Getting started with Loomo

VITA, EPFL

February 2019

1 Installation

1. Install *Android Studio*: `https://developer.android.com/studio/install`
2. Import project: File > New > Project from Version Control > Git > https://github.com/segway-robotics/loomo-algodev/tree/yuejiang/crowd
3. Add 3rd party librairies (follow *README.md* instructions).

Here is a list of possible issues:

- Missing NDK: File > Project Structure > SDK Location > Android NDK Location > Install
- *CMake* missing error: click on install *CMake* in *Android Studio* 'sync' window (easier than installing *CMake* before building)
- *Python* missing error: install *Python3* and add it to environment path
- *Ndk* toolchains error: if *mips64* is missing, download an archived version of *Ndk* that contains *mips64*, such as android-ndk-r16b

2 Development Kit

2.1 Classes

2.1.1 AlgoBase

This is the base class for the algorithms that should run on *Loomo*. Here are defined the important functions common to most projects such as:

1. *start()*,
2. *run()*,
3. *exit()*.

Most projects will need to create a separate class that is derived from this one and where the specific functionalities will be defined.

2.1.2 Algorithm class

The main algorithm shall be written in a class that inherits from the *AlgoBase* class. Here the parameters unique to the application are initialized and the main loop must be defined in the following function:

- *step()*.

This function is called at every iteration.

The C++ development kit for *Loomo* contains several example applications for:

- displaying sensor information,
- local mapping,
- object recognition.

Each of these applications are defined in their own source and header files where the main loop is defined in the *step()* function from the source file. A call to the *start()* function from the base class (*AlgoBase*) starts the loop and most applications should take inspiration from these examples.

2.1.3 RawData

The *RawData* class is used to retrieve sensor information from the robot and send commands to the actuators. Only one instance of this class must be created where all calls will be done. This instance shall be sent to the algorithm class as a parameter when it is instantiated.

As there is no official documentation for the functions in this class, it is necessary to refer to the *RawData.h* file[1] for information on the functions available and their parameters.

2.2 Launching an application

2.2.1 Connecting to *Loomo*

A computer can connect to the *Loomo* via the *Android Debug Bridge* (adb) command-line tool if they are on the same Wi-Fi network. To do this, it is necessary to obtain the IP address of the robot (Settings > System > Status > IP address). Once connected, it is possible to upload an application to the robot and receive debug information. Below are a few example commands to connect to a device and check for status, where the IP address must be replaced by the *Loomo* IP address.

```
adb connect 128.179.176.227
adb devices
adb shell
```

[1] '*dependency/algobaseinclude/interface/RawData.h*'

```
adb push / pull
adb disconnect 128.179.176.227
```

For more information about adb commands, please visit `https://developer.android.com/studio/command-line/adb`

2.2.2 *Loomo* screen

When running an application from the C++ development kit for *Loomo*, a screen as seen in figure 1 will appear on the *Loomo*. Here, the application running is the *AlgoTest* app that displays sensor information, but when launched, only the buttons on the top are visible and they represent several functions that can be started.

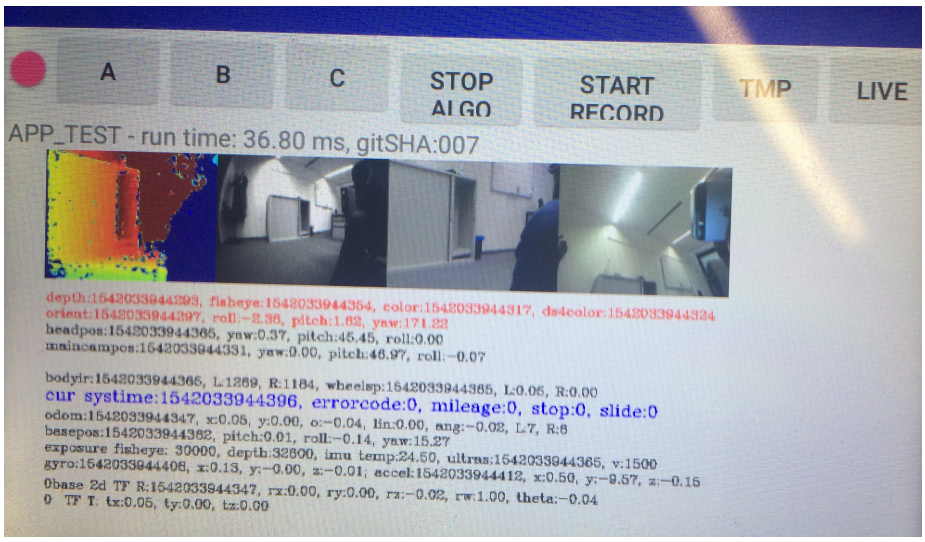

Figure 1: Loomo screen example running *AlgoTest* app from the development kit

2.2.3 Structure

The buttons on the *Loomo* screen are functions defined in '*core/AlgoImpl.cpp*'. To begin launching the application, the user must press the red button at the top left of figure 1 that will also run the *testAlgoStart(...)* function:
START ALGO/STOP ALGO buttons.
This function is the start of the main loop; here, an instance of *RawData* and an instance of the application algorithm class are created and the loop is started on a new thread with a call to the *start()* function.

Other functions can also be called via the *Loomo* screen (*funcA, funcB, funcC*) and can be customized by the user for each application in *AlgoImpl.cpp*.

Also in this file, it is necessary to define the cameras that must be used by the application (*USE_DEPTH*, *USE_FISHEYE*...).

2.2.4 CMakeLists.txt

The build process of the C++ code is managed by *CMake*, hence, the presence of the '*CMakeLists.txt*' file[2] in the project. Each application must be included in this file for compilation, where the macros must be modified to switch between applications. Also, for each application, it is necessary to list the libraries that must be used such as *Tensorflow* and *LocalMapping*.

2.3 External libraries

Third party libraries such as *OpenCV* and *Tensorflow* are included in the project and can be easily used. An example of a *Tensorflow* application that also uses *OpenCV* functions is included in the development kit.

2.4 ROS (Robot Operating System)

With this kit, you can build ROS-like threads. As such, some features from ROS can be found such as ROS request messages. An example of this is the *tf_message*[3] that is based on ROS's '*geometry_msgs::TransformedStamped*' messages. Here a user can send a request message to obtain the position of a robot part in a specific coordinate frame. Messages are received in a ROS format that use quaternions and sometimes must be converted, such as for 2D pose with the *tfmsgTo2DPose()* function. Using these messages is an efficient way to obtain the position of the robot parts such as the base or the head.

2.5 Movement

2.5.1 Robot base

Moving the robot's base is done with a call to:

- *ExecuteCmd(...)*.

This function takes in as parameters the translational velocity, the rotational velocity and the current timestamp. Similarly, stopping the robot is done with the same call but with zero-valued velocities and it is possible to use negative values for movement in the opposite direction.

It can be useful to use the robot odometry or other cues to stop the robot, however, given that the primary objective of the robot is to remain balanced, there will always be a delay while it tilts to adjust balance before stopping or moving.

An example of the use of the function is implemented in '*AlgoTest.cpp*' where the movement is started by pressing the B function button on the robot.

[2] '*algo_app/src/main/jni/CMakeLists.txt*'
[3] defined in '*dependancy/algobaseInclude/interface/tf_data_type.h*'

2.5.2 Robot head

The robot head is able to turn 150° each way for yaw and 180° to -90° for pitch. The head can be controlled in position or in speed for both yaw and pitch using the *ExecuteHeadMode()* function to change between speed and position control, and both *ExecuteHeadSpeed()* and *ExecuteHeadPos()* for the actual movement. With the position control, we are unable to control the head speed and the head moves slowly when close to its desired position due to the controller implemented, which is problematic when turning the base since the head is not looking in the correct direction to detect obstacles. As a result, it is more reliable to control the head by speed, while adding a certain position offset from the base using a PD controller, to look ahead in a good direction for object detection. A simple proportional controller cannot be used as it will accumulate error.

2.6 Socket-based computation offloading

The example application '*socket*' demonstrates how to send information between a client and a server, useful to process computationally heavy algorithms on a device that does not have the hardware limitations of the robot. It is also recommended to build fast and flexible prototypes using the robot-cloud socket.

Here the server (robot) sends the position of people detected, encoded as floats, to a client (cloud computer) that can track people with the information and send back the results to the robot.

On the cloud side[4], it is necessary to verify that the correct IP address of the robot is entered and that the executable is built again.

The example sends floats but there are several functions for sending chars or images as well, however, sending heavy loads over the network such as multiple images might come with a longer delay.

3 Basic walkthrough

3.1 Algo Test

AppTest is a sample application that extracts, displays and tests all basic elements of a robot such as sensors, robot joint positioning, camera inputs, odometry...

3.2 Algo Local Mapping

AlgoLocalMapping is a sample application that builds and maintains an occupancy map with sensory measurements. A local map centered at the robot frame can be queried by calling the function *getDepthMap()* or *getDepthMapWithFrontUltrasonic()*. For more details, refer to *LocalMapping.h* file.

[4] `https://github.com/vita-epfl/socket-loomo`, private link for now. Use *git checkout tags/cpp_standalone* to switch to an easy to run version

A local map can be initialized with a single call to the following example function:

```
bool AppName::initLocalMapping()
{
    if(m_p_local_mapping) {
        delete m_p_local_mapping;
        m_p_local_mapping = NULL;
    }
    float mapsize = 6.0;
    float m_map_resolution = 0.05;
    m_p_local_mapping = new ninebot_algo::local_mapping::
        LocalMapping(mapsize, m_map_resolution);
    if(m_p_local_mapping == NULL) return false;

    CalibrationInfoDS4T calib;
    mRawDataInterface->getCalibrationDS4T(calib);
    float fx = calib.depth.focalLengthX;
    float fy = calib.depth.focalLengthY;
    float px = calib.depth.pricipalPointX;
    float py = calib.depth.pricipalPointY;

    m_p_local_mapping->setLidarRange(5.0);
    m_p_local_mapping->setLidarMapParams(0.6, true);
    m_p_local_mapping->setDepthCameraParams(px, py, fx, fy, 1000);
    m_p_local_mapping->setDepthRange(3.5, 0.35, 0.9, 0.1);
    m_p_local_mapping->setDepthMapParams(1.0, 10, false, -1);
    m_p_local_mapping->setUltrasonicRange(0.5);

    m_map_width =  mapsize / m_map_resolution;
    m_map_height = mapsize / m_map_resolution;

    return true;
}
```

3.3 Algo Tensorflow

AlgoTensorflow presents an example that uses a *Tensorflow* model with all processing done with the on-board CPU. The pre-trained model[5] needs to be downloaded online and pushed to the corresponding folder of the robot.

3.4 Algo Socket

AlgoSocket demonstrates a recommended method for flexible prototyping and computation offloading. Efficient components such as constructing a local map

[5] *inception_v3_2016_08_28_frozen.pb*

and detecting blobs of persons are carried out on the robot, whereas core algorithms for tracking and planning are offloaded to the cloud. The socket latency between a robot and a cloud computer through the campus wifi is typically around 50 ms, fast enough for most real-time experiments.

4 Advanced examples

4.1 Running a multi-person detector

Modern person detectors are almost exclusively deep convolution neural networks as they yield more precise and consistent results than past methods. Existing models can be found online, however, since CNN-based object detectors are capable of classifying multiple objects, it is more common to find object detectors that happen to be capable of detecting people, rather than simple person detectors. As such, the model chosen for this example is capable of detecting 182 objects, including humans. The model, named "*ssd_mobilenet_v1_coco*"[6], is trained on the COCO dataset and is one of the most computationally efficient, while remaining precise enough for the *Loomo*, that only needs to locate people at a close range ($\approx$ 5m). However, this model does not use the depth information available for detection, and it usually is not necessary for person detection, but it can be useful for global positioning of a person with respect to the robot.

Other algorithms[7] using depth information for multi-person detection and tracking exist, but they are complicated to integrate (requiring full installation of ROS) and/or have limitations such as requiring manual user inputs, a specific height of the robot camera for better results or they require extra sensors (2D lasers) not available to *Loomo*. As such, a *Tensorflow* model is used for its simpler implementation, fast computation and precision in the scenario of a robot close to the ground locating people in a small radius.

The multi-person detector code[8] is inspired from the *Tensorflow* sample application, where the model is changed as well as its inputs/outputs and a few operations resulting from the different model. It can detect people at a range of approximately 5 meters from the robot depending on their size and runs in 0.9-1s on the *Loomo* CPU (with 640*480 images). This model is capable of detecting people in many different postures or with isolated body parts, such as having only legs visible, or only arms.

[6] `https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md`

[7] `https://github.com/lagadic/pepper_launch/wiki/People-detection-using-RGBD`

[8] `https://github.com/vita-epfl/loomo-algodev-mirror/tree/paul/elevator/algo_app/src/main/jni/app_tensorflow_person_detector`

4.2 Local mapping and path planning

4.2.1 Local mapping

Local mapping is performed using the *LocalMapping* class[9]. Here the user can create and update a local occupancy map around the robot using the depth camera only (*getDepthMap(...)*) or a combination of depth camera and ultrasonic sensors (*getDepthMapWithFrontUltrasonic(...)*) as performed in the example application. The ultrasonic sensors only help to detect obstacles at a short distance directly in front of the robot. To create a new map, the user can call *clearMap()* to remove all prior knowledge, otherwise the map is updated with the new sensor information in the robot field of view. The map created is typically a 6x6m image due to sensor range and is a probability map of encountering obstacles. Some limitations of the map include the robot's field of view which can require turning the head to obtain information on the sides, or the detection range for close objects which is of 25cm for the ultrasonic sensor, and there is also an inability to detect very small objects on the ground that can obstruct robot movement, such as the tip of feet, because the depth camera and ultrasonic sensors cannot orient themselves towards the ground.

4.2.2 Path planning and movement

In terms of path planning and movement, the *ModelBasedPlanner* class[10] can be used. This class allows the user to give the desired goal location for the robot (*reset_global_reference(...)* function) as a vector of x-y checkpoints and the class will take care of the path planning, movement and obstacle avoidance for the robot. However, it might be necessary to create multiple waypoints along the path as the class and its functions may not always know how to get to a goal far away with no clear path. There are a few issues with this class. For example, the planner does not allow the robot to make sharp turns when close to obstacle, which is sometimes necessary to avoid getting stuck. Furthermore, if the robot is quite close to an obstacle, it stops moving even if there is sufficient space because the planner stops all movement when it is very close to obstacles, which can be over-conservative planning in certain situations.

The functions use map coordinates that change at every new run when using local mapping because they depend on the starting position and orientation, so it is best to convert the robot pose and orientation to the local map coordinates before using the occupancy map and attempting to set a goal location.

A function implemented for movement is *safeControl(...)*, that takes as inputs the linear and rotational velocities for the robot. This function not only makes the robot move, but has also been modified to control the robot head[11] so that it turns to have objects on the path in the field of view of the depth camera, as there are many collisions otherwise. Furthermore, the function contains

[9] 'dependancy/algo_include/general/LocalMapping.h'

[10] 'dependancy/algo_include/general/model_based_planner.h'

[11] Only located in the elevator scenario application: *https://github.com/vita-epfl/loomo-algodev-mirror/tree/paul/elevator/algo_app/src/main/jni/app_elevator*

a part that will prevent the robot from moving as part of obstacle avoidance by using the current robot velocity and ultrasonic sensors to detect obstacles and stop before it is too late.

4.2.3 Map format and information

The local map constructed by the robot is in an array format visible in figure 2.

Figure 2: Local mapping example in an office.

The local map can be modified in size but is a 2D map, typically 6m x 6m and is constructed with the robot at the origin (center of the map). The map represents the probability of encountering obstacles where obstacles in the map appear in gray and the values of each pixel can range from 0 to 255. It is best to loop through all pixels to acquire information, where we can typically consider obstacles to be pixels with a value superior to 20 as seen in the following code:

```
for (int i = 0; i < localMap.rows; ++i) {
    for (int j = 0; j < localMap.cols; ++j) {
        if(localMap.at<uchar>(i,j) > 20){
            // obstacle
        }
        else {
            // not obstacle
        }
    }
}
```

The local map is a crop of a bigger fixed map that is of size 30m x 30m. However, the bigger the map, the more it is falsified by the accumulation of odometry error. The fixed map can be acquired with the *getFixMap(...)* function and an example map is visible in figure 3. In this map, we can observe

walls that become larger and unaligned with respect to the real world due to the odometry error. This map was constructed with a single carefully planned passage, in a straight line, with a 90° turn, but with more chaotic movements, the map can become completely distorted and must be corrected with loop closure algorithms.

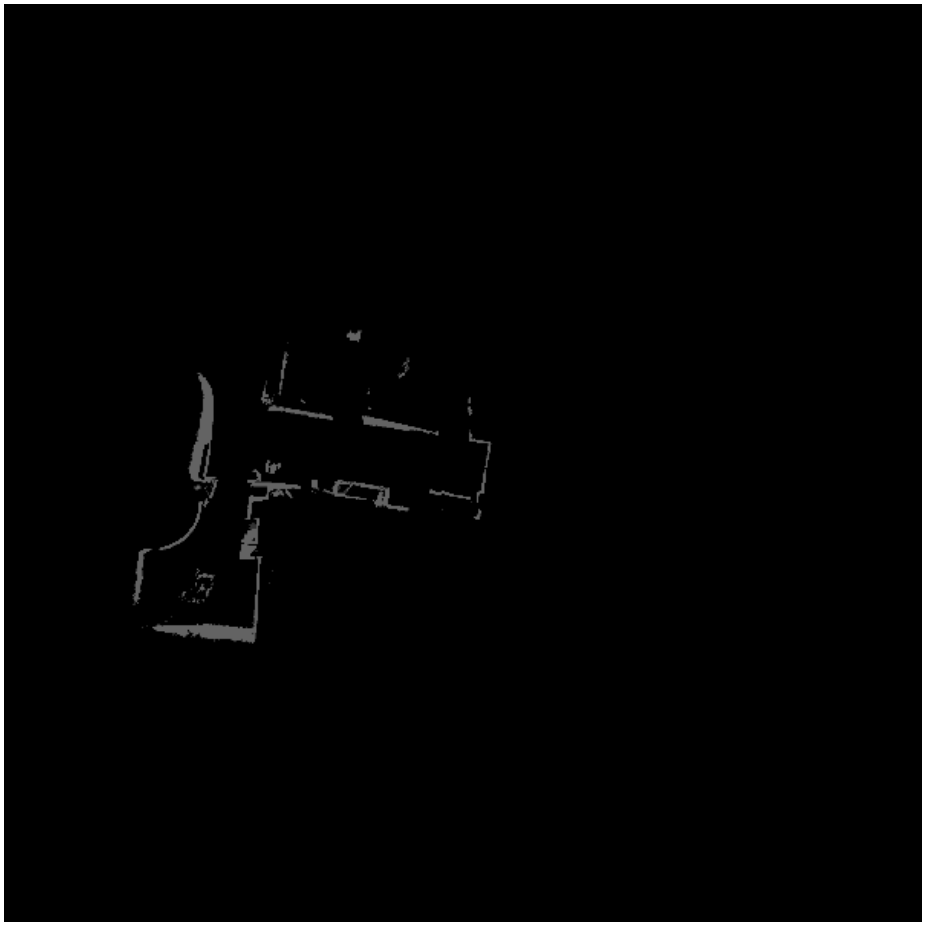

Figure 3: Fixed map example.

The maps can be saved in the robot permanent memory as png files with a single call to the *cv::imwrite(...)* function. At the moment, there is no code to localize in a map in C++; a competitive visual localization algorithm is implemented for *Loomo* in the base Java development kit but has not yet been ported to C++. In the meantime, Monte-Carlo localization is a good alternative but must be implemented.