
transit Documentation

Release v2.1.2

Michael A. DeJesus

May 06, 2018

1	Quick Links	3
2	Mailing List	5
2.1	Overview	5
2.2	Installation	12
2.3	Running TRANSIT	20
2.4	Features	23
2.5	Analysis Methods	28
2.6	Console Mode Cheat-Sheet	37
2.7	Tutorial: Essentiality Analysis in a Single Condition	38
2.8	Tutorial: Essentiality Analysis of the Entire Genome	43
2.9	Tutorial: Comparative Analysis - Glycerol vs Cholesterol	48
2.10	Tutorial: Normalize datasets	56
2.11	Tutorial: Export datasets	60
2.12	Overview	63
2.13	Installation	64
2.14	Running TPP	64
2.15	Overview of Data Processing Procedure	67
2.16	Statistics	68
2.17	transit package	70
	Bibliography	95
	Python Module Index	97

The main documentation for the site is organized into the following sections:

- *TRANSIT Manual*
- *TRANSIT Tutorials*
- *TPP Manual*
- *Code Documentation*

CHAPTER 1

Quick Links

- *Installation*
- *Console Mode Cheat-Sheet*

CHAPTER 2

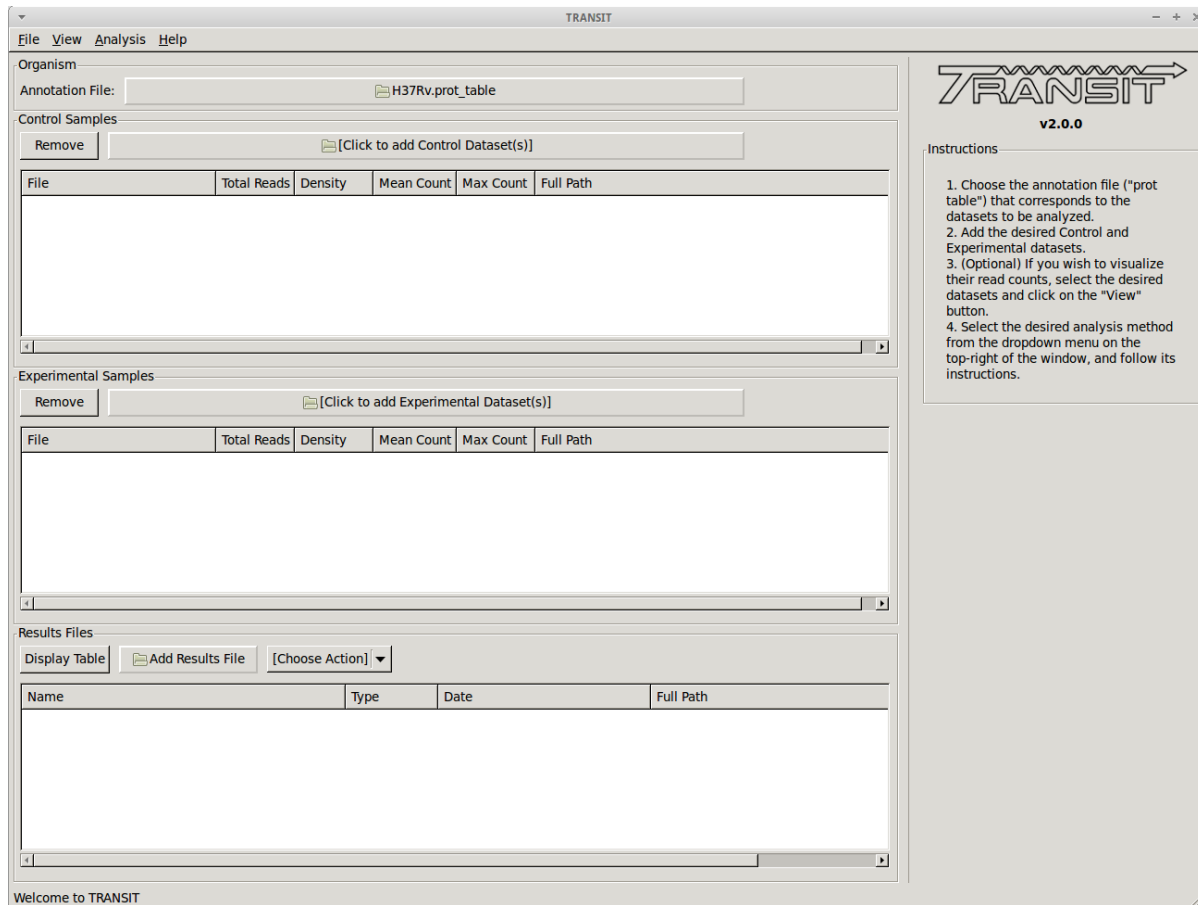
Mailing List

You can join our mailing list to get announcements of new versions, discuss any bugs, or request features! Just head over to the following site and enter your email address:

- <https://groups.google.com/forum/#!forum/tnseq-transit/join>

2.1 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 Ierger lab (Texas A&M) [*DeJesus2015TRANSIT*].



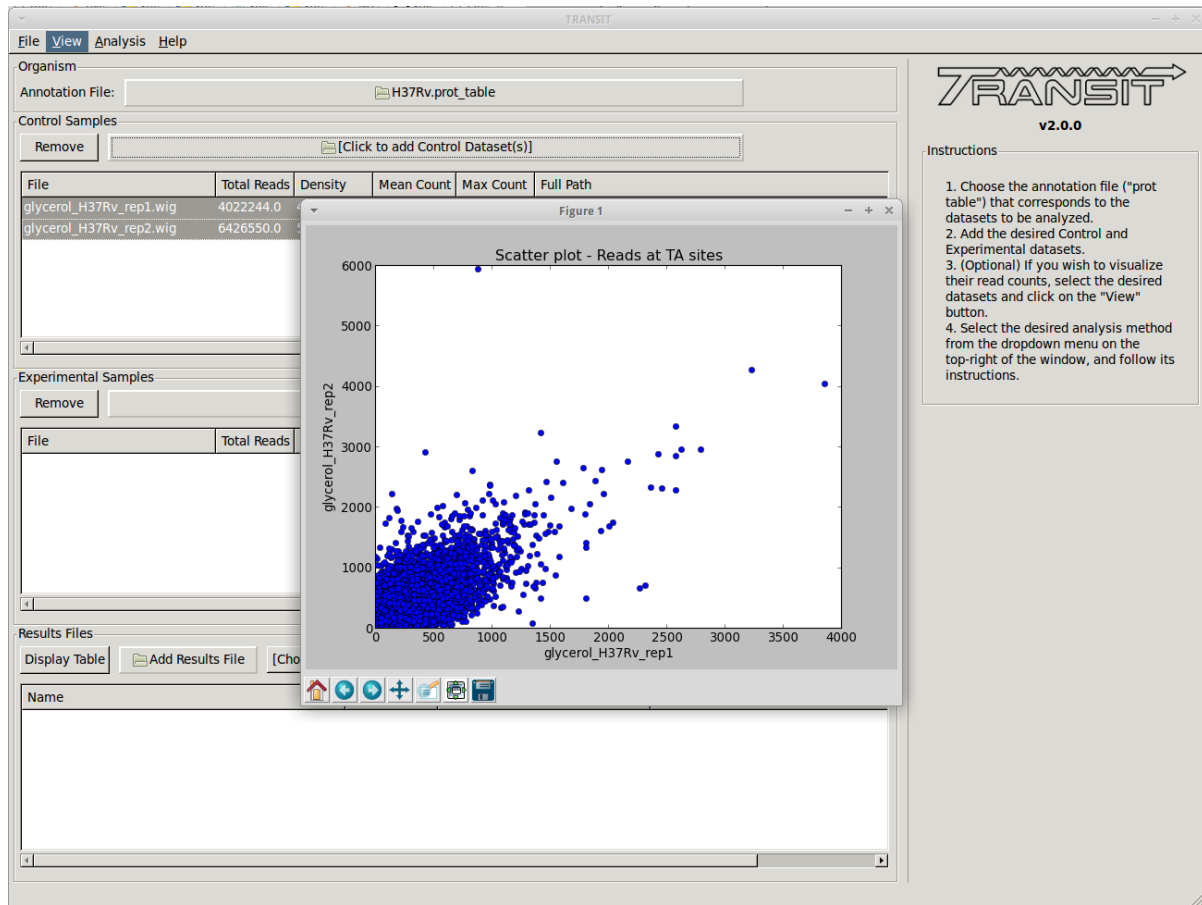
- TRANSIT is capable of analyzing TnSeq libraries constructed with Himar1 or *Tn5* datasets.
- TRANSIT assumes you have already done pre-processing of raw sequencing files (.fastq) and extracted read counts into a *.wig formatted file*. The .wig file should contain the counts at all sites where an insertion 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 2 main types of essentiality analyses: individual, comparative. 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 (insertion density 20-30%). But both methods have their merits and are complementary. For comparative analysis, TRANSIT uses ‘re-sampling’, which is analogous to a permutation test, to determine if the sum of read counts differs significantly between two conditions. Hence this can be used to identify conditionally essential regions and quantify the statistical significance.
- 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.

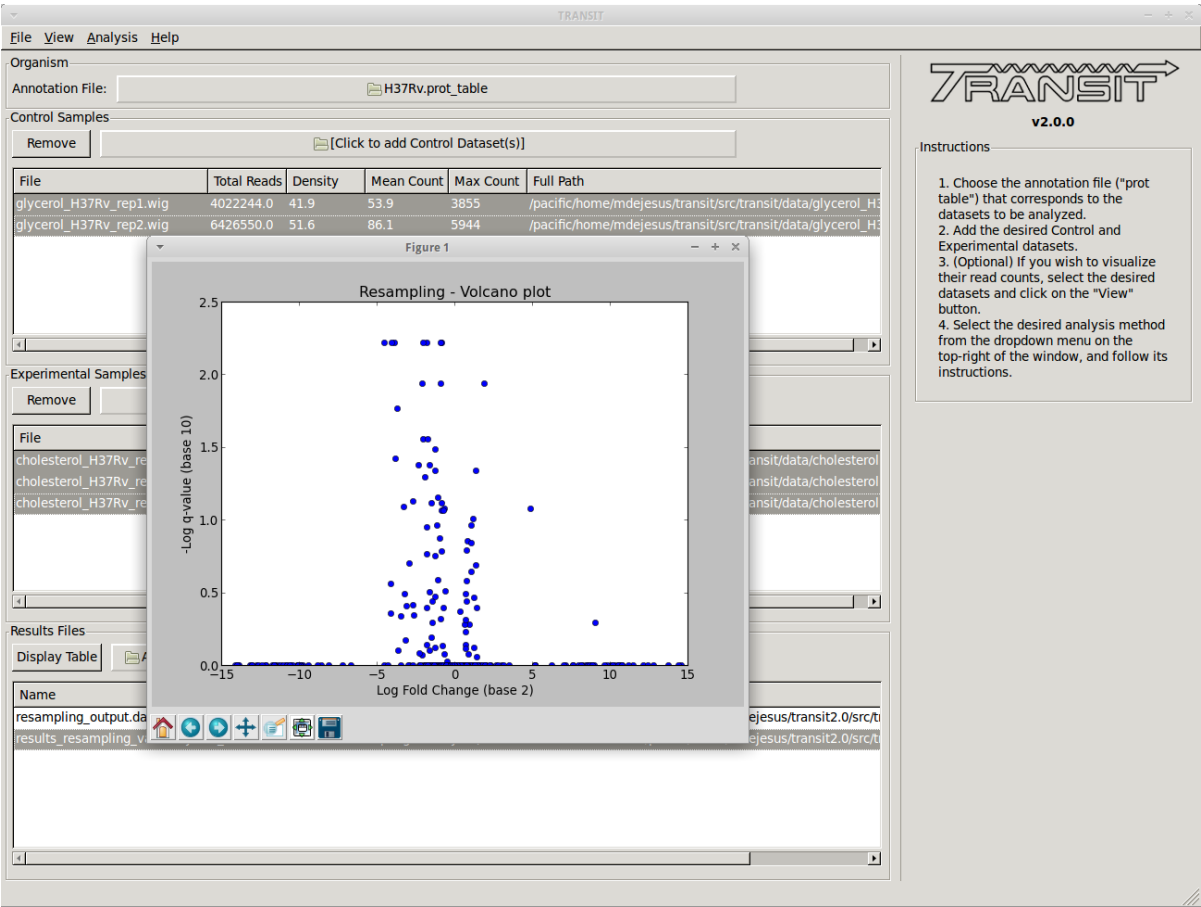
- TRANSIT incorporates many interesting ways of looking at your data.
- Track view shows you a visual representation of the read counts at each site at a locus of interest (for selected datasets) somewhat like IGV.

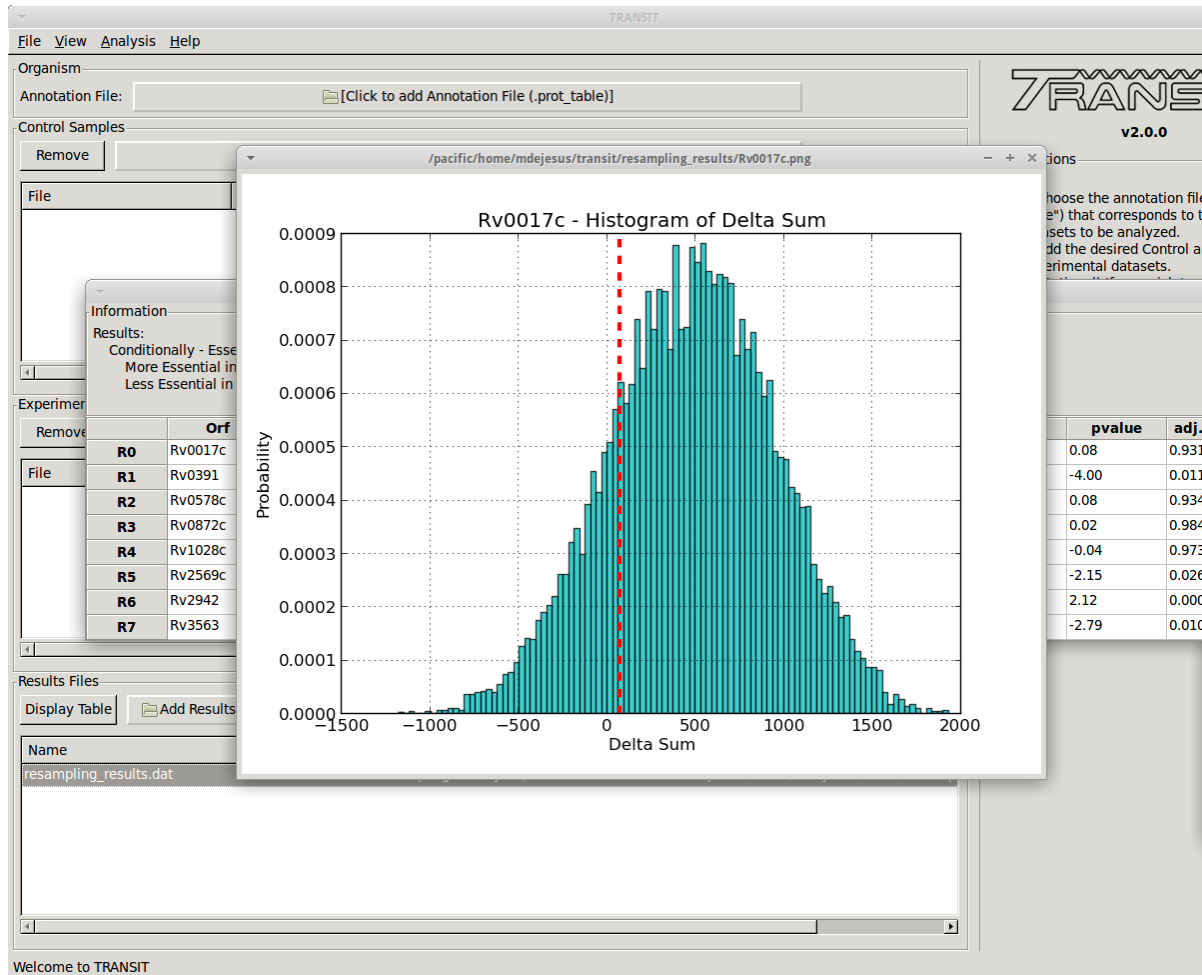


- Scatter plots can show the correlation of counts between 2 datasets.



+ 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 re-sample distributions for each gene.





- 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 instantaneous, 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 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.

2.1.2 Developers

Name	Time Active	Contact Information
Michael A. DeJesus	2015-Present	http://students.cs.tamu.edu/mad
Thomas R. Ioerger	2015-Present	http://faculty.cs.tamu.edu/ioerger/
Chaitra Ambadipudi	2015	
Eric Nelson	2016	

2.1.3 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. It can be downloaded with git as follows:

```
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

The following libraries/modules are required to run TRANSIT:

- [Python 2.7](#)
- [Numpy](#) (tested on 1.13.0)
- [Scipy](#) (tested on 0.19.1)
- [matplotlib](#) (tested on 2.0.2)
- [wxpython 2.8.0+](#) (for Mac OSX, use the **cocoa** version of wxPython; If using El Capitan, please see [OSX El Capitan notice](#) for special instructions)
- [PIL \(Python Imaging Library\)](#) or [Pillow](#).

Generally, these requirements are install using the appropriate methods for your operating system, i.e. apt-get or yum for unix machines, pip or easy_install for OSX, or binary installers on Windows. Below more detailed instructions are provided.

2.2.2 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. See the documentation of the package for further examples:

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

See also:

transit

2.2.3 Detailed Instructions: Linux

Method 1: Install as a Python Package

Most of the requirements are available in default package sources in most Linux distributions. The following commands will install python, pip, wxPython, and several other dependencies needed by the python modules:

```
#Ubuntu:
sudo apt-get install python python-dev python-pip pkg-config python-wxgtk3.0 libpng-
↳ dev libjpeg8-dev libfreetype6-dev

#Fedora:
sudo yum install python python-dev python-pip pkg-config python-wxgtk3.0 libpng-dev_
↳ libjpeg8-dev libfreetype6-dev
```

Finally 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*.

Method 2: Install Source Locally

Most of the requirements are available in default package sources in most Linux distributions. The following commands will install python, numpy, scipy, matplotlib on the Ubuntu or Fedora Linux distributions:

```
#Ubuntu:
sudo apt-get install python python-numpy python-scipy python-matplotlib python-wxgtk3.
↪0

#Fedora:
sudo yum install python numpy scipy python-matplotlib python-wxgtk3.0
```

The final requirement left to install is Pillow. First you need install pip which simplifies the process of installing certain python modules like Pillow:

```
#Ubuntu:
sudo apt-get install pip

#Fedora:
sudo yum install pip
```

Next, using pip you must have a clean installation of Pillow, and the desired libraries. You can achieve this through the following commands:

```
#Ubuntu:
pip uninstall pillow
pip uninstall Pillow
sudo apt-get install libjpeg-dev zlib1g-dev
pip install -I Pillow

#Fedora:
pip uninstall pillow
pip uninstall Pillow
sudo yum install install libjpeg-dev zlib1g-dev
pip install -I Pillow
```

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

2.2.4 Detailed Instructions: OSX

Method 1: Install as a Python Package

First, download and install the latest Python 2.7.x installation file from the official python website:

- <http://www.python.org/downloads/>

Next make sure you have pip installed. Pip can be installed through easy_install, which should come with OSX:

```
sudo easy_install pip
```

Download and install the OSX binary of wxpython (cocoa version) for python 2.7:

- <http://downloads.sourceforge.net/wxpython/wxPython3.0-osx-3.0.2.0-cocoa-py2.7.dmg>

Note: If you are running OSX El Capitan or later, you will need to use a repackaged version of the wxpython installer. You can [download a repackaged version from our servers](#) or you can follow [these detailed instructions to repackage the installer](#) if you prefer.

Finally you can use pip to install the TRANSIT package:

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

This will automatically download and install TRANSIT and all remaining requirements.

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

Method 2: Install Source Locally

First, download and install the latest Python 2.7.x installation file from the official python website:

- <http://www.python.org/downloads/>

Next make sure you have pip installed. Pip can be installed through easy_install, which should come with OSX:

```
sudo easy_install pip
```

Next install numpy, scipy, and matplotlib and pillow using pip:

```
sudo pip install numpy
sudo pip install scipy
sudo pip install matplotlib
sudo pip install pillow
```

Download and install the OSX binary of wxpython (cocoa version) for python 2.7:

- <http://downloads.sourceforge.net/wxpython/wxPython3.0-osx-3.0.2.0-cocoa-py2.7.dmg>

Note: If you are running OSX El Capitan or later, you will need to use a repackaged version of the wxpython installer. You can [download a repackaged version from our servers](#) or you can follow [these detailed instructions to repackage the installer](#) if you prefer.

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

2.2.5 Detailed Instructions: Windows

Method 1: Install as a Python Package

First, download and install the latest Python 2.7.x installation file from the official python website:

- <http://www.python.org/downloads/>

Next, you will need to install pip. If you are using python 2.7.9+ then pip will come pre-installed and included in the default script directory (i.e. C:\Python27\Scripts). If you are using python 2.7.8 or older, you will need to manually install pip by downloading and running the `get-pip.py` script:

```
python.exe get-pip.py
```

Make sure that “wheel” is installed. This is necessary to allow you to install .whl (wheel) files:

```
pip.exe install wheel
```

Next install the transit package using pip:

```
pip.exe install tnseq-transit
```

To use transit in GUI mode you will need to install wxPython versions 3.0 or earlier. We have provided .whl files which you can download and install below. (Note: Make sure to choose the files that match your Windows version i.e. 32/64 bit)

- `wxPython-3.0.2.0-cp27-none-win_amd64.whl` or [32 bit]
- `wxPython_common-3.0.2.0-py2-none-any.whl` or [32 bit]

Finally, install the files using pip:

```
pip.exe install wxPython-3.0.2.0-cp27-none-win_amd64.whl  
pip.exe install wxPython_common-3.0.2.0-py2-none-any.whl
```

making sure to replace the name with the file you downloaded (i.e. 32bit vs 64 bit)

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

Method 2: Install Source Locally

First, download and install the latest Python 2.7.x installation file from the official python website:

- <http://www.python.org/downloads/>

Next, you will need to install pip. If you are using python 2.7.9+ then pip will come pre-installed and included in the default script directory (i.e. C:\Python27\Scripts). If you are using python 2.7.8 or older, you will need to manually install pip by downloading and running the `get-pip.py` script:

```
python.exe get-pip.py
```

Make sure that “wheel” is installed. This is necessary to allow you to install .whl (wheel) files:

```
pip.exe install wheel
```

Download the .whl files for all the requirements (Note: Make sure to choose the files that match your Windows version i.e. 32/64 bit)

- `numpy-1.9.2+mkl-cp27-none-win_amd64.whl` or [32 bit]
- `scipy-0.15.1-cp27-none-win_amd64.whl` or [32 bit]
- `matplotlib-1.4.3-cp27-none-win_amd64.whl` or [32 bit]
- `Pillow-2.8.2-cp27-none-win_amd64.whl` or [32 bit]
- `wxPython-3.0.2.0-cp27-none-win_amd64.whl` or [32 bit]
- `wxPython_common-3.0.2.0-py2-none-any.whl` or [32 bit]

Source: These files were obtained from the Unofficial Windows Binaries for Python Extension Packages by Christoph Gohlke, Laboratory for Fluorescence Dynamics, University of California, Irvine.

Finally, install the files using pip:

```
pip.exe install numpy-1.9.2+mkl-cp27-none-win_amd64.whl
pip.exe install scipy-0.15.1-cp27-none-win_amd64.whl
pip.exe install matplotlib-1.4.3-cp27-none-win_amd64.whl
pip.exe install Pillow-2.8.1-cp27-none-win_amd64.whl
pip.exe install wxPython-3.0.2.0-cp27-none-win_amd64.whl
pip.exe install wxPython_common-3.0.2.0-py2-none-any.whl
```

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

2.2.6 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:

- <http://sourceforge.net/projects/bio-bwa/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`

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

2.2.7 Troubleshooting

1. Gtk-ERROR **: GTK+ 2.x symbols detected

This error can occur if you have GTK2 already installed and then install wxPython version 3.0+. To fix this, please try installing version 2.8 of wxPython or install a new version of GTK3. More information on this error to come.

2. wxPython & OSX: “The Installer could not install the software because there was no software found to install.”

If you are running OSX El Capitan or later, you will need to use a repackaged version of the wxpython installer as OSX El Capitan has removed support for older packaging methods still used by wxPython. You can [download a repackaged version of wxPython from our servers](#) or you can follow [these detailed instructions to repackage the installer](#) if you prefer.

3. No window appears when running in GUI mode.

This problem is likely due to running an unsupported version of matplotlib. Please download and install the version 2.0.2. You can download and manually install the source from the following location:

- [matplotlib-1.4.3](#)

Or, if you have pip installed, you can install using pip and specify the desired version:

```
pip install 'matplotlib' --force-reinstall
```

4. Unable to locate package python-wxgtk3.0

Your version of Linux might not have the repository address that includes python-wxgtk3.0. You can attempt to install version 2.8 instead:

```
sudo apt-get install python-wxgtk2.8
```

or you can add the repository that includes version 3.0 and install it:

```
# Add repo for 14.04
sudo add-apt-repository "deb http://archive.ubuntu.com/ubuntu utopic main restricted_
↳universe"

#Update repo information
sudo apt-get update

#Install wxPython 3.0
sudo apt-get install python-wxgtk3.0

#Remove repo to prevent version conflicts
sudo add-apt-repository --remove "deb http://archive.ubuntu.com/ubuntu utopic main_
↳restricted universe"
```

5. 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
```

6. 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
```

7. 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
```

8. “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

2.3.1 GUI Mode

To run TRANSIT in GUI mode (should be the same on Linux, Windows and MacOS), from the command line run:

```
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, because TRANSIT has a graphical user interface, if you are trying to run TRANSIT across a network, for example, 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).

2.3.2 Command line Mode

TRANSIT can also be run from the command line, without the GUI interface. This is convenient if you want to run many analyses in batch, as you can write a script that automatically runs that automatically runs TRANSIT from the command line. TRANSIT expects the user to specify which analysis method they wish to run. The user can choose from “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:

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

Gumbel

To run the Gumbel analysis from the command line, type “python PATH/src/transit.py gumbel” followed by the following arguments:

Argument	Type	Description	Default	Example
annotation	Required	Path to annotation file in .prot_table format		genomes/H37Rv. prot_table
control_files	Required	Comma-separated list of paths to the *.wig replicate datasets		data/glycerol_reads_rep1.wig,data/glycerol_reads_rep2.wig
output_file	Required	Name of the output file with the results.		results/gumbel_glycerol.dat
-s SAMPLES	Optional	Number of samples to take.	10000	-s 20000
-m MINREAD	Optional	Smallest read-count considered to be an insertion.	1	-m 2
-b BURNIN	Optional	Burn in period, Skips this number of samples before getting estimates. See documentation.	500	-b 100
-t TRIM	Optional	Number of samples to trim. See documentation.	1	-t 2
-r REP	Optional	How to handle replicates read-counts: 'Sum' or 'Mean'.	Sum	-r Mean
-iN IGNOREN	Optional	Ignore TAs occurring at X% of the N terminus.	5	-iN 0
-iC IGNOREC	Optional	Ignore TAs occurring at X% of the C terminus.	5	-iC 10

```
python PATH/src/transit.py gumbel genomes/H37Rv.prot_table data/glycerol_reads_rep1.
↪ wig,data/glycerol_reads_rep2.wig test_console_gumbel.dat -s 20000 -b 1000
```

Tn5 Gaps

To run the Tn5 Gaps analysis from the command line, type “python PATH/src/transit.py tn5gaps” followed by the following arguments:

Argument	Type	Description	Default	Example
annotation	Required	Path to annotation file in .prot_table format		genomes/Salmonella-Ty2.prot_table
control_files	Required	Comma-separated list of paths to the *.wig replicate datasets		data/salmonella_2122_rep1.wig,data/salmonella_2122_rep2.wig
output_file	Required	Name of the output file with the results.		results/test_console_tn5gaps.dat
-m MINREAD	Optional	Smallest read-count considered to be an insertion.	1	-m 2
-r REP	Optional	How to handle replicates read-counts: 'Sum' or 'Mean'.	Sum	-r Sum

Example Tn5 Gaps command:

```
python PATH/src/transit.py tn5gaps genomes/Salmonella-Ty2.prot_table data/salmonella_
↪ 2122_rep1.wig,data/salmonella_2122_rep2.wig results/test_console_tn5gaps.dat -m 2 -
↪ r Sum
```

Example HMM command:

```
python PATH/src/transit.py hmm genomes/H37Rv.prot_table data/glycerol_reads_rep1.wig,
↪ data/glycerol_reads_rep2.wig test_console_hmm.dat -r Sum
```

Resampling

To run the Resampling analysis from the command line, type “python PATH/src/transit.py resampling” followed by the following arguments:

Argument	Type	Description	Default	Example
annotation	Required	Path to annotation file in .prot_table format		genomes/H37Rv.prot_table
control_files	Required	Comma-separated list of paths to the *.wig replicate datasets for the control condition		data/glycerol_reads_rep1.wig,data/glycerol_reads_rep2.wig
exp_files	Required	Comma-separated list of paths to the *.wig replicate datasets for the experimental condition		data/cholesterol_reads_rep1.wig,data/cholesterol_reads_rep2.wig
output_file	Required	Name of the output file with the results.		results/gumbel_glycerol.dat
-s SAMPLES	Optional	Number of permutations performed.	10000	-s 5000
-H	Optional	Creates histograms of the permutations for all genes.	Not set	-H
-a	Optional	Performs adaptive approximation to resampling.	Not set	-a
-N	Optional	Select which normalizing procedure to use. Can choose between ‘TTR’, ‘nzmean’, ‘totreads’, ‘zinfnb’, ‘betageom’, and ‘nonorm’. See the parameters section for the Re-sampling method for a description of these normalization options.	nzmean	-N nzmean
-iN IGNORE NOREN	Optional	Ignore TAs occurring at X% of the N terminus.	5	-iN 0
-iC IGNORE NOREC	Optional	Ignore TAs occurring at X% of the C terminus.	5	-iC 10

Example Resampling command:

```
python PATH/src/transit.py resampling genomes/H37Rv.prot_table data/glycerol_reads_
→ repl1.wig,data/glycerol_reads_rep2.wig data/cholesterol_reads_rep1.wig,data/
→ cholesterol_reads_rep2.wig,data/cholesterol_reads_rep3.wig test_console_resampling.
→ dat -H -s 10000 -N nzmean
```

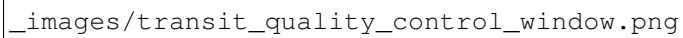
2.4 Features

TRANSIT has several useful features to help inspect the quality of datasets as and export them to different formats.

2.4.1 Quality Control

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.

The image is a large, empty rectangular box with a thin black border, intended for a screenshot of the transit quality control window. The text "_images/transit_quality_control_window.png" is located at the bottom left of this box.

_images/transit_quality_control_window.png

QC Metrics Table

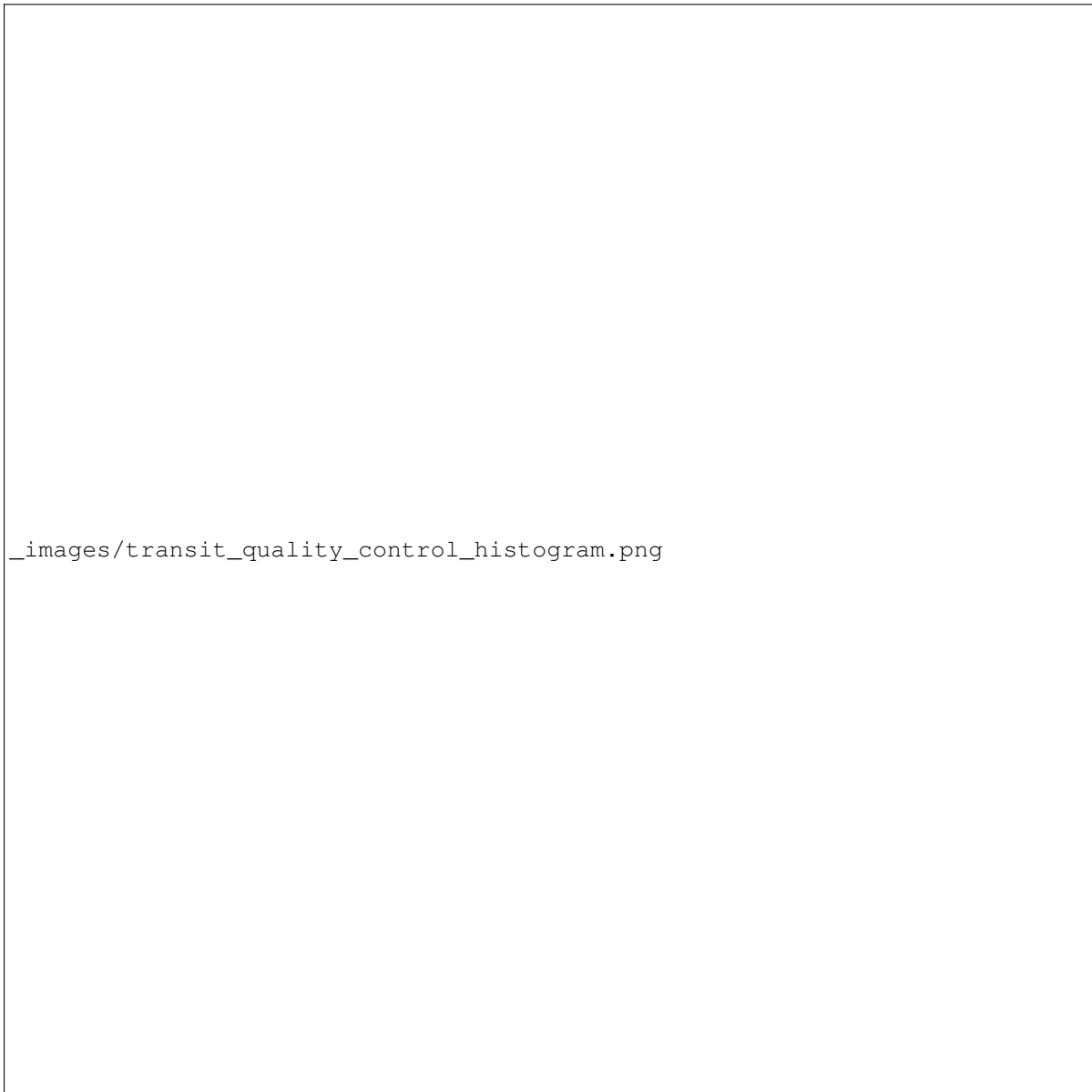
The Quality Control window contains a table of the datasets and metrics, similar to the one in the main TRANSIT interface. This table has an extended set of metrics to provide a better picture of the quality of the datasets:

Column Header	Column Definition	Comments
File	Name of dataset file.	
Density	Fraction of sites with insertions.	“Well saturated” Himar1 datasets have >30% saturation. Beneath this, statistical methods may have trouble.
Mean Read	Average read-count, including empty sites.	
NZMean Read	Average read-count, excluding empty sites.	A value between 30-200 is usually good for Himar1 datasets. Too high or too low can indicate problems.
NZMedian Read	Median read-count, excluding empty sites.	As read-counts can often have spikes, median serves as a good robust estimate.
Max Read	Largest read-count in the dataset.	Useful to determine whether there are outliers/spikes, which may indicate sequencing issues.
Total Reads	Sum of total read-counts in the dataset.	Indicates how much sequencing material was obtained. Typically >1M reads is desired for Himar1 datasets.
Skew	Skew of read-counts in the dataset.	Large skew may indicate issues with a dataset. Typically a skew < 50 is desired. May be higher when library is under strong selection
Kurtosis	Kurtosis of the read-counts in the dataset.	

QC Figures

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:

Figure 1: Histogram of Reads



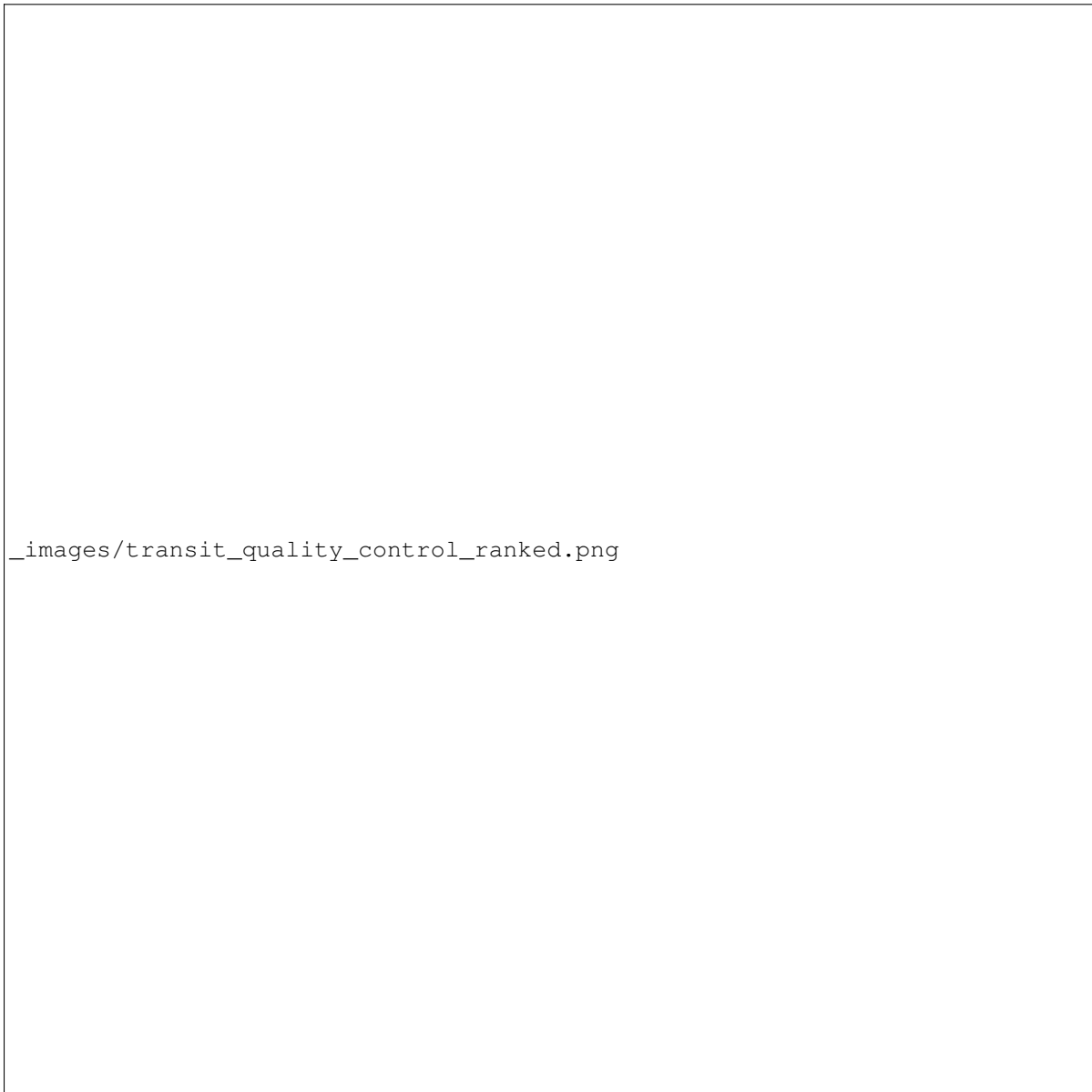
The first plot in the Quality Control window is a histogram of the non-zero read-counts in the selected dataset. While read-counts are not truly geometrically distributed, “well-behaved” datasets often look “Geometric-like”, i.e. low counts are more frequent than very large counts. Datasets which where this is not the case may reflect a problem.

Figure 2: QQ Plot of Reads vs Geometric Distribution

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.

Figure 3: Ranked plot of Read-Counts



The second plot in the Quality Control window is a plot of the read-counts in sorted order. This may be helpful in identifying 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).

2.5 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 [\[DeJesus2013\]](#):

DeJesus, M.A., Zhang, Y.J., Sassetti, C.M., Rubin, E.J., Sacchettini, J.C., and Ioerger, T.R. (2013). Bayesian analysis of gene essentiality based on sequencing of transposon insertion libraries. *Bioinformatics*, 29(6):695-703.

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 sample 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 effect 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:

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 ($< \sim 30\%$), 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 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 based on the original gumbel analysis method described in this paper:

Griffin, J.E., Gawronski, J.D., DeJesus, M.A., Ioerger, T.R., Akerley, B.J., Sasseti, C.M. (2011). [High-resolution phenotypic profiling defines genes essential for mycobacterial survival and cholesterol catabolism](#). *PLoS Pathogens*, 7(9):e1002251.

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:

Langridge GC1, Phan MD, Turner DJ, Perkins TT, Parts L, Haase J, Charles I, Maskell DJ, Peters SE, Dougan G, Wain J, Parkhill J, Turner AK. (2009). [Simultaneous assay of every Salmonella Typhi gene using one million transposon mutants](#). *Genome Res.*, 19(12):2308-16.

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.

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 summing read counts across datasets. This should not have an effect for the Gumbel method, aside from potentially affecting spurious reads.

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:

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

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.

2.5.3 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](#)]:

DeJesus, M.A., Ioerger, T.R. A Hidden Markov Model for identifying essential and growth-defect regions in bacterial genomes from transposon insertion sequencing data. *BMC Bioinformatics*. 2013. 14:303

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, it 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:

Column #	Column Definition
1	Coordinate of TA site
2	Observed Read Counts
3	Probability for ES state
4	Probability for GD state
5	Probability for NE state
6	Probability for GA state
7	State Classification (ES = Essential, GD = Growth Defect, NE = Non-Essential, GA = Growth-Defect)
8	Gene(s) that share(s) the TA site.

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.

Column Header	Column Definition
Orf	Gene ID
Name	Gene Name
Desc	Gene Description
N	Number of TA sites
n0	Number of sites labeled ES (Essential)
n1	Number of sites labeled GD (Growth-Defect)
n2	Number of sites labeled NE (Non-Essential)
n3	Number of sites labeled GA (Growth-Advantage)
Avg. Insertions	Mean insertion rate within the gene
Avg. Reads	Mean read count within the gene
State Call	State Classification (ES = Essential, GD = Growth Defect, NE = Non-Essential, GA = Growth-Defect)

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.

2.5.4 Re-sampling

The re-sampling method is a comparative analysis that allows that can be used 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.

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 genes are determined for each replicate of each condition. The total read-counts in condition A is subtracted from the total read counts at condition B, to obtain an observed difference in read counts. 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.

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 re-sampling 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.
- **Include Zeros:** By default resampling will ignore sites that are zero across all the datasets (i.e. completely empty), which is useful for decreasing running time (specially for large datasets like Tn5). This option allows the user to include these empty rows.
- **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.

Output and Diagnostics

The re-sampling 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.

Column Header	Column Definition
Orf	Gene ID.
Name	Name of the gene.
Description	Gene description.
N	Number of TA sites in the gene.
TAs Hit	Number of TA sites with at least one insertion.
Sum Rd 1	Sum of read counts in condition 1.
Sum Rd 2	Sum of read counts in condition 2.
Delta Rd	Difference in the sum of read counts.
p-value	P-value calculated by the permutation test.
p-adj.	Adjusted p-value controlling for the FDR (Benjamini-Hochberg)

Run-time

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

2.5.5 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.

2.6 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>
```

Positional Arguments	Definition
<method>	Short name of the desired analysis method e.g. gumbel, resampling, hmm
<wig-files>	Comma-separated list of paths read-count datasets in .wig format
<annotation>	Path to the annotation in .prot_table or .GFF3 format.
<output>	Desired path and name of the output file

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.

Positional Arguments	Argu-ments	Definition
<method>		Short name of the desired analysis method e.g. gumbel, resampling, hmm
<control-files>		Comma-separated list of paths read-count files in .wig format for the control datasets
<experimental-files>		Comma-separated list of paths read-count files in .wif format for the experimental datasets
<annotation>		Path to the annotation in .prot_table or .GFF3 format.
<output>		Desired path and name of the output file

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.

Positional Arguments	Definition
<wig-files>	Comma-separated list of paths read-count datasets in .wig format
<annotation>	Path to the annotation in .prot_table or .GFF3 format.
<output>	Desired path and name of the output file

Argument	Definition
-n <String>	Short name of the normalization method, e.g. -n TTR

```
python transit.py norm glycerol_H37Rv_rep1.wig, glycerol_H37Rv_rep2.wig H37Rv.prot_  
↳table glycerol_TTR.txt -n TTR
```

2.7 Tutorial: Essentiality Analysis in a Single Condition

To illustrate how TRANSIT works, we are going to go through a tutorial where we analyze datasets of H37Rv M. tuberculosis grown on glycerol and cholesterol.

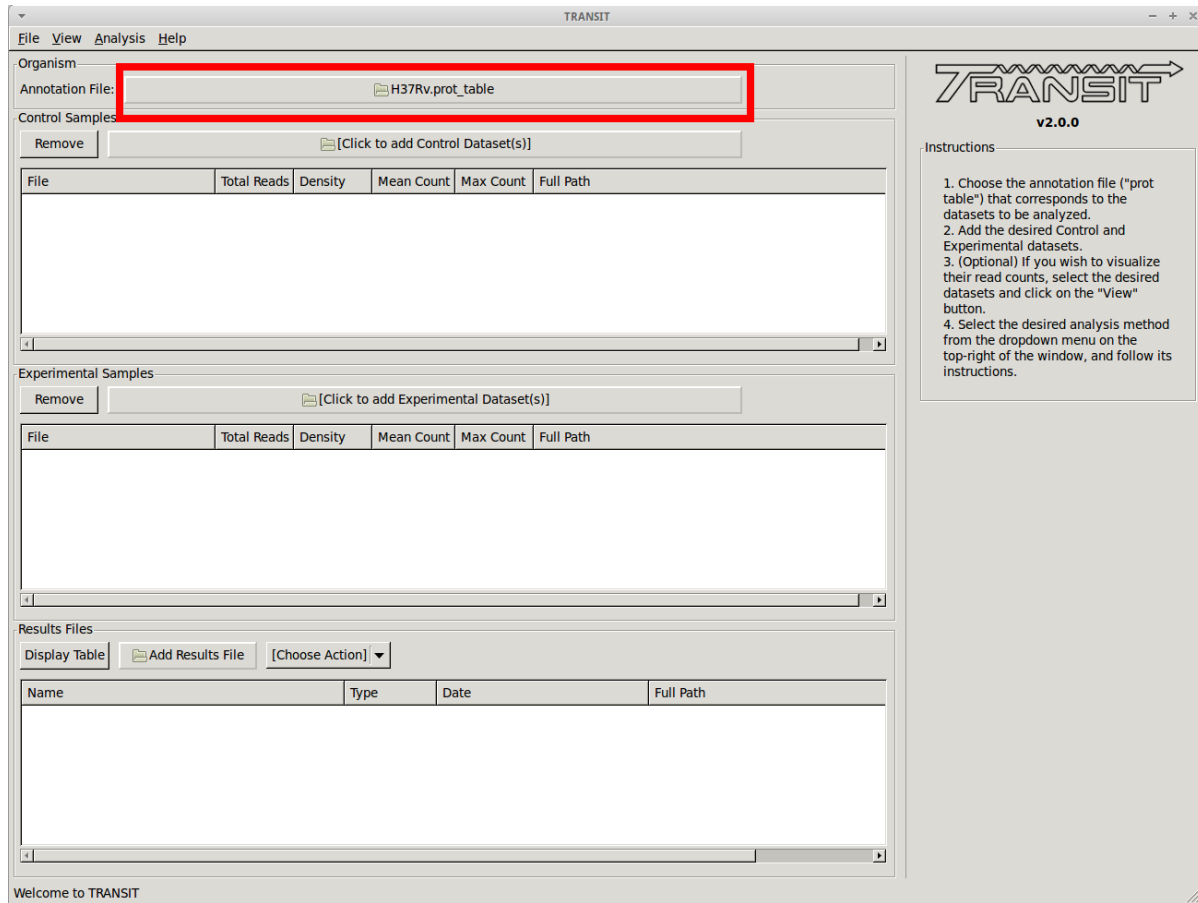
2.7.1 Run TRANSIT

Navigate to the directory containing the TRANSIT files, and run TRANSIT:

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

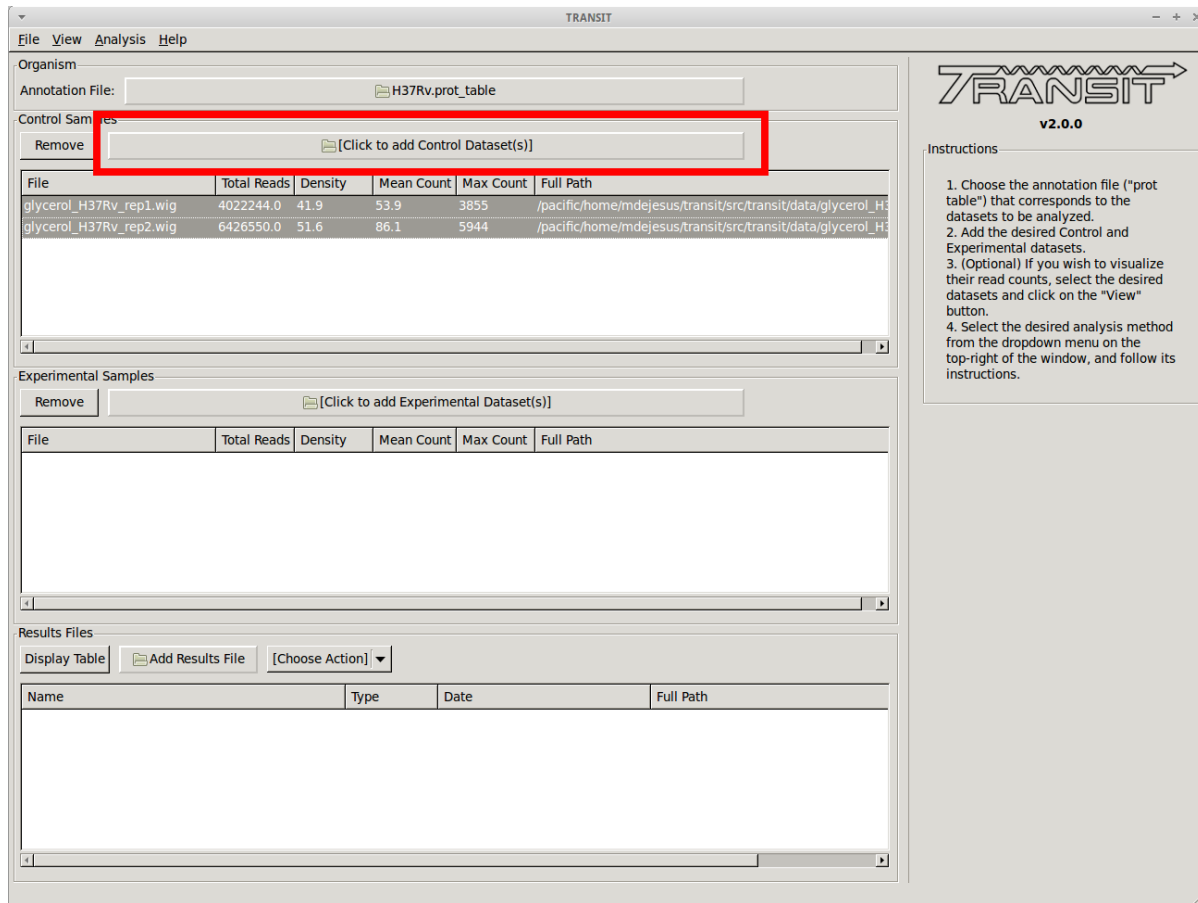
2.7.2 Adding the annotation file

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” format, described above.



2.7.3 Adding the control datasets

We want to analyze datasets grown in glycerol to those grown in cholesterol. We are choosing the datasets grown in glycerol as the “Control” datasets. 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.

2.7.4 Visualizing read counts

TRANSIT allows us to visualize the read-counts of the datasets we have already loaded. To do this, we must select the desired datasets ("Control+Click") and then click on "View -> Track View" in the menu bar at the top of the TRANSIT window. Only those selected datasets will be displayed:

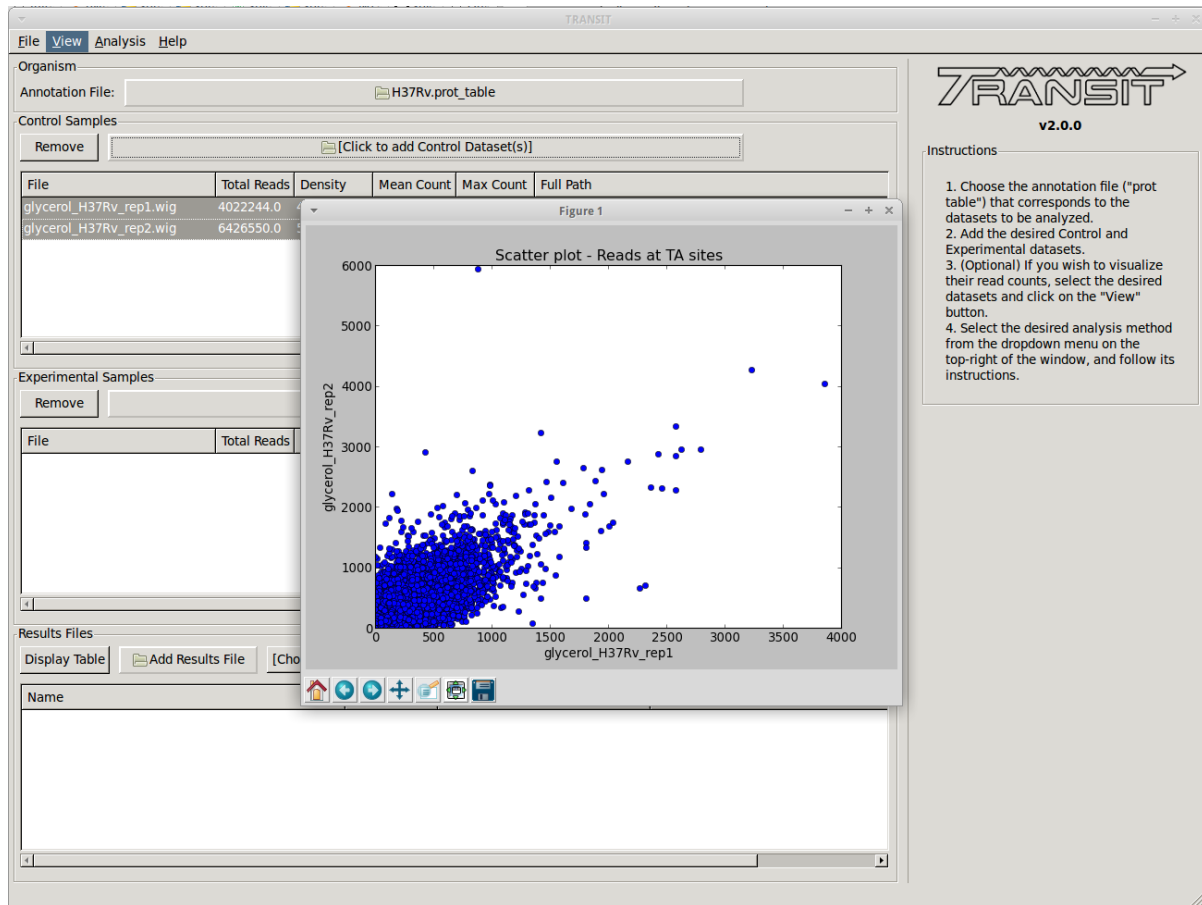


This will open a window that allows that shows a visual representation of the read counts at the TA sites throughout the genome. The scale of the read counts can be set by changing the value of the “Max Read” textbox on the right. We can browse around the genome by clicking on the left and right arrow, or search for a specific gene with the search text box.

This window also allows us to save a .png image of the canvas for future reference if desired (i.e. Save Img button).

2.7.5 Scatter plot

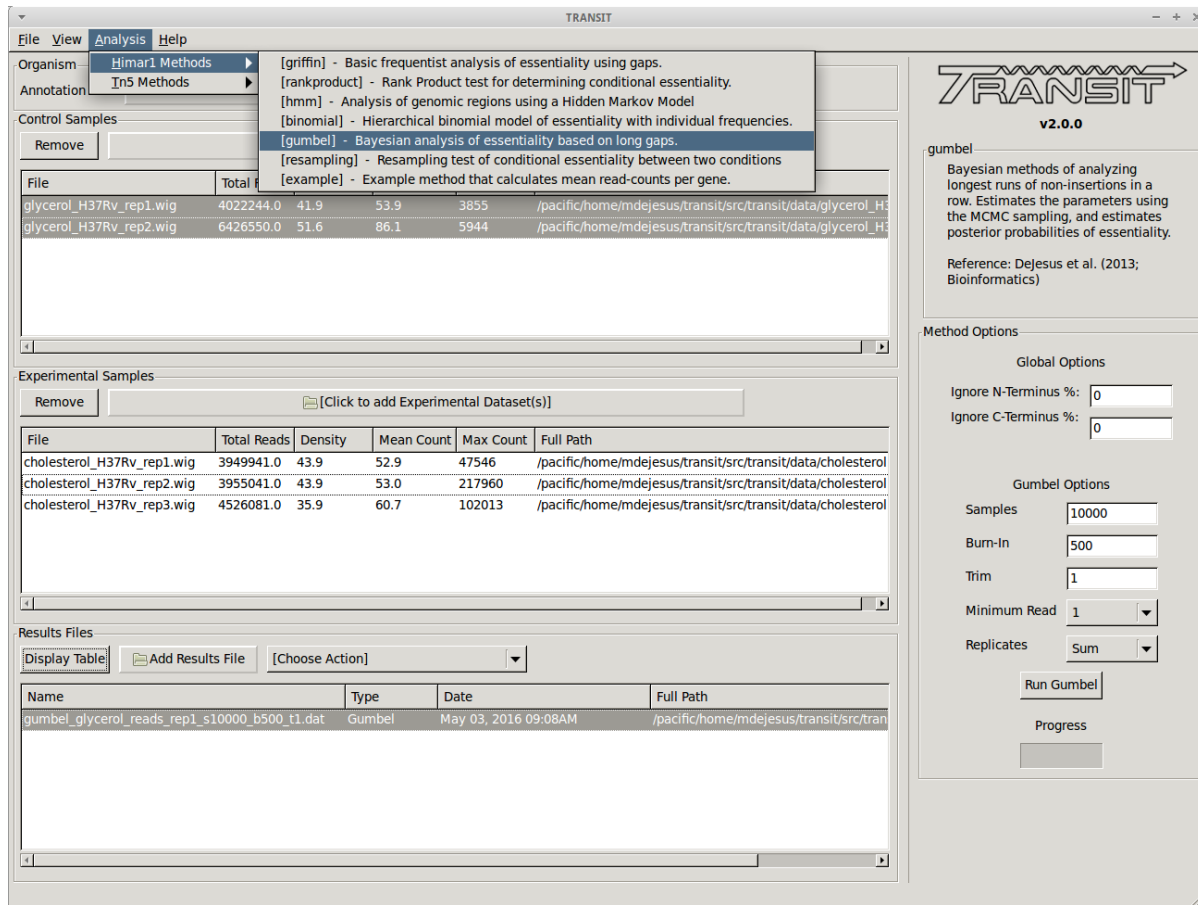
We can also view a scatter plot of read counts of two selected datasets. To achieve this we select two datasets (using “Control + Click”) and then clicking on “View -> Scatter Plot” in the menu bar at the top of the TRANSIT window.



A new window will pop-up, show a scatter plot of both of the selected datasets. This window contains controls to zoom in and out (magnifying glass), allowing us to focus in on a specific area. This is particularly useful when large outliers may throw off the scale of the scatter plot.

2.7.6 Essentiality analysis with the Gumbel method

Before comparing both conditions against each other, we may want to determine which genes are essential in a specific condition to get an idea of the genes which are required. To do this we can use the Gumbel or the HMM methods, which determine essentiality within one condition. First we chose the Gumbel method from the list of (Himar1) analysis methods in the menu on top:



For this particular case we leave the parameters at their default settings as these work with a wide variety of datasets (See above for an explanation of their function). We then click on the “Run Gumbel” button and wait until the analysis finished running. The progress bar will give us information about how much of the analysis is still left. Once the program finishes, the results file is automatically created (with the name chosen at run-time) and it is automatically added to the Results File section at the bottom of TRANSIT. We can visualize the results by selecting this file from the list, and clicking on the “Display Table” button. This will open a new window with a table of results:

From this window we can view results, and sort on a specific column (described above) by clicking on a column header. In addition, the top of this window contains a breakdown of the number of essential and non-essential genes found by the Gumbel method. We can see that 675 genes are found to be essential by the Gumbel method (16%), roughly matching expectations that 15% of the genomes is necessary for growth in bacterial organisms. Clicking on the “Zbar” column we can sort the data on the posterior probability of essentiality. If we sort in descending order, we get those genes which are most likely to be essential on the top. Among these are genes like GyrA (DNA gyrase A) and RpoB (DNA-directed polymerase), which are both well-known essential genes, and which obtain a posterior probability of essentiality of 1.0 (Essential).

2.8 Tutorial: Essentiality Analysis of the Entire Genome

To illustrate how TRANSIT works, we are going to go through a tutorial where we analyze datasets of H37Rv *M. tuberculosis* grown on glycerol and cholesterol.

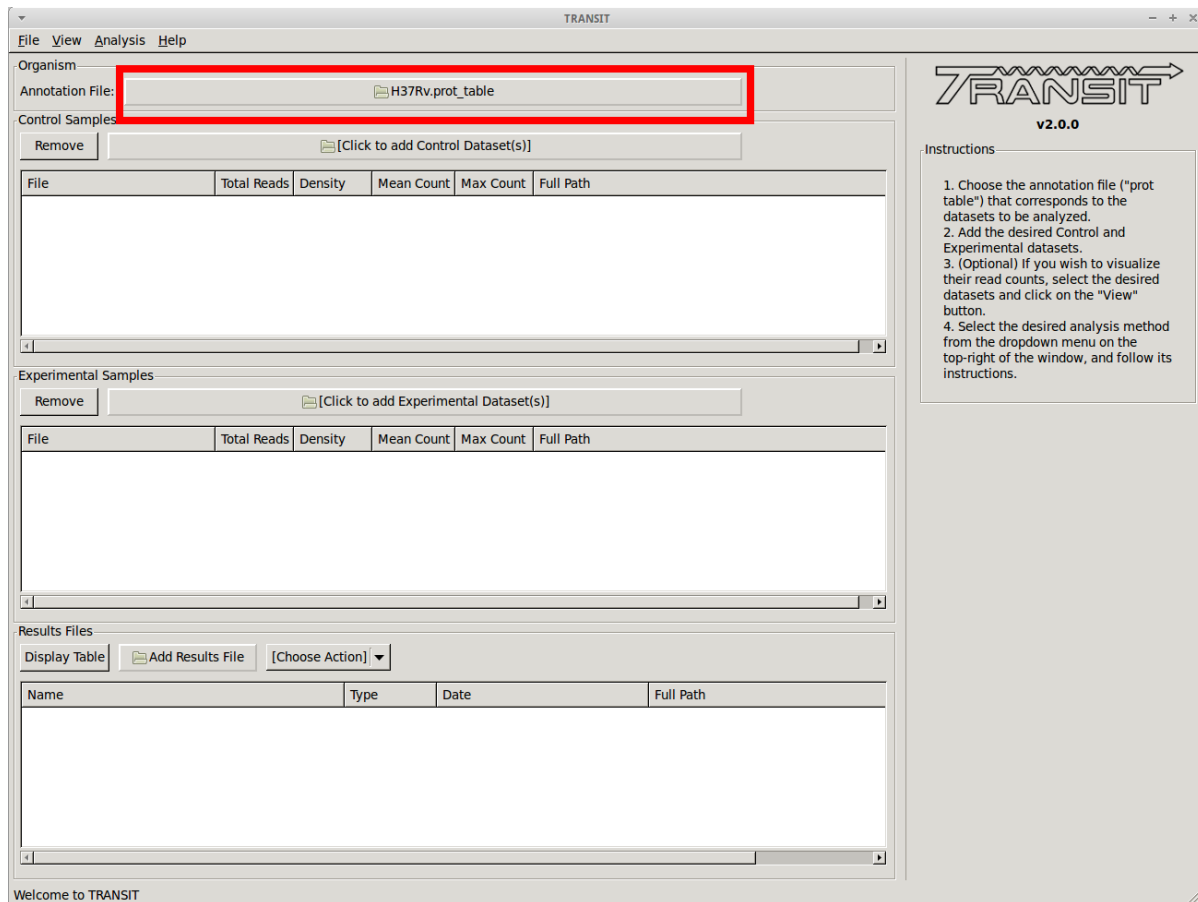
2.8.1 Run TRANSIT

Navigate to the directory containing the TRANSIT files, and run TRANSIT:

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

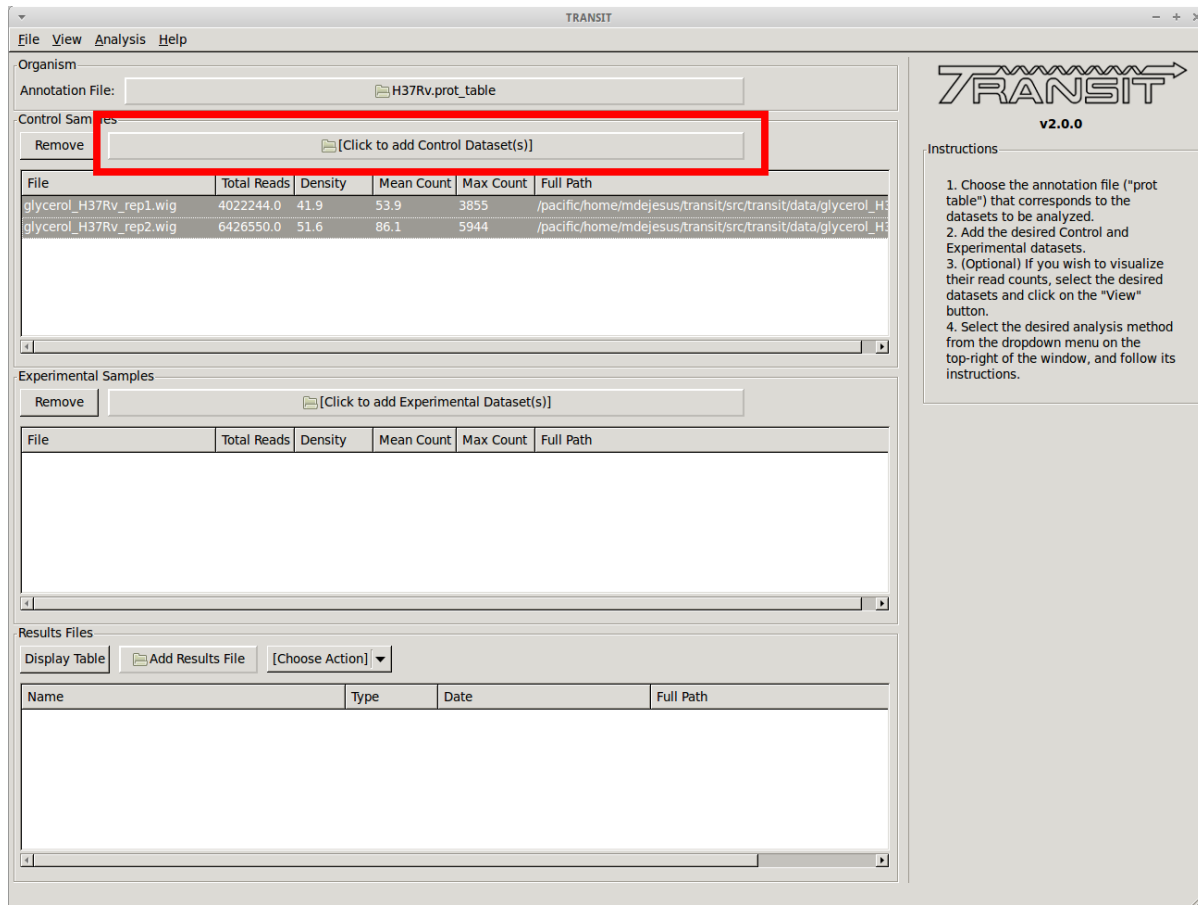
2.8.2 Adding the annotation file

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” format, described above.



2.8.3 Adding the control datasets

We want to analyze datasets grown in glycerol to those grown in cholesterol. We are choosing the datasets grown in glycerol as the “Control” datasets. 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.

2.8.4 Visualizing read counts

TRANSIT allows us to visualize the read-counts of the datasets we have already loaded. To do this, we must select the desired datasets ("Control+Click") and then click on "View -> Track View" in the menu bar at the top of the TRANSIT window. Only those selected datasets will be displayed:

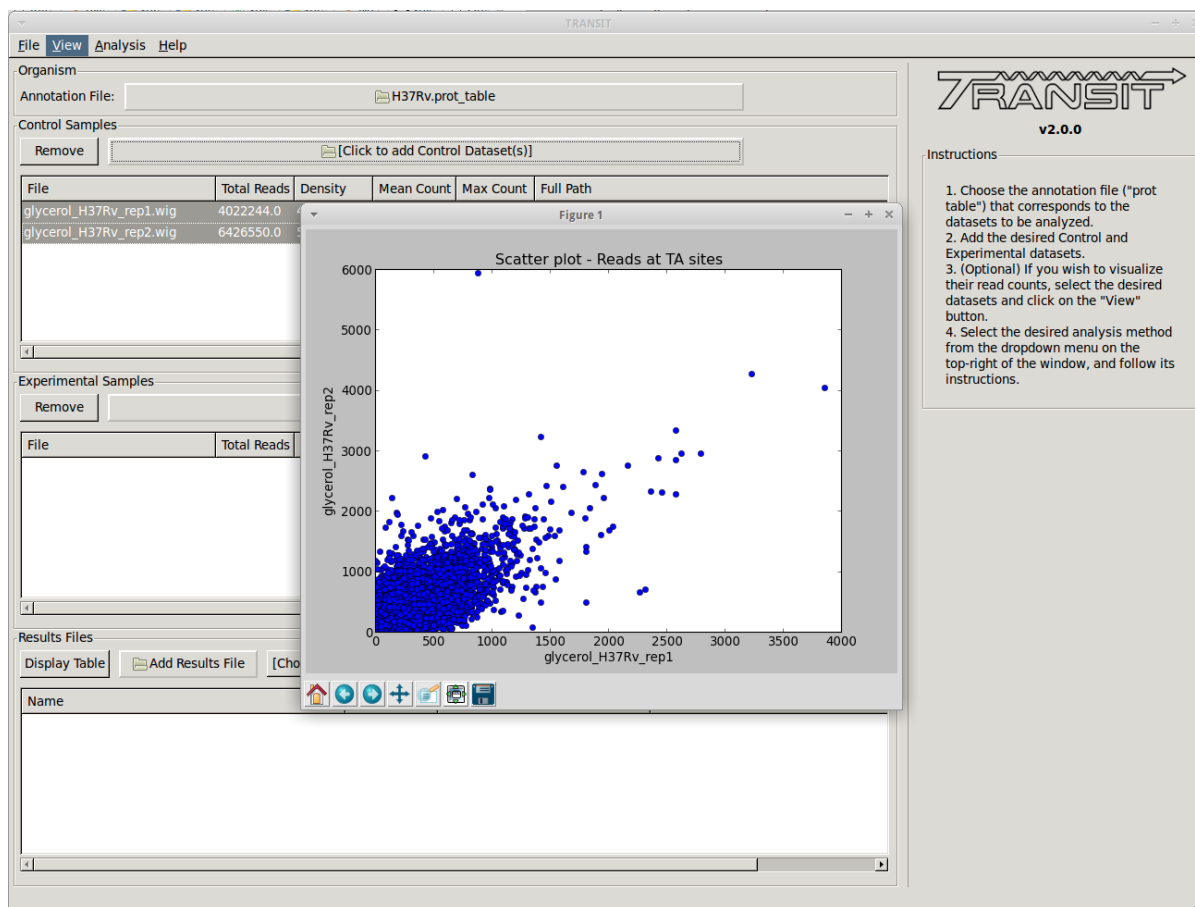


This will open a window that allows that shows a visual representation of the read counts at the TA sites throughout the genome. The scale of the read counts can be set by changing the value of the “Max Read” textbox on the right. We can browse around the genome by clicking on the left and right arrow, or search for a specific gene with the search text box.

This window also allows us to save a .png image of the canvas for future reference if desired (i.e. Save Img button).

2.8.5 Scatter plot

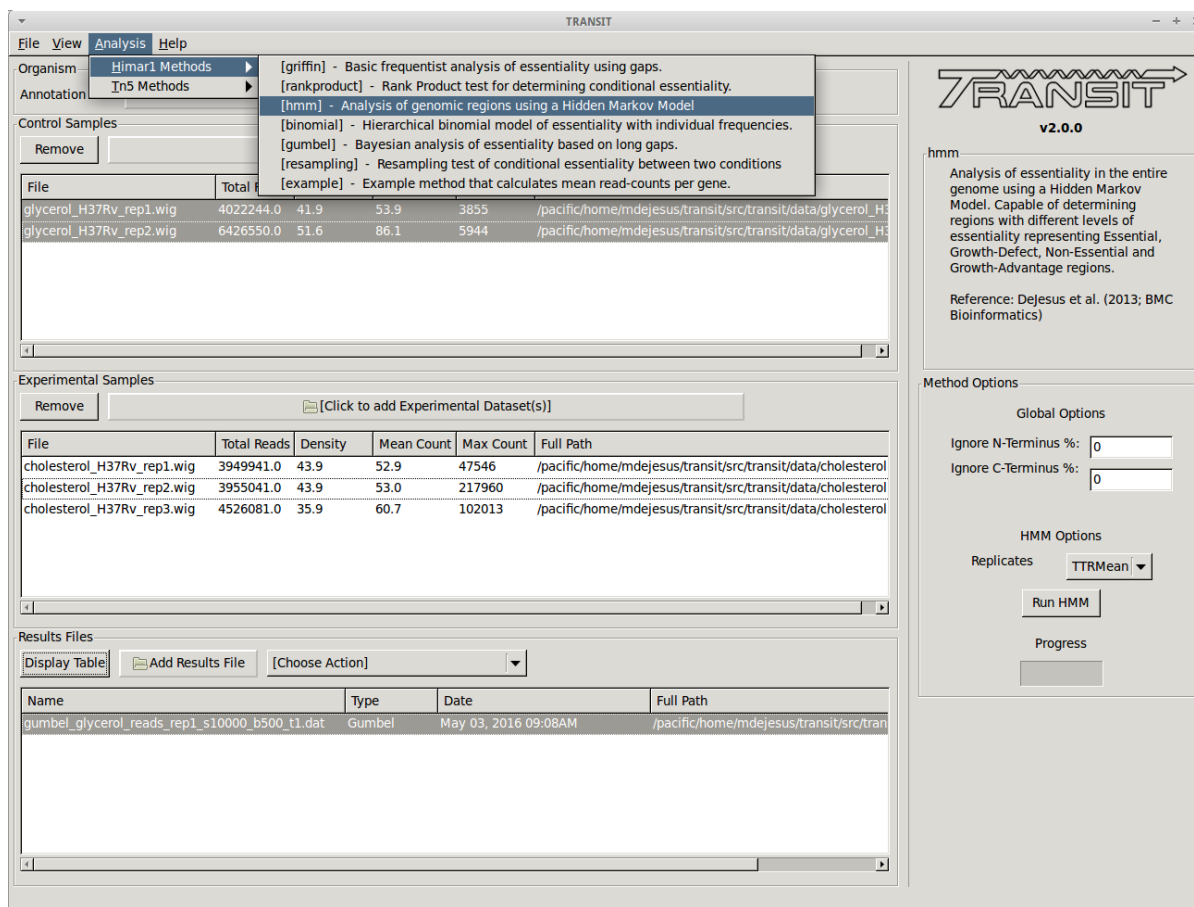
We can also view a scatter plot of read counts of two selected datasets. To achieve this we select two datasets (using “Control + Click”) and then clicking on “View -> Scatter Plot” in the menu bar at the top of the TRANSIT window.



A new window will pop-up, show a scatter plot of both of the selected datasets. This window contains controls to zoom in and out (magnifying glass), allowing us to focus in on a specific area. This is particularly useful when large outliers may throw off the scale of the scatter plot.

2.8.6 Essentiality analysis with the HMM method

An alternative method for determining essentiality is the HMM method. This method differs from the Gumbel method in that it is capable of assessing the essentiality of the entire genome, and is not limited to a gene-level analysis (See above for discussions of the pros and cons of each method). To run the HMM method we select it from the list of (Himar1) methods on the Analysis tab at the top. This automatically displays the available options for the HMM method. Because the HMM method estimates parameters by examining the datasets, there is no need to set parameters for the model. One important option provided is how to deal with replicate datasets. Because the glycerol replicates had a mean read-count between 53-85, we decide to sum read-counts together by selecting "Sum" from the drop-down option.



Finally we click on the “Run HMM” button, and wait for the method to finish. Once the analysis finishes, two new files will be created and automatically added to the list of files in the Results Files section. One file contains the output of states for each TA site in the genome. The other file contains the analysis for each gene. We can display each of the files by selecting them (individually) and clicking on the “Display Table” button (one at a time). Like for the Gumbel method, a break down of the states is provided at the top of the table. In the case of glycerol, the HMM analysis classifies 16.3% of the genome as belonging to the “Essential” state, 5.4% belonging to the Growth-Defect state, 77.1% to the Non-Essential state, and 1.2% to the Growth Advantage state. This break down can be used as a diagnostic, to see if the results match our expectations. For example, in datasets with very low read-counts, or very low density, the percentage of Growth-Defect states may be higher (e.g. > 10%), which could indicate a problem.

The HMM sites file contains the state assignments for all the TA sites in the genome. This file is particularly useful to browse for browsing the different types of regions in the genome. We can use this file to see how regions have different impacts on the growth-advantage (or disadvantage) of the organism. For example, the PDIM locus, which is required for virulence *in vivo*, results in a Growth-Advantage for the organism when disrupted. We can see this in the HMM Sites file by scrolling down to this region (Rv2930-Rv2939) and noticing the large read-counts at these sites, and how they are labeled “GA”.

2.9 Tutorial: Comparative Analysis - Glycerol vs Cholesterol

To illustrate how TRANSIT works, we are going to go through a tutorial where we analyze datasets of H37Rv *M. tuberculosis* grown on glycerol and cholesterol.

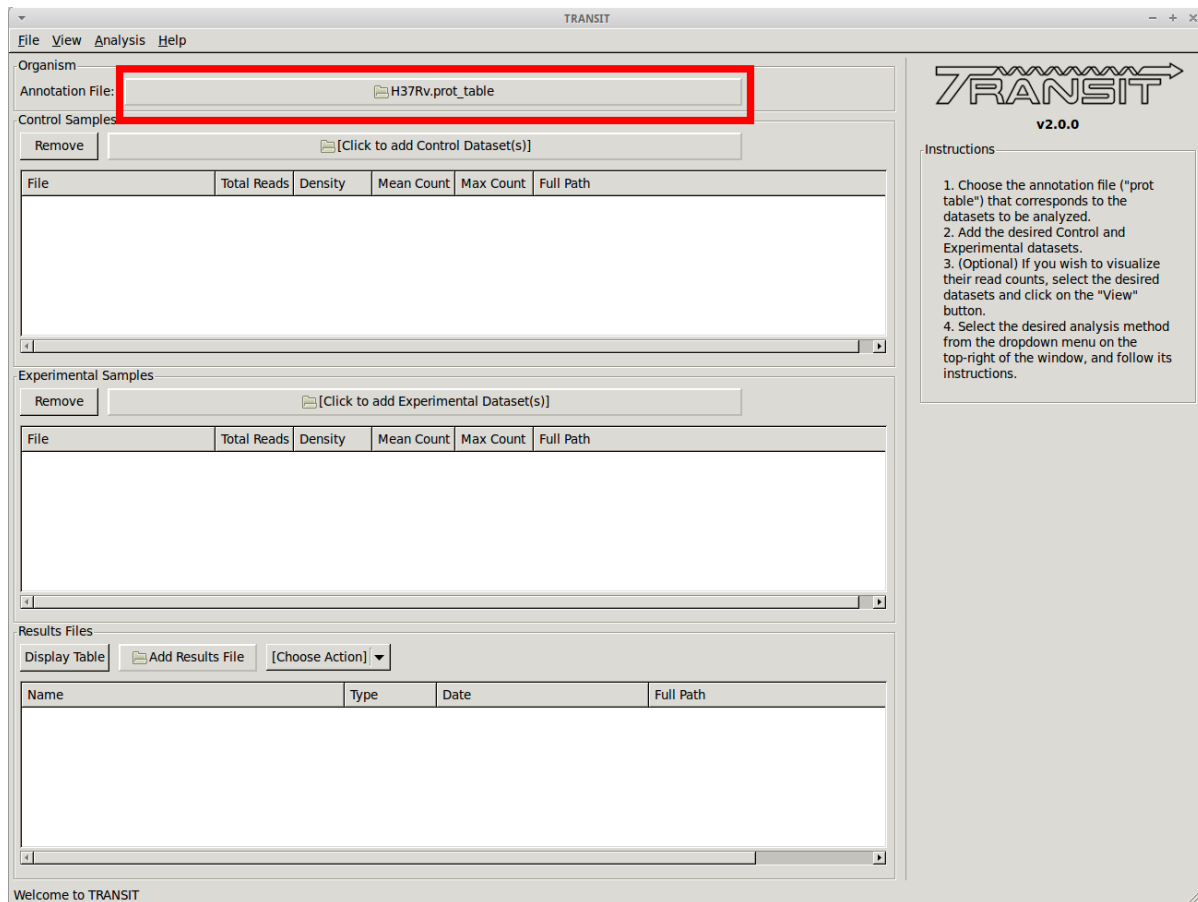
2.9.1 Run TRANSIT

Navigate to the directory containing the TRANSIT files, and run TRANSIT:

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

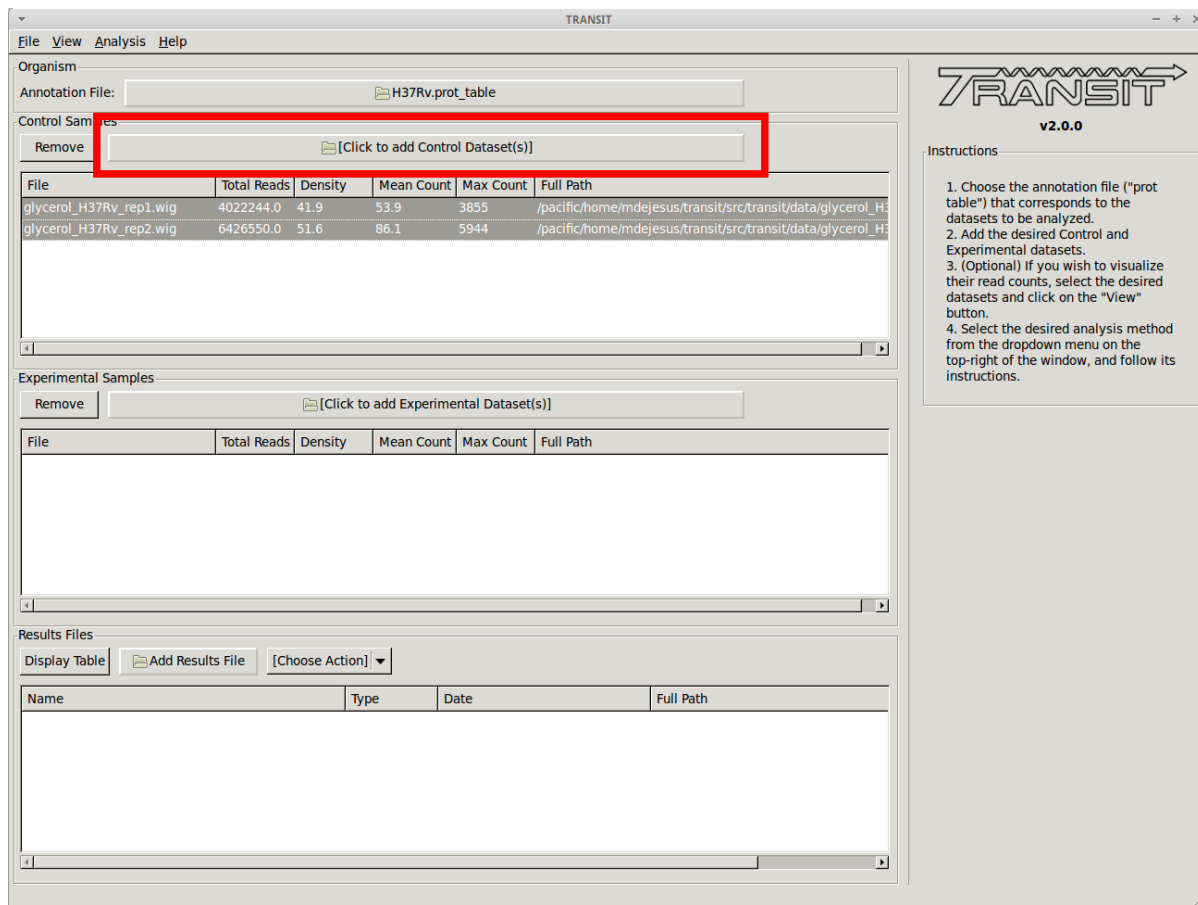
2.9.2 Adding the annotation file

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” format, described above.



2.9.3 Adding the control datasets

We want to analyze datasets grown in glycerol to those grown in cholesterol. We are choosing the datasets grown in glycerol as the “Control” datasets. 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.

2.9.4 Visualizing read counts

TRANSIT allows us to visualize the read-counts of the datasets we have already loaded. To do this, we must select the desired datasets ("Control+Click") and then click on "View -> Track View" in the menu bar at the top of the TRANSIT window. Only those selected datasets will be displayed:

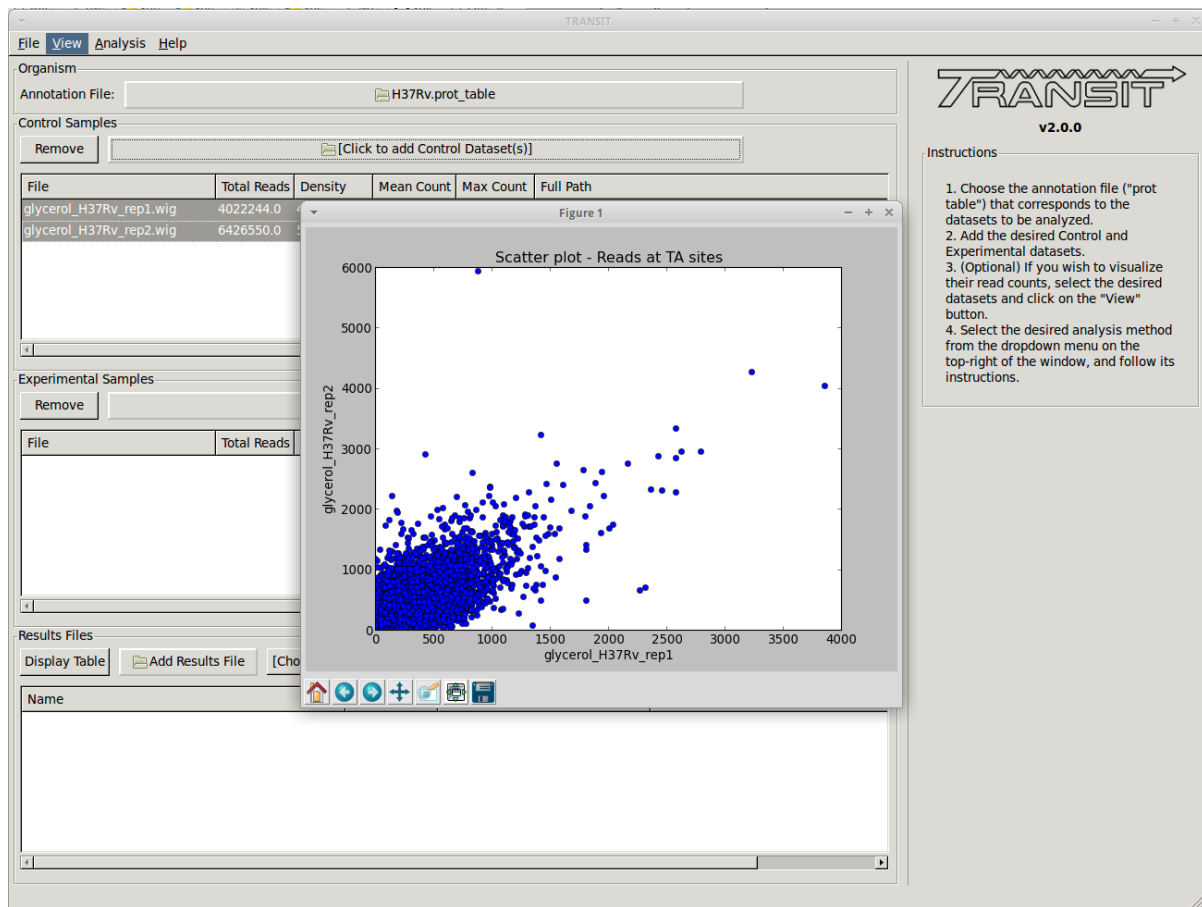


This will open a window that allows that shows a visual representation of the read counts at the TA sites throughout the genome. The scale of the read counts can be set by changing the value of the “Max Read” textbox on the right. We can browse around the genome by clicking on the left and right arrow, or search for a specific gene with the search text box.

This window also allows us to save a .png image of the canvas for future reference if desired (i.e. Save Img button).

2.9.5 Scatter plot

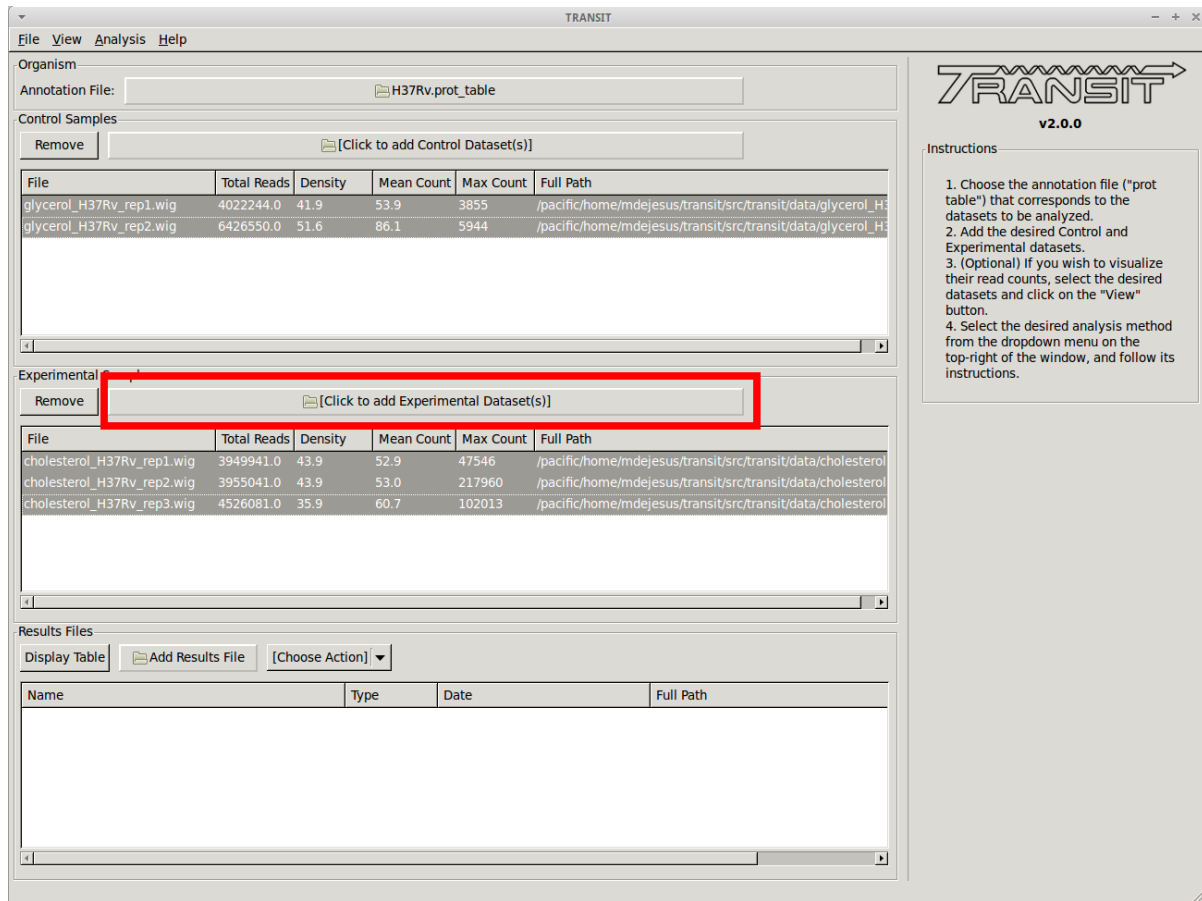
We can also view a scatter plot of read counts of two selected datasets. To achieve this we select two datasets (using “Control + Click”) and then clicking on “View -> Scatter Plot” in the menu bar at the top of the TRANSIT window.



A new window will pop-up, show a scatter plot of both of the selected datasets. This window contains controls to zoom in and out (magnifying glass), allowing us to focus in on a specific area. This is particularly useful when large outliers may throw off the scale of the scatter plot.

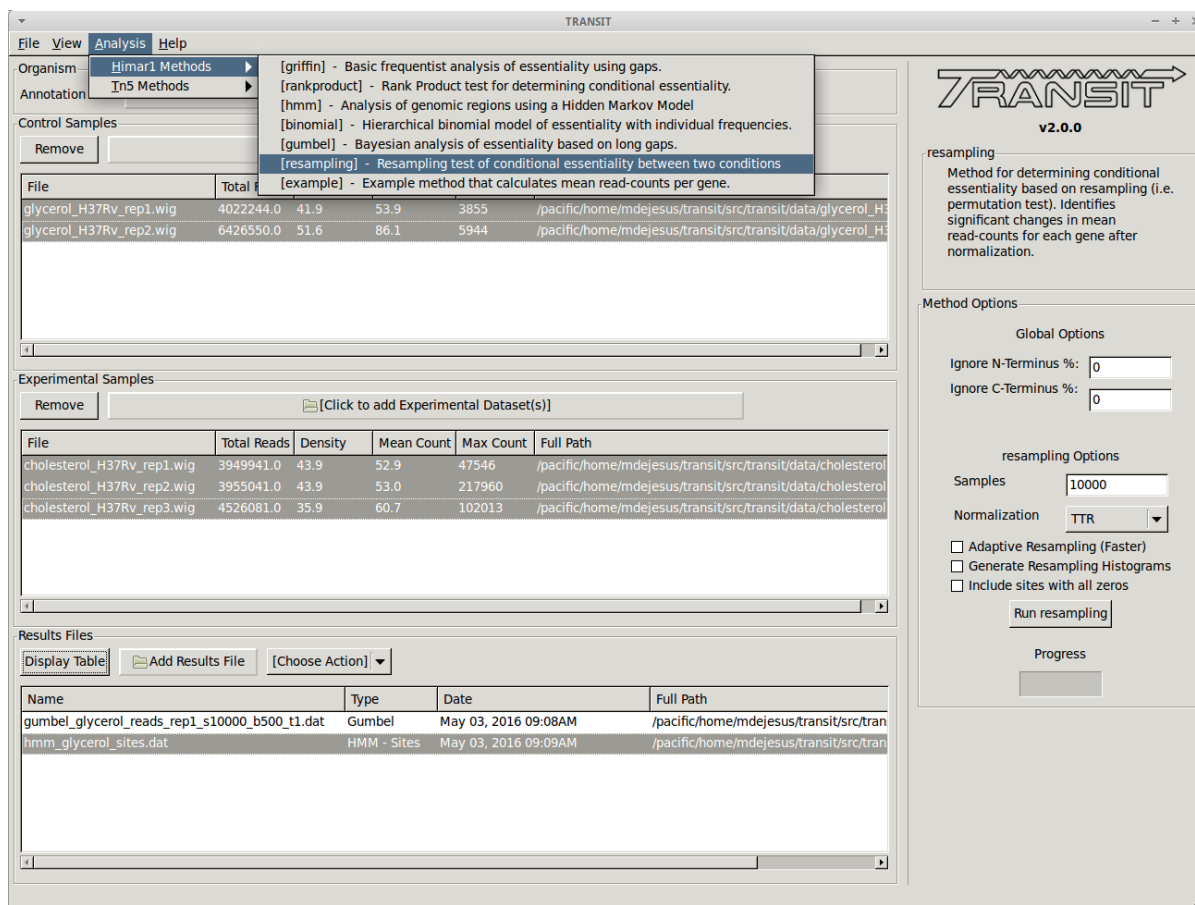
Adding the experimental datasets

We now repeat the process we did for control samples, for the experimental datasets that were grown on cholesterol. 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 three replicates:



Comparative analysis using Re-sampling

To compare the growth conditions and assess conditional essentiality, we select “Resampling” from the list of methods in the drop-down menu on the right side of the TRANSIT window:



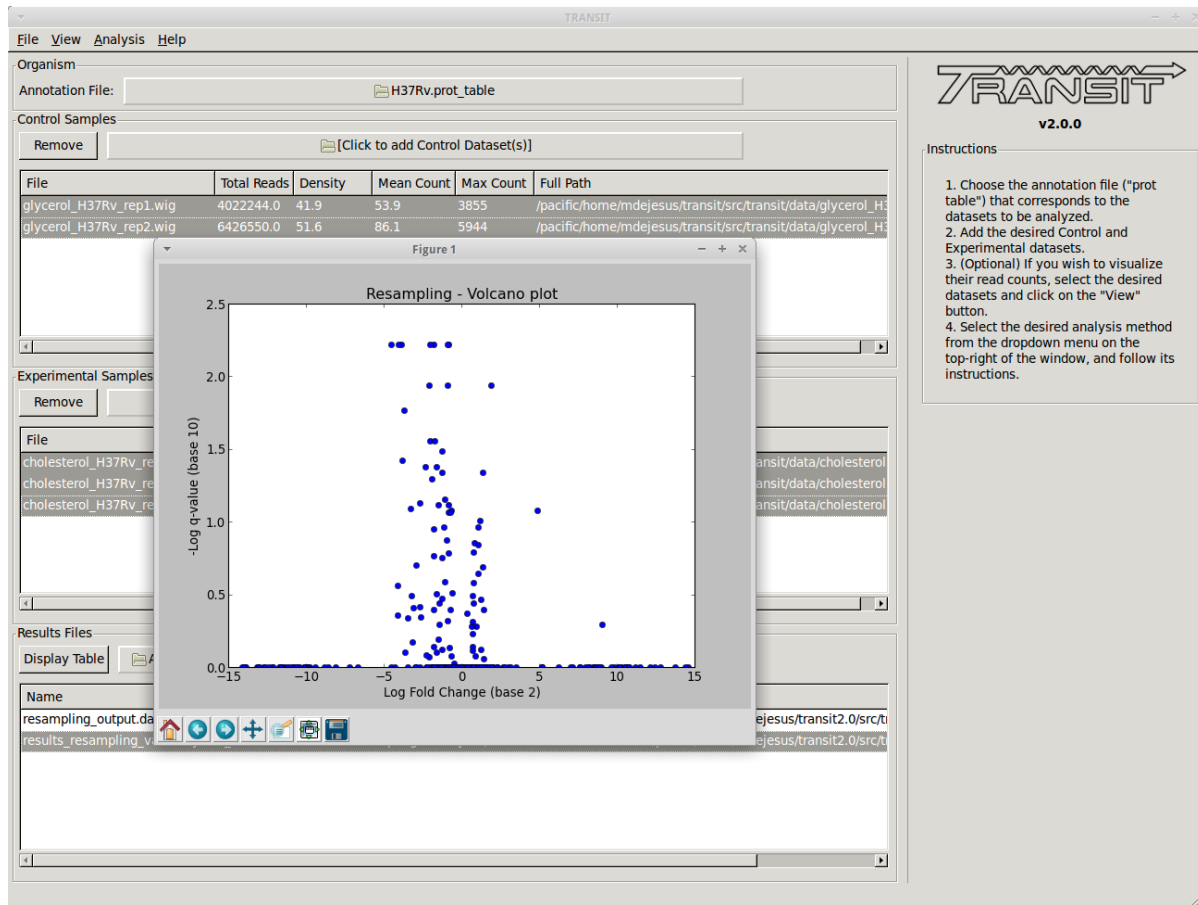
This will populate the right side with options specific to the Resampling method. In this case, we choose to proceed with the default settings. However, we could have set a different number of samples for the resampling method or chosen the “Adaptive Resampling” option if we were interested in quicker results. See the description of the method above for more information.

We click on the “Run Resampling” button to start the analysis. This will take several minutes to finish. The progress bar will give us an idea of how much time is left.

Viewing resampling results

Once TRANSIT finishes running, the results file will automatically be added to the Results Files section at the bottom of the window/

This window allows us to track the results files that have been created in this session. From here, we can display a volcano plot of the resampling results by selecting the file from the list and selecting the volcano option on the dropdown menu. This will open a new window containing the figure:



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 conditionally essential between glycerol and cholesterol, we can click on the column header labeled “q-value”, which represents p-values that have been adjusted for multiple comparisons. Sorting q-values in ascending order, we can see those genes which are most likely to be conditionally essential on the top. A typical threshold for significance is < 0.05 . We can use “Delta Sum” column to see which conditions had the most read counts in a particular gene. The sign of this value (+/-) lets us know on which condition the gene is essential and which condition it is non-essential. The magnitude lets us know how large the difference is. For example, glycerol kinase (GlpK) is necessary for growth on glycerol but it is not expected to be necessary when grown on another carbon source like cholesterol. We confirm our expectations by noticing that the sum of read counts in glycerol is only 22 reads (normalized), while there are a total of 2119 reads in cholesterol. The difference (2096) is positive, which means it is necessary for growth in glycerol but not cholesterol. Because we ran the resampling method with the “Histograms” options, we also have the ability to view the histograms of permutation differences for each of the genes by selecting a gene and right clicking:

From this menu we can display the histogram, or view the read-counts for that specific gene in Track View:

TRANSIT

File View Analysis Help

Organism

Information

Control Sam

Results:

Conditionally - Essentials: 28

More Essential in Experimental datasets: 8

Less Essential in Experimental datasets: 20

Remove

File

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

glycerol_H3

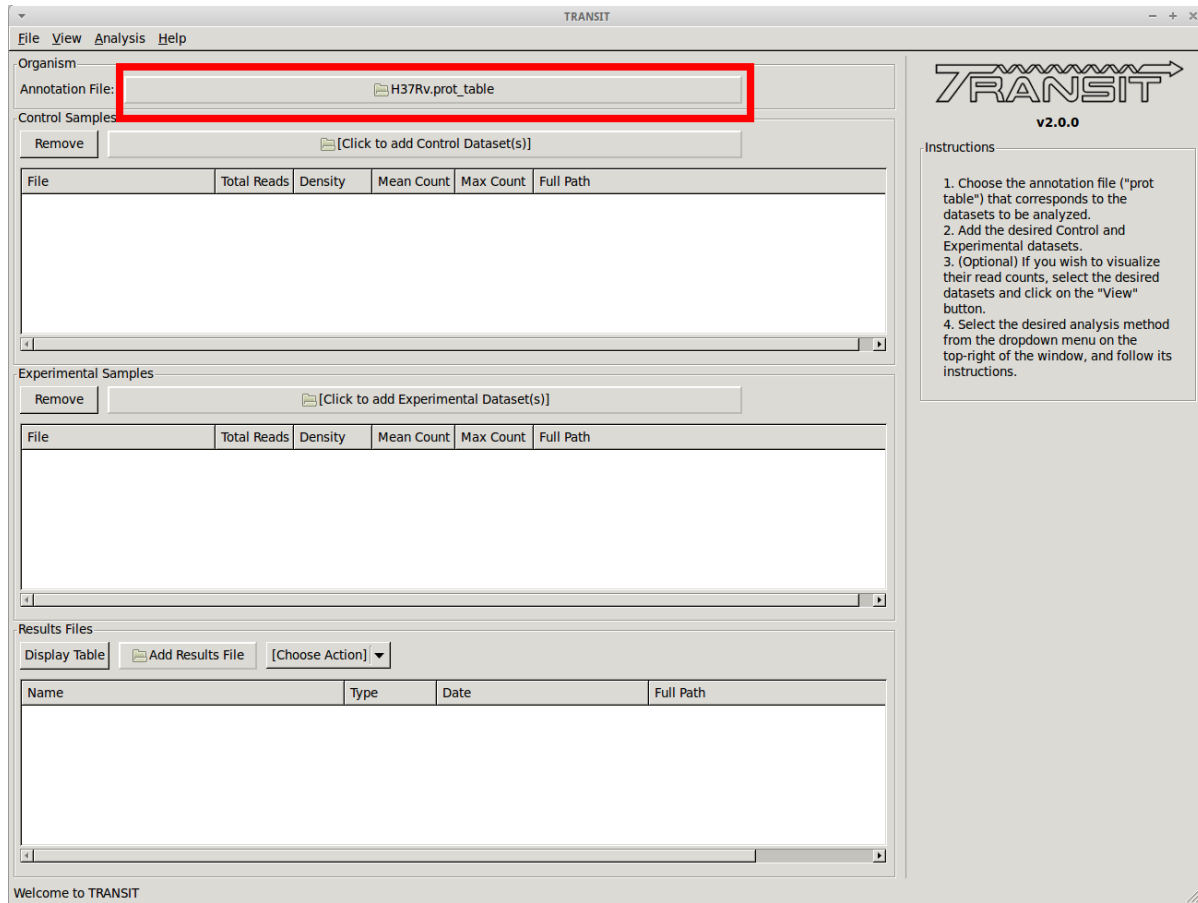
glycerol_H3

2.10 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.

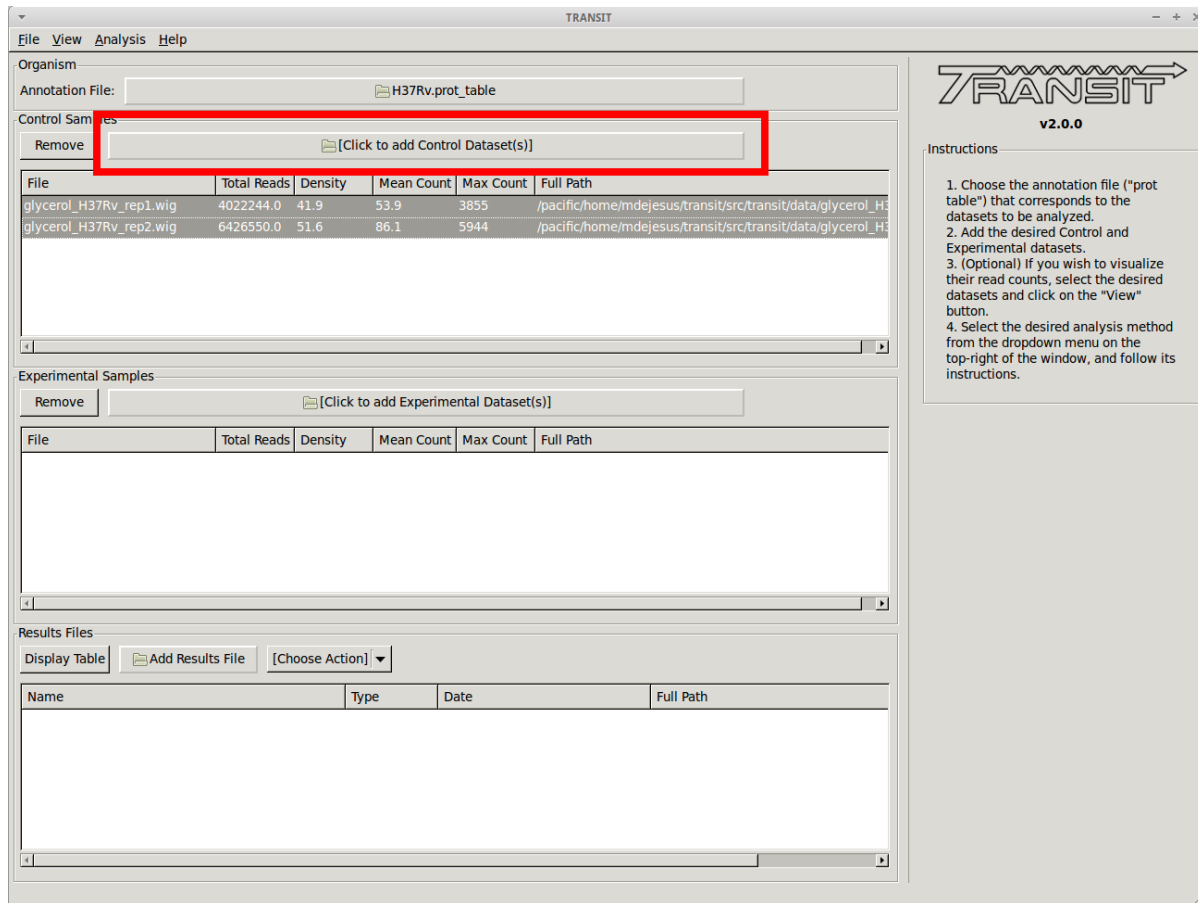
2.10.1 Adding the annotation file

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:



2.10.2 Add .wig datasets

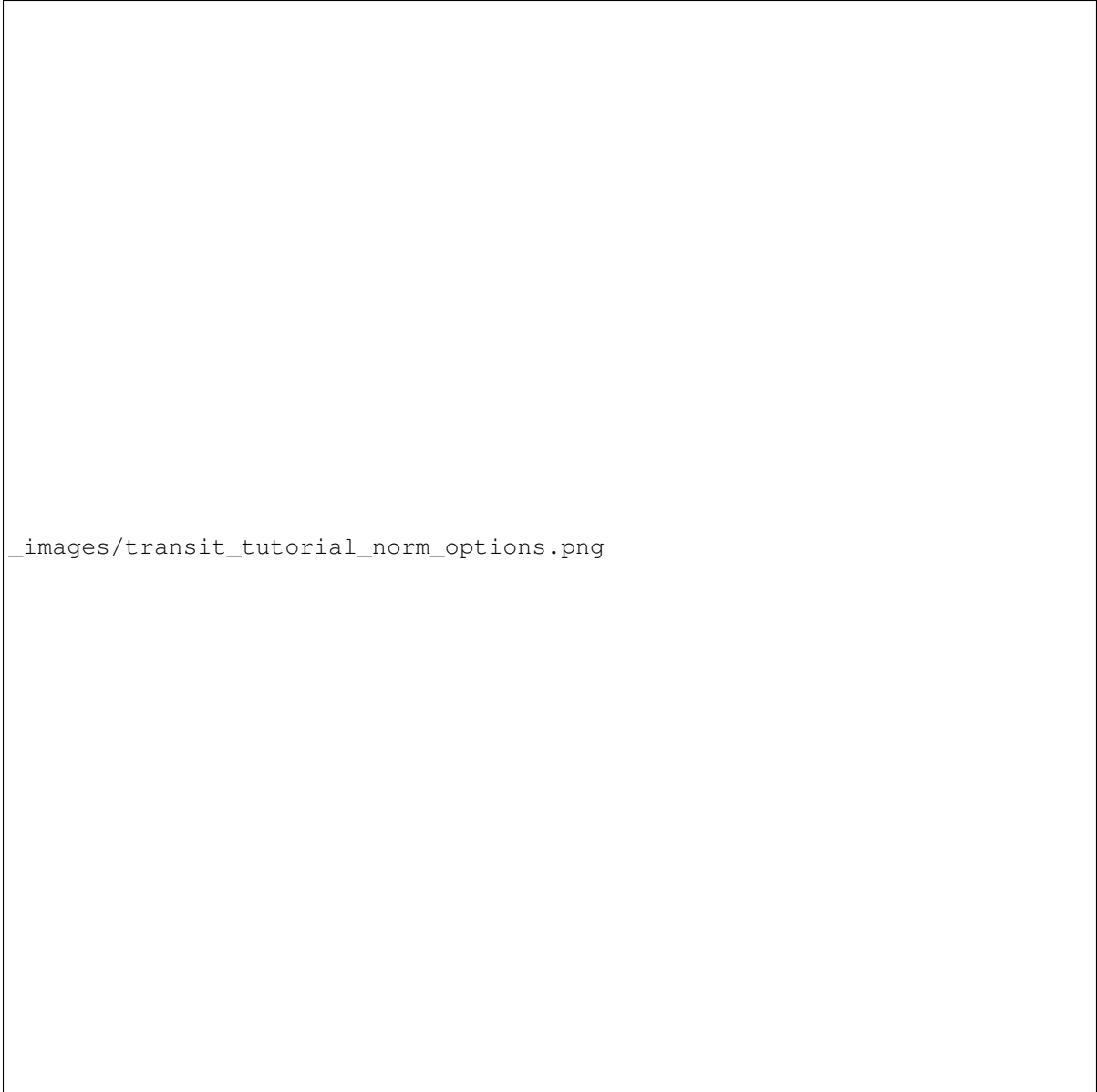
Next we must choose to add .wig formatted datasets what we wish to normalize to CombinedWig format. To add these, we click on the control sample file dialog (see image below), and select the desired datasets (one by one). In this example, we have two replicates:



As we add the datasets they will appear in the table below. Select the datasets you wish to normalize.

2.10.3 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.



`_images/transit_tutorial_norm_options.png`

2.10.4 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.

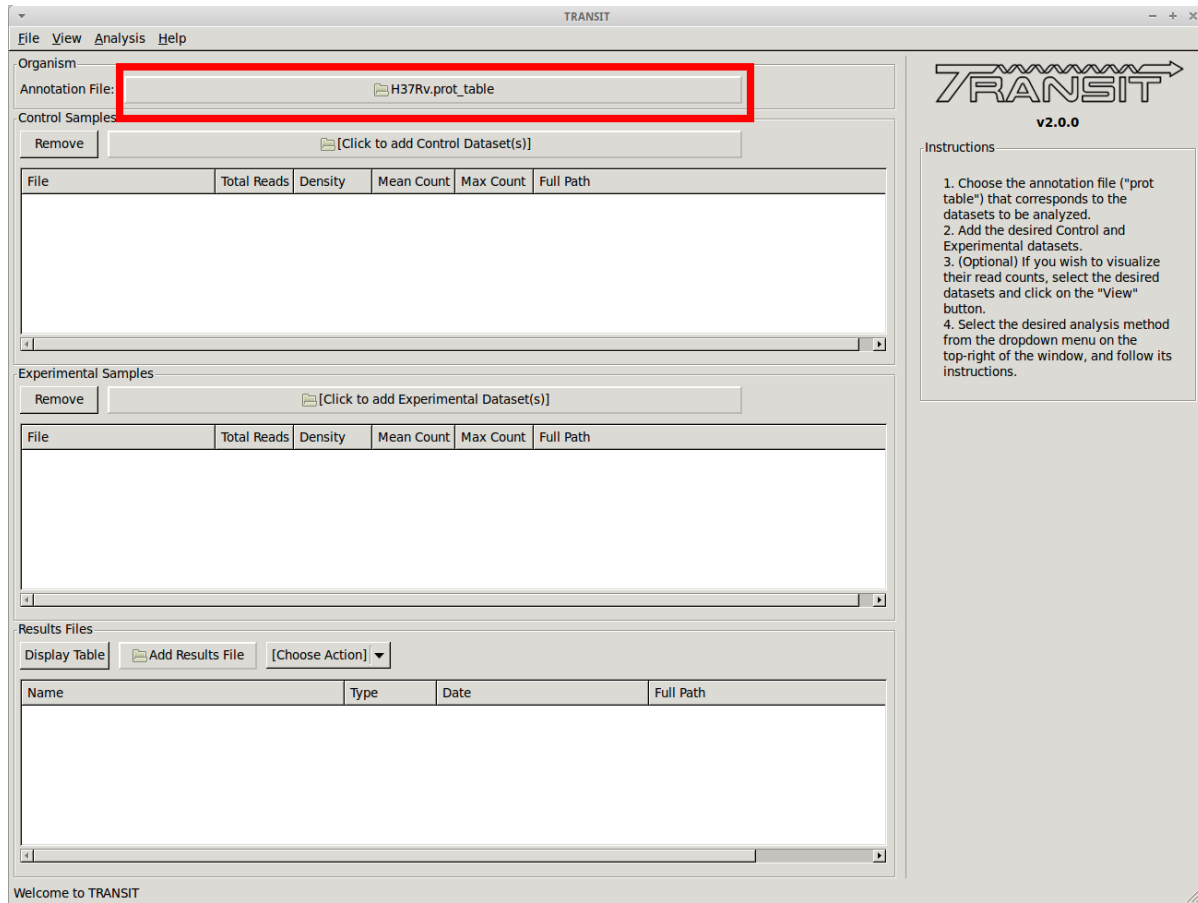
- **zinfnb:** Fits a zero-inflated negative binomial model, and then divides read-counts by the mean. The zero-inflated negative binomial model will treat some empty sites as belonging to the “true” negative binomial distribution responsible for read-counts while treating the others as “essential” (and 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.

2.11 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.

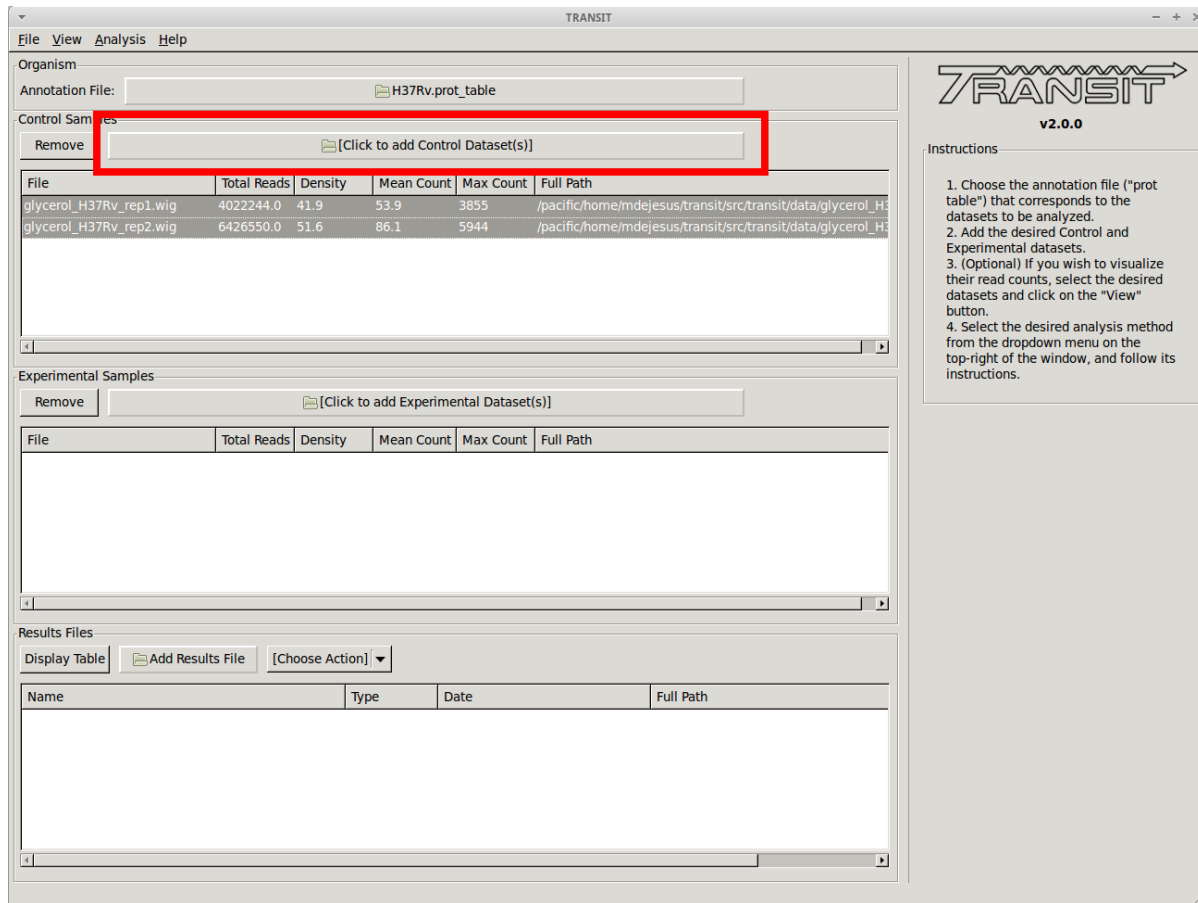
2.11.1 Adding the annotation file

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:



2.11.2 Add .wig datasets

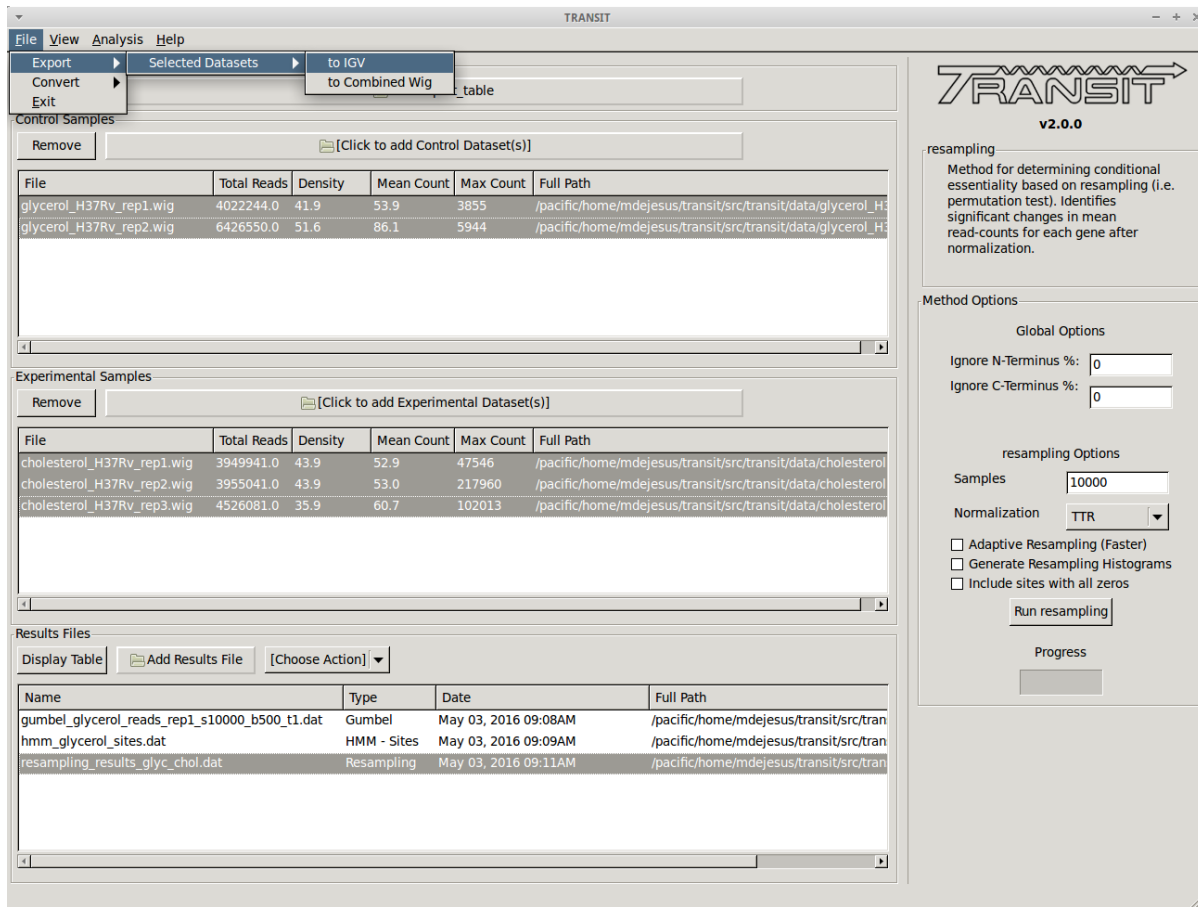
Next we must choose to add .wig formatted datasets what we wish to export to IGV format. To add these, we click on the control sample file dialog (see image below), and select the desired datasets (one by one). In this example, we have two replicates:



As we add the datasets they will appear in the table below.

2.11.3 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.



2.12 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 requires 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.

2.13 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 has the following requirements. If these are not already on your system, you will have to install them manually.

Requirements:

- python version 2.7
- wxPython 3.0.1 (the ‘cocoa’ version)
- 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’.

2.14 Running TPP

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

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

where PATH is the path to the TRANSIT installation directory. This should pop up the GUI window, looking like this...

Dataset (*.tn_stats)	total reads	TGTTA prefix	R1_mapped	R2_mapped	mapped reads	temp count
[08/02/16] temp3	25000	20353	16675	0	16675	16
[08/01/16] temp2	17245500	246399	197653	0	197653	178
[08/01/16] temp1	1000000	14243	11440	0	11440	10
[08/01/16] temp	3577277	3217784	203969	153376	136845	102
[05/25/15] Tn-BCG-5_2836	4429738	4327995	3921474	3381154	3098376	1707
[05/25/15] BCGcov	3893420	3411394	2480359	3238180	2374012	1853
[05/22/15] temp_salm_1M	1000000	514125	57353	0	57353	57

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...

- bwa executable - you'll have to find the path to where the executable is installed
- reference genome - this is the sequence in Fasta format against which the reads will be mapped
- 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 to use for output filename (for the multiple intermediate files that will get generated in the process; when you pick datasets, a temp file name will automatically be suggested for you, but you can change it to whatever you want)
- 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.
- 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

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 are 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.

```
For a list of possible command line arguments, type: python tpp.py -help
usage: python TRANSIT_PATH/src/tpp.py -bwa PATH_TO_EXECUTABLE -ref REF_SEQ -reads1_
↳PATH_TO_FASTQ_OR_FASTA_FILE [-reads2 PATH_TO_FASTQ_OR_FASTA_FILE] -output OUTPUT_
↳BASE_FILENAME [-maxreads N] [-tn5|-himar1] [-primer <seq>]
```

The input arguments and file types are as follows:

Flag	Value	Comments
-bwa	path executable	
-ref	reference genome sequence	FASTA file
-reads1	file of read 1 of paired reads	FASTA or FASTQ format (or gzipped)
-reads2	file of read 2 of paired reads (optional for single-end reads)	FASTA or FASTQ format (or gzipped)
-output†	base filename to use for output files	
-maxreads	subset of reads to process (optional); if blank, use	
-mismatches	how many to allow when searching reads for sequence patterns	
-tn5	process reads as a Tn5 library (Himar1 is assumed by default)	Reads mapping to any site will be considered.
-himar1	process reads as a Himar1 library (assumed by default)	Considers reads that map to TA sites only.
-primer	nucleotide sequence	Constant prefix of reads that TPP searches for.

† In earlier versions of Transit, this flag used to be ‘-prefix’, but we changed it to ‘-output’

(Note: if you have already run TPP once, then 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.)

2.15 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_API_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 GATGGCCGGTGGATTTGTGnnnnnnnnnnTGGTCGTG-GTAT”. 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, TGGTCGTGGTATxxxxxxTAACAGGTTGGCTGATAAG. 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”).
9. Output template counts at each TA site in a .wig file.
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.

2.16 Statistics

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.

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

Here is an example of a .tn_stats file:

```
# 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
```

(continues on next page)

(continued from previous page)

```

# read1: TnSeq_H37Rv_CB_1M_R1.fastq
# read2: TnSeq_H37Rv_CB_1M_R2.fastq
# ref_genome: H37Rv.fna
# total_reads 1000000 (or read pairs)
# TGTTA_reads 977626 (reads with valid Tn prefix, and insert size>20bp)
# reads1_mapped 943233
# reads2_mapped 892527
# mapped_reads 885796 (both R1 and R2 map into genome)
# read_count 879663 (TA sites only, for Himar1)
# template_count 605660
# template_ratio 1.45 (reads per template)
# TA_sites 74605
# TAs_hit 50382
# density 0.675
# max_count 356 (among templates)
# max_site 2631639 (coordinate)
# NZ_mean 12.0 (among templates)
# FR_corr 0.821 (Fwd templates vs. Rev templates)
# BC_corr 0.990 (reads vs. templates, summed over both strands)
# primer_matches: 10190 reads (1.0%) contain CTAGAGGGCCCAATTCGCCCTATAGTGAGT (Himar1)
# vector_matches: 5608 reads (0.6%) contain CTAGACCGTCCAGTCTGGCAGGCCGGAAC
→(phiMycoMarT7)
# adapter_matches: 0 reads (0.0%) contain GATCGGAAGAGCACACGTCTGAACTCCAGTCAC (Illumina/
→TruSeq index)
# misprimed_reads: 6390 reads (0.6%) contain Himar1 prefix but don't end in TGTTA
# read_length: 125 bp
# mean_R1_genomic_length: 92.9 bp
# mean_R2_genomic_length: 79.1 bp
TnSeq_H37Rv_CB_1M_R1.fastq  TnSeq_H37Rv_CB_1M_R2.fastq  1000000  977626  943233
→892527  885796  879663  605660  1.45240398904  74605  50382  356  0.675316667784
→2631639  12.0213568338  0.8209081083  0.989912222642  10190  5608  0  6390

```

Interpretation: To assess the quality of a dataset, I would recommend starting by looking at 3 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 at ~35% of TA sites for 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

2.17 transit package

2.17.1 Submodules

2.17.2 pytransit.norm_tools module

class pytransit.norm_tools.**AdaptiveBGCNorm**

Bases: *pytransit.norm_tools.NormMethod*

cleaninfgeom (*rho*)

Returns a ‘clean’ output from the geometric distribution.

ecdf (*x*)

Calculates an empirical CDF of the given data.

name = 'aBGC'

static normalize (*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

```
>>> 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*

cleaninfgeom (*rho*)

Returns a ‘clean’ output from the geometric distribution.

ecdf (*x*)

Calculates an empirical CDF of the given data.

```
name = 'betageom'
```

```
static normalize(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

```
>>> 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(args)
```

Objective function for the zero-inflated NB method.

```
name = 'emphist'
```

```
static normalize(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

```
>>> 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)

(continued from previous 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 (*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 (*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 (*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.

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*) –

14. 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(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

```
>>> import pytransit.norm_tools as norm_tools  
>>> import pytransit.tnseq_tools as tnseq_tools  
>>> (data, position) = tnseq_tools.get_data(["transit/data/glycerol_  
↪H37Rv_repl.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*) –
14. 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

```
>>> import pytransit.tnseq_tools as tnseq_tools  
>>> import pytransit.norm_tools as norm_tools  
>>> (data, position) = tnseq_tools.get_data(["transit/data/glycerol_  
↪H37Rv_repl.wig", "transit/data/glycerol_H37Rv_rep2.wig"])  
>>> print data
```

(continues on next page)

(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.TotReadsNormBases: *pytransit.norm_tools.NormMethod***name** = 'totreads'**static normalize** (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_
↳ 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.ZeroInflatedNBNormBases: *pytransit.norm_tools.NormMethod***name** = 'zinfb'**static normalize** (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 zinfnb 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.]])
>>> factors = norm_tools.zinfnb_factors(data)
>>> print factors
[[ 0.0121883 ]
 [ 0.00747111]]
```

See also:

`normalize_data`

`pytransit.norm_tools.cleaninfggeom(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*) –

14. *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`

`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

```
>>> 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=[], annotation-Path=')`

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

```
>>> 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") #_
↳Some methods require annotation and path to wig files.
>>> print normfactors
array([[ 1.         ],
       [ 0.62862886]])
>> print normdata
array([[ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.]])
```

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*) –
 14. *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

```
>>> 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.]])
>>> mu = norm_tools.trimmed_empirical_mu(data)
>>> print mu
120.73077107
```

See also:

TTR_factors

pytransit.norm_tools.zinfnb_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 zinfnb 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.]])
>>> factors = norm_tools.zinfnb_factors(data)
>>> print factors
[[ 0.0121883 ]
 [ 0.00747111]]
```

See also:

normalize_data

2.17.3 pytransit.stat_tools module

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

`pytransit.stat_tools.F_mean_diff_flat(A, B)`

`pytransit.stat_tools.F_shuffle_flat(X)`

`pytransit.stat_tools.F_sum_diff_flat(A, B)`

`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.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.fact(n)`

`pytransit.stat_tools.isEven(x)`

`pytransit.stat_tools.loess(X, Y, h=10000)`

`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.phi_coefficient(X, Y)`
Calculates the phi-coefficient for two bool arrays

`pytransit.stat_tools.qberndiff(d, peq, p01, p10)`

`pytransit.stat_tools.qbinomdiff(d, n, peq, p01, p10)`

```
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)
```

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

- **data1** – 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

```
>>> import pytransit.stat_tools as stat_tools
>>> 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.transformToRange(X, new_min, new_max, old_min=None,  
                                       old_max=None)
```

```
pytransit.stat_tools.tricoeff(N, S)
```

```
pytransit.stat_tools.tricube(X)
```

2.17.4 pytransit.tnseq_tools module

`pytransit.tnseq_tools.ExpectedRuns` (*n*, *pnon*)

Expected value of the run of non=insertions (Schilling, 1990):

$$ER_n = \log(1/p)(nq) + \gamma/\ln(1/p) - 1/2 + r_1(n) + E_1(n)$$

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

```
>>> 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:

Genes

__eq__ (*other*)

Compares against other gene object.

Returns True if the gene objects have same orf id.

Return type bool

__ge__ (*other*)

$x._\text{ge}_\text{(y)} \iff x \geq y$

__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*)

$x._\text{gt}_\text{(y)} \iff x > y$

__le__ (*other*)

$x._\text{le}_\text{(y)} \iff x \leq y$

__lt__ (*other*)

Compares against other gene object.

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

Return type bool

__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

get_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 pytransit.tnseq_tools.Genes(wigList, annotation, norm='nonorm', reps='All', minread=1, ignoreCodon=True, nterm=0.0, cterm=0.0, include_nc=False, data=[], position=[], genome="", transposon='himar1')
```

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

```
>>> 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
```

(continues on next page)

(continued from previous page)

```
>>> 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. 63. 0. 0.
  13. 0. 1. 0. 0.
  46. 0. 1. 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. 35.20321637 0.
  0. 0. 30.80281433 0. 101.20924707
  23.25926796 0. 16.97297932 8.17217523
  0. 0. 2.51451546 3.77177318 0.62862886
  0. 0. 69.14917502 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 ndarray

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 ndarray

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 ndarray

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 ndarray

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 ndarray

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 ndarray

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 ndarray

local_thetas()

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

Returns Numpy array with the density for all genes.

Return type ndarray

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):

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 ~ 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

```
>>> 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 to 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 to 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 to 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

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

`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

`pytransit.tnseq_tools.read_genome(path)`

Reads in FASTA formatted genome file.

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

Returns String with the genomic sequence.

Return type string

`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 tun.

Return type list

`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

2.17.5 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)`

`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.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

`pytransit.transit_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 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_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

```
>>> 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`

`pytransit.transit_tools.parseCoords` (*strand*, *aa_start*, *aa_end*, *start*, *end*)

`pytransit.transit_tools.transit_error` (*text*)

`pytransit.transit_tools.transit_message` (*msg*=",", *prefix*="")

`pytransit.transit_tools.validate_annotation` (*annotation*)

`pytransit.transit_tools.validate_both_datasets` (*ctrldata*, *expdata*)

`pytransit.transit_tools.validate_control_datasets` (*ctrldata*)

`pytransit.transit_tools.validate_filetypes` (*datasets*, *transposons*, *justWarn=True*)

`pytransit.transit_tools.validate_transposons_used` (*datasets*, *transposons*, *justWarn=True*)

`pytransit.transit_tools.validate_wig_format` (*wig_list*, *wxobj=None*)

2.17.6 Module contents

- `genindex`
- `modindex`
- `search`

Bibliography

- [DeJesus2015TRANSIT] DeJesus, M.A., Ambadipudi, C., Baker, R., Sassetti, C., and Ioerger, T.R. (2015). TRANSIT - a Software Tool for HimarI TnSeq Analysis. *PLOS Computational Biology*, 11(10):e1004401
- [DeJesus2013] DeJesus, M.A., Zhang, Y.J., Sassetti, C.M., Rubin, E.J., Sacchettini, J.C., and Ioerger, T.R. (2013). Bayesian analysis of gene essentiality based on sequencing of transposon insertion libraries. *Bioinformatics*, 29(6):695-703.
- [DeJesus2013HMM] DeJesus, M.A., Ioerger, T.R. A Hidden Markov Model for identifying essential and growth-defect regions in bacterial genomes from transposon insertion sequencing data. *BMC Bioinformatics*. 2013. 14:303
- [DeJesus2014] DeJesus, M.A. and Ioerger, T.R. (2014). Capturing uncertainty by modeling local transposon insertion frequencies improves discrimination of essential genes. *IEEE Transactions on Computational Biology and Bioinformatics*, 12(1):92-102.
- [DeJesus2016] DeJesus, M.A. and Ioerger, T.R. (2016). Normalization of transposon-mutant library sequencing datasets to improve identification of conditionally essential genes. *Journal of Bioinformatics and Computational Biology*, 14(3):1642004

p

- `pytransit`, [93](#)
- `pytransit.norm_tools`, [70](#)
- `pytransit.stat_tools`, [79](#)
- `pytransit.tnseq_tools`, [81](#)
- `pytransit.transit_tools`, [91](#)

Symbols

__contains__() (pytransit.tnseq_tools.Genes method), 84
 __eq__() (pytransit.tnseq_tools.Gene method), 82
 __ge__() (pytransit.tnseq_tools.Gene method), 82
 __getitem__() (pytransit.tnseq_tools.Gene method), 82
 __getitem__() (pytransit.tnseq_tools.Genes method), 84
 __gt__() (pytransit.tnseq_tools.Gene method), 82
 __le__() (pytransit.tnseq_tools.Gene method), 82
 __len__() (pytransit.tnseq_tools.Genes method), 84
 __lt__() (pytransit.tnseq_tools.Gene method), 82
 __str__() (pytransit.tnseq_tools.Gene method), 82
 __str__() (pytransit.tnseq_tools.Genes method), 84

A

AdaptiveBGCNorm (class in pytransit.norm_tools), 70
 aton() (in module pytransit.transit_tools), 92

B

basename() (in module pytransit.transit_tools), 92
 bayesian_ess_thresholds() (in module pytransit.stat_tools), 79
 BetaGeomNorm (class in pytransit.norm_tools), 70
 BH_fdr_correction() (in module pytransit.stat_tools), 79
 binom() (in module pytransit.stat_tools), 79
 binom_cdf() (in module pytransit.stat_tools), 79
 binom_test() (in module pytransit.stat_tools), 79
 boxcoxTable() (in module pytransit.stat_tools), 79
 boxcoxtransform() (in module pytransit.stat_tools), 79

C

check_wig_includes_zeros() (in module pytransit.tnseq_tools), 87
 cleanargs() (in module pytransit.transit_tools), 92
 cleaninfgeom() (in module pytransit.norm_tools), 76
 cleaninfgeom() (pytransit.norm_tools.AdaptiveBGCNorm method), 70
 cleaninfgeom() (pytransit.norm_tools.BetaGeomNorm method), 70

comb() (in module pytransit.stat_tools), 79
 comb1() (in module pytransit.stat_tools), 79
 combine_replicates() (in module pytransit.tnseq_tools), 87
 convertToCombinedWig() (in module pytransit.transit_tools), 92
 cterm (pytransit.tnseq_tools.Genes attribute), 83
 cumulative_average() (in module pytransit.stat_tools), 79

D

dberndiff() (in module pytransit.stat_tools), 79
 dbinomdiff() (in module pytransit.stat_tools), 79
 desc (pytransit.tnseq_tools.Gene attribute), 81
 dirname() (in module pytransit.transit_tools), 92

E

ecdf() (in module pytransit.norm_tools), 76
 ecdf() (pytransit.norm_tools.AdaptiveBGCNorm method), 70
 ecdf() (pytransit.norm_tools.BetaGeomNorm method), 70
 EmpHistNorm (class in pytransit.norm_tools), 71
 empirical_theta() (in module pytransit.norm_tools), 76
 empirical_theta() (pytransit.norm_tools.TTRNorm method), 73
 end (pytransit.tnseq_tools.Gene attribute), 81
 ExpectedRuns() (in module pytransit.tnseq_tools), 81

F

F_mean_diff_flat() (in module pytransit.stat_tools), 79
 F_shuffle_flat() (in module pytransit.stat_tools), 79
 F_sum_diff_flat() (in module pytransit.stat_tools), 79
 fact() (in module pytransit.stat_tools), 79
 fetch_name() (in module pytransit.transit_tools), 92
 Fzinfnb() (in module pytransit.norm_tools), 72
 Fzinfnb() (pytransit.norm_tools.EmpHistNorm static method), 71

G

Gene (class in pytransit.tnseq_tools), 81

Genes (class in `pytransit.tnseq_tools`), 83
genes (`pytransit.tnseq_tools.Genes` attribute), 83
get_coordinate_map() (in module `pytransit.tnseq_tools`), 87
get_data() (in module `pytransit.tnseq_tools`), 87
get_data_stats() (in module `pytransit.tnseq_tools`), 88
get_data_w_genome() (in module `pytransit.tnseq_tools`), 88
get_data_zero_fill() (in module `pytransit.tnseq_tools`), 88
get_extended_pos_hash() (in module `pytransit.transit_tools`), 92
get_extended_pos_hash_gff() (in module `pytransit.tnseq_tools`), 88
get_extended_pos_hash_pt() (in module `pytransit.transit_tools`), 88
get_file_types() (in module `pytransit.tnseq_tools`), 88
get_gap_span() (`pytransit.tnseq_tools.Gene` method), 82
get_gene_info() (in module `pytransit.tnseq_tools`), 88
get_gene_info() (in module `pytransit.transit_tools`), 92
get_gene_info_gff() (in module `pytransit.tnseq_tools`), 88
get_gene_info_pt() (in module `pytransit.tnseq_tools`), 89
get_gene_span() (`pytransit.tnseq_tools.Gene` method), 82
get_genes_in_range() (in module `pytransit.tnseq_tools`), 89
get_pos_hash() (in module `pytransit.tnseq_tools`), 89
get_pos_hash() (in module `pytransit.transit_tools`), 92
get_pos_hash_gff() (in module `pytransit.tnseq_tools`), 89
get_pos_hash_pt() (in module `pytransit.tnseq_tools`), 89
get_unknown_file_types() (in module `pytransit.tnseq_tools`), 90
get_validated_data() (in module `pytransit.transit_tools`), 92
get_wig_stats() (in module `pytransit.tnseq_tools`), 90
getE1() (in module `pytransit.tnseq_tools`), 87
getE2() (in module `pytransit.tnseq_tools`), 87
getGamma() (in module `pytransit.tnseq_tools`), 87
getR1() (in module `pytransit.tnseq_tools`), 87
getR2() (in module `pytransit.tnseq_tools`), 87
getTabTableData() (in module `pytransit.transit_tools`), 92
global_insertion() (`pytransit.tnseq_tools.Genes` method), 84
global_phi() (`pytransit.tnseq_tools.Genes` method), 85
global_reads() (`pytransit.tnseq_tools.Genes` method), 85
global_run() (`pytransit.tnseq_tools.Genes` method), 85
global_sites() (`pytransit.tnseq_tools.Genes` method), 85
global_theta() (`pytransit.tnseq_tools.Genes` method), 85
griffin_analysis() (in module `pytransit.tnseq_tools`), 90
GumbelCDF() (in module `pytransit.tnseq_tools`), 86

I

ignoreCodon (`pytransit.tnseq_tools.Genes` attribute), 83
include_nc (`pytransit.tnseq_tools.Genes` attribute), 83
isEven() (in module `pytransit.stat_tools`), 79

L

local_gap_span() (`pytransit.tnseq_tools.Genes` method), 85
local_gene_span() (`pytransit.tnseq_tools.Genes` method), 85
local_insertions() (`pytransit.tnseq_tools.Genes` method), 85
local_phi() (`pytransit.tnseq_tools.Genes` method), 85
local_reads() (`pytransit.tnseq_tools.Genes` method), 86
local_runs() (`pytransit.tnseq_tools.Genes` method), 86
local_sites() (`pytransit.tnseq_tools.Genes` method), 86
local_thetas() (`pytransit.tnseq_tools.Genes` method), 86
loess() (in module `pytransit.stat_tools`), 79
loess_correction() (in module `pytransit.stat_tools`), 79
log_fac() (in module `pytransit.stat_tools`), 79
loglik() (in module `pytransit.stat_tools`), 79

M

maxrun() (in module `pytransit.tnseq_tools`), 91
minread (`pytransit.tnseq_tools.Genes` attribute), 83
multinomial() (in module `pytransit.stat_tools`), 79
my_perm() (in module `pytransit.stat_tools`), 79

N

name (`pytransit.norm_tools.AdaptiveBGCNorm` attribute), 70
name (`pytransit.norm_tools.BetaGeomNorm` attribute), 71
name (`pytransit.norm_tools.EmphHistNorm` attribute), 71
name (`pytransit.norm_tools.NoNorm` attribute), 72
name (`pytransit.norm_tools.NormMethod` attribute), 72
name (`pytransit.norm_tools.NZMeanNorm` attribute), 72
name (`pytransit.norm_tools.QuantileNorm` attribute), 73
name (`pytransit.norm_tools.TotReadsNorm` attribute), 75
name (`pytransit.norm_tools.TTRNorm` attribute), 73
name (`pytransit.norm_tools.ZeroInflatedNBNorm` attribute), 75
name (`pytransit.tnseq_tools.Gene` attribute), 81
NoNorm (class in `pytransit.norm_tools`), 72
norm (`pytransit.tnseq_tools.Genes` attribute), 83
norm() (in module `pytransit.stat_tools`), 79
norm_to_target() (in module `pytransit.norm_tools`), 76
normalize() (`pytransit.norm_tools.AdaptiveBGCNorm` static method), 70
normalize() (`pytransit.norm_tools.BetaGeomNorm` static method), 71
normalize() (`pytransit.norm_tools.EmphHistNorm` static method), 71
normalize() (`pytransit.norm_tools.NoNorm` static method), 72
normalize() (`pytransit.norm_tools.NormMethod` static method), 72
normalize() (`pytransit.norm_tools.NZMeanNorm` static method), 72

normalize() (pytransit.norm_tools.QuantileNorm static method), 73
 normalize() (pytransit.norm_tools.TotReadsNorm static method), 75
 normalize() (pytransit.norm_tools.TTRNorm static method), 74
 normalize() (pytransit.norm_tools.ZeroInflatedNBNorm static method), 75
 normalize_data() (in module pytransit.norm_tools), 77
 NormMethod (class in pytransit.norm_tools), 72
 nterm (pytransit.tnseq_tools.Genes attribute), 83
 NZMeanNorm (class in pytransit.norm_tools), 72

O

orf (pytransit.tnseq_tools.Gene attribute), 81
 orf2index (pytransit.tnseq_tools.Genes attribute), 83

P

parseCoords() (in module pytransit.transit_tools), 93
 phi() (pytransit.tnseq_tools.Gene method), 82
 phi_coefficient() (in module pytransit.stat_tools), 79
 position (pytransit.tnseq_tools.Gene attribute), 81
 protTable (pytransit.tnseq_tools.Genes attribute), 83
 pytransit (module), 93
 pytransit.norm_tools (module), 70
 pytransit.stat_tools (module), 79
 pytransit.tnseq_tools (module), 81
 pytransit.transit_tools (module), 91

Q

qberndiff() (in module pytransit.stat_tools), 79
 qbinomdiff() (in module pytransit.stat_tools), 79
 QuantileNorm (class in pytransit.norm_tools), 72

R

read_genome() (in module pytransit.tnseq_tools), 91
 reads (pytransit.tnseq_tools.Gene attribute), 81
 regress() (in module pytransit.stat_tools), 79
 reps (pytransit.tnseq_tools.Genes attribute), 83
 resampling() (in module pytransit.stat_tools), 80
 runindex() (in module pytransit.tnseq_tools), 91
 runs() (in module pytransit.tnseq_tools), 91
 runs_w_info() (in module pytransit.tnseq_tools), 91

S

ShowAskWarning() (in module pytransit.transit_tools), 91
 ShowError() (in module pytransit.transit_tools), 91
 ShowMessage() (in module pytransit.transit_tools), 91
 start (pytransit.tnseq_tools.Gene attribute), 81
 strand (pytransit.tnseq_tools.Gene attribute), 81

T

theta() (pytransit.tnseq_tools.Gene method), 82

tosses() (pytransit.tnseq_tools.Genes method), 86
 tossify() (in module pytransit.tnseq_tools), 91
 total_reads() (pytransit.tnseq_tools.Gene method), 83
 total_reads() (pytransit.tnseq_tools.Genes method), 86
 TotReadsNorm (class in pytransit.norm_tools), 75
 transformToRange() (in module pytransit.stat_tools), 80
 transit_error() (in module pytransit.transit_tools), 93
 transit_message() (in module pytransit.transit_tools), 93
 tricoeff() (in module pytransit.stat_tools), 80
 tricube() (in module pytransit.stat_tools), 80
 trimmed_empirical_mu() (in module pytransit.norm_tools), 77
 trimmed_empirical_mu() (pytransit.norm_tools.TTRNorm method), 74
 TTRNorm (class in pytransit.norm_tools), 73

V

validate_annotation() (in module pytransit.transit_tools), 93
 validate_both_datasets() (in module pytransit.transit_tools), 93
 validate_control_datasets() (in module pytransit.transit_tools), 93
 validate_filetypes() (in module pytransit.transit_tools), 93
 validate_transposons_used() (in module pytransit.transit_tools), 93
 validate_wig_format() (in module pytransit.transit_tools), 93
 VarR() (in module pytransit.tnseq_tools), 86

W

wigList (pytransit.tnseq_tools.Genes attribute), 83

Z

ZeroInflatedNBNorm (class in pytransit.norm_tools), 75
 zinfnb_factors() (in module pytransit.norm_tools), 78