KU LEUVEN

universiteit hasselt
KNOWLEDGE IN ACTION

VLAAMS SUPERCOMPUTER CENTRUM

Vlaanderen
is supercomputing

Linux tools

Mag Selwa, Geert Jan Bex

ICTS Leuven

SCAN ME

25 October 2019

Overview

- Few important issues: quotes, pipes, redirections, output
- Some useful commands
- Grep
- Sed
- Awk
- Gnuplot
- ImageMagick

Objectives

- The material in this lecture is **not a complete** guide or manual.

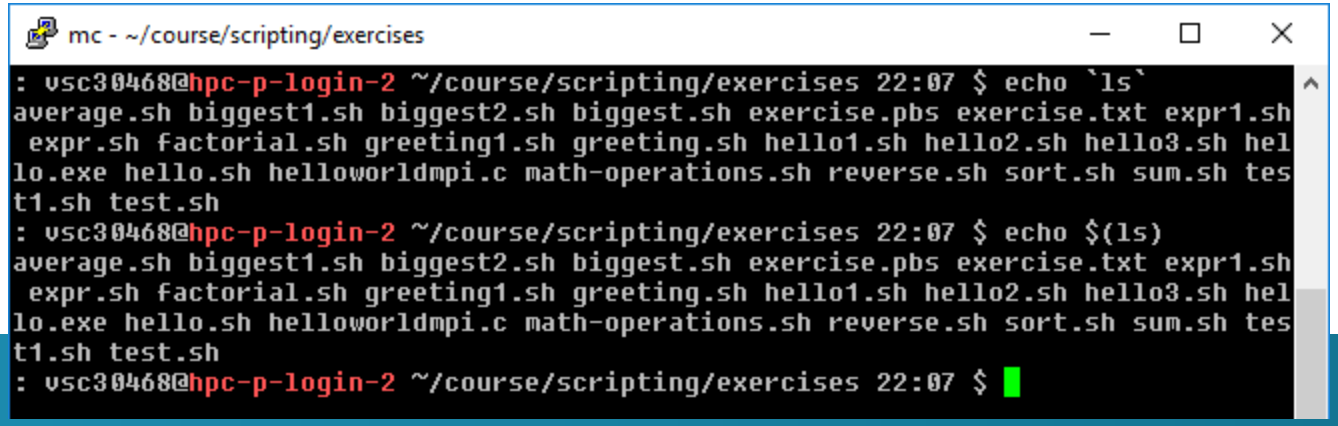
- The purpose is to get overview of the capabilities of the different tools and to underline their particular strengths and disadvantages.

- The course aims to promote the tools for use in your every day research work and urges you find solutions yourself rather than expecting a complete solution.

Difference Between Single, Double, and Backwards Quote

- Single quotes (‘) do not interpret any variables
- Double quotes (“) interpret variables
- Backwards quotes (`) interpret variables and treat them as a program to run and return the results of that program

- These two operators do the same thing. Compare these two lines: `` ` ` `` vs `$()`



<() vs $()

- <() is similar to $() in that the output of the command inside is re-used.
- In this case, though, the output is treated as a file. This file can be used as an argument to commands that take files as an argument.
- Try:

```
$ grep somestring file1 > /tmp/a
$ grep somestring file2 > /tmp/b
$ diff /tmp/a /tmp/b
```

- instead you can write:

```
$ diff <(grep somestring file1) <(grep
somestring file2)
```

Quote characters

There are three different quote characters with different behaviour. These are:

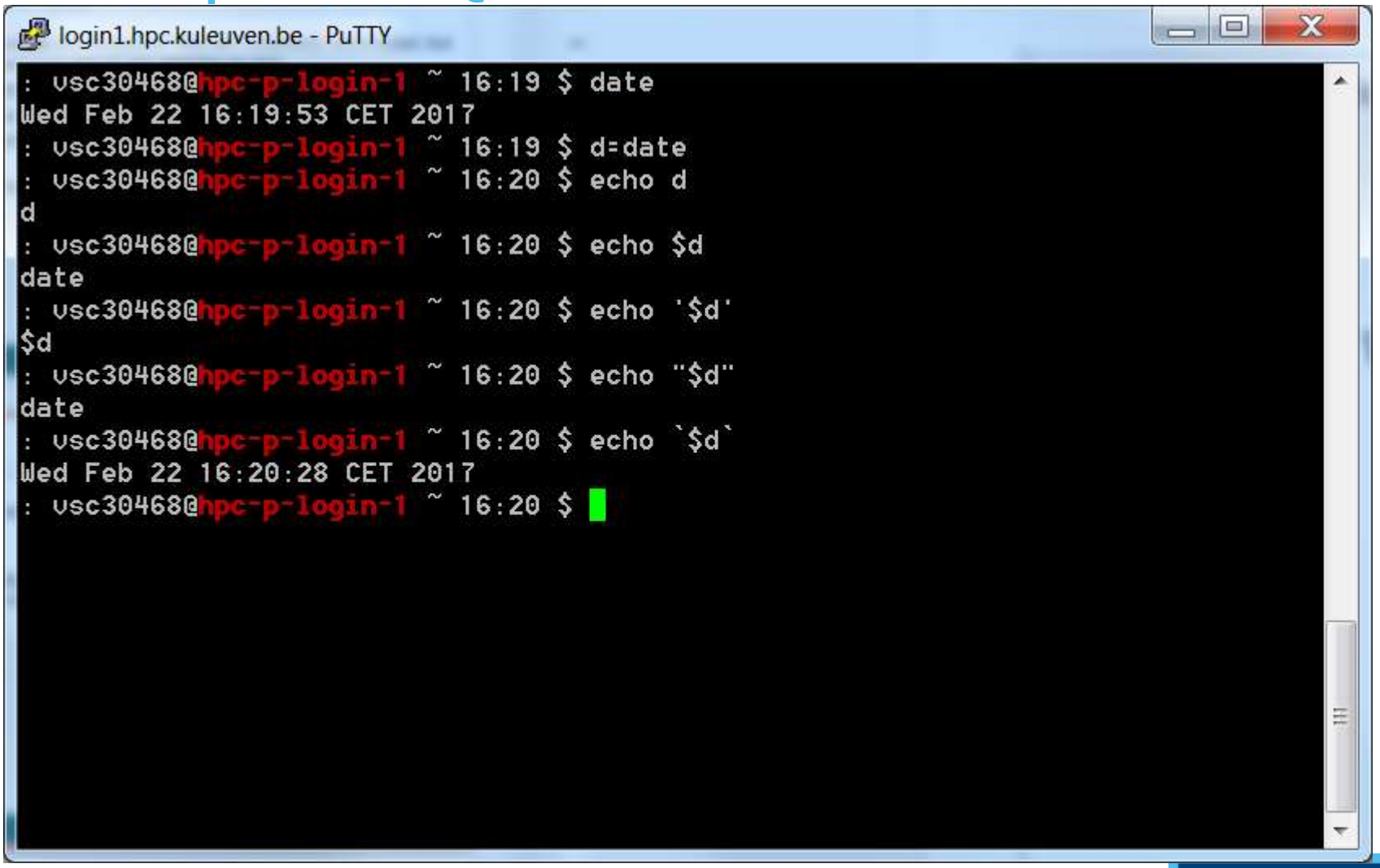
“ : double quote, weak quote. If a string is enclosed in “ ” the references to variables (i.e *$variable*) are replaced by their values. Also back-quote and escape \ characters are treated specially.

‘ : single quote, strong quote. Everything inside single quotes are taken literally, nothing is treated as special.

` : back quote. A string enclosed as such is treated as a command and the shell attempts to execute it. If the execution is successful the primary output from the command replaces the string.

Example: `echo “Today is:” `date``

Example Of Quote Difference



Input and Output

- Programs and commands can contain an input and output. These are called ‘streams’. UNIX programming is oftentimes stream based.

- STDIN – ‘standard input,’ or input from the keyboard
- SDTOUT – ‘standard output,’ or output to the screen
- STDERR – ‘standard error,’ error output which is sent to the screen.

File Redirection

- Often we want to save output (stdout) from a program to a file. This can be done with the 'redirection' operator.
  - **myprogram** > *myfile* – using the '>' operator we redirect the output from **myprogram** to file *myfile*

- Similarly, we can append the output to a file instead of rewriting it with a double '>>'
  - **myprogram** >> *myfile* – using the '>' operator we append the output from **myprogram** to file *myfile*

Input Redirection

- Input can also be given to a command from a file instead of typing it to the screen, which would be impractical.
  - mycommand < *programinput* – Here **mycommand** gets its input from *programinput* instead of waiting for what you type.

Redirecting stderr

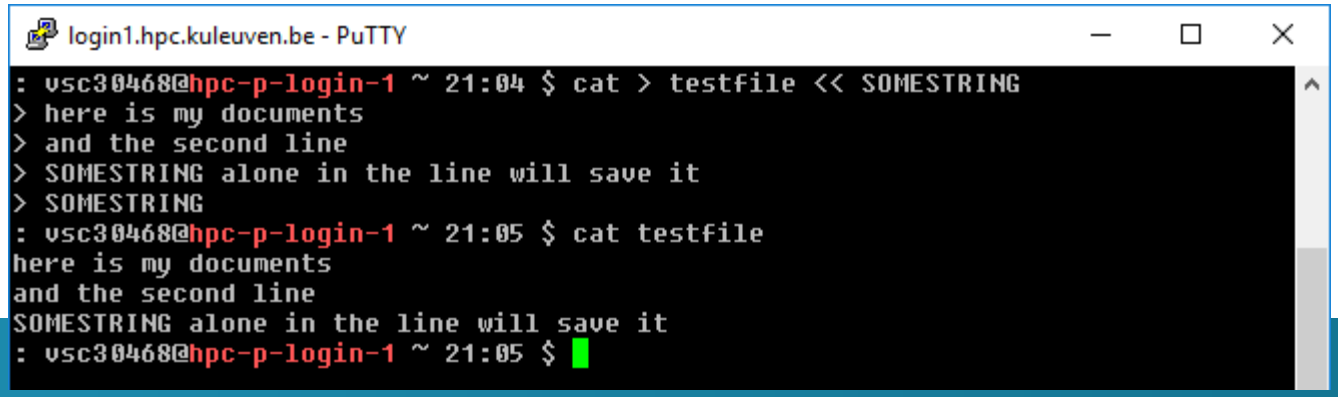
- Performing a normal redirection will not redirect sdterr. In Bash, this can be accomplished with ‘2>’
  - **command** *2> file1*

- Or, one can merge stderr to stdout (most popular) with ‘2>&1’
  - **command** *> file 2>&1*

Redirecting: here docs and here strings

- ‘Here docs’ are files created inline in the shell.
- The ‘trick’ is simple. Define a closing word, and the lines between that word and when it appears alone on a line become a file.
- Notice that:
  - the string could be included in the file if it was not ‘alone’ on the line
  - the string SOMEENDSTRING is more normally END, but that is just convention
- Lesser known is the ‘here string’:

```
$ cat > testfile <<< 'This file has one line'
```



Pipes

- Using a pipe operator ‘|’ commands can be linked together. The pipe will link the standard output from one command to the standard input of another.

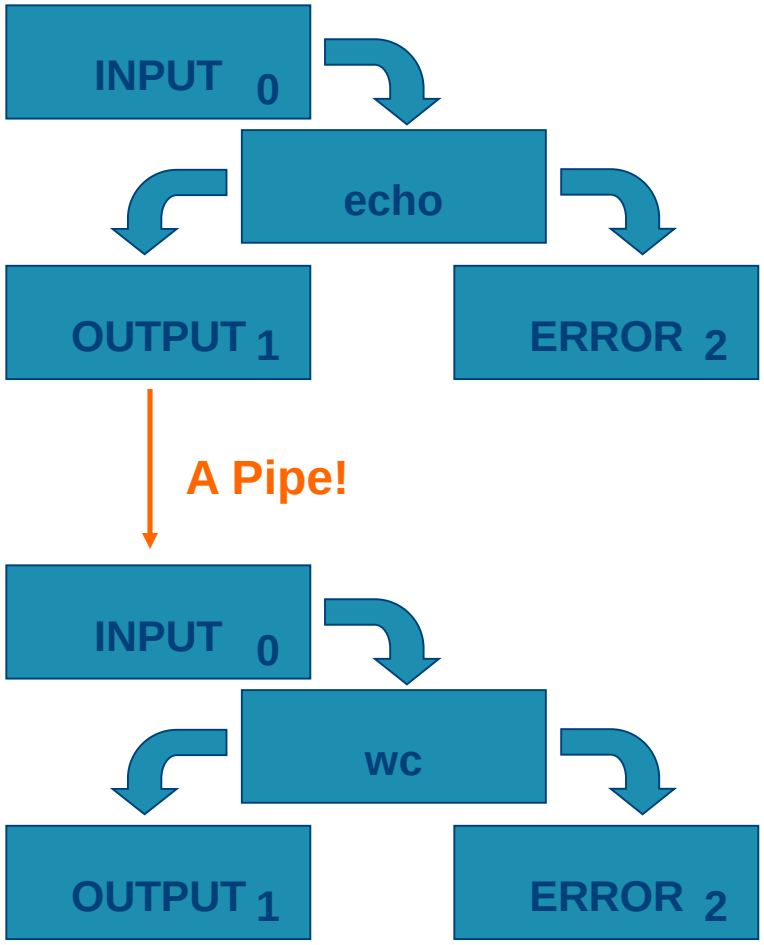
- Very helpful for searching files
- e.g. when we want to list the files, but only the ones that contain test in their name:

```
ls -la|grep test
```

Pipes

- Lots of Little Tools

```
echo “Hello” | \
 wc -c
```



Printing/output

- Basic use: `echo` (remember quots!)
  - `$ echo -n` -> do not output the trailing newline
  - `$ echo -e` -> enable interpretation of backslash escapes
    - \n new line
    - \r carriage return
    - \t horizontal tab
    - \v vertical tab

```
: vsc30468@tier2-p-login-3 ~ 15:49 $  echo -n "Enter your name:"
Enter your name:: vsc30468@tier2-p-login-3 ~ 15:49 $  echo -e "Enter your name:\n"
Enter your name:

: vsc30468@tier2-p-login-3 ~ 15:49 $  echo -e "Enter your name:\n and address:\n"
Enter your name:
 and address:

: vsc30468@tier2-p-login-3 ~ 15:49 $  echo -e "Enter your name:\t and address:\n"
Enter your name:         and address:

: vsc30468@tier2-p-login-3 ~ 15:49 $  echo -e "Enter your name:\v and address:\n"
Enter your name:
                and address:

: vsc30468@tier2-p-login-3 ~ 15:50 $  echo -e "1\v2\v3\t4\t5"
1
 2
  3     4       5
: vsc30468@tier2-p-login-3 ~ 15:50 $  echo -e "1\v2\v3\t4\r5"
1
 2
5 3     4
: vsc30468@tier2-p-login-3 ~ 15:50 $
```

Printing/output

- Alternatively (when formatting needed): `printf`
- `printf [-v var] format [arguments]`

  Writes the formatted *arguments* to the standard output under the control of the *format*. The -v option causes the output to be assigned to the variable *var* rather than being printed to the standard output.
  - d - Format a value as a signed decimal number.
  - u - Format a value as an unsigned decimal number.
  - s - Format a value as a string.
  - Format specifiers can be preceded by a field width to specify the minimum number of characters to print. A positive width causes the value to be right-justified; a negative width causes the value to be left-justified. A width with a leading zero causes numeric fields to be zero-filled. Usually, you want to use negative widths for strings and positive widths for numbers.

Printing/output

- Alternatively (when formatting needed): `printf`
- `printf [-v var] format [arguments]`

  Writes the formatted *arguments* to the standard output under the control of the *format*. The -v option causes the output to be assigned to the variable *var* rather than being printed to the standard output.

Printing/output

- d - Format a value as a signed decimal number.
- u - Format a value as an unsigned decimal number.
- f – format as a floating number.
- s - Format a value as a string.

- Format specifiers can be preceded by a field width to specify the minimum number of characters to print. A positive width causes the value to be right-justified; a negative width causes the value to be left-justified. A width with a leading zero causes numeric fields to be zero-filled. Usually, you want to use negative widths for strings and positive widths for numbers.

- Precision: The precision for a floating- or double-number can be specified by using .<DIGITS>, where <DIGITS> is the number of digits for precision

- \n – new line

Printing/output

- `$ printf "%50s\n" "This field is 50 characters wide..."` -> will be printed as a string of 50 characters ended with the new line

- `$ printf "%20s: %4d\n" "string 1" 12 "string 2" 122` -> prints string 1 in 20 character fiels and corresponding number 12 where 4 digits are reserved for it. Next it starts new line and prints string 2 and number 122.

- Note that printf reuses the format if it runs out of format specifiers, which in the examples above allows you to print two lines (four values) with only two format specifiers.

```
: vsc30468@tier2-p-login-3 ~ 16:15 $ printf "%20s: %4d\n" "string 1" 12 "string 2" 122
            string 1:   12
            string 2:  122
: vsc30468@tier2-p-login-3 ~ 16:15 $ printf "%-20s: %-4d\n" "string 1" 12 "string 2" 122
string 1            : 12
string 2            : 122
: vsc30468@tier2-p-login-3 ~ 16:15 $ printf "%-20s: %4d\n" "string 1" 12 "string 2" 122
string 1            :   12
string 2            :  122
: vsc30468@tier2-p-login-3 ~ 16:15 $
```

Printing/output

- The default behaviour of `%f` specifier is to print floating point numbers with 6 decimal places. To limit a decimal places to 2 we can specify a precision in a following manner:
  `$ printf "%.2f\n" 255` -> will print: `255.00`

- Formatting to three places with preceding with 0, separated with tab:
  `$ printf "%03d\t" 2 3` -> will print: `002	003`

Printing/output

- How to create a table with multiple items?
- Definition of formats:

```
header="\n %-10s %8s %10s %11s\n"
format=" %-10s %08d %10s %11.2f\n"
```

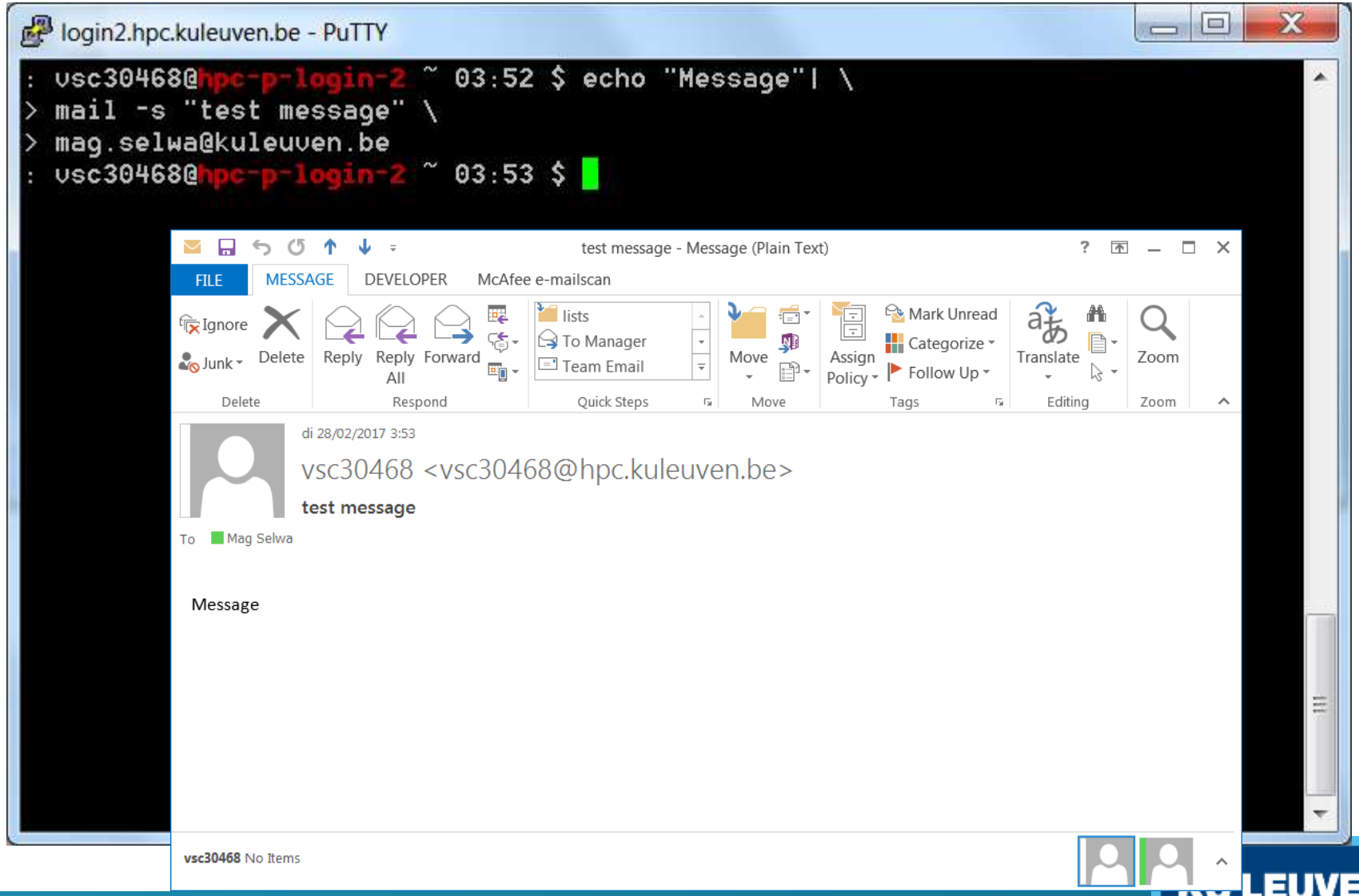
- Values to print:

```
printf "$header" "ITEM NAME" "ITEM ID" "COLOR"
"PRICE";printf "$format"  Triangle 13 red 20 Oval
204449 "dark blue" 65.656 Square 3145 orange .7
```

```
: x0076109@tier2-p-login-3 ~ 14:58 $ header="\n %-10s %8s %10s %11s\n"
: x0076109@tier2-p-login-3 ~ 14:58 $ format=" %-10s %08d %10s %11.2f\n"
: x0076109@tier2-p-login-3 ~ 14:58 $ printf "$header" "ITEM NAME" "ITEM ID" "COL
OR" "PRICE";printf "$format"  Triangle 13 red 20 Oval 204449 "dark blue" 65.656
Square 3145 orange .7

 ITEM NAME    ITEM ID      COLOR       PRICE
 Triangle   00000013        red       20.00
 Oval       00204449  dark blue       65.66
 Square     00003145     orange        0.70
: x0076109@tier2-p-login-3 ~ 14:58 $
```

Email Notification



Dates

```
$ DATESTRING=`date +%Y%m%d`
$ echo $DATESTRING
20170227
$ man date
```

Defining local variables

- As in any other programming language, variables can be defined and used in shell scripts.
- Unlike other programming languages, variables in Shell Scripts are not typed.
- Examples :

```
a=1234  # a is NOT an integer, a string instead
b=$a+1 # will not perform arithmetic but be the string '1234+1'
b=`expr $a + 1 `  will perform arithmetic so b is 1235 now.
```

Note : +,-,/,*,**, % operators are available.

```
b=abcde # b is string
b='abcde' # same as above but much safer.
b=abc  def # will not work unless 'quoted'
b='abc def' # i.e. this will work.
```

IMPORTANT: DO NOT LEAVE SPACES AROUND THE =

Some commands

- `cut` – cuts columns of text from a file
- `echo` – displays a line of text
- `paste` – will paste columns of text into a file
- `sort` – sorts a file of lines alfabetically
- `tr` – translates between characters (e.g. tr a-z A-Z)
- `sleep` - Delay for a specified amount of time
- `uniq` - Remove duplicate lines from a sorted file
- `bc` – command line calculator (scale=2 to see 2 digits)

Cut flags

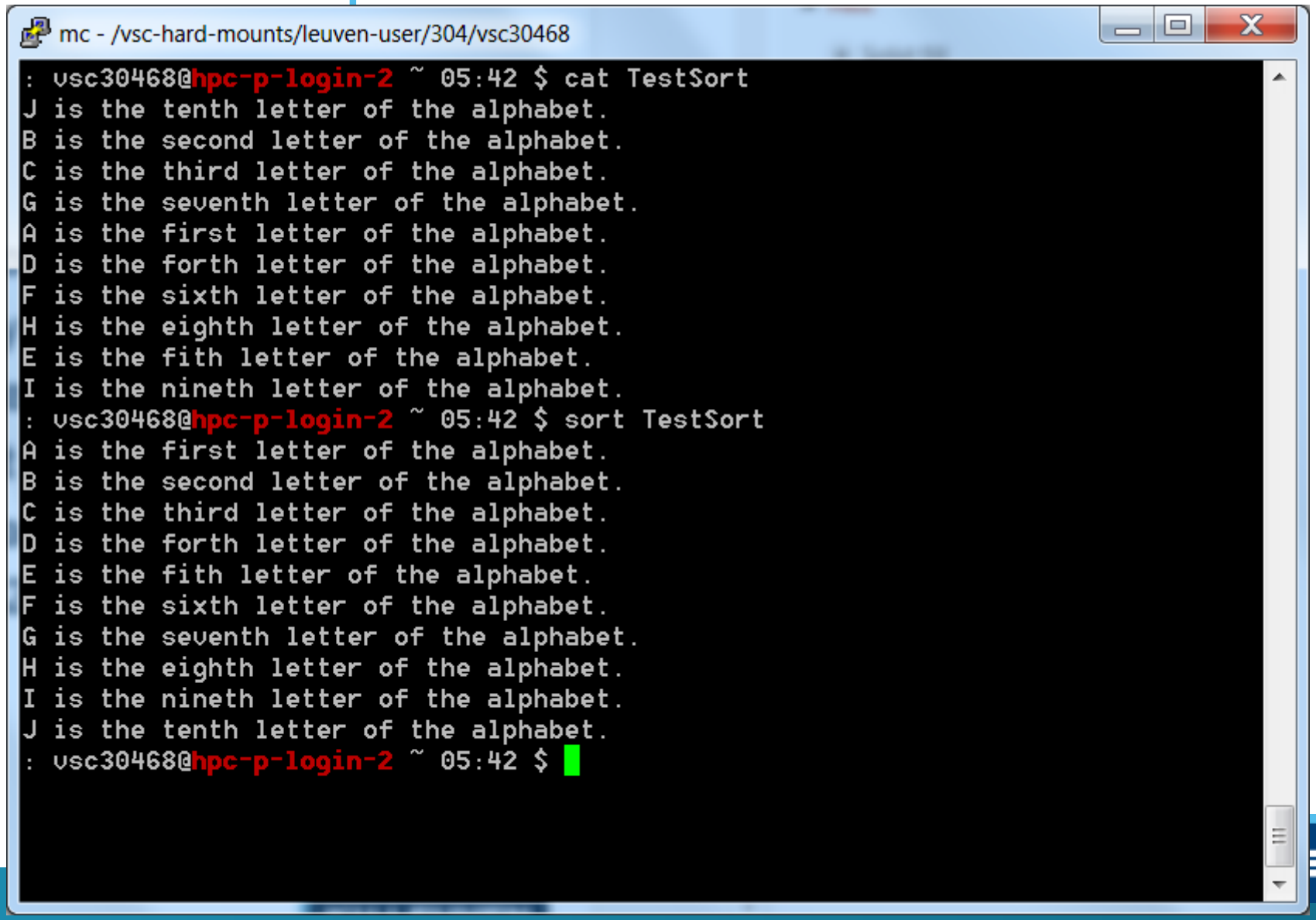
- -d
  - Delimiter
- -f
  - Field number
- Example
  - `cut –d' ' -f3 myFile`

Sort flags

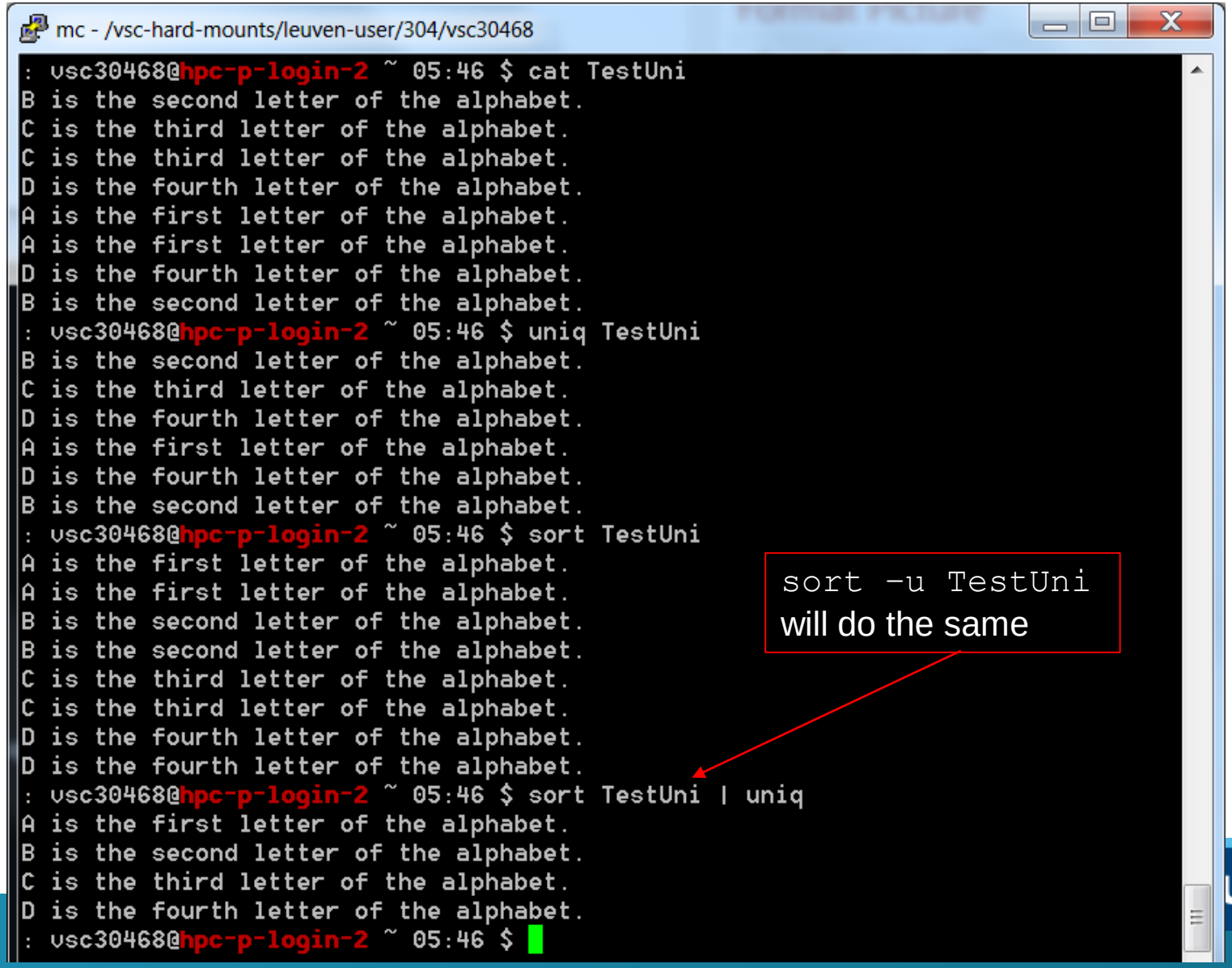
- -r
  - reverse
- -f
  - ignore case
- -k`n`
  - sort according to column `n`
- `-n`
  - compare according to string numerical value
- -V
  - natural sort of (version) numbers within text (for files like file-1.txt, …, file-100.txt)
- Example
  - `sort -r -k3 < myFile`

KU LEUVEN

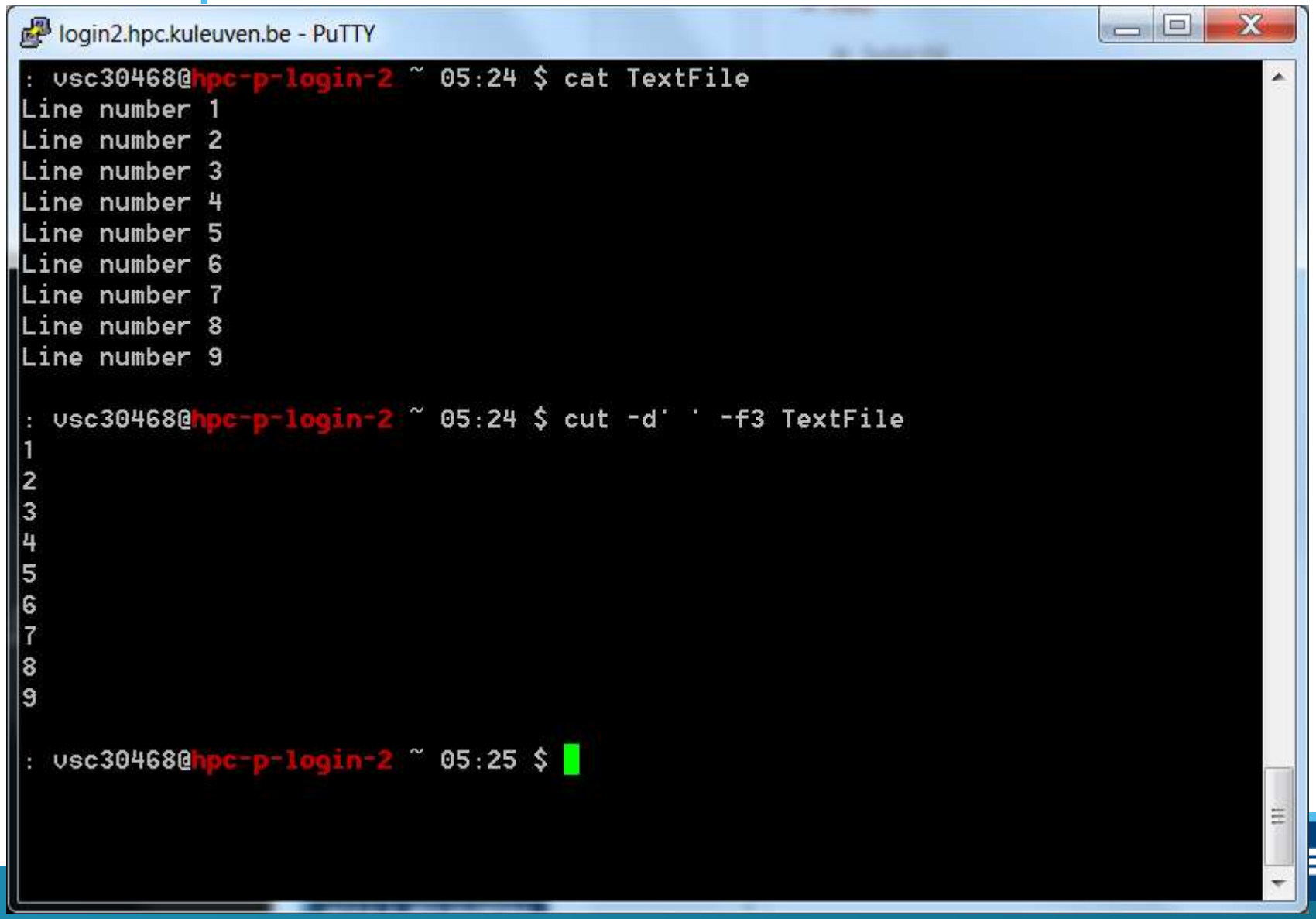
Sort example



Uniq example

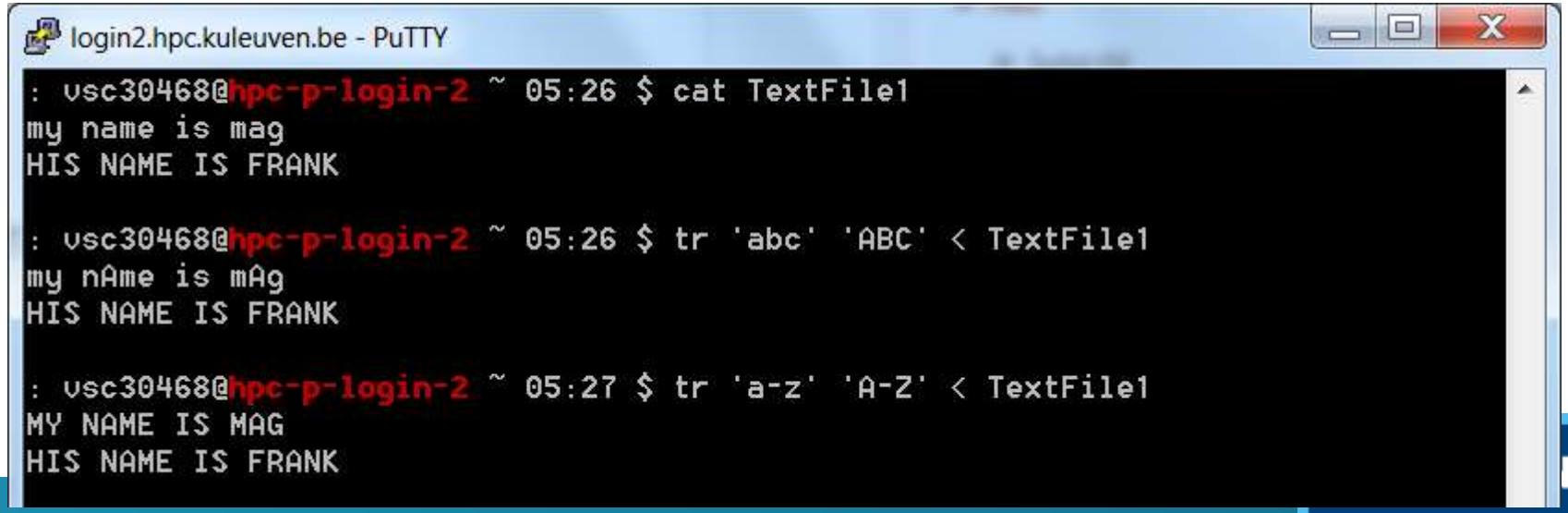


Example Of Cut



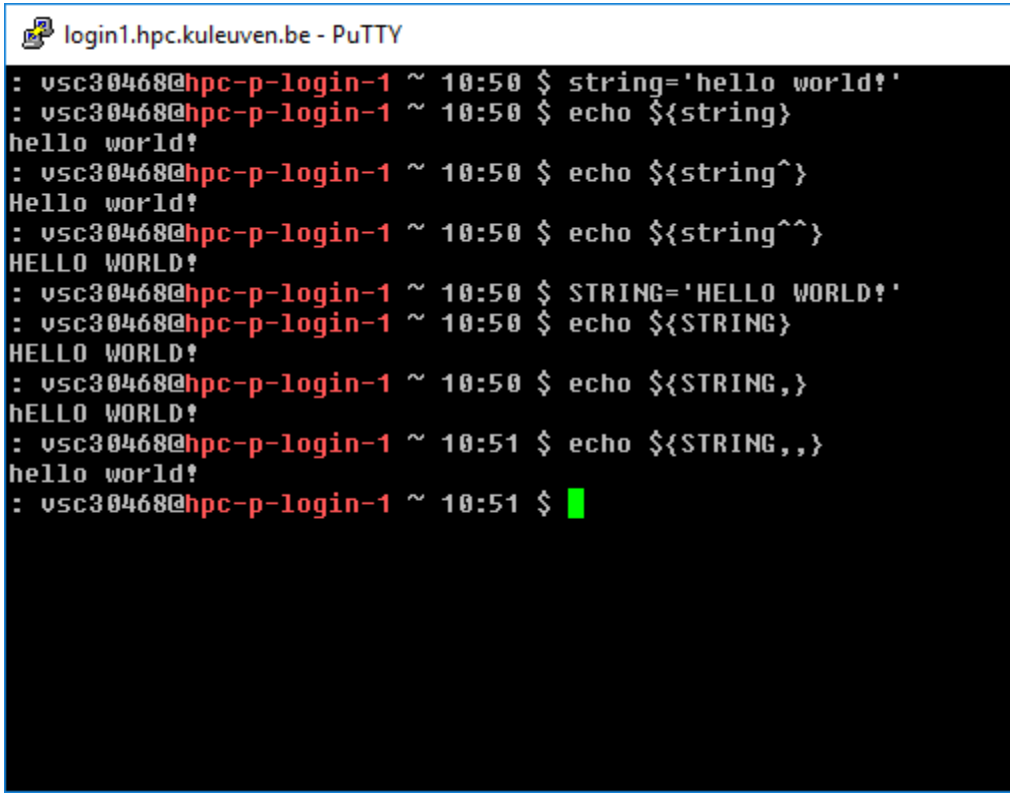
The ‘tr’ Command

- Translate
- Change on a one to one basis characters from one thing to another
- Usage: `tr ‘Set1’ ‘Set2’`
  - Example: `tr ‘abc’ ‘ABC’ < myFile`
  - `tr '[:lower:]' '[:upper:]' < myFile`



Lower/Upper case letters

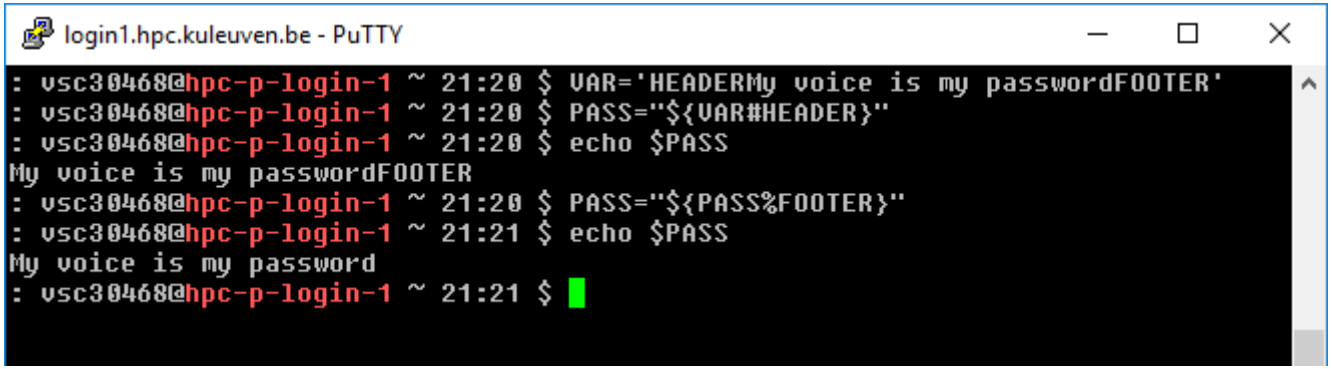
- Capitalize:
  - `^^` - all the letters
  - `^` - only the first letter
- Lowercase:
  - `,,` - all the letters
  - `,` - only the first letter



KU LEUVEN

String manipulation

- The # means ‘match and remove the following pattern from the start of the string’
- The % means ‘match and remove the following pattern from the end of the string



Globbing: use wildcard

Wildcard	Function
*	Matches 0 or more characters
?	Matches 1 character
[abc]	Matches one of the characters listed
[a-c]	Matches one character in the range
[!abc]	Matches any character not listed
[!a-c]	Matches any character not listed in the range
{tacos,nachos}	Matches one word in the list

grep – Global / Regular Expressions / Pattern

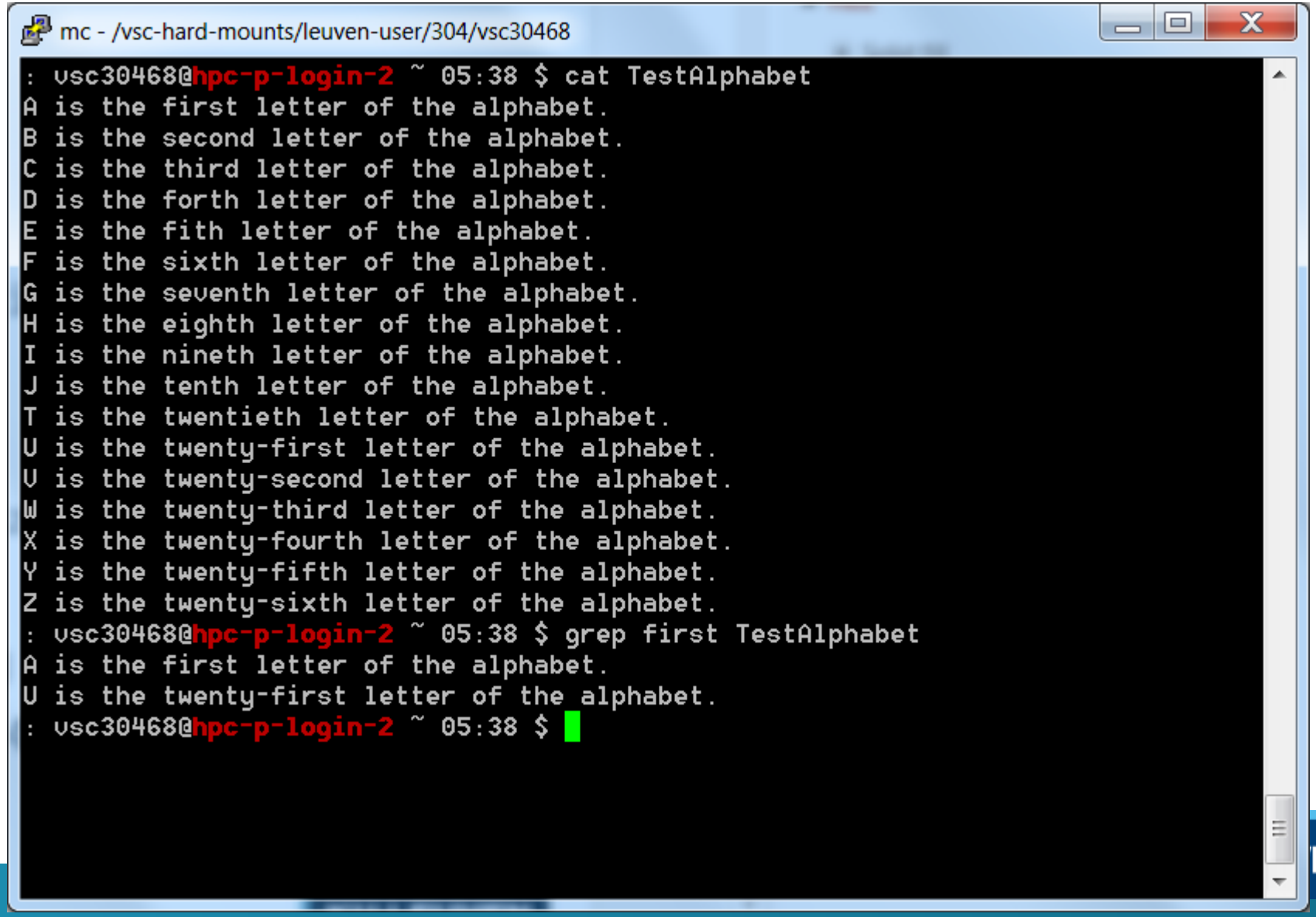
- Searches the internals of files and tries to match patterns
- Used to see if a file contains data you are looking for
- Will print out every line that contains a match for that pattern
- Usage: grep [OPTIONS] pattern [FILE]

Useful tools

- `grep`
  - Pattern searching
  - Example: `grep parameters pattern filename`
  - `-v` (invert the sense of matching, to select non-matching lines)
  - `-i` (ignore case)
  - `-e/-E` (Look for expression/extended expression)

  - `$ grep ";E;" $PBSLOGSDIR/$file | grep -E "gpu1\/|gpu2\/|gpu3" > $file`

Useful tools



Common Flags

- `-i`
  - Case insensitive (upper and lower cases are treated the same)
- `-n`
  - Print out the line numbers
- `-r`
  - Recursively traverse the directory
- `-v`
  - Invert the results (show all non-matching lines)
- `-l`
  - Shows only the name of matching file, not the matches (useful in combination with find)
- `alias grep='grep --color=auto'`
  - Possible alias in .bashrc to highlight matches in the output, making it easy to spot them.

Easiest grep Usage

- The easiest way to use grep is also the most common way to use grep
  - Search files for occurrences of a string (word)
- The pattern you search for can simply be a word

Regular Expressions

- grep can be used to find words that match a certain pattern, not just a given word
- The language of regular expressions is used to describe these patterns
- This includes wildcards, repetitions, and complex patterns

How grep Views Regular Expressions

- Unfortunately, grep's regular expressions are completely different than the shell wildcards
- Some of the symbols are the same, but they are used in different ways
- Always use quotes (') so that the wildcards are interpreted by grep and not the shell

Notation

- ^
  - Beginning of the line – left rooted
- $
  - End of the line – right rooted
- .
  - Any single character
- [xy]
  - Any character in the set
- [^a-z]
  - Any character not in the set
- B*
  - Zero or more occurrences of B

Examples

- .*
  - Zero or more of any character
  - Will match any pattern
- ^ab*
  - Any line that starts with a and has zero or more b’s immediately following
    - ab
    - abbbb
    - abb

Examples

- [0-9]
  - Any number
    - 1002
    - 0909
- bye$
  - The pattern “bye” located at the end of the line
    - Hello and goodbye

One More Slide Of Examples…

- [^g]$
  - Match any line that does not end in g
- [:alpha:]*
  - Any word that contains zero or more alphabetic characters

Inverting The Answers

- grep –v ‘this’ testFile
  - Will find all lines that do not contain the word this
  - Works exactly the same with regular expressions
- grep –v ‘[^g$]’ testFile
  - Finds all lines that end in g

Hands-on 1

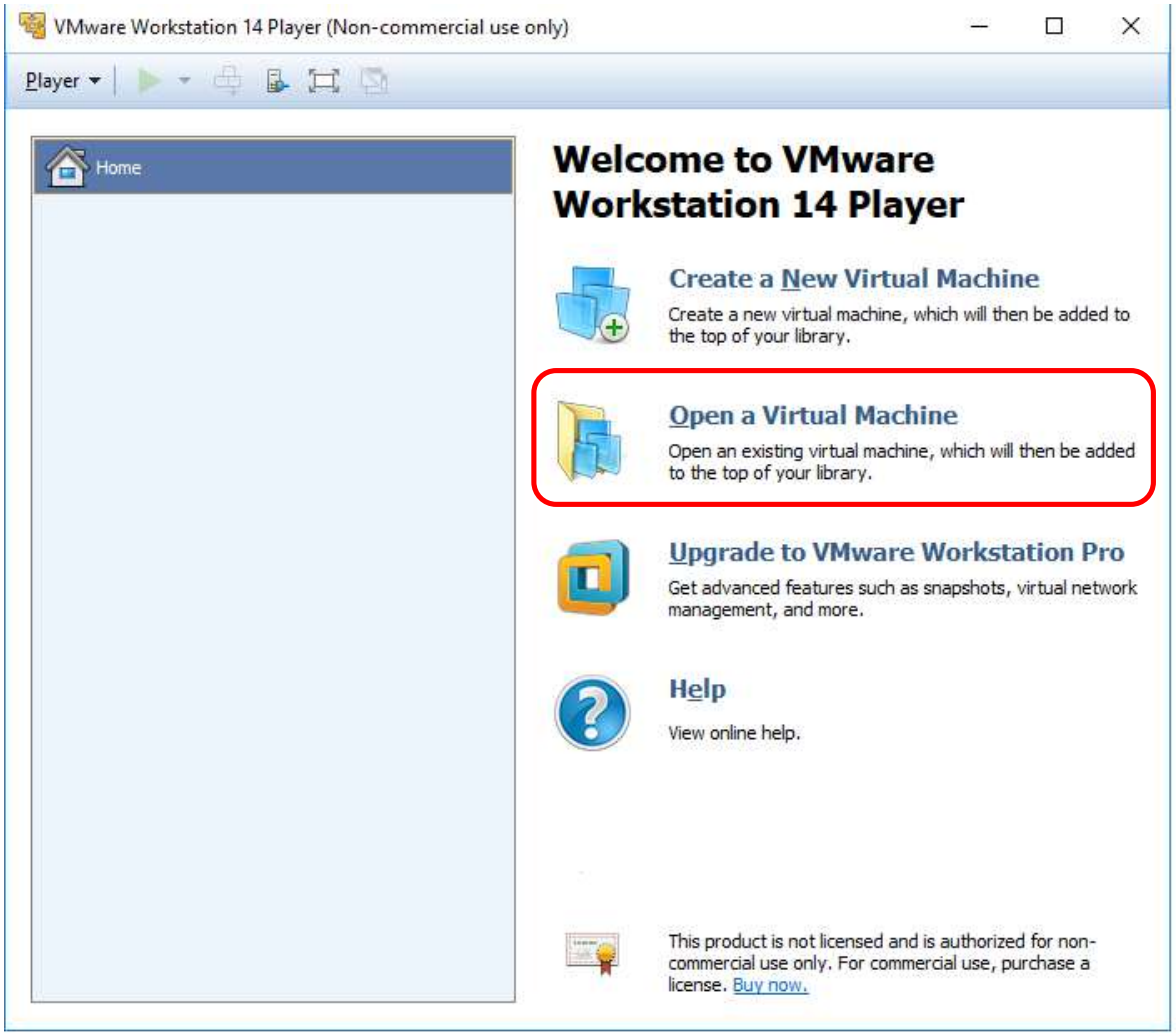
KU LEUVEN

Vmware in pc-klas

KU LEUVEN

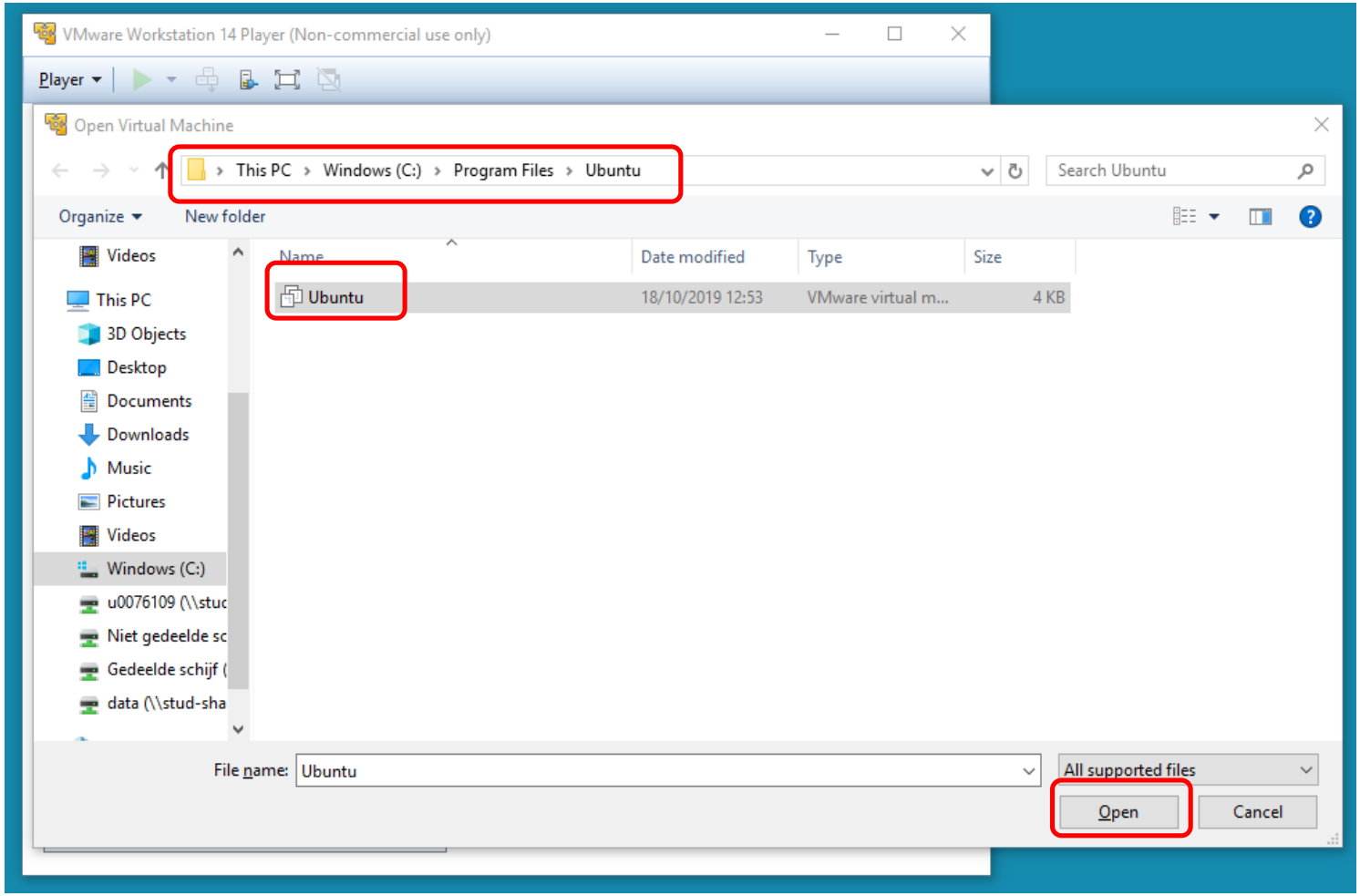
Vmware in pc-klas

Vmware in pc-klas

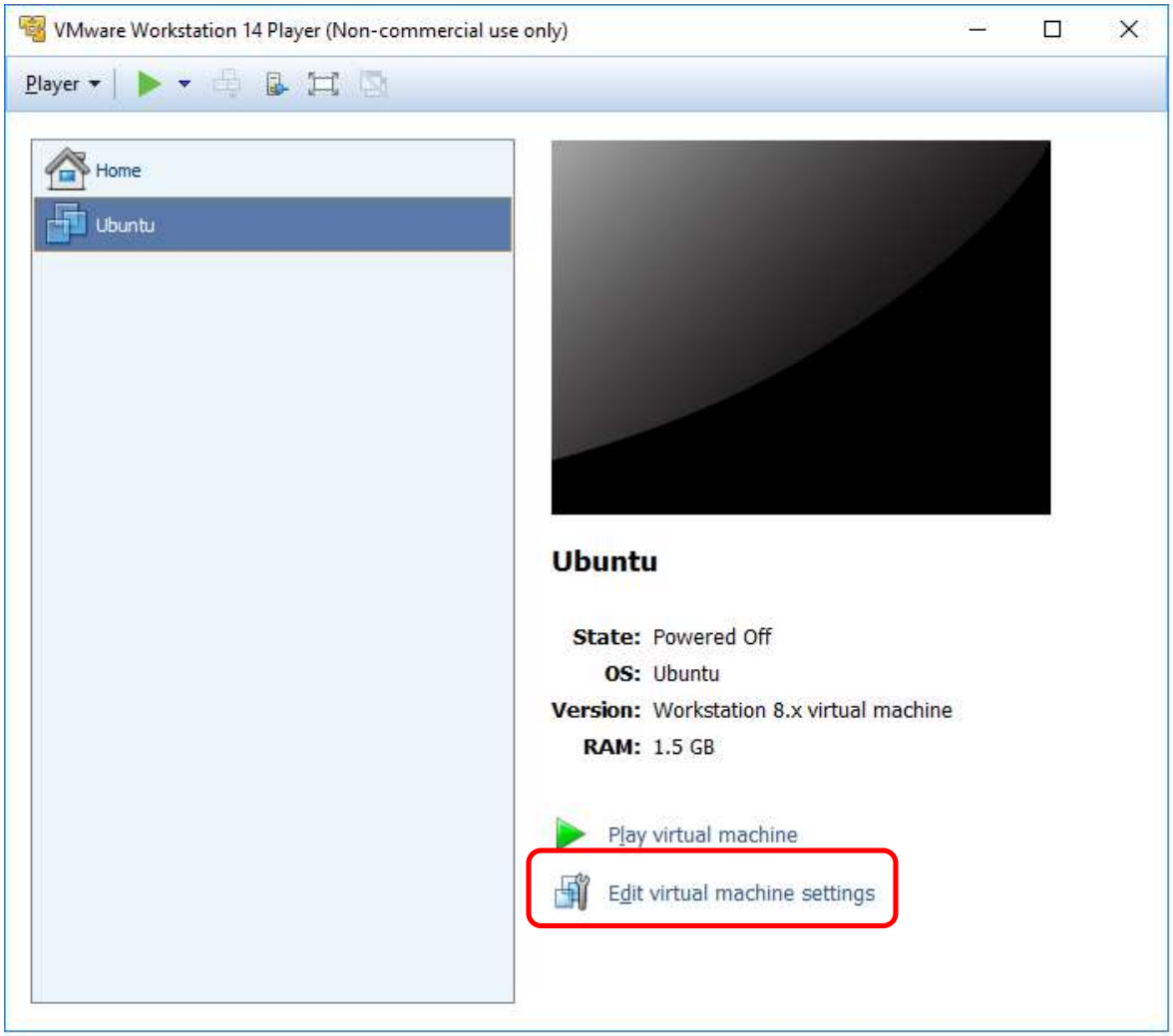


KU LEUVEN

Vmware in pc-klas

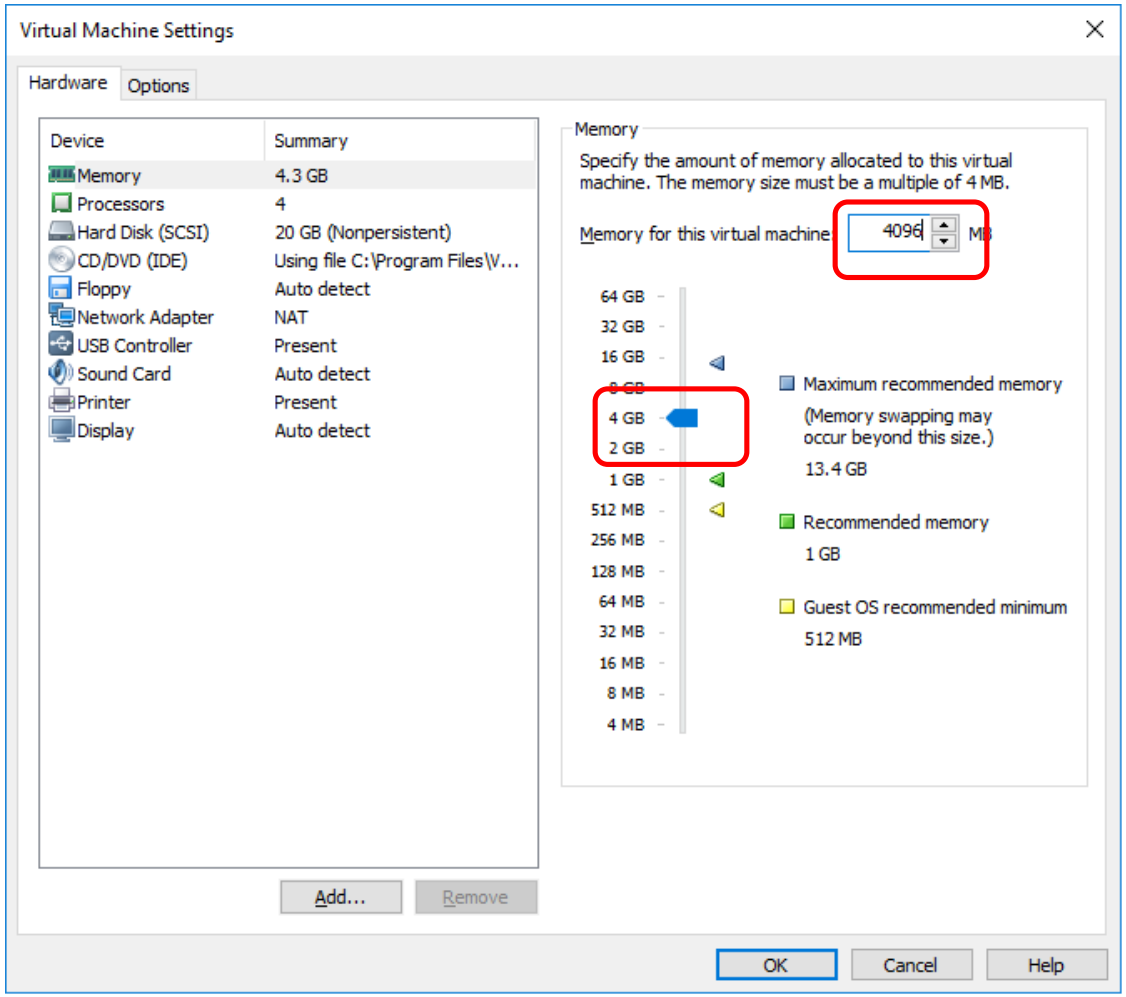


Vmware in pc-klas



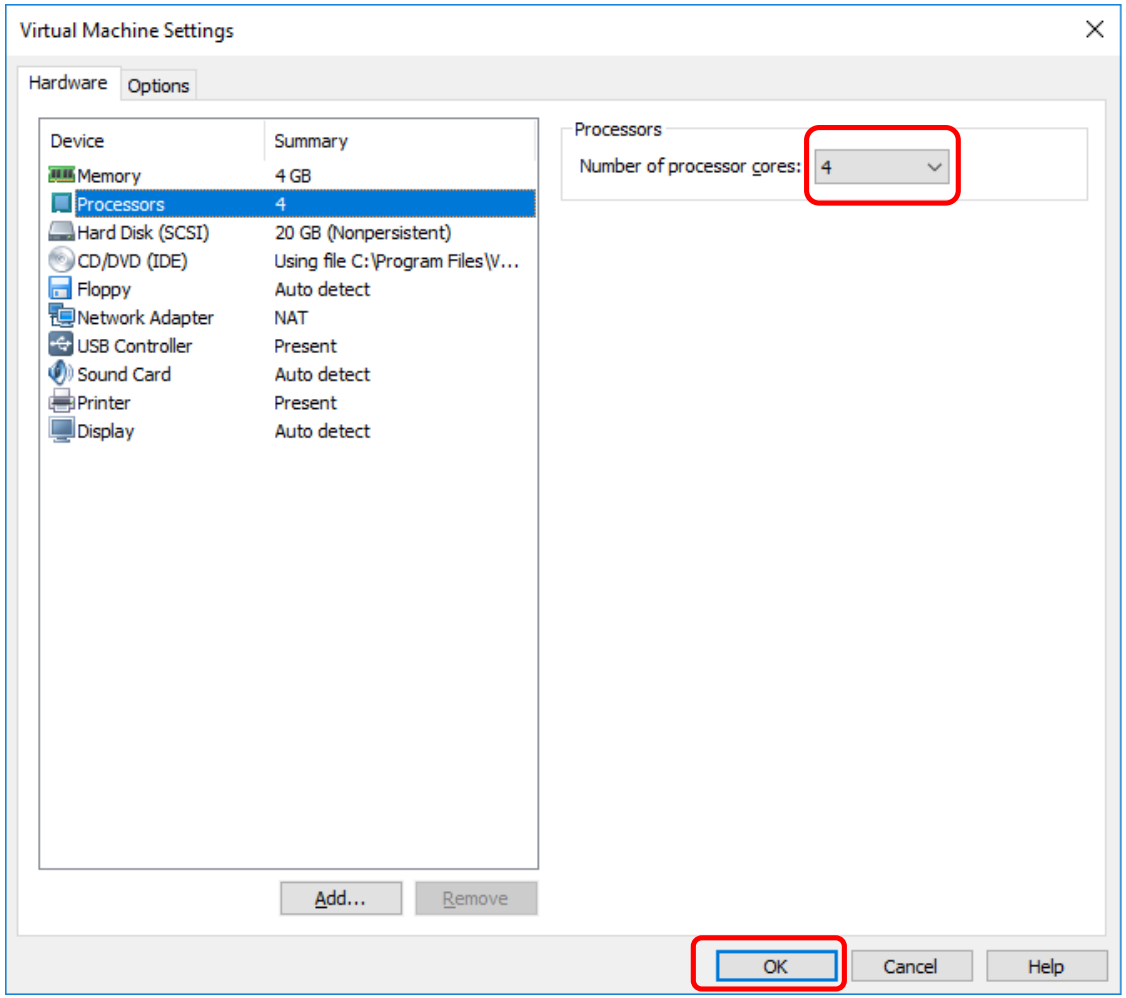
KU LEUVEN

Vmware in pc-klas



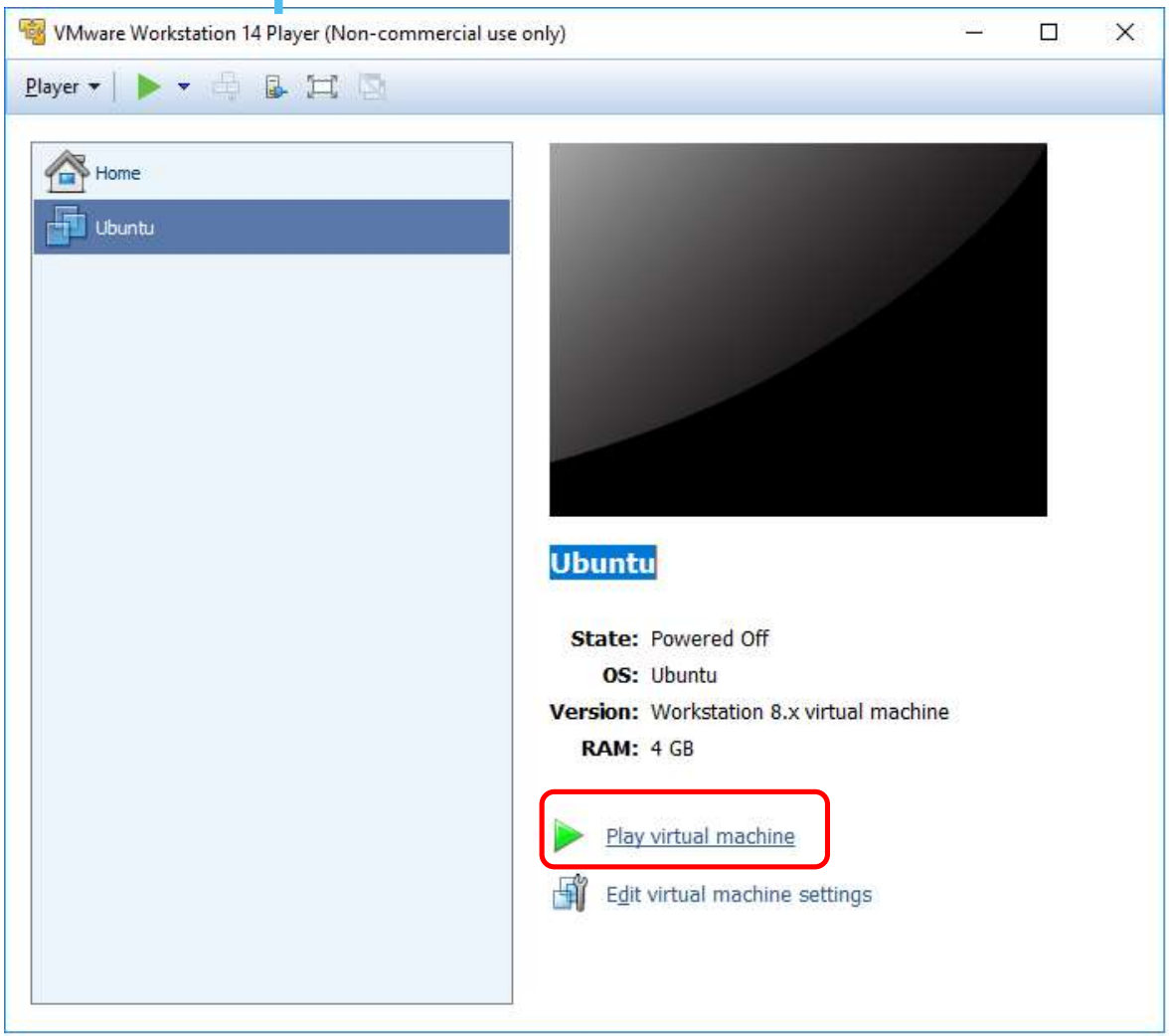
KU LEUVEN

Vmware in pc-klas



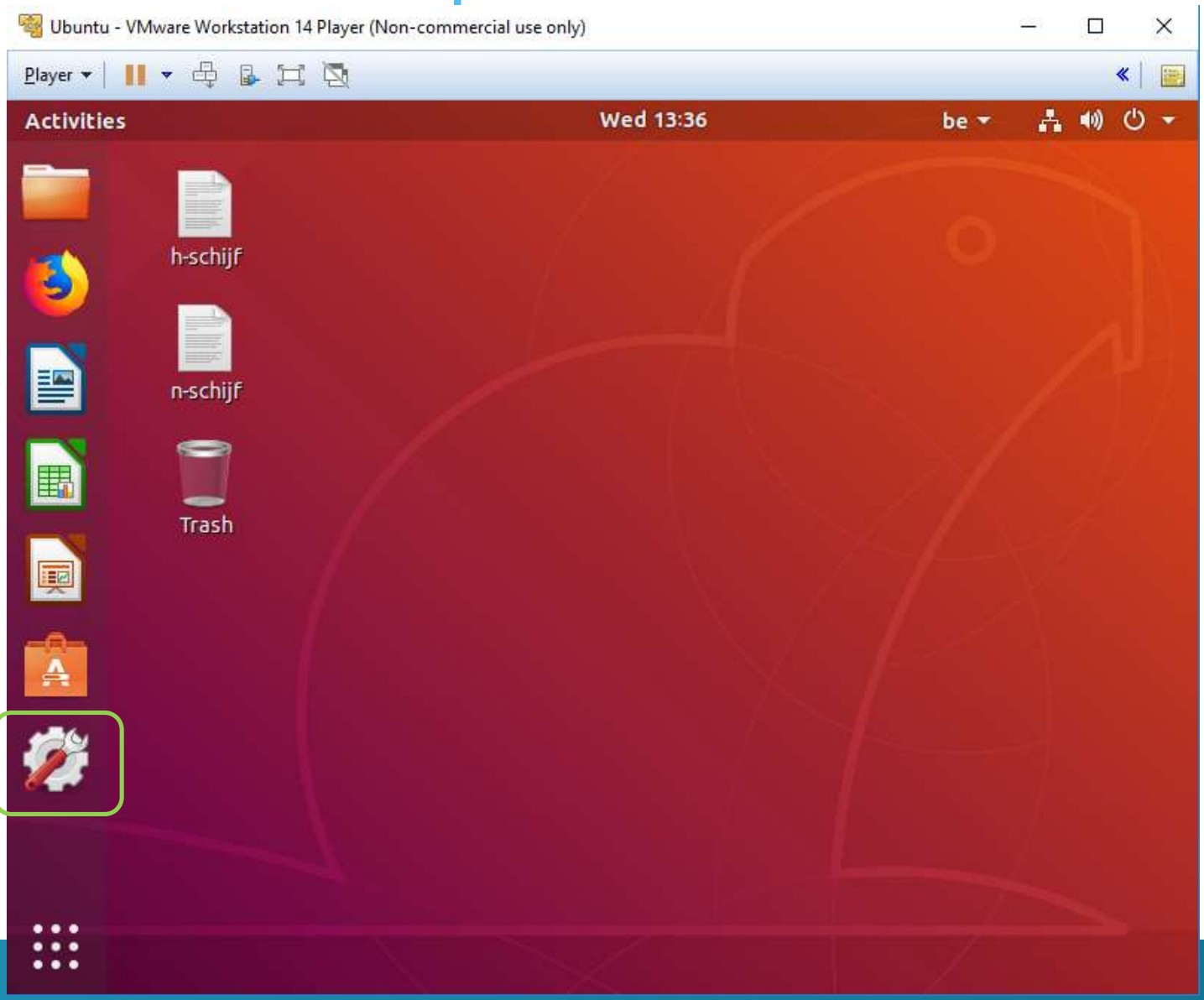
KU LEUVEN

Vmware in pc-klas



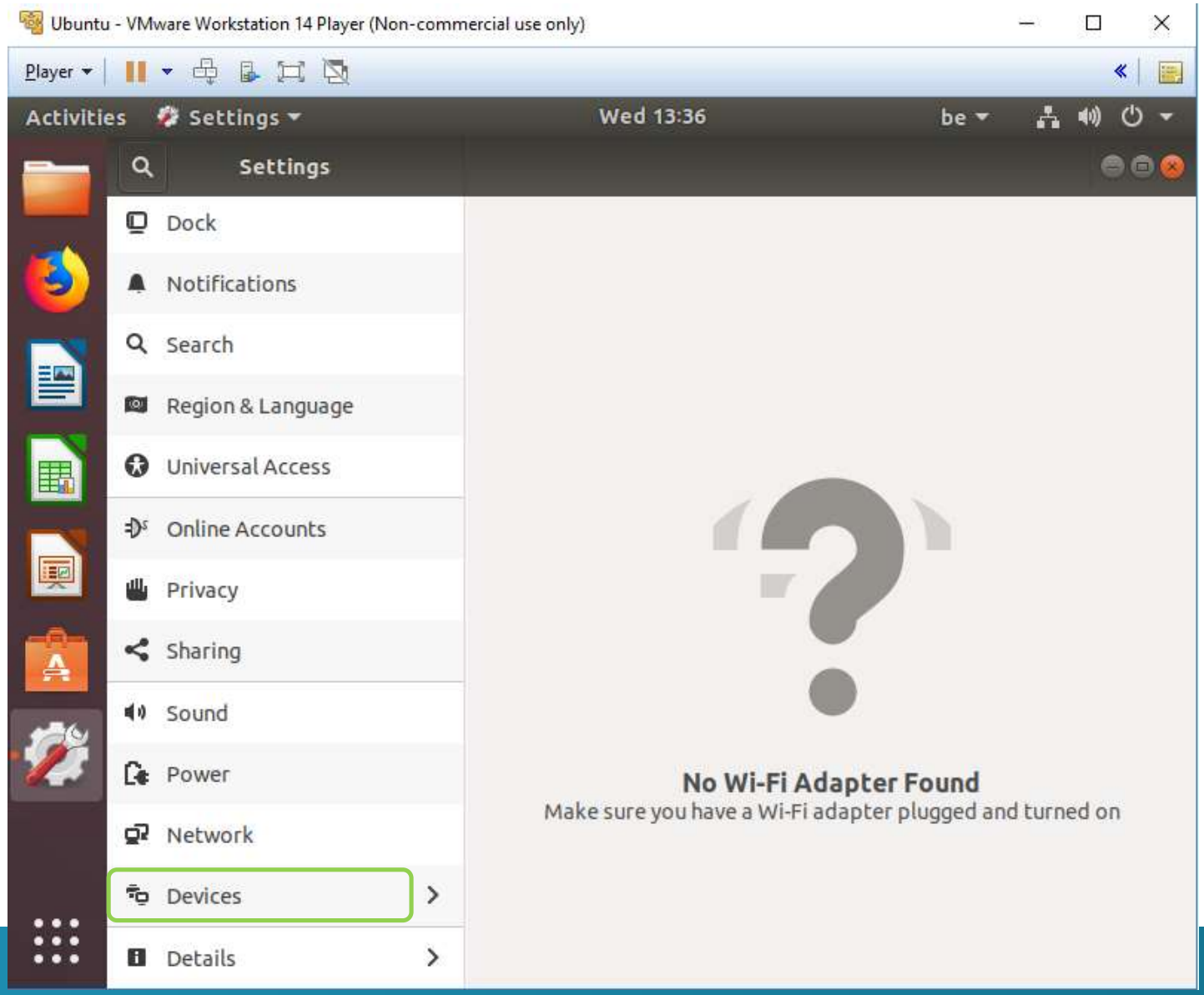
KU LEUVEN

Vmware in pc-klas



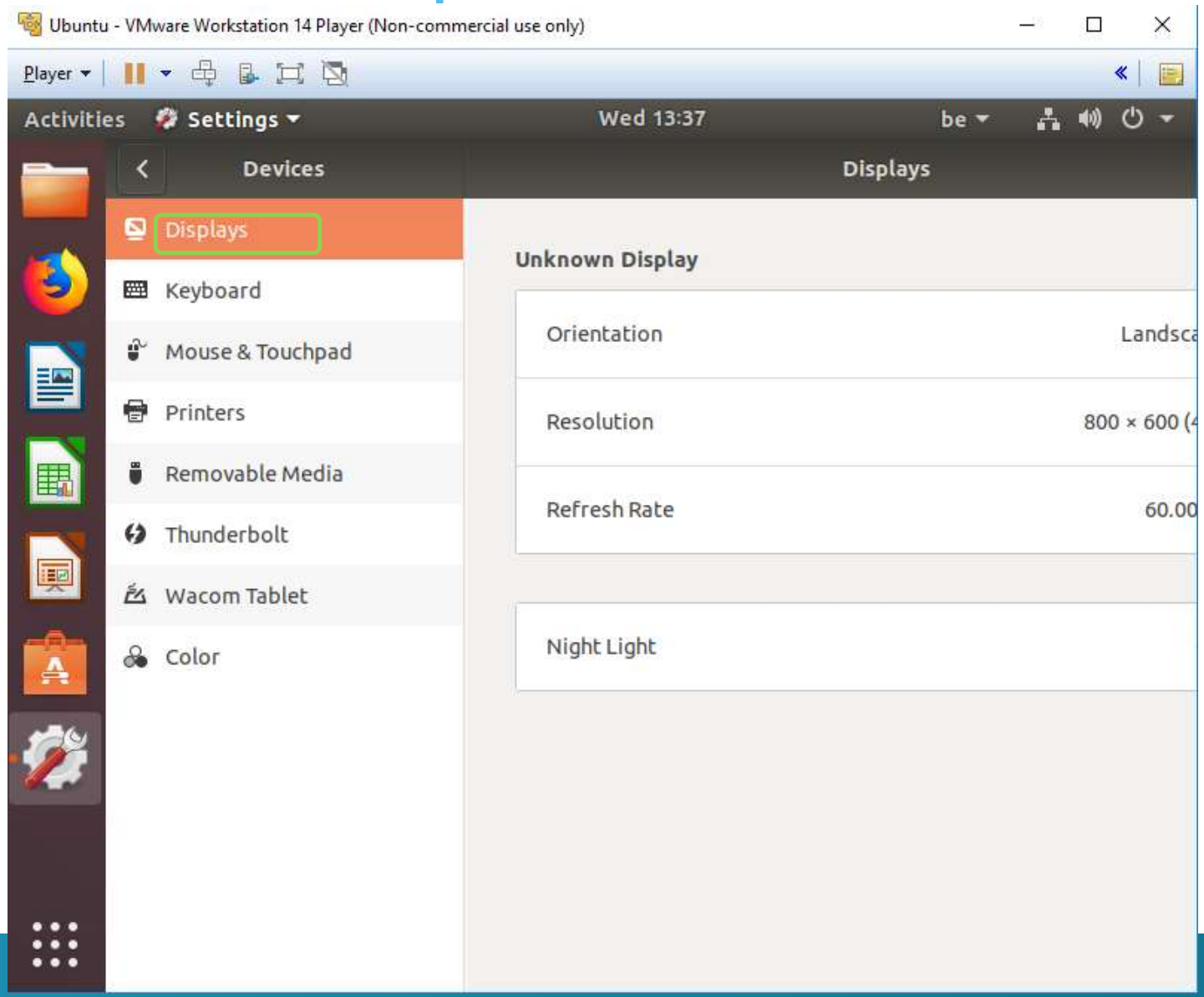
KU LEUVEN

Vmware in pc-klas



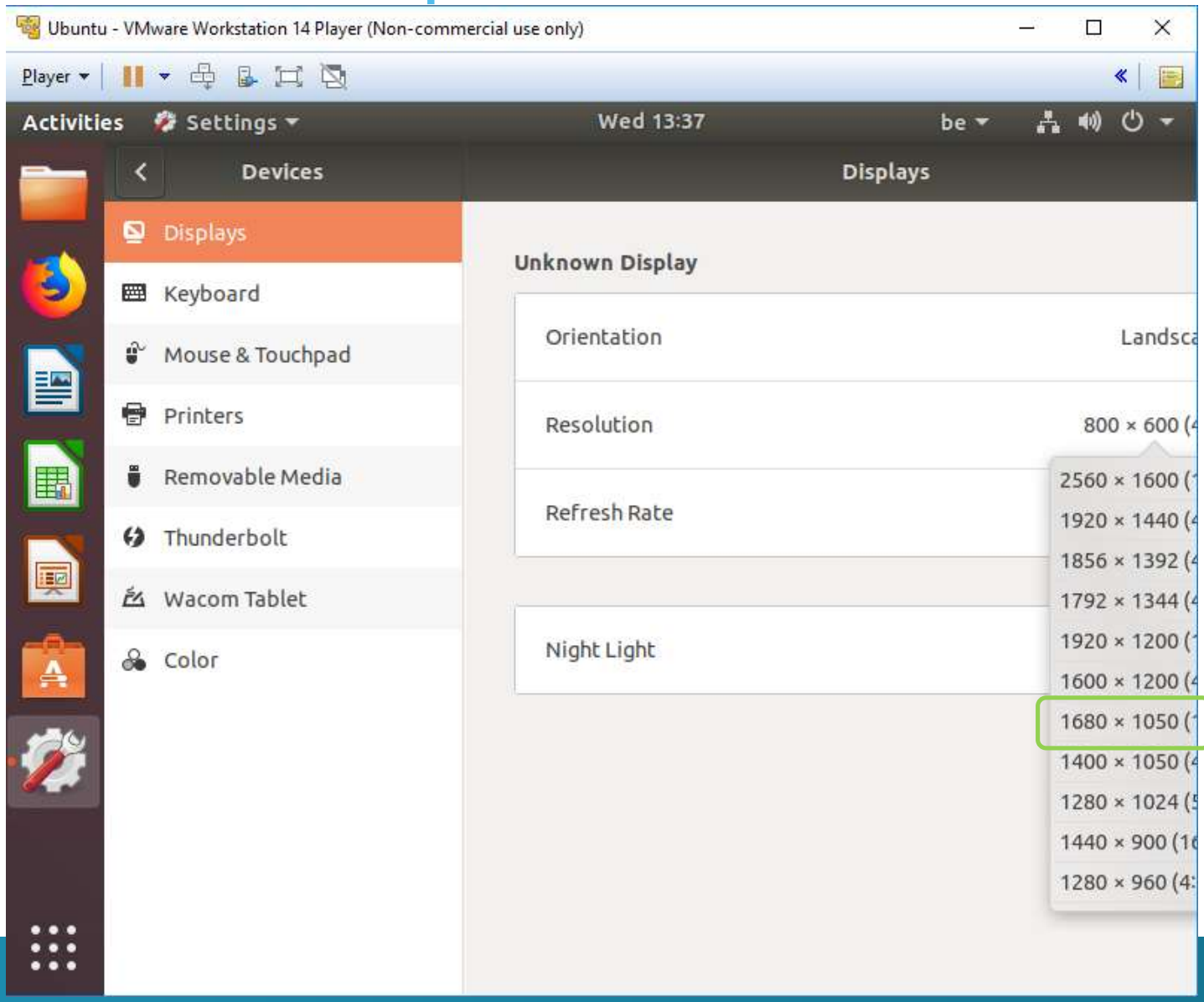
KU LEUVEN

Vmware in pc-klas

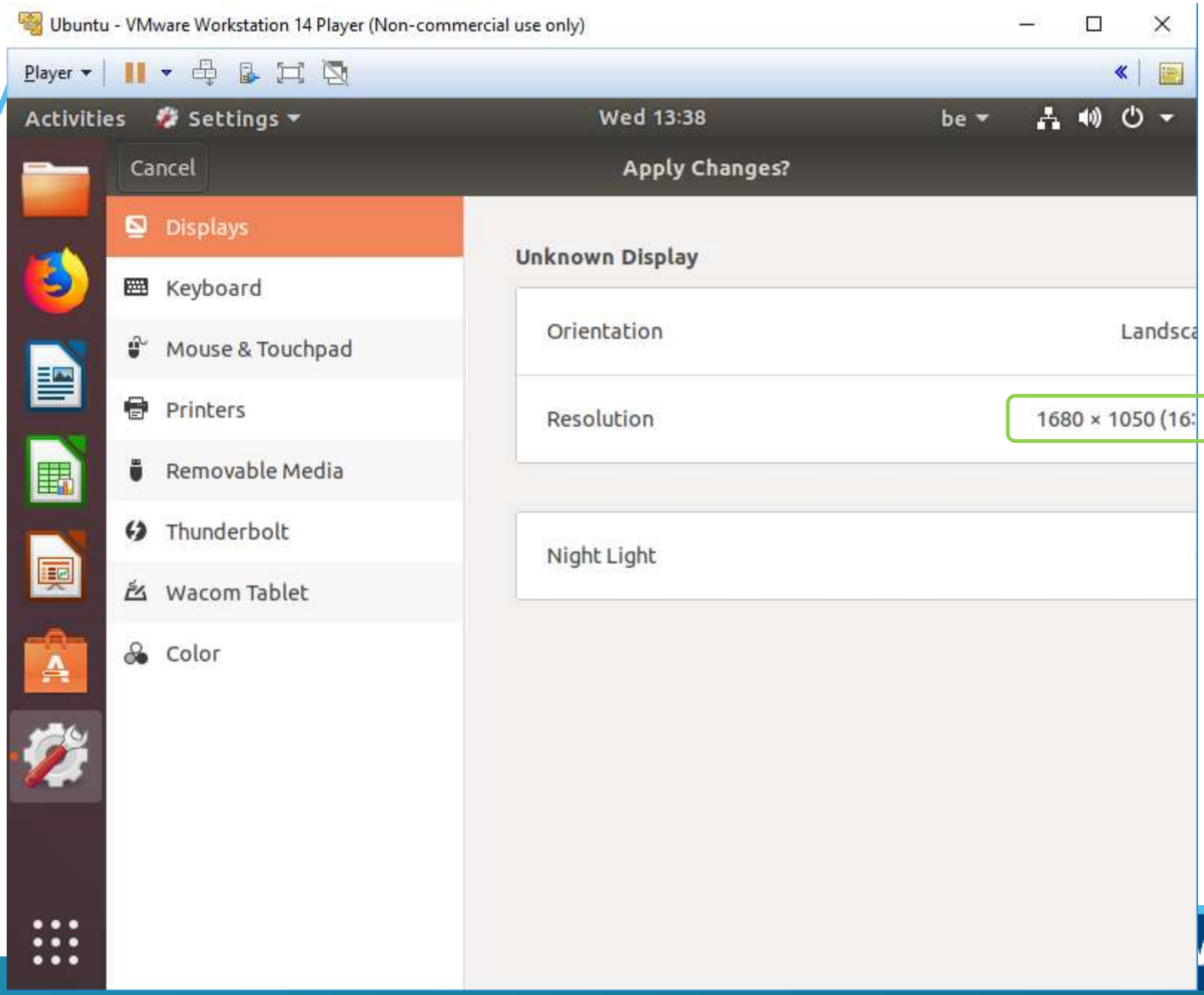


KU LEUVEN

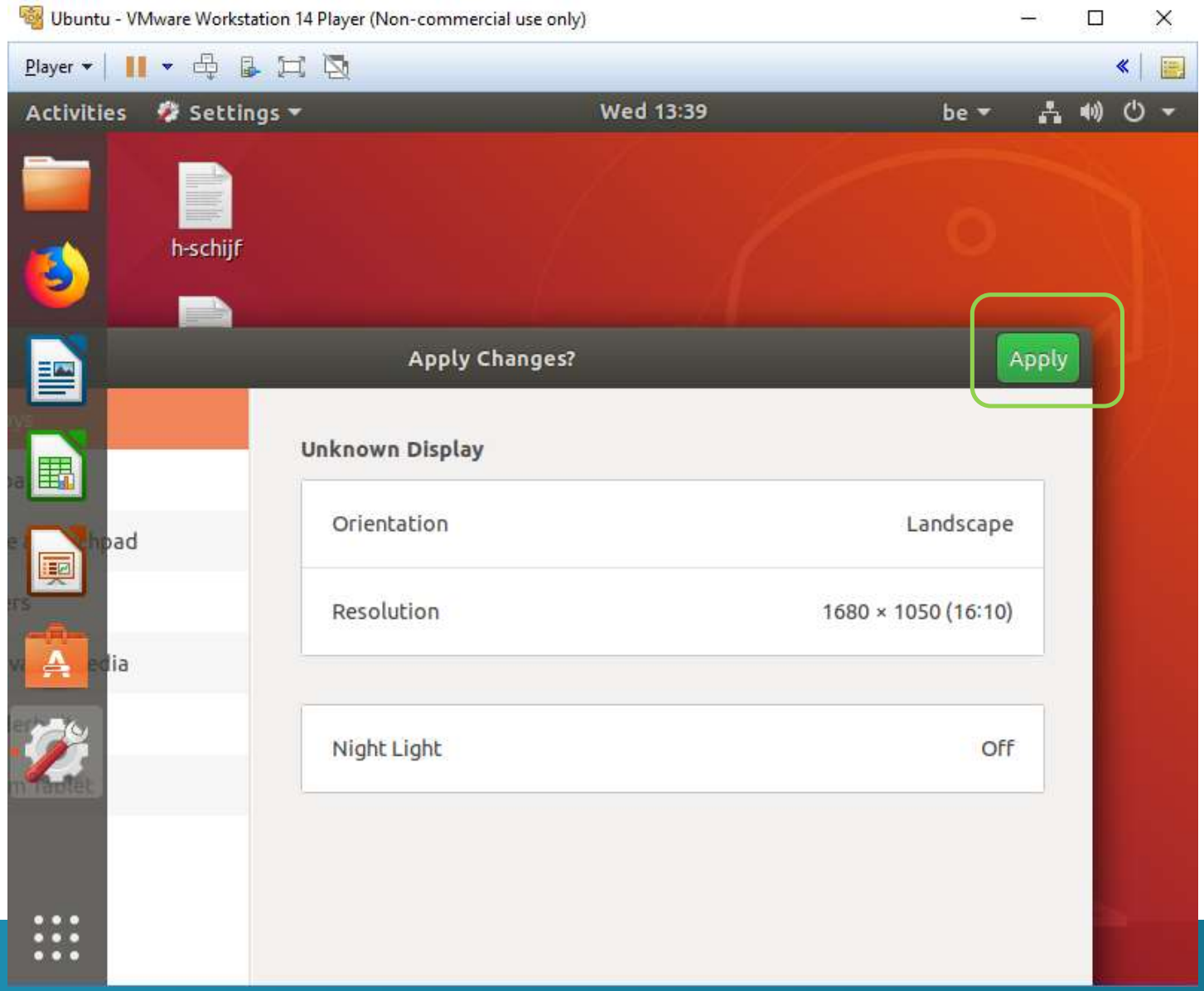
Vmware in pc-klas



KU LEUVEN

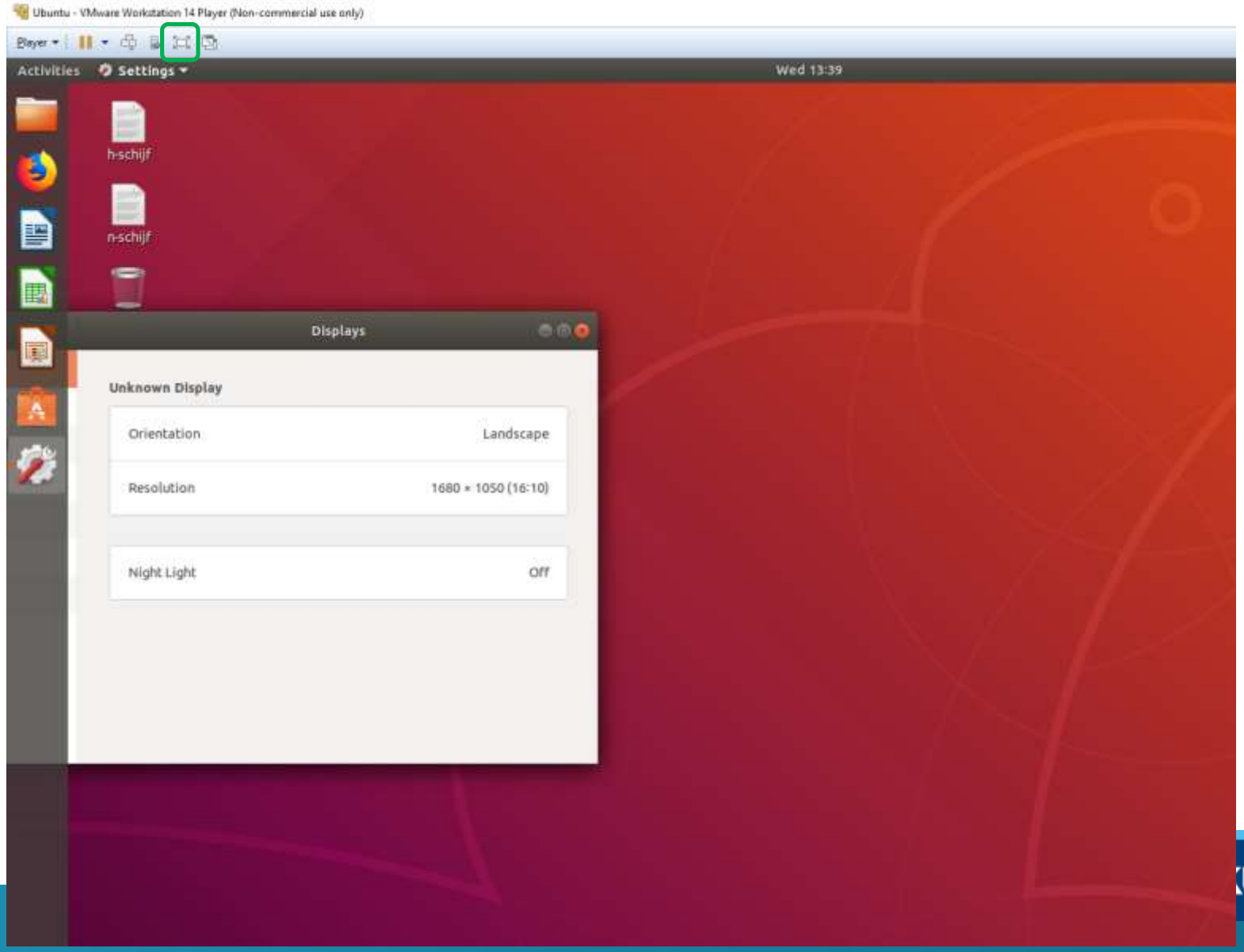

Ubuntu - VMware Workstation 14 Player (Non-commercial use only)
Player
Activities
Settings
Wed 13:38
be
Cancel
Apply Changes?
Displays
Keyboard
Mouse & Touchpad
Printers
Removable Media
Thunderbolt
Wacom Tablet
Color
Unknown Display
Orientation
Landsca
Resolution
1680 × 1050 (16:
Night Light

Vmware in pc-klas

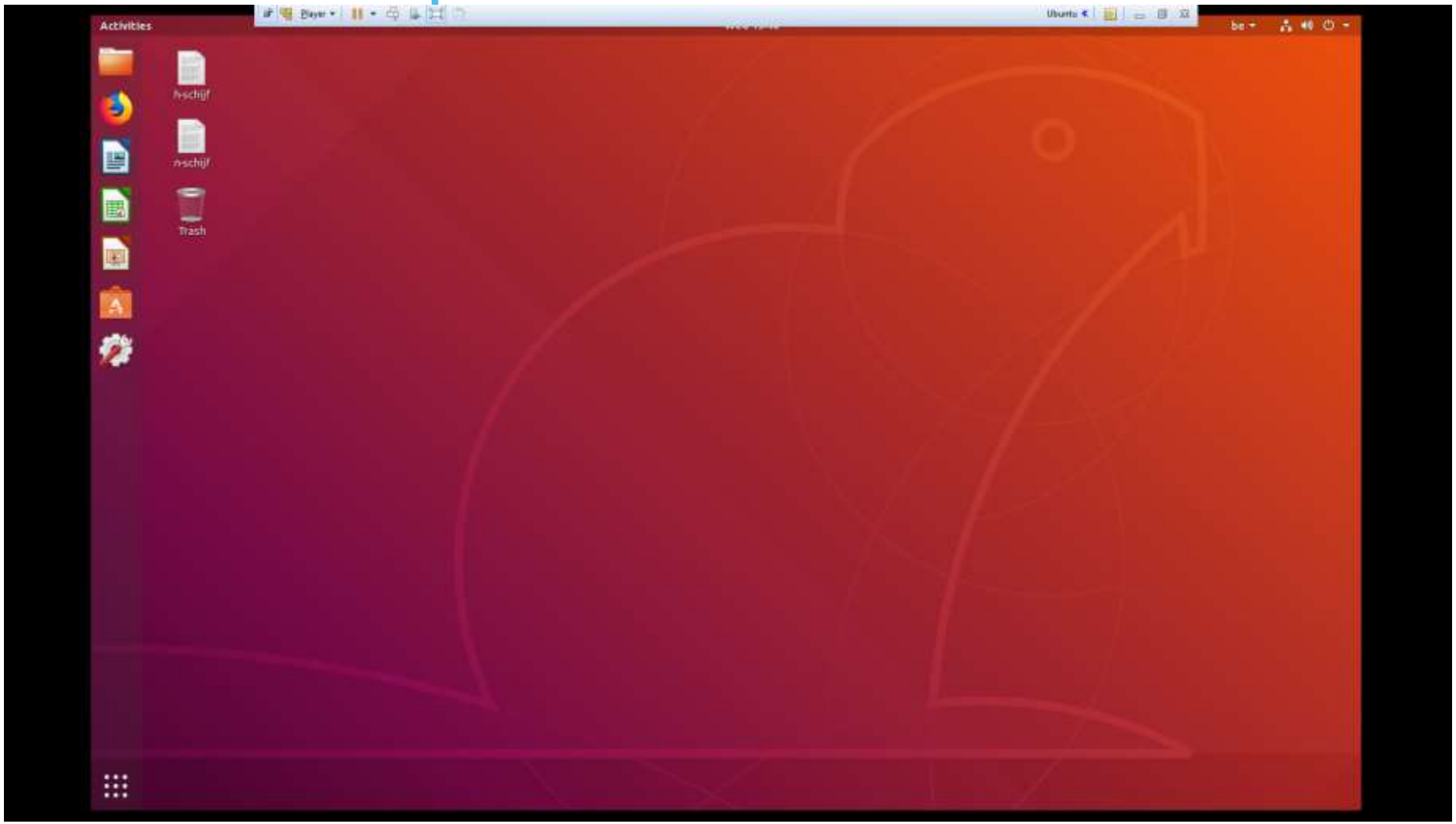


KU LEUVEN

Vmware in pc-klas



Vmware in pc-klas



KU LEUVEN

1. Create a file that has a filename of the date in the format DD-MM-YY.log
2. Download the file https://sites.google.com/site/magselwakuleuven/downloads/ex1.fa and change all the capital letter A and C into lower case (save into a new file)
3. Modify ex1.fa file so that A is changed into B, C into D, etc. (save into a new file)
4. Create a file containing some information, e.g. "*Hello Mag, it is a nice sunny day.*" Write a conversion tool for Rot13 (https://en.wikipedia.org/wiki/ROT13). Apply it once to the file, check the output, apply again and compare the output with the original file.
5. Download the file https://sites.google.com/site/magselwakuleuven/downloads/lp_hpcinfo and save into a new file only column 5
6. Remove duplicated lines from the file
7. Sort lp_hpcinfo according to the 5th column
8. Download the file https://sites.google.com/site/magselwakuleuven/downloads/20170601 and print only lines that contain Exit_status=0
9. From 20170601 print only lines that do not contain ";E;"
10. From 20170601 print only lines that contain r3i0n11 or r3i0n12
11. From 20170601 print only lines that contain both ";E; and "Exit_status=0"

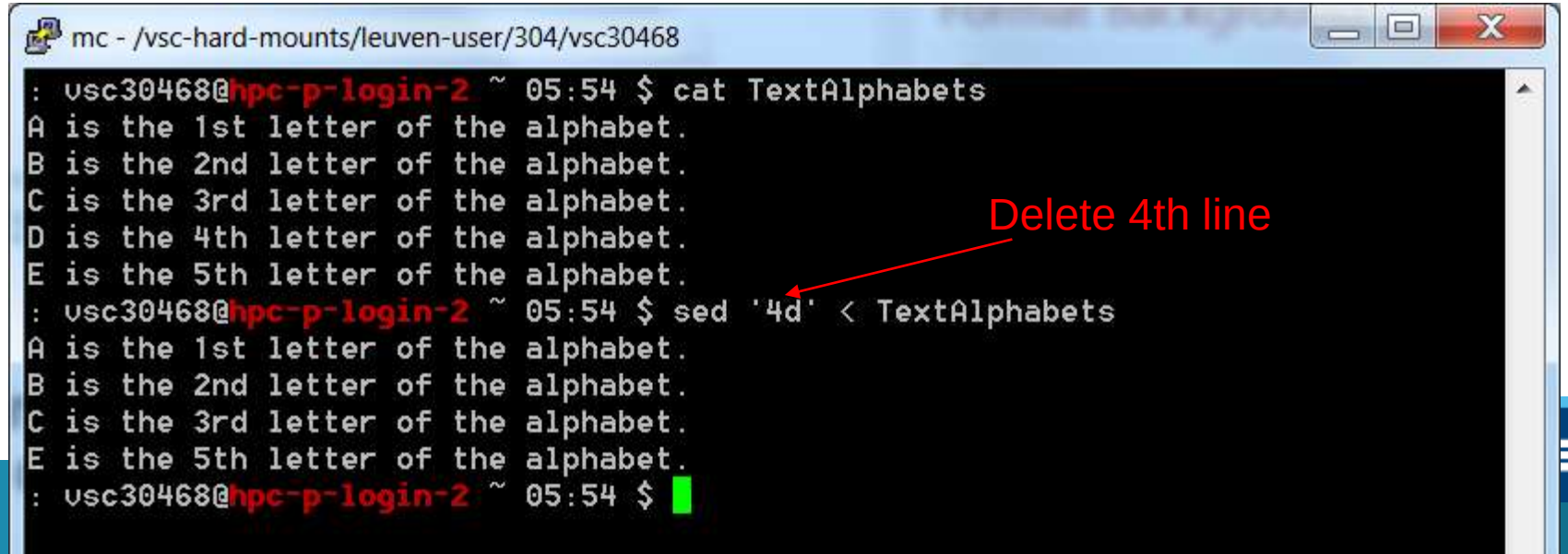
1. `touch `date +%d-%m-%y`.log`
2. `tr [A-C] [a-c] < ex1.fa  > ex1a.fa`
3. `tr '[A-Z]' '[B-ZA]' <ex1.fa`
4. `tr A-Za-z N-ZA-Mn-za-m < r13inp.txt > r13out1.txt`
   `tr A-Za-z N-ZA-Mn-za-m < r13out1.txt > r13out2.txt`
   `diff rot13-input.txt rot13-output2.txt`
5. `cut -d' ' -f5 lp_hpcinfo > hpcinfo.log`
6. `sort hpcinfo.log |uniq > hpcinfo`
   or: `sort –u hpcinfo.log > hpcinfo`
7. `sort lp_hpcinfo -k 5`
8. `grep "Exit_status=0" 20170601`
9. `grep -v ";E;" 20170601`
10. `grep  -E "r3i0n11|r3i0n12" 20170601`
    or `grep  -E "r3i0n1[1-2]" 20170601`
11. `grep  "Exit_status=0" 20170601 |grep ";E;"`

Useful tools

- `Sed` (Stream Editor)
  - Text editing
  - Example: `sed 's/XYZ/xyz/g' filename`
- `Awk` (Alfred Aho, Peter Weinberger, and Brian Kernighan)
  - Pattern scanning and processing
  - Example: `awk '{print $4, $7}' filename`

The ‘sed’ Command/Language

- Filter
  - Like grep, sort, or uniq, it takes input and performs some operation on it to filter the output
- Usage: sed ‘Address Command’
  - Address specifies where to make changes
  - Command specifies what change to make
  - Example:
    - sed ‘4d’ textFile



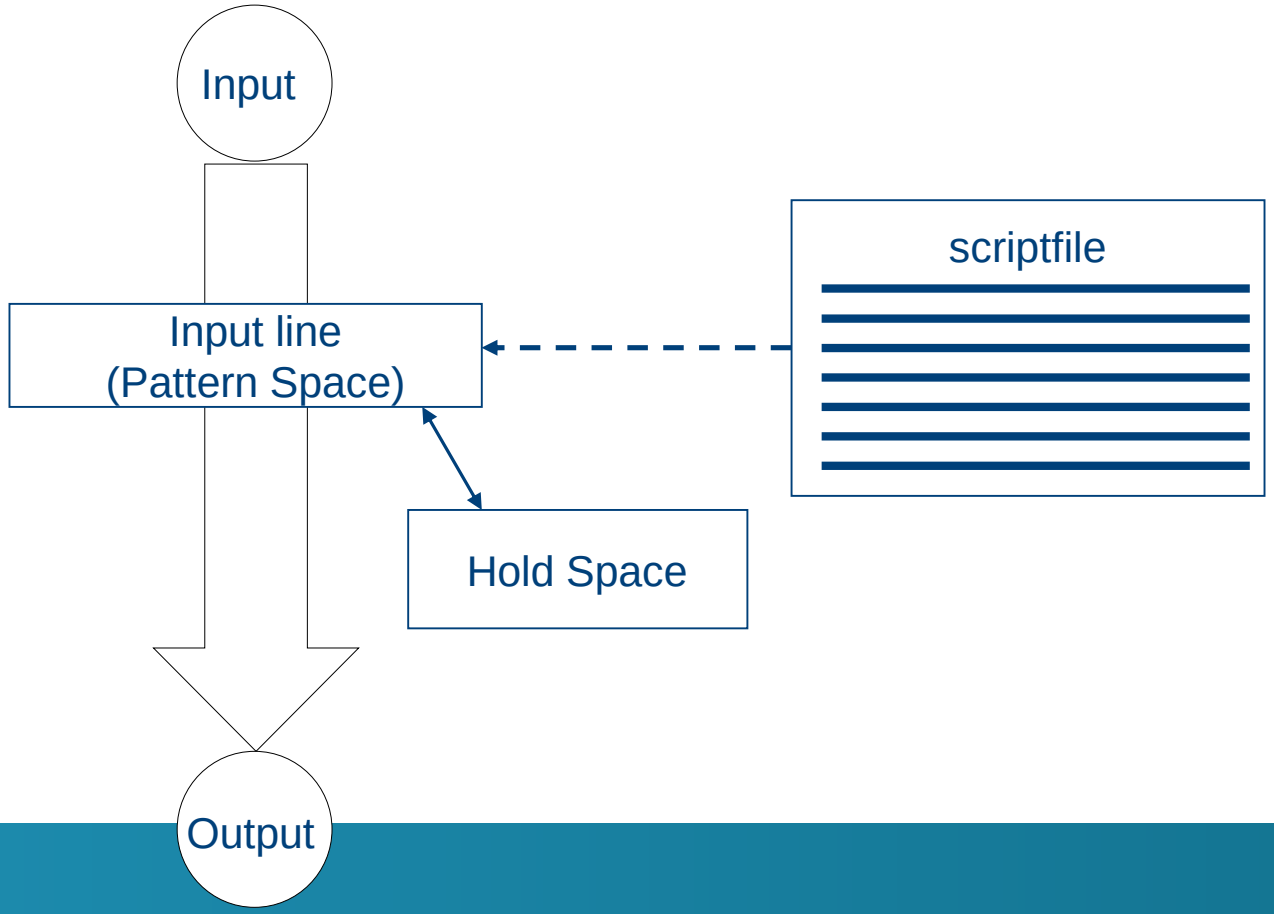
How Does sed Work?

- sed reads line of input
  - line of input is copied into a temporary buffer called pattern space
  - editing commands are applied
    - subsequent commands are applied to line in the pattern space, not the original input line
    - once finished, line is sent to output (unless –n option was used)
  - line is removed from pattern space
- sed reads next line of input, until end of file

Note: input file is unchanged

Pattern and Hold spaces

- Pattern space: Workspace or temporary buffer where a single line of input is held while the editing commands are applied
- Hold space: Secondary temporary buffer for temporary storage only



Address Specification

- Addresses could be line numbers or regular expressions
  - No address – each line
  - One address – only that line
  - Two comma separated addresses – All lines in between
  - ! – All other lines

Some commands Available To sed

- a\
  - Append text
- c\
  - Replace text
- i\
  - Insert text before
- d
  - Delete lines
- s
  - Make substitutions

More sed commands

p - print

r - read

w - write

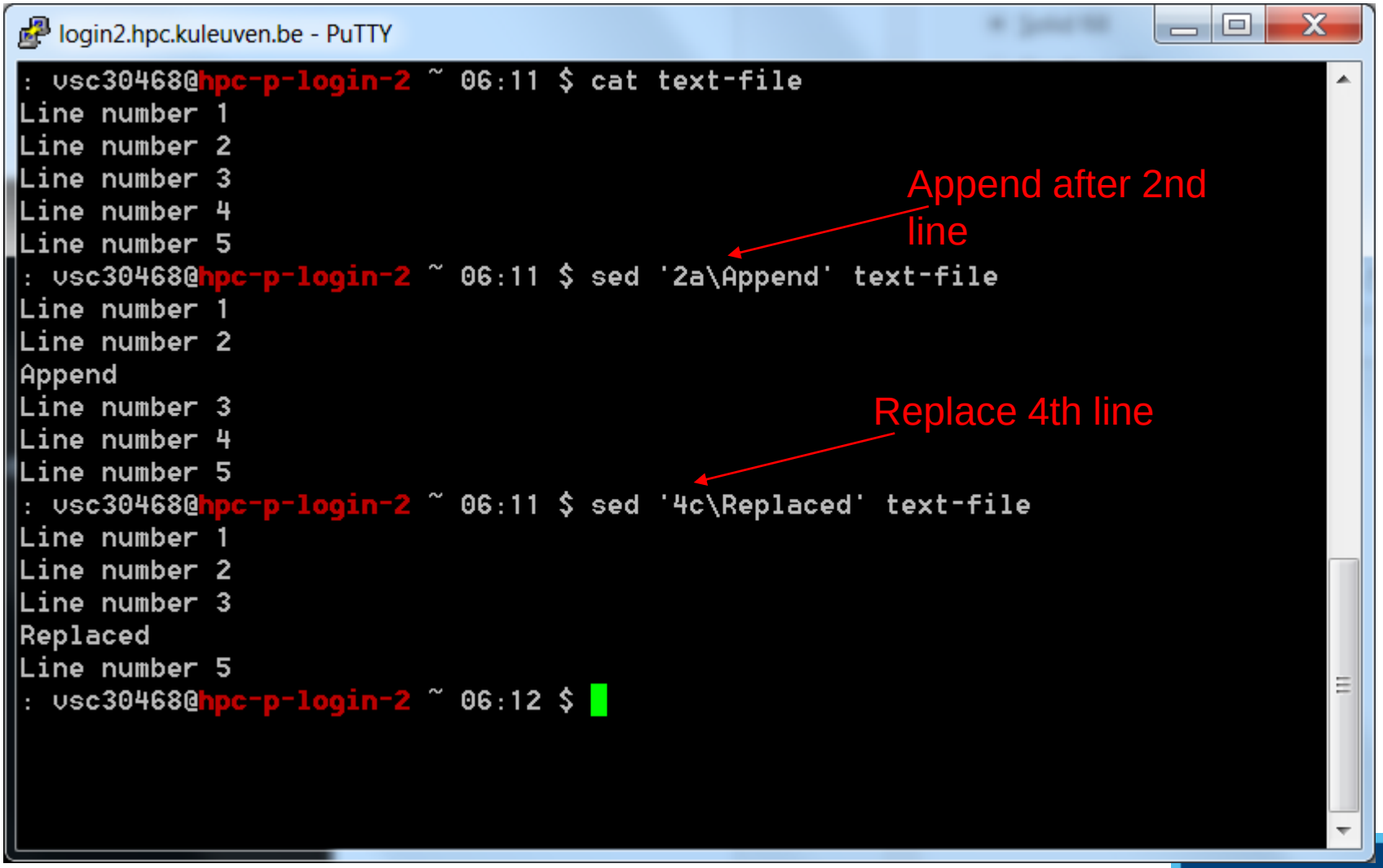
y - transform

= - display line number

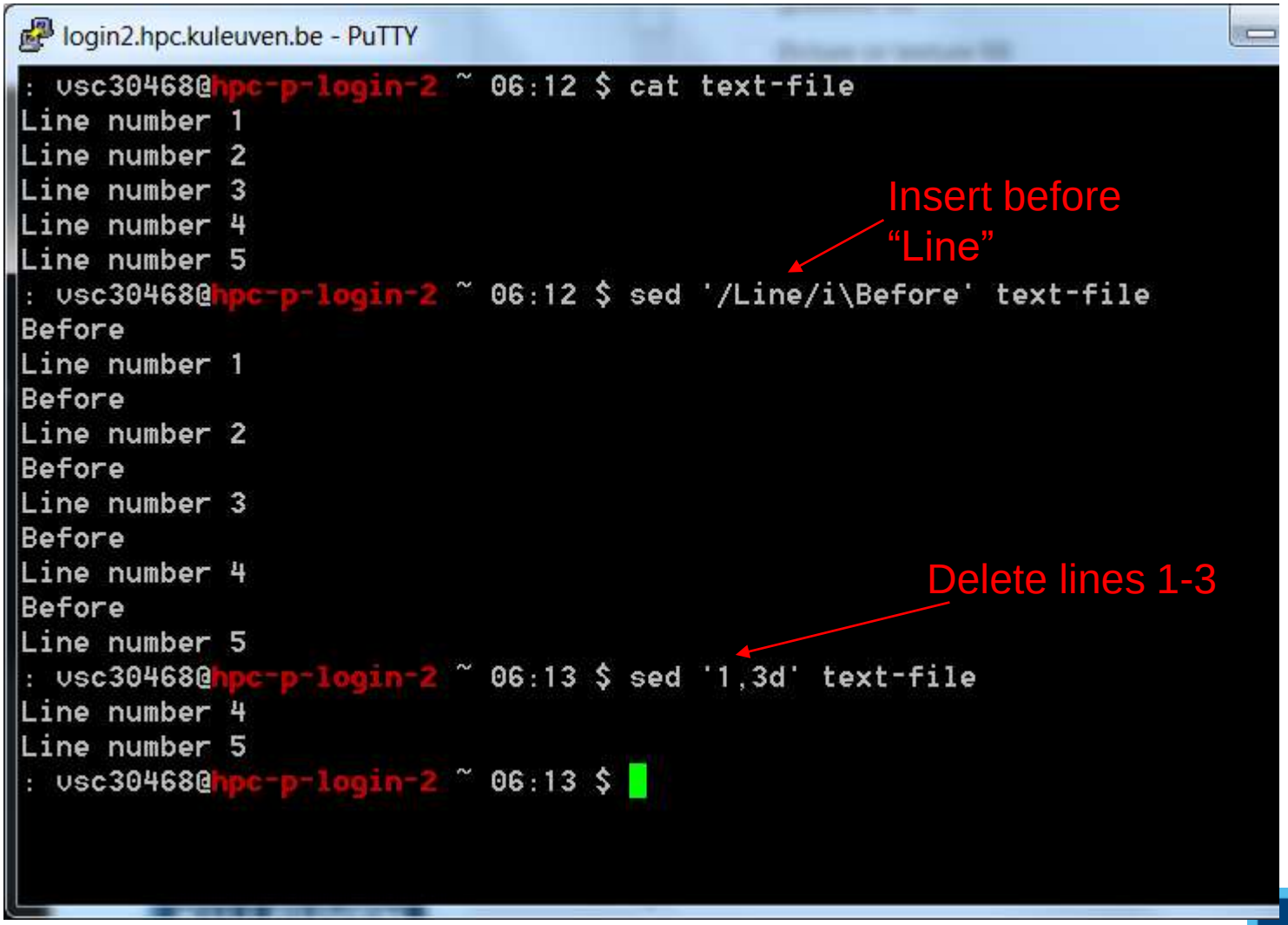
N - append the next line to the current one

q - quit

Examples


login2.hpc.kuleuven.be - PuTTY
: vsc30468@hpc-p-login-2 ~ 06:11 $ cat text-file
Line number 1
Line number 2
Line number 3
Line number 4
Line number 5
Append after 2nd line
: vsc30468@hpc-p-login-2 ~ 06:11 $ sed '2a\Append' text-file
Line number 1
Line number 2
Append
Line number 3
Line number 4
Line number 5
Replace 4th line
: vsc30468@hpc-p-login-2 ~ 06:11 $ sed '4c\Replaced' text-file
Line number 1
Line number 2
Line number 3
Replaced
Line number 5
: vsc30468@hpc-p-login-2 ~ 06:12 $

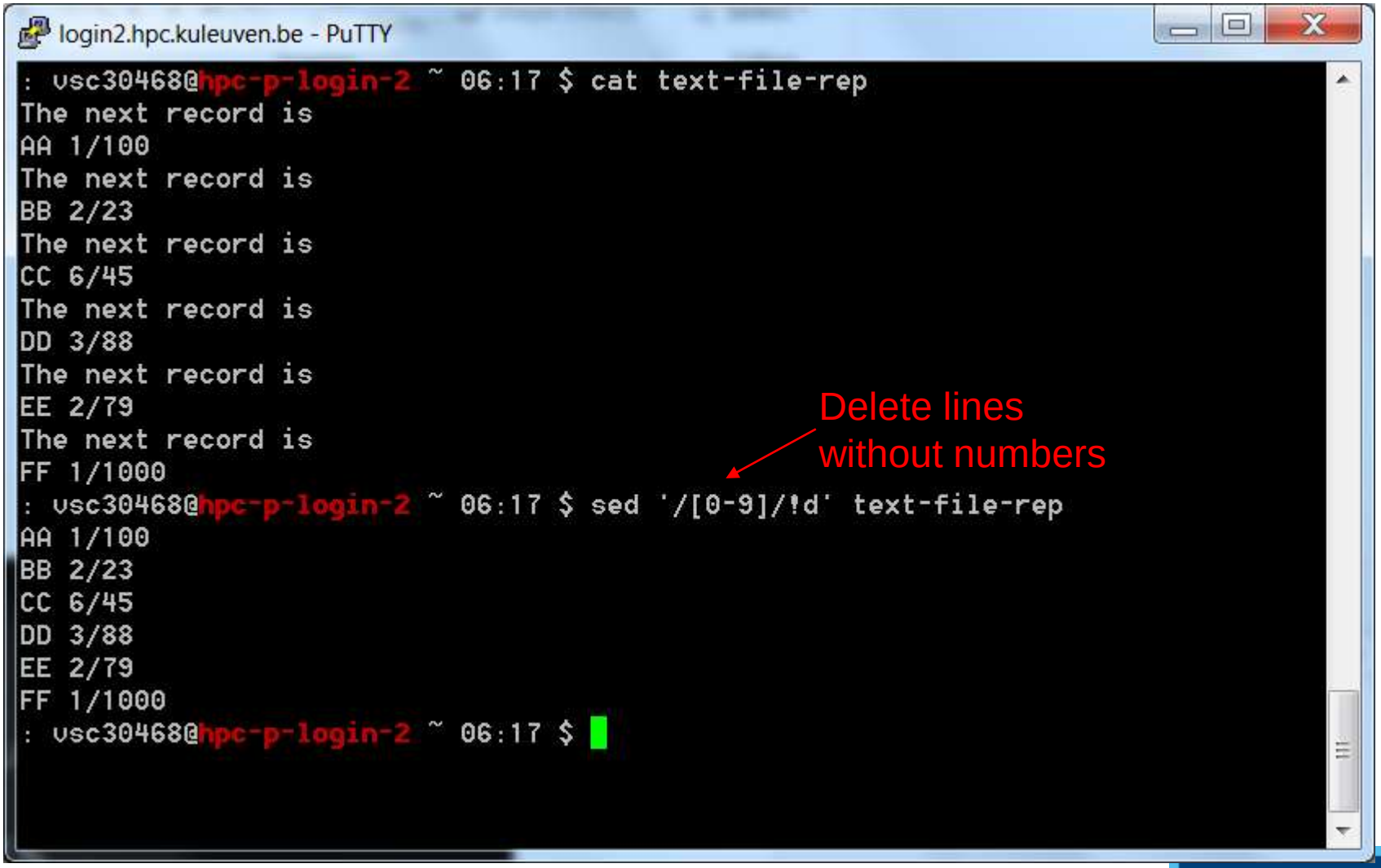
More Examples



When Would You Want To Use sed?

- sed works on streams, so it is perfect to be placed in the middle of a pipe in order to change the output from one format to another
- Example:
  - If a program always prints out 4 lines of junk for every good line, sed can be used to weed out the junk

Example


login2.hpc.kuleuven.be - PuTTY
: vsc30468@hpc-p-login-2 ~ 06:17 $ cat text-file-rep
The next record is
AA 1/100
The next record is
BB 2/23
The next record is
CC 6/45
The next record is
DD 3/88
The next record is
EE 2/79
The next record is
FF 1/1000
Delete lines
without numbers
: vsc30468@hpc-p-login-2 ~ 06:17 $ sed '/[0-9]/!d' text-file-rep
AA 1/100
BB 2/23
CC 6/45
DD 3/88
EE 2/79
FF 1/1000
: vsc30468@hpc-p-login-2 ~ 06:17 $

Sed Usage

- Edit files too large for interactive editing
- Edit any size files where editing sequence is too complicated to type in interactive mode
- Perform “multiple global” editing functions efficiently in one pass through the input
- Edit multiples files automatically
- Good tool for writing conversion programs

Conceptual overview

- A script is read which contains a list of editing commands
  - Can be specified in a file or as an argument
- Before any editing is done, all editing commands are compiled into a form to be more efficient during the execution phase.
- All editing commands in a sed script are applied in order to each input line.
- If a command changes the input, subsequent command address will be applied to the current (modified) line in the pattern space, not the original input line.
- The original input file is unchanged (sed is a filter), and the results are sent to standard output (but can be redirected to a file).

sed Syntax

- sed [-n] [-e] [‘command’] [file…]
- sed [-n] [-f scriptfile] [file…]
- -n - only print lines specified with the print command (or the ‘p’ flag of the substitute (‘s’) command)
- -f scriptfile - next argument is a filename containing editing commands
- -e command - the next argument is an editing command rather than a filename, useful if multiple commands are specified

sed syntax

```
$ sed -e 'address command' input_file
```

(a) Inline Script

```
$ sed -f script.sed input_file
```

(b) Script File

Commands

- command is a single letter

- Example:

Deletion: d

[address1][,address2]d

- Delete the addressed line(s) from the pattern space; line(s) not passed to standard output.
- A new line of input is read and editing resumes with the first command of the script.

Address and Command Examples: delete

d	deletes all lines
6d	deletes line 6
/^$/d	deletes all blank lines
1,10d	deletes lines 1 through 10
1,/^$/d	deletes from line 1 through the first blank line
/^$/,$d	deletes from the first blank line through the last line of the file
/^$/,10d	deletes from the first blank line through line 10
/^ya*y/,/[0-9]$/d	deletes from the first line that begins with yay, yaay, yaaay, etc. through the first line that ends with a digit

Multiple Commands

- Braces {} can be used to apply multiple commands to an address

```
[address][,address]{
    command1
    command2
    command3
}
```

- The opening brace must be the last character on a line
- The closing brace must be on a line by itself
- Make sure there are no spaces following the braces
- Alternatively, use “;” after each command:

```
[address][,address]{command1; command2; command3; }
```

Print

- The print command (p) can be used to force the pattern space to be output, useful if the -n option has been specified
- Syntax:

[address1[,address2]]p

- Note: if the -n or #n option has not been specified, p will cause the line to be output twice!
- Examples:

1,5p will display lines 1 through 5

/^$/,$p will display the lines from the first blank line through the last line of the file

Substitute

- Syntax:

[address(es)]s/pattern/replacement/[flags]

- pattern - search pattern
- replacement - replacement string for pattern
- flags - optionally any of the following

n	a number from 1 to 512 indicating which occurrence of pattern should be replaced
g	global, replace all occurrences of pattern in pattern space
p	print contents of pattern space

Substitute Examples

s/Puff Daddy/P. Diddy/

Substitute P. Diddy for the first occurrence of Puff Daddy in pattern space

s/Tom/Fred/2

Substitutes Fred for the second occurrence of Tom in the pattern space

s/wood/plastic/p

Substitutes plastic for the first occurrence of wood and outputs (prints) pattern space

Append, Insert, and Change

- Syntax for these commands is a little strange because they must be specified on multiple lines
- append

[address]a\

text

- insert

[address]i\

text

- change

[address(es)]c\

text

- append/insert for single lines only, not range

Append and Insert

- Append places text after the current line in pattern space

- Insert places text before the current line in pattern space

  - each of these commands requires a \ following it
  - text must begin on the next line.
  - if text begins with whitespace, sed will discard it unless you start the line with a \

Using !

- If an address is followed by an exclamation point (!), the associated command is applied to all lines that don't match the address or address range
- Examples:

1,5!d delete all lines except 1 through 5

/black/!s/cow/horse/

substitute "horse" for "cow" on all lines except those that contained "black"

- e.g.

"The brown cow" -> "The brown horse"

"The black cow" -> "The black cow"

Regular Expressions: use with sed

Metacharacter	Description/Matches…
.	Any one character, except new line
*****	Zero or more of preceding character
^	A character at beginning of line
$	A character at end of line
\char	Escape the meaning of *char* following it
[ ]	Any one of the enclosed characters
\(\)	Tags matched characters to be used later
x\{m\}	Repetition of character x, m times
\<	Beginning of word
\>	End of word

Sed – more info

- https://www.gnu.org/software/sed/manual/sed.html
- https://www.tutorialspoint.com/sed/
- http://www.grymoire.com/Unix/Sed.html
- ….

Hands-on 2

KU LEUVEN

1. Create the file test10.log containing

A is the first letter

B is the second letter

C is the third letter

D is the fourth letter

E is the fifth letter

F is the sixth letter

G is the seventh letter

H is the eighth letter

i is the ninth letter

j is the tenth letter

2. Make a new file named 1.sed which has only lines 1,2,3,4,5 and 6 from the original input file
3. Make a new file which has only lines 2 and 4 removed from the original file
4. Make a new file which has the line “The end” added to the end of the file
5. Make a new file which has the line “The end” added after the line containing “tenth”
6. Make a new file which has 8th line changed into “Change”
7. Make a new file which has the 1st line changed into “A is the 1st letter”
8. Make a new file which has only lines containing at least one upper case character

1. `nano test10.log`
2. `sed '1,6!d' test10.log > 1.sed`

   or

   `sed -n 1,6p test10.log > 1.sed`
3. `sed -e '2d;4d' test10.log > 2.sed`
4. `sed '$ a\ The end' test10.log > 3.sed`
5. `sed '/tenth/a\ The end' test10.log > 4.sed`
6. `sed 's/H is the eighth letter/Change/' test10.log > 5.sed`

   or

   `sed -e '/eighth/i\Change' test10.log|sed 9d > 5.sed`

   or

   `sed 8c\Change test10.log > 5.sed`
7. `sed '/first/c\A is the 1st letter' test10.log > 6.sed`

   or

   `sed s/first/1st/ test10.log > 6.sed`
8. `sed '/[A-Z]/!d' test10.log > 7.sed`

awk

- Answers the question:
  - What do I do if I want to search for a pattern and actually use it?
- Combination of grep and other commands
  - Searches for some pattern or condition and performs some command on it
- Complete programming language
  - Looks a lot like C syntactically
  - Variables are declared bash style

When Would You Want To Use awk?

- Whenever you want to search for some pattern and perform some action
- Example: I want to go through and calculate the average score on the Midterm

Advantages of Awk

- awk is an interpreted language so you can avoid the usually lengthy edit-compile-test-debug cycle of software development .
- Can be used for rapid prototyping.
- The awk language is very useful for producing reports from large amounts of raw data, such as summarizing information from the output of other utility programs like ls.

Combining Expression Options

- ||
  - Or
- &&
  - And
- \(\)
  - Grouping

Action Commands

- -print
  - Simply prints out the name of the file that was found
  - Most common action
- -exec
  - Executes a command
- -ok
  - Executes a command, but prompts the user first

Sed vs. awk

- **sed** is a pattern-action language, like **awk**
- awk processes fields while sed only processes lines
- sed +
  - regular expressions
  - fast
  - concise
- sed –
  - hard to remember text from one line to another
  - not possible to go backward in the file
  - no way to do forward references like /..../+1
  - no facilities to manipulate numbers
  - cumbersome syntax
- awk +
  - convenient numeric processing
  - variables and control flow in the actions
  - convenient way of accessing fields within lines
  - flexible printing
  - C-like syntax

A few basic things about awk

- awk reads from a file or from its standard input, and outputs to its standard output.
- awk recognizes the concepts of "file", "record" and "field".
- A file consists of records, which by default are the lines of the file. One line becomes one record.
- awk operates on one record at a time.
- A record consists of fields, which by default are separated by any number of spaces or tabs.
- Field number 1 is accessed with $1, field 2 with $2, and so forth. $0 refers to the whole record.

Program Structure in Awk

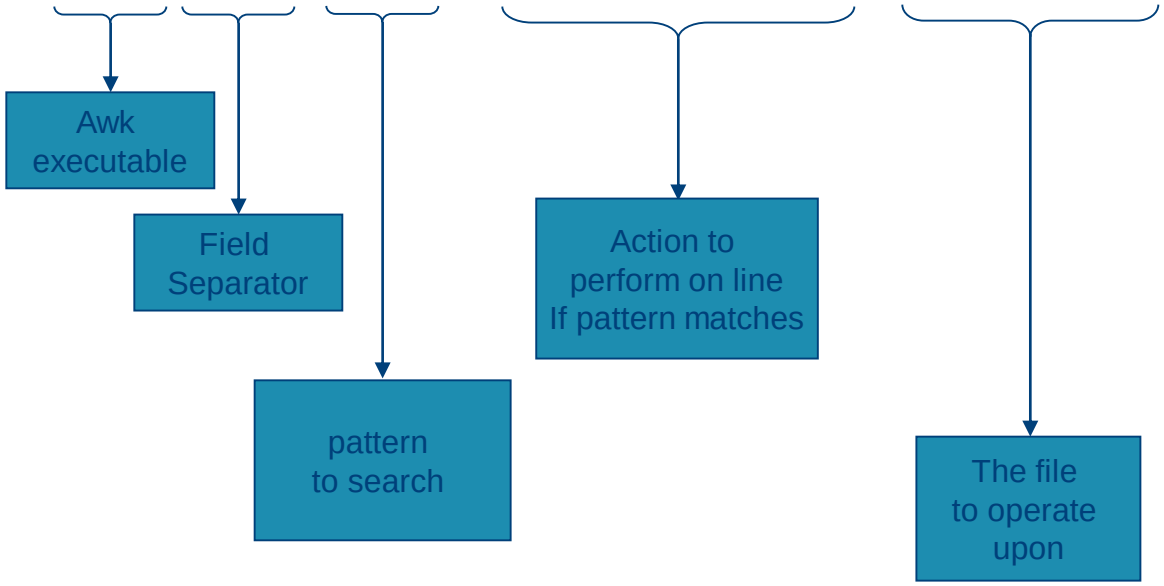
- An awk program is a sequence of statements of the form:

```
pattern   { action }
pattern   { action }
...
```

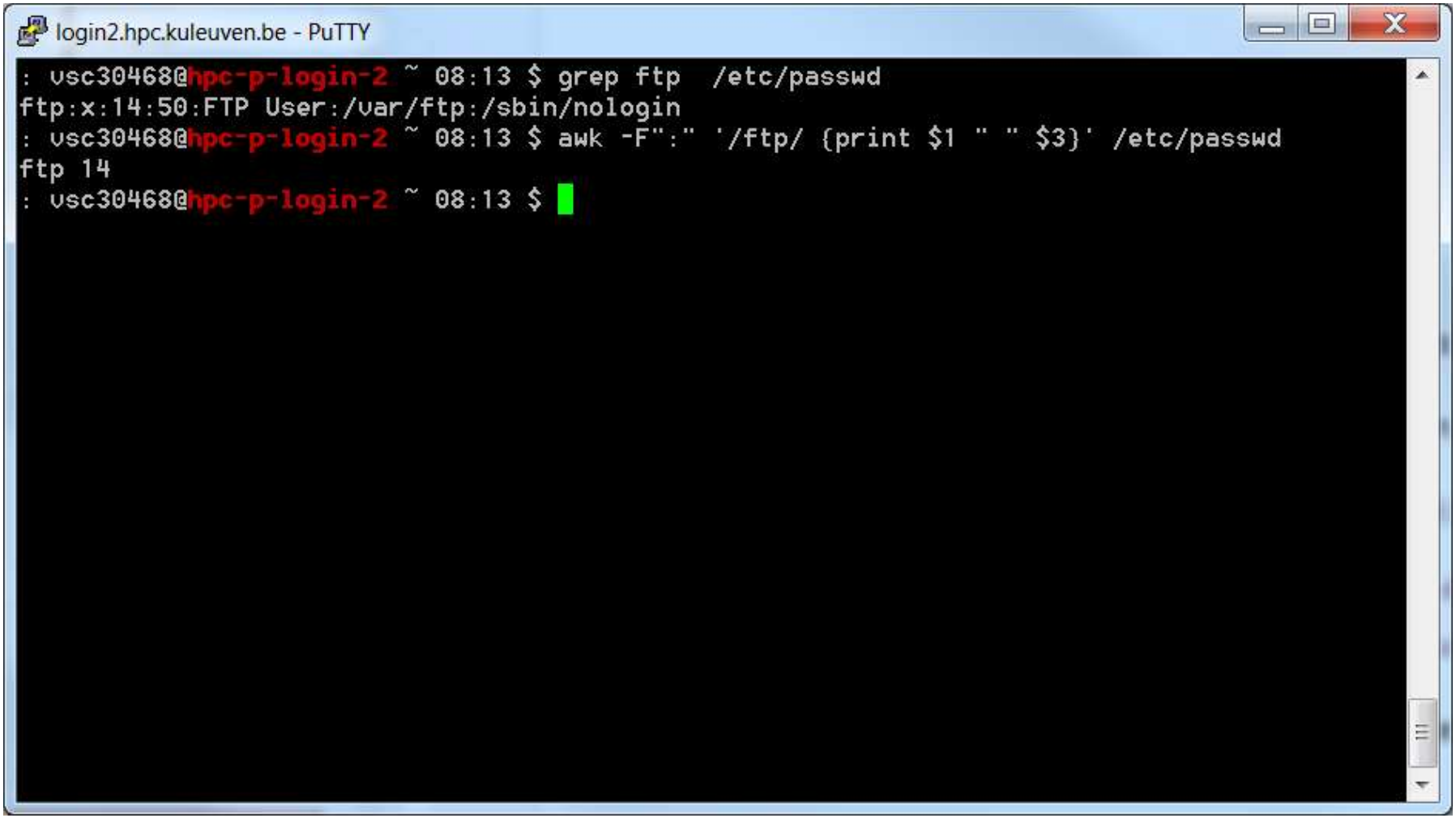
- pattern in front of an action acts as a selector that determines whether the action is to be executed.
- Patterns can be : regular expressions, arithmetic relational expressions, string-valued expressions, and arbitrary boolean combinations of these.
- action is a sequence of action statements terminated by newlines or semicolons.

A simple example

```
$ awk –F”:” ‘/ftp/ {print $1 “ “ $3}’ /etc/passwd
```



A simple example (cont..)



Running awk programs

There are four ways in which we can run awk programs

- One-shot: Running a short throw-away awk program.

  ```
  $ awk 'program' input-file1 input-file2
  ```

  ... where *program* consists of a series of patterns and actions.

- Read Terminal: Using no input files (input from terminal instead).

  ```
  $ awk 'program' <ENTER>
  <input lines>
  <input lines>
  ctrl-d
  ```

- Long: Putting permanent awk programs in files.

  ```
  $ awk -f source-file input-file1 input-file2 ...
  ```

- Executable: self-contained awk scripts, using the `#!' script mechanism.

  Self-contained awk scripts are useful when you want to write a program which users can invoke without their having to know that the program is written in awk.

Basic awk Syntax

- **`awk [options] 'script' file(s)`**
- **`awk [options] –f scriptfile file(s)`**

Options:

-F to change input field separator

-f to name script file

Basic awk Program

- consists of patterns & actions:

```
pattern {action}
```

  - if pattern is missing, action is applied to all lines
  - if action is missing, the matched line is printed
  - must have either pattern or action

Example:

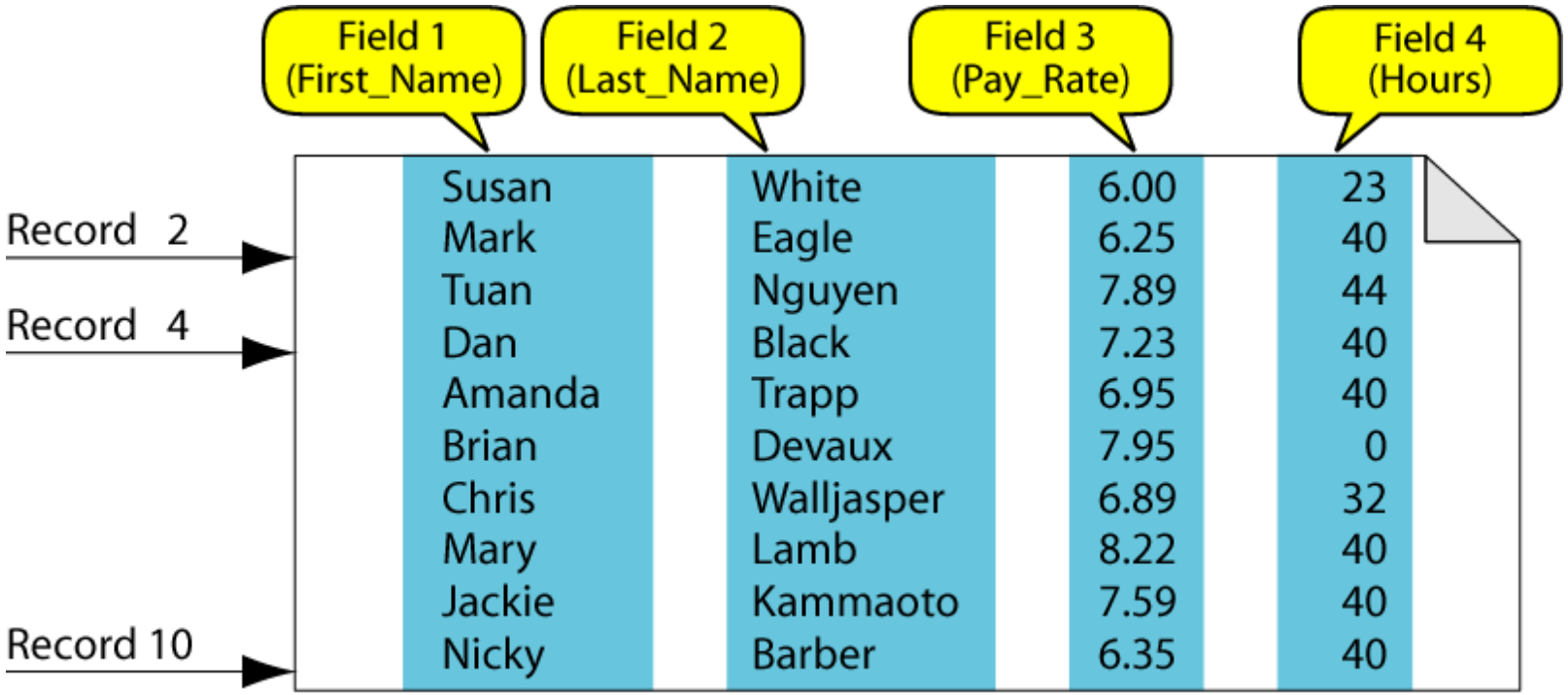
```
awk '/for/' testfile
```

  - prints all lines containing string “for” in testfile

Basic Terminology: input file

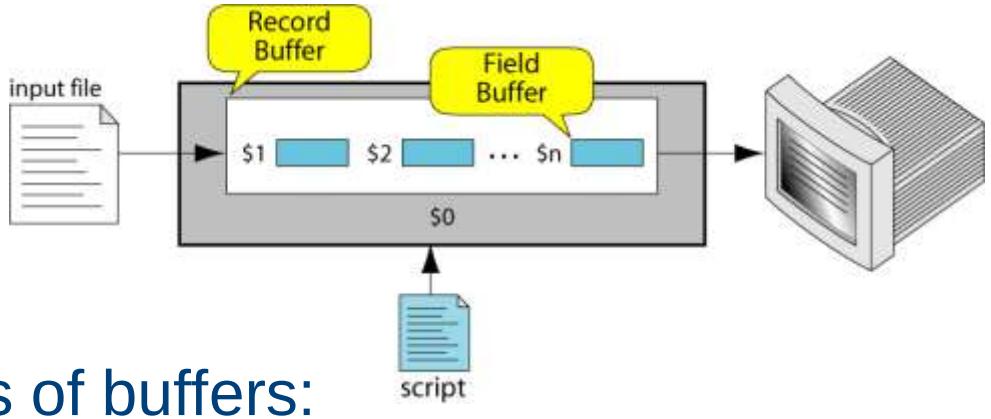
- A field is a unit of data in a line
- Each field is separated from the other fields by the field separator
  - default field separator is whitespace
- A record is the collection of fields in a line
- A data file is made up of records

Example Input File



A file with 10 records, each with four fields

Buffers



- awk supports two types of buffers: record and field

- field buffer:
  - one for each fields in the current record.
  - names: $1, $2, …

- record buffer :
  - $0 holds the entire record

Some System Variables

FS	Field separator (default=whitespace)
RS	Record separator (default=\n)
NF	Number of fields in current record
NR	Number of the current record
OFS	Output field separator (default=space)
ORS	Output record separator (default=\n)
FILENAME	Current filename

Example: Records and Fields

```
$ cat emps
Tom Jones          4424     5/12/66 543354
Mary Adams         5346     11/4/63 28765
Sally Chang        1654     7/22/54 650000
Billy Black        1683     9/23/44 336500

$ awk '{print NR, $0}' emps
1 Tom Jones        4424     5/12/66 543354
2 Mary Adams       5346     11/4/63 28765
3 Sally Chang      1654     7/22/54 650000
4 Billy Black      1683     9/23/44 336500
```

Example: Space as Field Separator

```
$ cat emps
Tom Jones          4424     5/12/66 543354
Mary Adams         5346     11/4/63 28765
Sally Chang        1654     7/22/54 650000
Billy Black        1683     9/23/44 336500

$ awk '{print NR, $1, $2, $5}' emps
1 Tom Jones 543354
2 Mary Adams 28765
3 Sally Chang 650000
4 Billy Black 336500
```

Example: Colon as Field Separator

```
$ cat em2
Tom Jones:4424:5/12/66:543354
Mary Adams:5346:11/4/63:28765
Sally Chang:1654:7/22/54:650000
Billy Black:1683:9/23/44:336500

$ awk -F: '/Jones/{print $1, $2}' em2
Tom Jones 4424
```

Advanced awk features

- Awk borrows a lot from the C language.
- The if loop, for loop and while loop have the same constructs as in C.
- Awk's variables are stored internally as strings.

```
eg. x = "1.01"
    x = x + 1
    print x
```

The above will print the value 2.01

- Comparison operators in awk are : `"==", "<", ">", "<=", ">=", "!=", "~"` and `"!~"`.
- "**~**" and "**!~**" operators mean "matches" and "does not match".

KU LEUVEN

Awk Examples

```
$ awk '{ print $0 }' /etc/passwd
```

Prints all the lines in /etc/passwd

```
$ awk -F":" '{ print "username: " $1 "\t\tuid:" $3" }' /etc/passwd
```

Prints the 1st and 3rd fields of each line in /etc/passwd. The fields are separated by ":"

```
$ awk –f script1.awk /etc/passwd
```

script1.awk

```
BEGIN{ x=0 }# The BEGIN block is executed before processing the file
/^$/ { x=x+1 } # For every null line increment the count
END { print "I found " x " blank lines. :)" } #Executed at the end
```

The above script calculates the number of null lines. Note that BEGIN and END are special patterns.

Awk examples (cont..)

```
$ awk 'BEGIN { RS = "/" } ; { print $0 }' file1.txt
```

RS is the record separator (default is \n). In this example the RS is modified to “/” and then the file is processed. So awk will distinguish between records by “/” character.

```
$ awk '$1 ~ /foo/ { print $0 }' file.txt
```

The pattern will print out all records from file file.txt whose first fields contain the string “foo”.

```
$ awk '{ print $(2*2) }' file.txt
```

In the above example the field number is an expression. So awk will print the 4th fields of all the records.

Awk examples (cont..)

```
$ awk '{ $3 = $2 - 10; print $2, $3 }' inventory-shipped
```

This example will subtract the second field of each record by 10 and store it in the third field.

```
$ awk 'BEGIN { FS = "," } ; { print $2 }' file.txt
```

FS is the field separator in awk. In the above example we are asking awk to separate the fields by “,” instead of default “ “.

```
$ awk 'BEGIN { OFS = ";"; ORS = "\n\n" }
            { print $1, $2 }'  file1.txt
```

OFS is the Output field Separator, ORS is Output record separator. This prints the first and second fields of each input record separated by a semicolon, with a blank line added after each line.

Awk examples

Consider that we have the following input in a file called

```
john 85 92 78 94 88
andrea 89 90 75 90 86
jasper 84 88 80 92 84
```

The following awk script will find the average

```
# average five grades
{ total = $2 + $3 + $4 + $5 + $6
  avg = total / 5
  print $1, avg }
```

```
$ awk -f grades.awk grades
```

Awk examples (cont..)

```
$ awk 'BEGIN { OFMT = "%d"  # print numbers as integers
               print 17.23 }‘
```

This will print 17. OFMT is the output format specifier.

```
$ awk –f mailerr.awk
  { report = "mail bug-system"
    print "Awk script failed:", $0 | report
    print "at record number", FNR, "of", FILENAME  | report
    close(report)
  }
```

This script opens a pipe to the mail command and prints output into the pipe. When the pipe is closed the mail is sent. Awk assumes that whatever comes after the “|” symbol is a command and creates a process for it.

Expression Pattern types

- match
  - entire input record

    regular expression enclosed by ‘/’s
  - explicit pattern-matching expressions

    ~ (match), !~ (not match)

- expression operators
  - arithmetic
  - relational
  - logical

Example: match input record

```
% cat employees2
Tom Jones:4424:5/12/66:543354
Mary Adams:5346:11/4/63:28765
Sally Chang:1654:7/22/54:650000
Billy Black:1683:9/23/44:336500

% awk -F: '/00$/' employees2
Sally Chang:1654:7/22/54:650000
Billy Black:1683:9/23/44:336500
```

Arithmetic Operators

Operator	Meaning	Example
+	Add	x + y
-	Subtract	x – y
*	Multiply	x * y
/	Divide	x / y
%	Modulus	x % y
^	Exponential	x ^ y

Example:

```
% awk '$3 * $4 > 500 {print $0}' file
```

Relational Operators

Operator	Meaning	Example
<	Less than	x < y
<=	Less than or equal	x < = y
==	Equal to	x == y
!=	Not equal to	x != y
>	Greater than	x > y
>=	Greater than or equal to	x > = y
~	Matched by reg exp	x ~ /y/
!~	Not matched by req exp	x !~ /y/

Logical Operators

Operator	Meaning	Example
&&	Logical AND	a && b
||	Logical OR	a || b
!	NOT	! a

Examples:

```
$ awk '($2 > 5) && ($2 <= 15)
                        {print $0}' file
$ awk '$3 == 100 || $4 > 50' file
```

awk variables

- A user can define any number of variables within an awk script
- The variables can be numbers, strings, or arrays
- Variable names start with a letter, followed by letters, digits, and underscore
- Variables come into existence the first time they are referenced; therefore, they do not need to be declared before use
- All variables are initially created as strings and initialized to a null string “”

awk Variables

Format:

```
variable = expression
```

Examples:

```
$ awk '$1 ~ /Tom/
          {wage = $3 * $4; print wage}'
       filename
$ awk '$4 == "CA"
          {$4 = "California"; print $0}'
       filename
```

awk assignment operators

=	assign result of right-hand-side expression to left-hand-side variable
++	Add 1 to variable
--	Subtract 1 from variable
+=	Assign result of addition
-=	Assign result of subtraction
*=	Assign result of multiplication
/=	Assign result of division
%=	Assign result of modulo
^=	Assign result of exponentiation

Output Statements

`print`

print easy and simple output

`printf`

print formatted (similar to C printf)

`sprintf`

format string (similar to C sprintf)

Function: print

- Writes to standard output
- Output is terminated by ORS
  - default ORS is newline
- If called with no parameter, it will print $0
- Printed parameters are separated by OFS,
  - default OFS is blank
- Print control characters are allowed:
  - \n \f \a \t \\ …

Redirecting print output

- Print output goes to standard output

unless redirected via:

> “file”

>> “file”

| “command”

- will open file or command only once
- subsequent redirections append to already open stream

print example

```
% awk '{print}' grades
john 85 92 78 94 88
andrea 89 90 75 90 86

% awk '{print $0}' grades
john 85 92 78 94 88
andrea 89 90 75 90 86

% awk '{print($0)}' grades
john 85 92 78 94 88
andrea 89 90 75 90 86
```

print Example

```
% awk '{print $1, $2}' grades
john 85
andrea 89

% awk '{print $1 "," $2}' grades
john,85
andrea,89
```

print Example

```
% awk '{OFS="-";print $1 , $2}' grades
john-85
andrea-89

% awk '{print $1 "," $2}' grades
john,85
andrea,89
```

print Example

```
$ awk '{print $1 , $2 > "file"}' grades

$ cat file
john 85
andrea 89
jasper 84
```

print Example

```
$ awk '{print $1,$2 | "sort"}' grades
andrea 89
jasper 84
john 85

$ awk '{print $1,$2 | "sort -k 2"}' grades
jasper 84
john 85
andrea 89
```

printf: Formatting output

Syntax:

```
printf(format-string, var1, var2, …)
```

- works like C printf
- each format specifier in “format-string” requires argument of matching type

Format specifiers

%d, %i	decimal integer
%c	single character
%s	string of characters
%f	floating point number
%o	octal number
%x	hexadecimal number
%e	scientific floating point notation
%%	the letter “%”

Format specifier examples

Given: x = 'A', y = 15, z = 2.3, and $1 = Bob Smith	
Printf Format Specifier	**What it Does**
%c	***printf("The character is %c \n", x)*** **output: The character is A**
%d	***printf("The boy is %d years old \n", y)*** **output: The boy is 15 years old**
%s	***printf("My name is %s \n", $1)*** **output: My name is Bob Smith**
%f	***printf("z is %5.3f \n", z)*** **output: z is 2.300**

Format specifier modifiers

- between “%” and letter

  %10s

  %7d

  %10.4f

  %-20s

- meaning:
  - width of field, field is printed right justified
  - precision: number of digits after decimal point
  - “-” will left justify

sprintf: Formatting text

Syntax:

```
sprintf(format-string, var1, var2, …)
```

- Works like printf, but does not produce output
- Instead it returns formatted string

Example:

```
{
  text = sprintf("1: %d – 2: %d", $1,
  $2)
  print text
}
```

awk builtin functions

`tolower(string)`

- returns a copy of string, with each upper-case character converted to lower-case. Nonalphabetic characters are left unchanged.

  Example: tolower("MiXeD cAsE 123")

  returns "mixed case 123"

`toupper(string)`

- returns a copy of string, with each lower-case character converted to upper-case.

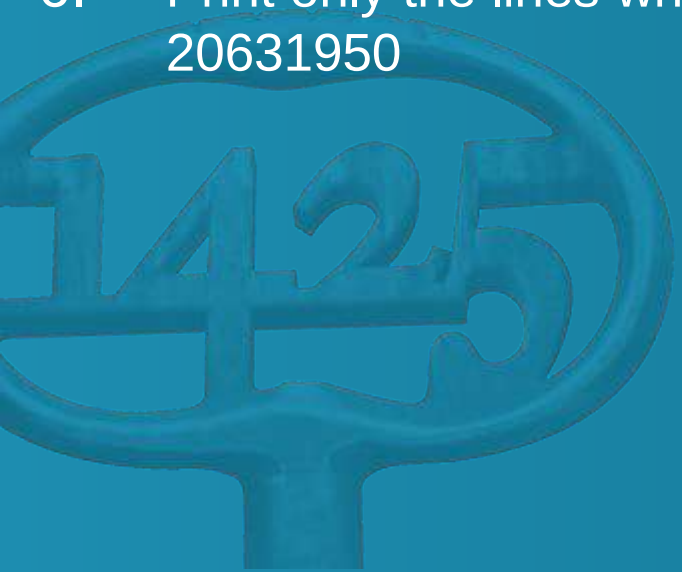
awk references

- https://www.gnu.org/software/gawk/manual/gawk.html
- http://www.grymoire.com/Unix/Awk.html
- https://www.tutorialspoint.com/awk/
- https://www.gnu.org/software/gawk/manual/gawk.pdf
- Sed and Awk 2nd Edition (O'reilly)
  http://linux.iingen.unam.mx/pub/Documentacion/Shell-Bash/OReilly%20-%20Sed%20&%20Awk%202nd%20Edition.pdf

Hands-on 3

KU LEUVEN

1. Download the file https://sites.google.com/site/magselwakuleuven/downloads/lp_hpcinfo and print the number of its lines with awk
2. Download the file https://sites.google.com/site/magselwakuleuven/downloads/lp_sys and print only the lines that contain vsc30468
3. From lp_sys print only the lines that contain both vsc30468 and 2045
4. From lp_sys print only column 3 and 4 of the lines from 30th line on
5. From lp_sys print only column 7 and the last column.
6. Print only the lines where the 3rd column has values higher than 20631950



KU LEUVEN

```
1.  awk 'END{print NR}' lp_hpcinfo
2.  awk '/vsc30468/ {print $0}' lp_sys
3.  awk '/vsc30468/ && /2045/ {print $0}' lp_sys
4.  awk 'NR > 30 { print $3 ',' $4}'  lp_sys
    awk 'NR > 30 { print $3 ", " $4}'  lp_sys
5.  awk '{print $3 ',' $NF}' lp_sys
6.  awk '$3 > 20631950' lp_sys
```

KU LEUVEN

Gnuplot

- Versatile visualization tool
  - Command driven, interactive function plotting program
- Designed for mathematics visualization
  - Student programmers looking to visualize classroom concepts
  - Open source code
- Multiple Platforms
  - Unix/Linux, Macs, lots of others
  - Windows version has a menu-driven interface
- Copyrighted but freely distributable
- Originally developed by Colin Kelley and Thomas Williams in 1986 to plot functions and data files on a variety of terminals.

Gnuplot - advantages

- Quality 2-D/3-D plots of functions and data
- Small set of commands - easy to learn
- Very transferable
  - Multiple platforms
  - Text file formats for input and command files
- Can be automated (allows it to be run by other programs)
- Can manipulate data with mathematical functions
- Existing support community
- Free! Free! Free!
  - www.gnuplot.info (gnuplot Central)

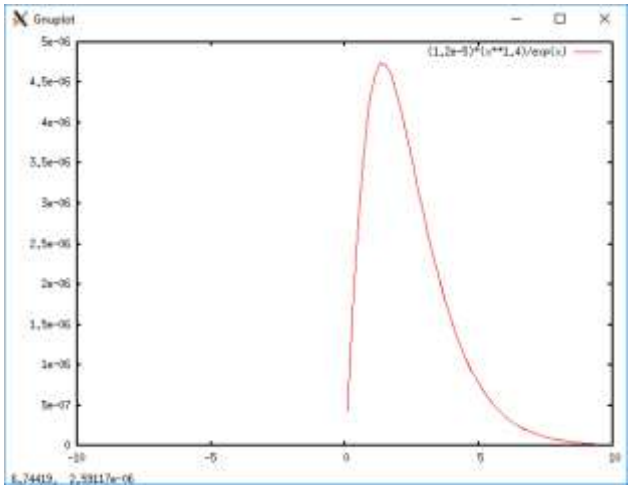
Gnuplot - disadvantages

- Limited types of plots
- Contour plot process a bit difficult
- ‘Programming’ style assumed
- Limited set of styles for lines and points
- Limited text abilities for titles and labels
  - Greek letters and symbols difficult/not possible
  - Formatted math formulas difficult/not possible
- Updates/development depends on volunteers

Gnuplot

- Console Window
  - Command line prompt gnuplot>
  - Menu selections for Windows version
- Plotting Window
- Help
  - ?
  - help <topic>
- Lines beginning with “#” are comments lines and are ignored.

Gnuplot – function plots



- 2-D variable is x

```
gnuplot> plot sin(x)
```

- Use replot to display a second function

```
gnuplot> replot cos(x)
```

- Math functions in fortran style

```
gnuplot> plot (1.2e-5)*(x**1.4)/exp(x)
```

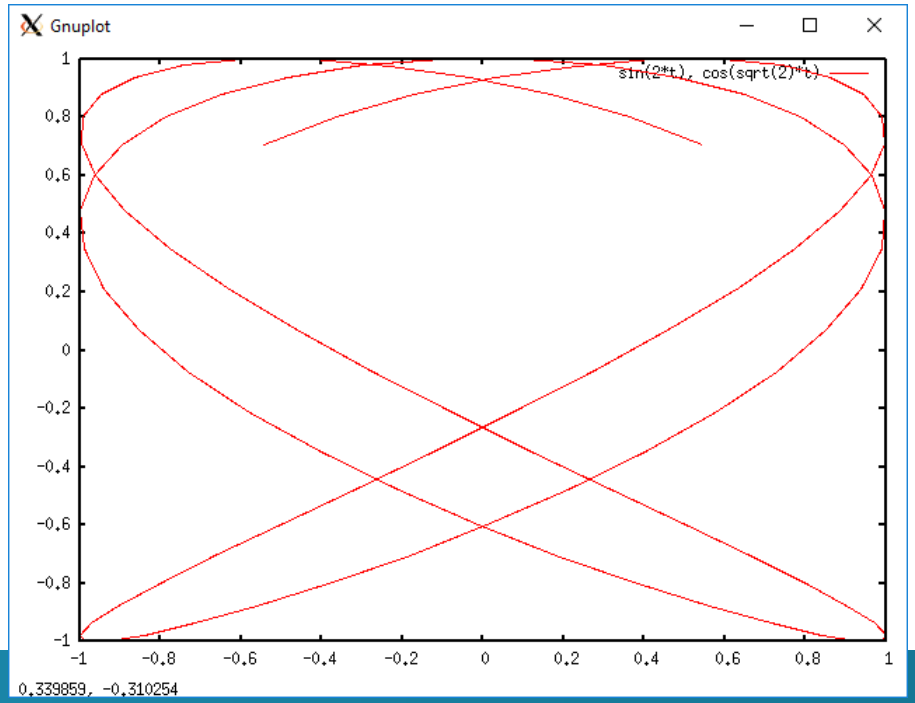
- Many functions (use '`help functions`' to list)
  - e.g. abs() sinh() log() rand() acos() sqrt()

Gnuplot – parametric plots

```
gnuplot> set parametric
```

- 2-D “dummy” variable is t

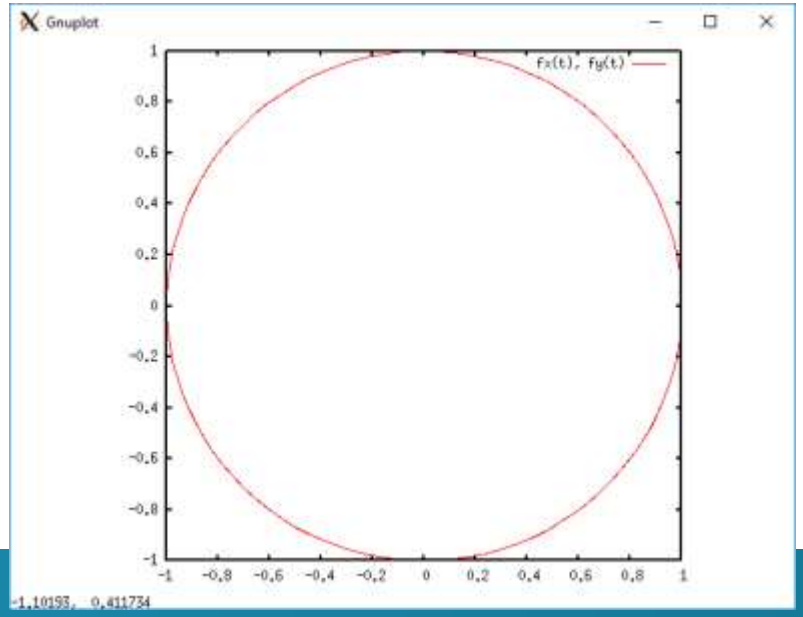
```
gnuplot> plot sin(2*t), cos(sqrt(2)*t)
gnuplot> unset parametric
```



KU LEUVEN

Gnuplot – parametric plots

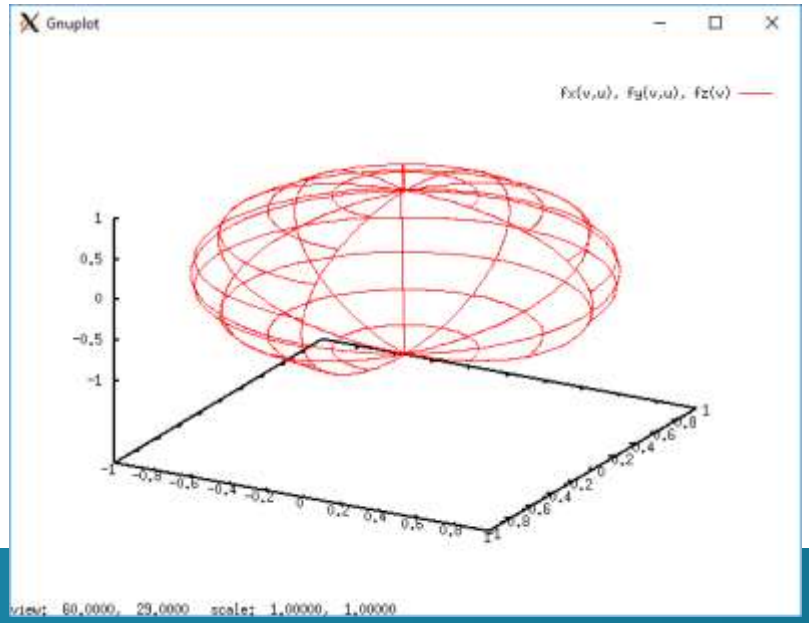
```
gnuplot> set size square
gnuplot> r=1
gnuplot> fx(t) = r*cos(t)
gnuplot> fy(t) = r*sin(t)
gnuplot> plot fx(t),fy(t)
```



KU LEUVEN

Gnuplot – 3D plots

```
gnuplot> set parametric
gnuplot> r=1
gnuplot> fx(v,u) = r*cos(v)*cos(u)
gnuplot> fy(v,u) = r*cos(v)*sin(u)
gnuplot> fz(v)   = r*sin(v)
gnuplot> splot fx(v,u),fy(v,u),fz(v)
```

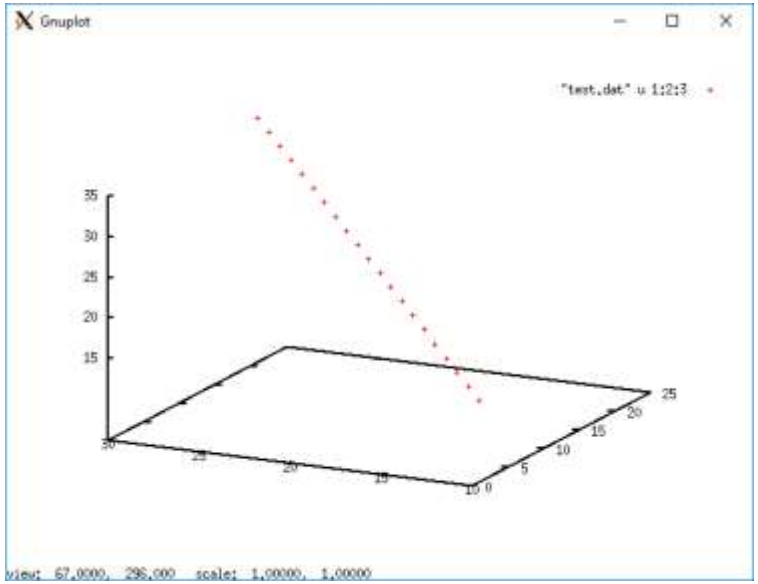


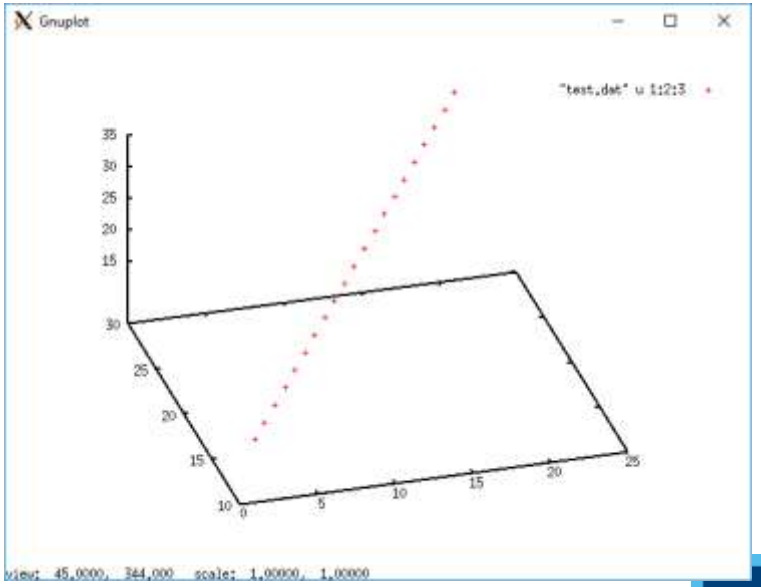
KU LEUVEN

Gnuplot – 3D plots

```
gnuplot> splot "test.dat" u 1:2:3
```

- For 3d graphs splot, the view and scaling of the graph can be changed with mouse buttons

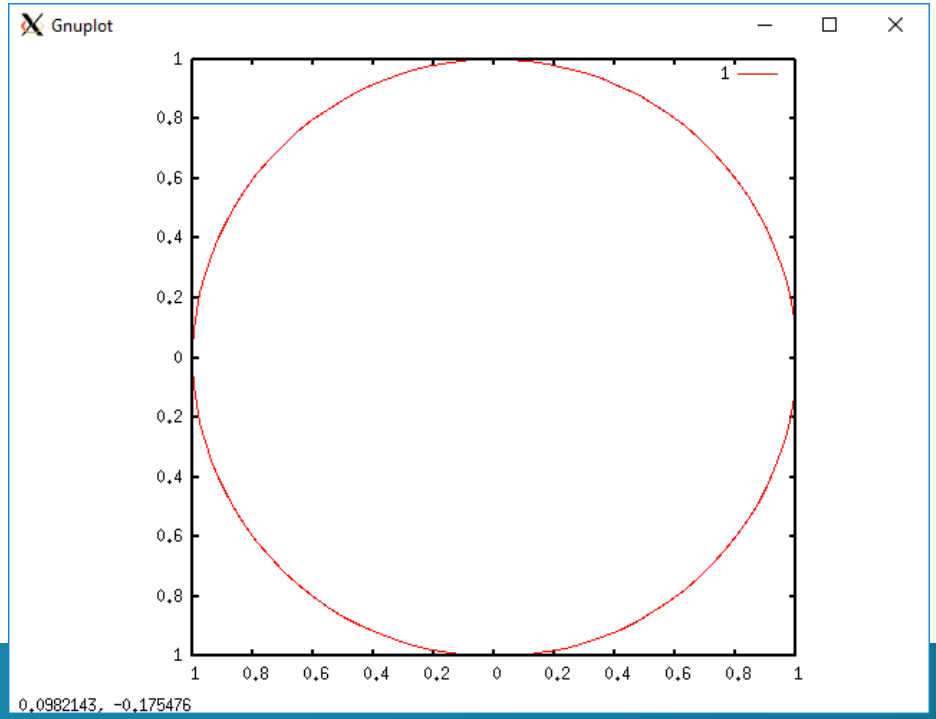




KU LEUVEN

Gnuplot – polar plots

```
gnuplot> set polar
gnuplot> set size ratio -1
gnuplot> plot 1
gnuplot> unset polar
```

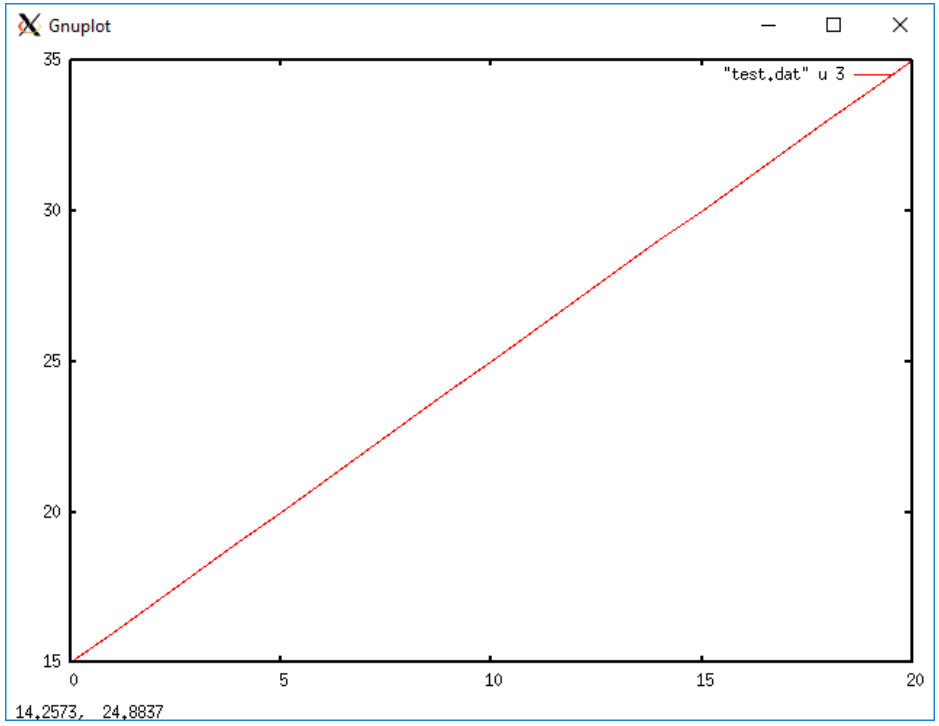


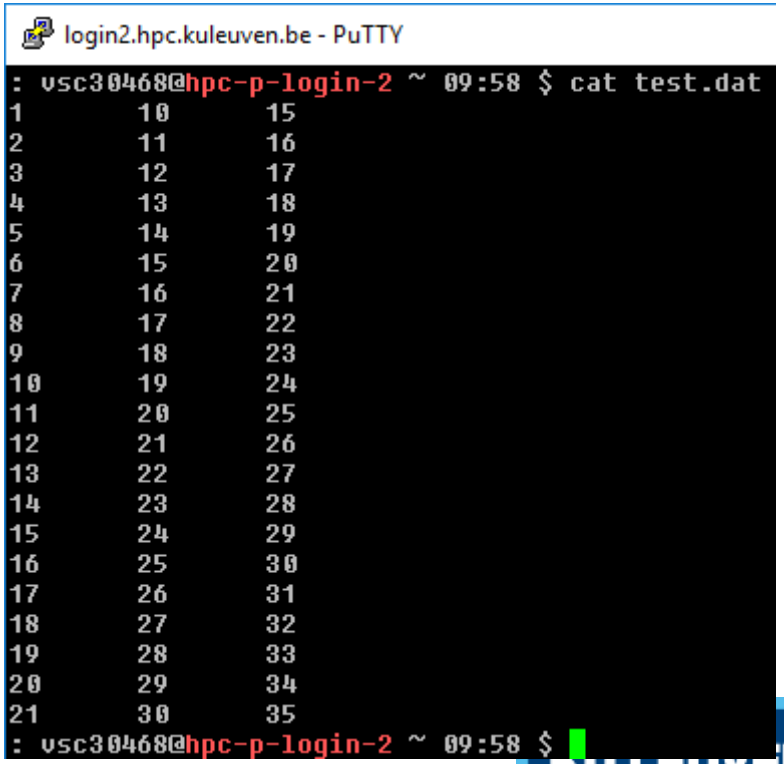
KU LEUVEN

Gnuplot - data plots

- Single channel

```
gnuplot> plot "test.dat" u 3 w l
gnuplot> plot "test.dat" using 3 with lines
```

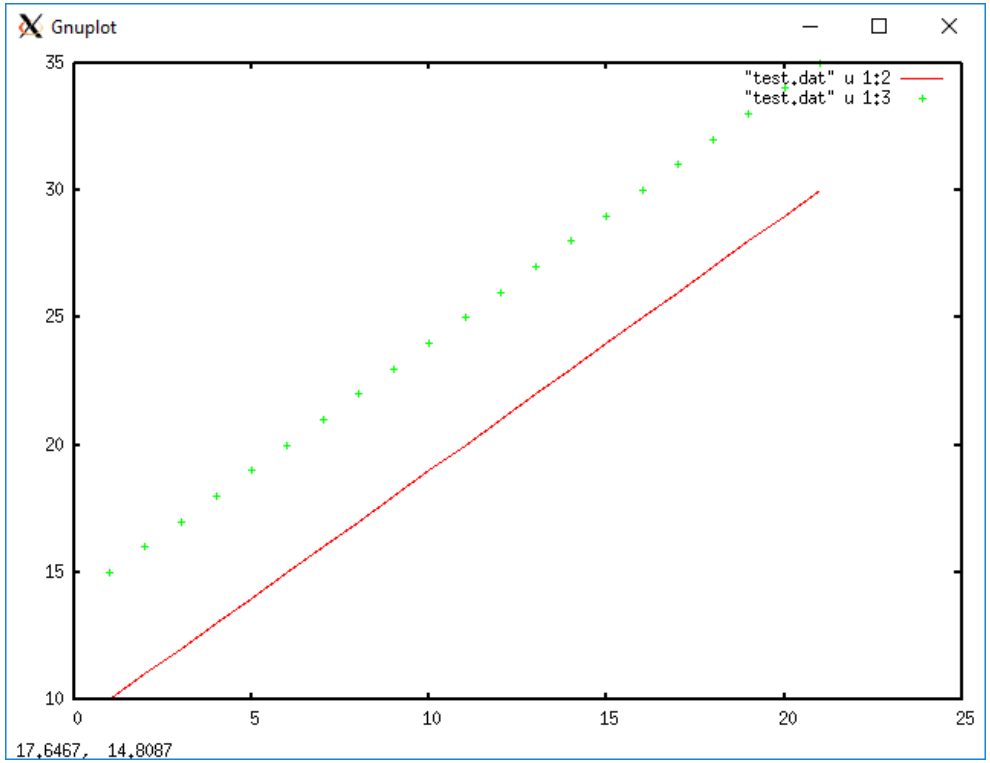


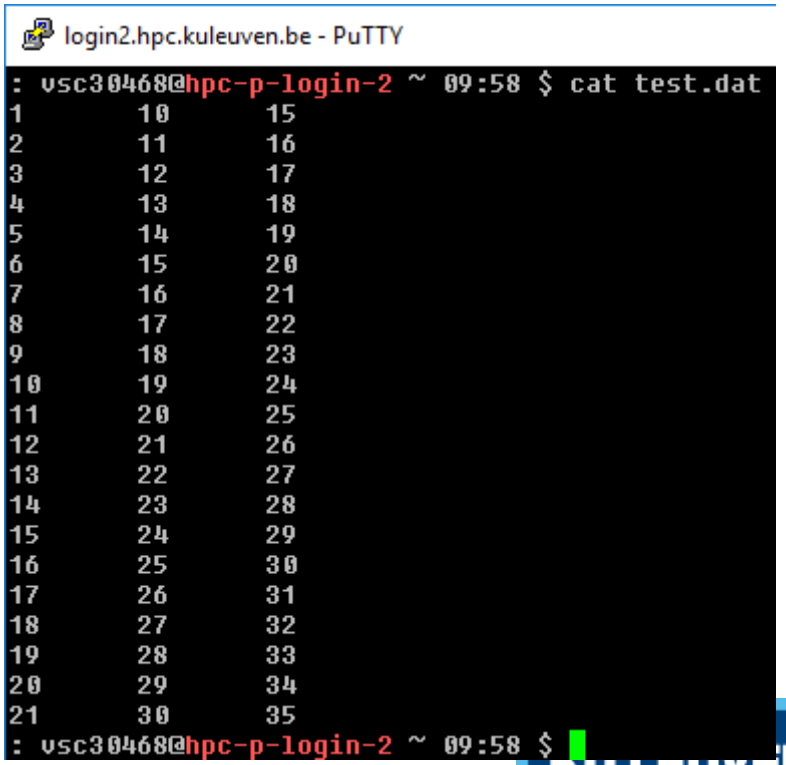


Gnuplot - data plots

- Cross plots

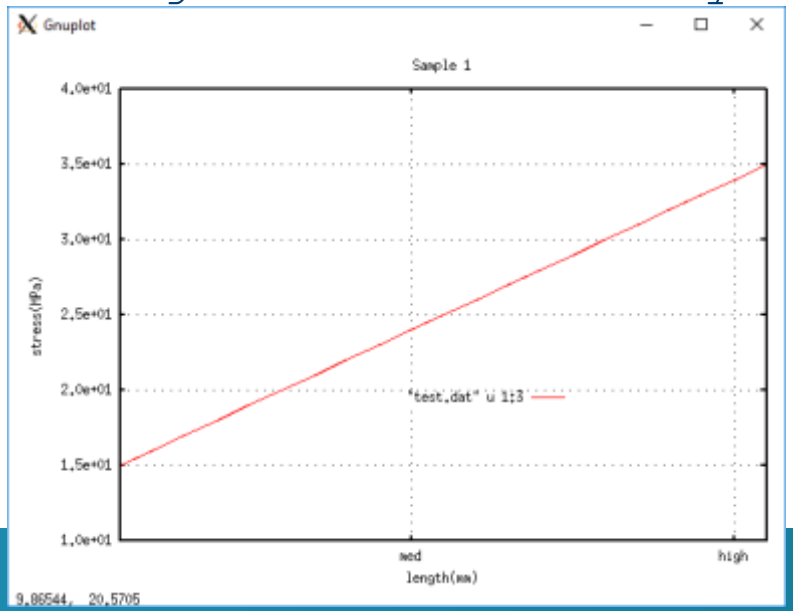
```
gnuplot> plot "test.dat" u 1:2 w l, "test.dat" u 1:3 w p
```





Gnuplot - settings

- `gnuplot> plot [:] [10:40]  "test.dat" u 1:3 w l`
- `gnuplot> set xlabel 'length(mm)'; set ylabel 'stress(MPa)' -1,0; replot`
- `gnuplot>  set title "Sample 1"; set key 15,20; replot`
- `gnuplot>  set xtics ('low' 0, 'med' 10,'high' 20); set grid; set format y '%.1e'; replot`



Plot with the new settings

KU LEUVEN

Gnuplot - settings

- Relative Graph size: `> set size 1,1`
- Function sampling pts: `> set samples 1000`
- Polar Coordinates: `> set polar`
- Contour plots: `> set contour base`
- Perspective: `> set view 90,0`
- Angle Units: `> set angles degrees`
- Style: `> set data style lines`

Gnuplot – basic functions

command line entry	description of action
exit or quit	Quits gnuplot
help or ? <topic>	help or help on a specific topic
set key or unset key	Adds or removes legend from display
set xrange [a:b]	Controls the range of values on the x axis
plot sin(x**2-1) with points <value>	plots sin(x**2 − 1) - lots of in-built math functions - <value> between 1 and 8 denotes different pointstyles
plot sin(x) with lines <value>	<value> between 1 and 8 denotes different linestyles
plot 'test.dat' u 1:2 t 'test' w l 1,'test.dat' u 1:3 t 'test' w p 2	plots multiple sets of data on same graph: Curve of cols 1 and 2 from <test.dat> plus points from cols 1 and 3 of <test.dat>. The t is an abbr. of title for labelling in the legend
set term postscript set term png	Output in postscript/png file rather than to the display
set output 'filename.eps'	Sets up an output file for output to go to
replot	redisplays plot with new settings
set term x11	sets output back to the screen, rather than a file

Gnuplot - settings

- place or hide key
  ```
  set key top center, unset key
  ```
- set a title
  ```
  set title "the title"
  ```
- define axis labels
  ```
  set xlabel "x [pc]", set ylabel "y [pc]"
  ```
- change the number format
  ```
  set format x "%10.3f"
  ```
- plot an arrow
  ```
  set arrow from 0.5,0 to 0.5,1
  ```
- define a label
  ```
  set label "rarefaction wave" at 0.5,0
  ```
- set border style
  ```
  set border lw 3
  ```

Gnuplot - settings

- color, width and shape of lines/points

  `linetype / lt, pointtype / pt,`

  `linewidth / lw, pointsize / ps`

- logscale

  `[un]set logscale [xy],`

- select zoom

  `set xrange [0:10] ... manually selected range of x-axis,`

  `set yrange [*:*] ... select zoom of y-axis automatically,`

  `set autoscale ... select zoom of any axis automatically`

Gnuplot

- Show command gives current settings

```
gnuplot> show xlabel
```

- Reset recalls default settings
- Save records current settings and plot statement

```
gnuplot> save '/home/mag/test.gp'
```

- Load executes settings from a file

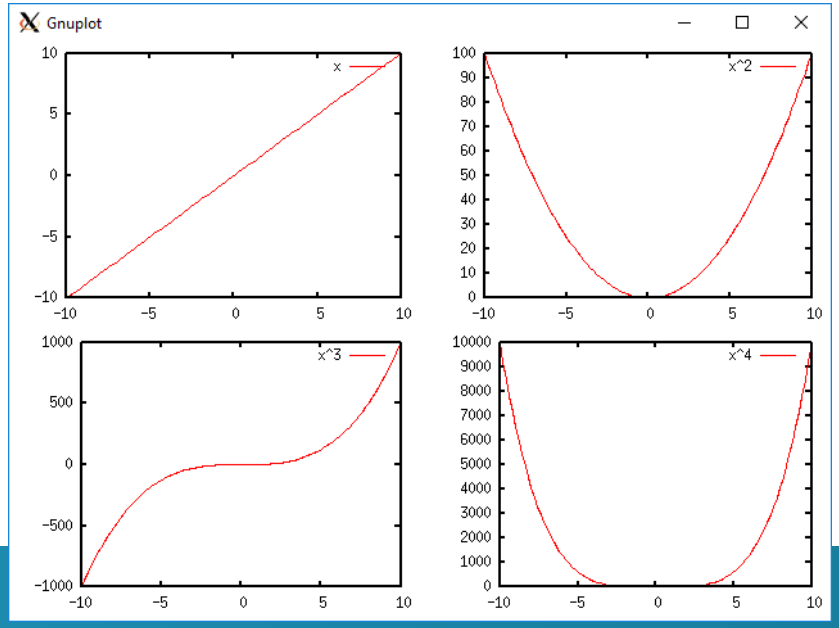
```
gnuplot> load '/home/mag/test.gp'
```

- Help is extremely useful in determining set syntax

```
gnuplot> help set
```

Gnuplot - multiplot

- stack several plot commands
  `set multiplot`
- scale the plot
  `set size`
- place the plot
  `set origin`
- leave multiplot mode
  `unset multiplot`

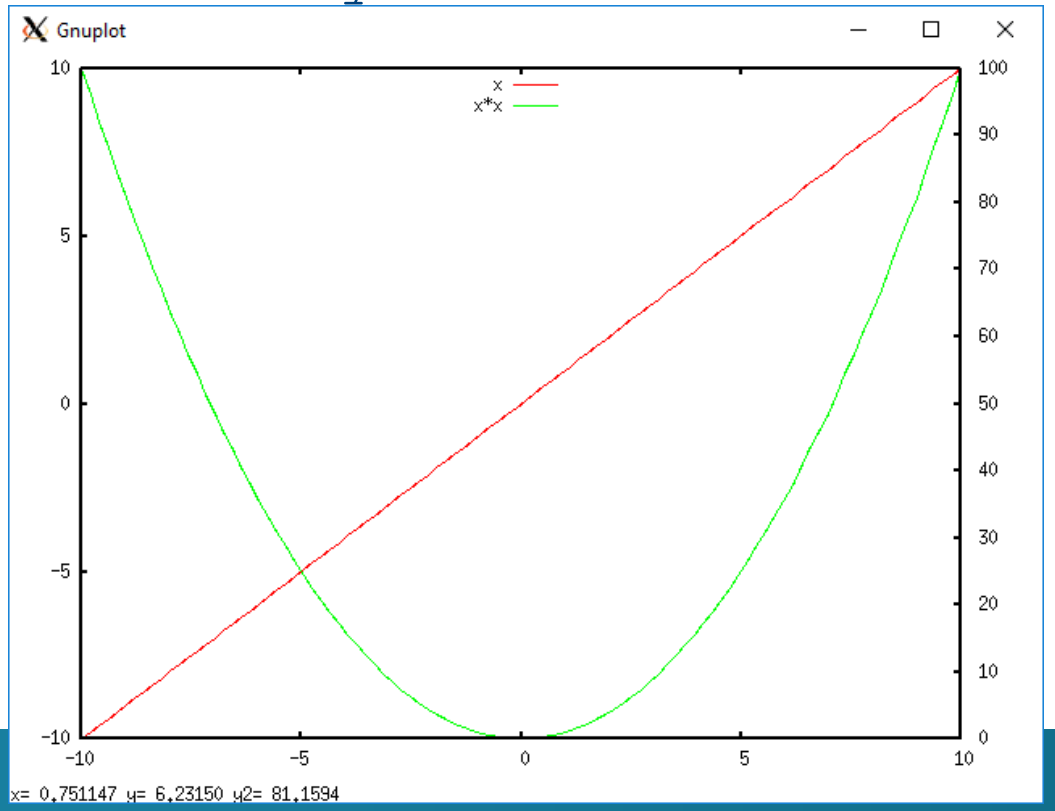



```
set multiplot
set origin 0 ,0
set size 0.5 ,0.5
plot x*x*x t 'x^3'
set origin 0 ,0.5
plot x t 'x'
set origin 0.5 ,0.5
plot x*x t 'x^2'
set origin 0.5 ,0
plot x*x*x*x t 'x^4'
```

KU LEUVEN

Gnuplot - multiple graphs - y1 and y2 axis

```
set ytics nomirror
set y2tics 0, 10
set key top center
plot x axis x1y1, x*x axis x1y2
```



Gnuplot - animations

- If you use gnuplot 4.6 you can use a do loop:
- If you want to store the snapshots use:

```
filename(n) = sprintf("file_%d", n)
plot for [i=1:10] filename(i) using 1:2 with lines
```

- In earlier versions you have to place your code in two files and to use “reread”:

File 1:

```
t=0; tmax=18;load "file2.gnuplot"
```

File 2:

```
t=t+1;outfile = sprintf('view%03.0f.png',t);
set view t*5,30,1,1;set output outfile; plot
data.txt; if(t<tmax) reread;
```

KU LEUVEN

Gnuplot - Fitting data

- Define the power law

```
d(x)=c+a*sqrt(b*x)
```

- Fit your data - - you might need to set initial values for a,b,c

```
fit d(x) 'cl1d_integrals.dat' u 1:5 via a,b,c
print a,b
plot [0:5000] c+a*sqrt(x*b) t '' w l ‘test1.dat' u 1:5 t '' w l
```

- Drawing error bars

```
plot ’data.txt’ u 0:1:2:3 w yerrorbars
```

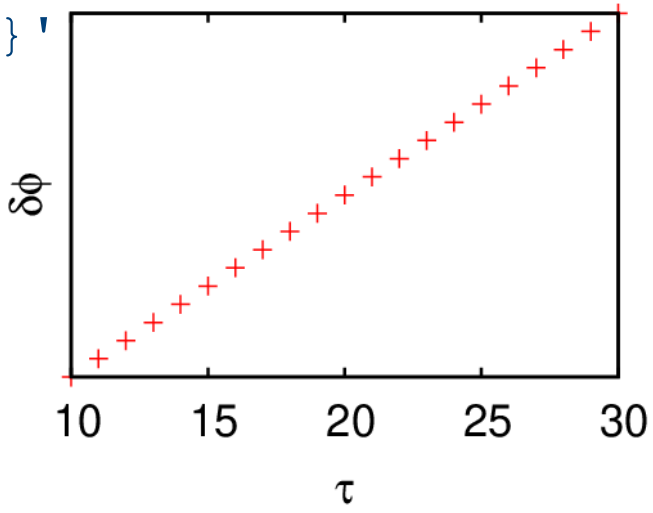
  Uses column 1 as y values, column 2 as lower end of the vertical error bar and column 3 as the upper end of the vertical error bar.

- Horizontal error bars are created with `xerrorbars`

Gnuplot and LaTeX

- Gnuplot is easy to produce postscript plots that an be used by scientific publications in LaTeX
  - Change the terminal type
  - Possibility to use LaTeX notation

```
set terminal postscript eps size 3.5,2.62 \
enhanced color font 'Helvetica,30' linewidth 2
set output 'test.eps'
set xlabel '{/Symbol t}'
set ylabel '{/Symbol d}{/Symbol f}'
unset ytics
set title ''
unset label
set pointsize 2
plot 'test.dat' u 2:3 t ""
reset
```



Gnuplot and LaTeX

```
gnuplot> set terminal latex
gnuplot> set output "example1.tex"
gnuplot> plot [-3.14:3.14] sin(x)
```

- Results in example1.tex

```
% GNUPLOT: LaTeX picture
\setlength{\unitlength}{0.240900pt}
\ifx\plotpoint\undefined\newsavebox{\plotpoint}\fi\
begin{picture}(1500,900)(0,0)
\sbox{\plotpoint}{\rule[-
  0.200pt]{0.400pt}{0.400pt}}%
\put(170.0,82.0){\rule[-0.200pt]{4.818pt}{0.400pt
…
\put(170.0,860.0){\rule[-
  0.200pt]{308.352pt}{0.400pt}}\end{picture}
```

KU LEUVEN

Gnuplot and LaTeX

- LaTeX document:

```
\documentclass{article}
\usepackage{geometry}
\usepackage{color}
\usepackage{graphicx}
\begin{document}
\begin{figure}
  \begin{center}
    \include{example1}
  \end{center}
  \caption{A caption}
\end{figure}
\end{document}
```

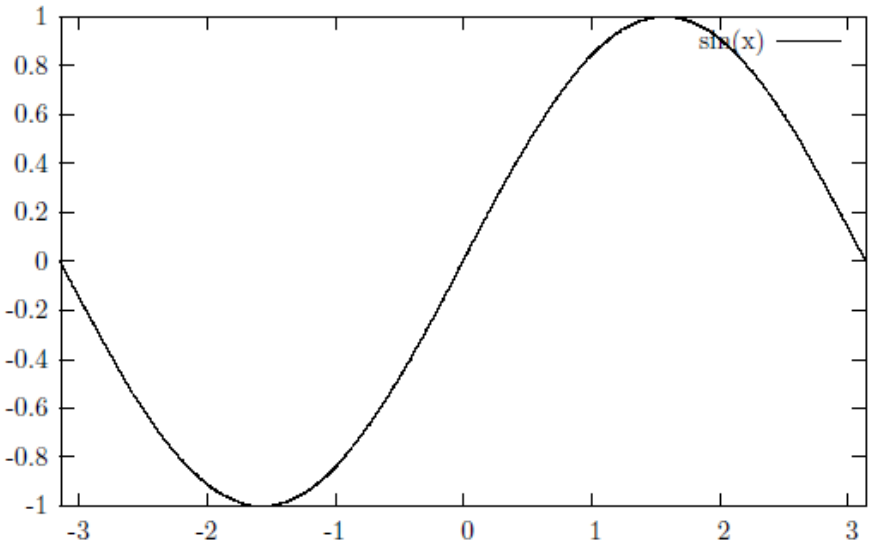


Figure 1: A caption

KU LEUVEN

ImageMagick

- ImageMagick (http://www.imagemagick.org) is a powerful, free software suite to create, edit, and compose bitmap images. It can read, convert and write images in a large variety of formats.

ImageMagick - Tools

- Convert
  - Convert image formats.
  - Many image transformations available – scale, rotate, text, artistic filters
- Display
- Import
  - Screen capture under Linux
- Identify
  - Find out characteristics of image
- Composite
  - Composite multiple images together

ImageMagick - processing

- Add a blue border around image (expand?, replace?, shrink?)
  - `convert -border 10x10 -bordercolor "#6699ff" fan.png fan2.png`
- Add a caption/attribution
  - `convert -font helvetica -fill red -pointsize 14 -draw 'text 446,404 "made by……"' fan2.png fan3.png`

ImageMagick - movies

- ImageMagick can create animations (animated GIFs only)
  - `convert -delay 20 -loop 0 file*.gif anim.gif`
  - `animate anim.gif`

ImageMagick - actions

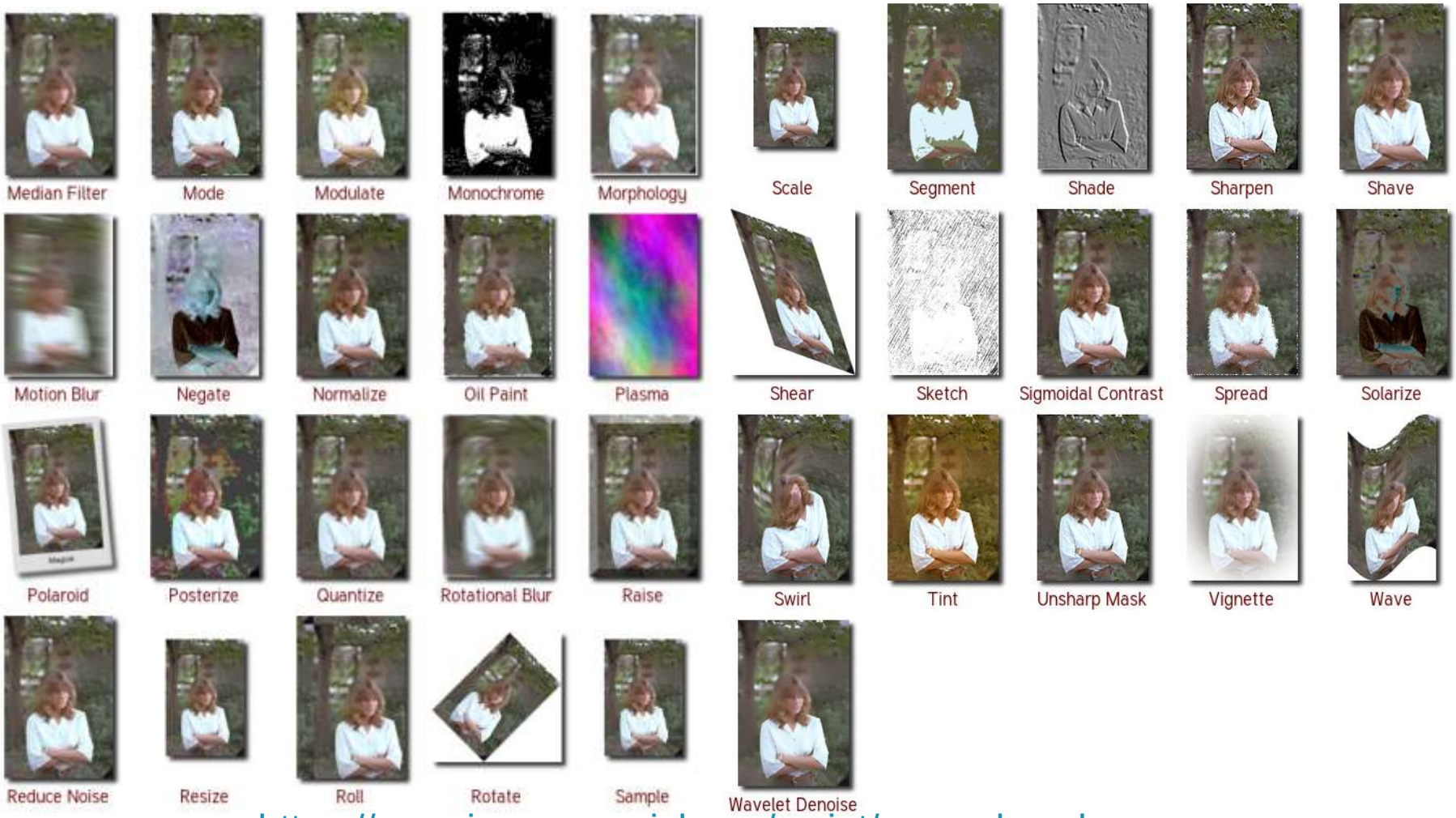
feature	action
Animation	create a GIF animation sequence from a group of images.
Color management	accurate color management with color profiles or in lieu of-- built-in gamma compression or expansion as demanded by the colorspace
Composite	overlap one image over another
Decorate	add a border or frame to an image.
Discrete Fourier Transf.	implements the forward and inverse DFT.
Draw	add shapes or text to an image.
Format conversion	convert an image from one format to another (e.g. PNG to JPEG).
Generalized pixel distortion	correct for, or induce image distortions including perspective.
Image calculator	apply a mathematical expression to an image or image channels.
Image gradients	create a gradual blend of two colors whose shape is horizontal, vertical, circular, or elliptical.
Image identification	describe the format and attributes of an image.
Transform	resize, rotate, deskew, crop, flip or trim an image.
Transparency	render portions of an image invisible.

ImageMagick - examples

https://www.imagemagick.org/script/examples.php

KU LEUVEN

ImageMagick - examples



https://www.imagemagick.org/script/examples.php

ImageMagick – batch processing

- Batch processing -> working with a set or sequence of images
  - All images generally must be the same size
  - Change colors:

```
convert -colorspace Gray example01.png
example001.png
```

  - Resize:

```
convert -scale 20%% example01.png
example001.png
```

  - Add a caption:

```
convert -size 320x85 -draw "text 25,60 'Made
by....'" -fill darkred  example01.png
example001.png
```

  - Process all files:

```
for file in *.png ;do  convert -colorspace Gray
$file ${file%.png}.jpg; done
```

KU LEUVEN

Hands-on 4

KU LEUVEN

1. Plot 10 png images in gnuplot. The first one should show 1 circle, next ones should add and an extra circle with increased radius.
2. Convert the images to grayscale and save them into *.jpg file
3. Make an animated gif
4. Open animation (in the web browser or file browser)

KU LEUVEN

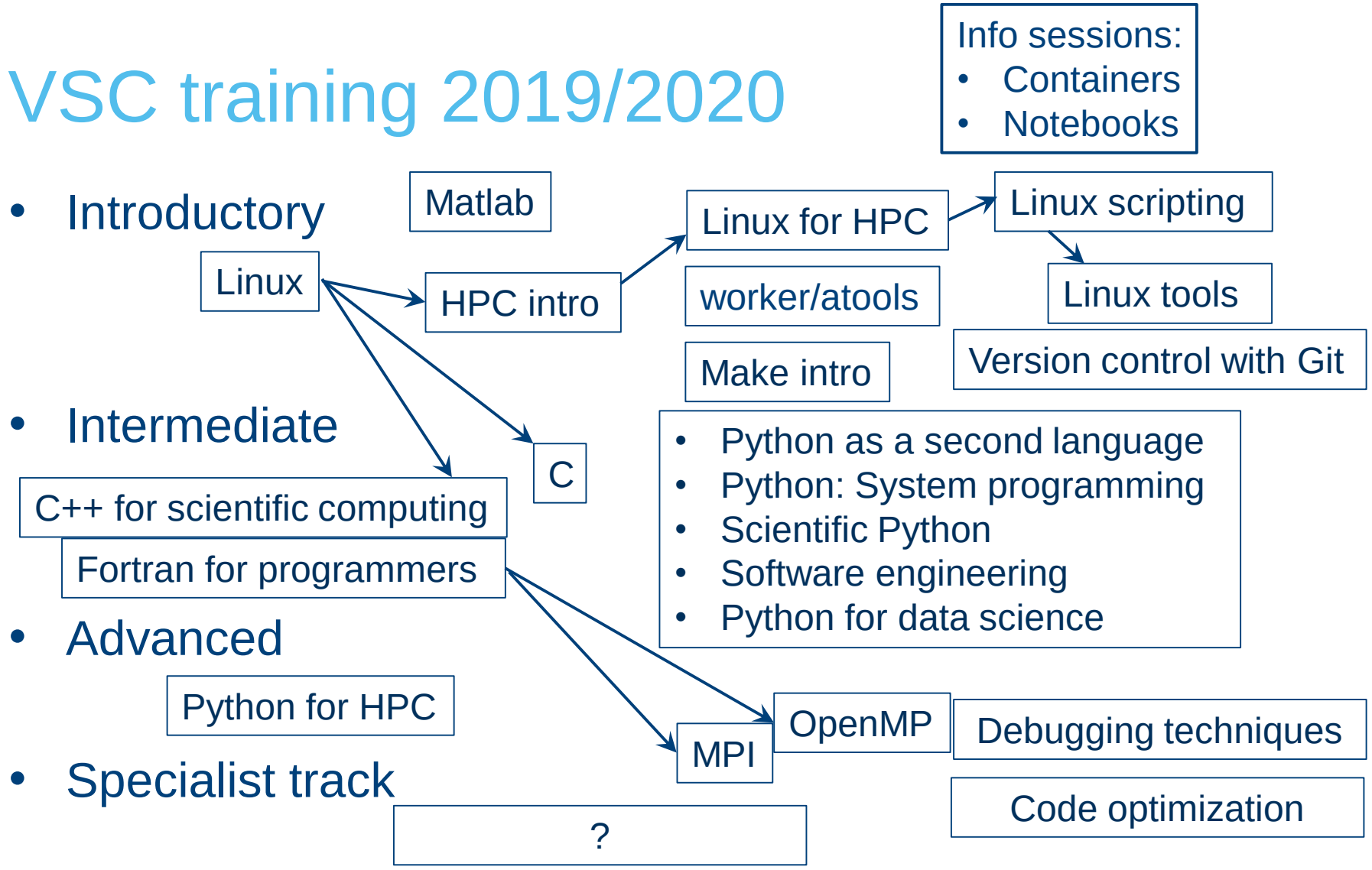
1.
```
set polar
set size square
set term png
set xrange [-10:10];set yrange [-10:10]
r=1
do for [r=1:10] {
outfile = sprintf('radius%02.0f.png',r)
set output outfile
plot for [i=0:r-1] r-i
}
```
2. for file in *.png ;do convert -colorspace Gray $file ${file%.png}.jpg; done
   or manually:
   ```
   convert -colorspace Gray radius.01.png radius.01.jpg
   ```
   ….
3. `convert -delay 20 *png radius.gif`
4. …..

KU LEUVEN

Questions

- Now
- Helpdesk: hpcinfo@kuleuven.be or https://admin.kuleuven.be/icts/HPCinfo_form/HPC-info-formulier
- VSC web site: http://www.vscentrum.be/
  - VSC documentation: https://vlaams-supercomputing-centrum-vscdocumentation.readthedocs-hosted.com/en/latest/
    VSC agenda: training sessions, events
- Systems status page: http://status.kuleuven.be/hpc

VSC training 2019/2020



Stay up-to-date https://www.vscentrum.be/training