CGRA352 2020 Assignment 2

Advanced Image Manipulation Tools

30 marks

Released: 26/03/2020

Due Date: 23/05/2020 11:59pm

Core (14 marks)

PatchMatch implementation

PatchMatch is a method for quickly finding dense correspondences between "patches" or small square regions in a pair of images. It is also in some features of Photoshop such as Content Aware Fill. It is significantly (10-100x) faster than previous techniques, so it has allowed some high quality image editing applications to run interactively for the first time.

Your assignment is to implement the core matching algorithm for PatchMatch. PatchMatch finds correspondences between densely overlapping patches, where a patch is defined to be around every pixel. Patches have a small fixed size such as 7x7. The algorithm attempts to find, for each patch in the "target" image A, what is the most similar patch in the "source" image B, under all possible (integer) translations. The algorithm is a fast approximation algorithm, which may not find the best matching patch, but finds an approximate solution quickly. It does this by optimizing an array containing the correspondence mapping from image A to image B, in three phases: initialization, propagation, and search.

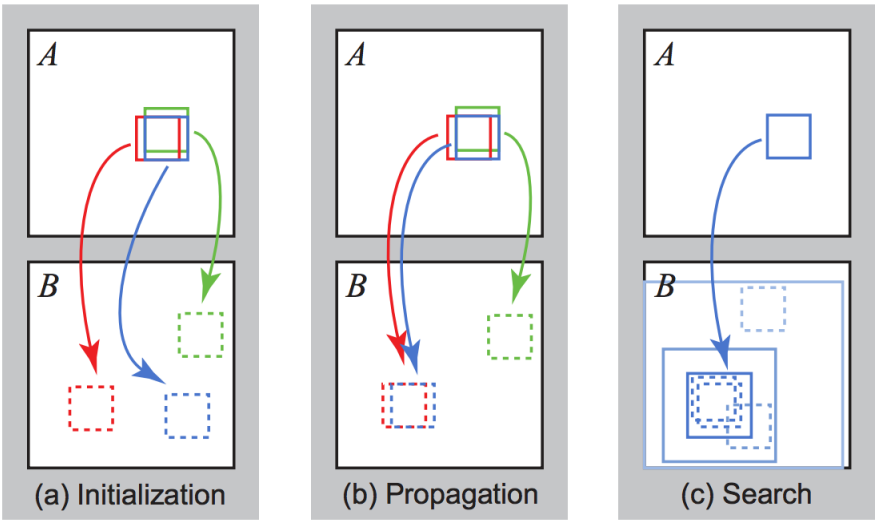


Figure From [Barnes 2009]

In this assignment, you will implement the PatchMatch algorithm to find the “approximate nearest neighbor,” and reconstruct the given Target Image "Target.jpg" using the pixels from the given Source Image "Source.jpg", just like in the original paper [Barnes 2009][1].

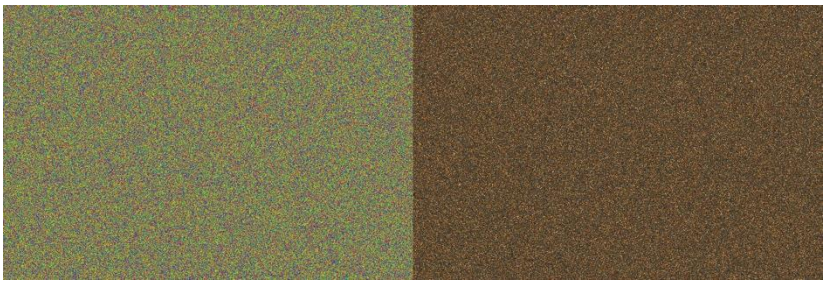
(a) originals (b) random (c) $\frac{1}{4}$ iteration (d) $\frac{3}{4}$ iteration (e) 1 iteration (f) 2 iterations (g) 5 iterations

Your program should include the following parts of the PatchMatch Algorithm to reconstruct the target image using the pixels from the source image:

1. **(3 marks) Initialization**.
   Implement the "Offset" vector for each pixel using a multi-channel matrix, which is called the Nearest Neighborhood Field (NNF) in the original paper. This NNF map f should be the same size as the target image. Store in the NNF map `F` the relative offset vector `F(Tx,Ty) = (Sx-Tx, Sy-Ty)`, which maps a pixel from the target image `(Tx, Ty)` to a pixel in the source image `(Sx, Sy)`.

   We recommend that you store the NNF as a 2-channel signed-integer array `CV_32SC2` where the first and second channel store the row and column offsets, and have a separate 1-channel floating point array for storing the patch distance/cost `D`. Initialize the NNF using "reasonable" random values where the referred position represented by each offset value should be in the valid region [0, height) and [0, width) of the source image.

   Hint: An alternative to using offset is to use absolute coordinates, with the notation, `F(ax,ay) = (bx, by)`. Bounds checking is easier with absolute coordinates. However, propagation with relative coordinates is easier because it just involves trying the same offset vector as your adjacent patches, whereas for absolute coordinates it requires an additional shift by +1 pixel (it is best to draw on paper if you are confused).



(left) result of random initialization and (right) the reconstruction.

[1] Barnes, 2009: “PatchMatch: A randomized correspondence algorithm for structural image editing”, Connelly Barnes et al., *ACM Transactions on Graphics* **28**(9), Article No. 24, 2009
https://gfx.cs.princeton.edu/pubs/Barnes_2009_PAR/patchmatch.pdf

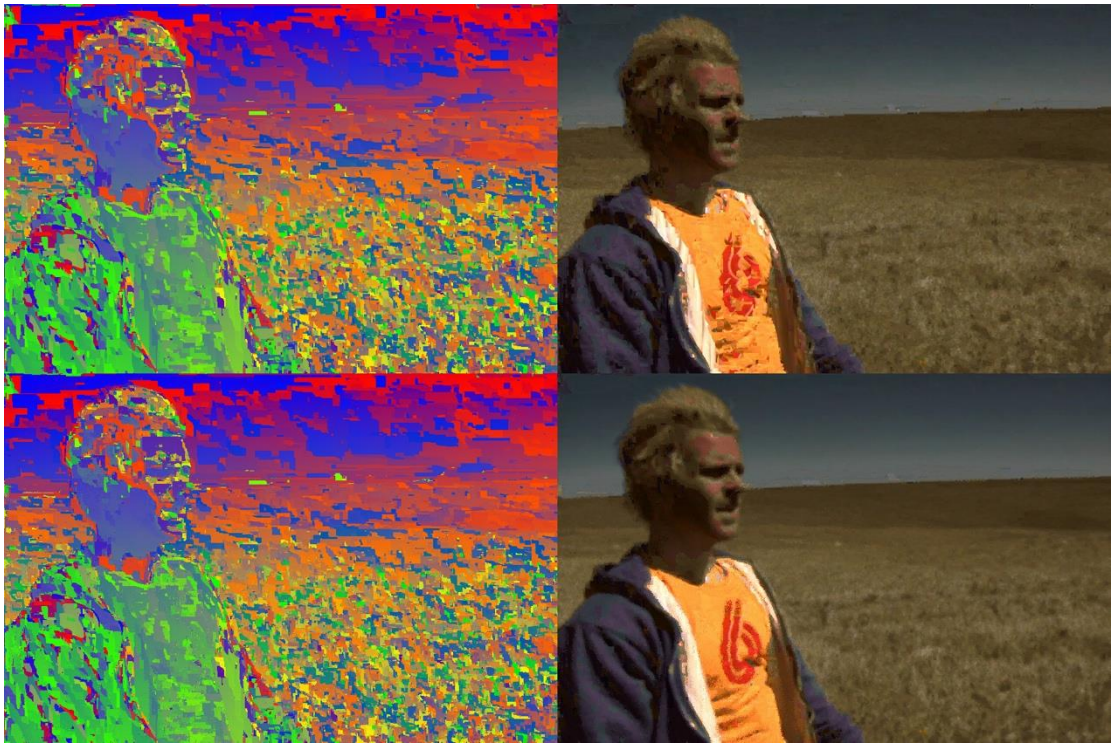
2. **(3 marks) Random Search**.
   To improve the NNF without doing an exhaustive search we use a random search to help globally optimize the solution. An offset candidate is randomly selected within a concentric radius around the current offset. If the random offset is a better match, it becomes the current offset. The search radius starts with the size of the image and is halved each time until it is equal to or less than 1, where the random search stops.

   Hint 1: To measure whether the new offset is better, we should compare the patch distance D. It can be an arbitrary function but the most reasonable choice is to sum up the squared differences (SSD) between corresponding pixels in the patch.

   Hint 2: We recommend you implement a method `improveNNF` which takes the Source and Target images, the patch coordinate `(Tx, Ty)` and a proposed source reference patch coordinate `(Sx,Sy)`, the NNF `F`, and the Distance/Cost matrix `D`. It would modify the NNF (and Distance/Cost matrix) if the target patch coordinate has a smaller patch distance (SSD) than the one currently stored in the Distance/Cost matrix. With this method it is then straightforward to implement the random search and the propagation in the following part.

Patch size = 7, (left) result of only random search and (right) the reconstruction.

3. **(5 marks) Propagation.**
   Improve the NNF `f(x, y)` using the neighboring offsets of `f(x-1, y)` and `f(x, y-1)`. Because it is unlikely that there are significant differences locally, if the neighboring pixel has a good match, it is likely that the same offset would also be a good match for our pixel. For example, if there is a good mapping at `(x-1, y)`, we try to use the translation of that mapping one pixel to the right for our mapping at `(x, y)`. Please refer to the lecture notes or the original paper for more details of this operation.



Patch size = 7, (top-left) result of only propagation and (top-right) the reconstruction, (bottom-left) result of random search + propagation and (bottom-right) the reconstruction.

4. **(2 marks) Iteration.**

   After initialization, we use the implemented Propagation and Random Search parts of the algorithm to perform an iterative process of improving the NNF. Each iteration of the algorithm proceeds as follows: Offsets are examined in scan order (from left to right, top to bottom), and each undergoes propagation followed by random search. These operations are interleaved at the patch level: if `Pj` and `Sj` denote, respectively, propagation and random search at patch `j`, then we proceed in the order: `P0,S0,P1,S2,...,Pn,Sn`.

   Moreover, on odd iterations we must propagate information up and left by examining offsets in reverse scan order, using `f(x+1, y)` and `f(x, y+1)` as our candidate offsets.

   Hint: For this application, 2-5 iterations of the algorithm should suffice.

Patch size = 7, iterations = 4 (left) result of finished algorithm and (right) the reconstruction.

5. **(2 marks) Reconstruction.**
   Using the final NNF, reconstruct the target image using the pixels from the source image. You just need to set a single pixel value in target by the pixel in the source image, which is referred by the offset vector stored in the final NNF. Then you can see whether PatchMatch has retrieved the correct pixels in the source image.

Completion (8 marks)

Image Quilting

Image quilting is a simple technique for creating a larger textured image from a smaller input. The idea of the algorithm is to sample square regions or patches from the input in such a way that they align like a jigsaw puzzle.

Implement the core part of the algorithm:

1. Randomly choose a 100x100 pixel-sized patch as the beginning of the synthesis.
2. Find a patch to overlap the current synthesis by 20 pixels on its right. The patch should minimize the difference in the overlapping region using SSD.
3. Use dynamic programming to find the best seam from the top edge to the bottom edge.
4. Repeat steps 2-3 until the synthesis is 500 pixels wide (5 iterations).

*An example of horizontal image quilting. Notice the seam on the far left is a perfect match. This is unlikely to occur for **every** seam when the synthesis is larger than the sample image.*

Hint: When finding a patch that overlaps with its already-placed neighbor you can either exhaustively sample all possible source patches or just randomly sample a subset of them. The best seam means along that seam, the overall color difference between the corresponding pixels from the two patches is minimal.

Challenge (5 marks)

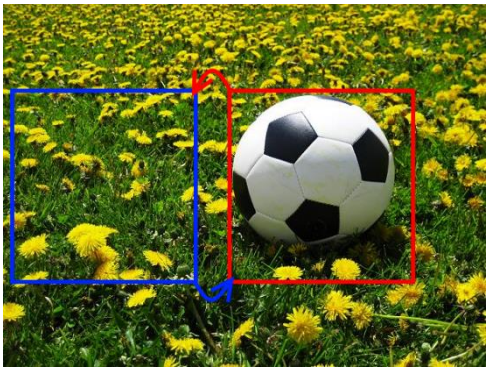
PatchMatch-based Image Reshuffling

Using the implemented PatchMatch core algorithm, write a program that can reshuffle the image content.

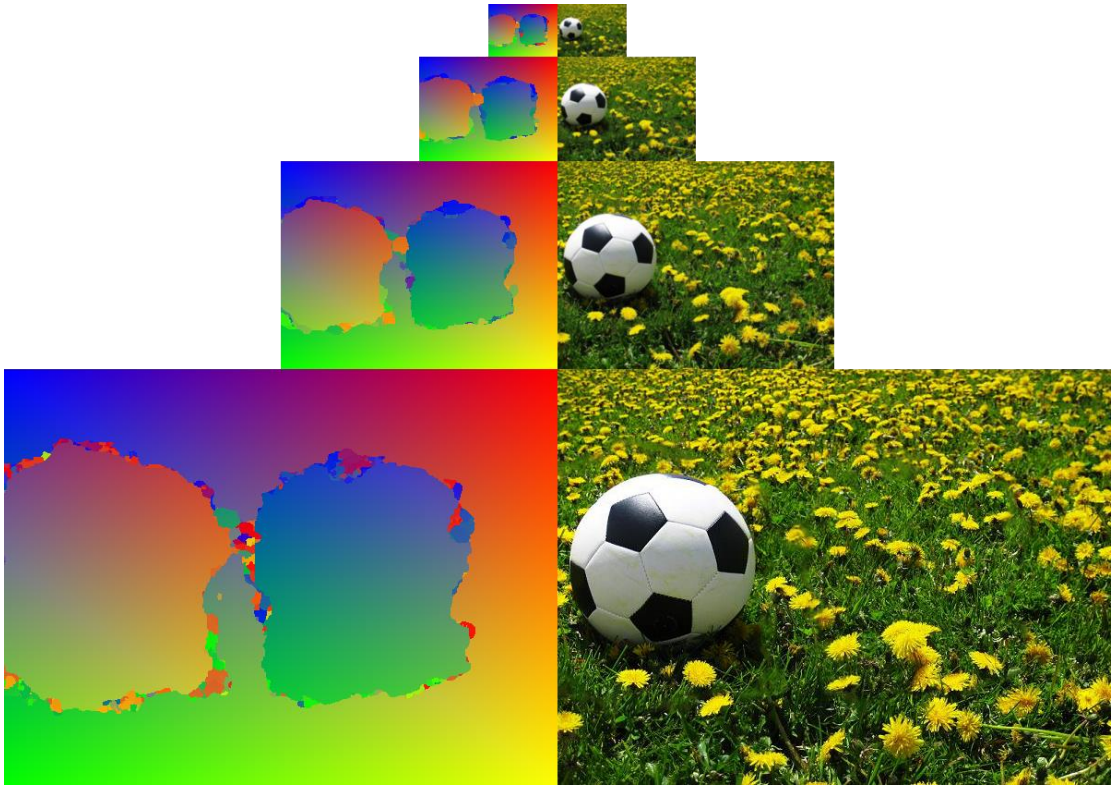
Implement the following steps:

1. Initialize the target with changes made using constraints given by users.
2. Generate a k-level Gaussian pyramid for the source and target.
3. Use PatchMatch to generate a new NNF at the coarsest level.
4. Iterate the following step:
    a. Reconstruct the image the image using the current NNF.
    b. Use PatchMatch to generate a new NNF.
    c. If number of Iterations > n, stop iteration.
5. Up-sample the target NNF and go back to Step 4, unless it is the finest pyramid level.
6. Reconstruct the final image using the last NNF.

Use your program to move the region represented by the binary mask ReshuffleMask.jpg at the source image ReshuffleSource.jpg by a translation of (-270, 0), which is 270 pixels to the left. For the remaining "hole" in the target image after you move this part to the left, you can initialize the NNF for those parts by pointing to the original pixels at the target position. It is kind of like "swapping" the patches as the initialization. Then you should generate a final optimized reshuffling result by the above program.



Hint: Unlike the core algorithm, you will have to implement a different reconstruction algorithm where instead of using a single pixel in the middle of a patch to reconstruct a pixel, every patch should contribute to the pixels it overlaps. Essentially for every pixel you will take the patch NNF and accumulate the entire 7x7 patch (or another appropriate size) in the reconstructed image, then divide by the number of contributing patches to get an average pixel value. Without this change to the reconstruction, the image will produce major artifacts during reshuffling.



A visualization of each step of the pyramid. On the left are the final NNF's and on the right are the final reconstructions for each level. Notice how the NNF changes during the optimization at each level to give a slightly better result.

Report (3 marks)

You should submit a 1-2 page PDF document reflecting your experience. In this report, you should include:

- Brief introduction of your functions in your programs.
- How to run your program to perform the functions required by the assignment.
- The results of core, completion and challenge (depending on how much you have done).

Images do not count towards the 2 page limit.

Practical matters

You are required to submit both your program and any supplementary material including the report. You may complete the assignment in a single or multiple programs, as long as it meets the assignment specification. Please zip your program(s) source and other material into a single compressed file to upload. You are required to show your program can run on the ECS machine OR your own computer.

Helpful code

```
// convert offset coordinates to color image
// assumes nnf stores a Matrix of Vec2i (i, j)
// absolute positions into the source of size s
cv::Mat nnf2img(cv::Mat nnf, cv::Mat s) {
	cv::Mat nnf_img(nnf.rows, nnf.cols, CV_8UC3, cv::Scalar(0, 0, 0));
	for (int i = 0; i < nnf.rows; ++i) {
		for (int j = 0; j < nnf.cols; ++j) {
			Vec2i p = nnf.at<Vec2i>(i, j);
			if (p[0]<0 || p[1]<0 || p[0]>=s.rows || p[1]>=s.cols) {
				/* coordinate is outside, insert error of choice */
			}
			int r = int(p[1] * 255.0 / s.cols);  // cols -> red
			int g = int(p[0] * 255.0 / s.rows); // rows -> green
			int b = 255 - max(r, g); // blue
			nnf_img.at<Vec3b>(i, j) = Vec3b(b, g, r);
		}
	}
	return nnf_img;
}
```