Documentation of S–MART

Matthias Zytnicki

March 14, 2012

Contents

1 Introduction

S–MART should be pronounced "ess-mart" (better pronounced with a slight Spanish accent) and stands for "Short reads MART". It could have many other meanings. The one you choose is the best.

It provides a set of Python scripts which transform your short reads which have been mapped to a genome. So, it supposes that you already have mapped your data to a reference genome. For that, you can use Maq, ZOOM, Mosaik or any other tool. S–MART supports many formats.

You can also compare S–MART with other data, such as RefSeq sequences or any kind of annotation, as long as their formats are supported by S–MART (which is usually the case). However, S–MART does not include these data, simply because they are too many of them, and too many organisms. If you want to known where I get my data, read Appendix B.

2 Installation and requirements

Depending on the system you are using installation can be different. However, in both cases, you will need Python, SQLite, R and Java.

Python Python is the language used to code the algorithms. Why Python? Well, why not?

SQLite SQLite is a database management system which is used to handle efficiently the reads (remember that there can be several millions reads, so that every algorithmic improvement is highly important). Normally, you should not even see that S–MART uses some databases, since it reads flat files (like GFF files), and outputs flat files. But, internally, it uses databases. If you want to know how SQLite is used, you can read Section 6.3.

R R is a statistical computing tool which can do many things, but, here, it is only used to plot the data.

Java Java is used here simply because it contains a good graphical user interface. So, it is used for the GUI in S–MART.

2.1 For Windows

I bundled everything you need (Python and R) in an archive. Simply choose the Windows version of S-MART on the Web page.

Otherwise, setting up your system for S–MART should not take more that 15 minutes.

Python You should have downloaded and extracted a bundle which contains all the Python scripts files. First check that Python version 2.5 is installed. You should get it from their Web site.

Each Python script uses many other scripts (basically, the structure of the files is organised by classes). So, you should add to your PATH variable the directory where you have installed the scripts. To do so, click on `My Computer`, then `Control Panel`, `System`, `Advanced`, `Environment variables`. Click on `New` and add the following variables:

name: `PYTHONPATH`, value: *`where_you_installed_S-MART`*

SQLite SQLite is already embedded into Python. Cool! No need for installation!

R Again, download R *using* CRAN mirror download page. Accept all default choices, but select "Yes" when the R installer asks you "Do you want to costumize the startup options?" (otherwise, no data is added into the registries and I will not be able to guess where you have installed R).

Two packages are furthermore needed: RColorBrewer, to have a good palette of colors (I am color blind, so it is important to me), and Hmisc, which does Spearman correlations (among others). There are many ways to do so. The simplest one is to start R (find it in you `Programs` menu), and write:

```
install.packages("RColorBrewer", dependencies = TRUE)
install.packages("Hmisc", dependencies = TRUE)
```

R will probably ask you some questions, that you could blindly answer `y` (yes). It may also ask you the repository from where you can download the packages. You can choose the closest location. Alternatively, I know that the mirror from Toulouse, France, works well. Downloading the second package, Hmisc, takes some time because it has a lot dependencies.

Quit with `q()`. Done.

Java In the very unlikely case you had not Java (it is now a standard tool used with your Web browser), you can download it and install it afterwards.

2.2 For Mac

This has been tested on a MacOS 10.6.3.

You should follow the same instruction as the next section, which is dedicated to Linux. MacOS is based on Linux after all.

However, if you experience some problems you may read the following advice.

Open a Terminal A terminal reminds the old way of the computer, when mice where not related to computers. It is a place where you can *write* instructions to the computer. It is sometimes very useful.

To open a terminal, open the Finder, then click on "Applications" on the left. Look for the "Utilities" directory, then double click on "Terminal."

Python Install Python 2.6.5 from their Web site.

Each Python script uses many other scripts (basically, the structure of the files is organised by classes). So, you should add to your PATH and PYTHONPATH variables the directory where you have installed the scripts. To do so, supposing you use the standard Bash shell, open your `.bashrc` file on your root directory and add the following line at the end of the file:

```
export PATH=$PATH:the_directory_where_you_installed_the_files
export PYTHONPATH=$PYTHONPATH:the_same_directory
```

SQLite SQLite is already embedded into Python. Cool! No need for installation!

R The last version of R, available for MacOS X 10.5 (Leopard) and higher, can be download for the R Website. You will also probably need the GFortran compiler (for the scientific librairy of R) and Tcl/Tk (for the graphical user interface).

If you have MacOS X 10.4 (Tiger), there is a nice installer with everything on it.

In the Finder, you should now find the `R` in the "Applications." Open it and type:

```
install.packages("RColorBrewer", dependencies = TRUE)
install.packages("Hmisc", dependencies = TRUE)
```

R may ask you the repository from where you can download the packages. You can choose the closest location. Alternatively, I know that the mirror from Toulouse, France, works well. Downloading the second package, Hmisc, takes some time because it has a lot dependencies.

Quit R with `q()`. It will ask you if you want to store the context. Answer no (`n`). Done.

Java S–MART needs Java 1.6 to work. Unfortunately (blame Apple for that!), Java 1.6 is not available for Mac OS X Tiger. For later versions, Java should not be a problem.

2.3 For Linux

This has been tested on a Debian 2.22.3.

Python You should have downloaded and extracted a bundle which contains all the Python scripts and the Java files. First check that Python is installed. I can guarantee that it works for version 2.5.2. Versions earlier than 2.4 will not work. Later versions may work. Or not.

Each Python script uses many other scripts (basically, the structure of the files is organised by classes). So, you should add to your PATH and PYTHONPATH variables the directory where you have installed the scripts. To do so, supposing you use the standard Bash shell, open your `.bashrc` file on your root directory and add the following line at the end of the file:

```
export PATH=$PATH:the_directory_where_you_installed_the_files
export PYTHONPATH=$PYTHONPATH:the_same_directory
```

SQLite SQLite is already embedded into Python. Cool! No need for installation!

R Again, download R *via* your package manager or using the CRAN mirror download page.

One package is furthermore needed: RColorBrewer, to have a good palette of colors, and Hmisc, for the Spearman correlations. There are many ways to do so. The simplest one is to start R (type `R` in a console), then write:

```
install.packages("RColorBrewer", dependencies = TRUE)
   install.packages("Hmisc", dependencies = TRUE)
```

R will probably ask you some questions, that you could blindly answer `y` (yes). It may also ask you the repository from where you can download the packages. You can choose the closest location. Alternatively, I know that the mirror from Toulouse, France, works well. Downloading the second package, Hmisc, takes some time because it has a lot dependencies.

Quit with `q()`. R will ask you if you want to store the context. Answer no (`n`). Done.

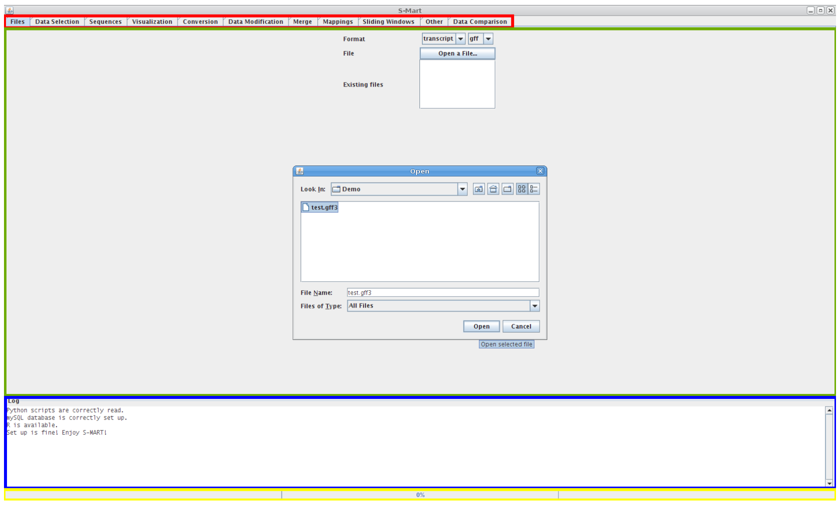
In case it does not work, you will have to download the RColorBrewer and Hmisc packages, then install them with:

```
R CMD INSTALL -l where/you/dowloaded/the/packages RColorBrewer_xxx.tar.gz
R CMD INSTALL -l where/you/dowloaded/the/packages Hmisc_xxx.tar.gz
```

Java I guess you already have a Java somewhere, have not you?

2.4 Test the configuration

To check if everything is correctly set up double-click on the `Smart.jar` icon, located where you download S–MART. On Linux, you can also open a terminal, go to the installation directory and write `java -jar Smart.jar`. The first time



Format: transcript, gff
File
Existing files

Open
Look In: Demo
test.gff3
File Name: test.gff3
Files of Type: All Files

Log

```
Python scripts are correctly read.
mySQL database is correctly set up.
R is available.
Set up is fine! Enjoy S-MART!
```

Figure 1: The "file" panel of the GUI.

you start it, S–MART might ask you where the executable files for Python and R are (for the Windows users, they might be at `C:\Python25\Python.exe` and `C:\Progam Files\R\R-2.10.0\bin\R.exe` respectively, depending on the versions of you executables).

Is it OK? Cool! You can now start with your S–MART experience. Otherwse, you may drop me email and I might help you with the configuration.

3 General description

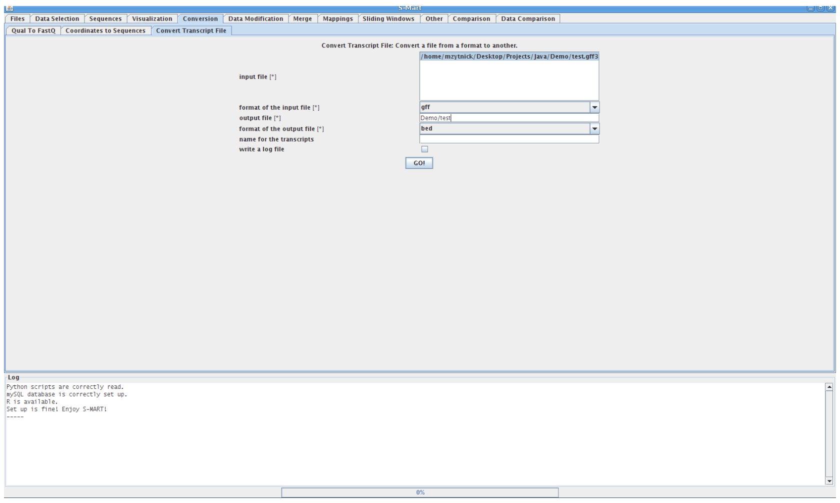
The GUI I have developped a graphical user interface so that every tool can be started easily.

On Windows, start it by double clicking on the `Smart.jar` file. On Linux, type `java -jar Smart.jar`.

You have to set the files that you will use (together with their formats) in the first panel. In the other panels, you will be able to start all the tools S–MART contains.

Figure 1 shows the GUI on the "file" panel. The GUI is divided into four regions, from top to bottom. On the top region (in the red rectangle), you can select the type of action you want to perform by selecting the right panel. The second area (the green one) lets you perform the task. The third region (the blue one) is the log area, where you can interactively read the output of the programs, or any relevant information. The last area (the yellow one) is a progress bar and shows you how much time the program will run to perform the task.

On the "file" panel, you can submit the files that you are going to use, together with their format. In the example, we enter the file `test.gff3`, which



Convert Transcript File: Convert a file from a format to another.

input file [*]: /home/mzytnick/Desktop/Projects/Java/Demo/test.gff3
format of the input file [*]: gff
output file [*]: Demo/test
format of the output file [*]: bed
name for the transcripts:
write a log file: ☐

```
Python scripts are correctly read.
mySQL database is correctly set up.
R is available.
Set up is fine! Enjoy S-MART!
-----
```

Figure 2: The "Convert Transcript File" program panel of the GUI.

is a transcript list in GFF3 format. First select the type of data: mapping data (coming from your mapper), transcripts and other files. Then, select the right format. As you can see, S–MART supports many formats. Finally, click on the button `Open a File` to browse your hard disk and select the right file.

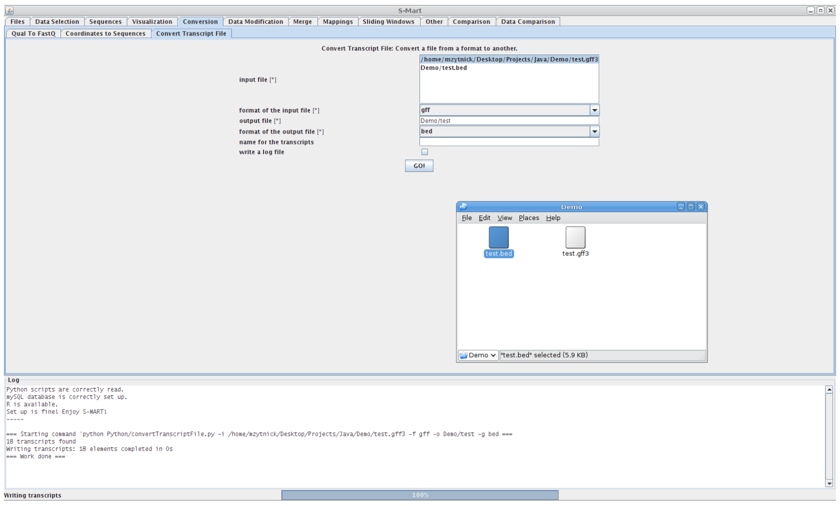
You can then use any tool of the toolbox by changing the panel. Figure 2 shows a conversion utility tool. Then, we select the file that we have mentionned (`test.gff3`) to convert it into a BED file. We specify that the input file is in GFF3 and that the output file is in BED format. We also specify the output file name (do not write the extension: S–MART adds it by itself). We click on the button, the program starts, as visible in the log area. Finally, the output file appears. We can open it in a file browser (see Figure 3)

The command line For the real hackers, every tool can be used in command line. All the scripts are in the `Python` directory and you can start them there. They all have several parameters that you can adjust depending on what you want to do, so a typical command would be:

```
python mapperAnalyzer.py -i mappedData.psl -f psl
 -q rawData.fasta -o coordinates -n 1 -s 100 -m 0
  -p 0 -e -x -r -b -B -g -G -u -U -2 -y -c green
             -t shortReads -v 50 -l
```

(It should be only one line but it does not fit in the page. Do not be afraid, commands usually are shorter.)

An important paramater is the `-v` *`number`* option, which gives the verbosity level (highest is most verbose). Another general option is the `-l` option, which writes a log file (usually utterly verbose, but sometimes useful). A last useful



Convert Transcript File: Convert a file from a format to another.

input file [*]: /home/mzytnick/Desktop/Projects/Java/Demo/test.gff3
Demo/test.bed

format of the input file [*]: gff
output file [*]: Demo/test
format of the output file [*]: bed
name for the transcripts:
write a log file: ☐

Demo "test.bed" selected (5.9 KB)

```
Python scripts are correctly read.
mySQL database is correctly set up.
R is available.
Set up is fine! Enjoy S-MART!
-----

=== Starting command 'python Python/convertTranscriptFile.py -i /home/mzytnick/Desktop/Projects/Java/Demo/test.gff3 -f gff -o Demo/test -g bed ===
18 transcripts found
Writing transcripts: 18 elements completed in 0s
=== Work done ===
```

Writing transcripts

Figure 3: The result of the tool.

option is `-h`, which displays a notice, a comment for each option and exits. Moreover, the `-y`, which is sometimes available, keeps the output file in the internal representation of S–MART (which is in a SQLite database). If you want to know why it could be useful to use this option, read Section 6.3.

4 Which tool for your need?

This section present the scripts that you may want to use for a particular task.

4.1 Mapping conversion

Once you have used your preferred mapping tool, you may want to play a bit with your data. Actually, it is not straightforward, since mapping formats (such as SAM, for instance) and transcript formats (such as GFF3) are quite different. Mapping formats generally include information about the reads (number of mismatches, number of gaps, number of mappings, etc.) which are usually not relevant for the transcript formats. So, first thing you can do is to convert your data into transcript format. You may also want to select the mappings using some criteria (you may want to exclude the reads which have mapped several times, for instance). The following tools do that for you.

`mapperAnalyzer.py` The first program you may use is `mapperAnalyzer.py`. It reads a set of mapping given by the tool you have used to map your data on the reference genome and translate it to a set of genomic coordinates. You also have the possibility to extract only those that you are interested in (few matches in the genome, few errors in the mapping, etc.). You can also select those reads which map less than a given of times in the genome. Moreover, you can output the data in various different formats, which you can use to visualize them *via* UCSC genome browser or GBrowse[1]. Unmatched reads can be written in an other file, in case you would like to try to map them with another tool (may sometimes work!).

The script can parse data given by the following programs (the corresponding option is given in parenthesis):

- Blast (use `-m 8` format for Blast and `-f blast`)
- Blat (`-f psl`)
- BowTie (`-f bowtie`)
- Exonerate[2] (`-f exo`)
- Maq (`-f maq`)
- Mosaik (output in axt format for Mosaik and use `-f axt`)
- Nucmer (`-f nucmer`)
- Rmap (`-f blast`)
- Seqmap (`-f seqmap`)
- Shrimp (`-f shrimp`)
- Soap (`-f soap`)
- Soap2 (`-f soap2`)
- and more. . .

You can filter your data according to:

- number of errors in the mapping
- number of occurrences of the mapping in the genome
- size of the read mapped
- number of gaps in the mapping

The script needs an input file (your mapped reads) together with its format and the read sequences file together with its format (FASTA or FASTQ). If you want, you can also append the results of this script to another GFF3 file. This is useful when the GFF3 file is the result of the mapping using another tool.

By default, any gap in the alignment to the reference sequence is treated like an exon. You can decide to remove this feature by merging short introns (actually, gaps).

[1]Look at Appendix A to know more about it.

[2]Exonerate can display its results in many formats. Currently, S–MART only support the following output format:
```
--ryo "%S %em %V\n" --showvulgar FALSE --showalignment FALSE
```
Please add these parameters to your command line while using Exonerate!

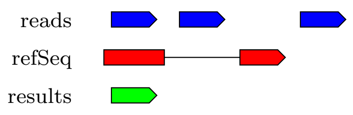


Figure 4: Simple comparison between your reads and RefSeq data, for example, using `compareOverlapping.py`.

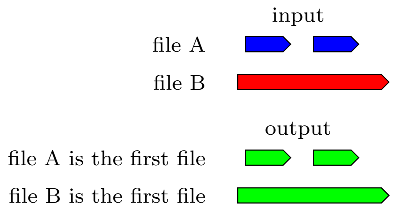


Figure 5: Graphical representation of the content of files A and B. Their order in `compareOverlapping.py` is important.

`mappingToCoordinates.py` If you just want to convert your mapping data to genomic coordinates, without any filtering, you can use `mappingToCoordinates.py`. It needs a mapping file (output of your mapper) together with its format, an output format (GFF3, BED) and prints you the corresponding file.

4.2 Data comparison

This section presents you several ways to compare to different sets of transcripts.

`compareOverlapping.py` This script may be the most important one. It basically compares two sets of transcripts and keeps those from the first set which overlap with the second one. The first set is considered as the query set (basically, your data) and the second one is the reference set (RefSeq data, for example, see Figure 4).

It is vital to understand that it will output the elements of the first file which overlap with the elements of the second one. Look at figure 5. If file A is the first file, it will output 2 elements, whereas if file B is the first file, it will output only 1 element.

Various modifiers are also available:

- Restrict query / reference set to the first nucleotide. Useful to check if the TSS of one set overlap with the other one.
- Extend query / reference set on the 5' / 3' direction. Useful to check if one set is located upstream / downstream the other one.
- Include introns in the comparison.
- Invert selection (report those which do not overlap).

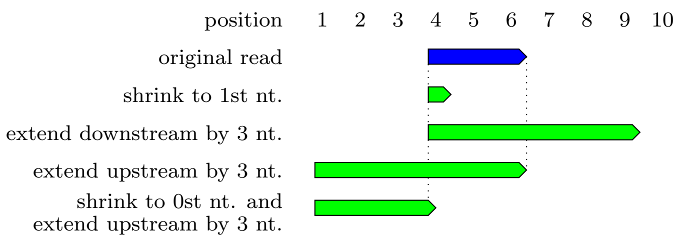


Figure 6: Shrinking and extending your data before comparison with `compareOverlapping.py`.

- Restrict to colinear / anti-sense overlapping data.
- Keep the query data even if they do not strictly overlap with the reference data, but are located not further away than *n* nucleotide from some reference data.
- Keep the query data with are strictly included into reference data, meaning that a query transcript such that at least 1 nucleotide does not overlap with reference data will not be presented as a solution.

The mechanism of shrinking and extending is also useful to make a fine grain comparison. For example, if you want to keep those such that the TSS is overlapping the reference set, you just shrink the query set to 1 nucleotide (see Figure 6). Now, if you want to keep those which are overlapping you data or located 2kb downstream of it, just extend the query data in the downstream direction, and you will have what you want. You can also extend in the opposite direction to get the possible transcript factor sites which are upstream.

Some option reverses the selection. Put in other words, it performs the comparison as usual, and outputs all those query data which do not overlap.

`FindOverlapsOptim.py` This tool is basically the same as the previous one, `compareOverlapping`, except that this tool is highly optimized. So there is hardly no option, which may slow down the procedure. So this tool quickly compares two sets of annotations, that is all.

`getDifference.py` This tools has two different (but similar) uses, as examplified in figure 7. When given two sets of transcripts, it trims the elements of the set so that they do not overlap with the second set (like the “result 1” line in the figure).

When only one set of transcripts is given, together with a reference genome, it produces a list of transcripts which complements the first set (like the “result 2” line in the figure).

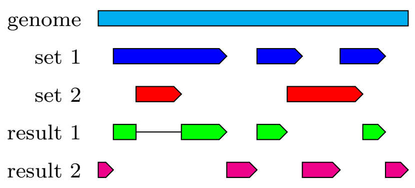


Figure 7: Examples of `getDifference.py`. The “result 1” line compares the two sets of transcripts. The “result 2” line compares the first set of transcripts with the genome.

`getDifferentialExpression.py` This tool compares two sets of data and find the differential expression. One very important component of the tool is the reference set. Actually, to use the tool, you need the two input sets of data, of course, and the reference set. The reference set is a set of genomic coordinates and, for each interval, it will count the number of feature on each sample and compute the differential expression. For each reference interval, it will output the direction of the regulation (up or down, with respect to the first input set), and a p-value from a Fisher exact test (see figure 8).

This reference set seems boring. Why not computing the differential expression without this set? The answer is: the differential expression of what? I cannot guess it. Actually, you might want to compare the expression of genes, of small RNAs, of transposable elements, of anything. . . So the reference set can be a list of genes, and in this case, you can compute the differential expression of genes. But you can also compute many other things.

Suppose that you cluster the data of your two input samples (you can do it with the `clusterize` and the `mergeTranscriptLists` tools). You now have a list of all the regions which are transcribed in at least one of the input samples. This can be your reference set. This reference set is interesting since you can detect the differential expression of data which is outside any annotation.

Suppose now that you clusterize using a sliding window the two input samples (you can do it with the `clusterizeBySlidingWindows` and the `mergeSlidingWindowsClusters` tools). You can now select all the regions of a given size which contain at least one read in one of the two input samples (do it with `selectByTag` and the tag `nbElements`). Again, this can be an other interesting reference set.

In most cases, the sizes of the two input samples will be different, so you should probably normalize the data, which is an available option. The —rather crude— normalization increases the number of data in the least populated sample and decreases the number of data in the most populated sample to the average number of data.

You can also plot the differential expression. A point (x, y) refers to a reference interval which contains x data in the first sample and y data in the second sample. If you normalized the data, then the plot reports the normalized figures.

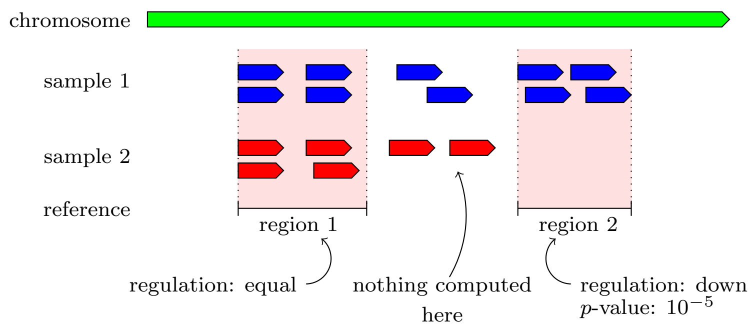


Figure 8: Differential expression computed on two reference intervals.

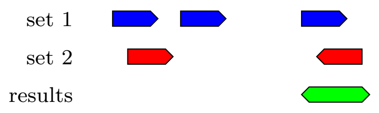


Figure 9: Finding transcription on both strands using `mergeTranscriptLists.py`.

4.3 Merging data

This section presents you some ways to merge two sets of transcripts, or clusterize a set of transcript.

4.3.1 Clustering

`mergeTranscriptLists.py` The script is similar to `compareOverlapping.py`, except that when data of two different sets overlap, they are merged. You can use the same parameters as `compareOverlapping.py` and use them to look for transcription on both strands, for example (see Figure 9).

Optionally, you can also add to the output all the elements from the first set which do not overlap with the second set.

`clusterize.py` The script clusterizes the reads. Two reads are clusterized when their genomic intervals overlap (see Figure 10). The output is a GFF3 file, where each element is a cluster. The number of elements in the cluster is given by the tag `nbElements`. The name of a cluster is the concatation of the names of its reads (like `read1--read2--read3`). Note that if the size of the name of the cluster exceeds 100 characters, it is truncated to the first 100 characters.

Some options may clusterize the features which are closer than a given distance.

By default, the tool clusterizes all features which overlap (or nearly overlap), even if they are on different strands. If you want to clusterize the features which

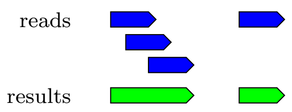


Figure 10: Clustering using `clusterize.py`.

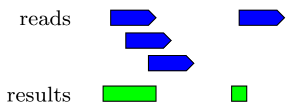


Figure 11: Finding TSS using `findTss.py`.

are on the same strand only, you can specify it.

`findTss.py` This script is specially useful when you have 5' capped reads, that is to say, when the reads that you have mark the beginning of the transcripts. This script find all the TSS that are found by your data (see Figure 11).

In some —most, actually— cases, there is no clear TSS, but a stretch of possible TSSs. So you can choose the maximal distance between two reads for them to mark the same transcription start (for example, two reads that are distant by 20 nt. can mark 1 or 2 TSS, depending on the value of a parameter).

You can plot the distribution of the number of reads per TSS: a point (x, y) tells you that y transcripts starts are marked by x reads. The plot has sometimes a long tail towards the high values in the x-axis, so you can zoom to plot only the first points on this axis by using some parameters.

`collapseReads.py` Merge two input genomic coordinates iff they are exactly the same (see figure 12). If two or more genomic coordinates are merged, the tag `nbElements` is updated accordingly (look at Section 6.2 if you want to know more about tags). As a consequence, all the reads which are exactly the same appear as one genomic coordinate.

This is especially useful for short RNA sequencing (where you want to count the number of read per miRNA, siRNA, etc.) or 5' capped short reads.

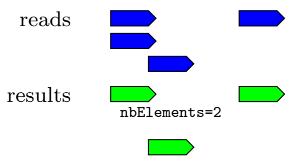


Figure 12: Example of execution of `collapseReads.py`.

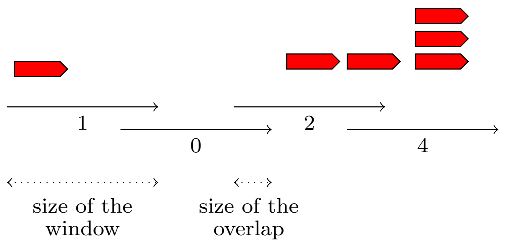


Figure 13: Sliding windows: counting the number of reads.

4.3.2 Sliding Windows

Sliding windows are a convenient ways to clusterize data mapped on the genome. There are two important parameters of a sliding window: the size of the window and the size of the overlap. In Figure 13, a sliding window counts the number of reads.

`clusterizeBySlidingWindows.py` By default, sliding windows count the number of reads in each window. However, you can basically merge any information which is contained in the tags. You can compute the average, sum, median, max or min of the tags for each window. For instance, every window can contain the average cluster size, if you merge clusters instead of reads.

The output file is a GFF3 file, where each element is a window. There is a special tag for each window, whose name is `nbElements` if you counted the number of transcripts per sliding window. However, if you performed a "min" (resp. "max", "sum", "median", "average") operation on the tags `value` of the transcripts, then the tag of the window will be `minValue` (resp. `maxValue`, `sumValue`, `medValue`, `avgValue`). You can also specify the name of your tag (which is actually advised: `nbReadsInSample1` will always be more informative than `nbElements`).

You also have different option, which can select the n% highest regions, or the regions with at least n features in it, or even the regions with at least n unique features. This last option is useful when you want to cluster the reads which have mapped only once, for instance.

`mergeSlidingWindowsClusters.py` Sliding windows are also useful to compare two (or more!) sets of data. This can be very valuable when you want to compare differential expression in two different conditions. When you have two different sliding windows sets, this function merges them into one, where each window contains the two pieces of information. You may want to plot the data afterwards using the `plot.py` function.

A good motivation for this tool is given in Section 5.4. Suppose that you have two sets of reads, for two different conditions on the same genome. What you can do is use a sliding window for each condition using `clusterizeBySlidingWindows.py`.

Now, to perform any comparison, you will have to merge the two conditions into a single file. This is were you need `mergeSlidingWindowsClusters.py`.

The tool needs two files given by `clusterizeBySlidingWindows.py` together with their format (GFF3, actually) and outputs a new file, in GFF3 format.

4.4 Data selection

This set of scripts reads a list of sequences or genomic coordinates and select those with some given simple properties.

4.4.1 Sequences

`restrictFromSize.py` Reads a list of sequences or genomic coordinates and outputs those which are longer and / or shorter than a given size —which you provide.

`getSequence.py` Get a sequence from you FASTA or FASTQ file, given the name of the sequence. If you provide a multi-FASTA/Q file and the name of a sequence, this script will fetch the sequence for you.

`restrictSequenceList.py` This tool is somewhat similar to `getSequence.py`, but it is used to fetch several sequences at once. It uses a list of sequences and a list of sequence names (in a flat file, one name per line), and select those sequences such that the name is in the sequence name.

`GetFlanking.py` This tool prints the elements from the first set of genomic intervals which are closest to (in other words, are flanking) the elements from the second set. You can also play on different parameters (look at Figure 14):

- restrict the search to downstream or upstream elements, or print downstream and upstream elements,
- only consider colinear flanking elements,
- only consider anti-sense flanking elements,
- only consider elements which are close enough (using some given distance),
- only consider flanking elements which do not overlap with the reference element.

Notice that elements from the second sets will be printed at most once, whether they are the flanking element of several elements from the first or not.

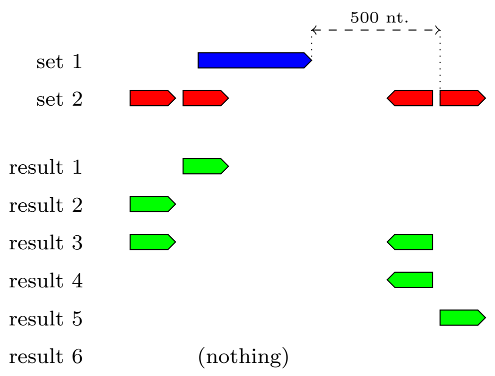


Figure 14: Results for different options for `GetFlanking`. Result 1: normal results. Result 2: do not allow overlapping. Result 3: same as before, and output flanking regions from both sides of the reference elements. Result 4: restrict to downstream regions. Result 5: same as before, and restrict to colinear flanking elements. Result 6: same as before, and only accept flanking elements within 100 nt.

4.4.2 Genomic coordinates

`restrictGenomicCoordinates.py` Reads a list of genomic coordinates and outputs those which on a given chromosome and / or between two given positions.

`selectByTag.py` Reads a list a list of transcripts and output all the transcripts with specific tag values. If you want to know more about tags, read Section 6.2.

The tools reads the input file (in GFF3 format) and more specifically the tag that you specified. You can mention a lower and a upper bound for its value, or a specific string, and the tool will print all the transcripts such that the tags are between the specified bounds or matches the string.

A tag has to be present for each transcript. If not, you can specify a default value which will be used if the tag is absent.

This tool can be used to select the clusters with a minimum number of elements (the tag `nbElements` counts the number of elements per clusters) or to select the reads which have mapped less than n times (the tag `nbOccurrences` counts the number of mappings per read).

4.5 Data modification

These tools do the “dirty job” that is sometimes useful to do: shrink or extend some genomic coordinates, get the first 20 nucleotides of your reads, etc.

4.5.1 Sequences

These tools are dedicated to data in FASTA or FASTQ files (usually, your reads).

`modifySequenceList.py` This tool reads a list of sequences (in multi-FASTA/Q format) that you provide and shrinks each sequence to the *n* first nucleotides or the *n* last nucleotides.

`trimSequences.py` This function removes the adaptor from the $5'$ or $3'$ end of your reads. It can even recognize the adaptators which are partially present. You can specify whether you are ready to accept indels or not.

4.5.2 Genomic coordinates

These tools are dedicated to data in a transcript file format such as GFF3 or BED (usually, your mapped reads).

`modifyGenomicCoordinates.py` This tool reads a list of transcripts and modifies each transcript by:

- shrinking it to the *n* first nucleotides or the *n* last nucleotides, or
- extending it to *n* nucleotides towards the 5' direction (upstream) or the 3' direction (downstream).

Note that the 5' or 3' direction depends on the orientation of the transcript (the 5' end of a transcript located on the minus strand is on the right hand of this transcript!).

The tool needs a transcript file, its format, and outputs a new transcript file.

`getExons.py` This tool split the transcripts such that an exon will be considered as a new transcript.

`getIntrons.py` This tools outputs all the introns of all the transcripts.

`changeTagName.py` It changes the name of a tag in a transcript list (see Section 6.2 to know more about tags). This may be useful to change the name of a tag which have been automatically addressed, like `nbElements` while clustering, to a more precise name (such as `nbReadsInRoot` for instance).

4.6 Data visualization

This set of tools do not modify your data, but simply outputs graphs.

4.6.1 Sequences

`getReadDistribution.py` This function analyzes your reads, count the number of times each read is sequenced and plots this distribution. A point (x, y) means that y different sequences have been sequenced x times. Put in other words, that you will find the word ACGUACGUACGU x times in you FASTA file, and $y - 1$ other different words like this.

You can also select the n most sequenced reads (or the $x\%$ highest).

`getLetterDistribution.py` Gets the nucleotide distribution of the input sequence list. It outputs two files. The first file shows the nucleotide distribution of the data (see Figure 15(a)). More precisely, a point (x, y) on the curve A shows that y sequences have $x\%$ of A.

The second plot shows the average nucleotide distribution for each position of the read (see Figure 15(b)). You can use it to detect a bias in the first nucleotides, for instance. A point (x, y) on the curve A shows that at the position x, there are $y\%$ of A. A point (x, y) on the curve # tells you that $y\%$ of the sequences contain not less than x nucleotides. By definition, this latter line is a decreasing function. It usually explains why the tail of the other curves are sometimes erratic: there are few sequences.

4.6.2 Genomic coordinates

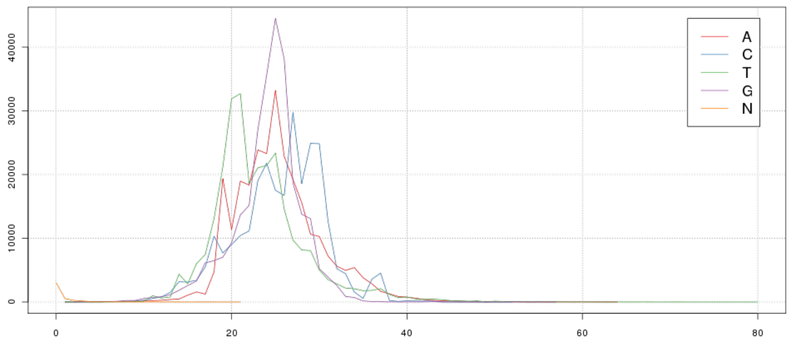
`getDistance.py` Give the distances between every data from the first input set and the data from the second input set. It outputs the size distribution (see Figure 16). Each point (x, y) tells you that there exists y pairs of elements which are separated by x nucleotides.

The general algorithm is the following. For each element of the first input set, it finds the closest element of the second set and computes the distance between the two elements. The distance is zero if the two elements overlap. This distance may not exist if the element of the first input set is alone on its chromosome (or contig).

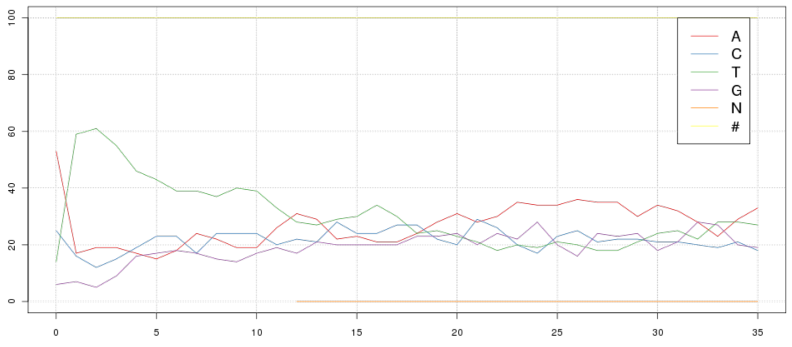
Actually, considering an element from the first input set, the algorithm will look at the vicinity of this element (1kb by default). You can increase the size of the vicinity using the appropriate option.

As in `compareOverlapping.py`, you can shrink or extend your sets of genomic coordinates, so that you can get the distance between starts of reads and starts or genes, for instance. You can also compute the distance from elements which are on the same strand only (which is not the case by default) or on the opposite strand only.

You have several options for the output plot. You can first choose the region on the x-axis you want to plot. You can also display histograms instead of line plot. In this case, the data are summed into buckets, whose sizes are given as an option. For instance, a bucket of size s at the point (x, y) means that there are y pairs of elements which are separated by x to $x + s$ nucleotides.



(a) Nucleotide profile for the whole distribution. The x-axis is the proportion each nucleotide and the the y-axis is the number of reads with the corresponding distribution. There is, for instance, more that 40,000 reads with around 25% of `G`.



(b) Nucleotide by nucleotide. The x-axis is the index of the nucleotides (start at 1), y-axis is the percentage of nucleotides. The # curve give the percentage of reads with at least the given size. In this example, all reads have exactly 36 nucleotides.

Figure 15: Nucleotide distributions using `getLetterDistribution.py`.

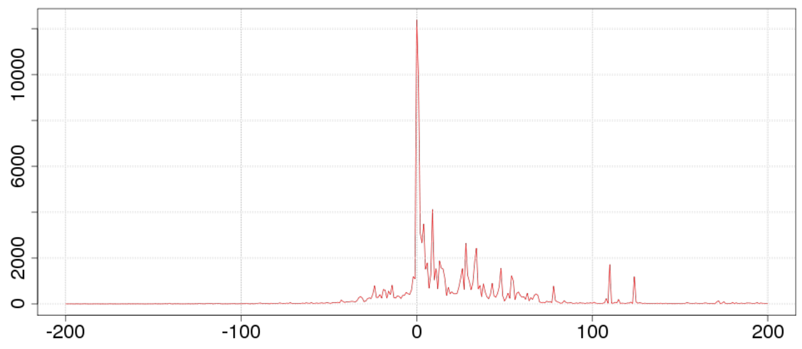


Figure 16: Distance between some reads and RefSeq genes in *Drosophila melanogaster*, using `getDistance.py`.

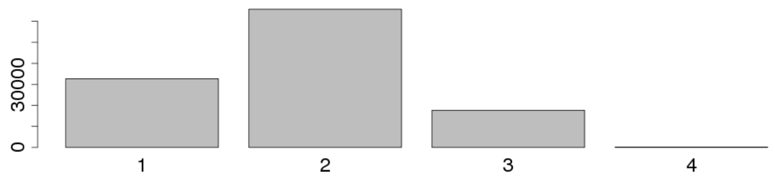


Figure 17: Number of exons per transcript of some 454 mapped reads, using `getNb.py`.

You can also save the distances into a GFF3 output file. In this case, the output file will the same as the first input file, except that some tags will be added (see Section 6.2 to know more about tags): the distance from the closest element in the second set, and the name of this latter element (using tags `distance` and `closestElement` respectively). If an element from the first input has no element from the second set in its vicinity, the tags will be set to `None`.

`getNb.py` Get the number times the reads have mapped, the number of exons for each mapping, or the number of elements in the clusters (see Figure 17). By default, the output is a line plot, but you can choose to have a bar plot instead.

`getDistribution.py` Print a density profile of the data for each chromosome, see Figure 18. You have to provide the reference genome, to know the sizes of the chromosomes. You can also provide the number of points (called *bins*) you want per chromosome.

By default, only one curve is plotted per chromosome, but you can plot one

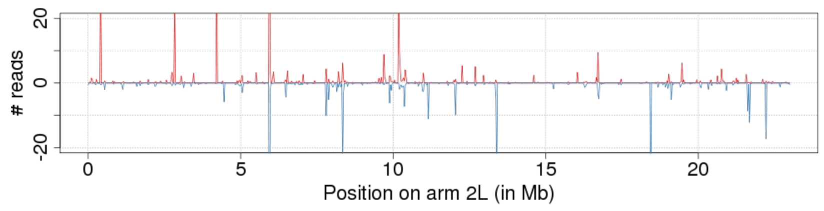


Figure 18: Density profile of some reads, using `getDistribution.py`. x-axis is the genomic coordinates of the chromosome 2L of *Drosophila melanogaster*, release 5. y-axis is the number of reads.

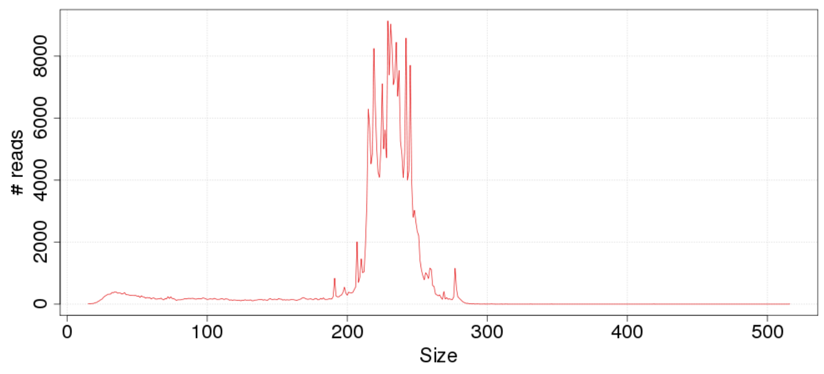


Figure 19: Size distribution of some 454 reads, using `getSizes.py`. x-axis is the size, y-axis is the number of reads.

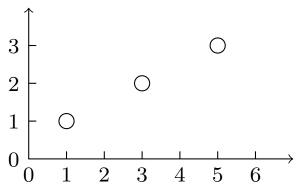
curve per strand and per chromosome (the minus strand will be plotted with non-positive values on the y-axis). Moreover, the density is plotted (i.e. the ratio of the number of reads divided by the size of the bins), but you can also plot the raw number of reads. Actually, the two results are equal (modulo a constant multiplication factor) except for the last bin of the chromosome, which is usually smaller than the other bins.

If you want, you can also plot a specific region, by mentionning the chromosome, the start and the end positions of the region.

`getSizes.py` Get the read size distribution, see Figure 19. A point (x, y) means that y reads have a size of x nucleotides.

When your mapping include exon/intron structures, you can decide to count the size of the introns, the sizes of the exons or the size of the first exons.

```
chr1 S-MART transcript 100 200 . + . ID=region1;nbReadsRoot=1;nbReadsLeaf=1
chr1 S-MART transcript 200 300 . + . ID=region2;nbReadsRoot=5;nbReadsLeaf=3
chr1 S-MART transcript 300 400 . + . ID=region3;nbReadsRoot=3;nbReadsLeaf=2
```
(a) A short GFF file, with the results of sliding windows counting the number of reads in two conditions (root and leaf).



(b) The corresponding plot.

Figure 20: A plot using `plotTranscriptList.py`.

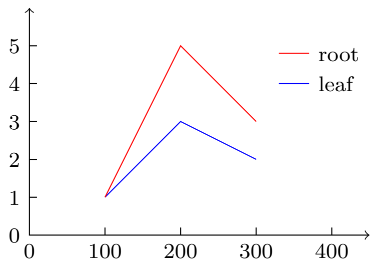


Figure 21: A plot using the data in 20(a) and `plotDistribution.py`.

`plotTranscriptList.py` Plot the data attached as tags in a transcript list. See Section 6.2 if you want to know more about tags. This can be used for displaying the comparison of different sets of sliding windows (see Figure 20).

The tool reads the tags of a transcript file (actually, a GFF3 file). It considers more specifically the tag names that you specify as parameter. If you use only one tag name, you can display a line plot. In this case, you have to specify a bucket size s (which is by defaut 1) and a point (x, y) tells you that there are y transcripts with tag values x to $x + s$.

You can display could plots if you use two tag names. Each point represents the values of the two tags of a transcript. If you use three variables, the third variable will be the color of the point. You can also use a log scale and name the axes of the plot.

Each transcript must contain the tags which are specified. If not, you should provide a default value, which is used when the tag is not present.

If you use a cloud plot, you can compute the Spearman's rho to quantify a correlation between your two tag values.

`plotDistribution.py` Plot the data attached as tags in a transcript list along the genome. See Section 6.2 if you want to know more about tags. This is espacially interesting for displaying the regions where different sets of sliding windows differ (see Figure 21).

`plotCoverage.py` Plot the coverage of the first set of genomic coordinates with respect to the second set of genomic coordinates (see figure 22). For each element of the second set (we will suppose that they are annotated genes), it computes the number of elements of the first set (reads, for instance) which overlap it.

The tool produces two plots per gene. The first plot gives the coverage: a point (x, y) means that y reads cover the xth nucleotide of the gene (see figure 22(a)). The second figure displays the (possibly spliced) gene in black, and the overlapping reads (blue is colinear, red is anti-sense, see figure 22(b)).

4.6.3 WIG tools

WIG Data WIG is the format used to store the kind of information which is attached to —nearly— every nucleotide of a genome. Examples include nucleosome occupancy using ChIP-Seq data. In this case, a high score means that there is a high probability that a nucleosome is close to this nucleotide.

`getWigData.py` Reads a transcript list, computes the average value of some WIG data for each transcript and adds a tag corresponding to this average value to the transcript.

`getWigData.py` finds all the data which correspond to the genomic coordinates of a transcript, average these data and store the result into a tag. Then, the transcripts are written in an output file, together with the tag (see Figure 24).

You can then plot your data using `plotTranscriptList.py`.

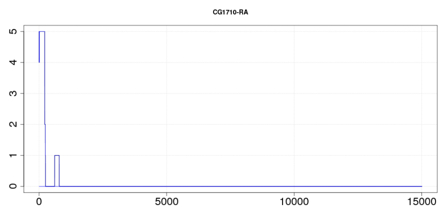
`getWigData.py` uses indices for each chromosome to fetch the data faster. This indices can be large, so if your are short of memory, you can remove them; S–MART will build them on the fly.

`getWigDistance.py` Plots the average data contained in a set of WIG files around the first nucleotides of a list of transcripts (see Figure 25).

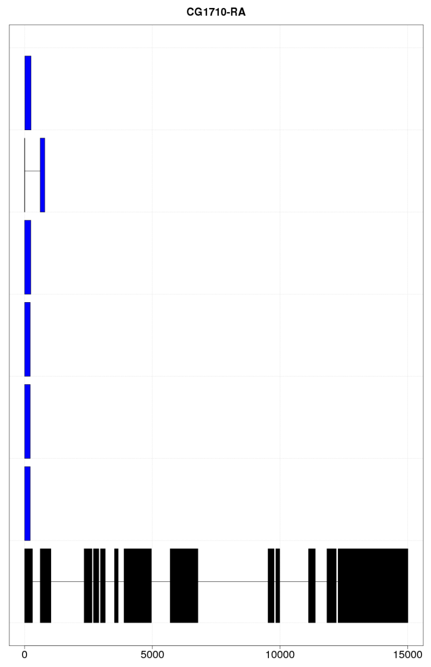
The tool needs an transcript list, some WIG files, and a distance. For each transcript, it collects all the values around its first nucleotide, the radius being given by the distance. Then, it computes the average value for each position. A point (x, y) means that the average value in the WIG file for a nucleotide distant by x nucleotides from the first nucleotide of an input transcript is y.

You can possibly use a log scale for the y-axis.

`getWigProfile.py` Computes the average distribution of the WIG data along the transcripts given in input, and possibly before and after the transcripts (see Figure 26).



(a) The coverage output.



(b) The overlap output.

Figure 22: A plot using `plotCoverage.py`. The first plot displays the number of reads which overlap a gene. The second plot displays the reads (here, spliced reads) which overlap the gene. Blue is + strand, black is the reference gene.

```
variableStep chrom=chr1
100 0.1
101 0.1
101 0.4
```

Figure 23: A short WIG file.

```
chr1 S-MART transcript 100 102 .  + .  Name=test
```

(a) A GFF line.

```
chr1 S-MART transcript 100 102 .  + .  Name=test;value=0.4
```

(b) The output of `getWigData.py`.

Figure 24: Including WIG data with `getWigData.py`. The first figure shows a toy GFF3. Using the WIG file of Figure 23, the tool computes the average values for the transcript and creates a new tag with this result.

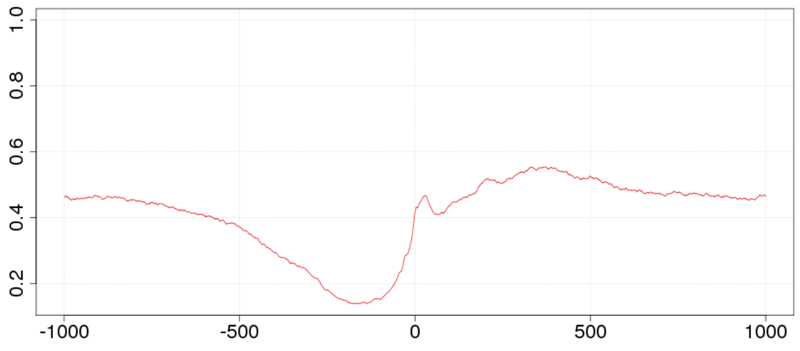


Figure 25: A plot using `getWigDistance.py`: the conservation given by PhastCons around the TSSs of very expressed regions (TSS is at $x = 0$).

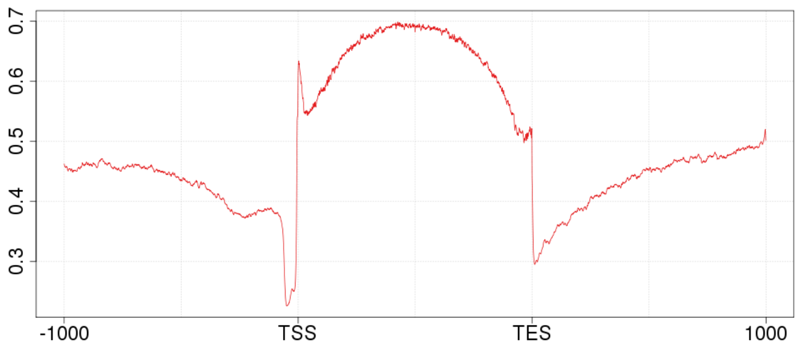


Figure 26: A plot using `getWigProfile.py`: the conservation given by PhastCons of the FlyBase genes.

The main inputs of the functions are a file containing a list of transcripts (or any sets of genomic interval) and a directory containing a set of WIG files (one file per chromosome, or one file per chromosome and per strand). The function then computes the WIG profile of each transcript. The user can also define a region around the transcripts that should also be plotted (in this case, the profile will include the WIG values which overlap with the transcript as well as the 5' and 3' regions). Since the transcript do not necessarily have the same sizes, all profiles will be extended or shrinked to fit in a size which is given by the user. If the resulting profile is a bit bumpy, the user can also smoothen the curve by using a linear smoothing function (the size of the smoothing window is given by the user). Finally, the user may want to plot the WIG data for the opposite strand too (if the strand specific WUG data are available).

4.7 Conversion tools

`convertTranscriptFile.py` Converts some data from one format to another format. You should provide the input and the output file names, together with their formats.

The input format has been previously listed. They can be transcript format (BED, GFF) or mapping format (SAM, Blast `-m 8`). The output formats are:

- Excel format
- BED format,
- GFF2 format,
- GFF3 format,
- GTF format,

- SAM format,
- WIG format,
- the format used by GBrowse,
- the format used by UCSC genome browser,

Note that the Excel format mentionned here is not precisely the XLS nor the XLSX format. It is the CSV format, which is simple enough to be understood by Excel (or OpenOffice). If you use OpenOffice, it will ask you some question about the separator of the fields and the lines. Simply say "OK" to its question.

`coordinatesToSequence.py` Provide a list of genomic coordinates, a reference genome, and it will output the sequence corresponding to the genomic coordinates, in a multi-FASTA file.

`qualToFastQ.py` Converts a QUAL file (together with the corresponding FASTA file) to a FASTQ file, that S–MART can use.

For some reasons (the first one being I am lazy), I have not implemented any QUAL parser. So, if you want to use them, you should convert them to FASTQ files, using this tool. It supposes that the base names of the files are the same in your QUAL file and your FASTA file (for instance `foo.qual` and `foo.fasta`).

4.8 Other tasks

`cleanGff.py` This tool tries to "clean" a GFF by removing all the unrelevant lines and possibly altering others. This is specially useful when you download a genome wide annotation file from NCBI, for instance. Look at Appendix D to see the kind of problems it can solve.

`getRandomRegions.py` Generates a set of random regions in a reference genome. Useful to compare your data with random data.

You have to provide a reference genome (otherwise, the tool does not know the number of chromosomes nor their sizes). You can choose the sizes of the regions (which will all be the same) and the number of them. By default, the elements are generated randomly in the genome.

Alternatively, if you provide a set of genomic coordinates, the function will move them to random regions in the genome.

You can try to somehow reproduce the "clustered" nature of the 5' capped reads (which gather around the TSS) by using the "cluster" mode: the elements will span (using a Gaussian law) around random loci in the genome. The number of elements per cluster follow an exponential law.

`removeAllTmpTables.py` The toolbox may generate some SQLite that you may want to drop. Use this script to do so.

Notice that if you interrupt the execution of a program while it is running, the tables and the temporary files which are currently used cannot be removed and you have to do it by yourself using this tool.

5 Possible pipe-lines

I will describe here a couple of "pipe-lines" that I have found useful. Actually they are inspired both by my own experience, and by papers I read.

I will first describe the general steps to perform the pipe-lines, then provide the commands to do so. I cannot precisely describe the corresponding scheme using the GUI (describing the set of clicks you have to perform is out my reach) but I hope it wil not be too complicated.

5.1 Use two mappers

Up to now, there is no clear consensus about which mapper gives best results (there is probably none, though). So I used two of them. In this example, I mapped with Blat and Exonerate. What you have to do is:

- Map your data with Blat (not described here).
- Map the same data with Exonerate.
- Check the mapping of Blat (`Mapper Analyzer`).
- Check the mapping of Exonerate on the sequences which have not been mapped by Blat (`Mapper Analyzer`).

```
python mapperAnalyzer.py -i blatOutput.psl -f psl
  -q reference.fasta -o mappingWithBlat
python mapperAnalyzer.py -i exonerateOutput.exo -f exo
  -q reference.fasta -a mappingWithBlat -r
  -o mappingWithBlatAndExo
```

5.2 Find piRNA clusters

Some papers show an interesting way to find potential clusters of piRNAs. They:

- sequence the transcriptome with RNA-Seq,
- map the data,
- keep the data which sizes between 25 and 31 nucleotides (`Restrict From Size`),
- exclude everything which overlaps with RefSeq data (`Compare Overlappin`),

- merge the mappings into clusters (20 kb) (`Clusterize`),
- keep those clusters which have at least ten elements (`Select By Tag`),
- convert the output to Excel format (`Convert Transcript File`).

Here is how it can been done.

```
python restrictFromSize.py -i mappedData.gff3 -f gff
  -m 25 -M 31 -o goodSize
python compareOverlapping.py -i goodSize.gff3 -f gff
  -j refSeqGenes.bed -g bed -c -x -o noGene
python clusterize.py -i noGene.gff3 -f gff
  -d 20000 -o clustered
python selectByTag.py -i clustered.gff3 -f gff
  -g nbElements -m 10 -o bigClusters
python convertTranscriptFile.py -i bigClusters.gff3 -f gff
  -o bigClusters -g excel
```

5.3 Get the letter distribution of the beginning of the data

It might be interesting to see if you have a bias in the distribution of the nucleotides at the beginning of the data that you have selected. For example, piRNA usually start with `U`, RefSeq data, with `ATT`, and so on. To visualize it, you can:

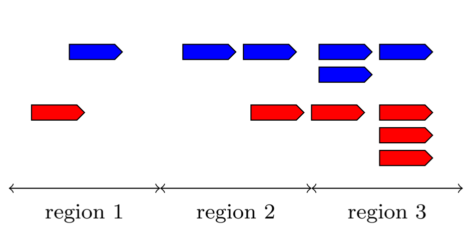
- select the first 30 nucleotides of your data (`Modify Genomic Coordinates`),
- get back the sequences from your genomic coordinates (`Coordinates To Sequence`),
- get the nucleotide distribution (`Get Letter Distribution`).

It can be done this way.

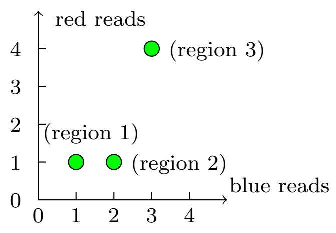
```
python modifyGenomicCoordinates.py -i data.gff3 -f gff -s 30
  -o cutData
python coordinatesToSequence.py -i cutData.gff3 -f gff
  -s reference.fasta -o sequences.fasta
python getLetterDistribution.py -i sequences.fasta
  -o nucleotideDistribution
```

5.4 Compare two sets of reads with sliding windows

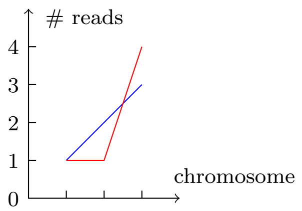
Suppose you have two sets of reads (for instance, from two conditions) and you want to have a broad picture of a possible correlation between the two sets. What you can do is to use a sliding window to cluster the two sets of reads (see



(a) Example of two sets of reads (one blue, one red), clustered by sliding windows into three regions.



(b) The corresponding plot. For each dot, the corresponding region is given in parenthesis.



(c) The compared distribution on the genome.

Figure 27: Comparing two sets of reads with sliding windows.

Figure 27(a)). Then, you can plot the results, where each point (x, y) comes from a sliding window which contains x reads from the first set and y reads from the second set (see Figure 27(b)).

Here are the steps you have to perform to do so:

- Use a sliding window to cluster each set (here the size of the window is 1kb, and the overlap is 500 nt.). You may or you may not consider that two reads on opposite strands can be in the same window. Use option `-2` to consider the strands independantly (`Clusterize By Sliding Windows`). You now have two GFF3 files, and the tag `set1Clustered` and `set2Clustered` counts the number of reads for each window in each file.
- Merge the two files (`Merge Sliding Windows Clusters`).
- Plot the distribution. Here, we will use log bases for x and y-axes (`-l` option). We also have to specify the default values for missing data with the `-X` and `-Y` options. We can specify a label for the axes with the `-m` and `-n` options (`Plot Transcript List`).

Supposing that the reads are in the files `set1.gff3` and `set2.gff3`, you can use the following commands to perform the steps:

```
python clusterizeBySlidingWindows.py -i set1.gff3 -f gff
  -s 1000 -e 500 -o set1Clustered -2 -w set1Clustered
```

```
python clusterizeBySlidingWindows.py -i set2.gff3 -f gff
  -s 1000 -e 500 -o set2Clustered -2 -w set2Clustered
python mergeSlidingWindowsClusters.py -i set1Clustered.gff3
  -f gff -j set2Clustered.gff3 -g gff -o set12Clustered
python plotTranscriptList.py -i set12Clustered.gff3 -f gff -x set1
  -y set2 -X 0 -Y 0 -l xy -n set1 -m set2 -s points
  -o set12ClusteredPlot
```

And, yes, you get the Spearman rho!

Moreover, if you want to see the distribution of the two sets of reads on the chromosome (see Figure 27(c)), you can do it with:

```
python plotDistribution.py -i set12Clustered.gff3 -n set1,set2
  -c blue,red -r -o set12ClusteredGenome
```

5.5 Compare RNA-Seq with tiling arrays using sliding windows

Let us suppose that you have some tiling array you want to compare with your sequencing. Let us also suppose you have analyzed your array data and you now have some *p*-value, *t*-value or anything which is a proxy to gene intensity. You may want to use sliding windows to compare with your reads. Of course, for a fair comparison, you want to compare the regions where you have at least 1 chip (you may have not covered the whole genome).

The steps are the following:

- Cluster the array data into sliding windows, with a window size of 1kb, and an overlap of 500 nt., just to count the number of chips per window (`Clusterize By Sliding Windows`).
- Cluster the array data into sliding windows by using the average intensity value in the window (but we can use min, max or median value). Here we will suppose you have a GFF3 file with a `intensity` tag (`Clusterize By Sliding Windows`).
- Merge the two previous files (`Merge Sliding Windows Clusters`).
- Keep the regions where you have at least 1 chip (`Select By Tag`).
- Cluster the reads as done in section 5.4 (`Clusterize By Sliding Windows`).
- Merge the array file and the reads file (`Merge Sliding Windows Clusters`).
- Plot the distribution with log base on *y*-axis (for the reads, `Plot Transcript List`).

```
python clusterizeBySlidingWindows.py -i array.gff3 -f gff
  -s 1000 -e 500 -o arrayNb -2
python clusterizeBySlidingWindows.py -i array.gff3 -f gff
  -s 1000 -e 500 -o arrayIntensity -g intensity -r avg -2
python mergeSlidingWindowsClusters.py -i arrayNb.gff3
  -f gff -j arrayIntensity.gff3 -g gff -o arrayNbIntensity
python selectByTag.py -i arrayNbIntensity.gff3 -f gff
  -g nbElements -m 1 -o arrayNbIntensity1chip
python clusterizeBySlidingWindows.py -i reads.gff3 -f gff
  -s 1000 -e 500 -o readsClustered -2
python changeTagName.py -i readsClustered -f gff -t nbElements
  -n nbReads -o readsClusteredGoodTag
python mergeSlidingWindowsClusters.py
  -i arrayNbIntensity1chip.gff3 -f gff
  -j readsClusteredGoodTag.gff3 -g gff -o arrayReadsClustered
python plotTranscriptList.py -i arrayReadsClustered.gff3 -f gff
  -x avgIntensity -y nbReads -Y 0 -l y -n array -m reads
  -s points -o arrayReadsPlot
```

5.6 Compute differential expression

Suppose you have two sets of reads, from two different conditions and you want to compare them. The main difficulty here is to decide where you are going to compare. You could decide to slice the genome into sliding windows and perform the comparison on the sliding windows, but that is probably not the best thing to do (although you can do it with S–MART). What you should do is to find the clusters of reads and compare the clusters.

For instance, look at figure 28, which represents two sets of reads mapped on a genome. Obviously, we have two clusters, altough the red reads are not present in the second cluster (which is a particularly good case of differential expression). Basically, we have to find these two regions from the two sets of data. Notice that in the first region, there is a small gap which is not covered by any read. So, we have to accept some small gaps to merge the reads into clusters.

Then, it could be nice to keep the regions where there is a clear differential expression and see where they are on the genome.

From the previous reasoning, we can conclude that we have to follow the following steps:

- Cluster the reads from sample 1 with some gap allowed (here, 10 nt.). Only cluster the reads which are on the same strand (`-c` option in `Clusterize`).
- Do the same for the sample 2.
- Merge the two sets of clusters. Again, the clusters on different strands should not be merged (`Merge Transcript Lists`).

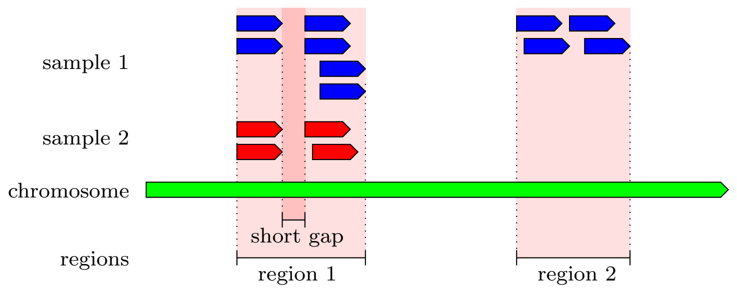


Figure 28: Finding the regions where to compare the two clusters.

- Compute the differential expression from sample 1 and sample 2 using the regions previously found. We now have a set of regions with a *p*-value associated to differential expression (`Get Differential Expression`).
- Select the regions with a very low *p*-value (here, 10^{-100}) (`Select By Tag`).
- Plot these regions onto the genome (you will need the reference genome `genome.fasta` for that, `Get Distribution`).

Expressed in the S–MART language, this becomes:

```
python clusterize.py -i sample1.gff3 -f gff -c -d 10
  -o sample1clustered
python clusterize.py -i sample2.gff3 -f gff -c -d 10
  -o sample2clustered
python mergeTranscriptLists.py -i sample1clustered.gff3
  -f gff -j sample2clustered -k -c -o sample12clustered
python getDifferentialExpression.py -i sample1.gff3 -f gff
  -j sample2.gff3 -g gff -k sample12clustered -l gff
  -o sample12differential
python selectByTag.py -i sample12differential -f gff
  -g pValue -M 10e-100 -o sample12differentialLowPValue
python getDistribution.py -i sample12differentialLowPValue
  -f gff -r genome.fasta -2 -o sample12differentialLowPValuePlot
```

5.7 Plot the H3K36me3 histone modification from ChIP–Seq data around gene TSS

Suppose that you have some ChIP–Seq reads (for instance, of H3K36me3 histone modification) and you would like to get the profile around the TSS of your genes with respect to this data: you suspect that most of the ChIP-Seq data accumulate at the beginning of the genes, but you would like to see it. We will suppose that you have mapped your reads with any software, and you end with a file `h3k36me3.sam` containing the ChIP-Seq data. You also have your set of genes in `genes.gff3`.

To perform this task, you should follow the following step:

- We should first create a directory of the ChIP–Seq data `ChipSeq` (call it `Chip-Seq`).
- Then, transform the data from a mapping format (here, the SAM format) to a WIG format (`Convert Transcript File`).
- Reduce each transcript to its first nucleotide (`Modify Genomic Coordinates`).
- Finally, compute the profile (you would like to plot 1kb around each TSS) (`Get Wig Distance`).

Expressed in the S–MART language, this becomes:

```
python convertTranscriptFile.py -i h3k36me3.sam -f sam
  -o h3k36me3Wig -g wig -v 10
python modifyGenomicCoordinates.py -i genes.gff3 -f gff3 -s 1
  -o geneTss -v 10
python getWigDistance.py -i geneTss.gff3 -f gff3
  -w h3k36me3Wig.wig -d 1000 -s -o output -v 10
```

Notice that computing the average ChIP–Seq value for each transcript or the average ChIP–Seq distribution is even easier: do not reduce each transcript to its first nucleotide and use `getWigData.py` and `getWigProfile.py` instead!

6 More about S–MART

I will give you here some details about the internal representations of the data in S–MART. Just in case you would like to know how it works. . .

6.1 Data structures

S–MART mainly use a data structure that models a transcript. A transcript can be decomposed as a set of exons, which basically are genomic intervals. So, a transcript is a set of genomic intervals. So is a mapped read; short reads usually have only one exon, but longer ones often have several.

A cluster is modeled the same way. While clusterizing, S–MART merges the exons of the transcripts one by one (if they overlap), thus forming a new set of new exons.

```
chr1 S-MART transcript 100 400 . + . ID=read1;nbMismatches=2
chr1 S-MART exon 100 200 . + . ID=read1-1;Parent=read1
chr1 S-MART exon 300 400 . + . ID=read1-2;Parent=read1
```

Figure 29: A short GFF3 file, containing a transcript with two exons. The tags are in the last field of each line.

6.2 Tags

S–MART for each "transcript", S–MART attaches some information, called *tags*. The information might be the number of mismatches of a mapped read, or the number of elements in a cluster.

S–MART automatically load the tags from a GFF3 file (see Figure 29 for an example of a GFF3 file with tags). It updates the tags while mapping the reads and clusterizing them.

6.3 How SQLite is used?

S–MART stores all the data in a SQLite table. That is to say, S–MART (see Figure 30):

1. reads your data,
2. stores them in a database,
3. processes them (it basically uses an B-tree on nested bins to compute the overlaps faster) and
4. outputs them into an output file.

So, in some cases, you can skip the steps 1 and 2. Suppose that you use the tool `compareOverlapping.py` to keep only those reads which overlap with RefSeq data, then `clusterize.py` to merge them into clusters. In `compareOverlapping.py`, you can use the `-y` option to keep into the database the data. Then, if you mention that the input format of `clusterize.py` is `-f sql`, S–MART directly uses the data which is available in your SQLite database. So, you can skip the steps 1 and 2, and save some time! Be aware, however, that it takes some place on your hard disk.

6.4 Contribute to S–MART!

If you want to add your own tool to S–MART, please do not hesitate. You can develop it using the API I have made. Look at the format I have used to generate the help from my Python scripts (using `OptionParser`) in the `Python` directory and add your own Python file in the same directory. That is it! It will be automatically included in the GUI and you will be able to start your own tool from there.

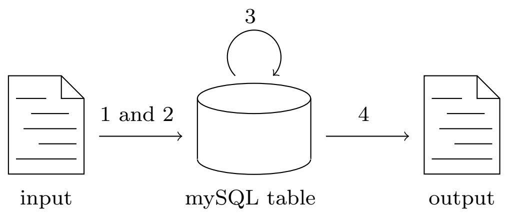


Figure 30: Use of SQLite tables in S–MART, in 4 steps.

7 Contact

For any comment, suggestion, remark, do not hesitate to contact me.

A Load data on your genome browser

UCSC genome browser Go to the UCSC genome browser. Select your species (if available). Click on the add custom track button. Upload your data in a "`.ucsc`" or "`.bed`" file.

GBrowse Go to your favorite GBrowse, for example the fly GBrowse. Go down, to the Add your own tracks section. Select your data in a "`.gb`" file after clicking on the Browse... button. Now watch amazed your reads.

B Get other data

UCSC As long as your organism is covered by UCSC (including Mammals, other Vertebrates, Insects, Nematodes and a few other species), this may be the right place to get your data. Go to their database to retrieve the annotation. Select the right organism, select the group, track and table where your data are (it may take some time to find the right combination), select BED as the output format, and mention an output file. This is it!

RefSeq NCBI provides a list of RefSeq sequences on their public FTP. You can browse it with your normal browser. Choose the right organism, then you should download the data chromosome by chromosome (which is pain, I reckon). Be sure that you get the GFF files. Afterwards, you should concatenate the files.

FlyBase If you work on an insect, you can go to the FlyBase Web site to download your annotation. Select your organism, then the release (latest is best), then the GFF folder. You can then download the whole annotation in a file whose name contains the word `all`. Be careful, you will download the whole annotation of your insect! It may not exactly be what you need!

Other data Many Web site are dedicated to a specific organism or some specific data (miRNAs, transcription factor binding sites, etc.). I cannot cover them all, but you can probably find there the data that you need. It is a matter of patience. Just make sure that the data is available in the usual formats (GFF or BED, for instance).

C Caveats

Never modify your data with Word (or a similar text processor)

I reckon Word is a very good tool, but is it not made to view and alter flat files. Word converts a flat file to its own format, which is very hard to read (change a `.doc` extension to a `.txt` extension and try to open the file... you might be surprised). Every file that we S–MART uses should be viewed with NotePad or the like. Except images, of course.

D Troubleshooting

My GFF file is not parsed! A problem with a GFF file may have different causes.

First, it seems that there is no gold standard for GFF files. Current files are usually produced in GFF3 format, which includes an `ID` and possibly a `Parent` fields. However, this convention is sometimes not followed. If they are not present, it is hard for S–MART to link a line which contains a transcript annotation to the lines which contains the annotations of its exons.

Second, please make sure that your files contains *only* the data you are interested in. For instance, when you download some genome wide annotation, you have information of the size of the chromosomes, annotation of the start and stop codon, of the 5' and 3' UTR. This is usually something you are not interested in and S–MART can be confused. Basically, you just want the annotation of the transcripts and their exons (see Figure 29 for an exemple of a very simple GFF3 file).

Third, different sources may name the chromosomes differently. Compare `chr_I` with `Chr1` (or even worse: `gi|157069709|gb|AABX02000103.1|`). S–MART cannot see they actually are the same chromosome (actually, UN-S–MART could have been a better name). So you have to change the names of the chromosomes accordingly.

In all cases, you can use the tool "Clean GFF" (see Section 4.8), which tries to produce a GFF3 file which can be understood by S–MART. But please check the output file to make sure the output is correct!

The GFF3 file which is produced by S–MART is larger than expected! Suppose that you use `compareOverlapping`, and it outputs:

```
...
output: 1000 sequences (3%)
```

. . .

So you expect to have 1000 lines in the output GFF3 file. If you actually count the them (you can use `wc -l` *`your_file.gff3`* on Linux to do so), you see that there are more lines (say 3000). Did S–MART lie? Is there a bug?

Actually, the reason comes from the GFF3 format itself. Suppose that you have a transcript which is composed of two exons:

- exon 1: from position 100 to 200
- exon 2: from position 300 to 400

This can formatted as:

```
chr1 S-MART gene 100 400 . + . ID=gene1
chr1 S-MART exon 100 200 . + . Parent=gene1
chr1 S-MART exon 300 400 . + . Parent=gene1
```

So you have 1 sequence on 3 lines!

As a general rule, if you want to count the number of sequences, you can count the number of lines which do not contain the key `Parent` in the last field (`grep -v Parent` *`your_file.gff3`* `-c` on Linux).

My Mac does not open the Excel file properly! On some Mac releases, the file which is produced for Excel (namely, a flat text CSV file) is not properly read by MacExcel. In case of problem, you may start Excel, then:

- go to `Data`,
- select `Get External Data`,
- select `Import Text File`,
- then choose the CSV file which has been generated by S–MART.

Index