# Suppl. 1 Python code for flower tracking algorithm

```
# -*- coding: utf-8 -*-
"""
### Multi flower tracking algorithm ###
Created 2021

@author:
Hjalte M. R. Mann
TECOLOGY.xyz

Detections are in the format: x_min, y_min, x_max, y_max

Max distance: Threshold for maximum distance between point and track before initiation of a new track is forced
Max dissapeared: Max number of frames a track can be disappeared before the track is terminated and a new point in that area are considered new tracks
Running mean: Number of previous frame on which to calculate track position (to calculate distance between point and track). If set to one, position is centroid of previous track, if a

The centroid tracking approach was based on: https://pyimagesearch.com/2018/07/23/simple-object-tracking-with-opencv/ (Accessed on 2022-06-10)


### Pseudocode of tracking algorithm: ###

-Go to next frame
    -If the frame has no detections
        -Add one to disappeared counter for any existing tracks
    -If the frame has detections
        -If we are not tracking objects
            -Initiate tracks for all points
        -If we are tracking objects, calculate all pairwise distances (If running mean above 1, we use the mean x and y position of the track, else we just use position in the previous
            -For all distances below max distance threshold, try to associate points and tracks (shortest distance from point to object get associated and removed, then the second shor
            -For points with distances above max distance
                -Initiate new tracks
                -Add one to track id counter
        -For points that did not get associated to an exisiting track
            -Initiate new tracks
            -Add one to track id counter
        -Add frame tracking data to result dataframe for final output
        -Update running mean dictionary with new data
            -For tracks that are at the running mean threshold, remove oldest position and add current
        -For tracks that did not get a point associated to them, add one to disappeared counter
            -For tracks that exceed disappeared threshold, remove from active tracking storage
"""

# Import global packages
from collections import OrderedDict
import numpy as np
from scipy.spatial import distance as dist
import time
from statistics import mean
import sys

br = "\n" # Line break for use in code


# ==================== SETTINGS ============================================

#verbose = True # Set to True if you want tracking process printed to screen and False if not


# ==================== PROGRAM ============================================

class tracker():
    def __init__(self, max_gap, max_distance, running_mean_threshold, results_filename, detections, verbose):
        self.nextObjectID = 0 # Counter for object ids
        self.objects = OrderedDict() # Dictionary. objectID is the key, centroid is the content
        self.means = OrderedDict() # Dictionary to keep track of running means of object coordinates

        self.disappeared = OrderedDict() # Keeps track of how long an objectID has been lost

        self.detections = detections  # Store parameters for use in the class
        self.max_gap = max_gap
        self.max_distance = max_distance
        self.running_mean_threshold = running_mean_threshold
        self.results_filename = results_filename

        self.tracks = [] # Create a list for storing tracking results as we go

        self.verbose = verbose

        if self.running_mean_threshold == 0:
            print("Running mean set to zero. Please set to minimum 1.")
            print("Tracking aborted.")
            sys.exit()

        with open(self.results_filename, 'a') as resultFile: # Write the header of the output file
            header = 'frame,filename,x_min,x_max,y_min,y_max,x_c,y_c,objectID\n'
            resultFile.write(header)

        if verbose:
            print("Here are the detections",br,self.detections,br)

   ### Functions for tracking ###
    def store_tracking_results(self, frame, centroid, objectID):
        self.tracks.append([frame, centroid[0], centroid[1], objectID])
        if self.verbose:
            print(f'Object ID {objectID} with centroid {centroid} in frame {frame} stored.')

    def write_tracks_file(self): # Write tracking data to the final result file
        starttime = time.time()

        with open(self.results_filename, 'a') as resultFile:
            for t in self.tracks:
                frame, x_c, y_c, objectID = t[0], t[1], t[2], t[3]
                filename = self.detections.loc[self.detections['frame'] == frame, 'filename'].iloc[0]
                x_min = self.detections.loc[((self.detections['frame'] == frame) & (self.detections['x_c'] == x_c) & (self.detections['y_c'] == y_c)), 'x_min'].iloc[0]
                x_max = self.detections.loc[((self.detections['frame'] == frame) & (self.detections['x_c'] == x_c) & (self.detections['y_c'] == y_c)), 'x_max'].iloc[0]
                y_min = self.detections.loc[((self.detections['frame'] == frame) & (self.detections['x_c'] == x_c) & (self.detections['y_c'] == y_c)), 'y_min'].iloc[0]
                y_max = self.detections.loc[((self.detections['frame'] == frame) & (self.detections['x_c'] == x_c) & (self.detections['y_c'] == y_c)), 'y_max'].iloc[0]

                resultFile.write(f'{frame},{filename},{x_min},{x_max},{y_min},{y_max},{x_c},{y_c},{objectID}{br}')
        endtime = time.time()
        if self.verbose:
            print(f'Writing done. That took {round(endtime-starttime, 4)} seconds. {br}File saved as: {self.results_filename}{br}')

    def get_frame_detections(self, frame): # Get the detections from the current frame
        block = self.detections.loc[self.detections['frame'] == frame]
        frame_detections = block[["x_c", "y_c"]] # We just need the centroid, so we'll grab that and return it
        return frame_detections

    def register(self, frame, centroid): # Initiate a new track
        if self.verbose:
            print(f'Registering point with centroid {centroid} in frame {frame}')
```

```
        self.objects[self.nextObjectID] = [centroid] # Set the new centroid as content for the new objectID in the Objects dictionary
        self.means[self.nextObjectID] = centroid

        self.disappeared[self.nextObjectID] = 0 # Set number of times the new object has disappeared to zero.

        self.length_dict = {key: len(value) for key, value in self.objects.items()} # For storing how many frames a track has been tracked (to check against running mean setting)

        self.store_tracking_results(frame, centroid, self.nextObjectID)

        self.nextObjectID += 1 # Add 1 to the objectID counter so it's ready for the next point

        if self.verbose:
            print(f'Current objects: {br}{self.objects}')

    def deregister(self, objectID): # Deregister object by deleting it from the objects dict and removing the associated counter from the disappeared dict.
        del self.objects[objectID]
        del self.disappeared[objectID]
        del self.means[objectID]

    def update_object(self, objectID, centroid): # Updating the dictionary storing the object centroids

        if self.verbose:
            print(f'Received in update {br} Object id: {objectID} {br}Centroid: {centroid}{br}')

        if len(self.objects[objectID]) < self.running_mean_threshold:
            if self.verbose:
                print(f'Length ({len(self.objects[objectID])}) is less than running mean threshold ({self.running_mean_threshold})')
                print(f'Appending {[centroid]} to {self.objects[objectID]}')
            self.objects[objectID].append(centroid)

        if len(self.objects[objectID]) == self.running_mean_threshold:
            if self.verbose:
                print(f'Length {len(self.objects[objectID])} of {self.objects[objectID]} is equal to running mean threshold.')
                print(f'Deleting first item in {self.objects[objectID]} ({self.objects[objectID][0]}) and appending {centroid}')
            del self.objects[objectID][0]
            self.objects[objectID].append(centroid)

    def update_means(self): # Calculate the new means and store
        for key, value in self.objects.items():
            if len(value) > 1:
                c_m = [mean([i[0] for i in value]), mean([i[1] for i in value])]
            if len(value) == 1:
                c_m = value[0]
            self.means[key] = c_m
        if self.verbose:
            print(f'Updated means dictionary{br}Current mean dict:{br}{self.means}')


    ### Tracking algorithm ###
    def track(self): # Start tracking

        #####   If the data set is continous and may have frames without objects present, use code below.
        # If the frame has no detections
        #frameRange = list(range(frames[0], frames[len(frames) - 1]+1))
        #for frame in frameRange:
            #if self.verbose:
                #print("Getting detections for frame ", frame)
            #frame_detections = self.get_frame_detections(frame)#.dropna() # Get the detections for the current frame

            # if self.verbose:
            #     print(f'FRAME {frame}. Contains {len(frame_detections)} points.')

            # if frame_detections.empty: # we will add 1 to disappeared for all objects that are being tracked.
                #if self.verbose:
                #   print("Empty frame encountered. Adding 1 to disappeared for current objects.")
                # for objectID in list(self.disappeared.keys()): # loop over any existing tracked objects and mark them as +1 in disappeared
                #     if self.verbose:
                #       #print("Object id in disappeared: ", objectID)
                #     self.disappeared[objectID] += 1

                #     if self.disappeared[objectID] > self.max_gap: # Deregister points that have been disappeared longer than max disappeared threshold
                #         self.deregister(objectID)
                # continue
        #####
        #####

        #####
        frames = list(sorted(set(self.detections['frame'].to_list())))
        for frame in frames:
            if self.verbose:
                print("Getting detections for frame ", frame)
            frame_detections = self.get_frame_detections(frame)#.dropna() # Get the detections for the current frame

            if self.verbose:
                print(f'FRAME {frame}. Contains {len(frame_detections)} points.')
        #####


            # If the frame has detections
            inputCentroids = frame_detections[['x_c', 'y_c']].values.tolist() # we'll grab the centroid coordinates and convert to a list

            if self.verbose:
                print(f'Input centroids: {br}{inputCentroids}')

            if not self.objects: # if Objects is empty, we are currently not tracking any objects, so we'll take the input centroids and register each of them
                for i in range(0, len(inputCentroids)):
                    self.register(frame, inputCentroids[i])

                if self.verbose:
                    print("Not tracking objects. Initiated tracking on the current points")
                    print(f'Current objects:{br}{self.objects}')

            else: # We are already tracking objects, so let's see if we can associate any current frame detections with objects that are being tracked.
                objectIDs = list(self.means.keys()) # Store the object IDs and their centroids
                objectCentroids = list(self.means.values())

                if self.verbose:
                    print("Object IDs: ", objectIDs)
                    print("Object centroids: ", objectCentroids)
                D = dist.cdist(objectCentroids, inputCentroids) # Calculate distances between new points and existing tracks.
                if self.max_distance != 0: # If the max_distance has been set to 0, we'll ignore the next step. (Otherwise 0 would force new tracks for each point).
                    D[D > self.max_distance] = np.nan # Set the distance to NA for the pairs that have distance above the threshold we have set. This will force the initiation of new

                if self.verbose:
                    print("We are tracking existing objects.")
                    print(f'Current object ids: {objectIDs}')
                    print(f' Current object centroids:{br}{objectCentroids}')
                    print(f'Input centroids from current frame:{br}{inputCentroids}')
                    print(f'Here\'s the distance matrix:{br}{D}')
```

```
                objectIndexes = list(range(0,len(D))) # Grab a list of the object indexes (rows of the D matrix).
                inputIndexes = list(range(len(D[0]))) # Grab a list of the input indexes (columns of the D matrix).

                for c in range(len(D[0])): # Loop over the input centroids
                    if not np.isnan(D).all(): # Continue if there a still distance values left in the matrix (i.e. not all NA)
                        result = np.unravel_index(np.nanargmin(D, axis=None), D.shape) # Find the row,column index of the lowest distance in the matrix
                        D[result[0], :] = np.nan # Set the row and column for this element as NA (since the input and object has now been associated and cannot be used again)
                        D[:,result[1]] = np.nan

                        objectIndexes.remove(result[0]) # Remove the object index from the list since it has now been used
                        inputIndexes.remove(result[1]) # Remove the object index from the list since it has now been used

                        self.update_object(objectIDs[result[0]], inputCentroids[result[1]]) # Update the object with the new centroid coordinates
                        self.store_tracking_results(frame, inputCentroids[result[1]], objectIDs[result[0]]) # And store the tracking information

                    else: # All elements in the distance matrix are NA.
                        if self.verbose:
                            print("Association based on distance done. Now dealing with points that were not associated.")
                        pass

                self.update_means() # Update the dictionary containing the running means of the points

                for o in objectIndexes: # We'll add 1 for the objects that were not associated with a point in the current frame (in dictionary (disappeared) containing the number of
                    objectID = objectIDs[o]
                    self.disappeared[objectID] += 1
                    if self.disappeared[objectID] > self.max_gap: # Deregister any tracks that have been disappeared for more frames than the max disappeared threshold.
                        self.deregister(objectID)
                        if self.verbose:
                            print(f'Deregistering object {objectID}')

                for i in inputIndexes: # And we'll initiate new tracks for the points in the current frame that were not associated with an existing track.
                    if self.verbose:
                        print(f'Registering point {i} with the centroid {inputCentroids[i]}')
                    self.register(frame, inputCentroids[i])
        self.write_tracks_file()

        finalTracks = self.tracks

        return finalTracks

### END OF SCRIPT ###
```