

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
<code>ls</code>	list contents of current directory
<code>ls <i>dirname</i></code>	list contents of directory named <i>dirname</i>
<code>cd <i>dirname</i></code>	change current directory to <i>dirname</i>
<code>cd ~</code>	go to home directory (default directory)
<code>cd ..</code>	go one directory up
<code>pwd</code>	print name of current directory
<code>mkdir <i>dirname</i></code>	make a new directory named <i>dirname</i>
<code>man <i>command</i></code>	show manual for <i>command</i>

1.2 File manipulation

Command	Explanation
<code>cp <i>filename1 filename2</i></code>	copy file
<code>cp -r <i>dirname1 dirname2</i></code>	copy directory
<code>mv <i>name1 name2</i></code>	move file or directory (can be used for renaming)
<code>rm <i>filename</i></code>	remove file (you won't be able to recover removed file)
<code>rm -r <i>dirname</i></code>	remove directory and its contents
<code>more <i>filename</i></code>	command for paging through text file one screenful at a time (q to quit)
<code>less <i>filename</i></code>	similar as more. Better for viewing large text files (q to quit)
<code>cat <i>filename1 filename2 filenameN</i></code>	concatenate text files in print output to terminal window
<code>head <i>filename</i></code>	print first 10 lines of a text file to terminal window
<code>head -n 20 <i>filename</i></code>	print first 20 lines of a text file to terminal window
<code>tail</code>	opposite of head
<code>wc <i>filename</i></code>	show the number of lines, words and characters in a text file
<code>cut -f 2 <i>tabDelimitedFilename</i></code>	extract 2nd column from a tab delimited file
<code>cut -f 3 -d , <i>comaSeperatedFilename</i></code>	extract 3rd column from a coma seperated file
<code>nano <i>filename</i></code>	open <i>filename</i> 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
<code>tar -cvf filename.tar filename</code>	compress file to .tar format
<code>tar -zcvf filename.tar.gz filename</code>	compress file to .tar.gz format
<code>tar -jcvf filename.tar.bz2 filename</code>	compress file to .tar.bz2 format
<code>zip filename.zip filename</code>	compress file to .zip format
<code>gzip filename</code>	compress file to .gz format

1.3.2 Uncompressing

Command	Explanation
<code>tar -xvf filename.tar</code>	uncompress from .tar format
<code>tar -zxvf filename.tar.gz</code>	uncompress from .tar.gz format
<code>tar -jxvf filename.tar.bz2</code>	uncompress from .tar.bz2 format
<code>unzip filename.zip</code>	uncompress from .zip format
<code>gzip -d file.gz</code>	uncompress from .gz format

1.4 Input/Output redirection

Command	Explanation
<code>command > filename</code>	Output of <i>command</i> is saved to <i>filename</i> , overwriting it
<code>command >> filename</code>	Output of <i>command</i> is appended at the end of <i>filename</i>
<code>command < filename</code>	<i>command</i> reads input from <i>filename</i>
<code>command1 command2</code>	<i>command2</i> takes the output of <i>command1</i> and produces result

1.5 Filters

Command	Explanation
<code>grep text filename</code>	Prints every line in <i>filename</i> containing <i>text</i>
<code>sed 's/red/green/' filename</code>	Prints every line in <i>filename</i> substituting word <i>red</i> with word <i>green</i>

1.6 Pattern matching

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

1.7 Miscellaneous

Command	Explanation
<code>echo text</code>	display a line of text
<code>history</code>	view your command line history
<code>wget someWebAddress</code>	download contents of <i>someWebAddress</i> to current directory
<code>make</code>	tool that is used to compile source code creating executables
<code>export name=value</code>	sets <i>value</i> to <i>name</i> . Type <code>echo \$name</code> to view <i>value</i>

2 Setup of the working environment

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_static.tar.bz2` 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 separate 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_Assembly
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**:

```
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 and copy resulting binary file to **~/binaries**)

4.2 Obtaining reference sequences

We will download reference sequences in a separate directory to avoid file cluttering. Let's make a new directory in our **ngs_work** directory named **reseq**, and there we will create a separate 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, type:

```
tmap map3 -f ref/chr_merged.fa \  
-r reseq_reads.fastq \  
-i fastq \  
-o 2 \  
-s reseq_reads_mapped.bam
```

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

Let's find out what is in the *BAM* file using `samtools`:

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

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

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

Alternatively, we could have defined the sample name during mapping process, using option `-R` with the complete tab delimited *RG* line of bam header:

```
tmap map3 -f ref/chr_merged.fa \  
-r reseq_reads.fastq \  
-R SM:Sample1
```

```
-R "@RG\tID:SomeID\tSM:Sample1" \
-i fastq \
-o 2 \
-s reseq_reads_mapped.reheaded.bam
```

Note that the tab is denoted as `\t` in our header line.

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. Variant callers need sorted by position *BAM* file to successfully detect variants. To sort our *BAM* file we will use `samtools`. In linux terminal type:

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

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 assumption 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.

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 will 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/`

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
```

To filter only those SNPs that are marked as nonsynonymous or stop gained, 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 how to execute these commands. One way 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 *Bourne-Again Shell* and is one of the most popular choices for performing shell scripting. Whenever we typed a command in linux terminal it was shell that was performing all the commands and giving as 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:

```
#!/bin/bash
tmap map3 -f ref/chr_merged.fa \
  -r reseq_reads.fastq \
  -R "@RG\tID:SomeID\tSM:Sample1" \
  -i fastq \
  -o 2 \
  -s reseq_reads_mapped.reheaded.bam

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

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

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
```