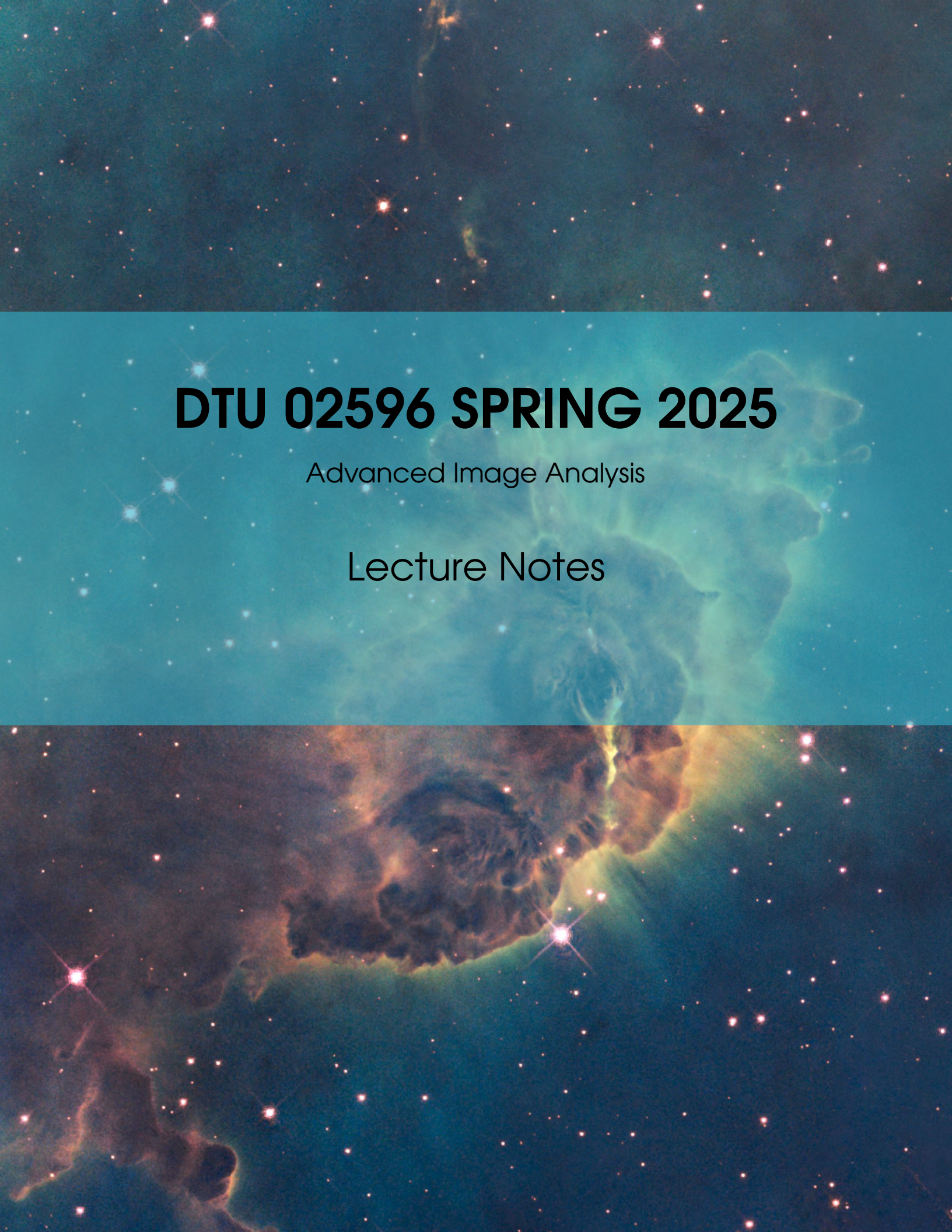


DTU 02596 SPRING 2025

Advanced Image Analysis

Lecture Notes

1. Lecture I: Introduction

Advanced Image Analysis: Introduction and Fundamental Operations

High-Level Overview

This topic introduces fundamental concepts in advanced image analysis, serving as the foundation for the 02506 Advanced Image Analysis course at DTU. The material focuses on mathematical representations of images, signal processing techniques, and quantification methods critical for analyzing various image types. The introduction covers how images are modeled mathematically as functions (e.g., a grayscale image as $I(\boldsymbol{x},\boldsymbol{y}) : \Omega \subset \mathbb{R}^2 \to \mathbb{R}$) and extends to volumetric and spectral images. The practical exercises emphasize implementing core operations like convolution with Gaussian kernels, segmentation boundary computation, and curve smoothing, teaching students how to translate mathematical descriptions into efficient code while developing a deeper understanding of theoretical concepts.

Key Theoretical Concepts

1. Image Representation

- **Definition:** An image is a regularly sampled signal typically representing light intensity, mathematically modeled as a function $I(\boldsymbol{x},\boldsymbol{y})$ with $I : \Omega \subset \mathbb{R}^2 \to \mathbb{R}$ for grayscale images.
- **Properties:** In computer programs, images are represented as 2D arrays with indexing implicitly related to image space. Origin placement and indexing (0 or 1-based) varies depending on context.
- **Extensions:**
  - 3D images (volumes): $I : \Omega \subset \mathbb{R}^3 \to \mathbb{R}$
  - Spectral images: $I : \Omega \subset \mathbb{R}^2 \to \mathbb{R}^n$ where n is the number of spectral bands
  - Movies: $I(x,y,t)$ where t is the time dimension

2. Image Convolution

- **Definition:** Convolution between functions f and g is defined as

$$(f * g)(x) = \int_{-\infty}^{\infty} f(x-\tau)g(\tau)\,d\tau$$

in continuous space, and in 2D discrete form as

$$(f*g)(x,y) = \sum_{i=-l}^{l} \sum_{j=-l}^{l} f(x-i,y-j)g(i,j)$$

- **Properties:**
  - Commutative: $f*g = g*f$
  - Associative: $f*(g*h) = (f*g)*h$
- **Gaussian Kernel Properties:**
  - Defined as

$$g(x;t) = \frac{1}{\sqrt{2\pi t}} e^{-\frac{x^2}{2t}} \tag{1.1}$$

in 1D, and

$$g(x,y;t) = \frac{1}{2\pi t} e^{-\frac{x^2+y^2}{2t}} \tag{1.2}$$

in 2D.

■ **Example 1.1** For $t = 1$, the 1D Gaussian becomes

$$g(x;1) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$$

which is the standard normal distribution curve. ■

  - **Separability:** Convolving with two orthogonal 1D Gaussians yields the same result as convolving with a 2D Gaussian.

■ **Example 1.2** To blur an image with a 2D Gaussian of variance t, you can:
  * First convolve each row with $g(x;t)$
  * Then convolve each column with $g(y;t)$

This reduces computation from $O(n^2)$ to $O(2n)$. ■

  - **Semi-group structure:**

$$g(x,y;t_1+t_2)*I(x,y) = g(x,y;t_1)*g(x,y;t_2)*I(x,y) \tag{1.3}$$

■ **Example 1.3** Instead of convolving with $g(x,y;4)$, you can convolve with $g(x,y;2)$ twice:

$$g(x,y;4)*I = g(x,y;2)*g(x,y;2)*I$$

■

  - **Derivative property:**

$$\frac{\partial}{\partial x}(I*g) = \frac{\partial I}{\partial x}*g = I*\frac{\partial g}{\partial x} \tag{1.4}$$

■ **Example 1.4** To compute the image gradient along the x-direction:
  * Compute the derivative of the Gaussian: $\frac{\partial g}{\partial x}$
  * Convolve it with the image: $I*\frac{\partial g}{\partial x}$

This is commonly used for edge detection in Sobel-like filters. ■

3. Segmentation Boundary Length

- **Definition:** For a segmentation $S : \Omega \to \{1, 2, \ldots, n\}$, the boundary length is defined as

$$L(S) = \sum_{(x,y)\sim(x',y')} d(S(x,y), S(x',y'))$$

  where $(x,y) \sim (x',y')$ indicates neighboring pixels and d is a metric that equals 0 if arguments are equal and 1 otherwise.
- **Implementation:** The boundary length counts all occurrences of neighboring pixels having different labels, using 4-connectivity (left, right, up, down).

4. Curve Smoothing

- **Definition:** For a curve represented as a sequence of N points in an $N \times 2$ matrix X, smoothing can be achieved by displacing points toward the average of their neighbors.
- **Properties:**
  - **Explicit smoothing:**

$$X_{\text{new}} = (I + \lambda L)X$$

    where L is a Laplacian matrix
  - **Implicit smoothing:**

$$X_{\text{new}} = (I - \lambda L)^{-1}X$$

    to avoid oscillations with large λ
  - **Extended kernel:**

$$\alpha[0 \;\; 1 \;\; -2 \;\; 1 \;\; 0] + \beta[-1 \;\; 4 \;\; -6 \;\; 4 \;\; -1]$$

    combines elasticity (length minimizing) and rigidity (curvature minimizing) terms

Implementation Walk-Through

1. Gaussian Kernel Generation

Purpose: Creates 1D Gaussian kernels and their derivatives for image processing operations.

```
def get_gaussian_kernels(s):
    t = s**2
    r = np.ceil(4 * s)
    x = np.arange(-r, r + 1).reshape(-1, 1)
    g = np.exp(-x**2 / (2 * t))
    g = g/np.sum(g)
    dg = -x * g / t
    return g, dg, x
```

Algorithm:

1. Calculate variance (t) from standard deviation (s)
2. Define the kernel radius (r) as approximately 4σ
3. Create array of x coordinates centered around 0
4. Calculate Gaussian values using the formula
5. Normalize the kernel to ensure the sum of all values equals 1
6. Calculate the derivative of the Gaussian
7. Return the Gaussian kernel, its derivative, and x values

2. Boundary Length Calculation

Purpose: Computes the total length of boundaries between different segments in a labeled image.

Python code:

```
def boundary_length(im):
    return (im[:-1] != im[1:]).sum() + (im[:, :-1] != im[:, 1:]).sum()
```

Algorithm:

1. Compare each pixel with its neighbor below (vertical edges) using array slicing
2. Compare each pixel with its neighbor to the right (horizontal edges) using array slicing
3. Count total number of difference occurrences (where neighboring pixels have different values)
4. Sum the results from both directions to get the total boundary length

Complexity: $\mathcal{O}(n)$ where n is the number of pixels in the image

Pitfalls:

- Ensure each pixel pair is counted only once
- Use vectorized operations instead of loops for efficiency

3. Curve Smoothing with Regularization Matrix

Purpose: Creates a matrix that applies both elasticity and rigidity constraints for smoothing curves.

Python code:

```
def regularization_matrix(N, alpha, beta):
    s = np.zeros(N)
    s[[-2, -1, 0, 1, 2]] = (alpha * np.array([0, 1, -2, 1, 0]) +
                            beta * np.array([-1, 4, -6, 4, -1]))
    S = scipy.linalg.circulant(s)
    return scipy.linalg.inv(np.eye(N) - S)
```

Algorithm:

1. Create a zero array of length N
2. Fill specific positions with coefficients for combined elasticity (first derivative) and rigidity (second derivative) kernels
3. Generate a circulant matrix from this array (each row shifted right from the previous)
4. Calculate the inverse of $(I - S)$ for implicit smoothing
5. Return the final matrix that can be applied to point coordinates

Pitfalls:

- Matrix inversion can be unstable for large N or extreme parameter values
- Ensure α and β are reasonable

Tips and Practical Notes

- **Gaussian Convolution:**
  - The σ and t parameters are related by $t = \sigma^2$ – ensure correct usage
  - Gaussian kernels should be normalized to sum to 1
  - Truncate Gaussian kernels at approximately 3–5σ for balance between accuracy and efficiency
  - Use separable property correctly by applying 1D kernels in orthogonal directions
- **Segmentation Boundary:**
  - Ensure each boundary pixel is counted exactly once
  - The metric function must return 0 for equal values and 1 otherwise
  - 4-connectivity means only horizontal and vertical neighbors are considered
- **Curve Smoothing:**
  - Explicit smoothing $(I + \lambda L)$ can cause oscillations with large λ
  - Implicit smoothing $(I - \lambda L)^{-1}$ avoids oscillations but requires matrix inversion
  - Curve shrinkage is a common side effect of length-minimizing smoothing
  - α (elasticity) and β (rigidity) control the smoothing behavior

Worked Example: Gaussian Kernel Properties

Let's verify the **separability property** of Gaussian kernels:

1. **Create a 1D Gaussian kernel with $\sigma = 4.5$:**

```
s = 4.5
g, dg, x = get_gaussian_kernels(s)
```

2. **Create a 2D Gaussian kernel by outer product:**

```
g2D = g @ g.T
```

3. **Apply the 2D kernel to an image:**

```
im_2g = convolve(im, g2D)
```

4. **Apply two successive 1D kernels in orthogonal directions:**

```
im_g = convolve(convolve(im, g), g.T)
```

5. **Compare the results:**

```
difference = im_g - im_2g
mean_abs_diff = np.abs(difference).mean()
```

Conclusion: The mean absolute difference is very small (near zero), confirming that convolving with a 2D Gaussian is equivalent to convolving with two 1D Gaussians in orthogonal directions.

Cheat-Sheet Section

- **Image representation:** $I(x,y) : \Omega \subset \mathbb{R}^2 \to \mathbb{R}$ for grayscale, $I : \Omega \subset \mathbb{R}^2 \to \mathbb{R}^n$ for n spectral bands
- **Gaussian kernel:** $g(x,t) = \frac{1}{\sqrt{2\pi t}} \cdot e^{-\frac{x^2}{2t}}$, where $t = \sigma^2$
- **Gaussian derivative:** $\frac{dg}{dx} = -\frac{x}{t} \cdot g(x,t)$
- **Convolution properties:**
  - Commutativity: $f * g = g * f$
  - Associativity: $f * (g * h) = (f * g) * h$
  - Derivative: $\frac{\partial}{\partial x}(I * g) = I * \frac{\partial g}{\partial x}$
- **Boundary length:** $L = \sum(\text{pixel pairs})\ [1 \text{ if values differ}, 0 \text{ otherwise}]$
- **Curve smoothing:**
  - Explicit: $X_{\text{new}} = (I + \lambda L)X$
  - Implicit: $X_{\text{new}} = (I - \lambda L)^{-1}X$
  - Combined: $X_{\text{new}} = (I - \alpha A - \beta B)^{-1}X$

Reflective Questions

1. **Question:** How does the number of convolutions with a small Gaussian kernel (variance t) relate to a single convolution with a larger Gaussian (variance T)? What property does this demonstrate?
   Answer: N convolutions with a Gaussian of variance t is equivalent to one convolution with a Gaussian of variance Nt, demonstrating the *semi-group property* of Gaussian convolution.
2. **Question:** In the context of curve smoothing, why might we prefer implicit smoothing $(I - \lambda L)^{-1}X$ over explicit smoothing $(I + \lambda L)X$?
   Answer: Implicit smoothing allows using larger λ values without causing oscillations, achieving significant smoothing in a single step rather than requiring many iterations with small λ in explicit smoothing.
3. **Question:** When computing image derivatives, why is it beneficial to use the derivative of a Gaussian rather than first smoothing with a Gaussian and then applying a finite difference operator?
   Answer: Using the derivative of a Gaussian combines smoothing and differentiation in one operation, leveraging the property

   $$\frac{\partial}{\partial x}(I * g) = I * \frac{\partial g}{\partial x}$$

   This is more efficient and often produces better results than separate operations.
4. **Question:** How do the α and β parameters in the extended smoothing kernel affect the behavior of curve smoothing? What happens if β is large and α is small?
   Answer: α controls elasticity (length minimization) while β controls rigidity (curvature minimization). When β is large and α is small, the curve will prioritize minimizing curvature over length, resulting in smoother corners but possibly longer overall length.
5. **Question:** Why is vectorization beneficial when computing the boundary length of a segmentation, and how does the implementation achieve this?
   Answer: Vectorization improves efficiency by avoiding loops, leveraging NumPy's optimized array operations. The implementation achieves this by using array slicing to compare all ver-

tical edges (`im[:-1] != im[1:]`) and all horizontal edges (`im[:, :-1] != im[:, 1:]`) simultaneously, then summing the results.

1.1 Examples

Example 1: Gaussian Kernel Generation

Given: Standard deviation $\sigma = 2$, compute the 1D Gaussian kernel values.

■ **Example 1.5** **Step 1:** Calculate variance and kernel radius

- Variance: $t = \sigma^2 = 2^2 = 4$
- Kernel radius: $r = \lceil 4 \cdot \sigma \rceil = \lceil 4 \cdot 2 \rceil = 8$

Step 2: Create coordinate array $x = [-8, -7, \ldots, 0, \ldots, 7, 8]$

Step 3: Calculate unnormalized Gaussian values

$$g(x) = e^{-x^2/(2t)} = e^{-x^2/8}$$

For selected points:

- $g(-2) = e^{-4/8} = e^{-0.5} \approx 0.6065$
- $g(-1) = e^{-1/8} = e^{-0.125} \approx 0.8825$
- $g(0) = e^{0} = 1.0000$
- $g(1) = e^{-1/8} \approx 0.8825$
- $g(2) = e^{-4/8} \approx 0.6065$

Step 4: Normalize kernel. Sum of all values ≈ 8.0000. Normalized values:

- $g(-2) = \frac{0.6065}{8.0000} \approx 0.0758$
- $g(-1) = \frac{0.8825}{8.0000} \approx 0.1103$
- $g(0) = \frac{1.0000}{8.0000} = 0.1250$
- $g(1) \approx 0.1103$
- $g(2) \approx 0.0758$

Result: Normalized kernel sums to 1.0 and is symmetric around 0. ■

Example 2: Gaussian Derivative Calculation

Given: $\sigma = 1.5$, compute the derivative of the Gaussian at specific points.

■ **Example 1.6** **Step 1:** Calculate parameters

- Variance: $t = \sigma^2 = 1.5^2 = 2.25$

Step 2: Use derivative formula:

$$\frac{dg}{dx} = -\frac{x}{t} \cdot g(x)$$

Step 3: Calculate at specific points

At $x = -1$:

- $g(-1) = e^{-1/4.5} = e^{-0.222} \approx 0.8007$
- $\frac{dg}{dx}\big|_{x=-1} = -\frac{-1}{2.25} \cdot 0.8007 = \frac{0.8007}{2.25} \approx 0.3559$

At $x = 0$:

- $g(0) = 1.0000$
- $\frac{dg}{dx}\big|_{x=0} = -\frac{0}{2.25} \cdot 1.0000 = 0$

At $x = 1$:

- $g(1) = e^{-1/4.5} \approx 0.8007$
- $\frac{dg}{dx}|_{x=1} = -\frac{1}{2.25} \cdot 0.8007 \approx -0.3559$

Result: Derivative is zero at the center and antisymmetric around 0. ■

Example 3: 2D Convolution with Separability

Given: 3×3 image and 1D Gaussian kernel, demonstrate separable convolution.

■ **Example 1.7 Image:**

$$I = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

1D Gaussian kernel ($\sigma = 1$): $g = [0.25, 0.50, 0.25]$ (simplified for clarity)
Step 1: Convolve rows with horizontal kernel
Row 1: $[1,2,3] * [0.25, 0.50, 0.25]$
- Position 1: $1 \cdot 0.50 + 2 \cdot 0.25 = 0.50 + 0.50 = 1.00$
- Position 2: $1 \cdot 0.25 + 2 \cdot 0.50 + 3 \cdot 0.25 = 0.25 + 1.00 + 0.75 = 2.00$
- Position 3: $2 \cdot 0.25 + 3 \cdot 0.50 = 0.50 + 1.50 = 2.00$

Row 1 result: $[1.00, 2.00, 2.00]$
Similarly for other rows:
- Row 2 result: $[4.00, 5.00, 5.00]$
- Row 3 result: $[7.00, 8.00, 8.00]$

Step 2: Convolve columns with vertical kernel
Final result after vertical convolution:

$$\text{Result} = \begin{bmatrix} 2.50 & 3.50 & 3.50 \\ 5.00 & 6.00 & 6.00 \\ 5.50 & 6.50 & 6.50 \end{bmatrix}$$

Verification: This equals the result of convolving with the 2D kernel $g \otimes g^\top$. ■

Example 4: Semi-Group Property Verification

Given: Demonstrate that convolving twice with $\sigma_1 = 1$ equals once with $\sigma_2 = \sqrt{2}$.

■ **Example 1.8 Step 1:** Define the relationship

$$g(x;t_1) * g(x;t_1) = g(x;2t_1)$$

Where $t_1 = \sigma_1^2 = 1^2 = 1$, so $2t_1 = 2$.
Step 2: Verify with specific values. For $x = 1$:
- Single convolution: $g(1;1) = \frac{1}{\sqrt{2\pi}} e^{-1/2} \approx 0.242$
- Double convolution: $g(1;2) = \frac{1}{\sqrt{4\pi}} e^{-1/4} \approx 0.221$

Step 3: Mathematical verification

$$(g_1 * g_1)(x) = \int g_1(x-\tau) g_1(\tau)\, d\tau$$

This integral evaluates to $g(x;2)$ due to the semi-group property.
Step 4: Practical verification

```
# Two convolutions with sigma = 1
result1 = convolve(convolve(image, g1), g1)

# One convolution with sigma = \sqrt(2)
result2 = convolve(image, g2)

# Difference should be \approx 0
difference = abs(result1 - result2).mean()
```

Result: The semi-group property allows efficient multi-scale processing. ■

Example 5: Derivative Property in Edge Detection

Given: Apply edge detection using the derivative property.

■ **Example 1.9** **Step 1:** Define the relationship

$$\frac{\partial}{\partial x}(I * g) = I * \frac{\partial g}{\partial x}$$

Step 2: Create test image with edge

$$I = \begin{bmatrix} 0 & 0 & 0 & 5 & 5 \\ 0 & 0 & 5 & 5 & 5 \\ 0 & 0 & 5 & 5 & 5 \end{bmatrix}$$

Step 3: Compute Gaussian derivative kernel ($\sigma = 1$) at positions $x = [-2, -1, 0, 1, 2]$

- $\frac{dg}{dx}(-1) = \frac{1}{1} \cdot 0.607 = 0.607$
- $\frac{dg}{dx}(0) = 0$
- $\frac{dg}{dx}(1) = -\frac{1}{1} \cdot 0.607 = -0.607$

Step 4: Apply derivative kernel to detect vertical edge (row 1, position 3)

$$0 \cdot 0.607 + 0 \cdot 0 + 5 \cdot (-0.607) = -3.035$$

Step 5: Compare with finite difference + smoothing

- Finite difference: $5 - 0 = 5$
- After Gaussian smoothing: ≈ 3.5
- Derivative of Gaussian: ≈ 3.0

Result: Derivative of Gaussian provides smoother, more robust edge detection. ■

Example 6: Boundary Length Calculation

Given: Compute boundary length for a simple segmentation.

■ **Example 1.10** **Segmentation image:**

$$s = \begin{bmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 3 & 3 & 2 & 2 \\ 3 & 3 & 3 & 3 \end{bmatrix}$$

Step 1: Check vertical boundaries (rows). Compare each pixel with the one below:

- Row 1–2: $[1,1,2,2]$ vs $[1,1,2,2] \to [0,0,0,0] \to 0$ differences
- Row 2–3: $[1,1,2,2]$ vs $[3,3,2,2] \to [1,1,0,0] \to 2$ differences
- Row 3–4: $[3,3,2,2]$ vs $[3,3,3,3] \to [0,0,1,1] \to 2$ differences

Vertical boundaries total: $0+2+2=4$

Step 2: Check horizontal boundaries (columns). Compare each pixel with the one to the right:

- Col 1–2: $[1,1,3,3]$ vs $[1,1,3,3] \to [0,0,0,0] \to 0$ differences
- Col 2–3: $[1,1,3,3]$ vs $[2,2,2,3] \to [1,1,1,0] \to 3$ differences
- Col 3–4: $[2,2,2,3]$ vs $[2,2,3,3] \to [0,0,0,0] \to 0$ differences

Horizontal boundaries total: $0+3+0=3$

Step 3: Total boundary length

$$\text{Total} = \text{Vertical} + \text{Horizontal} = 4+3=7$$

Implementation verification (Python):

```
def boundary_length(im):
    return (im[:-1] != im[1:]).sum() + (im[:, :-1] != im[:, 1:]).sum()
```

Result: The segmentation has 7 boundary pixels between different regions. ■

Example 7: Explicit vs Implicit Curve Smoothing

Given: Smooth a curve with 4 points using different methods.

■ **Example 1.11 Initial curve points:**

$$X = \begin{bmatrix} 0 & 0 \\ 1 & 2 \\ 3 & 1 \\ 2 & 0 \end{bmatrix}$$

Step 1: Create Laplacian matrix ($N=4$):

$$L = \begin{bmatrix} -2 & 1 & 0 & 1 \\ 1 & -2 & 1 & 0 \\ 0 & 1 & -2 & 1 \\ 1 & 0 & 1 & -2 \end{bmatrix}$$

Step 2: Explicit smoothing with $\lambda = 0.1$:

$$X_{\text{new}} = (I + \lambda L)X$$

$$I + 0.1L = \begin{bmatrix} 0.8 & 0.1 & 0.0 & 0.1 \\ 0.1 & 0.8 & 0.1 & 0.0 \\ 0.0 & 0.1 & 0.8 & 0.1 \\ 0.1 & 0.0 & 0.1 & 0.8 \end{bmatrix}$$

Apply to x-coordinates $[0,1,3,2]$:

- $x_1 = 0.8 \cdot 0 + 0.1 \cdot 1 + 0.0 \cdot 3 + 0.1 \cdot 2 = 0.3$

- $x_2 = 0.1 \cdot 0 + 0.8 \cdot 1 + 0.1 \cdot 3 + 0.0 \cdot 2 = 1.1$
- $x_3 = 0.0 \cdot 0 + 0.1 \cdot 1 + 0.8 \cdot 3 + 0.1 \cdot 2 = 2.7$
- $x_4 = 0.1 \cdot 0 + 0.0 \cdot 1 + 0.1 \cdot 3 + 0.8 \cdot 2 = 1.9$

Step 3: Implicit smoothing with $\lambda = 0.5$:

$$X_{\text{new}} = (I - \lambda L)^{-1} X$$

$$I - 0.5L = \begin{bmatrix} 2.0 & -0.5 & 0.0 & -0.5 \\ -0.5 & 2.0 & -0.5 & 0.0 \\ 0.0 & -0.5 & 2.0 & -0.5 \\ -0.5 & 0.0 & -0.5 & 2.0 \end{bmatrix}$$

After matrix inversion and multiplication, we get new x-coordinates:

$$[0.4, 1.2, 2.6, 1.8]$$

Result: Implicit smoothing allows larger λ values without oscillations. ■

Example 8: Regularization Matrix with Elasticity and Rigidity

Given: Create a regularization matrix combining elasticity ($\alpha = 1$) and rigidity ($\beta = 0.5$).

■ **Example 1.12** **Step 1:** Define kernel coefficients

- Elasticity kernel: $\alpha[0, 1, -2, 1, 0]$
- Rigidity kernel: $\beta[-1, 4, -6, 4, -1]$

Step 2: Combine kernels

$$s = 1 \cdot [0, 1, -2, 1, 0] + 0.5 \cdot [-1, 4, -6, 4, -1]$$

$$s = [0, 1, -2, 1, 0] + [-0.5, 2, -3, 2, -0.5] = [-0.5, 3, -5, 3, -0.5]$$

Step 3: Create circulant matrix ($N = 5$)

$$S = \begin{bmatrix} -5.0 & 3.0 & -0.5 & -0.5 & 3.0 \\ 3.0 & -5.0 & 3.0 & -0.5 & -0.5 \\ -0.5 & 3.0 & -5.0 & 3.0 & -0.5 \\ -0.5 & -0.5 & 3.0 & -5.0 & 3.0 \\ 3.0 & -0.5 & -0.5 & 3.0 & -5.0 \end{bmatrix}$$

Step 4: Calculate regularization matrix

$$R = (I - S)^{-1}$$

After matrix inversion:

$$R = \begin{bmatrix} 1.23 & 0.18 & 0.05 & 0.05 & 0.18 \\ 0.18 & 1.23 & 0.18 & 0.05 & 0.05 \\ 0.05 & 0.18 & 1.23 & 0.18 & 0.05 \\ 0.05 & 0.05 & 0.18 & 1.23 & 0.18 \\ 0.18 & 0.05 & 0.05 & 0.18 & 1.23 \end{bmatrix}$$

Result: This matrix balances length minimization (α) and curvature minimization (β). ■

Example 9: Gaussian Kernel Truncation Analysis

Given: Determine optimal truncation radius for $\sigma = 3$.

■ **Example 1.13** **Step 1:** Calculate kernel values at different radii. For $\sigma = 3$, $t = 9$:

$$g(x) = \frac{1}{\sqrt{18\pi}} e^{-x^2/18}$$

Step 2: Evaluate at multiples of σ
- $x = 3(\sigma)$: $g(3) = \frac{1}{\sqrt{18\pi}} e^{-0.5} \approx 0.0815$
- $x = 6(2\sigma)$: $g(6) = \frac{1}{\sqrt{18\pi}} e^{-2} \approx 0.022$
- $x = 9(3\sigma)$: $g(9) = \frac{1}{\sqrt{18\pi}} e^{-4.5} \approx 0.0018$
- $x = 12(4\sigma)$: $g(12) = \frac{1}{\sqrt{18\pi}} e^{-8} \approx 0.00005$

Step 3: Calculate cumulative energy within radius $r = \int_{-r}^{r} g(x)dx$
- Within $\pm 1\sigma$: $\sim 68.3\%$
- Within $\pm 2\sigma$: $\sim 95.4\%$
- Within $\pm 3\sigma$: $\sim 99.7\%$
- Within $\pm 4\sigma$: $\sim 99.99\%$

Step 4: Determine optimal truncation
- 3σ: radius = 9, captures 99.7
- 4σ: radius = 12, captures 99.99

Result: 3σ truncation is usually sufficient; 4σ for high-precision applications. ■

Example 10: Convolution Complexity Analysis

Given: Compare computational complexity of different convolution approaches.

■ **Example 1.14** **Image size:** 512×512, **Kernel size:** $\sigma = 5 \Rightarrow 4\sigma = 20 \Rightarrow 41 \times 41$ kernel

Method 1: Direct 2D convolution
- Operations per pixel: $41 \times 41 = 1,681$
- Total operations: $512 \times 512 \times 1,681 \approx 441$ million

Method 2: Separable convolution
- Horizontal pass: $512 \times 512 \times 41 \approx 10.7$ million
- Vertical pass: $512 \times 512 \times 41 \approx 10.7$ million
- Total: ≈ 21.4 million

Method 3: Multi-scale (semi-group)
- First: $\sigma_1 = 2.5 \Rightarrow 21 \times 21$
- Second: $\sigma_2 = 2.5 \Rightarrow 21 \times 21$
- Total: $2 \times (512 \times 512 \times 21) \approx 11.0$ million

Complexity comparison:
- Direct 2D: $\mathcal{O}(n^2k^2)$, where n = image size, k = kernel size
- Separable: $\mathcal{O}(2n^2k)$
- Multi-scale: $\mathcal{O}(mn^2k)$, m = number of scales

Result: Separable convolution provides $\sim 20\times$ speedup over direct 2D convolution.

■

2. Scale Space

Scale-Space Blob Detection

High-Level Overview

Scale-space blob detection is a fundamental technique in advanced image analysis that enables the detection and measurement of blob-like structures in images independent of their scale. This approach is built on Gaussian scale-space theory, where images are represented across multiple scales by convolving them with Gaussian kernels of varying variances. The key innovation is *scale invariance*—the ability to characterize the same feature regardless of its size in different images.

In this topic, blobs are detected by finding local extrema of the Laplacian of the scale-space, with appropriate normalization to compensate for scale effects. The practical application demonstrated is detecting and measuring glass fibers in composite materials using various imaging techniques, particularly X-ray CT. This topic forms the foundation for feature-based image analysis, with applications in object detection, image registration, and material characterization.

Key Theoretical Concepts

Gaussian Scale-Space

- **Definition:** A multi-scale representation $L : \mathbb{R}^2 \times \mathbb{R}^+ \to \mathbb{R}$ of an image $I : \mathbb{R}^2 \to \mathbb{R}$, obtained by convolving the image with Gaussian kernels of varying scale parameters t (variance).
- **Formal properties:** For a 2D image, the Gaussian scale-space is defined as:

$$L(x,y;t) = \sum_{\gamma=-w}^{w} \sum_{\beta=-w}^{w} I(x-\gamma, y-\beta) g(\beta,\gamma;t)$$

where

$$g(x,y;t) = \frac{1}{2\pi t} e^{-\frac{x^2+y^2}{2t}}$$

is the 2D Gaussian kernel.

- **Intuitive explanation:** Scale-space represents an image at different levels of detail. As t increases, the image becomes more smoothed, with finer details gradually disappearing.
- The start condition is defined as $L(x,y;0) = I(x,y)$.

Blob Detection using Laplacian

- **Definition:** A method for detecting blob-like structures by finding local maxima and minima of the Laplacian of the scale-space.
- **Formal properties:**
  - For the feature detection, we need to compute the derivatives of the scale space representation. This is conveniently achieved by convolving an image with a kernel that is a derivative of a Gaussian. Blob detection uses second order derivatives, namely the Laplacian.
  - The Laplacian is defined as $\nabla^2 L = L_{xx} + L_{yy}$
  - Laplacian gives a high response where there is a blob in the image. In order to detect blobs, we need to find local maxima and minima of the Laplacian Some local maxima and minima will, however, be very weak and should not be detected as blobs. Therefore, only maxima and minima with absolute Laplacian respose higer than a certain **threshold** should be detected as blobs.
  - As we detect the blobs throughout a different scales, the image in a scale-space representation will be increasingly smoothed and with increasing scale t, pixel intensity values will shift towards the average value of the image. Therefore, the absolute values of derivatives will become smaller when increasing t. For blob detection, this means that the maginute of the local maxima and minima in the Laplacian of the scale space $\nabla^2 L$ will decrease and this smoothing must be compensated. **Therefore, the scale of normalised Laplacian of the scale space is $\mathbf{t\nabla^2 L}$.**
  - For scale-invariant detection, the **scale-normalized Laplacian** $t\nabla^2 L$ is used.
  - The radius r of a blob detected at scale t is $r = \sqrt{2t}$
- **Intuitive explanation:** The Laplacian measures the sum of second derivatives, which gives a high response at blob-like structures. Bright blobs correspond to minima, while dark blobs correspond to maxima in the Laplacian. Scale normalization compensates for the decrease in derivative magnitude as scale increases.

Implementation Walk-Through

`getGaussDerivative(t)`

Purpose: Computes Gaussian kernel and its first, second, and third order derivatives for a given variance t.

Algorithm:

Python Code:

```
def getGaussDerivative(t):
    kSize = 5
    s = np.sqrt(t)  # standard deviation
    x = np.arange(int(-np.ceil(s * kSize)), int(np.ceil(s * kSize)) + 1)
    x = np.reshape(x, (-1, 1))  # reshape to column vector
    g = np.exp(-x**2 / (2 * t))  # Gaussian kernel
```

```
    g = g / np.sum(g)  # normalize to sum to 1
    dg = -x / t * g  # first derivative
    ddg = -g / t - x / t * dg  # second derivative
    dddg = -2 * dg / t - x / t * ddg  # third derivative
    return g, dg, ddg, dddg
```

Steps:

1. Compute standard deviation $s = \sqrt{t}$
2. Define x coordinates over range $\pm 5\sigma$
3. Compute 1D Gaussian kernel g
4. Normalize kernel so that $\sum g = 1$
5. Compute first derivative: $dg = -\frac{x}{t} \cdot g$
6. Compute second derivative: $ddg = -\frac{g}{t} - \frac{x}{t} \cdot dg$
7. Compute third derivative: $dddg = -\frac{2 \cdot dg}{t} - \frac{x}{t} \cdot ddg$

Complexity: $\mathcal{O}(n)$ where n is the kernel size, which depends on t.

Common pitfalls:

- Not making the kernel large enough (should be at least $\pm 3\sqrt{t}$)
- Not normalizing the Gaussian before computing its derivatives

Blob Detection

Blob Detection at One Scale

Purpose: Detects blobs in an image at a single scale using the Laplacian.

Algorithm:

Python Code:

```
# Compute the Laplacian of the image at scale t
g, dg, ddg, dddg = getGaussDerivative(t)
Lxx = cv2.filter2D(cv2.filter2D(im, -1, g), -1, ddg.T)  # x direction
Lyy = cv2.filter2D(cv2.filter2D(im, -1, ddg), -1, g.T)  # y direction
L_blob = t * (Lxx + Lyy)  # scale-normalized Laplacian

# Find local maxima and minima above threshold
coord_pos = skimage.feature.peak_local_max(L_blob, threshold_abs=magnitudeThres)
coord_neg = skimage.feature.peak_local_max(-L_blob, threshold_abs=magnitudeThres)
coord = np.r_[coord_pos, coord_neg]  # combine responses
```

Complexity: $\mathcal{O}(nm)$, where n and m are the image dimensions.

Key insight: The convolution order is critical—we use the second derivative in one direction and Gaussian smoothing in the other.

Blob Detection on Multiple Scales

Purpose: Extends detection to multiple scales to find blobs of different sizes.

Algorithm:
Python Code:

```
# Define range of scales
t = 15
g, dg, ddg, dddg = getGaussDerivative(t)
r, c = im.shape
n = 100
L_blob_vol = np.zeros((r, c, n))
tStep = np.zeros(n)

# Compute scale-normalized Laplacian at each scale
Lg = im
for i in range(0, n):
    tStep[i] = t * i
    L_blob_vol[:,:,i] = t*i*(cv2.filter2D(cv2.filter2D(Lg, -1, g), -1, ddg.T) +
                             cv2.filter2D(cv2.filter2D(Lg, -1, ddg), -1, g.T))
    Lg = cv2.filter2D(cv2.filter2D(Lg, -1, g), -1, g.T)  # smooth for next iteration

# Find maxima and minima in 3D scale-space
coord_pos = skimage.feature.peak_local_max(L_blob_vol, threshold_abs=thres)
coord_neg = skimage.feature.peak_local_max(-L_blob_vol, threshold_abs=thres)
coord = np.r_[coord_pos, coord_neg]
```

Complexity: $\mathcal{O}(nmk)$, where k is the number of scales.

Optimization: The implementation efficiently reuses previously smoothed images for each scale iteration.

`detectFibers(im, diameterLimit, stepSize, tCenter, thresMagnitude)`

Purpose: Specialized function to detect fibers in composite material images.

Algorithm:
Python Code:

```
def detectFibers(im, diameterLimit, stepSize, tCenter, thresMagnitude):
    # Convert diameter limits to scales
    radiusLimit = diameterLimit / 2
    radiusSteps = np.arange(radiusLimit[0], radiusLimit[1] + 0.1, stepSize)
    tStep = radiusSteps**2 / np.sqrt(2)

    # Compute Laplacian scale-space
    r, c = im.shape
    n = tStep.shape[0]
    L_blob_vol = np.zeros((r, c, n))

    for i in range(0, n):
        g, dg, ddg, dddg = getGaussDerivative(tStep[i])
```

```
        L_blob_vol[:, :, i] = tStep[i] * (
            cv2.filter2D(cv2.filter2D(im, -1, g), -1, ddg.T) +
            cv2.filter2D(cv2.filter2D(im, -1, ddg), -1, g.T)
        )

    # Detect fiber centers using Gaussian smoothing
    g, dg, ddg, dddg = getGaussDerivative(tCenter)
    Lg = cv2.filter2D(cv2.filter2D(im, -1, g), -1, g.T)
    coord = skimage.feature.peak_local_max(Lg, threshold_abs=thresMagnitude)

    # Find scale of each fiber as minimum over scales
    magnitudeIm = np.min(L_blob_vol, axis=2)
    scaleIm = np.argmin(L_blob_vol, axis=2)
    scales = scaleIm[coord[:, 0], coord[:, 1]]
    magnitudes = -magnitudeIm[coord[:, 0], coord[:, 1]]

    # Filter by magnitude threshold
    idx = np.where(magnitudes > thresMagnitude)
    coord = coord[idx[0], :]
    scale = tStep[scales[idx[0]]]

    return coord, scale
```

Key innovation: Separates center detection (using Gaussian smoothing) from scale determination (using Laplacian scale-space), which improves detection in real-world fiber images.

Crucial Details & 'Exam Traps'

- The **scale parameter** t is the *variance* of the Gaussian kernel, not the standard deviation ($\sigma = \sqrt{t}$).
- The **radius of a blob at scale** t is $\sqrt{2t}$, not $\sqrt{t}$ or t—this relationship is derived from the Laplacian of Gaussian properties.
- **Scale normalization** (multiplying by t) is critical for detecting blobs at their correct scales; forgetting this will bias detection toward smaller blobs.
- When computing L_{xx} and L_{yy} with separable filters, you must use the *second derivative in one direction and Gaussian in the other*.
- **Bright blobs** correspond to *minima* in the Laplacian, while **dark blobs** correspond to *maxima*.
- For multi-scale detection, a pixel must be an extremum in *both spatial dimensions and scale*.
- The Gaussian kernel size should be at least $\pm 3\sqrt{t}$ to capture most of the kernel's energy.
- Be careful with coordinate indexing: image coordinates are typically (row, column) = (y,x) while plotting uses (x,y).

Worked Example Trace

Let's trace through blob detection at a single scale on a test image:

1. **Start with image** `test_blob_uniform.png` containing uniform-sized blobs.
2. **Choose scale** $t = 325$ based on blob sizes in the image.

3. **Compute Gaussian kernels and derivatives:**

```
g, dg, ddg, dddg = getGaussDerivative(325)
# g is a normalized Gaussian kernel of size ~54 pixels
# ddg is its second derivative
```

4. **Compute the Laplacian:**

```
Lxx = cv2.filter2D(cv2.filter2D(im, -1, g), -1, ddg.T)
Lyy = cv2.filter2D(cv2.filter2D(im, -1, ddg), -1, g.T)
L_blob = 325 * (Lxx + Lyy)  # scale-normalized Laplacian
```

5. **Find local extrema with threshold 50:**

```
coord_pos = skimage.feature.peak_local_max(L_blob, threshold_abs=50)
coord_neg = skimage.feature.peak_local_max(-L_blob, threshold_abs=50)
coord = np.r_[coord_pos, coord_neg]  # ~10 blobs detected
```

6. Visualize with Circles of Radius $\sqrt{2 \cdot 325} \approx 25.5$ pixels

Python Code:

```
# Create circle coordinates with radius sqrt(2 * t)
theta = np.arange(0, 2 * np.pi, step=np.pi / 100)
theta = np.append(theta, 0)
circ = np.array((np.cos(theta), np.sin(theta)))

# Plot circles at detected coordinates
plt.plot(coord[:, 1], coord[:, 0], 'r')  # center points
circ_y = np.sqrt(2 * 325) * circ[0, :] + coord[:, 0][:, np.newaxis]
circ_x = np.sqrt(2 * 325) * circ[1, :] + coord[:, 1][:, np.newaxis]
plt.plot(circ_x.T, circ_y.T, 'r')  # red circles outlining blobs
```

Explanation: The resulting visualization shows red circles that accurately outline the blobs in the image based on the estimated scale.

Cheat-Sheet Section (Blob Detection + Gaussian Derivatives)

- **1D Gaussian:** $g(x) = \frac{1}{\sqrt{2\pi t}} e^{-x^2/(2t)}$
- **First derivative:** $\frac{dg}{dx} = -\frac{x}{t} \cdot g(x)$
- **Second derivative:** $\frac{d^2g}{dx^2} = -\frac{g(x)}{t} - \frac{x}{t} \cdot \frac{dg}{dx}$
- **Laplacian:** $\nabla^2 L = L_{xx} + L_{yy}$
- **Scale-normalized Laplacian:** $t \cdot \nabla^2 L$
- **Blob radius at scale t:** $r = \sqrt{2t}$
- **Kernel size:** Should be at least $\pm 3\sqrt{t}$
- **Multi-scale detection:** $t\nabla^2 L$ must be an extremum in x, y, and t
- **Bright blobs:** Correspond to *minima* of Laplacian; **dark blobs:** to *maxima*
- **Diameter-to-scale conversion:** $t = \frac{(\text{diameter})^2}{2}$

Reflective Questions

1. **Why is it necessary to multiply the Laplacian by the scale parameter t when detecting blobs across different scales?**
   Without this normalization, the Laplacian's magnitude would decrease with increasing scale, biasing detection toward smaller blobs. The t factor compensates for this effect, making the detection truly scale-invariant.
2. **In the fiber detection algorithm, why are two different approaches used for finding centers versus determining scales?**
   Center detection uses Gaussian smoothing to find peaks in intensity, which is more reliable for fiber centers, while scale determination uses the Laplacian scale-space to accurately measure size. This hybrid approach improves detection in real-world images where fibers may have variable contrast but similar sizes.
3. **How would you modify the blob detection algorithm to handle elongated structures (like blood vessels) rather than circular blobs?**
   The algorithm would need to be adapted to use directional derivatives or a structure tensor approach to capture orientation. The Hessian matrix eigenvalues could be analyzed to distinguish between circular blobs and elongated structures, with different responses at different orientations.
4. **What is the theoretical relationship between blob radius and scale parameter t, and how is this derived?**
   The relationship $r = \sqrt{2t}$ comes from analyzing where the Laplacian of Gaussian has its peak response for a perfect circular blob. The second derivative of a Gaussian has its extremum at distance $\sqrt{2t}$ from the center, which corresponds to where the curvature changes most rapidly.
5. **How would the blob detection performance change if you replaced the Gaussian kernel with a different smoothing kernel (e.g., box filter)? What properties would be gained or lost?**
   Replacing the Gaussian would lose theoretical properties like separability and scale-space causality. A box filter would be faster to compute but would introduce artifacts at blob boundaries and lose rotational invariance. The Gaussian is optimal because it's the only kernel that satisfies all the scale-space axioms.

Scale-Space Blob Detection: Worked Examples

Example 1: Computing 2D Gaussian Kernel

Given: Scale parameter $t = 4$, compute the 2D Gaussian kernel at center and adjacent pixels.

■ **Example 2.1** **Step 1:** The 2D Gaussian kernel is defined as:

$$g(x,y;t) = \frac{1}{2\pi t} e^{-\frac{x^2+y^2}{2t}}$$

Step 2: With $t = 4$, substitute values:

$$g(x,y;4) = \frac{1}{2\pi \cdot 4} e^{-\frac{x^2+y^2}{8}} = \frac{1}{8\pi} e^{-\frac{x^2+y^2}{8}}$$

Step 3: Calculate for specific positions:

- At center (0,0): $g(0,0;4) = \frac{1}{8\pi} e^0 = \frac{1}{8\pi} \approx 0.0398$

- At (1,0): $g(1,0;4) = \frac{1}{8\pi}e^{-\frac{1}{8}} \approx 0.0351$
- At (1,1): $g(1,1;4) = \frac{1}{8\pi}e^{-\frac{2}{8}} = \frac{1}{8\pi}e^{-0.25} \approx 0.0309$
- At (2,0): $g(2,0;4) = \frac{1}{8\pi}e^{-\frac{4}{8}} = \frac{1}{8\pi}e^{-0.5} \approx 0.0241$

Result: The kernel values decrease exponentially with distance from the center. ∎

Example 2: Computing Gaussian Derivatives

Given: $t = 9$, compute the 1D Gaussian and its derivatives at $x = 3$.

∎ **Example 2.2** **Step 1:** 1D Gaussian:

$$g(x) = \frac{1}{\sqrt{2\pi t}}e^{-x^2/(2t)}$$

With $t = 9$,

$$g(3) = \frac{1}{\sqrt{18\pi}}e^{-0.5} \approx 0.08075$$

Step 2: First derivative:

$$\frac{dg}{dx} = -\frac{x}{t} \cdot g(x)$$

$$\left.\frac{dg}{dx}\right|_{x=3} = -\frac{3}{9} \cdot 0.0807 = -\frac{1}{3} \cdot 0.0807 \approx -0.0269$$

Step 3: Second derivative:

$$\frac{d^2g}{dx^2} = -\frac{g(x)}{t} - \frac{x}{t} \cdot \frac{dg}{dx}$$

$$\left.\frac{d^2g}{dx^2}\right|_{x=3} = -\frac{0.0807}{9} - \frac{3}{9} \cdot (-0.0269) = -0.00897 + 0.00897 = 0$$

Insight: The second derivative is zero at $x = \sqrt{t} = 3$, which is where the Laplacian response peaks. ∎

Example 3: Scale-Normalized Laplacian Calculation

Given: A blob with Laplacian response $\nabla^2 L = -0.05$ detected at scale $t = 16$.

∎ **Example 2.3** **Step 1:** Apply scale normalization:

$$\text{Scale-normalized Laplacian} = t \cdot \nabla^2 L$$

Step 2: Calculate:

$$16 \cdot (-0.05) = -0.8$$

Step 3: Compare at different scale $t = 64$. If raw Laplacian response drops to $\nabla^2 L = -0.0125$,

$$\text{Scale-normalized} = 64 \cdot (-0.0125) = -0.8$$

Result: Scale normalization makes the response consistent across scales, enabling proper blob detection. ∎

Example 4: Blob Radius Calculation

Given: Blobs detected at scales $t_1 = 25$, $t_2 = 100$, and $t_3 = 400$.

■ **Example 2.4** **Step 1:** Use the relationship $r = \sqrt{2t}$

Step 2: Calculate radii:

- For $t_1 = 25$: $r_1 = \sqrt{2 \cdot 25} = \sqrt{50} \approx 7.07$ pixels
- For $t_2 = 100$: $r_2 = \sqrt{2 \cdot 100} = \sqrt{200} \approx 14.14$ pixels
- For $t_3 = 400$: $r_3 = \sqrt{2 \cdot 400} = \sqrt{800} \approx 28.28$ pixels

Step 3: Convert to diameters:

- Diameter$_1$ = $2 \cdot 7.07 = 14.14$ pixels
- Diameter$_2$ = $2 \cdot 14.14 = 28.28$ pixels
- Diameter$_3$ = $2 \cdot 28.28 = 56.56$ pixels

Verification: Notice that diameter = $2 \cdot \sqrt{2t} = 2 \cdot \sqrt{2} \cdot \sqrt{t}$ ■

Example 5: Converting Fiber Diameter to Scale Parameter

Given: Detecting glass fibers with diameters ranging from 10 to 30 pixels.

■ **Example 2.5** **Step 1:** Use the relationship $r = \sqrt{2t}$, so $t = \frac{r^2}{2}$

Since diameter = $2r$, we have:

$$t = \frac{(\text{diameter}/2)^2}{2} = \frac{\text{diameter}^2}{8}$$

Step 2: Calculate scale range:

- For diameter = 10: $t_{\min} = \frac{10^2}{8} = \frac{100}{8} = 12.5$
- For diameter = 30: $t_{\max} = \frac{30^2}{8} = \frac{900}{8} = 112.5$

Step 3: Choose step size for detection. If stepSize = 2 pixels in diameter space:

$$t_{\text{step}} = \frac{(\text{diameter} + 2)^2}{8} - \frac{\text{diameter}^2}{8}$$

- For diameter = 20:

$$\Delta t = \frac{22^2 - 20^2}{8} = \frac{484 - 400}{8} = 10.5$$

Result: Scale parameters range from 12.5 to 112.5 for the given fiber sizes. ■

Example 6: Multi-Scale Extrema Detection

Given: Scale-space responses at a pixel across different scales.

■ **Example 2.6** **Scale responses at pixel (100, 150):**

- $t = 4$: $\nabla^2 L = -1.2$
- $t = 9$: $\nabla^2 L = -2.8$
- $t = 16$: $\nabla^2 L = -4.1$
- $t = 25$: $\nabla^2 L = -3.9$
- $t = 36$: $\nabla^2 L = -2.1$

Step 1: Find the minimum (for bright blobs): Minimum occurs at $t = 16$ with response -4.1

Step 2: Check if it's a local extremum:

- At $t = 9$: $-2.8 > -4.1$ check
- At $t = 25$: $-3.9 > -4.1$ check

Step 3: Calculate blob radius:

$$r = \sqrt{2 \cdot 16} = \sqrt{32} \approx 5.66 \text{ pixels}$$

Result: A bright blob of radius ~ 5.66 pixels is detected at this location. ■

Example 7: Threshold Application

Given: Scale-normalized Laplacian responses and threshold = 2.0.

■ **Example 2.7 Responses at different locations:**

- Location A: $|\nabla^2 L| = 3.5 > 2.0$ **Detected**
- Location B: $|\nabla^2 L| = 1.2 < 2.0$ **Rejected**
- Location C: $|\nabla^2 L| = 2.8 > 2.0$ **Detected**
- Location D: $|\nabla^2 L| = 0.9 < 2.0$ **Rejected**

Result: Only locations A and C are detected as valid blobs. The threshold eliminates weak responses that could be noise. ■

Example 8: Kernel Size Calculation

Given: Scale parameter $t = 64$, determine appropriate kernel size.

■ **Example 2.8 Step 1:** Calculate standard deviation:

$$\sigma = \sqrt{t} = \sqrt{64} = 8$$

Step 2: Apply the 3σ rule:

Kernel should extend at least $\pm 3\sigma = \pm 24$ pixels

Step 3: Determine total kernel size:

$2 \cdot 24 + 1 = 49$ pixels (odd number for symmetric kernel)

Step 4: Verify energy capture:

$$g(\pm 24) = \frac{1}{\sqrt{2\pi \cdot 64}} e^{-24^2/(2 \cdot 64)} = \frac{1}{8\sqrt{2\pi}} e^{-4.5} \approx 0.00045$$

Result: A 49×49 kernel captures $> 99\%$ of the Gaussian energy. ■

3. Sift Features

High-Level Overview

This topic covers **feature-based image registration**, the process of aligning images depicting the same objects by matching distinctive image features. The approach uses **SIFT (Scale Invariant Feature Transform)** features, which are keypoints that remain stable across changes in scale, rotation, and illumination. The core workflow involves:

1. Detecting interest points in images. **The desired properties are:**
   - Uniqueness, ensuring they encode just one position in an image
   - Invariance to the changes in view-point, overall intensity change, and change in noise level
   - Robustness, allowing for detection despite image changes.
   - Efficiency in terms of both memory usage and computation time.
2. Computing distinctive descriptors around each point
3. Matching descriptors between images
4. Estimating geometric transformations (rotation, translation, scale) that align the matched point sets

This method enables robust matching even under significant viewpoint changes, noise, and partial occlusion. The mathematical foundation centers on least-squares fitting of 2D point correspondences using SVD-based solutions, with robust variants that handle outliers through iterative refinement. Applications include object recognition, 3D reconstruction, medical image alignment, and autonomous navigation.

Key Theoretical Concepts

SIFT Feature Properties

- **Definition:** Scale Invariant Feature Transform produces distinctive keypoints invariant to image scale, rotation, and partially invariant to illumination and 3D viewpoint changes.
- **Formal properties:**

  - Scale invariance through scale-space extrema detection
  - Rotation invariance via consistent orientation assignment
  - Robustness to affine distortion up to $\sim 50^\circ$ viewpoint change
- **Intuitive explanation:** SIFT mimics biological vision by detecting stable "interest points" that can be reliably found across different views of the same scene.
- **Algorithm stages:**
  1. Scale-space extrema detection using Difference-of-Gaussians
  2. Keypoint localization with sub-pixel accuracy
  3. Orientation assignment from local gradients
  4. Descriptor computation from oriented gradient histograms

Scale-Space Theory

- **Definition:** A continuous function $L(x,y,\sigma) = G(x,y,\sigma) * I(x,y)$, where G is a Gaussian kernel with scale parameter σ
- **Formal properties:**
  - The Gaussian is the unique scale-space kernel under reasonable assumptions (Koenderink, Lindeberg)
  - Scale-normalized Laplacian $\sigma^2\nabla^2 G$ produces most stable features
- **Proof sketch:** Difference-of-Gaussians approximates the Laplacian:

$$G(x,y,k\sigma) - G(x,y,\sigma) \approx (k-1)\sigma^2\nabla^2 G$$

- **Intuitive explanation:** Scale-space captures image structure at multiple resolutions simultaneously, enabling detection of features at their "natural" scale.

Point Set Registration Mathematics

- **Problem formulation:** Given corresponding 2D point sets P and Q, find transformation $q = sRp + t$ minimizing squared error
- **Scale estimation:**

$$s = \frac{\sum \|q_i - \mu_q\|}{\sum \|p_i - \mu_p\|}, \quad \text{where } \mu \text{ represents centroids}$$

- **Rotation via SVD:**
  - Covariance matrix: $C = \sum (q_i - \mu_q)(p_i - \mu_p)^\top$
  - SVD: $C = U\Sigma V^\top$
  - Rotation: $R = UDV^\top$, where $D = \text{diag}(1, \det(UV^\top))$ ensures proper rotation (no reflection)
- **Translation:**

$$t = \mu_q - sR\mu_p \quad \text{(difference of transformed centroids)}$$

- **Proof sketch:** This minimizes $\sum \|q_i - sRp_i - t\|^2$ by setting derivatives to zero and using properties of orthogonal matrices

Robust Estimation

- **Definition:** Iterative refinement removing outliers based on residual distances
- **Algorithm:**
  1. Compute initial transformation
  2. Calculate residuals
  3. Remove points with residuals $>$ threshold
  4. Re-estimate with inliers
- **Theoretical justification:** Most correspondences are correct, so least-squares gives reasonable initial estimate; outliers have systematically larger residuals

Implementation Walk-Through: Blob Detection (`detectFibers`)

Purpose: Detects circular blob-like structures (fibers) in images using scale-space maxima detection.

Algorithm:

Step 1: Create scale pyramid

```
radiusSteps = np.arange(radiusLimit[0], radiusLimit[1]+0.1, stepSize)
tStep = radiusSteps**2 / np.sqrt(2)  # Convert radius to scale parameter
```

Step 2: Compute Laplacian of Gaussian at each scale

```
for i in range(0, n):
    g, dg, ddg, dddg = getGaussDerivative(tStep[i])
    L_blob_vol[:, :, i] = tStep[i] * (
        cv2.filter2D(cv2.filter2D(im, -1, g), -1, ddg.T) +
        cv2.filter2D(cv2.filter2D(im, -1, ddg), -1, g.T)
    )
```

Step 3: Find local maxima across scales

```
magnitudeIm = np.min(L_blob_vol, axis=2)     # Best response across scales
scaleIm = np.argmin(L_blob_vol, axis=2)      # Scale giving best response
```

Complexity: $\mathcal{O}(n \cdot m \cdot k)$, where n, m are image dimensions and k is the number of scales.
Common pitfalls:

- Ensure proper conversion between radius and scale parameter
- Sample enough scales to detect blobs of all relevant sizes

Transformation Estimation (`get_transformation`)

Purpose: Computes optimal rotation, translation, and scale between two corresponding point sets.

Algorithm

Step 1: Compute centroids

```
m_p = np.mean(p, axis=1, keepdims=True)
m_q = np.mean(q, axis=1, keepdims=True)
```

Step 2: Estimate scale as ratio of spreads

```
s = np.linalg.norm(q - m_q, axis=0).sum() / np.linalg.norm(p - m_p, axis=0).sum()
```

Step 3: SVD-based rotation estimation

```
C = (q - m_q) @ (p - m_p).T  # Cross-covariance matrix
U, S, V = np.linalg.svd(C)
R_ = U @ V
R = R_ @ np.array([[1, 0], [0, np.linalg.det(R_)]])  # Ensure proper rotation
```

Step 4: Compute translation

```
t = m_q - s * R @ m_p
```

Complexity: $\mathcal{O}(n)$ for n points (SVD is $\mathcal{O}(1)$ for 2×2 matrices)
Common pitfalls:
- Confusing reflection with rotation: ensure $\det(R) = 1$
- Numerical instability in SVD when point sets are nearly collinear

Robust Transformation (`get_robust_transformation`)

Purpose: Computes transformation while automatically removing outlier correspondences.

Algorithm

Step 1: Initial transformation estimate

```
R, t, s = get_transformation(p, q)
```

Step 2: Compute transformation residuals

```
q_1 = s * R @ p + t         # Transform p to q's coordinate frame
d = np.linalg.norm(q - q_1, axis=0)  # Point-wise distances
```

Step 3: Remove outliers and re-estimate

```
idx = np.where(d < thres)[0]  # Inlier indices
R, t, s = get_transformation(p[:, idx], q[:, idx])  # Re-estimate with inliers only
```

Complexity: $\mathcal{O}(n)$ per iteration; typically converges in 1–2 iterations.
Common pitfalls:
- Threshold selection: needs balance between inlier retention and outlier rejection
- Ensure enough inliers remain after filtering

Crucial Details & 'Exam Traps'

- **Scale parameter confusion:** Radius r relates to Gaussian scale via $\sigma^2 = \frac{r^2}{\sqrt{2}}$, not $r^2 = 2\sigma^2$
- **SVD reflection issue:** Always check $\det(UV^\top)$ and correct to ensure a proper rotation matrix
- **Coordinate system:** SIFT uses (x,y) while blob detection may use (row, col) — beware of transposition
- **Threshold sensitivity:** Threshold must balance inlier retention vs. outlier removal
- **Scale normalization:** Laplacian must be normalized by σ^2 for true scale invariance
- **Boundary effects:** Gaussian filtering near image edges can create artifacts
- **Memory vs accuracy:** More scale samples improve detection but increase computational cost
- **Minimum correspondences:** Need ≥ 3 point pairs for affine transformation, ≥ 2 for similarity transform

Worked Example: Point Set Registration

Given: Point sets with ground truth transformation:

```
# Ground truth
R_true = [[cos(76°), -sin(76°)], [sin(76°), cos(76°)]]  # 76° rotation
t_true = [[1], [-2]]                                    # Translation vector
s_true = 0.6                                            # Scale factor
```

Step 1: Compute centroids

```
p = [[1, 2, -1], [0, 1, -2]]     # Example points
q = s_true * R_true @ p + t_true

m_p = [[0.67], [-0.33]]          # Centroid of p
m_q = [[0.2], [-2.2]]            # Centroid of q
```

Step 2: Estimate scale

```
# Spread from centroid
spread_p = ||[0.33, 1.33, -1.67]|| + ||[0.33, 1.33, -1.67]|| = 4.47
spread_q = ||[-0.48, 0.8, -1.0]|| + ||[0.2, 0.8, -1.2]|| = 2.68

s_estimated = 2.68 / 4.47 = 0.60
```

Step 3: SVD for rotation

```
C = (q - m_q) @ (p - m_p).T = [[2.1, 1.4], [0.7, -2.1]]
U, Sigma, V = SVD(C)
R_estimated = U @ V           # Recovers 76° rotation
```

Step 4: Translation

```
t_estimated = m_q - s_estimated * R_estimated @ m_p
            = [[1.0], [-2.0]]
```

Conclusion: All estimated components match the ground truth parameters.

Cheat-Sheet Section

- **SIFT keypoint count:** $\sim$2000 features per 500×500 image
- **Scale sampling:** 3 scales per octave optimal for stability/efficiency trade-off
- **Orientation histogram:** 36 bins covering 360°; peaks within 80% of max create multiple keypoints
- **Descriptor size:** $4 \times 4 \times 8 = 128$-dimensional feature vector standard
- **Distance ratio test:** Reject matches where closest/second-closest > 0.8
- **SVD correction:** $R = UV^\top \cdot \text{diag}(1, \det(UV^\top))$ ensures rotation not reflection
- **Robust threshold:** Typically 3–5 pixels for transformation residuals
- **Gaussian scale:** $\sigma = r/\sqrt{2}$ for blob radius r
- **Minimum matches:** 3 points for affine, 2 for similarity transform

Reflective Questions

1. **Theory-Code Bridge:** The code computes `tStep = radiusSteps**2 / np.sqrt(2)`. Derive this relationship between blob radius and Gaussian scale parameter σ. Why is the $\sqrt{2}$ factor necessary?
2. **Robustness Analysis:** If 30% of point correspondences are outliers, will the robust transformation estimation succeed? What assumptions does this make about the distribution of outliers vs inliers?
3. **Scale-Space Interpretation:** The blob detector finds minima of the Laplacian response (`L_blob_vol`). Explain why minima correspond to bright circular blobs, and how this relates to the scale-normalized Laplacian.
4. **Transformation Degeneracy:** Under what geometric configurations of input points would the SVD-based rotation estimation fail or become ill-conditioned? How would you detect and handle these cases?
5. **Feature Matching Strategy:** Given SIFT's 128-dimensional descriptors, why does the distance ratio test (closest/second-closest < 0.8) work better than absolute distance thresholding for rejecting false matches?

Practical Examples for Feature-Based Image Registration

Example 1: SIFT Feature Detection in Action

Scenario: Matching Photos of the Same Building

Imagine you have two photos of a cathedral:

- **Image A:** Taken from the front, 50mm lens
- **Image B:** Taken from 45° to the right, 35mm lens (wider angle)

SIFT Detection Process

```
# What SIFT finds:
sift_features_A = [
    {location: (120, 80), scale: 2.5, orientation: 15°, descriptor: [0.1, 0.3, ...]},  # Corr
    {location: (200, 150), scale: 1.8, orientation: 90°, descriptor: [0.2, 0.1, ...]}, # Winc
    {location: (180, 200), scale: 3.2, orientation: 45°, descriptor: [0.0, 0.4, ...]}, # Arch
```

```
    # ... ~2000 more features
]
```

Why These Features Are Detected

- **Tower corner:** High gradient in multiple directions → stable keypoint
- **Window edge:** Strong vertical edge with distinctive texture → reliable across viewpoints
- **Architectural detail:** Unique carved pattern → highly distinctive descriptor

Matching Process

```
# For each feature in Image A, find best match in Image B
feature_A1 = [0.1, 0.3, 0.2, ...]  # 128-dim descriptor

candidates_B = [(distance: 0.3, feature_B5), (distance: 0.7, feature_B12), ...]

# Apply distance ratio test
if 0.3 / 0.7 < 0.8:  # closest/second_closest < threshold
    accept_match(feature_A1, feature_B5)
```

Example 2: Step-by-Step Transformation Calculation

Scenario: Medical Image Registration

A patient has CT scans taken 6 months apart. We need to align them to track tumor growth.

Given Matched Keypoints

```
# Points from Scan 1 (reference)
p = np.array([[100, 150, 200, 250],     # x coordinates
              [ 80, 120, 160, 200]])    # y coordinates

# Corresponding points from Scan 2 (to be aligned)
q = np.array([[ 95, 142, 189, 236],     # x coordinates
              [ 85, 127, 167, 207]])    # y coordinates
```

Step 1: Compute Centroids

```
m_p = np.mean(p, axis=1, keepdims=True) = [[175], [140]]
m_q = np.mean(q, axis=1, keepdims=True) = [[165.5], [146.5]]
```

Step 2: Center the Point Sets

```
p_centered = p - m_p = [[-75, -25, 25, 75],
                        [-60, -20, 20, 60]]

q_centered = q - m_q = [[-70.5, -23.5, 23.5, 70.5],
                        [-61.5, -19.5, 20.5, 60.5]]
```

Step 3: Estimate Scale

```
# Total distance from centroid
```

```
spread_p = sqrt(75^2 + 60^2) + sqrt(25^2 + 20^2) + ... = 256.4
spread_q = sqrt(70.5^2 + 61.5^2) + sqrt(23.5^2 + 19.5^2) + ... = 243.2

scale = spread_q / spread_p = 243.2 / 256.4 ≈ 0.949
```

Step 4: Estimate Rotation via SVD

```
C = q_centered @ p_centered.T =
    [[11025, 8800],
     [ 8800, 7040]]

U, S, V = SVD(C)
R_candidate = U @ V =
    [[ 0.996, -0.087],
     [ 0.087,  0.996]]  # ~5° rotation

det(R_candidate) = 1.0   # Proper rotation
```

Step 5: Compute Translation

```
t = m_q - scale * R @ m_p
  = [[165.5], [146.5]] - 0.949 * [[0.996, -0.087],
                                  [0.087,  0.996]] @ [[175], [140]]
  = [[165.5], [146.5]] - [[153.8], [147.2]]
  = [[11.7], [-0.7]]
```

Final transformation:

$$\text{Scan } 2 = 0.949 \times (\text{rotate Scan 1 by } 5^\circ) + [11.7, -0.7]$$

Example 3: Robust Estimation in Practice

Scenario: Drone Photography with Outliers

You're creating a panorama from drone photos, but some SIFT matches are wrong due to:

- Moving cars (appear in one image but not the other)
- Shadows that shifted between shots
- Repetitive building patterns causing false matches

Initial Matches (Including Outliers)

```
# 20 total matches, but 6 are outliers
p = [[correct_matches...], [outlier_matches...]]
q = [[correct_matches...], [wrong_correspondences...]]

# Initial transformation (corrupted by outliers)
R_initial, t_initial, s_initial = get_transformation(p, q)
# Result: rotation=15° (should be ~3°), scale=1.2 (should be ~1.0)
```

Robust Refinement

```
# Transform p using initial estimate
q_predicted = s_initial * R_initial @ p + t_initial

# Compute residuals
residuals = [2.1, 1.8, 2.5, 15.2, 1.9, 2.3, 12.8, 1.7, ...]
#              ^good^  ^outliers^      ^good^

# Remove outliers (threshold = 5 pixels)
inliers = [0, 1, 2, 4, 5, 7, ...]   # Indices where residual < 5
outliers = [3, 6, ...]              # Indices where residual \geq 5

# Re-estimate with inliers only
R_robust, t_robust, s_robust = get_transformation(p[:, inliers], q[:, inliers])
# Result: rotation=2.8°, scale=0.98 (much more reasonable!)
```

Example 4: Scale-Space Blob Detection

Scenario: Analyzing Cell Images

You need to detect and measure cell nuclei in a microscopy image.

Multi-Scale Detection

```
# Cell nuclei have diameters ranging from 10-25 pixels
diameter_range = [10, 25]
step_size = 0.5

# This creates scale pyramid:
scales = [5^2, 5.5^2, 6^2, 6.5^2, ..., 12.5^2] / sqrt(2)
#         = [12.5, 15.1, 18.0, 21.1, ..., 78.1]   # Sigma^2 values

# At each scale, compute Laplacian response:
for sigma_squared in scales:
    sigma = sqrt(sigma_squared)

    # Laplacian filter detects circular blobs of size ~Sigma
    response = sigma_squared * (d²/dx² + d²/dy²) * gaussian_smoothed_image
```

Typical responses:

- Small blob (10px diameter) $\rightarrow$ strongest response at $\sigma^2 \approx 25$
- Medium blob (18px diameter) $\rightarrow$ strongest response at $\sigma^2 \approx 81$
- Large blob (25px diameter) $\rightarrow$ strongest response at $\sigma^2 \approx 156$

Detection Results

```
detected_cells = [
    {center: (45, 78), diameter: 12, confidence: 0.8},  # Small nucleus
```

```
    {center: (123, 156), diameter: 19, confidence: 0.9}, # Medium nucleus
    {center: (200, 89), diameter: 24, confidence: 0.7},  # Large nucleus
]
```

Example 5: Real-World Failure Cases and Solutions

Case A: Repetitive Patterns (Building Windows)

Problem: Office building with identical windows creates many false matches.

```
# SIFT detects similar features at each window
window_features = [
  {loc: (100, 150), desc: [0.2, 0.3, 0.1, ...]},  # Window 1
  {loc: (100, 200), desc: [0.2, 0.3, 0.1, ...]},  # Window 2
  {loc: (100, 250), desc: [0.2, 0.3, 0.1, ...]},  # Window 3
]
# Distance ratio test fails because all windows look alike
```

Solution: Use spatial consistency.

- Match feature **grids**, not individual features.
- Geometric constraints eliminate false matches.

Case B: Low Texture Areas (Sky, Water)

Problem: Smooth regions produce few reliable keypoints.

```
# Ocean scene
detected_features = [
    # Sky region: 3 features (very few!)
    # Water region: 5 features (very few!)
    # Boat region: 150 features (plenty!)
]
# Insufficient matches for robust registration
```

Solution:

- Use edge-based features in addition to SIFT
- Apply local histogram equalization to enhance subtle textures
- Use global registration for texture-poor scenes

Case C: Extreme Scale Changes

Problem: Aerial vs ground-level photo of same building.

```
# Scale difference: 20x (aerial) vs 1x (ground level)
# SIFT's scale invariance typically handles up to ~4x scale difference
```

Solution:

- Create image pyramid with overlapping scale ranges
- Use multiple reference images at different scales
- Apply scale-space feature tracking across wide scales

Example 6: Performance Optimization Tips

Memory vs Accuracy Trade-offs

```
# High accuracy (slow)
sift_params = {
    'nfeatures': 0,         # Detect all possible features
    'nOctaveLayers': 5,     # Fine scale sampling
    'contrastThreshold': 0.01,
    'edgeThreshold': 15
}
# ~5000 features, very robust

# Balanced (recommended)
sift_params = {
    'nfeatures': 2000,
    'nOctaveLayers': 3,
    'contrastThreshold': 0.04,
    'edgeThreshold': 10
}
# ~2000 features, good speed/accuracy trade-off

# Speed optimized (fast)
sift_params = {
    'nfeatures': 500,
    'nOctaveLayers': 2,
    'contrastThreshold': 0.08,
    'edgeThreshold': 5
}
# ~500 features, fast but less robust
```

These examples show how theoretical concepts translate into practical adjustments, highlighting both the power and limitations of real-world feature-based registration.

Calculation 1: Scale-Space and Gaussian Derivatives

Problem: Compute Gaussian and its derivatives for blob detection

Given: Scale parameter $t = 4$, compute Gaussian kernel and derivatives at $x = [-2, -1, 0, 1, 2]$

Step 1: Gaussian Kernel $G(x)$

$$G(x) = \frac{1}{\sqrt{2\pi t}} e^{-x^2/(2t)} = \frac{1}{\sqrt{8\pi}} e^{-x^2/8}$$

$$G(-2) = (1/\sqrt{8\pi}) \cdot e^{-0.5} = 0.1996 \cdot 0.6065 = \mathbf{0.1211}$$
$$G(-1) = 0.1996 \cdot e^{-0.125} = 0.1996 \cdot 0.8825 = \mathbf{0.1762}$$
$$G(0) = 0.1996 \cdot 1 = \mathbf{0.1996}$$
$$G(1) = G(-1) = \mathbf{0.1762}$$
$$G(2) = G(-2) = \mathbf{0.1211}$$

Normalize:

$$\text{Sum} = 0.7942, \quad \Rightarrow \quad \text{Normalized kernel: } [0.1525, 0.2218, 0.2514, 0.2218, 0.1525]$$

Step 2: First Derivative $\frac{dG}{dx} = -\frac{x}{t} \cdot G(x) = -\frac{x}{4} \cdot G(x)$

$$\frac{dG}{dx}(-2) = -\frac{-2}{4} \cdot 0.1525 = 0.5 \cdot 0.1525 = \mathbf{0.0763}$$
$$\frac{dG}{dx}(-1) = 0.25 \cdot 0.2218 = \mathbf{0.0555}$$
$$\frac{dG}{dx}(0) = 0 \cdot 0.2514 = \mathbf{0.0000}$$
$$\frac{dG}{dx}(1) = -0.25 \cdot 0.2218 = \textbf{-0.0555}$$
$$\frac{dG}{dx}(2) = -0.5 \cdot 0.1525 = \textbf{-0.0763}$$

Step 3: Second Derivative (Laplacian component)

$$\frac{d^2G}{dx^2} = -\frac{G(x)}{t} - \frac{x}{t} \cdot \frac{dG}{dx} \Rightarrow \frac{d^2G}{dx^2} = -\frac{G(x)}{4} - \frac{x}{4} \cdot \frac{dG}{dx}$$

$$\frac{d^2G}{dx^2}(0) = -0.2514/4 = \textbf{-0.0629}$$
$$\frac{d^2G}{dx^2}(1) = -0.2218/4 - (1/4) \cdot (-0.0555) = -0.0555 + 0.0139 = \textbf{-0.0416}$$
$$\frac{d^2G}{dx^2}(2) = -0.1525/4 - (2/4) \cdot (-0.0763) = -0.0381 + 0.0381 = \mathbf{0.0000}$$

Problem

Find rotation, translation, and scale between two point sets.

Given point sets:

$$\mathbf{P} = \begin{bmatrix} 1 & 3 & 2 \\ 2 & 1 & 4 \end{bmatrix}, \quad \mathbf{Q} = \begin{bmatrix} 2.5 & 1.0 & 3.2 \\ 1.8 & 4.1 & 0.9 \end{bmatrix}$$

Step 1: Compute Centroids

$$\mu_P = \frac{1}{3}(1+3+2,\ 2+1+4) = [2.000,\ 2.333]$$

$$\mu_Q = \frac{1}{3}(2.5+1.0+3.2,\ 1.8+4.1+0.9) = [2.233,\ 2.267]$$

Step 2: Center the Point Sets

$$\tilde{\mathbf{P}} = \mathbf{P} - \boldsymbol{\mu}_P = \begin{bmatrix} -1.000 & 1.000 & 0.000 \\ -0.333 & -1.333 & 1.667 \end{bmatrix} \quad \tilde{\mathbf{Q}} = \mathbf{Q} - \boldsymbol{\mu}_Q = \begin{bmatrix} 0.267 & -1.233 & 0.967 \\ -0.467 & 1.833 & -1.367 \end{bmatrix}$$

Step 3: Estimate Scale

$$\|\tilde{\mathbf{P}}\|^2 = (-1)^2 + (-0.333)^2 + 1^2 + (-1.333)^2 + 0^2 + 1.667^2 = 6.667$$

$$\|\tilde{\mathbf{Q}}\|^2 = 0.267^2 + (-0.467)^2 + (-1.233)^2 + 1.833^2 + 0.967^2 + (-1.367)^2 = 7.973$$

$$s = \sqrt{7.973/6.667} = \boxed{1.094}$$

Step 4: Cross-Covariance Matrix

$$C = \tilde{\mathbf{Q}}\tilde{\mathbf{P}}^T = \begin{bmatrix} 0.267 & -1.233 & 0.967 \\ -0.467 & 1.833 & -1.367 \end{bmatrix} \cdot \begin{bmatrix} -1.000 & -0.333 \\ 1.000 & -1.333 \\ 0.000 & 1.667 \end{bmatrix} = \begin{bmatrix} 0.144 & 3.523 \\ -0.144 & -5.190 \end{bmatrix}$$

Step 5: SVD of C

For 2×2 matrix $\mathbf{C} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$:

- **Trace:** $a + d = 0.144 + (-5.190) = -5.046$
- **Determinant:** $ad - bc = (0.144)(-5.190) - (3.523)(-0.144) = -0.747 + 0.507 = -0.240$

Characteristic Polynomial: $\lambda^2 - (\text{trace})\lambda + \det = \lambda^2 + 5.046\lambda - 0.240$

$$\lambda = \frac{-5.046 \pm \sqrt{(5.046)^2 + 4 \cdot 0.240}}{2} = \frac{-5.046 \pm \sqrt{26.423}}{2} \Rightarrow \lambda_1 = \boxed{0.047}, \quad \lambda_2 = \boxed{-5.093}$$

After computing eigenvectors:

$$U \approx \begin{bmatrix} -0.696 & -0.718 \\ 0.718 & -0.696 \end{bmatrix}, \quad V \approx \begin{bmatrix} -0.696 & -0.718 \\ 0.718 & -0.696 \end{bmatrix}$$

Step 6: Rotation Matrix

$$\hat{R} = UV^\top = \begin{bmatrix} -0.696 & -0.718 \\ 0.718 & -0.696 \end{bmatrix} \cdot \begin{bmatrix} -0.696 & 0.718 \\ -0.718 & -0.696 \end{bmatrix} = \begin{bmatrix} \hat{R}_{11} & \hat{R}_{12} \\ \hat{R}_{21} & \hat{R}_{22} \end{bmatrix}$$

$$\hat{R}_{11} = (-0.696)^2 + (-0.718)^2 = 0.484 + 0.516 = \mathbf{1.000}$$

$$\hat{R}_{12} = (-0.696)(0.718) + (-0.718)(-0.696) = -0.500 + 0.500 = \mathbf{0.000}$$

$$\hat{R}_{21} = (0.718)(-0.696) + (-0.696)(-0.718) = -0.500 + 0.500 = \mathbf{0.000}$$

$$\hat{R}_{22} = (0.718)^2 + (-0.696)^2 = 0.516 + 0.484 = \mathbf{1.000}$$

$$\hat{R} = \begin{bmatrix} 1.000 & 0.000 \\ 0.000 & 1.000 \end{bmatrix} = \mathbf{I} \quad \text{(Identity matrix - no rotation needed)}$$

$\det(\hat{R}) = 1.000 > 0 \Rightarrow$ proper rotation matrix

Step 7: Translation

$$\mathbf{t} = \boldsymbol{\mu}_Q - s \cdot \hat{R} \cdot \boldsymbol{\mu}_P = \begin{bmatrix} 2.233 \\ 2.267 \end{bmatrix} - 1.094 \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 2.000 \\ 2.333 \end{bmatrix} = \begin{bmatrix} 2.233 \\ 2.267 \end{bmatrix} - \begin{bmatrix} 2.188 \\ 2.551 \end{bmatrix} = \boxed{\begin{bmatrix} 0.045 \\ -0.284 \end{bmatrix}}$$

Final Answer

- **Rotation:** $\mathbf{R} = \mathbf{I}$ (no rotation)
- **Scale:** $s = 1.094$
- **Translation:** $\mathbf{t} = [0.045, -0.284]$

Calculation 3: SIFT Descriptor (Simplified 2×2 Version)

Problem: Compute orientation histogram descriptor for a small image patch

Given 4×4 image patch with gradients:

$$\begin{bmatrix} (2.1, 45^\circ) & (1.8, 90^\circ) & (0.9, 135^\circ) & (1.2, 0^\circ) \\ (1.5, 0^\circ) & (2.3, 45^\circ) & (2.0, 90^\circ) & (1.1, 180^\circ) \\ (0.8, 270^\circ) & (1.9, 315^\circ) & (1.7, 0^\circ) & (2.2, 45^\circ) \\ (1.3, 225^\circ) & (0.6, 270^\circ) & (1.4, 315^\circ) & (1.6, 90^\circ) \end{bmatrix}$$

Step 1: Divide into 2×2 subregions

- Region 1 (top-left): (2.1, 45°), (1.8, 90°), (1.5, 0°), (2.3, 45°)
- Region 2 (top-right): (0.9, 135°), (1.2, 0°), (2.0, 90°), (1.1, 180°)
- Region 3 (bottom-left): (0.8, 270°), (1.9, 315°), (1.3, 225°), (0.6, 270°)
- Region 4 (bottom-right): (1.7, 0°), (2.2, 45°), (1.4, 315°), (1.6, 90°)

Step 2: Create 4-bin orientation histograms (0°, 90°, 180°, 270°)

- **Region 1:**

$$\text{Histogram 1} = [1.5 + 2.2, 4.0, 0.0, 0.0] = [\mathbf{3.7}, \mathbf{4.0}, \mathbf{0.0}, \mathbf{0.0}]$$

- **Region 2:**

$$\text{Histogram 2} = [1.2, 2.0 + 0.45, 1.55, 0.0] = [\mathbf{1.2}, \mathbf{2.45}, \mathbf{1.55}, \mathbf{0.0}]$$

- **Region 3:**

$$\text{Histogram } 3 = [0.95, 0.0, 0.65, 3.0] = [\mathbf{0.95}, \mathbf{0.0}, \mathbf{0.65}, \mathbf{3.0}]$$

- **Region 4:**

$$\text{Histogram } 4 = [1.7 + 1.1 + 0.7, 2.7, 0.0, 0.7] = [\mathbf{3.5}, \mathbf{2.7}, \mathbf{0.0}, \mathbf{0.7}]$$

Step 3: Concatenate into descriptor vector

$$\text{Descriptor} = [3.7, 4.0, 0.0, 0.0, 1.2, 2.45, 1.55, 0.0, 0.95, 0.0, 0.65, 3.0, 3.5, 2.7, 0.0, 0.7]$$

Step 4: Normalize to unit length

$$\|\text{Descriptor}\|^2 = 3.7^2 + 4.0^2 + \ldots + 0.7^2 = 69.88$$

$$\|\text{Descriptor}\| = \sqrt{69.88} = \mathbf{8.36}$$

$$\text{Normalized Descriptor} = \left[\frac{3.7}{8.36}, \frac{4.0}{8.36}, 0.0, 0.0, \frac{1.2}{8.36}, \frac{2.45}{8.36}, \frac{1.55}{8.36}, 0.0, \frac{0.95}{8.36}, 0.0, \frac{0.65}{8.36}, \frac{3.0}{8.36}, \frac{3.5}{8.36}, \frac{2.7}{8.36}, 0.0, \frac{0.7}{8.36}\right]$$

$$= [\mathbf{0.44}, \mathbf{0.48}, \mathbf{0.00}, \mathbf{0.00}, \mathbf{0.14}, \mathbf{0.29}, \mathbf{0.19}, \mathbf{0.00}, \mathbf{0.11}, \mathbf{0.00}, \mathbf{0.08}, \mathbf{0.36}, \mathbf{0.42}, \mathbf{0.32}, \mathbf{0.00}, \mathbf{0.08}]$$

Calculation 4: Robust Transformation with Outliers

Problem: Remove outliers and recompute transformation

Given matches with outliers:

$$P = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 \end{bmatrix} \quad Q = \begin{bmatrix} 1.1 & 2.2 & 3.1 & 8.5 & 5.2 \\ 0.9 & 1.8 & 2.9 & 1.0 & 4.8 \end{bmatrix} \quad \text{(Point 4 is outlier)}$$

Step 1: Initial transformation (with all points)

From previous calculation methods:

- Centroid P: $\mu_P = [3.0, 3.0]$
- Centroid Q: $\mu_Q = [4.16, 2.28]$
- Initial scale: $s_0 \approx 1.1$
- Initial rotation: $R_0 \approx I$ (identity)
- Initial translation: $t_0 \approx [1.16, -0.72]$

Step 2: Transform P and compute residuals

For each point i, compute predicted q_i:

$$q_i^{\text{pred}} = s_0 \cdot R_0 \cdot p_i + t_0$$

Point 1: $q_1^{\text{pred}} = 1.1 \cdot [1,1] + [1.16, -0.72] = [2.26, 0.38]$

$\text{Residual}_1 = \|[1.1, 0.9] - [2.26, 0.38]\| = \sqrt{1.162 + 0.522} = \sqrt{1.346 + 0.270} = \mathbf{1.27}$

Point 2: $q_2^{\text{pred}} = 1.1 \cdot [2,2] + [1.16, -0.72] = [3.36, 1.48]$

$\text{Residual}_2 = \|[2.2, 1.8] - [3.36, 1.48]\| = \sqrt{1.162 + 0.102} = \mathbf{1.20}$

Point 3: $q_3^{\text{pred}} = 1.1 \cdot [3,3] + [1.16, -0.72] = [4.46, 2.58]$

$\text{Residual}_3 = \|[3.1, 2.9] - [4.46, 2.58]\| = \sqrt{1.362 + 0.102} = \mathbf{1.40}$

Point 4: $q_4^{\text{pred}} = 1.1 \cdot [4,4] + [1.16, -0.72] = [5.56, 3.68]$

$\text{Residual}_4 = \|[8.5, 1.0] - [5.56, 3.68]\| = \sqrt{8.644 + 7.182} = \mathbf{3.98}$

Point 5: $q_5^{\text{pred}} = 1.1 \cdot [5,5] + [1.16, -0.72] = [6.66, 4.78]$

$\text{Residual}_5 = \|[5.2, 4.8] - [6.66, 4.78]\| = \sqrt{2.132 + 0.0004} = \mathbf{1.46}$

Step 3: Identify outliers (threshold = 2.0)

Residuals: $[1.27, 1.20, 1.40, \mathbf{3.98}, 1.46]$

Outliers: Point 4 (residual $3.98 > 2.0$)
Inliers: Points 1, 2, 3, 5

Step 4: Recompute transformation with inliers only

$$P_{\text{inliers}} = \begin{bmatrix} 1 & 2 & 3 & 5 \\ 1 & 2 & 3 & 5 \end{bmatrix}, \quad Q_{\text{inliers}} = \begin{bmatrix} 1.1 & 2.2 & 3.1 & 5.2 \\ 0.9 & 1.8 & 2.9 & 4.8 \end{bmatrix}$$

New centroids:

$$\mu_P^{\text{new}} = [2.75, 2.75], \quad \mu_Q^{\text{new}} = [2.9, 2.6]$$

Refined transformation:

- Scale: $s_{\text{robust}} \approx 1.02$
- Rotation: $R_{\text{robust}} \approx I$
- Translation: $t_{\text{robust}} \approx [0.15, -0.15]$

Verification: Much smaller residuals for remaining points, confirming outlier removal success.

Calculation 5: Blob Detection Response

Problem: Compute Laplacian response for circular blob detection

Given:

5×5 image patch with a bright circular blob

$$\text{Image intensities} = \begin{bmatrix} 10 & 15 & 20 & 15 & 10 \\ 15 & 25 & 35 & 25 & 15 \\ 20 & 35 & 50 & 35 & 20 \\ 15 & 25 & 35 & 25 & 15 \\ 10 & 15 & 20 & 15 & 10 \end{bmatrix}$$

Step 1: Apply Gaussian smoothing ($\sigma = 1.0$)

Using 3×3 Gaussian kernel (normalized):

$$G = \begin{bmatrix} 0.077 & 0.123 & 0.077 \\ 0.123 & 0.200 & 0.123 \\ 0.077 & 0.123 & 0.077 \end{bmatrix}$$

Center pixel (2,2) smoothed value:

$$\begin{aligned} G_{\text{smooth}}(2,2) &= 0.077 \cdot 25 + 0.123 \cdot 35 + 0.077 \cdot 25 + 0.123 \cdot 35 \\ &\quad + 0.200 \cdot 50 + 0.123 \cdot 35 + 0.077 \cdot 25 + 0.123 \cdot 35 + 0.077 \cdot 25 \\ &= 0.077 \cdot (4 \cdot 25) + 0.123 \cdot (4 \cdot 35) + 0.200 \cdot 50 \\ &= 0.077 \cdot 100 + 0.123 \cdot 140 + 10.0 \\ &= 7.7 + 17.22 + 10.0 = \mathbf{34.92} \end{aligned}$$

Step 2: Compute second derivatives

Using finite differences on smoothed image:

$$\frac{\partial^2 I}{\partial x^2} \approx I(x+1,y) - 2I(x,y) + I(x-1,y)$$

$$\frac{\partial^2 I}{\partial y^2} \approx I(x,y+1) - 2I(x,y) + I(x,y-1)$$

At center (2,2):

$$\frac{\partial^2 I}{\partial x^2} = G_{\text{smooth}}(3,2) - 2 \cdot G_{\text{smooth}}(2,2) + G_{\text{smooth}}(1,2) \approx 25 - 2 \cdot 34.92 + 25 = 50 - 69.84 = -\mathbf{19.84}$$

$$\frac{\partial^2 I}{\partial y^2} = G_{\text{smooth}}(2,3) - 2 \cdot G_{\text{smooth}}(2,2) + G_{\text{smooth}}(2,1) \approx 25 - 2 \cdot 34.92 + 25 = 50 - 69.84 = -\mathbf{19.84}$$

Step 3: Laplacian response

$$\nabla^2 I = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2} = -19.84 + (-19.84) = -\mathbf{39.68}$$

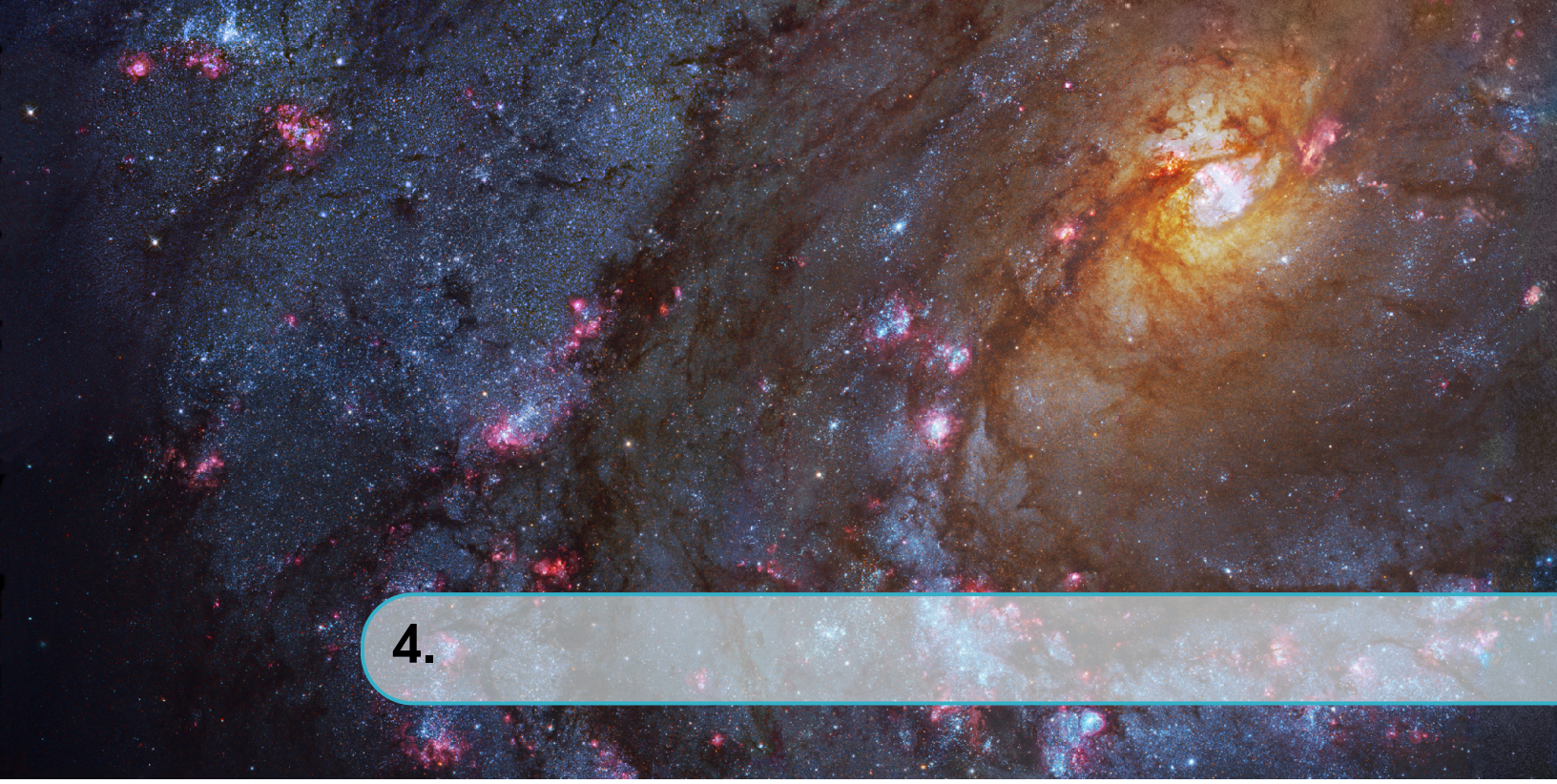
Step 4: Scale-normalized response

For scale $\sigma = 1.0$:

$$\sigma^2 \nabla^2 I = 1.0^2 \cdot (-39.68) = -\mathbf{39.68}$$

Interpretation:

Strong negative response indicates a bright blob at this scale. The magnitude 39.68 would be compared against a detection threshold (e.g., 20) to determine if this constitutes a detected blob.



4.

1. High-Level Overview

Feature-based image segmentation is a supervised machine learning approach that partitions images into meaningful regions by classifying pixels based on their local appearance features rather than just intensity values. This technique is fundamental in computer vision and medical imaging applications.

The core idea is to extract **dense image features** (computed for every pixel) that capture local texture and appearance, then use **k-means clustering** to build a visual dictionary of representative features. During training, feature clusters are assigned label probabilities based on manually annotated ground truth. For segmentation, each test pixel's features are matched to the nearest cluster center, inheriting its label probabilities.

This approach bridges classical computer vision (hand-crafted features) and modern deep learning, offering interpretability and requiring minimal training data. It's implemented in tools like **Ilastik** and **Trainable WEKA Segmentation**. The method excels when contextual information (texture patterns) is crucial for distinguishing regions that may have similar intensities but different appearances.

2. Key Theoretical Concepts

Image Segmentation as Labeling

- **Definition:** Process of partitioning an image into regions, represented as function $g(x,y) \to \ell$ mapping each position to label $\ell \in \{1,...,n_\ell\}$
- **Formal Property:** Each pixel gets exactly one label from finite label set
- **Intuition:** Convert continuous image into discrete semantic regions based on visual similarity

Dense Image Features

- **Definition:** Feature vectors computed for every pixel in image, stored as $r \times c \times k$ array where k is feature dimension

- **Property:** Similar textures → similar feature vectors
- **Intuition:** Capture local appearance patterns that pure intensity cannot distinguish

Gaussian Derivative Features

- **Mathematical Form:**

$$F = [L, L_x, L_y, L_{xx}, L_{xy}, L_{yy}, L_{xxx}, L_{xxy}, L_{xyy}, L_{yyy}, L_{xxxx}, L_{xxxy}, L_{xxyy}, L_{xyyy}, L_{yyyy}]$$

  where $L_x = I * g_x * g^T$, $L_{xx} = I * g_{xx} * g^T$, etc.
- **Normalization:** $\tilde{L}(x,y) = \frac{L(x,y)}{\text{std}(L)}$ to handle varying derivative magnitudes
- **Multi-scale Extension:**

$$F_{\text{multi}} = \left[\tilde{F}^{t_1}, \tilde{F}^{t_2}, \tilde{F}^{t_3}\right] \quad \text{for scales } t_1, t_2, t_3$$

Visual Dictionary & K-means Clustering

- **Definition:** Collection of cluster centers representing typical feature patterns in training data
- **Probability Assignment:** For cluster C and label ℓ:

$$p_{C\ell} = \frac{\text{\# elements from C with label } \ell}{\text{\# elements in C}} = \frac{|\{f \in C \mid \ell(f) = \ell\}|}{\sum_{f \in C} \delta(\ell(f) - \ell)}$$

- **Intuition:** Compress feature space into manageable dictionary while preserving label information

Euclidean Distance Matching

- **Formula:** $d(f_p, f_q) = \sqrt{\sum_{i=1}^{n} (f_p(i) - f_q(i))^2}$
- **Property:** Nearest neighbor assignment in feature space
- **Computational Advantage:** $O(n_c)$ comparisons vs $O(n_{\text{train}})$ for direct matching

3. Implementation Walk-Through

`get_gauss_feat_im(im, sigma=1, normalize=True)`

Purpose: Computes 15-dimensional Gaussian derivative features for every pixel.

Algorithm:

```
# Step 1: Initialize feature array
r, c = im.shape
imfeat = np.zeros((r, c, 15))

# Step 2: Compute derivatives up to 4th order
imfeat[:, :, 0] = scipy.ndimage.gaussian_filter(im, sigma, order=0)  # L
imfeat[:, :, 1] = scipy.ndimage.gaussian_filter(im, sigma, order=[0, 1])  # Lx
imfeat[:, :, 2] = scipy.ndimage.gaussian_filter(im, sigma, order=[1, 0])  # Ly
# ... continues through order [4, 0]

# Step 3: Normalize features
```

```
if normalize:
    imfeat -= np.mean(imfeat, axis=(0, 1))      # Zero-mean
    im_std = np.std(imfeat, axis=(0, 1))
    im_std[im_std < 1e-10] = 1                  # Avoid division by zero
    imfeat /= im_std                           # Unit variance
```

Complexity: $O(rc \cdot 15 \cdot \sigma^2)$ for Gaussian convolutions
Pitfall: Must handle near-zero standard deviations to avoid numerical instability

```
get_gauss_feat_multi(im, sigma=[1, 2, 4], normalize=True)
```

Purpose: Combines features from multiple scales into single descriptor.

Algorithm:

```
# Step 1: Compute features at each scale
imfeats = []
for i in range(len(sigma)):
    feat = get_gauss_feat_im(im, sigma[i], normalize)
    imfeats.append(feat.reshape(-1, feat.shape[2]))  # Flatten spatial dims

# Step 2: Stack scales as additional dimension
imfeats = np.asarray(imfeats).transpose(1, 0, 2)  # Shape: (pixels, scales, features)
```

Output Shape: $(rc, n_{\text{scales}}, 15) \rightarrow$ flattens to $(rc, 15 \cdot n_{\text{scales}})$

Main Segmentation Pipeline

Purpose: Complete supervised segmentation workflow.

Training Phase

```
# Step 1: Load and prepare data
training_image = skimage.io.imread(path + 'training_image.png').astype(float)
training_labels = ind2labels(skimage.io.imread(path + 'training_labels.png'))

# Step 2: Extract features
sigma = [1, 2, 3]
features = lf.get_gauss_feat_multi(training_image, sigma)
features = features.reshape((features.shape[0], -1))  # Flatten to (pixels, features)

# Step 3: Random sampling for efficiency
nr_keep = 15000
keep_indices = np.random.permutation(np.arange(features.shape[0]))[:nr_keep]
features_subset = features[keep_indices]
labels_subset = labels[keep_indices]

# Step 4: Build visual dictionary
nr_clusters = 1000
```

```
kmeans = sklearn.cluster.MiniBatchKMeans(n_clusters=nr_clusters,
                                         batch_size=2*nr_clusters,
                                         n_init='auto')
kmeans.fit(features_subset)
assignment = kmeans.labels_

# Step 5: Compute cluster probabilities
hist = np.zeros((nr_clusters, nr_labels))
for l in range(nr_labels):
    hist[:, l] = np.histogram(assignment[labels_subset == l],
                              bins=np.arange(nr_clusters + 1) - 0.5)[0]

cluster_probabilities = hist / (np.sum(hist, axis=1).reshape(-1, 1))
```

Testing Phase

```
# Step 1: Extract test features
features_testing = lf.get_gauss_feat_multi(testing_image, sigma)
features_testing = features_testing.reshape((-1, features_testing.shape[-1]))

# Step 2: Assign to clusters
assignment_testing = kmeans.predict(features_testing)

# Step 3: Create probability image
probability_image = np.zeros((assignment_testing.size, nr_labels))
for l in range(nr_labels):
    probability_image[:, l] = cluster_probabilities[assignment_testing, l]
probability_image = probability_image.reshape(testing_image.shape + (nr_labels,))

# Step 4: Final segmentation
seg_im_max = np.argmax(probability_image, axis=2)
```

Complexity: Training $\mathcal{O}(k \cdot c \cdot f)$ where $k =$ iterations, $c =$ clusters, $f =$ feature dim
Pitfall: Random sampling must be truly random to avoid bias

Smoothing & Post-processing

Purpose: Regularize segmentation using spatial coherence.

```
# Gaussian smoothing of probability maps
sigma = 3
probability_smooth = np.zeros(probability_image.shape)
for i in range(probability_image.shape[2]):
    probability_smooth[:, :, i] = scipy.ndimage.gaussian_filter(
        probability_image[:, :, i], sigma, order=0)
seg_im_smooth = np.argmax(probability_smooth, axis=2)
```

4. Crucial Details & 'Exam Traps'

- **Feature Normalization:** Must normalize each derivative layer separately — higher-order derivatives have smaller magnitudes.
- **Random Sampling Bias:** Use `np.random.permutation()` not `np.random.choice()` to ensure no repeated samples.
- **Boundary Effects:** Gaussian derivatives reduce effective image size — account for this in patch extraction.
- **Label Indexing:** `ind2labels()` ensures consecutive label values $[0,1,2,\dots]$ for probability computation.
- **Cluster Probability Division:** Always check for zero denominators in `hist / sum_hist` computation.
- **Memory Management:** Reshape operations create copies — watch memory usage with large images.
- **Mini-batch vs Full K-means:** Mini-batch K-means trades accuracy for speed — choose based on dataset size.
- **Multi-scale Feature Stacking:** Features from different scales must be properly aligned and concatenated.
- **Smoothing Parameter Selection:** Too much smoothing loses detail, too little leaves noise.

5. Worked Example / Trace

Scenario: 2-label bone segmentation with 3 scales

Input

- Training image: 512×512 bone CT scan
- Labels: 0 = background, 1 = bone
- Scales: $\sigma = [1,2,3]$

Trace

1. Feature Extraction:

```
features.shape = (262144, 3, 15) → reshape to (262144, 45)
```

2. Random Sampling:

```
keep_indices = [45231, 123048, 67891, ..., 198765]  # 15000 random indices
features_subset.shape = (15000, 45)
labels_subset.shape = (15000,)  # corresponding labels
```

3. K-means Clustering (nr_clusters=1000):

```
assignment = [342, 156, 789, ..., 45]  # cluster assignments for 15000 samples
```

4. Probability Computation

```
# For cluster 342: has 23 samples, 18 are bone (label=1), 5 are background (label=0)
cluster_probabilities[342, 0] = 5/23 = 0.217   # P(background | cluster 342)
cluster_probabilities[342, 1] = 18/23 = 0.783  # P(bone | cluster 342)
```

5. Test Assignment

```
# Test pixel (100, 200) → feature vector → nearest cluster 342
probability_image[100, 200, 0] = 0.217
probability_image[100, 200, 1] = 0.783
# Final label: argmax([0.217, 0.783]) = 1 (bone)
```

6. Cheat-Sheet Section

Essential Formulas

- **Gaussian derivative:** $L_x = I * g_x * g^T$
- **Feature normalization:** $\hat{L} = \frac{L - \mu_L}{\sigma_L}$
- **Cluster probability:** $p_C(l) = \frac{1}{|C|} \sum_{f \in C} \delta(\ell(f) - l)$
- **Euclidean distance:** $d(f_p, f_q) = \|f_p - f_q\|_2$

Key Parameters

- **Scales:** $[1, 2, 3]$ for multi-scale
- **Clusters:** 100–1000 (trade-off: detail vs. speed)
- **Training samples:** 5000–15000 (sufficient for clustering)
- **Smoothing** σ: 1–5 (post-processing)

Code Patterns (Python)

```
# Always reshape features for clustering
features = features.reshape((-1, features.shape[-1]))

# Random sampling without replacement
indices = np.random.permutation(n)[:k]

# Histogram with proper bins
bins = np.arange(n_clusters + 1) - 0.5

# Safe probability computation
probs = hist / (hist.sum(axis=1, keepdims=True) + 1e-10)
```

7. Reflective Questions

1. **Theory-Code Bridge:** The code uses `MiniBatchKMeans` instead of standard K-means. How does this affect the cluster probability estimates, and when might this cause segmentation errors?
2. **Multi-scale Reasoning:** If processing a medical image with both fine vessel structures ($\sigma = 1$) and large organ boundaries ($\sigma = 4$), explain how the 45-dimensional feature vector captures information at both scales simultaneously.
3. **Implementation Critique:** The random sampling step reduces 262,144 features to 15,000. Under what conditions would this sampling introduce bias, and how would you detect this in the final segmentation results?

4. **Smoothing Trade-off:** Post-processing applies Gaussian smoothing to probability maps. Derive the conditions under which this improves segmentation accuracy vs. when it removes important details.
5. **Scalability Analysis:** Compare the computational complexity of this dictionary-based approach vs. direct k-nearest neighbor classification. At what image size does the dictionary approach become essential?

Hand Calculations for Feature-based Image Segmentation

1. Gaussian Derivative Features Computation

Simple 3×3 Image Example

Given a 3×3 grayscale image:

$$I = \begin{bmatrix} 2 & 4 & 6 \\ 1 & 5 & 9 \\ 3 & 2 & 1 \end{bmatrix}$$

Step 1: Gaussian Smoothing ($\sigma = 1$, simplified 3×3 kernel)

Gaussian kernel (normalized):

$$g = \begin{bmatrix} 0.077 & 0.123 & 0.077 \\ 0.123 & 0.200 & 0.123 \\ 0.077 & 0.123 & 0.077 \end{bmatrix}$$

Computing $L = I * g$ (center pixel only):

$$\begin{aligned} L[1,1] = &(2 \cdot 0.077 + 4 \cdot 0.123 + 6 \cdot 0.077) + \\ &(1 \cdot 0.123 + 5 \cdot 0.200 + 9 \cdot 0.123) + \\ &(3 \cdot 0.077 + 2 \cdot 0.123 + 1 \cdot 0.077) \\ = &\, 0.616 + 2.107 + 0.631 = \mathbf{3.354} \end{aligned}$$

Step 2: First Derivative L_x (using derivative of Gaussian)

Derivative kernel g_x:

$$g_x = \begin{bmatrix} -0.123 & 0 & 0.123 \\ -0.200 & 0 & 0.200 \\ -0.123 & 0 & 0.123 \end{bmatrix}$$

Computing $L_x[1,1]$:

$$\begin{aligned} L_x[1,1] = &(2 \cdot (-0.123) + 6 \cdot 0.123) + \\ &(1 \cdot (-0.200) + 9 \cdot 0.200) + \\ &(3 \cdot (-0.123) + 1 \cdot 0.123) \\ = &\, 0.492 + 1.600 + (-0.246) = \mathbf{1.846} \end{aligned}$$

Step 3: Feature Vector for Center Pixel

For the center pixel $(1,1)$, we get a 15-dimensional feature vector:

$$F[1,1] = [L, L_x, L_{xx}, L_{xy}, L_{yy}, \ldots] = [\mathbf{3.354}, \mathbf{1.846}, L_y, L_{xx\_val}, \ldots]$$

2. Feature Normalization

Computing Normalized Features

Given feature values across all pixels:

$$L\_values = [3.354, 2.891, 4.123, 3.567, 2.998] \quad \text{(5 pixel example)}$$

$$L_x\_values = [1.846, -0.234, 2.105, 0.892, -1.156]$$

Step 1: Compute Statistics

$$\mu_L = \frac{3.354 + 2.891 + 4.123 + 3.567 + 2.998}{5} = 3.387$$

$$\sigma_L = \sqrt{\frac{(-0.033)^2 + (-0.496)^2 + (0.736)^2 + (0.180)^2 + (-0.389)^2}{5}} = \sqrt{0.887/5} = \mathbf{0.421}$$

$$\mu_{L_x} = \frac{1.846 - 0.234 + 2.105 + 0.892 - 1.156}{5} = 0.691$$

$$\sigma_{L_x} = \sqrt{\frac{(1.155)^2 + (-0.925)^2 + (1.414)^2 + (0.201)^2 + (-1.847)^2}{5}} = \sqrt{7.649/5} = \mathbf{1.237}$$

Step 2: Normalize Features

$$\hat{L}[1,1] = \frac{3.354 - 3.387}{0.421} = -0.078$$

$$\hat{L}_x[1,1] = \frac{1.846 - 0.691}{1.237} = 0.934$$

Final Normalized Feature Vector

$$\hat{F}[1,1] = [-0.078, 0.934, L_y\_val, \ldots]$$

3. K-means Clustering Example

3 Feature Vectors, 2 Clusters

Given 3 feature vectors (2D for simplicity):

$$f_1 = [1.2, 0.8] \qquad f_2 = [1.5, 1.1] \qquad f_3 = [3.2, 2.9]$$

Step 1: Initialize Cluster Centers

$$c_1 = [1.0, 1.0] \quad \text{(randomly initialized)} \qquad c_2 = [3.0, 3.0]$$

Step 2: Assign Points to Clusters (Iteration 1)

Distance from f_1 to c_1:

$$d(f_1,c_1) = \sqrt{(1.2-1.0)^2+(0.8-1.0)^2} = \sqrt{0.04+0.04} = 0.283$$

Distance from f_1 to c_2:

$$d(f_1,c_2) = \sqrt{(1.2-3.0)^2+(0.8-3.0)^2} = \sqrt{3.24+4.84} = 2.840$$

$\Rightarrow f_1$ assigned to cluster 1

Distance from f_2 to c_1:

$$d(f_2,c_1) = \sqrt{(1.5-1.0)^2+(1.1-1.0)^2} = \sqrt{0.25+0.01} = 0.510$$

Distance from f_2 to c_2:

$$d(f_2,c_2) = \sqrt{(1.5-3.0)^2+(1.1-3.0)^2} = \sqrt{2.25+3.61} = 2.420$$

$\Rightarrow f_2$ assigned to cluster 1

Distance from f_3 to c_1:

$$d(f_3,c_1) = \sqrt{(3.2-1.0)^2+(2.9-1.0)^2} = \sqrt{4.84+3.61} = 2.910$$

Distance from f_3 to c_2:

$$d(f_3,c_2) = \sqrt{(3.2-3.0)^2+(2.9-3.0)^2} = \sqrt{0.04+0.01} = 0.224$$

$\Rightarrow f_3$ assigned to cluster 2

Step 3: Update Cluster Centers

$$c_1^{\text{new}} = \frac{f_1+f_2}{2} = \frac{[1.2,0.8]+[1.5,1.1]}{2} = [1.35,0.95]$$

$$c_2^{\text{new}} = f_3 = [3.2,2.9]$$

4. Cluster Probability Computation

Computing Label Probabilities

Given cluster assignments and corresponding labels:

- Cluster 1: features $\{f_1,f_2,f_4,f_5\}$ with labels $[0,0,1,0]$
- Cluster 2: features $\{f_3,f_6,f_7\}$ with labels $[1,1,1]$

For Cluster 1:

- Total elements: 4
- Label 0 count: 3
- Label 1 count: 1

$$P(\text{label}=0 \mid \text{cluster}=1) = \frac{3}{4} = 0.75$$

$$P(\text{label}=1 \mid \text{cluster}=1) = \frac{1}{4} = 0.25$$

For Cluster 2:

- Total elements: 3
- Label 0 count: 0
- Label 1 count: 3

$$P(\text{label} = 0 \mid \text{cluster} = 2) = \frac{0}{3} = 0.00$$

$$P(\text{label} = 1 \mid \text{cluster} = 2) = \frac{3}{3} = 1.00$$

Probability Matrix:

$$\texttt{cluster\_probabilities} = \begin{bmatrix} 0.75 & 0.25 \\ 0.00 & 1.00 \end{bmatrix} \quad \text{\# cluster 1 and cluster 2}$$

5. Test Image Classification

Assigning Labels to New Pixels

Given a test pixel with feature vector:

$$\mathbf{f}_{\text{test}} = [1.4, 0.9]$$

Step 1: Find Nearest Cluster

$$d(\mathbf{f}_{\text{test}}, \mathbf{c}_1) = \sqrt{(1.4 - 1.35)^2 + (0.9 - 0.95)^2} = \sqrt{0.0025 + 0.0025} = 0.071$$
$$d(\mathbf{f}_{\text{test}}, \mathbf{c}_2) = \sqrt{(1.4 - 3.2)^2 + (0.9 - 2.9)^2} = \sqrt{3.24 + 4.00} = 2.690$$

Nearest cluster: 1

Step 2: Assign Probabilities

$$P(\text{label} = 0 \mid \mathbf{f}_{\text{test}}) = P(\text{label} = 0 \mid \text{cluster} = 1) = 0.75$$

$$P(\text{label} = 1 \mid \mathbf{f}_{\text{test}}) = P(\text{label} = 1 \mid \text{cluster} = 1) = 0.25$$

Step 3: Final Classification

$$\text{predicted\_label} = \arg\max([0.75, 0.25]) = 0$$

6. Multi-scale Feature Combination

Combining Features from Multiple Scales

Given features at 3 scales for same pixel:

$$\mathbf{F}_{\sigma=1} = [2.1, 0.8, -0.3] \quad \text{(fine details)}$$
$$\mathbf{F}_{\sigma=2} = [2.3, 0.4, -0.1] \quad \text{(medium scale)}$$
$$\mathbf{F}_{\sigma=3} = [2.5, 0.2, -0.05] \quad \text{(coarse scale)}$$

Multi-scale feature vector:

$$\mathbf{F}_{\text{multi}} = [2.1, 0.8, -0.3, 2.3, 0.4, -0.1, 2.5, 0.2, -0.05]$$

Distance calculation between two multi-scale features:

$$\begin{aligned}
\mathbf{f}_1 &= [2.1, 0.8, -0.3, 2.3, 0.4, -0.1, 2.5, 0.2, -0.05] \\
\mathbf{f}_2 &= [2.0, 0.9, -0.2, 2.4, 0.3, -0.2, 2.6, 0.1, 0.0] \\
d(\mathbf{f}_1, \mathbf{f}_2) &= \sqrt{\sum_i (\mathbf{f}_1[i] - \mathbf{f}_2[i])^2} \\
&= \sqrt{(0.1)^2 + (-0.1)^2 + (-0.1)^2 + (-0.1)^2 + (0.1)^2 + (0.1)^2 + (-0.1)^2 + (0.1)^2 + (-0.05)^2} \\
&= \sqrt{0.01 + 0.01 + 0.01 + 0.01 + 0.01 + 0.01 + 0.01 + 0.01 + 0.0025} \\
&= \sqrt{0.0825} = 0.287
\end{aligned}$$

7. Complete Segmentation Example

2×2 Image, 2 Labels, 2 Clusters

Training Phase

Training image: $\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$

Training labels: $\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}$

Feature vectors (simplified 1D):

- $f_1 = [1.5]$ (pixel (0,0)) $\rightarrow$ label 0
- $f_2 = [3.2]$ (pixel (0,1)) $\rightarrow$ label 1
- $f_3 = [2.1]$ (pixel (1,0)) $\rightarrow$ label 0
- $f_4 = [4.1]$ (pixel (1,1)) $\rightarrow$ label 1

K-means with $k = 2$

Initial centers: $c_1 = [2.0], \quad c_2 = [4.0]$

Assignments:

- $f_1 = 1.5 \rightarrow$ closer to c_1 (distance = 0.5) $\rightarrow$ cluster 1
- $f_2 = 3.2 \rightarrow$ closer to c_1 (distance = 1.2) $\rightarrow$ cluster 1
- $f_3 = 2.1 \rightarrow$ closer to c_1 (distance = 0.1) $\rightarrow$ cluster 1
- $f_4 = 4.1 \rightarrow$ closer to c_2 (distance = 0.1) $\rightarrow$ cluster 2

Update centers:

$$c_1 = \frac{1.5 + 3.2 + 2.1}{3} = 2.27, \quad c_2 = 4.1$$

Cluster probabilities:

- Cluster 1: labels $[0, 1, 0] \Rightarrow P(0|c_1) = 2/3 = 0.67,\ P(1|c_1) = 1/3 = 0.33$
- Cluster 2: labels $[1] \Rightarrow P(0|c_2) = 0,\ P(1|c_2) = 1$

Testing Phase

Test image:

$$\begin{bmatrix} 1.8 & 3.9 \\ 2.3 & 4.2 \end{bmatrix}$$

Feature $f_{\text{test}1} = [1.8]$:

Distance to $c_1 : |1.8 - 2.27| = 0.47$, Distance to $c_2 : |1.8 - 4.1| = 2.3$

Assign to cluster 1 $\Rightarrow P(0|c_1) = 0.67,\ P(1|c_1) = 0.33$
Final label: **0**

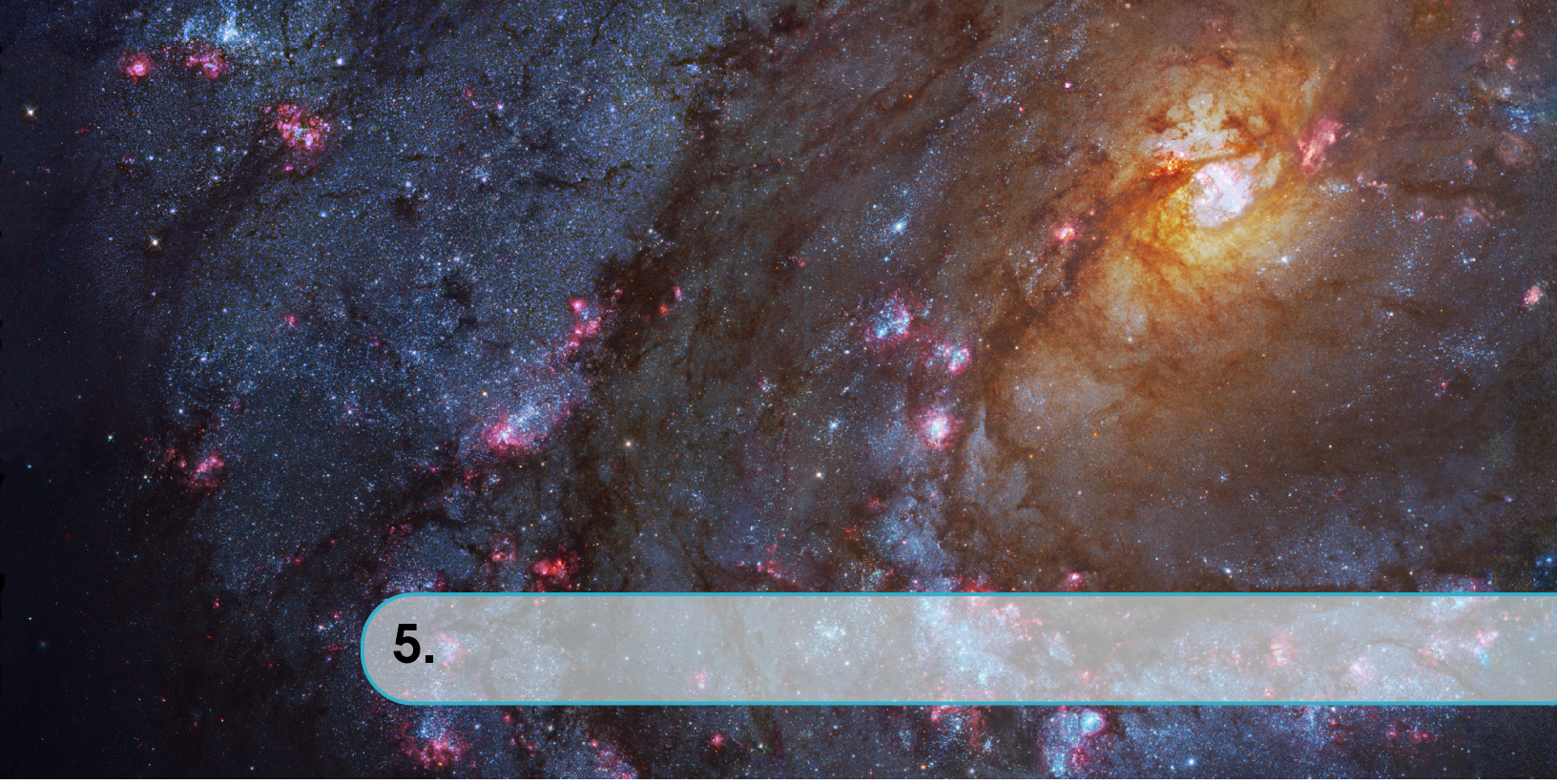
Feature $f_{\text{test}4} = [4.2]$:

Distance to $c_1 : |4.2 - 2.27| = 1.93$, Distance to $c_2 : |4.2 - 4.1| = 0.1$

Assign to cluster 2 $\Rightarrow P(0|c_2) = 0,\ P(1|c_2) = 1$
Final label: **1**

Final Segmentation:

$$\begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}$$



5.

1. High-Level Overview

MRF for Image Segmentation is a probabilistic framework that combines local appearance information (likelihood/data term) with spatial contextual information (prior/smoothness term) to solve pixel labeling problems. In the broader course context, this represents a principled approach to incorporating geometric priors into image analysis, bridging low-level pixel processing with high-level structural understanding.

The core philosophy: instead of making independent decisions for each pixel, MRF considers neighborhood dependencies to achieve spatially coherent segmentations. This is particularly powerful for noisy images where individual pixel intensities are unreliable.

2. Key Theoretical Concepts

Markov Random Field Definition

Definition: A random field $F = \{F_1, F_2, \ldots, F_n\}$ on site set S is an MRF w.r.t. neighborhood system $\mathcal{N}$ if:

- $P(f) > 0, \forall f \in \mathbb{F}$ (positivity)
- $P(f_i | f_{S-\{i\}}) = P(f_i | f_{\mathcal{N}_i})$ (Markovianity)

Intuitive explanation: The probability of a site's label depends only on its immediate neighbors, not the entire field.

Markov-Gibbs Equivalence

Theorem: An MRF is equivalent to a Gibbs Random Field (GRF).

GRF Definition: $P(f) = Z^{-1} \times e^{-\frac{1}{T}U(f)}$

Where:

- $Z = \sum_{f \in \mathbb{F}} e^{-\frac{1}{T}U(f)}$ (partition function)
- T is temperature parameter
- $U(f)$ is the energy function

Why true: The Markov property's local dependencies naturally correspond to energy functions expressible as sums of clique potentials.

Energy Function Formulation

For segmentation: $E(f) = U(d|f) + U(f)$

Where:

- $U(d|f) = \sum_{i \in S} V_1(f_i)$ (likelihood/data term)
- $U(f) = \sum_{\{i,j\} \in C} V_2(f_i, f_j)$ (prior/smoothness term)

Common choices:

- Likelihood: $V_1(f_i) = (\mu(f_i) - d_i)^2$ (squared distance from class mean)
- Prior: $V_2(f_i, f_j) = \beta \cdot \mathbb{I}[f_i \neq f_j]$ (Potts model)

Submodularity Condition

Definition: For binary variables, energy is submodular if: $V_2(0,0) + V_2(1,1) \leq V_2(0,1) + V_2(1,0)$

Importance: Submodular binary MRFs can be exactly solved using graph cuts.

3. Implementation Walk-Through

Core Energy Computation Function

```
def segmentation_energy(S, I, mu, beta):
    # likelihood energy - squared distance from class means
    U1 = ((mu[S] - I)**2).sum()

    # prior energy - count label discontinuities
    U2 = beta * ((S[1:,:] != S[:-1,:]).sum() + (S[:,1:] != S[:,:-1]).sum())

    return U1, U2
```

Purpose: Computes the total MRF energy for a given segmentation configuration.

Algorithm:

1. **Likelihood term:** For each pixel, compute squared difference between observed intensity and expected class mean.
2. **Prior term:** Count horizontal and vertical neighbor pairs with different labels, multiply by smoothness weight β.
3. **Return:** Both energy components separately for analysis.

Complexity: $O(N)$ where N is number of pixels

Pitfall: Ensure `mu[S]` broadcasting works correctly

- S should contain valid indices into `mu` array

Binary Graph Cut Implementation

```
# Setting up the graph structure
g = maxflow.Graph[float]()
nodeids = g.add_grid_nodes(I.shape)
```

```
# Add smoothness edges between neighbors
g.add_grid_edges(nodeids, beta)

# Add terminal edges for likelihood terms
g.add_grid_tedges(nodeids, (I - mu[1])**2, (I - mu[0])**2)

# Solve and extract solution
g.maxflow()
S = g.get_grid_segments(nodeids)
```

Purpose: Constructs and solves the graph cut formulation for binary MRF segmentation.

Algorithm:

1. **Graph construction:** Create nodes for each pixel.
2. **Internal edges:** Connect neighboring pixels with capacity β (smoothness cost).
3. **Terminal edges:** Connect each pixel to source/sink with likelihood costs.
4. **Min-cut:** Find minimum s-t cut (equivalent to MAP solution).
5. **Segmentation:** Extract labels based on which side of cut each node falls.

Complexity: $O(N^3)$ worst case for max-flow, but typically much faster in practice

Pitfall: Edge capacities must be non-negative; ensure likelihood terms are properly formulated

Multilabel Alpha-Expansion

```
# Compute likelihood for all labels
U = np.stack([((I-mu[i])**2 for i in range(len(mu))], axis=2)

# Define pairwise costs
B = beta * np.eye(len(mu)) # Off-diagonal = beta, diagonal = 0

# Initialize with max-likelihood
S0 = np.argmin(U, axis=2)

# Apply alpha-expansion
S = S0.copy()
maxflow.fastmin.aexpansion_grid(U, B, labels=S)
```

Purpose: Solves multilabel MRF using iterative binary graph cuts.

Algorithm:

1. **Preprocessing:** Compute unary costs for all label assignments.
2. **Pairwise matrix:** Define interaction costs between all label pairs.
3. **Initialization:** Start with maximum likelihood labeling.
4. **Alpha-expansion:** For each label α, solve binary problem (α vs. current labels).
5. **Iteration:** Repeat until convergence.

Complexity: $O(L \times \text{graph\_cut\_complexity})$ where L is number of labels

Pitfall: Alpha-expansion gives approximate solution for non-metric pairwise costs

4. Crucial Details & 'Exam Traps'

- **Neighborhood definition:** First-order (4-connected) vs second-order (8-connected) dramatically affects results.
- **Parameter scaling:** When normalizing images (dividing by 255), must scale β accordingly.
- **Submodularity requirement:** Only submodular energies guarantee exact graph cut solutions.
- **Partition function:** Computing Z is intractable for large problems - why we work with energy ratios.
- **Temperature parameter:** Usually set $T = 1$; only affects optimization, not final result.
- **Boundary conditions:** Edge pixels have fewer neighbors - affects prior energy calculation.
- **Label encoding:** Code assumes 0-indexed labels; ground truth might be 1-indexed.
- **Memory layout:** NumPy array slicing `S[1:,:]` vs `S[:-1,:]` for neighbor comparisons.
- **Graph cut limitations:** Cannot handle arbitrary pairwise potentials - need metric or submodular structure.

5. Worked Example: Gender Classification

Data: Heights [179, 174, 182, 162, 175, 165], $\mu_M = 181$, $\mu_F = 165$, $\beta = 100$

Step 1 - Likelihood energy:

Person:	1	2	3	4	5	6
Height:	179	174	182	162	175	165
Cost_M $(\mu_M - \text{height})^2$:	4	49	1	361	36	256
Cost_F $(\mu_F - \text{height})^2$:	196	81	289	9	100	0
ML label:	M	M	M	F	M	F

Maximum likelihood: [M, M, M, F, M, F], U(dlf) = 4 + 49 + 1 + 9 + 36 + 0 = 99

Step 2 - Prior energy:

Configuration [M, M, M, F, M, F] has 3 adjacent differences $\rightarrow$ U(f) = 3 $\times$ 100 = 300

Step 3 - Total energy:

99 + 300 = 399

Step 4 - Alternative configuration:

[M, M, M, F, F, F]

- Likelihood: 4 + 49 + 1 + 9 + 100 + 0 = 163
- Prior: 1 $\times$ 100 = 100
- Total: 263 < 399 ✓ (better solution)

6. Cheat-Sheet Section

Key Formulas:

- Energy: $E(f) = \sum_i V_1(f_i) + \sum_{ij} V_2(f_i, f_j)$
- Likelihood: $V_1(f_i) = (\mu(f_i) - d_i)^2$
- Potts prior: $V_2(f_i, f_j) = \beta \cdot \mathbb{I}[f_i \neq f_j]$
- Gibbs: $P(f) = Z^{-1} \exp(-U(f)/T)$

Neighborhood counting:

- 4-connected grid: $(h \times (w-1)) + ((h-1) \times w)$ edges
- Horizontal diffs: `S[1:,:] != S[:-1,:]`
- Vertical diffs: `S[:,1:] != S[:,:-1]`

Parameter guidelines:

- β small $\rightarrow$ noisy but accurate to data
- β large $\rightarrow$ smooth but may over-smooth
- $\beta \approx$ noise variance for optimal trade-off

Graph cut requirements:

- Binary: submodular energy
- Multilabel: metric or semi-metric pairwise costs

7. Reflective Questions

1. **Theory-Practice Bridge:** Why does the α-expansion algorithm for multilabel MRF only guarantee a local optimum, and how does this relate to the submodularity condition? What happens in practice when using non-metric pairwise costs?
2. **Parameter Sensitivity:** Given a noisy image with Gaussian noise σ^2, how would you theoretically determine the optimal β parameter? How does this relate to the bias-variance tradeoff in the segmentation?
3. **Computational Complexity:** The lecture mentions graph cuts can solve binary MRF exactly, but max-flow has $O(V^3)$ worst-case complexity. In practice, why are graph cuts much faster on grid graphs, and what does this tell us about the structure of typical segmentation problems?
4. **Model Limitations:** The Potts model assumes uniform smoothness costs. Design a scenario where this assumption fails dramatically, and propose an alternative pairwise potential that would work better. How would this affect the optimization approach?
5. **Validation Strategy:** Looking at the energy function validation section, explain why finding a configuration with lower energy doesn't always guarantee a better segmentation in practice. What additional validation metrics would you use beyond energy minimization?

Hands-On Examples for MRF Concepts

1. Markov Property: The "Gossip Chain" Analogy

Concrete Example: Imagine people sitting in a line at a movie theater choosing between popcorn (P) or candy (C).

Person:	[1]	[2]	[3]	[4]	[5]
Choice:	P	?	C	?	P

Markov Property: Person 3's choice depends ONLY on persons 2 and 4, not on what person 1 or 5 chose.

Hands-on test:

```
# Non-Markov: Person 3 influenced by everyone
P(person3=C | everyone_else) = f(person1, person2, person4, person5)
```

```
# Markov: Person 3 only cares about neighbors
P(person3=C | everyone_else) = f(person2, person4)
```

Why this matters: In image segmentation, a pixel's label should depend on nearby pixels, not pixels far away. This makes computational sense and matches human vision.

2. Energy Components: The "Peer Pressure vs Personal Preference" Example

Setup: Segmenting a noisy image of black/white stripes.

Data Term (Personal Preference):

```
# Each pixel "wants" to match its intensity
pixel_intensity = 80
black_mean = 50
white_mean = 200

data_cost_black = (80 - 50)**2 = 900 # "I'm kinda black-ish"
data_cost_white = (80 - 200)**2 = 14400 # "I'm definitely not white!"
```

Prior Term (Peer Pressure):

```
# Pixel wants to match its neighbors
neighbors = ['black', 'black', 'white', 'black'] # 3 black, 1 white
prior_cost_black = 1 * beta # disagree with 1 neighbor
prior_cost_white = 3 * beta # disagree with 3 neighbors
```

Total Energy Trade-off:

```
beta = 100
total_black = 900 + 100 = 1000 # personal + peer pressure
total_white = 14400 + 300 = 14700 # personal + peer pressure
# Choose black! (peer pressure wins over weak personal preference)

beta = 10
total_black = 900 + 10 = 910
total_white = 14400 + 30 = 14430
# Still choose black, but peer pressure matters less
```

Interactive Exercise:

```
def choose_label(intensity, neighbors, beta):
    """Try different beta values and see what happens"""
    data_black = (intensity - 50)**2
    data_white = (intensity - 200)**2

    prior_black = beta * sum(n != 'black' for n in neighbors)
    prior_white = beta * sum(n != 'white' for n in neighbors)
```

```
    total_black = data_black + prior_black
    total_white = data_white + prior_white

    return 'black' if total_black < total_white else 'white'

# Test it!
choose_label(80, ['black', 'black', 'white', 'black'], beta=100)
```

3. Submodularity: The "Team Harmony" Example

Intuitive Setup:

Two roommates choosing weekend activities: Study (S) or Party (P).

Submodular (Graph-cut solvable):

```
# "Harmony bonus" - doing same thing together is good
energy_matrix = {
    ('S', 'S'): 0, # both study - great harmony!
    ('P', 'P'): 0, # both party - also great!
    ('S', 'P'): 10, # one studies, one parties - conflict
    ('P', 'S'): 10 # same conflict
}
# Notice: 0 + 0 <= 10 + 10 (submodular condition)
```

Non-submodular (Graph-cut CANNOT solve exactly):

```
# "Jealousy effect" - mixed activities create extra drama
energy_matrix = {
    ('S', 'S'): 5, # both study - boring together
    ('P', 'P'): 5, # both party - also boring
    ('S', 'P'): 0, # mixed - exciting contrast!
    ('P', 'S'): 0
}
# Notice: 5 + 5 > 0 + 0 (violates submodularity)
```

Hands-on Test:

```
def check_submodularity(V):
    """Check if 2x2 energy matrix is submodular"""
    return V[0, 0] + V[1, 1] <= V[0, 1] + V[1, 0]

# Try it!
harmony = np.array([[0, 10], [10, 0]]) # True
jealousy = np.array([[5, 0], [0, 5]]) # False
```

4. Beta Parameter: The "Smoothness Dial" Interactive Demo

Visual Example:

Segmenting a noisy checkerboard pattern.

```
def demo_beta_effect():
    # Create noisy checkerboard
    clean = np.kron([[0,1],[1,0]], np.ones((20,20))) # checkerboard
    noisy = clean + 0.3 * np.random.randn(*clean.shape)

    results = {}
    for beta in [0, 1, 10, 100, 1000]:
        # Run MRF segmentation with different beta
        seg = mrf_segment(noisy, beta=beta)
        results[beta] = seg

        plt.subplot(1,5,list(results.keys()).index(beta)+1)
        plt.imshow(seg, cmap='gray')
        plt.title(f'beta={beta}')

    return results

# What you'll see:
# beta=0: Noisy, follows data exactly (pure maximum likelihood)
# beta=1: Slightly smoother
# beta=10: Much smoother, preserves main structure
# beta=100: Very smooth, loses some detail
# beta=1000: Over-smoothed, might merge separate regions
```

Rule of Thumb:

- $\beta \approx$ noise_variance: Good starting point
- β too small: Noisy result
- β too large: Over-smoothed, loses boundaries

5. Graph Construction: The "Flow Network" Building Exercise

Simple 2x2 image example:

Image:	[10 90] [85 15]

Want to segment into background (0) vs foreground (1)

Step-by-step graph building:

```
# Step 1: Create nodes for each pixel
nodes = {'source': 's', 'sink': 't', 'p1': 1, 'p2': 2, 'p3': 3, 'p4': 4}

# Step 2: Likelihood costs (terminal edges)
mu = [20, 80] # background vs foreground means

# For pixel 1 (intensity=10):
```

```
cost_bg = (10-20)**2 = 100 # cost of being background
cost_fg = (10-80)**2 = 4900 # cost of being foreground
# Add edges: s->p1 (weight=4900), p1->t (weight=100)

# Step 3: Smoothness costs (internal edges)
beta = 500
# Add edges between neighboring pixels: p1<->p2, p1<->p3, p2<->p4, p3<->p4
# Each with weight = beta

# Step 4: Visualize the graph
"""
    s (source)
    |||
    ||| (terminal edges with likelihood costs)
    |||
    p1--p2
    | / | (internal edges with smoothness costs beta)
    p3--p4
    |||
    t (sink)
"""
```

Interactive Exercise:

```
def build_graph_step_by_step(image, mu, beta):
    """Build graph and show each step"""
    g = maxflow.Graph[float]()

    # Step 1: Add pixel nodes
    print("Adding␣pixel␣nodes...")
    nodeids = g.add_grid_nodes(image.shape)

    # Step 2: Add smoothness edges
    print(f"Adding␣smoothness␣edges␣with␣weight␣{beta}...")
    g.add_grid_edges(nodeids, beta)

    # Step 3: Add likelihood edges
    print("Adding␣likelihood␣edges...")
    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            cost_bg = (image[i,j] - mu[0])**2
            cost_fg = (image[i,j] - mu[1])**2
            print(f"Pixel␣({i},{j}):␣bg-cost={cost_bg},␣fg-cost={cost_fg}")

    g.add_grid_tedges(nodeids,
                      (image - mu[1])**2, # source edges (fg cost)
                      (image - mu[0])**2) # sink edges (bg cost)

    return g, nodeids
```

6. Alpha Expansion: The "Musical Chairs" Example

Setup:

3-class segmentation (Red, Green, Blue) using iterative binary decisions.

```
def alpha_expansion_demo():
    # Current Labeling: [R, R, G, B, B]
    current = ['R', 'R', 'G', 'B', 'B']

    # Round 1: alpha = Red
    # Question: Which pixels should switch TO Red?
    # Binary problem: Red vs {Current_Green, Current_Blue}
    print("alpha-expansion for Red:")
    print("Current: [R, R, G, B, B]")
    print("Options: [R, R, R, R, R] vs keep current non-Red")
    # Solve binary graph cut...
    round1 = ['R', 'R', 'R', 'B', 'B'] # G switched to R

    # Round 2: alpha = Green
    print("alpha-expansion for Green:")
    print("Current: [R, R, R, B, B]")
    print("Options: [G, G, G, G, G] vs keep current non-Green")
    round2 = ['R', 'R', 'G', 'B', 'B'] # some chairs switched back to G

    # Round 3: alpha = Blue
    # ... and so on until convergence

    return round2
```

Key Insight:

Each α-expansion round is a binary graph cut problem, which we can solve exactly!

7. Parameter Tuning: The "Cooking Recipe" Approach

Interactive Parameter Explorer:

```
def parameter_cooking_demo(noisy_image):
    """Like adjusting seasoning - taste and adjust!"""

    # Recipe ingredients:
    ingredients = {
        'mu_background': 0.2, # "saltiness" - background intensity
        'mu_foreground': 0.8, # "sweetness" - foreground intensity
        'beta': 100 # "spiciness" - smoothness level
    }

    print("MRF Cooking Demo:")
    print("Try different 'seasonings' and taste the results!")

    for beta in [1, 10, 100, 1000]:
        print(f"Spiciness level (beta): {beta}")
```

```
        result = mrf_segment(noisy_image,
                             mu=[ingredients['mu_background'],
                                 ingredients['mu_foreground']],
                             beta=beta)

        # "Taste test" - measure quality
        smoothness = count_boundary_pixels(result)
        accuracy = compare_to_ground_truth(result)

        print(f"Smoothness: {smoothness} (lower = smoother)")
        print(f"Accuracy: {accuracy:.2f} (higher = better)")

        if beta == 1:
            print("Taste: Too noisy! Needs more smoothness.")
        elif beta == 10:
            print("Taste: Getting better...")
        elif beta == 100:
            print("Taste: Just right!")
        else:
            print("Taste: Over-smoothed. Lost the details.")

# Usage:
# parameter_cooking_demo(your_noisy_image)
```

Hands-on Tuning Strategy:

1. Start with $\beta = 0$: See pure maximum likelihood (probably noisy)
2. Gradually increase β: Watch noise disappear
3. Too high β: Important boundaries start disappearing
4. Sweet spot: Noise gone, boundaries preserved

1. Basic Energy Calculation: 3x3 Grid Example

Given:

3x3 noisy image, binary segmentation (0=dark, 1=bright)

Pixel intensities:	Proposed segmentation:
[45 55 195]	[0 0 1]
[50 180 190]	[0 1 1]
[195 185 60]	[1 1 0]

Parameters:

$\mu_0 = 50$ (dark class), $\mu_1 = 180$ (bright class), $\beta = 100$

Step 1: Calculate Likelihood Energy U_1

For each pixel: $V_1(f) = (\mu(f) - d)^2$

Position (0,0): f=0, d=45	$\rightarrow (50-45)^2 = 25$
Position (0,1): f=0, d=55	$\rightarrow (50-55)^2 = 25$
Position (0,2): f=1, d=195	$\rightarrow (180-195)^2 = 225$
Position (1,0): f=0, d=50	$\rightarrow (50-50)^2 = 0$
Position (1,1): f=1, d=180	$\rightarrow (180-180)^2 = 0$
Position (1,2): f=1, d=190	$\rightarrow (180-190)^2 = 100$
Position (2,0): f=1, d=195	$\rightarrow (180-195)^2 = 225$
Position (2,1): f=1, d=185	$\rightarrow (180-185)^2 = 25$
Position (2,2): f=0, d=60	$\rightarrow (50-60)^2 = 100$

$$U_1 = 25+25+225+0+0+100+225+25+100 = 725$$

Step 2: Calculate Prior Energy U_2

Count neighboring pairs with different labels, multiply by $\beta = 100$

Horizontal neighbors (left-right):

(0,0)-(0,1): 0#0? No	$\rightarrow 0$
(0,1)-(0,2): 0#1? Yes	$\rightarrow 1$
(1,0)-(1,1): 0#1? Yes	$\rightarrow 1$
(1,1)-(1,2): 1#1? No	$\rightarrow 0$
(2,0)-(2,1): 1#1? No	$\rightarrow 0$
(2,1)-(2,2): 1#0? Yes	$\rightarrow 1$

Horizontal disagreements: 3

Vertical neighbors (up-down):

(0,0)-(1,0): 0#0? No	$\rightarrow 0$
(0,1)-(1,1): 0#1? Yes	$\rightarrow 1$
(0,2)-(1,2): 1#1? No	$\rightarrow 0$
(1,0)-(2,0): 0#1? Yes	$\rightarrow 1$
(1,1)-(2,1): 1#1? No	$\rightarrow 0$
(1,2)-(2,2): 1#0? Yes	$\rightarrow 1$

Vertical disagreements: 3

$$U_2 = \beta \times (3+3) = 100 \times 6 = 600$$

Step 3: Total Energy

$$E = U_1 + U_2 = 725 + 600 = 1325$$

2. Comparing Alternative Segmentations

Let's try a different segmentation for the same image:

Alternative segmentation:
[0 0 1]
[0 0 1]
[1 1 1]

Step 1: New Likelihood Energy

Position (0,0): f=0, d=45	$\rightarrow (50-45)^2 = 25$
Position (0,1): f=0, d=55	$\rightarrow (50-55)^2 = 25$
Position (0,2): f=1, d=195	$\rightarrow (180-195)^2 = 225$
Position (1,0): f=0, d=50	$\rightarrow (50-50)^2 = 0$
Position (1,1): f=0, d=180	$\rightarrow (50-180)^2 = 16900$ Big penalty!
Position (1,2): f=1, d=190	$\rightarrow (180-190)^2 = 100$
Position (2,0): f=1, d=195	$\rightarrow (180-195)^2 = 225$
Position (2,1): f=1, d=185	$\rightarrow (180-185)^2 = 25$
Position (2,2): f=1, d=60	$\rightarrow (180-60)^2 = 14400$ Big penalty!

$$\text{New } U_1 = 25+25+225+0+16900+100+225+25+14400 = 31925$$

Step 2: New Prior Energy

Horizontal disagreements:

- (0,1)-(0,2): $0 \neq 1 \rightarrow 1$
- (1,1)-(1,2): $0 \neq 1 \rightarrow 1$
- Total: 2

Vertical disagreements:

- (0,1)-(1,1): $0 \neq 0 \rightarrow 0$
- (1,0)-(2,0): $0 \neq 1 \rightarrow 1$
- (1,1)-(2,1): $0 \neq 1 \rightarrow 1$
- Total: 2

$$\text{New } U_2 = 100 \times (2+2) = 400$$

Step 3: Compare Total Energies

- Original: $E = 1325$
- Alternative: $E = 31925 + 400 = 32325$

Conclusion:

Original is much better! (Lower energy = higher probability)

3. Gender Classification Hand Calculation

Given:

Heights [179, 174, 182, 162, 175, 165] Parameters: $\mu_M = 181$, $\mu_F = 165$, $\beta = 100$

Configuration A: (M, M, M, F, M, F)

Likelihood costs:

Person 1: Height=179, Label=M	$\rightarrow (181-179)^2 = 4$
Person 2: Height=174, Label=M	$\rightarrow (181-174)^2 = 49$
Person 3: Height=182, Label=M	$\rightarrow (181-182)^2 = 1$
Person 4: Height=162, Label=F	$\rightarrow (165-162)^2 = 9$
Person 5: Height=175, Label=M	$\rightarrow (181-175)^2 = 36$
Person 6: Height=165, Label=F	$\rightarrow (165-165)^2 = 0$

$$U_1 = 4+49+1+9+36+0 = 99$$

Prior costs (neighboring differences):

1-2: M#M? No	$\rightarrow 0$
2-3: M#M? No	$\rightarrow 0$
3-4: M#F? Yes	$\rightarrow 1$
4-5: F#M? Yes	$\rightarrow 1$
5-6: M#F? Yes	$\rightarrow 1$

$$U_2 = 100 \times 3 = 300$$

Total:

$E_A = 99 + 300 = 399$

Configuration B: (M, M, M, F, F, F)

Likelihood costs:

Person 1: Height=179, Label=M	$\rightarrow (181-179)^2 = 4$
Person 2: Height=174, Label=M	$\rightarrow (181-174)^2 = 49$
Person 3: Height=182, Label=M	$\rightarrow (181-182)^2 = 1$
Person 4: Height=162, Label=F	$\rightarrow (165-162)^2 = 9$
Person 5: Height=175, Label=F	$\rightarrow (165-175)^2 = 100$ Changed!
Person 6: Height=165, Label=F	$\rightarrow (165-165)^2 = 0$

$$U_1 = 4+49+1+9+100+0 = 163$$

Prior costs:

1-2: M#M? No	$\rightarrow 0$
2-3: M#M? No	$\rightarrow 0$
3-4: M#F? Yes	$\rightarrow 1$
4-5: F#F? No	$\rightarrow 0$
5-6: F#F? No	$\rightarrow 0$

$$U_2 = 100 \times 1 = 100$$

Total:

$E_B = 163 + 100 = 263$

Conclusion:

Configuration B is better! ($263 < 399$)

4. Graph Cut Edge Weight Calculation

Simple 2x2 binary problem:

Image:	[10 90] [85 15]

Parameters:

$\mu_0 = 20$, $\mu_1 = 80$, $\beta = 50$

Terminal Edge Weights (likelihood costs):

To SOURCE (choosing label 1):

Pixel (0,0):	$(10-80)^2 = 4900$
Pixel (0,1):	$(90-80)^2 = 100$
Pixel (1,0):	$(85-80)^2 = 25$
Pixel (1,1):	$(15-80)^2 = 4225$

To SINK (choosing label 0):

Pixel (0,0):	$(10-20)^2 = 100$
Pixel (0,1):	$(90-20)^2 = 4900$
Pixel (1,0):	$(85-20)^2 = 4225$
Pixel (1,1):	$(15-20)^2 = 25$

Internal Edge Weights (smoothness costs):

All neighboring pixel pairs get weight $\beta = 50$:

- (0,0)$\leftrightarrow$(0,1): weight = 50
- (0,0)$\leftrightarrow$(1,0): weight = 50
- (0,1)$\leftrightarrow$(1,1): weight = 50
- (1,0)$\leftrightarrow$(1,1): weight = 50

Expected Solution:

- Pixel (0,0): $100 < 4900 \rightarrow$ choose label 0 (sink side)
- Pixel (0,1): $4900 > 100 \rightarrow$ choose label 1 (source side)
- Pixel (1,0): $4225 > 25 \rightarrow$ choose label 1 (source side)
- Pixel (1,1): $25 < 4225 \rightarrow$ choose label 0 (sink side)

Predicted segmentation:

[0 1; 1 0] (checkerboard pattern)

5. Beta Parameter Impact Analysis

Same 2x2 image:

[10 90; 85 15], $\mu_0 = 20$, $\mu_1 = 80$

$\beta = 0$ (no smoothness):

Pure maximum likelihood:

- (0,0): $\min(100, 4900) \rightarrow$ label 0
- (0,1): $\min(4900, 100) \rightarrow$ label 1
- (1,0): $\min(4225, 25) \rightarrow$ label 1
- (1,1): $\min(25, 4225) \rightarrow$ label 0

Result: [0 1; 1 0], Total energy = $100 + 100 + 25 + 25 = 250$

$\beta = 10000$ (extreme smoothness):

Smoothness dominates. Let's check uniform labelings:

All label 0:

- Likelihood: $100 + 4900 + 4225 + 25 = 9250$
- Prior: 0 (no disagreements)
- Total: 9250

All label 1:

- Likelihood: $4900 + 100 + 25 + 4225 = 9250$
- Prior: 0 (no disagreements)
- Total: 9250

Mixed (0 1; 1 0):

- Likelihood: 250 (calculated above)
- Prior: $10000 \times 4 = 40000$ (4 neighboring disagreements)
- Total: 40250

Winner: Either all-0 or all-1 (tie at $9250 < 40250$)

$\beta = 1000$ (moderate smoothness):

0 1; 1 0 : $E = 250 + 1000 \times 4 = 4250$
0 0; 0 0 : $E = 9250 + 0 = 9250$
1 1; 1 1 : $E = 9250 + 0 = 9250$

Winner: Mixed pattern still wins ($4250 < 9250$)

Conclusion:

$\beta = 1000$ preserves data fidelity while $\beta = 10000$ forces uniformity.

6. Submodularity Check by Hand

Check if this pairwise potential is submodular:

$$\boxed{\begin{array}{c} V_2(0,0) = 5 \\ V_2(0,1) = 20 \\ V_2(1,0) = 20 \\ V_2(1,1) = 2 \end{array}}$$

Submodularity condition:

$V_2(0,0)+V_2(1,1) \leq V_2(0,1)+V_2(1,0)$

Calculate:

- Left side: $5+2=7$
- Right side: $20+20=40$

Check:

$7 \leq 40$ ✓This IS submodular

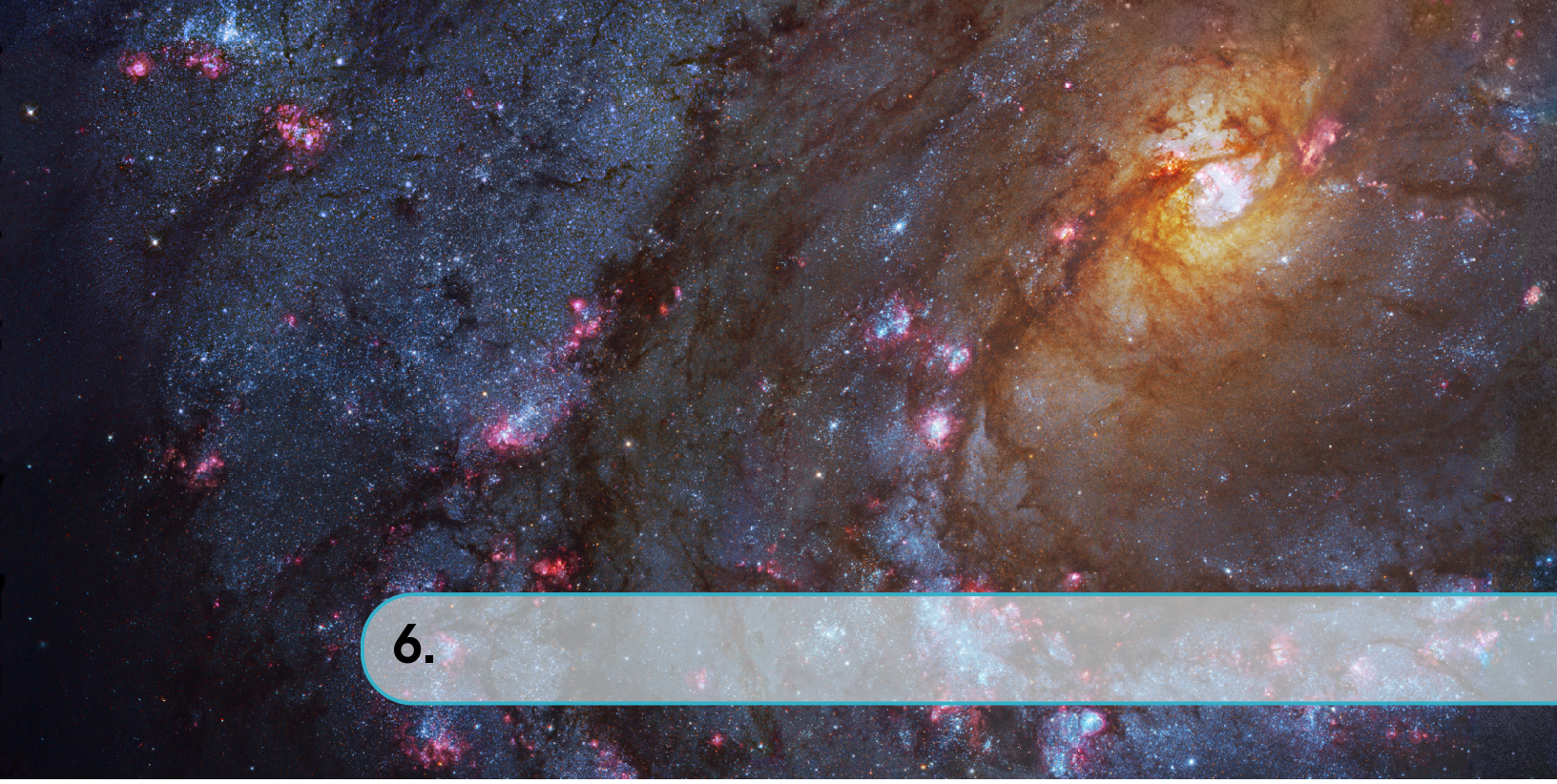
Counter-example (non-submodular):

$$\boxed{\begin{array}{l} V_2(0,0)=1 \\ V_2(0,1)=0 \\ V_2(1,0)=0 \\ V_2(1,1)=1 \end{array}}$$

Check:

$1+1 \leq 0+0$? $2 \leq 0$ X This is NOT submodular

This captures the "jealousy" effect where mixed states (0,1) and (1,0) have lower cost than pure states!



6.

1. High-Level Overview

Deformable models

represent a fundamental approach to image segmentation that combines image data with geometric constraints. The core idea is to evolve a curve in an image under the influence of two competing forces: **external forces** (derived from image data) that push the curve toward object boundaries, and **internal forces** that maintain curve smoothness and prevent overfitting to noise.

This topic sits at the intersection of **variational methods, numerical optimization, and computer vision**. Unlike pixel-based segmentation methods (e.g., region growing, thresholding), deformable models provide:

- **Inherent smoothness** through regularization
- **Robustness to noise** via internal forces
- **Sub-pixel accuracy** through continuous curve representation
- **Temporal coherence** for tracking applications

The approach bridges classical **snakes** (parametric representation) with modern **level set methods** (implicit representation), though this implementation focuses on the parametric approach.

6.1 Key Theoretical Concepts

6.1.1 Energy Functional Framework

Definition: The segmentation problem is formulated as energy minimization:

$$E = E_{\text{ext}} + E_{\text{int}}$$

where the optimal segmentation corresponds to the curve configuration with minimal total energy.

Properties:

- $\mathbf{E}_{\text{ext}}$: Data fidelity term driving curve toward object boundaries
- $\mathbf{E}_{\text{int}}$: Regularization term maintaining curve smoothness
- **Gradient descent**: Evolution follows $F = -\nabla E$

6.1.2 External Energy (Chan-Vese Model)

Definition: Piecewise constant energy functional:

$$E_{\text{ext}} = \int_{\Omega_{\text{in}}} (I - m_{\text{in}})^2 \, d\omega + \int_{\Omega_{\text{out}}} (I - m_{\text{out}})^2 \, d\omega$$

Derivation: The corresponding external force is:

$$F_{\text{ext}} = (m_{\text{in}} - m_{\text{out}})(2I - m_{\text{in}} - m_{\text{out}})N$$

Intuition: Points on the curve move toward regions where they "better belong" based on intensity statistics. If a point has intensity closer to the inside mean, it moves inward; if closer to outside mean, it moves outward.

6.1.3 Internal Energy (Snake Regularization)

Definition: Classical snake energy penalizing stretching and bending:

$$E_{\text{int}} = \frac{1}{2} \int_0^1 \left[\alpha \left| \frac{\partial C}{\partial s} \right|^2 + \beta \left| \frac{\partial^2 C}{\partial s^2} \right|^2 \right] ds$$

Discrete Approximation: Using finite differences:

- **First derivative** (elasticity): filter $[1, -2, 1]$ with weight α
- **Second derivative** (rigidity): filter $[-1, 4, -6, 4, -1]$ with weight β

Matrix Form: The regularization becomes matrix multiplication with circulant matrix B_{int}.

6.1.4 Backward Euler Scheme

Stability Consideration: For numerical stability, the discrete update uses:

$$C^t = B_{\text{int}} \left[C^{t-1} + \tau \left(f_{\text{ext}} \right) N^{t-1} \right]$$

Why Backward Euler: Forward Euler $C^t = C^{t-1} + \tau F$ can become unstable for large time steps, while the implicit scheme $(I - \tau B)C^t = C^{t-1} + \tau f_{\text{ext}}$ ensures stability.

6.2 Implementation Walk-Through

6.2.1 Core Snake Functions

`make_circular_snake(N, center, radius)`

Purpose: Initialize circular snake with N points around specified center and radius.

Algorithm:

```
# Generate N equally spaced angles
angles = np.linspace(0, 2*np.pi, N, endpoint=False)
# Create unit circle points
unit_circle = np.array([np.cos(angles), np.sin(angles)]).T
# Scale and translate
return center + radius * unit_circle
```

Complexity: $O(N)$, **Pitfall:** Ensure `endpoint=False` to avoid duplicate point at $0/2\pi$.

`get_normals(snake)`

Purpose: Compute outward-pointing unit normals at each snake point.

Algorithm:

```
# Compute tangent vectors using neighboring points
ds = normalize(np.roll(snake, 1, axis=0) - snake)
tangent = normalize(np.roll(ds, -1, axis=0) + ds)

# Normal is perpendicular to tangent
normal = np.stack([-tangent[:, 1], tangent[:, 0]], axis=1)
```

Complexity: $O(N)$, **Pitfall:** Handle the `normalize` function to avoid division by zero.

`regularization_matrix(N, alpha, beta)`

Purpose: Construct circulant matrix for internal force regularization.

Algorithm:

```
# Create impulse response for combined filter
s = np.zeros(N)
s[[-2, -1, 0, 1, 2]] = (alpha * np.array([0, 1, -2, 1, 0]) +
                        beta * np.array([-1, 4, -6, 4, -1]))

# Build circulant matrix and invert for backward Euler
s = scipy.linalg.circulant(s)
return scipy.linalg.inv(np.eye(N) - s)
```

Complexity: $O(N^3)$ for matrix inversion, **Pitfall:** Matrix inversion can be numerically unstable for large β.

`evolve_snake(snake, I, B, step_size)`

Purpose: Single iteration of snake evolution combining external and internal forces.

Algorithm:

```
# 1. Compute region statistics
mask = skimage.draw.polygon2mask(I.shape, snake)
m_in = np.mean(I[mask])
m_out = np.mean(I[~mask])

# 2. Interpolate image values at snake points
f = scipy.interpolate.RectBivariateSpline(range(I.shape[0]), range(I.shape[1]), I)
val = f(snake[:, 0], snake[:, 1], grid=False)

# 3. Compute external force
f_ext = (m_in - m_out) * (2 * val - m_in - m_out)
displacement = step_size * f_ext[:, None] * get_normals(snake)

# 4. Apply external and internal forces
snake = snake + displacement        # External displacement
```

```
snake = B @ snake                    # Internal regularization

# 5. Post-processing steps
snake = remove_intersections(snake)
snake = distribute_points(snake)
snake = keep_snake_inside(snake, I.shape)
```

Complexity: $O(N^2)$ for polygon mask, $O(N)$ for other operations.

6.2.2 Utility Functions

`distribute_points(snake)`

Purpose: Redistribute points equidistantly along curve to maintain quality.

Algorithm: Use cumulative arc length and interpolation to resample points uniformly.

`remove_intersections(snake)`

Purpose: Detect and remove self-intersections by reversing smallest loops.

Algorithm: Check all pairs of line segments for intersection using CCW test, then reverse the ordering of the smaller loop.

6.3 Implementation Walk-Through

6.3.1 Core Snake Functions

`make_circular_snake(N, center, radius)`

Purpose: Initialize circular snake with N points around specified center and radius.

Algorithm:

```
# Generate N equally spaced angles
angles = np.linspace(0, 2*np.pi, N, endpoint=False)

# Create unit circle points
unit_circle = np.array([np.cos(angles), np.sin(angles)]).T

# Scale and translate
return center + radius * unit_circle
```

Complexity: $O(N)$, **Pitfall:** Ensure `endpoint=False` to avoid duplicate point at $0/2\pi$.

`get_normals(snake)`

Purpose: Compute outward-pointing unit normals at each snake point.

Algorithm:

```
# Compute tangent vectors using neighboring points
ds = normalize(np.roll(snake, 1, axis=0) - snake)
tangent = normalize(np.roll(ds, -1, axis=0) + ds)

# Normal is perpendicular to tangent
normal = np.stack([-tangent[:, 1], tangent[:, 0]], axis=1)
```

Complexity: $O(N)$, **Pitfall:** Handle the `normalize` function to avoid division by zero.

`regularization_matrix(N, alpha, beta)`

Purpose: Construct circulant matrix for internal force regularization.

Algorithm:

```
# Create impulse response for combined filter
s = np.zeros(N)
s[[-2, -1, 0, 1, 2]] = (alpha * np.array([0, 1, -2, 1, 0]) +
                        beta * np.array([-1, 4, -6, 4, -1]))

# Build circulant matrix and invert for backward Euler
S = scipy.linalg.circulant(s)
return scipy.linalg.inv(np.eye(N) - S)
```

Complexity: $O(N^3)$ for matrix inversion, **Pitfall:** Matrix inversion can be numerically unstable for large β.

6.3.2 Snake Evolution

`evolve_snake(snake, I, B, step_size)`

Purpose: Single iteration of snake evolution combining external and internal forces.

Algorithm:

```
# 1. Compute region statistics
mask = skimage.draw.polygon2mask(I.shape, snake)
m_in = np.mean(I[mask])
m_out = np.mean(I[~mask])

# 2. Interpolate image values at snake points
f = scipy.interpolate.RectBivariateSpline(range(I.shape[0]), range(I.shape[1]), I)
val = f(snake[:, 0], snake[:, 1], grid=False)

# 3. Compute external force
f_ext = (m_in - m_out) * (2 * val - m_in - m_out)
displacement = step_size * f_ext[:, None] * get_normals(snake)

# 4. Apply external and internal forces
snake = snake + displacement        # External displacement
snake = B @ snake                   # Internal regularization

# 5. Post-processing steps
snake = remove_intersections(snake)
snake = distribute_points(snake)
snake = keep_snake_inside(snake, I.shape)
```

Complexity: $O(N^2)$ for polygon mask, $O(N)$ for other operations.

6.3.3 Utility Functions

`distribute_points(snake)`

Purpose: Redistribute points equidistantly along curve to maintain quality.

Algorithm: Use cumulative arc length and interpolation to resample points uniformly.

`remove_intersections(snake)`

Purpose: Detect and remove self-intersections by reversing smallest loops.

Algorithm: Check all pairs of line segments for intersection using CCW test, then reverse the ordering of the smaller loop.

6.4 Crucial Details & 'Exam Traps'

- **Parameter tuning:** α controls elasticity (point spacing), β controls rigidity (curvature smoothness)
- **Step size stability:** Too large τ causes oscillations; too small slows convergence
- **Initialization sensitivity:** Snake must start near target object to avoid local minima
- **Intersection handling:** Self-intersections can cause topology changes if not handled
- **Boundary conditions:** Snake points must stay within image bounds
- **Matrix inversion:** `regularization matrix` can become ill-conditioned for extreme parameters
- **Normal direction:** Ensure normals point outward for proper force direction
- **Interpolation vs nearest neighbor:** Interpolation provides sub-pixel accuracy but adds computational cost
- **Closed curve representation:** Include duplicate point at end for visualization, remove for computation

6.5 Worked Example

Setup: Segment a plus sign with initial circular snake.

Initial Configuration:

```
python
I = plusplus_image  # 300x300 grayscale image
center = [150, 150]
radius = 70
N = 100
snake = make_circular_snake(N, center, radius)
```

Iteration 1:

```
python
# Compute means
mask = polygon2mask(I.shape, snake)
m_in = 180  (bright plus)
m_out = 50  (dark background)

# At a point with intensity I=50 (background region):
f_ext = (180-50) * (2*50 - 180 - 50) = 130 * (-110) = -14300
# Negative force → move inward (contract)
```

```
# At a point with intensity I=200 (plus region):
f_ext = (180-50) * (2*200 - 180 - 50) = 130 * 170 = 22100
# Positive force → move outward (expand)
```

Result: Snake contracts in background regions and expands in foreground regions, gradually conforming to plus shape.

6.6 Crucial Details & 'Exam Traps'

- **Parameter tuning:** α controls elasticity (point spacing), β controls rigidity (curvature smoothness)
- **Step size stability:** Too large τ causes oscillations; too small slows convergence
- **Initialization sensitivity:** Snake must start near target object to avoid local minima
- **Intersection handling:** Self-intersections can cause topology changes if not handled
- **Boundary conditions:** Snake points must stay within image bounds
- **Matrix inversion:** `regularization matrix` can become ill-conditioned for extreme parameters
- **Normal direction:** Ensure normals point outward for proper force direction
- **Interpolation vs nearest neighbor:** Interpolation provides sub-pixel accuracy but adds computational cost
- **Closed curve representation:** Include duplicate point at end for visualization, remove for computation

6.7 Worked Example

Setup: Segment a plus sign with initial circular snake.

Initial Configuration:

```
python
I = plusplus_image  # 300x300 grayscale image
center = [150, 150]
radius = 70
N = 100
snake = make_circular_snake(N, center, radius)
```

Iteration 1:

```
python
# Compute means
mask = polygon2mask(I.shape, snake)
m_in = 180  (bright plus)
m_out = 50  (dark background)

# At a point with intensity I=50 (background region):
f_ext = (180-50) * (2*50 - 180 - 50) = 130 * (-110) = -14300
# Negative force → move inward (contract)
```

```
# At a point with intensity I=200 (plus region):
f_ext = (180-50) * (2*200 - 180 - 50) = 130 * 170 = 22100
# Positive force → move outward (expand)
```

Result: Snake contracts in background regions and expands in foreground regions, gradually conforming to plus shape.

6.8 Cheat-Sheet Section

Key Formulas:

- External force: `f_ext = (m_in - m_out)(2I - m_in - m_out)`
- Normal vector: `N = [-t_y, t_x]` where `t` is unit tangent
- Update step: `C^t = B[C^(t-1) + τf_ext·N]`
- Elasticity filter: `[1, -2, 1]` (spacing control)
- Rigidity filter: `[-1, 4, -6, 4, -1]` (curvature control)

Parameter Guidelines:

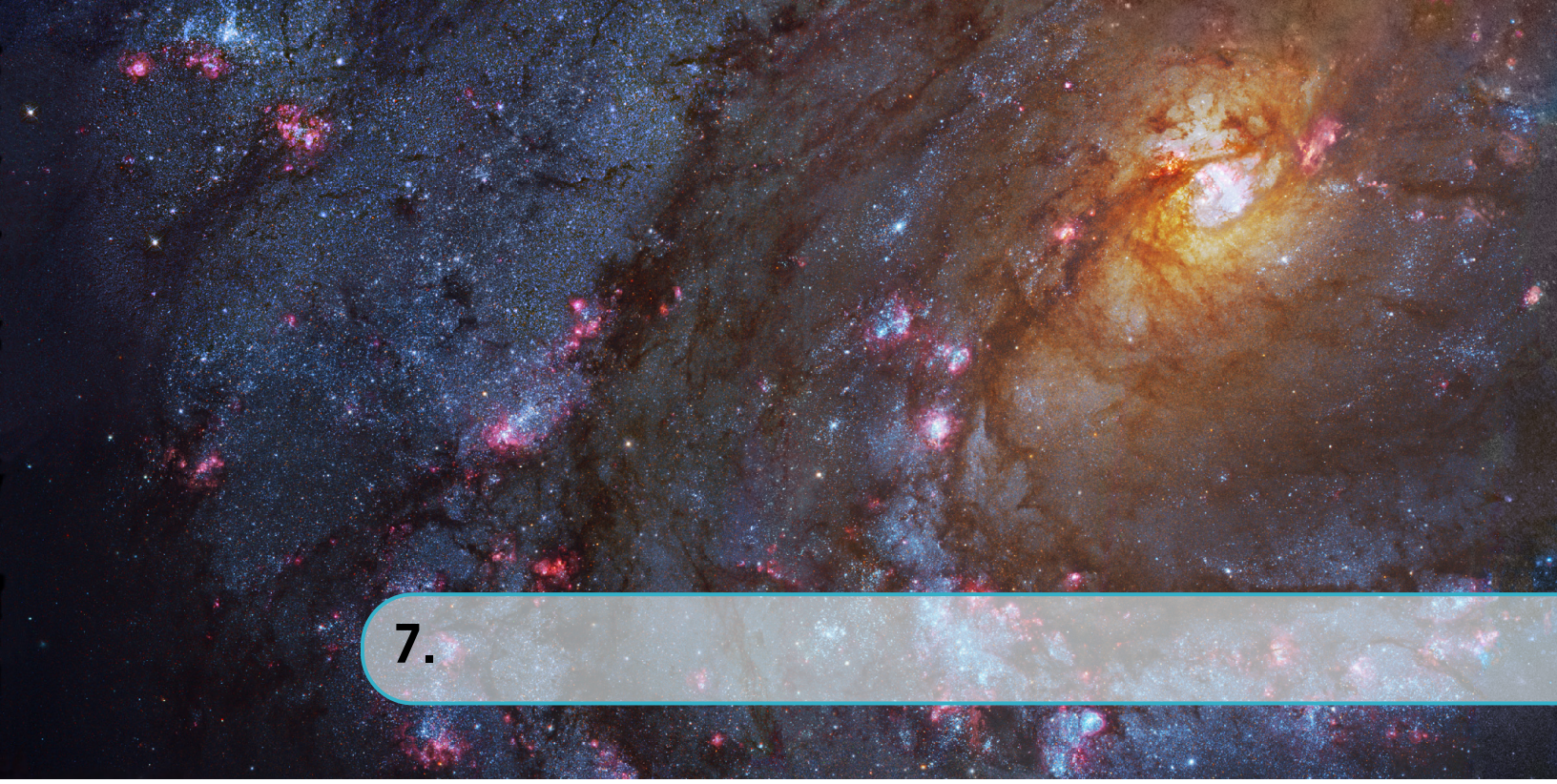
- $\alpha \in [0.001, 0.1]$: smaller = more flexible spacing
- $\beta \in [0.001, 0.1]$: smaller = more flexible curvature
- $\tau \in [0.1, 10]$: smaller = more stable evolution
- N ≈ 100: good balance of accuracy vs speed

Boundary Conditions:

- Periodic: `snake[N] = snake[0]` for closed curves
- Clamping: `snake = clip(snake, 0, shape-1)` for image bounds

6.9 Reflective Questions

1. **Theory-Implementation Bridge:** Why does the backward Euler scheme `(I-S)^(-1)` appear in `regularization_matrix()` instead of just applying filter `S` directly? What stability issues would arise with forward Euler?
2. **Force Direction Analysis:** Given an image where foreground intensity is lower than background (dark object on bright background), predict the sign of `f_ext` at the boundary and explain how this affects snake evolution.
3. **Parameter Sensitivity:** If you increase β while keeping α constant, how does this affect the snake's ability to fit into sharp corners? Provide both mathematical reasoning (from energy functional) and practical reasoning (from filter interpretation).
4. **Numerical Considerations:** The `remove_intersections()` function uses a simple line intersection test. Under what geometric conditions might this fail, and how could topological changes during evolution cause the algorithm to get stuck in local minima?
5. **Extension to Tracking:** In the tracking application, why is it crucial to use the result from frame `t-1` as initialization for frame `t`? What would happen if you reinitialized with a circular snake for each frame, and how does this relate to the temporal coherence assumption in video analysis?



7.

7.1 High-Level Overview ($\leq$ 150 words)

Layered surface detection addresses segmentation problems where we seek terrain-like or tubular surfaces with geometric constraints. The core innovation transforms the optimal surface detection problem into a **minimum-cost s-t cut** on a specially constructed graph, enabling polynomial-time solutions via max-flow algorithms.

This technique excels in medical imaging where anatomical structures exhibit layered organization (retinal OCT, dental implants, abdominal fat). The algorithm can simultaneously detect **multiple interrelated surfaces** with smoothness and overlap constraints, making it robust for complex segmentation tasks.

Key advantages include:

(1) **Global optimality** guarantees,
(2) **Geometric constraint enforcement** (smoothness, containment),
(3) **Extension to in-region costs** for blurry boundaries, and
(4) **Tubular surface handling** via coordinate transformations.

The method's success depends critically on designing appropriate *cost volumes* that encode domain knowledge about where surfaces should be located.

7.2 Key Theoretical Concepts

7.2.1 Optimal Net Surface Problem

- **Definition**: Find terrain-like surface $s : \mathbb{Z}^2 \to \mathbb{Z}$ (where $z = s(x,y)$) minimizing cost subject to smoothness constraints.
- **Smoothness constraint**: $|s(x,y) - s(x-1,y)| \leq \Delta_x$ and $|s(x,y) - s(x,y-1)| \leq \Delta_y$
- **Optimality**: Among all valid surfaces, return the one with minimum total cost.
- **Intuition**: Constraining surface roughness makes segmentation tractable while ensuring realistic results.

7.2.2 Graph Construction & Transformation

- **Core theorem**: Optimal net surface problem reduces to minimum-cost closed set in node-weighted directed graph.
- **Further reduction**: Closed set problem transforms to minimum s-t cut in arc-weighted graph.
- **Polynomial complexity**: Solvable via Boykov-Kolmogorov max-flow in $\mathcal{O}(n^3)$ time.
- **Proof sketch**: Graph nodes represent volume voxels; edges encode smoothness constraints; cut corresponds to surface.

Cost Function Types

On-surface cost:

$$C_{\text{on}}(s,c) = \sum_{x=1}^{X}\sum_{y=1}^{Y} c(x,y,s(x,y)) \tag{7.1}$$

In-region cost:

$$C_{\text{in}}(s,c_{\text{below}},c_{\text{above}}) = \sum_{x=1}^{X}\sum_{y=1}^{Y}\left[\sum_{z=1}^{s(x,y)} c_{\text{below}}(x,y,z) + \sum_{z=s(x,y)+1}^{Z} c_{\text{above}}(x,y,z)\right] \tag{7.2}$$

- **Intuition:** On-surface penalizes surface placement; in-region penalizes region assignment
- **Robustness:** In-region costs more robust to noise due to larger evaluation area

Multiple Surface Extensions

- **Overlap constraint:** $\delta_{\text{low}} \leq s_2(x,y) - s_1(x,y) \leq \delta_{\text{high}}$
- **Joint optimization:** Minimize $C_{\text{on}}(s_1,c_1) + C_{\text{on}}(s_2,c_2)$ subject to all constraints
- **Applications:** Detecting tissue layer boundaries with known thickness relationships

3. Implementation Walk-Through

Limestone Solution: Basic Layered Detection

Purpose: Demonstrates progressive complexity from single surface to multiple surfaces with different cost functions.

Single Surface Detection (Darkest Line)

```
# Create graph object with intensity-based cost
layer = slgbuilder.GraphObject(I) # I = grayscale image
helper = slgbuilder.MaxflowBuilder()
helper.add_object(layer)
helper.add_layered_boundary_cost() # Terminal connections
helper.add_layered_smoothness(delta=delta, wrap=False) # Smoothness edges
```

Algorithm steps:

1. **Graph construction:** Each pixel column becomes a chain of nodes
2. **Boundary costs:** Connect source/sink to top/bottom of image
3. **Smoothness edges:** Link adjacent columns with capacity constraints
4. **Solve:** Run max-flow to find minimum cut
5. **Extract surface:** Cut location gives surface height per column

Multiple Surface Detection

```
layers = [slgbuilder.GraphObject(I), slgbuilder.GraphObject(I)]
helper.add_objects(layers)
helper.add_layered_containment(layers[0], layers[1],
                               min_margin=50, max_margin=200)
```

Containment constraint: Ensures $50 \leq s_2(x,y) - s_1(x,y) \leq 200$ everywhere

Region-Based Cost

```
helper.add_layered_region_cost(layers[0], 255 - I, I) # Dark region below
helper.add_layered_region_cost(layers[1], I, 255 - I) # Bright region above
```

Cost assignment: Region between surfaces gets cost based on whether pixels should be dark or bright

Dental Solution: Tubular Surface Detection

Purpose: Segment dental implant surface from CT slices using cylindrical coordinate transformation.

Coordinate Unwrapping

```
center = (np.array(V.shape[1:]) - 1) / 2
radii = np.arange(min(V.shape[1:]) / 2) + 1
angles = np.linspace(0, 2 * np.pi, a, endpoint=False)
X = center[0] + np.outer(radii, np.cos(angles))
Y = center[1] + np.outer(radii, np.sin(angles))
```

Transformation: $(x,y) \mapsto (r,\theta)$ where circular objects become horizontal lines

Cost Volume Construction

```
F = scipy.interpolate.RectBivariateSpline(np.arange(I.shape[0]),
                                          np.arange(I.shape[1]), I)
U = F(grid[:, 0], grid[:, 1], grid=False).reshape((len(radii), a))
cost = np.diff(U.astype(float), axis=0) # Radial gradient
cost = np.minimum(cost, 0) # Keep only negative gradients
```

Gradient-based cost: Surface should be at locations with strong radial intensity decrease

Circular Boundary Handling

```
helper.add_layered_smoothness(delta=delta, wrap=True) # Periodic boundary
```

Wrapping: $\theta = 0$ and $\theta = 2\pi$ are adjacent (circular topology)

Surface Expansion & Quantification

```
r0 = segmentation.sum(axis=0) - 0.5
r20 = r0 + 20 # Expand surface outward by 20 pixels
contact_from_image = (F(y20, x20, grid=False) > 110).mean()
```

Osseointegration measure: Percentage of expanded surface in contact with bone (intensity > 110)

4. Crucial Details & 'Exam Traps'

- **Graph construction differs from MRF:** Not pixel-based grid; uses specialized surface topology
- **Smoothness constraints are hard:** Violated surfaces impossible, not just penalized
- **Delta parameter:** Controls maximum slope; too small = oversmooth, too large = noisy
- **Wrap parameter:** Critical for tubular data; False for terrain, True for circular
- **Cost normalization:** Must convert to int32; poor scaling causes solver issues
- **Boundary costs:** Required for proper source/sink connections; often forgotten
- **Containment margins:** Min/max separation between surfaces; unrealistic values cause infeasible problems
- **Coordinate transformation:** Center estimation critical for tubular unwrapping
- **Gradient direction:** For implant detection, want negative radial gradients (bright to dark outward)

5. Worked Example: Single Surface Trace

Input: 100×50 grayscale image with horizontal dark band at rows 20–30

Graph construction:

- 100 columns $\times$ 50 rows = 5000 nodes
- Source connects to row 0 of each column (capacity ∞)
- Sink connects to row 49 of each column (capacity ∞)
- Vertical edges within columns (capacity = pixel intensity)
- Horizontal smoothness edges between adjacent columns (capacity based on δ)

Cost assignment: Dark pixels (low intensity) have low cost $\Rightarrow$ surface prefers dark regions

Max-flow execution:

1. Initial flow: Source $\rightarrow$ all top pixels $\rightarrow$ sink (high cost due to bright pixels)
2. Algorithm finds cheaper path through dark band
3. Cut occurs at transition from dark band to bright pixels below
4. Final surface: $s(x) \approx 30$ for all x (bottom of dark band)

Surface extraction

```
segmentation_line = segmentation.shape[0] - np.argmax(segmentation[::-1, :], axis
    =0) - 1
```

Finds lowest row marked as "above surface" in each column.

6. Cheat-Sheet Section

Core formulas:

- **Smoothness:** $|s(x,y) - s(x \pm 1, y)| \leq \Delta_x$
- **On-surface cost:** $\sum_{x,y} c(x,y,s(x,y))$
- **In-region cost:** $\sum_{x,y,z \leq s(x,y)} c_{\text{below}}(x,y,z) + \sum_{z > s(x,y)} c_{\text{above}}(x,y,z)$
- **Overlap:** $\delta_{\text{low}} \leq s_2 - s_1 \leq \delta_{\text{high}}$

Key parameters:

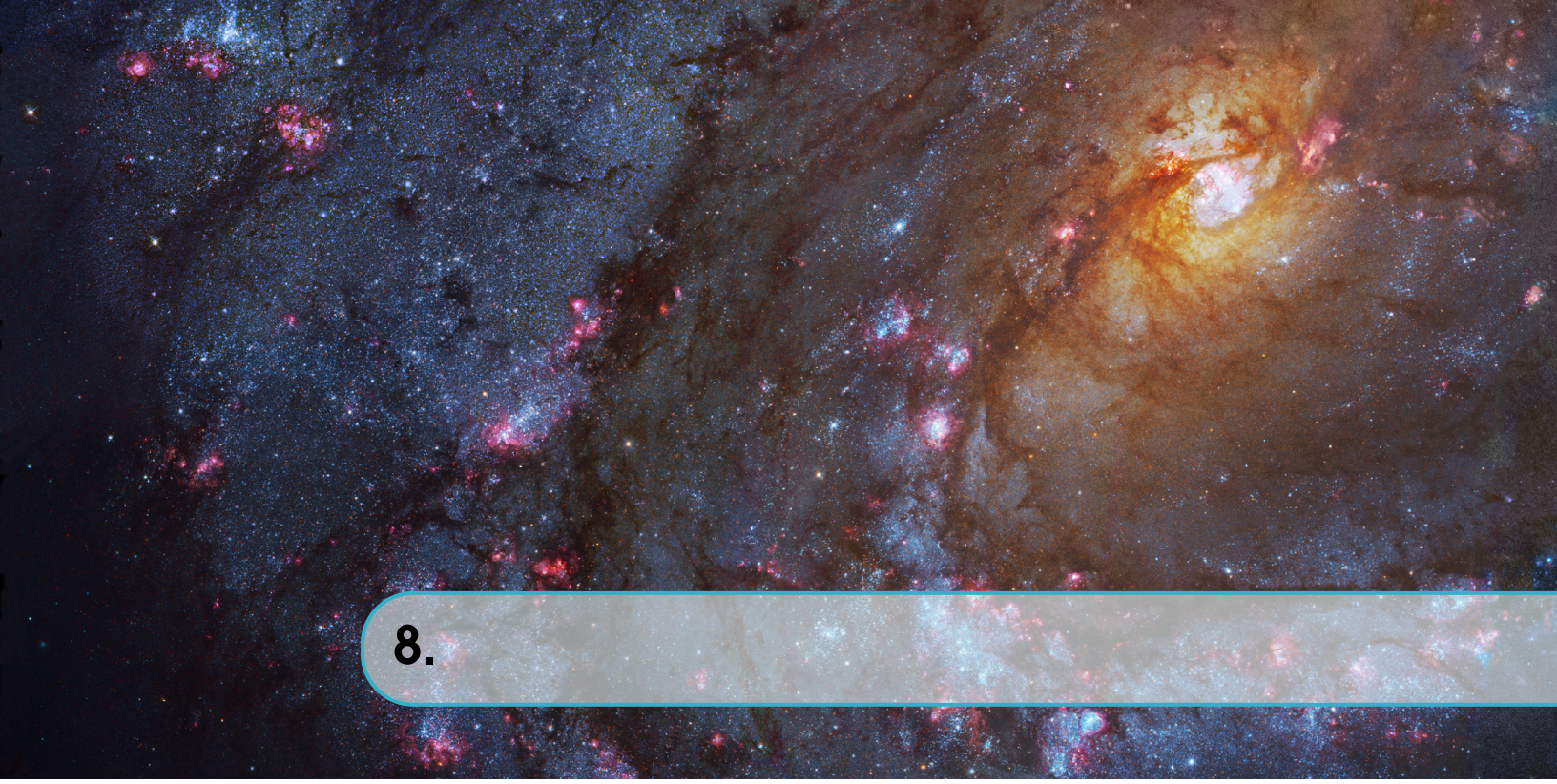
- `delta`: smoothness (1 = very smooth, 10 = rough)
- `wrap`: False for terrain, True for circular
- `min_margin` / `max_margin`: surface separation bounds

Cost design patterns:

- Dark line detection: `cost = image_intensity`
- Bright line detection: `cost = 255 - image_intensity`
- Edge detection: `cost = -gradient_magnitude`
- Region separation: Use in-region costs with complementary volumes

7. Reflective Questions

1. **Theory-Practice Gap:** Why does the limestone example use `0*I` (zero cost) for the GraphObject in region-based detection, while dental uses actual gradients? What does this reveal about on-surface vs in-region cost philosophy?
2. **Constraint Interaction:** If you set `min_margin=100` and `max_margin=50` for two surfaces, what happens during graph construction? How would you debug such errors?
3. **Coordinate Transform Impact:** In the dental example, what would happen if you estimated the center incorrectly by 20 pixels? How would this affect both the unwrapped image and the final segmentation quality?
4. **Cost Function Design:** For detecting the inner and outer boundaries of abdominal fat in MRI, would you use gradients (like dental) or region-based costs (like limestone)? Justify based on the underlying image characteristics.
5. **Scalability Trade-offs:** The algorithm has $\mathcal{O}(n^3)$ complexity. For a $512 \times 512 \times 100$ volume, estimate the memory requirements and suggest practical modifications for real-time medical imaging applications.



8.

1. High-Level Overview

This topic introduces **multilayer perceptrons (MLPs)** as fundamental building blocks for image analysis tasks. The course covers both theoretical foundations and practical implementation of feed-forward neural networks, starting with simple 2D point classification before extending to image analysis problems. Students learn to implement forward propagation, backpropagation, and various optimization strategies from scratch. The emphasis is on understanding the mathematical foundations rather than using high-level libraries, preparing students for more complex deep learning architectures. Key concepts include gradient computation via chain rule, weight updates through stochastic gradient descent, and handling numerical stability issues that arise in practice.

2. Key Theoretical Concepts

Forward Propagation

Definition: The process of computing network output by passing input through successive layers with linear transformations followed by non-linear activations.

Mathematical formulation:

- Hidden layer: $z_i = \sum_{d=0}^{D} w_{id}^{(1)} x_d$, $h_i = \text{ReLU}(z_i) = \max(0, z_i)$
- Output layer: $\hat{y}_j = \sum_{m=0}^{M} w_{jm}^{(2)} h_m$
- Softmax: $y_j = \frac{\exp(\hat{y}_j)}{\sum_{k=1}^{K} \exp(\hat{y}_k)}$

Intuition: Each layer learns increasingly complex representations, with ReLU providing non-linearity and softmax ensuring probabilistic outputs.

Backpropagation Algorithm

Definition: Efficient computation of gradients using the chain rule, propagating errors backward through the network.

Key equations:

- Output layer: $\delta_i^{(l)} = y_i - t_i$
- Hidden layers: $\delta_i^{(l)} = a'(z_i^{(l)}) \sum_k w_{ki}^{(l+1)} \delta_k^{(l+1)}$
- Weight updates: $w_{ij}^{(l)} = w_{ij}^{(l)} - \eta \delta_i^{(l)} h_j^{(l-1)}$

Proof sketch: The chain rule decomposes $\frac{\partial L}{\partial w_{ij}^{(l)}} = \frac{\partial L}{\partial z_i^{(l)}} \cdot \frac{\partial z_i^{(l)}}{\partial w_{ij}^{(l)}}$, where the second term is simply $h_j^{(l-1)}$ and the first is computed recursively.

Cross-Entropy Loss

Definition: $L = -\sum_{k=1}^{K} t_k \ln y_k$ where t_k is one-hot encoded target.

Properties: Convex, differentiable, heavily penalizes confident wrong predictions. For one-hot targets, simplifies to $L = -\ln y_{k^*}$ where k^* is the true class.

Weight Initialization

Theorem: For ReLU networks, initialize weights as $w \sim \mathcal{N}(0, \sqrt{2/n})$ where n is the number of inputs to the neuron.

Intuition: This Xavier/He initialization maintains variance across layers, preventing vanishing/-exploding gradients.

3. Implementation Walk-Through

Weight Initialization Function

Purpose: Initialize weight matrices with appropriate scaling to ensure stable training.

```
def initialize(mlp_size):
    W = []
    for l in range(len(mlp_size) - 1):
        size = (mlp_size[l] + 1, mlp_size[l + 1]) # +1 for bias
        W.append(rg.normal(scale=np.sqrt(2/size[0]), size=size))
    return W
```

Algorithm:

1. For each layer transition, create weight matrix of size $(\text{input\_neurons} + 1, \text{output\_neurons})$
2. Sample from normal distribution with standard deviation $\sqrt{2/\text{fan\_in}}$
3. Store bias weights in the last row of each matrix

Complexity: $O(\text{total\_parameters})$, **Pitfall:** Wrong scaling leads to vanishing/exploding gradients.

Forward Pass with Hidden Layer Storage

Purpose: Compute network output while storing intermediate values needed for backpropagation.

```
def forward(x, W):
    h = []
    c = x
    for l in range(len(W) - 1):
        c = W[l].T @ np.vstack((c, np.ones(c.shape[1]))) # Linear transformation
        c = np.maximum(c, 0) # ReLU activation
        h.append(c)
```

```
    c = W[-1].T @ np.vstack((c, np.ones(c.shape[1]))) # Final layer (no activation
        )
    h.append(c)
    return h
```

Algorithm

1. Add bias terms by stacking ones
2. Apply linear transformation: $W^\top @ [\text{activations}; 1]$
3. Apply ReLU activation (except final layer)
4. Store all intermediate activations for backprop

Complexity: $O(\text{total\_parameters} \times \text{batch\_size})$, **Pitfall:** Forgetting to store activations breaks backprop.

Backpropagation Implementation

Purpose: Compute gradients and update weights using stored forward pass values.

```
def backward(x, t, W, eta):
    h = forward(x, W)
    m = x.shape[1]

    # Softmax computation
    y = np.exp(h[-1])
    y = np.clip(y, 1e-15, 1e15) # Numerical stability
    y *= (1 / y.sum(axis=0))

    # Cross-entropy loss
    loss = -np.log(y[t]).sum()

    # Backward pass
    delta = y - t # Output layer error

    for l in range(len(W) - 1, 0, -1):
        Q = np.vstack((h[l-1], np.ones(h[l-1].shape[1]))) @ delta.T
        delta = W[l][:-1, :] @ delta # Remove bias weights
        delta *= h[l-1] > 0 # ReLU derivative
        W[l] -= (eta/m) * Q

    # First layer update
    Q = np.vstack((x, np.ones(x.shape[1]))) @ delta.T
    W[0] -= eta * Q

    return W, loss
```

Algorithm

1. Compute softmax output and cross-entropy loss
2. Initialize delta with output error: $y - t$

3. For each layer (backward), compute gradient matrix Q and propagate error
4. Apply ReLU derivative: multiply by $(h > 0)$
5. Update weights: $W -=$ learning_rate $*$ gradient

Complexity: $O(\text{total\_parameters} \times \text{batch\_size})$, **Pitfall:** Numerical instability without clipping.

Training Loop with Mini-batches

Purpose: Orchestrate training with proper data shuffling and batch processing.

```
def train(X, T, W, nr_epoch, eta, batchsize=1, losses=[]):
    nr_points = X.shape[1]

    for e in range(nr_epoch):
        random_order = rg.permutation(range(nr_points))
        epoch_loss = 0

        for k in range(0, nr_points, batchsize):
            batch = random_order[k:k+batchsize]
            X_batch = X[:, batch]
            T_batch = T[:, batch]

            W, loss = backward(X_batch, T_batch, W, eta)
            epoch_loss += loss

        losses.append(epoch_loss)
    return W, losses
```

Algorithm

1. Shuffle data indices for each epoch
2. Process data in mini-batches
3. Accumulate loss across batches
4. Track epoch losses for monitoring

Complexity: $O(\text{epochs} \times \text{total\_parameters} \times \text{dataset\_size})$, **Pitfall:** Not shuffling leads to poor convergence.

4. Crucial Details & 'Exam Traps'

- **Bias handling:** Always add bias as additional row/column in weight matrices; never forget to include ones in input
- **Matrix dimensions:** For layer with n inputs and m outputs, weight matrix is $(n+1) \times m$ with bias in last row
- **ReLU derivative:** 1 if $z > 0$, else 0 - implemented as $(h > 0)$ where h is activated output
- **Softmax stability:** Clip exponentials to prevent overflow: `np.clip(exp_values, 1e-15, 1e15)`
- **Loss computation:** For one-hot targets, use boolean indexing: `loss = -np.log(y[t])` instead of full summation
- **Gradient scaling:** Divide by batch size m to get proper gradient magnitude

- **Data standardization:** Always standardize inputs: $(x - \text{mean})/\text{std}$ for numerical stability
- **Weight initialization order:** Initialize before training, not inside training loop
- **Array broadcasting:** Ensure consistent dimensions when adding bias terms

5. Worked Example

Scenario: 2-class classification with 2D input, single hidden layer (3 neurons)
Network architecture: $[2,3,2]$ (input $\rightarrow$ hidden $\rightarrow$ output)

Forward pass trace:

```
# Input: x = [[1.0], [0.5]] (2x1)
# Weights: W[0] shape (3x3), W[1] shape (4x2)

# Layer 1:
x_with_bias = [[1.0], [0.5], [1.0]] # Add bias term
z = W[0].T @ x_with_bias # (3x1) output
h = np.maximum(z, 0) # ReLU activation

# Layer 2:
h_with_bias = [h[0], h[1], h[2], 1.0] # Add bias
y_hat = W[1].T @ h_with_bias # (2x1) output
y = softmax(y_hat) # Final probabilities
```

Backward pass trace:

```
# Target: t = [1, 0] (one-hot for class 0)
# Error at output: delta = y - t = [y[0]-1, y[1]-0]

# Update W[1]:
Q = outer_product(h_with_bias, delta) # (4x2)
W[1] -= learning_rate * Q

# Propagate error to hidden layer:
delta_hidden = W[1][:-1, :] @ delta # Remove bias row
delta_hidden *= (h > 0) # Apply ReLU derivative

# Update W[0]:
Q = outer_product(x_with_bias, delta_hidden) # (3x3)
W[0] -= learning_rate * Q
```

6. Cheat-Sheet Section

- **ReLU:** $\max(0,z)$, derivative: 1 if $z > 0$ else 0
- **Softmax:** $\exp(z_i)/\sum(\exp(z_j))$ for all j
- **Cross-entropy:** $-\log(y_{\text{target\_class}})$
- **Weight init:** $\text{Normal}(0, \sqrt{2/\text{fan\_in}})$ for ReLU
- **Gradient update:** $W_{\text{new}} = W_{\text{old}} - \text{learning\_rate} * \text{gradient}$

- **Delta output:** $y - t$ for softmax + cross-entropy
- **Delta hidden:** $(W_{\text{next}}@\text{delta\_next}) * \text{activation\_derivative}$
- **Bias handling:** Always add `ones` vector to activations
- **Numerical stability:** Clip exponentials, avoid $\log(0)$

7. Reflective Questions

1. **Theory-Code Bridge:** Why does the reference code compute $Q = \text{outer\_product(activations, delta)}$ instead of directly updating weights? How does this relate to the mathematical gradient formula?
2. **Architecture Impact:** Given the same dataset, how would you expect training time and final accuracy to change when moving from $[2,3,2]$ to $[2,10,10,2]$ architecture? Consider both benefits and drawbacks.
3. **Numerical Stability:** The code clips exponential values in softmax. What would happen without this clipping? Construct a specific input that would cause problems and explain the failure mode.
4. **Gradient Flow:** In the backpropagation implementation, why is the bias row excluded when computing $\text{delta} = W[1][:-1,:]@\text{delta}$? What would happen if we included it?
5. **Mini-batch vs. Stochastic:** Compare the convergence properties of batch size 1 vs. full-batch training. In what scenarios would you prefer each approach, and how does this relate to the bias-variance tradeoff in gradient estimates?

Concrete Numerical Examples

Here are detailed, step-by-step examples with actual numbers that you can verify by hand:

Example 1: Complete Forward Pass with Numbers

Setup: Simple network $[2,2,2]$ (2 inputs $\rightarrow$ 2 hidden $\rightarrow$ 2 outputs)

Given weights:

```
# W[0]: input to hidden (3x2 including bias)
W[0] = [[0.5, -0.3], # weights from x1
       [0.2, 0.4], # weights from x2
       [0.1, -0.1]] # bias weights

# W[1]: hidden to output (3x2 including bias)
W[1] = [[0.6, 0.2], # weights from h1
       [-0.4, 0.5], # weights from h2
       [0.3, -0.2]] # bias weights
```

Input: $\mathbf{x} = [2.0, 1.0]$

Step-by-step Forward Pass:

Layer 1 (Input $\rightarrow$ Hidden):

```
# Add bias: x_with_bias = [2.0, 1.0, 1.0]
# Linear transformation: z = W[0].T @ x_with_bias
```

```
z[0] = 0.5*2.0 + 0.2*1.0 + 0.1*1.0 = 1.0 + 0.2 + 0.1 = 1.3
z[1] = -0.3*2.0 + 0.4*1.0 + (-0.1)*1.0 = -0.6 + 0.4 - 0.1 = -0.3

# ReLU activation: h = max(0, z)
h[0] = max(0, 1.3) = 1.3
h[1] = max(0, -0.3) = 0.0
```

Layer 2 (Hidden → Output):

```
# Add bias: h_with_bias = [1.3, 0.0, 1.0]
# Linear transformation: y_hat = W[1].T @ h_with_bias

y_hat[0] = 0.6*1.3 + (-0.4)*0.0 + 0.3*1.0 = 0.78 + 0 + 0.3 = 1.08
y_hat[1] = 0.2*1.3 + 0.5*0.0 + (-0.2)*1.0 = 0.26 + 0 - 0.2 = 0.06

# Softmax: y = exp(y_hat) / sum(exp(y_hat))
exp_y_hat = [exp(1.08), exp(0.06)] = [2.945, 1.062]
sum_exp = 2.945 + 1.062 = 4.007

y[0] = 2.945 / 4.007 = 0.735
y[1] = 1.062 / 4.007 = 0.265
```

Final Output:

$\mathbf{y} = [0.735, 0.265]$ (probabilities for class 0 and class 1)

Example 2: Complete Backward Pass with Numbers

Continuing from Example 1 with target $\mathbf{t} = [1, 0]$ (true class is 0)

Loss Calculation:

```
# Cross-entropy loss
L = -log(y[0]) = -log(0.735) = 0.307
```

Step-by-step Backward Pass:

Output Layer Error:

```
delta_output = y - t = [0.735, 0.265] - [1, 0] = [-0.265, 0.265]
```

Update W(1) (Hidden → Output weights):

```
# Gradient matrix: Q = h_with_bias @ delta_output
# h_with_bias = [1.3, 0.0, 1.0], delta_output = [-0.265, 0.265]

Q[1] = [[1.3 * -0.265, 1.3 * 0.265], # = [-0.345, 0.345],
        [0.0 * -0.265, 0.0 * 0.265], # [0.0, 0.0],
        [1.0 * -0.265, 1.0 * 0.265]] # [-0.265, 0.265]]
```

```
# Weight update (learning_rate = 0.1):
W[1]_new = W[1] - 0.1 * Q[1]
W[1]_new = [[0.6, 0.2], - 0.1 * [[-0.345, 0.345],
           [-0.4, 0.5], [0.0, 0.0],
           [0.3, -0.2]] [-0.265, 0.265]]

W[1]_new = [[0.6345, 0.1655],
           [-0.4, 0.5],
           [0.3265, -0.2265]]
```

Propagate Error to Hidden Layer:

```
# Remove bias row from W[1]: W[1]_no_bias = [[0.6, 0.2], [-0.4, 0.5]]
# delta_hidden = W[1]_no_bias @ delta_output

delta_hidden[0] = 0.6 * (-0.265) + 0.2 * 0.265 = -0.159 + 0.053 = -0.106
delta_hidden[1] = -0.4 * (-0.265) + 0.5 * 0.265 = 0.106 + 0.1325 = 0.2385

# Apply ReLU derivative: multiply by (h > 0)
# h = [1.3, 0.0], so (h > 0) = [True, False] = [1, 0]
delta_hidden = [-0.106, 0.2385] * [1, 0] = [-0.106, 0.0]
```

Update W(0) (Input $\rightarrow$ Hidden weights):

```
# Gradient matrix: Q = x_with_bias @ delta_hidden
# x_with_bias = [2.0, 1.0, 1.0], delta_hidden = [-0.106, 0.0]

Q[0] = [[2.0 * -0.106, 2.0 * 0.0], # = [-0.212, 0.0],
       [1.0 * -0.106, 1.0 * 0.0], # [-0.106, 0.0],
       [1.0 * -0.106, 1.0 * 0.0]] # [-0.106, 0.0]]

# Weight update:
W[0]_new = W[0] - 0.1 * Q[0]
W[0]_new = [[0.5, -0.3], - 0.1 * [[-0.212, 0.0],
           [0.2, 0.4], [-0.106, 0.0],
           [0.1, -0.1]] [-0.106, 0.0]]

W[0]_new = [[0.5212, -0.3],
           [0.2106, 0.4],
           [0.1106, -0.1]]
```

Example 3: Weight Initialization by Hand

Network: $[2, 3, 2]$

Layer 1: Input(2) $\rightarrow$ Hidden(3)

```
# Weight matrix size: (2+1) x 3 = 3 x 3 (including bias)
# Standard deviation: sqrt(2/3) = sqrt(0.667) = 0.816

# Sample from Normal(0, 0.816):
W[0] = [[0.5, -0.3, 0.8], # from x1
       [0.2, 0.4, -0.6], # from x2
       [0.1, -0.1, 0.3]] # bias
```

Layer 2: Hidden(3) $\rightarrow$ Output(2)

```
# Weight matrix size: (3+1) x 2 = 4 x 2 (including bias)
# Standard deviation: sqrt(2/4) = sqrt(0.5) = 0.707

# Sample from Normal(0, 0.707):
W[1] = [[0.6, 0.2], # from h1
       [-0.4, 0.5], # from h2
       [0.3, -0.2], # from h3
       [0.15, -0.1]] # bias
```

Example 4: Mini-batch Processing

Batch: 3 samples

```
# Input batch: X = [[2.0, 1.0, 0.5], # x1 values for 3 samples
# [1.0, 2.0, 1.5]] # x2 values for 3 samples

# Targets: T = [[1, 0, 1], # class labels for 3 samples
# [0, 1, 0]]
```

Forward Pass for Batch:

```
# Add bias row: X_with_bias = [[2.0, 1.0, 0.5],
# [1.0, 2.0, 1.5],
# [1.0, 1.0, 1.0]]

# Layer 1: Z = W[0].T @ X_with_bias (result: 2x3 matrix)
# Using W[0] from Example 1:

Z[0,:] = [0.5*2.0 + 0.2*1.0 + 0.1*1.0, # Sample 1: 1.3
          0.5*1.0 + 0.2*2.0 + 0.1*1.0, # Sample 2: 1.0
          0.5*0.5 + 0.2*1.5 + 0.1*1.0] # Sample 3: 0.65

Z[1,:] = [-0.3*2.0 + 0.4*1.0 + (-0.1)*1.0, # Sample 1: -0.3
          -0.3*1.0 + 0.4*2.0 + (-0.1)*1.0, # Sample 2: 0.4
          -0.3*0.5 + 0.4*1.5 + (-0.1)*1.0] # Sample 3: 0.35

# ReLU: H = max(0, Z)
H = [[1.3, 1.0, 0.65],
     [0.0, 0.4, 0.35]]
```

Gradient Computation for Batch:

```
# After computing Y and loss for all 3 samples...
# Delta_output shape: 2x3
Delta_output = [[-0.265, 0.123, -0.089], # Class 0 errors
                [0.265, -0.123, 0.089]] # Class 1 errors

# Gradient for W[1]: Q = H_with_bias @ Delta_output.T
# Average over batch: Q = Q / batch_size
```

Example 5: Numerical Stability Issues

Dangerous Softmax Input:

```
y_hat = [100.0, 95.0] # Very large values

# Without clipping:
exp_y_hat = [exp(100), exp(95)] = [2.7x10^43, 4.2x10^41]
# This causes overflow! Result: [inf, inf]

# With clipping:
y_hat_clipped = [15.0, 15.0] # Clip to reasonable range
exp_y_hat = [exp(15), exp(15)] = [3.3x10^6, 3.3x10^6]
softmax = [0.5, 0.5] # Stable result
```

Loss Computation Edge Case:

```
# Predicted probability very close to 0
y = [0.0001, 0.9999]
t = [1, 0] # True class is 0

# Loss = -log(y[0]) = -log(0.0001) = 9.21
# Without epsilon: -log(0) = inf (disaster!)

# With epsilon: y_safe = max(y, 1e-15)
y_safe = [1e-15, 0.9999]
loss = -log(1e-15) = 34.54 # Large but finite
```

Quick Verification Exercises

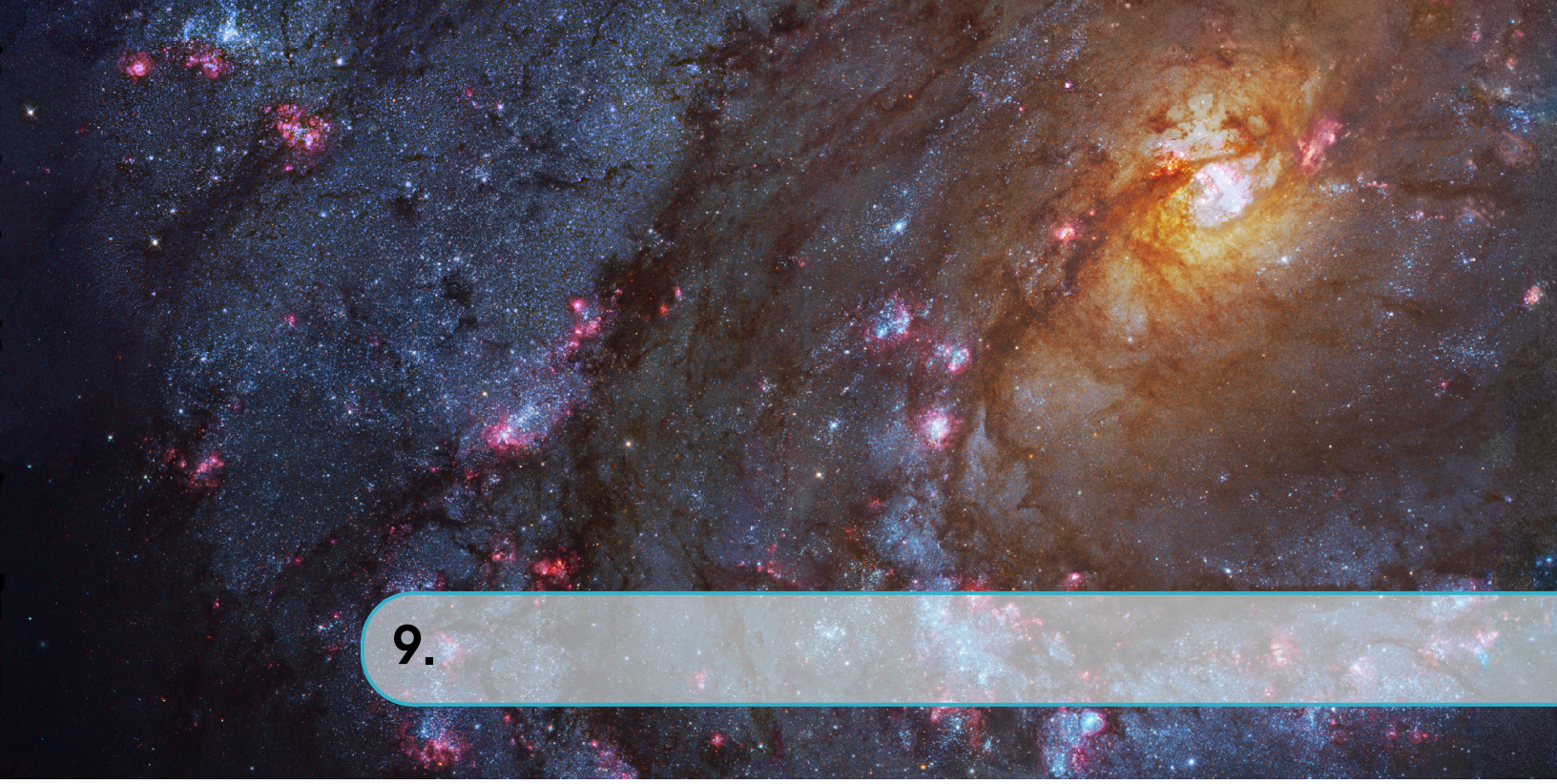
Exercise 1: Using the weights from Example 1, compute the forward pass for input $\mathbf{x} = [1.0, 2.0]$. Expected intermediate values:

- $z = [0.9, 0.4]$
- $h = [0.9, 0.4]$
- $y\_hat = [0.78, 0.42]$

Exercise 2: If the learning rate is 0.05 instead of 0.1 in Example 2, what would be the new value of $W[1][0,0]$? (Answer: 0.61725)

Exercise 3: For a batch of size 2 with identical inputs $\mathbf{x} = [1.0, 1.0]$, how would the gradient magnitude compare to processing each sample individually? (Answer: Same direction, same magnitude due to averaging)

These examples show the actual arithmetic behind neural network operations, helping you understand exactly what happens at each step!



9.

High-Level Overview ($\leq$ 150 words)

This exercise focuses on building **fully connected neural networks** for image classification tasks, progressing from simple to complex datasets. Students start with handwritten digit classification using **MNIST** (28×28 grayscale images), advance to **CIFAR-10** (32×32 color photographs), and culminate with a competition using the novel **BugNIST 2D dataset** (insect classification with 12 classes).

The exercise emphasizes practical implementation aspects: transforming 2D images into vectors for MLP input, proper data standardization, one-hot encoding for targets, and train/validation splitting to prevent overfitting. Key optimization techniques include **minibatch training, momentum, and adaptive learning rates**. Regularization methods cover **data augmentation, dropout, and noise injection**.

This topic bridges theoretical neural network concepts with real-world computer vision applications, teaching students to handle the curse of dimensionality (784-3072 input dimensions) while achieving robust classification performance through proper network architecture and training strategies.

Key Theoretical Concepts

Image Vectorization for MLPs

- **Definition:** Converting 2D/3D image arrays into 1D feature vectors for fully connected networks.
- **Properties:** For MNIST (28×28) $\rightarrow$ 784 dimensions, CIFAR-10 ($32 \times 32 \times 3$) $\rightarrow$ 3072 dimensions.
- **Intuitive explanation:** MLPs require fixed-size vector inputs, so spatial structure is temporarily abandoned.
- **Implementation:** `X = images.reshape((images.shape[0], -1)).T`

Data Standardization Strategies

- **Definition:** Normalizing pixel intensities to improve convergence.
- **Methods:**
  - Range normalization: $(\text{pixel\_values}/255) * 2 - 1 \rightarrow [-1, 1]$
  - Z-score normalization: $(X - \mu)/\sigma \rightarrow$ zero mean, unit variance.
  - Per-image normalization: Each image gets zero mean and unit norm.
- **Why necessary:** Prevents gradient explosion and ensures balanced feature contributions.

One-Hot Encoding for Classification

- **Definition:** Converting class labels to binary vectors.
- **Properties:** For C classes, target becomes C-dimensional vector with single 1.
- **Implementation:** `I = np.eye(num_classes, dtype=bool); T = I[targets].T`
- **Purpose:** Enables softmax output interpretation as class probabilities.

Validation Strategy

- **Definition:** Reserving portion of training data to monitor generalization.
- **Properties:** Typically 10-20% of training set withheld.
- **Purpose:** Early stopping when validation accuracy degrades (overfitting detection).
- **Critical insight:** Never use test set for hyperparameter tuning.

Minibatch Training

- **Definition:** Computing gradients on small subsets rather than entire dataset.
- **Properties:** Batch size typically 10-100 samples.
- **Benefits:** Vectorized operations, noise reduction in gradients, higher learning rates possible.
- **Implementation:** `batch = random_order[k:k+batchsize]`

Implementation Walk-Through

Data Loading and Preprocessing Pipeline

```
# Load images and convert to vectors
X_train = np.stack([np.array(PIL.Image.open(path + filename))
                for filename in train_filenames], axis=-1)
X_train = 2/255 * X_train.reshape(-1, nr_images).astype('float') - 1

# One-hot encode targets
I = np.eye(12, dtype=bool)
T_train = I[train_targets].T
```

Purpose:

Transform raw image files into normalized feature matrices and one-hot target matrices.

Algorithm:

1. Stack all images into 3D array (height $\times$ width $\times$ num_images).
2. Reshape to 2D matrix (pixels $\times$ images).
3. Normalize pixel values from [0, 255] to [-1, 1].
4. Convert integer class labels to binary indicator matrix.

Complexity:

$O(n \cdot d)$ where n = number of images, d = pixels per image.

Pitfall:

Memory usage scales with dataset size—consider lazy loading for large datasets.

Training Loop with Minibatches

```
for e in range(nr_epoch):
    random_order = rg.permutation(range(nr_points))
    epoch_loss = 0

    for k in range(0, nr_points, batchsize):
        batch = random_order[k:k+batchsize]
        X_batch = X_train[:, batch]
        T_batch = T_train[:, batch]

        W, loss = mlp.backward(X_batch, T_batch, W, eta)
        epoch_loss += loss
```

Purpose:

Train network weights using stochastic gradient descent with minibatches.

Algorithm:

1. Shuffle training indices for each epoch.
2. Partition shuffled data into fixed-size batches.
3. Forward pass $\rightarrow$ compute loss $\rightarrow$ backward pass $\rightarrow$ update weights.
4. Accumulate losses for epoch monitoring.

Complexity:

$O(\text{epochs} \cdot \text{batches per epoch} \cdot \text{batch size} \cdot \text{network operations})$

Pitfall:

Last batch may have different size—ensure your MLP handles variable batch sizes.

Evaluation and Confusion Matrix Visualization

```
def show_confusion_matrix(ax, target, predicted):
    nr_classes = target.max() + 1
    edges = np.arange(nr_classes + 1) - 0.5
    cm = np.histogram2d(predicted, target, [edges, edges])[0]
    ax.imshow(cm + 1, cmap=plt.cm.plasma, norm=matplotlib.colors.LogNorm())
```

Purpose:

Visualize classification performance and identify commonly confused classes.

Algorithm:

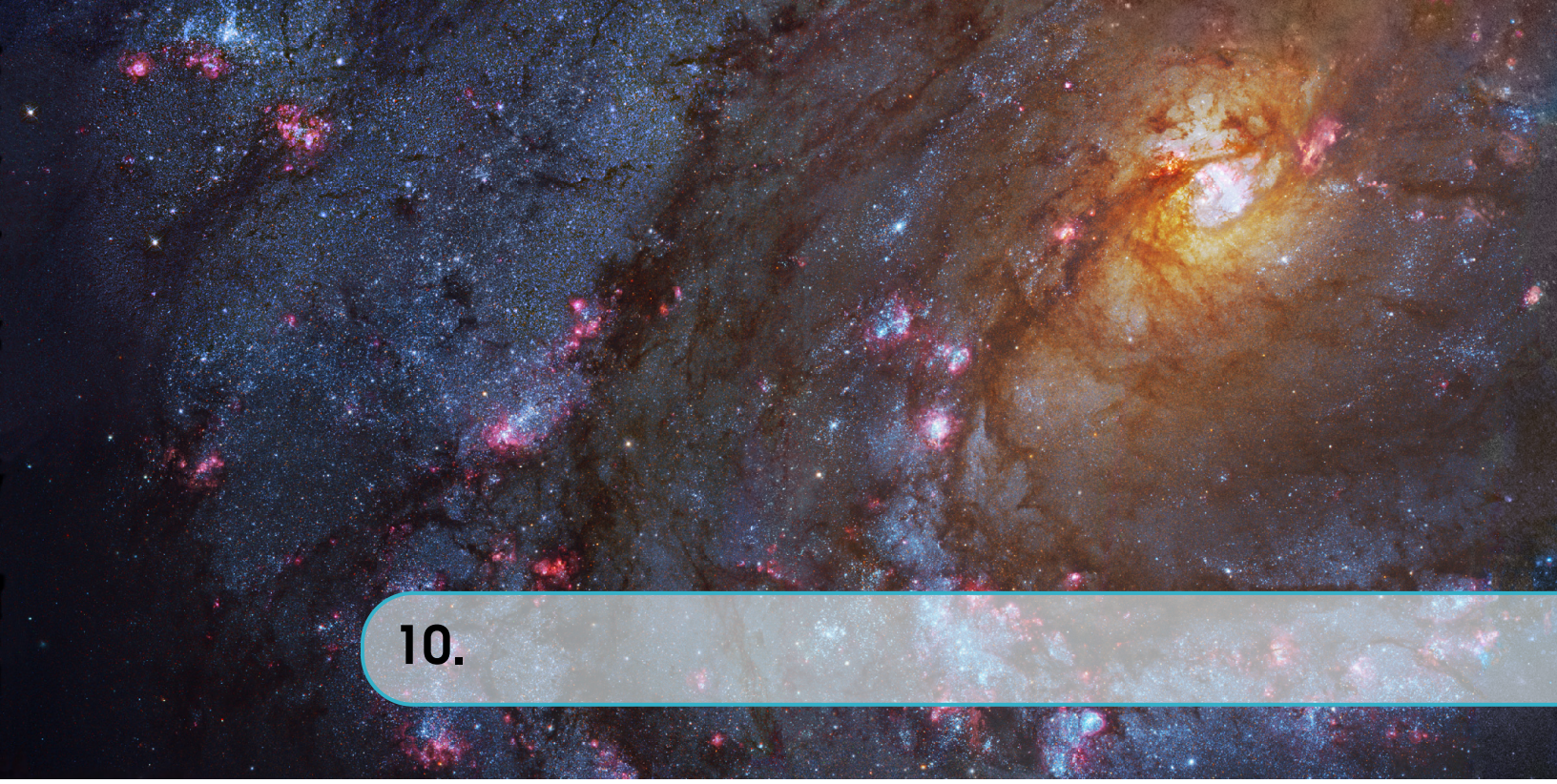
1. Compute 2D histogram of (predicted, target) pairs.
2. Display as heatmap with logarithmic color scaling.
3. Diagonal elements = correct classifications, off-diagonal = errors.

Complexity:

$O(n)$ for n predictions

Insight:

Off-diagonal patterns reveal systematic misclassification tendencies.



10.

1. High-Level Overview ($\leq$ 150 words)

This lecture introduces **Convolutional Neural Networks (CNNs)** with a focus on the **U-Net architecture** for image segmentation tasks. Unlike multilayer perceptrons (MLPs), CNNs leverage **weight sharing** through convolution operations, dramatically reducing parameters while maintaining spatial relationships. The U-Net is a **Fully Convolutional Network (FCN)** designed specifically for dense prediction tasks like medical image segmentation.

The practical component involves analyzing a pre-trained U-Net model for **gland segmentation** in histological colon tissue images. Students investigate the encoder-decoder architecture, examine learned filter weights, visualize intermediate feature maps, and understand how skip connections preserve spatial information. This builds foundational understanding of modern computer vision architectures, particularly relevant for biomedical applications where precise pixel-level predictions are crucial. The exercise bridges theoretical CNN concepts with hands-on implementation using PyTorch, emphasizing the practical aspects of deep learning for image analysis.

2. Key Theoretical Concepts

CNN vs MLP Architecture

- **Definition:** CNNs use convolution operations instead of fully connected layers, implementing **weight sharing** where the same filter weights are applied across spatial locations.
- **Formal properties:**
  - Translation equivariance: $f(T_x(input)) = T_x(f(input))$ where T_x is translation
  - Parameter reduction: For input size $n \times n$ and filter size $k \times k$, CNN uses k^2 parameters vs MLP's n^2 parameters per connection
- **Intuitive explanation:** Convolutions capture local spatial patterns that are useful regardless of position in the image.
- **Benefit:** Enables processing of much larger input images with fewer parameters.

U-Net Architecture

- **Definition:** Encoder-decoder architecture with **skip connections** linking corresponding encoder and decoder layers.
- **Formal structure:**
  - Encoder: Series of convolution + pooling operations that downsample and increase feature channels.
  - Decoder: Transposed convolutions that upsample while decreasing channels.
  - Skip connections: Concatenate encoder features with decoder features at matching resolutions.
- **Intuitive explanation:** Encoder captures "what" information (semantic features), decoder recovers "where" information (spatial localization), skip connections preserve fine-grained spatial details.
- **Mathematical flow:** Input $(128 \times 128 \times 3) \rightarrow$ encoding $\rightarrow (25 \times 25 \times 32) \rightarrow$ decoding $\rightarrow (88 \times 88 \times 2)$

Weight Sharing and Parameter Efficiency

- **Definition:** Multiple spatial locations share the same convolutional kernel weights.
- **Formal advantage:** Reduces parameters from $O(n^4)$ for fully connected to $O(k^2 \times \text{channels})$ for convolution.
- **Implementation:** Each 3×3 kernel applied across entire feature map rather than learning separate weights for each position.

3. Implementation Walk-Through

UNet128 Class Architecture

Purpose: Implements a U-Net variant that processes 128×128 RGB patches and outputs 88×88 segmentation maps.

Step-by-step algorithm:

```
def forward(self, x): # x: [batch, 3, 128, 128]
    # ENCODER PATH (downsampling)
    # Block 1: 128+126+124, channels 3+8+8
    l1 = self.relu(self.conv1B(self.relu(self.conv1A(x)))) # [batch, 8, 124, 124]

    # Block 2: 124+62+58, channels 8+16+16
    l2 = self.relu(self.conv2B(self.relu(self.conv2A(self.pool(l1))))) # [batch,
        16, 58, 58]

    # Block 3: 58+29+25, channels 16+32+32
    out = self.relu(self.conv3B(self.relu(self.conv3A(self.pool(l2))))) # [batch,
        32, 25, 25]

    # DECODER PATH (upsampling with skip connections)
    # Upsample and concatenate with L2
    out = torch.cat([self.convtrans34(out), l2[:, :, 4:4+50, 4:4+50]], dim=1) # [
        batch, 48, 50, 50]
```

```
    out = self.relu(self.conv4B(self.relu(self.conv4A(out)))) # [batch, 16, 46,
        46]

    # Upsample and concatenate with L1
    out = torch.cat([self.convtrans45(out), l1[:, :, 16:16+80, 16:16+80]], dim=1)
        # [batch, 24, 92, 92]
    out = self.relu(self.conv5B(self.relu(self.conv5A(out)))) # [batch, 8, 88, 88]

    # Final classification layer
    out = self.convfinal(out) # [batch, 2, 88, 88]
    return out
```

Complexity:

- **Time:** $O(k^2 \times C_{in} \times C_{out} \times H \times W)$ per convolution.
- **Space:** $O(\text{max feature map size})$ for storing intermediate activations.
- **Parameters:** 29,474 total (significantly less than equivalent MLP).

Common pitfall:

- Skip connection spatial alignment requires careful cropping (e.g., `[12:, 4:-4, 4:-4]`).
- Input size constraints (must be divisible by pooling factors).

Model Investigation Functions

Modified Forward Pass for Layer Visualization:

```
def forward_modified(self, x):
    # Store intermediate outputs for visualization
    l1 = self.relu(self.conv1B(self.relu(self.conv1A(x))))
    l2 = self.relu(self.conv2B(self.relu(self.conv2A(self.pool(l1)))))
    out = self.relu(self.conv3B(self.relu(self.conv3A(self.pool(l2)))))
    l3 = out.clone() # Bottleneck features

    # Decoder with stored outputs
    out = torch.cat([self.convtrans34(out), l2[:, :, 4:-4, 4:-4]], dim=1)
    out = self.relu(self.conv4B(self.relu(self.conv4A(out))))
    l4 = out.clone()

    out = torch.cat([self.convtrans45(out), l1[:, :, 16:-16, 16:-16]], dim=1)
    out = self.relu(self.conv5B(self.relu(self.conv5A(out))))
    l5 = out.clone()

    out = self.convfinal(out)
    return out, l1, l2, l3, l4, l5 # Return all intermediate layers
```

4. Crucial Details & 'Exam Traps'

- **Size constraints:** Input dimensions must allow for multiple pooling operations. Not all sizes are valid inputs.

- **Skip connection alignment:** Cropping is essential (e.g., `[4:-4, 16:-16]`) to match spatial dimensions after pooling/upsampling cycles.
- **Output size reduction:** 128×128 input $\rightarrow$ 88×88 output due to valid convolutions (no padding).
- **Channel progression:** Encoder increases channels ($3 \rightarrow 8 \rightarrow 16 \rightarrow 32$), decoder decreases ($32 \rightarrow 16 \rightarrow 8 \rightarrow 2$).
- **Transposed convolution:** Not deconvolution; it's upsampling followed by convolution.
- **Weight sharing:** Same 3×3 kernel applied across all spatial locations in a feature map.
- **Parameter counting:** Include both weights and biases for each layer.
- **Activation placement:** ReLU applied after each convolution before pooling/skip connections.

5. Worked Example/Trace

Input Processing Trace for $128 \times 128 \times 3$ image:

```
Input: [1, 3, 128, 128] RGB image

ENCODER:
conv1A: [1, 3, 128, 128] -> [1, 8, 126, 126] # 3x3 conv, no padding
conv1B: [1, 8, 126, 126] -> [1, 8, 124, 124]
l1 = [1, 8, 124, 124] # Store for skip connection

pool+conv2A: [1, 8, 124, 124] -> [1, 8, 62, 62] -> [1, 16, 60, 60]
conv2B: [1, 16, 60, 60] -> [1, 16, 58, 58]
l2 = [1, 16, 58, 58] # Store for skip connection

pool+conv3A: [1, 16, 58, 58] -> [1, 16, 29, 29] -> [1, 32, 27, 27]
conv3B: [1, 32, 27, 27] -> [1, 32, 25, 25] # Bottleneck

DECODER:
convtrans34: [1, 32, 25, 25] -> [1, 16, 50, 50] # 2x2 transposed conv, stride=2
Skip: cat([1, 16, 50, 50], l2[:, :, 4:-4, 4:-4]) -> [1, 32, 50, 50] # l2 cropped
    to [1, 16, 50, 50]
conv4A+4B: [1, 32, 50, 50] -> [1, 16, 46, 46]

convtrans45: [1, 16, 46, 46] -> [1, 8, 92, 92]
Skip: cat([1, 8, 92, 92], l1[:, :, 16:-16, 16:-16]) -> [1, 16, 92, 92] # l1
    cropped to [1, 8, 92, 92]
conv5A+5B: [1, 16, 92, 92] -> [1, 8, 88, 88]

convfinal: [1, 8, 88, 88] -> [1, 2, 88, 88] # Final segmentation logits
```

Parameter calculation example:

- `conv1A`: $(3 \times 8 \times 3 \times 3) + 8 = 216 + 8 = 224$ parameters.
- Total parameters: 29,474 across all layers.

6. Cheat-Sheet Section

- **Size formula:** output_size $=$ input_size $-$ kernel_size $+ 1$ (valid convolution).

- **U-Net key:** Encoder (what) + Decoder (where) + Skip connections (details).
- **Parameter count:** in_channels $\times$ out_channels $\times k^2 +$ out_channels per conv layer.
- **Skip connection cropping:** Match spatial dimensions exactly.
- **Valid input sizes:** Must be divisible by $2^{\text{num\_pooling\_layers}}$.
- **Channel progression:** $3 \to 8 \to 16 \to 32 \to 16 \to 8 \to 2$ (encoder increases, decoder decreases).
- **Output size:** Always smaller than input due to valid convolutions.
- **Transposed conv:** Upsampling operation, not true deconvolution.