Documentation of a SIFT Detector Implementation

April 9, 2025

Contents

1 Introduction

The Scale-Invariant Feature Transform (SIFT) is a robust technique widely used in computer vision for detecting and describing local features within images. Its main strength lies in its invariance to scale, rotation, and illumination changes. SIFT achieves this by exploring the scale-space of an image through a Gaussian pyramid and finding local extrema in the Difference-of-Gaussian (DoG) representation. These detected keypoints are then assigned orientations, and distinctive descriptors are generated to allow reliable matching across images.

Feature detection using SIFT is vital in many applications such as object recognition, image stitching, and 3D reconstruction. The multi-step nature of the SIFT algorithm—including Gaussian blurring, downsampling (quantization of scale-space), DoG computation, and keypoint detection—ensures that the features are stable and repeatable.

2 SIFT Detector Architecture and Theory

The architecture of the SIFT detector can be summarized in the following steps:

- **Scale-space Construction:** A Gaussian pyramid is built by successively blurring and downsampling the input image. The scale quantization is achieved by choosing a fixed number of scales per octave and a multiplicative factor $k = 2^{1/(n-3)}$ (where n is the number of scale levels).
- **Difference-of-Gaussians (DoG):** The DoG images are obtained by subtracting adjacent blurred images in each octave. This approximates the Laplacian of Gaussian, making it efficient for detecting scale-space extrema.
- **Keypoint Detection:** Local extrema in the DoG pyramid are detected by comparing each pixel with its neighbors across scales and spatial locations. Only those with sufficient contrast (i.e., above a certain threshold) are considered valid keypoints.
- **Descriptor Computation (Not covered in this code):** Typically, orientations are assigned to each keypoint based on local gradients and a descriptor vector is generated from the surrounding region. In our implementation, the focus is on building the pyramid and detecting keypoints.

Quantization is applied in both the spatial and scale domains: pixels represent discrete spatial locations, while scale levels are discretized through the choice of the number of scales per octave and a fixed multiplicative factor.

3 Code Breakdown and Detailed Explanation

The implementation is structured into several functions that gradually build the SIFT detector, from constructing the Gaussian pyramid to detecting keypoints, and finally performing matching using OpenCV's built-in SIFT. Below is a detailed explanation of the key components.

3.1 Gaussian Pyramid Construction: `build_guassian_pyramid`

Purpose: This function builds a Gaussian pyramid by applying Gaussian blur with varying sigma values and then downsampling the image after each octave.

Key Points:

- The multiplicative factor $k = 2^{1/(num\_scales-3)}$ quantizes the scale space.
- Each octave contains a list of blurred images (each corresponding to a different scale).

- The image is downsampled (by a factor of 2) after processing each octave.

Listing 1: Function: build$_{guassian_pyramid}$

```
def build_guassian_pyramid(image, num_octaves=4, num_scales=5,
    sigma=1.6):
  """
  Build a Gaussian Pyramid for the given image.

  Parameters:
    image        - Input grayscale image as a Numpy array.
    num_octaves - Number of octaves to generate.
    num_scales   - Number of scale levels per octave.
    sigma        - Base sigma value for Gaussian blurring.

  Returns:
    pyramid - A list of octaves, each containing a list of blurred
        images.
  """
  pyramid = []
  # k determines the multiplicative factor between scales
      (quantization in scale space)
  k = 2**(1.0/(num_scales - 3))
  current_image = image.copy()
  for octave in range(num_octaves):
    scales = []
    for s in range(num_scales):
      sigma_total = sigma * (k**s)
      blurred = gaussian_filter(current_image, sigma_total)
      scales.append(blurred)
    pyramid.append(scales)
    # Downsample the image by a factor of 2 for the next octave.
    current_image = current_image[::2, ::2]
  return pyramid
```

3.2 DoG Pyramid Computation: `compute_dog_pyramid`

Purpose: This function calculates the Difference-of-Gaussians for each octave in the Gaussian pyramid by subtracting adjacent scale images.

Key Points:

- For each octave, the difference between successive blurred images is computed.
- These differences highlight edges and blob-like regions that are potential keypoints.

Listing 2: Function: compute$_{dog_pyramid}$

```
def compute_dog_pyramid(gaussian_pyramid):
  """
  Compute the Difference-of-Gaussian (DoG) pyramid.

  Parameters:
    gaussian_pyramid - A Gaussian pyramid as generated by
        build_guassian_pyramid.

  Returns:
    dog_pyramid - A list of octaves, each containing the difference
        between consecutive scales.
  """
  dog_pyramid = []
  for scales in gaussian_pyramid:
    dog_scales = []
    for i in range(len(scales)):
      dog = scales[i] - scales[i - 1]
      dog_scales.append(dog)
    dog_pyramid.append(dog_scales)
  return dog_pyramid
```

3.3 Keypoint Detection: `detect_keypoints`

Purpose: This function detects candidate keypoints by locating local extrema in the DoG pyramid that exceed a specified contrast threshold.

Key Points:

- The function iterates through each octave and through each scale level (ignoring the first and last scales for neighborhood comparison).
- For every pixel (avoiding borders), a 3x3 patch is extracted from the previous, current, and next DoG scales.
- A pixel is considered a keypoint if it is either a maximum or minimum relative to its neighbors and if its contrast is above the threshold.

Listing 3: Function: detect$_k$*eypoints*

```
def detect_keypoints(dog_pyramid, contrast_threshold=0.03):
  """
  Detect keypoints by identifying local extrema in the DoG pyramid.

  Parameters:
    dog_pyramid          - A list of octaves, each containing the DoG
        images.
    contrast_threshold   - Minimum absolute value to accept a candidate
        keypoint.

  Returns:
    keypoints - List of detected keypoints as tuples: (octave,
        scale_index, x, y, value).
  """
  keypoints = []
  # Loop over each octave and scale (ignoring first and last scale for
      neighborhood comparison)
  for octave_index, dog_scales in enumerate(dog_pyramid):
    for s in range(1, len(dog_scales) - 1):
      current = dog_scales[s]
      prev = dog_scales[s - 1]
      nxt = dog_scales[s + 1]
      H, W = current.shape[:2]
      # Avoid border pixels
      for y in range(1, H - 1):
        for x in range(1, W - 1):
          value = current[y, x]
          if np.mean(abs(value)) < contrast_threshold:
            continue
          % Extract a 3x3 patch from previous, current, and next scales
```

```
            patch_prev = prev[y - 1:y + 2, x - 1:x + 2]
            patch_curr = current[y - 1:y + 2, x - 1:x + 2]
            patch_next = nxt[y - 1:y + 2, x - 1:x + 2]
            patch = np.concatenate((patch_prev.flatten(),
                                    patch_curr.flatten(),
                                    patch_next.flatten()))
            % Check if the center pixel is a local extremum
            if value == np.max(patch) or value == np.min(patch):
              keypoints.append((octave_index, s, x, y, value))
  return keypoints
```

3.4 Manual SIFT Detector: `manual_sift_detector`

Purpose: This function integrates the previous steps to perform a manual SIFT detection on an image. It converts the image to a float format for precision, builds the Gaussian pyramid, computes the DoG pyramid, and finally detects keypoints.

Listing 4: Function: manual$_{sift_detector}$

```
def manual_sift_detector(image, num_octaves=4, num_scales=5,
    sigma=1.6, contrast_threshold=0.03):
  """
  Apply a basic manual SIFT Detector to an image.

  Parameters:
    image              - Input grayscale image as a Numpy array.
    num_octaves        - Number of octaves used in the pyramid.
    num_scales         - Number of scale levels per octave.
    sigma              - Initial sigma for Gaussian blur.
    contrast_threshold - Threshold to filter out low-contrast
        keypoints.

  Returns:
    keypoints - List of keypoints detected across octaves.
  """
  # Convert image to float32 to ensure precision
  img = image.astype(np.float32)
  % Build the Gaussian pyramid
  gaussian_pyramid = build_guassian_pyramid(image, num_octaves,
      num_scales, sigma)
  % Compute the DoG pyramid
  dog_pyramid = compute_dog_pyramid(gaussian_pyramid)
  % Detect keypoints in the DoG pyramid
  keypoints = detect_keypoints(dog_pyramid, contrast_threshold)
  return keypoints
```

3.5 SIFT Detection and Matching using OpenCV

In addition to the manual SIFT functions, the code utilizes OpenCV's built-in SIFT implementation to detect and match keypoints between two images. Key steps include:

- Loading images in grayscale.
- Initializing the SIFT detector with `cv2.SIFT_create()`.
- Detecting keypoints and computing descriptors.

- Matching descriptors using the BFMatcher with L2 norm.
- Visualizing the detected keypoints and the matches.

Listing 5: SIFT-based Keypoint Detection and Matching

```
def detect_and_match_keypoints(image1_path, image2_path):
    """
    Detect and match keypoints between two images using OpenCV's SIFT
        and BFMatcher.

    Parameters:
        image1_path (str): Path to the first image.
        image2_path (str): Path to the second image.

    Returns:
        None: Displays a side-by-side visualization of matched
            keypoints.
    """
    # Load images in grayscale
    image1 = cv2.imread(image1_path, cv2.IMREAD_GRAYSCALE)
    image2 = cv2.imread(image2_path, cv2.IMREAD_GRAYSCALE)

    % Initialize SIFT detector
    sift = cv2.SIFT_create()

    % Detect keypoints and compute descriptors
    keypoints1, descriptors1 = sift.detectAndCompute(image1, None)
    keypoints2, descriptors2 = sift.detectAndCompute(image2, None)

    % Match descriptors using BFMatcher with L2 norm
    bf = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)
    matches = bf.match(descriptors1, descriptors2)

    % Sort matches by distance (best matches first)
    matches = sorted(matches, key=lambda x: x.distance)

    % Draw and display the top 50 matches
    matched_image = cv2.drawMatches(
        image1, keypoints1,
        image2, keypoints2,
        matches[:50],
        None,
        flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS
    )

    % Display the result using Matplotlib
    plt.figure(figsize=(20, 10))
    plt.imshow(cv2.cvtColor(matched_image, cv2.COLOR_BGR2RGB))
    plt.title("Matched␣Keypoints")
    plt.axis("off")
    plt.show()
```

An additional function `detect_visualize_and_match` wraps the SIFT detection, visualization, and matching processes. It includes error handling, descriptor information printouts, and displays images with annotated keypoints.

3.6 Image Download and Main Execution Block

To ensure that images are available for matching, the code also includes a helper function to download images from a URL if they do not exist locally. The main execution block defines URLs and filenames, attempts downloads, and then performs keypoint detection and matching.

Listing 6: Main Execution Block and Image Download

```
def download_image(url, save_path):
    """Downloads an image from a URL if it does not exist."""
    if not os.path.exists(save_path):
        print(f"Downloading {os.path.basename(save_path)} from {url}...")
        try:
            urllib.request.urlretrieve(url, save_path)
            img = cv2.imread(save_path)
            if img is None:
                print(f"Error: Failed to download or save image from {url} to {save_path}.")
                return False
            print(f"Successfully downloaded {os.path.basename(save_path)}")
            return True
        except Exception as e:
            print(f"Error downloading {url}: {e}")
            return False
    else:
        print(f"Image {os.path.basename(save_path)} already exists.")
        return True

if __name__ == "__main__":
    % Define URLs and local filenames
    url1 = "https://raw.githubusercontent.com/opencv/opencv/master/samples/data/box.png"
    url2 = "https://raw.githubusercontent.com/opencv/opencv/master/samples/data/box_in_
    image1_filename = "box.png"
    image2_filename = "box_in_scene.png"

    % Attempt to download images
    download_ok1 = download_image(url1, image1_filename)
    download_ok2 = download_image(url2, image2_filename)

    % Run the SIFT detection and matching if images are available
    if os.path.exists(image1_filename) and os.path.exists(image2_filename):
        detect_and_match_keypoints(image1_filename, image2_filename)
    else:
        print("Error: One or both image files do not exist.")
```

4 Conclusion

This document has provided a comprehensive and step-by-step explanation of a SIFT detector implementation. The report detailed the architecture of SIFT—including the construction of a Gaussian pyramid, computation of the Difference-of-Gaussian pyramid, and keypoint detection through local extrema analysis—and explained how these components work together to yield robust feature detection and matching. In addition, the integration of OpenCV's SIFT and

BFMatcher was presented to demonstrate practical feature matching between images. The quantization of scale space and careful handling of image resolution in the pyramid construction ensure that the SIFT detector is both efficient and robust, making it a powerful tool in various computer vision tasks.