KLT Point Tracking algorithm

(1) Clone the git repo and
(2) Get it to run

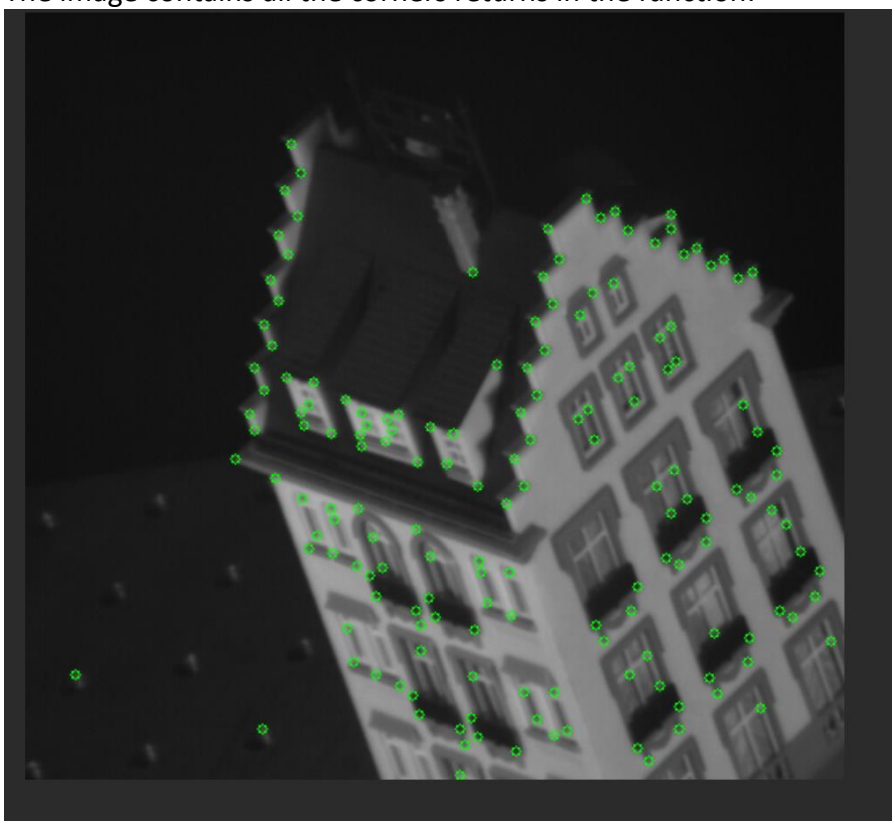
That was done as a copy on the private google colab account. The hotel images had to be copied so the it would work.

(3) Have a look at the code and spot he point detection (Harris Corners)

In the example, the writer uses a method called cv2.goodFeaturesToTrack to find starting points. The function, keypoint_detector, calculates Harris Corners. First, it figures out the range using the window's movement, then it sets up two filters, which are like the sable filter. These filters are used on the original image to find horizontal and vertical features. This process creates three matrices, which are used for each pixel in a certain range of the moving window. The window moves a lot (half the window size), but it checks each pixel one by one. This might not be very efficient. Then, the method for detecting Harris Corners is used with these matrices and it calculates the total for a window. This helps to work out the determinant and the trace for each pixel, using the formula det – k*(trace**2), where k is a special weight set at 0.04. If a pixel meets the threshold, it's added to the corners list. Next, the process goes through all the corners and sets any corner too close to another to zero, so it's not counted twice. The number 5 is hardcoded in the code, which is odd since it should be changeable (should be a parameter). Then, it goes through the list again and removes any corners set to zero. Finally, the list of corners is returned.

```python
def keypoint_detector(input_img, k, window_size, threshold):
    #print(input_img.shape)
    corners = []
    shift = int(window_size/2)
    y_range = input_img.shape[0] - shift
    x_range = input_img.shape[1] - shift

    kernel_x = np.array([[-1, 0, 1],[-2, 0, 2],[-1, 0, 1]])
    kernel_y = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]])

    dx = linear_filter(input_img, kernel_x)
    dy = linear_filter(input_img, kernel_y)

    Ixx = dx**2
    Ixy = dy*dx
    Iyy = dy**2

    for y in range(shift, y_range):
        for x in range(shift, x_range):
            start_y = y - shift
            end_y = y + shift + 1
            start_x = x - shift
            end_x = x + shift + 1
            windowIxx = Ixx[start_y : end_y, start_x : end_x]
            windowIxy = Ixy[start_y : end_y, start_x : end_x]
            windowIyy = Iyy[start_y : end_y, start_x : end_x]
            Sxx = windowIxx.sum()
            Sxy = windowIxy.sum()
            Syy = windowIyy.sum()

            det = (Sxx * Syy) - (Sxy**2)
            trace = Sxx + Syy
            r = det - k*(trace**2)

            if r > threshold:
                corners.append([x, y])
    # NMS over a 5x5 neighborhood
    for i in range(len(corners)):
      for j in range(i+1,len(corners)):
        if((corners[i][0] +5) >= corners[j][0] and (corners[i][0] -5) < corners[j][0] and (corners[i][1] +5) >= corners[j][1] and (corners[i][1] -5) < corners[j][1]):
          corners[j][0] =0
          corners[j][1] =0
    l = len(corners)
    i=0
    while(l):
      if(corners[i][0] == 0):
        corners.pop(i)
      else:
        i=i+1
      l=l-1

    corners = np.asarray(corners)
    return corners
```

The image contains all the corners returns in the function:



Based on the picture above, we can see that some points at the back were detected incorrectly on a homogenous background, and some points on the windows at the front were not detected as corner points.

(4) Have a look at the code and spot the KLT tracker function.

```
def trackPoints(xy, imageSequence):
  print ("In function trackPoints")
  print (f'length of imageSequence = {len(imageSequence)}')
  movedOutFlag = np.zeros(xy.shape[0])
  # initialize xyt to contain any information that is needed for drawing paths at the end of tracking
  # also add code in this function as needed to maintain xyt
  #xyt = 0

  # xyt is initialized as a list, to which the new points from getNextPoints can be added
  xyt = []

  for t in range(0, len(imageSequence)-1): # predict for all images except first in sequence
    print (f't = {t}; predicting for t = {t+1}')
    xy2, movedOutFlag = getNextPoints(imageSequence[t], imageSequence[t+1], xy, movedOutFlag)
    xy = xy2

    for pt in xy2:
      xyt.append(pt)

    # for selected instants in time, display the latest image with highlighted keypoints
    if ((t == 0) or (t == 10) or (t == 20) or (t == 30) or (t == 40) or (t == 49)):
      im2color = cv2.cvtColor(imageSequence[t+1], cv2.COLOR_GRAY2BGR)
      corners = np.intp(np.round(xy2))

      for c in range(0, corners.shape[0]):
        if movedOutFlag[c] == False:
          x = corners[c][0]
          y = corners[c][1]
          cv2.circle(im2color, (x, y), DISPLAY_RADIUS, GREEN)
      cv2_imshow(im2color)

  return xyt
```

The track function is named getNextPoints (as shown in the image below). It uses the getNextPoints function too. There are several issues with its simplicity and efficiency. It has many nested for loops and uses extra numpy functions for basic calculations. Inside these loops, at certain frames, it displays images with their tracked paths and new corner points. The code took 30 seconds to process 50 frames, which is too slow for a tracker that needs to work on every frame of a video (only 2 frames per second).

```python
def getNextPoints(im1, im2, xy, movedOutFlag):
  print("In function getNextPoints")

  xy2 = np.copy(xy).astype(float)
  im1 = im1.astype(np.float32)
  im2 = im2.astype(np.float32)

  # Gaussian kernel for smoothing
  kernel = np.array([
          [1, 4, 7, 4, 1],
          [4, 16, 26, 16, 4],
          [7, 26, 41, 26, 7],
          [4, 16, 26, 16, 4],
          [1, 4, 7, 4, 1,]], dtype=np.float32) / 273.0

  img = linear_filter(im1, kernel).astype(np.float32)
  Iy, Ix =  np.gradient(img)

  # The given KLT algorithm is implemented
  for i in range(len(xy)):
    patch_x = cv2.getRectSubPix(Ix, (15,15), (xy[i,0], xy[i,1]))
    patch_y = cv2.getRectSubPix(Iy, (15,15), (xy[i,0], xy[i,1]))
    A = np.array([[np.sum(patch_x * patch_x), np.sum(patch_x * patch_y)], [np.sum(patch_x * patch_y),
     np.sum(patch_y * patch_y)]])

    for j in range(25):
      patch_t = cv2.getRectSubPix(im2, (15,15), (xy2[i,0], xy2[i,1])) - cv2.getRectSubPix(img, (15,
       15), (xy[i,0], xy[i,1]))
      B = -1* np.array([[np.sum(patch_x*patch_t)],[np.sum(patch_y*patch_t)]])
      disp = np.matmul(np.linalg.pinv(A), B)

      u = disp[0]
      v = disp[1]

      xy2[i] = [xy2[i,0] + u, xy2[i,1] + v]

      # Checking if the norm of (u, v) is leser than the threshold (from the textbook section - 9.1.3
       Incremental refinement)
      if np.hypot(u, v) <= 0.01:
        break

  # Setting the movedOutFlag to 1 if the new pixels are out of bounds
  if xy2[i,0] >= len(im1) or xy2[i,0] < 0 or xy2[i,1] >= len(im1[0]) or xy2[i,1] < 0: Local variable 'i' mig
    movedOutFlag[i] = 1

  return(xy2, movedOutFlag)
```

In this part, the Jacobian matrix, referred to as A, is computed in the first for loop. In the second loop, I[t], which the author calls B, is calculated. The author uses "np.matmul(np.linalg.pinv(A), B)" to do this. (However, they could have used "np.linalg.inv(A)@B" or the QR algorithm for efficiency to calculate it). This process computes the vector that shows the displacement (u and v). Then, the window is moved by this displacement vector. The second loop runs a maximum of 25 times or stops when the Pythagorean sum of u and v is below a certain threshold (indicating minimal progress).

The result is the following image for each of the defined key frames. The tracking points on the house have a defined path that follows the camera movement, however the background points do not follow this pattern, since they can hardly be distinguished.