1. Importing the necessary module:

```
import numpy as np
```

Here, we import the `numpy` module, which provides numerical array manipulation capabilities.

2. Defining the quadratic B-spline basis functions:

```
def quadratic_b_spline(t):
    """
    Compute the quadratic B-spline basis functions
    """
    b0 = (1 - t) ** 2 / 2
    b1 = (2 - 2 * t ** 2) / 2
    b2 = t ** 2 / 2
    return np.array([b0, b1, b2])
```

The `quadratic_b_spline` function takes a parameter `t` and computes the quadratic B-spline basis functions for that value. The basis functions determine the weighting of neighboring pixels in the interpolation process.

3. Implementing the matrix interpolation using quadratic B-spline:

```
def interpolate_matrix(input_matrix, out_shape):
    """
    Interpolate a 2D matrix using quadratic B-spline.
    """
    in_m, in_n = input_matrix.shape
    out_m, out_n = out_shape

    # Compute interpolation indices
    x_indices = np.linspace(0, in_n - 1, out_n)
    y_indices = np.linspace(0, in_m - 1, out_m)

    # Initialize output matrix
    output_matrix = np.zeros((out_m, out_n))

    # Interpolate row by row
    for i, y in enumerate(y_indices):
        y_int = int(y)
        t = y - y_int

        # Get basis functions for row interpolation
        basis_y = quadratic_b_spline(t)

        # Interpolate column by column
        for j, x in enumerate(x_indices):
            x_int = int(x)
            t = x - x_int

            # Get basis functions for column interpolation
            basis_x = quadratic_b_spline(t)

            # Compute interpolated value
            for m in range(3):
                for n in range(3):
                    y_idx = y_int + m - 1
                    x_idx = x_int + n - 1
                    if 0 <= y_idx < in_m and 0 <= x_idx < in_n:
                        output_matrix[i, j] += (
                            basis_y[m] * basis_x[n] * input_matrix[y_idx, x_idx]
                        )

    return output_matrix
```

The `interpolate_matrix` function takes two parameters: `input_matrix`, which is the input 2D matrix to be interpolated, and `out_shape`, which is a tuple specifying the desired output shape of the interpolated matrix.

Inside the function:

- `in_m, in_n = input_matrix.shape` extracts the number of rows and columns of the input matrix.

- `out_m, out_n = out_shape` extracts the desired number of rows and columns of the output matrix.

- `x_indices = np.linspace(0, in_n - 1, out_n)` generates a set of equally spaced indices along the columns of the input matrix, which will be used for interpolation.

- `y_indices = np.linspace(0, in_m - 1, out_m)` generates a set of equally spaced indices along the rows of the input matrix, which will be used for interpolation.

- `output_matrix = np.zeros((out_m, out_n))` initializes an output matrix of zeros with the desired shape.

The interpolation is performed in a nested loop structure:

- The outer loop iterates over the rows of the

output matrix, using `i` as the row index and `y` as the corresponding interpolation index.

- The inner loop iterates over the columns of the output matrix, using `j` as the column index and `x` as the corresponding interpolation index.

- `y_int = int(y)` extracts the integer part of `y`, which represents the row index of the input matrix that is closest to the interpolation index `y`.

- `t = y - y_int` computes the fractional part of `y`, which will be used for determining the weights of the neighboring pixels in the row interpolation.

- `basis_y = quadratic_b_spline(t)` computes the quadratic B-spline basis functions for the fractional part `t`, giving the weights for the neighboring rows.

- Similarly, `x_int = int(x)` and `t = x - x_int` compute the integer and fractional parts of `x`, respectively, for column interpolation.

- `basis_x = quadratic_b_spline(t)` computes the quadratic B-spline basis functions for the fractional part `t`, giving the weights for the neighboring columns.

- The nested `for` loops iterate over a 3x3 neighborhood centered at the integer indices `(y_int, x_int)` in the input matrix.

- The interpolated value at `(i, j)` in the output matrix is computed by summing the products of the corresponding basis function weights and the corresponding input matrix pixel values.

- The `if` statement checks if the indices `(y_idx, x_idx)` are within the valid range of the input matrix to avoid accessing out-of-bounds elements.

- Finally, the computed interpolated value is added to `output_matrix[i, j]`.

After all the iterations, the interpolated matrix `output_matrix` is returned as the result.

4. Testing the interpolation:

```
input_matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
out_shape = (5, 5)  # Desired output shape

interpolated_matrix = interpolate_matrix(input_matrix, out_shape)
print(interpolated_matrix)
```

(interpolated_matrix)

```

This code snippet demonstrates how to use the `interpolate_matrix` function. It creates a simple 3x3 input matrix `input_matrix`, specifies the desired output shape as `(5, 5)`, and calls the `interpolate_matrix` function to obtain the interpolated matrix. Finally, it prints the interpolated matrix to the console.

For Picture

```
from PIL import Image
import numpy as np

# Get the image file path from the user
image_path = input("Enter the path to the image file: ")

# Load the original image
original_image = Image.open(image_path)

#grayscale image
grayscale_img= original_image.convert("L")

input_matrix = np.array(grayscale_img)

# Define desired output shape
out_shape = (10000, 10000)

interpolated_matrix = interpolate_matrix(input_matrix, out_shape)

interpolated_image = Image.fromarray(interpolated_matrix.astype(np.uint8))

interpolated_image.save("output.jpg")
```
```

To use the code for image interpolation, you need to follow these steps:

1. Load the input image using an image processing library such as OpenCV or PIL.

2. Convert the image to a 2D matrix representation. Each pixel value will correspond to an element in the matrix.

3. Determine the desired output shape for the interpolated image. This will depend on the specific requirements of your application.

4. Call the `interpolate_matrix` function with the input matrix and the desired output shape.

5. Convert the interpolated matrix back to an image representation.

6. Save or display the interpolated image.

Here's an example using the Python Imaging Library (PIL) to perform image interpolation:

In this example, we load an input image called "input.jpg", convert it to grayscale, and then convert it to a 2D matrix using `np.array()`. We define the desired output shape as `(800, 800)`, but you can adjust this based on your requirements.

Next, we call the `interpolate_matrix` function with the input matrix and the desired output shape to obtain the interpolated matrix. We then convert the interpolated matrix back to an image representation using `Image.fromarray()`.

Finally, we save the interpolated image as "output.jpg" and optionally display it using `.show()`.