Heeger-Bergen Steerable Pyramid-Based Texture Synthesis Problem Set

Fall 2023 CS 283 Advanced Computer Vision Final Project

Sree Harsha Tanneru
Harvard University
sreeharshatanneru@g.harvard.edu

Annabel Yim
Harvard University
annabelyim@g.harvard.edu

Abstract

Texture synthesis is a notable application of image pyramids, which are multi-resolution representations used for analyzing images across various spatial scales. These pyramids are utilized to generate coherent textures corresponding to a source texture. Among the different types of image pyramids, steerable pyramids offer several advantages. They provide a rich representation that includes information about image scale and orientation. Additionally, steerable pyramids are multi-scale, oriented, translation-invariant, non-aliased, over-complete, and self-invertible [6].

During our class lectures, we explored the properties of steerable pyramids and their role in texture synthesis. To further enhance our understanding of the practical applications of steerable pyramids, we propose an implementation-focused problem set. This problem set is designed to delve deeper into the subject, involving exploration of the Heeger-Bergen algorithm, used for synthesizing grayscale textures. It extends the algorithm to adapt the synthesis of grayscale textures to different image sizes from the source texture, explores color textures, and addresses the handling of edges in textures.

Our work is in the following GitHub Repository.

1. Introduction

The Heeger-Bergen algorithm assumes that the first-order statistics of steerable pyramids can capture all spatial information characterizing a texture image. The process begins with a white noise image and alternates between histogram matching in the image and steerable pyramid domains. The aim is to match the output histogram with that of the input texture [2].

Although an ANSI C implementation of the Heeger-Bergen algorithm exists, there has been no publicly accessible Python implementation that utilizes existing Python packages [1]. For our problem set, we have developed a Python implementation of the Heeger-Bergen algorithm. This implementation leverages Pyrtools—a Python package for multi-scale image processing, adapted from Eero Simoncelli's matlabPyrTools—for steerable pyramid decomposition and image reconstruction [5]. Our goal is to provide a more accessible tool for the computer vision community, as many recent projects are Python-based. Furthermore, this Python implementation is designed to integrate seamlessly with existing Python packages, thereby enhancing its utility and application in various computer vision tasks.

1.1. Heeger-Bergen Algorithm

The Heeger-Bergen algorithm for texture synthesis is presented as follows:

Algorithm 4: Heeger-Bergen texture synthesis algorithm for grayscale images (without extension)

Input : Number of scales P, number of orientations Q, texture image u of size $M \times N$ such that M and N are multiples of 2^P, number of iterations N_{iter}
Output: Texture image v of size $M \times N$

1. **Input analysis:**
2. Compute and store the steerable pyramid with P scales and Q orientations of the input texture u.
3. **Output synthesis:**
4. Initialize v with a Gaussian white noise.
5. Match the gray-level histogram of v with the gray-level histogram of the input u.
6. **for** iteration $i = 1$ **to** N_{iter} **do**
7. Compute the steerable pyramid of v.
8. For each of the $PQ + 2$ images of this pyramid, apply histogram matching with the corresponding image of the pyramid of u.
9. Apply the image reconstruction algorithm to this new histogram-matched pyramid and store the obtained image in v.
10. Match the gray-level histogram of v with the gray-level histogram of the input u.
11. **end**
12. Return v.

Figure 1. Heeger-Bergen Algorithm [1]

A fundamental component of this algorithm is the steerable pyramid decomposition and image reconstruction. Although the original paper provides a detailed mathematical description of the construction of the steerable pyramid algorithm, we have opted to use Pyrtools for the steerable pyramid decomposition and image reconstruction in this problem set, due to the complexity involved in manually constructing them. For students wishing to gain a better understanding of the steerable pyramid decomposition, we have included an implementation that we created using Python at the end of the problem set.

In Pyrtools, the orientations (height) and scales (order) of the steerable pyramid can be specified as follows:

```
pyr = pt.pyramids.SteerablePyramidSpace
      (source_image, height=4, order=3)
pyr.pyr_coeffs
```

In this example, we obtain the steerable pyramid coefficients of an image, totaling $4 \times 3 + 2$ (the additional 2 accounts for the high and low-frequency residuals), resulting in 14 sub-bands for an image. Furthermore, Heeger-Bergen algorithm also utilizes image reconstruction. Since the steerable pyramid is designed to be self-inverting, we can reconstruct an image from its pyramid using the same filters employed in the decomposition process. This reconstruction can also be achieved using Pyrtools, as demonstrated below [5]:

```
res = pyr.recon_pyr()
```

1.2. Histogram Matching Algorithm

In addition to steerable pyramid decomposition and image reconstruction, another fundamental component of texture synthesis is the histogram matching algorithm. There are several methods for implementing this algorithm.

The original Heeger-Bergen histogram matching algorithm involves adjusting the histogram of an input image to align with a desired histogram shape. This is achieved using the cumulative distribution function (CDF) of one image and the inverse CDF of another. The algorithm calculates the histogram of the input image using 256 bins, providing a detailed representation of the image's intensity distribution. The CDF, derived from this histogram, maps the pixel intensities to a normalized scale. The inverse CDF performs the reverse operation, mapping from the normalized scale back to the original pixel intensity range. While this method allows for high precision in texture synthesis, it has a limitation due to unnecessary quantizations. The subband histograms of many "natural" images exhibit characteristic shapes, suggesting that fewer parameters than the 256 bins might be sufficient [2].

Consequently, we opted for a simpler histogram matching implementation by [2]. This method sorts the pixel lists of two images, u and v, and assigns to the pixel of u having rank k the gray-value of the pixel of v having the same rank. This algorithm ensures that the output has exactly the same histogram as the source image while preserving the relative rank of the input pixels [1]. The algorithm is illustrated in Figure 2.

Additionally, histogram matching can be applied to images of different sizes. When the gaussian white noise initialized input image has dimensions that are multiples of the texturized source image, modifications are made. For

Algorithm 3: Histogram matching

Input : Input image u, reference image v (both images have size $M \times N$)
Output: Image u having the same histogram as v (the input u is lost)

1. Define $L = MN$ and describe the images as vectors of length L (e.g. by reading values line by line).
2. **Sort the reference image v:**
3. Determine the permutation τ such that $v_{\tau(1)} \leq v_{\tau(2)} \leq \cdots \leq v_{\tau(L)}$.
4. **Sort the input image u:**
5. Determine the permutation σ such that $u_{\sigma(1)} \leq u_{\sigma(2)} \leq \cdots \leq u_{\sigma(L)}$.
6. **Match the histogram of u:**
7. **for** rank $k = 1$ **to** L **do**
8. $\quad u_{\sigma(k)} \leftarrow v_{\tau(k)}$ (the k-th pixel of u takes the gray-value of the k-th pixel of v).
9. **end**

Figure 2. Histogram Matching Algorithm [1]

example, if the source image v has size $M \times N$ and the input image has size $r_1 M \times r_2 N$, where $r_1, r_2 = 2, 3, \ldots$, line 8 of the histogram matching algorithm is replaced by [1]:

$$u_{\sigma(r_1 r_2 (k-1)+i)} \leftarrow v_{\tau(k)}, \text{ for } i = 1, 2, \ldots, r_1 * r_2$$

1.3. Color Texture Synthesis

For color textures, independently synthesizing each of the three color channels often leads to inaccurate colors. This is because the RGB components of a typical texture image are not independent; they are often correlated. To address this, Heeger and Bergen proposed using Principal Component Analysis (PCA) to decorrelate the color channels. By transforming the RGB color space into the PCA color space, the color channels become decorrelated and nearly independent. It's important to note that while the PCA channels are decorrelated in terms of their intensity values, they are not completely independent. This partial decorrelation is due to PCA focusing solely on color correlations and not on spatial correlations [2]. The Heeger-Bergen texture synthesis algorithm for RGB color textures involves the following steps:

1. Compute the PCA (Principal Component Analysis) in the color space of the input image u.
2. Determine the C=channels of u in the PCA color space.
3. Apply the texture synthesis algorithm shown in 1 on each PCA channel. This gives an output texture v in the PCA color space.
4. Convert the image v in the RGB color space by applying the procedure described below. The obtained RGB image is the output of the algorithm [1].

$$(v_{1,i}, v_{2,i}, v_{3,i})^T \leftarrow (m_R, m_B, m_G)^T + P(v_{1,i}, v_{2,i}, v_{3,i})^T$$

where P is the orthogonal basis matrix from PCA, and (m_R, m_B, m_G) are channel-wise mean values in original image.

The final output is an RGB image that results from this algorithm, representing a synthesized texture that maintains the statistical properties of the original while overcoming the limitations of color channel interdependence. In the problem set, PCA from sklearn.decomposition can be used to extend the Heeger-Bergen algorithm to color textures [4].

1.4. Edge Handling

Another extension explored in the problem set, related to the Heeger-Bergen algorithm, concerns edge handling. In the real world, textures are never truly periodic. However, pyramid decomposition, which is based on the Discrete Fourier Transform (DFT) that assumes input images are periodic, introduces discontinuity in edge handling. Proper edge handling is crucial as it ensures the synthesized texture appears seamless, without abrupt transitions at the edges. These transitions can lead to visual artifacts that disrupt the texture's coherence and uniformity. A straightforward method to mitigate this issue is to use mirror symmetrization at the borders. This technique works by reflecting the image across its borders, creating a repeating pattern, which effectively smoothens out the edges of the synthesized image. Additionally, although mirror symmetrization ensures continuity at the borders, it is not perfect, as it introduces artificial orientations in the input texture. Therefore, we will explore another edge handling method [3] by replacing the input texture with its periodic component. The period component approach aligns the texture pattern at the edges, making it useful for textures that have an inherent repetitive nature. The implementation details for calculating the periodic component are explained in the following Figure 3:

1. Compute the discrete Laplacian $\Delta_i u$ of u.
2. Compute the DFT $\widehat{\Delta_i u}$ of $\Delta_i u$.
3. Compute the DFT $\hat{p}$ of p by inverting the discrete periodic Laplacian:

$$\begin{cases} \hat{p}_{m,n} = \left(4 - 2\cos\left(\frac{2m\pi}{M}\right) - 2\cos\left(\frac{2n\pi}{N}\right)\right)^{-1} \widehat{\Delta_i u}_{m,n}, & \text{if } (m,n) \in \hat{\Omega}_{M,N} \setminus \{(0,0)\} \\ \hat{p}_{0,0} = \sum_{k=0}^{M-1} \sum_{l=0}^{N-1} u_{k,l} & \text{if } (m,n) = (0,0) \end{cases}$$

4. Compute p by inverse DFT.

Figure 3. Edge Handling: Periodic Component [1]

2. Methodology

In the problem set, we have formulated seven questions, each comprising several sub-questions, pertaining to the Heeger-Bergen Texture Synthesis Algorithm. The Google Colab Notebook for this problem set, along with its solutions and related texture images, is included with the submission for this project.

Q1.1 focuses on constructing the steerable pyramid decomposition. Assuming that Q1.1 is completed, Q1.2 then asks students to reconstruct the image from its steerable pyramid representation.

Q2 is concerned with creating the histogram matching algorithm, as illustrated in Figure 2. Q3 extends this concept by adapting the histogram matching function for scaled images, where the synthesized image and the input texture image do not have the same number of pixels.

Q4 relates to constructing the main architecture of the Heeger-Bergen Algorithm, utilizing the components developed in Questions 1-3. This process will follow the pseudocode presented in Figure 1.

Q5 aims to extend the approach in Q4 to color textures. It involves performing Principal Component Analysis (PCA) on the color channels, identifying the most significant component, conducting HB texture synthesis on this component, and then mapping it back to the color space.

Q6 and Q7 explore edge handling, guiding students to try out both mirror symmetrization and periodic components.

3. Results

In the problem set solutions, we can visually inspect how the synthesized textured images compare to the source textured images throughout the different algorithms that are used for texture synthesis.

4. Conclusion

In this problem set, we have explored the multifaceted applications and theoretical underpinnings of steerable pyramids in image processing. Through a series of practical exercises and theoretical discussions, we hope students have gained insights into the nuances of multi-scale and multi-orientation image analysis. The exercises were designed not only to provide hands-on experience with steerable pyramids but also to foster a deeper understanding of their significance in various image processing tasks. We hope this problem set enhances the students' technical skills in image analysis. As we conclude, we encourage students to delve deeper into texture synthesis by attempting to recreate our implementation of the steerable pyramid decomposition, which can be found at the end of the problem set.

References

[1] Thibaud Briand et al. "The Heeger & Bergen Pyramid Based Texture Synthesis Algorithm". In: *Image Processing On Line* 4 (2014), pp. 276–299. DOI: 10.5201/ipol.2014.79.

[2] D.J. Heeger and J.R. Bergen. "Pyramid-based texture analysis/synthesis". In: *Proceedings., International Conference on Image Processing*. Vol. 3. 1995, 648–651 vol.3. DOI: 10.1109/ICIP.1995.537718.

[3] L Moisan. “L. Periodic Plus Smooth Image Decomposition.” In: *J Math Imaging Vis 39, 161–179 (2011)*. 4 (2011). DOI: 10.1007/s10851-010-0227-1.

[4] Fabian Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[5] Eero Simoncelli et al. *Pyrtools: tools for multi-scale image processing*. Version v1.0.2. 2023. DOI: 10.5281/zenodo.10161031. URL: https://github.com/LabForComputationalVision/pyrtools.

[6] Massachusetts Institute of Technology. *Lecture 7: Texture Analysis and Synthesis*. MIT Course 6.869: Advances in Computer Vision. 2019. URL: http://6.869.csail.mit.edu/fa19/lectures/notes_lecture_7.pdf.

HB_Texture_Synthesis_Problem_Set_Solution

December 13, 2023

0.1 Heeger & Bergen Texture Synthesis Algorithms

The Heeger & Bergen (HB) texture synthesis algorithm is a method used to generate realistic textures. It involves analyzing the statistical properties of a given texture and then using this information to synthesize new textures with similar characteristics.

The textured images in the `images` folder are from Thibaud Briand et al. "The Heeger & Bergen Pyra- mid Based Texture Synthesis Algorithm". In: Image Processing On Line 4 (2014), pp. 276–299. DOI: 10. 5201/ipol.2014.79.

1 Import required libraries

[1]:
```
!pip install pyrtools
```

```
Requirement already satisfied: pyrtools in /usr/local/lib/python3.10/dist-
packages (1.0.2)
Requirement already satisfied: numpy>=1.1 in /usr/local/lib/python3.10/dist-
packages (from pyrtools) (1.23.5)
Requirement already satisfied: scipy>=0.18 in /usr/local/lib/python3.10/dist-
packages (from pyrtools) (1.11.4)
Requirement already satisfied: matplotlib>=1.5 in
/usr/local/lib/python3.10/dist-packages (from pyrtools) (3.7.1)
Requirement already satisfied: tqdm>=4.29 in /usr/local/lib/python3.10/dist-
packages (from pyrtools) (4.66.1)
Requirement already satisfied: requests>=2.21 in /usr/local/lib/python3.10/dist-
packages (from pyrtools) (2.31.0)
Requirement already satisfied: contourpy>=1.0.1 in
/usr/local/lib/python3.10/dist-packages (from matplotlib>=1.5->pyrtools) (1.2.0)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-
packages (from matplotlib>=1.5->pyrtools) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in
/usr/local/lib/python3.10/dist-packages (from matplotlib>=1.5->pyrtools)
(4.46.0)
Requirement already satisfied: kiwisolver>=1.0.1 in
/usr/local/lib/python3.10/dist-packages (from matplotlib>=1.5->pyrtools) (1.4.5)
Requirement already satisfied: packaging>=20.0 in
/usr/local/lib/python3.10/dist-packages (from matplotlib>=1.5->pyrtools) (23.2)
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/dist-
```

```
packages (from matplotlib>=1.5->pyrtools) (9.4.0)
Requirement already satisfied: pyparsing>=2.3.1 in
/usr/local/lib/python3.10/dist-packages (from matplotlib>=1.5->pyrtools) (3.1.1)
Requirement already satisfied: python-dateutil>=2.7 in
/usr/local/lib/python3.10/dist-packages (from matplotlib>=1.5->pyrtools) (2.8.2)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.10/dist-packages (from requests>=2.21->pyrtools) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-
packages (from requests>=2.21->pyrtools) (3.6)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.10/dist-packages (from requests>=2.21->pyrtools) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.10/dist-packages (from requests>=2.21->pyrtools)
(2023.11.17)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-
packages (from python-dateutil>=2.7->matplotlib>=1.5->pyrtools) (1.16.0)
```

[2]:
```python
import numpy as np
import cv2
import os
import glob
import copy
import pyrtools as pt
import matplotlib.pyplot as plt
from scipy.special import factorial
import math
from pyrtools.pyramids.c.wrapper import pointOp
```

1.1 Q1.1 Given an image, contruct it's steerable pyramids representation

A steerable pyramid is a multi-resolution image representation that can be efficiently computed and analyzed. It is particularly useful for capturing and decomposing image information at different scales and orientations. The construction of a steerable pyramid involves applying a series of filtering and downsampling operations, often using filters that are steerable in different directions.

For P scales, and Q orientations of an image, a steerable pyramid representation has a total of $PQ + 2$ filters.

Consult the `pyrtools` documentation for creating the `steerable_pyramid_representation` function.

[3]:
```python
def steerable_pyramid_representation(image, P, Q):
    """
    Computes the steerable pyramid representation of an image.

    Parameters:
    - image: 2D array, input image
    - P: int, number of scales
    - Q: int, number of orientations
```

```
    Returns:
    - pyr_coeffs: dictionary, coefficients of the steerable pyramid
    - pyr_sizes: dictionary, sizes of the pyramid levels
    """
    pyramid = pt.pyramids.SteerablePyramidFreq(image, P, Q)
    return pyramid.pyr_coeffs, pyramid.pyr_size
```

1.1.1 Test Steerable Pyramid Representation

Test the steerable pyramid representation using `images/paperwall.jpeg` by plotting.

[4]:
```
import cv2
image = cv2.cvtColor(cv2.imread("images/paperwall.jpeg"), cv2.COLOR_BGR2GRAY)
P = 3
Q = 4
pyr_coeffs, pyr_sizes = steerable_pyramid_representation(image, P, Q)
```

1.2 Q1.2 Given a steerable pyramid representation of an image, reconstruct the image

Since the steerable pyramid is designed to be self-inverting, we can reconstruct an image from its pyramid using the same filters employed in the decomposition process.

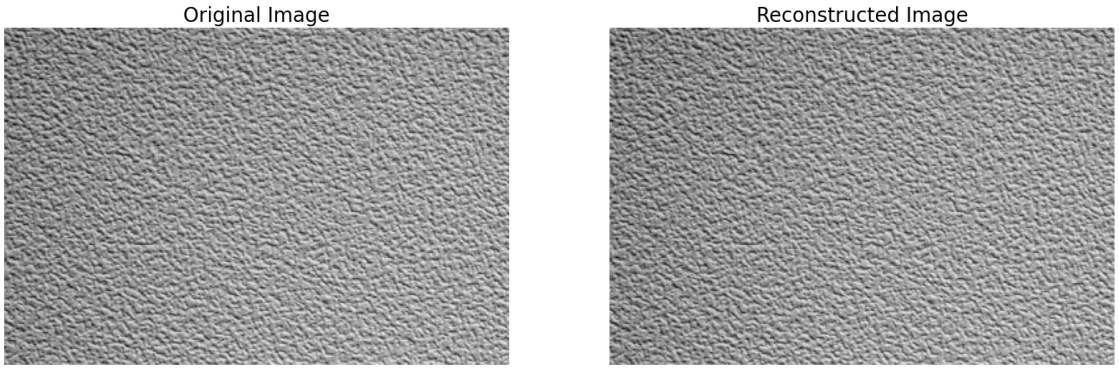
Consult the `pyrtools` documentation for creating the `reconstruct` function. Test the `reconstruct` function using the texturized image `images/paperwall.jpeg` by plotting.

[5]:
```
def reconstruct_image(pyr_coeffs, pyr_sizes):
    """
    Computes the reoncsturcted image using the coefficients of the steerable␣
 ↪pyramid.

    Parameters:
    - pyr_coeffs: dictionary, coefficients of the steerable pyramid
    - pyr_sizes: dictionary, sizes of the pyramid levels

    Returns:
    reconstructed_image
    """

    height, width = pyr_sizes['residual_highpass']
    recon_keys = list(pyr_coeffs.keys())
    P = max([key[0] for key in recon_keys if type(key) is tuple]) + 1
    Q = max([key[1] for key in recon_keys if type(key) is tuple])
    pyramid = pt.pyramids.SteerablePyramidFreq(np.zeros((height, width)), P, Q)
    pyramid.pyr_coeffs = pyr_coeffs
    pyramid.pyr_size = pyr_sizes
    return pyramid.recon_pyr()
```

1.2.1 Test Image Reconstruction

[6]:
```
reconstructed_image = reconstruct_image(pyr_coeffs, pyr_sizes)
fig, ax = plt.subplots(1, 2, figsize=(20, 10))
ax[0].imshow(image, cmap='gray')
ax[0].set_title('Original Image', fontsize=20)
ax[0].axis('off')
ax[1].imshow(reconstructed_image, cmap='gray')
ax[1].set_title('Reconstructed Image', fontsize=20)
ax[1].axis('off')
```

[6]:
```
(-0.5, 383.5, 255.5, -0.5)
```



1.3 Q2. Histogram Matching Function

Histogram matching is used to adjust the color distribution of an image to match a specified histogram. In this context, the histogram denotes the spread of pixel intensities. It is important to note that histogram matching is an ill-posed problem due to the absence of a unique solution. This arises from the fact that different images can possess similar or identical histograms, leading to ambiguity in determining a unique solution.

1.3.1 Algorithm: Histogram Matching

Input: Input image u, reference image v (both images have size $M \times N$)

Output: Image u having the same histogram as v (the input u is lost)

1. Define $L = MN$ and describe the images as vectors of length L (e.g., by reading values line by line).
2. Sort the reference image v:
3. Determine the permutation τ such that $v_{\tau(1)} \leq v_{\tau(2)} \leq ... \leq v_{\tau(L)}$.
4. Sort the input image u:
5. Determine the permutation σ such that$u_{\sigma(1)} \leq u_{\sigma(2)} \leq ... \leq u_{\sigma(L)}$.
6. Match the histogram of u:
   - for rank$k = 1$ to L do
     - $u_{\sigma(k)} \leftarrow v_{\tau(k)}$(the k-th pixel of u takes the gray-value of the k-th pixel of v).

- end

Given an input image u and a reference image v of the same size, histogram matching consists in changing the gray-level values of the input u so that it gets the same histogram as the reference image v. Complete the `histogram_matching` function in the cell below.

[7]:
```python
import numpy as np

def histogram_matching(v, u):
    """
    Aligns the image u onto the histogram of the reference image v.

    Parameters:
    - v: Reference image
    - u: Input image to be modified

    Returns:
    - Modified image u with aligned histogram
    """
    # Ensure that the sizes of v and u are the same
    assert v.size == u.size
    # Total number of elements in the images
    L = u.size
    # Get indices of v in sorted order
    indices_v = np.unravel_index(np.argsort(v, axis=None), v.shape)
    # Get indices of u in sorted order
    indices_u = np.unravel_index(np.argsort(u, axis=None), u.shape)
    # Rearrange pixels in u to match the histogram of v
    for k in range(L):
        row_u, col_u = indices_u[0][k], indices_u[1][k]
        row_v, col_v = indices_v[0][k], indices_v[1][k]
        u[row_u][col_u] = v[row_v][col_v]
    return u
```

1.4 Q3. Histogram Matching Function for scaled images

The above function aligns histograms when the image u and reference image v have the same shape. Often, we would expect the synthesized texture to be much larger than the reference texture in texture synthesis. In this case number of pixels in synthesized image and input texture image are not the same. Adapt the histogram matching function for scaled images.

Consider an image u which is magnified by (`width_scale`, `height_scale`) compared to image u on width and height dimensions. How can the histogram matching function be modified to effectively operate with images of different scales?

[8]:
```python
def histogram_matching_scaled(v, u, width_scale, height_scale):
    # Align u onto the histogram of v
    # Calculate the scaling factor for the number of pixels in u compared to v
    factor_scale = width_scale * height_scale
```

```
    # Ensure that the total number of pixels in u is the same as v after scaling
    assert v.size * factor_scale == u.size
    # L is the size of v (number of pixels)
    L = v.size
    # Get indices of pixels in v in sorted order
    indices_v = np.unravel_index(np.argsort(v, axis=None), v.shape)
    # Get indices of pixels in u in sorted order
    indices_u = np.unravel_index(np.argsort(u, axis=None), u.shape)
    # Iterate over the pixels in v
    for k in range(L):
        # Assign pixels in u based on the sorted order of pixels in v
        # Get the row and column indices of the k-th pixel in v
        row_v, col_v = indices_v[0][k], indices_v[1][k]
        # Iterate over the scaled range corresponding to the k-th pixel in v
        for _ in range(int(k * factor_scale), int(k * factor_scale +␣
↪factor_scale)):
            # Get the row and column indices of the corresponding pixel in u
            row_u, col_u = indices_u[0][_], indices_u[1][_]
            # Assign the pixel value from v to the corresponding location in u
            u[row_u][col_u] = v[row_v][col_v]
    # Return the modified u after histogram matching
    return u
```

1.5 Q4. Heeger Bergen Texture Synthesis Algorithm

We can now present the texture synthesis algorithm developed by Heeger and Bergen. Starting from a white noise image, histogram matchings are performed to the texture image alternatively in the image domain and in the steerable pyramid domain. After a few iterations, all the output histograms will match the ones of the input texture, and thus, the output texture will be visually similar to the input texture. The pseudo code of the algorithm is detailed below.

1.5.1 Algorithm: Heeger-Bergen Texture Synthesis Algorithm for Grayscale Images (without extension)

Input: - Number of scales P - Number of orientations Q - Texture image u of size M × N (where M and N are multiples of 2^P) - Number of iterations Niter

Output: - Texture image v of size M × N

1. **Input Analysis:**
   - Compute and store the steerable pyramid with P scales and Q orientations of the input texture u.
2. **Output Synthesis:**
   - Initialize v with a Gaussian white noise.
   - Match the gray-level histogram of v with the gray-level histogram of the input u.
3. **Iteration Loop:** `for iteration i = 1 to Niter do`
   - Compute the steerable pyramid of v.
   - For each of the P Q + 2 images of this pyramid, apply histogram matching with the corresponding image of the pyramid of u.

- Apply the image reconstruction algorithm to this new histogram-matched pyramid and store the obtained image in v.
- Match the gray-level histogram of v with the gray-level histogram of the input u.

4. **Return Result:** `end  Return v.`

Complete the `heeger_bergen_texture_synthesis` algorithm, which takes in: - A reference texture `texture` - Number of scales `P` - Number of orientations `Q` for steerable pyramid representation - `Niter` number of iterations - `width_scale` and `height_scale` denoting how much we want to magnify the synthesized texture by.

Test the function by synthesizing for all images found in the `images` folder.

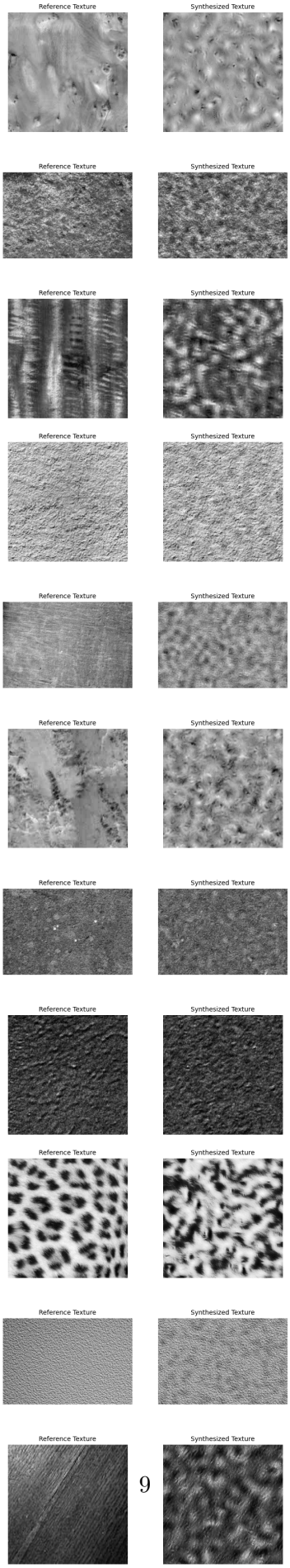
[9]:
```
def heeger_bergen_texture_synthesis(texture, P, Q, Niter, width_scale=1,␣
↪height_scale=1):
    """
    Heeger-Bergen Texture Synthesis Algorithm for Grayscale Images.

    Parameters:
    - texture: Reference texture image
    - P: Number of scales
    - Q: Number of orientations for steerable pyramid representation
    - Niter: Number of iterations
    - width_scale: Magnification factor for width (default is 1)
    - height_scale: Magnification factor for height (default is 1)

    Returns:
    - Synthesized texture image
    """
    # Input analysis
    pyramid_texture_pyr_coeffs, pyramid_texture_pyr_sizes =␣
↪steerable_pyramid_representation(texture, P, Q)

    # Output synthesis
    texture_synth = np.random.normal(size=(texture.shape[0] * height_scale,␣
↪texture.shape[1] * width_scale))  # Initialize v with Gaussian white noise

    # Match the gray-level histogram of v with u
    texture_synth = histogram_matching_scaled(texture, texture_synth,␣
↪width_scale=width_scale, height_scale=height_scale)

    for i in range(Niter):
        # Compute the steerable pyramid of v
        pyramid_texture_synth_pyr_coeffs, pyramid_texture_synth_pyr_sizes =␣
↪steerable_pyramid_representation(texture_synth, P, Q)

        # Apply histogram matching for each image in the pyramid
        for pq_key in pyramid_texture_synth_pyr_coeffs:
```

```
            matched_image =␣
↪histogram_matching_scaled(pyramid_texture_pyr_coeffs[pq_key],␣
↪pyramid_texture_synth_pyr_coeffs[pq_key], width_scale=width_scale,␣
↪height_scale=height_scale)
            pyramid_texture_synth_pyr_coeffs[pq_key] = copy.
↪deepcopy(matched_image)

        # Apply image reconstruction algorithm
        texture_synth = reconstruct_image(pyramid_texture_synth_pyr_coeffs,␣
↪pyramid_texture_synth_pyr_sizes)

        # Match the gray-level histogram of v with u
        texture_synth = histogram_matching_scaled(texture, texture_synth,␣
↪width_scale=width_scale, height_scale=height_scale)

    return texture_synth
```

1.5.2 Synthesize textures for all references images (with same shape)

[10]:
```
import glob
from tqdm.auto import tqdm
image_files = glob.glob("images/*.jpeg")
n_images = len(image_files)
fig, ax = plt.subplots(n_images, 2, figsize=(8, 4 * n_images))
for idx, image_file in tqdm(enumerate(image_files)):
    texture = cv2.cvtColor(cv2.imread(image_file), cv2.COLOR_BGR2GRAY)
    texture_synth = heeger_bergen_texture_synthesis(texture, P, Q, Niter=10,␣
↪width_scale=1, height_scale=1)
    ax[idx][0].imshow(texture, cmap='gray')
    ax[idx][0].set_title('Reference Texture', fontsize=10)
    ax[idx][0].axis('off')
    ax[idx][1].imshow(texture_synth, cmap='gray')
    ax[idx][1].set_title('Synthesized Texture', fontsize=10)
    ax[idx][1].axis('off')
```

```
0it [00:00, ?it/s]
```


Reference Texture
Synthesized Texture
Reference Texture
Synthesized Texture
Reference Texture
Synthesized Texture
Reference Texture
Synthesized Texture
Reference Texture
Synthesized Texture
Reference Texture
Synthesized Texture
Reference Texture
Synthesized Texture
Reference Texture
Synthesized Texture
Reference Texture
Synthesized Texture
Reference Texture
Synthesized Texture

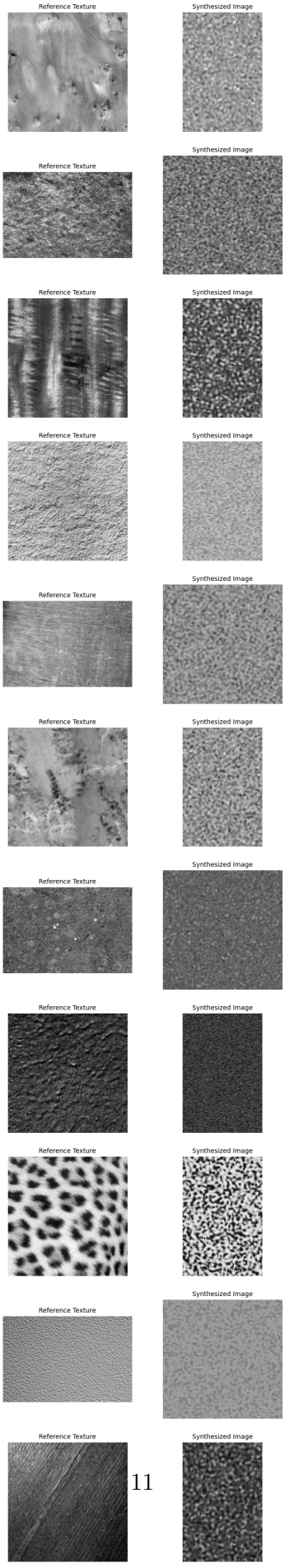
Observation: From visual inspection, we can inspect that the final grayscale synthesized texture imges looks pretty close to the grayscale referene texture images.

1.5.3 Synthesize textures for all references images with magnified shape

[11]:
```
bimage_files = glob.glob("images/*.jpeg")
n_images = len(image_files)
fig, ax = plt.subplots(n_images, 2, figsize=(8, 4 * n_images))
for idx, image_file in tqdm(enumerate(image_files)):
    print(image_file)
    texture = cv2.cvtColor(cv2.imread(image_file), cv2.COLOR_BGR2GRAY)
    texture_synth = heeger_bergen_texture_synthesis(texture, P, Q, Niter=10,␣
 ↪width_scale=2, height_scale=3)
    ax[idx][0].imshow(texture, cmap='gray')
    ax[idx][0].set_title('Reference Texture', fontsize=10)
    ax[idx][0].axis('off')
    ax[idx][1].imshow(texture_synth, cmap='gray')
    ax[idx][1].set_title('Synthesized Image', fontsize=10)
    ax[idx][1].axis('off')
```

```
0it [00:00, ?it/s]
```

```
images/randomwood.jpeg
images/stonewall.jpeg
images/straightwood.jpeg
images/pinkwall.jpeg
images/cementwall.jpeg
images/coloredwall.jpeg
images/concretewall.jpeg
images/sand.jpeg
images/stainedfur.jpeg
images/paperwall.jpeg
images/wood_edge_blending.jpeg
```


Reference Texture
Synthesized Image
Reference Texture
Synthesized Image
Reference Texture
Synthesized Image
Reference Texture
Synthesized Image
Reference Texture
Synthesized Image
Reference Texture
Synthesized Image
Reference Texture
Synthesized Image
Reference Texture
Synthesized Image
Reference Texture
Synthesized Image
Reference Texture
Synthesized Image
Reference Texture
Synthesized Image

Observation: From visual inspection, we can inspect that the final grayscale synthesized texture imges can scale the grayscale referene texture image and still looks visually similar.

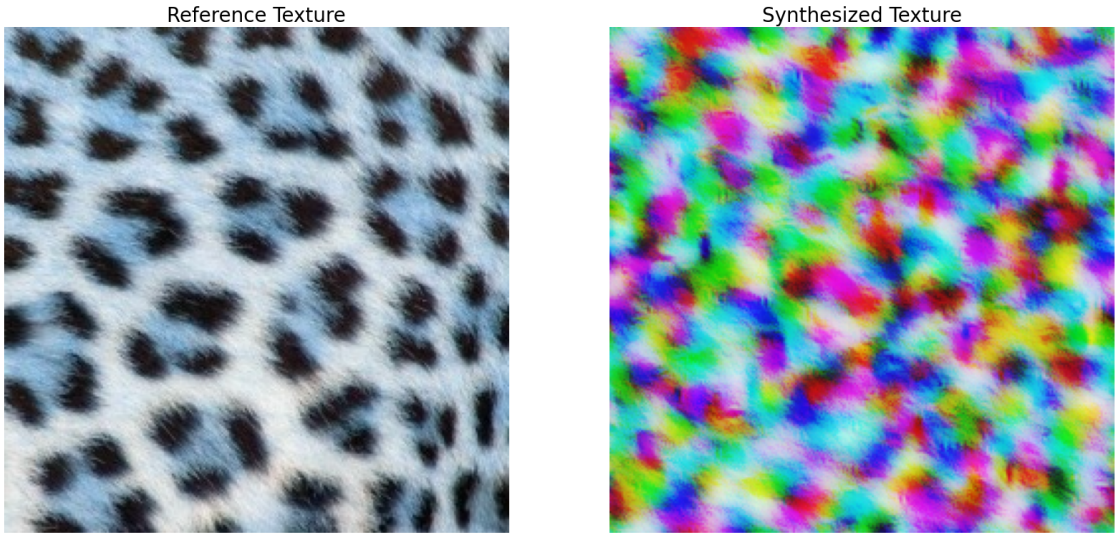
1.6 Q5 Extending HB Texture Synthesis to Color Space

Question The Heeger-Bergen algorithm relies on histogram matching, making it well-defined only for grayscale images. How can we extend it to color spaces? Is it possible to run texture synthesis independently on each color and then combine the results? Why or why not? Try testing with the color textured image `images/stainedfur.jpeg`.

Solution: Synthesizing a color texture by considering the three RGB color channels as independent is not satisfying. The RGB color channels are highly correlated. Therefore, synthesizing a color texture by independently synthesizing its three color channels does not respect the channel correlations. This approach leads to the creation of wrong colors. An example of why this doesn't work is given below.

[12]:
```
texture = cv2.imread("images/stainedfur.jpeg")

n_channels = texture.shape[2]
P = 3
Q = 4
texture_synth = np.array([heeger_bergen_texture_synthesis(texture[:, :,␣
 ↪channel], P, Q, Niter=10) for channel in range(n_channels)])
texture_synth = texture_synth.transpose(1, 2, 0)

fig, ax = plt.subplots(1, 2, figsize=(20, 10))
ax[0].imshow(texture, cmap='gray')
ax[0].set_title('Reference Texture', fontsize=20)
ax[0].axis('off')
ax[1].imshow(texture_synth.astype(np.int32), cmap='gray')
ax[1].set_title('Synthesized Texture', fontsize=20)
ax[1].axis('off')
```

[12]:
```
(-0.5, 255.5, 255.5, -0.5)
```



The Heeger-Bergen texture synthesis algorithm for RGB color textures is the following: 1. Compute the PCA color space of the input image u. 2. Determine the channels of u in the PCA color space. 3. Apply the texture synthesis algorithm implemented above on each PCA channel. This gives an output texture v in the PCA color space. 4. Convert the image v in the RGB color space by applying the procedure described above. The obtained RGB image is the output of the algorithm.

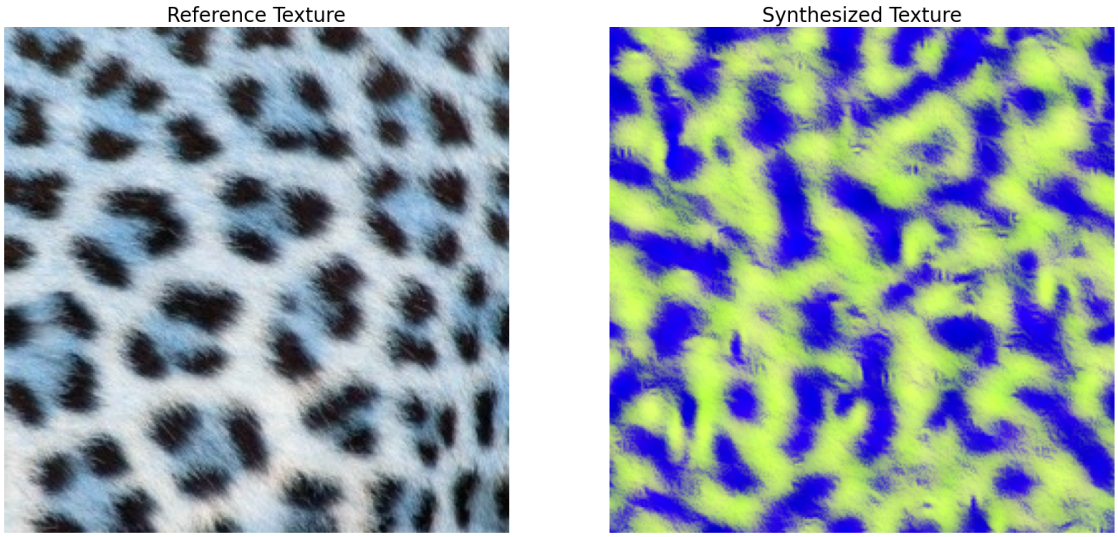
Complete the `heeger_bergen_texture_synthesis_color` function with the above Heeger-Bergen texture synthesis for RGB color textures, consulting the documentation for `PCA` found in `sklearn.decomposition`. Test the function using the color textured image in `images/stainedfur.jpeg` by plotting.

[13]:
```
from sklearn.decomposition import PCA

def heeger_bergen_texture_synthesis_color(image, P, Q, Niter, width_scale=1,␣
↪height_scale=1):
    # Step 2: Convert RGB to PCA Color Space
    # Reshape the image to a 2D array (pixels as rows, color channels as␣
↪columns)
    pixels = image.reshape((-1, 3))

    # Perform PCA
    num_components = 3  # Number of principal components to retain
    pca = PCA(n_components=num_components)
    pca_result = pca.fit_transform(pixels)

    # Step 3: Determine Channels in PCA Color Space
    # The channels in the PCA color space are the transformed features along␣
↪the principal components
    pca_channels = pca_result[:, :num_components]
```

```
    # Now, pca_channels contains the channels in the PCA color space
    pca_channels_synth = []
    for channel in range(3):
        pca_channels_synth.append(heeger_bergen_texture_synthesis(pca_channels[:
 ↪, channel].reshape(*image.shape[:2]), P, Q, Niter))
    pca_channels_synth = np.array(pca_channels_synth).transpose(1, 2, 0)

    texture_synth = np.zeros(image.shape)

    for i in range(texture_synth.shape[0]):
        for j in range(texture_synth.shape[1]):
            texture_synth[i, j, :] = pca.mean_ + pca.components_ @␣
 ↪pca_channels_synth[i, j, :]

    return texture_synth


texture = cv2.imread("images/stainedfur.jpeg")
n_channels = texture.shape[2]
P = 3
Q = 4
Niter = 10
texture_synth = heeger_bergen_texture_synthesis_color(texture, P, Q, Niter,␣
 ↪width_scale=1, height_scale=1)

fig, ax = plt.subplots(1, 2, figsize=(20, 10))
ax[0].imshow(texture, cmap='gray')
ax[0].set_title('Reference Texture', fontsize=20)
ax[0].axis('off')
ax[1].imshow(texture_synth / 255.0, cmap='gray')
ax[1].set_title('Synthesized Texture', fontsize=20)
ax[1].axis('off')
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with
RGB data ([0..1] for floats or [0..255] for integers).
```

[13]: (-0.5, 255.5, 255.5, -0.5)



Observation: From the color texture synthesis algorithm using PCA, we can see that the synthesized texture for the color image looks more visually close to the source texture compared to just using the Heeger-Bergen Texture algorithm.

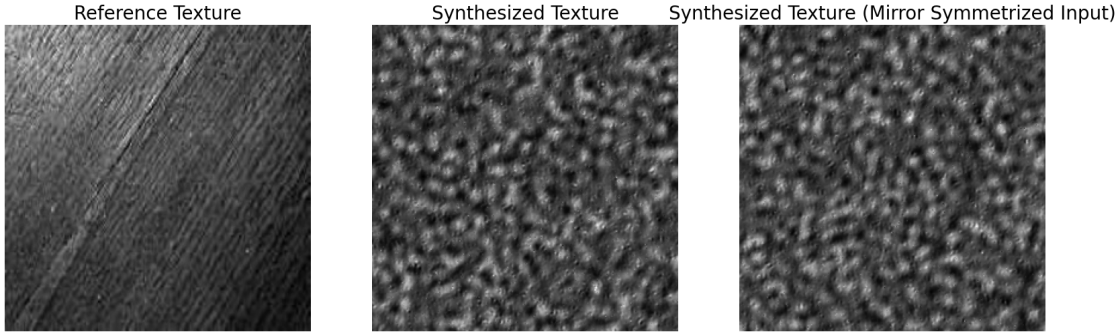
1.7 Q6. Edge Blending - Mirror Symmetrization

In the real world, textures are never periodic. However, the pyramid decomposition, which is based on the Discrete Fourier Transform (DFT) that treats input images as periodic, introduces discontinuity in handling edges. A simple method to address this issue is to use mirror symmetrization at the borders. Construct a mirror symmetrization function and test it using the image `images/randomwood.jpeg` by plotting.

[14]:
```
def edge_handling_mirror_symmetrization(image):
    # Get the width and height of the input image
    height, width = image.shape[:2]

    # Create an empty array for the mirrored image with double the width and␣
 ↪height
    image_mirror = np.zeros((2 * height, 2 * width))

    # Mirror symmetrization loop
    for i in range(width):
        for j in range(height):
            # Top-left quadrant
            image_mirror[j][i] = image[j][i]
            # Top-right quadrant
            image_mirror[j][i + width] = image[j][width - 1 - i]
            # Bottom-left quadrant
            image_mirror[height + j][i] = image[height - j - 1][i]
            # Bottom-right quadrant
```

```
            image_mirror[height + j][i + width] = image[height - j - 1][width -␣
 ↪i - 1]

    # Generate steerable pyramid representation for the original and mirrored␣
 ↪images
    pyramid_texture_pyr_coeffs, pyramid_texture_pyr_sizes =␣
 ↪steerable_pyramid_representation(image, P, Q)
    pyramid_texture_mirror_pyr_coeffs, pyramid_texture_mirror_pyr_sizes =␣
 ↪steerable_pyramid_representation(image_mirror, P, Q)

    # Adjust the mirrored pyramid coefficients to match the original image size
    for pq_key in pyramid_texture_mirror_pyr_coeffs:
        sub_band_height, sub_band_width =␣
 ↪pyramid_texture_mirror_pyr_sizes[pq_key]
        pyramid_texture_pyr_coeffs[pq_key] =␣
 ↪pyramid_texture_mirror_pyr_coeffs[pq_key][:sub_band_height // 2, :
 ↪sub_band_width // 2]

    # Reconstruct the image from the modified pyramid coefficients
    reconstructed_image = reconstruct_image(pyramid_texture_pyr_coeffs,␣
 ↪pyramid_texture_pyr_sizes)

    # Return the reconstructed image
    return reconstructed_image


P = 2
Q = 2
Niter = 10
texture = cv2.cvtColor(cv2.imread("images/wood_edge_blending.jpeg"), cv2.
 ↪COLOR_BGR2GRAY)
texture_mirror = edge_handling_mirror_symmetrization(texture)
texture_synth = heeger_bergen_texture_synthesis(texture, P, Q, Niter)
texture_synth_edge_handled = heeger_bergen_texture_synthesis(texture_mirror, P,␣
 ↪Q, Niter)

fig, ax = plt.subplots(1, 3, figsize=(20, 10))
ax[0].imshow(texture, cmap='gray')
ax[0].set_title('Reference Texture', fontsize=20)
ax[0].axis('off')
ax[1].imshow(texture_synth / 255.0, cmap='gray')
ax[1].set_title('Synthesized Texture', fontsize=20)
ax[1].axis('off')
ax[2].imshow(texture_synth_edge_handled / 255.0, cmap='gray')
ax[2].set_title('Synthesized Texture (Mirror Symmetrized Input)', fontsize=20)
ax[2].axis('off')
```

[14]: (-0.5, 255.5, 255.5, -0.5)



1.8 Q7. Edge Blending - Periodic Component

In Q6, we explored how to handle edges using mirror symmetrization. Although mirror symmetrization ensures continuity at the borders, it is not perfect, as it introduces artificial orientations in the input texture. Therefore, we will explore another edge handling method: replacing the input texture with its periodic component.

1. Compute the discrete Laplacian $\Delta_i u$ of u.
2. Compute the DFT $\widehat{\Delta_i u}$ of $\Delta_i u$.
3. Compute the DFT $\hat{p}$ of p by inverting the discrete periodic Laplacian:

$$\hat{p}_{m,n} = \begin{cases} (4 - 2\cos(\frac{2m\pi}{M}) - 2\cos(\frac{2n\pi}{N}))^{-1} \widehat{\Delta_i u}_{m,n}, & \text{if } (m,n) \in \hat{\Omega}_{M,N} \quad \{(0,0)\} \\ \hat{p}_{0,0} = \sum_{k=0}^{M-1} \sum_{l=0}^{N-1} u_{k,l}, & \text{if } (m,n) = (0,0) \end{cases}$$

4. Compute p by inverse DFT.

Use `images/randomwood.jpeg` to test the periodic component function following the above steps.

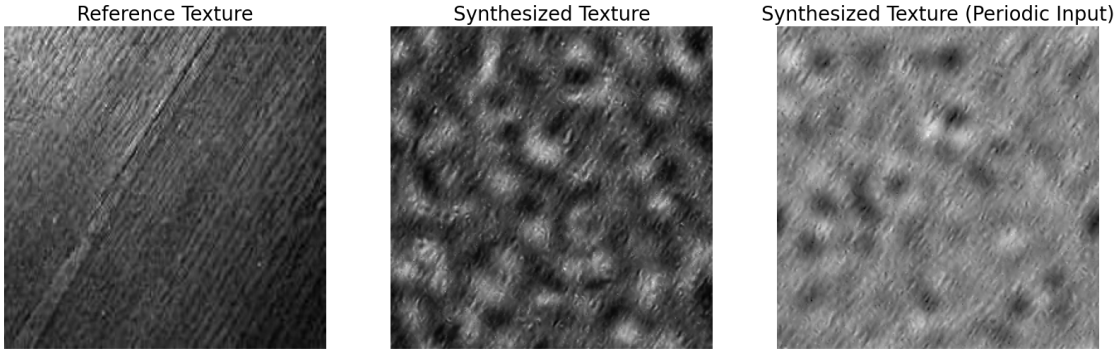
[15]:
```
image = cv2.imread('images/randomwood.jpeg')

def compute_p_hat(image, image_laplacian_dft):
    """
    Compute the function p_hat for given parameters.

    Parameters:
    - M, N: Size of the 2D grid.
    - omega: Set of excluded coordinates (m, n) as a list of tuples.
    - uk: 2D array representing the input grid.

    Returns:
    - p_hat: Resulting 2D array.
    """
    M, N = image.shape
```

```
    p_hat = np.zeros((M, N), dtype=complex)
    for m in range(M):
        for n in range(N):
            if m != 0 or n != 0:
                # Compute p_hat for (m, n) not in omega
                p_hat[m, n] = image_laplacian_dft[m, n] / (4 - 2 * np.cos(2 *␣
 ↪np.pi * m / M) - 2 * np.cos(2 * np.pi * n / N))
    p_hat[0, 0] = np.sum(image)
    return p_hat


def edge_handling_periodic_component(image):
    # Compute the Laplacian using the cv2.Laplacian function
    image_laplacian = cv2.Laplacian(image, cv2.CV_64F)
    image_laplacian_dft = np.fft.fft2(image_laplacian)
    image_laplacian_inverse = compute_p_hat(image, image_laplacian_dft)
    periodic_component = np.fft.ifft2(image_laplacian_inverse)
    return periodic_component
```

[16]:
```
P = 3
Q = 4
Niter = 10
texture = cv2.cvtColor(cv2.imread("images/wood_edge_blending.jpeg"), cv2.
 ↪COLOR_BGR2GRAY)
texture_periodic = edge_handling_periodic_component(texture)
texture_synth = heeger_bergen_texture_synthesis(texture, P, Q, Niter)
texture_synth_periodic = heeger_bergen_texture_synthesis(texture_periodic, P,␣
 ↪Q, Niter)

fig, ax = plt.subplots(1, 3, figsize=(20, 10))
ax[0].imshow(texture, cmap='gray')
ax[0].set_title('Reference Texture', fontsize=20)
ax[0].axis('off')
ax[1].imshow(texture_synth / 255.0, cmap='gray')
ax[1].set_title('Synthesized Texture', fontsize=20)
ax[1].axis('off')
ax[2].imshow(texture_synth_periodic / 255.0, cmap='gray')
ax[2].set_title('Synthesized Texture (Periodic Input)', fontsize=20)
ax[2].axis('off')
```

```
/usr/local/lib/python3.10/dist-packages/pyrtools/pyramids/pyramid.py:52:
ComplexWarning: Casting complex values to real discards the imaginary part
  self.image = np.array(image).astype(float)
<ipython-input-8-69c2e4b10afb>:23: ComplexWarning: Casting complex values to
real discards the imaginary part
  u[row_u][col_u] = v[row_v][col_v]
```

[16]: (-0.5, 255.5, 255.5, -0.5)



Observation: From the periodic component edge handling algorithm, we can see that the synthesized texture with the periodic input looks closer to the reference texture than the synthesized texture.

2 Appendix

2.1 Reference Steerable Pyramid Representation

The following code implements the steerable pyramid representation of an image. The code begins by preparing the image, creating coordinate grids, and computing various mathematical transformations like angle and log-polar coordinates. It then applies high-pass and low-pass filters to extract different frequency components of the image.

The code iterates over multiple scales, at each step computing bandpass filters for various orientations to capture different directional details, which is achieved through using Fourier transforms. The code returns a collection of coefficients stored in a dictionary representing the original image at various scales and orientations.

[17]:
```python
def rcosFn(width=1, position=0, values=(0, 1)):
    # Define the number of points in the waveform
    sz = 256   # arbitrary!
    # Generate a time vector with values between -pi and 2*pi
    X = np.pi * np.arange(-sz - 1, 2) / (2 * sz)
    # Generate the raised cosine waveform using the specified values
    Y = values[0] + (values[1] - values[0]) * np.cos(X) ** 2
    # Set the boundary values to ensure continuity
    Y[0] = Y[1]
    Y[sz + 2] = Y[sz + 1]
    # Adjust the position and width of the waveform
    X = position + (2 * width / np.pi) * (X + np.pi / 4)
    # Return the time vector (X) and the corresponding waveform (Y)
    return (X, Y)
```

```python
def steerable_pyramid_representation(image, P, Q):
    """
    Computes the steerable pyramid representation of an image.

    Parameters:
    - image: 2D array, input image
    - P: int, number of scales
    - Q: int, number of orientations

    Returns:
    - pyr_coeffs: dictionary, coefficients of the steerable pyramid
    - pyr_sizes: dictionary, sizes of the pyramid levels
    """

    # Initialize dictionaries to store pyramid coefficients and sizes
    pyr_coeffs = {}
    pyr_sizes = {}

    # Number of scales and orientations
    num_scales = P
    num_orientations = Q + 1

    # Width of the transition function
    twidth = 1

    # Dimensions of the input image
    dims = np.array(image.shape)

    # Center of the image
    ctr = np.ceil((np.array(dims) + 0.5) / 2).astype(int)

    # Create coordinate grids
    (xramp, yramp) = np.meshgrid(np.linspace(-1, 1, dims[1]+1)[:-1],
                                 np.linspace(-1, 1, dims[0]+1)[:-1])

    # Compute angle and log-polar coordinates
    angle = np.arctan2(yramp, xramp)
    log_rad = np.sqrt(xramp ** 2 + yramp ** 2)
    log_rad[ctr[0] - 1, ctr[1] - 1] = log_rad[ctr[0] - 1, ctr[1] - 2]
    log_rad = np.log2(log_rad)

    # Radial transition function (a raised cosine in log-frequency)
    (Xrcos, Yrcos) = rcosFn(twidth, (-twidth / 2.0), np.array([0, 1]))
    Yrcos = np.sqrt(Yrcos)
    YIrcos = np.sqrt(1.0 - Yrcos**2)
    loOmask = pointOp(log_rad, YIrcos, Xrcos[0], Xrcos[1]-Xrcos[0])
```

```
    # Compute the Fourier transform of the input image
    imdft = np.fft.fftshift(np.fft.fft2(image))

    # High-pass filter the image
    hi0mask = pointOp(log_rad, Yrcos, Xrcos[0], Xrcos[1] - Xrcos[0])
    hi0dft = imdft * hi0mask.reshape(imdft.shape[0], imdft.shape[1])
    hi0 = np.fft.ifft2(np.fft.ifftshift(hi0dft))

    # Store the high-pass residual
    pyr_coeffs['residual_highpass'] = np.real(hi0)
    pyr_sizes['residual_highpass'] = hi0.shape

    # Low-pass filter the image
    lo0mask = lo0mask.reshape(imdft.shape[0], imdft.shape[1])
    lodft = imdft * lo0mask

    # Iterate over scales
    for i in range(num_scales):
        Xrcos -= np.log2(2)

        # Generate lookup table for cosine values
        lutsize = 1024
        Xcosn = np.pi * np.arange(-(2 * lutsize + 1), (lutsize + 2)) / lutsize

        # Compute steerable filters' coefficients
        const = (2 **( 2 * Q)) * (factorial(Q, exact=True) ** 2)/␣
↪float(num_orientations * factorial(2 * Q, exact=True))
        Ycosn = np.sqrt(const) * (np.cos(Xcosn)) ** Q

        # Compute the frequency masks
        log_rad_test = np.reshape(log_rad, (1, log_rad.shape[0] * log_rad.
↪shape[1]))
        himask = pointOp(log_rad_test, Yrcos, Xrcos[0], Xrcos[1]-Xrcos[0])
        himask = himask.reshape((lodft.shape[0], lodft.shape[1]))

        # Compute angle masks
        anglemasks = []
        for b in range(num_orientations):
            angle_tmp = np.reshape(angle, (1, angle.shape[0] * angle.shape[1]))
            anglemask = pointOp(angle_tmp, Ycosn, Xcosn[0] + np.pi * b /␣
↪num_orientations, Xcosn[1] - Xcosn[0])

            anglemask = anglemask.reshape(lodft.shape[0], lodft.shape[1])
            anglemasks.append(anglemask)

            # Compute bandpass filter in the Fourier domain
```

```
            banddft = (-1j) ** Q * lodft * anglemask * himask
            band = np.fft.ifft2(np.fft.ifftshift(banddft))

            # Store the bandpass coefficients
            pyr_coeffs[(i, b)] = np.real(band.copy())
            pyr_sizes[(i, b)] = band.shape

        # Update dimensions for the next scale
        dims = np.array(lodft.shape)
        ctr = np.ceil((dims+0.5)/2).astype(int)
        lodims = np.ceil((dims-0.5)/2).astype(int)
        loctr = np.ceil((lodims+0.5)/2).astype(int)
        lostart = ctr - loctr
        loend = lostart + lodims

        # Update log_rad, angle, and lodft for the next scale
        log_rad = log_rad[lostart[0]:loend[0], lostart[1]:loend[1]]
        angle = angle[lostart[0]:loend[0], lostart[1]:loend[1]]
        lodft = lodft[lostart[0]:loend[0], lostart[1]:loend[1]]

        # Update low-pass masks
        YIrcos = np.abs(np.sqrt(1.0 - Yrcos**2))
        log_rad_tmp = np.reshape(log_rad, (1, log_rad.shape[0] * log_rad.
↪shape[1]))
        lomask = pointOp(log_rad_tmp, YIrcos, Xrcos[0], Xrcos[1]-Xrcos[0])
        lomask = lomask.reshape(lodft.shape[0], lodft.shape[1])

        lodft = lodft * lomask

    # Inverse Fourier transform of the final low-pass residual
    lodft = np.fft.ifft2(np.fft.ifftshift(lodft))

    # Store the final low-pass residual coefficients
    pyr_coeffs['residual_lowpass'] = np.real(np.array(lodft).copy())
    pyr_sizes['residual_lowpass'] = lodft.shape

    return pyr_coeffs, pyr_sizes
```

2.2 Reference: Reconstructing Image from steerable pyramid representation

The following code reconstructs an image from its steerable pyramid representation. The code begins by identifying key parameters, such as the number of scales and orientations, based on the provided pyramid coefficients. It then proceeds to create coordinate grids, calculate angles, and determine log-polar coordinates. The reconstruction process involves applying a series of filters—high-pass, low-pass, and bandpass—at different scales and orientations. These filters capture and reconstruct the various frequency components and directional details of the image from its pyramid representation. Finally, the code combines all these filtered components using Fourier transforms

to reconstruct the original image.

[18]:
```
def reconstruct_image(pyr_coeffs, pyr_sizes):

    recon_keys = list(pyr_coeffs.keys())

    P = max([key[0] for key in recon_keys if type(key) is tuple]) + 1
    Q = max([key[1] for key in recon_keys if type(key) is tuple])
    num_scales = P
    num_orientations = Q + 1

    # make list of dims and bounds
    bound_list = []
    dim_list = []
    # we go through pyr_sizes from smallest to largest
    for dims in sorted(pyr_sizes.values()):
        if dims in dim_list:
            continue
        dim_list.append(dims)
        dims = np.array(dims)
        ctr = np.ceil((dims+0.5)/2).astype(int)
        lodims = np.ceil((dims-0.5)/2).astype(int)
        loctr = np.ceil((lodims+0.5)/2).astype(int)
        lostart = ctr - loctr
        loend = lostart + lodims
        bounds = (lostart[0], lostart[1], loend[0], loend[1])
        bound_list.append(bounds)
    bound_list.append((0, 0, dim_list[-1][0], dim_list[-1][1]))
    dim_list.append((dim_list[-1][0], dim_list[-1][1]))

    # matlab code starts here
    dims = np.array(pyr_sizes['residual_highpass'])
    ctr = np.ceil((dims+0.5)/2.0).astype(int)

    (xramp, yramp) = np.meshgrid((np.arange(1, dims[1]+1)-ctr[1]) / (dims[1]/2.
↪),
                                 (np.arange(1, dims[0]+1)-ctr[0]) / (dims[0]/2.
↪))
    angle = np.arctan2(yramp, xramp)
    log_rad = np.sqrt(xramp**2 + yramp**2)
    log_rad[ctr[0]-1, ctr[1]-1] = log_rad[ctr[0]-1, ctr[1]-2]
    log_rad = np.log2(log_rad)

    # Radial transition function (a raised cosine in log-frequency):
    (Xrcos, Yrcos) = rcosFn(1, (-1/2.0), np.array([0, 1]))
    Yrcos = np.sqrt(Yrcos)
    YIrcos = np.sqrt(1.0 - Yrcos**2)
```

```
    # from reconSFpyrLevs
    lutsize = 1024

    Xcosn = np.pi * np.arange(-(2*lutsize+1), (lutsize+2)) / lutsize

    const = (2**(2*Q))*(factorial(Q, exact=True)**2) /␣
↪float(num_orientations*factorial(2*Q, exact=True))
    Ycosn = np.sqrt(const) * (np.cos(Xcosn))**Q

    # lowest band
    # initialize reconstruction
    if 'residual_lowpass' in recon_keys:
        nresdft = np.fft.fftshift(np.fft.fft2(pyr_coeffs['residual_lowpass']))
    else:
        nresdft = np.zeros_like(pyr_coeffs['residual_lowpass'])
    resdft = np.zeros(dim_list[1]) + 0j

    bounds = (0, 0, 0, 0)
    for idx in range(len(bound_list)-2, 0, -1):
        diff = (bound_list[idx][2]-bound_list[idx][0],
                bound_list[idx][3]-bound_list[idx][1])
        bounds = (bounds[0]+bound_list[idx][0], bounds[1]+bound_list[idx][1],
                  bounds[0]+bound_list[idx][0] + diff[0],
                  bounds[1]+bound_list[idx][1] + diff[1])
        Xrcos -= np.log2(2.0)
    nlog_rad = log_rad[bounds[0]:bounds[2], bounds[1]:bounds[3]]

    nlog_rad_tmp = np.reshape(nlog_rad, (1, nlog_rad.shape[0]*nlog_rad.
↪shape[1]))
    lomask = pointOp(nlog_rad_tmp, YIrcos, Xrcos[0], Xrcos[1]-Xrcos[0])
    lomask = lomask.reshape(nresdft.shape[0], nresdft.shape[1])
    lomask = lomask + 0j
    resdft[bound_list[1][0]:bound_list[1][2],
           bound_list[1][1]:bound_list[1][3]] = nresdft * lomask

    # middle bands
    for idx in range(1, len(bound_list)-1):
        bounds1 = (0, 0, 0, 0)
        bounds2 = (0, 0, 0, 0)
        for boundIdx in range(len(bound_list) - 1, idx - 1, -1):
            diff = (bound_list[boundIdx][2]-bound_list[boundIdx][0],
                    bound_list[boundIdx][3]-bound_list[boundIdx][1])
            bound2tmp = bounds2
            bounds2 = (bounds2[0]+bound_list[boundIdx][0],
                       bounds2[1]+bound_list[boundIdx][1],
                       bounds2[0]+bound_list[boundIdx][0] + diff[0],
```

```
                    bounds2[1]+bound_list[boundIdx][1] + diff[1])
            bounds1 = bound2tmp
        nlog_rad1 = log_rad[bounds1[0]:bounds1[2], bounds1[1]:bounds1[3]]
        nlog_rad2 = log_rad[bounds2[0]:bounds2[2], bounds2[1]:bounds2[3]]
        dims = dim_list[idx]
        nangle = angle[bounds1[0]:bounds1[2], bounds1[1]:bounds1[3]]
        YIrcos = np.abs(np.sqrt(1.0 - Yrcos**2))
        if idx > 1:
            Xrcos += np.log2(2.0)
            nlog_rad2_tmp = np.reshape(nlog_rad2, (1, nlog_rad2.
↪shape[0]*nlog_rad2.shape[1]))
            lomask = pointOp(nlog_rad2_tmp, YIrcos, Xrcos[0],
                             Xrcos[1]-Xrcos[0])
            lomask = lomask.reshape(bounds2[2]-bounds2[0],
                                    bounds2[3]-bounds2[1])
            lomask = lomask + 0j
            nresdft = np.zeros(dim_list[idx]) + 0j
            nresdft[bound_list[idx][0]:bound_list[idx][2],
                    bound_list[idx][1]:bound_list[idx][3]] = resdft * lomask
            resdft = nresdft.copy()

        # reconSFpyrLevs
        if idx != 0 and idx != len(bound_list)-1:
            for b in range(num_orientations):
                nlog_rad1_tmp = np.reshape(nlog_rad1,
                                           (1, nlog_rad1.shape[0]*nlog_rad1.
↪shape[1]))
                himask = pointOp(nlog_rad1_tmp, Yrcos, Xrcos[0],␣
↪Xrcos[1]-Xrcos[0])

                himask = himask.reshape(nlog_rad1.shape)
                nangle_tmp = np.reshape(nangle, (1, nangle.shape[0]*nangle.
↪shape[1]))
                anglemask = pointOp(nangle_tmp, Ycosn,
                                    Xcosn[0]+np.pi*b/num_orientations,
                                    Xcosn[1]-Xcosn[0])

                anglemask = anglemask.reshape(nangle.shape)
                # either the coefficients will already be real-valued (if
                # self.is_complex=False) or complex (if self.is_complex=True).␣
↪in the
                # former case, this np.real() does nothing. in the latter, we␣
↪want to only
                # reconstruct with the real portion
                curLev = num_scales-1 - (idx-1)
                band = np.real(pyr_coeffs[(curLev, b)])
```

```
                if (curLev, b) in recon_keys:
                    banddft = np.fft.fftshift(np.fft.fft2(band))
                else:
                    banddft = np.zeros(band.shape)
                resdft += ((np.power(-1+0j, 0.5))**(num_orientations-1) *
                           banddft * anglemask * himask)

    # apply lo0mask
    Xrcos += np.log2(2.0)
    lo0mask = pointOp(log_rad, YIrcos, Xrcos[0], Xrcos[1]-Xrcos[0])

    lo0mask = lo0mask.reshape(dims[0], dims[1])
    resdft = resdft * lo0mask

    # residual highpass subband
    hi0mask = pointOp(log_rad, Yrcos, Xrcos[0], Xrcos[1]-Xrcos[0])

    hi0mask = hi0mask.reshape(resdft.shape[0], resdft.shape[1])
    hidft = np.fft.fftshift(np.fft.fft2(pyr_coeffs['residual_highpass']))
    resdft += hidft * hi0mask
    outresdft = np.real(np.fft.ifft2(np.fft.ifftshift(resdft)))
    return outresdft
```