Neural Data Science

Lecturer: Dr. Jan Lause, Prof. Dr. Philipp Berens

Tutors: Jonas Beck, Fabio Seel, Julius Würzler

Summer term 2025

Student names: *Jannis Ahlert, Julius Grote, Hun Seok Choi*

LLM Disclaimer: *Github Copilot and Chat GPT, used for autompletion and debugging.*

Coding Lab 7 : Transcriptomics

In [65]:
```
import numpy as np
import pylab as plt
import pandas as pd
import matplotlib.pyplot as plt

# We recommend using openTSNE for experiments with t-SNE
# https://github.com/pavlin-policar/openTSNE
from openTSNE import TSNE

%matplotlib inline

%load_ext jupyter_black

%load_ext watermark
%watermark --time --date --timezone --updated --python --iversions --watermark -
```

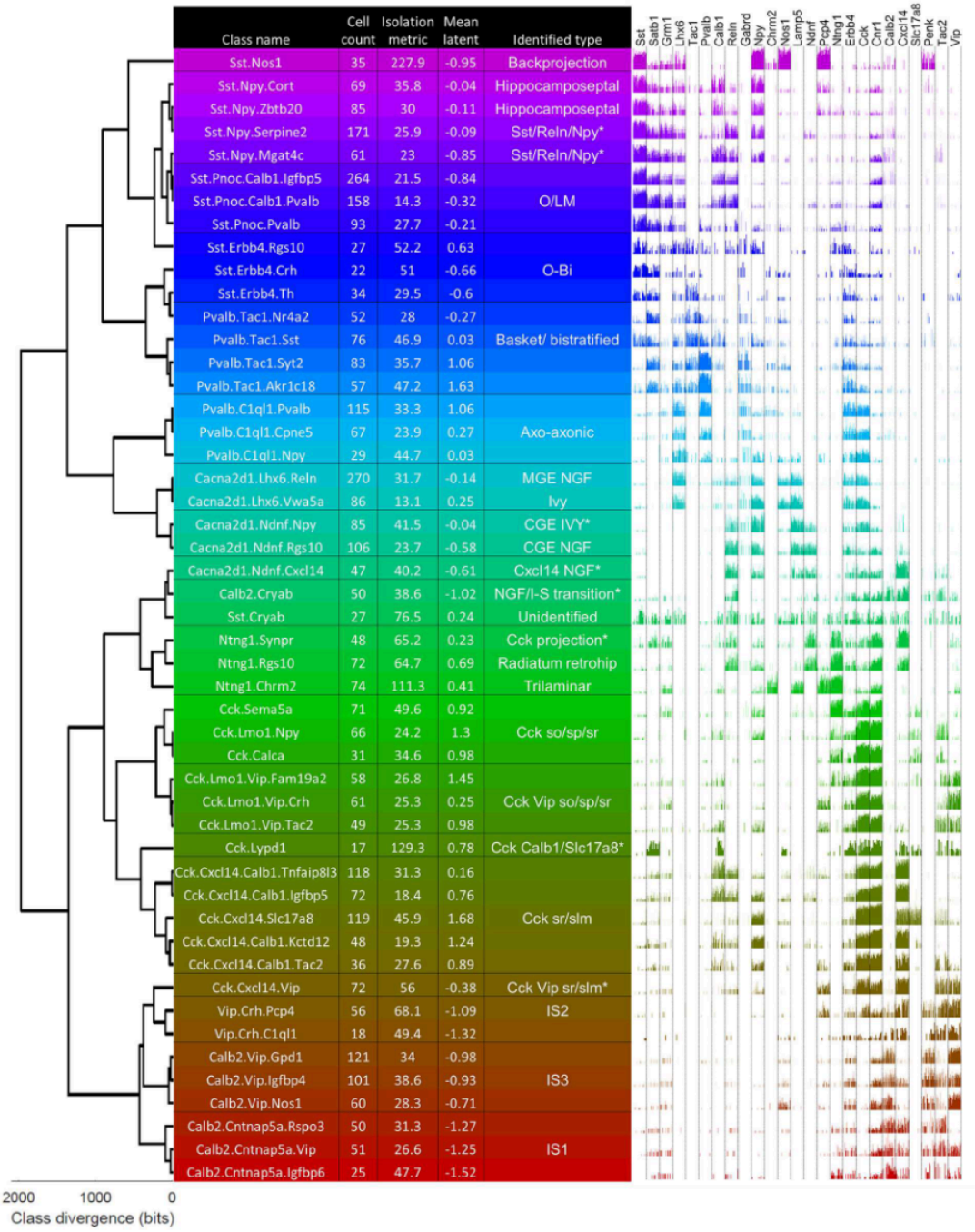
```
The jupyter_black extension is already loaded. To reload it, use:
  %reload_ext jupyter_black
The watermark extension is already loaded. To reload it, use:
  %reload_ext watermark
Last updated: 2025-06-08 20:59:10Mitteleuropäische Sommerzeit

Python implementation: CPython
Python version       : 3.10.11
IPython version      : 8.22.2

sklearn: 1.4.2

openTSNE  : 1.0.2
numpy     : 1.26.4
pandas    : 2.2.1
leidenalg : 0.10.2
sklearn   : 1.4.2
igraph    : 0.11.8
matplotlib: 3.8.3

Watermark: 2.5.0
```

In [66]:
```
plt.style.use("../matplotlib_style.txt")
```

Introduction

In this notebook you are going to work with transcriptomics data, in particular single-cell RNA sequencing (scRNA-seq) data from the paper by Harris et al. (2018). They recorded the transcriptomes of 3,663 inhibitory cells in the hippocampal area CA1. Their analysis divided these cells into 49 fine-scale clusters coresponding to different cell subtypes. They asigned names to these cluster in a hierarchical fashion according to strongly expressed gene in each clusters. The figure below shows the details of their classification.

You will first analyze some of the most relevant statistics of UMI gene counts distributions, and afterwards follow the standard pipeline in the field to produce a visualization of the data.



Load data

Download the data from ILIAS, move it to the `data/` directory and unzip it there. The read counts can be found in `counts`, with rows corresponding to cells and columns to genes. The cluster assignments for every individual cell can be found in `clusters`, along with the colors used in the publication in `clusterColors`.

In [67]:
```
# LOAD HARRIS ET AL DATA

# Load gene counts
data = pd.read_csv("../data/nds_cl_7/harris-data/expression.tsv.gz", sep="\t")
genes = data.values[:, 0]
cells = data.columns[1:-1]
counts = data.values[:, 1:-1].transpose().astype("int")
data = []

# Kick out all genes with all counts = 0
genes = genes[counts.sum(axis=0) > 0]
counts = counts[:, counts.sum(axis=0) > 0]
print(counts.shape)

# Load clustering results
data = pd.read_csv("../data/nds_cl_7/harris-data/analysis_results.tsv", sep="\t"
clusterNames, clusters = np.unique(data.values[0, 1:-1], return_inverse=True)

# Load cluster colors
data = pd.read_csv("../data/nds_cl_7/harris-data/colormap.txt", sep="\s+", heade
clusterColors = data.values

# Note: the color order needs to be reversed to match the publication
clusterColors = clusterColors[::-1]

# Taken from Figure 1 - we need cluster order to get correct color order
clusterOrder = [
    "Sst.No",
    "Sst.Npy.C",
    "Sst.Npy.Z",
    "Sst.Npy.S",
    "Sst.Npy.M",
    "Sst.Pnoc.Calb1.I",
    "Sst.Pnoc.Calb1.P",
    "Sst.Pnoc.P",
    "Sst.Erbb4.R",
    "Sst.Erbb4.C",
    "Sst.Erbb4.T",
    "Pvalb.Tac1.N",
    "Pvalb.Tac1.Ss",
    "Pvalb.Tac1.Sy",
    "Pvalb.Tac1.A",
    "Pvalb.C1ql1.P",
    "Pvalb.C1ql1.C",
    "Pvalb.C1ql1.N",
    "Cacna2d1.Lhx6.R",
    "Cacna2d1.Lhx6.V",
    "Cacna2d1.Ndnf.N",
    "Cacna2d1.Ndnf.R",
    "Cacna2d1.Ndnf.C",
```

```
    "Calb2.Cry",
    "Sst.Cry",
    "Ntng1.S",
    "Ntng1.R",
    "Ntng1.C",
    "Cck.Sema",
    "Cck.Lmo1.N",
    "Cck.Calca",
    "Cck.Lmo1.Vip.F",
    "Cck.Lmo1.Vip.C",
    "Cck.Lmo1.Vip.T",
    "Cck.Ly",
    "Cck.Cxcl14.Calb1.Tn",
    "Cck.Cxcl14.Calb1.I",
    "Cck.Cxcl14.S",
    "Cck.Cxcl14.Calb1.K",
    "Cck.Cxcl14.Calb1.Ta",
    "Cck.Cxcl14.V",
    "Vip.Crh.P",
    "Vip.Crh.C1",
    "Calb2.Vip.G",
    "Calb2.Vip.I",
    "Calb2.Vip.Nos1",
    "Calb2.Cntnap5a.R",
    "Calb2.Cntnap5a.V",
    "Calb2.Cntnap5a.I",
]

reorder = np.zeros(clusterNames.size) * np.nan
for i, c in enumerate(clusterNames):
    for j, k in enumerate(clusterOrder):
        if c[: len(k)] == k:
            reorder[i] = j
            break
clusterColors = clusterColors[reorder.astype(int)]
```

```
(3663, 17965)
```

Task 1: Data inspection

Before we use t-SNE or any other advanced visualization methods on the data, we first want to have a closer look on the data and plot some statistics. For most of the analysis we will compare the data to a Poisson distribution.

1.1. Relationship between expression mean and fraction of zeros

Compute actual and predicted gene expression. The higher the average expression of a gene, the smaller fraction of cells will show a 0 count. Plot the data and explain what you see in the plot.

(3 pts)

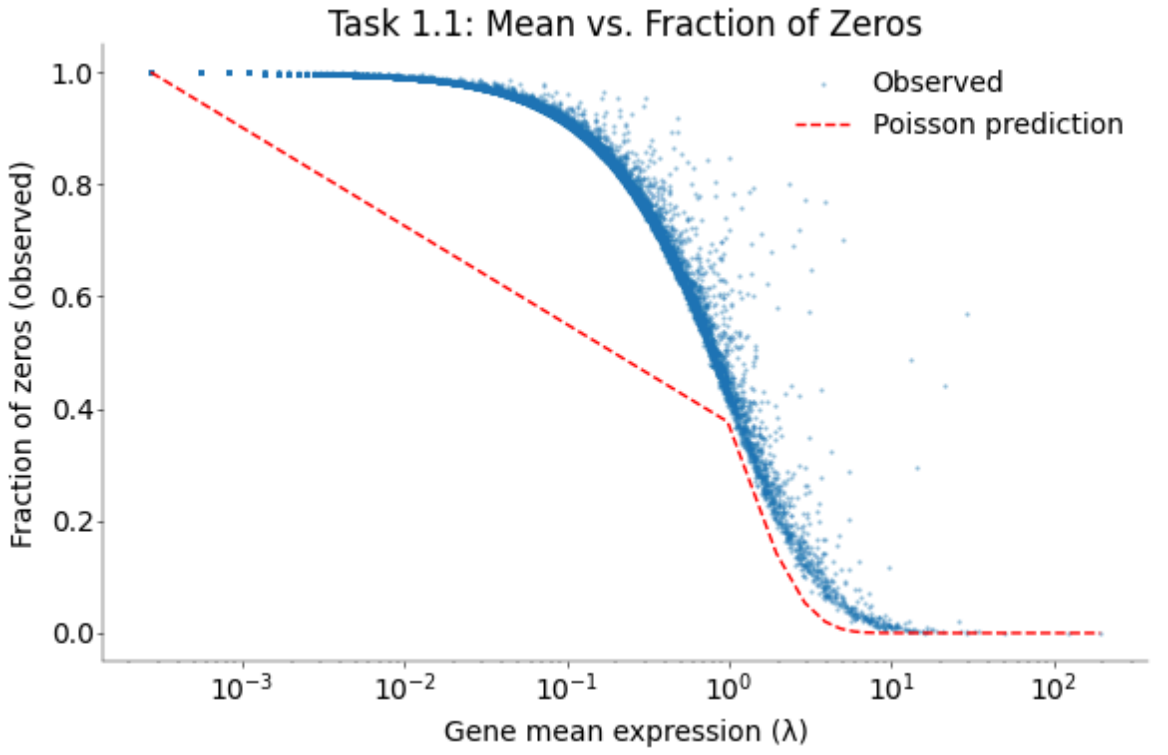
In [68]:
```
# ------------------------------------------------------
# Compute actual and predicted gene expression (1 pt)
```

```
# ------------------------------------------------------

n_cells, n_genes = counts.shape
# Compute the average expression for each gene
gene_means = counts.mean(axis=0)

# Compute the fraction of zeros for each gene
gene_frac_zeros = (counts == 0).sum(axis=0) / n_cells
```

In [69]:
```
# Compute the Poisson prediction
# (what is the expected fraction of zeros in a Poisson distribution with a given
gene_frac_zeros_pred = np.exp(-gene_means)
```

In [70]:
```
# ------------------------------------------------
# plot the data and the Poisson prediction (1 pt)
# ------------------------------------------------

plt.figure(figsize=(6, 4))
plt.scatter(gene_means, gene_frac_zeros, s=1, alpha=0.5, label="Observed")

lambda_vals = np.linspace(gene_means.min(), gene_means.max(), 200)
plt.plot(lambda_vals, np.exp(-lambda_vals), "r--", lw=1, label="Poisson predicti

plt.xlabel("Gene mean expression (λ)")
plt.ylabel("Fraction of zeros (observed)")
plt.title("Task 1.1: Mean vs. Fraction of Zeros")
plt.xscale("log")
plt.legend()
plt.tight_layout()
plt.show()
```

```
C:\Users\Julius\AppData\Local\Temp\ipykernel_21440\1960362292.py:16: UserWarning:
The figure layout has changed to tight
  plt.tight_layout()
```



The plot shows the relationship between the mean expression of each gene (log scale) and the fraction of cells where that gene has zero counts. The red dashed line represents the expected fraction of zeros if the gene counts followed a Poisson distribution ($P(X=0) = e^{-\lambda}$).

We observe that for many genes, especially those with low to moderate mean expression, the observed fraction of zeros is higher than what a Poisson distribution would predict. This phenomenon is known as "zero-inflation" and is common in single-cell RNA sequencing data. It can be caused by biological factors (e.g., stochastic gene expression, true absence of expression in some cells) or technical factors (e.g., low capture efficiency, dropout events where transcripts are not detected even if present). Genes with very high mean expression tend to align more closely with the Poisson prediction, as they are less likely to be affected by dropout.

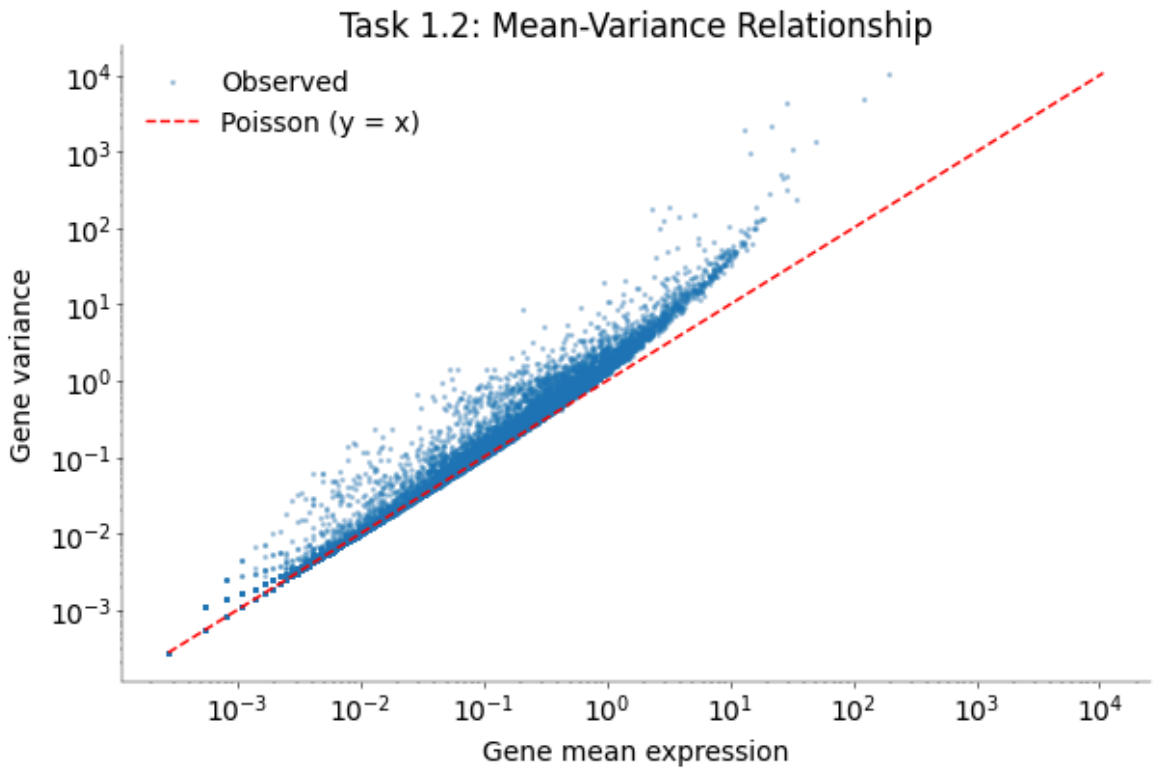
1.2. Mean-variance relationship

If the expression follows Poisson distribution, then the mean should be equal to the variance. Plot the mean-variance relationship and interpret the plot.

(2.5 pts)

In [71]:
```
# ----------------------------------------------------------------
# Compute the variance of the expression counts of each gene (0.5 pt)
# ----------------------------------------------------------------

gene_vars = counts.var(axis=0, ddof=1)  # ddof=1, unbiased
```

In [72]:
```
# ------------------------------------------------------------
# Plot the mean-variance relationship on a log-log plot (1 pt)
# Plot the Poisson prediction as a line
# ------------------------------------------------------------

plt.figure(figsize=(6, 4))
plt.scatter(gene_means, gene_vars, s=5, alpha=0.3, label="Observed")

# y = x
min_val = min(gene_means.min(), gene_vars.min())
max_val = max(gene_means.max(), gene_vars.max())
plt.plot([min_val, max_val], [min_val, max_val], "r--", lw=1, label="Poisson (y


# set log scale
plt.yscale("log")
plt.xscale("log")

plt.xlabel("Gene mean expression")
plt.ylabel("Gene variance")
plt.title("Task 1.2: Mean-Variance Relationship")
plt.legend()
plt.show()
```



This log-log plot compares the mean expression of each gene to its variance. The red dashed line (y=x) represents the expectation for a Poisson distribution, where the variance is equal to the mean.

The observed data points generally lie above the y=x line (especially at higher mean expression levels > 1), indicating that the variance is greater than the mean for most genes. This is a characteristic of "overdispersion" relative to the Poisson model meaning there is more variability in the gene counts than would be expected under a simple Poisson process. This can be due to biological heterogeneity (e.g., different cell states or subtypes expressing genes at different levels) or unmodelled technical variability. Genes with higher mean expression tend to show greater deviation (higher variance) from the Poisson expectation.

1.3. Relationship between the mean and the Fano factor

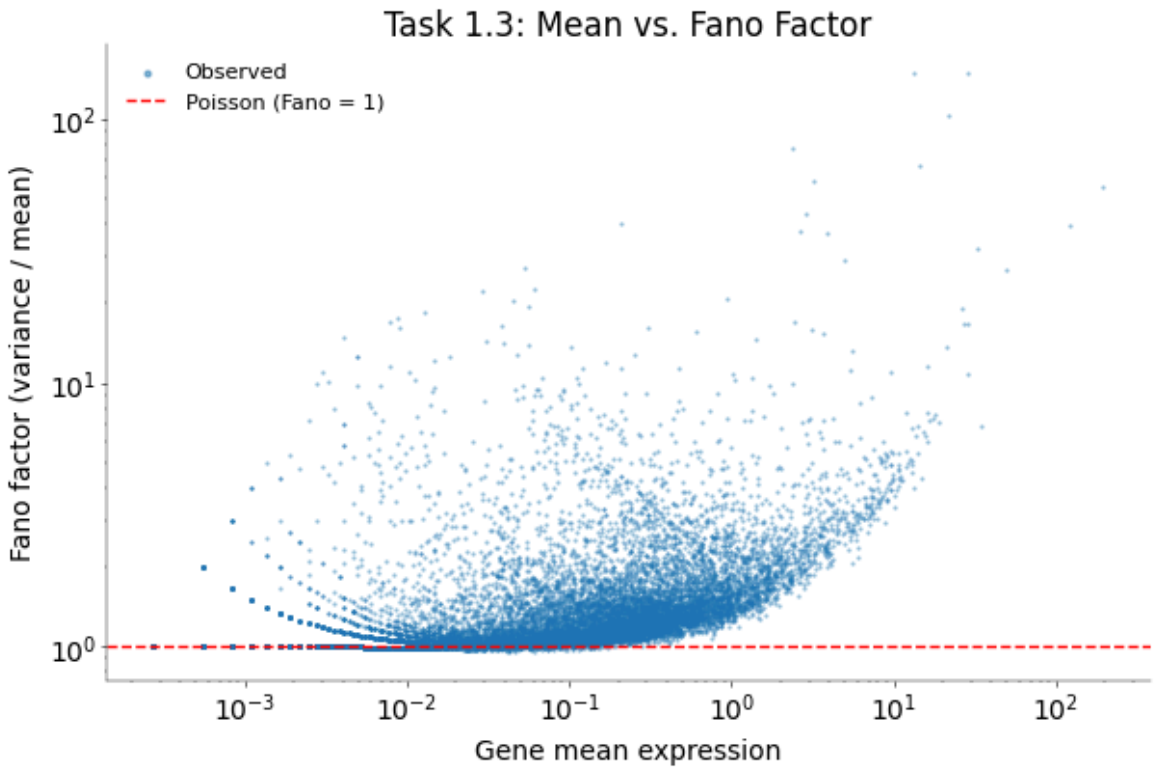
Compute the Fano factor for each gene and make a scatter plot of expression mean vs. Fano factor in log-log coordinates, and interpret what you see in the plot. If the expression follows the Poisson distribution, then the Fano factor (variance/mean) should be equal to 1 for all genes.

(2.5 pts)

In [73]:
```
# ------------------------------------------
# Compute the Fano factor for each gene (0.5 pt)
# ------------------------------------------

fano_factors = gene_vars / gene_means
```

In [74]:
```
# ------------------------------
# plot fano-factor vs mean (1 pt)
```

```
# incl. fano factor
# ------------------------------
# Plot a Poisson prediction as line
# Use the same style of plot as above.

plt.figure(figsize=(6, 4))
plt.scatter(gene_means, fano_factors, s=1, alpha=0.5, label="Observed")

# y = 1 (Poisson baseline)
plt.axhline(1, color="r", linestyle="--", lw=1, label="Poisson (Fano = 1)")

plt.xlabel("Gene mean expression")
plt.ylabel("Fano factor (variance / mean)")
plt.title("Task 1.3: Mean vs. Fano Factor")
# set log scale
plt.xscale("log")
plt.yscale("log")
# set visibility
plt.legend(markerscale=4, loc="best", fontsize=8)
plt.show()
```



This plot shows the Fano factor (variance/mean) for each gene against its mean expression (log scale on the x-axis). For a Poisson distribution, the Fano factor is expected to be 1, indicated by the red dashed line.

Most genes exhibit a Fano factor greater than 1, supporting the observation of overdispersion from the previous plots. This indicates their variance is higher than their mean. The Fano factor is elevated across a range of mean expression levels. Notably, the highest Fano factors are observed for genes with a mean expression around 10 and are lowest at around 0.1. For genes with very moderate mean expression (0.1), the Fano factor tends to decrease and approach 1. A Fano factor greater than 1 suggests that the gene's expression is more variable across cells than would be expected from random

Poisson fluctuations alone, possibly due to biological regulation, transcriptional bursting, or other sources of biological heterogeneity.

Conclusion

Given the observed overdispersion and zero-inflation in the data, it is clear that the simple Poisson model is not sufficient to describe the gene expression distributions. In order to tackle this issue, one can apply a variance-stabilizing transformation, such as the log transformation or the square root transformation (or using analytic Pearson residuals transformation) to the data - reducing the impact of outliers and making the data more suitable for downstream analyses like clustering or dimensionality reduction.

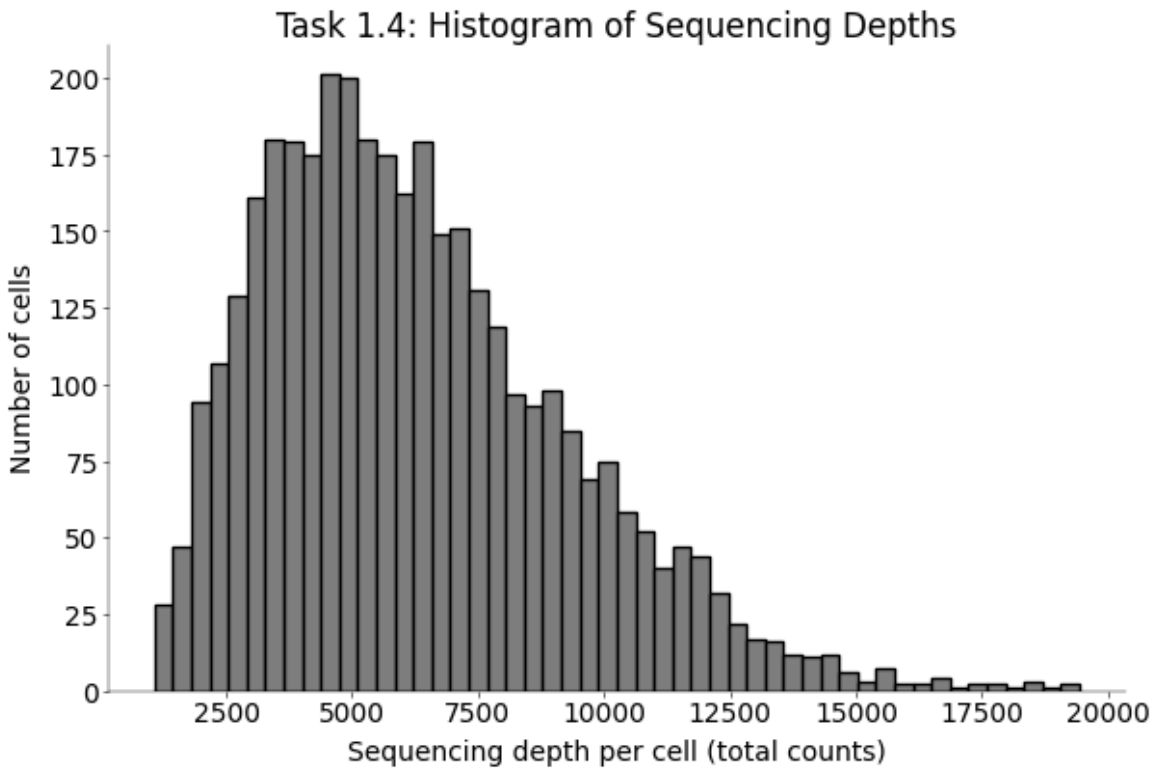
1.4. Histogram of sequencing depths

Different cells have different sequencing depths (sum of counts across all genes) because the efficiency can change from droplet to droplet due to some random expreimental factors. Make a histogram of sequencing depths.

(1.5 pts)

In [75]:
```
# ------------------------------
# Compute sequencing depth (0.5 pt)
# ------------------------------
cell_depths = counts.sum(axis=1)
```

In [76]:
```
# ----------------------------------------
# Plot histogram of sequencing depths (1 pt)
# ----------------------------------------

fig, ax = plt.subplots(figsize=(6, 4))
plt.hist(cell_depths, bins=50, color="gray", edgecolor="black")
plt.xlabel("Sequencing depth per cell (total counts)")
plt.ylabel("Number of cells")
plt.title("Task 1.4: Histogram of Sequencing Depths")
plt.show()
```



1.5. Fano factors after normalization

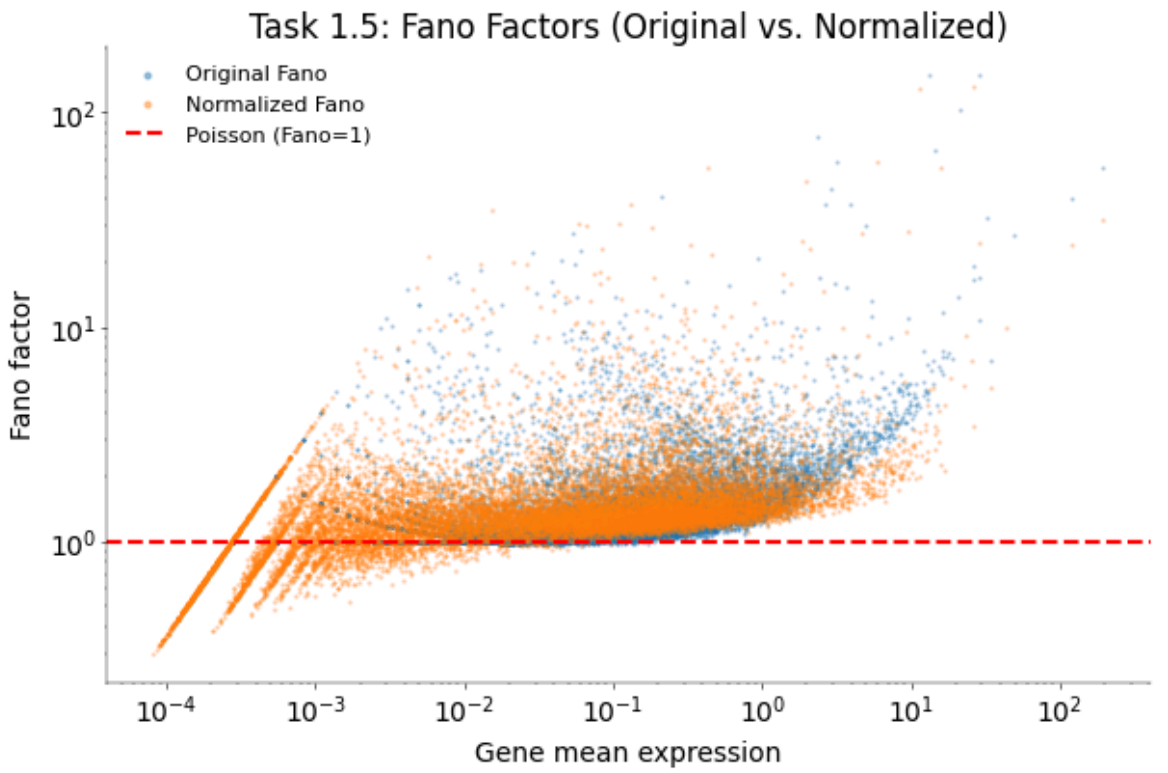
Normalize counts by the sequencing depth of each cell and multiply by the median sequencing depth. Then make the same expression vs Fano factor plot as above. After normalization by sequencing depth, Fano factor should be closer to 1 (i.e. variance even more closely following the mean). This can be used for feature selection.

(2.5 pts)

In [77]:
```
# ------------------------------------------------
# compute normalized counts and fano factor (1 pt)
# ------------------------------------------------
median_depth = np.median(cell_depths)
normalized_counts = counts / cell_depths[:, np.newaxis] * median_depth
norm_gene_means = np.mean(normalized_counts, axis=0)
norm_gene_vars = np.var(normalized_counts, axis=0, ddof=1)
norm_fano_factors = norm_gene_vars / norm_gene_means
```

In [78]:
```
# ------------------------------------------------------------
# plot normalized counts and find the top 10 genes (1 pt)
# hint: keep appropriate axis scaling in mind
# ------------------------------------------------------------

plt.figure(figsize=(6, 4))
plt.scatter(gene_means, fano_factors, s=1, alpha=0.4, label="Original Fano")
plt.scatter(norm_gene_means, norm_fano_factors, s=1, alpha=0.4, label="Normalize
plt.axhline(1, color="r", linestyle="--", lw=1.5, label="Poisson (Fano=1)")


plt.xlabel("Gene mean expression")
plt.ylabel("Fano factor")
plt.xscale("log")
plt.yscale("log")
```

```
plt.title("Task 1.5: Fano Factors (Original vs. Normalized)")
plt.legend(markerscale=4, fontsize=8)
plt.show()
# add plot
```



In [79]:
```
# ----------------------------------------------------------------------
# Find top-10 genes with the highest normalized Fano factor (0.5 pts)
# Print them sorted by the Fano factor starting from the highest
# Gene names are stored in the `genes` array
# ----------------------------------------------------------------------
sorted_idx_desc = np.argsort(-norm_fano_factors)

# choose top-10 genes
top10_idx = sorted_idx_desc[:10]

print("Top 10 genes with highest normalized Fano factor:")
print("-----------------------------------------------")
for i, idx in enumerate(top10_idx, 1):
    gene_name = genes[idx]
    fano_value = norm_fano_factors[idx]
    print(f"{i}. {gene_name}: {fano_value:.4f}")
```

```
Top 10 genes with highest normalized Fano factor:
-----------------------------------------------
1. Sst: 131.1773
2. Npy: 128.4341
3. Vip: 59.1311
4. Cck: 55.6663
5. Cpne2: 55.2354
6. Pcp4: 47.6631
7. Ptpn23: 37.2564
8. Pdzd9: 35.0650
9. Malat1: 31.4312
10. Armc2: 30.5979
```

Task 2: Low dimensional visualization

In this task we will construct a two dimensional visualization of the data. First we will normalize the data with some variance stabilizing transformation and study the effect that different approaches have on the data. Second, we will reduce the dimensionality of the data to a more feasible number of dimensions (e.g. $d = 50$) using PCA. And last, we will project the PCA-reduced data to two dimensions using t-SNE.

2.1. PCA with and without transformations

Here we look at the influence of variance-stabilizing transformations on PCA. We will focus on the following transformations:

- Square root (`sqrt(X)`): it is a variance-stabilizing transformation for the Poisson data.
- Log-transform (`log2(X+1)`): it is also often used in the transcriptomic community.

We will only work with the most important genes. For that, transform the counts into normalized counts (as above) and select all genes with normalized Fano factor above 3 and remove the rest. We will look at the effect that both transformations have in the PCA-projected data by visualizing the first two components. Interpret qualitatively what you see in the plot and compare the different embeddings making use of the ground truth clusters.

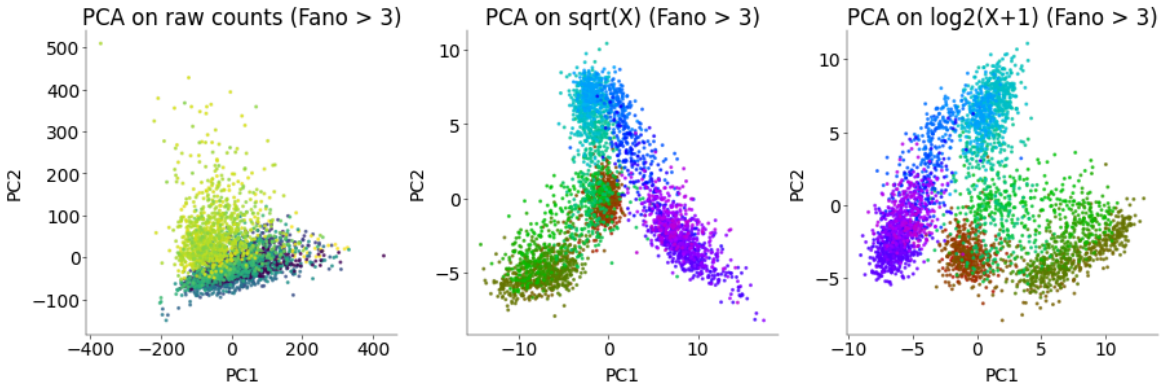
(3.5 pts)

In [80]:
```
# --------------------------------
# Select important genes (0.5 pts)
# --------------------------------
fano_threshold = 3
selected_counts = normalized_counts[:, norm_fano_factors > fano_threshold]
print(
    f"Selected {selected_counts.shape[1]} genes with Fano factor > {fano_thresho
)
```

```
Selected 708 genes with Fano factor > 3 out of 17965 total genes for 3663 cells.
```

In [81]:
```
# ---------------------------------------
# transform data and apply PCA (1 pt)
# ---------------------------------------

sqrt_sub = np.sqrt(selected_counts)  # sqrt-transform
log_sub = np.log2(selected_counts + 1)  # log2(X+1)-transform

from sklearn.decomposition import PCA

# perform PCA
pca_raw = PCA(n_components=2).fit_transform(selected_counts)
pca_sqrt = PCA(n_components=2).fit_transform(sqrt_sub)
pca_log = PCA(n_components=2).fit_transform(log_sub)
```

In [82]:
```
# ------------------------------------------------
# plot first 2 PCs for each transformation (1 pt)
# ------------------------------------------------

fig, axs = plt.subplots(1, 3, figsize=(9, 3))
# add plot
# (4-1) Raw counts PCA
axs[0].scatter(pca_raw[:, 0], pca_raw[:, 1], c=[clusters], s=5, alpha=0.7)
axs[0].set_title("PCA on raw counts (Fano > 3)")
axs[0].set_xlabel("PC1")
axs[0].set_ylabel("PC2")
# (4-2) Sqrt-transformed PCA
axs[1].scatter(
    pca_sqrt[:, 0], pca_sqrt[:, 1], c=clusterColors[clusters], s=5, alpha=0.7
)
axs[1].set_title("PCA on sqrt(X) (Fano > 3)")
axs[1].set_xlabel("PC1")
axs[1].set_ylabel("PC2")

# (4-3) Log-transformed PCA
axs[2].scatter(pca_log[:, 0], pca_log[:, 1], c=clusterColors[clusters], s=5, alp
axs[2].set_title("PCA on log2(X+1) (Fano > 3)")
axs[2].set_xlabel("PC1")
axs[2].set_ylabel("PC2")

plt.show()
```



- **PCA on raw counts (Fano > 3) (left plot):** Even with a more informed gene selection (Fano factor > 3), the PCA on raw counts is challenging to interpret. The data points are spread widely, particularly along PC1. While some broad separation of colors is visible, many ground truth clusters appear heavily mixed and compressed, making it difficult to discern distinct groupings. This is characteristic of raw count data where a few genes with very high expression can dominate the variance.

- **PCA on sqrt(X) (Fano > 3) (middle plot):** The square-root transformation significantly improves the PCA visualization. The spread of data points is more contained, and the scales of PC1 and PC2 are more comparable. There is a noticeable improvement in the separation of the ground truth clusters compared to the raw counts. Different colored groups (e.g., purples, blues, greens) start to form more coherent, albeit still overlapping, regions.

- **PCA on log2(X+1) (Fano > 3) (right plot):** The log2(X+1) transformation appears to provide the clearest visualization of the underlying cluster structure among the three methods. The separation between different ground truth color groups is more pronounced than in the sqrt(X) plot, and the clusters themselves appear somewhat more compact and defined. The overall distribution of points is well-balanced in the 2D space. This transformation is comparably effective at reducing skewness and making the contributions of different genes more comparable. Yet, clusters still overlap significantly.

  → Even in the gene-transformed plots the clusters still overlap, showing that while the transformation helps, the data is still complex and not fully separable by a PCA projection alone.

2.2. tSNE with and without transformations

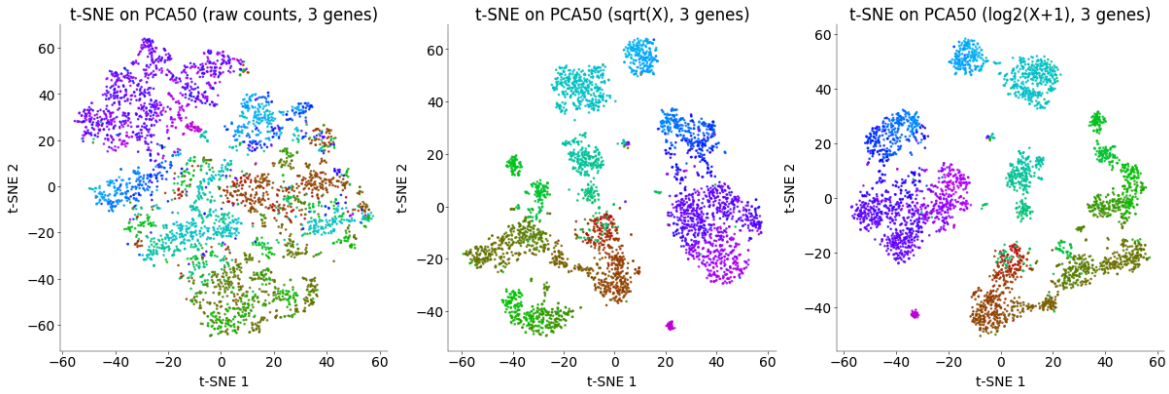
Now, we will reduce the dimensionality of the PCA-reduced data further to two dimensions using t-SNE. We will use only $n = 50$ components of the PCA-projected data. Plot the t-SNE embedding for the three versions of the data and interpret the plots. Do the different transformations have any effect on t-SNE?

(1.5 pts)

In [83]:
```
# ----------------------
# Perform tSNE (0.5 pts)
# ----------------------
from sklearn.manifold import TSNE

n_cells, n_genes_sub = selected_counts.shape
n_pc = min(50, n_cells - 1, n_genes_sub)
if n_pc < 2:
    raise ValueError(
        f"n_cells={n_cells}, n_genes_sub={n_genes_sub}: impossible to take PCA d
    )

# PCA(components =50, 50dimension)
pca_sub_raw = PCA(n_components=n_pc).fit_transform(selected_counts)  # (n_cells,
pca_sub_sqrt = PCA(n_components=n_pc).fit_transform(sqrt_sub)  # (n_cells, n_pc)
pca_sub_log = PCA(n_components=n_pc).fit_transform(log_sub)  # (n_cells, n_pc)

# perform tSNE
tsne_sub_raw = TSNE(n_components=2, random_state=0).fit_transform(pca_sub_raw)
tsne_sub_sqrt = TSNE(n_components=2, random_state=0).fit_transform(pca_sub_sqrt)
tsne_sub_log = TSNE(n_components=2, random_state=0).fit_transform(pca_sub_log)
```

In [84]:
```
# ------------------------------------------------
# plot t-SNE embedding for each dataset (1 pt)
# ------------------------------------------------

fig, axs = plt.subplots(1, 3, figsize=(12, 4))
# add plot
# Raw counts t-SNE
axs[0].scatter(
    tsne_sub_raw[:, 0], tsne_sub_raw[:, 1], c=clusterColors[clusters], s=3, alph
)
```

```
axs[0].set_title(f"t-SNE on PCA{n_pc} (raw counts, 3 genes)")
axs[0].set_xlabel("t-SNE 1")
axs[0].set_ylabel("t-SNE 2")

# (5-2) sqrt(X) 기반 t-SNE
axs[1].scatter(
    tsne_sub_sqrt[:, 0], tsne_sub_sqrt[:, 1], c=clusterColors[clusters], s=3, al
)
axs[1].set_title(f"t-SNE on PCA{n_pc} (sqrt(X), 3 genes)")
axs[1].set_xlabel("t-SNE 1")
axs[1].set_ylabel("t-SNE 2")

# (5-3) log2(X+1) 기반 t-SNE
axs[2].scatter(
    tsne_sub_log[:, 0], tsne_sub_log[:, 1], c=clusterColors[clusters], s=3, alph
)
axs[2].set_title(f"t-SNE on PCA{n_pc} (log2(X+1), 3 genes)")
axs[2].set_xlabel("t-SNE 1")
axs[2].set_ylabel("t-SNE 2")

plt.show()
```



Effect of the different transformations on t-SNE:

Both transformations increase the separation of clusters dramatically. Between them there are small differences in how the greenish and the blueish points are clustered with the log2(X+1) transformation showing a more distinct clustering overall.

2.3. Leiden clustering

Now we will play around with some clustering and see whether the clustering methods can produce similar results to the original clusters from the publication. We will apply Leiden clustering (closely related to the Louvain clustering), which is standard in the field and works well even for very large datasets.

Choose one representation of the data (best transformation based in your results from the previous task) to use further in this task and justify your choice. Think about which level of dimensionality would be sensible to use to perform clustering. Visualize in the two-dimensional embedding the resulting clusters and compare to the original clusters.

(1.5 pts)

In [85]:
```
# To run this code you need to install leidenalg and igraph
# conda install -c conda-forge python-igraph leidenalg

import igraph as ig
from sklearn.neighbors import NearestNeighbors, kneighbors_graph
import leidenalg as la
import matplotlib.cm as cm
```

In [86]:
```
# Define some contrast colors

clusterCols = [
    "#FFFF00",
    "#1CE6FF",
    "#FF34FF",
    "#FF4A46",
    "#008941",
    "#006FA6",
    "#A30059",
    "#FFDBE5",
    "#7A4900",
    "#0000A6",
    "#63FFAC",
    "#B79762",
    "#004D43",
    "#8FB0FF",
    "#997D87",
    "#5A0007",
    "#809693",
    "#FEFFE6",
    "#1B4400",
    "#4FC601",
    "#3B5DFF",
    "#4A3B53",
    "#FF2F80",
    "#61615A",
    "#BA0900",
    "#6B7900",
    "#00C2A0",
    "#FFAA92",
    "#FF90C9",
    "#B903AA",
    "#D16100",
    "#DDEFFF",
    "#000035",
    "#7B4F4B",
    "#A1C299",
    "#300018",
    "#0AA6D8",
    "#013349",
    "#00846F",
    "#372101",
    "#FFB500",
    "#C2FFED",
    "#A079BF",
    "#CC0744",
    "#C0B9B2",
    "#C2FF99",
    "#001E09",
    "#00489C",
```

```
    "#6F0062",
    "#0CBD66",
    "#EEC3FF",
    "#456D75",
    "#B77B68",
    "#7A87A1",
    "#788D66",
    "#885578",
    "#FAD09F",
    "#FF8A9A",
    "#D157A0",
    "#BEC459",
    "#456648",
    "#0086ED",
    "#886F4C",
    "#34362D",
    "#B4A8BD",
    "#00A6AA",
    "#452C2C",
    "#636375",
    "#A3C8C9",
    "#FF913F",
    "#938A81",
    "#575329",
    "#00FECF",
    "#B05B6F",
    "#8CD0FF",
    "#3B9700",
    "#04F757",
    "#C8A1A1",
    "#1E6E00",
    "#7900D7",
    "#A77500",
    "#6367A9",
    "#A05837",
    "#6B002C",
    "#772600",
    "#D790FF",
    "#9B9700",
    "#549E79",
    "#FFF69F",
    "#201625",
    "#72418F",
    "#BC23FF",
    "#99ADC0",
    "#3A2465",
    "#922329",
    "#5B4534",
    "#FDE8DC",
    "#404E55",
    "#0089A3",
    "#CB7E98",
    "#A4E804",
    "#324E72",
    "#6A3A4C",
    "#83AB58",
    "#001C1E",
    "#D1F7CE",
    "#004B28",
    "#C8D0F6",
```

```
    "#A3A489",
    "#806C66",
    "#222800",
    "#BF5650",
    "#E83000",
    "#66796D",
    "#DA007C",
    "#FF1A59",
    "#8ADBB4",
    "#1E0200",
    "#5B4E51",
    "#C895C5",
    "#320033",
    "#FF6832",
    "#66E1D3",
    "#CFCDAC",
    "#D0AC94",
    "#7ED379",
    "#012C58",
]

clusterCols = np.array(clusterCols)
```

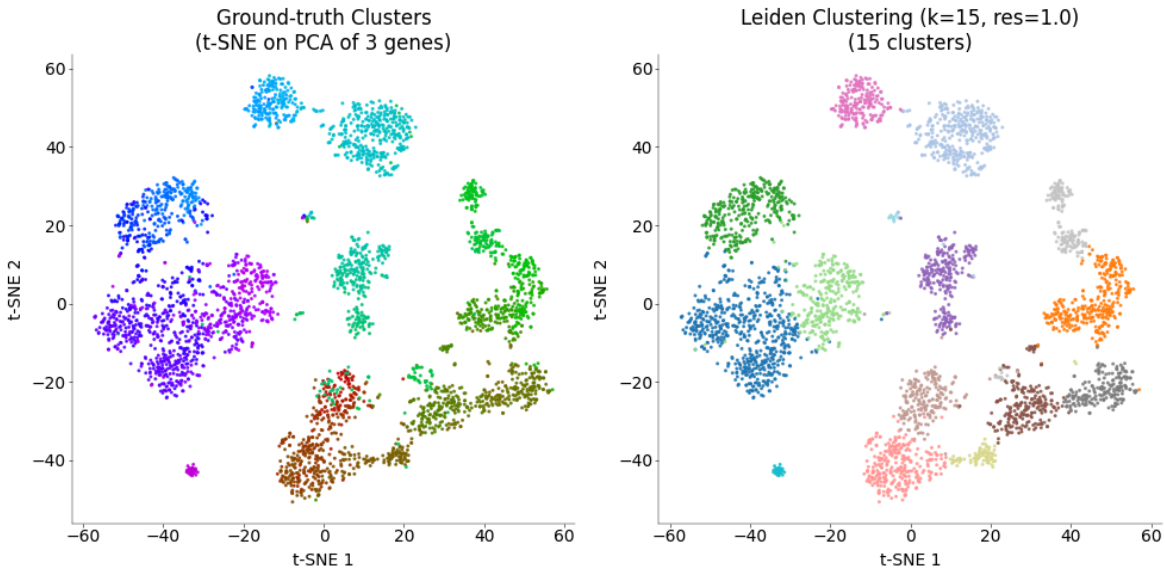
In [87]:
```
# ----------------------------------------------------
# create graph and run leiden clustering on it (0.5 pts)
# hint: use `la?`, `la.find_partition?` and `ig.Graph?`
# to find out more about the provided packages.
# ----------------------------------------------------

# Construct kNN graph with k=15

X_use = pca_sub_log.copy()  # PCA representation (n_cells × n_pc)
tsne_use = tsne_sub_log.copy()

max_neighbors = n_cells - 1  # set n_neighbors not exceeding maximum number of c
k = min(15, max_neighbors)

A = kneighbors_graph(X_use, n_neighbors=k, mode="connectivity", include_self=Fal

A_bool = (A.toarray() > 0).astype(int)

# Transform it into an igraph object
G = ig.Graph.Adjacency((A_bool > 0).tolist())
G.to_undirected()

# Run Leiden clustering
partition = la.find_partition(
    G, la.RBConfigurationVertexPartition, resolution_parameter=1.0
)
leiden_labels = np.array(partition.membership)  # shape = (n_cells,)
n_leiden = len(np.unique(leiden_labels))
print(f"Leiden clustering (resolution=1.0) → {n_leiden} clusters found")

# you can use `la.RBConfigurationVertexPartition` as the partition type
```

```
Leiden clustering (resolution=1.0) → 15 clusters found
```

In [88]:
```
# -------------------------
# Plot the results (1 pt)
# -------------------------
```

```
fig, axes = plt.subplots(1, 2, figsize=(10, 5))

# Ground-truth clusters
axes[0].scatter(
    tsne_use[:, 0], tsne_use[:, 1], c=clusterColors[clusters], s=5, alpha=0.8
)
axes[0].set_title("Ground-truth Clusters\n(t-SNE on PCA of 3 genes)")
axes[0].set_xlabel("t-SNE 1")
axes[0].set_ylabel("t-SNE 2")

# Leiden clusters
cmap_leiden = cm.get_cmap("tab20", n_leiden)
colors_leiden = cmap_leiden(leiden_labels)

axes[1].scatter(tsne_use[:, 0], tsne_use[:, 1], c=colors_leiden, s=5, alpha=0.8)
axes[1].set_title(f"Leiden Clustering (k={k}, res=1.0)\n({n_leiden} clusters)")
axes[1].set_xlabel("t-SNE 1")
axes[1].set_ylabel("t-SNE 2")

plt.tight_layout()
plt.show()
```

```
C:\Users\Julius\AppData\Local\Temp\ipykernel_21440\3385153839.py:16: MatplotlibDe
precationWarning: The get_cmap function was deprecated in Matplotlib 3.7 and will
be removed two minor releases later. Use ``matplotlib.colormaps[name]`` or ``matp
lotlib.colormaps.get_cmap(obj)`` instead.
  cmap_leiden = cm.get_cmap("tab20", n_leiden)
C:\Users\Julius\AppData\Local\Temp\ipykernel_21440\3385153839.py:24: UserWarning:
The figure layout has changed to tight
  plt.tight_layout()
```



2.4. Change the clustering resolution

The number of clusters can be changed by modifying the resolution parameter. How many clusters did we get with the default value? Change the resolution parameter to yield 2x more and 2x fewer clusters Plot all three results as t-SNE overlays (same as above).

(1.5 pts)

In [89]:
```
# ------------------------------------------------------------------
# run the clustering for 3 different resolution parameters (0.5 pts)
# ------------------------------------------------------------------

resolutions = [0.5, 1.0, 2.0]
results = {}

for res in resolutions:
    partition_res = la.find_partition(
        G, la.RBConfigurationVertexPartition, resolution_parameter=res
    )
    labels_res = np.array(partition_res.membership)  # shape = (n_cells,)
    results[res] = labels_res
    print(f"Resolution {res:.1f} → {len(np.unique(labels_res))} clusters")
```

```
Resolution 0.5 → 11 clusters
Resolution 1.0 → 14 clusters
Resolution 2.0 → 21 clusters
```

In [90]:
```
# --------------------------
# Plot the results (1 pt)
# --------------------------

fig, axes = plt.subplots(1, 3, figsize=(15, 5))

for ax, res in zip(axes, resolutions):
    labels_res = results[res]
    n_res = len(np.unique(labels_res))
    cmap_res = cm.get_cmap("tab20", n_res)
    colors_res = cmap_res(labels_res)

    ax.scatter(tsne_use[:, 0], tsne_use[:, 1], c=colors_res, s=5, alpha=0.8)
    ax.set_title(f"Leiden (res={res:.1f}, {n_res} clusters)")
    ax.set_xlabel("t-SNE 1")
    ax.set_ylabel("t-SNE 2")

plt.show()
```

```
C:\Users\Julius\AppData\Local\Temp\ipykernel_21440\1805275859.py:10: MatplotlibDeprecationWarning: The get_cmap function was deprecated in Matplotlib 3.7 and will be removed two minor releases later. Use ``matplotlib.colormaps[name]`` or ``matplotlib.colormaps.get_cmap(obj)`` instead.
  cmap_res = cm.get_cmap("tab20", n_res)
```