1 Basic linux commands

Here will be briefly explained some basic linux commands. For information about more commands, feel free to consult with the internet.

1.1 Getting around

Command	Explanation
`ls`	list contents of current directory
`ls` *dirname*	list contents of directory named *dirname*
`cd` *dirname*	change current directory to *dirname*
`cd ~`	go to home directory (default directory)
`cd ..`	go one directory up
`pwd`	print name of current directory
`mkdir` *dirname*	make a new directory named *dirname*
`man` *command*	show manual for *command*

1.2 File manipulation

Command	Explanation
`cp` *filename1 filename2*	copy file
`cp -r` *dirname1 dirname2*	copy directory
`mv` *name1 name2*	move file or directory (can be used for renaming)
`rm` *filename*	remove file (you won't be able to recover removed file)
`rm -r` *dirname*	remove directory and its contents
`more` *filename*	command for paging through text file one screenful at a time
`less` *filename*	similar as `more`. Better for viewing large text files (`q` to quit)
`cat` *filename1 filename2 filenameN*	concatenate text files in print output to terminal window
`head` *filename*	print first 10 lines of a text file to terminal window
`head -n 20` *filename*	print first 20 lines of a text file to terminal window
`tail`	opposite of `head`
`wc` *filename*	show the number of lines, words and characters in a text file
`cut -f 2` *tabDelimitedFilename*	extract 2nd column from a tab delimited file
`cut -f 3 -d ,` *comaSeperatedFilename*	extract 3rd column from a coma seperated file
`nano` *filename*	open *filename* in text editor `nano`

1.3 Archiving and unarchiving of files

Note that for the `tar` utility, option `c` stands for compress, `x` - for uncompress or extract, `z` - for dealing with *tar.gz*, and `j` - for dealing with *tar.bz2*

1.3.1 Compressing

Command	Explanation
`tar -cvf` *filename.tar filename*	**compress** file to **.tar** format
`tar -zcvf` *filename.tar.gz filename*	**compress** file to **.tar.gz** format
`tar -jcvf` *filename.tar.bz2 filename*	**compress** file to **.tar.bz2** format
`zip` *filename.zip filename*	**compress** file to **.zip** format
`gzip` *filename*	**compress** file to **.gz** format

ll

1.3.2 Uncompressing

Command	Explanation
`tar -xvf` *filename.tar*	**uncompress** from **.tar** format
`tar -zxvf` *filename.tar.gz*	**uncompress** from **.tar.gz** format
`tar -jxvf` *filename.tar.bz2*	**uncompress** from **.tar.bz2** format
`unzip` *filename.zip*	**uncompress** from **.zip** format
`gzip -d` *file.gz*	**uncompress** from **.gz** format

1.4 Input/Output redirection

Command	Explanation
command `>` *filename*	output of *command* is saved to *filename*, overwriting it
command `>>` *filename*	output of *command* is appended at the end of *filename*
command `<` *filename*	*command* reads input from *filename*
command1 `|` *command2*	*command2* takes the output of *command1* and produces result

1.5 Lists of commands

Command	Explanation
command1 `;` *command2*	*command2* is executed after *command1*
command1 `&&` *command2*	*command2* is executed if *command1* was successful
command1 `||` *command2*	*command2* is executed if *command1* has failed

1.6 Filters

Command	Explanation
`grep` *text filename*	Prints every line in *filename* containing *text*
`sed 's/`*red*`/`*green*`/'` *filename*	Prints every line in *filename* substituting word *red* with word *green*

1.7 Pattern matching

Pattern	Explanation
`*`	matches zero or more characters
`?`	matches one character
`~`	refers to user's default (home) directory

1.8 Miscellaneous

Command	Explanation
`echo` *text*	display a line of text
`history`	view your command line history
`wget` *someWebAddress*	download contents of *someWebAddress* to current directory
`make`	tool that is used to compile source code creating executables
`export` *name*=*value*	sets *value* to *name*. Type `echo $`*name* to view *value*
`source` *filename*	read and execute commands from the *filename* argument

1.9 Syntax

Syntax element	Explanation
\	allows to split command over multiple lines

1.10 `for` loop

`for` loop allows to iterate over a list of items and apply commands on them:

```
for item in item1 item2 item3 item4; do
   echo $item
done
```

Note that we are assigning to `item` each value in list of items (`item1 item2 item3 item4`) in the first line, and we are retrieving the value of `item` by putting $ before it.

We can also write `for` loop in a single line:

```
for item in item1 item2 item3 item4; do echo $item; done
```

2 Setup of the working environment

In our home folders there is only one directory named `data/day1`. You will find there *IonTorrent* sequencing reads that we are going to analyze and some other necessary resources.

Let's create two seperate directories: one directory where we will be performing all analysis for sequencing data, and other - where we will be downloading and compiling necessary software:

```
mkdir ngs_work
mkdir programs
```

Since we do not have the administrator's rights on this server, we can't install software on the system. However, we can still install software locally in our home directories. We will create a special directory where all our executables will be stored. Let's create this directory and name it `binaries`. In linux terminal type:

```
mkdir binaries
```

Let's make this directory special - every executable file we put there, we will be able to easily execute, just by typing the executable's file name from anywhere in the system. To achieve this, we will be adding `~/binaries` folder to the `$PATH` system's environment variable by editing a text file named `.bashrc` using text editor `nano`. In linux terminal type:

```
nano ~/.bashrc
```

Text editor will open the file and you can edit it. Navigate to the bottom of the text file using arrows and type in following text:

```
export PATH=~/binaries:$PATH
```

Hit `Ctrl o` and `Enter` to save file and `Ctrl x` to exit. Reload `.bashrc`:

```
source .bashrc
```

> `$PATH` variable contains a list of directories that the system will look in, when we are entering a command. To view contents of `$PATH`, type in terminal:
>
> ```
> echo $PATH
> ```

To install software we will simply have to copy executable to our special directory `binaries`.

Many of the open source tools are deposited in the `https://github.com` repository. To download software from `https://github.com` easily, we will use a tool called `git`. `git` is already preinstalled on our servers, however, on your own Ubuntu servers you can install it by typing:

```
sudo apt-get install git
```

3 *De-novo* assembly of sequenced reads

3.1 Installation of *de-novo* assembler

For *de-novo* assembly we will use `mira` assembler. You can download it from `http://sourceforge.net/projects/mira-assembler/`. Click on `Files` → `MIRA` → `stable`. Rightclick on `mira_4.0.2_linux-gnu_x86_64_s` and `Copy link address`. To download it on our linux server, we will be using command `wget`. In linux terminal type:

```
cd ~/programs
wget -O mira.tar.bz2
```

paste the copied location and hit `Enter`. The program will be downloaded in our `~/programs` directory. The downloaded software is archived in *.tar.bz2* format, therefore we need to extract it from archive. To extract it from archive, type in terminal:

```
tar -jxvf mira.tar.bz2
```

A new folder named `mira_4.0.2_linux-gnu_x86_64_static` will appear. This is our extracted software. MIRA is already precompiled for us, so we just need to find the compiled executable files in the folder and install them copy them to our binaries:

```
cd mira_4.0.2_linux-gnu_x86_64_static
cd bin
cp mira ~/binaries
cd ~
```

Check if MIRA was installed successfully:

```
mira
```

If you see

```
The program 'mira' is currently not installed.
```

then reload `.bashrc` again:

```
source .bashrc
```

and check the installation again.

The sequenced reads we obtain from sequencing platform usually are in *FASTQ* format. Each record in *FASTQ* file consists of four entries:

1. read ID, beginning with symbol @
2. DNA sequence of read
3. symbol +
4. *ASCII* encoded quality of each nucleotide in read

Let's create a seperate directory for our *de-novo* assembly project and copy the reads in it:

```
cd ngs_work
mkdir denovo
cd denovo
cp ~/data/day1/denovo_reads.fastq .
```

MIRA needs a manifest file for performing *de-novo* assembly. We will create a basic manifest file. Our manifest file will consist of 5 entries. From MIRA's manual, these entries are:

- **project** - name of our assemblies project. The project name will be used by MIRA in project's directory naming
- **job** - tells the assembler whether
    1. we want to perform *de-novo* assembly or map reads against reference genome

2. genomic DNA or transcripts were sequenced
3. we want accurate (slow) or draft (fast) assembly

- **readgroup** - tells assembler which reads can be pooled together when assembling reads from multiple sequencing technologies
- **data** - tells the assembler where are our reads
- **technology** - tells the assembler what sequencing technology was used for generating reads

To create manifest file, we will use text editor `nano`. To start the text editor, type

```
nano
```

and the editor will open. Now, to create the manifest file, in the text editor type:

```
project=denovo_reads
job=denovo,genome,draft
readgroup
data=denovo_reads.fastq
technology=iontor
```

To save the text file hit `Ctrl o`, enter the name of the file (e.g. `denovo_reads.mnfst`), hit `Enter` to save and `Ctrl x` to quit `nano`. To launch MIRA, type:

```
mira denovo_reads.mnfst
```

If you wish to gain finer control of some aspects of the assembling process, then, please, do refer to the MIRA's manual.

4 Building variant calling pipeline

We have *IonTorrent* targeted resequencing data from human chromosomes 13., 17. and 20. Our task is to find all nonsynonymous and stop mutations present in the data and to automatize this process by building data analysis pipeline. To accomplish this task we can divide our work in following subtasks:

- Installation of relevant tools
- Obtaining of reference sequences
- Read mapping against reference sequence
- SNP calling
- SNP annotation and filtering of nonsynonymous and stop mutation variants
- Pipeline building

4.1 Tool installation

4.1.1 Installation of *IonTorrent* mapping software `tmap`

To detect variants present in the data we need to map sequencing reads against reference sequence. For reads generated with *IonTorrent* sequencing platform we will use program `tmap`.

`tmap` and its installation instructions can be found at `https://github.com/iontorrent/TS/tree/master/Analysis/TMAP`. Since we do not have the administrator's rights on this server, we can't install software on the server. However, we can still use it locally. Compilation instructions:

```
cd ~/programs
git clone git://github.com/iontorrent/TMAP.git
cd TMAP
git submodule init
git submodule update
sh autogen.sh
./configure
make
```

Lets test the program to confirm that it was compiled successfully:

```
./tmap
```

If you see the programs interface, the program was compiled successfully. Move the compiled binary file to our `binaries` folder:

```
cp tmap ~/binaries
```

4.1.2 Installation of `samtools` and `bcftools`

We will also need two tools named `samtools` and `bcftools` which are used for manipulation of mapped reads and variation calling. You can obtain these tools from `http://sourceforge.net/projects/samtools/`. Click on `Files` → `samtools` → `1.2` Rightclick on `samtools-1.2.tar.bz2` and choose `Copy link address`. We will download these tools in directory `programs`:

```
cd ~/programs
wget -O samtools.tar.bz2
```

paste the copied location and hit `Enter`. Repeat this process for `bcftools` - `Copy link address` and download it with `wget`:

```
wget -O bcftools.tar.bz2
```

paste the copied location and hit `Enter`.

The tools are compressed in *tar.bz2* format, so we need to extract them:

```
tar -jxvf samtools.tar.bz2
```

```
tar -jxvf bcftools.tar.bz2
```

> Note that if we had a lot more files to extract and it would be too time consuming to manually extract them, we could use a `for` loop and pattern matching to extract archives automatically:
>
> ```
> for archive in *.tar.bz2; do
>    tar -jxvf $archive
> done
> ```

We have downloaded and extracted source code of the tools, but to make these tools usable, we need to compile the source code. Source code compiling is performed with command `make`:

```
cd samtools-1.2
make
```

If (hopefully) no errors were encountered, then `samtools` was compiled correctly. Type:

```
./samtools
```

to test the tool. If you see the program's interface, then the program was compiled successfully. Move compiled binary file to a directory where we are storing our compiled software:

```
cp samtools ~/binaries
```

Repeat the same process for `bcftools` (go to `bcftools` source code directory, compile it using `make` and copy resulting binary file to `~/binaries`)..

4.2 Obtaining reference sequences

We will download reference sequences in a seperate directory to avoid file cluttering. Let's make a new directory in our `ngs_work` directory named `reseq`, and there we will create a seperate folder `ref` for our reference sequences:

```
cd ~
cd ngs_work
mkdir reseq
cd reseq
mkdir ref
cd ref
```

Our reference sequences can be accessed from a database made by University of California, Santa Cruz. The web address of the database is `http://genome.ucsc.edu/`. To find the necessary references sequences for chromosomes 13., 17. and 20. click on `Downloads` → `human` → `Data set by chromosome`. Right click on `chr13.fa.gz` and choose `Copy link location`. In terminal type

```
wget
```

paste the copied location and hit `Enter`. The reference sequence is compressed in `.gz` format, so we need to extract it:

```
gzip -d chr13.fa.gz
```

File `chr13.fa` will appear in our folder.

Repeat the process for chromosomes 17. and 20.

After we have obtained and extracted our reference sequences, we need to concatenate them. To accomplish this task we will use command `cat`:

```
cat chr13.fa chr17.fa chr20.fa > chr_merged.fa
```

And we have the needed reference for further data processing steps.

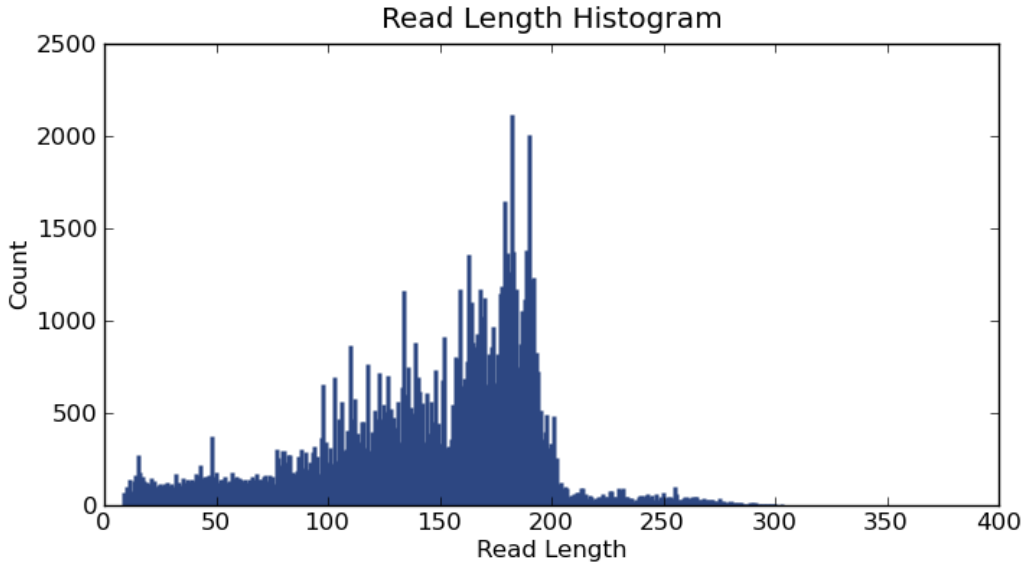
4.3 Read mapping against reference sequence

To enable read mapper to efficiently read and process reference sequence, we need to index reference sequence using mapping software's provided indexing function. Often generated indexes are incompatible between different read mappers. As a consequence, almost every read mapping software has their own indexing algorithms.

To perform reference sequence indexing with `tmap` software, type in linux terminal:

```
tmap index -f ref/chr_merged.fa
```

To perform the actual read mapping against our reference there are six different mapping algorithms. Some of them are suitable for short reads, some - for longer reads. By looking at read length distribution of our sample, we can conclude that most of the reads are longer than 150 bp.



We will choose from one of the algorithms that are suited for longer reads, and one of such algorithms is `map3`. To perform the read mapping we need to:

- provide it with reference sequence (`-f ref/chr_merged.fa`)
- provide it with sequencing reads (`-r reseq_reads.fastq`)
- tell that sequencing reads are in *FASTQ* format (`-i fastq`)
- tell that we want alignment to be in compressed *BAM* format(`-o 1`)
- tell that we want output to be saved in a file named `reseq_reads_mapped.bam` (`-s reseq_reads_mapped.bam`)

In linux terminal type:

```
tmap map3 -f ref/chr_merged.fa \
   -r reseq_reads.fastq \
   -i fastq \
   -o 1 \
   -s reseq_reads_mapped.bam
```

We have obtained *BAM* file. *BAM* stands for **B**inary S**AM** file. *SAM*, in turn, stands for **S**equence **A**lignment/**M**ap format.

4.3.1 *BAM* file viewing and manipulation

Let's find out what is in the first 20 lines of *BAM* file using `samtools` and `head`:

```
samtools view reseq_reads_mapped.bam | head -n 20
```

It is also possible to extract some specific region from the *BAM* file with `samtools view` by adding coordinates in form `chr`*`N`*`:`*`start-end`* at the end of command:

```
samtools view reseq_reads_mapped.bam chr13:27925800-27925900
```

To interpret `FLAG` field (2nd column in *BAM* file), you can use `samtools flags` command providing it with the flag you are interested in:

```
samtools flags 4
```

To view *BAM* file header, use `-H` option in `samtools view` command:

```
samtools view -H reseq_reads_mapped.bam
```

As you can see, we have not defined our read group identifier (`ID:NOID`) and sample name (`SM:NOSM`). We can correct it using `samtools` command `reheader`. It takes a *BAM* header and *BAM* file as input. It replaces *BAM* file's header with the provided header and outputs new *BAM* file.

We will read our *BAM* files header and modify it using `sed` utility. The modified header will be given to `samtools reheader` along with the original bam file. The output will be saved using `>`

operator:

```
samtools view -H reseq_reads_mapped.bam \
   | sed 's/ID:NOID/SM:SomeID/;s/SM:NOSM/SM:Sample1/' \
   | samtools reheader - reseq_reads_mapped.bam \
   > reseq_reads_mapped.reheaded.bam
```

Alternatively, we could have simply defined the read group ID and sample name during mapping process, using `tmap`'s option `-R`:

```
tmap map3 -f ref/chr_merged.fa \
   -r reseq_reads.fastq \
   -i fastq \
   -R ID:SomeID \
   -R SM:Sample1 \
   -o 1 \
   -s reseq_reads_mapped.reheaded.bam
```

Detailed information about *SAM/BAM* file format, see `http://samtools.github.io/hts-specs/SAMv1.pdf`.

We can use `samtools` command `tview` to view alignment in a more human readable format. `tview` accepts only sorted and indexed *BAM* files. Forunately, we can do this easily using `samtools` commands `sort` and `index`:

```
samtools sort reseq_reads_mapped.reheaded.bam reseq_reads_mapped.reheaded.sorted
samtools index reseq_reads_mapped.reheaded.sorted.bam
```

BAM file indexing allows programs to efficiently access any part of the *BAM* file. *BAM* index file names usually end with `.bam.bai`.

After sorting and indexing, we can view our *BAM* file. In linux terminal type:

```
samtools tview reseq_reads_mapped.reheaded.sorted.bam ref/chr_merged.fa
```

Alignment will open in terminal. To get help on viewing alignment, type `?`. Type `q` to quit viewer. A good place to view mapped reads and test the viewer is `chr13:27925825`. To go to this position, type `g` and enter this coordinate as it is shown.

To get overall *BAM* file statistics and number of mapped reads for each chromosome, you can use `samtools` commands `flagstat` and `idxstats`:

```
samtools flagstat reseq_reads_mapped.reheaded.sorted.bam
samtools idxstats reseq_reads_mapped.reheaded.sorted.bam
```

To use these commands, *BAM* file must be sorted and indexed.

4.4 SNP calling

To call SNPs we will use `samtools` and `bcftools`. Using `samtools` we will generate our data in *pileup* format which is then accessed by `bcftools` to call SNPs. *Pileup* format is text based

description of mapped bases at each position of reference sequence. You can think of it as a vertical version of alignment viewing. Variant callers need sorted by position *BAM* file to successfuly detect variants and we have already done that, so we can move to variant calling.

There are two choices of variant callers in `bcftools`: `--consensus-caller` or `--multiallelic-caller`. `consensus-caller` is older variant calling model and assumes only biallelic sites. It can miss somatic mutations. `multiallelic-caller` does not have such an asumption and is more sensitive than `consensus-caller`. `multiallelic-caller` is recommended for most tasks, and we will also use it. For this tutorial, we will skip insertions/deletions:

```
samtools mpileup -uf ref/chr_merged.fa \
   reseq_reads_mapped.reheaded.sorted.bam \
   | bcftools call -mv --skip-variants indels \
   > reseq_reads.bcftools_snps.vcf
```

4.5 SNP annotation and filtering with `snpEff` and `SnpSift`

After generating a list of SNPs in *VCF* format, we will use a tool `snpEff` to predict whether they are causing amino acid change and `SnpSift` to filter nonsynonymous and stop gained SNPs.

4.5.1 Setting up and configuring `snpEff` and `SnpSift`

`snpEff` and `SnpSift` are bundled together and are available at `http://snpeff.sourceforge.net`. To download and install them, type in linux terminal:

```
cd ~/programs
wget http://sourceforge.net/projects/snpeff/files/snpEff_latest_core.zip
```

paste the copied location and hit `Enter`. To `unzip snpEff` type:

```
unzip snpEff_latest_core.zip
```

Unfortunately we wil not be able to launch both programs from `binaries` folder and we can leave the software in `programs` directory.

`snpEff` requires databases to predict effects of SNPs. The needed database is already downloaded for us and resides in `~/data/day1/snpEffData`. Our task is to configure `snpEff` to tell it where this database can be found.

After downloading and unzipping of `snpEff` we will need to edit `snpEff's` configuration file named `snpEff.config` and change entry `data.dir = ./data/` to `data.dir = ~/data/day1/snpEffData/`

`SnpSift` and `snpEff` are *.jar* files which means that these are `java` applications. To run `java` applications through linux terminal, use command `java -jar` *filename.jar*. To sucessfully perfrom SNP annotation, we need to provide `snpEff` with the command we are executing (`ann` - telling `snpEff` that we want to annotate SNPs), its configuration file (`-c snpEff.config`), database we are annotating against (`GRCh38.76`) and our *VCF* file. Since `snpEff` by default prints output to terminal, we can save it using `>` operator. To annotate called variants, type:

```
java -jar ~/programs/snpEff/snpEff.jar \
   ann \
   -c ~/programs/snpEff/snpEff.config \
```

```
   GRCh38.76 \
   reseq_reads.bcftools_snps.vcf \
   > reseq_reads.bcftools_snps.annotated.vcf
```

In the resulting *VCF* file `snpEff` adds a new entry in the `INFO` field - `ANN`. To filter only those SNPs that are marked as nonsynonymous or stop gained we are going to use `SnpSift`'s command `filter`. `SnpSift`'s `filter` command needs *VCF* file annotated by `snpEff` and filtering expression. We will use expression `"(ANN[*].EFFECT = 'missense_variant)' || (ANN[*].EFFECT = 'stop_gained')"` which can be translated as: if any "ANN" field for this variant, has effect of missense or stop gain, then print this variant to terminal. For more examples on filtering expressions, see `http://snpeff.sourceforge.net/SnpSift.html`. To perform variant filtering, type:

```
java -jar ~/programs/snpEff/SnpSift.jar \
   filter \
   -f reseq_reads.bcftools_snps.annotated.vcf \
   "(ANN[*].EFFECT = 'missense_variant') || (ANN[*].EFFECT = 'stop_gained')" \
   > reseq_reads.bcftools_snps.annotated.nonsyn_stop.vcf
```

4.6 Pipeline building

Writing all these commands by hand is time consuming, tedious and error prone and we need a better way to execute these commands. One way how to solve this problem is to save all our commands in a text file and, when the time comes to analyze our data, copy these commands in the terminal. While this method can save our time on looking in manuals for the correct commands, it is still time consuming to wait for each command to end, so that we can copy the next one in. We need a better way how to automatically execute these commands and here *bash* scripting comes in handy.

bash stands for ***B**ourne-**A**gain **Sh**ell* and is one of the most popular choices for perfoming shell scripting. Whenever we typed a command in linux terminal it was shell that was performing all the commands and giving us output. So we have been using *bash* all along not knowing it.

Starting a *bash* script is really easy - open a text editor, on the first line write `#!/bin/bash`, on the next lines some commands, and save the file. You can tell *bash* to launch script by typing in terminal:

`bash` *yourFileName*

To build SNP calling pipeline we will use exactly this approach - open text editor and copy following lines (note that text starting with `#` is a comment line where you can write anything you want to make the code readable):

```
#!/bin/bash

#Read mapping
tmap map3 -f ref/chr_merged.fa \
  -r reseq_reads.fastq \
  -i fastq \
  -R ID:SomeID \
  -R SM:Sample1 \
  -o 1 \
  -s reseq_reads_mapped.reheaded.bam
```

```
#Read sorting
samtools sort reseq_reads_mapped.reheaded.bam \
reseq_reads_mapped.reheaded.sorted

#SNP calling with bcftools
samtools mpileup -uf ref/chr_merged.fa \
  reseq_reads_mapped.reheaded.sorted.bam \
  | bcftools call -mv --skip-variants indels \
  > reseq_reads.bcftools_snps.vcf

#SNP annotating with snpEff
java -jar ~/programs/snpEff/snpEff.jar \
  ann \
  -c ~/programs/snpEff/snpEff.config \
  GRCh38.76 \
  reseq_reads.bcftools_snps.vcf \
  > reseq_reads.bcftools_snps.annotated.vcf

#filtering of missense and stop SNPs
java -jar ~/programs/snpEff/SnpSift.jar \
  filter \
  -f reseq_reads.bcftools_snps.annotated.vcf \
  "(ANN[*].EFFECT='missense_variant')||(ANN[*].EFFECT='stop_gained')"
  > reseq_reads.bcftools_snps.annotated.nonsyn_stop.vcf
```

Save it as `snp_pipeline.sh` and launch it by typing in terminal:

```
bash snp_pipeline.sh
```

This is good if we want to repeat this analysis over and over again on this one sample. What if we have a new sample we want to analyse? We could replace filenames of previous sample, to filenames of the new sample by hand. However, there is a better way - we can save our sample name in a variable, and use that variable throughout the script. In *bash* we assign value to a variable with `=` operator, and retrieve value from variable by adding `$` at the beginning of variable's name:

```
#!/bin/bash

#Setting sample name
sample_name=reseq_reads.fastq

#Read mapping
tmap map3 -f ref/chr_merged.fa \
  -r $sample_name \
  -i fastq \
  -R ID:SomeID \
  -R SM:Sample1 \
  -o 1 \
  -s $sample_name.mapped.reheaded.bam

#Read sorting
samtools sort $sample_name.mapped.reheaded.bam \
$sample_name.mapped.reheaded.sorted

#SNP calling with bcftools
samtools mpileup -uf ref/chr_merged.fa \
  $sample_name.mapped.reheaded.sorted.bam \
  | bcftools call -mv --skip-variants indels \
  > $sample_name.bcftools_snps.vcf
```

```
#SNP annotating with snpEff
java -jar ~/programs/snpEff/snpEff.jar \
  ann \
  -c ~/programs/snpEff/snpEff.config \
  GRCh38.76 \
  $sample_name.bcftools_snps.vcf \
  > $sample_name.bcftools_snps.annotated.vcf

#filtering of missense and stop SNPs
java -jar ~/programs/snpEff/SnpSift.jar \
  filter \
  -f $sample_name.bcftools_snps.annotated.vcf \
  "(ANN[*].EFFECT='missense_variant')||(ANN[*].EFFECT='stop_gained')"
  > $sample_name.bcftools_snps.annotated.nonsyn_stop.vcf
```

We can go even further - we can provide sample name as an argument to our pipeline. There is a predefined *bash* variable (`$1`) that stores everything we supply as an argument for our script:

```
#!/bin/bash

#Setting sample name
sample_name=$1

#Read mapping
tmap map3 -f ref/chr_merged.fa \
  -r $sample_name \
  -i fastq \
  -R ID:SomeID \
  -R SM:Sample1 \
  -o 1 \
  -s $sample_name.mapped.reheaded.bam

#Read sorting
samtools sort $sample_name.mapped.reheaded.bam \
$sample_name.mapped.reheaded.sorted

#SNP calling with bcftools
samtools mpileup -uf ref/chr_merged.fa \
  $sample_name.mapped.reheaded.sorted.bam \
  | bcftools call -mv --skip-variants indels \
  > $sample_name.bcftools_snps.vcf

#SNP annotating with snpEff
java -jar ~/programs/snpEff/snpEff.jar \
  ann \
  -c ~/programs/snpEff/snpEff.config \
  GRCh38.76 \
  $sample_name.bcftools_snps.vcf \
  > $sample_name.bcftools_snps.annotated.vcf

#filtering of missense and stop SNPs
java -jar ~/programs/snpEff/SnpSift.jar \
  filter \
  -f $sample_name.bcftools_snps.annotated.vcf \
  "(ANN[*].EFFECT='missense_variant')||(ANN[*].EFFECT='stop_gained')"
  > $sample_name.bcftools_snps.annotated.nonsyn_stop.vcf
```

To launch it, type:

```
bash snp_pipeline.sh reseq_reads
```

Suppose that we have multiple *FASTQ* files that we want to analyze. Using pattern matching and `for` loop, we can automatically process them with our pipeline:

```
for fastqFile in *.fastq; do
   bash snp_pipeline.sh $fastqFile
done
```