Indoor Positioning Ultra-Wide Band   
Server (IPS UWB)

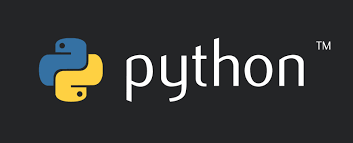
Requirements:

The following requirements must be satisfied to be able to run the IPS UWB server.

1. Git: Download and install git on your device from the following link <https://git-scm.com/downloads>



1. Python: You must also have python installed. Please consult this tutorial to install Python and add it to the environment variables on your device https://www.digitalocean.com/community/tutorials/install-python-windows-10



Installation & Setup:

Once you have git and python installed, you may clone the repository:

1. Open a terminal window on your device and use

git clone https://github.com/MohammedMAmir/IPS\_UWB.git

1. The project folders should be cloned onto your device
2. Use the following command to navigate to the project folder

cd IPS\_UWB

Now you can setup all the dependencies for the project:

1. In your terminal window, inside the IPS\_UWB folder, enter

python3 pip install virtualenv

1. Once the installation is finished, use

python3 -m venv .venv

This will create a virtual environment folder that lets you import all the dependencies of the project without overriding dependency versions already installed on your device.

1. To activate the virtual environment use the command

source .venv/Scripts/activate

in your terminal window. You will need to do this everytime you open a new terminal and want to run the server.

1. Navigate into the "ips\_app" folder using the command

cd ips\_app

in the terminal window

1. Finally, install all of the modules used by the project by entering the command

python3 -m install -r requirements.txt

1. Wait for all of the modules to install

* if an error occurs or a module is missing, just install it manually using the command

python3 -m pip install [missing module]

(where missing module is replaced by whatever module triggered the error and is missing)

1. Now you're ready to start the server!

Running the Server

Running the server for this project is as simple as running the server script

1. Open a terminal window and navigate to the ips\_app folder using

cd /path/to/folder/IPS\_UWB

1. Start up your virtual environment using

source .venv/Scripts/activate

1. Once the virtual environment is running, navigate into the ips\_app folder using

cd ips\_app

1. Once in the ips\_app folder, to start the server, use the command

python3 -m server

1. You should get a message that the server is running on 127.0.0.1:81. You can now type https://127.0.0.1:81 into any browser to view the server responses

Using the Database

The database currently contains two tables:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| tag\_id | senior\_name | senior\_x | senior\_y | num\_anchors |
|  |  |  |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| anchor\_id | tag\_id | anch\_x | anch\_y | anchor\_distance |
|  |  |  |  |  |

Tags:

Each tag has 5 fields:

1. tag\_id (integer)

* A unique id that identifies each tag
* Autoincrements anytime a new tag is added

1. senior\_name (string)

* The senior this tag identifies

1. senior\_x (float)

* The current x position of the tag

1. senior\_y (float)

* The current y position of the tag

1. num\_anchors

* The number of anchors associate with this tag

Anchors:

Each anchor has 5 fields:

1. anchor\_id (integer)

* A unique id that identifies each anchor
* Autoincrements anytime a new anchor is added

1. tag\_id (integer):

* The tag that this anchor is associated with
* There must be a valid tag in the tag table for this to reference

1. anch\_x (float):

* The fixed x position of the anchor in its coordinate space

1. anch\_y (float):

* The fixed y position of the anchor in its coordiante space

1. anchor\_distance (float):

* The distance from the anchor to it's corresponding tag
* Updated by the anchors using a PATCH request

Calling the API

The api for the project can be used to query and update the database. It is broken down into 4 parts. Although all of the functionalities of the API can be accessed through website dashboard, they can also be accessed using HTTP GET/POST requests and therefore, from anchors and tags. The API is broken down as follows:

Tags (available at the url: https://127.0.0.1:81/api/tags)

* Used to get information on all tags stored in the database
* [https://127.0.0.1:81 can later be replaced with whatever url the server is hosted at]
* Endpoints:
  + GET:
    - Request takes no parameters and returns a list of all of the tags currently stored in the database
  + POST:
    - Request takes a post request with the following JSON message body:

{"senior\_name": "[some senior name]"}

* + - The corresponding tag will be created in the database with a unique tag ID and initial senior x and y position of (0, 0)
    - The request will return a list of all the tags in the database

Anchors (available at the url: <https://127.0.0.1:81/api/anchors>)

* Used to get information on all anchors stored in the database
* [https://127.0.0.1:81 can later be replaced with whatever url the server is hosted at]
* Endpoints:
  + GET:
    - Request takes no parameters and returns a list of all of the anchors currently stored in the database
  + POST:
    - Request takes a post request with the following JSON message body:

{"tag\_id": "[some tag id]", "anch\_x": "[the anchor x position]", "anch\_y": "[the anchor y position]"}

* + - The corresponding anchor attached to the specified tag will be created in the database with a unique anchor ID and anchor x and y positions specified

Tag (available at the url: https://127.0.0.1:81/api/tag/)

* Used to get, update, and delete information about tag with tag\_id = id
* [https://127.0.0.1:81 can later be replaced with whatever url the server is hosted at]
* Endpoints:
  + GET:
    - Request takes no parameters and returns the tag in the url if it is stored in the database
  + PATCH:
    - Request makes a patch request with the following JSON message body:

{"senior\_name": "[some senior name]"}

* + - The senior name of the tag in the url will then be updated to instead be registered to the senior name specificed in the request
  + DELETE:
    - Request takes no parameters and deletes the tag that in the url if it is stored in the database
    - If the tag has corresponding anchors attached to it, they will also be deleted from the database

Anchor (available at the url: <https://127.0.0.1:81/api/anchor/>)

* Used to get, update, and delete information about tag with anchor\_id = id
* [https://127.0.0.1:81 can later be replaced with whatever url the server is hosted at]
* Endpoints
  + GET:
    - Request takes no parameters and returns the anchor in the url if it is stored in the database
  + PATCH:
    - Request makes a patch request with the following JSON message body: Request makes a patch request with the following JSON message body:

{"anchor\_distance": "[the anchor's updated distance from itself to it's tag]"}

OR

{"anchor\_distance:" "[the anchor's updated distance from itself to it's tag] "}

* + DELETE:
    - Request takes no parameters and deletes the anchor with the anchor\_id specified in the url if it is stored in the database

Positioning Calculation

The positioning algorithm is called everytime that an anchor makes a PATCH request to the database. Specifically, everytime an anchor updates the distance to its associated anchor, the following two functions are called by the server to calcualate the new position of the tag:

While it is definitely true that trilateration can be seen (and solved) as a geometrical matrix problem, this is often impractical. Relying on the mathematical modelling requires us to have an incredibly high accuracy on our measurements. Worst case scenario: if the circles do not meet in a single point, the set of equations will have no solution. This leaves us with nothing. Even assuming we do have perfect precision, the mathematical approach does not scale nicely. What if we have not three, but four points? What if we have one hundred?

The problem of trilateration can be approached from an optimisation point of view. Ignoring circles and intersections, what is the point that provides us with the best approximation to the actual position P?

Given a point X, we can estimate how well it replaces P. We can do this simply by calculating its distance from each anchor Li. If those distances perfectly match with their respective distances di, then X is indeed P. The more X deviates from these distances, the further it is assumed from P.

Under this new formulation, we can see trilateration as an optimisation problem. We need to find the point X that minimises a certain error function. For our X, we have not one but n sources of error: one for each anchor:

A very common way to merge these different contributions is to average their squares. This takes away the for possibility of negative and positive errors to cancel each others out, as squares are always positive. The quantity obtained is known as mean squared error:

What is really nice about this solution is that it can be used to take into account an arbitrary number of points. The piece of code below calculates the mean square error of a point x, given a list of locations and their relative distances from the actual target.

Mean Square Error

# Mean Square Error

# locations: [ (x1, y1), ... ]

# distances: [ distance1, ... ]

def mse(x, locations, distances):

mse = 0.0

for location, distance in zip(locations, distances):

distance\_calculated = math.dist(x, location)

mse += math.pow(distance\_calculated - distance, 2.0)

return mse / len(distances)

Minimize

What’s left now is to find the point x that minimises the mean square error. Luckily, scipy comes with several optimisation algorithms that we can use.

# Used to calculate location of tag using anchor distances as specified by:

# https://www.alanzucconi.com/2017/03/13/positioning-and-trilateration/

# input: the anchor making the update

# output: the updated x, y position of the senior stored as a tuple in result

def update\_location(anchor: AnchorModel):

# Get the tag\_id of the tag associated with anchor making the update

anchortag = anchor.tag\_id

# Grab the tag from the database

tag = tagModel.query.filter\_by(tag\_id=anchortag).first()

# Last known x and y positions of the senior

xToUpdate = float(tag.senior\_x)

yToUpdate = float(tag.senior\_y)

# Grab all of the anchors associated with this tag from the database

anchorsIntag = AnchorModel.query.filter\_by(tag\_id=anchortag).all()

locations = []

distances = []

# Create a list of anchor locations and a list of anchor distances

for anchors in anchorsIntag:

locations.append((int(anchors.anch\_x), int(anchors.anch\_y)))

distances.append((float(anchors.anchor\_distance)))

# Set the initial guess to the previous known location of the senior

initial\_guess = (xToUpdate, yToUpdate)

# Find the new x and y position of the senior by minimizing the error

# of a guessed location and the actual distances measured

result = minimize(

mse, # The error function

initial\_guess, # The initial guess

args=(locations, distances), # Additional parameters for mse

method='BFGS', # The optimisation algorithm

options={

'ftol':1e-4, # Tolerance

'maxiter': 1e+8 # Maximum iterations

})

# Set the new location of the senior in the database

tag.senior\_x = result.x[0]

tag.senior\_y = result.x[1]

db.session.commit()

return result

The math of the algorithm is explained at the following link https://www.alanzucconi.com/2017/03/13/positioning-and-trilateration/ .