ORB_SLAM2 with Webots Setup Documentation

V1.1

By Noah Grzywacz

August 20, 2021

Table of Contents:

1. Introduction and Assumptions:

This documentation is meant to show how to set up the ORB_SLAM2 library to work on sequences of images generated from the Webots robotics simulator. This section introduces the documentation by explaining what ORB_SLAM2 and Webots are, some assumptions about the user's knowledge and what they should be comfortable with before going through this documentation or extending it. Lastly, this documentation has some notes about steps for extending this system.

1.1 What is ORB_SLAM2?

ORB_SLAM2 is an open-source SLAM library developed by a team in Spain, led by Raul Mur-Artal, that supports real-time localization and mapping. The library supports Monocular, Stereo, and RGB-D cameras and computes camera trajectory as well as a reconstruction of the 3D environment. The library uses camera images in real time to detect loops and re-localize the camera while visually mapping the environment.

1.2 What is Webots?

Webots is an open-source robotics simulator created by Cyberbotics that allows the creation, development, and test of various robotic systems. This open-source software is free to download, and easy to set up thanks to the extensive documentation Cyberbotics provides. Webots provides an easy-to-use interface for developing robot controllers and testing these controllers through simulation as well as providing lots of additional tools and information for the purposes of debugging and documenting any project.

1.3 Assumptions and Prerequisites

There are several prerequisite topics that the user is assumed to have at least an introductory knowledge in before working their way through this documentation or extending it. Knowledge in the following topics is recommended, if not required, to fully understand the integration of ORB_SLAM2 with Webots and the (eventual) extension of the system:

Programming with Python – This system uses Python as the language for controlling the robot as well as the scripts necessary for generating the data that ORB needs. Therefore, a firm grasp in programming with python will help the user understand the syntax and styling of the controller and how to set up ORB for the execution of the system.

Robotic Kinematics – Because the robot requires a controller to be able to move, it is useful for the user to understand some of the basic kinematics of robots. The robot used in the development of the system described by this documentation is a two-wheeled robot, so is easy enough to learn to control with a basic understanding of robot kinematics. Regardless, the user needs to have at least some understanding and experience with how robots are controlled.

Linux Bash Environment – Setting up and running this system requires lots of shell writing. There is nothing as extensive as writing shell scripts, but the user should be comfortable with actions in the shell such as moving around, executing commands, and debugging are useful and necessary in the installation, setup, and running of this system.

ROS2 – Robot Operating System (ROS) is a set of libraries and tools that aid robotics engineers in developing and testing their systems. ROS2 is the second generation of ROS that includes many new features as well as better performance than the previous ROS1 generation. In the system described by this documentation, ROS2 will be used to run and control the robot from outside the Webots simulator. This makes extension of this system much easier as the original intent of this system was to have the robot-generated images from Webots transferred to the running ORB-SLAM2 library through the ROS2 network in real-time. Unfortunately, at the time of this documentation, ROS2 has some bugs in its network transfer protocol that the author was not able to amend. The Github server that this documentation belongs to should have additional documentation on Webots with ROS2 to allow the user to understand ROS2 as well as learn a little about Webots.

1.4 Extending the System and Documentation Notes

As mentioned in the previous section, the original vision for this system is unfinished because of current (at the time of writing this) issues with ROS2 as well as limitations with Webots. The system described by this documentation is intended to be extended to fully realize the original vision of transferring robot images and data from Webots to the ORB-SLAM2 library in real-time through the ROS2 network to localize and map the environment. This documentation provides information on how to set up the system to achieve this in part so that once the limitations from ROS2 and Webots no longer exist, finalization is simple.

Because the original system was impossible at the time, the system described by this documentation involves simulating the robot in Webots with ROS2 while exporting the robot-generated images and other sensor data to files outside of the simulator. Afterwards, the ORB-SLAM2 library is run on those files. That is, the system is a non-real-time system that runs the robot simulation to create a dataset then has the SLAM library localize and map the environment from the dataset.

Although this system is not the original intention, its lays the foundation for the final product once the following issues have been resolved:

ROS2 Foxy RTPS Issues – The ROS2 distribution used for this system is the Foxy release. Within the current release of this distribution (as well as others, e.g., Galactic), there are issues with the RTPS memory transfer protocol used by the ROS2 network. RTPS, or Real-Time Publish Subscribe protocol, is the middleware transfer protocol that ROS2 uses to communicate within its network. As of writing this documentation, there are some issues with the protocol that are not allowing ROS2 nodes to communicate with one another or transfer messages. This makes sending the image and sensor data from Webots to the ORB library impossible at the moment.

Webots Camera Limitations – This is the main issue with the execution of the system. The camera device provided by Webots that is attached to the robot is severely limited by both the image size and the resolution of the image. As a result of this, the current system can run the ORB_SLAM2 algorithm on the dataset generated by Webots, but unable to localize and map because it simply isn’t getting enough data from the images to perform visual SLAM. Thus, until a higher resolution camera device is added to Webots, the efficacy of Webots with ORB-SLAM2 is limited. This issue may also arise due to the camera distortion parameters not being configured correctly, however, I was unable to find the right parameters at the time.

2. Installing ORB_SLAM2:

This section discusses the installation of ORB_SLAM2 on a Linux Ubuntu operating system. Because of this, it is assumed that the user already has either a Linux virtual machine or some sort of dual boot set up to run the Linux operating system. The Github server this documentation belongs to also contains documentation on how to set up a Linux VM if the user does not currently have the operating system set up.

The ORB_SLAM2 library has been officially tested by the developers on Ubuntu systems 12.04, 14.04, and 16.04. The system implemented during the development of this documentation took place on a Linux Ubuntu 20.04 virtual machine. The library was still able to compile and run after testing many of the provided datasets that come with the ORB_SLAM2 library, so it is safe to assume this version should be safe for the reader to use.

The ORB_SLAM2 library Github repository can be found here. The prerequisite software for the base ORB_SLAM2 library is found in section two of the README file. However, note that the system in this documentation also includes the use of Webots and ROS2. For installation instructions for that software, please refer to the Webots ROS2 Documentation included in the Github server.

When installing new software, it is important to install the dependencies in order and always good practice to test them intermittently and verify that they are properly installed. The base ORB_SLAM2 library has four required dependencies along with the library itself. This section will go through each of those dependencies in order. Keep in mind that many of these dependencies also contain sub dependencies, which should be installed before installing the higher-level dependencies. Installation of the sub dependencies is not covered, but each dependency provides ample documentation for installing sub dependencies.

2.1 Installing C++ Compiler

A C++ compiler is the first tool that ORB_SLAM2 will need to work properly. This is because the ORB_SLAM2 library is primarily in C++ and thus it needs the C++ compiler to build correctly. Luckily, most Ubuntu systems come with a C++ compiler already installed. To check if the compiler is installed you can run the following command in a shell window:

```
g++ --version
```

This command checks the version of the C++ compiler, called G++, that typically is included with each Ubuntu system. If you don’t see a version response pop up such as…

```
g++ (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0
```

then you can install the G++ compiler by running the command

```
sudo apt install build-essential
```

and providing your system password. After the installation is complete, you can run the first command again to test if the compiler has been installed. Though it isn’t shown here, to fully test that the compiler is working correctly, you can create a quick “Hello World!” C++ script, compile and run it to verify that the compiler is working correctly.

2.2 Installing Pangolin

The Pangolin library is found here and is used for managing display and interaction for better video I/O without sacrificing performance. Pangolin is one of the ORB dependencies that has multiple sub dependencies, so be sure to install those first using the commands they provide on the repository page before cloning and building the Pangolin source code itself.

Once you have all the sub dependencies installed, you'll need to clone the Pangolin code and build it. When developing software systems that use multiple different subsystems, it is typically best to put each subsystem into a single workspace to have everything for the overall system in one place. You can create a simple workspace by simply creating a folder in your home directory. For example, if you wanted to create a workspace called `dev_ws` in your home folder, you can use the following command in your home directory.

```
mkdir -p dev_ws
```

This will create a new folder called `dev_ws` in your home directory where you can download and install all the dependencies. But first, because there are two different parts to the Webots ORB system (Webots w/ ROS2 subsystem and ORB_SLAM2 subsystem), create one more subdirectory called `orb_ws` where the ORB_SLAM2 library will reside. If you have not already installed Webots to work with ROS2, you can create another subdirectory called `webots_ws` to store your Webots_ROS2 packages.

Note: If you have built ROS2 from the source binaries, it should not go in the `webots_ws`. Those binaries should be in a separate location. The only packages that should be in the `webots_ws` are the default webots_ros2 packages and the packages you will create. The exact repository commit used was `8d72ddae2bac5e6d2696ee43f8152c15429d8bba`. Refer to the Webots ROS2 Documentation for more information.

In a shell window, move into the `orb_ws` you just created and execute the following command.

```
git clone https://github.com/stevenlovegrove/Pangolin.git
```

This will clone the Pangolin source code into the `orb_ws`. If you wish to create this exact project, the exact commit hash was `dd801d244db3a8e27b7fe8020cd751404aa818fd`. To build the Pangolin source code without the optional Doxygen dependency installed, move into the Pangolin folder that you just cloned, create a `build` directory, and move into that `build` directory. Then, run the following commands

```
cmake ..
```

```
cmake --build .
```

which will build the Pangolin source code. If you installed Doxygen or run into any issues with the installation, refer to the repo for resources to resolve the issues.

2.3 Installing OpenCV

The OpenCV library is used to manipulate images and features for computer vision systems. To install OpenCV, go here. The original ORB_SLAM2 library was tested on OpenCV versions 2.4 and 3.2, but the system in this documentation worked with version 3.4 as well.

Clone the OpenCV repo into your newly created `orb_ws` using the following command in a shell window at the `orb_ws`:

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

If you wish to recreate this exact project, the exact commit used is `828304d587b3a204ebc34ce8573c3adec5a41ad2.` Then move into the `opencv` directory you just cloned and switch to the version 3.4 branch using the command:

```
git checkout 3.4
```

With that, you have OpenCV 3.4 set up and ready to use with the ORB_SLAM2 library.

2.4 Installing Eigen

Eigen is a software package library dedicated to linear algebra algorithms that help in ORB's localization and mapping functionality.

To install Eigen, visit the gitlab repo here. ORB_SLAM2 requires at least version 3.1 for proper execution. The version used for the system in this documentation is 3.3. Start by cloning the repo into the `orb_ws` using the command:

```
git clone git@gitlab.com:libeigen/eigen.git
```

Again, you can switch branches once the repo is cloned using the `git checkout` command to select which version you wish to use. The exact commit used for this project was `4ad30a73fc8b707bf90b9ec712a12a7f63ae823a.` Remember that a minimum version of 3.1 is required for ORB_SLAM2.

With all of ORB_SLAM2's dependencies installed, you can move on to installing and building the ORB_SLAM2 library.

2.6 Installing and Building ORB_SLAM2

Now that all the dependencies are installed, you need to install and build the ORB_SLAM2 library. Start by cloning the repo from here into the `orb_ws` using the following command:

```
git clone https://github.com/raulmur/ORB_SLAM2.git ORB_SLAM2
```

This will clone the ORB_SLAM2 source code into a directory labelled `ORB_SLAM2`. The exact commit used was `f2e6f51cdc8d067655d90a78c06261378e07e8f3`. Now you need to build the library using the following commands. First move into the `ORB_SLAM2` directory using the command:

```
cd ORB_SLAM2
```

Then modify the `build.sh` script to be executable using the command:

```
chmod +x build.sh
```

Lastly, build the library by running the build script using the command:

```
./build.sh
```

You should now have the ORB_SLAM2 library fully installed and set up. If you wish, you can test the library by downloading one of the datasets linked in the ORB repo and running the appropriate commands depending on the dataset.

3. Webots with ROS2 Controller:

This section is dedicated to setting up the ROS2 package controller to control the robot. As it is assumed that the user already has experience or knowledge with ROS2 and robot controllers, this section will not go in depth on how the controller is set up and works. Rather, it will cover the parts of the controller that are necessary for creating the dataset that the ORB_SLAM2 library needs to perform its SLAM algorithm. For more in-depth info on how to set up a Webots controller that uses ROS2, refer to the Custom Webots ROS2 Controller documentation on the Github server, which will be referenced many times throughout this section.

3.1 Controller and Dataset Overview

The controller in this system itself is a simplified version of the controller implemented in the Custom Webots ROS2 Controller documentation. This controller simply moves the robot around the environment while avoiding obstacles, hence the name of the controller package `wanderer`. The behavior of the robot was assigned this way to provide unique sensor and image data to the ORB_SLAM2 library. The controller and therefore the behavior of the robot can surely be made to be more sophisticated. However, in its current state, the controller sufficiently accomplishes the goal of moving the robot around the environment while exporting the appropriate images and data from the robot.

For the ORB_SLAM2 library to run on a dataset, the dataset must contain files and directories associated with the following data:

RGB Images – These are the color images gathered by the robot's camera which must be saved to a directory named `rgb` with each image named after the timestamp in which it was captured.

Depth Images – These are the depth images gathered by the robot's camera or depth camera which must be saved to a directory named `depth` with each image named after the timestamp in which it was captured.

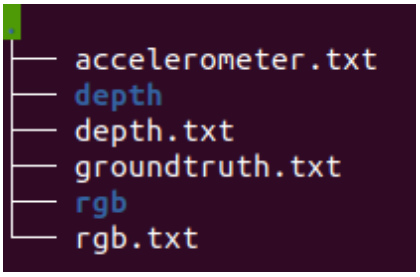
Ground-truth Data – This is the transform data that is gathered from the robot as it moves throughout the environment that is saved into `groundtruth.txt`. Transform data is generated by the robots positional and rotational movements based on its origin point.

Accelerometer Data – This is the acceleration data that is gathered from the robot as it moves throughout the environment that is saved into `accelerometer.txt`. Acceleration data is generated based on the vectored movements of the robot based on its origin point.

RGB/Depth Associative Data – These are the files that contain the associations between the timestamp, the RGB images, and the Depth images that ORB_SLAM2 needs to run. This data is saved into the `rgb.txt`, `depth.txt`, and `associations.txt`.

Note: The file `associations.txt` is required for the RGB-D mode that this system will run the ORB_SLAM2 library in. The Monocular mode is also usable, but requires further development, as in testing it was not working properly with the generated dataset.

The final sequence directory tree is shown in the image below.



3.2 Generating the Dataset

Besides the behavioral changes to the base controller, there were a few other changes that needed to be made to export the data that the ORB_SLAM2 library needed. These changes include the following, which will be covered more extensively in the following subsections:

1.) Changing the Webots robot node in the ROS2 network from a simple WebotsNode to a WebotsDifferentialDriveNode to create transform data.

2.) Adding a subscriber to read from the `/tf` transform topic generated by the WebotsDifferentialDriverNode to read transform data.

3.) Adding both Rangefinder and Accelerometer sensors to the robot in Webots to obtain the depth images and acceleration data, respectively, from the robot.

4.) Writing the data gathered by the sensors and cameras to the dataset appropriately.

3.2.1 Converting from WebotsNode to WebotsDifferentialDriverNode

The original controller described in the Custom Webots ROS2 Controller documentation launched the Webots node, containing the robot, as a simple WebotsNode. While this type of node is sufficient for simple robotic systems and controllers, it does not provide the transformation data necessary to localize and map. Thus, it is necessary that the system launches the Webots node as a WebotsDifferentialDriveNode in `slave.py` as shown below.

Base Controller Webots Node Definition:

```
class ServiceNodeVelocity(WebotsNode):
    def __init__(self, args):
        super().__init__('slave_node', args)
```

Updated Controller Webots Node Definition:

```
class ServiceNodeVelocity(WebotsDifferentialDriveNode):
    def __init__(self, args):
        super().__init__('slave_node', args,
            wheel_distance=DEFAULT_WHEEL_DISTANCE,
            wheel_radius=DEFAULT_WHEEL_RADIUS)
```

With this bulkier node type, additional data such as transform data is published to the ROS2 network by default, which is necessary to write to the `groundtruth.txt` file.

3.2.2 Subscribing to Transform Topic

The previous subsection discussed modifying the node type to publish the robot's transform data into the ROS2 network. Thus, this subsection describes the creation of a subscriber to read the transform data from the network that needs to be written to the `groundtruth.txt` file.

Like other subscribers in ROS2, the subscriber is initialized as shown below.

```
self.transform_subscriber = self.create_subscription(
    TFMessage, 'tf', self.transform_callback, 1
)
```

This subscriber reads the `tf` transform topic data as a `TFMessage` through the `transform_callback` function into the `trans` variable which is converted to a `TransformStamped` type variable and finally to a `Transform` type variable as shown.

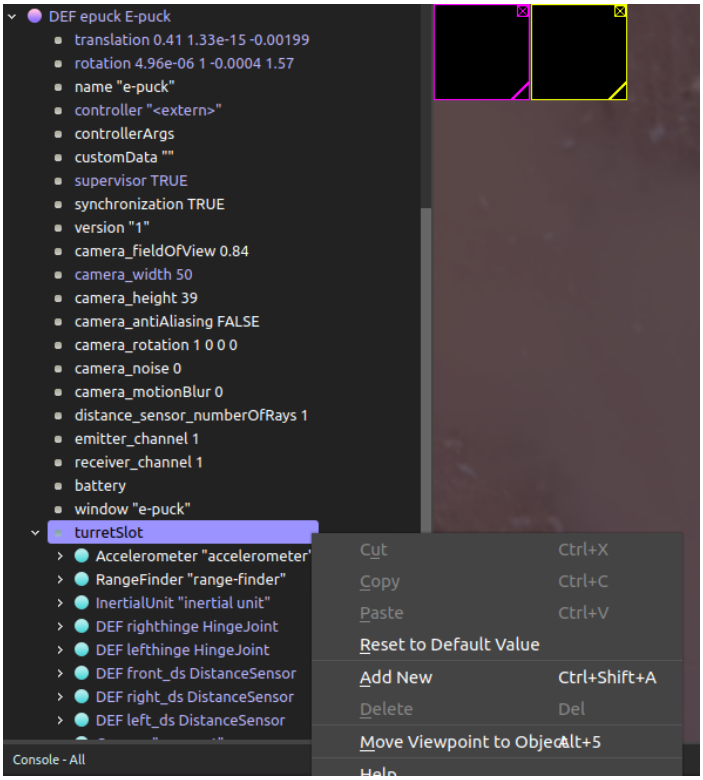
```
def transform_callback(self, msg):
    self.trans = msg.transforms
    self.meta_trans = self.trans[0]
    self.transform = self.meta_trans.transform
```

The data from this message is then extracted to a `translation` and `rotation` variable where it is written into the `groundtruth.txt` file with the timestamp of the system.

(File writing code segments shown in section 3.2.4)

3.2.3 Gathering Depth Images and Accelerometer Data

As the ORB_SLAM2 library requires the depth images and acceleration data to run properly, sensors that read those types of data must be added to the robot in Webots. This can be done by selecting and right clicking the robot's turretSlot component in the Webots Scene Tree hierarchy. Then, clicking on the "Add New" button pulls up a window in which you can add the Accelerometer and RangeFinder devices found under "Base nodes," as shown on the next page.



Once these devices are added to the robot in Webots, they must be instantiated and enabled in the ROS2 controller. This is done the same way as any other device in the Webots ROS2 integration:

```
#depth camera
self.depth_cam = self.robot.getDevice('range-finder')
self.depth_cam.enable(self.timestep)
self.depth_publisher = self.create_publisher(Image, 'depth', 1)

self.get_logger().info('Enabled depth camera')

#add accelerometer sensor here
self.accelerometer = self.robot.getDevice('accelerometer')
self.accelerometer.enable(self.timestep)
self.accel_publisher = self.create_publisher(Vector3, 'accel', 1)

self.get_logger().info('Enabled accelerometer')
```

Lastly, the image and sensor data are collected in the `sensor_callback` function where the depth image is obtained from the depth camera device with the data type set as `buffer` to allow for the correct image encoding. The Accelerometer data is obtained and saved into a list `ac` which is then used to assign an `acceleration` variable of type `Vector3`. Thus, the depth image and accelerometer data are gathered successfully and ready to export to the dataset.

3.2.4 Writing Gathered Data

With all the necessary data gathered in the controller, we still need a way to export it to the dataset that ORB_SLAM2 will use for localization and mapping.

We start by saving both the RGB camera image and the depth RangeFinder image to the respective `rgb` and `depth` directories with names corresponding to the timestamp they were captured at, as shown below. If the images are not saved correctly, the system prints an error message.

```
timestamp = time.time()

#try saving images here then run ORB on image dataset
if(self.camera.saveImage(f"rgb/{timestamp:.7f}.png", 100) == -1 or
   self.depth_cam.saveImage(f"depth/{timestamp:.7f}.png", 100) == -1):
    self.get_logger().info("Issue capturing images")
    return
```

We also write the acceleration data associated with the timestamp `timestamp` to `accelerometer.txt` as shown below.

```
f = open("/home/noah/Webots/sequence/accelerometer.txt", 'a')
f.write("\n{:.7f} {:.7f} {:.7f} {:.7f}".format(timestamp, ac[0], ac[1], ac[2]))
f.close()
```

The ground truth data gathered from the `tf` topic subscriber and processed is then written to `groundtruth.txt` as shown.

```
self.translation = self.transform.translation
self.rotation = self.transform.rotation

f = open("/home/noah/Webots/sequence/groundtruth.txt", 'a')
f.write("\n{:.7f} {:.7f} {:.7f} {:.7f} {:.7f} {:.7f} {:.7f} {:.7f}".format(
    timestamp,
    self.translation.x * 39.3701,
    self.translation.y * 39.3701,
    self.translation.z * 39.3701,
    self.rotation.x,
    self.rotation.y,
    self.rotation.z,
    self.rotation.w
))
f.close()
```

Lastly, the associative data for the RGB and depth images are written to `rgb.txt` and `depth.txt` using a simple separate script after the simulation has stopped running. The script `populate.py` is

shown below and simply takes the names from each of the image directories and the timestamp and fills out `rgb.txt` and `depth.txt` with two text columns.

```python
from os import walk, listdir
from os.path import isfile, join, splitext

path = "/home/noah/Webots/sequence"
rgb_path = path + "/rgb"
depth_path = path + "/depth"

file_rgb = open("rgb.txt", "a")

for f in listdir(rgb_path):
    if isfile(join(rgb_path, f)):
        n = splitext(f)[0]
        file_rgb.write("{} rgb/{}\n".format(n,f))

file_rgb.close()

file_depth = open("depth.txt", "a")

for f in listdir(depth_path):
    if isfile(join(depth_path, f)):
        n = splitext(f)[0]
        file_depth.write("{} depth/{}\n".format(n, f))

file_depth.close()
```

The file `associations.txt` is required for running ORB_SLAM2 in the RGB-D configuration as was intended by the creation of this system and dataset. This file is generated using the `associate.py` script which can be downloaded from here. The original ORB_SLAM2 repository shows how to use this script to generate the file.

Finally, with the robot dataset generated, it's time to run the ORB_SLAM2 system.

4. Running ORB_SLAM2:

The original ORB_SLAM2 repository describes how to run the ORB_SLAM2 library in the RGB-D configuration with the following command for the provided datasets.

```
./Examples/RGB-D/rgbd_tum Vocabulary/ORBvoc.txt Examples/RGB-D/TUMX.yaml PATH_TO_SEQUENCE_FOLDER ASSOCIATIONS_FILE
```

This command can be modified, while retaining the same format, to process generated sequences that don't belong to the provided datasets. This section will show how to run ORB_SLAM2 using a modified version of this command.

To preface, the dataset that was generated over the previous sections follows the dataset format of the provided TUM datasets. Thus, it follows that we run ORB_SLAM2 with some of the same configuration settings as those datasets.

For example, the executable shared directory `rgbd_tum`, while created for the provided TUM datasets, also works for datasets that don't belong to the pre-generated ones referenced in the ORB_SLAM2 repository. Thus, we can use the same executable shared directory. If you wish to have a separate executable, however, you can simply duplicate it and rename the duplicate.

Like the shared directory executable, the `ORBvoc.txt` file argument is the same for all datasets run with the ORB_SLAM2 library. Thus, that argument remains the same no matter the sequence.

The `TUMX.yaml` file is a file that is unique to the camera generates the images that ORB_SLAM2 is run on. To initially run the ORB_SLAM2 library with the system described in this documentation, there are a few settings within this file that must be modified. This includes the reduction of the "ORB Parameters" to better adapt the ORB_SLAM2 system to the robot camera, as shown below.

```
#--------------------------------------------------------------------------------------------
# ORB Parameters
#--------------------------------------------------------------------------------------------

# ORB Extractor: Number of features per image
ORBextractor.nFeatures: 200 #1000

# ORB Extractor: Scale factor between levels in the scale pyramid
ORBextractor.scaleFactor: 1 #1.2

# ORB Extractor: Number of levels in the scale pyramid
ORBextractor.nLevels: 3 #8

# ORB Extractor: Fast threshold
# Image is divided in a grid. At each cell FAST are extracted imposing a minimum response.
# Firstly we impose iniThFAST. If no corners are detected we impose a lower value minThFAST
# You can lower these values if your images have low contrast
ORBextractor.iniThFAST: 5 #20
ORBextractor.minThFAST: 1  #7
```

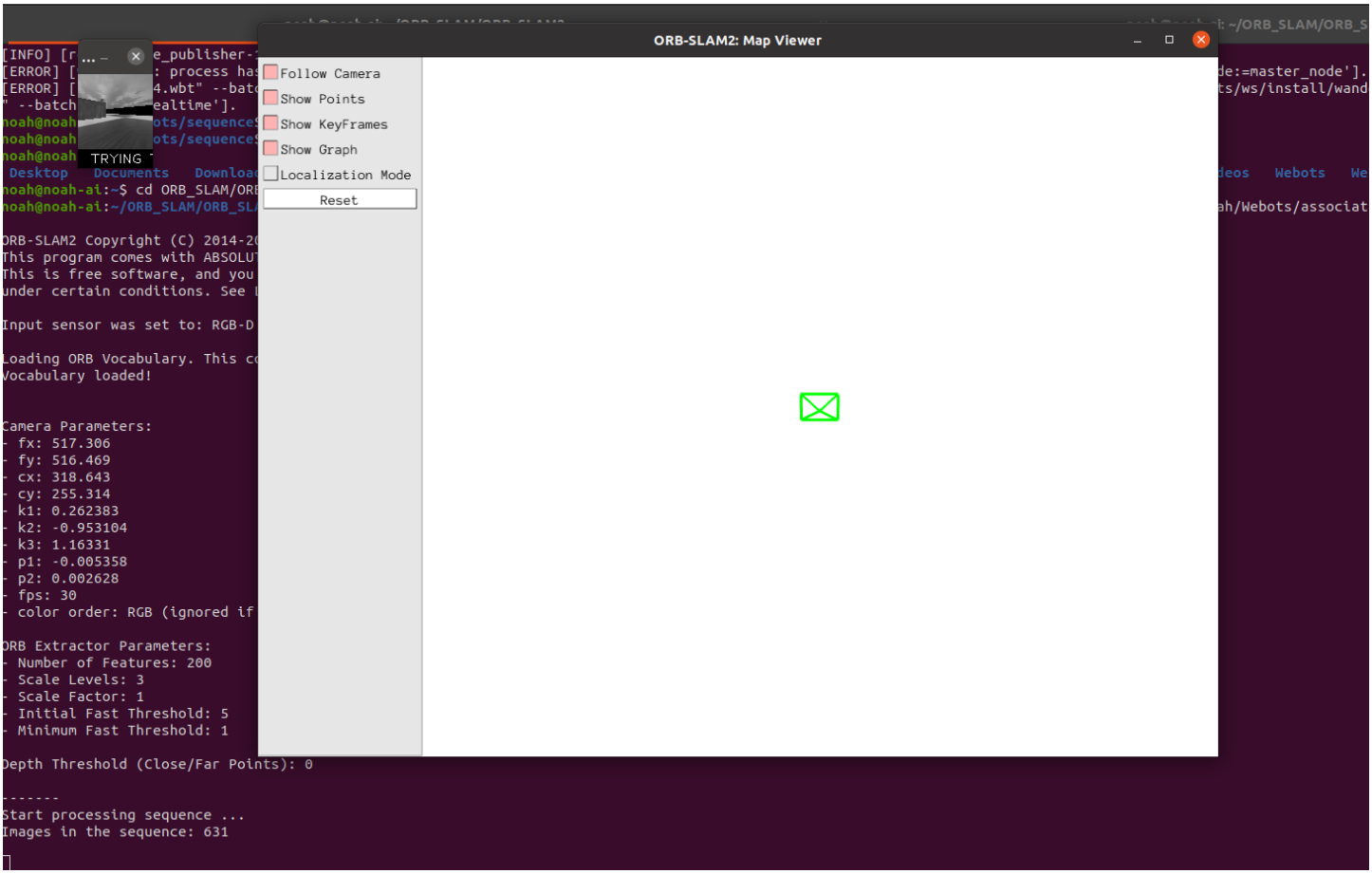
However, as mentioned at the beginning of this documentation, the ORB_SLAM2 library is unable to localize and map with the dataset described by the documentation despite the system being able to run. This is because of the limitations of the camera. While the camera calibration settings provided in the `CAM.yaml` file are correct, the correct values of the distortion parameters including `k1`, `k2`, `p1`,

p2, and k3 are currently unknown. More on this, however, in the last section of this documentation. The full version of this file, CAM.yaml, can be downloaded from the Github server.

The PATH_TO_SEQUENCE_FOLDER argument corresponds to the directory containing the generated dataset files listed in section 3.1 that the previous sections discuss.

Lastly, the ASSOCIATIONS_FILE argument corresponds to the generated associations.txt file that we used the associate.py script to generate following the format described in the ORB_SLAM2 repository.

You should now be able to run the ORB_SLAM2 library on the dataset you have generated. The program running should look like the image below where you see the images being iterated through in the top left with the ORB_SLAM2 library viewer ready to localize and map based on the captured images.



However, although the system is running, it is not working as wanted or intended. If you tested with some of the TUM datasets linked in the ORB_SLAM2 repository, you will know that while the images are being iterated through, the system plots small green points on the images that it saves and uses to track the robot's movement. However, as shown in the image viewer in the image above, none of these points exist on the robot-generated images. This is the point of contention that needs to be further investigated to allow the system to eventually be further extended.

5. Notes on Extension:

This section discusses steps that need to be taken to realize the original vision of the system described at the beginning of this documentation. That is, to have the Webots simulation running at the same time as the ORB_SLAM2 library and feeding images and sensor data to the ORB_SLAM2 library in real-time through the ROS2 network. For this system to be fully accomplished, there are a couple steps necessary to complete:

1.) Appending the existing non-real-time system described by this documentation to have ORB_SLAM2 correctly localize and map based on a Webots with ROS2 generated dataset – As this system currently is, ORB_SLAM2 is unable to localize and map because tracking points are not being generated on the dataset images. This may be happening for a couple reasons:

a) The Camera node in Webots doesn't provide a distinct enough image with a high enough resolution for ORB_SLAM2 to recognize significant features and discrepancies between objects and obstacles in the environment. From research and troubleshooting online, the camera provided by Webots seems to be the lowest resolution camera that anyone has tried to run the ORB_SLAM2 library with. The E-puck camera generates an image with a maximum resolution of 640x480 which is a much smaller resolution than any other dataset image found. Thus, it is believed that ORB_SLAM2 has a minimum image resolution that the library can work on which would require Webots to include a higher resolution camera to its software for the Webots/ORB_SLAM2 system to work.
b) The second possible issue is that the `CAM.yaml` camera settings file is simply not configured correctly. Specifically, the distortion parameters and ORB parameters are not set to the appropriate values. As an authors note, I searched everywhere, short of emailing the camera manufacturer, trying to find the camera distortion settings to correctly modify the camera settings file with no luck. Perhaps, with proper configuration of the E-puck camera, the system will work. Though, even without proper camera settings, the ORB_SLAM2 library should still track a couple distinct points on the image. Thus, the primary suspect of the SLAM issue is the low camera resolution.

Once the issue with the non-real-time system is properly diagnosed and amended, the system could be extended to fully realize the real-time Webots-ORB_SLAM2 communication through ROS2 localization and mapping.

2.) Integration of Webots and ORB_SLAM2 with ROS2 in real-time – This step already contains support through the creation of the Webots ROS2 controller discussed in this documentation as well as the ROS2 ORB_SLAM2 wrapper repository found here. As you may have noticed in the original ORB_SLAM2 repository, ORB already supports ROS. However, the ROS mentioned there is ROS1. The ROS2 ORB_SLAM2 wrapper repo linked above is essentially a ROS2 version of the ROS1 system converted by Alberto Soragna. With both the ROS2/ORB_SLAM2 system and ROS2/Webots system already setup, combining the two to have them communicate over the ROS2 network shouldn't be too difficult. However, this also

requires the ROS2 RTPS middleware issue, mentioned at the beginning of this documentation, to be patched.

6. Video:

The video linked below shows the system at work as well as commentary on setup and improvements that can (and should) be made in the future.

Link: https://youtu.be/uz_wd_5OWr4