Assignment 2
Dong Han

Aims: Harris detector and Orientation and Display

Harris detector

1. Compute image derivative (∂I_x , ∂I_y) in x and y direction

Before computing image derivative we need create 2D gaussian kernel and 2D convolution. In gaussian kernel function, the sigma values determine the size of kernel. In convolution function, we set the padding to keep the original image size.

```
## create 2D kernel by hand
def Gaussian_2D_Kernel(sigm):

  K = 2.5 #  K is any value between 2.5 and 3
  Lon = math.floor(K*sigm) # half-size of the mask
  size = 2*K*sigm+1 # full-size of the mask
  G_= []
  for x in range(-Lon,Lon+1): # x is from [-Lon:Lon]
    G = np.exp(-x*x /(2 *sigm * sigm)) # 1D gaussian formula when k = 1
    G_.append(G)
  sum_ = np.sum(G_) # compute the sum of output values
  G = np.array(G_ / sum_) # normalize to (0-1)

  return G[:, np.newaxis]*G[:, np.newaxis].T # compute 2D kernel by multiplying 1D kernel

## create 2D convolution
def convolve(image, kernel):

    Hi, Wi = image.shape # get image size
    Hk, Wk = kernel.shape # get kernel size
    out = np.zeros((Hi, Wi))  # initiate output

    pad_height = Hk // 2  # define the pading height
    pad_width = Wk // 2  # define the pading width

    pading = None
    pading = np.zeros((Hi+2*pad_height, Wi+2*pad_width))# fill zero
    pading[pad_height:pad_height+Hi, pad_width:pad_width+Wi] = image # insert image to the center the border is filed by zero


    h_flip = np.flip(np.flip(kernel, 0), 1) # np.flip funtion, parameter 0: flip up and down, parameter 1: flip left and right

    for i in range(Hi):
        for j in range(Wi):
            out[i][j] = np.sum(np.multiply(h_flip, pading[i:(i+Hk), j:(j+Wk)])) # write the result after weighted summation
    # out = abs(out) # eliminate negative values
    # out = out*(255.0/out.max()) # normalize to [0-255]

    return out
```

The 2D Prewitt kernel is used to compute image derivative in our case.

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

Figure 1. The left is Prewitt kernel in horizontal direction and right is in vertical direction

The results image shows below, we can see clearly the corresponding image derivative shows correctly both in horizontal and vertical direction. For visual comparison purpose, the results also are rescaled for displaying.

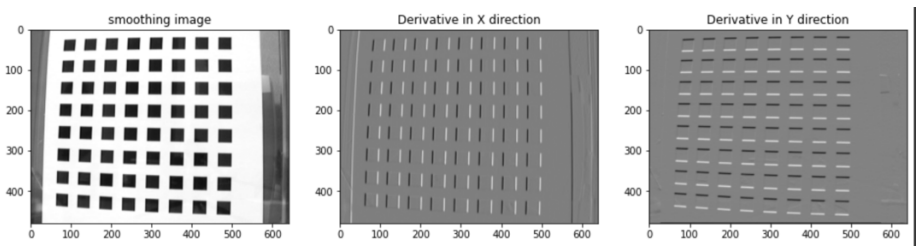


Figure 2. The first derivative of images

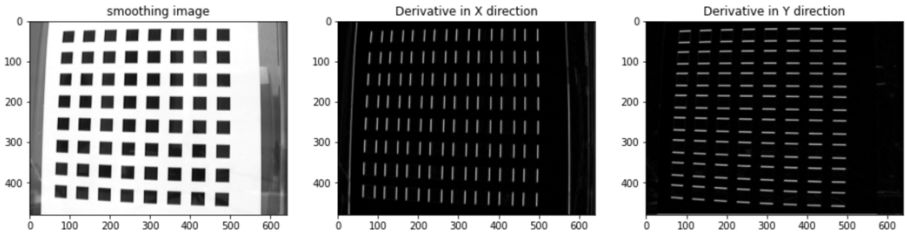


Figure 3. the first derivative of images which are scale to [0-255]

2. Compute the matrices ∂I_x^2, ∂I_y^2, $\partial I_x \partial I_y$

The Numpy's numerical operations can be used to compute those 3 values easily.

```
def Image_2rd_Derivative(Ix,Iy):
  ## compute Ixx Iyy Ixy Iyx
  Ixx = np.square(Ix)
  Iyy = np.square(Iy)
  Ixy = np.multiply(Ix, Iy)
  Iyx = np.multiply(Ix, Iy)

  return Ixx, Iyy, Ixy, Iyx

Ixx, Iyy, Ixy, Iyx = Image_2rd_Derivative(Ix,Iy)
```

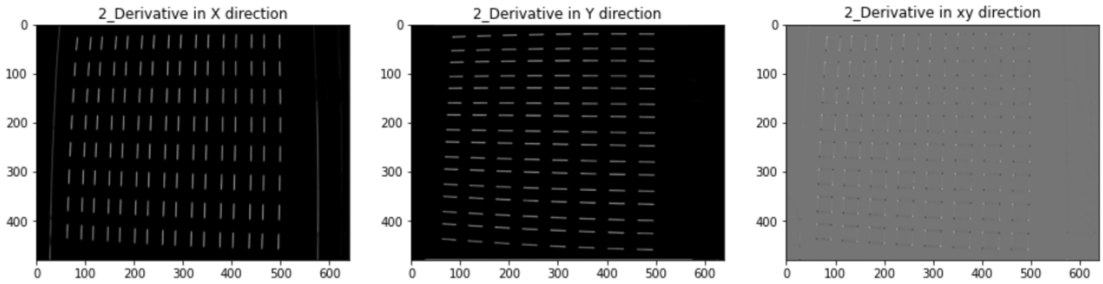


Figure 4. ∂Ix2, ∂Iy2, ∂Ix∂Iy results

We can notice that in $\partial I_x \partial I_y$ image, the x and y derivatives are computed at same time, and the corner region in the image are detected.

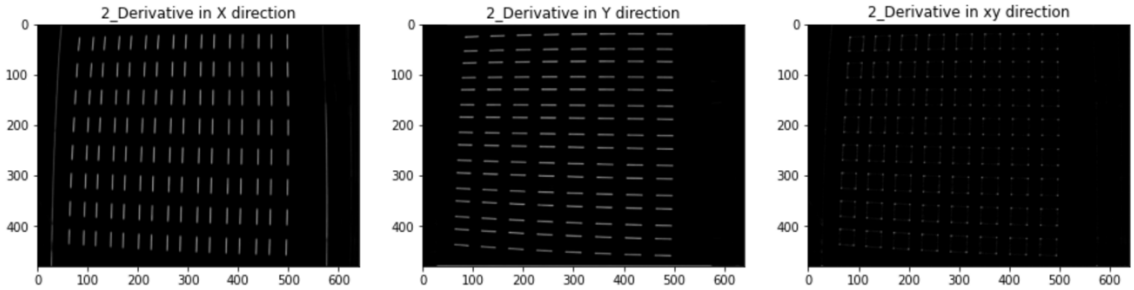


Figure 5. ∂Ix2, ∂Iy2, ∂Ix∂Iy results which are rescale to [0-255]

The points show more clearly when the image is rescaled.

3. Compute Trace and Determinant in each pixel

When we get ∂Ix∂Iy and ∂Ix2, ∂Iy2, ∂Ix∂Iy, the second-moment matrix M can be computed. The trace and determinant can be computed afterwards.

$$\begin{bmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{bmatrix} \qquad detM = \lambda_1\lambda_2 \qquad traceM = \lambda_1 + \lambda_2$$

The np.linalg.det() and np.trace() can be used to compute trace and determinant of M

```
## Compute Trace and Determinant in each pixel
w,h = img.shape
detM = np.zeros((w, h), np.float32) # initiate detM array
traceM= np.zeros((w, h), np.float32) # initiate traceM array

for row in range(w):
  for col in range(h):
    M = np.array([[Ixx[row][col], Ixy[row][col]], [Iyx[row][col], Iyy[row][col]]]) # compute Harris matrix: M
    detM[row][col] = np.linalg.det(M) # Determinant
    traceM[row][col] = np.trace(M) # Trace

print(detM)
print(traceM)
```

4. Compute Harris-feature map

According to Harris response formula.

$$R = detM - \alpha(traceM)^2$$

The Harris response of each pixel in image are computed. We set constant k to 0.06.

```
def Harris_feature_map(img,Ixx,Iyy,Ixy,Iyx):
  w,h = img.shape # get image size
  k = 0.06 # k is the sensitivity factor to separate corners from edges, typically a value close to zero
  R = np.zeros((w, h), np.float32) # initiate R array

  for row in range(w):
    for col in range(h):
      M = np.array([[Ixx[row][col], Ixy[row][col]], [Iyx[row][col], Iyy[row][col]]]) # compute Harris matrix: M
      R[row][col] = np.linalg.det(M) - (k * np.square(np.trace(M))) # compute Harris-feature map: R
  return R

R = Harris_feature_map(img,Ixx,Iyy,Ixy,Iyx)
```

5. Identify the local maxima applying Non-Maxima suppression on the 5x5 neighborhood

The idea of non-maximal suppression is that if the (corner) response of a pixel is not the highest in a neighborhood, then we don't keep it. The 5x5 size window are defined to compare the neighborhood values.

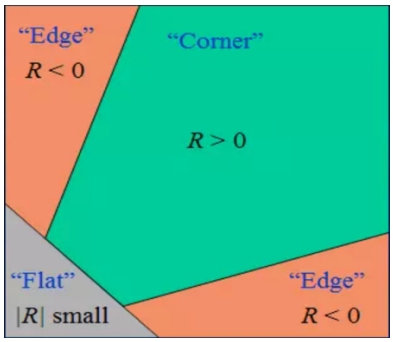
```python
## create Non-Maxima suppression funtion
def Strong_Corner(img,R,threshold):
  Threshold = threshold # select Strong Corner above threshold

  Selected_Points = []
  w,h = img.shape # get image size
  for row in range(w):
      for col in range(h):
          if R[row][col] > Threshold:
              local_maxima = R[row][col] # Identify the local maxima

              # Local non-maxima suppression
              jump_to_next = False
              for i in range(5): # define the window size
                  for k in range(5): # define the window size
                      if row + i - 2 < w and col + k - 2 < h:
                          if R[row + i - 2][col + k - 2] > local_maxima: # if response is highest in a neighborhood than keep it
                              jump_to_next = True
                              break

              if not jump_to_next:
                 Selected_Points.append((row, col)) # store the selected points
  return Selected_Points
```

6. Keep only the first 1000 highest values.

```python
## create Non-Maxima suppression funtion
def Strong_Corner(img,R,threshold):
  Threshold = threshold # select first 1000 Strong Corner
  Selected_Points = []
  w,h = img.shape # get image size
  for row in range(w):
      for col in range(h):
          if R[row][col] > Threshold:
              local_maxima = R[row][col] # Identify the local maxima

              # Local non-maxima suppression
              jump_to_next = False
              for i in range(5): # define the window size
                  for k in range(5): # define the window size
                      if row + i - 2 < w and col + k - 2 < h:
                          if R[row + i - 2][col + k - 2] > local_maxima: # if response is highest in a neighborhood than keep it
                              jump_to_next = True
                              break

              if not jump_to_next:

                 Selected_Points.append((row, col)) # store the selected points
  return Selected_Points


Selected_Points=Strong_Corner(img,R,7020) # compute selected strong points
```

The threshold of Harris response determines which type of region we are interested.

In this case, we are interested in corner in image, so we adjust threshold accordingly to selected first 1000 strong points. In our case, when the threshold is set to **7020**, 1000 points are computed.



The length of list which store selected points are printed out and also each location of points is also computed.

```
1000
[(1, 1), (1, 49), (1, 567), (1, 618), (1, 638), (2, 35), (2, 627), (3, 13), (3, 591), (3, 600), (7, 611),
```

7. Refine the spatial coordinates

The cv2.cornerSubPix are used in this case with window size 5x5.

```
## merge two list into tuple in coordinates format
def merge(list1, list2):
  merged_list = [(list1[i], list2[i]) for i in range(0, len(list1))]
  return merged_list

## define corner refine funtion
def corner_refine(img,Selected_Points):
  winSize = (5, 5) # window of size 5x5.
  zeroZone = (-1, -1)
  criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 40, 0.001)  # termination criteria
  gray = img # get gray image

  ## Order points
  points = Selected_Points # load 1000 keypoint locations from previous step
  points = np.array(points, dtype='float32') # convert to proper data type

  ## Calculate the refined corner locations
  corners = cv2.cornerSubPix(gray, points, winSize, zeroZone, criteria)

  ## Convert output refined corner locations into tuple
  r = corners[:,0] # get row locations
  c = corners[:,1] # get column locations
  refined_locations = merge(r, c)
  return refined_locations

refined_locations = corner_refine(img,Selected_Points)
```

The length of list which store refined points are printed out and also each refined location is also computed. Compare with original locations, the refined location gives us more precise coordinates values.
Original location:

```
1000
[(1, 1), (1, 49), (1, 567), (1, 618), (1, 638), (2, 35), (2, 627), (3, 13), (3, 591), (3, 600), (7, 611), (10, 598), (14, 597), (15, 630), (16, 611),
```

Refined location:

```
1000
[(1.0, 1.0), (4.3905973, 45.234936), (1.0, 567.0), (1.0, 618.0), (1.0, 638.0), (2.0, 35.0), (2.0, 627.0), (3.0, 13.0), (3.0, 591.0), (3.0, 600.0), (7.0, 611.0),
```

Orientation and Display

1. Extract a signature vector $SV = (x, y, \theta, \lambda_1, \lambda_2, \sigma_I)$ for each Harris-point

First we initiate parameters of signature vector. x and y coordinates are refined location in signature vector.

```
refined_locations = corner_refine(img,Selected_Points)
r = np.array(refined_locations)[:,0] # get refined row coordinates of Harris points
c = np.array(refined_locations)[:,1]  # get refined col coordinates of Harris points

## Extract a signature vector SV = (x, y, θ, λ1, λ2, σI) for each Harris-point
## initiate parameters of signature vector
x_refined = r
y_refined = c
lamda1 = [] # Eigenvalue λ1
lamda2 = [] # Eigenvalue λ2
v1 = [] # Eigenvector v1
v2 = [] # Eigenvector v2
σI = 0.5*1.4*1.5      # detection scale σI=1.5σD  (σD = 1.4σ0  σ0=0.5)
σI = [σI]*1000 # repeate 1000 times since for each location we used same detection scale σI
coordinates = Selected_Points # load origial 1000 keypoint locations without refining
coordinates = np.array(coordinates) # convert to proper data type
```

The eigenvalues and eigenvectors can be computed by np.linalg.eig() and the theta indicate the angle region orientation θ between the eigenvector associated with the highest eigenvalue and the axis x of the image.

```
## create funtion to compute signature vector
def signature_vector(coordinates,Ixx,Iyy,Ixy,Iyx):
  x = coordinates[:,0] # row coordinates of Harris points
  y = coordinates[:,1]  # col coordinates of Harris points
  x_v = [1,0] # vector of the axis x of the image

  ## compute Eigenvalues and Eigenvectors
  for i in range(len(x)):
    M = np.array([[Ixx[x[i]][y[i]], Ixy[x[i]][y[i]]], [Iyx[x[i]][y[i]], Iyy[x[i]][y[i]]]]) # compute Harris matrix M which is associated selected 1000 points
    w, v = np.linalg.eig(M) # compute Eigenvalues and Eigenvectors
    lamda1.append(w[0]) # store eigenvalues lamda 1
    lamda2.append(w[1]) # store eigenvalues lamda 2
    v1.append(v[0]) # store eigenvectors vector 1
    v2.append(v[1]) # store eigenvectors vector 2

  ## compute eigenvector associated with the highest eigenvalue
  max_v = []
  for k in range(len(lamda1)):
    if lamda1[k] > lamda2[k]: # compare values of two eigenvalues
      max_v.append(v1[k]) # store eigenvector which has higher eigenvalue
    else:
      max_v.append(v2[k]) # store eigenvector which has higher eigenvalue

  ## compute region orientation θ between the eigenvector associated with the highest eigenvalue and the axis x of the image
  theta  = [] # angle in degrees
  for g in range(len(max_v)):
    dot_product = np.dot(x_v, max_v[g]) # compute dot product of two vector
    arccos = np.arccos(dot_product) # compute arccos values from dot product values
    angle = math.degrees(arccos) # get angle in degree
    theta.append(angle) # store all angles

  ## Extract a signature vector: SV = (x, y, θ, λ1, λ2, σI)
  sv = []
  for i in range(len(theta)):
    s = ([x_refined.tolist()[i], y_refined.tolist()[i], theta[i], lamda1[i], lamda2[i], σI[i]]) # assign values to signature vector
    sv.append(s)

  return lamda1,lamda2,theta,sv

lamda1, lamda2, theta, sv = signature_vector(coordinates,Ixx,Iyy,Ixy,Iyx)
```

2. Show all detected points drawing on the original image circles of radius proportional to the detection scale σ_I

First, we create our keypoints by using parameter in signature vector then the cv2.drawKeypoints is used to display results.

```
## Show all detected points by generating keypoints and drawkeypoints
def draw_keypoints(x_refined,y_refined,σI,theta):
  ## keypoint parameters
  '''
  x x-coordinate of the keypoint
  y y-coordinate of the keypoint
  _size keypoint diameter
  _angle  keypoint orientation
  _response keypoint detector response on the keypoint (that is, strength of the keypoint)
  _octave pyramid octave in which the keypoint has been detected
  _class_id object id
  '''
  x_ = x_refined.tolist() # compute row coordinate to list
  y_ = y_refined.tolist() # compute col coordinate to list

  keypoint=[]
  for i in range(len(x_)):
    kp = cv2.KeyPoint(y_[i], x_[i], σI[i], theta[i], 0, 0, -1) # compute keypoint, note: coordinate in (col,row) order

    keypoint.append(kp) # store all keypoints into a list

  ## flags parameter
  '''
    # DEFAULT = 0,
    # DRAW_OVER_OUTIMG = 1,
    # NOT_DRAW_SINGLE_POINTS = 2,
    # DRAW_RICH_KEYPOINTS = 4
  '''
  img2 = cv2.drawKeypoints(RGB,keypoint,RGB,color=(0,255,0), flags=0) # draw keypoints

  # Displaying the image
  mp.figure(figsize = (16,16))
  mp.imshow(img2)
  mp.show()
  return

draw_keypoints(x_refined,y_refined,σI,theta)
```

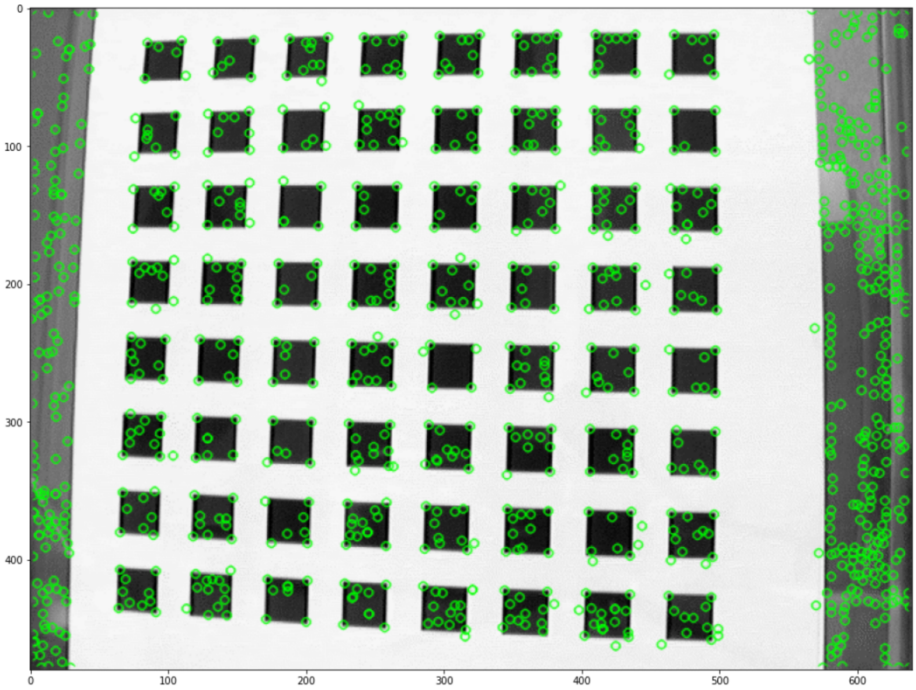


Figure 6. Keypoints result shows in circle

We can see all the corners of black squares in image are detected.

3. Show again all detected points but now draw ellipses

In this time, we draw the points in ellipses instead of circles. The cv2.ellipse() is used.

```
## Show detected points in ellipse according to signature vector SV = (x, y, θ, λ1, λ2, σI)
inversed_refined_coordinates = merge(c, r) # reorganise refined coordinates since cv2.ellipse requires in order (col, row)
def draw_ellipse(inversed_refined_coordinates,lamda1,lamda2,theta):
  center_coordinates = inversed_refined_coordinates # reorganise refined coordinates since cv2.ellipse requires in order (col, row)
  startAngle = 0
  endAngle = 360

  lamda1_n = [math.ceil(20*((float(i)/np.amax(lamda1)))) for i in lamda1] # normalize lamda1 and scale properly for drawing purpose
  lamda2_n = [math.ceil(20*((float(i)/np.amax(lamda2)))) for i in lamda2] # normalize lamda2 and scale properly for drawing purpose
  axesLength = merge(lamda1_n, lamda2_n) # organise eigenvalues into coordiates format to represent axes length of ellipse

  angle = theta # angle of ellipse
  color = (0, 255, 0) # define the color of ellipse
  thickness = 1 # Line thickness of 1 px

  for i in range(len(angle)):
    image = cv2.ellipse(RGB, center_coordinates[i], axesLength[i], angle[i], startAngle, endAngle, color, thickness) # use cv2.ellipse to draw

  # Displaying the image
  mp.figure(figsize = (16,16))
  mp.imshow(image)
  mp.show()
  return

draw_ellipse(inversed_refined_coordinates,lamda1,lamda2,theta)
```

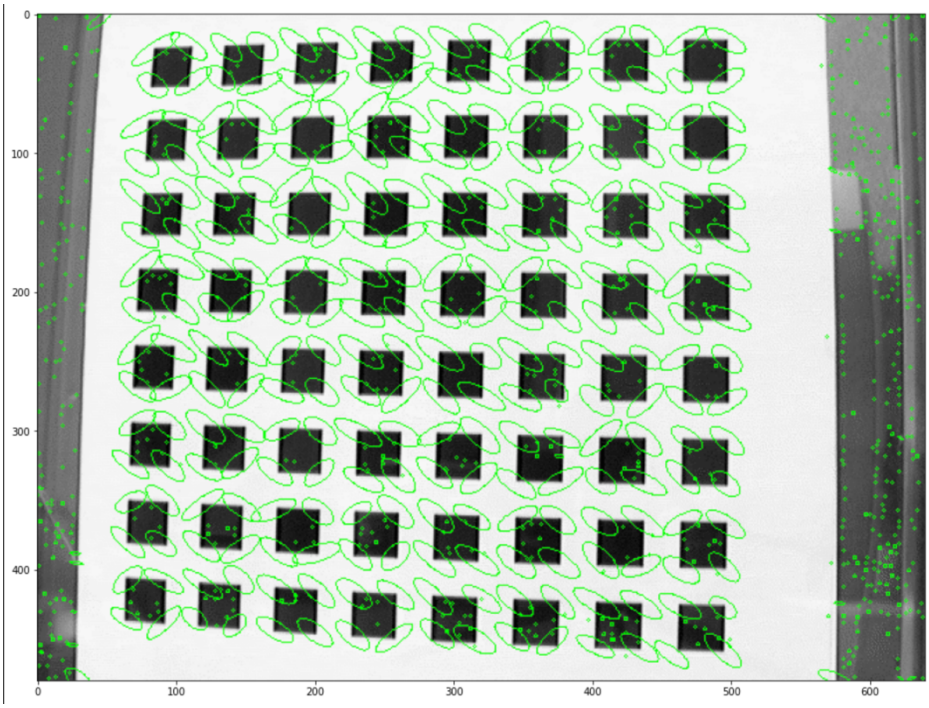


Figure 7. Keypoints result shows in ellipses

We can see all the corners of black square in the image has larger axes length but with different direction. Other ellipses have very small axes length since they are not very strong compare with those in corner of black square.

Verify the Harris algorithm using CalibIm1.png image with different number of strong points selected.

We calibrate the Harris-criteria minimum value to obtain a large enough set (> 2000) of HARRIS points.

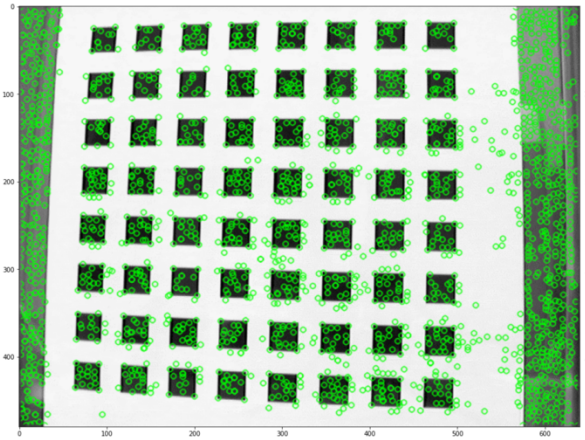


Figure 8. 2000 Keypoints selected (threshold: 953)

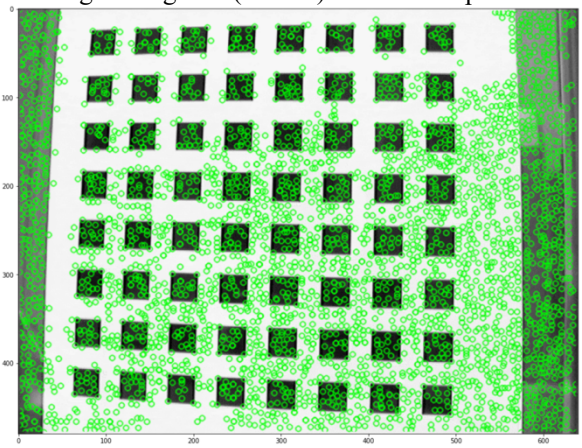


Figure 9. 3505 Keypoints selected (threshold: 90.8)

We can see when we decrease the threshold of R. The more Harris points we compute, the flat regions of image are started been detected. It means that if we want to detect the corner regions of image, it is better to set larger threshold value.

Verify the Harris algorithm using 001.jpg image. 1000 strong points selected in image.

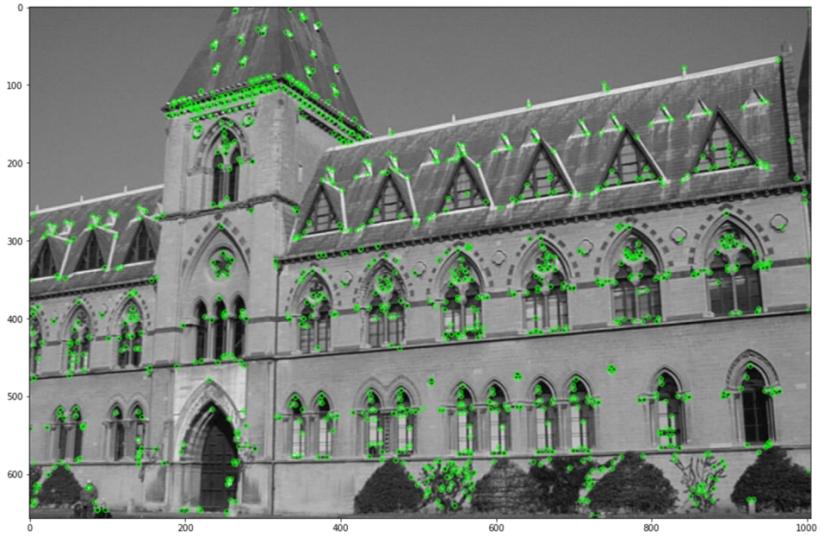


Figure 10. 1000 Keypoints selected in 001.jpg

We can see the most corner region are properly detected.

Bonus

Corner detected in 4 levels gaussian scale-space. Assume $\sigma 0 = 0.5$ and $\sigma k = 1.4 \times \sigma k-1$, $k = 1,2,3,4$, as smoothing sigma on each scale.
We can see when the sigma value increased, the image has higher smooth, the detected points which do not have very corner information will be dismissed. Only the strong corners in image are remained.

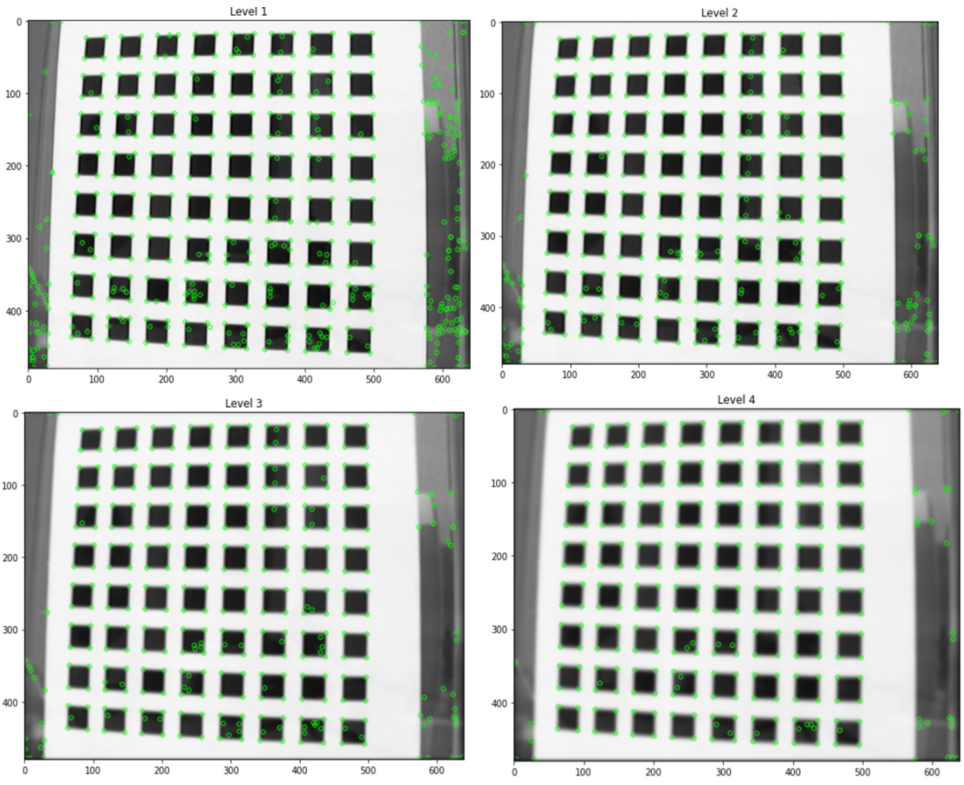


Figure 11. Harris corner detection in 4 level gaussian scale-space