Hin Chan
913645769

I.

1. Suppose we have image $f = \begin{bmatrix} 4 & 2 & 6 \\ 6 & 4 & 2 \\ 2 & 6 & 4 \end{bmatrix}$ and filter $g = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ and $h = [3 \quad 2 \quad 1]$.

$$(f * g) * h = \left(\begin{bmatrix} 4 & 2 & 6 \\ 6 & 4 & 2 \\ 2 & 6 & 4 \end{bmatrix} * \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \right) * [3 \quad 2 \quad 1] = [22 \quad 28 \quad 22] * [3 \quad 2 \quad 1]$$

$$f * (g * h) = \begin{bmatrix} 4 & 2 & 6 \\ 6 & 4 & 2 \\ 2 & 6 & 4 \end{bmatrix} * \left(\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} * [3 \quad 2 \quad 1] \right) = \begin{bmatrix} 4 & 2 & 6 \\ 6 & 4 & 2 \\ 2 & 6 & 4 \end{bmatrix} * \begin{bmatrix} 3 & 2 & 1 \\ 6 & 4 & 2 \\ 9 & 6 & 3 \end{bmatrix}$$

As we can see, $(f * g) * h$ will end up with a 1x3 matrix * 1x3 matrix. Whereas, $f * (g * h)$ will end up with a 3x3 matrix * 3x3 matrix. Since it takes less operation to multiply a 1x3 matrix with a 1x3 matrix, $(f * g) * h$ will more efficiently filter an image than its associative property equivalent $f * (g * h)$. As the size of the matrix gets bigger, we will be able to save a lot more operations.

2. The dilated picture will be $[1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1]$

3. Representing image noise using additive Gaussian noise will blocks out high frequencies. If the high frequencies are supposed to be a feature in the image, it will get even out.

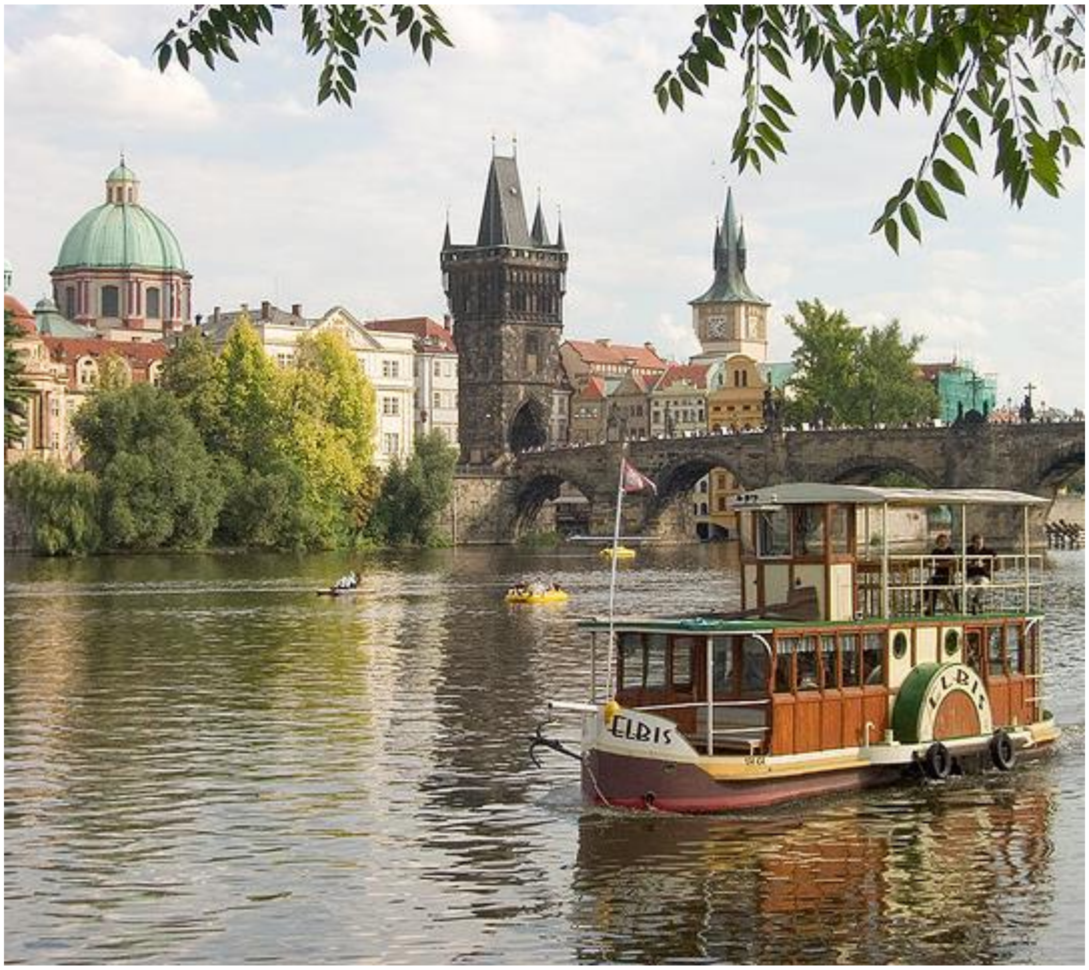
4. Assumptions: the environment never changes, anything not in the control footage is considered wrong (flaw).

   1. We will need to record a control footage of the assembly (human-supervised to make sure every step in this footage is correct). We can record without supervision after this.
   2. To detect any flaws in the assembly of a part, we will first create a binary image (subtracting the control footage by the recorded footage) frame by frame.
   3. Set a threshold value for it to be considered as a “flaw”. It should not be too small since it may pick up minor differences like the conveyor belt is 1mm left in some frame. It should not be too big either as it may ignore the “flaw” all together.
   4. Now, we can have a script to read over the resulting images and have it report error if there are non-black pixels in the image.

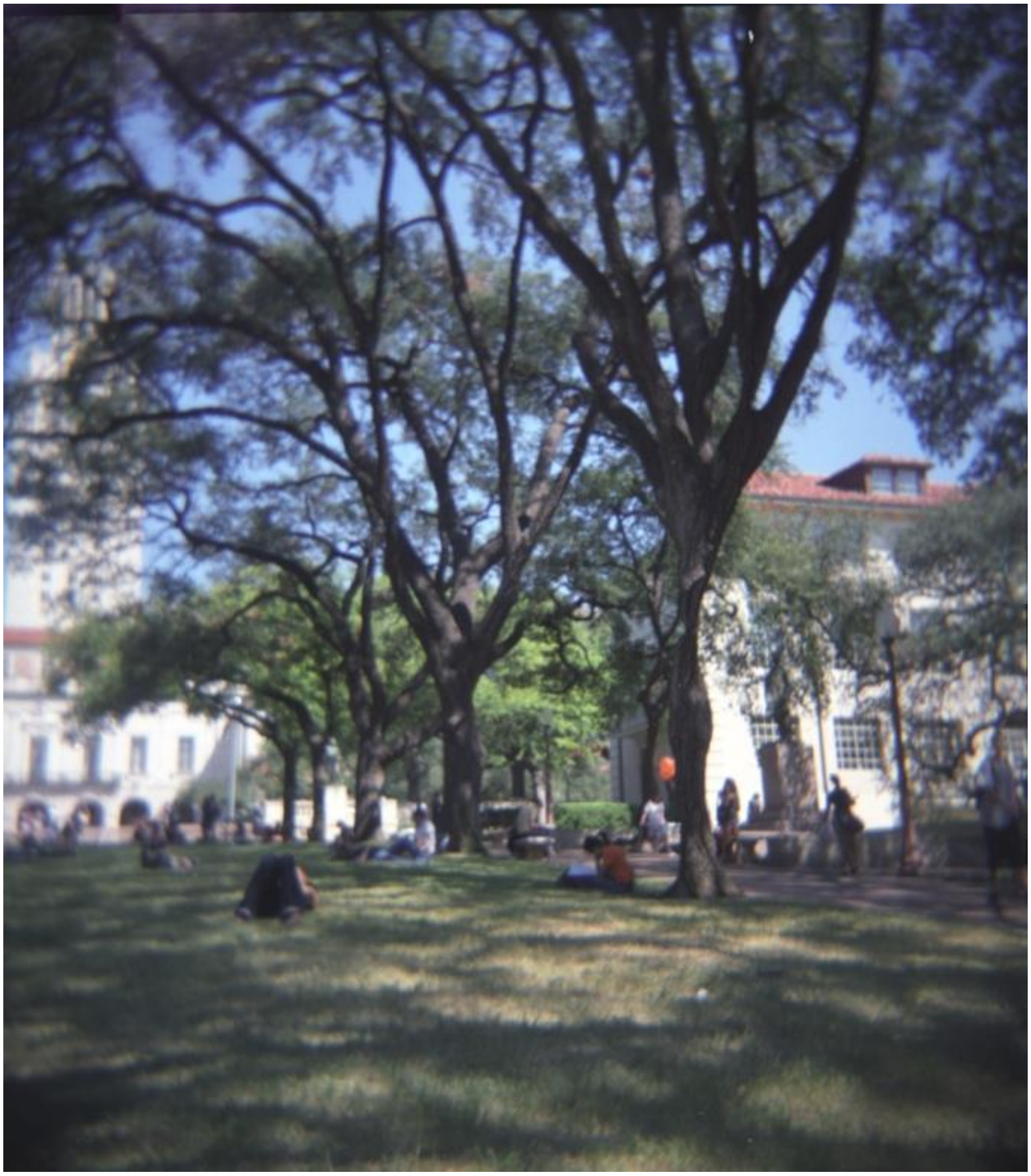
II.

1. Run: >> seam_carving_decrease_width

   outputReduceWidthPrague.png

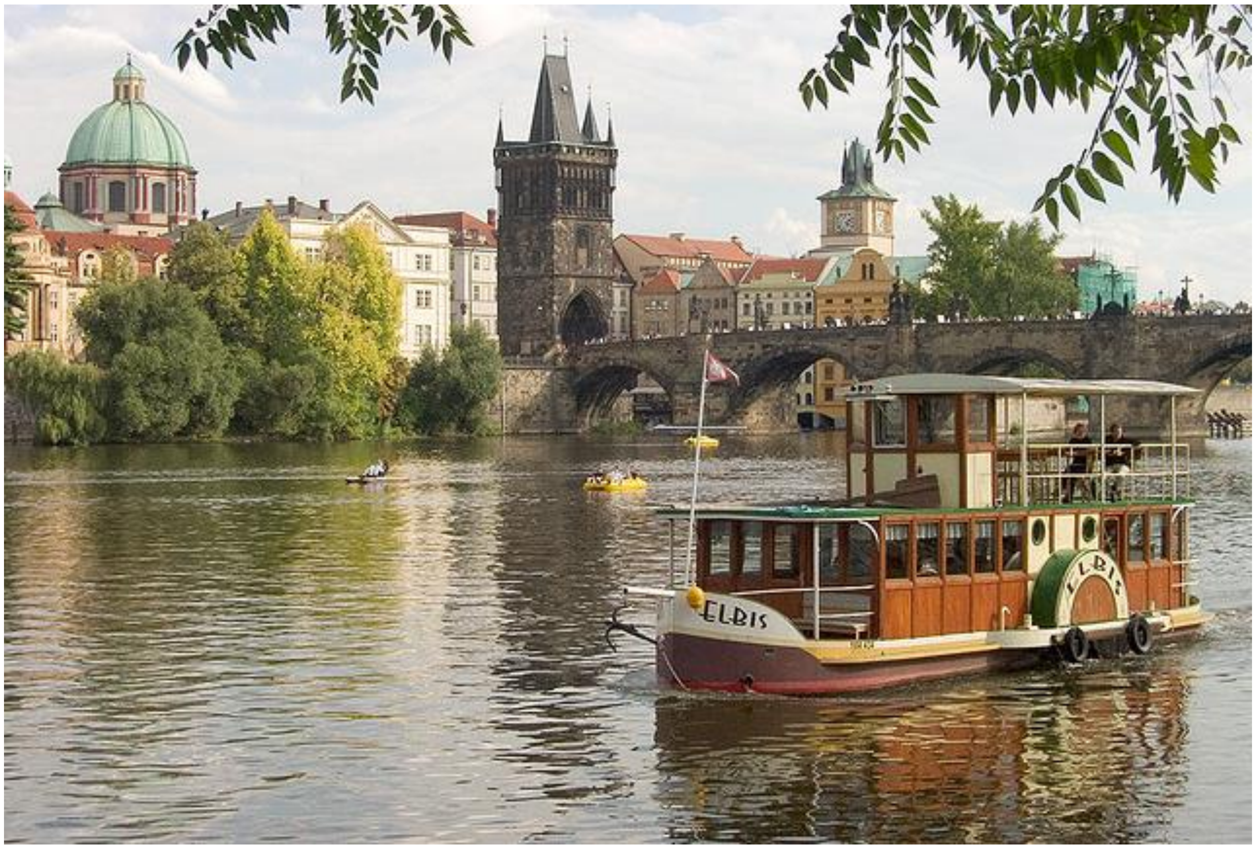


outputReduceWidthMall.png

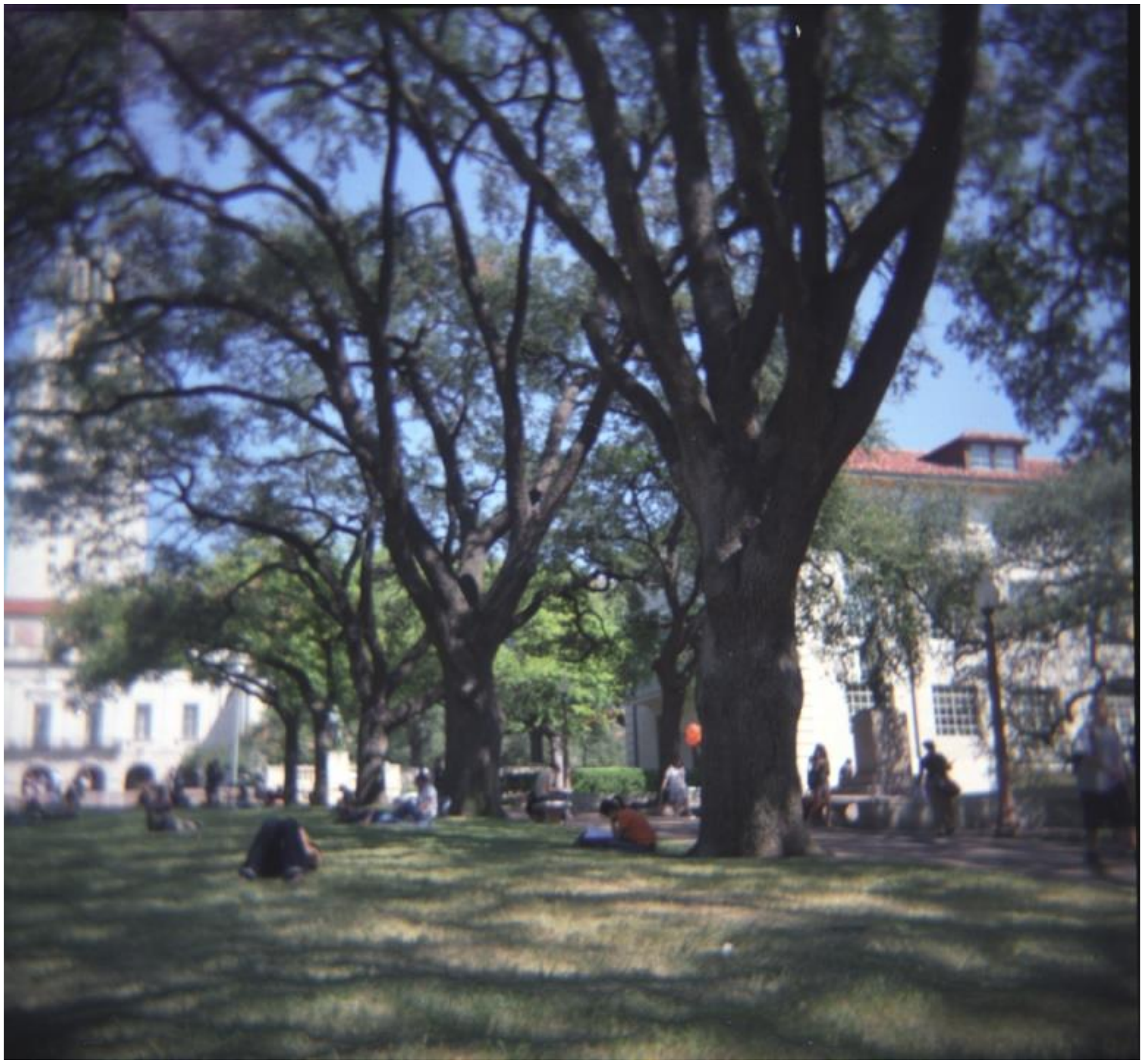


2. Run: >> seam_carving_decrease_height

   outputReduceHeightPrague.png

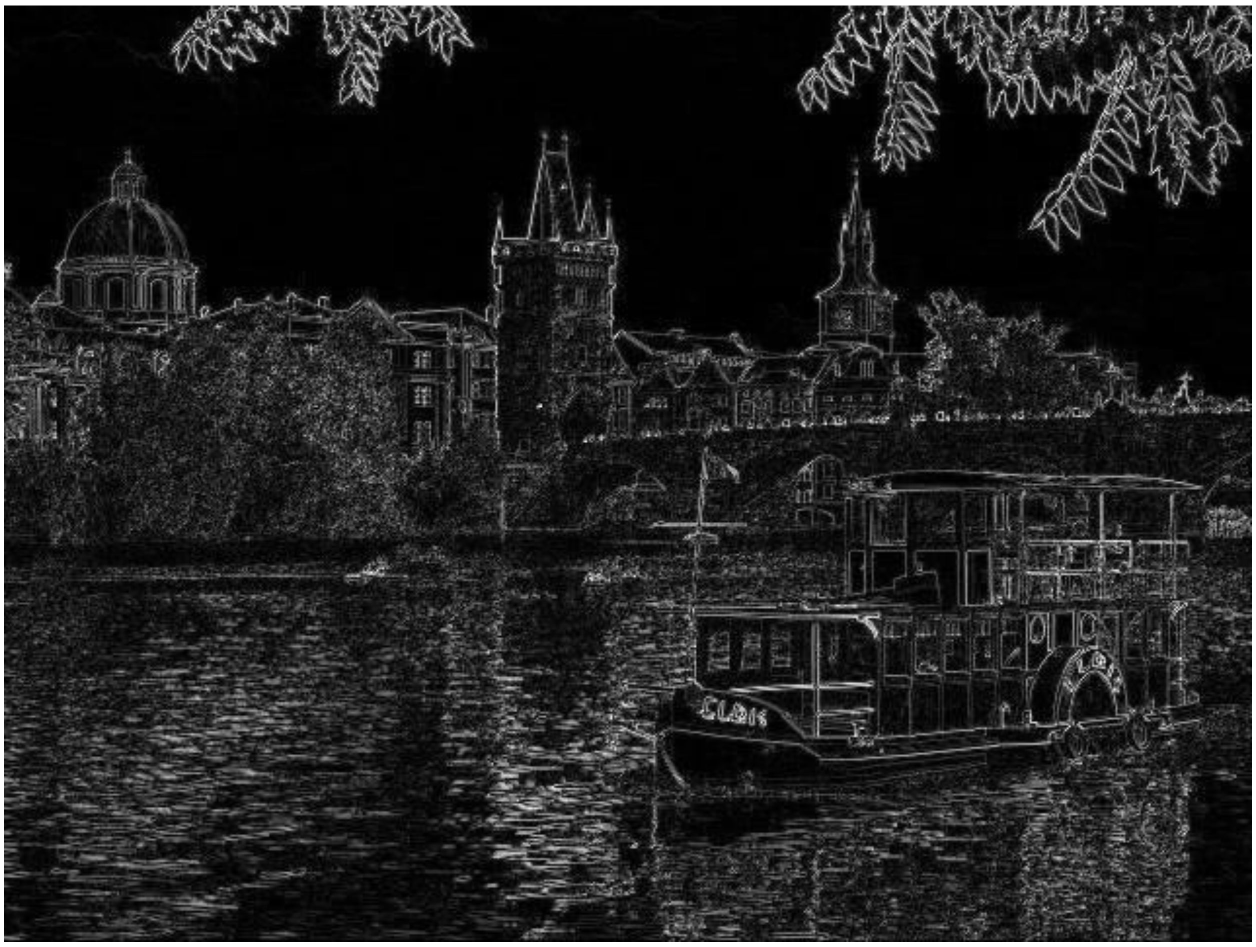


outputReduceHeightMall.png



3. Run: >> imwrite(energy_img(imread('inputSeamCarvingPrague.jpg')), 'energyImg_Prague.png', 'PNG');

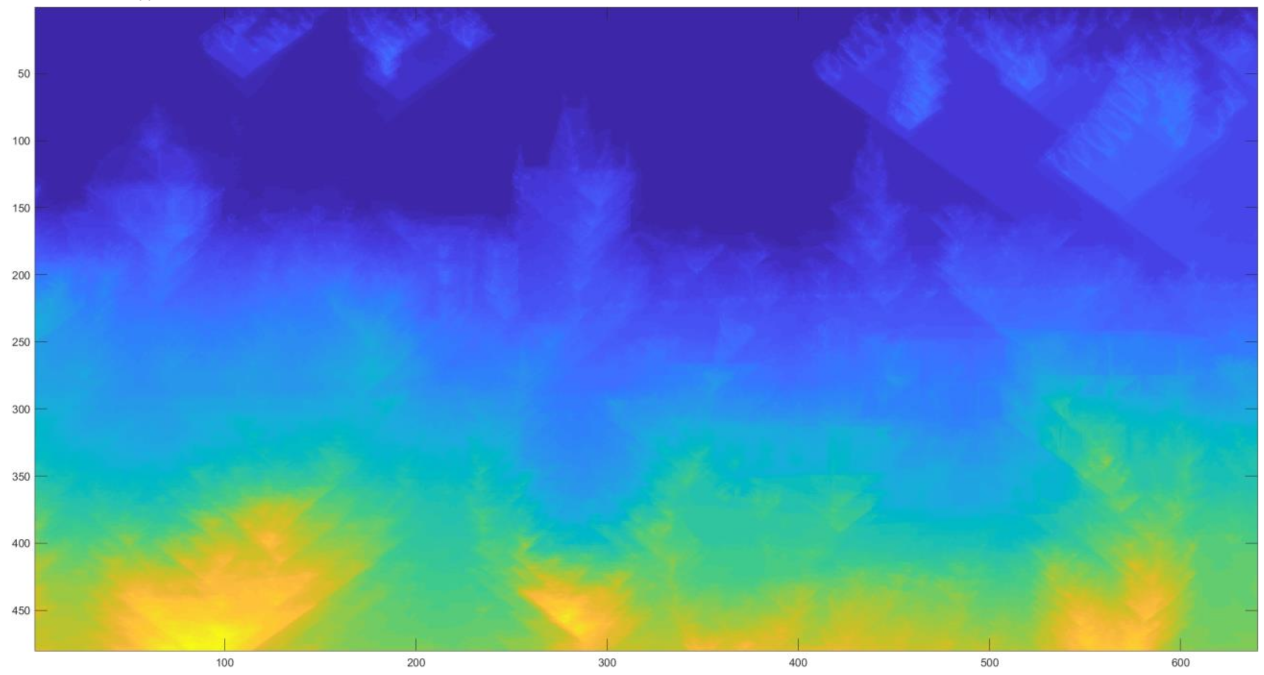
   Energy image of Prague:



The energy map is the gradient of the original image (in gray scale). The more drastic the change, the brighter it will be in the energy map. In our case, the white lines (or the edges) are the result of the drastic changes between two objects (sky vs. building etc.).

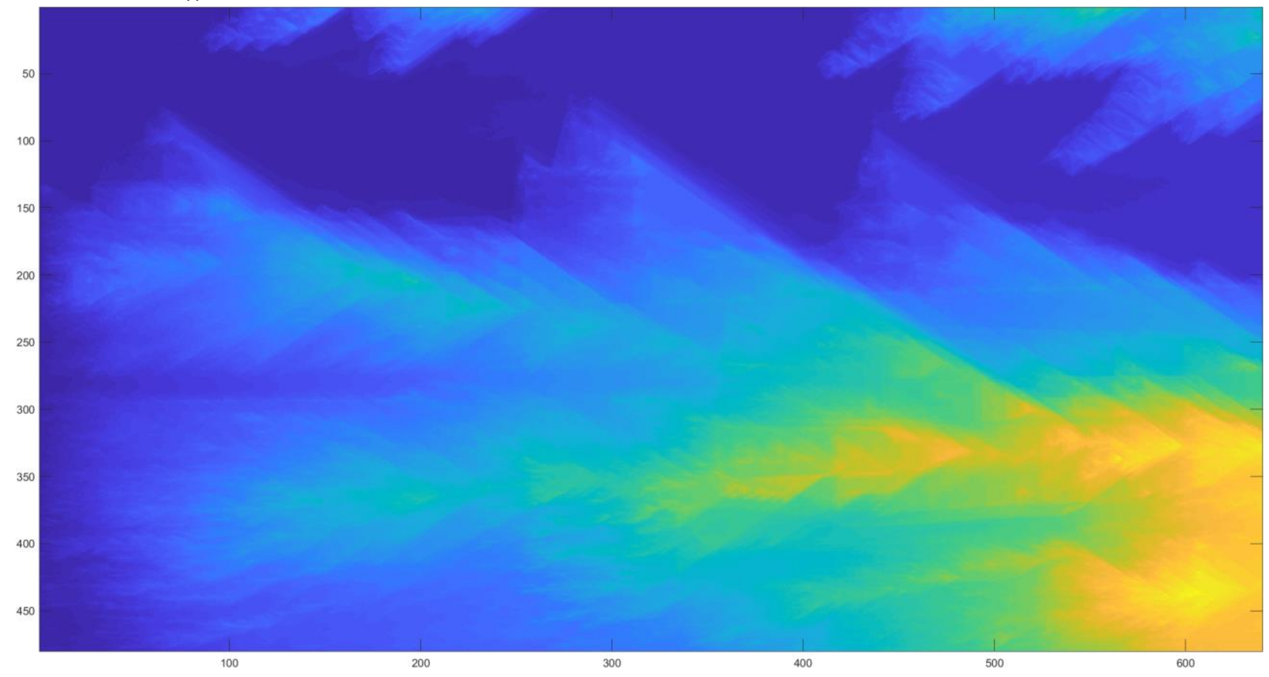
Cumulative min energy map:

Run: >> imagesc(cumulative_min_energy_map(energy_img(imread('inputSeamCarvingPrague.jpg')), 'VERTICAL'));



The vertical cumulative energy map (CEM) looks like this because the algorithm tries to find the smallest cumulative energy from top to bottom. The equation goes as following: $M(i, j) = e(i, j)+ \min(M(i-1, j -1),M(i-1, j),M(i-1, j +1))$. On the top region, there are mostly skies and very few clouds. We are able to see some clouds (different shade of blue), but then we see that it slowly turns back to the darker shade of blue eventually as it goes lower. This is because the algorithm found smaller energy values (sky and sky vs. sky and cloud). Similar principle applies to the rest of the picture. We can see that the bottom region of the picture starts turning yellow. This is because the minimum cumulative energy path still adds up to a high value.

```
Run: >> imagesc(cumulative_min_energy_map(energy_img(imread('inputSeamCarvingPrague.jpg')),
'HORIZONTAL'));
```

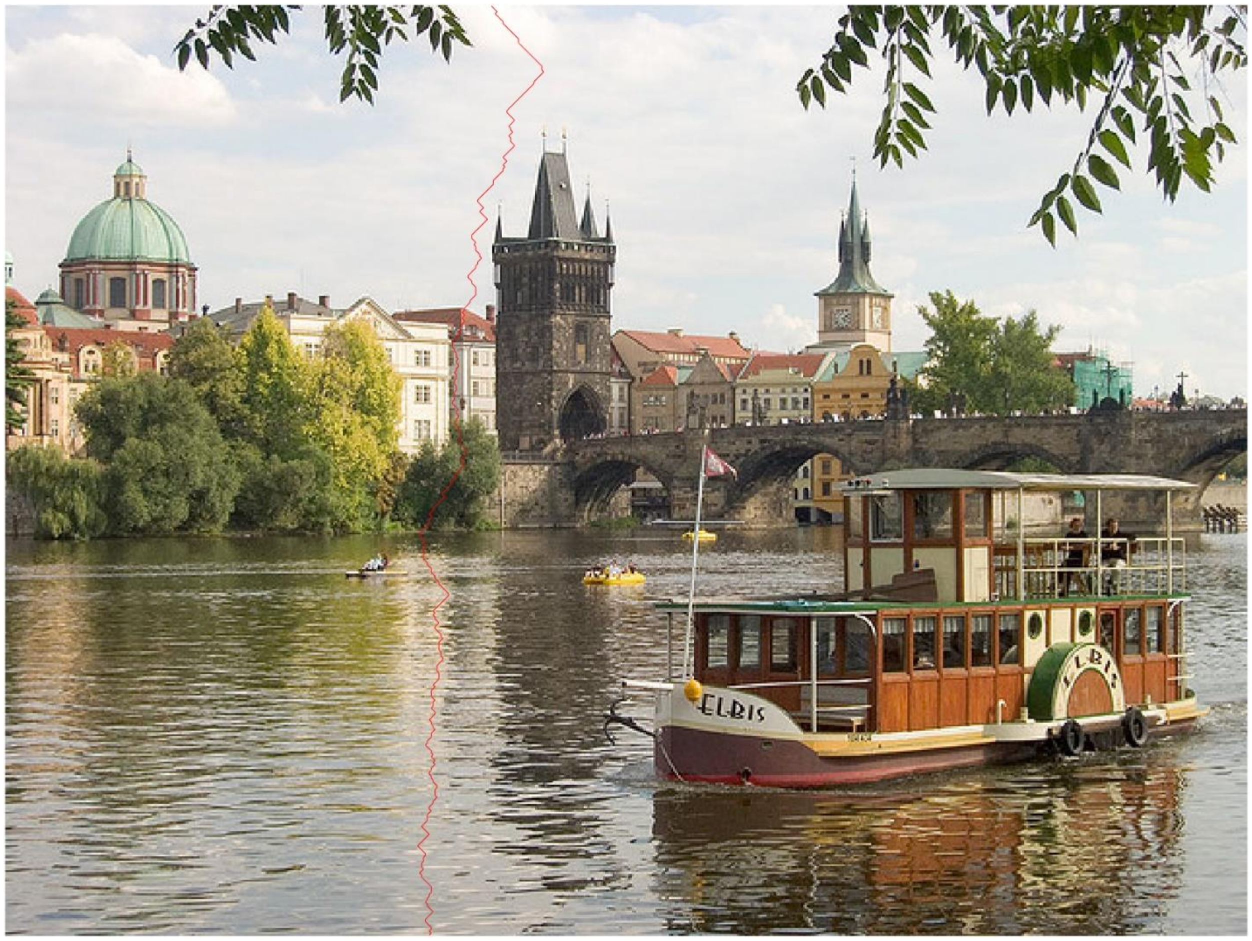


The horizontal CEM follows the same principle as the vertical one. We can see that this horizontal CEM started from left to right. The more “features” in the way, the higher the cumulated energy value (result in brighter color)

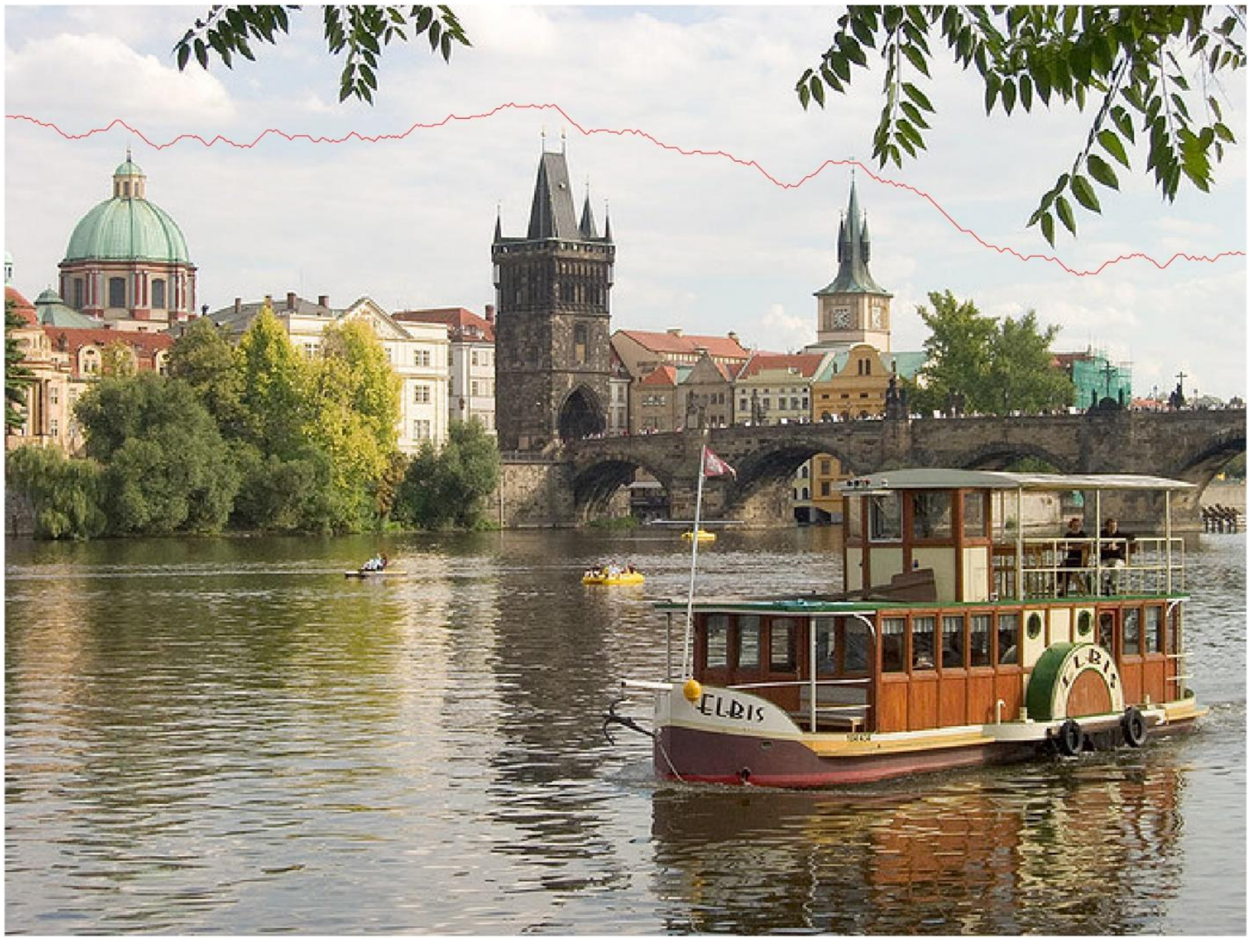
4. Run:

```
>> im = imread('img/inputSeamCarvingPrague.jpg');
>> sv = find_vertical_seam(cumulative_min_energy_map(energy_img(im), 'VERTICAL'));
>> sh = find_horizontal_seam(cumulative_min_energy_map(energy_img(im), 'HORIZONTAL'));
>> view_seam(im,  sv, 'VERTICAL');
>> view_seam(im,  sh, 'HORIZONTAL');
```

First vertical seam:

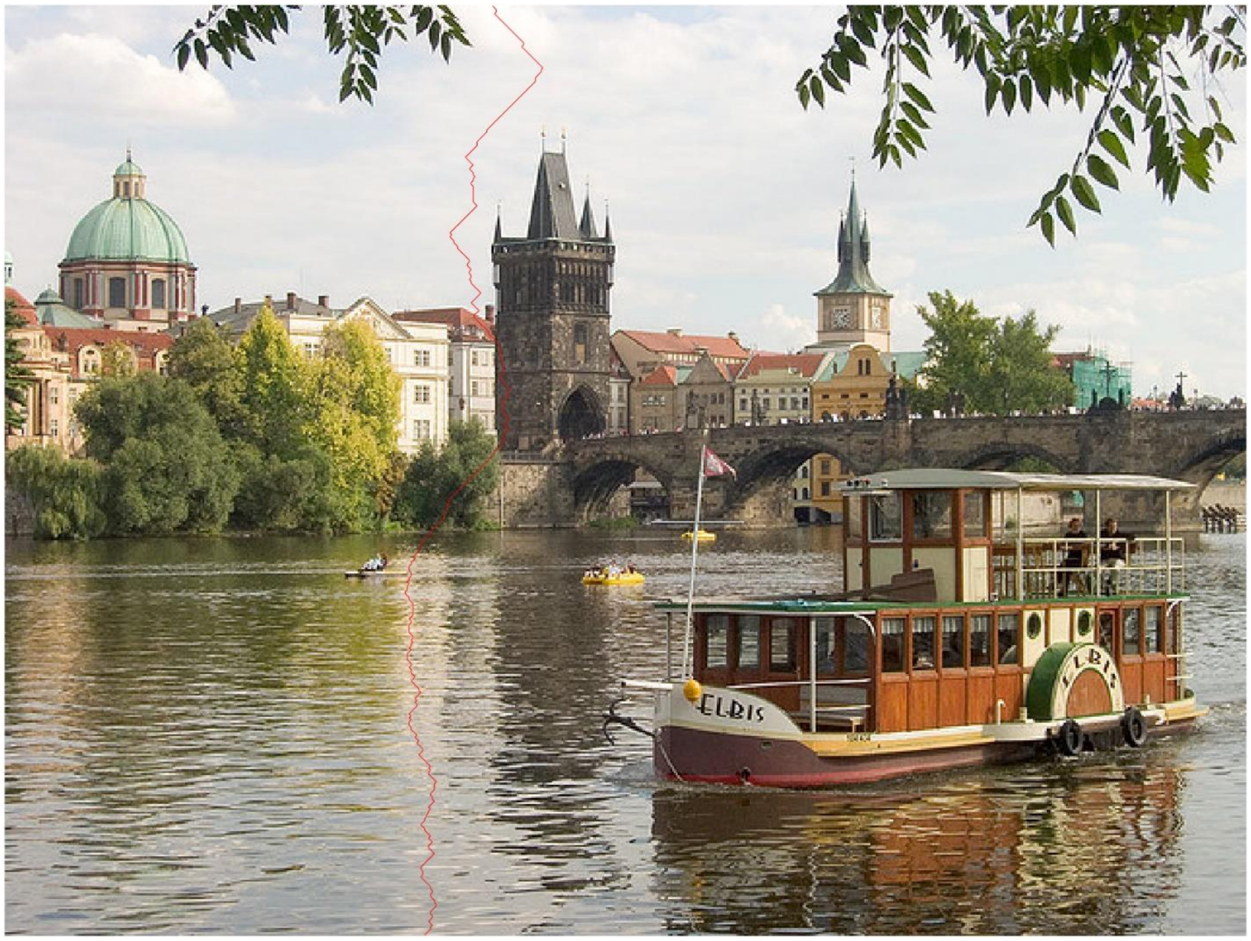


First horizontal seam:

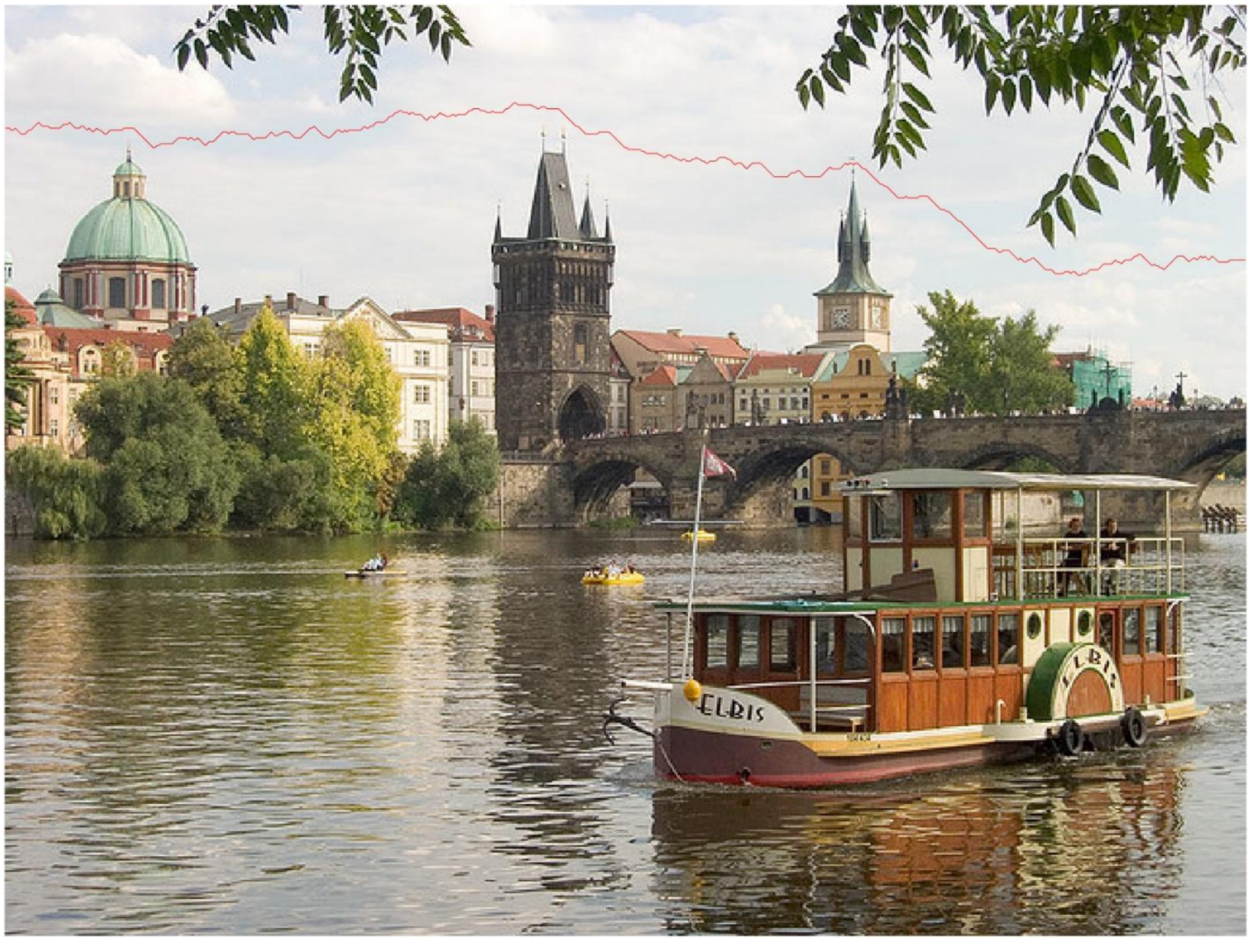


5. Changed imfilter to [1,2,1;0,0,0;-1,-2,-1] for dx and [1,0,-1;2,0,-2;1,0,-1] for dy

   First vertical seam:

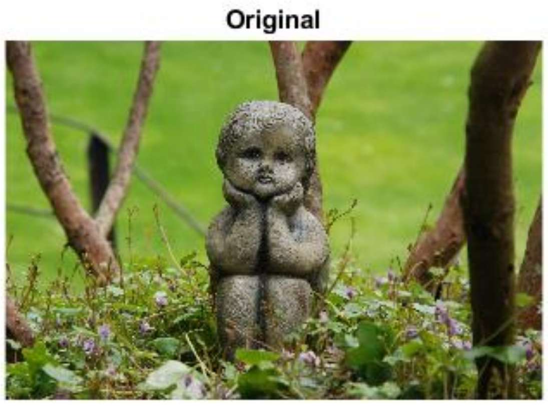


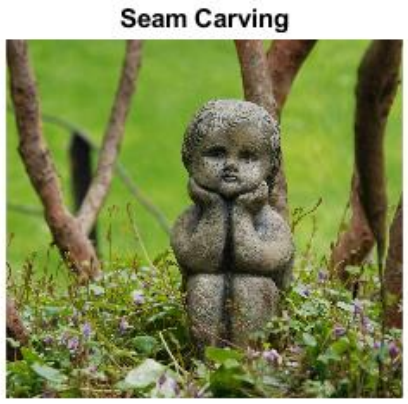
First horizontal seam:

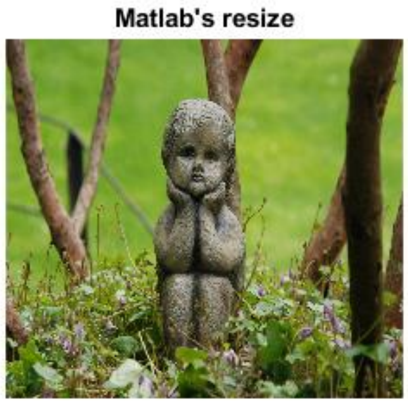


6. Run: >> run_seam(filename, px, direc);

   Original: (1920x1285) **Image by MrsBrown from Pixabay**
   Resized: (1420x1285) Reduced width by 500 pixels with reduced width



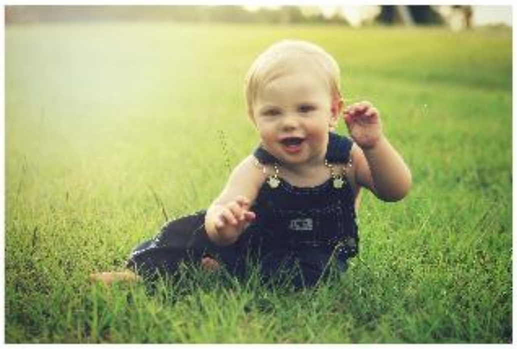




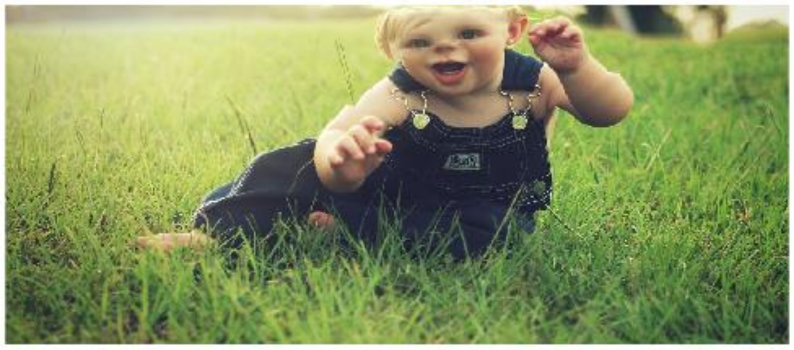
The seam carving approach better preserved the stone child than the matlab's default resize. The tree is turned into a dead branch without looking too odd at the first glance. I would say this is a great success.

Original: (1280x853) **Image by lisa runnels from Pixabay**
Resized: (1280x553) Reduced height by 300 pixels

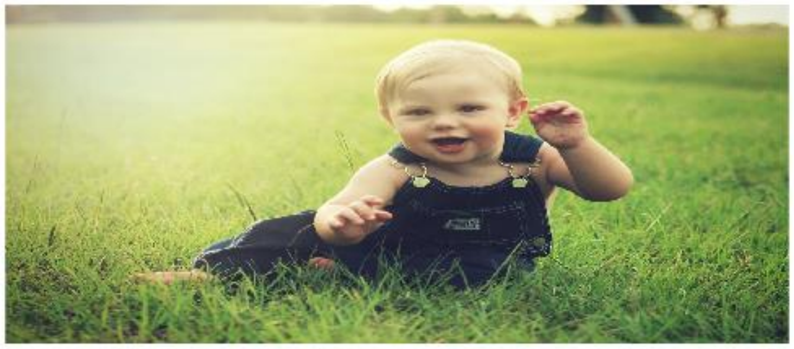
Original



Seam Carving



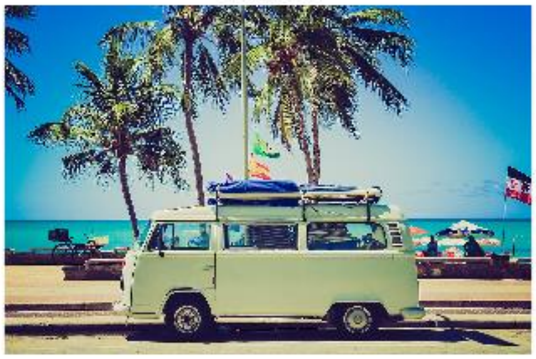
Matlab's resize



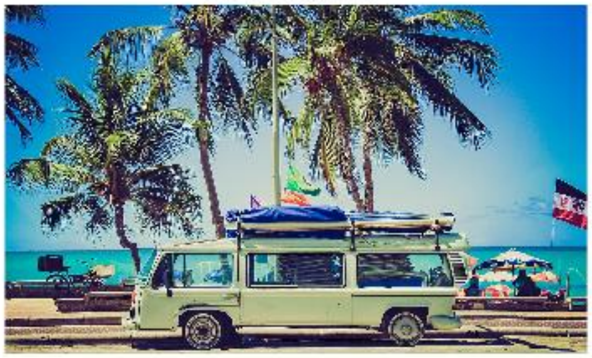
The matlab’s resize is definitely better in this case. The seam carving created an abomination. However, we can see that the features on the face (eyes, mouth, nose) are better preserved on the seam carving than the matlab’s resize. If only the face color pixel is not removed, seam carving will probably yield a better result than matlab’s default resize.

Original: (1280x847) Image by crgutman from Pixabay
Resized: (1080x647) Reduced width by 200 pixels and height by 200 pixels

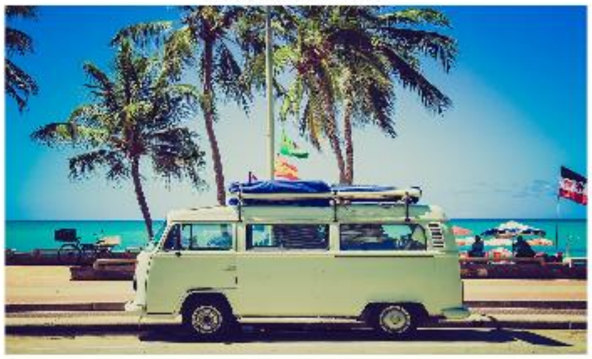
Original



Seam Carving



Matlab's resize

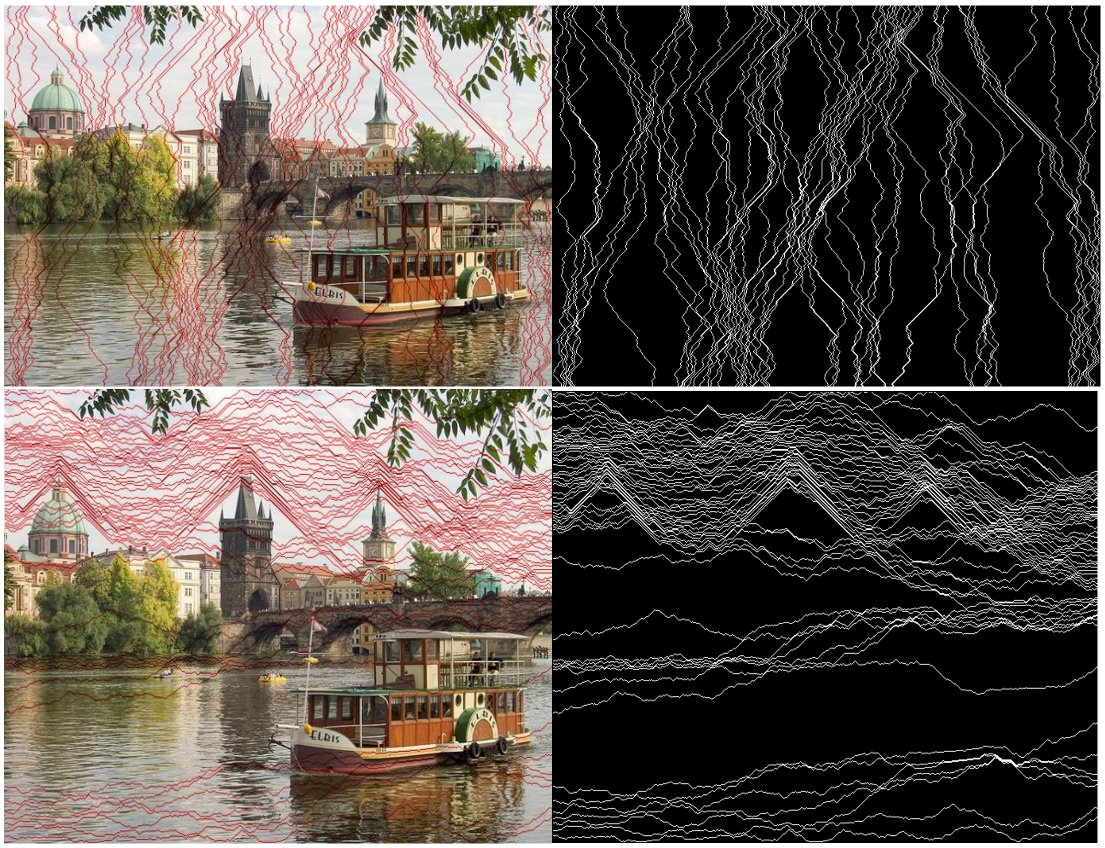


The seam carving preserved the window in the van and removed the consistent pixels on the van. As a result, we get a squished van with enormous windows. It is a cool image, but not the result we wanted. The van is definitely the focus in this picture, and matlab's default resize has done a better job preserving that.

III. Extra Credit

3. Increase width or height

   To implement this, I first find all N seams in the picture (N = pixel to expand). For easier implementation, I expanded the images and left the expanded spaces with an shade of red and on a black image with white lines as such: (see output_expand_img_w_seams.png and output_expand_img_w_seams_h.png for a bigger picture)

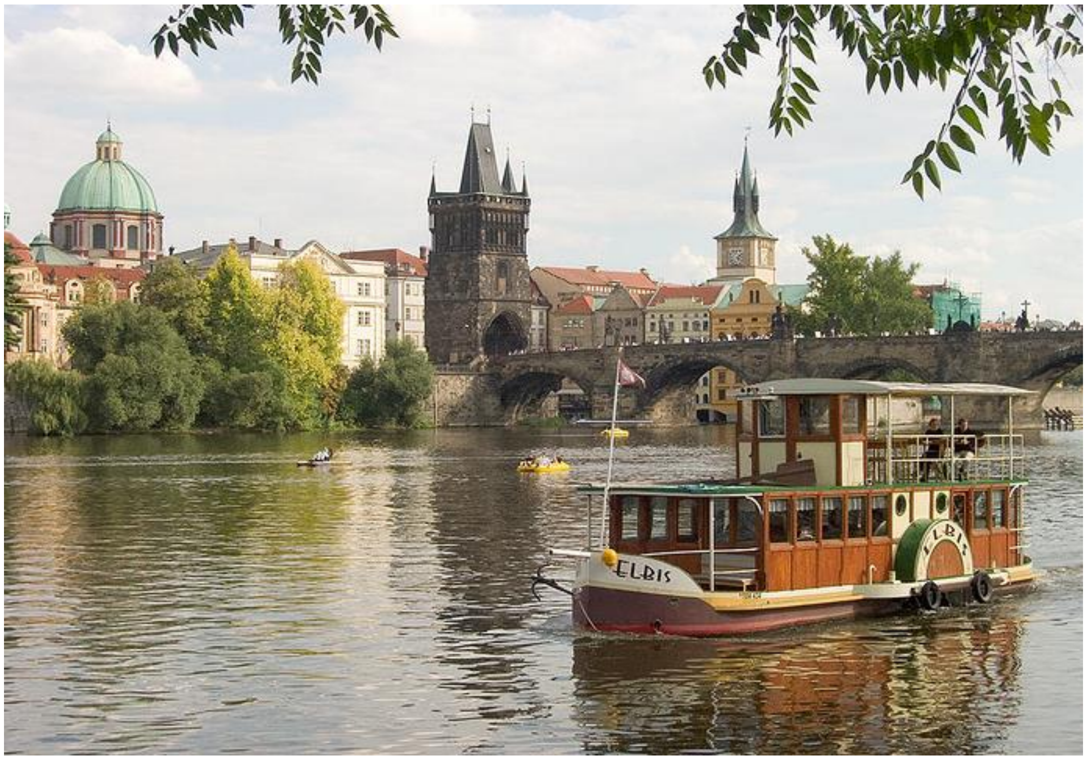


   Then, I take the index at which the pixel is white in the black image and replace the red pixels in the original image (with neighbors' color value). This is one easy way I thought of to find all the seams and handle them all together at the end. Since expanding the image immediately with the average of neighbors' value will likely to cause the energy function to mark the newly expanded pixel with low energy value. If that happens, the seam finder will only pick the seam region again and again. It will most likely end up with a particular region stretched out all the way which isn't what we want. Marking the expanded spaces with red (or any odd color that does not belong in the picture) will prevent energy function Obviously, this is not the best way to treat it as I can imagine any color with red pixel will easily mess this up, but the result is very satisfying for such lazy implementation.

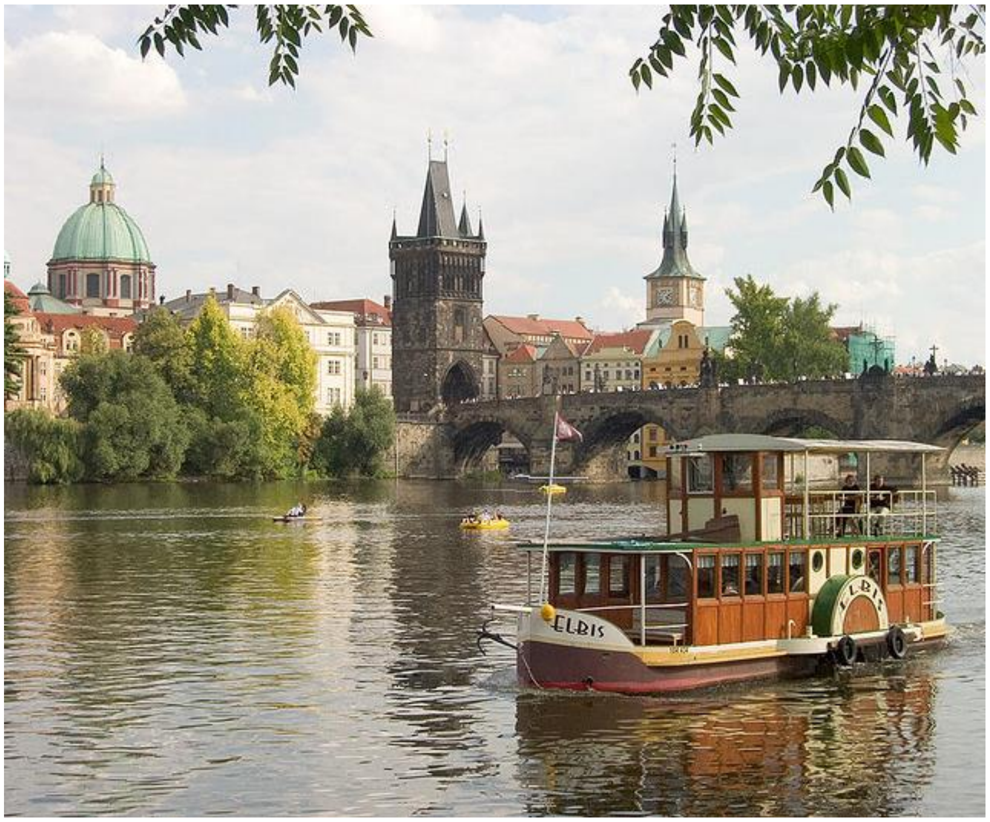
   Run: >> run_seam_expand(filename, px, direc);

Results:

Prague extended width by 50 pixels (690x480)



Prague extended height by 50 pixels (640x530)



Custom picture 1 extended width by 100 pixels (2020x1285)

Custom picture 2 extended height by 100 pixels (1280x953)

Custom picture 2 extended height by 250 pixels (1280x1103)

4. Greedy

   Take the minimum energy value from the first row/col as the starting point of the seam. Then, navigate down from that starting point (choose the smallest energy value path).

   Run: >> run_greedy(filename, px, direc);
   Note that undefined behavior may occur if px is greater than image size

   For easy comparison, I ran greedy on images that I used above. The results are fairly disappointing as it ended up with an unnatural transition in every image I tested. It is understandable that why it happens. Since it is not using dynamic programming, it picks the lowest value on the top row and progress from there. But, the rest of the lower values are usually around this lowest value. So, when reducing an image by 100 pixels, it is possible that all 100 seam comes from the same area (which creates this unnatural cut).