

Welcome to Dilan's folder

Here, I am currently working on the OpenCV portion of the project

Hopefully, this personal log of occurrences will prove useful down the line as we work on the final report and the weeklies. *(Also this is just an excuse to learn Markdown xdd)*

Week of 4-28-2023

Tasks Completed

Finished serialization

5-2

Integration has begun

We've finally gone ahead and started integration of all our components together. There were hiccups as expected but by the end of the lab period, we managed to get the apriltag portion and the networking portion working together. Some other things that were addressed were the actual placement of the webcam, which will be on one of those pillar-looking things that hang from the ceiling.

For the apriltag integration, most of the issues arose from the directory needing to be changed and the formatting of the code itself, since the code for the apriltags was in a different block relative to where it was placed in `Server_Cmd_Center.py`.

For the networking integration, we started with just getting one robot to communicate with a base station. The base station was at first Nick's laptop, then from there, we moved to the Jetson. Using `ifconfig`, we got the jetson's IP address and used it in `Server_Cmd_Center.py`. Once we could verify that the networking, well, worked, we moved on to adding more robots to the network, which was also successful.

Another wrinkle that had to be ironed out was the fact that Python 3.8 does not support switch statements, which was the version that we have on the jetson. Since I had been using the Python that I had installed on my laptop, that version was 3.11, so it handled everything fine. Rather than going through and having to update every library to make sure they worked in tandem with 3.11, I simply re wrote the `match` statement to an `if-else` block. Same crap, different toilet.

5-1

Serialization is finished. Integration will start tomorrow.

```
mouse_tt = tags_oob(res_arr)
mice_tags = list(map(tuple, mice_tags))
corners = list(map(tuple, corners))
cv2.putText(ud_img, "{}".format(mouse_tt), (5, 350), cv2.FONT_HERSHEY_COMPLEX,
.5, (0, 0, 255), 1)
cv2.putText(ud_img, "{}".format(mice_tags), (5, 400), cv2.FONT_HERSHEY_COMPLEX,
.5, (0, 0, 255), 1)
cv2.putText(ud_img, "{}".format(corners), (5, 450), cv2.FONT_HERSHEY_COMPLEX,
```

```
.5, (0, 0, 255), 1)
fin_arr = [mouse_tt] + [mice_tags] + [corners]
print(fin_arr)
```

I made the `tags_oob` function in order to parse the projections into a more human-readable format

I've also decided to make `fin_arr` a list of lists. That way, each relevant array may be pulled with ease, i.e the array indicating whether a certain mouse is in bounds is at index 1 and so on.

```
def tags_oob(arr):
    tag00_i = [0, 3, 6, 9]
    tag01_i = [1, 4, 7, 10]
    tag02_i = [2, 5, 8, 11]

    res = [True, True, True]
    for x in range(len(arr)):
        # We have detected a tag to be outside of the arena, check for which tag
        # it is.
        if arr[x] < 0:
            # Check which index x corresponds to
            if x in tag00_i:
                res[0] = False
            if x in tag01_i:
                res[1] = False
            if x in tag02_i:
                res[2] = False
    return res
```

`tags_oob` will look through the projections array and look for a negative number. If a negative number is detected, we must identify which mouse it corresponds to, and change the array accordingly.

Tasks to be completed

5-1 Integration

So far, all of my stuff is in a while loop. Once we start integration, we're just gonna paste everything into one file since we don't have time to come up with a more sophisticated implementation. We can save some overhead by removing the drawing functions of the `apriltags.py` file.

5-2 Finish integration, begin testing

So far, it looks like, at the very least, we will have all components of our code working together in the same file. This coming Thursday, we will be meeting again to hopefully finish integration and begin playing around with the system, seeing how the different parts of our code interact with each other. Fingers crossed we don't run into any catastrophic failures or set backs **one week before the demo**. Right?

DCF

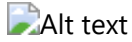
Week of 4-28-2023

Tasks Completed

Finished boundary detection

IT'S... ALIVE!!!

Boundary detection works for all mice in that may be within the bounds! I submitted and did a bit of hardcoding by way of a switch case, but other than that, it works as it should



The only issue now is that the display frame's frame rate plummets when there are three mice in the frame, but that should not matter too much since, in the end as I mentioned before, this display frame won't be used, as its simply for debugging purposes right now. Another thing that could be slowing things down is the use of print statements, which should be easily remedied.

```
# If there is an additional tag in the array of detected tags, we want to perform
boundary detection
# print("sorted_dict: ", sorted_dict)
if len(detect_keys) > 4:
    # Arrays to hold our stuff
    corners = []
    mice_tags = []
    b_arr = {}
    d_arr = []
    a_arr = []
    res_arr = []
    # Coordinates of center of tags
    # # corner_tags (List) -> [(x_coord, y_coord)]
    for x in corner_tags:
        corners.append(np.array([int(sorted_dict[x].center[0]),
int(sorted_dict[x].center[1]))])
    print("corners: ", corners)
    # Find difference from each pair of tags
    # Arrowhead - nock
    # d_arr (List) -> [(x, y)]
    for x in range(3):
        # print(corners[x+1][0] - corners[x][0], corners[x+1][1] - corners[x][1])
        d_arr.append(np.array((corners[x+1][0] - corners[x][0], corners[x+1][1] -
corners[x][1])))
        d_arr.append(np.array((corners[0][0] - corners[3][0], corners[0][1] -
corners[3][1])))
    print("d_arr: ", d_arr)
    # Rotation matrix
    rot = np.array([[0, -1], [1, 0]])
    # Take dot product of each difference and the rotation matrix to make the norm
    (?)
    for x in range(len(corner_tags)):
        a_arr.append(np.dot(rot, d_arr[x]))
    print("a_arr: ", a_arr)
    # print("")
```

```

# Coordinates of the mouse
for x in range(4, len(detect_keys)):
    mice_tags.append(np.array((int(sorted_dict[x].center[0]),
int(sorted_dict[x].center[1]))))
    print("mice_tags: ", mice_tags)
# Hardcoding cases for how many mice are detected instead of dynamically
adding them like a pleb :(
match len(mice_tags):
    case 1:
        b_arr.update({0:[np.array((mice_tags[0][0] - corners[0]
[0],mice_tags[0][1] - corners[0][1])),
np.array((mice_tags[0][0] - corners[1]
[0],mice_tags[0][1] - corners[1][1])),
np.array((mice_tags[0][0] - corners[2]
[0],mice_tags[0][1] - corners[2][1])),
np.array((mice_tags[0][0] - corners[3]
[0],mice_tags[0][1] - corners[3][1]))]})
    case 2:
        b_arr.update({0:[np.array((mice_tags[0][0] - corners[0]
[0],mice_tags[0][1] - corners[0][1])),
np.array((mice_tags[0][0] - corners[1]
[0],mice_tags[0][1] - corners[1][1])),
np.array((mice_tags[0][0] - corners[2]
[0],mice_tags[0][1] - corners[2][1])),
np.array((mice_tags[0][0] - corners[3]
[0],mice_tags[0][1] - corners[3][1]))]})

        b_arr.update({1:[np.array((mice_tags[1][0] - corners[0]
[0],mice_tags[1][1] - corners[0][1])),
np.array((mice_tags[1][0] - corners[1]
[0],mice_tags[1][1] - corners[1][1])),
np.array((mice_tags[1][0] - corners[2]
[0],mice_tags[1][1] - corners[2][1])),
np.array((mice_tags[1][0] - corners[3]
[0],mice_tags[1][1] - corners[3][1]))]})

    case 3:
        b_arr.update({0:[np.array((mice_tags[0][0] - corners[0]
[0],mice_tags[0][1] - corners[0][1])),
np.array((mice_tags[0][0] - corners[1]
[0],mice_tags[0][1] - corners[1][1])),
np.array((mice_tags[0][0] - corners[2]
[0],mice_tags[0][1] - corners[2][1])),
np.array((mice_tags[0][0] - corners[3]
[0],mice_tags[0][1] - corners[3][1]))]})

        b_arr.update({1:[np.array((mice_tags[1][0] - corners[0]
[0],mice_tags[1][1] - corners[0][1])),
np.array((mice_tags[1][0] - corners[1]
[0],mice_tags[1][1] - corners[1][1])),
np.array((mice_tags[1][0] - corners[2]
[0],mice_tags[1][1] - corners[2][1])),
np.array((mice_tags[1][0] - corners[3]
[0],mice_tags[1][1] - corners[3][1]))]})

```

```

        b_arr.update({2:[np.array((mice_tags[2][0] - corners[0]
[0],mice_tags[2][1] - corners[0][1])),
                        np.array((mice_tags[2][0] - corners[1]
[0],mice_tags[2][1] - corners[1][1])),
                        np.array((mice_tags[2][0] - corners[2]
[0],mice_tags[2][1] - corners[2][1])),
                        np.array((mice_tags[2][0] - corners[3]
[0],mice_tags[2][1] - corners[3][1]))]})
    case _:
        print("Mouse length array OOB")

    print("b_arr: ", b_arr)
    print("b_arr[0]: ", b_arr[0])
    # print("a_arr: ", a_arr)
    # Find the result from each dot product
    print("a_arr and b_arr (BEFORE): ", a_arr[0], b_arr[0][0])
    # a_arr = np.reshape(4,1)
    # b_arr = np.reshape(4,1)
    for x in range(len(a_arr)):
        for y in range(len(b_arr)):
            print("a_arr and b_arr(AFTER): ", a_arr[x], b_arr[y])
            res_arr.append(np.dot(a_arr[x], b_arr[y][x]))
    for x in range(0,len(res_arr)):
        print(res_arr[x])

```

It's a bit messy, but it takes the main points from the skeleton code I wrote last week and expands it for the purposes of supporting multiple mice that may occur on the field. We store the x and y coordinates of each corner tag's center into arrays, which are then stored into a list since lists are easier to iterate through

```

for x in corner_tags:
    corners.append(np.array([int(sorted_dict[x].center[0]),
int(sorted_dict[x].center[1]))))

```

The `for` loop iterates over the tag numbers specified in the array `corner_tags`. After that, we find the vectors between tags, and store then in a similar fashion; a List made of numpy arrays

```

# Find difference from each pair of tags
# Arrowhead - nock
# d_arr (List) -> [()]
for x in range(3):
    # print( corners[x+1][0] - corners[x][0],corners[x+1][1] - corners[x][1])
    d_arr.append(np.array((corners[x+1][0] - corners[x][0],corners[x+1][1] -
corners[x][1])))
d_arr.append(np.array((corners[0][0] - corners[3][0],corners[0][1] - corners[3]

```

```
[1]))))
```

The reason we only use the `for` loop for three vectors and not all 4 is that the last vector is the one from `tag03` to `tag00`, which could not be cleanly implemented. Again, `d_arr` is a List of numpy arrays.

```
# Rotation matrix
rot = np.array([[0, -1], [1, 0]])
# Take dot product of each difference and the rotation matrix to make the norm (?)
for x in range(len(corner_tags)):
    a_arr.append(np.dot(rot, d_arr[x]))
```

Here, we rotate the vector in order to get a vector that is normal to the original. This will be used to determine the projection of the mice onto the sides of the arena.

```
# Coordinates of the mouse
for x in range(4, len(detect_keys)):
    mice_tags.append(np.array((int(sorted_dict[x].center[0]),
int(sorted_dict[x].center[1]))))
```

`detect_keys` is an array that contains the ID numbers of the tags that were detected. Since we know the first four tags are for the corners, we start from tag 4 and work our way till the end of the list, appending these tags to a new list that will store the coordinates as arrays.

```
# Hardcoding cases for how many mice are detected instead of dynamically adding
them like a pleb :(
match len(mice_tags):
    case 1:
        b_arr.update({0:[np.array((mice_tags[0][0] - corners[0][0],mice_tags[0][1]
- corners[0][1])),
                        np.array((mice_tags[0][0] - corners[1][0],mice_tags[0][1]
- corners[1][1])),
                        np.array((mice_tags[0][0] - corners[2][0],mice_tags[0][1]
- corners[2][1])),
                        np.array((mice_tags[0][0] - corners[3][0],mice_tags[0][1]
- corners[3][1]))]))
    case 2:
        b_arr.update({0:[np.array((mice_tags[0][0] - corners[0][0],mice_tags[0][1]
- corners[0][1])),
                        np.array((mice_tags[0][0] - corners[1][0],mice_tags[0][1]
- corners[1][1])),
                        np.array((mice_tags[0][0] - corners[2][0],mice_tags[0][1]
- corners[2][1]))]))
```

```

- corners[2][1])),
                                np.array((mice_tags[0][0] - corners[3][0],mice_tags[0][1]
- corners[3][1]))))})

        b_arr.update({1:[np.array((mice_tags[1][0] - corners[0][0],mice_tags[1][1]
- corners[0][1])),
                                np.array((mice_tags[1][0] - corners[1][0],mice_tags[1][1]
- corners[1][1])),
                                np.array((mice_tags[1][0] - corners[2][0],mice_tags[1][1]
- corners[2][1])),
                                np.array((mice_tags[1][0] - corners[3][0],mice_tags[1][1]
- corners[3][1]))))})
        case 3:
            b_arr.update({0:[np.array((mice_tags[0][0] - corners[0][0],mice_tags[0][1]
- corners[0][1])),
                                np.array((mice_tags[0][0] - corners[1][0],mice_tags[0][1]
- corners[1][1])),
                                np.array((mice_tags[0][0] - corners[2][0],mice_tags[0][1]
- corners[2][1])),
                                np.array((mice_tags[0][0] - corners[3][0],mice_tags[0][1]
- corners[3][1]))))})

            b_arr.update({1:[np.array((mice_tags[1][0] - corners[0][0],mice_tags[1][1]
- corners[0][1])),
                                np.array((mice_tags[1][0] - corners[1][0],mice_tags[1][1]
- corners[1][1])),
                                np.array((mice_tags[1][0] - corners[2][0],mice_tags[1][1]
- corners[2][1])),
                                np.array((mice_tags[1][0] - corners[3][0],mice_tags[1][1]
- corners[3][1]))))})

            b_arr.update({2:[np.array((mice_tags[2][0] - corners[0][0],mice_tags[2][1]
- corners[0][1])),
                                np.array((mice_tags[2][0] - corners[1][0],mice_tags[2][1]
- corners[1][1])),
                                np.array((mice_tags[2][0] - corners[2][0],mice_tags[2][1]
- corners[2][1])),
                                np.array((mice_tags[2][0] - corners[3][0],mice_tags[2][1]
- corners[3][1]))))})
        case _:
            print("Mouse length array OOB")

```

Here's the nasty hardcoded part. Within the context of this project, there can be from 1 to 3 mice in the arena. We need to find the difference between each mice and each of the four corners in order to calculate the projections onto them. `b_arr` is a dictionary, meaning it can be indexed by something other than an integer. I did this mostly for organization, but looking at it now, it may have been simpler to store everything into an array. Oh well.

```

for x in range(len(a_arr)):

```

```
for y in range(len(b_arr)):
    print("a_arr and b_arr(AFTER): ", a_arr[x], b_arr[y])
    res_arr.append(np.dot(a_arr[x], b_arr[y][x]))
for x in range(0, len(res_arr)):
    print(res_arr[x])
```

Finally, here we are finding the projection of the mice tags onto the vectors. If a value comes back positive, we know that the tag is within the vector space, i.e. the arena. If a value comes back negative, we know that it is outside of the vector space. Currently, with three mice, we get the following output from this snippet:

```
...
14349
13522
5074
23803
9365
14965
15748
16169
25321
5702
20546
14242
```

We get back the projections of the mice tags on the vectors, but all we really care about is the sign of the numbers. For example, when all the mice are out of the arena:



We get the following output:

```
...
10828
18135
3938
35590
36201
35537
13948
6768
20732
-12017
-12755
```


-11858

As we can see, the output of the program yields 12 total values, each corresponding to the projections as mentioned before. It seems to be in order, vector-wise then mouse-wise. There are four groups of three numbers, each number within the groups represents the first, second, and third mouse respectively. This can be done to pinpoint which mouse is currently out of the bounds.

The main headache was just getting around how numpy arrays behave compared to Python lists. Conceptually they are the same, however the functions associated with them are distinct.

Tasks to be completed

Serialize output

Now that this is out of the way, I can focus on serializing the output for my teammates to use. What I have in mind is the following:

```
[Loc. tag00, Loc. tag01, Loc. tag02, Loc. tag03, Loc. mouse00, Loc. mouse01, Loc.
mouse02, ...
| Tuple | Tuple | Tuple | Tuple | Tuple | Tuple |
Tuple |

... mouse00 OOB, mouse01 OOB, mouse01 OOB]
| Boolean | Boolean | Boolean |
```

From here, it would simply be up to the user to parse the information. If we had more time, I would have liked to serialize this as an object, thereby adding functions to make it easier to parse the information. I am tempted to try it, but I feel like that would just lead me down another rabbithole of Geeks4Geeks and documentation that hasn't been updated since Bush was in office. Once serialization is done, I will be able to help with any problems that will inevitably arise from implementing this into the other portions of the project.

DCF

Week of 4-21-2023

Tasks Completed

- Gained comprehension of boundary detection
- Started implementation of boundary detection

4-18

As expected, implementation is proving to be a pain. However, I did manage to get it working for a single point within the bounaries set by the four corners.

```
import numpy as np

# Coordinates of tags
tag0 = np.array([5,20])
tag1 = np.array([20,20])
tag2 = np.array([20,5])
tag3 = np.array([5,5])

# Coordinates of the mouse
mouse = np.array([0,10])

# Find difference from each pair of tags
# Arrowhead - nock
da = tag1 - tag0
da = np.transpose(da)

db = tag2 - tag1
db = np.transpose(db)

dc = tag3 - tag2
dc = np.transpose(dc)

dd = tag0 - tag3
dd = np.transpose(dd)

# Rotation matrix
rot = np.array([[0,-1],[1,0]])

# Take dot product of each difference and the rotation matrix to make the norm (?)
aa = np.dot(rot,da)
print(aa)

ab = np.dot(rot,db)
print(ab)

ac = np.dot(rot,dc)
print(ac)

ad = np.dot(rot,dd)
print(ad)

# Take difference between mouse and each tag
ba = mouse - tag0

bb = mouse - tag1

bc = mouse - tag2

bd = mouse - tag3

# Find the result from each dot product
res0 = np.dot(aa,ba)
```

```
res1 = np.dot(ab,bb)

res2 = np.dot(ac,bc)

res3 = np.dot(ad,bd)

# Print results (Should all have the same sign, I think negative)
print(res0)
print(res1)
print(res2)
print(res3)
```

Output:

```
dilancf@DESKTOP-PIM6G3Q:~$ python3 norms.py
[ 0 15]
[15  0]
[ 0 -15]
[-15  0]
-150
-150
-75
-75
```

All the results came out to have the same sign (in this case, negative), which is good. This means that, relative to all the lines making up the boundaries, the point is outside of their plane. If even one of them had a different sign, we would know that the point was no longer inside of the arena, and was instead on the other side of one of the lines. Knowing this, we can act accordingly by sending a signal/ flag to the movement part of our system. I did manage to implement this for one mouse tag, and it worked as expected. The only thing that was different from the snippet above was that the inside was positive instead of negative. Unfortunately, I didn't manage to get a picture of it working at this stage before moving on. Currently, I am trying to get boundary detection working for all mice. This has led to many `for` loops, since I need to take into account the possibility that mice may come into and out of frame as the program goes on. I almost managed to get it completely working, I'm just getting an out of bounds error currently:

```
if len(detect_keys) > 4:
    # Arrays to hold our stuff
    corners = []
    mice_tags = []
    b_arr = []
    d_arr = []
    a_arr = []
    res_arr = []
    # Coordinates of tags
    for x in range(len(corner_tags)):
        corners.append(np.array([int(sorted_dict[x].center[0]),
int(sorted_dict[x].center[1]))])
```

```

# Find difference from each pair of tags
# Arrowhead - nock
for x in range(3):
    d_arr.append(corners[x+1] - corners[x])
    d_arr.append(corners[0] - corners[3])

# Rotation matrix
rot = np.array([[0, -1], [1, 0]])

# Take dot product of each difference and the rotation matrix to make the norm
(?)
for x in range(len(corner_tags)):
    a_arr.append(np.dot(rot, d_arr[x]))

    print("")
    #print(a_arr)

# Coordinates of the mouse
for x in range(4, len(detect_keys)):
    mice_tags.append(np.array([int(sorted_dict[x].center[0]),
int(sorted_dict[x].center[1]))])

# Take difference between mouse and each tag
for x in range(len(mice_tags)):
    b_arr.append([mice_tags[x] - corners[0], mice_tags[x] - corners[1],
mice_tags[x] - corners[2], mice_tags[x] - corners[3]])

print(b_arr)

# Find the result from each dot product
for x in range(len(a_arr)):
    for y in range(len(b_arr)):
        res_arr.append(np.dot(a_arr[x], b_arr[x][y]))

for x in len(res_arr):
    print(res_arr[x])

```

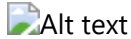
I'm 99% sure that the bug has to do with how im storing the differences between each mouse and each of the four points of each line in `b_arr`, since it doesnt make much sense to store them that way. I think this may be a simple fix, since it's just a matter of sitting down, looking at the code, think of a better implementation, berate myself for the previous implementation, and fix the problem. Jokes aside, once this is done, the last part would be to package this information into a format that can be used by the rest of the group for whatever needs they may have.

4-17

So far, there isn't much to say in the way of boundary detection except that I finally understand how matrices work. It all essentially boils down to the following equation:

$$a \cdot b = |a||b|\cos(\theta)$$

Where $|a||b|$ is essentially a scalar, which does not hold as much relevance for our purposes. What we really care is the $\cos(\theta)$ part.



Essentially, the cosine graph remains positive while it is less than 90 degrees, negative otherwise, and this repeats infinitely. The nice part about this is that it will work no matter where in space the points are. A few things that need to be attended to are

1. The norm of the line that will be our boundary
2. The vector from a point on the line to our mouse

Knowing these things, we will be able to take the dot product and, based on how wide the angle is between the vectors, we will know where the mouse tag is.



Here, the green circle represents the position of the mouse in space, the blue line is the boundary we are taking into account, the lime arrow is the vector of the norm, and finally the pink arrow is the vector of the mouse from the line. As we can see, the norm is at a 90 degree angle with the line. If the angle from the norm (lime arrow) to the vector of the mouse (pink arrow) is greater than 90, we understand that it will be on the opposite side of the norm, whereas if the angle is less than 90, we can interpret that to mean that the mouse is on the same side as the norm. Depending on how we define the boundaries, we can simply look at the sign of the result and determine on which side we are on.

The fine details were a bit hard to follow since I haven't studied linear algebra in a brick, so let's break it down step by step:

First, let's look back at equation (1):

$$a \cdot b = |a||b|\cos(\theta)$$

The point (pun intended) of having the $\cos(\theta)$ in the equivalence is so that we understand where the negative is coming from, and how it allows us to make the distinction between "In" and "Out". " a " is the norm of the line, which may be found by the following equation:

$$a = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \cdot (d_1 - d_0)$$

Where $(d_1 - d_0)$ is nothing more than the difference between the two endpoints that make up the boundary line. This will yield a 2×1 array which we will need to transpose in order to dot it with the rotation matrix. The rotation matrix works by correlating the input y with the output x and the input x with the output y by their respective values. In this case, this will yield a 90 degree rotation counter clockwise. Depending on which line we are observing, this rotation matrix will need to be altered.

Finally, the b vector is nothing more than the vector from a point on the line to the mouse's position, and is given by

$$b = p_1 - p_0$$

Where p_0 is a point on the boundary line and p_1 is the mouse's coordinates. Note that it's crucial to subtract the "Arrowhead" by the "Tail" (or "Nock" if you want to be precious). Note that p_0 could also

very well be either d_0 or d_1 . For our purposes, we have done this just to keep things simple. Finally, we dot both a and b as shown in equation (1).

Another neat thing about this method is, due to the equivalence from equation (1), we do not need to calculate $\cos(\theta)$ every time. This will save us some computing power, though its not as crucial in our case since our application is so small.

So far, I have made a simple program that simulates this with a line and a dot on the same coordinate space.

```
import numpy as np

# Coordinates for the line
endpoint_0 = np.array([15,5])
endpoint_1 = np.array([5,20])

# Coordinates of the mouse
mouse = np.array([5,5])

# Find difference from both endpoints for the equation
# Arrowhead - nock
d = endpoint_0 - endpoint_1
d = np.transpose(d)

# Rotation matrix
rot = np.array([[0,-1],[1,0]])

# Take dot product
a = np.dot(rot,d)
print(a)

# Arrowhead - nock
b = mouse - endpoint_0

res = np.dot(a,b)
print(res)
```

Simulating this, it works as intended, however the norm is not what I expected. I think it may have to do with my rotation matrix, which is something I'll have to iron out tomorrow.

The motivation behind this is that we may have some rotation in the camera, which would render the "Compare x and y values of coordinates" useless, since that relies on the points being in an xy plane. This method, however, works regardless of axes, and it's just plain ✨ ELEGANT ✨

Tasks to be completed

4-18: Finish implementation of boundary detection into *apriltag.py* and start packaging for the rest of the team to use This part will be tricky, since we want to send only the most relevant information over the Wi-Fi in order to prevent lag issues, or information taking too long to be sent and processed for the chasing to occur. Once we can confirm that the information regarding the locations and boundary detections of the mice can be communicated and recieved, then we can focus on how to use the information best.

4-17: Implement boundary detection into *apriltag.py*

Since we now know better how to implement the boundary detection on a mathematical level, we can implement it hopefully without much headache, but thinking implementation will be painless is like thinking Evil Geniuses are still a T1 CSGO team: You're either huffing massive amounts of copium, delusional, or both (probably both).

Also, apparently I've been understanding arrays in *Numpy* incorrectly, but it doesn't seem to matter because I still got an expected value. :shrug:

DCF

Week of 4-14-2023

Tasks Completed

Midpoints are drawn

4-11

Relearning algebra 😊

Yeah... that's not going so well

Right now, I'm having trouble finding the correct midpoint for the normal vector. The equation for finding the midpoint between two points is usually:

$$M(\frac{x_1 + x_2}{2}, \frac{y_1 + y_2}{2})$$

But since the coordinate plane that we're using is kinda flipped, with the top left corner being the minimum, and the bottom right being the maximum, I think that it's throwing things off. So far, my code for finding the midpoint (and other necessary variables) looks as the following:

```
# Finding slopes for each line between points
# tag00 -> tag01
m0 = (int(sorted_dict[1].center[1]) -
int(sorted_dict[0].center[1]))/(int(sorted_dict[1].center[0]) -
int(sorted_dict[0].center[0]))
# tag01 -> tag02
m1 = (int(sorted_dict[2].center[1]) -
int(sorted_dict[1].center[1]))/(int(sorted_dict[2].center[0]) -
int(sorted_dict[1].center[0]))
# tag02 -> tag03
m2 = (int(sorted_dict[3].center[1]) -
int(sorted_dict[2].center[1]))/(int(sorted_dict[3].center[0]) -
int(sorted_dict[2].center[0]))
# tag03 -> tag00
m3 = (int(sorted_dict[0].center[1]) -
int(sorted_dict[3].center[1]))/(int(sorted_dict[0].center[0]) -
int(sorted_dict[3].center[0]))
```

```

# Finding the y-intercept for each line
# We can do this by using one of our x y coords
#  $y = mx + b$ 
#  $b_i = y_i - m_i(x_i)$ 
# tag 00 -> tag01
b0 = int(sorted_dict[1].center[1]) - (m0 * int(sorted_dict[1].center[0]))
# tag 01 -> tag02
b1 = int(sorted_dict[2].center[1]) - (m1 * int(sorted_dict[2].center[0]))
# tag 02 -> tag03
b2 = int(sorted_dict[3].center[1]) - (m2 * int(sorted_dict[3].center[0]))
# tag 03 -> tag00
b3 = int(sorted_dict[0].center[1]) - (m3 * int(sorted_dict[0].center[0]))

# Finally, we need the midpoints in order to find the normal vector
mid0 = (int(sorted_dict[0].center[0] + sorted_dict[1].center[0]/2),
int(sorted_dict[0].center[1] + sorted_dict[1].center[1]/2))
mid1 = (int(sorted_dict[1].center[0] + sorted_dict[2].center[0]/2),
int(sorted_dict[1].center[1] + sorted_dict[2].center[1]/2))
mid2 = (int(sorted_dict[2].center[0] + sorted_dict[3].center[0]/2),
int(sorted_dict[2].center[1] + sorted_dict[3].center[1]/2))
mid3 = (int(sorted_dict[3].center[0] + sorted_dict[0].center[0]/2),
int(sorted_dict[3].center[1] + sorted_dict[0].center[1]/2))

tag00 = (int(sorted_dict[0].center[0]), int(sorted_dict[0].center[1]))
tag01 = (int(sorted_dict[1].center[0]), int(sorted_dict[1].center[1]))
tag02 = (int(sorted_dict[2].center[0]), int(sorted_dict[2].center[1]))
tag03 = (int(sorted_dict[3].center[0]), int(sorted_dict[3].center[1]))

```

Note that as of now, the `tagxx` variables are just used for debugging, as I want to make sure that I am doing the math correctly. By looking at the output, I apparently am not.

```

Tag coord
(347, 169)
(525, 157)
(544, 369)
(345, 364)
Midpoints
(610, 247)
(797, 341)
(716, 551)
(519, 448)

```

!! ALERT! I AM AN IDIOT !!

```

mid0 = (int(sorted_dict[0].center[0] + sorted_dict[1].center[0]/2),
int(sorted_dict[0].center[1] + sorted_dict[1].center[1]/2))
mid1 = (int(sorted_dict[1].center[0] + sorted_dict[2].center[0]/2),

```



```
int(sorted_dict[1].center[1] + sorted_dict[2].center[1]/2))
mid2 = (int(sorted_dict[2].center[0] + sorted_dict[3].center[0]/2),
int(sorted_dict[2].center[1] + sorted_dict[3].center[1]/2))
mid3 = (int(sorted_dict[3].center[0] + sorted_dict[0].center[0]/2),
int(sorted_dict[3].center[1] + sorted_dict[0].center[1]/2))
```

See what's missing from this picture? I'll give you a hint: it rhymes with "harentheses"

```
mid0 = (int((sorted_dict[0].center[0] + sorted_dict[1].center[0])/2),
int((sorted_dict[0].center[1] + sorted_dict[1].center[1])/2))
mid1 = (int((sorted_dict[1].center[0] + sorted_dict[2].center[0])/2),
int((sorted_dict[1].center[1] + sorted_dict[2].center[1])/2))
mid2 = (int((sorted_dict[2].center[0] + sorted_dict[3].center[0])/2),
int((sorted_dict[2].center[1] + sorted_dict[3].center[1])/2))
mid3 = (int((sorted_dict[3].center[0] + sorted_dict[0].center[0])/2),
int((sorted_dict[3].center[1] + sorted_dict[0].center[1])/2))
```

Now, with the parentheses around the quantity of the two coordinates, the computer draws the midpoints accurately

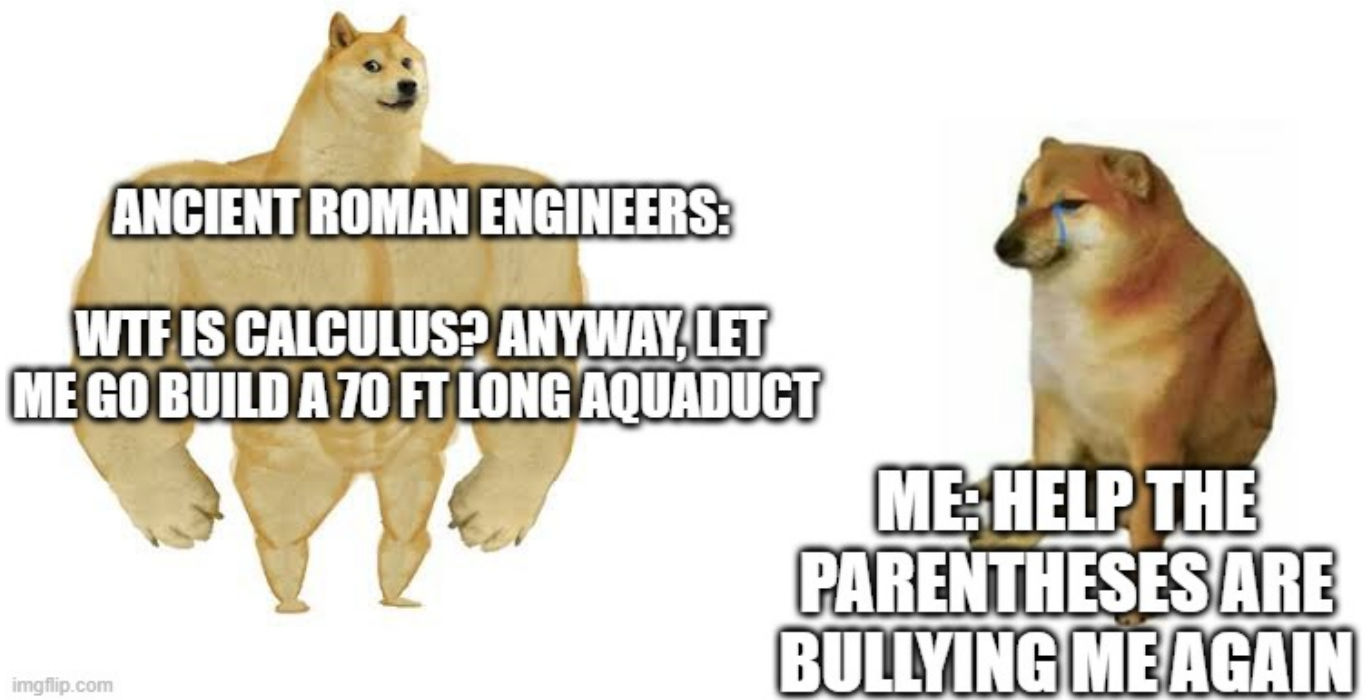
 Alt text

-1 hour 

~~Now, with what time I have left, I'll try drawing the vectors for the vector space. Once this math is done, I can use it to detect when a tag is outside of this vector space.~~

Nevermind, turns out what I had to do was even more straightforward than that. Apparently, all I need to do is take the dot product of the mouse tuple and the midpoint, which will tell me implicitly on which side of the norm the mouse is on.

Additionally, I will sometimes get a **Divide by zero** error. I expected that, since the math is slapped together pretty haphazardly. So I think it can be solved pretty easily as long as it doesn't involve **parentheses**.



4-10

Relearning algebra 😊

So far, I only have a skeleton of an implementation. I need to find the functions of all the lines and draw a vector onto the screen based on that.

Tasks to be completed

~~4-10 plot normal vectors in order to implement boundary detection~~

As of now, Levi has suggested a way to implement boundary detection by using vectors and vector spaces. In short, if a tag is detected outside of a given vector space, we should be able to tell and send some kind of signal in order to correct the mouse.

4-11 Implement boundary detection by way of projections

Now that I have a clearer picture of how to approach this problem, I think I'll have more success in implementation. The most important part is defining what we consider "In" and what we consider "Out", as well as how to best offload this information to the different parts of the project that need it. Nick mentioned that he needed some information from the tags for the evasion algorithm, so I need to make sure that whatever `apriltag.py` returns, it will be in a succinct and readable manner for him/ anyone else who uses this code. You know what they say:

A programmer is only as good as their documentation
-They

I really want to get this working by the end of the week, mostly to offset lost time from today, and hopefully start integration hopefully next week, week after next at the latest.

DCF


Week of 4-7-2023

Tasks completed

Started work on the arena drawing

4-4

I've implemented the code in order to draw the arena on the frame. As of writing, it seems to work for a split second before committing sudoku.

 Alt text

```
for res in results:
    for x in range(len(results)):
        if x not in detect_arr:
            detect_arr.add(results[x].tag_id)

    print(detect_arr)
    if corner_tags.intersection(detect_arr) == corner_tags:
        cv2.line(ud_img, (int(results[0].center[0]),
int(results[0].center[1])), (int(results[1].center[0]),
int(results[1].center[1])), color=(0, 255, 0), thickness=5)
        cv2.line(ud_img, (int(results[1].center[0]),
int(results[1].center[1])), (int(results[2].center[0]),
int(results[2].center[1])), color=(0, 255, 0), thickness=5)
        cv2.line(ud_img, (int(results[2].center[0]),
int(results[2].center[1])), (int(results[3].center[0]),
int(results[3].center[1])), color=(0, 255, 0), thickness=5)
        cv2.line(ud_img, (int(results[3].center[0]),
int(results[3].center[1])), (int(results[0].center[0]),
int(results[0].center[1])), color=(0, 255, 0), thickness=5)
```

This is the code that's causing me to age faster. The problem is that `results` is dynamic, and therefore changes depending on how many detections there are. I'm pretty sure I need to adjust the way detections are added and removed to `detect_arr`, that it, to actually remove tags that are no longer in frame / detected.

Update

It's finished! With a nudge from Levi, I managed to implement the tag storage as a ~~hash~~ dictionary (sorry, the Ruby programmer in me made an unexpected appearance). Here's how I managed it

```
corner_tags = [0,1,2,3]
while True:
    ...
    try:
        detect_arr = dict()
        ...
        for res in results:
            # For however many detections there are, we add the detection
            # into the 'detect_arr' dictionary if it is one of the pre-determined
```

```

# tag numbers for the corners.
for x in range(len(results)):
    if x in corner_tags:
        detect_arr.update({results[x].tag_id: results[x]})
# Turn the keys of the dictionary into a list so we can sort the
dictionary
detect_keys = list(detect_arr.keys())
detect_keys.sort()
# Using a codeblock, we can iterate through the array and change
# the indices so that the detections are ordered by the tags.
sorted_dict = {i: detect_arr[i] for i in detect_keys}
print(detect_keys)

# Since we have sorted our dictionary, we can go ahead and draw the
rectangle
# confident that the indicies shown below will be accurate.
if set(detect_keys) & set(corner_tags) == set(corner_tags):
    cv2.line(ud_img, (int(sorted_dict[0].center[0]),
int(sorted_dict[0].center[1])), (int(sorted_dict[1].center[0]),
int(sorted_dict[1].center[1])), color=(0, 255, 0), thickness=5)
    cv2.line(ud_img, (int(sorted_dict[1].center[0]),
int(sorted_dict[1].center[1])), (int(sorted_dict[2].center[0]),
int(sorted_dict[2].center[1])), color=(0, 255, 0), thickness=5)
    cv2.line(ud_img, (int(sorted_dict[2].center[0]),
int(sorted_dict[2].center[1])), (int(sorted_dict[3].center[0]),
int(sorted_dict[3].center[1])), color=(0, 255, 0), thickness=5)
    cv2.line(ud_img, (int(sorted_dict[3].center[0]),
int(sorted_dict[3].center[1])), (int(sorted_dict[0].center[0]),
int(sorted_dict[0].center[1])), color=(0, 255, 0), thickness=5)

```

As of right **now**, this will only work with the tags that are in `corner_tags`. Any tag that is not a part of `corners_tags` will not be added to `detect_arr`. I want to change this, since we will need to hold the information from the tags on the mice. That can be implemented later. For now, the camera can now detect the four tags and draw the outline of the arena comfortably.

The main problem was that `detect_arr` was holding stale data from the previous instance. When a tag was no longer detected, the above code would try to access `result[missing index]`, and thus throw a fit. By placing `detect_arr = dict()` within the while loop, the dictionary is cleared for the new data being processed.

4-3

So far, we've gotten the webcam calibrated and undistorted (see week 3-31). Today, we start work on the printing of the detections, consequently drawing the box of the arena on the screen. Last week, we managed to print out the coordinates of the tags. Today, we want to use these detections and distinguish between the different tags detected. This is proving difficult because ~~I am a dumbass and~~ the documentation is ~~ass~~ fussy. So far, this is my approach to tag distinction:

```

for res in results:
    for x in range(len(results)):

```

```
if x not in detect_arr:
    detect_arr.append(results[x].tag_id)
print(detect_arr)
```

It bugs me because this isn't the most elegant way to do this, and it's also still not working as intended. When I detect two tags, it works fine-ish, but since I'm currently using my laptop screen to display the tags, sometimes the camera picks up the smaller tags that show up in the cv frame, and it freaks out and starts appending more and more to `detect_arr`.

[0, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

ASOASF you get the idea.

I think it has something to do with the funky way **Detections** are stored / organized. I had hardcoded some limitations on the number of elements that would trigger the code snippet, but 216 taught me that hard-coding is a cardinal sin, so I wanted something more flexible.

Literally 5 min after writing this I fixed it lol. I changed `detect_arr` to a set, that way duplicates are not allowed.

```
detect_arr = set()

...

for res in results:
    for x in range(len(results)):
        if x not in detect_arr:
            detect_arr.add(results[x].tag_id)
    print(detect_arr)
```

Now, sets are a bit fussier than arrays, so this may come back to bite me in the ass. But I'll just stick my head in the ground until it becomes too big to ignore. For now, it seems to work fine, even with more than 2 tags.

 Alt text

The real test will be tomorrow, where I plan to print out some tags on paper and try drawing lines between them. However, since the mice will also have tags on them, we need to make sure that only the tags that we

have set to be corners (AKA 0 - 3) are the only ones that get lines drawn between them. I was thinking something along the lines of:

```
corners = Array of tag numbers 0 - 3
for res in results loop:
  If "corners" intersects with "detect_arr":
    line(center of 0 -> center of 1)
    line(center of 1 -> center of 2)
    line(center of 2 -> center of 3)
    line(center of 3 -> center of 1)
```

As is, this looks pretty messy, so I think I may just make this a function to make it a bit cleaner.

Tasks to be completed

~~4-3 Finish drawing the arena's box in the frame~~

4-4 Incorporate additional tags for the mice, start work on boundary detection

I expect that adding on the tags for the mice will be a quick affair (😁) and afterwards, the real pain will come with boundary detection. I think it may take some calculus to detect when a collision occurs, since we only have the starting and ending points of the lines that make up the arena. In theory, we could do boundary detection within the frame displayed, but this wouldn't be ideal, since we would need a screen on at all times. Realistically, the drawing on the frame is only really necessary for us humans to understand what's going on and for debugging purposes. Ideally, boundary detection would happen in the background in order to save on computing power (however slim those savings may be).

DCF

Week of 3-31-2023

Tasks completed

Webcam is calibrated

3-28

We've finalized how we're gonna use the AprilTags. We will be suspending a camera above the arena, using five tags to denote the four corners of the arena as well as the center point of the arena for the mice to go towards if it gets too close to the outer bounds of the arena. Because we're no longer using sound localization via time delay, we can simply use the camera's frame of reference instead of real world frame of reference, which will make life a whole lot simpler (a rare commodity in these times)

Thus, the frame now displays the current coordinates for each AprilTag's center

 Alt text

The next step would be to find the distance between two tags in the camera frame and hopefully draw lines between the four corners. From here, detecting new tags on the mice would be the next step for boundary detection. Apart from this, I did some grunt work, getting tags from the AprilTag repo, enlarging them in


order to not have to rip off the wallpaper from the lab's walls, and some documentation here and there. (Mostly because my memory is like a collander with three inch holes and I forget everything, or as Gen-Z says it, i forgor 🧠)

3-27

Started work on the webcam calibration. We took some very rough measurements of the area covered by the webcam from one of those hanging electrical outlet thingies. It's roughly 6x8 ft.

I took 80 pictures for the calibration of the webcam, however following the same steps as for the mouse's webcam yielded unusable results. Need to investigate further.

Removing the errata does not seem to work... until you remove literally half of the data lol. See for yourself

 Alt text

 Alt text

The change is not as significant as with the smaller mouse camera, which is to be expected. The webcams' lens is not as convex as the smaller one. Regardless, this is still important, since the four corners will contain the four tags on the ground that will be used to calculate the distances between the mice.

Tasks to be completed

~~3-27: Start distance calculations~~

3-28: Real-life distance is no longer necessary, but we still need a reliable way to measure distance between tags in the camera frame

-DCF

Week of 3-24-2023

Spring break, cya next week



DCF

Week of 3-17-2023

Tasks completed

Successfully got the camera calibrated

This was pretty challenging, since I had to get into the weeds of how the OpenCV library worked. Regardless, I went ahead and added comments on what certain functions returned, what were their params, etc. This is with the hope that, as the project advances, everyone is on the same page and understands how the technology works. Here are some before and after pics.

 Alt text  Alt text

Pretty substantial change huh? Like a Proactiv commercial. Anyways, the straightening now occurs on the camera live feed, which was done by cropping out the black edges on the frame. This led to a pretty clear (albiet now reduced) representation of what the camera was seeing.

Tasks to be completed

Calculate distance between tags IRL

Now that the camera is properly calibrated, we can move on to the question of distance calculation. Some potential challenges may be the camera itself, which is ~~dog-shit~~ of lesser quality. Currently, the camera does not detect tags farther than about a yard, which may limit the size in which we can run the game, in turn making for a very boring demo. To remedy this, there are some articles on how to increase the range of the detection by using CV *M A G I C*. More likely, we'll just do the top-down view plan, as we can just use a calibrated webcam. It might even make the distance calculations more straightforward.

-DCF

Week of 3-10-2023

Tasks completed

Successfully rectified a still image taken by the camera

The `straighten_image_fisheye.py` code now works and rectified a still image that was passed to it. Before that, there was a lot of data collection that had to be done. This was done with `snap.py`, a short program that can take pictures and write them to a specific folder. At first, the pictures taken were not the best for the calibration, since they were taken at more or less the same angle and in the same position without consistent variance. To remedy this, `snap.py` was given a grid in the GUI in order to take consistent sample photos for the training. The resulting photos did not include the grid itself, since I had two cam streams open, one for the sample to be collected and one for the user's convenience.

```
while True:
    ret0, frame0 = cam.read()
    draw_grid(frame0, (3, 3))
    ret1, frame1 = cam.read()
```


Additionally, some versatility has been added in order to ease the switch between the Jetson and my Windows machine when accessing certain directories.

Tasks to be completed

Finish calibration of the camera

So far, the calibration is almost done. There are some black edges that need to be cropped out (easy) and there needs to be a reliable way to calibrate the live feed of the camera (not so easy). However, with the groundwork set this week, I'm confident that it won't be too bad to integrate. Surely, right?



-DCF
