IPSA: Integrative Pipeline for Splicing Analysis

Dmitri D. Pervouchine*

Centre for Genomic Regulation (CRG), Barcelona, Spain

July 15, 2015

Contents

*email: dp@crg.eu

1 Synopsis

This document contains the description of IPSA, an integrative splicing analysis pipeline. It quantifies splicing events by directly analyzing split alignments in BAM files. The advantage of IPSA is that it quantifies annotated splice junctions as well as novel ones and that there is no annotation-related interference between independent splicing events. An obvious disadvantage, however, is that only a part of the sequencing information is being used, one that comes from local alignments at splice junctions, while the reads that are fully contained in exons or in introns are being ignored.

The pipeline starts with a BAM file, which is processed by *sjcount* utility and converted to a tab-delimited output of SJ-read counts with offsets (see sjcount documentation for more detail). Additionally, *sjcount* produces counts of the continuous reads that overlap splice junctions (SS-reads). These outputs are aggregated, matched against genome and annotation, and passed to strand disambiguation. The resulting stranded counts are subject to irreproducibility assessment (npIDR) and subsequent filtering based on IDR, entropy, and staggered alignments. This workflow is unversal for many types of data, i.e., for stranded and unstranded data, for data with and without bioreplicates, etc.

2 Detailed description of the components

2.1 Data index file

Data index file is a two-column tab-delimited file, which is supplied as an input and whose first column contains the file name or a URL, and the second column contains a list of attributes of the form `attribute1="value1"; attribute2="value2";`.

2.2 Pipeline generator

The *make.pl* utility takes as an input the RNA dashboard index file and outputs to the standard output a *GNU* makefile that will be executed in order to to run the pipeline.

```
Perl/make.pl -dir <dirname> -param <params> -by <attribute> -margin
          <length> -annot <gtf> -genome <name> -merge <name>
Inputs: -annot ..., the annotation (gtf), obligatory
        -deltaSS ..., distance threshold for splice sites, default=10
        -dir ..., the output directory, obligatory
```

```
        -entropy ..., entropy lower threshold, default=1.5
        -genome ..., the genome (without .dbx or .idx), obligatory
        -group ..., the grouping field for IDR, default=labExpId
        -idr ..., IDR upper threshold, default=0.1
        -in ..., the index file, obligatory
        -margin ..., margin for aggregate, default=5
        -merge ..., the name of the output to merge if blocks are missing
        -mincount ..., min number of counts for the denominator, default=10
        -param ..., parameters passed to sjcount
        -repository ..., the repository subdirectory for bam files
        -smpid ..., sample id field, default=labExpId
        -status ..., annotation status lower threshold, default=0
Output: STDOUT: a GNU makefile
        make -f file.mk all
```

For example, the command

```
Perl/make.pl -repository input/ -dir output/ -group idrGroup -param '-read1 0'
        -annot hg19v18.gff -genome homSap19 -merge pooled -in example.dat
         > example.mk
```

creates a file called 'example.mk' based on the input from 'example.dat', where all the intermediate steps will be stored in 'output/' with the annotation file hg19v18.gtf in the current directory, and with homSap19.idx and homSap19.dbx both being readable genome files.

2.3 Pre-processing

2.3.1 Annotation pre-processing

Genome annotation files often come in GTF format and contain many feature types simultaneously. Often these are large files and it takes a lot of time just to read them. Sometimes they may be incomplete (e.g. contain exons, but not introns) and not fully correspond to GFF3 standarts.

The *transcript_elements.pl* utility reads a gtf, extracts only exons, and outputs for each exon the list of transcripts to which this exon belongs. It outputs also a gtf which contains (1) exons and (2) introns inferred from the input exon information, but the output file is shorter. For example, in the input gtf were

... ...

```
chr2L FlyBase exon 100 200 . + . gene_id "8"; transcript_id "1";
chr2L FlyBase exon 300 400 . + . gene_id "8"; transcript_id "1";
chr2L FlyBase exon 500 600 . + . gene_id "8"; transcript_id "1";
chr2L FlyBase exon 100 200 . + . gene_id "8"; transcript_id "2";
chr2L FlyBase exon 500 600 . + . gene_id "8"; transcript_id "2";
...   ...     ...  ... ... . . . ...
```

then the processed form would be

```
chr2L SJPIPE  exon    100   200   .    +    transcript_id "1,2";
chr2L SJPIPE  intron  200   300   .    +    transcript_id "1";
chr2L SJPIPE  intron  200   500   .    +    transcript_id "2";
chr2L SJPIPE  exon    300   400   .    +    transcript_id "1";
chr2L SJPIPE  intron  400   500   .    +    transcript_id "1";
chr2L SJPIPE  exon    500   600   .    +    transcript_id "1,2";
```

Formally speaking, the script groups transcript id for each exon and applies the function *join(",",uniq(sort(...)))* to the sorted array of transcript ids. Similarly, the script can group any other feature and apply any other function to the array, as well as it can put any value for the source field, e.g.

```
Perl/transcript_elements.pl -features transcript_id,source,position:avg
```

will return uniq(transcript_id), uniq(source), and avg(position) in the column 9 of GTF. Note that transcripts and introns are not shown in the input gtf.

NB In what follows, the processed annotation has to be sorted. Therefore it is necessary to pipe it with the sort step as follows.

```
Perl/transcript_elements.pl - < input.gtf | sort -k1,1 -k4,5n > output.gff
```

2.3.2 Genome pre-processing

The following two utilities, *transf* and *getsegm*, which belong to the *maptools* package, are used to pre-process genomes in a more compact and readable form. *maptools* can be obtained from github. The scripts in maptools/bin shall be made accessible by declaring the path to that directory.

The use of *transf* utility is as follows

```
transf -dir genome_directory/any.fa -dbx output.dbx -idx output.idx
```

It takes all the files in *genome_directory/* and creates two output files, *output.dbx* and *output.idx*; the former storing the data and the latter storing the index table tot hat data. The format is similar to 2bit.

The *getsegm* doesn't have to run on its own. Instead, it is used in *annotate.pl* to get genomic nucleotides.

2.4 Main functions

2.4.1 Counting splice junctions and reads that overlap splice sites

The *sjcount* utility takes a BAM file and counts the number of split reads supporting splice junctions (SJ) and continuous reads that overlap splice sites (SS). Splice sites are defined by the splice junctions that are present in the alignments. Version v3.1 also counts the abundance of split reads that span multiple junctions.

The utility returns the number of counts for each combination of chromosome, begin, end, strand, and offset, where offset is defined to be the position (within the short read sequence) of the last nucleotide preceding the splice junction. For the exact definitions of SJ, SS, offset, and examples see the help page of *sjcount* at github.

```
Usage: ./sjcount-3.1/sjcount -bam bam_file [-ssj junctions_output]
    [-ssc boundaries_output] [-log log_file] [-read1 0|1] [-read2 0|1]
    [-nbins number_of_bins] [-lim number_of_lines] [-quiet]
sjcount v3.1
Input:  a sorted BAM file with a header
Options:
        -read1 0/1, reverse complement read1 no/yes (default=1)
        -read2 0/1, reverse complement read2 no/yes (default=0)
        -nbins number of offset bins, (default=1)
        -maxnh, the max value of the NH tag, (default=none)
        -lim nreads stop after nreads, (default=no limit)
        -unstranded, force strand to be '.'
        -continuous, no mismatches when overlapping splice boundaries
        -quiet, suppress verbose output

Output: -ssj: Splice Junction counts, tab-delimited  (default=stdout)
        Columns are: chr_begin_end_strand, n_splits, offset, count
        -ssc: Splice boundary counts, tab-delimited  (default=none)
        Columns are: chr_position_strand, 0, offset, count
```

NB: the coordinates are 1-based

2.4.2 Aggregating SJ counts over offsets

The *aggregate.pl* utility takes the output of *sjcount* on STDIN and performs aggregation by the 3rd column (offset) using three different aggregation functions (see examples below). It outputs a TSV file with three extra columns: total count, staggered read count, and entropy. The output is sent to *stdout*.

```
Perl/aggregate.pl
Input:  TSV (ssj or ssc) on STDIN
Output: TSV on STDOUT
Options:-logfile ..., name of the log file
        -margin ..., the margin for offset, default=0
        -maxintron ..., max intron length; only for 1-splits, default=0
        -minintron ..., min intron length; only for 1-splits, default=4
        -prefix ..., prefix in the chromosome name
        -readLength ..., the read length, default=0
Columns in the output are: chr, begin, end, strand,
        count, staggered, entropy
```

It is possible to exclude short reads with small overhangs on either side by using -margin and -readLength parameters. This is particularly important when such a margin was imposed during the mapping step, but in order to be comparable when counting reads that overlap splice sites one should use the same restriction. Also, it is possible to exclude SJs that are too long or too short (-minintron/-maxintron).

The aggregation functions are applied to the sample $\{x_k\}$ of counts for each combination of chromosome, begin, end, and strand vs. the offset value k. The aggregation function therefore has the general form $f(x_1, \ldots, x_n)$.

When $f(x_1, \ldots, x_n) = x_1 + \cdots + x_n$, the result coincides with the collapsed (total) number of counts, i.e., as if offsets were ignored. For for $f(x_1, \ldots, x_n) = \theta(x_1) + \cdots + \theta(x_n)$, where $\theta(x) = 1$ for $x > 0$ and $\theta(x) = 0$ for $x \leq 0$, the result is the number of *staggered* read counts. The function

$$f(x_1, \ldots, x_n) = \log_2(\sum_{i=1}^{n} x_i) - \frac{\sum\limits_{i=1}^{n} x_i \log_2(x_i)}{\sum\limits_{i=1}^{n} x_i}$$

gives the entropy of the distribution, which can be used later to filter out non-uniform distiburtion of read counts. For example, if the input were

```
chr1_50_90_+    1    20       5
```

```
chr1_100_200_-  1    10      25
chr1_100_200_-  1    11      12
chr1_100_200_-  1    15      4
chr1_100_200_+  1    10      1
chr1_100_300_+  1    11      12
```

the output would have been

```
...             ...   ...     ...     ...
chr1_100_200_-  1     41      3       1.28
...             ...   ...     ...     ...
```

where 1.28 is the entropy of the distribution.

2.4.3 Checking the annotation status of a SJ and retrieving splice site nucleotides

The *annotate.pl* takes an aggregated TSV file (the output of *aggregate.pl*), the genomic annotation, and the genome, and outputs a TSV with two more columns: (8) the annotation status and (9) splice sites. The output is sent to *stdout*.

```
Perl/annotate.pl
Input:  -annot ..., the annotation (gtf), obligatory
        -dbx ..., the genome (dbx), obligatory
        -deltaSS ..., distance threshold for splice sites, default=0
        -idx ..., the genome (idx), obligatory
        -in ..., the input tsv file, obligatory
        -logfile ..., name of the log file
Outout: TSV on STDOUT
```

The annotation file is a simplified, processed form of the standard annotation gtf. It can be obtained by the *transcript_elements.pl* utility (see section 2.3.1). The genome consists of two compressed files, *.dbx and *.idx, which can be obtained from the genomic fasta sequence by using *transf* utility of the *maptools* package.

In previous versions the annotation status was defined numerically as follows:

(0) None of the splice sites of the given SJ is annotated;

(1) One of the splice sites of the given SJ is annotated, and the other is not;

(2) Both splice sites of the given SJ are annotated but the intron between them is not;

(3) Both splice sites of the given SJ are annotated, and so is the intron between them.

In the current version the annotation status is defined for each splice site as

(0) not within `deltaSS` nucleotides from an annotated SS;

(1) within `deltaSS` nucleotides from an annotated SS;

(2) annotated.

The status of a splice junction is defined according to the table (rows – donor site; columns – acceptor site)

	0	1	2
	not annot	within deltaSS	annotated
0	0	0.5	1
1	0.5	0.75	1.5
2	1	1.5	2 or 3

The splice site nucleotides are the four intronic nucleotides, two flanking ones from each end such as GTAG or ATAC. Since for this field and for the annotation status strand has to be defined, two lines are produced in the case of unstranded data (one for each strand). For instance, if the input were

```
...     ...   ...   ...   ...   ...
chr1_100_200_+     41    3     1.28
chr1_100_200_-     41    3     1.28
...     ...   ...   ...   ...   ...
```

and there were, indeed, an annotated junction at (chr1, 100, 200, –) with GTAG, then the output would have been

```
...     ...   ...   ...   ...   ...    ...   ...
chr1_100_200_+     41    3     1.28   0     CTAC
chr1_100_200_-     41    3     1.28   3     GTAG
...     ...   ...   ...   ...   ...    ...   ...
```

Note that sequence retriever uses the *`getsegm`* program of the *`maptools`* package, so maptools has to be installed and path has to be added.

2.4.4 Strand choice

At this step, a unique value of strand is chosen for each SJ. This is done by *choose_strand.pl* utility.

```
Perl/choose_strand.pl
Input:  TSV file on STDIN
Output: TSV file on STDOUT
Options:
        -annot ..., annotation column, default=5
        -logfile ..., name of the log file
        -sites ..., splice site column, default=6
```

For each combination of chromosome, begin, and end, the strand with greater annotation status (see section 2.4.3) is chosen. In case of a tie (usually 0 on both strands), the strand is chosen based on the "largest" splice site nucleotides in terms of lexicographic order (TTTT>GTAG> ...). There will be an option to choose a custom order of trustable splice site sequences (e.g., GTAG>ATAC>others).

For instance, if the input were

```
...     ...   ...   ...   ...   ...    ...   ...
chr1_100_200_+     41    3     1.28   0     CTAC
chr1_100_200_-     41    3     1.28   3     GTAG
chr1_150_200_+     21    2     1.01   0     GTAG
chr1_150_200_-     21    2     1.01   0     CTAC
...     ...   ...   ...   ...   ...    ...   ...
```

the output would have been

```
...     ...   ...   ...   ...   ...    ...   ...
chr1_100_200_-     41    3     1.28   3     GTAG
chr1_150_200_+     21    2     1.01   0     GTAG
...     ...   ...   ...   ...   ...    ...   ...
```

2.4.5 Constraining splice site counts

Since now a unique value of strand is chosen for each SJ, the counts of reads overlapping splice sites have to be constrained to a smaller set of splice sites. This is done by *constrain_ssc.pl* utility.

```
Perl/constrain_ssc.pl
```

```
Input:  TSV ssc (reads overlapping splice sites) on STDIN
        -ssj = TSV file to constrain to, obligatory
Output: TSV on STDOUT
```

Here -ssj is the splice junction file after strand choice was made, -ssc is the output of *sjcount*. The output of *constrain_ssc.pl*is sent to *stdout*. If the ssc input is unstranded, then the strand of a splice site is taken from ssj, where the strand is already defined. In some cases it will lead to two lines being produced in the case of unstranded data (one for each strand). For example, if the ssj input were

```
...    ...   ...   ...   ...    ...   ...   ...   ...   ...   ...
chr1   100   200   .     536    -     41    3     1.28  3     GTAG
chr1   150   200   .     439    +     21    2     1.01  0     GTAG
...    ...   ...   ...   ...    ...   ...   ...   ...   ...   ...
```

then (chr1, 200, +) and (chr1, 200, –) are both valid splice sites and the corresponding ssc counts will be reported for each of the two strands.

2.5 Ascertainment of reproducibility (IDR)

In this step the number of counts from (generally, as many as possible) bioreplicates are assessed for irreproducibility. This step is done by *idr4sj.r*

```
Rscript R/idr4sj.r inp1.tsv [inp2.tsv] ... [inpN.tsv] output.tsv
```

where inp1,2,...N the bioreplicates and output is the file is the last in the command line. The output contains one extra column (10) equal to IDR score. Columns 7, 8, and 9 are summed (not averaged!) between bioreplicates. In case if only one input file, the IDR score is set to 0. Currently, in case of more than two bioreplicates, only the first two files will be considered (others ignored).

2.5.1 Filtering

There is no specific routine for filtering because it can be done by *awk* by requiring column 7 (entropy) to be greater than threshold (usually, 3 bits) and the column 10 (IDR) be not greater than 0.1.

2.5.2 Calculation of splicing indices

As soon as the SJ (and splice site) counts were assessed for reproducibility and filtered, the next step is to compute the inclusion and processing rates by

zeta.pl utility. The inclusion and processing rates can be defined for exons and for introns and exist under different names [1]. Since, by definition, splice junctions know nothing about the set of exons that one might want to assess, the global exon inclusion and processing rates are computed for for a given set of annotated exons, as specified in the annotation file. In contrast, the inclusion and processing rates of SJ are computed for all splice junctions that remain intact after filtering, but also the annotated SJ are also assessed and reported.

```
Perl/zeta.pl -ssj <file> -ssc <file> -annot <gtf>
Inputs: -ssj and -ssc = SJ and SS counts in column (5);
        -annot, the annotation (gtf) with exons and introns
Options:
        -mincount = min denominator (will produce NA
            if denominator is smaller than this value)
        -stranded = 1(yes) or 0(no), default=1
```

Here, whenever a ratio is calculated, we usually relate inclusion quantity to the sum of inclusion and exclusion quantities. The latter, however, can be a small integer number and, therefore, a threshold is needed to cut off estimates with large standard errors. This threshold is -mincount. There is also an option to enforce strandless computation, but it will be deprecated in future versions.

The procedure of *zeta.pl* is to read and to index all SJs and then for each splice site to create a list of exons which start or end at the given splice site. Then, reading sequentially the count file, the program increments the counters for exon inclusion, exon exclusion, and also for SJ usage. The output is a GFF with the corresponding features, e.g.

```
chr1 SJPIPE exon   15 87 10 - . cosi "0.93962"; exc "0"; inc "747";
                                psi "1"; ret "48";
chr1 SJPIPE intron 65 67 83 - . cosi3 "1"; cosi5 "1"; nA "0"; nD "0";
                                nDA "22"; nDX "232"; nXA "253";
                                psi3 "0.08"; psi5 "0.08661";
```

where psi and cosi are exon percent-spliced-in and completeness of splicing rates; psi5, psi3, cosi5, cosi3 are the respective percent-spliced-in and completeness of splicing indices of an intron, measured from the 5'-end and from the 3'-end. The rest of the parameters are counts.

2.6 Master tables and endpoints

The *merge_gff.pl* utility is formally not a part of any pipeline but it can be used to merge the content of a number of gff/gtf files into a square (R-readable) matrix.

```
Perl/merge_gff.pl -i <input_file1> <label_1> ... -i <input_file_n> <label_n>
            -o feature_1 <output_1> -o feature_2 <output_2> ...
```

The program reads input files specified in the '-i' paramemter (could be many) one by one and selects features specified in '-o' paramemter. For instance,

```
Perl/merge_gff.pl -i cell_line1.gtf HELAS3 -i cell_line2.gtf NHEK
            -o psi result_psi.tsv -o cosi result_cosi.tsv
```

will generate two files, result_psi.tsv and result_cosi.tsv, each containing a square table, e.g.,

```
HELAS3    NHEK
chr1_100_200_+    0.52    0.75
chr1_300_400_+    0.00    1.00
...               ...     ...
```

It can be applied to any feature that was specified in the gtf input in cell_line1.gtf and cell_line2.gtf.

3 Quality control functions

3.1 Distribution of offsets

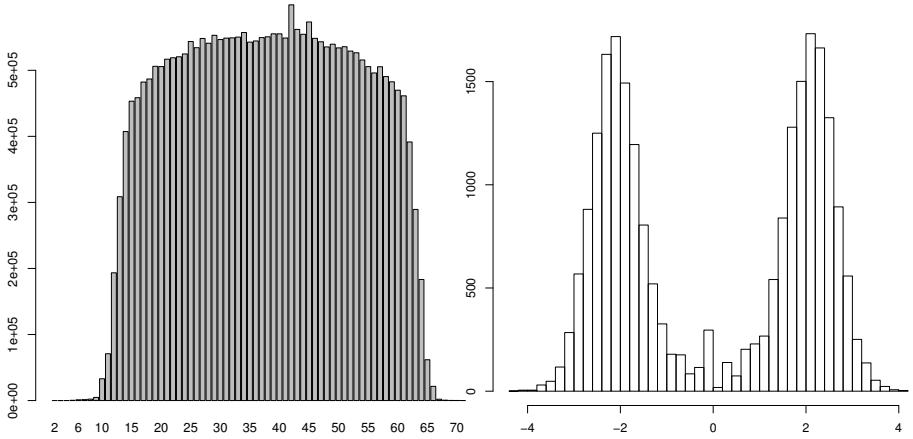
The following utility plot the distribution of offset frequencies, i.e., how frequently each offset value was seen. This distribution may be helpful in guessing the correct margin value because some mappers have intrinsic thresholds that may be different for reads overlapping SJ and SB.

```
Rscript offset.r <file.tsv> <file.pdf>
```

An example such diagram is shown in Figure 1a.

3.2 Strand disproportion

Another useful test is the distribution of $\log(c_+) - \log(c_-)$, where c_+ and c_- are the number of counts on the plus and the minus strand, respectively. SJ with



(a) Offset distribution (b) Disproportion test

$c_+ = 0$ or $c_- = 0$ are excluded. If the data is stranded and read1/read2 flags were set up correctly, then the distribution of $\log(c_+) - \log(c_-)$ shall be bimodal as shown in Figure 1b, reflecting the fact that one strand has much more split reads than the other.

```
Rscript disproportion.r <file.tsv> <file.pdf>
```

3.3 Summary stats on annotation status and splice sites

As soon as splice junction counts are computed, it makes sense to ask what proportion of splice junctions is annotated and what is the distribution of frequencies of splice site nucleotides. This is done by ***sjstat.r*** utility.

```
R/sjstat.r <file.tsv>  >  <file.log>
```

References

[1] D. D. Pervouchine, D. G. Knowles, and R. Guigo. Intron-centric estimation of alternative splicing from RNA-seq data. *Bioinformatics*, 29(2):273–274, Jan 2013.