IMPLEMENTATION OF SOME TOPICS LEARNT IN EC405 – COMPUTER VISION

Submitted toward the 3rd test of EC405 – Computer Vision

DEVVJIIT BHUYAN
ECB19050
B. TECH(ECE)
7th semester

Introduction

This report contains the study and implementation of the following topics:

1. Template matching
    a. Using scikit-learn
    b. Using OpenCV
2. Corner detection
    a. Using Harris corner detector
    b. Using Shi-Tomasi corner detector
3. Depth perception (using concept of triangulation)

This report is submitted toward the 3rd test of EC405 – Computer Vision

Template Matching

Template matching is a technique in digital image processing for finding small parts of an image which match a template image. A template matching algorithm uses a 'template', or a part of the image, and tries to find matches of that template in the main image. This is popularly used to search for patterns or objects in images. For example, to track the movement of a football in a game, or to detect eyes or faces in an image, or even in military applications such as aircraft tracking. There can be numerous more applications of template matching in the real world (even in quality control, or VLSI manufacturing, etc.)

Methodology of matching templates

Template matching works similar to most filtering techniques, as in both use a mask which traverses throughout the image and some mathematical operation is performed as it traverses through the pixels. Only that in the case of template matching: the template is the mask, and a correlation operation is performed instead of the regular convolution operation for filters. The correlation operation yields a heatmap, with the probable matches shown by brighter regions in the heatmap.

Simple step-by-step algorithm of template matching:

1. Choose input image [I(x, y)] for template matching
2. Choose a template [T(x, y)] such that it is part of the image or closely resembles an object in the image [I(x, y)].
3. [Optional] Histogram equalization can be applied to enhance contrast as correlation may not work very well in low contrast images. Also, images can be filtered using advanced filters such as Weiner2 filter to remove most of the noise.
4. Start traversing the mask [T(x, y)] through the Image matrix.
    a. Compute cross correlation:

$$\text{CC} = \sum_{s=-1}^{1}\sum_{t=-1}^{1} T(s,t) * I(s,t)$$

    b. Normalize the cross correlation:

$$\text{NCC} = \frac{CC(I,T)}{std\_dev(I) * std\_dev(T)}$$

5. Select maximum in the NCC heatmap if single object is to be detected. Else locate multiple maxima
6. Map the maximum(s) into the original image and draw squares around the detected objects.

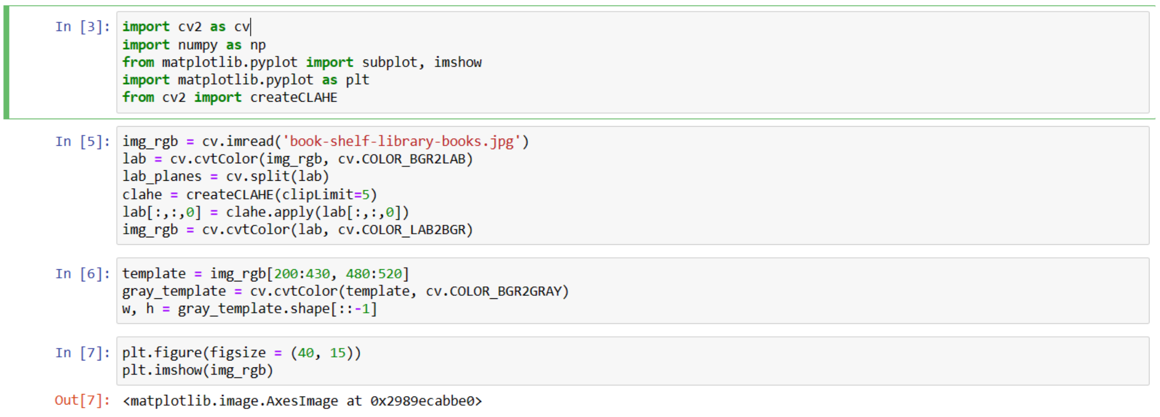
Robust algorithm for stable template matching of two images [in cases where the distortion between prominent objects within the template]:

Also known as 'Distortion Tolerant Matching'

1. Compute distance matrix D_{ij};
   i: i th region of template,
   j: j th region of image.
2. Calculate forward matching matrix C_{ij}:
   $C_{ij} = 1$ if $Dij < D_{ik}$ for all k 6= j;
   otherwise, $C_{ij} = 0$.
3. Calculate backward matching matrix B_{ij}:
   $B_{ij} = 1$ if $D_{ij} < D_{kj}$ for all k 6= i;
   otherwise, $B_{ij} = 0$.
4. Match regions i and j if $C_{ij}B_{ij} = 1$.
5. Remove established correspondences from D_{ij}.
6. Iterate until no further matching is possible.

Implementation in Python

The scikit-learn library can only process grayscale images, and also is much less efficient for template matching purpose, hence we have discarded the sklearn implementation and we shall stick to the OpenCV implementation:



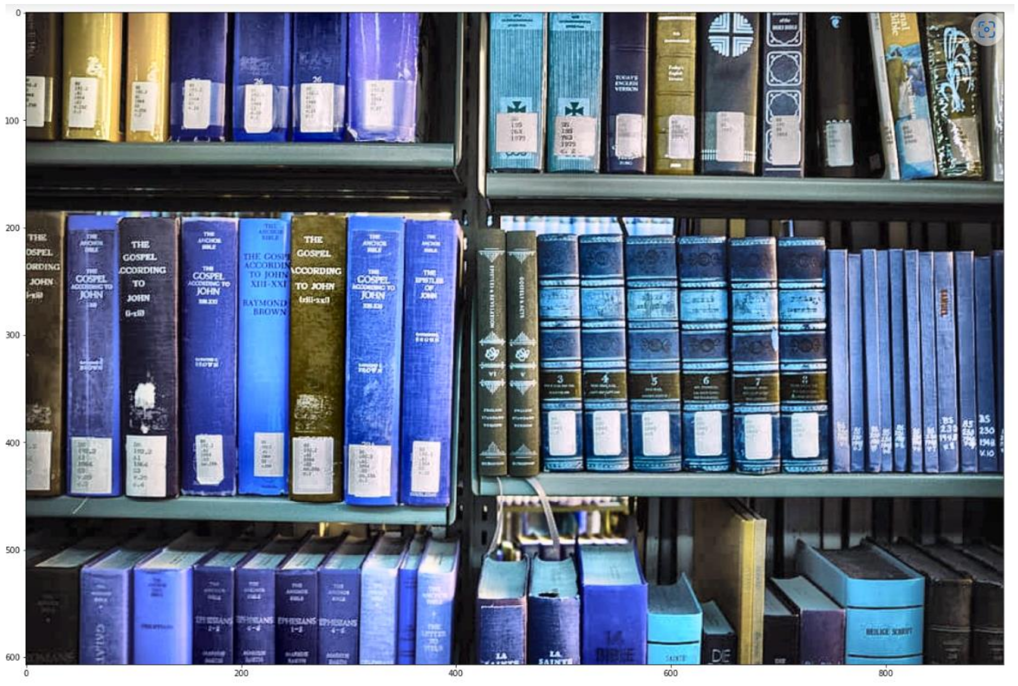
```
In [3]: import cv2 as cv
        import numpy as np
        from matplotlib.pyplot import subplot, imshow
        import matplotlib.pyplot as plt
        from cv2 import createCLAHE
```

```
In [5]: img_rgb = cv.imread('book-shelf-library-books.jpg')
        lab = cv.cvtColor(img_rgb, cv.COLOR_BGR2LAB)
        lab_planes = cv.split(lab)
        clahe = createCLAHE(clipLimit=5)
        lab[:,:,0] = clahe.apply(lab[:,:,0])
        img_rgb = cv.cvtColor(lab, cv.COLOR_LAB2BGR)
```

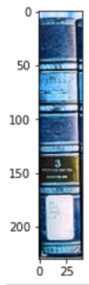
```
In [6]: template = img_rgb[200:430, 480:520]
        gray_template = cv.cvtColor(template, cv.COLOR_BGR2GRAY)
        w, h = gray_template.shape[::-1]
```

```
In [7]: plt.figure(figsize = (40, 15))
        plt.imshow(img_rgb)
```

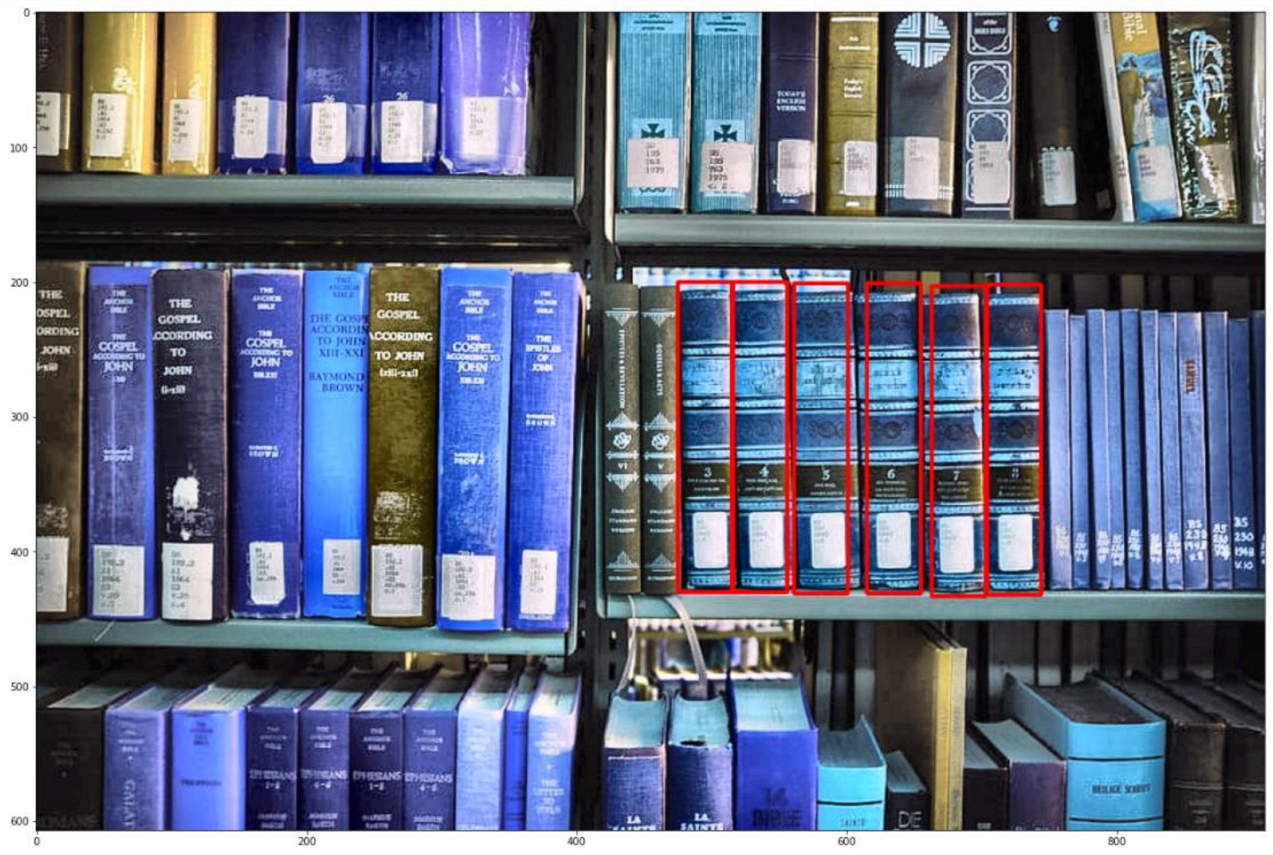
```
Out[7]: <matplotlib.image.AxesImage at 0x2989ecabbe0>
```



```
In [8]:  imshow(template)
Out[8]:  <matplotlib.image.AxesImage at 0x2989eb73bb0>
```



```
In [13]: res = cv.matchTemplate(gray, gray_template, cv.TM_CCOEFF_NORMED)
         threshold = 0.6
         loc = np.where( res >= threshold)
         locs = []
         prev = (0, 0)
         for i in sorted(zip(*loc[::-1])):
             if abs(i[0]-prev[0]) + abs(i[1]-prev[1])>29:
                 locs.append(i)
             prev = i
         for pt in locs:
             cv.rectangle(img, pt, (pt[0] + w, pt[1] + h), (255,0,0), 2)
         plt.figure(figsize=(40, 15))
         plt.imshow(img)
Out[13]: <matplotlib.image.AxesImage at 0x2989fa349a0>
```



```
In [10]: locs
Out[10]: [(475, 200), (516, 200), (561, 201), (615, 201), (663, 203), (704, 202)]
```

In our case, we have used CLAHE (Contrast-Limited Adaptive Histogram Equalization) for equalizing the contrast, and used the Normalized cross-correlation method to find matches. This method so far gives good results in single as well as multiple object detection, as well as tackles any brightness/contrast issues.

The weiner filter can be applied to an image if there is excessive noise, however, it would limit the channels of the image to 1, hence we haven't used it in this example.

Corner detection

Corners are locations where variations of intensity in both x and y directions are high. These are important interest points in an image because unlike edges, they are unaffected by the aperture problem, and are highly directional. They are local, precisely localised, and distinctive.

Harris Corner detector

The Harris detector works by traversing a window over the image, generating a cornerness map, and then calculating the responses. Unlike Moravec detector, which detects only in 8 directions, Harris detector doesn't detect in discrete directions, rather it relies on directional gradients to create the cornerness map. Hence it doesn't give large cornerness value to edges which are not in the 8 principal directions.

Harris operator uses a window function of squared differences:

$$S_W(\Delta x, \Delta y) = \sum_{x_i, y_i} w(x_i, y_i)\left[f(x_i, y_i) - f(x_i - \Delta x, y_i - \Delta y)\right]^2$$

Where w(x, y) is a Gaussian window.

This equation after Taylor approximation yields:

$$S_W(\Delta x, \Delta y) = [\Delta x, \Delta y]\, A_W(x, y) \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix}$$

Where:

$$A_W(x,y) = \begin{bmatrix} \sum_{x_i \in W} \sum_{y_i \in W} \left(\frac{\partial f(x_i, y_i)}{\partial x}\right)^2 & \sum_{x_i \in W} \sum_{y_i \in W} \frac{\partial f(x_i, y_i)}{\partial x} \frac{\partial f(x_i, y_i)}{\partial y} \\ \sum_{x_i \in W} \sum_{y_i \in W} \frac{\partial f(x_i, y_i)}{\partial x} \frac{\partial f(x_i, y_i)}{\partial y} & \sum_{x_i \in W} \sum_{y_i \in W} \left(\frac{\partial f(x_i, y_i)}{\partial y}\right)^2 \end{bmatrix}$$

OR,

$$A_W = \begin{bmatrix} \sum_{x,y} w(x,y) f_x^2 & \sum_{x,y} w(x,y) f_x f_y \\ \sum_{x,y} w(x,y) f_x f_y & \sum_{x,y} w(x,y) f_y^2 \end{bmatrix} = \sum_{x,y} w(x,y) \begin{bmatrix} f_x^2 & f_x f_y \\ f_x f_y & f_y^2 \end{bmatrix}$$

Which can be decomposed as:

$$A_W = R^{-1} \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} R$$

Where λ_1 and λ_2 denote the eigenvalues of A_W

If λ_1 and λ_2 are both small: Location is a <u>blob</u>

If <u>only one</u> of λ_1 and λ_2 is large: Location is an <u>edge</u>

If λ_1 and λ_2 are both large: Location is a <u>corner</u>

The Cornerness response is given by:

$$R = \det M - k\left(\operatorname{trace} M\right)^2$$

Large positive values indicate corners

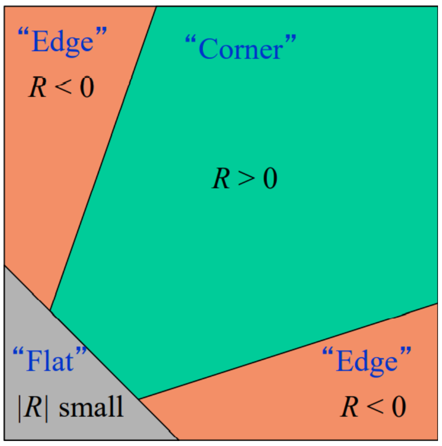


Fig: Cornerness response mapping

Algorithm:

1. Compute image gradients I_x, I_y for all pixels
2. For each pixel:
   a. Compute A_w
   b. Compute $R(A_w)$
3. Find points with large corner response function R (R > threshold)
4. Take the points of locally maximum R as the detected feature points (i.e., pixels where R is bigger than for all the 4 or 8 neighbours).

Implementation in Python

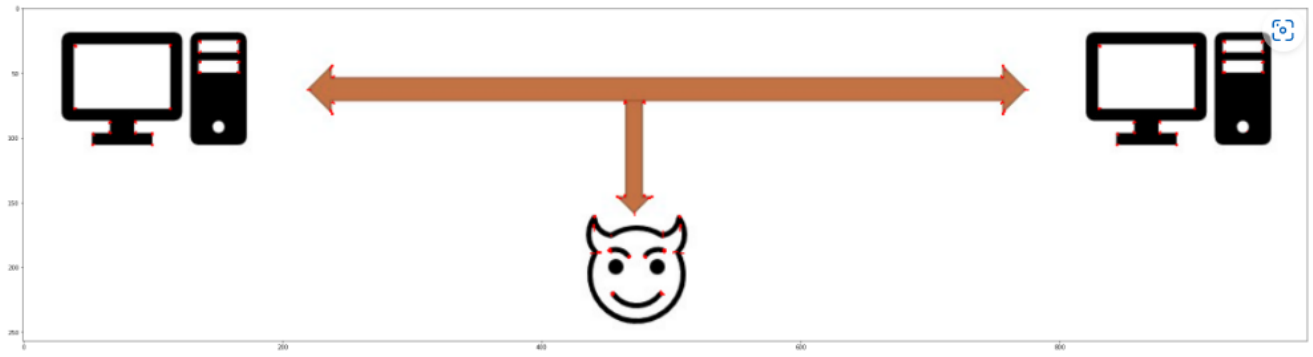
Using OpenCV

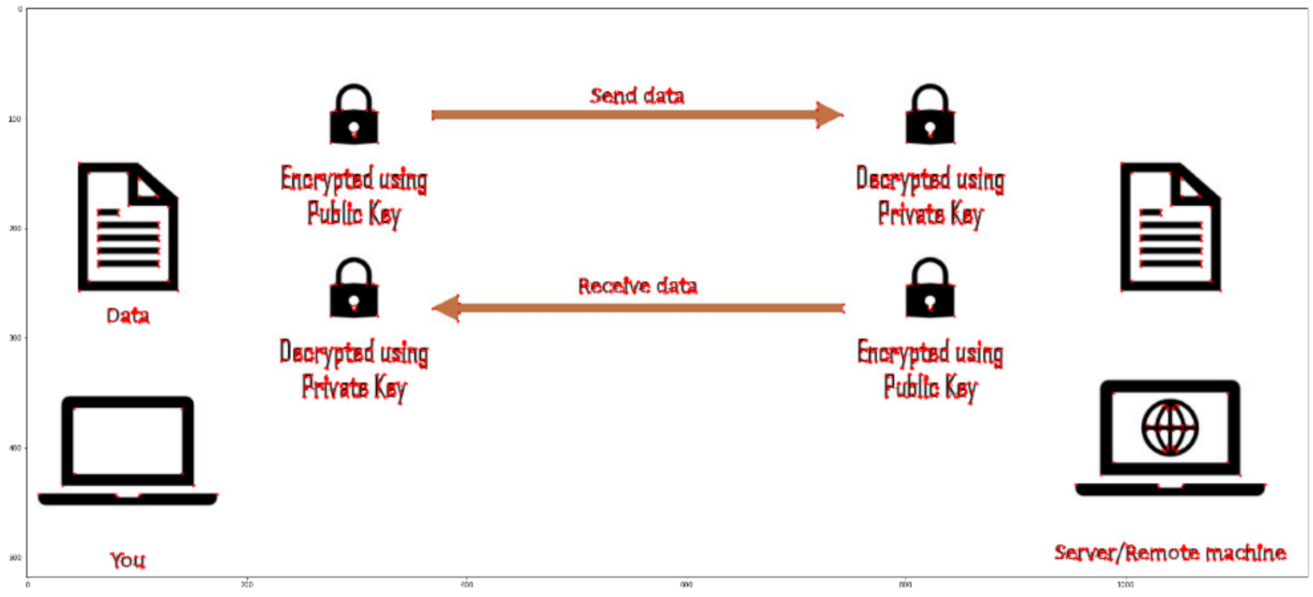
```
In [1]: import numpy as np
        import cv2 as cv
        import matplotlib.pyplot as plt
        import os
```

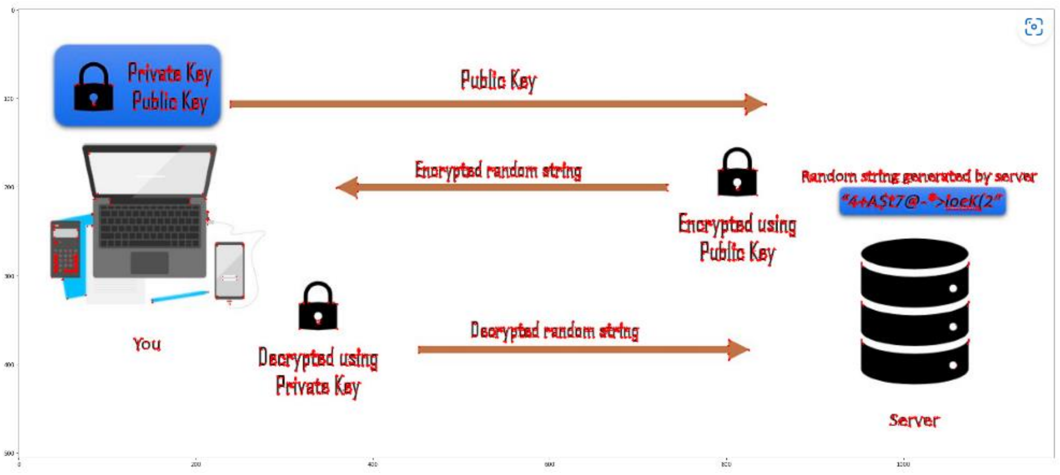
```
In [2]: cwd = 'C:/Users/Asus/Desktop/'
        os.chdir(cwd)
```

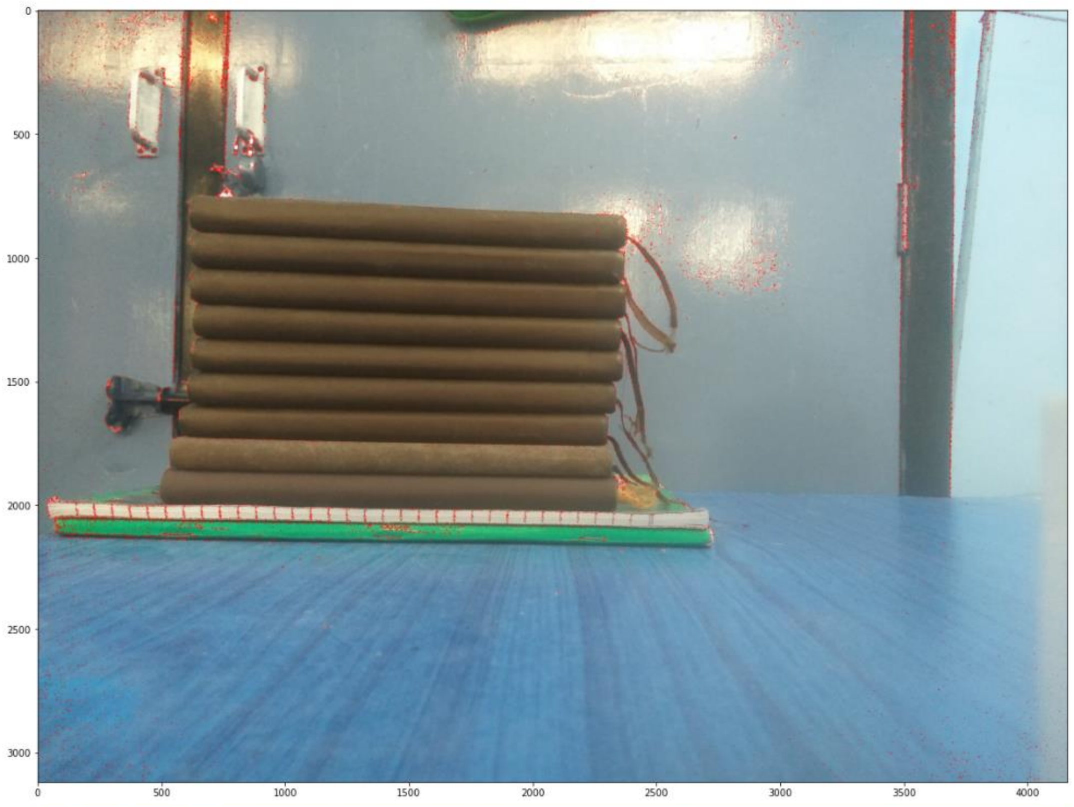
```
In [3]: images = os.listdir('C:/Users/Asus/Desktop/')
        imgs = []
        for i in images:
            if i[-3:] == 'jpg':
                imgs.append(i)
```

```
In [4]: for file in imgs:
            img = cv.imread(file)
            gray = cv.cvtColor(img,cv.COLOR_BGR2GRAY)
            dst = cv.cornerHarris(gray,2,3,0.04)
            mx = dst.max()
            threshold = 0.00018
            img[dst>threshold*mx]=[255,0,0]
            plt.figure(figsize=(40, 15))
            plt.imshow(img)
```




Send data
Encrypted using Public Key
Decrypted using Private Key
Data
Receive data
Decrypted using Private Key
Encrypted using Public Key
You
Server/Remote machine


Private Key
Public Key
Public Key
Encrypted random string
Random string generated by server
"4+A$t7@-*>loeK(2"
Encrypted using Public Key
You
Decrypted using Private Key
Decrypted random string
Server



At this fine-tuning, the algorithm is pretty much robust to multiple kinds of images, even those with lower contrast.

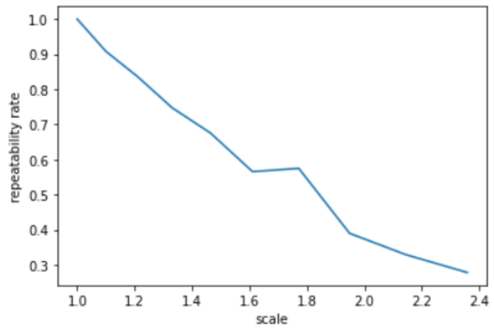
Repeatability rate:

```
In [23]: numc_prev = 1
         rr = []
         scale = []
         for i in range(10):
             img = cv.imread('C:/Users/devbh/Desktop/raptor.jpg')
             img = cv.resize(img, (0, 0), fx = 1.1**i, fy = 1.1**i)
             gray = cv.cvtColor(img,cv.COLOR_BGR2GRAY)
             dst = cv.cornerHarris(gray,2,3,0.04)
             mx = dst.max()
             threshold = 0.00018
             img_r = img
             img_r[dst>threshold*mx]=[255,0,0]
             plt.figure(figsize=(40, 15))
             plt.imshow(img_r)
             numc = numcorners(dst, threshold, mx)
             repeatability = 3715/numc
             rr.append(repeatability)
             scale.append(1.1**i)
             print(numc, repeatability)
```

```
3715 1.0
4091 0.9080909313126375
4442 0.8363349842413327
4972 0.747184231697506
5499 0.6755773777050372
6568 0.5656211936662606
6462 0.5748994119467657
9532 0.3897398237515736
11279 0.32937317138044153
13347 0.2783397018056492
```

```
In [26]: import numpy as np
         rr = np.asarray(rr)
         scale = np.asarray(scale)
```

```
In [29]: plt.plot(scale, rr)
         plt.xlabel('scale')
         plt.ylabel('repeatability rate')
         plt.show()
```



The repeatability falls with increase in scale as Harris corner detector isn't scale invariant. A better version of the Harris detector, the Harris-Laplace detector, which is also scale invariant, can be used to detect corners in multiple scales, however in our case, a Harris corner detector serves the purpose for detecting corners in real world images. For more advanced applications such as Image matching or image stitching, a Harris-Laplace, or SIFT can be used.

Depthmap computation

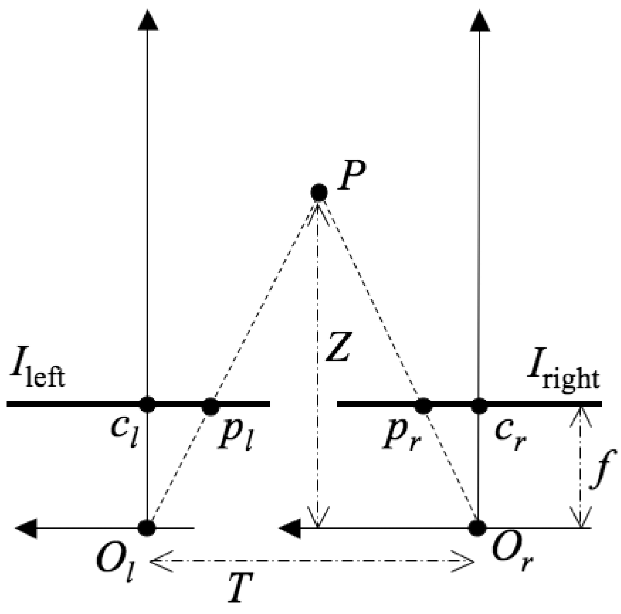
Stereo vision can be used to perceive real world objects using a set of 2-D cameras. The two views from both cameras can be used to reconstruct a 3-D view of the actual object. It uses concepts of projection, parallax, and concepts such as triangulation to accurately locate an object in the real-world space. Stereo reconstruction is built upon this aspect of stereo vision.

A distance map can be used in many applications such as 3-D Reconstruction, finding the location of distant celestial bodies, etc.

In simple terms it works on the concept of parallax – "A farther object will move lesser than a closer object when the viewing angle is changed". This parallax concept is extended to form the Triangulation technique, which is used to compute the Depthmap.

Triangulation

This technique is applied to pinpoint the depth of an object in real world space from the camera baseline. All we need is a disparity map (a map which shows how each keypoint varies between the two stereo images), knowledge about the camera parameters and stereo setup.



In the adjacent figure, considering T is the baseline, Z is the object depth, f is the focal length, and x_l = c_lp_l, and x_r = c_rp_r. Then we can derive:

$$Z = \frac{T*f}{x_l+(-x_r)}$$

Using this formula, for a given disparity map, we can create a depthmap for the scene.

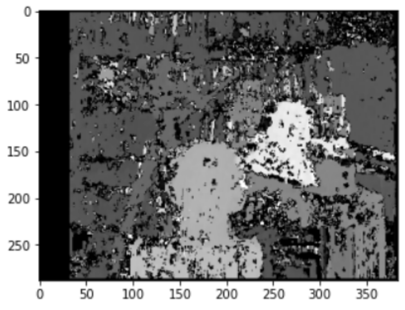
Algorithm:

1. Draw correspondences between the two stereo images using methods such as SIFT (Scale-Invariant Feature Transform)
2. Use these correspondences to create a disparity map
3. Use the triangulation technique to find the depth corresponding to that pixel in the disparity map.

Implementation in Python

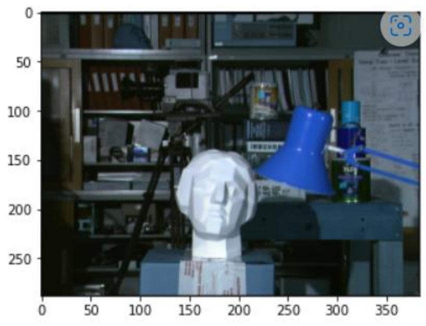
We have used the famous tsukaba stereo images for this task. The two images differ by a slight camera translation. Here the camera parameters are: focal length = 615 pixels (given), baseline = 10 cm.

```
In [1]: import numpy as np
        import cv2
        import matplotlib.pyplot as plt

        imgL = cv2.imread('C:/Users/Asus/Desktop/tsukaba_l.png', 0)
        imgR = cv2.imread('C:/Users/Asus/Desktop/tsukaba_r.png', 0)

        stereo = cv2.StereoBM_create(numDisparities=32, blockSize=5)
        disparity = stereo.compute(imgL, imgR)
        plt.imshow(disparity,'gray')
        plt.show()
        print(32, 5)
```

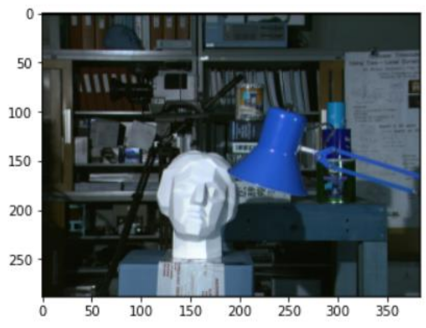


```
32 5
```

```
In [2]: im = cv2.imread('C:/Users/Asus/Desktop/tsukaba_l.png')
        plt.imshow(im)

Out[2]: <matplotlib.image.AxesImage at 0x1dcf3652790>
```



```
In [3]: im = cv2.imread('C:/Users/Asus/Desktop/tsukaba_r.png')
        plt.imshow(im)

Out[3]: <matplotlib.image.AxesImage at 0x1dcf36bfe80>
```

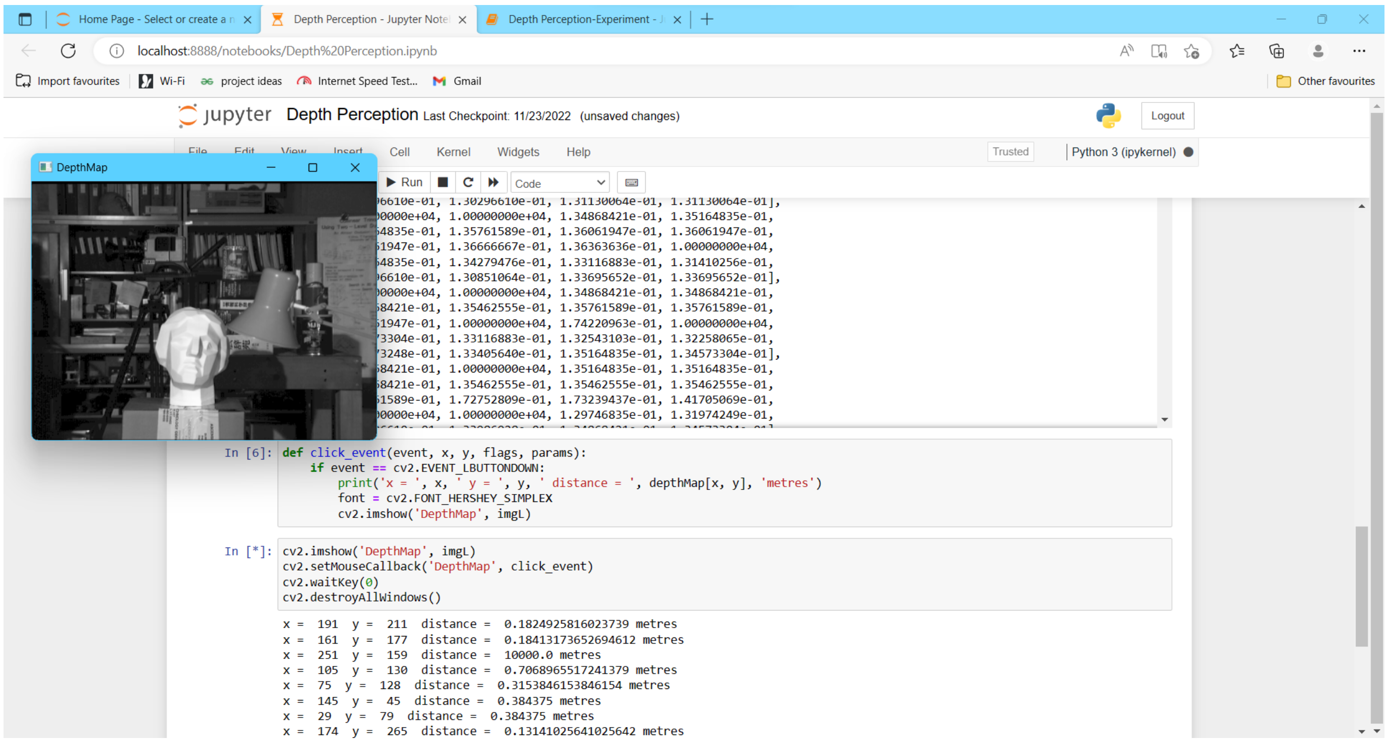


```
In [4]: depthMap = np.zeros(disparity.shape)
        Focal_length = 615
        cameraBaseline = 0.1
        h, w = disparity.shape
        for i in range(h):
            for j in range(w):
                if disparity[i, j] > 0:
                    depthMap[i, j] = (cameraBaseline * Focal_length) / disparity[i, j]
                else:
                    depthMap[i, j] = 10000
```

```
In [6]: def click_event(event, x, y, flags, params):
            if event == cv2.EVENT_LBUTTONDOWN:
                print('x = ', x, ' y = ', y, ' distance = ', depthMap[x, y], 'metres')
                font = cv2.FONT_HERSHEY_SIMPLEX
                cv2.imshow('DepthMap', imgL)
```

```
In [9]: cv2.imshow('DepthMap', imgL)
        cv2.setMouseCallback('DepthMap', click_event)
        cv2.waitKey(0)
        cv2.destroyAllWindows()
```

```
x =  191  y =  211  distance =  0.1824925816023739 metres
x =  161  y =  177  distance =  0.18413173652694612 metres
x =  251  y =  159  distance =  10000.0 metres
x =  105  y =  130  distance =  0.7068965517241379 metres
x =  75  y =  128  distance =  0.3153846153846154 metres
x =  145  y =  45  distance =  0.384375 metres
x =  29  y =  79  distance =  0.384375 metres
x =  174  y =  265  distance =  0.13141025641025642 metres
```



```
localhost:8888/notebooks/Depth%20Perception.ipynb
Jupyter Depth Perception Last Checkpoint: 11/23/2022 (unsaved changes)
```

```
)6610e-01, 1.30296610e-01, 1.31130064e-01, 1.31130064e-01],
)0000e+04, 1.00000000e+04, 1.34868421e-01, 1.35164835e-01,
54835e-01, 1.35761589e-01, 1.36061947e-01, 1.36061947e-01,
51947e-01, 1.36666667e-01, 1.36363636e-01, 1.00000000e+04,
54835e-01, 1.34279476e-01, 1.33116883e-01, 1.31410256e-01,
)6610e-01, 1.30851064e-01, 1.33695652e-01, 1.33695652e-01],
)0000e+04, 1.00000000e+04, 1.34868421e-01, 1.34868421e-01,
58421e-01, 1.35462555e-01, 1.35761589e-01, 1.35761589e-01,
51947e-01, 1.00000000e+04, 1.74220963e-01, 1.00000000e+04,
73304e-01, 1.33116883e-01, 1.32543103e-01, 1.32258065e-01,
73248e-01, 1.33405640e-01, 1.35164835e-01, 1.34573304e-01],
58421e-01, 1.00000000e+04, 1.35164835e-01, 1.35164835e-01,
58421e-01, 1.35462555e-01, 1.35462555e-01, 1.35462555e-01,
51589e-01, 1.72752809e-01, 1.73239437e-01, 1.41705069e-01,
)0000e+04, 1.00000000e+04, 1.29746835e-01, 1.31974249e-01,
```

```
In [6]: def click_event(event, x, y, flags, params):
            if event == cv2.EVENT_LBUTTONDOWN:
                print('x = ', x, ' y = ', y, ' distance = ', depthMap[x, y], 'metres')
                font = cv2.FONT_HERSHEY_SIMPLEX
                cv2.imshow('DepthMap', imgL)

In [*]: cv2.imshow('DepthMap', imgL)
        cv2.setMouseCallback('DepthMap', click_event)
        cv2.waitKey(0)
        cv2.destroyAllWindows()

        x =  191  y =  211  distance =  0.1824925816023739 metres
        x =  161  y =  177  distance =  0.18413173652694612 metres
        x =  251  y =  159  distance =  10000.0 metres
        x =  105  y =  130  distance =  0.7068965517241379 metres
        x =  75  y =  128  distance =  0.3153846153846154 metres
        x =  145  y =  45  distance =  0.384375 metres
        x =  29  y =  79  distance =  0.384375 metres
        x =  174  y =  265  distance =  0.13141025641025642 metres
```

Reference Image for point selection

In this attempt we have been able to fetch distance values for various points, and as per our assumptions they seem to be close to the actual values. However, since we do not have the data of the actual distances from the camera setup, we cannot prove the viability of this method.

An actual analysis of the method on a set of real-world test images in a real environment can be performed to validate the results.