```
In [1]: import cv2
        import numpy as np

In [2]: def display_cv(image):
            cv2.imshow("Image",image)
            if cv2.waitKey(0) == 27:
                cv2.destroyAllWindows()
            else:
                print("Error")
                cv2.destroyAllWindows()
            if cv2.waitKey(1) == ord('q'):
                break
            elif cv2.waitKey(1) == ord('s'):
                cv2.imwrite('Image.jpg',image)
                print("Saved")
                break
            else:
                pass
            cv2.destroyAllWindows()

In [3]: display_cv(img_path)
```

## Optical Flow

Optical flow is known as the pattern of apparent motion of objects. It is the motion of objects between every two consecutive frames of the sequence, which is caused by the relative movement between the camera and the scene. Assuming it. Consider an object with intensity I(x, y, t) which moves by distance (x, y) in the next frame taken after a time step. So the same intensity would be. Consider an object with intensity $I(x, y, t)$ which moves by distance $(x, y)$ in the next frame taken after a time step. So the same intensity would be.

$$I(x, y, t) \longrightarrow I(x+dx, y+dy, t+dt)$$

### Shi-Tomasi Corner Detector & Good Features

- Tomasi made a small modification to it in their paper Good Features to Track which shows better results.

$$M = \lambda_1 \lambda_2$$

- It is a question that a threshold value, it is considered as a corner. If we set it to $\lambda_1$ and $\lambda_2$, it will go in image as below:

- From this figure, you can see that only with $\lambda_1$ and $\lambda_2$ greater than a minimum value, $\lambda_1 = \lambda_2 = 0.5$ is considered as a corner(green region).

- Image score if it is floating point 32 bit, single-channel image.
- Image quality specifies minimum pixel intensity.
- Image quality and minimum distance are floating values, which are then scaled and thus are thus the corresponds of them to minimum. `qualityLevel = 0.01` means that any corner with quality less than 1% of the best corner will be rejected. Other corners are also rejected if they are too close to the best corner.
- Image minimum distance specifies the minimum pixel distance between corners. This parameter value is a factor of the image width. Suppose, if the image width is 640 and the minimum distance is 0.05, then the minimum distance between two corners will be 640 * 0.05 = 32 pixels. Any corner which is more than 32 pixels away from the best corner is considered.
- Image mask specifies the region of interest in which the corners are to be detected. This parameter value is a binary image of the same size as the image. Pixels with non-zero values are considered for corner detection. Image mask can be used to specify the region of interest in which the corners are to be detected. Image mask can be used to specify the region of interest in which the corners are to be detected.
- Image block size specifies the size of the neighborhood considered for corner detection. This parameter value is an odd number, typically 3, 5, or 7. Image block size specifies the size of the neighborhood considered for corner detection. This parameter value is an odd number, typically 3, 5, or 7.
- Image use Harris detector specifies whether to use the Harris corner detector or the Shi-Tomasi corner detector. Image use Harris detector specifies whether to use the Harris corner detector or the Shi-Tomasi corner detector.
- Image k specifies the Harris detector free parameter. This parameter value is a small positive number, typically 0.04. Image k specifies the Harris detector free parameter. This parameter value is a small positive number, typically 0.04.

### Lucas-Kanade Optical Flow

#### Parameters for Lucas Kanade Optical Flow

Where the motion of a pixel is equal to the negative of the gradient of intensity in modelling the workflow sequence. This determines the integration window size. Small windows are more sensitive to noise and may miss larger motions. Large windows will "survive" an occlusion.

- maxLevel 0 means that only the original image is used. If set to 1, the algorithm will use the original image and the first pyramid level, and so on.
- Criteria specifies the termination criteria for the iterative search algorithm (CV_criteria). Where iterations mean a more exhaustive search, and a smaller epsilon means better. These are particularly useful in tracking scenes in accuracy, but mainly slay the same.
- Flags: maxLevel = 0 is the same algorithm without using pyramids, so maxLevel=0 and Flags=cv2.OPTFLOW_USE_INITIAL_FLOW means that the initial estimate of the displacement vector is taken from the input nextPts array, instead of "nuclear" an estimate.

```
In [4]: # Lucas-kanade optical flow parameter
        lk_params = dict(winSize = (15,15),
                         maxLevel = 2,
                         criteria = (cv2.TERMINATE_EPSILON | cv2.TERM_CRITERIA_COUNT, 10, 0.03))

In [5]: def lucas_kanade(image,lk_params):
            cap = cv2.VideoCapture(image)
            ret, old_frame = cap.read()
            old_gray = cv2.cvtColor(old_frame, cv2.COLOR_BGR2GRAY)
            p0 = np.array([[1000, 1000]], dtype=np.float32)
            mask = np.zeros(old_frame.shape)
            while True:
                ret, frame = cap.read()
                if ret:
                    frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
                    p1, st, err = cv2.calcOpticalFlowPyrLK(old_gray, frame_gray, p0, None, **lk_params)
                    if p1 is not None:
                        good_new = p1[st==1]
                        good_old = p0[st==1]
                    for i, (new, old) in enumerate(zip(good_new, good_old)):
                        a, b = new.ravel()
                        c, d = old.ravel()
                        mask = cv2.line(mask, (a, b), (c, d), [255,255,255], 2)
                        frame = cv2.circle(frame, (a, b), 5, [255,255,255], -1)
                    img = cv2.add(frame, mask)
                    cv2.imshow('Image', img)
                    old_gray = frame_gray.copy()
                    p0 = good_new.reshape(-1, 1, 2)
                    if cv2.waitKey(1) == ord('q'):
                        break
                    else:
                        pass
                else:
                    print("Error")
                    break
            cv2.destroyAllWindows()
            cap.release()

In [6]: lucas_kanade(img_path,lk_params)
```

### Dense Optical Flow in OpenCV

calcOpticalFlowFarneback(prev, next, flow, pyr_scale, levels, winsize, iterations, poly_n, poly_sigma, flags) → flow

This function computes a dense optical flow using the Gunnar Farneback's algorithm.

Here are the parameters for the function and what they represent:

- prev first 8-bit single-channel input image.
- next second input image of the same size and the same type as prev.
- flow computed flow image that has the same size as prev and type CV_32FC2.
- pyr_scale specifies the image scale (<1) to build pyramids for each image; pyr_scale = 0.5 means a classical pyramid, where each next layer is twice smaller than the previous one.
- levels number of pyramid layers including the initial image; levels = 1 means that no extra layers are created and only the original images are used.
- winsize averaging window size.
- iterations number of iterations the algorithm iterates at its image.
- poly_n size of the pixel neighborhood used to find polynomial expansion in each pixel.
- poly_sigma standard deviation of the Gaussian that is used to smooth derivatives used as a basis for the polynomial expansion. For poly_n=5, you can set poly_sigma=1.1. For poly_n=7, a good value would be poly_sigma=1.5.
- flags operation flags that can be a combination of the following:
  - OPTFLOW_USE_INITIAL_FLOW uses the initial estimates stored in flow as the first approximation. For poly_n=5, you can set poly_sigma=1.1. For poly_n=7, a good value would be poly_sigma=1.5.
  - OPTFLOW_FARNEBACK_GAUSSIAN uses the Gaussian winsize x winsize filter instead of a box filter of the same size for optimal quality but more slow.

```
In [7]: def dense_flow(path):
            cap = cv2.VideoCapture(path)
            ret, frame1 = cap.read()
            prvs = cv2.cvtColor(frame1,cv2.COLOR_BGR2GRAY)
            hsv = np.zeros(frame1.shape)
            hsv[...,1] = 255
            while True:
                ret, frame2 = cap.read()
                if ret:
                    next = cv2.cvtColor(frame2,cv2.COLOR_BGR2GRAY)
                    flow = cv2.calcOpticalFlowFarneback(prvs, next, None, 0.5, 3, 15, 3, 5, 1.2, 0)
                    mag, ang = cv2.cartesian(flow[...,0], flow[...,1])
                    hsv[...,0] = ang*180/np.pi/2
                    hsv[...,2] = cv2.normalize(mag,None,0,255,cv2.NORM_MINMAX)
                    rgb = cv2.cvtColor(hsv,cv2.COLOR_BGR2BGR)
                    cv2.imshow('Image', rgb)
                    prvs = next
                else:
                    print("Error")
                    break
            cv2.destroyAllWindows()
            cap.release()

In [8]: dense_flow(img_path)
```

### BackgroundSubtractorMOG2

Parameters:

- history
  The number of last frames.
- varThreshold
  It is a threshold on the squared distance between the pixel and the sample to decide whether a pixel is close to that sample. This parameter does not affect background update.
- detectShadows
  If true, the algorithm will detect shadows and mark them. It decreases the speed a bit, so if you do not want this feature, set the parameter to false.

```
In [9]: def backgroundsub_mog(path):
            cap = cv2.VideoCapture(path)
            fgbg = cv2.createBackgroundSubtractorMOG2(history = 500, varThreshold = 16, detectShadows = True)
            while True:
                ret, frame = cap.read()
                mask = fgbg.apply(frame)
                if ret:
                    cv2.imshow('Image', mask)
                    else:
                        print("Error")
                        break
                if cv2.waitKey(1) == ord('q'):
                    break
                else:
                    pass
            cv2.destroyAllWindows()
            cap.release()

In [10]: backgroundsub_mog(path_to_vid)
```

### Creates mixture-of-gaussian background subtractor

Parameters:

- history
  Number of last frames.
- nmixtures
  Number of Gaussian mixtures.
- backgroundRatio
  Background ratio.
- noise
  Noise strength (standard deviation of the brightness or each color channel). 0 means some automatic value.

```
In [11]: def backgroundsub_mog(path):
             cap = cv2.VideoCapture(path)
             fgbg = cv2.bgsegm.createBackgroundSubtractorMOG(history = 500, nmixtures = 5, backgroundRatio = 0.7, noise=0)
             while True:
                 ret, frame = cap.read()
                 mask = fgbg.apply(frame)
                 if ret:
                     cv2.imshow('Image', mask)
                     else:
                         print("Error")
                         break
                 if cv2.waitKey(1) == ord('q'):
                     break
                 else:
                     pass
             cv2.destroyAllWindows()
             cap.release()

In [12]: backgroundsub_mog(path_to_vid)
```

### Creates a GMG Background Subtractor

Parameters:

initializationFrames - number of frames used to initialize the background models.

decisionThreshold - Threshold value, above which it is marked foreground, else background.

- updateBackgroundModel
  Total number of frames before background model is updated.
- smoothingRadius
  Radius for morphological filtering.

```
In [13]: def backgroundsub_gmg(path):
             cap = cv2.VideoCapture(path)
             fgbg = cv2.bgsegm.createBackgroundSubtractorGMG(initializationFrames = 120, decisionThreshold = 0.8)
             while True:
                 ret, frame = cap.read()
                 mask = fgbg.apply(frame)
                 if ret:
                     cv2.imshow('Image', mask)
                     else:
                         print("Error")
                         break
                 if cv2.waitKey(1) == ord('q'):
                     break
                 else:
                     pass
             cv2.destroyAllWindows()
             cap.release()

In [14]: backgroundsub_gmg(path_to_vid)
```

## Background Subtraction

We can assume most of the time, some pixel have the same piece of the background because the camera is not moving. Occasionally, a car or other moving object enters the frame and occupies a portion of the frame.

For a video sequence, we can randomly select a few frames = 100

In other words, for every pixel, we can look at the previous 100 frames and take the median value. As long as a pixel is not covered by a car or other moving object (look back 100 frames at this time), the median of the last 100 frames will give a good estimate of the background at that pixel.

```
In [15]: def backgroundsub_mog(path):
             cap = cv2.VideoCapture(path)
             fgbg = cv2.bgsegm.createBackgroundSubtractorMOG()
             while True:
                 ret, frame = cap.read()
                 mask = fgbg.apply(frame)
                 if ret:
                     cv2.imshow('Image', mask)
                     else:
                         print("Error")
                         break
                 if cv2.waitKey(1) == ord('q'):
                     break
                 else:
                     pass
             cv2.destroyAllWindows()
             cap.release()

In [16]: backgroundsub_mog(path_to_vid)
```

## Bitwise Operations

This includes the bitwise AND, OR, NOT, and XOR operations. They will highly useful while exploring any part of the image.

Bitwise operations are used in image manipulation and in the creation of masks. Masks are used to hide or show parts of the image that are of no interest. Masks are used to hide or show parts of the image that are of no interest. For example, if we are only interested in the face in an image, we can use a mask to hide everything else in the image. Bitwise operations are also used in the creation of masks. For example, if we are only interested in the face in an image, we can use a mask to hide everything else in the image. Bitwise operations are also used in the creation of masks. For example, if we are only interested in the face in an image, we can use a mask to hide everything else in the image.

```
In [17]: def backgroundsub_mog(path):
             cap = cv2.VideoCapture(path)
             fgbg = cv2.bgsegm.createBackgroundSubtractorMOG(history = 500, nmixtures = 5, backgroundRatio = 0.7, noise=0)
             while True:
                 ret, frame = cap.read()
                 mask = fgbg.apply(frame)
                 if ret:
                     cv2.imshow('Image', mask)
                     else:
                         print("Error")
                         break
                 if cv2.waitKey(1) == ord('q'):
                     break
                 else:
                     pass
             cv2.destroyAllWindows()
             cap.release()

In [18]: backgroundsub_mog(path_to_vid)
```

```
In [19]: def backgroundsub_gmg(path):
             cap = cv2.VideoCapture(path)
             fgbg = cv2.bgsegm.createBackgroundSubtractorGMG(initializationFrames = 120, decisionThreshold = 0.8)
             while True:
                 ret, frame = cap.read()
                 mask = fgbg.apply(frame)
                 if ret:
                     cv2.imshow('Image', mask)
                     else:
                         print("Error")
                         break
                 if cv2.waitKey(1) == ord('q'):
                     break
                 else:
                     pass
             cv2.destroyAllWindows()
             cap.release()

In [20]: backgroundsub_gmg(path_to_vid)
```