

Table of Contents



**Project
Goal**

Dependencies

Documentation

Script & flow

Using

Table of Contents



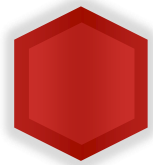
**Project
Goal**

Dependencies

Documentation

Script & flow

Using



Project Goal

The GO2 robot is limited in its physical capabilities and therefore there are tasks it cannot do alone, for example - press an elevator button, move an obstacle and so on. Because of this, it will need the help of the people around him.

The goal of the project is to make the robot recognize the people around him, choose one of them, and ask him for help.

Table of Contents



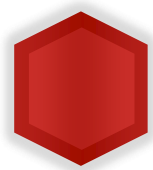
Project
goal

Dependencies

Documentation

Script & flow

Using



Dependencies



Python SDK

<https://support.unitree.com/home/en/developer/Python>
https://github.com/unitreerobotics/unitree_sdk2_python



OpenCV

`pip install opencv-python`



NumPy

`pip install numpy`



InsightFace

`pip install insightface`



PyRealSense2

`pip install pyrealsense2`

Table of Contents

Project
goal

Dependencies

Documentation

Script & flow

Using



Documentation

main.py

```
main()  
deciding(person: Person)
```

RS_API.py

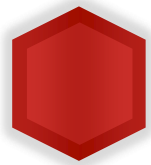
```
main()  
capture_rgb_images(num_pictures)  
capture_depth_images(num_pictures)  
calculate_median_image(num_pictures)  
create_clearer_depth_image()  
rotate_points(points, alpha_degrees,  
beta_degrees)  
load_intrinsics(file_path)  
save_intrinsics(file_path)  
get_XYZ_from_pictures_with_file(px, py)
```

Person.py

```
Name  
Angle  
center_XY  
XYZ
```

recognize_persons.py

```
train_faces()  
cosine_similarity(a,b)  
face_orientation_degrees(frame,  
landmarks)  
recognize_faces_from_embeddings(image  
_path, embeddings_file)  
get_persons(num_images)
```

Documentation

start.py

```
main(str(person : Person))  
calculate_route(robot :SportModeTest,  
angle, x,y,z)
```

GO2_API.py

```
class SportModeTest:  
    init_robot()  
    rotate_robot(radii)  
    goMeteresAhead(meters)
```



Person.py

- >> **Name** The serial name the person gets
- >> **Angle** The angle of the face in relation to the camera. If the face is looking straight at the camera, the angle is said to be 0 degrees. If the face is looking to a place that is to the right of the camera, the angle is said to be positive, and vice versa.
- >> **Center_XY** The center of the person's faces in the picture. It's a tuple of [x_pixel, y_pixel]
- >> **XYZ** Approximation of the person's face location in reality, measure by cm. it's a tuple of [x, y ,z].



RS_API.py

>>

save_intrinsics(file_path)

This function captures the intrinsic parameters of the connected RealSense camera's RGB stream and saves them to a specified file in JSON format. The intrinsics include parameters such as focal length (fx, fy), principal point (ppx, ppy), and distortion coefficients. The function uses the pyrealsense2 library to interact with the camera and retrieve these parameters. The pipeline is created, configured for RGB streaming, and started. The intrinsics are then extracted and saved, and the pipeline is stopped.

It should be noted that in practice, it is not necessary to use it in every run; You can save the camera information once, and use the saved file.

>>

load_intrinsics(file_path)

This function loads the intrinsic parameters of a camera from a file (in JSON format) and reconstructs the rs.intrinsics object. It handles different distortion models supported by the RealSense camera and converts the dictionary back into the appropriate RealSense intrinsics format.

>>

create_images(no_pictures)

This function captures and saves aligned RGB and depth images using the RealSense camera. It starts a pipeline with RGB and depth streams, aligns the depth images to the RGB images, and saves each pair as a PNG file. Aligning the images is essential in order to match a pixel in the RGB image with its position in the depth image. The function captures the specified number of image pairs and waits 1 second between captures. Finally, it calls the calculate_median_image function to compute and save the median depth image.



RS_API.py

>>

**calculate_median_image
(no_pictures)**

This function calculates the median depth image from a set of aligned depth images captured by the create_images function. It loads the depth images, stacks them along a new dimension, computes the median along this dimension, and saves the resulting median image as a PNG file.

Reasons for Median Calculation:

- Depth images are sometimes not accurate enough, so averaging them (via the median) helps to mitigate inaccuracies.
- Occasionally, some pixels in the depth images may lack information, resulting in empty pixels. The median helps to handle these cases by providing a more reliable representation of the depth data.

>>

**rotate_points(points,
alpha, beta)**

Input:

points: An array of points in 3D space (XYZ), typically in the form of an Nx3 matrix.

alpha_degrees: The rotation angle around the X-axis, in degrees.

beta_degrees: The rotation angle around the Z-axis, in degrees.

Output:

A rotated array of 3D points (XYZ) after applying the rotation around the X-axis and then around the Z-axis based on the given angles.

The function first converts the input angles from degrees to radians, calculates the rotation matrices for the X and Z axes, combines them, and then applies the combined rotation matrix to the points.



RS_API.py

>>

**get_XYZ_from_pictures_
with_file(pixel_x, pixel_y)**

Parameters: pixel_x: The x-coordinate of the pixel in the image.

pixel_y: The y-coordinate of the pixel in the image.

Description: This function calculates the real-world XYZ coordinates of a point corresponding to a specific pixel in the depth image.

The process involves the following steps:

1. **Loading Depth Image:** The function reads the median depth image from a file (aligned_depth_image_m.png).
2. **Loading Camera Intrinsics:** The camera's intrinsic parameters are loaded from a file (intrinsics_file) using the load_intrinsics function.
3. **Extracting Depth Value:** The depth value at the specified pixel location (pixel_x, pixel_y) is retrieved from the depth image. The depth value is in millimeters.
4. **Deprojection:** Using the RealSense rs2_deproject_pixel_to_point function, the pixel coordinates and depth value are deprojected into real-world 3D coordinates (X, Y, Z).
5. **Conversion to Centimeters:** The resulting coordinates are converted from millimeters to centimeters.
6. **Adjusting Y-Coordinate:** The Y-coordinate is adjusted using a linear transformation.
7. **Rotation:** The function rotates the calculated XYZ coordinates using the rotate_points function, though the rotation angle is set to 0 degrees in this case, meaning no actual rotation is applied.
8. **Return:** Finally, the function returns the XYZ coordinates in centimeters, with the adjusted and rotated values.



GO2_API.py

Important clarification:

We did not find any functions in UNITREE's library that explicitly command the robot to move X meters straight/left, nor did we find any functions that explicitly command the robot to turn by an alpha angle in a certain direction. The MOVE command provided by the library acts more like a "stream," meaning the robot will continue moving in a loop until we manually stop it. The functions presented here were written using a combination of HYPER PARAMETERS that were empirically calculated through experiments. As a result, you may need to adjust these parameters to make the code more accurate for your robot.

For example: if the code instructs the robot to move forward 2 meters, but it only moves 1 meter, you will need to divide the parameter in the code by 2. The same applies to the ROTATE function. The Hyperparameters in the code are marked by HP.

In the code, we left examples of functions that were included when we downloaded the library, even though we didn't use them.

>>	goMeteresAhead(meters)	Input: meters (the distance in meters to move forward). Output: The robot moves forward by the given distance
>>	rotate(radii)	Input: alpha (the angle in radians to rotate the robot). Output: The robot rotates by the given angle. Description: This function moves the robot in a rotational motion using the specified angle alpha. The movement is based on the calculated iterations using a set velocity.



GO2_API.py

>>

Init_robot()

Input: None.

Output: Returns a robot object (SportModeTest) that allows interaction with the Unitree robot.

Description: Initializes communication with the Unitree robot and prepares the system to execute further commands like rotating or moving.



start.py

>>

main(string)

Input:

A string of coordinates and an angle, provided via the command line.

Example input:

“EVYATAR -16.92136929607713 -32.387353515625 130.20380488691146
168.88039459708455”

EVYATAR: A label (name).

-16.92136929607713, -32.387353515625, 130.20380488691146: The x,
y, z coordinates in 3D space.

168.88039459708455: The angle in degrees.

Execution Steps:

The script checks if the coordinates are provided as an argument.

If not, it prints an error message and exits.

If coordinates are provided, they are split into individual components: the name, x, y, z coordinates, and the angle.

The robot is initialized using the `init_robot` function.

The `calculate_route` function is called with the robot, the parsed coordinates, and the angle to move the robot accordingly.

Purpose:

The MAIN section processes the command-line input, initializes the robot, and calculates the route for the robot to position itself directly in front of the person using the `calculate_route` function.



start.py

```
>> calculate_route(robot, angle,  
x,y,z)
```

Input:

robot: A SportModeTest object, representing the robot.

angle: The angle in degrees, representing the rotation required to face the target.

x, y, z: Coordinates in 3D space representing the person's position.

Output:

The robot will move and rotate to position itself directly in front of the person's face based on the provided coordinates and angle.

Purpose:

This function calculates the necessary movements and rotations for the robot to position itself directly in front of a person's face, given their 3D coordinates. It calculates vectors and angles required for rotation and movement, using trigonometric formulas to align the robot properly.



recognize_persons.py

- >> face_orientation_degrees(frame, landmarks)**
Input: frame: The image frame containing the face.
landmarks: The facial landmarks (coordinates of key points like the eyes, nose, and mouth).
Output: yaw, pitch, roll: The three angles representing the face orientation.
Purpose: This function estimates the orientation of a person's face in 3D space based on their facial landmarks, returning the yaw, pitch, and roll angles.
- >> recognize_faces_from_embeddings(image_path, embeddings_file)**
Input: image_path: Path to the image in which faces will be recognized.
embeddings_file: Path to the saved facial embeddings JSON file.
Output: A list of recognized persons along with their name, yaw angle, pixel coordinates, and real-world XYZ coordinates.
Purpose: This function detects faces in an image, compares them with known embeddings to recognize the person, and calculates their real-world location and face orientation (yaw).
- >> get_persons(num_pictures)**
Input: num_images: The number of images to process for face recognition.
Output: A list of Person objects, each representing a recognized person with their name, yaw angle, image coordinates, and real-world XYZ coordinates.
Purpose: This function processes multiple images, recognizes faces in them, and returns a list of identified persons, ensuring each person is only counted once.



recognize_persons.py

>>

**train_faces(dataset_dir,
embeddings_file)**

Input: dataset_dir: The directory containing subfolders, each representing a person with their images.

embeddings_file: The path where the embeddings (face features) will be saved in JSON format.

Output: Saves a JSON file containing the facial embeddings of all people in the dataset.

Purpose: This function prepares and updates the facial recognition model. If a new face is detected that is not already in the dataset, it allows for adding that person's embeddings to the database. It ensures that the facial database can be expanded dynamically as new faces are encountered, enabling the system to recognize more individuals over time.

Table of Contents

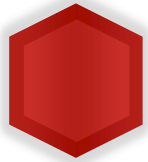
Project
goal

Dependencies

Documentation

Script & flow

Using



Script & flow

```
#!/bin/bash
```

```
# Step 1: Run RS_API.py (if needed)
```

```
echo "Running RS_API.py..."
```

```
python3 RS_API10.py
```

```
# Check if RS_API.py executed successfully
```

```
if [ $? -ne 0 ]; then
```

```
    echo "RS_API.py execution failed. Exiting..."
```

```
    exit 1
```

```
fi
```

```
# Step 2: Get coordinates from get_route.py
```

```
coordinates="$(python3 main.py | tail -n 1)"
```

```
echo "Coordinates: $coordinates"
```

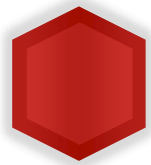


Script & flow

```
# Step 3: SSH into the robot, select ROS Foxy (1), set environment variables, and run start.py
echo "Start SSH"
```

```
# Use the -t flag to force allocation of a pseudo-terminal, ensuring all output is shown
sshpass -p "123" ssh -tt -o PubkeyAuthentication=no unitree@192.168.123.18 << EOF
echo '1'
cd ~/Documents/project/from_lola/FINAL
python3 start.py "$coordinates"
EOF
```

```
# Check if the SSH command executed successfully
if [ $? -ne 0 ]; then
    echo "SSH command execution failed. Exiting..."
    exit 1
fi
echo "Process completed."
```



Detail

```
# Step 1: Run RS_API.py (if needed)
echo "Running RS_API.py..."
python3 RS_API10.py
```

The first step in the script runs RS_API10.py with the goal of capturing Depth, RGB images, and a median Depth image. The median image is necessary because the Depth images can be noisy, and capturing multiple images with a time delay between captures reduces this noise. There are two files, RS_API.py and RS_API10.py, and they contain different versions of the functions. The idea is to eventually merge them, but for now, we're using functions from both files.

RS_API10.py: The main() Function:

The process starts in the main() function in RS_API10.py. It calls several important functions for capturing images and processing depth data.

Breakdown of the Functions Called by main() in RS_API10.py:

1. capture_depth_images(num):

- Goal: This function captures 4 RGB and 4 Depth images using the RealSense camera.
- How it works:
 - The function uses the pyrealsense2 library (RS) to interact with the camera and get the depth and color frames.
 - It captures the frames with some time delay between them to reduce noise. The Depth frames are aligned to the RGB frames for better accuracy.
- Output: The RGB images are saved as aligned_rgb_image_X.png, and the Depth images are saved as aligned_depth_image_X.png (where X is the image number).
- Explanation: The captured images are used for further face recognition and XYZ coordinate calculations later on. The aligned images ensure that both the RGB and Depth data are synchronized.

2. calculate_median_image(num):

- Goal: After capturing multiple depth images, this function computes the median depth image to reduce noise.


```
# Step 2: Get coordinates from get_route.py
coordinates="$(python3 main.py | tail -n 1)"
echo "Coordinates: $coordinates"
```

Goal:

In Step 2, the system:

1. Detects faces in the RGB images.
2. Matches them to the known faces from the `DATASET`.
3. Calculates their real-world XYZ coordinates.
4. Computes their face angle (Yaw).
5. Encapsulates this information in the Person object, which is used later by the robot to determine how to approach and interact with the detected person.

Detailed Breakdown of Step 2:

1. Face Detection and Recognition:

- ``get_persons()`` (in ``get_persons_proccess.py``) is responsible for detecting faces and recognizing them using embeddings from the ``face_embeddings.json`` file. This function processes multiple RGB images and calls ``recognize_faces_from_embeddings()`` for face recognition.

2. Structure of the `Person` Object:

- Each detected person is represented by a ``Person`` object (in ``Person.py``), which contains:
 - ``name``: The name of the detected person (or "Unknown" if not recognized).
 - ``center_XY``: The center of the detected face in pixel coordinates (X, Y).
 - ``real_XYZ``: The real-world XYZ coordinates of the person. Tuple of floats.
 - ``X_angle_degrees``: The Yaw (face angle) of the person.

3. Face Orientation (Yaw):

- The face angle (Yaw) is calculated using ``face_orientation_degrees()`` in ``get_persons_proccess.py``. This function estimates the face's orientation based on key facial landmarks (eyes, nose, mouth).
- Yaw represents the horizontal rotation of the person's face:
 - 0 degrees: The person is looking straight at the camera.
 - Positive Yaw: The person is looking to the right.
 - Negative Yaw: The person is looking to the left.

4. Face Recognition:

- ``recognize_faces_from_embeddings()`` compares the detected faces with known embeddings stored in ``face_embeddings.json``. This function uses the insightface library's FaceAnalysis class to extract facial embeddings and perform a similarity check.

5. Real-World XYZ Coordinates from the Depth Image:

- The real-world XYZ coordinates are calculated using the depth image with the function ``get_XYZ_from_pictures_with_file()`` (in ``RS_API.py``).

- The system first identifies the center pixel of the detected face in the RGB image.

- Then, it retrieves the corresponding depth value from the median depth image (``aligned_depth_image_m.png``).

- Finally, it uses the RealSense API's ``rs2_deproject_pixel_to_point()`` function to convert the XY pixel coordinates into real-world XYZ coordinates.

- The function also adjusts the coordinates using matrix multiplication to correct for the camera's orientation:

- Since the camera is angled upwards and to the side, the rotation matrix adjusts the coordinates back to the robot's frame of reference.

6. Final Decision:


- After detecting all the persons, the system calls ``deciding()`` (in ``get_route.py``) to select the person the robot should approach. The selection is typically based on criteria such as proximity or face orientation.

7. Returning the Coordinates:

- The final result is a string containing the person's name, real-world XYZ coordinates, and face angle (Yaw). This string is passed to the next step (Step 3) where the robot will use it to approach the person.

Function Call Flow in Step 2:


1. ``get_persons()`` (in ``get_persons_proccess.py``):
 - Main function responsible for detecting faces, recognizing them, and creating ``Person`` objects.
 - Calls the following functions:
 - ``recognize_faces_from_embeddings()``: This function detects faces in the image and matches them to known embeddings stored in ``face_embeddings.json``.
 - ``get_XYZ_from_pictures_with_file()``: This function calculates the real-world XYZ coordinates of the detected person using the depth image.
 - ``face_orientation_degrees()``: This function calculates the Yaw (face angle) of the person's face based on key facial landmarks.
2. ``recognize_faces_from_embeddings()`` (in ``get_persons_proccess.py``):
 - Called by: ``get_persons()``.
 - Uses the ``FaceAnalysis`` class from insightface to extract embeddings from the detected faces and compare them with ``face_embeddings.json`` to recognize known individuals.
3. ``get_XYZ_from_pictures_with_file()`` (in ``RS_API.py``):
 - Called by: ``get_persons()``.
 - Uses the RealSense API's built-in function ``rs2_deproject_pixel_to_point()`` to map XY pixel coordinates to real-world XYZ coordinates using the depth image.



4. `face_orientation_degrees()` (in `get_persons_process.py`):

- Called by: `get_persons()`.
- Calculates the face orientation (Yaw) using facial landmarks, which indicates the direction the person is facing.

5. `deciding()` (in `get_route.py`):

- Called by: `get_persons()` after all persons are detected.
 - Selects the person the robot should approach based on criteria such as proximity and face angle.
- 

Summary of Step 2:

1. Face Detection:

- The system detects faces in the captured RGB images using the `FaceAnalysis` class from the `insightface` library.
- It compares the detected faces to the known faces in `face_embeddings.json` using the function `recognize_faces_from_embeddings()`.

2. Person Selection:

- The `decide()` function will be used to select the person the robot will interact with, based on criteria like proximity or facing direction.

3. Calculating XYZ Coordinates:

- The system calculates the real-world coordinates (XYZ) of the detected person using depth data and intrinsic camera parameters loaded via `load_intrinsics()`.
- The coordinates are adjusted using matrix multiplication to account for the camera's orientation.

4. Dependencies:

- `face_embeddings.json`: Stores known face embeddings for comparison.
- `aligned_rgb_image_X.png` and `aligned_depth_image_m.png`: Captured RGB and depth images used for face detection and XYZ calculations.
- `intrinsics_file`: Contains the camera's intrinsic parameters for accurate 3D positioning.

Step 3: SSH into the robot, select ROS Foxy (1), set environment variables, and run start.py
echo "Start SSH"

Use the -t flag to force allocation of a pseudo-terminal, ensuring all output is shown
sshpass -p "123" ssh -tt -o PubkeyAuthentication=no unitree@192.168.123.18 << EOF
echo '1'
cd ~/Documents/project/from_lola/FINAL
python3 start.py "\$coordinates"
EOF

1. Object-Oriented Programming (OOP) is used to structure the way we interact with the robot. We create a robot object (from the `SportClient` class), and then apply various actions (move, rotate) to this object.

- Initialization:

- `my_robot = init_robot()` # Creates the robot object using OOP principles

- Here, `my_robot` is an instance of the `SportClient` class, which represents the robot and provides methods to control it.

2. Actions on the Robot Object:

- Once the robot object is created, we call various methods on it to make the robot move or rotate.

These methods include `goMeteresAhead()` and `rotate()`, both of which use UNITREE's built-in functions to control the robot.

- `goMeteresAhead()` uses the built-in `Move()` function from UNITREE's `SportClient` API. This function is called multiple times to send movement commands to the robot's motors, allowing it to move forward.

- UNITREE's `Move()` function is designed to take in velocity parameters and move the robot in a straight line.

- `rotate()` also uses the UNITREE API's `Move()` function, but this time to rotate the robot.

- The third parameter in `Move()` controls rotation. Positive values cause the robot to turn right, and negative values cause it to turn left:

- To compute the robot's turn while walking, we use $\cos(\alpha)$ based on vector multiplication.

Formula:

- $\cos(\alpha) = (\text{vector1} \cdot \text{vector2}) / (|\text{vector1}| * |\text{vector2}|)$.
- Vector1: Initial direction of the robot.
- Vector2: Target direction (where the person is).
- The dot product between these vectors gives the angle, and $\cos(\alpha)$ helps calculate how much the robot should turn.

Summary of Step 3):

1. SSH into the Robot:

- Establish a remote connection to the robot and select ROS Foxy as the environment.

2. OOP Approach:

- The robot is treated as an object using OOP principles. An instance of the SportClient class represents the robot, and we call methods like `goMetersAhead()` and `rotate()` on this object.

3. `goMetersAhead()` and `rotate()`:

- Both methods use the UNITREE API's `Move()` function multiple times to send movement and rotation commands to the robot:
 - `goMetersAhead()`: Moves the robot forward by a certain distance.
 - `rotate()`: Rotates the robot by a specified angle.

4. Walking Turns ($\cos \alpha$):

- When the robot needs to turn while walking, you calculate the turn using the formula $\cos(\alpha) = (\text{vector1} \cdot \text{vector2}) / (|\text{vector1}| * |\text{vector2}|)$, which determines the angle of rotation based on vector multiplication.

Table of Contents

Project
goal

Dependencies

Documentation

Script & flow

Using



Using

0. Initial Setup:

- Connect the computer to the robot using an Ethernet cable.
- Connect the camera on the robot to the computer using a USB3 cable.

1. Download the ZIP file:

Download the project ZIP file from the GitHub repository:

https://github.com/tallavi561/GO2_project.git

https://github.com/tallavi561/GO2_project

2. Unzip the File:

Extract the contents of the downloaded ZIP file:

```
unzip go2.zip
```

3. Transfer the `FROM_ROBOT` Directory:

Transfer the contents of the `FROM_ROBOT` directory to the robot in an appropriate directory, for example:

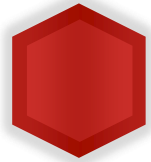
```
sshpass -p "123" scp -r FROM_ROBOT/ unitree@192.168.123.18:~/Documents/project/
```

4. Ensure All Dependencies Are Installed:

Ensure that all necessary dependencies are installed on both the robot and the computer.

On the robot: Make sure the robot has the required software

On the computer: Ensure that any necessary Python libraries (such as `insightface`, `pyrealsense2`, etc.) are installed.



Using

5. Run the Script:

Run the main script on your computer:

```
./main.sh
```

6. Permissions (if needed):

If the script doesn't run due to permission issues, give it execute permissions:

```
chmod +x main.sh
```

Future Work

1. DECIDE Function:

- Currently, the `DECIDE` function selects the first person detected in the image. The instructor has the flexibility to modify this function based on desired criteria for selecting a person.

2. Exiting SSH:

- After interacting with the robot, to allow further commands from the computer, exit the SSH session.

3. Voice Assistance:

- The ZIP file includes a script `voice.py`, which can be used to request assistance from the detected person.

- The voice can be played through a Bluetooth-connected speaker attached to the robot.