

# 1 NGS Data Formats

## 1.1 Introduction

In this tutorial we will introduce several common data formats used for sequence data. We will cover the following formats:

**FASTA** - This format is used to store nucleotide sequences

**FASTQ** - This format is used to store nucleotide sequences and corresponding quality scores

**SAM/BAM** - This format is used to store unaligned or aligned (matched to a reference genome) nucleotide sequences

**CRAM** - This format is similar to BAM but has better compression than BAM

**VCF/BCF** - This format is used to store sequence variation (SNPs, indels, structural variations)

**GFF** - This format is used to store sequence feature information (genes, repeats, tRNAs)

## 1.2 Learning outcomes

On completion of the tutorial, you can expect to be able to:

- Describe the different data formats used for sequence data (FASTA, FASTQ, SAM/BAM, CRAM, VCF/BCF, GFF)
- Perform conversions between the different data formats

## 1.3 Tutorial sections

This tutorial comprises the following sections:

1. [Data formats for NGS](#)
2. [Converting between formats](#)

## 1.4 Authors and License

This tutorial was written by [Jacqui Keane](#) and [Sara Sjunnebo](#) based on material from [Petr Danecek](#) and [Thomas Keane](#).

The content is licensed under a [Creative Commons Attribution 4.0 International License \(CC-BY 4.0\)](#).

## 1.5 Running the commands in this tutorial

You can follow this tutorial by typing all the commands you see in a terminal window on your computer. Remember, the terminal window is similar to the “Command Prompt” window on MS Windows systems, which allows the user to type DOS commands to manage files.

To get started, open a terminal window and type the command below followed by the Enter key:



```
cd ~/course_data/data_formats/data
```

## 1.6 Prerequisites

This tutorial assumes that you have the following software and their dependencies installed on your computer. The software used in this tutorial may be updated from time to time so, we have also given you the version which was used when writing this tutorial.

Package name	Link for download/installation instructions	Version
samtools	<a href="https://github.com/samtools/samtools">https://github.com/samtools/samtools</a>	1.17
bcftools	<a href="https://github.com/samtools/bcftools">https://github.com/samtools/bcftools</a>	1.17
picard-slim	<a href="https://broadinstitute.github.io/picard/">https://broadinstitute.github.io/picard/</a>	3.0.0

The easiest way to install the required software is using `conda`, a software package manager. These software have already been installed on the computer for you. To activate them type:



```
conda activate formats
```

After the software is activated type the following commands:



```
samtools --help
```



```
bcftools --help
```



```
picard -h
```

This should return the help message for these tools.

To get started with the tutorial, go to the first section: [Data formats for NGS](#)

## 2 Data formats for NGS

Here we will take a closer look at some of the most common data formats used in NGS analysis.

First check you are in the right directory



```
pwd
```

It should display something like

```
/home/manager/course_data/data_formats/data
```

### 2.1 FASTA

The FASTA format is one of the most common and simplest file formats for representing nucleotide sequence data. Each sequence in a FASTA file is composed of two parts, a header line and the actual sequence. The header always starts with the symbol “>” and is followed by information about the sequence, such as a sequence name/unique identifier.

Let’s look at an example:

```
>Sequence_1
CTTGACGACTTGAAAAATGACGAAATCACTAAAAACGTGAAAAATGAGAAATG
AAATCATGACGACTTGAAGTGAAAAAGTGAAAAATGAGAAATGAACGTGACGAC
AAAATGACGAAATCACTAAAAACGTGACGACTTGA AAAATGACCAC
```

We can see that for each sequence we get two lines of text:

- The first line begins with ‘>’ indicating that it is the ‘header’ line. This is immediately followed by ‘Sequence\_1’, which is the unique identifier for this sequence.
- The second line is the actual nucleotide sequence, split over several lines, beginning with ‘CTTGACGACTTGAA...’ and ending with ‘...TGACCAC’.

It is also possible to have multiple sequences in one multi-FASTA file like this:

```
>Sequence_1
AAATCATGACGACTTGAAGTGAAAAAGTGAAAAATGAGAAATGAACGTGACGAC
CGAATGACGAAATCACTAAAAACGTGACGACTTGA AAAATGACCAC
>Sequence_2
CTTGAGACGAAATCACTAAAAACGTGACGACTTGAAGTGAAAAATGAGAAATG
AAAATGACGAAATCATGACGACTTGAAGTGAAAAATAAATGACC
```

#### 2.1.1 Exercises

**Q1: How many sequences are there in the fasta file example.fasta? (hint: is there a grep command you can use?)**



If you get stuck here, do not spend too much time trying to solve this and move on. A solution will be provided during the session.

## 2.2 FASTQ

Often we need to accompany our sequence data with quality scores that estimate our confidence in the accuracy of the sequence data. The FASTQ format is an extension of the FASTA file format, and includes a quality score for each nucleotide in the sequence.

Let's look at an example:

```
@ERR007731.739 IL16_2979:6:1:9:1684/1
CTTGACGACTTGAAAAATGACGAAATCACTAAAAACGTGAAAAATGAGAAATG
+
BBCBCBBBBBBBABBBABBBBBBBABBBBBBBBBBBBBBABAABBBBBB=@>B
```

We can see that for each sequence we get four lines of text:

- The first line is a 'header' containing a unique identifier for the sequence and, optionally, further information.
- The second line contains the nucleotide sequence
- The third line starts with + and optionally contains the ID again. This line is redundant and can be safely ignored.
- The fourth line contains a string of characters that encode quality scores for each nucleotide in the sequence.

### 2.2.1 FASTQ and illumina data

Paired-end illumina sequencing involves reading from both ends of a sequence fragment as illustrated below.

#### Paired-End Reads



The above image has been taken from <https://www.illumina.com/science/technology/next-generation-sequencing/plan-experiments/read-length.html#>

During this process two FASTQ files are produced. One of the FASTQ files is usually named with `_1.fastq` at the end and contains the sequencing data from ‘reading’ one end of the fragment (reads). The other FASTQ file is usually named with `_2.fastq` at the end and contains the sequencing data from ‘reading’ the other end of the fragment (reads). Each fragment has a unique name and the sequence read for one end of the fragment is labelled with the fragment name followed by `/1`. The corresponding sequence read for the other end of the fragment is labelled with the fragment name followed by `/2`. The order in which the reads appear in the FASTQ files is also important as many tools assume that the first read in the `_1.fastq` file and first read in the `_2.fastq` file are from the same fragment and so on.

So FASTQ data for an illumina paired end sequencing run might look like:

In one fastq file (e.g. `run_1.fastq`):

```
@ERR007731.739 IL16_2979:6:1:9:1684/1
CTTGACGACTTGAAAAATGACGAAATCACTAAAAAACGTGAAAAATGAGAAATG
+
BBCBCBBBBBBBABBBABBBBBBBABBBBBBBBBBBBBBABAABBBB=Q>B
@ERR007731.740 IL16_2979:6:1:9:1419/1
TAGGGGAAAGTTCTCATGTACATCCGAAAAGAGGGCAACCCCAAATCAAAAG
+
BBABBBABABAABABABBABBBAAA>@B@BBAA@4AAA>.>BAA@779:AAA@A
```

The other fastq file (e.g. `run_2.fastq`):

```
@ERR007731.739 IL16_2979:6:1:9:1684/2
GATGACGACCCGAAAAATTACGAAATCACTAGCCAACGTGAATTTTGAGAACTA
+
BBABCBCCCCBBBABBBABBBBBBBABBBBBBBBBBBBBBAAAAAABBBBBBA@>B
@ERR007731.740 IL16_2979:6:1:9:1419/2
AAAAAAAAAGATGTCATCAGCACATCAGAAAAGAAGGCAACTTTAAACTTTTC
+
BBCBBBABABBABABABBABBBAAA>@BBBBAA@4AAA>.>BAA@779:AA>@A
```

### 2.2.2 Quality scores

In a FASTQ file, a single character encodes a quality score, typically a number between 0 and 40 (but in theory this can range between 0-93). Each character maps to an ASCII value which in turn can be converted to a quality score.

Character	ASCII	FASTQ quality score (ASCII – 33)
!	33	0
“	34	1
#	35	2
\$	36	3
%	37	4
...	...	...

Character	ASCII	FASTQ quality score (ASCII – 33)
C	67	34
D	68	35
E	69	36
F	70	37
G	71	38
H	72	39
I	73	40

The first 32 ASCII codes are reserved for characters which are not printable (e.g. tab, return, space etc.). None of these can be used in the quality string, so we subtract 33 from the ASCII value of the character to determine the quality score. For example, the ASCII code for “C” is 67, so the corresponding quality is:

$$Q = 67 - 33 = 34$$

So, in the FASTQ examples above, most of the base calls have scores in the 30s, which indicates a high degree of confidence in their accuracy. A score of 30 denotes a 1 in 1000 chance of an error; i.e. 99.9% accuracy.

Quality Score	Probability of incorrect base call	Base call accuracy
10	1 in 10	90%
20	1 in 100	99%
30	1 in 1000	99.9%
40	1 in 10,000	99.99%
50	1 in 100,000	99.999%
60	1 in 1,000,000	99.9999%

You don’t need to worry about being able to convert the characters to a quality score as most of the software tools can interpret them automatically. But the following perl command will print the quality score for an ASCII character. Try changing the “A” to another character, for example one from the quality strings above (e.g. @, = or B).



```
perl -e 'printf "%d\n",ord("A")-33;'
```

### 2.2.3 Exercises

**Q2:** How many reads are there in the file `example.fastq`? (Hint: remember that @ is a possible quality score. Is there something else in the header that is unique?)



Again, don’t worry if you cannot solve this, a solution will be provided during the practical session.

**Note:** The FASTQ format is a text based file, however it is possible (and good practice) to compress these files with gzip. A gzipped fastq file is usually suffixed with `.fastq.gz` or `.fq.gz`.

## 2.3 SAM/BAM

A common task with sequence data is to match or align it to a reference genome. [SAM \(Sequence Alignment/Map\)](#) is a standard format for storing sequence read alignments to a reference genome. If no reference genome is available, the data can be stored unaligned. SAM is a text based file. BAM is the compressed binary version of SAM. Compressed binary files are not readable by a human but are smaller than the corresponding uncompressed file meaning they take up less disk space and make it easier and quicker to copy files between locations.

SAM/BAM files consist of a header section (optional) and an alignment section. The alignment section contains one record (a fragment alignment) per line describing the alignment between fragment and reference. Each record has 11 fixed columns and optional key:type:value tuples. Open the [SAM/BAM file specification document](#) as you may need to refer to it throughout this tutorial.

Now let us have a closer look at the different parts of the SAM/BAM files.

### 2.3.1 Header Section

Each line or record in the SAM header starts with an @, followed by a two-letter code defining the record type (the different types are defined in the [SAM/BAM format specification document](#)). Each line or record contains meta-data for that specific record which is captured as a series of key-value pairs in the format of 'TAG:VALUE'.

**Read groups** One useful record type is RG which can be used to describe each unit of sequencing, e.g. a barcode or lane of sequencing data for Illumina. The RG code can be used to capture extra meta-data for the unit of sequencing. Some common RG TAGs are:

- ID: Read group identifier
- PL: Sequencing platform
- LB: Library name
- PI: Predicted insert/fragment size
- DS: Description
- SM: Sample identifier
- CN: Sequencing centre

### 2.3.2 Exercises

Look at the following line from the header of the SAM/BAM file and answer the questions that follow:

```
@RG ID:ERR003612 PL:ILLUMINA LB:g1k-sc-NA20538-T0S-1 PI:2000 DS:SRP000540 SM:NA20538 CN:SC
```

You may want to refer to section 1.3 of the SAM specification.

**Q3: What does RG stand for?**



**Q4: What platform was used to produce the data?**



**Q5: Where was the sequence data produced?**




**Q6: What is the expected insert/fragment size?**

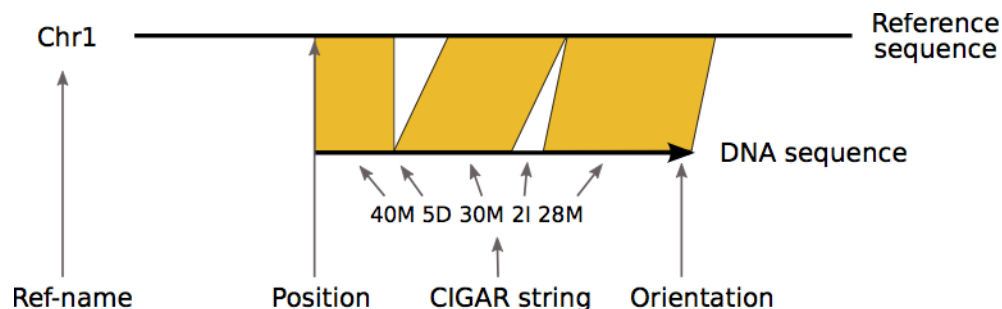



### 2.3.3 Alignment Section

The alignment section of a SAM file contains one line per alignment. Each line consists of 12 field-/columns described below. The first 11 columns are mandatory.

1. QNAME: Query NAME of the read or the read pair
2. FLAG: Bitwise FLAG (pairing, strand, mate strand, etc.)
3. RNAME: Reference sequence NAME
4. POS: 1-Based leftmost POSition of clipped alignment
5. MAPQ: MAPping Quality (Phred-scaled)
6. CIGAR: Extended CIGAR string (operations: MIDNSHPX=)
7. MRNM: Mate Reference NaMe ('=' if same as RNAME)
8. MPOS: 1-Based leftmost Mate POSition
9. ISIZE: Inferred Insert SIZE
10. SEQ: Query SEQUENCE on the same strand as the reference
11. QUAL: Query QUALity (ASCII-33=Phred base quality)
12. OTHER: Optional fields

The image below provides a visual guide to some of the fields/columns of the SAM format.



In a SAM file, the alignment in this image would be represented in a SAM/BAM file as:

```
fragment001    163  Chr1    19999970    23  40M5D30M2I28M    =    20000147    213
GGTGC GTGAT...    <= @A @?? @= A...
```

### 2.3.4 Exercises

Let's have a look at example.sam. Notice that we can use the standard Linux operations like **less** on this file.



```
less -S example.sam
```



**Q7: What is the mapping quality of ERR003762.5016205? (Hint: can you use grep and awk to find this?)**




**Q8: What is the CIGAR string for ERR003814.6979522? (We will go through the meaning of CIGAR strings in the next section)**




**Q9: What is the inferred insert/fragment size for ERR003814.1408899?**




### 2.3.5 CIGAR string

Column 6 of the alignment is the CIGAR string for that alignment. The CIGAR string provides a compact representation of sequence alignment. Have a look at the table below. It contains the meaning of all different symbols of a CIGAR string:

	Symbol	Meaning
M		alignment match or mismatch
=		sequence match
X		sequence mismatch
I		insertion into the reference
D		deletion from the reference
S		soft clipping (clipped sequences present in SEQ)
H		hard clipping (clipped sequences NOT present in SEQ)
N		skipped region from the reference
P		padding (silent deletion from padded reference)

Below are two examples describing the CIGAR string in more detail.

#### Example 1:

Ref: ACGTACGTACGTACGT

Read: ACGT- - - ACGTACGA

Cigar: 4M 4D 8M

The first four bases in the read are the same as in the reference, so we can represent these as 4M in the CIGAR string. Next is a deletion of 4 bases, represented by 4D, followed by 7 alignment matches and one alignment mismatch, represented by 8M. Note that the mismatch at position 16 is included in 8M. This is because it still aligns to the reference.

#### Example 2:

Ref: ACTCAGTG- - GT

Read: ACGCA- TGCAGTtagacgt

Cigar: 5M 1D 2M 2I 2M 7S

Here we start with 5 alignment matches and mismatches, followed by a deletion of one base. Then we have two more alignment matches, an insertion of 2 bases and two more matches. At the end, we have a soft clipping of 7 bases, 7S. These are clipped sequences that are present in the read but do not match the reference.

### 2.3.6 Exercises

**Q10: What does the CIGAR from Q8 mean?**



**Q11: How would you represent the following alignment with a CIGAR string?**

Ref: ACGT- - - ACGTACGT

Read: ACGTACGTACGTACGT



### 2.3.7 Flags

Column 2 of the alignment contains a combination of bitwise FLAGS providing detailed information about the alignment. The following table details the meaning of each flag

			Hex	Dec	Flag	Description
0x1	1	PAIRED				paired-end (or multiple-segment) sequencing technology
0x2	2	PROPER_PAIR				each segment properly aligned according to the aligner
0x4	4	UNMAP				segment unmapped
0x8	8	MUNMAP				next segment in the template unmapped
0x10	16	REVERSE				SEQ is reverse complemented
0x20	32	MREVERSE				SEQ of the next segment in the template is reversed
0x40	64	READ1				the first segment in the template
0x80	128	READ2				the last segment in the template
0x100	256	SECONDARY				secondary alignment
0x200	512	QCFAIL				not passing quality controls
0x400	1024	DUP				PCR or optical duplicate
0x800	2048	SUPPLEMENTARY				supplementary alignment

For example, if you have an alignment with FLAG set to 113, this can only be represented by decimal numbers  $64 + 32 + 16 + 1$ , so we know that these four flags apply to the alignment and the alignment is paired, reverse complemented, the sequence of the next template/read in the fragment is reversed and the read aligned is the first read in the template.

**Primary, secondary and supplementary alignments** A read that aligns to a single position in a reference (including insertions, deletions, skips and clipping but not direction changes), is a **linear alignment**. If a read cannot be represented as a linear alignment, but instead is represented as a group of linear alignments without large overlaps, it is called a **chimeric alignment**. These can for

instance be caused by structural variations. Usually, one of the linear alignments in a chimeric alignment is considered to be the **representative** alignment, and the others are called **supplementary**.

Sometimes a read maps equally well to more than one location. In these cases, one of the possible alignments is marked as the **primary** alignment and the rest are marked as **secondary** alignments.

### 2.3.8 BAM

BAM (Binary Alignment/Map) format, is a binary compressed version of SAM. This means that, while SAM is human readable, BAM is only readable for computers. BAM files can be viewed using `samtools`, and will then have the same format as a SAM file. The key features of BAM are:

- Stores alignments from most mapping tools
- Supports multiple sequencing technologies
- Supports indexing for quick retrieval/viewing of alignments
- Compact size (e.g. 112Gbp Illumina = 116GB disk space)
- Reads can be grouped into logical groups e.g. lanes, libraries, samples
- Widely supported by variant calling packages and genome viewers

### 2.3.9 Exercises

Since BAM is a binary format, we can't use the standard Linux operations (`cat`, `less`, `head`, `grep` etc.) directly on this format. `Samtools` is a set of programs for interacting with SAM and BAM files. Using the **`samtools view`** command, print the header of the BAM file:



```
samtools view -H NA20538.bam
```

**Q12: What version of the human assembly was used to perform the alignments? (Hint: Can you spot this somewhere in the @SQ records?)**



**Q13: How many sequencing runs/lanes are in this BAM file? (Hint: Do you recall what RG represents?)**



**Q14: What programs were used to create this BAM file? (Hint: have a look for the program record, @PG)**



**Q15: What version of bwa was used to align the reads? (Hint: is there anything in the @PG record that looks like it could be a version tag?)**



Running **`samtools view`** on a BAM file without any options will produce SAM format without the header information. This is printed to the STDOUT in the terminal (screen). Let's have a look at

the first read of the BAM file:

```
samtools view NA20538.bam | head -n 1
```

Note we only want to look at the first line of the alignment section of the BAM file so we have piped the output of `samtools view` to the `head` command.

**Q16: What is the name of the first read? (Hint: have a look at the [alignment section](#) if you can't recall the different fields)**

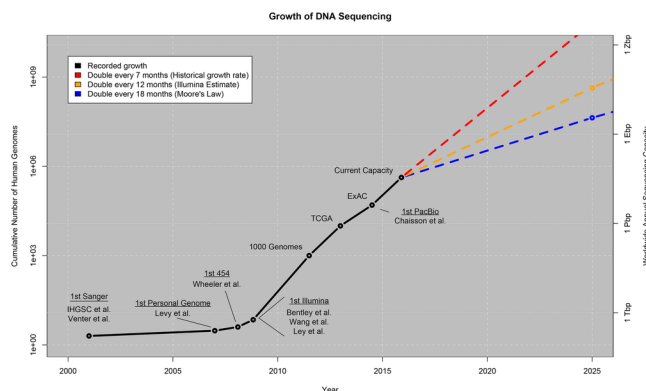


**Q17: What position does the alignment start at?**



## 2.4 CRAM

Even though BAM files are compressed, they are still very large. Typically they use 1.5-2 bytes for each base pair of sequencing data that they contain, and while disk capacity is ever improving, increases in disk capacity are being far outstripped by sequencing technologies.



BAM stores all the data for a sequence read, this includes every base call and every base quality, and it uses a single compression technique for all types of data (numbers, characters etc.). Therefore, CRAM was designed to provide a way to store the same information as BAM but using less disk space. CRAM uses three important concepts:

- Reference based compression
- Controlled loss of quality information
- Different compression methods to suit the type of data, e.g. base qualities vs. metadata vs. extra tags

The figure below displays how reference-based compression works. Instead of storing all the bases of all the reads, only the nucleotides that differ from the reference, and their positions, are kept.

```

Reference sequence: ACGTACGTACGTACGTACGTACGTACGTACGTAC
read 1:             ACGTACGTACGTACGTACGTGC
read 2:             TACGTACGCACGTACGTGCGTA
read 3:             CGTACGCACGTACGTACGTACG
read 4:             TACGTACGTACGTGCGTACGTA
read 5:             CGCACGTACGTACGTACGTACG
read 6:             TACGTGCGTACGTACGTAC

```

```

Reference sequence: ACGTACGTACGTACGTACGTACGTACGTACGTAC
read 1:             .....G.
read 2:             .....C.....
read 3:             .....C.....
read 4:             .....G.....
read 5:             .....C.....
read 6:             .....G.....

```

This means that the same information from a BAM file can be stored in CRAM file but using a fraction of the disk space.

## 2.5 Sorting and Indexing

The reads in a BAM and CRAM file can be ordered or sorted in one of two ways:

- sorted by name, meaning the reads are ordered based on the fragment name so reads from the same fragment (read pairs) will appear next to each other in the file
- coordinate-sorted, meaning the reads that align to the leftmost position or start of the genome appear first in the file.

To allow for fast random access of regions in BAM and CRAM files, they can be indexed. The files must first be coordinate-sorted. This can be done using **samtools sort**. If no options are supplied, it will by default sort by the left-most position.



```
samtools sort -o NA20538_sorted.bam NA20538.bam
```

Now we can use **samtools index** to create an index file (.bai) for our sorted BAM file:



```
samtools index NA20538_sorted.bam
```

You can think on an index file as a lookup table that tools like samtools can use to easily and quickly retrieve a segment of the file without having to read the entire file into memory. For example, to look for reads mapped to a specific region, we can use **samtools view** and specify the region we are interested in as: `RNAME[:STARTPOS[-ENDPOS]]`.

If we wanted to look at all the reads mapped to chromosome 1, we could use:

```
samtools view NA20538_sorted.bam 1 | head -10
```

To look at the region on chromosome 1 beginning at position 25,000,000 and ending at the end of the chromosome, we can do:

```
samtools view NA20538_sorted.bam 1:25000000
```

And to explore the 1001bp long region on chromosome 1 beginning at position 20,000,000 and ending at position 20,001,000, we can use:

```
samtools view NA20538_sorted.bam 1:20000000-20001000 | tail -10
```

### 2.5.1 Exercises

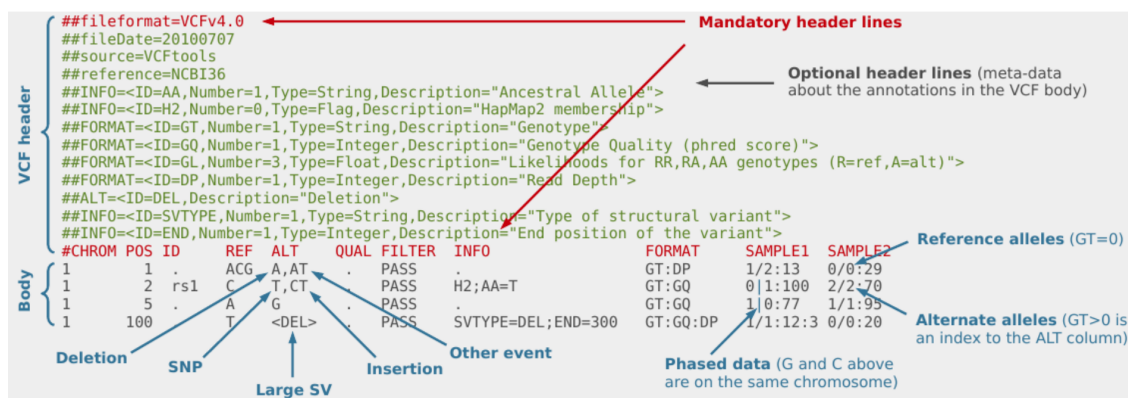
Q18: How many reads are mapped to region 20025000-20030000 on chromosome 1?



## 2.6 VCF/BCF

The VCF format is a standard format for storing sequence variation data. The BCF format is the compressed binary version of VCF. Remember that a compressed binary file is not human readable.

VCF is a text based tab-delimited file that is parsable by standard Linux commands. It is composed of two parts, the VCF header and the body. The figure below provides an overview of the different components of a VCF file:



### 2.6.1 VCF header

Header lines are denoted with ## and provide metadata about the file (e.g. fileformat, fileDate and reference) and metadata defining the fields used in the body of the file (e.g. INFO, FILTER, and FORMAT). These header lines consist of key=value pairs and can consist of multiple pairs enclosed by <>. More information about these fields is available in the [VCF specification](#).

All header lines are optional and can be put in any order, except for *fileformat*. This holds the information about which version of VCF is used and must come first in the file.

### 2.6.2 VCF body

The body of the VCF follows the header, and is tab separated into 8 mandatory columns and an unlimited number of optional columns that may be used to record other information about the sample(s).

**Header line** The header line starts with # and contains the names of the columns used in the file:

1. CHROM: an identifier from the reference genome
2. POS: the reference position
3. ID: a list of unique identifiers (where available)
4. REF: the reference base(s)
5. ALT: the alternate base(s)
6. QUAL: a phred-scaled quality score
7. FILTER: filter status
8. INFO: additional information

If the file contains genotype data, additional fields are included in the file. These are a FORMAT column, and then a number of sample IDs. The FORMAT field defines the data types and order of the information for each sample. Some examples of these data types are:

- GT: Genotype, encoded as allele values separated by either / or |
- DP: Read depth at this position for this sample
- GQ: Conditional genotype quality, encoded as a phred quality

**Positions** Following the header line is a series of rows containing information about a position in the genome along with genotype information at that position for each of the samples.

Let's look at a specific example:

```
#CHROM POS ID REF ALT QUAL FILTER INFO FORMAT SAMPLE1 SAMPLE2
1 3 . A G . PASS AC=67;AN=5400;DP=2809 GT:PL:DP:GQ 1/1:0,9,73:26:22 0/0:0,9,73:13:31 0/0:0,9,73:48:99 1/0:255,0,75:32:15 1/0:255,0,75:32:15
```

The locations is position 3 on chromosome 1, the variant site does not have an identifier in a standard database. The reference base is an A at this position and all the alternative alleles called in all samples is G. There is no quality score for the site and it passes the quality filters that have been applied. The values for AC, AN and DP for this site across all samples is AC=67, AN=5400, DP=2809. The definition of AC, AN and DP can be found in the INFO lines in the header of the VCF and AC is allele count, AN is allele number and DP is read depth. The remainder of the columns provide information about the site in each sample. First the FORMAT column describes the information listed for each sample, GT, PL, DP, GQ. Again the definition of these will be found in the VCF header and are defined as genotype, genotype likelihoods, read depth and genotype quality. This means for SAMPLE1 the genotype is 1/1 (G/G), the genotype likelihoods are 0,9,73, the depth at this position in this sample is 26 and the genotype quality is 22. Similarly for SAMPLE2 the genotype is 0/0 (A/A), the genotype likelihoods are 0,9,73, the depth at this position in this sample is 13 and the genotype quality is 22.

### 2.6.3 BCF

VCF files can be compressed (for example with gzip), but even compressed they can still be very large. For example, a compressed VCF with 3781 samples of human data will be 54 GB for chromo-

some 1, and 680 GB for the whole genome.

VCFs can also be slow to parse, as text conversion is slow. The main bottleneck is the “FORMAT” fields. For this reason the BCF format, a binary representation of VCF, was developed. In BCF files the fields are rearranged for better compression and fast access. The following images show the process of converting a VCF file into a BCF file.

```
##fileformat=VCFv4.0
##fileDate=20180707
##source=VCFtools
##ALT=<ID=DEL,Description="Deletion">
##INFO=<ID=END,Number=1,Type=Integer,Description="End position of the variant">
#CHROM POS ID REF ALT QUAL FILTER INFO FORMAT SAMPLE1 SAMPLE2
1 3 . A G . PASS AC=67;AN=5400;DP=2809 GT:PL:DP:GQ 1/1:0,9,73:26:22 0/0:0,9,73:13:31 0/0:0,9,73:48:99 1/0:255,0,75:32:15 1/0:255,0,75:32:15
1 4 . A T . PASS AC=15;AN=6800;DP=6056 GT:PL:DP:GQ 0/0:0,9,73:13:31 1/0:255,0,75:32:15 0/0:0,2,80:14:90 1/1:0,9,73:26:22 0/0:0,9,73:13:31
1 5 . C T . PASS AC=20;AN=6701;DP=5234 GT:PL:DP:GQ 1/0:255,0,75:32:15 0/0:0,2,170:14:90 1/1:0,9,73:13:31 0/0:0,6,50:13:80 0/0:0,2,80:14:90
1 6 . A G . PASS AC=67;AN=5400;DP=2809 GT:PL:DP:GQ 1/1:0,9,73:26:22 0/0:0,9,73:13:31 0/0:0,9,73:48:99 1/0:255,0,75:32:15 1/0:255,0,75:32:15
1 7 . A T . PASS AC=15;AN=6800;DP=6056 GT:PL:DP:GQ 0/0:0,9,73:13:31 1/0:255,0,75:32:15 0/0:0,2,80:14:90 1/1:0,9,73:26:22 0/0:0,9,73:13:31
```

The diagram illustrates the conversion of a VCF record to a BCF record. The VCF record (top) has fields: #CHROM, POS, ID, REF, ALT, QUAL, FILTER, INFO, FORMAT, and sample names. The BCF record (bottom) has fields: #CHROM, POS, ID, REF, ALT, QUAL, FILTER, INFO, and sample names. Arrows indicate the mapping of fields between the two formats. The VCF FORMAT field is split into GT, PL, DP, and GQ, which are then mapped to the corresponding fields in the BCF record. The sample names are also mapped to the corresponding fields in the BCF record.

Bcftools comprises a set of programs for interacting with VCF and BCF files. It can be used to view or extract records from a region and to convert between VCF and BCF formats.

**bcftools view** Let’s have a look at the header of the file 1kg.bcf in the data directory. Note that bcftools uses `-h` to print only the header, while samtools uses `-H` for this.

```
bcftools view -h 1kg.bcf
```

Similarly to BAM, BCF supports random access, that is, fast retrieval from a given region. For this, the file must be indexed:

```
bcftools index 1kg.bcf
```

Now we can extract all records from the region 20:24042765-24043073, using the `-r` option. The `-H` option will make sure we don’t include the header in the output:

```
bcftools view -H -r 20:24042765-24043073 1kg.bcf
```

**bcftools query** The versatile **bcftools query** command can be used to extract any VCF field. Combined with standard Linux commands, this gives a powerful tool for quick querying of VCFs. Have a look at the usage options:

```
bcftools query -h
```

Let’s try out some useful options. As you can see from the usage, `-l` will print a list of all the samples in the file. Give this a go:

```
bcftools query -l 1kg.bcf
```



Another useful option is `-s` which allows you to extract all the data relating to a particular sample. Try this for sample HG00131:

```
bcftools view -s HG00131 1kg.bcf | head -n 50
```

The format option, `-f` can be used to select what gets printed from your query command. For example, the following will print the position, reference base and alternate base for sample HG00131, separated by tabs:

```
bcftools query -f '%POS\t%REF\t%ALT\n' -s HG00131 1kg.bcf | head
```

### 2.6.4 Exercises

Now, try and answer the following questions about the file 1kg.bcf in the data directory. For more information about the different usage options you can open the [bcftools query manual page - http://samtools.github.io/bcftools/bcftools.html#query](http://samtools.github.io/bcftools/bcftools.html#query) in a new tab.

**Q19: What version of the human assembly do the coordinates refer to?**



**Q20: How many samples are there in the BCF?**

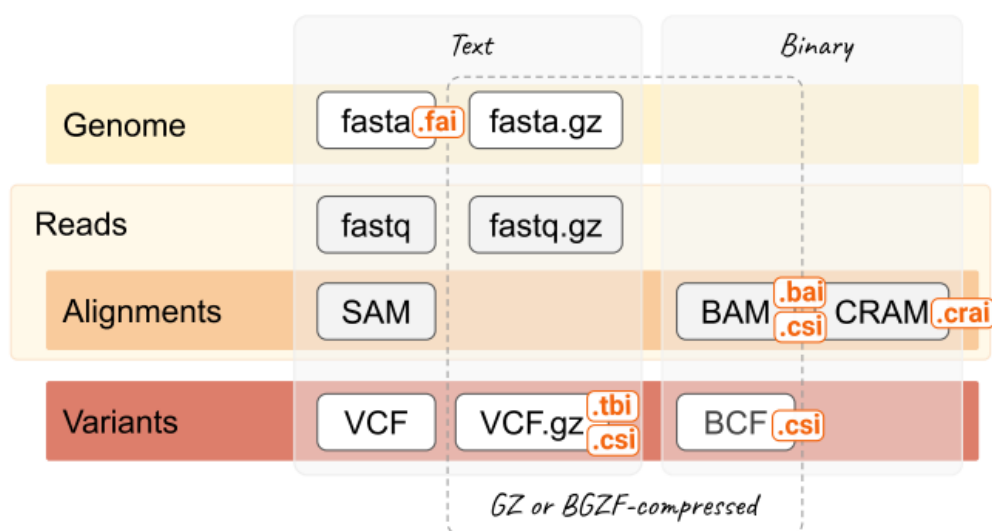


**Q21: What is the genotype of the sample HG00107 at the position 20:24019472? (Hint: use the combination of `-r`, `-s`, and `-f` options)**



## 2.7 Summary

The figure below summarises the data formats we have looked at so far.



## 2.8 GFF

One final format worth mentioning is the GFF format. The general feature format (gene-finding format, generic feature format, GFF) is a file format used for describing genes and other features of DNA, RNA and protein sequences. The format consists of one line per feature, each containing 9 columns of data, separated by a tab.

Let's look at an example:

```
1  Prodigal    CDS 210 1422    .   -   0   ID=s01
1  Prodigal    CDS 508 2464    .   -   0   ID=s02;product=hypothetical protein
1  Prodigal    CDS 967 3525    .   -   0   ID=s03;Name=rfuC;db_xref=COG:COG403
```

We can see that for each line we have nine fields all separated by the tab character:

1. **seqname** - The name of the sequence where the feature is located.
2. **source** - The algorithm or procedure that generated the feature. This is typically the name of a software or database.
3. **feature** - The feature type name, like “gene”, “exon” or “cds”.
4. **start** - The start position of the feature, with sequence numbering starting at 1.
5. **end** - The end position of the feature, with sequence numbering starting at 1.
6. **score** - A numeric value that generally indicates the confidence of the source in the annotated feature. A value of “.” (a dot) is used to define a null value.
7. **strand** - Single character that indicates the strand of the feature. This can be “+” (positive, or 5'→3'), “-” (negative, or 3'→5'), “.” (undetermined), or “?” for features with relevant but unknown strands.
8. **frame** - One of ‘0’, ‘1’ or ‘2’. ‘0’ indicates that the first base of the feature is the first base of a codon, ‘1’ that the second base is the first base of a codon, and so on..
9. **attribute** - A semicolon-separated list of tag-value pairs, providing additional information about each feature.

Congratulations you have reached the end of the data formats tutorial!

If you have time then continue to the additional (optional) section of the tutorial: [Converting between formats](#).


## 3 Converting between formats

In this section we are going to look at how to convert from one data format to another. There are many tools available for converting between formats, and we will use some of the most common ones: `samtools`, `bcftools` and `Picard`. Although it might not be obvious yet how this is useful, the tasks are a good way to become familiar with and understand the different options for the `samtools` and `bcftools`.


### 3.1 SAM/BAM

To convert between SAM and BAM formats you can use the `samtools view` command.


To convert a BAM file to a SAM file try:

```
 samtools view -h NA20538.bam > NA20538.sam
```

Note we use the `-h` option here to include the header in the SAM file. Now, have a look at the first five lines of the SAM file.

```
 head -5 NA20538.sam
```

Well that was easy! And converting SAM to BAM is just as straightforward, try:

```
 samtools view -b NA20538.sam > NA20538_2.bam
```

This time there is no need for the `-h` option, however we have to tell `samtools` that we want the output in BAM format. We do this by adding the `-b` option. Note we have called the file `NA20538_2.bam` so that the original `NA20538.bam` file is not overwritten.

Check the conversion was successful.

First check the header section:

```
 samtools view -h NA20538_2.bam | head -5
```


Then check the alignment section:

```
 samtools view NA20538_2.bam | head -5
```

### 3.2 BAM/CRAM

The `samtools view` command can also be used to convert between BAM and CRAM formats. In the data directory there is a BAM file called `yeast.bam` that was created by aligning *S. cerevisiae* Illumina data to the reference genome `Saccharomyces_cerevisiae.EF4.68.dna.toplevel.fa`.

To convert a BAM to a CRAM, try

```
 samtools view -C -T Saccharomyces_cerevisiae.EF4.68.dna.toplevel.fa \  
-o yeast.cram yeast.bam
```

Here, we use the `-C` option to tell `samtools` we want the output as CRAM, and the `-T` option to specify what reference file to use for the conversion. We also use the `-o` option to specify the name of the output file.

Have a look at what files were created:

```
ls -l
```

As you can see, this has created an index file for the reference genome called `Saccharomyces_cerevisiae.EF4.68.dna.toplevel.fa.fai` and the CRAM file `yeast.cram`.

Check the conversion was successful:

```
samtools view yeast.cram | head -5
```

### 3.2.1 Exercises

**Q1:** Since CRAM files use reference-based compression, we expect the CRAM file to be smaller than the BAM file. What is the size of the CRAM file?



**Q2:** Why do we need to provide the reference genome when converting to CRAM format?



**Q3:** Convert the CRAM file back to a BAM file called `yeast_2.bam`?



## 3.3 FASTQ to SAM/BAM/CRAM

SAM format is mainly used to store alignment data, however in some cases we may want to store the unaligned data in SAM format and for this we can use the `picard` tools `FastqToSam` application. [Picard tools](#) is a Java application with a number of useful options for manipulating sequence data.

To convert the FASTQ files from sequencing run 13681\_1\_18 to unaligned SAM format, try:

```
picard FastqToSam F1=13681_1_18_1.fastq.gz F2=13681_1_18_2.fastq.gz \
O=13681_1_18.sam SM=13681_1_18
```

Here, the `F1` and `F2` options are the paired end FASTQ files, the `O` option is used to specify the name of the output SAM file and `SM` is the sample name which will be stored in the header of the SAM file. There are also multiple other options for specifying what metadata to include in the SAM header. To see all available options, run:

```
picard FastqToSam -h
```


Check the conversion was successful:

```
 head -5 13681_1_18.sam
```

From here you can convert the SAM file to BAM and CRAM, as described previously.

### 3.4 CRAM to FASTQ

The `samtools fastq` command can be used to convert CRAM to FASTQ directly. However, for many applications we need the FASTQ files ordered so that the order of the reads in the first file match the order of the reads in the second file. This is to allow the reads in a fragment to be matched up based on their location in the FASTQ file. For this reason, we first use `samtools collate` to produce a collated BAM file which ensures that reads of the same name (and therefore from the same fragment) are grouped together in contiguous groups.

```
 samtools collate yeast.cram yeast.collated
```

Have a look at what files were created:

```
 ls
```

The newly produced BAM file will be called `yeast.collated.bam`. Have a look at the contents and notice how the order of the reads differs from the `yeast.cram` file:

```
 samtools view yeast.collated.bam | head -6
```

Let's use this to create two FASTQ files, one for the first reads of the fragment and one for the second reads of the fragment:

```
 samtools fastq -N -1 yeast.collated_1.fastq -2 yeast.collated_2.fastq \  
yeast.collated.bam
```

Here, we use the `-N` option to tell `samtools` to include the `/1` and `/2` in the read name.

Check the conversion was succesful:

```
 head -4 yeast.collated_1.fastq  
head -4 yeast.collated_2.fastq
```

### 3.5 VCF/BCF


In a similar way that `samtools view` can be used to convert between SAM, BAM and CRAM, `bcftools view` can be used to convert between VCF and BCF.

To convert a BCF file to a VCF file try:

```
 bcftools view -O v -o 1kg.vcf 1kg.bcf
```


The `-O` option allows us to specify in what format we want the output, compressed BCF (b), uncompressed BCF (u), compressed VCF (z) or uncompressed VCF (v). With the `-o` option we can select the name of the output file.

Have a look at what files were generated (the options `-lrt` will list the files in reverse chronological order):




```
ls -lrt
```

Check the conversion was successful:



```
bcftools view 1kg.vcf | head -10
```

To convert a VCF file to BCF, try:



```
bcftools view -O b -o 1kg_2.bcf 1kg.vcf
```

Note we have called the file `1kg_2.bcf` so that the original `1kg.bcf` file is not overwritten.

Check the conversion was successful:



```
bcftools view 1kg_2.bcf | head -10
```

### 3.5.1 Exercises

**Q4: Convert the BCF file `1kg.bcf` to a compressed VCF file called `1kg.vcf.gz`**



**Congratulations** you have reached the end of the tutorial!