MRT Electronics and Communications - Task 2
ArUco Codes

Shridhar Patil

October 2024

1.1 Introduction

A server in python with custom message types is created which can:

- Scan all ArUco codes present inside an image.
- Scan all ArUco codes in a video frame by frame.
- Scan ArUco codes through webcam/camera input.

To do this we create two packages, arucosrvmsg, which defines the custom message formats and can be easily imported into the server package, arucoserver.

The arucoserver package contains a service node and client nodes for the different input types. It also contains launch files in the /launch directory to start the service and client both at once.

1.2 Procedure

1.2.1 arucosrvmsg

This package contains the .srv files which defines the request-response format of the service created between nodes.

$$\textbf{ros2 pkg create –build-type ament\_cmake arucosrvmsg} \tag{1.1}$$

This must be a cmake file. A directory /srv is created inside the package directory to house the .srv file.

Dependencies:

The custom .srv request response message requires certain dependencies on other packages which make it able to send images and bounding box coordinates.

- sensor_msgs: This package contains a Image.msg message type which allows sending images as a request. The Image.msg type contains various attributes which describe any requested image.
- geometry_msgs: This package is required to send the coordinates of the bounding boxes of the ArUco codes in a structured manner.
  It contains a Polygon.msg file which is a array of points in 3D space (For 2D coordinates, z is set to 0).
  (Polygon.msg: polygon_msgs/Point32[ ] points)
- cv_bridge and OpenCV: cv_bridge and OpenCV are required for the detection of the ArUco codes. cv_bridge is required to convert the sensor_msgs/Images image to OpenCV cv::Mat type image since both use different image storing techniques.
- rosidl_default_generators: Package required to create interface code (code which helps which different parts of ROS2 to communicate with services and messages) for the .srv files created in the package.

Both these dependencies are added by editing the CMakeLists.txt file and the packages.xml file of the arucosrvmsg package.

CMakeLists.txt:

find_package(<package name> REQUIRED) is added for each new package required as a dependency.

```
# Find dependencies
find_package(ament_cmake REQUIRED)
find_package(geometry_msgs REQUIRED)
find_package(sensor_msgs REQUIRED)
find_package(rosidl_default_generators REQUIRED)
find_package(cv_bridge REQUIRED)
find_package(OpenCV REQUIRED)
```

Figure 1.1: Finding packages and including them.

For rosisl_generate_interfaces to generate the interface code for .msg and .srv files, the following code is added:

```
# Generate ROS interfaces (messages and services)
rosidl_generate_interfaces(${PROJECT_NAME}
  "msg/ArucoMarkers.msg"
  "srv/ArucoMarkers.srv"
  DEPENDENCIES sensor_msgs geometry_msgs
)
```

Figure 1.2: Locating .srv and .msg files for package to generate interface code for them.

For OpenCV packages, it is nessesary to locate directories where compiler should look for header file definitions when compiling code. This is done with the code:

```
# Include directories (necessary for using OpenCV and cv_bridge)
include_directories(
  ${OpenCV_INCLUDE_DIRS}
  ${cv_bridge_INCLUDE_DIRS}
)
```

Figure 1.3: The directory where header files of OpenCV are defined are located.

package.xml:

The dependencies on packages are stated in package.xml file as well. There are different ways of adding dependency:

- <build_depend> : dependency only required at buildtime with colcon build.
- <exec_build> : dependency required only at execution time/runtime.
- <depend> : dependency required both at build and execution time (This includes both build_depend and exec_depend)

Packages found in CMakeLists.txt are stated in package.xml file again with respective dependency format.

```
<buildtool_depend>ament_cmake</buildtool_depend>
<build_depend>rosidl_default_generators</build_depend>
<depend>sensor_msgs</depend>
<depend>geometry_msgs</depend>
<depend>cv_bridge</depend>
<depend>opencv4</depend>
```

Figure 1.4: Dependencies added.

Creation of srv file:

In the /srv directory, a ArucoMarkers.srv file is created. This file will contain the request and response data types for client service interaction. The request and response are seperated with "—".

```
# Request
sensor_msgs/Image image

---
# Response
int32[] ids
geometry_msgs/Polygon[] bounding_boxes
```

Figure 1.5: ArucoMarkers.srv

- Request: sensor_msgs/Image is a .msg file. An Image variable of this message type is sent as a request.
- Response: What we get back is a int32 array of bounding box id's of the ArUco codes and an array of type geometry_msgs/Polygon[]. This array is basically a collection of polygons each of which in itself is a collection of float64 x,y,z points which are the corners of the polygon.

This concludes the creation of package for srv file definition.

1.2.2 arucoserver:

This package includes the arucosrvmsg package and uses its defined srv files to send requests and recieve responses from service and client nodes defined inside it.

It is created inside the /src directory with the command:

ros2 pkg create –build-type ament_python arucoserver (1.2)

This is a ament_python type package (unlike arucosrvmsg which was required to be ament_cmake in order to define custom srv files).

Dependencies:

Dependencies inside ament_python type packages are defined under the package.xml file. This process is similar to how packages were included in the arucosrvmsg package. They are listed in the code with their dependency inclusion type (build/runtime or both).

- arucosrvmsg: Refers to package created earlier to source its code **and its dependencies**.

Other packages (OpenCV, geometry_msgs and sensor _msgs) are aldready a dependency of arucosrvmsg and hence do not require to be addressed here.

```

<depend>arucosrvmsg</depend>

```

Figure 1.6: arucosrvmsg dependency added.

Creation of service node:

Important: All nodes are created in the arucoserver/arucoserver directory.
A file service.py is created which houses our service node. The libraries imported are:

- rclpy: imported to make it possible to write ROS2 nodes and functions in python.
- sensor_msgs.msg import Image: From the .msg files of sensor_msgs, we import Image.msg which has all attributes to send an image as a message.
- arucosrvmsg.srv import ArucoMarkers: From the arucosrvmsg package we added a dependency to, search the .srv files and import ArucoMarkers.srv.
- geometry_msgs import Polygon, Point32: From the geomtery_msgs dependency (from arucosrvmsg), search the .msg files and import Polygon.msg and Point32.msg.
- OpenCV library imports: Since OpenCV and cv_bridge are dependencies in arucosrvmsg, they are accessable here. These header files are present in the directory located with include_directories() function in the CMakeLists.txt file of arucosrvmsg.

```
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Image
from arucosrvmsg.srv import ArucoMarkers
from geometry_msgs.msg import Polygon, Point32
from cv_bridge import CvBridge
import cv2
import cv2.aruco as aruco
```

Figure 1.7: service.py imports

rclpy allows us to create a node class which is derived from the Node base class of rclpy.

It is called 'aruco_marker_service' (These commands are inside the constructor hence only executed once per node).

A service is created using self.service = self.create_service(ArucoMarkers (.srv file), 'detect_aruco' (name of service), self.detect_aruco_callback (function executed when service is called))

self.bridge = CvBridge(). This code creates a bridge object of class CvBridge which can convert between sensor_msgs/Image files and OpenCV image file.

self.get_logger().info('Aruco Marker Service is ready'). This will fetch the get_logger() object and will send a message of 'info' level importance to the screen.
detect_aruco_callback(self, request, response): This function is called whenever the service is called (gets a request).

This function is passed variables of type request and response (as stated in .srv file). Hence, request has a variable of type sensor_msgs/Image called image and Response is of type geometry_msgs/Polygon[].

Inside this function, the bridge object of CvBridge() class is used to create a 'bgr8' format image from the request.image data.

A dictionary of of predefined aruco markers is loaded from the OpenCV library onto instance aruo_dict. This dictionary contains 250 unique 4x4 ArUco codes. (aruco_dict is an object of cv2.aruco_Dictionary type)

Here, parameters are a structure which define exactly how OpenCV will detect the ArUco codes. If the values in paramaters are changed, then the way/method in which the ArUco codes are detected can be altered. By default, parameters = aruco.DetectorParameters() returns the default parameters used.

corners, ids, ignored are 3 values assigned data each time aruco.detectMarkers() is executed. This function gets the cv_image, aruco_dict and the parameters on which detection occurs as inputs.

- corners: List of detected markers corners. Each marker is assigned its 4 points.
- ids: Array of integer ids of the ArUco markers detected in the scene. These are stored in a numpy array.
- ignored: All cases of ArUco codes which were incomplete are returned here. This case is ignored.

Addition of information to response:

response.ids = ids.flatten().tolist() : this function assigns the int32 array (int32[ ]) of ids detected the array of id's found. This must be assigned after some conversions.

`ids` must be flattened (converted from multidimensional NumPy array to single dimensional 1D array. This array is later converted to a list with .tolist() function since response.ids is a int32 1D array).

1.2.3 Appending the response bounding_box:

The corners response is a numpy array (3 dimensional). The variable marker_corners flows through the polygons (ArUco Code squares) which are elements of the corners structure. We create an instance of Polygon class called polygon (this is just an array of (x,y,z) positions to represent each point of the polygon).

We then create corners, a variable which flows through each point in marker_corners (the current polygon) and adds each points x and y coordinates onto a Point32 instance, which is later appended into polygon.

After each Polygon is done with in corners, the polygon instance is pushed into response.bounding_boxes.

In the end, response is returned back, and the cycle continues each time a new request is received.

main function:

- rclpy.init(args=args) : args allows us to communicate with ROS2 using command line inputs. args = args means all command line inputs are given to ROS.

```python
        if ids is not None:
            response.ids = ids.flatten().tolist()


            for marker_corners in corners:
                polygon = Polygon()
                for corner in marker_corners[0]:
                    point = Point32()
                    point.x = float(corner[0])  #x
                    point.y = float(corner[1])  #y
                    polygon.points.append(point)
                response.bounding_boxes.append(polygon)
```

Figure 1.8: Transferring data onto response.bounding_boxes.

- node = ArucoMarkerService() : This creates an instance of ArucoMarkerService class (derived from Node parent class). The ArucoMarkerService constructor executes and creates all required objects of the instance.
- rclpy.spin(node) : It makes ROS continuously search for requests to the service. Each time time server gets a request, the detect_aruco_callback() function of the node is executed. (Continiously searching for requests/information is called spinning a node).
- rclpy.shutdown(): It shuts down the node when service is not required anymore.

```python
def main(args=None):
    rclpy.init(args=args)
    node = ArucoMarkerService()
    rclpy.spin(node)
    rclpy.shutdown()
```

Figure 1.9: main() function of service node.

Detection with video:

A node is created which scans through a video sent as a request, creates jpg images of each frame in the video and passes these videos one by one into the service and displays output for each frame.

Creating frames from video:
cam = cv2.VideoCapture("File path")

This function starts the video capturing. cam is an instance of cv2.VideoCapture() class.

This is done with the createImagePath() function. It sources a video file from a predefined location. A data folder is created (if does not exist) or used (if aldready exits). This data folder is the place where all frame jpg images will be saved (and later deleted after all frames send as request).

$$\textbf{ret, frame} = \textbf{cam.read()} \tag{1.3}$$

- ret: This is a True or False boolean which returns True if a frame image was successfully created and False if frame image could not be created (due to no remaining frames).
- frame: This contains the actual image created of the frame of the video.

cv2.imwrite(frame,name) will save the image file as a jpg image at path with name passed through name.

currentframe is then incremented with one.

The next frame is then read into ret and frame with cam.read() again. This loop continues until ret = False (no more frames remaining).

cam.release() at the end frees all resources required for video processing.

send_all_images():

An ArucoMarkerClient() instance is created. For each iteration, a global variable i keeps track of number of frames passed and i < totalframes is condition till which images are sent to service.

A request is sent to 'detect_aruco' service with service message type ArucoMarkers (imported from arucosrvmsg package).

```
self.cli = self.create_client(ArucoMarkers, 'detect_aruco')
```

while rclpy.ok() (i.e, all ROS nodes are functioning properly), rclpy.spin_once(node) is executed. The current image is sent as request to the service (rclpy.send_request(image_path))

The response is in the form as defined in the ArucoMarkers srv file. The detected id's are displayed by logging the info while the bounding boxes are displayed with a flowing for loop through response.bounding_boxes.

After each image is sent and response is gotten, it is deleted from the save file.

```
    while i < currentframe:
        image_path = "/home/shridhar/Workspaces/mrt_ws/Task2/data/frame"+str(i)+".jpg"

        i += 1


        if not image_path:
            print("No images found in the directory.")
            return


        node.get_logger().info(f"Processing image: {image_path}")
        node.send_request(image_path)

        while rclpy.ok():
            rclpy.spin_once(node)
            if node.future.done():
                try:
                    response = node.future.result()
                    node.get_logger().info(f"Detected IDs: {response.ids}")
                    print("")
                    for bbox in response.bounding_boxes:
                        node.get_logger().info(f"Bounding Box: {bbox}")
                        print("")
                except Exception as e:
                    node.get_logger().info(f'Service call failed {e}')

                # Delete the image after processing
                try:
                    os.remove(image_path)

                except Exception as e:

                    print("",end='')
                break
```

Figure 1.10: Loop to send requests to service.

1.2.4 Entry points:

It is important to add the command name to be put in terminal to execute running each node and the entry point function of each node. This is done though the setup.py file of the package.

First all node python files are made executable with the following command:

$$\texttt{chmod +x <filename>.py} \tag{1.4}$$

In the setup.py file, under console_scripts, the following script is added for each node:

`<NodeName>=<package>.<pythonfilename>:<EntryPointFunction>` (1.5)

1.2.5 Creation of Launch Files:

It is easier to launch the service node and another client node through one code only. This done with help of launch files. These files are present in the src/arucoserver/launch directory. The launch files have a common syntax.

```python
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package='arucoserver',
            executable='service',
        ),
        Node(
            package='arucoserver',
            executable='webcam',
        ),
    ])

```

Figure 1.11: Launch File for Webcam Service Node.

For different nodes, different command name is added under "executable = " section (these command names are as those in the setup.py file for each python node).

`ros2 launch <launchfilename/path>` (1.6)

The launch file is executed with above command.