COMP 3317 Computer Vision

Tutorial 5

TA: Hao Shaozhe

In This Tutorial

- Reminders & Hints on Assignment 2

Assignment - 2

- Topic
  - **Digital Image Processing** & **Feature Extraction** (lecture 2 & 3)
  - A **Programming** assignment
- Submission
  - Please submit **a ZIP file** to Moodle by the deadline.
    - A Python source code file (assign2.py)
    - A plain text file (readme.txt) describing the features you have implemented, especially when you have turned in a partially finished implementation.
  - Release: **22 February 2024 (Thu)**
  - Deadline : **23:59, Mar 6, 2024 (Wed)**
  - We **DO NOT accept** any late submission.

- Download assignment sheets / files

Assignment 2 (Deadline: 23:59, Mar 6, 2024)

Available from **22 February 2024, 4:30 PM** (hidden otherwise)

Details of Assignment 2:

- Deadline: 23:59, Mar 6, 2024 (Wed)
- Assignment sheet: <download here>
- Program template: <download here>
- Sample output: <download here>
- Sample image: <download here>

- If you have any **questions** about Assignment 2, we encourage you to post your questions in the corresponding **discussion forum** on HKU Moodle.

Assignment 2 discussion forum

Available from **22 February 2024, 4:30 PM** (hidden otherwise)

❖ Task

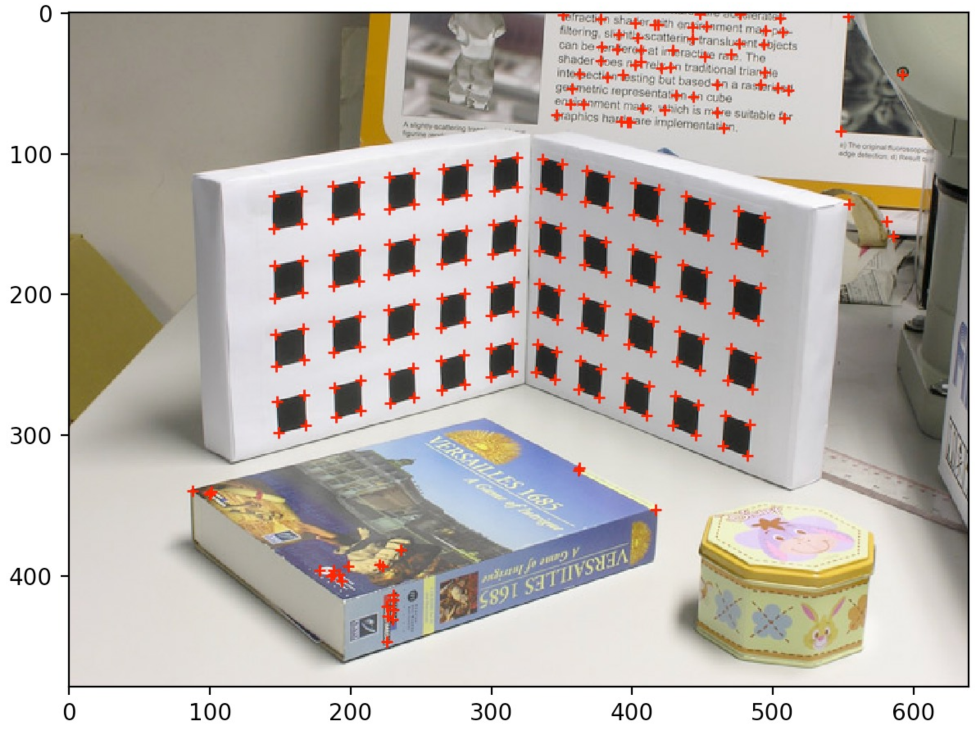
- In this assignment, you are going to implement the **functions** for performing
  1) color-to-grayscale image conversion
  2) corner detection

- To guide your coding, a **partially completed** Python program is provided to you, which provides implementations for
  - parsing the program arguments,
  - loading an input image,
  - plotting the corner detection result,
  - loading/saving the detected corners from/to a file.

❖ Task

- In completing this assignment, you only need to modify the following functions:

  1) **rgb2gray()**
  2) **smooth1D()**
  3) **smooth2D()**
  4) **harris()**



- Please refer to the tutorial notes as well as the comments in the source code for details of these functions.

- You can compare your result against the **sample output** (corners.lst) for checking the correctness of your program.

❖ Requirements

- Use the formula for the Y-channel of the **YIQ model** in performing the **color-to-grayscale** image conversion.
- Compute **I_x and I_y** correctly by finite differences.
- Construct images of **I_x^2, I_y^2, and $I_x I_y$** correctly.
- Compute a **proper filter size** for a Gaussian filter based on its **sigma** value.
- Construct a proper **1D Gaussian filter**.
- **Smooth a 2D image** by convolving it with two 1D Gaussian filters.
- Handle the image border using **partial filters** in smoothing.
- Construct an image of the **cornerness function R** correctly.
- Identify potential corners at **local maxima** in the image of the cornerness function R.
- Compute the cornerness value and coordinates of the potential corners up to **sub-pixel accuracy** by **quadratic approximation**.
- Use the **threshold** value to identify strong corners for output.

❖ Requirements

- **Use the formula for the Y-channel of the YIQ model in performing the color-to-grayscale image conversion.**
- Compute I_x and I_y correctly by finite differences.
- Construct images of I_x^2, I_y^2, and $I_x I_y$ correctly.
- Compute a proper filter size for a Gaussian filter based on its sigma value.
- Construct a proper 1D Gaussian filter.
- Smooth a 2D image by convolving it with two 1D Gaussian filters.
- Handle the image border using partial filters in smoothing.
- Construct an image of the cornerness function R correctly.
- Identify potential corners at local maxima in the image of the cornerness function R.
- Compute the cornerness value and coordinates of the potential corners up to sub-pixel accuracy by quadratic approximation.
- Use the threshold value to identify strong corners for output.

✓ **rgb2gray()**

➢ rgb2gray()

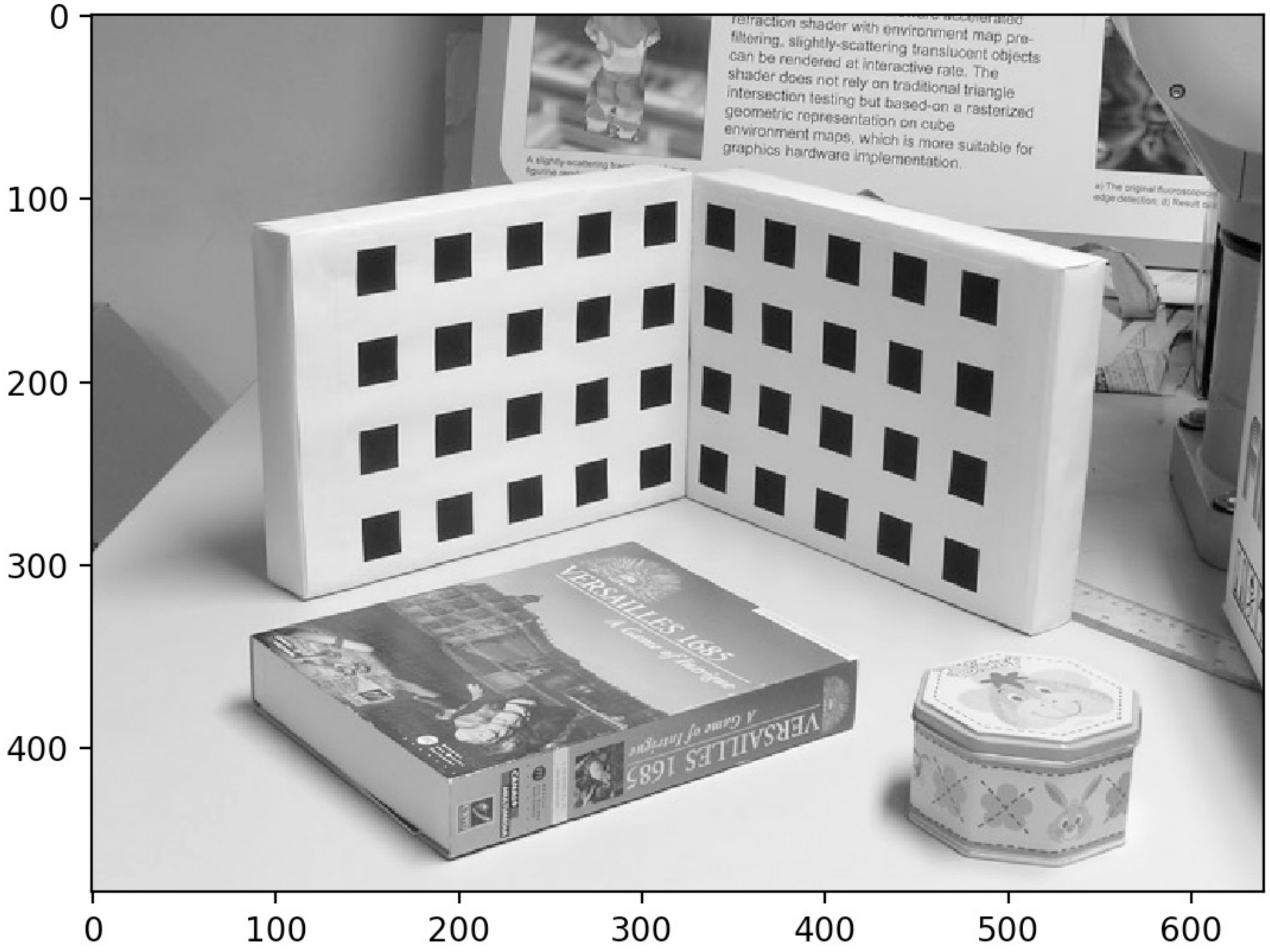
- RGB to YIQ conversion:

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.275 & -0.321 \\ 0.212 & -0.523 & 0.311 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

- A color image can be easily converted into a monochrome image by taking only the Y component (iluminance) of the above conversion

$$\boxed{I(i,j) = 0.299 \times R(i,j) + 0.587 \times G(i,j) + 0.114 \times B(i,j)}$$

- If you implement rgb2gray() correctly, you may get a similar output as follows:



Requirements

- Use the formula for the Y-channel of the YIQ model in performing the color-to-grayscale image conversion.
- Compute I_x and I_y correctly by finite differences.
- Construct images of I_x^2, I_y^2, and $I_x I_y$ correctly.
- **Compute a proper filter size for a Gaussian filter based on its sigma value.**
- **Construct a proper 1D Gaussian filter.**
- **Smooth a 2D image by convolving it with two 1D Gaussian filters.**
- **Handle the image border using partial filters in smoothing.**
- Construct an image of the cornerness function R correctly.
- Identify potential corners at local maxima in the image of the cornerness function R. Compute the cornerness value and coordinates of the potential corners up to sub-pixel accuracy by quadratic approximation.
- Use the threshold value to identify strong corners for output.

✓ **smooth1D()**

✓ **smooth2D()**

➢ smooth1D(), smooth2D()

- In this assignment, we are going to detect corners in an image with implementing Harris Corner Detection Algorithm.
  - In this algorithm, we are going to make use of a **smoothed** image.
  - In smoothing an image, we need to apply a **Gaussian Filter**.
  - That's why we should complete **smooth1D()** and **smooth2D().**
- Perform smooth2D() by calling smooth1D() twice.
  - In smoothing an image, it involves a 2D convolution. But convolving with a 2D Gaussian Kernel is computation expensive. (lecture 3, pp.31-32)

$$G_\sigma(x,y) * I(x,y) = g_\sigma(x) * \left[g_\sigma(y) * I(x,y)\right]$$

  - A better approach to perform 2D smoothing is using **two 1D convolutions**. Therefore, forming a 1D Gaussian Kernel is required.

❖ smooth1D()

- 1D Gaussian Filter

⬇

- Proper filter size – sigma

⬇

- Convolve the image with the filter

⬇

- Deal with the borders

➢ 1D Gaussian Filter

- Gaussian Filter is a class of low-pass filters which is based on the Gaussian Probability Distribution Function.
- Gaussian Distribution is actually a **Normal Distribution**.

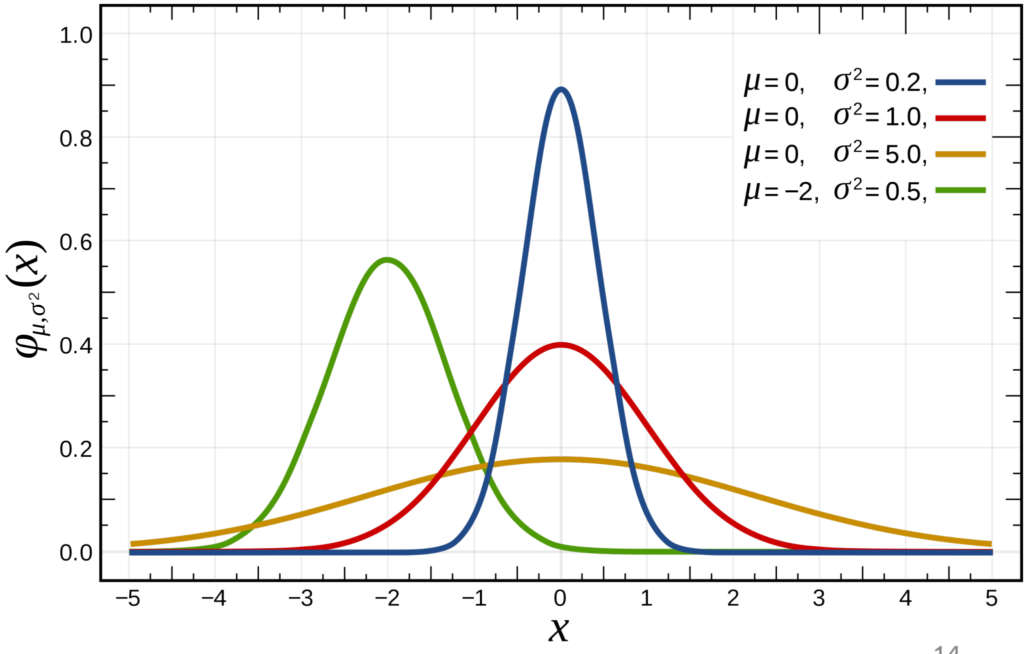
$$g(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\right).$$

Here $\boldsymbol{\mu}$ is the mean value, gives the location of peak;

And $\boldsymbol{\sigma}$ is the variance, controls how wide the peak is.

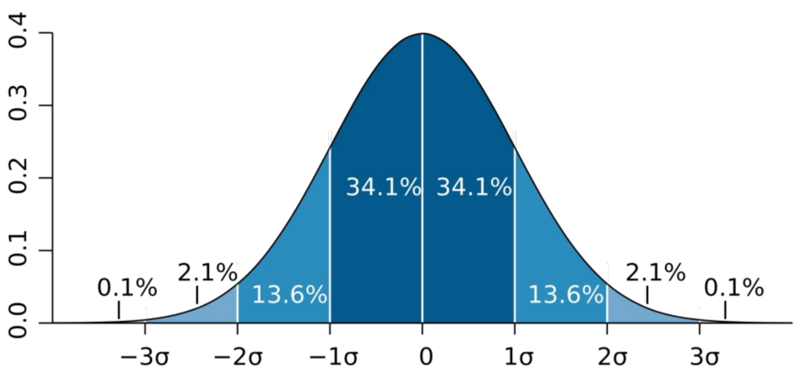
The bigger σ is, the more we smooth the image.

✓ To form a Gaussian filter, we will set $\boldsymbol{\mu} = 0$, because we want each pixel to be the one that has the biggest effect on its new, smoothed value.



➢ 1D Gaussian Filter

- So, the Gaussian function will be
  - 1D Gaussian function --- $g_\sigma(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma^2}}$
  - 2D Gaussian function --- $G_\sigma(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$



- In practice, both the image and the kernel are discrete quantities, and the convolutions are performed as **truncated summations**
- The size of smooth kernel is defined by **2n+1** (i.e., an odd number).
- For acceptable accuracy, kernels are generally truncated so that the discarded samples are less than **1/1000 of the peak value** (lecture 3, pp. 31)

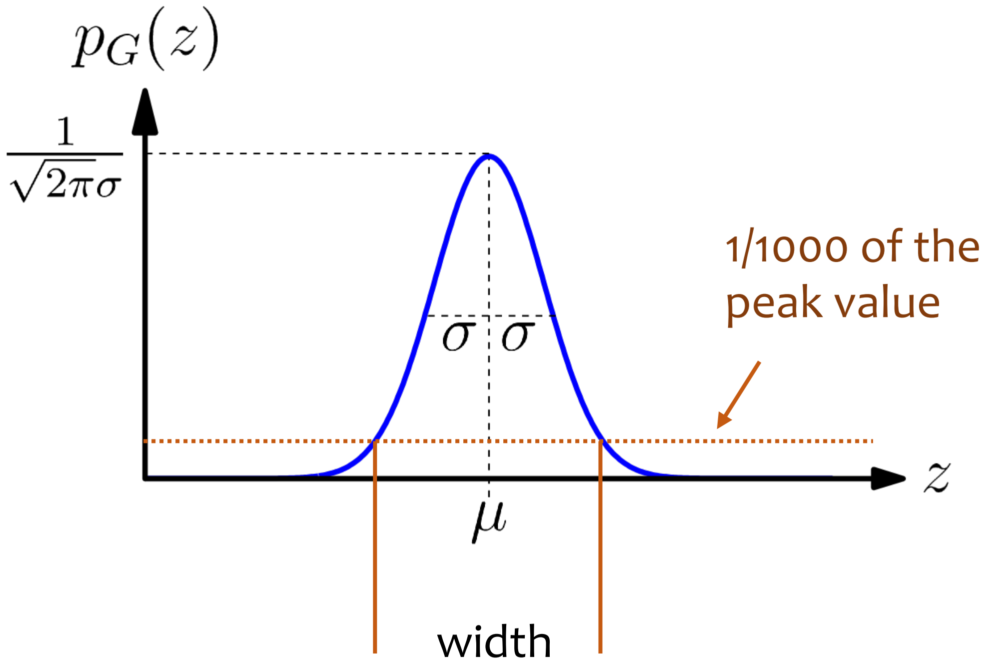
➢ Proper filter size – sigma

$$\frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x^2}{2\sigma^2}} \leq \left[\frac{1}{\sigma\sqrt{2\pi}}\right] \times \frac{1}{1000}$$

$$x^2 \geq 2\sigma^2 \ln 1000$$

$$|x| \geq \sigma\sqrt{2 \ln 1000}$$

➡ x in this range should be truncated out.



➢ 1D Gaussian Filter

- To form a 1D Gaussian filter:

$$g_\sigma(x) = \boxed{\frac{1}{\sigma\sqrt{2\pi}}} e^{-\frac{x^2}{2\sigma^2}}$$

Normalization Constant (It is set so that the area under the curve would be 1.)

✓ we can **ignore the normalization constant** since we will normalize the output later.

1D Gaussian Filter

- Example

```
import numpy as np
```

```
# find proper filter size
sigma = 1
n = int(sigma * (2*np.log(1000))**0.5)
print(n)
```

```
# form a kernel with the proper size
x = np.arange(-n, n + 1)
print(x)
```

```
# create a 1D Gaussian filter
filter = np.exp((x ** 2) / -2 / (sigma ** 2))
print(filter)
```

- Output

```
3
```

```
[-3 -2 -1 0 1 2 3]
```

```
[0.011109 0.13533528
0.60653066 1. 0.60653066
0.13533528 0.011109 ]
```

➢ convolve the image with the filter

- Don’t forget to normalize!

```
# Normalize the filter
filter /= filter.sum()
print(filter)
```

```
[0.00443305 0.05400558 0.24203623 0.39905028 0.24203623 0.05400558 0.00443305]
```

- Convolve the image with the filter

```
# get the smoothed result
from scipy.ndimage import convolve1d
result = convolve1d(img, filter, 1, np.float64, 'constant', 0, 0)
```

convolve1d

- **Input :** *array_like*

  The input array.

- **Weights :** *ndarray*

  1-D sequence of numbers.

- **Axis :** *int, optional*

  The axis of *input* along which to calculate. Default is -1.

- **Output :** *array or dtype, optional*

  The array in which to place the output, or the dtype of the returned array. By default an array of the same dtype as input will be created.

- **Cval :** *scalar, optional*

  Value to fill past edges of input if *mode* is 'constant'. Default is 0.0.

- **Origin :** *int, optional*

  Controls the placement of the filter on the input array's pixels. A value of 0 (the default) centers the filter over the pixel, with positive values shifting the filter to the left, and negative ones to the right.

➢ Reference: Check the document for convolve1d

- **Mode** *{'reflect', 'constant', 'nearest', 'mirror', 'wrap'}, optional*

  The *mode* parameter determines how the input array is extended beyond its boundaries. Default is 'reflect'. Behavior for each valid value is as follows:

  - 'reflect' (*d c b a* | *a b c d* | *d c b a*)

    The input is extended by reflecting about the edge of the last pixel. This mode is also sometimes referred to as half-sample symmetric.

  - 'constant' (*k k k k* | *a b c d* | *k k k k*)

    The input is extended by filling all values beyond the edge with the same constant value, defined by the *cval* parameter.

  - 'nearest' (*a a a a* | *a b c d* | *d d d d*)

    The input is extended by replicating the last pixel.

  - 'mirror' (*d c b* | *a b c d* | *c b a*)

    The input is extended by reflecting about the center of the last pixel. This mode is also sometimes referred to as whole-sample symmetric.

  - 'wrap' (*a b c d* | *a b c d* | *a b c d*)

    The input is extended by wrapping around to the opposite edge.

➢ Reference: [Check the document for convolve1d]

- Use scipy.ndimage function **convolve1d** to conduct 1D conv for smoothing

```
from scipy.ndimage import convolve1d
res = convolve1d([2, 8, 0, 4, 1, 9, 9, 0], weights=[1, 3])
print(res)
```

✓ 1.8s

```
[14 24  4 13 12 36 27  0]
```

- Before convolving the image with the 1D Gaussian Filter, you have to decide **how to handle the image border**.
    - ✓ In our Assignment 2, **partial filter** is applied

- However, we do not have an option for partial filters in convolve1d(). Therefore, we need some ways to achieve so.
    - You can either choose to deal with the borders separately, or
    - Create a matrix whose elements are all ones, and apply the Gaussian Filter to the matrix.
        - each element in this matrix represents a weight.
        - divide the matrix of smoothed image by the matrix of weight.
        - You can skip the normalization for filter if doing so.

➢ Explanation

12	16	18
13	11	30
15	15	16

Original image

$\frac{1}{3}$

1	1	1

A 1x3 filter
(Normalized)

14	15.3	17
12	18	20.5
15	15.3	20.5

Filtered image
(Normalized)

1	1	1
1	1	1
1	1	1

1	1	1

A 1x3 filter
(Un-normalized)

28	46	34
24	54	41
30	46	31

Filtered image
(Un-normalized)

2	3	2
2	3	2
2	3	2

Weight
(Un-normalized)

$\div$

1. Create a matrix whose elements are **all ones**

2. convolve1D() with the filter by **zero-padding**

3. **Divide** the matrix of smoothed image by the matrix of weight.

❖ smooth1D()

- 1D Gaussian Filter

- Proper filter size – sigma

- Convolve the image with the filter

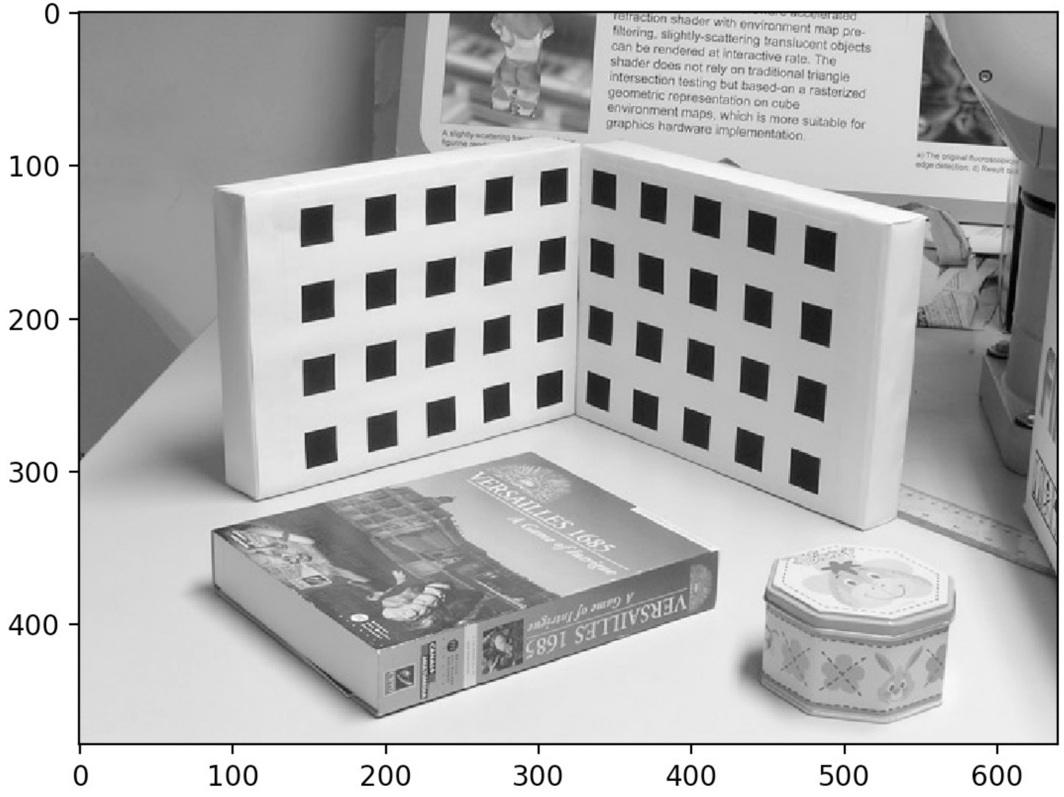
- Deal with the borders

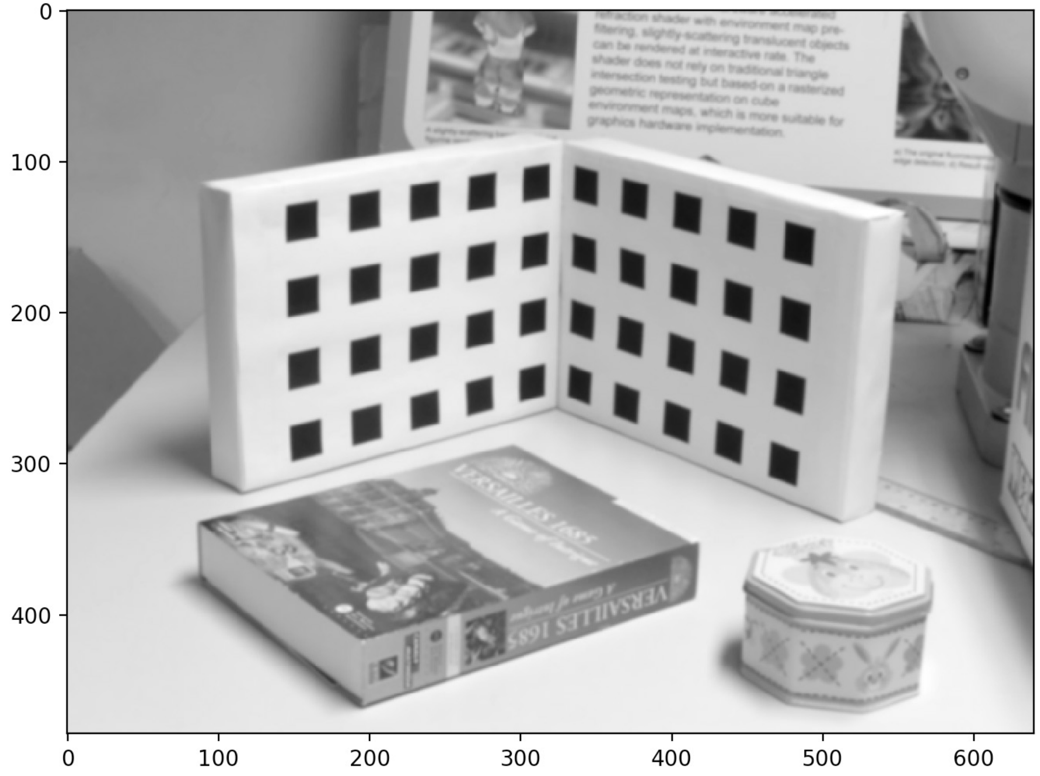
❖ smooth2D()

- Call **smooth1D()** twice
  - Convolve each row and column with the 1D Gaussian Filter
  - Use **matrix.T** to transpose a matrix

```
>>> import numpy as np
>>> a = np.array([(1, 2, 3), (4, 5, 6), (7, 8, 9)])
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> a.T
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])
```

- If you implement smooth1D() and smooth2D() correctly, you may get a similar output as follows (if applied on grayscale image):





Requirements

- Use the formula for the Y-channel of the YIQ model in performing the color-to-grayscale image conversion.
- **Compute I_x and I_y correctly by finite differences.**
- **Construct images of $I_x{}^2$, $I_y{}^2$, and $I_x\, I_y$ correctly.**
- Compute a proper filter size for a Gaussian filter based on its sigma value.
- Construct a proper 1D Gaussian filter.
- Smooth a 2D image by convolving it with two 1D Gaussian filters.
- Handle the image border using partial filters in smoothing.
- **Construct an image of the cornerness function R correctly.**
- **Identify potential corners at local maxima in the image of the cornerness function R.**
- **Compute the cornerness value and coordinates of the potential corners up to sub-pixel accuracy by quadratic approximation.**
- **Use the threshold value to identify strong corners for output.**

✓ **harris()**

➢ **Harris corner detection** [lecture 3, pp. 29]

Corner Detection

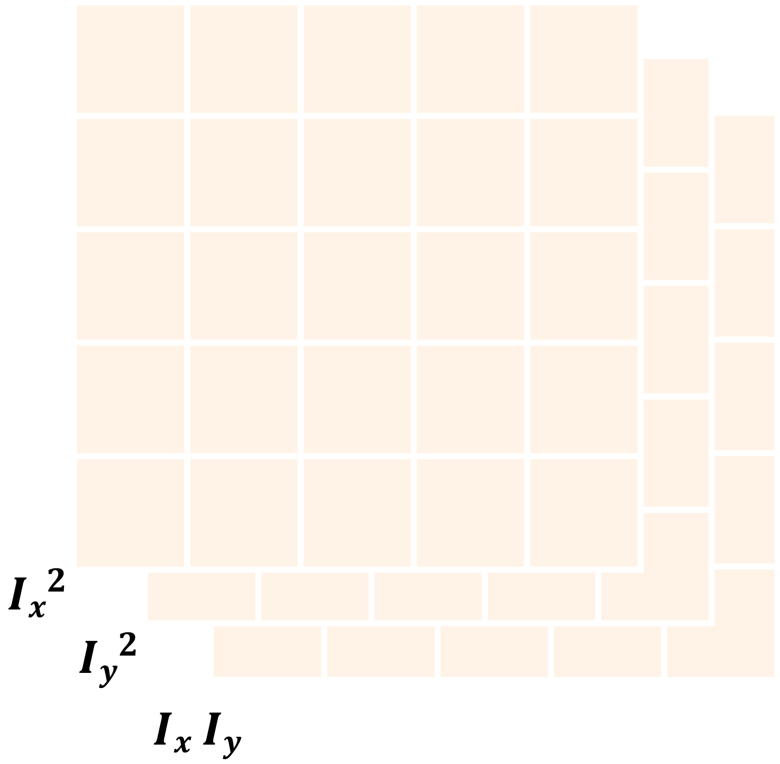
- Summary of Harris corner detection algorithm:
  1. Compute I_x and I_y at each pixel $I(x, y)$
  2. Form the images of I_x^2, I_y^2 and I_xI_y respectively
  3. Smooth the images of squared image derivatives
  4. Form an image of the cornerness function R using the smoothed images of squared derivatives (i.e., $\langle I_x^2 \rangle$, $\langle I_y^2 \rangle$ and $\langle I_xI_y \rangle$)
  5. Locate local maxima in the image of R as corners
  6. Compute the coordinates of the corners up to sub-pixel accuracy by quadratic approximation using values in the neighborhood
  7. Threshold the corners so that only those with a value of R above a certain value are retained

- **Compute I_x and I_y correctly by finite differences.**
  - Compute the **Horizontal (I_x)** and **Vertical (I_y)** Derivatives

- **Construct images of $I_x{}^2$, $I_y{}^2$, and $I_x\, I_y$ correctly.**
  - $\boldsymbol{I_x{}^2 = I_x \times I_x}$
  - $\boldsymbol{I_y{}^2 = I_y \times I_y}$
  - $\boldsymbol{I_x\, I_y = Ix \times I_y}$

$$\mathbf{A} = \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

- **Smooth $I_x{}^2$, $I_y{}^2$, and $I_x\, I_y$**



➢ Use numpy function `np.gradient` to calculate I_x I_y

```python
import numpy as np

arr = np.array([[6, 3, 5, 1],
    [3, 8, 6, 3],
    [7, 3, 3, 4]])
grad = np.gradient(arr)
grad0 = np.gradient(arr, axis=0)
grad1 = np.gradient(arr, axis=1)
print(arr)
print(grad)
print(grad0)
print(grad1)
```

✓ 0.5s

```
[[6 3 5 1]
 [3 8 6 3]
 [7 3 3 4]]
[array([[-3. ,  5. ,  1. ,  2. ],
       [ 0.5,  0. , -1. ,  1.5],
       [ 4. , -5. , -3. ,  1. ]]), array([[-3. , -0.5, -1. , -4. ],
       [ 5. ,  1.5, -2.5, -3. ],
       [-4. , -2. ,  0.5,  1. ]])]
[[-3.   5.   1.   2. ]
 [ 0.5  0.  -1.   1.5]
 [ 4.  -5.  -3.   1. ]]
[[-3.  -0.5 -1.  -4. ]
 [ 5.   1.5 -2.5 -3. ]
 [-4.  -2.   0.5  1. ]]
```

- **Cornerness function R** [lecture 3, pp. 27]

- In Harris corner detection algorithm, corners are marked at points where the quantity $R = \lambda_1\lambda_2 - \kappa(\lambda_1 + \lambda_2)^2$ exceeds some threshold, here κ is a parameter set to 0.04 as suggested by Harris
- Note that $\lambda_1\lambda_2 = \det(\mathbf{A})$ and $\lambda_1+\lambda_2 = \mathrm{trace}(\mathbf{A})$

$$\mathbf{A} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \qquad \det(\mathbf{A}) = a_{00}a_{11} - a_{01}a_{10} \qquad \mathrm{trace}(\mathbf{A}) = a_{00} + a_{11}$$

and hence direct computation of the eigenvalues is not necessary

- Corners are defined as local maxima of the cornerness function R, and sub-pixel accuracy is achieved through a ***quadratic approximation*** of the neighborhood of the local maxima (using 4-neighbours and the center pixel):

$$f(x,y)=ax^2+by^2+cx+dy+e$$

➢ Calculate cornerness function R

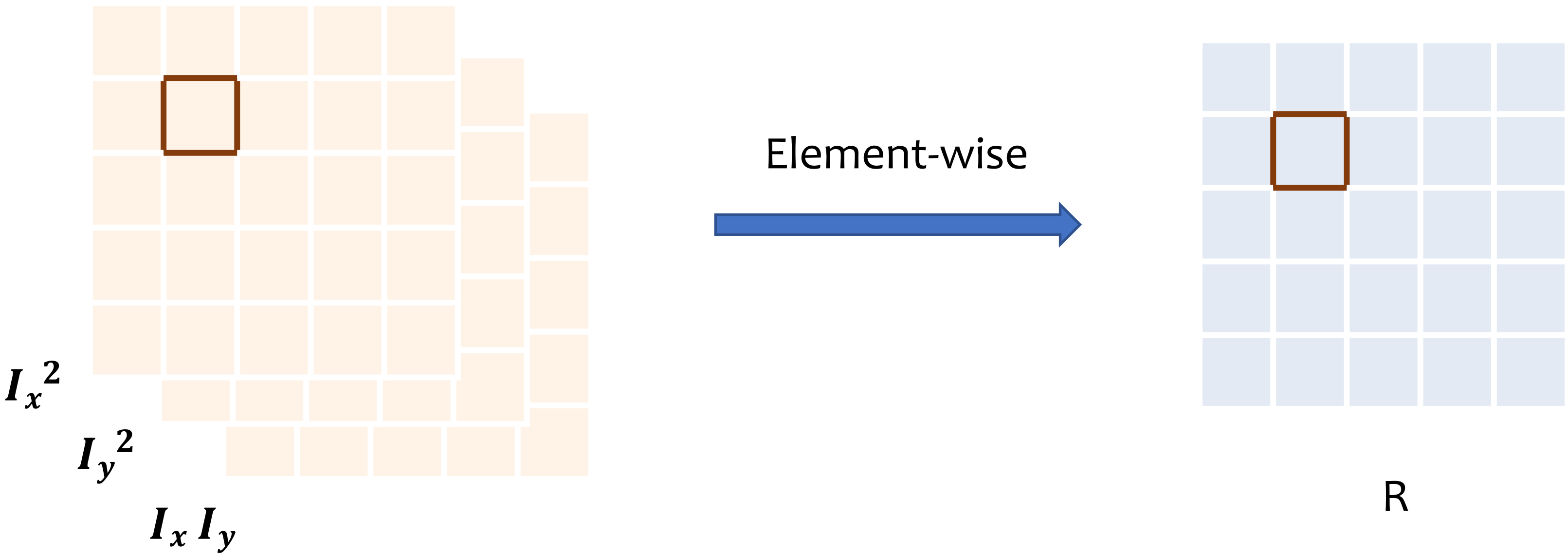
$$\mathbf{A} = \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \Rightarrow \begin{cases} \det(A) = I_x{}^2 I_y{}^2 - (I_x I_y)^2 \\ \text{trace}(A) = I_x{}^2 + I_y{}^2 \end{cases}$$

$$\Rightarrow R = \lambda_1\lambda_2 - \kappa(\lambda_1 + \lambda_2)^2$$

$$= \det(A) - \kappa\big(trace(A)\big)^2$$

$$\kappa = 0.04$$

$$= (I_x{}^2 I_y{}^2 - (I_x I_y)^2) - 0.04(I_x{}^2 + I_y{}^2)^2$$



➢ Find the local maxima

✓ Perform **non-maximal suppression** by considering **8-neighbors**

2	2	4	7	5
1	3	4	5	8
1	2	2	4	3
3	8	7	9	6
1	7	8	3	6

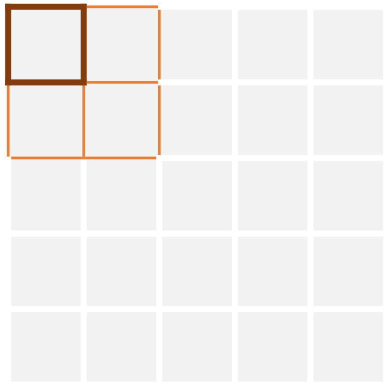
R

2	2	4	7	5
1	3	4	5	8
1	2	2	4	3
3	8	7	**9**	6
1	7	8	3	6

R

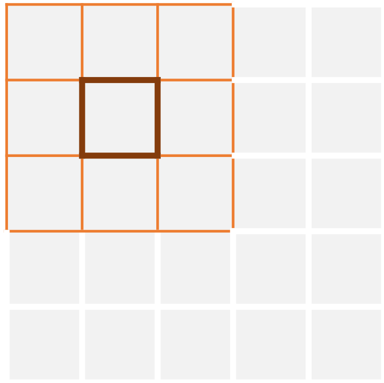
- smooth1D()

  ⟹ Consider boundary. (partial filter)



- Find local maxima

  ⟹ Do not Consider boundary

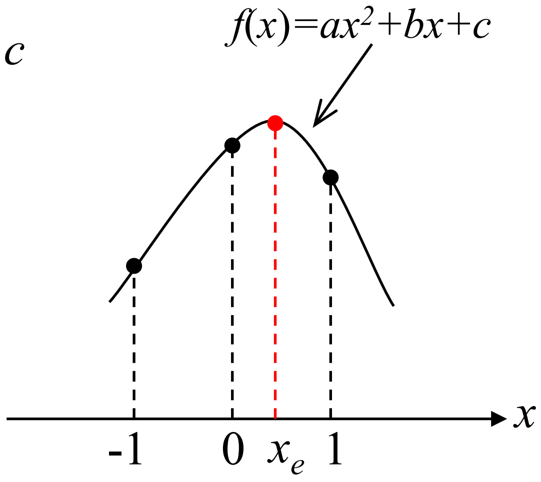


➢ **1D Quadratic approximation** [lecture 3, pp. 14]

1D Edge Detection

- Having obtained the derivative $S'(x)$, interpolation can be used to locate any maxima or minima to ***sub-pixel accuracy***
  - Approximate the function locally by $f(x) = ax^2 + bx + c$
  - Without loss of generality, let the sample maximum and its immediate neighbors have coordinates $x = 0$, -1 and 1 respectively
  - This gives

$$\begin{cases} f(-1) = a - b + c \\ f(0) = c \\ f(1) = a + b + c \end{cases} \Rightarrow \begin{cases} a = \frac{f(1)+f(-1)-2f(0)}{2} \\ b = \frac{f(1)-f(-1)}{2} \\ c = f(0) \end{cases}$$



  - Locate the maximum/minimum by solving $f'(x) = 2ax + b = 0$ which gives

$$x_e = -\frac{b}{2a} = -\frac{f(1) - f(-1)}{2[f(1) + f(-1) - 2f(0)]}$$

- Finally, an edge is marked at each maximum or minimum whose magnitude exceeds some thresholds

➢ 2D Quadratic approximation

➢ Perform quadratic approximation to local corner up to **sub-pixel accuracy** (using 4-neighbours and the center pixel)

$$f(x,y)=ax^2+by^2+cx+dy+e$$

- $f(0,0)$
- $f(-1,0)$
- $f(1,0)$
- $f(0,-1)$
- $f(0,1)$

2	2	4	7	5
1	3	4	5	8
1	2	2	4	3
3	8	7	**9**	6
1	7	8	3	6

R(x,y)

R

2	2	4	7	5
1	3	4	5	8
1	2	2	4	3
3	8	7	**9**	6
1	7	8	3	6

Max(R)

R

2D Quadratic approximation

$$f(x,y)=ax^2+by^2+cx+dy+e$$

$$\Downarrow$$

$$\begin{aligned} f(0,0) &= e \\ f(-1,0) &= a-c+e \\ f(1,0) &= a+c+e \\ f(0,-1) &= b-d+e \\ f(0,1) &= b+d+e \end{aligned} \Rightarrow \begin{aligned} a &= \frac{f(-1,0)+f(1,0)-2f(0,0)}{2} \\ b &= \frac{f(0,-1)+f(0,1)-2f(0,0)}{2} \\ c &= \frac{f(1,0)-f(-1,0)}{2} \\ d &= \frac{f(0,1)-f(0,-1)}{2} \\ e &= f(0,0) \end{aligned} \Rightarrow \begin{cases} x = -\dfrac{c}{2a} \\ y = -\dfrac{d}{2b} \end{cases}$$

- **Use the threshold value to identify strong corners for output.**
  - Cornerness > Threshold. $\Longrightarrow$ consider it as a corner

- Follow the hints given in program template

```
# TODO: compute Ix & Iy

# TODO: compute Ix2, Iy2 and IxIy

# TODO: smooth the squared derivatives

# TODO: compute cornesness function R

# TODO: mark local maxima as corner candidates;
#       perform quadratic approximation to local corners upto sub-pixel
accuracy

# TODO: perform thresholding and discard weak corners
```