# Computer Laboratory

Adopted from Martin Dahlö martin.dahlo@scilifelab.uu.se

Software Introduction

This laboratory will run some of the more popular next generation sequencing (NGS) analysis programs on small datasets, to serve as an example when you are doing your real analysis on real data sometime in the future. It will cover the more general steps of analysis that more or less every project will have to do.

The lab will cover the following analysis types:

- Exome and whole genome sequencing (WGS)
  - Alignment with Bowtie or BWA
  - Conversion of file types with Samtools
  - SNP calling with Samtools
  - SNP annotation with wAnnovar
- Transcriptome sequencing
  - Alignment with Tophat
  - Transcript detection and quantification with Cufflinks
  - Differential expression with Cufflinks
- De novo sequencing
  - Genome assembly with Velvet

**For the impatient, all the commands for each topic is written down in a file in each topics subdirectory. The file is called <topic name>-cmds.txt**

Links to program homepages:

- Bowtie: http://bowtie-bio.sourceforge.net
- BWA: http://bio-bwa.sourceforge.net
- Tophat: http://tophat.cbcb.umd.edu
- Cufflinks: http://cufflinks.cbcb.umd.edu/
- Samtools: http://samtools.sourceforge.net
- Velvet: http://www.ebi.ac.uk/~zerbino/velvet/
- Annovar: http://www.openbioinformatics.org/annovar/
- wAnnovar: http://wannovar.usc.edu

# Preparation

### Get the data

First, you will have to get the datasets we will be working with. The lab instructor will show you how and where to download them.

**Unpack the data**

It will be a .tar.gz archive you will download, so the first thing you will do is to unpack the data:

*# downloaded the tutorial files .*

***$ wget http://hpc.ilri.cgiar.org/beca/training/ngs-tutorial.zip***

*# unpack the file*

***$ unzip ngs-tutorial.zip***

*# enter the unpacked tutorial directory*

***$ cd ngs-tutorial***

# Exome and WGS

The analysis steps for exome and WGS are more or less identical in the beginning of the analysis, so we will cover them both in these steps.

There are many different alignment programs to choose from, and we will cover two of them in this tutorial; Bowtie and BWA. They will both end up producing a BAM file, and after that the analysis is the same regardless of which aligner you used.

You can do both Bowtie and BWA if you would like to. The datasets are so small that it should not take more than 1-2 minutes for each aligner to align them.

The dataset in this part of the tutorial is a simulated single end 100bp reads WGS dataset from the adenovirus type 2 genome. Let's say we are interested in seeing if our strain of adenovirus differs from the previously annotated one.

**Alignment with Bowtie**

*# Enter the wgs-exome directory*

***$ cd wgs-exome***

The first thing you have to do with Bowtie is to index the reference genome you want to align to. The indexing allows the aligner to work much faster, and you only have to build the index if you do not already have it.

#first we need to module load the program bowtie

***$ module load bowtie***

*# build bowtie index*

***$ bowtie-build reference/adenovirus.fasta ../index/adenovirus-bowtie***

This command will build the index for the adenovirus reference genome, which is placed in the subdirectory called *references*. The index will be saved in the *index* directory located in the root of the tutorial directory (ngs-tutorial/index).

Once the index is finished building, it is time for the alignment step.

*# align reads with bowtie*

***$ bowtie -S ../index/adenovirus-bowtie reads/reads.fq > 1bowtie/alignment.sam***

This command will tell Bowtie to output a SAM file (-S), use the index called *adenovirus-bowtie* located in the *../index* directory, align the reads in the file *reads.fq* in the *reads* subdirectory, and save the result to the file *alignment.sam* in the *1bowtie* subdirectory.

Now, the reads are aligned and saved to a SAM file. The SAM format is a human readable text file which is very space inefficient compared to compressed binary formats. The BAM format is the same as the SAM format, the only difference being that it is compressed to save space.

#first we need to module load the program samtools

***$ module load samtools***

*# convert the sam file to a bam file*

***$ samtools view -hSb 1bowtie/alignment.sam > 1bowtie/alignment.bam***

This command will tell samtools to open a SAM file (-S), print the SAM header (-h), output the BAM format (-b), from the file 1bowtie/alignment.sam, and store the result in the file 1bowtie/alignment.bam

To be able to access the BAM file quickly, most programs require the BAM file to be sorted and indexed. This is easily done with Samtools.

*# sort the bam file*

***$ samtools sort 1bowtie/alignment.bam 1bowtie/alignment.sorted***

*# index the bam file*

***$ samtools index 1bowtie/alignment.sorted.bam***

*# rename the index to the same name as the bam file, but with a .bai extension*

***$ mv 1bowtie/alignment.sorted.bam.bai 1bowtie/alignment.sorted.bai***

You have now mapped your reads to the genome successfully with bowtie you can view the files by running ls on the 1bowtie folder

*#view the files created*

***$ ls 1bowtie***

**Alignment with BWA**

This section will describe how to align the reads with BWA. If you already have aligned the reads with Bowtie and have your 1bowtie/alignment.sorted.bam file, this part is not necessary (but still educational!).

*# Ensure you are in the wgs-exome directory*

***$ cd wgs-exome***

*#We need to load the bwa tool into our environment*

***$ module load bwa***

Like with Bowtie, the first thing you have to do is to index the references genome. Unfortunately, there is no standard way of indexing genomes, so each alignment program does it their own way.

*# build the bwa index*

***$ bwa index -p ../index/adenovirus-bwa -a is reference/adenovirus.fasta***

This command will tell BWA to save the index to the folder index in the nsg-tutorial root and call it adenovirus-bwa, to use the indexing algorithm called *is*, and use the file references/adenovirus.fasta as the references genome.

There are two different indexing algorithms in BWA; *is* and *bwtsw*. There are only two rules when choosing which algorithm to use:

- *is* can only index genomes smaller than 2 gigabases
- *bwtsw* can only index genomes larger than 10 megabases

Other than that, it should not matter too much which you use.

The step after indexing is alignment.

*# align the reads using the index file*

***$ bwa aln ../index/adenovirus-bwa reads/reads.fq > 1bwa/alignment.sai***

An odd thing with BWA is that it does not create a SAM file directly, instead it creates a .sai file which in turn can be used to create a SAM file.

*# create a sam file from the alignment*

***$ bwa samse ../index/adenovirus-bwa 1bwa/alignment.sai reads/reads.fq > 1bwa/alignment.sam***

This step will use the index file, .sai file, and the reads file to create a SAM file.

The SAM file will, just like in the Bowtie alignment, have to be converted to a BAM file to allow downstream analysis to be done fast enough.

*# convert the sam to a bam file*

***$samtools view -hSb  1bwa/alignment.sam > 1bwa/alignment.bam***

*# sort the bam file*

***$samtools sort 1bwa/alignment.bam 1bwa/alignment.sorted***

*# index the bam file*

***$samtools index 1bwa/alignment.sorted.bam***

*# rename the created bai file to match the bam files name*

***mv 1bwa/alignment.sorted.bam.bai 1bwa/alignment.sorted.bai***

## SNP calling

The next step is to locate all the SNPs in the data, and from this point on the analysis of the BAM files produced by either Bowtie or BWA is the same. Select one of the *alignment.sorted.bam* files to continue with. The following examples will be using the file from BWA because the name is shorter.

To search for SNPs faster, Samtools has to index the references genome as well.

*# ensure you are in the right directory*

***$cd***

***$cd ngs-tutorial/wgs-exome/***

*# index the reference genome*

***$ samtools faidx reference/adenovirus.fasta***

When the indexing is done, it is time to pick out **all the sites** where our sample differs from the reference genome. It creates a .fai file

#module load bcf tools

***$module load bcftools***

*# find all snps and send the result to bcftools to convert it to a .bcf file (binary format) (–uf uncompressed vcf formart, fasta reference)*

***$ samtools mpileup -uf reference/adenovirus.fasta 1bwa/alignment.sorted.bam | bcftools view -bvcg - > 2samtools/var-bwa.unfiltered.bcf***

*# since some sites might be false positives created by either chemistry or bad alignments, we want to avoid sites with abnormally high coverage. A decent value for this could be 2x average coverage.*

*# the example below uses 100 as the cutoff. To use 523 instead, use -D523 etc. The output will be saved in .vcf format (human readable).*

***$ bcftools view 2samtools/var-bwa.unfiltered.bcf | vcfutils.pl varFilter -D100 > 2samtools/var- bwa.filtered.vcf***

The *var-bwa.filtered.vcf* file can be opened by any text editor, or imported into Excel etc for easy viewing.

The next step after this would be to try to annotate the SNPs and see if they are previously know SNPs, if they are non-synonymous, creates a stop codon etc. A program called Annovar / SNPeff can be used for this, but the limited time in our tutorial will not allow this. However, there is some sample data in the *3annovar* folder which you can look at. It is the same as our *var-bwa.filtered.vcf* file, but based on human samples. This can be uploaded to the Annovar web service, wAnnovar. (http://wannovar.usc.edu/) if you want to see the kind of information you could get from Annovar.

## De Novo Assembly

The program used in this part of the tutorial will be Velvet. It will take all the reads you supply it with and try to assemble it to a complete genome.

Velvet consists of two programs; *velveth* and *velvetg*. *Velveth* will take the files with reads you specify and prepare a dataset to be used by *velvetg*. Apart from the name of the read files, it also takes the output path and the hash length as arguments.

The hash length is related to how *velveth* prepares the dataset, and has a significant impact on how good the final assembly by *velvetg* will be. Therefore it is good to try running your assemblies multiple times using different hash lengths.

The dataset used in this tutorial are simulated dataset of WGS of adenovirus type 2. There are 3 datasets; one with 10 000 100bp reads, one with 100 000 100bp reads, and one with 1000 450bp reads.

*#To start the tutorial, enter the denovo directory and run velveth.*

***$cd ~/ngs-tutorial/denovo/***

***$module load velvet***

*# run velveth for multiple hash lengths# a good coverage with short reads can get you a good way*

***$ velveth hash 21,33,2 -short -fastq reads/10000x100bp.fq***

This command will output each hash length in a directory called *hash_<hash length>* (this can be changed to anything really. Ex. $ velvet my_runs 21,33,2

.....), run with multiple hash lengths (start with 21, stop when reaching 33, just 2 steps each time, i.e. [21, 23, 25, 27, 29, 31]), and it will use the file 10000x100bp.fq which is a fastq file containing short reads (I.e not long, 454 reads).

The next step is to run *velvetg*. It has to be run once per directory created by *velveth,* 6 of them in this tutorial (one per hash length). One could go about this by running:

*# the boring way of doing it*

***$ velvetg hash_21 -exp_cov auto***

***$ velvetg hash_23 -exp_cov auto***

***$ velvetg hash_25 -exp_cov auto***

***$ velvetg hash_27 -exp_cov auto***

***$ velvetg hash_29 -exp_cov auto***

***$ velvetg hash_31 -exp_cov auto***

Or we could write a single line using the bash for-loop:

*# the smart way of doing it*

***$ for((n=21; n<33; n=n+2)); do velvetg hash_"$n" -exp_cov auto; done***

which will do the exact same thing.*Velvetg* has only one mandatory argument, which is which directory it can find the dataset prepared by *velveth*. In this case it is the directories name *hash_<hash length>*.There are many different extra arguments you can give to *velvetg,* and the Velvet manual (found on their homepage, http://www.ebi.ac.uk/~zerbino/velvet/) has good explanations of them all.

This is where the art of de novo assembly comes in. All genomes are different, and every sequencing run is unique, so there are no default settings that will always give you the best assembly. You simply have to try different things and see what works best. In this tutorial I have added the argument *-exp_cov auto* which will try to approximate the expected average coverage. If you know that your genome should be somewhere around 2 gigabases long, and you have 1 000 000 150bp reads, your average coverage **should** be 1 000 000 * 150 / 2 000 000 = 75. Supply it if you know it, otherwise set it to auto.

While running the *velvetg* command above, the screen will be flooded with information about each run. If you only want to know how many nodes (the minimum number of contigs that can be created using the reads. Ideally one contig per chromosome in the genome, which would be 1 contig for our viral genome.) you can use the following command instead:

*# same as above, but will only print out the 'scores' to the screen*

***$ for((n=21; n<33; n=n+2)); do velvetg hash_"$n" -exp_cov auto; done | grep Final***

This command will take all the output from *velvetg* and only print the rows that contains the word *Final*.

To see how the number of nodes can change depending on which dataset you run on, please run the reset.sh file to delete all the hash_<hash length> directories:

*# reset the tutorial*

***$ sh reset.sh***

Try running the previous steps using the different datasets:

*# too much cover can ruin the assembly*

***$ velveth hash 21,33,2 -short -fastq reads/100000x100bp.fq***

*# run velvetg for each hash length*

***$ for((n=21; n<33; n=n+2)); do velvetg hash_"$n" -exp_cov auto; done | grep Final***

This gives you the picture that more reads are not always a good things. If Velvet get too many reads, it can be confused and result in a worse assembly. This is also something you will have to experiment with and see which coverage suits your dataset, but try to keep between 20 and 100 fold coverage.

To see how good it works only using 454 data:

*# reset the tutorial*

***$ sh reset.sh***

*# only a few long reads will not get you so far*

***$ velveth hash 21,33,2 -long -fastq reads/1000x450bp.fq***

*# run velvetg for each hash length*

***$ for((n=21; n<33; n=n+2)); do velvetg hash_"$n" -exp_cov auto; done | grep Final***

And the best result is when combining short and long reads:

*# reset the tutorial*

***$ sh reset.sh***

*# a combination of good coverage short reads and a couple of long ones will be best*

***$ velveth hash 21,33,2 -short -fastq reads/10000x100bp.fq -long -fastq reads/1000x450bp.fq***

*# run velvetg for each hash length*

***$ for((n=21; n<33; n=n+2)); do velvetg hash_"$n" -exp_cov auto; done | grep Final***

When you have gotten an assembly you are happy with, you can find the file with the assembled sequence in the directory of the most successful hash length, named contigs.fa.

velvetg' produces a "stats.txt" file that can be examined to look at the k-mer coverage across the genome - or, rather, across the nodes in the de Bruijn graph that has been constructed. One way to look at this is using a script that was contributed to the Velvet distribution - velvet-estimate-exp_cov.pl:

#run velvet-estimate-exp_cov.pl

***$ /export/apps/velvet/1.2.10-kmer64/contrib/estimate-exp_cov/velvet-estimate-exp_cov.pl hash_25/stats.txt***

The script gives you a sideways distribution plot, where the vertical axis is k-mer coverage, and the horizontal axis is the count of nodes that have that k-mer coverage (averaged across all k-mers that make up the node). It also gives you suggestions for setting two very important options to the velvetg tool: expected coverage (-exp_cov) and coverage cutoff (-cov_cutoff). You must use these two options when calling velvetg if you want to use certain simplification routines on the graph. In addition, you must supply the insert size of fragments in your paired-end library (-ins_length) if you want to use paired end-specific routines in velvetg. Since we're pretending that our reads aren't paired for this first run, we'll just set

the -ext_cov and -cov_cutoff parameters as suggested by the Perl script we've just used, in a more optimized re-run of velvetg:

***$ for((n=21; n<33; n=n+2)); do velvetg hash_"$n" -exp_cov 9 –cov_cutoff 0 ; done | grep Final***

# Transcriptome

The RNAseq analysis in this tutorial will be Tophat and Cufflinks. Tophat is a program that uses the Bowtie aligner to align the reads, but also detect splice junctions. The reads covering a splice junction will be thrown away by ordinary aligners since there is nowhere in the references genome they fit in. Half the read will align to one exon, and the other half to another. Tophat on the other hand can find these reads and split them into smaller parts and align each part separately.

The datasets in this part of the tutorial are two simulated single end 100bp RNAseq reads from the human chromosome 21. We can pretend the two datasets are samples from two different tissue types, or a medical trial with one treated patient and one untreated control, and we are interested to see if the gene expression is different between them.

*# like before, bowtie will need to have an indexed reference genome.*

***$module load bowtie***

***$module load tophat***

***$module unload samtools/0.1.19***

***$module load cufflinks***

*# build bowtie index*

***$ bowtie-build reference/chr21.fa ../index/chr21***

When the index is built, align each of the sample to the reference genome.

*# align with tophat (use -p n where n=number of cores in the machine to speed things up (Ex: tophat -o tophat/0 -p 8 index/chr21 reads/0.fq))*

***$ tophat -o 1tophat/a ../index/chr21 reads/a.fastq***

***$ tophat -o 1tophat/b ../index/chr21 reads/b.fastq***

This will put the output from the alignment of the reads in a.fastq in the directory 1tophat/a, and the output from the alignment of the read in b.fastq in the directory 1tophat/b.

Since the default name of all the aligned reads from Tophat is accepted_hits.bam, it will be easy to confuse our two samples. The solution to this is to rename the files to something more descriptive.

*# rename files to avoid mixing them*

***$ mv 1tophat/a/accepted_hits.bam 1tophat/a/a.bam***

***$ mv 1tophat/b/accepted_hits.bam 1tophat/b/b.bam***

Now it is time to let Cufflinks look at the data. Cufflinks is a collection of programs, and the first step will take the alignments from Tophat and try to detect groups of reads that look like they belong to the same mRNA. It will even try to detect different splice forms of the same gene and try to quantify the individual splice forms expression. This step is done separately for each sample.

*# assemble transcripts with cufflinks*

***$ cufflinks -o 2cufflinks/a 1tophat/a/a.bam***

***$ cufflinks -o 2cufflinks/b 1tophat/b/b.bam***

When we have to transcripts from each sample, it is time to merge them info one single file which will be like the union of the transcript from the two samples. The expression from each sample will be stored separate columns in the file. The cuffmerge program takes a text file as input, which should contain the path of the files to be merged.

We could write it manually, but that could be painfully boring if there are many files. It is much faster and convenient to get the computer to do it for us.

*# create list of files to merge*

***$ find 2cufflinks/ -name transcripts.gtf > 3cuffmerge/merge.txt***

This command will search the directory *2cufflinks/* for files named *transcripts.gtf* and write the path to them. The > operator will take any output from the *find* program (i.e. the paths) and write it to the *merged.txt* file in the *3cuffmerge* directory. Now, to run the merging program:

*# merge results from cufflinks*

***$ cuffmerge -s reference/chr21.fa -o 3cuffmerge 3cuffmerge/merge.txt***

This will merge all the files specified in merge.txt in the 3cuffmerge directory, and include some information about the references genome.

The next step is optional, but highly recommended. It will take annotations of the references genome in gtf format and try to match all detected transcript to it by looking at where they are located in the genome. If a match is found, the name of the gene will be included in the results.

*# annotate merged.gtf with chr21 refseq annotations (optional step)*

***$ cuffcompare -o 4cuffcompare/merged -s reference/chr21.fa -r reference/chr21.gtf 3cuffmerge/merged.gtf***

This command will store all its output in 4cuffcompare, using merged as prefix for all the files, guided by the fasta reference genome and its gtf annotations, trying to assign genes to the transcripts found by the earlier steps.

The last step in the analysis is to detect differential expression between the samples. The program cuffdiff will do that for us.

*# detect differential expression*

***$ cuffdiff -o 5cuffdiff/ -T 4cuffcompare/merged.combined.gtf 1tophat/a/a.bam 1tophat/b/b.bam***

This command will store all output in the 5cuffdiff directory, use the merged and annotated transcripts in merged.combined.gtf, and use the expression based on the reads in a.bam and b.bam.

This analysis will produce many files in the 5cuffdiff directory, which you can read about in more detail in the programs manual, http://cole-trapnell-lab.github.io/cufflinks/manual/

To see the differential expression of all found transcripts, look in gene_exp.diff or isoform_exp.diff. They are tab separated text files with one row per found gene/transcript, which can easily be imported into Excel etc for further analysis or sorting.

*#Sort the *exp.diff files by p-value to bring the DE genes to the top*

***$sort -g -k12 5cuffdiff/gene_exp.diff > 5cuffdiff/sorted_gene_exp.diff***

***$sort -g -k12 5cuffdiff/isoform_exp.diff > 5cuffdiff/sorted_isoform_exp.diff***

# Differential expression with edgeR

DESeq is a Bioconductor R package for calculating differential expression. In addition to performing pairwise comparison (like a vs b), it can also handle multi-factorial models (see their documentation for more on this). Let's re-run our a vs b analysis with deseq. First we need to generate a table of raw counts. We'll use the python program htseq-count:

*#load htseq*

***$module load htseq***

***$cd 6htseq***

***$ cp ../1tophat/a/a.bam .***

***$ cp ../1tophat/b/b.bam .***

***$ cp ../4cuffcompare/merged.combined.gtf .***

*#Run htseq to count how many reads map to each feature in dataset a*

***$ htseq-count -f bam -r pos -s no a.bam merged.combined.gtf > a.counts***

*#Run htseq to count how many reads map to each feature in dataset b*

***$ htseq-count -f bam -r pos -s no b.bam merged.combined.gtf > b.counts***

You can take a look at (with less or head at the counts file) we now need to combine them into a table for edgeR

*#combine dataset a and b*

***$ echo "id a b" | sed 's/ /\t/g' > raw_counts.txt***

***$join a.counts b.counts | sed 's/ /\t/g' | grep -v "^_" >> raw_counts.txt***

*#let us practice on edge R*

***$module load R***

***$R***

*#Install edge R package from biconductor*

***$source("http://bioconductor.org/biocLite.R")***

***$biocLite("DESeq")***

*#load the deseq R package*

***$library(DESeq)***

*#read in the count  data and run head to see it*

***$counts = read.table("raw_counts.txt", header=T, row.names=1)***

*#load in the experimental design provided in your enviroment*

***$ expdesign = read.table("expdesign.txt")***

*# the counts that were loaded as a data.frame are now used to create a new type of object: count data set*

***$cds = newCountDataSet(counts, expdesign$condition)***

*#Lets estimate the size factor based on the number of aligned reads from each sample.*

***$cds = estimateSizeFactors(cds)***

*#Estimate dispertion*

***$cds = estimateDispersions( cds , method="blind", sharingMode="fit-only", fitType = "local")***

*#visualise the dispersion*

***dispersionFile = "Dispersion.pdf"***
***pdf(dispersionFile)***
***plotDispEsts( cds )***
***dev.off()***

*#Finally to perform the negative binomial test on the dataset to identify differentially expressed genes.*

```
$res = nbinomTest( cds, "untreated","treated")

#An MA plot allows us to see the fold change vs level of expression. In the plot, the
red points are for genes that have FDR of 10%.

maFile = "MAplot.pdf"
pdf(maFile)
plotMA(res)
dev.off()

#upregulated genes

$resSigind = res[ which(res$padj < 0.1 & res$log2FoldChange > 1), ]

#downregulated genes

$resSigrep = res[ which(res$padj < 0.1 & res$log2FoldChange < -1), ]

# save results as a table

indoutfile = "Deseq.indresults.txt"
repoutfile = "Deseq.represults.txt"

write.table(resSigind,
        indoutfile,
        sep="\t",
        col.names=T,
        row.names=F,
        quote=F)

write.table(resSigrep,
        repoutfile,
        sep="\t",
        col.names=T,
        row.names=F,
        quote=F)

Finished
```