**README Notes on TnSeq R software**

* Software to process, quantify, display, and summarize TnSeq data.
* Written, as a stand-alone set of functions inside one source code file named “tnseq.R”, that leverage the DuffyTools package.
* This document will try to record the intent, implementation, and run-time procedure for processing Tn-seq datasets.

**TnSeq Raw Data, File Organization & Naming:**

Based on the initial set of test data, the follow assumptions are made about the raw data to be processed:

1. Every data file will be WIG format of {location, depth} pairs containing an identical set of transposon element insertion site locations. Depths will be non-negative read counts, with zero denoting no observed read events at that insertion site location. The WIG files were created using the TRANSIT tool (DeJesus 2015 PLoS Comp Bio).
2. All data files to be analyzed together will be located in one folder, and all data files in the folder will be processed together and treated as replicates of one or more experimental conditions.
3. Each file name will follow a pre-defined naming scheme that allows the software to infer all needed facts about each file. We will use the naming scheme of “<TimePoint>.<Replicate>\_<Condition>.wig” as implemented in function “loadSampleDetails()”, where TimePoint is one of {‘T1’,’T2’}, Replicate is a positive integer, and Condition is any character string excluding special delimiting characters {“ “, “.”, “\_”} etc.
4. Each new family of data sets will be kept in a separate folder, and each folder will be processed independently.
5. We will specify the genome for a dataset as a command line argument at runtime.

A few runtime tests about the file naming format have been implemented, to try to catch and prevent unintended file comparisons.

**TnSeq Data Processing:**

The data will be processed following the strategy of “van Opijnen, *Nature Methods* (2009)”. Each discrete processing step is implemented in separate R functions as outlined below, with any custom changes or modifications explicitly noted. There are a small set of pre-defined constants that effect runtime behavior, that are noted where appropriate. The full processing pipeline is implement by the top level R function:

do.all( path, speciesID=getCurrentSpecies(), offset.readCount=NULL,

nGenePlots=N\_HTML\_GENES, reload=FALSE)

**Step #1 - Read in all WIG data files:** R function “loadWigFileData(path)” reads up all the insertion site count WIG data files from one folder, validates the assumption about all files having identical site locations, and creates one data frame table of raw counts. Each WIG data file becomes a named column, each insertion site becomes a row, with extra feature columns for the gene ID, gene symbol, product description, and nucleotide location of each insertion site. This newly created counts table is written back to the same folder with name “RawCounts.csv”. All subsequent processing steps will operate on this counts table instead of the separate raw WIG files.

Modifications: The original methods make no allowance for the destabilizing effect of low total raw read counts in a file, which could strongly perturb the calculation of fitness. Implemented a sanity check for number of raw reads per file, with a flagging warning on files with too few reads. User settable threshold MIN\_COUNTS\_PER\_WIG defaults to 1 million raw reads minimum per file.

**Step #2 - Auto-creation of “SampleKey.csv” metadata file:** R function “loadSampleDetails(path)” builds a small file of metadata that will be used to control the runtime processing of all samples in the experiment. It extracts the sample names from the counts table created in step #1, deduces the time point / replicate / experiment name for each; and creates a placeholder for the user to later enter details such as: the culture expansion factor ( variable “d” in van Opijnen’s equation for calculation of fitness), the exclude/include calls, etc. The function creates a new file called “SampleKey.csv” in the same folder as the WIG files and the counts table. Subsequent calls to this function will reread this metadata file, to influence the run-time processing of all samples.

After initial creation of this file, the processing will halt to allow the user to hand enter the actual expansion factor values. Then rerun the top-level processing function “do.all()” to use the actual expansion factor values.

**Step #3 - Process the data to generate Fitness scores:** R function “processData()” is the main step for turning raw counts into fitness scores for every insertion site. It calls lower level functions to implement the details of calculating fitness metric “Wi” as outlined in the ‘methods online’ supplement to the van Opijnen paper. As parts of their methods descriptions are not fully self-explanatory, any comments/changes/modifications in how the processing is actually implemented are detailed below. For all steps below, any samples flagged for exclusion in the SampleKey.csv file are omitted from processing and ignored.

This function is given the entire matrix of counts as its input, but now implements the processing in an iterative fashion by operating on exactly 2 columns at a time – the T1 and T2 columns of one replicate of one condition. The steps below describe how this small set of exactly 2 columns of count data get processed.

**Step 3.1 - Drop low-count insertion sites:** This step will remove rows from the counts data, for insertion sites that have too few read counts to be reliable. Implemented in R function “dropLowCountInsertions()”. Because the fitness is a ratio of read counts, very low/zero counts will cause undefined or extreme ratios that are not biologically meaningful. Their methods state “*only insertions with eight or more reads at one time point were included in the analyses*”. But 2 problems arise here:

A) Semantic: they did not have biological replicates. By “*one time point*”, do they mean that as long as at least one sample has 8 reads – even if all other ‘N-1’ replicate samples have zero reads – that the insertion site should be kept? Doubtful.

b) Practical: These are raw read counts, before normalization, so the number of reads per sample may be highly variant. A static value like 8 means completely different things to a sample with 100,000 reads versus a sample with 10,000,000 reads. Further, their data was to a different organism with a different number of insertion sites, so the expected counts per site may be vastly different for MTb. Judging from their comment of observing a mean of 82 reads per site in their samples, we may want to instead use a self-normalizing threshold metric like “at least 10% of the mean count depth per site” for our cutoff for calling a site as having too few reads to be meaningful.

To partially address these uncertainties, I have added another method for dropping rows, based in percentages, that calculates the minimum number of reads separately for each column, based on that column’s mean count depth. This behavior is selected by argument “min.mode”=”percent”, and the “min.count” value will be interpreted as a percentage in the range 0 to100. Default behavior is “percent” mode with a cutoff of 10%.

Recent change: rather than actually dropping low/zero rows, we now just flag the rows for exclusion during later steps. This keeps the algorithm simpler, and allows each replicate to have its own control over which insertion sites get kept or ‘removed’. So the term “exclude” is more precise than “remove”.

**Step 3.2 - Normalize read counts:** This step will scale both columns in the counts data, to force identical total read counts. Implemented in R function “normalizeInsertionCounts()”. This step is essential, as the fitness ratio is intended to measure just change due to survival – not anything related to raw read counts per sample. Total read counts per sample are summed (after exclusion of low-count rows in 3.1 above), and the median total count is selected as the target total count per sample, that both samples will be scaled to. Their paper suggested only minor changes due to scaling, which seems unrealistic for real world data. Added a scaling magnitude test, to flag any samples that needed extreme scaling as possible candidates for exclusion.

**Step 3.3 - Calculate Fitness scores:** This step will generate the fitness scores “Wi” from the “TimePoint1 vs TimePoint2” change in read counts for each insertion site, for each replicate pair, over all experimental conditions in the counts table. Implemented in R function “calculateOneFitness()”. This function implements the details of calculating the term “Wi” as defined by van Opijnen, with the following modifications:

1. They make no provision for preventing division by zero, which occurs very frequently in our data. (This reiterates my concern in step 3.1 above about what they meant by “*with eight or more reads at one time point*”, as if all rows with any zero values were previously removed). To alleviate this math error, we add a small linear offset representing approximately ‘min.count’ reads to all values in the calculation of the ratio. This has minimal effect on large count sites, moderates the extreme ratios generated from low count sites, and prevents division by zero.

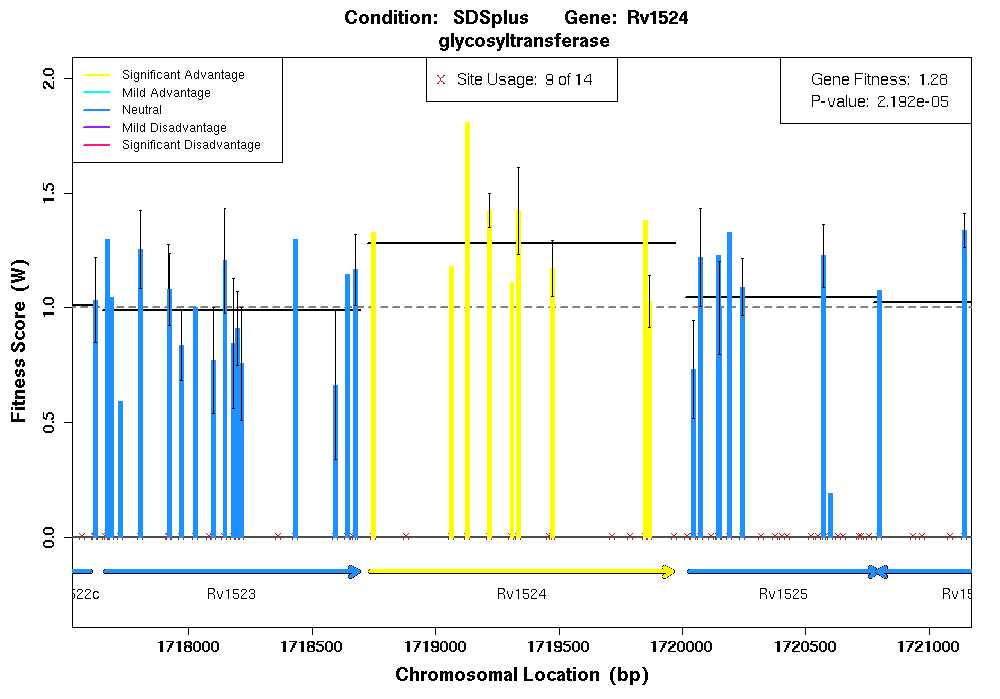
2) Global re-normalization of fitness values: they assert the use of a control set of genes known to never vary in fitness, to be used to re-normalize the calculated fitness scores to force these genes to have fitness of 1.0 exactly. We as of yet have no such control set. We could conceivably use the global set of all genes and re-normalize to push the median fitness to 1.0. We have implemented the ‘global renormalization’ method for now.

Recent change: we have included a more robust method to auto-select the optimal small linear offset. Given that no change in fitness is defined as 1.0, and values above or below 1.0 represent a change in fitness, and that the value of the small linear offset determines the magnitudes of the fitness score, we can in software find the offset that best satisfies a pre-set expectation of the range of fitness values. To that end, we define the expected range of fitness scores to be the interval zero to 2.0, with a center point of the ‘no change’ fitness score of 1.0. The tool will allow no more that 5% of all measured fitness scores to land outside the expected 0..2 range. So the tool starts with a read count offset of 0, calculates all fitness, evaluates how many values fall outside that range, and iteratively increase the read count offset in units of 5 reads, and re-measures fitness until it forces 95% of the fitness scores to be in the 0 to 2.0 range. This final offset is used, and printed as part of the runtime summary. In this way, each replicate of each condition can auto-determine the optimal offset for its read counts, expansion factor, etc.

The result of all fitness calculations is a new table with a row for each insertion site and a column for each replicate of each experiment. This table is written to the same folder as all other files, with a name of “FitScores.csv”. Any insertion sites dropped for low read counts will have NA for its fitness score.

**Step 4 - Gene Fitness and Visualization:** This step will generate the final fitness score for each gene for each condition, as an average over all replicates and insertion sites for each gene, and is implemented by R function “calculateGeneFitness()”. It operates on the table of fitness scores from step #3. For the columns from each condition, it will take all insertion site rows for each gene, and calculate the average fitness for all sites/replicates that have fitness scores generated. Average method is user-settable, current default is ‘median’. The result is a new file for each condition called “<condition>.GeneFitness.csv” and written to the same folder. Each row of the file has several metrics for each gene, including: total number of non-NA sites; mean number of non-NA sites per replicate; total number of non-NA fitness scores; the average fitness score for the gene; the standard deviation of the fitness scores; and a P-value of the 1-sample T.test that the set of values is not the null hypothesis fitness score of 1.0.

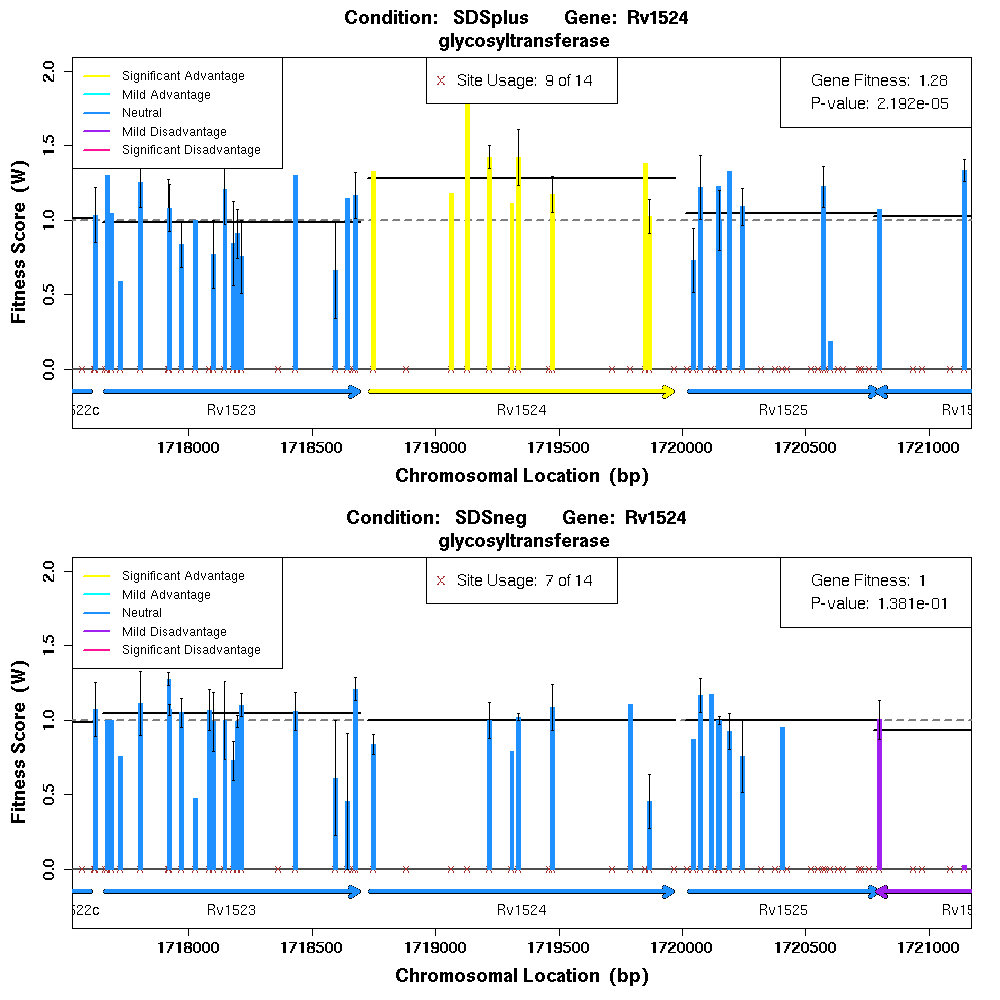
For visualization, we also take this file of all genes and create 2 small HTML files that highlight the top N (default=100) genes that have the largest change in fitness. Their file names contain “UP.Gene.html” and “DOWN.Genes.html” respectively. The HTML files also have hyperlinks to gene plot images that try to convey the most important features about the gene’s fitness. An example image is shown and explained below:



Each small brown “x” is an insertion site as defined in the WIG files. Sites with at least 1 non-NA fitness score among the replicates are drawn as colored vertical bars with black standard error bars. All sites from all replicates landing inside the gene boundaries are averaged to give a single value for the gene’s fitness, drawn as a solid black horizontal line. Colors reflect the gene’s overall fitness level: pink for reduced fitness, green for no net fitness change, blue for modest increase, and yellow for major fitness increase, brown for sites in intergenic regions. Implemented by function “visualizeGeneFitness()”

Recent change: changing the coloring schemes and thresholds to better convey up/down and significance.

**Step 5 - Condition Comparison and Visualization:** This final step will compare the difference in each gene’s net fitness between any 2 conditions, to find genes with the most significant difference in fitness as caused by the difference in experimental condition. Pereviously implemented by just comparing the gene level results. As of 2019, using the more precise method of using a 2-sample T.test of all the underlying fitness measurements from both conditions. Implemented by R function “compareConditionFitness()”. Generates a file for each condition versus condition 2-way comparison, called “DeltaFit.<condition1>.v.<condition2>.csv”. Separate HTML files of the top N most different genes for each comparison are generated, with hyperlinked plot images that show each gene in both condtions simultaneously.



Materials & Methods blurb:

**Fitness calculation**

Insertion fitness calculations followed that of van Opijnen *Nat Method*s 2009, with minor modifications. Briefly, for each pair of time points (t1 & t2), insertion site read counts were converted to fitness scores Wi by a multistep process. First, insertions with very low read counts were excluded. Rather than use a fixed exclusion cutoff of 8 reads, a similar but more robust approach was to select 10% of the average read count as the cutoff value for that time point. Second, normalization to equalize total reads per time point was done. Most normalization factors were very close to 1.0, and if any exceeded 2.0 that replicate pair was excluded. Third, normalized counts were converted to frequencies Ni(t1) and Ni(t2) for calculating the fitness ratios Wi. Lastly, rather than selecting an explicit set of control genes with no fitness effect for final normalization, all fitness values were normalized such that the median fitness value was exactly 1.

Since ratios of very small frequencies can result in extreme fitness values, and low total read counts per time point affects the precision of the frequencies, two modifications to the fitness calculation were added.

1. A small offset term (the smallest non-zero value from all Ni from either time point) was added to both Ni terms before division, making all fitness ratios somewhat closer to 1.0 than if no offset was used, having the largest impact on the smallest frequencies.

2. The set of all fitness ratios should be normally distributed with a mean of 1.0 and extreme ratios (Wi values above 2.0 or below 0) should be very rare. If more than 5% of the fitness ratios were extreme, it was deemed to be an artifact due to low total read counts. In this case, a raw read count offset of 5 reads was added to all non-excluded insertion sites in both time points and the fitness calculation was repeated, until at least 95% of all Wi land in the expected fitness interval 0 to 2.

The net effect of these two modifications is to force all fitness ratios somewhat closer to 1.0 to minimize the occurance of false positives, at the risk of increasing false negatives.

Final fitness calls for each gene were generated by combining all replicate experiments and all insertion sites located within the gene’s boundary. The gene fitness score Wg was calculated as the median of all insertion fitness Wi. Significance was calculated using the one sample T test with expected mean of 1.0. Change in fitness calls between two experimental conditions were generated by taking the difference of the median fitness from each condition, with significance calculated from the two sample T test using all insertion fitness Wi from each condition.