Optical Flow depth calculation

The procedure to claculate the depth map is as follows:

1. load up a video file and iterate over the frames
2. find the Aruco markers and calulate the coordinates of the centers
3. Using the markers find the locations of the Opitcal Flow zone centers (OFCn)
4. Calculate the dense optical flow of successive frames using Farneback algorithm
5. Extract the flow from the Optical Flow Zones (OFZn) and:
    A. remove outliers (> than 1 stddev)
    B. find the mean value for each zone
6.

In [1]:
```
user = 'marcvanzyl'
```

In [2]:
```
import numpy as np
import cv2, os
from cv2 import aruco
import matplotlib.pyplot as plt
import matplotlib as mpl
import pandas as pd
%matplotlib inline
```

The board

The following line of the board was updated to reflect the correct scale of the board. This is necessary becuase the `board` object is used later

`board = aruco.CharucoBoard_create(7, 5, .04026, .8*.04026, aruco_dict)`

Notice the board object returned. Is contains all the magic of the CharUco pattern. For the image interpretation and to calibrate the cameras the cameras need a picture of the board and also information about the picture (ie. the details of the board). This is all contained in the board object.

In [3]:
```
board_size = 'experimental setup'

if board_size == '7x5':
    chessboard_num_squares_across = 7
    chessboard_num_squares_up = 5
    chessboard_square_size = 0.04026
    chessboard_aruco_ratio = 0.8   # this is a fraction of chessboard_square_
    aruco_dict = aruco.Dictionary_get(aruco.DICT_6X6_250)


if board_size == '12x8':
    chessboard_num_squares_across = 12
    chessboard_num_squares_up = 8
    chessboard_square_size = 1
    chessboard_aruco_ratio = 0.7   # this is a fraction of chessboard_square_
    aruco_dict = aruco.Dictionary_get(aruco.DICT_5X5_250)

elif board_size == 'experimental setup':
    aruco_dict = aruco.Dictionary_get(aruco.DICT_4X4_50)
```

The board object contains all the vectors (pointing from the bottom left corner) to each of the corners present on the board. The two kinds of objects are **aruco markers** and the **checkerboard**. The aruco markers are called ' `markers` ' in the code and documentation. The term `marker corners` means the set of 4 corners around each aruco maker.

The aruco markers can be extracted from the board using the `aruco.getBoardObjectAndImagePoints(board, makerCorners command)` .

Once the algorithm detected the markerCorners then it can interpolate between the marker corners to find the checkerboard corners. The positions of checkerboard "inside" corners can be extracted using the folowing.

The corners are labeled starting from 0 (bottom left) and going right

Now using Charuco

This function:

1. finds the locations of the corners of the aruco squares (`cv2.aruco.detectMarkers`)
2. if markers were found interpolates to find the checkerboard markers between them (`cv2.aruco.interpolateCornersCharuco`)
3. zooms into each checkerboard corner to get sub-pixel accuracy using (`cv2.cornerSubPix`)

In [4]:
```
def find_checkerboard_corners(img, board, clipLimit=2.0, verbose=False):

    # These are parameters used by the cv2.cornerSubPix function
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 100, 0.00

    # increase the contraxt
    img = increase_contrast(img, clipLimit=clipLimit)

    # convert the image to grayscale
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    detect_params = aruco.CORNER_REFINE_SUBPIX

    # find the aruco corners and the ids of each corner
    corners, ids, rejectedImgPoints = cv2.aruco.detectMarkers(gray, aruco_dic

    if verbose:
        print('Found {} aruco marker corners'.format(len(ids)))

    if len(ids)>0:
        (retval, charucoCorners,
         charucoIds) = cv2.aruco.interpolateCornersCharuco(corners, ids, gray
        if verbose:
            print('Found {} checker corners'.format(len(charucoIds)))
        if len(charucoIds)>0:
            # SUB PIXEL DETECTION
            for corner in charucoCorners:
                if verbose:
                    print('Sub pixel optimization:')
                    print(corner)
                cv2.cornerSubPix(gray, corner,
                                 winSize = (5,5),
                                 zeroZone = (-1,-1),
                                 criteria = criteria)
                if verbose:
                    print(corner)
                    print('+++')

    return charucoCorners, charucoIds, gray.shape
```

One big problem is that you need to give the stereo calibration only coners that appear in both cameras. Fortunately, the detection returns the ids of the corners in the `charucoIds`. These `charucoIds` correspond to the numbering system mentioned above (bottom left is 0 and starts going across to the right)

In [5]:
```
import pickle
# load the camera calibration data
cam_calOF = pickle.load( open('OFCameraCalibration.p', 'rb'))
```

In [6]:
```
cam_calOF.keys()
```

Out[6]:
```
dict_keys(['camera_name', 'ret', 'mtx', 'dist', 'rvecs', 'tvecs'])
```

Now we can check

Camera features:

- sensor size = 3.68 x 2.76 mm
- sensor resolution = 3280 × 2464
- focal length = 3.04 mm

$$d_{mm} = \frac{pix \times 3.68}{3280}$$

The depth can now be found

$$Z = \frac{T \times f}{d_{mm}}$$

In [7]:
```
import pandas as pd
```

In [8]:

```
# adapted from here https://stackoverflow.com/questions/39308030/how-do-i-inc
def increase_contrast(img, clipLimit=3.0, verbose=False):

    if clipLimit>0.0:
        #-----Converting image to LAB Color model----------------------------
        lab= cv2.cvtColor(img, cv2.COLOR_BGR2LAB)
        if verbose:
            cv2.imshow("lab",lab)

        #-----Splitting the LAB image to different channels-------------------
        l, a, b = cv2.split(lab)
        if verbose:
            cv2.imshow('l_channel', l)
            cv2.imshow('a_channel', a)
            cv2.imshow('b_channel', b)

        #-----Applying CLAHE to L-channel-------------------------------------
        clahe = cv2.createCLAHE(clipLimit=clipLimit, tileGridSize=(8,8))
        cl = clahe.apply(l)
        if verbose:
            cv2.imshow('CLAHE output', cl)

        #-----Merge the CLAHE enhanced L-channel with the a and b channel-----
        limg = cv2.merge((cl,a,b))
        if verbose:
            cv2.imshow('limg', limg)

        #-----Converting image from LAB Color model to RGB model--------------
        final = cv2.cvtColor(limg, cv2.COLOR_LAB2BGR)
        if verbose:
            cv2.imshow('final', final)
    else:
        final = img.copy()

    return final
```

In [9]:

```
# finds the corners of all teh identifiable Aruco markers

def find_charuco_marker_corners(img, aruco_dict, clipLimit=3.0, verbose=False

    # These are parameters used by the cv2.cornerSubPix function
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10000, 1e

    # increase the contraxt
    img = increase_contrast(img, clipLimit=clipLimit)



    # convert the image to grayscale
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)


    # find the aruco corners and the ids of each corner
    corners, ids, rejectedImgPoints = cv2.aruco.detectMarkers(gray, aruco_dic

    if verbose:
        print('Found {} aruco marker corners'.format(len(ids)))


    #  now find the central points of the aruco markers by averaging the four
    if  len(ids) > 0:
        # SUB PIXEL DETECTION
        for corner in corners:
            if verbose:
                print('Sub pixel optimization:')
                print(corner)
            cv2.cornerSubPix(gray, corner,
                             winSize = (4,4),
                             zeroZone = (-1,-1),
                             criteria = criteria)
            if verbose:
                print(corner)
                print('+++')



    return np.array(corners), ids, gray.shape
```

In [10]:
```
def find_aruco_center(corners, ids):
    # find the center point of the 4 corner of the aruco markers
    centers = []
    for cnrs in corners:
        center = np.array((np.average(cnrs[0][:,0]), np.average(cnrs[0][:,1])
        centers.append(center)
    centers = np.array(centers)
    centers = np.array(centers).reshape((-1,1,2))
    center_ids = np.arange(centers.shape[0]).reshape(-1,1)

    return centers, ids
```

In [11]:
```
image_number = 0
```

Load the list of files

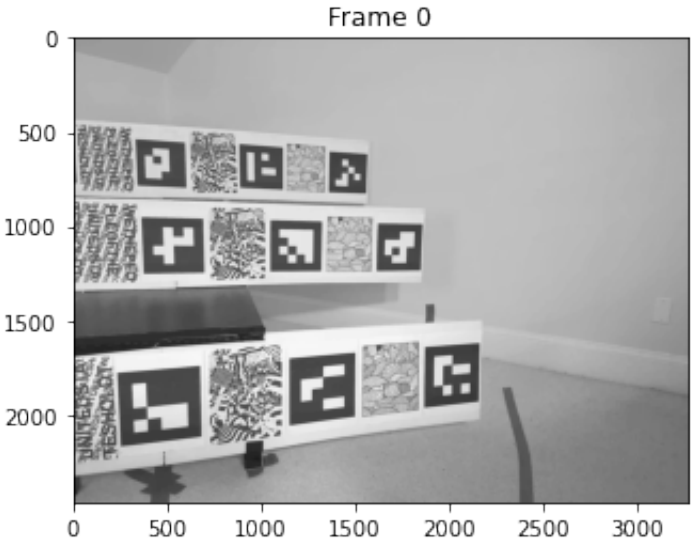
In [12]:
```
datadir = "/Users/{}/Google Drive/ScienceFair2021/DataCapture/smooth/".format


video_files = np.array([f for f in os.listdir(datadir) if f.endswith(".mp4")

# just sorts the files according to the number so we match picture 1_A with 1
#orderR = np.argsort([int((p.split('_')[-1]).split('.')[0]) for p in video_fi
#video_files = video_files[orderR]
video_files.sort()
```

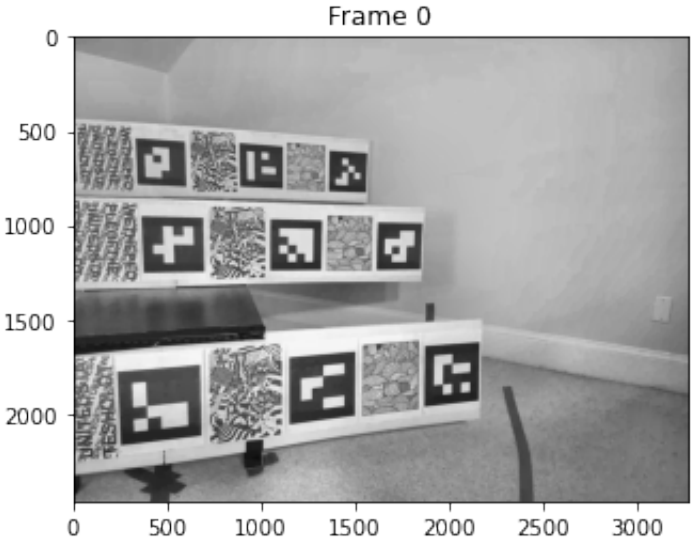
In [13]:
```
video_files.shape
```

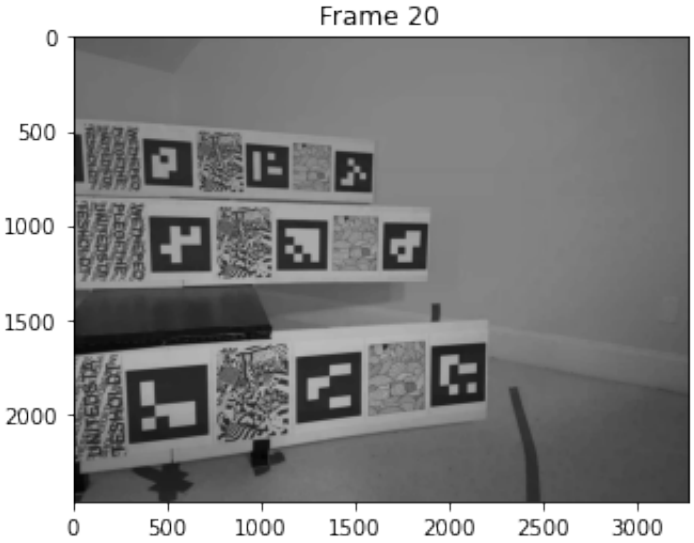
Out[13]:
```
(80,)
```

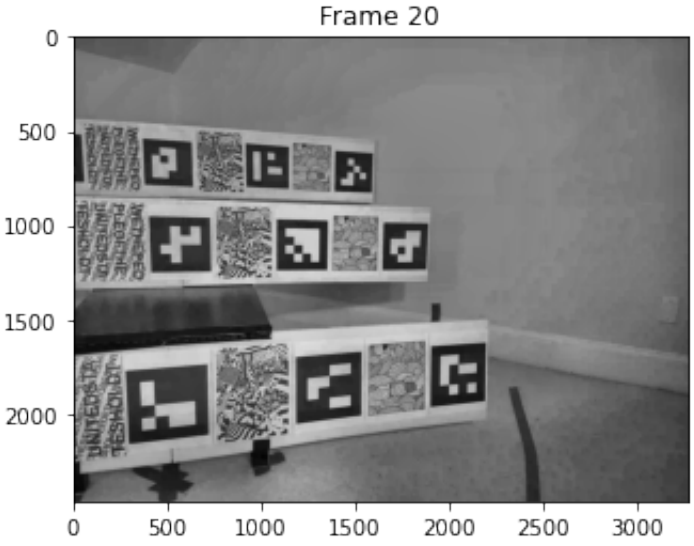
In [24]:

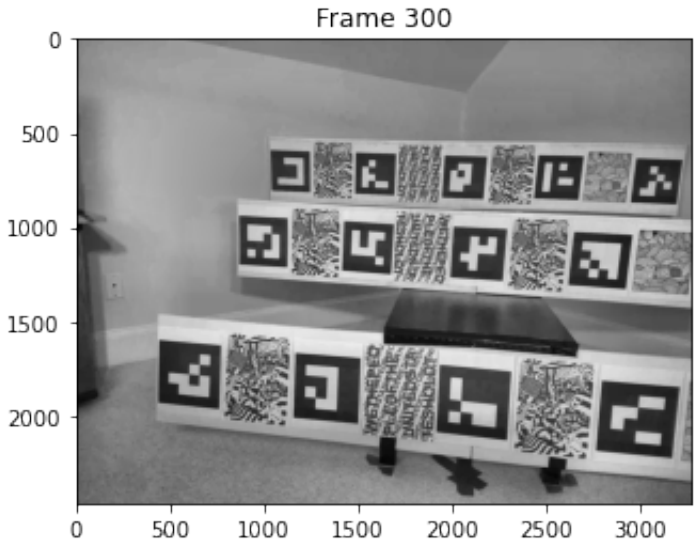
```
video_capture = cv2.VideoCapture('{}{}'.format(datadir, video_files[1]))
#video_capture.set(cv2.CAP_PROP_FOURCC, cv2.VideoWriter_fourcc(*'h264'))

video_capture.set(cv2.CAP_PROP_FOURCC, cv2.VideoWriter_fourcc(*'H264'))

frame = 0
while video_capture.isOpened():
    ret, image = video_capture.read()


    if not ret:
        break
    if frame%20 == 0:
        image2 = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

        plt.imshow(image2,cmap='gray')
        plt.title('Frame {}'.format(frame))
        plt.show()
        plt.imshow(cv2.cvtColor(increase_contrast(image, clipLimit=2.0, verbo
        plt.title('Frame {}'.format(frame))
        plt.show()
    frame += 1
print('Done!')
```

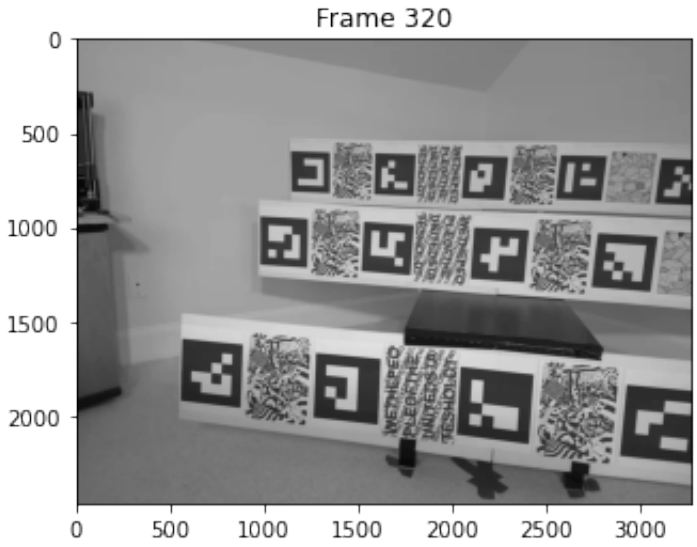


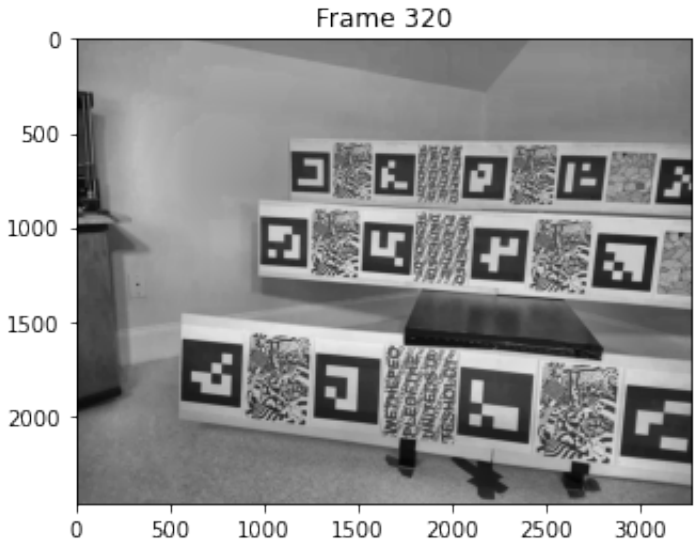

Frame 0
0
500
1000
1500
2000
0
500
1000
1500
2000
2500
3000


Frame 20
0
500
1000
1500
2000
0
500
1000
1500
2000
2500
3000


Frame 20
0
500
1000
1500
2000
0
500
1000
1500
2000
2500
3000








```
Done!
```

Run the analysis

In [14]:

```
# function to find the centers of the optical flow zones
# sometimes an aruco marker may be outside of the view, then extrapolate to f
def find_target_centers(sorted_centers, sorted_center_ids):
    front_row = [1,2,3] # the OFZones in the front row are located between ar
    middle_row = [6,7,8]
    back_row = [11, 12, 13]
    results = []

    rows = [{'row':'front', 'aruco_ids':front_row}, {'row':'middle', 'aruco_i

    for row in rows:
        result_row = row.copy()

        aruco_target = row['aruco_ids']

        # need to check we have 2 of three aruco_ids per row
        ids_count = 0
        for ids in aruco_target:
            if ids in sorted_center_ids:
                ids_count += 1

        if ids_count > 1:
            if aruco_target[1] in sorted_center_ids:
                if aruco_target[0] in sorted_center_ids:
                    # calc by interpolation
                    target_l = (sorted_centers[np.where(sorted_center_ids==ar
                else:
                    # calc by extrapolation
                    target_l = ( sorted_centers[np.where(sorted_center_ids==a
                     (sorted_centers[np.where(sorted_center_ids==aruco_target

                if aruco_target[2] in sorted_center_ids:
                    target_r = (sorted_centers[np.where(sorted_center_ids==ar
                else:
                    target_r = ( sorted_centers[np.where(sorted_center_ids==a
                     (sorted_centers[np.where(sorted_center_ids==aruco_target
            else:  # we have [0] and [2]
                delta = (sorted_centers[np.where(sorted_center_ids==aruco_tar

                target_l = sorted_centers[np.where(sorted_center_ids==aruco_t

                target_r = sorted_centers[np.where(sorted_center_ids==aruco_t


            result_row['left_zone_center'] = target_l.reshape(2)
            result_row['right_zone_center'] = target_r.reshape(2)

        else:
            result_row['left_zone_center'] = np.array([np.nan,np.nan])
            result_row['right_zone_center'] = np.array([np.nan,np.nan])



        results.append(result_row)
    return results
```

In [15]:
```
# remove outliers using statistics
def clean_OF(of_array, std=2):

    of_x = of_array[:,:,0]
    mean_x = np.nanmean(of_x)
    std_x = np.std(of_x)

    # create boolean arrays of where the readings are out-of-bounds
    flow_too_large = of_x > mean_x + std*std_x
    flow_too_small = of_x < mean_x - std*std_x

    # create a copy of flow_x
    cleaned_flow_x = of_x.copy()
    # set all the our-of-bound elements to np.NaN
    cleaned_flow_x[np.logical_or(flow_too_large, flow_too_small)] = np.nan

    return cleaned_flow_x
```

In [16]:
```
def read_frame(handle, counter):
    counter += 1
    ret, frame = handle.read()
    return ret, frame, counter


def get_frame(handle, counter):
    counter += 1
    ret = handle.grab()
    return ret, counter

import math
```

In [17]:
```
# this is the main function that processes the file
def process_file(file_path,
                 skip_frames=2,
                 mean_pixels=10,
                 verbose=0,
                grabframe=-1):

    result_list = []

    counter = 0
    result_df = pd.DataFrame(columns=['AM0','AM1','AM2','AM3','AM4','AM5','AM
                                      'OFC0', 'OFC1', 'OFC2', 'OFC3', 'OFC4',
                                      'OFZ0','OFZ1','OFZ2','OFZ3','OFZ4','OFZ5


    # the size of the square to average the flow over
    center_range = mean_pixels

    cap = cv2.VideoCapture(file_path)
    print("Working with: {}".format(file_path))

    # Thresholds to extract the right frames
    num_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
    mid = num_frames/2
```

```
lower_bound = int(mid - min(60,mid-21)) # drops the first 25% frames
upper_bound = int(mid + min(60,mid-21)) # drops the last 25% second
print('Frames: {}  mid: {} lower: {} uuper: {}'.format(num_frames, mid, l


# skip the initial frames
for _ in range(lower_bound):
    ret, counter = get_frame(cap, counter)

# read the first frame
ret, frame1, counter = read_frame(cap, counter)
last_pic_color = cv2.undistort(frame1,cam_calOF['mtx'],cam_calOF['dist'],
last_pic_grey = cv2.cvtColor(last_pic_color,cv2.COLOR_BGR2GRAY)

# this creates the array for the sorted centers
sorted_centers = np.zeros([10,2])

# initialze the array of flows
flow = np.zeros_like(frame1)
first_optical_flow_frame = True



while(counter < upper_bound):

    # skip the frames
    for _ in range(skip_frames-1):
        ret, counter = get_frame(cap, counter)

    # read a frame
    ret, curr_pic_color, counter = read_frame(cap, counter)

    curr_pic_color = cv2.undistort(curr_pic_color,cam_calOF['mtx'],cam_ca
    curr_pic_grey = cv2.cvtColor(curr_pic_color, cv2.COLOR_BGR2GRAY)

    if first_optical_flow_frame:
        flags = 0
        first_optical_flow_frame = False
    else:
        flags = cv2.OPTFLOW_USE_INITIAL_FLOW


    flow = cv2.calcOpticalFlowFarneback(last_pic_grey, curr_pic_grey, flo

    if verbose>0:
        print('Image Frame Number: {}'.format(counter))


    # find the four corners of each aruco marker
    arucoCorners, arucoIds, imsize =  find_charuco_marker_corners(curr_pi
                                                  aruco_dict, clipL
                                                  verbose=False)

    # now find the center point of each aruco marker
    centers, center_ids = find_aruco_center(arucoCorners, arucoIds)

    # next sort the centers by center_ids from 0 to 9
```

```
        center_ids = center_ids.reshape(-1)
        centers = centers.reshape(-1,2)
        if verbose>1:
            print('CenterIds {}'.format(center_ids))

        # insert the centers into the sorted centers array in the right order
        arr1inds = center_ids.argsort()
        sorted_centers = centers[arr1inds]
        sorted_center_ids = center_ids[arr1inds]

        # add sorted centers to the frame results series
        frame_res = pd.Series(index=['AM0','AM1','AM2','AM3','AM4','AM5','AM6
                                     'OFC0', 'OFC1', 'OFC2', 'OFC3', 'OFC4',
                                     'OFZ0','OFZ1','OFZ2','OFZ3','OFZ4','OFZ5

        # save the aruco marker centers to the series
        for ind, ids in enumerate(sorted_center_ids):
            frame_res['AM{}'.format(ids)] = sorted_centers[ind]

        frame_result_list = []

        # find the centers of all the target zones
        targets = find_target_centers(sorted_centers, sorted_center_ids)

        of_zone_number = 0

        for row in targets:
            frame_res['OFC{}'.format(of_zone_number)] = row['left_zone_center
            frame_res['OFZ{}'.format(of_zone_number)] = 777 #get_cleaned_OF(f

            of_zone_number += 1

            frame_res['OFC{}'.format(of_zone_number)] = row['right_zone_cente
            frame_res['OFZ{}'.format(of_zone_number)] = 777 #get_cleaned_OF(f

            of_zone_number += 1




        # extract the optical flow in the areas around the target zones
        for of_center in frame_res[frame_res.index.str.contains('OFC')].index
            center = frame_res[of_center]

            if verbose>1:
                print('OFCenter {} : {} - flow.shape {}'.format(of_center, ce

            center_x = center[0]
            center_y = center[1]

            if not (math.isnan(center_x) or math.isnan(center_y)):
                sub_OF_array = flow[int(center_y-mean_pixels/2):int(center_y+
                                    int(center_x-mean_pixels/2):int(center_x+


                # extract x component of OF and clean the optical flow
                cleaned_of_x = clean_OF(sub_OF_array, std=2)
```

```
            # find the mean of the optical flow in x and y
            mean_of_x = np.nanmean(cleaned_of_x)/skip_frames


            # save the results in a df

            frame_res['OFZ{}'.format(of_center[3:])] = mean_of_x
        else:
            print('WARNING: Nan OF center')
            frame_res['OFZ{}'.format(of_center[3:])] = np.nan

    if verbose>0:
        print(frame_res)


    result_df.loc[counter] = frame_res

    if verbose>1:

        hsv = np.zeros_like(frame1)
        hsv[...,1] = 255


        x = sorted_centers[:,0]
        y = sorted_centers[:,1]
        mag = (cv2.multiply(cv2.add(flow[...,0],-10),5))

        mag[mag<0.0] = 0
        mag[mag>255] = 255

        print("mean: {} max: {}  min: {} {}".format(mag.mean(), mag.max()

        # edge detection for the outlines
        edges = cv2.Canny(curr_pic_color, 50, 100)


        hsv[...,0] = 0
        #hsv[...,2] = cv2.normalize(mag,None,0,255,cv2.NORM_MINMAX)
        hsv[...,2] = mag
        rgb = cv2.cvtColor(hsv,cv2.COLOR_HSV2BGR)

        annot_font = {'fontname':'Arial', 'size':'14','weight':'bold'}

        fig, axs = plt.subplots(figsize=(36,24))
        rgb[edges>100,1] = 255
        axs.imshow(rgb)
        axs.scatter(x,y, color='r')
        for i, ids in enumerate(sorted_center_ids):
            axs.annotate('{}'.format(int(ids)), (x[i]+20, y[i]+30), color

        for ind in frame_res[frame_res.index.str.contains('OFC')].index:
            axs.annotate('{}'.format(ind), frame_res[ind], color='w', **a

        plt.show()
```

```
            if counter == grabframe:  # this is a hack to save a particular i
                save_dict = {}
                save_dict['filename'] = file_path
                save_dict['frame'] = counter
                save_dict['img'] = curr_pic_grey

                save_dict['flow'] = flow

                save_dict['centers'] = sorted_centers

                save_dict['center_ids'] = sorted_center_ids

                save_dict['frame_res'] = frame_res


                pickle.dump(save_dict, open('{}frame_results.p'.format(datadi



        last_pic_grey = np.copy(curr_pic_grey)

    #pickle.dump(result_df, open('{}{}_res.p'.format(datadir, file[:-5]), 'wb
    return result_df
```

Process File with Frame interleave

In [18]:
```
def read_frame(handle, counter):
    counter += 1
    ret, frame = handle.read()
    return ret, frame, counter


def get_frame(handle, counter):
    counter += 1
    ret = handle.grab()
    return ret, counter

import math
```

In [39]:
```
#New version that interleaves frames
def process_file2(file_path,
                  skip_frames=4,
                  mean_pixels=120,
                  verbose=0,
                  grabframe=-1):

    result_list = []

    counter = 0
    result_df = pd.DataFrame(columns=['AM0','AM1','AM2','AM3','AM4','AM5','AM
                                      'OFC0', 'OFC1', 'OFC2', 'OFC3', 'OFC4',
                                      'OFZ0','OFZ1','OFZ2','OFZ3','OFZ4','OFZ5
```

```
pics = []

# the size of the square to average the flow over
center_range = mean_pixels

cap = cv2.VideoCapture(file_path)
print("Working with: {}".format(file_path))

# Thresholds to extract the right frames
num_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
mid = num_frames/2
lower_bound = int(21) # drops the first 25% frames
upper_bound = int(num_frames-21) # drops the last 25% second
print('Frames: {}  mid: {} lower: {} uuper: {}'.format(num_frames, mid, l


# skip the initial frames
for _ in range(lower_bound):
    ret, counter = get_frame(cap, counter)

for i in range(skip_frames):

    # read the frames
    ret, frame1, counter = read_frame(cap, counter)
    last_pic_color = cv2.undistort(frame1,cam_calOF['mtx'],cam_calOF['dis
    pics.append(cv2.cvtColor(last_pic_color,cv2.COLOR_BGR2GRAY))

# this creates the array for the sorted centers
sorted_centers = np.zeros([10,2])

# initialze the array of flows
flow = np.zeros_like(frame1)
first_optical_flow_frame = True



while(counter < upper_bound):

    # skip the frames
    #for _ in range(skip_frames-1):
    #    ret, counter = get_frame(cap, counter)

    # read a frame
    ret, curr_pic_color, counter = read_frame(cap, counter)

    curr_pic_color = cv2.undistort(curr_pic_color,cam_calOF['mtx'],cam_ca
    curr_pic_grey = cv2.cvtColor(curr_pic_color, cv2.COLOR_BGR2GRAY)

    if first_optical_flow_frame:
        flags = 0
        first_optical_flow_frame = False
    else:
        flags = cv2.OPTFLOW_USE_INITIAL_FLOW
        # clear the y flow to give a better start
        flow[:,:,1] = 0
```

```
        flow = cv2.calcOpticalFlowFarneback(pics[0], curr_pic_grey, flow, 0.5

        if verbose>0:
            print('Image Frame Number: {}'.format(counter))


        # find the four corners of each aruco marker
        arucoCorners, arucoIds, imsize =  find_charuco_marker_corners(curr_pi
                                                                      aruco_dict, clipL
                                                                      verbose=False)

        # now find the center point of each aruco marker
        centers, center_ids = find_aruco_center(arucoCorners, arucoIds)

        # next sort the centers by center_ids from 0 to 9
        center_ids = center_ids.reshape(-1)
        centers = centers.reshape(-1,2)
        if verbose>1:
            print('CenterIds {}'.format(center_ids))

        # insert the centers into the sorted centers array in the right order
        arr1inds = center_ids.argsort()
        sorted_centers = centers[arr1inds]
        sorted_center_ids = center_ids[arr1inds]

        # add sorted centers to the frame results series
        frame_res = pd.Series(index=['AM0','AM1','AM2','AM3','AM4','AM5','AM6
                                     'OFC0', 'OFC1', 'OFC2', 'OFC3', 'OFC4',
                                     'OFZ0','OFZ1','OFZ2','OFZ3','OFZ4','OFZ5

        # save the aruco marker centers to the series
        for ind, ids in enumerate(sorted_center_ids):
            frame_res['AM{}'.format(ids)] = sorted_centers[ind]

        frame_result_list = []

        # find the centers of all the target zones
        targets = find_target_centers(sorted_centers, sorted_center_ids)
        """
        {'row': 'front',
          'aruco_ids': [1, 2, 3],
          'left_zone_center': array([2208.1284, 2125.8706], dtype=float32),
          'right_zone_center': array([3100.6724, 2243.5234], dtype=float32)}
          """
        of_zone_number = 0

        for row in targets:
            frame_res['OFC{}'.format(of_zone_number)] = row['left_zone_center
            frame_res['OFZ{}'.format(of_zone_number)] = 777 #get_cleaned_OF(f

            of_zone_number += 1

            frame_res['OFC{}'.format(of_zone_number)] = row['right_zone_cente
            frame_res['OFZ{}'.format(of_zone_number)] = 777 #get_cleaned_OF(f

            of_zone_number += 1
```

```
    # extract the optical flow in the areas around the target zones
    for of_center in frame_res[frame_res.index.str.contains('OFC')].index
        center = frame_res[of_center]

        if verbose>1:
            print('OFCenter {} : {} - flow.shape {}'.format(of_center, ce

        center_x = center[0]
        center_y = center[1]

        if not (math.isnan(center_x) or math.isnan(center_y)):
            sub_OF_array = flow[int(center_y-mean_pixels/2):int(center_y+
                                int(center_x-mean_pixels/2):int(center_x+


            # extract x component of OF and clean the optical flow
            cleaned_of_x = clean_OF(sub_OF_array, std=2)

            # find the mean of the optical flow in x and y
            mean_of_x = np.nanmean(cleaned_of_x)/skip_frames


            # save the results in a df

            frame_res['OFZ{}'.format(of_center[3:])] = mean_of_x
        else:
            print('WARNING: Nan OF center')
            frame_res['OFZ{}'.format(of_center[3:])] = np.nan

    if verbose>0:
        print(frame_res)


    result_df.loc[counter] = frame_res

    if verbose>1:

        hsv = np.zeros_like(frame1)
        hsv[...,1] = 255


        x = sorted_centers[:,0]
        y = sorted_centers[:,1]
        mag = (cv2.multiply(cv2.add(flow[...,0],-10),5))

        mag[mag<0.0] = 0
        mag[mag>255] = 255

        print("mean: {} max: {}  min: {} {}".format(mag.mean(), mag.max()

        # edge detection for the outlines
        edges = cv2.Canny(curr_pic_color, 50, 100)
```

```
            hsv[...,0] = 0
            #hsv[...,2] = cv2.normalize(mag,None,0,255,cv2.NORM_MINMAX)
            hsv[...,2] = mag
            rgb = cv2.cvtColor(hsv,cv2.COLOR_HSV2BGR)

            annot_font = {'fontname':'Arial', 'size':'14','weight':'bold'}

            fig, axs = plt.subplots(figsize=(36,24))
            rgb[edges>100,1] = 255
            axs.imshow(rgb)
            axs.scatter(x,y, color='r')
            for i, ids in enumerate(sorted_center_ids):
                axs.annotate('{}'.format(int(ids)), (x[i]+20, y[i]+30), color

            for ind in frame_res[frame_res.index.str.contains('OFC')].index:
                axs.annotate('{}'.format(ind), frame_res[ind], color='w', **a

            plt.show()

            if counter == grabframe:  # this is a hack to save a particular i
                save_dict = {}
                save_dict['filename'] = file_path
                save_dict['frame'] = counter
                save_dict['img'] = curr_pic_grey

                save_dict['flow'] = flow

                save_dict['centers'] = sorted_centers

                save_dict['center_ids'] = sorted_center_ids

                save_dict['frame_res'] = frame_res


                pickle.dump(save_dict, open('{}frame_results.p'.format(datadi


        for i in range(skip_frames-1):
            pics[i] = np.copy(pics[i+1])
        if skip_frames == 1:
            pics[0] = np.copy(curr_pic_grey)


    #pickle.dump(result_df, open('{}{}_res.p'.format(datadir, file[:-5]), 'wb
    return result_df
```

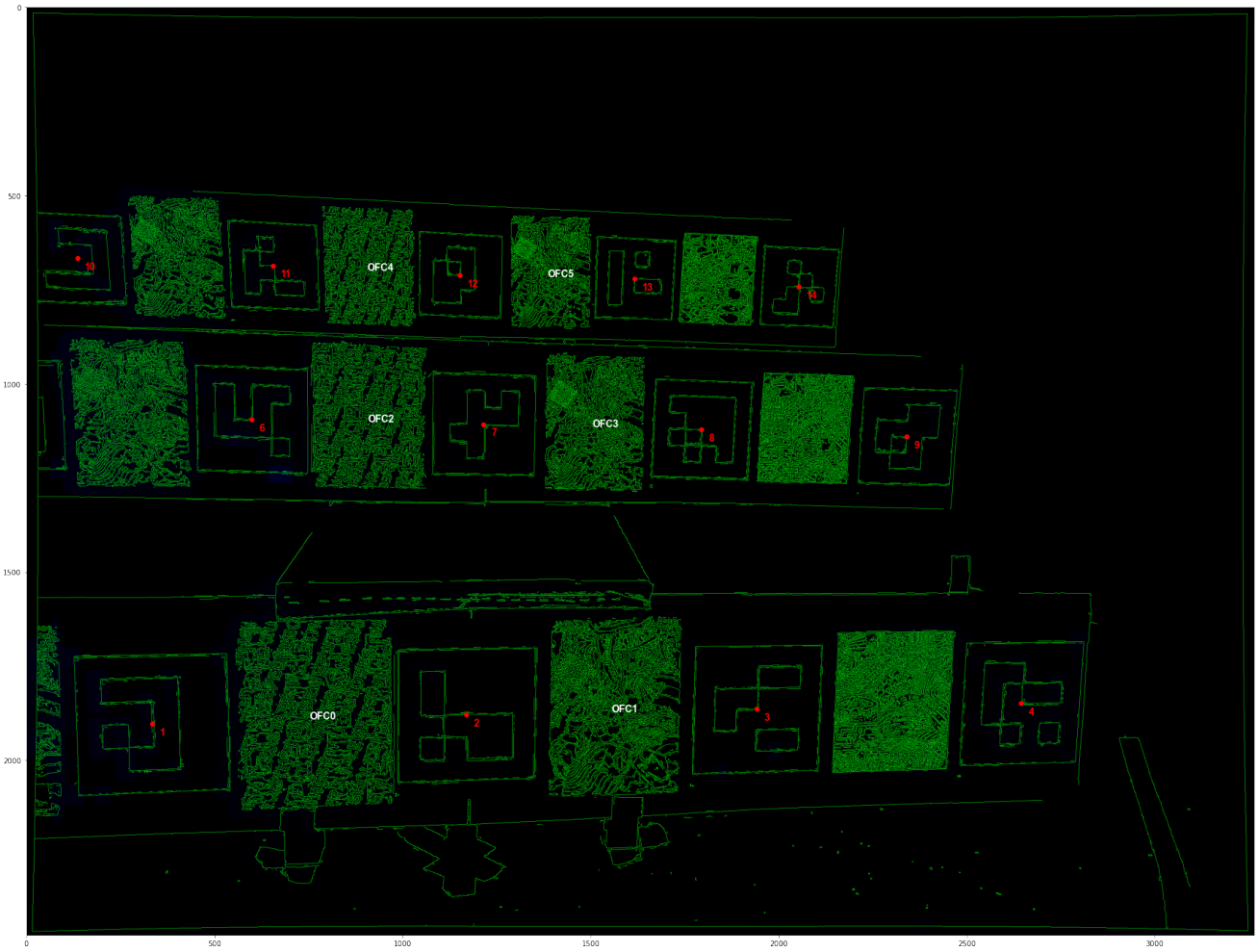
In [20]:

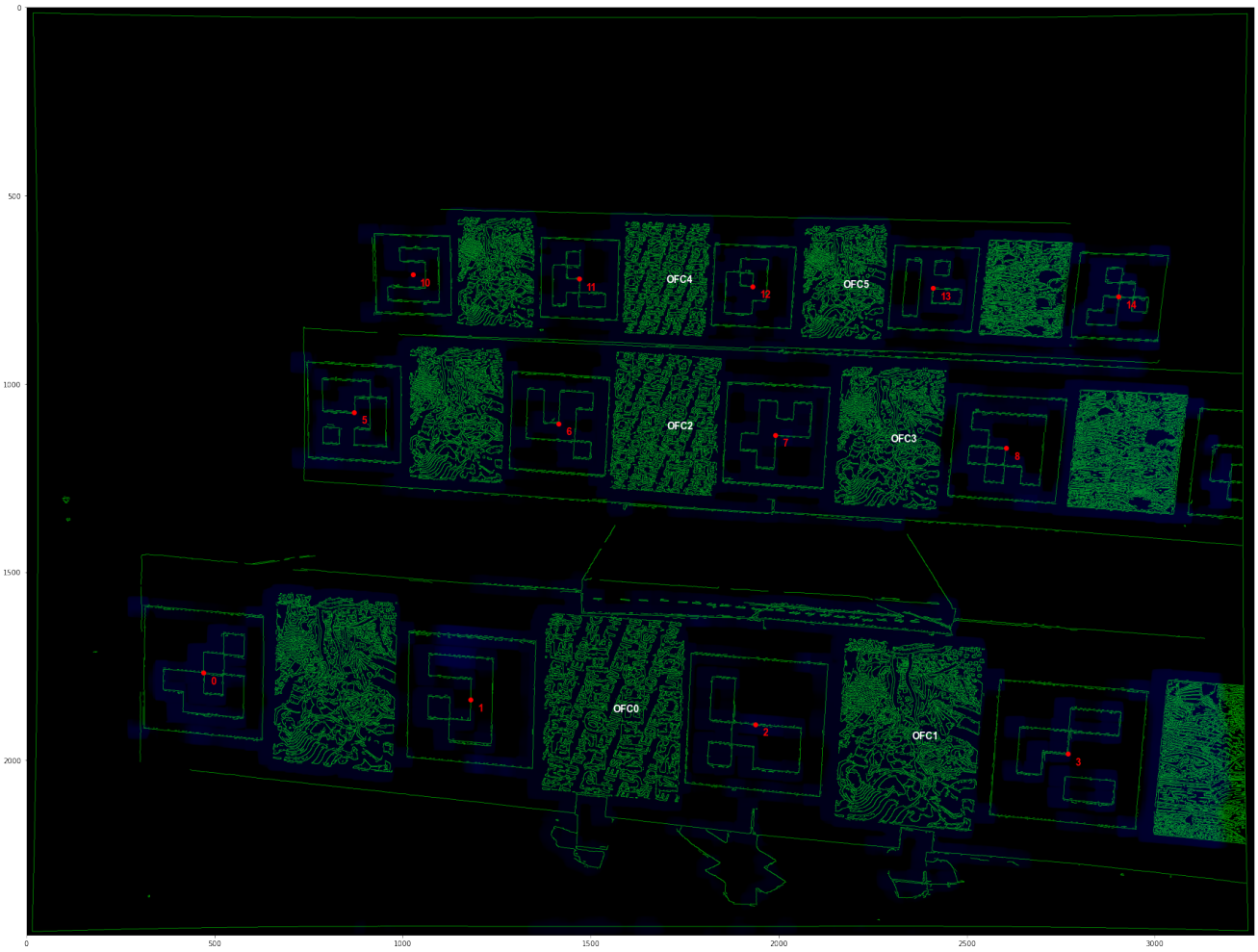
```
frame_res = pd.Series(index=['AM0','AM1','AM2','AM3','AM4','AM5','AM6','AM7',
                             'OFC0', 'OFC1', 'OFC2', 'OFC3', 'OFC4',
                             'OFZ0','OFZ1','OF20','OFZ3','OFZ4','OFZ5
```

In [21]:

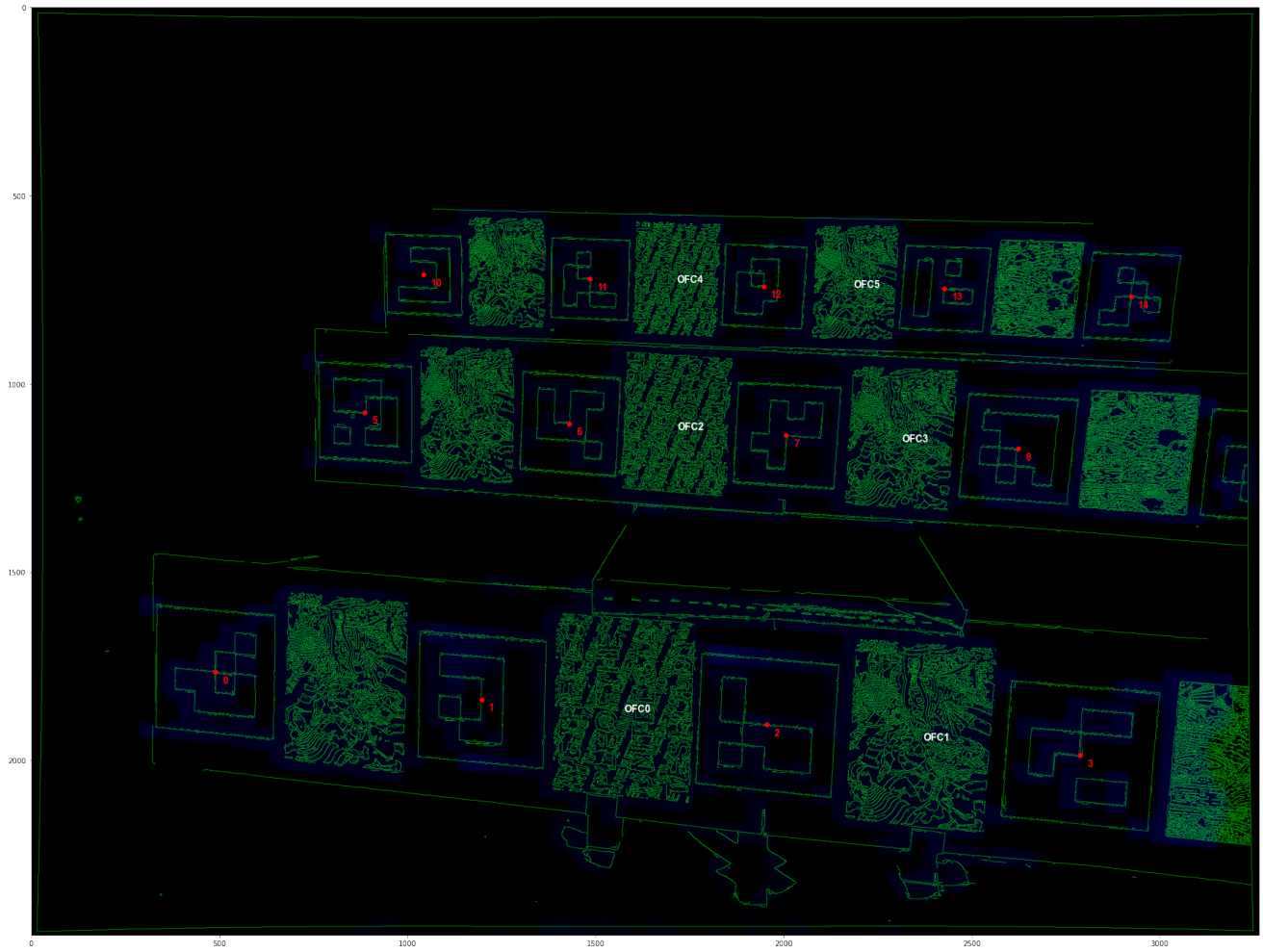
```
# set verbose = True to make plots of each frame
verbose = 3

for file in [video_files[0]]:
    res = process_file(datadir+file, mean_pixels=120, verbose=verbose, grabfr
    pickle.dump(res, open('{}{}_res.p'.format(datadir, file[:-4]), 'wb'))
```

```
Working with: /Users/gerrie/Google Drive/ScienceFair2021/DataCapture/smooth/Sm
ooth_15_-2_0_4623HF.mp4
Frames: 340  mid: 170.0 lower: 110 uuper: 230
Image Frame Number: 113
CenterIds [ 2  3  4  9  8  7 14 13 12  1  6 11 10]
OFCenter OFC0 : [ 751.5952 1890.7046] - flow.shape (2464, 3264, 2)
OFCenter OFC1 : [1554.9553 1871.1199] - flow.shape (2464, 3264, 2)
OFCenter OFC2 : [ 906.70496 1101.1432 ] - flow.shape (2464, 3264, 2)
OFCenter OFC3 : [1504.4266 1114.0641] - flow.shape (2464, 3264, 2)
OFCenter OFC4 : [904.594   698.36896] - flow.shape (2464, 3264, 2)
OFCenter OFC5 : [1384.9799  715.6706] - flow.shape (2464, 3264, 2)
AM0                         NaN
AM1       [334.2342, 1902.7257]
AM2      [1168.9563, 1878.6835]
AM3      [1940.9542, 1863.5562]
AM4      [2643.6606, 1847.8883]
AM5                         NaN
AM6       [598.3934, 1094.9148]
AM7      [1215.0166, 1107.3716]
AM8      [1793.8367, 1120.7566]
AM9      [2340.2007, 1139.8815]
AM10     [135.48676, 665.83594]
AM11     [656.31934, 685.77246]
AM12     [1152.8687, 710.96545]
AM13     [1617.0911, 720.37573]
AM14      [2053.6245, 742.2656]
OFC0      [751.5952, 1890.7046]
OFC1     [1554.9553, 1871.1199]
OFC2     [906.70496, 1101.1432]
OFC3     [1504.4266, 1114.0641]
OFC4       [904.594, 698.36896]
OFC5      [1384.9799, 715.6706]
OFZ0                    5.74583
OFZ1                    5.20019
OFZ2                     5.6365
OFZ3                    5.23698
OFZ4                     5.4388
OFZ5                    5.11074
dtype: object
mean: 1.7340022325515747 max: 39.090606689453125  min: 0.0 (2464, 3264, 2)
```

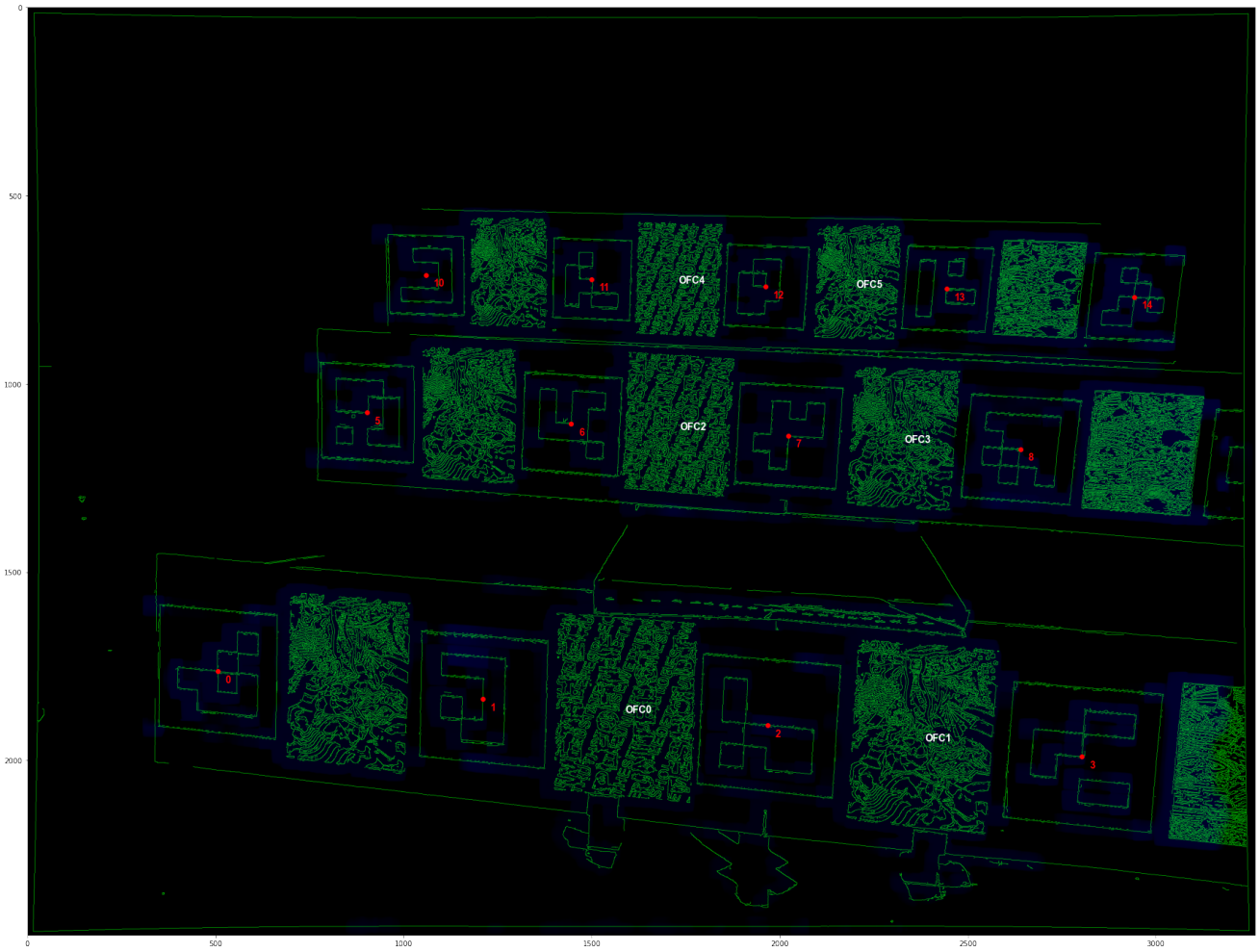

OFC0
OFC1
OFC2
OFC3
OFC4
OFC5
1
2
3
4
6
7
8
9
10
11
12
13
14
0
500
1000
1500
2000
2500
3000


OFC4
OFC5
OFC2
OFC3
OFC0
OFC1
0
1
2
3
5
6
7
8
10
11
12
13
14
0
500
1000
1500
2000
2500
3000

```
Image Frame Number: 108
CenterIds [ 3  2  1  0  8  7  6  5 14 13 12 11 10]
OFCenter OFC0 : [1575.3342 1871.8815] - flow.shape (2464, 3264, 2)
OFCenter OFC1 : [2370.4973 1945.9419] - flow.shape (2464, 3264, 2)
OFCenter OFC2 : [1718.4539 1120.8533] - flow.shape (2464, 3264, 2)
OFCenter OFC3 : [2314.6382 1154.1742] - flow.shape (2464, 3264, 2)
OFCenter OFC4 : [1715.4033  731.2466] - flow.shape (2464, 3264, 2)
OFCenter OFC5 : [2186.8572   743.83765] - flow.shape (2464, 3264, 2)
AM0     [488.62396, 1765.0272]
AM1     [1196.5454, 1838.2032]
AM2     [1954.1229, 1905.5597]
AM3      [2786.8716, 1986.324]
AM4                       NaN
AM5     [886.40326, 1075.4048]
AM6     [1430.2751, 1105.3422]
AM7     [2006.6326, 1136.3645]
AM8     [2622.6436, 1171.9839]
AM9                       NaN
AM10    [1043.0569, 709.62634]
AM11      [1484.143, 721.4212]
AM12     [1946.6636, 741.0719]
AM13    [2427.0508, 746.60345]
AM14      [2923.22, 768.46716]
OFC0    [1575.3342, 1871.8815]
OFC1    [2370.4973, 1945.9419]
OFC2    [1718.4539, 1120.8533]
OFC3    [2314.6382, 1154.1742]
OFC4     [1715.4033, 731.2466]
OFC5    [2186.8572, 743.83765]
OFZ0                   15.1176
OFZ1                   16.5159
OFZ2                   15.2337
OFZ3                   16.3365
OFZ4                   15.4386
OFZ5                   16.1943
dtype: object
mean: 10.331489562988281 max: 63.23373794555664  min: 0.0 (2464, 3264, 2)
```

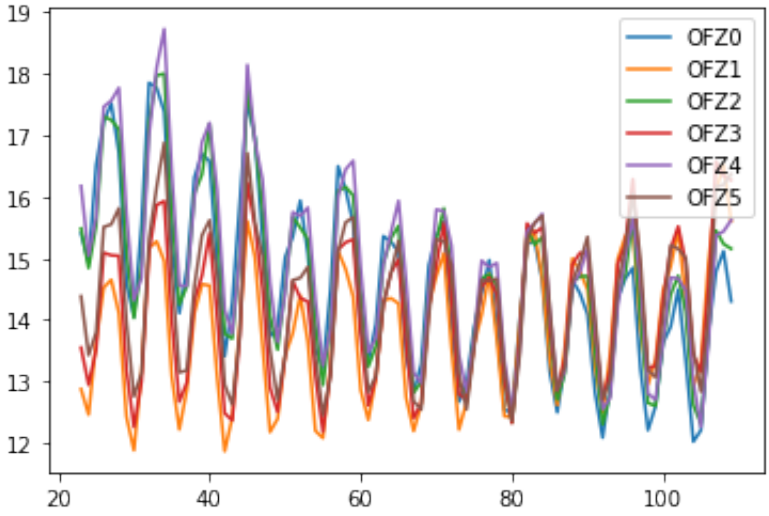

OFC4
OFC5
OFC2
OFC3
OFC0
OFC1
0
1
2
3
5
6
7
8
10
11
12
13
14
0
500
1000
1500
2000
2500
3000

```
Image Frame Number: 109
CenterIds [ 3  2  1  0  8  7  6  5 14 13 12 11 10]
OFCenter OFC0 : [1590.0269 1872.4312] - flow.shape (2464, 3264, 2)
OFCenter OFC1 : [2386.268  1948.4915] - flow.shape (2464, 3264, 2)
OFCenter OFC2 : [1733.9072 1121.531 ] - flow.shape (2464, 3264, 2)
OFCenter OFC3 : [2331.2456 1155.3657] - flow.shape (2464, 3264, 2)
OFCenter OFC4 : [1731.1917  731.8461] - flow.shape (2464, 3264, 2)
OFCenter OFC5 : [2203.328    744.39355] - flow.shape (2464, 3264, 2)
AM0     [505.54984, 1763.1472]
AM1     [1211.2096, 1837.6774]
AM2      [1968.844, 1907.1848]
AM3     [2803.6924, 1989.7982]
AM4                        NaN
AM5      [903.0447, 1075.4497]
AM6     [1445.4713, 1105.8295]
AM7      [2022.343, 1137.2325]
AM8      [2640.1484, 1173.499]
AM9                        NaN
AM10     [1059.4662, 710.4115]
AM11    [1499.8933, 722.09644]
AM12     [1962.4899, 741.5958]
AM13     [2444.166, 747.19135]
AM14    [2942.5496, 768.91693]
OFC0    [1590.0269, 1872.4312]
OFC1     [2386.268, 1948.4915]
OFC2     [1733.9072, 1121.531]
OFC3    [2331.2456, 1155.3657]
OFC4     [1731.1917, 731.8461]
OFC5     [2203.328, 744.39355]
OFZ0                   14.3037
OFZ1                   15.6023
OFZ2                    15.157
OFZ3                    16.269
OFZ4                   15.6213
OFZ5                    16.369
dtype: object
mean: 9.478686332702637 max: 53.9112548828125  min: 0.0 (2464, 3264, 2)
```



In [41]:
```
res.plot()
```

Out[41]: <AxesSubplot:>



In [40]:
```
cv2
```

Out[40]:
```
<module 'cv2' from '/Users/gerrie/opt/anaconda3/envs/OpenCV/lib/python3.8/site
-packages/cv2.cpython-38-darwin.so'>
```

In [42]:
```
of_res = res[['OFZ0','OFZ1', 'OFZ2', 'OFZ3', 'OFZ4', 'OFZ5']].copy()
of_res
```

Out[42]:

	OFZ0	OFZ1	OFZ2	OFZ3	OFZ4	OFZ5
23	15.405675	12.877311	15.484019	13.550748	16.173586	14.384924
24	14.931154	12.464343	14.843074	12.948333	15.046368	13.429861
25	16.515097	13.656087	15.541600	13.524177	15.666541	13.801317
26	17.218218	14.528187	17.305103	15.090350	17.459633	15.509411
27	17.515217	14.652087	17.244055	15.050405	17.560143	15.552138
...	...	...	...	...	...	...
105	12.210277	13.306490	12.315461	13.157657	12.263429	12.836651
106	13.792394	14.921235	13.731971	14.545489	13.644016	14.115857
107	14.787488	16.106730	15.450961	16.572363	15.372490	16.160898
108	15.117647	16.515930	15.233692	16.336452	15.438622	16.194305
109	14.303685	15.602341	15.157029	16.269026	15.621257	16.369022

87 rows × 6 columns

In [379…
```
of_res2 = res[['OFZ0','OFZ1', 'OFZ2', 'OFZ3', 'OFZ4', 'OFZ5']].copy()
of_res2 = of_res2.iloc[::2, :]
of_res2
```

Out[379…

	OFZ0	OFZ1	OFZ2	OFZ3	OFZ4	OFZ5
29	-1.572343	3.992443	2.253613	-5.242497	8.538836	6.089372
31	11.132250	10.085154	14.235498	15.013303	9.688616	12.526033
33	17.183636	14.958532	12.382684	11.699794	9.378431	13.335948
35	13.190943	12.243046	11.856373	11.586879	10.691122	10.609481
37	12.958664	12.125822	13.688688	13.071508	10.639080	12.939234
39	18.266813	16.041958	16.210045	14.999073	15.115362	14.636245
41	12.322905	11.246502	11.365714	10.463751	7.799138	8.062328
43	16.636375	14.752152	11.790757	11.047338	13.414128	12.345029
45	13.708220	12.669010	14.984334	14.311564	11.573172	11.003612
47	15.726617	14.133150	12.449100	11.737396	14.853395	14.053324
49	15.017895	14.755777	14.177994	13.813873	12.926779	12.504218
51	13.452872	12.891533	13.408951	12.797500	11.892874	12.749800
53	15.228659	13.895089	11.394537	10.994549	9.892563	10.382196
55	12.731632	12.076080	13.596768	13.101466	10.386911	10.647268
57	18.570820	17.638687	10.824599	10.521172	11.781981	11.743516
59	9.203575	8.870371	12.503448	13.243082	11.288908	11.041690
61	13.048148	12.783699	10.665875	11.186439	11.472002	12.191060
63	13.312933	13.621954	11.151202	11.102070	9.833710	10.367677
65	12.822662	12.817784	14.334427	14.279930	7.354418	12.789553
67	16.947319	16.596523	15.066135	14.985056	9.125352	14.960848
69	12.378387	12.442639	11.097144	11.452181	6.874989	9.119702
71	12.244410	14.454185	11.230292	11.574728	3.864800	12.561245
73	12.196362	12.917872	10.393580	11.912546	8.553345	10.366045
75	11.621029	12.699776	10.292237	12.700593	8.381921	12.715178
77	14.850699	15.094637	12.900728	13.284010	11.570292	12.279891
79	12.383931	12.642597	12.379601	12.500674	9.029530	10.022239
81	14.717395	14.816461	10.890416	11.097810	10.348811	12.303905

In [43]:
```
of_res.mean()
```

Out[43]:
```
OFZ0    14.581544
OFZ1    13.825301
OFZ2    14.696161
OFZ3    14.158553
OFZ4    14.913833
OFZ5    14.329033
dtype: float64
```

In [385…
```
of_res2.mean()
```

Out[385…
```
OFZ0    13.343808
OFZ1    13.083831
OFZ2    12.130546
OFZ3    11.823548
OFZ4    10.232239
OFZ5    11.642468
dtype: float64
```

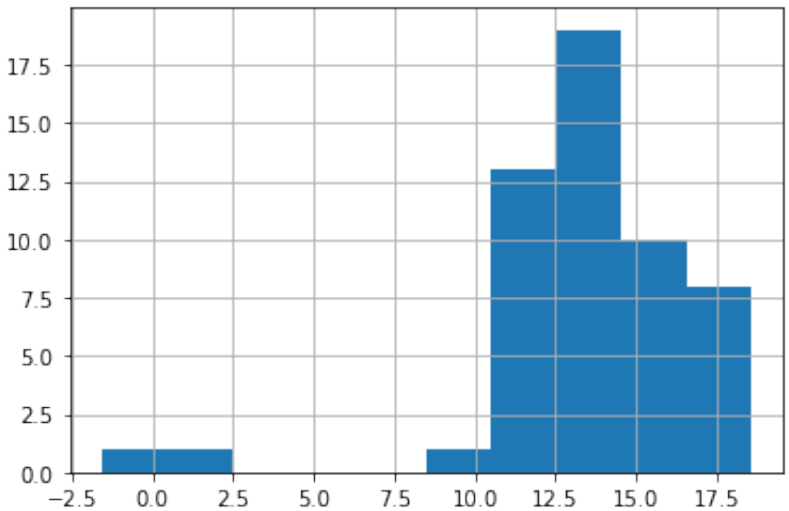
In [384…
```
of_res.std()
```

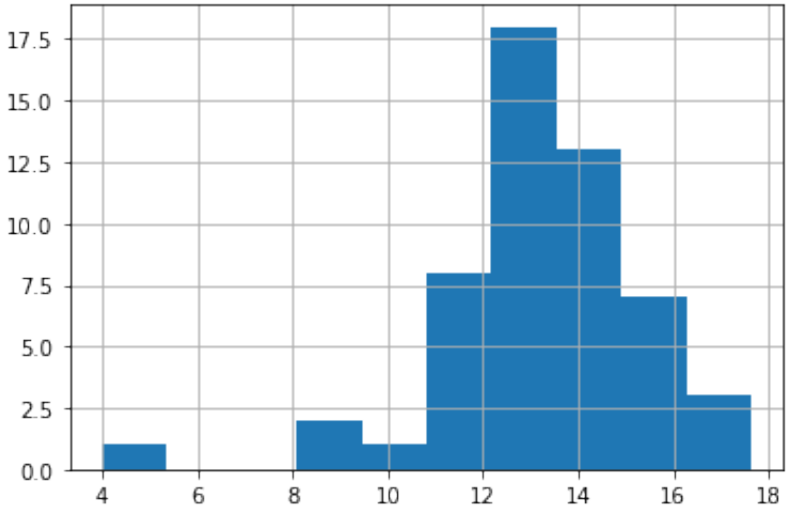
Out[384…
```
OFZ0    3.426351
OFZ1    2.205522
OFZ2    2.369845
OFZ3    3.500570
OFZ4    2.605586
OFZ5    1.767063
dtype: float64
```

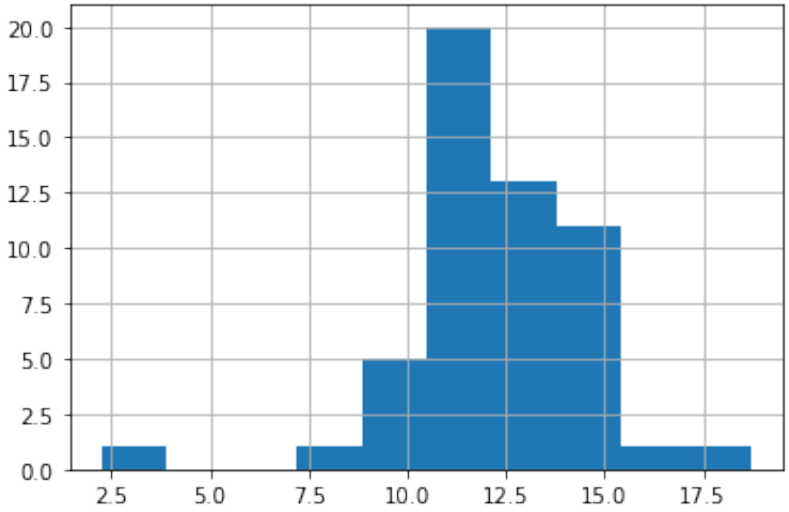
In [386…
```
of_res2.std()
```

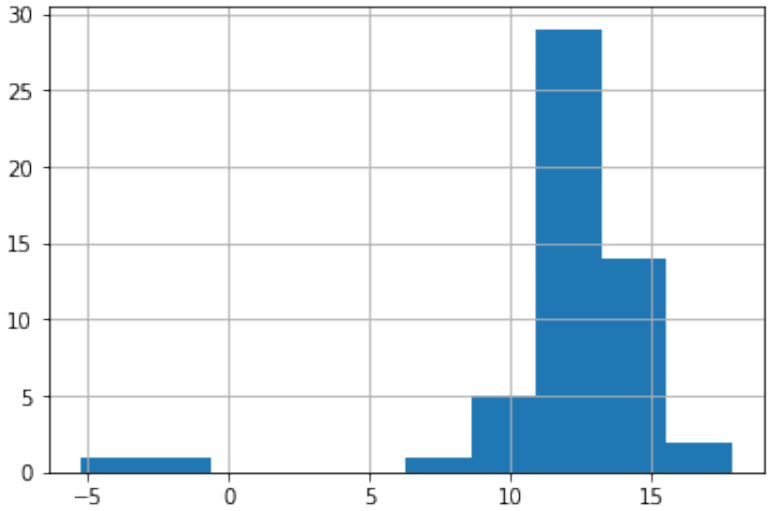
Out[386…
```
OFZ0    3.722503
OFZ1    2.619945
OFZ2    2.538541
OFZ3    3.688748
OFZ4    2.429765
OFZ5    1.944991
dtype: float64
```

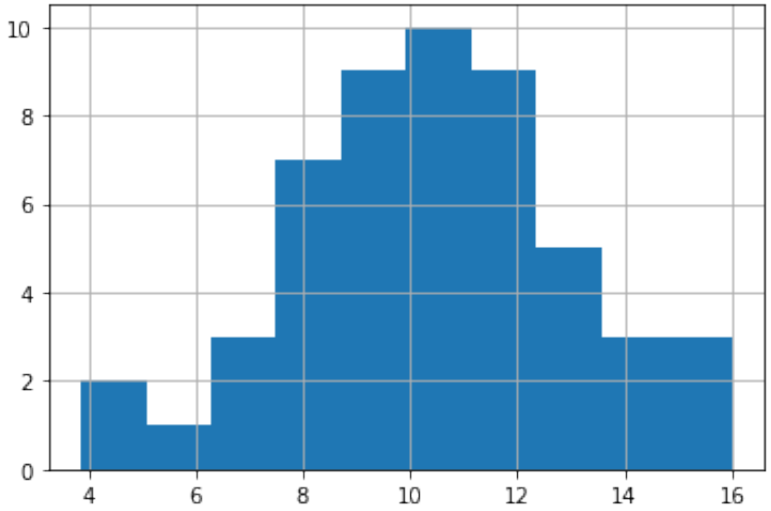
In [387…
```
for col in of_res.columns:
    res[col].hist()
    plt.title=col
    plt.show()
```

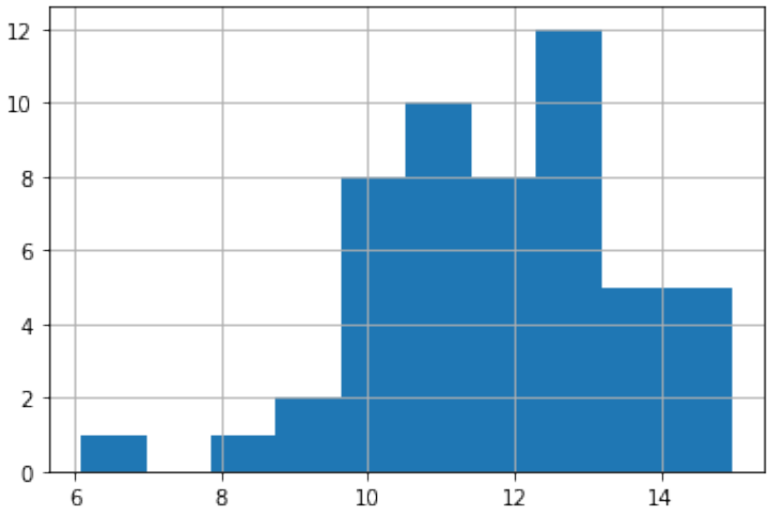



17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
4
6
8
10
12
14
16
18


20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

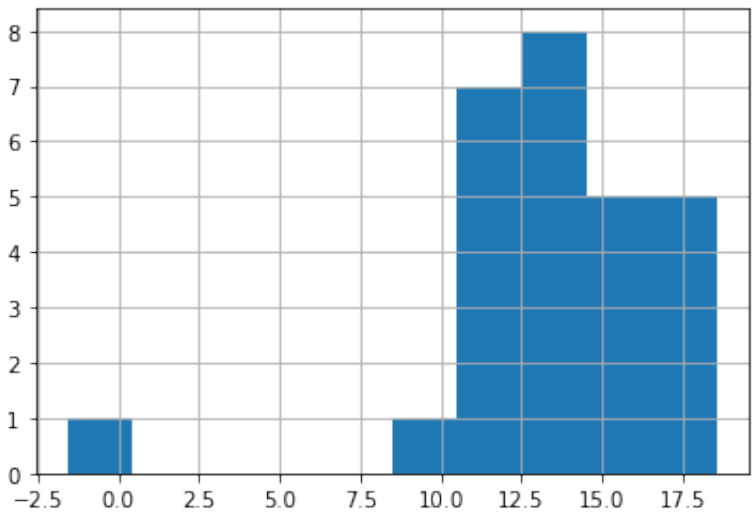

30
25
20
15
10
5
0
−5
0
5
10
15

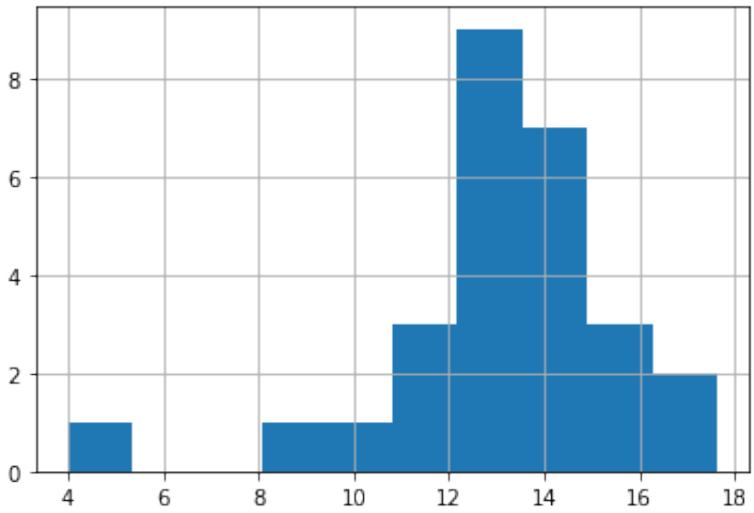


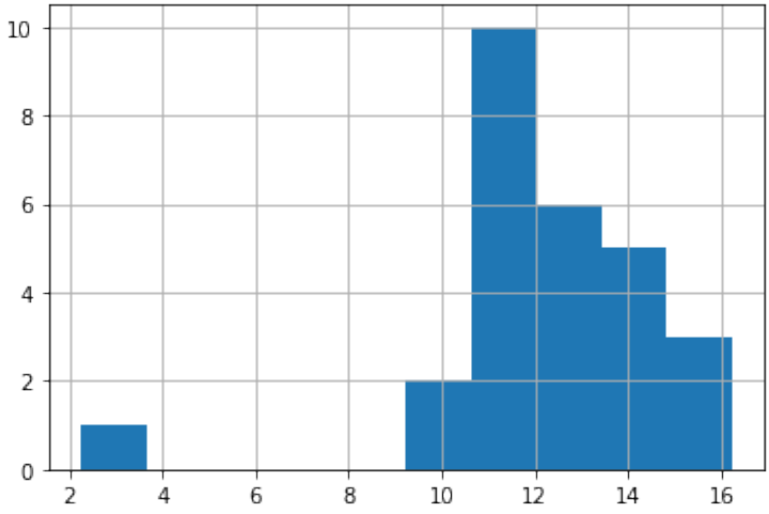


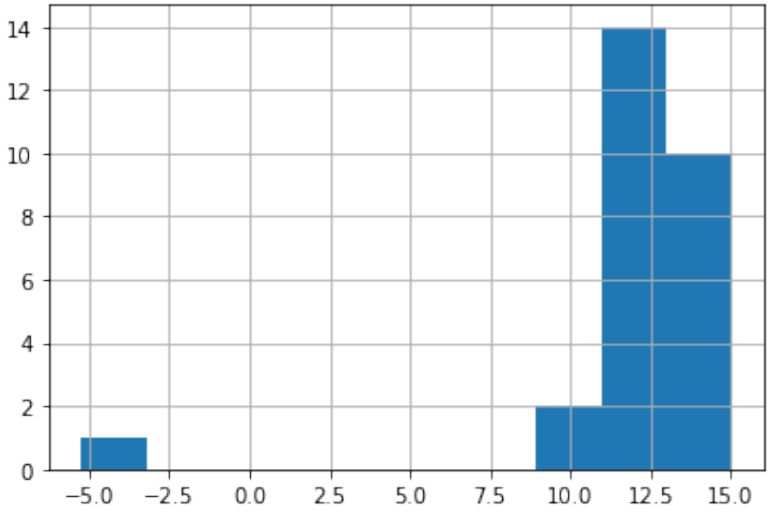
In [388…

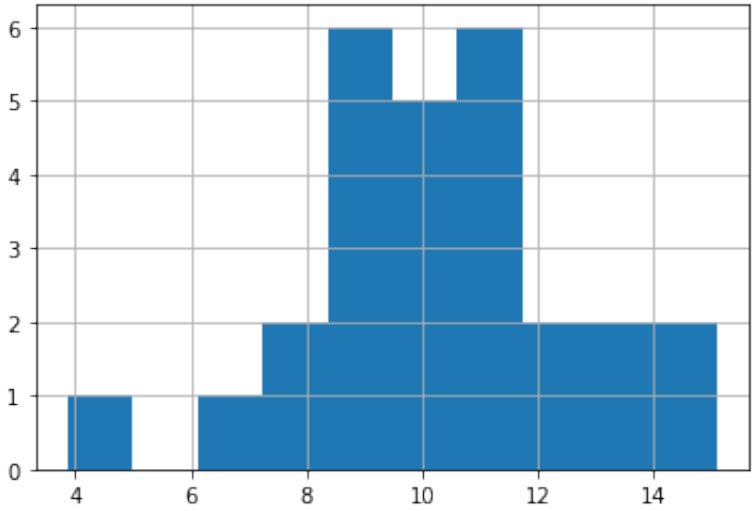
```
for col in of_res2.columns:
    of_res2[col].hist()
    plt.title=col
    plt.show()
```

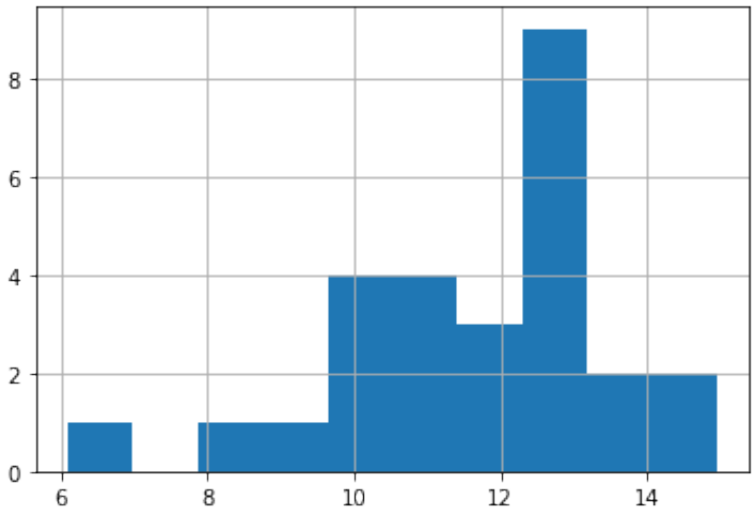



0
2
4
6
8
4
6
8
10
12
14
16
18


0
2
4
6
8
10
2
4
6
8
10
12
14
16


0
2
4
6
8
10
12
14
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0





In [389…

```
means = of_res.mean()
stds = of_res.std()

num_std_dev =1.0

for col in of_res.columns:
    of_res.loc[:,col] = of_res[col].apply(lambda x: x if ((x>means[col]-num_s
```

In [390…

```
means = of_res2.mean()
stds = of_res2.std()

num_std_dev =1.0

for col in of_res2.columns:
    of_res2.loc[:,col] = of_res2[col].apply(lambda x: x if ((x>means[col]-num
```

In [392…

```
of_res
```

Out[392…

	OFZ0	OFZ1	OFZ2	OFZ3	OFZ4	OFZ5
29	NaN	NaN	NaN	NaN	8.538836	NaN
30	NaN	NaN	NaN	NaN	NaN	NaN

31	11.132250	NaN	14.235498	15.013303	9.688616	12.526033
32	15.011619	13.945156	11.922591	11.140269	11.169121	10.735797
33	NaN	14.958532	12.382684	11.699794	9.378431	13.335948
34	13.328088	12.160350	NaN	13.914412	7.938611	11.987364
35	13.190943	12.243046	11.856373	11.586879	10.691122	10.609481
36	16.334723	14.469913	11.901287	11.096518	8.110641	12.026337
37	12.958664	12.125822	13.688688	13.071508	10.639080	12.939234
38	12.981710	11.842814	14.265706	13.094186	10.909926	10.608641
39	NaN	NaN	NaN	14.999073	NaN	NaN
40	NaN	15.049453	12.649611	12.006533	NaN	NaN
41	12.322905	11.246502	11.365714	10.463751	NaN	NaN
42	12.864378	12.183592	12.166539	11.749986	11.357794	10.809071
43	16.636375	14.752152	11.790757	11.047338	NaN	12.345029
44	13.366566	12.447551	14.296321	12.942886	NaN	12.636865
45	13.708220	12.669010	NaN	14.311564	11.573172	11.003612
46	NaN	NaN	12.755699	11.979865	12.188268	11.596287
47	15.726617	14.133150	12.449100	11.737396	NaN	NaN
48	NaN	NaN	NaN	NaN	NaN	NaN
49	15.017895	14.755777	14.177994	13.813873	12.926779	12.504218
50	13.900838	12.760178	NaN	8.971888	NaN	NaN
51	13.452872	12.891533	13.408951	12.797500	11.892874	12.749800
52	12.924383	12.143703	14.004603	13.569209	10.955235	10.753780
53	15.228659	13.895089	11.394537	10.994549	9.892563	10.382196
54	14.232697	13.360779	11.196365	10.891352	12.950160	12.739339
55	12.731632	12.076080	13.596768	13.101466	10.386911	10.647268
56	12.079449	12.414211	11.068219	10.966220	10.070384	NaN
57	NaN	NaN	10.824599	10.521172	11.781981	11.743516
58	14.482899	13.993057	NaN	14.592409	NaN	NaN
59	NaN	NaN	12.503448	13.243082	11.288908	11.041690
60	13.762835	13.159024	9.896024	9.745101	9.319455	NaN
61	13.048148	12.783699	10.665875	11.186439	11.472002	12.191060
62	12.276169	11.884491	13.478354	13.291267	9.728992	10.337687
63	13.312933	13.621954	11.151202	11.102070	9.833710	10.367677
64	15.545821	14.943596	11.505071	11.357318	9.928352	13.163181

65	12.822662	12.817784	14.334427	14.279930	NaN	12.789553
66	12.567348	12.599550	11.777807	12.895911	10.539652	10.958622
67	NaN	NaN	NaN	14.985056	9.125352	NaN
68	13.774661	14.569185	11.793566	13.517179	8.288504	NaN
69	12.378387	12.442639	11.097144	11.452181	NaN	NaN
70	10.721054	14.196287	11.491734	11.934621	NaN	10.928280
71	12.244410	14.454185	11.230292	11.574728	NaN	12.561245
72	12.133365	12.439136	12.896333	13.338473	NaN	11.688183
73	12.196362	12.917872	10.393580	11.912546	8.553345	10.366045
74	14.676450	14.619541	11.104069	11.308165	NaN	11.769105
75	11.621029	12.699776	10.292237	12.700593	8.381921	12.715178
76	16.048496	NaN	14.165811	NaN	9.352729	10.112720
77	14.850699	15.094637	12.900728	13.284010	11.570292	12.279891
78	11.857265	11.614022	NaN	NaN	12.812758	NaN
79	12.383931	12.642597	12.379601	12.500674	9.029530	10.022239
80	11.989946	12.878304	10.978407	11.650862	10.093853	10.386934
81	14.717395	14.816461	10.890416	11.097810	10.348811	12.303905

In [393…

```
of_res2
```

Out[393…

	OFZ0	OFZ1	OFZ2	OFZ3	OFZ4	OFZ5
29	NaN	NaN	NaN	NaN	8.538836	NaN
31	11.132250	NaN	14.235498	15.013303	9.688616	12.526033
33	NaN	14.958532	12.382684	11.699794	9.378431	13.335948
35	13.190943	12.243046	11.856373	11.586879	10.691122	10.609481
37	12.958664	12.125822	13.688688	13.071508	10.639080	12.939234
39	NaN	NaN	NaN	14.999073	NaN	NaN
41	12.322905	11.246502	11.365714	10.463751	NaN	NaN
43	16.636375	14.752152	11.790757	11.047338	NaN	12.345029
45	13.708220	12.669010	NaN	14.311564	11.573172	11.003612
47	15.726617	14.133150	12.449100	11.737396	NaN	NaN
49	15.017895	14.755777	14.177994	13.813873	NaN	12.504218
51	13.452872	12.891533	13.408951	12.797500	11.892874	12.749800
53	15.228659	13.895089	11.394537	10.994549	9.892563	10.382196
55	12.731632	12.076080	13.596768	13.101466	10.386911	10.647268
57	NaN	NaN	10.824599	10.521172	11.781981	11.743516
59	NaN	NaN	12.503448	13.243082	11.288908	11.041690
61	13.048148	12.783699	10.665875	11.186439	11.472002	12.191060
63	13.312933	13.621954	11.151202	11.102070	9.833710	10.367677
65	12.822662	12.817784	14.334427	14.279930	NaN	12.789553
67	16.947319	NaN	NaN	14.985056	9.125352	NaN
69	12.378387	12.442639	11.097144	11.452181	NaN	NaN
71	12.244410	14.454185	11.230292	11.574728	NaN	12.561245
73	12.196362	12.917872	10.393580	11.912546	8.553345	10.366045
75	11.621029	12.699776	10.292237	12.700593	8.381921	12.715178
77	14.850699	15.094637	12.900728	13.284010	11.570292	12.279891
79	12.383931	12.642597	12.379601	12.500674	9.029530	10.022239
81	14.717395	14.816461	10.890416	11.097810	10.348811	12.303905

In [394…

```
of_res.mean()
```

Out[394…
```
OFZ0    13.454506
OFZ1    13.249121
OFZ2    12.193598
OFZ3    12.300686
OFZ4    10.343478
OFZ5    11.596395
dtype: float64
```

In [395…
```
of_res2.mean()
```

Out[395…
```
OFZ0    13.574105
OFZ1    13.335157
OFZ2    12.130896
OFZ3    12.479934
OFZ4    10.214077
OFZ5    11.782134
dtype: float64
```