Practical Workshop: de novo genome assembly

Author: Yung Shwen Ho
Contact: ho.shwen@kaust.edu.sa

Introduction

The aim of this workshop is to introduce the basic workflow for *de novo* genome assembly using Velvet and highlight some key steps within the process which you can manipulate to improve the final result.

Setting up the environment

You will be working in the virtual environment along with some data file that will be given to you during the workshop. Those data files will have to be copied into your virtual box share folder before you can continue beyond this point.

Compiling Velvet

The virtual machine has velvet pre-installed for this workshop. However, it is possible that you might have to install velvet by yourself in the future. Thus we will go through the process of unzipping and compiling the software in the first exercise.

```
cd /home/b308/Desktop/Practicals/01_compiling_velvet
tar -xzvf velvet_1.2.10.tgz
cd ./velvet
make
ll -rt
```

Note: `all -rt` will sort the files in the directory and list the contents in an ascending order of their modified date. Thus, the most recently added files at listed at the bottom.

Which new files have been created?

Choosing the right hash length is crucial to obtain optimal assemblies. Depending on the dataset, you might wish to use long hash lengths. By default, velvet is compiled with a maximum of k-mer 31, but you can increase this limit by adjusting the MAXKMERLENGTH parameter. The example below will allow velvet to run with a maximum k-mer length of 75bp. However, by storing longer words, Velvet will be requiring more memory, so adjust this variable according to your needs and memory resources.

```
make MAXKMERLENGTH=75
```

Sometimes you might want to use libraries having different insert sizes or having different samples altogether. By default, there are only two short read categories, but this variable can be extended to your needs. In that case, you will need to compile it with the `CATEGORIES` flag. The example below will allow for 10 different libraries.

```
make CATEGORIES=10
```

Read lengths are stored on signed 16bit integers, meaning that if you assemble contigs longer than 32kb long, then more memory is required to store coordinates. In that case, you will need to compile it with the `LONGSEQUENCES` flag.

```
make LONGSEQUENCES=1
```

How about combining them all?

```
make CATEGORIES=10 LONGSEQUENCES=1 MAXKMERLENGTH=75
```

There are 2 main programs that you will need to run:

1) velveth – reads processing
2) velvetg – de Bruijn graph assembly

Let us look at what they are by executing the command.

```
./velveth
./velvetg
```

The executable files are currently stored in the `"01_compiling_velvet" folder.` To make it accessible from the rest of your folders, you can either add that to your PATH variable or create a symbolic link to those 2 executable programs where you will most likely use them. We will not be demonstrating either of them as I have preinstalled a version of velvet for you in the virtual environment.

Creating your first assembly

Now that you know how to configure and install velvet, let us move ahead and generate your first *de novo* genome assembly. You will have to access the data files stored in the "02_default_assembly" folder by moving to that location. You will be doing all your assembly there. Velveth is the first command you will need to run to pre-process the reads for assembly. Can you guess what does each of the parameter means?

```
cd /home/b308/Desktop/Practicals/02_default_assembly
ls -l
velveth single_51 51 -short -fastq.gz single.fastq.gz
```

Let us see what has been created by velveth. Go into the directory single_51 that has been created.

```
cd ./single_51
ls -lh
cat Log
head Sequences
```

What is recorded in the log file?

What information can you see in the Sequence file?

How big is the Sequence file?

Now, let us generate the assembly by using the velvetg command. The minimum parameter you have to provide in velvetg is the folder which you created using velveth.

```
cd ..
time velvetg single_51
```

Let us see what is the result of the final assembly.

```
cat ./single_51/Log
```

What is the maximum contig size for our assembly?

What is the total assembly size?

What is the n50?

How many contigs are there in the assembly?

What if we were to try different K-mer size?

```
velveth single_71 71 -short -fastq.gz single.fastq.gz
velvetg single_71
cat ./single_71/Log
```

What are the differences in the assembly stats?

	Assembly with K-mer 51	Assembly with K-mer 71
Total assembly Size		
Largest coting size		
N50 size		
Number of contigs		

If paired-end reads are available, it will help better with the assembling process. Let us see how the assembly will look if we were to use a normal Illumina Paired end library from the same library as single.fastq.gz.

```
velveth paired_71 71 –fastq.gz –separate -shortPaired fwd.fastq.gz
rev.fastq.gz

velvetg paired_71

cat ./paired_71/Log
```

What are the differences in the assembly stats?

	Single-end reads only	Paired-end reads
Total assembly Size		
Largest contig size		
N50 size		
Number of contigs		

Let us have an even closer look at the assembly stats other than N50, total size and largest contig size.

```
astat ./paired_71/contig.fa
```

The astat command is actually a shortcut to another command in the PAGIT toolkit (which we will discuss later). It gives us more stats than the default Velvet assembler log. It might be interesting for you to find out where this shortcut points. But let us focus on the output from astat.

What does N80 represent?

What does the n = x number mean for each line?

What is the N80 of ./paired_71/contig.fa ?

What is the N100 of ./paired_71/contig.fa ?

Most likely if you ran velvet by default, it would have generated many small fragments (about selected K-mer size x 2) and are not usually useful. Let us see if we can filter those small fragments out of the assembly and calculate the statistics for contigs greater than 1000bp.

```
astat -l 1000 ./paired_71/contig.fa
```

What are the differences in the assembly stats?

	Paired-end (default)	Paired-end (default) min100bp
Total assembly Size		
Largest contig size		
N50 size		
Number of contigs		

It is quite likely that the short contigs accounted for most of the fragments within the assembly. They are probably noise within the assembly and removing them at an earlier stage will help you with your workflow. You can automatically filter out short contigs by putting this option "`-min_contig_lgth 1000`" when you run velvetg. There are quite a few options which you can tweak within velvetg and you can learn more about them either in the Velvet manual or by running velvetg without any other parameter. However we will look at how we can improve the assembly by setting the -cov_cutoff and –exp_cov parameters.

```
mv ./paired_71/contig.fa ./paired_71/contig.fa.0
velvetg paired_71 -cov_cutoff auto -exp_cov auto -min_contig_lgth 1000
astat ./paired_71.contig.fa
```

What is the velvet estimated expected coverage ?

What is the velvet estimated coverage cutoff value?

What are the improvements when we use set the –cov_cutoff and-exp_cov parameters?

There are other ways to assess the quality of the assembly instead of N50s. As the reads were used in the assembly, they should all be mapped onto the final assembly and "properly paired". A proper pairing read is a read that is mapped in the correct (facing towards each other) orientation with its other mated read and is within the expected insert size distance. In order to find out about that, we can use an aligner to map the reads back onto the genome.

```
cd ./paired_71/
bwa index contigs.fa
bwa aln contigs.fa ../fwd.fastq.gz > fwd.sai
bwa aln contigs.fa ../rev.fastq.gz > rev.sai
bwa sampe contigs.fa fwd.sai rev.sai ../fwd.fastq.gz ../rev.fastq.gz >
    contigs.sam
samtools view -bS -T contigs.fa contigs.sam > contigs.bam
samtools sort contigs.bam contigs.sorted
samtools index contigs.sorted.bam
samtools flagstat contigs.sorted.bam
```

How many reads are mapped onto the genome?

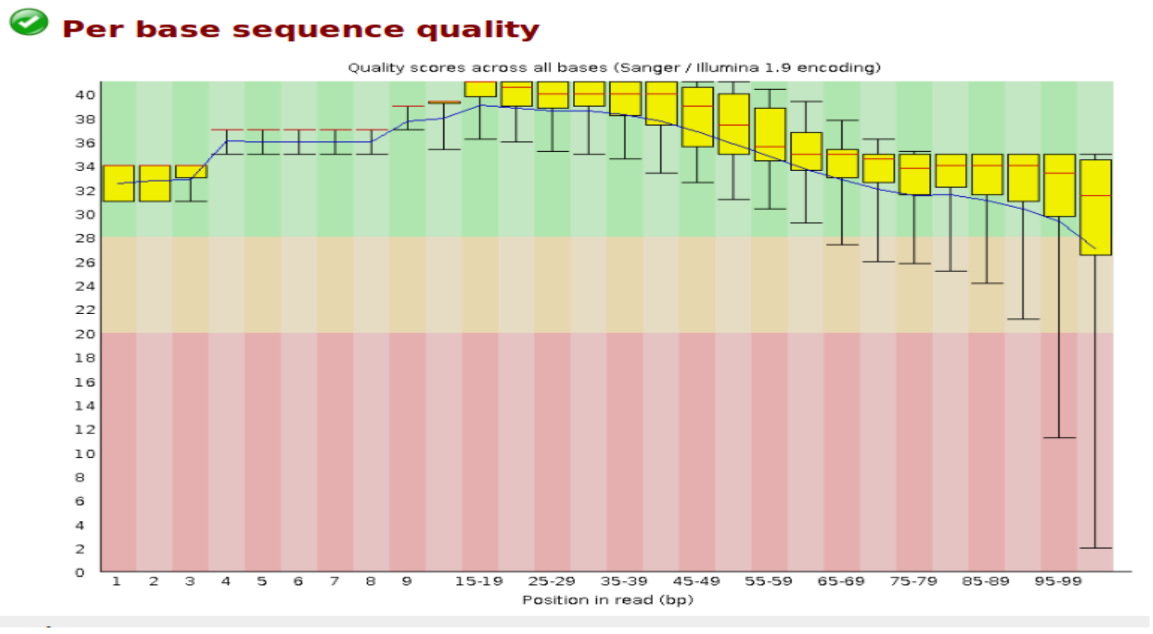
How many mapped reads are properly paired? How many percent of the total reads does that represent?

You might want to try repeating this step for the other assemblies you have created so far. It is always a good idea to calculate these stats as part of your assembly workflow.

One thing that we haven't surveyed till now is how velvet handles the quality scores within the FastQ file. Unfortunately, **velvet does not take into account quality information at all.** That means that it is left to you to preprocess your data before assembling them. In order to understand the quality profile of your initial data, we will first do a health check.

```
cd ..
fastqc fwd.fastq.gz
fastqc rev.fastq.gz
firefox ./fwd_fastqc/fastqc_report.html
```

the firefox command will launch the web-browser and open up the fastqc report for fwd.fastq.gz.



From the quality plot above, you can see that the quality drops below Q20 at the end of the read after base 90. Those poor quality bases at the end of each read might create some error within our assembly. There are various ways to remove those erroneous bases but we will concentrate on using the tool Trimmomatic to dynamically remove those low quality bases.

```
java -classpath ~/softwares/Trimmomatic-0.22/trimmomatic-0.22.jar
      org.usadellab.trimmomatic.TrimmomaticPE  -phred33
      fwd.fastq.gz rev.fastq.gz p.1.fastq up.1.fastq
      p.2.fastq up.2.fastq LEADING:3 TRAILING:3 SLIDINGWINDOW:4:15
      MINLEN:50
```

You can find out more about Trimmomatic and see what each of the options (LEADING, TRAILING, SLIDINGWINDOW and MINLEN) means by going to this link:
http://www.usadellab.org/cms/index.php?page=trimmomatic

Trimmomatic created 4 files p.1.fastq, p.2.fastq, up.1.fastq and up.2.fastq. Files prefixed with "p." contained trimmed reads where their mates are still present after the filtering process. This is what you want to use. Files prefixed with "up." represent reads that had their other mate removed and they are not singleton. We will not be using it for this workshop and we can remove them to save some space. However, you might want to keep them somewhere for your own assembly. We are also going to zip up the "p" files to save some space.

```
rm up.*.fastq
gzip p.1.fastq
gzip p.2.fastq
fastqc p.1.fastq.gz
firefox ./p.1_fastqc/fastqc_report.html
```

How many paired reads are left after trimming?

How many singleton reads are left after trimming?

Now that we have the trimmed data files, Let us try running the assembly with the new dataset.

```
velveth trim_paired_71 71 -fastq.gz -separate -shortPaired p.1.fastq.gz
       p.2.fastq.gz

velvetg trim_paired_71 -cov_cutoff auto -exp_cov auto -min_contig_lgth
       1000

astat ./trim_paired_71/contig.fa
cd ./trim_paired_71
bwa index contigs.fa
bwa aln contigs.fa ../p.1.fastq.gz > fwd.sai
bwa aln contigs.fa ../p.2.fastq.gz > rev.sai
bwa sampe contigs.fa fwd.sai rev.sai ../p.1.fastq.gz ../p.2.fastq.gz >
    contigs.sam
samtools view -bS -T contigs.fa contigs.sam > contigs.bam
samtools sort contigs.bam contigs.sorted
samtools index contigs.sorted.bam
samtools flagstat contigs.sorted.bam
```

What are the differences in the assembly stats?

	Paired-end (no trim)	Paired-end (trimmed)
Total assembly Size		
Largest contig size		
N50 size		
Number of contigs		
Percentage of PP mapped reads		

Practical Workshop: Genome assembly improvement

Author: Yung Shwen Ho
Contact: ho.shwen@kaust.edu.sa

Introduction

The aim of this section is to show you the familiarized with some basic workflow for genome assembly improvement and validation.

Where we were previously

You were introduced to velvet and some basic steps that you can take to improve the outcome of the assembly. However that is often not the final genome that you will be publishing. This is because there are often errors within the genome assembly that needs to be fixed. There will also be regions within the genome that can still be improved.

Scaffolding of the genome assembly

If you have access to either short pair-end or long mate-pair libraries, you can use the pairing information to stitch contigs into scaffold by estimating their distances. This can be incredibly useful to improve your assembly. Let us try to scaffold our previously assembly. First copy your final contig.fa file from the previous lesson to the new directory.

```
cd /home/b308/Desktop/Practicals/03_Scaffold_exercise
cp ../02_default_assembly/trim_paired_71/contig.fa step02.fa
mv ../02_default_assembly/p.1.fastq.gz .
mv ../02_default_assembly/p.2.fastq.gz .
```

We will be using the program SSPACE to scaffold our assembly and you can learn more about it from this link: http://www.baseclear.com/landingpages/basetools-a-wide-range-of-bioinformatics-solutions/sspacev12/

Before you can run SSPACE, you will need to inform SSPACE about some information regarding your libraries. The filename, their insert size information, expected insert size error you expect to see within your library and their orientation.

We will have to unzip the data files as SSPACE will not process zip files.

```
gunzip p.1.fastq.gz
gunzip p.2.fastq.gz
```

Create a file call sspace.config.txt by calling the text editor gedit

```
gedit sspace.config.txt
```

and put in the following line (be mindful of the spaces):

lib p.1.fastq p.2.fastq 350 0.25 FR

Next, we can run the SSPACE program and attempt to scaffold the previous assembly. The final scaffold assembly is stored in the file standard_output.final.scaffolds.fasta

```
SSPACE_Basic_v2.0.pl -l sspace.cnfig.txt  -s step02.fa
cat standard_output.summaryfile.txt
```

What is the maximum scaffold size for our assembly?

What is the total assembly size?

What is the Scaffold n50?

How many Scaffolds are there in the assembly?

How many N-bases are there in the assembly?

How does every stat compares with the previous contigs?

Closing the assembly gaps

In order to close the gaps within your scaffolds, you can use GapFiller or Image tool to close them. Both work in a similar manner where reads are mapped to the edges to the gaps or edges (for the contig extension option) of the contig. Reads that are successfully mapped are then used to fill in the missing bases. This process is repeated multiple times to close bigger gaps. In this exercise, we will use GapFiller to close the gaps that were created from the scaffolding assembly that you have generated earlier.

```
cd /home/b308/Desktop/Practicals/04_Gap_fill_exercise
cp ../03_Scaffold_exercise/standard_output.final.scaffolds.fasta step03.fa
cp ../03_Scaffold_exercise/p.1.fastq .
cp ../03_Scaffold_exercise/p.2.fastq .
```

Create a file call gapfill.config.txt by calling the text editor gedit

```
gedit gapfill.config.txt
```

and put in the following line (be mindful of the spaces):

lib bowtie p.1.fastq p.2.fastq 350 0.25 FR

Next, we can run the gapfiller program to fill in the assembly gaps with the paired – end data. The final scaffold assembly is saved in: standard_output.final.scaffolds.fasta .

```
GapFiller.pl -l gapfill.config.txt -s step_03.fa
cat standard_output/standard_output.summaryfile.final.txt
```

How many cycle did GapFiller run?

How many gaps did GapFiller close within the first cycle?

How many single N's remain after the execution?

Are there any changes to the other assembly stats?

Visualizing the genome assembly and identifying errors.

Now that you have filled in the gaps, we should visualize the genome in the context of the published reference genome to have an overview picture of your assembly. To do that, we first have to order and orientate our scaffold according to the reference genome. This way, we will be able to compare them effectively. To do that, we will be using the tool abacas from PAGIT.

```
cd /home/b308/Desktop/Practicals/05_Visualize_1_exercise
cp ../04_Gap_fill_exercise/standard_output/
    standard_output.gapfilled.final.fa step04.fa
abacas.pl -r FN649414.fasta -q step04.fa -p nucmer -a -c
```

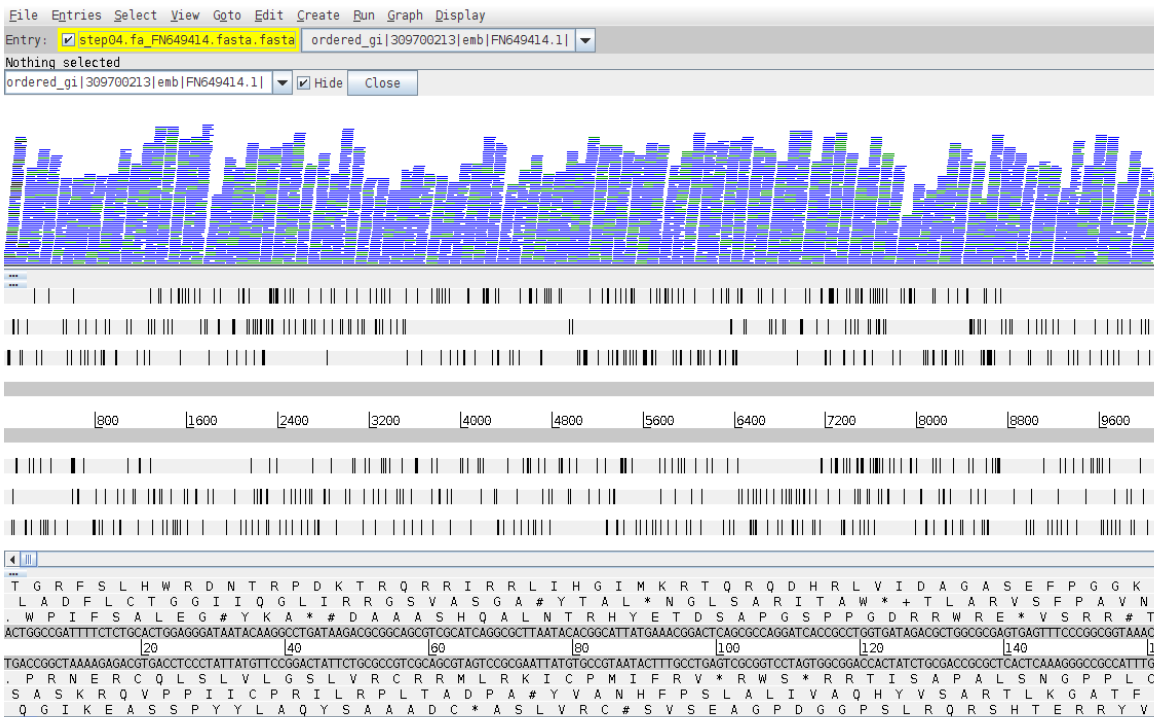
Abacas will create a pseudo molecule based on the reference you have provided. The file that you can use is: step04.fa_FN649414.fasta.fasta . In addition, Abacas will provide you with a crunch comparison file: step04.fa_FN649414.fasta.crunch where we will be using later to compare it to the reference. It is a good idea now to map the reads back to both the reference genome and pseudo molecule.

```
cd /home/b308/Desktop/Practicals/05_Visualize_1_exercise
cp ../04_Gap_fill_exercise/standard_output/
    standard_output.gapfilled.final.fa step04.fa
abacas.pl -r FN649414.fasta -q step04.fa -p nucmer -a -c

cp ../03_Scaffold_exercise/p.1.fastq .
cp ../03_Scaffold_exercise/p.2.fastq .


bwa index step04.fa_FN649414.fasta.fasta
bwa aln step04.fa_FN649414.fasta.fasta p.1.fastq.gz > fwd.sai
bwa aln step04.fa_FN649414.fasta.fasta p.2.fastq.gz > rev.sai
bwa sampe step04.fa_FN649414.fasta.fasta fwd.sai rev.sai p.1.fastq
     p.2.fastq.gz > pseudo.sam

samtools view -bS -T step04.fa_FN649414.fasta.fasta pseudo.sam >
      pseudo.bam
samtools sort pseudo.bam pseudo.sorted
samtools index pseudo.sorted.bam
samtools flagstat pseudo.sorted.bam > pseudo.sorted.bam.stats


bwa index FN649414.fasta
bwa aln FN649414.fasta p.1.fastq.gz > fwd.sai
bwa aln FN649414.fasta p.2.fastq.gz > rev.sai
bwa sampe FN649414.fasta fwd.sai rev.sai p.1.fastq p.2.fastq.gz >
      reference.sam

samtools view -bS -T FN649414.fasta reference.sam > reference.bam
samtools sort reference.bam reference.sorted
samtools index reference.sorted.bam
samtools flagstat reference.sorted.bam > reference.sorted.bam.stats
```

Now that you have two bam files (`pseudo.sorted.bam`, `reference.sorted.bam`), we can now open them in artemis to visualize the genome. You can either click the shortcut button to Artemis on the side task bar or enter this command line to launch artemis.

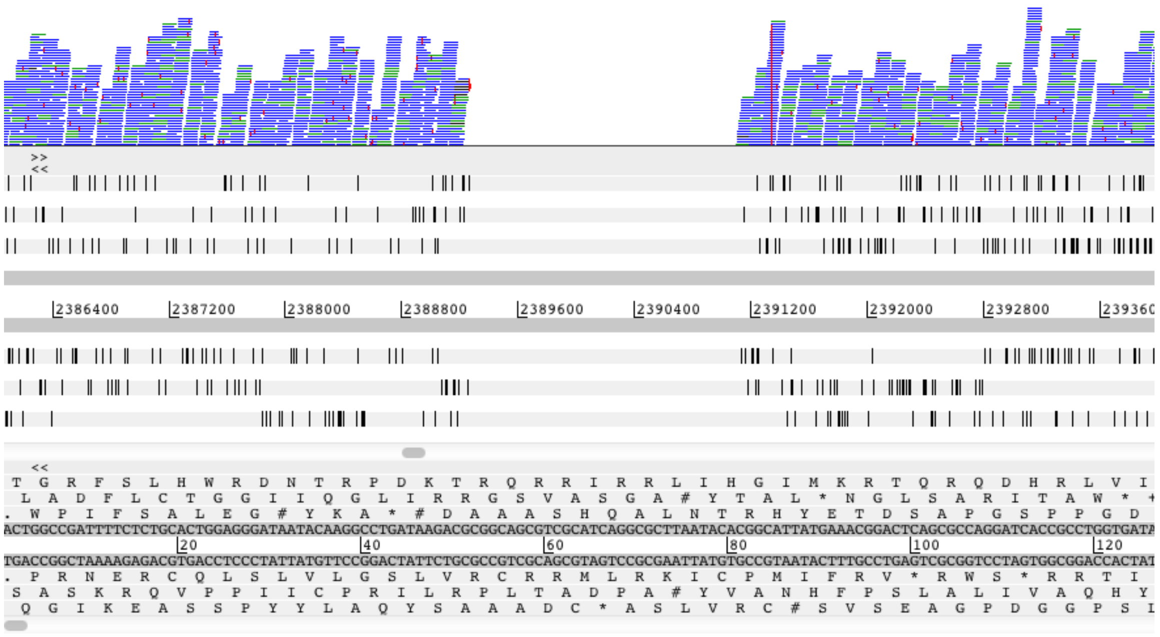
```
art
```

After artemis has started, you can open the fasta file you have prepared earlier and load the pseudo.sorted.bam file onto the assembly. You should see something similar to this.

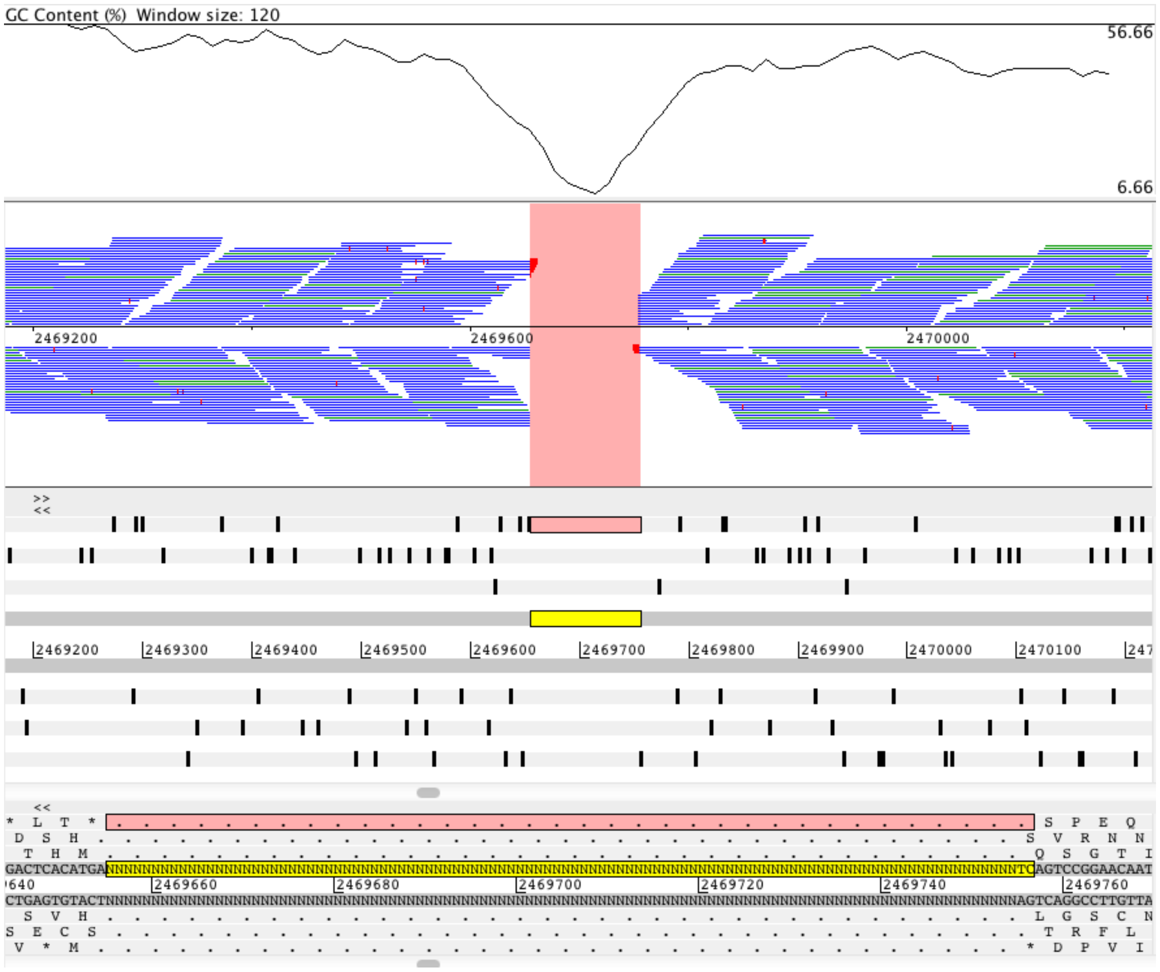

File Entries Select View Goto Edit Create Run Graph Display
Entry: step04.fa_FN649414.fasta.fasta
ordered_gi|309700213|emb|FN649414.1|
Nothing selected
ordered_gi|309700213|emb|FN649414.1|
Hide
Close
800
1600
2400
3200
4000
4800
5600
6400
7200
8000
8800
9600

```
T  G  R  F  S  L  H  W  R  D  N  T  R  P  D  K  T  R  Q  R  R  I  R  R  L  I  H  G  I  M  K  R  T  Q  R  Q  D  H  R  L  V  I  D  A  G  A  S  E  F  P  G  G  K
 L  A  D  F  L  C  T  G  G  I  I  Q  G  L  I  R  R  G  S  V  A  S  G  A  #  Y  T  A  L  *  N  G  L  S  A  R  I  T  A  W  *  +  T  L  A  R  V  S  F  P  A  V  N
.  W  P  I  F  S  A  L  E  G  #  Y  K  A  *  #  D  A  A  A  S  H  Q  A  L  N  T  R  H  Y  E  T  D  S  A  P  G  S  P  P  G  D  R  R  W  R  E  *  V  S  R  R  #  T
ACTGGCCGATTTTCTCTGCACTGGAGGGATAATACAAGGCCTGATAAGACGCGGCAGCGTCGCATCAGGCGCTTAATACACGGCATTATGAAACGGACTCAGCGCCAGGATCACCGCCTGGTGATAGACGCTGGCGCGAGTGAGTTTCCCGGCGGTAAAC
                  20                  40                  60                  80                  100                 120                 140
TGACCGGCTAAAAGAGACGTGACCTCCCTATTATGTTCCGGACTATTCTGCGCCGTCGCAGCGTAGTCCGCGAATTATGTGCCGTAATACTTTGCCTGAGTCGCGGTCCTAGTGGCGGACCACTATCTGCGACCGCGCTCACTCAAAGGGCCGCCATTTG
.  P  R  N  E  R  C  Q  L  S  L  V  L  G  S  L  V  R  C  R  R  M  L  R  K  I  C  P  M  I  F  R  V  *  R  W  S  *  R  R  T  I  S  A  P  A  L  S  N  G  P  P  L  C
 S  A  S  K  R  Q  V  P  P  I  I  C  P  R  I  L  R  P  L  T  A  D  P  A  #  Y  V  A  N  H  F  P  S  L  A  L  I  V  A  Q  H  Y  V  S  A  R  T  L  K  G  A  T  F
  Q  G  I  K  E  A  S  S  P  Y  Y  L  A  Q  Y  S  A  A  A  D  C  *  A  S  L  V  R  C  #  S  V  S  E  A  G  P  D  G  G  P  S  L  R  Q  R  S  H  T  E  R  R  Y  V
```

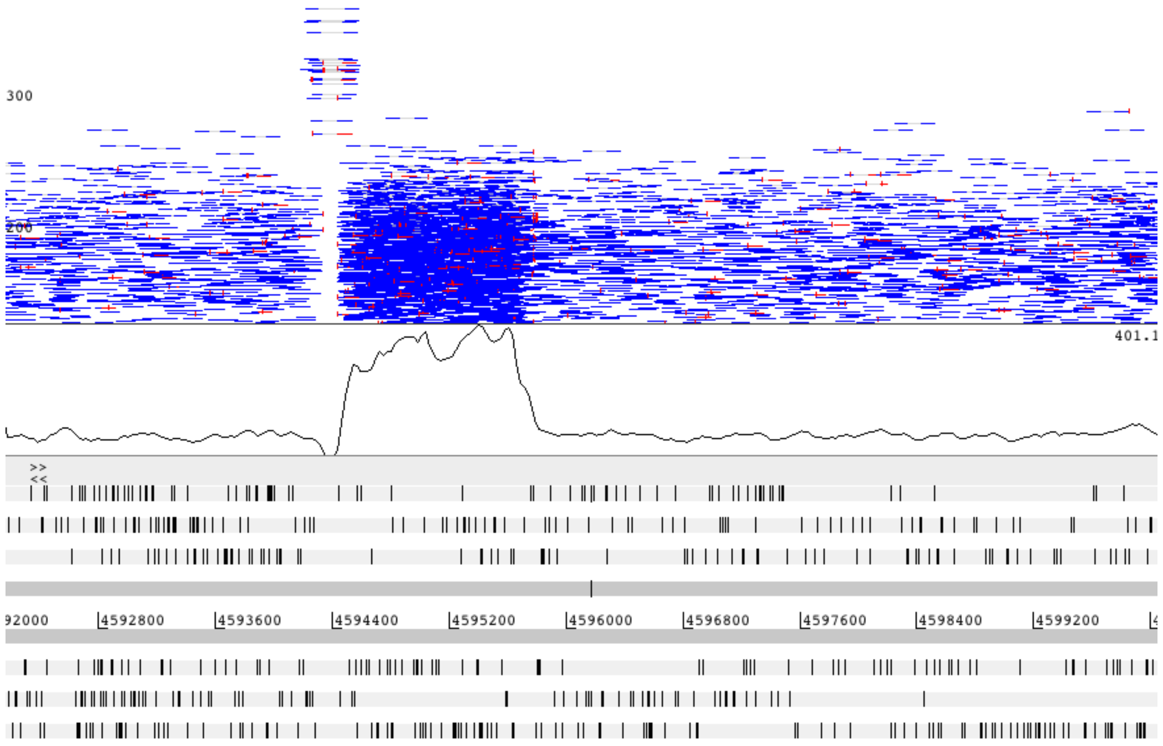
Turn on the feature “Mark ambiguity” , strand stack visualization with SNP marking. Scroll through the pseudo molecule and see if you can identify any assembly error. Example of SNPs within the pseudo molecule can be seen in the images below:



```
T  G  R  F  S  L  H  W  R  D  N  T  R  P  D  K  T  R  Q  R  R  I  R  R  L  I  H  G  I  M  K  R  T  Q  R  Q  D  H  R  L  V  I
 L  A  D  F  L  C  T  G  G  I  I  Q  G  L  I  R  R  G  S  V  A  S  G  A  #  Y  T  A  L  *  N  G  L  S  A  R  I  T  A  W  *  +
.  W  P  I  F  S  A  L  E  G  #  Y  K  A  *  #  D  A  A  A  S  H  Q  A  L  N  T  R  H  Y  E  T  D  S  A  P  G  S  P  P  G  D
ACTGGCCGATTTTCTCTGCACTGGAGGGATAATACAAGGCCTGATAAGACGCGGCAGCGTCGCATCAGGCGCTTAATACACGGCATTATGAAACGGACTCAGCGCCAGGATCACCGCCTGGTGATA
                  20                  40                  60                  80                  100                 120
TGACCGGCTAAAAGAGACGTGACCTCCCTATTATGTTCCGGACTATTCTGCGCCGTCGCAGCGTAGTCCGCGAATTATGTGCCGTAATACTTTGCCTGAGTCGCGGTCCTAGTGGCGGACCACTAT
.  P  R  N  E  R  C  Q  L  S  L  V  L  G  S  L  V  R  C  R  R  M  L  R  K  I  C  P  M  I  F  R  V  *  R  W  S  *  R  R  T  I
 S  A  S  K  R  Q  V  P  P  I  I  C  P  R  I  L  R  P  L  T  A  D  P  A  #  Y  V  A  N  H  F  P  S  L  A  L  I  V  A  Q  H  Y
  Q  G  I  K  E  A  S  S  P  Y  Y  L  A  Q  Y  S  A  A  A  D  C  *  A  S  L  V  R  C  #  S  V  S  E  A  G  P  D  G  G  P  S  I
```



If you turn on insert size distribution, you might be able to find regions in the genome that were misassembled.



Which region in the genome has a single SNP?

Are there any particular region of the genome where SNPs clusters

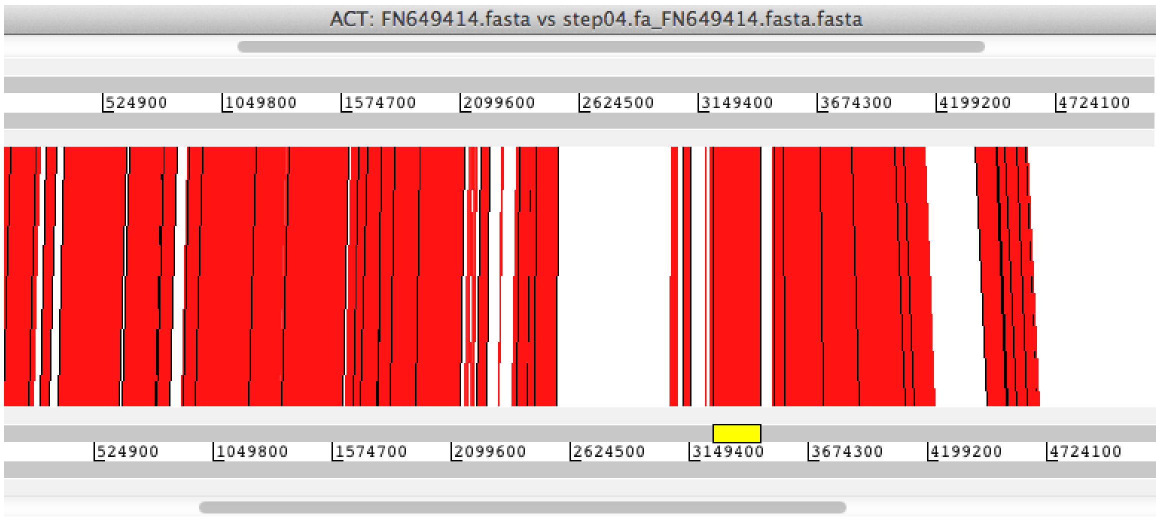
Let us see how the genome looks when compared to the reference genome. You can activate act from the shortcut on the left taskbar or enter this command line to launch act.

```
act
```

in the next window, select the following files for comparison:

Sequence file 1:	FN649414.fasta
Comparison file 1:	step04.fa_FN649414.fasta.crunch
Sequence file 2:	step04.fa_FN649414.fasta.fasta

You should see something similar to the image below. ACT allows you to have an overview of how the assembly compares to your reference genome.



Correcting assembly error using reapr.

Now that we have identified some errors, Let us see if we can correct them. The pseudo molecule that you have just created is mainly used for visualizing the genome. It is better to use the last assembly file that you have created from the GapFiller stage. This process will take a very long time and it might be worth leaving it overnight if you were working on your own data.

```
cd /home/b308/Desktop/Practicals/06_Correct_errors
cp ../04_Gap_fill_exercise/standard_output/
    standard_output.gapfilled.final.fa step04.fa
cp ../03_Scaffold_exercise/p.1.fastq .
cp ../03_Scaffold_exercise/p.2.fastq .
reapr facheck step04.fa step04_checked
reapr smaltmap step04_checked.fa p.1.fastq p.2.fastq mapping.bam
reapr pipeline step04_checked.fa mapping.bam outdir
```

How many bases are corrected by reapr?

How many scaffold did reapr break?

It is always good visualize the final corrected assembly by mapping the reads back onto the result fasta file and seeing it in Artemis to confirm if the reapr has not left out anything. You can repeat the previous step above to redo the visualization process or write this up as a script for ease.