ECSE 6940
Practical Pose Estimation for Swarm Mobile Robots

Project Report

Chukwuemeka Osaretin Ike

Table of Contents

Introduction

Pose estimation plays a vital role in the overall task of robotic control, as the ability of a robot in an open environment to complete its tasks can often depend on having some knowledge of its position. Multiple approaches exist to estimate this positional information via the use of sensors and/or knowledge of the physical properties of the robot. Some of these sensors include inertial measurement units (IMU), ultrasonic sensors, and optical cameras, each with their own merits and drawbacks. One popular method of estimation is dead reckoning, or odometry, where the measurements of wheel speeds and robot heading can be combined with previous position to estimate the robot's new position. While it is simple to do, odometry is known to be useful mainly in situations with high error tolerances, as errors due to wheel slippage and incorrect encoder readings build up the longer the method is used.

In mapped and structured environments, the difficulty of estimating the robot's position can be mitigated with the presence of waypoints or location markers. These markers can increase the accuracy of positional estimates, and improve the overall efficacy of the robot.

Swarm robotics, on the other hand, is an approach to the coordination of multiple robots as a single system aimed at achieving a common goal or multiple complementary goals. Swarms are made up of multiple agents, and it is often their collective behavior that is used in completing a specific task. In the swarm case, there is an added layer of complexity that comes from needing to estimate the generalized location of the swarm as opposed to the location of the individual robots. For starters, there are multiple ways that this position can be defined, and the choice needs to be made based on the overall goals of the swarm. Moreover, there are additional concerns regarding the communications within the swarm and possibility of adversarial/faulty agents.

In our work, we chose to focus on the simpler aspects of the problem by choosing a 2-dimensional map and defining the swarm's position to be the center of mass - i.e. the center point of all agents in the swarm. We also assumed there were no adversarial or defective members in the swarm. We further capitalized on the lower difficulty of mapped environments by using fiducial markers which each robot used in estimating its own location via inverse projections. These choices allowed us to tackle and assess some of the finer points of vision-based pose estimation for robots in a more forgiving setting.

Overall, we built a system for the estimation of a swarm's 2-dimensional position in a mapped environment using asynchronous communications between the agents in the swarm. We anticipate that this system can then be built into a controller that specifies the movements of the swarm needed to complete tasks.

For our work, we employed the Khepera IV mobile robot, a differential robot running an embedded Linux operating system. We combined this robot with the Robot Operating System

(ROS) running on an Ubuntu host machine to deliver the complete functionality. The delivered artifacts for this project are this report, and the Project's Github Repository.

This report serves as a description of the main C++ package that was delivered as part of the course requirements, as well as a simple working manual for software development for the Khepera robot, and another for ROS. The report is divided into four major parts. Section 1 details the software and hardware components of our experimental setup, along with how to properly set up each part. Section 2 describes the Pose Estimation package that was developed using the tools and frameworks described in section 1. Section 3 details the development process for the Khepera robot by walking the reader through developing a package for the robot. The final section serves as a learning guide for new users of ROS with a practical example demonstrating key features of the framework.

Experimental Setup

This section details the elements that made up this project, including instructions on how to set up and use the relevant parts. A list of relevant links detailing aspects are made available where applicable. The following is a list of the components before we go into the details of each:

1. Host Computer with the following software:
    a. Windows Subsystem for Linux 2 (*Ubuntu 20.04*)
    b. Khepera Library
    c. OpenCV Library
    d. Robot Operating System (*Noetic*)
    e. ROS Serial Package
2. Khepera IV Mobile Robots
3. Map & ArUco Markers

Host Computer Setup

Windows Subsystem for Linux

Development for the Khepera requires access to a Linux system, so we used the opportunity to explore the relatively new Windows Subsystem for Linux (WSL). WSL provides a native Linux experience within the familiar Windows operating system. We used version 20.04 of the Ubuntu distribution.

Installation

WSL requires a Windows 10 system or above. To install it, open up a *PowerShell* instance in *administrator* mode, and run the command below

```
wsl –install
```

This command will enable the features for WSL and install the Ubuntu distribution by default. The first time you launch this Linux installation, some files will automatically decompress and be placed in the appropriate locations. Once this is done, you will need to set up the Linux user info when you launch the distribution from the Start Menu. The console will request the user enter a username, followed by a password, similar to the figure below.

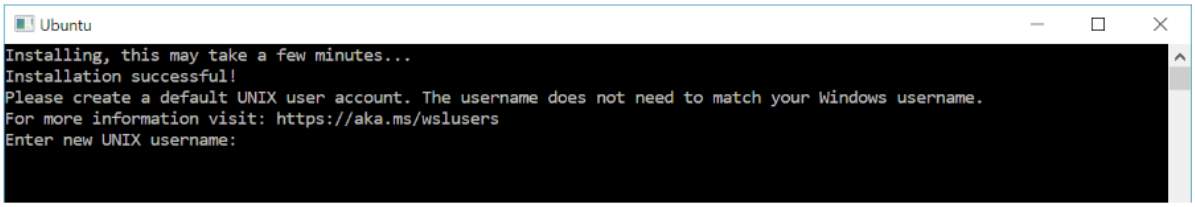

Figure : Initial Linux Launch Screen with Username Prompt.

These will serve as the login credentials when using WSL. The password will be requested most often when the *sudo* command is used.

Usage

Once WSL is installed, the terminal can be used just as it would be in a native Linux system.

Port Forwarding

WSL 2 has a virtualized ethernet adapter with its own unique IP address, similar to how a virtual machine would. While this does not have an effect in using SSH to connect to the Khepera, Rosserial (discussed below) needs access to a TCP port on the host machine. In order for the Kheperas running Rosserial to access the WSL system from your local network, any ports that are needed have to be forwarded from the Windows IP address to the WSL IP. To do this, I used a PowerShell script based on this post and this post to forward the specific TCP port I needed to the Linux system. This post explains why forwarding is necessary, but following it directly did not work for me.

The *port_forwarding.ps1* script is included in the project's repo to demonstrate how to perform the port forwarding. Note that it cannot be run if the script is on the Linux system due to Powershell's security policies. Copy it to your Windows' drive before attempting to run it.

Unfortunately, the script cannot be run automatically without any user input, since it requires *administrator* privileges. It can be automated slightly by setting up Windows to run the script on startup. Here's an example.

Development Environment Setup

In order to develop applications for the Khepera, the host Linux machine needs to be set up for cross-compilation. The following instructions assume the user is using a Bash shell in Ubuntu 20.04 on an x64 machine. I also assume we're using the CMake utility to build all code and packages.

Prerequisites

1. Install the following packages on the host to ensure a smooth installation of the Khepera and OpenCV libraries.

```
sudo apt install libv4l-0 v4l-utils libv4l-dev libc6-armel-cross
libc6-dev-armel-cross lib32z1 binutils-arm-linux-gnueabi
```

Khepera Toolchain

The toolchain contains the compiler needed to build the binaries that will run on the Khepera robot. The version used is available here. After downloading the toolchain, do the following:

1. Uncompress the toolchain into */usr/local*

```
sudo tar -xjf khepera4-yocto-light-kb1.0.tar.bz2 -C /usr/local
```

It is important that the toolchain is placed in this folder because of hard-coded links in the code.

2. Set the environment variables to allow the shell to recognize the installation.

```
export
K4_YOCTO=/usr/local/khepera4-yocto/build/tmp/sysroots/i686-linux/usr/
bin
export PATH=$PATH:$K4_YOCTO/armv7a-vfp-neon-poky-linux-gnueabi
```

These will have to be run in every bash instance to ensure the toolchain is available for use. To automate this process a little, add those two lines to the end of the ~/.bashrc file. Doing so eliminates the need to run the commands every time the shell is opened.

Usage

To use the toolchain, specify the system name, processor, and the C/C++ compilers in the CMakeLists.txt file similar to below. A sample CMakeLists.txt is also available in the project repo.

```
set( CMAKE_SYSTEM_NAME Linux )
set( CMAKE_SYSTEM_PROCESSOR arm )

set( CMAKE_C_COMPILER
${tools}$ENV{K4_YOCTO}/armv7a-vfp-neon-poky-linux-gnueabi/arm-poky-linux-gn
ueabi-gcc )

set( CMAKE_CXX_COMPILER
${tools}$ENV{K4_YOCTO}/armv7a-vfp-neon-poky-linux-gnueabi/arm-poky-linux-gn
ueabi-g++ )
```

Note that the compiler locations require that the *K4_YOCTO* environment variable is set. An error would be thrown if it is not.

Khepera Library

The version of the Khepera library used in this package is v2.0, and can be downloaded here.

1. Extract the library to `<khepera_dev_path>` and then build it.

```
tar -xjf libkhepera-2.0.tar.bz2 -C <khepera_dev_path>
cd <khepera_dev_path>/libkhepera-2.0
make clean
make all
```

2. Copy the *libkhepera.so.2.0* and *libkhepera.a* files from the *libkhepera-2.0/build-khepera-3.5.7-custom/lib/* folder to the Khepera's */usr/lib/* folder.

```
scp <khepera_dev_path>/libkhepera-2.0/build-khepera-3.5.7-custom/lib/
root@<khepera_ip>:/usr/lib/
```

3. The header files that are generated in the *libkhepera-2.0/build-khepera-3.5.7-custom/include/* folder will need to be included when building applications for the robot.

OpenCV Library

Instructions on how to cross-compile the OpenCV library for a generic ARM-based Linux system are available here, but certain aspects of those instructions need to be adapted to fit our use case. We do not need the *gcc-arm-linux-gnueabi* tools since our toolchain is the *arm-poky-linux-gnueabi* tools we installed in the Toolchain section above.

1. Clone the OpenCV and OpenCV_contrib repositories into the same folder. OpenCV contains the core OpenCV files, while the contrib repo contains additional modules, including the ArUco library.

```
mkdir <opencv_dir>
cd <opencv_dir>
git clone https://github.com/opencv/opencv.git
git clone https://github.com/opencv/opencv_contrib.git
```

2. Download the three cmake files available in the *opencv-toolchain-files* folder and place them in *<opencv_dir>*.

3. Set up CMake with the appropriate settings.

```
cd <opencv_dir>
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release
-DOPENCV_EXTRA_MODULES_PATH=../opencv_contrib/modules/aruco
-DOPENCV_ENABLE_CXX11=OFF -DSOFTFP=ON -DBUILD_TESTS=OFF
-DBUILD_EXAMPLES=OFF
-DCMAKE_TOOLCHAIN_FILE=<opencv_dir>/arm-gnueabi.toolchain.cmake
-DBUILD_OPENCV_PYTHON=OFF ../opencv
```

   Information on these and more parameters is available here. Note that *DCMAKE_TOOLCHAIN_FILE* should point to the location where the toolchain files were placed in (2). This specifies that the make utility uses the Toolchain, allowing us to run the OpenCV binaries and our own code on the Khepera IV.

4. Build the library.

```
make -j5
make install
```

5. Copy the files from the *lib/* folder in your build directory to the robot's */usr/lib/* folder.

```
scp <opencv_dir>/build/lib/* root@<khepera_ip>:/usr/lib/
```

The built OpenCV library should now be compatible with the Khepera. It can be included when building code for the robot by modeling your *CMakeLists.txt* after the snippet below.

```
set( OpenCV_DIR “<opencv_dir>/build” )
find_package( OpenCV REQUIRED )
include_directories( ${OpenCV_INCLUDE_DIRS} )

add_executable( <target_name> <target_source_file> )
target_link_libraries( <target_name> PUBLIC ${OpenCV_LIBS})
```

Robot Operating System

In this project, we used the Robot Operating System (ROS) 1 *Noetic* distribution for the message passing, and to interface the Khepera robots with the host machine. With WSL installed, ROS can be installed by following these instructions for a native Linux system. We dedicate a subsequent section to more detailed explanations of ROS usage.

ROS Serial

Rosserial is a ROS package that allows robots that cannot run ROS to connect over multiple channels (USB, bluetooth, or WiFi) to a host machine running ROS. To do this, the package creates ROS nodes that listen over specified ports on the host machine for communications from the robots. The Kheperas cannot run ROS natively, so they rely on this package to communicate with the host computer, which then serves as the backbone for all the communications done within the swarm. In our project, we used the WiFi channel which listens on a TCP port on the host machine for messages from the Khepera, then publishes those messages to the appropriate topics. This section describes how to install the package, while the Pose Estimation section explains how to use the package for the project.

Rosserial can be installed from Source based on these instructions which are repeated here:

1. Create a catkin workspace in the home directory and a *src/* subdirectory within it.

   ```
   cd ~
   mkdir <ws> && mkdir <ws>/src
   ```

2. Clone the rosserial repository into the src subdirectory.

   ```
   cd <ws>/src
   git clone https://github.com/ros-drivers/rosserial.git
   ```

3. Catkin make and install the package.

   ```
   cd <ws>
   ```

```
catkin_make
catkin_make install
```

4. The commands above generate the rosserial_msgs and build the *ros_lib* library in the *<ws>/install/* directory. To make the rosserial packages available to ROS, run the following command.

```
source <ws>/install/setup.bash
```

5. The above command has to be run in every terminal you use, so we suggest adding it to *~/.bashrc* to automate the process. Once the packages have been built, select a directory to hold the library files for the embedded system:

```
mkdir <some_directory> && cd <some_directory>
rm -rf ros_lib examples
rosrun rosserial_embeddedlinux make_libraries.py
```

6. Add the newly created *ros_lib/* folder's contents to the include path for the cross-compiler you're using for the project. In our case, we copied its contents directly into the Khepera's include/ folder which all the packages' *CMakeLists.txt* files reference, and we got the desired result.

Khepera IV Robot

The Khepera IV is a differential mobile robot with a Gumstix Overo 800 MHz ARM Cortex-A8 Processor based on the armv7l architecture. It runs an embedded Linux Angstrom Distribution and provides a standard C/C++ environment for its applications. This section details how to work with the robot over USB and over the network.

USB Setup

To connect with and use the Khepera over USB, the user will need either Putty or TeraTerm. First, connect the Khepera using the USB cable, then check Device Manager to confirm which COM port it is connected to. Assuming we're using Putty, select the "Serial" *Connection type*, specify the *Serial line* with the correct port, and set the *Speed* to 115200 as shown in the figure below. Finally, click *Open*.

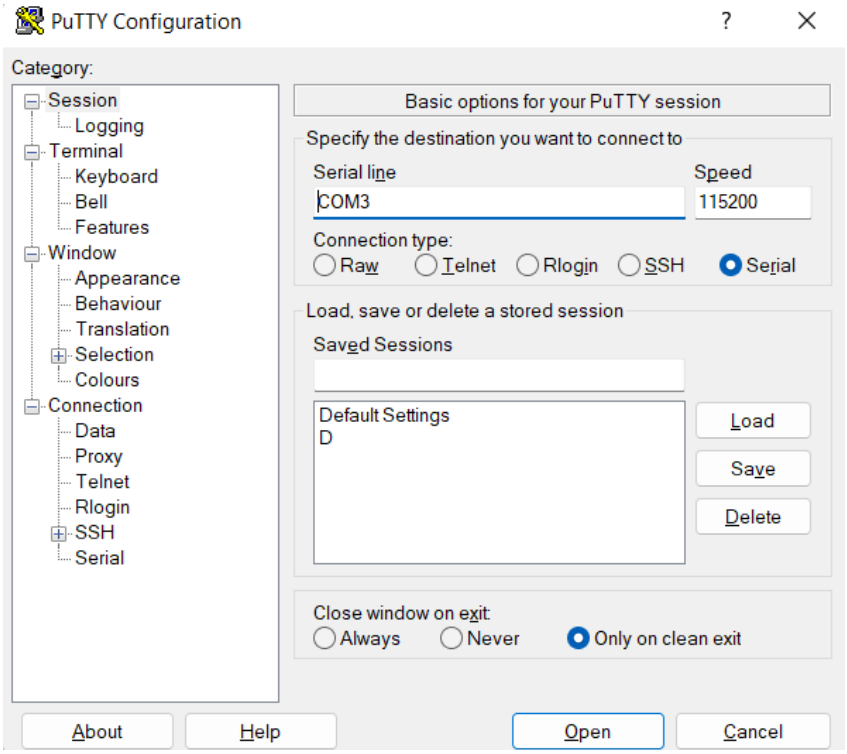

Figure : Putty Configuration Screen.

If the connection is successful, the console should open, and it should show the Khepera's login prompt. If the terminal is blank. Press *Enter*, and the prompt should then show up.

WiFi Setup

Here, we detail how to connect the Khepera to a WiFi network with WPA2-Personal encryption. Follow the following instructions on the Khepera command line over USB.

1. Run the command

   ```
   wpa_passphrase My_SSID My_Passphrase
   ```

   which generates an output similar to

   ```
   network={
           ssid="My_SSID"
           #psk="My_Passphrase"
           psk=My_PSK
   }
   ```

2. Copy the output of the above wpa_passphrase command into the file at */etc/wpa_supplicant/wpa_supplicant-<interface>.conf* where <interface> is the device you'd like to use.

3. Assign a static IP address to the robot on the network by editing */etc/systemd/network/wifi.network* to

```
[Match]
Name=<interface>

[Network]
#DHCP=v4
Address=<IP Address>/<netmask>

[DHCP]
RouteMetric=20
```

   The <IP Address> and <netmask> can be chosen based on the specific network that the user is working with.

4. Reboot the Khepera.

```
reboot
```

5. Run the following command and crosscheck that the address following inet matches what was specified in wifi.network.

```
ip a show <interface>
```

   You can also check that the Khepera is accessible over the network by pinging it from the Linux host.

```
ping <khepera_ip>
```

   If this gives a positive response, then the Khepera is now connected to the network and accessible from the host.

Once the Khepera is connected to the local network, the user can also SSH into the robot and avoid using the USB. This is most helpful for connecting to several Kheperas. To connect, run the following command from the Linux host.

```
ssh root@<khepera_ip>
```

Camera Calibration

To be able to use the Khepera's camera for pose estimation, the camera needs to be calibrated. The files contained in this folder are intended to streamline this process. This link explains the process of calibrating the camera with a detailed description of the C++ code. Here, I give a quick summary of the steps involved in calibration using the repo's *camera-calibration* package.

1. Print a calibration chessboard and attach it to a solid object you can move around easily - e.g back of a hardcover notebook. You can also keep the chessboard fixed, and move the Khepera around instead.
2. Edit the <Square_Size>, <BoardSize_Width>, <BoardSize_Height> entries in the *config.xml* file to match the dimensions of the chessboard you printed above.
3. Build the *cameraCalibration* binary file.

```
cd camera-calibration/
cmake .
make
```

4. The *CMakeLists.txt* file included in this folder generates the output in a bin/ subfolder, so transfer the cameraCalibration binary to the camera-calibration/ folder, then transfer the entire folder to the Khepera.

```
cd camera-calibration/
mv bin/cameraCalibration .
scp -r ../camera-calibration/ root@<khepera_ip>:~/
```

5. On the Khepera, take calibration images with the camera by running the *takeCalibrationImages* bash script which takes 80 photos and saves them in an *images/* subfolder. As the script runs, rotate the chessboard through several different positions in order to give the calibration utility enough angles in the next step. Be sure to do this in a well-lit environment with the whole chessboard in the camera's field of view. This video shows the image-capture process you should try to emulate albeit without the helpful GUI.

```
cd camera-calibration/
chmod +x takeCalibrationImages.bash
./takeCalibrationImages
```

6. Once the bash script is done taking images, run the *cameraCalibration* binary. This will use the settings in config.xml to calibrate the camera and write the camera data to the filename specified in its <Write_outputFileName> tag.

```
./cameraCalibration
```

   In any subsequent work that uses OpenCV for pose estimation, *posePublisher* for example, <Write_outputFileName> needs to be in the binary's working directory. If you don't change anything within the code, the camera data will be written to *cameraData.yaml*. We will refer to this file later on in the Pose Estimation and the example Khepera application sections.

Map and ArUco Markers

To facilitate pose estimation on the robots, we used ArUco Markers with known locations. These markers are a part of the OpenCV library that we built earlier. The following two figures show a virtual and real example of the mapped environment used in the pose estimation. Whenever we refer to the map in this report, this is what we are talking about.

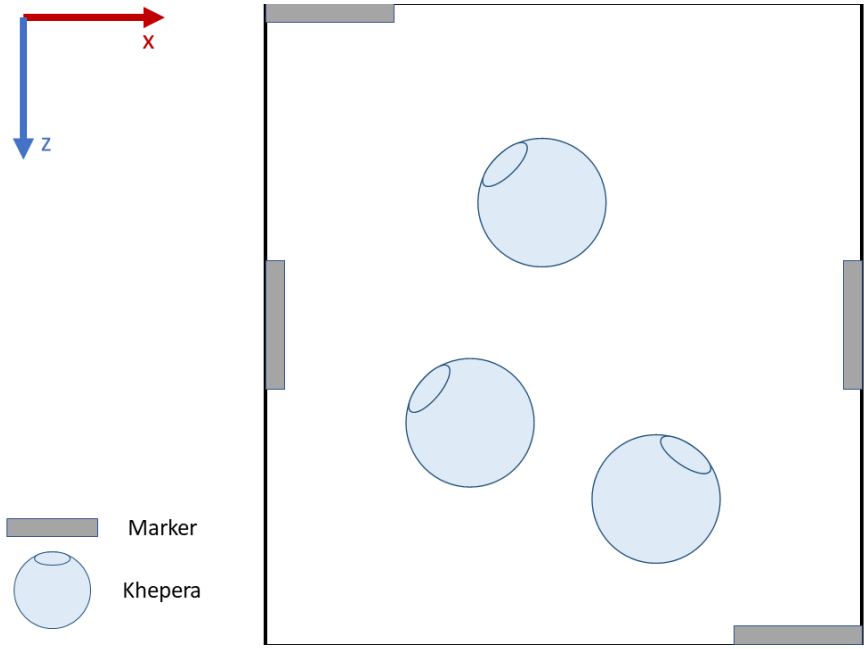

Figure : Virtual Mapped Environment with Markers Placed Throughout.

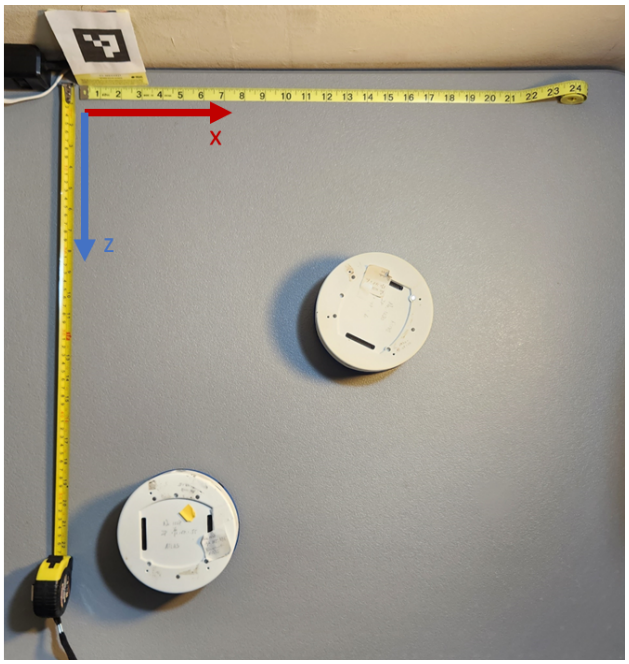

Figure : Physical Implementation of Mapped Environment with One Tag.

Pose Estimation

Description

As part of the requirements of this course, we were tasked with developing a functional ROS package that implemented pose estimation for a swarm of Khepera robots in a simple mapped environment. As discussed in the introduction, pose estimation in mapped environments can be simplified with the use of waypoints, which the robots can readily use to estimate their location. To this end, we used ArUco markers which come pre-packaged as part of the OpenCV contributed library (install instructions above), to aid in the process. This allowed each robot to know its precise location in the map assuming there was a marker in its field of vision. With this information, all robots then performed consensus averaging to estimate the swarm's pose. The swarm's pose is taken to be the swarm's 2-dimensional center of mass as shown in the figure below.

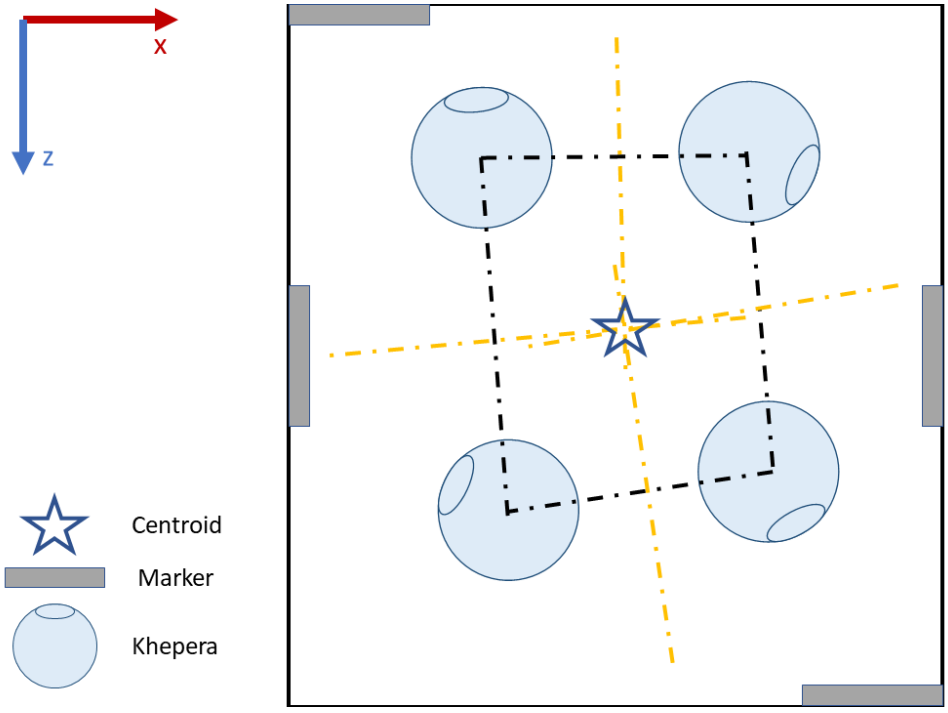

Figure : Mapped Environment Showing Swarm's 2D Center of Mass.

Consensus averaging is a method used in distributed settings to generate a global estimate of a value in which agents only possess incomplete information and a route for asynchronous message passing. The agents, by iteratively communicating among themselves, are able to agree on a value that is close to the true value under weak conditions[1]. In this project, we modified the original algorithm slightly to accommodate the possibility of dropped communications. The process for a single Khepera is detailed below:

1. Estimate the Khepera's position using visible ArUco tags
2. Compute the Khepera's current estimate of the Swarm's position:
    a. If a neighbor's estimate is available:

$$myEstimate = a * myEstimate + b * neighborEstimate$$

   b. If no neighbor's estimate is available:

$$myEstimate = c * myEstimate + d * myPose$$

3. Publish the updated estimate onto the ROS topic.

The coefficients a and b are weights that represent the comparative level of trust the Khepera has in its estimate versus its neighbors'. In the case where the neighbor drops out, c and d control how quickly the Khepera's estimate will return to its own position - this value reflects the assumption that it is the only robot in the swarm. These estimates will converge, provided the weights satisfy

$$a + b = 1,$$
$$c + d = 1.$$

With this algorithm running on multiple Kheperas, every agent's estimate converged on a value close to the swarm's centroid given sufficient time.

Usage

In order to use the swarm centroid estimation package that was developed for this project, the user will need two components - the Khepera package and the host package. The Khepera package is available in the *khepera/centroid-estimator* folder, while the host package is available in the host's *swarm centroid estimator* folder. As mentioned in a previous section, we used Rosserial to facilitate the communications between the Kheperas by creating two topics - one which each robot publishes its centroid estimates to, and another which each publishes its position to. The first topic, called *centroid_estimate*, is a stream where all the Kheperas in the swarm publish their estimates of the swarm's position and receive their neighbors' estimates for use in the consensus averaging. The second topic, called *khepera_pose*, is used by a node on the host to calculate the true centroid as a form of error checking. The node publishes this true centroid on a third topic called *true_centroid*.

Setup

First, we set up the packages with the following steps.

1. Clone the project's repository

```
git clone https://github.com/Chukwuemeka-Ike/swarmPoseEstimation.git
```

2. Navigate to the host folder, run catkin make, then source the workspace to make the packages available to the overall ROS system.

```
cd swarmPoseEstimation/host
catkin_make
source devel/setup.bash
```

   This builds the host's swarm_centroid_estimator ROS package which contains the centroid_calculator node. The package also contains a launch file that spawns that node

and two Rosserial nodes that connect to two Kheperas over TCP ports 11411 and 11412 on the host machine.

3. Navigate to the Khepera's *centroid-estimator/* folder and compile the code.

```
cd swarmPoseEstimation/khepera/centroid-estimator/
cmake .
make
```

This builds the centroidEstimator binary in a *bin/* subdirectory. If the *ros_lib* libraries are not placed in the compiler's path, an error will be thrown regarding the include statements in the source file.

4. On each Khepera, create a folder called *centroid-estimator/* and copy the camera data file from the calibration folder to it.

```
cd ~ && mkdir centroid-estimator
cp camera-calibration/cameraData.xml centroid-estimator
```

5. On the host, Create a *tagLocations.yaml* file modeled after this file and copy it to the Kheperas' *centroid-estimator/* folder.

```
scp tagLocations.yaml root@<khepera_ip>:~/centroid-estimator/
```

6. Copy the binary that was compiled on the host to each Khepera's *centroid-estimator/* folder.

```
scp bin/centroidEstimator root@<khepera_ip>:~/centroid-estimator/
```

Running

This section, like the code in the repo, assumes we're working with a swarm size of 2.

1. Launch the host-side functionality with the command below.

```
roslaunch swarm_centroid_estimator centroid_calculator.launch
swarm_size:=2
```

2. On each Khepera, launch the centroid estimator

```
cd ~/centroid-estimator
./centroidEstimator
```

Overall, when the pose estimation package is running for a swarm of *n* Kheperas, the user will have the following running:

1. ROS core on the host
2. *n* Rosserial_python nodes on the host connected to *n* Kheperas on *n* TCP ports
3. centroidEstimator program running on *n* Kheperas

The ROS node graph should look similar to the figure below.

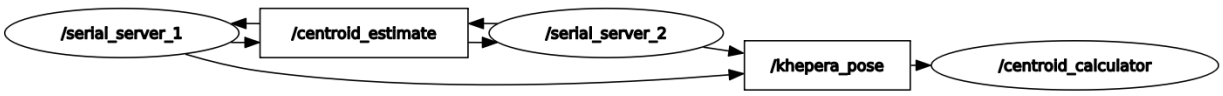

Figure : ROS Node Graph with Centroid Estimation Running on 2 Kheperas.

References

1. J. N. Tsitsiklis, D. P. Bertsekas and M. Athans, "Distributed Asynchronous Deterministic and Stochastic Gradient Optimization Algorithms," *IEEE Transactions on Automatic Control*, Vol. 31, No. 9, 1986, pp. 803-812.

Khepera Development

This section builds on the groundwork laid in the Khepera and Development Environment Setup sections from earlier. To use the instructions here, all the necessary components and communication channels would have to be functioning properly.

In this tutorial, we will create a package that uses OpenCV to look for ArUco tags and rotates the Khepera by ~90 degrees each time it sees one. To do this, our code needs two main components - the OpenCV tag detection, and control of the Khepera motors. We'll create the C++ file, CMake instructions, build the binary, then copy it to the Khepera and run it. The code from this tutorial is also available in the *khepera-examples* package in the project's repo.

tag_dodger.cpp

Create a C++ file called *tag_dodger.cpp* where we'll place the tag dodging code. We'll walk through the code step-by-step below.

Globals

Start with the library includes and namespaces. Here, we add the Khepera and OpenCV libraries and three C++ standard libraries.

```
#include <khepera/khepera.h>
#include <opencv2/opencv.hpp>
#include <opencv2/aruco.hpp>
#include <signal.h>
#include <stdlib.h>
#include <iostream>

using namespace std;
using namespace cv;
```

Next, we create a pointer to the robot's microcontroller and a variable to exit the program gracefully when *ctrl-c* is pressed.

```
static knet_dev_t * dsPic; // robot pic microcontroller access.
static int quitReq = 0; // quit variable for loop.
```

Prototype all the functions which we'll explain with their individual definitions.

```
static void ctrlc_handler(int sig);
void getCameraParams(Mat &cameraMatrix, Mat &distCoeffs, string
camera_params_file);
void captureInputImage(string capture_command, string capture_file, Mat
&inputImage);
bool areTagsPresent(
```

```
  Mat &inputImage, Mat &outputImage, Mat &cameraMatrix, Mat &distCoeffs,
  vector<Vec3d> &rvecs, vector<Vec3d> &tvecs,
  vector<int> &markerIds, Ptr<aruco::Dictionary> &dictionary,
  Ptr<aruco::DetectorParameters> &parameters,
  vector<vector<Point2f> > &markerCorners,
  vector<vector<Point2f> > &rejectedCandidates,
  string output_image_file
);
void rotateKhepera(knet_dev_t * dsPic, int &lpos, int &rpos);
```

Main Function

Open the main function and create some variables for the Khepera library, motor control, and OpenCV.

```
int main(int argc, char **argv) {
  // Motor control and position variables.
  int lpos, rpos;
  int kp,ki,kd;
  int pmarg,maxsp,accinc,accdiv,minspacc, minspdec; // Speed profiles

  bool tagsPresent;

  // Set the libkhepera debug level - Highly recommended for development.
  kb_set_debug_level(2);

  // File names and camera capture command.
  string capture_file = "input_img.jpg";
  string output_image_file = "output_img.jpg";
  string camera_params_file = "cameraData.xml";
  string capture_command = "v4l2grab -d /dev/video6 -o "
                           + capture_file + " -W 752 -H 480 -q 85 -I -1";

  // OpenCV and ArUco variables for images, camera parameters, and pose
info.
  Mat inputImage, outputImage, cameraMatrix, distCoeffs;
  vector<int> markerIds;
  vector<vector<Point2f> > markerCorners, rejectedCandidates;
  vector<Vec3d> rvecs, tvecs;
  Ptr<aruco::DetectorParameters> parameters =
                                      aruco::DetectorParameters::create();
  Ptr<aruco::Dictionary> dictionary =
                    aruco::getPredefinedDictionary(aruco::DICT_4X4_50);
```

Initialize the Khepera library, and open communication with the robot's microcontroller.

```
  // Initialize libkhepera and robot access.
```

```
  if ( kh4_init(0,NULL)!=0) {
      printf("\nERROR: could not initiate the libkhepera!\n\n");
      return -1;
  }

  // Open the robot socket and store the handle in a pointer.
  dsPic  = knet_open( "Khepera4:dsPic", KNET_BUS_I2C, 0, NULL);

  if (dsPic == NULL) {
      printf("\nERROR: could not initiate communication with Kh4
dsPic\n\n");
      return -2;
  }
```

Initialize the motor controllers by setting the tuned motor parameters and some speed profile information, then set the robot as idle. The parameters were obtained from the Khepera's example code available with the Khepera library installed earlier.

```
  // Initialize the motor controllers.
  // Tuned motor parameters.
  pmarg = 20;
  kh4_SetPositionMargin(pmarg, dsPic);                    // position
control margin

  // Configure the PID controller. Not useful for non error-based control.
  kp = 10;
  ki = 5;
  kd = 1;
  kh4_ConfigurePID( kp, ki, kd, dsPic);

  // Acceleration increment,  Acceleration divider, Minimum speed acc,
  // Minimum speed dec, maximum speed.
  accinc = 3;
  accdiv = 0;
  minspacc = 20;
  minspdec = 1;
  maxsp = 400;
  kh4_SetSpeedProfile(accinc, accdiv, minspacc, minspdec, maxsp, dsPic);

  // Put in idle mode (no control).
  kh4_SetMode(kh4RegIdle, dsPic);
```

```
// Set the signal for catching ctrl-c.
signal( SIGINT , ctrlc_handler );
```

Get the camera's parameters that we got and stored in *cameraData.yaml* when we calibrated the robot earlier.

```
// Get the camera parameters.
getCameraParams(cameraMatrix, distCoeffs, camera_params_file);
```

According to the manual, before we can take pictures with the robot, we need to set the camera format specifications. This has to be done every time the robot starts, so we do that here. The system command sends a bash command directly to the shell. This is bad practice, but no alternative was found before the project deadline.

```
  // Set up camera format.
  cout << "Setting Pipes\n";
  system("media-pipes.sh");
  cout << "Setting Format\n";
  system("media-formats.sh 752 480");
```

With everything set up, we can now enter the main loop that repeatedly takes a photo, checks whether any ArUco tags are visible, then rotates the Khepera by 90° if tags are visible.

```
  // Loop until ctrl-c key.
  while(quitReq == 0)
  {
    captureInputImage(capture_command, capture_file, inputImage);
    tagsPresent = areTagsPresent(
      inputImage, outputImage, cameraMatrix, distCoeffs,
      rvecs, tvecs, markerIds, dictionary, parameters,
      markerCorners, rejectedCandidates,
      output_image_file
    );

    if (tagsPresent) {
      rotateKhepera(dsPic, lpos, rpos);
    }
  }
}
```

Function Definitions

The *ctrlc_handler* function ends the program gracefully when Ctrl-C is pressed. It sets the motor speed to 0, sets the motor to idle, then turns off all the LEDs.

```
static void ctrlc_handler( int sig )
{
  quitReq = 1;
```

```
  // Stop the robot and set it to idle mode.
  kh4_set_speed(0, 0, dsPic);
  kh4_SetMode(kh4RegIdle, dsPic);

  // Clear RGB LEDs.
  kh4_SetRGBLeds(0,0,0,0,0,0,0,0,0, dsPic);

  // Revert to the original terminal if it was changed.
  kb_change_term_mode(0);

  exit(0);
}
```

getCameraParams opens the camera parameter file and reads the camera matrix and distortion coefficients into the OpenCV matrices.

```
void getCameraParams(Mat &cameraMatrix, Mat &distCoeffs, string
camera_params_file)
{
  FileStorage fs(camera_params_file, FileStorage::READ);
  if(!fs.isOpened())
  {
    cerr << "Unable to access camera parameters!" << endl;
    return;
  }
  fs["camera_matrix"] >> cameraMatrix;
  fs["distortion_coefficients"] >> distCoeffs;
  fs.release();
}
```

captureInputImage uses the provided capture command to take a photo and save the image into an OpenCV matrix.

```
void captureInputImage(string capture_command, string capture_file, Mat
&inputImage)
{
  cout << "Taking Image\n";
  system(capture_command.c_str());

  cout << "Reading image into matrix.\n";
  inputImage = imread(capture_file);
}
```

This function scans the input image for visible ArUco tags. If any are present, it draws the estimated pose of the tag and saves the photo to an output file before returning true. If none are present, it returns false.

```
bool areTagsPresent(
  Mat &inputImage, Mat &outputImage, Mat &cameraMatrix, Mat &distCoeffs,
  vector<Vec3d> &rvecs, vector<Vec3d> &tvecs,
  vector<int> &markerIds, Ptr<aruco::Dictionary> &dictionary,
  Ptr<aruco::DetectorParameters> &parameters,
  vector<vector<Point2f> > &markerCorners,
  vector<vector<Point2f> > &rejectedCandidates,
  string output_image_file
)
{
  // Perform detection on the input image.
  cout << "Detecting markers.\n";
  aruco::detectMarkers(inputImage, dictionary, markerCorners,
                               markerIds, parameters, rejectedCandidates);

  // If at least one marker is detected, we can attempt to estimate
  // the robot's pose.
  if (markerIds.size() > 0) {
     // Copy the input image to output and draw the detected markers.
    outputImage = inputImage.clone();

    cout << "Drawing detected markers.\n";
    aruco::drawDetectedMarkers(outputImage, markerCorners, markerIds);

    cout << "Estimating pose of the detected markers.\n";
    aruco::estimatePoseSingleMarkers(markerCorners, 0.053,
                               cameraMatrix, distCoeffs, rvecs, tvecs);

    int i = 0;
    drawFrameAxes(outputImage, cameraMatrix, distCoeffs, rvecs[i],
tvecs[i], 0.1);

    // Save the output image. Uncomment this for debugging.
    imwrite(output_image_file, outputImage);
    return true;
  }
  else {
    return false;
  }

}
```

rotateKhepera uses the Khepera library to set the controller in speed regulation mode, sets the speed, waits long enough for the robot to turn, then sets the speed to zero. The combination of motor speeds and wait time currently rotates the Khepera approximately 90 degrees to the right every time.

```
void rotateKhepera(knet_dev_t * dsPic, int &lpos, int &rpos) {
```

```
  long motspeed;

  // Get the current motor position.
  kh4_get_position(&lpos, &rpos, dsPic);
  printf("\nMotor positions: left %ld | right %ld\n", lpos, rpos);

  // Set control mode to speed regulation, then set the speed and wait n
seconds.
  kh4_SetMode(kh4RegSpeed, dsPic);
  long speed = 80.0;
  motspeed = (long)(speed/KH4_SPEED_TO_MM_S);
  kh4_set_speed(motspeed, -motspeed, dsPic);
  printf("\nRotating 2.25s at %.1f mm/s (pulse speed %ld) with speed
only\n", speed, motspeed);
  sleep(1.125);

  // Stop rotating, then check new position.
  kh4_set_speed(0 , 0, dsPic);
  kh4_get_position(&lpos, &rpos, dsPic);
  printf("\nMotor positions: left %ld | right %ld\n", lpos, rpos);
}
```

Note: Majority of the functions in this file take in arguments by reference, which allows us to reduce the memory requirements by altering the original variables directly.

CMakeLists.txt

With the C++ source file written, we can set up the compiler instructions on how to build the program.

```
cmake_minimum_required( VERSION 2.8 )

project( khepera_examples )

set( CMAKE_SYSTEM_NAME Linux )
set( CMAKE_SYSTEM_PROCESSOR arm )
set( CMAKE_C_COMPILER
${tools}/usr/local/khepera4-yocto/build/tmp/sysroots/i686-linux/usr/bin/arm
v7a-vfp-neon-poky-linux-gnueabi/arm-poky-linux-gnueabi-gcc )
set( CMAKE_CXX_COMPILER
${tools}/usr/local/khepera4-yocto/build/tmp/sysroots/i686-linux/usr/bin/arm
v7a-vfp-neon-poky-linux-gnueabi/arm-poky-linux-gnueabi-g++ )
add_compile_options(-std=c++11)

set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/bin)
```

```
set( OpenCV_DIR ~/kh4_dev/opencv/build )
find_package( OpenCV REQUIRED )

include_directories(${OpenCV_INCLUDE_DIRS} ${PACKAGE_INCLUDE_DIR}  khepera
)

add_executable( tagDodger tag_dodger.cpp )
target_link_libraries( tagDodger PUBLIC ${OpenCV_LIBS} khepera )
```

Build and Run the Code

On the Khepera, create a folder called *khepera-examples* in the home directory, and copy the camera data file from the calibration folder to the *khepera-examples/* folder.

```
cd ~
mkdir khepera-examples
cp camera-calibration/cameraData.xml khepera-examples
```

On your Linux host, run the following to set the CMake instructions, compile the code, then send it to the Khepera.

```
cmake .
make
scp tagDodger root@<khepera_ip>:~/khepera-examples
```

Finally, we can run the program on the robot.

```
cd ~/khepera-examples
./tagDodger
```

All things being equal, whenever the Khepera sees an ArUco tag, it will rotate 90 degrees to the right until *ctrl-c* is pressed.

There might be a set of warnings similar to those in the figure below that are displayed when the program starts. These are a result of redundancies in the Khepera library's definitions of IO pins and motor connections and should not prevent the program from functioning correctly.

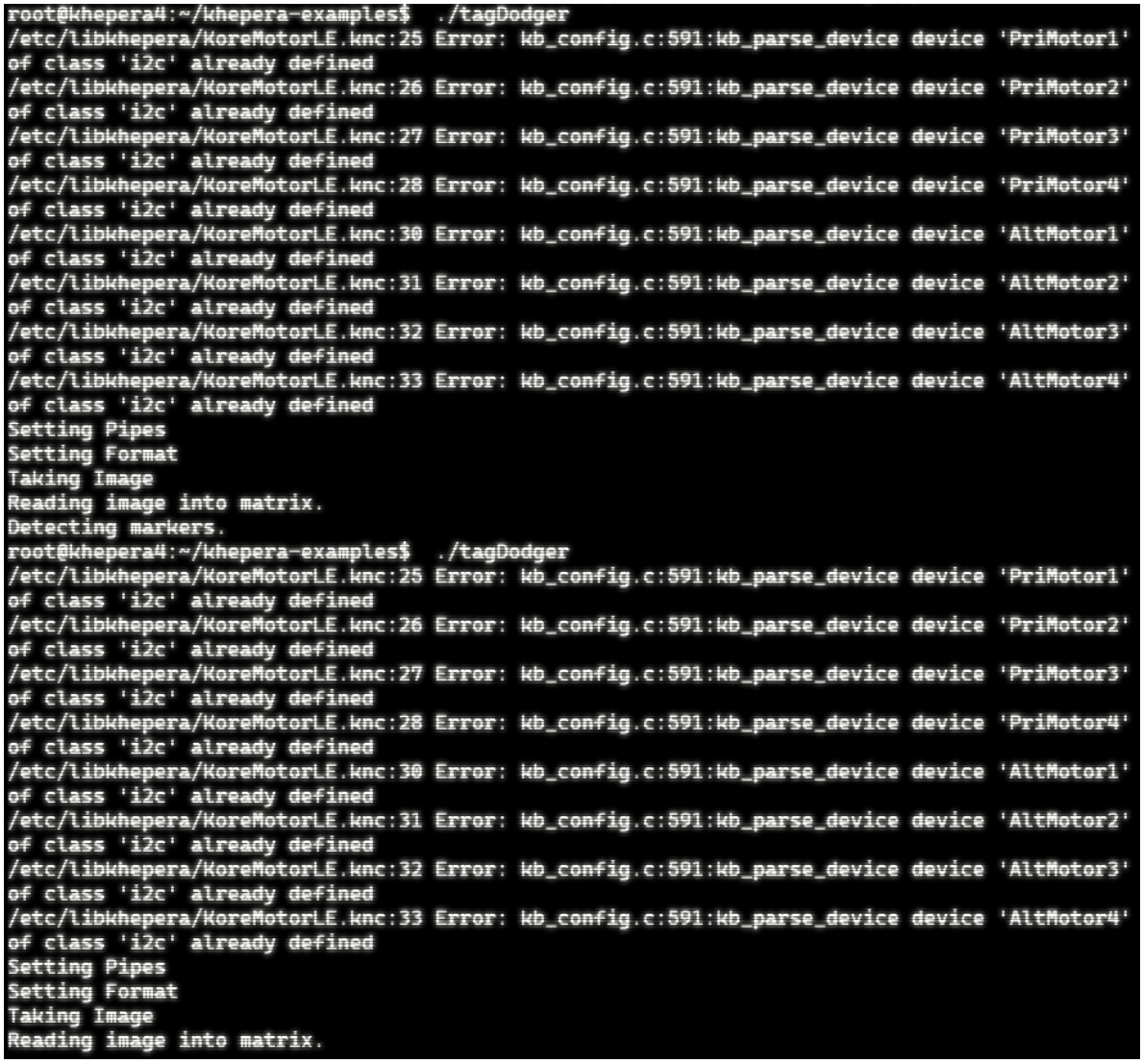

Figure : Initial Warnings when running tagDodger on the Khepera.

Robot Operating System

The Robot Operating System (ROS) is a powerful open-source robotics middleware project that provides numerous libraries and tools that streamline the process of building robotics applications. It provides drivers and algorithms that allow for quick iteration while developing packages for robots. This level of power, however, comes with a steep learning curve for many potential users. This section attempts to flatten that curve by describing how to install and use ROS 1 *Noetic* for basic package development on a Linux machine. We assume the user is running Ubuntu 20.04 with apt as the package manager.

Installation

This section is taken from here with some editing for brevity and consistency.

1. Configure your Ubuntu repositories to allow "restricted," "universe," and "multiverse." You can follow the Ubuntu guide for instructions on doing this.

2. Setup your computer to accept software from *packages.ros.org*.

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu
$(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

3. Set up your keys.

```
sudo apt install curl # if you haven't already installed curl
curl -s
https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc | sudo
apt-key add -
```

4. Make sure your Debian package index is up-to-date.

```
sudo apt update
```

5. Install the full ROS Noetic distribution.

```
sudo apt install ros-noetic-desktop-full
```

6. Add the following to the *~/.bashrc* file to ensure that ROS is available for use in every terminal you use.

```
source /opt/ros/noetic/setup.bash
```

7. Add some dependencies to build packages.

```
sudo apt update
```

8. Up to this point, you have installed what you need to run the core ROS packages. To create and manage your own ROS workspaces, there are various tools and requirements that are distributed separately. For example, rosinstall is a frequently used command-line tool that enables you to easily download many source trees for ROS packages with one command.

   To install this tool and other dependencies for building ROS packages, run:

   ```
   sudo apt install python3-rosdep python3-rosinstall
   python3-rosinstall-generator python3-wstool build-essential
   ```

9. Before you can use many ROS tools, you will need to initialize *rosdep*. *rosdep* enables you to easily install system dependencies for source you want to compile and is required to run some core components in ROS. If you have not yet installed rosdep, do so as follows.

   ```
   sudo rosdep init
   rosdep update
   ```

Usage

In this section, we will run through building a simple ROS package to help with getting started with ROS. By the end of the section, the user will be able to:

1. Install an existing ROS package
2. Create a catkin workspace
3. Create a ROS package
4. Design a simple ROS node in C++
5. Create ROS publisher and subscriber nodes in C++
6. Visualize the ROS node graph
7. Use a simple launch file to launch multiple ROS nodes

Package Installation

To install additional packages, you can run:

```
sudo apt install ros-noetic-PACKAGE
# e.g.
sudo apt install ros-noetic-slam-gmapping
```

To find available packages, see ROS Index or use:

```
apt search ros-noetic
```

Create a Catkin Workspace and ROS Package

First, we'll create a catkin workspace to hold our package. This can be done with the *mkdir* command. Create a subdirectory called *src* where the packages will go.

```
mkdir ws
cd ws
mkdir src
cd src
```

Next, create the package. We'll work with a package called *ros_examples*.

```
catkin_create_pkg ros_examples
```

This command creates two files in the newly created *ros_examples/* folder - *CMakeLists.txt*, and *package.xml*:

- *CMakeLists.txt* is a script containing build instructions for the *CMake* utility, and
- *package.xml* is the package's manifest where we'll define dependencies needed to function properly. The file does not affect your local development, but is useful when your package is published.

This pair of files hold the bulk of information that catkin will need to build your package correctly. We will give more details on how to edit both files once we've written the ROS program.

Build and Run a Simple ROS Node

Now that the package has been created, we will write a simple ROS node that prints to ROS's Info stream. Create a file called *helloRos.cpp* with the following contents.

```
// This is a ROS version of "hello, world", and was lifted from
// A Gentle Introduction to ROS by Jason O'Kane.

// Define the standard ROS classes.
#include <ros/ros.h>

int main(int argc, char **argv) {
    // Initialize the ROS system.
    ros::init(argc, argv, "hello_ros");

    // Create the ROS node.
    ros::NodeHandle nh;

    // Write the output log message.
    ROS_INFO_STREAM("Hello, ROS World!");
}
```

Next, edit *CMakeLists.txt*. The file will already hold a large amount of comments explaining different components. Replace the file's contents with the following.

```
cmake_minimum_required(VERSION 3.0.2)
project( ros_examples )

## Compile as C++11, supported in ROS Kinetic and newer
add_compile_options(-std=c++11)

set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/bin)

# Find catkin macros and libraries
find_package(catkin REQUIRED COMPONENTS
    roscpp
    geometry_msgs
)
include_directories(${catkin_INCLUDE_DIRS})

catkin_package(
   CATKIN_DEPENDS
   DEPENDS system_lib
)

add_executable( helloRos helloRos.cpp )
target_link_libraries( helloRos PUBLIC ${catkin_LIBRARIES} )
```

This instructs catkin to build our *helloRos.cpp* and find the *roscpp* and *geometry_msgs* dependencies we will soon need from ROS. Lastly, edit *package.xml* to the following.

```
<?xml version="1.0"?>
<package format="2">
  <name>ros_examples</name>
  <version>0.0.0</version>
  <description>TODO: Describe the package.</description>

  <maintainer email="jane.doe@example.com">Jane Doe</maintainer>
  <license>TODO</license>

  <buildtool_depend>catkin</buildtool_depend>

  <depend>roscpp</depend>
  <depend>geometry_msgs</depend>
</package>
```

This ensures the same dependencies are also specified in the manifest, which is useful when you're distributing the package to others. Now that we have the barebones package defined, we can compile the program.

```
cd ws
catkin_make
```

This command should be run from the main directory because it is designed to build all the packages in the workspace. If it completes successfully, you have now successfully written and built a ROS program.

Lastly, to run the ROS node, we need to first source the workspace into our shell.

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

Once this is done, the package should become available to the rest of the ROS system. To cross-check this, run the following.

```
rospack find ros_examples
```

This will output the directory where your package is defined. If it does, you can then run the node.

```
rosrun ros_examples helloRos
```

Note that if the ROS master is not already running, this command will throw an error. In order to run anything on ROS, roscore has to already be running. It is a set of nodes and programs that form the backbone of the ROS system. To run roscore, open another terminal and run:

```
roscore
```

Keep this terminal open as long as you need to run any ROS components. With roscore running, rerun the *helloRos* node. The node should now run correctly and output “Hello, ROS World!”

Create a Publisher and a Subscriber

Now that we have built a simple ROS node, we will create two more nodes in the *ros_examples* package that communicate over the ROS network.

To ensure modularity on the ROS network, the system makes use of *Topics*, *Publishers*, and *Subscribers*. Topics serve as named routes over which nodes can exchange messages. Each node can *publish* to a topic or *subscribe* to the topic by using the topic’s name. Topics serve as a means to broadcast information to whomever might be listening, or to gather it from multiple sources. In this way, they decouple the publishers and subscribers. Once a node publishes the results of its work on the topic, it can continue its operations without having to wait for an acknowledgement. Similarly, a subscriber that needs information can latch on to the topic without care of the source of the information it is receiving. This method of message passing is particularly useful for hybrid simulations, where real components can subscribe to topics whose

publishers are virtual, and vice versa, and complete their operations as if everything was implemented on real hardware.

To demonstrate this functionality, we will build a publisher and subscriber that work together to average a set of random values. Once every three seconds, the publisher will generate 3 random numbers, which it will then publish in a message over the */average* topic. Once the subscriber receives this message, it will compute the average of the numbers and write the answer to ROS_INFO_STREAM. Topics have message types that define the format of the content being sent and received over them. We use the geometry_msgs/Vector3 type here, as will be seen in the code below.

We will start by writing the publisher's code. Create a file called *average_publisher.cpp* in the *ros_examples/* folder and copy the following into it.

```
#include <ros/ros.h>
#include <geometry_msgs/Vector3.h>
#include <stdlib.h> // For rand().

int main (int argc, char **argv) {
    // Initialize the ROS system and create the node.
    ros::init(argc, argv, "average_publisher");
    ros::NodeHandle nh;

    // Create the publisher object.
    ros::Publisher pub = nh.advertise<geometry_msgs::Vector3>("/average",
1000);

    // Seed the random number generator.
    srand(time(0));

    // Loop at 0.5 Hz until the node is shut down.
    ros::Rate rate(0.5);

    while(ros::ok()) {
        // Create the message and fill it in with 3 random numbers
        // between 0 and 100.
        geometry_msgs::Vector3 vec;
        vec.x = int(double(rand())/double(RAND_MAX)*100);
        vec.y = int(double(rand())/double(RAND_MAX)*100);
        vec.z = int(double(rand())/double(RAND_MAX)*100);

        // Publish the message.
        pub.publish(vec);

        // Send a message to rosout with the details.
        ROS_INFO_STREAM("Sending random vector with:\n"
            << "\t\tx = " << vec.x << " "
            << "y = " << vec.y << " "
```

```
            << "z = " << vec.z
        );

        // Wait until the next iteration.
        rate.sleep();
    }
}
```

The subscriber has no way of knowing exactly when new messages will arrive. To accommodate for this, we need to write a callback function that will be called whenever messages arrive. This is where we will compute the average of the values that the publisher sent. Create a file called *average_subscriber.cpp* in the same folder, and copy the following into it.

```
#include <ros/ros.h>
#include <geometry_msgs/Vector3.h>
#include <iomanip> // for std::setprecision and std::fixed.

// The callback for each message received.
void inputMessageReceived(const geometry_msgs::Vector3& msg) {
    ROS_INFO_STREAM(std::setprecision(4) << std::fixed
        << "The average of \n\t\t" << msg.x << ", " << msg.y << ", and " <<
msg.z
        << " is: " << double((msg.x+msg.y+msg.z)/3)
    );
}

int main(int argc, char **argv) {
    //Initialize the ROS system and create the node.
    ros::init(argc, argv, "average_subscriber");
    ros::NodeHandle nh;

    // Create the subscriber object.
    ros::Subscriber sub = nh.subscribe("/average", 1000,
&inputMessageReceived);

    // Give ROS control.
    ros::spin();
}
```

Lastly, we need to edit *CMakeLists.txt* so CMake builds these two new nodes. Copy the following into the existing file from above.

```
add_executable( average_publisher average_publisher.cpp )
target_link_libraries( average_publisher PUBLIC ${catkin_LIBRARIES} )

add_executable( average_subscriber average_subscriber.cpp )
```

```
target_link_libraries( average_subscriber PUBLIC ${catkin_LIBRARIES} )
```

Build the new nodes with *catkin_make* from the workspace directory

```
cd ws
catkin_make
```

We can now run both nodes by opening two new terminals and running the following

```
# In both terminals
cd ws
source devel/setup.bash

# Terminal 1
rosrun ros_examples average_publisher

# Terminal 2
rosrun ros_examples average_subscriber
```

We will use the rqt tool to visualize the ROS network and see the information being written to the console (ROS_INFO_STREAM). With the subscriber and publisher nodes running, open the rqt GUI by running the following command in a new terminal.

```
rqt
```

Open the node graph by clicking ***Plugins>Introspection>Node Graph*** as is shown in the figure below.

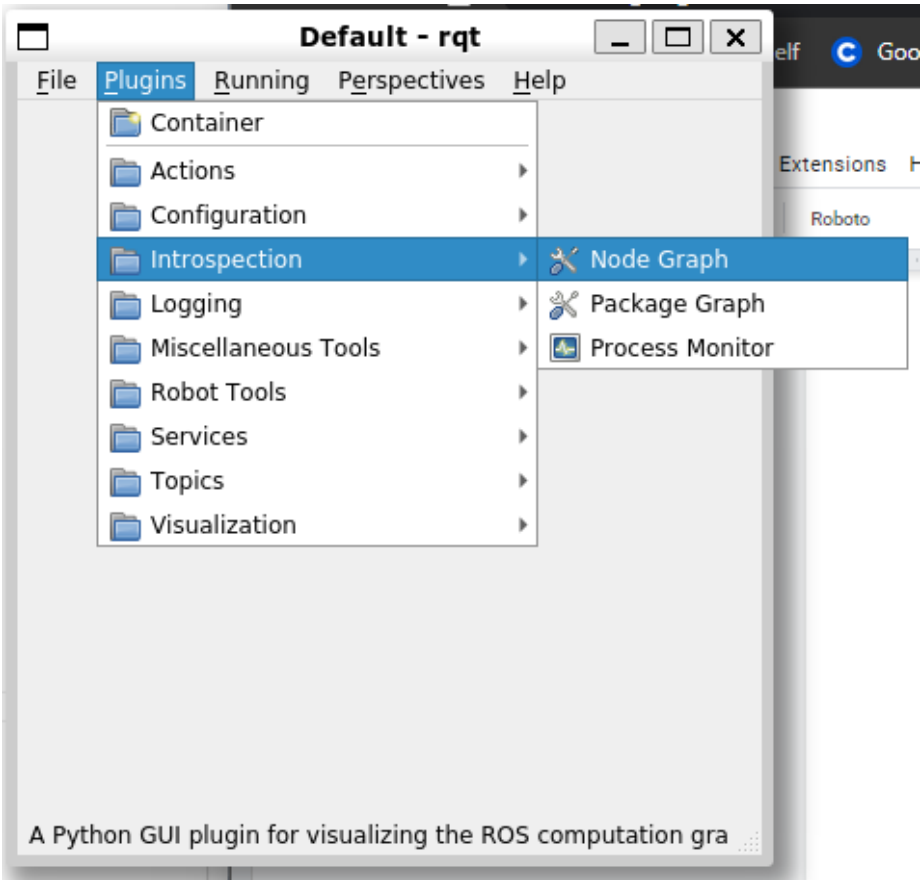

Figure : ROS RQT Utility GUI.

The graph should look similar to the figure below, which indicates that */average_publisher* is publishing to the */average* topic to which */average_subscriber* is subscribed.

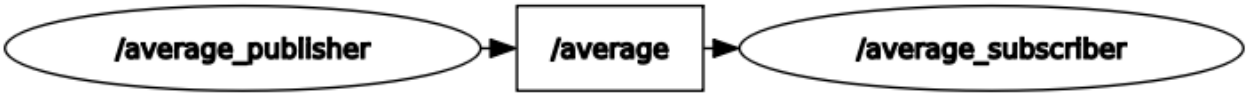

Figure : RQT Node Graph with Average Publisher and Subscriber.

The rqt tool provides a host of useful plugins that allow for debugging applications built with ROS. The plugin at ***Plugins>Logging>Console*** displays the information that we placed in the ROS_INFO_STREAM function in our code above.

The user has now been able to build and run two more nodes that publish and subscribe over a simple ROS topic. The contents of the */average* topic can also be shown from the command line with the rostopic package. To use it, run

```
rostopic list
```

which shows the list of topics currently available. Since we have our nodes still running, we can run

```
rostopic echo average
```

This will echo the messages being sent by */average_publisher* onto the topic.

One more thing to note is that communication via topics is many-to-many. This keeps the nodes loosely coupled. For example, if we run multiple publisher and subscriber nodes, they would all be linked over the same topic unless they use specific namespaces. I won't cover namespaces here, but they can be useful in ensuring subscribers only get data from certain publishers.

The rosrun commands used above use the default node name for each node. As a result, if you try to run the same node from multiple terminals, each new iteration will cause the previous one to stop running. To avoid this, you can specify a new node name with the command below.

```
rosrun khepera average_subscriber __name:=<new_node_name>
# E.g.
rosrun khepera average_subscriber __name:=sub_2
```

The following figure shows an example of the many-to-many communications with the same topic.

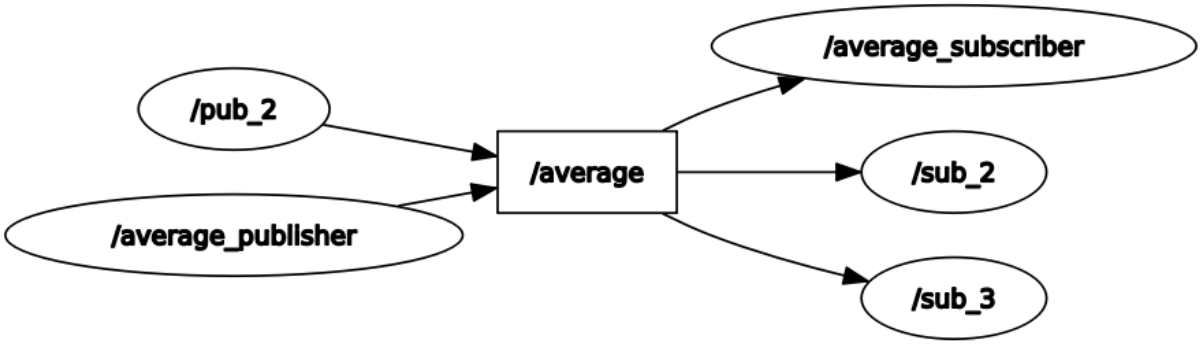

Figure : Multiple Nodes Publishing and Subscribing to the Same Topic.

Run Multiple Nodes with a Launch File

In cases where we need to run several nodes simultaneously, it can quickly get tedious to have to open a single terminal for each node. Using a launch file streamlines the process, allowing us to start every node, and the roscore, from a single terminal. The only tradeoff is that we have to choose one node whose output will be printed to the terminal. Every other node's output will be suppressed.

Create a file named *average_calculator.launch* and fill it with the following contents.

```
<launch>
    <!-- Create 3 publishers with sequential names. -->
    <node
        name="average_publisher_1"
        pkg="ros_examples"
        type="average_publisher"
    />
    <node
        name="average_publisher_2"
        pkg="ros_examples"
        type="average_publisher"
    />
    <node
        name="average_publisher_3"
        pkg="ros_examples"
        type="average_publisher"
    />

    <!-- Create 3 subscribers with sequential names. -->
    <node
        name="average_subscriber_1"
        pkg="ros_examples"
        type="average_subscriber"
    />
    <node
        name="average_subscriber_2"
        pkg="ros_examples"
        type="average_subscriber"
    />
    <node
        name="average_subscriber_3"
        pkg="ros_examples"
        type="average_subscriber"
    />
</launch>
```

Note that the *type* argument is set to average_publisher/average_subscriber because that is what the node types (executables) were named in our *CMakeLists.txt*. To use this launch file, we can use the *roslaunch* command below.

```
roslaunch khepera average_calculator.launch
```

The ROS node graph in rqt should now look like the figure below.

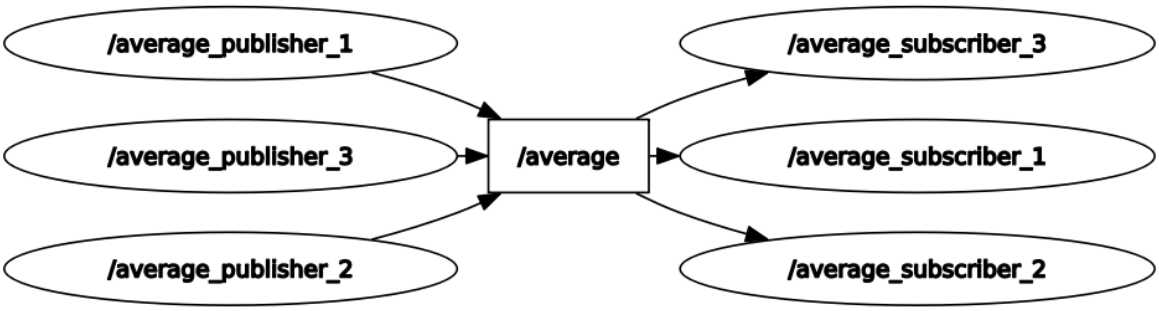

Figure : Multiple Nodes Publishing and Subscribing to the Same Topic.

We can also view the console output in rqt or echo the */rosout* topic and confirm that the subscribers are printing the averages. The user can now employ launch files to run multiple nodes and the roscore at once. More detailed tutorials are available here.

Summary

In this section, we described how to install ROS 1 Noetic. Afterwards, we showed the reader how to create a simple ROS package and its node within a catkin workspace, then we raised the complexity by developing a publisher and subscriber. Finally, we showed how to simplify launching several nodes at once using *roslaunch*. These instructions also gave reference links to detailed information about the tools we were using.

Lastly, I suggest *A Gentle Introduction to ROS* by Jason O'Kane as a good guide to understanding a lot of the components of ROS in greater detail.