seqpac - A guide to sRNA analysis using sequence-based counts

Daniel Nätt

2021-12-10

Seqpac is available on github:

```
## Installation
devtools::install_github("Danis102/seqpac", upgrade="never",
                         build_manual=TRUE, build_vignettes=TRUE)
```

Introduction

1.1 Overview

Seqpac contains an extensive toolbox for the analysis of short sequenced reads. The package was originally developed for small RNA (sRNA) analysis, but can be applied on any type of data generated by large scale nucleotide sequencing, where the user wish to maintain sequence integrity throughout the whole analysis. This involves, for example, regular RNA-seq with long RNA. But note, however, that it currently does not automatically support paired-end sequencing.

To preserve sequence integrity, seqpac applies sequence-based counting. This can be contrasted against feature-based counting, where reads are counted over known genomic features, such protein coding genes or sRNA. This will result in a count table where each count is made per feature (e.g. number of read counts mapping to a gene). In seqpac, such annotations are initially disregarded, as a count table is generated solely by counting unique sequences in the raw sequence files (fastq). Annotations against known reference (such as databases containing sequences of full genomes, miRNA, tRNA, protein coding genes, or genomic coordinates of such features) are done after the count table has been generated.

The advantage of using sequence-based counting, compared to feature-based counting, is that for example mismatches in the classification of reads, can be applied any time during the analysis. In feature-based counting, allowing a mismatch in the mapping will result in a loss of information (loss of sequence integrity).

For example, say that you have 506 read counts for miR-185 (a miRNA) when allowing 3 mismatches against the reference. In feature-based analysis this will result in 1 entry in the count table. Unless you have saved the exact alignments elsewhere, you have lost the information about what sequences that are included in that count. The only thing you know is that the sequence may differ from the reference sequence by any 3 mismatches. In sequence-based counting, reads are counted as unique sequences and each unique sequence is then mapped against the target reference databases. Information about whether a sequence was aligning against a reference is maintained in a linked annotation table that can contain both perfect alignments and alignments allowing mismatches. Thus, in any stage of the analysis mismatches and classifications into features can be dynamically controlled, meaning that you can easily observe the effect of for example allowing more mismatches or changes in hierarchical classifications (often applied in sRNA analysis when a sequence align with multiple classes of sRNA).

But the best with sequence-based counting where you preserve sequence integrity, is probably that you will be able blast your candidate sequences at your favorite database, to verify and find out more about them.

1.2 The seqpac workflow

The input file format for seqpac is fastq, which can be generated from most sequencing platforms. Two types of objects sustain the core of the seqpac workflow:

- The PAC object
- The Targeting objects

The PAC object stores the Phenotypic information (P), Annotations (A) and Counts (C) needed for all downstream analysis, while the targeting object is used for accessing specific subsets of data within the PAC object.

Seqpac provides two versions of the PAC object. In its simplest form it is a S3 list containing 3 data.frames (Pheno, Anno and Counts). As default, however, an S4 PAC object is generated. By using the coercion method `as(pac-object, "list")`, you can turn an S4 PAC into an S3 list. Similarly, by using `as.PAC(pac-object)` you can turn an S3 PAC into an S4. Plaese see `?as.PAC` an `?PAC` for more details and examples.

The targeting object is always a list of 2 character objects.

The workflow is roughly divided into four steps:

- Generate a PAC object from fastq-files
- Preprocess the PAC object
- Annotate the PAC object
- Analyze the PAC object.

1.3 Example data

Contained within this package is one example fastq file. This file was originally generated by extracting sRNA from a single fruit fly embryo. The original fastq file was aquired by Illumina NextSeq 500 sequencer using a High Output 75 cycles flow cell (product no: 20024906) and NEBNext® Small RNA Library Prep Set for Illumina (E7330). Due to package size restriction, this fastq was randomly down-sampled to a fraction of its original size.

There is also a prepared example PAC-object that were similarly generated from randomly down-sampled fastq files (but here we used 9 unique embryos). In addition, there are a few fasta reference file. These files are used in examples of this vignette and in the function manuals.

Now, lets start by randomly sample the included fastq to obtain some multi-sample data:

```r
library(seqpac)

 # Use the ShortRead-package to randomly sample a fastq file
 sys_path = system.file("extdata", package = "seqpac", mustWork = TRUE)
 fq <- list.files(path = sys_path, pattern = "fastq", all.files = FALSE,
                 full.names = TRUE)

 sampler <- ShortRead::FastqSampler(fq, 20000)
 set.seed(123)
 fqs <- list(fq1=ShortRead::yield(sampler),
            fq2=ShortRead::yield(sampler),
            fq3=ShortRead::yield(sampler),
            fq4=ShortRead::yield(sampler),
            fq5=ShortRead::yield(sampler),
            fq6=ShortRead::yield(sampler))

 # Now generate a temp folder were we can store the fastq files
 # (you may of course use your own destination folder)
 # (path style depends on platform; autonomous example need empty folder)
 path_to_fastq <- paste0(tempdir(), "/seqpac_temp")
 if(grepl("windows", .Platform$OS.type)){
  path_to_fastq <- gsub( "\\\\", "/", path_to_fastq)
 }
 unlink(path_to_fastq, recursive = TRUE)
 dir.create(path_to_fastq, showWarnings=FALSE)

 # And then write the random fastq to the folder
 for (i in 1:length(fqs)){
  input_file <- paste0(path_to_fastq, "/", names(fqs)[i], ".fastq.gz")
  ShortRead::writeFastq(fqs[[i]], input_file, mode="w",
                        full=FALSE, compress=TRUE)
 }
```

Making a PAC object

2.1 Before making a PAC object (merge_lanes)

Seqpac works best with merged fastq, meaning that if sequencing was done in separate lanes on the flow cell, where the same sample was present in all lanes, lane files must be merged for each unique sample.

Conveniently, seqpac contains a function that can be used to merge lane files: `merge_lanes`. Please see, `?merge_lanes` for examples on how to merge fastq-files. Specifically, this function searches the input folder for similar file names and append files with similar names. As long as the user uses unique sample names (e.g. sample1_lane1.fastq.gz, sample1_lane2.fastq.gz, sample2_lane1.fastq.gz, sample2_lane2.fastq.gz etc) then `merge_lanes` will merge lane files into the same sample (e.g. sample1.fastq.gz, sample2.fastq.gz).

2.2 Generating a count table (make_counts)

Count tables are the core components in most sequence analysis. Seqpac uses sequence-based counts (see 1.1 Overview). Thus, we need to generate a count table with the number of unique sequences in each sample.

There are three ways of generating a count table in seqpac.

- Trim raw fastq files using internal functions in seqpac/R.
- Trim raw fastq files using external functions outside R.
- Import reads from already trimmed fastq files.

The two first involves adaptor trimming, while the last involves already trimmed reads. All input options are contained within the `make_counts`. function.

2.2.1 Internal trimming

Generating a count table from untrimmed fastq files using nothing but R packages (internal) depends on the `make_trim` function. This function goes through a series of trimming cycles using the `trimLRPatterns` function in `Biostrings` package to generate highly comparable adaptor trimming as generated by popular fastq trimming softwares, such as cutadapt.

Seqpac trimming is efficient, particularly when trimming many fastq files at the same time. It parallelizes trimming on multiple processor cores/threads by applying functions in the foreach package. Thus, if threads=12 and you input 12 fastq files, all files will be trimmed simultaneously in parallel. Note, however, that each thread will make a substantial impact on your computers memory performance. Thus, if you know that you are low in ram memory use fewer number of threads.

When make_trim is used on its own it results in trimmed fastq files stored at a user defined location. When trimming="seqpac" is set in the make_counts function, this function parses settings to make_trim to generate a count table from temporary stored trimmed fastq files.

```
## Using default settings for NEB type adaptor
# NEB= NEBNext® Small RNA Library Prep Set for Illumina (E7300/E7330)
# For illumina type adaptors, 'parse="default_illumina"' may be used

count_list <- make_counts(input=path_to_fastq, trimming="seqpac",
                          parse="default_neb")

# That was quite slow, lets speed it up a bit by increasing threads
# You may check how many threads you can run with "parallel::detectCores()"
count_list <- make_counts(input=path_to_fastq, trimming="seqpac",
                          parse="default_neb", threads=6)
```

2.2.2 External trimming

As an alternative to seqpac internal trimming, two external softwares, cutadapt and fastq_quality_filter, can be used to trim the adapter sequence and remove low quality reads prior of making a count table.

Importantly, cutadapt and fastq_quality_filter are not available on windows:
https://cutadapt.readthedocs.io/en/stable/installation.html
http://hannonlab.cshl.edu/fastx_toolkit/commandline.html

For linux users with all dependencies installed:

```
###-------------------------------------------------------------------
# Generate counts by parsing commands to cutadapt
# and fastq_quality_filter.
# (Note, the 'default_neb' parse argument is designed for sRNA trimming)

count_list <- make_counts(input=path_to_fastq,
                          trimming="cutadapt", parse="default_neb")
```

See the manuals for 'make_counts', 'make_trim', 'make_cutadapt' for more details.

2.2.3 Already trimmed fastq

Setting `trimming=NULL` in the `make_counts` function will pass the adapter trimming and a count table will be generated directly from the fastq files. Thus, this option can be used if you already have trimmed your fastq files.

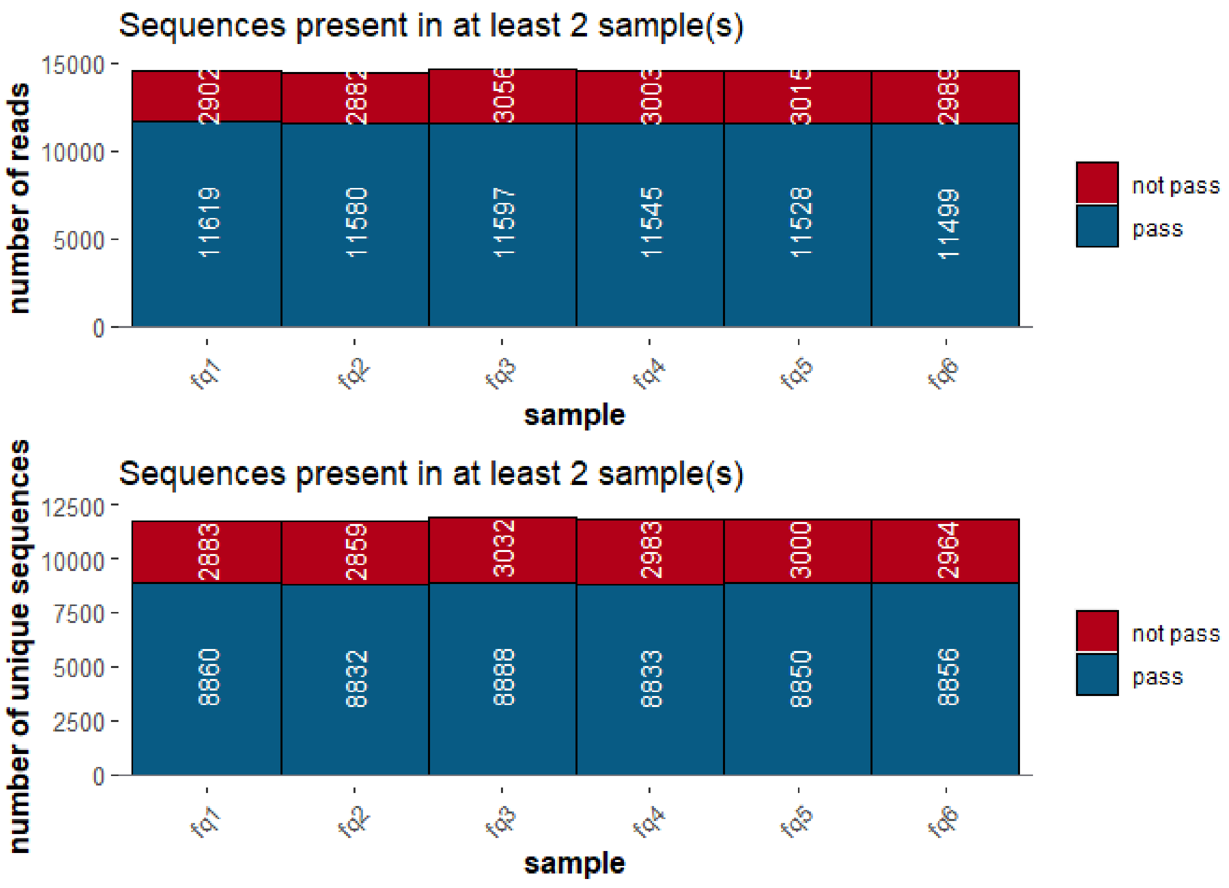
2.2.4 Outputs from make_counts

Besides the counts table, `make_counts` automatically generates a progress report for the trimming and evidence filter, as well as bar graphs showing the impact of the evidence filter (if plots=TRUE).

2.2.5 Evidence filtering

Users may already when making the counts table with `make_counts` markedly reduce the noise in the data by specifying the number of independent evidence that a specific sequence must reach to be included. This is controlled by the `evidence` argument. As default, `evidence=c(experiment=2, sample=1)` will include all sequences that have >=1 counts in at least 2 independent fastq files.

```
## Lets plot the graphs from the default output
cowplot::plot_grid(plotlist=count_list$evidence_plot, nrow=2, ncol=1)
```



Importantly, since the test data is heavily down-sampled compared to the fastq file they originated from, the default evidence filter results in a rather dramatic loss of total number of reads. This will not be the case when running good quality fastq files with saturated sequence depths. We routinely observe that 95-99% of all reads are maintained when applying the default evidence filter, while >50% of the unique sequences will be dropped because they can not be replicated across two independent samples. For an example of how a saturated experiment may look like, see the original seqpac paper, Skog et al. from 2021.

As you may have guessed, the 'experiment' argument controls the number of independent fastq evidence needed across the whole experiment. Note, however, that 'sample' does not control the number of counts needed in each sample. The evidence filter will always use >=1 counts in X number of fastq files. Instead 'sample' controls when a sequence should be included despite not reaching the 'experiment' threshold. Thus, if `evidence=c(experiment=2, sample=10)`, sequences that reach 10 counts in a single sample will also be included. Here is an example on how to use the 'sample' argument:

```
###-----------------------------------------------------------------
## Evidence over two independent samples, saving single sample sequences
## reaching 10 counts (remember, these are very down-sampled example fastq).
test <- make_counts(input=path_to_fastq, trimming="seqpac",
                    parse="default_neb", threads=6,
                    evidence=c(experiment=2, sample=10))
extras <- apply(test$counts, 1, function(x){sum(!x==0)})
table(extras==1) # None have 10 alone

## Evidence over two independent samples, saving single sample sequences
## reaching 3 counts
test <- make_counts(input=path_to_fastq, trimming="seqpac",
                    parse="default_neb", threads=6,
                    evidence=c(experiment=2, sample=3))
extras <- apply(test$counts, 1, function(x){sum(!x==0)})
test$counts[extras==1,] # A few still have 3 alone
```

Lastly, if `evidence=NULL` all unique sequences will be saved. Remember, however, that the count table will become larger in well saturated experiments that may lead to performance issues on some systems, since each unique sequence occupies a row in `PAC$Counts` and `PAC$Anno`. Apply other filters prior to annotating your sequences may solve such problems.

2.3 Generate the phenotype table (make_pheno)

Next, we generate a Pheno table (P) and merge it with the progress report that was stored from the `make_counts` output. The Pheno table contains sample specific information such as sample name and possible interventions, with each sample as a row.

The `make_pheno` function can import a Pheno table both externally by providing a path to a .cvs file, as well as internally by just passing a data.frame. In common for both options, is that a unique column named "Sample_ID" needs to be present, and must contain the sample names (column names) present in the counts table generated by `make_counts`. Now, `make_pheno` will attempt to match similar names, but to avoid problems use identical names.

In addition, `make_pheno` may add the progress report generated by `make_counts`. Lets have a look at how this works:

```
###---------------------------------------------------------------------
## Generate a Pheno table using the file names of test fastq
Sample_ID <- colnames(count_list$counts)
pheno_fl <- data.frame(Sample_ID=Sample_ID,
                    Treatment=c(rep("heat", times=3),
                                rep("control", times=3)),
                    Batch=rep(c("1", "2", "3"), times=2))

pheno <- make_pheno(pheno=pheno_fl,
                    progress_report=count_list$progress_report,
                    counts=count_list$counts)
```

2.4 make_PAC

Finally, we put everything together as a master PAC object using the `make_PAC` function. Conveniently, skipping the anno= argument will automatically add a very simple annotation table to the PAC-object that can be filled later.

```
###---------------------------------------------------------------------
## Generate PAC object (default S4)
pac_master <- make_PAC(pheno=pheno, counts=count_list$counts)
pac_master
names(pac_master)
isS4(pac_master)

lapply(as(pac_master, "list"), head)

## If TRUE the PAC object is ok
PAC_check(pac_master)
```

Preprocessing the PAC object

3.1 Targeting objects

Seqpac uses a simple targeting system to divide and focus the analysis of a PAC object to certain sample groups (Pheno) or specific sequences (Anno). If not specified otherwise in the function manual, a targeting object is always a two item list, where the first object is naming a column in a specific table in the PAC and the second object tells the function which categories in that column that should be targeted.

- `pheno_target`: extracts samples from a column in PAC$Pheno (S4: `pheno(PAC)`)
- `anno_target`: extracts sequences from a column in PAC$Anno (S4: `anno(PAC)`)

While `pheno_target` and `anno_target` are the most common targeting objects, there are other targeting object. Common for all is that they target a specific sub-table in the PAC.

Now, lets take a look at the provided example PAC-object in Seqpac. As noted by the name, this PAC has already been filtered and annotated, but it will still show you the principle of preprocessing using targeting objects.

```
###-----------------------------------------------------------------------
## Load the PAC-object  and inspect the columns in Pheno and Anno
library(seqpac)
load(system.file("extdata", "drosophila_sRNA_pac_filt_anno.Rdata", 
                 package = "seqpac", mustWork = TRUE))

pac$Pheno[,1:5]
pac$Anno[1:5,1:5]

## This PAC is an S3 object (a list with data.frames), but you can turn it to 
S4
## using as.PAC(), which gives you some additional functionality:
isS4(pac)
pac_s4 <- as.PAC(pac)
isS4(pac_s4)
pheno(pac_s4)
colnames(pac_s4)
head(anno(pac_s4))

methods(class="PAC") #all S4 methods
```

A pheno_target object that will exclusively target stage 1 and 5 samples would look like this: `pheno_target=list("stage", c("Stage1", "Stage5"))`

While an anno_target that will target sequences of 22 nt in length would look like this: `anno_target=list("Size", "22")`

Notice that both targeting objects are lists holding two secondary objects: one character string targeting a specific column, and one character vector targeting specific categories within the target column. The only difference being that each targeting object target different tables in the PAC object.

Important, if the second object is set to NULL or is left out this will automatically target all samples in the specified column. If the anno_target is a character vector (not a list) some function will attempt to extract matches in the row names of either Pheno or Anno (see example with `PAC_filtsep` below).For more, please refer to the original Seqpac publication: Skog et al. 2021.

3.2 Filtering

A master PAC object will contain sequences in your original fastq files that passed the trimming and the initial evidence filter. Now, in many experiments this still involves millions of unique sequences, making the master PAC difficult to work with. Thus, it is often wise to further reduce noise prior to normalization and annotation.

Seqpac contains two functions for this purpose:

- `PAC_filter`
- `PAC_filtsep`

While `PAC_filter` gives a variety of choices to subset the PAC object according to for example sequence size or minimum counts in a certain proportion of the samples,`PAC_filtsep` extracts the sequence names that pass a filter within sample groups.

Here are some examples:

```
 ###-----------------------------------------------------------------------
## Extracts all sequences between 20-30 nt in length with at least 5 counts
## in 20% of all samples.
pac_filt <- PAC_filter(pac, size=c(20,30), threshold=5,
                          coverage=20, norm = "counts",
                          pheno_target=NULL, anno_target=NULL)
#> -- Size filter will retain: 3416 of 9131 seqs.
#> -- Count filter was specified.
#> -- The chosen filters will retain: 826 of 9131 seqs.

###-----------------------------------------------------------------------
## Optionally, a simple coverage/threshold graph at different filter depths
## can be plotted with stat=TRUE (but it will promt you for a question).
pac_filt <- PAC_filter(pac, threshold=5, coverage=20,
                       norm = "counts", stat=TRUE,
                       pheno_target=NULL, anno_target=NULL)

###-----------------------------------------------------------------------
## Extracts all sequences with 22 nt size and the samples in Batch1 and Batch
2.
```

```
pac_filt <- PAC_filter(pac, subset_only = TRUE,
                       pheno_target=list("batch", c("Batch1", "Batch2")),
                       anno_target=list("Size", "22"))
#> -- Pheno target was specified, will retain: 6 of 9 samples.
#> -- Anno target was specified, will retain: 378 of 9131 seqs.


###-----------------------------------------------------------------------
## Extracts sequences with >=5 counts in 100% of samples within each stage
filtsep <- PAC_filtsep(pac, norm="counts", threshold=5,
                       coverage=100, pheno_target= list("stage"))

pac_filt <- PAC_filter(pac, subset_only = TRUE,
                       anno_target= unique(do.call("c", as.list(filtsep))))
#> -- Anno target was specified, will retain: 1093 of 9131 seqs.
```

Notice that `PAC_filtsep` only extracts the sequence names and stores them in a data.frame. Converted into a character vector with unique names (using `unique` + `do.call`) they can be applied as an anno_target that targets sequence names in Anno.

Hint, the data.frame output of `PAC_filtsep` has been designed for the purpose of generating Venn and Upset diagrams that visualize overlaps across groups:

```
###-----------------------------------------------------------------------
## Venn diagram using the venneuler package
olap <- reshape2::melt(filtsep,
                       measure.vars = c("Stage1", "Stage3", "Stage5"),
                       na.rm=TRUE)
plot(venneuler::venneuler(olap[,c(2,1)]))

###-----------------------------------------------------------------------
## Setting output="binary", prepares data for upset plots (UpSetR package):
filtsep_bin <- PAC_filtsep(pac, norm="counts", threshold=5,
                           coverage=100, pheno_target= list("stage"),
                           output="binary")

UpSetR::upset(filtsep_bin, sets = colnames(filtsep_bin),
                           mb.ratio = c(0.55, 0.45), order.by = "freq",
                           keep.order=TRUE)
```

3.3 Storage of processed data in PAC

While the simplest PAC only contains three data.frame objects (P, A and C), additional ‘folders’ will be stored in the PAC object as the analysis progresses. In fact, if using a S3 PAC (list), you may choose to store any number of objects in your PAC as long as they do not conflict with the names of the standard folders, which are:

```
Pheno  (P):    data.frame
Anno   (A):    data.frame
Counts (C):    data.frame
```

```
norm:           list of data.frames
summary:        list of data.frames
```

The *norm* folder will be used to store counts that have been normalized in different ways, while the *summary* folder will contain the data that has been summarized across groups of samples. In both cases, applying a filter using `PAC_filter` will automatically subset not only P, A, and C, but also all tables stored in `PAC$norm` and `PAC$summary`.

3.4 Normalization

The built in normalization methods in seqpac are handled by the `PAC_norm` function. In the current version three methods are available:

- Counts per million reads (cpm)
- Variance-Stabalizing Transformation (vst)
- Regularized log transformation (rlog)

More specifically, cpm normalize the dataset in relation to the total number of reads, while vst and rlog uses both mean dispersion and size factor to produce homoskedastic values with functions available in the DESeq2 package.

It is easy to generate your own normalized count table using your method of choice, and import it as a data.frame to the *norm* folder in the PAC object. Just remember that row and column names must be identical to the original Counts table. Use `PAC_check` to verify.

```
###---------------------------------------------------------------------
## Example normalization in seqpac
load(system.file("extdata", "drosophila_sRNA_pac_filt_anno.Rdata",
                 package = "seqpac", mustWork = TRUE))
pac <- pac[1:3] # remove norm 'folder' in S3 PAC

# A reminder on how to convert between S3 and S4:
pac_s4 <- as.PAC(pac)
pac_s3 <- as(pac, "list")


# PAC_norm works on both
pac_cpm <- PAC_norm(pac_s4, norm="cpm")
pac_vst <- PAC_norm(pac_s3, norm="vst")
```

From this, it may sometimes be wise to run further filtering steps to focus on reads with high cpm values.

```
###---------------------------------------------------------------------
## Filtering using cpm instead of raw counts
## filter >=100 cpm in 100% of samples in >= 1 full group
filtsep <- PAC_filtsep(pac_cpm, norm="cpm", threshold=100, coverage=100,
                       pheno_target= list("stage"))
pac_cpm_filt <- PAC_filter(pac_cpm, subset_only = TRUE,
                       anno_target= unique(do.call("c", as.list(filtsep))))
```

Generate annotations

From the start a PAC object contains a very limited Anno table such as this:

```
load(system.file("extdata", "drosophila_sRNA_pac_filt_anno.Rdata",
                 package = "seqpac", mustWork = TRUE))
pac$Anno <- pac$Anno[,1, drop = FALSE]
head(pac$Anno)
```

To facilitate easy adoption of seqpac to any species, we provide a fast and flexible workflow for mapping PAC sequences against user defined reference databases. We call this reannotation, since we map the sequences in Anno object sequentially over one or multiple references.

Important, seqpac relies on the popular bowtie aligner (R package Rbowtie) for mapping: http://bowtie-bio.sourceforge.net/manual.shtml. This means that references must be indexed using the `bowtie_build` function before they are compatible with the reannotation workflow (see below).

4.1 Fasta references

Small RNA annotation/mapping depends heavily on choosing your references wisely and not to be too greedy if possible. If your hypothesis target a specific type of class (like tRNA), it might for instance be wiser to target such references only.

Many databases provide their references in the fasta file format. Seqpac can use any correctly formated fasta file as input in the reannotation workflow. However, we distinguish two kinds of fasta references:

- Reference genomes.
- Other specialized references.

You can download the files manually from the source database. Below are some helpful example links where to retrieve useful fasta references. Note that all compressed files must be uncompressed to work with the bowtie aligner.

Reference genomes: Ensembl Animals Ensembl Plants UCSC iGenomes

Hint: Unmasked top level fasta files are best for most purposes, e.g. Ensembl file name *.dna.toplevel.fa.gz.

Other specialized references: miRNA (mirbase) miRNA (mirgenedb) piRNA (piRBase) Several ncRNA (Ensembl Animals) Several ncRNA (Ensembl Plants) tRNA (GtRNAbd) repeats (repeatMasker)

There are also several packages in Bioconductor/R such as BSgenome and biomaRt that allow you to download reference genomes and other specialized references directly into r. Actually, using the `download.file` function makes all urls (https/http/ftp etc) available for download. Custom fasta files can also be used, and you can easily merge, add and remove features to the references by functions in the Biostrings package, such as `readDNAStringSet` (read fasta) and `writeXStringSet` (save fasta).

As default, bowtie only reports the names up to the first white space. Sometimes the sequence names need to be modified or rearranged for the bowtie aligner to best report the names in the mapping output. This can be done using the Biostrings package to read, rearrange and save the fasta. A useful expression to verify that you have caught the relevant info before the the first white space is: `do.call("rbind", strsplit(names(<your_DNAStringSet>), " "))[,1]`

You may also want to add tRNAs from the mitochrondrial genome to your tRNA reference, which is not always provided in the GtRNAdb fasta. This can be done by moving them from Ensembl ncRNA fasta or scan the mitochondrial genome manually at tRNAscan-SE. The mitochonridal genome is found in the reference genome fasta.

## 4.2 Indexing the fasta references Before we can use the fasta references we need to index them for the bowtie aligner. You can do this by using the `bowtie_build` function in the Rbowtie package or by running bowtie externally, outside R. For more information see ?Rbowtie::bowtie, Rbowtie::bowtie_build_usage() and http://bowtie-bio.sourceforge.net/manual.shtml.

Important, the bowtie prefix must have the same basename (`prefix=`) and the indexes must be saved in the same directory (`outdir=`) as the original fasta file. Since the `bowtie_build` naming is often confusing for newbies, here are some examples:

```
###-----------------------------------------------------------------------
## Examples of generating bowtie indexes

path <- "/some/path/to/your/folder"
Rbowtie::bowtie_build(paste0(path, "/biomartr_genome/chromsomes.fa"),
                      outdir=paste0(path, "/biomartr_genome/"),
                      prefix="chromosomes", force=TRUE)
Rbowtie::bowtie_build(paste0(path, "/GtRNAdb/trna.fa"),
                      outdir=paste0(path, "/GtRNAdb/"),
                      prefix="trna", force=TRUE)
Rbowtie::bowtie_build(paste0(path, "/mirbase/mirna.fa"),
                      outdir=paste0(path, "/mirbase/"),
                      prefix="mirna", force=TRUE)
```

Building indexes may take some time for large fasta, such as a reference genome. You only need to generate the index once for every fasta reference, however.

4.3 GTF feature coordinates

After the sequences have been mapped against a reference genome they can also acquire annotations from genomic feature files, such as the gtf and gff formats. These files contains the coordinates for different genomic features associated with a specific reference genome, such as repetitive regions, exons and CpG islands etc.

GTF files can often be acquired at the same places as the fasta reference files (see above). The biomartr package also have functions for downloading some gtf files (e.g. `getGTF`).In addition, you may create your own by downloading/importing different tables, for example using rtracklayer (see `readGFF`, `getTable`, `ucscTableQuery`) and then create a GRanges object (GenomicRanges package) where you add your preferred table info. Finally you may save it using rtracklayer again (`export(<GRanges object>, <destination_path>, format="gtf"`)

4.4 The reannotation workflow

The procedure to annotate sequences in seqpac is called reannotation.

The PAC reannotation family of functions carries out the following tasks:

- Maps sequences against fasta references
- Stores the output on hard drive
- Reads selected parts into R
- Creates a reanno object
- Generates an overview from the reanno object
- Classifies and simplifies annotations
- Adds classifications to a PAC object

4.4.1 The functions in the reanno workflow

map_reanno To accommodate most users on most platforms, seqpac provides two alternatives for calling bowtie using the `map_reanno` function: internally from within R (`type="internal"`) or externally from outside R (`type="external"`). In the internal mode, seqpac uses the `bowtie` function in the Rbowtie package, while in the external mode bowtie is called by a system command to a locally installed version of bowtie. Both options gives identical results, but since the internal Rbowtie package is rarely updated, we provide the external option.

Since the input commands for external bowtie and internal Rbowtie differs slightly, `map_reanno` has two parse options: `parse_internal` and `parse_external`. These can be used to control the two versions of bowtie as if they were run from the console or command line, respectively.

While bowtie has its own way to handle mismatches in the alignments, seqpac disregards this and instead runs bowtie sequentially, increasing the number of mismatches for each cycle. After each cycle all sequences that received a match in any of the provided references will be subtracted. Therefore, in the next cycle only sequences without a match will be 'reannotated' but allowing one additional mismatch. The `map_reanno` function controls this by sequentially increasing the number of mismatches after subtracting the matches.

import_reanno This function is called by `map_reanno` to import the bowtie results after each cycle. If you would run `import_reanno` separately it simply provides you with options on how to import bowtie results into R. For example, when aligning against a reference genome the mapping coordinates is important, while mapping against more specialized references used for creating sRNA classifications (rRNA, tRNA, miRNA etc), 'hit' or 'no hit' might be enough, and will be a much faster option. The `map_reanno` function, therefore, has two default import modes, import="genome" and import="biotype", which automatically configure `import_reanno` for genome or class annotation.

make_reanno After each annotation cycle, `map_reanno` will store an .Rdata file (Full_reanno_mis0/1/2/3.Rdata) containing a list with all alignment hits for each PAC-sequence-to-fasta-reference mismatch cycle. Calling `make_reanno` will search for these .Rdata files in a given folder and create a reanno object in R. This involves extracting, ordering and preparing annotations in a format that can be merged with the existing Anno table in a PAC object. It will also generate an overview table, that summarizes the annotations if multiple fasta references were used by `map_reanno`.

add_reanno Before merging the new annotations with a PAC object, unless we have provided a fasta reference with only one type (e.g. miRNA for mirBase), classification not only *between* fasta references but also *within* references might be necessary (e.g. snoRNA, tRNA, miRNA, rRNA in the Ensembl_ncRNA fasta). Classification of the fasta reference names, which was first saved by bowtie and then caught in our reanno object as an annotation, is done with the `add_reanno` function. Again, there are two primary modes, either "genome" or "biotype". When type="genome" a standardized workflow anticipating genomic coordinates are applied. When type="biotype" a list of search term vectors directed against each fasta reference needs to be provided. Matches between search term (written as regular expressions) and text strings in the fasta reference names will result in a classification of the PAC sequence. Finally, `add_reanno` generates an annotation table that can either be saved separately or merged with the original PAC object.

simplify_reanno While `add_reanno` starts the process of simplifying the reference name annotations into useful classifications, it will generate multiple classifications for each counted sequence if it matches multiple search terms. The `simplify_reanno` function boils down multi-classifications into one classification for each sequence. By doing so, an hierarchy must be set, where priority of some classifications over others is done (e.g. miRNA over piRNA). While this is a controversial topic, using such hierarchies is still the only way to provide overview statistics in experiments targeting highly multimapping sequences, such as small RNA. Nonetheless, the seqpac workflow with its sequence based counts, where hierarchical classification is done in the very end of the annotation process, makes it easy to generate classifications using alternative hierarchies and observe the consequences of your choices in for example a pie chart.

In summary:

- **map_reanno** Bowtie reference mapping over mismatch cycles
- **import_reanno** Used by `map_reanno` to import bowtie output

- **make_reanno** Organizes `map_reanno` output into a reanno object
- **add_reanno** Classifies annotations according to reference name
- **simplify_reanno** Hierarchical classification

Already at this point it is good to know that seqpac provides a 'backdoor' function, `PAC_mapper`, that carries out many of the steps in the reanno workflow using smaller references and pac objects. Please see '6.1 Advanced alignment analysis' or ?PAC_mapper for more details.

4.4.2 Reannotation in action

Lets observe the functions in action using the drosophila test dataset.

First we map against a reference genome, using `map_reanno`. Note that import="genome" must be set to catch the genomic coordinates. Since `map_reanno` can handle multiple fasta references. Remember, each fasta must have bowtie indexes (see 4.2).

```
# Lets reload our pac
library(seqpac)
load(system.file("extdata", "drosophila_sRNA_pac_filt_anno.Rdata",
                 package = "seqpac", mustWork = TRUE))
pac$Anno <- pac$Anno[,1, drop = FALSE] # Remove previous Anno
```

```
###-----------------------------------------------------------------------
## Genome mapping

# Path to your output folder
outpath_genome <- "/some/path/to/reanno_folder"
# or a temp folder
outpath_genome <- paste0(tempdir(), "/seqpac_reanno/reanno_genome")

# Look up the your own Bowtie indexed reference genomes (fasta)
ref_paths <- list(myco="<some/path/to/your/genome.fa>")

# For testing, you can use a small mycoplasma genome provided with seqpac
# But remember to provide single genomes as a named list anyway.
# The name will be used to keep track of your genome.
myco_path <- system.file("extdata/mycoplasma_genome", "mycoplasma.fa",
                         package = "seqpac", mustWork = TRUE)
ref_paths <- list(myco=myco_path)

# Run map_reanno
map_reanno(PAC=pac, ref_paths=ref_paths, output_path=outpath_genome,
           type="internal", mismatches=3, import="genome",
           threads=8, keep_temp=TRUE)
```

Similarly, we can map against other specialized references, that can be used for classifying each sequence into biotypes. Note, unlike genome mapping, at this stage we are not interested in where a sequence matches a reference for the specialized references, only if it matches or not. Thus, we use `import="biotype"`.

```
###-----------------------------------------------------------------------
## sRNA mapping using internal bowtie

# Path to output folder
outpath_biotype <- "/some/path/to/reanno_biotype"
# or a temp folder
outpath_biotype <- paste0(tempdir(), "/seqpac_reanno/reanno_biotype")


# seqpac contains two fasta for tRNA and rRNA we can use for testing
trna_path <- system.file("extdata/trna", "tRNA.fa",
                         package = "seqpac", mustWork = TRUE)
rrna_path <- system.file("extdata/rrna", "rRNA.fa",
                           package = "seqpac", mustWork = TRUE)

# Provide paths to bowtie indexed fasta references as a list
ref_paths <- list(tRNA=trna_path,
                  rRNA=rrna_path)
```

```
map_reanno(pac, ref_paths=ref_paths, output_path=outpath_biotype,
           type="internal", mismatches=3,  import="biotype",
           threads=6, keep_temp=TRUE)
```

Now, the output .Rdata files for all mismatch cycles should have been stored in the output folders. Lets import everything into R, generate a reanno objects and plot some pie charts from the Overview table.

```

###-----------------------------------------------------------------------
## Generate a reanno object with make_reanno
reanno_genome <- make_reanno(outpath_genome, PAC=pac, mis_fasta_check = TRUE)
reanno_biotype <- make_reanno(outpath_biotype, PAC=pac, mis_fasta_check = TRU
E)

# Note, setting mis_fasta_check=TRUE will double check that the number of
# sequences that failed to recieve an alignment in the last mismatch cycle
# agrees with the number sequences in the reanno object without an annotation
.
# (these sequences are stored in mis_fasta_x.txt where x is max mismatches+1)

# List structure
str(reanno_genome, max.level = 3, give.attr = FALSE)
str(reanno_biotype, max.level = 3, give.attr = FALSE)

# Sometimes its more convenient to using S3 than S4
# (or you may use @ to obtain s4 slot)
reanno_bio_s3 <- as(reanno_biotype, "list")

# Simple pie charts using the Overview table
pie(table(reanno_genome@Overview$myco)) # Very few mycoplasma with 0 mismatch
pie(table(reanno_bio_s3$Overview$Any)) # Many hits for either rRNA or tRNA
pie(table(overview(reanno_biotype)$Any)) # S4 provides other receiver functio
ns
```

Next step is to reorganize and classify using `add_reanno`. For mapping against a reference genome this is easy. Just set type=“genome” and set the maximum number of alignments to be reported by `genome_max` (Warning: `genome_max="all"` will give you all alignments).

```
###-----------------------------------------------------------------------
### Genomic coordinates using add_reanno

# Output as separate table
anno_genome <- add_reanno(reanno_genome, type="genome",
                          genome_max=10, mismatches=3)

# Output merged with provided PAC object
pac <- add_reanno(reanno_genome, type="genome",
                  genome_max=10, mismatches=3, merge_pac=pac)

```

```
# Example of original reference name annotations
head(full(reanno_genome)$mis3$myco)

# Finished genome annotation
head(anno_genome)
head(pac$Anno)
```

For specialized references, `type="biotype"` should be used. This allows for classification based on match or no match between the search terms provided in the `bio_search` input and the reference names annotated with each counted sequence. Correct classification is all about finding the best search terms that discriminates between the different names in the original fasta reference files (up to the first white space; see 4.1).

With the `bio_perfect` option you may control how conservative the matching between search term and annotation should be. With `bio_perfect=FALSE`, every reference hit that fail to match your search terms, will be classified as 'other'. Using `bio_perfect=TRUE` will instead guarantee that your search terms will cover all reference hits.

Hint: See `?add_reanno` for a trick on how to succeed with `bio_perfect=TRUE`.

```
###---------------------------------------------------------------------
## Classify sequences using add_reanno

# Lets start by exploring the names in the original fasta reference:
ref_path <- "/some/path/to/fasta_reference"
# For testing use the the fasta reference for tRNA provided with seqpac:
trna_path <- system.file("extdata/trna", "tRNA.fa",
                         package = "seqpac", mustWork = TRUE)

#  Read fasta names
fasta <- names(Biostrings::readDNAStringSet(trna_path))
# Different naming standards
table(substr(fasta, 1, 10))
# Starting with FBgn discriminate mitochondrial tRNA
fasta[grepl("FBgn", fasta)]
# The other are nuclear
head(fasta[!grepl("FBgn", fasta)])

# We can use as 'mt:tRNA' as search term to catch mitochondrial tRNA
# and '_tRNA' to catch nuclear.

# A useful expression for extracting strings up to 1st white space is
# fasta <- do.call("rbind", strsplit(fasta, " "))[,1]

# Similarly load the rrna reference provided with seqpaq
rrna_path <- system.file("extdata/rrna", "rRNA.fa",
                         package = "seqpac", mustWork = TRUE)
#  Read fasta names
fasta <- names(Biostrings::readDNAStringSet(rrna_path))
```

```
# Many rRNA subtypes
table(substr(fasta, 1, 10))

# Lets try two search term lists directed against each reference and written
as regular expressions:

# Names in the search list needs to be the same as in the reanno object
head(overview(reanno_biotype)) # 'rRNA' and 'tRNA' with some capital letters

# Generate a search list with  search terms
bio_search <- list(
                rRNA=c("5S", "5.8S", "12S", "16S", "18S", "28S", "pre_S45"),
                tRNA =c("_tRNA", "mt:tRNA")
                    )

test <- add_reanno(reanno_biotype, bio_search=bio_search,
                      type="biotype", bio_perfect=FALSE,
                      mismatches = 2, merge_pac=pac)

# Throws an error because perfect matching was required:
anno_temp <- add_reanno(reanno_biotype, bio_search=bio_search,
                        type="biotype", bio_perfect=TRUE, mismatches = 3)

# References with no search term hits are classified as "Other":
anno_temp <- add_reanno(reanno_biotype, bio_search=bio_search,
                        type="biotype", bio_perfect=FALSE, mismatches = 3)

# Increasing number of hits allowing mismatches
table(anno_temp$mis0)
table(anno_temp$mis3)

# You may also add your new annotaion with a PAC object using the 'merge_pac'
# option. For even more options see ?add_reanno.
pac <- add_reanno(reanno_biotype, bio_search=bio_search,
                        type="biotype", bio_perfect=FALSE, mismatches = 3,
                        merge_pac=pac)
head(pac$Anno)#S3
head(anno(as.PAC(pac))) #S4
```

To make overview statistics and graphs we need to boil down the classifications to a factor column with only a few unique biotypes per factor. This is what `simplify_reanno` does. Just like `add_reanno` in the `type="biotype"` mode, `simplify_reanno` needs a list of search terms, an `hierarchy`. This time, however, the list is order sensitive and targets the output table from `add_reanno` that contains the columns with new classifications (e.g. "mis0_bio", "mis1_bio", "mis2_bio" etc). If a match occurs between the first search term and a classification, the counted sequence will be annotated to this classification, no matter if other search terms matches further down the list. This is done sequentially, allowing one additional mismatch until a maximum (specified in `mismatches`) has been reached. Thus, if a match occurs in tRNA with 0 mismatch, and rRNA with 1 mismatch, it will be reported as tRNA even though rRNA where higher in the hierarchy. A better match will always be prioritized over the hierarchy.

```
###---------------------------------------------------------------------
## Hierarchical classification with simplify_reanno

#  Currently classification does not discriminate
table(pac$Anno$mis3_bio)

# Set the hierarchy and remember that it is order sensitive.
# Here: S5_rRNA >> Other_rRNA >> mt:tRNA >> tRNA
# Remember to use 'regular expressions' if you wish to catch all:
hierarchy_1 <- list(rRNA_5S="rRNA_5S",
                    Other_rRNA="5.8S|12S|16S|18S|28S|pre_S45|rRNA_other",
                    Mt_tRNA="tRNA_mt:tRNA",
                    tRNA="tRNA__tRNA"
                    )

# What happens if you don't catch all:
hierarchy_2 <- list(rRNA_5S="rRNA_5S",
                    Mt_tRNA="tRNA_mt:tRNA",
                    tRNA="tRNA__tRNA")

# No mismatch allowed using hierarchy_1
test <- simplify_reanno(input=pac, hierarchy=hierarchy_1, mismatches=0,
                        bio_name="Biotypes_mis0", merge_pac=FALSE)
table(test) # All remaining rRNA are classified as 'Other_rRNA'

# Instead using hierarchy_2
test <- simplify_reanno(input=pac, hierarchy=hierarchy_2, mismatches=0,
                        bio_name="Biotypes_mis0", merge_pac=FALSE)
table(test)# Non-hits are classified as 'other'

# Note, setting 'merge_pac=FALSE' returns a one-column data.frame only
# containing the new hierarchical classifications.

# Lets increase number of mismatches an merge it with the PAC
# (How many mismatches depends on what you allowed previously in the workflow
)
colnames(pac$Anno)  # mis0-3_bio = Upto 3 mismatches are available

pac_test <- simplify_reanno(input=pac, hierarchy=hierarchy_1, mismatches=3,
                          bio_name="Biotypes_mis3", merge_pac=TRUE)

# Now we also have some mitochondrial tRNA
table(pac_test$Anno$Biotypes_mis3)
```

Analysis and visualization

There are several functions in seqpac that handle statistical analysis and visualization. We will only present a few of them here, so please refer to seqpac's reference manual for a complete list. Just like many of the preprocessing functions, these functions are named PAC_xxx to clearly show that they are applied on a PAC object.

Hint: If you are unsure if your manually constructed/manipulated PAC object are compatible, you can always run `PAC_check`.

5.1 Summarize data over groups

To make simple summaries over column categories/groups in the `PAC$Pheno` table such as means, standard deviation (SD), standard error (SE), and log2 fold changes (log2FC) etc, seqpac has a convenient function called `PAC_summary`. This function uses a target_pheno object to generate a summary over raw counts or normalized counts across sample groups. Processed data will be stored in the `PAC$summary` folder.

Important, summarized data in the `PAC$summary` folder will always have the same rownames (unique sequences) as `PAC$Counts` and `PAC$Anno`, while the column names will be derived from the `PAC$Pheno` table. Summarize using an anno_target object is not allowed, because that would disrupt sequence names (loss of sequence integrity). Summaries over sequences (such as generating means over sRNA classes) is instead handled by each plot/statistical function, and are often stored in the output from these functions. Nonetheless, user may always generate their own derived tables, using functions like `aggregate` and `reshape::melt`.

Lets have some examples on how PAC_summary works:

```
load(system.file("extdata", "drosophila_sRNA_pac_filt_anno.Rdata",
                 package = "seqpac", mustWork = TRUE))

###-----------------------------------------------------------------------
## PAC_summary in seqpac

# Make means of counts over stages and return a data.frame
tab <- PAC_summary(pac, norm = "counts", type = "means",
                   pheno_target=list("stage"), merge_pac=FALSE)

# When merge_pac=TRUE the table is added to the PAC$summary 'folder'
pac_test <- PAC_summary(pac, norm = "counts", type = "means",
                        pheno_target=list("stage"), merge_pac=TRUE)


as.PAC(pac)       # Structure of PAC before PAC_summary (S4)
#> PAC object with:
#>    9 samples
#>    9131 sequences
#>    mean total counts: 26554 (min:20353/max:38275)
#>    best sequence: 1998 mean counts
#>    worst sequence: 0 mean counts
#> normalized tables: 1
#> cpm


as.PAC(pac_test)  # Structure of PAC after PAC_summary  (S4)
#> PAC object with:
#>    9 samples
#>    9131 sequences
#>    mean total counts: 26554 (min:20353/max:38275)
#>    best sequence: 1998 mean counts
#>    worst sequence: 0 mean counts
#> normalized tables: 1
#> cpm
#> summarized tables: 1
#> countsMeans_stage


names(pac_test$summary)
#> [1] "countsMeans_stage"


head(pac_test$summary$countsMeans_stage)
#>                                      Stage1     Stage3     Stage5
#> TATTGCACTTGAGACGGCCTGAAAA         15.666667   2.666667  9.0000000
#> CATGAGGACTGTGCT                    8.333333  16.333333  6.0000000
#> TGCTTGGACTACATATGGTTGAGGGTTGTA  2203.666667 378.333333 74.0000000
#> CTGCTTGGACTACATATGGTTGAGGGTTGTA 1790.666667 456.000000 57.6666667
#> CAATGAGGGACCAGTACATGAGGACTCTGCT    6.000000  16.000000  0.6666667
#> CATGAGGACTGTGCC                    7.333333  15.333333  2.0000000

# You may want to use normalized counts
pac_test <- PAC_summary(pac_test, norm = "cpm", type = "means",
                        pheno_target=list("stage"), merge_pac=TRUE)

# Maybe only include a subset of the samples
pac_test <- PAC_summary(pac_test, norm = "cpm", type = "means",
                        pheno_target=list("batch", c("Batch1", "Batch2")),
                        merge_pac=TRUE)

# Generate standard errors
```

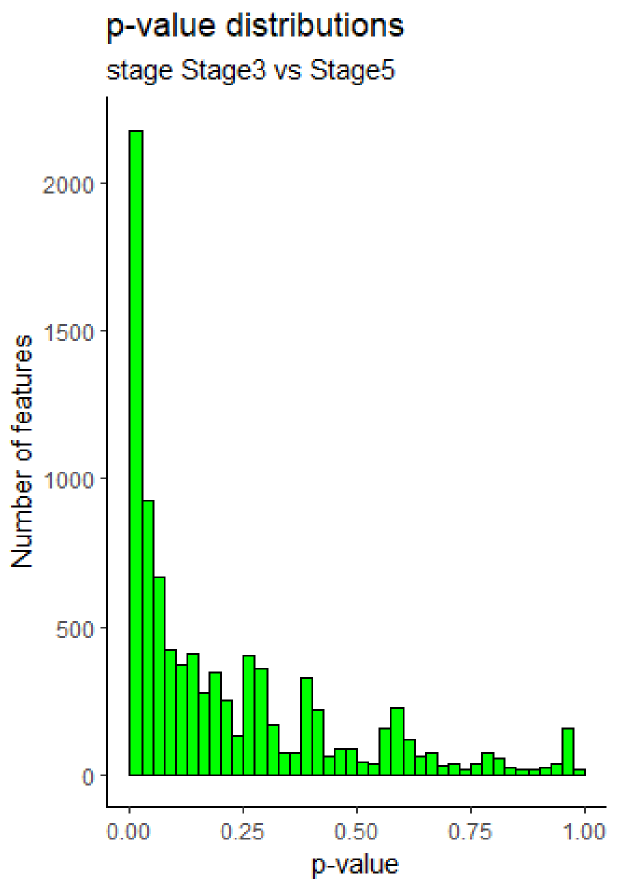
```
pac_test <- PAC_summary(pac_test, norm = "cpm", type = "se",
                        pheno_target=list("stage"), merge_pac=TRUE)

# log2FC
pac_test <- PAC_summary(pac_test, norm = "cpm", type = "log2FC",
                        pheno_target=list("stage"), merge_pac=TRUE)

# log2FC generated from a grand mean over all samples
pac_test <- PAC_summary(pac_test, norm = "cpm", type = "log2FCgrand",
                        pheno_target=list("stage"), merge_pac=TRUE)

# All summarized tables have identical row names that can be merged
names(pac_test$summary)
lapply(pac_test$summary, function(x){
  identical(rownames(x), rownames(pac_test$summary[[1]]))
  })
head(do.call("cbind", pac_test$summary))
```

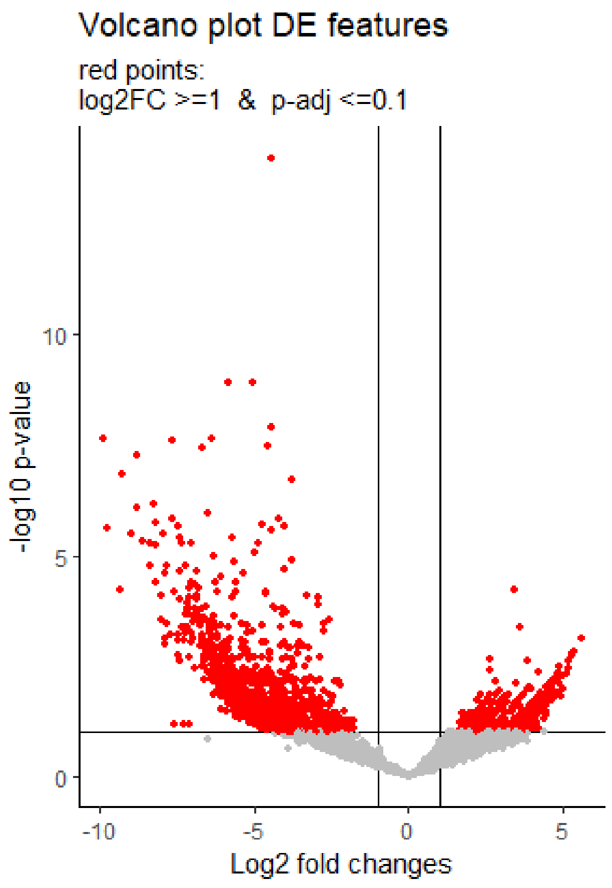
5.2 Differential Expression using PAC object

Seqpac has a useful function, `PAC_deseq`, that lets you make omic scale expressional analysis on PAC objects using the popular `DESeq2` package. This involves fitting generalized linear models with negative binomial distributions and subsequent correction for multiple testing using the PAC tables. For more information (such as how to correctly build models), please see the DESeq2 vingette: https://bioconductor.org/packages/release/bioc/html/DESeq2.html. In addition to preparing a PAC object for DESeq2, `PAC_deseq` will organize and visualize the output. Lets have an example using the test dataset:

```
load(system.file("extdata", "drosophila_sRNA_pac_filt_anno.Rdata",
                 package = "seqpac", mustWork = TRUE))

###-----------------------------------------------------------------------
## Differential expression in seqpac

# Simple model testing stages against using Wald test with local fit (default
)
table(pac$Pheno$stage)
#>
#> Stage1 Stage3 Stage5
#>      3      3      3
output_deseq <- PAC_deseq(pac, model= ~stage, threads=6)
```

```
#> ** stage Stage3 vs Stage5 **
#>
#> out of 9131 with nonzero total read count
#> adjusted p-value < 0.1
#> LFC > 0 (up)       : 820, 9%
#> LFC < 0 (down)     : 1975, 22%
#> outliers [1]       : 3, 0.033%
#> low counts [2]     : 2125, 23%
#> (mean count < 1)
```





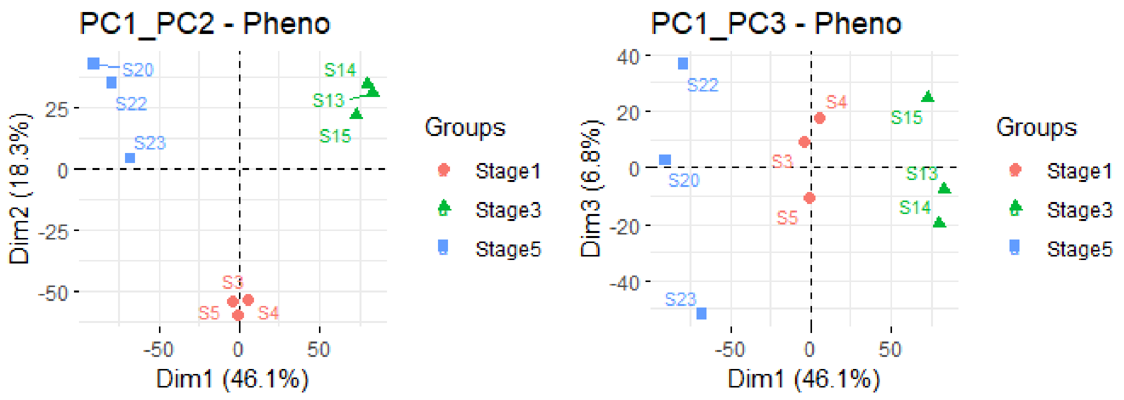
```
# More complicated, but still graphs will be generated from 'stage' since it
# is first in model
output_deseq <- PAC_deseq(pac, model= ~stage + batch, threads=6)

# Using pheno_target we can change the focus to batch
# (no batch effect)
output_deseq <- PAC_deseq(pac, model= ~stage + batch,
                          pheno_target=list("batch"))
```

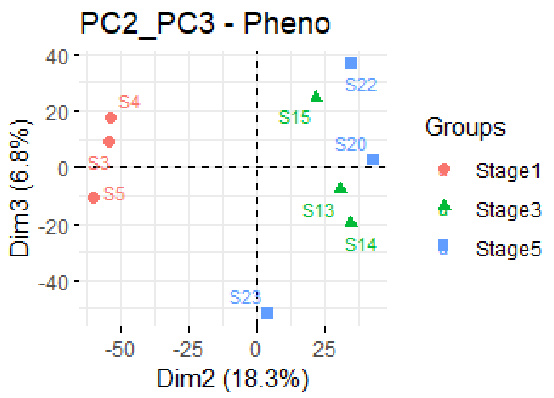
```r
# With pheno_target we can also change the direction of the comparison
# (zygotic transcription has not started)
output_deseq <- PAC_deseq(pac, model= ~stage,
                          pheno_target=list("stage", c("Stage3", "Stage1")))
# (start of zygotic transcription)
output_deseq <- PAC_deseq(pac, model= ~stage,
                          pheno_target=list("stage", c("Stage5", "Stage3")))

## In the output you find PAC-merged results, target plots and output_deseq
names(output_deseq)
head(output_deseq$result)
```

5.3 Principal component analysis (PAC_pca)

The `PAC_pca` function uses the FactoMineR package to make a principle component analysis, from which scatter plots are plotted using the ggplot2 package. Here are some examples on how to use PAC_pca:

```r
###-----------------------------------------------------------------
## PCA analysis in seqpac

# As simple as possible
output_pca <- PAC_pca(pac)
# Two 'folders' in the output
names(output_pca)

# Using pheno_target
output_pca <- PAC_pca(pac, pheno_target =list("stage"))


# Using pheno_target with sample labels
output_pca <- PAC_pca(pac, pheno_target =list("stage"),
                      label=pac$Pheno$sample)
```





```
# Plotting sequences instead
output_pca <- PAC_pca(pac, type = "anno",
                      anno_target =list("Biotypes_mis0"))
```

5.4 Size distribution and nucleotide bias

There are two function that can plot size distributions using the ggplot2 package:

-*PAC_nbias:* First nucleotide bias

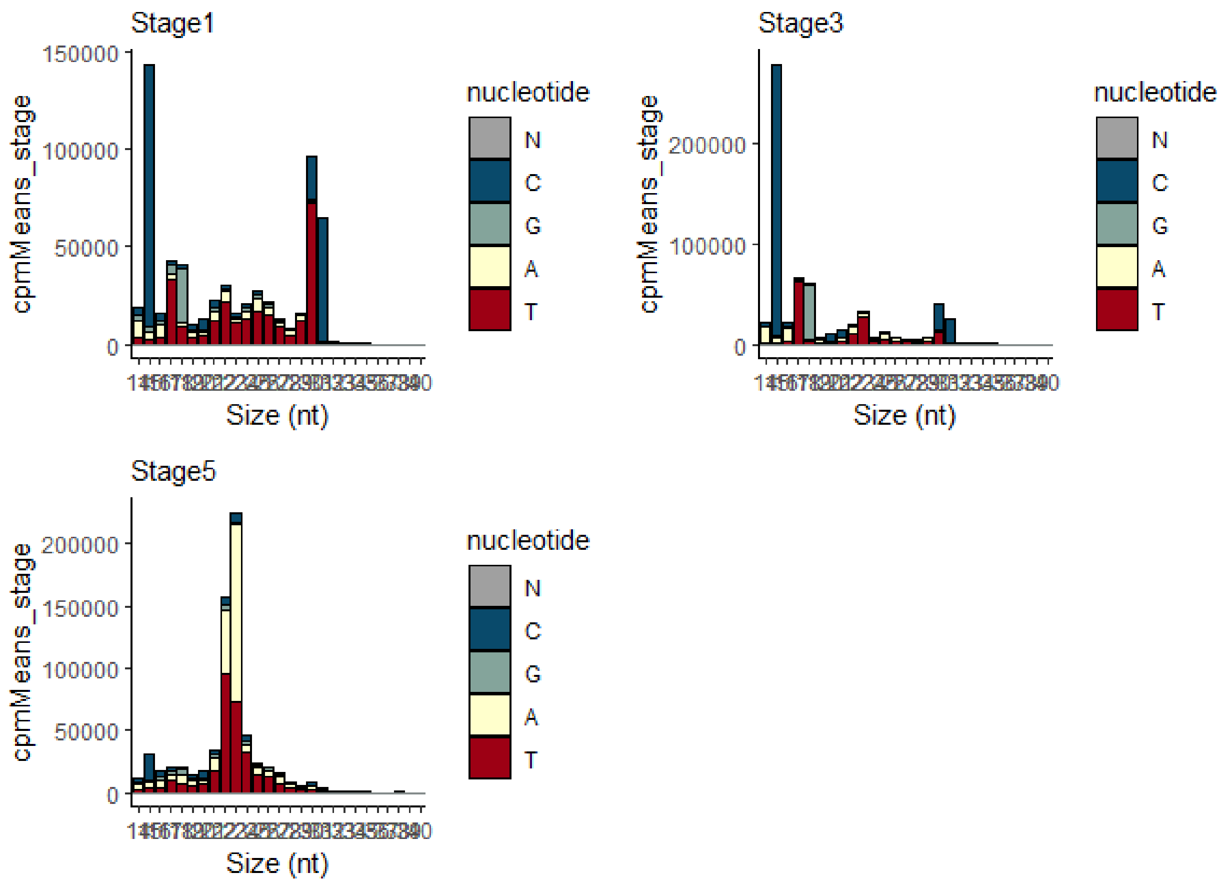
-*PAC_sizedist:* Class size distribution

PAC_nbias

Note, first nucleotide bias can be visualized without any advanced annotations added to the PAC$Anno table

```
###-------------------------------------------------------------------
## Nucleotide bias in seqpac

load(system.file("extdata", "drosophila_sRNA_pac_filt_anno.Rdata",
                 package = "seqpac", mustWork = TRUE))
```

```r
# Test without annotations
# (note, is equivalent to an unfiltered master PAC)
pac_test <- pac
pac_test$Anno <- pac_test$Anno[,1, drop = FALSE]

output_nbias <- PAC_nbias(pac_test)
cowplot::plot_grid(plotlist=output_nbias$Histograms)

# Now lets use an anno_target in an annotated PAC targetting miRNA
# (Oops, heavy T-bias on 1st nt; are they piRNA?)
table(pac$Anno$Biotypes_mis0)
output_nbias <- PAC_nbias(pac, anno_target = list("Biotypes_mis0", "miRNA") )
cowplot::plot_grid(plotlist=output_nbias$Histograms)

# Switch to 10:th nt bias
output_nbias <- PAC_nbias(pac, position=10,
                          anno_target = list("Biotypes_mis0", "miRNA"))
cowplot::plot_grid(plotlist=output_nbias$Histograms)

# Summarized over group cpm means
pac <- PAC_summary(pac, norm = "cpm", type = "means",
                   pheno_target=list("stage"), merge_pac=TRUE)
output_nbias <- PAC_nbias(pac, summary_target = list("cpmMeans_stage") )
```



```
cowplot::plot_grid(plotlist=output_nbias$Histograms)
```

PAC_sizedist

The size distribution works in a similar fashion, but instead divides the stacked columns using an anno_target:

```
###-----------------------------------------------------------------------
## Biotype size distribution

# Divide stacked bars by biotype with no mismatch allowed
output_sizedist_1 <- PAC_sizedist(pac, anno_target = list("Biotypes_mis0"))
cowplot::plot_grid(plotlist=c(output_sizedist_1$Histograms), ncol=3, nrow=3)

# Divide stacked bars by biotype with allowing up to 3 mismaches
output_sizedist_2 <- PAC_sizedist(pac, anno_target = list("Biotypes_mis3"))
cowplot::plot_grid(plotlist=c(output_sizedist_2$Histograms), ncol=3, nrow=3)



# anno_target is order sensitive, thus can take care of color order issues:
ord_bio <- as.character(unique(pac$Anno$Biotypes_mis3))
ord_bio <- ord_bio[c(1,5,2,4,3,6,7)]
output_sizedist_2 <- PAC_sizedist(pac,
                                  anno_target = list("Biotypes_mis3", ord_bio
))

# And again, we can use a summary_target instead:
output_sizedist_sum <- PAC_sizedist(pac,
                                  summary_target = list("cpmMeans_stage"),
                                  anno_target = list("Biotypes_mis3", ord_bio
))
cowplot::plot_grid(plotlist=output_sizedist_sum$Histograms)


## Note: ##################################################################
# 1. miRNA is clearly associated with the correct size (21-23) nt, which are
# dramatically increased in Stage 5 when zygotic transcription has started.
#
# 2. piRNA was deliberately left out from the fasta references. Note however,
# that there is a broad peak with no annotations between 20-30 nt in Stage 1,
# which also showed a T-bias at the first nt. Possibly piRNA?
#
###########################################################################
```

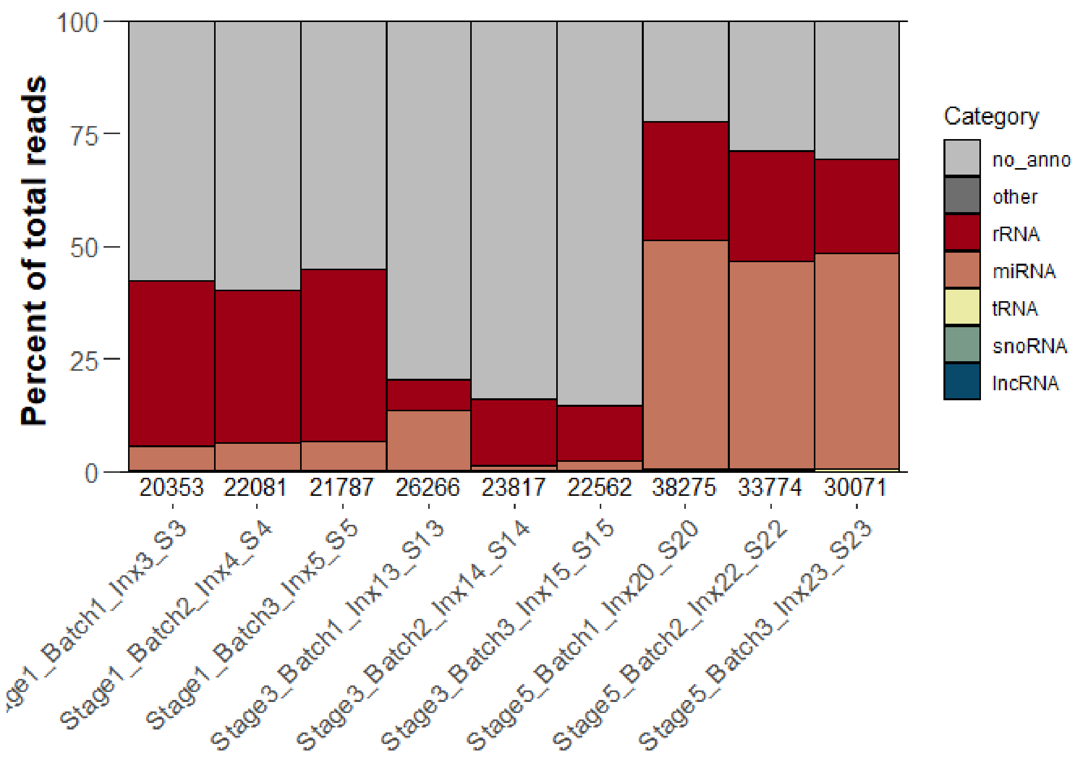
5.5 Simple stacked bars and pie charts

The `PAC_stackbar` and `PAC_pie` have the same arguments as input but with slightly different outputs. Summaries over groups in PAC$Pheno are controlled på the 'summary' argument.

```
summary= "all"    | % are generate over a mean of all samples
summary= "sample" | % are generate for each sample
summary= "pheno"  | Group % are derived from the pheno_target object
```

PAC_stackbar

```
###-----------------------------------------------------------------
## Stacked bars in seqpac
# Choose an anno_target and plot samples (summary="samples")
PAC_stackbar(pac, anno_target=list("Biotypes_mis0"))
```



```
# '

no_anno' and 'other' will always end on top not matter the order
ord_bio <- as.character(sort(unique(pac$Anno$Biotypes_mis3)))
p1 <- PAC_stackbar(pac, anno_target=list("Biotypes_mis0", ord_bio))
p2 <- PAC_stackbar(pac, anno_target=list("Biotypes_mis0", rev(ord_bio)))
cowplot::plot_grid(plotlist=list(p1, p2))
# (Hint: if you want them to appear not on top, rename them)
# Reorder samples by pheno_targets
PAC_stackbar(pac, pheno_target=list("batch"), summary="samples",
             anno_target=list("Biotypes_mis0"))

# Instead, summarized over pheno_target (summary="pheno")
PAC_stackbar(pac, anno_target=list("Biotypes_mis0"), summary="pheno",
             pheno_target=list("stage"))
```

PAC_pie

```
###---------------------------------------------------------------------
## Pie charts in seqpac

# Choose an anno_target and plot samples (summary="samples"; default)
PAC_pie(pac, anno_target=list("Biotypes_mis0"))

# Ordered pie charts of grand mean percent of all samples labeled with percen
t
ord_bio <- as.character(sort(unique(pac$Anno$Biotypes_mis3)),
                        unique(pac$Anno$Biotypes_mis0))

output_pie_1 <- PAC_pie(pac, anno_target=list("Biotypes_mis0", ord_bio),
                        summary="all", labels="percent")
output_pie_2 <- PAC_pie(pac, anno_target=list("Biotypes_mis3", ord_bio),
                        summary="all", labels="percent")

cowplot::plot_grid(plotlist=c(output_pie_1, output_pie_2),
                   labels = c("mis 0","legend", "mis 3"),
                   ncol=2, nrow=2, greedy=TRUE, scale=1.15)

# Rotate
PAC_pie(pac, anno_target=list("Biotypes_mis0"), summary="all", angle=180)
PAC_pie(pac, anno_target=list("Biotypes_mis0"), summary="all", angle=40)

# Compare biotype mapping with or without mismatches and group by PAC$Pheno
ord_bio <- as.character(sort(unique(pac$Anno$Biotypes_mis0)))
output_mis0 <- PAC_pie(pac, pheno_target=list("stage"), summary="pheno",
                       anno_target=list("Biotypes_mis0", ord_bio))
output_mis3 <- PAC_pie(pac, pheno_target=list("stage"), summary="pheno",
                       anno_target=list("Biotypes_mis3", ord_bio))

cowplot::plot_grid(plotlist=c(output_mis0, output_mis3),
                   labels = names(c(output_mis0, output_mis3)), nrow=2,
                   greedy = TRUE, scale=1.5)
```

Specilized analysis in seqpac

Seqpac functions may be used for more advanced analysis than simple classification. For example, we can dwell deeper into the details between the alignments of reads and references, by classifying reads depending on where it align. The best example of such advanced alignment analysis is tRNA loop and range classification. We may also want to make further annotations of specific classes of reads, such as classifying piRNA depending on overlaps with transposable elements and genes.

In the current version of this guide we have only included a section on how to do advanced alignment analysis for the purpose of tRNA classification, but please see the `PAC_gtf` function to find out how to use coordinate overlap annotations in seqpac.

6.1 Advanced alignment analysis (tRNA class analysis)

So far, classification of sRNA has only involved whether reads align or not to for example a tRNA reference sequence. Type classification of tRNA, such as 5', 3', i' and half, tRF (as nicely described in the MINTmap tool ://pubmed.ncbi.nlm.nih.gov/28220888/), resembles mapping against a reference genome. Obtaining the exact coordinates of where a read align to the reference allows us to plot coverage plots showing the coverage of sRNA across the reference.

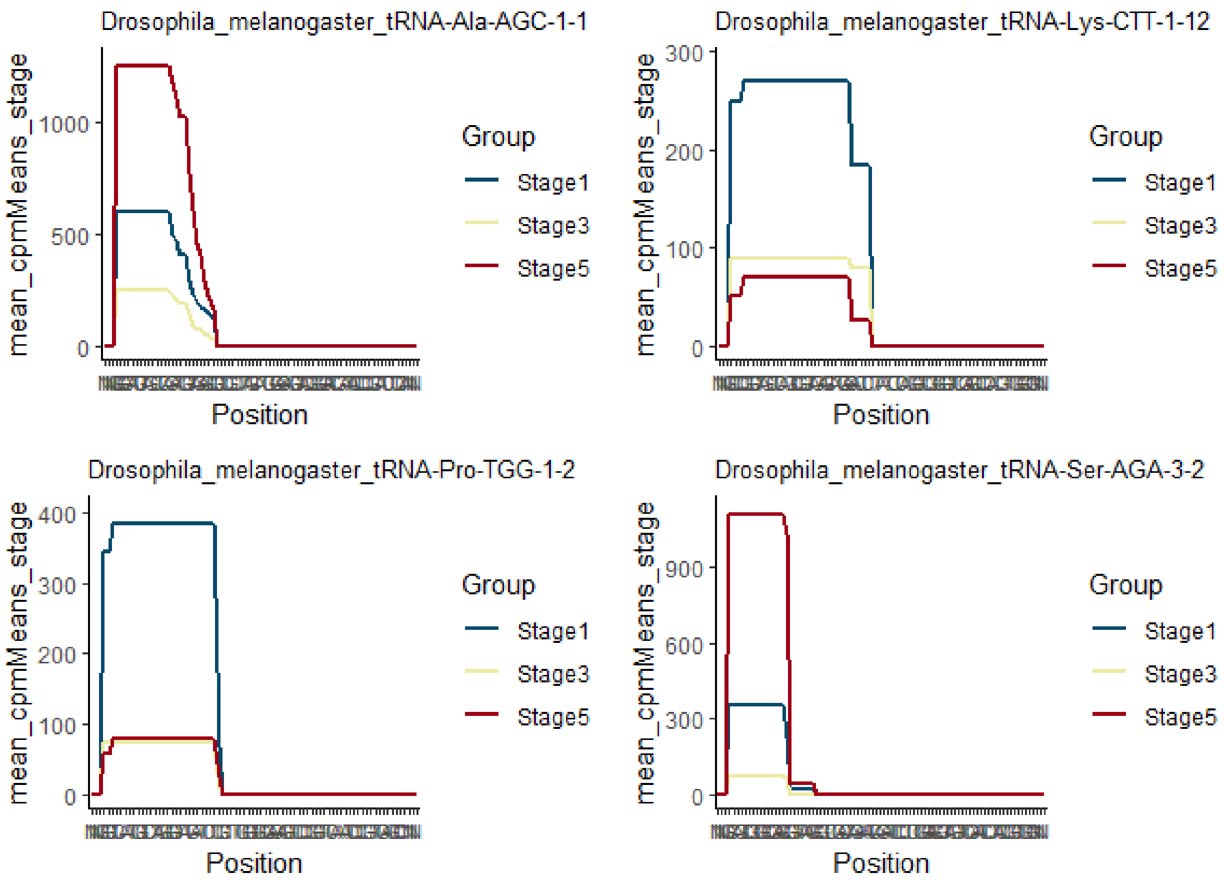
Conveniently, seqpac contains a 'backdoor' to the reanno workflow, which allows the user to quickly obtain mapping coordinates of sequences in a PAC object that align to a reference fasta file. This is done by the `PAC_mapper` function. This function uses temporary output files from bowtie to generate a map object, which simply lists the reads that mapped to each sequence in the reference.

Importantly, if the Bowtie index is missing for a fasta reference, `PAC_mapper` will automatically generate such an index. Thus, even though larger references may be possible to use, they are discouraged with this function.

After a map object has been generated, the `PAC_covplot` function can make coverage plots of reads mapping to each reference. Lets have a look on how it works in seqpac for advanced tRNA analysis using the test dataset:

```
library(seqpac)
load(system.file("extdata", "drosophila_sRNA_pac_filt_anno.Rdata",
                 package = "seqpac", mustWork = TRUE))
```

```r
###-----------------------------------------------------------------------
## tRNA analysis in seqpac

# First create an annotation blank PAC with group means
pac$Anno <- pac$Anno[,1, drop=FALSE]
pac_trna <- PAC_summary(pac, norm = "cpm", type = "means",
                        pheno_target=list("stage"), merge_pac = TRUE)

# Then reannotate only tRNA using the PAC_mapper function
ref <- "/some/path/to/trna/fasta/recerence.fa"
# or use the provided fasta
ref <- system.file("extdata/trna_no_index", "tRNA_copy.fa",
                          package = "seqpac", mustWork = TRUE)

map_object <- PAC_mapper(pac_trna, ref=ref, N_up = "NNN", N_down = "NNN",
                          mismatches=0, threads=8, report_string=TRUE)

# Hint: By adding N_up ad N_down you can make sure that modified fragments (l
ike 3' -CAA in mature tRNA are included).

## Plot tRNA using xseq=TRUE gives you the reference sequence as X-axis:
# (OBS! Long reference will not show well.)
cov_tRNA <- PAC_covplot(pac_trna, map_object,
                        summary_target = list("cpmMeans_stage"),
                        xseq=TRUE, style="line",
                        color=c("red", "black", "blue"))
cov_tRNA[[1]]

# Targeting a single tRNA using a summary data.frame
PAC_covplot(pac_trna, map=map_object, summary_target= list("cpmMeans_stage"),
            map_target="tRNA-Lys-CTT-1-9")

# Find tRNAs with many fragments
# 1st extract number of rows from each alignment
n_tRFs <- unlist(lapply(map_object, function(x){nrow(x[[2]])}))
# The test data is highly down an filterd sampled, but still some with tRNA h
ave
# a few alignment
table(n_tRFs)
names(map_object)[n_tRFs>2]
# Lets select a few of them and plot them
selct <- names(map_object)[n_tRFs>2][c(1, 16, 27, 37)]
cov_plt <- PAC_covplot(pac_trna, map=map_object,
                      summary_target= list("cpmMeans_stage"),
                      map_target=selct)


cowplot::plot_grid(plotlist=cov_plt, nrow=2, ncol=2)
```



Now, tRNAs can be further classified into isodecoder and isoacceptors. This information is usually provided within the reference names in a tRNA fasta reference file. Thus, it can be extracted from these names. Please see section, `4.1 Fasta references` for some examples on how to work with fasta references in R.

Classification of cleavage products, like 5'and 3' halves etc, are more complicated. Such classification involves knowledge of where in the tRNA loops (A-loop, anticodon loop and T-loop) cleavage may occur. There are external software that predict loops from models of secondary structures. One of the most popular tools for predicting tRNA secondary structures are tRNAscan-SE (http://lowelab.ucsc.edu/tRNAscan-SE/). The output from this software is an ss file, which stores loop information in the following format:

```
Full length tRNA: tRNA-Pro-CGG-2-1_chrX:18565764-18565835_(+)
GGCTCGTTGGTCTAGAGGTATGATTCTCGCTTCGGGTGCGAGAGGTCCCGGGTTCAATTCCCGGACGAGCCC
>>>>>>>..>>>.........<<<.>>>>>.......<<<<<.....>>>>>.......<<<<<<<<<<<<.
                Loop 1              Loop 2                   Loop 3
```

You may notice that each loop is contain between ">>>...<<<" and that each character (".><") corresponds to one nucleotide in the original full length tRNA. Ss-files for most common genomes are readily available for download at http://gtrnadb.ucsc.edu/.

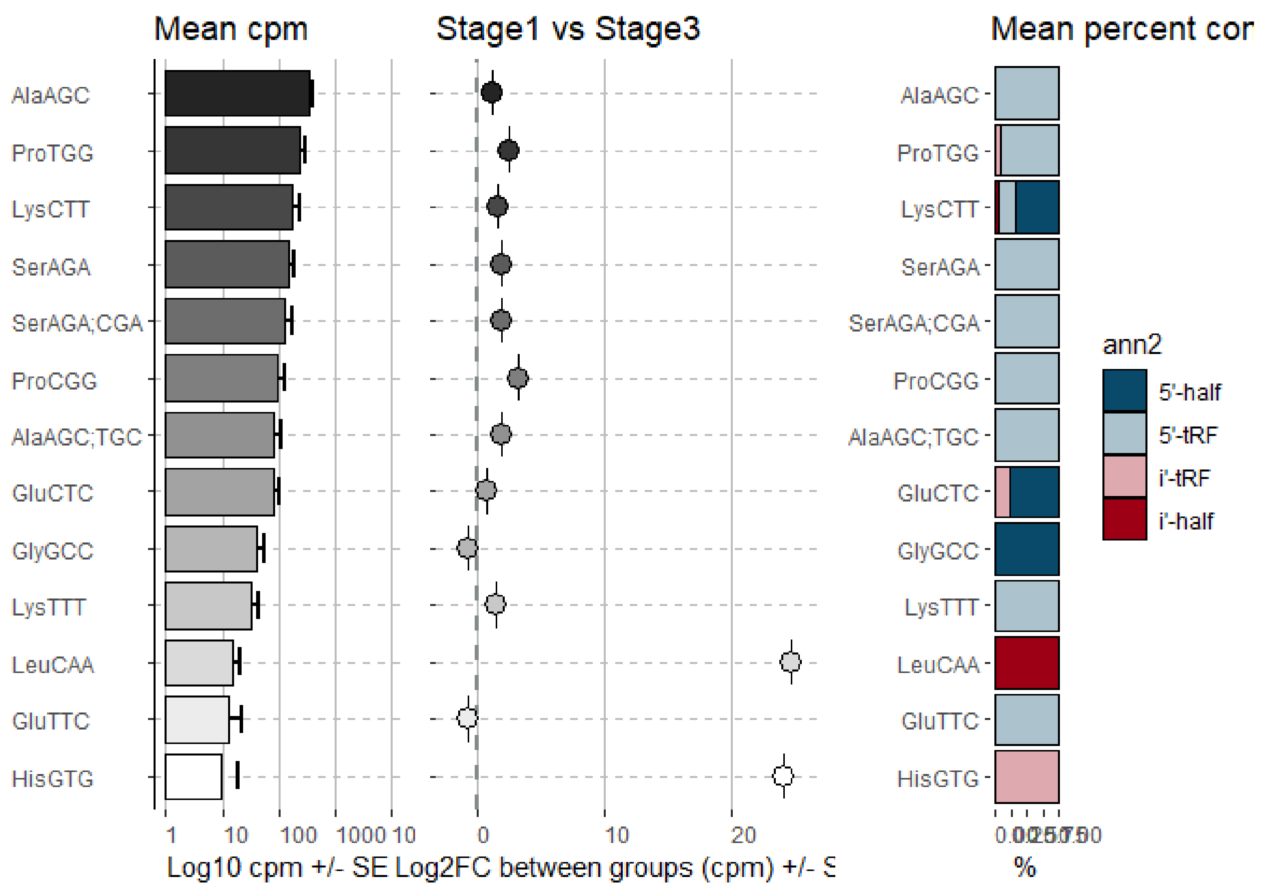
In seqpac, the `map_rangetype` function is used to further annotate the resulting map object obtained using the `PAC_mapper` function. First, if tRNA reference names were provided in the correct format (eg. “tRNA-Pro-CGG-2-1_chrX:18565764-18565835_(+): format: name -> genome coordinates -> strand), it will automatically extract the isodecoder (e.g. ”Pro”) and isoacceptor (e.g. “CGG”) and make them seperate factors. It may also classify each mapped PAC sequence in relation to the start and end terminals of the full length tRNA. Lastly, given an ss file, it can annotate PAC sequences in relation to tRNA loops. The `tRNA_class` function can then combine the information in the mapping object and to classify each read sequence in the PAC object. Lets have a look on how this is done:

```r
###-----------------------------------------------------------------------
## Generate range types using a ss-file

# Download ss object from GtRNAdb
# ("http://gtrnadb.ucsc.edu/")
ss_file <- "/some/path/to/GtRNAdb/trna.ss"
# or for testing use the provided ss-file in secpac
ss_file <- system.file("extdata/trna_no_index", "tRNA.ss",
                          package = "seqpac", mustWork = TRUE)

# Classify fragments according to loop cleavage (small loops are omitted)
map_object_ss <- map_rangetype(map_object, type="ss",
                               ss=ss_file, min_loop_width=4)

# Remove reference tRNAs with no hits
map_object_ss <-  map_object_ss[
  !unlist(lapply(map_object_ss, function(x){x[[2]][1,1] == "no_hits"}))]

# Now we have quite comprehensive tRNA loop annotations
names(map_object_ss)
map_object_ss[[1]][[2]]

# Don't forget to check ?map_rangetype to obtain more options


###-----------------------------------------------------------------------
## Function classifying 5'-tRF, 5'halves, i-tRF, 3'-tRF, 3'halves

# Does all tRNAs have 3 loops?
table(unlist(lapply(map_object_ss, function(x){unique(x$Alignments$n_ssloop)}
)))

# Set tolerance for classification as a terminal tRF
tolerance <- 5  # 2 nucleotides from start or end of full-length tRNA)
```

```
### Important:
# We set N_up and N_down to "NNN" in the PAC_mapper step. To make sure
# that we have a tolerance that include the original tRNA sequence
# we set terminal= 2+3 (5).

## tRNA classifying function
# Apply the tRNA_class function and make a tRNA type column
pac_trna <- tRNA_class(pac_trna, map=map_object_ss, terminal=tolerance)

pac_trna$Anno$type <- paste0(pac_trna$Anno$decoder, pac_trna$Anno$acceptor)
pac_trna$Anno[1:5, 1:4]
                                   Size   class decoder acceptor
#> CGGCTAGCTCAGTCGGTAGAGCATGAGACTC   31 i'-half     Lys      CTT
#> CGTGATCGTCTAGTGGTTAGGACCCCA       27  i'-tRF     His      GTG
#> CTCAATGGTCTAGGGGTATGATTCT         25  i'-tRF     Pro      TGG
#> GCAGTCGTGGCCGAG                   15  5'-tRF     Ser  AGA;CGA
#> GCAGTCGTGGCCGAGC                  16  5'-tRF     Ser      AGA
```

When classification by isoacceptor/decoder and cleavage type has been done, the `PAC_trna` function can be applied to generate pheno group differences in relation to tRNA fragmentation:

```
###-------------------------------------------------------------------
## Plot tRNA types

# Now use PAC_trna to generate some graphs based on grand means
trna_result <- PAC_trna(pac_trna, norm="cpm", filter = NULL,
  join = TRUE, top = 15, log2fc = TRUE,
  pheno_target = list("stage", c("Stage1", "Stage3")),
  anno_target_1 = list("type"),
  anno_target_2 = list("class"))
#>
#> -- Pheno target was specified, will retain: 6 of 9 samples.
#> -- Anno target was specified, will retain: 29 of 29 seqs.
#> -- Anno target was specified, will retain: 29 of 29 seqs.

names(trna_result)
#> [1] "plots" "data"
names(trna_result$plots)
#> [1] "Expression_Anno_1" "Log2FC_Anno_1"     "Percent_bars"
names(trna_result$plots$Expression_Anno_1)
#> [1] "Grand_means"

cowplot::plot_grid(trna_result$plots$Expression_Anno_1$Grand_means,
                   trna_result$plots$Log2FC_Anno_1,
                   trna_result$plots$Percent_bars$Grand_means,
                   nrow=1, ncol=3)
```



```r
# By setting join = FALSE you will get group means
trna_result <- PAC_trna(pac_trna, norm="cpm", filter = NULL,
  join = FALSE, top = 15, log2fc = TRUE,
  pheno_target = list("stage", c("Stage1", "Stage3")),
  anno_target_1 = list("type"),
  anno_target_2 = list("class"))

names(trna_result$plots$Expression_Anno_1)

cowplot::plot_grid(trna_result$plots$Expression_Anno_1$Stage1,
                   trna_result$plots$Expression_Anno_1$Stage3,
                   trna_result$plots$Log2FC_Anno_1,
                   trna_result$plots$Percent_bars$Stage1,
                   trna_result$plots$Percent_bars$Stage3,
                   nrow=1, ncol=5)
```

That completes this vignette. Good luck!

```
sessionInfo()
#> R version 4.1.2 (2021-11-01)
#> Platform: x86_64-w64-mingw32/x64 (64-bit)
#> Running under: Windows 10 x64 (build 19044)
#> 
#> Matrix products: default
#> 
#> locale:
#> [1] LC_COLLATE=English_United States.1252 
#> [2] LC_CTYPE=English_United States.1252 
#> [3] LC_MONETARY=English_United States.1252
#> [4] LC_NUMERIC=C 
#> [5] LC_TIME=English_United States.1252 
#> 
#> attached base packages:
#> [1] stats     graphics  grDevices utils     datasets  methods   base     
#> 
#> other attached packages:
#> [1] seqpac_0.99.1
#> 
#> loaded via a namespace (and not attached):
#>   [1] colorspace_2.0-2           ggsignif_0.6.3            
#>   [3] hwriter_1.3.2              ellipsis_0.3.2            
#>   [5] Rbowtie_1.34.0             XVector_0.34.0            
#>   [7] GenomicRanges_1.46.0       rstudioapi_0.13           
#>   [9] ggpubr_0.4.0               farver_2.1.0              
#>  [11] ggrepel_0.9.1              DT_0.20                   
#>  [13] bit64_4.0.5                AnnotationDbi_1.56.2      
#>  [15] fansi_0.5.0                codetools_0.2-18          
#>  [17] splines_4.1.2              leaps_3.1                 
#>  [19] doParallel_1.0.16          cachem_1.0.6              
#>  [21] geneplotter_1.72.0         knitr_1.36                
#>  [23] Rsamtools_2.10.0           gginnards_0.1.0-1         
#>  [25] broom_0.7.10               annotate_1.72.0           
#>  [27] cluster_2.1.2              png_0.1-7                 
#>  [29] compiler_4.1.2             httr_1.4.2                
#>  [31] backports_1.3.0            Matrix_1.3-4              
#>  [33] fastmap_1.1.0              cli_3.1.0                 
#>  [35] htmltools_0.5.2            tools_4.1.2               
#>  [37] gtable_0.3.0               glue_1.4.2                
#>  [39] GenomeInfoDbData_1.2.7     reshape2_1.4.4            
#>  [41] dplyr_1.0.7                FactoMineR_2.4            
#>  [43] ShortRead_1.52.0           Rcpp_1.0.7                
#>  [45] carData_3.0-4              Biobase_2.54.0            
#>  [47] vctrs_0.3.8                Biostrings_2.62.0         
#>  [49] iterators_1.0.13           xfun_0.28                 
#>  [51] stringr_1.4.0              lifecycle_1.0.1           
#>  [53] rstatix_0.7.0              XML_3.99-0.8              
#>  [55] factoextra_1.0.7           zlibbioc_1.40.0           
#>  [57] MASS_7.3-54                scales_1.1.1              
#>  [59] MatrixGenerics_1.6.0        parallel_4.1.2
#>  [61] SummarizedExperiment_1.24.0 RColorBrewer_1.1-2
#>  [63] yaml_2.2.1                  memoise_2.0.0
#>  [65] ggplot2_3.3.5               latticeExtra_0.6-29
#>  [67] stringi_1.7.5               RSQLite_2.2.8
#>  [69] highr_0.9                   genefilter_1.76.0
#>  [71] S4Vectors_0.32.2            foreach_1.5.1
#>  [73] BiocGenerics_0.40.0         BiocParallel_1.28.0
#>  [75] GenomeInfoDb_1.30.0         rlang_0.4.12
#>  [77] pkgconfig_2.0.3             matrixStats_0.61.0
#>  [79] bitops_1.0-7                evaluate_0.14
#>  [81] lattice_0.20-45             purrr_0.3.4
#>  [83] GenomicAlignments_1.30.0    htmlwidgets_1.5.4
#>  [85] labeling_0.4.2              cowplot_1.1.1
#>  [87] bit_4.0.4                   tidyselect_1.1.1
#>  [89] plyr_1.8.6                  magrittr_2.0.1
#>  [91] DESeq2_1.34.0               R6_2.5.1
#>  [93] IRanges_2.28.0              generics_0.1.1
#>  [95] DelayedArray_0.20.0         DBI_1.1.1
#>  [97] pillar_1.6.4                survival_3.2-13
#>  [99] KEGGREST_1.34.0             scatterplot3d_0.3-41
#> [101] abind_1.4-5                 RCurl_1.98-1.5
#> [103] tibble_3.1.6                crayon_1.4.2
#> [105] car_3.0-12                  utf8_1.2.2
#> [107] rmarkdown_2.11              jpeg_0.1-9
#> [109] locfit_1.5-9.4              grid_4.1.2
#> [111] data.table_1.14.2           blob_1.2.2
#> [113] digest_0.6.28               flashClust_1.01-2
#> [115] xtable_1.8-4                tidyr_1.1.4
#> [117] gridGraphics_0.5-1          stats4_4.1.2
#> [119] munsell_0.5.0
```