**Event Analysis: description of output files**

This document describes the python workflow of Event Analysis, and provides descriptions of the the inputs and outputs of each script. To help illustrate the Event Analysis workflow, the example input and output files found on the git repository for Event Analysis will be used, and provide in *italicized blue text* in parenthesis at the appropriate sub-step.

The python workflow of Event Analysis can be divided into three main steps:

(1) Generation of annotations and the junction catalog (“Build annotations”)

(2) Alignments and coverage counts

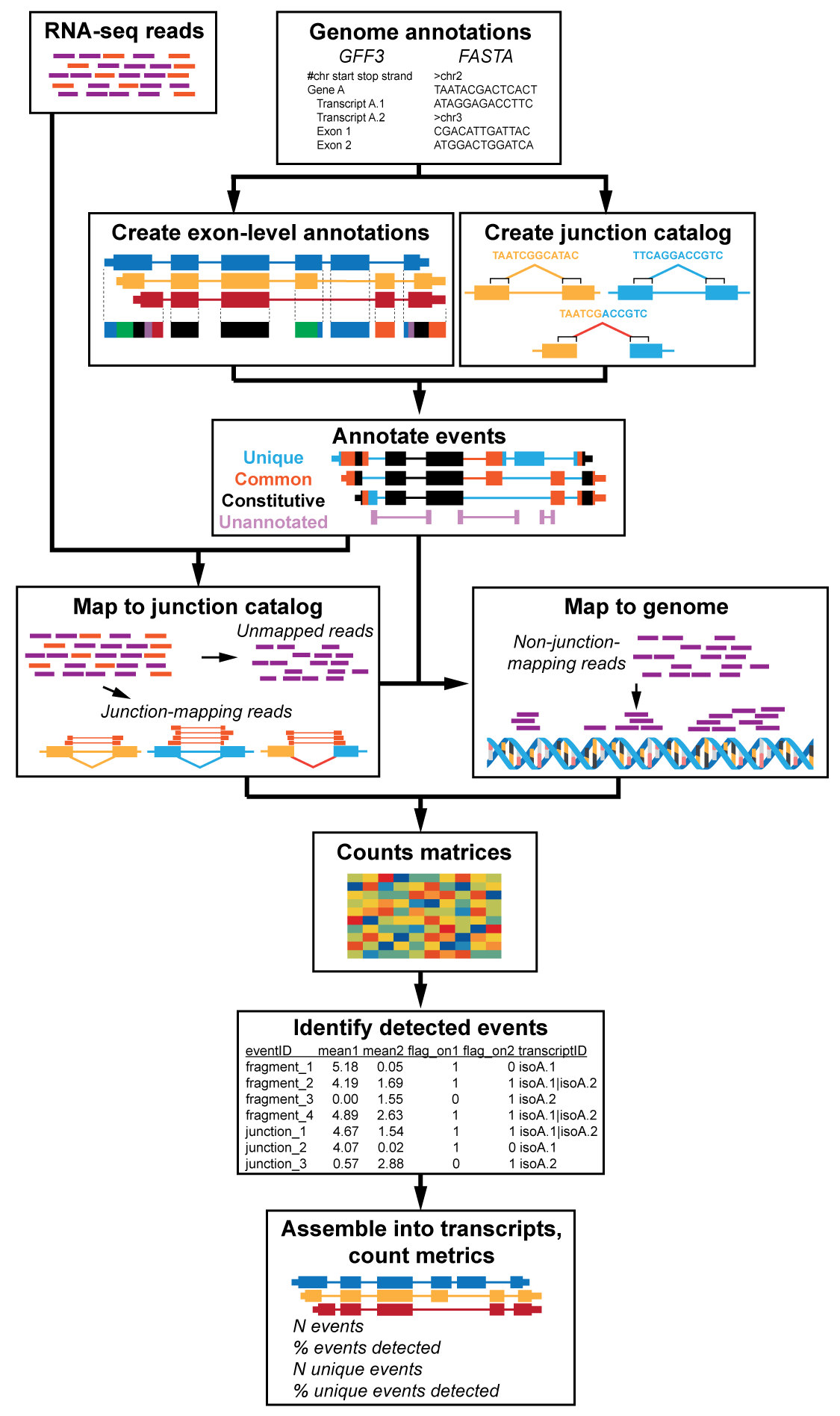
(3) Events to transcripts, and transcriptome reduction (“Identify transcriptome”)

Each of these three steps are further divided into smaller sub-steps, each of which is either a standalone python script (for Steps 1 and 3), or a collection of shell/python scripts used (for Step 2). Steps 1 and 3 form the core components of Event Analysis, namely building annotations and using alignments/coverage to summarize transcripts.

To demonstrate how to run these scripts, two shell scripts/makefiles are provided (xx and yy). These comprised the (multple py scripts, explain why each is separate etc.). If you are only interested in running the scripts without delving into the code, example shell scripts (xx and xx) have been provided to demonstrate how to invoke the two makefiles above.

A diagram of the workflow can be found in Figure 3 of the main manuscript, and is also provided below:

**Overview of Event Analysis**



**Prior to running the Event Analysis workflow**

In order to create annotations, Event Analysis requires a GFF3 file formatted in the style used by FlyBase. An example of this is provided in *input\_files/aconesa\_refseq\_gff3\_v4.gff*. The minimum required set of features are:

* chromosome
* gene
* transcript/mRNA
* exon

Note that in some GFF/GTF formats, if an exon is utilized by multiple transcripts there will be a separate entry for each exon-transcript pair. Event Analysis requires exons to be unique by their genomic coordinates and gene assignment, with all transcripts that include that exon listed together. A program (*programs/*r*convertGTF2GFF3.py*) is included to aid in convert any GFF or GTF file into the FlyBase format with the minimum set of required information. Note that this output MUST be sorted by chromosome, start and stop positions prior to generating annotations and the junction catalog.

An example shell script to demonstrate how to do this can be found in *example\_create\_formatted\_gff3.sh*

**Example output and data**

Example data in the form of the two simulations as described in the manuscript are available from the git, as well as from: <https://bio.rc.ufl.edu/pub/mcintyre/papers/newman_events_2017/simulated_data/>

Contained in this link are the FASTA files of simulated reads, example coverage counts for events, Event Analysis annotation files, input GFF and FASTA, and output for both simulations.

Example output from the mouse neural data used in the development of the approach can be found:

<https://bio.rc.ufl.edu/pub/mcintyre/papers/newman_events_2017/mouse_neural_data/>

**Example execution scripts**

While Event Analysis can be run as individual scripts, two example shell scripts are bundled in the Event Analysis git:

*example\_running\_run\_buildAnnotations.sh* and *example\_running\_run\_identifyTranscriptome.sh.* These can be used as templates for your own analysis, or for writing your own shell scripts to run Event Analysis.

The first of these (*example\_running\_run\_buildAnnotations.sh*) asks the user to input details about the annotations they want to build, such as the path to the correctly-formatted GFF file, genome FASTA file, read length, output directory for annotations, etc. It then calls the script *event\_analysis/run\_buildAnnotations.sh* with these as input and creates your annotations.

The second script (*example\_running\_run\_identifyTranscriptome.sh*) asks the use to input details about the data they wish to analyze, such as path to directories of counts, design file, grouping variable, detection parameters, etc. It then calls the script *event\_analysis/run\_identifyTranscriptome.sh* with these as input, and summarizes transcriptional events and reports summary statistics for each transcript in your annotations.

It is recommended that beginner users take the time to example each of the scripts mentioned here (both the example scripts and analysis shell scripts) to understand what Event Analysis requires, what it provides as output, what order to run programs in, etc.

**Phase 1: Generation of annotations and the junction catalog**

The generation of annotations and the junction catalog involves four main steps:

a. Extraction of transcriptional events

b. Generation of exon-level annotations

c. Creation of the junction catalog

d. Creation of event indices

Examples on how to run the programs associated with these steps can be found in thethe makefile: *event\_analysis/*r*un\_buildAnnotations.sh* (an example on how to run the makefile is also provided in *example\_running\_run\_buildAnnotations.sh).* Example output files using the RefSeq mm10 annotations are also listed in this document in parathesis in the “Output files” field

A workflow diagram showing the overall process and the relationships between each program is presented in the file *event\_analysis\_create\_annotations.png* located on the git.

**1.1 Extraction of transcriptional events**

Note: this step requires the use of the GFFutils python package, and was implemented in python 2.7.6**.** This should be compatible with Python3.5+.

*1.1.1 Extracting exons*

**Program:** programs/extractExons.py

**Description:** Extract exon annotations from GFF3 file, including their coordinates and associated transcripts and genes. The output is a BED file of genomic coordinates for each exon, and a CSV file with annotations

**Input files:** Sorted GFF3 file and GFF3 db file(*aconesa\_refseq\_gff3\_v4.gff*)

**Output files:** 6-column BED file of genomic coordinates of exons (*mm10\_exons.bed*)

Comma-delimited CSV file relating exons to genomic coordinates, transcripts and genes (*mm10\_exons.csv*)

The output BED file has the following fields:

chromosome, genomic start position of exon, genomic stop position of exon, exon ID, score (set to “.” or missing), strand

The CSV file has the following fields:

chromosome (chrom)

genomic start position of exon (start)

genomic stop position of exon (stop)

exon ID (exon\_id)

list of transcript ID separated by a “|” (transcript\_id)

gene ID (gene\_id)

*1.1.2 Extracting exonic regions*

**Program:** programs/extractFusions.py

**Description:** Build fusions from GFF3 file. A “fusion”, also called an “exonic region”, is a set of exons with overlapping genomic coordinates collapsed into a single region (in essence, a “union exon”). The output is a BED file genome coordinates for each fusion, and a CSV file with annotations (associated exon, gene). If the fusion contains exons from multiple genes, it is assigned a value of “1” to the binary indicator “flag\_multigene”

**Input files:** Sorted GFF3 file and GFF3 db file (*aconesa\_refseq\_gff3\_v4.gff*)

**Output files:** 6-column BED file of genomic coordinates of exonic regions

(*mm10\_fusions.bed*)

Tab-delminted TSV file relating exonic regions to exons and genes (*mm10\_fusions.csv*)

The output BED file has the following fields:

chromosome, genomic start position of exon, genomic stop position of exon, exon ID, score (set to “.” or missing), strand

The TSV file has the following fields:

Exonic region ID (fusion\_id)

Exon ID (exon\_id)

Gene ID (primary\_Fbgn)

indicator of whether exonic region is associated with more than one gene (flag\_multigene). 0=no, 1=yes

*1.1.3 Extracting exon fragments*

**Program:** programs/extractExonFragments.py

**Description:** Identify each distinct exon fragment in the genome. A fusion is divided into individual fragments (exon fragments) based on transcript assignment, and is used to identify which pieces of sequence within a fusion belong to which transcript(s). The output is a CSV file of all distinct exon fragments in the genome, the coordinates of its corresponding fusion, and the exons assigned to each fragment

**Input files:** Sorted GFF3 file and GFF3 db file (*aconesa\_refseq\_gff3\_v4.gff*)

**Output files:** CSV file of exon fragment regions, and associated exons (*mm10\_exon\_fragments.csv*)

The output CSV file contains the following columns:

chromosome (chr)

genomic start position of exonic region (region\_start)

genomic stop position of exonic region (region\_stop)

genomic start position of exon fragment (fragment\_start)

genomic stop position of eexon fragment (fragment\_stop)

“|”-delimited list of exon IDs associated with exon fragment (exon\_id)

*1.1.4 Extracting unambiguous introns*

**Program:** programs/build\_unambiguous\_introns\_from\_fusions.py

**Description:** Take the fusion annotations and derive introns by inverting fusions within a gene. Introns derived from multigene fusions are not reported, as the introns sequences may not be completely unambiguous. The output is a BED file for calculating coverage for intronic regions, and an annotation CSV linking introns to their flanking fusions

**Input files:** 6-column BED file of genomic coordinates of fusions (*mm10\_fusions.bed*)

TSV file of fusions, associated exons, genes, and whether the fusion contains exons from multiple genes (*mm10\_fusions.tsv*)

**Output files:**

BED file for calculating coverage for intronic regions (*mm10\_introns\_from\_fusions.bed*)

CSV file linking introns to their flanking fusions (*mm10\_introns\_from\_fusions.csv*)

The output BED file (*mm10\_fusions.bed*) contains the following columns:

chromosome

genomic start position of intron

genomic stop position of intron

intron ID in the format “[5' exonic region ID]|intron|[3' exonic region ID]”

The output CSV file (*mm10\_introns\_from\_fusions.csv*) contains the following columns:

intron ID in the format “[5' exonic region ID]|intron|[3' exonic region ID]” (intron\_id)

chromosome (chr)

genomic start position of intron (intron\_start)

genomic stop position of intron (intron\_stop)

Length of the intron (intron\_size)

Gene associated with intron (gene\_id)

5' exonic region IDs (exonic\_region\_id\_5prime)

3' exonic region IDs (exonic\_region\_id\_3prime)

Two indicator variables to denote whether either flanking exonic region contains exons from multiple genes (flag\_multigene\_5prime\_exon, flag\_multigene\_3prime\_exon). Note that these should always be set to “0”, and only serve as checks.

*1.1.5 Extracting all possible, logical junctions*

**Program:** programs/extractJunctions.py

**Description:** Extract all possible, logical junctions within a gene from GFF3 file. A logical junction consists of any possible pair of exons within a gene. Junctions are not made between overlapping exons, or from exons where the donor exon is 3' of the acceptor exon. The output is a 12-column BED file of all possible junctions

**Input files:** Sorted GFF3 file and GFF3 db file (*aconesa\_refseq\_gff3\_v4.gff*)

**Output files:** 12-column BED file of all possible junctions. (*mm10\_logical\_junctions.bed*)

The output BED file has the following columns (standard BED-12 format):

chromosome

genomic start position

genomic stop position

junction ID in the form of the donor-acceptor exon combination

score (always “.” or missing)

strand

genomic start position

genomic stop position

RGB color (always 255,0,0)

number of blocks

block sizes

block start positions relative the to genomic start position

*1.1.6 Extracting annotated junctions*

**Program:** programs/extractTranscriptJunctions.py

**Description:** Extract all junctions from transcripts (“annotated junctions”) from the GFF3 file. These will be used to identify which logical junctions are annotated, and which are not. The output is a CSV file consisting of the junctions coordinates, associated transcripts and genes

**Input files:** Sorted GFF3 file and GFF3 db file (*aconesa\_refseq\_gff3\_v4.gff*)

**Output files:** A CSV file of junctions, their coordinates, associated transcripts and genes (mm10\_annotated\_junctions.csv)

The output CSV file contains the following columns:

junction ID in the form of the donor-acceptor exon combination (junction\_id)

Junction coordinates in the format [chr]:[5'\_position]:[3'\_position]:strand (junction\_coordinates)

transcript ID (transcript\_id)

gene ID (gene\_id)

*1.1.7 Extracting exon-skipping junction annotations*

**Program:** programs/extractSkippedExons.py

**Description:** Create exon-skipping annotations for all possible logical junctions. For each junction, the number of exons that junction “skips” over are counted. There are two outputs: (1) a CSV file containing the number of exons skipped by each junction and a concatentated list of skipped exons, and (2) a list of exons skipped by each junction

**Input files:** Sorted GFF3 file and GFF3 db file (*aconesa\_refseq\_gff3\_v4.gff*)

**Output files:** CSV file containing the number of exons skipped by each junction and a concatentated list of skipped exons (*mm10\_exonskipping\_junctions.csv*)

CSV list of exons skipped by each junction (*mm10\_skipped\_exons\_list.csv*)

The first output CSV file (number of exons skipped by each junction, etc.; e.g. *mm10\_exonskipping\_junctions.csv*) contains the following columns:

junction ID in the form of the donor-acceptor exon combination (junction\_id)

indicatior of whether the junction skips at least one exon (flag\_exonskip)

Number of exons skipped by junction (num\_skipped\_exons)

List of skipped exons, separated by a “|” (cat\_skipped\_exons)

The second output CSV file (list of exons skipped; e.g. *mm10\_skipped\_exons\_list.csv*) contains the following columns:

junction ID in the form of the donor-acceptor exon combination (junction\_id)

ID of exons skipped by junction (skipped\_exon\_id)

indicatior of whether the junction skips at least one exon (flag\_exonskip)

*1.1.8 Extracting exon-intron border junctions*

**Program:** programs/build\_donor\_border\_junctions.py,

programs/build\_acceptor\_border\_junctions.py

**Description:** Create exon-intron “border junctions” (sequences that span the border of exons and introns) from the 5'-end (donor end) of introns from the GFF3 file. It outputs two files: a 12-column BED file for donor border junctions , and a CSV describing exon(s) associated with an event. Intron retention events are only made for introns that are unambiguously intronic sequence. Events are not made for intron sequences that overlap with exonic regions, For plus-strand genes, retained introns are made from the end of exons. For minus-strand genes, retained introns are made from the start of exons. Events are not made for the last exons for a gene (genomic 3' last for plus-strand genes, genomic 5' first for minus-strand genes).

This is repeated for the 3'-end (acceptor end) of introns.

**Input files:** Sorted GFF3 file and GFF3 db file (*aconesa\_refseq\_gff3\_v4.gff*)

**Output files:** 12-column BED file for donor/acceptor border junctions (*mm10\_donor\_border\_junctions.bed, mm10\_acceptor\_border\_junctions.bed*)

CSV describing exon(s) associated with a border junction (*mm10\_donor\_border\_junctions.csv, mm10\_acceptor\_border\_junctions.csv*)

The output BED files has the following columns : (e.g *mm10\_donor\_border\_junctions.bed, mm10\_acceptor\_border\_junctions.bed*)

chromosome

genomic start position

genomic stop position

junction ID in the form of the donor-acceptor exon combination

score (always “.” or missing)

strand

RGB color (always 255,0,0)

number of blocks

block sizes

block start positions relative the to genomic start position

The output CSV files contain the following columns: (e.g *mm10\_donor\_border\_junctions.csv, mm10\_acceptor\_border\_junctions.csv*)

Gene ID (gene\_id)

ID of border junction (event\_id)

chromosome (chr)

strand (strand)

Position of intron start site within border junction (intron\_position). This will be set to “-999999” if the border junction is situated at the very start or end of a gene, and will be removed. The indicator flag\_lastexon will also be set to 1.

Associated donor/acceptor exon ID (exon\_id)

List of all exon IDs potentially associated with that donor site (exon\_cat)

Indicator as to whether the border junction is derived from the final exon in the gene (flag\_lastexon). If set to 1, then this border junction will be removed

**1.2 Generation of exon-level annotations**

Note: this step has been implemented in python 3.6

*1.2.1 Create exon annotations*

**Program:** programs/import\_and\_format\_exons.py

**Description:** Import exon annotations and reformat, indicating whether each exon has an alternative donor or acceptor, the set of associated transcripts, and its gene

**Input files:** 6-column BED file of genomic coordinates of exons (*mm10\_exons.bed*)

Comma-delimited CSV file relating exons to genomic coordinates, transcripts and genes (*mm10\_exons.csv*)

**Output files:** CSV file of formatted exon annotations, including whether the exon has an alternative donor/acceptor (*mm10\_exon\_annotations.csv*)

The output CSV file contains the following columns:

Gene ID (gene\_id)

Exon ID (exon\_id)

Exon group number (exon\_group). Overlapping exons within a gene are grouped together to check for the presence of alternative donor/acceptor sites

chromosome (chrom)

genomic start position (start)

genomic stop position (stop)

strand (strand)

List of associated transcript ID separated by a “|” (transcript\_id)

Indicator as to whether there are multiple donor sites within the exon group. If an exon groups has more than one distinct donor site, then all exons within this group are flagged as having alternative donors (flag\_alt\_donor)

Indicator as to whether there are multiple acceptor sites within the exon group. If an exon groups has more than one distinct acceptor site, then all exons within this group are flagged as having alternative acceptors (flag\_alt\_acceptor)

*1.2.2 Create exonic region (fusion) annotations*

**Program:** programs/import\_and\_format\_fusions.py

**Description:**  Import fusion annotations and reformat, indicating the annotation frequency of each fusion (unique, common, consitutive), the exons, transcripts and genes associated with each fusion. The outputs are a CSV file of formatted fusion annotations and a 4-column BED file for calculating fusion coverage

**Input files:** 6-column BED file of genomic coordinates of exonic regions

(*mm10\_fusions.bed*)

Tab-delminted TSV file relating exonic regions to exons and genes (*mm10\_fusions.csv*)

Comma-delimited CSV file relating exons to genomic coordinates, transcripts and genes (*mm10\_exons.csv*)

**Output files:** CSV file of formatted fusion annotations, including the annotation frequency (unique, common, constitutive) (*mm10\_fusion\_annotations.csv*)

4-column BED file for calculating fusion coverage (*mm10\_fusions\_coverage.bed*)

The output CSV has the following columns: (e.g *mm10\_fusion\_annotations.csv*)

Exonic region ID (fusion\_id)

chromosome (chr)

genomic start position (fusion\_start)

genomic stop position (fusion\_stop)

strand (strand)

Number of exons in exonic region (exons\_per\_fusion)

List of exon IDs separated by a “|” (exon\_id)

Number of transcripts assoicated with region (transcripts\_per\_fusion)

List of transcriptIDs separated by a “|” (transcript\_id)

Number of genes associated with region (genes\_per\_fusion)

List of gene IDs separated by a “|” (gene\_id)

Annotation frequency as either “unique”, “common”, “constitutive”, “unannotated” (annotation\_frequency)

indicator of whether exonic region is associated with more than one gene (flag\_multigene). 0=no, 1=yes

Total number of transcripts associated with all genes associated with region (total\_transcripts\_per\_genes)

The output BED file has the following columns: (e.g. *mm10\_fusions\_coverage.bed*)

chromosome

genomic start position

genomic stop position

exonic region ID

*1.2.3 Create exon fragment annotations*

**Program:** programs/import\_and\_format\_fragments.py

**Description:** Import fragment and fusion annotations and reformat, indicating the annotation frequency of each fusion (unique, common, consitutive), the exons, transcripts and genes associated with each fusion. Each fragment is assigned an identifier consisting of its corresponding fusion, and it's 5'-genomic order within the fusion. The outputs are a CSV file of formatted fusion annotations and a 4-column BED file for calculating fusion coverage

**Input files:** CSV file of exon fragment regions, and associated exons (*mm10\_exon\_fragments.csv)*

6-column BED file of genomic coordinates of exonic regions

(*mm10\_fusions.bed*)

Tab-delminted TSV file relating exonic regions to exons and genes (mm10\_fusions.csv)

Comma-delimited CSV file relating exons to genomic coordinates, transcripts and genes (*mm10\_exons.csv*)

**Output files:** CSV file of formatted exon fragment annotations, including the annotation frequency (unique, common, constitutive) (*mm10\_exon\_fragment\_annotations.csv*)

4-column BED file for calculating exon fragment coverage (*mm10\_exon\_fragments\_coverage.bed*)

The output CSV has the following columns: (e.g. *mm10\_exon\_fragment\_annotations.csv*)

ID of exon fragment, formatted as [associated exonic region]:[fragment number within exonic region] (fragment\_id)

chromosome (chr)

genomic start position of exon fragment (fragment\_start)

genomic stop position of exon fragment (fragment\_stop)

ID of exonic region associated with exon fragment (fusion\_id)

genomic start position of associated exonic region (fusion\_start)

genomic stop position of associated exonic region (fusion\_stop)

list of exon IDs, separated by a “|” (exon\_id)

number of exons associated with exon fragment (num\_exons\_per\_frag)

list of transcript IDs, separated by a “|” (transcript\_id)

number of transcripts associated with exon fragment (num\_xscripts\_per\_frag)

list of gene IDs, separated by a “|” (gene\_id)

number of genes associated with exon fragment (num\_genes\_per\_frag)

Annotation frequency as either “unique”, “common”, “constitutive”, “unannotated” (annotation\_frequency)

indicator of whether exonic region is associated with more than one gene (flag\_multigene). 0=no, 1=yes

Total number of transcripts associated with all genes associated with region (total\_transcripts\_per\_genes)

The output BED file has the following columns: (e.g. *mm10\_exon\_fragments\_coverage.bed*)

chromosome

genomic start position

genomic stop position

exon fragment ID

**1.3 Creation of the junction catalog**

Note: this step has been implemented in python 3.6

*1.3.1 Import and format junctions*

**Program:** import\_and\_format\_junctions.py

**Description:** Imports the junctions BED file created in step 1.1.5 and reformats it for adding in donor and acceptor exon information, and other annotations

**Input files:** 12-column BED file of all possible junctions (*mm10\_logical\_junctions.bed*)

**Output files:** CSV file consisting of the junctions, their coordinates, and donor/acceptor exons (*mm10\_logical\_junctions\_formatted.csv*)

The output CSV file has the following columns:

Chromosome (chr)

Strand (strand)

ID of junction in the format [5' exon ID]:[3' exon ID] (event\_id)

\* note: Exons are denoted as 5'/3' relative to the + strand of the chromosome

ID of 5'/donor exon (donor\_exon)

size (in bp) of extracted donor site (donor\_size)

genomic position of donor site start position (donor\_start)

genomic position of donor site stop position (donor\_stop)

ID of 3'/acceptor exon (acceptor\_exon)

size (in bp) of extracted acceptor site (acceptor\_size)

genomic position of acceptor site start position (acceptor\_start)

genomic position of acceptor site stop position (acceptor\_stop)

Splicing event type, will be populated as “exon\_junction” (event\_type)

*1.3.2 Flag annotated junctions*

**Program:** import\_and\_flag\_transcript\_junctions.py

**Description:** Flag which of the set of all possible, logical junctions derived from GFF are also annotated to known transcripts ("annotated junctions").

**Input files:** CSV file consisting of the junctions, their coordinates, and donor/acceptor exons (*mm10\_logical\_junctions\_formatted.csv*)

CSV file consisting of annotated junctions coordinates, associated transcripts and genes (*mm10\_annotated\_junctions.csv*)

**Output files:** CSV file of junctions and coordinates, indicating if junction is annotated or not, and what transcripts they come from (*mm10\_logical\_junctions\_flag\_annotated.csv*)

The output CSV file adds the following columns:

Number of transcripts assoicated with exon pair/junction (num\_transcripts)

List of transcripts separated by a “|”. If there are no transcripts associated with the exon pair/junction, then this populated with “Unannotated” (transcript\_id)

(flag\_junction\_annotated)

indicator of whether the exon pair/junction has at least one associated transcript, and thus can be classified as an “annotated junction” (flag\_junction\_annotated). 0=no, 1=yes

*1.3.3 Flag exon-skipping junctions*

**Program:** import\_exon\_skipping\_annotations.py

**Description:** Import exon-skipping annotations for each logical junction and add to junction annotations

**Input files:** CSV file of junctions and coordinates, indicating if junction is annotated or not, and what transcripts they come from (*mm10\_logical\_junctions\_flag\_annotated.csv*)

CSV file containing the number of exons skipped by each junction and a concatentated list of skipped exons (*mm10\_exonskipping\_junctions.csv*)

**Output files:** CSV file of junctions and coordinates, indicating if junction is annotated or not, what transcripts they come from, and whether they skip over any exons (*mm10\_logical\_junctions\_flag\_exonskip.csv*)

The output CSV file adds the following columns:

Indicator of whether the exon pair/junction skips at least one exon, and thus can be classified as an “exon-skipping junction” (flag\_exonskip). 0=no, 1=yes

Number of exons skipped by junctions (num\_skipped\_exons)

List of exons skipped by junction separated by a “|” (cat\_skipped\_exons)

*1.3.4 Combine exon-exon junctions and exon-intron border junctions*

**Program:** append\_border\_junctions.py

**Description:** Append border junctions (sequences that span exon-intron borders) to annotated junction catalog and add missing information

**Input files:** CSV file of junctions and coordinates, indicating if junction is annotated or not, what transcripts they come from, and whether they skip over any exons (*mm10\_logical\_junctions\_flag\_exonskip.csv*)

12-column BED file for donor border junctions (*mm10\_donor\_border\_junctions.csv*)

12-column BED file for acceptor border junctions(*mm10\_acceptor\_border\_junctions.csv*)

**Output files:** CSV file consisting of logical junctions and border junctions (*mm10\_logical\_junctions\_and\_border\_junctions.csv*)

The output CSV file adds the following columns:

Indicator of whether the junction is a border junction or not (flag\_border\_junction)

*1.3.5 Add exon information*

**Program:** add\_exon\_info\_to\_junctions.py

**Description:** Add exon annotations to the set of logical junctions and border junctions

**Input files:** CSV file consisting of logical junctions and border junctions (*mm10\_logical\_junctions\_and\_border\_junctions.csv*)

CSV file of exons, genomic coordinates, transcripts and genes (*mm10\_exon\_annotations.csv*)

**Output files:** CSV file consisting of logical junctions and border junctions with exon annotations (*mm10\_junctions\_w\_exon\_info.csv*)

The output CSV file adds the following columns:

Gene associated with donor/5' exon (donor\_gene)

Group number of donor/5' exon. Exons within a gene that have overlapping genomic coordinates are assigned to the same group (donor\_group)

Indicator of whether there are multiple annotated donor sites within the donor group (flag\_alt\_donor)

Gene associated with acceptor/3' exon (acceptor\_gene)

Group number of acceptor/3' exon. Exons within a gene that have overlapping genomic coordinates are assigned to the same group (acceptor\_group)

Indicator of whether there are multiple annotated acceptor sites within the acceptor group (flag\_alt\_acceptor)

*1.3.6 Collapse duplicates*

**Program:** collapse\_duplicate\_junctions.py

**Description:** Collapse all junctions to their genomic coordinates (format chr:donor:acceptor:strand), concatenating/collapsing all annotations and flags as well

**Input files:** CSV file consisting of logical junctions and border junctions with exon annotations (*mm10\_junctions\_w\_exon\_info.csv*)

CSV file of exons, genomic coordinates, transcripts and genes (*mm10\_exons.csv*)

**Output files:** CSV file consisting of the set of distinct junctions, collapsed by genomic coordinate (*mm10\_junctions\_full\_annotation.csv*)

The output CSV has the following columns:

ID of junction in the format [chr]:[donor]:[acceptor]:[strand] (junction\_id)

chromosome (chr)

genomic position of donor/5' site start position (donor\_start)

genomic position of donor/5' site stop position (donor\_stop)

genomic position of acceptor/3' site start position (acceptor\_start)

genomic position of acceptor/3' site stop position (acceptor\_stop)

strand (strand)

Number of transcripts associated with junction (num\_transcripts)

List of transcripts associated with junction separated by a “|”. If there are no transcripts, then this is populated with “Unannotated” (transcript\_id)

Number of genes associated with junction (num\_genes)

List of genes associated with junction separated by a “|” (gene\_id)

Total number of transcripts associated with all genes associated with junction (total\_transcripts\_per\_gene)

Annotation frequency as either “unique”, “common”, “constitutive”, “unannotated” (annotation\_frequency)

(flag\_multigene)

Number of donor/5' exons assoicated with junction (num\_donor\_exons)

List of donor/5' exons associated with junction separated by a “|” (donor\_exon\_id)

Number of acceptor/3' exons assoicated with junction (num\_acceptor\_exons)

List of acceptor/3' exons associated with junction separated by a “|” (acceptor\_exon\_id)

Indicator of whether the junction is associated with at least transcript (flag\_junction\_annotated)

Indicator of whether the junction is a border junction (flag\_border\_junction)

Indicator of whether junction uses an alternative donor/5' site (flag\_alt\_donor)

Indicator of whether junction uses an alternative acceptor/3' site (flag\_alt\_acceptor)

Indicator of whether the junction skips at least one exon (flag\_exonskip). 0=no, 1=yes

List of exons skipped by junction separated by a “|” (skipped\_exon\_id)

Number of exons skipped by junctions (num\_skipped\_exons)

*1.3.7 Create junction sequence references*

**Program:** extract\_junction\_sequence.py

**Description:** Extract junctions sequences for collapsed junctions. Junctions with identical sequences (but different coordinates) are also collapsed: these will be treated as a single sequence for alignments

**Input files:** CSV file consisting of the set of distinct junctions, collapsed by genomic coordinate (*mm10\_junctions\_full\_annotation.csv*)

Genome FASTA file, containing sequences for all chromosomes (*mm10\_for\_bedtools\_v2.fa*)

**Output files:** CSV file of junction annotations (*mm10\_junction\_annotations.csv*)

CSV file relating junctions to sequences (*mm10\_junction\_to\_sequence\_index.csv*)

FASTA file of all junction sequences (*mm10\_junctions.fa*)

BED file of calculating junction coverage (*mm10\_junctions\_coverage.bed*)

The output CSV of junction annotations (e.g. *mm10\_junction\_annotations.csv*) contains the following columns:

ID of junction in the format [chr]:[donor]:[acceptor]:[strand] (junction\_id)

chromosome (chr)

genomic position of donor/5' site start position (donor\_start)

genomic position of donor/5' site stop position (donor\_stop)

genomic position of acceptor/3' site start position (acceptor\_start)

genomic position of acceptor/3' site stop position (acceptor\_stop)

strand (strand)

List of transcripts associated with junction separated by a “|”. If there are no transcripts, then this is populated with “Unannotated” (transcript\_id)

List of genes associated with junction separated by a “|” (gene\_id)

Annotation frequency as either “unique”, “common”, “constitutive”, “unannotated” (annotation\_frequency)

Indictor of whether junction is associated with more than 1 gene (flag\_multigene)

Indicator of whether the junction is associated with at least transcript (flag\_junction\_annotated)

Indicator of whether the junction is a border junction (flag\_border\_junction)

Indicator of whether junction uses an alternative donor/5' site (flag\_alt\_donor)

Indicator of whether junction uses an alternative acceptor/3' site (flag\_alt\_acceptor)

Indicator of whether the junction skips at least one exon (flag\_exonskip). 0=no, 1=yes

The output CSV relating junctions to sequences (*mm10\_junction\_to\_sequence\_index.csv*) contains the following columns:

Junction ID in the format [chr]:[donor]:[acceptor]:[strand] (junction\_id)

The ID of unique junction sequence, as junction\_####### (sequence\_id)

The nucleotide sequence of the unique junction (sequence)

The output FASTA file of all junction sequences (*mm10\_junctions.fa*) contains the list of unique junction sequences and their associated sequence ID

The output BED file of calculating junction coverage (*mm10\_junctions\_coverage.bed*) contains the following columns:

The ID of unique junction sequence, as junction\_#######

The 0-based start position of the junction sequence (always 0)

The stop position of the junction sequence (equivalent to its length)

**1.4 Creation of event indices**

Note: this step has been implemented in python 3.6

*1.4.1 Event-to-transcript index*

**Program:** build\_Event2Transcript\_index.py

**Description:** Take the exon fragment annotations and junction annotations and create an event-to-transcript-to-gene index. For unannotated events (unannotated junctions, border junctions), the transcript\_id is assigned as “unannotated”

**Input files:** CSV file of junction annotations (*mm10\_junction\_annotations.csv*)

CSV file of formatted exon fragment annotations, including the annotation frequency (unique, common, constitutive) (*mm10\_exon\_fragment\_annotations.csv*)

CSV file of exons, genomic coordinates, transcripts and genes (*mm10\_exons.csv*)

**Output files:** CSV file of events, their transcripts and genes (one transcript/gene per line entry) and the annotation frequency of each event (unique, common, constitutive, unannotated) (*mm10\_event2transcript2gene\_index.csv*)

The output CSV (*mm10\_event2transcript2gene\_index.csv*) contains the following columns:

ID of the exon fragment or junction (event\_id)

Transcript ID (“Unannotated” if no transcript) transcript\_id)

Gene ID (gene\_id)

Annotation frequency as either “unique”, “common”, “constitutive”, “unannotated” (annotation\_frequency)

Indictor of whether junction is associated with more than 1 gene (flag\_multigene)

*1.4.2 Border junction-to-intron index*

**Program:** build\_intron2border\_junction\_index.py

**Description:** Creates an intron-to-border junction index file used by Event Analysis to report the read coverage of introns, their associated border junctions and flanking exonic regions (fusions), to aid the user in deciding whether there is evidence on intron retention, alternative/novel splice usage, etc.

**Input files:** CSV file linking introns to their flanking exonic regions (*mm10\_introns\_from\_fusions.csv*)

CSV file of junction annotations (*mm10\_junction\_annotations.csv*)

**Output files:** CSV file of introns, 5' and 3' border junctions, and 5' and 3' fusions (*mm10\_intron2border\_junction\_index.csv*)

The output CSV (*mm10\_intron2border\_junction\_index.csv*) has the following columns:

intron ID in the format “[5' exonic region ID]|intron|[3' exonic region ID]” (intron\_id)

chromosome (chr)

genomic start position of intron (intron\_start)

genomic stop position of intron (intron\_stop)

Gene associated with intron (gene\_id)

5' exonic region ID (exonic\_region\_id\_5prime)

5' border junction ID (border\_junction\_id\_5prime)

3' exonic region IDs (exonic\_region\_id\_3prime)

3' border junction ID (border\_junction\_id\_3prime)

**Phase 2: Alignments and coverage counts**

Alignments occur in two steps: (1) align all reads to junction sequences and quantify junctions; (2) align non-junction reads to the genome and quantify exonic sequences.

Any aligner can be used for these steps. For the development of Event Analysis, Bowtie (v0.12.9) was used for junction alignments, and BWA-MEM (v0.7.12) was used for genome alignments. We have found that the choice of aligner will impact event detection (particularly junctions). We have also found that using Bowtie as the aligner for quantifying junction sequences performs better than others (e.g. STAR, SOAP2) Examples of the scripts are provided on the git, and are also presented here to illustrate the workflow.

**2.1 Build alignment reference**

*2.1.1 Junction reference database*

**Submission script:** submit\_build\_bowtie\_junction\_index.sbatch

**Program:** bowtie (v0.12.9), samtools (v1.3.1)

**Description:** Create Bowtie reference for junction sequences

**Input files:** Junction sequence FASTA (*mm10\_junctions.fa*)

**Output files:** Indexed junction sequence FASTA (*mm10\_junctions.fa.fai*)

Bowtie sequence index (*mm10\_junctions\_BT1.\**)

*2.1.2 Genome reference database*

**Submission script:** submit\_build\_bwa\_index.sbatch

**Program:** bwa (v0.7.12) samtools (v1.3.1)

**Description:** Create BWA reference for genome sequence

**Input files:** genome FASTA file (*mm10\_for\_bedtools\_v2.fa*)

**Output files:** Indexed genome FASTA (*mm10\_for\_bedtools\_v2.fa.fai*)

BWA sequence index (*mm10\_for\_bedtools\_v2\_BWA.\**)

**2.2 Junction alignments**

*2.2.1 Align reads to junctions*

**Submission script:** aln\_junctions.sbatch

**Program:** bowtie (v0.12.9), python (v2.7.6), alignment\_functions.sh, identify\_quality.py

**Description:** Align reads to junction sequences.

**Input files:** Design file, RNA-seq read files (FASTQ/FASTA), Bowtie sequence index (*mm10\_junctions\_BT1.\**)

**Output files:** SAM file of junction alignments (1 per sample), FQ files of unaligned reads (1 per sample)

*2.2.2 Generate junction mpileups*

**Submission script:** mpileup\_splicing.sbatch

**Program:** samtools (v1.3.1)

**Description:** Generate mpileups for junctions

**Input files:**  Design file, SAM file of junction alignments, Indexed junction sequence FASTA (*mm10\_junctions.fa.fai*)

**Output files:** BAM file of junction alignments, MPILEUP file of junction sequence counts at each base (1 each per sample)

*2.2.3 Calculate junction coverage*

**Submission script:** coverage\_counts\_splicing.sbatch

**Program:** python (v2.7.6), rpkm\_calculate.py

**Description:** Calcuate coverage for junction sequences (1 per sample)

**Input files:** Design file, SAM file of junction alignments, MPILEUP file of junction sequence counts at each base, BED file of junction sequences (*mm10\_junctions\_coverage.bed*)

**Output files:** CSV file of coverage counts (1 per sample)

CSV file has the following columns:

Sample ID (sample\_id)

Reference sequence/region ID, as listed in the BED file (fusion\_id)

Number of mapped reads (mapped\_reads)

Read length (read\_length)

Length of reference sequence/region ID (region\_length)

Region sequence depth (region\_depth)

Number of reads in the region/sequences, calculcate as region\_depth/read\_length (reads\_in\_region)

Average per nucleotide coverage, calculated as region\_depth/region\_length (apn)

Reads per kilobase per million mapped reads (rpkm)

average coverage across region (mean)

standard deviation of coverage across region (std)

coefficient of variation (cv)

**2.3 Genome alignments**

*2.3.1 Align reads to genome*

**Submission script:** aln\_genome\_bwa.sbatch

**Program:** samtools (v1.3.1), bwa (v0.7.12), python (v2.7.6)

**Description:** Align non-junction reads to genome.

**Input files:** Design file, Non-junction RNA-seq reads from step 2.2.1 (FASTQ/FASTA), BWA sequence index (*mm10\_for\_bedtools\_v2\_BWA.\**)

**Output files:** SAM file of genome alignments (1 per sample)

*2.3.2 Generate genome mpileups*

**Submission script:** mpileup\_genome.sbatch

**Program:** samtools (v1.3.1)

**Description:** Generate mpileups for genome

**Input files:**  Design file, SAM file of genome alignments, Indexed genome sequence FASTA (*mm10\_for\_bedtools\_v2.fa.fai*)

**Output files:** BAM file of genome alignments, MPILEUP file of genome sequence counts at each base (1 each per sample)

*2.3.3 Calculate exonic region coverage*

**Submission script:** coverage\_counts\_fusions.sbatch

**Program:** python (v2.7.6), rpkm\_calculate.py, BWASplitSAM.py

**Description:** Calcuate coverage for exonic regions (1 per sample)

**Input files:** Design file, SAM file of genome alignments, MPILEUP file of genome sequence counts at each base, BED file of exonic region/fusion genomic positions (*mm10\_fusions\_coverage.bed*)

**Output files:** CSV file of coverage counts (1 per sample)

CSV file has the following columns:

Sample ID (sample\_id)

Reference sequence/region ID, as listed in the BED file (fusion\_id)

Number of mapped reads (mapped\_reads)

Read length (read\_length)

Length of reference sequence/region ID (region\_length)

Region sequence depth (region\_depth)

Number of reads in the region/sequences, calculcate as region\_depth/read\_length (reads\_in\_region)

Average per nucleotide coverage, calculated as region\_depth/region\_length (apn)

Reads per kilobase per million mapped reads (rpkm)

average coverage across region (mean)

standard deviation of coverage across region (std)

coefficient of variation (cv)

*2.3.4 Calculate exon fragment coverage*

**Submission script:** coverage\_counts\_fragments.sbatch

**Program:** python (v2.7.6), rpkm\_calculate.py

**Description:** Calcuate coverage for exon fragments (1 per sample)

**Input files:** Design file, SAM file of genome alignments, MPILEUP file of genome sequence counts at each base, BED file of exon fragment genomic positions (*mm10\_exon\_fragments\_coverage.bed*)

**Output files:** CSV file of coverage counts (1 per sample)

CSV file has the following columns:

Sample ID (sample\_id)

Reference sequence/region ID, as listed in the BED file (fusion\_id)

Number of mapped reads (mapped\_reads)

Read length (read\_length)

Length of reference sequence/region ID (region\_length)

Region sequence depth (region\_depth)

Number of reads in the region/sequences, calculcate as region\_depth/read\_length (reads\_in\_region)

Average per nucleotide coverage, calculated as region\_depth/region\_length (apn)

Reads per kilobase per million mapped reads (rpkm)

average coverage across region (mean)

standard deviation of coverage across region (std)

coefficient of variation (cv)

*2.3.5 Calculate intron coverage*

**Submission script:** coverage\_counts\_introns.sbatch

**Program:** python (v2.7.6), rpkm\_calculate.py

**Description:** Calcuate coverage for introns (1 per sample)

**Input files:** Design file, SAM file of genome alignments, MPILEUP file of genome sequence counts at each base, BED file of intron genomic positions (*mm10\_introns\_from\_fusions.bed*)

**Output files:** CSV file of coverage counts (1 per sample)

CSV file has the following columns:

Sample ID (sample\_id)

Reference sequence/region ID, as listed in the BED file (fusion\_id)

Number of mapped reads (mapped\_reads)

Read length (read\_length)

Length of reference sequence/region ID (region\_length)

Region sequence depth (region\_depth)

Number of reads in the region/sequences, calculcate as region\_depth/read\_length (reads\_in\_region)

Average per nucleotide coverage, calculated as region\_depth/region\_length (apn)

Reads per kilobase per million mapped reads (rpkm)

average coverage across region (mean)

standard deviation of coverage across region (std)

coefficient of variation (cv)

*mm10\_introns\_from\_fusions.bed*

**2.4 Combine counts and prepare counts matrices**

Event Analysis requires coverage counts (junctions, exon fragments, exonic regions, etc.) as a set of wide-formatted data, where each coloum is an individual sample, and each row is an individual event (junction, exon fragment, etc.). A python script has been written that converts all single coverage count files in a folder (as generated by *rpkm\_calculate.py*) into one wide-formatted dataset.

**Note:** Junctions, exonic regions, exon fragments, and introns MUST all have their own, separate counts matrix. Sample names between each of these datasets should also match.

*2.4.1 Junction counts matrix*

**Program:** python (v2.7.6), import\_counts\_and\_convert\_to\_wide.py

**Description:** Combine junction coverage counts output from a series of single tall-formatted CSVs into a single wide-formatted counts matrix

**Input files:** Directory of junction coverage counts, as output from *rpkm\_calculate.py*

**Output files:** TSV file counts matrix, with columns as samples and rows as junction sequences

*2.4.2 Exonic region counts matrix*

**Program:** python (v2.7.6), import\_counts\_and\_convert\_to\_wide.py

**Description:** Combine exonic region coverage counts output from a series of single tall-formatted CSVs into a single wide-formatted counts matrix

**Input files:** Directory of exonic region coverage counts, as output from *rpkm\_calculate.py*

**Output files:** TSV file counts matrix, with columns as samples and rows as junction sequences

*2.4.3 Exon fragment counts matrix*

**Program:** python (v2.7.6), import\_counts\_and\_convert\_to\_wide.py

**Description:** Combine exon fragment coverage counts output from a series of single tall-formatted CSVs into a single wide-formatted counts matrix

**Input files:** Directory of exon fragment coverage counts, as output from *rpkm\_calculate.py*

**Output files:** TSV file counts matrix, with columns as samples and rows as junction sequences

*2.4.4 Intron counts matrix*

**Program:** python (v2.7.6), import\_counts\_and\_convert\_to\_wide.py

**Description:** Combine intron coverage counts output from a series of single tall-formatted CSVs into a single wide-formatted counts matrix

**Input files:** Directory of intron coverage counts, as output from *rpkm\_calculate.py*

**Output files:** TSV file counts matrix, with columns as samples and rows as junction sequences

**Phase 3: Events to transcripts, and transcript reduction**

The generation of annotations and the junction catalog involves four main steps:

a. Extraction of transcriptional events

b. Generation of exon-level annotations

c. Creation of the junction catalog

d. Creation of event indices

Examples on how to run the programs associated with these steps can be found in thethe makefile: *event\_analysis/run\_identifyTranscriptome.sh* (an example on how to run the makefile is also provided in *example\_running\_run\_identifyTranscriptome.sh).* Example output files using the RefSeq mm10 annotations are also listed in this document in parathesis in the “Output files” field

**Note:** the script *event\_analysis/run\_identifyTranscriptome.sh* also contains examples for Step 2.4 above.

**3.1 Event detection**

**Program:** flag\_event\_detection.py

**Description:** Flag events as on/off per group, using user-defined criteria. Input will be counts for fusions, fragments, introns and junctions. Additional inputs will be abundance cut-off and the minimum proportion of subjects within a group for which to consider an event as “detected” (e.g. more than 50% of samples in group1 where the APN>0).

**Note:** This should be run for each event type individually (junctions, exonic regions, exon fragments, OR introns)

**Input files:** Design file relating samples to treatment/condition groups

TSV file counts matrix from step 2.4 (junctions, exonic regions, exon fragments, OR introns)

**User defined parameters:**

Group variable name in design file

Minimum abundance (APN) cut-off for considering an event as “detected”

Minimum proportion of sample per group where event is “detected”, for the event to be considered “detected” in the group

**Output files:** Output TSV of events, with indicator flags for each group in design file. Indicator flags are coded as 0 or 1, to indicate that the event is considered detected (1) / not detected (0) in each group (junctions, exonic regions, exon fragments, OR introns)

**3.2 Event-level summaries**

**Program:** create\_event\_summaries.py

**Description: C**reate summaries for each event in the given input file. It takes a wide-formatted dataset of counts by event, annotations, detection flags, and a design file, and outputs a summary file, detailing group means, group detection, annotation frequency of events, transcripts and genes.

**Note:** This should be run for each event type individually (junctions, exonic regions, exon fragments, OR introns)

**Input files:** Design file relating samples to treatment/condition groups

TSV file counts matrix from step 2.4 (junctions, exonic regions, exon fragments, OR introns)

annotation file for appropriate event type

detection flags file for appropriate event type

**User defined parameters:**

Group variable name in design file

Minimum size (in bp) of event. Any event short than this will be eliminated from analysis. For exonic regions, exon fragments and introns, it is recommend this be set to 10bp. For junctions, it is recommended to set this to the read size.

**Output files:** Summary TSV file of events, with mean counts per group, detection flags and event annotations

**3.3 Identify expressed genes**

**Program:** identify\_expressed\_genes.py

**Description:** Creates an output table that flags whether a gene is expressed or not, and whether it has any associated exonic regions of genic ambiguity (multigene fusions).

**Input files:** Design file relating samples to treatment/condition groups, exonic region summary TSV from Step 3.2

**User defined parameters:**

Group variable name in design file

**Output files:** Summary TSV indication number of exonic regions per gene are detected in each group, and whether the gene is considered “expressed” in each group.

Output TSV has the following columns:

GeneID (gene\_id)

Number of exonic regions assoicated with gene (num\_exonic\_regions)

Number of exonic regions detected in group (num\_exonic\_regions\_detected\_\*, one for each group in design file)

Flag to indicate whether gene is considered expressed in group (flag\_gene\_expressed\_\*, one for each group in design file)

Flag to indicate whether a gene has an exonic region with ambiguous genic identity (i.e. coordinates of gene overlap those of another gene) (flag\_gene\_has\_multigene\_exon)

**3.4 Create intron-border junction summaries**

**Program:** create\_intron\_border\_summary.py

**Description:** creates an output table that summarizes introns, border junctions and adjacent exonic regions and classifies these events into possible novel donor/acceptor, intron retention, or unprocessed transcript, based on some simple rules. Only events from expressed genes are considered.

**Input files:** Design file relating samples to treatment/condition groups, exonic region summary TSV from Step 3.2, intron summary TSV from Step 3.2, junction summary TSV from Step 3.2, intron-border junction index from 1.4.2 (*mm10\_intron2border\_junction\_index.csv*)

**User defined parameters:**

Group variable name in design file

Minimum mean APN of donor/5' border junction for event to be considered supported

Minimum mean APN of acceptor/3' border junction for event to be considered supported

Minimum mean APN of intron for event to be considered supported

**Output files:** TSV summarizing intron-border junction classifications

The output TSV contains the following columns:

Index (index)

Intron ID (intron\_id)

Chromosome (chr)

Intron start position (intron\_start)

Intron stop position (intron\_stop)

Gene ID (gene\_id)

5' exonic region ID (exonic\_region\_id\_5prime)

5' border junction ID (border\_junction\_id\_5prime)

3' exonic region ID (exonic\_region\_id\_3prime)

3' border junction ID(border\_junction\_id\_3prime)

Mean APN of intron (mean\_apn\_intron)

Flag to indicate if intron is considered detected (flag\_intron\_detected)

Mean APN of 5' exonic region (mean\_apn\_5prime\_exon)

Flag to indicate if 5' exonic region is considered detected(flag\_5prime\_exon\_detected)

Mean APN of 3' exonic region (mean\_apn\_3prime\_exon)

Flag to indicate if 3' exonic region is considered detected (flag\_3prime\_exon\_detected)

Mean APN of 5' border junction (mean\_apn\_5prime\_border)

Flag to indicate if 5' border junction is considered detected (flag\_5prime\_border\_detected)

Mean APN of 3' border junction (mean\_apn\_3prime\_border)

Flag to indicate if 3' border junction is considered detected (flag\_3prime\_border\_detected)

Flag to indicate if intron event is resolvable (I.e exonic regions are detected) (flag\_resolvable)

Flag to indicate if there is evidence of a possible novel donor/5' site at the level of event detection (flag\_novel\_donor\_dtct)

Flag to indicate if there is evidence of a possible novel donor/5' site at the level of read support (flag\_novel\_donor\_support)

Flag to indicate if there is evidence of a possible novel acceptor/3' site at the level of event detection (flag\_novel\_acceptor\_dtct)

Flag to indicate if there is evidence of a possible novel acceptor/3' site at the level of read support (flag\_novel\_acceptor\_support)

Flag to indicate if there is evidence of a possible intron retention at the level of event detection (flag\_intron\_retention\_dtct)

Flag to indicate if there is evidence of a possible intron retention at the level of read support (flag\_intron\_retention\_support)

Event classification based on detection of events (event\_classification\_detection)

Event classification based on read support of events (event\_classification\_support)

Treatment group as listed in design file (treatment\_group)

**3.5 Summarize by transcript**

**Program:** summarize\_transcripts\_by\_group.py

**Description:** [text]

**Input files:** Input gene summary from step 3.3; exon fragment and junction summaries from step 3.2; event-to-transcript-to-gene index from step 1.4.1 (*summarize\_transcripts\_by\_group.py*); design file

**User defined parameters:**

Group variable name in design file

**Output files:** Summary TSV of transcript detection metrics

The output TSV contains the following columns:

Gene ID (gene\_id)

Transcript ID (transcript\_id)

Treatment group (treatment\_group)

Flag to indicate if gene has an exonic region with ambiguous genic identity (flag\_gene\_has\_multigene\_exon)

Number of exon fragments associated with transcript (num\_fragments)

Number of associated exon fragments detected (num\_detected\_fragments)

Proportion of fragments detected (perc\_fragments\_detected)

Number of unique exon fragments associated with transcript (num\_unique\_fragments)

Number of associated unique exon fragments detected (num\_detected\_unique\_fragments)

Proportion of unique fragments detected (perc\_unique\_fragments\_detected)

Number of junctions associated with transcript (num\_junctions)

Number of associated junctions detected (num\_detected\_junctions)

Proportion of junctions detected (perc\_junctions\_detected)

Number of unique junctions associated with transcript (num\_unique\_junctions)

Number of associated unique junctons detected (num\_detected\_unique\_junctions)

Proportion of unique junctions detected (perc\_unique\_junctions\_detected)

Number of events associated with transcript (num\_events)

Number of associated events detected (num\_detected\_events)

Proportion of events detected (perc\_events\_detected)

Number of unique events associated with transcript (num\_unique\_events)

Number of associated unique events detected (num\_detected\_unique\_events)

Proportion of unique events detected (perc\_unique\_events\_detected)