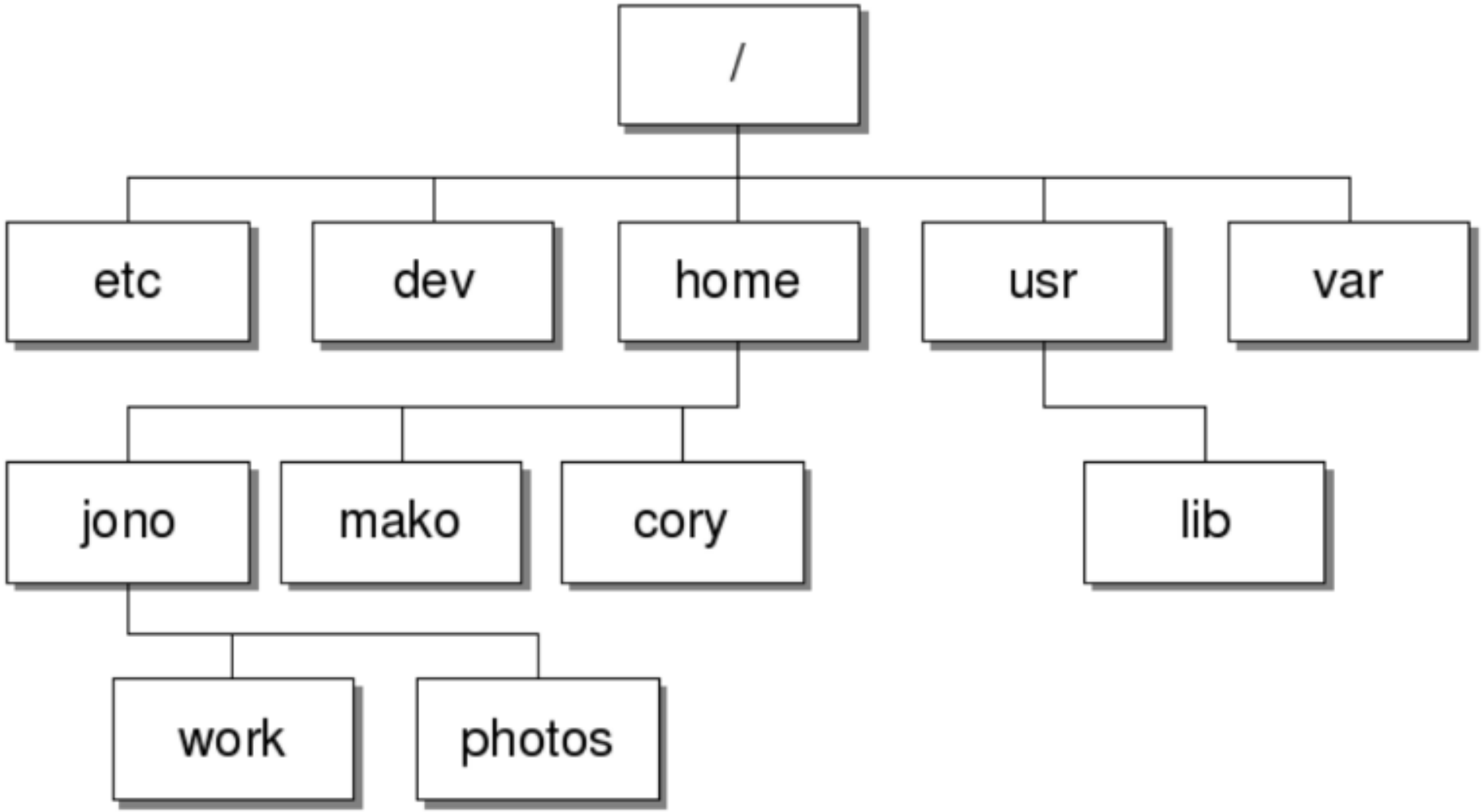
- Everybody: ssh to your “your name”19@user@129.59.116.196

  Example: **ssh rowan19@**129.59.116.196

  - Change your temporary password to something secure using the command: **passwd**

- See who else is there with the “**w**” command.
- See processor usage with the “**top**” command. (use q or ctrl+C to quit)

Linux Directory Structure



Some Intro

- Current location in the directory structure is “working directory”
- Directories can contain files and other directories
  - **ls** lists contents of working directory (more info use **ls –l**)
  - **pwd** shows working directory, **cd** changes it
- Make/remove a directory: **mkdir**, **rmdir** (or **rm -r** if not empty)
- Remove a file: **rm**
- No undelete command
- Paths,
  - “./” in this directory
  - “.” can mean several things
  - “..” previous directory
- Use the **?** as a wildcard for any 1 character and * for any string

Permissions in Linux

- Permissions for owner, group, and everyone else.
- View file permissions with **ls –l**
- Change permissions with **chmod 744 file.dat**
  - r=read
  - w=write
  - x=execute

#	Permission	rwx
7	read, write and execute	rwx
6	read and write	rw-
5	read and execute	r-x
4	read only	r--
3	write and execute	-wx
2	write only	-w-
1	execute only	--x
0	none	---

Some Commands

Command	Effect
exit	logout
ls	List directory contents
cd <directory>	Changes working directory
cat <filename>	Dump file contents to screen
man <command>	Get help for a command
pwd	Show present working directory
nano	Start the nano text editor
df -h	Show disk usage
top	Show CPU/memory usage (q to quit)
rm <name>	Remove file

mv <name_in> <name_out> move or rename file

history print history

du –h disk usage within working directory

Remote Access: SSH

To login to one machine from another, use ssh:

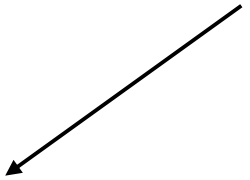
ssh user@computer.net

If you want to use something graphical, use –X or –Y (– Y is better)

ssh -Y user@computer.net

Copy a file from local directory to another computer:

scp myfile.txt user@compter.net:.



Period used here:
Keep file name
the same

Copy a file from another computer to local directory:

scp user@compter.net:file2.txt .

More Advanced Command Line Usage (BASH)

- Regex: grep, sed, awk
- Standard Streams: stdin, stdout, and stderr
- Pipes
- Variables
- Scripting
- Using the cluster
  - Torque
- Environment Configuration

Warning about special characters

- Expressions with variables and special characters require special treatment.
- Special characters
  - # indicates a comment
  - ; separates commands
  - Space is a kind of special character
  - Others (see http://docstore.mik.ua/orelly/unix/upt/ch08_19.htm)

    . , / \ $ ~ ^ & * ( ) [ ] { } ‘ “ : ? | !
    - To use some of these characters, the escape character \ is needed
    - Ex: **echo “\$”**
  - File names should not contain spaces! Use the underscore ‘_’ or dash ‘-’

Regex: grep, sed, awk

A **regular expression**, **regex** or regexp is a sequence of characters that define a search pattern.

- grep – Use to search files for string expressions
  - Ex find lines that contain the string banana: **grep -in "banana" file.dat**
  - Useful options –i –n –A –B –F –w –v –x –f
    - v = reverse A = after B = before i = case insensitive n = with line number
- sed – Use to alter contents of files
  - Ex finds a string and replaces with new string:
  - **sed 's/BANANA/banana/g' file.dat**
  - Use –i option to alter files in place
  - Ex Delete line 1: **sed '1d' file.dat**
  - Ex print line 1: **sed '1!d' file.dat**

Regex: grep, sed, awk

- awk – Can process/manipulate contents of a file
  - Ex print columns 1 & 3: **awk '{print $1,'\t', $3}' file.dat**
  - Ex multiply column 1 by a constant: **awk ‘{print $1*5.5}’ file.dat**
  - Ex multiply column 1 by a constant (variable): **awk –v a=5.5 ‘{print $1*a}’ file.dat**
  - Ex print file with line numbers: **awk '{print NR" " $0}' file.dat**
  - Logical constructs are possible
  - Ex print column 1 if larger than 10: **awk ‘{ if($1>10.0) print $1}’ file.dat**
  - Ex Changing the Field Separator to comma: **awk –F, ‘{print $2}’ file.dat**
  - Can also find strings like grep, but grep is better for that
    - **awk '/EX/ { print $0 }' file.dat**

Other Useful Utilities

- paste – paste 2 files together column wise
- cut -- remove sections from each line of files
- diff – compare 2 files
- tar – create an archive
- sort – sort file by text alphabetically or numerically
- seq – print a sequence of numbers

Standard Streams: stdin, stdout, and stderr

- **Standard input** (stdin) - this is the *file handle* that your process reads to get information from you.
- **Standard output** (stdout) - your process writes normal information to this file handle.
- **Standard error** (stderr) - your process writes error information here.

-- redirect operator < > |

::examples

my_prog <input_file >output_file 2>error_file (redirect stdout and stderr to files)

my_prog <input_file > output_file 2>&1 (redirect stderr to stdout (&1), and then redirect stdout to a file)

my_prog <input_file &> output_file (Redirect both to a file)

my_prog <input_file >> outfile (Redirect to a file, but append to what is already there)

my_prog <input_file > /dev/null (discard any output to stdout)

Pipes | *the same key as *

A **pipe** is a form of redirection that is used in **Linux** and other Unix-like operating systems to send the output of one program to another program for further processing.

Example:

ls

ls | wc

ls | head –n3

ls | head –n3 | tail –n1

grep “banana” file.dat | awk ‘{print “total cost=”$1*3.99 }’

grep “banana” file.dat | awk –F, ‘{print $2}’ | awk ‘{print $1}’

File name goes on the first execution of the program

No file name at the end

Variables

- Defining variables is easy. No type declaration (string by default)
  - Ex: **i=1**
  - Ex: **fn=myfile.dat**
  - Ex: **list="thing1 thing2 thing3"**
- Using variables is done by prepending the variable name with $
  - Ex: echo $fn
  - Ex multiple variables: echo fni
  - Ex mixed variables and text: echo $fn"_"$i
  - Storing stdout in a variable: **filelist=$(ls *.dat)**
- Some variables are used by the shell, so don't use them! They will typically be in all caps. Use **printenv** to see them all
  - Ex: HOME, PATH, TERM,

If then, elif, and else in BASH

- Use comparison/file test operator.
  - For a full list see: http://tldp.org/LDP/abs/html/fto.html and http://tldp.org/LDP/abs/html/comparison-ops.html
- Example:

```
if [[ “$i” == 1 ]]; then
    echo $i equals 1
elif [[ “$i” == 2 ]]; then
    echo $i equals 2
else
   echo $i is not equal to 1 or 2
fi
```

Double bracket and single bracket are different!

integer comparison

-eq is equal to
if ["$a" -eq "$b"]
-ne is not equal to
if ["$a" -ne "$b"]
-gt is greater than
if ["$a" -gt "$b"]
-ge is greater than or equal to
if ["$a" -ge "$b"]
-lt is less than
if ["$a" -lt "$b"]
-le is less than or equal to
if ["$a" -le "$b"]

Integer math, for, and while constructs in BASH

- Integer math: let i=i+1
  - (cannot do floating point math. Use awk for that)
- For loop examples:

for i in 2 4 6 ; do echo $i ; done

for i in a b c ; do echo $i ; done

Note use of semicolon to put multiple commands on the same line

- Ex:

```
j=0; k=0
for i in {1..100}; do
let f=j+k
if [[ $f == 0 ]]; then f=1 ; fi
k=$j; j=$f
echo $f
done
```

What do you get?

Integer math, for, and **while** constructs in BASH

- While construct very useful to read from a file
- Ex read line by line from file to variable named '**getline'**:

```
while read getline; do
  echo I just read line $getline from a file, and I liked it
done < file.dat
```

Scripting in BASH

- Use scripts (files with sets of commands) to do repetitive tasks or submit jobs
- Use **chmod** to make the script executable
- Start file with line to indicate the interpreter for the script:

#!/bin/bash

 # comment lines start with #

echo hello world

exit 0

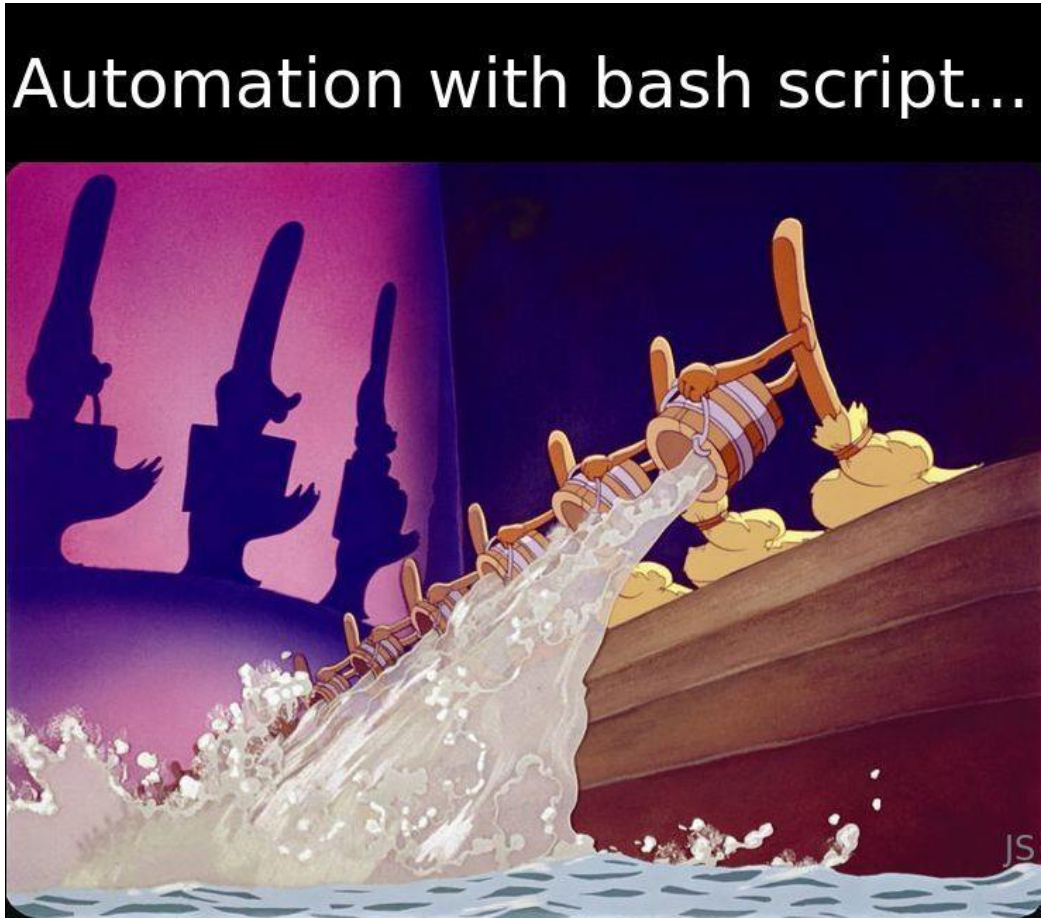
Scripting in BASH (cont)

- The varables $1, $2, $3, … are set as the n^{th} argument of that was used to call the script. Example: myscript.bash

```
#!/bin/bash
arg1=$1
arg2=$2
grep $arg2 $arg1 | awk ‘{print $2}’
grep -v $arg2 $arg1 | awk ‘{print $2}’
exit 0
```

Execute with: **./myscript.bash argment1 argument2**

Be warned about bash scripting



- Test out that your script will do what it is supposed to do
- Ex a small mistake with big consequences:

fn=“file1”

echo rm /$file1

#rm /$file1

Terminal Tip: Tab Completion

- A partially entered file name can be completed using the Tab key
  - There must be only one unique possibility for the command

Ex you have only file1.dat and file2.dat in the working dr

You type “**cat fi**” and then press the Tab key before you press enter.

The command should fill in to “**cat file**” but cannot finish because there are 2 files that could match this string.

Type “**cat file1**” and press Tab. You should see the command completed to “**cat file1.dat**”, the command should not execute until you press enter.

- Arrange file names so as to take advantage of command completion.

Environment Configuration (BASH)

- BASH uses several files to initialize the environment (dot at beginning indicates a hidden file).
  - **.bashrc** .profile .bash_profile .bash_aliases .bash_login
  - Can use to define aliases, variables and functions that are used frequently
  - Ex: **IP_4_computer=123.13.98.42**

- Aliases Ex:

alias ls='ls –l'

alias clipin='xclip -in -selection clipboard'

Function Ex:

```
catwc(){ # print file w/ comma separated lines
awk '{ for(s=1;s<=NF;s++) printf $s"," };{print ""}' $1
}
```

Better History Function

- Put these lines in your .bashrc file

```
export uuid=$(uuidgen)
export HISTFILE=$HOME/.history_$uuid
export HISTTIMEFORMAT="%h/%d - %H:%M:%S "
PROMPT_COMMAND="T_DATE=\$(date +%Ym%md%d-%H:%M:%S);history -a; T_HIST=\$(tail -n 1 $HISTFILE); echo \$T_DATE \$T_HIST >> .hist; echo \$T_DATE \$T_HIST >> $HOME/.full_hist;“
```

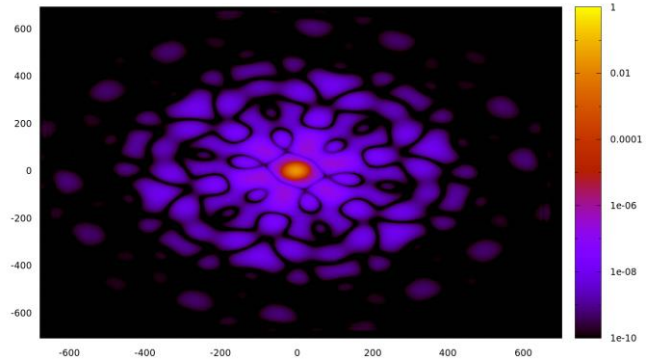
- Keeps track of commands executed in the directory they were executed in (it is not perfect because of execution order of PROMPT_COMMAND variable)
- You will have to occasionally clean up the extra history files that will build up in your home directory

Useful Software

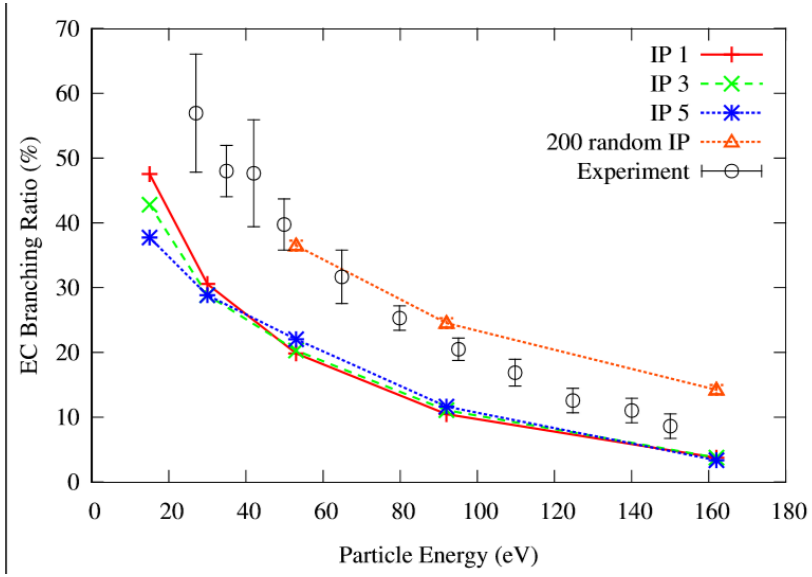
- 2D plotting: gnuplot, xmgrace
- 3D plotting: VisIt, gnuplot
- Molecular viewers: Vesta, jmol, pymol, VisIt, Avogadro, (gnuplot)

Gnuplot

it is pronounced as one syllable with a hard g, like “grew” but with the letter “n” instead of “r”.



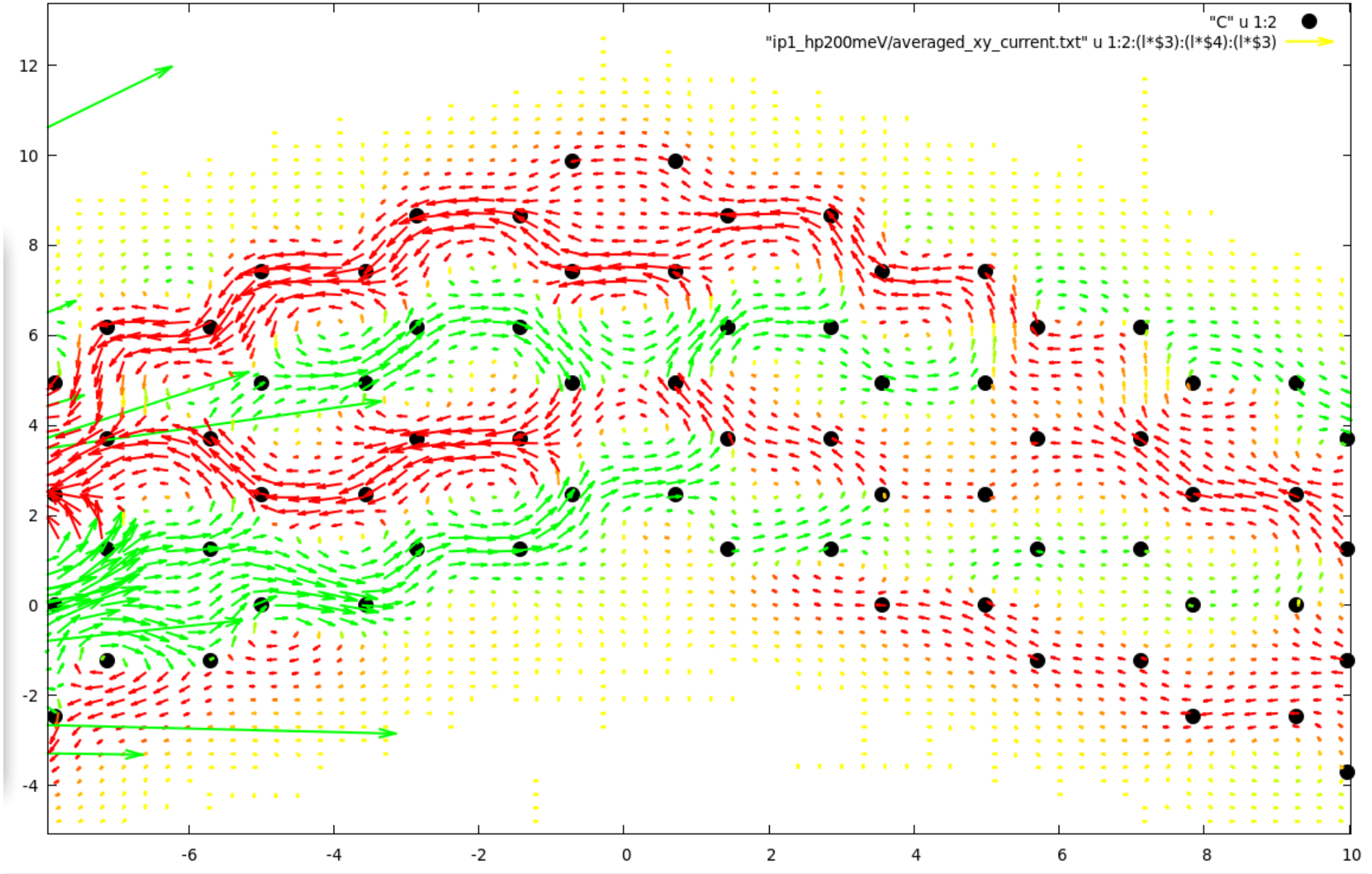
- Very powerful plotting and fitting software.
- Also command line based
- Call with **gnuplot**
  - basic example: **plot sin(x)**
- Can run scripts
- Logical constructs and loops
- Useful bash function:




```
gpdumb(){ #arg 1 is the file, arg 2 is x axis, arg 3 is y axis
echo "set term dumb 140,40; p \"./$1\" u $2:$3 " | gnuplot
}
```

see: http://people.duke.edu/~hpgavin/gnuplot.html

And https://www.cs.hmc.edu/~vrable/gnuplot/using-gnuplot.html

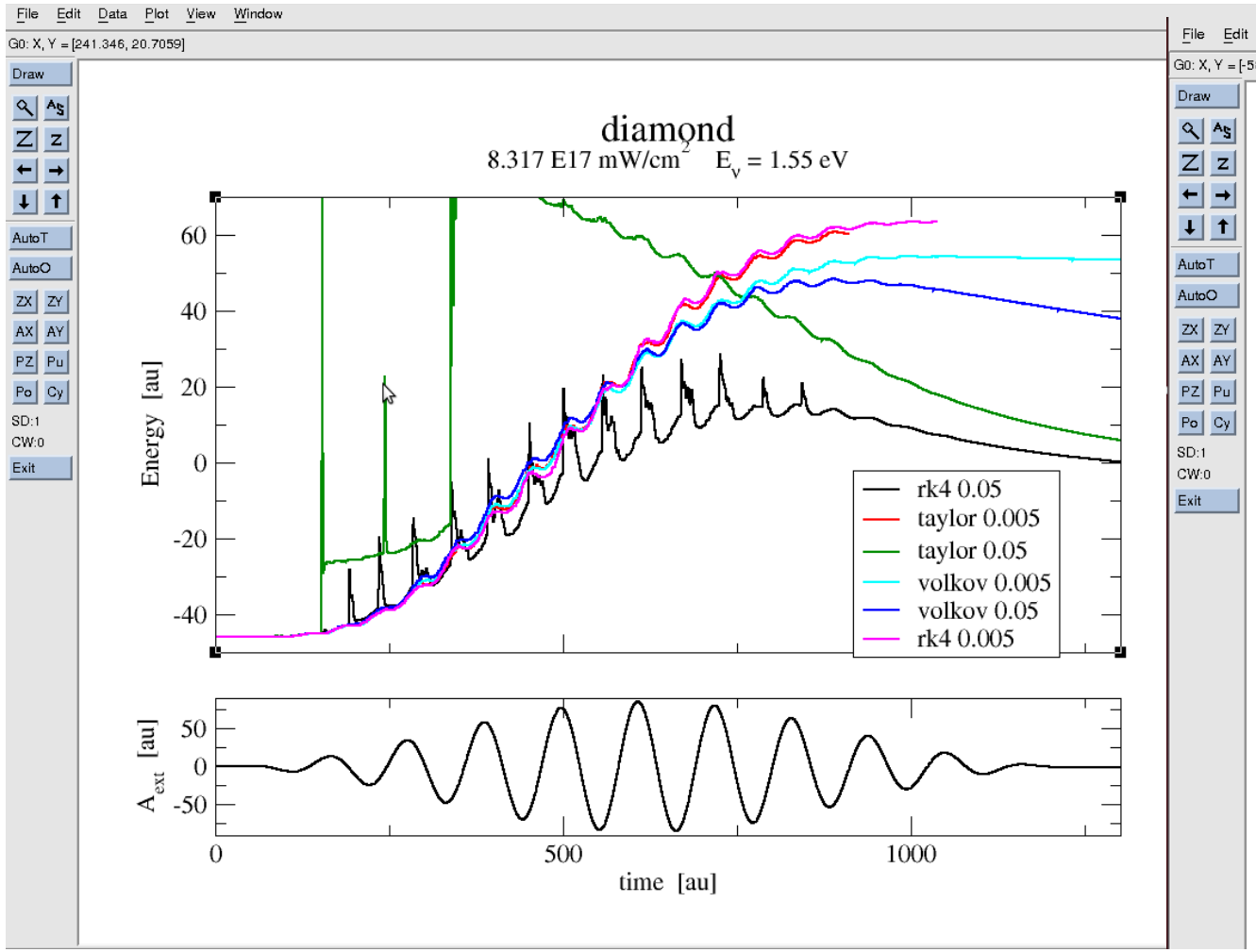


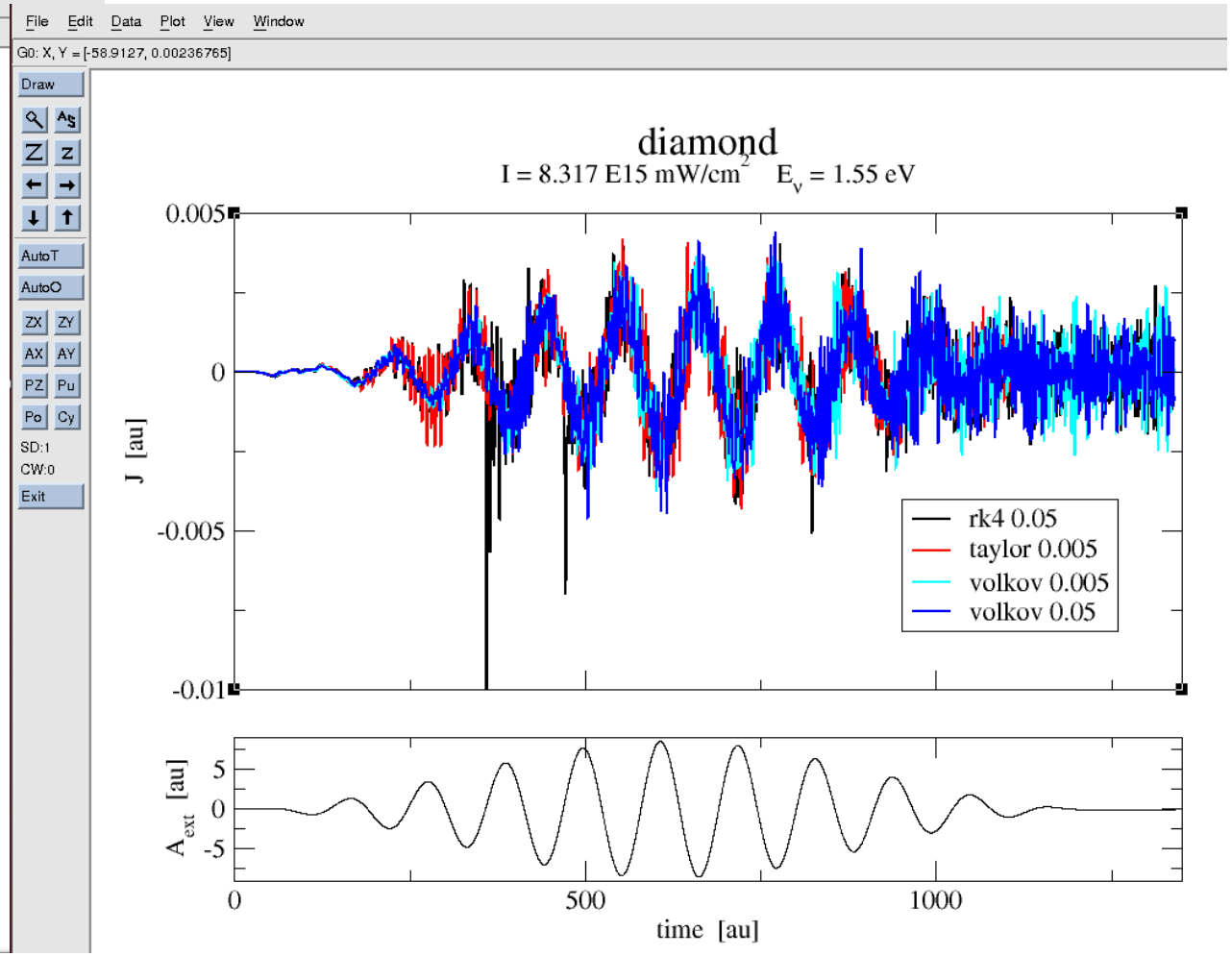
Visualization of current flow in a bent graphene nanoribbon done with Gnuplot

Advanced Gnuplot Usage and Environment

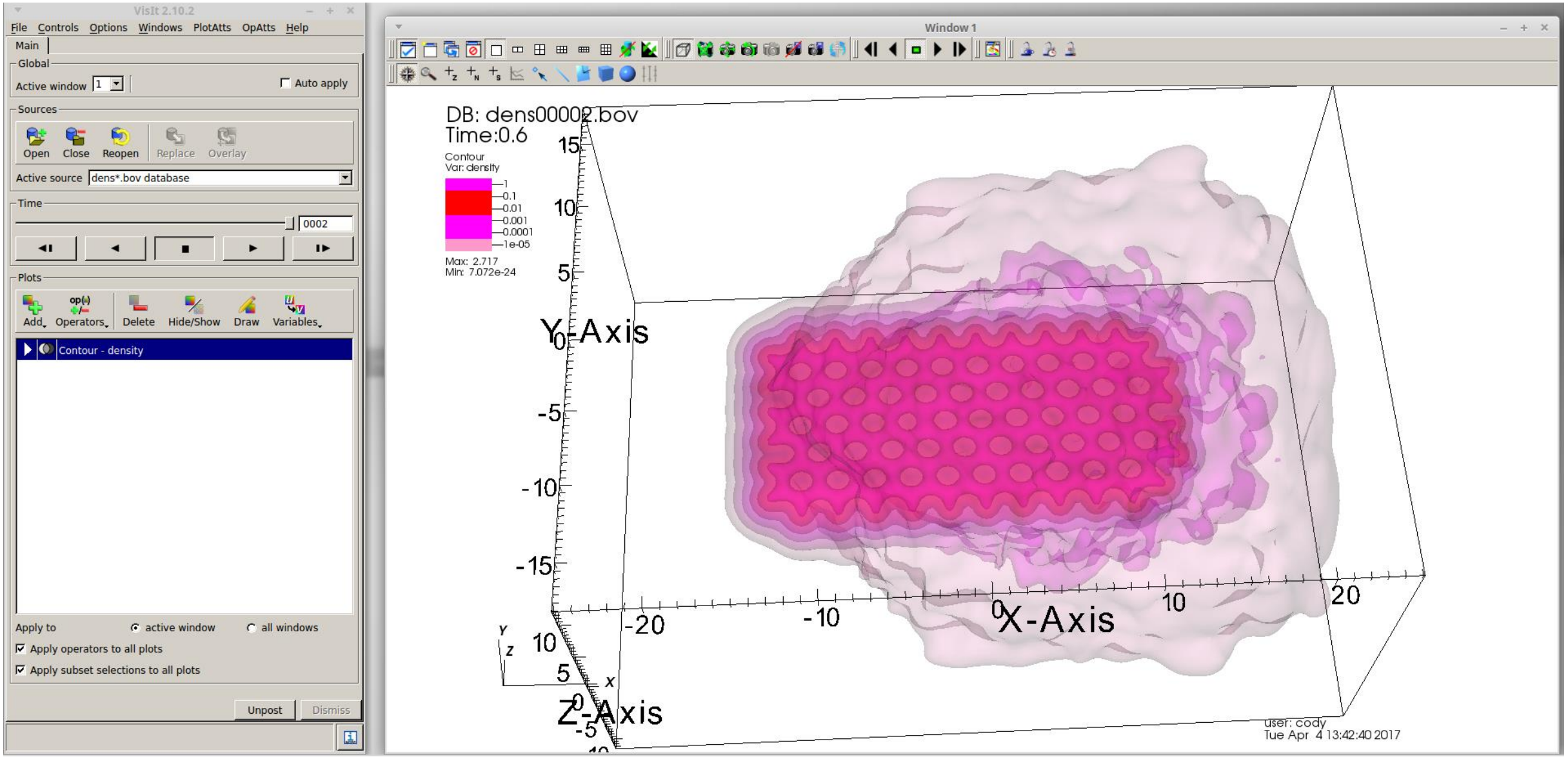
- Data manipulation
  - plot “file1.dat” using ($1*1.23456):($2/$1) with lines title “my title”
- Other usage
  - plot “file1.dat” u 1:2:3 lc palette t “line color given by column 3”
  - plot “file1.dat” u 1:2 t “file1”, “file2.dat” u 1:2 t “file2” # 2 plots together
- Set up environment in $HOME/.gnuplot
- Can define macros
- See: https://sites.google.com/site/kvargagroup/home/group-lectures/computational-nanoscience-2017/gp_plot_output2file.plt?attredirects=0&d=1

XMGRACE


File Edit Data Plot View Window
G0: X, Y = [241.346, 20.7059]
Draw
AutoT
AutoO
ZX ZY
AX AY
PZ Pu
Po Cy
SD:1
CW:0
Exit
diamond
8.317 E17 mW/cm^2 E$_v$ = 1.55 eV
Energy [au]
60
40
20
0
-20
-40
rk4 0.05
taylor 0.005
taylor 0.05
volkov 0.005
volkov 0.05
rk4 0.005
A$_{ext}$ [au]
50
0
-50
0
500
1000
time [au]


File Edit Data Plot View Window
G0: X, Y = [-58.9127, 0.00236765]
Draw
AutoT
AutoO
ZX ZY
AX AY
PZ Pu
Po Cy
SD:1
CW:0
Exit
diamond
I = 8.317 E15 mW/cm^2 E$_v$ = 1.55 eV
J [au]
0.005
0
-0.005
-0.01
rk4 0.05
taylor 0.005
volkov 0.005
volkov 0.05
A$_{ext}$ [au]
5
0
-5
0
500
1000
time [au]

VisIt


VisIt 2.10.2
File Controls Options Windows PlotAtts OpAtts Help
Main
Global
Active window 1
Auto apply
Sources
Open Close Reopen Replace Overlay
Active source dens*.bov database
Time
0002
Plots
Add Operators Delete Hide/Show Draw Variables
Contour - density
Apply to active window all windows
Apply operators to all plots
Apply subset selections to all plots
Unpost Dismiss
Window 1
DB: dens00002.bov
Time:0.6
Contour
Var: density
1
0.1
0.01
0.001
0.0001
1e-05
Max: 2.717
Min: 7.072e-24
Y-Axis
X-Axis
Z-Axis
user: cody
Tue Apr 4 13:42:40 2017

LaTeX

- Like a low-level word processor
  - Can make many kind of documents
  - Control over everything
- Converts plain text and commands from an input file into final document
- Style files
- Latex, dvips, dvipdf
- Bibtex

Fortran

- Fortran is a high-level language, same level as C/C++
- Source code: plain text file with code written in Fortran language
- Compiler/linker: combines source code with libraries to make an executable program

```
PROGRAM hello
write(6,*)”Hello world”
END PROGRAM hello
```

```
gfortran -o hello hello.f90
./hello
```

Fortran: Fibonacci example

```
implicit none
 integer    :: i,j,f,n
 real*8     :: gr

 i=0
 j=0
 gr=0
```

```
do n=1,100
  f=i+j
            if(f.EQ.0.OR.f.LT.0) then
             f=1
            else
             gr=f/dfloat(i) ! don't divide by zero
            end if
            j=i
            i=f
            write(*,'(i7,i12,es24.16e3)') n,f,gr
 end do
            write(*,'(a)') "actually the golden ratio is
1.6180339887498948482045868343656381177203091798057
6286213544862270526046281890244970720720418939113748
8475"
end
```

Fortran: Simple Program

```
implicit none

integer,parameter :: N_Points =100
real*8,parameter  :: dt = 0.25d0
real*8,parameter  :: omega = 0.1d0, Amplitude = 5d0
real*8  :: x, f

open(1,file='output.dat')
do i = 0,N_points-1
  t = i*dt
  f = Amplitude * sin( omega * t )
  write(1,*) t, f
end do
close(1)

end
```

Using Libraries

- Don't reinvent the wheel
- BLAS, LAPACK, FFTW3
- Call library's subroutines from your source code
- Need to tell linker which libraries to use
- Use a Makefile

gfortran -o hello hello.f90 -lblas –llapack

- Some debug options for Intel compiler

ifort -O0 -g -traceback -fpe0 -check -o hello hello.f90

Setting the Intel Environment

- Must set up the environment to use intel compilers
- Source the proper script (depends upon directory that the compilers were installed)
  - The new way (NOTE there is a space between the . and the /)

    . /opt/intel/oneapi/setvars.sh
  - The old way (NOTE there is a space between the . and the /)

    . /opt/intel/composer_xe_2013_sp1.2.144/bin/compilervars.sh intel64

Fortran and Gnuplot exercise
Solving the 1d Schrodinger EQ

- Compile and use the program 1d_solveSE.f90 (located in your home directories on kalman-all-series1 ip=129.59.117.235)
- Use gnuplot to visualize the solutions using different potentials.

- Some tips in gnuplot
  - **fn=system("ls eigen*")**
  - **p for [f in fn] f w l**
  - **p for [i=1:10] word(fn,i) u 1:($2+i*0.5) w l**

See also

https://docs.google.com/document/d/1m2uzWzBbn0GC0Hsckv1rDV_OqM6pHVhIpIyudf_ZT-8/edit (old examples)

https://docs.google.com/document/d/1n_V0iUK2AyP2nVi0aPv6qtvVN4wsscowh6_2NH9SnmA/edit#heading=h.gjdgxs (new examples)