transit Documentation

Release v3.2.0

Michael A. DeJesus

Oct 26, 2020
This page contains the documentation for TRANSIT. Below are a few quick links to some of the most important sections of the documentation, followed by a brief overview of TRANSIT’s features.
Quick Links

- Installation
- TRANSIT Manual
- tutorial-link
- TPP Manual
- Code Documentation
CHAPTER 2

Features

TRANSIT offers a variety of features including:

- More than 8 analysis methods, including methods for determining conditional essentiality as well as genetic interactions.
- Ability to analyze himar1 or tn5 transposons datasets.
- TrackView to help visualize read-counts across the genome.
- Can export datasets into a variety of formats, including IGV.
- Includes a variety of normalization methods.
- Quality Control diagnostics, to identify poor quality datasets.
- Ability to install as a python package, to import and use in your own personal scripts.

2.1 TRANSIT Overview

- This is a software that can be used to analyze Tn-Seq datasets. It includes various statistical calculations of essentiality of genes or genomic regions (including conditional essentiality between 2 conditions). These methods were developed and tested as a collaboration between the Sassetti lab (UMass) and the Ioerger lab (Texas A&M) [DeJesus2015TRANSIT].
• TRANSIT is capable of analyzing TnSeq libraries constructed with Himar1 or Tn5 datasets.

• TRANSIT assumes you have already done pre-processing of raw sequencing files (.fastq) and extracted read counts into a .wig formatted file. The .wig file should contain the counts at all sites where an insertion could take place (including sites with no reads). For Himar1 datasets this is all TA sites in the genome. For Tn5 datasets this would be all nucleotides in the genome.

• Note that while refer to “read-counts” throughout the documentation, the current Himar1 protocol utilizes internal barcodes that can be used to reduce raw read counts to unique template counts, and this this is the intended input to TRANSIT from Himar1 datasets.
• There are various methods available for pre-processing (converting .fastq files to .wig files). You might have your own scripts (if so, massage the data into .wig format), or you might get the scripts used in the Sassetti lab. For convenience, we are including a separate tool called TPP (Tn-Seq Pre-Processor) with this distribution that encodes the way we process .fastq files in the Ioerger lab. It’s a complicated process with many steps (removing transposon prefixes of reads, mapping into genome, identifying barcodes and reducing read counts to template counts).

• Most of the analysis methods in TRANSIT require an annotation to know the gene coordinates and names. This is the top file input in the GUI window. The annotation has to be in a somewhat non-standard format called a “.prot_table”. If you know what you are doing, it is easy to convert annotations for other organisms into .prot_table format. But for convenience, we are distributing the prot_tables for 3 common versions of the H37Rv genome: H37Rv.prot_table (NC_000962.2, from Stewart Cole), H37RvMA2.prot_table (sequenced version from the Sassetti lab), and H37RvBD.prot_table (sequenced by the Broad Institute). All of these are slightly different, and it is critical that you use the same annotation file as the reference genome sequence used for mapping the reads (during pre-processing).

• There are three main types of essentiality analyses: individual, comparative (pairwise), and multi-condition.
  – In individual analysis, the goal is to distinguish essential vs. non-essential in a single growth condition, and to assess the statistical significance of these calls. Two methods for this are the Gumbel method and the HMM. They are computationally distinct. The Gumbel method is looking for significant stretches of TA sites lacking insertions, whereas the HMM looks for regions where the mean read count is locally suppressed or increased. The HMM can detect ‘growth-advantaged’ and ‘growth-defect’ regions. The HMM is also a bit more robust on low-density datasets (with insertion density as low as 20-30%). But both methods have their merits and are complementary.

• For comparative analysis, the goal is to determine if the sum of read counts differs significantly between two conditions, for which TRANSIT uses resampling (a non-parameteric test analogous to a permutation test). Hence this can be used to identify conditionally essential regions and quantify the statistical significance. A rank-based Mann-Whitney U-test is also available.

• For multi-condition analysis, there are two methods for determining whether insertion counts in a gene vary significantly across conditions: ZINB (Zero-Inflated Negative Binomial) regression, and ANOVA. In general, we find that ZINB finds more (and better) hits than ANOVA (and even out-performs resampling, for cases with 2 conditions). Furthermore, ZINB can incorporate additional covariates.
• TRANSIT has been designed to handle multiple replicates. If you have two or more replicate dataset of the same library selected in the same condition, you can provide them, and more of the computational methods will do something reasonable with them.

• For those methods that generate p-values, we often also calculate adjusted p-value (or 'q-values') which are corrected for multiple tests typically the Benjamini-Hochberg procedure. A typical threshold for significance would be q<0.05 (not p<0.05).

• It is important to understand the GUI model that TRANSIT uses. It allows you to load up datasets (.wig files), select them, choose an analysis method, set parameters, and start the computation. It will generate output files in your local directory with the results. These files can then be loaded into the interface and browser with custom displays and graphs. The interface has 3 main windows or sections: ‘Control Samples’, ‘Experimental Samples’, ‘Results Files.’ The first two are for loading input files (‘Control Samples’ would be like replicate datasets from a reference condition, like in vitro, rich media, etc.; ‘Experimental Samples’ would be where you would load replicates for a comparative conditions, like in vivo, or minimal media, or low-iron, etc.) The ‘Results Files’ section is initially empty, but after a computation finishes, it will automatically be populated with the corresponding output file. See the ‘Tutorial’ section below in this documentation for an illustration of the overall process for a typical work-flow.

• TRANSIT incorporates many interesting ways of looking at your data.

• Track view shows you a visual representation of the read counts at each site at a locus of interest (for selected datasets) somewhat like IGV.
• Scatter plots can show the correlation of counts between 2 datasets.
Volcano plots can be used to visualize the results of resampling and assess the distribution between over- and under-represented genes in condition B vs. condition A. In addition, you can look at histogram of the resampling distributions for each gene.
2.1. TRANSIT Overview

Instructions:
1. Choose the annotation file ("prot table") that corresponds to the datasets to be analyzed.
2. Add the desired Control and Experimental datasets.
3. (Optional) If you wish to visualize their read counts, select the desired datasets and click on the "View" button.
4. Select the desired analysis method from the dropdown menu on the top-right of the window, and follow its instructions.

Figure 1

[Graph showing a volcano plot]
Most of the methods take a few minutes to run. (it depends on parameters, CPU clock speed, etc., but the point is, a) these calculations are complex and not instaneous, but b) we have tried to implement it so that they don’t take hours)

Note: in this version of TRANSIT, most of the methods are oriented toward gene-level analysis. There are methods for analyzing essentiality of arbitrary genomic regions (e.g. sliding windows, HMMs…). We plan to incorporate some of these in future versions.

2.1.1 Developers

<table>
<thead>
<tr>
<th>Name</th>
<th>Time Active</th>
<th>Contact Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thomas R. Ioerger</td>
<td>2015-Present</td>
<td><a href="http://faculty.cs.tamu.edu/ioerger/">http://faculty.cs.tamu.edu/ioerger/</a></td>
</tr>
<tr>
<td>Michael A. DeJesus</td>
<td>2015-2018</td>
<td><a href="http://mad-lab.org">http://mad-lab.org</a></td>
</tr>
<tr>
<td>Chaitra Ambadipudi</td>
<td>2015</td>
<td></td>
</tr>
<tr>
<td>Eric Nelson</td>
<td>2016</td>
<td></td>
</tr>
<tr>
<td>Siddharth Subramaniyam</td>
<td>2018</td>
<td></td>
</tr>
</tbody>
</table>

2.1.2 References

If you use TRANSIT, please cite the following reference:
Development of TRANSIT is funded by the National Institutes of Health (www.nih.gov/) grant U19 AI107774. Other references, including methods utilized by TRANSIT:

2.2 Installation

TRANSIT can be downloaded from the public GitHub server, http://github.com/mad-lab/transit. It is released under a GPL License. An archive with the lastest version fo the source code can be downloaded at the following link:

Source code.zip

If you know how to utilize git, you can clone the git repository as follows:

```bash
git clone https://github.com/mad-lab/transit/
```

TRANSIT is python-based You must have python installed (installed by default on most systems). In addition, TRANSIT relies on some python packages/libraries/modules that you might need to install (see Requirements).

If you encounter problems, please contact us or head to the Troubleshooting section.

2.2.1 Requirements

TRANSIT runs on both python2.7 and python3. But the dependencies vary slightly.

2.2.2 Python 2.7:

The following libraries/modules are required to run TRANSIT:

- Python 2.7
- Numpy (tested on 1.15.0)
- Statsmodels (tested on 0.9.0)
- Scipy (tested on 1.1)
- matplotlib (tested on 2.2)
- Pillow 5.0
- wxpython 4+
- PyPubSub 3.3 (Version 4.0 does not support python2 Github Issue)

All of these dependencies can be installed using the following command.

```bash
pip install numpy scipy pillow "pypubsub<4.0" "matplotlib<3.0" statsmodels wxPython
```

Pip and Python are usually preinstalled in most modern operating systems.
2.2.3 Python 3:

The following libraries/modules are required to run TRANSIT:

- Python 3+
- Numpy (tested on 1.16.0)
- Statsmodels (tested on 0.9.0)
- Scipy (tested on 1.2)
- matplotlib (tested on 3.0)
- Pillow 6.0
- wxpython 4+
- PyPubSub 4+ (tested on 4.0.3)

All of these dependencies can be installed using the following command.

```
pip3 install numpy scipy pillow pypubsub matplotlib statsmodels wxPython
```

Pip and Python are usually preinstalled in most modern operating systems.

Additional Requirements: R (statistical analysis package)

R is called by Transit for certain commands, such as ZINB, corrplot, and heatmap. As of now, installing R is optional, and requires these additional steps...

Additional Installation Requirements for R:

- install R (tested on v3.5.2)
- R packages: MASS, pscl, corrplot, gplots (run “install.packages(MASS)” etc. in R console)
- Python packages (for python3): rpy2 (v>=3.0) (run “pip3 install rpy2” on command line)
- Python packages (for python2.7): rpy2 (v<2.9.0) (run “pip install ‘rpy2<2.9.0’ ” on command line)

2.2.4 Use as a Python Package

TRANSIT can be (optionally) installed as a python package. This can simplify the installation process as it will automatically install most of the requirements. In addition, it will allow users to use some of transit functions in their own scripts if they desire. Below is a brief example of importing transit functions into python. In this example, pair of .wig files are parsed into their read-counts (data) and genomic positions (position), and then normalization factors are calculated. See the documentation of the package for further examples:

```
>>> import pytransit.norm_tools as norm_tools
>>> import pytransit.tnseq_tools as tnseq_tools

>>> (data, position) = tnseq_tools.get_data("transit/data/glycerol_H37Rv_rep1.wig", 
..."transit/data/glycerol_H37Rv_rep2.wig")

>>> print(data)
array([[ 0., 0., 0., ..., 0., 0., 0.],
[ 0., 0., 0., ..., 0., 0., 0.]])
```

(continues on next page)
You can use pip to install the TRANSIT package.

```
sudo pip install tnseq-transit
```

This will automatically download and install TRANSIT as a package, and all remaining required python packages. Once TRANSIT is installed as a package, it can be executed as

**Note:** If you will be using the pre-processor, TPP, you will also need to install install BWA.

**Note:** The Transit package does not install wxPython. For graphical interface usage, this has to be done by the user. See install wxPython

## 2.2.5 Optional: Install BWA to use with TPP pre-processor

If you will be using the pre-processor, TPP, you will also need to install BWA.

### Linux & OSX Instructions

Download the source files:


Extract the files:

```
tar -xvjf bwa-0.7.12.tar.bz2
```

Go to the directory with the extracted source-code, and run make to create the executable files:

```
cd bwa-0.7.12
make
```

### Windows Instructions

For Windows, we provide a windows executable (.exe) for Windows 64 bit:

- [bwa-0.7.12_windows.zip](http://sourceforge.net/projects/bio-bwa/files/)

The 32-bit version of Windows is not recommended as it is limited in the amount of system memory that can be used.
2.2.6 Upgrading

The process of upgrading transit will depend on how you installed transit initially.

**Method 1: Upgrading package installation**

If you installed TRANSIT as a package, then to upgrade, simply use pip to install tnseq-transit again, but this time include the ‘--upgrade’ flag. For example:

```
sudo pip install tnseq-transit --upgrade
```

This will automatically download and install the latest version of TRANSIT, as well as upgrade any of its requirements if necessary for compatibility.

**Method 2: Upgrading source installation**

If you installed TRANSIT by downloading the raw source, then you can upgrade TRANSIT simply by replacing the old source code with the latest version. You can obtain a .zip archive with the latest version of the source through the following link:

[https://github.com/mad-lab/transit/archive/master.zip](https://github.com/mad-lab/transit/archive/master.zip)

Simply extract the code, and replace your existing files or delete the directory with the old source doe and use the newest version.

---

**Note:** If an an older version of wxPython is already installed (< 4.0), you may have to remove it and install version 4.0+.

---

2.2.7 Installing wxPython

wxPython 4+ can be installed using pip

```
pip install wxPython
```

If the above command fails and you already have wxPython < 4.0 installed, you may have to manually remove it. See [https://stackoverflow.com/questions/50688630/cannot-uninstall-wxpython-3-0-2-0-macos](https://stackoverflow.com/questions/50688630/cannot-uninstall-wxpython-3-0-2-0-macos) for details.

2.2.8 Troubleshooting

1. No window appears when running in GUI mode.

This problem is likely due to running OSX and previouslyUnsupported versions of matplotlib. Please upgrade matplotlib to the latest version using:
pip install 'matplotlib' --upgrade

2. pip: SystemError: Cannot compile ‘Python.h’.

This occurs when you do not have the development libraries for python. You can fix this by installing the python-dev packages:

sudo apt-get install python-dev

3. pip: “The following required packages can not be built: freetype, png,” etc.

This occurs when you do not have some dependencies that are necessary to build some of the python modules TRANSIT requires (usually matplotlib). Installing the following linux dependencies should fix this:

sudo apt-get install libpng-dev libjpeg8-dev libfreetype6-dev

4. pip: “No lapack/blas resources found”

This occurs when you do not have some dependencies that are necessary to build some of the python modules TRANSIT requires (usually numpy/scipy). Installing the following linux dependencies should fix this:

sudo apt-get install libblas-dev liblapack-dev libatlas-base-dev gfortran

5. “resources.ContextualVersionConflict (six 1.5.2)…”

This occurs some of the python modules are out of date. You can use pip to upgrade them as follows:

sudo pip install six --upgrade

2.3 Running TRANSIT

TRANSIT can be run in either GUI mode or in console mode. GUI Mode will be run if TRANSIT is not given any command-line arguments. If any arguments are given, TRANSIT will run in console-mode.

The exact commands will vary depending on the method of installation. Details are given below:
2.3.1 GUI Mode

In general, if you installed TRANSIT as a python package (e.g. using `pip install tnseq-transit`), then the proper way to run TRANSIT in GUI mode is simply to type the following into a console:

```
transit
```

**Note:** In windows, you will likely have to navigate to C:\Python2.7\Scripts to be able to recognize the transit.exe file.

If, however, you installed transit by downloading and extracting the source-code archive, you can run TRANSIT in GUI mode by typing in the command line:

```
python PATH/src/transit.py
```

where PATH is the path to the TRANSIT installation directory. You might be able to double-click on icon for transit.py, if your OS associates .py files with python and automatically runs them.

**Note:** Note, because TRANSIT has a graphical user interface, if you are trying to run TRANSIT in GUI mode across a network, for example by running on a unix server but displaying on a desktop machine, you will probably need to use `ssh -Y` and a local X11 client (like Xming or Cygwin/X on PCs). This will allow the GUI component to be properly displayed across the network connection.

2.3.2 Command line Mode

TRANSIT can also be run purely the command line, without a GUI interface. This is convenient if you want to run many analyses in batch, as you can write a script that automatically runs several analyses in parallel or in sequence.

If you installed TRANSIT as a python package, you can get a list of possible arguments by typing:

```
transit -h
```

Or if you installed it by downloading and extracting an archive with the source code:

```
python PATH/src/transit.py -h
```

In most cases TRANSIT expects the user to specify which analysis method they wish to run as their first argument. The user will need to type the short-name of the analysis method desired, e.g. “gumbel”, “hmm”, or “resampling”. By choosing a method, and adding the “-h” flag, you will get a list of all the necessary parameters and optional flags for the chosen method.

If you installed TRANSIT as a python package, you can achieve this by typing:

```
transit gumbel -h
```

Or if you installed it by downloading and extracting an archive with the source code:
See example usages of supported methods in *Analysis Methods* section.

### 2.3.3 Prot_tables (Annotations)

Most of the methods in Transit use a custom format for genome annotations called a ‘.prot_table’. It is a simple tab-separated text file with specific columns, as originally defined for genomes in Genbank many years ago.

The required columns are:

1. gene function description
2. start coordinate
3. end coordinate
4. strand
5. length of protein product (in amino acids)
6. don’t care
7. don’t care
8. gene name (like “dnaA”)
9. ORF id (like Rv0001)

It is crucial to use the same .prot_table corresponding to the genome sequence that was used to generate the wig file (count insertions) by TPP. This is because the coordinates of TA sites in the wig file and the coordinates of ORF boundaries must use the same coordinate system (which can be thrown out of register by indels).

Suppose you have a .prot_table for genome A, and you want to map reads to another genome B which is closely related, but for which you do not have an annotation. You can use the following web-app (Prot_table Adjustment Tool) to convert the annotation for A to B by adjusting all the coordinates of ORFs from A to B according to a genome alignment. For example, you could use this to map known ORFs in H37Rv to sequences of other strains, like HN878 or CDC1551. (Even though they have their own annotations, it might be helpful to use the genes as defined in H37Rv)

While some Transit methods can also work with .gff (or .gff3) files, the flexibility of the .gff format makes it difficult to anticipate all possible encoding schemes. Therefore, to simplify things, we recommend you convert your .gff file to .prot_table format once at the beginning and then use that for all work with Transit, which can be done through the GUI (under ‘Convert’ in menu), or on the command-line as follows:

```bash
> python transit.py convert gff_to_prot_table <.gff> <.prot_table>
```
2.3.4 Tn5 Datasets

Transit can now process and analyze Tn5 datasets. This is a different transposon than Himar1. The major difference is Tn5 can insert at any site in the genome, and is not restricted to TA dinucleotides (and saturation is typically much lower). This affects the statistical analyses (which were originally designed for Himar1 and can’t directly be applied to Tn5). Therefore, Resampling was extended to handle Tn5 for comparative analysis, and Tn5Gaps is a new statistical model for identifying essential genes in single Tn5 datasets. Amplification of Tn5 libraries uses different primers, and this affects the pre-processing by TPP. But TPP has been modified to recognize the primer sequence for the most widely used protocol for Tn5. Furthermore, TPP now has an option for users to define their own primer sequences, if they use a different sample prep protocol.

2.4 Quality Control

TRANSIT has several useful features to help inspect the quality of datasets as and export them to different formats. (see also TPP Statistics)

As you add datasets to the control or experimental sections, TRANSIT automatically provides some metrics like density, average, read-counts and max read-count to give you an idea of how the quality of the dataset.

However, TRANSIT provides more in-depth statistics in the Quality Control window. To use this feature, add the annotation file for your organism (in .prot_table or GFF3 format). Next, add and highlight/select the desired read-count datasets in .wig format. Finally, click on View -> Quality Control. This will open up a new window containing a table of metrics for the datasets as well as figures corresponding to whatever dataset is currently highlighted.

2.4.1 QC Metrics Table

The Quality Control window contains a table of the datasets and metrics, similar to the one in the main TRANSIT interface. This table has an extended set of metrics to provide a better picture of the quality of the datasets:
<table>
<thead>
<tr>
<th>Column Header</th>
<th>Column Definition</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>Name of dataset file.</td>
<td></td>
</tr>
<tr>
<td>Density</td>
<td>Fraction of sites with insertions.</td>
<td>“Well saturated” Himar1 datasets have &gt;30% saturation. Beneath this, statistical methods may have trouble.</td>
</tr>
<tr>
<td>Mean Read</td>
<td>Average read-count, including empty sites.</td>
<td></td>
</tr>
<tr>
<td>NZMean Read</td>
<td>Average read-count, excluding empty sites.</td>
<td>A value between 30-200 is usually good for Himar1 datasets. Too high or too low can indicate problems.</td>
</tr>
<tr>
<td>NZMedian Read</td>
<td>Median read-count, excluding empty sites.</td>
<td>As read-counts can often have spikes, median serves as a good robust estimate.</td>
</tr>
<tr>
<td>Max Read</td>
<td>Largest read-count in the dataset.</td>
<td>Useful to determine whether there are outliers/spikes, which may indicate sequencing issues.</td>
</tr>
<tr>
<td>Total Reads</td>
<td>Sum of total read-counts in the dataset.</td>
<td>Indicates how much sequencing material was obtained. Typically &gt;1M reads is desired for Himar1 datasets.</td>
</tr>
<tr>
<td>Skew</td>
<td>Skew of read-counts in the dataset.</td>
<td>Large skew may indicate issues with a dataset. Typically a skew &lt; 50 is desired. May be higher when library is under strong selection</td>
</tr>
<tr>
<td>Kurtosis</td>
<td>Kurtosis of the read-counts in the dataset.</td>
<td></td>
</tr>
</tbody>
</table>

### 2.4.2 QC Plots

The Quality Control window also contains several plots that are helpful to visualize the quality of the datasets. These plots are unique to the dataset selected in the Metrics Table (below the figures). They will update depending on which row in the Metrics Table is selected:
The first plot in the Quality Control window is a histogram of the non-zero read-counts in the selected dataset. While read-counts are not truly geometrically distributed, “well-behaved” datasets often look “Geometric-like”, i.e. low counts are more frequent than very large counts. Datasets which where this is not the case may reflect a problem.
The second plot in the Quality Control window is a quantile-quantile plot ("QQ plot") of the non-zero read-counts in the selected dataset, versus a theoretical geometric distribution fit on these read-counts. While read-counts are not truly geometrically distributed, the geometric distribution (a special case of the Negative Binomial distribution), can serve as a quick comparison to see how well-behaved the datasets are.

As the read-counts are not truly geometric, some curvature in the QQplot is expected. However, if the plot curves strongly from the identity line (y=x) then the read-counts may be highly skewed. In this case, using the "betageom" normalization option when doing statistical analyses may be a good idea as it is helpful in correcting the skew.
The second plot in the Quality Control window is a plot of the read-counts in sorted order. This may be helpful in indentifying outliers that may exist in the dataset. Typically, some large counts are expected and some normalization methods, like TTR, are robust to such outliers. However, too many outliers, or one single outlier that is overwhelmingly different than the rest may indicate an issue like PCR amplification (especially in libraries constructed older protocols).

2.4.3 Interpretation of Data Quality

It is important to be able to evaluate the quality of datasets. In a nutshell, we look at statistics like saturation, and mean read count, but also things like max count and skewness.

There are two ways to do QC in Transit - via the GUI and command-line. In the GUI, one can load a set of wig files a select “View->Quality Control” in the menu; this will display some plots of read-count distribution. Ideally, you want most of your datasets to fall along the diagonal on a QQ-plot. Real data will often deviate somewhat (I will try to be
more quantitative about this in the future), but if a dataset skews far off from the diagonal, it could cause problems with analytical methods like resampling or the HMM.

You can also generate the same table to statistics as on the QC panel from the command-line using the `tnseq_stats` command.

Below the plots are a table of statistics. While there are not rigorous criteria for defining “bad” datasets, rules of thumb I use for “good” datasets are: density>30% (ideally >50%) and NZmean>10 (ideally >50). In addition, I look at MaxReadCount and Skewness as indicators. Typically, MaxReadCount will be in the range of a few thousand to tens-of-thousands. If you see individual sites with counts in the range of $10^5$-$10^6$, it might mean you have some positive selection at a site (e.g. biological, or due to PCR jackpotting), and this can have the effect of reducing counts and influencing the distribution at all the other sites. If MaxReadCount<100, that is also probably problematic (either not enough reads, or possibly skewing). Also, skewness>30 often (but not always) signals a problem. Kurtosis doesn’t seem to be very meaningful. The reason it is not easy to boil all these down to a simple set of criteria is that some of the metrics interact with each other.

### 2.4.4 Beta-Geometric Correction

If you have a “bad” or poorly-behaving or “skewed” dataset (e.g. with mostly low counts, dominated by a few high counts), right now the only remedy you can try is applying the Beta-Geometric correction (BGC), which is a non-linear adjustment to the insertion counts in a wig file to make them more like an ideal Geometric distribution (DeJesus & Ioerger, 2016). (Note, all the other normalizations, like TTR, are linear adjustments, and so they can’t correct for skewing.)

In the GUI, when you are looking, you can change the normalization (e.g. from TTR to betageom) using the dropdown. Be aware that the Beta-Geometric normalization is compute-intensive and might take few minutes.

If it looks like it might help (i.e. if the QQ-plot fits the diagonal better using BG normalization), you can created BG-corrected versions of individual wig files by exporting them using the `normalize command` on the command-line with ‘-n betageom’ to specify normalization.
2.5 Analysis Methods

TRANSIT has analysis methods capable of analyzing Himar1 and Tn5 datasets. Below is a description of some of the methods.

2.5.1 Gumbel

The Gumbel can be used to determine which genes are essential in a single condition. It does a gene-by-gene analysis of the insertions at TA sites with each gene, makes a call based on the longest consecutive sequence of TA sites without insertion in the genes, calculates the probability of this using a Bayesian model.

**Note:** Intended only for Himar1 datasets.

**How does it work?**

For a formal description of how this method works, see our paper [DeJesus2013]:


**Example**

```bash
python3 transit.py gumbel <comma-separated .wig files> <annotation .prot_table or GFF3> <output file> [Optional Arguments]
```

Optional Arguments:
- `-s <integer>` := Number of samples. Default: `-s 10000`
- `-b <integer>` := Number of Burn-in samples. Default: `-b 500`
- `-m <integer>` := Smallest read-count to consider. Default: `-m 1`
- `-t <integer>` := Trims all but every t-th value. Default: `-t 1`
- `-r <string>` := How to handle replicates. Sum or Mean. Default: `-r Sum`
- `-iN <float>` := Ignore TAs occurring at given percentage (as integer) of the N terminus. Default: `-iN 0`
- `-iC <float>` := Ignore TAs occurring at given percentage (as integer) of the C terminus. Default: `-iC 0`

**Parameters**

- **Samples:** Gumbel uses Metropolis-Hastings (MH) to generate samples of posterior distributions. The default setting is to run the simulation for 10,000 iterations. This is usually enough to assure convergence of the sampler and to provide accurate estimates of posterior probabilities. Less iterations may work, but at the risk of lower accuracy.

- **Burn-In:** Because the MH sampler may not have stabilized in the first few iterations, a “burn-in” period is defined. Samples obtained in this “burn-in” period are discarded, and do not count towards estimates.
• **Trim**: The MH sampler produces Markov samples that are correlated. This parameter dictates how many samples must be attempted for every sampled obtained. Increasing this parameter will decrease the auto-correlation, at the cost of dramatically increasing the run-time. For most situations, this parameter should be left at the default of “1”.

• **Minimum Read**: The minimum read count that is considered a true read. Because the Gumbel method depends on determining gaps of TA sites lacking insertions, it may be susceptible to spurious reads (e.g. errors). The default value of 1 will consider all reads as true reads. A value of 2, for example, will ignore read counts of 1.

• **Replicates**: Determines how to deal with replicates by averaging the read-counts or summing read counts across datasets. This should not have an affect for the Gumbel method, aside from potentially affecting spurious reads.

**Outputs and diagnostics**

The Gumbel method generates a tab-separated output file at the location chosen by the user. This file will automatically be loaded into the Results Files section of the GUI, allowing you to display it as a table. Alternatively, the file can be opened in a spreadsheet software like Excel as a tab-separated file. The columns of the output file are defined as follows:

<table>
<thead>
<tr>
<th>Column Header</th>
<th>Column Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORF</td>
<td>Gene ID.</td>
</tr>
<tr>
<td>Name</td>
<td>Name of the gene.</td>
</tr>
<tr>
<td>Description</td>
<td>Gene description.</td>
</tr>
<tr>
<td>k</td>
<td>Number of Transposon Insertions Observed within the ORF.</td>
</tr>
<tr>
<td>n</td>
<td>Total Number of TA dinucleotides within the ORF.</td>
</tr>
<tr>
<td>r</td>
<td>Length of the Maximum Run of Non-Insertions observed.</td>
</tr>
<tr>
<td>s</td>
<td>Span of nucleotides for the Maximum Run of Non-Insertions.</td>
</tr>
<tr>
<td>zbar</td>
<td>Posterior Probability of Essentiality.</td>
</tr>
<tr>
<td>Call</td>
<td>Essentiality call for the gene. Depends on FDR corrected thresholds. E=Essential U=Uncertain, NE=Non-Essential, S=too short</td>
</tr>
</tbody>
</table>

Note: Technically, Bayesian models are used to calculate posterior probabilities, not p-values (which is a concept associated with the frequentist framework). However, we have implemented a method for computing the approximate false-discovery rate (FDR) that serves a similar purpose. This determines a threshold for significance on the posterior probabilities that is corrected for multiple tests. The actual thresholds used are reported in the headers of the output file (and are near 1 for essentials and near 0 for non-essentials). There can be many genes that score between the two thresholds (t1 < zbar < t2). This reflects intrinsic uncertainty associated with either low read counts, sparse insertion density, or small genes. If the insertion_density is too low (< ~30%), the method may not work as well, and might indicate an unusually large number of Uncertain or Essential genes.
Run-time

The Gumbel method takes on the order of 10 minutes for 10,000 samples. Run-time is linearly proportional to the ‘samples’ parameter, or length of MH sampling trajectory. Other notes: Gumbel can be run on multiple replicates; replicate datasets will be automatically merged.

2.5.2 griffin

This is an earlier version of the Gumbel method that identifies essential genes based on how unlikely ‘gaps’ (or consecutive runs of TA sites with 0 insertions) are, given the overall level of saturation. It is a frequentist (non-Bayesian) model that uses the Gumbel Extreme-Value Distribution as a likelihood function. This is the analysis used in our paper on cholesterol catabolism (Griffin et al., 2011). All things considered, you are probably better off using the hierarchical-Bayesian Gumbel model above, which does a better job of estimating internal parameters.

2.5.3 Tn5Gaps

The Tn5Gaps method can be used to determine which genes are essential in a single condition for Tn5 datasets. It does an analysis of the insertions at each site within the genome, makes a call for a given gene based on the length of the most heavily overlapping run of sites without insertions (gaps), calculates the probability of this using a the Gumbel distribution.

Note: Intended only for Tn5 datasets.

How does it work?

This method is loosely is based on the original gumbel analysis method described in this paper:


The Tn5Gaps method modifies the original method in order to work on Tn5 datasets, which have significantly lower saturation of insertion sites than Himar1 datasets. The main difference comes from the fact that the runs of non-insertion (or “gaps”) are analyzed throughout the whole genome, including non-coding regions, instead of within single genes. In doing so, the expected maximum run length is calculated and a p-value can be derived for every run.
A gene is then classified by using the p-value of the run with the largest number of nucleotides overlapping with the gene.

This method was tested on a Salmonella Tn5 dataset presented in this paper:


This data was downloaded from SRA (located here), and used to make wig files (base and bile) and the following 4 baseline datasets were merged to make a wig file: (IL2_2122_1,3,6,8). Our analysis produced 415 genes with adjusted p-values less than 0.05, indicating essentiality, and the analysis from the above paper produced 356 essential genes. Of these 356 essential genes, 344 overlap with the output of our analysis.

### Usage

```
python3 ../../../transit.py tn5gaps <comma-separated .wig files> <annotation .prot_table or GFF3> <output file> [Optional Arguments]
```

Optional Arguments:
- `-m <integer>` := Smallest read-count to consider. Default: `-m 1`
- `-r <string>` := How to handle replicates. Sum or Mean. Default: `-r Sum`
- `-IN <float>` := Ignore TAs occurring within given percentage (as integer) of the N terminus. Default: `-IN 0`
- `-IC <float>` := Ignore TAs occurring within given percentage (as integer) of the C terminus. Default: `-IC 0`

### Parameters

- **Minimum Read:** The minimum read count that is considered a true read. Because the Gumbel method depends on determining gaps of TA sites lacking insertions, it may be susceptible to spurious reads (e.g. errors). The default value of 1 will consider all reads as true reads. A value of 2, for example, will ignore read counts of 1.
- **Replicates:** Determines how to deal with replicates by averaging the read-counts or suming read counts accross datasets. This should not have an affect for the Gumbel method, aside from potentially affecting spurious reads.
- `-IN:` Trimming of insertions in N-terminus (given as percentage of ORF length, e.g. “5” for 5%; default=0)
- `-IC:` Trimming of insertions in C-terminus (given as percentage of ORF length, e.g. “5” for 5%; default=0)

### Example

```
python3 PATH/src/transit.py tn5gaps salmonella_baseline.wig Salmonella-Ty2.prot_table ...
```

These input and output files can be downloaded from the **Example Data** section on the *Transit home page*.  

### 2.5. Analysis Methods
Outputs and diagnostics

The Tn5Gaps method generates a tab-separated output file at the location chosen by the user. This file will automatically be loaded into the Results Files section of the GUI, allowing you to display it as a table. Alternatively, the file can be opened in a spreadsheet software like Excel as a tab-separated file. The columns of the output file are defined as follows:

<table>
<thead>
<tr>
<th>Column Header</th>
<th>Column Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORF</td>
<td>Gene ID.</td>
</tr>
<tr>
<td>Name</td>
<td>Name of the gene.</td>
</tr>
<tr>
<td>Desc</td>
<td>Gene description.</td>
</tr>
<tr>
<td>k</td>
<td>Number of Transposon Insertions Observed within the ORF.</td>
</tr>
<tr>
<td>n</td>
<td>Total Number of TA dinucleotides within the ORF.</td>
</tr>
<tr>
<td>r</td>
<td>Length of the Maximum Run of Non-Insertions observed.</td>
</tr>
<tr>
<td>ovr</td>
<td>The number of nucleotides in the overlap with the longest run partially covering the gene.</td>
</tr>
<tr>
<td>lenovr</td>
<td>The length of the above run with the largest overlap with the gene.</td>
</tr>
<tr>
<td>pval</td>
<td>P-value calculated by the permutation test.</td>
</tr>
<tr>
<td>padj</td>
<td>Adjusted p-value controlling for the FDR (Benjamini-Hochberg).</td>
</tr>
<tr>
<td>call</td>
<td>Essentaility call for the gene. Depends on FDR corrected thresholds. Essential or Non-Essential.</td>
</tr>
</tbody>
</table>

Run-time

The Tn5Gaps method takes on the order of 10 minutes. Other notes: Tn5Gaps can be run on multiple replicates; replicate datasets will be automatically merged.

2.5.4 HMM

The HMM method can be used to determine the essentiality of the entire genome, as opposed to gene-level analysis of the other methods. It is capable of identifying regions that have unusually high or unusually low read counts (i.e. growth advantage or growth defect regions), in addition to the more common categories of essential and non-essential.

Note: Intended only for Himar1 datasets.
How does it work?

For a formal description of how this method works, see our paper [DeJesus2013HMM]:


Example

```shell
python3 transit.py hmm <comma-separated .wig files> <annotation .prot_table or GFF3> ←<output file>
```

Optional Arguments:
- `-r <string>` := How to handle replicates. Sum, Mean. Default: `-r Mean`
- `-l` := Perform LOESS Correction; Helps remove possible genomic position bias. Default: Off.
- `-iN <float>` := Ignore TAs occuring at given percentage (as integer) of the N terminus. Default: `-iN 0`
- `-iC <float>` := Ignore TAs occuring at given percentage (as integer) of the C terminus. Default: `-iC 0`

Parameters

The HMM method automatically estimates the necessary statistical parameters from the datasets. You can change how the method handles replicate datasets:

- **Replicates**: Determines how the HMM deals with replicate datasets by either averaging the read-counts or summing read counts across datasets. For regular datasets (i.e. mean-read count > 100) the recommended setting is to average read-counts together. For sparse datasets, summing read-counts may produce more accurate results.

Output and Diagnostics

The HMM method outputs two files. The first file provides the most likely assignment of states for all the TA sites in the genome. Sites can belong to one of the following states: “E” (Essential), “GD” (Growth-Defect), “NE” (Non-Essential), or “GA” (Growth-Advantage). In addition, the output includes the probability of the particular site belonging to the given state. The columns of this file are defined as follows:
The second file provides a gene-level classification for all the genes in the genome. Genes are classified as “E” (Essential), “GD” (Growth-Defect), “NE” (Non-Essential), or “GA” (Growth-Advantage) depending on the number of sites within the gene that belong to those states.

Note: Libraries that are too sparse (e.g. < 30%) or which contain very low read-counts may be problematic for the HMM method, causing it to label too many Growth-Defect genes.

**Run-time**

The HMM method takes less than 10 minutes to complete. The parameters of the method should not affect the running-time.
2.5.5 Resampling

The resampling method is a comparative analysis that allows one to determine conditional essentiality of genes. It is based on a permutation test, and is capable of determining read-counts that are significantly different across conditions.

See Pathway Enrichment Analysis for post-processing the hits to determine if the hits are associated with a particular functional category of genes or known biological pathway.

**Note:** Can be used for both Himar1 and Tn5 datasets

**How does it work?**

This technique has yet to be formally published in the context of differential essentiality analysis. Briefly, the read-counts at each gene are determined for each replicate of each condition. The mean read-count in condition A is subtracted from the mean read-count in condition B, to obtain an observed difference in means. The TA sites are then permuted for a given number of “samples”. For each one of these permutations, the difference in read-counts is determined. This forms a null distribution, from which a p-value is calculated for the original, observed difference in read-counts.

**Usage**

```bash
python3 transit.py resampling <comma-separated .wig control files> <comma-separated .wig experimental files> <annotation .prot_table or GFF3> <output file> [Optional Arguments]
```

Optional Arguments:

- `s <integer>` := Number of samples. Default: -s 10000
- `n <string>` := Normalization method. Default: -n TTR
- `h` := Output histogram of the permutations for each gene.

- Default: Turned Off.
- `-PC <float>` := Pseudocounts used in calculating LFC. (default: 1)
- `-l` := Perform LOESS Correction; Helps remove possible genomic position bias. Default: Turned Off.
- `-IN <float>` := Ignore TAs occurring at given percentage (as integer) of the N terminus. Default: -IN 0
- `-iC <float>` := Ignore TAs occurring at given percentage (as integer) of the C terminus. Default: -iC 0
- `--ctrl_lib` := String of letters representing library of control files e.g. 'AABB'. Default empty. Letters used must also be used in
- `--exp_lib` := String of letters representing library of experimental files

(continues on next page)
If non-empty, resampling will limit permutations to within-libraries.

--exp_lib := String of letters representing library of experimental files in order e.g. 'ABAB'. Default empty. Letters used must also be used in --ctrl_lib

--in --ctrl_lib

If non-empty, resampling will limit permutations to within-libraries.

### Parameters

The resampling method is non-parametric, and therefore does not require any parameters governing the distributions or the model. The following parameters are available for the method:

- **Samples:** The number of samples (permutations) to perform. The larger the number of samples, the more resolution the p-values calculated will have, at the expense of longer computation time. The resampling method runs on 10,000 samples by default.

- **Output Histograms:** Determines whether to output .png images of the histograms obtained from resampling the difference in read-counts.

- **Adaptive Resampling:** An optional “adaptive” version of resampling which accelerates the calculation by terminating early for genes which are likely not significant. This dramatically speeds up the computation at the cost of less accurate estimates for those genes that terminate early (i.e. deemed not significant). This option is OFF by default. (see Notes below)

- **Include Zeros:** Select to include sites that are zero. This is the preferred behavior, however, unselecting this (thus ignoring sites that) are zero across all dataset (i.e. completely empty), is useful for decreasing running time (specially for large datasets like Tn5).

- **Normalization Method:** Determines which normalization method to use when comparing datasets. Proper normalization is important as it ensures that other sources of variability are not mistakenly treated as real differences. See the Normalization section for a description of normalization method available in TRANSIT.

- **-ctrl_lib, -exp_lib:** These are for doing resampling with datasets from multiple libraries, see below.

- **-iN, -iC:** Trimming of TA sites near N- and C-terminus. The default for trimming TA sites in the termini of ORFs is 0. However, TA sites in the stop codon (e.g. TAG) are automatically excluded. Trimming is specified as a percentage (as an integer), so, for example, if you want to trim TA sites within 5% of the termini, you would add the flags ‘-iN 5 -iC 5’ (not 0.05).

- **-PC:** Pseudocounts used in calculation of LFCs (log-fold-changes, see Output and Diagnostics) in resampling output file. To suppress the appearance of artifacts due to high-magnitude of LFCs from genes with low insertion counts (which are more susceptible to noise), one can increase the pseudocounts using ‘-PC’. Increasing PC to a value like 5 (which is reasonable, given that TTR normalization scales data so average insertion counts is around 100) can further reduce the appearance of artifacts (genes with low counts but large LFCs). However, changing pseudocounts only affects the LFCs, and will not change the number of significant genes.
Notes

I recommend using -a (adaptive resampling). It runs much faster, and the p-values will be very close to a full non-adaptive run (all 10,000 samples).

Occasionally, people ask if resampling can be done on intergenic regions as well. It could be done pretty easily (for example by making a prot_table with coordinates for the regions between genes). But it is usually not worthwhile, because most intergenic regions are small (<100 bp) contain very few TA sites (often 0-2), making it difficult to make confident calls on essentiality.

Doing resampling with a combined_wig file

Resampling can also now take a combined_wig file as input (containing insertion counts for multiple sample), along with a samples_metadata file that describes the samples. This mode is indicated with a ‘-c’ flag. If you want to compare more than two conditions, see ZINB.

usage:

```
python3 transit.py resampling -c <combined_wig> <samples_metadata> <control_condition_name> <experimental_condition_name> <annotation.prot_table or GFF3> <output file>
```

example:

```
python3 transit.py resampling -c antibiotic_combined_wig.txt antibiotic_samples_metadata.txt Untreated Isoniazid H37Rv.prot_table results.txt -a
```

Doing resampling with datasets from different libraries.

In most cases, comparisons are done among samples (replicates) from the same library evaluated in two different conditions. But if the samples themselves come from different libraries, then this could introduce extra variability, the way resampling is normally done. To compensate for this, if you specify which libraries each dataset comes from, the permutations will be restricted to permuting counts only among samples within each library. Statistical significance is still determined from all the data in the end (by comparing the observed difference of means between the two conditions to a null distribution). Of course, this method makes most sense when you have at least 1 replicate from each library in each condition.

Doing resampling between different strains.

The most common case is that resampling is done among replicates all from the same Tn library, and hence all the datasets (fastq files) are mapped to the same reference genome. Occasionally, it is useful to compare TnSeq datasets between two different strains, such as a reference strain and a clinical isolate from a different lineage. Suppose for simplicity that you want to compare one replicate from strain A (e.g. H37Rv) and one replicate from strain B (e.g. CDC1551). Resampling was not originally designed to handle this case. The problem is that the TA sites in the .wig files with insertion counts might have different coordinates (because of shifts due to indels between the genomes). Furthermore, a given gene might not even have the same number of TA sites in the two strains (due to SNPs). A simplistic solution is to just map both datasets to the same genome sequence (say H37Rv, for example). Then a resampling comparison could be run as usual, because the TA sites would all be on the same coordinate system. This is not ideal, however, because some reads of strain B might not map properly to genome A due to SNPs or indels.
between the genomes. In fact, in more divergent organisms with higher genetic diversity, this can cause entire regions to look artificially essential, because reads fail to map in genes with a large number of SNPs, resulting in the apparent absence of transposon insertions.

A better approach is to map each library to the custom genome sequence of its own strain (using TPP). It turns out the resampling can still be applied (since it is fundamentally a test on the difference of the mean insertion count in each gene). The key to making this work, aside from mapping each library to its own genome sequence, is that you need an annotation (prot_table) for the second strain that has been “adapted” from the first strain. This is because, to do a comparison between conditions for a gene, Transit needs to be able to determine which TA sites fall in that gene for each strain. This can be achieved by producing a “modified” prot_table, where the START and END coordinates of each ORF in strain B have been adjusted according to an alignment between genome A and genome B. You can use this web app: Prot_table Adjustment Tool, to create a modified prot_table, given the prot_table for one strain and the fasta files for both genomes (which will be aligned). In other words, the app allows you to create ‘B.prot_table’ from ‘A.prot_table’ (and ‘A.fna’ and ‘B.fna’).

Once you have created B.prot_table, all you need to do is provide both prot_tables to resampling (either through the GUI, or on the command-line), as a comma-separated list. For example:

```bash
> python3 transit.py resampling Rv_1_H37Rv.wig,Rv_2_H37Rv.wig 632_1_632WGS.wig,632_2_632WGS.wig H37Rv.prot_table,632WGS.prot_table resampling_output.txt -a
```

In this example, 2 replicates from H37Rv (which had been mapped to H37Rv.fna by TPP) were compared to 2 replicates from strain 632 (which had been mapped to 632WGS.fna, the custom genome seq for strain 632). The important point is that two annotations are given in the 3rd arg on the command-line: H37Rv.prot_table,632WGS.prot_table. The assumption is that the ORF boundaries for H37Rv will be used to find TA sites in Rv_1_H37Rv.wig and Rv_2_H37Rv.wig, and the ORF boundaries in 632WGS.prot_table (which had been adapted from H37Rv.prot_table using the web app above) will be used to find TA sites in the corresponding regions in 632_1_632WGS.wig and 632_2_632WGS.wig.

Note that, in contrast to handling datasets from different libraries discussed above, in this case, the assumption is that all replicates in condition A will be from one library (and one strain), and all replicates in condition B will be from another library (another strain).

### Output and Diagnostics

The resampling method outputs a tab-delimited file with results for each gene in the genome. P-values are adjusted for multiple comparisons using the Benjamini-Hochberg procedure (called “q-values” or “p-adj.”). A typical threshold for conditional essentiality on is q-value < 0.05.
<table>
<thead>
<tr>
<th>Column Header</th>
<th>Column Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orf</td>
<td>Gene ID.</td>
</tr>
<tr>
<td>Name</td>
<td>Name of the gene.</td>
</tr>
<tr>
<td>Description</td>
<td>Gene description.</td>
</tr>
<tr>
<td>Sites</td>
<td>Number of TA sites in the gene.</td>
</tr>
<tr>
<td>Mean Ctrl</td>
<td>Mean of read counts in condition 1. (avg over TA sites and reps)</td>
</tr>
<tr>
<td>Mean Exp</td>
<td>Mean of read counts in condition 2.</td>
</tr>
<tr>
<td>log2FC</td>
<td>Log-fold-change of exp (treatment) over ctrl (untreated)</td>
</tr>
<tr>
<td>Sum Ctrl</td>
<td>Sum of read counts in condition 1.</td>
</tr>
<tr>
<td>Sum Exp</td>
<td>Sum of read counts in condition 2.</td>
</tr>
<tr>
<td>Delta Mean</td>
<td>Difference in the MEAN insertion counts.</td>
</tr>
<tr>
<td>p-value</td>
<td>P-value calculated by the permutation test.</td>
</tr>
<tr>
<td>Adj. p-value</td>
<td>Adjusted p-value controlling for the FDR (Benjamini-Hochberg)</td>
</tr>
</tbody>
</table>

**log2FC**: (log-fold-change, LFC) For each gene, the LFC is calculated as the log-base-2 of the ratio of mean insertion counts in the experimental (treated) condition vs. the control condition (untreated, reference). The default is PC=1, which avoids the result being undefined for genes with means of 0 in either condition. Pseudocounts can be changed using the -PC flag (above).

\[ \text{LFC} = \log_2 \left( \frac{\text{mean_insertions_in_exp} + \text{PC}}{\text{mean_insertions_in_ctrl} + \text{PC}} \right) \]

**Run-time**

A typical run of the resampling method with 10,000 samples will take around 45 minutes (with the histogram option ON). Using the adaptive resampling option (-a), the run-time is reduced to around 10 minutes.

### 2.5.6 Mann-Whitney U-test (utest)

This is a method for comparing datasets from a TnSeq library evaluated in two different conditions, analogous to resampling. This is a *rank-based* test on whether the level of insertions in a gene or chromosomal region are significantly higher or lower in one condition than the other. Effectively, the insertion counts at the TA sites in the region are pooled and sorted. Then the combined ranks of the counts in region A are compared to those in region B, and p-value is calculated that reflects whether there is a significant difference in the ranks. The advantage of this method is that it is less sensitive to outliers (a unusually high insertion count at just a single TA site). A reference for this method is (Santa Maria et al., 2014).
2.5.7 Genetic Interactions

The genetic interactions (GI) method is a comparative analysis used to determine genetic interactions. It is a Bayesian method that estimates the distribution of log fold-changes (logFC) in two strain backgrounds under different conditions, and identifies significantly large changes in enrichment (delta_logFC) to identify those genes that imply a genetic interaction.

Note: Can be used for both Himar1 and Tn5 datasets

How does it work?

GI performs a comparison among 4 groups of datasets, strain A and B assessed in conditions 1 and 2 (e.g. control vs treatment). It looks for interactions where the response to the treatment (i.e. effect on insertion counts) depends on the strain.

If you think of the effect of treatment as a log-fold-change (e.g. of the insert counts between control and treatment in strain A), which is like a “slope”, then the interacting genes are those that exhibit a difference in the effect of the treatment between the strains, and hence a difference in the slopes between strain A and B (represented by ‘delta_LFC’ in the output file).

For a formal description of how this method works, see our paper [DeJesus2017NAR]:


Statistical Significance

The computation that is done by GI is to compute the posterior distribution of the delta_LFC (or mean change in slopes) through Bayesian sampling. The primary method to determine significance of genes is whether the mean_delta_LFC is significantly different than 0. However, since the mean_delta_LFC is a distribution, we represent it by a Highest Density Interval, HDI, which is similar to a 95% confidence interval. Furthermore, rather than asking whether the HDI overlaps 0 exactly, we expand the interval around 0 to a Region of Probable Equivalence (ROPE), which is set to [-0.5,0.5] by default. Hence the significant genes are those for which the HDI does not overlap the ROPE. GI has a flag to adjust the size of the ROPE, if desired.

In the GI output file, the final column give the significance call, along with type of interaction. If a gene is not significant, it will be marked with “No Interaction” (for the HDI method, meaning HDI overlaps the ROPE). If a gene is significant, then its interaction will be categorized in 3 types (see NAR paper):

- **Aggravating** - mean_delta_LFC is negative; gene is more required in treatment than control in the B strain, compared to the A strain
- **Suppressive** - mean_delta_LFC is positive, and the gene was not conditionally essential in strain A (flat slope), but becomes conditionally non-essential in strain B when treated (positive slope)
- **Alleviating** - mean_delta_LFC is positive, but the conditional requirement (negative slope) of the gene in strain A with treatment is “cancelled” by the modification in strain B

A limitation of this HDI approach is that it is discrete (i.e. overlap is either True or False), but does not provide a quantitative metric for the degree of overlap. Thus a second method for assessing significance of genetic interactions is to compute the probability of overlap. The lower the probability, the more different the delta_LFC is from 0, indicating a more significant interaction. In this case, genes with prob < 0.05 are considered interactions and classified by the 3 types above, while genes with prob >= 0.05 are marked as “No Interaction”.

In addition, since we are calculating significance for thousands of genes in parallel, many researchers prefer to have some method for correcting for multiple tests, to control the false discovery rate. However, FDR correction is generally used only for frequentist analyses, and the GI method is fundamentally a Bayesian approach. Technically, in a Bayesian framework, FDR correction is not needed. Any adjustment for expectations about number of hits would be achieved through adjusting parameters for prior distributions. Nonetheless, GI includes options for two methods that approximate FDR correction: **BFDR** (Bayesian False Discovery Rate correction, Newton M.A., Noueiry A., Sarkar D., Ahlquist P. (2004). Detecting differential gene expression with a semiparametric hierarchical mixture method. Biostatistics, 5:155–176.) and FWER (Family-Wise Error Rate control). When these corrections are applied, a threshold of 0.05 for the adjusted probability of overlap is used for each, and this determines which genes are classified as interacting (1 of 3 types) or marked as “No Interaction”, as above.

In order to enable users to evaluate these various methods for determining significance of interactions, a ‘-signif’ flag is provided for the GI method. The options are:

- **-signif HDI**: significant genes are those for which the HDI does not overlap the ROPE
- **-signif prob**: significant genes are those with prob < 0.05, where ‘prob’ is porbability that HDI overlap the ROPE (default)
- **-signif BFDR**: significant genes are those with adjusted prob < 0.05, where prob is adjusted by the BFDR method
- **-signif FWER**: significant genes are those with adjusted prob < 0.05, where prob is adjusted by the FWER method

‘-signif prob’ is the default method.
In the output file, the genes are sorted by the probability that the HDI overlaps the ROPE. The genes at the top are roughly the genes with the highest absolute value of mean_delta_LFC.

Usage

```python
python3 /pacific/home/ioerger/transit/src/transit.py GI <wigs_for_strA_cond1> <wigs_for_strA_cond2> <wigs_for_strB_cond1> <wigs_for_strB_cond2> <annotation .prot_table> or GFF3 <output file> [Optional Arguments]
```

GI performs a comparison among 4 groups of datasets, strain A and B assessed in conditions 1 and 2 (e.g. control vs treatment).

- It looks for interactions where the response to the treatment (i.e. effect on insertion counts) depends on the strain (output variable: delta_LFC).
- Provide replicates in each group as a comma-separated list of wig files.
- HDI is highest density interval for posterior distribution of delta_LFC, which is like a confidence interval on difference of slopes.
- Genes are sorted by probability of HDI overlapping with ROPE. (genes with the highest abs(mean_delta_logFC) are near the top, approximately)
- Significant genes are indicated by 'Type of Interaction' column (No Interaction, Aggravating, Alleviating, Suppressive).
- By default, hits are defined as "Is HDI outside of ROPE?"=TRUE (i.e. non-overlap of delta_LFC posterior distribution with Region of Probably Equivalence around 0)
- Alternative methods for significance: use -signif flag with prob, BFDR, or FWER. These affect 'Type of Interaction' (i.e. which genes are labeled 'No Interaction')

Optional Arguments:
- -s <integer> := Number of samples. Default: -s 10000
- --rope <float> := Region of Practical Equivalence. Area around 0 (i.e. 0 +/- ROPE) that is NOT of interest. Can be thought of similar to the area of the null-hypothesis. Default: --rope 0.5
- -n <string> := Normalization method. Default: -n TTR
- -iz := Include rows with zero across conditions.
- -l := Perform LOESS Correction; Helps remove possible genomic position bias. Default: Turned Off.
- -iN <float> := Ignore TAs occuring at given percentage (as integer) of the N terminus. Default: -iN 0
- -iC <float> := Ignore TAs occuring at given percentage (as integer) of the C terminus. Default: -iC 0
- -signif HDI := (default) Significant if HDI does not overlap ROPE; if HDI overlaps ROPE, 'Type of Interaction' is set to 'No Interaction'
- -signif prob := Optionally, significant hits are re-defined based on probability (degree) of overlap of HDI with ROPE, prob<0.05 (no adjustment)
- -signif BFDR := Apply "Bayesian" FDR correction (see doc) to adjust HDI-ROPE overlap probabilities so that significant hits are re-defined as BFDR<0.05
- -signif FWER := Apply "Bayesian" FWER correction (see doc) to adjust HDI-ROPE overlap probabilities so that significant hits are re-defined as FWER<0.05

Example

In this example, the effect of a knockout of SigB is being evaluated for its effect on tolerance of isoniazid. Some genes may become more essential (or less) in the presence of INH in the wild-type strain. The genes implied to interact with SigB are those whose response to INH changes in the knock-out strain compared to the wild-type. Note there are 2 replicates in each of the 4 groups of datasets.
### Parameters

The resampling method is non-parametric, and therefore does not require any parameters governing the distributions or the model. The following parameters are available for the method:

- **Samples:** The number of samples (permutations) to perform. The larger the number of samples, the more resolution the p-values calculated will have, at the expense of longer computation time. The resampling method runs on 10,000 samples by default.

- **ROPE:** Region of Practical Equivalence. This region defines an area around 0.0 that represents differences in the log fold-change that are practically equivalent to zero. This aids in ignoring spurious changes in the logFC that would otherwise be identified under a strict null-hypothesis of no difference.

- **Include Zeros:** Select to include sites that are zero. This is the preferred behavior, however, unselecting this (thus ignoring sites that are zero across all dataset (i.e. completely empty), is useful for decreasing running time (especially for large datasets like Tn5).

- **Normalization Method:** Determines which normalization method to use when comparing datasets. Proper normalization is important as it ensures that other sources of variability are not mistakenly treated as real differences. See the [Normalization section](#) for a description of normalization method available in TRANSIT.

- **Significance Method:**
  - **-signif HDI:** significant genes are those for which the HDI does not overlap the ROPE
  - **-signif prob:** significant genes are those with prob \(< 0.05\), where ‘prob’ is porbability that HDI overlap the ROPE (default)
  - **-signif BFDR:** significant genes are those with adjusted prob \(< 0.05\), where prob is adjusted by the BFDR method
  - **-signif FWER:** significant genes are those with adjusted prob \(< 0.05\), where prob is adjusted by the FWER method

### Output and Diagnostics

The GI method outputs a tab-delimited file with results for each gene in the genome. All genes are sorted by significance using the probability that the HDI overlaps the ROPE. Significant genes are those NOT marked with ‘No Interaction’ in the last column.
<table>
<thead>
<tr>
<th>Column Header</th>
<th>Column Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orf</td>
<td>Gene ID.</td>
</tr>
<tr>
<td>Name</td>
<td>Name of the gene.</td>
</tr>
<tr>
<td>Number of TA Sites</td>
<td>Number of TA sites in the gene.</td>
</tr>
<tr>
<td>Mean count (Strain A Condition 1)</td>
<td>Mean read count in strain A, condition 1</td>
</tr>
<tr>
<td>Mean count (Strain A Condition 2)</td>
<td>Mean read count in strain A, condition 2</td>
</tr>
<tr>
<td>Mean count (Strain B Condition 1)</td>
<td>Mean read count in strain B, condition 1</td>
</tr>
<tr>
<td>Mean count (Strain B Condition 2)</td>
<td>Mean read count in strain B, condition 2</td>
</tr>
<tr>
<td>Mean logFC (Strain A)</td>
<td>The log2 fold-change in read-count for strain A</td>
</tr>
<tr>
<td>Mean logFC (Strain B)</td>
<td>The log2 fold-change in read-count for strain B</td>
</tr>
<tr>
<td>Mean delta logFC</td>
<td>The difference in log2 fold-change between B and A</td>
</tr>
<tr>
<td>Lower Bound delta logFC</td>
<td>Lower bound of the difference (delta logFC)</td>
</tr>
<tr>
<td>Upper Bound delta logFC</td>
<td>Upper bound of the difference (delta logFC)</td>
</tr>
<tr>
<td>Prob. of delta-logFC being within ROPE</td>
<td>Portion of the delta-logFC within ROPE</td>
</tr>
<tr>
<td>Adjusted Probability (BFDR)</td>
<td>Posterior probability adjusted for comparisons</td>
</tr>
<tr>
<td>Is HDI outside ROPE?</td>
<td>True/False whether the delta-logFC overlaps ROPE</td>
</tr>
<tr>
<td>Type of Interaction</td>
<td>Final classification.</td>
</tr>
</tbody>
</table>

### 2.5.8 ANOVA

The Anova (Analysis of variance) method is used to determine which genes exhibit statistically significant variability of insertion counts across multiple conditions. Unlike other methods which take a comma-separated list of wig files as input, the method takes a combined_wig file (which combined multiple datasets in one file) and a samples_metadata file (which describes which samples/replicates belong to which experimental conditions).

#### How does it work?

The method performs the One-way anova test for each gene across conditions. It takes into account variability of normalized transposon insertion counts among TA sites and among replicates, to determine if the differences among the mean counts for each condition are significant.

#### Example

```bash
python3 transit.py anova <combined wig file> <samples_metadata file> <annotation .prot_table> <output file> [Optional Arguments]
```

Optional Arguments:
- `-n <string>` := Normalization method. Default: `-n TTR`
- `--ignore-conditions <cond1,...>` := Comma separated list of conditions to ignore, for the analysis. Default: `None`
- `--include-conditions <cond1,...>` := Comma separated list of conditions to include, for the analysis. Default: `All`

(continues on next page)
The output file generated by ANOVA identifies which genes exhibit statistically significant variability in counts across conditions (see Output and Diagnostics below).

Note: the combined_wig input file can be generated from multiple wig files through the Transit GUI (File->Export->Selected_Datasets->Combined_wig), or via the ‘export’ command on the command-line (see combined_wig).

Format of the samples metadata file: a tab-separated file (which you can edit in Excel) with 3 columns: Id, Condition, and Filename (it must have these headers). You can include other columns of info, but do not include additional rows. Individual rows can be commented out by prefixing them with a ‘#’. Here is an example of a samples metadata file: The filenames should match what is shown in the header of the combined_wig (including pathnames, if present).

<table>
<thead>
<tr>
<th>ID</th>
<th>Condition</th>
<th>Filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>glyc1</td>
<td>glycerol</td>
<td>/Users/example_data/glycerol_rep1.wig</td>
</tr>
<tr>
<td>glyc2</td>
<td>glycerol</td>
<td>/Users/example_data/glycerol_rep2.wig</td>
</tr>
<tr>
<td>chol1</td>
<td>cholesterol</td>
<td>/Users/example_data/cholesterol_rep1.wig</td>
</tr>
<tr>
<td>chol2</td>
<td>cholesterol</td>
<td>/Users/example_data/cholesterol_rep2.wig</td>
</tr>
<tr>
<td>chol2</td>
<td>cholesterol</td>
<td>/Users/example_data/cholesterol_rep3.wig</td>
</tr>
</tbody>
</table>

Parameters

The following parameters are available for the method:

- **Ignore Conditions, Include Conditions**: Can use this to drop conditions not of interest or specify a particular subset of conditions to use for ANOVA analysis.

- **Normalization Method**: Determines which normalization method to use when comparing datasets. Proper normalization is important as it ensures that other sources of variability are not mistakenly treated as real differences. See the *Normalization* section for a description of normalization method available in TRANSIT.

- **-PC** Pseudocounts to use in calculating LFCs (see below). Default: -PC 5

Output and Diagnostics

The anova method outputs a tab-delimited file with results for each gene in the genome. P-values are adjusted for multiple comparisons using the Benjamini-Hochberg procedure (called “q-values” or “p-adj.”). A typical threshold for conditional essentiality on is q-value < 0.05.

<table>
<thead>
<tr>
<th>Column Header</th>
<th>Column Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orf</td>
<td>Gene ID.</td>
</tr>
<tr>
<td>Name</td>
<td>Name of the gene.</td>
</tr>
<tr>
<td>TAs</td>
<td>Number of TA sites in Gene</td>
</tr>
<tr>
<td>Means...</td>
<td>Mean readcounts for each condition</td>
</tr>
<tr>
<td>LFCs...</td>
<td>Log-fold-changes of counts in each condition vs mean across all conditions</td>
</tr>
<tr>
<td>p-value</td>
<td>P-value calculated by the Anova test.</td>
</tr>
<tr>
<td>p-adj</td>
<td>Adjusted p-value controlling for the FDR (Benjamini-Hochberg)</td>
</tr>
<tr>
<td>status</td>
<td>Debug information (If any)</td>
</tr>
</tbody>
</table>
LFCs (log-fold-changes): For each condition, the LFC is calculated as the log-base-2 of the ratio of mean insertion count in that condition relative to the mean of means across all the conditions. Pseudocount are incorporated to reduce the impact of noise on LFCs, based on the formula below. The pseudocounts can be adjusted using the -PC flag. Changing the pseudocounts (via -PC) can reduce the artificial appearance of genes with high-magnitude LFCs but that have small overall counts (which are susceptible to noise). Changing the pseudocounts will not affect the analysis of statistical significance and hence number of varying genes, however.

\[
LFC = \log_2(\frac{\text{mean insertions in condition} + PC}{\text{mean of means across all conditions} + PC})
\]

Run-time

A typical run of the anova method takes less than 1 minute for a combined wig file with 6 conditions, 3 replicates per condition.

2.5.9 ZINB

The ZINB (Zero-Inflated Negative Binomial) method is used to determine which genes exhibit statistically significant variability in either the magnitude of insertion counts or local saturation, across multiple conditions. Like ANOVA, the ZINB method takes a combined_wig file (which combines multiple datasets in one file) and a samples_metadata file (which describes which samples/replicates belong to which experimental conditions).

ZINB can be applied to two or more conditions at a time. Thus it subsumes resampling. Our testing suggests that ZINB typically identifies 10-20% more varying genes than resampling (and vastly out-performs ANOVA for detecting significant variability across conditions). Furthermore, because of how ZINB treats magnitude of read counts separately from local saturation in a gene, it occasionally identifies genes with variability not detectable by resampling analysis.

Note: ZINB analysis requires R (statistical analysis software) to be installed on your system, along with the 'pscl' R package. See Installation Instructions.

How does it work?

For a formal description of how this method works, see our paper [ZINB]:

Example

```
python3 transit.py zinb <combined wig file> <samples_metadata file> <annotation .prot_table> <output file> [Optional Arguments]

Optional Arguments:
- n <string> := Normalization method. Default: -n TTR
- ignore-conditions <cond1,cond2> := Comma separated list of conditions to ignore, for the analysis. Default: None
- include-conditions <cond1,cond2> := Comma separated list of conditions to include, for the analysis. Default: All
- iN <float> := Ignore TAs occurring within given percentage of the N terminus. Default: -iN 5
- iC <float> := Ignore TAs occurring within given percentage of the C terminus. Default: -iC 5
- PC <N> := Pseudocounts used in calculating LFCs in output file. Default: -PC 5
- condition := columnname (in samples_metadata) to use as the Condition. Default: "Condition"
- covars <covar1,covar2...> := Comma separated list of covariates (in metadata file) to include, for the analysis.
- interactions <covar1,covar2...> := Comma separated list of covariates to include, that interact with the condition for the analysis.
- v := verbose, print out the model coefficients for each gene.
- gene <Orf id or Gene name> := Run method for one gene and print model output.
```

Combined wig files

Transit now supports a new file format called ‘combined_wig’ which basically combines multiple wig files into one file (with multiple columns). This is used for some of the new analysis methods for larger collections of datasets, like Anova, ZINB. Combined_wig files can be created through the Transit GUI (File->Export->Selected_Datasets->Combined_wig), or via the command line. You can specify the normalization method you want to use with a flag. TTR is the default, but other relevant normalization options would be ‘nonorm’ (i.e. preserve raw counts) and ‘beta-geom’ (this corrects for skew, but is slow).

```
> python3 src/transit.py export combined_wig --help
usage: python3 src/transit.py export combined_wig <comma-separated .wig files>
                  <annotation .prot_table> <output file>

> python3 ../transit/src/transit.py export combined_wig Rv_1_H37RvRef.wig,Rv_2_H37RvRef.wig,Rv_3_H37RvRef.wig H37Rv.prot_table clinicals_combined_TTR.wig -n TTR
```

Samples Metadata File

Format of the samples_metadata file: a tab-separated file (which you can edit in Excel) with 3 columns: Id, Condition, and Filename (it must have these headers). You can include other columns of info, but do not include additional rows. Individual rows can be commented out by prefixing them with a ‘#’. Here is an example of a samples metadata file:

```
The filenames should match what is shown in the header of the combined_wig (including pathnames, if present).
```

<table>
<thead>
<tr>
<th>ID</th>
<th>Condition</th>
<th>Filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>glyc1</td>
<td>glycerol</td>
<td>/Users/example_data/glycerol_rep1.wig</td>
</tr>
<tr>
<td>glyc2</td>
<td>glycerol</td>
<td>/Users/example_data/glycerol_rep2.wig</td>
</tr>
<tr>
<td>chol1</td>
<td>cholesterol</td>
<td>/Users/example_data/cholesterol_rep1.wig</td>
</tr>
</tbody>
</table>

(continues on next page)
Parameters

The following parameters are available for the method:

- **Ignore Conditions**: Ignores the given set of conditions from the ZINB test.
- **Include Conditions**: Includes the given set of conditions from the ZINB test. Conditions not in this list are ignored.
- **Normalization Method**: Determines which normalization method to use when comparing datasets. Proper normalization is important as it ensures that other sources of variability are not mistakenly treated as real differences. See the Normalization section for a description of normalization method available in TRANSIT.
- **Covariates**: If additional covariates distinguishing the samples are available, such as library, timepoint, or genotype, they may be incorporated in the test.

Covariates and Interactions

While ZINB is focus on identifying variability of insertion counts across conditions, the linear model also allows you to take other variables into account. There are two types of auxilliary variables: **covariates** and **interactions**. These can be provided as extra columns in the samples metadata file. Covariates are attributes of the individual samples that could have a systematic effect on the insertion counts which we would like to account for and subsequently ignore (like nuisance variables). Examples include things like batch or library.

Interactions are extra variables for which we want to test their effect on the main variable (or condition). For example, suppose we collect TnSeq data at several different timepoints (e.g. length of incubation or infection). If we just test time as the condition, we will be identifying genes that vary over time (if timepoints are numeric, think of the model as fitting a ‘slope’ to the counts). But suppose we have data for both a wild-type and knock-out strain. Then we might be interested in genes for which the time-dependent behavior **differs** between the two strains (think: different ‘slopes’). In such a case, we would say strain and time interact.

If covariates distinguishing the samples are available, such as batch or library, they may be incorporated in the ZINB model by using the **–covars** flag and samples metadata file. For example, consider the following samples metadata file, with a column describing the batch information of each replicate.

<table>
<thead>
<tr>
<th>ID</th>
<th>Condition</th>
<th>Filename</th>
<th>Batch</th>
</tr>
</thead>
<tbody>
<tr>
<td>glyc1</td>
<td>glycerol</td>
<td>/Users/example_data/glycerol_rep1.wig</td>
<td>B1</td>
</tr>
<tr>
<td>glyc2</td>
<td>glycerol</td>
<td>/Users/example_data/glycerol_rep2.wig</td>
<td>B2</td>
</tr>
<tr>
<td>chol1</td>
<td>cholesterol</td>
<td>/Users/example_data/cholesterol_rep1.wig</td>
<td>B1</td>
</tr>
<tr>
<td>chol2</td>
<td>cholesterol</td>
<td>/Users/example_data/cholesterol_rep2.wig</td>
<td>B2</td>
</tr>
<tr>
<td>chol2</td>
<td>cholesterol</td>
<td>/Users/example_data/cholesterol_rep3.wig</td>
<td>B2</td>
</tr>
</tbody>
</table>

This information can be included to eliminate variability due to batch by using the **–covars** flag.

```
python3 transit.py zinb combined.wig samples.metadata prot.table output.file --covars Batch
```

Similarly, an interaction variable may be included in the model. This is specified by the user with the **–interactions** flag, followed by the name of a column in the samples metadata to test as the interaction with the condition. If there are multiple interactions, they may be given as a comma-separated list.
To give an example, consider an experiment where the condition represents a treatment (e.g. with values ‘treated’ and ‘control’), and we have another column called Strain (with values ‘wild-type’ and ‘mutant’). If we want to test whether the effect of the treatment (versus control) differs depending on the strain, we could do this:

```
python3 transit.py zinb combined.wig samples.metadata prot.table output.file --interactions Strain
```

In this case, the condition is implicitly assumed to be the column in the samples metadata file labeled ‘Condition’. If you want to specify a different column to use as the primary condition to test (for example, if Treatment were a distinct column), you can use the `--condition` flag:

```
python3 transit.py zinb combined.wig samples.metadata prot.table output.file --condition Treatment --interactions Strain
```

The difference between how covariates and interactions are handled in the model is discussed below in the section on Statistical Significance.

**Categorical vs Numeric Covariates**

In some cases, covariates are intended to be treated as categorical variables, like ‘batch’ or ‘library’ or ‘medium’. In other cases, a covariate might be a numeric value, such as ‘time’ or ‘concentration’, in which the ordering of values is relevant. The ZINB implementation tries to guess the type of each covariate. If they are strings, they are treated as discrete factors (each with their own distinct parameter). If the given covariate can be parsed as numbers, the model interprets them as real values. In this case, the covariate is treated as a linear factor (regressor), and is incorporated in the model as a single coefficient, capturing the slope or trend in the insertion counts as the covariate value increases.

**Statistical Significance - What the P-values Mean in the ZINB Output**

Formally, the P-value is from a likelihood ratio test (LRT) between a condition-dependent ZINB model \( m_1 \) and a condition-independent (null) ZINB model \( m_0 \).

\[
2 \ln \frac{L(m_1)}{L(m_0)} \sim \chi^2_{df}
\]

where \( L(.) \) is the ZINB likelihood function, and \( \chi^2_{df} \) is the chi-squared distribution with degrees of freedom (df) equal to difference in the number of parameters between the two models. The p-value is calculated based on this distribution.

In a simple case where variability across a set of conditions X is being tested, you can think of the model approximately as:

\[
m_1 : \ln \mu = \alpha_0 + \tilde{\alpha}X
\]

where \( \mu \) is an estimate of the mean (non-zero) insertion count in a gene (a parameter in the likelihood function for ZINB), \( \alpha_0 \) is a constant (the mean across all conditions), and \( \tilde{\alpha} \) is a vector of coefficients representing the deviation of the mean count in each condition. (There is a corresponding equation for estimating the saturation as a function of condition.)

To evaluate whether the variability across conditions is significant, we compare to a null model, where the counts are estimated by the global mean only (dropping the condition variable X).

\[
m_0 : \ln \mu = \alpha_0
\]

When a covariate C is available, it is incorporated in both models (additively), to account for the effect of the covariate in \( m_1 \). Coefficients in \( \tilde{\beta} \) represent systematic effects on the mean count due to the covariate, and effectively get

2.5. Analysis Methods
subtracted out of the condition coefficients, but \( \vec{\beta} \) is also included in the null model \( m_0 \), since we want to discount the effect of C on the likelihood and focus on evaluating the effect of X.

\[
m_1 : \ln \mu = \alpha_0 + \vec{\alpha}X + \vec{\beta}C \\
m_0 : \ln \mu = \alpha_0 + \vec{\beta}C
\]

When an interaction I is being tested, it is incorporated \textit{multiplicatively} in the main model \( m_1 \) and \textit{additively} in the null model \( m_0 \):

\[
m_1 : \ln \mu = \alpha_0 + \vec{\alpha}X + \vec{\beta}I + \vec{\gamma}X \times I \\
m_0 : \ln \mu = \alpha_0 + \vec{\alpha}X + \vec{\beta}I
\]

The meaning of this is that the coefficients \( \vec{\alpha} \) and \( \vec{\beta} \) capture the additive effects of how the mean insertion count in a gene depends on the condition variable and the interaction variable, respectively, and the \( X \times I \) term captures additional (non-additive) deviations (which is the traditional way interactions are handled in generalized linear models, GLMs). Thus, if there were no interaction, one would expect the mean in datasets representing the combination of X and I to be predicted by the offsets for each independently. To the extend that this is not the case, we say that X and I interaction, and the coefficients \( \vec{\gamma} \) for \( X \times I \) capture these deviations (non-additive effects).

For example, think of condition X as Strain (e.g. wild-type vs mutant), and interaction I as Treatment (e.g. treated versus control). Then the main model would look like this:

\[
m_1 : \ln \mu = \alpha_0 + \alpha_1 WT + \alpha_2 mutant + \beta_1 control + \beta_2 treated + \gamma_{mutant} \times treated
\]

and this would be compared to the following null model (without the interaction term):

\[
m_0 : \ln \mu = \alpha_0 + \alpha_1 WT + \alpha_2 mutant + \beta_1 control + \beta_2 treated
\]

**Output and Diagnostics**

The ZINB method outputs a tab-delimited file with results for each gene in the genome. P-values are adjusted for multiple comparisons using the Benjamini-Hochberg procedure (called “q-values” or “p-adj.”). A typical threshold for conditional essentiality on is q-value < 0.05.

<table>
<thead>
<tr>
<th>Column Header</th>
<th>Column Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orf</td>
<td>Gene ID.</td>
</tr>
<tr>
<td>Name</td>
<td>Name of the gene.</td>
</tr>
<tr>
<td>TAs</td>
<td>Number of TA sites in Gene</td>
</tr>
<tr>
<td>Means...</td>
<td>Mean read-counts for each condition</td>
</tr>
<tr>
<td>LFCs...</td>
<td>Log-fold-change (base 2) of mean insertion count relative to mean across all conditions. Pseudocounts of 5 are added. If only 2 conditions, LFC is based on ratio of second to first.</td>
</tr>
<tr>
<td>NZmeans...</td>
<td>Mean read-counts at non-zero sites for each condition</td>
</tr>
<tr>
<td>NZpercs...</td>
<td>Saturation (percentage of non-zero sites) for each condition</td>
</tr>
<tr>
<td>p-value</td>
<td>P-value calculated by the ZINB test.</td>
</tr>
<tr>
<td>p-adj</td>
<td>Adjusted p-value controlling for the FDR (Benjamini-Hochberg)</td>
</tr>
<tr>
<td>status</td>
<td>Diagnostic information (explanation for genes not analyzed)</td>
</tr>
</tbody>
</table>

LFCs (log-fold-changes): For each condition, the LFC is calculated as the log-base-2 of the ratio of mean insertion count in that condition \textit{relative to the mean of means across all the conditions}. Pseudocounts are incorporated to reduce the impact of noise on LFCs, based on the formula below. The pseudocounts can be adjusted using the -PC flag. Changing the pseudocounts (via -PC) can reduce the artificial appearance of genes with high-magnitude LFCs but that have small overall counts (which are susceptible to noise). Changing the pseudocounts will not affect the analysis of statistical significance and hence number of varying genes, however.
Run-time

A typical run of the ZINB method takes ~5 minutes to analyze a combined wig file with 6 conditions, 3 replicates per condition. It will, of course, run more slowly if you have many more conditions.

2.5.10 Normalization

Proper normalization is important as it ensures that other sources of variability are not mistakenly treated as real differences in datasets. TRANSIT provides various normalization methods, which are briefly described below:

- **TTR**: Trimmed Total Reads (TTR), normalized by the total read-counts (like totreads), but trims top and bottom 5% of read-counts. This is the recommended normalization method for most cases as it has the benefit of normalizing for difference in saturation in the context of resampling.
- **nzmean**: Normalizes datasets to have the same mean over the non-zero sites.
- **totreads**: Normalizes datasets by total read-counts, and scales them to have the same mean over all counts.
- **zinfnb**: Fits a zero-inflated negative binomial model, and then divides read-counts by the mean. The zero-inflated negative binomial model will treat some empty sites as belonging to the “true” negative binomial distribution responsible for read-counts while treating the others as “essential” (and thus not influencing its parameters).
- **quantile**: Normalizes datasets using the quantile normalization method described by Bolstad et al. (2003). In this normalization procedure, datasets are sorted, an empirical distribution is estimated as the mean across the sorted datasets at each site, and then the original (unsorted) datasets are assigned values from the empirical distribution based on their quantiles.
- **betageom**: Normalizes the datasets to fit an “ideal” Geometric distribution with a variable probability parameter $p$. Specially useful for datasets that contain a large skew. See Beta-Geometric Correction.
- **nonorm**: No normalization is performed.

Command-line

In addition to choosing normalization for various analyses in the GUI, you can also call Transit to normalize wig files from the command-line, as shown in this example:
Example

```bash
> python3 src/transit.py normalize --help
usage: python3 src/transit.py normalize <input.wig> <output.wig> [-n TTR|betageom]
   or: python3 src/transit.py normalize -c <combined_wig> <output> [-n TTR|betageom]
> python3 src/transit.py normalize Rv_l_H37RvRef.wig Rv_l_H37RvRef_TTR.wig -n TTR
> python3 src/transit.py normalize Rv_l_H37RvRef.wig Rv_l_H37RvRef_BG.wig -n betageom
```

The normalize command now also works on `combined_wig` files too. If the input file is a `combined_wig` file, indicate it with a `-c` flag.

2.5.11 Pathway Enrichment Analysis

Pathway Enrichment Analysis provides a method to identify enrichment of functionally-related genes among those that are conditionally essential (i.e. significantly more or less essential between two conditions). The analysis is typically applied as post-processing step to the hits identified by a comparative analysis, such as resampling. Several analytical method are provided: Fisher’s exact test (FET, hypergeometric distribution), GSEA (Gene Set Enrichment Analysis) by Subramanian et al (2005), and Ontologizer. For Fisher’s exact test, genes in the resampling output file with adjusted p-value < 0.05 are taken as hits, and evaluated for overlap with functional categories of genes. The GSEA methods use the whole list of genes, ranked in order of statistical significance (without requiring a cutoff), to calculate enrichment.

Three systems of categories are provided for (but you can add your own): the Sanger functional categories of genes determined in the original annotation of the H37Rv genome (Cole et al, 1998, with subsequent updates), COG categories (Clusters of Orthologous Genes) and also GO terms (Gene Ontology). The supporting files for *M. tuberculosis* H37Rv are in the `src/pytransit/data/` directory.

For other organisms, it might be possible to download COG categories from [http://www.ncbi.nlm.nih.gov/COG/](http://www.ncbi.nlm.nih.gov/COG/) and GO terms from [http://www.geneontology.org](http://www.geneontology.org) or [http://patricbrc.org](http://patricbrc.org). If these files can be obtained for your organism, they will have to be converted into the `associations` file format described below. (The `pathways` files for COG categories and GO terms in the Transit data directory should still work, because they just encode pathways names for all terms/ids.)

At present, pathway enrichment analysis is only implemented as a command-line function, and is not available in the Transit GUI.

**Usage**

```bash
python3 src/transit.py pathway_enrichment <resampling_file> <associations> <pathways> ...
   <output_file> [-M <FET|GSEA|ONT>] [-PC <int>]
```

**Parameters**

- **Resampling File** The resampling file is the one obtained after using the resampling method in Transit. (It is a tab separated file with 11 columns.) GSEA method makes usage of the last column (adjusted P-value)
• **Associations File**  This is a tab-separated text file with 2 columns: pathway id, and pathway name. If a gene is in multiple pathways, the associated ids should be listed on separate lines. It is OK if there are no associations listed for some genes. Important: if pathways are hierarchical, you should expand this file to explicitly include associations of each gene with all parent nodes. Files with GO term associations will have to be pre-processed this way too.

Example: H37Rv_sanger_roles.dat

| Rv3823c | II.C.4 |
| Rv3823c | II.C  |
| Rv3823c | II    |
| Rv0337c | I.D.2  |
| Rv0337c | I.D   |
| Rv0337c | I     |
| ...     |

• **Pathways File**  This is a tab-separated text file with 2 columns: pathway id, and pathway name.

Example: sanger_roles.dat

| I     | Small-molecule metabolism |
| I.A   | Degradation               |
| I.A.1 | Carbon compounds          |
| I.A.2 | Amino acids and amines    |
| I.A.3 | Fatty acids               |
| I.A.4 | Phosphorous compounds     |
| ...   |

• **-M** Methodology to be used. FET is used by default (even without specifying -M).

**FET**  This implements Fisher’s Exact Test (hypergeometric distribution) to determine a p-value for each pathway, based on the proportion of pathway member observed in list of hits (conditionally essential gene by resampling, padj<0.05) compared to the background proportion in the overall genome, and p-values are adjusted post-hoc by the Benjamini-Hochberg procedure to limit the FDR to 5%.

In the output file, an “enrichment score” is reported, which is the ratio of the observed number of pathway members among the hits to the expected number. Pseudocounts of 2 are included in the calculation to reduce the bias toward small pathways with only a few genes; this can be adjusted with the -PC flag (below).

FET can be used with GO terms.

Additional flags for FET:

- `-PC <int>`: Pseudocounts used in calculating the enrichment score and p-value by hypergeometric distribution. Default: PC=2.

**GSEA**  Gene Set Enrichment Analysis. GSEA assess the significance of a pathway by looking at how the members fall in the ranking of all genes. The genes are first ranked by significance from resampling. Specifically, they are sorted by signed-log-p-value, SLPV=sign(LFC)*(log(pval)), which puts them in order so that the most significant genes with negative LFC are at the top, the most significant with positive LFC are at the bottom, and insignificant genes fall in the middle. Roughly, GSEA computes the mean rank of pathway members, and evaluates significance based on a simulated a null distribution. p-values are again adjusted at the end by BH.


GSEA can be used with GO terms.
Additional flags for GSEA:

- **-ranking SLPV|LFC**: method used to rank all genes; SLPV is signed-log-p-value (default); LFC is log2-fold-change from resampling
- **-p <float>**: exponent to use in calculating enrichment score; recommend trying ‘-p 0’ (default) or ‘-p 1’ (as used in Subramaniam et al, 2005)
- **-Nperm <int>**: number of permutations to simulate for null distribution to determine p-value (default=10000)

**ONT** Ontologizer is a specialized method for GO terms that takes parent-child relationships into account among nodes in the GO hierarchy. This can enhance the specificity of pathways detected as significant. (The problem is that there are many GO terms in the hierarchy covering similar or identical sets of genes, and often, if one node is significantly enriched, then several of its ancestors will be too, which obscures the results with redundant hits; Ontologizer reduces the significance of nodes if their probability distribution among hits can be explained by their parents.) Hierarchical relationships among GO terms are encoded in an OBO file, which is included in the src/pytransit/data/ directory.


**Auxiliary Pathway Files in Transit Data Directory**

These files for pathway analysis are distributed in the Transit data directory (e.g. transit/src/pytransit/data/).

<table>
<thead>
<tr>
<th>system</th>
<th>num cats</th>
<th>applicable methods</th>
<th>associations of genes with roles</th>
<th>pathway definitions/role names</th>
</tr>
</thead>
<tbody>
<tr>
<td>COG</td>
<td>20</td>
<td>FET*, GSEA</td>
<td>H37Rv_COG_roles.dat</td>
<td>COG_roles.dat</td>
</tr>
<tr>
<td>Sanger</td>
<td>153</td>
<td>FET*, GSEA*</td>
<td>H37Rv_sanger_roles.dat</td>
<td>sanger_roles.dat</td>
</tr>
<tr>
<td>GO</td>
<td>2545</td>
<td>FET, GSEA</td>
<td>H37Rv_GO_terms.txt</td>
<td>GO_term_names.dat</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ONT*</td>
<td>GO_terms_for_each_Rv.obo-3-11-18.txt</td>
<td>gene_ontology.1_2.3-11-18.obo</td>
</tr>
</tbody>
</table>

asterisk means ‘recommended’ combination of method with system of functional categories

**Current Recommendations**

Here are the recommended combinations of pathway methods to use for different systems of functional categories:

- For COG, use ‘-M FET’
- For Sanger roles, try both FET and GSEA
- For GO terms, use ‘M -ONT’

**Examples**

```
# uses Fisher's exact test by default (with PC=2 as pseudocounts)
> transit pathway_enrichment resampling_glyc_chol.txt $DATA/H37Rv_sanger_roles.dat
   ->$DATA/sanger_roles.dat pathways_glyc_chol_Sanger.txt

# can do this with GO terms too
> transit pathway_enrichment resampling_glyc_chol.txt $DATA/H37Rv_GO_terms.txt $DATA/
   ->GO_term_names.dat pathways_glyc_chol_GO.txt
```

(continues on next page)
2.5.12 tnseq_stats

You can generate the same table to statistics as on the Quality Control panel in the GUI from the command-line using the ‘tnseq_stats’ command. Here is an example:

```bash
> python3 src/transit.py tnseq_stats --help
usage: python3 src/transit.py tnseq_stats <file.wig> [ -o <output_file> ]
python3 src/transit.py tnseq_stats -c <combined_wig> [ -o <output_file> ]

> python3 src/transit.py tnseq_stats -c src/pytransit/data/cholesterol_glycerol_combined.dat

<table>
<thead>
<tr>
<th>dataset</th>
<th>density</th>
<th>mean_ct</th>
<th>NZmean</th>
<th>NZmedian</th>
<th>max_ct</th>
<th>total_cts</th>
<th>skewness</th>
<th>kurtosis</th>
</tr>
</thead>
<tbody>
<tr>
<td>src/pytransit/data/cholesterol_H37Rv_rep1.wig</td>
<td>0.44</td>
<td>139.6</td>
<td>317.6</td>
<td>147</td>
<td>125355.5</td>
<td>10414005.0</td>
<td>54.8</td>
<td>2324.7</td>
</tr>
<tr>
<td>src/pytransit/data/cholesterol_H37Rv_rep2.wig</td>
<td>0.44</td>
<td>171.4</td>
<td>390.5</td>
<td>148</td>
<td>704662.8</td>
<td>12786637.9</td>
<td>105.8</td>
<td>14216.2</td>
</tr>
<tr>
<td>src/pytransit/data/cholesterol_H37Rv_rep3.wig</td>
<td>0.36</td>
<td>173.8</td>
<td>484.2</td>
<td>171</td>
<td>292294.8</td>
<td>12968502.500000002</td>
<td>42.2</td>
<td>2328.0</td>
</tr>
<tr>
<td>src/pytransit/data/glycerol_H37Rv_rep1.wig</td>
<td>0.42</td>
<td>123.3</td>
<td>294.5</td>
<td>160</td>
<td>9195672.4</td>
<td>8.0</td>
<td>33.0</td>
<td></td>
</tr>
<tr>
<td>src/pytransit/data/glycerol_H37Rv_rep2.wig</td>
<td>0.52</td>
<td>123.8</td>
<td>240.1</td>
<td>127</td>
<td>9235984.2</td>
<td>4.0</td>
<td>33.5</td>
<td></td>
</tr>
</tbody>
</table>
```

2.5.13 corrplot

A useful tool when evaluating the quality of a collection of TnSeq datasets is to make a correlation plot of the mean insertion counts (averaged at the gene-level) among samples. While it is difficult to state unequivocally how much correlation there should be between samples from different conditions (or even between replicates of the same condition), the corrplot can often reveal individual samples which stand out as being far less correlated with all the others (which subsequently might be excluded from analyses).

Note: The corrplot command calls R, which must be installed on your system, and relies on the ‘corrplot’ R package. See Installation Instructions.
**Usage:**

```
python3 src/transit.py corrplot <mean_counts> <output.png> [-anova|-zinb]
```

The simplest usage is without the flags at the end. The mean_counts file is generated by the `export mean_counts` command, and gives the mean insertion count for each gene in each sample.

Here is an example of making a corrplot:

```
> transit corrplot glyc_chol_combined.wig.txt glyc_chol_corrplot.png
```

A corrplot can also be generated from the output of ANOVA or ZINB analysis, showing relationships among the conditions themselves (i.e. with replicates merged, rather than correlations among individual samples). Importantly, the correlations are based only on the *subset* of genes identified as significantly varying (Padj < 0.05) in order to enhance the patterns, since otherwise they would be washed out by the rest of the genes in the genome, the majority of which usually do not exhibit significant variation in counts.

Here is an example which generates the following image showing the corrplot among several different growth conditions:

```
> python3 src/transit.py corrplot anova_iron.txt iron_corrplot_anova.png -anova
```

correlations based on 229 genes
Note that is an ANOVA or ZINB output file (both of which contain mean counts for each gene in each condition) is supplied in place of mean_counts, the last argument of corrplot must be set to either `-anova` or `-zinb` to indicate the type of file being provided as the first argument.

Note: corrplot requires R (statistical analysis software) to be installed on your system. See Installation Instructions.

### 2.5.14 heatmap

The output of ANOVA or ZINB can be used to generate a heatmap that simultaneously clusters the significant genes and clusters the conditions, which is especially useful for shedding light on the relationships among the conditions apparent in the data.

**Note:** The heatmap command calls R, which must be installed on your system, and relies on the `gplots` R package. See Installation Instructions.

**Usage:**

```bash
cpython3 src/transit.py heatmap <anova_or_zinb_output> <heatmap.png> -anova|-zinb [-topk <int>] [-qval <float>]
```

Note that the first optional argument (flag) is required to be either `-anova` or `-zinb`, a flag to indicate the type of file being provided as the second argument.

By default, genes are selected for the heatmap based on qval<0.05. However, the user may change the selection of genes through 2 flags:

- `-qval <float>`: change qval threshold for selecting genes (default=0.05)
- `-topk <int>`: select top k genes ranked by significance (qval)

Here is an example which generates the following image showing the similarities among several different growth conditions:
Importantly, the heatmap is based only on the subset of genes identified as significantly varying (Padj < 0.05, typically only a few hundred genes) in order to enhance the patterns, since otherwise they would be washed out by the rest of the genes in the genome, the majority of which usually do not exhibit significant variation in counts.

2.6 Console Mode Cheat-Sheet

TRANSIT has the capability of running in Console mode, without depending on libraries for GUI elements. More hands-on users can utilize transit in this manner to quickly run multiple jobs in parallel. Below is brief

2.6.1 Analysis Methods

TRANSIT has the capacity of determining essentiality within a single condition, or between conditions to determine conditional essentiality.

Single Condition Essentiality

Analysis methods in a single condition require at least 4 positional arguments followed by optional flags.

```
python transit.py <method> <wig-files> <annotation> <output>
```

<table>
<thead>
<tr>
<th>Positional Arguments</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;method&gt;</td>
<td>Short name of the desired analysis method e.g. gumbel, resampling, hmm</td>
</tr>
<tr>
<td>&lt;wig-files&gt;</td>
<td>Comma-separated list of paths read-count datasets in .wig format</td>
</tr>
<tr>
<td>&lt;annotation&gt;</td>
<td>Path to the annotation in .prot_table or .GFF3 format.</td>
</tr>
<tr>
<td>&lt;output&gt;</td>
<td>Desired path and name of the output file</td>
</tr>
</tbody>
</table>

Example

```
python transit.py gumbel glycerol_H37Rv_rep1.wig,glycerol_H37Rv_rep2.wig H37Rv.prot_table glycerol_TTR.txt -r Sum -s 10000
```
## Conditional Essentiality

Analysis methods between two conditions require at least 5 positional arguments followed by optional flags.

<table>
<thead>
<tr>
<th>Positional Arguments</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;method&gt;</code></td>
<td>Short name of the desired analysis method e.g. gumbel, resampling, hmm</td>
</tr>
<tr>
<td><code>&lt;control-files&gt;</code></td>
<td>Comma-separated list of paths read-count files in .wig format for the control datasets</td>
</tr>
<tr>
<td><code>&lt;experimental-files&gt;</code></td>
<td>Comma-separated list of paths read-count files in .wif format for the experimental datasets</td>
</tr>
<tr>
<td><code>&lt;annotation&gt;</code></td>
<td>Path to the annotation in .prot_table or .GFF3 format.</td>
</tr>
<tr>
<td><code>&lt;output&gt;</code></td>
<td>Desired path and name of the output file</td>
</tr>
</tbody>
</table>

**Example**

```python
transit.py resampling glycerol_H37Rv_rep1.wig,glycerol_H37Rv_rep2.wig
 cholesterol_H37Rv_rep1.wig,cholesterol_H37Rv_rep2.wig H37Rv.prot_table glycerol_TTR.txt -n TTR -s 10000
```

### 2.6.2 Normalizing datasets

TRANSIT also allows users to normalize datasets and export them afterwards. To normalize datasets, 3 positional arguments followed by optional flags.

<table>
<thead>
<tr>
<th>Positional Arguments</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;wig-files&gt;</code></td>
<td>Comma-separated list of paths read-count datasets in .wig format</td>
</tr>
<tr>
<td><code>&lt;annotation&gt;</code></td>
<td>Path to the annotation in .prot_table or .GFF3 format.</td>
</tr>
<tr>
<td><code>&lt;output&gt;</code></td>
<td>Desired path and name of the output file</td>
</tr>
</tbody>
</table>

**Optional Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>-n &lt;String&gt;</td>
<td>Short name of the normalization method, e.g. -n TTR</td>
</tr>
</tbody>
</table>

**Example**

```python
transit.py norm glycerol_H37Rv_rep1.wig,glycerol_H37Rv_rep2.wig H37Rv.prot_table glycerol_TTR.txt -n TTR
```
2.7 Tutorial: Genetic Interactions Analysis

The feature implements the method described in the following publication:


To illustrate how TRANSIT can be used to analyze genetic interactions, we are going to go through a tutorial where we analyze datasets of *M. tuberculosis* in two different strain backgrounds (H37Rv, and delta-Rv2680 mutant), grown under two different conditions each (in-vivo day 0, and in-vivo day 32).

This tutorial provides instructions to accomplish the analysis in either Console Mode or GUI Mode.

2.7.1 Console Mode Tutorial

The Genetic Interactions (GI) method can be run in Console Mode without the need of the graphical interface. One can get an idea of how to run the method as well as the necessary and optional arguments by simply running the following command:

```python
python PATH/src/transit.py gi
```

In this tutorial, we’re analyzing datasets of *M. tuberculosis* in the H37Rv reference strain and a delta-Rv2680 mutant strain, grown under two different conditions each (in-vivo day 0, and in-vivo day 32). The method requires the comma-separated lists to the path of the .wig files.

- The first list corresponds to the Control datasets in the first condition (i.e. H37Rv at day 0)
- The second list corresponds to the Control datasets in the second condition (i.e. H37Rv at day 32)
- The third list corresponds to the Experimental datasets in the first condition (i.e. delta-Rv2680 at day 0)
- The fourth list corresponds to the Experimental datasets in the second condition (i.e. delta-Rv2680 at day 32)

Below is an example command to run this analysis, assuming the files are in the same directory where the files are located:

```python
python PATH/src/transit.py gi H37Rv_day0_rep1.wig,H37Rv_day0_rep2.wig H37Rv_day32_rep1.wig,H37Rv_day32_rep2.wig H37Rv_day32_rep3.wig Rv2680_day0_rep1.wig,Rv2680_day0_rep2.wig Rv2680_day32_rep1.wig,Rv2680_day32_rep2.wig Rv2680_day32_rep3.wig H37Rvprot_table results_gi_Rv2680.dat -s 10000
```

2.7.2 GUI Mode Tutorial

Run TRANSIT

Navigate to the directory containing the TRANSIT files, and run TRANSIT (or run ‘transit’ if you installed as a package):

```python
python PATH/src/transit.py
```
Adding the annotation file

Before we can analyze datasets, we need to add an annotation file for the organism corresponding to the desired datasets. Click on the file dialog button, on the top of the TRANSIT window (see image below), and browse and select the appropriate annotation file. Note: Annotation files must be in “.prot_table” or GFF3 format.

Adding datasets grown under condition A

The analysis of genetic interactions requires four sets of data. The TRANSIT interface allows for two sets of data at a time: Control and Experimental. For the purposes of genetic interactions, the initial set will represent the two strains under the first condition.

Adding the control datasets in condition A

In this context, the Control datasets in the first condition are the datasets for the H37Rv reference strain at day 0. To add these, we click on the control sample file dialog (see image below), and select the desired datasets (one by one). In this example, we have two replicates:
As we add the datasets they will appear in the table in the Control samples section. This table will provide the following statistics about the datasets that have been loaded so far: Total Number of Reads, Density, Mean Read Count and Maximum Count. These statistics can be used as general diagnostics of the datasets.

**Adding the experimental datasets in condition A**

We now repeat the process we did for control samples, for the experimental datasets. In this tutorial, the experimental datasets come from the KnockOut strain, delta-Rv2680, and the first condition, in-vivo day 0. To add these, we click on the experimental sample file dialog (see image below), and select the desired datasets (one by one). In this example, we have two replicates:
Running the Genetic Interactions method

We are now ready to proceed with the genetic interactions (GI) method in TRANSIT. In the menu bar, click on Analysis => [gi] to select the GI method. The panel on the right-hand side will populate with options. The analysis of genetic interactions requires four sets of data. After you are done setting these options are desired, click on the “Run GI” button. This will open a new window that will allow you to add the remaining two sets of data grown under the second condition:
Adding the control datasets in condition B

In this context, the Control datasets in the second condition (B) are the datasets for the H37Rv reference strain at day 0. To add these, we click on the control sample file dialog in the window that opened after clicking the “Run” button (see image below), and select the desired datasets (one by one). In this example, we have three replicates:
As we add the datasets they will appear in the table in the Control samples section. This table will provide the following statistics about the datasets that have been loaded so far: Total Number of Reads, Density, Mean Read Count and Maximum Count. These statistics can be used as general diagnostics of the datasets.

**Adding the experimental datasets in condition B**

We now repeat the process we did for control samples, for the experimental datasets. In this tutorial, the experimental datasets come from the Knock0ut strain, delta-Rv2680, and the first condition, in-vivo day 32. To add these, we click on the experimental sample file dialog (see image below), and select the desired datasets (one by one). In this example, we have two replicates:
Viewing GI results

Once TRANSIT finishes running, the results file will automatically be added to the Results Files section at the bottom of the window. To view the actual results, we can open the file in a new window by selecting it from the list and clicking on the “Display Table” button.

The newly opened window will display a table of the results. We can sort the results by clicking on the column header. For example, to focus on the genes that are most likely to be interacting with Rv2680 (i.e. the gene Knocked out) we can click on the column header labeled “Type of Interaction”, which represents the final classification by the GI method.

You can also sort by “Mean delta logFC” to see the estimated change in enrichment between the two strains and conditions.

2.8 Tutorial: Normalize datasets

TRANSIT has the capability to normalize datasets with different methods, and export them to IGV from the Broad Institute or a CombinedWig format. This tutorial shows a quick overview of how to normalize datasets save them using the GUI mode of transit or through the Console mode.
2.8.1 Adding the annotation file

Before we can normalize .wig datasets, we need to add an annotation file for the organism. Click on the file dialog button, on the top of the TRANSIT window (see image below), and browse and select the appropriate annotation file. Note: Annotation files must be in “.prot_table” or GFF3 format, described above:

![Image of TRANSIT window with annotation file selection]

2.8.2 Add .wig datasets

Next we must choose to add .wig formatted datasets what we wish to normalize to CombinedWig format. To add these, we click on the control sample file dialog (see image below), and select the desired datasets (one by one). In this example, we have two replicates:
As we add the datasets they will appear in the table below. Select the datasets you wish to normalize.

### 2.8.3 Normalize and Save

After you have selected the desired datasets in the list of datasets added, click on “Export -> Selected Datasets” in the menu bar at the top of the TRANSIT window, and select the format you desire (e.g. “to IGV” or “to CombinedWig”). You will be prompted to pick a normalization method, and a filename. Note: Only selected datasets (“Control+Click”) will be normalized and saved.
2.8.4 Normalization

Proper normalization is important as it ensures that other sources of variability are not mistakenly treated as real differences in datasets. TRANSIT provides various normalization methods, which are briefly described below:

- **TTR**: Trimmed Total Reads (TTR), normalized by the total read-counts (like totreads), but trims top and bottom 5% of read-counts. This is the recommended normalization method for most cases as it has the benefit of normalizing for difference in saturation in the context of resampling.

- **nzmean**: Normalizes datasets to have the same mean over the non-zero sites.

- **totreads**: Normalizes datasets by total read-counts, and scales them to have the same mean over all counts.

- **zinfnb**: Fits a zero-inflated negative binomial model, and then divides read-counts by the mean. The zero-inflated negative binomial model will treat some empty sites as belonging to the “true” negative binomial distribution responsible for read-counts while treating the others as “essential” (and thus not influencing its parameters).

- **quantile**: Normalizes datasets using the quantile normalization method described by Bolstad et al. (2003). In this normalization procedure, datasets are sorted, an empirical distribution is estimated as the mean across the sorted datasets at each site, and then the original (unsorted) datasets are assigned values from the empirical distribution based on their quantiles.

- **betageom**: Normalizes the datasets to fit an “ideal” Geometric distribution with a variable probability parameter $p$. Specially useful for datasets that contain a large skew.

- **nonorm**: No normalization is performed.
2.9 Tutorial: Export datasets

TRANSIT has the capability to export .wig files into different formats. This tutorial shows a quick overview of how to export to the IGV format. This can be useful to be able to import read-count data into IGV from the Broad Institute and use its visualization capabilities.

2.9.1 Adding the annotation file

Before we can export .wig datasets to IGV format, we need to add an annotation file for the organism. Click on the file dialog button, on the top of the TRANSIT window (see image below), and browse and select the appropriate annotation file. Note: Annotation files must be in “.prot_table” or GFF3 format, described above:

2.9.2 Add .wig datasets

Next we must choose to add .wig formatted datasets what we wish to export to IGV format. To add these, we click on the control sample file dialog (see image below), and select the desired datasets (one by one). In this example, we have two replicates:
As we add the datasets they will appear in the table below.

### 2.9.3 Export to IGV

Finally, to export the datasets we click on “Export” in the menu bar at the top of the TRANSIT window, and select the option that matches which datasets we wish to export. Note: Only selected datasets (“Control+Click”) will be exported.
2.10 TPP Overview

TPP is a software tool for processing raw reads (e.g. .fastq files, untrimmed) from an Tn-Seq experiment, extracting counts of transposon insertions at individual TA dinucleotides sites in a genome (“read counts”, or more specifically “template counts”, see below), and writing this information out in .wig format suitable for input to TRANSIT. In addition, TPP calculates some useful statistics and diagnostics on the dataset.

There are many way to do pre-processing of Tn-Seq datasets, and it can depend on the the protocol used for Tn-Seq, the conventions used by the sequencing center, etc. However, TPP is written to accommodate the most common situation among our collaborating labs. In particular, it is oriented toward the Tn-Seq protocol developed in the Sassetti lab and described in (Long et al, 2015), which uses a barcoding system to uniquely identifying reads from distinct transposon-junction DNA fragments. This allows raw read counts to be reduced to unique template counts, eliminating effects of PCR bias. The sequencing must be done in paired-end (PE) mode (with a minimum read-length of around 50 bp). The transposon terminus appears in the prefix of read1 reads, and barcodes are embedded in read2 reads.

The suffixes of read1 and read2 contain nucleotides from the genomic region adjacent to the transposon insertion. These subsequences must be mapped into the genome. TPP uses BWA (Burroughs-Wheeler Aligner) to do this mapping. It is a widely-used tool, but you will have to install it on your system. Mapping large datasets takes time, on the order of 15 minutes (depending on many factors), so you will have to be patient.

Subsequent to the BWA mapping step, TPP does a bunch of post-processing steps. Primarily, it tabulates raw read counts at each TA site in the reference genome, reduces them to template counts, and writes this out in .wig format (as input for TRANSIT). It also calculates and reports some statistics on the dataset which a useful for diagnostic purposes. These are saved in local file caled “.tn_stats”. The GUI automatically reads all the .tn_stats files from...
previously processed datasets in a directory and displays them in a table.

The GUI interface is set-up basically as a graphical front-end that allows you to specify input files and parameters to get a job started. Once you press START, the graphical window goes away, and the pre-processing begins, printing out status messages in the original terminal window. You can also run TPP directly from the command-line with the GUI, by providing all the inputs via command-line arguments.

TPP has a few optional parameters in the interface. We intend to add other options in the future, so if you have suggestions, let us know. In particular, if you have some datasets that require special processing (such as if different primer sequences were used for PCR amplification, or a different barcoding system, or different contaminant sequences to search for, etc.), we might be able to add some options to deal with this.

2.11 Installation

TPP should work equivalently on Macs, PCs running Windows, or Unix machines. TPP is fundamentally a python script that has a graphical user interface (GUI) written in wxPython. Its major dependency is that it calls BWA to map reads. TPP is packaged as a part of TRANSIT.

See: Transit/TPP installation

Requirements (in addition to TRANSIT requirement):

- BWA version 0.7.12 (can put this directory anywhere; be sure to run ‘make’ to build bwa executable pre-compiled version for 64-bit Windows)

Since TPP is a python script, there is nothing to compile or ‘make’.

2.12 Running TPP

TPP may be run from the command line (e.g. of a terminal window or shell) by typing:

```python
python PATH/src/tpp.py
```

# If installed as PyPI package
tpp

where PATH is the path to the TRANSIT installation directory. This should pop up the GUI window, looking like this…
Note, TPP can process paired-end reads, as well as single-end datasets. (just leave the filename for read2 blank)

The main fields to fill out in the GUI are...

- **Reference genome** - This is the sequence in Fasta format against which the reads will be mapped. The reference genome may contain multiple contigs (hence a ‘multi-fasta’ file, with multiple headers starting with ‘>’), or in fact may include a comma-separated list of fasta files.

- **Replicon ids** - If your genome sequence has only one contig (the usual case), you don’t have to do anything here (leave blank). If you have multiple contigs (e.g. multiple chromosomes, or plasmids included, etc.), you can give them unique labels/ids as a comma-separated list. This will be used as filename suffixes for the output .wig files (a separate file with insertion counts at TA sites for each replicon). If you have many (anonymous) contigs, e.g. from a de novo assembly, you can enter ‘auto’ in this field, and it will generate numerical ids for filename suffixes, 1,2...n for however many contigs are in the file.

- **Reads1 file** - This should be the raw reads file (untrimmed) for read1 in FASTQ or FASTA format, e.g. DATASET_NAME_R1.fastq
  
  – Note: you can also supply gzipped files for reads, e.g. *.fastq.gz

- **Reads2 file** - this should be the raw reads file (untrimmed) for read2 in FASTQ or FASTA format, e.g. DATASET_NAME_R2.fastq
  
  – Note: if you leave read2 blank, it will process the dataset as single-ended. Since there are no barcodes, each read will be counted as a unique template.
• **Prefix** - base to use for output filenames (for the multiple intermediate files that will get generated in the process)

• **Protocol used** - Currently, the following TnSeq sample prep protocols are supported. These set the default transposon and primer sequence that are typically used with each protocol (which can be overridden by -primer), and have a few minor differences in processing reads.

• **Sassetti** - DNA is sheared into fragments, sequencing adapters are ligated, and then transposon:genomic junctions are amplified by PCR. Thus a portion (~20bp) of the Himar1 terminus appears as a prefix in the reads, which is stripped off prior to mapping to genome. Also, read 2 contains a random nucleotide barcode, which is used to reduce read counts at TA sites to unique template counts, which reduces noise. See Long et al. (2015).

• **Tn5** - This is a different transposon than Himar1. The main difference between Tn5 and Himar1 is that it is not restricted to insertions at TA dinucleotides, and can insert randomly anywhere in the genome. In principle, this could result in higher saturation (more insertions per gene; insertions at hundreds of thousands of sites are common). Thus the .wig files generated list insertion counts at every coordinate genome-wide (not just TA sites), though most counts are still 0. Analysis of Tn5 datasets has some unique challenges, which are discussed in Transit Methods. See Langridge et al. (2009).

• **Mme1** - This can be used with a variant of the Himar1 transposon, but the method of selecting and amplifying transposon:genomic junctions is different. The Mme1 (or MmeI) restriction enzyme is used to recognize a site in the terminus of the transposon, and makes a cut 18-20bp downstream into the genomic region. Thus the reads are much shorter, and also there is no need for read 2 (these are typically single-ended datasets). For now, most users pre-trim their raw reads down to 16-20bp by using another tool to strip off the transposon prefix and adapter suffix. If you do this, you should set your primer sequence to “” in TPP. See Santiago et al. (2015).

• **Transposon used** - Himar1 is assumed by default, but you can set it to Tn5 to process libraries of that type. The main consequences of this setting are: 1) the selected transposon determines the nucleotide prefix to be recognized in read 1, and 2) for Himar1, reads are counted only at TA sites, whereas for Tn5, reads are counted at ALL sites in the genome (since it does not have significant sequence specificity) and written out in the .counts and .wig files.

• **Primer sequence** - This represents the end of the transposon that appears as a constant prefix in read 1 (possibly shifted by a few random bases), resulting from amplifying transposon:genomic junctions. TPP searches for this prefix and strips it off, to map the suffixes of reads into the genome. TPP has default sequences defined for both Himar1 and Tn5 data, based on the most commonly used protocols (Long et al. (2015); Langridge et al. (2009)). However, if you amplify junctions with a different primer, this field gives you the opportunity to change the sequence TPP searches for in each read. Note that you should not enter the ENTIRE primer sequence, but rather just the part of the primer sequence that will show up at the beginning of every read. If you preprocess your reads by trimming off the 5’ transposon prefixes, you could set this to blank, and TPP will process all your reads; but we don’t recommend doing it this way.

• **Max reads** - Normally, leave this blank by default, and TPP will process all reads. However, if you want to do a quick run on a subset of the data, you can select a smaller number. This is mainly for testing purposes.

• **Mismatches** - this is for searching for the sequence patterns in reads corresponding to the transposon prefix in R1 and the constant adapter sequences surrounding the barcode in R2; we suggest using a default value of 1 mismatch

• **Primer start window** - a pair of integers separated by a comma (P,Q), which constrains the location in the read to search for the start of the primer sequence; default is set to 0,20 (which is typically where it will be found for samples prepared using the Sassetti protocol, i.e. near the beginning of reads, with some small random shifts)

• **BWA executable** - you’ll have to find the path to where the executable is installed

• **BWA algorithm** - there are 2 options: ‘aln’ and ‘mem’. ‘aln’ was originally used in Transit, but the default has now been switched to ‘mem’, which should be able to map more reads

• **BWA flags** - if you want to pass through options to BWA

• **BarSeq Catalog** - this is not finished yet, but we are working on it. Stay tuned...
Once you have filled all these fields out, you can press START (or QUIT). At this point the GUI window will disappear, and the data processing commences in the original terminal/shell windows. It prints out a lot of information to let you know what it is doing (and error messages, if anything goes wrong). Many intermediate files get generated. It takes awhile (like on the order of 15 minutes), most of which is taken up by the mapping-reads step by BWA.

Subsequent to the BWA mapping step, TPP does a bunch of post-processing steps. Primarily, it tabulates raw read counts at each TA site in the reference genome, reduces them to template counts, and writes this out in .wig format (as input for essentiality analysis in TRANSIT). It also calculates and reports some statistics on the dataset which a useful for diagnostic purposes. These are saved in local file called “.tn_stats”. The GUI automatically reads all the .tn_stats files from previously processed datasets in a directory and displays them in a table.

TPP uses a local config file called “tpp.cfg” to remember parameter settings from run to run. This makes it convenient so that you don’t have to type in things like the path to the BWA executable or reference genome over and over again. You just have to do it once, and TPP will remember.

**Command-line mode:** TPP may be run on a dataset directly from the command-line without invoking the user interface (GUI) by providing it filenames and parameters as command-line arguments.

```
> python tpp.py --help
usage: python PATH/src/tpp.py -bwa <EXECUTABLE_WITH_PATH> -ref <fasta-file|commaSeparated_list> -reads1 <FASTQ_OR_FASTA_FILE> [-reads2 <FASTQ_OR_FASTA_FILE>] -output <BASE_FILENAME> [OPTIONAL ARGS]
OPTIONAL ARGS:

-protocol [Sassetti|Tn5|Mme1]  # which sample prep protocol was used?; sasseti
-protocol is the default; this sets the default transposon and primer sequence

-primer <seq>  # prefix of reads corresponding to end of transposon at junction

-maxreads <INT>

-mismatches <INT>  # when searching for constant regions in reads 1 and 2; default is 1

-flags "<STRING>"  # args to pass to BWA

-primer-start-window INT,INT  # position in read to search for start of primer; default is [0,20]

-window-size INT  # automatic method to set window

-barseq_catalog_in|-barseq_catalog_out <file>

-replicon-ids <comma_separated_list_of_names>  # if multiple replicons/genomes/contigs/

sequences were provided in -ref, give them names.

Enter 'auto' for autogenerated ids.
```

The input arguments and file types are as follows:
<table>
<thead>
<tr>
<th>Flag</th>
<th>Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>-bwa</td>
<td>path executable</td>
<td></td>
</tr>
<tr>
<td>-bwa-alg</td>
<td>‘mem’ (default) or ‘aln’ (the old way)</td>
<td></td>
</tr>
<tr>
<td>-flag</td>
<td>parameters to pass to BWA</td>
<td></td>
</tr>
<tr>
<td>-ref</td>
<td>reference genome sequence</td>
<td>FASTA file or comma-separated list of files</td>
</tr>
<tr>
<td>-replicon-ids</td>
<td>comma-separated list of names to use for contigs</td>
<td>necessary only if genome seq has multiple contigs. Enter ‘auto’ for autogenerating ids.</td>
</tr>
<tr>
<td>-reads1</td>
<td>file of read 1 of paired reads</td>
<td>FASTA or FASTQ format (or gzipped)</td>
</tr>
<tr>
<td>-reads2</td>
<td>file of read 2 of paired reads (optional for single-end reads)</td>
<td>FASTA or FASTQ format (or gzipped)</td>
</tr>
<tr>
<td>-output</td>
<td>base filename to use for output files</td>
<td></td>
</tr>
<tr>
<td>-maxreads</td>
<td>subset of reads to process (optional)</td>
<td>default is to use all reads</td>
</tr>
<tr>
<td>-mismatches</td>
<td>how many to allow when searching reads for sequence patterns</td>
<td>default is 1 mismatch</td>
</tr>
<tr>
<td>-protocol</td>
<td>Sassetti (default)</td>
<td>The Sassetti sample prep protocol (with barcodes in read2). Assumes Himar1 transposon.</td>
</tr>
<tr>
<td>Tn5</td>
<td></td>
<td>Reads can map to any site, not just TAs.</td>
</tr>
<tr>
<td>Mme1</td>
<td></td>
<td>Use of restriction enzyme recognizing the terminus of the Himar1 transposon.</td>
</tr>
<tr>
<td>-primer</td>
<td>nucleotide sequence</td>
<td>Constant prefix of reads that TPP searches for. default: ACT-TATCAGGCAACCTGTTA (terminus of Himar1)</td>
</tr>
<tr>
<td>-primer-start-window</td>
<td>INT,INT (default is 0,20)</td>
<td>Start and end nucleotides in read 1 in which to search for start of Tn prefix.</td>
</tr>
</tbody>
</table>

(Note: if you have already run TPP once, the you can leave out the specification of the path for BWA, and it will automatically take the path stored in the config file, tpp.cfg. Same for ref, if you always use the same reference sequence.)

(The -primer-start-window flag specifies the range of nucleotide in read 1 to search for the start of the primer sequence (which is the end of the transposon). This is useful to narrow the down the region to search from the whole read (especially if the primer sequence is short, e.g. <10bp), to avoid spurious matches in reads not representing true transposon:genomic junctions. Depending on the protocol and primer design, the constant sequence corresponding the the end of the transposon usually occurs near the beginning of the read, possibly at varying (shifted) positions. However, if your primer sequence is long enough (e.g >16bp), then the changes of spurious matches (e.g. to the reference genome) is quite low.)

### 2.13 Mapping to Genomes with Multiple Contigs

Occasionally, it is useful to process TnSeq data where the reference genome consists of multiple sequences, such as multiple chromosomes (e.g. *Vibrio cholera*), or a chromosome + plasmid, or it might be a new strain with an incomplete assembly (multiple contigs not yet assembled into a single continuous scaffold). While TPP was originally designed for mapping reads to one sequence at a time, it has recently been extended to process multiple contigs in parallel (with help from Robert Jenquin and William Matern).

You can provide either a single merged reference sequence (multi-fasta file, with several header lines and sequences), or a comma-separated list of input fasta files (command-line only). If multiple sequences are provided to TPP, you
will have to include an additional flag on the command line called `-replicon-ids` (again, a comma-separated list; the number of ids needs to match the number of input sequences. Use ‘auto’ to autogenerate ids).

In the GUI, there is a new field for specifying replicon-ids as well. If there is just one sequence or contig, you can leave replicon-ids blank; you do not have to specify it in the GUI or on the command line.

In such situations, TPP will generate multiple .wig files, each with the base filename (arg of ‘-output’ flag), suffixed with a replicon-id.

For example, consider the following example command:

```bash
> python tpp.py python -bwa ../bwa-0.7.12/bwa -ref avium104.fna,pMAH135.fna -
   ->replicon-ids avium104,pMAH135 -reads1 TnSeq-avium-7H10-A1_R1.fastq -reads2 TnSeq-
   avium-7H10-A1_R2.fastq -output TnSeq-avium-7H10-A1
```

This command would generate output these files:


### 2.14 Overview of Data Processing Procedure

Here is a brief summary of the steps performed in converting raw reads (.fastq files) into template counts:

1. Convert .fastq files to .fasta format (.reads).

2. Identify reads with the transposon prefix in R1. The sequence searched for is ACTTATCAGCCAACCTGTTA (or TAAGAGACAG for Tn5), which must start between cycles 5 and 10 (inclusive). (Note that this ends in the canonical terminus of the Himar1 transposon, TGTTA.) The “staggered” position of this sequence is due to insertion a few nucleotides of variable length in the primers used in the Tn-Seq sample prep protocol (e.g. 4 variants of Sol_AP1_57, etc.). The number of mismatches allowed in searching reads for the transposon sequence pattern can be adjusted as an option in the interface; the default is 1.

3. Extract genomic part of read 1. This is the suffix following the transposon sequence pattern above. However, for reads coming from fragments shorter than the read length, the adapter might appear at the other end of R1, TACCACGACCA. If so, the adapter suffix is stripped off. (These are referred to as “truncated” reads, but they can still be mapped into the genome just fine by BWA.) The length of the genomic part must be at least 20 bp.

4. Extract barcodes from read 2. Read 2 is searched for GATGGCCGGTGGATTTGTGnnnnnnnnnnTGGTCGTGGTAT”. The length of the barcode is typically 10 bp, but can be variable, and must be between 5-15 bp.

5. Extract genomic portions of read 2. This is the part following TGGTCGTGGTAT…. It is often the whole suffix of the read. However, if the read comes from a short DNA fragment that is shorter than the read length, the adapter on the other end might appear, in which case it is stripped off and the nucleotides in the middle representing the genomic insert, TGGTCGTGGTATnnnnnnnnnnTGGTCGTGGTAT”. The insert must be at least 20 bp long (inserts shorter than this are discarded, as they might map to spurious locations in the genome).

6. Map genomic parts of R1 and R2 into the genome using BWA. Mismatches are allowed, but indels are ignored. No trimming is performed. BWA is run in ‘sampe’ mode (treating reads as pairs). Both reads of a pair must map (on opposite strands) to be counted.

7. Count the reads mapping to each TA site in the reference genome (or all sites for Tn5).

8. Reduce raw read counts to unique template counts. Group reads by barcode AND mapping location of read 2 (aka fragment “endpoints”).

10. Calculate statistics like insertion\_density and NZ\_mean. Look for the site with the max template count. Look for reads matching the primer or vector sequences.

## 2.15 Statistics

See also: Transit Quality Control

Here is an explanation of the statistics that are saved in the .tn_stats file and displayed in the table in the GUI. For convenience, all the statistics are written out on one line with tab-separation at the of the .tn_stats file, to make it easy to add it as a row in a spreadsheet, as some people like to do to track multiple datasets.

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>total_reads</td>
<td>total number of reads in the original .fastq/.fasta</td>
</tr>
<tr>
<td>truncated_reads</td>
<td>reads representing DNA fragments shorter than the read length; adapter appears at end of read 1 and is stripped for mapping</td>
</tr>
<tr>
<td>TGTTA_reads</td>
<td>number of reads with a proper transposon prefix (ending in TGTTA in read1)</td>
</tr>
<tr>
<td>reads1_mapped</td>
<td>number of R1 mapped into genome (independent of R2)</td>
</tr>
<tr>
<td>reads2_mapped</td>
<td>number of R2 mapped into genome (independent of R1)</td>
</tr>
<tr>
<td>mapped_reads</td>
<td>number of reads which mapped into the genome (requiring both read1 and read2 to map)</td>
</tr>
<tr>
<td>read_count</td>
<td>total reads mapping to TA sites (mapped reads excluding those mapping to non-TA sites)</td>
</tr>
<tr>
<td>template_count</td>
<td>reduction of mapped reads to unique templates using barcodes</td>
</tr>
<tr>
<td>template_ratio</td>
<td>read_count / template_count</td>
</tr>
<tr>
<td>TA_sites</td>
<td>total number of TA dinucleotides in the genome</td>
</tr>
<tr>
<td>TA_hit</td>
<td>number of TA sites with at least 1 insertion</td>
</tr>
<tr>
<td>insertion_density</td>
<td>TAs_hit / TA_sites</td>
</tr>
<tr>
<td>max_count</td>
<td>the maximum number of templates observed at any TA site</td>
</tr>
<tr>
<td>max_site</td>
<td>the coordinate of the site where the max count occurs</td>
</tr>
<tr>
<td>NZ_mean</td>
<td>mean template count over non-zero TA sites</td>
</tr>
<tr>
<td>FR_corr</td>
<td>correlation between template counts on Fwd strand versus Rev strand</td>
</tr>
<tr>
<td>BC_corr</td>
<td>correlation between read counts and template counts over non-zero sites</td>
</tr>
<tr>
<td>primer_matches</td>
<td>how many reads match the Himar1 primer sequence (primer-dimer problem in sample prep)</td>
</tr>
<tr>
<td>vector_matches</td>
<td>how many reads match the phiMycoMarT7 sequence (transposon vector) used in Tn mutant library construction</td>
</tr>
<tr>
<td>adapter</td>
<td>how many reads match the Illumina adapter (primer-dimers, no inserts).</td>
</tr>
<tr>
<td>misprimed</td>
<td>how many reads match the Himar1 primer but lack the TGTTA, meaning they primed at random sites (non-Tn junctions)</td>
</tr>
</tbody>
</table>

Here is an example of a .tn_stats file:

```bash
# title: Tn-Seq Pre-Processor
# date: 08/03/2016 13:01:47
# command: python ../../src/tpp.py -bwa /pacific/home/ioerger/bwa-0.7.12/bwa -ref H37Rv.fna -reads1 TnSeq_H37Rv_CB_1M_R1.fastq -reads2 TnSeq_H37Rv_CB_1M_R2.fastq -output TnSeq_H37Rv_CB
# transposon type: Himar1
# read1: TnSeq_H37Rv_CB_1M_R1.fastq
# read2: TnSeq_H37Rv_CB_1M_R2.fastq
# ref_genome: H37Rv.fna
# total_reads 1000000 (or read pairs)
```

(continues on next page)
Interpretation: To assess the quality of a dataset, I would recommend starting by looking at these primary statistics:

1. **mapped reads**: should be on the order of several million; mapped reads; if there is a significant reduction from total reads, look at reads1_mapped and reads2_mapped and truncated reads to figure what might have gone wrong; you might try allowing 2 mismatches

2. **primer/vector matches**: check whether a lot of the reads might be matching the primer or vector sequences; if they match the vector, it suggests your library still has phage contamination from the original infection; if there are a lot of primer reads, these probably represent “primer-dimers”, which could be reduced by improving fragment size selection during sample prep.

3. **insertion density**: good libraries should have insertions in at least ~35% of TA sites for meaningful statistical analysis

4. **NZ_mean**: good datasets should have a mean of around 50 templates per site for sufficient dynamic range

If something doesn’t look right, the other statistics might be helpful in figuring out what went wrong. If you see a significant reduction in reads, it could be due to some poor sequencing cycles, or using the wrong reference genome, or a contaminant of some type. Some attrition is to be expected (loss of maybe 10-40% of the reads). The last 2 statistics indicate 2 common cases: how many reads match the primer or vector sequences. Hopefully these counts will be low, but if they represent a large fraction of your reads, it could mean you have a problem with your sample prep protocol or Tn mutant library, respectively.

Comments or Questions?

TPP was developed by Thomas R. Ioerger at Texas A&M University. If you have any comments or questions, please feel free to send me an email at: ioerger@cs.tamu.edu
2.16 transit package

2.16.1 Submodules

2.16.2 pytransit.norm_tools module

class pytransit.norm_tools.AdaptiveBGCNorm
    Bases: pytransit.norm_tools.NormMethod

cleanupgeom(\(\rho\))
    Returns a ‘clean’ output from the geometric distribution.

cdf(\(x\))
    Calculates an empirical CDF of the given data.

name = ‘aBGC’

static normalize(data, wigList=[], annotationPath=”, doTotReads=True, bgsamples=200000)
    Returns the normalized data using the aBGC method.

Parameters
    • data (numpy array) – (K,N) numpy array defining read-counts at N sites for K datasets.
    • doTotReads (bool) – Boolean specifying whether to do TTR normalization as well.
    • bgsamples (int) – Integer specifying how many samples to take.

Returns Array with the normalized data.

Return type numpy array

Example

```python
>>> import pytransit.norm_tools as norm_tools
>>> import pytransit.tnseq_tools as tnseq_tools

>>> (data, position) = tnseq_tools.get_data(“transit/data/glycerol_H37Rv_rep1.wig”, “transit/data/glycerol_H37Rv_rep2.wig”)

>>> print(data)
array([[ 0., 0., 0., ..., 0., 0., 0.],
[ 0., 0., 0., ..., 0., 0., 0.]]

>>> normdata = norm_tools.aBGC_norm(data)

>>> print(normdata)
array([[ 0., 0., 0., ..., 0., 0., 0.],
[ 0., 0., 0., ..., 0., 0., 0.]]
```

See also:

normalize_data

class pytransit.norm_tools.BetaGeomNorm
    Bases: pytransit.norm_tools.NormMethod

cleanupgeom(\(\rho\))
    Returns a ‘clean’ output from the geometric distribution.

cdf(\(x\))
    Calculates an empirical CDF of the given data.
name = 'betageom'

static normalize(data, wigList=[], annotationPath='', doTTR=True, bgsamples=200000)

Returns normalized data according to the BGC method.

Parameters

- **data** (numpy array) – (K,N) numpy array defining read-counts at N sites for K datasets.
- **doTTR** (bool) – Boolean specifying whether to do TTR norm as well.
- **bgsamples** (int) – Integer specifying how many samples to take.

Returns Array with the data normalized using the betageom method.

Return type numpy array

Example

```python
>>> import pytransit.norm_tools as norm_tools
>>> import pytransit.tnseq_tools as tnseq_tools

>>> (data, position) = tnseq_tools.get_data(['transit/data/glycerol_˓→H37Rv_rep1.wig', 'transit/data/glycerol_H37Rv_rep2.wig'])

>>> print(data)
array([[ 0., 0., 0., ..., 0., 0., 0.],
       [ 0., 0., 0., ..., 0., 0., 0.]])

>>> normdata = norm_tools.betageom_norm(data)

>>> print(normdata)
[[ 0. 0. 0. ..., 0. 0. 0.]
 [ 0. 0. 0. ..., 0. 0. 0.]]
```

See also:

normalize_data

class pytransit.norm_tools.EmpHistNorm

Bases: pytransit.norm_tools.NormMethod

static Fzinfnb(params, args)

Objective function for the zero-inflated NB method.

name = 'emphist'

static normalize(data, wigList=[], annotationPath='')

Returns the normalized data, using the empirical hist method.

Parameters

- **wigList** (list) – List of paths to wig formatted datasets.
- **annotationPath** (str) – Path to annotation in .prot_table or GFF3 format.

Returns Array with the normalization factors for the emphist method.

Return type numpy array

Example

```python
>>> import pytransit.norm_tools as norm_tools
>>> import pytransit.tnseq_tools as tnseq_tools

>>> (data, position) = tnseq_tools.get_data(['transit/data/glycerol_˓→H37Rv_rep1.wig', 'transit/data/glycerol_H37Rv_rep2.wig'])

>>> print(data)

(continues on next page)
array([[ 0., 0., 0., ..., 0., 0., 0.],
       [ 0., 0., 0., ..., 0., 0., 0.]]))
>>> factors = norm_tools.emphist_factors(['transit/data/glycerol_→H37Rv_rep1.wig',
                                               'transit/data/glycerol_H37Rv_rep2.wig'],
                                               'transit/genomes/H37Rv.prot_table')
>>> print(factors)
array([[ 1. ],
       [ 0.63464722]])

See also:

normalize_data

pytransit.norm_tools.Fzinfnb(params, args)
Objective function for the zero-inflated NB method.

class pytransit.norm_tools.NZMeanNorm
   Bases: pytransit.norm_tools.NormMethod
       name = 'nzmean'
       static normalize(data, wigList=[], annotationPath='')
       Returns the normalization factors for the data, using the NZMean method.

Parameters

   data (numpy array) – (K,N) numpy array defining read-counts at N sites for K datasets.

Returns

   Array with the normalization factors for the nzmean method.

Return type

   numpy array

Example

>>> import pytransit._tools.norm_tools as norm_tools
>>> import pytransit.tnseq_tools as tnseq_tools

>>> (data, position) = tnseq_tools.get_data(['transit/data/glycerol_→H37Rv_rep1.wig',
                                             'transit/data/glycerol_H37Rv_rep2.wig'])
>>> print(data)
array([[ 0., 0., 0., ..., 0., 0., 0.],
       [ 0., 0., 0., ..., 0., 0., 0.]]))
>>> factors = norm_tools.nzmean_factors(data)
>>> print(factors)
array([[ 1.14836149],
       [ 0.88558737]])

See also:

normalize_data

class pytransit.norm_tools.NoNorm
   Bases: pytransit.norm_tools.NormMethod
       name = 'nonorm'
       static normalize(data, wigList=[], annotationPath='')

class pytransit.norm_tools.NormMethod

       name = 'undefined'
       static normalize()
```
class pytransit.norm_tools.QuantileNorm
    Bases: pytransit.norm_tools.NormMethod
    name = 'quantile'
    static normalize(data, wigList=[], annotationPath="")
    Performs Quantile Normalization as described by Bolstad et al. 2003
    Parameters
data (numpy array) – (K,N) numpy array defining read-counts at N sites for K datasets.
    Returns Array with the data normalized by the quantile normalization method.
    Return type numpy array
    Example
        >>> import pytransit.norm_tools as norm_tools
        >>> import pytransit.tnseq_tools as tnseq_tools
        >>> (data, position) = tnseq_tools.get_data(['transit/data/glycerol_→H37Rv_rep1.wig', 'transit/data/glycerol_H37Rv_rep2.wig'])
        >>> print(data)
        array([[ 0., 0., 0., ..., 0., 0., 0.],
               [ 0., 0., 0., ..., 0., 0., 0.]]
        >>> normdata = norm_tools.quantile_norm(data)
        >>> print(normdata)

See also:
    normalize_data

class pytransit.norm_tools.TTRNorm
    Bases: pytransit.norm_tools.NormMethod
    empirical_theta()
    Calculates the observed density of the data.
    Parameters
data (numpy array) –
    (N) numpy array defining read-counts at N sites.
    Returns Density of the given dataset.
    Return type float
    Example
        >>> import pytransit.tnseq_tools as tnseq_tools
        >>> import pytransit.norm_tools as norm_tools
        >>> (data, position) = tnseq_tools.get_data(['transit/data/glycerol_→H37Rv_rep1.wig', 'transit/data/glycerol_H37Rv_rep2.wig'])
        >>> print(data)
        array([[ 0., 0., 0., ..., 0., 0., 0.],
               [ 0., 0., 0., ..., 0., 0., 0.]]
        >>> theta = norm_tools.empirical_theta(data)
        >>> print(theta)
        0.467133570136

See also:
    TTR_factors
```
name = 'emphist'

static normalize(data, wigList=[], annotationPath='', thetaEst=<function empirical_theta>,
muEst=<function trimmed_empirical_mu>, target=100.0)

Returns the normalization factors for the data, using the TTR method.

Parameters

- **data** *(numpy array)* – (K,N) numpy array defining read-counts at N sites for K datasets.
- **thetaEst** *(function)* – Function used to estimate density. Should take a list of counts as input.
- **muEst** *(function)* – Function used to estimate mean count. Should take a list of counts as input.

Returns

Array with the normalization factors for the TTR method.

Return type

*numpy array*

Example

```python
>>> import pytransit.norm_tools as norm_tools
>>> import pytransit.tnseq_tools as tnseq_tools

>>> (data, position) = tnseq_tools.get_data(['transit/data/glycerol_H37Rv_rep1.wig', 'transit/data/glycerol_H37Rv_rep2.wig'])
>>> print(data)
array([[ 0., 0., 0., ..., 0., 0., 0.],
    [ 0., 0., 0., ..., 0., 0., 0.]])

>>> factors = norm_tools.TTR_factors(data)
>>> print(factors)
array([[ 1.],
    [ 0.62862886]])
```

See also:

normalize_data

trimmed_empirical_mu *(t=0.05)*

Estimates the trimmed mean of the data.

This is used as an estimate of mean count by some normalization methods. May be improved by more sophisticated ways later on.

Parameters

- **data** *(numpy array)* – (N) numpy array defining read-counts at N sites.
- **t** *(float)* – Float specifying fraction of start and end to trim.

Returns

(Trimmed) Mean of the given dataset.

Return type

*float*

Example

```python
>>> import pytransit.tnseq_tools as tnseq_tools
>>> import pytransit.norm_tools as norm_tools

>>> (data, position) = tnseq_tools.get_data(['transit/data/glycerol_H37Rv_rep1.wig', 'transit/data/glycerol_H37Rv_rep2.wig'])
>>> print(data)
```
transit Documentation, Release v3.2.0

(continued from previous page)

```
array([[ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.]])
>>> mu = norm_tools.trimmed_empirical_mu(data)
>>> print(mu)
120.73077107
```

See also:

TTR_factors

class pytransit.norm_tools.TotReadsNorm
Bases: pytransit.norm_tools.NormMethod
name = 'totreads'
static normalize(data, wigList=[], annotationPath="")

Returns the normalization factors for the data, using the total reads method.

Parameters

- **data** *(numpy array)* – (K,N) numpy array defining read-counts at N sites for K datasets.

Returns

Array with the normalization factors for the totreads method.

Return type

numpy array

Example

```
>>> import pytransit.norm_tools as norm_tools
>>> import pytransit.tnseq_tools as tnseq_tools

>>> (data, position) = tnseq_tools.get_data(['transit/data/glycerol\n˓
H37Rv_rep1.wig', 'transit/data/glycerol_H37Rv_rep2.wig'])

>>> print(data)
array([[ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.]])

>>> factors = norm_tools.totreads_factors(data)
>>> print(factors)
array([[ 1.2988762],
       [ 0.8129396]])
```

See also:

normalize_data

class pytransit.norm_tools.ZeroInflatedNBNorm
Bases: pytransit.norm_tools.NormMethod
name = 'zinfb'
static normalize(data, wigList=[], annotationPath="")

Returns the normalization factors for the data using the zero-inflated negative binomial method.

Parameters

- **data** *(numpy array)* – (K,N) numpy array defining read-counts at N sites for K datasets.

Returns

Array with the normalization factors for the zinfb method.

Return type

numpy array

Example

```
```
```python
>>> import pytransit.norm_tools as norm_tools
>>> import pytransit.tnseq_tools as tnseq_tools

>>> (data, position) = tnseq_tools.get_data(['transit/data/glycerol_→H37Rv_rep1.wig', 'transit/data/glycerol_H37Rv_rep2.wig'])

>>> print(data)
array([[[ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.]]])

>>> factors = norm_tools.zinfnb_factors(data)

>>> print(factors)
[[ 0.0121883 ]
 [ 0.00747111]]
```

See also:

- `normalize_data`
- `pytransit.norm_tools.cleaninfgeom(x, rho)`
  Returns a ‘clean’ output from the geometric distribution.
- `pytransit.norm_tools.ecdf(S, x)`
  Calculates an empirical CDF of the given data.
- `pytransit.norm_tools.empirical_theta(X)`
  Calculates the observed density of the data. This is used as an estimate insertion density by some normalization methods. May be improved by more sophisticated ways later on.

Parameters

- **data** (*numpy array*)
  (N) numpy array defining read-counts at N sites.

Returns

- **Density of the given dataset.**

Return type

- float

Example

```python
>>> import pytransit.tnseq_tools as tnseq_tools
>>> import pytransit.norm_tools as norm_tools

>>> (data, position) = tnseq_tools.get_data(['transit/data/glycerol_→H37Rv_rep1.wig', 'transit/data/glycerol_H37Rv_rep2.wig'])

>>> print(data)
array([[[ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.]]])

>>> theta = norm_tools.empirical_theta(data)

>>> print(theta)
0.467133570136
```

See also:

- `TTR_factors`
- `pytransit.norm_tools.norm_to_target(data, target)`
  Returns factors to normalize the data to the given target value.

Parameters

- **data** (*numpy array*) – (K,N) numpy array defining read-counts at N sites for K datasets.
- **target** (*float*) – Floating point specifying the target for the mean of the data/

Returns

- **Array with the factors necessary to normalize mean to target.**
Return type  numpy array

Example

```python
>>> import pytransit.norm_tools as norm_tools
>>> import pytransit.tnseq_tools as tnseq_tools
>>> (data, position) = tnseq_tools.get_data(["transit/data/glycerol_→H37Rv_rep1.wig", "transit/data/glycerol_H37Rv_rep2.wig")
>>> print(data)
array([[ 0., 0., 0., ..., 0., 0., 0.],
      [ 0., 0., 0., ..., 0., 0., 0.]]
>>> factors = norm_tools.norm_to_target(data, 100)
>>> print(factors)
[ 1.8548104 ]
[ 1.16088726]
```

See also:

normalize_data

pytransit.norm_tools.normalize_data(data, method='nonorm', wigList=[], annotationPath=)

Normalizes the numpy array by the given normalization method.

Parameters

- **data** (numpy array) – (K,N) numpy array defining read-counts at N sites for K datasets.
- **method** (str) – Name of the desired normalization method.
- **wigList** (list) – List of paths for the desired wig-formatted datasets.
- **annotationPath** (str) – Path to the prot_table annotation file.

Returns

Array with the normalized data. list: List containing the normalization factors. Empty if not used.

Return type  numpy array

Example

```python
>>> import pytransit.norm_tools as norm_tools
>>> import pytransit.tnseq_tools as tnseq_tools
>>> (data, position) = tnseq_tools.get_data(["transit/data/glycerol_→H37Rv_rep1.wig", "transit/data/glycerol_H37Rv_rep2.wig")
>>> print(data)
array([[ 0., 0., 0., ..., 0., 0., 0.],
      [ 0., 0., 0., ..., 0., 0., 0.]])
>>> (normdata, normfactors) = norm_tools.normalize_data(data, "TTR")
```

Note: Some normalization methods require the wigList and annotationPath arguments.

pytransit.norm_tools.trimmed_empirical_mu(X, t=0.05)

Estimates the trimmed mean of the data.
This is used as an estimate of mean count by some normalization methods. May be improved by more sophisticated ways later on.

**Parameters**

- **data** *(numpy array)* - 
  (N) numpy array defining read-counts at N sites.
- **t** *(float)* - Float specifying fraction of start and end to trim.

**Returns** (Trimmed) Mean of the given dataset.

**Return type** float

**Example**

```python
>>> import pytransit.tnseq_tools as tnseq_tools
>>> import pytransit.norm_tools as norm_tools

>>> (data, position) = tnseq_tools.get_data("transit/data/glycerol_\n→H37Rv_rep1.wig", "transit/data/glycerol_H37Rv_rep2.wig")

>>> print(data)
array([[ 0., 0., 0., ..., 0., 0., 0.],
       [ 0., 0., 0., ..., 0., 0., 0.]])

>>> mu = norm_toolstrimmed_empirical_mu(data)

>>> print(mu)
120.73077107
```

**See also:**

- TTR_factors
- pytransit.norm_tools.zinfb_factors(data)

Returns the normalization factors for the data using the zero-inflated negative binomial method.

**Parameters** **data** *(numpy array)* - (K,N) numpy array defining read-counts at N sites for K datasets.

**Returns** Array with the normalization factors for the zinfb method.

**Return type** numpy array

**Example**

```python
>>> import pytransit.norm_tools as norm_tools

>>> (data, position) = tnseq_tools.get_data("transit/data/glycerol_\n→H37Rv_rep1.wig", "transit/data/glycerol_H37Rv_rep2.wig")

>>> print(data)
array([[ 0., 0., 0., ..., 0., 0., 0.],
       [ 0., 0., 0., ..., 0., 0., 0.]])

>>> factors = norm_tools.zinfb_factors(data)

>>> print(factors)
[[ 0.0121883 ]
 [ 0.00747111]]
```

**See also:**

- normalize_data
2.16.3 pytransit.stat_tools module

pytransit.stat_tools.BH_fdr_correction(X)
    Adjusts p-values using the Benjamini Hochberg procedure

pytransit.stat_tools.FWER_Bayes(X)

pytransit.stat_tools.F_mean_diff_dict(*args, **kwargs)

pytransit.stat_tools.F_mean_diff_flat(*args, **kwargs)

pytransit.stat_tools.F_shuffle_dict_libraries(*args, **kwargs)

pytransit.stat_tools.F_shuffle_flat(*args, **kwargs)

pytransit.stat_tools.F_sum_diff_dict(*args, **kwargs)

pytransit.stat_tools.F_sum_diff_flat(*args, **kwargs)

pytransit.stat_tools.HDI_from_MCMC(posterior_samples, credible_mass=0.95)

pytransit.stat_tools.bFDR(X)

pytransit.stat_tools.bayesian_ess_thresholds(Z_raw, ALPHA=0.05)
    Returns Essentiality Thresholds using a BH-like procedure

pytransit.stat_tools.binom(k, n, p)
    Binomial distribution. Uses Normal approximation for large ‘n’

pytransit.stat_tools.binom_cdf(k, n, p)
    CDF of the binomial distribution

pytransit.stat_tools.binom_test(k, n, p, type='two-sided')
    Does a binomial test given success, trials and probability.

pytransit.stat_tools.boxcoxTable(X, minlambda, maxlambda, dellambda)
    Returns a table of (loglik function, lambda) pairs for the data.

pytransit.stat_tools.boxcoxtransform(x, lambdax)
    Performs a box-cox transformation to data vector X. WARNING: elements of X should be all positive! Fixed: ‘>’ has changed to ‘<’

pytransit.stat_tools.comb(n, k)

pytransit.stat_tools.comb1(n, k)

pytransit.stat_tools.combine_lib_dicts(L1, L2)

pytransit.stat_tools.cumulative_average(new_x, n, prev_avg)

pytransit.stat_tools.dberndiff(d, peq, p01, p10)

pytransit.stat_tools.dbinomdiff(d, n, P)

pytransit.stat_tools.loess_correction(X, Y, h=10000, window=100)

pytransit.stat_tools.loess_correction(X, Y, h=10000, window=100)

pytransit.stat_tools.log_fac(n)

pytransit.stat_tools.loglik(X, lambdax)

    Computes the log-likelihood function for a transformed vector Xtransform.
pytransit.stat_tools.multinomial \((K, P)\)

pytransit.stat_tools.my_perm \((d, n)\)

pytransit.stat_tools.norm \((x, \mu, \sigma)\)

Normal distribution

pytransit.stat_tools.parse_lib_index \((nData, libstr, nTAs)\)

pytransit.stat_tools.phi_coefficient \((X, Y)\)

Calculates the phi-coefficient for two bool arrays

pytransit.stat_tools.qberndiff \((d, p_{eq}, p_{01}, p_{10})\)

pytransit.stat_tools.qbinomdiff \((d, n, p_{eq}, p_{01}, p_{10})\)

pytransit.stat_tools.regress \((X, Y)\)

Performs linear regression given two vectors, \(X, Y\).

pytransit.stat_tools.resampling \((data1, data2, S=10000, testFunc=<function F_mean_diff_flat>, permFunc=<function F_shuffle_flat>, adaptive=False, lib_str1="", lib_str2="", PC=1)\)

Does a permutation test on two sets of data.

Performs the resampling / permutation test given two sets of data using a function defining the test statistic and a function defining how to permute the data.

**Parameters**

- \(ar\) – List or numpy array with the first set of observations.
- \(data2\) – List or numpy array with the second set of observations.
- \(S\) – Number of permutation tests (or samples) to obtain.
- \(testFunc\) – Function defining the desired test statistic. Should accept two lists as arguments. Default is difference in means between the observations.
- \(permFunc\) – Function defining the way to permute the data. Should accept one argument, the combined set of data. Default is random shuffle.
- \(adaptive\) – Cuts-off resampling early depending on significance.

**Returns**

**Tuple with described values**

- \(test\_obs\) – Test statistic of observation.
- \(mean1\) – Arithmetic mean of first set of data.
- \(mean2\) – Arithmetic mean of second set of data.
- \(log2FC\) – Normalized log2FC the means.
- \(pval\_ltail\) – Lower tail p-value.
- \(pval\_utail\) – Upper tail p-value.
- \(pval\_2tail\) – Two-tailed p-value.
- \(test\_sample\) – List of samples of the test statistic.

**Example**
```python
>>> import pytransit.stat_tools as stat
>>> import numpy

>>> X = numpy.random.random(100)
>>> Y = numpy.random.random(100)

>>> (test_obs, mean1, mean2, log2fc, pval_ltail, pval_utail, pval_2tail, test_sample) = stat_tools.resampling(X,Y)

>>> pval_2tail
0.2167

>>> test_sample[:3]
[0.076213992904990535, -0.0052513291091412784, -0.0038425140184765172]
```

pytransit.stat_tools.sample_trunc_norm_post (data, S, mu0, s20, k0, nu0)

pytransit.stat_tools.text_histogram (X, nBins=20, resolution=200, obs=None)

pytransit.stat_tools.transformToRange (X, new_min, new_max, old_min=None, old_max=None)

pytransit.stat_tools.tricube (X)

### 2.16.4 pytransit.tnseq_tools module

pytransit.tnseq_tools.ExpectedRuns (n, pnon)

- **Expected value of the run of non-insertions (Schilling, 1990):**
  
  \[ ER_n = \log(1/p)(nq) + \gamma/\ln(1/p) -1/2 + r1(n) + E1(n) \]

- **Parameters**
  
  - `n` *(int)* – Integer representing the number of sites.
  
  - `pins` *(float)* – Floating point number representing the probability of non-insertion.

- **Returns** Size of the expected maximum run.

- **Return type** float

**class** pytransit.tnseq_tools.Gene (orf, name, desc, reads, position, start=0, end=0, strand="")

- **Class defining a gene with useful attributes for TnSeq analysis.**

  - This class helps define a “gene” with attributes that facilitate TnSeq analysis. Here “gene” can be defined to be any genomic region. The Genes class (with an s) can be used to define list of Gene objects with more useful operations on the “genome” level.

  - **orf**
    
    A string defining the ID of the gene.

  - **name**
    
    A string with the human readable name of the gene.

  - **desc**
    
    A string with the description of the gene.

  - **reads**
    
    List of lists of read-counts in possible site replicate dataset.

  - **position**
    
    List of coordinates of the possible sites.
**start**
An integer defining the start coordinate for the gene.

**end**
An integer defining the end coordinate for the gene.

**strand**
A string defining the strand of the gene.

**Example**

```python
>>> import pytransit.tnseq_tools as tnseq_tools

>>> G = tnseq_tools.Gene("Rv0001", "dnaA", "DNA Replication A", 
[0,0,-0,0,1,3,0,1], [1,21,32,37,45,58,66,130], strand="+")

>>> print(G)
Rv0001 (dnaA) k=3 n=8 r=4 theta=0.37500

>>> print(G.phi())
0.625

>>> print(G.tosses)
array([ 0., 0., 0., 0., 1., 1., 0., 1.])
```

See also:

**__eq__(other)**
Compares against other gene object.

- **Returns**: True if the gene objects have same orf id.
- **Return type**: bool

**__ge__(other)**

- **x.__ge__(y) <==> x>=y**

**__getitem__(i)**

- **Return** read-counts at position i.

- **Parameters**: i (**int**) – integer of the index of the desired site.

- **Returns**: Reads at position i.

- **Return type**: list

**__gt__(other)**

- **x.__gt__(y) <==> x>y**

**__le__(other)**

- **x.__le__(y) <==> x<=y**

**__lt__(other)**

- **x.__lt__(y) <==> x<y**

- **Compare** against other gene object.

- **Returns**: True if the gene object id is less than the other.

- **Return type**: bool

**__ne__(other)**

- **x.__ne__(y) <==> x!=y**

**__str__()**

- **Return** a string representation of the object.

- **Returns**: Human readable string with some of the attributes.
Return type str

get_gap_span()  
Returns the span of the maxrun of the gene (i.e. number of nucleotides).
   
Returns Number of nucleotides spanned by the max run.
Return type int

gene_span()  
Returns the number of nucleotides spanned by the gene.
   
Returns Number of nucleotides spanned by the gene’s sites.
Return type int

phi()  
Return the non-insertion density (“phi”) for the gene.
   
Returns Non-insertion density (i.e. 1 - theta)
Return type float

theta()  
Return the insertion density (“theta”) for the gene.
   
Returns Density of the gene (i.e. k/n )
Return type float

total_reads()  
Return the total reads for the gene.
   
Returns Total sum of read-counts.
Return type float

Class defining a list of Gene objects with useful attributes for TnSeq analysis.

This class helps define a list of Gene objects with attributes that facilitate TnSeq analysis. Includes methods that calculate useful statistics and even rudimentary analysis of essentiality.

wigList  
List of paths to datasets in .wig format.

protTable  
String with path to annotation in .prot_table format.

norm  
String with the normalization used/

reps  
String with information on how replicates were handled.

minread  
Integer with the minimum magnitude of read-count considered.

ignoreCodon  
Boolean defining whether to ignore the start/stop codon.

nterm  
Float number of the fraction of the N-terminus to ignore.
cterm
Float number of the fraction of the C-terminus to ignore.

include_nc
Boolean determining whether to include non-coding areas.

orf2index
Dictionary of orf id to index in the genes list.

genes
List of the Gene objects.

Example

```python
>>> import pytransit.tnseq_tools as tnseq_tools
>>> G = tnseq_tools.Genes(["transit/data/glycerol_H37Rv_rep1.wig", "transit/data/glycerol_H37Rv_rep2.wig"], "transit/genomes/H37Rv.prot_table", norm="TTR")
>>> print(G)
Genes Object (N=3990)
>>> print(G.global_theta())
0.40853707222816626
>>> print(G["Rv0001"] # Lookup like dictionary)
Rv0001 (dnaA) k=0 n=31 r=31 theta=0.00000
>>> print(G[2] # Lookup like list)
Rv0003 (recF) k=5 n=35 r=14 theta=0.14286
>>> print(G[2].reads)
[[ 62. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [ 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [ 0. 0. 63. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [ 13. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [ 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [ 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [ 3.14314432 67.26328843 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [ 0. 0. 0. 0. 35.20321637 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [ 0. 0. 0. 30.80281433 0. 0. 101.20924707 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [ 0. 23.25926796 0. 0. 0. 0. 16.97297932 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [ 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [ 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [ 0. 0. 0. ]]
```

See also:

Gene

__contains__(item)
Defines __contains__ to check if gene exists in the list.

Parameters item (str) – String with the id of the gene.

Returns Boolean with True if item is in the list.
Return type bool

__getitem__(i)
Defines __getitem__ method so that it works as dictionary and list.

Parameters i (int) – Integer or string defining index or orf ID desired.

Returns A gene with the index or ID equal to i.

Return type Gene

__len__()
Defines __len__ returning number of genes.

Returns Number of genes in the list.

Return type int

__str__()
Defines __str__ to print(a generic str with the size of the list.)

Returns Human readable string with number of genes in object.

Return type str

global_insertion()
Returns total number of insertions, i.e. sum of 'k' over all genes.

Returns Total sum of reads across all genes.

Return type float

global_phi()
Returns global non-insertion frequency, of the library.

Returns Complement of global theta i.e. 1.0-theta

Return type float

global_reads()
Returns the reads among the library.

Returns List of all the data.

Return type list

global_run()
Returns the run assuming all genes were concatenated together.

Returns Max run across all genes.

Return type int

global_sites()
Returns total number of sites, i.e. sum of ‘n’ over all genes.

Returns Total number of sites across all genes.

Return type int

global_theta()
Returns global insertion frequency, of the library.

Returns Total sites with insertions divided by total sites.

Return type float
local_gap_span()
Returns numpy array with the span of nucleotides of the largest gap, 's', for each gene.

Returns Numpy array with the span of gap for all genes.
Return type narray

local_gene_span()
Returns numpy array with the span of nucleotides of the gene, 't', for each gene.

Returns Numpy array with the span of gene for all genes.
Return type narray

local_insertions()
Returns numpy array with the number of insertions, 'k', for each gene.

Returns Numpy array with the number of insertions for all genes.
Return type narray

local_phis()
Returns numpy array of non-insertion frequency, 'phi', for each gene.

Returns Numpy array with the complement of density for all genes.
Return type narray

local_reads()
Returns numpy array of lists containing the read counts for each gene.

Returns Numpy array with the list of reads for all genes.
Return type narray

local_runs()
Returns numpy array with maximum run of non-insertions, 'r', for each gene.

Returns Numpy array with the max run of non-insertions for all genes.
Return type narray

local_sites()
Returns numpy array with total number of TA sites, 'n', for each gene.

Returns Numpy array with the number of sites for all genes.
Return type narray

local_thetas()
Returns numpy array of insertion frequencies, 'theta', for each gene.

Returns Numpy array with the density for all genes.
Return type narray

tosses()
Returns list of bernoulli trials, 'tosses', representing insertions in the gene.

Returns Sites represented as bernoulli trials with insertions as true.
Return type list

total_reads()
Returns total reads among the library.

Returns Total sum of read-counts accross all genes.
Return type  float

pytransit.tnseq_tools.GumbelCDF(x, u, B)
CDF of the Gumbel distribution:
\[ e^{-e^{(u-x)/B}} \]

Parameters
- x (int) – Length of the max run.
- u (float) – Location parameter of the Gumbel dist.
- B (float) – Scale parameter of the Gumbel dist.

Returns  Cumulative probability of the Gumbel distribution.

Return type  float

pytransit.tnseq_tools.VarR(n, pnon)
Variance of the expected run of non-insertions (Schilling, 1990):
\[ VarR_n = (p^2)/(6 * ln(1/p)^2) + 1/12 + r2(n) + E2(n) \]

Parameters
- n (int) – Integer representing the number of sites.
- pnon (float) – Floating point number representing the probability of non-insertion.

Returns  Variance of the length of the maximum run.

Return type  float

pytransit.tnseq_tools.check_wig_includes_zeros(wig_list)
Returns boolean list showing whether the given files include empty sites (zero) or not.

Parameters  wig_list (list) – List of paths to wig files.

Returns  List of boolean values.

Return type  list

pytransit.tnseq_tools.combine_replicates(data, method='Sum')
Returns list of data merged together.

Parameters
- data (list) – List of numeric (replicate) data to be merged.
- method (str) – How to combine the replicate dataset.

Returns  List of numeric dataset now merged together.

Return type  list

pytransit.tnseq_tools.getE1(n)
Small Correction term. Defaults to 0.01 for now

pytransit.tnseq_tools.getE2(n)
Small Correction term. Defaults to 0.01 for now

pytransit.tnseq_tools.getGamma()
Euler-Mascheroni constant \( \approx 0.577215664901 \)

pytransit.tnseq_tools.getR1(n)
Small Correction term. Defaults to 0.000016 for now
pytransit.tnseq_tools.getR2(n)
Small Correction term. Defaults to 0.00006 for now

pytransit.tnseq_tools.get_coordinate_map(galign_path, reverse=False)
Attempts to get mapping of coordinates from galign file.

Parameters
  • path (str) – Path to .galign file.
  • reverse (bool) – Boolean specifying whether to do A to B or B to A.

Returns Dictionary of coordinate in one file to another file.

Return type dict

pytransit.tnseq_tools.get_data(wig_list)

Returns a tuple of (data, position) containing a matrix of raw read-counts, and list of coordinates.

Parameters wig_list (list) – List of paths to wig files.

Returns Two lists containing data and positions of the wig files given.

Return type tuple

Example

```python
>>> import pytransit.tnseq_tools as tnseq_tools
>>> (data, position) = tnseq_tools.get_data(["data/glycerol_H37Rv_rep1.wig", "data/glycerol_H37Rv_rep2.wig"])
>>> print(data)
array([[ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.]])
```

See also:
get_file_types combine_replicates get_data_zero_fill pytransit.norm_tools.normalize_data

pytransit.tnseq_tools.get_data_stats(reads)

pytransit.tnseq_tools.get_data_w_genome(wig_list, genome)

pytransit.tnseq_tools.get_data_zero_fill(wig_list)

Returns a tuple of (data, position) containing a matrix of raw read counts, and list of coordinates. Positions that are missing are filled in as zero.

Parameters wig_list (list) – List of paths to wig files.

Returns Two lists containing data and positions of the wig files given.

Return type tuple

pytransit.tnseq_tools.get_extended_pos_hash_gff(path, N=None)

pytransit.tnseq_tools.get_extended_pos_hash_pt(path, N=None)

pytransit.tnseq_tools.get_file_types(wig_list)

Returns the transposon type (himar1/tn5) of the list of wig files.

Parameters wig_list (list) – List of paths to wig files.

Returns List of transposon type (“himar1” or “tn5”).
Return type list

`pytransit.tnseq_tools.get_gene_info(path)`

Returns a dictionary that maps gene id to gene information.

Parameters path (str) – Path to annotation in .prot_table or GFF3 format.

Returns

Dictionary of gene id to tuple of information:

- name
- description
- start coordinate
- end coordinate
- strand

Return type dict

`pytransit.tnseq_tools.get_gene_info_gff(path)`

Returns a dictionary that maps gene id to gene information.

Parameters path (str) – Path to annotation in GFF3 format.

Returns

Dictionary of gene id to tuple of information:

- name
- description
- start coordinate
- end coordinate
- strand

Return type dict

`pytransit.tnseq_tools.get_gene_info_pt(path)`

Returns a dictionary that maps gene id to gene information.

Parameters path (str) – Path to annotation in .prot_table format.

Returns

Dictionary of gene id to tuple of information:

- name
- description
- start coordinate
- end coordinate
- strand

Return type dict

`pytransit.tnseq_tools.get_genes_in_range(pos_hash, start, end)`

Returns list of genes that occur in a given range of coordinates.

Parameters
• **pos_hash** (*dict*) – Dictionary of position to list of genes.
• **start** (*int*) – Start coordinate of the desired range.
• **end** (*int*) – End coordinate of the desired range.

**Returns**  List of genes that fall within range.

**Return type**  list

`pytransit.tnseq_tools.get_pos_hash(path)`

Returns a dictionary that maps coordinates to a list of genes that occur at that coordinate.

**Parameters**  path (*str*) – Path annotation in .prot_table or GFF3 format.

**Returns**  Dictionary of position to list of genes that share that position.

**Return type**  dict

`pytransit.tnseq_tools.get_pos_hash_gff(path)`

Returns a dictionary that maps coordinates to a list of genes that occur at that coordinate.

**Parameters**  path (*str*) – Path annotation in GFF3 format.

**Returns**  Dictionary of position to list of genes that share that position.

**Return type**  dict

`pytransit.tnseq_tools.get_pos_hash_pt(path)`

Returns a dictionary that maps coordinates to a list of genes that occur at that coordinate.

**Parameters**  path (*str*) – Path annotation in .prot_table format.

**Returns**  Dictionary of position to list of genes that share that position.

**Return type**  dict

`pytransit.tnseq_tools.get_unknown_file_types(wig_list, transposons)`

`pytransit.tnseq_tools.get_wig_stats(path)`

Returns statistics for the given wig file with read-counts.

**Parameters**  path (*str*) – String with the path to the wig file of interest.

**Returns**

**Tuple with the following statistical measures:**

• density
• mean read
• non-zero mean
• non-zero median
• max read
• total reads
• skew
• kurtosis

**Return type**  tuple

`pytransit.tnseq_tools.griffin_analysis(genes_obj, pins)`

Implements the basic Gumbel analysis of runs of non-insertion, described in Griffin et al. 2011.
This analysis method calculates a p-value of observing the maximum run of TA sites without insertions in a row (i.e., a “run”, r). Unusually long runs are indicative of an essential gene or protein domain. Assumes that there is a constant, global probability of observing an insertion (tantamount to a Bernoulli probability of success).

**Parameters**

- `genes_obj (Genes)` – An object of the Genes class defining the genes.
- `pins (float)` – The probability of insertion.

**Returns**

A list of lists with results and information for the genes. The elements of the list are as follows:

- ORF ID.
- Gene Name.
- Gene Description.
- Number of TA sites with insertions.
- Number of TA sites.
- Length of largest run of non-insertion.
- Expected run for a gene this size.
- p-value of the observed run.

**Return type** list

```python
pytransit.tnseq_tools.maxrun(lst, item=0)
```

Returns the length of the maximum run an item in a given list.

**Parameters**

- `lst (list)` – List of numeric items.
- `item (float)` – Number to look for consecutive runs of.

**Returns** Length of the maximum run of consecutive instances of item.

**Return type** int

```python
pytransit.tnseq_tools.read_combined_wig(fname)
```

Read the combined wig-file generated by Transit :: Filename -> Tuple([Site], [WigData], [Filename]) Site :: Integer WigData :: [Number] Filename :: String

```python
pytransit.tnseq_tools.read_genes(fname, descriptions=False)
```

(Filename, Options) -> [Gene] Gene :: {start, end, rv, gene, strand}

```python
pytransit.tnseq_tools.read_genome(path)
```

Reads in FASTA formatted genome file.

**Parameters** path (str) – Path to .galign file.

**Returns** String with the genomic sequence.

**Return type** string

```python
pytransit.tnseq_tools.read_samples_metadata(metadata_file, condition_name='Condition', covarsToRead=[], interactionsToRead=[]) Filename -> ConditionMap ConditionMap :: {Filename: Condition}, [{Filename: Covar}], [{Filename: Interaction}] Condition :: String Covar :: String Interaction :: String
```

100 Chapter 2. Features
pytransit.tnseq_tools.runindex(runs)
    Returns a list of the indexes of the start of the runs; complements runs().
    Parameters runs (list) – List of numeric data.
    Returns List of the index of the runs of non-insertions. Non-zero sites are treated as runs of zero.
    Return type list

pytransit.tnseq_tools.runs(data)
    Return list of all the runs of consecutive non-insertions.
    Parameters data (list) – List of numeric data.
    Returns List of the length of the runs of non-insertions. Non-zero sites are treated as runs of zero.
    Return type list

pytransit.tnseq_tools.runs_w_info(data)
    Return list of all the runs of consecutive non-insertions with the start and end locations.
    Parameters data (list) – List of numeric data to check for runs.
    Returns List of dictionary from run to length and position information of the run.
    Return type list

pytransit.tnseq_tools.rv_siteindexes_map(genes, TASiteindexMap, nterm=0.0, cterm=0.0)
    ([Gene], {TAsite: Siteindex}) -> {Rv: Siteindex}

pytransit.tnseq_tools.tossify(data)
    Reduces the data into Bernoulli trials (or 'tosses') based on whether counts were observed or not.
    Parameters data (list) – List of numeric data.
    Returns Data represented as bernoulli trials with >0 as true.
    Return type list

2.16.5 pytransit.transit_tools module

pytransit.transit_tools.ShowAskWarning(MSG=")
pytransit.transit_tools.ShowError(MSG=")
pytransit.transit_tools.ShowMessage(MSG=")
pytransit.transit_tools.aton(aa)
pytransit.transit_tools.basename(filepath)
pytransit.transit_tools.cleanargs(rawargs)
    Returns a list and a dictionary with positional and keyword arguments.
    -This function assumes flags must start with a ‘-‘ and and cannot be a number (but can include them).
    -Flags should either be followed by the value they want to be associated with (i.e. -p 5) or will be assigned a value of True in the dictionary.
    -The dictionary will map flags to the name given minus ONE ‘-‘ sign in front. If you use TWO minus signs in the flag name (i.e. --verbose), the dictionary key will be the name with ONE minus sign in front (i.e. {'--verbose':True}).
    Parameters rawargs (list) – List of positional/keyword arguments. As obtained from sys.argv.
    Returns
List of positional arguments (i.e. arguments without flags), in order provided.

dict: Dictionary mapping flag (key is flag minus the first “-“) and their values.

Return type  list

pytransit.transit_tools.convertToCombinedWig(dataset_list, annotationPath, outputPath, normchoice='nonorm')
Normalizes the input datasets and outputs the result in CombinedWig format.

Parameters

• dataset_list (list) – List of paths to datasets in .wig format
• annotationPath (str) – Path to annotation in .prot_table or GFF3 format.
• outputPath (str) – Desired output path.
• normchoice (str) – Choice for normalization method.

pytransit.transit_tools.convertToGeneCountSummary(dataset_list, annotationPath, outputPath, normchoice='nonorm')
Normalizes the input datasets and outputs the result in CombinedWig format.

Parameters

• dataset_list (list) – List of paths to datasets in .wig format
• annotationPath (str) – Path to annotation in .prot_table or GFF3 format.
• outputPath (str) – Desired output path.
• normchoice (str) – Choice for normalization method.

pytransit.transit_tools.dirname(filepath)

pytransit.transit_tools.fetch_name(filepath)

pytransit.transit_tools.getTabTableData(path, colnames)

pytransit.transit_tools.get_extended_pos_hash(path)
Returns a dictionary that maps coordinates to a list of genes that occur at that coordinate.

Parameters  path (str) – Path to annotation in .prot_table or GFF3 format.

Returns  Dictionary of position to list of genes that share that position.

Return type  dict

pytransit.transit_tools.get_gene_info(path)
Returns a dictionary that maps gene id to gene information.

Parameters  path (str) – Path to annotation in .prot_table or GFF3 format.

Returns  Dictionary of gene id to tuple of information:

• name
• description
• start coordinate
• end coordinate
• strand
Return type  dict

**get_pos_hash**(*path*)

Returns a dictionary that maps coordinates to a list of genes that occur at that coordinate.

**Parameters**

- **path** (*str*) – Path to annotation in .prot_table or GFF3 format.

**Returns**

Dictionary of position to list of genes that share that position.

**Return type**  dict

**get_validated_data**(*wig_list*, *wxobj=None*)

Returns a tuple of (data, position) containing a matrix of raw read-counts, and list of coordinates.

**Parameters**

- **wig_list** (*list*) – List of paths to wig files.
- **wxobj** (*object*) – wxPython GUI object for warnings

**Returns**

Two lists containing data and positions of the wig files given.

**Return type**  tuple

**Example**

```python
>>> import pytransit.tnseq_tools as tnseq_tools

>>> (data, position) = tnseq_tools.get_validated_data(["data/glycerol_H37Rv_rep1.wig", "data/glycerol_H37Rv_rep2.wig"])

>>> print(data)
array([[ 0., 0., 0., ..., 0., 0., 0.],
      [ 0., 0., 0., ..., 0., 0., 0.]])
```

**See also:**

- get_file_types
- combine_replicates
- get_data_zero_fill
- pytransit.norm_tools.normalize_data

**parseCoords**(*strand, aa_start, aa_end, start, end*)

**transit_error**(*text*)

**transit_message**(*msg=", prefix="*)

**validate_annotation**(*annotation*)

**validate_both_datasets**(*ctrldata, expdata*)

**validate_control_datasets**(*ctrldata*)

**validate_filetypes**(*datasets, transposons, justWarn=True*)

**validate_transposons_used**(*datasets, transposons, justWarn=True*)

**validate_wig_format**(*wig_list, wxobj=None*)

### 2.16.6 Module contents

- genindex
- modindex
- search
For any questions or comments, please contact Dr. Thomas Ioerger, ioerger@cs.tamu.edu.


Symbols

__contains__() (pytransit.tnseq_tools.Genes method), 93
__eq__() (pytransit.tnseq_tools.Gene method), 91
__ge__() (pytransit.tnseq_tools.Gene method), 91
__getitem__() (pytransit.tnseq_tools.Gene method), 91
__getitem__() (pytransit.tnseq_tools.Genes method), 94
__gt__() (pytransit.tnseq_tools.Gene method), 91
__le__() (pytransit.tnseq_tools.Gene method), 91
__len__() (pytransit.tnseq_tools.Genes method), 94
__lt__() (pytransit.tnseq_tools.Gene method), 91
__ne__() (pytransit.tnseq_tools.Gene method), 91
__str__() (pytransit.tnseq_tools.Gene method), 91
__str__() (pytransit.tnseq_tools.Genes method), 94

A
AdaptiveBGCNorm (class in pytransit.norm_tools), 79
aton() (in module pytransit.transit_tools), 101

B
basename() (in module pytransit.transit_tools), 102
bayesian_ess_thresholds() (in module pytransit.stat_tools), 88
BetaGeomNorm (class in pytransit.norm_tools), 79
bFDR() (in module pytransit.stat_tools), 88
BH_fdr_correction() (in module pytransit.stat_tools), 88
binom() (in module pytransit.stat_tools), 88
binom_cdf() (in module pytransit.stat_tools), 88
binom_test() (in module pytransit.stat_tools), 88
boxcoxTable() (in module pytransit.stat_tools), 88
boxcoxtransform() (in module pytransit.stat_tools), 88

c
check_wig_includes_zeros() (in module pytransit.tnseq_tools), 96
cleanargs() (in module pytransit.transit_tools), 101
cleaninfgeom() (in module pytransit.norm_tools), 85
cleaninfgeom() (pytransit.norm_tools.AdaptiveBGCNorm method), 79
cleaninfgeom() (pytransit.norm_tools.BetaGeomNorm method), 79
comb() (in module pytransit.stat_tools), 88
combl() (in module pytransit.stat_tools), 88
combine_lib_dicts() (in module pytransit.stat_tools), 88
combine_replicates() (in module pytransit.tnseq_tools), 96
convertToCombinedWig() (in module pytransit.transit_tools), 102
convertToGeneCountSummary() (in module pytransit.transit_tools), 102
cterm (pytransit.tnseq_tools.Genes attribute), 92
cumulative_average() (in module pytransit.stat_tools), 88

d
dberndiff() (in module pytransit.stat_tools), 88
dbinomdiff() (in module pytransit.stat_tools), 88
desc (pytransit.tnseq_tools.Gene attribute), 90
dirname() (in module pytransit.transit_tools), 102

e
ecdf() (in module pytransit.norm_tools), 85
ecdf() (pytransit.norm_tools.AdaptiveBGCNorm method), 79
ecdf() (pytransit.norm_tools.BetaGeomNorm method), 79
EmpHistNorm (class in pytransit.norm_tools), 80
tempirical_theta() (in module pytransit.norm_tools), 85
empirical_theta() (pytransit.norm_tools.TTRNorm method), 82
end (pytransit.tnseq_tools.Gene attribute), 91
ExpectedRuns() (in module pytransit.tnseq_tools), 90

F
F_mean_diff_dict() (in module pytransit.stat_tools), 88
F_mean_diff_flat() (in module pytransit.stat_tools), 88
F_shuffle_dict_libraries() (in module pytransit.stat_tools), 88
F_shuffle_flat() (in module pytransit.stat_tools), 88
F_sum_diff_dict() (in module pytransit.stat_tools), 88
F_sum_diff_flat() (in module pytransit.stat_tools), 88
fact() (in module pytransit.stat_tools), 88
fetch_name() (in module pytransit.transit_tools), 102
FWER_Bayes() (in module pytransit.stat_tools), 88
Fzinfnb() (in module pytransit.norm_tools), 81
Fzinfnb() (pytransit.norm_tools.EmpHistNorm static method), 80

G
Gene (class in pytransit.tnseq_tools), 90
Genes (class in pytransit.tnseq_tools), 92
genes (pytransit.tnseq_tools.Genes attribute), 93
get_coordinate_map() (in module pytransit.tnseq_tools), 97
get_data() (in module pytransit.tnseq_tools), 97
get_data_stats() (in module pytransit.tnseq_tools), 97
get_data_w_genome() (in module pytransit.tnseq_tools), 97
get_data_zero_fill() (in module pytransit.tnseq_tools), 97
get_extended_pos_hash() (in module pytransit.tnseq_tools), 102
get_extended_pos_hash_gff() (in module pytransit.tnseq_tools), 97
get_extended_pos_hash_pt() (in module pytransit.tnseq_tools), 97
get_file_types() (in module pytransit.tnseq_tools), 97
get_gap_span() (pytransit.tnseq_tools.Gene method), 92
get_gene_info() (in module pytransit.tnseq_tools), 98
get_gene_info() (in module pytransit.tnseq_tools), 102
get_gene_info_gff() (in module pytransit.tnseq_tools), 98
get_gene_info_pt() (in module pytransit.tnseq_tools), 98
get_gene_span() (pytransit.tnseq_tools.Gene method), 92
get_genes_in_range() (in module pytransit.tnseq_tools), 98
get_lib_data_dict() (in module pytransit.stat_tools), 88
get_pos_hash() (in module pytransit.tnseq_tools), 99
get_pos_hash() (in module pytransit.transit_tools), 103
get_pos_hash_gff() (in module pytransit.tnseq_tools), 99
get_pos_hash_pt() (in module pytransit.tnseq_tools), 99
get_unknown_file_types() (in module pytransit.tnseq_tools), 99
get_validated_data() (in module pytransit.transit_tools), 103
get_wig_stats() (in module pytransit.tnseq_tools), 99
gE1() (in module pytransit.tnseq_tools), 96
gE2() (in module pytransit.tnseq_tools), 96
gGamma() (in module pytransit.tnseq_tools), 96
gR1() (in module pytransit.tnseq_tools), 96
gR2() (in module pytransit.tnseq_tools), 96
gTabTableData() (in module pytransit.transit_tools), 102
global_insertion() (pytransit.tnseq_tools.Genes method), 94
global_phi() (pytransit.tnseq_tools.Genes method), 94
global_reads() (pytransit.tnseq_tools.Genes method), 94
global_run() (pytransit.tnseq_tools.Genes method), 94
global_sites() (pytransit.tnseq_tools.Genes method), 94
global_theta() (pytransit.tnseq_tools.Genes method), 94
griffin_analysis() (in module pytransit.tnseq_tools), 99
GumbelCDF() (in module pytransit.tnseq_tools), 96

H
HDI_from_MCMC() (in module pytransit.stat_tools), 88

I
ignoreCodon (pytransit.tnseq_tools.Genes attribute), 92
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Module/Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>include_nc</td>
<td>pytransit.tnseq_tools.Genes attribute, 93</td>
</tr>
<tr>
<td>isEven()</td>
<td>(in module pytransit.stat_tools), 88</td>
</tr>
<tr>
<td>local_gap_span()</td>
<td>(pytransit.tnseq_tools.Genes method), 94</td>
</tr>
<tr>
<td>local_gene_span()</td>
<td>(pytransit.tnseq_tools.Genes method), 95</td>
</tr>
<tr>
<td>local_insertions()</td>
<td>(pytransit.tnseq_tools.Genes method), 95</td>
</tr>
<tr>
<td>local_phis()</td>
<td>(pytransit.tnseq_tools.Genes method), 95</td>
</tr>
<tr>
<td>local_reads()</td>
<td>(pytransit.tnseq_tools.Genes method), 95</td>
</tr>
<tr>
<td>local_runs()</td>
<td>(pytransit.tnseq_tools.Genes method), 95</td>
</tr>
<tr>
<td>local_sites()</td>
<td>(pytransit.tnseq_tools.Genes method), 95</td>
</tr>
<tr>
<td>local_thetas()</td>
<td>(pytransit.tnseq_tools.Genes method), 95</td>
</tr>
<tr>
<td>loess()</td>
<td>(in module pytransit.stat_tools), 88</td>
</tr>
<tr>
<td>loess_correction()</td>
<td>(in module pytransit.stat_tools), 88</td>
</tr>
<tr>
<td>log_fac()</td>
<td>(in module pytransit.stat_tools), 88</td>
</tr>
<tr>
<td>loglik()</td>
<td>(in module pytransit.stat_tools), 88</td>
</tr>
<tr>
<td>maxrun()</td>
<td>(in module pytransit.tnseq_tools), 100</td>
</tr>
<tr>
<td>minread()</td>
<td>(pytransit.tnseq_tools.Genes attribute), 92</td>
</tr>
<tr>
<td>multinomial()</td>
<td>(in module pytransit.stat_tools), 88</td>
</tr>
<tr>
<td>my_perm()</td>
<td>(in module pytransit.stat_tools), 89</td>
</tr>
<tr>
<td>name</td>
<td>(pytransit.norm_tools.AdaptiveBGCNorm attribute), 79</td>
</tr>
<tr>
<td>name</td>
<td>(pytransit.norm_tools.BetaGeomNorm attribute), 80</td>
</tr>
<tr>
<td>name</td>
<td>(pytransit.norm_tools.EmpHistNorm attribute), 80</td>
</tr>
<tr>
<td>name</td>
<td>(pytransit.norm_tools.NoNorm attribute), 81</td>
</tr>
<tr>
<td>name</td>
<td>(pytransit.norm_tools.NormMethod attribute), 81</td>
</tr>
<tr>
<td>name</td>
<td>(pytransit.norm_tools.NZMeanNorm attribute), 81</td>
</tr>
<tr>
<td>name</td>
<td>(pytransit.norm_tools.QuantileNorm attribute), 82</td>
</tr>
<tr>
<td>name</td>
<td>(pytransit.norm_tools.TotReadsNorm attribute), 84</td>
</tr>
<tr>
<td>name</td>
<td>(pytransit.norm_tools.TTRNorm attribute), 83</td>
</tr>
<tr>
<td>name</td>
<td>(pytransit.norm_tools.ZeroInflatedNBNorm attribute), 84</td>
</tr>
<tr>
<td>name</td>
<td>(pytransit.tnseq_tools.Genes attribute), 92</td>
</tr>
<tr>
<td>name</td>
<td>(pytransit.tnseq_tools.Gene attribute), 90</td>
</tr>
<tr>
<td>normalize()</td>
<td>(pytransit.norm_tools.AdaptiveBGCNorm static method), 79</td>
</tr>
<tr>
<td>normalize()</td>
<td>(pytransit.norm_tools.BetaGeomNorm static method), 80</td>
</tr>
<tr>
<td>normalize()</td>
<td>(pytransit.norm_tools.EmpHistNorm static method), 80</td>
</tr>
<tr>
<td>normalize()</td>
<td>(pytransit.norm_tools.NoNorm static method), 81</td>
</tr>
<tr>
<td>normalize()</td>
<td>(pytransit.norm_tools.NormMethod static method), 81</td>
</tr>
<tr>
<td>normalize()</td>
<td>(pytransit.norm_tools.NZMeanNorm static method), 81</td>
</tr>
<tr>
<td>normalize()</td>
<td>(pytransit.norm_tools.QuantileNorm static method), 82</td>
</tr>
<tr>
<td>normalize()</td>
<td>(pytransit.norm_tools.TotReadsNorm static method), 84</td>
</tr>
<tr>
<td>normalize()</td>
<td>(pytransit.norm_tools.TTRNorm static method), 83</td>
</tr>
<tr>
<td>normalize()</td>
<td>(pytransit.norm_tools.ZeroInflatedNBNorm static method), 84</td>
</tr>
<tr>
<td>nterm</td>
<td>(class in pytransit.norm_tools), 81</td>
</tr>
<tr>
<td>orf (pytransit.tnseq_tools.Gene attribute), 90</td>
<td>orf2index (pytransit.tnseq_tools.Genes attribute), 93</td>
</tr>
<tr>
<td>orf (pytransit.tnseq_tools.Genes attribute), 92</td>
<td>parse_lib_index() (in module pytransit.stat_tools), 89</td>
</tr>
<tr>
<td>parse_coords()</td>
<td>(in module pytransit.transit_tools), 103</td>
</tr>
<tr>
<td>phi (pytransit.tnseq_tools.Gene method), 92</td>
<td>phi_coefficient() (in module pytransit.stat_tools), 89</td>
</tr>
<tr>
<td>position (pytransit.tnseq_tools.Gene attribute), 90</td>
<td>protTable (pytransit.tnseq_tools.Genes attribute), 92</td>
</tr>
<tr>
<td>protTable</td>
<td>(pytransit.tnseq_tools.Genes attribute), 92</td>
</tr>
<tr>
<td>pytransit (module), 103</td>
<td>pytransit.tnseq_tools (module), 88</td>
</tr>
<tr>
<td>pytransit.tnseq_tools (module), 90</td>
<td>pytransit.transit_tools (module), 101</td>
</tr>
<tr>
<td>qberndiff()</td>
<td>(in module pytransit.stat_tools), 89</td>
</tr>
<tr>
<td>qbinomdiff()</td>
<td>(in module pytransit.stat_tools), 89</td>
</tr>
<tr>
<td>QuantileNorm (class in pytransit.norm_tools), 81</td>
<td>qbinomdiff() (in module pytransit.stat_tools), 89</td>
</tr>
</tbody>
</table>
R

read_combined_wig() (in module pytransit.tnseq_tools), 100
read_genes() (in module pytransit.tnseq_tools), 100
read_genome() (in module pytransit.tnseq_tools), 100
read_samples_metadata() (in module pytransit.tnseq_tools), 100
reads (pytransit.tnseq_tools.Gene attribute), 90
regress() (in module pytransit.stat_tools), 89
reps (pytransit.tnseq_tools.Genes attribute), 92
resampling() (in module pytransit.stat_tools), 89
runs() (in module pytransit.tnseq_tools), 101
runs_w_info() (in module pytransit.tnseq_tools), 101
rv_siteindexes_map() (in module pytransit.tnseq_tools), 101

S

sample_trunc_norm_post() (in module pytransit.stat_tools), 90
ShowAskWarning() (in module pytransit.transit_tools), 101
ShowError() (in module pytransit.transit_tools), 101
ShowMessage() (in module pytransit.transit_tools), 101
start (pytransit.tnseq_tools.Gene attribute), 90
strand (pytransit.tnseq_tools.Gene attribute), 91

text_histogram() (in module pytransit.stat_tools), 90
theta() (pytransit.tnseq_tools.Gene method), 92
tosses() (pytransit.tnseq_tools.Genes method), 95
tossify() (in module pytransit.tnseq_tools), 101
total_reads() (pytransit.tnseq_tools.Gene method), 92

total_reads() (pytransit.tnseq_tools.Genes method), 95
TotReadsNorm (class in pytransit.norm_tools), 84
transformToRange() (in module pytransit.stat_tools), 90
transit_error() (in module pytransit.transit_tools), 103
transit_message() (in module pytransit.transit_tools), 103
tricoeff() (in module pytransit.stat_tools), 90
tricube() (in module pytransit.stat_tools), 90
trimmed_empirical_mu() (in module pytransit.norm_tools), 86
trimmed_empirical_mu() (pytransit.norm_tools.TTRNorm method), 83

T

TTRNorm (class in pytransit.norm_tools), 82

V

validate_annotation() (in module pytransit.transit_tools), 103
validate_both_datasets() (in module pytransit.transit_tools), 103
validate_control_datasets() (in module pytransit.transit_tools), 103
validate_filetypes() (in module pytransit.transit_tools), 103
validate_transposons_used() (in module pytransit.transit_tools), 103
validate_wig_format() (in module pytransit.transit_tools), 103
VarR() (in module pytransit.tnseq_tools), 96

W

wigList (pytransit.tnseq_tools.Genes attribute), 92

Z

ZeroInflatedNBNNorm (class in pytransit.norm_tools), 84
zinfnb_factors() (in module pytransit.norm_tools), 87