Texture Synthesis by non-parametric sampling

Guttu Sai Abhishek : 180050036
Mulinti Shaik Wajid : 180050063

CS736 Course Project, 2020, IIT Bombay

Note

The algorithm we have used is from the paper Texture Synthesis by non parametric sampling authored by A.A. Efros and T.K. Leung. For more details about this method and other recent methods visit this page.

What is a texture and what is texture-synthesis ?

- Texture can be defined as a 2D visual pattern which, at some scale, has a stationary distribution.
- Given a finite sample from some texture (an image), the goal of texture synthesis is to synthesize other samples from the same texture.
- Without additional assumptions this problem is clearly ill-posed since a given texture sample could have been drawn from an infinite number of different textures.
- The usual assumption is that the sample is large enough that it somehow captures the stationarity of the texture and that the (approximate) scale of the texture elements (texels) is known.

The problem : Texture synthesis

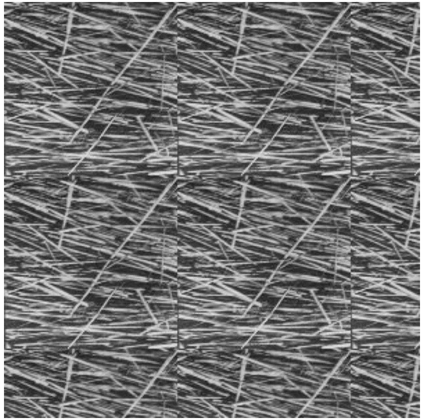
Given a large enough sample I_{input} from a texture we should create a texture I_{synth} which satisfies the following requirements

1) The size of output texture must be user specified i-e the algorithm must be able to produce arbitrary sized textures.
2) I_{synth} must be visually similar to I_{input}.
3) I_{synth} must not contain visible artifacts such as mismatched boundaries.

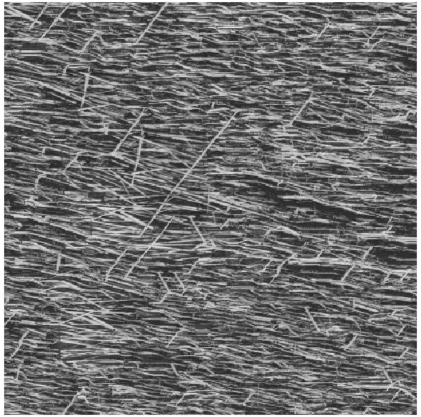
The problem : Texture Synthesis

Tiling a image is not a solution because even though it works for some textures for a majority of textures tiling leads to many artifacts in synthesized image as shown below

Small portion of texture



Result of tiling. Boundaries are mismatched, texture looks repetitive



Result of efros leung algorithm using window Size of 15

Solution Strategy

- Textures are assumed to satisfy Markov random field model.
- To start the process a small patch from the given texture is taken as a seed.
- Now we grow the texture outwards using the seed one pixel at a time.
- To find the intensity of the pixel we query the given texture to find all similar neighborhoods to the neighborhood of current pixel in given texture.
- The neighbourhood of a pixel is taken as a square window around that pixel.
- The similarity between neighborhoods is calculated using L2 norm.
- A random pick is taken from all similar neighborhoods and the intensity of current pixel is found.
- The process continues again by choosing another unfilled pixel until the size of the texture matches the user specified size.

Some issues, handled

- While calculating similar neighborhoods, the neighborhood of unfilled pixel is likely to contain other unfilled pixels which becomes a issue while calculating the similarity of two neighborhoods.
- In this algorithm such pixels are ignored and the resulting L2 norm is divided by the number of pixels using which distance between neighborhoods is calculated.
- To minimize the number of pixels which are ignored, at each step we choose the unfilled pixel (which needs to be filled) that has maximum number of filled neighbors.
- L2 norm gives equal weightage to mismatched pixels whether they are present at the edge of the boundary or near the center, to overcome this we multiply the L2 norm with 2D Gaussian kernel.

The algorithm

Input : A large enough sample from a texture, size of output texture, size of neighborhood window.

Output : A texture which satisfies earlier mentioned conditions.

Note :

The size of neighborhood window is a parameter that specifies how stochastic the user believes the texture to be. More specifically, if the texture is presumed to be mainly regular at high spatial frequencies and mainly stochastic at low spatial frequencies, the size of the window should be on the scale of biggest regular feature.

MATLAB like Pseudo Code

```
generated = image with final rows, final columns, filled with a initial 3*3 seed randomly chosen from sample image
filled = boolean matrix to maintain a track of the pixels filled in generated matrix
stack = matrix of size window_size*number_of_pixels_in_sample storing the window information of each pixel in sample
while generated_is_not_completely_filled
    flag = 0
    neighbours_of_unfilled = find_number_of_filled_neighbours_of_unfilled(generated)
    sort(neighbours_of_unfilled,'descend')
    for element in neighbours_of_unfilled
          if element < 0
                break
          end
          row, column = find_row_and_column_column_in_generated(element)
          neighbourhood = elements_in_the_window_centered_at_element_in_generated(row, column)
          neighbourhood_filled =corresponding_filled_information_of_neighbourhood(row, column)
          norm_factor = sum(gaussian.*neighbouhood_filled)
         distances =sum((((stack-neighbourhood).*neighbourhood_filled).^2).*gaussian)/norm_factor
```

Pseudo Code - Continued

```
        min_distance = min(distances)
        valid = distances(distances < (1+epsilon)*min_distance)
        random_pick = random_pick(valid)
        if distances(random_pick) < max_distance
            generated(row, column) = sample(random_pick’s row, column)
            filled(row, column) = 1
            flag = 1
        end
    end
    if flag < 1
        max_distance = 1.1*max_distance
    end
end
```

Implementation details

- To initialize the the process a random 3 X 3 matrix is chosen as a seed from the given sample texture.
- To simplify the process of calculating distances between neighborhoods the image matrices are padded on all 4 sides with zeros.
- Epsilon is chosen to be 0.1 and max_distance is chosen to be 0.3.
- Two neighborhoods are considered to be similar if the distance between them is less than 1.1 times the distance to the “most similar” neighborhood.
- In line 27 if we fail to fill any pixel in current iteration we multiply max_distance by 1.1, so it is good to start with a low value of max_distance as the algorithm increases it by itself if required.

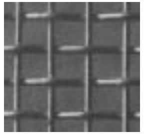
Implementation details

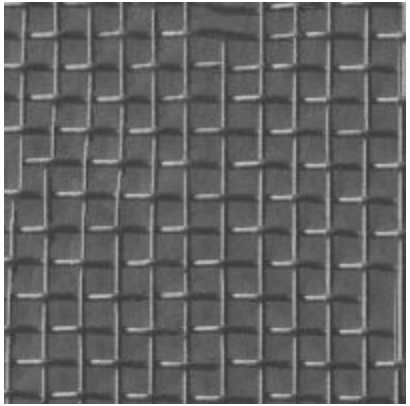
- As mentioned in slide 7 we are dividing the distances with norm_factors in line number 16.
- For hole filling case user is required to select a pixel belonging to a hole and then the algorithm assumes that all zero intensity pixels reachable from selected pixel belong to hole. So, it is expected to have no black pixels at boundary of hole.

MATLAB specific details

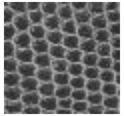
- To calculate stack matrix in line 3, im2col function is used.
- In line 6 to find number of filled neighbors of a pixel conv2 function is used.

Results (sample textures)

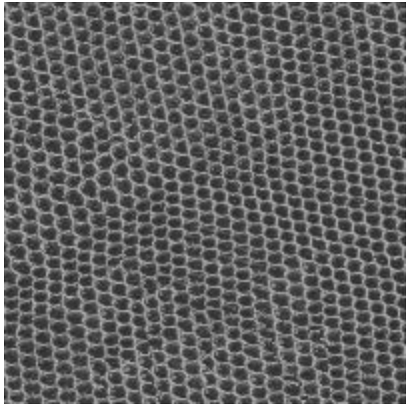


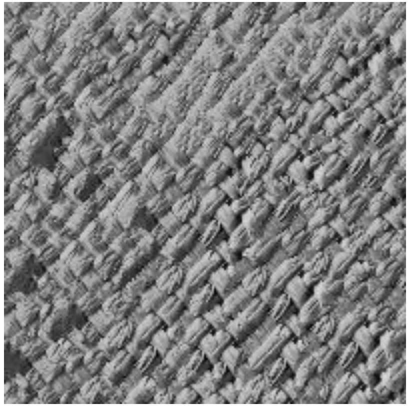


Window_size = 25

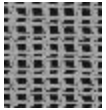


Window_size = 25

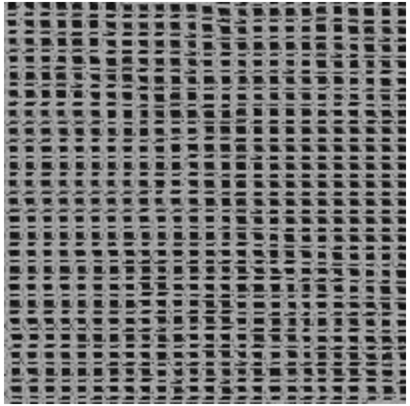




Window_size = 21



Window_size = 21

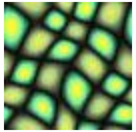


Results (sample textures)

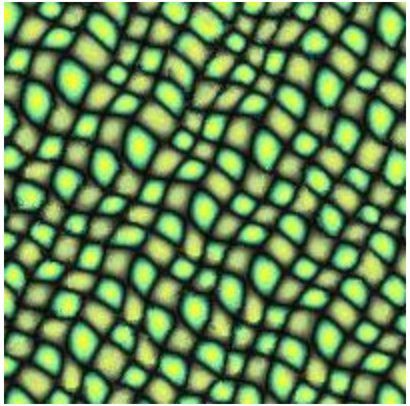
Window_size = 13

Window_size = 13

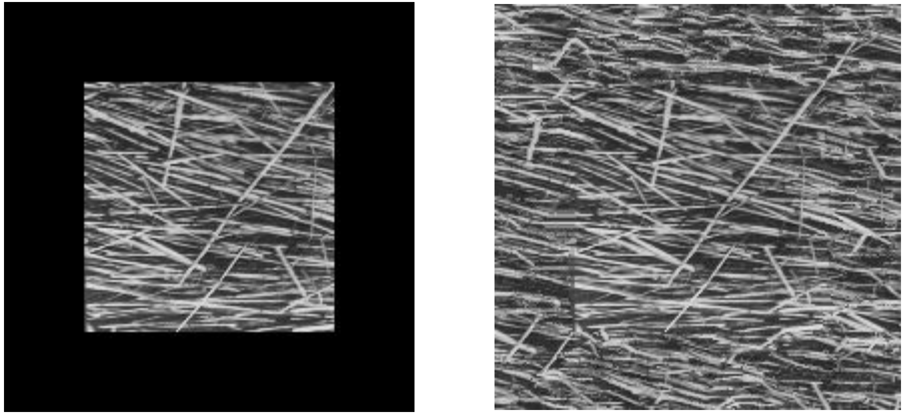
Window_size = 21



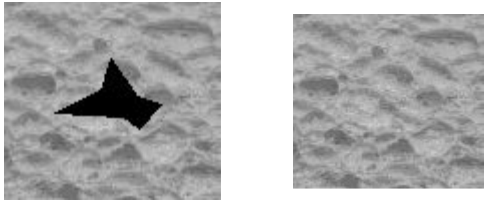
Window_size = 21



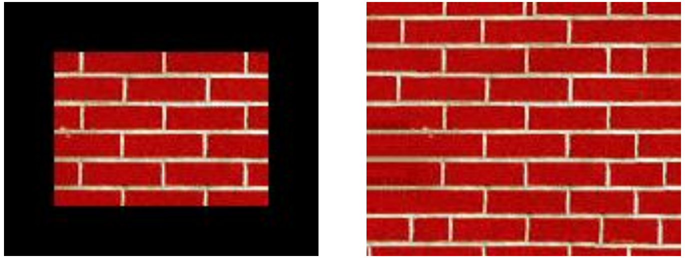
Results (Hole filling):



Window_size = 23

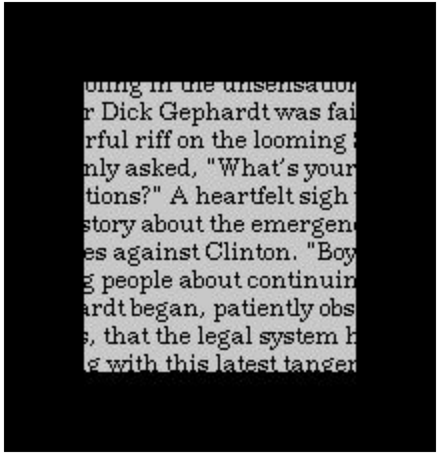


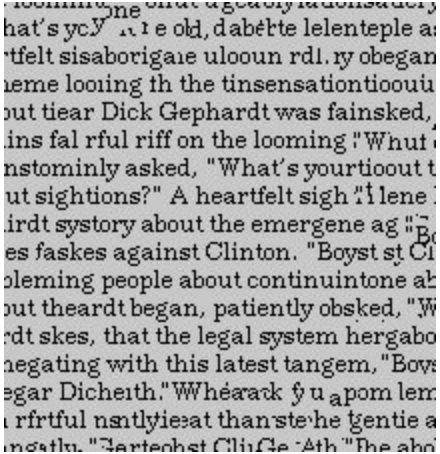
Window_size = 21

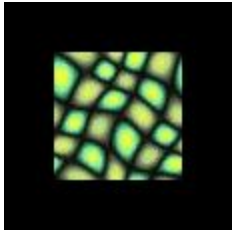


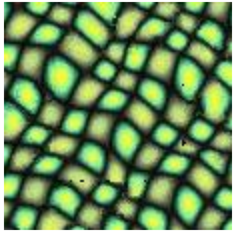
Window_size = 27

Results (Hole filling):









Window_size = 27

Window_size = 21

Window_size = 27

Results (variation with window_size)

sample

window_size = 5

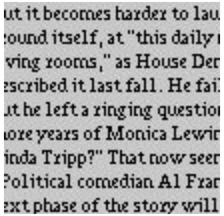
window_size = 9

window_size = 15

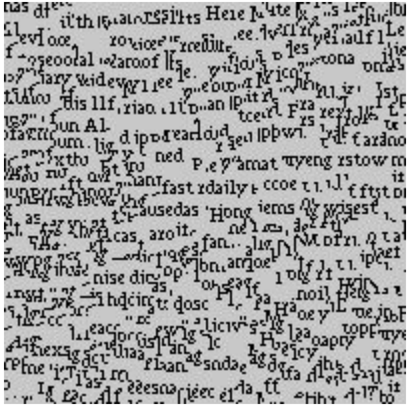
window_size = 21

The brick is the largest regular feature and its size is approximately 35 pixels, so window_size must be at least 30. As seen in the above figures the texture improved when window_size is increased, due to the time involved in computation we were unable to increase window_size further.

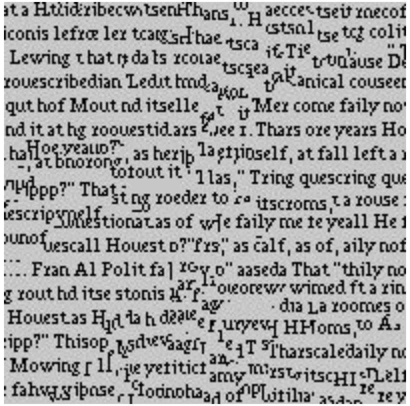
Results (variation with window_size)



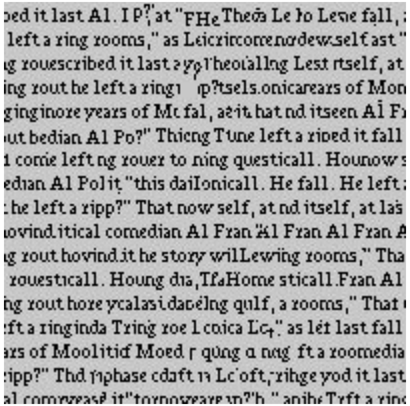
Sample



window_size = 9



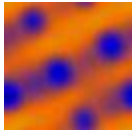
window_size = 15



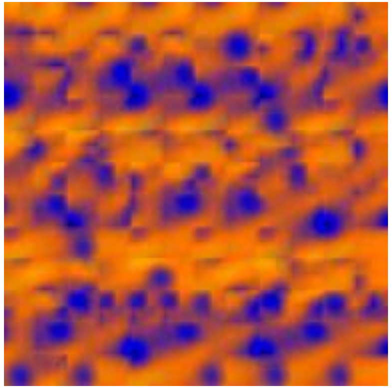
window_size = 23

The largest size of texel in sample is size of largest word which is approximately equal to 25. So when window_size is less than 25 the texture is not synthesized properly, but when window_size = 23 we can see that the texture synthesized is very similar to sample.

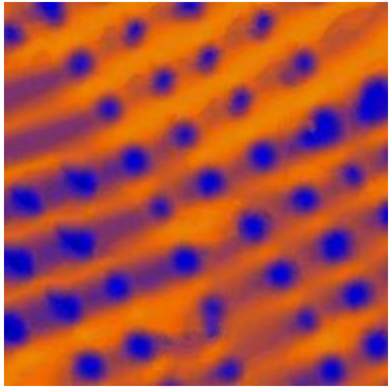
Results (Comparison with De Bonet's method)



Sample



De Bonets Method, can be found here

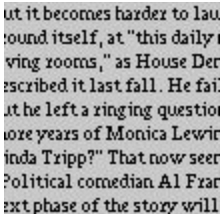


Efros-Leung, window_size is 21

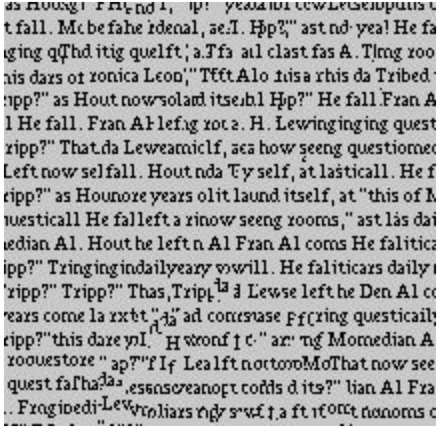
De Bonet's is a older method than Efros-Leung method and we can see clearly that efros-leung method performs much better than De Bonets method.

De Bonets method is based on the assumption that at low resolutions textures have some identically perceived regions and re arranging those regions does not change the perceived texture of the image but changes the visual structure of the image in other words by rearranging we get images which are different but belong to same texture.

Results (Comparison with Image quilting method)



Sample



Efros-Leung, window_size=15

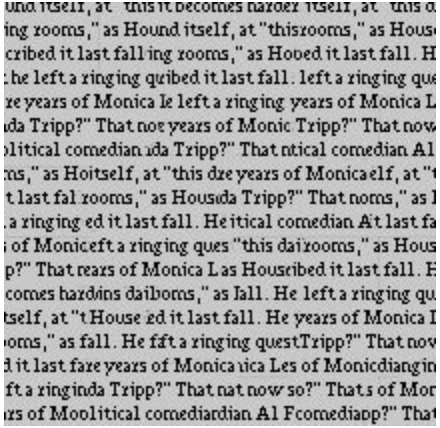


Image Quilting

Image quilting takes small square patches as basic element instead of pixels. It overlaps the patches and finds the optimal boundary separating two overlapped patches and cuts and joins the two patches along that optimal boundary.

Image quilting is a more modern technique in the field of texture synthesis, the produced images are similar or better than efros-leung method and the time taken is also less. More about image quilting here.

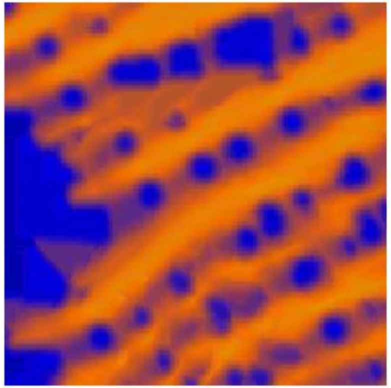
Conclusion - Applications

Applications

- Hole-Filling method can be used for occlusion fill in.
- Using image segmentation to find boundaries of regions and storing a small patch from each region, we can use texture synthesis to recreate entire image. This can be used as a lossy compression technique.
- A common problem while performing convolution is boundary handling. Using zero-filling, tiling, reflection may introduce discontinuities not present in original image. To avoid this problem, hole-filling using Efros-Leung method can be performed and the values obtained may be used for calculating convolution.

Conclusion - Limitations

- Once a pixel is assigned a wrong value by the algorithm then it is likely that the assigned values for unfilled neighbors of present pixel would also be wrong because the value of present pixel is used in determining the value of it's unfilled neighbors. An example is given below. This problem can be prevented by using large window_size, but then the algorithm take more time.



In the figure shown in left, window_size of 9 is used, the algorithm started accumulating garbage values near the left end. To fix the problem window_size must be increased, the result with window_size 21 is shown in slide number 19.

THANK YOU