# Localization and Networking Documentation Version 2.0

Keegan Kelly

August 3, 2022

## Contents

## 1 Introduction

This document will cover camera calibration, how OpenCV is used to find Aruco markers, how the position of all markers is calculated, how the web-server used to host the position data for communication between the host and Arduinos, how to use the tracker class, and how to get the data from the web-server to the ESP8266 and then to the Arduino. The code for this project is available on my GitHub. If you require information on the robot control systems, please refer to the Robot Control Documentation. If you require more details on how the systems work refer to the detailed comments in the code.

## 2 Installation Requirements

In order to use the localization component of this project a few libraries and programs must be installed. First you require a Python installation with the following libraries installed:

- numpy
- opencv-python
- opencv-contrib-python
- pandas
- requests
- asyncio
- aiohttp
- flask

These libraries can easily be installed from the command line by typing

```
pip install LibraryName
```

Due to the use of openCV, C++ could likely be used instead of Python. However, the performance increase would be minimal as the Python openCV package runs in native C++.

On the Arduino side, you will need the ArduinoJson and WiFiClient libraries and the ESP8266 board installed with the Arduino IDE. To add the ESP8266 board to the IDE, go to preferences and paste this link into the Additional Boards Manager URLs field.

```
http://arduino.esp8266.com/stable/package_esp8266com_index.json
```

## 3 Camera Calibration

In order to use computer vision to estimate the position of robots, the camera matrix, and distortion matrix for the webcam must be calculated. This process is very simple, but some careful steps must be taken as the accuracy of all measurements will be directly tied to the quality of the calibration. First, the checkerboard found in the file $Checkerboard - A3 - 50mm - 7x4.svg$, within the $CameraCalibration$ folder, must be printed and placed onto a flat surface such as a clipboard. With this, many pictures (500-600) will be taken of the checkerboard moved and rotated throughout the frame and a script will find the checkerboard in each picture and use it to discover the properties of the camera. The focus of the camera must be fixed to the focal length that will be used during the position tracking. One of the properties in the camera matrix is the focal length. If auto focus is enabled or a different focus is used in calibration and operation, the obtained matrix will be inaccurate and the results will not be usable. If you are using openCV to take the pictures, these settings can be used with the resolution and frame rate set according to the specifications of the camera used.

```
# open the camera and sets the resolution , frame rate , and focus
cap = cv2.VideoCapture(0, cv2.CAP_DSHOW)
cap.set(cv2.CAP_PROP_AUTOFOCUS, 0)
# lower focus focuses further away from the camera
focus = 0  # min: 0, max: 255, increment:5
cap.set(cv2.CAP_PROP_FOCUS, focus)
cap.set(cv2.CAP_PROP_FRAME_WIDTH, 1920)
cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 1080)
cap.set(cv2.CAP_PROP_FPS, 30)
```

Included in the *CameraCalibration* folder is *calibrationCameraCapture.py*. This script will take pictures every 0.5 seconds, so you should slowly walk around the frame with the chessboard in different orientations and elevations as the script runs. Getting the chessboard in different positions will get a better calibration, however, a lot of pictures should be taken with the board on the ground as that is the elevation where the robots will be. When calibrating, I took about 600 pictures and then went through and deleted all pictures with any blur on the chessboard and ended up with  250 good pictures. Having any blur on the chessboard will not produce a good camera matrix or distortion parameters.

After taking the pictures, place them into the *CameraCalibration* folder and run the *calibrate.py* script. The runtime for generating the matrix does not scale well with the number of pictures so be prepared to wait (with  250 pictures the runtime on my desktop was about 30 minutes). This script will print the camera matrix and store the camera matrix *mtx* and the distortion matrix *dist* in the *calibration.npz* file to be used with the position tracking script. Copy this file into the host system folder to use the new camera properties for position tracking. Unless the camera is changed, focal properties of the camera is changed, or the position estimates are inaccurate, the calibration process only needs to be done once.

## 4 ArUco Markers

For this project, markers with ids 10, 11, 12, and 13 will be used with 10 being placed on the origin, 11 on agent 1, 12 on agent 2, and 13 on agent 3. The *.png* files for the markers can be found in the *ArUcoMarkers* folder. If you wish to use different marker styles, sizes, or ids, the markers can be generated using *makeMarker.py* in the same folder. The markers included in the repository are $4 \times 4 - 1000$ and were generated with a size of 600×600 and printed at full scale from Google Chrome. Using Microsoft's built in photos software will scale the image poorly when printing making the it blurry. If you need to print off a different marker, just change the filename on line 31 if the script and the marker id on line 30. When using the markers, the background must be distinct from the black boarder of the marker or the corners can not be effectively found. It is recommended to leave a sizable white boarder around the markers when placing them on top of the robots. ArUco markers also have a correct orientation that will be used in calculations. Using the *.png* file's orientation as reference, the $+y$ direction is pointing from the bottom to top and the $+x$ direction is pointing from left to right. When placing markers on the robot or ground, align the $(x, y)$ axis of the marker with that of the robot or the origin. If you are unsure about the correct orientation, the *main.py* file in the host system folder will put the X,Y,Z axis of each marker on the image. Make sure to measure the side length of all the markers to ensure they are the same size. If they are not, the position estimates will not be accurate.

## 5 ArUco Tracking and Position Estimation

To find the markers in the frame and estimate their position, the ArUco functions within OpenCV are used. The function

```
cv2.aruco.detectMarkers(frame, arucoDict, arucoParams)
```

returns the corners and ids of all ArUco markers within the frame. These are not ordered by id, so it is recommended that they should be sorted into a dictionary by id. The function takes an image (use grayscale for best results), the type of ArUco marker to find, and the ArUco detection parameters as arguments. The detection parameters are generated by

```
params = cv2.aruco.DetectorParameters_create()
```

and it is recommended that you also assign a corner refinement method to increase accuracy using

```
params.cornerRefineMethod = cv2.aruco.CORNER_REFINE_SUBPIX
```

The arucoDict is generated with

```
arucoDict = cv2.aruco.Dictionary_get(ARUCO_DICT[aruco_type])
```

where ARUCO_DICT is a dictionary containing types of aruco markers and aruco_type is a string with the key for the desired type of marker in the dictionary. With one set of corners, the function

```
cv2.estimatePoseSingleMarkers(corners, markerWidth, cameraMatrix, cameraDistortion)
```

will return the rotation vector, translation vector, and marker position of the marker. The marker position return is not needed. If you pass multiple sets of corners, it will return the rotation and translation vectors for each set in an array. The rotation vector describes how the marker is rotated relative to the camera frame and the translation vector describes the displacement of the marker relative to the camera frame. The function

```
cv2.Rodrigues(rotation)
```

will return a rotation matrix if a rotation vector is given or return a rotation vector if the matrix form is given. With the rotation and translation vectors of two markers, (one referred to as the origin and the other referred to as the marker) the position of the marker relative to the origin marker can be found by doing the following:

$$MarkerPosition = OriginRotationMatrix \cdot (MarkerTranslation - OriginTranslation)$$

The rotation matrix of the marker relative to the origin can then be found by computing

$$NewMarkerRotationMatrix = OriginRotationMatrix \cdot OldMarkerRotationMatrix$$

After converting this rotation matrix back to a rotation vector, the Z rotation will be the heading of the marker relative to the origin marker. Make sure to add back an offset of $\pi/2$ to get a proper heading. If the order of the rotation matrices is reversed, the heading will be measured in the opposite direction (this would be the yaw of the robot). If this direction of measure is desired, make sure this interpretation is consistent with the robot's odometry.

If there is considerable noise on the position estimate there are a few likely causes. If the axis on the markers are oscillating considerably, it is likely that the marker is too small in the camera frame and the corners can not be located with accuracy. Typically, this issue also presents with the detected marker outline oscillating. To remedy the issue the focal length of the camera could be adjusted or the size of the marker could be increased. Before changing the marker, try picking the robot up off of the floor and see if the noise on the position estimate goes away.

ChAruco boards and boards of Aruco markers were also experimented with as they increase the accuracy of detection. However at a similar total size, the individual markers must be smaller, which drastically worsens the results making it better to use a single, larger marker.

## 6 Getting Camera Capture

In the *hostSystem* folder, the file *webcamvideostream.py* contains the class WebcamVideoStream. This class is used to get frames from the webcam on a thread so it does not limit processing. If you want to change the properties of the camera capture, edit the function __init__(). To use the class, declare an object (default arguments will likely be correct) and then call the member function *start*() to begin capture and call *stop*() to end the capture. After calling stop, you should also call *objectname.stream.release*() to free up the webcam.

## 7 Showing Images

To display an image with openCV, use

```
cv2.imshow('frame', self.outFrame)
if cv2.waitKey(1) & 0xFF == ord('q'):
    #if you are getting frames from a webcam insert code to close the webcam feed
```

It is recommended that this process is done on a separate thread in a loop to improve performance.

## 8 Using the API Web Server

Before starting the server, navigate to the *api.py* file and in the call to app.run(), change the IP address to the local IPV4 address of the host computer and set the port to the desired port. To find the local IPV4 on Windows, open a command prompt and type *ipconfig*.

To start the server you need to run two commands in a terminal window navigated to the main server/api folder (api). Just run

```
python api.py
```

This will start the server. Whenever a HTTP request is made to the server, the type and address of the request made will be shown along with the status code and time the request was received. A code in the 200 range indicates success, a code in the 400 range indicates an error with the request sent, and a code in the 500 range indicates a server error.

To close the server press Ctrl+C. This will save the current state of the data to the db.json file.

With the server, data is stored in memory as a dictionary (more or less equivalent to a Json), sent to clients as Json data, and then written to *db.json*. The position of each agent is stored in the JSON array "agents". Within the array there are objects for each agent with the id of the agent and a position array where the localized position of the agent will be stored $(x, y, \theta)$. If you wanted to access the localized position of agent 1, you would make a request to "http://serverAddress:serverPort/agents/1". To receive the data stored there you would make a GET request and to update the data there, you would make a PUT request. Using the same structure as the localized position, the server also stores the current position estimate from the robot in the JSON array "agentsLocal". This field is used for debug and testing purposes to wirelessly transmit the position estimate of the robot as in normal operation this data is unnecessary.

## 9 Getting and Sending Target Positions

The target positions are stored in a JSON array that is segmented in multiple arrays due to the memory limitations on the Arduino. These arrays are stored in objects called goalid, where id is the id of the agent. Within each object there are multiple objects. The id field contains the index of that array (starting at 1), the total field contains the number of segmented arrays, dt contains the time step between paths, and path is the array of target positions. If you wanted to access the 2nd array segment for agent 3, your request would be to "goal3/2".

The path arrays are imported from a excel file using pandas in main.py. If you want to change the filename, change the variable filename at the top of the script. If your path data is segmented differently, be sure to change how dt is calculated and how the temp array is spliced in the creation of the data dictionary.

To get the path on the Arduino, use the Robot class function getPath(). This function will get the path field from the server and store it in the pathDoc JSON document. The data is not extracted into an external array due to memory limitations.

## 10 Sending System Ready Signals

By default the server field /agentGo/1 should be set to "ready":0 by the main.py script to indicate that the agents are not all ready. When each agent is at the first desired position, it should send "ready" : 1 to the server field /agentReady/id where id is the id of the agent. This is done by calling the Robot class function setReady(). The host computer should continually check the ready status of all 3 agents to see if all are in the ready position. If they are, it should then update /agentGo/1 to "ready":1. At the same time, each robot should be checking if that field is 1 with the Robot class function getReady() that returns a 1 if it is ready or 0 if it is not or there was an error.

## 11 Sending and Receiving Data Between a Computer and the Server with Python

To send HTTP requests with Python, the libraries *asyncio* and *aiohttp* should be used. These libraries allow requests to be sent asynchronously, reducing the total process time if you need to send multiple requests at once. For example, if three requests needed to be made, as is required for sending the localized position of each agent, normally it would take ∼30ms for each operation. Instead, these libraries send the requests one after the other, but it does not wait for each response before sending the next request resulting in a execution time of about 35-40ms instead of 90ms.

For examples on sending requests here is a tutorial. In the file *debugPositionRead.py* an example of a GET request can be seen and in the *tracker.py* file the function *put_data*(*data*) is used to send PUT requests. Note that *put_data*(*data*) accepts a dictionary containing the key and value pairs to be sent to the server and put in db.json. It also uses an address specified in the constructor for the Tracker class.

The return of a GET request will depend on what is stored wherever the request is made. JSON arrays will return an array and objects will be returned in a dictionary with the same key, value pairs as the JSON object. When sending a PUT request, the data to send can be formatted in a variety of ways. After the argument for the address, use json=data as the data argument where data is a dictionary. This will send the data to the server as a JSON. Using data=data will send the data to the server as a text string.

## 12 Programing the ESP8266

In order to program the ESP8266 you will need to setup a serial connection with the computer. You can use a USB to UART connection or connect it to the computer through a Arduino. Simply connect the 5V from the Arduino to VBAT or V+ on the ESP8266 (the board only requires 3.3V but it has a built in regulator up to a 6V input and the 3.3V output on the Arduino can't supply enough current), short the reset pin of the Arduino to ground, then connect the TX's of the Arduino and ESP8266, and connect the RX's of the Arduino and ESP8266.

To enable programming mode on the ESP826, press and hold the GPIO0 button on the ESP8266, press and hold the reset button, then release the reset button, followed by releasing the GPIO0 button. While the GPIO0 button is depressed, the indicator LED will glow bright red and when successfully put into programming mode, it will glow a much dimmer red. Instead of using the buttons on the board you can short the GPIO0 and reset pins to ground with an external button. Programming mode will remove whatever software is currently on the board.

On the robot external buttons have been added to make it easier to program. Just set the 3 position switch to PGRM ESP and then instead of depressing the switches on the module, depress the external ones. An image of the button layout has been included below.

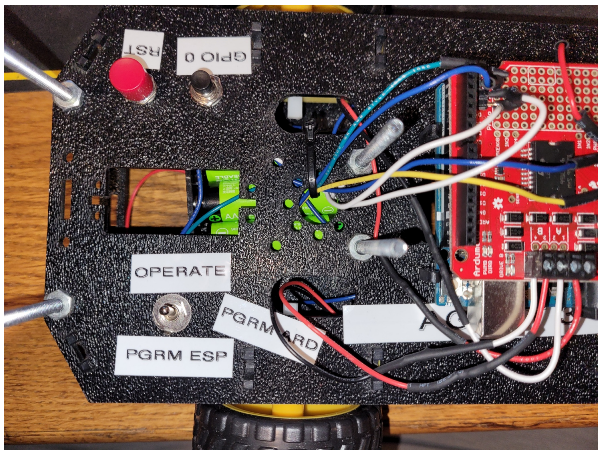

Figure 1: Image of the switches on the robot

Once the doce is uploaded make sure to switch the 3 position switch to PGRM ARD to program the Arduino or Operate to run the robot as normal.

To program the board using the Arduino IDE, import the board to the board manager as instructed in the Installation Requirements section. Next select the Adafruit Feather HUZZAH ESP8266 as the board from the ESP8266 boards section and pick the COM port for the Arduino it is connected through. Now you can program the ESP8266 in the same way you would and Arduino. In the bottom terminal section of the IDE, you should see Connecting followed by a series of ellipses. While ellipses continue to print to the screen, check your connections or try putting the board into programming mode again. If there is an error message while uploading the program, try putting the board in programming mode again and try another upload.

After uploading, press and release the reset button or power cycle the board to run the program.

## 13 Sending and Receiving Data Between the ESP8266 and Server

The ESP8266 is a powerful micro-controller that is almost capable of running the entire robot (this was tested, however, there were crashing issues with setting PWM for the motor drivers). GET and PUT requests are very simple to implement following this guide. With ArduinoJson, the serial returns/sends can be made/turned into a JSON very easily. All code to be put on the ESP8266 is in the $ESP8266$ folder.

Before connecting to the server you will also need to connect to WiFi. This is simple and accomplished with the function *void connectWiFi*() in the file $ESP8266$*.ino*.

## 14 Communicating Between the ESP8266 and Arduino

To send data between the Arduino and ESP8266 hardware serial must be used. Since an Arduino Uno is being used on the robot this will eliminate the ability to print to the serial monitor as the Uno only has one serial line. Additionally to program the Arduino you will need to remove the serial connection between the Arduino and ESP8266. Software Serial was tested however the performance was poor and noise in the transmission would make the transmissions fail frequently even at low baud rates.

### 14.1 Hardware Connections Guide

Simply connect both boards to a common ground, and connect the TX of each board to the RX of the other (TX is transmit and RX is receive). This is the opposite connection needed to program the ESP8266. Make sure that the baud rates of both boards are the same or the transmissions will not work.

### 14.2 Serial Communication with JSON Guide

The boards can communicate by printing to and reading from the Serial line with basic text, however, a system using this form of communication will be likely inefficient and complicated. The Arduino Json library has functions to serialize and deserialize a JSON document (translate the data structure into a text string or vice versa) which can send and receive directly to and from the serial line. To accomplish this communication (Arduino to ESP) the ESP8266 has been set in a loop checking if there is data available on the serial line. If there is, it will deserialize the JSON sent and determine what type of request to make and what the address to send it to is. In the current implementation the JSON sent will have a few fields, ”type” and ”address”. The type will either be GET or PUT and the address is a string with the address where the request is to be made. If it is a PUT request, there will also be a fields for ”id” and whatever data you are sending. In a PUT request, when the JSON is received, it will check if a certain key is in the JSON. The presence of a key can be used to determine how to setup the request to be sent in the payload string to the PUT function.

After receiving the type of request from the Arduino, the ESP8266 will make the request and if it is a GET request it will serialize the JSON output to the serial line. If it is a PUT request, there will no output to sent to the Arduino.

On the Arduino side, after sending the JSON to make the request, if it is a GET request, it will wait in a loop until there is data on the serial line to be received. It will then deserialize the JSON and extract the desired data from it. If it is a PUT request it will continue running the program as it expects no return.

## 15 Python Code Overview

### 15.1 WebcamVideoStream Class

This class gets a video feed from a webcam. The most recent frame is available at $objectName.frame$ and a Boolean value for the success of the getting the frame is *objectName.grabbed*.

#### 15.1.1 WebcamVideoStream(name="WebcamVideoStream", height=480, width=852, fps=60, focus=0)

The class constructor function initializes the capture and sets the properties of the camera. This function can be called without arguments and they should only be provided if you wish to change the resolution, frame rate or focus. If you need to change any other camera properties edit the function or access the camera capture directly at *objectName.stream*.

#### 15.1.2 start()

This function starts the capture thread.

#### 15.1.3 update()

This function is the main loop to be ran, which continually gets frames from the camera

#### 15.1.4 read()

Returns the most recent frame.

#### 15.1.5 stop()

Stops the camera capture. Does not release the camera capture.

### 15.2 Tracker Class

This class contains all the functions necessary to track Aruco markers, estimate their position, send their position to the server, as well as the functions to do all of these tasks on separate threads to improve performance.

#### 15.2.1 Tracker(marker_width, aruco_type, address)

This class constructor takes the width of the markers in m, a string Key for the type of marker in the dictionary, and the address of the server. It will initialize the parameters for detecting Arucos.

### 15.2.2 fixAngle(angle)

This function returns the argument angle in the range $[-\pi, \pi]$.

### 15.2.3 find_markerPos(frame)

This function accepts a frame as an argument and will find the Aruco markers within the frame and add their corners to the class' corner dictionary so the corners can be accessed by the marker's id. It will then estimate the position of each marker relative to marker 10 (the origin) and store their position the class' position matrix. This function will only update the origin's position relative to the camera if the Boolean self.originFound is false to eliminate some noise from position estimates. The function also computes the frame rate and draws it on the frame.

### 15.2.4 get_tasks(session, data)

This function is used for asycronous PUT requests and generates a list of tasks for the asyncronous function to perform.

### 15.2.5 put_data(data)

This function accepts a dictionary to be sent to the server and will execute the PUT requests.

### 15.2.6 startThreads()

This function will start the threads for getting camera frames, detecting markers within the frames, displaying the frames, checking the reads status, and sending the position estimates to the server.

### 15.2.7 stopThread()

This function will stop all threads started with startThreads() and releases the webcam capture.

### 15.2.8 runProcessFrame()

This function will continually run find_makerPos(frame) on the newest frame from the camera.

### 15.2.9 runGetFrame()

This function will initialize a WebcamVideoStream class and set an initial frame to be displayed to avoid a crash on startup.

### 15.2.10 runShowFrame()

This function will display the most recent frame and print the current position of each robot to the screen. If tabbed into the display window, pressing q will end the program and pressing r will set originFound to False so the program will refresh the location of the origin. This refresh functionality is meant to be used if the camera moves while the script is running or the initial estimate of the origin is poor.

### 15.2.11 checkReady()

This function runs a loop to make requests to the server for the ready status of all three robots. If all three are ready then it will send a request to the server to set the go variable to 1.

### 15.3 main.py Script

This script should be run to start tracking markers, sending their position to the server, sending the desired path to the server, and checking if all agents are ready to begin following the path. To stop the script tab into the video feed window and hold q. If the server is not on when the script is started it will likely crash. When ran, this script will prompt the user to see if each agent will be used.

### 15.4 debugPositionRead.py Script

This script is for debug purposes and will GET the odometry estimates of each robot from the server and print them to the screen.

## 16 API Server Code

All code for the server is located in the api folder in *api.py*. The API server is very simple, it stores the Json data in dictionaries, reads and writes the data to db.json on startup and shutdown, and handles requests with the flask framework. The code is very easy to follow with a simple understanding of routes. For more info on routes in flask check out this guide that goes much further than is needed to understand the code.

## 17 Arduino and ESP8266 Localization Code Overview

### 17.1 ESP8266

#### 17.1.1 void GET(String address)

This function will make a GET request to the address and serialize the response and send it to the Arduino.

#### 17.1.2 void PUT(String address, String payload)

This function will make a PUT request with the payload string being a serialized JSON.

### 17.2 Arduino

In the Robot class there are four functions and two variables related to localization.

#### 17.2.1 Public Variables

- int id: the id of the Robot assigned from an argument to the constructor
- char serverAddress[30]: the address of the sever assigned from an argument to the constructor

#### 17.2.2 int localize()

This function makes a GET request to the server for the localized position of the Robot and then updates the x,y, and theta of the robot. It returns a 1 if the localization was successful and a 0 if it was not.

#### 17.2.3 void putPosition()

This function will make a PUT request to the server with the current position estimate of the robot from the odometry.

#### 17.2.4 void setReady()

This function will send data to the server indicating that the agent is ready to begin moving.

#### 17.2.5 int getReady()

This function will make a request to the server to see if the agent should start moving. It will return a 1 if all agents are ready to begin and a 0 if they are not all ready or there was a error.