Buckling Collaborative Genome Assembly Pipeline

Daniel Padfield

July 6, 2025

Table of contents

Outline

This documents the steps used to assemble a high quality genome from long- and (optionally) short-read sequencing. Here, we describe the steps taken, provide links to the relevant packages, and code chunks that can be used in your own analysis. While somme sequencing services may provide you with a complete genome, there are several reasons why you might want to keep the raw reads and assemble the genome yourself:

- It is fun to learn new things!
- Can result in a standardised pipeline across sequencing service providers.
- Can do custom checks to identify contamination.
- Gives you more freedom to do custom steps during assembly.

Inspiration

This pipeline relies heavily on other resources, primarily Autocycler by Ryan Wick, and the methods of the ATCC Genome Portal managed by Jonathan Jacobs. The code in the pipeline was written with input from GitHub Copilot.

Contact

If you have any questions about this pipeline or how to implement any of it, please contact Dan Padfield at d.padfield@exeter.ac.uk.

Other resources

- Example script demonstrating how to download necessary packages for some of the pipeline.

Contribute

Please contribute to this documentation! Fixing any typos or

- Turn this script into a bash scripts that can be ran on the server. Likely just the assembly steps, and maybe **sourmash** looking at contamination.

Overview of the pipeline

Installing necessary packages

This is going to be one of the most laborious, and possibly anger inducing, steps. Bioinformatics software can be notoriously difficult to install. However, there is a code chunk that can helpfully help with this. It first installs miniforge that installs **conda** and **mamba** package managers. It then sets a one time configuration for which repositories to use when installing packages, prioritising **conda-forge** and **bioconda** based on current recommendations.

Firstly we install **tmux** which allows us to run multiple terminal sessions at once, and is particularly useful for running long-running commands. A cheatsheet for tmux is here. The think I found most awkward was the `Ctrl-b`, then `d` for detach, but is easier when you realise `Ctrl-b` is the prefix action to all **tmux** commands.

We then create a new environment (which is a bit like a virtual machine) for all of the tools in the pipeline. This is useful to avoid conflicts between different projects. You can learn more about conda environments here.

If there are conflicts between any of the packages in your install, it might be useful for you to create different environments for some of the different packages. This can be done easily by copying the line of code and editing it, an example is shown below.

This is all currently run on the RStudio Servers, which run Linux. If you are running this on a Mac or Windows machine, you may need to adapt the code slightly. The code is run in the Terminal using bash, I SSH into the RStudio Server using Positron, the new IDE from Posit (formerly RStudio), that works much like VSCode.

```
# install new mamba/conda installation if needed
curl -L -O
 ↪  "https://github.com/conda-forge/miniforge/releases/latest/download/M
 ↪  -m).sh"

bash Miniforge3-$(uname)-$(uname -m).sh

# update conda if needed
conda update conda

# set up channel configuration based on current best
 ↪  practice
conda config --add channels bioconda
conda config --add channels conda-forge
conda config --set channel_priority strict

# install tmux
conda install -c conda-forge tmux
```

```
# install all the tools used in the pipeline - ADD OR REMOVE
 ↪  AS NEEDED
conda create -n autocycler -c conda-forge -c bioconda -c
 ↪  defaults autocycler canu flye metamdbg miniasm necat
 ↪  raven-assembler plassembler nextdenovo nextpolish fastp
 ↪  filtlong sourmash seqkit dnappler

# we also need to install a database for plassembler and
 ↪  make it available to the conda environment
conda activate autocycler
plassembler download -d ~/databases/plassembler
conda env config vars set
 ↪  PLASSEMBLER_DB=~/databases/plassembler
conda deactivate
```

Check for contamination

This step is highly recommended, especially when working with environmental isolates! It is common that the

We want a quick way to check for contamination of the short and long reads. This can be done using

Filtering short reads

It is super important to filter your short reads before assembly. Typically sequencing services do some baseline filtering, but we may want to do more. At ATCC, Illumina reads must pass the following quality control:

- Median Q score, all bases > 30
- Median Q score, per base > 25
- Ambiguous content (%N bases) $< 5\%$

This is quite stringent, we will implement a slightly less stringent filter:

- Median Q score, all bases > 30
- Expected read length > 95% of expected read length (e.g. for 2x 150bp reads, this would be 143bp, for 2x 300bp reads, this would be 285bp)

This section assumes you have a folder with your short reads in fastq.gz format. If you have paired-end reads (e.g. forward and reverse) you will need to know how they are named. In this example from MicrobesNG, the forward reads end in **1_trimmed.fastq.gz** and the reverse reads end in **2_trimmed.fastq.gz**. You can change this to match the naming of your files.

TIP. When running for loops in Terminal, I will often run a version to check the right filenames are being called before running the filtering step. To do this you can un-hashtag echo $fwd and echo $rev and hash the fastp command.

```
# make directory for short trimmed short reads
mkdir -p short_reads/trimmed
trimmed_short=short_reads/trimmed

conda activate autocycler

# run fastp on all the short reads
for file in short_reads/*1_trimmed.fastq.gz; do

    fwd=$file
    # replace 1_trimmed.fastq.gz with 2_trimmed.fastq.gz to
     ↪  get the reverse read
    rev=${fwd%1_trimmed.fastq.gz}2_trimmed.fastq.gz

    #echo $fwd
    #echo $rev
```

```
    # run fastp on the file
    fastp -i $fwd -I $rev -o  "$trimmed_short/$(basename
 ↪  $fwd)" -O "$trimmed_short/trimmed/$(basename $rev)" -w 4
 ↪  --detect_adapter_for_pe -l 237 -q 30 -j
 ↪  "$short_reads/fastp_reports/$(basename
 ↪  ${fwd%1_trimmed.fastq.gz}fastp.json)" -h
 ↪  "$short_reads/fastp_reports/$(basename
 ↪  ${fwd%1_trimmed.fastq.gz}fastp.html)"
done
```

Filtering long reads

We try keep as many long reads as possible. Having too stringent a filter on minimum length can result in losing plasmid sequences. For example, if you have small plasmids of ~2000bp, removing any sequences smaller than 1000bp will result in losing these plasmids. Consequently our filters are:

- Minimum read length > 1000bp
- Minimum mean read quality > 10

```
# make directory
mkdir -p long_reads/trimmed

# run for loop
for file in long_reads/*.fastq.gz; do
    # run filtlong on the file
    filtlong --min_length 1000 --min_mean_q 10 "$file" |gzip
 ↪  > "long_reads/trimmed/$(basename $file)"
done
```

Checking estimated coverage

At this stage you need to pass an estimated **genome size** to the command. Methods to do this can be found here.

Subsampling long reads

We can now start the fun bits with Autocycler! The first step is to subsample the long reads. **autocycler subsample** creates multiple read subsets from a single long-read dataset, minimising overlap to ensure each subset is as independent as possible.

Each subset is then used to generate a separate assembly. Creating diverse input assemblies increases the likelihood of capturing all sequences in the genome while reducing the risk of shared assembly errors.

The below command takes all the files in **long_reads/trimmed** and creates a **subsampled_reads** directory which will contain the subsampled read files. Each file with have its own sub-folder within **subsampled_reads**.

```
# subset reads
for file in long_reads/trimmed/*.fastq.gz; do
    autocycler subsample --reads $file --out_dir
 ↪  long_reads/subsampled/$(basename ${file%.fastq.gz})
 ↪  --genome_size 4.6M
done
```

Creating multiple assemblies

We will now use **autocycler helper** to produce multiple alternative assemblies of the same genome. It is beneficial to use different assemblers,

as different tools can make different errors. **autocycler helper** acts as a wrapper to run multiple assemblers.

This command will create an assembly using **canu**, **flye**, **metamdbg**, **miniasm**, **necat**, **nextdenovo**, and **raven**. **canu** takes the longest to run, but can often by very accurate.

The following code runs autocycler on all of the subsampled long reads. It will create a folder called **assemblies** and put the assemblies in there. The assemblies will be named with the original long read file name, assembler name, and the sample name. This code could likely be sped up or parallelised, but for now it is run sequentially.

```
# set threads to use for assembly
threads=15

# how many threads do we have
nproc

genome_size=4.6M

file="long_reads/subsampled/307501E/sample_01.fastq"
filename=$(basename ${file%.fastq})

for folder in long_reads/subsampled/*;do
    foldername=$(basename "$folder")

    for file in "$folder"/*.fastq; do
        filename=$(basename ${file%.fastq})

        # run each assembler
        for assembler in canu flye metamdbg miniasm necat
         ↪ nextdenovo raven; do
            autocycler helper "$assembler" --reads "$file"
 ↪ --out_prefix
 ↪ assemblies/"$foldername"_"$assembler"_"$filename"
 ↪ --threads "$threads" --genome_size "$genome_size"
```

```
            conda deactivate
        done
    done
done
```

If you think you might have plasmids, you can also run **plassembler** to assemble plasmids. Assemblies made by **plassembler** will only include plasmids.

```
for folder in long_reads/subsampled/*;do
    foldername=$(basename "$folder")

    for file in "$folder"/*.fastq; do
        filename=$(basename ${file%.fastq})
        # run plassembler
        autocycler helper plassembler --reads "$file"
 ↪  --out_prefix
 ↪  assemblies/"$foldername"_plassembler_"$filename"
 ↪  --threads "$threads" --genome_size "$genome_size"
    done
done
```

Manual checking of assemblies

Before proceeding with Autocycler to make a consensus assembly, inspect each input assembly to identify and remove incomplete or problematic assemblies. This step helps ensure the rest of Autocycler's pipeline proceeds smoothly.

Things to check are:

- Chromosome length: For most bacteria, a single chromosome constitutes the majority of the genome. If an assembly lacks a contig

of the expected chromosomal length, it is likely fragmented. For example, an E. coli chromosome is ~5 Mbp in length, so a complete E. coli assembly should contain a ~5 Mbp contig. If an E. coli assembly instead contains a 3 Mbp and 2 Mbp contig, it is almost certainly incomplete.
- Visual inspection of the GFA output (if available).
- What to remove. You can remove sequences from each assembly that are not close to the expected genome size, by +/-20%. For example, if you expect a genome size of 4 Mbp, you can remove any sequences that are not between 3.2 Mbp and 4.8 Mbp in length. If expecting a plasmid, you can create new fasta files that include those.

To view assemblies that have GFA output, you can use **bandage** to visualise the assemblies. To view fasta files, you can use **AliView**.

This next code in R creates a dataset summarising the sequences in each assembly.

```
# install librarian package if not already installed
if (!requireNamespace("librarian", quietly = TRUE)) {
    install.packages("librarian")
}

# load and install tidyverse and Biostrings
librarian::shelf(tidyverse, Biostrings)

# list all assembly files, needs to be in fasta format
assembly_files <- list.files("assemblies", pattern =
 ↪  ".fasta", full.names = TRUE)

# function to get key information
get_assembly_info <- function(file){
    # read the fasta file
    seqs <- Biostrings::readDNAStringSet(file)
```

```
    num_sequences <- length(seqs)

    # keep the longest sequence only
    seqs <- seqs[which.max(nchar(seqs))]

    length_seq <- nchar(seqs)

    # longest sequence is circular
    is_circular <- ifelse(grepl("circular=yes",
 ↪  tolower(names(seqs))), 'yes', 'no')

    file <- basename(file) %>% tools::file_path_sans_ext()

    # return a data frame with the information
    data.frame(
        filename = file,
        n_seqs = num_sequences,
        length_longest_seq = length_seq,
        is_circular = is_circular
    )
}

info <- assembly_files %>% map_df(get_assembly_info)

# separate the filename
info <- separate_wider_delim(info, filename, names =
 ↪  c("sample", "assembler", 'subsample'), delim = "_",
 ↪  too_many = "merge", cols_remove = FALSE)

head(info)
```

sample	assembler	subsample	filename	n_seqs	length_longest	circular
307501E	canu	sample_01	307501E_canu_sample_01	2	4638966	yes
307501E	canu	sample_02	307501E_canu_sample_02	2	4638967	yes
307501E	canu	sample_03	307501E_canu_sample_03	1	4711370	no
307501E	canu	sample_04	307501E_canu_sample_04	1	4638967	yes
307501E	flye	sample_01	307501E_flye_sample_01	1	4638969	no
307501E	flye	sample_02	307501E_flye_sample_02	1	4638955	no

To filter fasta files, you can use **seqkit**. The following code will filter the assemblies based on the expected genome size. It will create a new folder called **filtered_assemblies** and put the filtered assemblies in there. It will keep all sequences that are greater than 80% of the expected genome size. If the filtered assembly is empty, it will be removed.

```
# make directory for filtered assemblies
mkdir -p assemblies/filtered_assemblies

# calculate 80% of the expected genome size
to_cut=$(echo "$genome_size * 0.8 * 1000000" | bc | cut
 ↪  -d'.' -f1)

# run for loop to filter assemblies
for file in assemblies/*.fasta; do
    # get the filename without the extension
    filename=$(basename ${file%.fasta})

    # filter the fasta file based on expected genome size
    seqkit seq -g -m "$to_cut" "$file" >
 ↪  "assemblies/filtered_assemblies/${filename}_filtered.fasta"

    # if the filtered file is empty, remove it
    if [ ! -s
     ↪  "assemblies/filtered_assemblies/${filename}_filtered.fasta"
     ↪  ]; then
```

```
        rm -f
 ↪  "assemblies/filtered_assemblies/${filename}_filtered.fasta"
    fi
done
```

After filtering, we can re run the R code to check the sequences in the filtered assemblies.

```
# list all assembly files, needs to be in fasta format
assembly_files <-
 ↪  list.files("assemblies/filtered_assemblies", pattern =
 ↪  ".fasta", full.names = TRUE)

info <- assembly_files %>% map_df(get_assembly_info)

# separate the filename
info <- separate_wider_delim(info, filename, names =
 ↪  c("sample", "assembler", 'subsample'), delim = "_",
 ↪  too_many = "merge", cols_remove = FALSE)

head(info)
```

sample	assembler	subsample	filename	n_seqs	length_longest_scaf	circular
307501E	canu	sample_01	307501E_filtered_canu_...	1	4638966	yes
307501E	canu	sample_02	307501E_filtered_canu_...	1	4638967	yes
307501E	canu	sample_03	307501E_filtered_canu_...	1	4711070	no
307501E	canu	sample_04	307501E_filtered_canu_...	1	4638967	yes
307501E	flye	sample_01	307501E_filtered_flye_...	1	4638969	no
307501E	flye	sample_02	307501E_filtered_flye_...	1	4638955	no

From this filtering process, you may lose some samples that do not pass this QC, but this is fine as long as you still have several assemblies to work with.

If you expect plasmids, you may want to adjust these criteria. In our case we have plasmids of interest (R1 and RP4) that are between 50,000 and 100,000 bp in length. We can filter the assemblies to keep only those that are within this range. This can be done using **seqkit** again, but this time we will use the `-m` and `-M` options to specify the minimum and maximum length of the sequences to keep.

```
# our plasmids are between 50000 and 100000bp

# run for loop to filter assemblies
for file in assemblies/*.fasta; do
    # get the filename without the extension
    filename=$(basename ${file%.fasta})

    # filter the fasta file based on expected genome size
    seqkit seq -g -m 50000 -M 110000 "$file" >
 ↪  "assemblies/filtered_assemblies/${filename}_filtered.fasta"

    # if the filtered file is empty, remove it
    if [ ! -s
     ↪  "assemblies/filtered_assemblies/${filename}_filtered_plasmid.fas
     ↪  ]; then
        rm -f
 ↪  "assemblies/filtered_assemblies/${filename}_filtered_plasmid.fasta"
    fi
done
```

We are now ready to proceed to the next steps of the pipeline, actually using the **autocycler** commands. At this stage you may want to manually separate your assemblies into individual folders. This should all be much less computationally intensive than the previous steps, so you can probably easily run these on your local machine.

Finding a consensus assembly

We follow the steps as documented by **Autocycler**. The Wiki pages are extremely useful and I am mainly summarising content from there. They contain lots of FAQs and troubleshooting tips if you come into trouble.

First**autocycler compress** uses the input assemblies to build a compacted De Bruijn graph (a.k.a. a unitig graph). This command takes an input directory named **filtered/assemblies** which contains the input assemblies in FASTA format. It will create the **debruijn** directory which will contain the graph as file named **input_assemblies.gfa**.

```
# run autocycler compress to create a de bruijn graph of all
 ↪ the assemblies
autocycler compress -i assemblies/filtered_assemblies -a
 ↪ assemblies/filtered_assemblies/debruijn
```

Next, **autocycle cluster** groups input contigs into clusters. A cluster is a group of contigs which represent the same genomic sequence. It also decides which of these clusters should be included in the final assembly (QC-pass) and which should not (QC-fail). This command takes a directory created by **autocycler compress** (named **debruijn** in the example) which must contain an **input_assemblies.gfa** file. It will create a subdirectory named **clustering** which in turn contains a **qc_pass** directory for the good clusters and a **qc_fail** directory for the bad clusters.

```
# run autocycler cluster to see which bits of the assembly
 ↪ should be included in the final assembly
autocycler cluster -a
 ↪ assemblies/filtered_assemblies/debruijn
```

autocycler trim then trims excess sequences that can be present in long read contigs. This command goes through each cluster and trims each contig looking for hairpin and start-end circular overlaps. At the end of

this step, remaining contigs in each cluster are in close agreement to each other.

```
# run autocycler trim to get rid of excess repetitive
 ↪ sequences
for c in
 ↪ assemblies/filtered_assemblies/debruijn/clustering/qc_pass/cluster_*
 ↪ do
    autocycler trim -c "$c"
done
```

autocyler resolve aims to resolve repeats and ambiguities in each cluster. By the end of it, hopefully there is a single consentig for each cluster. This command runs through each cluster in **qc_pass**.

```
# run autocycler resolve to resolve repeats and ambiguities
 ↪ in each cluster
for c in
 ↪ assemblies/filtered_assemblies/debruijn/clustering/qc_pass/cluster_*
 ↪ do
    autocycler resolve -c "$c"
done
```

Finally, the resolved sequences for each cluster are combined into a single assembly. This command takes an autocycler directory (**debuijn** in this example) and one or more cluster graphs (typically **5_final.gfa**) and creates **consensus_assembly.fasta** and **consensus_assembly.gfa** files in the **debuijn** directory.

```
# combine the resolved sequences into a final assembly
autocycler combine -a
 ↪ assemblies/filtered_assemblies/RP4/debruijn -i
 ↪ assemblies/filtered_assemblies/RP4/debruijn/clustering/qc_pass/clust
```

If this has worked, you will hopefully receive the message.

```
Consensus assembly is fully resolved
```

You can check the output of **autocycler combine** by looking at the **consensus_assembly.gfa** in Bandage. You can also look at the **consensus_assembly.yaml** file which contains the following information:

- consensus_assembly_bases: 4638969
- consensus_assembly_unitigs: 1
- consensus_assembly_fully_resolved: true
- consensus_assembly_clusters:
  - length: 4638969
  - unitigs: 1
  - topology: circular

It is important to have a single unitig, for the assembly to be fully resolved, and you want the topology to be circular. If this is not the case, you can try running **autocycler clean**.

Reorient assembly

Dnaapler is a tool for reorienting complete microbial genome assemblies so that each sequence starts at a consistent location – commonly at a gene like dnaA or repA. You can run Dnaapler on an Autocycler consensus assembly to reorient its sequences which can make downstream steps like annotation or submission easier, and allow easier comparison between strains.

The command **dnaapler all** works for prokaryote genomes, and **dnaapler plasmid** for plasmids. All of the options are the same and it is important to use the **.gfa**.

```
# run dnaapler to reorient the assembly
dnaapler all -i
 ↪  assemblies/filtered_assemblies/ecoli/debruijn/consensus_assembly.gfa
 ↪  -o
 ↪  assemblies/filtered_assemblies/ecoli/debruijn/dnaapler
 ↪  -p ecoli_mg1655 -t 8
```

Polishing the assembly

Since Autocycler assemblies are long-read-only, they may still contain small-scale errors produced by systematic errors in the reads.

If you are assembling Oxford Nanopore reads, performing long-read polishing with Medaka can help. For PacBio reads you could use Racon

If you also have short reads, there are many tools. Ryan Wick recommends Polypolish and Pypolca, both of which are conservative (unlikely to introduce new errors).

A paper benchmarking different polishing tools can be found here.

This command runs **medaka** and then **Polypolish** on the consensus assembly.

Check assembly using shortreads

Annotate assembly

Check quality