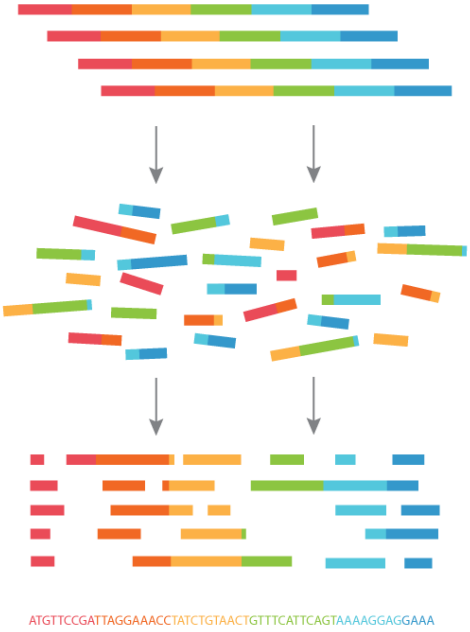
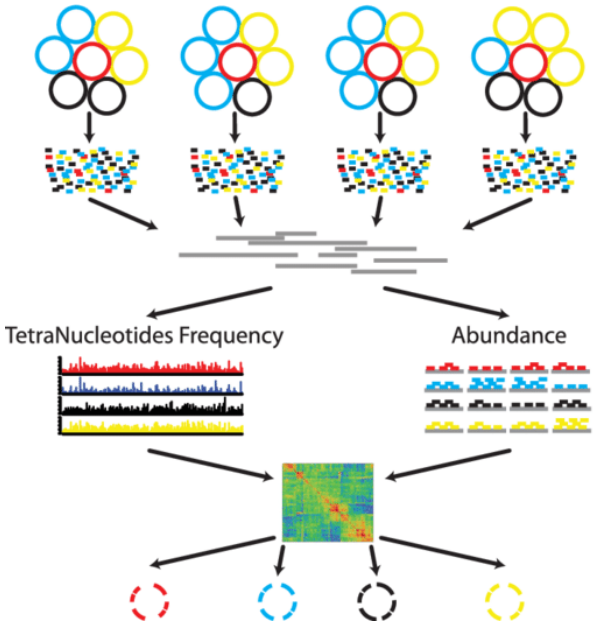
Metagenomics – A basic overview by Bram van Dijk

April 26, 2023

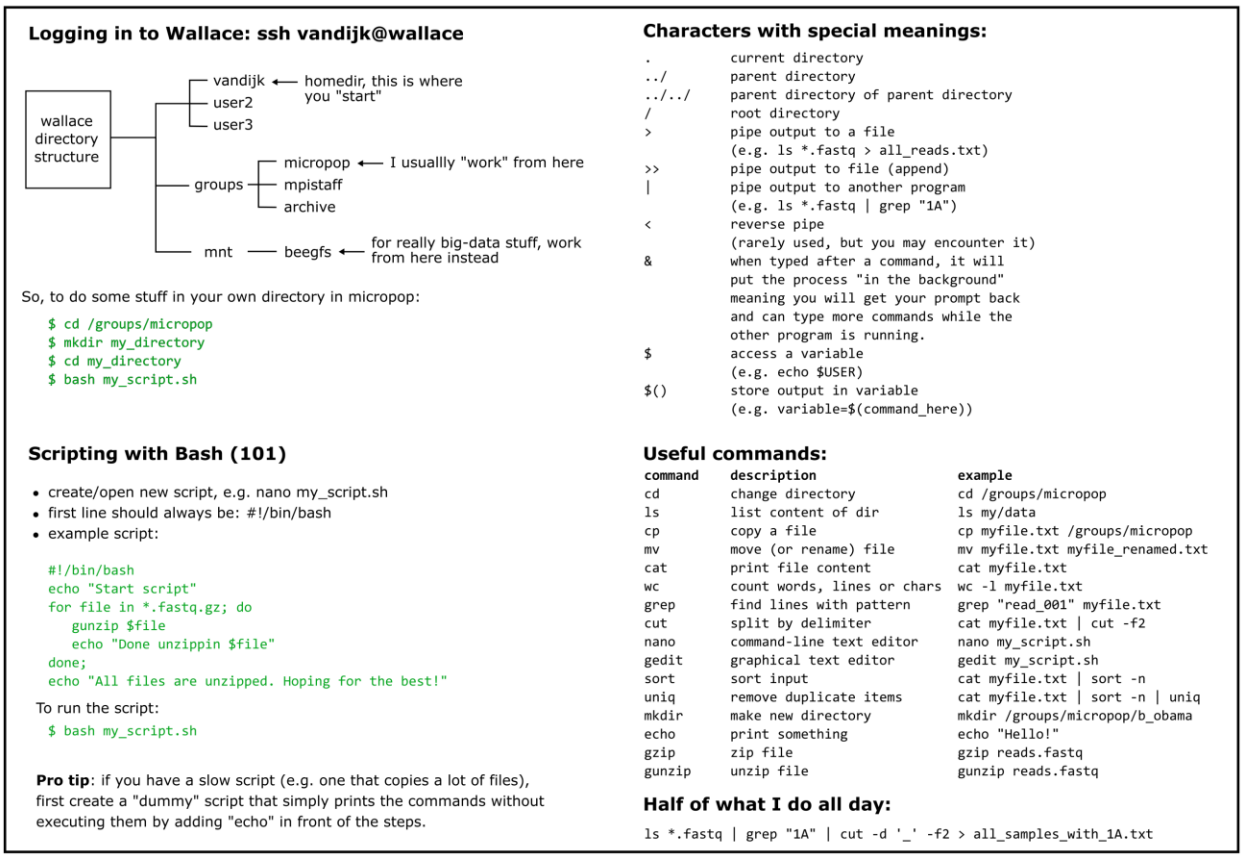




This overview of metagenomics is the result of a workshop given at the Max Planck Institute for Evolutionary Biology (April 24-25, 2023). If you're reading this in the future, or reading this as an outside observer, than please note that many instructions are specific for the workshop (such as running the setup scripts). That said, the basic steps that are explained should apply generally for anyone that wants to improve their understanding of metagenomics.

Preface for the workshop

For this workshop, I will not be covering all the basics of how to use Linux, the command-line, etc. Carsten offers great courses on this, and that's simply not my goal. However, if you want a quick refresher of the types of things you do as a bioinformatician, take a look at this overview:



Logging in to Wallace: ssh vandijk@wallace

wallace directory structure
- vandijk ← homedir, this is where you "start"
- user2
- user3
- groups
  - micropop ← I usuallly "work" from here
  - mpistaff
  - archive
- mnt
  - beegfs ← for really big-data stuff, work from here instead

So, to do some stuff in your own directory in micropop:

```
$ cd /groups/micropop
$ mkdir my_directory
$ cd my_directory
$ bash my_script.sh
```

Scripting with Bash (101)

- create/open new script, e.g. nano my_script.sh
- first line should always be: #!/bin/bash
- example script:

```
#!/bin/bash
echo "Start script"
for file in *.fastq.gz; do
   gunzip $file
   echo "Done unzippin $file"
done;
echo "All files are unzipped. Hoping for the best!"
```

To run the script:

```
$ bash my_script.sh
```

Pro tip: if you have a slow script (e.g. one that copies a lot of files), first create a "dummy" script that simply prints the commands without executing them by adding "echo" in front of the steps.

Characters with special meanings:

.	current directory
../	parent directory
../../	parent directory of parent directory
/	root directory
>	pipe output to a file (e.g. ls *.fastq > all_reads.txt)
>>	pipe output to file (append)
|	pipe output to another program (e.g. ls *.fastq | grep "1A")
<	reverse pipe (rarely used, but you may encounter it)
&	when typed after a command, it will put the process "in the background" meaning you will get your prompt back and can type more commands while the other program is running.
$	access a variable (e.g. echo $USER)
$()	store output in variable (e.g. variable=$(command_here))

Useful commands:

command	description	example
cd	change directory	cd /groups/micropop
ls	list content of dir	ls my/data
cp	copy a file	cp myfile.txt /groups/micropop
mv	move (or rename) file	mv myfile.txt myfile_renamed.txt
cat	print file content	cat myfile.txt
wc	count words, lines or chars	wc -l myfile.txt
grep	find lines with pattern	grep "read_001" myfile.txt
cut	split by delimiter	cat myfile.txt | cut -f2
nano	command-line text editor	nano my_script.sh
gedit	graphical text editor	gedit my_script.sh
sort	sort input	cat myfile.txt | sort -n
uniq	remove duplicate items	cat myfile.txt | sort -n | uniq
mkdir	make new directory	mkdir /groups/micropop/b_obama
echo	print something	echo "Hello!"
gzip	zip file	gzip reads.fastq
gunzip	unzip file	gunzip reads.fastq

Half of what I do all day:

```
ls *.fastq | grep "1A" | cut -d '_' -f2 > all_samples_with_1A.txt
```

Besides these commands, you may want to familiarise yourself with how to install/use tools on a shared computer cluster. The best way is to use conda (or mamba, a faster alternative that I haven't tested yet), which creates a virtual environment in which you can install your own tools, without needing any "sudo" (admin) rights. Kristian, Derk, Carsten, and other people around here will be able to help you with this in the future.

For illustrative purposes during the workshop, I have prepared a small dataset (with only 4 samples) which illustrate the types of things you can do with metagenomic/illumina data. Very often, you will have far more data, but this at least gives you a basic understanding on how to get started.

The scripts used in this course can be found here: */groups/mpistaff/MGX_Workshop/src*. Before you can use these files for the workshop, we have to copy of them, so we won't overwrite anything. There's also some other setup stuff that needs to happen. This takes some time, but only has to be done once:

Log into wallace (using your evolbio username)

```
$ ssh user@wallace
```

Note that the dollar sign ($) is not typed, but simply the start of the terminal prompt. During the workshop, let's not all use the same node on the cluster. So let's find a free node with the command "sinfo", and use that one:

```
$ sinfo | grep "idle"
$ ssh <an_idle _node_on_wallace>
```

Then, execute the workshop environment script.

```
$ bash /groups/mpistaff/MGX_Workshop/src/00_mount_env.sh
```

Execute the setup script. Takes ~5 minutes the first time (copying files), but is instant the second time around (it skips copying and just does some quality-of-life things for the course):

```
$ source /setup
```

Now you are ready for the workshop. There should be a directory on micropop now (*/groups/mpistaff/MGX_Workshop/MGX_Workshop_<YOURNAME>*), which you are also currently already in:

```
$ ls .
00_mount_env.sh
00_setup.rc
00_setup.sh
01_retrieve_data.sh
02_process_reads.sh
… etcetera
```

The scripts (ending in .sh) are all numbered more or less in the order we will use them. The scripts with 00 are already executed, so it is time to start counting at 1, and do some metagenomics!

Table of contents

0. Workshop introduction

As you all know, we have learned a lot about microorganisms by culturing them in the lab. But you all also know, many species cannot be cultured (or better said: we don't know how). There are many strategies to nevertheless improve our understanding of complex microbiomes, and metagenomics (MGX) is one of them. Where classical genomics tries to find the structure, evolution, function, and mappings of genes in a single genome, metagenomics was originally defined as doing exactly this, but for large collections of genomes. But how does one get access to information about large collections of genomes?

Instead of relying on culturability, MGX tries to learn about microbial communities by large-scale sequencing of DNA found in environmental samples or otherwise complex mesocosms. Somewhat ironically, the most popular form of "metagenomics" over the last decades has been **amplicon sequencing** (*e.g.* 16S metagenomics), where one sequences a single conserved locus as a marker for various bacterial species found in a sample. Not only is this strategy not really fitting of the term metagenomics (no genes or genomes are even analysed!), there is another urgent issue.

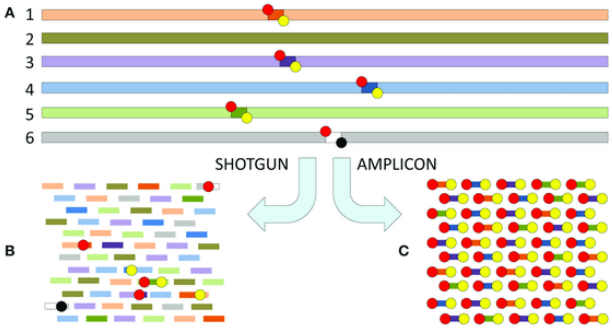


Figure 1 - Shotgun vs. amplicon sequencing (from Sekse et al. 2017)

The bacterial species concept is under fire. Gain and loss of genes is so frequent in the microbial world, that by the time a single nucleotide substitution has taken place, multiple genes have come and left the genome (see Puigbò *et al.*, 2014). Because of this massive gene flux, two closely related strains with the same 16S sequence can have remarkably different genotypes. Thus, while 16S can give us information about how many species are in a sample, it probably is an unreliable indicator of the functioning of this microbiome. So, in this workshop, we will work on "real" metagenomics, which I will refer to as MGX for short.

When I say MGX, I refer to **shotgun metagenomics**: you try to get all the DNA from a sample or microbial community, without any intended biases or selection procedures. The DNA from a sample is extracted, optionally fragmented into smaller pieces, and then sequenced. The tiny DNA sequences that are the result of this process, are what is referred to as a **read**. With Illumina sequencing, reads in a sample are all the same length, although the actual length can very from protocol to protocol (usually somewhere between 100 and 300 base pairs long). If you use Oxford Nanopore, PacBio, or another long-read sequencing technology, your read lengths vary from small (500 base pairs) to very large (100k base pairs). Whatever technology you use, each one of them comes with advantages and disadvantages. For example, while Illumina reads are very short, they also have a much lower error rate than nanopore sequences.

For this workshop, we will use the computer cluster. This cluster runs on Linux, but can be accessed with other operating systems too. But you will have to use the command line, since trying to open large sequencing files with something notepad is simply a terrible idea. So, without any further ado, let's go take a look at some data using the command-line!

1. Hands on: merging and renaming your data

When you get your data (*e.g.* when a company sends you a download link), make sure our IT department first copies these to the archive. We, as users, cannot change the files on the archive, so we protect ourselves from making mistakes. However, these raw reads are usually stored on the archive with confusing names like "*A02_S22_L001_R1.fastq.gz*", which really doesn't contain a lot of information about your experiment. Moreover, your reads may be distributed over multiple files, which happens when the sequencing was done on multiple lanes (L001, L002, L003, *etc.*). Finally, the raw reads are usually zipped (gz extension).

So, the very first thing we want to do is find all the files belonging to one sample, unzip these, and merge them in a two separate fastq files: one for forward reads (R1) and one for reverse reads (R2). Of course, if you have single-end reads, you will want to have only a single fastq file per sample. Simultaneously, we want to rename the files to something more meaningful. For example, a smaple from timepoint 1, community 1, with a certain concentration of ampicillin, could be name "T1_C1_7amp.fastq", and not "*A02_S22_L001_R1.fastq.gz".*

To achieve the above, it is very useful to first make what I call a *rosetta stone*[1], a simple plain text which translates has the undesired names (A02_S22…) in the first column, and the preferred names that match your experimental labels in the second column (*e.g.* T01_COM1_H_R1).

You need to make this file with a plain text editor (notepad, *pico, nano* or *vim*), just make sure it is encoded for UNIX, not Windows[2]. Typically, this text file can be easily generated by copy/pasting from the sample sheet you provided to the sequencing facility. In the example directory, let's look at the rosetta stone file by using cat:

```
$ cat rosetta_stone.txt
5316_AD         T1_C1_7amp_H
5316_BN         T2_C1_7amp_H
5316_CX         T3_C1_7amp_H
5316_EB         Tminus1_C1_powersoil
```

Note that the samples containing the string from the first column are stored in "raw_reads_course", but would usually be found on the archive somewhere.

To convert the files, we make a simple **bash script** (see videos on Youtube if you don't know what that is) that reads through this text file line by line, finds the file that matches the file in sample directory / read archive (*i.e.* using the undesired filename from the first column), and then unzips the reads with the desired filename (the second column) into my own directory called "*raw_reads_renamed*".

In "*pseudocode*", the script you need to make will do the following:

---

[1] Rosetta stone refers to a black basalt stone found in 1799 which was essential for translating hieroglyphics. This term "rosetta stone" isn't standard MGX jargon, but just something I made up. In any case, this file is **very important for reproducibility**, because raw data will always be stored with the original labels. So, for reproducibility, make sure this table is stored along with your analysis, such that people in the future can check your work!

[2] Windows notepad encodes files as LF, not UTF8, which can give problems when using the computer cluster. Trying to go through your file line-by-line may not work properly. You can download notepad++, which enables you to save your files as something UNIX compatible (UTF8). See also this link.

1. Make directory called raw_reads
2. Then, go through the rosetta_stone.txt file line by line, doing:

    a. Find file(s) containing "5316_AD" in Wallace's archive (or here, in the example directory)
    b. Unzip the file(s) to a new directory under the name "T1_C1_7amp_H.fastq"
    c. Go to the next line

An example of the code describe above is found in the script "01_retrieve_data.sh", and can be run by typing "*bash 01_retrieve_data.sh*". If your sequences are found in multiple directories on the archive, you may need to modify / duplicate the script to get all your data in order, but the basic recipe in this script should still work. The script should take only a few minutes for the example data, but for bigger data, you may have to get a coffee. Or 10.

2. Hands on: processing raw reads

Now that the raw reads are uncompressed, you should have a bunch of fastq files. Fastq files are like fasta files, but contain four lines per sequence: a header that starts with @, the sequence itself, a single '+' symbol, and finally the Phred-score string. The Phred-scores give the quality (or rather: confidence) of each base call, ranging from 0 to 60, and are shown as different symbols in the Phred-score string (see tables below). For Illumina reads, a quality above 30 (99.9% confidence) is expected, but for nanopore or other long-read sequencing technologies, much higher error rates are observed.

Table 1 ASCII Characters Encoding Q-scores 0–40

Symbol	ASCII Code	Q-Score	Symbol	ASCII Code	Q-Score	Symbol	ASCII Code	Q-Score
!	33	0	/	47	14	=	61	28
"	34	1	0	48	15	>	62	29
#	35	2	1	49	16	?	63	30
$	36	3	2	50	17	@	64	31
%	37	4	3	51	18	A	65	32
&	38	5	4	52	19	B	66	33
'	39	6	5	53	20	C	67	34
(	40	7	6	54	21	D	68	35
)	41	8	7	55	22	E	69	36
*	42	9	8	56	23	F	70	37
+	43	10	9	57	24	G	71	38
,	44	11	:	58	25	H	72	39
-	45	12	;	59	26	I	73	40
.	46	13	<	60	27			

Phred Quality Score	Error	Accuracy (1 - Error)
10	1/10 = 10%	90%
20	1/100 = 1%	99%
30	1/1000 = 0.1%	99.9%
40	1/10000 = 0.01%	99.99%
50	1/100000 = 0.001%	99.999%
60	1/1000000 = 0.0001%	99.9999%

Figure 2 - Phred scores are depicted as symbols (!#"@ABC) in a fastq file, which translates to a Q-score (0-60), which translates to error rates. For illumina, a cut-off of 30 is often used.

Using the information in the fastq file, tools have been developed that throw out reads that are really bad. The reads may also still have some (left over) adapter sequences that we may want to remove, and we also want to remove exact duplicate reads in the data, which are likely PCR artefacts. For these steps, we need to install some tools that are not installed on Wallace (the computer cluster). For this, we will use the tool fastp, which can do all of the above, and generate a nice HTML report which we can read in the browser.

Once again, a bash script can be made to do all this for each sample. An example of such a script is "02_process_reads.sh". This pseudocode for this script is:

1. Make directory called 'reads'
2. Then, for every forward read (R1) in the 'raw_reads_renamed', do:
    a. Extract the 'base_name' of the read file (same for R1 and R2)
    b. Pass base_name_R1.fastq and base_name_R2.fastq to fastp, store trimmed reads in the output directory 'reads', with the added suffix "_trimmed.fastq"

Depending on your computer, this script may take a bit of time (an hour or so). Don't worry though, the output is already in the course directory, so you can skip it if you want :)

3. Did we sequence "deep" enough?

Sequencing depth of a single isolate

Before analysing anything, the first thing one may want to check is if your sample was sequenced "deep enough". When you have a single species (e.g. you sequenced a single isolate, which strictly speaking is not metagenomics at all), you can calculate this yourself if you know the approximate genome size. For example, if you have 1 million random reads, you cover **at best** 1*150 million base pairs (assuming every read is 100% random, so this is the bare minimum number of reads you need). Thus, if your genome is 6 million base pairs (i.e. close to SBW25), your coverage is expected to be around 150/6 = 25. That means that every position in the SBW25 genome is covered with, **on average**, 25 reads. Depending on what you want to do, this may be enough. If you want to find high-frequency SNPs (present in 20-100% of the population), you can definitely pick those up with 25 reads per position. However, very rare SNPs (e.g. 1% of population), will be indistinguishable from sequencing errors. Generally, people recommend 100x coverage for reliable assembly and SNP statistics.

For "real" metagenomics, however, we're often interested in **de novo assembly** rather than SNPs. With a read coverage of 25, you will likely be able to assemble most of SBW25. However, because of repetitive regions in the genome, assembling a full 6 million base pair sequence is impossible with only short reads (this is where long-read technology can help, e.g. Oxford Nanopore, which I will not discuss further).

With multiple species, things get a bit more complicated

If your sample contains many species (*e.g.* our compost mesocosms), it will be hard to do a good back-of-the-envelope calculation of exactly how many reads you need. Moreover, you may not even *know* what is in the sample yet! To estimate the abundances of unknowns, we can use a technique called *rarefaction*. In ecology, rarefaction is a technique to assess species richness from the results of sampling by counting the number of species at various plots, until analysing more plots no longer results in more species. The resulting graph is called a rarefaction curve, from which you can infer whether more observations would still increase diversity (red line), or that adding more observations is pointless (blue line).

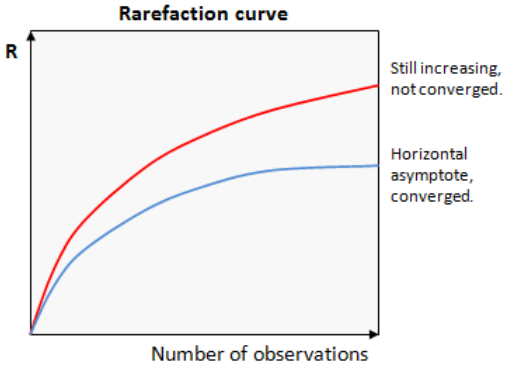


Figure 3 - Graph of rarefaction curves

By analogy of the above , we can create a rarefaction curve from MGX sequencing samples. The problem is, we don't have "species" to count. While we could annotate all our reads with certain species, this process is (i) very time-consuming, (ii) biased towards species existing in databases, and (iii) relies on the accuracy of said databases, which… isn't great.

A better approach is to estimate the diversity of the sequences themselves, rather than what species we think they belong to. For this, I recommend using the tool nonpareil. This tool estimates the sequencing efforts by checking Kmer frequencies[3]. The idea is simple. First, nonpareil checks the Kmer frequencies in 1 read, then in 2 reads, then in 4, 8, etc, until it has seen all reads, and it will keep track of which Kmers it has already seen. If the curve saturates (is flat at the end), you know you have sequenced very deeply, because adding even more reads wouldn't increase the observed Kmers anymore. Nonpareil comes with

[3] https://en.wikipedia.org/wiki/K-mer

an accompanying R package ("Nonpareil") that plots the coverage estimate (although you can also simply plot the NPO files yourself, but it it hard to compare across samples). In the example figure below (data from Pauline Buffard's cellulose communities), a bunch of curves is drawn (solid lines), and sigmoids are fitted to them (dashed lines). Ideally, the solid lines go well beyond the horizontal dashed line, and approach 1. These samples get close, but from the fitted lines we can also infer that it would be valuable to sequence approximately 10-fold deeper.

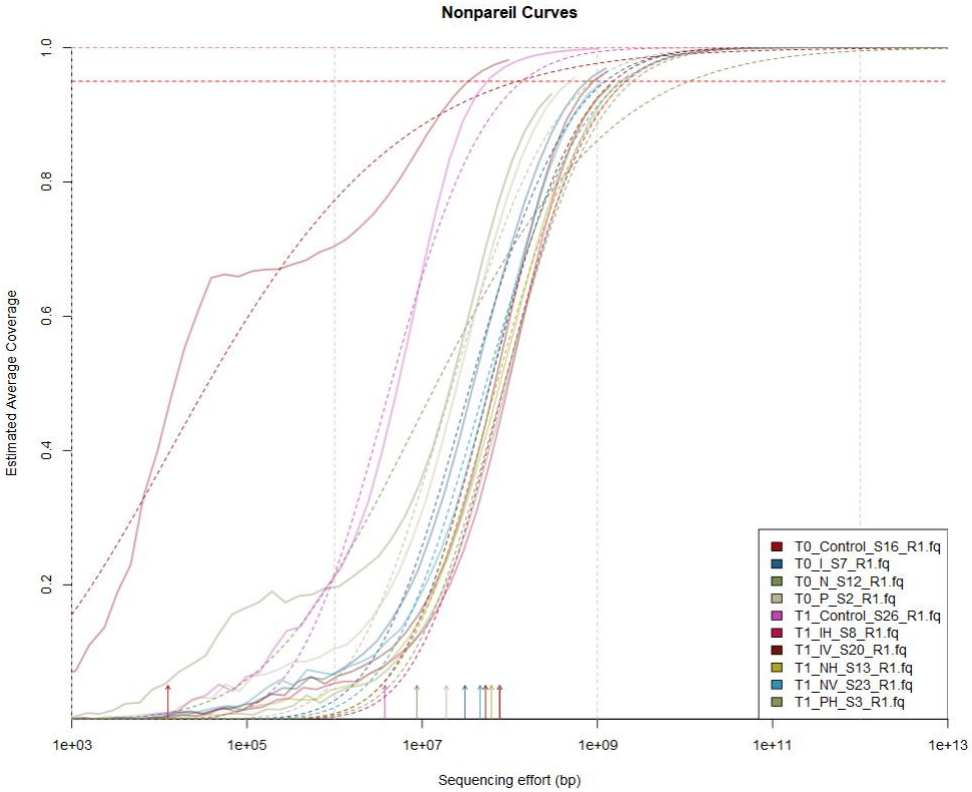


Figure 4 - Nonpareil curves for Pauline Buffards cellulose community data.

You can also run nonpareil online, if you are already tired of the command line:

http://enve-omics.ce.gatech.edu/nonpareil/submit

4. Too messy, too soon: why I don't recommend annotating reads

After having checked sequencing efforts and read quality, a common thing is to directly annotate the reads (what biological taxon they belong to). Tools like Kraken will align all your reads to a database, and tell you how many reads will map against varies taxa. I do not advice doing this[4], for a few reasons. First of all, reads are short. If a good representative sequence is present in the database, this will not be a problem as 150 base pairs are unlikely to align to anything else. However, more often than not, there are multiple hits in a database. Sometimes one is clearly "best", but often they are all mediocre.

But there is an even bigger problem. Many species in the databases are not sequenced in the same way. Some of them are carefully sequenced and annotated, and others are just some arbitrary "Vibrio sp." that was picked up in a metagenomic screening. The authors may have identified hundreds of species, with one of them being annotated as Vibrio based on unsupervised phylogenetic analysis (*e.g.* by an automated pipeline). However, other bacterial genera also live in this community, and bits of DNA from these bacteria may accidentally have snuck into this metagenome (it happens!). Moreover, this Vibrio may live in/on a bobtail squid, and a few squid-reads may have snuck in too! This is how you end up concluding that there is 0.1% of another species in your community. But how can you ever tell the difference between the rare species *really* being there, and erroneous assignments? With read annotation, you hardly can. Even if the database is relatively clean, there will always be risks of single reads that are either wrongly mapped, or have a wrongly annotated reference in the database.

Another issue I have with this approach is that it is distracting our ape brain by trying to assign a label to a "species", even though as discussed, gene content does not always correlate well with species. This means that "function" and "species" cannot be derived from one another anyway. Although it is always interesting to know which species you may be looking at, I suggest doing all this "annotation stuff" much further down the line. Let's first look at the sequences themselves, assemble them, study their abundance, and perhaps what they are doing a little while later (if you have time-resolved data).

[4] If you are working with very well-characterised samples such as the gut microbiome, you may be able to get away with this

4. Assembling reads into contigs

Now comes the most important part of metagenomics: assembling all the tiny reads into longer **contig**uous DNA sequences (called contigs for short). The process is quite easy to imagine: reads that show overlapping flanks can be merged into larger chunks. This naïve process is called "greedy extension":

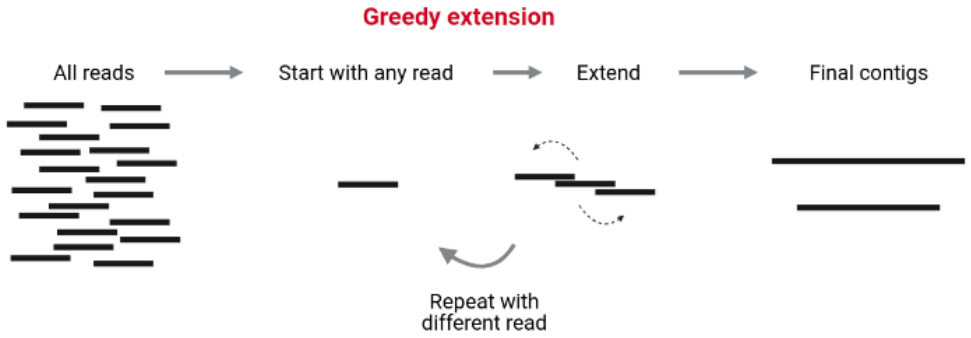


Figure 5 – Assembly process with greedy extension. Image from https://training.galaxyproject.org/

However, programs that are written to assemble are a bit more sophisticated nowadays. Greedy algorithms are really slow, and it would take forever to assemble the samples we typically work with this way. The most popular method of assembly is now the usage of De Bruijn graphs (no I didn't misspell De Bruijn, the picture below from the Galaxy website is wrong). De Bruijn graphs are based on Kmers (in the example below, 4-mers), and a path is traced through the kmers where basepair 2 to 4 match base pair 1 to 3 on another kmer. This operation is computationally very cheap, and when applied to a large batch or reads will give you possible paths through the kmers that will yield a large sequence:

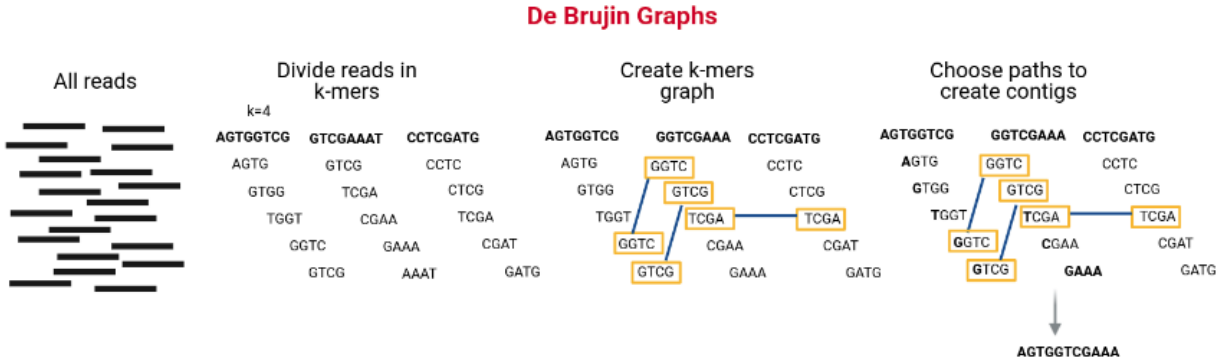


Figure 6 - Assembly process can be very efficient when using De Bruijn graphs based on Kmers, rather than having to align all reads with all reads. Image from: https://training.galaxyproject.org/. If you want to know more about De Bruijn graphs, see https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5531759/#BX1

However, any graph can contain bifurcations: what does the algorithm do when it can go two ways? The simplest solution would be to just split the sequence here, and make two new sequences. However, this then yields three contigs (before the split, and the two after), while one of the paths is likely the one we are interested in. So which information can we use to know which direction we should extend the contig in?

Here, most tools typically rely on kmer frequency and kmer coverage. Each contiguous path through the De Bruijn graph is associated with a certain number of reads representing them, information that is stored while generating the De Bruijn graph. If two paths are connected and have very similar coverage patterns and kmer frequencies, they are likely to belong to the same DNA molecule. If not, then the contig cannot be extended, as they may be part of different DNA molecules (a plasmid and a chromosome, or two chromosomes of different species). Such breaks can also occur when DNA is repetitive, as this impacts the kmer frequencies and kmer coverage. At the end of the day, contigs are much larger than reads (hundreds of thousands of base pairs), but they are **not** anywhere near "genomes". Note that sometimes two paired reads end up on a separate contig, but because they are a read pair, the two contigs can be merged because of this extra information. Such combined contigs are then often referred to as **scaffolds**, which will contain a NNNNNN-bridge to fill the gap between them. For this workshop, we will not work with scaffolds.

5. Hands-on: assembling reads into contigs

If you have trimmed your reads, and you are confident in the quality, you can start assembling reads. For this course, we will use megahit, which is a rapid assembler that does not use a lot of memory[5]. Megahit assembles reads by means of generating de Bruijn graphs based on kmers (see previous section). The basic command for running megahit is:

```
$ megahit -1 forward_reads.fastq -2 reverse_reads.fastq -t <threads> -m <memory> -o Output_dir
```

Depending on the computer you are working on, you can adjust the -t option to use more CPUs than the default. The memory option (-m) sets RAM limit. If this is overflown, an error is thrown. This memory can be given in a number of bytes, e.g in scientific notation -m 10e9 to use 10 gigabytes of memory. You can also give a number between 0 and 1 to use a percentage of the memory on the current computer, e.g. -m 0.8 to use 80% of memory. As a general advice: **try to never use 100% of the CPUs or memory** in your computer. Not only will that not allow you to do anything else with the computer, but it also yields diminishing returns because it becomes increasingly hard for the computer to "schedule" or "reserve" available CPUs/RAM. Using 80-90% is usually a sweet spot. Note that there are a lot more options in megahit, such as the kmers used in the de Bruijn graph construction and the option to remove small contigs from the output. For this workshop I won't discuss these, and Google is your friend. Generally, the defaults or other presets given by the developers are pretty good, and my own experimentation with the options mostly made the results worse :')

In the directory, there is a script that assembles all the available reads. Instead of looping over the files (as we did before), this script makes a custom function called "assemble". A function is a set of instructions that can be repeatedly called on different inputs. The pseudocode for this script is:

1. Define a variable for the read directory
2. Define a function called 'assembly' that takes a sample name as input, without the _R1.fastq and _R2.fastq extensions. Then, it calls megahit using -1 <SAMPLE>_R1.fastq -2 <SAMPLE>_R2.fastq as the inputs, and -o Assembly_<SAMPLE> as the output.
3. Call the function defined in step 2 on all samples

Although megahit is quite fast, doing this step for all samples during the course will take some time (it still took 3 hours for me). For that reason, the output is already there for you in the directory "03_Assembly_output", which contain directories for all 4 samples and in those directories, and a file called 'final.contigs.fa'.

So, I don't recommend that we all run the script during the workshop, but you can use and modify it to work for your own data in your own time! :)

[5] Note that other assemblers have shown to be more accurate, so if you have time and memory on your hands, I recommend you try metaSPAdes instead.

6. Statistics to assess assembly quality

Now that we have contigs and scaffolds, how do we quantify how "good" the assembly is? It is hard to capture this in one number, as the average length probably isn't very informative, with so many different players in the sample (bacterial genomes that vary in length, plasmids, phages). There are however a couple of useful statistics that you may want to check, and they can all be generated with the program 'bbstats', part of the BBMap package and natively installed on our computer cluster. All you do is type the name of the script (bbstats.sh) and give it the contig/scaffold file:

```
$ bbstats.sh 03_Assembly_output/Assembly_Tminus1_C1_powersoil_trimmed/final.contigs.fa –format=7
```

Bbstats can print information about contigs and scaffolds, but since this is a simple contig file, I added the last option (-format=7) to only print that information. The output should look something like this:

```
A       C       G       T       GC      GC_stdev
0.2165  0.2867  0.2839  0.2129  0.5706  0.1037

Main genome contig total:               70248
Main genome contig sequence total:      51.507 MB
Main genome contig N/L50:               12052/743
Main genome contig N/L90:               54622/353
Max contig length:                      480260
Number of contigs > 1 KB:               6108
% main genome in contigs > 1 KB:        40.22%


Minimum         Number          Total
Contig          of              Contig
Length          Contigs         Length
--------        --------------  --------------
     200                70248        51507245
     500                28086        35351508
    1000                 6124        20730498
    2500                 1413        14246144
    5000                  747        11913512
   10000                  314         8884571
   25000                   55         5034830
   50000                   28         4109315
  100000                   13         3095026
  250000                    6         1836172
```

We can quickly glance at some things. Apparently, the GC content is 57% (first line), and 6 contigs are longer than 250,000 base pairs (last line). But there's a lot more. Let's take a look at the N50 and L50 statistics. These two statistics can be a bit confusing, and the labels are quite confusing too. So let's discuss how they are calculated.

Usually, N50/L50 is calculated from a single genome by ranking the contigs by size:

From the above figure, we can see that **at least 5 contigs** are needed to span half the genome, and the shortest one of those 5 would be 30kb. Here's the confusing part: the Number of contigs needed to span half the genome is referred to as L50, and the Length of the shortest one is called N50. So… we're talking lumbers and nength here. But it gets worse: the developers of bbstats also got confused, and actually

use N50 to mean number, and L50 to mean length. Great. Thank's a lot, biologists. Usually you can figure out which is which based on context though. :')

In any case, these statistics can also be used for metagenomic samples, although we have to be careful not to draw any super strong conclusions from them (because we don't know the expected numbers!). In general, though, we know what we want: **large N50 and a low L50**. This means that very few contigs are needed to span half of your total assembly. Although you can't compare your N50 with the one from your neighbour that works in a different environment, you can still use these statistics to check which experimental protocol, sequencing technique and depth, trimming and assembly parameters etc. give you the best statistics. Note that **N90 and L90** are the same principles, but for 90% of the genome/assembly. However, for MGX one would not expect very good N90 and L90 scores, as there will be many smaller players like phages and plasmids that skew the sample to small DNA sequences. So I usually just look at N50/L50 instead.

7. How abundant are the contigs?

Mapping reads is perhaps where MGX gets really exciting and interesting. It is the Swiss army knife of any genomics analysis, but with MGX it can really open up to many insights, but it can also open a can of worms. So let's start with the basics.

While contigs can give you information on **what** sequences are present in the sample, it does not give you information on how abundant these sequences actually are in your sample. In fact, getting reliable abundance estimates in MGX is very challenging. First, let's peek at the first 10 lines of the megahit output (i.o.w. our contigs):

```
$ head 03_Assembly_output/Assembly_T1_C1_7amp_H_trimmed/final.contigs.fa
>k141_35866 flag=1 multi=3.0000 len=369
GCCATCGAGAGCATCGCGTTTCAAAGCGCCGATCTCCTCGCCGCCATGCAGGCCGACTCG…
>k141_15372 flag=1 multi=2.0000 len=354
CCCTCGTAATTGATGATTTCAGGCGGGTGATGTTCTATCCGAAAGCCATTGCACTGGGCAT…
>k141_0 flag=1 multi=2.0000 len=317
CGCGGCGCCGACCCGACGACATCGAGATATTCGGGAAAGGTTTCCCAGGTCCAGGGAAG…
>k141_5124 flag=1 multi=2.0000 len=456
TACTGTGCTCGGCACCAAGACATCATGTCCCCAGTTGCCGCGTCTCTTGCGGATGCCAGCT…
>k141_30743 flag=1 multi=2.0000 len=395
GGGCTCGCCTTCGTCGCCGAGCAGCTGCTGCCCTACCTGACCGGGCTGGGCGCCGAACCA…
```

As you can see, this is a regular fasta file. The headers contain some information about the contig. Every assembler may use slightly different naming, but you often find information about the length, kmer coverage, and sometimes the nature of the path in the de Bruijn graph (in megahit, this is what flag indicates, with flag=2 happening when you have a looped structure like a complete plasmid). The first keyword (k141_35866) is simply the name of this contig, with k141 being the longest kmer used to assemble this sequence (megahit starts with small kmers and then gradually goes up to improve assembly). The "multi" flag actually gives an indication of the abundance of this sequence, as it is an estimate of the kmer coverage. This is nice as an initial guess, but these are known to not be very accurate to get actual estimates of abundance, so we're going to have to do some work ourselves.

Like kmer coverage, we also know each contig should be represented by a certain number of reads. For example, a 50kb contig may be represented by 15,000 reads out of a sample of 1 million reads. That means that this contig represents 1.5% of the sample. But how do we know how many reads gave us this contig? Well, we can map the reads we used for the assembly, back onto the contigs. That may sound circular, but the truth is: assemblers using De Bruijn graphs simply don't give you this information. Luckily, like De Bruijn graphs, read mapping is also highly optimized, so by doing this two-step process way we actually end up saving a lot of time!

Before we go into the hands-on part, let me clarify some terminology. While people often refer to coverage as a single number (the number of reads on a single position, or the average number of reads mapping across the genome or contig), this is just one way of defining coverage. This particular coverage measure is usually called **depth** (sometimes this is called 'vertical' coverage). Depth can range anywhere from between 0 and thousands of reads, depending on how many different species are in your sample and how much DNA you were able to extract and sequence. When you are mapping reads back onto the assembly itself, depth gives insights into **relative abundances**: one contig may attract twice as many reads as another contig of the same size, indicating that the former is twice as abundant. I generally advice against making any statements about the **absolute abundances**, as there is no guarantee separate samples have the same sequencing efforts or underwent the same number of PCR cycles (if any) in the sequencing facility. If you really want to know the absolute abundance of a species, it is best to rely on a good experimental protocol instead (qPCR or colony screening).

Using depth as an estimate of abundance works very well when depth does not vary too much across the genome. This is usually the case when you map read back onto the assembly itself, because remember: the assembler also used coverage statistics to split contigs. So any contig is usually (i) completely covered and (ii) shows consistent depth across the sequence, like the one shown below.

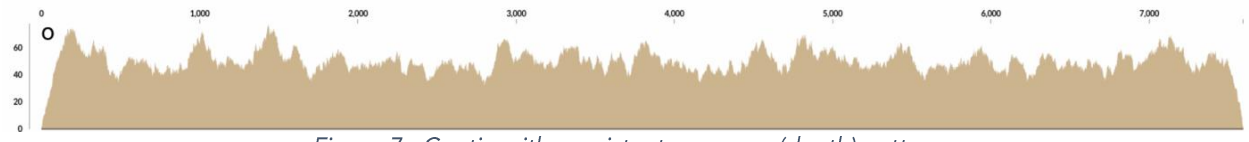


Figure 7 - Contig with consistent coverage (depth) pattern

However, you will sometimes find large variations in the coverage of a contig, with certain regions having much lower depth than others (see panel in the image below). This can happen when the assembler failed to split two closely related strains, and created a "chimera": a contig that represents multiple lineages in your sample. In the case of the image below, it indicates that one strain does not contain the middle region, but both contain the flanking regions.

Figure 8 – Strain diversity can result in areas of lower read coverage

The above problem can become more extreme when you map reads from different samples against your contigs. Perhaps this different sample has an entirely different strain with a deletion, and you get entire gaps in your coverage pattern:

Figure 9 - Mapping reads across samples can even have entire areas where nothing maps, indicating a deletion

Sometimes, a single conserved region (like 16s) can attract a lot of reads to a single position, while the rest of the contig is entirely absent. Under these circumstances, **depth** can clearly give the wrong impression of which contigs are present.

To solve the problem mentioned above, you may want to switch to a different coverage statistic: the **breadth** of coverage. The breadth describes the percentage of all positions that are covered by at least X reads (where X is variable, often set to 1). So, other than depth which could be any number, breadth is always a number between 0 and 100 (or 0 and 1, when it is instead a fraction). As mentioned, breadth is especially important when you do not know beforehand whether a certain sequence is expected to be present in a sample. This can also happen because you have a reference genome (from NCBI or isolated yourself) for which you are testing the presence in the sample. In this case, you want to make sure your sequences are scoring well in terms of both **breadth** and **depth**. This way, when some areas are not covered (or worse, all reads are attracted to a single conserved region), you won't misinterpret that as the whole contig as being "present" in the sample.

8. Hands-on: Mapping reads back onto the assembled contigs

Okay, so now that we know how read mapping can give us insight into the presence of contigs, let's map reads back onto our own assemblies, and inspect the output. For this, we first use the script 04_map_back.sh. This is very similar to script 03, but now uses a self-defined function called "map_back", which uses the Burrows-Wheeler Aligner to map reads. For those who want to know how burrows-wheeler algorithms work, see https://youtu.be/4n7NPk5lwbI. For those who just want to map reads, all you need to know is this: BWA first generates an 'index', which can be used (an re-used) to very rapidly map reads onto your contigs or genome.

Let's take a look at the pseudocode of the script:

1. Create a function called "map_back" which takes three arguments: the contigs file ($1), the sample to map against it ($2), and an output directory ($3). The function will then first check whether the contigs file exists, and if not, print an error. If the contig file DOES exist, it first makes a **BWA index**, and then it maps the reads and stores the output of this in a file called "read_mapping.bam", stored in the output directory
2. Call the function on a sample (1 example given)

The output file is what is known as BAM-file, which stands for "binary alignment map". This is not a file humans can read. The human-readable version of a BAM file is a SAM-file, which stands for "sequence alignment map", which contains information on **every read** (not just the ones that mapped!). To read a BAM file in a SAM-format, we will use samtools. Since it is a large file, I pipe the output to `head` using the '`|`' symbol, so it will only print the first 3 lines:

```
$ bash 04_map_back.sh # already ran for you, takes approximately 10 minutes
$ cd 04_Mapping_back/T1_C1_7amp_H
$ samtools view read_mapping.bam | head -n 3
VH00578:2:AAAJVKHHV:1:1101:65040:12018  99      k141_29350      2320    60      151M    =
2497    328
CTACCGAACACCGCGGCCGTCACTGGCCACGGGGAGAACTTTACGGGGCGCGGTGTGCTGCGGTTCGCGGAAACACGCGGGTTTGAACGCCCTGTTTCT
GGTTCTGGCGCATAAACCGCCGACCGTCACGACTCGCTTTCCTGATCCATCT
CCCCCCCCCCCCCCCCCCCC;CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCC;CCCCCCCCCCCC;CCCCCCCCCCCCCCCCCCCCCC  NM:i:1  MD:Z:149G1      AS:i:149
XS:i:0
VH00578:2:AAAJVKHHV:1:1101:65040:12018  147     k141_29350      2497    60      151M    =
2320    -328
TCCACGGCCGTTTCCGGCGTTTCGTCCGGGCCGATGACGCCATCGCCCCACGCCTCAACGCACTGGGCGTGCGTGAGCCCGGTCCGCTGCTGCAGCAAC
GCAATGAACCGTGCCTTGAAGGCTTCTTGTGCCTGTCTATCCATTCCTGGCC -
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC;CCCCCCCCCCCCCCCCCC  NM:i:0  MD:Z:151        AS:i:151        XS:i:0
VH00578:2:AAAJVKHHV:1:1101:67710:12037  77      *       0       0       *       *       0       0
GGCCGGCAGGAGAGTAACTTCCACCCATACCCTCCGTGATCCGCAATTGTGGTCTGTAGAAACGCCGCAATTATACCGGGTAAAAACTACCGTAAAAGA
TAAAAAGACGGGCCGGGTTTGGGAACAGACCACCCACACTTTCGGACTCCGG
CCCCCCCCCC;C;CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC;C;CCCCCCCC;CCCCCCCCC-
CCCCCCCCCC;CCCCCCCCCC;CCCCCCCCCCC;CCCCC;CCCCCCCCCCCCCC  AS:i:0  XS:i:0
```

As you can see, samtools view shows a LOT of information, which I won't explain in great detail. See http://www.htslib.org/doc/samtools-view.html for more information. Instead of staring at the output of `samtools view` (which may be relevant for you in some other circumstance), let's instead look at some insightful ways to inspect the data. For example, let's look at `samtools coverage`. This requires the bamfile to be sorted, which means reads appear in order from the left-most coordinate on the first contig, to the right-most coordinates on the last contig. For this, we use samtools sort. Together, we can then do:

```
$ samtools sort read_mapping.bam > read_mapping.sorted.bam
$ samtools coverage read_mapping.sorted.bam | head
```

Which will give:

```
#rname startpos endpos numreads covbases coverage meandepth meanbaseq meanmapq
k141_35866      1       369     9       369     100     3.30623 33.4    60
k141_15372      1       354     6       354     100     1.92373 33.7    60
k141_0  1       317     6       317     100     2.50473 33.1    60
k141_5124       1       456     9       456     100     2.83553 33.3    60
k141_30743      1       395     9       395     100     2.51646 33.8    46.7
k141_25620      1       489     10      489     100     2.79755 33.2    60
k141_10248      1       621     18      621     100     4.34944 32.4    60
k141_15373      1       352     7       352     100     2.3892  31.7    60
k141_20496      1       652     10      652     100     1.96933 33.6    60
```

This gives you insight into both read depth (the meandepth column), and also the breadth (covbases). Again, when mapping reads back onto the assembly, covbases **should**[6] be identical to the length (here, endpos) of the contig. As you can see, these are all really short contigs. The majority of contigs in any assembly are always short. This makes sense because even if only 1% of the reads did not get assembled (or only got connected to one other read), that is still tens of thousands of sequences. If one contig is really large (>200kb), that is an exception. If you want to mostly inspect the longer contigs, you can sort the output of samtools coverage by piping it to the `sort` command, and then showing the top 10 lines using `head`.

```
$ samtools coverage read_mapping.sorted.bam | sort -rn -k 3 | head
k141_35817      1       322628  100306  322628  100     46.4404 33.3    60
k141_5465       1       305961  89755   305961  100     43.7398 33.3    59.8
k141_9919       1       278561  85350   278561  100     45.7621 33.3    60
k141_6423       1       271521  79942   271521  100     43.9838 33.3    60
k141_15895      1       263632  85145   263632  100     48.2665 33.3    59.8
k141_21575      1       257785  73636   257785  100     42.6497 33.3    60
k141_20655      1       218689  73607   218689  100     49.9409 33.3    59.6
k141_16758      1       203518  62069   203514  99.998  45.5716 33.3    60
k141_4745       1       190366  54700   190366  100     42.8967 33.3    60
k141_7765       1       175419  49735   175419  100     42.3281 33.3    60
```

Of course, you can also just import the output into R or python and sort it for yourself. If you want to store the whole output (and not just the head), you can pipe your output to a file like this:

```
$ samtools coverage read_mapping.sorted.bam > read_mapping_coverage.txt
```

The extension (.txt) does not really matter here, but since the output is simply plain text I gave it the txt extension. You can call it whatever you like. If you want to know what the columns mean, you can find that in the help-page of samtools coverage by typing it into the console without any further arguments:

```
$ samtools coverage
Usage: samtools coverage [options] in1.bam [in2.bam [...]]

Input options:
(…)

See manpage for additional details.
  rname       Reference name / chromosome
  startpos    Start position
  endpos      End position (or sequence length)
  numreads    Number reads aligned to the region (after filtering)
  covbases    Number of covered bases with depth >= 1
  coverage    Percentage of covered bases [0..100]
  meandepth   Mean depth of coverage
  meanbaseq   Mean baseQ in covered region
  meanmapq    Mean mapQ of selected reads
```

[6] There are exceptions to this, because assembly is a strange process sometimes. So always check! :)

While samtools coverage is a nice way of summarizing the coverage, you only get averages and other summary statistics. If you want to actually visualize the read depths however, you need the read depth per position. This can be done with samtools depth:

```
$ samtools depth read_mapping.sorted.bam | head -n 20
k141_35866      1       2
k141_35866      2       2
k141_35866      3       2
k141_35866      4       3
k141_35866      5       3
k141_35866      6       3
k141_35866      7       3
k141_35866      8       3
k141_35866      9       3
k141_35866      10      3
k141_35866      11      3
k141_35866      12      3
k141_35866      13      3
k141_35866      14      3
k141_35866      15      3
k141_35866      16      3
k141_35866      17      3
k141_35866      18      3
k141_35866      19      3
k141_35866      20      4
```

In this output, the first column is the contig ID, the second is the position along the contig (ranging from 1 to the length of the contig), and the third is how many reads covered that particular position. As we did with the coverage, let's store the whole output in a file again:

```
$ samtools depth read_mapping.sorted.bam > read_mapping_depth.txt
```

I won't go into great detail on how to use R/python to plot something like this, but there's an R-script that plots the longest 3 contigs available at 04_Mapping_back/plot_depth.R. Notice that these files are really large (equal to the total length of all contigs!), so it can take a while. In any case, this is what the results look like:

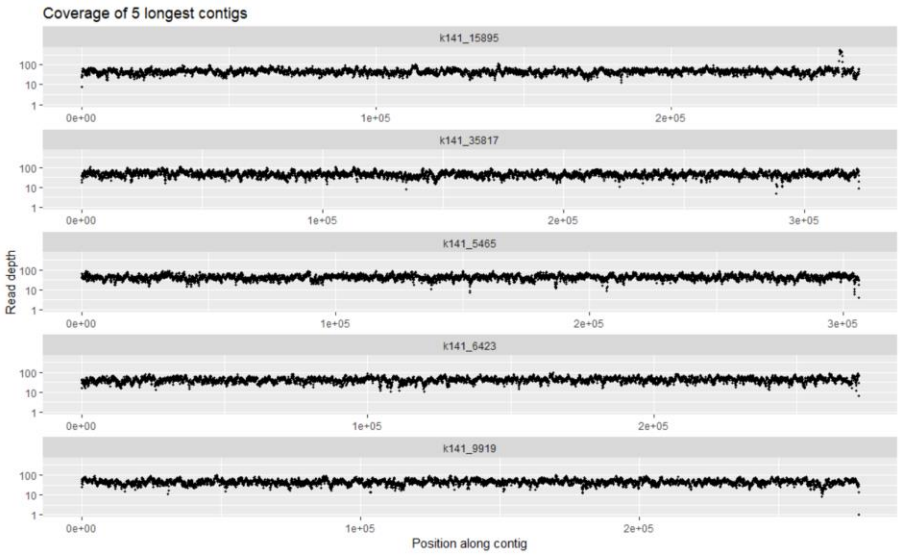


Figure 10 - Read depth visualised for the 5 longest contigs

Inspecting figure 10, you may see that the contig in the top panel has 1 region with much higher coverage than the rest (notice the y axis is log scaled, so it's even worse than it looks!). What do you think could be the reason for this?

9. Metagenome binning

As discussed in the earlier sections, contigs are not genomes. If you get lucky, you may have assembled a complete plasmid or a complete phage, but you will not likely get entire bacterial chromosomes back. But perhaps a set of 20 contigs together can form a set, that covers an entire chromosome. The process of figuring out which contigs "belong together" is called **binning,** and it's one of the biggest current challenges in MGX. Because how can you really tell which ones belong together?

The most naïve approach (and one that is still quite popular) is simply lumping contigs with similar coverage statistics together. While your sample might be quite diverse, what are the odds that two contigs have precisely the same coverage statistics? They are not 0, but that doesn't mean the approach is entirely meaningless. In the next section, we will use a program that uses this approach. However, before we move on, I just want to quickly talk about two alternative methods that can improve binning/assembly of longer contigs:

Hi-C metagenomics:

While it takes a lot more work, binning can be improved by gathering more information about the sequences experimentally. One such approach is Hi-C metagenomics (see Figure below). Hi-C metagenomics requires extra experimental steps, where the DNA is cross-linked, then digested, and subsequently ligated again. Since this can all be done without lysing the cells, the resulting chimeric sequences can be inferred to be from the same compartments. This information can be used to link phages to hosts, but also to link up bacterial contigs that belong together in 1 bin!

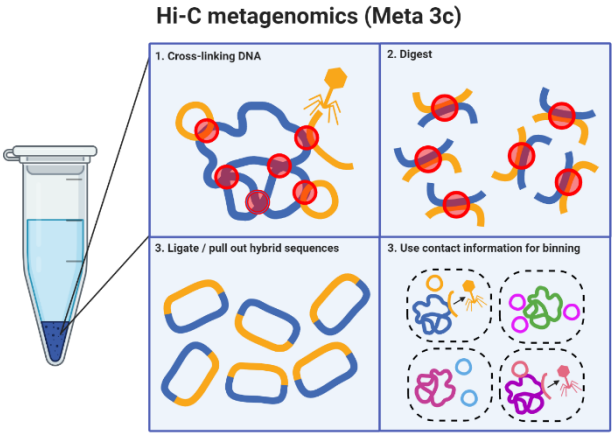


Figure 11 - Hi-C can give more information for "binning"

Hybrid assembly:

Another way to get longer contigs (while this is not technically "binning"), is to combine short-read sequencing with long-read sequencing. While long-read sequencing typically comes with very high error rates, it can yield a lot of information to bridge the overlaps between contigs, not only improving the lengths of the contigs, but also reducing the risk of chimera's:

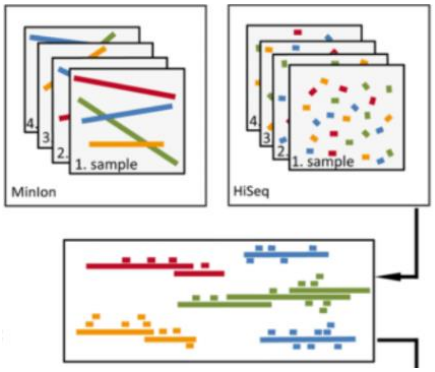


Figure 12 - Hybrid assembly of long- and short reads

Metagenome-assembled genomes from only Illumina reads:

While the above two strategies are great, they require more experimental efforts. So for this course, we will stick with binning-approaches that do not rely on these extra steps. Instead, we will stick with the naïve approach discussed earlier: using **differential coverage and tetranucleotide frequencies** to determine which contigs belong together. After binning (and perhaps some refinement steps like deduplication which I won't discuss), we will have what is commonly referred to as "Metagenomic-assembled genomes", or MAGs. It is possible that such a MAG is a good representation of a genome or

species, but should always be remembered that we don't know for sure. A MAG could be contaminated with contigs from other species, or a MAG could be incomplete. In the next section, we will use **metabat** to create bins ("MAGs") from our data, and investigate the completeness and contamination using CheckM, a tool that uses **single-copy marker genes** to estimate whether or not the genome is "**complete**" (every marker gene present), and whether it is **contaminated** (single-copy marker gene present more than once!).

10. Hands-on: binning and assessing completeness/contamination

In this part, we will first use metabat to bin our contigs into bins, or "MAGs". Metabat is a simple binning algorithm that uses two statistics for binning: tetranucleotide frequencies, and abundance estimates:

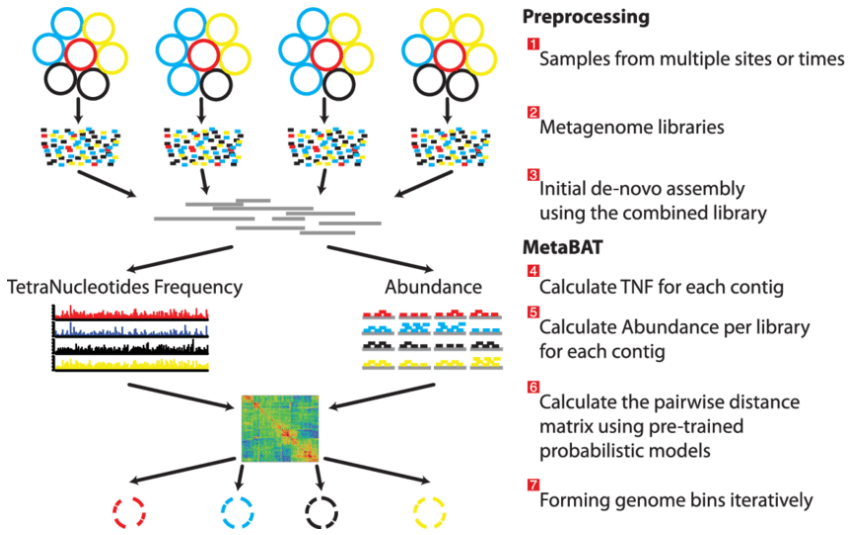


Figure 13 - Metabat uses tetranucleotide frequencies and abundance to estimate which contigs belong together

In the previous section, we generated bam files that could yield abundance estimates. However, metabat wants these statistics in a particular file format, which can be generated by the script (that comes with metabat) called `jgi_summarise_bam_contig_depths`:

```
$ cd 05_Binning
$ jgi_summarize_bam_contig_depths ../04_Mapping_back/T1_C1_7amp_H/read_mapping.sorted.bam --
outputDepth metabat_abundance_file.txt
$ metabat -i ../03_Assembly_output/Assembly_T1_C1_7amp_H/final.contigs.fa -a
metabat_abundance_file.txt --unbinned -o MAG
MetaBAT 2 (v2.12.1) using minContig 2500, minCV 1.0, minCVSum 1.0, maxP 95%, minS 60, and maxEdges
200.
4 bins (11738609 bases in total) formed.
```

Metabat has formed four bins. Indeed, by listing the directory's contents we see..

```
$ls
MAG.1.fa  MAG.2.fa  MAG.3.fa  MAG.4.fa  MAG.lowDepth.fa  MAG.tooShort.fa  MAG.unbinned.fa
metabat_abundance_file.txt
```

… that there are four numbered MAG files, and some left-over bins for contigs that were short, too poorly covered, or unbinned for other reasons. Remember the bbstats tool we used before? Let's inspect one of the MAGs with bbstatsh.sh:

```
$ bbstats.sh MAG.2.fa -format=7
A       C       G       T       GC      GC_stdev
0.3269  0.1734  0.1728  0.3269  0.3462  0.0097

Main genome contig total:               78
Main genome contig sequence total:      3.096 MB
Main genome contig N/L50:               16/74535
Main genome contig N/L90:               48/19106
Max contig length:                      167244
```

```
Number of contigs > 1 KB:                78
% main genome in contigs > 1 KB:         100.00%


Minimum         Number          Total
Contig          of              Contig
Length          Contigs         Length
--------        --------------  --------------
   2772                    78         3095686
  10000                    62         2998612
  25000                    40         2624293
  50000                    22         1960566
 100000                     6          773507
```

From the bbstats.sh output we can try to infer whether this MAG indeed represents a single genome. The total size is around 3MB, which is totally reasonable for a bacterium. The N50[7] is also really low (16), and L50 is quite high (74535). All these things indicate that MAG.2.fa is of high quality, perhaps close to being a 100% complete genome! But let's not only look at the assembly statistics, and also use CheckM (see previous section for a brief introduction). CheckM has various workflows for assessing completion, but we will use the lineage-specific workflow (`lineage_wf`), which (i) places the genome in a reference genome tree, (ii) finds a set of suitable **single-copy** marker genes, (iii) analyses the number of times these marker genes are identified in the genome of interest. By default, checkm will run on all *.fa files in a directory, so this will include the "bad" MAGs such as the one called MAG.unbinned.fa, and MAG.tooShort.fa. But let's run it an all of these too, which can serve as a reference for what "bad" completion/contamination looks like: / uncontaminated)

```
$ checkm lineage_wf -x fa -t 8 . CheckM
```

This will run the CheckM lineage workflow for all files with the 'fa' file extension (-x fa), using 8 threads (-t 8), in the current directory (.), and stores the output in a folder CheckM. When CheckM is done, it will print a summary for each MAG, sorted by completeness:

Bin Id	Marker lineage	# genomes	# markers	# marker sets	0	1	2	3	4	5+	Completeness	Contamination	Strain heterogeneity
MAG.3	o__Cytophagales (UID2936)	47	454	336	1	445	7	0	0	1	99.70	2.53	5.88
MAG.2	p__Bacteroidetes (UID2591)	364	303	203	2	299	2	0	0	0	99.01	0.74	0.00
MAG.tooShort	k__Bacteria (UID203)	5449	104	58	7	8	6	12	20	51	88.79	324.33	2.54
MAG.1	f__Xanthomonadaceae (UID4214)	55	659	290	88	562	9	0	0	0	84.68	0.83	0.00
MAG.unbinned	k__Bacteria (UID203)	5449	104	58	14	31	25	17	9	8	78.45	71.41	3.46
MAG.4	k__Bacteria (UID203)	5449	104	58	98	6	0	0	0	0	10.34	0.00	0.00

From this, we can see that MAG.3 and MAG.2 are placed into the marker lineages Cytophagales (order) and Bacteroidetes (phylum) respectively. As we already hastily concluded above, MAG.2 has high completion, and MAG.3 has even higher completion, meaning that almost all marker genes are accounted for. They also have a little bit of contamination, meaning that some marker genes occur twice, while no other close genome in that lineage has that gene twice. Although it could be possible that the sequenced MAG is the first exception that does have this gene twice, these marker genes are finely curated. Thus, contamination (wrongly binned contig) is a likelier explanation. While MAG.1 still scores okay-ish (a 85% complete Xanthomonadaceae with low contamination), MAG.4 looks pretty incomplete. Moreover, the "marker lineage" it was assigned to is the entire kingdom of bacteria, which basically means that there is not enough information to know where to place this genome in the tree of **all bacteria**. Although this genome has very low contamination, this only means that no contigs were annotated outside of the kingdom of bacteria, which isn't too impressive. Perhaps it is a rare species that was not captured sufficiently. Finally, the left-over MAGs ("MAG.tooShort tooShort" and "MAG.unbinned") both have very

[7] Again, remember that bbstats flipped these concepts that were already confusingly named…

high contamination, which is what we would expect given that this is almost certainly a set of contigs from various species. Also, these were not placed in the tree in a lower-level taxonomic rank.

Although the marker lineages give us some information on on what players we might be looking at, this is still not super informative. We may want to take a deeper dive into the **gene contents** of these players.

11. How to improve the assembly of rare species?

In the previous section we saw that a single compost sample had two or three "good" genomes in it and a lot of junk. This is very often the case in metagenomics, especially in samples with a lot of rare species. We can visualize by plotting the abundance of identified genera (which could be done by 16S or using marker genes, as done above), and sorting the identified genera by their abundance. The resulting rank abundance curve for compost looks like this:

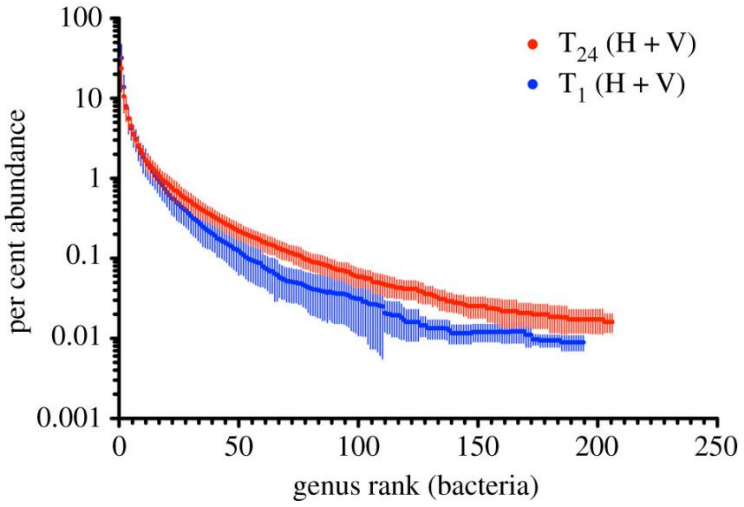


Figure 14 - Rank abundance curve for compost before (blue) and after (red) a 1-year experiment

Looking at these data, we can see that only a few genera in the sample constitute about 99% of the total diversity. However, many genera are rare, with much less than 1% abundance. While it is unlikely that we will get complete genomes of all of these, there is a technique that can drastically improve their detection.

Cross-assembly (sometimes called co-assembly)

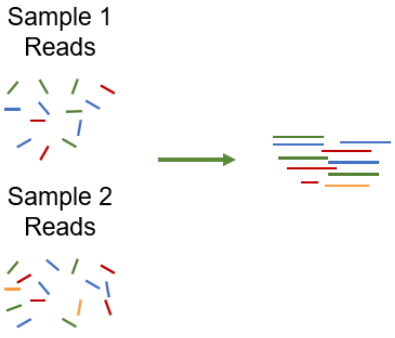


Figure 15 - Cross-assembly by combining multiple samples

Consider why the genomes of the rare species will not assemble very well: we simply did not extract enough DNA from a single sample. The naïve thing to do here is to sample deeper and deeper, which will of course help. However, very often you have 100 samples, and sequencing all of these super deep will cost you a fortune. But if you have 100 samples, perhaps some of them are "similar enough" (such as time series samples) to be considered for cross-assembly: you combine the reads from multiple samples and assemble those. By doing so, you may have a few samples where those rare species are slightly more represented, which drastically improves their detection. But even if they were equally rare in all samples, the extra read information is likely going to help. Using the same concept of "back-mapping" reads discussed in an earlier section, we can then figure out in high-sight which species were present or abundant in which sample. Cross-assembly will generally give better/longer assemblies, but come at a greater computational cost (often you will need a high memory machine), and also has an increased risk of chimeras. That said, let's look at how one would do a cross-assembly.

The process for cross-assembly is surprisingly simple: concatenate all the forward reads, then concatenate the reverse reads, and feed the concatenated files to the assembler (*e.g.* megahit). So, something like this:

```
for sample in Sample_[1,2,3]*_R1.fastq; do cat $sample >> All_reads_R1.fastq; done;
for sample in Sample_[1,2,3]*_R2.fastq; do cat $sample >> All_reads_R2.fastq; done;
megahit -1 All_reads_R1.fastq -2 All_reads_R2.fastq -o CrossAssembly -m <MORE_MEMORY!>
```

Then, by mapping back reads you can tell which species is where. I will repeat myself though: cross-assembly comes at an increased risk of chimeras, so **make sure all coverage statistics look okay, and not merely inspect the average depth!**

12. Studying gene content

In the previous sections, we have implemented an "annotation-free" workflow. That is to say, without annotating reads or contigs, we have looked at the quality of reads, contigs, and MAGs (bins). This is the part where we start actually looking at the gene content, which will give us indications of microbial function. I want to give a disclaimer first: I am no expert on microbial metabolism, functioning, or anything related to "real" microbes (some of you may know I'm an expert on virtual microbes though ;P). I will therefore show you what kind of things you can do, but there will always be new tools and techniques, so make sure you check Google every now and then.

The first step in knowing gene contents is to know what open reading frames (ORFs) we have. The most popular tool used for this is called Prodigal, which has become part of more than half the tools that annotate genomes. In fact, I will not even show you how to use prodigal stand-alone, but we will use Prokka (https://academic.oup.com/bioinformatics/article/30/14/2068/2390517), a tool that identifies ORFs and assigns an annotation to these using hidden-markov models (HMMs). If no annotation is found, it will be annotated as a "hypothetical protein".

Prokka is very fast, so running…

```
$ prokka 05_Binning/MAG.3.fa --prefix 06_Prokka_MAG3
```

… only takes about 5 minutes. The output will consist of tables of genes, but also a gff file that you can import into software like Geneious prime or plot with packages like gggenomes (R) or clinker (biopython). For example, importing the gff file in Geneious give you this:

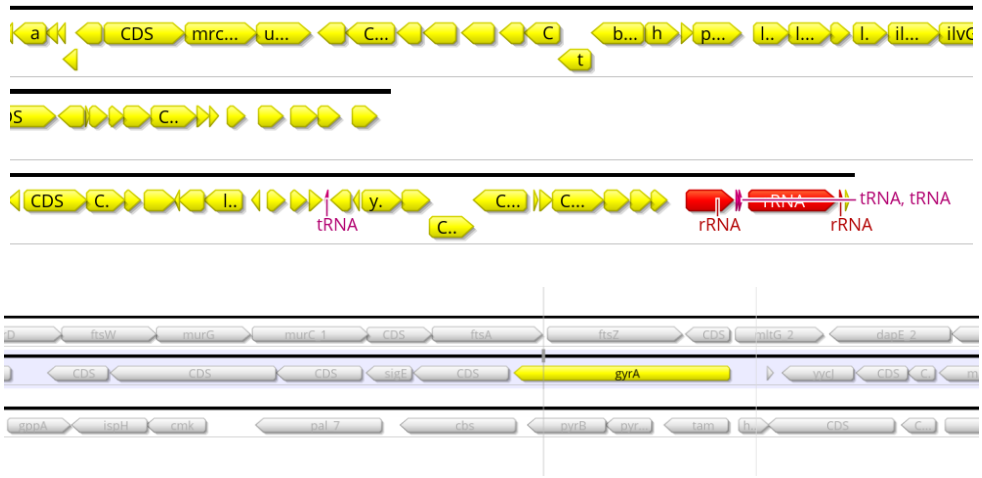


Figure 16 - Example output from Prokka. The top shows the identified 16/23s rRNA, and the bottom shows a house-keeping gene gyrA that should be present in all prokaryotic genomes. And lo and behold: it is.

While having annotated genomes is nice, you are often faced with massive amounts of proteins for which you have little to no knowledge. Because of this reason, protein content is often simplified in specific classes, with databases of COGs (clusters of orthologous genes), KEGG (Kyoto Encyclopedia of Genes and Genomes), or BioCyc. By doing this, you can for example analyse whether samples are enriched in certain types of genes, such as "carbohydrate metabolism" or "defense systems". Another thing you could do is to extract the UniProtIDs from the prokka output, which is part of the provided script 06_prokka.sh:

```
$ prokka 05_Binning/MAG.2.fa --prefix 06_Prokka_MAG2
$ grep -o 'UniProt.*' 06_Prokka_MAG2/06_Prokka_MAG2.gff | cut -d';' -f1 | cut -d':' -f2 | sed
's/^/UNIPROT:/g' > 06_Prokka_MAG2/UniProt_IDs.txt
```

The first line uses prokka to annotate your MAG, and the second line extracts UniProt IDs (for each ORF that has one) by searching for the string "Uniprot.*", splitting the string by ';' and selecting the first column, and then splitting the string by ':' and selecting the second column. Then, a sed command pastes UNIPROT in front of it because this is required by iPATH3, which can be used to visualize which metabolic pathways are present:

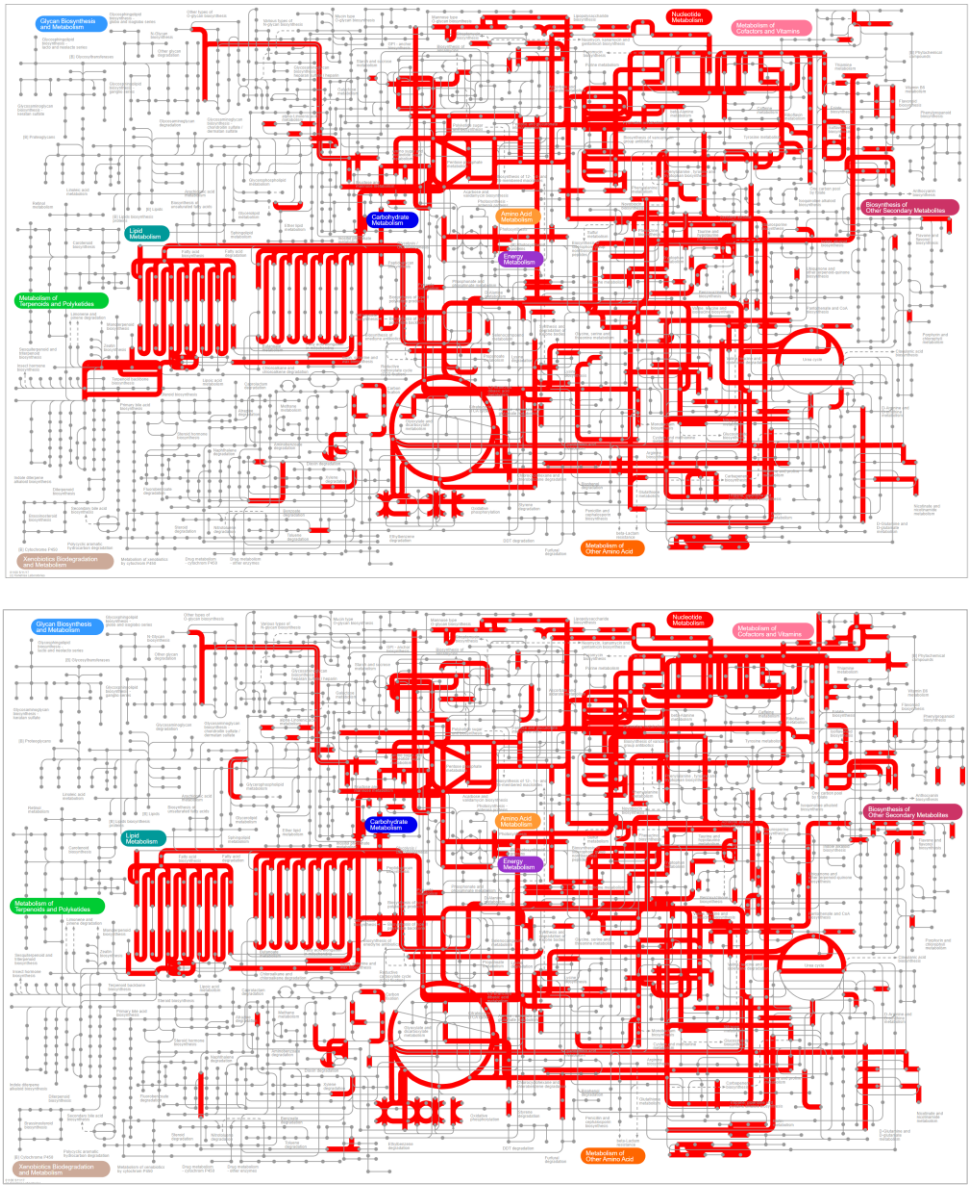


Figure 17 - iPATH3 can project UniProtIDs onto known metabolic pathways to visualise your MAGs after annotating them with prokka. The top picture shows MAG2, and the bottom one MAG3. Alternatively, you can try uploading the protein sequences to blastKOALA.

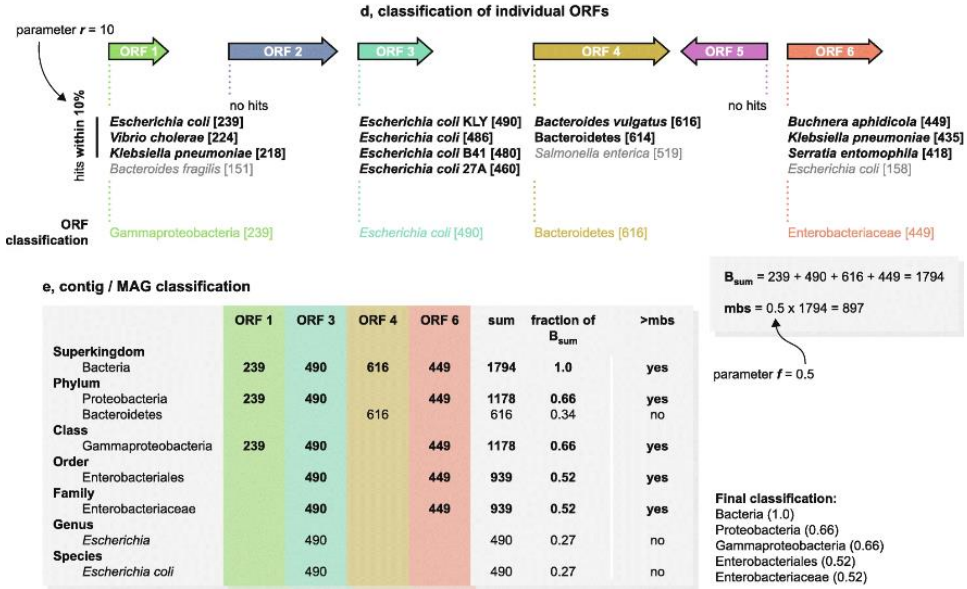
A few other tools that I recommend if you want to get into those types of analyses are reCOGnizer (for finding COG categories) and Superfocus (for finding SEED categories). There are a LOT of tools out there! (including some simple R/python packages)

If instead of wanting to "zoom out", you want to "zoom in" on a particular protein sequence, and you want to learn more about this protein sequence, I recommend HHPhred and Interproscan, tools that take amino acid sequences (that Prokka has provided in the .faa file) and give all kinds of extra insights. For example, I took an arbitrary hypothetical protein from the output above and fed it to HHPhred.

From this, I have hints that the protein may be a sulfate permease (https://www.rcsb.org/structure/3TX3), involved in cysteine biosynthesis. Interproscan will also give you domain-information and indeed confirms that the protein has intermembrane regions, further indicating we are looking at a transporter protein. I could do this all day. No, I can't, let's move on.

13. Annotation with CAT, RAT, and BAT

There are many read annotation tools out there, but as mentioned, read annotation is very tricky, because a single read may not contain enough information to reliably predict species. But contigs are a lot larger, and may therefore contain more information to reliably predict species. However, a contig contains many ORFs, and it is not guaranteed that all ORFs will be predicted to be of the same species. So how can we resolve conflicts between them? This is the problem solved by the contig annotation tool (CAT), which annotates each ORF individually and weighs the evidence against each other (see Figure from the paper below). Briefly, CAT does not only assign every ORF a species but also looks at the second-best hit, the third-best hit, etc. Then, all hits that are within a certain range of the best hit are collapsed into a single prediction. If all of these predict the same species, CAT reports that this ORF is of that species. However, if they are not, it will look they are from the same genus, and will assign a genus. If that fails, it will try to assign a family, *etc.* In other words, CAT attempts to assign the deepest possible taxon to each ORF. Then, a similar process happens to resolve the conflicts between ORFs, where a certain cut-off determines whether or not the entire contigs will be annotated at the species-level, genus-level, *etc.* CAT can also take bins (MAGs) as input, in which case it annotates collections of contigs and tries to figure out which single species this MAG may belong to. It is one of the newer tools in this field, and it is performing quite well in terms of accuracy. That said, it is still reliant on databases so a lot of stuff will still be unknown, or simply wrong. When writing papers about your MGX research, perhaps try to always use words like "predicted species" or "inferred function". If you want to be sure, you're going to have to try and isolate your critter and actually study it in more detail.



	ORF 1	ORF 3	ORF 4	ORF 6	sum	fraction of B_{sum}	>mbs
Superkingdom							
Bacteria	239	490	616	449	1794	1.0	yes
Phylum							
Proteobacteria	239	490		449	1178	0.66	yes
Bacteroidetes			616		616	0.34	no
Class							
Gammaproteobacteria	239	490		449	1178	0.66	yes
Order							
Enterobacteriales		490		449	939	0.52	yes
Family							
Enterobacteriaceae		490		449	939	0.52	yes
Genus							
Escherichia		490			490	0.27	no
Species							
Escherichia coli		490			490	0.27	no

Figure 18 - Contig annotation tool takes evidence from all ORFs, and predicts the most likely lineage with a corresponding score. The default parameters of r=10 and f=0.5 have been benchmarked to yield the most reliable information down to the genus level. Species-level, as with many tools, should always be taken with a grain of salt but can be a good indicator.

Remember when I said read annotation is problematic? Well, now that we annotated the contigs/MAGs with some certainty, we can of course translate this knowledge to the reads, and do much more reliable read annotation! This way, we can say not only which contigs are which taxon, but also get good abundance estimates (as mentioned in the read-mapping section, these are **relative** abundance estimates!).

Together, this swat of tools has been published as the contig annotation tool, bin annotation tool, and the read annotation tool. In other words: CAT, BAT, and RAT (sometimes stuff just works out).

14. Hands-on: CAT, RAT, and Krona

In the workshop directory, you will find a script called 07_Contig_annotation.sh. This script uses a combination of CAT, RAT, and a tool called "Krona" to generate interactive graphs of your assemblies. Unfortunately, contig annotation can take quite some time (at least a few hours, sometimes days), so I do not recommend running this during the workshop. But, the tool should be able to run on your data as well, as long as you have (i) trimmed reads, and (ii) assembled contigs. The script can then be run like this:

```
$ bash contig_annotation_tool.sh -c 03_Assembly_output/Assembly_T2_C1_7amp_H/final.contigs.fa -1
reads/T2_C1_7amp_H_trimmed_R1.fastq -2 reads/T2_C1_7amp_H_trimmed_R2.fastq -o
07_Contig_annotation_2
```

With the following options:

- -c for your contigs
- -1 for your forward reads (or single end reads)
- -2 for your reverse reads (if not given, script assumed single end reads)
- -o for your output directory
- (optional) --skip_cat, --skip_rat, and --skip_krona to skip subroutines if you already ran them in a previous run of the script

You could run this script on the workshop data, but it will take far too long. If you would to try in your own time, see the script "07_contig_annotation.sh". For now though, check out the data in the example output (/groups/mpistaff/MGX_Workshop/MGX_Workshop_Results).

The output consists of a few tables with annotations (which you can analyse further), the easiest way to inspect your data is to open up the file Krona.html which is found in the provided directory:

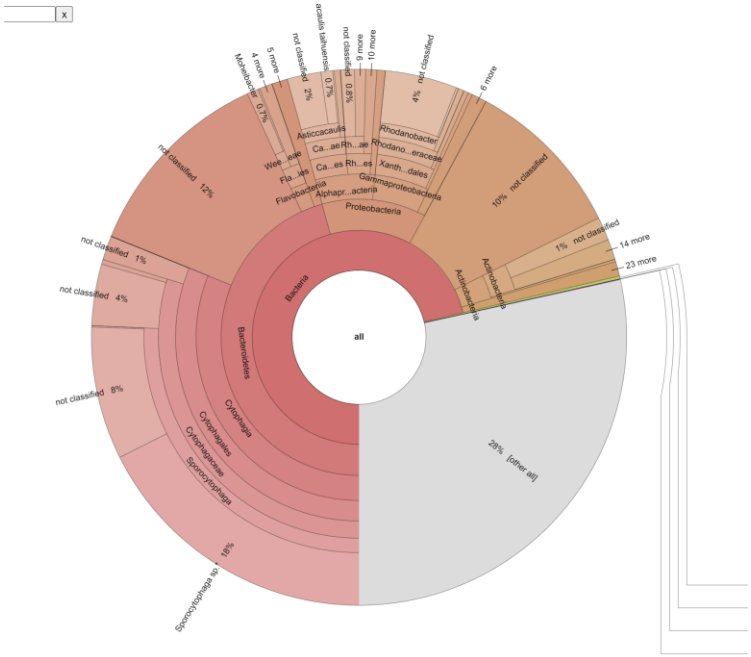


Figure 19 - Krona plot generated from the RAT output

From the figure above we can already see quite a lot. First of all, the sample contains mostly bacteria (71% of the sample), many unknown contigs (28%, which could be small contigs or actual unknown species!), and hardly any Eukaryotes and Archaea (although there are some if you zoom in, see the HTML file yourself). Of the bacterial kingdom, most of them belong to the phylum Bacteroidetes, class Cytophagia, etc. Note that the image above is static, but you can interactively explore the plot in your browser!

Another thing you may notice is that many sections are present with the label "not classified". While this indicates that this specific taxonomic rank could not be determined, higher taxonomic ranks are often present. For example, the 8% of contigs not classified as Sporocytophaga sp. **do** have the genus label Sporocytophaga. This could either indicate that another strain is present, or that these 8% were below the threshold that CAT deems reliable (this also depends on CAT's parameters -r and -f, which you could play with).

15. Detecting horizontal gene transfer

The last decade of metagenomics has revealed that the timescales of ecology and evolution are very much intertwined when thinking about microbes. Microbes are known to rapidly adapt to new environmental conditions, which in the lab can entail very simple single nucleotide substitutions (point mutations). However, recent advances have suggested that point mutations are by far outpaced by horizontal gene transfer (HGT) and gene loss. By the time a single point mutation can occur in any individual gene, multiple genes have come and gone (for more on this, see for example Puigbo *et al*, 2014). This indicates that in nature, microbes are mostly evolving by rapidly changing the contents of their genomes, rather than by gradual sequence evolution. Being able to identify events of HGT, and perhaps tracking how genes spread through communities, is therefore important to better understand the eco-evolutionary dynamics of microbes.

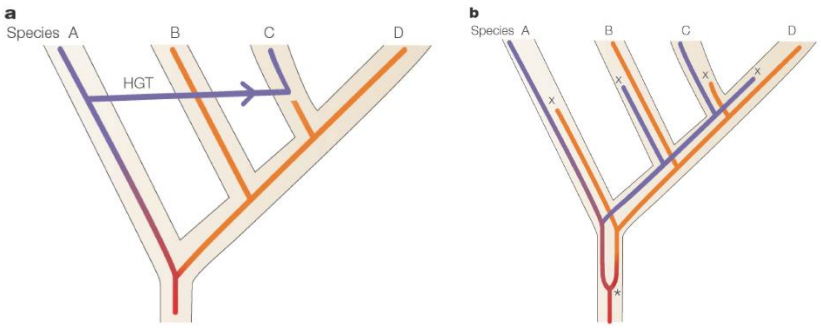


Figure 20 - HGT and differential gene loss can have the same outcome

One of the challenges in identifying HGT is discerning it from differential gene loss. To illustrate this, take a look at the example above. **Figure 20.a** shows how horizontal gene transfer can result in distinct alleles in two closely related species (C and D). However, **Figure 20.b** demonstrates that the same can happen through gene duplication followed by differential gene loss. How do we tell the difference between scenario a and b?

Every bacterial species has its own mutational biases, codon usage patterns, and nucleotide frequencies. These "genetic fingerprints" can be used to detect discrepancies in the genome. For example, if the GC-content is consistent across 59% of the genome, but one single island has a significantly reduced GC-content, that hints at horizontal gene transfer (**Figure 21.1**). Another "golden standard" to solve the problem described above is to generate two types of phylogenetic trees: one for the species, and one for an individual gene of interest. If the species tree and the gene tree are identical, that indicates no horizontal gene transfer has happened. If the trees are different (for example, two genes from two distant species appear close together on the gene tree), it is likely that HGT has happened (**Figure 21.2**).

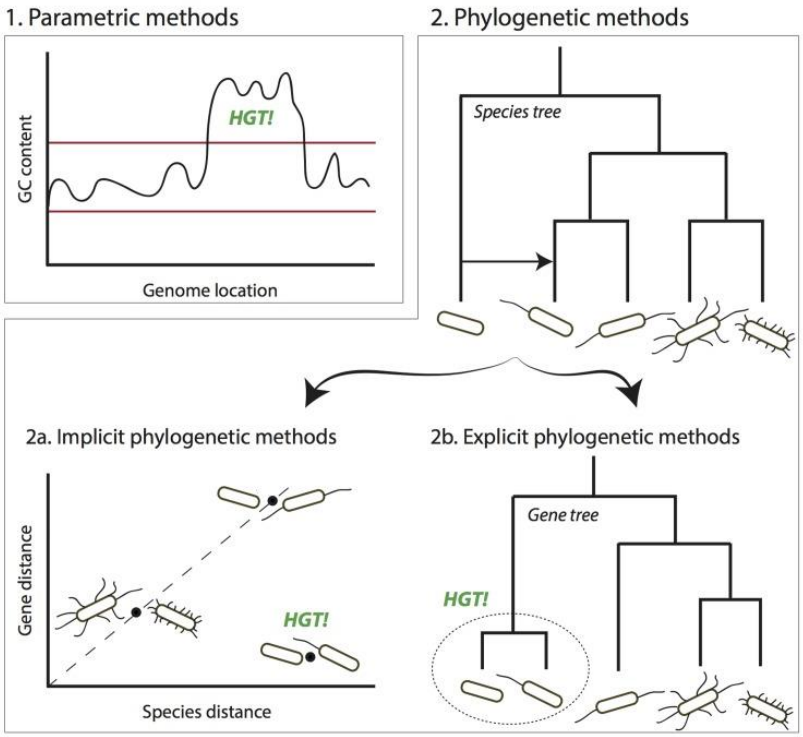


Figure 21 - Classical genomics has methods to detect HGT (Ravenhall et al., 2015)

While the above strategies can work to detect HGT, they share a set of common limitations. First of all, most of the approaches require having access to a large collection of genomes. When you discover two closely related genomes in a metagenomic dataset, it becomes unlikely to detect any statistically significant effect on the GC content, and you can of course not make a phylogenetic tree out of just two genomes! The tree-building strategy only works if the focal genes have accumulated mutations (such that a gene-tree can be made). Even if the GC contents are clearly distinct, this signal will eventually water out over time as the sequence slowly becomes adjusted to the host's nucleotide/codon biases. Eventually, these islands will get the same GC-content and other biases, making it more and more challenging to capture. While machine learning can help to detect even the weakest indication of HGT (e.g. by incorporating a lot of variables), it is possible that HGT between closely related strains (in short timescales) will fly completely under the radar. Since a lot of microbial diversity is understood to be at the level of strains, we may need alternative mechanisms to detect HGT. Metagenomics provides a few opportunities here.

Leveraging read mapping

As said, read mapping is the Swiss army knife of metagenomics, and it can therefore also help to detect HGT events between donor-acceptor pairs. As discussed in the section on read mapping, consistent areas with lower read coverage are indicative of strain-level diversity, where strain A has a certain stretch of DNA that strain B does not have. These variations are often referred to as structural variants (SVs). Evidence for SVs being the result of HGT can be improved by leveraging read pair information. If the forward read maps to the acceptor, and the reverse read maps to the donor, this is indicative of an SV. Similarly, while a single read may not map in its entirety, it is possible that the first half of the read maps very well to the acceptor and the second half maps very well to the donor (split mapping). Again, this

indicates an SV. Trappe et al. (2016) have worked a lot on improving such methods to detect (Figure 22). But what is the cause of the SVs? How do we know it's HGT, and when it happened?

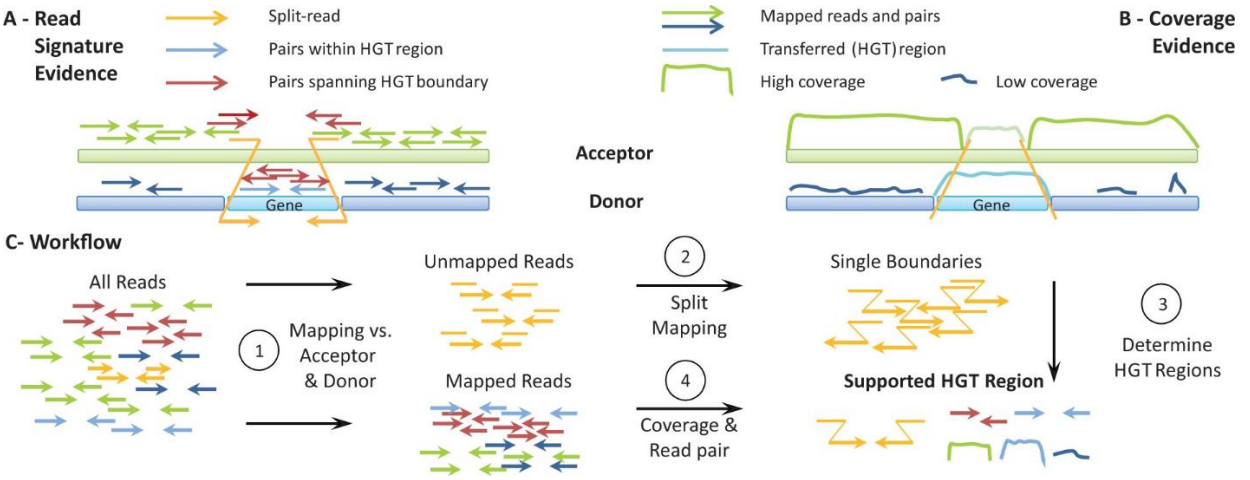


Figure 22 - Detecting HGT by using read (pair) information (from Trappe et al., 2016)

Leveraging time-resolved samples

Inferring causes can be very difficult in science. If we want to know for sure whether an SV is the cause of HGT, then it is probably best if we can catch the event in action. While some HGT events are rare (for example, between distantly related species), HGT events between closely related strains occur all the time. If, in this scenario, we have a metagenomic sample from **before** and **after** the HGT-event, this may help us in our search for evidence for HGT. To help figure this out, I have developed a pipeline during my time here at MPI, called Xenoseq. Xenoseq got it's name for its ability to detect what we call "xenotypic" sequence (xeno referring to foreign/other). Instead of relying on read mapping within samples (as is done by Trappe et al. 2016), xenoseq relies on read mapping **across** samples (see **Figure 23**). When mapping reads from an evolved community (a later timepoint) back onto an assembly of an earlier timepoint, the default expectation is that all reads map. While sequences in the original sample may have shifted, they should all be accounted for. However, when the parallel communities exchange DNA (as is the case in the experiment shown in **Figure 23a**), this expectation can be broken. DNA may have appeared "from the outside" (*i.e.* be xenotypic!), and therefore **not** map back onto the original assembly. Xenoseq tries to find these xenotypic sequences by mapping back reads to the original assemblies, assembling these unmapped reads, and figuring out whether or not they came from another community.

With the right experimental setup, for example, communities evolved in parallel, xenoseq can give you information on the flux of DNA movement between communities, rather than attempting to identify single donor-acceptor pairs. That said, it is not without its own limitations. For example, it can be hard to tell whether a sequence has really transferred, or just "missed" by your earlier sequencing efforts (**Figure 23b**). While xenoseq does some extra tests to try and reduce these issues, it is always good to further investigate whatever it spits out. So if we find some interesting sequences… then what? We could of course do what we did in previous sections: annotate with Prokka, or CAT, and see what we have. But neither of these tools are particularly well suited for detecting viruses, for example. So, let's talk about some tools that can show us more about the identity of these sequences.

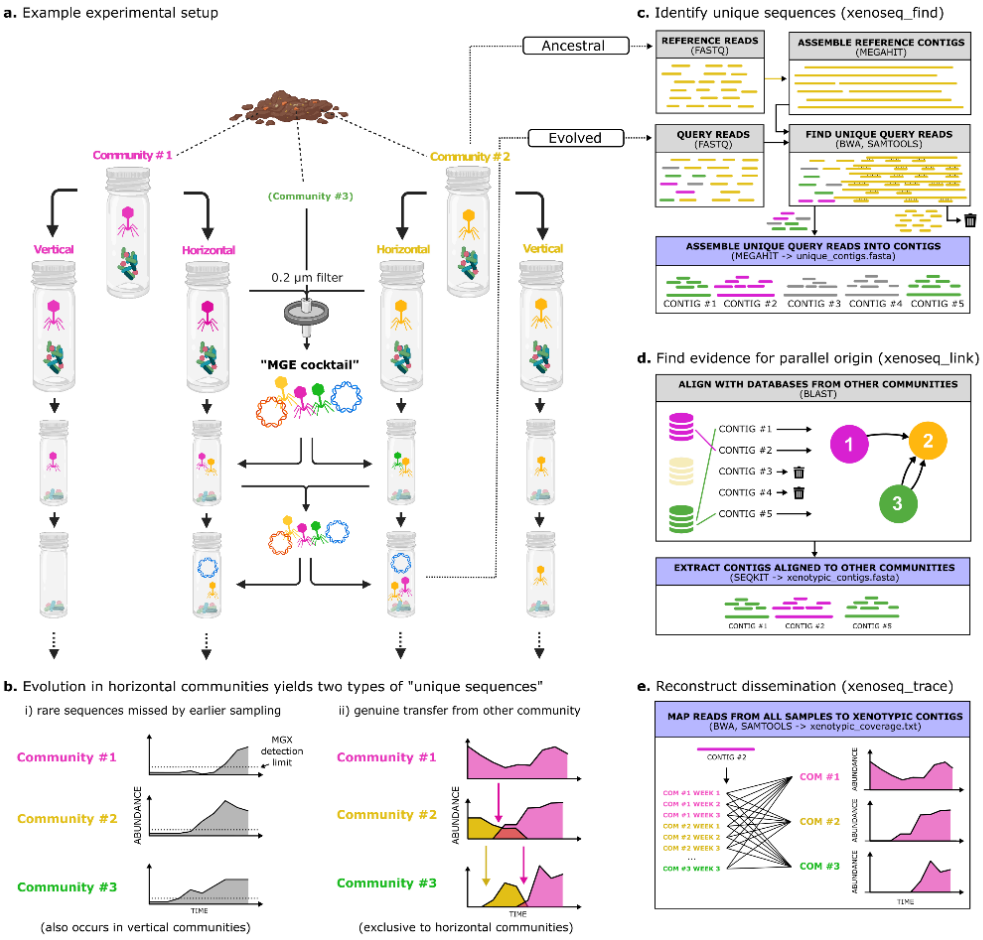


Figure 23 - Overview of the xenoseq pipeline

Detecting mobile elements

Mobile genetic elements (MGEs) all carry some marker genes. Viral genes carry integrases and head and/or tail proteins, plasmids carry partitioning systems, transposons carry transposases, *etc.* So it should be possible to know whether we are looking at either of these canonical MGEs. The list of tools available to detect such entities is far too long to share here, but I have compiled a shared document online. Many of these tools are cutting-edge, relying on combinations of machine learning, marker genes, assembly graphs, *etc.* For our compost data, I have applied nine different MGE detection tools 1756 xenotypic sequences identified by xenoseq, and found that only 40% of the sequences showed any indication of being a known MGE. Either these tools are still really bad at detecting MGEs, or, we have identified some interesting new MGEs that are moving DNA around!

16. Hands-on: Local blast and the xenoseq pipeline

While the xenoseq pipeline is designed to do a lot of work for you, all it essentially does is a combination of the steps you have already learned. It trims reads, assembles reads into contigs, maps reads against those contigs, and tries to figure out which reads appear to be newly introduced into communities. The only step we haven't discussed is running a local BLAST search. While you all have used BLAST using the NCBI web page, some of you may not know what a local BLAST search is. While the BLAST functions online compares your sequence against the non-redundant (NR) protein database to find evidence homology, you can also compile your own (much smaller) database and BLAST against that.

Doing a local blast search:

For example, if I want to know whether the sequence ACACTGCGTAC…CACATCAAA (the query) occurs in one of our assemblies (the database), I can make a database out of the assembly using makeblastdb and then use blastn to look for it. BLAST is natively installed on Wallace, and is also part of the 'xenoseq' environment that is provided with the workshop. So if you are in the MGX environment for the workshop, you may need to activate these commands before you can use BLAST:

```
$ source /usr/local/opt/miniconda/etc/profile.d/conda.sh
$ conda activate xenoseq
```

The randomly made-up sequence provided above is also saved in the workshop environment, saved under random_sequence.fasta. Note that the 'query' may also be a multifasta file, so it can contain many sequences that you want to query for at once. To see if the sequence is found, we need to first make a blast database out of the assembly we are going to look in:

```
$ makeblastdb -in 05_Binning/MAG.3.fa -dbtype nucl
```

The 'dbtype' argument should be set to nucleotides since the fasta file contains nucleotide (not amino acid) sequences. Then, we can look for the query:

```
$ blastn -db 05_Binning/MAG.3.fa -query random_sequence.fasta
```

If you did that correctly, you will see that this random sequence I just made up does *not* occur in MAG.3.fa, and the results will show *** No significant hits found ***. That is not unexpected, since I just smashed the keyboard randomly (which is actually not random, but whatever). However, when copy-pasting a bit of sequence from MAG.3.fa into a file (nonrandom_sequence.fasta), and searching for that, you will see that you get significant results:

```
$ blastn -db 05_Binning/MAG.3.fa -query nonrandom_sequence.fasta -outfmt 6
Sequence_From_MAG3      k141_5160       100.000 56      0       0       1       56      125     180
5.50e-24        104
```

The extra option '-outfmt 6' makes sure the data is printed as a neat table (see explanation here). This table shows that the sequence was indeed found in MAG.3.fa, with 100.00 percent identity (third column) over its entire sequence length (1-56). This is a good blast result, and as illustrated with the random sequence, it is unlikely to get ANY hit with an arbitrary sequence.

Xenoseq:

When you want to detect horizontal transfer between communities (*i.e.* in the H vs. V experimental design), you don't need to manually do any of the steps you have learned so far. It's good that you know them, however, because it makes it easier to interpret potential results. That said, instead of having to do any manual labour, xenoseq requires only a single file that indicates where your raw reads are stored, and

which reads belong to the "reference" for that community. The reference, in most cases, is the ancestral community, meaning that all reads that are **not** found in the reference, but **are** found in a derived (evolved) community, will be identified as **unique sequences**. This is done by comparing reads to assemblies and assembling unmapped reads. Next, to distinguish between genuine transfer and false positives, xenoseq blasts all unique sequences against a local database of the other communities. If a good hit is found, it is assumed the sequence came from this community. While it is technically still possible the sequence did not transfer, we cannot increase the certainty any further without risking our ability to detect true positives.

To run xenoseq, you need to prepare a "metadata" file, which is a table that describes your samples. It should look something like this:

```
#query                reference
Community_1_T1        Ancestral_1
Community_1_T2        Ancestral_1
Community_1_T3        Ancestral_1
Community_2_T1        Ancestral_2
Community_2_T2        Ancestral_2
Community_2_T3        Ancestral_2
```

The first line is not necessary, and will be ignored as it starts with a '#'. An example of such a metadata file is given for the example workshop data too:

```
#Query                 reference
T1_C1_7amp_H_trimmed   Tminus1_C1_powersoil_trimmed
T2_C1_7amp_H_trimmed   Tminus1_C1_powersoil_trimmed
T3_C1_7amp_H_trimmed   Tminus1_C1_powersoil_trimmed
```

Which was run with the following command:

```
$ xenoseq -m xenoseq_metadata.txt -p reads -r _R*.fastq -l -t -o 08_Xenoseq
```

The -p option gives the path to the reads, and the -r option sets the affix for the reads (_R1.fastq and _R2.fastq). If you want to know more about the other options, try to look at the help page:

```
$ xenoseq -h
/  |  /  |
AT |  GC |  ______   _______    ______    _______   ______    ______
GG  \/AG/  /      \ /       \  /      \  /       | /      \  /      \
 TA  TT<  /GATACT  |TAAATGG  |/CCGTAA  |/AATAAAS/ /ATTTCT  |/ATGTTA  |
  TGAC  \ AA    GA |TG |  GG |AG |  AG |TT      \ TT    CG |GA |  AT |
 AA /AT  |GATCCCGT/ TA |  AA |CG \__AG | GGAACT  |TACGGGTA/ GT \__AT |
AA |  GC |CC       |GA |  GT |GC    GT/ /     GG/ TA       |AG    AT |
GG/   TG/  GTAGGCC/ CA/   TT/  TAAATG/  ATGCGCG/   ATGCAAT/  AGGGTTT |
                                                              AA |
                                                              AA |
                                                              AA/

------------------------------------------------------------------------
                         (XENOSEQ v1.2.3)
------------------------------------------------------------------------

xenoseq v1.2.3

contact: bramvandijk88@gmail.com

Usage:
         xenoseq -m <meta_data_tsv> -o <output_dir> -c <num_cores> -t
Mandatory:
        -m/--metadata           File containing the metadata (tsv file with query-reference sets)
Optional options:
```

```
        -p/--path_to_reads <STRING>     Path to reads for samples in metadata (default =
samples/reads)
        -r/--read_suffix <STRING>       Read suffix corresponding to metadata n ames (e.g. when
read filenames are Sample1_R1.fq and Sample1_R2.fq, use _R*.fq) (default = _R*.fq)
        -l/--link                       After detecting unique contigs, attempt to link them to
other reference samples.
        -t/--trace                      After detecting xenotypic contigs, trace them across all
samples.
        -c/--cores <INT>                Number of CPUs to use for smaller tasks (passed on to bwa,
samtools, etc.) (default = 4)
        -C/--assembly_cores <INT>               Number of CPUs to use for assembly (megahit)
(default = 4)
        -j/--jobs  <INT>                Maximum number of parallel jobs (default = 4)
        -J/--max_assembly_jobs  <INT>   Maximum number of parallel jobs for assembly (default = )
        -o/--output <STRING>            Output directory to put all the data
        -L/--alignment_length           Minimal alignment length to link unique sequences to other
reference samples.
        -S/--single_end <STRING>        Assume single-end reads (e.g. use only Sample1_R1.fq and
skip read merging)
        -P/--alignment_pid              Minimal percent identity to link unique sequences to other
reference samples.
        -f/--force_relink               Link unique sequences to reference samples, even when this
step is already performed.
```

The pipeline outputs the following files:

- unique_contigs.fasta; sequences in query not present in reference (e.g. ancestral) samples
- unique_contig_all_links_L300_P97.tbl; table linking contigs to other communities using BLAST, with options L and P in the filename
- xenotypic_contigs.fasta; the subset of unique contigs that can be linked to another reference using BLAST
- xenotypic_coverage.txt; text files to estimate the abundance of xenotypic sequences in non-ancestral samples
- source_contigs; directory with fasta files from reference samples that are themselves / are linked to MGEs

It also prints a summary at the end to give you an idea of how many unique/xenotypic sequences were identified:

```
[xenoseq_find     13-04_10:05:47] Xenoseq completed for all samples
[xenoseq_link     13-04_10:05:51] Found 18 unique contigs in Horizontal1. 16 of these are xenotypic.
[xenoseq_link     13-04_10:05:51] Found 10 unique contigs in Horizontal2. 8 of these are xenotypic.
[xenoseq_link     13-04_10:05:51] Found 17 unique contigs in Horizontal3. 17 of these are xenotypic.
[xenoseq_link     13-04_10:05:51] Found 12 unique contigs in Horizontal4. 10 of these are xenotypic.
[xenoseq_link     13-04_10:05:51] Found 0 unique contigs in Vertical1. 0 of these are xenotypic.
[xenoseq_link     13-04_10:05:51] Found 1 unique contigs in Vertical2. 0 of these are xenotypic.
[xenoseq_link     13-04_10:05:51] Found 0 unique contigs in Vertical3. 0 of these are xenotypic.
[xenoseq_link     13-04_10:05:51] Found 0 unique contigs in Vertical4. 0 of these are xenotypic.
[xenoseq          13-04_10:05:51] Xenoseq finished! :)
```

The above output is run on the test data that ships with xenoseq (which is a mock community with artificially introduced events of HGT). For this "clean" data, it is very good at detecting the horizontal gene transfer events in "horizontal" communities, and does not show signs of false positives as is indicated by the lack of xenotypic contigs in the "vertical" communities (that received no MGEs).

When run on the workshop data, the pipeline will not be able to detect any "xenotypic" contigs, because we only included a single community in the workshop data to save space. However, when run on a dataset of multiple communities, all these files will allow you to retrace exactly which sequences moved where:

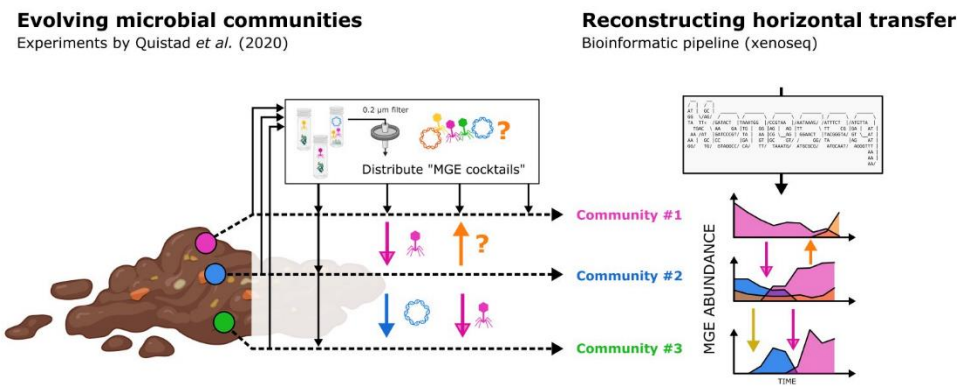


Figure 24 - Xenoseq allows capture and tracing of MGEs moving between communities. For more details see the Github page.

To nevertheless get an idea of what kind of output you could be working with, let's take a look at some of Andy Farr's data that was put into xenoseq:

```
$ cd /groups/micropop/Bram/Andy/Ampicilin_Mesocosms
```

Andy has 3 compost communities evolving in parallel, exchanging mobile elements via a "MGE cocktail" (community filtrate using a .2 micron filter). After renaming Andy's data to have appropriate labels (which I explained in an earlier section), I only needed to prepare a "metadata" file:

```
$ cat xenoseq_metafile_R2.tsv
T1_C1_0amp_H_R2 T0_C1_0amp_R2_deep
T1_C1_1amp_H_R2 T0_C1_1amp_R2_deep
T1_C1_7amp_H_R2 T0_C1_7amp_R2_deep
T1_C2_0amp_H_R2 T0_C2_0amp_R2_deep
T1_C2_1amp_H_R2 T0_C2_1amp_R2_deep
T1_C2_7amp_H_R2 T0_C2_7amp_R2_deep
T1_C3_0amp_H_R2 T0_C3_0amp_R2_deep
T1_C3_1amp_H_R2 T0_C3_1amp_R2_deep
T1_C3_7amp_H_R2 T0_C3_7amp_R2_deep
T2_C1_0amp_H_R2 T0_C1_0amp_R2_deep
T2_C1_1amp_H_R2 T0_C1_1amp_R2_deep
T2_C1_7amp_H_R2 T0_C1_7amp_R2_deep
T2_C2_0amp_H_R2 T0_C2_0amp_R2_deep
T2_C2_1amp_H_R2 T0_C2_1amp_R2_deep
T2_C2_7amp_H_R2 T0_C2_7amp_R2_deep
T2_C3_0amp_H_R2 T0_C3_0amp_R2_deep
T2_C3_1amp_H_R2 T0_C3_1amp_R2_deep
T2_C3_7amp_H_R2 T0_C3_7amp_R2_deep
T3_C1_0amp_H_R2 T0_C1_0amp_R2_deep
T3_C1_1amp_H_R2 T0_C1_1amp_R2_deep
T3_C1_7amp_H_R2 T0_C1_7amp_R2_deep
T3_C2_0amp_H_R2 T0_C2_0amp_R2_deep
T3_C2_1amp_H_R2 T0_C2_1amp_R2_deep
T3_C2_7amp_H_R2 T0_C2_7amp_R2_deep
T3_C3_0amp_H_R2 T0_C3_0amp_R2_deep
T3_C3_1amp_H_R2 T0_C3_1amp_R2_deep
T3_C3_7amp_H_R2 T0_C3_7amp_R2_deep
```

In the first column, there are three time points for each sample, and three different treatments: 0 amp, 1 amp, and 7 amp. These are different ampicillin treatments (frequency and concentration of the added antibiotic). In the second column, there are only samples from T0: the ancestral communities. These samples are sequenced extra deep. The reasoning is this: the ancestral communities will serve as a reference for newly appeared sequences in subsequent time points. To minimize the risk of false positives,

that is to say, sequenced that were not actually "new" but merely missed in earlier sampling, we sequenced this community extra deep. Nevertheless, false positives occur sometimes. This is because in the first weeks, there are a lot of shifts in the community, which cause very rare sequences to become abundant. Further improvements could be made to the experimental design by first letting the communities evolve for a month or so, and only then establishing the "ancestral" populations. But I digress.

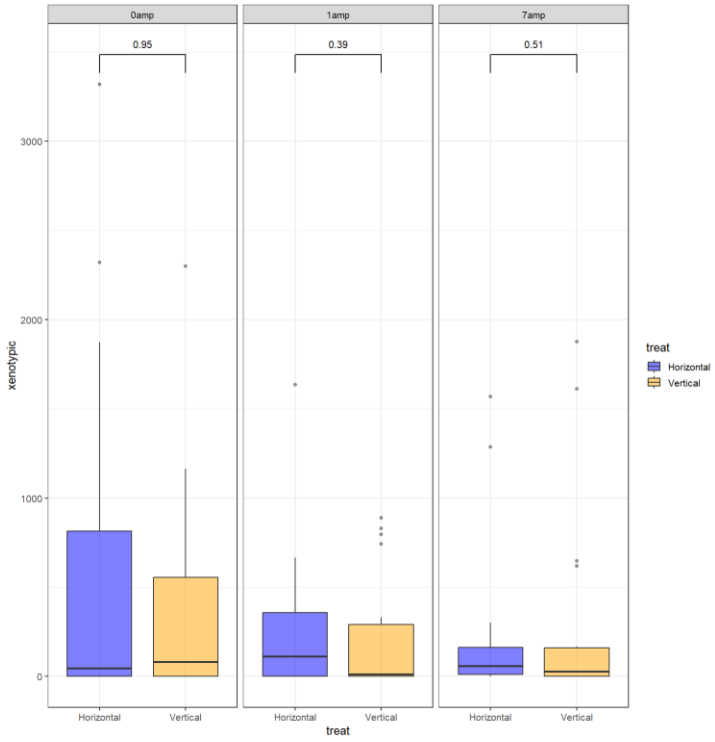
To run xenoseq on this data, we don't need to do any extra things. The metadata table (and using -p and -r to tell xenoseq where to find the reads!) is all you need to do:

```
$ xenoseq -m xenoseq_metafile.tsv -p raw_reads -r _R*.fastq -o Xenoseq_Amp_Deep
```

Xenoseq will automatically make the output (-o) directory, and many subdirectories within that directory to organise your results. I saved the output of the xenoseq program to a summary file here:

```
$ cat Xenoseq_Amp_Deep/summary.txt
```

Interestingly, summarizing this text file in a boxplot, we can see that (unlike the mock dataset above), there is no significant enrichment in xenotypic sequences in any of the horizontal communities!



However, in our preprint of the pipeline, we found a massive enrichment, so what is going on here? I have a few hypotheses:

1. Andy's treatment (ampicillin) causes such large shifts in the community, that the false positive rate due to metagenomic detection limits becomes very high
2. Andy's experiment was shorter than Steven's original experiment, and not a lot of movement occurs in complex communities in a short time window (although Yansong has verified that for a single species, it doesn't take long for it to pick something up!)
3. Our threshold for distinguishing "xenotypic" from "unique" wasn't strict enough.

From here, there are two possible future trajectories. Either we analyse the xenotypic sequences found in horizontal communities and see if they are plasmids, phages, and other MGEs encoding ampicillin resistance. Then we need to verify those same elements do not appear in the set of vertical "xenotypic" sequences. But we could also think about how to rerun the pipeline and make things better. Here are a few ways to make xenoseq more sensitive:

1. Solving problem number 1 described above instead of using a single reference sample (T0), you use T0 and all derivatives from it. This was the strategy used for Steven's work (in the pipeline paper), and it seems to work very well. So the reference for horizontal communities then becomes T0 + T1_V + T2_V, etc. This ensures that the sequence you identify has *never* been seen in any of the communities, even after massive shifts have occurred. Another way to solve problem number 1 is to allow communities to first establish for a little longer, such that the biggest community shifts are over with. If you want to be extra careful, you do both, making sure that (i) the ancestral community is slightly used to the laboratory conditions, and (ii) you sample untreated (vertical) communities to use as extra references for the pipeline.
2. Adjust the blast threshold to identify which are "true" xenotypic sequences. The analysis above was run with pretty loose blast-thresholds, which could explain why so many sequences were accepted as "xenotypic" even though they were not. I reran this part of the pipeline to see if it improves things, but there are still a lot of sequences popping up in vertical communities. So doing 2 will help, but ideally, you do it together with option 1.

To do option 1, all you need to do is merge all the samples belonging to an entire lineage of experiments into one. From the raw_read directory, one could make a nested loop to access all the appropriate files:

```
#!/bin/bash

for C in {1..3}; do
        for amp in 0 1 7; do
                for R in 1 2; do
                      echo "Merging for com ${C}, amp ${amp} and replicate ${R}";
                      ls T0_C${C}_${amp}amp_R${R}*_R1.fastq;
                      ls T*_C${C}_${amp}amp_V_R${R}*_R1.fastq;
                      # you could merge these by replacing 'ls' with 'cat' and piping it to a file
                done;
        done;
done;
```

Then, you can replace the reference column in the metadata files with these files and rerun xenoseq. (let's all wish Andy good luck :D)

Glossary

Here's a glossary of terms used in the course (and some other terms you may encounter in MGX)

Word	Description
Aligning	Finding regions of similarity between DNA or protein sequences to infer evolutionary relationships
Alpha diversity	The diversity within a sample (however measured)
Annotation	The process of identifying and characterising genes or functions in a genome or metagenome.
Assembly/Assembling	Combining reads into larger sequences based on sequence overlap (and typically, other factors than overlap are also taken into account)
BAT	Bin annotation tool
Beta diversity	The diversity of samples (however measured)
Binning	Trying to figure out which collection of contigs belongs to a single compartment (cellular or viral)
Breadth	The number of base pair covered by at least N reads
CAT	Contig annotation tool
COG	Cluster of orghologous groups. The COG protein database was generated by comparing predicted and known proteins in all completely sequenced microbial genomes to infer sets of orthologs. Each COG consists of a group of proteins found to be orthologous across at least three lineages and likely corresponds to an ancient conserved domain.
Completeness	When binning has resulted in a complete set of *marker genes*, the MAG is inferred to be complete
Completeness	After creating a MAG (e.g. by binning contigs), completeness gives the percentage of single-copy marker genes present. These marker genes are well curated, and different depending on the estimated species.
Contamination	When binning wrongly assigned contigs to belong to the same compartment, this is called 'contamination'. This can happen when two strains are closely related and equally abundant
Contamination	Like completeness, contamination is estimated using single-copy marker genes. If a single-copy marker genes occurs more than once, it can be inferred that the genome may have been contaminated with sequence information from another strain or species.
Contig	Long DNA sequences, result of assembling reads
Coverage	How reads cover each site in a genome, contig or MAG
Cross-assembly (sometimes co-assembly)	Combining reads from multiple files into a single file, and assembling the result. Can help with the assembly of rare species, but can also result in chimera genomes that do not belong together.
Dereplication	Removing MAGs/bins that are very similar to get a better representation of the available species.
DNA fragment	This word is usually reserved to refer to the DNA fragment as it was in the sample, for which the reads are only a "read-out". For example, preparing for Illumina sequencing often yields DNA fragments of 300-500 base pairs, so your 150 base pair reads do not cover the whole fragment!
Forward read	DNA fragment read from 5' to 3'.
Gene onthology (GO)	A hierarchical vocabulary used to describe the functions and properties of genes
Global alignment	Align sequences over their entire lengths. Useful when comparing two similar sequences and you want to identify single nucleotide

	polymorphisms. Will not be useful when one sequence is much shorter than the other.
Homologous gene	Genes with similar structure or function because of common descent
Local alignment	Align a query sequence to a reference. Here, we are not only interested in **if** it aligns, but also **where** the query aligns to the reference. This is what BLAST does.
MAG	After binning contigs (and if you're doing a good job, also a lot of effort to check the quality), bins are often called Metagenome Assembled Genomes (MAGs)
MGX	Acronym for metagenomics (meta-genome-X… I dunno)
N50/L50	Length of the shortest contig needed to span half your assembly (N50) and number of contigs needed (same for N90,L90). Usually, N50 refers to length, and L50 refers to the number. Bbstats reports these metrics the wrong way around, to make it even more confusing. You're welcome.
ORF	Open reading frame: A DNA sequence that can potentially encode a protein.
Ortholog	Orthologs are defined as genes in different species that have evolved through speciation events only
OTU	Operational taxonomic unit: A unit of microbial diversity used to define clusters of sequences that share a high degree of similarity
Paralog	A gene with common ancestry in the same species (*i.e.* result of gene duplication)
Rarefaction analysis	A statistical method used to estimate observed diversity based on the number of sampled individuals/reads
RAT	Read annotation tool
Read	Short (illumina) or long (nanopore) DNA sequences that are a direct representation of your sample. Note that a read may not cover the whole section of the **DNA fragment** that was sequenced.
Read depth	The average number of reads covering all sites in a larger sequence (this reduces coverage to a single number)
Read mapping	The process of using local alignments to determine not only **which**, but **where** a set of sequences (e.g. reads) aligns with a longer sequence (e.g. a reference genome).
Read pair	Most illumina samples will be paired reads, where the DNA fragments are sequenced in one direction (forwards) and in the other direction (backwards). These may overlap in the middle, but not always. In any case, it is very useful for assemblers (and ourselves) to have this extra information: these two sequences were close together on the DNA!
Reverse read	DNA fragment read from 3' to 5'. Complement may partially overlap with forward reads if the DNA fragment is < 300 base pairs.
Scaffold	When two contigs are linked by a read pair (see read pair), they are inferred to be part of the same replicon. They are therefore joined by unknown basepairs (N) in a separate file (scaffolds.fasta)
Sequence similarity	The degree of similarity between two sequences (e.g. percent identity reported by BLAST). Should not be automatically assumed to indicate homology, which is an evolutionary claim.
Sequencing depth	Not to be confused with read depth, sequencing depth represents how "well" you sequenced your community, especially relevant for rare species that you may miss if your sequencing efforts are poor
Shotgun sequencing	Sequencing a sample by fragmenting *all* the DNA into smaller pieces
Trimming	Removing adapters, low quality tails, or otherwise bad sequences from your collection of reads

Unique contig	Previously undetected contig in a sample, identified by read mapping against the ancestral community whatever other samples used as "reference".
Xenotypic contig	Previously undetected contig in a sample detected by xenoseq, which aligns well to one of the other communities put in the pipeline

Appendix I – Using in-house conda and installing Xenoseq

During the workshop, I have explained how to set up a conda environment[8]: a package manager that makes sure your programs (and how they depend on one another) don't break all the time. Before I explain how to do this, you can actually use the pre-installed conda on Wallace too:

```
$ source /data/modules/python/python-anaconda3/etc/profile.d/conda.sh
```

After which you will be able to activate conda environments, or install a new one (as some of you did for my pipeline xenoseq):

```
$ git clone https://github.com/bramvandijk88/xenoseq.git
$ cd xenoseq
$ conda env create -f environment.yml
$ conda activate xenoseq
```

Note that during the workshop, we copy-pasted these commands to the terminal. In some terminals, the '-' symbol gets transformed into a different type of dash (and the command silently fails…). So perhaps be careful and type it out for yourself.

You should now be able to run the example dataset:

```
$ ./xenoseq -m example_metadata.tsv -o Xenoseq_example -l -t            # Full full
pipeline
```

Note that if you are in a different directory, typing "xenoseq" will give you the following error:

```
$ xenoseq
bash: xenoseq: command not found
```

This is because the computer does not know where xenoseq is installed. Linux looks for installed programs in all "paths" present in the PATH variable. To inspect your current PATH variable, you can echo it:

```
$ echo $PATH
/usr/local/opt/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

This variable contains numerous paths, separated by a ":" symbol. When typing any command, this is where the computer will look for installed programs with a matching name (starting with the left-most path). To enable us to get access to xenoseq from any directory, we need to add the xenoseq install directory to this PATH variable. If you installed xenoseq in your home directory:

```
$ export PATH='$PATH:~/xenoseq'
```

If you do not want to run this every time you login to Wallace, you can add this line of code to a file in your home directory, that is sourced whenever you login to Wallace, called either ".bashrc" or ".bash_profile" (ask IT if this does not work for you, some people do not have this enabled). You can append this line of code to your source file like this:

```
$ echo export PATH='$PATH:~/xenoseq' >> ~/.bashrc
```

Now, every time you login to wallace 'xenoseq' should be found from whereever you currently are. Huzzah!

[8] We spent about an hour installing Conda ourselves during the workshop. While not necessary when working on Wallace, you may perhaps be interested in setting this up on your own computer. For this, instructions can be found here.