Martin Mayr, Vincent Christlein, Mathias Seuret, Thomas Köhler, Prathmesh Madhu

Panorama & Seam Carving

Deadline: November 22nd 2021

Download Stuff

Images will be uploaded on StudOn in the afternoon. Download them and and use them for the exercises (using your own data would also be possible).

Panorama Creation

For this exercise, use at first the low resolution photos of a Swedish harbor. Once it works, you might try it with a higher resolution, and/or your own photos.

Your panoramas will be composed of three photos. The second one, when sorted alphabetically, should be in the middle.

Exercise 2.1: Extract & match SIFT Keypoints and Descriptors

For each image, use `cv2.SIFT_create()` to extract SIFT keypoints and descriptors. I recommend to first convert the images to grayscale.

For the central image, make a copy of the points, and offset the copy to the right by a distance equal to the width of the image. We will use these as anchor points.

For the keypoints of all images, create a matching with the anchor points. You can use `cv2.BFMatcher`. As you will get many keypoints, you might want to keep only some of the best ones – you can sort them, from most similar to most different with

```
matches = sorted(matches, key = lambda x:x.distance)
```

Create numpy arrays out of your points; each row should be a point, and columns the coordinates.

Exercise 2.2: RANSAC Algorithm for Homography Computation

The following has to be done for each image. Select 4 random matches, and compute an homography using `cv2.findHomography`. Transform all points from the image, and compute their distance to the corresponding anchor points. Note that to use `cv2.perspectiveTransform`, you will have to reshape your arrays. Compute the median distance m, and consider as inlier all points closer than $\alpha \times m$, with, for example, $\alpha = 0.1$. Count the number of inliers. Repeat this process many times (e.g., 10000), and keep as optimal matches the largest set of inliers found. Compute the homography using this optimal set of points.

Martin Mayr, Vincent Christlein, Mathias Seuret, Thomas Köhler, Prathmesh Madhu

Exercise 2.3: Create the Panorama

Using the optimal homography found in Exercise 2.2, transform the three images with `cv2.warpPerspective`. As size, give three times the width of the central image, and its height.

You will then get three large images, but only some of the pixels will be non-black. For each of these, create masks with 0 for black pixels, and 1 for non-black pixels. Divide, pixel-wise, the sum of the images by the sum of the masks[1] to get the average image.

You can improve this by changing the weights of the pixels in the mask – giving less weight to the ones close to the edges will lead to a better blending of the images.

Crop the image automatically to remove the black areas which might appear on the left and right. To do so, you can sum the color channels, compute the minimum value of the columns, and look for the first and last non-zero values. They will correspond to where you have to crop the image.

Seam Carving: Energy Function

For this exercise, you will have to compute the energy function of an image. This will be done in three steps: first on a simple 2D array, then on a small 3D array mimicking a color image, and finally on a real image.

Exercise 2.4: Removing columns and rows

For this first exercise, you will have to compute an energy function for an image, and remove the rows and columns which have the smallest energy to downscale the image. First implement the method without mask, and try it on the provided common kreskel image, and replicate the result shown in the lecture. Then, apply it either to the kingfishers image, or to another image of your choice. You will notice that important elements get distorded. As a final step, you will have to implement the mask approach to preserve some elements, and apply it to the kingfishers picture, or any other of your choice.

Energy function

To compute the energy function, you can use the Laplacian operator:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{1}$$

It is also possible to use other functions with larger kernels; do not hesitate to try things out!

[1] Where the sum of the masks is non-zero.

You can convolve it on the image using `scipy.signal.convolve`. As the image has three channels, apply the filter on each of them, and make the sum of the results.

If the operator is applied only where it would fit on the image, then the result would have a smaller size[2]. A typical solution is to pad the input by appending rows and columns such that the output has the same size as the original input. You can either do it manually, or use `'same'` as `method` parameter of `convolve`.

Hint: you will compute energy functions many times – why don't you define a function for it? The same will apply to other parts of the code.

Deleting rows/columns

You can compute the cost of a row or a column by using `numpy.sum`. Use 0 or 1 as `axis` parameter to indicate whether the sum has to be made horizontally or vertically.

Find the index of the row or column to delete using `numpy.argmin`.

Hint: you can have a single implementation (removing either rows or columns), and rotate your data with `numpy.rot90`.

Hint: remove rows and columns on both the RGB and grayscale images, and update the cost array.

Using a mask

Load the mask as a grayscale image, and add it element-wise to the cost array. You will also need to resize this mask along with the image.

Apply the resizing method to the common kestrel picture, using the provided mask.

Optional: some optimizations

For later exercises, it will be beneficial (but not mandatory) to store the different arrays as multiple channels of a single array, as described during the lecture.

Exercise 2.5: path finding

You will find the array shown in the slides in the file `ex2array.py`.

Cumulative cost

Compute the cumulative cost as shown in the slides. You will have to process each row one after another, starting from the top.

You can compute the minimum filter either manually, or by using the function `scipy.ndimage.minimum_filter1d`. Note that you will have to be careful with boundary

[2] It would not be possible to compute the operator on the border of the image.

issues: for the left-most column, for example, you will have to compute the filter on only two elements.

hint: if you use `minimum_filter1d`, then you can set the `mode` parameter to `'reflect'`. Padding the input with zeros would cause issues, as the cost sum along the padded area would always be null.

Computing the path

As seen in the lecture, compute the path from bottom to top. Note that you will then have to inverse the list of coordinates which you get – using a stack or a deque would be appropriate.

Exercise 2.6: use seam carving

Still working on the same array, apply seam carving – that is, compute paths, and remove cells which are along the minimum cost paths.

Hint: use ':' when indexing the arrays to avoid using too many loops.

Exercise 2.7: seam carving on images

For this exercise, you will have to merge together the codes which you wrote in the previous exercises.

Base your code on what you wrote for Ex. 2.4. Replace the search of a row/column by the computation of the lowest cost path, and the deletion of rows/columns by the deletion of cells along the path.

As before, do not forget to update the energy function and cumulative cost after each image update.

Hint: it will be slow; you will maybe want to work on smaller scales of the images until your code works properly. I would suggest to start with images which largest dimension is lower than 500 pixels.

Hint: save frequently the resulting image (with different filenames) so that you can check what is happening. It will be useful for debugging your code.

Keep track of seams

As seen in the part of the lecture about upscaling images, it might be beneficial to avoid having too many seams at the same place. Keep track of the seams by adding values to the mask where the seams are.

Compare the seam carving of an image with, and without this process.

Hint: make sure that you do not lose these values that you added when resizing the mask. Saving the mask as an image and checking how it looks like can be useful for debugging purpose.

Martin Mayr, Vincent Christlein, Mathias Seuret, Thomas Köhler, Prathmesh Madhu

Make the kingfishers get closer

The kingfishers are sitting far from each others. Use seam carving to bring them closer.

How close can you make them before distortions become annoyingly visible?

Make it faster

Seam carving for resizing images is unfortunately slow. The two main parts which cause this are computing the energy function and the cumulative cost. However, when a path is removed, then we can assume it would not be necessary to update these arrays for the whole image, but only for the neighborhood of the path.

Compute the standard deviation σ of the path. Then, we can assume that, in most cases, neighbor paths would have similar standard deviations, so it is unlikely that erasing a path will have an impact on the cumulative cost array further than a few σ.

Thus, instead of updating the whole energy and cumulative cost arrays, you can resize them in the same way as the image, and update only the cells which are at a distance to the path inferior to $2 \cdot \sigma_r + 1$, where σ_r is σ rounded to the upper value.

Enjoy your new cruise speed.

Exercise 2.8: upscaling images

Upscaling images is very similar to downscaling them. Instead of deleting the cells along the path, insert some on the left of the path. This can be done by increasing the width of the arrays, and offsetting to the right by one pixel the values of all cells starting from the path. This leaves a gap to fill – use the average of the values on its left and right.

First, try to upscale the picture of Vincent Christlein on a cliff to make it twice larger. You should get a result similar to the one shown in the slides. Do not forget to make a mask so that you do not distort your teacher!

Then, try it on a few other images of your choice – either photos you took, or photos found on Internet.

Exercise 2.9: Compare nature landscales and cityscapes

Find or take a landscape and a citiscape. Use seam carving to resize them (downscale or upscale). Compare results obtained on these two kinds of images. Some parts of the images are rescaled without it being visible, while it's clear in other ones. Can you identify what kind of elements don't work well with seam carving?

Optional exercise 2.10: share an image

Apply heavy modifications to an image of your choice using seam carving, and share the results in the forum :-)

Optional exercise 2.11: share a video

Select a portrait photo, either of yourself or of a star you like/dislike. Apply heavy seam carving on it, saving all different scales. Then, resize all images to their original size (with a "normal" method, e.g., mogrify), and make a video out of them, as in the one given in the additional material.

Share the result in the forum so that everybody can enjoy it.

Exercise 2.12: Create & Carve a Panorama

Take three pictures to produce a panorama, if possible containing two nice elements that are far away. Protect them with a mask, and use seam carving to reduce the size of the panorama, so that these two elements get closer.

Optional, final exercise 2.13: Use Seam Carving in Panorama Creation

In Ex. 2.3, you merged the images together by using pixel averaging. It looks okay, but is not that good. You can do better with seam carving!

Consider two overlapping images. You want to find a cut between them that looks as nice as possible. In other words, you want to cut where the images are the most similar.

Attribute a moderately high cost to pixels which do not belong to both images[3]. For pixels belonging to both images, use the absolute or squared pixel differences as cost.

Compute the seam, and select pixels from the left or right image based on this seam. As the seam passes where there are very little differences between the images, we can assume that there won't be a clearly visible boundary where the images were stitched. This will look better than averaging pixel values.

If you do this exercise, please do show your result in the forum :-)

[3] In other words, pixels which do not belong to any, or to only one of the images.