Python has been updated since this video was made, please download the 3.7 version.

In this lesson you will learn how to setup the development environment.

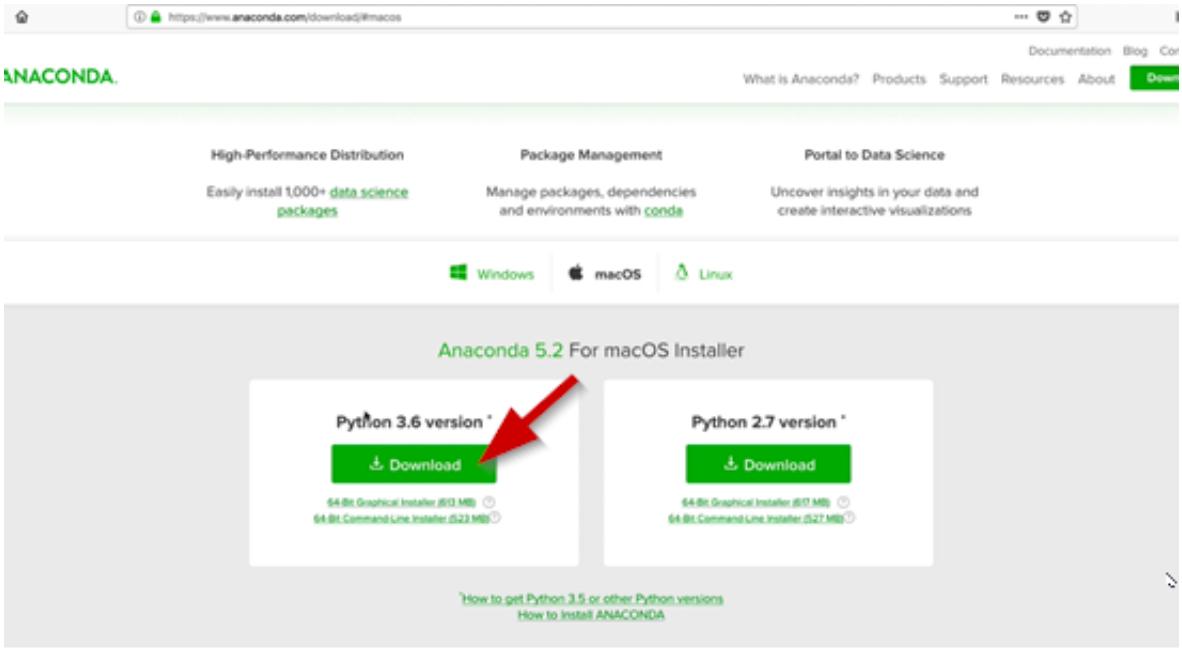
For managing our dependencies we are going to use this tool called **“Anaconda.”** Essentially you can use Anaconda to manage our dependencies and update them very easily.

You can download Anaconda from here: https://www.anaconda.com/

The direct link is here: Anaconda Download

Choose the version of Anaconda you need based on the current operating system you are using.

Please download the latest version of Python.

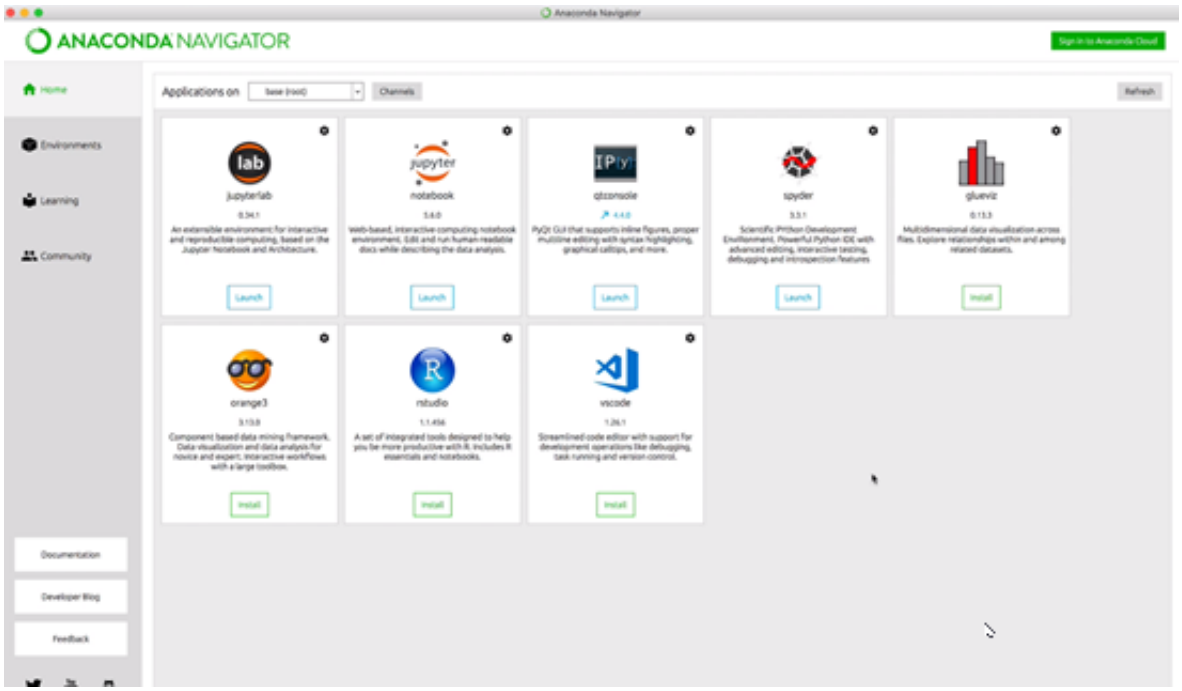


Get Started

Once you have Anaconda downloaded go ahead and follow the setup wizard’s instructions.

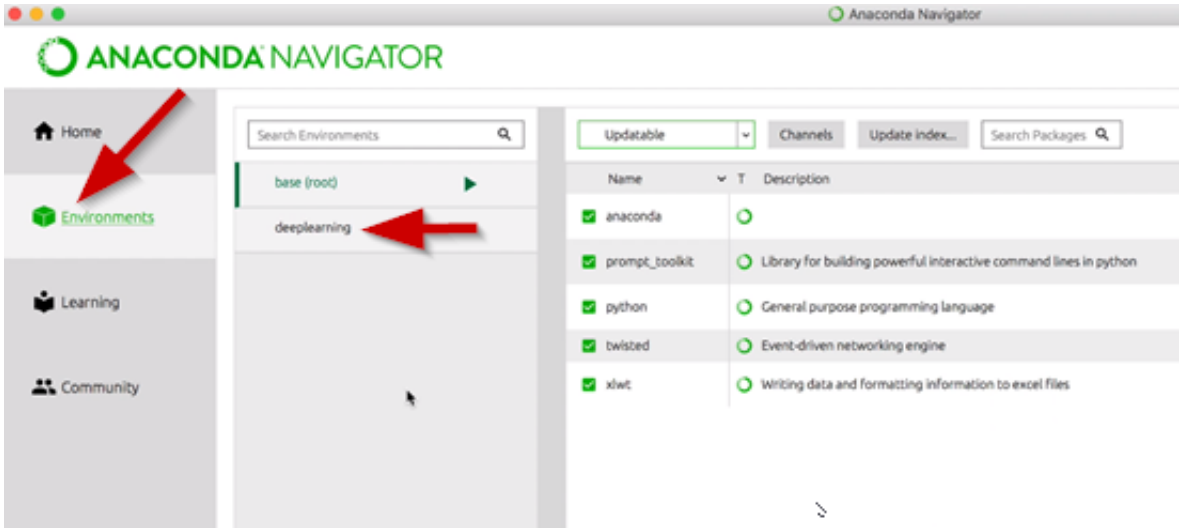
Once Anaconda is installed go ahead and open the Anaconda Navigator application.

Once Anaconda is opened on your system it should look similar to this:

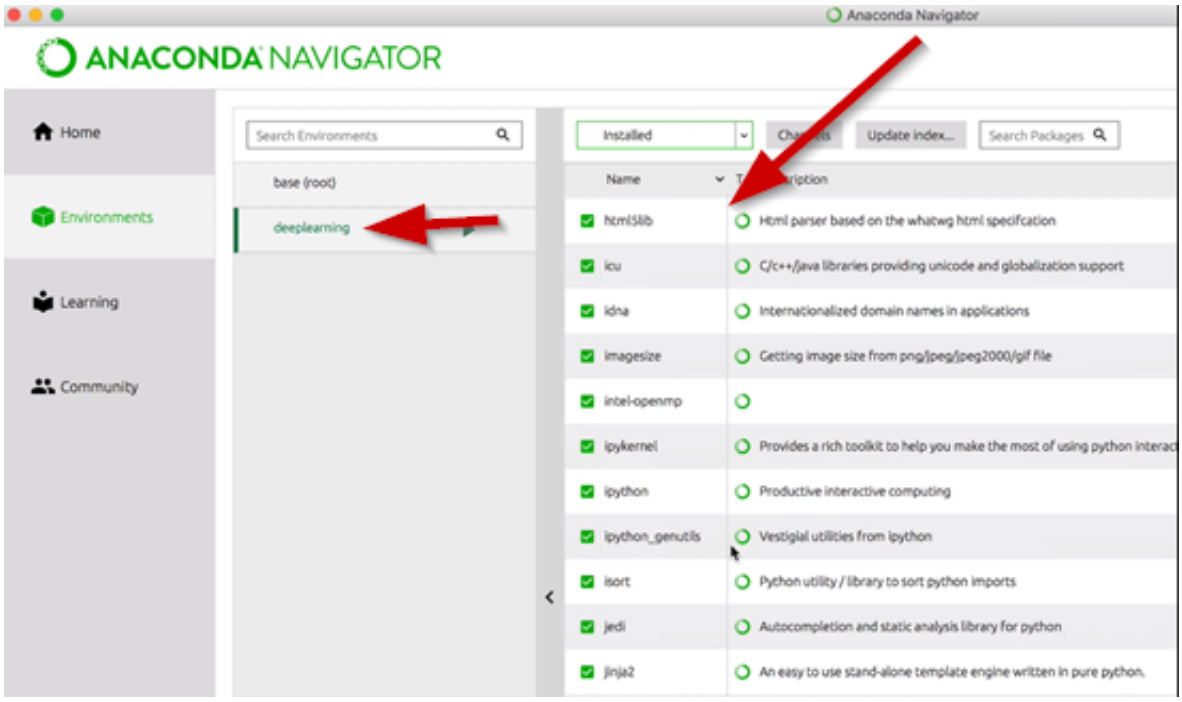


This already has all our dependencies bundled into these things called environments.

So **select the Environments Tab from the Anaconda menu.**



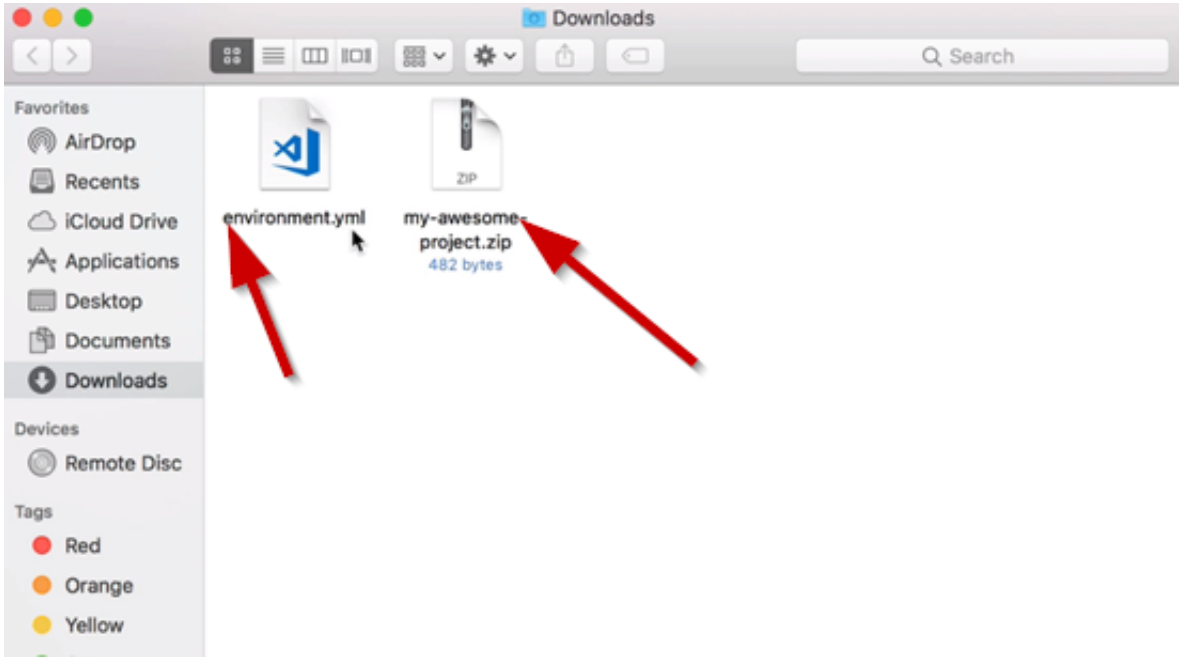
You can change your environment by clicking on one of them in the menu. So if you clicked on the deeplearning one, you will then see all of the installed packages listed on the right.



In the projects that we will be working on for this course, you will be provided with an environment file. All you will need to do is import the environment file that was provided and then it will create the environment, download all the dependencies packages and install them.

You can download the environment file from the Course Home page.

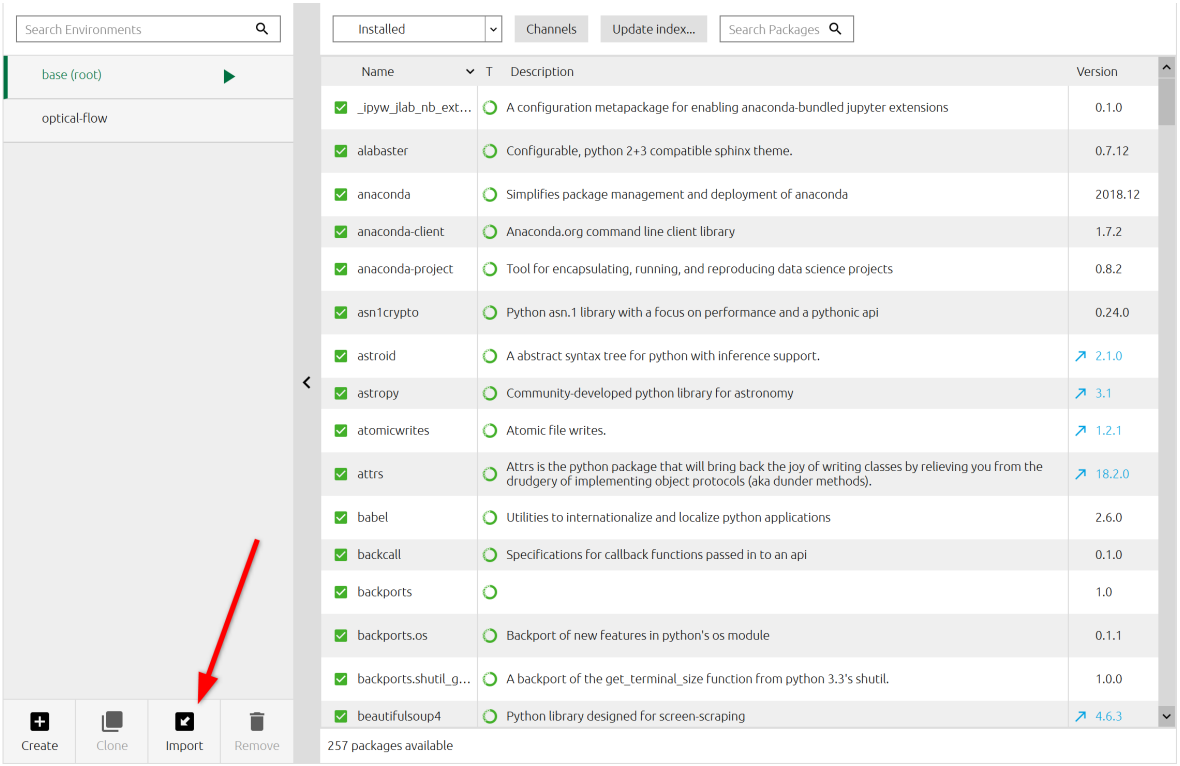
The **file name will be related to the course** and once you have it downloaded you will need to **extract it.** Once extracted **you will have one single file called “environment.yml”**



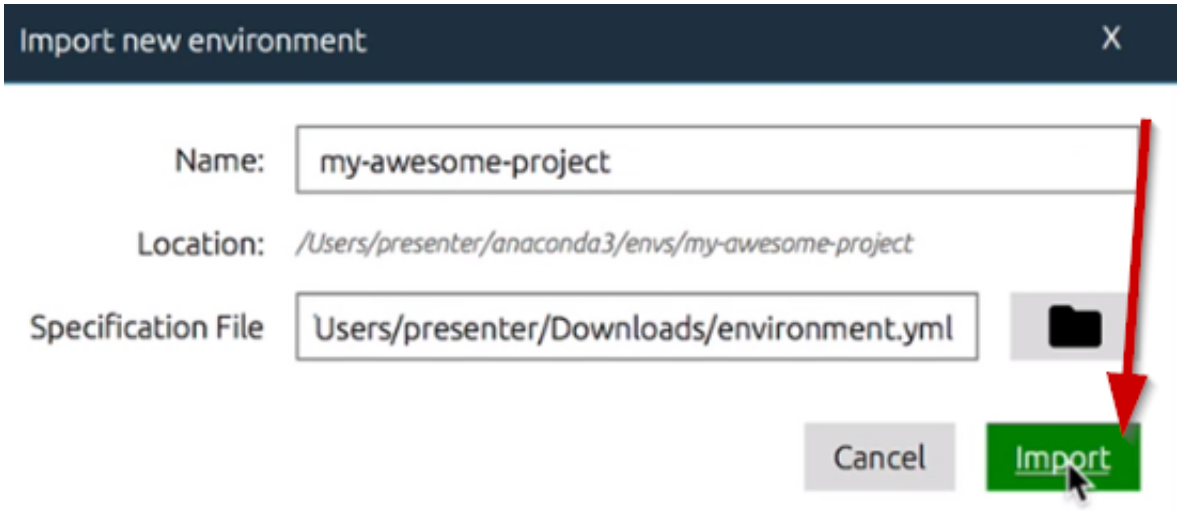
The **environment.yml file contains all of dependencies of the Python packages and the version numbers.**

All we need to do now is **import the environment file.** At the bottom of the Anaconda

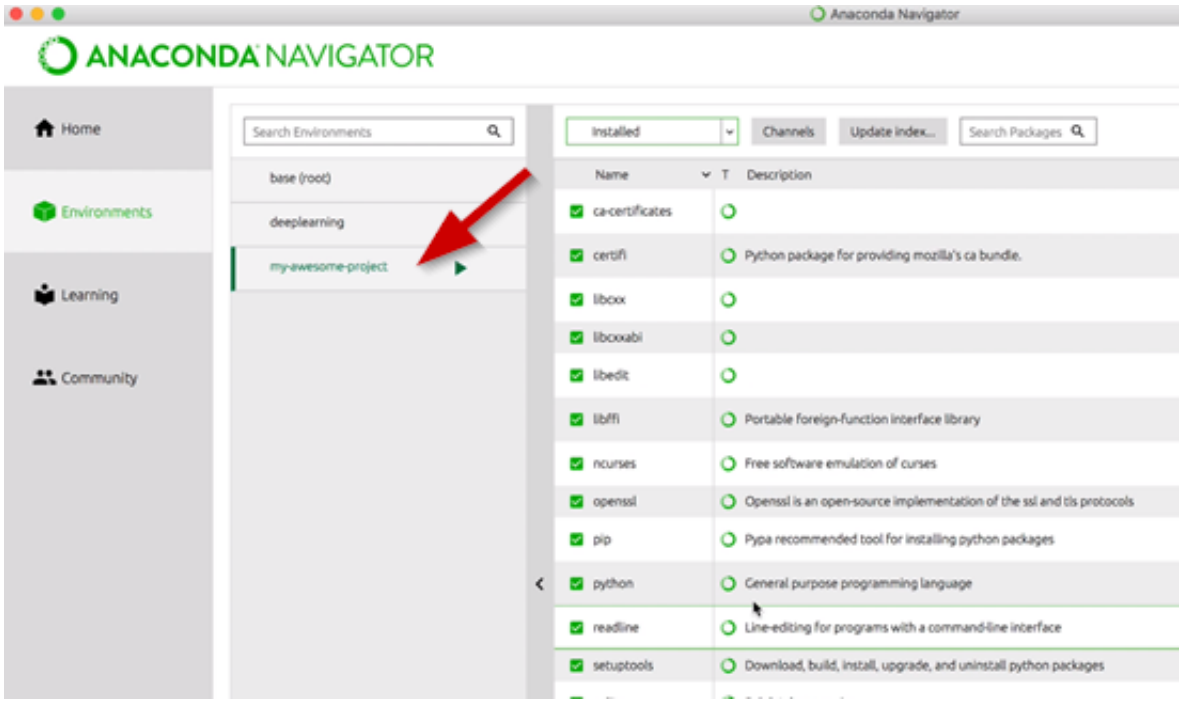
Environments tab you can see a toolbar to create and import environments. Select **import**.



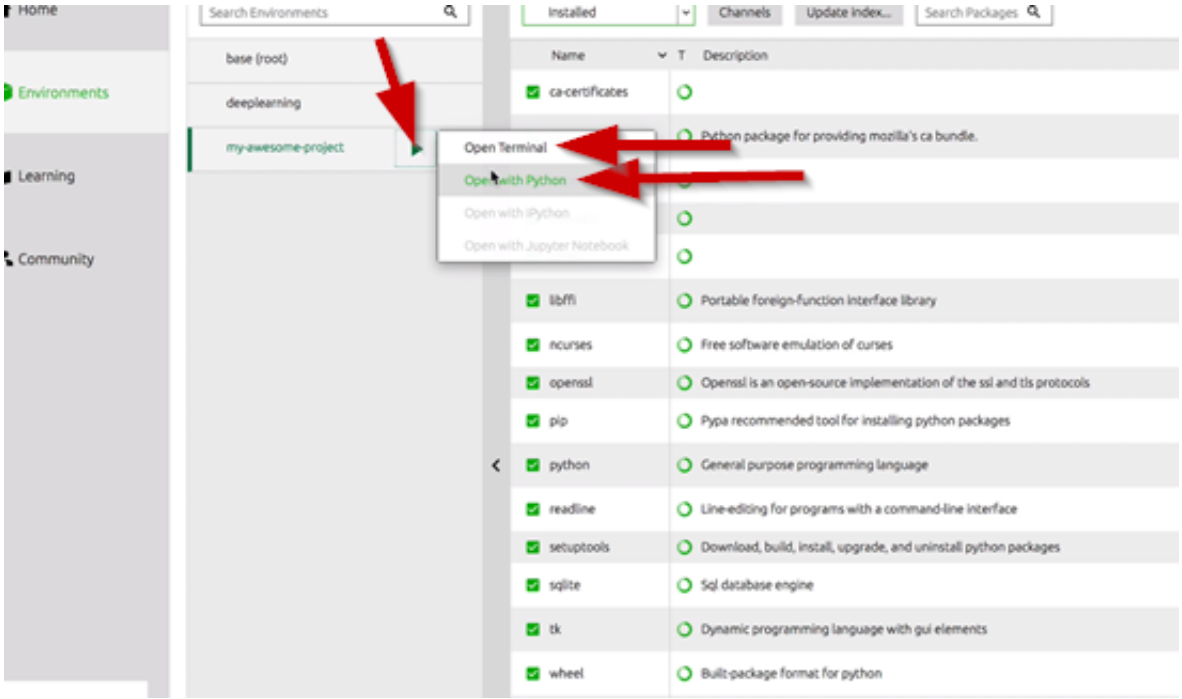
This will pull up a window for you to select the environment file you extracted. Name it and then click the import button.



After its done loading you will now see it in the list to choose:



If you **click on the green arrow next to the my-awesome-project you will open a drop down menu,** and from this menu you can choose to open with the terminal or with Python.



In this lesson, you will learn the basics of videos, and how function notation can be applied to find pixel intensities of videos.

Videos are a sequence of images (called **frames**), which allows image processing to handle videos.

The rate at which the images change is called the frames per second, and is known as the **FPS** of the video. A common example of this is 60 FPS, which means the video shows 60 frames every second.

For images, a function **I** can be applied to the image so that **I** (x,y) = **p**, where x and y are coordinates, and **p** is the pixel intensity.

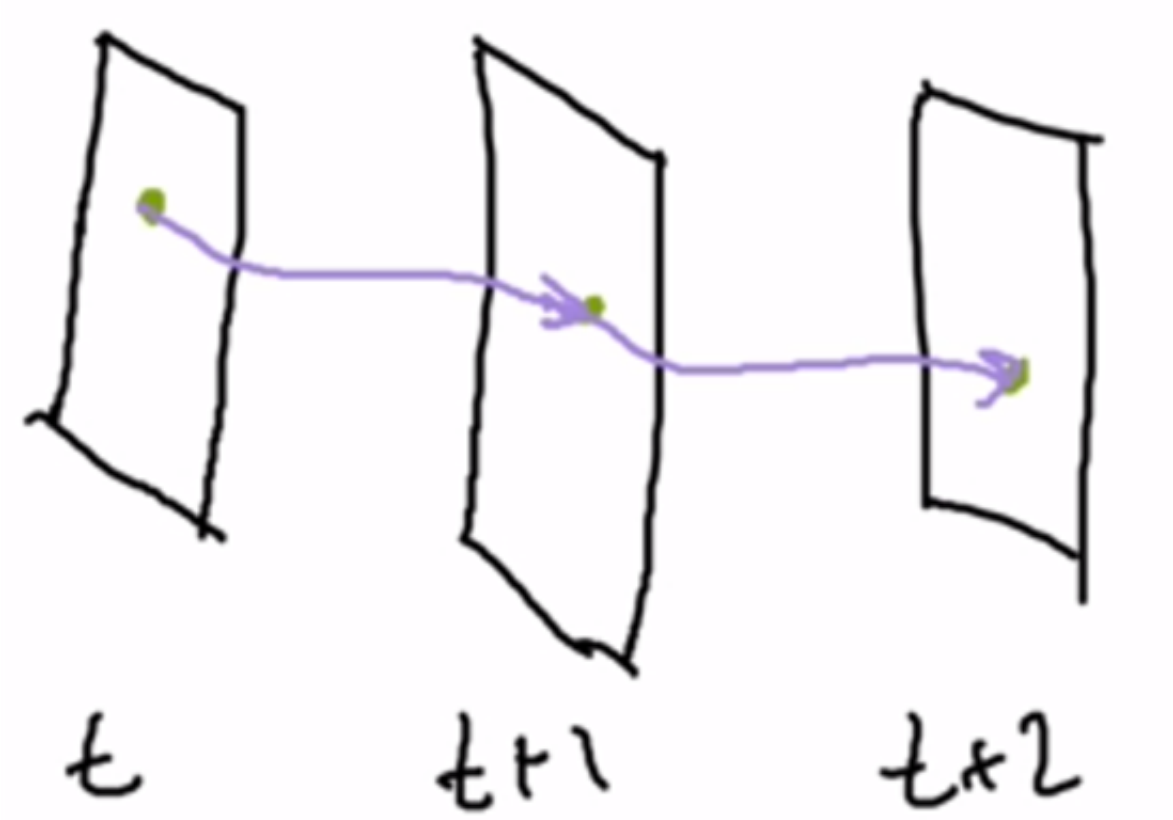
For videos, there needs to be additional information to find the pixel intensity, as there are numerous frames to choose from. The parameter **t** needs to be added, with **t** being when in the video the desired frame is located. Then, adding **t** to the image function, **I** (x,y,t) = **p** for videos.

In this lesson, you will learn about the basics of optical flow and the mathematics of it.

For images, the only information that you can access is the spatial positioning in relation to other pixels. The benefit of videos over static images is that it adds temporal information, so that you can know not only the location spatially, but also when it exists. This is what allows optical flow to function.

Optical flow is type of computer vision technique that is used to track the apparent motion of objects in a video. Optical flow can be used to track objects, stabilize and compress videos, and allow AI to generate descriptions of videos.

Optical flow functions by tracking a pixel through consecutive frames. This allows for the path of that pixel's movement to be generated, as shown in this image.

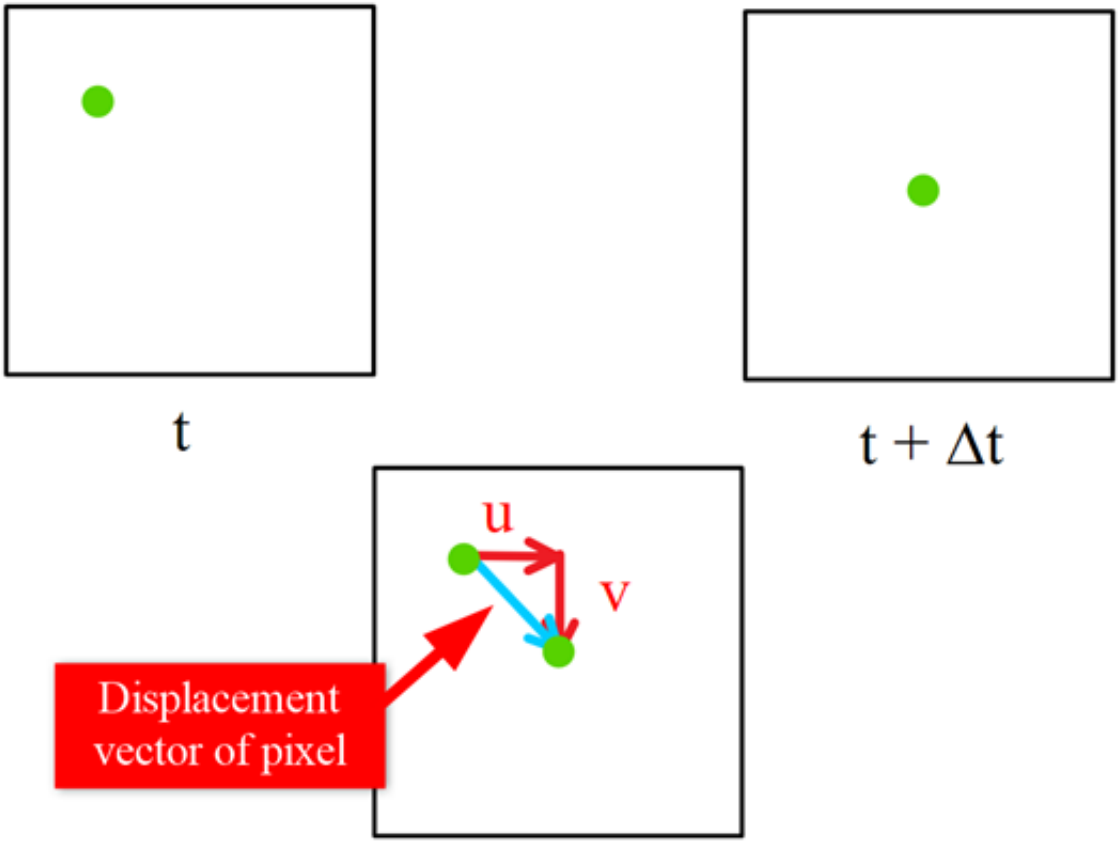


Optical flow makes two assumptions that drastically simplify this process. These are:

1. Pixel intensities don't rapidly change between consecutive frames.
2. Groups of pixels move together.

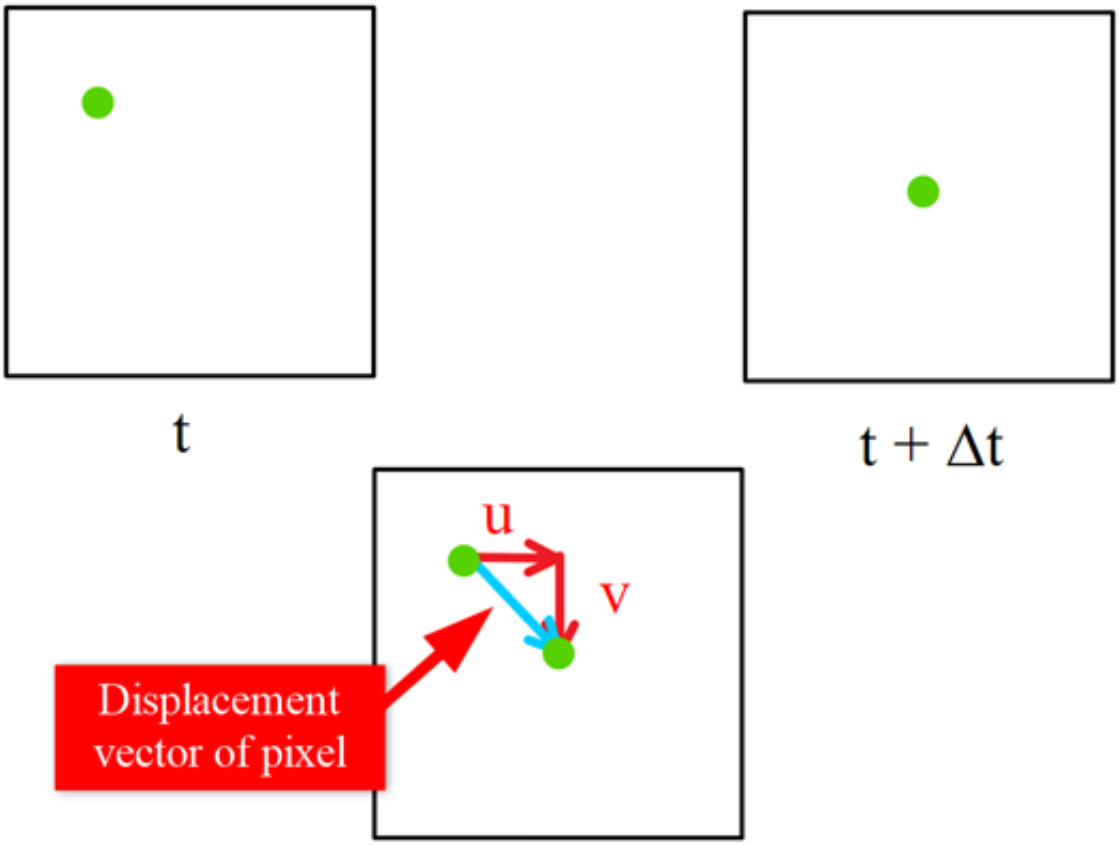
These two assumptions apply when a video is smooth, as pixels should change gradually and not teleport around the image. These assumptions apply for almost all real world videos, but can be broken if someone specifically edits a video to make that occur.

When analyzing videos, the pixel you are tracking will be displaced from its original position by some value u in the x direction and some value v in the y direction, as shown in the image below. The goal of optical flow is to find the u and v values, as they allow you to create a displacement vector and track the pixel's path.

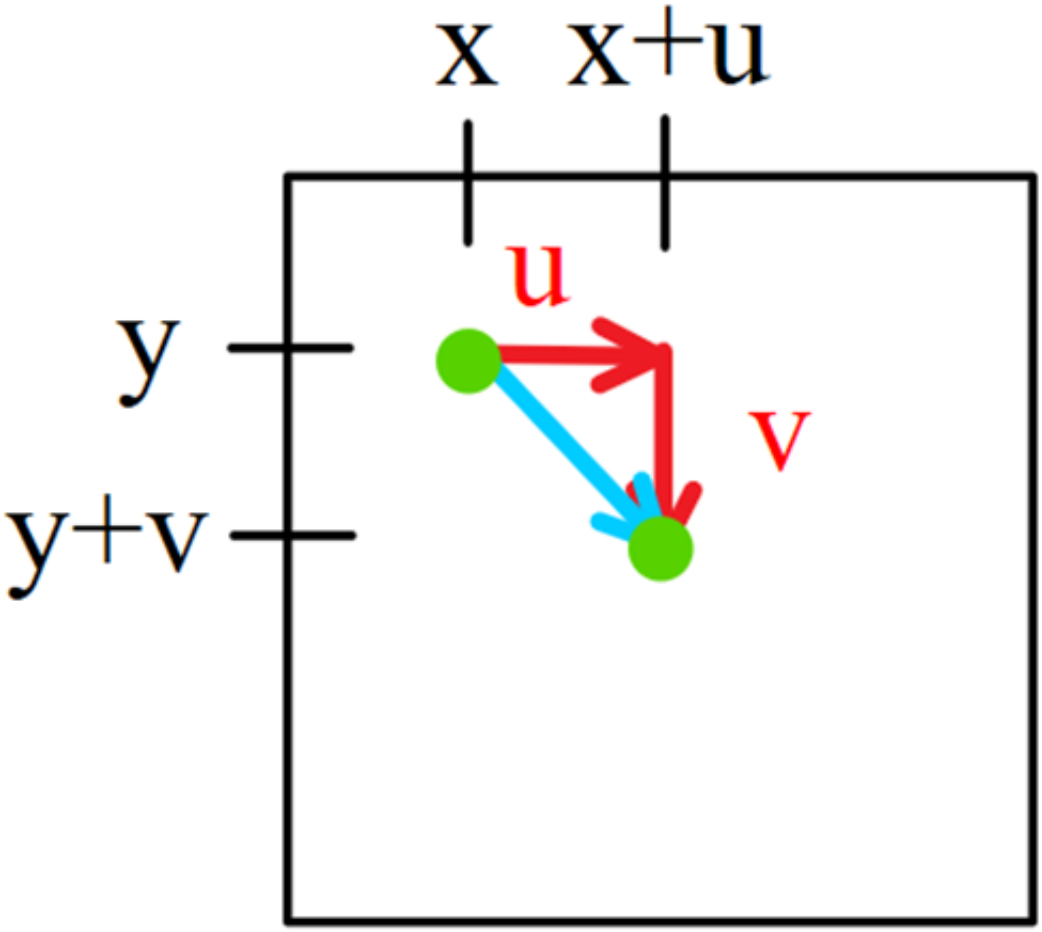


In this lesson, you will continue to learn about optical flow, with this lesson providing more information about the mathematics behind this technique.

To recap from the previous lesson, the point of optical flow is to find the displacement vector of a pixel, with u representing the change in the x direction and v representing the change in the y direction. In the image, t represents the time at which the frame occurs, with Δt representing how much time has passed since the previous frame.



Using the function notation from the earlier lessons, the pixel intensity for the frame at time t can be stated as I(x,y,t). The pixel intensity for the second frame can be stated as I(x+u, y+v, t+Δt).



These two functions should be equal, due to the first assumption of optical flow, meaning that I(x,y,t) = I(x+u, y+v, t+Δt).

After using some calculus, this equality turns in to the equation $I_x u + I_y v + I_t = 0$. I_x represents how much the frame changes horizontally, I_y represents how much the frame changes vertically, and I_t represents the difference in time between the frames.

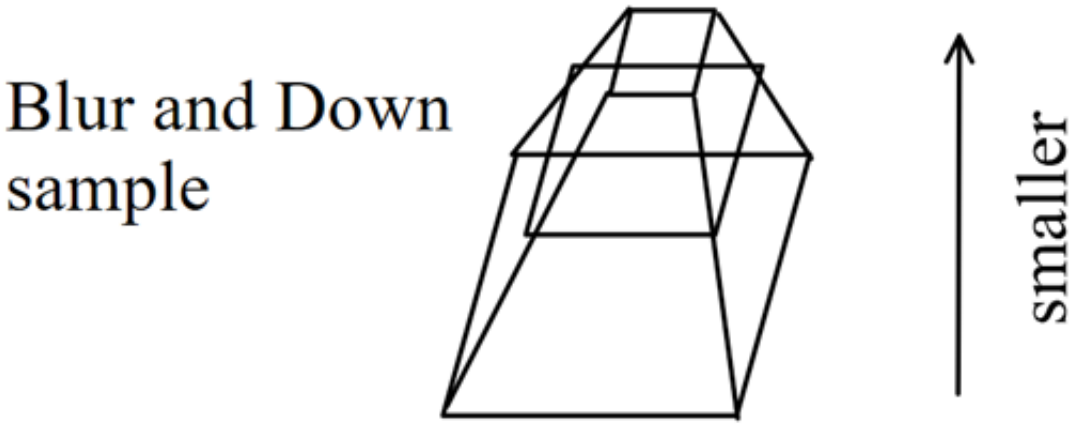
I_x, I_y, and I_t can all be computed, so the equation comes down to solving for u and v. There are numerous methods to solve this, but as they require calculus and linear algebra, this course will not be covering all of them. Certain techniques which will allow you to find u and v will be addressed in later lessons.

In this lesson, you will learn more about the second assumption of optical flow, as well as how pixel tracking is handled.

The Lucas-Kanade method can be used to approximate the u and v from the equation: $I_x u + I_y v + I_t = 0$. For more information about this method, please see this page on Wikipedia here: https://en.wikipedia.org/wiki/Lucas%E2%80%93Kanade_method. OpenCV will be able to run the calculations required for you.

The Second Assumption

Given the second assumption about using optical flow, that groups of pixels move together, a method known as image pyramids needs to be used to account for large u and v values. This is done by repeatedly blurring and downsampling the image to shrink the size of the image.



What this does is reduce the size of the pixel motions in the image. Large motions become small ones, and small motions disappear entirely. By using this method, the second assumption can always hold. This can be automatically done by OpenCV.

Pixel Tracking

It is important when running optical flow to not track every pixel in the image. This would dramatically increase the time it would take to run the computations.

One approach to remedy this issue is to find some good features in the image to track. An algorithm known as Shi-Tomasi can be used to do this. It functions by returning the strongest corners in the image, which can then be used for tracking and drawing the flow patterns. Corners are ideal for tracking, according to the mathematics behind the Lucas-Kanade method.

In this lesson, you will learn how to determine the pixel conversion rate of your camera.

One of the crucial pieces of information needed to operate with real world quantities is the conversion between pixels and cm, in, or any other unit of distance you would like to use. This conversion rate will be different for each camera, so you will need to follow the process outlined below to find your particular conversion rate.

1. Position your camera so that it is stable and will not move while you are collecting data.
2. Use a reference object that you can take measurements of. Flat objects will be easiest for this. If available, a ruler is ideal, but a coin or piece of paper works as well.
3. Align the reference object so that it is squared-up with the camera lens. This is to ensure that there is no tilt or skewed perspective.
4. Take a picture of your reference object.
5. In a photo editing program, take the same measurements of your reference object, this time measuring in pixels.
6. Take the pixel measurement from step 5, and divide by the corresponding measurement from step 2. This will give you the conversion rate for your camera, in pixels per unit of distance. To get the inverse, divide 1 by the conversion rate, or swap the measurements in the division.

This process will allow you to calculate real-world distances from pixels in an image. However, it will only be accurate if there is no tilt in the camera, and may need to be repeated if your camera positioning changes significantly.

In this lesson, you will learn about three different applications of optical flow.

Video Compression

Optical flow can be utilized for compressing videos, as the methods for tracking changes can allow a video to be thought of as a “running sum” of changes to the initial image. Each successive image isn’t a new image, it is a displacement that you add to the current frame. When there are fewer changes from one frame to the next, this “running sum” method means that the displacement matrix is mostly empty. As there are very efficient ways to work with such matrices, you can compress a video down so that it uses less file space.

Video Stabilization

In recent years, there has been research done that uses flow to stabilize jittery video. One particular process from a 2014 research paper is called steady flow. This method tracks pixels in the image, and smoothes out the sharp, jittery pixel movements, resulting in a much more stable video. Optical flow is a very commonly used technique for video stabilization.

Video Descriptions

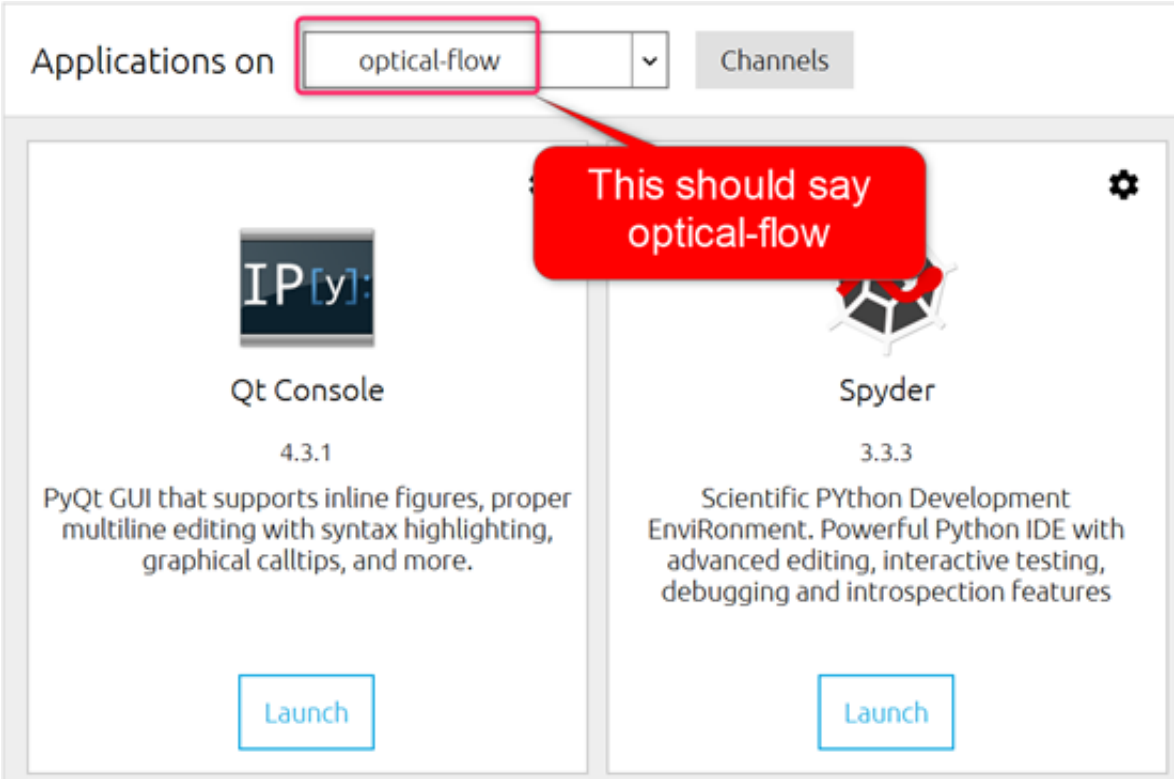
This is a very advanced usage of optical flow, and was created very recently. Initially, this was done with static images. An AI would be given an image, and it would generate a short description of the image. Researchers are currently still working out how to combine optical flow with these techniques to generate a better description for videos.

There are many more applications of optical flow, these three are some of the most popular and illustrate some of the alternative usages of the technique. Optical flow is incredibly versatile, and you will be able to use the techniques you learn in this course to do a lot of different things.

In this lesson, you will learn how to work with video files using Python.

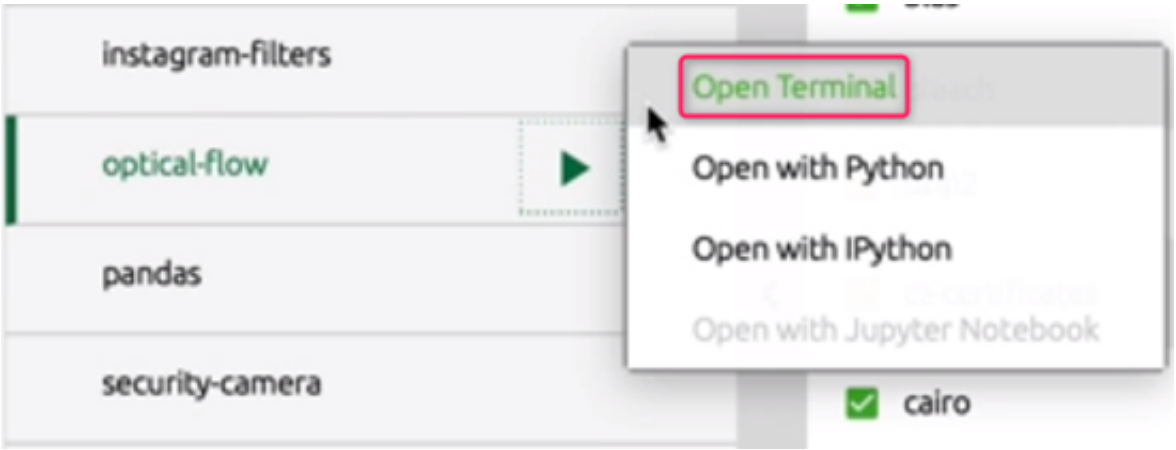
The first step is to download the source code from the course files, which you can find by clicking on the “Course Home” button under the video. Make sure to unzip the folder so that files can be saved to it.

Next, you will go into the Anaconda Navigator. You’ll want to confirm that you are using the correct environment.



If you’re not using the correct environment, please review the Development Environment – Anaconda lesson.

You will then go to the environment tab, and open the optical-flow environment in a terminal.



All of the Python files you will be using will need to be opened this way, as OpenCV does not interact well with the Spyder UI. Here are some helpful commands for navigating in the terminal:

- **pwd / cd** : This will tell you what directory you currently have open in the terminal, which is known as your present working directory. **pwd** is for Mac and Linux, **cd** is for Windows.
- **cd x : This will change your present working directory to the new directory you input.**
- **cd ..** : **This will cause you to move up one directory level. For example, if you are in C:\Users and use this command, your present working directory will now be C:**
- **ls / dir : This command will show you a list of all files that are in your current directory. ls is for Mac and Linux, dir is for Windows.**

You will then want to launch Spyder, and save the file to the same directory as your video. If you have downloaded the source code, make sure to save the file with a unique name so that you can ensure you're working with the correct file.

The python code will have two main sections, one for loading in the video, and another for running the UI. It is helpful to create comments for these sections. You will want to import both **cv2** and **numpy** as shown in the code snippet below.

```
import cv2
import numpy as np

# load video


# main UI loop
```

To load in the video, you will use a class from **cv2** called VideoCapture, which takes a filename.

```
# load video
video_cap = cv2.VideoCapture('test.mov')
```

In the main UI loop, you will make an infinite while loop. This loop will handle reading in the frames of the video, displaying them, and allow you to quit the video display.

```
# main UI loop
while True:
    # read in a video frame
    _, frame = video_cap.read()
    if frame is None:
        break

    cv2.imshow('frame', frame)
    if cv2.waitKey(10) == ord('q'):
        break
```

When the frame object becomes empty because the video has ended, the if statement will become true and the while loop will break.

This section of the code is crucial.

```
    cv2.imshow('frame', frame)
    if cv2.waitKey(10) == ord('q'):
        break
```

This if statement means that the video will display the image for 10 milliseconds, while checking if the q key has been pressed. If it has, it will break out of the infinite while loop. If not, it continues to display frames. waitKey() is frequently used in OpenCV to process GUI events, and allows the frames to actually display. If you don't include the waitKey(), the video will not display.

You will need to include the follow code at the end of your script to ensure that all of the resources used will be unallocated properly.

```
# clean up resources
cv2.destroyAllWindows()
video_cap.release()
```

The full script is included in the following snippet.

```
# -*- coding: utf-8 -*-
import cv2
import numpy as np

# load video
video_cap = cv2.VideoCapture('test.mov')

# main UI loop
while True:
    # read in a video frame
    _, frame = video_cap.read()
    if frame is None:
        break

    cv2.imshow('frame', frame)
    if cv2.waitKey(10) == ord('q'):
        break

# clean up resources
cv2.destroyAllWindows()
video_cap.release()
```

Now you can run the Python script. You will open up the terminal to the directory where you saved the file, and type in **python** and then the name of your script file. This will display the video, which you should be able to exit out of early by pressing the 'q' key.

In this lesson, you will learn how to get and display the features in your image that the app will be tracking.

Optical flow needs strong features to track throughout the video frames. OpenCV has a function that will get these for you, by giving you corners in the video.

You will need to comment out the **# main UI loop** and the **# clean up resources** sections of your code. This will allow run the function on just the first frame of the video, so that you can see the features it will be tracking.

To read in the first frame, add the following code to your **# load video section**.

```
# load video
video_cap = cv2.VideoCapture('test.mov')
_, frame = video_cap.read()
```

Now you can write the code to generate the points that will be tracked. See the code below:

```
# feature parameters
feature_params = {
        'maxCorners': 100,
        'qualityLevel': 0.3,
        'minDistance': 7
}

# load video
video_cap = cv2.VideoCapture('test.mov')
_, frame = video_cap.read()

# convert first frame to grayscale and pick points to track
old_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)    #sets the first frame to graysc
ale
prev_pts = cv2.goodFeaturesToTrack(image=old_gray, **feature_params)
```

As optical flow compares changes from one frame to the next, you will need to have variables to keep track of the information from the previous frame. These will be more important in later videos when you will be dealing with the full video.

The ****feature_params** in the **goodFeaturesToTrack** function is a Python shortcut that will allow you to have the parameters for that function at the top of the script, making testing different values easier. To find out more about the parameters of **goodFeaturesToTrack**, check the documentation here:
https://docs.opencv.org/3.0-beta/modules/imgproc/doc/feature_detection.html?highlight=goodfeaturestotrack#cv2.goodFeaturesToTrack

The next step is to draw the points returned by the **goodFeaturesToTrack** function. See the code below:

```
for pt in prev_pts:
    x, y = pt.ravel()
    cv2.circle(frame, (x, y), 5, (0,255,0), -1)
```



```
cv2.imshow('features', frame)
cv2.waitKey(0)
```

What this code does is draw a circle on the frame at the given coordinates, and fills it with the color specified. Feel free to play with these values to change how the points appear on your image. **cv2.imshow** and **cv2.waitKey** are needed to display the image.

The full code for this lesson is shown below:

```
# -*- coding: utf-8 -*-
import cv2
import numpy as np

# feature parameters
feature_params = {
        'maxCorners': 100,
        'qualityLevel': 0.3,
        'minDistance': 7
}

# load video
video_cap = cv2.VideoCapture('test.mov')
_, frame = video_cap.read()

# convert first frame to grayscale and pick points to track
old_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
prev_pts = cv2.goodFeaturesToTrack(image=old_gray, **feature_params)

for pt in prev_pts:
    x, y = pt.ravel()
    cv2.circle(frame, (x, y), 5, (0,255,0), -1)
cv2.imshow('features', frame)
cv2.waitKey(0)

"""
# main UI loop
while True:
    # read in a video frame
    _, frame = video_cap.read()
    if frame is None:
        break

    cv2.imshow('frame', frame)
    if cv2.waitKey(10) == ord('q'):
        break

# clean up resources
cv2.destroyAllWindows()
video_cap.release()
"""
```


Now, using the terminal, you can run your program. You should see the following image.

In this lesson, you will learn how to compute optical flow and how to create a mask which you will use to display the lines that will show where the tracked features move.

You will need to comment out the **for** loop from last lesson, as that was only for displaying the features on the first frame. You will also need to uncomment out the **# main ui loop** so that you can use that section again. See the code below:

```
"""
for pt in prev_pts:
    x, y = pt.ravel()
    cv2.circle(frame, (x, y), 5, (0,255,0), -1)
cv2.imshow('features', frame)
cv2.waitKey(0)
"""

# main UI loop
while True:
    # read in a video frame
    _, frame = video_cap.read()
    if frame is None:
        break

    cv2.imshow('frame', frame)
    if cv2.waitKey(10) == ord('q'):
        break

# clean up resources
cv2.destroyAllWindows()
video_cap.release()
```

Setting up the mask

To track the lines, you will need a **mask**, which is essentially separate image that OpenCV will overlay on the video frames. This **mask** will be where the lines are drawn. To prevent the **mask** from being overly cluttered, you will need to refresh at regular intervals. To make it easier to adjust this refresh rate, you should make an **# app parameters** section below the **# feature parameters**, as well as a **frame_counter**. See the code below:

```
# app parameters
REFRESH_RATE = 20    # New code

# load video
video_cap = cv2.VideoCapture('test.mov')
_, frame = video_cap.read()
frame_counter = 1    # New code

# convert first frame to grayscale and pick points to track
old_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
prev_pts = cv2.goodFeaturesToTrack(image=old_gray, **feature_params)

# show features on first frame
"""
for pt in prev_pts:
```



```
    x, y = pt.ravel()
    cv2.circle(frame, (x, y), 5, (0,255,0), -1)
cv2.imshow('features', frame)
cv2.waitKey(0)
"""

# create a mask
mask = np.zeros_like(frame)    # New code
```

Now you can add code to your **# main ui loop** to handle the mask refresh. See the code below

```
# main UI loop
while True:
    # reset the lines    # New code
    if frame_counter % REFRESH_RATE == 0:
        mask.fill(0)

    # read in a video frame
    _, frame = video_cap.read()
    if frame is None:
        break

    cv2.imshow('frame', frame)

    # update for next frame
    frame_counter += 1    # New code
    if cv2.waitKey(10) == ord('q'):
        break
```

The first if statement will reset the **mask** when the **frame_counter** divided by the **REFRESH_RATE** parameter that you set does not have a remainder. The **while** loop will also increment the **frame_counter** after displaying each frame of the video.

Optical flow calculation

Now you can add the code to calculate the optical flow. See below:

```
# main UI loop
while True:
    # reset the lines
    if frame_counter % REFRESH_RATE == 0:
        mask.fill(0)

    # read in a video frame
    _, frame = video_cap.read()
    if frame is None:
        break
    frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)    # New code

    # compute optical flow points    # New code
    next_pts, statuses, _ = cv2.calcOpticalFlowPyrLK(prevImg=old_gray, nextImg=frame_
```



```
gray, prevPts=prev_pts, nextPts=None)

    cv2.imshow('frame', frame)

    # update for next frame
    frame_counter += 1
    if cv2.waitKey(10) == ord('q'):
        break
```

As you have already converted the first frame to grayscale and found the points to track, you can use that information in the **cv2.calcOpticalFlowPyrLK** function, as shown in the code snippet. This function will return the location of the previous points in the current frame, along with **statuses**, which will indicate if any points became invalid, and an error array that you will not be using. See the documentation here for more information: https://docs.opencv.org/3.0-beta/modules/video/doc/motion_analysis_and_object_tracking.html?highlight=calcopticalflowpyrlk#cv2.calcOpticalFlowPyrLK

You can use the **statuses** to keep the optical flow points that are still valid in the current frame. See the code below:

```
# main UI loop
while True:
    # reset the lines
    if frame_counter % REFRESH_RATE == 0:
        mask.fill(0)

    # read in a video frame
    _, frame = video_cap.read()
    if frame is None:
        break
    frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    # compute optical flow points
    next_pts, statuses, _ = cv2.calcOpticalFlowPyrLK(prevImg=old_gray, nextImg=frame_
gray, prevPts=prev_pts, nextPts=None)

    # only keep the optical flow points that are valid    # New code
    good_next_pts = next_pts[statuses == 1]
    good_old_pts = prev_pts[statuses == 1]

    cv2.imshow('frame', frame)

    # update for next frame
    frame_counter += 1
    if cv2.waitKey(10) == ord('q'):
        break
```

With these valid points, you will be able to draw the lines between them on the **mask**. This will be covered in the next lesson.

The full code for this lesson is shown below:

```
import cv2
import numpy as np

# feature parameters
feature_params = {
        'maxCorners': 100,
        'qualityLevel': 0.3,
        'minDistance': 7
}

# app parameters
REFRESH_RATE = 20

# load video
video_cap = cv2.VideoCapture('test.mov')
_, frame = video_cap.read()
frame_counter = 1

# convert first frame to grayscale and pick points to track
old_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
prev_pts = cv2.goodFeaturesToTrack(image=old_gray, **feature_params)

# show features on first frame
"""
for pt in prev_pts:
    x, y = pt.ravel()
    cv2.circle(frame, (x, y), 5, (0,255,0), -1)
cv2.imshow('features', frame)
cv2.waitKey(0)
"""

# create a mask
mask = np.zeros_like(frame)

# main UI loop
while True:
    # reset the lines
    if frame_counter % REFRESH_RATE == 0:
        mask.fill(0)

    # read in a video frame
    _, frame = video_cap.read()
    if frame is None:
        break
    frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    # compute optical flow points
    next_pts, statuses, _ = cv2.calcOpticalFlowPyrLK(prevImg=old_gray, nextImg=frame_
gray, prevPts=prev_pts, nextPts=None)

    # only keep the optical flow points that are valid
    good_next_pts = next_pts[statuses == 1]
    good_old_pts = prev_pts[statuses == 1]
```



```
    cv2.imshow('frame', frame)

    # update for next frame
    frame_counter += 1
    if cv2.waitKey(10) == ord('q'):
        break

# clean up resources
cv2.destroyAllWindows()
video_cap.release()
```

In this lesson, you will learn how to display the optical flow lines on your video.

To generate the line, you will first need to get location of the valid new points and the corresponding old points. This will be done in the **# main ui loop**. See the code below:

```
# draw optical flow lines
for good_next_pt, good_old_pt in zip(good_next_pts, good_old_pts):
    # get new and old points
    x, y = good_next_pt.ravel()
    r, s = good_old_pt.ravel()
```

You will then be able to use a **cv2** function called **line** to draw the lines on the **mask**. The function **line** takes a layer, a starting point, and an ending point, which you will have just created from the previous code segment. It also has a color and thickness parameter, which you can change to whatever you would like. See the code below:

```
# draw optical flow lines
for good_next_pt, good_old_pt in zip(good_next_pts, good_old_pts):
    # get new and old points
    x, y = good_next_pt.ravel()
    r, s = good_old_pt.ravel()

    # draw the optical flow line
    cv2.line(mask, (x, y), (r, s), (0,255,0), 2)    # New code
```

To better visualize the movement, you will draw the corner points. This will be drawn directly on the frame, so will not use the **mask**. See the code below:

```
# draw optical flow lines
for good_next_pt, good_old_pt in zip(good_next_pts, good_old_pts):
    # get new and old points
    x, y = good_next_pt.ravel()
    r, s = good_old_pt.ravel()

    # draw the optical flow line
    cv2.line(mask, (x, y), (r, s), (0,255,0), 2)

    # draw the coordinate of the corner points in this frame
    cv2.circle(frame, (x, y), 10, (255,255,0), -1)    # New code
```

The next step is to combine the **mask** with the current frame. OpenCV has a function specifically for this, called **add**, which you will use. Note that you need to change the **cv2.imshow** function to the newly generated **frame_final**. See the code below.

```
# combine mask with frame
frame_final = cv2.add(frame, mask)

cv2.imshow('frame', frame_final)
```


The final step is to update **old_gray** and **prev_pts** to be the current frame and points. You will use the function **copy** to make sure that you are storing a copy of the current frame in grayscale. To save some computation, you will use **good_next_pts**, though you will have to **reshape** it to prevent passing the bad points. See the code below:

```
# update for next frame
    old_gray = frame_gray.copy()
    prev_pts = good_next_pts.reshape(-1, 1, 2)
    frame_counter += 1
    if cv2.waitKey(1) == ord('q'):
        break
```

You should now be able to run the app through the terminal. The tracking will not be perfect, as the pencil in the video is moving a little too fast for the optical flow function to work. That is something to keep in mind when attempting to use optical flow for something. The less pixel difference there is from one frame to the next, the better optical flow will be able to function.

Here is the full code for this lesson.

```
# -*- coding: utf-8 -*-
import cv2
import numpy as np

# feature parameters
feature_params = {
        'maxCorners': 100,
        'qualityLevel': 0.3,
        'minDistance': 7
}

# app parameters
REFRESH_RATE = 20

# load video and read in the first frame
video_cap = cv2.VideoCapture('test.mov')
_, frame = video_cap.read()
frame_counter = 1

# convert first frame to grayscale and pick points to track
old_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
prev_pts = cv2.goodFeaturesToTrack(image=old_gray, **feature_params)

# show features on first frame
"""
for pt in prev_pts:
    x, y = pt.ravel()
    cv2.circle(frame, (x, y), 5, (0,255,0), -1)
cv2.imshow('features', frame)
cv2.waitKey(0)
"""

# create a mask for the lines
mask = np.zeros_like(frame)
```



```
# main UI loop
while True:
    # reset the lines
    if frame_counter % REFRESH_RATE == 0:
        mask.fill(0)

    # read in a video frame
    _, frame = video_cap.read()
    if frame is None:
        break
    frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

    # compute optical flow points
    next_pts, statuses, _ = cv2.calcOpticalFlowPyrLK(prevImg=old_gray, nextImg=frame_
gray, prevPts=prev_pts, nextPts=None)

    # only keep the optical flow points that are valid
    good_next_pts = next_pts[statuses == 1]
    good_old_pts = prev_pts[statuses == 1]

    # draw optical flow lines
    for good_next_pt, good_old_pt in zip(good_next_pts, good_old_pts):
        # get new and old points
        x, y = good_next_pt.ravel()
        r, s = good_old_pt.ravel()

        # draw the optical flow line
        cv2.line(mask, (x, y), (r, s), (0,255,0), 2)

        # draw the coordinate of the corner points in this frame
        cv2.circle(frame, (x, y), 5, (0,255,0), -1)

    # combine mask with frame
    frame_final = cv2.add(frame, mask)

    cv2.imshow('frame', frame_final)

    # update for next frame
    old_gray = frame_gray.copy()
    prev_pts = good_next_pts.reshape(-1, 1, 2)
    frame_counter += 1
    if cv2.waitKey(10) == ord('q'):
        break

# clean up resources
cv2.destroyAllWindows()
video_cap.release()
```


In this lesson, you will learn how to compute the velocity of tracked objects in your video using optical flow.

You will need to add some information about the video and the camera you are using, as well as a distance parameter. If you are using your own video, this would be information you gathered in lesson 1-5. See the code below for what to add for the video in the course notes:

```
# camera and video intrinsics
FPS = 30
PX_PER_CM = 370

# app parameters
REFRESH_RATE = 20
DISTANCE_THRESH = 20    # New code
```

This information needs to be included because it provides information about the movement of objects in the video. Optical flow works best when objects are far away from the camera, and move in a side-to-side or up and down fashion. If objects are moving towards or away from the camera, optical flow has a hard time accurately tracking the speed of the object, as the perspective would be changing. These kind of movements are called **planar objects**, which are objects that are far away enough that they appear to be on a flat surface or a plane.

In order to calculate the distance, you will need to define a new function **d2**, which will return the Euclidean distance between the points using the **numpy** function **linalg.norm**. This is the same as using the distance formula, but more efficient. In order for the **numpy** function to work, you will need to convert points to arrays using **np.array**. See the code below:

```
# compute Euclidean distance (distance formula)
def d2(p, q):
    return np.linalg.norm(np.array(p) - np.array(q))
```

Now inside of the **# draw optical flow lines** section, you will add code to compute and display the speed of the points. See the code below:

```
# draw speed if the distance criteria is met
distance = d2((x, y), (r, s))
if distance > DISTANCE_THRESH:
    # compute speed
    speed_str = str(distance / PX_PER_CM * FPS) + ' cm/s'
    print(speed_str)
```

This code checks if the distance between the two points is greater than the given threshold parameter, and if it is, uses the intrinsic properties of the video to calculate the speed of the point. For now, you will just print this to the console. In the next lesson, you will learn how to display this screen on the video frames.

You can confirm that your app is working correctly by running it in the terminal now.

Here is the full code for this lesson:

```
# -*- coding: utf-8 -*-
import cv2
import numpy as np

# feature parameters
feature_params = {
        'maxCorners': 100,
        'qualityLevel': 0.3,
        'minDistance': 7
}

# camera and video intrinsics
FPS = 30
PX_PER_CM = 370

# app parameters
REFRESH_RATE = 20
DISTANCE_THRESH = 20

# compute Euclidean distance (distance formula)
def d2(p, q):
    return np.linalg.norm(np.array(p) - np.array(q))

# load video and read in the first frame
video_cap = cv2.VideoCapture('test.mov')
_, frame = video_cap.read()
frame_counter = 1

# convert first frame to grayscale and pick points to track
old_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
prev_pts = cv2.goodFeaturesToTrack(image=old_gray, **feature_params)

# show features on first frame
"""
for pt in prev_pts:
    x, y = pt.ravel()
    cv2.circle(frame, (x, y), 5, (0,255,0), -1)
cv2.imshow('features', frame)
cv2.waitKey(0)
"""

# create a mask for the lines
mask = np.zeros_like(frame)

# main UI loop
while True:
    # reset the lines
    if frame_counter % REFRESH_RATE == 0:
        mask.fill(0)

    # read in a video frame
    _, frame = video_cap.read()
    if frame is None:
```

```
            break
        frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

        # compute optical flow points
        next_pts, statuses, _ = cv2.calcOpticalFlowPyrLK(prevImg=old_gray, nextImg=frame_
gray, prevPts=prev_pts, nextPts=None)

        # only keep the optical flow points that are valid
        good_next_pts = next_pts[statuses == 1]
        good_old_pts = prev_pts[statuses == 1]

        # draw optical flow lines
        for good_next_pt, good_old_pt in zip(good_next_pts, good_old_pts):
            # get new and old points
            x, y = good_next_pt.ravel()
            r, s = good_old_pt.ravel()

            # draw the optical flow line
            cv2.line(mask, (x, y), (r, s), (0,255,0), 2)

            # draw the coordinate of the corner points in this frame
            cv2.circle(frame, (x, y), 5, (0,255,0), -1)

            # draw speed if the distance criteria is met
            distance = d2((x, y), (r, s))
            if distance > DISTANCE_THRESH:
                # compute speed
                speed_str = str(distance / PX_PER_CM * FPS) + ' cm/s'
                print(speed_str)

        # combine mask with frame
        frame_final = cv2.add(frame, mask)

        cv2.imshow('frame', frame_final)

        # update for next frame
        old_gray = frame_gray.copy()
        prev_pts = good_next_pts.reshape(-1, 1, 2)
        frame_counter += 1
        if cv2.waitKey(10) == ord('q'):
            break

# clean up resources
cv2.destroyAllWindows()
video_cap.release()
```


In this video, you will be displaying the speed of the points on the video frame.

The first step for doing this is to create an additional mask layer, which will be called **mask_text**. See the code below:

```
# create a mask for the lines
mask = np.zeros_like(frame)
mask_text = np.zeros_like(frame)

# main UI loop
while True:
    # reset the lines
    if frame_counter % REFRESH_RATE == 0:
        mask.fill(0)
        mask_text.fill(0)
```

Now you can start drawing on the **mask_text**. This will be done in the **# compute speed section**. See the code below:

```
# draw speed if the distance criteria is met
distance = d2((x, y), (r, s))
if distance > DISTANCE_THRESH:
    # compute speed
    speed_str = str(distance / PX_PER_CM * FPS) + ' cm/s'
    print(speed_str)
    cv2.putText(mask_text, speed_str, (x, y), cv2.FONT_HERSHEY_TRIPLEX, 0.5, (0,255,0
))
```

This uses the OpenCV function **cv2.putText**. To find out more about this function and how its parameters operate, please check the documentation here: https://docs.opencv.org/3.0-beta/modules/imgproc/doc/drawing_functions.html?highlight=puttext#cv2.putText. You can put in whatever you looks best to you for the fontFace, fontScale, and color.

Now you need to use **cv2.add** to add **mask_text** to your frame. As **cv2.add** can only add two things together, you have to add the new mask to **frame_final**. See the code below:

```
# combine mask with frame
frame_final = cv2.add(frame, mask)
frame_final = cv2.add(frame_final, mask_text)
```

You can now run the app in the terminal. As a tip to enable quickly changing parameters, you can press the up key in the terminal to get the last used command.

Here is the full code for this lesson:

```
# -*- coding: utf-8 -*-
import cv2
import numpy as np
```



```
# feature parameters
feature_params = {
        'maxCorners': 100,
        'qualityLevel': 0.3,
        'minDistance': 7
}

# camera and video intrinsics
FPS = 30
PX_PER_CM = 370

# app parameters
REFRESH_RATE = 20
DISTANCE_THRESH = 20

# compute Euclidean distance (distance formula)
def d2(p, q):
    return np.linalg.norm(np.array(p) - np.array(q))

# load video and read in the first frame
video_cap = cv2.VideoCapture('test.mov')
_, frame = video_cap.read()
frame_counter = 1

# convert first frame to grayscale and pick points to track
old_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
prev_pts = cv2.goodFeaturesToTrack(image=old_gray, **feature_params)

# show features on first frame
"""
for pt in prev_pts:
    x, y = pt.ravel()
    cv2.circle(frame, (x, y), 5, (0,255,0), -1)
cv2.imshow('features', frame)
cv2.waitKey(0)
"""

# create a mask for the lines
mask = np.zeros_like(frame)
mask_text = np.zeros_like(frame)

# main UI loop
while True:
    # reset the lines
    if frame_counter % REFRESH_RATE == 0:
        mask.fill(0)
        mask_text.fill(0)

    # read in a video frame
    _, frame = video_cap.read()
    if frame is None:
        break
    frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```



```
    # compute optical flow points
    next_pts, statuses, _ = cv2.calcOpticalFlowPyrLK(prevImg=old_gray, nextImg=frame_
gray, prevPts=prev_pts, nextPts=None)

    # only keep the optical flow points that are valid
    good_next_pts = next_pts[statuses == 1]
    good_old_pts = prev_pts[statuses == 1]

    # draw optical flow lines
    for good_next_pt, good_old_pt in zip(good_next_pts, good_old_pts):
        # get new and old points
        x, y = good_next_pt.ravel()
        r, s = good_old_pt.ravel()

        # draw the optical flow line
        cv2.line(mask, (x, y), (r, s), (0,255,0), 2)

        # draw the coordinate of the corner points in this frame
        cv2.circle(frame, (x, y), 5, (0,255,0), -1)

        # draw speed if the distance criteria is met
        distance = d2((x, y), (r, s))
        if distance > DISTANCE_THRESH:
            # compute speed
            speed_str = str(distance / PX_PER_CM * FPS) + ' cm/s'
            print(speed_str)
            cv2.putText(mask_text, speed_str, (x, y), cv2.FONT_HERSHEY_TRIPLEX, 0.5,
(0,255,0))

    # combine mask with frame
    frame_final = cv2.add(frame, mask)
    frame_final = cv2.add(frame_final, mask_text)

    cv2.imshow('frame', frame_final)

    # update for next frame
    old_gray = frame_gray.copy()
    prev_pts = good_next_pts.reshape(-1, 1, 2)
    frame_counter += 1
    if cv2.waitKey(10) == ord('q'):
        break

# clean up resources
cv2.destroyAllWindows()
video_cap.release()
```

Congratulations on finishing this course. Thank you for learning with Zenva.

