

Registration No.

OPSEC400



Autonomous UAV Landing in a Box on a Moving Platform

Yi Feng, Wenli Xu, Cong Zheng, & Stanley Baek (PI)
Intelligent Robotics Laboratory
Department of Electrical and Computer Engineering
University of Michigan - Dearborn

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

U.S. Army Tank Automotive Research,
Development, and Engineering Center
Detroit Arsenal
Warren, Michigan 48397-5000

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188		
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.					
1. REPORT DATE (DD-MM-YYYY) 14-12-2017		2. REPORT TYPE Final Report		3. DATES COVERED (From - To) 06-01-2017 to 14-12-2017	
4. TITLE AND SUBTITLE Autonomous UAV Landing in a Box on a Moving Platform Final Report			5a. CONTRACT NUMBER W56HZV-17-P-L507		
			5b. GRANT NUMBER N/A		
			5c. PROGRAM ELEMENT NUMBER N/A		
6. AUTHOR(S) Yi Feng, Wenli Xu, Cong Zheng, & Stanley Baek (PI) Intelligent Robotics Laboratory Department of Electrical and Computer Engineering University of Michigan - Dearborn			5d. PROJECT NUMBER N/A		
			5e. TASK NUMBER A001		
			5f. WORK UNIT NUMBER N/A		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) REGENTS OF THE UNIVERSITY OF MICHIGAN PROFESSOR PAUL R ICHARDSON 503 THOMPSON ST ANN ARBOR MI 48109-1340			8. PERFORMING ORGANIZATION REPORT NUMBER		
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) US Army RDECOM -TARDEC RD TA-RS/MS 263 Warren, MI 48397-5000			10. SPONSOR/MONITOR'S ACRONYM(S)		
			11. SPONSOR/MONITOR'S REPORT NUMBER(S)		
12. DISTRIBUTION / AVAILABILITY STATEMENT DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT This document reports the results of research performed by the Intelligent Robotics Laboratory (IRL) at the University of Michigan-Dearborn to investigate, evaluate, and validate the autonomous landing of unmanned aerial vehicle under the Autonomous UAV Landing in a Box on a Moving Platform project. The project plan envisioned the IRL to develop and evaluate a prototype quadrotor for visual sensing of the environment, investigate algorithms for flight control, navigation, planning, target tracking, and sensor fusion. This report details hardware configuration of quadrotor, software architecture, experimental results of target tracking and autonomous landing on a stationary platform, discussions, and future work.					
15. SUBJECT TERMS					
16. SECURITY CLASSIFICATION OF: Unclassified			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE	Unclassified	79	Neil Cooper
Unclassified	Unclassified	Unclassified			19b. TELEPHONE NUMBER (include area code) (586) 282-4892

Autonomous UAV Landing in a Box on a Moving Platform

Final Report

December 14, 2017

Prepared for

U.S. Army TARDEC

U.S. Army Research, Development and Engineering Command (RDECOM)

Prepared by

Yi Feng, Wenli Xu, Cong Zheng, & Stanley Baek (PI)

Intelligent Robotics Laboratory

Department of Electrical and Computer Engineering

University of Michigan - Dearborn

1 Introduction

This document reports the results of research performed by the Intelligent Robotics Laboratory (IRL) at the University of Michigan-Dearborn to investigate, evaluate, and validate the autonomous landing of unmanned aerial vehicle under the Autonomous UAV Landing in a Box on a Moving Platform project. The project plan envisioned the IRL to develop and evaluate a prototype quadrotor for visual sensing of the environment, investigate algorithms for flight control, navigation, planning, target tracking, and sensor fusion. This report details hardware configuration of quadrotor, software architecture, experimental results of target tracking and autonomous landing on a stationary platform, discussions, and future work.

2 Hardware configuration

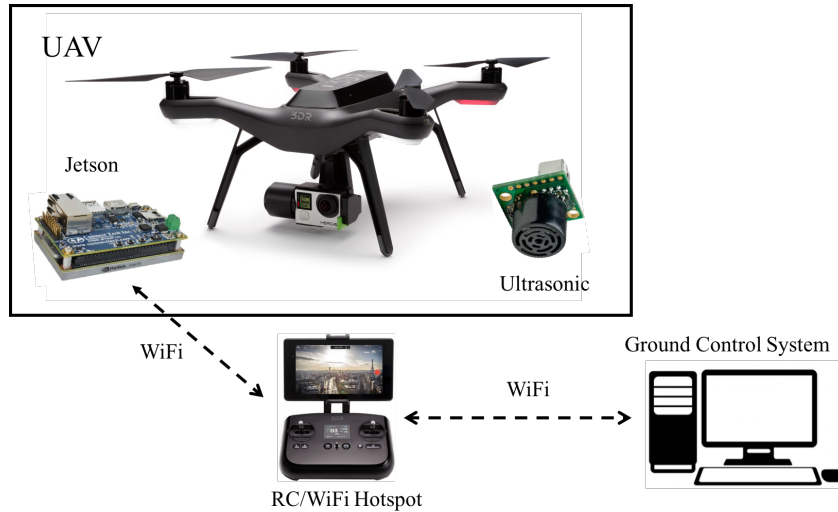


Figure 1: Aircraft equipment and ground station configuration

2.1 UAV Platform

The UAV platform used in this work is 3DRobotics Solo quadrotor. It is equipped with a flight control system (Pixhawk 2.1 green cube), a 9-axis inertial measurement unit, and a GPS receiver. The accompanying remote controller (RC) provides WiFi hotspot for the drone as well as ground control system (GCS).

2.2 Peripherals

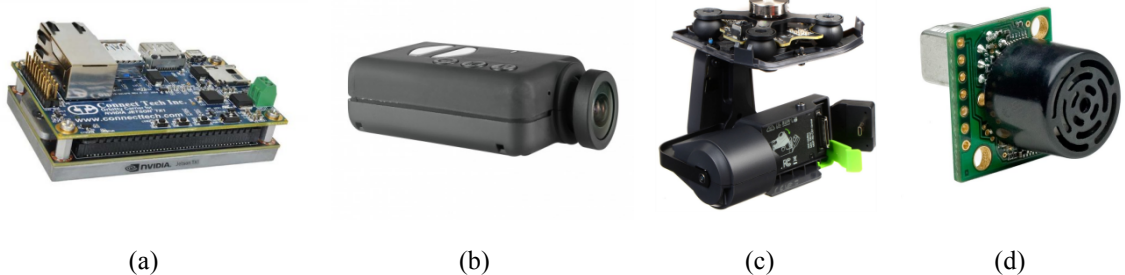


Figure 2: UAV peripherals. (a) NVIDIA Jetson TX1 (b). Mobius Action camera. (c) 3DR Solo gimbal. (d) Maxbotix ultrasonic distance sensor.

- **Onboard Computer:** NVIDIA Jetson TX1 is an embedded computer with Quad-core 64-bit ARM A57 Processor and 256 NVIDIA CUDA cores. This lightweight and high-performance processor is suitable for an onboard control processing unit for small UAVs.
- **Camera:** The UAV is equipped with a Mobius ActionCam camera. The camera can provide 1080P videos at up to 30 FPS or 720P at up to 60 FPS. The USB interface on the camera provides low latency video transmission to the onboard processor for time-critical vision process.
- **Gimbal:** The 3 degrees-of-freedom (DOF) gimbal provides yaw ranges of $\pm 30^\circ$, pitch ranges from $+15^\circ$ to -90° , and roll ranges of $\pm 15^\circ$.
- **Distance sensor:** We integrated an ultrasonic distance sensor (Maxbotic MB1212) to measure the distance between the UAV and the mobile landing platform. It provides an I2C bus interface and measurement ranges from 25cm to 765cm with 1cm resolution at 10Hz.

2.3 Ground station

The main tasks of the ground station include a remote access to the onboard computer and real-time aircraft monitoring. Using the Solo Wi-Fi hotspot, we can remotely access the onboard computer via Secure Shell (SSH) to run ROS modules to initiate, monitor, and abort flight missions.

3 Software Configuration

In this work, we have employed Robot Operating System (ROS) as a main software framework since ROS provides commonly-used functionalities in robotic software, low-level device control libraries, and seamless implementation of multithreading across heterogeneous computer units. The operating system-like functionalities in ROS can significantly reduce software developer's efforts on multithread operations across multiple workstations. Depicted in Figure 3 are the software modules running on the onboard computer (Jetson TX1) and the workstation on the ground (NUC6i7).

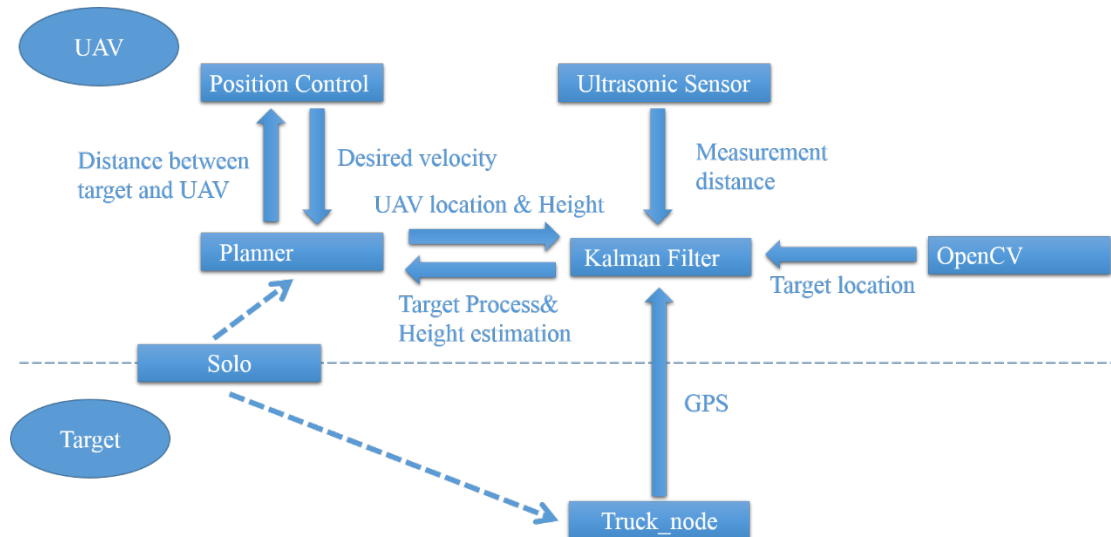


Figure 3: Software architecture

The 3DR Dronekit provides application programming interface (API) to communicate with the 3DR Solo autopilot, Pixhawk, using the MAVLink unmanned aerial vehicle communication protocol. We have selected Dronekit-Python among other possible APIs, such as mavros, because it provides Python interface, which is easy to use and can be seamlessly integrated with ROS.

3.1 Planner ROS Package

Planner is a ROS package in Python we developed for a high-level navigation and flight behavior control for the UAV. It consists of four ROS nodes: Planner, Truck, AltitudeEstimator, and TargetTracking.

3.1.1 Planner node

The Planner node is the main coordinator for the entire flight mission from takeoff to landing. The heart of the node is a finite state machine (*Figure 4*) triggered by a ROS timer running at 20 Hz. At each state, it focuses only on designated tasks defined in the routine and transitions to the next state if certain conditions are met. For example, at the FlyingToTarget state, the task for the drone is simply to fly to the GPS location sent from the landing platform. If it arrives near the target location, the state transitions to the SearchingForTarget state. Otherwise, it stays at the FlyingToTarget state to keep processing the current tasks. In this approach, it is very easy to add new tasks and remove obsolete/unnecessary tasks. It is also easy for code maintenance.

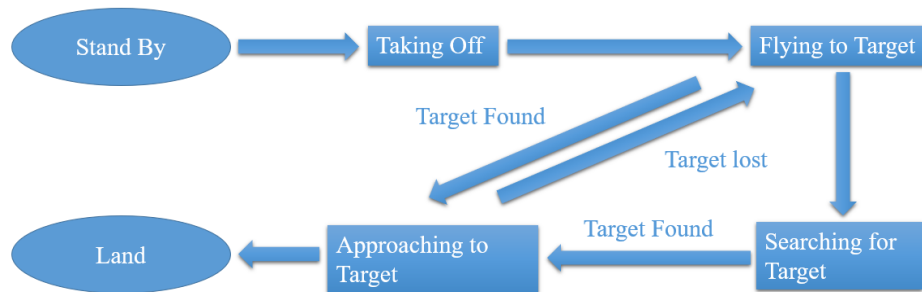


Figure 4: Finite state machine

3.1.2 Truck node

The main function of the Truck node is to provide the GPS location of mobile landing platform. This node, running on a computer on the landing platform with a GPS receiver, publishes the GPS data via a ROS message. Then, the onboard processing unit receives the message and provide the data to the sensor fusion center for target localization.

3.2 Sensor fusion for optimal estimation

We have implemented sensor fusion algorithms based on extended Kalman filter to estimate altitude of the drone and the mobile target location.

3.2.1 Altitude Estimator

Accurate measurements of drone's altitude (or the distance from the landing platform) are important for autonomous landing. However, the altimeter came with 3DR Solo is not designed to accurately measure the altitude of the drone when the drone is near the ground surface (less than 10 meters above the ground). Moreover, the altimeter has a large amount of sensor drift depending

on weather conditions. In order to compensate this sensor drift, we have integrated an ultrasonic distance sensor. The ultrasonic sensor, however, has a limited sensing range of less than 2 meters. Therefore, we have implemented a sensor fusion algorithm to estimate the distance between the drone and the landing platform.

To achieve this, we have constructed a state space model given by

$$\begin{bmatrix} h(k+1) \\ \dot{h}(k+1) \\ \theta(k+1) \end{bmatrix} = \begin{bmatrix} 1 & \Delta T & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} h(k) \\ \dot{h}(k) \\ \theta(k) \end{bmatrix} + \mathbf{w}(k)$$

$$\begin{bmatrix} z_u \\ z_c \\ z_b \\ z_v \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} h(k) \\ \dot{h}(k) \\ \theta(k) \end{bmatrix} + \mathbf{v}(k)$$

where h and $\dot{h}$ are, respectively, the altitude of the drone and its time derivative (or vertical velocity), θ is the sensor drift of the barometer, z_u is the ultrasonic measurement, z_c is the vertical distance between drone and landing platform calculated by a computer vision algorithm, z_b is the barometer measurement, z_v is the vertical speed measurement, ΔT is the time step, $\mathbf{w}$ is the zero mean Gaussian random noise with covariance of Q , and $\mathbf{v}$ is the zero mean Gaussian random noise with covariance of R . Here, $\mathbf{w}$ and $\mathbf{v}$ represent, respectively, the system uncertainty and the sensor noise, and they are independent.

3.2.2 Target Tracking

We have implemented a target tracking method based on Extended Kalman filter to accurately estimate the location of the mobile landing platform. The state-space model for target tracking is given by

$$X_{hat(k+1)} = \begin{bmatrix} x(k+1) \\ y(k+1) \\ v_x(k+1) \\ v_y(k+1) \end{bmatrix} = \begin{bmatrix} 1 & 0 & \Delta T & 0 \\ 0 & 1 & 0 & \Delta T \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x(k) \\ y(k) \\ v_x(k) \\ v_y(k) \end{bmatrix} + \mathbf{w}(k)$$

$$\begin{bmatrix} z_{px}(k) \\ z_{py}(k) \\ z_{vx}(k) \\ z_{vy}(k) \\ z_{cx}(k) \\ z_{cy}(k) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x(k) \\ y(k) \\ v_x(k) \\ v_y(k) \end{bmatrix} + \mathbf{v}(k)$$

where x and y are, respectively, the lateral and longitudinal location of the landing platform with respect to the predefined home position, v_x and v_y are, respectively, the lateral speed and longitudinal speed of the landing platform, z_{px} and z_{py} are, respectively, the landing platform's lateral and longitudinal location calculated from its GPS module, z_{vx} and z_{vy} are, respectively, the measurement of the landing platform's velocity, z_{cx} and z_{cy} are, respectively, the landing platform's position computed by a vision-based target detection algorithm, ΔT is the time step, $\mathbf{w}$ is the zero mean Gaussian random noise with covariance of Q , and $\mathbf{v}$ is the zero mean Gaussian random noise with covariance of R . Here, $\mathbf{w}$ and $\mathbf{v}$ represent, respectively, the system uncertainty and the sensor noise, and they are independent.

Since Kalman filters can provide prediction of state variable before observations arrive, we can predict the target location at the next time step. In this way, the drone can fly toward the predicted location instead of the estimated location at the current time step. Therefore, we can avoid the drone flying behind the target.

3.3 Visual Target Tracking



Figure 5: (a) A marker with ID equals to 213. (b) Sample of detecting both big and small markers.

Detecting and identifying a mobile landing platform using electro-optical sensors is not an easy task mainly due to the computational complexity of object recognition in computer vision. To reduce the computational loads for computer vision processing, we have implemented a marker-based augmented reality using the OpenCV library to detect a marker secured on the mobile landing platform. As shown in Figure 5 (a), our marker is a modified version of the Hamming code with 10 bits for each ID, so we can detect up to 1024 different makers.

We can estimate the marker pose from the following routine.

1. Converting to grayscale and binarization: Images captured by the onboard camera are transformed into black-and-white images.
2. Contour detection: This process finds all contours in the image.
3. Filtering: We find rectangular contours with a size greater than a predefined dimension.
4. Decoding: We calculate the marker ID through Hamming code decoding process and decide if this candidate marker is what we are looking for. As shown in *Figure 5* (b), we successfully detected the markers.
5. Pose Estimation: We calculate the position and attitude of marker in the NED coordinates.

3.4 Ultrasonic Distance Sensor

We use an ultrasonic sensor to measure the height of UAV because the barometer of the UAV has a bias which causes the measurement of height is not accuracy. The sensor also provides a sensitive measurement which can help the height control more smoothly. Now, we are using MB1212 I2CXL-MaxSonar-EZ1 ultrasonic sensor (I2CXL-MaxSonar-EZ Products, 2017). The ultrasonic sensor is working at the I2C bus of TX1 with 10Hz. The control code is using smbus2 package

which is pure Python implementation of the python-smbus package (smbus2 0.1.2, 2017) and running as a ROS node which will publish distance message every 0.1s (10Hz).

4 Experiments and Analyses

4.1 Visual Target Tracking

We have used a Vicon real-time motion tracking system to test the performance of vision-based target detection algorithm discussed in Section 3.3. Shown in Figure 6 are the distance between a marker and the onboard gimbaled camera and the orientation of the marker with respect to the camera frame. Sometimes, we have measurement errors of less than 20 cm due to numerical errors and computational delays. It can be compensated with a sensor fusion algorithm.

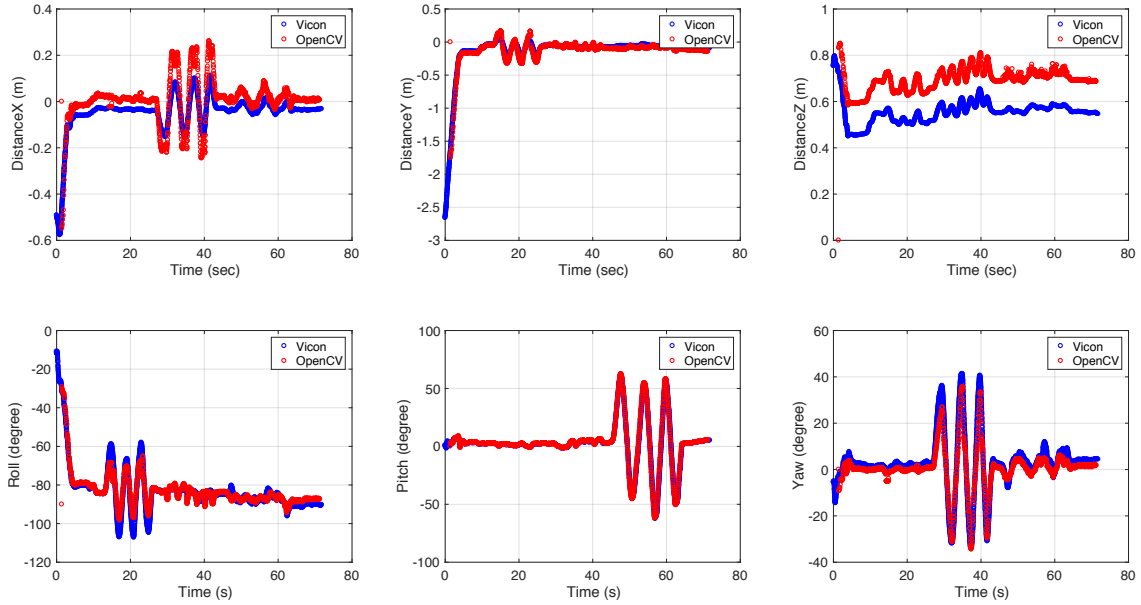


Figure 6: Performance of visual target tracking compared with Vicon real-time motion tracking system. The top three plots show the distance between camera and marker and the bottom three plots show the orientation with respect to the marker.

4.2 Kalman Filter

Figure 7 shows the altitude estimate produced by the sensor fusion method discussed in Section 3.2.1. The ultrasonic measurements (red dot) have many outliers when the distance is greater than 2 meters while the barometer measurements (green dot) have a sensor drift especially when the drone is at a low altitude. The vision-based target detection (light blue) produce the most accurate and precise measurements if the target is in the field of view. Note that the target detection produces no measurement at the beginning in Figure 7. The optimal altitude estimates are shown in dark blue.

Shown in Figure 8 are the GPS measurements of the landing platform, visual target localization, and Kalman filter estimates of mobile landing platform. When the landing platform is inside the camera's field of view, the estimates rely more on the computer vision data than the GPS data.

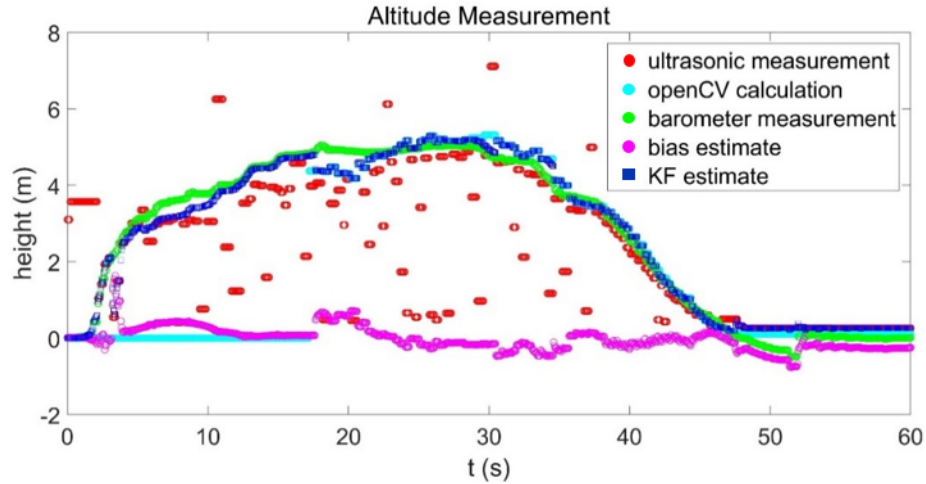


Figure 7: Altitude measurements and estimates by Kalman Filter.

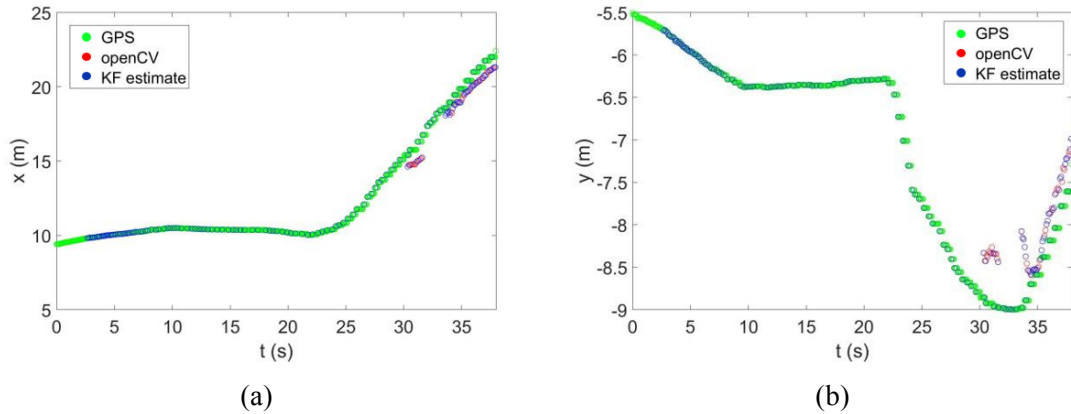


Figure 8: GPS measurements and Kalman filter estimate. (a) East. (b) North

4.3 Target Following and Autonomous Landing

Shown in Figure 9 are the flight paths of a quadcopter flying to and landing on the target platform. At the beginning of the experiment, the target localization relies only on the GPS measurements sent from the landing platform. As the drone approaches the target location, the onboard camera can detect the marker on the landing platform and the sensor fusion algorithm makes use of the visual target tracking as well as the GPS measurements for target localization. We have achieved autonomous landing on a stationary platform with an average error of 25 cm.

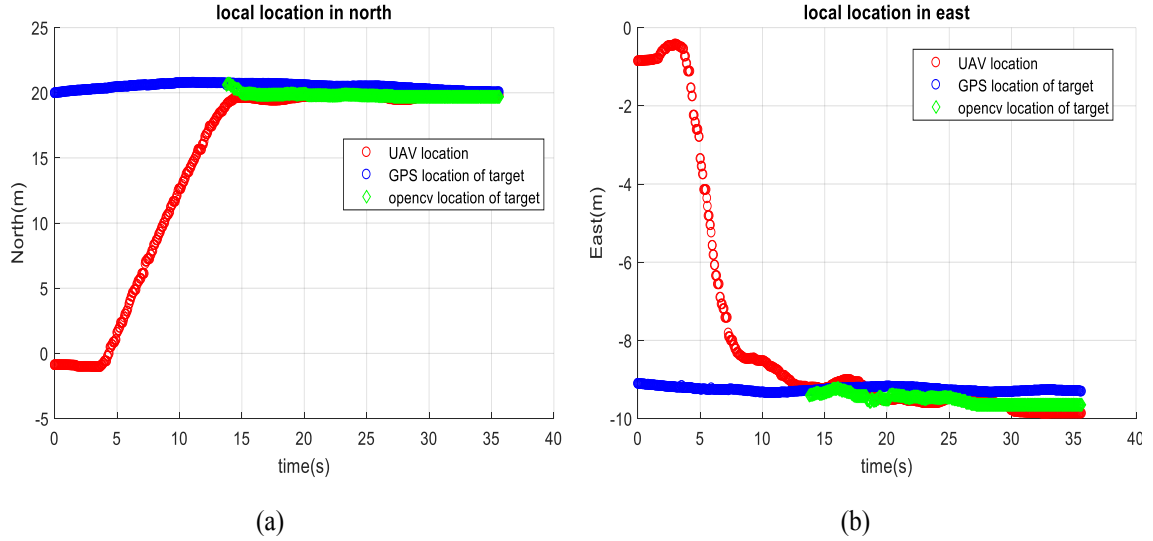


Figure 9: Autonomous landing on a stationary target. (a) north. (b) east.

Figure 10 shows the performance of target following without landing. The drone uses the GPS measurements and the visual target tracking data to update the control input and successfully follows the target.

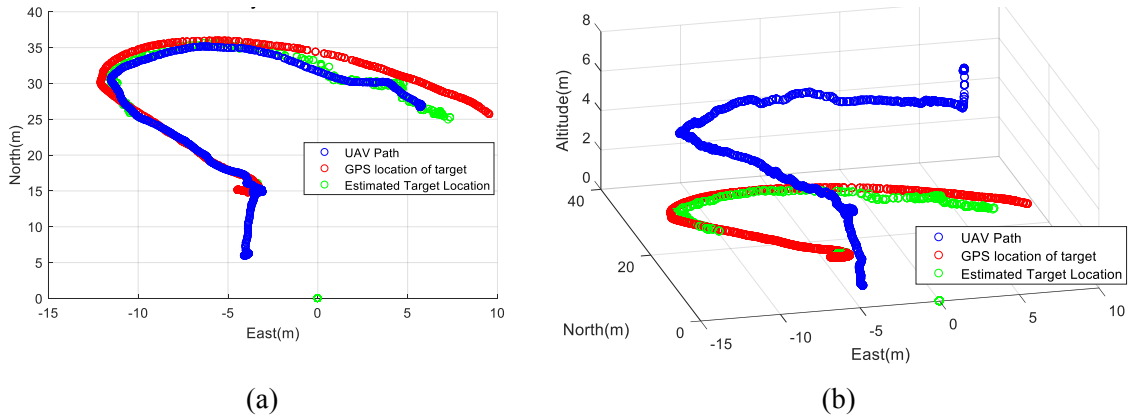


Figure 10: Following a target moving at 5 m/s. (a) 2-dimensional. (b) 3-dimensional.

5 Discussions

5.1 GPS

The GPS update rate of 3DR Solo is only 5 Hz, which is not enough for robust position control of the drone. It is evident that in this work the distance between the drone and the landing platform is more important than their absolute positions. The distance can be calculated based on the images captured by the gimbaled camera using computer vision libraries. However, the vision-based target localization is not robust enough for precise landing mainly due to limited field of view. Therefore, it is necessary to calculate the distance using the GPS measurements of the drone and the landing platform, this calculation along with the vision-based computation can be sent to a sensor fusion center to optimally estimate the target location.

5.2 Gimbal

Solo provides no way of measuring of gimbal angles, which makes the visual tracking algorithm very complicated for target localization and orientation estimates. Still, we have tried to estimate both distance and orientation of the drone relative to the marker with an assumption that the marker is laid down without tilting. However, the computational errors become augmented throughout computations, and the resulting errors sometimes become about 20 cm. For this reason, we set the gimbal pitch to -90 degrees in the inertial reference frame for the entire flight experiments. In this way, the gimballed camera always points down at -90 degrees regardless of the drone orientation – it is controlled by the built-in autopilot in 3DR Solo. However, this approach introduces another problem – it is now easy to lose target because the effective field of view of the camera is significantly reduced.

5.3 Altitude Measurements

The barometer of 3DR Solo has an unacceptable drift to measure the altitude of the drone for autonomous landing. Because of the drift, it was very difficult to stabilize the altitude of the drone, which made the onboard camera difficult to detect the target. Therefore, we have integrated an ultrasonic distance sensor to compensate this issue. However, the measurements of the ultrasonic distance sensors become very noisy when the distance is greater than 2 meters.

As discussed in Section 3.2.1, we have developed a sensor fusion algorithm that estimates the barometer bias as well as altitude of the drone. In the sensor fusion algorithm, the estimation relies more on ultrasonic measurements at low altitudes while it relies more on bias-fixed barometer measurements at high altitudes by adaptively adjusting the covariance matrix of measurement noise.

The altitude error without the sensor fusion algorithm was sometimes greater than 5 – 10 meters, but it became less than 10 cm at a low altitude where a camera can detect the target. Once the camera measurements were also fused, the altitude error was usually less than 5 cm, which was good enough for landing.

5.4 Computer Vision Processing

The contour detection algorithm discussed in Section 3.3 is computationally the most expensive process in the vision-based target tracking we have developed in this work, and it takes about a half of the overall processing time. Although its computational speed is feasible in controlled environments, such as indoor flight test environments, it becomes dramatically slow in a cluttered environment with a variety of objects – grass, buildings, mobile landing platform, etc. The current processing rate we have is ~ 8-10 frames per second.

5.5 Aircraft

The weight of 3DR Solo is only 1.5 kg and it seems it is very susceptible to disturbances, which requires more control effort. As a comparison, DJI Matrice 100 weighs 3.3 kg, and we have observed that this aircraft is less susceptible to disturbances.

6 Future opportunities

6.1 Aircraft

When 3DR released Solo in 2015, it was marketed at hobbyists who are interested in aerial photography. Although there exist software development kits (SDKs) for Solo (developed by 3DR

and hobbyists), we have discerned that the drone platform itself has a number of limitations for this project, such as peripheral expandability, sensor accuracy and bandwidth, and payload.

There are many drones designed for researchers and developers, such as DJI Matrice 100, AscTec Neo, AscTec FireFly. These drones have a few advantages for autonomous landing compared to 3DR Solo. The advantages include high-bandwidth sensor data, large developer's communities, easy integration of variety of sensors, customization, etc.

6.2 Sensors and Peripherals

- **LIDAR:** LeddarTech's solid-state LiDAR products would be the right choice for the distance measurement between the drone and ground/landing platform. They have a higher resolution with a bandwidth of 100 Hz. It is worthy to note that barometers are not really designed for precise measurement at low altitude and ultrasonic sensors become very noisy for uneven surfaces such as a grass area.
- **DJI Guidance:** DJI Guidance is an obstacle detection sensor module consisting of a vision processing core, stereo cameras, and ultrasonic sensors. It can accurately measure the distance between the UAV and the landing platform using its built-in sensor fusion algorithm running at 50 Hz.
- **Gimbal:** It is required to have a gimbal that can provide orientation measurements with low latency for real-time visual target tracking and localization. Providing wide angle rotations as well as three degrees of freedom rotation independent of the drone orientation is essential for real-time target tracking.

6.3 Software

- **Kalman Filter:** The most arduous step in the implementation of a Kalman filter is finding the elements of the **Q** and **R** matrices, which are the most influential in the estimation performance. The variances of sensor noise (the elements in the **R** matrix) can be measured from GPS and an inertial measurement unit (IMU) whereas the variances of system noise (elements in the **Q** matrix) are not easy to determine. Data collection from flight experiments in various conditions are necessary to find the KF parameters that can provide the optimal estimates of target location and drone states.
- **Computer Vision:** We can improve the processing speed of visual target tracking by utilizing GPUs and parallel processing. Jetson TX1 comes with a parallel computing platform and application programming interface called CUDA that allows developers to use a CUDA-enabled GPU for general purpose processing. We expect it will take 8 – 12 months to implement computer vision algorithm with this API along parallel processing.

7 Bill of Materials

Part Name	Description	Price
3DR Solo	Quadcopter platform	\$250.00
NVIDIA Jetson TX1 Module	Onboard processing unit	\$399.00
ConnectTech Orbitty Carrier	Jetson TX1 module carrier board	\$188.00
MikroTik R11E-2HND	Solo WiFi upgrade cards (for drone and RC)	\$49.00
mRo GPS u-Blox Neo-M8N	Solo GPS upgrade	\$69.00

Mobius Action camera	Record video in super sharp 1080p HD resolution with USB port	\$69.95
3DR Solo gimbal	3-axis gimbal	\$359.00
I2CXL-MaxSonar-EZ sensors	Ultrasonic distance sensor with I2C interface	\$44.95

8 Acknowledgement

We would like to express our special thanks of gratitude to Space and Naval Warfare Systems Command (SPAWAR) as well as the Distributed Aerospace Systems and Control Laboratory at the University of Michigan – Ann Arbor, which helped us in hardware/software interface.

Appendix

1 Software Installation

To prepare a Jetson TX1, we need to install Dronekit, ROS, and OpenCV.

1.1 Installation of *Dronekit_python*

- a. Go to http://python.dronekit.io/guide/quick_start.html
- b. Follow the steps in the webpage to install Dronekit_Python:

```
$ sudo apt-get install python-pip python-dev
$ sudo pip install dronekit
```

1.2 Installation of *ROS kinetic*

- a. Go to <http://wiki.ros.org/Installation/UbuntuARM>
 - b. Click on kinetic and follow the steps in the webpage to install ROS kinetic:
- ```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" >
/etc/apt/sources.list.d/ros-latest.list'
$ sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-key
421C365BD9FF1F717815A3895523BAEEB01FA116
```

```
$ sudo apt-get update
$ sudo apt-get install ros-kinetic-desktop
$ sudo c_rehash /etc/ssl/certs
$ sudo rosdep init
$ rosdep update
$ echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```

```
$ sudo apt-get install python-rosinstall python-rosinstall-generator python-wstool build-essential
```

#### 1.3 Configuration of *ROS*

- a. Create catkin\_ws

```
$ source /opt/ros/kinetic/setup.bash
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/
$ catkin_make
$ source devel/setup.bash

$ cd src

$ echo "source /home/ubuntu/catkin_ws/devel/setup.bash" >> ~/.bashrc
```

```
$ sudo su
```

```
$ cd ~
```

```
$ echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
```

```
$ echo "source /home/ubuntu/catkin_ws/devel/setup.bash" >> ~/.bashrc
```

- b. Copy the source files to ~/catkin\_ws/src

- c. Compile the code

```
$ roscore &
```

```
$ cd ~/catkin_ws
```

```
$ source devel/setup.bash
```

```
$ catkin_make
```

#### **1.4 Installation of OpenCV 3.2.0**

- a. Go to [http://docs.opencv.org/3.2.0/d6/d15/tutorial\\_building\\_tegra\\_cuda.html](http://docs.opencv.org/3.2.0/d6/d15/tutorial_building_tegra_cuda.html)

- b. Follow the step in the webpage to install OpenCV

```
$ mkdir ~/opencv_ws
```

```
$ cd ~/opencv_ws
```

```
$ git clone https://github.com/opencv/opencv.git
```

```
$ cd opencv
```

```
$ git checkout -b v3.1.0 3.1.0
```

```
$ git cherry-pick 10896
```

# sometimes here is an error. just type git commit -m "xxxxxxx" (manually commit) would be OK.

```
$ git cherry-pick cdb9c
```

```
$ git cherry-pick 24dbb
```

```
$ cd ~/opencv_ws
```

```
$ git clone https://github.com/opencv/opencv_extra.git
```

```
$ cd opencv_extra
```

```
$ git checkout -b v3.1.0 3.1.0
```

```
$ sudo apt-add-repository universe
```

```
$ sudo apt-get update
```

```
$ sudo apt-get install \
```

```
 libglew-dev \
```

```
 libtiff5-dev \
```

```
 zlib1g-dev \
```

```
 libjpeg-dev \
```

```
libpng12-dev \
libjasper-dev \
libavcodec-dev \
libavformat-dev \
libavutil-dev \
libpostproc-dev \
libswscale-dev \
libeigen3-dev \
libtbb-dev \
libgtk2.0-dev \
pkg-config
$ sudo apt-get install python-dev python-numpy python-py python-pytest
$ cd ~/opencv_ws/opencv
$ mkdir build
$ cd build
```

c. Configuring OpenCV for Building

```
$ cmake \
-DCMAKE_BUILD_TYPE=Release \
-DCMAKE_INSTALL_PREFIX=/usr \
-DBUILD_PNG=OFF \
-DBUILD_TIFF=OFF \
-DBUILD_TBB=OFF \
-DBUILD_JPEG=OFF \
-DBUILD_JASPER=OFF \
-DBUILD_ZLIB=OFF \
-DBUILD_EXAMPLES=ON \
-DBUILD_opencv_java=OFF \
-DBUILD_opencv_python2=ON \
-DBUILD_opencv_python3=OFF \
-DENABLE_PRECOMPILED_HEADERS=OFF \
-DWITH_OPENCL=OFF \
-DWITH_OPENMP=OFF \
-DWITH_FFMPEG=ON \
-DWITH_GSTREAMER=OFF \
-DWITH_GSTREAMER_0_10=OFF \
-DWITH_CUDA=ON
```

```
-DWITH_GTK=ON \
-DWITH_VTK=OFF \
-DWITH_TBB=ON \
-DWITH_1394=OFF \
-DWITH_OPENEXR=OFF \
-DCUDA_TOOLKIT_ROOT_DIR=/usr/local/cuda-8.0 \
-DCUDA_ARCH_BIN=5.3 \
-DCUDA_ARCH_PTX="" \
-DINSTALL_C_EXAMPLES=ON \
-DINSTALL_TESTS=OFF \
-DOPENCV_TEST_DATA_PATH=./opencv_extra/testdata \
$ make -j4
$ make install
```

## 2 Standard Operating Procedure

### 2.1 Preparation

Here are the steps to set up ROS communications between 2 machines.

In our case, we set Jetson TX1 on Solo as master, and NUC6i7 as slave.

NUC6i7: hostname: NUC6i7, IP: 10.1.1.183

Jetson TX1: hostname: tegra-ubuntu152, IP: 10.1.1.152

- 1) Connect NUC6i7 and Jetson TX1 to the same wireless network (Solo WIFI of controller).
- 2) On NUC6i7:
  - a. Open a new terminal and type “sudo gedit /etc/hostname”. Then change hostname into " NUC6i7 ".
  - b. Type “ sudo gedit /etc/hosts” in terminal and modify into this:

|            |                 |
|------------|-----------------|
| 127.0.0.1  | localhost       |
| 127.0.1.1  | NUC6i7          |
| 10.1.1.183 | NUC6i7          |
| 10.1.1.152 | tegra-ubuntu152 |
  - c. sudo /etc/init.d/networking restart
  - d. sudo apt-get install chrony openssh-server
  - e. gedit .bashrc and add these in the bottom  
export ROS\_HOSTNAME=`hostname`

```
export ROS_MASTER_URI=http:// tegra-ubuntu152:11311
```

- f. `sudo su & gedit .bashrc` and add these in the bottom on root:

```
export ROS_HOSTNAME='hostname'
```

```
export ROS_MASTER_URI=http:// tegra-ubuntu152:11311
```

- g. `source ~/.bashrc`

3) On Jeston TX1:

- a. Open a new terminal and type “`sudo gedit /etc/hostname`”. Then change the hostname into " tegra-ubuntu152 "

- b. Type “`sudo gedit /etc/hosts`” and modify into this:

```
127.0.0.1 localhost
```

```
127.0.1.1 tegra-ubuntu152
```

```
10.1.1.183 NUC6i7
```

```
10.1.1.152 tegra-ubuntu152
```

- c. `sudo /etc/init.d/networking restart`

- d. `sudo apt-get install chrony openssh-server`

- e. `gedit .bashrc` and add these in the bottom

```
export ROS_HOSTNAME='hostname'
```

```
export ROS_MASTER_URI=http:// tegra-ubuntu152:11311
```

- f. `sudo su & gedit .bashrc` and add these in the bottom on root:

```
export ROS_HOSTNAME='hostname'
```

```
export ROS_MASTER_URI=http:// tegra-ubuntu152:11311
```

- g. `source ~/.bashrc`

## 2.2 *Operating Procedure*

- 1) Connect all JestonTX1 and NUC6i7 with Solo WIFI from controller

- 2) On NUC6i7: open a new terminal and ssh JestonTX1\_UAV. Then,

- a. `sudo vi /etc/hosts`

- b. `sudo date --set="$(ssh ubuntu@10.1.1.184 date)"`

- 3) On NUC6i7: open a new terminal and ssh JestonTX1\_UAV. Then,

- a. `roscore`

- 4) On NUC6i7: open a new terminal and ssh JestonTX1\_target. Then,

- a. `sudo vi /etc/hosts`

- b. `sudo date --set="$(ssh ubuntu@10.1.1.184 date)"`

- 5) On NUC6i7: open a new terminal and ssh JestonTX1\_target. Then,
  - a. `roslaunch planner truck.py`
- 6) On NUC6i7: open a new terminal and ssh JestonTX1\_UAV. Then,
  - a. `sudo su`
  - b. `roslaunch ultrasonic_sensor I2C_ultrasonic_test.py`
- 7) On NUC6i7: open a new terminal and ssh JestonTX1\_UAV. Then,
  - a. `roslaunch opencv opencv_node.py`
- 8) On NUC6i7: open a new terminal and ssh JestonTX1\_UAV. Then,
  - a. `roslaunch ~/catkin_ws/src/Drone_rosbag.launch`
- 9) On NUC6i7: open a new terminal and `roslaunch ~/catkin_ws/src/NUC_image_view.launch`
- 10) On NUC6i7: open a new terminal and `roslaunch image_view video_recorder image:=/opencv/grayImage_marker_max_depth_range:=0"`
- 11) On NUC6i7: open a new terminal and `roslaunch user_control user_command_node.py`

```

1 # Solo code
2
3 #####
4 # launch file
5 #####
6
7 # Drone_rosbag.launch
8 <launch>
9
10 <node pkg="planner" type="Planner.py" name="Planner" output="screen">
11 </node>
12 <node pkg="planner" type="AltitudeEstimator.py" name="AltitudeEstimator" output="screen">
13 </node>
14 <node pkg="planner" type="TargetTracking.py" name="TargetTracking" output="screen">
15 </node>
16 <node pkg="opencv" type="opencv_node.py" name="opencv_node" output="screen">
17 </node>
18
19 <node pkg="rosbag" type="record" name="rosbag_record_solo"
20 args="-o /home/cong/.ros/SoloBag
21
22 /AltitudeEstimator/Altitude
23 /TargetTracking/TargetLocation
24
25 /solo/attitude
26 /solo/location_GPS
27 /solo/location_local
28 /solo/velocity
29
30 /planner/truck_velocity
31 /planner/state
32 /truck/local_location
33 /opencv/marker_location
34
35 /truck/attitude
36 /truck/ground_speed
37 /truck/location_GPS
38 /truck/velocity
39
40 /opencv/info
41
42 /solo/ultrasonic_sensor
43 /ultrasonic/long_distance" >
44 </node>
45
46 </launch>
47
48 # *****
49 # NUC_image_view.launch
50 <launch>
51 <node pkg="image_view" type="image_view" name="image_view" respawn="false"
52 output="screen">
53 <remap from="image" to="/opencv/grayImage_marker"/>
54 </node>
55 <node pkg="planner" type="guiTest.py" name="guiTest" output="screen">
56 </node>
57 </launch>
58
59 #####
60 # opencv code
61 #####
62
63 # marker.py
64
65 #!/usr/bin/env python2
66 # -*- coding: UTF-8 -*-
67
68 """
69 This class is to record marker information: id and pixel points.
70 """
71 class marker:
72
73 def __init__(self):
74 self.markerId = -1
75 # self.markerPoints = []
76
77
78 def saveMarker(self, inMarker, inMarkerId):
79 self.markerId = inMarkerId
80 self.markerPoints = inMarker
81
82 #####
83 # OpencvLib.py

```

```

84
85 #!/usr/bin/env python2
86 # -*- coding: UTF-8 -*-
87
88 import numpy as np
89 import cv2
90 import math
91 import time
92 import marker
93
94
95 """
96 class: OpencvLib
97 Mainly collect the useful opencv functions to detect markers,
98 recognize the id, calculate the pose and position.
99 """
100 class OpencvLib(object):
101
102 # Constructor.
103 def __init__(self):
104
105 # camera parameters
106 horizontalResolution = 640
107 verticalResolution = 480
108
109 # Algorithm parameters
110 self.minContourPerimeter = verticalResolution/5
111 self.minContourLengthAllowed = 100
112 # self.minContourPerimeter = verticalResolution/4
113 # self.minContourLengthAllowed = 120
114
115
116 """
117 Marker transfer flags
118 """
119 self.marker213 = marker.marker()
120 self.marker456 = marker.marker()
121 self.isMarker213Detected = False
122 self.isMarker456Detected = False
123 self.currentMarkerFlag = 213
124
125
126 """
127 Decide what kind of camera len and marker size we used.
128 See decide_obj_points & decide_cam_lens functions for more info.
129 """
130 self.markerSizeDecider = 'big'
131 self.cameraLensDecider = 'normal'
132
133
134 """
135 Debug data.
136 """
137 self.camShotTime = 0
138 self.camShotTimeHMS = time.strftime("%H:%M:%S")
139 self.lastImageMsgTime = 0
140
141
142 """
143 Control flags
144 """
145 # target lost flag
146 self.targetLostCount = -1
147 self.isTargetLost = True
148
149 self.numOfMarkerDetected = 0
150 self.isAnyTargetDetected = False
151
152
153 """
154 Estimated position & attitude data
155 """
156 self.marker_attitude_yaw = 0
157
158 self.distance_camera_x = 0
159 self.distance_camera_y = 0
160 self.distance_camera_z = 0
161 self.distance_target_x = 0
162 self.distance_target_y = 0
163 self.distance_target_z = 0
164 self.distance_north = 0
165 self.distance_east = 0
166 self.distance_down = 0

```

```

167
168 self.camera_Euler_roll_rads = 0
169 self.camera_Euler_pitch_rads = 0
170 self.camera_Euler_yaw_rads = 0
171
172 self.gimbal_pitch_current = 0
173 self.gimbal_pitch_delta = 0
174 self.gimbal_pitch_desired = 0
175
176 # distance_NED correction
177 # if 213 is used, deducted the center difference
178 self.distance_correction = 0.162
179
180
181 # Use opencv built-in function to capture image.
182 def connect_camera(self, indexOfCamera):
183 print("trying to connect the index=%d usb port for camera." % indexOfCamera)
184 self.capture = cv2.VideoCapture(indexOfCamera)
185
186
187 # Since we use two markers, here is to decide which marker size we are currently using.
188 # Object points means the real marker size in real world.
189 def decide_obj_points(self, decider='big'):
190
191 if decider == 'big':
192 objPoints = np.float32([[-0.121, -0.121, 0.0],
193 [+0.121, -0.121, 0.0],
194 [+0.121, +0.121, 0.0],
195 [-0.121, +0.121, 0.0]])
196 return objPoints
197 elif decider == 'small':
198 objPoints = np.float32([[-0.030, -0.030, 0.0],
199 [+0.030, -0.030, 0.0],
200 [+0.030, +0.030, 0.0],
201 [-0.030, +0.030, 0.0]])
202 return objPoints
203 else:
204 raise NameError("NO THIS SIZE OF MARKER!!")
205
206
207 # Since we use two markers, here is to decide which marker size we are currently using.
208 # Image points means the marker size in the image from camera.
209 def decide_img_points(self, decider='big'):
210
211 if decider == 'big':
212 imgPoints = np.float32([self.marker213.markerPoints[0][0],
213 self.marker213.markerPoints[1][0],
214 self.marker213.markerPoints[2][0],
215 self.marker213.markerPoints[3][0]])
216 return imgPoints
217
218 elif decider == 'small':
219 imgPoints = np.float32([self.marker456.markerPoints[0][0],
220 self.marker456.markerPoints[1][0],
221 self.marker456.markerPoints[2][0],
222 self.marker456.markerPoints[3][0]])
223 return imgPoints
224
225 else:
226 raise NameError("NO THIS SIZE OF MARKER!!")
227
228
229 # Here is to decide which camera are we using.
230 # Since different camera have slightly different camera matrix and distortion vector.
231 def decide_cam_lens(self, decider='normal'):
232 # There are two kinds of camera lens: wide & normal
233 if decider == 'camA':
234 camMat = np.float32([[463.6725, 0.0, 317.3370],
235 [0.0, 521.3451, 240.6362],
236 [0.0, 0.0, 1.0]])
237 distortionVec = np.float32([-0.3575, 0.1281, -0.000010532, -0.0006737, 0.0])
238 return camMat, distortionVec
239
240 elif decider == 'camB':
241 camMat = np.float32([[466.3010, 0.0, 313.6820],
242 [0.0, 524.3901, 245.4320],
243 [0.0, 0.0, 1.0]])
244 distortionVec = np.float32([-0.3490, 0.1198, -0.0000084383, -0.00091099, 0.0])
245 return camMat, distortionVec
246
247 else:
248 raise NameError("NO THIS KIND OF LENS!!")
249

```

```

250
251
252 #####
253 # Main algorithm functions starts
254 #####
255
256 """
257 Member Function: prepare_image
258 Mainly save the raw image from camera as original image,
259 and convert the raw image into gray scale image and binary image.
260 """
261 def prepare_image(self, image):
262
263 originalImage = image
264
265 self.grayImage = cv2.cvtColor(originalImage, cv2.COLOR_BGR2GRAY)
266
267 # Binarization
268 # Adaptive method can find the contour
269 self.binImage = cv2.adaptiveThreshold(
270 self.grayImage, 255, cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY_INV, 7, 7)
271 # ret, binImage = cv2.threshold(grayImage, 127, 255, cv2.THRESH_BINARY_INV)
272
273
274
275 """
276 Member Function: detect_markers
277 To find all the contours and delete some impossible ones from these requirments: too small, not rectangle.
278 Also checked the point order, we hope the order is anti-clockwise.
279 """
280 def detect_markers(self):
281
282 # Find all the contours
283 _, allContours, _ = cv2.findContours(self.binImage, cv2.RETR_LIST,
284 cv2.CHAIN_APPROX_NONE)
285
286 # Skip the small impossible contours
287 indexOfPassedTestMarker = []
288 possibleContours = []
289
290 for i in range(len(allContours)):
291 real_perimeter = cv2.arcLength(allContours[i], True)
292 if real_perimeter > self.minContourPerimeter:
293 indexOfPassedTestMarker.append(i)
294
295 # print(indexOfPassedTestMarker)
296 for i in range(len(indexOfPassedTestMarker)):
297 possibleContours.append(allContours[indexOfPassedTestMarker[i]])
298
299
300
301 # Find candidate marker
302 # Only the ones can pass the tests
303 self.detectedMarkers = []
304
305 for i in range(len(possibleContours)):
306 epsilon = 0.05*cv2.arcLength(possibleContours[i], True)
307 approxCurve = cv2.approxPolyDP(possibleContours[i], epsilon, True)
308
309 if len(approxCurve) != 4:
310 continue
311
312 if cv2.isContourConvex(approxCurve) == False:
313 continue
314
315 # Ensure that the distance between consecutive points is large enough
316 minDistance = 9999
317 for i in range(4):
318 side = approxCurve[i][0] - approxCurve[(i+1)%4][0]
319 squaredSideLength = side[0]*side[0]+side[1]*side[1]
320 # print(squaredSideLength)
321 minDistance = min(minDistance, squaredSideLength)
322
323 if minDistance < self.minContourLengthAllowed:
324 continue
325
326
327 # Sort the points in anti-clockwise order
328 # Trace a line between the first and second point.
329 # If the third point is at the right side,
330 # then the points are anti-clockwise
331 tempApproxCurve = np.array([[0, 0]], [[0, 0]], [[0, 0]], [[0, 0]])
332 v1 = approxCurve[1][0] - approxCurve[0][0]

```

```

333 v2 = approxCurve[2][0] - approxCurve[0][0]
334 o = (v1[0]*v2[1]) - (v1[1]*v2[0])
335 if o < 0:
336 tempApproxCurve[0][0][0] = approxCurve[0][0][0]
337 tempApproxCurve[0][0][1] = approxCurve[0][0][1]
338 tempApproxCurve[1][0][0] = approxCurve[3][0][0]
339 tempApproxCurve[1][0][1] = approxCurve[3][0][1]
340 tempApproxCurve[2][0][0] = approxCurve[2][0][0]
341 tempApproxCurve[2][0][1] = approxCurve[2][0][1]
342 tempApproxCurve[3][0][0] = approxCurve[1][0][0]
343 tempApproxCurve[3][0][1] = approxCurve[1][0][1]
344 self.detectedMarkers.append(tempApproxCurve)
345 continue
346
347 self.detectedMarkers.append(approxCurve)
348
349
350
351 """
352 Member Function: identify_markerID
353 Mainly we use this function to identify the marker ID. Now we got 213 and 456.
354 getPerspectiveTransform and warpPerspective function are used to transform the image to front view.
355 After filter out those marker whose id = -1, we save the marker contours info into marker class.
356 """
357 def identify_markerID(self):
358
359 self.realMarkers = []
360 finalMarker = []
361
362 for i in range(len(self.detectedMarkers)):
363
364 if self.detectedMarkers[i][1][0][0] == self.detectedMarkers[i][3][0][0] \
365 and self.detectedMarkers[i][1][0][1] == self.detectedMarkers[i][3][0][1]:
366 continue
367
368 markerPoints = np.float32([self.detectedMarkers[i][3][0],
369 self.detectedMarkers[i][0][0],
370 self.detectedMarkers[i][1][0],
371 self.detectedMarkers[i][2][0]])
372
373 markerSize = (105, 105)
374 markerCorners2d = np.float32([[0,0],
375 [markerSize[0]-1,0],
376 [markerSize[0]-1,markerSize[1]-1],
377 [0,markerSize[1]-1]])
378
379 markerTransform = cv2.getPerspectiveTransform(markerPoints,markerCorners2d)
380 tempCanonicalMarkerImage = cv2.warpPerspective(self.grayImage,
381 markerTransform,
382 markerSize)
383
384 tempMarkerID, tempRotations = self.get_markerID(tempCanonicalMarkerImage)
385
386 if tempMarkerID == -1:
387 continue
388
389 # rotate the marker to the oriented direction
390 finalMarker = self.detectedMarkers[i]
391 finalMarker = self.rotate_marker(finalMarker, tempRotations)
392
393 # save the real marker and its info
394 if tempMarkerID == 213:
395 self.isMarker213Detected = True
396 self.marker213.saveMarker(finalMarker, tempMarkerID)
397 self.realMarkers.append(finalMarker)
398 elif tempMarkerID == 456:
399 self.isMarker456Detected = True
400 self.marker456.saveMarker(finalMarker, tempMarkerID)
401 self.realMarkers.append(finalMarker)
402 else:
403 continue
404
405 """
406 Member Function: estimate_position_pose
407 To calculate the position and attitude of marker, first we get relative distance in camera frame
408 and then rotate into target frame and then NED frame based on relative heading angle between target and camera.
409 Then we calculate the marker local location in planner.py.
410 """
411 def estimate_position_pose(self):
412
413 # If detect any markers, then clear all the properties and re-give values
414 # Otherwise just stay the old ones
415 self.numOfMarkerDetected = len(self.realMarkers)

```

```

416 self.isAnyTargetDetected = self.numOfMarkerDetected > 0
417 if not self.isAnyTargetDetected:
418
419 if self.targetLostCount >= 0:
420 self.targetLostCount = self.targetLostCount + 1
421
422 if self.targetLostCount == 3:
423 self.isTargetLost = True
424 self.targetLostCount = -1
425
426 return
427
428
429 # initialize the failure handler parameters
430 self.targetLostCount = 0
431 self.isTargetLost = False
432
433
434 # Switch marker decision part
435 if self.distance_camera_z == 0:
436 if self.isMarker456Detected:
437 self.currentMarkerFlag = 456
438 else:
439 self.currentMarkerFlag = 213
440 elif self.distance_camera_z > 0.7:
441 if self.isMarker213Detected:
442 self.currentMarkerFlag = 213
443 else:
444 self.currentMarkerFlag = 456
445 else:
446 if self.isMarker456Detected:
447 self.currentMarkerFlag = 456
448 else:
449 self.currentMarkerFlag = 213
450
451 # If deicde, initialize obj/img points
452 if self.currentMarkerFlag == 213:
453 self.markerSizeDecider = 'big'
454 elif self.currentMarkerFlag == 456:
455 self.markerSizeDecider = 'small'
456 else:
457 raise NameError("SO WE DECIDE ANOTHER MARKER???")
458
459
460
461
462 for i in range(self.numOfMarkerDetected):
463
464 # prepare objPoints & imgpoints
465 objPoints = self.decide_obj_points(self.markerSizeDecider)
466 imgPoints = self.decide_img_points(self.markerSizeDecider)
467
468 # prepare some more about imgPoints
469 criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.01)
470 imgPoints = cv2.cornerSubPix(self.grayImage, imgPoints,
471 (5,5), (-1,-1), criteria)
472
473 camMat, distortionVec = self.decide_cam_lens(self.cameraLensDecider)
474
475 # solvePnP function
476 # retval, temprvecs, temptvecs = cv2.solvePnP(
477 # objPoints, imgPoints, self.camMat, self.distortionVec)
478 retval, temprvecs, temptvecs = cv2.solvePnP(
479 objPoints, imgPoints, camMat, distortionVec)
480
481
482 # if tvecs and rvecs are valid, then save and calculate Euler angles
483 isCalculationReliable = temptvecs[2][0] > 0
484 if not isCalculationReliable:
485 continue
486
487 # # save rvec and tvec
488 # self.rvecs.append(temprvecs)
489 # self.tvecs.append(temptvecs)
490
491
492 #####
493 distance_x = temptvecs[0][0]
494 distance_y = temptvecs[1][0]
495 distance_z = temptvecs[2][0]
496
497 # save distance in camera frame
498 # camera's right: x axis

```

```

499 # camera's down: y axis
500 # camera's out: z axis
501 self.distance_camera_x = distance_x
502 self.distance_camera_y = distance_y
503 self.distance_camera_z = distance_z
504
505
506 #####
507 # careful! rotMat is not a list like self.rvecs.
508 rotMat = cv2.Rodrigues(tempRvecs)
509 # get rotation values for calculation
510 r11 = rotMat[0][0][0]
511 r21 = rotMat[0][1][0]
512
513
514 # # Calculate Euler angles. Order is z-y-x. Intrinsic rotation
515 yaw_rads = math.atan2(r21, r11)
516 self.camera_Euler_yaw_rads = yaw_rads
517
518 #####
519 # get distance of target frame
520 # rotate 3 times to target frame
521 # math.cos() only takes radians!
522 theta_z = self.camera_Euler_yaw_rads
523
524
525 # start to rotate
526 # the distance in target frame should wait now
527 # because the gimbal needs these distance in camera frame
528 distance_x, distance_y = self.rotate_by_z_axis(-theta_z, distance_x, distance_y)
529
530 # after rotation, the distance is in target frame
531 # put - before them, the distance is in drone frame
532 # This is from the website, o means white, * means black
533 # o***
534 # *ooo*
535 # o*ooo
536 # o*ooo
537 # o*ooo
538 # target right: x axis
539 # target back: y axis
540 # tempDistance_target_x = -distance_x
541 # tempDistance_target_y = -distance_y
542 # tempDistance_target_z = -distance_z
543
544 # In order to recognize, I modify the
545 # distance in target modified frame, origin is below the drone
546 # target placed like this, o means white, * means black
547 # **ooo
548 # *oooo
549 # *oooo
550 # *o***
551 # o*ooo
552 # target front: x axis
553 # target right: y axis
554 tempDistance_target_x = -distance_y
555 tempDistance_target_y = distance_x
556 tempDistance_target_z = distance_z
557
558 # save distance in target frame
559 self.distance_target_x = tempDistance_target_x
560 self.distance_target_y = tempDistance_target_y
561 self.distance_target_z = tempDistance_target_z
562
563
564 #####
565 # distance in world frame
566 target2north = self.marker_attitude_yaw
567
568 tempDistance_north, tempDistance_east = self.rotate_by_z_axis(
569 target2north, tempDistance_target_x, tempDistance_target_y)
570 tempDistance_down = tempDistance_target_z
571
572 self.distance_north = tempDistance_north
573 self.distance_east = tempDistance_east
574 self.distance_down = tempDistance_down
575
576 # distance_NED correction
577 if self.currentMarkerFlag == 213:
578 self.distance_north += self.distance_correction * math.cos(target2north)
579 self.distance_east += self.distance_correction * math.sin(target2north)
580
581

```

```

582 self.isMarker213Detected = False
583 self.isMarker456Detected = False
584
585
586 #####
587 # Main algorithm functions ends
588 #####
589
590
591 # Method to calculate marker ID.
592 # First we divide the marker part into 7*7 cube. Calculate the number of zeros(black), and calculate
593 # the distance. Rotate the marker until we find the min distance. Then we calculate the ID.
594 def get_markerID(self, tempCanonicalMarkerImage):
595
596 grayImage = tempCanonicalMarkerImage
597
598 ret, binImage = cv2.threshold(grayImage,125,255,
599 cv2.THRESH_BINARY+cv2.THRESH_OTSU)
600
601 cellSize = 15
602
603 for y in range(7):
604 inc = 6
605 if y == 0 or y == 6:
606 inc = 1
607
608 for x in range(0,7,inc):
609 cellX = x * cellSize
610 cellY = y * cellSize
611 cell = binImage[cellX:cellX+cellSize, cellY:cellY+cellSize]
612
613 nZ = cv2.countNonZero(cell)
614 if nZ > cellSize*cellSize/2:
615 return -1, -1
616
617 bitMatrix = [[0,0,0,0,0], [0,0,0,0,0], [0,0,0,0,0], [0,0,0,0,0], [0,0,0,0,0]]
618
619 for x in range(5):
620 for y in range(5):
621 cellX = (x+1) * cellSize
622 cellY = (y+1) * cellSize
623 cell = binImage[cellX:cellX+cellSize, cellY:cellY+cellSize]
624
625 nZ = cv2.countNonZero(cell)
626 if nZ > cellSize*cellSize/2:
627 bitMatrix[x][y] = 1
628
629
630
631 rotations = [[[0,0,0,0,0], [0,0,0,0,0], [0,0,0,0,0], [0,0,0,0,0], [0,0,0,0,0]],
632 [[0,0,0,0,0], [0,0,0,0,0], [0,0,0,0,0], [0,0,0,0,0], [0,0,0,0,0]],
633 [[0,0,0,0,0], [0,0,0,0,0], [0,0,0,0,0], [0,0,0,0,0], [0,0,0,0,0]],
634 [[0,0,0,0,0], [0,0,0,0,0], [0,0,0,0,0], [0,0,0,0,0], [0,0,0,0,0]]]
635 distances = [0,0,0,0]
636
637 rotations[0] = bitMatrix
638 distances[0] = self.hamm_dist_marker(rotations[0])
639
640 minDist = [distances[0], 0]
641
642 for i in range(1,4):
643 rotations[i] = self.rotate(rotations[i-1])
644 distances[i] = self.hamm_dist_marker(rotations[i])
645
646 if distances[i] < minDist[0]:
647 minDist[0] = distances[i]
648 minDist[1] = i
649
650 nRotations = minDist[1]
651 if minDist[0] == 0:
652 return self.mat2id(rotations[minDist[1]]), nRotations
653
654 return -1, -1
655
656
657 # Method to rotate matrix. The effect equals to transposition.
658 def rotate(self, inMatrix):
659
660 outMatrix = [[0,0,0,0,0], [0,0,0,0,0], [0,0,0,0,0], [0,0,0,0,0], [0,0,0,0,0]]
661 for i in range(5):
662 for j in range(5):
663 outMatrix[i][j] = inMatrix[5-j-1][i]
664

```

```

665 return outMatrix
666
667
668 # Method to rotate marker to align with target frame. Basically it's the same idea with the rotate matrix method.
669 # For example, nRotations = 2, rotate the marker points anti-clockwisely 90 degree for 2 time.
670 def rotate_marker(self, inMarker, nRotations):
671
672 outMarker = np.array([[[0, 0]], [[0, 0]], [[0, 0]], [[0, 0]]])
673
674 if nRotations == 0:
675 outMarker[0][0][0] = inMarker[0][0][0]
676 outMarker[0][0][1] = inMarker[0][0][1]
677 outMarker[3][0][0] = inMarker[3][0][0]
678 outMarker[3][0][1] = inMarker[3][0][1]
679 outMarker[2][0][0] = inMarker[2][0][0]
680 outMarker[2][0][1] = inMarker[2][0][1]
681 outMarker[1][0][0] = inMarker[1][0][0]
682 outMarker[1][0][1] = inMarker[1][0][1]
683 return outMarker
684
685 if nRotations == 1:
686 outMarker[0][0][0] = inMarker[3][0][0]
687 outMarker[3][0][0] = inMarker[2][0][0]
688 outMarker[2][0][0] = inMarker[1][0][0]
689 outMarker[1][0][0] = inMarker[0][0][0]
690 outMarker[0][0][1] = inMarker[3][0][1]
691 outMarker[3][0][1] = inMarker[2][0][1]
692 outMarker[2][0][1] = inMarker[1][0][1]
693 outMarker[1][0][1] = inMarker[0][0][1]
694
695 return outMarker
696
697 if nRotations == 2:
698 outMarker[0][0][0] = inMarker[2][0][0]
699 outMarker[1][0][0] = inMarker[3][0][0]
700 outMarker[2][0][0] = inMarker[0][0][0]
701 outMarker[3][0][0] = inMarker[1][0][0]
702 outMarker[0][0][1] = inMarker[2][0][1]
703 outMarker[1][0][1] = inMarker[3][0][1]
704 outMarker[2][0][1] = inMarker[0][0][1]
705 outMarker[3][0][1] = inMarker[1][0][1]
706
707 return outMarker
708
709 if nRotations == 3:
710 outMarker[0][0][0] = inMarker[1][0][0]
711 outMarker[1][0][0] = inMarker[2][0][0]
712 outMarker[2][0][0] = inMarker[3][0][0]
713 outMarker[3][0][0] = inMarker[0][0][0]
714 outMarker[0][0][1] = inMarker[1][0][1]
715 outMarker[1][0][1] = inMarker[2][0][1]
716 outMarker[2][0][1] = inMarker[3][0][1]
717 outMarker[3][0][1] = inMarker[0][0][1]
718
719 return outMarker
720
721
722 # Calculate the hamming distance of markers.
723 def hamm_dist_marker(self, inMatrix):
724
725 ids = [[1,0,0,0,0], [1,0,1,1,1], [0,1,0,0,1], [0,1,1,1,0]]
726 dist = 0
727
728 for y in range(5):
729 minSum = 100000
730 for p in range(4):
731 Sum = 0
732 for x in range(5):
733 if inMatrix[y][x] == ids[p][x]:
734 Sum = Sum + 0
735 else:
736 Sum = Sum + 1
737
738 if minSum > Sum:
739 minSum = Sum
740
741 dist = dist + minSum
742
743 return dist
744
745
746 # Calculate the id of the code inside marker.
747 def mat2id(self, inMatrix):

```

```

748 val = 0
749 for i in range(5):
750 val = val << 1
751 if inMatrix[i][1] == 1:
752 val = val | 1
753 val = val << 1
754 if inMatrix[i][3] == 1:
755 val = val | 1
756
757 return val
758
759
760
761 # Draw contours alone marker, and put timestamp on each frame.
762 def draw_contours(self):
763
764 self.grayImageWithMarker = cv2.drawContours(self.grayImage,
765 self.realMarkers,
766 -1, (0,255,0), 10)
767
768 # draw time stamp on image
769 font = cv2.FONT_HERSHEY_SIMPLEX
770 text = self.camShotTimeHMS
771 cv2.putText(self.grayImageWithMarker, text, (400,50), font, 1, (255,255,255), 3)
772
773
774 # Coordinate transform of matrix. Alone z axis.
775 def rotate_by_z_axis(self, theta, x, y):
776 inx = x
777 iny = y
778 rz = theta
779 # math.cos() only takes radians
780 outx = math.cos(rz)*inx - math.sin(rz)*iny
781 outy = math.sin(rz)*inx + math.cos(rz)*iny
782 return outx, outy
783
784
785 # Coordinate transform of matrix. Alone y axis.
786 def rotate_by_y_axis(self, theta, x, z):
787 inx = x
788 inz = z
789 ry = theta
790 # math.cos() only takes radians
791 outx = math.cos(ry)*inx + math.sin(ry)*inz
792 outz = math.cos(ry)*inz - math.sin(ry)*inx
793 return outx, outz
794
795
796 # Coordinate transform of matrix. Alone x axis.
797 def rotate_by_x_axis(self, theta, y, z):
798 iny = y
799 inz = z
800 rx = theta
801 # math.cos() only takes radians
802 outy = math.cos(rx)*iny - math.sin(rx)*inz
803 outz = math.cos(rx)*inz + math.sin(rx)*iny
804 return outy, outz
805
806 *****
807 # opencv_node.py
808
809 #!/usr/bin/env python2
810 # -*- coding: UTF-8 -*-
811
812 # import built-in module and opencv
813 import cv2
814 import time
815 import math
816 # import our own module: OpencvLibrary
817 import OpencvLib
818 # import ROS related library
819 import rospy
820 from cv_bridge import CvBridge
821 from sensor_msgs.msg import Image
822 from geometry_msgs.msg import PointStamped, PoseStamped
823
824
825 # Enable to record raw video without detection
826 ENABLE_VIDEO_RECORD = False
827
828 """
829 Function: sigint_handler
830 Mainly deal with the keyboard interrupt

```

```

831 Clean up everything.
832 """
833 def sigint_handler():
834 print("shutdown time!")
835
836 if ENABLE_VIDEO_RECORD:
837 rawImOut.release()
838
839 """
840 Function: try_connect_camera
841 Mainly initial camera.
842 Usually try 5 times to connect, in case port index changes.
843 """
844 def try_connect_camera():
845
846 try:
847 imageObj.connect_camera(0)
848 finally:
849 print
850
851 for i in range(1,5):
852 if imageObj.capture.isOpened():
853 break
854 try:
855 imageObj.connect_camera(i)
856 finally:
857 print
858
859 """
860 Function: calculate_opencv_data
861 Mainly calculate the distance & pose from a capture image.
862 Output: distance_NED which is saved in imageObj attributes.
863 """
864 def calculate_opencv_data():
865
866 # capture image and record cam shot time
867 # camShotTime is for Kalman Filter
868 # camShotTimeHMS is for displaying on video
869 ret, originalImage = imageObj.capture.read()
870 imageObj.camShotTime = time.time()
871 imageObj.camShotTimeHMS = time.strftime("%H:%M:%S")
872
873 # record raw video
874 if ENABLE_VIDEO_RECORD:
875 rawImOut.write(originalImage)
876
877 # start calculate
878 imageObj.prepare_image(originalImage)
879 imageObj.detect_markers()
880 imageObj.identify_markerID()
881 imageObj.estimate_position_pose()
882
883 """
884 Function: publish_opencv_data
885 Mainly translate the opencv_data into messages
886 So that other pkg can receive via ROS.
887 """
888 def publish_opencv_data():
889
890 # opencv info message
891 # The position part is about distance_NED
892 # The orientation part is about gimbal angle, camshot time and target lost or not.
893 # msgOpencvInfo.pose.orientation.w = -1 means target lost, 1 means target found
894 msgOpencvInfo = PoseStamped()
895 msgOpencvInfo.header.frame_id = str(imageObj.currentMarkerFlag)
896 msgOpencvInfo.header.stamp = rospy.Time.now()
897 msgOpencvInfo.pose.position.x = imageObj.distance_north
898 msgOpencvInfo.pose.position.y = imageObj.distance_east
899 msgOpencvInfo.pose.position.z = imageObj.distance_down
900 msgOpencvInfo.pose.orientation.x = imageObj.gimbal_pitch_current
901 msgOpencvInfo.pose.orientation.y = imageObj.gimbal_pitch_desired
902 msgOpencvInfo.pose.orientation.z = imageObj.camShotTime
903 msgOpencvInfo.pose.orientation.w = -1 if imageObj.isTargetLost else 1
904 opencvInfoPub.publish(msgOpencvInfo)
905
906 rospy.loginfo(msgOpencvInfo)
907
908 """
909
910
911
912
913

```

```

914 # draw and publish image message
915 # Make sure image pub at 4 Hz
916 current_time = time.time()
917 if current_time - imageObj.lastImageMsgTime < 0.23: return
918 imageObj.draw_contours()
919 bridge = CvBridge()
920 msgGrayImageWithMarker = bridge.cv2_to_imgmsg(
921 imageObj.grayImageWithMarker, encoding="mono8")
922 imagePub.publish(msgGrayImageWithMarker)
923
924
925
926 """
927 Function: timer_callback
928 Mainly a scheduler to decide how fast to calculate and publish opencv data
929 """
930 def timer_callback(event):
931
932 calculate_opencv_data()
933 publish_opencv_data()
934
935
936 """
937 Function: marker_attitude_callback
938 Mainly get truck yaw data to calculate distance_NED
939 """
940 def marker_attitude_callback(data):
941 imageObj.marker_attitude_yaw = data.point.z
942
943
944
945
946 if __name__ == '__main__':
947
948 # setup objects for global variables
949 # cameraLensDecider is to decide which camera we are using,
950 # since different camera has different camera matrix
951 imageObj = OpencvLib.OpencvLib()
952 imageObj.cameraLensDecider = 'camA'
953
954
955 # Initial the node opencv_node
956 # set the excutive rate to 25Hz
957 rospy.init_node('opencv_node', anonymous=True)
958
959
960 # setup camera
961 try_connect_camera()
962
963 # setup video saver
964 if ENABLE_VIDEO_RECORD:
965 fourcc = cv2.VideoWriter_fourcc(*'XVID')
966 fps = imageObj.capture.get(cv2.CAP_PROP_FPS)
967 size = (int(imageObj.capture.get(cv2.CAP_PROP_FRAME_WIDTH)),
968 int(imageObj.capture.get(cv2.CAP_PROP_FRAME_HEIGHT)))
969 rawFilename = 'rawImage'+time.strftime("%H-%M-%S")+'.avi'
970 rawImOut = cv2.VideoWriter(rawFilename, fourcc, fps, size)
971
972
973
974 # setup publishers
975 opencvInfoPub = rospy.Publisher("/opencv/info", PoseStamped, queue_size=10)
976 imagePub = rospy.Publisher("/opencv/grayImage_marker", Image, queue_size=1)
977
978 # setup subscribers
979 rospy.Subscriber("/truck/attitude", PointStamped, marker_attitude_callback)
980
981
982 # setup timer
983 # Use timer interupt to make sure publish at 20.8Hz
984 dTimeStep = 0.048
985 rospy.Timer(rospy.Duration(dTimeStep), timer_callback)
986
987
988 # deal with some left-behinds
989 # After spin() no code will be excuted.
990 rospy.on_shutdown(sigint_handler)
991 rospy.spin()
992
993 #####
994 # planner code
995 #####
996 # AltitudeEstimator.py

```

```

997
998 #!/usr/bin/env python
999 # -*- coding: utf-8 -*-
1000
1001 import time
1002 import rospy
1003 import math
1004 import logging
1005 import sys
1006
1007 from io import FileIO, BufferedWriter
1008
1009 # from pymavlink import mavutil
1010 import numpy as np
1011 import matplotlib.pyplot as plt
1012 # from numpy import dot
1013 from geometry_msgs.msg import PointStamped, PoseStamped
1014
1015 LoggingLevel = logging.DEBUG
1016 #LoggingLevel = logging.INFO
1017 #LoggingLevel = logging.WARNING
1018 #LoggingLevel = logging.ERROR
1019
1020 AltitudeEstimatorLog = 'AltitudeEstimatorLog'
1021 AltitudeEstimatorData = 'AltitudeEstimatorData'
1022
1023
1024 # IsVelocityEstimateEnabled determines if we want to use velocity as a state
1025 # False: do not use velocity
1026 # True: use velocity, and the state space will be totally different from the one without velocity
1027 IsVelocityEstimateEnabled = False
1028
1029 # The skipped number of points are determined by 'NumRepeatingBarometerDataToIgnore'
1030 # If the measurement data keep being the same, we will ignore the same data points
1031 # the max ignore points of which is value of NumRepeatingBarometerDataToIgnore
1032 NumRepeatingBarometerDataToIgnore = 0
1033
1034 # R matrix, measurement noise covariance matrix
1035 VarianceUltrasonic = 1.0
1036 VarianceOpencv = 0.08
1037 VarianceBarometer = 0.01
1038 VarianceVelocity = 0.005
1039
1040 # Q matrix, system uncertainty covariance matrix
1041 VarianceAltitudeUncertainty = 0.01
1042 VarianceVelocityUncertainty = 0.05
1043 VarianceBiasUncertainty = 0.005
1044
1045
1046 """
1047 AltitudeEstimator class implements kalman filter, and fuse barometer, ultrasonic and opencv data.
1048 We use opencv to calculate the relative height between marker and camera.
1049 """
1050 class AltitudeEstimator:
1051
1052
1053 def __init__(self):
1054
1055 """
1056 ROS related parameters
1057 """
1058 self.AltitudePub = rospy.Publisher("/AltitudeEstimator/Altitude", PoseStamped, queue_size=10)
1059
1060 # Initialize Barometer data
1061 self.PrevBaromenter = -1;
1062 self.NumRepeatingBarometerData = 0;
1063
1064 # If true, it means we found the target.
1065 self.IsOpencvValid = False
1066
1067 # Save the previous time stamp so we can calculate dt.
1068 # initialize previous time data
1069 self.PreviousMeasurementTime = -1 # -1 indicates the variable has not been used.
1070
1071
1072 """
1073 Control variables
1074 """
1075
1076 # We implement a lock for threads of callbacks.
1077 self.isCallbackLocked = False
1078
1079

```

```

1080 # Initialize predicted data and estimated data.
1081 # x_hat means estimated data, x_pred means predicted data.
1082 if IsVelocityEstimateEnabled:
1083 self.x_hat = np.matrix([[0],
1084 [0],
1085 [0],
1086 [0]])
1087 self.x_pred = np.matrix([[0],
1088 [0],
1089 [0],
1090 [0]])
1091 else:
1092 self.x_hat = np.matrix([[0],
1093 [0]])
1094 self.x_pred = np.matrix([[0],
1095 [0]])
1096
1097 # Initialize 'self.IsFirstMeasurementProcessed'
1098 # We use this to demand that the first data we want to process is barometer data
1099 self.IsFirstMeasurementProcessed = False
1100
1101 self.logger = logging.getLogger(AltitudeEstimatorLog)
1102 formatter = logging.Formatter('%(asctime)s %(levelname)s %(message)s')
1103
1104 self.LogFileHandler = logging.StreamHandler(BufferedWriter(FileIO(AltitudeEstimatorLog + \
1105 '_' + time.strftime("%Y%m%d-%H%M%S") + ".log", "w"))))
1106 self.LogFileHandler.setFormatter(formatter)
1107 self.logger.addHandler(self.LogFileHandler)
1108
1109 # streamHandler = logging.StreamHandler(sys.stdout)
1110 # streamHandler.setFormatter(formatter)
1111 # self.logger.addHandler(streamHandler)
1112
1113 self.logger.setLevel(LoggingLevel)
1114
1115
1116 """
1117 Initialize the model(A matrix) and noise(P matrix)/uncertainty(Q matrix).
1118 """
1119 def StateSpace(self, data):
1120
1121
1122 if IsVelocityEstimateEnabled:
1123 self.Q = np.matrix([[VarianceAltitudeUncertainty, 0, 0],
1124 [0, VarianceVelocityUncertainty, 0],
1125 [0, 0, VarianceBiasUncertainty]])
1126
1127 self.P = np.identity(3)
1128
1129 self.A = np.matrix([[1, self.dt, 0],
1130 [0, 1, 0],
1131 [0, 0, 1]])
1132
1133 else:
1134 self.Q = np.matrix([[VarianceAltitudeUncertainty, 0],
1135 [0, VarianceBiasUncertainty]])
1136
1137 self.P = np.identity(2)
1138
1139 self.A = np.matrix([[1, 0],
1140 [0, 1]])
1141
1142 self.B = 0
1143 self.u = 0
1144
1145
1146 '''
1147 Update predicted data and estimated data
1148 '''
1149 def Update(self, obs):
1150
1151 x = self.x_hat
1152
1153 x_pred = self.A * x + self.B * self.u
1154 self.P = self.A * self.P * self.A.T + self.Q
1155 S = self.C * self.P * self.C.T + self.R
1156 S_inv=np.linalg.inv(S)
1157 K = self.P * self.C.T * S_inv
1158 y = obs - self.C * self.x_pred;
1159 x_hat = self.x_pred + K * y;
1160 self.P = self.P - K * self.C * self.P
1161 x_pred = self.A * x_hat + self.B * self.u
1162

```

```

1163 self.x_hat = x_hat
1164 self.x_pred = x_pred
1165
1166
1167 """
1168 CallbackBarometer function is to receive the Barometer message from ROS network and be processed by Kalman Filter.
1169 """
1170 def CallbackBarometer(self, data):
1171
1172 # We would like to let Barometer data as the first message we got
1173 # Since the barometer has bias, which equals to the first measurement of Barometer.
1174 # This part only for first time measurement.
1175 if not self.IsFirstMeasurementProcessed:
1176
1177 # if self.isCallbackLocked is ture, that means one other callback function is processing.
1178 # Then we can not process this callback function because we only process one callback function at a time.
1179 if self.isCallbackLocked:
1180 self.logger.debug("%d.%d, CallbackBarometer: First measurement has not processed, but callback is locked." \
1181 %(data.header.stamp.secs, data.header.stamp.nsecs))
1182 return
1183
1184 # After receieve the first barometer data, we will never go back in this part 'if' loop
1185 self.IsFirstMeasurementProcessed = True
1186
1187 # To guarantee we only process one callback function everytime, just in case we receive 2 data at the same time.
1188 self.isCallbackLocked = True
1189
1190 # Critical section
1191 self.PreviousMeasurementTime = data.header.stamp.secs + data.header.stamp.nsecs/1000000000.0
1192 barometer = data.point.z
1193
1194 # After received all the data, we can open the callback lock, then data can be processed in other callback function.
1195 self.isCallbackLocked = False
1196
1197 # At the beginning the barometer data must be zero because the drone has not taken off yet
1198 # But the measurement is not, so the first data equals to bias.
1199 barometer_bias = barometer
1200 altitude = 0
1201 velocity = 0
1202
1203
1204 if IsVelocityEstimateEnabled:
1205 self.x_hat = np.matrix([[altitude], [velocity], [barometer_bias]])
1206 self.logger.debug("%d.%d, barometer: %f, alt: %f, vel: %f, bias: %f" %(data.header.stamp.secs, \
1207 data.header.stamp.nsecs, barometer, self.x_hat[0], self.x_hat[1], self.x_hat[2]))
1208 else:
1209 self.x_hat = np.matrix([[altitude], [barometer_bias]])
1210 self.logger.debug("%d.%d, barometer: %f, alt: %f, bias: %f" %(data.header.stamp.secs, \
1211 data.header.stamp.nsecs, barometer, self.x_hat[0], self.x_hat[1]))
1212
1213 self.x_pred = self.x_hat
1214
1215 self.publish_altitude(data)
1216
1217 return
1218
1219 # if self.isCallbackLocked is ture, that means one other callback function is processing.
1220 # Then we can not process this callback function because we only process one callback function at a time.
1221 if self.isCallbackLocked:
1222 return
1223
1224 # To guarantee we only process one callback function everytime, just in case we receive 2 data at the same time.
1225 self.isCallbackLocked = True
1226 # Critical section
1227
1228 measurement_time = data.header.stamp.secs + data.header.stamp.nsecs/1000000000.0
1229
1230 self.dt = measurement_time - self.PreviousMeasurementTime
1231
1232 # if self.dt < 0, it is out of time order. We want to ignore those measurements at that point.
1233 if self.dt < 0:
1234 self.isCallbackLocked = False
1235 self.logger.info("CallbackTruckLocation: dt is less than 0.")
1236 return
1237
1238 self.PreviousMeasurementTime = measurement_time
1239
1240 barometer = data.point.z
1241
1242 # deal with repeating data
1243 # if we got some repeating data, ignore some based on NumRepeatingBarometerDataToIgnore
1244 # the reason we ignore repeating data is that most of time, the repeating data is invalid for the unsensitive of senses.
1245 # Measurements should change a little bit everytime.

```

```

1246
1247 if barometer == self.PrevBarometer:
1248 self.logger.debug("..... Repeating barometer data, %d" %self.NumRepeatingBarometerData)
1249 self.NumRepeatingBarometerData = self.NumRepeatingBarometerData + 1
1250 if self.NumRepeatingBarometerData <= NumRepeatingBarometerDataToIgnore:
1251 self.logger.debug("..... Ignore the barometer data")
1252 self.isCallbackLocked = False
1253 return
1254 else:
1255 self.NumRepeatingBarometerData = 0
1256 self.logger.debug("..... Process the barometer data")
1257
1258 else:
1259 self.NumRepeatingBarometerData = 0
1260 self.PrevBarometer = barometer
1261 self.logger.debug("..... Process the barometer data")
1262
1263 # After received all the data, we can open the callback lock, then data can be processed in other callback function.
1264 self.isCallbackLocked = False
1265
1266
1267 # Update predicted data and estimated data when the input data is barometer
1268 obs = np.matrix([[barometer]])
1269
1270 if IsVelocityEstimateEnabled:
1271 self.C = np.matrix([[1, 0, 1]])
1272 else:
1273 self.C = np.matrix([[1, 1]])
1274
1275 self.R = VarianceBarometer
1276
1277 self.StateSpace(data)
1278 self.Update(obs)
1279
1280 self.publish_altitude(data)
1281
1282
1283 if IsVelocityEstimateEnabled:
1284 self.logger.debug("%d.%d, barometer: %f, alt: %f, vel: %f, bias: %f" %(data.header.stamp.secs,\
1285 data.header.stamp.nsecs, barometer, self.x_hat[0], self.x_hat[1], self.x_hat[2]))
1286 else:
1287 self.logger.debug("%d.%d, barometer: %f, alt: %f, bias: %f" %(data.header.stamp.secs,\
1288 data.header.stamp.nsecs, barometer, self.x_hat[0], self.x_hat[1]))
1289
1290
1291 """
1292 CallbackUltrasonic function is to receive the Ultrasonic sensor message from ROS network and be processed by Kalman Filter.
1293 """
1294 def CallbackUltrasonic(self, data):
1295
1296 # If the first data we receive is ultrasonic data, then ignore it.
1297 if not self.IsFirstMeasurementProcessed:
1298 self.logger.debug("%d.%d, The first barometer measurement has not been processed." \
1299 %(data.header.stamp.secs, data.header.stamp.nsecs))
1300 return
1301
1302 # If anyother callback function is receiving data, then ignore this ultrasonic data.
1303 if self.isCallbackLocked:
1304 self.logger.debug("%d.%d, Callback is locked at ultrasonic." %(data.header.stamp.secs, data.header.stamp.nsecs))
1305 return
1306
1307 # To guarantee we only process one callback function everytime, just in case we receive 2 data at the same time.
1308 self.isCallbackLocked = True
1309
1310 # Critical section
1311 measurement_time = data.header.stamp.secs + data.header.stamp.nsecs/1000000000.0
1312
1313 self.dt = measurement_time - self.PreviousMeasurementTime
1314
1315 # if self.dt < 0, it is out of time order. We want to ignore those measurements at that point.
1316 if self.dt < 0:
1317 self.isCallbackLocked = False
1318 self.logger.info("CallbackUltrasonic: dt is less than 0.")
1319 return
1320 else:
1321 self.PreviousMeasurementTime = measurement_time
1322
1323 ultrasonic = data.point.x
1324
1325 # After received all the data, we can open the callback lock, then data can be processed in other callback function.
1326 self.isCallbackLocked = False
1327
1328

```

```

1329 # In the situation of ultrasonic > 2.5 or self.x_hat[0] > 2.5,
1330 # We want to ignore these ultrasonic data because it has a lot of noises.
1331 if ultrasonic > 2.5 or self.x_hat[0] > 2.5:
1332 if IsVelocityEstimateEnabled:
1333 self.logger.debug("%d.%d, Ultrasonic measurement is ignored. ultrasonic: %f, alt: %f, vel: %f, bias: %f" \
1334 %(data.header.stamp.secs, data.header.stamp.nsecs, ultrasonic, self.x_hat[0], self.x_hat[1], self.x_hat[2]))
1335 else:
1336 self.logger.debug("%d.%d, Ultrasonic measurement is ignored. ultrasonic: %f, alt: %f, bias: %f" \
1337 %(data.header.stamp.secs, data.header.stamp.nsecs, ultrasonic, self.x_hat[0], self.x_hat[1]))
1338
1339 return
1340
1341 # Update predicted data and estimated data when the input data is ultrasonic.
1342 obs = np.matrix([[ultrasonic]])
1343
1344
1345 if IsVelocityEstimateEnabled:
1346 self.C = np.matrix([[1, 0, 0]])
1347 else:
1348 self.C = np.matrix([[1, 0]])
1349
1350 self.R = VarianceUltrasonic
1351
1352 self.StateSpace(data)
1353 self.Update(obs)
1354 self.publish_altitude(data)
1355
1356 if IsVelocityEstimateEnabled:
1357 self.logger.debug("%d.%d, ultrasonic: %f, alt: %f, vel: %f, bias: %f" %(data.header.stamp.secs, \
1358 data.header.stamp.nsecs, ultrasonic, self.x_hat[0], self.x_hat[1], self.x_hat[2]))
1359 else:
1360 self.logger.debug("%d.%d, ultrasonic: %f, alt: %f, bias: %f" %(data.header.stamp.secs, \
1361 data.header.stamp.nsecs, ultrasonic, self.x_hat[0], self.x_hat[1]))
1362
1363
1364 """
1365 CallbackOpencvInfo function is to receive the opencv estimated height message from ROS network and be processed by Kalman Filter.
1366 """
1367 def CallbackOpencvInfo(self, data):
1368
1369 # If the first data we receive is opencv data, then ignore it.
1370 if not self.IsFirstMeasurementProcessed:
1371 self.logger.debug("%d.%d, The first barometer measurement has not been processed." \
1372 %(data.header.stamp.secs, data.header.stamp.nsecs))
1373 return
1374
1375 # If anyother callback function is receiving data, then ignore this opencv data.
1376 if self.isCallbackLocked:
1377 self.logger.debug("%d.%d, Callback is locked at opencv." %(data.header.stamp.secs, data.header.stamp.nsecs))
1378 return
1379
1380 # To guarantee we only process one callback function everytime, just in case we receive 2 data at the same time.
1381 self.isCallbackLocked = True
1382
1383 # Critical section
1384 # w = -1 means opencv lost, then we ignore this data.
1385 if data.pose.orientation.w == -1:
1386 self.logger.debug("%d.%d, Opencv measurement is invalid." %(data.header.stamp.secs, data.header.stamp.nsecs))
1387 self.isCallbackLocked = False
1388 return
1389
1390 measurement_time = data.header.stamp.secs + data.header.stamp.nsecs/1000000000.0
1391 self.dt = measurement_time - self.PreviousMeasurementTime
1392
1393 # if self.dt < 0, it is out of time order. We want to ignore those measurements at that point.
1394 if self.dt < 0:
1395 self.isCallbackLocked = False
1396 self.logger.info("CallbackOpencv: dt is less than 0.")
1397 return
1398 else:
1399 self.PreviousMeasurementTime = measurement_time
1400
1401
1402 distance_down = data.pose.position.z
1403
1404 # After received all the data, we can open the callback lock, then data can be processed in other callback function.
1405 self.isCallbackLocked = False
1406
1407 # Update predicted data and estimated data when the input data is opencv data.
1408 obs = np.matrix([[distance_down]])
1409
1410
1411 if IsVelocityEstimateEnabled:

```

```

1412 self.C = np.matrix([[1, 0, 0]])
1413 else:
1414 self.C = np.matrix([[1, 0]])
1415
1416 self.R = VarianceOpencv
1417
1418 self.StateSpace(data)
1419 self.Update(obs)
1420 self.publish_altitude(data)
1421
1422 if IsVelocityEstimateEnabled:
1423 self.logger.debug("%d.%d, opencv: %f, alt: %f, vel: %f, bias: %f" %(data.header.stamp.secs,\
1424 data.header.stamp.nsecs, distance_down, self.x_hat[0], self.x_hat[1], self.x_hat[2]))
1425 else:
1426 self.logger.debug("%d.%d, opencv: %f, alt: %f, bias: %f" %(data.header.stamp.secs,\
1427 data.header.stamp.nsecs, distance_down, self.x_hat[0], self.x_hat[1]))
1428
1429
1430 """
1431 CallbackSoloVelocity is to receive the drone's velocity message from ROS network.
1432 Velocity is used as a part of system kinetics model, to predict the next state in Kalman Filter.
1433 """
1434 def CallbackSoloVelocity(self, data):
1435
1436 self.logger.debug("CallbackSoloVelocity: measurement time: %d.%d." %(data.header.stamp.secs, data.header.stamp.nsecs))
1437
1438 # If the first data we receive is velocity data, then ignore it.
1439 if not self.IsFirstMeasurementProcessed:
1440 return
1441
1442 # If we do not use velocity as a state, then we can ignore the whole velocity part.
1443 if not IsVelocityEstimateEnabled:
1444 return
1445
1446 # If anyother callback function is receiving data, then ignore this velocity data.
1447 if self.isCallbackLocked:
1448 self.logger.debug("CallbackSoloVelocity: Callback is locked.")
1449 return
1450
1451 # To guarantee we only process one callback function everytime, just in case we receive 2 data at the same time.
1452 self.isCallbackLocked = True
1453
1454 # Critical section
1455 measurement_time = data.header.stamp.secs + data.header.stamp.nsecs/1000000000.0
1456 self.dt = measurement_time - self.PreviousMeasurementTime
1457
1458 # if self.dt < 0, it is out of time order. We want to ignore those measurements at that point.
1459 if self.dt < 0:
1460 self.isCallbackLocked = False
1461 self.logger.info("CallbackSoloVelocity: dt is less than 0.")
1462 return
1463 else:
1464 self.PreviousMeasurementTime = measurement_time
1465
1466 velocity = data.point.z
1467
1468 # After received all the data, we can open the callback lock, then data can be processed in other callback function.
1469 self.isCallbackLocked = False
1470
1471 # Update predicted data and estimated data when tne input data is velocity.
1472 obs = np.matrix([[velocity]])
1473
1474 self.C = np.matrix([[0, 1, 0]])
1475
1476 self.R = VarianceVelocity
1477
1478
1479 self.StateSpace(data)
1480 self.Update(obs)
1481 self.publish_altitude(data)
1482
1483 self.logger.debug("%d.%d, velocity: %f, alt: %f, vel: %f, bias: %f" %(data.header.stamp.secs,\
1484 data.header.stamp.nsecs, velocity, self.x_hat[0], self.x_hat[1], self.x_hat[2]))
1485
1486
1487 """
1488 Publish the prediction(x_pred) and state estimate(x_hat) data in ROS network.
1489 """
1490 def publish_altitude(self, data):
1491
1492 msgAltitude = PoseStamped()
1493 msgAltitude.header.stamp = data.header.stamp
1494

```

```

1495
1496 if IsVelocityEstimateEnabled:
1497 # x_hat[0]: height, x_hat[1]: velocity, self.x_hat[2]: bias
1498 msgAltitude.pose.position.x = self.x_hat[0]
1499 msgAltitude.pose.position.y = self.x_hat[1]
1500 msgAltitude.pose.position.z = self.x_hat[2]
1501 msgAltitude.pose.orientation.x = self.x_pred[0]
1502 msgAltitude.pose.orientation.y = self.x_pred[1]
1503 msgAltitude.pose.orientation.z = self.x_pred[2]
1504 msgAltitude.pose.orientation.w = 0
1505 else:
1506 msgAltitude.pose.position.x = self.x_hat[0]
1507 msgAltitude.pose.position.y = 0
1508 msgAltitude.pose.position.z = self.x_hat[1]
1509 msgAltitude.pose.orientation.x = self.x_pred[0]
1510 msgAltitude.pose.orientation.y = 0
1511 msgAltitude.pose.orientation.z = self.x_pred[1]
1512 msgAltitude.pose.orientation.w = 0
1513
1514
1515 self.AltitudePub.publish(msgAltitude)
1516 #rospy.loginfo(msgAltitude)
1517
1518
1519 def sigint_handler(self):
1520
1521 # self.LogFileHandler.flush()
1522 # self.LogFileHandler.close()
1523 # self.Plot()
1524 print("Altitude Estimator is shutdown!")
1525
1526
1527 def main():
1528
1529 estimator = AltitudeEstimator()
1530
1531 rospy.init_node('AltitudeEstimator', anonymous=True)
1532
1533 # rospy.Subscriber("/solo/attitude", PointStamped, kalman.CallbackAttitude)
1534 rospy.Subscriber("/solo/location_GPS", PointStamped, estimator.CallbackBarometer)
1535 rospy.Subscriber("/ultrasonic/long_distance", PointStamped, estimator.CallbackUltrasonic)
1536 rospy.Subscriber("/opencv/info", PoseStamped, estimator.CallbackOpencvInfo)
1537 rospy.Subscriber("/solo/velocity", PointStamped, estimator.CallbackSoloVelocity)
1538
1539 rospy.on_shutdown(estimator.sigint_handler)
1540 rospy.spin()
1541
1542
1543 if __name__ == '__main__':
1544
1545 main()
1546
1547 *****
1548 # guiTest.py
1549 #!/usr/bin/env python2
1550 # -*- coding: UTF-8 -*-
1551
1552 import tkinter as tk
1553 import tkinter.messagebox as messagebox
1554 import time
1555 import rospy
1556 from std_msgs.msg import String
1557 from geometry_msgs.msg import PointStamped, PoseStamped
1558
1559
1560
1561 class ROSGUI:
1562
1563 """
1564 Initialize variables for data shown in the interface.
1565 """
1566 def __init__(self):
1567
1568 # Control flags
1569 self.stopUpdate = False
1570
1571 self.window = tk.Tk()
1572 self.window.title('ROS Message')
1573 self.window.geometry('600x800')
1574
1575 self.varState = tk.StringVar()
1576
1577 self.varDronePosition_lat = tk.DoubleVar()

```

```

1578 self.varDronePosition_lon = tk.DoubleVar()
1579 self.varDronePosition_alt = tk.DoubleVar()
1580 self.varDronePosition_North = tk.DoubleVar()
1581 self.varDronePosition_East = tk.DoubleVar()
1582 self.varDronePosition_Down = tk.DoubleVar()
1583 self.varDronePosition_KF_North = tk.DoubleVar()
1584 self.varDronePosition_KF_East = tk.DoubleVar()
1585 self.varKF_Altitude = tk.DoubleVar()
1586 self.varKF_Altitude_Prediction = tk.DoubleVar()
1587
1588 self.varTruckPosition_North = tk.DoubleVar()
1589 self.varTruckPosition_East = tk.DoubleVar()
1590 self.varTruckPosition_Down = tk.DoubleVar()
1591
1592 self.varDroneAttitude_Roll = tk.DoubleVar()
1593 self.varDroneAttitude_Pitch = tk.DoubleVar()
1594 self.varDroneAttitude_Yaw = tk.DoubleVar()
1595
1596 self.varTruckAttitude_Roll = tk.DoubleVar()
1597 self.varTruckAttitude_Pitch = tk.DoubleVar()
1598 self.varTruckAttitude_Yaw = tk.DoubleVar()
1599
1600 self.varDroneVelocity_X = tk.DoubleVar()
1601 self.varDroneVelocity_Y = tk.DoubleVar()
1602 self.varDroneVelocity_Z = tk.DoubleVar()
1603
1604 self.varTruckVelocity_X = tk.DoubleVar()
1605 self.varTruckVelocity_Y = tk.DoubleVar()
1606 self.varTruckVelocity_Z = tk.DoubleVar()
1607
1608 self.varFound = tk.BooleanVar(False)
1609
1610 self.create_widgets()
1611
1612
1613
1614 """
1615 Create widgets to display labels and data.
1616 """
1617 def create_widgets(self):
1618
1619 self.buttonStart = tk.Button(self.window, text='Start', command=self.start_data)
1620 self.buttonStart.place(x=200, y=15, anchor="center")
1621
1622 levelOne = 30
1623 self.labelState = tk.Label(self.window, text='Current state', width=30, height=1, justify='left')
1624 self.labelState.place(x=0, y=levelOne+20, anchor="nw")
1625 self.textState = tk.Label(self.window, textvariable=self.varState, width=20, height=1, justify='left')
1626 self.textState.place(x=300, y=levelOne+20, anchor="nw")
1627
1628 self.labelFound = tk.Label(self.window, text='Found', width=30, height=1, justify='left')
1629 self.labelFound.place(x=0, y=levelOne+40, anchor="nw")
1630 self.textFound = tk.Label(self.window, textvariable=self.varFound, width=20, height=1, justify='left')
1631 self.textFound.place(x=300, y=levelOne+40, anchor="nw")
1632
1633 levelTwo = 100
1634 self.labelDronePosition_lat = tk.Label(self.window, text='DronePosition_lat', width=30, height=1, justify='left')
1635 self.labelDronePosition_lat.place(x=0, y=levelTwo, anchor="nw")
1636 self.textDronePosition_lat = tk.Label(self.window, textvariable=self.varDronePosition_lat, width=20, height=1, justify='left')
1637 self.textDronePosition_lat.place(x=300, y=levelTwo, anchor="nw")
1638
1639 self.labelDronePosition_lon = tk.Label(self.window, text='DronePosition_lon', width=30, height=1, justify='left')
1640 self.labelDronePosition_lon.place(x=0, y=levelTwo+20, anchor="nw")
1641 self.textDronePosition_lon = tk.Label(self.window, textvariable=self.varDronePosition_lon, width=20, height=1, justify='left')
1642 self.textDronePosition_lon.place(x=300, y=levelTwo+20, anchor="nw")
1643
1644 self.labelDronePosition_alt = tk.Label(self.window, text='DronePosition_alt', width=30, height=1, justify='left')
1645 self.labelDronePosition_alt.place(x=0, y=levelTwo+40, anchor="nw")
1646 self.textDronePosition_alt = tk.Label(self.window, textvariable=self.varDronePosition_alt, width=20, height=1, justify='left')
1647 self.textDronePosition_alt.place(x=300, y=levelTwo+40, anchor="nw")
1648
1649 self.labelDronePosition_North = tk.Label(self.window, text='DronePosition_North', width=30, height=1, justify='left')
1650 self.labelDronePosition_North.place(x=0, y=levelTwo+70, anchor="nw")
1651 self.textDronePosition_North = tk.Label(self.window, textvariable=self.varDronePosition_North,\
1652 width=20, height=1, justify='left')
1653 self.textDronePosition_North.place(x=300, y=levelTwo+70, anchor="nw")
1654
1655 self.labelDronePosition_East = tk.Label(self.window, text='DronePosition_East', width=30, height=1, justify='left')
1656 self.labelDronePosition_East.place(x=0, y=levelTwo+90, anchor="nw")
1657 self.textDronePosition_East = tk.Label(self.window, textvariable=self.varDronePosition_East,\
1658 width=20, height=1, justify='left')
1659 self.textDronePosition_East.place(x=300, y=levelTwo+90, anchor="nw")
1660

```

```

1661 self.labelDronePosition_Down = tk.Label(self.window, text='DronePosition_Down', width=30, height=1, justify='left')
1662 self.labelDronePosition_Down.place(x=0, y=levelTwo+110, anchor="nw")
1663 self.textDronePosition_Down = tk.Label(self.window, textvariable=self.varDronePosition_Down,\
1664 width=20, height=1, justify='left')
1665 self.textDronePosition_Down.place(x=300, y=levelTwo+110, anchor="nw")
1666
1667 self.labelDronePosition_KF_North = tk.Label(self.window, text='DronePosition_KF_North', width=30, height=1, justify='left')
1668 self.labelDronePosition_KF_North.place(x=0, y=levelTwo+140, anchor="nw")
1669 self.textDronePosition_KF_North = tk.Label(self.window, textvariable=self.varDronePosition_KF_North,\
1670 width=20, height=1, justify='left')
1671 self.textDronePosition_KF_North.place(x=300, y=levelTwo+140, anchor="nw")
1672
1673 self.labelDronePosition_KF_East = tk.Label(self.window, text='DronePosition_KF_East', width=30, height=1, justify='left')
1674 self.labelDronePosition_KF_East.place(x=0, y=levelTwo+160, anchor="nw")
1675 self.textDronePosition_KF_East = tk.Label(self.window, textvariable=self.varDronePosition_KF_East,\
1676 width=20, height=1, justify='left')
1677 self.textDronePosition_KF_East.place(x=300, y=levelTwo+160, anchor="nw")
1678
1679 self.labelKF_Altitude = tk.Label(self.window, text='KF_Altitude', width=30, height=1, justify='left')
1680 self.labelKF_Altitude.place(x=0, y=levelTwo+190, anchor="nw")
1681 self.textKF_Altitude = tk.Label(self.window, textvariable=self.varKF_Altitude, width=20, height=1, justify='left')
1682 self.textKF_Altitude.place(x=300, y=levelTwo+190, anchor="nw")
1683
1684 self.labelKF_Altitude_Prediction = tk.Label(self.window, text='KF_Altitude_Prediction', width=30, height=1, justify='left')
1685 self.labelKF_Altitude_Prediction.place(x=0, y=levelTwo+210, anchor="nw")
1686 self.textKF_Altitude_Prediction = tk.Label(self.window, textvariable=self.varKF_Altitude_Prediction,\
1687 width=20, height=1, justify='left')
1688 self.textKF_Altitude_Prediction.place(x=300, y=levelTwo+210, anchor="nw")
1689
1690 levelThree = 350
1691 self.labelTruckPosition_North = tk.Label(self.window, text='TruckPosition_North', width=30, height=1, justify='left')
1692 self.labelTruckPosition_North.place(x=0, y=levelThree, anchor="nw")
1693 self.textTruckPosition_North = tk.Label(self.window, textvariable=self.varTruckPosition_North,\
1694 width=20, height=1, justify='left')
1695 self.textTruckPosition_North.place(x=300, y=levelThree, anchor="nw")
1696
1697 self.labelTruckPosition_East = tk.Label(self.window, text='TruckPosition_East', width=30, height=1, justify='left')
1698 self.labelTruckPosition_East.place(x=0, y=levelThree+20, anchor="nw")
1699 self.textTruckPosition_East = tk.Label(self.window, textvariable=self.varTruckPosition_East,\
1700 width=20, height=1, justify='left')
1701 self.textTruckPosition_East.place(x=300, y=levelThree+20, anchor="nw")
1702
1703 self.labelTruckPosition_Down = tk.Label(self.window, text='TruckPosition_Down', width=30, height=1, justify='left')
1704 self.labelTruckPosition_Down.place(x=0, y=levelThree+40, anchor="nw")
1705 self.textTruckPosition_Down = tk.Label(self.window, textvariable=self.varTruckPosition_Down,\
1706 width=20, height=1, justify='left')
1707 self.textTruckPosition_Down.place(x=300, y=levelThree+40, anchor="nw")
1708
1709 levelFour = 430
1710 self.labelDroneAttitude_Roll = tk.Label(self.window, text='DroneAttitude_Roll', width=30, height=1, justify='left')
1711 self.labelDroneAttitude_Roll.place(x=0, y=levelFour, anchor="nw")
1712 self.textDroneAttitude_Roll = tk.Label(self.window, textvariable=self.varDroneAttitude_Roll,\
1713 width=20, height=1, justify='left')
1714 self.textDroneAttitude_Roll.place(x=300, y=levelFour, anchor="nw")
1715
1716 self.labelDroneAttitude_Pitch = tk.Label(self.window, text='DroneAttitude_Pitch', width=30, height=1, justify='left')
1717 self.labelDroneAttitude_Pitch.place(x=0, y=levelFour+20, anchor="nw")
1718 self.textDroneAttitude_Pitch = tk.Label(self.window, textvariable=self.varDroneAttitude_Pitch,\
1719 width=20, height=1, justify='left')
1720 self.textDroneAttitude_Pitch.place(x=300, y=levelFour+20, anchor="nw")
1721
1722 self.labelDroneAttitude_Yaw = tk.Label(self.window, text='DroneAttitude_Yaw', width=30, height=1, justify='left')
1723 self.labelDroneAttitude_Yaw.place(x=0, y=levelFour+40, anchor="nw")
1724 self.textDroneAttitude_Yaw = tk.Label(self.window, textvariable=self.varDroneAttitude_Yaw,\
1725 width=20, height=1, justify='left')
1726 self.textDroneAttitude_Yaw.place(x=300, y=levelFour+40, anchor="nw")
1727
1728 levelFive = 510
1729 self.labelTruckAttitude_Roll = tk.Label(self.window, text='TruckAttitude_Roll', width=30, height=1, justify='left')
1730 self.labelTruckAttitude_Roll.place(x=0, y=levelFive, anchor="nw")
1731 self.textTruckAttitude_Roll = tk.Label(self.window, textvariable=self.varTruckAttitude_Roll,\
1732 width=20, height=1, justify='left')
1733 self.textTruckAttitude_Roll.place(x=300, y=levelFive, anchor="nw")
1734
1735 self.labelTruckAttitude_Pitch = tk.Label(self.window, text='TruckAttitude_Pitch', width=30, height=1, justify='left')
1736 self.labelTruckAttitude_Pitch.place(x=0, y=levelFive+20, anchor="nw")
1737 self.textTruckAttitude_Pitch = tk.Label(self.window, textvariable=self.varTruckAttitude_Pitch,\
1738 width=20, height=1, justify='left')
1739 self.textTruckAttitude_Pitch.place(x=300, y=levelFive+20, anchor="nw")
1740
1741 self.labelTruckAttitude_Yaw = tk.Label(self.window, text='TruckAttitude_Yaw', width=30, height=1, justify='left')
1742 self.labelTruckAttitude_Yaw.place(x=0, y=levelFive+40, anchor="nw")
1743 self.textTruckAttitude_Yaw = tk.Label(self.window, textvariable=self.varTruckAttitude_Yaw,\

```

```

1744 width=20, height=1, justify='left')
1745 self.textTruckAttitude_Yaw.place(x=300, y=levelFive+40, anchor="nw")
1746
1747 levelSix = 590
1748 self.labelDroneVelocity_X = tk.Label(self.window, text='DroneVelocity_X', width=30, height=1, justify='left')
1749 self.labelDroneVelocity_X.place(x=0, y=levelSix, anchor="nw")
1750 self.textDroneVelocity_X = tk.Label(self.window, textvariable=self.varDroneVelocity_X,\
1751 width=20, height=1, justify='left')
1752 self.textDroneVelocity_X.place(x=300, y=levelSix, anchor="nw")
1753
1754 self.labelDroneVelocity_Y = tk.Label(self.window, text='DroneVelocity_Y', width=30, height=1, justify='left')
1755 self.labelDroneVelocity_Y.place(x=0, y=levelSix+20, anchor="nw")
1756 self.textDroneVelocity_Y = tk.Label(self.window, textvariable=self.varDroneVelocity_Y,\
1757 width=20, height=1, justify='left')
1758 self.textDroneVelocity_Y.place(x=300, y=levelSix+20, anchor="nw")
1759
1760 self.labelDroneVelocity_Z = tk.Label(self.window, text='DroneVelocity_Z', width=30, height=1, justify='left')
1761 self.labelDroneVelocity_Z.place(x=0, y=levelSix+40, anchor="nw")
1762 self.textDroneVelocity_Z = tk.Label(self.window, textvariable=self.varDroneVelocity_Z,\
1763 width=20, height=1, justify='left')
1764 self.textDroneVelocity_Z.place(x=300, y=levelSix+40, anchor="nw")
1765
1766 levelSeven = 670
1767 self.labelTruckVelocity_X = tk.Label(self.window, text='TruckVelocity_X', width=30, height=1, justify='left')
1768 self.labelTruckVelocity_X.place(x=0, y=levelSeven, anchor="nw")
1769 self.textTruckVelocity_X = tk.Label(self.window, textvariable=self.varTruckVelocity_X,\
1770 width=20, height=1, justify='left')
1771 self.textTruckVelocity_X.place(x=300, y=levelSeven, anchor="nw")
1772
1773 self.labelTruckVelocity_Y = tk.Label(self.window, text='TruckVelocity_Y', width=30, height=1, justify='left')
1774 self.labelTruckVelocity_Y.place(x=0, y=levelSeven+20, anchor="nw")
1775 self.textTruckVelocity_Y = tk.Label(self.window, textvariable=self.varTruckVelocity_Y,\
1776 width=20, height=1, justify='left')
1777 self.textTruckVelocity_Y.place(x=300, y=levelSeven+20, anchor="nw")
1778
1779 self.labelTruckVelocity_Z = tk.Label(self.window, text='TruckVelocity_Z', width=30, height=1, justify='left')
1780 self.labelTruckVelocity_Z.place(x=0, y=levelSeven+40, anchor="nw")
1781 self.textTruckVelocity_Z = tk.Label(self.window, textvariable=self.varTruckVelocity_Z,\
1782 width=20, height=1, justify='left')
1783 self.textTruckVelocity_Z.place(x=300, y=levelSeven+40, anchor="nw")
1784
1785 self.buttonStop = tk.Button(self.window, text='Stop!', command=self.clear_data)
1786 self.buttonStop.place(x=400, y=15, anchor="center")
1787
1788 """
1789 Receive state.
1790 """
1791 def CallbackState(self, data):
1792
1793 self.varState.set(data.data)
1794
1795 """
1796 Receive drone position from Kalman filter.
1797 """
1798 def CallbackDronePosition_KF(self, data):
1799
1800 self.varDronePosition_KF_North.set(data.pose.orientation.y)
1801 self.varDronePosition_KF_East.set(data.pose.orientation.z)
1802
1803 """
1804 Receive drone altitude.
1805 """
1806 def CallbackKF_Altitude(self, data):
1807
1808 self.varKF_Altitude.set(data.pose.position.x)
1809 self.varKF_Altitude_Prediction.set(data.pose.orientation.x)
1810
1811 """
1812 Receive drone position from gps.
1813 """
1814 def CallbackDronePosition_GPS(self, data):
1815
1816 self.varDronePosition_lat.set(data.point.x)
1817 self.varDronePosition_lon.set(data.point.y)
1818 self.varDronePosition_alt.set(data.point.z)
1819
1820 """
1821 Receive drone velocity.
1822 """
1823 def CallbackDroneVelocity(self, data):
1824
1825 self.varDroneVelocity_X.set(data.point.x)
1826 self.varDroneVelocity_Y.set(data.point.y)

```

```

1827 self.varDroneVelocity_Z.set(data.point.z)
1828
1829 """
1830 Receive truck velocity.
1831 """
1832 def CallbackTruckVelocity(self, data):
1833
1834 self.varTruckVelocity_X.set(data.point.x)
1835 self.varTruckVelocity_Y.set(data.point.y)
1836 self.varTruckVelocity_Z.set(data.point.z)
1837
1838 """
1839 Receive truck attitude.
1840 """
1841 def CallbackTruckAttitude(self, data):
1842
1843 self.varTruckAttitude_Roll.set(data.point.x)
1844 self.varTruckAttitude_Pitch.set(data.point.y)
1845 self.varTruckAttitude_Yaw.set(data.point.z)
1846
1847 """
1848 Receive drone attitude.
1849 """
1850 def CallbackDroneAttitude(self, data):
1851
1852 self.varDroneAttitude_Roll.set(data.point.x)
1853 self.varDroneAttitude_Pitch.set(data.point.y)
1854 self.varDroneAttitude_Yaw.set(data.point.z)
1855
1856 """
1857 Receive truck position.
1858 """
1859 def CallbackTruckPosition(self, data):
1860
1861 self.varTruckPosition_North.set(data.point.x)
1862 self.varTruckPosition_East.set(data.point.y)
1863 self.varTruckPosition_Down.set(data.point.z)
1864
1865 """
1866 Receive drone position.
1867 """
1868 def CallbackDronePosition(self, data):
1869
1870 self.varDronePosition_North.set(data.point.x)
1871 self.varDronePosition_East.set(data.point.y)
1872 self.varDronePosition_Down.set(data.point.z)
1873
1874 """
1875 Receive found target or not.
1876 """
1877 def CallbackFound(self, data):
1878
1879 self.varFound.set(data.pose.orientation.w)
1880
1881 """
1882 Start receive data.
1883 """
1884 def start_data(self):
1885 self.stopUpdate = False
1886
1887 """
1888 Clear data.
1889 """
1890 def clear_data(self):
1891 self.stopUpdate = True
1892
1893 """
1894 Exit.
1895 """
1896 def ExitHandler(self):
1897 self.window.destroy()
1898 self.window.quit()
1899
1900
1901 if __name__ == '__main__':
1902
1903 ui = ROSGUI()
1904 rospy.init_node('guiTest', anonymous=True)
1905
1906 """
1907 Subscirbe topics to receive data.
1908 """
1909 rospy.Subscriber("/solo/location_GPS", PointStamped, ui.CallbackDronePosition_GPS)

```

```

1910 rospy.Subscriber("/solo/velocity" , PointStamped, ui.CallbackDroneVelocity)
1911 rospy.Subscriber("/planner/truck_velocity" , PointStamped, ui.CallbackTruckVelocity)
1912 rospy.Subscriber("/truck/attitude" , PointStamped, ui.CallbackTruckAttitude)
1913 rospy.Subscriber("/solo/attitude" , PointStamped, ui.CallbackDroneAttitude)
1914 rospy.Subscriber("/truck/local_location" , PointStamped, ui.CallbackTruckPosition)
1915 rospy.Subscriber("/solo/location_local" , PointStamped, ui.CallbackDronePosition)
1916 rospy.Subscriber("/AltitudeEstimator/Altitude" , PoseStamped , ui.CallbackKF_Altitude)
1917 rospy.Subscriber("/opencv/info" , PoseStamped , ui.CallbackFound)
1918 rospy.Subscriber("/TargetTracking/TargetLocation", PoseStamped , ui.CallbackDronePosition_KF)
1919 rospy.Subscriber("/planner/state" , String , ui.CallbackState)
1920
1921 ui.window.mainloop()
1922
1923 rospy.on_shutdown(ui.ExitHandler)
1924 rospy.spin()
1925
1926 #*****
1927 # PID.py
1928
1929 import time
1930
1931 class PID:
1932 """PID Controller
1933 """
1934
1935 def __init__(self, P=0.5, I=0.0, D=0.2):
1936
1937 self.Kp = P
1938 self.Ki = I
1939 self.Kd = D
1940 self.windup_guard = 1.5
1941 # self.windup_guard = 20.0
1942 self.sample_time = 0.05
1943 self.current_time = time.time()
1944 self.last_time = self.current_time
1945
1946 self.clear()
1947
1948 def clear(self):
1949 """Clears PID computations and coefficients"""
1950 self.SetPoint = 0.0
1951
1952 self.PTerm = 0.0
1953 self.ITerm = 0.0
1954 self.DTerm = 0.0
1955 self.last_error = 0.0
1956
1957 # Windup Guard
1958 self.int_error = 0.0
1959 self.windup_guard = 10.0
1960
1961 self.output = 0.0
1962
1963 def update(self, error):
1964 """Calculates PID value for given reference feedback
1965 .. math::
1966 u(t) = K_p e(t) + K_i \int_0^t e(t)dt + K_d \{de\}/{dt}
1967 """
1968
1969 self.current_time = time.time()
1970 delta_time = self.current_time - self.last_time
1971 delta_error = error - self.last_error
1972
1973 # if (delta_time >= self.sample_time):
1974 self.PTerm = self.Kp * error
1975 self.ITerm += error * delta_time
1976
1977 if (self.ITerm < -self.windup_guard):
1978 self.ITerm = -self.windup_guard
1979 elif (self.ITerm > self.windup_guard):
1980 self.ITerm = self.windup_guard
1981
1982 # reset the I Term
1983 if (self.last_error < 0 != error < 0):
1984 self.ITerm = 0
1985
1986 if delta_time > 0: # safety check although this should always be true
1987 self.DTerm = delta_error / delta_time
1988 else:
1989 self.DTerm = 0.0
1990
1991 # Remember last time and last error for next calculation
1992 self.last_time = self.current_time

```

```

1993 self.last_error = error
1994
1995 self.output = self.PTerm + (self.Ki * self.ITerm) + (self.Kd * self.DTerm)
1996
1997 return self.output, self.PTerm, self.Ki * self.ITerm, self.Kd * self.DTerm
1998
1999
2000 def setKp(self, proportional_gain):
2001 """Determines how aggressively the PID reacts to the current error with setting Proportional Gain"""
2002 self.Kp = proportional_gain
2003
2004 def setKi(self, integral_gain):
2005 """Determines how aggressively the PID reacts to the current error with setting Integral Gain"""
2006 self.Ki = integral_gain
2007
2008 def setKd(self, derivative_gain):
2009 """Determines how aggressively the PID reacts to the current error with setting Derivative Gain"""
2010 self.Kd = derivative_gain
2011
2012 def setWindup(self, windup):
2013 """Integral windup, also known as integrator windup or reset windup,
2014 refers to the situation in a PID feedback controller where
2015 a large change in setpoint occurs (say a positive change)
2016 and the integral terms accumulates a significant error
2017 during the rise (windup), thus overshooting and continuing
2018 to increase as this accumulated error is unwound
2019 (offset by errors in the other direction).
2020 The specific problem is the excess overshooting.
2021 """
2022 self.windup_guard = windup
2023
2024 def setSampleTime(self, sample_time):
2025 """PID that should be updated at a regular interval.
2026 Based on a pre-determined sampe time, the PID decides if it should compute or return immediately.
2027 """
2028 self.sample_time = sample_time
2029
2030 # *****
2031 # Planner.py
2032
2033 #!/usr/bin/env python
2034 # -*- coding: utf-8 -*-
2035
2036 import time
2037 import math
2038 import tkinter as tk
2039 import atexit
2040
2041 import rospy
2042 from geometry_msgs.msg import PointStamped, PoseStamped
2043 from std_msgs.msg import String
2044 from dronekit import connect, VehicleMode, LocationLocal, LocationGlobalRelative, LocationGlobal
2045
2046 from Solo import *
2047 from position_controller import *
2048 import logging
2049 import sys
2050
2051 from io import StringIO, BufferedWriter
2052
2053 '''
2054 Set the initial conditions
2055 '''
2056
2057 # Note: the units is meters, meters pre second, degrees
2058
2059 # If IsDataCollectingEnable == True,
2060 # the drone will only record and publish all data and ignore any control command.
2061 # If IsDataCollectingEnable == False, the drone will do the normal flight
2062 IsDataCollectingEnable = False
2063
2064 # Choose the truckMode = 'moving',
2065 # the searching point will use the moving point list and add the target velocity to the position control
2066 # Choose the truckMode = 'stationary', the searching point will use the stationary point list
2067 truckMode = 'moving'
2068
2069 # Choose the gimbalMode = 'move', the gimbal can moving base on the opencv data
2070 # Choose the gimbalMode = 'fix', the gimbal will always point down
2071 gimbalMode = 'fix'
2072
2073
2074 # Set velocity in different states
2075 # Initial velocity only works on take-off state and fly-to-target state.

```

```

2076 initial_velocity = 3
2077 initial_velocity_down = 0.5
2078
2079 searching_velocity_limitation = 2
2080 searching_velocity_limitation_down = 0.5
2081
2082 approaching_velocity_limitation = 1
2083
2084 landing_velocity_limitation = 0.5
2085 landing_velocity_limitation_down = 0.5
2086
2087 # Set search point detials
2088 search_height_max = 3
2089 search_height_min = 2
2090 search_height_stationary = 2
2091
2092
2093 # Set the take off desired height and the acceptable landing height for disarm the motors
2094 TakeOffAltitude = 3
2095 landing_altitude_error = 0.35
2096
2097
2098 # Set the log file level
2099 # record all data
2100 LoggerWarningLevel_data = logging.DEBUG
2101
2102 # record message and error information
2103 LoggerWarningLevel = logging.DEBUG
2104 # LoggerWarningLevel = logging.INFO
2105 # LoggerWarningLevel = logging.WARNING
2106 # LoggerWarningLevel = logging.ERROR
2107
2108
2109
2110 # The Planner class is used to control drone land on the target base on the OpenCV and target GPS data.
2111 class Planner:
2112
2113 # Initial class variables, publishers, logger.
2114 def __init__(self):
2115
2116 """
2117 Class init variables
2118 """
2119 self.solo = Solo()
2120 self.positionController = PositionController()
2121 self.countNumber = 0
2122
2123 """
2124 Publishers setup
2125 """
2126 self.markerLocaPub = rospy.Publisher("/opencv/marker_location", PointStamped, queue_size=10)
2127 self.vehicleGpsPub = rospy.Publisher("/solo/location_GPS", PointStamped, queue_size=10)
2128 self.vehicleLocalLocationPub = rospy.Publisher("/solo/location_local", PointStamped, queue_size=10)
2129 self.vehicleAttitudePub = rospy.Publisher("/solo/attitude", PointStamped, queue_size=10)
2130 self.vehicleVelocityPub = rospy.Publisher("/solo/velocity", PointStamped, queue_size=10)
2131 self.truckLocalLocationPub = rospy.Publisher("/truck/local_location", PointStamped, queue_size=10)
2132 self.truckVelocityPub= rospy.Publisher("/planner/truck_velocity", PointStamped, queue_size=10)
2133 self.statePub= rospy.Publisher("/planner/state", String, queue_size=10)
2134
2135 """
2136 Common-use global variables
2137 """
2138 self.altitude = 0
2139
2140 self.maxNorthVelocity = 0
2141 self.maxEastVelocity = 0
2142 self.maxDownVelocity = 0
2143
2144 self.home_location_north = 0
2145 self.home_location_east = 0
2146
2147 self.truck_GPS_lat = 0
2148 self.truck_GPS_lon = 0
2149 self.truck_GPS_alt = 0
2150
2151 self.truck_local_north = 0
2152 self.truck_local_east = 0
2153 self.truck_local_down = 0
2154 self.truck_velocity_north = 0
2155 self.truck_velocity_east = 0
2156
2157 self.truck_attitude_yaw = 0
2158

```

```

2159 self.target_location_north = 0
2160 self.target_location_east = 0
2161
2162 """
2163 OpenCV init variables
2164 """
2165 self.currentMarkerID = 0
2166
2167 self.marker_location_north = 0
2168 self.marker_location_east = 0
2169 self.marker_location_down = 0
2170 self.distance_north = 0
2171 self.distance_east = 0
2172 self.distance_down = 0
2173
2174 self.gimbal_pitch_current = 0
2175 self.gimbal_pitch_desired = -45
2176
2177 self.opencv_is_target_found = False
2178 self.opencv_is_target_found_b = 0
2179
2180 """
2181 Gimbal control init variables
2182 """
2183 # gimbal pitch control from 0 to -90 degrees
2184 # 0 means the camera point to front
2185 # -90 means the camera point to down/ground
2186 self.gimbal_pitch_stairs = -90
2187 self.isGoingDown = True
2188
2189 # Value for cricle search
2190 # Max index is 8
2191 self.search_location_index = 0
2192
2193 self.search_location_north = 0
2194 self.search_location_east = 0
2195
2196 """
2197 Set search point
2198 """
2199 self.searchTimes = 0
2200
2201 # -----
2202 # Stationary search point 0 to 8:
2203 #
2204 # 7----->8
2205 # ^
2206 # | 3
2207 # | ^ \
2208 # | / \
2209 # | / \
2210 # | 2 ^ 0 4
2211 # | \ | \
2212 # | \ | \
2213 # | v \
2214 # | 1 \
2215 # | v
2216 # 6<-----5
2217 #
2218 # Note: index 0 means the target location
2219
2220 self.search_point_north = [0, -1.5, 0, 1.5, 0, -2.5, -2.5, 2.5, 2.5]
2221 self.search_point_east = [0, 0, -1.5, 0, 1.5, 2.5, -2.5, -2.5, 2.5]
2222 self.search_point_down = [search_height_stationary, search_height_stationary, \
2223 search_height_stationary, search_height_stationary, search_height_stationary, search_height_stationary, \
2224 search_height_stationary, search_height_stationary, search_height_stationary]
2225
2226 self.search_point_north_moving = [0, 0, 0, 0, 0, 0, 0, 0, 0]
2227 self.search_point_east_moving = [0, 0, 0, 0, 0, 0, 0, 0, 0]
2228 self.search_point_down_moving = [search_height_min, search_height_max, search_height_min, search_height_max, \
2229 search_height_min, search_height_max, search_height_min, search_height_max, search_height_min]
2230 # -----
2231 """
2232 User control variables
2233 """
2234 self.user_desired_north = 0
2235 self.user_desired_east = 0
2236 self.user_desired_down = 0
2237 self.user_desired_yaw = 0
2238
2239 self.user_last_msg_time = 0
2240
2241 self.start_flying = False

```

```

2242 self.stop_flying = False
2243 self.return_to_home = False
2244
2245 """
2246 State control variables
2247 """
2248 self.CurrentState = "StandByState"
2249 self.CurrentStateIndex = 0
2250 self.state_start_time = 0
2251
2252 """
2253 Log init variables
2254 """
2255
2256 self.logger = logging.getLogger('Planner_message')
2257 self.fileHandler_message = logging.StreamHandler(BufferedWriter(FileIO("planner_message" + \
2258 time.strftime("%Y%m%d-%H%M%S") + ".log", "w"))))
2259 self.logger.addHandler(self.fileHandler_message)
2260 self.formatter_message = logging.Formatter('%(asctime)s %(levelname)s %(message)s')
2261 self.fileHandler_message.setFormatter(self.formatter_message)
2262 self.logger.setLevel(LoggerWarningLevel)
2263
2264
2265 self.logger_data = logging.getLogger('Planner_data')
2266 self.fileHandler_data = logging.StreamHandler((FileIO("planner_data" + time.strftime("%Y%m%d-%H%M%S") + ".log", "w"))))
2267 self.logger_data.addHandler(self.fileHandler_data)
2268 self.formatter_data = logging.Formatter('%(message)s')
2269 self.fileHandler_data.setFormatter(self.formatter_data)
2270 self.logger_data.setLevel(LoggerWarningLevel_data)
2271 self.logger_data.info('Time,CurrState;/TYaw,VYaw;/TNorth,TEast,MNorth,MEast,TGNorth,TGeast,VNorth,VEast,VDown,Alt;\
2272 /DNorth,DEast,DDown,VVeloX,VVeloY,VVeloZ,Found,MID')
2273
2274
2275 """
2276 Function: GimbalActionDecider
2277 Input : pitch, roll=0, yaw=0
2278 Note: roll and yaw cannot be controlled with Solo's gimbal
2279
2280 Here define the gimbal movements, which suits all the states.
2281 The inner mechanism is everytime needs to rotate? Replace it with -90.
2282 """
2283 def GimbalActionDecider(self, pitch, roll=0, yaw=0):
2284
2285 if gimbalMode == 'fix':
2286 self.solo.SetGimbalPitchDeg(-90, roll, yaw)
2287 self.gimbal_pitch_stairs = -95
2288
2289
2290 if gimbalMode == 'move':
2291 self.solo.SetGimbalPitchDeg(pitch, roll, yaw)
2292
2293
2294 '''
2295 Function: CheckDistance
2296 Input : hori_dist, limit_dist
2297 Output : The Boolean value
2298 Here define the horizon distance between target and drone as a condition to decide the state transfer
2299 '''
2300 def CheckDistance(self, hori_dist, limit_dist):
2301
2302 return True if hori_dist < limit_dist else False
2303
2304
2305 '''
2306 Function: RunStandByStateTask
2307 Output : return 'state'
2308 RunStandByStateTask is used to connect with drone, then arm the drone and save the home local location.
2309 '''
2310 def RunStandByStateTask(self):
2311
2312 # Build communication with drone
2313 self.solo.Connect(rate = 12, mode = 'Solo')
2314
2315 # Enable the Data Collecting mode which will only record all data and don't send any control commands
2316 if IsDataCollectingEnable:
2317 return 'LoiteringState'
2318 else:
2319 # Start flight and begin arm the drone
2320 self.solo.Arm()
2321 if self.solo.IsArmed:
2322
2323
2324 cmds = self.solo.vehicle.commands

```

```

2325 cmds.download()
2326 cmds.wait_ready()
2327
2328 if not self.solo.vehicle.home_location:
2329 self.logger.debug(' Waiting for home location ...')
2330 return 'StandByState'
2331 else:
2332 # Save the home local location
2333 msgVehicleLocalLocation = self.solo.GetLocalPosition()
2334 self.home_location_north = msgVehicleLocalLocation.point.x
2335 self.home_location_east = msgVehicleLocalLocation.point.y
2336
2337 self.logger.debug('home local location north&east:%f,%f\' \
2338 %(self.home_location_north,self.home_location_east))
2339
2340 self.solo.vehicle.mode = VehicleMode("GUIDED")
2341
2342 return 'TakingOffState'
2343
2344 else:
2345 return 'StandByState'
2346
2347 '''
2348 Function: RunTakingOffStateTask
2349 Output : return 'state'
2350 RunTakingOffStateTask is used to take off and make sure the drone reaches the desired height.
2351 When the drone reaches the desired height, drone will wait the self.start_flying == True.
2352 if self.start_flying == True, the state will change to 'Flying2TargetState'.
2353 Otherwise, the drone will keep the desired height.
2354 '''
2355 def RunTakingOffStateTask(self):
2356
2357 self.SetMaxVelocityNed(north_velocity = initial_velocity, east_velocity = initial_velocity, \
2358 down_velocity = initial_velocity_down)
2359
2360 altitude = self.solo.GetAltitude()
2361
2362 if altitude < 0.5:
2363 if self.countNumber > 10:
2364 self.solo.vehicle.mode = VehicleMode("GUIDED")
2365 self.solo.TakeOff(TakeOffAltitude)
2366 self.countNumber = 0
2367 else:
2368 self.countNumber += 1
2369 return 'TakingOffState'
2370
2371 else:
2372 self.logger.debug('\n ultrasonic distance & take off altitude * 0.85:%f ,%f', \
2373 self.altitude, TakeOffAltitude * 0.85)
2374 vertical_distance = self.altitude - TakeOffAltitude
2375 self.FlyToTargetNed(north_distance = 0, east_distance = 0, down_distance = vertical_distance)
2376
2377 if self.altitude < TakeOffAltitude * 0.85:
2378 return 'TakingOffState'
2379 else:
2380 if self.start_flying:
2381 self.logger.warning('Start flying')
2382 return 'Flying2TargetState'
2383 else:
2384 self.logger.warning('Wait for start command')
2385 return 'TakingOffState'
2386
2387 '''
2388 Function: RunLoiteringStateTask
2389 Output : return 'state'
2390 In the normal situation, RunLoiteringStateTask is used to make the drone stay in the air with the desired height.
2391 When IsDataCollectingEnable = True, RunLoiteringStateTask will do nothing and keep the state in 'LoiteringState'.
2392 '''
2393 def RunLoiteringStateTask(self):
2394
2395 if self.CurrentState == 'ManualOperationState':
2396 self.logger.warning('Enter Manual Operation State')
2397 return 'ManualOperationState'
2398
2399 if self.return_to_home:
2400 self.logger.warning('Flying to home location')
2401 return 'ReturnToHomeState'
2402
2403 if IsDataCollectingEnable:
2404 self.logger.warning('Collecting data.....')
2405 else:
2406 self.logger.error ('State witch failed. Now is Loiter state, 20s later will land.')
2407

```

```

2408 # Maintain the height
2409 vertical_distance = self.altitude - TakeOffAltitude
2410 self.FlyToTargetNed(north_distance = 0, east_distance = 0, down_distance = vertical_distance)
2411
2412 return 'LoiteringState'
2413
2414
2415 '''
2416 Function: RunFlying2TargetStateTask
2417 Output : return 'state'
2418 RunFlying2TargetStateTask is according the target GPS location to calculate the horizontal distance
2419 between target and drone in north and east.
2420 During the flight, the drone will keep the height (same as the take-off height).
2421 When the total horizontal distance less than 1m, the state will change to 'Searching4TagState'.
2422 '''
2423 def RunFlying2TargetStateTask(self):
2424
2425 # Prepare for the User Control Mode
2426 if self.CurrentState == 'ManualOperationState':
2427 self.logger.warning('Enter Manual Operation State')
2428 return 'ManualOperationState'
2429
2430 if self.stop_flying:
2431 self.logger.warning('Stop flying, enter the Loitering State from Flying2TargetState')
2432 return 'LoiteringState'
2433
2434 # Calculate & Print out info to control
2435 horizontal_distance, dNorth, dEast = self.GetHorizontalDistanceAndNedComponents(self.target_location_north,\
2436 self.target_location_east)
2437 self.logger.debug("vehicle_yaw, velocity_x, velocity_y, velocity_z:%f,%f,%f,%f" %(self.solo.vehicle.attitude.yaw,\
2438 self.solo.vehicle.velocity[0], self.solo.vehicle.velocity[1], self.solo.vehicle.velocity[2]))
2439
2440 """
2441 Main control part for this state
2442 """
2443 self.GimbalActionDecider(self.gimbal_pitch_stairs, 0, 0)
2444
2445 # Turn to desired yaw
2446 self.solo.SetYawDeg(self.truck_attitude_yaw)
2447
2448 # Fly to target & Maintain the height
2449 vertical_distance = self.altitude - TakeOffAltitude
2450 self.FlyToTargetNed(north_distance = dNorth, east_distance = dEast, down_distance = vertical_distance)
2451
2452 """
2453 End of control part for this state
2454 """
2455
2456 # Conditions to switch state
2457 if self.opencv_is_target_found:
2458 return 'Searching4TagState'
2459
2460 # For stationary target, we check if hori-dist<1m, then we switch state. Also time limit.
2461 # For moving target, we just let it infinitely approach to the target
2462 ret = self.CheckDistance(horizontal_distance, limit_dist = 1)
2463 if ret: return 'Searching4TagState'
2464
2465 return 'Flying2TargetState'
2466
2467
2468 '''
2469 Function: SearchTarget
2470 According to the search point list to calculate the search point location, and fly to the search point.
2471 For stationary target, it will do a square searching without the height changing.
2472 For moving target, it will be using the target GPS to following the target and change the height for the searching.
2473 '''
2474 def SearchTarget(self):
2475 # For different Mode to calcaulate the search point location base on the search_point list.
2476 if truckMode == "moving":
2477 self.search_location_north = self.target_location_north + \
2478 self.search_point_north_moving[self.search_location_index]
2479 self.search_location_east = self.target_location_east + \
2480 self.search_point_east_moving[self.search_location_index]
2481 horizontal_distance, dNorth, dEast = self.GetHorizontalDistanceAndNedComponents(self.search_location_north,\
2482 self.search_location_east)
2483 vertical_distance = self.altitude - self.search_point_down_moving[self.search_location_index]
2484 else:
2485 self.search_location_north = self.truck_local_north + self.search_point_north[self.search_location_index]
2486 self.search_location_east = self.truck_local_east + self.search_point_east[self.search_location_index]
2487 horizontal_distance, dNorth, dEast = self.GetHorizontalDistanceAndNedComponents(self.search_location_north,\
2488 self.search_location_east)
2489 vertical_distance = self.altitude - self.search_point_down[self.search_location_index]
2490

```

```

2491 self.FlyToTargetNed(north_distance = dNorth, east_distance = dEast, down_distance = vertical_distance)
2492
2493 self.logger.debug("self.search_location_north & self.search_location_east:%f,%f" %(self.search_location_north,\
2494 self.search_location_east))
2495 self.logger.debug('vehicle_yaw, velocity_x, velocity_y, velocity_z: %f %f %f %f ', self.solo.vehicle.attitude.yaw,\
2496 self.solo.vehicle.velocity[0], self.solo.vehicle.velocity[1], self.solo.vehicle.velocity[2])
2497
2498 # For different Mode to check if drone reaches the search point.
2499 # if reaching, update the search_point. Otherwise, wait the drone reaches the location.
2500 if truckMode == "moving":
2501 reach_altitude = abs(min(0.2, max(-0.2, vertical_distance)))
2502 if reach_altitude != 0.2 :
2503 self.GimbalActionDecider(self.gimbal_pitch_stairs, 0, 0)
2504 self.solo.SetYawDeg(self.truck_attitude_yaw)
2505 self.search_location_index += 1
2506 self.logger.debug('search index: %f %f %f %f',self.search_location_index,reach_altitude,\
2507 self.search_point_down_moving[self.search_location_index],self.searchTimes)
2508 else:
2509 if horizontal_distance < 0.3:
2510 self.GimbalActionDecider(self.gimbal_pitch_stairs, 0, 0)
2511 self.solo.SetYawDeg(self.truck_attitude_yaw)
2512 self.search_location_index += 1
2513 self.logger.debug('search index: %f %f %f %f',self.search_location_index,horizontal_distance,\
2514 self.search_point_down[self.search_location_index],self.searchTimes)
2515
2516
2517 '''
2518 Function: RunSearching4TagStateTask
2519 Output : return 'state'
2520 RunSearching4TagStateTask is used to find the target label with the camera.
2521 During the searching processing, if the drone finds the target label, it will change to Approaching2TagState.
2522 Otherwise, it will still do the searching point.
2523 If the searching list is used by 3 times, the drone will enter the LandingState and land base on the target GPS location.
2524 '''
2525 def RunSearching4TagStateTask(self):
2526
2527 # Initial some parameters
2528
2529 self.SetMaxVelocityNed(north_velocity = searching_velocity_limition, east_velocity = searching_velocity_limition,\
2530 down_velocity = searching_velocity_limition_down)
2531
2532 # Prepare for the User Control Mode
2533 if self.CurrentState == 'ManualOperationState':
2534 self.logger.warning('Enter Manual Operation State')
2535 return 'ManualOperationState'
2536
2537 if self.stop_flying:
2538 self.logger.warning('Stop flying, enter the Loitering State from Searching4TagState')
2539 return 'LoiteringState'
2540
2541 # If target found, go to approach state
2542 if self.opencv_is_target_found:
2543 return 'Approaching2TagState'
2544
2545 """
2546 Main control part for this state
2547 """
2548
2549 if self.search_location_index > 0 and self.search_location_index < 8:
2550 self.SearchTarget()
2551 else:
2552
2553 self.searchTimes += 1
2554 self.search_location_index += 1
2555 self.FlyToTargetNed(north_distance = 0, east_distance = 0, down_distance = 0)
2556 self.logger.warning("Can not find target. Re-searching...")
2557
2558 """
2559 End of control part for this state
2560 """
2561
2562 # Conditions to switch state
2563 # ret = self.TruckModeDecider(hori_dist=2, limit_dist = 1, limitTime=180)
2564 # if ret: return 'LandingState'
2565
2566 # If finished 3 times, then land
2567 if self.searchTimes > 3:
2568 self.searchTimes = 0
2569 return 'LandingState'
2570 else:
2571 if self.search_location_index > 8:
2572 self.search_location_index = 0
2573 return 'Flying2TargetState'

```

```

2574 else:
2575 return 'Searching4TagState'
2576
2577 '''
2578
2579 Function: RunApproaching2TagTask
2580 Output : return 'state'
2581 RunApproaching2TagTask is used to reduce the horizontal distance. It will make preparations for the 'LandingState'.
2582 When the total horizontal_distance is less than 1.5m, it will change to 'LandingState'.
2583 Otherwise, it will continue to reduce the distance.
2584 If the drone lost the target label, it will change to 'Flying2TargetState'.
2585 '''
2586 def RunApproaching2TagTask(self):
2587
2588 # Prepare for the User Control Mode
2589 if self.CurrentState == 'ManualOperationState':
2590 self.logger.warning('Enter Manual Operation State from approaching state')
2591 return 'ManualOperationState'
2592
2593 if self.stop_flying:
2594 self.logger.warning('Stop flying, enter the Loitering State')
2595 return 'LoiteringState'
2596
2597 self.SetMaxVelocityNed(north_velocity = approaching_velocity_limitation,\
2598 east_velocity = approaching_velocity_limitation, down_velocity = 0.5)
2599 # Calculate & Print out info to control
2600
2601 horizontal_distance, dNorth, dEast = self.GetHorizontalDistanceAndNedComponents(self.target_location_north,\
2602 self.target_location_east)
2603
2604 """
2605 Main control part for this state
2606 """
2607 self.GimbalActionDecider(self.gimbal_pitch_desired, 0, 0)
2608 # # Make the gimbale follow the target.
2609 self.solo.SetYawDeg(self.truck_attitude_yaw)
2610 # Maintain height
2611 vertical_distance = self.altitude - TakeOffAltitude
2612 self.FlyToTargetNed(north_distance = self.distance_north, east_distance = self.distance_east,\
2613 down_distance = vertical_distance)
2614 self.logger.debug('vehicle_yaw, velocity_x, velocity_y, velocity_z: %f %f %f %f ', self.solo.vehicle.attitude.yaw,\
2615 self.solo.vehicle.velocity[0], self.solo.vehicle.velocity[1], self.solo.vehicle.velocity[2])
2616 """
2617 End of control part for this state
2618 """
2619
2620 # Conditions to switch state
2621 # For stationary target, we check if hori-dist<1.5m, then we switch state. Also time limit.
2622 # For moving target, we just let it infinitely approach to the target
2623 ret = self.CheckDistance(horizontal_distance, limit_dist = 1.5)
2624 if ret: return 'LandingState'
2625
2626 if not self.opencv_is_target_found: return 'Flying2TargetState'
2627
2628 return 'Approaching2TagState'
2629
2630 '''
2631
2632 Function: RunLandingStateTask
2633 Output : return 'state'
2634 RunLandingStateTask is used to land with the target location.
2635 During the landing, the drone will still be controlled for reducing the distance between the drone and target.
2636 During the landing, if the target label lost and the height great than landing_altitude_error
2637 it will change to the Flying2TargetState.
2638 If the height less than landing_altitude_error (base on which distance the camera cannot detect the target)
2639 it will land with the last target location.
2640 '''
2641 def RunLandingStateTask(self):
2642
2643 if self.stop_flying:
2644 self.logger.warning('Stop flying, enter the Loitering State from landing state')
2645 return 'LoiteringState'
2646
2647 self.SetMaxVelocityNed(north_velocity = landing_velocity_limitation, east_velocity = landing_velocity_limitation,\
2648 down_velocity = landing_velocity_limitation_down)
2649 # Make the gimbale follow the target.
2650 self.solo.SetYawDeg(self.truck_attitude_yaw)
2651 self.GimbalActionDecider(self.gimbal_pitch_desired, 0, 0)
2652
2653 # Calculate & Print out info to control
2654 horizontal_distance, dNorth, dEast = self.GetHorizontalDistanceAndNedComponents(self.target_location_north,\
2655 self.target_location_east)

```

```

2657
2658
2659 """
2660 Main control part for this state
2661 """
2662
2663 self.logger.debug('vehicle_yaw, velocity_x, velocity_y, velocity_z: %f %f %f %f ' \
2664 %(self.solo.vehicle.attitude.yaw, self.solo.vehicle.velocity[0], \
2665 self.solo.vehicle.velocity[1], self.solo.vehicle.velocity[2]))
2666
2667 if self.opencv_is_target_found:
2668 desiredHeight = self.altitude if horizontal_distance < 0.5 else 0
2669
2670 if self.altitude < landing_altitude_error:
2671 self.SimpleLand()
2672 if self.altitude < 0.1:
2673 self.Disarm()
2674 return 'LandingState'
2675
2676 self.FlyToTargetNed(north_distance = dNorth, east_distance = dEast, down_distance = desiredHeight)
2677
2678 else:
2679 if self.altitude < landing_altitude_error:
2680 self.SimpleLand()
2681 if self.altitude < 0.1:
2682 self.Disarm()
2683 return 'LandingState'
2684 else:
2685 if truckMode == 'moving': return 'Flying2TargetState'
2686 desiredHeight = self.altitude if horizontal_distance < 0.5 else 0
2687 self.FlyToTargetNed(north_distance = dNorth, east_distance = dEast, down_distance = desiredHeight)
2688 """
2689 End of control part for this state
2690 """
2691
2692 # Conditions to switch state
2693 return 'LandingState'
2694
2695
2696 '''
2697 Function: RunManualOperationState
2698 Output : return 'state'
2699 RunManualOperationState is used to control the drone's moving with the keyboard commands.
2700 Base on the callback data to control the drone's horizontal moving and change the altitude.
2701 It also can control the gimbal and the drone's yaw.
2702 If the callback data don't update after 10s, it will change to 'Searching4TagState'.
2703 '''
2704
2705 def RunManualOperationState(self):
2706
2707 # Calculate & Print out info to control
2708 horizontal_distance, dNorth, dEast = self.GetHorizontalDistanceAndNedComponents(self.user_desired_north, \
2709 self.user_desired_east)
2710 dDown = self.altitude - self.user_desired_down
2711
2712 """
2713 Main control part for this state
2714 """
2715
2716 self.GimbalActionDecider(pitch, 0, 0)
2717
2718 self.solo.SetYawDeg(self.user_desired_yaw)
2719
2720 # fly to desired location
2721 self.FlyToTargetNed(dNorth, dEast, dDown)
2722
2723 """
2724 End of control part for this state
2725 """
2726
2727 # Conditions to switch state
2728 current_time = time.time()
2729 return 'Searching4TagState' if current_time - self.user_last_msg_time > 10 else 'ManualOperationState'
2730
2731 '''
2732
2733 Function: ReturnToHomeState
2734 Output : return 'state'
2735 ReturnToHomeState is used to control the drone go back to the home location and then land.
2736 According the callback command, if the state change to ReturnToHomeState, it will not change anymore.
2737 '''
2738
2739 def ReturnToHomeState(self):
2740
2741 # Prepare for the User Control Mode
2742 if self.CurrentState == 'ManualOperationState':
2743 self.logger.warning('Enter Manual Operation State')
2744 return 'ManualOperationState'
2745

```

```

2740 if self.stop_flying:
2741 self.logger.warning('Stop flying, enter the Loitering State from return to home state')
2742 return 'LoiteringState'
2743
2744 self.logger.warning('Return to home')
2745 self.SetMaxVelocityNed(north_velocity = landing_velocity_limition, east_velocity = landing_velocity_limition,\
2746 down_velocity = landing_velocity_limition_down)
2747 # Calculate & Print out info to control
2748
2749 horizontal_distance, dNorth, dEast = self.GetHorizontalDistanceAndNedComponents(self.home_location_north,\
2750 self.home_location_east)
2751 self.logger.debug('vehicle_yaw, velocity_x, velocity_y, velocity_z: %f %f %f %f ' %(self.solo.vehicle.attitude.yaw,\
2752 self.solo.vehicle.velocity[0], self.solo.vehicle.velocity[1], self.solo.vehicle.velocity[2]))
2753
2754 """
2755 Main control part for this state
2756 """
2757
2758 self.GimbalActionDecider(self.gimbal_pitch_stairs, 0, 0)
2759 # Turn to desired yaw
2760 self.solo.SetYawDeg(self.truck_attitude_yaw)
2761
2762 desiredHeight = self.altitude - 0.2 if horizontal_distance < 0.5 else 0
2763 self.FlyToTargetNed(north_distance = dNorth, east_distance = dEast, down_distance = desiredHeight)
2764
2765 if self.altitude < landing_altitude_error:
2766 self.SimpleLand()
2767 if self.altitude < 0.1:
2768 self.Disarm()
2769 """
2770 End of control part for this state
2771 """
2772 # Conditions to switch state
2773
2774 return 'ReturnToHomeState'
2775
2776 '''
2777 Function: RunStateMachine
2778 RunStateMachine is running at 20Hz, every loop it will decide which state it needs running.
2779 '''
2780 def RunStateMachine(self):
2781
2782 if self.CurrentState == "StandByState":
2783 nextState = self.RunStandByStateTask()
2784 self.CurrentStateIndex = 0
2785
2786 elif self.CurrentState == "TakingOffState":
2787 nextState = self.RunTakingOffStateTask()
2788 self.CurrentStateIndex = 1
2789
2790 elif self.CurrentState == "LoiteringState":
2791 nextState = self.RunLoiteringStateTask()
2792 self.CurrentStateIndex = 2
2793
2794 elif self.CurrentState == "Flying2TargetState":
2795 nextState = self.RunFlying2TargetStateTask()
2796 self.CurrentStateIndex = 3
2797
2798 elif self.CurrentState == "Searching4TagState":
2799 nextState = self.RunSearching4TagStateTask()
2800 self.CurrentStateIndex = 4
2801
2802 elif self.CurrentState == "Approaching2TagState":
2803 nextState = self.RunApproaching2TagTask()
2804 self.CurrentStateIndex = 5
2805
2806 elif self.CurrentState == "LandingState":
2807 nextState = self.RunLandingStateTask()
2808 self.CurrentStateIndex = 6
2809
2810 elif self.CurrentState == "ManualOperationState":
2811 nextState = self.RunManualOperationState()
2812 self.CurrentStateIndex = 7
2813
2814 elif self.CurrentState == "ReturnToHomeState":
2815 nextState = self.ReturnToHomeState()
2816 self.CurrentStateIndex = 8
2817
2818 else: # since something is wrong here, stay where it is.
2819 nextState = self.RunLoiteringStateTask()
2820
2821 if (self.CurrentState != nextState):
2822 self.state_start_time = time.time()

```

```

2823 self.logger.warning ('\n\n State Transition Current: %s ---->>> Next state: %s', \
2824 self.CurrentState, nextState)
2825 self.logger.warning ('\n\n Current State: %s ', nextState)
2826
2827 self.statePub.publish(nextState)
2828
2829 self.CurrentState = nextState
2830
2831
2832 """
2833 Function: FlyToTargetNed
2834 Input: north_distance, east_distance, down_distance.
2835 Base on the input data fly to the desired location with control velocity of drone.
2836
2837 Note: this function is using PID, so it needs to update the distance every 0.05s
2838 to achieve position control.
2839 =====
2840 Example:
2841
2842 FlyToTargetNed(north_distance = 1, east_distance = 1, down_distance = 0)
2843
2844 ++++++
2845 The drone from current location fly to north and east with specifical
2846 velocity(calculate the input distance).
2847
2848 Note: if north_distance/east_distance great than zero, the drone fly to
2849 north and east, otherwise, the drone fly to south and west.
2850 if down_distance equal zero, the drone will keep current altitude.
2851 if down_distance less than zero, the drone will fly up.
2852 if down_distance great than zero, the drone will fly down.
2853 =====
2854 """
2855 def FlyToTargetNed(self, north_distance, east_distance, down_distance = 0):
2856
2857 # calculate control values by PID with desired distance
2858 northVelocity, eastVelocity, downVelocity = self.positionController.UpdateDistance2Target3d(\
2859 north_distance,east_distance, down_distance)
2860
2861 # limit the velocity
2862 northVelocity = min(self.maxNorthVelocity, max(-self.maxNorthVelocity, northVelocity))
2863 eastVelocity = min(self.maxEastVelocity, max(-self.maxEastVelocity, eastVelocity))
2864 downVelocity = min(self.maxDownVelocity, max(-self.maxDownVelocity, downVelocity))
2865
2866 # add the target velocity for keep the drone and target be relatively static
2867 northVelocity_real = self.truck_velocity_north + northVelocity
2868 eastVelocity_real = self.truck_velocity_east + eastVelocity
2869
2870 # Conditions to switch the control method
2871 if (self.CurrentState == "ReturnToHomeState" or self.CurrentState == "LoiteringState"):
2872 self.solo.SetVelocityNed(northVelocity, eastVelocity, downVelocity)
2873 else:
2874 self.solo.SetVelocityNed(northVelocity_real, eastVelocity_real, downVelocity)
2875
2876 '''
2877 Function: SetMaxVelocityNed
2878 Input: north_velocity, east_velocity, down_velocity
2879 SetMaxVelocityNed is used to set the velocity limitation for the velocity control.
2880 '''
2881 def SetMaxVelocityNed(self, north_velocity = 0, east_velocity = 0, down_velocity = 0):
2882
2883 self.maxNorthVelocity = north_velocity
2884 self.maxEastVelocity = east_velocity
2885 self.maxDownVelocity = down_velocity
2886
2887 """
2888 Function: SetDesiredVehicleYaw
2889 Input: dNorth, dEast
2890 SetDesiredVehicleYaw is used to control the vehicle yaw by the desired yaw angle from 0 to 360 degrees.
2891 """
2892 def SetDesiredVehicleYaw(self, dNorth, dEast):
2893
2894 vehicle_yaw = math.atan2(dEast, dNorth) * 57.2957795
2895
2896 if vehicle_yaw < 0:
2897 vehicle_yaw += 360.00
2898
2899 self.logger.debug('desired_vehicle_yaw: %f', vehicle_yaw)
2900 self.solo.SetYawDeg(vehicle_yaw)
2901
2902 """
2903 Function: GetHorizontalDistanceAndNedComponents
2904 Input: desired_location_north, desired_location_east
2905 Output: math.sqrt((dNorth*dNorth) + (dEast*dEast)), dNorth, dEast

```

36

```

2989 self.distance_east,
2990 self.distance_down,
2991 msgVehicleVelocity.point.x,
2992 msgVehicleVelocity.point.y,
2993 msgVehicleVelocity.point.z,
2994 self.opencv_is_target_found_b,
2995 self.currentMarkerID
2996))
2997
2998
2999 """
3000 Function: CallbackManualOperation
3001 Ros subscriber callback function is used to receive all ManualOperation data from the user_control
3002 and change state to 'ManualOperationState'.
3003 """
3004 def CallbackManualOperation(self, data):
3005
3006 self.user_desired_north = data.pose.orientation.x
3007 self.user_desired_east = data.pose.orientation.y
3008 self.user_desired_down = data.pose.orientation.z
3009 self.user_desired_yaw = math.degrees(data.pose.orientation.z)
3010 self.user_last_msg_time = time.time()
3011 self.CurrentState = 'ManualOperationState'
3012
3013 """
3014 Function: CallbackOpencvInfo
3015 Ros subscriber callback function is used to receive all OpencvInfo data from the opencv_node
3016 and calculate the target location and publish the location message.
3017 """
3018 def CallbackOpencvInfo(self, data):
3019
3020 # Save all the data
3021 self.currentMarkerID = int(data.header.frame_id)
3022 self.distance_north = data.pose.position.x
3023 self.distance_east = data.pose.position.y
3024 self.distance_down = data.pose.position.z
3025 self.gimbal_pitch_current = data.pose.orientation.x
3026 self.gimbal_pitch_desired = data.pose.orientation.y
3027 # just to record if target lost: -1=lost, 1=found
3028 self.opencv_is_target_found = data.pose.orientation.w == 1
3029 self.opencv_is_target_found_b = data.pose.orientation.w
3030
3031 # Protect mechanism: if lost, don't update
3032 if not self.opencv_is_target_found:
3033 return
3034
3035 # target local location
3036 msgVehicleLocalLocation = self.solo.GetLocalPosition()
3037 self.marker_location_north = msgVehicleLocalLocation.point.x + self.distance_north
3038 self.marker_location_east = msgVehicleLocalLocation.point.y + self.distance_east
3039 self.marker_location_down = 0
3040
3041 # distance in NED frame message
3042 msgMarkerLocation = PointStamped()
3043 msgMarkerLocation.header.frame_id = str(self.opencv_is_target_found_b)
3044 msgMarkerLocation.header.stamp = rospy.Time.now()
3045 msgMarkerLocation.point.x = self.marker_location_north
3046 msgMarkerLocation.point.y = self.marker_location_east
3047 msgMarkerLocation.point.z = self.marker_location_down
3048 self.markerLocaPub.publish(msgMarkerLocation)
3049 # # Print debug data
3050 # rospy.loginfo(msgMarkerLocation)
3051
3052 """
3053 Function: CallbackAltitudeKF
3054 Ros subscriber callback function is used to receive all AltitudeKF data from Kalman filter
3055 and use it as the drone's altitude data
3056 """
3057 def CallbackAltitudeKF(self, data):
3058
3059 x_hat0 = data.pose.position.x
3060 x_last_pred = data.pose.orientation.x
3061 self.altitude = x_hat0
3062
3063 """
3064 Function: CallbackTargetGps
3065 Ros subscriber callback function is used to receive all TargetGps data from the target to target GPS
3066 and calculate the target local frame, and publish the local location.
3067 """
3068 def CallbackTargetGps(self, data):
3069
3070 self.truck_GPS_lat = data.point.x
3071 self.truck_GPS_lon = data.point.y

```

```

3072 self.truck_GPS_alt = data.point.z
3073
3074 truck_GPS = LocationGlobalRelative(self.truck_GPS_lat, self.truck_GPS_lon, self.truck_GPS_alt)
3075
3076 truck_north_distance, truck_east_distance, truck_down_distance = self.GetDistanceGlobal2Local(truck_GPS)
3077 msgVehicleLocalLocation = self.solo.GetLocalPosition()
3078 self.truck_local_north = msgVehicleLocalLocation.point.x + truck_north_distance
3079 self.truck_local_east = msgVehicleLocalLocation.point.y + truck_east_distance
3080 # print "NED:",self.truck_local_north, self.truck_local_east, self.truck_local_down
3081
3082 msgTruckLocalLocation = PointStamped()
3083 msgTruckLocalLocation.header.stamp = rospy.Time.now()
3084 msgTruckLocalLocation.point.x = self.truck_local_north
3085 msgTruckLocalLocation.point.y = self.truck_local_east
3086 msgTruckLocalLocation.point.z = self.truck_local_down
3087 self.truckLocalLocationPub.publish(msgTruckLocalLocation)
3088 self.logger.debug("send truck_local_location")
3089
3090
3091 """
3092 Function: CallbackTargetAttitude
3093 Ros subscriber callback function is used to receive all TargetAttitude data
3094 from the target to get the target's yaw,
3095 and transfer the unit from radians to degrees.
3096 """
3097 def CallbackTargetAttitude(self, data):
3098 truck_attitude_yaw_radians = math.degrees(data.point.z)
3099
3100 if truck_attitude_yaw_radians < 0:
3101 truck_attitude_yaw_radians += 360
3102 self.truck_attitude_yaw = truck_attitude_yaw_radians
3103
3104 """
3105 Function: CallbackTargetVelocity
3106 Ros subscriber callback function is used to receive all TargetVelocity data
3107 from the target to get the target velocity
3108 and publish the velocity data
3109 if the truckMode == 'stationary', ignore the truck velocity data.
3110 """
3111 def CallbackTargetVelocity(self, data):
3112 if truckMode == 'stationary':
3113 self.truck_velocity_north = 0
3114 self.truck_velocity_east = 0
3115 else:
3116 self.truck_velocity_north = data.point.x
3117 self.truck_velocity_east = data.point.y
3118
3119 msgTruckVelocity = PointStamped()
3120 msgTruckVelocity.header.stamp = rospy.Time.now()
3121 msgTruckVelocity.point.x = data.point.x
3122 msgTruckVelocity.point.y = data.point.y
3123 msgTruckVelocity.point.z = data.point.z
3124 self.truckVelocityPub.publish(msgTruckVelocity)
3125 self.logger.debug("send truck_velocity")
3126
3127 """
3128 Function: CallbackTargetLocationKF
3129 Ros subscriber callback function is used to receive all TargetLocationKF data
3130 from Kalman filter to get target local location.
3131 """
3132 def CallbackTargetLocationKF(self, data):
3133
3134 self.target_location_north = data.pose.orientation.y
3135 self.target_location_east = data.pose.orientation.z
3136 self.logger.debug("recieved the target_location")
3137
3138 """
3139 Function: CallbackUserCommand
3140 Ros subscriber callback function is used to receive all CallbackUserCommand data from user_command,
3141 and transfer them to the Boolean value.
3142 """
3143 def CallbackUserCommand(self, data):
3144
3145 self.start_flying = data.point.x == 1
3146 self.stop_flying = data.point.y == 1
3147 self.return_to_home = data.point.z == 1
3148
3149 """
3150 Function: TimerCallback
3151 Ros Timer Callback function, every 0.05s running the TimerCallback once.
3152 """
3153 def TimerCallback(self, event):
3154 self.RunStateMachine()

```

```

3155 self.PublishVehicleMsg()
3156
3157 """
3158 Function: sigint_handler
3159 Close the logger
3160 """
3161 def sigint_handler(self):
3162 self.logger.waring("shutdown!")
3163 self.positionController.ExitHandler()
3164 logging.shutdown()
3165 """
3166 Function: main
3167 Intital the Planner class and ros nod, and Setup all subscriber.
3168 Rununing all control code at 20Hz.
3169 """
3170 def main():
3171
3172 planner = Planner()
3173
3174 rospy.init_node('Planner', anonymous=True)
3175
3176 # Setup subscriber
3177 rospy.Subscriber("/opencv/info", PoseStamped, planner.CallbackOpencvInfo, queue_size=2)
3178 rospy.Subscriber("/TargetTracking/TargetLocation", PoseStamped, planner.CallbackTargetLocationKF, queue_size=2)
3179
3180 rospy.Subscriber("/user/command_data", PointStamped, planner.CallbackUserCommand, queue_size=2)
3181 rospy.Subscriber("/user/drone_pose_desired", PoseStamped, planner.CallbackManualOperation, queue_size=2)
3182
3183 rospy.Subscriber("/truck/location_GPS", PointStamped, planner.CallbackTargetGps, queue_size=2)
3184 rospy.Subscriber("/truck/attitude", PointStamped, planner.CallbackTargetAttitude, queue_size=2)
3185 rospy.Subscriber("/truck/velocity", PointStamped, planner.CallbackTargetVelocity, queue_size=2)
3186
3187 rospy.Subscriber("/user/drone_pose_desired", PoseStamped, planner.CallbackManualOperation, queue_size=2)
3188 rospy.Subscriber("/AltitudeEstimator/Altitude", PoseStamped, planner.CallbackAltitudeKF, queue_size=2)
3189
3190
3191 # Use timer interupt to make sure State Machine running as 20Hz
3192 dTimeStep = 0.05
3193 rospy.Timer(rospy.Duration(dTimeStep), planner.TimerCallback)
3194
3195 rospy.on_shutdown(planner.sigint_handler)
3196 rospy.spin()
3197
3198
3199
3200 if __name__ == '__main__':
3201
3202 main()
3203
3204 # *****
3205 # position_controller.py
3206
3207 #!/usr/bin/env python
3208 # -*- coding: utf-8 -*-
3209
3210 import time
3211 import math
3212 import atexit
3213 import PID
3214 import logging
3215 import sys
3216
3217 from io import StringIO, BufferedWriter
3218
3219 LoggerWarningLevel = logging.DEBUG
3220 # LoggerWarningLevel = logging.INFO
3221 # LoggerWarningLevel = logging.WARNING
3222 # LoggerWarningLevel = logging.ERROR
3223
3224 # PositionController class is used to calculate the PID value base on the input
3225 class PositionController:
3226
3227 # Initial class variables.
3228 def __init__(self, kp=0.5, ki=0.08, kd=0.06):
3229
3230 # setting the PID controller
3231 self.PositionControllerNorth = PID.PID(kp, ki, kd)
3232 self.PositionControllerEast = PID.PID(kp, ki, kd)
3233 self.PositionControllerDown = PID.PID(0.5, 0, 0.2)
3234
3235 self.logger = logging.getLogger('PID_data')
3236 self.fileHandler_message = logging.StreamHandler(BufferedWriter(FileIO("PID_data" + \
3237 time.strftime("%Y%m%d-%H%M%S") + ".log", "w"))))

```

```

3238 self.logger.addHandler(self.fileHandler_message)
3239 self.formatter = logging.Formatter('%(message)s')
3240 self.fileHandler_message.setFormatter(self.formatter)
3241 self.logger.setLevel(LoggerWarningLevel)
3242 self.logger.debug('Time,NorthDistance,EastDistance,DownDistance,NorthVelocity,EastVelocity,DownVelocity,\
3243 NorthPTerm,NorthITerm, NorthDTerm,EastPTerm, EastITerm, EastDTerm,DownPTerm, DownITerm, DownDTerm')
3244
3245 """
3246 Function: UpdateDistance2Target3d
3247 Input: northDistance, eastDistance, downDistance
3248 Output: northVelocity, eastVelocity, downVelocity
3249
3250 UpdateDistance2Target3d is used to calacuate the PID values.
3251 """
3252 def UpdateDistance2Target3d(self, northDistance, eastDistance, downDistance):
3253
3254 northVelocity, NorthPTerm, NorthITerm, NorthDTerm = self.PositionControllerNorth.update(northDistance)
3255 eastVelocity, EastPTerm, EastITerm, EastDTerm = self.PositionControllerEast.update(eastDistance)
3256 downVelocity, DownPTerm, DownITerm, DownDTerm = self.PositionControllerDown.update(downDistance)
3257
3258 self.logger.debug('%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f' %\
3259 (
3260 time.time(),
3261 northDistance,
3262 eastDistance,
3263 downDistance,
3264 northVelocity,
3265 eastVelocity,
3266 downVelocity,
3267 NorthPTerm,
3268 NorthITerm,
3269 NorthDTerm,
3270 EastPTerm,
3271 EastITerm,
3272 EastDTerm,
3273 DownPTerm,
3274 DownITerm,
3275 DownDTerm
3276))
3277
3278 return northVelocity, eastVelocity, downVelocity
3279
3280 """
3281 Function: Close
3282 Close the the logger
3283 """
3284 def Close(self):
3285
3286 logging.shutdown()
3287
3288 """
3289 Function: ExitHandler
3290 ExitHandler is used to close the the logger
3291 """
3292 def ExitHandler(self):
3293 self.Close()
3294
3295 def test():
3296
3297 pos_ctrl = PositionController()
3298
3299 for north, east, down in zip(range(10, 0, -1), range(10, 0, -1), range(10, 0, -1)):
3300
3301 velNorth, velEast, velDown = pos_ctrl.UpdateDistance2Target3d(north, east, down)
3302
3303 print velNorth, velEast, velDown
3304
3305 time.sleep(1)
3306
3307 pos_ctrl.Close()
3308
3309 if __name__ == '__main__':
3310
3311 #main()
3312 test()
3313
3314
3315 # *****
3316 # Solo.py
3317
3318 #!/usr/bin/env python
3319 # -*- coding: utf-8 -*-
3320

```

```

3321 import time
3322 from dronekit import connect, VehicleMode, LocationLocal
3323 from pymavlink import mavutil
3324 import math
3325 import rospy
3326 from geometry_msgs.msg import PointStamped, PoseStamped
3327 from std_msgs.msg import String
3328
3329
3330 # The Solo class is used to apply the dronekit package and send the command to drone
3331 class Solo:
3332
3333 # Initial the vehicle object
3334 vehicle = None
3335
3336 # If the IsConnected == True, the class will skip the connect command
3337 # Otherwise, it will run the connect command first
3338 IsConnected = False
3339
3340 # initial the class value
3341 def __init__(self):
3342
3343 __msgVehicleGlobalHomePosition = PointStamped()
3344 __msgVehicleGlobalHomePosition.header.stamp = 0
3345
3346 __msgVehicleGlobalHomePosition.point.x = 0
3347 __msgVehicleGlobalHomePosition.point.y = 0
3348 __msgVehicleGlobalHomePosition.point.z = 0
3349
3350
3351 __msgVehicleLocalHomePosition = PointStamped()
3352 __msgVehicleLocalHomePosition.header.stamp = 0
3353 __msgVehicleLocalHomePosition.point.x = 0
3354 __msgVehicleLocalHomePosition.point.y = 0
3355 __msgVehicleLocalHomePosition.point.z = 0
3356
3357
3358
3359
3360
3361 """
3362 Function: IsArmed
3363 Output: return the Boolean value
3364 """
3365 def IsArmed(self):
3366
3367 return Solo.vehicle.armed
3368
3369 """
3370 Getters and Setters
3371 """
3372
3373
3374 def GetAltitude(self):
3375
3376 return Solo.vehicle.location.global_relative_frame.alt
3377
3378
3379 def GetGlobalPosition(self):
3380
3381 # vehicle GPS information x=lat, y=lon, z=alt
3382 msgVehicleGPS = PointStamped()
3383 msgVehicleGPS.header.stamp = rospy.Time.now()
3384 msgVehicleGPS.point.x = Solo.vehicle.location.global_relative_frame.lat
3385 msgVehicleGPS.point.y = Solo.vehicle.location.global_relative_frame.lon
3386 msgVehicleGPS.point.z = Solo.vehicle.location.global_relative_frame.alt
3387 #rospy.loginfo(msgVehicleGPS)
3388 return msgVehicleGPS
3389
3390 def GetLocalPosition(self):
3391 # vehicle velocity information x=North, y=East, z=Down in m, z is negative altitude
3392 msgVehicleLocalLocation = PointStamped()
3393 msgVehicleLocalLocation.header.stamp = rospy.Time.now()
3394 msgVehicleLocalLocation.point.x = Solo.vehicle.location.local_frame.north
3395 msgVehicleLocalLocation.point.y = Solo.vehicle.location.local_frame.east
3396 msgVehicleLocalLocation.point.z = Solo.vehicle.location.local_frame.down
3397 # rospy.loginfo(msgVehicleLocalLocation)
3398 return msgVehicleLocalLocation
3399
3400
3401 def GetAttitude(self):
3402
3403 msgVehicleAttitude = PointStamped()

```

```

3404 msgVehicleAttitude.header.stamp = rospy.Time.now()
3405 msgVehicleAttitude.point.x = Solo.vehicle.attitude.roll
3406 msgVehicleAttitude.point.y = Solo.vehicle.attitude.pitch
3407 msgVehicleAttitude.point.z = Solo.vehicle.attitude.yaw
3408 # rospy.loginfo(msgvehicleAttitude)
3409 return msgVehicleAttitude
3410
3411 def GetVelocity(self):
3412 # vehicle velocity information x=North, y=East, z=Alt in m/s
3413 msgVehicleVolecity = PointStamped()
3414 msgVehicleVolecity.header.stamp = rospy.Time.now()
3415 msgVehicleVolecity.point.x = Solo.vehicle.velocity[0]
3416 msgVehicleVolecity.point.y = Solo.vehicle.velocity[1]
3417 msgVehicleVolecity.point.z = Solo.vehicle.velocity[2]
3418 # rospy.loginfo(msgvehicleVolecity)
3419 return msgVehicleVolecity
3420
3421 def GetGroundSpeed(self):
3422 # vehicle groundspeed information x=groundspeed, y=0, z=0 in m/s
3423 msgVehicleGroundSpeed = PointStamped()
3424 msgVehicleGroundSpeed.header.stamp = rospy.Time.now()
3425 msgVehicleGroundSpeed.point.x = Solo.vehicle.groundspeed
3426 msgVehicleGroundSpeed.point.y = 0
3427 msgVehicleGroundSpeed.point.z = 0
3428 # rospy.loginfo(msgvehicleGroundspeed)
3429 return msgVehicleGroundSpeed
3430
3431 def GetAirSpeed(self):
3432 # vehicle velocity information x=airspeed, y=0, z=0 in m/s
3433 msgVehicleAirSpeed = PointStamped()
3434 msgVehicleAirSpeed.header.stamp = rospy.Time.now()
3435 msgVehicleAirSpeed.point.x = Solo.vehicle.airspeed
3436 msgVehicleAirSpeed.point.y = 0
3437 msgVehicleAirSpeed.point.z = 0
3438 # rospy.loginfo(msgvehicleAirspeed)
3439 return msgVehicleAirSpeed
3440
3441
3442 def SetGroundSpeed(self, speed):
3443
3444 if speed <= 0 or speed > 10:
3445 print("ground speed is out of allowed range")
3446 return
3447
3448 Solo.vehicle.groundspeed = speed
3449
3450
3451 def __SetGlobalHomePosition(self, lat, lon, alt):
3452
3453 self.__msgVehicleGlobalPosition.header.stamp = rospy.Time.now()
3454 self.__msgVehicleGlobalPosition.point.x = lat
3455 self.__msgVehicleGlobalPosition.point.y = lon
3456 self.__msgVehicleGlobalPosition.point.z = alt
3457
3458
3459 def __SetLocalHomePosition(self, north, east, down):
3460
3461 self.__msgVehicleLocalHomePosition.header.stamp = rospy.Time.now()
3462 self.__msgVehicleLocalHomePosition.point.x = north
3463 self.__msgVehicleLocalHomePosition.point.y = east
3464 self.__msgVehicleLocalHomePosition.poivehiclent.z = down
3465
3466
3467 """
3468 Function: Connect
3469 Input: rate, mode
3470 Returns a Vehicle object connected to the address specified by string parameter ip.
3471 Set rate (Data stream refresh rate. The default is 4Hz (4 updates per second).) as 12.
3472 Choose the mode == 'Simulation', for simulation test
3473 Choose the mode == 'Serial', for usb connect
3474 Choose the mode == 'WIFI', for wifi test
3475 """
3476 def Connect(self, rate = 12, mode = 'Simulation'):
3477
3478 if Solo.IsConnected:
3479 print "The drone is already connected."
3480 return
3481
3482
3483 if mode == 'Simulation':
3484 connection_string = '127.0.0.1:14550' # For simulation test
3485 elif mode == 'Serial':
3486 connection_string = '/dev/serial/by-id/usb-3D_Robotics_PX4_FMU_v2.x_0-if00'

```

```

3487 # connection_string = '/dev/ttyACM0'
3488 else:
3489 connection_string = 'udpin:0.0.0.0:14550' # For simulation test
3490
3491 # Connect to the Vehicle
3492 print('Connecting to vehicle on: %s' % connection_string)
3493 Solo.vehicle = connect(connection_string, wait_ready = False, rate = rate)
3494 Solo.vehicle.mode = VehicleMode("GUIDED")
3495
3496 Solo.IsConnected = True
3497
3498
3499
3500
3501 """
3502 Function: Arm
3503 Doing the pre-arm test and finish arm the drone
3504 """
3505 def Arm(self):
3506
3507 print("Basic pre-arm checks")
3508 # Don't try to arm until autopilot is ready
3509 while not Solo.vehicle.is_armable:
3510 print(" Waiting for vehicle to initialize...")
3511 time.sleep(1)
3512
3513 print("Arming motors")
3514
3515 # Copter should arm in GUIDED mode
3516 Solo.vehicle.mode = VehicleMode("GUIDED")
3517
3518 Solo.vehicle.armed = True
3519
3520 # Confirm vehicle armed before attempting to take off
3521 while not Solo.vehicle.armed:
3522 print(" Waiting for arming...")
3523 time.sleep(1)
3524
3525 print("Armed Successfully.")
3526
3527
3528 """
3529 Function: TakeOff
3530 TakeOff the drone to desired altitude
3531 """
3532 def TakeOff(self, aTargetAltitude):
3533
3534 # Confirm vehicle armed before attempting to take off
3535 if not Solo.vehicle.armed:
3536 print("Error: Vehicle is not armed...")
3537 return
3538
3539 print("Taking off!")
3540 Solo.vehicle.simple_takeoff(aTargetAltitude) # Take off to target altitude
3541
3542
3543 def FlyToLocalPositionNed(self, north, east, down):
3544
3545 self.goto_position_target_local_ned(north, east, down)
3546
3547
3548 def SetVelocityNed(self, velocity_x, velocity_y, velocity_z):
3549
3550 self.send_ned_velocity(velocity_x, velocity_y, velocity_z)
3551
3552
3553 def SetYawDeg(self, heading, relative = False):
3554 # direction -1 ccw, 1 cw
3555
3556 if heading <180 and heading > 0:
3557
3558 self.condition_yaw(heading, relative = False, direction = 1)
3559
3560 else:
3561
3562 self.condition_yaw(heading, relative = False, direction = -1)
3563
3564
3565 def SetYawWithDirectionDeg(self, heading, relative = False, direction = 1):
3566 # direction -1 ccw, 1 cw
3567
3568 self.condition_yaw(heading, relative = False, direction = 1)
3569

```

```

3570
3571 def SetGimbalPitchDeg(self, pitch, roll, yaw):
3572 # The solo gimbal can only control the pitch angle, the effective range is from 0 to -90 degrees
3573
3574 Solo.vehicle.gimbal.rotate(pitch, roll, yaw)
3575
3576
3577 def SetRoi(self, targetLocation):
3578 """
3579 Send MAV_CMD_DO_SET_ROI message to point camera gimbal at a
3580 specified region of interest (LocationGlobal).
3581 The vehicle may also turn to face the ROI.
3582
3583 """
3584 self.set_roi(targetLocation)
3585
3586
3587 def goto_position_target_local_ned(self, north, east, down):
3588 """
3589 Send SET_POSITION_TARGET_LOCAL_NED command to request the vehicle fly to a specified
3590 location in the North, East, Down frame.
3591 """
3592 msg = Solo.vehicle.message_factory.set_position_target_local_ned_encode(
3593 0, # time_boot_ms (not used)
3594 0, 0, # target system, target component
3595 mavutil.mavlink.MAV_FRAME_LOCAL_NED, # frame
3596 0b00000111111111000, # type_mask (only positions enabled)
3597 north, east, down,
3598 0, 0, 0, # x, y, z velocity in m/s (not used)
3599 0, 0, 0, # x, y, z acceleration (not supported yet, ignored in GCS_Mavlink)
3600 0, 0) # yaw, yaw_rate (not supported yet, ignored in GCS_Mavlink)
3601 # send command to vehicle
3602 Solo.vehicle.send_mavlink(msg)
3603
3604
3605 def send_ned_velocity(self, velocity_x, velocity_y, velocity_z):
3606 """
3607 Move vehicle in direction based on specified velocity vectors.
3608 """
3609 msg = Solo.vehicle.message_factory.set_position_target_local_ned_encode(
3610 0, # time_boot_ms (not used)
3611 0, 0, # target system, target component
3612 mavutil.mavlink.MAV_FRAME_LOCAL_NED, # frame
3613 0b000001111111000111, # type_mask (only speeds enabled)
3614 0,0,0, # x, y, z positions (not used)
3615 velocity_x, velocity_y, velocity_z, # x, y, z velocity in m/s
3616 0, 0, 0, # x, y, z acceleration (not supported yet, ignored in GCS_Mavlink)
3617 0, 0) # yaw, yaw_rate (not supported yet, ignored in GCS_Mavlink)
3618 Solo.vehicle.send_mavlink(msg)
3619
3620
3621 def set_roi(self, location):
3622 # create the MAV_CMD_DO_SET_ROI command
3623 msg = Solo.vehicle.message_factory.command_long_encode(
3624 0, 0, # target system, target component
3625 mavutil.mavlink.MAV_CMD_DO_SET_ROI, #command
3626 0, #confirmation
3627 0, 0, 0, 0, #params 1-4
3628 location.lat,
3629 location.lon,
3630 location.alt
3631)
3632 # send command to vehicle
3633 Solo.vehicle.send_mavlink(msg)
3634
3635
3636 def condition_yaw(self, heading, relative = False, direction = 1):
3637 if relative:
3638 is_relative=1 #yaw relative to direction of travel
3639 else:
3640 is_relative=0 #yaw is an absolute angle
3641 # create the CONDITION_YAW command using command_long_encode()
3642 msg = Solo.vehicle.message_factory.command_long_encode(
3643 0, 0, # target system, target component
3644 mavutil.mavlink.MAV_CMD_CONDITION_YAW, #command
3645 0, #confirmation
3646 heading, # param 1, yaw in degrees
3647 0, # param 2, yaw speed deg/s
3648 direction, # param 3, direction -1 ccw, 1 cw
3649 is_relative, # param 4, relative offset 1, absolute angle 0
3650 0, 0, 0) # param 5 ~ 7 not used
3651 # send command to vehicle
3652 Solo.vehicle.send_mavlink(msg)

```

```

3653
3654 # *****
3655 # TargetTracking.py
3656
3657 #!/usr/bin/env python
3658 # -*- coding: utf-8 -*-
3659
3660 import time
3661 import rospy
3662 import math
3663 import logging
3664 import sys
3665
3666 from io import StringIO, BufferedWriter
3667
3668 # from pymavlink import mavutil
3669 import numpy as np
3670 import matplotlib.pyplot as plt
3671 # from numpy import dot
3672 from geometry_msgs.msg import PointStamped, PoseStamped
3673
3674 LoggingLevel = logging.DEBUG
3675 #LoggingLevel = logging.INFO
3676 #LoggingLevel = logging.WARNING
3677 #LoggingLevel = logging.ERROR
3678
3679 KalmanFilterLogFileName = 'KalmanFilterLog.log'
3680 KalmanFilterDataFileName = 'KalmanFilterData.txt'
3681
3682
3683 # IsKalmanFilterEnabled determines if we want to use Kalman Filter
3684 # False: do not use Kalman Filter
3685 # True: use Kalman Filter
3686 IsKalmanFilterEnabled = True
3687
3688 # IsVelocityEstimateEnabled determines if we want to use velocity as a state
3689 # False: do not use velocity
3690 # True: use velocity, and the state space will be totally different from the one without velocity
3691 IsVelocityEstimateEnabled = False
3692
3693 # The skipped number of points are determined by 'NumRepeatingGpsDataToIgnore'
3694 # If the measurement data keep being the same, we will ignore the same data points
3695 # the max ignore points of which is value of NumRepeatingGpsDataToIgnore
3696 # Here we set the number equal 100, which means we want to ignore all the same data when the data keep being the same.
3697 NumRepeatingGpsDataToIgnore = 100
3698
3699 # R matrix, measurement noise covariance matrix
3700 VarianceTruck = 0.01
3701 VarianceTruckVelocity = 0.002
3702 VarianceOpencv = 0.0025
3703 VariancePosition = 0.001
3704
3705 # Q matrix, system uncertainty covariance matrix
3706 sigma_ax = 1.3
3707 sigma_ay = 1.3
3708
3709
3710 """
3711 TargetTracking class implements kalman filter, and fuse truck location data, opencv data and truck velocity data.
3712 We use opencv to calculate the relative distance between marker and camera.
3713 """
3714 class TargetTracking:
3715
3716
3717 def __init__(self):
3718
3719 """
3720 ROS related parameters
3721 """
3722 self.DistanceKFPub = rospy.Publisher("/TargetTracking/TargetLocation", PoseStamped, queue_size=10)
3723
3724 # Initialize truck location data
3725 self.TruckPrevGpsX = -1;
3726 self.TruckPrevGpsY = -1;
3727
3728 self.NumRepeatingGpsData = 0;
3729
3730 # If true, it means we found the target.
3731 self.IsOpencvValid = False
3732
3733 # Save the previous time stamp so we can calculate dt.
3734 # initialize previous time data
3735 self.PreviousMeasurementTime = -1 # -1 indicates the variable has not been used.

```

```

3736
3737
3738 """
3739 Control variables
3740 """
3741 # We implement a lock for threads of callbacks.
3742 self.isCallbackLocked = False
3743
3744
3745 # Initialize predicted data and estimated data.
3746 # x_hat means estimated data, x_pred means predicted data.
3747 if IsVelocityEstimateEnabled:
3748 self.x_pred = np.matrix([[0],
3749 [0],
3750 [0],
3751 [0]])
3752 else:
3753 self.x_pred = np.matrix([[0],
3754 [0]])
3755
3756
3757 # Initialize 'self.IsFirstMeasurementProcessed'
3758 # We use this to demand that the first data we want to process is barometer data
3759 self.IsFirstMeasurementProcessed = False
3760
3761 self.logger = logging.getLogger('KalmanFilterLog')
3762 formatter = logging.Formatter('%(asctime)s %(levelname)s %(message)s')
3763
3764 fileHandler = logging.StreamHandler(BufferedWriter(FileIO("KalmanFilterLogFileName" + \
3765 time.strftime("%Y%m%d-%H%M%S") + ".log", "w"))))
3766 fileHandler.setFormatter(formatter)
3767 self.logger.addHandler(fileHandler)
3768
3769 # streamHandler = logging.StreamHandler(sys.stdout)
3770 # streamHandler.setFormatter(formatter)
3771 # self.logger.addHandler(streamHandler)
3772
3773 self.logger.setLevel(LoggingLevel)
3774
3775
3776 # self.data_logger = logging.getLogger('KalmanFilterData')
3777 # formatter = logging.Formatter('%(asctime)s %(levelname)s %(message)s')
3778
3779 # self.fileHandler = logging.StreamHandler(BufferedWriter(FileIO(KalmanFilterDataFileName, "w")))
3780 # self.fileHandler.setFormatter(formatter)
3781 # self.logger.addHandler(fileHandler)
3782
3783 # self.logger.setLevel(logging.NOTSET) # we really want to have it logged.
3784
3785
3786 """
3787 Initialize the model(A matrix) and noise(P matrix)/uncertainty(Q matrix).
3788 """
3789 def StateSpace(self, data):
3790
3791
3792 if IsVelocityEstimateEnabled:
3793 self.Q = np.matrix([[0.00025, 0, 0.0005, 0],
3794 [0, 0.00025, 0, 0.0005],
3795 [0.0005, 0, 0.0005, 0],
3796 [0, 0.0005, 0, 0.0005]])
3797
3798 self.P = np.identity(4)
3799 self.A = np.matrix([[1, 0, self.dt, 0],
3800 [0, 1, self.dt, 0],
3801 [0, 0, 1, 0],
3802 [0, 0, 0, 1]])
3803
3804 else:
3805 self.Q = np.matrix([[VariancePosition, 0],
3806 [0, VariancePosition]])
3807 self.P = np.identity(2)
3808 self.A = np.matrix([[1, 0],
3809 [0, 1]])
3810
3811 self.B = 0
3812 self.u = 0
3813
3814
3815 '''
3816 Update predicted data and estimated data
3817 '''
3818 def Update(self, obs):

```

```

3819
3820 x_pred = self.A * self.x_hat + self.B * self.u
3821 self.P = self.A * self.P * self.A.T + self.Q
3822 S = self.C * self.P * self.C.T + self.R
3823 K = self.P * self.C.T * np.linalg.inv(S)
3824 y = obs - self.C * x_pred;
3825 self.x_hat = x_pred + K * y;
3826 self.P = self.P - K * self.C * self.P
3827
3828 self.x_pred = self.A * self.x_hat + self.B * self.u
3829
3830
3831 """
3832 CallbackTruckLocation function is to receive the truck location message
3833 from ROS network and be processed by Kalman Filter.
3834 """
3835 def CallbackTruckLocation(self, data):
3836
3837 # We would like to let truck location data as the first message we got
3838 # This part only for first time measurement.
3839 self.logger.debug("CallbackTruckLocation: measurement time: %d.%d." \
3840 %(data.header.stamp.secs, data.header.stamp.nsecs))
3841 if not self.IsFirstMeasurementProcessed:
3842
3843 # if self.isCallbackLocked is ture, that means one other callback function is processing.
3844 # Then we can not process this callback function
3845 # because we only process one callback function at a time.
3846 if self.isCallbackLocked:
3847 self.logger.debug("CallbackTruckLocation:\
3848 First measurement has not processed, but callback is locked.")
3849 return
3850
3851 # After receive the first truck location data, we will never go back in this part 'if' loop
3852 self.IsFirstMeasurementProcessed = True
3853
3854 # To guarantee we only process one callback function everytime
3855 # just in case we receive 2 data at the same time.
3856 self.isCallbackLocked = True
3857
3858 # Critical section
3859 self.PreviousMeasurementTime = data.header.stamp.secs + data.header.stamp.nsecs/1000000000.0
3860 truck_north = data.point.x
3861 truck_east = data.point.y
3862
3863 # After received all the data, we can open the callback lock
3864 # then data can be processed in other callback function.
3865 self.isCallbackLocked = False
3866
3867 self.dt = 0.05
3868
3869 truck_First_north = truck_north
3870 truck_First_east = truck_east
3871
3872 truck_velocity_north = 0
3873 truck_velocity_east = 0
3874
3875 if IsVelocityEstimateEnabled:
3876 self.x_hat = [[truck_First_north],[truck_First_east],[truck_velocity_north],[truck_velocity_east]]
3877 else:
3878 self.x_hat = [[truck_First_north],[truck_First_east]]
3879
3880 self.x_pred = self.x_hat
3881
3882 self.logger.debug("Truck GPS: %f %f, estimated: %f" \
3883 %(truck_First_north, truck_First_east, self.x_hat[0], self.x_hat[1]))
3884
3885 return
3886
3887 # if self.isCallbackLocked is ture, that means one other callback function is processing.
3888 # Then we can not process this callback function because we only process one callback function at a time.
3889 if self.isCallbackLocked:
3890 self.logger.debug("CallbackTruckLocation: Callback is locked.")
3891 return
3892
3893 # To guarantee we only process one callback function everytime
3894 # just in case we receive 2 data at the same time.
3895 self.isCallbackLocked = True
3896
3897 # Critical section
3898 measurement_time = data.header.stamp.secs + data.header.stamp.nsecs/1000000000.0
3899
3900 self.dt = measurement_time - self.PreviousMeasurementTime
3901

```

```

3902 # if self.dt < 0, it is out of time order. We want to ignore those measurements at that point.
3903 if self.dt < 0:
3904 self.isCallbackLocked = False
3905 self.logger.info("CallbackTruckLocation: dt is less than 0.")
3906 return
3907
3908 self.PreviousMeasurementTime = measurement_time
3909
3910 truck_north = data.point.x
3911 truck_east = data.point.y
3912
3913 # deal with repeating data
3914 # if we got some repeating data, ignore some based on NumRepeatingGpsData
3915 # the reason we ignore repeating data is that most of time, the repeating data is invalid for the unsensitive of senses.
3916 # Measurements should change a little bit everytime.
3917 if NumRepeatingGpsDataToIgnore > 0:
3918
3919 if truck_north == self.TruckPrevGpsX and truck_east == self.TruckPrevGpsY:
3920 self.NumRepeatingGpsData = self.NumRepeatingGpsData + 1
3921 self.logger.debug("..... Repeating GPS data, %d" %self.NumRepeatingGpsData)
3922 if self.NumRepeatingGpsData <= NumRepeatingGpsDataToIgnore:
3923 self.logger.debug("..... Ignore the GPS data")
3924 self.isCallbackLocked = False
3925 return
3926 else:
3927 self.NumRepeatingGpsData = 0
3928 self.logger.debug("..... Process the GPS data")
3929 else:
3930 self.NumRepeatingGpsData = 0
3931 self.TruckPrevGpsX = truck_north
3932 self.TruckPrevGpsY = truck_east
3933 self.logger.debug("..... Process the GPS data")
3934
3935 # After received all the data, we can open the callback lock, then data can be processed in other callback function.
3936 self.isCallbackLocked = False
3937
3938 # Update predicted data and estimated data when tne input data is truck_north and truck_east.
3939 obs = np.matrix([[truck_north],[truck_east]])
3940
3941 if IsVelocityEstimateEnabled:
3942 self.C = np.matrix([[1,0,0,0],
3943 [0,1,0,0]])
3944 else:
3945 self.C = np.identity(2)
3946
3947 self.R = np.identity(2) * VarianceTruck
3948
3949 self.StateSpace(data)
3950 self.Update(obs)
3951
3952 msgDistanceKF = PoseStamped()
3953 msgDistanceKF.header.stamp = data.header.stamp
3954
3955 if IsKalmanFilterEnabled:
3956
3957 if IsVelocityEstimateEnabled:
3958 msgDistanceKF.pose.position.x = self.x_hat[0]
3959 msgDistanceKF.pose.position.y = self.x_hat[1]
3960 msgDistanceKF.pose.position.z = self.x_hat[2]
3961 msgDistanceKF.pose.orientation.x = self.x_hat[3]
3962 msgDistanceKF.pose.orientation.y = self.x_pred[0]
3963 msgDistanceKF.pose.orientation.z = self.x_pred[1]
3964 msgDistanceKF.pose.orientation.w = 0
3965 else:
3966 msgDistanceKF.pose.position.x = self.x_hat[0]
3967 msgDistanceKF.pose.position.y = self.x_hat[1]
3968 msgDistanceKF.pose.position.z = 0
3969 msgDistanceKF.pose.orientation.x = 0
3970 msgDistanceKF.pose.orientation.y = self.x_pred[0]
3971 msgDistanceKF.pose.orientation.z = self.x_pred[1]
3972 msgDistanceKF.pose.orientation.w = 0
3973 self.DistanceKFPub.publish(msgDistanceKF)
3974 # rospy.loginfo(msgDistanceKF)
3975
3976 elif not self.IsOpencvValid:
3977 # distance in KF frame message
3978 msgDistanceKF.pose.position.x = data.point.x
3979 msgDistanceKF.pose.position.y = data.point.y
3980 msgDistanceKF.pose.position.z = 0
3981 msgDistanceKF.pose.orientation.x = 0
3982 msgDistanceKF.pose.orientation.y = 0
3983 msgDistanceKF.pose.orientation.z = 0
3984 msgDistanceKF.pose.orientation.w = 1

```

```

3985 self.DistanceKFPub.publish(msgDistanceKF)
3986 # rospy.loginfo(msgDistanceKF)
3987 # Print debug data
3988
3989 # [Todo] add velocity component
3990 if IsVelocityEstimateEnabled:
3991 self.logger.debug("truck north & east: %f %f, estimated: %f %f %f %f"\
3992 %(truck_north, truck_east, self.x_hat[0], self.x_hat[1], self.x_hat[2], self.x_hat[3]))
3993 else:
3994 self.logger.debug("truck north & east: %f %f, estimated: %f %f"\
3995 %(truck_north, truck_east, self.x_hat[0], self.x_hat[1]))
3996
3997
3998
3999 """
4000 CallbackTargetVelocity is to receive the truck's velocity message from ROS network.
4001 Velocity is used as a part of system kinetics model, to predict the next state in Kalman Filter.
4002 """
4003 def CallbackTargetVelocity(self, data):
4004
4005 self.logger.debug("CallbackTargetVelocity: measurement time: %d.%d." \
4006 %(data.header.stamp.secs, data.header.stamp.nsecs))
4007
4008 # If the first data we receive is velocity data, then ignore it.
4009 if not self.IsFirstMeasurementProcessed:
4010 return
4011
4012 # If we do not use velocity as a state, then we can ignore the whole velocity part.
4013 if not IsVelocityEstimateEnabled:
4014 return
4015
4016 # If anyother callback function is receiving data, then ignore this velocity data.
4017 if self.isCallbackLocked:
4018 self.logger.debug("CallbackTargetVelocity: Callback is locked.")
4019 return
4020
4021 # To guarantee we only process one callback function everytime
4022 # just in case we receive 2 data at the same time.
4023 self.isCallbackLocked = True
4024
4025 # Critical section
4026 measurement_time = data.header.stamp.secs + data.header.stamp.nsecs/1000000000.0
4027 self.dt = measurement_time - self.PreviousMeasurementTime
4028
4029 # if self.dt < 0, it is out of time order. We want to ignore those measurements at that point.
4030 if self.dt < 0:
4031 self.isCallbackLocked = False
4032 self.logger.info("CallbackTargetVelocity: dt is less than 0.")
4033 return
4034 else:
4035 self.PreviousMeasurementTime = measurement_time
4036
4037 truck_velocity_north = data.point.x
4038 truck_velocity_east = data.point.y
4039
4040 # After received all the data, we can open the callback lock
4041 # then data can be processed in other callback function.
4042 self.isCallbackLocked = False
4043
4044 # Update predicted data and estimated data
4045 # when the input data is truck_velocity_north and truck_velocity_east.
4046 obs = np.matrix([[truck_velocity_north],[truck_velocity_east]])
4047
4048 self.C = np.matrix([[0,0,1,0],
4049 [0,0,0,1]])
4050
4051
4052 self.R = np.identity(2) * VarianceTruckVelocity
4053
4054 self.StateSpace(data)
4055 self.Update(obs)
4056
4057 msgDistanceKF = PoseStamped()
4058 msgDistanceKF.header.stamp = data.header.stamp
4059
4060 if IsKalmanFilterEnabled:
4061
4062 msgDistanceKF.pose.position.x = self.x_hat[0]
4063 msgDistanceKF.pose.position.y = self.x_hat[1]
4064 msgDistanceKF.pose.position.z = self.x_hat[2]
4065 msgDistanceKF.pose.orientation.x = self.x_hat[3]
4066 msgDistanceKF.pose.orientation.y = self.x_pred[0]
4067 msgDistanceKF.pose.orientation.z = self.x_pred[1]

```

```

4068 msgDistanceKF.pose.orientation.w = 0
4069 else:
4070 msgDistanceKF.pose.position.x = 0
4071 msgDistanceKF.pose.position.y = 0
4072 msgDistanceKF.pose.position.z = data.point.x
4073 msgDistanceKF.pose.orientation.x = data.point.y
4074 msgDistanceKF.pose.orientation.y = 0
4075 msgDistanceKF.pose.orientation.z = 0
4076 msgDistanceKF.pose.orientation.w = 1
4077
4078 self.DistanceKFPub.publish(msgDistanceKF)
4079 # rospy.loginfo(msgDistanceKF)
4080
4081 self.logger.debug("truck_velocity_north & east: %f %f, estimated: %f %f"\
4082 %(truck_velocity_north, truck_velocity_east, self.x_hat[2], self.x_hat[3]))
4083
4084
4085
4086 """
4087 CallbackOpencvMarkerLocation function is to receive the marker location message from ROS network
4088 and be processed by Kalman Filter.
4089 """
4090 def CallbackOpencvMarkerLocation(self, data):
4091
4092 self.logger.debug("CallbackOpencvMarkerLocation: measurement time: %d.%d." \
4093 %(data.header.stamp.secs, data.header.stamp.nsecs))
4094
4095 # If the first data we receive is Opencb marker location data, then ignore it.
4096 if not self.IsFirstMeasurementProcessed:
4097 return
4098
4099 # If anyother callback function is receiving data, then ignore this Opencv marker location data.
4100 if self.isCallbackLocked:
4101 self.logger.debug("CallbackOpencv: First measurement has not processed, but callback is locked.")
4102 return
4103
4104 # To guarantee we only process one callback function everytime
4105 # just in case we receive 2 data at the same time.
4106 self.isCallbackLocked = True
4107
4108 # Critical section
4109 measurement_time = data.header.stamp.secs + data.header.stamp.nsecs/1000000000.0
4110 self.dt = measurement_time - self.PreviousMeasurementTime
4111
4112 # if self.dt < 0, it is out of time order. We want to ignore those measurements at that point.
4113 if self.dt < 0:
4114 self.isCallbackLocked = False
4115 self.logger.info("CallbackOpencv: dt is less than 0.")
4116 return
4117 else:
4118 self.PreviousMeasurementTime = measurement_time
4119
4120 # Just to record if target lost: -1=lost, 1=found
4121 if float(data.header.frame_id) == -1: # target lost
4122 self.IsOpencvValid = False
4123 self.isCallbackLocked = False
4124 return
4125 else:
4126 self.IsOpencvValid = True
4127
4128 distance_north = data.point.x
4129 distance_east = data.point.y
4130
4131 # After received all the data, we can open the callback lock
4132 # then data can be processed in other callback function.
4133 self.isCallbackLocked = False
4134
4135 # Update predicted data and estimated data when tne input data is distance_north and distance_east data.
4136 obs = np.matrix([[distance_north],[distance_east]])
4137
4138 if IsVelocityEstimateEnabled:
4139 self.C = np.matrix([[1,0,0,0],
4140 [0,1,0,0]])
4141 else:
4142 self.C = np.identity(2)
4143
4144 self.R = np.identity(2) * VarianceOpencv
4145
4146 self.StateSpace(data)
4147 self.Update(obs)
4148
4149
4150 msgDistanceKF = PoseStamped()

```

```

4151 msgDistanceKF.header.stamp = data.header.stamp
4152
4153 if IsKalmanFilterEnabled:
4154
4155 if IsVelocityEstimateEnabled:
4156 msgDistanceKF.pose.position.x = self.x_hat[0]
4157 msgDistanceKF.pose.position.y = self.x_hat[1]
4158 msgDistanceKF.pose.position.z = self.x_hat[2]
4159 msgDistanceKF.pose.orientation.x = self.x_hat[3]
4160 msgDistanceKF.pose.orientation.y = self.x_pred[0]
4161 msgDistanceKF.pose.orientation.z = self.x_pred[1]
4162 msgDistanceKF.pose.orientation.w = 0
4163 else:
4164 msgDistanceKF.pose.position.x = self.x_hat[0]
4165 msgDistanceKF.pose.position.y = self.x_hat[1]
4166 msgDistanceKF.pose.position.z = 0
4167 msgDistanceKF.pose.orientation.x = 0
4168 msgDistanceKF.pose.orientation.y = self.x_pred[0]
4169 msgDistanceKF.pose.orientation.z = self.x_pred[1]
4170 msgDistanceKF.pose.orientation.w = 0
4171 else:
4172 msgDistanceKF.pose.position.x = data.point.x
4173 msgDistanceKF.pose.position.y = data.point.y
4174 msgDistanceKF.pose.position.z = 0
4175 msgDistanceKF.pose.orientation.x = 0
4176 msgDistanceKF.pose.orientation.y = 0
4177 msgDistanceKF.pose.orientation.z = 0
4178 msgDistanceKF.pose.orientation.w = 1
4179
4180 self.DistanceKFPub.publish(msgDistanceKF)
4181 # rospy.loginfo(msgDistanceKF)
4182
4183 if IsVelocityEstimateEnabled:
4184 self.logger.debug("opencv_north, opencv_east: %f %f, estimated: %f %f %f %f"\
4185 %(distance_north, distance_east, self.x_hat[0], self.x_hat[1], self.x_hat[2], self.x_hat[3]))
4186 else:
4187 self.logger.debug("opencv_north, opencv_east: %f %f, estimated: %f %f"\
4188 %(distance_north, distance_east, self.x_hat[0], self.x_hat[1]))
4189
4190
4191
4192 def sigint_handler(self):
4193 self.logger.debug("shutdown!")
4194
4195
4196 def main():
4197
4198 tracker = TargetTracking()
4199
4200 rospy.init_node('TargetTracking', anonymous=True)
4201
4202 rospy.Subscriber("/truck/local_location", PointStamped, tracker.CallbackTruckLocation)
4203
4204 #rospy.Subscriber("/truck/location_GPS", PointStamped, kalman.CallbackTargetGps)
4205
4206 rospy.Subscriber("/planner/truck_velocity", PointStamped, tracker.CallbackTargetVelocity)
4207 rospy.Subscriber("/opencv/marker_location", PointStamped, tracker.CallbackOpencvMarkerLocation)
4208
4209 rospy.on_shutdown(tracker.sigint_handler)
4210 rospy.spin()
4211
4212
4213 if __name__ == '__main__':
4214
4215 main()
4216
4217 # *****
4218 # truck.py
4219
4220 #!/usr/bin/env python
4221 # -*- coding: utf-8 -*-
4222
4223 import time
4224 import math
4225
4226 from dronekit import connect, VehicleMode, LocationLocal, LocationGlobalRelative
4227 from pymavlink import mavutil
4228
4229 import rospy
4230 from std_msgs.msg import String
4231 from geometry_msgs.msg import PointStamped, PoseStamped
4232 from Solo import *
4233 import logging

```

```

4234 import sys
4235 from io import StringIO, BufferedWriter
4236
4237
4238 # The Truck class is used to set the target and publish all target data
4239 # Truck class also can be used in the simulation to simulate the moving target
4240
4241 class Truck:
4242
4243 LAT_PER_NORTH = 0.0000089354
4244 LON_PER_EAST = 0.0000121249
4245 LoggerWarningLevel = logging.DEBUG
4246 LoggerWarningLevel = logging.INFO
4247 LoggerWarningLevel = logging.WARNING
4248 LoggerWarningLevel = logging.ERROR
4249
4250 # Initial class variables, publishers, logger.
4251 def __init__(self, bIsSimulationMode = False):
4252 # initial the class
4253 self.truck = Solo()
4254 # self.ImageObj = OpencvLib()
4255
4256 self.truckGpsPub = rospy.Publisher("/truck/location_GPS", PointStamped, queue_size=10)
4257 self.truckAttitudePub = rospy.Publisher("/truck/attitude", PointStamped, queue_size=10)
4258 self.truckVelocityPub = rospy.Publisher("/truck/velocity", PointStamped, queue_size=10)
4259 self.truckGroundSpeedPub = rospy.Publisher("/truck/ground_speed", PointStamped, queue_size=10)
4260
4261 self.isSimulationMode = bIsSimulationMode
4262
4263 if not bIsSimulationMode:
4264 self.truck.Connect(rate = 12, mode = 'Serial')
4265
4266
4267 self.timeStep = 0.1
4268
4269 self.logger = logging.getLogger('Truck_message')
4270 self.fileHandler_message = logging.StreamHandler(BufferedWriter(StringIO("truck_message" + \
4271 time.strftime("%Y%m%d-%H%M%S") + ".log", "w"))))
4272 self.streamHandler_message = logging.StreamHandler(sys.stdout)
4273 self.logger.addHandler(self.fileHandler_message)
4274 self.logger.addHandler(self.streamHandler_message)
4275 # record the data in log file
4276 self.formatter_message = logging.Formatter('%(asctime)s %(levelname)s %(message)s')
4277 # only display the data message in screen
4278 self.formattter = logging.Formatter('%(message)s')
4279
4280 self.fileHandler_message.setFormatter(self.formatter_message)
4281 self.streamHandler_message.setFormatter(self.formattter)
4282 self.logger.setLevel(self.LoggerWarningLevel)
4283
4284 """
4285 Function: SetSimulationPath
4286 Input : path_name, simLatitudeStart, simLongitudeStart, speed_north = 2, speed_east = 2
4287
4288 SetSimulationPath is used to set all simulation data.
4289 """
4290 def SetSimulationPath(self, path_name, simLatitudeStart, simLongitudeStart, speed_north = 2, speed_east = 2):
4291
4292 # simPathName is for the future use.
4293 self.simPathName = path_name
4294
4295 self.simGlobalLocation = PointStamped()
4296 self.simGlobalLocation.point.x = simLatitudeStart # latitude
4297 self.simGlobalLocation.point.y = simLongitudeStart # longitude
4298
4299 self.simVehicleAttitude = PointStamped()
4300 self.simVehicleAttitude.point.x = 0 # roll
4301 self.simVehicleAttitude.point.y = 0 # pitch
4302 self.simVehicleAttitude.point.z = 0 # yaw
4303
4304 self.simVehicleVelocity = PointStamped()
4305 self.simVehicleVelocity.point.x = speed_north # north speed
4306 self.simVehicleVelocity.point.y = speed_east # east speed
4307 self.simVehicleVelocity.point.z = 0 # down speed
4308
4309 speed = math.sqrt((speed_north*speed_north)+(speed_east*speed_east))
4310 self.simVehicleGroundSpeed = PointStamped()
4311 self.simVehicleGroundSpeed.point.x = speed
4312
4313 self.logger.debug("Truck simulation has started.")
4314
4315 """
4316 Function: PublishVehicleMsg

```

```

4317 PublishVehicleMsg is used to publish all truck data.
4318 """
4319 def PublishVehicleMsg(self):
4320
4321 msgVehicleGps = self.truck.GetGlobalPosition()
4322 msgVehicleAttitude = self.truck.GetAttitude()
4323 msgVehicleVelocity = self.truck.GetVelocity()
4324 msgVehicleGroundSpeed = self.truck.GetGroundSpeed()
4325
4326 self.truckGpsPub.publish(msgVehicleGps)
4327 self.truckAttitudePub.publish(msgVehicleAttitude)
4328 self.truckVelocityPub.publish(msgVehicleVelocity)
4329 self.truckGroundSpeedPub.publish(msgVehicleGroundSpeed)
4330
4331 self.logger.debug("send the truck data")
4332
4333 """
4334 Function: PublishSimulationMsg
4335 PublishSimulationMsg is used to publish all simulation data.
4336 """
4337 def PublishSimulationMsg(self):
4338
4339 self.simGlobalLocation.header.stamp = rospy.Time.now()
4340 self.simVehicleAttitude.header.stamp = rospy.Time.now()
4341 self.simVehicleVelocity.header.stamp = rospy.Time.now()
4342 self.simVehicleGroundSpeed.header.stamp = rospy.Time.now()
4343
4344 self.truckGpsPub.publish(self.simGlobalLocation)
4345 self.truckAttitudePub.publish(self.simVehicleAttitude)
4346 self.truckVelocityPub.publish(self.simVehicleVelocity)
4347 self.truckGroundSpeedPub.publish(self.simVehicleGroundSpeed)
4348
4349 self.simGlobalLocation.point.x += (self.timeStep * self.simVehicleVelocity.point.x) * Truck.LAT_PER_NORTH
4350 self.simGlobalLocation.point.y += (self.timeStep * self.simVehicleVelocity.point.y) * Truck.LON_PER_EAST
4351 self.logger.debug("Simulation: send the truck data")
4352
4353 """
4354 Function: TimerCallback
4355 Ros Timer Callback function, every 0.1s running the TimerCallback once.
4356 If isSimulationMode == True, it will publish the simulation data.
4357 If isSimulationMode == False, it will publish the real data.
4358 """
4359 def TimerCallback(self, event):
4360
4361 if self.isSimulationMode:
4362 self.PublishSimulationMsg()
4363 else:
4364 self.PublishVehicleMsg()
4365
4366 """
4367 Function: sigint_handler
4368 Close the logger
4369 """
4370 def sigint_handler(self):
4371 self.logger.debug("shutdown!")
4372 logging.shutdown()
4373
4374 """
4375 Function: main
4376 Intital the Truck class and ros nod
4377 Rununing all code at 10Hz.
4378 """
4379 def main():
4380
4381 # Enable the isSimulationMode
4382 # If isSimulationMode == True, it will publish the simulation data.
4383 # If isSimulationMode == False, it will publish the real data.
4384 isSimulationMode = False
4385
4386 truck = Truck(isSimulationMode)
4387
4388 # # GPS location of Soccer field.
4389 # simLatitudeStart = 42.3147439
4390 # simLongitudeStart = -83.2299671
4391
4392 # # GPS location of Simulation
4393 simLatitudeStart = -35.362975
4394 simLongitudeStart = 149.165441
4395
4396 # SetSimulationPath has no effect if isSimulation is False

```

```

4400 truck.SetSimulationPath("StraightNorth", simLatitudeStart, simLongitudeStart, speed_north = 0, speed_east = 0)
4401
4402 rospy.init_node('truck', anonymous=True)
4403
4404 # Use timer interrupt to make sure State Machine running as 10Hz
4405 dTimeStep = 0.1
4406 rospy.Timer(rospy.Duration(dTimeStep), truck.TimerCallback)
4407 rospy.on_shutdown(truck.sigint_handler)
4408 rospy.spin()
4409
4410
4411 if __name__ == '__main__':
4412 main()
4413
4414 #####
4415 # ultrasonic_sensor code
4416 #####
4417 # I2C_ultrasonic_test.py
4418
4419 #!/usr/bin/env python
4420 # -*- coding: utf-8 -*-
4421
4422 import smbus2
4423 import time
4424 import rospy
4425 import atexit
4426
4427
4428 from geometry_msgs.msg import PointStamped, PoseStamped
4429 import logging
4430 import sys
4431 from io import StringIO, BufferedWriter
4432
4433 LoggerWarningLevel = logging.DEBUG
4434 # LoggerWarningLevel = logging.INFO
4435 # LoggerWarningLevel = logging.WARNING
4436 # LoggerWarningLevel = logging.ERROR
4437
4438 # Ultrasonic class is used for the I2C ultrasonic sensor
4439 # For the TX1, the address in bus 1 is 0x70
4440 class Ultrasonic:
4441
4442 # Initail the bus and I2C address
4443 def __init__(self):
4444 # init ultrasonic
4445 self.bus = smbus2.SMBus(1)
4446 self.address = 0x70
4447 self.Write(224)
4448 """
4449 Publishers setup
4450 """
4451 self.ultrasonicI2CPub = rospy.Publisher("/ultrasonic/long_distance", PointStamped, queue_size=10)
4452 """
4453 Log init variables
4454 """
4455 self.logger = logging.getLogger('Ultrasonic_message')
4456 self.fileHandler_message = logging.StreamHandler(BufferedWriter(FileIO("Ultrasonic_message" + \
4457 time.strftime("%Y%m%d-%H%M%S") + ".log", "w")))
4458 self.logger.addHandler(self.fileHandler_message)
4459 self.formatter_message = logging.Formatter('%(asctime)s %(levelname)s %(message)s')
4460 self.fileHandler_message.setFormatter(self.formatter_message)
4461 self.logger.setLevel(LoggerWarningLevel)
4462
4463 def Write(self, value):
4464 self.bus.write_byte_data(self.address, 0, value)
4465
4466 def GetRange(self):
4467 range = self.bus.read_i2c_block_data(self.address, 0, 2)
4468 final_range = range[0]*256 + range[1]
4469 return final_range
4470
4471 def PublishData(self, range):
4472 range = float(range)/100
4473
4474 msgUltrasonicI2C = PointStamped()
4475 msgUltrasonicI2C.header.stamp = rospy.Time.now()
4476 msgUltrasonicI2C.point.x = range
4477
4478 msgUltrasonicI2C.point.y = 0
4479 msgUltrasonicI2C.point.z = 0
4480 self.ultrasonicI2CPub.publish(msgUltrasonicI2C)
4481
4482 # # Print debug data

```

```

4483 rospy.loginfo(msgUltrasonicI2C)
4484 self.logger.debug('%f,%f\n' % (time.time(), range))
4485
4486
4487 def TimerCallback(self, event):
4488
4489 self.Write(81)
4490 time.sleep(0.1)
4491 rng = self.GetRange()
4492 self.PublishData(rng)
4493
4494 def sigint_handler(self):
4495 self.logger.warning("shutdown!")
4496 logging.shutdown()
4497
4498 def main():
4499 rospy.init_node('Ultrasonic_I2C', anonymous=True)
4500 ult = Ultrasonic()
4501
4502 # set the timer frequency
4503 dTimeStep = 0.1
4504 rospy.Timer(rospy.Duration(dTimeStep), ult.TimerCallback)
4505 rospy.on_shutdown(ult.sigint_handler)
4506 rospy.spin()
4507
4508
4509
4510 if __name__ == '__main__':
4511 main()
4512
4513 #####
4514 # user_control code
4515 #####
4516 # User.py
4517
4518
4519 #!/usr/bin/env python
4520 # * coding: UTF 8 *
4521
4522 import time
4523 import math
4524
4525
4526 class User:
4527
4528
4529 def __init__(self):
4530
4531 self.local_location_north = 0
4532 self.local_location_east = 0
4533 self.local_location_down = 0
4534
4535 # self.attitude_roll = 0
4536 # self.attitude_pitch = 0
4537 self.attitude_yaw = 0
4538 self.start = 0
4539 self.stop = 0
4540 self.return_to_home = 0
4541
4542 # *****
4543 # user_command_node.py
4544
4545 #!/usr/bin/env python
4546 # * coding: UTF 8 *
4547
4548 import time
4549 import math
4550
4551 import rospy
4552 from geometry_msgs.msg import PointStamped, PoseStamped
4553
4554 import User
4555 import logging
4556 import sys
4557 from io import StringIO, BufferedWriter
4558
4559
4560 LoggerWarningLevel = logging.DEBUG
4561 # LoggerWarningLevel = logging.INFO
4562 # LoggerWarningLevel = logging.WARNING
4563 # LoggerWarningLevel = logging.ERROR
4564
4565

```

```

4566 logger = logging.getLogger('User_message')
4567 fileHandler_message = logging.StreamHandler(BufferedWriter(FileIO("User_message" + \
4568 time.strftime("%Y%m%d-%H%M%S") + ".log", "w"))))
4569 logger.addHandler(fileHandler_message)
4570 formatter_message = logging.Formatter('%(asctime)s %(levelname)s %(message)s')
4571 fileHandler_message.setFormatter(formatter_message)
4572 logger.setLevel(LoggerWarningLevel)
4573
4574
4575 def sigint_handler():
4576 logger.warning("shutdown!")
4577 logging.shutdown()
4578
4579
4580
4581 def display_user_menu():
4582
4583 print("\n")
4584
4585 print("+-----+")
4586 print("| [s] start flight test |")
4587 print("| [f] stop all flight command |")
4588 print("| [r] return to home |")
4589 print("| [99] Exit |")
4590 print("|Note: return to home enable only after stop all flight command|")
4591 print("+-----+")
4592
4593 def publish_command_data():
4594 # Start to publish data
4595 msgCommandData = PointStamped()
4596 msgCommandData.header.stamp = rospy.Time.now()
4597 msgCommandData.point.x = user.start
4598 msgCommandData.point.y = user.stop
4599 msgCommandData.point.z = user.return_to_home
4600 CommandDataPub.publish(msgCommandData)
4601 logger.info('send command data')
4602
4603 def clear():
4604 user.start = 0
4605 user.stop = 0
4606 user.return_to_home = 0
4607 logger.info('reset command data')
4608
4609
4610 if __name__ == '__main__':
4611
4612 # Initial the node
4613 rospy.init_node('user_node', anonymous=True)
4614
4615 user = User.User()
4616
4617 # setup publisher
4618 CommandDataPub = rospy.Publisher("/user/command_data", PointStamped, queue_size=10)
4619
4620
4621 while not rospy.is_shutdown():
4622
4623 display_user_menu()
4624
4625 inputKey = raw_input("input a key then press enter:")
4626
4627 if inputKey == 's':
4628 user.start = 1
4629 logger.warning('start flying')
4630
4631 elif inputKey == 'f':
4632 user.stop = 1
4633 logger.warning('stop flying')
4634
4635 elif inputKey == 'r':
4636 logger.warning('return to home')
4637 user.return_to_home = 1
4638
4639 elif inputKey == '99':
4640 logger.warning("exiting!")
4641 sigint_handler()
4642 break
4643
4644 else:
4645 logger.warning("there is no such key!")
4646 continue
4647
4648

```

```

4649 publish_command_data()
4650 clear()
4651
4652 # deal with some left-behinds
4653 # After spin() no code will be excuted.
4654 rospy.on_shutdown(sigint_handler)
4655
4656 rospy.spin()
4657
4658 # *****
4659 # user_control_node.py
4660
4661 #!/usr/bin/env python
4662 # * coding: UTF 8 *
4663
4664 import time
4665 import math
4666
4667 import rospy
4668 from geometry_msgs.msg import PointStamped, PoseStamped
4669
4670 import User
4671 import logging
4672 import sys
4673
4674
4675
4676 LoggerWarningLevel = logging.DEBUG
4677 # LoggerWarningLevel = logging.INFO
4678 # LoggerWarningLevel = logging.WARNING
4679 # LoggerWarningLevel = logging.ERROR
4680
4681
4682
4683 logger = logging.getLogger('User_message')
4684 fileHandler_message = logging.StreamHandler(BufferedWriter(FileIO("User_message" + \
4685 time.strftime("%Y%m%d-%H%M%S") + ".log", "w"))))
4686 logger.addHandler(fileHandler_message)
4687 formatter_message = logging.Formatter('%(asctime)s %(levelname)s %(message)s')
4688 fileHandler_message.setFormatter(formatter_message)
4689 logger.setLevel(LoggerWarningLevel)
4690
4691
4692
4693 def sigint_handler():
4694
4695 logger.warning("shutdown time!")
4696 logger.close()
4697
4698
4699 def display_user_menu():
4700
4701 print("\n")
4702 print("+----- < Main menu > -----+")
4703 print("|")
4704 print("|")
4705 print("| ^")
4706 print("| |: north 30cm")
4707 print("| ^")
4708 print("| \ w: up 30cm")
4709 print("| \ ^")
4710 print("| \ |")
4711 print("| \ |")
4712 print("| \ |")
4713 print("| \ |")
4714 print("| \ |")
4715 print("| <--: west 30cm -----> --: east 30cm")
4716 print("| |")
4717 print("| |")
4718 print("| |")
4719 print("| |")
4720 print("| s: down 30cm \")
4721 print("| \")
4722 print("| |: south 30cm")
4723 print("| v")
4724 print("|")
4725 print("|")
4726 print("+-----+")
4727 print("| [w] Go up 30 cm | [i] go north 30 cm |")
4728 print("| [s] Go down 30 cm | [k] go south 30 cm |")
4729 print("| [a] Turn left 30 degree | [j] go west 30 cm |")
4730 print("| [d] Turn right 30 degree| [l] go east 30 cm |")

```

```

4732 print("| [99] Exit | [h] return to home |")
4733 print("| |")
4734 print("+-----+")
4735 # print("input a key then press enter:")
4736 # print("use `rostopic echo` to query commands")
4737 # print("-----")
4738
4739
4740
4741
4742 def listener_callback_local_location(data):
4743 # all in meters
4744 user.local_location_north = data.point.x
4745 user.local_location_east = data.point.y
4746
4747 def listener_callback_ultrasonic(data):
4748 user.local_location_down = data.point.z # in meters
4749
4750 def listener_callback_attitude(data):
4751 user.attitude_yaw = data.point.z # in radians
4752
4753
4754 def publish_desired_drone_pose():
4755 # Start to publish data
4756 msgDesiredPose = PoseStamped()
4757 msgDesiredPose.header.frame_id = "marker_pose_msg"
4758 msgDesiredPose.header.stamp = rospy.Time.now()
4759 msgDesiredPose.pose.position.x = 0
4760 msgDesiredPose.pose.position.y = 0
4761 msgDesiredPose.pose.position.z = 0
4762 msgDesiredPose.pose.orientation.x = user.local_location_north
4763 msgDesiredPose.pose.orientation.y = user.local_location_east
4764 msgDesiredPose.pose.orientation.z = user.local_location_down
4765 msgDesiredPose.pose.orientation.w = user.attitude_yaw
4766 dronePosePub.publish(msgDesiredPose)
4767 # # Print debug data
4768 # rospy.loginfo(msgDesiredPose)
4769
4770
4771
4772
4773 if __name__ == '__main__':
4774
4775 # Initial the node
4776 rospy.init_node('user_node', anonymous=True)
4777 rospy.on_shutdown(sigint_handler)
4778
4779 user = User.User()
4780
4781 # setup publisher
4782 dronePosePub = rospy.Publisher("/user/drone_pose_desired", PoseStamped, queue_size=10)
4783
4784 # setup subscriber
4785 rospy.Subscriber("/solo/location_local", PointStamped, listener_callback_local_location)
4786 rospy.Subscriber("/solo/attitude", PointStamped, listener_callback_attitude)
4787 rospy.Subscriber("/solo/ultrasonic_sensor", PointStamped, listener_callback_ultrasonic)
4788
4789 # # setup timer
4790 # # Use timer interrupt to make sure publish at 20Hz
4791 # dTimeStep = 0.05
4792 # rospy.Timer(rospy.Duration(dTimeStep), timer_callback)
4793
4794 while(1):
4795
4796 display_user_menu()
4797
4798 inputKey = raw_input("input a key then press enter:")
4799 if inputKey == 'w':
4800 user.local_location_down += 0.2
4801 elif inputKey == 's':
4802 user.local_location_down -= 0.2
4803 elif inputKey == 'a':
4804 user.attitude_yaw -= 0.174533
4805 if user.attitude_yaw > 6.28:
4806 user.attitude_yaw = user.attitude_yaw - 6.28
4807 if user.attitude_yaw < 0:
4808 user.attitude_yaw = user.attitude_yaw + 6.28
4809 elif inputKey == 'd':
4810 user.attitude_yaw += 0.174533
4811 if user.attitude_yaw > 6.28:
4812 user.attitude_yaw = user.attitude_yaw - 6.28
4813 if user.attitude_yaw < 0:
4814 user.attitude_yaw = user.attitude_yaw + 6.28

```

```
4815
4816 elif inputKey == 'i':
4817 user.local_location_north += 0.3
4818 elif inputKey == 'k':
4819 user.local_location_north -= 0.3
4820 elif inputKey == 'j':
4821 user.local_location_east -= 0.3
4822 elif inputKey == 'l':
4823 user.local_location_east += 0.3
4824
4825 elif inputKey == '99':
4826 logger.warning("exiting!")
4827 sigint_handler()
4828 break
4829
4830 else:
4831 logger.warning("there is no such key!")
4832 continue
4833
4834
4835 publish_desired_drone_pose()
4836
4837
4838 # deal with some left-behinds
4839 # After spin() no code will be excuted.
4840 rospy.spin()
```