utils

- open image (path-> np array)
- to grayscale (המרה לגווני אפור)
- normalize image (נרמול הערכים להיות בין 0 ל-1)
- get gradients (מקבל תמונה nparray ומחזיר גווני אפור 0-255 שמייצג גרדיאנטים)
- save images (מקבל מילון תמונות ושומר אותן בנתיב)

Seam Carving

Deadline: 11/4/2022 (23:55)

Introduction

Content-aware image resizing changes the image resolution while maintaining the aspect ratio of important regions. In this exercise, you will implement and explore image resizing using the seam carving algorithm and other basic image processing operations.

You are provided with a basic skeleton project which supports the following features:

- It resizes an input image using simple nearest-neighbor interpolation
- It outputs the image gradients

נתון

- nearest neighbors resizing
- image gradients

You need to extend the provided project and support the following features:

- Image resizing via basic seam carving algorithm with a given energy function
- Image resizing via seam carving algorithm with forward energy looking energy function

צריך:

- SC כשנתונה פונקציית האנרגיה
- SC עם forwarding

Your program should output the following:

- Resized image with the specified output dimension
- If seam carving method is chosen, you will also output visualization images with the chosen seams colored in red and black for horizontal and vertical seams, respectively. See next section for further details.

להוסיף קווים של ה-seams שנבחרו

We will use python as our primary programming language. Please refer to the instructions on moodle on how to setup a python project using PyCharm and Anaconda environment.

Seam Carving - Overview:

In this exercise we will implement the seam carving algorithm. In the basic algorithm you need to define an energy function that specifies the “importance” of each image pixel. You can calculate the pixel importance using the image gradient. There are several ways you can calculate the image gradient, one option is given below:

חישוב הגרדיאנטים - נתון ב- utils

Image Gradient:

To compute the image gradient, you can first convert the image into grayscale image (see utils.to_greysacale in the provided skeleton). Then the gradient magnitude $E(i,j)$ at pixel (i,j) can be defined as:

$$E(i,j) = \sqrt{\frac{\Delta_x^2 + \Delta_y^2}{2}}$$

where Δ_x^2 is the squared difference between the current and next horizontal pixel (in grayscale), and Δ_y^2 is the squared difference between the current and next vertical pixel (in grayscale). More specifically:

$$\Delta_x^2 = \left(I_{gs}[i,j] - I_{gs}[i,j+1]\right)^2$$
$$\Delta_y^2 = \left(I_{gs}[i,j] - I_{gs}[i+1,j]\right)^2$$

This function is already implemented for you (see utils.py). Go over it and understand the implementation.

Algorithm outline:

Reducing/Increasing the image width by k pixels:

שינוי רוחב תמונה ב-k פיקסלים

• התמונה האפורה היא תמונת העבודה ובסוף ממירים למקורית.

It is recommended to set the grayscale image as your working image. This implementation can be relevant when implementing the forward energy function.

- Compute the image energy function (image gradient) E as defined in the previous section
- For 1…k:
  - Use dynamic programming to find the optimal vertical seam by calculating the cost matrix M.
  - Find the actual seam by finding the smallest cost in the bottom row, then start going up on a path of minimal costs.
  - Remove the seam from the grayscale image.
  - Store the order and pixels removed in each iteration.
- To reduce image size by k pixels, remove all chosen seams from the original image.
- To enlarge by k pixels, duplicate all chosen seams from the original image.

1) חישוב הגרדיאנטים
(utils)

2) לכל 1,...,k:

• מחשבים עם DP את M
מטריצת העלויות

• מוצאים את
ה-seam עם העלות המינימלית

• מוחקים אותו מה-grayscale

• לשמור את הסדר והאינדקסים של מה שמחקנו

3) למחוק/לשכפל
מהתמונה המקורית

Reducing/Increasing the image height by k pixels:

שינוי גובה תמונה ב-k פיקסלים

To change the size in the height-dimension, you can rotate the image by 90 degrees counter-clockwise and apply the algorithm we outlined above on the rotated image. Once done, you can undo the rotation.

לסובב ב-90 נגד
כיוון השעון,
אלגוריתם של מקודם
ואז לסובב חזרה.

Reducing/Increasing the image width and height:

שינוי גובה ורוחב

There are many ways to change the image size in both dimensions (height and width). For simplicity, we will first change the image width, and only then change the image height.

מומלץ לשנות
רוחב ואז גובה

- The first selected cell at the bottom row, is the one with the minimal value.
- The next cell will be the one that accepts the following rule:

```
if(M_{i,j} == pixelEnergy(i,j) + $M_{i-1,j} + C_V(i,j)$)
        minX = j;
else if(M_{i,j} == pixelEnergy(i,j) + $M_{i-1,j-1} + C_L(i,j)$)
        minX = j-1;
else
        minX = j+1
```

• מוצאים את
ה-seam עם העלות המינימלית

Forward Looking Energy Function:

שיטה טובה יותר

Seam-carving can introduce artifacts to the resized images. To reduce these artifacts, you will need to implement the forward-looking energy function:

גרדיאנטים

חישוב ה-DP כך:

$$M(i,j) = \mathrm{E(i,j)} + \min\begin{cases} M(i-1,j-1) + C_L(i,j) \\ M(i-1,j) + C_V(i,j) \\ M(i-1,j+1) + C_R(i,j) \end{cases}$$

Where (recall I_{gs} is the grayscale image of the input):

תמונה גווני אפור

$$C_L(i,j) = \left|I_{gs}(i,j+1) - I_{gs}(i,j-1)\right| + \left|I_{gs}(i-1,j) - I_{gs}(i,j-1)\right|$$

$$C_V(i,j) = \left|I_{gs}(i,j+1) - I_{gs}(i,j-1)\right|$$

$$C_R(i,j) = \left|I_{gs}(i,j+1) - I_{gs}(i,j-1)\right| + \left|I_{gs}(i-1,j) + I_{gs}(i,j+1)\right|$$

Again, use dynamic programming to find the cost matrix M. We will set E(i,j) to the gradient function from previous section.

Once the matrix is found, you need find the optimal seam. Be careful when retrieving the seam from the final cost matrix. You need to “**reverse**” back the decisions that led to the final optimal seam cost. This can be found by:

שתצור ה-seam המינימלי

- Find the index of the optimal cost in the bottom row.
    - For every row from bottom to up:
        - Chose the direction that led to the current pixel cost
        - Go up in that direction, this will be the next seam pixel

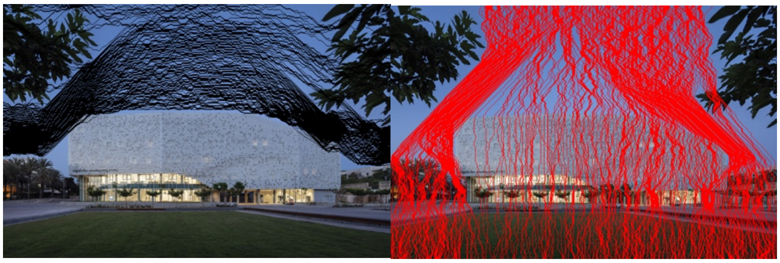
Seam Visualization:

הצגת התפרים

In addition to the resized image, you will need to return **two additional** images which visualize the chosen seams in each direction (vertical and horizontal). To do so, you will need to handle vertical seams and only then do horizontal seams. You need to create a copy of the original image (before resizing) and color the vertical seams with red. Then, handle the vertical seams and resize the image width. Finally, create a copy of the partially resized image and similarly color the horizontal seams; use black instead. See the illustration below (in this example, we didn’t perform resizing in both directions simultaneously):

הוספת שתי תמונות, אנכי אדום אופקי שחור.

1. יוצרים עותק של המקורית
2. לצבוע תפרים אנכיים באדום
3. לחתוך / לשכפל אנכיים
4. עותק של התמונה הערוכה חלקית
5. לצבוע תפרים אופקיים בשחור
6. לחתוך / לשכפל אופקיים



• פייתון 3.8
• מותר pillow, numpy

• ארגומנטים:
1) נתיב לתמונה
2) נתיב לתיקיית הפלטים
3) גובה רצוי
4) רוחב רצוי
5) שיטה (שכנים / sc)
6) האם רוצים forward או לא (bool)
7) אופציונלי - קידומת לשם הפלטים, דיפולטי img

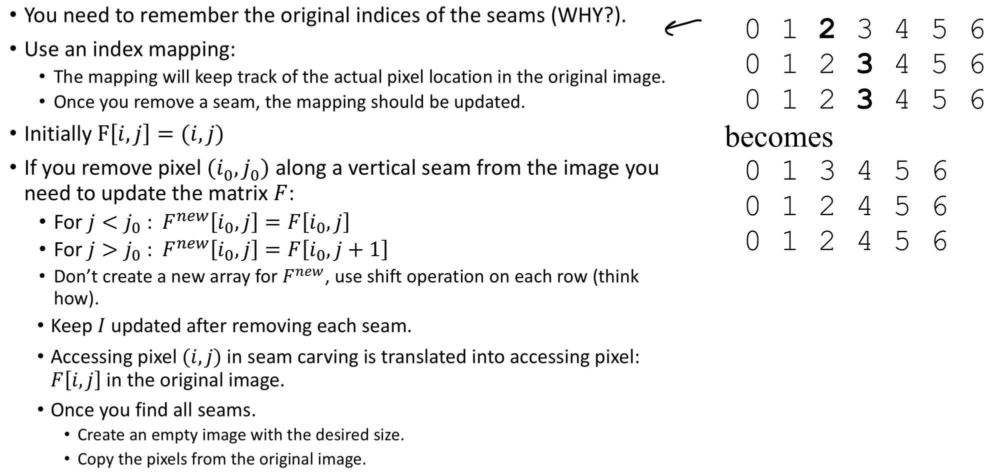
Requirements:

- Your solution should be implemented using Python (use python version 3.8)
- You may use the python packages (Pillow, numpy) – other libraries are prohibited.
- Your program should be a command-line application with the following options:
    - --image_path (str)– An absolute/relative path to the image you want to process
    - --output_dir (str)– The output directory where you will save your outputs.
    - --height (int) – the output image height
    - --width (int) – the output image width
    - --resize_method (str) – a string representing the resize method. Could be one of the following: [‘nearest_neighbor’,
      ‘seam_carving’]
    - --use_forward_implementation – a boolean flag indicates if forward looking energy function is used or not.
    - --output_prefix (str) – an optional string which will be used as a prefix to the output files. If set, the output files names will start with the given prefix. For seam carving, we will output two images, the resized image, and visualization of the chosen seems. So if --output_prefix is set to “my_prefix” then the output will be my_prefix_resized.png and my_prefix_horizontal _seams.png, my_prefix_vertical_seams.png. If the prefix is not set, then we will chose “img” as a default prefix.

Tips and Guidelines:

- When applying seam-carving, especially in the forward-looking implementation, make sure you work on a grayscale image. Once you find the seams, you can then handle them by removing/duplicating their corresponding pixels in the original image.
- When you remove a seam, all pixels to the right of it are shifted left by one. You will have to remember their original position. Use a helper array for that, e.g. an array with the size of the image, with the original column indices in each row:

לעבוד על
התמונה
האפורה
ואז על המקורית



- You need to remember the original indices of the seams (WHY?).
- Use an index mapping:
    - The mapping will keep track of the actual pixel location in the original image.
    - Once you remove a seam, the mapping should be updated.
- Initially $\mathrm{F}[i,j] = (i,j)$
- If you remove pixel (i_0, j_0) along a vertical seam from the image you need to update the matrix F:
    - For $j < j_0$: $F^{new}[i_0, j] = F[i_0, j]$
    - For $j > j_0$: $F^{new}[i_0, j] = F[i_0, j+1]$
    - Don’t create a new array for F^{new}, use shift operation on each row (think how).
    - Keep I updated after removing each seam.
    - Accessing pixel (i,j) in seam carving is translated into accessing pixel: $F[i,j]$ in the original image.
    - Once you find all seams.
        - Create an empty image with the desired size.
        - Copy the pixels from the original image.

0 1 **2** 3 4 5 6
0 1 2 **3** 4 5 6
0 1 2 **3** 4 5 6

becomes

0 1 3 4 5 6
0 1 2 4 5 6
0 1 2 4 5 6

להשתמש במערך עזר
את האינדקסים
המקוריים ולהציג

- At the sides (left/right), you don't have all three options for the costs. What you should do, is using the options you can.
  For example, at a leftmost pixel ($x = 0$), the cost would be:

  $$M_{i,0} = E(i,0) + \min \begin{cases} M_{i-1,0} + C_V(i,0) \\ M_{i-1,1} + C_R(i,0) \end{cases}$$

  להשתמש בערכים
  דיפולטיים בקצוות

  The case of the rightmost pixels is equivalent.
  Note, CV(i,0) is not defined in this case, use some default value (CV(i,0) = 255.0) to avoid boundary biases.
- At the top row ($y = 0$), the cost would be:

  מחיר בשורה הראשונה

  $$M_{0,j} = E(0,j)$$
- Make a small synthetic test image that would make it easy for you to debug (e.g. 5x5).

  לבדוק על תמונה
  קטנה 5x5
- Useful functions: numpy.rot90, numpy.zeros_like, numpy.zeros, numpy.copy, np.roll, np.concatenate, np.vstack, np.hstack
- You may use utility functions given in utils.py
- Avoid sequential access to the arrays, and use whole-row and whole-matrix operations (see the presentation slides 17-21 for insights)

  להשתמש בפעולות שורה ומטריצה
  ולא אחד אחד

  1. Calculate C_R, C_V, C_L using matrix operations, without double for-loops.
  2. When handling row i, you can calculate the values:

     $$M(i-1, j-1) + C_L(i,j)$$
     $$M(i-1, j) + C_V(i,j)$$
     $$M(i-1, j+1) + C_R(i,j)$$

     without a for-loop - think how.
  3. See how you can perform shift-left efficiently using indexing and assignments (slide 17)

Bonus

Real-time applications in computer vision are essential. Therefore, we will give bonus points for submissions with fast implementation. Optimize your code so it will be as fast as possible. The bonus will be given to the top-k performing submissions (k will be determined later).

בונוס ל-k הפותרים הכי יעילים (מהירות).

Submission Guidelines:

הגשת קובץ במודל

1. Submission is in pairs.
2. Submit your zip files through the Moodle site of the course.
3. Submit **.zip** file titled ex1_<id1>_<id2>.zip, where <id1> and <id2> are your ids
    a. Make sure your main.py file is in the top-directory inside of the zip file. This means the zip file structure should be something like:

    ```
    /
    |    main.py
    |    utils.py
    |    seam_carving.py
    |    nearest_neighbor.py
    |__ /directory1
    |     |    file1.py
    |     |    file2.py
    |     |__________
    |
    |__ /directory2
          |    file1.py
          |    file2.py
          |__________
    ```

    עם תיקיות

    b. You don't need to add new files or directories if you don't want to.
4. Your code should run in a reasonable time: for example, scaling an image of size 600x600 by 50% in each dimension should take no more than 5 minutes.

זמן סביר

Change Log (22/3/2022)

[1] Use the gradient as pixel energy in the forward looking cost matrix
[2] The gradient function was updated in utils.py, and minor issue was fixed in main.py. Use the new implementation.
[3] Added presentation from Class (with modified slides)
[4] Tips and guidelines were updated: added edge-cases and hints and tips on how to improve your implementation.
[5] Runtime complexity requirement was relaxed a-bit.
[6] Bonus section was added
[7] Added a new directory with sample outputs: the output shapes are part of the filesnames.

Feel free to email me for any question: moabarar@mail.tau.ac.il

Good Luck!

“Ex1”

HW1 – Overview

- Implement Seam Carving
    - Basic energy formula
    - Forward-looking formula
- Visualize chosen seams:
    - Both vertical and horizontal
- You are provided with a skeleton project:
    - Nearest neighbor interpolation
    - You are provided with different utility functions that you can use
- We will use Python (version 3.8)
    - You are provided with setup instructions (pycharm + anaconda)
    - Note in the instructions, python 3.7 is used (change to 3.8)
    - Credits: Ben Maman
- You may only use **numpy** and **pilllow** packages

Seam Carving – Vertical Seams

Steps:

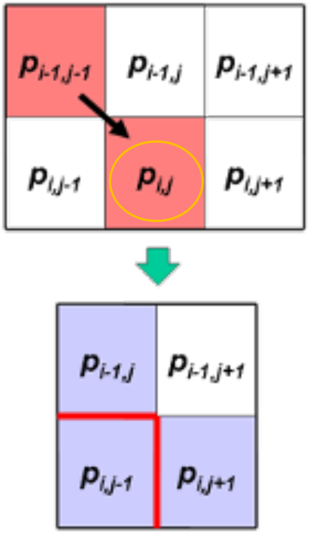
1. Find seams while image not in the desired size
    1. Calculate cost matrix
    2. Find optimal seam (dynamic programming)
    3. Remove the seam (remember the original indices of the removed seam)
2. In case of downsizing:
    1. Remove the chosen seams.
3. In case of upsizing:
    1. Replicate the chosen seams.

Calculating cost matrix - Vertical Seams

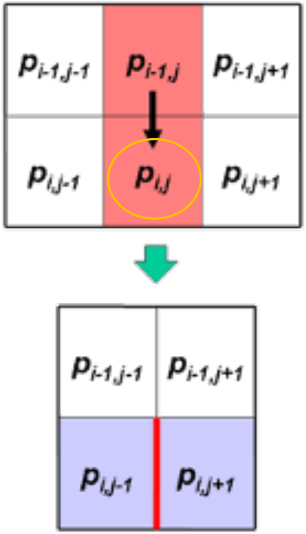
$M_{i,j}$ is the cost at row i column j

$p_{i,j}$ is the greyscale image intensity

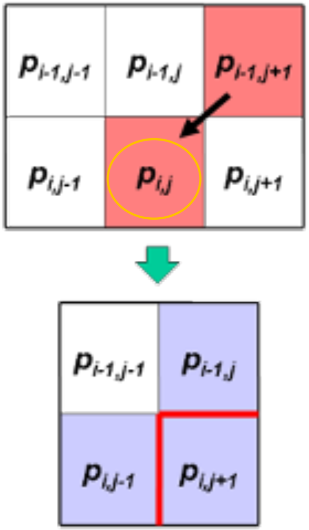
$$M_{i,j} = pixelEnergy(i,j) + \min \begin{cases} M_{i-1,j-1} + C_L(i,j) \\ M_{i-1,j} + C_V(i,j) \\ M_{i-1,j+1} + C_R(i,j) \end{cases}$$



C_L



C_V



C_R

Calculating pixel’s energy

$p_{i,j}$ is the greyscales image intensity

$$E_{Vertical}(i,j) := \begin{cases} |p_{i,j} - p_{i,j+1}|, & if \quad j < width - 1 \\ |p_{i,j} - p_{i,j-1}|, & otherwise \end{cases}$$

$$E_{Horizontal}(i,j) := \begin{cases} |p_{i,j} - p_{i+1,j}|, & if \quad i < height - 1 \\ |p_{i,j} - p_{i-1,j}|, & otherwise \end{cases}$$

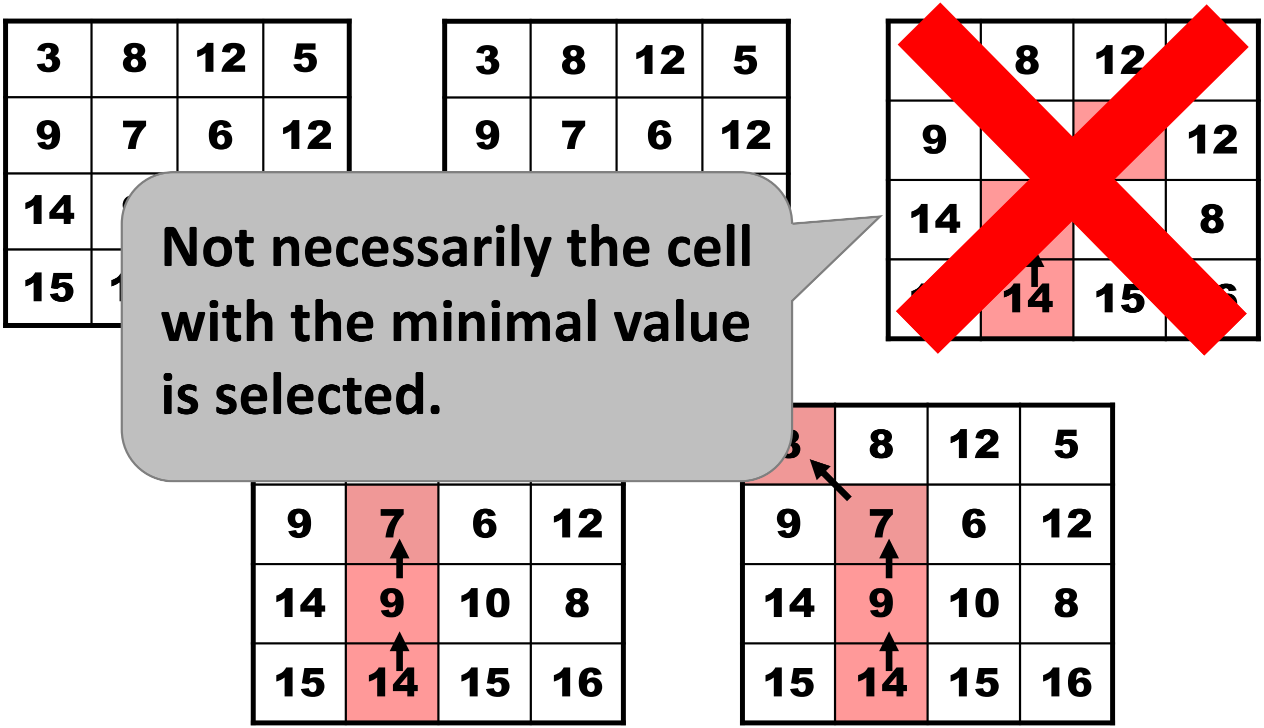
Calculating pixel’s energy

$p_{i,j}$ is the greyscales image intensity

$$pixelEnergy(i,j) \\ = \sqrt{0.5 * (E_{Vertical}(i,j))^2 + 0.5 * (E_{Horizontal}(i,j))^2}$$

Backtracking the best seam (inverting the formula)

Costs matrix M:



Not necessarily the cell with the minimal value is selected.

3	8	12	5
9	7	6	12
14	9	10	8
15	14	15	16

Backtracking the best seam (inverting the formula) - Vertical Case

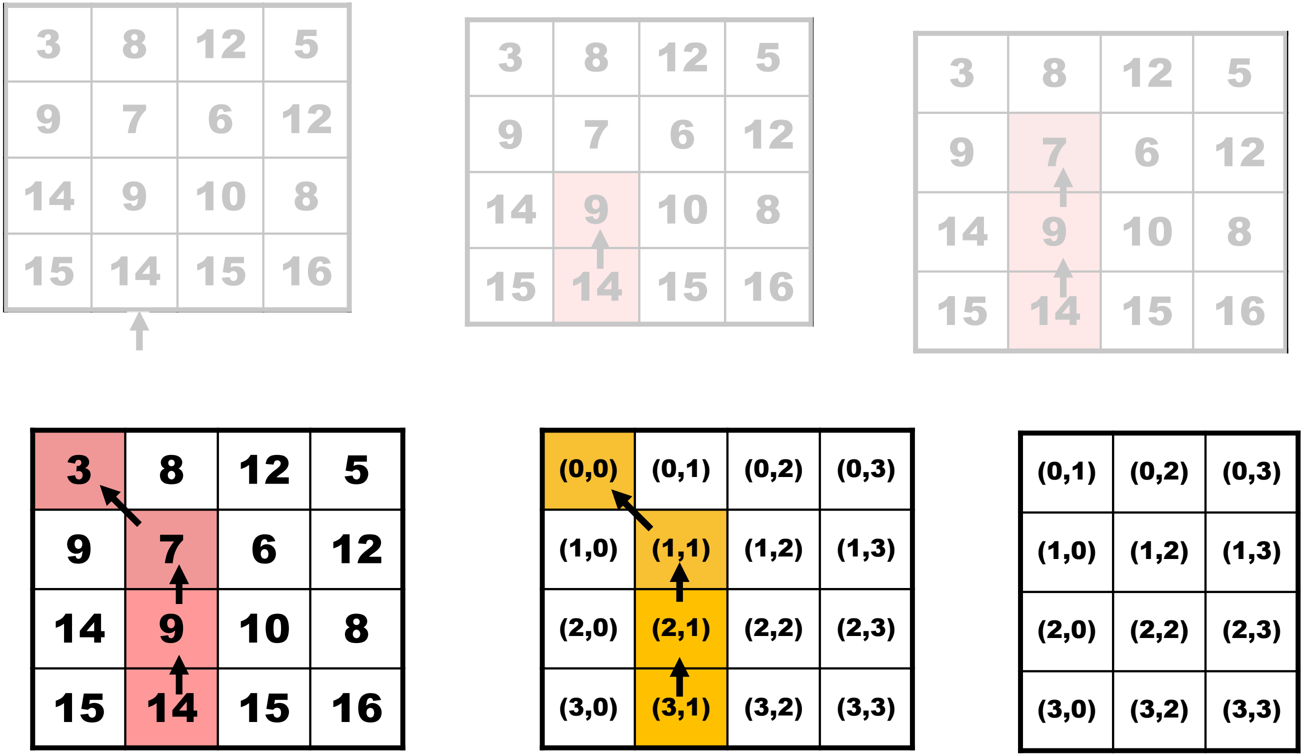
- The first selected cell at the bottom row, is the one with the minimal value.
- The next cell will be the one that accepts the following rule:

```
if($M_{i,j}$ == pixelEnergy(i,j) + $M_{i-1,j} + C_V(i,j)$)
        minX = j;
else if($M_{i,j}$ == pixelEnergy(i,j) + $M_{i-1,j-1} + C_L(i,j)$)
        minX = j-1;
else
        minX = j+1
```

Remembering the indices - Vertical Case

- You need to remember the original indices of the seams (WHY?).
- Use an index mapping:
  - The mapping will keep track of the actual pixel location in the original image.
  - Once you remove a seam, the mapping should be updated.
- Initially $\mathrm{F}[i, j] = (i, j)$
- If you remove pixel (i_0, j_0) along a vertical seam from the image you need to update the matrix F:
  - For $j < j_0$: $F^{new}[i_0, j] = F[i_0, j]$
  - For $j > j_0$: $F^{new}[i_0, j] = F[i_0, j + 1]$
  - Don’t create a new array for F^{new}, use shift operation on each row (think how).

Remembering the indices - Vertical Case



3	8	12	5
9	7	6	12
14	9	10	8
15	14	15	16

3	8	12	5
9	7	6	12
14	9	10	8
15	14	15	16

3	8	12	5
9	7	6	12
14	9	10	8
15	14	15	16

3	8	12	5
9	7	6	12
14	9	10	8
15	14	15	16

(0,0)	(0,1)	(0,2)	(0,3)
(1,0)	(1,1)	(1,2)	(1,3)
(2,0)	(2,1)	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	(3,3)

(0,1)	(0,2)	(0,3)
(1,0)	(1,2)	(1,3)
(2,0)	(2,2)	(2,3)
(3,0)	(3,2)	(3,3)

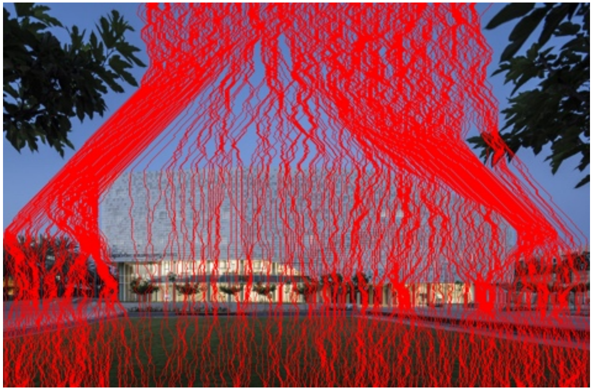
Remembering the indices - Vertical Case

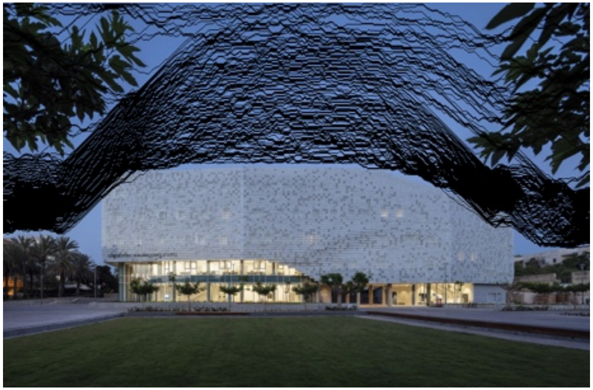
- Keep I updated after removing each seam.
- Accessing pixel (i, j) in seam carving is translated into accessing pixel: $F[i, j]$ in the original image.
- Once you find all seams.
    - Create an empty image with the desired size.
    - Copy the pixels from the original image.

Seam Carving – Horizontal case

- We saw how to handle vertical seams (change image width)
- Horizontal case - very similar
- To avoid code duplication:
  - Rotate image 90 degrees **C**ounter **C**lock-**W**ise (CCW)
  - Handle vertical seams on rotated image
  - Undo rotation - rotate image 90 **C**lock-**W**ise (CW)
- Note - there are many ways to change image dimension:
  - Handle width resizing
  - Rotate and handle height resizing
  - Rotate back

Seam Visualization





Python

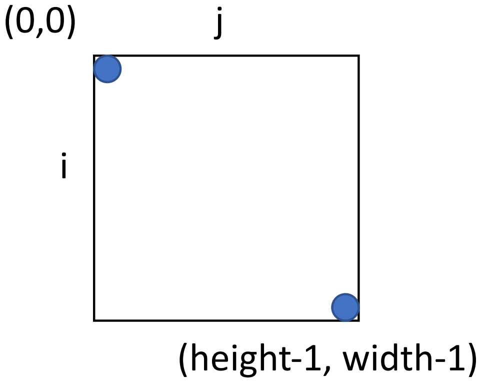
The basics of image IO and manipulation

Open Image with Pillow

- pil_image = PIL.Image.open(image_path)
- numpy_image = np.array(pil_image,
  dtype=**np.float32**)
  - We will represent the pixels using floats so that we will be able to perform divison (relevant for the pixel energy formula)
- numpy_image.shape:
  - Gives the dimensions of the image, for an RGB image, this should return the following tuple - (H,W,3) where H and W are height and with of the image

Working with images

- Access pixel values with
  - numpy_image[i,j,:]
- Rotating an image 90 Degrees CCW
  - np.rot90(numpy_image, k=1, axes=(0,1))
- Rotating an image 90 Degrees CW
  - np.rot90(numpy_image, **k=-1**, axes=(0,1))
- Copy an image:
  - image_copy = numpy_image.copy()
- Create a zero_like copy of an image:
  - zeros_like = np.zeros_like(numpy_image)
  - zeros_like will have the same shape and dtype as `numpy_image, but all values are zero.`
- Create a new zeros array:
  - np.zeros((H,W,3), dtype=np.float32)



Working with images – efficient imeplmentation

- Efficient implementation of shifting an array at index j:
    - **Fast:** A[j:-1] = A[j+1:]
    - **Slow:** Equivilant to:
      for idx in range(j, A.shape[0]-1):
        - A[idx] = A[idx+1]
    - Example, A=[1,2,3,4,5,6], j=2, then the array becomes: [1,2,4,5,6,6]

- Works for 2D arrays as well (images):
    - A[:, j:-1] = A[:, j+1:] # Along axis=1, i.e. width
    - A[j:-1, :] = A[j+1:, :] # Along axis=0, i.e. height

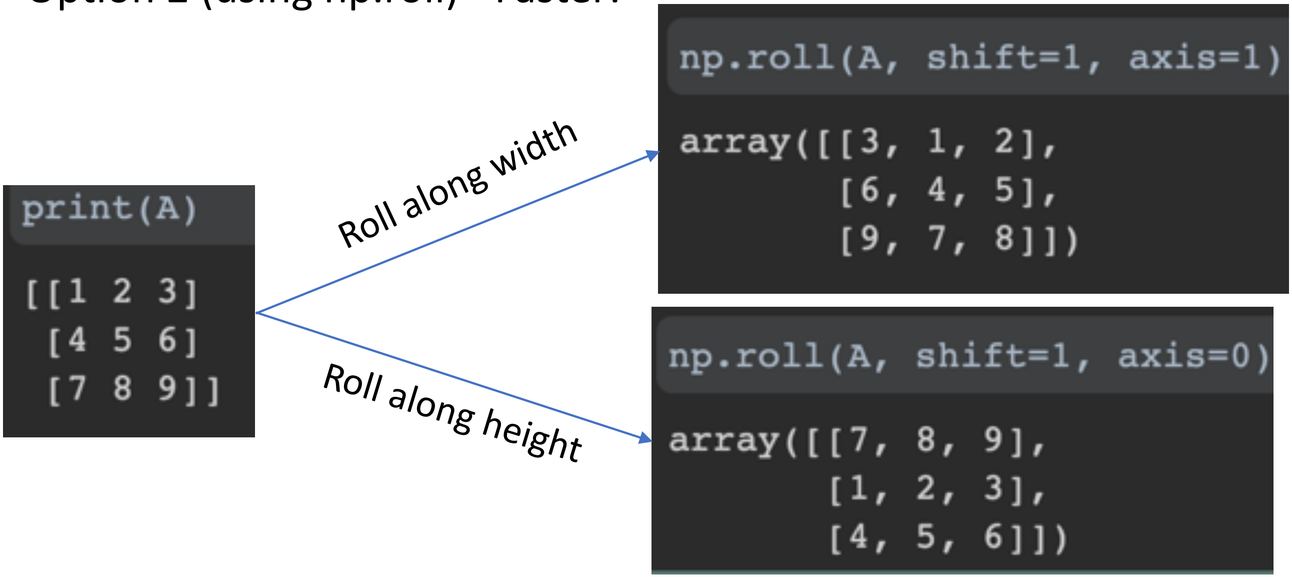
Working with images

- Let us try to calculate the difference between the left and right neighbor of a pixel (i,j):
  - Output[i,j] = |A[i,j-1] – A[i,j+1]|
  - If an elemnt is out of boundry just set to zero.
- Option 1 (very slow):

  -
    ```
    for i in range(A.shape[0]): #i = 0…, H-1
        for j in range(A.shape[1]): #j = 0,…,W-1
            left = 0 if j == 0 else A[i,j-1]
            right = 0 if j == A.shape[1] -1 else A[i,j+1]
            Output[I,j] = abs(A[i,j-1] – A[i,j+1])
    ```

Working with images

- Let us try to calculate the difference between the left and right neighbor of a pixel (i,j):
  - Output[i,j] = |A[i,j-1] – A[i,j+1]|
  - If an elemnt is out of boundry just set to zero.
- Option 2 (using np.roll) - Faster:



```
print(A)

[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

```
np.roll(A, shift=1, axis=1)

array([[3, 1, 2],
       [6, 4, 5],
       [9, 7, 8]])
```

```
np.roll(A, shift=1, axis=0)

array([[7, 8, 9],
       [1, 2, 3],
       [4, 5, 6]])
```

Working with images

- Let us try to calculate the difference between the left and right neighbor of a pixel (i,j):
  - Output[i,j] = |A[i,j-1] – A[i,j+1]|
  - If an elemnt is out of boundry just set to zero.
- Option 2 (using np.roll) – Faster:

  -
    ```
    left = np.roll(A, shift = 1, axis=1)
    left[:,0] = 0 # Zero the first column in array
    ### Now left[i,j] = A[i,j-1] if j > 0 else 0
    right = np.roll(A, shift = -1, axis=1)
    right[:, -1] = 0 # Zero the last column in array
    ### Now right[i,j] = A[i,j+1] if j < W-1 else 0
    Output = np.abs(left-right)
    ```

Working with images

- Let us try to calculate the difference between the left and right neighbor of a pixel (i,j):
  - Output[i,j] = |A[i,j-1] – A[i,j+1]|
  - If an elemnt is out of boundry just set to zero.
- Option 3 (using np.concatenate – self read) – Fastest:

  -
    ```
    zero_column = np.broadcast_to([0.], [A.shape[1], 1])
    left = np.concatenate([zero_column, A[:, 0:-1]], axis=1)
    right = np.concatenate([A[:, 1:], zero_column], axis=1)
    output = np.abs(left-right)
    ```

Working with images

- Let us compare the two implementations:
  - Implementation 1 on input 100X100: 6.02 ms
  - Implementation 2 on input 100x100: 44.7 μs
  - ~x120 times faster!
  - Implementation 3 on input 100x100: 33.1 μs (30% faster)

Save Image with Pillow

- img_data = img_data.astype(np.uint8)
  - Pixels are stored as 3-bytes (RGB)
- pil_img= PIL.Image.fromarray(img_data)
  - Convert from numpy to PIL
- pil_img.save(path)
  - Save image using pillow

Exercise 1:
Any other questions?

moabarar@mail.tau.ac.il