scAPAtrap tutorial

Xiaohui Wu, Tao Liu

11, 2023

Contents

1 Overview

The scAPAtrap toolkit is applicable to majority of poly(A)-captured scRNA-seq protocols. Currently scAPAtrap supports the following library construction technologies: 10x, Drop-seq, InDrops, CEL-seq, CEL-seq2. Due to the different data format of 10x from other protocols, we provide two procedures to show the use of scAPAtrap on 10x and other scRNA-seq protocols (hereinafter called non-10x), respectively.

This tutorial describes two ways of running scAPAtrap: one-step and step-by-step.

The only required input is a BAM file (`inputBam`) which can be retrieved from CellRange, STARsolo, etc. Four tools should be installed (umi_tools, samtools, featureCounts, and STAR).

1.1 Installation

scAPAtrap can be installed from github.

```
devtools::install_github("BMILAB/scAPAtrap")
```

Please also install the following tools before running scAPAtrap: * samtools * subread * umi_tools * star

You can also install the above software using conda.

```
conda install samtools -c bioconda
conda install subread -c bioconda
conda install umi_tools -c bioconda
conda install star -c bioconda
```

1.2 Demo data `fly_demo`

- A BAM file from drosophila `fly_demo.bam`.
- A barcode file `barcode.txt` that stores barcodes in one column (optional).

Please visit this link to download demo data. The output files and log file during running one-step and step-by-step scAPAtrap can also be found here or on GitHub site.

1.3 Raw data processing

The raw scRNA-seq data of Drosophila rhythmic neurons can be obtained from NCBI (GSM4768020), which was sequenced by CEL_seq2. Since the raw data is sequenced from CEL-seq2, STARsolo was used for read mapping to generate the BAM file `fly_demo.bam`. For 10X data, CellRanger can be used for read mapping.

When STARsolo or CellRanger is used for read mapping, we should set `TenX=TRUE` in the scAPAtrap pipeline even if the data is obtained from non-10X library.

The STARsolo command is like the following:

```
STAR --runMode genomeGenerate
--runThreadN 16
   --genomeFastaFiles
   --sjdbGTFfile
   --genomeDir

STAR --runThreadN 5
--genomeDir
--readFilesIn
--soloType CB_UMI_Simple
--soloCBwhitelist
--soloCBlen=6
--soloCBstart=1
--soloUMIstart=7
--soloUMIlen=6
--soloBarcodeReadLength 0
--outSAMtype BAM SortedByCoordinate
--outSAMattributes CR UR CY UY CB UB
```

Because there are some barcodes that not present in the BAM file after STARsolo mapping (like CB:Z), we need to filter out these lines before running scAPAtrap.

```
samtools view -@ 48 -D CB:barcodes.txt -o fly_demo.bam ./Aligned.sortedByCoord.out.bam
```

2 scAPAtrap parameters

There are some important parameters should be set carefully to call and determine peaks (pAs), which are passed to scAPAtrap's parameter list (`TRAP.PARAMS`, see `scAPAtrap::setTrapParams`).

- `TenX`: Whether the inputBam is 10X-format (e.g., BAM file from CellRanger or STARsolo), default is TRUE. **For non-10X protocols like CEL-seq, when STARsolo is used, the BAM file will also contain cell barcode tag (CB) and UMI barcode tag (UB), then can set TenX=TRUE.**
- `barcode`: a string vector to define barcodes, default is NULL (not provided).
- `chrs`: a string vector of chromosome names (e.g., chr1, chr2, chr3…), default is NULL. This parameter is used in `findPeaksByStrand`. If chrs=NULL and run one-step `scAPAtrap`, then will try to get all chrs from inputBAM. In such case, if chrs=NULL and there are more than 100 chrs from inputBAM, users should provide trap.params$chrs explicitly to run scAPAtrap just in case there are too many usefulness chrs (e.g., scaffolds) to run.
- `maxwidth`: max width (nt) to define a valid peak, default is 1000. Peaks wider than `maxwidth` will be automatically split until meet requirement. This parameter is used in `findPeaksByStrand`.
- `readlength`: R2 read length, default is 49. `readlength` determines the min peak width; peaks wider than `readlength` will be retained. This parameter is used in `findPeaksByStrand`.
- `cov.cutoff`: min read coverage for peak calling, default is 10. Peak regions with coverage greater than `cov.cutoff` will be considered as candidate peaks. This parameter is used in `findPeaksByStrand`.
- `min.cells`: number of cells that a peak is expressed (i.e., with read count over `min.count`), default is 10. `min.cells` is used to filter peaks present in at least `min.cells` cells. This parameter is used in `reducePeaks` and `generatescExpMa`.

- `min.count`: number of read counts a peak has, default is 10. `min.count` is used to filter peaks with at least `min.count` reads. This parameter is used in `reducePeaks` and `generatescExpMa`.
- `tails.search`: the strategy to search polyA-tails, can be set `peaks`, `genome`, `no` (default).
  - `no`: Not search tails, which means the final peaks are not linked with any A-tails.
  - `genome`: Search tails genome-wide, which may be suitable for small genome. Otherwise it will take very long time or MEM.
  - `peaks`: Search tails within the peak regions by `d` nt. This way may be suitable for big genome but with relatively small number of peaks.
- `d`: min distance from a peak to any tail to call a peak as real PA, default is 100 (this means a peak with at least one tail within 100 bp range is considered as real peak). Only applicable when `tails.search` is `peaks` or `genome`. This parameter is used in `findTailsByPeaks` and `generatescExpMa`.

3 One-step running scAPAtrap

The wrapper function `scAPAtrap` can be used to run scAPAtrap for a BAM file in one step.

3.1 Preparation

- Four tools (samtools, umi_tools, featureCounts, STAR) have been installed. We recommend to use `conc

- An input BAM file.

- A barcode list (if available).

- Chromosome names (if available).

- Correct parameters settings (e.g., 10X or not; read length; search tail or not; see `TRAP.PARAMS()`

3.2 Run `scAPAtrap`

```
library(scAPAtrap)

## input BAM
dir0='/demoFly/'
inputBam=paste0(dir0, "fly_demo.bam")

## log file to LOG all information (time, command, output file names..)
logf=gsub('.bam', '.APA.notails.onestep.log', inputBam, fixed=TRUE)

## output dir (will be under inputBam's dir if only dirname is provided; otherwise use the full path)
outputDir="APA.notails.onestep"

## full path of tools
tools=list(samtools='/home/dell/miniconda3/envs/scAPA/bin/samtools',
           umitools='/home/dell/miniconda3/envs/scAPA/bin/umi_tools',
           featureCounts="/home/dell/miniconda3/envs/scAPA/bin/featureCounts",
           star='/home/dell/miniconda3/envs/scAPA/bin/STAR')
```

```
## set parameters, first load default parameters and then modify
trap.params=setTrapParams()
## set tail search way
trap.params$tails.search='no'
## set chromosome names
trap.params$chrs=c('2L','2R','3L','3R','4','X','Y')

## we can also get chromosome names from the BAM file
## however, we should also check the number of chrs to avoid too many sccafolds or fragments.
chrs=scAPAtrap:::.getBAMchrs(inputBam)
length(chrs)

## barcode (if have)
trap.params$barcode <- read.delim2(paste0(dir0, 'barcode.txt'), header = F)$V1

## Run scAPAtrap
scAPAtrap(tools=tools,
          trap.params=trap.params,
          inputBam=inputBam,
          outputDir=outputDir,
          logf=logf)
```

3.3 Output files

Output files are recorded in LOG file `logf`. Files marked with star (*) are temporary files can be deleted.

However, if errors occur during running `scAPAtrap`, these temporary files are important for starting from that step to save time.

If `tails.search='no'`, then the final output peak (pA) file are: **`scAPAtrapData.rda, peaks.saf, counts.tsv.gz`** under the `outputDir`.

`scAPAtrapData.rda` stores a object named `scAPAtrapData`, which is a list containing two matrices – peaks.meta and peaks.count. This file can be further loaded into other tools like movAPA by scAPAtrap's function `convertAPAtrapData` for downstream analysis. Please refer to movAPA_on_scAPAtrap_results for details about how to read the scAPAtrapData and analyze it with movAPA.

Some output files in this tutorial can be found here or on GitHub site.

File description	File name
*findUniqueMap: samtools sort	dir0/fly_demo.UniqSorted.bam
*findUniqueMap: samtools index	dir0/fly_demo.UniqSorted.bam.bai
*dedupByPos: umitools dedup	dir0/fly_demo.UniqSorted.dedup.bam
*separateBamBystrand: samtools forward BAM	dir0/fly_demo.UniqSorted.dedup.forward.bam
*separateBamBystrand: samtools reverse BAM	dir0/fly_demo.UniqSorted.dedup.reverse.bam
*separateBamBystrand: samtools forward index	dir0/fly_demo.UniqSorted.dedup.forward.bam.bai
*separateBamBystrand: samtools reverse index	dir0/fly_demo.UniqSorted.dedup.reverse.bam.bai
*findPeaksByStrand: peak (+)	dir0/fly_demo.UniqSorted.dedup.forward.bam.peaks
*findPeaksByStrand: peak (-)	dir0/fly_demo.UniqSorted.dedup.reverse.bam.peaks
generateSAF peaks.saf	dir0/**APA.tails.no/peaks.saf**
*generateFinalBam: featureCounts log	dir0/peak_assigned
*generateFinalBam: featureCounts BAM	dir0/fly_demo.bam.featureCounts.bam
*generateFinalBam: samtools sort	dir0/final.bam

File description	File name
countPeaks: umitools counts	dir0/**APA.tails.no/counts.tsv.gz**
generatescExpMa: scAPAtrapData	dir0/**APA.tails.no/scAPAtrapData.rda**

If `tails.search='genome'`, one more file `..genome.tails` would be output. If `tails.search='peaks'`, will first remove lowly expressed peaks to retain less peaks for searching (`..saf.reduced` and `...counts.tsv.gz.reduced`) and then get tails near these peaks (`...peaks.tails`).

File description	File name
reducePeaks: reduced peaksfile	dir0/APA.tails.peak/peaks.saf.reduced
reducePeaks: reduced countsfile	dir0/APA.tails.peak/counts.tsv.gz.reduced
findTails: tails-coords	dir0/fly_demo.bam.**peaks.tails**
findTails: tails-coords	dir0/fly_demo.bam.**genome.tails**

4 Step by step running scAPAtrap

Step by step running scAPAtrap is equivalent to running each step of the wrapper function `scAPAtrap`. All LOG information will be appended to `logf` file during each step.

4.1 Step 1. `initScAPAtrap` to set-up

`initScAPAtrap` initiates scAPAtrap pipeline's several parameters including `logf` (LOG file), `trap.params` (parameters list), `tools` (Shell tools like umitools), `TRAPFILES` (a global variable that stores output file names and description, which is automatically set).

```
################### 1. initiate ########################
library(scAPAtrap)

## input BAM and output dir
dir0='/mnt/64cf3476-350c-46ad-bc48-574fa64a0334/test/xwu/demoFly/'
inputBam=paste0(dir0, "fly_demo.bam")
outputDir='APA.notails.stepbystep'

## logf
logf=gsub('.bam', '.APA.notails.stepbystep.log', inputBam, fixed=TRUE)
unlink(logf)

## tools
tools=list(samtools='/home/dell/miniconda3/envs/scAPA/bin/samtools',
           umitools='/home/dell/miniconda3/envs/scAPA/bin/umi_tools',
           featureCounts="/home/dell/miniconda3/envs/scAPA/bin/featureCounts",
           star='/home/dell/miniconda3/envs/scAPA/bin/STAR')

## trap.params: first get default parameters and set specific ones
trap.params=setTrapParams()

## chrs
trap.params$chrs=c('2L','2R','3L','3R','4','X','Y')
```

```
## barcode (if have)
trap.params$barcode <- read.delim2(paste0(dir0, 'barcode.txt'), header = F)$V1

## initiation, will print these settings in logf and on screen
initScAPAtrap(tools=tools,
              trap.params=trap.params,
              inputBam=inputBam,
              outputDir=outputDir,
              logf=logf)
```

4.2 Step 2. `findUniqueMap` to get unique mapped reads with samtools

The BAM file obtained from CellRanger normally contains reads mapped at multiple positions, which would affect the detection especially the quantification of poly(A) sites. Therefore, first we obtain uniquely mapped reads from the input BAM file.

The `index` parameter and `sort` parameter in `findUniqueMap` indicate whether to sort and build the index of the BAM file, which is equivalent to the `samtools sort` and `samtools index` commands. It is recommended to sort and build an index here, because in the next steps, the BAM file and its corresponding index file are required.

Not all BAM files require `findUniqueMap`. If the BAM file contains multiple comparison results, such as the BAM file obtained by CellRanger, you need to use the `findUniqueMap` function to perform filtering.

After each step, it is better to call `scAPAtrap:::.checkAndPrintFiles` to check the existence of the output file and print the files to assure each step runs successfully.

```
################## 2. findUniqueMap ######################
# findUniqueMap: uniq >> sort >> index with samtools
# Skip this step if the inputBam contains only unique mappings.
inputBam.u=inputBam
if (trap.params$findUniqueMap)
  inputBam.u <- findUniqueMap(tools$samtools,
                              inputBam,
                              thread=trap.params$thread,
                              sort = trap.params$findUniqueMap,
                              index = trap.params$findUniqueMap.index,
                              logf=logf)

scAPAtrap:::.checkAndPrintFiles(varnames=c("inputBam.u"),
                                filenames=inputBam.u, logf=logf)
```

4.3 Step 3. `dedupByPos` to remove duplicates with umitools

This step removes duplicates and saves only one read per UMI for each position.

The input of `dedupBypos` is a BAM file with the corresponding index. The output is the file path where `dedupBypos` generates the BAM file. The `TenX` parameter indicates whether the current input BAM file is obtained by the 10x library. However, even for non-10X protocols like CEL-seq, when **STARsolo** is used, the BAM file will also contain cell barcode tag (CB) and UMI barcode tag (UB), then can also set `TenX=TRUE`.

```
#################### 3. dedupByPos ####################
## dedupByPos: remove duplicates with umitools
```

```
inputBam.u.dedup <- dedupByPos(tools$umitools,
                               input=inputBam.u,
                               TenX=trap.params$TenX,
                               logf=logf)

scAPAtrap:::.checkAndPrintFiles(varnames=c("inputBam.u.dedup"),
                                filenames=inputBam.u.dedup, logf=logf)
```

4.4 Step 4. `separateBamBystrand` to split BAM file with samtools

This step separates BAM file by strand to output two files: `...reverse.bam` and `...forward.bam`, and builds index for these two files (generating `...bam.bai`).

```
#################### 4. separateBamBystrand ####################
## separateBamBystrand: split the BAM file by strand to forward.bam and reverse.bam
inputBam.u.dedup.seps <- separateBamBystrand(tools$samtools,
                                             input=inputBam.u.dedup,
                                             thread=trap.params$thread,
                                             logf=logf)

scAPAtrap:::.checkAndPrintFiles(varnames=c("inputBam.u.dedup.seps[1]", "inputBam.u.dedup.seps[2]"),
                                filenames=inputBam.u.dedup.seps)
```

4.5 Step 5. `findPeaksByStrand` to call peaks from a BAM file

This step is the most important step in scAPAtrap's pipeline, which finds peaks from the BAM file of one strand. `findPeaksByStrand` first loads BAM coverages and then gets peak regions based on `cutoff`, `readlength`, and `maxwidth`.

The `maxwidth` parameter represents the maximum width of the peak. If the identified peak is greater than `maxwidth`, the peaks will be split until the width of all peaks is less than 1000 bp.

```
#################### 5. findPeaksByStrand ######################
## findPeaksByStrand: find peaks based on BAM coverages
forwardPeaksFile=paste0(inputBam.u.dedup.seps[1], '.peaks')
forwardPeaksFile <-findPeaksByStrand(bamFile=inputBam.u.dedup.seps[1],
                                     chrs=trap.params$chrs,
                                     strand='+',
                                     L=trap.params$readlength,
                                     maxwidth=trap.params$maxwidth,
                                     cutoff = trap.params$cov.cutoff,
                                     ofile=forwardPeaksFile,
                                     logf=logf)

scAPAtrap:::.checkAndPrintFiles(varnames='forwardPeaksFile',
                                filenames=forwardPeaksFile, logf=logf)

reversePeaksFile=paste0(inputBam.u.dedup.seps[2], '.peaks')
reversePeaksFile <-findPeaksByStrand(bamFile=inputBam.u.dedup.seps[2],
                                     chrs=trap.params$chrs, strand='-',
                                     L=trap.params$readlength,
                                     maxwidth=trap.params$maxwidth,
```

```
                                    cutoff = trap.params$cov.cutoff,
                                    ofile=reversePeaksFile, logf=logf)

scAPAtrap:::.checkAndPrintFiles(varnames='reversePeaksFile',
                                filenames=reversePeaksFile, logf=logf)
```

4.6 Step 6. generateSAF to get the full peak list

This step simply combines the two output files from `findPeaksByStrand`.

```
################## 6. generateSAF ######################
## generateSAF: generate a .SAF file that records the information of identified peaks
peaksfile <- generateSAF(forwardPeaks=forwardPeaksFile,
                         reversePeaks=reversePeaksFile,
                         outputdir=outputDir,
                         logf=logf)

scAPAtrap:::.checkAndPrintFiles(varnames='peaksfile',
                                filenames=peaksfile, logf=logf)
```

4.7 Step 7. generateFinalBam to generate a new peaks' BAM file with featureCounts

This step generates a new BAM file for the peak file generated from `generateSAF` with `featureCounts`. The new BAM file will be used by the `step 8. countPeaks` step for quantifying peaks, i.e., counting reads in each peak. Here we used `inputBam`, which would be similar to using the BAM file after de-duplication `inputBam.u.dedup` from `step 3. dedupByPos`. This is because the following `step 8` counts peaks with `umitools`, which will do de-duplication.

```
################# 7. generateFinalBam ######################
## generateFinalBam: get a small BAM that contains the peak regions.
## Here the inputBam is used, which will contain duplicated reads.
final.bam <- generateFinalBam(tools$featureCounts,
                              tools$samtools,
                              input=inputBam,
                              peakfile=peaksfile,
                              thread=trap.params$thread,
                              logf=logf)

scAPAtrap:::.checkAndPrintFiles(varnames='final.bam',
                                filenames=final.bam, logf=logf)
```

4.8 Step 8. countPeaks to quantify peaks with umitools

This step quantifies peaks using the BAM file generated by `step 7. generateFinalBam` with `umitools`, which will do de-duplication on the BAM file. This step may take a few minutes depending on the size of the BAM file.

```
################# 8. countPeaks ########################
## countPeaks: count read number in each peak with umitools
countsfile<- countPeaks(tools$umitools,
                        input=final.bam,
                        outputdir=outputDir,
                        TenX=trap.params$TenX,
                        logf=logf)

scAPAtrap:::.checkAndPrintFiles(varnames='countsfile',
                                filenames=countsfile, logf=logf)

## record output file for reducePeaks (if tails.search=peaks)
```

4.9 Step 9. `findTails` to search polyA tails genome-wide or peak-wide

The peak calling step identifies potential regions where poly(A) sites are located. We can then arbitrarily set the 3' end of each peak as the coordinate of the respective poly(A) site. However, this is not the best solution to determine the precise position of the poly(A) site. We can incorporate reads with A/T stretches to pinpoint poly(A) sites, which was implemented as function `findTails`.

The `trap.params$tails.search` parameter sets the strategy to search polyA tails, which can be set `peaks`, `genome`, `no` (default).

- `no`: Not search tails, which means the final peaks are not linked with any polyA-tails.
- `genome`: Search tails genome-wide, which may be suitable for small genome. Otherwise it will take very long time or MEM.
- `peaks`: Search tails within the peak regions by `d` nt. This way may be suitable for big genome but with relatively small number of peaks.

Here the `inputBam` (which has duplication) is used for tail search, which would search tails as many as possible. For searching tails peak-wide (`tails.search=peaks`), lowly expressed peaks were first removed to reduce peak number (based on two parameters – `min.cells` and `min.counts`), as these peaks will eventually removed in the last step which no need searching tails.

Two issues should be considered using this step:

1) It may take very long time or big MEM to search tails for large genome (`tails.search=genome`) or very large number of peaks (`tails.search=peaks`). In such case, it is recommended to skip this step (set `tails.search=no`). **But we can still search tails for a small number of peaks that with small read counts to validate these lowly expressed peaks.** Because it is very probable that peaks with large counts are real but it is unsure for peaks with low counts.
2) Linking tails with peaks will definitely remove a quite large number of peaks, which should be used in caution. Again, we can search tails for a small number of peaks that with small read counts (using `reducePeaks`), and retain peaks with large counts without linking tails.

It is encouraged to use `tails.search=no` (means not searching tails) to run scAPAtrap first. And then we can search tails and link tails to peaks later. Please see the below `Different tails.search strategy` section for details.

```
################## 9. findTails ######################
## tails.search=no
tailsfile=NULL

countsPeaksFiles=list(countsfile=countsfile, peaksfile=peaksfile)

scAPAtrap:::.checkAndPrintFiles(varnames='tailsfile',
                                filenames=tailsfile, logf=logf)
```

In the meantime, we can also try `tails.search=genome` or `tails.search='peaks'`. Then additional `.tails` files will be generated. Caution: this step may take long time for large genome.

```
## findTails: peaks, genome

## tails.search=peaks
if (trap.params$tails.search=='peaks') {
  #8.1# reducePeaks: remove small peaks to speed up tail-searching
  countsPeaksFiles <- reducePeaks(countsfile=countsfile,
                                  peaksfile=peaksfile,
                                  min.cells = trap.params$min.cells,
                                  min.count = trap.params$min.count,
                                  suffix='.reduced', logf=logf)

  peaksfile.reduced=countsPeaksFiles$peaksfile

  scAPAtrap:::.checkAndPrintFiles(varnames='peaksfile.reduced',
                                  filenames=peaksfile.reduced, logf=logf)

  #8.2# findTailsByPeaks: find tails nearing peaks.
  tailsfile=paste0(inputBam, '.peaks.tails')
  tailsfile <- findTailsByPeaks(bamfile = inputBam,
                                peaksfile=peaksfile.reduced,
                                d=trap.params$d,
                                tailsfile=tailsfile, logf=logf)

  scAPAtrap:::.checkAndPrintFiles(varnames='tailsfile',
                                  filenames=tailsfile, logf=logf)

## tails.search=genome
} else if (trap.params$tails.search=='genome') {
  #8# findTails: find tails genome-wide.
  tailsfile=paste0(inputBam, '.genome.tails')
  tailsfile <- findTails(bamfile = inputBam,
                         tailsfile=tailsfile, logf=logf)

  scAPAtrap:::.checkAndPrintFiles(varnames='tailsfile',
                                  filenames=tailsfile, logf=logf)
}
```

4.10 Step 10. generatescExpMa to generate final scAPAtrapData with peaks.meta and peaks.count

This step combines peaks' meta data (chr/strand/start/end) and peaks' count data (peak-cell count matrix) to a `scAPAtrapData`. `scAPAtrapData` is stored in scAPAtrapData.rda, which is a list containing two matrices – `peaks.meta` and `peaks.count`. This RDA file can be further loaded into other tools like `movAPA` or `Seurat` by scAPAtrap's function `convertAPAtrapData` for downstream analysis.

```
################### 10. generatescExpMa ######################
## generatescExpMa: generate final PA data including peaks and counts information
outputfile=paste0(outputDir, '/scAPAtrapData.rda')
outputfile <- generatescExpMa(countsfile=countsPeaksFiles$countsfile,
                              peaksfile=countsPeaksFiles$peaksfile,
                              barcode=trap.params$barcode,
                              tails=tailsfile,
                              d=trap.params$d,
                              min.cells=trap.params$min.cells,
                              min.count=trap.params$min.count,
                              ofile=outputfile,
                              logf=logf)
## Final scAPAtrapData.rda
scAPAtrap:::.checkAndPrintFiles(varnames='outputfile',
                                filenames=outputfile, logf=logf)
```

4.11 Step 11. list and clean output files

`TRAPFILES` is a global variable that stores output file names and description, which is automatically appended during running functions in scAPAtrap. We can print `TRAPFILES` at any time to see current output files. We can also call `TRAPFILES(clear=TRUE)` to clear the file list in `TRAPFILES`.

```
################### 10. clean TRAPFILES ######################
## show TRAPFILES for cleanning
scAPAtrap:::.printTRAPFILES(nc=50, logf=logf)
```

5 Downstream analysis

`scAPAtrapData.rda` stores a object named `scAPAtrapData`, which is a list containing two matrices – peaks.meta and peaks.count. This file can be further loaded into other tools like movAPA by scAPAtrap's function `convertAPAtrapData` for downstream analysis. Please refer to movAPA_on_scAPAtrap_results for details about how to read the `scAPAtrapData` and analyze it with movAPA.

```
install.packages("devtools")
require(devtools)
install_github("BMILAB/movAPA")
library(movAPA)
browseVignettes('movAPA')
```

Furthermore, we can also use the vizAPA package for visualization and DE detection.

```
install_github("BMILAB/vizAPA")
library(vizAPA)
browseVignettes('vizAPA')
```

6 Restart scAPAtrap from a middle step

Due to some reasons, such as the BAM file being too large, tail search failure, unusual chromosome name, or other unknown reasons, the one step program may terminate abnormally at a certain step. Users can view the `logf` log file, which records all steps and their command lines, intermediate information, and output files. Users can check and see which step they have reached, as well as the output file they have obtained so far (check `TRAPFILES()` or log file `logf`), and then continue running according to the tutorial of `step by step running scAPArap` from that step.

7 Different `tails.search` strategies

Due to the time and memory consumption of the `findTails` step, and the results of tails will used to filter out peaks, which may wrongly remove real peaks. And different tails.search strategies may have very different but significant impact on the final peak results. **Therefore, it is encouraged to run `scAPAtrap` with `tails.search='no'` first to generate only peaks without considering polyA tails.** Users can set `min.cells` and `min.counts` in `generatescExpMa` to filter peaks with required expression level for downstream analysis.

However, it is also probably that peaks that do not meet the requirement are real. Then users can run `findTails` or `findTailsByPeaks` separately to get `tailsfile`, which is also a peak-like file storing tail positions. `findTailsByPeaks` allows searching tails near and within peaks, which could be useful to find more polyA-evidence for not highly expressed peaks. After `findTailsByPeaks` or `findTails`, users can then run `generatescExpMa` again to link peaks with tails, which can output those peaks with tail supporting. These tail-supported peaks can be combined with those highly expressed peaks, which are considered as final peaks.

The following shows some code examples to use customized peak list for tail searching.

```
#### to search tails genome-wide
tailsfile=paste0(inputBam, '.genome.tails')
tailsfile <- findTails(bamfile = inputBam, tailsfile=tailsfile)

#### or to search tails peak-wide
# After running the scAPAtrap's pipeline, countsfile and peaksfile stores the peaks.meta and peak.count
# It is easy to filter peaks that are not highly-expressed but moderately expressed.

# first, generate a peak file with reduced peaks, here peaks with moderately expression level can be us
# For example, if we consider peaks with 50 cells and 100 counts are definitely true peaks,
# and consider peaks with <10 cells and <20 counts are definitely false peaks,
# but peaks between 10~50 cells and 20 can 100 counts could be true or false.
# Then we can retain these moderately peaks for tail searching.

# to retain peaks with >=10 cells but <50 cells and >=20 counts and <100 cells as unsure peaks
peaks.unsure <- reducePeaks(countsfile, peaksfile,
                        min.cells = 10, min.count = 20,
                        max.cells = 49, max.count = 99,
                        suffix='.reduced', ...)
```

```
# to retain those definitely true peaks too (>=50 cells, and >=100 counts)
peaks.true <- reducePeaks(countsfile, peaksfile,
                          min.cells = 50, min.count = 100,
                          max.cells = NULL, max.count = NULL,
                          suffix='.true', ...)

# to search tails for peaks.unsure
tailsfile=paste0(inputBam, '.peaks.unsure.tails')
tailsfile <- findTailsByPeaks(bamfile = inputBam, peaksfile=peaks.unsure, d=trap.params$d, tailsfile=ta

# then use these tails to filter peaks, which are unsure-peaks supported by tails
outputfile=paste0(tailsfile,'.peaks.rda')
outputfile <- generatescExpMa(countsfile=peaks.unsure[1],
                              peaksfile=peaks.unsure[2],
                              barcode=trap.params$barcode,
                              tails=tailsfile, d=trap.params$d,
                              min.cells=1, min.count=1,
                              ofile=outputfile, logf=logf)

# then we can combine peaks.true and outputfile
```

8 Issues and potential solution

- chromosome names are not consistent between BAM file and peaks.

If any provided `chrs` start with digit (like '1', '2L'), then in some functions of scAPAtrap's pipeline `chrs` will be automatically added "chr" (like 'chr1', 'chr2L'). Normally, chr names will be restored after running these functions. However, if there are still inconsistency, users can modify chr names in the file (e.g., `peaksfile`) or some other matrix or data.frame variables. And run the function again.

```
## some code like this to remove 'chr'
dataframe$chrs=gsub('chr', '', dataframe$chrs)

## some code like this to add 'chr'
dataframe$chrs=paste0('chr', dataframe$chrs)
```

- Insufficient memory or program running for too long.

This issue normally happens in `findTails`. Please see the above `step 9` and `Different tails.search strategy` section for details.

9 Another demo data

Please visit this link to download some demo fastq files.

Run `umi_tools` to extract barcodes and UMIs, and add them to read names. For detailed usage of the `umi_tools` tool, refer to its help document

```
input.seq <- c('./demo_1.fastq','./demo_2.fastq')
pattern <- 'CCCCCCCCNNNN'
whitelist <- './whitelist.txt'
extractBcAndUb(umitools.path, pattern, input.seq, whitelist)
```

Next, we can build the reference genome index and align the fastq reads. For raw fastq file, please refer to the CellRanger tool on the official website of 10x. For 3'-enriched scRNA-seq technologies other than 10x (e.g., CEL-seq2), cellranger is not applicable to the processing of raw fastq file, users may use STARsolo instead.

Please visit this link to download another demo BAM data. This demo data is the data of chromosome 1 in GSE104556. You can also use the entire GSE104556 to run the entire process.