Documentation for processing SATAY

This documentation gives a complete overview for the processing of the data from SAturated Transposon Analysis in Yeast (SATAY). It includes a short introduction to SATAY and a detailed discussion on to perform the processing from the raw sequencing data to the postprocessing and checking of the results.

For the processing a pipeline is created using Bash and Python. The workflow and the python codes can be found at github.com/leilaicruz/LaanLab-SATAY-DataAnalysis. More information about satay analysis and experimental protocols can be found at the satayusers website from the Kornmann lab or, for more questions, visit the satayusers forum.

> Date last update: 26-03-2021
>
> Author: Gregory van Beek
>
> Contact: Leila Inigo De La Cruz (L.M.InigoDeLaCruz@tudelft.nl)
>
> Laanlab, Delft University of Technology

Introduction

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.

File types

During the processing, different file types are being used and created. This section gives an overview of all those files, how to implement them and when to use which file.

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(P) 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
+
AAAAAEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
```

```
@NB501605:544:HLHLMBGXF:1:11101:2258:1051 1:N:0:TACAGCTA
TGAGGCACCTATCTCAGCGATCGTATCGGTTTTCGATTACCGTATTTATCCCGTTCGTTTTCGTTGCCGCTATTT
+
AAAAAEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE6EEEEEEEE<EEEAEEEEEEE/E/E/EA///
@NB501605:544:HLHLMBGXF:1:11101:26723:1052 1:N:0:TGCAGCTA
TGTCAACGGTTTAGTGTTTTCTTACCCAATTGTAGAGACTATCCACAAGGACAATATTTGTGACTTATGTTATGCG
+
AAAAAEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
```

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/EEEEEAEAEEEEEEEEEEEEEEEEEEEEAAAAA 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 EEEEEEEEEEE6EEEEEEEEEEEEEE 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, or the genomicfeatures_dataframe.py python script can be used which creates a pandas dataframe that includes the different gene names (this python script itself makes also use of the Yeast_Protein_Names.txt file).

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.

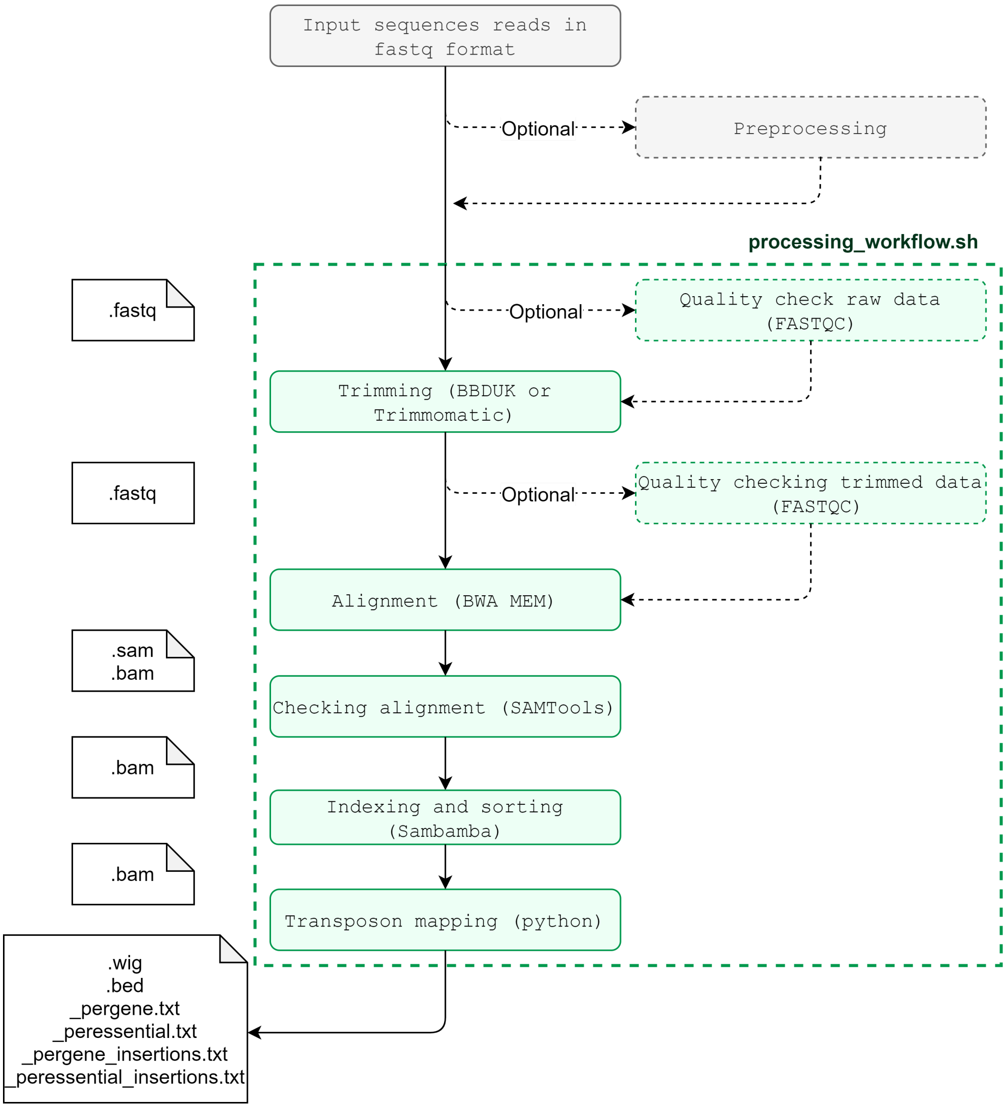
This file is uniquely created in the processing workflow described below. To create this file from a dataset processed in another workflow, store the bam file and the corresponding .bam.bai index file on the Linux desktop (see How to use the Linux desktop). Go to the python folder in the Terminal with the following command: `cd /home/laanlab/satay/Documents/software/python_codes/`. Run the transposonmapping_satay.py script with the bam file using the command `python3 transposonmapping_satay.py [path]/[filename.bam]` (see How does it work for more explanation about the python script). 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. Run the sorted.bam file in the python script using the command `python3 transposonmapping_satay.py [path]/[filename.sorted.bam]`.

Example of peressential_insertions.txt file:

```
Essential gene name chromosome Start location End location Insertion locations Reads per insertion location
EFB1 I 142174 143160 [142325, 142886] [1, 1]
MAK16 I 100225 101145 100229, 100407, 100422, 100791, 101022, 101129] [4, 1, 5, 1, 1, 1]
PRE7 II 141247 141972 [141262, 141736, 141742, 141895] [1, 1, 1, 1]
RPL32 II 45978 46370 [46011, 46142, 46240] [1, 3, 1]
```

Software Processing

This section discusses the main pipeline for processing satay datasets, from the raw fastq files of the sequencing output to the bed, wig and pergene text files which can be directly used for analysis. See the image below for a schematic overview of the pipeline with the used software tools between brackets and the file type after each processing step on the left. Everything that is shown in green can be automatically performed in a single workflow as will be discussed in the satay section below.


Input sequences reads in fastq format
Optional
Preprocessing
processing_workflow.sh
.fastq
Optional
Quality check raw data (FASTQC)
Trimming (BBDUK or Trimmomatic)
.fastq
Optional
Quality checking trimmed data (FASTQC)
Alignment (BWA MEM)
.sam
.bam
Checking alignment (SAMTools)
.bam
Indexing and sorting (Sambamba)
.bam
Transposon mapping (python)
.wig
.bed
_pergene.txt
_peressential.txt
_pergene_insertions.txt
_peressential_insertions.txt

satay.sh

Note that the workflow only runs in Linux (or Mac, but this is untested) as some of its dependencies are specifically designed for Unix machines.

The main program for the data processing is called satay.sh and is a bash script which automatically calls all required software tools.

Dependencies

Below is a list of dependencies. Note the folder structure for the python scripts and modules and the data files.

1. Zenity (by default installed in Linux Ubuntu 20.10)
2. YAD
3. FASTQC
4. BBMap
5. Trimmomatic
6. BWA
7. SAMTools
8. Sambamba
9. Python > v3.7

    1. numpy
    2. pysam
    3. timeit
    4. python_scripts/transposonmapping_satay.py
    5. python_scripts/python_modules/chromosome_and_gene_positions.py
    6. python_scripts/python_modules/gene_names.py
    7. python_scripts/python_modules/samflag.py
10. data_files/Saccharomyces_cerevisiae.R64-1-1.99.gff3
11. data_files/Cerevisiae_AllEssentialGenes_List.txt
12. data_files/Yeast_Protein_Names.txt

Input, Output

The bash script satay.sh accepts raw fastq files, either compressed (gzip) or uncompressed, with one of the following extensions: .fastq, .fq, fastq.gz, fq.gz. The fastq files can be either paired-end or single-end.

It is assumed that each data file contains one sample, so in case multiple samples are sequenced together, it might be necessary to demultiplex the fastq files. This is not integrated in the pipeline and should be performed before the pipeline is started.

When the processing was completed successfully, a number of output folders are created at the same location where the input fastq file is stored. The creation of some files and folders are depending on which options are selected in satay.sh (see below for more information about the options). The output may include the following folders and files:

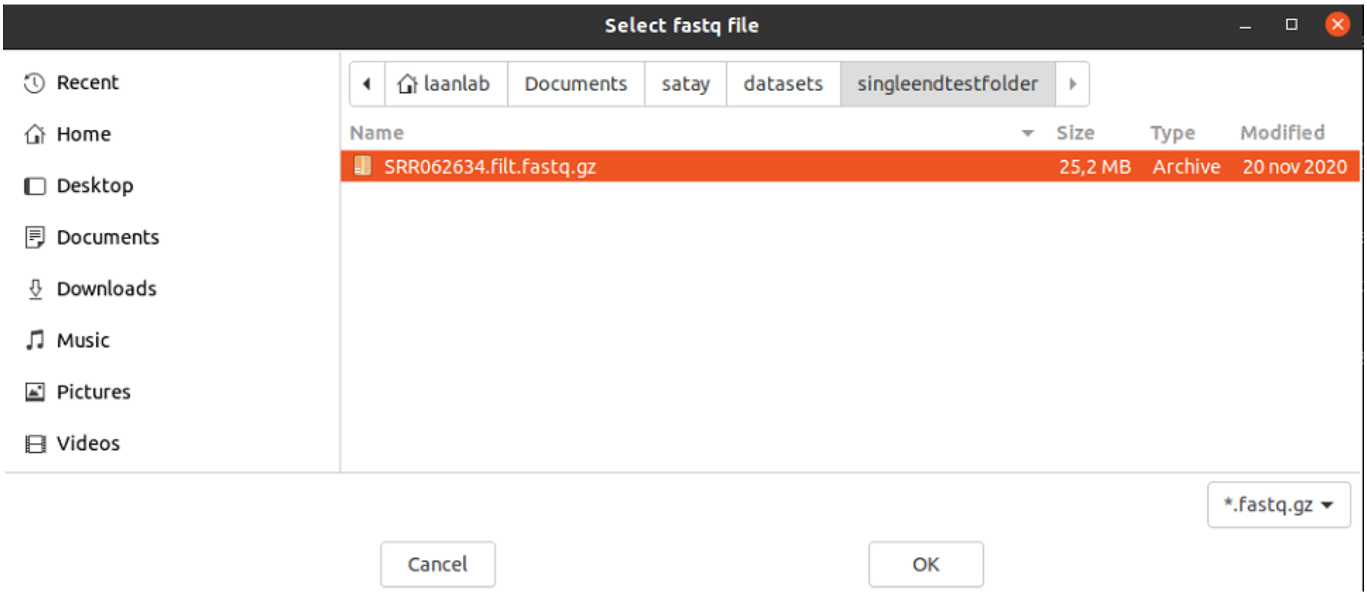
1. `align_out/`
    1. **.sam** (depending on whether the option for deleting sam files was selected)
    2. **.bam**
    3. **.sorted.bam** (depending on whether the option for `Sort and index bam files` is selected); The reads are sorted which speeds up downstream processing.
    4. **.sorted.bam.bai** (depending on whether the option for `Sort and index bam files` is selected);
    5. **flagstatreport.txt** (depending on whether `Create flagstat report is selected` is selected); Stores some basic information about the alignment.
    6. **.bed** (depending on whether `Transposon mapping` is selected)
    7. **.wig** (depending on whether `Transposon mapping` is selected)
    8. **pergene.txt** (depending on whether `Transposon mapping` is selected)
    9. **peressential.txt** (depending on whether `Transposon mapping` is selected)
    10. **pergene_insertions.txt** (depending on whether `Transposon mapping` is selected)
    11. **peressential_insertions.txt** (depending on whether `Transposon mapping` is selected)
2. `trimm_out/` (depending on whether the option for trimming is selected)
    1. **trimmed.fastq** (depending on whether trimming is selected)
3. `fastqc/` (depending on whether the option for quality checking is selected)
    1. **.html** (depending on whether Quality checking is selected); contains an a number of figures showing the quality of the data. This can be created for both raw data and/or trimmed data.
    2. **zipped folder** (depending on whether Quality checking is selected); contains the data for creating the figures in the .html file.

How to use

The workflow satay.sh only runs in Linux and is designed to be used as a commandline tool (see How to use the Linux desktop). For a step-by-step guide, see the Tutorial.

In the commandline, navigate to the software folder (`cd /home/laanlab/Documents/satay/software/`). The workflow can be started either using commandline arguments or by using a graphical user interface (GUI). Using the GUI is the most userfriendly approach. Access the help text using `bash satay.sh --help` or `bash satay.sh -h` and check the current version with `bash satay.sh -v`. The help text explains the different commandline arguments that can be set and briefly how to use the workflow. The commandline arguments are the same options that are set when using the GUI, so here only the GUI is explained.

Start the GUI with `bash satay.sh` without any arguments. This opens a window where the data file(s) can be selected. Navigate to the folder containing the data file(s) and select one or two files (select only two files when using paired-end data that is not interleaved by holding the ctrl button and clicking the files). Use the drop down window at the bottom right to select the right extension.



Select fastq file

laanlab / Documents / satay / datasets / singleendtestfolder

Name	Size	Type	Modified
SRR062634.filt.fastq.gz	25,2 MB	Archive	20 nov 2020

*.fastq.gz

After pressing `OK` a second window opens that shows all the settings with some default values that can be changed. The first two lines indicate the file(s) that were selected in the previous window (if only one file was selected, the secondary reads file is set to `none`). The third line is a drop-down menu to set whether to process the data as single-end (default) or paired-end reads. The fourth line sets which trimming tool to use. There are two trimming tools installed, BBDuk and Trimmomatic, which can be selected in this drop-down menu (default is BBDuks). If the trimming should be skipped (and therefore the alignment is performed with the raw input data), select `donottrim` in this menu. The fifth line sets the trimming arguments. When trimming is skipped, what is put in this line is ignored. Otherwise, use the arguments allowed by the selected trimming software.

Quality checking FASTQC: A quality report of the fastq files can be created before and after the trimming. When the dataset is processed for the first time, it is good to make a quality report of the raw data and check if there are no weird artefacts, reads with (unexpectedly) low quality and to see the overrepresented sequences. Overrepresented sequences are normal to occur, but most of the time it is good to trim those sequences as these can influence the alignment (unless when you're sure that the sequences belong in the reads and should not be trimmed). For example sequences that typically occur are adapter and primer sequences. Note that not always all sequences that need to be trimmed occur in the list of overrepresented sequences. Therefore is also good to manually check the fastq files and try to recognize any of the adapter or primer sequences. After trimming, create a new quality report and compare this with the raw data quality report. For example, check if any trimmed sequences do not occur anymore in the overrepresented sequences and, when trimming was performed based on the sequencing quality, see if the overall quality of the data has improved.

Trimming settings BBDuk: For BBDuk, a typical argument line looks like this: `ktrim=l k=15 mink=10 hdist=1 qtrim=r trimq=10 minlen=30`. The first four options are for trimming unwanted (adapter) sequences which in BBDuk is done using k-mers (e.g. when k=3, all 3-mers for ATTGCAAT are ATT, TTG, TGC, GCA, CAA, AAT). Any unwanted sequences that need to be trimmed (e.g. adapter, primer and transposon sequences) should be entered in the adapters file. Clicking the `Open adapters file` opens a text file in which these unwanted sequences can be placed in fasta format. Fasta format is similar to the fastq format, but without the last two lines (i.e. the dummy line and the quality line). The header line of each read should start with a `>` and can contain any text (preferably no special symbols to be sure) and the next line should contain the unwanted sequence, for example:

```
> header line first unwanted sequence
GATC
> header line second unwanted seqence
CATG
```

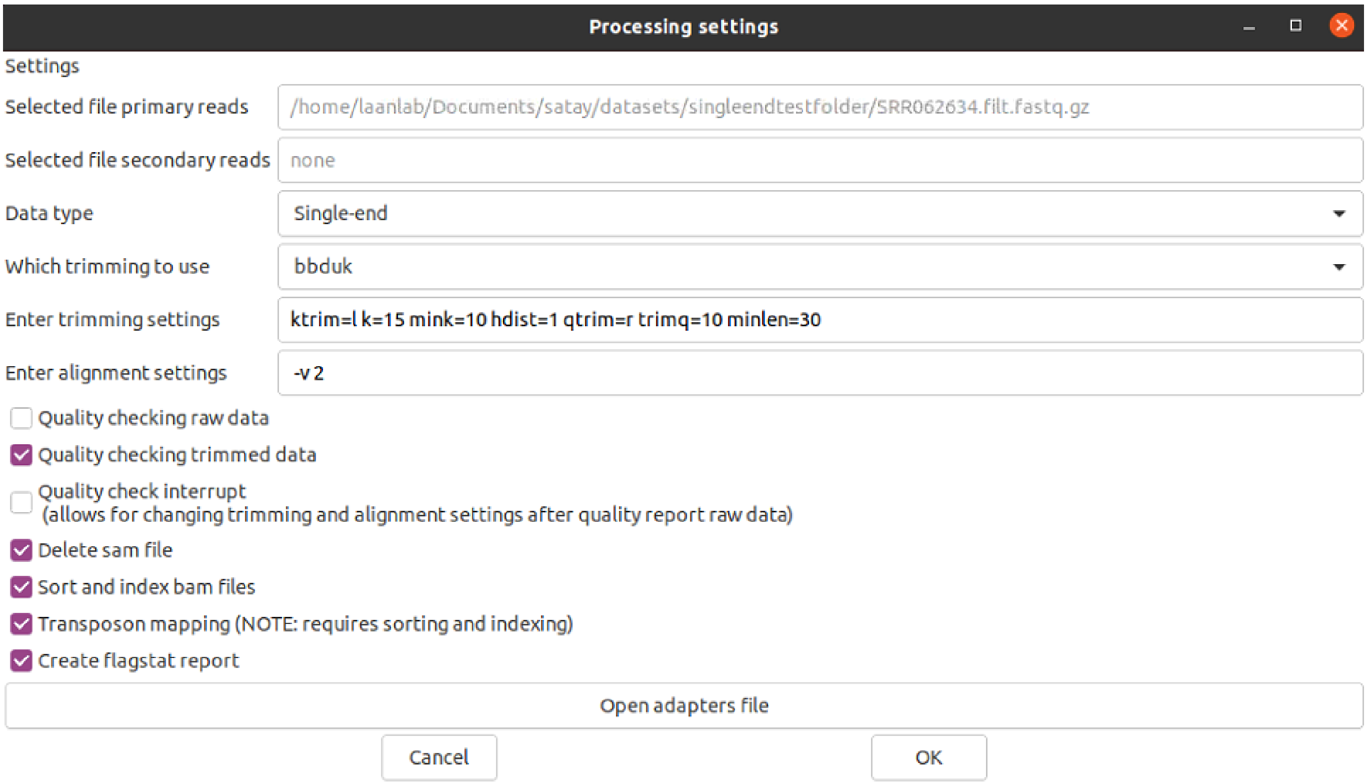
When all unwanted sequences are set, save the text file and close it. The value for k is set by the `k` parameter and should not be higher than the length of the shortest unwanted sequence (e.g. if the shortest sequence that needs to be trimmed has length 10, then k < 10). The higher this number, the smaller the chances are that false positives are found when trimming. This method, however, might skip over the last few basepairs of a read when the length of a read is not a multiple of k. To solve for this, the `mink` option can be set which allows for shorter k-mers at the end of a read. Therefore, `mink` < `k`. The `ktrim` option set to which side to trim when encountering an unwanted sequence (right (`r`) or left (`l`)). The number set with `hdist` determines the number of mistakes allowed when matching the unwanted sequences with the reads. Next are some options for quality trimming of the reads. The `trimq` option sets the minimum Q-score that basepairs are allowed to have on average. When the average quality of either first few or last few basepairs is lower than this threshold, those basepairs will be removed. Whether the basepairs on the left or on the right have to be trimmed is determined by the `qtrim` parameter (either `r` or `l` or `rl` to trim everything in that read). Finally the `minlen` parameter checks for the minimum length of a read after trimming. When this is smaller then this threshold, the read is completely removed. Note that the trimming can have a serious influence on the alignment and choosing good options can be important. For more information about these and many other settings check the BBDuk manual.

Trimming settings Trimmomatic: Alternatively, instead of using BBDuk for the trimming, Trimmomatic can be used that requires different input arguments. A typical input for Trimmomatic can look like this (note the capital letters): `ILLUMINACLIP:1:30:10 LEADING:15 TRAILING:15 SLIDINGWINDOW:5:15 MINLEN:10`. The `ILLUMINACLIP` argument performs the trimming of the unwanted sequences from the adapters file. Use the same file as for BBDuk which can be accessed using the `Open adapters file` button at the bottom of the window and also enter the reads in the same fasta format. The first number of this argument indicates how many mismatches are allowed. The second number is mainly for paired-end reads and determines how accurate a match should be when considering a read pair (even though this is only for paired-end reads, it is required to enter this value, but it is not important for single-end reads). The third value sets the accuracy of a match between an unwanted sequence and the read. The `SLIDINGWINDOW` is a quality control and uses a sliding window approach to determine the average quality of the reads within the window (length in basepairs is given by the first value) and trim the read when the quality drops below a threshold given by the second value. The `LEADING` and `TRAILING` arguments check the quality of the basepairs at the beginning or end of the read, respectively. If the quality is below the threshold given by the value, that basepair is trimmed. This is repeated for all basepairs from the start of the read or at the end of the read until a basepair is found that has a quality above the threshold. The `MINLEN` argument sets the minimum length of the read after trimming. If the length is below the threshold, the entire read is removed. For more information about these and many other settings check the Trimmomatic manual.

Alignment settings: The alignment settings line takes the arguments for the alignment software. The default value is `-v 2` which reduces the verbose level and prevents a whole lot of text to be printed during alignment, but doesn't do anything with the way the aligner works. The aligner determines for each read the position in the genome where it has the highest mapping score. The read is aligned to a location and then this score is determined by giving points for each letter in the read that match with the reference genome, but adding penalty points for evey time a letter in the read doesn't match with the genome (e.g. because of mismatches, extra letters in the read or missing letters in the read relative to the reference genome). This score is determined for each location where the read is aligned and the read is mapped to the location with the highest score. When the read cannot be unambiguously aligned (e.g. because all the mapping scores are too low or when there are mulitple locations that have the same highest score), the reads are aligned to a random location. The most important parameters that can be set alters the way the mapping score is influenced by matches, mismatches, insertions, deletions etc. Useful parameters to consider are: `-A`, `-B`, `-O`, `-E` and `-L`. When using paired end data it is also good to think about using the parameter `-U` to change the penalty for unpaired read pairs. For explanation about the parameters and their default values, see the BWA MEM manual.

Next are check boxes that can be set to turn on or off certain parts of the workflow. The first two checkboxes turn on or off quality checking before and after trimming. When `Quality check interrupt` is set, the program pauses after the raw quality checking (before the trimming) and asking if the user want to continue the processing (press `n` to stop and `y` to continue). Stopping the program allows the user to check the quality report of the raw data. After checking the quality report, the program can be restarted by typing `bash satay.sh` after which the program remembers the settings that were previously set (it will skip the file selection window). Changes can be made for the trimming (including the adapters file) and alignment tools and press `OK` to continue. This time the raw quality report will be skipped.

The next options determine if the sam file needs to be deleted after processing, whether the bam file needs to be sorted and indexed, if the transposon mapping needs to be performed and if a flagstat report needs to be created (the quality report of the alignment). Check if all the output files that are expected based on the settings are created (see the Input, Output section). See the section Summary and additional tips about how to check if the output makes sense based on the input data and the applied settings.



Settings

Selected file primary reads: /home/laanlab/Documents/satay/datasets/singleendtestfolder/SRR062634.filt.fastq.gz

Selected file secondary reads: none

Data type: Single-end

Which trimming to use: bbduk

Enter trimming settings: ktrim=l k=15 mink=10 hdist=1 qtrim=r trimq=10 minlen=30

Enter alignment settings: -v 2

☐ Quality checking raw data

☑ Quality checking trimmed data

☐ Quality check interrupt
(allows for changing trimming and alignment settings after quality report raw data)

☑ Delete sam file

☑ Sort and index bam files

☑ Transposon mapping (NOTE: requires sorting and indexing)

☑ Create flagstat report

The whole pipeline might take several hours to run, depending on the size of the dataset.

Tutorial

Note before use: When down-/uploading files to the N-drive or M-drive on the Linux Desktop, the drives will disconnect automatically after 10 minutes of inactivity. Since the files are typically large the down-/uploading can therefore take more then 10 minutes to complete. To prevent disconnection while down-/uploading you should click back and forth to some folders once every few minutes in the drives to reset the timer. See also How to use the Linux desktop.

- ☐ Log in to the Linux desktop (log in credentials can be found at `N:\tnw\BN\LL\Shared\LinuxMachines\LinuxDesktop_LoginCredentials.txt`).
  - ☐ When using remote access, install the free version of Teamviewer on your computer and connect with the computer using the credentials found in the file mentioned above.
- ☐ If your data file is on the N-drive, it should be copied to the Linux desktop for the processing to work smoothly (if you have no data set, there is a test fastq file already on the Linux desktop, so you can skip the following steps):
  - ☐ For this, go to `Files` (located in the left bar) and click `Other Locations` on the left menu in Files.
  - ☐ On the bottom at `Connect to Server` enter: `sftp://sftp.tudelft.nl/` and click `Connect`.
  - ☐ If requested, log in with your personal tudelft credentials.
  - ☐ Navigate to your data file (staff-bulk is the N-drive and staff-groups is the M-drive).
  - ☐ Copy the data set to the desktop, for example to `Documents/satay/datasets/`.
  - ☐ Close `Files`.
- ☐ Open the `Terminal` (located in the left bar) and enter `cd ~/Documents/satay/software/`.
- ☐ To start the workflow, enter `bash satay.sh`. This should open a window to select a fastq file.

- □ At the bottom right, select the right extension to be able to find your file, navigate to the location of your data set and select the fastq file and press `OK`. (For the test fastq, navigate to `Documents/satay/datasets/singleendtestfolder/SRR062634.fastq.gz`).
- □ In the next window (options window) enter the parameters and options. Click `Open adapters file` to change the sequences that need to be trimmed (if any) (this file should always be in fasta format, see How to use section) and save and close the adapters file. Press `OK` to start processing.
- □ If the option `quality check interrupt` was selected:
    - □ Wait for the processing to finish the quality checking of the raw data.
    - □ The workflow will ask if you want to continue. Enter `y` to continue with the parameters you have set and enter `n` to stop the workflow to check the quality report of the raw data.
    - □ Navigate to the folder where your data file is stored and open the .html file (in the `fastqc_out` folder) and the check the quality report.
    - □ When finished, restart the workflow by typing `bash satay.sh` at `~/Documents/satay/software/`.
    - □ The workflow should skip the file selection window and immediately start with the processing options window where the parameters you have previously entered are set by default. Change the parameters if needed and press `OK` to continue processing.
    - □ The workflow should skip over the raw quality checking and continue with the processing.
- □ When processing is finished, navigate to folder where your dataset is located and check if all expected files are present (depending on what options yout have set, but at least the `align_out` folder and a log file should be present, see the Input, Output section).
- □ Copy the results to the N-drive following the same procedure as described in step 2. (Be aware to prevent the sftp connection to automatically disconnect, see the note at the beginning of this section).
- □ After the processing is finished, most information is stored in the bed and wig files. But there can be some artifacts present in these files which can be removed using clean_bedwigfiles.py. This is only necessary for specific downstream analysis tools like the genome browser.

How does it work

When the pipeline is started after the files and options are selected, it starts with checking all the paths for the software tools and data files. The paths to the software tools and some other files can be adjusted by opening the bash script and change the paths in the `DEFINE PATHS` section at the beginning of the script. Also it checks if the options that were given by the user don't conflict (e.g. when two data files were selected, but the option for processing the data as single-end reads in which case the second data file is ignored). Next it defines names for all output files and folders.

After this the actual processing starts. Here the full processing workflow is explained, but some parts may be skipped depending on which options are set by the user.

The first step is quality checking of the raw data in the fastq files using FASTQC. This creates a quality report in the `fastqc` folder and will take a few minutes depending on the size of the datafile. After this first quality report the script will pause and ask the user to continue (enter `y` or `n`). If the program is stopped, the quality report can be checked after which the program can be restarted (again with the command `bash satay.sh`). All the previous settings will be remembered and do not have to be entered again. But some options (e.g. trimming and alignment) can be altered if this is prefered by the user after checking the quality report.
Next the trimming is started (either with BBMap or Trimmomatic). If the option for paired-end data is selected, automatically these options are also selected for the trimming (e.g. `interleaved=t` for BBDuk or `PE` for

Trimmomatic). Thus this should not be set manually by the user. This step also reads the adapter sequences file to search for unwanted sequences in the reads. The trimming can take more than half an hour to an hour (potentially even more for large datasets) and the results are stored in the `trimm_out` folder. After the trimming, another quality report is created for the trimmed data which stored in the `fastqc` folder.
Next up is the alignment by BWA MEM. Just like with the trimming, arguments for indicating whether paired-end data is used (`-p`) is automatically set and therefore the user should not set these parameters manually. The alignment can require more than one hour to complete and the results are stored in the `align_out` folder.
After the alignment a quality report is created using SAMTools which checks how many reads were aligned and if there are any things to be aware of (e.g. read pairs that were not mapped as proper pairs). The results are stored as `_flagstatreport.txt` in the `align_out` folder. The sam file is then converted to a bam file and it is checked whether the general format and structure of the bam file is correct. Both tasks are done using SAMTools. Next the bam file is sorted and an index file is created using Sambamba. All the steps so far after the the alignment should not take more than a few minutes and all the results are stored in the `align_out` folder.
As a final step the python script transposonmapping_satay.py is started (see below for more details). This creates all the files that store the information about the insertions and reads in the `align_out` folder (i.e. bed, wig, pergene.txt, peressential.txt, pergene_insertions.txt and peressential_insertions.txt). This python script can take well over an hour to complete.
After the transposon mapping the sam file is deleted and a log file is created that stores information about which file(s) has been processed, a time stamp and all the options and adapter sequences set by the user. This log file is stored together with the input fastq file.

The python script transposonmapping_satay.py consist of a single python function called `transposonmapper`. This takes a sorted bam file (which should have a bam index file present in the same folder) and potentially some additional information files. It starts with checking if all input files are present. Next if reads the bam file using the `pysam` function and gets some basic information about the chromosomes like the names and lengths and some statistics about the mapped reads in the bam file.
Then a for-loop is started over all chromosomes and for each chromosome a for-loop is started over all reads. For each read the samflag and the CIGAR string is taken to determine how the read was mapped (e.g. the orientation of the read). By default the leftmost position is taken as insertion site, but if the read is reversely mapped, the length of the mapped part of the read needs to be added to the insertion position as the actual insertion happened at the rightmost position of the read. After this correction the start position and the flags of each read are stored in arrays which are used later for creating the output files.
Lists are created for all genes and all essential genes including their start and end positions in the genome. Next the chromosomes are all concatenated in one large array, so that each chromosome does not start anymore with basepair position 1, but rather continues counting from the previous chromosome (e.g. chrI starts at basepair 1 and ends at basepair 230218, then chrII starts at basepair 230219 instead of basepair 1). This makes the next a bit easier where for each gene the insertions and reads are found that are mapped within the genes. The last part of code consists of creating the different output files. First the bed file is generated where the reads are saved using the equation (read_count*20)+100. Then the pergene and peressential text files are created where the number of insertions and read count are stored together with the standard names of the genes. The same is done for the pergene_insertions and peressential_insertions text files where all individual insertions and corresponding reads are stored which can be used later for determining the distribution of insertions within genes. Finally the wiggle file is created where first the insertions that were mapped to the same location but with a different orientation are taken as a single

insertion rather than two unique insertions. The corresponding reads are added up and this information is stored as a wig file.

Notes

- □ The alignment software arguments all require to start with a dash (`-`). For some reason this sometimes gives errors and to prevent this make sure there is a space before the first argument in the `Enter alignment settings` line.
- □ If the workflow unexpectedly skips over the file selection window and immediately shows the options window, it might be that a previous processing run was not correctly finished. Press `cancel` and restart the workflow. It should now start with the file selection window. (This most likely has to do with a cachefile, that is created after the `quality check interrupt`, that was not deleted. This can be deleted manually as well by removing `processing_workflow_cache.txt` at the same location where the workflow is stored).
- □ When using the commandline for entering the arguments, use `Paired-end` and `Single-end` with first capital letters.
- □ For all software in the pipeline, whether the data is paired-end or single-end is automatically set, so do not set options like `interleaved=t` (for BBDuk), `PE` (for Trimmomatic) or `-p` (for BWA) in the options window of the workflow.
- □ The names of the chromosomes in the reference genome that is used during alignment are not roman numerals as what is typically used for chromosomes. The output files (e.g. bed and wig files) get the chromosome names from this reference genome and since these are not roman numerals as what is generally expected, it might confuse some downstream analysis tools. There are two ways of solving this, either change the names in the reference genome (before indexing it, see the section S288C_reference_sequence_R64-2-1_20150113.fsa) or by using the function chromosome_names_in_files.py which can be used for converting the chromosome names found bed or wig files to roman numerals. To make the analysis tools as general as possible, the latter method is chosen with using the function and therefore the chromosome names in the reference genome are not changed. The reference genome mentioned in S288C_reference_sequence_R64-2-1_20150113.fsa uses the following chromosome names:
    - I: ref|NC_001133|
    - II: ref|NC_001134|
    - III: ref|NC_001135|
    - IV: ref|NC_001136|
    - V: ref|NC_001137|
    - VI: ref|NC_001138|
    - VII: ref|NC_001139|
    - VIII: ref|NC_001140|
    - IX: ref|NC_001141|
    - X: ref|NC_001142|
    - XI: ref|NC_001143|
    - XII: ref|NC_001144|
    - XIII: ref|NC_001145|
    - XIV: ref|NC_001146|
    - XV: ref|NC_001147|
    - XVI: ref|NC_001148|
    - Mito: ref|NC_001224|

Software analysis

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, see the Input, Output section.

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 compare 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.

clean_bedwigfiles.py

- **Main tasks**

Remove reads mapped outside chromosomes in .bed and .wig files, clean up those files and create custom header and optionally create additional bed and wig files containing information for a single chromosome.

- **Dependencies**

chromosome_and_gene_positions.py
chromosome_names_in_files.py

- **How and when to use**

This script consists of a single function called `strip_redundant_ins` with the following arguments:

`filepath=[path]` (required)
`custom_header=[text]`
`split_chromosomes=True||False`

Input of the function is a path to a bed or wig file (required). Optionally a string for the custom header can be added to the input that changes the header line of each file. By default the name in the header is the same name as the original datafile (fastq file), but this can be a long and unclear name. Therefore, it can be useful to change this when the bed or wig files are used for other analysis tools that use the name stated in the header, for example the genome browser. If no custom header is provided, the original header is used. Also the names of the chromosomes are the names that are used in the reference genome (see satay.sh notes for more explanation). In this function these names are replaced by roman numerals which is more commonly used.

Sometimes insertions can be mapped outside chromosomes (i.e. the insertion position of a read is larger than the length of the chromosome it is mapped to). The reason for this result is unclear, but is likely to be the result of either the alignment or transposon mapping in the processing workflow. It can cause issues with downstream analysis tools and therefore this function removes those reads. The removed reads are shown in the terminal where the python script runs. Check that only few reads are removed.

Finally, when the `split_chromosomes` argument is set to True, separate files are created with the contents of each individual chromosome in bed or wig format. These files are stored in a dedicated folder with the same name as the bed or wig file with the extension `_chromosomesplit`. In this folder the files are stored, again with the same name as the input bed or wig file with roman numeral as extension indicating from which chromosome it stores the information.

- **Output**

A new file is created at the same location as the input file with `_clean` added to the name and with the same extension as the input name (e.g. input WT.bed results in WT_clean.bed being created). This contains the same information as the original bed or wig file, but with roman numerals for the chromosome names and the reads outside the chromosomes removed. Optionally the header is changed when this was provided by the user. A dedicated folder can be created where multiple files are stored, each containing the information for a specific chromosome.

- **Notes**

- □ In the wig file, each chromosome starts with a VariableStep line. By default this is written with capital V and S, but some tools may have issues with the capital letters. Therefore this script also converts everything to lowercase letters (i.e. variablestep).

genomicfeatures_dataframe.py

- **Main tasks**

Create a pandas dataframe that stores information for a specific chromosome including all genomic features, positions and the number of insertions and reads within those features for an entire chromosome or specific genomic region. Optionally it can create a barplot with the number of insertions or reads within each feature.

- **Dependencies**

pandas
matplotlib
chromosome_and_gene_positions.py
chromosome_names_in_files.py
gene_names.py
read_sgdfeatures.py

Cerevisiae_AllEssentialGenes_List.txt
SGD_features.tab
Saccharomyces_cerevisiae.R64-1-1.99.gff3
Yeast_Protein_Names.txt

- **How and when to use**

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 telomeres, 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`).

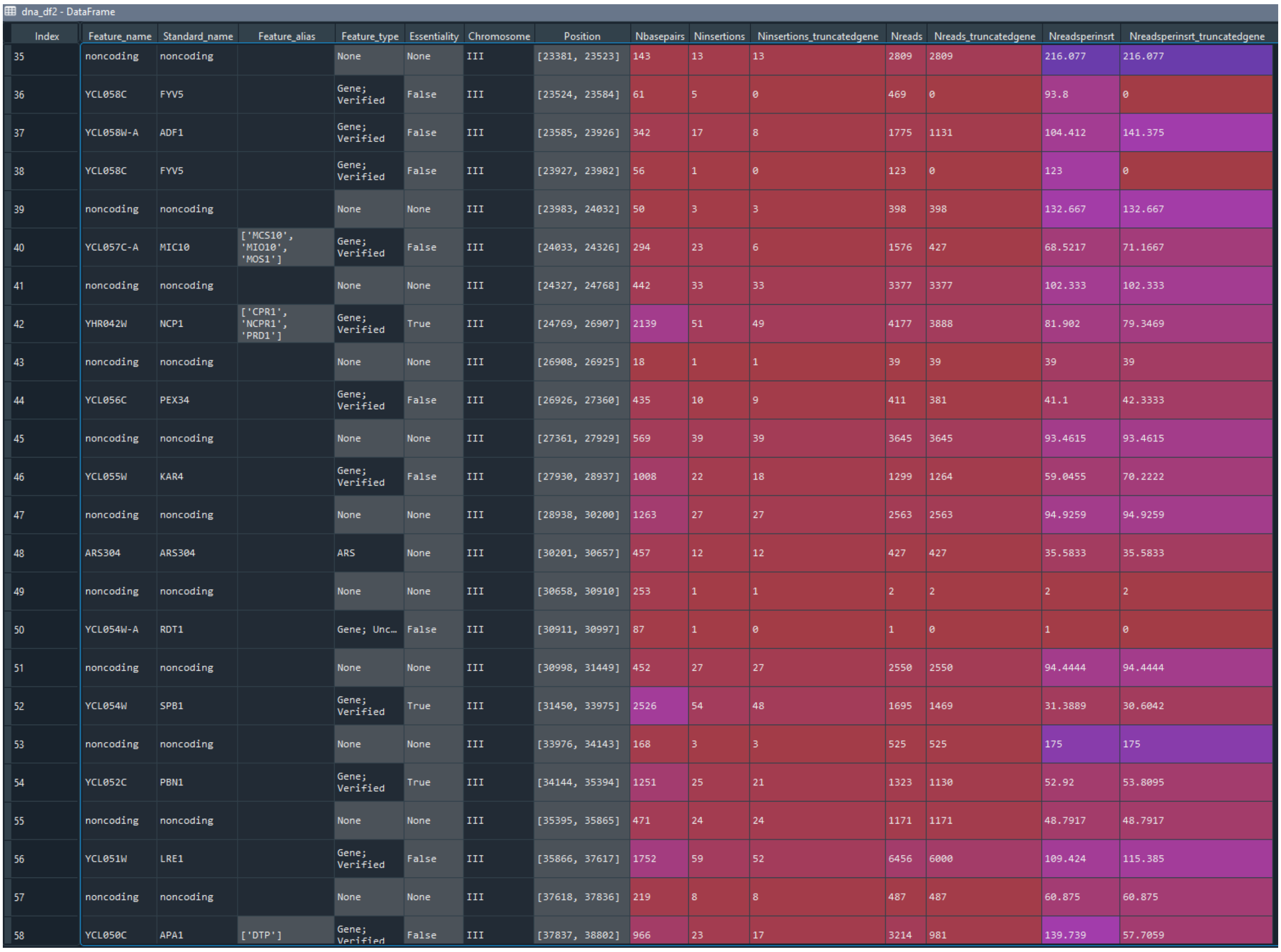
This function can be useful for other python functions when information is required about the positions, insertions and read counts of various genomic features. The full list of genomic features that is regarded in the dataframe is mentioned in read_sgdfeatures.py.

- **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)

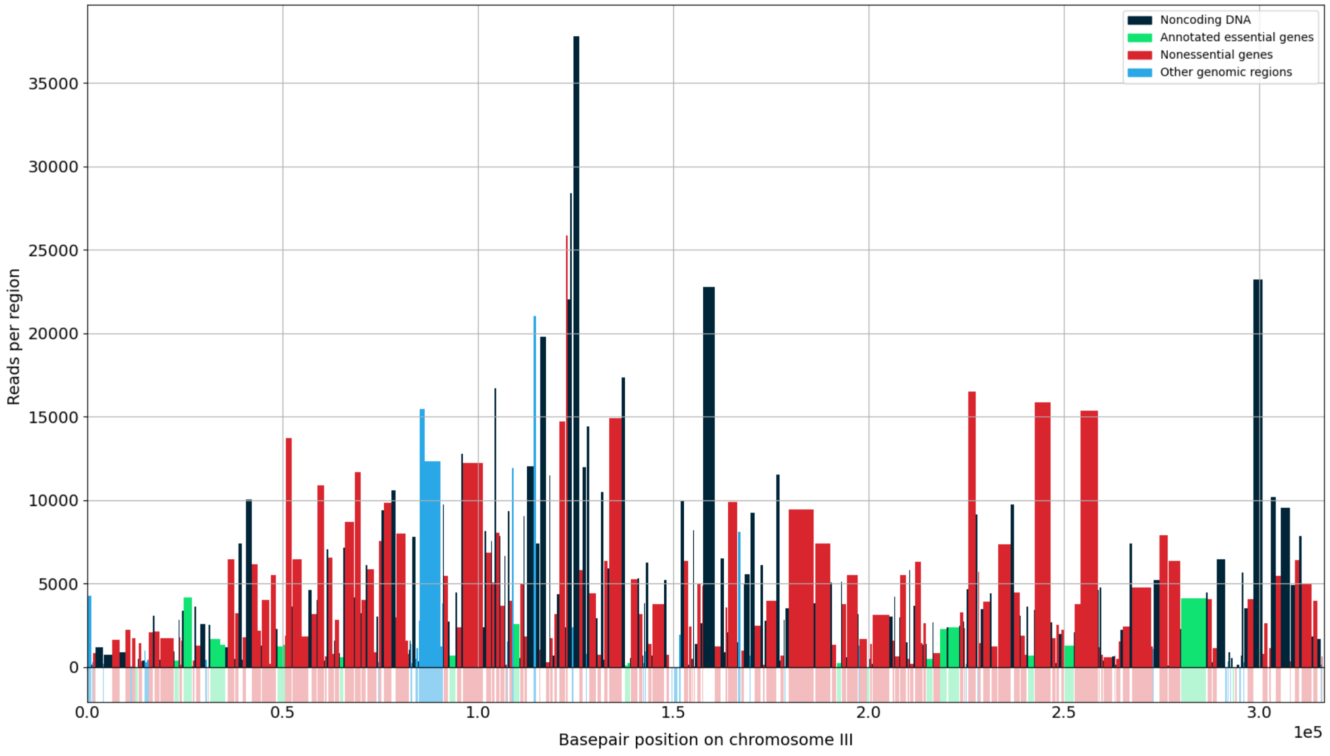
The truncated feature columns ignores basepairs at the beginning and end of a gene. This can be useful as it is mentioned that insertions located at the beginning or end of a gene can result in a protein that is still functional (although truncated) (e.g. see Michel et.al. 2017) (see Notes for a further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



- **Notes**

- □ 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).

- □ A stripped down version of this script exists as a python module called dataframe_from_pergene.py.

transposonread_profileplot.py

- **Main tasks**

Create a barplot to show the read or insertion count for one chromosome and indicate the location of the genes which are colorcoded based on their essentiality.

- **Dependencies**

numpy
matplotlib
chromosome_and_gene_positions.py
chromosome_names_in_files.py
essential_genes_names.py

- **How and when to use**

This code consists of a single function `profile_plot` which takes the following arguments:

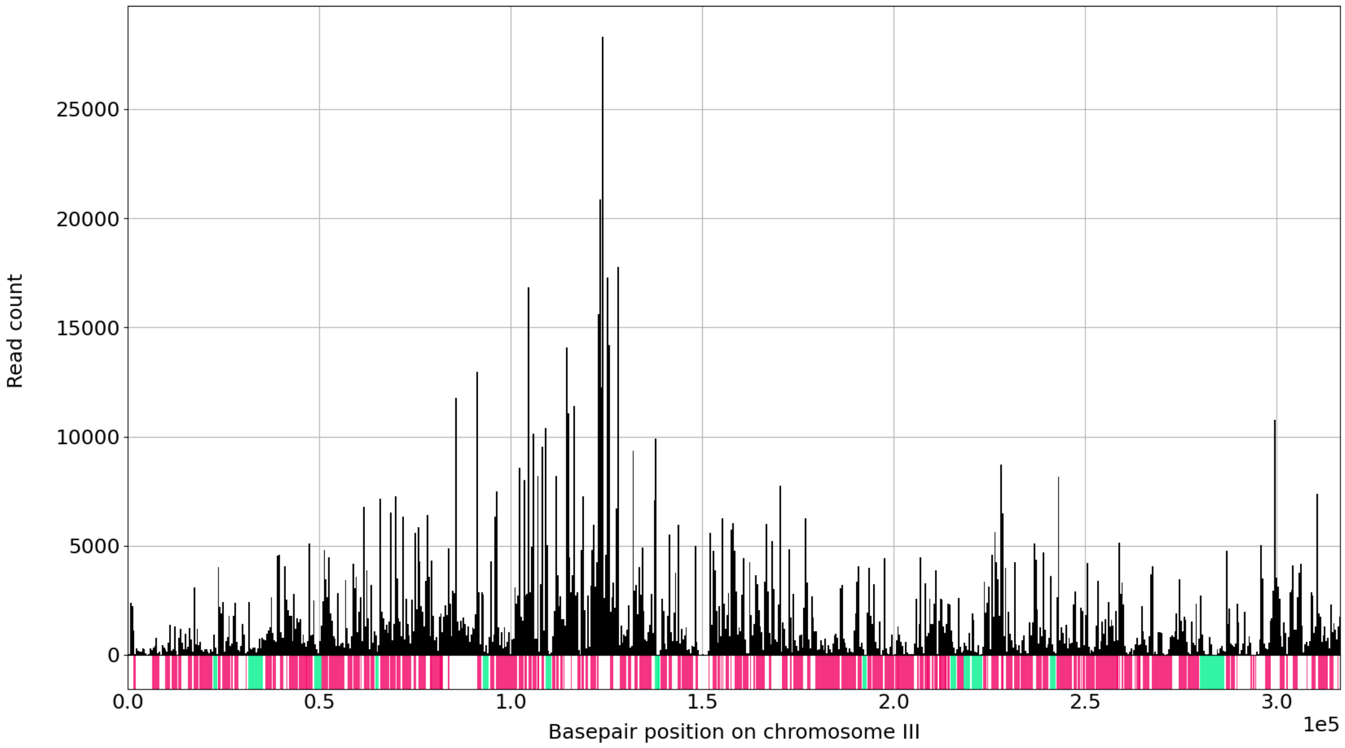
`bed_file=[path]` (required)
`variable="transposons"||"reads"`
`chrom=[roman numeral]`
`bar_width=[int]`
`savefig=True||False`

The `variable` arguments determines whether to plot the number of transposons or the number of reads for a chromosome defined in `chrom`. The `bar_width` argument defines the width of the bars (in basepairs). All insertions or reads within a bar are added up. Wider bars makes the script significantly faster, but this might cause regions with low abundance less visible where the smaller bars provide more detailed information. By default the width of the bars are chosen as the length of the chromosome divided by 800.

Optionally, the figure can be automatically saved at the same location where the bed file is stored with the `savefig` argument. The name given to the figure is the name of the bed file with the extension _[variable]plot_chrom[chrom].png (where the names between the brackets are replaced with their respective names from the input).

- **Output**

A barplot is created which shows the transposon or read count throughout the genome together with a strip that is colorcoded at the location of genes based on their essentiality to compare the insertion or read abundance relative to the location of the genes.



- **Notes**

- [ ]

transposonread_profileplot_genome.py

- **Main tasks**

Create a barplot to show the read or insertion count for the whole genome and indicate the location of the genes which are colorcoded based on their essentiality.

- **Dependencies**

numpy
matplotlib
chromosome_and_gene_positions.py
chromosome_names_in_files.py
essential_genes_names.py

- **How and when to use**

This code consists of a single function `profile_genome` which takes the following arguments:

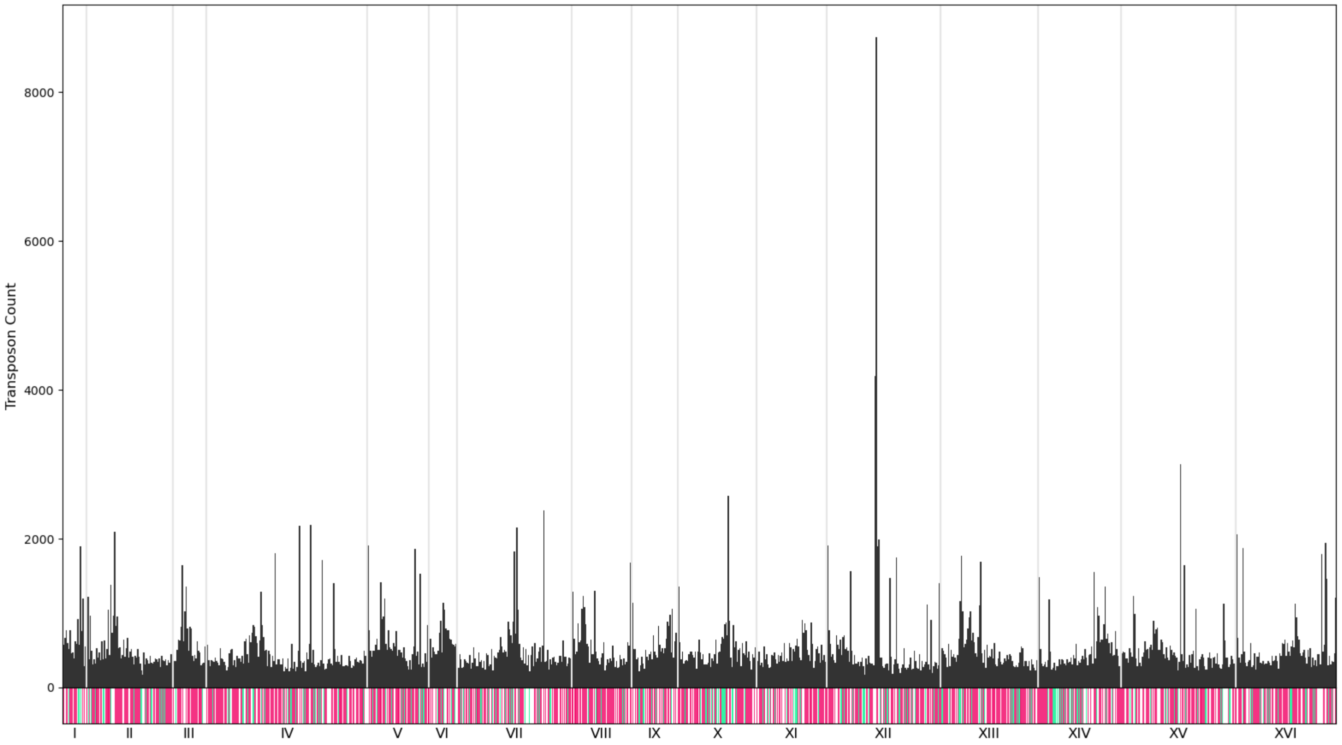
`bed_file=[path]` (required)
`variable="transposons"||"reads"`
`bar_width=[int]`
`savefig=True||False`

This code is very similar to transposonread_profileplot.py, instead this code creates a barplot for the entire genome. The `variable` arguments defines whether the transposon or read count is plotted and the `bar_width` sets the number of basepairs that is considered in each bin. All the reads or insertions are summed within each bin. A lower value allows for more details to be visible, but can take more time to plot. Higher values may obscure areas of low abundance. Default setting is length of the genome divided by 1000 (i.e. aprroximately 1200bp per bin).

Optionally the figures might be automatically saved by setting `savefig` to True where the figure is saved at the same location and name as the bed file with the extension [variable]plot_genome.png (where [variable] is replaced with whatever is set in the `variable` argument).

- **Output**

A barplot is created that shows the tranposon or read abundance for the genome together with the location of the genes that are colorcoded based on their essentiality.



- **Notes**

- ☐ Sometimes the colorcoding of the strip below the figure changes when the figure is saved. This is not due to the code (which always loads the same gene positions from the same files), but this likely due to how the figure was saved by the used editor (tested with Spyder4). Probably the resolution is not high enough to show the fine details in the strip.

TransposonRead_Profile_Compare.py

- **Main tasks**

Create a barplot for two datasets, either with transposon counts or reads counts, and visualize the differences between the two datasets.

- **Dependencies**

numpy
matplotlib
chromosome_and_gene_positions.py
chromosome_names_in_files.py
essential_genes_names.py
gene_names.py

Saccharomyces_cerevisiae.R64-1-1.99.gff3
Yeast_Protein_Names.txt
Cerevisiae_AllEssentialGenes_List.txt

- **How and when to use**

This code consists of a single function called `compareplot`. This function takes the following inputs:

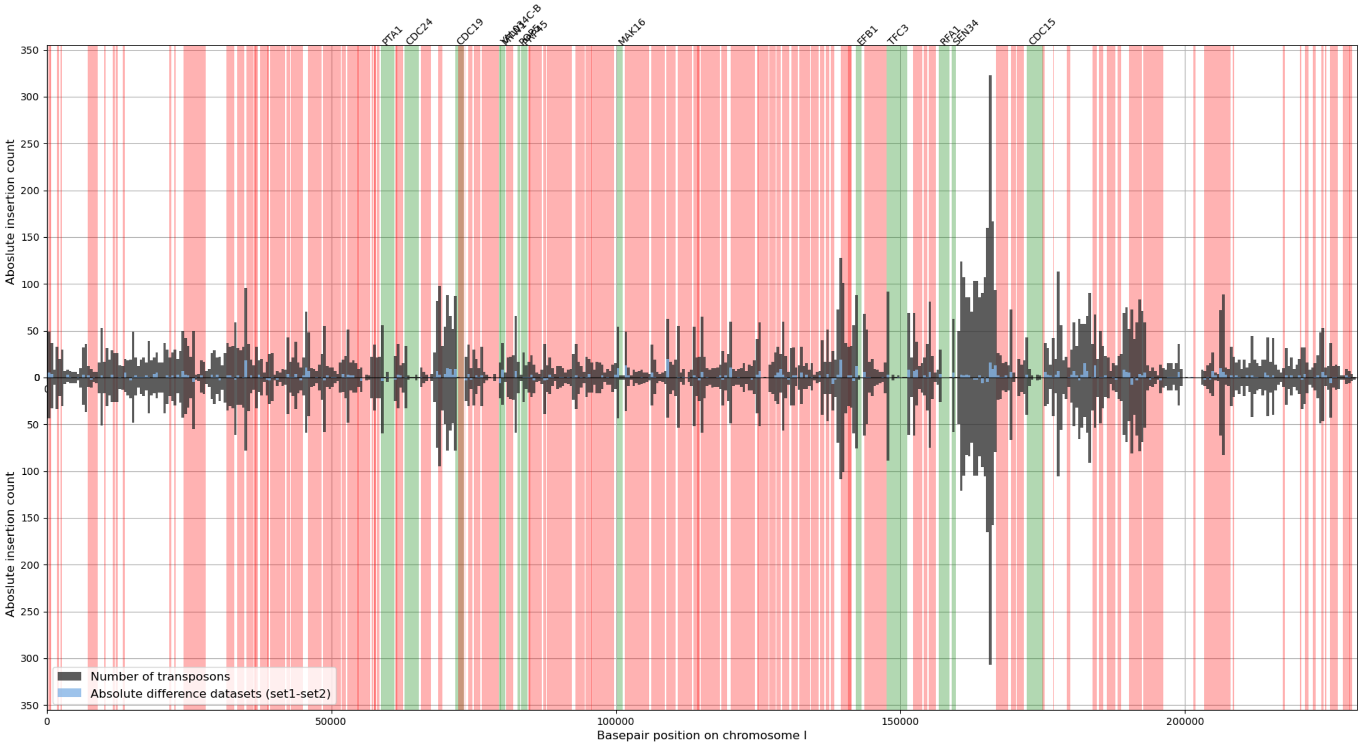
`bed_files=list of 2 paths` (required)
`variable="insertions"||"reads"`
`chromosome=roman numeral||list of roman numerals`
`set_barwidth=integer`
`set_logscale=True||False`
`savefig=True||False`

There are two bed files compared and therefore the `bed_files` arguments takes a list containing the paths of two bed files. This first one is called `set1` (shown in the top graph in the figure) and the second is called `set2` in the output figure (shown in the bottom graph in the figure) and the difference is determined by set1-set2. The plot can be created for both transposon counts (`variable="insertions"`) or for read counts (`variable="reads"`). The `chromosome` takes a roman numeral for the chromosome that needs to be analyzed. Optionally a list of roman numerals can be given that will be analyzed all at once. This creates a number of figures equal to the number of chromosomes entered in the list. The `bar_width` can be set to determine the width of the bars in basepairs (default is length of the chromosome divided by 500). The differences between datasets can be small relative to the number of counts, especially when comparing reads. To make the differences better visible, the y axis can be set in log scale using the `set_logscale` argument to `True`. This visually enhances the differences, but can give a distorded view as the differences appear much bigger than they actually are. Finally, the `savefig` argument determines whether the figure needs to be saved. If a figure is saved, it won't be shown and it will be stored at the location of the bed file that occurs first in the `bed_files` list. The name that is given to the saved figures is the name of the first occuring bed file with the extension `_compareplot_chrom[romannumeral].png` where [romannumeral] is replaced by the chromosome number. The background of the figure is colorcoded at the location where the genes are located where red are the non-essential genes and green are the annotated essential genes in wild type.

This figure can be useful to create after processing the same dataset multiple times with different settings (e.g. for the trimming and alignment).

- **Output**

Two barplots are created plotted on top of each other where black bars show the insertion or read count for either of two datasets. Note that the y axis of the bottom graph is inverted. The overlapping blue bars indicate the absolute difference between the datasets where the bottom graph (set2) is substracted from the top graph (set1). The colorcoded background indicate the location of essential and non-essential genes. On top the graph the names of the essential genes are plotted.



- **Notes**

- □ When automatically saving the figure the names of the compared files are not stored, so it is good to (manually) create a file stating what is compared.

scatterplot_genes.py

- **Main tasks**

Create a sorted scatterplot that shows the number of reads per insertion per gene for both annotated essential and non-essential genes. This is plotted together with a histogram of the reads per insertion per gene to indicate potential differences in the distribution between essential and non-essential genes.

- **Dependencies**

numpy
matplotlib
seaborn
pandas
essential_genes_names.py

Cerevisiae_EssentialGenes_List_1.txt and Cerevisiae_EssentialGenes_List_2.txt

- **How and when to use**

This code consists of a single function called `scatterplot` that takes a single argument for the path of the pergene.txt file. From this file it takes, for each gene, the number of insertions and the number of reads and with this it calculates the number of reads per insertion. This is stored in a dataframe together with the essentiality (True or False) for each gene which is returned as output.

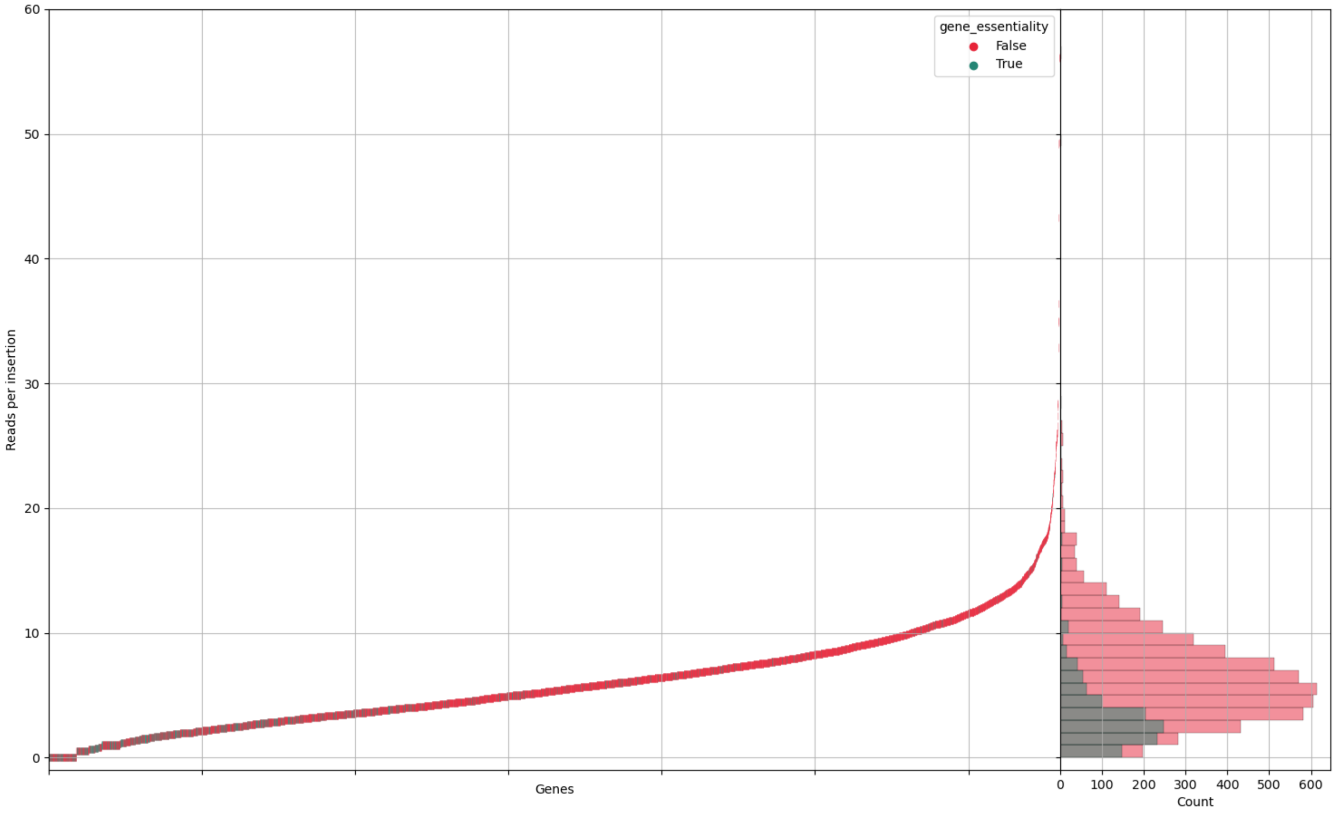
The genes are sorted based on their reads per insertion value and therefore the genes with the lowest reads per insertion are on the left of the graph and the genes with the highest values are always on the right of the graph. The genes are colorcoded based on their essentiality.

Next to the sorted scatterplot a histogram is shown for the y-axis of the scatterplot that shows the distribution of the reads per insertion for the essential genes and the non-essential genes in two overlapping histograms.

- **Output**

The main output is a sorted scatterplot (i.e. a hockeystickplot) together with a histogram showing the distribution of the reads per insertion for essential and non-essential genes.

Next to the plots it returns a dataframe with data from the pergene.txt file together with the essentiality of each gene.



- **Notes**

- ☐ This code is relatively bare bones and could be extended with statistics to more thoroughly compare the essential and the non-essential gene distributions.

volcanoplot.py

- **Main tasks**

Creates a a volcano plot that shows the fold change per gene between two datasets and the significance of this fold change.

- **Dependencies**

scipy
matplotlib
dataframe_from_pergene.py

- **How and when to use**

This script consists of a single function called `volcano` which takes the following inputs:

`path_a=[path]` (required)
`filelist_a=[list of pergene.txt filenames in path_a]` (required)
`path_b=[path]` (required)
`filelist_b=[list of pergene.txt filenames in path_b]` (required)
`variable="read_per_gene"||"tn_per_gene"||"Nreadsperinsrt"` (default=read_per_gene)
`siginificance_threshold=[int]` (default=0.01)
`normalize=True||False` (default=True)
`trackgene_list=[list of gene names]`
`figure_title=[str]`

A volcanoplot is useful to create to compare two strains with each other that each has multiple datasets (e.g. 2 wt strains and 4 mutant strains). Therefore, the paths of the two datasets (`path_a` and `path_b`) are separated from the pergene.txt filenames, which should be given in a list (`filelist_a` and `filelist_b`).

It plots the fold change per gene vs the significance as determined by an independent t-test. The fold change can be determined for the reads per gene (`variable=reads_per_gene`), insertions per gene (`variable=tn_per_gene`) or reads per inserions per gene (`variable=Nreadsperinsrt`). The fold change is calculated by the log base 2 of the mean per gene for the experimental strain divided by the mean per gene for the reference strain (e.g. mean reads per gene mutant strain / mean reads per gene wt strain).

Before the mean is determined, the reads or insertions can be normalized by the total number of reads or insertions in each dataset. This normalization can be skipped by setting `normalize=False`.

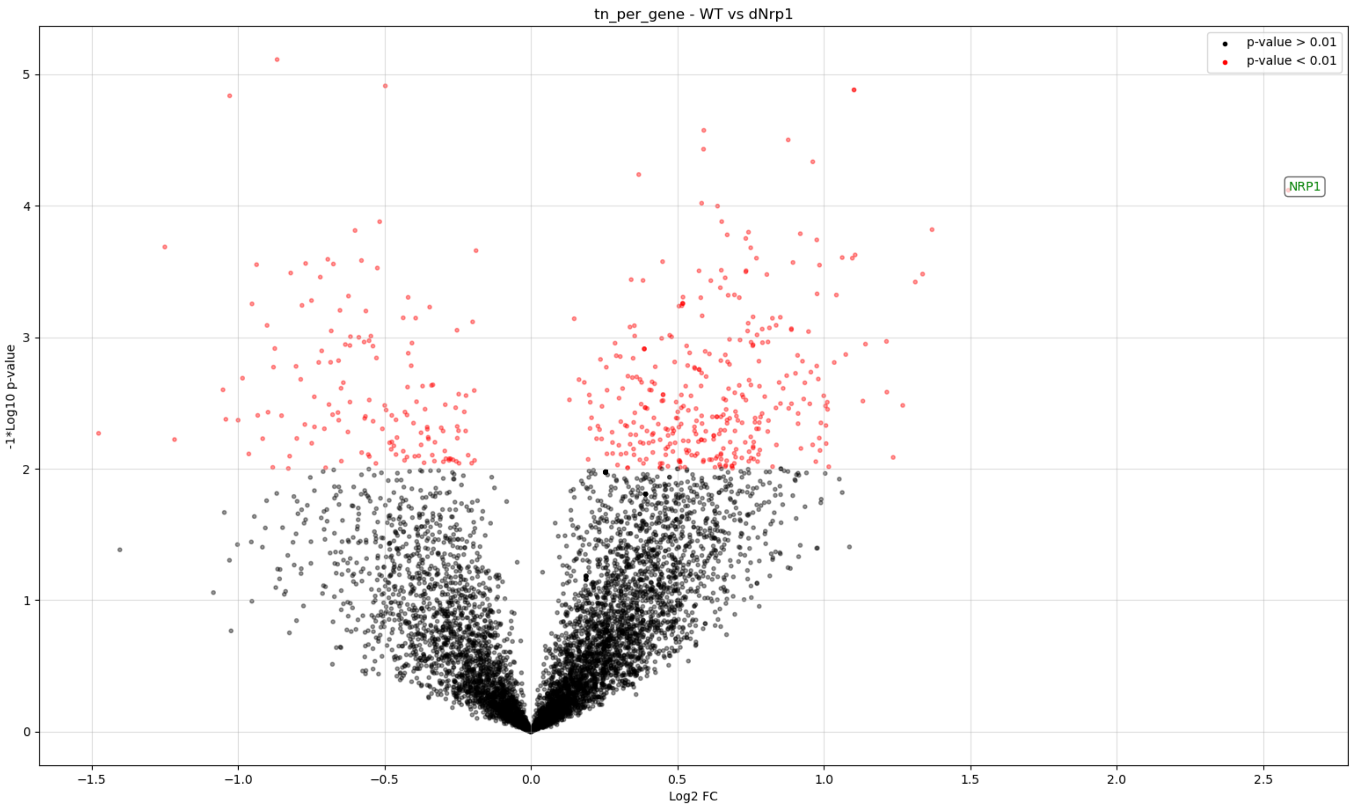
When the figure is created, the genes (each represented by a single dot) are colored depending on their significance. The significance is determined using an independent t-test (calculated using `scipy.stats.ttest_ind()`) and it represented as the -log base 10. This means that genes that are high up on the y-axis and far away from the 0 on the x-axis are the genes with a large and significant fold change. When the significance of a gene is higher then the threshold set by `threshold`, then the genes are considered significant and are colored red, otherwise they are colored black.

Optionally, a list of genes can be given which are shown with a fixed label in the graph. For this enter the gene names as a list in the `trackgene_list` variable and use the same names as they occur in the files. The other genes can be found in the graph by hovering the cursor over each datapoint and then a label will appear that show the name of the gene that the datapoint is representing. The title of the graph can be changed using the `figure_title` by entering a string that will be printed with the figure. Next to the custom title, the title indicates what variable is plotted in the graph (i.e. reads per gene, insertions per gene or reads per insertion).

- **Output**

The main output of the script is the interactive volcano plot of fold change vs significance. Also it returns the variable `volcano_df` that includes the information used for plotting. In the dataframe each row consists of a gene and the following columns are used:

- gene names
- fold change
- t statistic
- p value
- significance (True or False)



- **Notes**

- ☐ The fold change is calculated as the ratio between the means of the datasets. But this is an issue when either of the dataset is 0. To prevent this issue, for each gene in each dataset, 5 insertions and 25 reads are added. This ensures there is never a 0 in the division. This chosen because the processing of the Kornmann-lab also does this and by integrating the same method in this script allows for direct comparison of the figures. However, this is not completely fair as it changes the data. For example, if the initial number of insertions were 1 and 2 it yields a fold change of 0.5. By adding 5 insertions to it it becomes 6 and 7, respectively, yielding a fold change of 0.85.

create_essentialgenes_list.py

- **Main tasks**

Combine mulitple files with essential genes into one.

- **Dependencies**

gene_names.py

- **How and when to use**

This code was initially created because there were multiple text files with essential genes for Cerevisiae. However, none of the lists was complete and thus there were genes in one file that were not in the other and vice versa. Also the files used different naming conventions. This code takes a list of paths to multiple text files containing gene names. It assumes there are three header lines in each text file (which can be empty) and after that each line contains a single name of a gene.

This code iterates over all files given in the input list and creates a new text file at the same location as the first occuring path in the input list and gives this new file the name `Cerevisiae_AllEssentialGenes_List.txt`.

It takes the gene names from each file and converts the names in the oln naming convention. While iterating over all input files it searches if a gene already occurs in the newly created file. If a gene already exists, it will be skipped to prevent the same genes are being stored multiple times.

The current Cerevisiae_AllEssentialGenes_List.txt file is created with this code.

- **Output**

A text file is created that contains all known essential genes with the names `Cerevisiae_AllEssentialGenes_List.txt`.

- **Notes**

- ☐ This code is hardly used, but it was needed when the creating the list of essential genes from multiple sources. It might be useful when either of the lists are updated or other lists are added.

python modules

The python modules always contain functions that perform a certain task that is often required in multiple python scripts. To use these python modules in a python script a path needs to be defined to the module. This can be done best using relative paths stated at the very beginning of a script with the following lines:

```
import os, sys
file_dirname = os.path.dirname(os.path.abspath('__file__'))
sys.path.insert(1,os.path.join(file_dirname,'python_modules'))
```

Then an import can be done as usual in the form

```
from module_name import function
```

For example to import the `chromosome_position` function from the `chromosome_and_gene_positions.py` module:

```
from chromosome_and_gene_positions import chromosome_position
```

The output of these functions are given in the `return` statement. If there are multiple outputs in the return statement, the function also expects multiple variables to assigned when calling this function in a python scripts (e.g. if function `f` has two outputs, call this the function as `var1, var2 = f([inputs])` or `var1 =`

`f([inputs])[0]` if only `var1` is needed, where `[inputs]` are to be replaced with the input variable(s) (if any)).

chromosome_and_gene_positions.py

This module can be used for obtaining information regarding gene and chromosome positions and lengths and converting roman to arabic numerals for chromosomes. It contains 3 functions:

- `chromosome_position`

This function outputs 3 dictionaries:

`chr_start_pos_dict` which contains the start position of all chromosomes
`chr_end_pos_dict` which contains the end position of all chromosomes
`chr_length_dict` which contains the lenghts of all chromosomes

The input is a .gff3 file which, in case no output is given, is obtained from the data_files folder.

- `chromosomename_roman_to_arabic`

This function outputs 2 dictionaries:

`arabic_to_roman_dict` for which the keys are arabic numerals and the values are roman numerals
`roman_to_arabic_dict` for which the keys are roman numerals and the values are arabic numerals

There is no input for this function.

- `gene_position`

This function outputs 1 dictionary:

`gene_pos_dict`

The keys in this dictionary are all gene names and each value is a list with the following information:

1. Chromosome where the gene is positioned
2. Start basepair for the gene
3. End basepair for the gene
4. `+` or `-` depending on whether it is on the forward or reverse strand.

The input the path to a gff3 file which, if not provided, is taken from the data_files folder.

This module is also present as a python module in the python_transposonmapping folder for the processing workflow, although in a slightly altered version.

chromosome_names_in_files.py

This module extracts information about where the chromosomes start and end in a bed or wig file and gets the names of the chromosomes as used in these files. It consists of 2 functions:

- `chromosome_name_bedfile`

This function outputs 3 dictionaries:

`chrom_names_dict` which contains as keys roman numerals for the chromosome and the values are the chromosome names as are present in the bed file
`chrom_start_line_dict` containing the lines where the chromosomes start in the bed file as values and the keys are roman numerals representing the chromosomes
`chrom_end_line_dict` containing the lines where the chromosomes end in the bed file as values and the keys are roman numerals representing the chromosomes

This can be useful when only a specific chromosome is required, that not the entire file needs to be searched, but rather a range of lines in the bed and wig files can be read where the specified chromosome occurs.

The input of this function is a required path to a bed file.

- `chromosome_name_wigfile`

This function is very similar as for the bed file, but altered to work specifically for the wig file and also outputs the same 3 dictionaries:

`chrom_names_dict` which contains as keys roman numerals for the chromosome and the values are the chromosome names as are present in the wig file
`chrom_start_line_dict` containing the lines where the chromosomes start in the wig file as values and the keys are roman numerals representing the chromosomes
`chrom_end_line_dict` containing the lines where the chromosomes end in the wig file as values and the keys are roman numerals representing the chromosomes

The input of this function is a required path to a wig file.

dataframe_from_pergene.py

This module creates a dataframe with the number of insertions and reads per gene and can be seen as a stripped down version of the python script genomicfeatures_dataframe.py. It contains a single function:

- `dataframe_from_pergenefile`

This requires as input a path to a pergene.txt file and outputs a pandas dataframe where each row contains a gene and has the following columns:

1. gene names
2. gene essentiality
3. transposons per gene
4. read per gene
5. number of reads per insertion

This dataframe is stored in `read_gene_df`. The module has the following dependencies:

numpy
pandas
re
Cerevisiae_EssentialGenes_List_1.txt
Cerevisiae_EssentialGenes_List_2.txt

essential_genes_names.py

This module creates a list with all known essential genes in different naming conventions. It contains a single function:

- `list_known_essentials`

This inputs a lists of files with essential gene names. If no input is given, the files are collected from the data_files folder. It simply takes all genes from all input files and put those in one list. This list is therefore likely to contain redundant genes with different naming conventions. This list is stored in the variable `known_essential_gene_list` and is useful to check whether a gene is annotated as essential without having to worry about naming conventions.

gene_names.py

This module can be used for getting all gene names and their aliases. It consists of two functions:

- `list_gene_names`

This function outputs a list of all genes names including aliases. It is not indicated which gene names refer the same gene, but rather this is a complete overview of all gene names that are known for yeast. This can be used, for example, to check whether an inputted gene name exists or not by checking if the gene name occurs in this list. The input of this function a Yeast_Protein_Names.txt file, which if not provided is taken from the data_files folder. The output function is `gene_name_list`.

- `gene_aliases`

This function the more extensive version of the first function and outputs 3 dictionaries:

`aliases_designation_dict` which contains as keys the gene names in systematic naming convention and the values are lists containing all aliases in the standard naming convention
`aliases_sgd_dict` which contains as keys the gene names in systematic naming convention and the values are the SGD ID (this can be used on yeastgenome.org)
`aliases_swissprot_dict` which contains as keys the gene names in systematic naming convention and the values are the swissprot ID (this can be used on uniprot.org)

This function also inputs a Yeast_Protein_Names.txt file, which if not provided is taken from the data_files folder.

This module is also present as a python module in the python_transposonmapping folder for the processing workflow.

mapped_reads.py

This module can be used for counting the number of insertions and number of reads in a bed or wig file. It consists of a single function:

- `total_mapped_reads`

This function counts the number of lines in the bed or wig file (minus the header line and empty lines at the end) to get the number of insertions and sums the number of reads. The output is a dictionary containing 3 elements:

`Ninsertions` which represents the number of insertions
`Nreads` representing the number of reads
`Median` representing the median read count per insertion

The input is either a wig or a bed file and a verbose argument can be set to False to suppress any printing of the values.

Note that using a bed file is more accurate than using a wig file. In the wig file the insertions that were mapped to the same location, but have a different orientation are summed. This means that the number of reads should be the same as the number of reads found in the bed file, but the number of insertions can deviate depending how often reads with a different orientation are mapped to the same location. See for more information see the section about the wig file.

This module has the following dependencies: numpy

read_sgdfeatures.py

This module creates dictionaries for many types of genomic features containing general information about these features. It consists of a single function:

- `sgd_features`

This function reads the SGD_features.tab file and extracts the following information:

- ORF (genes)
- ARS
- Telomere
- long terminal repeat
- Centromere
- X element
- Intron
- ncRNA gene
- Noncoding exon
- tRNA gene
- snoRNA gene
- transposable element gene
- five prime UTR intron
- matrix attachment site
- snRNA gene
- rRNA gene
- external transcribed spacer region
- internal transcribed spacer region
- origin of replication
- telomerase RNA gene

The function returns 20 dictionaries, one for each feature type, and it creates a list with the names of all feature types (called `genomicregions_list`). In each dictionary the keys are the names of the features and the values consists of a single list with the following elements:

- feature type
- feature qualifier (`Verified` or `Dubious`)
- standard name
- aliases (when more than one, separated by `|`)
- parent feature name (typically 'chromosome ...')
- chromosome
- start coordinate (starting at 0 for each chromosome)
- end coordinate (starting at 0 for each chromosome)

The input of the function is a path to the SGD_features.tab file and in case of no input, this file is taken from the data_files folder.

samflag.py

This module determines the parameters based on the alignment flag found in sam files. It consists of single function:

-`samflag`

This takes an integer as input and outputs a list of parameters corresponding to that integer. This function executes the method described in the sam, bam section. It converts the integer to a 12-bit binary number and each of the 12 bits correspond to an entry in a list of parameters. The location of the ones (read from right to left) in the binary number determine which parameters are true given the input integer.

There are 2 outputs:

`flag_binary` which returns the 12-bit binary sequence of the input integer
`flagprop_list` which is a list containing all the parameters corresponding to the input integer.

This script is mainly used during the processing pipeline in the transposonmapping_satay.py and therefore this module is also present as a python module in the python_transposonmapping folder.

Data files

Cerevisiae_AllEssentialGenes_List.txt

This text file contains all known annotated essential genes in wild type according to SGD. It is created using create_essentialgenes_list.py where the genes are taken from both Cerevisiae_EssentialGenes_List_1.txt and Cerevisiae_EssentialGenes_List_2.txt. Both latter two files contain essential genes, but in different naming format and both contain genes not found in the other file. The Cerevisiae_AllEssentialGenes_List.txt file is a combination of the two files with all genes from both files and is therefore the most complete version. Each file contains a header with the source where the information is downloaded from.

S288C_reference_sequence_R64-2-1_20150113.fsa

This fasta file is the reference genome for yeast used for alignment and downloaded from yeastgenome.org. Note that before using a fasta file for alignment, the file has to be indexed. This can be done in Linux using the command `bwa index [path]/S288C_reference_sequence_R64-2-1_20150113.fsa` where `[path]` is replaced with the path to the reference fasta file.

SGD_features.tab

This file contains information for each genomic feature (e.g. genes, telomeres, centromers etc.) and was downloaded from SGD. It comes with a readme explaining the contents.

Saccharomyces_cerevisiae.R64-1-1.99.gff3

This file contains all information about the genes. The saccharomyces_cereivisae.gff.gz can be downloaded from SGD.

Note, the current downloadable file is a newer version with a slightly updated format. This is not integrated in the current python scripts, hence the old version is still used.

Yeast_Protein_Names.txt

This file is downloaded from uniprot and contains all known genes with all aliases and different naming conventions (i.e. designation, oln, swiss-prot and SGD cross-reference names). This file can be used for searching different namings for genes, for example using gene_names.py or genomicfeatures_dataframe.py. This file is still regularly updated on uniprot.

From this file, the files S_Cerevisiae_protein_designation_name_full_genome.txt and S_Cerevisiae_protein_oln_name_full_genome.txt are created using gene_names.py, each containing all genes in a consistent naming convention.

Other tools

IGV

The integrative Genomics Viewer is a tool to visualize the mapped reads and can be used for manually checking the output. When opening the tool for the first time, load a reference genome by going to `Genomes` in the task bar, click `Load genome from file` and select the reference fasta. Alternatively in the top bar, in the left most drop-down menu click `More...` and select the S. cerevisiae (sacCer3) genome (which comes standard with IGV). Next, select `File` from the task bar and click `Load from file`. Select a bam file which should have an index (.bam.bai) file stored at the same location. In the top of the window check if the right reference genome is loaded and select a chromosome to view (in the drop-down menu that says 'All' or a chromosome name). Zoom in to a region of interest which should then show all the reads present. Hovering the cursor over a read should give more information about that read. For example, this can be used to check the mapping quality of reads (MAPQ) and to check if the location of the reads correspond to the insertion locations stored in the output files from the workflow.

genome browser

Another useful tool the genome browser, which is an online tool for showing alignment data together with a reference genome. This also shows different features, for example where the genes are located. To use this tool, go to `My Data` in the top bar and select `My Sessions`. Here you should create an account if you haven't done this before. After this, go to the `Session Management` (on the same page) and at `Save Settings`, enter a name for your dataset at `Save current settings as named session`. Optionally you can set the checkbox `allow this session to be loaded by others` to give other people the possibility to see your data, but this is not required. Press `Submit`. While still on the same page, you should now see your dataset name in the section `My Sessions`. Click the name to open the browser. Here you should see an interactive figure with some default tracks. It may happen that not the right organism is selected (e.g. by default the Human genome is loaded), this will be resolved in the next step. Right below the browser, there are some functions for the browser. Click here the `add custom tracks` option. This should load a new page where you can select the right genome and add you own data. For loading the genome, select the `Other` in the drop-down menu next to `clade` on the top of the page. Then, select `S. cerevisiae` next to `genome` and then select the assembly `Apr. 2011`. On the same page, upload a bed or wig file next to `Paste URLs or data`. Note that for this the bed or wig files need to be cleaned using clean_bedwigfiles.py. Press `Submit` and after loading, press `Go` or add another track. This should now show you again the browser, but now with the right genome and your track(s) loaded. Note that by default the browser is zoomed in quite far, so you may want to zoom out somewhat or select a region or gene in the search bar.

This now allows you to see the insertions against all genomic features in the genome. Your tracks are saved automatically and when the `allow this session to be loaded by others` was selected, it can be shared with others.

How to use the Linux desktop

For those who are not familiar with Linux, here is a brief summary of how to work with the Linux desktop and the basic tools that are necessary for using the workflow. For a more thorough tutorial of the linux commandline and many other useful commands, see for example this course from datacarpentry.org/shell-genomics.

When using the Linux desktop, log in with the credentials found at `N:\tnw\BN\LL\Shared\LinuxMachines\LinuxDesktop_LoginCredentials.txt`. For remote access, download Teamviewer on your computer and connect with the Linux desktop using the credentials found in the file mentioned above. Enter the ID number in the 'partner ID', click connect and enter the password.

The workflow can be used via the commandline. For this, open the Terminal located in the left bar in Linux. This allows you to navigate through the different files and folders that are located on the desktop. These files and folders can also be accessed using Files (also located on the left bar) which gives a more Windows like experience with clicking on pictograms, but this is missing some useful features from the commandline including starting the workflow. The Terminal by default opens in the home directory (indicated either as `~/` or `/home/laanlab/`). For navigating through different directories (folders) use the change directory command `cd` (e.g. `cd ~/Documents/satay/` would bring you to the folder called satay which is located within the Documents folder). Everything needed for the satay analysis is located in `~/Documents/satay`, which contains the following folders (among others):

- `software`; containing all the software packages and the workflow script `satay.sh`.
- `datasets`; here the datasets are stored, including some test datasets.
- `reference_sequences`; containing the reference genome sequences.

While navigating through the folders, see what is inside a folder using `ls` (e.g. just typing `ls` shows the contents of the current directory and `ls ~/Documents/satay` would show the contents of the satay folder regardless in which folder you currently are). There are several extensions of this command, like `ls -a` which shows also all hidden folders (starting with `.`) and `ls -l` shows some basic information about each file and folder.

To show the contents of a file use `less` or `head`. The `head` command shows the first 10 lines of the file (e.g. `head ~/Documents/satay/datasets/singleendtestfolder/singleendtest.fastq` shows the first 10 lines of the fastq file singleendtest fastq file) and `less` also shows the contents of a file, but allows you to scroll through the file using the arrow keys. The advantage of these commands are that they only load the lines that are being shown, so this is a nice tool for checking large files without using a lot of memory (what would normally happen when opening a file completely). Also, the `less` command in Linux allows for seeing the contents of gzipped files without fully unzipping them (the `head` command only works with unzipped files).

To normally open a file with the default program for that filetype, use the command `xdg-open` (e.g. `xdg-open ~/Documents/satay/software/satay.sh` opens a text editor with satay.sh).

To access the webdrives, open the Files program and go to `Other Locations` (in the left menu bar in Files). On the bottom of the window it should say `Connect to server`. Enter here the following adress: `sftp://sftp.tudelft.nl/` Press `Connect` and, if requested, log in with your TUDelft credentials. This should show a number of folders, among of which is `staff-bulk` which is the N-drive and `staff-groups` which is the M-drive. Here you can down-/upload any files to and from the desktop, but after about 10 minutes of inactivity, the connection to the drives is automatically broken. This means that all the down-/uploads will be aborted. To prevent this, switch back and forth to some folders within the drives every few minutes to reset the timer.

The workflow can be accessed by going to the following path: `~/Documents/satay/software/` Start the workflow by stating the tool to which to run the script with (i.e. bash) and then the name of the script (satay.sh), thus entering `bash satay.sh`. Any commands you want to put in come right after this statement, e.g. `bash satay.sh -h` to open the help text. Datafiles can be stored at the following location: `~/Documents/satay/datasets/`. To keep things organized, create a new folder for each dataset using the command `mkdir ~/Documents/satay/datasets/[foldername]` (where `[foldername]` is to be replaced with any name you want). Prevent spaces in namegiving, but rather use underscores. (Alternatively, when using the Files program, right mouse click and select `New Folder` at the location where the new folder has to be located).

Note that this desktop has no backup system, thus make sure to always put any files directly on the drives and remove the files you do not need from the desktop to prevent filling up the memory. Removing files can be done using the command `rm` (e.g. `rm [pathtofile]/[filename]`). Note that this command does not have an undo and also the removed files are not moved to a trash folder. This command works immediately and is permanent, so use with caution and double check before you enter it. To remove folders including the contents, use `rm -r`. Alternatively, you can use the Files program to delete files, which does first move the files to a trash folder where you then can remove them permanently.

Once in a while, the desktop may require an update. Sometimes the desktop will ask for this and you can follow the instructions of the pop up window. To update manually enter the following commands in the commandline:

- `sudo apt-get update`; This checks for the latest versions of different system software tools and files. This command only updates the list of things to update, but doesn't do any downloading or installing itself. This requires a user password (required by the `sudo` command) which is the same password as for logging in. (Note that when typing a password in the commandline, it doesn't show any signs that you're typing, but just type the password and press enter).

- `sudo apt-get upgrade`; This does the actual downloading and installation of the updates that are found by the previous command. It may ask for a confirmation before it starts the installation and may restart the desktop.

To have commandline like tool on windows (e.g. to access large datafiles on windows as well using the `less` and `head` commands), try git bash. For Mac users, the terminal that is by default installed in MacOS works very similar as the one in Linux, so there is no real need for downloading any special tools.

Summary

The workflow discussed in this documentation processes the data from raw sequencing data to a list of insertions sites and read counts. Here a summary is given of the most important steps to take starting after retrieving the sequencing data.

1. Manually check the fastq files to see if the reads appear to be ok and if you can recognise adapter, primer or transposon sequences (if any) that need to be trimmed.
2. If needed, preprocess the data. For example demultiplexing when multiple samples were sequenced simultaneously. Store the data for each sample in individual fastq files, preferably as single-end reads.
3. Input the (preprocessed) fastq files in the workflow. For this, store the fastq files on the Linux desktop and run the workflow using the command `bash satay.sh` (located in the software folder).
4. When it is the first time processing the data, it is advised to set the `Quality checking raw data` and `Quality check interrupt` to `True`. This allows you to check the quality of the raw fastq files before the actual processing starts and to spot any artefacts or unwanted (overrepresented) sequences.
5. Run the processing workflow with trimming settings and alignment settings that suits the data. Determining the right settings is not always straightforward and often require some trial and error. It is advised to always trim away the unwanted sequences (e.g. adapter, primer and transposon sequences or overrepresented sequences found by the raw quality check) by copying those sequences to the adapters file. It is good to do some quality trimming for the last basepairs of each read as well (see quality report of the raw data how many basepairs would be needed, if any). For the alignment it is advised to set the settings a little more stringent compared with the default values (see the BWA MEM manual), but not too stringent as this can have a negative impact on the alignment.
6. After processing, check the output of each function to see how many reads were trimmed and aligned. If many cells were removed after trimming or when many reads were not aligned, you may have set some options too stringent. Also check the quality report of the trimmed reads and compare this with the quality report for the raw reads. Any overrepresented sequences that were entered in the adapters file are expected to be gone and the overall quality should have improved when trimming low quality basepairs was set.
7. When the processing is finished and the quality report and outputs all seem reasonable, check the resulting output files (i.e. bed, wig and multiple text files, see the Input, Output section). Preferably, run the bed and wig files in the clean_bedwigfiles.py to remove any insertions mapped outside the chromosomes. Run the transposonread_profileplot_genome.py script to check if the overall coverage of the insertions in the genome is good. There should be no large empty regions or large peaks, except for those around the centromeres of each chromosome, one large peak near the middle of chromosome 12 and a peak at the Ade2 gene in chromosome 15.
8. Next check the individual chromosomes using transposonread_profileplot.py. This generates similar figures as transposonread_profileplot.py, but this is on chromosome level and allows to check if the annotated essential gene have significant less insertions compared to the other regions. To check the

distribution of reads per gene and compare this between essential and non-essential genes, the scatterplot_genes.py script can be used.

9. When processing a mutant strain, also check if the deleted gene(s) are devoid of insertions as well. For this the pergene.txt file can be used or the gene can be checked in IGV or the genome browser.
10. For further analysis, the genomicfeatures_dataframe.py function can be used that contains the most important information from the processing files. This function can be integrated in other python scripts.

Links

Github page: github.com/leilaicruz/LaanLab-SATAY-DataAnalysis/tree/satay_processing

Jupyter notebook about this project: leilaicruz.github.io/SATAY-jupyter-book/Introduction.html

Laanlab: tudelft.nl/laanlab

Satay website: sites.google.com/site/satayusers/

Satay forum: groups.google.com/g/satayusers

eLife paper satay: elifesciences.org/articles/23570#content

Yeast genome: yeastgenome.org/

Yeast mine: yeastmine.yeastgenome.org/yeastmine/begin.do

Datacarpentry course about genomics (also contains a course in using the Linux bash and command line tools): datacarpentry.org/lessons/#genomics-workshop

Appendices

PHRED table (base33)

ASCII symbol	ASCII value	Q	P_err
!	33	0	1,000000
"	34	1	0,794328
#	35	2	0,630957
$	36	3	0,501187
%	37	4	0,398107
&	38	5	0,316228
'	39	6	0,251189
(	40	7	0,199526
)	41	8	0,158489
*	42	9	0,125893
+	43	10	0,100000

ASCII symbol	ASCII value	Q	P_err
,	44	11	0,079433
-	45	12	0,063096
.	46	13	0,050119
/	47	14	0,039811
0	48	15	0,031623
1	49	16	0,025119
2	50	17	0,019953
3	51	18	0,015849
4	52	19	0,012589
5	53	20	0,010000
6	54	21	0,007943
7	55	22	0,006310
8	56	23	0,005012
9	57	24	0,003981
:	58	25	0,003162
;	59	26	0,002512
<	60	27	0,001995
=	61	28	0,001585
>	62	29	0,001259
?	63	30	0,001000
@	64	31	0,000794
A	65	32	0,000631
B	66	33	0,000501
C	67	34	0,000398
D	68	35	0,000316
E	69	36	0,000251
F	70	37	0,000200
G	71	38	0,000158
H	72	39	0,000126
I	73	40	0,000100

ASCII symbol	ASCII value	Q	P_err
J	74	41	0,000079
K	75	42	0,000063
L	76	43	0,000050
M	77	44	0,000040
N	78	45	0,000032
O	79	46	0,000025
P	80	47	0,000020
Q	81	48	0,000016
R	82	49	0,000013
S	83	50	0,000010

PHRED table (base64)

ASCII symbol	ASCII value	Q	P_err
@	64	0	1,000000
A	65	1	0,794328
B	66	2	0,630957
C	67	3	0,501187
D	68	4	0,398107
E	69	5	0,316228
F	70	6	0,251189
G	71	7	0,199526
H	72	8	0,158489
I	73	9	0,125893
J	74	10	0,100000
K	75	11	0,079433
L	76	12	0,063096
M	77	13	0,050119
N	78	14	0,039811
O	79	15	0,031623
P	80	16	0,025119

ASCII symbol	ASCII value	Q	P_err
Q	81	17	0,019953
R	82	18	0,015849
S	83	19	0,012589
T	84	20	0,010000
U	85	21	0,007943
V	86	22	0,006310
W	87	23	0,005012
X	88	24	0,003981
Y	89	25	0,003162
Z	90	26	0,002512
[	91	27	0,001995
\	92	28	0,001585
]	93	29	0,001259
^	94	30	0,001000
_	95	31	0,000794
`	96	32	0,000631
a	97	33	0,000501
b	98	34	0,000398
c	99	35	0,000316
d	100	36	0,000251
e	101	37	0,000200
f	102	38	0,000158
g	103	39	0,000126
h	104	40	0,000100
i	105	41	0,000079
j	106	42	0,000063
k	107	43	0,000050
l	108	44	0,000040
m	109	45	0,000032
n	110	46	0,000025

ASCII symbol	ASCII value	Q	P_err
o	111	47	0,000020
p	112	48	0,000016
q	113	49	0,000013
r	114	50	0,000010

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