ChatGPT

Image Segmentation Techniques for Foreground Extraction

Otsu's Thresholding

Otsu's method automatically selects an optimal intensity threshold by analyzing the image histogram. It treats pixel intensities as two groups (foreground vs. background) and finds the threshold that **minimizes intra-class variance** (equivalently maximizes inter-class variance) [1]. Intuitively, Otsu's algorithm sweeps through all possible thresholds, computing class variances, and picks the one that best separates dark and light peaks in the histogram [1].

Steps:
1. Convert image to grayscale (if needed).
2. Compute the grayscale histogram of pixel intensities.
3. For each candidate threshold *T*, split pixels into two classes: below-T and above-T, and compute the weighted sum of variances of these classes.
4. Choose the *T* that minimizes this intra-class variance (or maximizes between-class variance). This *T* becomes the binarization threshold.
5. Apply threshold: pixels $\geq T \rightarrow$ foreground (1), else background (0).

Strengths: Simple and fast; no parameter tuning needed (it automatically finds *T*) [1]. Works well when the object and background intensities form distinct peaks in the histogram. Because it is based on global histogram statistics, it is robust to uniform lighting and noise to some extent.

Weaknesses: Assumes a bimodal histogram. If the foreground and background intensity distributions overlap or the image has uneven illumination, Otsu's threshold can fail, producing poor separation. It yields only a single global threshold (no multi-level segmentation) and may misclassify pixels in cases of gradual intensity change or shadows.

Example Illustration: A typical use-case is segmenting a dark object on a lighter background (or vice versa) by thresholding. For instance, a grayscale image of a document text can be binarized so that text becomes white (foreground) on black (background). The method is visualized by overlaying the chosen threshold on the histogram, which aligns between two peaks.

K-Means Clustering (Color-Based)

K-Means segmentation treats each pixel as a data point in color space and groups them into *k* clusters [2]. Intuitively, it partitions the image so that pixels within each cluster have similar color (or intensity) and are as distinct as possible from other clusters [2]. For color images, the algorithm often operates in an appropriate color space (e.g. RGB or Lab) to separate colors more effectively.

Steps:
1. Choose number of clusters *k* (e.g. 2 or 3 for simple foreground/background).
2. Reshape the image into an array of pixel color vectors (size *N*×3 for RGB).
3. Run K-Means: randomly initialize *k* centroids, assign each pixel to the nearest centroid, then update centroids to the mean of assigned pixels, iterating until convergence.
4. Assign each pixel the color of its cluster centroid to create a **clustered image** (for visualization) or generate a **label map**.
5. Identify which cluster(s) correspond to the foreground (e.g. the cluster with the object's predominant color or based on cluster size/brightness). Create a binary mask by selecting those cluster labels as foreground.

Strengths: Can separate multiple objects or colors (multi-modal). It is unsupervised (no training) and can capture complex color distributions. K-Means can work in different color spaces (e.g. Lab separates luminance from color [3]), improving segmentation when lighting varies. It is flexible: by adjusting *k* or color space, one can control granularity.

Weaknesses: Must pre-specify *k* and (often) which cluster(s) are foreground. If the background contains multiple colors or the object blends with background colors, K-Means may cluster incorrectly. It can be sensitive to initialization and may converge to local optima (random restarts or `k-means++` can help). Computationally, K-Means is more expensive than simple thresholding, especially for large images. Also, it ignores spatial continuity – pixels far apart but same color are clustered together, which may produce speckled masks.

Example Illustration: For a simple example, imagine segmenting a red apple on a green lawn. In RGB space, pixels cluster into "red" and "green" groups. Running K-Means with *k=2* will group the apple pixels together. The result can be visualized by coloring each pixel by its cluster centroid. One cluster mask then isolates the apple (foreground) from the grass (background).

Watershed Segmentation

Watershed views the grayscale image as a topographic surface: bright pixels = peaks, dark = valleys. It "floods" this landscape from chosen markers until basins meet at watershed lines. This effectively separates touching or overlapping objects by their **"watersheds"** (boundaries). Starting from markers (which can be user-defined or computed), the algorithm treats pixel values as elevations and floods regions, building barriers where different floods meet [4].

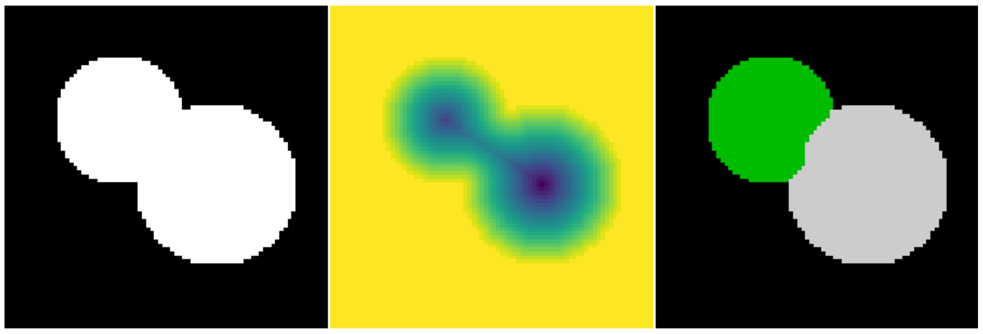

Figure: Illustration of watershed segmentation. Left: binary image with two overlapping objects. Middle: distance transform highlighting object centers. Right: final watershed labels (green/gray regions).

Working Principle: Each pixel flows "downhill" towards the nearest local minimum. By placing markers (e.g. one per object), water flooding from these markers will meet at the ridges. Those ridges define object boundaries. In practice, one often computes a distance transform on the binary foreground to get peaks at object centers and uses those peaks as markers [5].

Steps:
1. (Preprocessing) Obtain a clean binary foreground mask (e.g. with simple threshold or Otsu).
2. Compute the distance transform of this binary mask to emphasize object centers.
3. Find local maxima of the distance map to serve as **markers** (one per object). Label each marker distinctly.
4. Apply the watershed algorithm (e.g. OpenCV's `cv2.watershed` or skimage's `watershed`) using these markers. The pixels "flood" from markers until meeting.
5. Extract the segmented regions: each original object is separated by watershed lines. The algorithm assigns each pixel to a marker (region) or the boundary. A binary mask can be created by merging marker labels of interest.

Strengths: Excels at separating overlapping or touching objects. It is region-based, so even irregularly shaped objects can be separated if markers are placed correctly. Watershed can use prior information via markers, allowing for interactive or guided segmentation. It is deterministic given markers and handles noise by filling basins from peaks.

Weaknesses: Without proper markers, watershed often **over-segments**, producing too many regions (especially if the gradient has many local minima). It is sensitive to noise; spurious local minima create extra basins. Choosing or computing markers is critical. The method is more complex and computationally heavier. It typically requires good preprocessing (denoising, smoothing) and post-processing (merging small regions).

Example Illustration: Watershed is often demonstrated on two touching circles. By computing the distance transform (center panel above) and placing a marker at each circle's peak, watershed floods separate them along the touching boundary, yielding distinct labels (right panel). In practice, one might also smooth the gradient or apply morphological operations to clean results before flooding [5].

Canny Edge Detection + Morphology-Based Mask

This hybrid method first finds object **edges** and then builds a filled mask via morphological operations. **Canny Edge Detection** is a classic multi-stage algorithm that detects a wide range of edges with low error [6]. It involves Gaussian smoothing, gradient computation, non-maximum suppression, and double-thresholding (hysteresis) to produce thin edges. The output is a binary edge map highlighting object boundaries.

Steps (Canny):
1. **Noise Reduction:** Smooth the image with a Gaussian filter to reduce noise.
2. **Edge Gradient:** Compute intensity gradients (e.g. Sobel filter) in x and y directions to get edge magnitude and orientation.

3. **Non-Maximum Suppression:** Thin out the edges by keeping only local gradient maxima along the edge direction.
4. **Double Threshold & Hysteresis:** Use two thresholds (low and high). Edges above the high threshold are "strong"; those between low and high are "weak." Keep weak edges only if connected to strong edges. This yields a binary edge mask.

After obtaining edges, we apply **morphological operations** to close the contours and fill holes, producing a mask:
5. **Closing/Growing:** Dilate and then erode the edge image (a **closing** operation) to close small gaps in the edges [7]. This connects broken edge segments into continuous boundaries.
6. **Fill and Smooth:** Optionally, find contours from the closed-edge image and fill them (e.g. using `cv2.findContours` and `cv2.drawContours` with thickness = -1) to create a solid foreground mask. Finally, a slight smoothing (e.g. Gaussian blur on the mask) can soften edges.

Strengths: Canny reliably finds crisp edges even in noisy images, due to its built-in noise filtering and hysteresis. It is effective for objects with well-defined boundaries. The subsequent closing operation can join fragmented edges into a cohesive object mask. This approach is relatively parameter-light (two thresholds and one kernel size) and can precisely follow object shapes.

Weaknesses: Canny detects edges, not areas; it may miss edges if contrast is low or thresholds are not tuned. Edges may be discontinuous or incomplete, requiring careful morphological closing to avoid holes. The method is sensitive to noise and texture: too many edges (clutter) can confuse filling. Choosing appropriate thresholds and kernel size is crucial. It may struggle with objects without strong edges or with open contours (e.g. very thin parts).

Example Illustration: For example, detecting a leaf on a cluttered background: Canny will outline the leaf's perimeter. A morphological closing (dilation+erosion) will bridge small gaps along the outline, yielding a closed contour. Filling that closed contour produces a mask of the leaf (foreground) on a grayscale-desaturated background.

Python Implementation Example

Below is a modular Python pipeline implementing all four methods and a background-blending function. Each segmentation technique is encapsulated in a function. Logging statements record the method name, processing time, and basic mask statistics. For Otsu, an intensity histogram is plotted with the chosen threshold. Developers can reuse and adapt these functions in their pipelines.

```python
import cv2
import numpy as np
import logging
import time
import matplotlib.pyplot as plt
from scipy import ndimage as ndi
from skimage.feature import peak_local_max
from skimage.segmentation import watershed
```

```
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(message)s')

def segment_otsu(image):
    start = time.time()
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    # Compute Otsu's threshold
    ret, mask = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
    mask = mask.astype(np.uint8)
    elapsed = time.time() - start
    logging.info(f"Otsu's method: threshold={ret:.2f}, time={elapsed:.3f}s, "
                 f"mask coverage={(np.mean(mask>0)*100):.2f}%")
    # Plot histogram and threshold
    plt.figure(figsize=(4,3))
    plt.hist(gray.ravel(), bins=256, range=(0,256), color='gray', alpha=0.7)
    plt.axvline(ret, color='red', linestyle='--', label=f'Threshold={ret:.0f}')
    plt.legend(); plt.title("Otsu Histogram")
    plt.show()
    return mask

def segment_kmeans(image, k=3):
    start = time.time()
    pixel_vals = image.reshape((-1,3))
    pixel_vals = np.float32(pixel_vals)
    # Define criteria and apply kmeans()
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 100, 0.2)
    _, labels, centers = cv2.kmeans(pixel_vals, k, None, criteria, 10,
cv2.KMEANS_RANDOM_CENTERS)
    centers = np.uint8(centers)
    labels = labels.flatten()
    # Map each pixel to corresponding centroid color (for visualization)
    segmented = centers[labels].reshape(image.shape)
    # Choose one cluster as foreground (e.g. cluster 0 here; adjust as needed)
    fg_cluster = 0
    mask = (labels == fg_cluster).reshape(image.shape[:2]).astype(np.uint8) *
255
    elapsed = time.time() - start
    logging.info(f"K-Means (k={k}): time={elapsed:.3f}s, "
                 f"cluster sizes={np.bincount(labels)}, chosen_fg={fg_cluster}")
    return mask, segmented

def segment_watershed(image):
    start = time.time()
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    # Initial threshold to get sure background
    _, thresh = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
    # Distance transform for sure foreground
    dist = cv2.distanceTransform(thresh, cv2.DIST_L2, 5)
    # Marker: local maxima of distance transform
    local_max = peak_local_max(dist, indices=False, min_distance=20,
labels=thresh)
    markers, _ = ndi.label(local_max)
    # Apply watershed
    labels = watershed(-dist, markers, mask=thresh)
    # Construct mask: consider everything with label > 0 as foreground
    mask = np.uint8(labels > 0) * 255
    elapsed = time.time() - start
    logging.info(f"Watershed: time={elapsed:.3f}s, markers={markers.max()}, "
                 f"mask coverage={(np.mean(mask>0)*100):.2f}%")
    return mask

def segment_canny(image, low_thresh=50, high_thresh=150):
    start = time.time()
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    # Blur to reduce noise
    blur = cv2.GaussianBlur(gray, (5,5), 1.4)
    # Canny edge detector
    edges = cv2.Canny(blur, low_thresh, high_thresh)
    # Morphological closing to fill gaps (dilate then erode)
    kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (5,5))
    closed = cv2.morphologyEx(edges, cv2.MORPH_CLOSE, kernel)
    # Fill holes by finding contours
    contours, _ = cv2.findContours(closed, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
    mask = np.zeros_like(gray)
    cv2.drawContours(mask, contours, -1, 255, thickness=cv2.FILLED)
    elapsed = time.time() - start
    logging.info(f"Canny+Morphology: time={elapsed:.3f}s, "
                 f"edges pixels={(np.mean(edges>0)*100):.2f}%, mask
coverage={(np.mean(mask>0)*100):.2f}%")
    return mask

def blend_background(image, mask, blur_radius=21):
    """
    Desaturate and blur background given a foreground mask.
    Foreground remains in color; background is grayscale with smooth alpha
blending.
    """
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    gray_color = cv2.cvtColor(gray, cv2.COLOR_GRAY2BGR)
    # Create soft alpha matte by blurring the mask edges
    mask_float = mask.astype(np.float32) / 255.0
    alpha = cv2.GaussianBlur(mask_float, (blur_radius, blur_radius), 0)
    alpha = np.clip(alpha, 0, 1)[..., np.newaxis]
    # Blend: foreground color * alpha + gray background * (1 - alpha)
    blended = (image.astype(np.float32) * alpha + gray_color.astype(np.float32)
* (1 - alpha))
```

```
    blended = np.uint8(blended)
    return blended

# Example usage:
# img = cv2.imread('input.jpg')
# mask1 = segment_otsu(img)
# mask2, seg_img = segment_kmeans(img, k=3)
# mask3 = segment_watershed(img)
# mask4 = segment_canny(img)
# final = blend_background(img, mask3)  # blend using watershed mask, for
example
```

Each function logs the method, execution time, and mask statistics. Adjust parameters (like *k* or thresholds) as needed for your images. The blending function desaturates the background using a blurred alpha mask, giving a smooth transition at edges. This structured code can be integrated into a Flask-based pipeline or a Jupyter notebook for experimentation.

Sources: Principles of each algorithm are standard in image processing [1] [2] [4] [6] [7]. The above references provide mathematical and intuitive explanations of these methods.

---

[1] Otsu's method - Wikipedia
https://en.wikipedia.org/wiki/Otsu%27s_method

[2] [3] Color-Based Segmentation Using K-Means Clustering - MATLAB & Simulink Example
https://www.mathworks.com/help/images/color-based-segmentation-using-k-means-clustering.html

[4] [5] Watershed segmentation — skimage 0.25.2 documentation
https://scikit-image.org/docs/0.25.x/auto_examples/segmentation/plot_watershed.html

[6] Canny edge detector - Wikipedia
https://en.wikipedia.org/wiki/Canny_edge_detector

[7] OpenCV Morphological Operations - PyImageSearch
https://pyimagesearch.com/2021/04/28/opencv-morphological-operations/