Arthur Liou
NetID: arthurl3
CS445

MP2 - Image Quilting

Report Prompt

Use words and images to show what you've done. Please:

- Compare random, overlapping, and seam-finding methods for one texture sample (similar to Figure 2 of the paper, but with a different texture).
- Create an illustration of the seam finding. Include three images: the two overlapping patches and the cost (squared difference). On top of the cost image, plot the min-cost path. You can edit cut function in utils.py to create this display or to output the path to display.
- Using the above figures as illustrations, briefly explain the image quilting texture synthesis method. Show 4 more results, including at least two from your own images.
- Show at least two results for texture transfer, including at least one result that uses your own images (either for the target or source texture or both). Briefly explain (one or two sentences is fine) how the texture transfer works.
- In your explanations, be sure to include any special design decisions or difficulties that you encounter.
- Describe and bells and whistles under a separate heading.

Part 1: Randomly Sampled Texture

- Create a function quilt_random(sample, out_size, patch_size) that randomly samples square patches of size patch_size from a sample in order to create an output image of size out_size. Start from the upper-left corner, and tile samples until the image are full. If the patches don't fit evenly into the output image, you can leave black borders at the edges. This is the simplest but least effective method. Save a result from a sample image to compare to the next two methods.
- Implementation
    - Open up image matrix array (or Image)
    - Create a results matrix array (or Image)
    - Iteration:
        - Starting from the upper-left corner,
            - generate random patches of size “patch_size” (using the helper function) to create an output image of size “out_size”
                - Note: Implemented a helper function random_patch() that would allow me to take in the sample image and return a randomized patch that can be return output.
            - Paste it in the corresponding “patch size” on the results array
    - Return Array / image
- Results and Parameters/Special Things

400, 80	420, 80
	

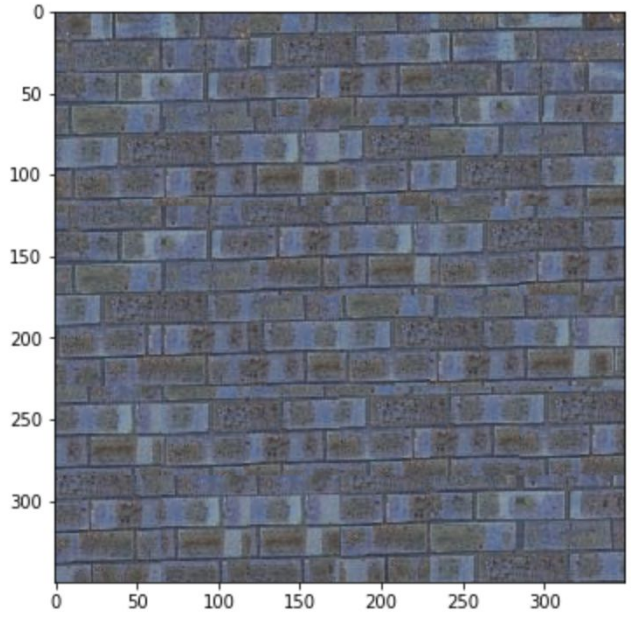
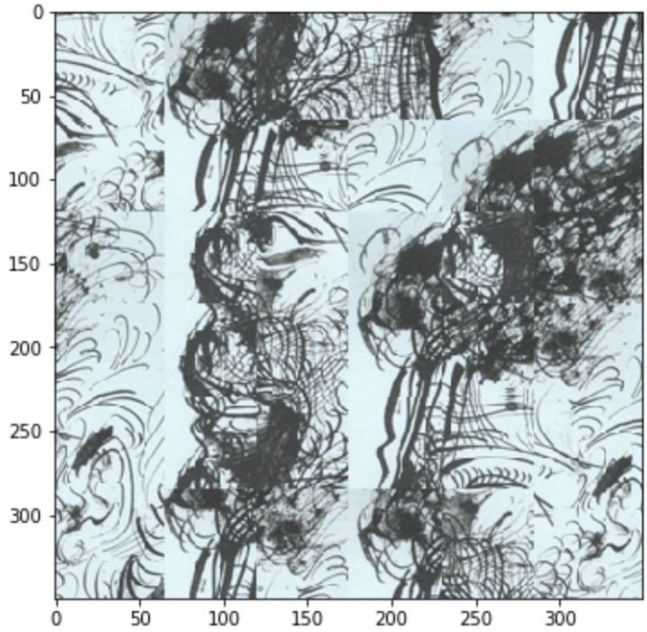
- Parameters
    - I originally started the out_size and 100 and patch_size at 20, tried a 20, 115 where the patches wouldn't fit evenly.
    - The resulting image in the notebook itself was tiny. So I bumped it up to 200, 40, and then 400, 80, and also did a 420/80

Part 2: Overlapping Patches

- Create a function quilt_simple(sample, out_size, patch_size, overlap, tol) that randomly samples square patches of size patch_size from a sample in order to create an output image of size out_size.
    - # a) Generate scan all possible permutations of patches via an AllPatches, store in obj/dict
    - # b) Scan the result with ssd_patch (template matching with the overlapping region, computing the cost of sampling each patch, based on (SSD) of the overlapping regions) to calculate cost
    - # c) Use choose_sample() to run ssd_patch calculations over a random selection from the allPatches() function, build a candidates array with lowest SSD scores, random selection sample from one of the K lowest-cost patches.
    - # d) Use quilt_simple() to iterate through the canvas, grab sample patches that fit the low-cost criteria, and "quilt" ie add the patches in output image via a helper function
        - # also added a helper function, to be used locally within a quilt function, that takes in a number of parameters (x, y, overlap, patch_size, updatedSample, output), and will add the sample patch to

the output / resulting image according to the patch size, overlap, and the starting point of x and y

- Results and Parameters/Special Things

Brick - Simple	Image 2
	

Part 3: Seam Finding

- Update helper functions from Part 2
  - ssd_patch2 - original is not the correct type "invalid index to scalar variable", so required adjustment
  - Choose_sample_with_cost - returns the cost too now
- quilt_cut(sample, out_size, patch_size, overlap, tol): Similar to quilt_simple, but requires additional logic to handle cutting / seams. Use of masked template with the provided cut() function. Depending on where the patch is (ie if it has top and left over flaps), I'd calculate two seams. For vertical paths, I made sure to apply the cut to the transposed patch.
- Results and Parameters/Special Things (5 Results)

Brick - Seam (Find 3x Comparison After)	Image 2 - Toast

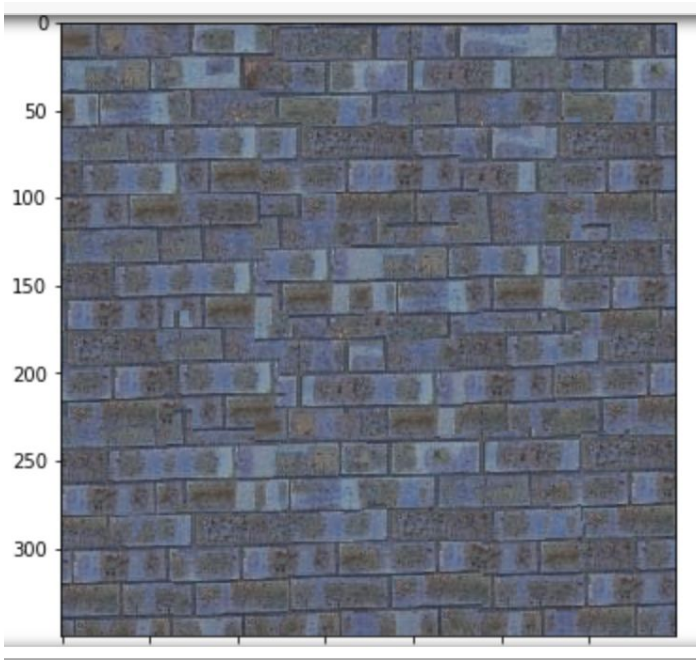


Image 3 - Personal Image 1

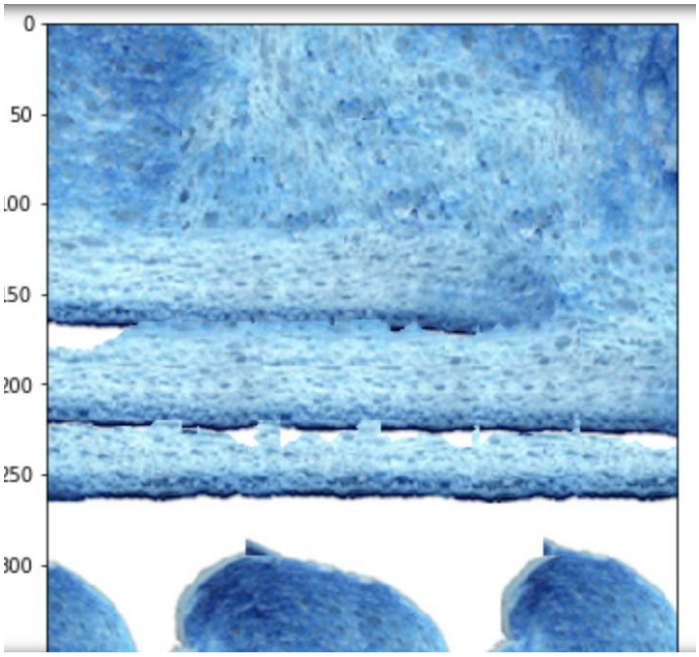


Image 4 - Personal Image 2

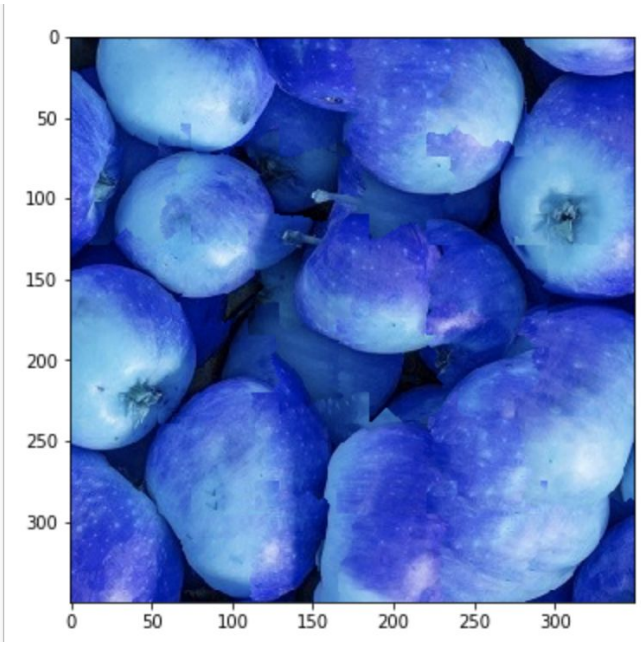
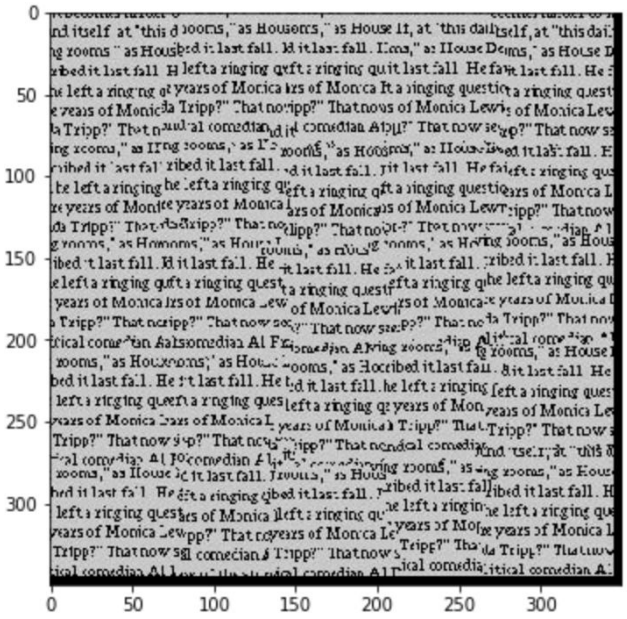


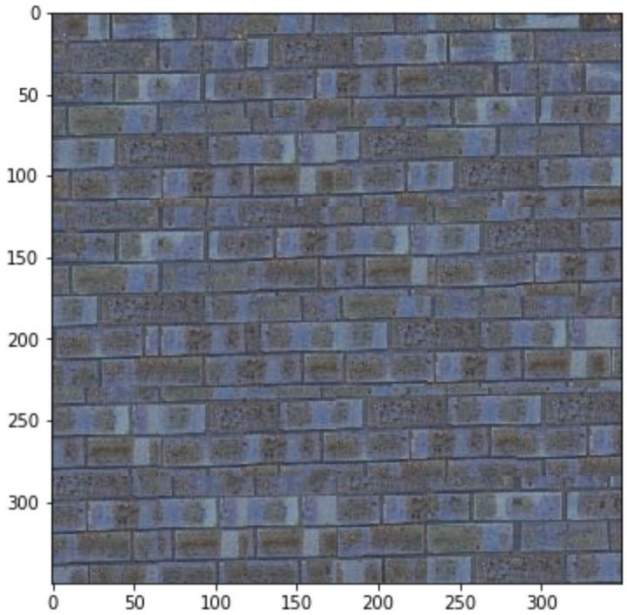
Image 5



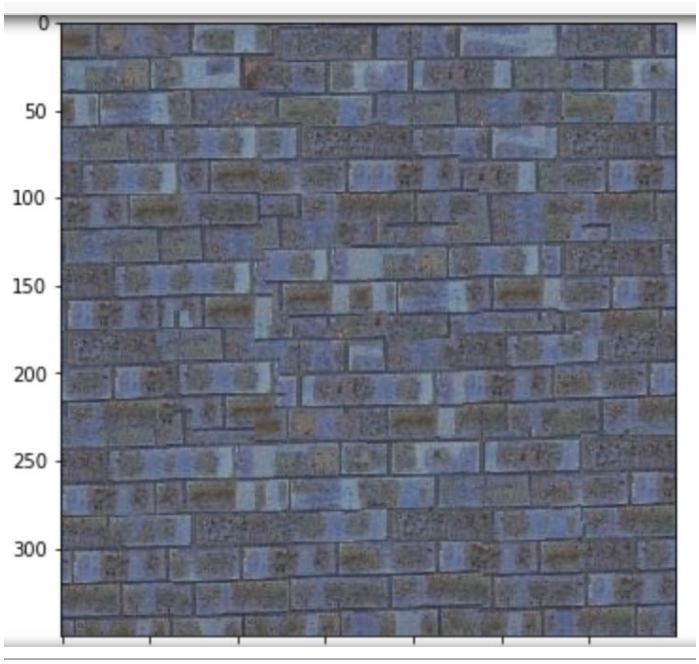
Compare random, overlapping, and seam-finding methods for one texture sample (similar to Figure 2 of the paper, but with a different texture).

1 - Randomly Sampled

2 - Overlapping

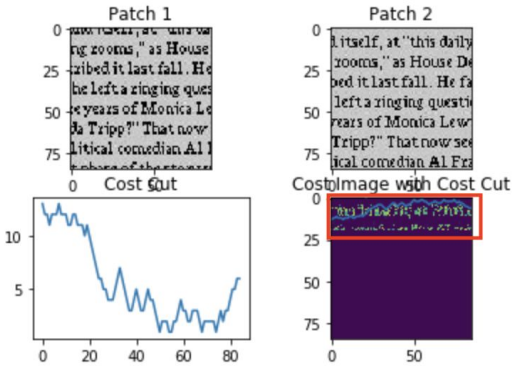


3- Seam Finding



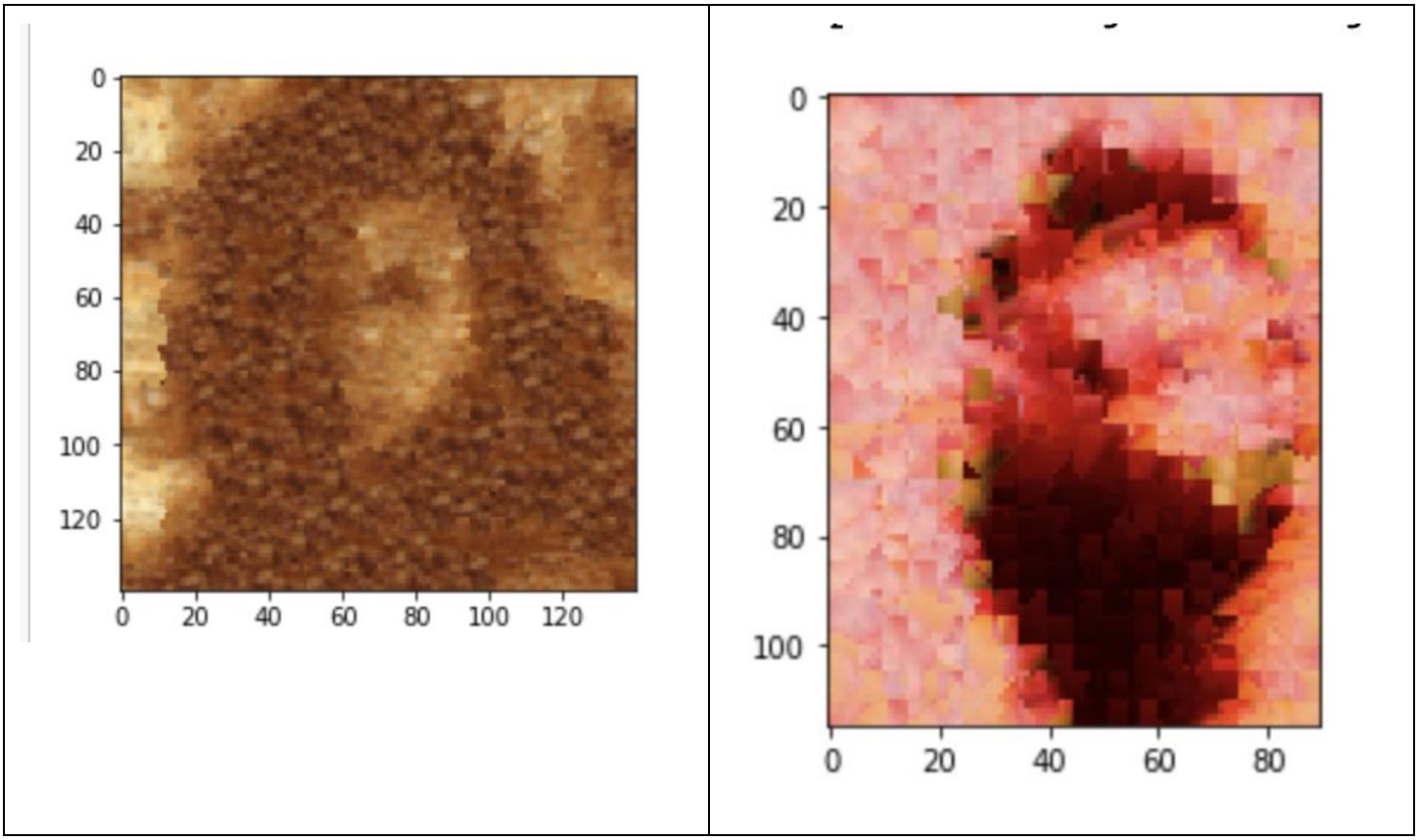
- Special Design decisions or difficulties
  - Difficulty: When using portraits or the toast, I noticed that the quilting process algorithm took significantly longer than some of the provided samples, like the brick and text images. In the case of portraits, multiple minutes. When using one of my own images, I decided to use an aquaman image that I had from Project 1. That took maybe 10 minutes before completing and also consumed heavy resources, slowing my computer down. I realized that these images were not optimal for quilting and were likely high cost, and went with the other images that were much easier to quilt and my quilting functions completed quickly.
  - Illustration / Implementation: I duplicated the cut and my quilt_cut function, renamed them, adapted the logic to return just the best_patch and return two overlapping patches and cost image, which I then plotted.

Illustration Plot



Part 4: Texture Transfer

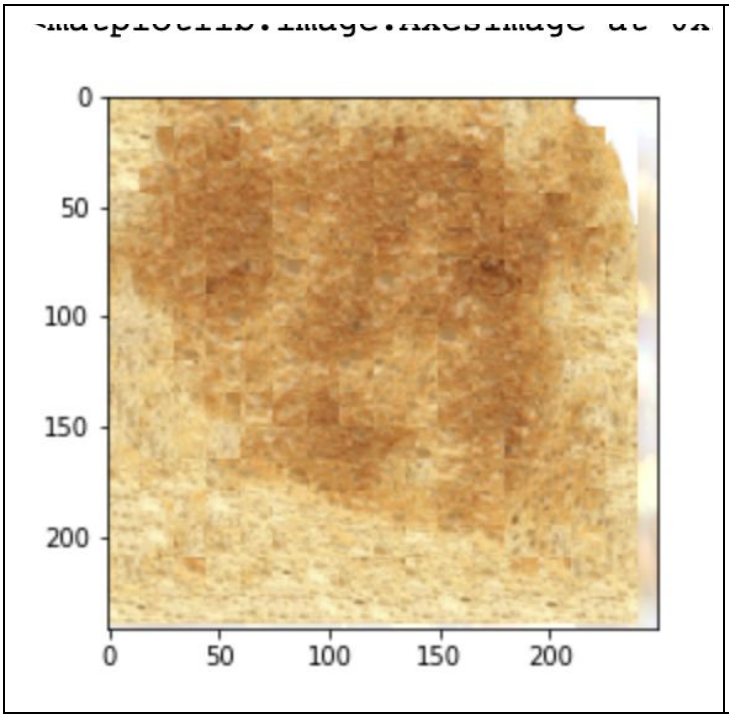
- Similar to Part 3, I copied over my functions from the previous part, updated my helper functions to accommodate the new requirements, ie the main difference of the additional cost term (based on the difference between the sampled source patch and the target patch at the location to be filled), and updated my function parameters and results arrays accordingly as, compared to the prior sections, I'm working with two images, rather than one.
    - For the additional cost term, I utilized the alpha factor from page 5 of the paper: "We modify the error term of the image quilting algorithm to be the weighted sum, times the block overlap matching error plus (1) times the squared error between the correspondence map pixels within the source texture block and those at the current target image position. The parameter determines the tradeoff between the texture synthesis and the fidelity to the target image correspondence map.
- Also before running the function for this part, I made sure the images were appropriately sized. Ran on two sets
- Results and Parameters/Special Things



Bells and Whistles

- (up to 20 pts) Use a combination of texture transfer and blending to create a face-in-toast image like the one on top. To get full points, you must use some type of blending, such as feathering or Laplacian pyramid blending.

- Implementation:
  - Utilized Blending (Gaussaian) from the first project and then did a texture transfer, putting the hybrid from the blending on to the toast (target)



Expected Points Outline

The core assignment is worth 100 points, as follows:

- 10 points for the random patch texture synthesis with one result.
    - **Done**
- 30 points for the overlapping patch texture synthesis with one result.
    - **Done**, along with seam illustration of two patches and the min-cost path plotted on the cost image
- 20 points for the seam finding texture synthesis with five results (including at least two from your own images).
    - **Done**. 5 results total, 2 from own images
- 30 points for texture transfer with at least two results (including at least one from your own images).
    - **Done**, both sets have my images
- 10 points for quality of results (e.g., 0=poor 5=average 10=great)
    - **Great:** The core assignment photos are great, but the B&W photo for the face-in-toast is not great. So likely not a full 20 in B&W due to quality.

- You can also earn up to 75 extra points for the bells & whistles mentioned above. To get full points, you must implement the method, show the requisite results, and explain and display results clearly.
    - (15 pts) Implement the iterative texture transfer method described in the paper. Compare to the non-iterative method for two examples.
    - (up to 20 pts) Use a combination of texture transfer and blending to create a face-in-toast image like the one on top. To get full points, you must use some type of blending, such as feathering or Laplacian pyramid blending.
        - **Implementation and Notes**
            - It took some wrangling but to achieve this, I ran the toast through a low-pass filter and used the face as the high-pass during my blending.
            - I passed the resulting hybrid image to the texture transfer function from Part 43, using the original toast image as the texture, and the hybrid as the target
    - (up to 40 pts) Extend your method to fill holes of arbitrary shape for image completion. In this case, patches are drawn from other parts of the target image. For the full 40 pts, you should implement a smart priority function (e.g., similar to Criminisi et al.).

Regular: 10+30+20+30+10 = 100
Bells and Whistles: Up to 20

Sources and Additional Notes

- Useful Python functions: cv2.filter2D, matplotlib.pyplot.plot, numpy.where, numpy.ndarray.sum(), numpy.square

- For efficiency, use filtering to compute SSD.
    - Note that part of the computation only needs to be done once (not for each patch), so that can be cached.
    - Suppose I have a template T, a mask M, and an image I: then, ssd = ((M*T)**2).sum() - 2 * cv2.filter2D(I, ddepth=-1, kernel = M*T) + cv2.filter2D(I ** 2, ddepth=-1, kernel=M)
- https://www.geeksforgeeks.org/working-images-python/. use open image library from path, crop a patch, paste the patch
- https://code-maven.com/create-images-with-python-pil-pillow -> Results array
- https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.astype.html
- https://docs.scipy.org/doc/numpy/reference/arrays.ndarray.html
- https://www.geeksforgeeks.org/change-data-type-of-given-numpy-array/
- https://stackoverflow.com/questions/26649716/how-to-show-pil-image-in-ipython-notebook
- # Tip: For efficiency, use filtering to compute SSD.
- # Note that part of the computation only needs to be done once (not for each patch), so that can be cached.
- # Suppose I have a template T, a mask M, and an image I: then, ssd = ((M*T)**2).sum() - 2 * cv2.filter2D(I, ddepth=-1, kernel = M*T) + cv2.filter2D(I ** 2, ddepth=-1, kernel=M)
- # def ssd_patch(T, M, I):
- # ssd = ((M*T)**2).sum() - 2 * cv2.filter2D(I, ddepth=-1, kernel = M*T) + cv2.filter2D(I ** 2, ddepth=-1, kernel=M)
- # return ssd