Neuro-Genomics
Exercise 1 - Principles of Sequencing Data Analysis
Solution by Lior Avrahami (i.d: 301091369)

The purpose of this exercise is to learn the principles of sequencing data analysis. We will first discuss the importance of sequencing data to modern biology and the expected data structure (Part 1), then we will use R to inspect actual sequencing data (Part 2). We will use descriptive statistics to explore aspects of the data and consider some of the ways to analyze it (Part 3). Next, we will detect differentially expressed genes (Part 4). Then we will analyze time-course sequencing experiments and use the Fourier Transform to detect rhythmic patterns in gene expression (Part 5). Finally, we will integrate a few of the concepts to detect genes with variable expression levels (Part 6).

General instructions to all sections:

a) We expect some prior knowledge in R and in programming in general. Please consult us if this is not the case. In addition, vectorization in R is an important concept to be familiar with: Vectorized Operations | R Programming for Data Science
b) The symbol ## marks tasks to perform
   The symbol # marks comments related to the way the task should be performed
c) For each computational task write the lines of code that you use below the actual task and paste the output as well
d) For tasks that don't require code write your answer below the actual task. Trivial tasks, like installation of packages, don't require a reply.
e) If you don't know the R functions to perform specific tasks, use the web to find the appropriate R functions. For example, R Documentation and manuals are a great source.
f) This exercise can also be done in Matlab. The 'pasilla' data of Part 2 can be found here.
g) Solving most of this exercise in python is also possible, but it is less recommended since not all the functions we use here have one-to-one equivalents in python.

Part 1 - General introduction to sequencing data

Over the last decade next-generation sequencing of RNA has revolutionized molecular biology by starting to provide the molecular basis behind the function of tissues. The idea is simple, we can collect any tissue that we are interested in, extract the RNA from it, and then sequence the extracted RNA using a sequencing machine. [Note: typically only the mRNA are sequenced, using poly(A) enrichment of the mRNA. However, non-poly(A) transcripts can also be sequenced, for example mature microRNA sequencing, and therefore below we will use the term 'RNA' in the context of sequencing].
The output is millions of sequencing fragments ("reads") of the RNA molecules. These reads can then be aligned against the genome and the genes from which they were transcribed can be detected. Therefore, the result is a comprehensive estimate of the expression levels (i.e. number of RNA copies) for all the genes in the tissue. Here we won't focus on the raw sequencing data (the reads), nor will we discuss the alignment process. Instead, we will start exploring the sequencing data after it was processed into a vector with the following structure:

NameOfGene	Counts (the number of reads that align to the specific gene)
Gene1	23
Gene2	7
…	
GeneN	452

Where N is the number of genes in the organism in question.

This expression vector typically represents the information (RNA profile) of one tissue sample for one particular experimental condition. However, the sample may be only a small part of a tissue, several tissues combined or even a small organism (for example, fish larvae). The sample can also be from cells grown outside of a tissue (that is, cultured cells or cell lines).

Read about the leading next-generation sequencing technology, focus on Figure 3 and the process of ‘bridge amplification, which is explained in the Glossary section:
An Introduction to Next-Generation Sequencing Technology

(note: Figure 3 starts from double stranded genomic DNA, but the process is exactly the same for RNA as well, the only difference is that the starting point is double stranded cDNA)
In Illumina sequencing, both the library preparation and the bridge amplification steps contain PCR amplification. However, PCR can induce distortions, read:
Sources of PCR-induced distortions in high-throughput sequencing data sets

Focus on the abstract section, and the beginning of each one of the following results sections: ‘Perfect PCR’, ‘PCR bias’ and ‘Stochastic amplification of low copy number amplicons’.
Can you explain why we can only get an **estimation** of the expression levels and not the actual number of RNA molecules for each gene?

Answer:
In perfect PCR, the relative abundance of molecules from each gene would be preserved, and the sequencing read counts would exactly reflect the true number of molecules. However, In practice RNA-seq provides only an estimate of expression levels rather than absolute RNA molecule counts because PCR amplification introduces several distortions:

- **Bridge Amplification Variability**: In Illumina sequencing, bridge amplification generates clusters on the flow cell, with each cluster producing many reads. The sequencer therefore measures amplified RNA clusters rather than the original RNA molecules, and variability in cluster formation efficiency affects read counts.
- **PCR Bias**: Different sequences amplify with different efficiencies depending on factors such as GC content, secondary structure, and fragment length. As a result, some sequences are preferentially amplified, causing read counts to deviate from the original RNA proportions.
- **Stochastic Amplification of Low Copy Number Amplicons**: When starting from few molecules, PCR amplification is inherently random. Chance events in early PCR cycles can cause some molecules to be amplified much more than others, leading to further distortion of the original expression levels.

Consequently, sequencing read counts reflect relative gene expression rather than the true absolute number of RNA molecules per gene.

Part 2 - Explore sequencing data using R

In this section we will use a dataset from an experiment performed on Drosophila melanogaster cell cultures. The experiment was designed to investigate the effect of RNAi knock-down of the splicing factor pasilla (see Conservation of an RNA regulatory map between Drosophila and mammals). Several samples were collected from untreated fly cells (i.e. control samples), and several samples were collected from fly cells that were treated with RNAi. The dataset is stored in the package 'pasilla'.

In most sequencing experiments, including this one, more than one sample is examined. This allows the detection of genes that have different expression levels between the tested samples (more about that in the next section). However, when testing more than one sample, how can we reliably compare the different expression vectors? In technical terms, the different expression vectors need to be normalized. The easiest normalization method is to account for the fact that different samples can produce different total number of sequencing reads.

To illustrate this point consider the following example: say that RNA was separately extracted from two samples, one of diseased tissue and one of control. The same amount of RNA was loaded into the sequencing machine and sequenced. In principle, the sequencing machine should get a fixed amount X of RNA material as input and produce a fixed amount Y of sequencing reads (X and Y varies between the different types of machines). In practice however the total number of reads varies between different machine runs. Suppose that the first sample in our example generated 20 million reads and the second one generated 10 million reads. One might falsely conclude that many of the genes in the first sample have higher expression levels compared to the second sample. However, a simple normalization which enforces the constraint that the total number of reads to be the same for each run will allow a more valid comparison of the two samples.

We will use the matrix cts. Examine the first 10 lines of this matrix - what kind of information is in the matrix?

Answer:

The matrix contains gene expression count data where the rows represent different genes (gene IDs), the columns represent different samples (treated1, treated2, etc.) and each cell contains the number of sequencing reads aligned to that gene (level of expression).

```
# Load pasilla data from local file
cts = pd.read_csv("neuro genomics - exercise 1/pasilla_gene_counts.tsv", sep='\t',
index_col='gene_id')
# Examine the first 10 lines of the cts matrix
print("\nFirst 10 lines of the cts matrix:")
print(cts.head(10))
```

Output:

```
First 10 lines of the cts matrix:
             untreated1  untreated2  untreated3  untreated4  treated1  treated2  treated3
gene_id
FBgn0000003           0           0           0           0         0         0         1
FBgn0000008          92         161          76          70       140        88        70
FBgn0000014           5           1           0           0         4         0         0
FBgn0000015           0           2           1           2         1         0         0
FBgn0000017        4664        8714        3564        3150      6205      3072      3334
FBgn0000018         583         761         245         310       722       299       308
FBgn0000022           0           1           0           0         0         0         0
FBgn0000024          10          11           3           3        10         7         5
FBgn0000028           0           1           0           0         0         1         1
FBgn0000032        1446        1713         615         672      1698       696       757
```

Define a variable that will hold the dimensions of this matrix and print the matrix dimensions

```
# Define a variable to hold matrix dimensions and print them
dims = cts.shape
print(f"\nMatrix dimensions:")
```

```
print(f"Number of rows (genes): {dims[0]}")
print(f"Number of columns (samples): {dims[1]}")
```

Output:
```
Matrix dimensions:
Number of rows (genes): 14599
Number of columns (samples): 7
```

Is the sum of reads the same for each one of the samples (the different columns)?

```
# Sum the number of reads for each sample
sample_sums = cts.sum(axis=0)
print("\nSum of reads per sample:")
print(sample_sums)
```

Output:
```
Sum of reads per sample:
untreated1    13972512
untreated2    21911438
untreated3     8358426
untreated4     9841335
treated1      18670279
treated2       9571826
treated3      10343856
```

Answer:
No, the sum of reads is not the same for each sample. Different samples have different total read counts due to technical variability in the sequencing process.Hence, normalization is necessary for valid comparison between samples.

Create a normalized version of the cts matrix
multiply each one of the columns (except the first one) by a factor so that the total number of reads per column will be equal to the first column

```
# Multiply each column by a factor so total reads equal the first column
target_sum = sample_sums.iloc[0]
normalization_factors = target_sum / sample_sums
# Create normalized cts matrix
cts_normalized = cts.multiply(normalization_factors, axis=1)
```

Make sure that in the normalized matrix the sum of reads is the same in all samples

```
# Verify that normalized sums are equal
normalized_sums = cts_normalized.sum(axis=0)
print("\nSum of reads per sample after normalization:")
print(normalized_sums)
```

Output:
```
Sum of reads per sample after normalization:
untreated1    13972512.0
untreated2    13972512.0
untreated3    13972512.0
untreated4    13972512.0
treated1      13972512.0
treated2      13972512.0
treated3      13972512.0
```

Part 3 - Basic statistics of sequencing data

A typical sequencing experiment, like the one that we are examining, is normally designed to address the following question - which genes have different expression levels when comparing two (or more) experimental conditions. However, detecting differentially expressed genes is not trivial. Consider the following example: say that we have two normalized expression vectors, one of condition A and one of condition B, and gene X has expression level 150 in condition A and 200 in condition B. How can we know if gene X is differentially expressed between the two conditions? One can conclude from the increase in counts that gene X is influenced by the conditions tested. However, another likely explanation is that the increase in counts is due to either technical variations (noise introduced by the sequencing procedure) or biological variations (for example, difference in expression levels due to the inherent stochasticity of gene expression). Therefore, technical repeats (samples generated from the same biologically material) and/or biological repeats (samples generated from biologically distinct material) are needed in each sequencing experiment. Repeats are essential, but unfortunately the number of repeats that can be performed is usually limited by experimental and financial considerations.

Given that usually only a handful of repeats are performed for each condition, it's practically impossible to deduce the technical and biological variations in the expression level of each gene from the data alone. Therefore we need to model the data (here 'data' is defined as the expression levels of each gene in one experimental condition but with several repeats) using well-known distributions. Knowing the distribution will provide the expected variability in the expression level of each gene, and allow comparison of different experimental conditions.

However, if we use a well-known distribution that doesn't really model the data, then our estimation about the expected variability will be wrong. In this section we will model the data using two well-known distributions.

Does the data fit a Poisson distribution?

recall that the variance of the Poisson distribution is equal to its mean

first calculate the variance and the mean in the expression of each gene for the untreated conditions

then log transform the variance and mean (because the resulting values span multiple orders of magnitude). Tip: add 1 to all the values to avoid log of 0.

plot the variance vs the mean. Add the line x=y to your plot. The line represents a perfect Poisson distribution

Does the data fit a Negative binomial distribution?

In Negative binomial distribution the relationship between the variance and the mean is: variance = mean + a*mean^2 where 'a' is the dispersion parameter, a measure of how dispersed are the data compared to the Poisson distribution

use the nls function to fit a curve of variance = mean + a*mean^2 to the log transformed variance and mean. Tip - set the initial guess of the value of a to 0.

plot the resulting fitted curve on the variance vs mean plot

What is the value of the dispersion parameter?

Perform the same analysis (including the plots) for the treated samples. What is the dispersion parameter?

Read about the statistics of sequencing experiments:

Why sequencing data is modeled as negative binomial

Judging by the results obtained, do you think that the different untreated and treated samples in this experiment are technical or biological repeats?

Answer:

In Poisson distribution the variance equals the mean. Looking at the scatter plot of Pasilla data, the variance is consistently higher than the mean. Hence it does not fit Poisson distribution. This is called "over-dispersion" and is typical of RNA-seq data.

The data better fits a Negative Binomial distribution, which has an additional dispersion parameter to account for this extra variance.

The treated and untreated samples are biological repeats, because the variance is larger than the mean, indicating biological variability rather than sequencing noise. In technical repeats, the dispersion parameter $a$would be close to zero, consistent with a Poisson model. In contrast, both treated and untreated samples show $a > 0$, reflecting overdispersion. The different dispersion levels between conditions arise from true biological differences in gene expression.

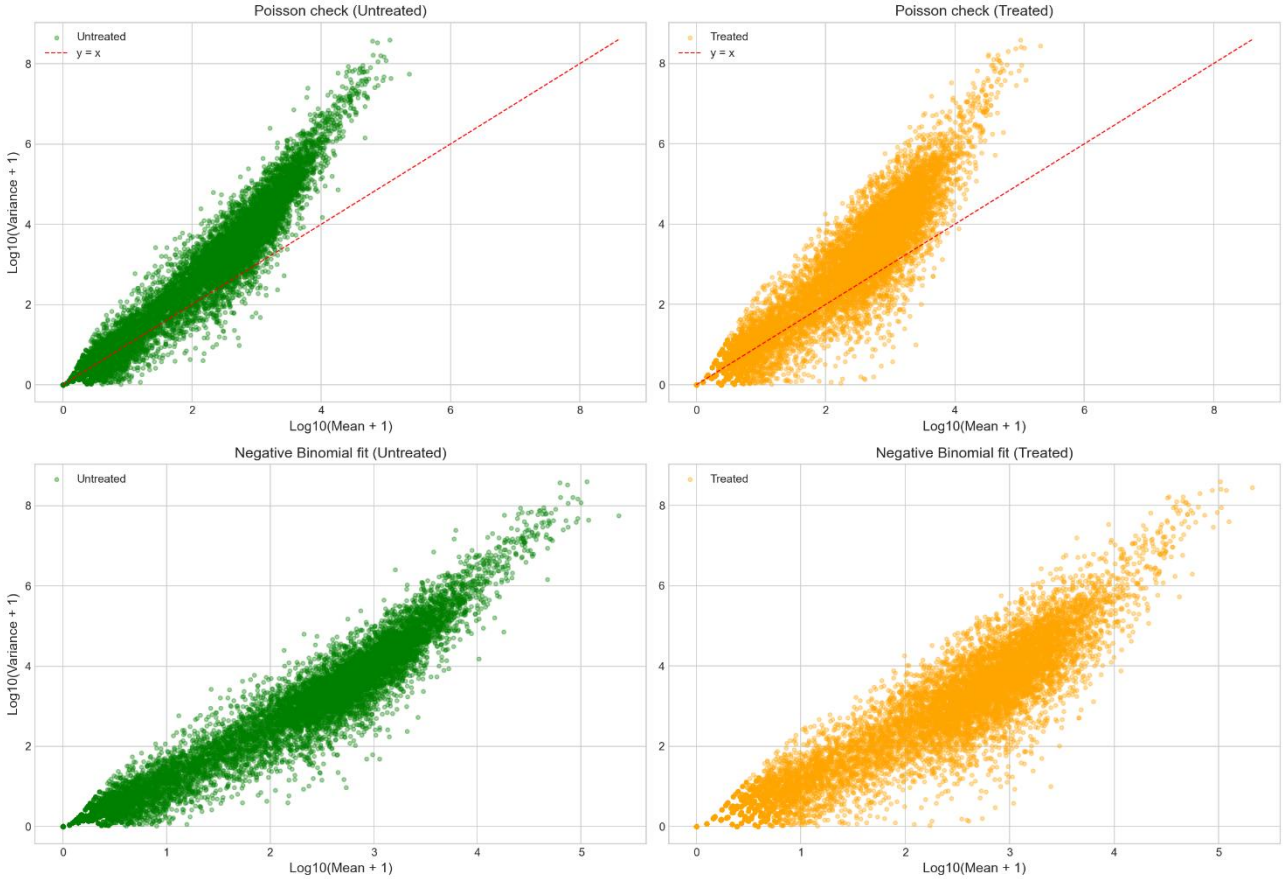
```
# ================================================================================
# PART 3 - Basic Statistics of Sequencing Data
# ================================================================================

print("\n" + "=" * 80)
print("PART 3 - Basic Statistics of Sequencing Data")
print("=" * 80)

# Define untreated and treated sample columns
untreated_cols = [col for col in cts.columns if 'untreated' in col]
treated_cols = [col for col in cts.columns if 'treated' in col and 'untreated' not
in col]

# Compute means and variances per group (all, treated, untreated)
means = {}
vars_ = {}

# means['all'] = cts_normalized.mean(axis=1)
# vars_['all'] = cts_normalized.var(axis=1)

means['treated'] = cts_normalized[treated_cols].mean(axis=1)
vars_['treated'] = cts_normalized[treated_cols].var(axis=1)

means['untreated'] = cts_normalized[untreated_cols].mean(axis=1)
vars_['untreated'] = cts_normalized[untreated_cols].var(axis=1)

# Log transform (adding 1 to avoid log(0)) for treated and untreated groups
log_means_treated = np.log10(means['treated'] + 1)
log_vars_treated = np.log10(vars_['treated'] + 1)

log_means_untreated = np.log10(means['untreated'] + 1)
log_vars_untreated = np.log10(vars_['untreated'] + 1)

# Fit dispersion a in variance = mean + a*mean^2 using curve_fit instead of nls
function in R
x_t = means['treated'].values
y_t = vars_['treated'].values
popt_t, _ = curve_fit(lambda mu, a: mu + a * (mu ** 2), x_t, y_t, p0=[0.0],
maxfev=10000)
a_treated = popt_t[0]

x_u = means['untreated'].values
y_u = vars_['untreated'].values
popt_u, _ = curve_fit(lambda mu, a: mu + a * (mu ** 2), x_u, y_u, p0=[0.0],
maxfev=10000)
a_untreated = popt_u[0]

# Build 2x2 subplot: top row Poisson checks, bottom row Negative Binomial fits
fig, axes = plt.subplots(2, 2, figsize=(16, 12), sharey=False)

# Poisson - Untreated (top-left)
ax00 = axes[0, 0]
```

```
ax00.scatter(log_means_untreated, log_vars_untreated, alpha=0.35, s=12, c='green',
label='Untreated')
min_u = min(log_means_untreated.min(), log_vars_untreated.min())
max_u = max(log_means_untreated.max(), log_vars_untreated.max())
ax00.plot([min_u, max_u], [min_u, max_u], 'r--', linewidth=1, label='y = x')
ax00.set_xlabel('Log10(Mean + 1)', fontsize=12)
ax00.set_ylabel('Log10(Variance + 1)', fontsize=12)
ax00.set_title('Poisson check (Untreated)', fontsize=13)
ax00.legend()

# Poisson - Treated (top-right)
ax01 = axes[0, 1]
ax01.scatter(log_means_treated, log_vars_treated, alpha=0.35, s=12, c='orange',
label='Treated')
min_t = min(log_means_treated.min(), log_vars_treated.min())
max_t = max(log_means_treated.max(), log_vars_treated.max())
ax01.plot([min_t, max_t], [min_t, max_t], 'r--', linewidth=1, label='y = x')
ax01.set_xlabel('Log10(Mean + 1)', fontsize=12)
ax01.set_title('Poisson check (Treated)', fontsize=13)
ax01.legend()

# Negative Binomial - Untreated (bottom-left)
ax10 = axes[1, 0]
ax10.scatter(log_means_untreated, log_vars_untreated, alpha=0.35, s=12, c='green',
label='Untreated')
xfit = np.linspace(means['untreated'].min(), means['untreated'].max(), 200)
yfit = xfit + a_untreated * (xfit ** 2)
ax10.plot(np.log10(xfit + 1), np.log10(yfit + 1), 'k-', linewidth=1.5, label=f'NB
fit (a={a_untreated:.3g})')
ax10.set_xlabel('Log10(Mean + 1)', fontsize=12)
ax10.set_ylabel('Log10(Variance + 1)', fontsize=12)
ax10.set_title('Negative Binomial fit (Untreated)', fontsize=13)
ax10.legend()

# Negative Binomial - Treated (bottom-right)
ax11 = axes[1, 1]
ax11.scatter(log_means_treated, log_vars_treated, alpha=0.35, s=12, c='orange',
label='Treated')
xfit = np.linspace(means['treated'].min(), means['treated'].max(), 200)
yfit = xfit + a_treated * (xfit ** 2)
ax11.plot(np.log10(xfit + 1), np.log10(yfit + 1), 'k-', linewidth=1.5, label=f'NB
fit (a={a_treated:.3g})')
ax11.set_xlabel('Log10(Mean + 1)', fontsize=12)
ax11.set_title('Negative Binomial fit (Treated)', fontsize=13)
ax11.legend()

plt.suptitle('Mean vs Variance: Poisson (top) and Negative Binomial (bottom)',
fontsize=16)
plt.tight_layout(rect=[0, 0, 1, 0.95])
plt.savefig('Part3_dispersion_poisson_nb.png', dpi=150)
plt.close()
print(f"Dispersion parameter (a) untreated: {a_untreated:.4g}")
print(f"Dispersion parameter (a) treated:   {a_treated:.4g}")
```

Output:

```
Dispersion parameter (a) untreated: 0.00561
Dispersion parameter (a) treated:   0.0105
```

Mean vs Variance: Poisson (top) and Negative Binomial (bottom)



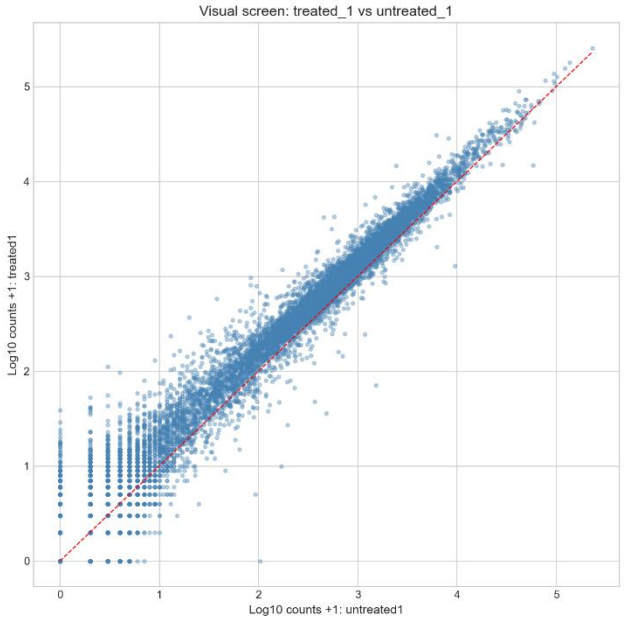
Part 4 - Detect differentially expressed genes

In this section we will first detect different expression levels between the two conditions (treated and untreated) using visual inspection. However, as discussed in the previous section, we can use the observation that the data fit a well-known distribution to estimate the expected variation in the expression level and to calculate the p-value that the gene is differentially expression between the two conditions. For that, we will use the R software DESeq, that will allow detection of all the differentially expressed genes between the treated and untreated conditions.

Using visual inspection, detect at least one gene that has different expression levels in the first treated sample compared to the first untreated sample
The first step is to plot the log of expression in one sample against the log of expression in the other sample (log transformation is used because the resulting values span multiple orders of magnitude)
Then use the obtained plot to visually locate gene(s) that has strikingly different expression level in the two conditions
Characterize the expression of the gene(s) you selected using a grid-like manner: for example, gene(s) that has expression levels higher than value X in the treated condition, and lower than expression Y in the untreated condition
Use the grid-like characterization to computationally detect the name of the gene(s)

```
# Visual inspection: first treated vs first untreated
first_untreated = untreated_cols[0]
first_treated = treated_cols[0]

log_u1 = np.log10(cts[first_untreated] + 1)
log_t1 = np.log10(cts[first_treated] + 1)

fig, ax = plt.subplots(figsize=(8, 8))
ax.scatter(log_u1, log_t1, alpha=0.35, s=10, c='steelblue')
ax.set_xlabel(f'Log10 counts +1: {first_untreated}', fontsize=11)
ax.set_ylabel(f'Log10 counts +1: {first_treated}', fontsize=11)
ax.set_title('Visual screen: treated_1 vs first untreated_1', fontsize=13)
ax.plot([log_u1.min(), log_u1.max()], [log_u1.min(), log_u1.max()], 'r--',
linewidth=1)
plt.tight_layout()
plt.savefig('Part4_visual_first_pair.png', dpi=150)
plt.close()
```

Output:



```
# Grid-like filter to flag striking genes based on visual inspection of the scatter
plot
treat_thresh = 2.0  # Log10 scale: genes with counts > 100 in treated
untreat_thresh = 1.0  # Log10 scale: genes with counts < 10 in untreated

grid_mask = ((log_t1 >= treat_thresh) & (log_u1 <= untreat_thresh))
grid_genes = cts.index[grid_mask].tolist()

print("\nGenes visually flagged (treated high, untreated low):")
print(grid_genes)
```

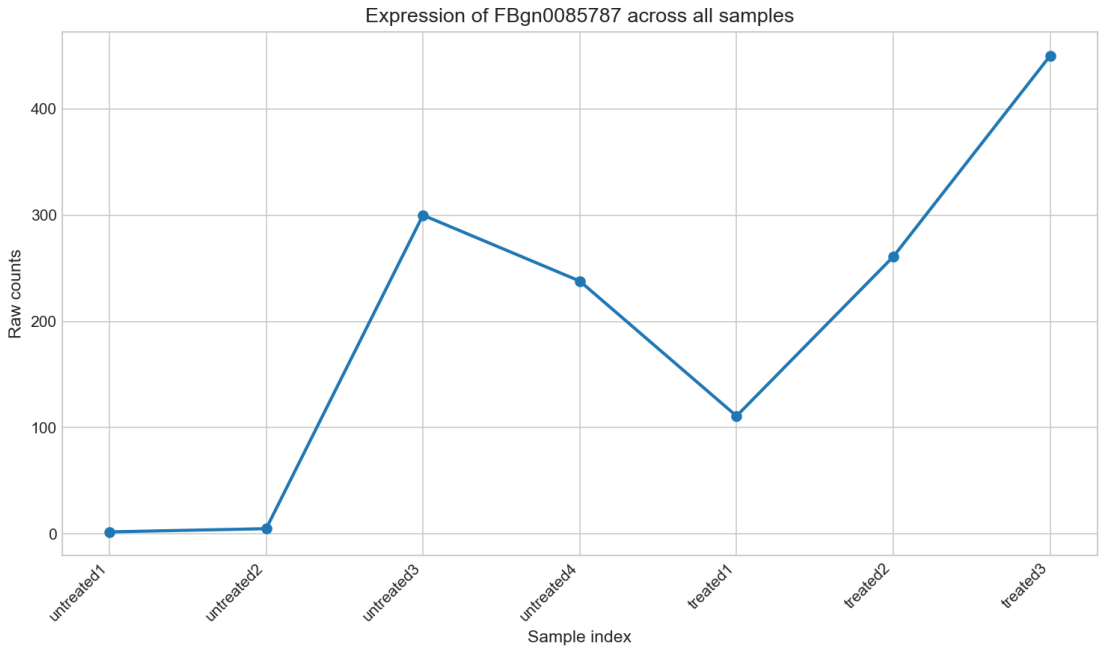
Output:

```
Genes visually flagged (treated high, untreated low):
['FBgn0028513', 'FBgn0085787']
```

Choose one of the genes you detected using visual inspection and plot the expression of this gene in all the conditions in this experiment. The x-axis should be a running index and the y-axis should be the expression of this gene in the three treated conditions and the four untreated conditions, in the same order as in the 'cts' matrix. Is the expression of this gene consistently different between all the treated and untreated conditions? YES

```
# Pick one flagged gene for full-profile plot
selected_gene = grid_genes[1]

selected_expr = cts.loc[selected_gene]
fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(range(len(cts.columns)), selected_expr.values, 'o-', linewidth=2,
markersize=6)
ax.set_xticks(range(len(cts.columns)))
ax.set_xticklabels(cts.columns, rotation=45, ha='right')
ax.set_xlabel('Sample index', fontsize=11)
ax.set_ylabel('Raw counts', fontsize=11)
ax.set_title(f'Expression of {selected_gene} across all samples', fontsize=13)
plt.tight_layout()
plt.savefig('Part4_selected_gene_profile.png', dpi=150)
plt.close()
```

Output:



read the abstract part of the manuscripts describing the R tool 'DEseq', which allows the detection of differentially expressed genes: Differential expression analysis for sequence count data Does the data we have fit the basic assumptions of DEseq? YES

use DESeq to detect the probability that each one of the genes is differentially expressed between the two conditions

the matrix 'res' contains the p-value for all the genes - sort the genes according to the p-value, such that lower p-values will appear first

detect 10 genes that are most different between the treated and untreated according to the obtained p-values. Display only the column ‘log2FoldChange’ (the log transformed fold change between the two conditions) and the column ‘padj’ (the adjusted p-value, see: Multiple comparisons problem) of these 10 genes. Is the gene that you detected using visual inspection among them?

```
# The DESeq2 methodology was implemented in Python using the PyDESeq2 package
# Condition labels and metadata
conditions = ['treated' if 'treated' in col and 'untreated' not in col else
'untreated' for col in cts.columns]
metadata = pd.DataFrame({'condition': conditions}, index=cts.columns)

# PyDESeq2 expects samples as rows, genes as columns
counts_t = cts.T.copy()

# Create DeseqDataSet and run DESeq2
dds = DeseqDataSet(
    counts=counts_t,
    metadata=metadata,
    design_factors="condition",
    ref_level=['condition', 'untreated']  # Set untreated as reference
)
dds.deseq2()

# Get results (treated vs untreated)
stats_result = DeseqStats(dds, contrast=['condition', 'treated', 'untreated'])
stats_result.summary()
results = stats_result.results_df

# Sort by adjusted p-value
results_sorted = results.sort_values('padj')

# Get top 10 DE genes
top10_de = results_sorted.head(10)[['log2FoldChange', 'padj']]

print("\nTop 10 differentially expressed genes (sorted by padj):")
print("=" * 60)
print(top10_de)

# Check if visually inspected gene is in top 10
print(f"\nVisually inspected gene: {selected_gene}")
if selected_gene in top10_de.index:
    print(f"{selected_gene} is in the top 10 DE genes!")
    print(f"log2FoldChange: {top10_de.loc[selected_gene, 'log2FoldChange']:.4f}")
    print(f"padj: {top10_de.loc[selected_gene, 'padj']:.4e}")
else:
    print(f"{selected_gene} is not in the top 10 DE genes")
    if selected_gene in results.index:
```

```
        rank = list(results_sorted.index).index(selected_gene) + 1
        print(f"(It ranks #{rank} overall with padj = {results.loc[selected_gene,
'padj']:.4e})")
```

Output:

```
Top 10 differentially expressed genes (sorted by padj):
============================================================
             log2FoldChange           padj
gene_id
FBgn0039155       -4.618657    3.016360e-159
FBgn0025111        2.899780    2.208245e-109
FBgn0029167       -2.197045    4.497807e-105
FBgn0003360       -3.179663    3.168454e-103
FBgn0035085       -2.560296    1.935523e-73
FBgn0039827       -4.162618    4.720095e-69
FBgn0034736       -3.511584    1.307244e-57
FBgn0029896       -2.445075    5.905634e-55
FBgn0000071        2.679649    3.921820e-46
FBgn0051092        2.327568    2.182553e-37

Visually inspected gene: FBgn0085787
FBgn0085787 is not in the top 10 DE genes
(It ranks #6546 overall with padj = 9.1920e-01)
```

Part 5 - Detect rhythmical patterns using Fourier transform

In the previous sections we focused on differential genes expression (DGE), which is the classical type of analysis of RNA sequencing data. DGE can be framed as follows - given two experimental conditions A and B, each with one or more technical and biological repeats, detect a set of genes that have different expression levels in A compared to their expression levels in B. Naturally, this analysis (and any other kind of analysis for that matter), can't fit all experimental designs and questions.

In this section we will focus on time-course experiments. In these experiments a sample is collected at different points in time and the RNA is sequenced. This allows following the time-dependent behavior of gene expression on a genome-wide scale. These experiments are often technically challenging because a given sample can only be sequenced at one time point because extraction of the RNA for sequencing destroys the sample. This technical challenge is usually overcome using biological replicates: consider for example an experiment designed to follow the time-dependent gene expression following administration of a drug to a model animal - several animals can be treated with the drug simultaneously, and at each time point, one or more animals are sacrificed and the RNA from the tissue of interest is collected for sequencing. The data analysis aspects of time-course experiments are also challenging. If the data contains some rhythmical patterns, then the problem can be addressed with Fast Fourier transform. This is the case with Circadian rhythms. While physiological examples of circadian rhythms have been known for hundreds of years, and the key genes that make up the core circadian clock have been known for almost 50 years, only over the last decade has next-generation sequencing of RNA revealed the full extent of circadian rhythms; it turns out that cellular circadian clocks are present in most (if not all) cell types, tissues and organisms. The cellular circadian clocks in turn cause circadian rhythms in hundreds of genes, which relate to multiple physiological pathways.

We will explore RNA sequencing data generated in zebrafish. This is a time-course experiment designed to identify genes with circadian expression patterns. We will start by visual exploration of the data in the time domain, and then move to the frequency domain to process the data.

Load the file CircadianRNAseq.csv which contains the processed RNA sequencing data into R. Examine the last 5 rows of the matrix, what is the time step (i.e. the time in hours between each measurement)?

download the file and store it locally in the computer
use the function 'read.csv' to read this csv file
use the function 'as.matrix' so that the dataset will be read as a matrix

Additional information regarding the data:

Each row represents one gene. The first column 'RefSeqID' gives the official gene annotation, and the last column 'GeneSymbol' gives the common name of the gene. Columns 2 to 13 contain the time-course measurements which were performed over two consecutive days ('A' means day 1 and 'B' is day 2). Each value in columns 2 to 13 represents the number of sequencing reads that were aligned against the gene in a given row. However, these values are not raw count data - these are normalized count values (therefore - no further normalization is needed).

Answer:

The time step is 4 hours, and the total duration is 48 hours with 12 time points.

```
# Load circadian data
circadian_file = "neuro genomics - exercise 1\CircadianRNAseq.csv"
circadian_data = pd.read_csv(circadian_file)
# Examine last 5 rows
print("\nLast 5 rows of the circadian matrix:")
print(circadian_data.tail(5))
```

Output:

```
Last 5 rows of the circadian matrix:
          RefSeqID     A_11PM     A_3AM     A_7AM    A_11AM     A_3PM     A_7PM    B_11PM     B_3AM     B_7AM    B_11AM     B_3PM     B_7PM GeneSymbol
14834  NM_001040052  255.99900  209.97600  125.42200  143.07200  112.50300  101.81700  201.12300  118.10400  255.72500  89.71120  92.12400  86.27090  zgc:136896
14835     NM_131320    5.72658    6.62624    6.62286    9.62324    5.98339    5.37718    3.90986    9.07169    6.51965   6.85435   5.85492   4.42235        LCP1
14836  NM_001077788   11.32120   14.44910   15.91110   11.89190   13.33890   12.65970   10.15590   16.13480   13.85730  15.99760  15.21870  16.85700   Rab3gap1
14837  NM_001003944   50.64750   56.36470   66.65160   85.98560   54.14030   59.21070   53.61420   58.49680   51.49100  46.21930  50.31010  45.71560       ACTR3
14838     NM_213041   13.65380   15.55590   23.63540   18.67810   18.75290   16.04330   13.86520   20.21520   15.92240  19.05500  19.76430  19.02300       UBXN4
```

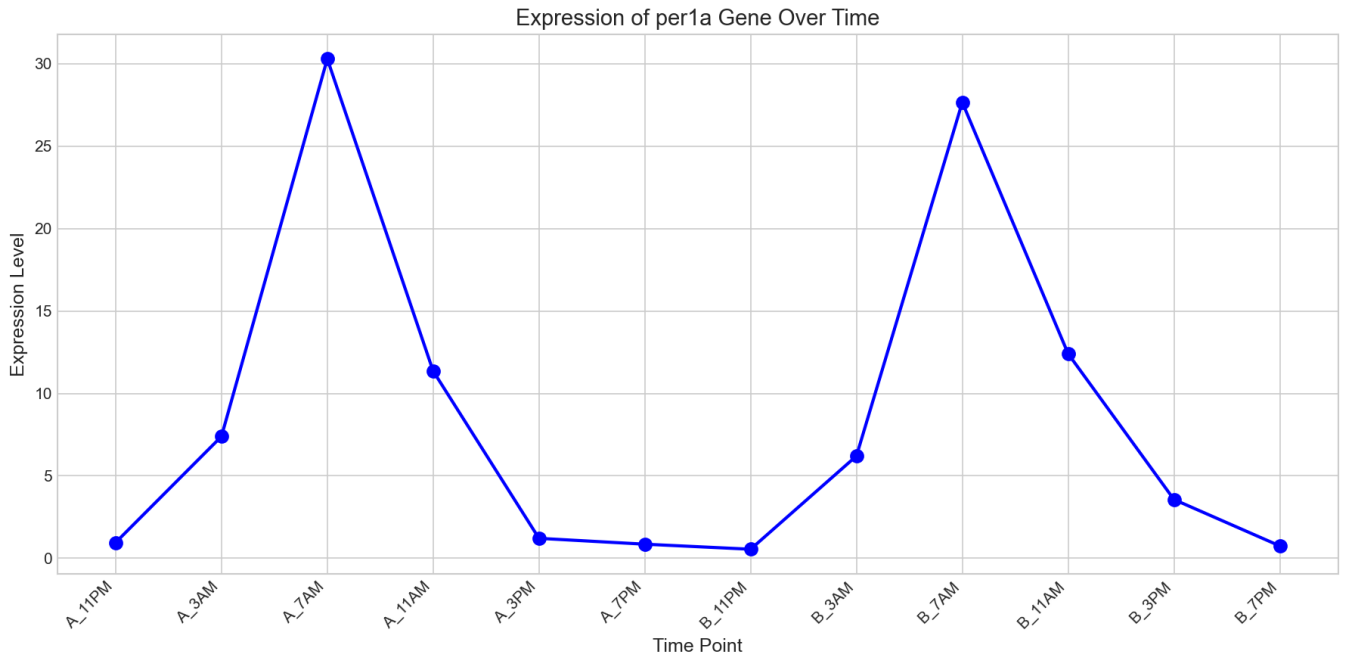
Plot the expression of the gene 'per1a' at all the time points. The x-axis labels should be all the time points ("A_11PM"...). In the plot show both the data points and a line that connects them. Does the expression of this gene seem circadian? Read about the gene Per1 - should this gene be circadian?
first detect the location of this gene in the last column ('GeneSymbol') of the matrix
use xaxt="n" in the 'plot' function to remove the default running index in the x-axis #
then use the function 'axis' to define the x-axis labels

Answer:

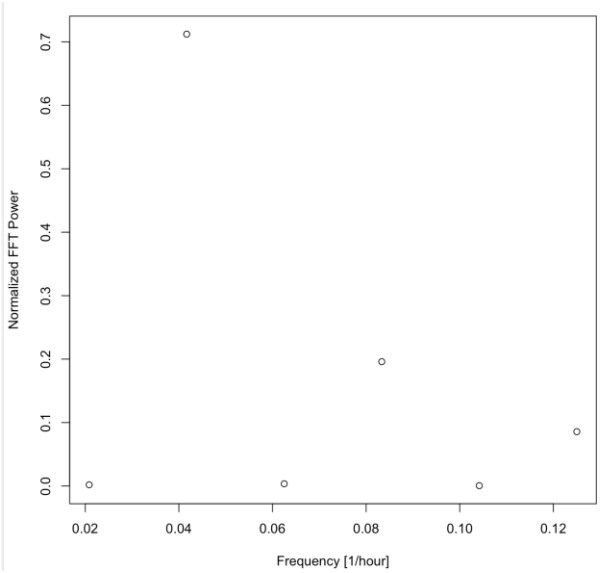
The per1a shows clear circadian expression patterns with oscillation across the 24-hour cycle. Per1 (Period 1) is a core clock gene that is part of the molecular circadian oscillator. It is one of the best-known circadian genes, showing robust ~24-hour rhythms in all organisms studied.

```
# Identify columns
time_cols = [col for col in circadian_data.columns if col not in ['RefSeqID',
'GeneSymbol']]

# Find the gene 'per1a'
per1a_mask = circadian_data['GeneSymbol'] == 'per1a'
per1a_row = circadian_data[per1a_mask].iloc[0]
per1a_expr = per1a_row[time_cols].values.astype(float)

# Plot expression of gene 'per1a'
fig, ax = plt.subplots(figsize=(12, 6))
ax.plot(range(len(per1a_expr)), per1a_expr, 'b-o', linewidth=2, markersize=8)
ax.set_xticks(range(len(time_cols)))
ax.set_xticklabels(time_cols, rotation=45, ha='right')
ax.set_xlabel('Time Point', fontsize=12)
ax.set_ylabel('Expression Level', fontsize=12)
ax.set_title('Expression of per1a Gene Over Time', fontsize=14)
plt.tight_layout()
plt.savefig('Part5_per1a_expression.png', dpi=150)
plt.close()
```

Output:



We will convert the data from the time domain to the frequency domain to determine if the expression profile indeed corresponds to a 24 hr period. For the gene ‘per1a’, calculate, using the fast Fourier transform, the power (squared amplitude) of the different frequencies measured in this experiment. The aim is to generate the following plot (which clearly shows that the power in the circadian frequency 1/24 is the highest):



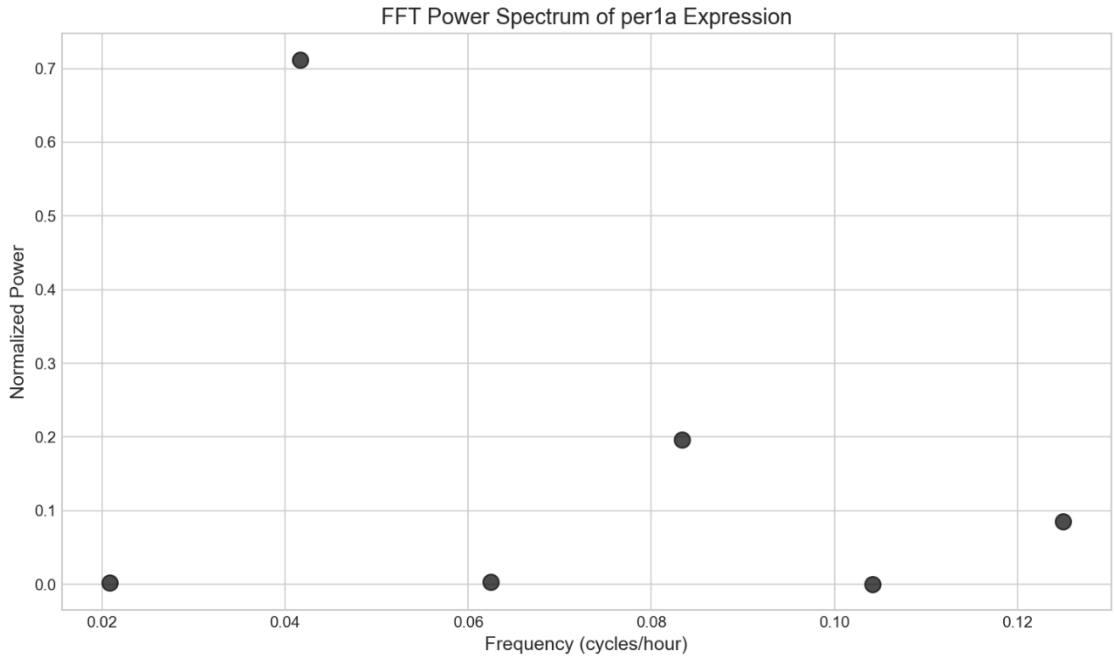
first use the function ‘as.numeric’ to convert the values in the matrix into numerical values # compute the fast discrete Fourier transform using the function ‘fft’
compute the power of the discrete Fourier transform by multiplying the resulting values of the ‘fft’ function by the corresponding conjugate values (recall that the discrete Fourier transform generate complex values, and therefore multiplying them by the conjugated values generate real numbers)
examine the vector of the discrete Fourier transform powers. The first value in this vector represents the power of frequency zero and it will be ignored. The rest of the vector is palindromic (the power spectrum is reflected around N/2) with only 6 unique values in locations 2:7 - these represent the powers in the FFT frequencies (that will be computed below), starting from the lowest frequency and moving to the highest. Normalize this subset of powers (which correspond to the FFT frequencies) by dividing each member with the sum of all powers.

create a vector of the frequencies that are present in the discrete Fourier transform of this time-course experiment. Recall that in discrete Fourier transform the highest frequency is the Nyquist frequency, which is half of the sampling rate, namely 1/(2*DeltaT) whereas DeltaT is the time step (the sampling rate fs is one over the time step). Both the lowest frequency and the frequency step are fs/N or 1/(N*DeltaT) whereas N is the number of time points. Use the function 'seq' to create the vector of the frequencies.
finally, plot the powers against the corresponding frequencies

```
# TASK: Calculate FFT power spectrum for per1a
# ------------------------------------------------------------------------

N = len(per1a_expr)  # Number of time points (12)
delta_t = 4  # Time step in hours

# Compute FFT
fft_result = fft(per1a_expr)

# Compute power (multiply by complex conjugate)
power = np.abs(fft_result) ** 2

# The relevant powers are in positions 1 to N//2
relevant_powers = power[1:N//2 + 1]
```

```
# Normalize powers
normalized_powers = relevant_powers / np.sum(relevant_powers)

# Create frequency vector
# Frequency resolution: 1/(N*delta_t) = 1/(12*4) = 1/48 cycles per hour
# Frequencies: from 1/48 to (N/2)/(N*delta_t) = 6/48 = 1/8 (Nyquist)
frequencies = np.arange(1, N//2 + 1) / (N * delta_t)

# Convert to period in hours
periods = 1 / frequencies

# Plot power spectrum
fig, ax = plt.subplots(figsize=(10, 6))
ax.scatter(frequencies, normalized_powers, s=100, color='black', alpha=0.7,
edgecolors='black', linewidth=1)
ax.set_xlabel('Frequency (cycles/hour)', fontsize=12)
ax.set_ylabel('Normalized Power', fontsize=12)
ax.set_title('FFT Power Spectrum of per1a Expression', fontsize=14)
plt.tight_layout()
plt.savefig('Part5_per1a_FFT_power.png', dpi=150)
plt.close()
```

Output:



Assume that you want to improve the design of this experiment to better detect the circadian frequency using the FFT. Which option do you think is better:

(a) to design an experiment in which the time step will be shorter, for example 2 hours between each time measurement, while keeping the overall measured time 48 hours.

Or:

(b) to increase the overall number of days measured, for example 4 days instead of 2 days, while keeping the time step the same.

Answer:

Increasing the number of days measured is better because it improves frequency resolution. With more days, we can better distinguish the circadian frequency from nearby frequencies, reducing spectral leakage and providing more precise detection of the 24h period.

discuss the potential advantages of detecting circadian genes in the frequency domain versus analyzing the data in the time domain (for example - by trying to fit the expression pattern of a gene with a cosine with 24 hours period).

Answer:

Detecting circadian genes in the frequency domain has several advantages over analyzing the data in the time domain. Using FFT provides an objective, single numerical measure (power at the circadian frequency) that can be directly compared across genes, without subjective curve fitting. Frequency-domain methods are robust to noise because true periodic signals concentrate power at specific frequencies. They also separate circadian signals from slow trends and fast noise, making the 24-hour component easier to isolate even in imperfect data. FFT makes no assumptions about waveform shape, works efficiently at genome scale, and can reveal multiple periodicities such as 24-hour rhythms and their harmonics. Overall, frequency-domain analysis is well suited for large, noisy, or unevenly sampled datasets and works well as a screening tool before detailed time-domain modeling.

Process all the genes in the dataset and sort them according to the normalized FFT power in the circadian frequency of 1/24. Print the common names ('GeneSymbol') of the 10 genes with the highest normalized powers. Look them up using the web and find at least one known circadian gene among them to validate the analysis.
store all the circadian frequency powers in a vector

use the function ‘order’ to sort the vector and report the indexes of the sorted vector. Tip - use the ‘abs’ function to make sure that R stores the powers as real numbers. Also make sure that NA values (NA stands for ‘not available’ and will be generated if the fft was performed on a vector of zeros) are ordered last.

```
# TASK: Find top 10 genes with highest circadian power
# ------------------------------------------------------------------------------

# Store all the circadian frequency powers in a vector
circadian_powers = []

N = len(time_cols)  # Number of time points (12)
delta_t = 4  # Time step in hours

for idx, row in circadian_data.iterrows():
    expr = row[time_cols].values.astype(float)

    # Compute FFT
    fft_result = fft(expr)
    power = np.abs(fft_result) ** 2

    # Get relevant powers (exclude DC component at position 0)
    relevant_powers = power[1:N//2 + 1]

    # Normalize powers
    total_power = np.sum(relevant_powers)
    normalized_powers = relevant_powers / total_power

    circadian_power = normalized_powers[1]  # 24-hour component
    circadian_powers.append(circadian_power)

# Add circadian powers to dataframe
circadian_data['circadian_power'] = circadian_powers
# Sort by circadian power (NAs will be at the end)
sorted_data = circadian_data.sort_values('circadian_power', ascending=False,
na_position='last')
```

```python
# Get top 10 genes (excluding NAs)
top10_genes = sorted_data.dropna(subset=['circadian_power']).head(10)

print("\nTop 10 genes with highest normalized circadian power:")
print("-" * 60)
for i, (idx, row) in enumerate(top10_genes.iterrows()):
    print(f"{i+1:2d}. {row['GeneSymbol']:<20} Power: {row['circadian_power']:.4f}")
```

Output:

```
Top 10 genes with highest normalized circadian power:
------------------------------------------------------------
 1. atxn1b               Power: 0.9659
 2. fus                  Power: 0.9576
 3. nr1d2b               Power: 0.9519
 4. rdh11                Power: 0.9505
 5. ankrd10a             Power: 0.9502
 6. phyhd1               Power: 0.9460
 7. arntl1b              Power: 0.9420
 8. ARNTL2               Power: 0.9276
 9. Ldhd                 Power: 0.9251
10. aclya                Power: 0.9218
```

Answer:

The clearly known circadian genes from the top 10 list above are:

- arntl1b – this is a paralog of ARNTL (BMAL1), a core circadian clock transcription factor.
- ARNTL2 – a close homolog of BMAL1, a part of the core molecular clock.
- nr1d2b – a paralog of NR1D2 (REV-ERBβ), a core circadian repressor that regulates BMAL1/ARNTL activity.

Part 6 - Detecting genes with variable expression levels

One of key tasks in RNA sequencing data analysis is revealing genes with aberration of expression levels between different treatments/conditions or between different time points. These genes are generally termed 'variable genes'. Detection of variable genes is possible even without a robust statistical model of the biological and technical repeats and therefore it can be useful in cases where detection of differentially expressed genes (described previously) can't be performed. Moreover, detection of variable genes is often used as a form of dimension reduction; instead of analyzing all genes in the experiment (usually >10,000 genes), focusing only on the variable genes can significantly reduce the dimensionality by an order of magnitude or more. Lastly, detection of variable genes is an essential step in single-cells RNA sequencing, in which methods for detection of differentially expressed genes are not established yet.

How do we define and detect variable genes? As demonstrated in the previous parts, for a given gene, the variance in the expression levels is highly dependent on the average expression level. Thus, variable genes are often defined as genes with variance (between different treatments/conditions/time points etc) that is higher than the expected variance for genes with similar expression levels (see for example the methods section of: Highly parallel genome-wide expression profiling of individual cells using nanoliter droplets). In this part we will again use the circadian dataset to detect variable genes and examine if and how they relate to circadian genes detection.

Identify the set of genes that was most variable across the circadian dataset, after controlling for the relationship between mean expression and variability. Calculate the mean and a variance for each gene across the different time points, and place genes into 20 bins based on their average expression, starting from a minimal average expression of 3 (log-transformed). Within each bin, z-normalize the variance of all genes within the bin, in order to identify outlier genes whose expression values were highly variable even when compared to genes with similar average expression. Print the common names ('GeneSymbol') of the 40 genes with the highest z-values. Can you detect known circadian genes among them? Is it expected that circadian genes will also be variable genes?

Detailed instructions:

Create a numerical matrix from all the count data in CircadianRNAseq.csv, by applying the function 'as.numeric' only on the columns which contain numerical data. Note that the function 'as.numeric' returns a vector, which should be assigned back to a matrix form using the function 'matrix'

Calculate the variance and mean for each gene in all the time points

As before, log transform the variance and the mean (Tip: add 1 before applying the log transformation)

To allow binning of the data according to the mean expression levels, sort the vector of the mean in ascending order. Note that the vector of the variance and a vector which holds the original index information should be sorted in parallel, for example:

Index Mean Var

1 15 2.7

2 7 5.4

3 12 18

after sorting it should be:

Index Mean Var

2 7 5.4

3 12 18

1 15 2.7

Use this link to learn about sorting in R: How to Sort Data in R

Bin the mean expression values into 20 groups starting from the minimal value of 3 and ending with the maximal mean expression. Then, calculate the z-score of the variance of each gene, in each bin separately. The z-score will be the variance of each gene minus the mean variance in that

bin, dividing by the standard deviation of the variance in that bin.
sort the resulting z-values in a descending order. Note that a vector which holds the index information should be sorted in parallel.

```
# ============================================================================
# PART 6 - Detecting Genes with Variable Expression Levels
# ============================================================================

print("\n" + "=" * 80)
print("PART 6 - Detecting Genes with Variable Expression Levels")
print("=" * 80)

# Create numerical matrix from count data
count_matrix = circadian_data[time_cols].values.astype(float)

# Calculate variance and mean for each gene
gene_vars = np.var(count_matrix, axis=1)
gene_means = np.mean(count_matrix, axis=1)

# Log transform (add 1 before log)
log_vars = np.log(gene_vars + 1)
log_means = np.log(gene_means + 1)

# Create dataframe for analysis
gene_df = pd.DataFrame({
    'original_idx': range(len(log_means)),
    'log_mean': log_means,
    'log_var': log_vars,
    'gene_symbol': circadian_data['GeneSymbol'].values
})

# Filter genes with log_mean >= 3
gene_df_filtered = gene_df[gene_df['log_mean'] >= 3].copy()
print(f"Number of genes with log(mean+1) >= 3: {len(gene_df_filtered)}")

# Sort by mean expression
gene_df_filtered = gene_df_filtered.sort_values('log_mean').reset_index(drop=True)

# Bin into 20 groups based on mean expression
n_bins = 20
gene_df_filtered['bin'] = pd.qcut(gene_df_filtered['log_mean'], q=n_bins,
labels=False, duplicates='drop')

# Calculate z-score of variance within each bin
gene_df_filtered['z_score'] = gene_df_filtered.groupby('bin')['log_var'].transform(
    lambda x: (x - x.mean()) / x.std())

# Sort by z-score (descending)
gene_df_sorted = gene_df_filtered.sort_values('z_score', ascending=False)

# Get top 40 variable genes
top40_variable = gene_df_sorted.head(40)

print("\nTop 40 genes with highest variance z-scores:")
print("-" * 55)

for i, (idx, row) in enumerate(top40_variable.iterrows()):
    print(f"{i+1:2d}. {row['gene_symbol']:<20} z-score: {row['z_score']:7.3f}")
```

Output:

```
Number of genes with log(mean+1) >= 3: 2530

Top 40 genes with highest variance z-scores:
------------------------------------------------------
 1. CS                    z-score:   4.868
 2. LOC792433             z-score:   4.445
 3. rho                   z-score:   3.766
 4. exorh                 z-score:   3.729
 5. zgc:136930            z-score:   3.549
 6. zgc:194737            z-score:   3.526
 7. nan                   z-score:   3.478
 8. si:ch211-132b12.7     z-score:   3.466
 9. zgc:92061             z-score:   3.380
10. zgc:103748            z-score:   3.340
11. fabp7b                z-score:   3.215
12. regulation of transcription, DNA-dependent  z-score:   3.168
13. olfm1b                z-score:   3.148
14. nan                   z-score:   3.138
15. bhlhb3l               z-score:   3.094
16. nr1d2b                z-score:   3.081
17. cldna                 z-score:   3.068
18. nan                   z-score:   3.040
19. scinlb                z-score:   3.026
20. FOS                   z-score:   3.021
21. aqp3a                 z-score:   3.014
22. Cox7c                 z-score:   2.980
23. Cyp11b2               z-score:   2.963
24. arg2                  z-score:   2.925
25. zgc:162144            z-score:   2.882
26. CFL1                  z-score:   2.849
27. ndrg1l                z-score:   2.805
28. UQCRQ                 z-score:   2.785
29. nan                   z-score:   2.779
30. Camk2n1               z-score:   2.776
31. icn2                  z-score:   2.771
32. per1b                 z-score:   2.763
33. cldnb                 z-score:   2.754
34. zgc:63920             z-score:   2.713
35. Cry3                  z-score:   2.712
36. gria2b                z-score:   2.697
37. si:dkey-7c18.24       z-score:   2.673
38. LOC563601             z-score:   2.666
39. CYP27C1               z-score:   2.654
40. zgc:92061             z-score:   2.636
```

Answer:

The list includes core circadian clock genes (per1b, nr1d2b, Cry3) as well as clock-associated or clock-controlled genes (bhlhb3l, FOS, rho). Circadian genes are expected to appear as variable genes because, by definition, their expression oscillates over the 24-hour cycle, leading to high variance across time points. The amplitude of these oscillations directly increases variance, so genes with strong circadian rhythms naturally show higher-than-expected variability compared to non-rhythmic genes with similar average expression levels.