Principles of Robot Autonomy I

The Robot Operating System (ROS)
The Pub/Sub design pattern

- Stands for Publish-Subscribe
- Each component (i.e. node) can:
  - *Publish*: send messages regardless of whether someone is listening
  - *Subscribe*: receive messages if anyone is sending them regardless of who
The Pub/Sub design pattern

Note: there are countless ways to IMPLEMENT pub/sub!
What is ROS?

• An implementation of a network-aware pub/sub* geared towards robotic applications

• Lots of open-source software shared by the community:
  • SLAM (gmapping, amcl)
  • Vision (OpenCV, PCL, OpenNI)
  • Arm Navigation (MoveIt)
  • Simulation (Gazebo)
The main components

- Nodes
  - talk to each other over topics (think chat rooms).
- Master
  - coordinates the whole thing
- Message types: abstraction away from specific hardware
  - Camera image
  - Laser scan data
  - Motion control
ROS Node

- A process (typically Python or C++) that runs some computation
- The “fundamental” building block
- Can act as a subscriber, publisher or both
- Nodes talk to each other over “topics”
- Run them using `rosrun <package> <node>`
- Initialize using `rospy.init_node()`

Note: nodelets are different. They are not individual processes, they share memory
ROS Master

- A process that is in charge of coordinating nodes, publishers and subscribers
- Exactly one of them running at any time
- Nodes will not be able to find each other without Master
Abstraction vs Implementation

- Pub/sub is only an **abstraction**, a way to think about the architecture of your software
  - Ex: Messages do NOT go through Master
A bit of networking...

- Two important environment variables:
  - ROS_MASTER_URI
    - The IP address of the computer running master
  - ROS_IP
    - The IP address of your computer
Getting help

- ROS wiki (http://wiki.ros.org/)
- Github
- Stack Overflow
- The Construct / Robot Ignite Academy
- Google :)

4.2 face_recognition

Recognize faces in ROS sensor_msgs/Image using Face Recognition and outputs detected faces with labels as ROS openv cv_apps/FaceArrayStamped message. See Tutorials for more info.

4.2.1 Subscribed Topics

Image (sensor_msgs/Image)

The image topic. Should be remapped to the name of the real image topic.

Faces (opencv_apps/FaceArrayStamped)

Array of detected face location in image coordinates.

4.2.2 Published Topics

-output (opencv_apps/FaceArrayStamped)

A copy of input image which indicates detected faces position as a circle

-debug_image (sensor_msgs/Image)

A copy of input image which indicates detected faces' positions, labels and confidences.

4.2.3 Parameters

-approximate_sync (bool, default: false)

Approximately synchronize messages of input image and face array

-queue_size (int, default: 100)

Size of queue for subscribing topics

-model_method (string, default: "eigen")

Method for face recognition (Either "eigen", "fisher" or "LBPH")

-use_samed_data (bool, default: true)

Load and train data from path specified by -data_dir

-save_train_data (bool, default: true)

Save train data to path specified by -data_dir for retraining

-data_dir (string, default: "/root/opencv_apps/fac e_data/"")

Path to directory for saving train data

-face_model_width (int, default: 190)

Width of training face image

-face_model_height (int, default: 90)

Height of training face image

-face_padding (double, default: 0.1)

Padding ratio for each face

-model_num_components (int, default: 0)

Number of components for face recognizer model (0 is treated as unlimited)

-model_threshold (double, default: 0.0000)

Threshold for face recognizer model
Example 1: camera

• Installing packages
  • apt-get / system-wide
  • From source

• Live demo
Example 2: publisher

- A bit of networking
- Talking to an Arduino (rosserial_python)
- Moveit (MoveGroup)
- Combined publisher/subscriber
  - Alternate version
- Live demo
import rospy
import std_msgs.msg

class BallGripper:
    def __init__(self):
        rospy.init_node('ball_gripper', anonymous=True)

        self.command_listener = rospy.Subscriber('/ball_gripper/command', std_msgs.msg.String, self.callback)
        self.servo_publisher = rospy.Publisher('/servo', std_msgs.msg.UInt16, queue_size=10)

    def callback(self, msg):
        rospy.loginfo(rospy.get_caller_id() + 'I heard %s', msg.data)
        if msg.data == "release":
            release_msg = std_msgs.msg.UInt16(180)
            self.servo_publisher.publish(release_msg)

if __name__ == '__main__':
    ball_gripper = BallGripper()
    rospy.spin()
Offline question 1

• What are some other kinematic models that are commons in robotics? Is it common to have to derive kinematic model for every new robotic system/component, or they usually share similar kinematic model for each module that can be easily reused?

• Unicycle model is part of a family of models often used for wheeled robots (bicycle model, Dubin car etc.).
• Stanford teaches an entire class on this: ME 227
• For most complicated robot, people rely on urdf’s and dedicated packages that compute dynamics from them
  • Bullet, Drake, MuJoCo, Matlab Simscape, RigidBodyDynamics.jl …
Offline question 2

• In slide 31, we briefly went through a `catkin_create_pkg` command to build ROS package. Do dependencies always have to be pass via command-line? Or if there's a way for us to specify dependencies via the XML or a config file?

• `catkin_create_pkg` is just a helper function to get you started
• The xml files in the package can be edited to add dependencies (package.xml in this case)
• `rosdep` is another tool that lets you deal with system dependencies
Other questions?