Problem Set 1

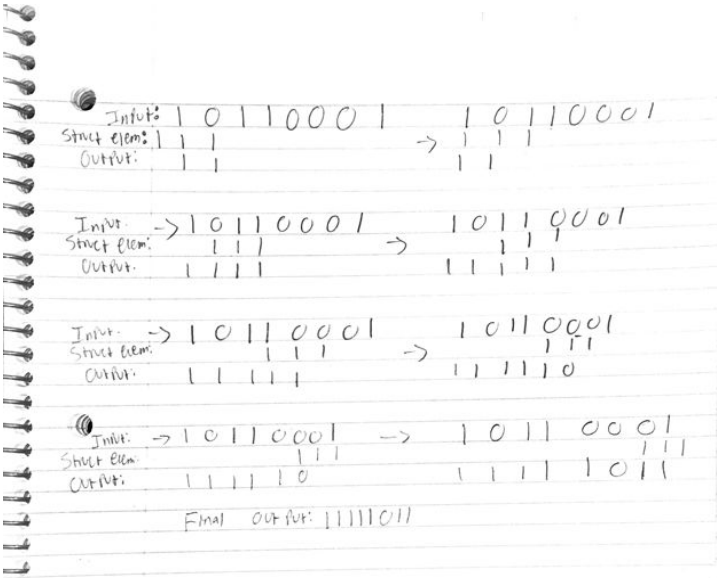
Wyatt Robertson, Gabriel Brusman
913920853, 914060880

1. Short Answer

1.1 Give an example of how one can exploit the associative property of convolution to more efficiently filter an image.

Suppose you want to apply two filters, say *h* and *f* to a large image *g*. One could first convolve the first filter with the image (*h* * *g* = *g*') and then convolve the second filter with the new image (*g*' * *f* = *result*), but this is computationally expensive, since one would have to slide both filters across the image and compute the results one at a time, which is a lot of multiplication operations. Given the associative property of convolution, one could first multiply the two filters together (*h* * *f* = *k*), and then simply convolve the image with the new filter (*k* * *g* = *result*). This is much more efficient since we only have to filter the whole image once, and so we are performing significantly fewer multiplications.

1.2 This is the input image: [1 0 1 1 0 0 0 1]. What is the result of dilation with a structuring element [1 1 1]?



Final Result = 11111011

1.3 Describe a possible flaw in the use of additive Gaussian noise to represent image noise

In using additive Gaussian noise to represent image noise, we are assuming that all the noise in the image is random and independent. This is the case in many situations, but not always. Consider an image taken where there was a significant amount of dust only on the bottom-left quadrant of the lens. Then there will be significantly more noise in the bottom left of the image than in the rest of the image. This means that the noise isn't entirely independent, which is problematic. Additionally, the use of additive Gaussian noise with no thresholding could introduce pixel values to the image that are greater than the maximum (255) and values that are less than 0.

1.4 Design a method that takes video data from a camera perched above a conveyor belt at an automotive equipment manufacturer, and reports any flaws in the assembly of a part. Your response should be a list of concise, specific steps, and should incorporate several techniques covered in class thus far. Specify any important assumptions your method makes.

I. First we should convert the video to grayscale, since the same parts may be manufactured in different colors and we don't want the color to affect the output of our method.
II. We will want to remove noise from our video, so assuming we have consistent conditions for our camera and conveyor belt we can use Gaussian smoothing to denoise the video
III. We can use Canny Edge Detection with Hysteresis Thresholding to extract the edges from the video.
IV. Then we can scan for certain features we're looking for by using Chamfer Distance.
    A. We can scan for features that we want to ensure are present on the equipment, and we can also scan for features that we do not want to be present on the equipment. We can use the results of these scans to determine what flaws are present in each piece of equipment.

2. Programming Problem

2.1 Seam Carving Decrease Width

In the following section, we will display the code and attach the graphics for the corresponding section in Problem Set 1. This problem is dedicated to using the functions defined earlier to reduce the width of two different images by 100px.

```
function seam_carving_decrease_width()
% Prague Image
im = imread('inputSeamCarvingPrague.jpg');
eim = energy_img(im);

[a,b] = decrease_width(im, eim);
for i = 1:99
    [a,b] = decrease_width(a, b);
end

p1 = imshow(a);
title("Reduced Width (100px)");
saveas(p1, "outputReduceWidthPrague.png");
```


```
% Mall Image
im = imread('inputSeamCarvingMall.jpg');
eim = energy_img(im);

[a,b] = decrease_width(im, eim);
for i = 1:99
    [a,b] = decrease_width(a, b);
end

p2 = imshow(a);
title("Reduced Width (100px)");
saveas(p2, "outputReduceWidthMall.png");

end
```

Reduced Width (100px)

2.2 Seam Carving Decrease Height

In the following section, we will display the code and attach the graphics for the corresponding section in Problem Set 1. This problem is dedicated to using the functions defined earlier to reduce the height of two different images by 50px.

```
function seam_carving_decrease_height()
im = imread('inputSeamCarvingPrague.jpg');
eim = energy_img(im);

[a,b] = decrease_height(im, eim);
```

```
for i = 1:49
    [a,b] = decrease_height(a, b);
end

p1 = imshow(a);
title("Reduced Height (50px)");
saveas(p1, "outputReduceHeightPrague.png");
```


```
im = imread('inputSeamCarvingMall.jpg');
eim = energy_img(im);

[a,b] = decrease_height(im, eim);
for i = 1:49
    [a,b] = decrease_height(a, b);
end

p2 = imshow(a);
title("Reduced Height (50px)");
```

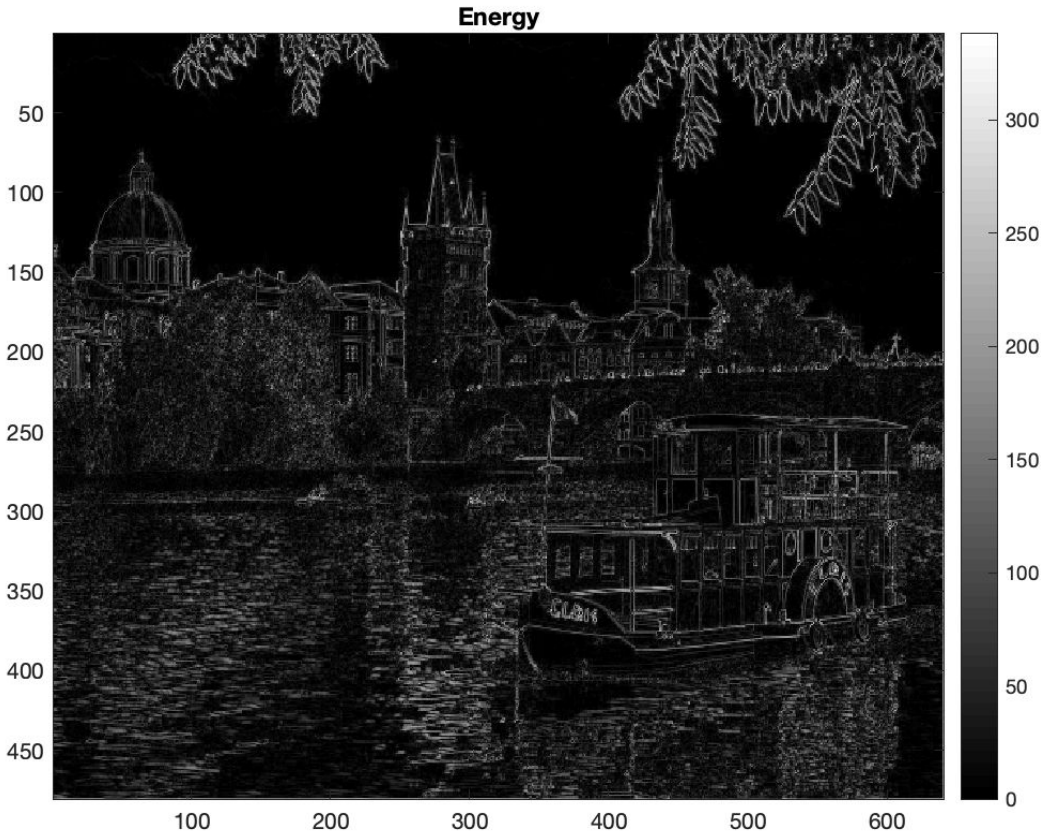
```
saveas(p2, "outputReduceHeightMall.png");
end
```

Reduced Height (50px)

2.3 Energy and Cumulative Min Energy

Energy, as defined by Avidan and Shamir, 2007, is given by the formula $Energy = \sqrt{dx^2 + dy^2}$, where *dx* is the gradient of the greyscale image in the x-direction, and *dy* is the gradient of the greyscale image in the y-direction. Following form the formula, pixels corresponding to high energy in the below image indicate a high change in intensity in either in x, y, or both directions.

Intuitively then, shapes with clear boundaries or textures such as the boat or buildings in the image below have defined silhouettes because they or their subcomponents lie on top of other images in the photo, which disrupt the gradients of the greyscale image in either or both directions.

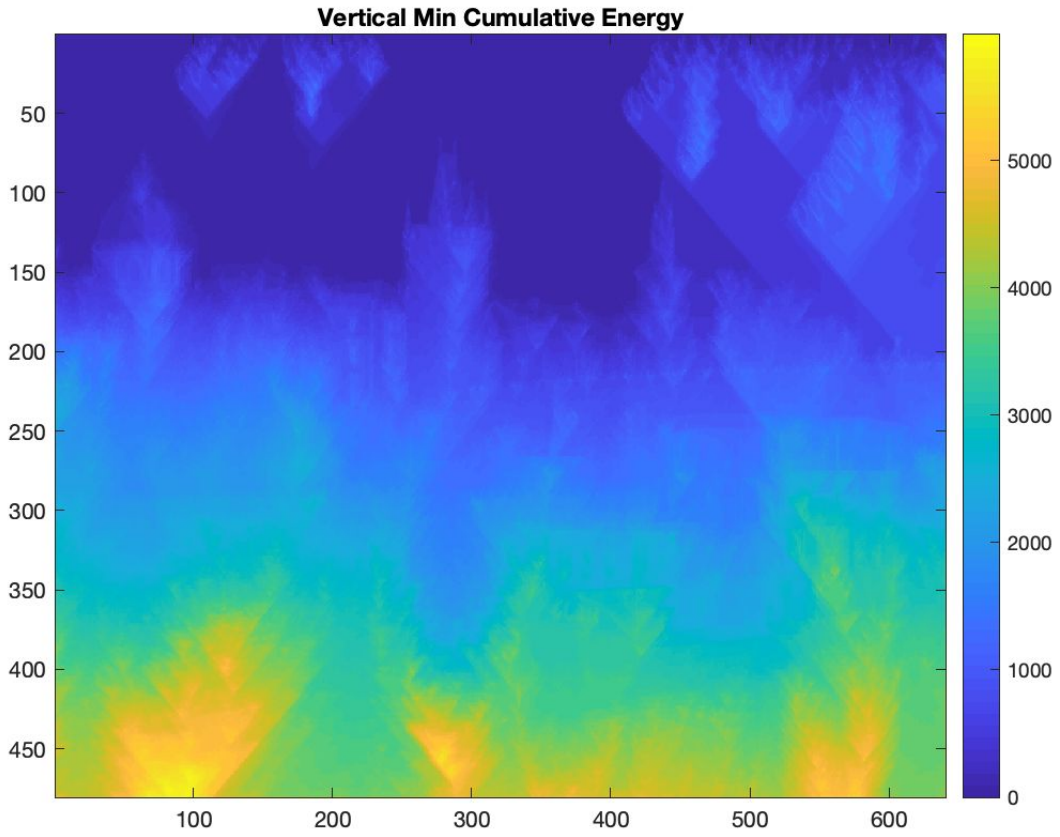


Minimum cumulative energy in the vertical direction, defined by Avidan and Shamir, 2007, is given by the equation

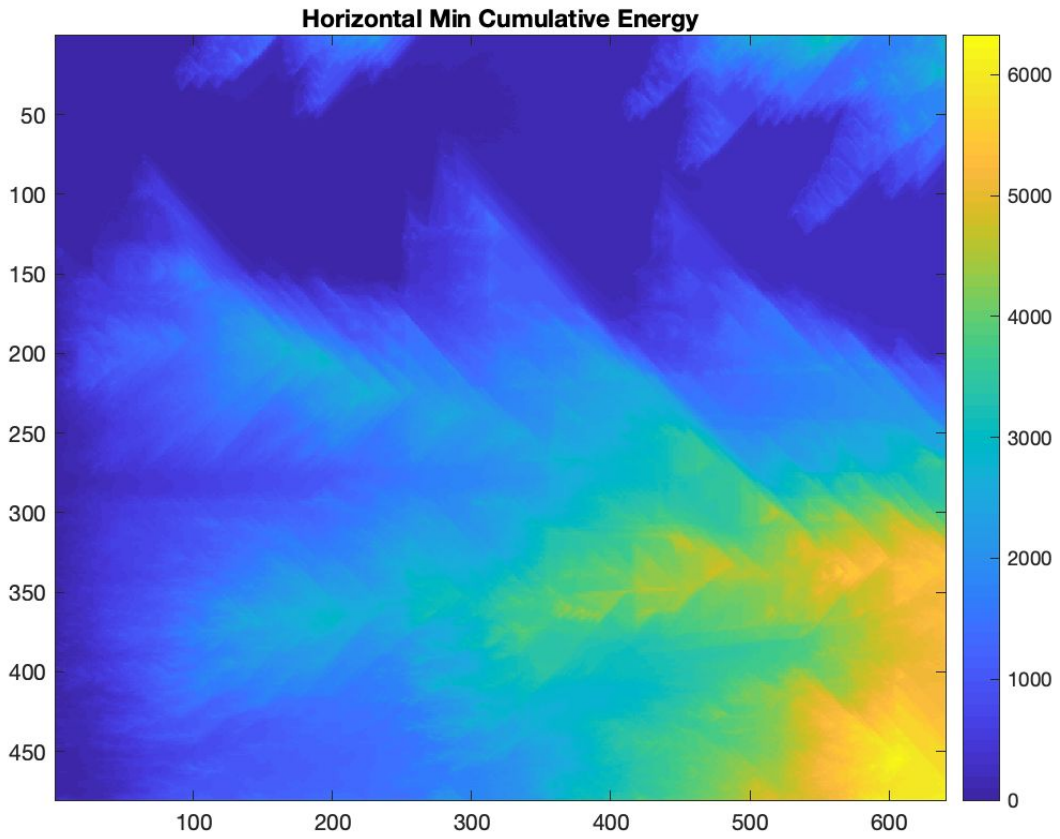
$$M(i,j) = e(i,j) + min(M(i-1,j-1),\ M(i-1,j),\ M(i-1,j+1))$$

Minimum cumulative vertical energy calculated from the above image is given below. Areas of low intensity correspond to areas of low energy “seams”, which are 8-connected paths of pixels across an image. Areas of low energy such as the sky are noticeably low in intensity in both photos, since it contains very little energy.

You will also notice the top half of the image is relatively low intensity, since the image hasn’t had the opportunity to accumulate large amounts of energy.



The next figure is a representation of minimum cumulative energy in the horizontal direction. It shares features similar to the vertical direction for the reasons described above.



2.4 Optimal Seams

Using the dynamic programming method described by Avidan and Shamir, 2007, optimal vertical and horizontal seams can be calculated using a traceback on the minimum cumulative energy maps. They are calculated by finding the index of the minimum of the last row or column in the image, and traveling upwards in an 8-connected path to the top of the image in the direction of least energy.

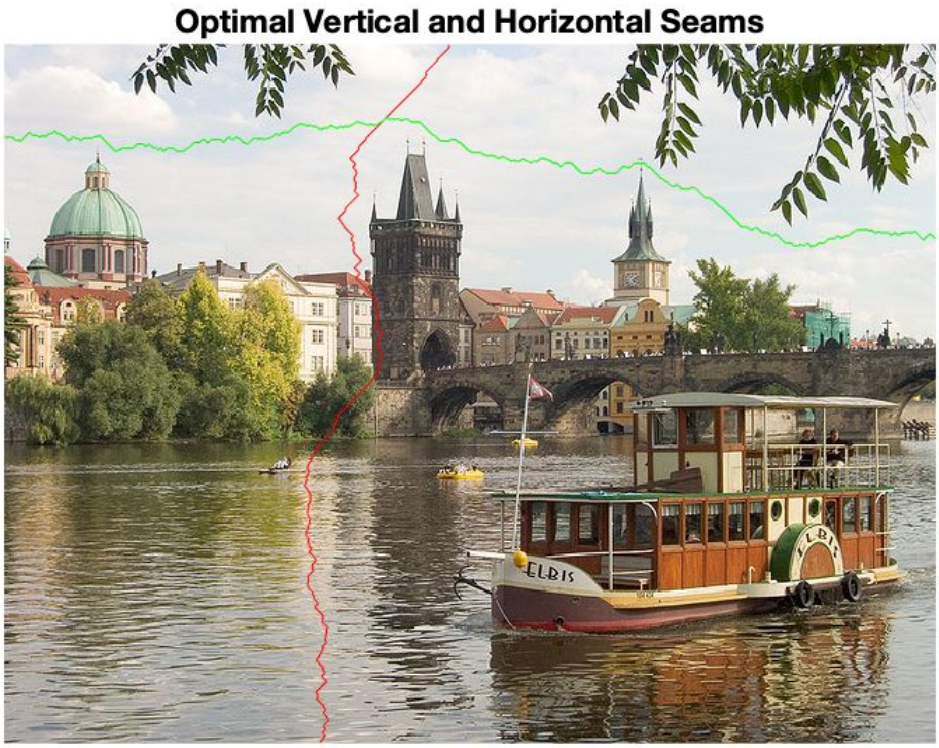
```
function verticalSeam = find_vertical_seam(cumulativeEnergyMap)
% Initialize seam
[n,m] = size(cumulativeEnergyMap);
verticalSeam = zeros(n,1);

% Find min index of last row for traceback
[~,y] = min(cumulativeEnergyMap(end,:));
verticalSeam(n) = y;

% Track column indices during traceback
for i = n-1 : -1 : 1
```

```
        [~,y2] = min([borderV(cumulativeEnergyMap, i, y-1, m),
borderV(cumulativeEnergyMap, i, y, m), borderV(cumulativeEnergyMap, i,
y+1, m)]);
        y = y + y2 - 2;
        verticalSeam(i) = y;
end
end
```

In this image, the optimal seams travel through the sky in the horizontal direction (which contains a small amount of energy), and through the building and water in the horizontal direction (which is the path of *least* cumulative energy in the vertical direction).

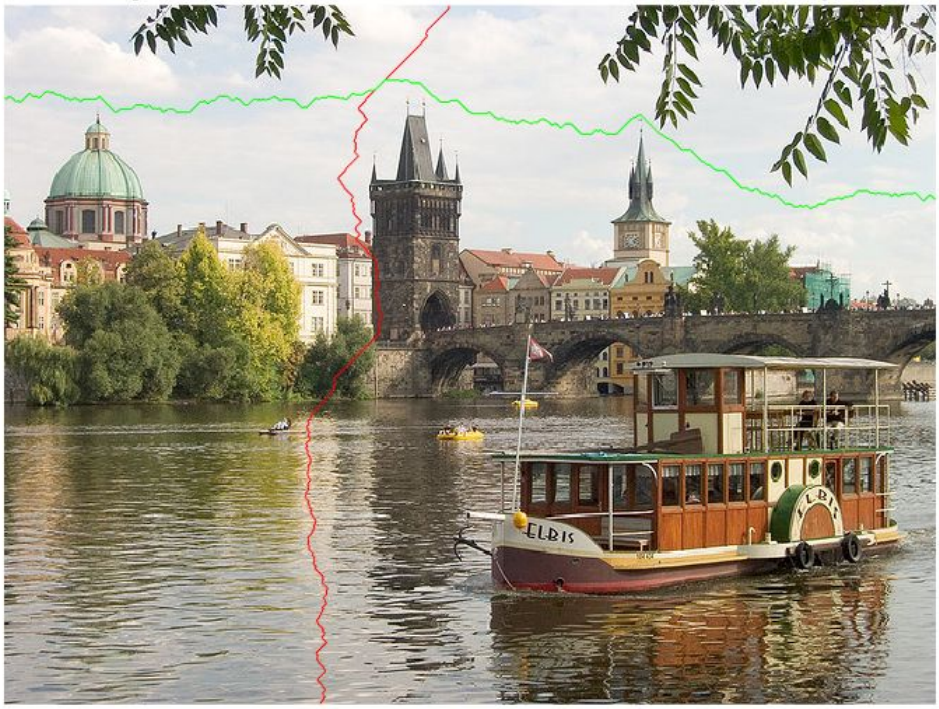


2.5 Optimal Seams (Prewitt)

Changing the filter used to find the gradient of the image to a Prewitt filter changes the results of the energy map, which changes cumulative minimum energy, and therefore changes the direction and locations of the optimal seams.

Below is the output of the optimal seams using a Prewitt filter.

Optimal Vertical and Horizontal Seams (Prewitt)



You will notice the optimal seam in the vertical direction diverges from the seam in the previous example near the tall building. Additionally, the seams cross at a different point in the image.

2.6 Original Results

The following section will be experiments performed on my own original images.

Original

Carving

Resize

For this skull image (403px, 604px), seam carving and resize were used to reduce the width of the image by 100px (403px, 504px) and compared to the results of Matlab's resize function. We had expected seam carving to generally retain the shape of the skulls in the image and reduce the vertical space around them. While seam carving did remove some of this space, it also removed a large section from the back of the left skull. This was most likely due to the traceback preferring the left side of the image due to the diagonal slanted shape at its base, and the long

consistent groove and shadow on the left side of the skull. Resize did not contain this artifact, but it also squeezed the right skull, whereas seam carving seemed to ignore it.

The next image is of Torii gates (631px, 473px). Seam carving and resize were used to reduce the height and width of the image by 100px (531px, 373px). We expected seam carving to have significant artifacts on this image since it contains text near optimal seams, which would most likely not be removed. As expected, significant warping appeared on the supports of the torii gates near text, since the seams around the text were deemed optimal and removed. Resizing did a better job in this image maintaining the integrity of the text and other objects.

The last image used is a photo of street art (662px, 717px). Seam carving and resize were used to reduce the height and width of the image by 100px (562px, 617px). Space in between the two figures was expected to be reduced since it contained no art, however because of the graffiti in the center of the wall and at its base, seam carving seems to have caused warping between the figures, as the traceback was thrown off balance with the bottom graphitti and a refusal to remove the center wall graffiti.

3. Extra Credit

3.4 Greedy Solution

First we will include the code we needed to modify for the greedy solution. It only required slight modifications from the dynamic programming solution.

```
function horizontalSeam = find_horizontal_seam_greedy(energyImg)
    % Initialize seam
    [numRows,numCols] = size(energyImg);
    horizontalSeam = zeros(1,numCols);

    % Find min index of first column for tracing forwards
    [~,y] = min(energyImg(:,1));
    horizontalSeam(1) = y;

    % Track column indices during trace forward
    for j = 2 : 1 : numCols
            [~,y2] = min([borderH(energyImg, y-1, j, numRows),
borderH(energyImg, y, j, numRows), borderH(energyImg, y+1, j, numRows)]);
            y = y + y2 - 2;
            horizontalSeam(j) = y;
    end
end
-----------------------------------------------------------

function verticalSeam = find_vertical_seam_greedy(energyImg)
    %finding horizontal seam of energyImg' is the same as finding vertical
    %seam of energyImg.
    verticalSeam = find_horizontal_seam_greedy(energyImg');
-----------------------------------------------------------

function [reducedColorImg,reducedEnergyImg] =
decrease_width_greedy(im,energyImg)
    n = size(energyImg, 1);
    vs = find_vertical_seam_greedy(energyImg);

    % Move the contents of im and energyImg back one space horizontally to fill
in seam
    % Append a placeholder number (0) to get chopped off the image later
    for i = 1 : n
        energyImg(i,vs(i):end) = [energyImg(i,vs(i)+1:end) 0];
        im(i,vs(i):end, :) = [im(i,vs(i)+1:end, :) zeros(1,1,3)];
    end
```

```
    % Crop placeholder column off of the right side of the image
    reducedEnergyImg = energyImg(:,1:end-1);
    reducedColorImg = im(:,1:end-1, :);
    end
-----------------------------------------------------------

function [reducedColorImg,reducedEnergyImg] =
decrease_height_greedy(im,energyImg)
    m = size(energyImg, 2);
    hs = find_horizontal_seam_greedy(energyImg);

    % Move the contents of im and energyImg up one space vertically to fill in
seam
    % Append a placeholder number (0) to get chopped off the image later
    for j = 1 : m
        energyImg(hs(j):end,j) = vertcat(energyImg(hs(j)+1:end,j), 0);
        im(hs(j):end,j, :) = vertcat(im(hs(j)+1:end,j, :), zeros(1,1,3));
    end

    % Crop placeholder row off of the bottom side of the image
    reducedEnergyImg = energyImg(1:end-1,:);
    reducedColorImg = im(1:end-1, :, :);
    end
-----------------------------------------------------------

function seam_carving_decrease_width_greedy()
    im = imread('inputSeamCarvingPrague.jpg');
    eim = energy_img(im);

    [a,b] = decrease_width_greedy(im, eim);
    for i = 1:99
        [a,b] = decrease_width_greedy(a, b);
    end

    p1 = imshow(a);
    title("Reduced Width (100px)");
    saveas(p1, "outputReduceWidthPragueGreedy.png");



    im = imread('inputSeamCarvingMall.jpg');
    eim = energy_img(im);

    [a,b] = decrease_width_greedy(im, eim);
    for i = 1:99
        [a,b] = decrease_width_greedy(a, b);
    end
```

```
    p2 = imshow(a);
    title("Reduced Width (100px)");
    saveas(p2, "outputReduceWidthMallGreedy.png");

end
-----------------------------------------------------------


function seam_carving_decrease_height_greedy()
    im = imread('inputSeamCarvingMall.jpg');
    eim = energy_img(im);

    [a,b] = decrease_height_greedy(im, eim);
    for i = 1:49
        [a,b] = decrease_height_greedy(a, b);
    end

    p1 = imshow(a);
    title("Reduced Height (50px)");
    saveas(p1, "outputReduceHeightPragueGreedy.png");



    im = imread('inputSeamCarvingMall.jpg');
    eim = energy_img(im);

    [a,b] = decrease_height_greedy(im, eim);
    for i = 1:49
        [a,b] = decrease_height_greedy(a, b);
    end

    p2 = imshow(a);
    title("Reduced Height (50px)");
    saveas(p2, "outputReduceHeightMallGreedy.png");
    end
-----------------------------------------------------------
```

Below we will compare results between the Dynamic Programming solution and the Greedy solution using the Mall and Prague examples from earlier.

3.4.1 Decrease Width:

Dynamic Programming Solution	Greedy Solution
	

As we can see here, the dynamic programming solution removed more of the boat, and you can also see that the dynamic programming solution left a slight artifact at the top of the right spire (it is slanted diagonally up and to the left). The greedy solution mostly removed seams closer to the far left side of the image, and so the greedy result looks more cropped from the left side of the image. In the greedy solution, since it removed so many pixels from the left side of the image, there are slight artifacts in the water near the left side (where the bright reflection is), as well as significant artifacts on the left side of the green domed building. The greedy solution also took out a large portion of the middle spire, which looks strange but is hard to notice on its own.

Dynamic Programming Solution	Greedy Solution
Reduced Width (100px) 	Reduced Width (100px) 

There are significant differences between these two images. The Dynamic Programming solution removed a large portion of the rightmost (front) tree, resulting in a awkward looking narrow tree. The Greedy solution did the same but for the second to right tree (the second closest tree). You can notice that one of the branches in the second tree (near top middle of the image) that intersects with a branch in the first tree is slightly out of place. It goes behind the branch of the first tree in one location and comes out in a slightly different location since the Greedy solution couldn't account for the cumulative energy. The people in both images remained intact.

3.4.2 Decrease Height

Dynamic Programming Solution	Greedy Solution
	

There are significant differences between these two images. The Dynamic Programming solution shrinks the top of the right spire, while the Greedy solution does not. However, the Greedy solution shrinks the top of the domed building on the left, while the Dynamic Solution leaves it intact. The Greedy solution also shrinks the top of the grey spire in the middle of the image in an unnatural looking way. The Dynamic programming solution erases more of the sky, bringing the foliage at the top of the image further down in the image, closer to the top of the buildings. This is especially noticeable on the foliage in the top right of the image. The Greedy solution erases more of the foliage itself, leaving compressed looking leaves and branches. That being said, the Dynamic Programming solution does result in an unnatural curvature of the cloudline between the domed building and the grey tower which is not present in the Greedy solution.

Dynamic Programming Solution	Greedy Solution
Reduced Height (50px) 	Reduced Height (50px) 

The differences between these two images are much harder to notice than the previous examples. The Dynamic Programming solution deletes more of some of the strips of shadow in the grass, leaving thinner shadow patches than the Greedy solution. The Greedy solution has more seams cutting across the branches in the closest tree, making it appear slightly shorter. Most of the seams in the Dynamic Programming solution focus on the shadow covered grass, whereas most seams in the Greedy solution focus on the tree branches, although both solutions have some seams that focus on the grass and some that focus on the tree branches.

4. Sources Cited

1. Avidan and Shamir, 2007. “Seam Carving for Content-Aware Image Resizing”.