Contents

Hello there 😄 Welcome to this Jupyter Book instance to document our data processing pipeline to analyze data from the technique : **Sa**turated **T**ransposition **A**nalysis in **Y**east.

Status of the project

- Beta release on Zenodo

Next steps

Publication in JOSS ✍️
Make the software package ready to be published in Journal of Open Source Software

Reproducible data analysis 🚀
Reproducible data analysis through a configuration file. • e.g. Make a manifest (docker compose)

Multiple files analysis 🎁
Allow mulitple files to be processed at once using a for-loop in the bash script. Currently only one dataset at the time can be processed, but this can be extended to allow a series of datasets to be processed in a for-loop.

Python package 🔪
Build a python package that is pip installable for the ease of use for other researchers.

> 💡 **Learn more and get involved**
>
> 💡 **Open an issue**
> We track enhancement requests, bug-reports, and to-do items via GitHub issues.
>
> 🙌 **Send a pull request**
> We appreciate that contributors send a pull request to improve the code base or introduce a new feature.

What do we have currently?

- DOI 10.5281/zenodo.4636310
- Download here the Public release v1.0.beta-1

Software License

- License Apache 2.0
- This work is licensed under Apache 2.0 . The 2.0 version of the Apache License, approved by the ASF in 2004, helps us achieve our goal of providing reliable and long-lived software products through collaborative open source software development.

Current Contact

- Leila Inigo de la Cruz
  - email: L.M.InigoDeLaCruz@tudelft.nl

Acknowledgements 💙

- Many thanks to the group of Prof. Benoit Kornmann , in particular to Prof. Agnes Michel , for all the nice discussions and colaboration with our team. Also thanks to them due to the SATAY forum they keep on running for all the community of satayers 🎉 .
- Many thanks to the DCC department at TU Delft, specially to Maurits Kok who in his role of Research Software Engineer has supported us in developing the software package to be distributed to the scientific community.

Overview of the pipeline functions

Step 1 data analysis; From raw data to essential protein overview

SATAY experiments need to be sequenced which results in a FASTQ file. The sequence reads from this file needs to be aligned to create a SAM file (and/or the compressed {bash}ary equivalent BAM file). Using the BAM file, the number of transposons can be determined for each insertion location.

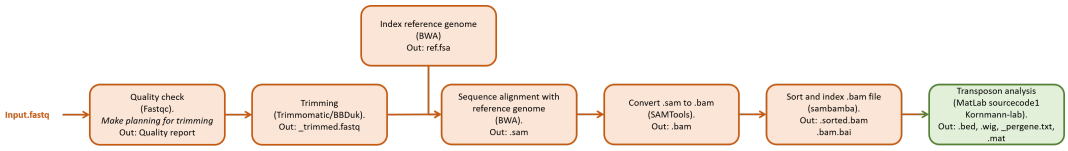
Raw data (.FASTQ file) discussed in the paper of Michel et.al. 2017 can be found at [https://www.ebi.ac.uk/arrayexpress/experiments/E-MTAB-4885/samples/].

Workflow

The results from the sequencing is typically represented in FASTA or FASTQ format. This needs to be aligned according to a reference sequence to create a SAM and BAM file. Before alignment, the data needs to be checked for quality and possibly trimmed to remove unwanted and unnecessary sequences. When the location of the reads relative to a reference sequence are known, the insertion sites of the transposons can be determined. With this, a visualization can be made that shows the number of transposon insertions per gene.

In the description given in this document, it is chosen to do the quality checking and the trimming in windows and the alignment of the reads with a reference genome in a virtual machine running Linux. It is possible to do the quality checking and trimming in Linux as well or to do the whole process in windows. To do quality checking and/or trimming in Linux requires more memory of the Linux machine since both the raw sequencing data needs to be stored and the trimming needs to be stored (both which can be relatively large files). Since a virtual machine is used, both the computation power and the amount of storage is limited, and therefore it chosen to do the trimming on the main windows computer (this problem would not exists if a computer is used running on Linux, for example a computer with an alternative startup disc running Linux). Sequence alignment can be done on Windows machines (e.g. using BBmap), but this is not ideal as many software tools are designed for Unix based computer systems (i.e. Mac or Linux). Also, many tools related to sequence alignment (e.g. converting .sam files to .bam and sorting and indexing the bam files) are done with tools not designed to be used in windows, hence this is performed in Linux.

An overview of the different processing steps are shown in the figure below.



A short overview is given for different software tools that can be used for processing and analyzing the data. Next, a step-by-step tutorial is given as an example how to process the data. Most of this is done using command line based tools.

An overview of the different steps including some software that can handle this is shown here:

1. Checking the raw FASTA or FASTQ data can be done using the (**FASTQC**) software (Windows, Linux, Mac. Requires Java). This gives a quality report (see accompanying tutorial) for the sequence reads and can be run non-interactively using the command line.
2. Based on the quality report, the data needs to be trimmed to remove any unwanted sequences. This can be done with for example (**FASTX**) (Linux, Mac) or (**Trimmomatic**) (Windows,requires Java). An easy graphical user interface that com{bash}es the FASTQC and Trimmomatic is (**123FASTQ**). Also **BBDuk** can be used for trimming (which belongs to BBMap).

3. The trimmed sequence reads need to be aligned using a reference sequence, for example the *S. Cerevisiae* S288C Ref64-2-1 reference sequence or the W303 reference sequence from SGD. Aligning can be done, for example, using (**SnapGene**) (Windows, Linux, Mac. This does not import large files and is therefore not suitable for whole genome sequencing), (**BBMap**) (Linux, Mac, Windows (seems to give problems when installing on windows machines), might be possible to integrate it in Python, (**BWA**) (Linux, Mac), (**Bowtie2**) (Linux, Mac) or (**ClustalOmega**) (Windows, Linux, Mac). This step might require defining scores for matches, mismatches, gaps and insertions of nucleotides.
4. After aligning, the data needs to be converted to SAM and BAM formats for easier processing hereafter. This requires (**SAMtools**) (Linux, Mac) or (**GATK**) (Linux, Mac). Conversion from SAM to BAM can also be done in Matlab if preferred using the 'BioMap' function.
5. Using the BAM file with the aligned sequence reads, the transposon insertion sites can be determined using the **Matlab** script given by Benoit Kornmann Lab (including the name.mat and yeastGFF.mat files). The results from this step are three files (a .txt file, a .bed file and a .wig file) that can be used for visualization.
6. If more processing is required, (**Picard**) (Linux, Mac) might be useful, as well as (**GATK**) (Linux, Mac). Visualization of the genomic dataset can be done using (**IGV**) (Windows, Linux, Mac) or SAMtools' tview function. Also (**sambamba**) (Linux, Mac) can be used, especially for sorting and indexing the bam files.
7. Creating transposon insertion maps for the genome (see the satay users website) and comparison essential genes between different genetic backgrounds using Venn diagrams, customized software needs to be created.

Quality checking of the sequencing reads; FASTQC (0.11.9)

FASTQC creates a report for the quality of sequencing data. The input should be a fastq (both zipped and unzipped), sam or bam file (it can handle multiple files at once). The program does not need to be installed, but after downloading only requires to be unzipped. FASTQC can be ran as an interactive program (i.e. using a GUI) or non-interactively using the command line options.

If using interactively, open the 'run_fastqc.bat' file in the FASTQC folder and load a file to be checked. Alternatively using the 123FASTQ (version 1.1) program, open this and use the 'Quality Check' menu on the right. The advantage of using 123FASTQ is that it can also do trimming (using Trimmomatic).

If using the command line for checking a single file use the command:

(Note that the output directory should already exist, as the program does not create directories). In the output directory, a .html file and a (zipped) folder is created, both with the same name as the input file. The .html file can be used to quickly see the graphs using a browser. Also, a zipped folder is created where the raw data of the quality check is stored. For explanation about the different graphs, see the fastqc_manual pdf or [https://www.bioinformatics.babraham.ac.uk/projects/fastqc/Help/3 Analysis Modules/] (or the paper 'Preprocessing and Quality Control for Whole-Genome' from Wright et.al. or the 'Assessing Read Quality' workshop from the Datacarpentry Genomics workshop).

For more commands, type `fastqc --help`. Some useful commands might be:

- –contaminants Reads a file where sequences are stored of (potential) contaminants. The .txt-file should be created before running the software. Each contaminant is presented on a different line the text file and should have the form name 'tab' sequence.
- –adapters Similar as the contaminants command, but specifically for adapter sequences. Also here a text file should be created before running and this file should have the same layout as the contaminants file.
- –min_length where a minimal sequence length can be set, so that the statistics can be better compared between different reads of different length (which for example might occur after trimming).
- –threads Preferably leave this unchanged, especially when an error is shown that there 'could not reserve enough space for object heap' after setting this command.
- –extract Set this command (without following of parameters) to extract the zipped folder from the results.

The output of the FASTQC program is:

- **Per base sequence quality**: Box and whisker plot for the quality of a basepair position in all reads. The quality should be above approximately 30 for most reads, but the quality typically drops near the end of the sequences. If the ends of the reads are really bad, consider trimming those in the next step.
- **Per tile sequence quality**: (Shows only when Illumina Library which retains there sequence identifier). Shows a heat map of the quality per tile of the sequence machines. Blueish colours indicate that the quality score is about or better than average and reddish colours indicates scores worse than average.
- **Per sequence quality score**: Shows an accumulative distribution to indicate which mean quality score per sequence occurs most often.

- **Per base sequence content**: Shows the percentage of nucleotide appearance in all sequences. Assuming a perfectly random distribution for all four nucleotides, each nucleotide should be present about 25% over the entire sequence. In the beginning this might be a bit off due to for example adapters present in the sequences. If this is present, it might difficult/impossible to cut these during the trimming part, but it should typically not seriously affect further analysis.
- **Per sequence GC content**: Indicates the distribution of the G-C nucleotides appearances in the genome. The ideal distribution that is expected based on the data is shown as a blue curve. The red curve should, ideally follow the blue curve. If the red curve is more or less a normal distribution, but shifted from the blue curve, is might indicate a systematic bias which might be caused by an inaccurate estimation of the GC content in the blue curve. This does not necessarily indicate bad data. When the red curve is not normal or show irregularities (peaks or flat parts), this might indicate contaminants in the sample or overrepresented sequences.
- **Per base N content**: Counts the number of N appearances in the data for each basepair position of all reads. Every time a nucleotide cannot be accurately determine during sequencing, it is flagged with a N (No hit) in the sequence instead of one of the nucleotides. Ideally this should never occur in the data and this graph should be a flat line at zero over the entire length. Although at the end of the sequences it might occur few times, but it should not occur more than a few percent.
- **Sequence length distribution**: Shows the length of all sequences. Ideally all reads should have the same length, but this might change, for example, after trimming.
- **Sequence duplication level**: Indicates how often some sequences appear the data. Ideally, all reads occur only few times and a high peak is expected near 1. If peaks are observed at higher numbers, this might indicate enrichment bias during the sequencing preparation (e.g. over amplification during PCR). Only the first 100000 sequences are considered and when the length of the reads is over 75bp, the reads are cut down to pieces of 50bp. Some duplication might not be bad and therefore a warning or error here does not need to concern.
- **Overrepresented sequences**: List of sequences that appear in more 0.1% of the total (this is only considered for the first 100000 sequences and reads over 75bp are truncated to 50bp pieces). The program gives a warning (when sequences are found to be present between 0.1% and 1% of the total amount of sequences) or an error (when there are sequences occurring more 1% of all sequences), but this does not always mean that the data is bad and might be ignored. For Illumina sequencing for satay experiments, the sequences often start with either ‘CATG’ or ‘GATC’ which are the recognition sites for NlaIII and DpnII respectively.
- **Adapter content**: Shows an accumulative percentage plot of repeated sequences with a positional bias appearing in the data. So if many reads have the same sequence at (or near) the same location, then this might trigger a warning in this section. Ideally this is a flat line at zero (meaning that there are no repeated sequences present in the data). If this is not a flat line at zero, it might be necessary to cut the reported sequences during the trimming step. If this section gives a warning, a list is shown with all the repeated sequences including some statistics. It can be useful to delete these sequences in the trimming step.
- **Kmer content**: indicates sequences with a position bias that are ofen repeated. If a specific sequence occurs at the same location (i.e. basepair number) in many reads, then this module will show which sequence at which location turns up frequently. Note that in later editions (0.11.6 and up) this module is by default turned off. If you want to turn this module on again, go to the Configuration folder in the Fastqc folder and edit the limits.txt file in the line where it says ‘kmer ignore 1’ and change the 1 in a 0.

Trimming of the sequencing reads

Next is the trimming of the sequencing reads to cut out, for example, repeated (adapter) sequences and low quality reads. There are two software tools adviced, Trimmomatic and BBDuk. Trimmomatic is relative simple to use and can be used interactively together with FASTQC. However, the options can be limiting if you want more control over the trimming protocol. An alternative is BBDuk, which is part of the BBMap software package. This allows for more options, but can therefore also be more confusing to use initially. Both software packages are explained below, but only one needs to be used. Currently, it is advised to use BBDuk (see section 2b).

For a discussion about trimming, see for example the discussion in MacManes et.al. 2014, Del Fabbro et.al. 2013 or Delhomme et. al. 2014 or at basepairtech.com (although this discussion is on RNA, similar arguments hold for DNA sequence analysis).

Trimming of the sequencing reads; Trimmomatic (0.39)

Trimmomatic alters the sequencing result by trimming the reads from unwanted sequences, as is specified by the user. The program does not need to be installed, but after downloading only requires to be unzipped. Trimmomatic can be ran as an interactive program (for this 123FASTQ needs to be used) or non-interactively using the command line options.

If using interactively, use 123FASTQ (version 1.1) and run the ‘Runner.bat’ file in the 123FASTQ folder. Use the ‘Trimmer’ in the ‘Trim Factory’ menu on the right.

If using non-interactively in the command line use the command:

Before running Trimmomatic, a .fasta file needs to be created that allows clipping unwanted sequences in the reads. For example, the 'overrepresented sequences' as found by Fastqc can be clipped by adding the sequences to the .fasta file. Easiest is to copy an existing .fasta file that comes with Trimmomatic and adding extra sequences that needs to be clipped. For MiSeq sequencing, it is advised to use the TruSeq3 adapter file that needs to be copied to the data folder (see below for detailed explanation). For this use the command:

Open the .fa file and copy any sequences in the file using a similar style as the sequences that are already present. Typically it is useful to clip overrepresented sequences that start with 'CATG' or 'GATC' which are the recognition sites for NlaIII and DpnII respectively. Note that the trimming is performed in the order in which the steps are given as input. Typically the adapter clipping is performed as one of the first steps and removing short sequences as one of the final steps.

A typical command for trimmomatic looks like this:

Check the quality of the trimmed sequence using the command:

The following can be set to be set by typing the following fields after the above command (the fields must be in the given order, the optional fields can be ignored if not needed, see also http://www.usadellab.org/cms/?page=trimmomatic):

- `SE` (Single End) or `PE` (Paired End) [required];
- `-phred33` or `-phred64` sets which quality coding is used, if not specified the program tries to determine this itself which might be less accurate. usually the phred33 coding is used. If not sure, check if the .fastq file contains, for example, an exclamation mark (!), a dollar sign ($), an ampersand (&) or any number (0-9) since these symbols only occur in the phredd33 coding and not in the phred64 coding [optional];
- `Input filename`. Both forward and reverse for paired end in case of PE [required];
- `Output filename`. Both paired and unpaired forward and paired and unpaired reverse for paired end (thus 4 output in total) in case of PE. In case of SE, a single output file needs to be specified. Needs to have the same extension as the input file (e.g. .fastq) [required];
- `ILLUMINACLIP:TruSeq3-SE.fa:2:15` or `ILLUMINACLIP:TruSeq3-PE.fa:2:30:10` (for Single End reads or Paired End reads respectively). This cuts the adapter and other Illumina specific sequences from the reads. The first parameter after `:` indicates a FASTA file (this should be located in the same folder as the sequencing data). The second paramter indicates the Seed Mismatches which indicates the maximum mismatch count that still allows for a full match to be performed. The third parameter (for SE, fourth parameter for PE) is the Simple Clip Threshold which specifies how accurate the match between the adapter and the read. The third parameter for PE sets the Palindrome Clip Threshold specifies how accurate the match between the two 'adapter ligated' reads must be for PE palindrome read alignment.

A number of adapters are stored in the 'adapters' folder at the location where the trimmomatic program is saved. In case of MiSeq sequencing, the TruSeq3 adapter file is advised. The way the adapter sequences are aligned is by cutting the adapters (in the FASTA file) into 16bp pieces (called seeds) and these seeds are aligned to the reads. If there is a match, the entire alignment between the read and the complete adapter sequence is given a score. A perfect match gets a score of 0.6. Each mismatching base reduces the score by Q/10. When the score exceeds a threshold, the adapter is clipped from the read. The first number in the parameter gives the maximal number of mismatches allowed in the seeds (typically 2). The second value is the minimal score before the adapter is clipped (typically between 7 (requires $\frac{7}{0.6}=12$) perfect matches) and 15 (requires $\frac{15}{0.6}=25$) perfect matches)). High values for short reads (so many perfect matches are needed) allows for clipping adapters, but not for adapter contaminations. Note a bug in the software is that the FASTA file with the adapters need to be located in your current folder. A path to another folder with the adapter files yields an error. [optional] [https://wiki.bits.vib.be/index.php/Parameters_of_Trimmomatic];

- `SLIDINGWINDOW` Sliding window trimming which cuts out sequences witin the window and all the subsequent basepairs in the read if the average quality score within the window is lower than a certain threshold. The window moves from the 5'-end to the 3'-end. Note that if the first few reads of a sequence if of low quality, but the remaining of the sequence is of high quality, the entire sequence will be removed just because of the first few bad quality nucleotides. If this sitatuation occurs, it might be useful to first apply the HEADCROP option (see below). Parameters should be given as `SLIDINGWINDOW:L_window:Q_min` where `L_window` is the window size (in terms of basepairs) and `Q_min` the average threshold quality. [optional];
- `LEADING` Cut the bases at the start (5' end) of a read if the quality is below a certain threshold. Note that when, for example, the parameter is set to 3, the quality score Q=0 to Q=2 will be removed. Parameters should be given as `LEADING:Q_min` where `Q_min` is the threshold quality score. All basepairs will be removed until the first basepair that has a quality score above the given threshold. [optional];

- `TRAILING` Cut the bases at the end (3' end) of a read if the quality is below a certain threshold. Note that when, for example, the parameter is set to 3, the quality score Q=0 to Q=2 will be removed. All basepairs will be removed until the first basepair that has a quality score above the given threshold. [optional];
- `CROP` Cuts the read to a specific length by removing a specified amount of nucleotides from the tail of the read (this does not discriminate between quality scores). [optional];
- `HEADCROP` Cut a specified number of bases from the start of the reads (this does not discriminate between quality scores). [optional];
- `MINLEN` Drops a read if it has a length smaller than a specified amount [optional];
- `TOPHRED33` Converts the quality score to phred33 encoding [optional];
- `TOPHRED64` Converts the quality score to phred64 encoding [optional].

Note that the input files can be either uncompressed FASTQ files or gzipped FASTQ (with an extension fastq.gz or fq.gz) and the output fields should ideally have the same extension as the input files (i.e. .fastq or .fq). The convention is using field:parameter, where 'parameter' is typically a number. (To make the (relative long commands) more readable in the command line, use \ and press enter to continue the statement on the next line) (See https://datacarpentry.org/wrangling-genomics/03-trimming/index.html for examples how to run software). Trimmomatic can only run a single file at the time.If more files need to be trimmed using the same parameters, use:

Where `xxx` should be replaced with the commands for trimmomatic.

2b. Trimming of the sequencing reads; BBDuk (38.84)

BBDuk is part of the BBMap package and alters the sequencing result by trimming the reads from unwanted sequences, as is specified by the user. The program does not need to be installed, but after downloading only requires to be unzipped.

Before running BBDuk, a .fasta file can to be created that allows clipping unwanted sequences in the reads. For example, the 'overrepresented sequences' as found by Fastqc can be clipped by adding the sequences to the .fasta file. A .fasta file can be created by simply creating a text file and adding the sequences that need to be clipped, for example, in the form:

```
> Sequence1
CATG
> Sequence2
GATC
```

Or a .fasta can be copied from either Trimmomatic software package or the BBDuk package, both which are provided with some standard adapter sequences. In the case of Trimmomatic it is advised to use the TruSeq3 adapter file when using MiSeq sequencing. To copy the .fasta file to the data folder (see below for detailed explanation) use the following command:

When using the adapter file that comes with BBMap, use the command:

The adapters.fa file from the BBDuk package includes many different adapter sequences, so it might be good to remove everything that is not relevant. One option can be to create a custom fasta file where all the sequences are placed that need to be trimmed and save this file in the bbmap/resources folder. To do this, See here to know how to do it.

- Note to not put empty lines in the text file, otherwise BBDuk might yield an error about not finding the adapters.fa file.
- Typically it is useful to clip overrepresented sequences that were found by FASTQC and sequences that start with 'CATG' or 'GATC' which are the recognition sites for NlaIII and DpnII respectively. Note that the trimming is performed in the order in which the steps are given as input. Typically the adapter clipping is performed as one of the first steps and removing short sequences as one of the final steps.

BBDuk uses a kmers algorithm, which basically means it divides the reads in pieces of length k. (For example, the sequence 'abcd' has the following 2mers: ab,bc,cd). For each of these pieces the software checks if it contains any sequences that are present in the adapter file. The kmers process divides both the reads and the adapters in pieces of length k and it then looks for an exact match. If an exact match is found between a read and an adapter, then that read is trimmed. If the length k is chosen to be bigger than the length of the smallest adapter, then there will never be an exact match between any of the reads and the adapter sequence and the adapter will therefore never be trimmed. However, when the length k is too small the trimming might be too specific and it might trim too many reads. Typically, the length k is chosen about the size of the smallest adapter sequence or slighty smaller. For more details, see this webpage.

A typical command for BBDuk looks like this: **`${path_bbduk_software}/bbduk.sh -Xmx1g in=${pathdata}/${filename} out=${path_trimm_out}${filename_trimmed} ref=${path_bbduk_software}/resources/adapters.fa ktrim=r k=23 mink=10 hdist=1 qtrim=r trimq=14 minlen=30`**

Next an overview is given with some of the most useful options. For a full overview use call `bbduk.sh` in the {bash} without any options.

1. `-Xmx1g`. This defines the memory usage of the computer, in this case 1Gb (`1g`). Setting this too low or too high can result in an error (e.g. 'Could not reserve enough space for object heap'). Depending on the maximum memory of your computer, setting this to `1g` should typically not result in such an error.
2. `in` and `out`. Input and Output files. For paired-end sequencing, use also the commands `in2` and `out2`. The input accepts zipped files, but make sure the files have the right extension (e.g. file.fastq.gz). Use `outm` (and `outm2` when using paired-end reads) to also save all reads that failed to pass the trimming. Note that defining an absolute path for the path-out command does not work properly. Best is to simply put a filename for the file containing the trimmed reads which is then stored in the same directory as the input file and then move this trimmed reads file to any other location using: `mv ${pathdata}${filename::-6}_trimmed.fastq ${path_trimm_out}`
3. `qin`. Set the quality format. 33 for phred 33 or 64 for phred 64 or auto for autodetect (default is auto).
4. `ref`. This command points to a .fasta file containg the adapters. This should be stored in the location where the other files are stored for the BBDuk software (`${path_bbduk_software}/resources`) Next are a few commands relating to the kmers algorithm.
5. `ktrim`. This can be set to either don't trim (`f`, default), right trimming (`r`), left trimming (`l`). Basically, this means what needs to be trimmed when an adapter is found, where right trimming is towards the 5'-end and left trimming is towards the 3'-end. For example, when setting this option to `ktrim=r`, then when a sequence is found that matches an adapter sequence, all the basepairs on the right of this matched sequence will be deleted including the matched sequence itself. When deleting the whole reads, set `ktrim=rl`.
6. `kmask`. Instead of trimming a read when a sequence matches an adapter sequence, this sets the matching sequence to another symbol (e.g. `kmask=N`).
7. `k`. This defines the number of kmers to be used. This should be not be longer than the smallest adapter sequences and should also not be too short as there might too much trimmed. Typically values around 20 works fine (default=27).
8. `mink`. When the lenght of a read is not a perfect multiple of the value of `k`, then at the end of the read there is a sequence left that is smaller than length k. Setting `mink` allows the software to use smaller kmers as well near the end of the reads. The sequence at the end of a read are matched with adapter sequences using kmers with length between mink and k.
9. `minkmerhits`. Determines how many kmers in the read must match the adapter sequence. Default is 1, but this can be increased when for example using short kmers to decrease the chance that wrong sequences are trimmed that happen to have a single matching kmer with an adapter.
10. `minkmerfraction`. A kmer in this read is considered a match with an adapter when at least a fraction of the read matches the adapter kmer.
11. `mincovfraction`. At least a fraction of the read needs to match adapter kmer sequences in order to be regarded a match. Use either `minkmerhits`, `minkmerfraction` or `mincovfraction`, but setting multiple results in that only one of them will be used during the processing.
12. `hdist`. This is the Hamming distance, which is defined as the minimum number of substitutions needed to convert one string in another string. Basically this indicates how many errors are allowed between a read and and an adapter sequence to still count as an exact match. Typically does not need to be set any higher than 1, unless the reads are of very low quality. Note that high values of hdist also requires much more memory in the computer.
13. `restrictleft` and `restrictright`. Only look for kmers left or right number bases.
14. `tpe` and `tbo`: This is only relevant for paired-end reads. `tpe` cuts both the forward and the reverse read to the same length and `tbo` trims the reads if they match any adapter sequence while considering the overlap between two paired reads.

So far all the options were regarding the adapter trimming (more options are available as well, check out the user guide). These options go before any other options, for example the following:

1. `qtrim`. This an option for quality trimming which indicates whether the reads should be trimmed on the right (`r`), the left (`l`), both (`fl`), neither (`f`) or that a slidingwindow needs to be used (`w`). The threshold for the trimming quality is set by `trimq`.
2. `trimq`. This sets the minimum quality that is still allowed using phred scores (e.g. Q=14 corresponds with \(P_{error} = 0.04\)).
3. `minlength`. This sets the minimum length that the reads need to have after all previous trimming steps. Reads short than the value given here are discarded completely.

4. `mlf`. Alternatively to the `minlen` option, the Minimum Length Fraction can be used which determines the fraction of the length of the read before and after trimming and if this drops below a certain value (e.g 50%, so `mlf=50`), then this read is trimmed.
5. `maq`. Discard reads that have an average quality below the specified Q-value. This can be useful after quality trimming to discard reads where the really poor quality basepairs are trimmed, but the rest of the basepairs are of poor quality as well.
6. `ftl` and `ftr` (`forcetrimleft` and `forcetrimright`). This cuts a specified amount of basepairs at the beginning (`ftl`) or the last specified amount of basepairs (`ftr`). Note that this is zero based, for example `ftl=10` trims basepairs 0-9.
7. `ftm`. This force Trim Modulo option can sometimes be useful when an extra, unwanted and typically very poor quality, basepair is added at the end of a read. So when reads are expected to be all 75bp long, this will discard the last basepair in 76bp reads. When such extra basepairs are present, it will be noted in FastQC.

Finally, to check the quality of the trimmed sequence using the command:

3. Sequence alignment and Reference sequence indexing; BWA (0.7.17) (Linux)

The alignment can be completed using different algorithms within BWA, but the 'Maximal Exact Matches' (MEM) algorithm is the recommended one (which is claimed to be the most accurate and fastest algorithm and is compatible with many downstream analysis tools, see documentation for more information). BWA uses a FM-index, which uses the Burrows Wheeler Transform (BWT), to exactly align all the reads to the reference genome at the same time. Each alignment is given a score, based on the number of matches, mismatches and potential gaps and insertions. The highest possible score is 60, meaning that the read aligns perfectly to the reference sequence (this score is saved in the SAM file as MAPQ). Besides the required 11 fields in the SAM file, BWA gives some optional fields to indicate various aspects of the mapping, for example the alignment score (for a complete overview and explanation, see the documentation). The generated SAM file also include headers where the names of all chromosomes are shown (lines starting with SQ). These names are used to indicate where each read is mapped to.

Before use, the reference sequence should be indexed so that the program knows where to find potential alignment sites. This only has to be done *once* for each reference genome. Index the reference genome using the command

This creates 5 more files in the same folder as the reference genome that BWA uses to speed up the process of alignment.

The alignment command should be given as

where `[options]` can be different statements as given in the documentation. Most importantly are:

- `-A` Matching scores (default is 1)
- `-B` Mismatch scores (default is 4)
- `-O` Gap open penalty (default is 6)
- `-E` Gap extension penalty (default is 1)
- `-U` Penalty for unpaired reads (default is 9; only of use in case of paired-end sequencing).

Note that this process might take a while. After BWA is finished, a new .sam file is created in the same folder as the .fastq file.

4. Converting SAM file to BAM file; SAMtools (1.7) and sambamba (0.7.1) (Linux)

SAMtools allows for different additional processing of the data. For an overview of all functions, simply type samtools in the command line. Some useful tools are:

- `view` This converts files from SAM into BAM format. Enter samtools view to see the help for all commands. The format of the input file is detected automatically. Most notably are:
  - `–b` which converts a file to BAM format.
  - `–f` int Include only the reads that include all flags given by int.
  - `–F` int Include only the reads that include none of the flags given by int.
  - `–G` int Exclude only the reads that include all flags given by int.
- `sort` This sorts the data in the BAM file. By default the data is sorted by leftmost coordinate. Other ways of sorting are:
  - `–n` sort by read name
  - `–t` tag Sorts by tag value
  - `–o` file Writes the output to file.

- `flagstats` Print simple statistics of the data.
- `stats` Generate statistics.
- `tview` This function creates a text based Pileup file that is used to assess the data with respect to the reference genome. The output is represented as characters that indicate the relation between the aligned and the reference sequence. The meaning of the characters are:
    - . :base match on the forward strand
    - , :base match on the reverse strand
    - </> :reference skip
    - AGTCN :each one of these letters indicates a base that did not match the reference on the forward strand.
    - Agtcn : each one of these letters indicates a base that did not match the reference on the reverse strand.
    - + [0-9]+[AGTCNagtcn] : Denotes (an) insertion(s) of a number of the indicated bases.
    - -[0-9]+[AGTCNagtcn] : Denotes (an) deletion(s) of a number of the indicated bases.
    - ^ : Start of a read segment. A following character indicates the mapping quality based on phred33 score.
    - $: End of a read segment.
    - * : Placeholder for a deleted base in a multiple basepair deletion.
    - `quickcheck` Checks if a .bam or .sam file is ok. If there is no output, the file is good. If and only if there are warnings, an output is generated. If an output is wanted anyways, use the command `samtools quickcheck –v [input.bam] &&echo 'All ok' || echo 'File failed check'`

Create a .bam file using the command

Check if everything is ok with the .bam file using

- This checks if the file appears to be intact by checking the header is valid, there are sequences in the beginning of the file and that there is a valid End-Of_File command at the end.
- It thus check only the beginning and the end of the file and therefore any errors in the middle of the file are not noted.
- But this makes this command really fast.
- If no output is generated, the file is good.

If desired, more information can be obtained using :

Especially the latter can be a bit overwhelming with data, but this gives a thorough description of the quality of the bam file. For more information see this documentation.

For many downstream tools, the .bam file needs to be sorted. This can be done using SAMtools, but this might give problems. A faster and more reliable method is using the software sambamba using the command

(where `–m` allows for specifying the memory usage which is 500MB in this example). This creates a file with the extension .sorted.bam, which is the sorted version of the original bam file. Also an index is created with the extension .bam.bai. If this latter file is not created, it can be made using the command

Now the reads are aligned to the reference genome and sorted and indexed. Further analysis is done in windows, meaning that the sorted .bam files needs to be moved to the shared folder.

Next, the data analysis is performed using custom made codes in Matlab in Windows.

5. Determining transposon insertions: Matlab (Code from Benoit [Michel et. al. 2017])

Before the data can be used as an input for the Matlab code provided by the Kornmann lab, it needs to be copied from the shared folder to the data folder using the command:

The Matlab code is provided by Benoit (see the website) and is based on the paper by Michel et. al.. Running the code requires the user to select a .bam or .sorted.bam file (or .ordered.bam which is similar to the .sorted.bam). If the .bam file is chosen or there is no .bam.bai (bam index-)file present in the same folder, the script will automatically generate the .sorted.bam and a .bam.bai file. In the same folder as the bam file the Matlab variables 'yeastGFF.mat' and 'names.mat' should be present (which can be found on the website cited above). The script will generate a number of files (some of them are explained below):

1. .sorted.bam (if not present already)
2. .bam.bai (if not present already)
3. .sorted.bam.linearindex
4. .sorted.bam.mat (used as a backup of the matlab script results)
5. .sorted.bam_pergene.txt (contains information about transposons and reads for individual genes)
6. .sorted.bam.bed (contains information about the location and the number of reads per transposon insertion)

7. .sorted.bam.wig (contains information about the location and the number of reads per transposon insertion)

The line numbers below correspond to the original, unaltered code.

[line1-13] After loading the .BAM file, the 'baminfo' command is used to collect the properties for the sequencing data. These include (among others) [https://nl.mathworks.com/help/bioinfo/ref/baminfo.html]:

- `SequenceDictionary`: Includes the number of basepairs per chromosome.
- `ScannedDictionary`: Which chromosomes are read (typically 16 and the mitochondrial chromosome).
- `ScannedDictionaryCount`: Number of reads aligned to each reference sequence.

[line22-79] Then, a for-loop over all the chromosomes starts (17 in total, 16 chromosomes and a mitochondrial chromosome). The for-loop starts with a BioMap command which gets the columns of the SAM-file. The size of the columns corresponds with the number of reads aligned to each reference sequence (see also the 'baminfo' field `ScannedDictionaryCount`). The collected information is:

- `SequenceDictionary`: Chromosome number where the reads are collected for (given in roman numerals or 'Mito' for mitochondrial). (QNAME)
- `Reference`: Chromosome number that is used as reference sequence. (RNAME)
- `Signature`: CIGAR string. (CIGAR)
- `Start`: Start position of the first matched basepair given in terms of position number of the reference sequence. (POS)?
- `MappingQuality`: Value indicating the quality of the mapping. When 60, the mapping has the smallest chance to be wrong. (MAPQ)
- `Flag`: Flag of the read. (FLAG)
- `MatePosition`: (PNEXT)?
- `Quality`: Quality score given in FASTQ format. Each ASCII symbol represents an error probability for a base pair in a read. See [https://drive5.com/usearch/manual/quality_score.html] for a conversion table. (QUAL)
- `Sequence`: Nucleotide sequence of the read. Length should math the length of the corresponding 'Quality' score. (SEQ)
- `Header`: Associated header for each read sequence.
- `Nseqs`: Integer indicating the number of read sequences for the current chromosome.
- `Name`: empty

(Note: Similar information can be obtained using the 'bamread' command (although his is slower than 'BioMap'), which gives a structure element with fields for each piece of information. This can be accessed using: `bb = bamread(file,infobam.SequenceDictionary(kk).SequenceName,\[1 infobam.SequenceDictionary(kk).SequenceLength\]))` `bb(x).'FIELD' %where x is a row (i.e. a specific read) and FIELD is the string of the field.`)

After extracting the information from the SAM-file (using the 'BioMap' command), the starting site is defined for each read. This is depended on the orientation of the read sequence. If this is normal orientation it has flag=0, if it is in reverse orientation it has flag=16. If the read sequence is in reverse orientation, the length of the read sequence ('readlength' variable) needs to be added to the starting site ('start' variable). The corrected starting sites of all reads are saved in a new variable ('start2'). Note that this changes the order of which the reads are stored. To correct this, the variables 'start2' and 'flag2' are sorted in ascending order.

Now, all the reads in the current chromosome are processed. Data is stored in the ''tncoordinates' variable. This consists of three numbers; the chromosome number ('kk'), start position on the chromosome ('start2') and the flag ('flag2'). All reads that have the same starting position and the same flag (i.e. the same reading orientation) are stored as a single row in the 'tncoordinates' variable (for this the average starting position is taken). This results in an array where each row indicates a new starting position and the reading orientation. The number of measurements that were averaged for a read is stored in the variable 'readnumb'. This is important later in the program for determining the transposons.

This is repeated for each chromosome due to the initial for-loop. The 'tnnumber' variable stores the number of unique starting positions and flags for each chromosome (this does not seem to be used anywhere else in the program).

[line94-120] After getting the needed information from the SAM-file, the data needs to be compared with the literature. For this yeastGFF.mat is used (provided by Benoit et. al.) that loads the variable 'gff'. This includes all genes (from SGD) and the essential genes (from YeastMine). (Note that a similar list can be downloaded from the SGD website as a text file). The file is formatted as a matrix with in each row a DNA element (e.g. genes) and each column represent a different piece of information about that element. The used columns are:

1. Chromosome number (represented as roman numerals)
2. Data source (either SGD, YeastMine or landmark. Represented as a string)

3. Type of the DNA element, e.g. gene. The first element of a chromosome is always the ‘omosome’, which is the entire chromosome (Represented as a string)
4. Start coordinates (in terms of base pairs. Represented as an integer)
5. End coordinates (in terms of base pairs. Represented as an integer)
6. A score value. Always a ‘.’ Represented as a string indicating a dummy value.
7. Reading direction (+ for forward reading (5’-3’), - for reverse reading (3’-5’), ‘.’ If reading direction is undetermined. Represented as a string)
8. Always a ‘.’ Except when the element is a Coding DNA Sequence (CDS), when this column become a ‘0’. A CDS always follows a gene in the list and the value indicates how many basepairs should be removed from the beginning of this feature in order to reach the first codon in the next base (Represented as a string)
9. Other information and notes (Represented as a string)

From the ‘gff’ variable, all the genes are searched and stored in the variable ‘features.genes’ (as a struct element). The same thing is done for the essential genes by searching for genes from the ‘YeastMine’ library and these are stored in ‘features.essential’ (as a struct element). This results in three variables:

- `features`: struct element that includes the fields ‘genes’ and ‘essential’ that include numbers representing in which rows of the ‘gff’ variable the (essential) gene can be found (note that the essential genes are indicated by ‘ORF’ from ’Yeastmine’ in the ‘gff’ variable).
- `genes`: struct element storing the start and end coordinates (in basepairs) and the chromosome number where the gene is found.
- `essential`: struct element. Same as ‘genes’, but then for essential genes as found by the ‘YeastMine’ database.

This can be extended with other features (e.g. rRNA, see commented out sections in the code).

[line124-160] Next all the data is stored as if all the chromosomes are put next to each other. In the `tncoordinates` variable, for each chromosome the counting of the basepairs starts at 1. The goal of this section is to continue counting such that the first basepair number of a chromosome continues from the last basepair number of the previous chromosome (i.e. the second chromosome is put after the first, the third chromosome is put after the second etc., to create one very long chromosome spanning the entire DNA). The first chromosome is skipped (i.e. the for-loop starts at 2) because these basepairs does not need to be added to a previous chromosome. This is repeated for the start and end coordinates of the (essential) genes.

[line162-200] Now the number of transposons is determined which is done by looking at the number of reads per gene. First, all the reads are found that have a start position between the start and end position of the known genes. The indices of those reads are stored in the variable ‘xx’. In the first for-loop of the program (see lines 22-79) the reads (or measurements) that had the same starting position and flag where averaged and represented as a single read. To get the total number of reads per gene the number of measurements that were used for averaging the reads corresponding to the indices in ‘xx’ are summed (the value stored in the variable ‘readnumb’). This is repeated for all genes and the essential genes using a for-loop. The maximum value is subtracted as a feature to suppress noise or unmeaningful data (see a more detailed explanation the discussion by Galih in the forum of Benoit [https://groups.google.com/forum/#!category-topic/satayusers/bioinformatics/uaTpKsmgU6Q]).

[line226-227] Next all variables in the Matlab workspace are saved using the same name as the .bam file, but now with the .mat extension. The program so far does not need to be ran all over again but loading the .mat file loads all the variables.

Next a number of files are generated (.bed, .txt and .wig).

[line238-256] A .bed file is generated that can be used for visualization of the read counts per insertion site. This contains the information stored in the ‘tncoordinates’ variable. This includes the chromosome number and the start position of the reads. The end position of the reads is taken as the start position +1 (The end position is chosen like this just to visualize the transposon insertion site). The third column is just a dummy variable and can be ignored. As the reads were averaged if multiple reads shared the same location on the genome, the total number of reads is taken from the ‘readnumb’ variable and is stored in the fourth column of the file using the equation 100+readnumb(i)*20 (e.g. a value of 4 in `readnumb` is stored as 180 in the .bed file).

In general a .bed file can contain 12 columns, but only the first three columns are obligatory. These are the chromosome number, start position and end position (in terms of basepairs), respectively. More information can be added as is described in the links below. If a column is filled, then all the previous columns need to be filled as well. If this information is not present or wanted, the columns can be filled with a dummy character (typically a dot is used) [https://bedtools.readthedocs.io/en/latest/content/general-usage.html] [https://learn.gencore.bio.nyu.edu/ngs-file-formats/bed-format/].

[line238-256] Next a text file is generated for storing information about the transposon counts per gene. This is therefore a summation of all the transposons that have an insertion site within the gene. (To check a value in this file, look up the location of a gene present in this file. Next look how many transposon are located within the range spanned by the gene using the .bed file). To create this the names.mat file is needed to create a list of gene names present in the first column. The transposon count is taken from the `tnpergene` variable and is stored in the second column of the file. The third is the number of reads per gene which is taken from the `readpergene` variable (which is calculated by `readnumb-max(readnumb)` where the `readnumb` variable is used for keeping track of the number of reads that were used to average the reads).

[line260-299] Creating a .wig file. This indicates the transposon insertion sites (in terms of basepairs, starting counting from 1 for each new chromosome). The file consists of two columns. The first column represent the insertion site for each transposons and the second column is the number of reads in total at that location. The information is similar to that found in the .bed file, but here the read count is the actual count (and thus not used the equation 100+transposon_count*20 as is done in the .bed file).

Bibliography

Chen, P., Wang, D., Chen, H., Zhou, Z., & He, X. (2016). The nonessentiality of essential genes in yeast provides therapeutic insights into a human disease. Genome research, 26(10), 1355-1362.

Del Fabbro, C., Scalabrin, S., Morgante, M., & Giorgi, F. M. (2013). An extensive evaluation of read trimming effects on Illumina NGS data analysis. PloS one, 8(12).

Delhomme, N., Mähler, N., Schiffthaler, B., Sundell, D., Mannepperuma, C., & Hvidsten, T. R. (2014). Guidelines for RNA-Seq data analysis. Epigenesys protocol, 67, 1-24.

MacManes, M. D. (2014). On the optimal trimming of high-throughput mRNA sequence data. Frontiers in genetics, 5, 13.

Michel, A. H., Hatakeyama, R., Kimmig, P., Arter, M., Peter, M., Matos, J., … & Kornmann, B. (2017). Functional mapping of yeast genomes by saturated transposition. Elife, 6, e23570.

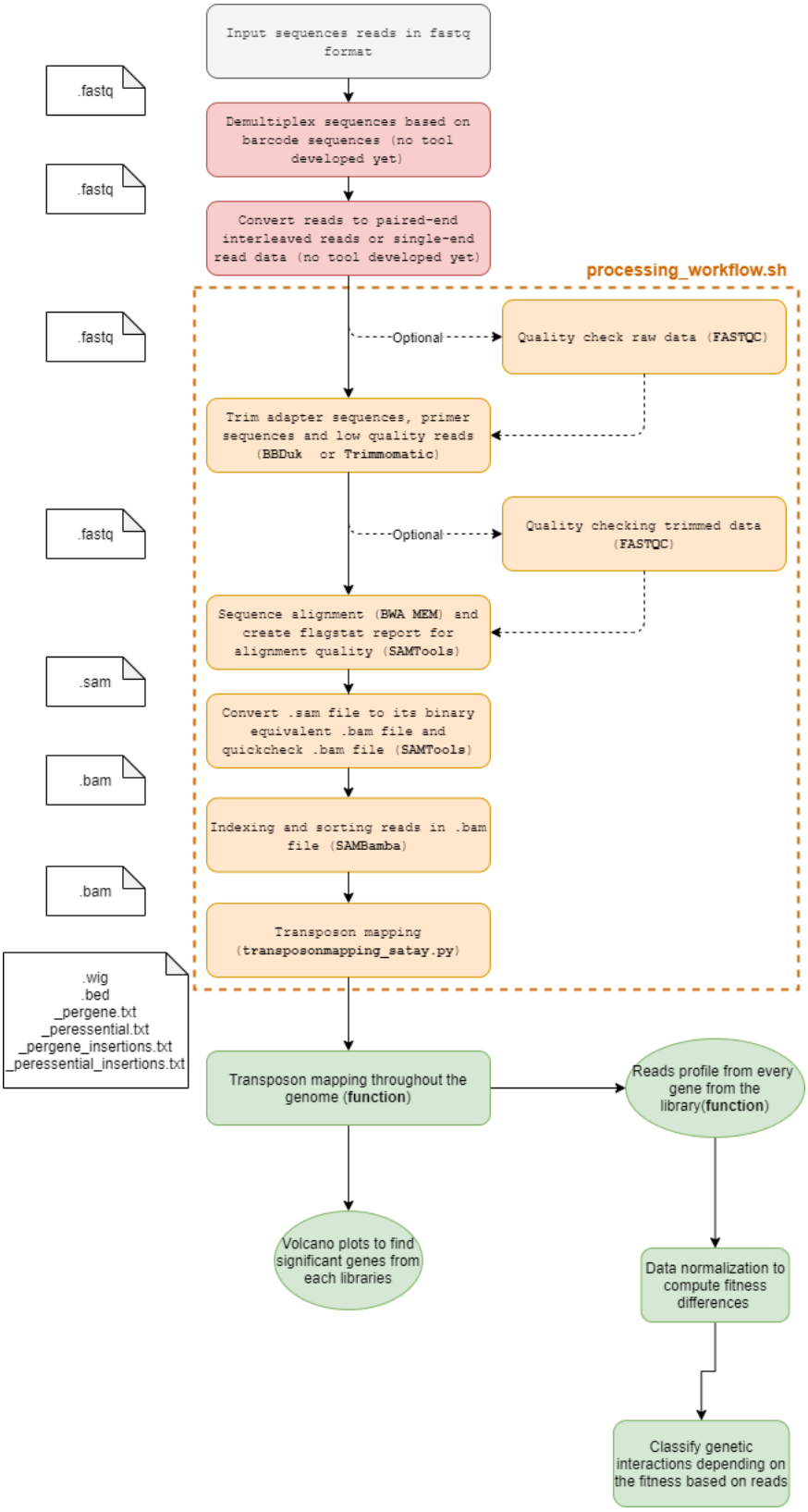
Pfeifer, S. P. (2017). From next-generation resequencing reads to a high-quality variant data set. Heredity, 118(2), 111-124.

Segal, E. S., Gritsenko, V., Levitan, A., Yadav, B., Dror, N., Steenwyk, J. L., … & Kunze, R. (2018). Gene essentiality analyzed by in vivo transposon mutagenesis and machine learning in a stable haploid isolate of Candida albicans. MBio, 9(5), e02048-18.

Usaj, M., Tan, Y., Wang, W., VanderSluis, B., Zou, A., Myers, C. L., … & Boone, C. (2017). TheCellMap. org: A web-accessible database for visualizing and mining the global yeast genetic interaction network. G3: Genes, Genomes, Genetics, 7(5), 1539-1549

> ***"I want to be a healer, and love all things that grow and are not barren"***
>
> - J.R.R. Tolkien

Flowchart of the pipeline



What is SATAY

SAturated Transposon Analysis in Yeast (SATAY) is method of transposon analysis optimised for usage in Saccharomyces Cerevisiae. This method uses transposons (short DNA sequences, also known as jumping genes) which can integrate in the native yeast DNA at random locations. A transposon insertion in a gene inhibits this gene to be translated into a functional protein, thereby inhibiting this gene function. The advantage of this method is that it can be applied in many cells at the same time. Because of the random nature of the transposons the insertion coverage will be more or less equal over the genome. When enough cells are used it is expected that, considering the entire pool of cells, all genes will be inhibited by at least a few transposons. After a transposon insertion, the cells are given the opportunity to

grow and proliferate. Cells that have a transposon inserted in an essential genomic region (and thus blocking this essential function), will proliferate only very little or not at all (i.e. these cells have a low fitness) whilst cells that have an insertion in a non-essential genomic region will generate significantly more daughter cells (i.e. these cells have a relative high fitness). The inserted transposon DNA is then sequenced together with a part of the native yeast DNA right next to the transposon. This allows for finding the genomic locations where the transposon is inserted by mapping the sequenced native DNA to a reference genome. Non-essential genomic regions are expected to be sequenced more often compared to the essential regions as the cells with a non-essential insertion will have proliferated more. Therefore, counting how often certain insertion sites are sequenced is a method for probing the fitness of the cells and therefore the essentiality of genomic regions. For more details about the experimental approach, see the paper from Michel et.al. 2017 and this website from the Kornmann-lab.

This method needs to be performed on many cells to ensure a high enough insertion coverage such that each gene is inhibited in at least a few different cells. After transposon insertion and proliferation, the DNA from each of these cells is extracted and this is sequenced to be able to count how often each genomic region occurs. This can yield tens of millions of sequencing reads per dataset that all need to be aligned to a reference genome. To do this efficiently, a processing workflow is generated which inputs the raw sequencing data and outputs lists of all insertion locations together with the corresponding number of reads. This workflow consists of quality checking, sequence trimming, alignment, indexing and transposon mapping. This documentation explains how each of these steps are performed, how to use the workflow and discusses some python scripts for checking the data and for postprocessing analysis.

A SATAY FORUM

If you want to get started, just join this FORUM. And the website for the library generation can be found HERE

Gene Essentiality

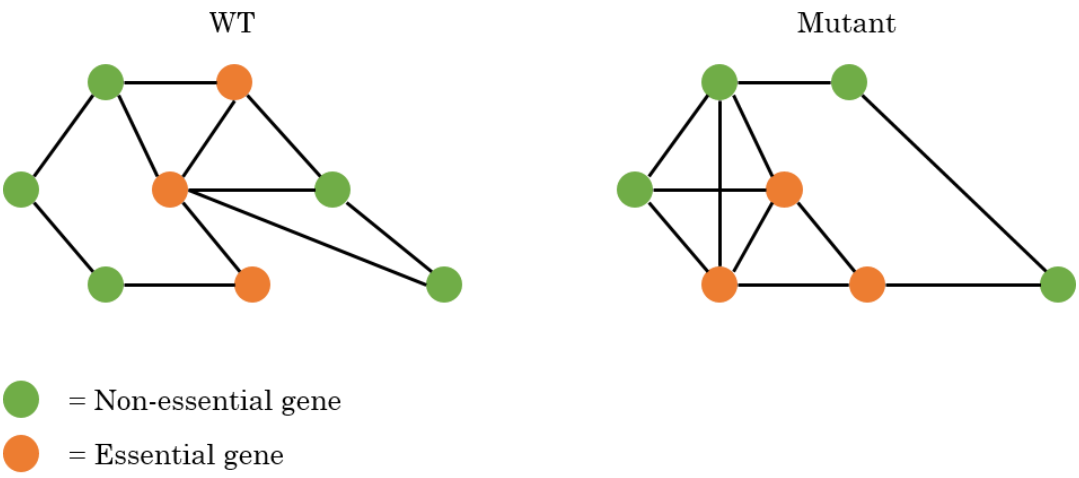
Essentiality of genes are defined as its deletion is detrimental to cell in the form that either the cell cannot grow anymore and dies, or the cell cannot give rise to offspring. Essentiality can be grouped in two categories, namely type I and type II [Chen et.al. 2016].

- Type I essential genes are genes, when inhibited, show a loss-of-function that can only be rescued (or masked) when the lost function is recovered by a gain-of-function mechanism.
- Typically these genes are important for some indispensable core function in the cell (e.g. Cdc42 in *S. Cerevisiae* that is type I essential for cell polarity).
- Type II essential genes are the ones that look essential upon inhibition, but the effects of its inhibition can be rescued or masked by the deletion of (an)other gene(s).
- These genes are therefore not actually essential, but when inhibiting the genes some toxic side effects are provoked that are deleterious for the cells.

The idea is that the essentiality of genes (both type I and type II), may change between different genetic backgrounds. For changes in essentiality four cases are considered:

1. A gene is **essential** in WT and remains **essential** in the mutant.
2. A gene is **non-essential** in WT and remains **non-essential** in the mutant.
3. A gene is **essential** in WT and becomes **non-essential** in the mutant.
4. A gene is **non-essential** in WT and becomes **essential** in the mutant.

An example is given in the figure below, where an interaction map is shown for WT cells and a possible interaction map for a mutant where both the essentiality and the interactions are changed.



Situation 1 and 3 are expected to be the trickiest since those ones are difficult to validate. To check the synthetic lethality in cells, a double mutation needs to be made where one mutation makes the genetic background and the second mutation should confirm whether the second mutated gene is actually essential or not. This is typically made by sporulating the two mutants, but deleting a gene that is already essential in wild type prevents the cell from growing or dividing and can therefore not be sporulated with the mutant to create the double deletion. Therefore, these double mutants are hard or impossible to make.

Another point to be aware of is that some genes might be essential in specific growth conditions (see also the subsection). For example, cells that are grown in an environment that is rich of a specific nutrient, the gene(s) that are required for the digestion of this nutrient might be essential in this condition. The SATAY experiments will therefore show that these genes are intolerant for transposon insertions. However, when the cells are grown in another growth condition where mainly other nutrients are present, the same genes might now not be essential and therefore also be more tolerant to transposon insertions in that region. It is suggested to compare the results of experiments with cells from the same genetic background grown in different conditions with each other to rule out conditions specific results.

We want to check the essentiality of all genes in different mutants and compare this with both wild type cells and with each other. The goal is to make an overview of the changes in the essentiality of the genes and the interaction network between the proteins. With this we aim to eventually be able to predict the synthetic lethality of multiple mutations based on the interaction maps of the individual mutations.

The yeast interaction network

The yeast interaction network is already made based on previous research see here >> thecellmap.org.

Current progress in Machine Learning workflows!

Currently the idea is to use machine learning that uses the results from the transposon sequencing experiments, the interaction map of genes and possibly GO-terms. Some implementations of Machine learning workflows can be seen here, in another Jupyter Book :)

Interpreting Transposon Counts & Reads

Once cells have a transposon inserted somewhere in the DNA, the cells are let to grow so they can potentially generate a few new generations. A cell with a transposon inserted in an essential part of its DNA grows very slowly or might not grow at all (due to its decreased fitness). Since the sequencing starts only at the location of a transposon insertion (see experimental methods section), it can be concluded that roughly each read from the sequencing corresponds with a transposon insertion (roughly mainly because transposon inserted in essential genes can generate no reads). Cells with a transposon inserted in an essential genomic region, will not have divided and therefore will not have a contribution to the sequencing reads. When the reads are aligned to a reference sequence and the number of reads are mapped against the genome, empty regions indicate possible essential genes. Negative selection can thus be found by looking for empty regions in the reads mapping. When a transposon is inserted in a non-essential genomic region, these cells can still divide and give rise to offspring and after sequencing the non-essential regions will be represented by relatively many reads.

During processing the genes can be analysed using the number of transposon insertions per gene (or region) or the number of reads per gene. Reads per gene, instead of transposons per gene, might be a good measure for positive selection since it is more sensitive (bigger difference in number of reads between essential and non-essential genes), but also tends to be nosier. Transposons per gene is less noisy, but is also less sensitive since the number of transposons inserted in a gene does not change in subsequent generations of a particular cell. Therefore it is hard to tell the fitness of cells when a transposon is inserted a non-essential region solely based on the number of transposon insertions.

Ideally only the transposons inserted in non-essential genomic regions will have reads (since only these cells can create a colony before sequencing), creating a clear difference between the essential and non-essential genes. However, sometimes non-essential genes also have few or no transposon insertion sites. According to Michel et.al. this can have 3 main reasons.

1. During alignment of the reads, the reads that represent repeated DNA sequences are discarded, since there is no unique way of fitting them in the completed sequence. (Although the DNA sequence is repeated, the number of transposon counts can differ between the repeated regions) Transposons within such repeated sequences are therefore discarded and the corresponding reads not count. If this happens at a non-essential gene, it appears that it has no transposons, but this is thus merely an alignment related error in the analysis process.
2. Long stretches of DNA that are without stop codons, called Open Reading Frames (ORF), typically code for proteins. Some dubious ORF might overlap with essential proteins, so although these ORF themselves are not essential, the overlapping part is and therefore they do not show any transposons.
3. Some genes are essential only in specific conditions. For example, genes that are involved in galactose metabolism are typically not essential, as inhibition of these genes block the cell's ability to digest galactose, but it can still survive on other nutrition's. In lab conditions however, the cells are typically grown in galactose rich media, and inhibiting the genes for galactose metabolism cause starvation of the cells.
4. A gene might not be essential, but its deletion might cripple the cell so that the growth rate decreases significantly. When the cells are grown, the more healthy cells grow much faster and, after some time, occur more frequently in the population than these crippled cells and therefore these cells might not generate many reads or no reads at all. In the processing, it might therefore look as if these genes are essential, but in fact they are not. The cells just grow very slowly.

The other way around might also occur, where essential genes are partly tolerant to transposons. This is shown by Michel et.al. to be caused that some regions (that code for specific subdomains of the proteins) of the essential genes are dispensable. The transposons in these essential genes are clearly located at a specific region in the gene, the one that codes for a non-essential subdomain. However, this is not always possible, as in some cases deletion of non-essential subdomains of essential genes create unstable, unexpressed or toxin proteins. The difference in essentiality between subdomains in a single protein only happens in essential genes, not in non-essential genes. Michel et.al. devised an algorithm to estimate the likelihood L of a gene having an essential subdomain:

$$L = \frac{\text{d }N_{\text{cds}}}{l_{\text{cds}}}$$

where d is the longest interval (in terms of base pairs) between 5 neighbouring transposons in a Coding DNA Sequence (cds) (≥ 300 bp), N_{cds} is the total number transposons mapping in the cds (≥ 20) transposons) and l_{cds} is the total length of the CDS. Additionally, it must hold that $0.1\, l_{cds} \leq d \leq 0.9\, l_{cds}$.

It is expected that only essential genes carry essential subdomains, and indeed what was found by Michel et.al. that the genes with the highest likelihood were mostly also genes previously annotated as essential by other studies.

Because of the reasons mentioned before, not a simple binary conclusion can be made solely based on the amount of transposon insertions or the number of reads. Instead, a gene with little reads *might* be essential, but to be sure the results from other experiments need to be implemented as well, for example where the cells were grown in a different growth conditions. Therefore, SATAY analysis only says something about the relative fitness of cells where a specific gene is inhibited in the current growth conditions.

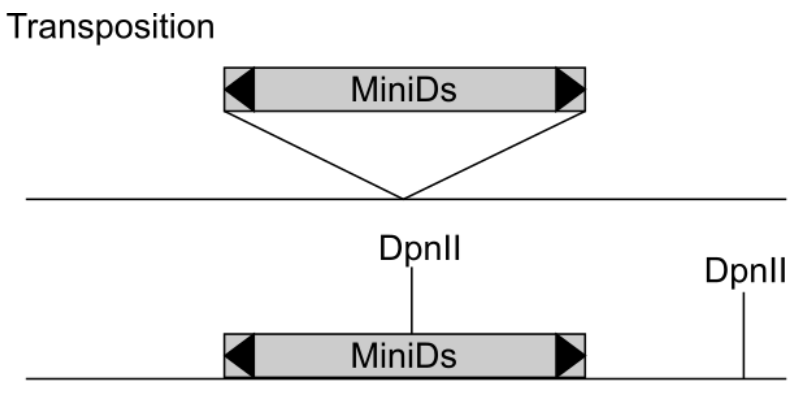
> **Quality check**
>
> Comparison of our results with those obtained by the Kornmann lab might confirm the quality of our experimental and analysis methods. PUT FIGURE

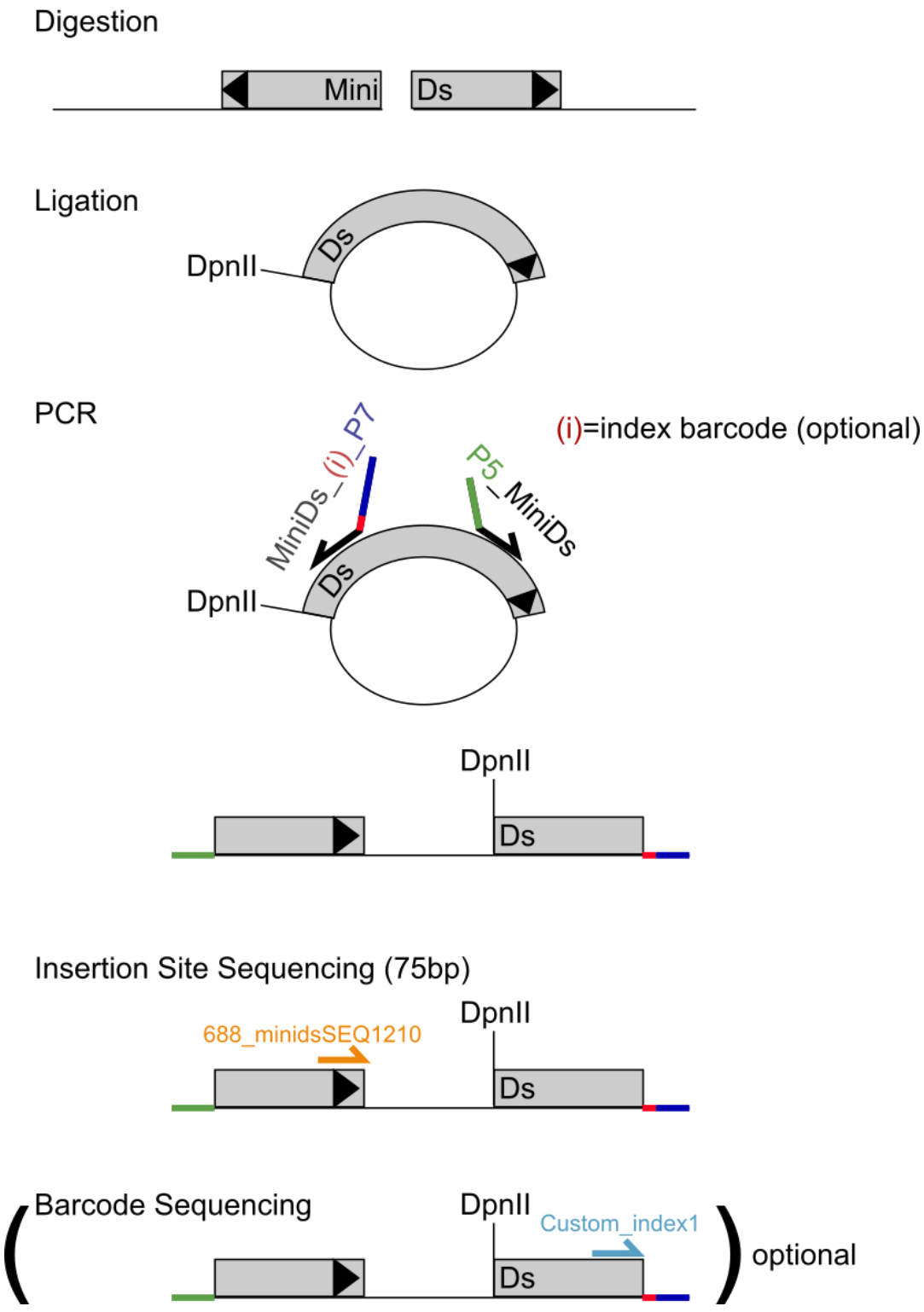
Experimental Process

- The process of SATAY starts with inserting a plasmid in the cells that contains a transposase (TPase) and the transposon (MiniDs) flanked on both sides by adenine (ADE). The transposon has a specific, known, sequence that codes for the transposase that cuts the transposon from the plasmid (or DNA) to (another part of) the DNA.

Simplified example for the transposon insertion plasmid.

- The MiniDs transposon is cut loose from the plasmid and randomly inserted in the DNA of the host cell.
- If the transposon is inserted in a gene, the gene can still be transcribed by the ribosomes, but typically cannot be (properly) translated in a functional protein.
- The genomic DNA (with the transposon) is cut in pieces for sequencing using enzymes, for example DpnII.
- This cuts the DNA in many small pieces (e.g. each 75bp long) and it always cuts the transposon in two parts (i.e. digestion of the DNA).
- Each of the two halves of the cut transposon, together with the part of the gene where the transposon is inserted in, is ligated meaning that it is folded in a circle.
- A part of the circle is then the half transposon and the rest of the circle is a part of the gene where the transposon is inserted in.
- Using PCR and primers, this can then be unfolded by cutting the circle at the halved transposon.
- The part of the gene is then between the transposon quarters.
- Since the sequence of the transposon is known, the part of the gene can be extracted.
- This is repeated for the other half of the transposon that includes the other part of the gene.
- When both parts of the gene are known, the sequence from the original gene can be determined.





Sequence alignment

- To get the order of nucleotides in a genome, shotgun sequencing is used where the genome is cut is small pieces called reads (typically tens to a few hundred basepairs long).
- The reads have overlapping regions that can be used to identify their location with respect to a reference genome and other reads (i.e. mapping of the reads).
- Mapping of the reads result in contigs, which are multiple mapped reads that form continuous assembled parts of the genome (contigs can be the entire target genome itself).
- All contigs should be assembled to form (a large part of) the target genome.

- The sequence assembly problem can be described as: *Given a set of sequences, find the minimal length string containing all members of the set as substrings*.

The reads from the sequencing can be single-end or paired-end, which indicates how the sequencing is performed. In paired-end sequencing, the reads are sequenced from both directions, making the assembly easier and more reliable, but results in twice as many reads as single-end reads. The reason of the more reliable results has to do with ambiguous reads that might occur in the single-end sequencing. Here, a read can be assigned to two different locations on the reference genome (and have the same alignment score). In these cases, it cannot be determined where the read should actually be aligned (hence its position is ambiguous). In paired-end sequencing, each DNA fragment has primers on both ends, meaning that the sequencing can start in both the 5'-3' direction and in the 3'-5' direction. Each DNA fragment therefore has two reads both which have a specified length that is shorter than the entire DNA fragment. This results that a DNA fragment is read on both ends, but the central part will still be unknown (as it is not covered by these two particular reads, but it will be covered by other reads). Since you know that the two reads belong close together, the alignment of one read can be checked by the alignment of the second read (or paired mate) somewhere in close vicinity on the reference sequence. This is usually enough for the reads to become unambiguous.

The resulting data from the sequencing is stored in a FASTQ file where all individual reads are stored including a quality score of the sequencing. The reads are in random order and therefore the first step in the processing is aligning of the reads in the FASTQ files with a reference genome.

Note that the quality of the reads typically decreases near the 3'-end of the reads due to the chemistry processes required for sequencing (this depends on the kind of method used). For Illumina sequencing, the main reasons are signal decay and dephasing, both causing a relative increase in the background noise. Dephasing occurs when a DNA fragment is not de-blocked properly. A DNA fragment is copied many times and all copies incorporate a fluorescent nucleotide that can be imaged to identify the nucleotide. If there are 1000 copies of the DNA fragment, there are 1000 fluorescent nucleotides that, ideally, are all the same to create a high quality signal. After imaging, the DNA fragment is de-blocked to allow a new fluorescent nucleotide to bind. This deblocking might not work for all copies of the DNA fragment. For example, 100 copies might not be deblocked properly, so for the next nucleotide only 900 copies agree for the next incorporated nucleotide. For the third round, the 100 copies that were not deblocked properly in the second round, might now be deblocked as well, but now they are lagging behind one nucleotide, meaning that in the coming rounds they have consistently the wrong nucleotide. As the reads increases in length, more rounds are needed and therefore the chances of dephasing increases causing a decrease in the quality of the reads. This gives noise in the signal of the new nucleotide and therefore the quality of the signal decreases. For example, take the next 6bp sequence that is copied 5 times:

1. `GATGTC`
2. `GATGTC`
3. `G ATGT`
4. `GAT GT`
5. `G AT G`

The first two reads are deblocked properly and they give all the right nucleotides. But the third and fourth have one round that is not deblocked properly (indicated by the empty region between the nucleotides), hence the nucleotide is always lagging one bp after the failed deblocking. The fifth copy has two failed deblocking situations, hence is lagging two bp. The first nucleotide is a G for all 5 copies, therefore the quality of this nucleotide is perfect. But, by the end of the read, only two out of five copies have the correct bp (i.e. a C), therefore the quality is poor for this nucleotide. (It can either be a C or a T with equal likelyhood or potentially a G, so determining which nucleotide is correct is ambiguous without knowing which reads are lagging, which you don't know). (See for example this question on seqanswers or the paper by [Pfeifer, 2016])

File Types

fastq

This is the standard output format for sequencing data. It contains all (raw) sequencing reads in random order including a quality string per basepair. Each read has four lines:

1. Header: Contains some basic information from the sequencing machine and a unique identifier number.
2. Sequence: The actual nucleotide sequence.
3. Dummy: Typically a '+' and is there to separate the sequence line from the quality line.
4. Quality score: Indicates the quality of each basepair in the sequence line (each symbol in this line belongs to the nucleotide at the same position in the sequence line). The sequence and this quality line should always have the same length.

The quality line is given as a phred score. There are two versions, base33 and base64, but the base64 is outdated and hardly used anymore. In both versions the quality score is determined by Q = -10*log10§ where P is the error probability determined during sequencing (0 < P < 1). A Q-score of 0 (i.e. an error probability of P=1) is defined by ascii symbol 33 ('!') for base33 and by ascii symbol 64 ('@') for base64. A Q-score of 1 (p=0.79) is then given by ascii 34 (' " ') (for base33) etcetera. For a full table of ascii symbols and probability scores, see the appendices of this document PHRED table (base33) and PHRED table (base64). Basically all fastq files that are being created on modern sequencing machines use the base33 system.

The nucleotide sequence typically only contains the four nucleotide letters (A, T, C and G), but when a nucleotide was not accurately determined (i.e. having a error probability higher than a certain threshold), the nucleotide is sometimes converted to the letter N, indicating that this nucleotide was not successfully sequenced.

Fastq files tend to be large in size (depending on how many reads are sequenced, but >10Gb is normal). Therefore these files are typically compressed in gzip format (.fastq.gz). The pipeline can handle gzipped files by itself, so there is no need to convert it manually.

Example fastq file:

```
@NB501605:544:HLHLMBGXF:1:11101:9938:1050 1:N:0:TGCAGCTA
TGTCAACGGTTTAGTGTTTTCTTACCCAATTGTAGAGACTATCCACAAGGACAATATTTGTGACTTATGTTATGCG
+
AAAAAEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
@NB501605:544:HLHLMBGXF:1:11101:2258:1051 1:N:0:TACAGCTA
TGAGGCACCTATCTCAGCGATCGTATCGGTTTTCGATTACCGTATTTATCCCGTTCGTTTTCGTTGCCGCTATTT
+
AAAAAEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE6EEEEEEEE<EEEAEEEEEEE/E/E/EA///
@NB501605:544:HLHLMBGXF:1:11101:26723:1052 1:N:0:TGCAGCTA
TGTCAACGGTTTAGTGTTTTCTTACCCAATTGTAGAGACTATCCACAAGGACAATATTTGTGACTTATGTTATGCG
+
AAAAAEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
```

sam, bam

When the reads are aligned to a reference genome, the resulting file is a Sequence Alignment Mapping (sam) file. Every read is one line in the file and consists of at least 11 tab delimited fields that are always in the same order:

1. Name of the read. This is unique for each read, but can occur multiple times in the file when a read is split up over multiple alignments at different locations.
2. Flag. Indicating properties of the read about how it was mapped (see below for more information).
3. Chromosome name to which the read was mapped.
4. Leftmost position in the chromosome to which the read was mapped.
5. Mapping quality in terms of Q-score as explained in the fastq section.
6. CIGAR string. Describing which nucleotides were mapped, where insertions and deletions are and where mismatches occurs. For example, `43M1I10M3D18M` means that the first 43 nucleotides match with the reference genome, the next 1 nucleotide exists in the read but not in the reference genome (insertion), then 10 matches, then 3 nucleotides that do not exist in the read but do exist in the reference genome (deletions) and finally 18 matches. For more information see this website.
7. Reference name of the mate read (when using paired end datafiles). If no mate was mapped (e.g. in case of single end data or if it was not possible to map the mate read) this is typically set to `*`.
8. Position of the mate read. If no mate was mapped this is typically set to `0`.
9. Template length. Length of a group (i.e. mate reads or reads that are split up over multiple alignments) from the left most base position to the right most base position.
10. Nucleotide sequence.
11. Phred score of the sequence (see fastq section).

Depending on the software, the sam file typically starts with a few header lines containing information regarding the alignment. For example for BWA MEM (which is used in the pipeline), the sam file start with `@SQ` lines that shows information about the names and lengths for the different chromosome and `@PG` shows the user options that were set regarding the alignment software. Note that these lines might be different when using a different alignment software. Also, there is a whole list of optional fields that can be added to each read after the first 11 required fields. For more information, see wikipedia.

The flag in the sam files is a way of representing a list of properties for a read as a single integer. There is a defined list of properties in a fixed order:

1. read paired
2. read mapped in proper pair
3. read unmapped
4. mate unmapped
5. read reverse strand
6. mate reverse strand
7. first in pair
8. second in pair
9. not primary alignment
10. read fails platform/vendor quality checks
11. read is PCR or optical duplicate
12. supplementary alignment

To determine the flag integer, a 12-bit binary number is created with zeros for the properties that are not true for a read and ones for those properties that are true for that read. This 12-bit binary number is then converted to a decimal integer. Note that the binary number should be read from right to left. For example, FLAG=81 corresponds to the 12-bit binary 000001010001 which indicates the properties: 'read paired', 'read reverse strand' and 'first in pair'. Decoding of sam flags can be done using this website or using samflag.py.

Example sam file (note that the last read was not mapped):

```
NB501605:544:HLHLMBGXF:1:11101:25386:1198     2064    ref|NC_001136|  362539  0
42H34M  *       0       0       GATCACTTCTTACGCTGGGTATATGAGTCGTAAT
EEEAEEEEEEAEEEAEEAEEEEEEEEEEEAAAAA      NM:i:0  MD:Z:34 AS:i:34 XS:i:0
SA:Z:ref|NC_001144|,461555,+,30S46M,0,0;
NB501605:544:HLHLMBGXF:1:11101:20462:1198       16      ref|NC_001137|  576415  0
75M     *       0       0
CTGTACATGCTGATGGTAGCGGTTCACAAAGAGCTGGATAGTGATGATGTTCCAGACGGTAGATTTGATATATTA
EEEAEEEEEEEEEEEEEAEEAE/EEEEEEEEEEEAEEEEEE/EEEEEAEAEEEEEEEEEEEEEEEEEEEAAAAA     NM:i:1
MD:Z:41C33      AS:i:41 XS:i:41
NB501605:544:HLHLMBGXF:1:11101:15826:1199       4       *       0       0       *
*       0       0       ACAATATTTGTGACTTATGTTATGCG      EEEEEEEEEEE6EEEEEEEEEEEEE
AS:i:0  XS:i:0
```

Sam files tend to be large in size (tens of Gb is normal). Therefore the sam files are typically stored as compressed binary files called bam files. Almost all downstream analysis tools (at least all tools discussed in this document) that need the alignment information accept bam files as input. Therefore the sam files are mostly deleted after the bam file is created. When a sam file is needed, it can always be recreated from the bam file, for example using `SAMTools` using the command `samtools view -h -o out.sam in.bam`. The bam file can be sorted (creating a .sorted.bam file) where the reads are typically ordered depending on their position in the genome. This usually also comes with an index file (a .sorted.bam.bai file) which stores some information where for example the different chromosomes start within the bam file and where specific often occuring sequences are. Many downstream analysis tools require this file to be able to efficiently search through the bam file.

bed

A bed file is one of the outputs from the transposon mapping pipeline. It is a standard format for storing read insertion locations and the corresponding read counts. The file consists of a single header, typically something similar to `track name=[file_name] userscore=1`. Every row corresponds to one insertion and has (in case of the satay analysis) the following space delimited columns:

1. chromosome (e.g. `chrI` or `chrref|NC_001133|`)
2. start position of insertion
3. end position of insertion (in case of satay-analysis, this is always start position + 1)
4. dummy column (this information is not present for satay analysis, but must be there to satisfy the bed format)
5. number of reads at that insertion location

In case of processing with `transposonmapping.py` (final step in processing pipeline) or the matlab code from the kornmann-lab, the number of reads are given according to `(reads*20)+100`, for example 2 reads are stored as 140.

The bed file can be used for many downstream analysis tools, for example genome_browser.

Sometimes it might occur that insertions are stored outside the chromosome (i.e. the insertion position is bigger than the length of that chromosome). Also, reference genomes sometimes do not have the different chromosomes stored as roman numerals (for example `chrI`, `chrII`, etc.) but rather use different names (this originates from the chromosome names used in the reference genome). These things can confuse some analysis tools, for example the genome_browser. To solve this, the python function clean_bedwigfiles.py is created. This creates a _clean.bed file where the insertions outside the chromosome are removed and all the chromosome names are stored with their roman numerals. See clean_bedwigfiles.py for more information.

Example bed file:

```
track name=leila_wt_techrep_ab useScore=1
chrI 86 87 . 140
chrI 89 90 . 140
chrI 100 101 . 3820
chrI 111 112 . 9480
```

wig

A wiggle (wig) file is another output from the transposon mapping pipeline. It stores similar information as the bed file, but in a different format. This file has a header typically in the form of `track type=wiggle_0 maxheightPixels=60 name=[file_name]`. Each chromosome starts with the line `variablestep chrom=chr[chromosome]` where `[chromosome]` is replaced by a chromosome name, e.g. `I` or `ref|NC_001133|`. After a variablestep line, every row corresponds with an insertion in two space delimited columns:

1. insertion position
2. number of reads

In the wig file, the read count represent the actual count (unlike the bed file where an equation is used to transform the numbers).

There is one difference between the bed and the wig file. In the bed file the insertions at the same position but with a different orientation are stored as individual insertions. In the wig file these insertions are represented as a single insertion and the corresponding read counts are added up.

Similar to the bed file, also in the wig insertions might occur that have an insertion position that is bigger then the length of the chromosome. This can be solved with the same python script as the bed file. The insertions that have a position outside the chromosome are removed and the chromosome names are represented as a roman numeral.

Example wig file:

```
track type=wiggle_0 maxheightPixels=60 name=WT_merged-techrep-a_techrep-
b_trimmed.sorted.bam
variablestep chrom=chrI
86 2
89 2
100 186
111 469
```

pergene.txt, peressential.txt

A pergene.txt and peressential.txt file are yet another outputs from the transposon mapping pipeline. Where bed and wig files store *all* insertions throughout the genome, these files only store the insertions in each gene or each essential gene, respectively. Essential genes are the annotated essential genes as stated by SGD for wild type cells. The genes are taken from the Yeast_Protein_Names.txt file, which is downloaded from uniprot. The positions of each gene are determined by a gff3 file downloaded from SGD. Essential genes are defined in Cerevisiae_AllEssentialGenes.txt.

The pergene.txt and the peressential.txt have the same format. This consists of a header and then each row contains three tab delimited columns:

1. gene name
2. total number of insertions within the gene

3. sum of all reads of those insertions

The reads are the actual read counts. To suppress noise, the insertion with the highest read count in a gene is removed from that gene. Therefore, it might occur that a gene has 1 insertion, but 0 reads.

Note that when comparing files that include gene names there might be differences in the gene naming. Genes have multiple names, e.g. systematic names like 'YBR200W' or standard names like 'BEM1' which can have aliases such as 'SRO1'. The above three names all refer to the same gene. The Yeast_Protein_Names.txt file can be used to search for aliases when comparing gene names in different files.

Example of pergene.txt file:

```
Gene name Number of transposons per gene Number of reads per gene
YAL069W 34 1819
YAL068W-A 10 599
PAU8 26 1133
YAL067W-A 12 319
```

pergene_insertions.txt, peressential_insertions.txt

The final two files that are created by the transposon mapping pipeline are the pergene_insertions.txt and the peressential_insertions.txt. The files have a similar format as the pergene.txt file, but are more extensive in terms of the information per gene. The information is taken from Yeast_Protein_Names.txt, the gff3 file and Cerevisiae_AllEssentialGenes.txt, similar as the pergene.txt files.

Both the pergene_insertions.txt and the peressential_insertions.txt files have a header and then each row contains six tab delimited columns:

1. gene name
2. chromosome where the gene is located
3. start position of the gene
4. end position of the gene
5. list of all insertions within the gene
6. list of all read counts in the same order as the insertion list

This file can be useful when not only the number of insertions is important, but also the distribution of the insertions within the genes. Similarily as the pergene.txt and peresential.txt file, to suppress noise the insertion with the highest read count in a gene is removed from that gene.

- This file is uniquely created in the processing workflow described below.
- If the index file .bam.bai is not present, create this before running the python script. The index file can be created using the command `sambamba-0.7.1.-linux-static sort -m 1GB [path]/[filename.bam]`. This creates a sorted.bam file and a sorted.bam.bai index file.

Example of peressential_insertions.txt file:

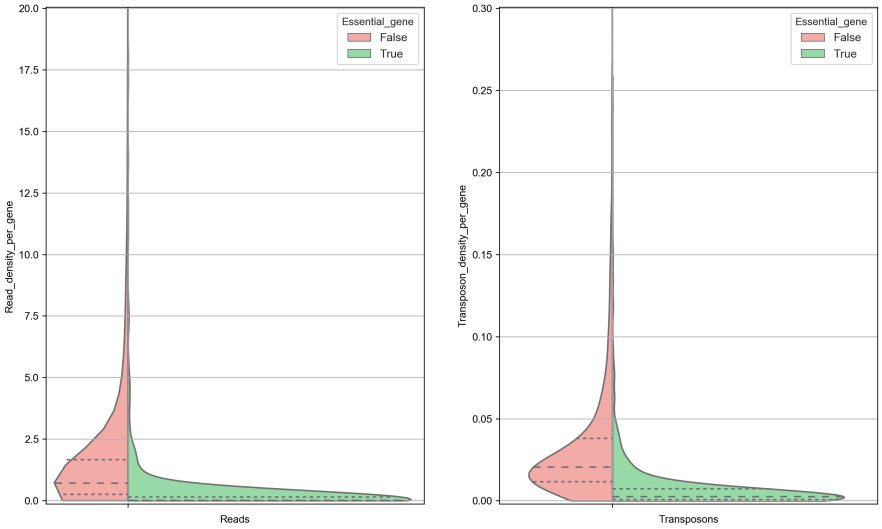
```
EFB1 I 142174 143160 [142325, 142886] [1, 1]
PRE7 II 141247 141972 [141262, 141736, 141742, 141895] [1, 1, 1, 1]
RPL32 II 45978 46370 [46011, 46142, 46240] [1, 3, 1]
```

Determine essentiality based on transposon counts

- Using the number of transposons and reads, it can be determined which genes are potentially essential and which are not.
- To check this method, the transposon count for wild type cells are determined. Currently, genes that are taken as essential are the annotated essentials based on previous research.
- We can use statitiscal learning methods to find what is the expected number of transposons per essential gene. See this Matlab Code done by one of our Master students in our lab, Wessel Teunisse.

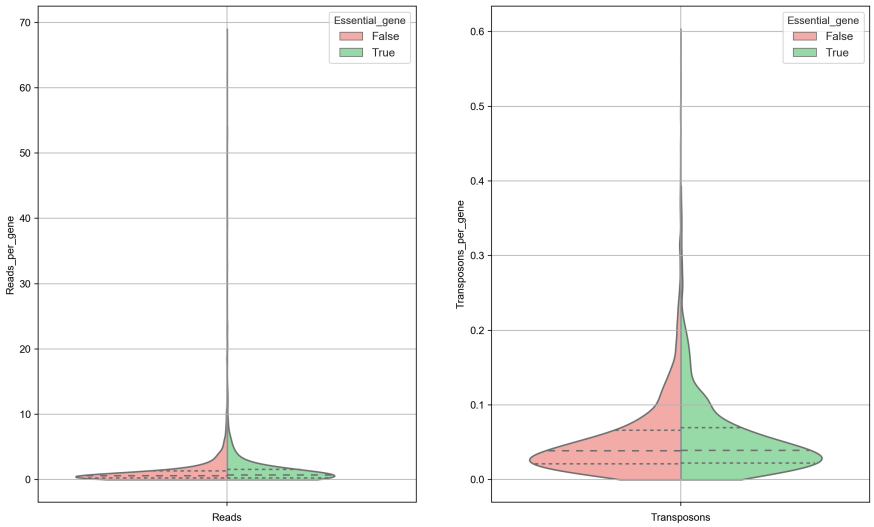
Distribution number of insertions and reads compared with essential and non-essential genes

Ideally, the number of transposon insertions of all essential genes are small and the number of insertions in non-essential genes are is large so that there is a clear distinction can be made. However, this is not always so clear. For example, the distribution of transposons in WT cells in the data from Michel et. al. looks like this:



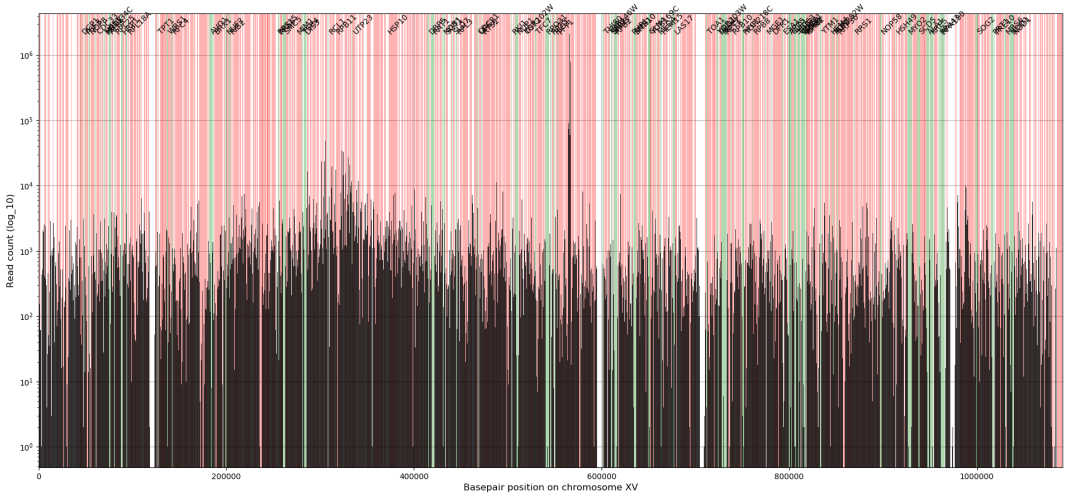
In this figure, both the reads and the transposon counts are normalized with respect to the length of each gene (hence the graph represents the read density and transposon density). High transposon counts only occurs for non-essential genes, and therefore when a high transposon count is seen, it can be assigned nonessential with reasonable certainty. However, when the transposon count is low the there is a significant overlap between the two distributions and therefore there is no certainty whether this gene is essential or not (see also the section about ‘Interpreting Transposon Counts & Reads’).

The data is also sensitive to postprocessing. It is expected that the trimming of the sequences is an important step. The graph below shows the same data as in the previous graph, but with different processing as is done by Michel et. al… This has a significant influence on the results and as a consequence, no distinction can be made between essential and nonessential genes based on the transposon counts. Significant attention needs to be given to the postprocessing of the data.



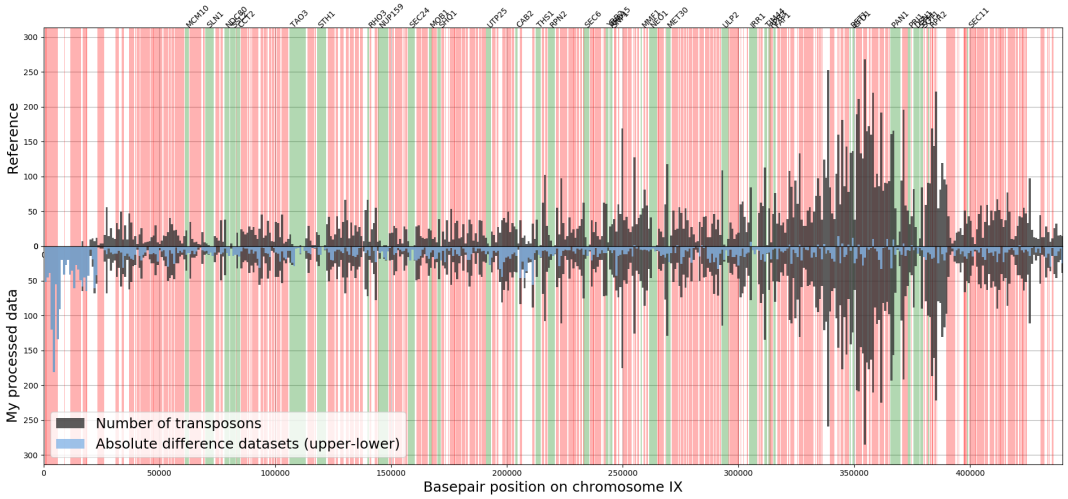
Profile plot for number of reads

To create a visual overview where the insertions are and how many reads there are for each insertion, a profile plot is created for each chromosome.



The bars indicate the absolute number of reads for all insertions located in the bars (bar width is 545bp). The colored background indicate the location of genes, where green are the annotated essential genes and red the non-essential genes. In general, the essential genes have no or little reads whereas the non-essential genes have many reads. Note that at location 564476 the ADE2 gene is located that has significant more reads than any other location in the genome, which has to do the way the plasmid is designed (see Michel et.al. 2017). The examples used here are from a dataset discussed in the paper by Michel et.al. 2017 which used centromeric plasmids where the transposons are excised from. The transposons tend to reinsert in the chromosome near the chromosomal pericentromeric region causing those regions to have about 20% more insertions compared to other chromosomal regions.

This figure gives a rough overview that allows for identifying how well the data fits the expectation. Also an alternative version of this plot is made (`TransposonRead_Profile_Compare.py`) that makes a similar plot for two datasets at the same time, allowing the comparison of the two datasets with each other and with the expectation.



How to use the pipeline in the docker container

1. Pull the docker image from Docker Hub

```
docker pull mwakok/satay:latest
```

1. Build the image and create the docker container locally in your computer

```
docker build . -t mwakok/satay:latest
```

Run the pipeline

- Move to the location where you have the data you would like to mount to the container , to use `$(pwd)` in the command bellow (simplest option) , otherwise indicate the absolute path from your computer you would like to be loaded.

```
# For Windows (and WSL):
docker run --rm -it -e DISPLAY=host.docker.internal:0 -v /$(pwd):data/ mwakok/satay:latest
```

```
# For macOS
docker run --rm -it -e DISPLAY=docker.for.mac.host.internal:0 -v $(pwd):/data mwakok/satay
```

```
# For Linux
docker run --rm -it --net=host -e DISPLAY=:0 -v $(pwd):/data mwakok/satay
```

Access the terminal of the docker container

```
# For Windows (and WSL):
docker run --rm -it -e DISPLAY=host.docker.internal:0 -v /$(pwd):data/ mwakok/satay:latest bash
```

```
# For macOS
docker run --rm -it -e DISPLAY=docker.for.mac.host.internal:0 -v $(pwd):/data mwakok/satay bash
```

```
# For Linux
docker run --rm -it --net=host -e DISPLAY=:0 -v $(pwd):/data mwakok/satay bash
```

- The flag `-e` enables viewing of the GUI outside the container via the Xserver
- The flag `-v` mounts the current directory (pwd) on the host system to the data/ folder inside the container

Creating a custom adapterfile.fa in your data/ folder

To know the adapters sequence , one way is to look for the overrepresented sequences in your dataset. Steps:

1. Run the pipeline with the
    - [x] Quality checking raw data CHECKED
    - [x] Quality check interrupt CHECKED
2. When the GUI ask you to continue , say NO and go to your local `/data/fastqc_out/`
    - Open the corresponding html file and go to the “Overrepresented sequences” section
    - Copy the sequence that has more than 15% of representation.
3. Create the adapterfile.fa in your local data folder
    - Open a bash terminal and move to the location where you have the data you would like to mount in the pipeline (fastq files)

    ```
    cd /data
    ```

    - Create the adapterfile file customized to your dataset.

    ```
    nano adapterfile.fa
    ```

    - Inside the `nano` editor , edit the file as follows:

    ```
    > \> Sequence1
    >
    > Overrepresented sequence 1
    >
    > \> Sequence2
    >
    > Overrepresented sequence 2
    ```

    - Ctrl-O save , Ctrl-X and quit the editor

> **Note**
>
> Note to not put empty lines in the text file, otherwise BBDuk might yield an error about not finding the adapters.fa file.!

4. Run again the container and it will automatically look for that file (adapterfile.fa) in the data folder .

Troubleshooting

- When running the container , mainly for the 1st time , after a reboot of your PC, this may pops up:

```
Gtk-WARNING **: cannot open display: :0
```

There is a solution in Linux is typing the following command in the terminal : `xhost +`

Code for the bash GUI

> **Click here to see the code!**

Python scripts

The software discussed in the previous section is solely for the processing of the data. The codes that are discussed here are for the postprocessing analysis. These are all python scripts that are not depended on Linux (they run and are tested in Windows) and only use rather standard python package like numpy, matplotlib, pandas, seaborn and scipy. The python version used for creating and testing is Python v3.8.5.

The order in which to run the programs shouldn't matter as these scripts are all independed of each other except for genomicfeatures_dataframe.py which is sometimes called by other scripts. However, most scripts are depending on one or more python modules, which are all expected to be located in a python_modules folder inside the folder where the python scripts are located (see github for an example how this is organized). Also many python scripts and modules are depending on data files stored in a folder called data_files located in the same folder of the python_scripts folder. The input for most scripts and modules are the output files of the processing.

This is a typical order which can be used of the scripts described below:

1. clean_bedwigfiles.py (to clean the bed and wig files).
2. transposonread_profileplot_genome.py (to check the insertion and read distribution throughout the genome).
3. transposonread_profileplot.py (to check the insertions and read distribution per chromosome in more detail).
4. scatterplot_genes.py (to check the distribution for the number of insertions per gene and per essential gene).
5. volcanoplot.py (only when comparing multiple datasets with different genetic backgrounds to see which genes have a significant change in insertion and read counts).

Most of the python scripts consists of one or more functions. These functions are called at the end of each script after the line `if __name__ == '__main__':`. The user input for these functions are stated at the beginning of the script in the `#INPUT` section. All packages where the scripts are depending on are called at the beginning of the script. The scripts also contain a help text about how to use the functions.

Cleaning bed and wig files

This script removes transposon insertions in .bed and .wig files that were mapped outside the chromosomes, creates consistent naming for chromosomes and change the header of files with custom headers.

How to use the code in the same script

```
#%%
if __name__ == '__main__':
        cleanfiles(filepath=filepath, custom_header=custom_header,
split_chromosomes=split_chromosomes)
```

Click here to see the code!

Transposon profile for the whole genome

Note

NOTE WHEN USING SPYDER3: WHEN RESCALING THE FIGURE SIZE, THE COLORCODING IN THE BARPLOT MIGHT CHANGE FOR SOME INEXPLICABLE REASON. THIS HAS NOTHING TO DO WITH THE WAY THE PYTHON CODE IS PRORAMMED, BUT RATHER DUE TO THE WAY SPYDER DISPLAYS THE PLOTS.

```
import os, sys
import numpy as np
import matplotlib.pyplot as plt

file_dirname = os.path.dirname(os.path.abspath('__file__'))
sys.path.insert(1,os.path.join(file_dirname,'python_modules'))
from chromosome_and_gene_positions import chromosome_position, gene_position
from essential_genes_names import list_known_essentials
from chromosome_names_in_files import chromosome_name_bedfile




#%%
bed_file=r""
variable="transposons" #"reads" "transposons"
bar_width=None
savefig=False

#%%
def profile_genome(bed_file=None, variable="transposons", bar_width=None, savefig=False):
    '''This function creates a bar plot along the entire genome.
    The height of each bar represents the number of transposons or reads at the genomic position
indicated on the x-axis.
    The input is as follows:
        - bed file
        - variable ('transposons' or 'reads')
        - bar_width
        - savefig

    The bar_width determines how many basepairs are put in one bin. Little basepairs per bin may be
slow. Too many basepairs in one bin and possible low transposon areas might be obscured.
    For this a list for essential genes is needed (used in 'list_known_essentials' function) and a
.gff file is required (for the functions in 'chromosome_and_gene_positions.py') and a list for gene
aliases (used in the function 'gene_aliases')
    '''


#%%
    gff_file = os.path.join(file_dirname,'..','data_files','Saccharomyces_cerevisiae.R64-1-
1.99.gff3')
    essential_genes_files =
[os.path.join(file_dirname,'..','data_files','Cerevisiae_EssentialGenes_List_1.txt'),

os.path.join(file_dirname,'..','data_files','Cerevisiae_EssentialGenes_List_2.txt')]


    chrom_list = ['I', 'II', 'III', 'IV', 'V', 'VI', 'VII', 'VIII', 'IX', 'X', 'XI', 'XII', 'XIII',
'XIV', 'XV', 'XVI']

    chr_length_dict, chr_start_pos_dict, chr_end_pos_dict = chromosome_position(gff_file)


    summed_chr_length_dict = {}
    summed_chr_length = 0
    for c in chrom_list:
        summed_chr_length_dict[c] = summed_chr_length
        summed_chr_length += chr_length_dict.get(c)


    l_genome = 0
    for chrom in chrom_list:
        l_genome += int(chr_length_dict.get(chrom))
    print('Genome length: ',l_genome)
    if bar_width == None:
        bar_width = l_genome/1000


    middle_chr_position = []
    c1 = summed_chr_length_dict.get('I')
    for c in summed_chr_length_dict:
        if not c == 'I':
            c2 = summed_chr_length_dict.get(c)
            middle_chr_position.append(c1 + (c2 - c1)/2)
            c1 = c2
    c2 = l_genome
    middle_chr_position.append(c1 + (c2 - c1)/2)


    gene_pos_dict = gene_position(gff_file)
    genes_currentchrom_pos_list = [k for k, v in gene_pos_dict.items()]
    genes_essential_list = list_known_essentials(essential_genes_files)
```

```
    with open(bed_file) as f:
        lines = f.readlines()


    chrom_names_dict, chrom_start_index_dict, chrom_end_index_dict= chromosome_name_bedfile(lines)

    allcounts_list = np.zeros(l_genome)
    if variable == "transposons":
        for line in lines[chrom_start_index_dict.get("I"):chrom_end_index_dict.get("XVI")+1]:
            line = line.strip('\n').split()
            chrom_name = [k for k,v in chrom_names_dict.items() if v == line[0].replace("chr",'')]
[0]
            allcounts_list[summed_chr_length_dict.get(chrom_name) + int(line[1])-1] += 1
    elif variable == "reads":
        for line in lines[chrom_start_index_dict.get("I"):chrom_end_index_dict.get("XVI")+1]:
            line = line.strip('\n').split()
            chrom_name = [k for k,v in chrom_names_dict.items() if v == line[0].replace("chr",'')]
[0]
            allcounts_list[summed_chr_length_dict.get(chrom_name) + int(line[1])-1] +=
(int(line[4])-100)/20


    allcounts_binnedlist = []
    val_counter = 0
    sum_values = 0
    for n in range(len(allcounts_list)):
        if int(val_counter % bar_width) != 0:
            sum_values += allcounts_list[n]
        elif int(val_counter % bar_width) == 0:
            allcounts_binnedlist.append(sum_values)
            sum_values = 0
        val_counter += 1
    allcounts_binnedlist.append(sum_values)


    if bar_width == (l_genome/1000):
        allinsertionsites_list = np.linspace(0,l_genome,int(l_genome/bar_width+1))
    else:
        allinsertionsites_list = np.linspace(0,l_genome,int(l_genome/bar_width+2))



    plt.figure(figsize=(19.0,9.0))#(27.0,3))
    grid = plt.GridSpec(20, 1, wspace=0.0, hspace=0.0)

    textsize = 12
    textcolor = "#000000"
    binsize = bar_width
    ax = plt.subplot(grid[0:19,0])

    ax.bar(allinsertionsites_list,allcounts_binnedlist,width=binsize,color="#333333")
    ax.grid(False)
    ax.set_xlim(0,l_genome)

    for chrom in summed_chr_length_dict:
        ax.axvline(x = summed_chr_length_dict.get(chrom), linestyle='-', color=(0.9,0.9,0.9,1.0))

    ax.set_xticks(middle_chr_position)
    ax.set_xticklabels(chrom_list, fontsize=textsize)
    ax.tick_params(axis='x', which='major', pad=30)
    if variable == "transposons":
        plt.ylabel('Transposon Count', fontsize=textsize, color=textcolor)#, labelpad=30)
    elif variable == "reads":
        plt.ylabel('Read Count', fontsize=textsize, color=textcolor)#, labelpad=30)

    axc = plt.subplot(grid[19,0])
    for gene in genes_currentchrom_pos_list:
        if not gene_pos_dict.get(gene)[0] == 'Mito':
            gene_start_pos = summed_chr_length_dict.get(gene_pos_dict.get(gene)[0]) +
int(gene_pos_dict.get(gene)[1])
            gene_end_pos = summed_chr_length_dict.get(gene_pos_dict.get(gene)[0]) +
int(gene_pos_dict.get(gene)[2])
            if gene in genes_essential_list:
                axc.axvspan(gene_start_pos,gene_end_pos,facecolor="#00F28E",alpha=0.8)
            else:
                axc.axvspan(gene_start_pos,gene_end_pos,facecolor="#F20064",alpha=0.8)
    axc.set_xlim(0,l_genome)
    axc.tick_params(
        axis='x',          # changes apply to the x-axis
        which='both',      # both major and minor ticks are affected
        bottom=False,      # ticks along the bottom edge are off
        top=False,         # ticks along the top edge are off
        labelbottom=False) # labels along the bottom edge are off

    axc.tick_params(
        axis='y',          # changes apply to the y-axis
        which='both',      # both major and minor ticks are affected
```

```
        left=False,         # ticks along the bottom edge are off
        right=False,        # ticks along the top edge are off
        labelleft=False)    # labels along the bottom edge are off


    if savefig == True and variable == "transposons":
        savepath = os.path.splitext(bed_file)
        print('saving figure at %s' % savepath[0]+'_transposonplot_genome.png')
        plt.savefig(savepath[0]+'_transposonplot_genome.png', dpi=400)
        plt.close()
    elif savefig == True and variable == "reads":
        savepath = os.path.splitext(bed_file)
        print('saving figure at %s' % savepath[0]+'_readplot_genome.png')
        plt.savefig(savepath[0]+'_readplot_genome.png', dpi=400)
        plt.close()
    else:
        plt.show()
```

How to use the code in the same script

```
if __name__ == '__main__':
    profile_genome(bed_file=bed_file, variable=variable, bar_width=bar_width, savefig=savefig)
```

Transposon profile for a chromosome of interest

Click here to see the code!

How to use the code in the same script

```
if __name__ == '__main__':
    profile_plot(bed_file=bed_file, variable=variable, chrom=chrom, bar_width=bar_width,
savefig=savefig)
```

Create a scatterplot for all genes and all essential genes.

Click here to see the code!

How to use the code in the same script

```
if __name__ == '__main__':
    read_gene_df = scatterplot(pergenefile)
```

Volcano plots to two different libraries

This script creates a volcanoplot to show the significance of fold change between two datasets.It is based on this website: - https://towardsdatascience.com/inferential-statistics-series-t-test-using-numpy-2718f8f9bf2f - https://www.statisticshowto.com/independent-samples-t-test/

Code for showing gene name when hovering over datapoint is based on: - https://stackoverflow.com/questions/7908636/possible-to-make-labels-appear-when-hovering-over-a-point-in-matplotlib

T-test is measuring the number of standard deviations our measured mean is from the baseline mean, while taking into account that the standard deviation of the mean can change as we get more data. Look here for more information:

- Independent T-test : https://www.statisticshowto.com/independent-samples-t-test/

Click here to see the code!

How to use the code in the same script

```
if __name__ == '__main__':
    volcano_df = volcano(path_a=path_a, filelist_a=filelist_a,
            path_b=path_b, filelist_b=filelist_b,
            variable=variable,
            significance_threshold=significance_threshold,
            normalize=normalize,
            trackgene_list=trackgene_list,
            figure_title=figure_title)
```

Searching for genomic features of transposition events

This scripts takes a user defined genomic region (i.e. chromosome number, region or gene) and creates a dataframe including information about all genomic features in the chromosome (i.e. genes, nc-DNA etc.). This can be used to determine the number of reads outside the genes to use this for normalization of the number of reads in the genes. To run this script, the following files are required and should be placed in the same directory as this file: - chromosome_and_gene_positions.py - chromosome_names_in_files.py - gene_names.py - read_sgdfeatures.py - normalize_reads.py

For the ipython Notebook version of this script with more extensive explanation, see here

This script consists of two functions, `dna_features` and `feature_position`. The function `dna_features` is the main function that takes user inputs and this function calls the function `feature_position` which is not intended to be used directly. The function `dna_features` takes the following arguments:

`region=[int]||[string]||[list]` (required)
`wig_file=[path]` (required)
`pergene_insertions_file=[path]` (required)
`variable="reads"||"insertions"`
`plotting=True||False`
`savefigure=True||False`
`verbose=True||False`

The script takes a wig file and a pergene_insertions_file and a genomic region. This genomic region can be:

- chromosome number (either roman numeral or integer between 1 and 16)
- a list with three arguments: first a number defining the chromosome (again either roman numeral or integer between 1 and 16) second an integer defining the start basepair and third an integer defining an end basepair (e.g. [“I”, 10000, 20000] to get the region between basepair 10000 and 20000 on chromosome 1)
- a gene name (e.g. “CDC42”) which will automatically get the corresponding chromosome and basepair position

The `plotting` argument (True or False) defines whether to create a barplot. The `variable` argument determines what to plot, either reads or insertions and the `savefigure` whether to automatically save the figure at the same location as where this script is stored. Finally the `verbose` determines if any printing output should be given (this is mostly useful for when calling this script from other python scripts).

This scripts does not only look at genes, but also at other genomic regions like telomere, centromeres, rna genes etc. All these features are stored in one dataframe called `dna_df2` that includes naming and positional information about the features and the insertion and read counts (see output). The dataframe will always be created for one entire chromosome (regardless if a basepair region or gene name was entered in the `region` argument). When the plotting is set to the True, it will also create a barplot for the same chromosome within the region that is defined in the `region` variable. The plot distinguishes between nonessential genes, essential genes, other genomic features (e.g. telomeres, centromeres etc.) and noncoding dna. The width of the bars is determined by the length of the genomic feature and the height represents either the number of reads or insertions (depending what is defined in `variable`).

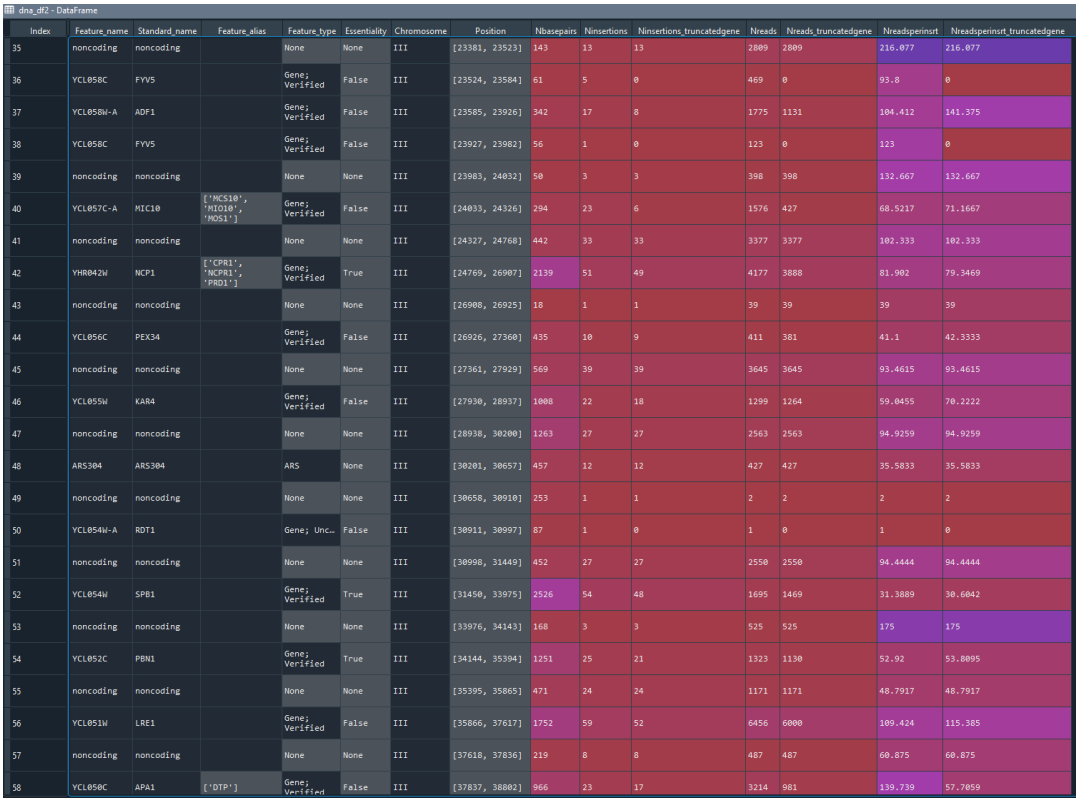
This function can be useful for other python functions as well when information is required about the positions, insertions and read counts of various genomic features. The full list of genomic features that is regarded in the dataframe is mentioned in read_sgdfeatures.py.

- **Output**

The main output is the `dna_df2` dataframe in which each row represents a genomic feature and has the following columns:

- Feature name
- Feature standard name
- Feature aliases (different names for the same feature, mainly for genes)
- Essentialilty (only for genes)
- Chromosome where feature is located
- Position in terms in basepairs
- Length of feature in terms of basepairs
- Number of insertions in feature
- Number of insertions in truncated feature (only for genes, the insertions in the first and last 100bp are ignored, see below)
- Sum of reads in feature
- Sum of reads in truncated feature (only for genes, where the reads in the first and last 100bp are ignored)
- Number of reads per insertion
- Number of reads per insertion (only for genes, where the insertions and reads in the first and last 100bp are ignored)

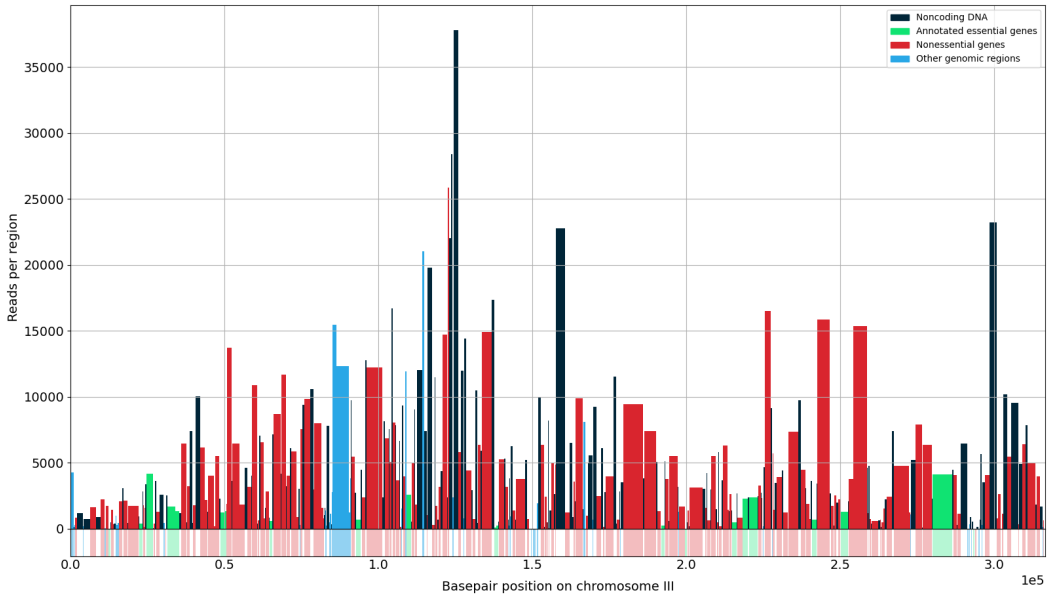
The truncated feature columns ignores basepairs at the beginning and end of a gene. This can be useful as it is mentioned that insertions located at the beginning or end of a gene results in a protein that is still functional (although truncated) (e.g. see Michel et.al. 2017) (see Notes for a further discussion about how this is defined).



dna_df2 - DataFrame

Index	Feature_name	Standard_name	Feature_alias	Feature_type	Essentiality	Chromosome	Position	Nbasepairs	Ninsertions	Ninsertions_truncatedgene	Nreads	Nreads_truncatedgene	Nreadsperinsrt	Nreadsperinsrt_truncatedgene
35	noncoding	noncoding		None	None	III	[23381, 23523]	143	13	13	2809	2809	216.077	216.077
36	YCL058C	FYV5		Gene; Verified	False	III	[23524, 23584]	61	5	0	469	0	93.8	0
37	YCL058W-A	ADF1		Gene; Verified	False	III	[23585, 23926]	342	17	8	1775	1131	104.412	141.375
38	YCL058C	FYV5		Gene; Verified	False	III	[23927, 23982]	56	1	0	123	0	123	0
39	noncoding	noncoding		None	None	III	[23983, 24032]	50	3	3	398	398	132.667	132.667
40	YCL057C-A	MIC10	['MCS10', 'MIO10', 'MOS1']	Gene; Verified	False	III	[24033, 24326]	294	23	6	1576	427	68.5217	71.1667
41	noncoding	noncoding		None	None	III	[24327, 24768]	442	33	33	3377	3377	102.333	102.333
42	YHR042W	NCP1	['CPR1', 'NCPR1', 'PRD1']	Gene; Verified	True	III	[24769, 26907]	2139	51	49	4177	3888	81.902	79.3469
43	noncoding	noncoding		None	None	III	[26908, 26925]	18	1	1	39	39	39	39
44	YCL056C	PEX34		Gene; Verified	False	III	[26926, 27360]	435	10	9	411	381	41.1	42.3333
45	noncoding	noncoding		None	None	III	[27361, 27929]	569	39	39	3645	3645	93.4615	93.4615
46	YCL055W	KAR4		Gene; Verified	False	III	[27930, 28937]	1008	22	18	1299	1264	59.0455	70.2222
47	noncoding	noncoding		None	None	III	[28938, 30200]	1263	27	27	2563	2563	94.9259	94.9259
48	ARS304	ARS304		ARS	None	III	[30201, 30657]	457	12	12	427	427	35.5833	35.5833
49	noncoding	noncoding		None	None	III	[30658, 30910]	253	1	1	2	2	2	2
50	YCL054W-A	RDT1		Gene; Unc…	False	III	[30911, 30997]	87	1	0	1	0	1	0
51	noncoding	noncoding		None	None	III	[30998, 31449]	452	27	27	2550	2550	94.4444	94.4444
52	YCL054W	SPB1		Gene; Verified	True	III	[31450, 33975]	2526	54	48	1695	1469	31.3889	30.6042
53	noncoding	noncoding		None	None	III	[33976, 34143]	168	3	3	525	525	175	175
54	YCL052C	PBN1		Gene; Verified	True	III	[34144, 35394]	1251	25	21	1323	1130	52.92	53.8095
55	noncoding	noncoding		None	None	III	[35395, 35865]	471	24	24	1171	1171	48.7917	48.7917
56	YCL051W	LRE1		Gene; Verified	False	III	[35866, 37617]	1752	59	52	6456	6000	109.424	115.385
57	noncoding	noncoding		None	None	III	[37618, 37836]	219	8	8	487	487	60.875	60.875
58	YCL050C	APA1	['DTP']	Gene; Verified	False	III	[37837, 38802]	966	23	17	3214	981	139.739	57.7059

When plotting is set to True, a barplot is created where the width of the bars correspond to the width of the feature the bar is representing. This can be automatically saved at the location where the python script is stored. The plot is created for an entire chromosome, or it can be created for a specific region, for example when a list is provided in the `region` argument or when a gene name is given.



Note

- The definition for a truncated gene is currently that the first and last 100bp are ignored. This is not completely fair as this is much more stringent for short genes compared to long genes. Alternatively this can be changed to a percentage, for example ignoring the first and last 5 percent of a gene, but this can create large regions in long genes. There is no option to set this directly in the script, but if this needs to be changed, search the script for the following line: `#TRUNCATED GENE DEFINITION`. This should give the `N10percent` variable that controls the definition of a truncated gene.
- The barplot currently only takes reads or insertions, but it might be useful to include reads per insertions as well.
- Sometimes it can happen that two genomic regions can overlap. When this happens, the dataframe shows a feature, in the next row another feature and then the next row from that it continues with the first feature (e.g. row1; geneA, row2; overlapping feature, row3; geneA). This issue is not automatically solved yet, but best is to, whenever you find a feature, to search if that feature also exists a couple of rows further on. If yes, sum all rows between the first and last occurance of your gene (e.g. in the example above for geneA, sum the values from row1, row2 and row3).

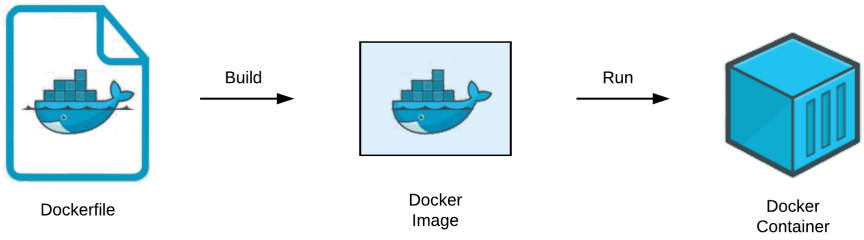
Click here to see the code!

How to use the code in the same script

```
region = 1 #e.g. 1, "I", ["I", 0, 10000"], gene name (e.g. "CDC42")
wig_file = r""
pergene_insertions_file = r""
plotting=True
variable="reads" #"reads" or "insertions"
savefigure=False
verbose=True
if __name__ == '__main__':
    dna_df2 = dna_features(region=region,
                wig_file=wig_file,
                pergene_insertions_file=pergene_insertions_file,
                variable=variable,
                plotting=plotting,
                savefigure=savefigure,
                verbose=verbose)
```

Build your own docker container with ours satay pipeline

Docker is an open platform for developing, shipping, and running applications. Docker provides the ability to package and run an application in a loosely isolated environment called a container. Containers are lightweight and contain everything needed to run the application, so you do not need to rely on what is currently installed on the host system. You can easily share containers while you work, and be sure that everyone you share with gets the same container that works in the same way.



Dockerfile – is a text document that contains all the commands you would normally execute manually in order to build a Docker image. The instructions include a choice of operating system and all the libraries we need to install into it. Docker can build images automatically by reading the instructions from a Dockerfile.
Docker Images – are the basis of containers. A Docker image is an immutable (unchangeable) file that contains the source code, libraries, dependencies, tools, and other files needed for an application to run.
Docker Container – A container is, ultimately, just a running image.

Docker installation

Docker can be installed on Windows, macOS, and Linux. Please visit the Docker website for downloading and installation instructions. Note, you will need admin access to your system.

Verify Docker installation

Run the following commands in the terminal (see below) to verify your installation:

- `docker --version`
  Will output the version number
- `docker run hello-world`
  Will output a welcome message. If you haven't run this command before, you will receive the message *Unable to find image: 'hello-world:latest' locally*. Docker will then proceed by downloading and running the latest version from DockerHub.

TL;DR

1. Start your Xserver
2. Open your (bash) terminal and use the following command to start a container from the satay image, mounting the current directory `$(pwd)` as the folder `/data` inside the container.

Windows (and WSL): `docker run --rm -it -e DISPLAY=host.docker.internal:0 -v /$(pwd):/data mwakok/satay`
Linux: `docker run --rm -it --net=host -e DISPLAY=:0 -v $(pwd):/data mwakok/satay`
macOS: `docker run --rm -it -e DISPLAY=docker.for.mac.host.internal:0 -v $(pwd):/data mwakok/satay`

Terminal access

Linux
The default Unix Shell for Linux operating systems is usually Bash. On most versions of Linux, it is accessible by running the (Gnome) Terminal or (KDE) Konsole or xterm, which can be found via the applications menu or the search bar. If your machine is set up to use something other than bash, you can run it by opening a terminal and typing `bash`.

macOS
For a Mac computer, the default Unix Shell is Bash, and it is available via the Terminal Utilities program within your Applications folder. To open Terminal, try one or both of the following:

- Go to your Applications. Within Applications, open the Utilities folder. Locate Terminal in the Utilities folder and open it.
- Use the Mac 'Spotlight' computer search function. Search for: Terminal and press Return.

For more info: How To use a terminal on Mac

Windows
Computers with Windows operating systems do not automatically have a Unix Shell program installed. We encourage you to use an emulator included in Git for Windows, which gives you access to both Bash shell commands and Git. To install, please follow these instructions.

X Windows System

Docker doesn't have any build-in graphics, which means it cannot run desktop applications (such as the SATAY GUI) by default. For this, we require the X Windows System. The X Window System (X11, or simply X) is a windowing system for bitmap displays, common on Unix-like operating systems. X provides the basic framework for a GUI environment: drawing and moving windows on the display device and interacting with a mouse and a keyboard.

If you are on a desktop Linux, you already have one. For macOS, you can download XQuartz, and for Windows, we tested VcXsrv.

Desktop applications will run in Docker and will try to communicate with the X server you're running on your PC. They don't need to know anything but the location of the X server and an optional display that they target. This is denoted by an environmental variable named `DISPLAY`, with the following syntax: `DISPLAY=xserver-host:0`. The number you see after the `:` is the display number; for the intents and purpose of this article, we will consider this to be equivalent to "0 is the primary display attached to the X server."

In order to set up the environment variable, we need to add the following code to the `docker run` command in the terminal:

For macOS: `-e DISPLAY=docker.for.mac.host.internal:0`
For Windows: `-e DISPLAY=host.docker.internal:0`
For Linux: `--net=host -e DISPLAY=:0`

With these commands (and an active X server on the host system), any graphical output inside the container will be shown on your own desktop.

Mount a volume

The docker image with which you can spawn a container contains all the software and general datafiles. However, we still need to give the container access to your dataset. To do so, we can mount a directory on your own system inside the container with the following command structure: `-v <abs_path_host>:<abs_path_container>`. Assuming your terminal

is opened inside the data folder on your system, the specific commands for the different operating systems mount this folder as the `/data` folder inside the container, are:

For Windows in GitBash: `-v /$(pwd):/data`
For Windows in cmd: `-v %cd%:/data`
For Linux and macOS: `-v $(pwd):/data`

`$(pwd)` can be replaced with the absolute path of the datafolder, or be used to access subdirectories (e.g. `$(pwd)/data:/data`).

For more info about mounting volumes, check this StackOverflow question

Running a satay container

To start a container from an image, we use the command `docker run <image_name>`. We also pass the additional flags `--rm` to delete the container after closing and `-it` to be able to interact with the container. Combining all arguments then leads to the following commands to run (and automatically close) the `satay` container:

Windows (and WSL): `docker run --rm -it -e DISPLAY=host.docker.internal:0 -v /$(pwd):/data mwakok/satay`
Linux: `docker run --rm -it --net=host -e DISPLAY=:0 -v $(pwd):/data mwakok/satay`
macOS: `docker run --rm -it -e DISPLAY=docker.for.mac.host.internal:0 -v $(pwd):/data mwakok/satay`

Issues

For Linux users encountering the error *Unable to init server*, please run `xhost +` in the terminal and rerun the `docker run` command.

References

- https://betterprogramming.pub/running-desktop-apps-in-docker-43a70a5265c4
- https://coderefinery.github.io/installation/shell-and-git/#shell-and-git
- https://ucsbcarpentry.github.io/2019-10-24-gitbash/setup.html

Contributing guidelines for developers

We welcome any kind of contribution to our software, from simple comment or question to a full fledged pull request. Please note we have a Code of Conduct, please follow it in all your interactions with the project.

A contribution can be one of the following cases:

1. you have a question;
2. you think you may have found a bug (including unexpected behavior);
3. you want to make some kind of change to the code base (e.g. to fix a bug, to add a new feature, to update documentation);
4. you want to make a new release of the code base.

The sections below outline the steps in each case.

You have a question

1. use the search functionality here to see if someone already filed the same issue;
2. if your issue search did not yield any relevant results, make a new issue;
3. apply the “Question” label; apply other labels when relevant.

You think you may have found a bug

1. use the search functionality here to see if someone already filed the same issue;
2. if your issue search did not yield any relevant results, make a new issue, making sure to provide enough information to the rest of the community to understand the cause and context of the problem. Depending on the issue, you may want to include:

    - the SHA hashcode of the commit that is causing your problem;
    - some identifying information (name and version number) for dependencies you're using;
    - information about the operating system;
3. apply relevant labels to the newly created issue.

You want to make some kind of change to the code base

1. (**important**) announce your plan to the rest of the community *before you start working*. This announcement should be in the form of a (new) issue;
2. (**important**) wait until some kind of consensus is reached about your idea being a good idea;
3. if needed, fork the repository to your own Github profile and create your own feature branch off of the latest master commit. While working on your feature branch, make sure to stay up to date with the master branch by pulling in changes, possibly from the 'upstream' repository (follow the instructions here and here);
4. make sure the existing tests still work by running `pytest`;
5. add your own tests (if necessary);
6. update or expand the documentation;
7. update the `CHANGELOG.md` file with change;
8. push your feature branch to (your fork of) the transposonmapper repository on GitHub;
9. create the pull request, e.g. following the instructions here.

In case you feel like you've made a valuable contribution, but you don't know how to write or run tests for it, or how to generate the documentation: don't let this discourage you from making the pull request; we can help you! Just go ahead and submit the pull request, but keep in mind that you might be asked to append additional commits to your pull request.

You want to make a new release of the code base

To create release you need write permission on the repository.

1. Check author list in `citation.cff` and `.zenodo.json` files
2. Bump the version using `bump2version <major|minor|patch>`. For example, `bump2version major` will increase major version numbers everywhere its needed (code, meta, etc.) in the repo.
3. Update the `CHANGELOG.md` to include changes made
4. Goto GitHub release page
5. Press draft a new release button
6. Fill version, title and description field
7. Press the Publish Release button

Code of Conduct

Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, nationality, personal appearance, race, religion, or sexual identity and orientation.

Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks

- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at L.M.InigoDeLaCruz@tudelft.nl. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at http://contributor-covenant.org/version/1/4

Tutorials

Tutorial-1 : How to run the transposonmapper python package in a specific environment.

Installing python package for users

```
git clone https://github.com/SATAY-LL/Transposonmapper.git Transposonmapper
cd Transposonmapper
conda env create --file conda/environment.yml
conda activate satay
pip install -e .
```

Installing python package for developers

```
git clone https://github.com/SATAY-LL/Transposonmapper.git Transposonmapper
cd Transposonmapper
conda env create --file conda/environment-dev.yml
conda activate satay-dev
pip install -e .[dev]
```

Importing the required python libraries

```
import os, sys
import warnings
import timeit
import numpy as np
import pysam
import pandas as pd

# importing the transposon mapping function

from transposonmapper  import transposonmapper
```

Invoking the transposonmapper package with a dummy file

```
bamfile= '../transposonmapper/data_files/files4test/SRR062634.filt_trimmed.sorted.bam'
filename='SRR062634.filt_trimmed.sorted.bam'
assert os.path.isfile(bamfile), "Not a file or directoy"

transposonmapper(bamfile=bamfile)
```

Tutorial-2 : How to clean the wig and bed files.

After following the steps in Tutorial-1 : How to run the transposonmapper python package in a specific environment.

Here we will remove transposon insertions in .bed and .wig files that were mapped outside the chromosomes, creates consistent naming for chromosomes and change the header of files with custom headers.

Import the function

```
from transposonmapper.processing.clean_bedwigfiles import cleanfiles
```

Lets save the wig and bed files as variables to clean them and call the function.

Lets use our dummy files that were outputed after running transposonmapper in Tutorial-1 : How to run the transposonmapper python package in a specific environment.

```
wig_files=[]
bed_files=[]

#data_dir= "../satay/data_files/data_unmerged/"
data_dir="../transposonmapper/data_files/files4test/"
for root, dirs, files in os.walk(data_dir):
    for file in files:
        if file.endswith("sorted.bam.wig"):
            wig_files.append(os.path.join(root, file))
        elif file.endswith("sorted.bam.bed"):
             bed_files.append(os.path.join(root, file))
```

Cleaning the files

What for?

Clean wig files for proper visualization in the genome Browser http://genome-euro.ucsc.edu/cgi-bin/hgGateway

```
custom_header = ""
split_chromosomes = False
for files in zip(wig_files,bed_files):
    cleanfiles(filepath=files[0], custom_header=custom_header, split_chromosomes=split_chromosomes)
    cleanfiles(filepath=files[1], custom_header=custom_header, split_chromosomes=split_chromosomes)
```

Tutorial-3: Visualize the insertions and reads per gene throughout the genome

After following the steps in Tutorial-2 : How to clean the wig and bed files. we have proper clean files to continue our analysis.

Import the function

```
from transposonmapper.processing.transposonread_profileplot_genome import profile_genome
```

Lets save the cleaned files as variables to clean them and call the function.

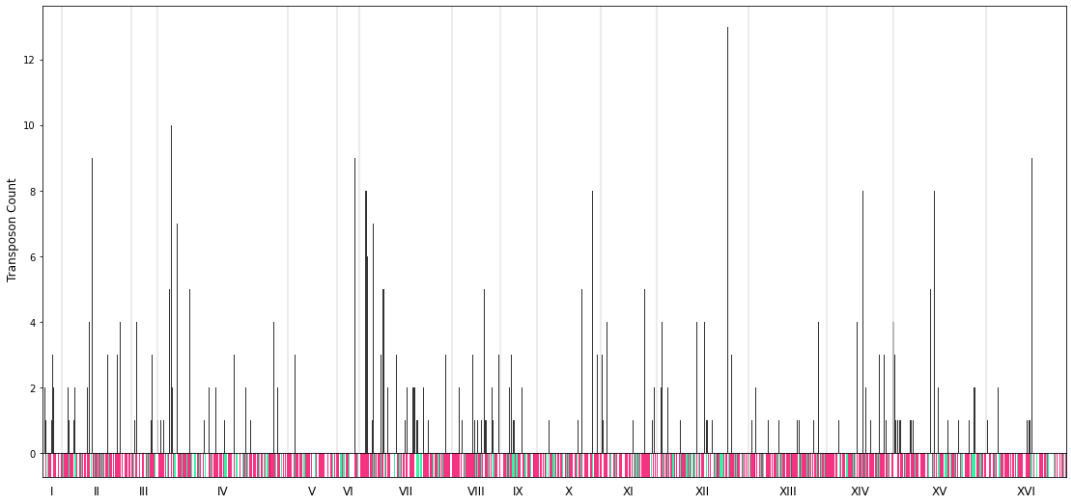
Lets use our dummy files that were outputed after running transposonmapper in Tutorial-2 : How to clean the wig and bed files.

```
cleanbed_files=[]
for root, dirs, files in os.walk(data_dir):
    for file in files:
        if file.endswith("clean.bed"):
            cleanbed_files.append(os.path.join(root, file))

cleanwig_files=[]
for root, dirs, files in os.walk(data_dir):
    for file in files:
        if file.endswith("clean.wig"):
            cleanwig_files.append(os.path.join(root, file))
```

Vizualization

```
bed_file=cleanbed_files[0] # example for the 1st file
variable="transposons" #"reads" "transposons"
bar_width=None
savefig=False

profile=profile_genome(bed_file=bed_file, variable=variable, bar_width=bar_width,
savefig=savefig,showfig=True)
```

This is the plot for the case of the dummy sample files.



Tutorial-4: Zoom in into the chromosomes

Here we will use the files generated in Tutorial-2 : How to clean the wig and bed files..

Import the function

```
from transposonmapper.processing.genomicfeatures_dataframe import dna_features
```

Getting the pergene file

```
pergene_files=[]

data_dir="../transposonmapper/data_files/files4test/"
for root, dirs, files in os.walk(data_dir):
    for file in files:
        if file.endswith('sorted.bam_pergene_insertions.txt'):
            pergene_files.append(os.path.join(root, file))
```

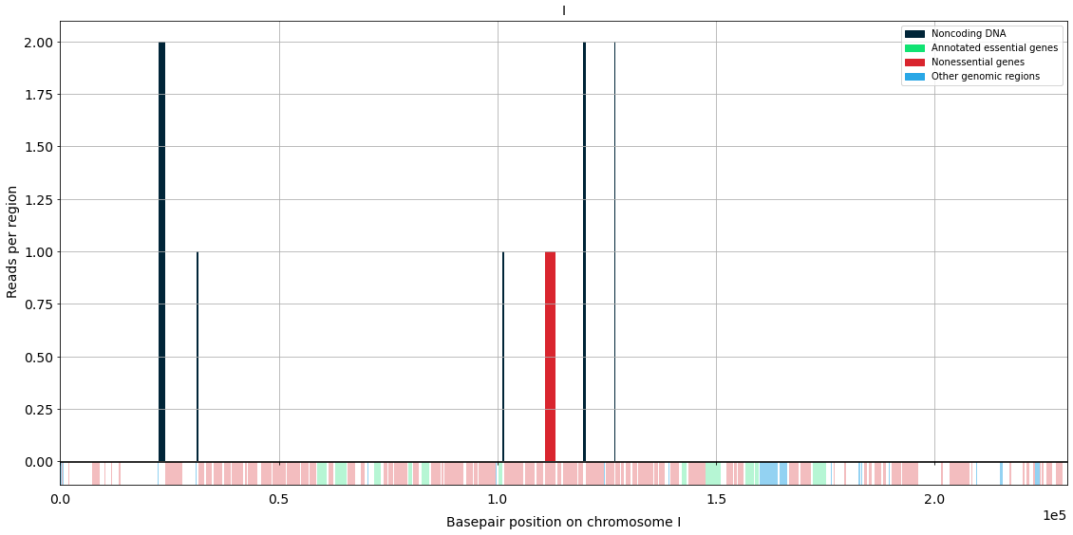
Vizualization

```
wig_file = cleanwig_files[0]
pergene_insertions_file = pergene_files[0]
plotting=True
variable="reads" #"reads" or "insertions"
savefigure=False
verbose=True


region = "I" #e.g. 1, "I", ["I", 0, 10000"], gene name (e.g. "CDC42")
dna_features(region=region,
                wig_file=wig_file,
                pergene_insertions_file=pergene_insertions_file,
                variable=variable,
                plotting=plotting,
                savefigure=savefigure,
                verbose=verbose)
```

This will create a dataframe with the following columns per region:

```
Feature_name
Standard_name
Feature_alias
Feature_type
Essentiality
Chromosome
Position
Nbasepairs
Ninsertions
Ninsertions_truncatedgene
Nreads
Nreads_list
Nreads_truncatedgene
Nreadsperinsrt
Nreadsperinsrt_truncatedgene
```

This is the plot for the case of the dummy sample files for chromosome I.



Tutorial-5: Volcano plots

**Do you want to compare two differente libraries to discover which genes stood out from their comoparison? **

Then do volcano plots!

Import the function

```
from transposonmapper.statistics import volcano
```

Getting the volcano plot

Look at the help of this function , HERE

```
path_a = r""
filelist_a = ["",""]
path_b = r""
filelist_b = ["",""]


variable = 'read_per_gene' #'read_per_gene' 'tn_per_gene', 'Nreadsperinsrt'
significance_threshold = 0.01 #set threshold above which p-values are regarded significant
normalize=True

trackgene_list = ['my-favorite-gene'] # ["cdc42"]


figure_title = " "

volcano_df = volcano(path_a=path_a, filelist_a=filelist_a,
            path_b=path_b, filelist_b=filelist_b,
            variable=variable,
            significance_threshold=significance_threshold,
            normalize=normalize,
            trackgene_list=trackgene_list,
            figure_title=figure_title)
```

This is a volcano plot made with real data!

By Gregory van Beek , Maurits Kok, Leila Iñigo de la Cruz