1. Dependencies

- fastqc
- trinity v2.5.1
- BUSCO v3.0.1
- trinotate v3.1.1
- Transdecoder v5.0.2
- BLAST+ >= v2.2.1+
- hmmer >= v3.1b1
- signalP v4
- tmhmm v2
- R: tximport, readr, DESeq2, tidyr, dplyr, magrittr, ggplot2, cowplot, tibble

    for DESeq2 make sure you check the version via sessionInfo() if you are working on different computers

2. Quality Control (https://www.bioinformatics.babraham.ac.uk/projects/fastqc/)

The quality scores should stay high along the length of the read since they are relatively short. Many of the metrics will be pretty meaningless for RNAseq data. The overrepresented sequences should be mostly mitochondrial.

```
fastqc -o qc -t 12 *_R*_fastq.gz
```

3. De novo assembly with Trinity (https://github.com/trinityrnaseq/trinityrnaseq/wiki)

The following uses 4 GB of RAM (--max_memory 4G) and 4 cpus (--CPU 4), trims (--trimmomatic), normalizes (--normalize_reads), and records messages in a log file (&> log00.txt). Since there are potentially a lot of reads, Trinity first normalizes each set of reads (if it instead normalized all at once it might run out of RAM) before finally combining the remainder and normalizing a final time (--normalize_by_read_set).

% $TRINITY_HOME/Trinity --seqType fq --max_memory 4G --left $file1_1,$file2_1 --right $file1_2,$file2_2 --CPU 4 --trimmomatic --normalize_reads --normalize_by_read_set --output trinity &> log00.txt

[~/opt/trinityrnaseq-Trinity-v2.5.1/Trinity --seqType fq --max_memory 8G --left Control_1_R1.fq,Control_2_R1.fq,Control_3_R1.fq,Treated_1_R1.fq,Treated_2_R1.fq,Treated_3_R1.fq --right Control_1_R2.fq,Control_2_R2.fq,Control_3_R2.fq,Treated_1_R2.fq,Treated_2_R2.fq,Treated_3_R2.fq --CPU 4 --trimmomatic --normalize_reads --normalize_by_read_set --output trinity]

(Problems with libtbb-so-2 http://www.metagenomics.wiki/tools/bowtie2/install/libtbb-so-2)

4. Evaluating the quality of the assembly

There are several ways to quantitatively as well as qualitatively assess the overall quality of the assembly, and we outline many of these methods at our Trinity wiki.

a. Assembly Statistics that are NOT very useful

You can count the number of assembled transcripts by using 'grep' to retrieve only the FASTA header lines and piping that output into 'wc' (word count utility) with the '-l' parameter to just count the number of lines.

```
% grep '>' trinity_out_dir/Trinity.fasta | wc -l
```

It's useful to know how many transcript contigs were assembled, but it's not very informative. The deeper you sequence, the more transcript contigs you will be able to reconstruct. It's not unusual to assemble over a million transcript contigs with very deep data sets and complex transcriptomes, but as you 'll see below (in the section containing the more informative guide to assembly assessment) a fraction of the transcripts generally best represent the input RNA-Seq reads.

Examine assembly stats

Capture some basic statistics about the Trinity assembly:

```
% $TRINITY_HOME/util/TrinityStats.pl trinity_out_dir/Trinity.fasta
```

which should generate data like so. Note your numbers may vary slightly, as the assembly results are not deterministic.

The total number of reconstructed transcripts should match up identically to what we counted earlier with our simple 'grep | wc' command. The total number of 'genes' is also reported - and simply involves counting up the number of unique transcript identifier prefixes (without the _i isoform numbers). When the 'gene' and 'transcript' identifiers differ, it's due to transcripts being reported as alternative isoforms for the same gene. In our tiny example data set, we reconstruct only a small number of alternative isoforms, and note that alternative splicing in this yeast species may be fairly rare. Tackling an insect or mammal transcriptome would be expected to yield many alternative isoforms.

You'll also see 'Contig N50' values reported. The 'N50 statistic indicates that at least half of the assembled bases are in contigs of at least that contig length'.

Most of this is not quantitatively useful, and the values are only reported for historical reasons - it's simply what everyone used to do in the early days of transcriptome assembly. The N50 statistic in RNA-Seq assembly can be easily biased in the following ways:

•Overzealous reconstruction of long alternatively spliced isoforms: If an assembler tends to generate many different 'versions' of splicing for a gene, such as in a combinatorial way, and those isoforms tend to have long sequence lengths, the N50 value will be skewed towards a higher value.

•Highly sensitive reconstruction of lowly expressed isoforms: If an assembler is able to reconstruct transcript contigs for those transcirpts that are very lowly expressed, these contigs will tend to be short and numerous, biasing the N50 value towards lower values. As one sequences deeper, there will be more evidence (reads) available to enable reconstruction of these short lowly expressed transcripts, and so deeper sequencing can also provide a downward skew of the N50 value.

b. Representation of reads

A high quality transcriptome assembly is expected to have strong representation of the reads input to the assembler. By aligning the RNA-Seq reads back to the transcriptome assembly, we can quantify read representation. Use the Bowtie2 aligner to align the reads to the Trinity assembly, and in doing so, take notice of the read representation statistics reported by the bowtie2 aligner.

First build a bowtie2 index for the Trinity assembly, required before running the alignment:

```
%  bowtie2-build trinity_out_dir/Trinity.fasta trinity_out_dir/Trinity.fasta
```

Now, align the reads to the assembly:

```
%   bowtie2 --local --no-unal -x trinity_out_dir/Trinity.fasta -q -1 data/wt_SRR1582651_1.fastq -2 data/wt_SRR1582651_2.fastq | samtools view -Sb - | samtools sort -o - - > bowtie2.bam
```

```
[bowtie2 --local --no-unal -x trinity/Trinity.fasta -q -1 Control_1_R1.fq,Control_2_R1.fq,Control_3_R1.fq,Treated_1_R1.fq,Treated_2_R1.fq,Treated_3_R1.fq -2 Control_1_R2.fq,Control_2_R2.fq,Control_3_R2.fq,Treated_1_R2.fq,Treated_2_R2.fq,Treated_3_R2.fq |samtools view -Sb |samtools sort -o -- >bowtie2.bam]
```

Generally, in a high quality assembly, you would expect to see at least ~70% aligned and at least ~70% of the reads to exist as proper pairs. Our tiny read set used here in this workshop does not provide us with a high quality assembly, as only ~30% of aligned reads are mapped as proper pairs - which is usually the sign of a fractured assembly. In this case, deeper sequencing and assembly of more reads would be expected to lead to major improvements here.

c. Benchmarking Universal Single-Copy Orthologs (http://busco.ezlab.org/)

It's not particularly meaningful but will give an indication as to whether the two assemblies are wildly different in terms of completeness. Of course for an RNAseq assembly we don't necessarily expect all conserved single copy orthologs to be expressed in the sampled tissues/stages but this doesn't stop reviewers from asking for it.

Retrieve BUSCO:

% git clone https://gitlab.com/ezlab/busco

install BUSCO:

% python setup.py install –user

install dependents: NCBI BLAST+ & HMMER (HMMER v3.1b2)

edit configuration file:

% cp config.ini.default config.ini

run BUSCO

%python /path/to/BUSCO_v1.22/BUSCO_v1.22.py -o busco -in cm_Trinity.fasta -l\
/path/to/BUSCO_v1.22/arthropoda -m tran

5. Annotation with Trinotate (http://trinotate.github.io/)

a. Bioinformatics analyses to gather evidence for potential biological functions

Identification of likely protein-coding regions in transcripts

TransDecoder is a tool we built to identify likely coding regions within transcript sequences. It identifies long open reading frames (ORFs) within transcripts and scores them according to their sequence composition. Those ORFs that encode sequences with compositional properties (codon frequencies) consistent with coding transcripts are reported.

Running TransDecoder is a two-step process. First run the TransDecoder step that identifies all long ORFs.

% $TRANSDECODER_HOME/TransDecoder.LongOrfs -t ../trinity_out_dir/Trinity.fasta

Now, run the step that predicts which ORFs are likely to be coding.

% $TRANSDECODER_HOME/TransDecoder.Predict -t ../trinity_out_dir/Trinity.fasta

You'll now find a number of output files containing 'transdecoder' in their name:

% ls -1 |grep transdecoder

The file we care about the most here is the 'Trinity.fasta.transdecoder.pep' file, which contains the protein sequences corresponding to the predicted coding regions within the transcripts.

Go ahead and take a look at this file:

% less Trinity.fasta.transdecoder.pep

There are a few items to take notice of in the above peptide file. The header lines includes the protein identifier composed of the original transcripts along with '|m.(number)'. The 'type' attribute indicates whether the protein is 'complete', containing a start and a stop codon; '5prime_partial', meaning it's missing a start codon and presumably part of the N-terminus; '3prime_partial', meaning it's missing the stop codon and presumably part of the C-terminus; or 'internal', meaning it's both 5prime-partial and 3prime-partial. You'll also see an indicator (+) or (-) to indicate which strand the coding region is found on, along with the coordinates of the ORF in that transcript sequence.

This .pep file will be used for various sequence homology and other bioinformatics analyses below.

Sequence homology searches

Earlier, we ran blastx against our mini SWISSPROT datbase to identify likely full-length transcripts. Let's run blastx again to capture likely homolog information, and we'll lower our E-value threshold to 1e-5 to be less stringent than earlier.

```
% blastx -db ../data/mini_sprot.pep -query ../trinity_out_dir/Trinity.fasta -num_threads 2 –\
max_target_seqs 1 -outfmt 6 -evalue 1e-5  > swissprot.blastx.outfmt6
```

Now, let's look for sequence homologies by just searching our predicted protein sequences rather than using the entire transcript as a target:

```
% blastp -query Trinity.fasta.transdecoder.pep  -db ../data/mini_sprot.pep -num_threads 2 \

-max_target_seqs 1 -outfmt 6 -evalue 1e-5  > swissprot.blastp.outfmt6
```

Using our predicted protein sequences, let's also run a HMMER search against the Pfam database, and identify conserved domains that might be indicative or suggestive of function:

```
% hmmscan --cpu 2 --domtblout TrinotatePFAM.out  ../data/trinotate_data/Pfam-A.hmm \
Trinity.fasta.transdecoder.pep
```

Note, hmmscan might take a few minutes to run.

Computational prediction of sequence features

The signalP and tmhmm software tools are very useful for predicting signal peptides (secretion signals) and transmembrane domains, respectively.

To predict signal peptides, run signalP like so:

% signalp -f short -n signalp.out Trinity.fasta.transdecoder.pep

Running tmHMM to predict transmembrane regions:

% tmhmm --short < transdecoder.pep > tmhmm.out

b. Preparing and Generating a Trinotate Annotation Report

Generating a Trinotate annotation report involves first loading all of our bioinformatics computational results into a Trinotate SQLite database. The Trinotate software provides a boilerplate SQLite database called 'Trinotate.sqlite' that comes pre-populated with a lot of generic data about SWISSPROT records and Pfam domains (and is a pretty large file consuming several hundred MB). Below, we'll populate this database with all of our bioinformatics computes and our expression data.

Preparing Trinotate (loading the database)

Copy the provided Trinotate.sqlite boilerplate database into your Trinotate working directory.

Load your Trinotate.sqlite database with your Trinity transcripts and predicted protein sequences:

% $TRINOTATE_HOME/Trinotate Trinotate.sqlite init --\
gene_trans_map ../trinity_out_dir/Trinity.fasta.gene_trans_map \

--transcript_fasta ../trinity_out_dir/Trinity.fasta \

--transdecoder_pep Trinity.fasta.transdecoder.pep

Load in the various outputs generated earlier:

% $TRINOTATE_HOME/Trinotate Trinotate.sqlite LOAD_swissprot_blastx \
swissprot.blastx.outfmt6

% $TRINOTATE_HOME/Trinotate Trinotate.sqlite \

LOAD_swissprot_blastp swissprot.blastp.outfmt6

% $TRINOTATE_HOME/Trinotate Trinotate.sqlite LOAD_pfam TrinotatePFAM.out

% $TRINOTATE_HOME/Trinotate Trinotate.sqlite LOAD_signalp signalp.out

Generate the Trinotate Annotation Report

% $TRINOTATE_HOME/Trinotate Trinotate.sqlite report > Trinotate.xls

The above file can be very large. It's often useful to load it into a spreadsheet software tools such as MS-Excel. If you have a transcript identifier of interest, you can always just 'grep' to pull out the annotation for that transcript from this report.

6. Differential expression with kallisto/R/DESeq2

a. Trinity Transcript Quantification (https://github.com/trinityrnaseq/trinityrnaseq/wiki/Trinity-Transcript-Quantification)

There are now several methods available for estimating transcript abundance in a genome-free manner, and these include alignment-based methods (aligning reads to the transcript assembly) and alignment-free methods (typically examining k-mer abundances in the reads and in the resulting assemblies).

In Trinity, we provide direct support for running the alignment-based quantification methods RSEM and eXpress, as well as the ultra-fast alignment-free method kallisto and 'wicked-fast' salmon.

The Trinity software does not come pre-packaged with any of these software tools, so be sure to download and install any that you wish to use. The tools should be available via your PATH setting (so, typing 'which kallisto' or 'which express' on the linux command line returns the path to where the tool is installed on your system).

Estimating Transcript Abundance

#kallisto install(https://pachterlab.github.io/kallisto/source)

prep reference

% $TRINITY_HOME/util/align_and_estimate_abundance.pl --transcripts cm_Trinity.fasta --est_method kallisto --trinity_mode --prep_reference --output_dir ./

align library

% $TRINITY_HOME/util/align_and_estimate_abundance.pl --transcripts cm_Trinity.fasta --seqType fq --left cm_c_1_S1_R1_001.fastq.gz --right cm_c_1_S1_R2_001.fastq.gz --est_method kallisto --trinity_mode --output_dir cm_c_1

switch the columns and add a header

% awk '{ print $2 " " $1}' cm_Trinity.fasta.gene_trans_map | sed '1 i\TXNAME\tGENEID' | sed 's/ /\t/g' > cm_tx2gene.tsv

The key columns in the above salmon output are the transcript identifier 'Name', the 'NumReads' corresponding to the number of RNA-Seq fragments predicted to be derived from that transcript, and the 'TPM' column indicates the normalized expression values for the expression of that transcript in the sample (measured as Transcripts Per Million).

Build Transcript and Gene Expression Matrices

Using the transcript and gene-level abundance estimates for each of your samples, construct a matrix of counts and a matrix of normalized expression values using the following script:

% $TRINITY_HOME/util/abundance_estimates_to_matrix.pl --est_method kallisto \

--gene_trans_map Trinity.fasta.gene_trans_map --out_prefix kallisto \

--name_sample_by_basedir sampleA/abundance.tsv sampleB/abundance.tsv

b. Defferential expression analysis DESeq2 (https://bioconductor.org/packages/release/bioc/vignettes/DESeq2/inst/doc/DESeq2.html)

Read in data

% library(tximport)

% library(readr)

% library("tximportData")

this assumes you are the directory containing the kallisto output

% dir <- "./"

% run <- c("cm_c_1", "cm_c_2", "cm_t_1", "cm_t_2")

% files <- file.path(dir, run, "abundance.tsv")

```
% names(files) <- run

% all(file.exists(files))

% tx2gene <- read_tsv("cm_tx2gene.tsv")

% txi <- tximport(files, type = "kallisto", tx2gene = tx2gene, reader = read_tsv)

#sample information and the design formula

% library(tidyr)

% library(dplyr)

% library(cowplot)

% library(tibble)

# this derives the sample names from the count dataframe - safer than doing it manually

% sampleTable <- as.data.frame(colnames(txi$counts))

% colnames(sampleTable) <- "library"

% sampleTable <- separate(sampleTable, library, into = c("species", "treatment"), sep = "_",
remove = FALSE, extra = "drop")

% sampleTable$treatment <- gsub("^c$", "control", sampleTable$treatment)

% sampleTable$treatment <- gsub("^t$", "treatment", sampleTable$treatment)

% rownames(sampleTable) <- sampleTable$library

# drop variables that won't be fitted

% sampleTable <- dplyr::select(sampleTable, -library, -species)

# The DESeqDataSet object produce

% library(DESeq2)

% dds <- DESeqDataSetFromTximport(txi, sampleTable, ~ treatment)
```

Data transformations and visualization

Count data transformations

In order to test for differential expression, we operate on raw counts and use discrete distributions as described in the previous section on differential expression. However for other downstream analyses – e.g. for visualization or clustering – it might be useful to work with transformed versions of the count data.

Maybe the most obvious choice of transformation is the logarithm. Since count values for a gene can be zero in some conditions (and non-zero in others), some advocate the use of pseudocounts, i.e. transformations of the form:

y=log 2 (n+n 0)

y=log2⁡(n+n0)

where n represents the count values and n 0

is a positive constant.

In this section, we discuss two alternative approaches that offer more theoretical justification and a rational way of choosing parameters equivalent to nabove. One makes use of the concept of variance stabilizing transformations (VST) (Tibshirani 1988; Huber et al. 2003; Anders and Huber 2010), and the other is the regularized logarithm or rlog, which incorporates a prior on the sample differences (Love, Huber, and Anders 2014). Both transformations produce transformed data on the log2 scale which has been normalized with respect to library size or other normalization factors.

The point of these two transformations, the VST and the rlog, is to remove the dependence of the variance on the mean, particularly the high variance of the logarithm of count data when the mean is low. Both VST and rlog use the experiment-wide trend of variance over mean, in order to transform the data to remove the experiment-wide trend. Note that we do not require or desire that all the genes have exactly the same variance after transformation. Indeed, in a figure below, you will see that after the transformations the genes with the same mean do not have exactly the same standard deviations, but that the experiment-wide trend has flattened. It is those genes with row variance above the trend which will allow us to cluster samples into interesting groups.

Note on running time: if you have many samples (e.g. 100s), the rlog function might take too long, and so the vst function will be a faster choice. The rlog and VST have similar properties, but the rlog requires fitting a shrinkage term for each sample and each gene which takes time. See the DESeq2 paper for more discussion on the differences (Love, Huber, and Anders 2014).

% `rld <-` rlog`(dds,` blind=FALSE`)`

% head`(`assay`(rld),` 3`)`

Data quality assessment by sample clustering and visualization

1. **Heatmap of the sample-to-sample distances**

Another use of the transformed data is sample clustering. Here, we apply the *dist* function to the transpose of the transformed count matrix to get sample-to-sample distances.

```
% sampleDists <- dist(t(assay(rld)))

% library("RColorBrewer")

% sampleDistMatrix <- as.matrix(sampleDists)

% rownames(sampleDistMatrix) <- paste(vsd$condition, vsd$type, sep="-")

% colnames(sampleDistMatrix) <- NULL

% colors <- colorRampPalette( rev(brewer.pal(9, "Blues")) )(255)

% pheatmap(sampleDistMatrix,  clustering_distance_rows=sampleDists,\
clustering_distance_cols=sampleDists,  col=colors)
```

2. **Principal component plot of the samples**

Related to the distance matrix is the PCA plot, which shows the samples in the 2D plane spanned by their first two principal components. This type of plot is useful for visualizing the overall effect of experimental covariates and batch effects.

```
% plotPCA(rld, intgroup=c("condition", "type"))
```

It is also possible to customize the PCA plot using the ggplot function.

```
% pcaData <- plotPCA(vsd, intgroup=c("condition", "type"), returnData=TRUE)

% percentVar <- round(100 * attr(pcaData, "percentVar"))

% ggplot(pcaData, aes(PC1, PC2, color=condition, shape=type)) +

 geom_point(size=3) +

 xlab(paste0("PC1: ",percentVar[1],"% variance")) +

 ylab(paste0("PC2: ",percentVar[2],"% variance")) +

 coord_fixed()
```

Differential expression analysis

The standard differential expression analysis steps are wrapped into a single function, *DESeq*. The estimation steps performed by this function are described below, in the manual page for `?DESeq` and in the Methods section of the DESeq2 publication (Love, Huber, and Anders 2014).

Results tables are generated using the function *results*, which extracts a results table with log2 fold changes, *p* values and adjusted *p* values. With no additional arguments to *results*, the log2 fold change and Wald test *p* value will be for the last variable in the design formula, and if this is a factor, the comparison will be the last level of this variable over the first level. However, the order of the variables of the design do not matter so long as the user specifies the comparison using the `name` or `contrast` arguments of *results* (described later and in `?results`).

```
% dds <- DESeq(dds)

% res <- results(dds)
```

Exploring and exporting results

MA-plot

In DESeq2, the function *plotMA* shows the log2 fold changes attributable to a given variable over the mean of normalized counts for all the samples in the *DESeqDataSet*. Points will be colored red if the adjusted *p* value is less than 0.1. Points which fall out of the window are plotted as open triangles pointing either up or down.

```
% plotMA(res, ylim=c(-2,2))
```

It is more useful visualize the MA-plot for the shrunken log2 fold changes, which remove the noise associated with log2 fold changes from low count genes without requiring arbitrary filtering thresholds.

```
% plotMA(resLFC, ylim=c(-2,2))
```

Plot counts

It can also be useful to examine the counts of reads for a single gene across the groups. A simple function for making this plot is plotCounts, which normalizes counts by sequencing depth and adds a pseudocount of 1/2 to allow for log scale plotting. The counts are grouped by the variables in intgroup, where more than one variable can be specified. Here we specify the gene which had the smallest p value from the results table created above. You can select the gene to plot by rowname or by numeric index.

% plotCounts(dds, gene=which.min(res$padj), intgroup="condition")

Write results out

% ResSig<- subset(res, padj<0.05 & abs(log2FoldChange)>1)

% ResUpRe<- subset(res, padj<0.05 & log2FoldChange>1)

% ResDownRe<- subset(res, padj<0.05 & log2FoldChange<-1)

% Write.csv(ResSig, “ResSig.csv”)

Combine DE results with annotation files to find interesting differential expression genes.