Introduction:

ROS (Robot Operating System) is a widely used open-source robotics framework that provides a collection of software libraries and tools for building robots. It was developed at Stanford University in 2007 and is now maintained by the Open Robotics organization. ROS provides a comprehensive set of features for building robotic systems, including hardware abstraction, message-passing between processes, and package management. ROS is used in a variety of applications, including industrial automation, autonomous vehicles, and drones.

Features of ROS:

ROS provides a wide range of features that make it a popular choice for building robotic systems. Some of the key features of ROS include:

Node-based architecture: ROS follows a distributed architecture where each component of the system is implemented as a node. Nodes can communicate with each other through a message-passing mechanism, making it easier to build complex systems.

Package management: ROS provides a package management system that makes it easy to share and reuse code. This helps developers save time and effort while building complex robotic systems.

Hardware abstraction: ROS provides a hardware abstraction layer that allows developers to write code that can run on different types of hardware platforms. This makes it easier to develop robotic systems that can be deployed on a variety of devices.

Visualization tools: ROS provides visualization tools that allow developers to visualize the behavior of their robotic systems. This helps in debugging and testing of the system.

Applications of ROS:

ROS has been used in a variety of applications, including industrial automation, autonomous vehicles, and drones. Some of the popular applications of ROS include:

Autonomous vehicles: ROS has been used in the development of autonomous vehicles, including self-driving cars, trucks, and drones. ROS provides a powerful set of tools for building autonomous systems, including perception, mapping, and navigation.

Industrial automation: ROS is widely used in industrial automation, including manufacturing and logistics. ROS provides a flexible and modular architecture that makes it easier to build custom automation systems.

Robotics research: ROS is a popular choice for robotics research, as it provides a comprehensive set of features for building robotic systems. ROS makes it easier to

experiment with different algorithms and techniques, making it an ideal platform for robotics research.

In short, ROS is a powerful open-source framework for building robotic systems. It provides a wide range of features, including hardware abstraction, message-passing, and package management, which make it easier to build complex robotic systems. ROS has been used in a variety of applications, including autonomous vehicles, industrial automation, and robotics research. With its wide range of features and flexible architecture, ROS is likely to continue to be a popular choice for building robotic systems in the future.

ROSCORE:

ROS (Robot Operating System) is a framework for building robotic applications. ROS provides a set of tools and libraries that allow developers to create complex and powerful robotic systems. One of the core components of ROS is *roscore*.

roscore is a command-line tool that serves as the central point of communication for all the nodes in a ROS system. It manages the communication between different nodes and provides the naming and registration services required for nodes to find and communicate with each other.

When you start *roscore*, it starts several processes that together make up the ROS master, which is responsible for maintaining a registry of all the active nodes in the system. The master node keeps track of the topics that nodes are publishing and subscribing to and manages the connections between the nodes.

roscore provides the following services:

Parameter server: The parameter server is a central repository for storing key-value pairs. It provides a way for nodes to share configuration parameters and other data.

Naming and registration: *roscore* provides a naming service that allows nodes to find each other by name. Nodes register with the ROS master when they start up, and this registration is used to manage the communication between nodes.

Topic management: *roscore* provides a mechanism for nodes to publish and subscribe to topics. Topics are used to communicate data between nodes, and the ROS master manages the connections between publishers and subscribers.

Service management: ROS provides a mechanism for nodes to call services, which are functions provided by other nodes. *roscore* manages the registration of services and the communication between nodes that use them.

Therefore, *roscore* is a critical component of ROS that manages the communication between nodes and provides a centralized naming and registration service. It provides a parameter server for sharing data and a mechanism for managing topics and services. Without *roscore*, it would be difficult to build complex ROS systems that require communication between multiple nodes.

Workspace:

In ROS, a workspace is a directory where you can create and manage your ROS packages. A ROS package is the basic unit of software in ROS, which might contain nodes, libraries, datasets, configuration files, third-party software, or anything else that constitutes a useful module.

A workspace is created by creating a directory and initializing it as a catkin workspace. Catkin is the build system used in ROS and it simplifies the process of building, installing and managing packages. Once a workspace is created, you can create or modify existing ROS packages within it. The structure of a catkin package includes directories such as src, include, msg, srv, scripts, and CMakeLists.txt.

A workspace provides an isolated environment for your ROS packages, which can simplify development and testing. Additionally, using workspaces makes it easier to manage dependencies between packages and to reuse code across multiple projects.

Steps to create a workspace in ROS using command line:

1. Open a terminal window.
2. Create a new directory for your workspace. This directory should be named "catkin_ws" to be consistent with ROS conventions. You can create this directory using the following command:

   mkdir -p ~/catkin_ws/src
3. This command creates a directory named "catkin_ws" in your home directory, with a subdirectory named "src" for your source code.
4. Navigate to the "src" directory in your workspace using the following command:

   cd ~/catkin_ws/src

5. Initialize the workspace using the following command:

   catkin_init_workspace

   This command creates a CMakeLists.txt file in the "src" directory.
6. Return to the root of your workspace by running:

   cd ~/catkin_ws

7. Build the workspace using the following command:

   catkin_make

   This command generates the build and devel directories for your workspace.

The workspace is now set up and ready for you to start creating and building ROS packages.

Python File in ROS workspace:

To create a new Python file inside a ROS workspace, follow the steps given below;

1. Open a terminal and navigate to your ROS workspace using the cd command.
2. Once you are inside your workspace, navigate to the src directory.
3. Create a new package using the following command:

catkin_create_pkg my_package rospy

"my_package" with the name the package. This command will create a new package with the name "my_package" and with rospy as a dependency.

4. Once the package is created, navigate to the package directory using the following command:

cd my_package

5. Create a new Python file using any text editor of your choice, and save it with a .py extension.
6. In your Python file, import the necessary ROS libraries by including the following lines at the top of your file:

```
#!/usr/bin/env python
import rospy
```

7. Write your Python code and save the file.
8. Once you have created your Python file, you can use it as a node in your ROS system or as a module in your package. To make sure your Python file is executable, you may need to set the file permissions using the following command:

chmod +x your_python_file.py

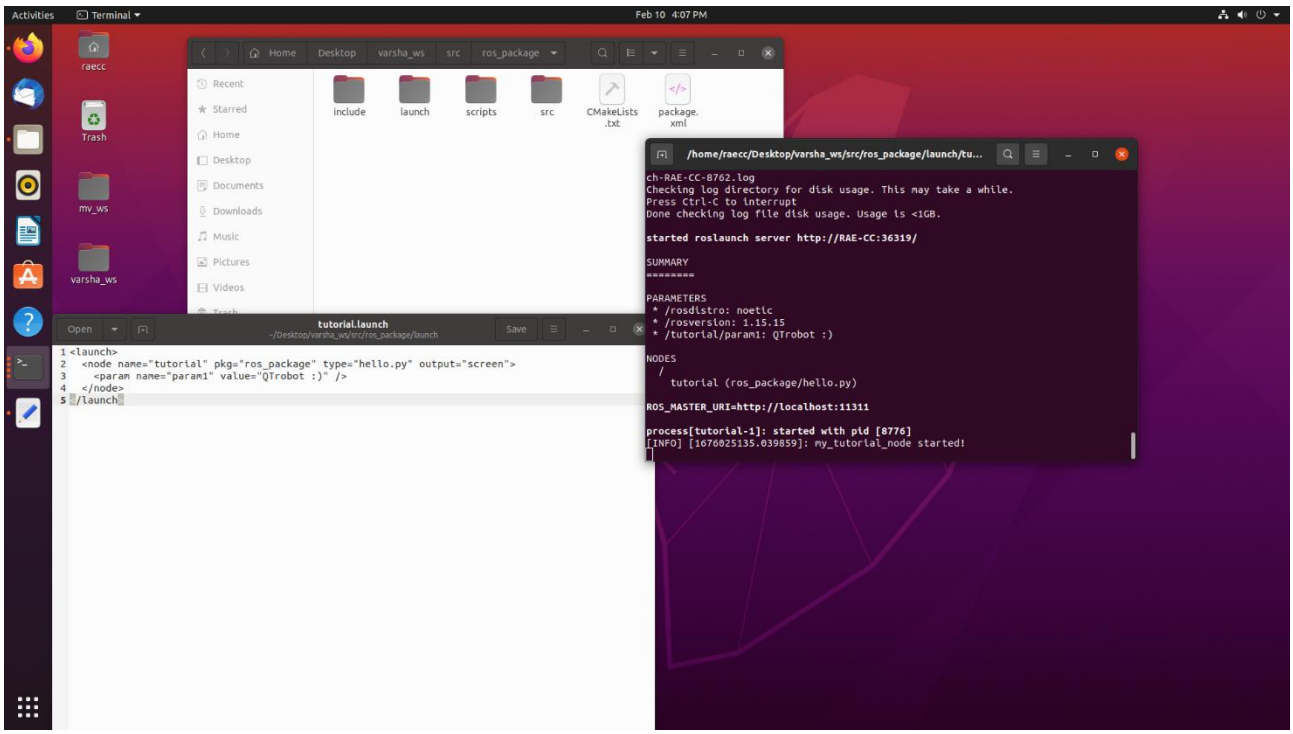
This will make your Python file executable and allow you to run it as a node in your ROS system.

9. To run the python file type the following command

rosrun my_package python_file.py

10. You can also create a launch file and launch the python file using roslaunch by the following command.

roslaunch my_package python_launch_file.launch



Publisher and Subscriber:

In ROS (Robot Operating System), a publisher is a node that publishes messages to a specific topic, while a subscriber is a node that receives and processes messages from a topic.

Publishers and subscribers allow ROS nodes to communicate with each other by sending and receiving messages. Publishers broadcast messages on a specific topic, while subscribers receive these messages and perform actions based on their content.

This publish-subscribe model is one of the core communication patterns in ROS, and it enables modular and distributed robotics systems. By decoupling the production and consumption of data, ROS allows different nodes to be developed, tested, and used independently, as long as they comply with the same message types and topics.

Steps to create publisher and subscriber files:

1. Open a terminal window and navigate to your ROS workspace.
2. Create a new package to hold your publisher and subscriber nodes using the following command:

   catkin_create_pkg my_package rospy

3. Create a scripts directory inside the package using the following command:

   mkdir -p my_package/scripts

4. Create a Python file for your publisher node inside the scripts directory. For example, you can create a file called my_publisher.py and add the following code:

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String
def publisher():
    pub = rospy.Publisher('my_topic', String, queue_size=10)
    rospy.init_node('my_publisher', anonymous=True)
    rate = rospy.Rate(10) # 10hz
    while not rospy.is_shutdown():
        message = "Hello ROS!"
        rospy.loginfo(message)
        pub.publish(message)
        rate.sleep()
if __name__ == '__main__':
    try:
        publisher()
    except rospy.ROSInterruptException:
        pass
```

5. Make the Python file executable using the following command:

```
chmod +x my_publisher.py
```

6. Create another Python file for your subscriber node inside the scripts directory. For example, you can create a file called my_subscriber.py and add the following code:

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String
def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)
def subscriber():
    rospy.init_node('my_subscriber', anonymous=True)
    rospy.Subscriber("my_topic", String, callback)
    rospy.spin()
if __name__ == '__main__':
    try:
        subscriber()
    except rospy.ROSInterruptException:
        pass
```

7. Make the Python file executable using the chmod command :

chmod +x my_subscriber.py

8. Build your ROS workspace using the following command:

catkin_make

9. Open a new terminal window and source your ROS workspace using the following command:

Source devel/setup.bash

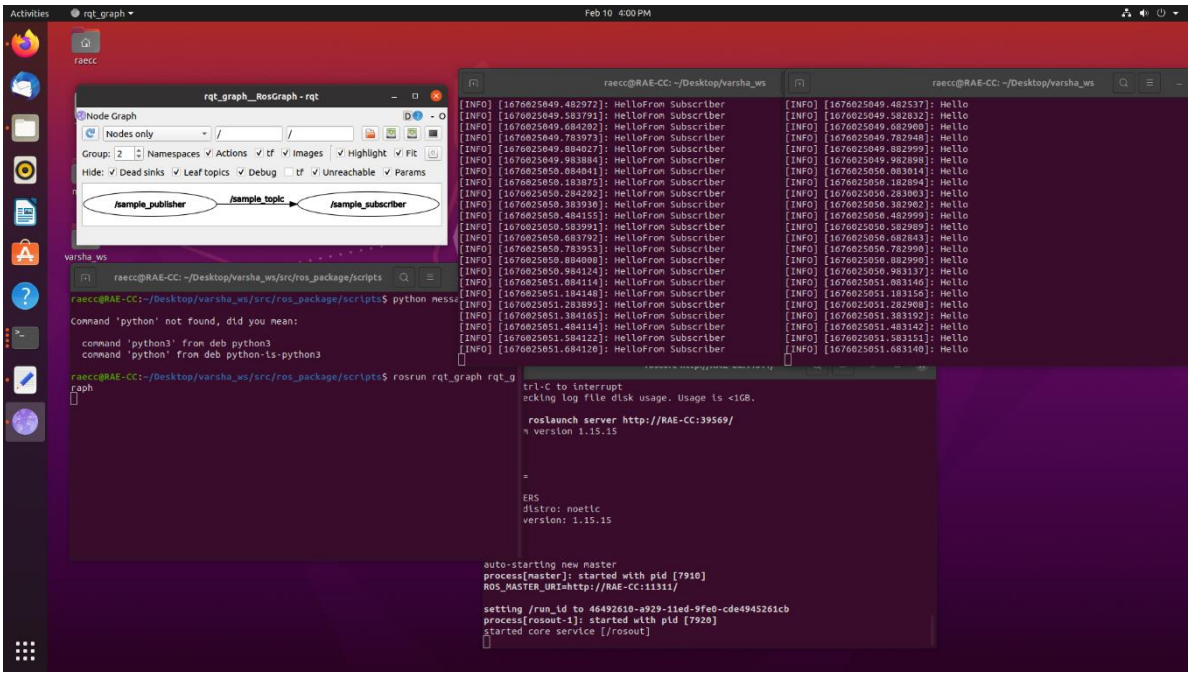
10. Run the publisher node using the following command:

rosrun my_package my_publisher.py

11. Open another terminal window and source your ROS workspace.
12. Run the subscriber node using the following command:

rosrun my_package my_subscriber.py

Now your publisher and subscriber nodes are running and communicating with each other through the my_topic topic.



Opencv-Ros:

OpenCV (Open Source Computer Vision) is a library of programming functions mainly aimed at real-time computer vision. It is widely used in the field of robotics and is integrated with ROS (Robot Operating System), which is a popular framework for developing robotic systems.

In ROS, the OpenCV library can be used for a variety of computer vision tasks, such as image and video processing, object detection and tracking, and machine learning. ROS provides various packages for OpenCV, which can be used to access the functionality of the library within a ROS system.

The most commonly used OpenCV package in ROS is the "cv_bridge" package, which provides a bridge between ROS image messages and OpenCV image formats. This package allows you to easily convert ROS image messages to OpenCV image formats, and vice versa.

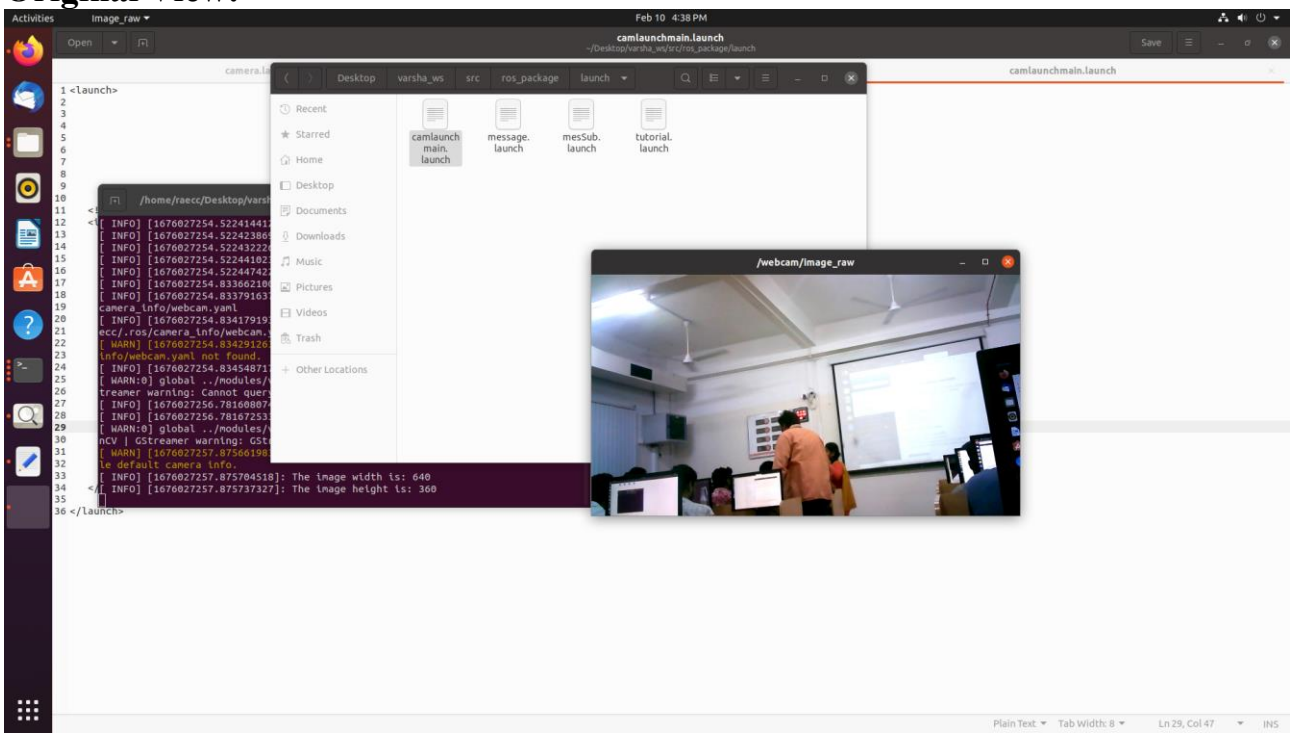
Other OpenCV packages in ROS include "image_proc" for basic image processing, "image_geometry" for 3D geometry calculations using image data, and "camera_calibration" for camera calibration using a chessboard pattern.

To use OpenCV in ROS, you can simply include the necessary ROS packages in your project, and then use the standard OpenCV library functions within your ROS nodes.

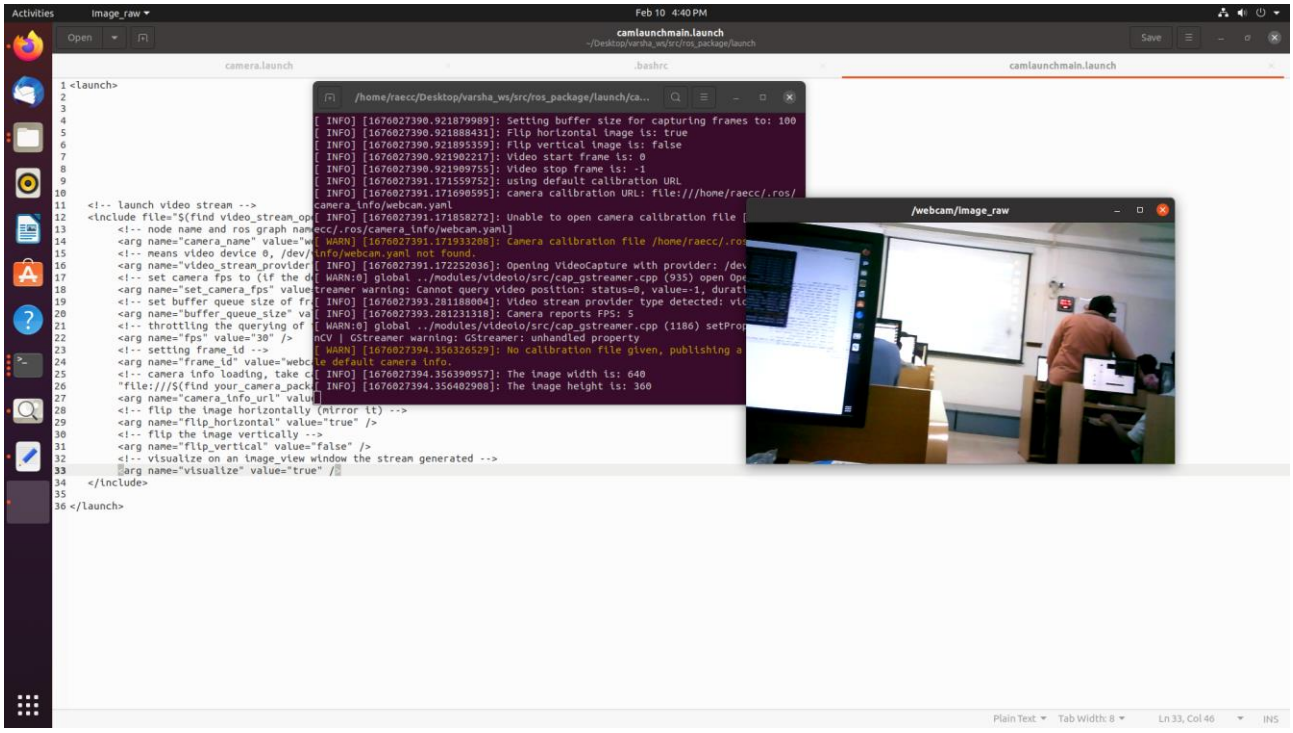
I've created a publisher and subscriber module that uses webcam to display the live video.

In that, I have edited the code such that the output video to flip horizontally and flip vertically.

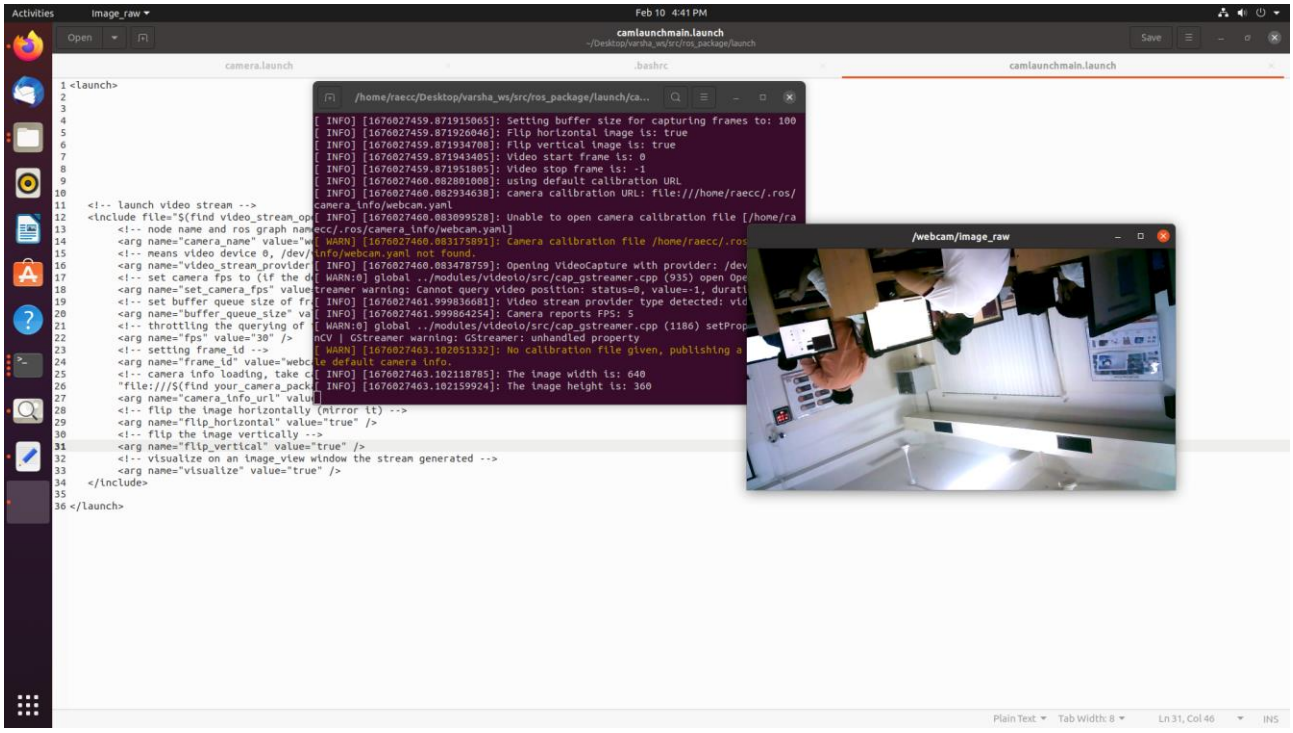
Original View:



Horizontal Flip:



Vertical Flip:



Using opencv-bridge, I've inserted my name in the live video using the cv1.puttext() function of opencv

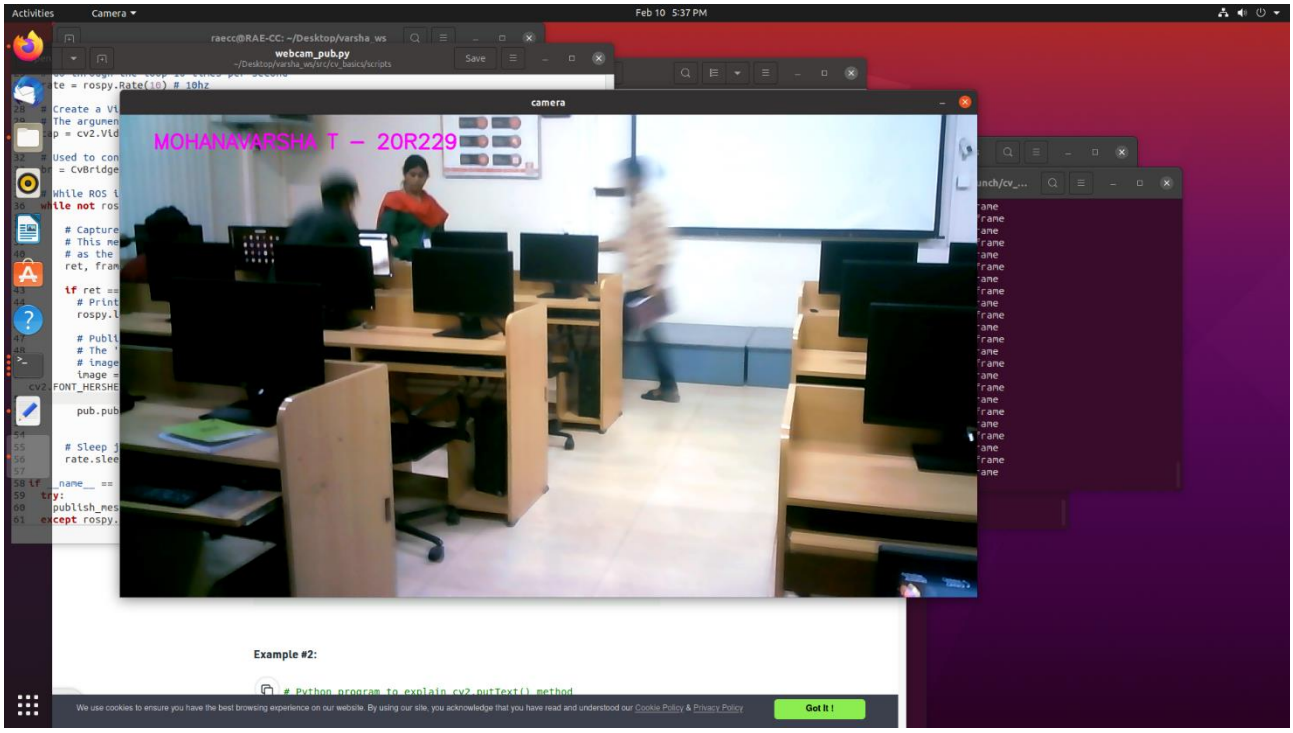
Publisher code:

```
#!/usr/bin/env python3
# Import the necessary libraries
import rospy                                # Python library for ROS
from sensor_msgs.msg import Image # Image is the message type
from cv_bridge import CvBridge
import cv2                                  # OpenCV library
def publish_message():
  pub = rospy.Publisher('video  frames', Image, queue  size=10)
  # Tells rospy the name of the node.
  # Anonymous = True makes sure the node has a unique name. Random
  # numbers are added to the end of the name.
  rospy.init  node('video  pub  py', anonymous=True)
  # Go through the loop 10 times per second
  rate = rospy.Rate(10) # 10hz
  # Create a VideoCapture object
  # The argument '0' gets the default webcam.
  cap = cv2.VideoCapture(0)
  # Used to convert between ROS and OpenCV images
  br = CvBridge()
  # While ROS is still running.
  while not rospy.is_shutdown():
     ret, frame = cap.read()
     if ret == True:
       # Print debugging information to the terminal
       rospy.loginfo('publishing video frame')
       # Publish the image.
       # The 'cv2_to_imgmsg' method converts an OpenCV
       # image to a ROS image message
       image = cv2.putText(frame, 'A', (150,150), cv2.FONT_HERSHEY_SIM-
PLEX,
                1, (255,0, 0), 2, cv2.LINE_AA)
       pub.publish(br.cv2_to_imgmsg(image))
     # Sleep just enough to maintain the desired rate
     rate.sleep()

if __name__ == '__main__':
  try:
    publish_message()
  except rospy.ROSInterruptException:
    pass
```

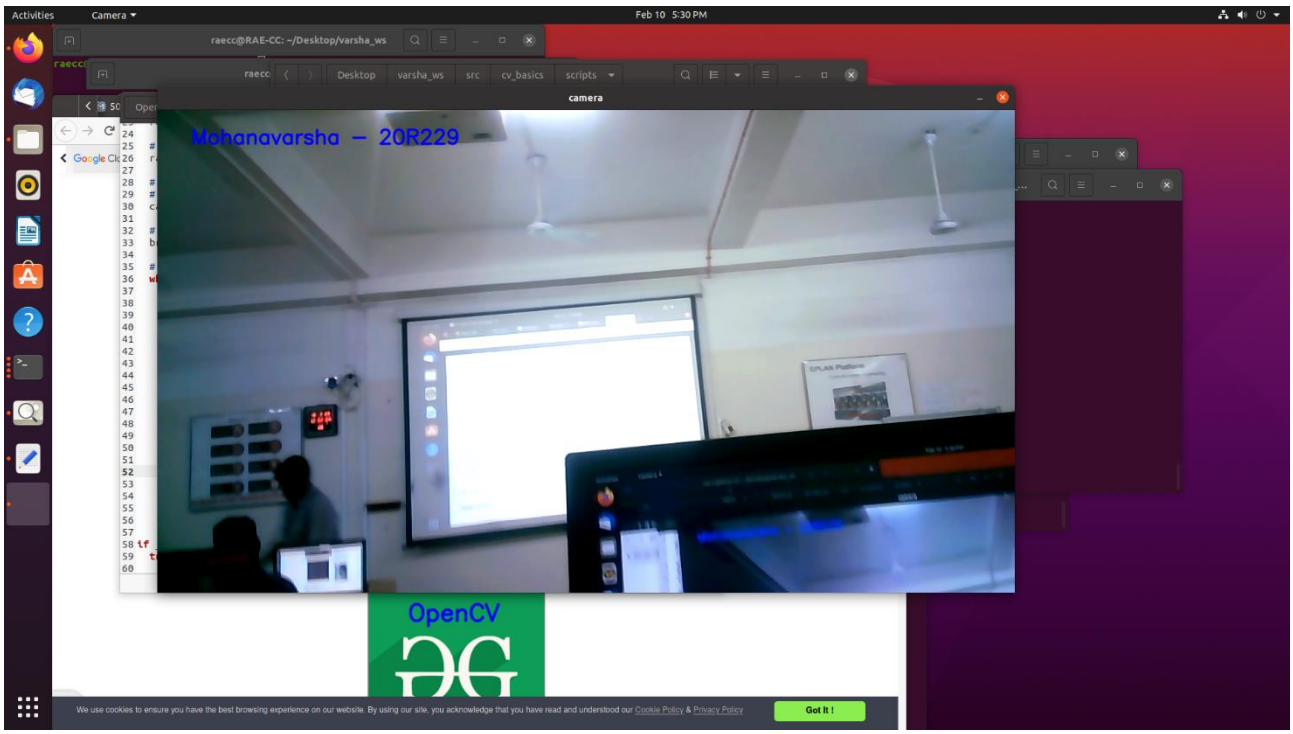
Subscriber Code:

```
#!/usr/bin/env python3
# Import the necessary libraries
import rospy # Python library for ROS
from sensor_msgs.msg import Image # Image is the message type
from cv  bridge import CvBridge # Package to convert between ROS and
OpenCV Images
import cv2 # OpenCV library

def callback(data):

  # Used to convert between ROS and OpenCV images
  br = CvBridge()

  # Output debugging information to the terminal
  rospy.loginfo("receiving video frame")

  # Convert ROS Image message to OpenCV image
  current_frame = br.imgmsg_to_cv2(data)

  # Display image
  cv2.imshow("camera", current_frame)

  cv2.waitKey(1)

def receive_message():

  # Tells rospy the name of the node.
  # Anonymous = True makes sure the node has a unique name. Random
  # numbers are added to the end of the name.
  rospy.init_node('video_sub_py', anonymous=True)

  # Node is subscribing to the video_frames topic
  rospy.Subscriber('video_frames', Image, callback)

  # spin() simply keeps python from exiting until this node is stopped
  rospy.spin()

  # Close down the video stream when done
  cv2.destroyAllWindows()

if __name__ == '__main__':
  receive_message()
```

Output:



I can also change color, position, font size, font style of my text as in opencv.



TURTLESIM:

To run the turtlesim node in ROS, you can follow these steps:

1. Launch the roscore by opening a terminal and running:

   Roscore

2. Open a new terminal and run the turtlesim_node:

   rosrun turtlesim turtlesim_node

This will launch the turtlesim graphical user interface.

3. Open another terminal and run the turtle_teleop_key node to control the turtle:

   rosrun turtlesim turtle_teleop_key

This will allow you to move the turtle around using the arrow keys.