Image Resizing Using Seam Carving

Project Report

Submitted by

Punit Ranawat

Yuvraj Biren Parekh

Nidhish Saini

IE211: Programming and Computing Lab IIT Bombay

Aug 2025 - Nov 2025

Abstract

Traditional image resizing methods such as scaling or cropping often distort or remove important regions. Seam carving provides a content-aware approach that resizes images by iteratively removing low-importance pixel paths called seams. This project implements seam carving from scratch, showing how dynamic programming and efficient data structures can be applied to computer vision. The algorithm computes an energy map using image gradients, identifies and removes the seam of minimum cumulative energy, and repeats until the target dimensions are achieved—preserving important objects and scene structure. The complete implementation and accompanying code are available in the project repository at `https://github.com/mr-entr0py/seam_carving/tree/main`.

Contents

1. Core Seam Carving Implementation (Yuvraj Biren Parekh)

1.1 Seam Carving Core Implementation: `seam_carver.py`

This file implements the core logic for content-aware image resizing using seam carving. It defines a `SeamCarver` class that encapsulates image data and operations, together with a `main()` entry point that runs the algorithm on a sample image and saves and displays the result. The main tasks performed in this file are: computing an energy function over the image using Sobel filters, using dynamic programming to find a minimum-energy vertical seam, iteratively removing such seams to reduce the image width, extending the same idea to height reduction using rotation, and providing simple visualization and output functionality.

1.1.1 Imports and Backend Setup

```
import matplotlib.pyplot as plt
import cv2 as cv
import json
import numpy as np
import matplotlib
matplotlib.use("TkAgg")
```

The code uses:

- **cv2 (OpenCV)** for image reading, color conversion, and gradient (Sobel) computation.
- **NumPy** for numerical operations, dynamic programming arrays, and seam removal via array manipulation.
- **Matplotlib** for plotting the original and seam-carved images in a window, with the backend explicitly set to `TkAgg` so that `plt.show()` opens a GUI window.

- **json** for reading the required output dimensions from a JSON file.

1.1.2 The `SeamCarver` Class: Overview

```
class SeamCarver:
    def __init__(self, input_img, specifications) -> None:
        self.input_img = input_img
        self.specifications = specifications
        self.reqd_h = specifications["height"]
        self.reqd_w = specifications["width"]
        self.output_img = input_img.copy()
        self.img_h, self.img_w = input_img.shape[:2]
```

The constructor accepts an OpenCV image array of shape $(H, W, 3)$ in BGR format and a dictionary containing the desired output height and width. It stores the original image, makes a copy for carving, extracts the required dimensions, and records the original image height and width.

1.1.3 Validation of Requested Dimensions

```
def validate(self):
    print(f"Input Dimensions: {self.img_w} x {self.img_h}")
    print(f"Required Dimensions: {self.reqd_w} x {self.reqd_h}")
    return self.reqd_h <= self.img_h and self.reqd_w <= self.img_w
```

This method ensures that seam carving is applied only for reduction. The constraints are

$$\text{reqd}_w \leq \text{img}_w, \quad \text{reqd}_h \leq \text{img}_h.$$

The method returns a Boolean indicating whether the operation is valid.

1.1.4 Vertical Seam Carving: `seam_carving_vertical`

```
def seam_carving_vertical(self, img, target_width):
    img_gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY).astype(np.float64)
    num_seams = img.shape[1] - target_width
    carved_img = img.copy()
```

The number of seams to remove is computed as

$$\text{num\_seams} = W - W_t.$$

1.1.5 Energy Computation

```
gx = cv.Sobel(img_gray, cv.CV_64F, 1, 0, ksize=3)
gy = cv.Sobel(img_gray, cv.CV_64F, 0, 1, ksize=3)
energy = (np.abs(gx) + np.abs(gy)).astype(np.float64)
```

The energy function is defined as

$$E(i,j) = |G_x(i,j)| + |G_y(i,j)|.$$

1.1.6 Dynamic Programming Formulation

```
dp = energy.copy()
for row in range(1, H):
    for col in range(W):
        if col == 0:
            dp[row, col] += min(dp[row-1, col], dp[row-1, col+1])
        elif col == W-1:
            dp[row, col] += min(dp[row-1, col-1], dp[row-1, col])
        else:
            dp[row, col] += min(dp[row-1, col-1], dp[row-1, col], dp[row-1,
                col+1])
```

The recurrence is

$$M(i,j) = E(i,j) + \min\{M(i-1,j-1), M(i-1,j), M(i-1,j+1)\}.$$

Boundary cases:

$$M(i,0) = E(i,0) + \min\{M(i-1,0), M(i-1,1)\},$$

$$M(i,W-1) = E(i,W-1) + \min\{M(i-1,W-2), M(i-1,W-1)\}.$$

1.1.7 Seam Backtracking

```
seam = np.zeros(H, dtype=np.int32)
seam[-1] = np.argmin(dp[-1])
for row in range(H-2, -1, -1):
    prev = seam[row+1]
    if prev == 0:
        choices = [dp[row, prev], dp[row, prev+1]]
```

```
        seam[row] = prev + np.argmin(choices)
    elif prev == W-1:
        choices = [dp[row, prev-1], dp[row, prev]]
        seam[row] = prev - 1 + np.argmin(choices)
    else:
        choices = [dp[row, prev-1], dp[row, prev], dp[row, prev+1]]
        seam[row] = prev - 1 + np.argmin(choices)
```

The bottom seam index is

$$j^* = \arg\min_j M(H-1, j).$$

The backtracking relation is

$$j_i = \arg\min_{k \in N(j_{i+1})} M(i, k).$$

1.1.8 Seam Removal

```
img_gray = np.array([np.delete(img_gray[row], seam[row]) for row in range(H)])
carved_img = np.array([np.delete(carved_img[row], seam[row], axis=0) for row
    in range(H)])
```

This removes exactly one pixel from each row, reducing the width by one pixel.

The total time complexity is

$$O(kHW),$$

where k is the number of seams removed.

1.1.9 Full 2D Seam Carving

```
def seam_carving(self):
    self.output_img = self.seam_carving_vertical(self.output_img, self.reqd_w)
    rotated = np.rot90(self.output_img, k=-1)
    carved_rotated = self.seam_carving_vertical(rotated, self.reqd_h)
    self.output_img = np.rot90(carved_rotated, k=1)
```

Dimension transformation:

$$(H_0, W_0) \to (H_0, W_r) \to (W_r, H_0) \to (W_r, H_r) \to (H_r, W_r).$$

1.1.10 Visualization

```
def show_images(self):
    plt.figure(figsize=(12, 6))
    plt.subplot(1, 2, 1)
    plt.imshow(cv.cvtColor(self.input_img, cv.COLOR_BGR2RGB))
    plt.title("Original")
    plt.axis("off")

    plt.subplot(1, 2, 2)
    plt.imshow(cv.cvtColor(self.output_img, cv.COLOR_BGR2RGB))
    plt.title("Seam Carved")
    plt.axis("off")
    plt.show()
```

1.1.11 Saving the Output

```
def save_output(self, path="output.png"):
    cv.imwrite(path, self.output_img)
```

1.1.12 Program Entry Point

```
def main():
    input_img = cv.imread("data/human.png")
    with open("data/specifications.json") as f:
        specifications = json.load(f)

    carver = SeamCarver(input_img, specifications)
    if not carver.validate():
        print("Required dimensions are larger than image dimensions.")
        return

    carver.seam_carving()
    carver.save_output("output_carved.png")
    carver.show_images()
```

1.1.13 Summary

The file `seam_carver.py` implements the complete seam carving pipeline: Sobel-gradient energy computation, dynamic programming for optimal seam identification, backtracking for seam extraction, iterative seam removal for width reduction, geometric rotation for height reduction, and visualization and storage of results. It faithfully translates the mathematical formulation of seam carving into an executable and modular software implementation.

2. Image Quality Metrics and Extended Evaluation (Nidhish Saini)

2.1 Image Quality Metrics and Extended Evaluation

This section describes the two image quality metrics used to assess the effect of seam carving, namely Peak Signal-to-Noise Ratio (PSNR) and Structural Similarity Index (SSIM). It then explains the extended implementation that computes these metrics for the original and carved images and visualizes their values. Conceptually, `seam_carver.py` performs the content-aware resizing, while this evaluation layer quantifies how close the carved output is to the original image.

2.1.1 Role of the Evaluation Layer

The project consists of three main components:

- **`seam_carver.py`**: Implements the actual seam carving algorithm (energy computation, dynamic programming, seam backtracking, seam removal, and rotation-based height reduction).
- **`app.py`** (Streamlit): Provides an interactive front-end for uploading images, specifying target dimensions, running seam carving, and displaying outputs and basic quality metrics.
- **Evaluation code** (this section): Defines PSNR and SSIM formally, implements them in Python, runs seam carving, and produces numerical and visual summaries (bar plots) comparing the original and carved images.

The evaluation code therefore acts as a quantitative validation layer on top of the algorithmic core.

2.1.2 Peak Signal-to-Noise Ratio (PSNR)

PSNR is a classical pixel-wise quality metric used to quantify the fidelity of a processed image with respect to a reference image. It is derived from the mean squared error (MSE) between the two images. A lower MSE implies higher similarity and thus a higher PSNR value. PSNR is expressed in decibels (dB).

Mean Squared Error (MSE)

Let I_{orig} and I_{proc} denote the original and processed images, respectively, both of size $m \times n$ for a single channel. The mean squared error is

$$\text{MSE} = \frac{1}{mn} \sum_{i=1}^{m} \sum_{j=1}^{n} \Big(I_{\text{orig}}(i,j) - I_{\text{proc}}(i,j) \Big)^2 .$$

For color images of size $H \times W \times C$, it is common to treat all pixels and channels as a single vector of length $N = H \cdot W \cdot C$:

$$\text{MSE} = \frac{1}{N} \sum_{p=1}^{N} \Big(I_p - \hat{I}_p \Big)^2 ,$$

where I_p and $\hat{I}_p$ are corresponding pixel values from the original and processed images.

PSNR Definition

Once MSE is known, PSNR is defined as

$$\text{PSNR} = 10 \log_{10} \left(\frac{\text{MAX}^2}{\text{MSE}} \right) ,$$

where MAX is the maximum possible pixel value. For 8-bit images, MAX = 255. If MSE = 0, the images are identical and PSNR is taken to be $+\infty$.

In practice, for typical seam carving experiments, PSNR values in the range of 30–50 dB correspond to acceptable to very high visual quality.

PSNR Implementation

The following function implements PSNR for two NumPy arrays of the same shape:

```
1 import numpy as np
2 import math
3
```

```python
def calculate_psnr(img1, img2):
    """
    Calculate Peak Signal-to-Noise Ratio between two images.

    Mathematical Formulation:
    MSE = (1/mn) *  [I1(i,j) - I2(i,j)]
    PSNR = 10 * log10(MAX / MSE)

    Parameters:
    img1, img2: numpy arrays of the same shape

    Returns:
    PSNR value in decibels (higher is better)
    """
    # Convert to float for precise calculations
    img1 = img1.astype(np.float64)
    img2 = img2.astype(np.float64)

    # Calculate Mean Squared Error
    # MSE = (1/N) * (x - x)
    mse = np.mean((img1 - img2) ** 2)

    # If images are identical, return infinity
    if mse == 0:
        return float('inf')

    # PSNR = 10 * log10(MAX / MSE)
    # Using 20*log10(MAX/sqrt(MSE)) is equivalent
    max_pixel = 255.0
    psnr = 20 * math.log10(max_pixel / math.sqrt(mse))

    return psnr
```

The function:

- Converts both images to `float64` for numerical stability.
- Computes MSE via `np.mean((img1 - img2)2)`.
- Returns $+\infty$ if the MSE is zero.

- Computes PSNR using the algebraically equivalent expression

$$20 \log_{10} \left(\frac{\text{MAX}}{\sqrt{\text{MSE}}} \right) = 10 \log_{10} \left(\frac{\text{MAX}^2}{\text{MSE}} \right).$$

2.1.3 Structural Similarity Index (SSIM)

While PSNR measures pixel-wise errors, SSIM is designed to better correlate with human visual perception by comparing structures rather than absolute differences. SSIM ranges typically between 0 and 1, with:

- SSIM = 1: perfect structural similarity,
- values close to 0: very poor similarity.

Motivation and Decomposition

SSIM decomposes similarity into three components for local image patches x and y:

1. **Luminance similarity** $l(x, y)$ (mean intensity),
2. **Contrast similarity** $c(x, y)$ (local variance/standard deviation),
3. **Structure similarity** $s(x, y)$ (normalized covariance).

The luminance term is

$$l(x, y) = \frac{2\mu_x \mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1},$$

where μ_x, μ_y are local means and C_1 is a stabilizing constant. The contrast term is

$$c(x, y) = \frac{2\sigma_x \sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2},$$

where σ_x, σ_y are local standard deviations and C_2 is another stabilizing constant. The structure term is

$$s(x, y) = \frac{\sigma_{xy} + C_3}{\sigma_x \sigma_y + C_3},$$

where σ_{xy} is the local covariance and C_3 is typically set to $C_2/2$.

Combined SSIM Formula

The general SSIM index over a patch is

$$\text{SSIM}(x, y) = [l(x, y)]^\alpha [c(x, y)]^\beta [s(x, y)]^\gamma.$$

In the widely used simplified form, $\alpha = \beta = \gamma = 1$, giving

$$\text{SSIM}(x, y) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)}.$$

The constants are set as

$$C_1 = (K_1 L)^2, \quad C_2 = (K_2 L)^2, \quad C_3 = \frac{C_2}{2},$$

where L is the dynamic range of pixel values (e.g. $L = 255$) and $K_1 = 0.01$, $K_2 = 0.03$ are small constants.

SSIM Implementation Using `skimage`

The project uses the implementation from `skimage.metrics.structural_similarity`:

```python
import numpy as np
from skimage.metrics import structural_similarity as ssim
import matplotlib.pyplot as plt

def calculate_ssim(img1, img2):
    """
    Calculate Structural Similarity Index (SSIM) between two images.

    Parameters:
    img1, img2: numpy arrays of the same shape and type.

    Returns:
    SSIM value (float) where 1.0 means perfect similarity.
    """
    # Ensure images are float for SSIM calculation if they are not already
    img1_float = img1.astype(float)
    img2_float = img2.astype(float)

    # Calculate SSIM. data_range is important for correct normalization.
    # For 8-bit images, data_range is 255.
    # We also need to specify 'channel_axis' for color images.
    score = ssim(img1_float, img2_float,
                 data_range=img2_float.max() - img2_float.min(),
                 channel_axis=-1)
    return score
```

Here:

- Images are converted to `float` arrays.
- `data_range` is set from the minimum and maximum of the image, consistent with the dynamic range L in the SSIM formula.
- `channel_axis=-1` instructs the function to treat the last dimension as color channels so that SSIM is computed appropriately for RGB images.

2.1.4 Extended Seam Carver and Evaluation Pipeline

To evaluate seam carving more systematically, an extended `SeamCarver` class is used in a notebook or separate evaluation script. It mirrors the core implementation in `seam_carver.py` but adds explicit horizontal carving and integrates with PSNR/SSIM evaluation.

Extended `SeamCarver` Class

```
import matplotlib.pyplot as plt
import cv2 as cv
import json
import numpy as np
import matplotlib

class SeamCarver:
    def __init__(self, input_img, specifications) -> None:
        self.input_img = input_img
        self.specifications = specifications
        self.reqd_h = specifications["height"]
        self.reqd_w = specifications["width"]
        self.output_img = input_img.copy()
        self.img_h, self.img_w = input_img.shape[:2]

    def validate(self):
        print(f"Input Dimensions: {self.img_w} x {self.img_h}")
        print(f"Required Dimensions: {self.reqd_w} x {self.reqd_h}")
        return self.reqd_h <= self.img_h and self.reqd_w <= self.img_w

    def seam_carving_vertical(self, img, target_width):
        """Carve vertical seams from a given image array."""
        carved_img = img.copy() # Operate on a copy for each carving pass
        img_gray = cv.cvtColor(carved_img,
            cv.COLOR_BGR2GRAY).astype(np.float64)
```

```
25
26      num_seams = img.shape[1] - target_width
27
28      for i in range(num_seams):
29          print(f"Removing vertical seam {i+1}/{num_seams}")
30          H, W = img_gray.shape
31
32          # Calculate energy map using Sobel filters
33          gx = cv.Sobel(img_gray, cv.CV_64F, 1, 0, ksize=3)
34          gy = cv.Sobel(img_gray, cv.CV_64F, 0, 1, ksize=3)
35          energy = (np.abs(gx) + np.abs(gy)).astype(np.float64)
36
37          # DP table
38          dp = energy.copy()
39
40          # Fill DP table
41          for row in range(1, H):
42              for col in range(W):
43                  if col == 0:
44                      dp[row, col] += min(dp[row-1, col],
45                                          dp[row-1, col+1])
46                  elif col == W-1:
47                      dp[row, col] += min(dp[row-1, col-1],
48                                          dp[row-1, col])
49                  else:
50                      dp[row, col] += min(dp[row-1, col-1],
51                                          dp[row-1, col],
52                                          dp[row-1, col+1])
53
54          # Backtrack to find the seam
55          seam = np.zeros(H, dtype=np.int32)
56          seam[-1] = np.argmin(dp[-1]) # Start from the minimum in the last
                row
57
58          for row in range(H-2, -1, -1):
59              prev = seam[row+1]
60              if prev == 0:
61                  choices = [dp[row, prev], dp[row, prev+1]]
62                  seam[row] = prev + np.argmin(choices)
63              elif prev == W-1:
64                  choices = [dp[row, prev-1], dp[row, prev]]
```

```
                seam[row] = prev - 1 + np.argmin(choices)
            else:
                choices = [dp[row, prev-1], dp[row, prev], dp[row, prev+1]]
                seam[row] = prev - 1 + np.argmin(choices)

        # Remove seam
        img_gray = np.array(
            [np.delete(img_gray[row], seam[row]) for row in range(H)]
        )
        carved_img = np.array(
            [np.delete(carved_img[row], seam[row], axis=0) for row in
               range(H)]
        )

    return carved_img

def seam_carving_horizontal(self, img, target_height):
    """Carve horizontal seams from a given image array.
    Implemented by rotating, applying vertical carving, and rotating
       back."""
    # Rotate image 90 degrees clockwise
    rotated_img = cv.rotate(img, cv.ROTATE_90_CLOCKWISE)

    # Apply vertical seam carving on the rotated image
    carved_rotated = self.seam_carving_vertical(rotated_img, target_height)

    # Rotate back 90 degrees counter-clockwise
    final_img = cv.rotate(carved_rotated, cv.ROTATE_90_COUNTERCLOCKWISE)
    return final_img

def seam_carving(self):
    current_h, current_w = self.output_img.shape[:2]

    if self.reqd_w < current_w: # If we need to reduce width
        self.output_img = self.seam_carving_vertical(self.output_img,
                                                     self.reqd_w)

    if self.reqd_h < current_h: # If we need to reduce height
        # Reduce height via horizontal seam carving
        self.output_img = self.seam_carving_horizontal(self.output_img,
                                                       self.reqd_h)
```

```
104
105     def show_images(self):
106         plt.figure(figsize=(12, 6))
107         plt.subplot(1, 2, 1)
108         plt.imshow(cv.cvtColor(self.input_img, cv.COLOR_BGR2RGB))
109         plt.title("Original")
110         plt.axis("off")
111
112         plt.subplot(1, 2, 2)
113         plt.imshow(cv.cvtColor(self.output_img, cv.COLOR_BGR2RGB))
114         plt.title(f"Seam Carved
                ({self.output_img.shape[1]}x{self.output_img.shape[0]})")
115         plt.axis("off")
116         plt.show()
117
118     def save_output(self, path="output_carved.png"):
119         cv.imwrite(path, self.output_img)
```

This class is algorithmically equivalent to the core version in `seam_carver.py`, but it is embedded in the evaluation context and used together with PSNR and SSIM.

Main Evaluation Routine

The evaluation script then runs seam carving and computes PSNR and SSIM:

```
1  def main():
2      # Load from data/ folder (same as test_app.py and seam_carver.py)
3      try:
4          input_img = cv.imread("data/human.png")
5          with open("data/specifications.json") as f:
6              specifications = json.load(f)
7          if input_img is None:
8              raise FileNotFoundError("Image not found")
9          print("Loaded image from data/human.png")
10     except FileNotFoundError:
11         print("Error: data/human.png or data/specifications.json not found.")
12         print("Please run the Streamlit app (test_app.py) first to upload an
               image.")
13         return
14
15     carver = SeamCarver(input_img, specifications)
16     if not carver.validate():
```

```
        print("Required dimensions are larger than image dimensions. "
              "Please adjust specifications.")
        return

    # Perform seam carving (reduces width first, then height)
    carver.seam_carving()

    # Save FIRST, then show (in case visualization crashes)
    carver.save_output("output_carved.png")
    print("Saved output_carved.png")
    carver.show_images()


if __name__ == '__main__':
    main()
```

2.1.5 Computing PSNR and SSIM for Original vs. Carved Image

After seam carving, the script loads both the original and carved images from disk:

```
import cv2 as cv
import numpy as np
import json

# --- Load the original image from data folder ---
original_image_path = "data/human.png"
original_image = None
specifications = None

try:
    original_image = cv.imread(original_image_path)
    with open("data/specifications.json") as f:
        specifications = json.load(f)
    if original_image is None:
        raise FileNotFoundError("Image not found")
    print(f"Loaded original image from {original_image_path}")
except FileNotFoundError:
    print(f"Error: {original_image_path} or data/specifications.json not
        found.")
    print("Please run the Streamlit app (test_app.py) first to upload an
        image.")
```

```
20
21 # --- Load the carved image ---
22 carved_image_path = "output_carved.png"
23 carved_image = cv.imread(carved_image_path)
24
25 if original_image is None or carved_image is None:
26     print("Error: Could not load original or carved image. "
27           "Please ensure files exist.")
28 else:
29     # Convert both to RGB
30     original_image_rgb = cv.cvtColor(original_image, cv.COLOR_BGR2RGB)
31     carved_image_rgb = cv.cvtColor(carved_image, cv.COLOR_BGR2RGB)
32
33     # Resize original to match carved dimensions
34     carved_h, carved_w = carved_image_rgb.shape[:2]
35     resized_original_rgb = cv.resize(original_image_rgb,
36                                      (carved_w, carved_h),
37                                      interpolation=cv.INTER_AREA)
38
39     # Calculate PSNR and SSIM
40     psnr_value = calculate_psnr(resized_original_rgb, carved_image_rgb)
41     ssim_value = calculate_ssim(resized_original_rgb, carved_image_rgb)
42
43     print(f"\nPSNR between resized original and carved image: {psnr_value:.2f}
           dB")
44     print(f"SSIM between resized original and carved image: {ssim_value:.4f}")
```

Here:

- Both images are converted from BGR to RGB for consistency.
- The original image is resized to the carved image's resolution. This is crucial because PSNR and SSIM require both inputs to have identical dimensions.
- `calculate_psnr` and `calculate_ssim` are applied to the resized original and the carved image, providing quantitative measures of fidelity and structural similarity.

The script also prints image dimensions and metric values:

```
1 original_height, original_width, _ = original_image.shape
2 target_height = specifications["height"]
3 target_width = specifications["width"]
4
```

```
print(f"Original Image Dimensions: {original_width}x{original_height}")
print(f"Target Image Dimensions: {target_width}x{target_height}")
print(f"PSNR: {psnr_value:.2f}")
print(f"SSIM: {ssim_value:.4f}")
```

2.1.6 Visualization of PSNR and SSIM

Finally, the notebook generates bar charts for PSNR and SSIM with annotations.

PSNR bar chart.

```
import matplotlib.pyplot as plt

# Prepare data for plotting
metric_name = 'PSNR'
metric_value = psnr_value

# Create the bar chart for PSNR
fig, ax = plt.subplots(figsize=(6, 5))
bar = ax.bar(metric_name, metric_value, color='skyblue')

# Set title and labels
ax.set_title('Image Quality Metric: PSNR After Seam Carving')
ax.set_ylabel('PSNR Value (dB)')

# Add annotation for the PSNR value on top of the bar
yval = bar[0].get_height()
ax.text(bar[0].get_x() + bar[0].get_width()/2,
        yval + 0.05, f'{yval:.2f}',
        ha='center', va='bottom', fontsize=10)

# Add annotations for image dimensions
annotation_text = (f"Original: {original_width}x{original_height}\n"
                   f"Target: {target_width}x{target_height}")
ax.text(0.5, 0.95, annotation_text,
        transform=ax.transAxes,
        fontsize=10, ha='center', va='top',
        bbox=dict(boxstyle="round,pad=0.3",
                  fc="yellow", alpha=0.5))

# Set y-axis limits to better visualize the value
ax.set_ylim(0, max(metric_value * 1.2, 20))
```

```

plt.tight_layout()
plt.show()
print("Generated PSNR bar chart with annotations.")
```

SSIM bar chart.

```
import matplotlib.pyplot as plt

# Prepare data for plotting
metric_name = 'SSIM'
metric_value = ssim_value

# Create the bar chart for SSIM
fig, ax = plt.subplots(figsize=(6, 5))
bar = ax.bar(metric_name, metric_value, color='lightcoral')

# Set title and labels
ax.set_title('Image Quality Metric: SSIM After Seam Carving')
ax.set_ylabel('SSIM Value')

# Add annotation for the SSIM value on top of the bar
yval = bar[0].get_height()
ax.text(bar[0].get_x() + bar[0].get_width()/2,
        yval + 0.02, f'{yval:.4f}',
        ha='center', va='bottom', fontsize=10)

# Add annotations for image dimensions
annotation_text = (f"Original: {original_width}x{original_height}\n"
                   f"Target: {target_width}x{target_height}")
ax.text(0.5, 0.95, annotation_text,
        transform=ax.transAxes,
        fontsize=10, ha='center', va='top',
        bbox=dict(boxstyle="round,pad=0.3",
                  fc="yellow", alpha=0.5))

# Set y-axis limits to highlight the SSIM score range (0 to 1)
ax.set_ylim(0, 1)

plt.tight_layout()
plt.show()
print("Generated SSIM bar chart with annotations.")
```

These charts provide an immediate visual summary of the numeric metrics:

- The PSNR plot shows how high the decibel value is, together with a reminder of the original and target sizes.
- The SSIM plot uses a fixed range $[0, 1]$ on the vertical axis, visually emphasizing how close the structural similarity is to the ideal value of 1.

Together, PSNR and SSIM give complementary evidence about seam carving quality: PSNR measures pixel-wise fidelity, whereas SSIM measures perceptual and structural similarity. This evaluation layer thus connects the algorithmic core to quantitative, human-interpretable quality assessment. “‘

3. Streamlit Interface and Quality Evaluation (Punit Ranawat)

3.1 Streamlit Interface and Quality Evaluation (`app.py`)

This file implements a Streamlit-based user interface for the seam carving system and adds quantitative evaluation of the carved output using Peak Signal-to-Noise Ratio (PSNR) and Structural Similarity Index (SSIM). It is responsible for three main tasks: (i) accepting an input image and desired target size from the user and saving them in a format that the core seam carving script (`seam_carver.py`) expects; (ii) displaying the seam-carved output once it is generated; and (iii) computing and presenting objective quality metrics comparing the output against the original image. In contrast to `seam_carver.py`, which focuses entirely on the algorithm, this file focuses on user interaction, data flow to and from disk, and image quality assessment.

3.1.1 Imports and Path Configuration

```
import json
from pathlib import Path

import streamlit as st
from PIL import Image
import numpy as np
import cv2 as cv

# Use current working directory
BASE_DIR = Path.cwd()
DATA_DIR = BASE_DIR / "data"
```

The file imports `streamlit` for the web interface, `PIL.Image` for image loading and saving, `numpy` for numerical operations, and `cv2` for image processing functions such as color space conversion, blurring, and resizing. The `Path` class from `pathlib` is used to

build filesystem paths in a portable manner. `BASE_DIR` is set to the current working directory, and `DATA_DIR` is defined as the `data` subdirectory under the current directory. This aligns with the expectations of `seam_carver.py`, which reads `"data/human.png"` and `"data/specifications.json"`.

3.1.2 Paths and Naming Consistency with `seam_carver.py`

To ensure that this interface cooperates with the core seam carving script, the file defines specific paths that match what `seam_carver.py` expects:

```
# Paths that seam_carver.py expects
INPUT_IMAGE_PATH = DATA_DIR / "human.png"
PARAMS_PATH = DATA_DIR / "specifications.json"
OUTPUT_IMAGE_PATH = BASE_DIR / "output_carved.png"
```

The image uploaded via the UI is saved as `data/human.png`, and the target width and height are saved in `data/specifications.json`. These are exactly the locations used by the `main()` function in `seam_carver.py`. The output image generated by the seam carver is expected at `output_carved.png` in the base directory. This naming convention allows the core algorithm and the UI to communicate indirectly via the filesystem.

3.1.3 Streamlit Application Layout: Input Stage

The UI begins with a title and a file uploader:

```
st.title("Seam Carving Interface")

uploaded = st.file_uploader("Upload an image", type=["png", "jpg", "jpeg"])
```

When a user uploads an image, the code reads and displays it, and offers interactive controls for specifying target dimensions:

```
if uploaded is not None:
    img = Image.open(uploaded).convert("RGB")
    st.image(img, caption="Uploaded image", use_column_width=True)

    w, h = img.size
    st.write(f"**Original size:** {w}  {h}")

    target_w = st.number_input("Target width",
                               min_value=1, max_value=w,
                               value=w, step=1)
```

```
    target_h = st.number_input("Target height",
                               min_value=1, max_value=h,
                               value=h, step=1)
```

The uploaded image is opened via PIL and converted to RGB. Its original width and height are shown. The `st.number_input` widgets allow the user to choose `target_w` and `target_h`, bounded between 1 and the original dimensions, with default values equal to the original sizes. This enforces that the requested size does not exceed the original, consistent with the reduction-only behavior of seam carving.

Upon clicking the "Save image and parameters" button, this section writes both the image and the parameter file into the `data` directory:

```
    if st.button("Save image and parameters"):
        # Ensure data directory exists
        DATA_DIR.mkdir(exist_ok=True)

        # Save the input image as data/human.png
        img.save(INPUT_IMAGE_PATH)

        # Save parameters as JSON with keys seam_carver.py expects
        params = {
            "width": int(target_w),
            "height": int(target_h),
        }
        with open(PARAMS_PATH, "w", encoding="utf-8") as f:
            json.dump(params, f, indent=2)

        st.success(f"Saved `{INPUT_IMAGE_PATH.name}` and "
                   f"`{PARAMS_PATH.name}` in `{DATA_DIR}`")
        st.code("python seam_carver.py", language="bash")
        st.info("Run the command above to perform seam carving, "
                "then click 'Refresh output'")
```

The `data` directory is created if it does not exist. The PIL image is saved as `human.png` in this directory, and a JSON file is created with keys `"width"` and `"height"` containing the user-selected target dimensions. This ensures that when the user runs `python seam_carver.py` in a terminal, the seam carving script will read precisely the image and specifications just saved via the UI. A horizontal rule visually separates the input section from the output section:

```
st.write("---")
```

3.1.4 Output Stage: Loading and Displaying the Carved Image

The second part of the UI is dedicated to displaying the output image and computing quality metrics:

```
st.subheader("Output image")

st.write(f"Looking for: ‘{OUTPUT_IMAGE_PATH.name}‘ in ‘{BASE_DIR}‘")

if st.button("Refresh output"):
    st.rerun()
```

The application informs the user which file and directory it is monitoring. The “Refresh output” button triggers a rerun of the Streamlit script, causing it to re-check whether `output_carved.png` now exists. If the output file is present, it is loaded and displayed:

```
if OUTPUT_IMAGE_PATH.exists():
    out_img = Image.open(OUTPUT_IMAGE_PATH).convert("RGB")
    ow, oh = out_img.size
    st.image(out_img,
             caption=f"Output image ({ow}  {oh})",
             use_column_width=True)
```

The carved image is opened with PIL, converted to RGB, and shown with its current dimensions.

3.1.5 Quality Metrics Computation and Display

If the original input image still exists at `data/human.png`, the code proceeds to calculate PSNR and SSIM:

```
    # Calculate and display quality metrics if original image exists
    if INPUT_IMAGE_PATH.exists():
        original_pil = Image.open(INPUT_IMAGE_PATH).convert("RGB")
        original_np = np.array(original_pil)
        carved_np = np.array(out_img)

        # Resize original to match carved dimensions for comparison
        carved_h, carved_w = carved_np.shape[:2]
        resized_original = cv.resize(original_np,
                                     (carved_w, carved_h),
                                     interpolation=cv.INTER_AREA)
```

Here the original and carved images are both converted to NumPy arrays. Since seam carving changes the image dimensions, the original is resized using `cv.resize` to match the carved image dimensions (carved_w, carved_h) before computing metrics. This ensures that PSNR and SSIM are computed between two arrays of identical shape. The metrics are then computed:

```
        # Calculate metrics
        psnr_value = calculate_psnr(resized_original, carved_np)
        ssim_value = calculate_ssim(resized_original, carved_np)

        st.write("---")
        st.subheader("Quality Metrics")
        col1, col2 = st.columns(2)
        with col1:
            st.metric("PSNR",
                      f"{psnr_value:.2f} dB" if psnr_value else "N/A")
        with col2:
            st.metric("SSIM",
                      f"{ssim_value:.4f}" if ssim_value else "N/A")

        st.caption("PSNR: Higher is better (typical: 20-40 dB). "
                   "SSIM: Closer to 1 is better.")
```

The numerical values returned by `calculate_psnr` and `calculate_ssim` are displayed in two columns with `st.metric`. PSNR is presented in decibels and formatted to two decimal places; SSIM is presented as a unitless value typically between 0 and 1, formatted to four decimal places. A brief caption explains the interpretation: higher PSNR (typically 20–40 dB) indicates less distortion, and SSIM values closer to 1 indicate higher structural similarity between the original and processed images.

If `output_carved.png` does not yet exist, the interface informs the user:

```
else:
    st.info(f"No ‘{OUTPUT_IMAGE_PATH.name}‘ found yet. "
            "Run ‘python seam_carver.py‘ and click ’Refresh output’.")
```

This completes the interaction loop: upload and specify target size, save data, run the seam carver externally, then return to the UI to inspect and evaluate the result.