UNIVERSITY OF OXFORD

Command-line course part II

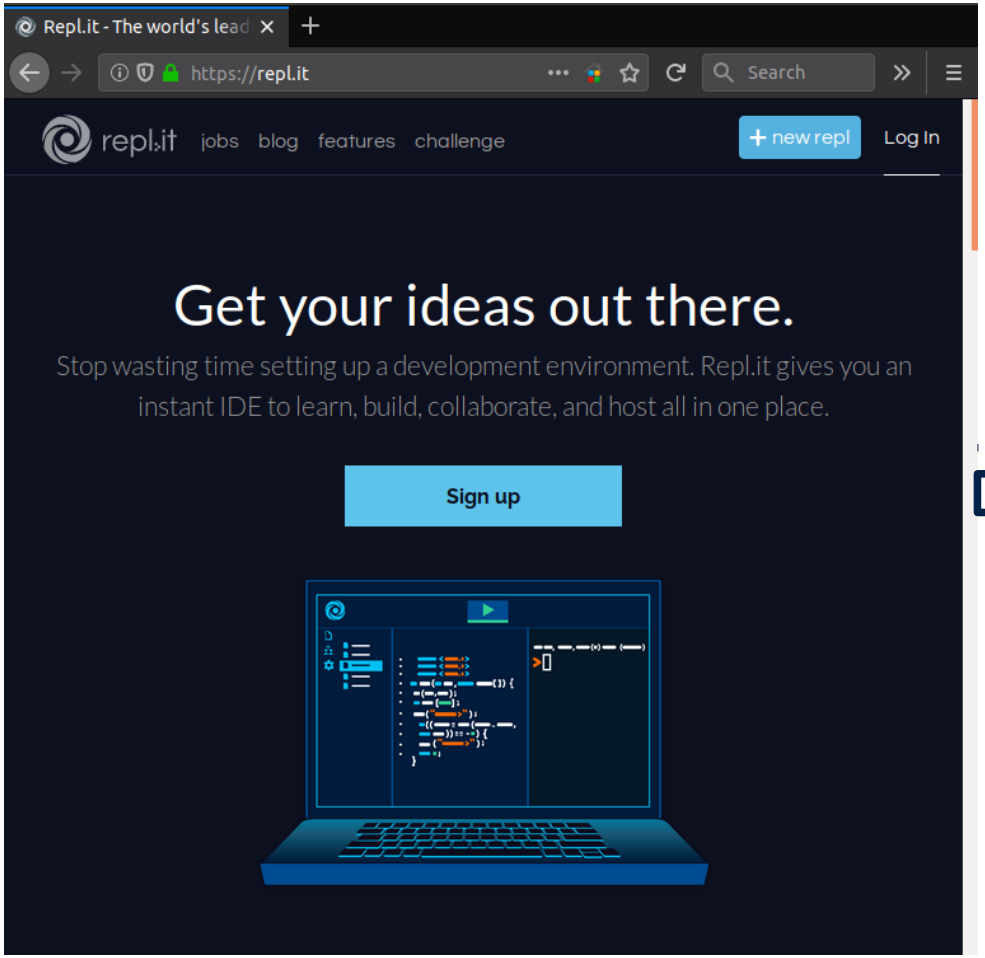
Silvia Salatino, PhD

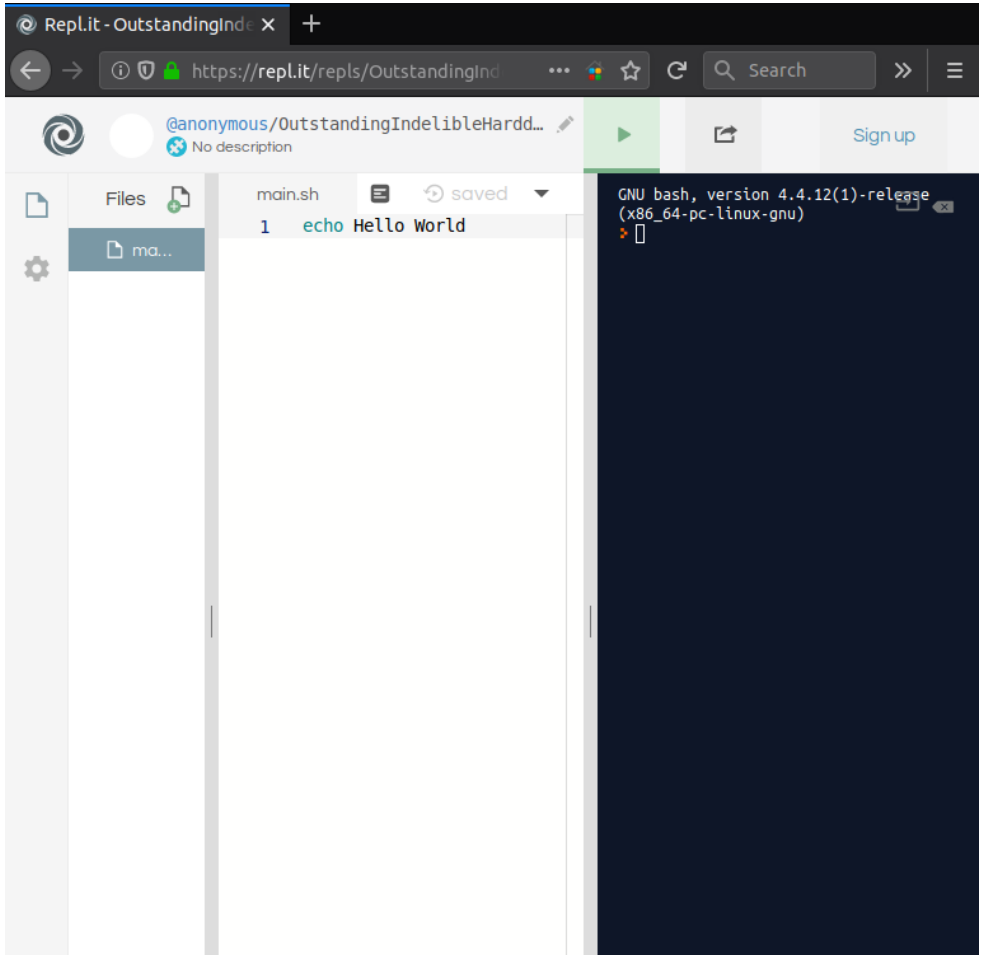
15.10.2019 – Wellcome Centre for Human Genetics, Oxford

Setup

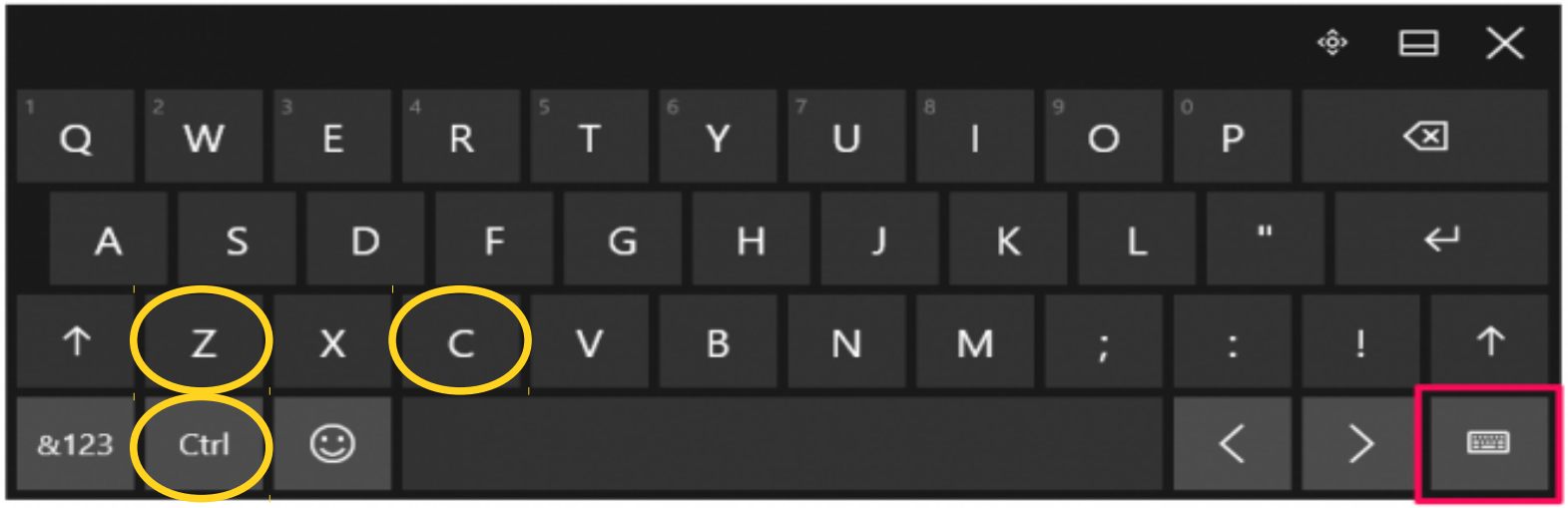
Open a browser and go to this URL **https://repl.it** Then, click on "**+ new repl**" and select "**bash**" from the drop-down menu. Et voilà!

You now have a terminal on the right-hand side, a text editor in the central panel, and a file navigation panel on the left-hand side. Let's start playing with it!





A couple of very useful shortcuts to keep in mind



Control + C

- This key combination will ***interrupt a process***, usually causing it to ***abort***, but it is up to the application to decide.

Control + Z

- This key combination will ***send a foreground process to the background***, in a suspended state (= still alive but not running).

  To view the suspended processed, type "***jobs***":

```
[1]-  Stopped                 cat
[2]+  Stopped                 vi
```

  To go back to the application, type "***fg***" and it be resumed ("***bg***" does the opposite).

  To list the processes running, type "***ps***".

  To kill a suspended process in the background, type "***kill %n***", where n is the number displayed by the jobs command (e.g. kill %1 to terminate the cat process)

Looking for files: locate, find, quoting

Let's say you forgot where a file or a given program is and you need to look for it. What can you do?

- ***locate*** has one advantage over find: speed; it allows you to quickly find a particular file by name
- ***find*** has many advantages over locate: a rich expression synthax, allowing to select files not only by name, but also by date, size, owner, permissions, depth, etc.; it can search a subset of the filesystem; do actions on found files (e.g. -delete, -exec); runs in real time, so the output is always up-to-date (locate relies on a pre-built database)

When looking for files, keep in mind that if you place some text inside ***double quotes***, any special character used by the shell will be treated as an ordinary character, with the only exception of "**$**", "****", "**`**". This is useful if your file name contains white spaces, for example:

```
silvia@zenbook:~/Desktop$ ls -l my file.txt
ls: cannot access 'my': No such file or directory
ls: cannot access 'file.txt': No such file or directory
silvia@zenbook:~/Desktop$ ls -l "my file.txt"
-rw-rw-r-- 1 silvia silvia 0 Nov 12 11:20 my file.txt
```

Instead, ***single quotes*** suppress all expansions, while ***back quotes*** execute the content of variables:

```
silvia@zenbook:~/Desktop$ foo=who
silvia@zenbook:~/Desktop$ echo "$foo"
who
silvia@zenbook:~/Desktop$ echo `$foo`
silvia tty7 2018-11-12 11:20 (:0)
silvia@zenbook:~/Desktop$ echo '$foo'
$foo
```

File compression: gzip, bgzip, gunzip

In Bioinformatics, it is very common to use compressed file formats as files tend to be very big, particularly the raw data ones (like FASTA or FASTQ). Here's a couple useful commands to handle them:

- To compress and uncompress in .gz format use ***gzip*** and ***gunzip***, respectively. NOTE: these commands do overwrite the input file!

```
silvia@zenbook:~/Documents$ ls
sequences.fa
silvia@zenbook:~/Documents$ gzip sequences.fa
silvia@zenbook:~/Documents$ ls
sequences.fa.gz
silvia@zenbook:~/Documents$ gunzip sequences.fa.gz
silvia@zenbook:~/Documents$ ls
sequences.fa
```

- To compress and uncompress in .zip format use ***zip*** and ***unzip***, respectively. NOTE: these commands do not overwrite the input file

```
silvia@zenbook:~/Documents$ zip sequences.fa.zip sequences.fa
  adding: sequences.fa (stored 0%)
silvia@zenbook:~/Documents$ ls
sequences.fa  sequences.fa.zip
```

- To read and parse a compressed file w/o uncompressing it, there are the "Z commands": ***zcat***, ***zless***, ***zgrep***, ***zdiff***, ***zmore***, which behave exactly the same as their corresponding non-Z ones. Some of them uncompress the input file temporarily in the /tmp directory, others uncompress it on the fly. In either cases, they allow you to do operations on compressed files without having to worry about the overhead of uncompressing the file before performing a given operation.

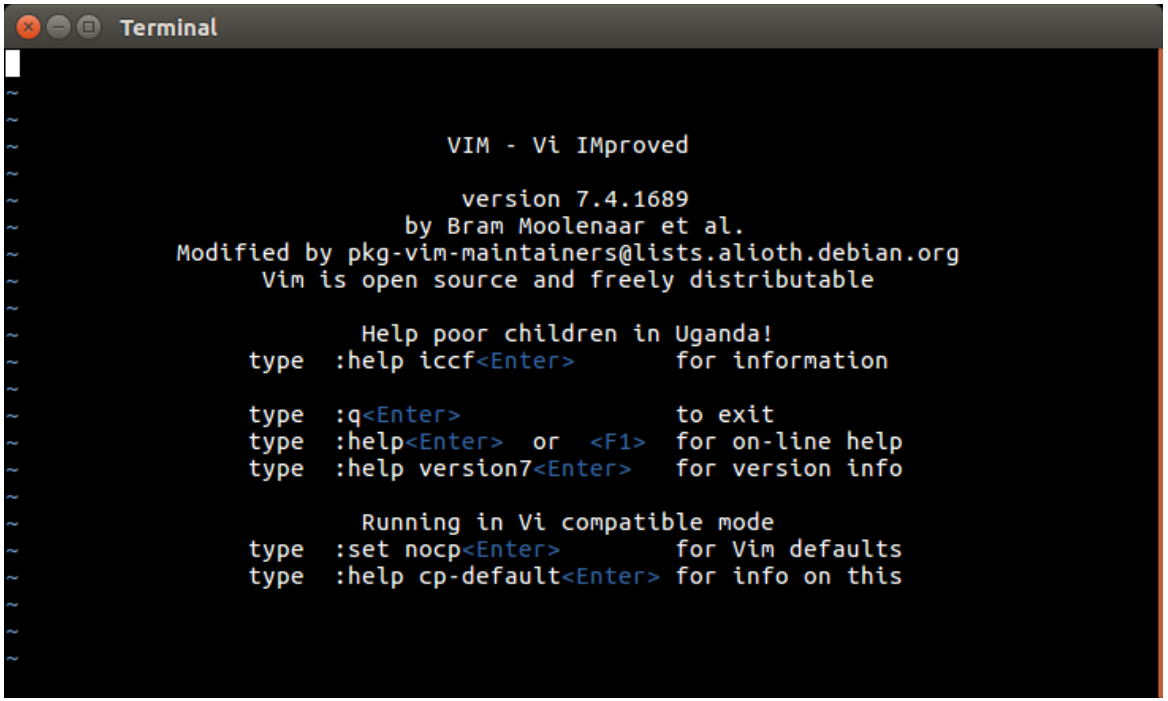
Shell scripts and text editors (1)

A shell script is a file containing a series of bash commands, executed in the command line in the order they've been written in the script.

To write a script, you need a ***text editor***. Some of them work from command-line (e.g. vi / vim, nano, emacs), whereas others have a graphical interface (e.g. gedit, sublime, atom). Most of them need to be installed (especially the newest ones). However, the text editor you'll always find already installed on Linux and Max is ***vi*** / ***vim*** (= vi improved); despite being infamous for its difficulty, it is lightweight and fast.

Let's go quickly through the basic commands:

- Enter "vi" or "vim" in the terminal
- To write some text, move the cursor to the position you want with the arrows, press "**i**" and start typing
- To exit press "**ESC**" to enter normal mode, then "**:**" and enter "**q!**"
- For help type ":help", to save type "**:w**", to save and quit type "**:wq**"
- Google "vi cheat sheet" for the full list of features/options (e.g. type "**:set number**" to display line #)



Now that you've done this rite of passage, feel free to use a more user-friendly editor!

Shell scripts and text editors (2)

Whichever text editor you chose, it's time to start writing a short shell script!

Open a new file in your favourite editor, save it as "hello_world.sh" and start typing these lines:

```
#!/bin/bash
# My first script

echo "Hello World!"
```

The first line is very important. It tells the shell which program is used to interpret the script (in this case it's bash, but it might well be python, awk, etc.). The second line is just a comment; everything written after the "#" is ignored by bash. Comments are crucial to document your code, particularly if you have to share it with others.

Next, change permissions to make your script executable and run it:

```
silvia@zenbook:~/Documents$ ls -l
total 4
-rw-rw-r-- 1 silvia silvia 53 Nov 13 10:44 hello_world.sh
silvia@zenbook:~/Documents$ chmod u+x hello_world.sh
silvia@zenbook:~/Documents$ ls -l
total 4
-rwxrw-r-- 1 silvia silvia 53 Nov 13 10:44 hello_world.sh
silvia@zenbook:~/Documents$ ./hello_world.sh
Hello World!
```

Shell scripts and text editors (3)

Sometimes you might need to repeat the same operation multiple times. To do this, you can use a ***for loop***, which will apply the same operation to a given variable representing each element of an array, set, or list.

Here are a few examples:

script_1.sh (~/Desktop) - gedit

```
#! /bin/bash
# This script prints integer numbers from 1 to 3.
for i in 1 2 3
do
  echo $i
done
```

```
silvia@zenbook:~/Desktop$ ./script_1.sh
1
2
3
```

script_2.sh (~/Desktop) - gedit

```
#! /bin/bash
# This script prints odd numbers between 1 and 10.
for (( i=1; i<=10; i+=2 ))
do
  echo $i
done
```

```
silvia@zenbook:~/Desktop$ ./script_2.sh
1
3
5
7
9
```

script_3.sh (~/Desktop) - gedit

```
#! /bin/bash
# This script prints each element of the array "fruit"
fruit=("apple" "banana" "orange")
for i in ${fruit[@]}
do
  echo $i
done
```

```
silvia@zenbook:~/Desktop$ ./script_3.sh
apple
banana
orange
```

Shell scripts and text editors (4)

...and here are some more examples, ***using as input the output of another command*** (ls in this case):

script_4.sh (~/Desktop) - gedit

```
#! /bin/bash
# This script prints all files in the current directory
for i in $(ls)
do
  echo $i
done
```

```
silvia@zenbook:~/Desktop$ ./script_4.sh
script_1.sh
script_2.sh
script_3.sh
script_4.sh
```

...***or user-defined arguments provided from command line*** (the first argument is assigned to the variable $1, the second to $2, etc.).

script_5.sh (~/Desktop) - gedit

```
#! /bin/bash
# This script prints first and third input arguments
echo $1
echo $3
```

```
silvia@zenbook:~/Desktop$ ./script_5.sh apple banana orange
apple
orange
```

If the number of input arguments can vary from run to run, it's better to use ***$@*** to catch them all (the variable ***$#*** is assigned the total number of input arguments, instead):

script_6.sh (~/Desktop) - gedit

```
#! /bin/bash
# This script prints all the input arguments
for i in $@
do
  echo $i
done
```

```
silvia@zenbook:~/Desktop$ ./script_6.sh apple banana orange
apple
banana
orange
```

Shell scripts and text editors (5)

Often you need to do an operation only when a certain condition is verified, or to add a control that checks if the number of input arguments is correct. This can be done using the ***if - elif - else*** statements:

guess_my_age.sh (~/Desktop) - gedit

```
#! /bin/bash
# This script shows the usage of conditional statements
my_num=$1
if [ $my_num -ge 33 ]
then
  echo "I am not that old! Grr..."
elif [ $my_num -le 31 ]
then
  echo "Thanks, very kind of you :)"
else
  echo "Yay, correct answer!"
fi
```

```
silvia@zenbook:~/Desktop$ ./guess_my_age.sh 50
I am not that old! Grr...
silvia@zenbook:~/Desktop$ ./guess_my_age.sh 27
Thanks, very kind of you :)
silvia@zenbook:~/Desktop$ ./guess_my_age.sh 32
Yay, correct answer!
```

However, you don't necessarily need to use a text editor... bash commands can also be entered directly in the terminal (provided they're really short and you don't need to re-run them over and over!):

```
silvia@zenbook:~/Desktop$ for i in 1 2 3; do echo $i; done
1
2
3
```

Practical session 

Practical session 3 – exercises

1) Open a text editor of your choice. Write a bash script that uses a for loop to print the number of lines in each **`.txt`** file contained in your current working directory (which should still contain the two files created in Practical session II).

2) Modify the script above such that it prints the filename and **`Okay`** only if the word count is greater or equal 6, otherwise it prints the filename and message **`Too few lines!`**

Practical session 3 – solutions

1)

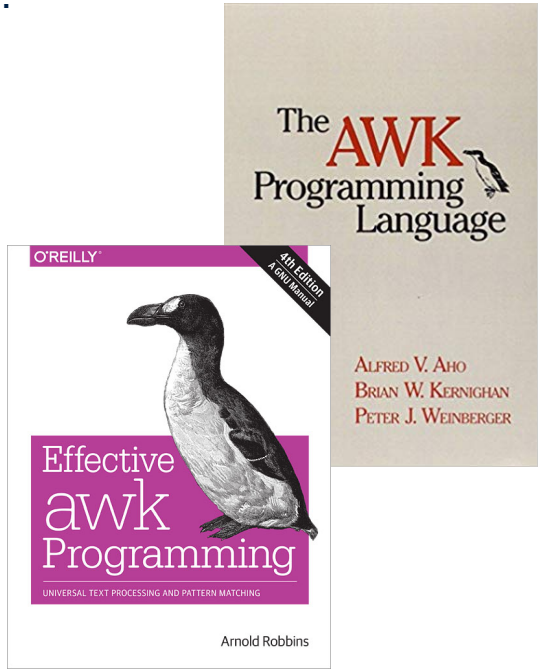
```
#! /bin/bash
for f in `ls *.txt`
do
  echo $f
  wc -l $f
done
```

2)

```
#! /bin/bash
for f in `ls *.txt`
do
  if [ `wc -l $f | cut -d ' ' -f 1` -ge 6 ]
  then
    echo $f
    echo "Okay"
  else
    echo $f
    echo "Too few lines"
  fi
done
```

An extremely powerful programming language: AWK

AWK is a very powerful programming language specifically designed for text processing. It is often used as a data extraction tool, but it can do many other things, including but not limited to:



- operations (e.g. sum of the numbers in a given column of a file)
- printing selected fields in a given order
- reporting matching lines and/or substituting them with a new word/character
- counting the number of non-empty lines
- deleting white spaces or adding certain characters before even/odd lines
- converting hex strings to decimal
- etc.

A very useful thing about AWK is the possibility to do quite complicated operations in the so-called ***one liners***. The following are two examples of AWK one liners; the first converting the file we saw in the previous slide into a FASTA format, and the second reporting lines longer than 10 bp (and their line #):

```
silvia@zenbook:~/Desktop$ awk '{n++; print ">seq_"n; print}' my_sequences.txt
>seq_1
TCGAAAG
>seq_2
AAGTCGAAACT
silvia@zenbook:~/Desktop$ awk 'length($0)>10 {print FNR, $0}' my_sequences.txt
2 AAGTCGAAACT
```

Samtools, Bamtools, Bedtools, Bcftools (1)

The standard file formats used in genomics / transcriptomics are only a few (e.g. FASTA, FASTQ, BAM, CRAM, SAM, BED, GTF/GFF, VCF), but they are quite commonly used. A number of suites have been developed to handle these files:

- ***Samtools*** is *"a set of utilities that manipulate alignments in the BAM format. It imports from and exports to the SAM format, does sorting, merging and indexing, and allows to retrieve reads in any regions swiftly."*

  Several samtools commands can be piped after each other by using "-" to indicate standard input / output. Warnings and error messages are printed to the standard error.

Samtools Home Download Workflows Documentation Support

Manual page from samtools-1.9
released on 18 July 2018

NAME

samtools – Utilities for the Sequence Alignment/Map (SAM) format

SYNOPSIS

```
samtools view -bt ref_list.txt -o aln.bam aln.sam.gz
samtools sort -T /tmp/aln.sorted -o aln.sorted.bam aln.bam
samtools index aln.sorted.bam
samtools idxstats aln.sorted.bam
samtools flagstat aln.sorted.bam
samtools stats aln.sorted.bam
samtools bedcov aln.sorted.bam
samtools depth aln.sorted.bam
samtools view aln.sorted.bam chr2:20,100,000-20,200,000
samtools merge out.bam in1.bam in2.bam in3.bam
samtools faidx ref.fasta
samtools fqidx ref.fastq
samtools tview aln.sorted.bam ref.fasta
samtools split merged.bam
samtools quickcheck in1.bam in2.cram
samtools dict -a GRCh38 -s "Homo sapiens" ref.fasta
samtools fixmate in.namesorted.sam out.bam
samtools mpileup -C50 -f ref.fasta -r chr3:1,000-2,000 in1.bam in2.bam
samtools flags PAIRED,UNMAP,MUNMAP
samtools fastq input.bam > output.fastq
samtools fasta input.bam > output.fasta
samtools addreplacerg -r 'ID:fish' -r 'LB:1334' -r 'SM:alpha' -o output.bam input.bam
samtools collate -o aln.name_collated.bam aln.sorted.bam
samtools depad input.bam
samtools markdup in.algnsorted.bam out.bam
```

sort samtools sort [**-l** *level*] [**-m** *maxMem*] [**-o** *out.bam*] [**-O** *format*] [**-n**] [**-t** *tag*] [**-T** *tmpprefix*] [**-@** *threads*] [*in.sam*|*in.bam*|*in.cram*]

Sort alignments by leftmost coordinates, or by read name when **-n** is used. An appropriate **@HD-SO** sort order header tag will be added or an existing one updated if necessary.

The sorted output is written to standard output by default, or to the specified file (*out.bam*) when **-o** is used. This command will also create temporary files *tmpprefix.%d.***bam** as needed when the entire alignment data cannot fit into memory (as controlled via the **-m** option).

Options:

-l *INT* Set the desired compression level for the final output file, ranging from 0 (uncompressed) or 1 (fastest but minimal compression) to 9 (best compression but slowest to write), similarly to **gzip**(1)'s compression level setting.

If **-l** is not used, the default compression level will apply.

-m *INT* Approximately the maximum required memory per thread, specified either in bytes or with a **K**, **M**, or **G** suffix. [768 MiB]

Samtools, Bamtools, Bedtools, Bcftools (2)

- ***Bamtools*** is a suite of utilities for handling BAM files.

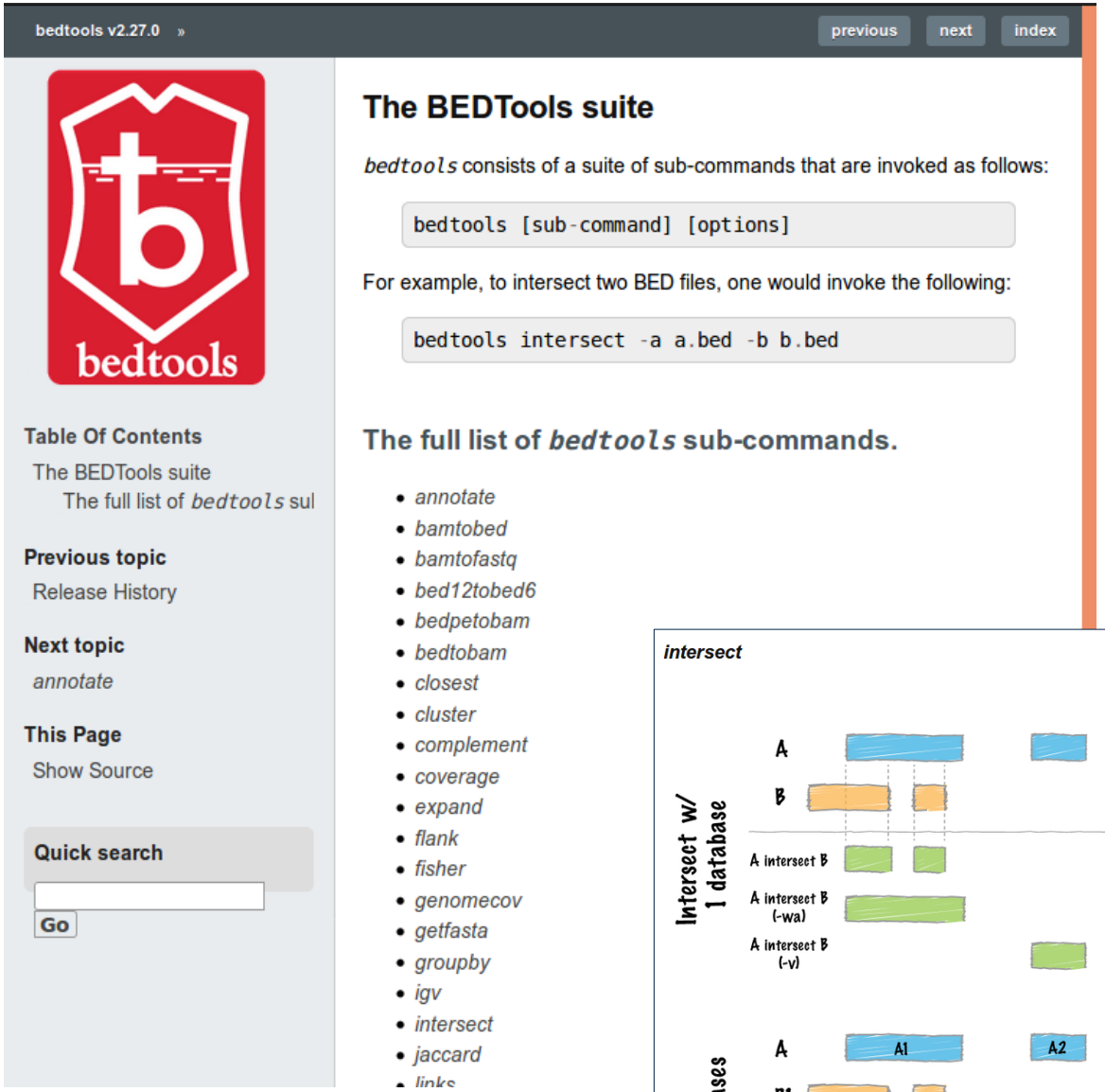
  Most of the operations can also be done with Samtools, but this is more intuitive to use (e.g. no numerical flags to remember), although it is slower...

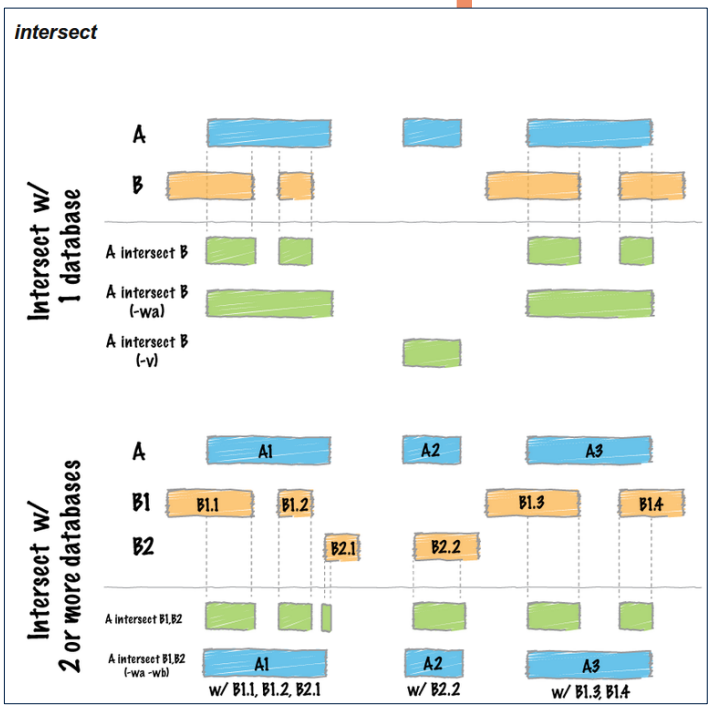
```
[silvia@rescomp1 Teaching]$ /apps/well/bamtools/2.3.0/bin/bamtools --help

usage: bamtools [--help] COMMAND [ARGS]

Available bamtools commands:
        convert         Converts between BAM and a number of other formats
        count           Prints number of alignments in BAM file(s)
        coverage        Prints coverage statistics from the input BAM file
        filter          Filters BAM file(s) by user-specified criteria
        header          Prints BAM header information
        index           Generates index for BAM file
        merge           Merge multiple BAM files into single file
        random          Select random alignments from existing BAM file(s), intended more as a
        resolve         Resolves paired-end reads (marking the IsProperPair flag as needed)
        revert          Removes duplicate marks and restores original base qualities
        sort            Sorts the BAM file according to some criteria
        split           Splits a BAM file on user-specified property, creating a new BAM output
        stats           Prints some basic statistics from input BAM file(s)

See 'bamtools help COMMAND' for more information on a specific command.

[silvia@rescomp1 Teaching]$ /apps/well/bamtools/2.3.0/bin/bamtools filter --help
Description: filters BAM file(s).

Usage: bamtools filter [-in <filename> -in <filename> ... | -list <filelist>] [-out <filename>
egion <REGION>] [ [-script <filename] | [filterOptions] ]

Input & Output:
  -in <BAM filename>                the input BAM file(s) [stdin]
  -list <filename>                  the input BAM file list, one
                                    line per file
  -out <BAM filename>               the output BAM file [stdout]
  -region <REGION>                  only read data from this
                                    genomic region (see documentation for more
                                    details)
  -script <filename>                the filter script file (see
                                    documentation for more details)
  -forceCompression                 if results are sent to stdout
                                    (like when piping to another tool),
                                    default behavior is to leave output
                                    uncompressed. Use this flag to override
                                    and force compression

General Filters:
  -alignmentFlag <int>              keep reads with this *exact*
                                    alignment flag (for more detailed queries,
```

Samtools, Bamtools, Bedtools, Bcftools (3)

- ***Bedtools*** is "*a swiss-army knife of tools for a wide-range of genomics analysis tasks. The most widely-used tools enable genome arithmetic: that is, set theory on the genome.*

  For example, bedtools allows one to intersect, merge, count, complement, and shuffle genomic intervals from multiple files in widely-used genomic file formats such as BAM, BED, GFF/GTF, VCF. While each individual tool is designed to do a relatively simple task (e.g., intersect two interval files), quite sophisticated analyses can be conducted by combining multiple bedtools operations on the UNIX command line."





Certain commands also have schemes to describe in a clear graphical way all the different options

Samtools, Bamtools, Bedtools, Bcftools (4)

- ***Bcftools*** is "*a set of utilities that manipulate variant calls in the Variant Call Format (VCF) and its binary counterpart BCF. All commands work transparently with both VCFs and BCFs, both uncompressed and BGZF-compressed.*"

  Like Samtools, "*BCFtools is designed to work on a stream. It regards an input file "-" as the standard input (stdin) and outputs to the standard output (stdout). Several commands can thus be combined with Unix pipes.*"

LIST OF COMMANDS

For a full list of available commands, run **bcftools** without arguments. For a full list of available options, run **bcftools** *COMMAND* without arguments.

- **annotate** .. edit VCF files, add or remove annotations
- **call** .. SNP/indel calling (former "view")
- **cnv** .. Copy Number Variation caller
- **concat** .. concatenate VCF/BCF files from the same set of samples
- **consensus** .. create consensus sequence by applying VCF variants
- **convert** .. convert VCF/BCF to other formats and back
- **csq** .. haplotype aware consequence caller
- **filter** .. filter VCF/BCF files using fixed thresholds
- **gtcheck** .. check sample concordance, detect sample swaps and contamination
- **index** .. index VCF/BCF
- **isec** .. intersections of VCF/BCF files
- **merge** .. merge VCF/BCF files files from non-overlapping sample sets
- **mpileup** .. multi-way pileup producing genotype likelihoods
- **norm** .. normalize indels
- **plugin** .. run user-defined plugin
- **polysomy** .. detect contaminations and whole-chromosome aberrations
- **query** .. transform VCF/BCF into user-defined formats
- **reheader** .. modify VCF/BCF header, change sample names
- **roh** .. identify runs of homo/auto-zygosity
- **sort** .. sort VCF/BCF files
- **stats** .. produce VCF/BCF stats (former vcfcheck)
- **view** .. subset, filter and convert VCF and BCF files

Thank you for your attention!

Questions?

silvia@well.ox.ac.uk

bioinformatics@well.ox.ac.uk