Lab Instructions - session 10

Image Pyramid, Multiscale corner detection

Image Downsampling

When we **downsample** (shrink) an image by some factor, a naive approach is to simply pick every *sth* pixel in each dimension. For example, to downsize an image by a factor of `s = 4`, we could take every 4th pixel in the x and y directions. However, the output image from this naive sampling can look strange or **aliased**. Aliasing often appears as jagged edges or moiré patterns – essentially, artifacts caused by high-frequency content in the original image that confuses the sampling. To avoid this, we should **pre-filter** the image (typically by blurring) before downsampling. Blurring removes some of the high-frequency details, making the image less prone to aliasing when sampled.

The code below demonstrates image downsampling. It shows three versions of a sample image being downsized by a factor of `s`: one by naive sampling, one by blurring with a **box filter** before sampling, and one by blurring with a **Gaussian filter** before sampling. Initially, the blurred versions are commented out so that all three outputs are identical. We will then modify the code to see the effect of proper blurring.

File: **downsize.py**

```
import numpy as np
import cv2
from matplotlib import pyplot as plt

I = cv2.imread('toosi.jpg')      # Load the image
s = 4                            # downsize factor

# Downsize by sampling every s-th pixel (naive downsampling)
J = I[::s, ::s, :]               # pick every s-th pixel in both dimensions

# Initialize Jb and Jg as copies of the naive downsample (they will be replaced
after blur)
Jb = J
Jg = J

# Blur with a box filter, then downsample (anti-aliasing)
# ksize = s + 1
# Ib = cv2.boxFilter(I, -1, (ksize, ksize))
# Jb = Ib[::s, ::s, :]

# Blur with a Gaussian filter, then downsample (anti-aliasing)
# sigma = (s+1)/np.sqrt(12)  # roughly equivalent sigma for Gaussian kernel of
size s+1
# Ig = cv2.GaussianBlur(I, (0,0), sigma)
# Jg = Ig[::s, ::s, :]
```

```
# Prepare a figure to display results
f, ax = plt.subplots(2, 3, gridspec_kw={'height_ratios': [s, 1]})
for a in ax.ravel():
    a.axis('off')  # turn off axes for all subplots

# Show the original image (top row, center)
ax[0,1].set_title('Original')
ax[0,1].imshow(I[:, :, ::-1])  # convert BGR to RGB for displaying with plt

# Show the downsampled images (bottom row)
ax[1,0].set_title('Downsized (naive)')
ax[1,0].imshow(J[:, :, ::-1], interpolation='none')  # no interpolation to
emphasize raw pixels

ax[1,1].set_title('Box Blur + Downsized')
ax[1,1].imshow(Jb[:, :, ::-1], interpolation='none')

ax[1,2].set_title('Gaussian Blur + Downsized')
ax[1,2].imshow(Jg[:, :, ::-1], interpolation='none')

plt.show()
```

After running the above code, observe the output figure. The top row shows the original image. The bottom row shows, side by side, the naive downsampling and (currently identical) placeholders for box-blur and Gaussian-blur downsampling. Now, let's think and experiment:

- **What looks "weird" about the naive downsized image?** Why do you think this is happening?
- **Anti-aliasing with a box filter:** Uncomment the three lines under *"Blur with a box filter, then downsample"* in the code. This first blurs the image using a `cv2.boxFilter` (which averages pixel values in an `(s+1)×(s+1)` neighborhood) and then downsamples. Run the code again. How does this change the output image compared to the naive approach?

- **Anti-aliasing with a Gaussian filter:** Similarly, uncomment the lines under *"Blur with a Gaussian filter, then downsample"*. This uses a Gaussian blur (with a sigma chosen roughly equivalent to the box filter size) before sampling. Run the code. What differences do you see between using a box filter versus a Gaussian filter for pre-blur? Which produces a more visually pleasing downsized image?

- Try changing the downsampling factor `s` to a larger value (say 8). How does the aliasing in the naive approach worsen as `s` increases? Do the pre-blur methods continue to produce good results at higher downsampling factors?

- For a more direct approach, try using OpenCV's built-in resize function instead of manual indexing. For example, use `cv2.resize(I, None, fx=1/s, fy=1/s, interpolation=cv2.INTER_NEAREST)` for naive downsampling, and `cv2.resize` with `cv2.INTER_AREA` (which is good for shrinking)

Image pyramid

An image pyramid is created by repeatedly blurring and downsampling an image. The OpenCV function `pyrDown` creates the next level of the pyramid from the previous level. A pyramid can be created by repeatedly calling this function. The following code creates an image pyramid and displays it. Here, the pyramid gets stored in a Python list (not a numpy array).

File: **pyramid.py**

```python
import numpy as np
import cv2
from matplotlib import pyplot as plt

I = cv2.imread('toosi.jpg')
psize = 6  # number of levels in the pyramid

# Build the image pyramid
J = I.copy()
Pyr = [J]                   # level 0: original image
for i in range(psize - 1):
    J = cv2.pyrDown(J)      # blur and downsample by factor of 2
    Pyr.append(J)           # append the smaller image to the pyramid list

# Display the pyramid levels side by side
# Create a wide figure with subplots for each level
size_list = [2**(psize - 1 - i) for i in range(psize)]  # just for nicer display
scaling
f, ax = plt.subplots(1, psize, gridspec_kw={'width_ratios': size_list})
for a in ax.ravel():
    a.axis('off')           # turn off axes

# Plot each pyramid level in a subplot
for l in range(psize):
    ax[l].set_title(f"Level {l}")
    J = Pyr[l]
    ax[l].imshow(J[:, :, ::-1], interpolation='none')  # show BGR image as RGB
plt.show()
```

- **Pyramid scale relationship.** Verify the size of each pyramid level. What is the size of level 1 relative to level 0? What about level 2 relative to level 0? In general, if an image is downsampled `n` times (each by a factor of 2), what is the scale of the resulting image relative to the original?

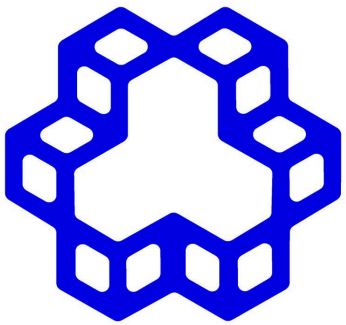
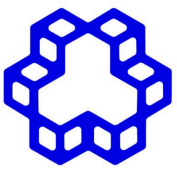
- **Effect of blurring.** Observe the smallest pyramid level (level 5 in this case). Although it's tiny, does it still roughly resemble the original image content (just very blurry)? This demonstrates how repeated blurring+downsampling preserves coarse content while removing fine details. Why is preserving coarse structure important for multi-scale analysis?

- Try increasing `psize` to include more levels (e.g., 7 or 8). How small does the image get? Is there a point at which downsampling further doesn't make sense? *(Hint: When the image gets to just a few pixels wide, it might not contain meaningful information to detect features.)*

- We used `cv2.pyrDown` to build the pyramid. For learning purposes, try to construct a pyramid manually using what you learned in the downsampling section: use Gaussian blurring (`cv2.GaussianBlur`) followed by manual decimation (picking alternate pixels) in a loop. Compare your results with `cv2.pyrDown` – they should be very similar if done correctly.

Multiscale corner detection:

We have 3 images saved to the files **kntu1.jpg**, **kntu2.jpg** and **kntu4.jpg**. All images are of the same size (800 by 800 pixels). However, the logo in images kntu2.jpg and kntu4.jpg are, respectively, 2 and 4 times smaller than the logo in kntu1.jpg.

		
kntu1.jpg	kntu2.jpg	kntu4.jpg

We want to find the correct window size for detecting Harris corners in each image. For this, we run the Harris corner detection algorithm for window sizes 2, 4, 8, 16, 32, and 64 for each image. Run the following code and find a proper window size for detecting corners in **kntu1.jpg**. The logo has **78** corners.

File: **multiscale_corner.py**

```
import cv2
import numpy as np

I = cv2.imread('kntu1.jpg')
G = cv2.cvtColor(I,cv2.COLOR_BGR2GRAY)
G = np.float32(G)

for k in range(1,7):
    win_size = 2**k # 2^k
    soble_kernel_size  = 3 # kernel size for gradients
    alpha = 0.04
    H = cv2.cornerHarris(G,win_size,soble_kernel_size,alpha)
    H = H / H.max()

    C = np.uint8(H > 0.01) * 255
    nc,CC = cv2.connectedComponents(C);

    J = I.copy()
    J[C != 0] = [0,0,255]
    cv2.putText(J,'winsize=%d, corners=%d'%(win_size, nc-1),(20,40), \
                cv2.FONT_HERSHEY_SIMPLEX, 1,(0,0,255),2)

    cv2.imshow('corners',J)

    if cv2.waitKey(0) & 0xFF == ord('q'):
        break
```

- What is the first win_size for which the algorithm detects the right number of corners?
- Test the above code on kntu2.jpg and kntu4.jpg. In each case take note of the **smallest win_size** for which the algorithm correctly detects the corners. What do these numbers say?

Task 1:

We want to use the above concept to decide the logo is how many times larger or smaller in one image compared to the other. To do this, we write a function named `first_correct_winsize` which finds the smallest window size (as a power of 2, i.e. 2,4,8,16,32,64) that correctly detects all the **78** corners. By comparing the window size for two images you can compare the sizes of the logos. Just fill the function body and leave the rest of the code unchanged.

File: **task1.py**

```
import cv2
import numpy as np

NO_CORNERS = 78

def first_correct_winsize(I):
    "find the smallest win_size for which all corners are detected"
    # write your code here


    return 4 # incorrect


I1 = cv2.imread('kntu1.jpg')
I2 = cv2.imread('kntu4.jpg')

s1 = first_correct_winsize(I1)
s2 = first_correct_winsize(I2)

J = np.concatenate((I1,I2), 1)

if s1 < s2:
    txt = 'Logo 1 is %d times smaller than logo 2'%(s2/s1)
elif s1 > s2:
    txt = 'Logo 1 is %d times larger than logo 2'%(s1/s2)
else:
    txt = 'Logo 1 is about the same size as logo 2'

cv2.putText(J,txt,(20,40), \
                cv2.FONT_HERSHEY_SIMPLEX, 1,(0,0,255),2)

cv2.imshow('scale',J)
cv2.waitKey(0)
```

- **Implement and verify `first_correct_winsize`.** After writing the function, test it on `kntu1.jpg`, `kntu2.jpg`, and `kntu4.jpg`. Do the returned window sizes match what you observed manually in the previous section?

- Try using the function to compare `kntu1.jpg` vs `kntu2.jpg`, and `kntu2.jpg` vs `kntu4.jpg` as well. Does it correctly report the size difference (for example, "2 times larger/smaller") in those cases?

- **Why smallest window?** We specifically look for the *smallest* window size that detects all corners. Why might it be important to use the smallest sufficient window rather than just any window that works?

- In an ideal scenario, we'd like our corner detector to automatically handle scale differences without us manually tuning window sizes. What are some ways you might achieve scale-invariant corner detection?

Multiscale corner detection with image pyramid

An alternative approach is to keep the Harris window size fixed and change the image size instead. In the following, we use a fixed window size (win_size = 4) and run the corner detection algorithm for different image sizes in an image pyramid. Run the code and see the result. Then move on quickly to **task 2**.

File: **multiscale_corner_pyramid.py**

```
import cv2
import numpy as np

I = cv2.imread('kntu1.jpg')

psize = 6 # size of the pyramid (no. of levels)

# building the pyramid
J = I.copy()
Pyr = [J] # the first element is simply the original image
for i in range(psize-1):
    J = cv2.pyrDown(J) # blurs, then downsampled by a factor of 2
    Pyr.append(J)

for k in range(psize): # k = 0,1,..., psize-1
    J = Pyr[k]
    G = cv2.cvtColor(J,cv2.COLOR_BGR2GRAY)
    G = np.float32(G)

    win_size = 4 # do not change this
```

```
    soble_kernel_size  = 3 # kernel size for gradients
    alpha = 0.04
    H = cv2.cornerHarris(G,win_size,soble_kernel_size,alpha)
    H = H / H.max()

    C = np.uint8(H > 0.01) * 255
    nc,CC = cv2.connectedComponents(C);

    J[C != 0] = [0,0,255]

    JJ = np.zeros(I.shape,dtype=I.dtype)
    JJ[:J.shape[0],:J.shape[1],:] = J;
    cv2.putText(JJ,'scale=1/%d, corners=%d'%(2**k, nc-1),(360,30), \
                cv2.FONT_HERSHEY_SIMPLEX, 1,(0,0,255),2)

    cv2.imshow('corners',JJ)

    if cv2.waitKey(0) & 0xFF == ord('q'):
        break
```

Task 2

The job is the same as **task 1**. But this time we do it using an image pyramid. This time you need to write a function called `first_correct_scale` which finds the first image scale in a pyramid that correctly detects all the **78** corners. By comparing the image scales for the two images you will find their relative size. Notice that you just need to add a little piece of code and fill in the function body after the line saying `#! write your code here! **********`

File: **task2.py**

```
import cv2
import numpy as np

NO_CORNERS = 78

def first_correct_scale(I):
    "find the smallest scale for which all corners are detected"

    psize = 6 # size of the pyramid

    # building the pyramid
    J = I.copy()
    Pyr = [J] # the first element is simply the original image
    for i in range(psize-1):
```

```
        J = cv2.pyrDown(J) # blurs, then downsampled by a factor of 2
        Pyr.append(J)

    for k in range(psize): # k = 0,1,..., psize-1
        J = Pyr[k]
        G = cv2.cvtColor(J,cv2.COLOR_BGR2GRAY)
        G = np.float32(G)

        win_size = 4 # do not change this!!
        soble_kernel_size  = 3 # kernel size for gradients
        alpha = 0.04

        #! write your code here! ****************************************
        nc = 79 # !!! delete this line!

        if nc-1 == NO_CORNERS: # if the connected components
            return 2**k

I1 = cv2.imread('kntu1.jpg')
I2 = cv2.imread('kntu4.jpg')

sc1 = first_correct_scale(I1)
sc2 = first_correct_scale(I2)

J = np.concatenate((I1,I2), 1)

if sc1 < sc2:
    txt = 'Logo 1 is %d times smaller than logo 2'%(sc2/sc1)
elif sc1 > sc2:
    txt = 'Logo 1 is %d times larger than logo 2'%(sc1/sc2)
else:
    txt = 'Logo 1 is about the same size as logo 2'

cv2.putText(J,txt,(20,40), \
                cv2.FONT_HERSHEY_SIMPLEX, 1,(0,0,255),2)
cv2.imshow('scale',J)
cv2.waitKey(0)
```

- **Implement `first_correct_scale`.** Test it on `kntu1.jpg`, `kntu2.jpg`, `kntu4.jpg`. Does it find the correct scale factors that correspond to the earlier window size results? (Each downsampling by 2 in the pyramid roughly corresponds to doubling the required Harris window size in the original image.)

- **Compare pyramid vs window scaling.** Do Task 1 and Task 2 give the same relative size conclusions for the image pairs?

- **Efficiency considerations.** The pyramid method and the variable window method both achieve the goal of multi-scale corner detection. Which method do you think is

more efficient or easier in practice? *(Think about cases with very large images or the need to detect a wide range of scales. Often, working with an image pyramid can be faster because most of the heavy processing happens on smaller images.)* Also, consider memory: the pyramid method stores multiple copies of the image, but each smaller; the multi-window method processes the full-size image multiple times.

- Beyond corners – many computer vision tasks use image pyramids for scale invariance (from object detection to texture analysis). Can you think of a scenario where an image pyramid would be useful, aside from corner detection?

Have some fun!

[This is not part of your lab] Run the next code and see the result. You will learn how to do this soon. For the time being, just run it. You can use your own photo.

File: **create_pyramid.py**

```
import numpy as np
import cv2

I = cv2.imread('toosi.jpg')
m,n,_ = I.shape

P1 = np.array([[0,0], [0, m-1], [n-1,0], [n-1,m-1]])

psize = 7 # size of the pyramid (no. of levels)

J = np.ones((600,500,3), dtype=np.uint8)*255
m2,n2,_ = J.shape

v  = np.array([(n2//2,0)])
P2 = np.array([(0,4*m2//5),
               (5*n2//6,m2),
               (3*n2//12,7*m2//12),
               (n2,8*m2//12)])

cv2.line(I, (0,0), (0,m-1), (1,1,1),4)
cv2.line(I, (0,0), (n-1,0), (1,1,1),4)
cv2.line(I, (n-1,m-1), (0,m-1), (1,1,1),4)
cv2.line(I, (n-1,m-1), (n-1,0), (1,1,1),4)

for i in range(4):
    cv2.line(J, (v[0,0],v[0,1]), (P2[i,0],P2[i,1]), (0,0,0),2)

p21 = P2[1].copy()

for i in range(psize):
    H, status = cv2.findHomography(P1, P2)
```

```
K = cv2.warpPerspective(I, H, (J.shape[1],J.shape[0]))
msk = K.max(axis=2) != 0
J[msk,:] = K[msk,:]

cv2.line(J, (v[0,0],v[0,1]), (p21[0],p21[1]), (0,0,0),2)
cv2.imshow('',J)
cv2.waitKey()
P2 = (P2 + v)/2
```

References

- [OpenCV-Python Tutorials - Image Pyramids]