Some ROS Slides

D.A. Forsyth
Credits

• I didn’t make these slides
  • I’ve cut them from a series of 10 lectures by Roi Yehoshua, at Bar-Ilan
  • without permission (though I’ll try and fix this!)
  • URL to full slides on website
    • I’ve cut slides with details of code, etc - get these from full slides

• Purpose:
  • enough framework to get you started on ROS
  • further tutorial material, etc on website, too
ROS - Lecture 1

ROS Introduction
Main concepts
Basic commands

Lecturer: Roi Yehoshua
roiyeho@gmail.com
The Problem

• Lack of standards for robotics
What is ROS?

- ROS is an open-source robot operating system
- A set of software libraries and tools that help you build robot applications that work across a wide variety of robotic platforms
- Originally developed in 2007 at the Stanford Artificial Intelligence Laboratory and development continued at Willow Garage
- Since 2013 managed by OSRF (Open Source Robotics Foundation)
ROS Main Features

ROS has two "sides"

• The operating system side, which provides standard operating system services such as:
  – hardware abstraction
  – low-level device control
  – implementation of commonly used functionality
  – message-passing between processes
  – package management

• A suite of user contributed packages that implement common robot functionality such as SLAM, planning, perception, vision, manipulation, etc.
ROS Main Features

- navigation
- task executive
- visualization
- simulation
- perception
- control
- planning
- message passing
- data logging
- real-time capabilities
- device drivers
- web browser
- memory management
- window manager
- email client
- process management
- scheduler
- device drivers
- file system

Taken from Sachin Chitta and Radu Rusu (Willow Garage)
ROS Philosophy

• **Peer to Peer**
  – ROS systems consist of numerous small computer programs which connect to each other and continuously exchange *messages*

• **Tools-based**
  – There are many small, generic programs that perform tasks such as visualization, logging, plotting data streams, etc.

• **Multi-Lingual**
  – ROS software modules can be written in any language for which a *client library has been written*. Currently client libraries exist for C++, Python, LISP, Java, JavaScript, MATLAB, Ruby, and more.

• **Thin**
  – The ROS conventions encourage contributors to create stand-alone libraries and then *wrap those libraries so they send and receive messages to/from other ROS modules*.

• **Free and open source**
ROS Wiki

• http://wiki.ros.org/
• Installation: http://wiki.ros.org/ROS/Installation
• Tutorials: http://wiki.ros.org/ROS/Tutorials
• ROS Tutorial Videos
  – http://www.youtube.com/playlist?list=PLDC89965A56E6A8D6
• ROS Cheat Sheet
Robots using ROS

http://wiki.ros.org/Robots

- Fraunhofer IPA Care-O-bot
- Aldebaran Nao
- Willow Garage PR2
- Merlin miabotPro
- Clearpath Robotics Husky
- Videre Erratic
- Lego NXT
- iRobot Roomba
- AscTec Quadrotor
- Clearpath Robotics Kingfisher
- TurtleBot
- Shadow Hand
- Robotnik Guardian
- CoroWare Corobot
- Festo Didactic Robotino
ROS Core Concepts

- Nodes
- Messages and Topics
- Services
- ROS Master
- Parameters
- Stacks and packages
ROS Nodes

• Single-purposed executable programs
  – e.g. sensor driver(s), actuator driver(s), mapper, planner, UI, etc.

• Individually compiled, executed, and managed

• Nodes are written using a ROS client library
  – roscpp - C++ client library
  – rospy - python client library

• Nodes can publish or subscribe to a Topic
• Nodes can also provide or use a Service
ROS Topics

• A topic is a name for a stream of messages with a defined type
  – e.g., data from a laser range-finder might be sent on a topic called scan, with a message type of LaserScan

• Nodes communicate with each other by publishing messages to topics

• Publish/Subscribe model: 1-to-N broadcasting
ROS Topics

- **Kinect Node**: Publishes 3D data from Kinect as messages.
- **3D Processing Node**: Processes Kinect data and publishes directions.
- **Arduino Node**: Subscribes to directions and commands motors.
ROS Messages

• Strictly-typed data structures for inter-node communication
• For example, `geometry_msgs/Twist` is used to express velocity commands:
  
  Vector3 linear
  Vector3 angular

  – Vector3 is another message type composed of:

  float64 x
  float64 y
  float64 z
ROS Services

• Synchronous inter-node transactions / RPC
• Service/Client model: 1-to-1 request-response
• Service roles:
  – carry out remote computation
  – trigger functionality / behavior
• Example:
  – map_server/static_map - retrieves the current grid map used by the robot for navigation
ROS Master

• Provides connection information to nodes so that they can transmit messages to each other
  – Every node connects to a master at startup to register details of the message streams they publish, and the streams to which that they to subscribe
  – When a new node appears, the master provides it with the information that it needs to form a direct peer-to-peer connection with other nodes publishing and subscribing to the same message topics
• Let’s say we have two nodes: a Camera node and an Image_viewer node
• Typically the camera node would start first notifying the master that it wants to publish images on the topic "images":

![Diagram of ROS Master](image-url)
• Now, Image_viewer wants to subscribe to the topic "images" to see if there's maybe some images there:
Now that the topic "images" has both a publisher and a subscriber, the master node notifies Camera and Image_viewer about each others existence, so that they can start transferring images to one another:
Parameter Server

- A shared, multi-variate dictionary that is accessible via network APIs
- Best used for static, non-binary data such as configuration parameters
- Runs inside the ROS master
ROS Packages

• Software in ROS is organized in *packages*.
• A package contains one or more nodes and provides a ROS interface
• Most of ROS packages are hosted in GitHub
ROS Package System

Taken from Sachin Chitta and Radu Rusu (Willow Garage)
# ROS Distribution Releases

<table>
<thead>
<tr>
<th>Distro</th>
<th>Release date</th>
<th>Poster</th>
<th>Tuturtle, turtle in tutorial</th>
<th>EOL date</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROS Kinetic Kame (Recommended)</td>
<td>May 23rd, 2016</td>
<td><img src="image1" alt="Poster" /></td>
<td><img src="image2" alt="Turtle" /></td>
<td>May, 2021</td>
</tr>
<tr>
<td>ROS Jade Turtle</td>
<td>May 23rd, 2015</td>
<td><img src="image3" alt="Poster" /></td>
<td><img src="image4" alt="Turtle" /></td>
<td>May, 2017</td>
</tr>
<tr>
<td>ROS Indigo Igloo</td>
<td>July 22nd, 2014</td>
<td><img src="image5" alt="Poster" /></td>
<td><img src="image6" alt="Turtle" /></td>
<td>April, 2019 (Trusty EOL)</td>
</tr>
<tr>
<td>ROS Hydro Medusa</td>
<td>September 4th, 2013</td>
<td><img src="image7" alt="Poster" /></td>
<td><img src="image8" alt="Turtle" /></td>
<td>May, 2015</td>
</tr>
</tbody>
</table>
ROS Supported Platforms

• ROS is currently supported only on Ubuntu
  – other variants such as Windows and Mac OS X are considered experimental (will be supported on ROS 2.0)
• ROS distribution supported is limited to <=3 latest Ubuntu versions
• ROS Jade supports the following Ubuntu versions:
  – Vivid (15.04)
  – Utopic (14.04)
  – Trusty (14.04 LTS)
• ROS Indigo supports the following Ubuntu versions:
  – Trusty (14.04 LTS)
  – Saucy (13.10)
ROS Installation

• If you already have Ubuntu installed, follow the instructions at:

• http://wiki.ros.org/ROS/Installation

• note: there are different distributions, etc described there
• ROS relies on the notion of combining spaces using the shell environment
  – This makes developing against different versions of ROS or against different sets of packages easier
• After you install ROS you will have setup.*sh files in '/opt/ros/<distro>/', and you could source them like so:
  
  $ source /opt/ros/indigo/setup.bash
• You will need to run this command on every new shell you open to have access to the ros commands, unless you add this line to your bash startup file (~/.bashrc)
  – If you used the pre-installed VM it’s already done for you
ROS Basic Commands

• roscore
• rosrund
• rosnodes
• rostopic
• roscore is the first thing you should run when using ROS

$ roscore

• roscore will start up:
  – a ROS Master
  – a ROS Parameter Server
  – a rosout logging node
rosrun

• rosrun allows you to run a node
• Usage:

$ rosrun <package> <executable>

• Example:

$ rosrun turtlesim turtlesim_node
rosnode

- Displays debugging information about ROS nodes, including publications, subscriptions and connections

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rosnodes list$</td>
<td>List active nodes</td>
</tr>
<tr>
<td>rosnodes ping$</td>
<td>Test connectivity to node</td>
</tr>
<tr>
<td>rosnodes info$</td>
<td>Print information about a node</td>
</tr>
<tr>
<td>rosnodes kill$</td>
<td>Kill a running node</td>
</tr>
<tr>
<td>rosnodes machine$</td>
<td>List nodes running on a particular machine</td>
</tr>
</tbody>
</table>
**rostopic**

- Gives information about a topic and allows to publish messages on a topic

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rostopic list$</td>
<td>List active topics</td>
</tr>
<tr>
<td>rosnode echo /topic$</td>
<td>Prints messages of the topic to the screen</td>
</tr>
<tr>
<td>rostopic info /topic$</td>
<td>Print information about a topic</td>
</tr>
<tr>
<td>rostopic type /topic$</td>
<td>Prints the type of messages the topic publishes</td>
</tr>
<tr>
<td>rostopic pub /topic type args$</td>
<td>Publishes data to a topic</td>
</tr>
</tbody>
</table>
ROS - Lecture 3

ROS topics
Publishers and subscribers
roslaunch
Custom message types

Lecturer: Roi Yehoshua
roiyeho@gmail.com
## ROS Communication Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Best used for</th>
</tr>
</thead>
<tbody>
<tr>
<td>Topic</td>
<td>One-way communication, especially if there might be multiple nodes listening (e.g., streams of sensor data)</td>
</tr>
<tr>
<td>Service</td>
<td>Simple request/response interactions, such as asking a question about a node's current state</td>
</tr>
<tr>
<td>Action</td>
<td>Most request/response interactions, especially when servicing the request is not instantaneous (e.g., navigating to a goal location)</td>
</tr>
</tbody>
</table>
ROS Topics

• Topics implement a *publish/subscribe* communication mechanism
  – one of the more common ways to exchange data in a distributed system.
• Before nodes start to transmit data over topics, they must first announce, or *advertise*, both the topic name and the types of messages that are going to be sent
• Then they can start to send, or *publish*, the actual data on the topic.
• Nodes that want to receive messages on a topic can *subscribe* to that topic by making a request to roscore.
• After subscribing, all messages on the topic are delivered to the node that made the request.
ROS Topics

• In ROS, all messages on the same topic must be of the same data type
• Topic names often describe the messages that are sent over them
• For example, on the PR2 robot, the topic / wide_stereo/right/image_color is used for color images from the rightmost camera of the wide-angle stereo pair
Topic Publisher

• Manages an advertisement on a specific topic
• Created by calling `NodeHandle::advertise()`
  – Registers this topic in the master node
• Example for creating a publisher:

```cpp
ros::Publisher chatter_pub = node.advertise<std_msgs::String>("chatter", 1000);
```
  – First parameter is the topic name
  – Second parameter is the queue size
• Once all the publishers for a given topic go out of scope the topic will be unadvertised
Running the Nodes From Terminal

- Run roscore
- Run the nodes in two different terminals:

  $ rosrun chat_pkg talker
  $ rosrun chat_pkg listener
Running the Nodes From Terminal

• You can use rosnode and rostopic to debug and see what the nodes are doing

• Examples:
  $ rosnode info /talker
  $ rosnode info /listener
  $ rostopic list
  $ rostopic info /chatter
  $ rostopic echo /chatter
rqt_graph

- rqt_graph creates a dynamic graph of what's going on in the system
- Use the following command to run it:

```
$ rosrun rqt_graph rqt_graph
```
ROS Names

• ROS names must be unique
• If the same node is launched twice, roscore directs the older node to exit
• To change the name of a node on the command line, the special `__name` remapping syntax can be used
• The following two shell commands would launch two instances of talker named talker1 and talker2

```bash
$ rosrun chat_pkg talker __name:=talker1
$ rosrun chat_pkg talker __name:=talker2
```
Instantiating two talker programs and routing them to the same receiver
roslaunch

• a tool for easily launching multiple ROS nodes as well as setting parameters on the Parameter Server

• roslaunch operates on launch files which are XML files that specify a collection of nodes to launch along with their parameters
  – By convention these files have a suffix of .launch

• Syntax:
  
  $ roslaunch PACKAGE LAUNCH_FILE

• roslaunch automatically runs roscore for you
Launch File Example

- Launch file for launching the talker and listener nodes:

  ```xml
  <launch>
    <node name="talker" pkg="chat_pkg" type="talker" output="screen"/>
    <node name="listener" pkg="chat_pkg" type="listener" output="screen"/>
  </launch>
  ```

- Each `<node>` tag includes attributes declaring the ROS graph name of the node, the package in which it can be found, and the type of node, which is the filename of the executable program.

- `output="screen"` makes the ROS log messages appear on the launch terminal window.
Launch File Example

```bash
$ roslaunch chat_pkg chat.launch
```
PACMOD

- Publishes a set of relevant vehicle messages
- Subscribes to a set of relevant vehicle messages
- https://github.com/astuff/pacmod3
# Published Topics

<table>
<thead>
<tr>
<th>Message Type</th>
<th>Topic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>can_msgs/Frame</td>
<td>can_rx</td>
<td>All data published on this topic is intended to be sent to the PACMod system via a CAN interface.</td>
</tr>
<tr>
<td>pacmod_msgs/GlobalRpt</td>
<td>parsed_tx/global_rpt</td>
<td>High-level data about the entire PACMod system.</td>
</tr>
<tr>
<td>pacmod_msgs/SystemRptFloat</td>
<td>parsed_tx/accel_rpt</td>
<td>Status and parsed values [pct] of the throttle subsystem.</td>
</tr>
<tr>
<td>pacmod_msgs/SystemRptFloat</td>
<td>parsed_tx/brake_rpt</td>
<td>Status and parsed values [pct] of the steering subsystem.</td>
</tr>
<tr>
<td>pacmod_msgs/SystemRptInt</td>
<td>parsed_tx/turn_rpt</td>
<td>Status and parsed values [enum] of the turn signal subsystem.</td>
</tr>
<tr>
<td>pacmod_msgs/SystemRptInt</td>
<td>parsed_tx/shift_rpt</td>
<td>Status and parsed values [enum] of the gear/transmission subsystem.</td>
</tr>
<tr>
<td>pacmod_msgs/SystemRptFloat</td>
<td>parsed_tx/steer_rpt</td>
<td>Status and parsed values [rad] of the steering subsystem.</td>
</tr>
<tr>
<td></td>
<td>parsed_tx/vehicle_speed_rpt</td>
<td>The vehicle's current speed [mph], the validity of the speed message [bool], and the raw CAN message from the vehicle CAN.</td>
</tr>
<tr>
<td>std_msgs/Float64</td>
<td>as_tx/vehicle_speed</td>
<td>The vehicle's current speed [m/s].</td>
</tr>
<tr>
<td>std_msgs/Bool</td>
<td>as_tx/enable</td>
<td>The current status of the PACMod's control of the vehicle. If the PACMod is enabled, this value will be true. If it is disabled or overridden, this value will be false.</td>
</tr>
<tr>
<td>pacmod_msgs/PacmodCmd</td>
<td>as_rx/accel_cmd</td>
<td>Commands the throttle subsystem to seek a specific pedal position [pct - 0.0 to 1.0].</td>
</tr>
<tr>
<td>pacmod_msgs/PacmodCmd</td>
<td>as_rx/brake_cmd</td>
<td>Commands the brake subsystem to seek a specific pedal position [pct - 0.0 to 1.0].</td>
</tr>
<tr>
<td>pacmod_msgs/PacmodCmd</td>
<td>as_rx/shift_cmd</td>
<td>Commands the gear/transmission subsystem to shift to a different gear [enum].</td>
</tr>
<tr>
<td>pacmod_msgs/PositionWithSpeed</td>
<td>as_rx/steer_cmd</td>
<td>Commands the steering subsystem to seek a specific steering wheel angle [rad] at a given rotation velocity [rad/s].</td>
</tr>
<tr>
<td>pacmod_msgs/PacmodCmd</td>
<td>as_rx/turn_cmd</td>
<td>Commands the turn signal subsystem to transition to a given state [enum].</td>
</tr>
<tr>
<td>std_msgs/Bool</td>
<td>as_rx/enable</td>
<td>Enables [true] or disables [false] PACMod's control of the vehicle.</td>
</tr>
</tbody>
</table>
Creating Custom Messages

- These primitive types are used to build all of the messages used in ROS
- For example, (most) laser range-finder sensors publish sensor_msgs/LaserScan messages

```python
# Single scan from a planar laser range-finder

Header header
# stamp: The acquisition time of the first ray in the scan.
# frame id: The laser is assumed to spin around the positive Z axis
# (counterclockwise, if Z is up) with the zero angle forward along the x axis

float32 angle_min # start angle of the scan [rad]
float32 angle_max # end angle of the scan [rad]
float32 angle_increment # angular distance between measurements [rad]

float32 time_increment # time between measurements [seconds] - if your scanner
# is moving, this will be used in interpolating position of 3d points
float32 scan_time # time between scans [seconds]

float32 range_min # minimum range value [m]
float32 range_max # maximum range value [m]

float32[] ranges # range data [m] (Note: values < range_min or > range_max should be discarded)
float32[] intensities # intensity data [device-specific units]. If your
# device does not provide intensities, please leave the array empty.
```
Creating Custom Messages

- Using standardized message types for laser scans and location estimates enables nodes can be written that provide navigation and mapping (among many other things) for a wide variety of robots
- However, there are times when the built-in message types are not enough, and we have to define our own messages
msg Files

- ROS messages are defined by special message-definition files in the `msg` directory of a package.
- These files are then compiled into language-specific implementations that can be used in your code.
- Each line in the file specifies a type and a field name.
Using rosmsg

- That's all you need to do to create a msg
- Let's make sure that ROS can see it using the rosmmsg show command:

```bash
$ rosmmsg show [message type]
```
ROS - Lecture 4

Gazebo simulator
Reading Sensor Data
Wander-Bot

Lecturer: Roi Yehoshua
roiyeho@gmail.com
Simulators

- In simulation, we can model as much or as little of reality as we desire
- Sensors and actuators can be modeled as ideal devices, or they can incorporate various levels of distortion, errors, and unexpected faults
- Automated testing of control algorithms typically requires simulated robots, since the algorithms under test need to be able to experience the consequences of their actions
- Due to the isolation provided by the messaging interfaces of ROS, a vast majority of the robot’s software graph can be run identically whether it is controlling a real robot or a simulated robot
ROS Stage Simulator

- [http://wiki.ros.org/simulator_stage](http://wiki.ros.org/simulator_stage)
- A 2D simulator that provides a virtual world populated by mobile robots, along with various objects for the robots to sense and manipulate
ROS Stage Simulator

• In perspective view of the robot
Gazebo

• A multi-robot simulator
• Like Stage, it is capable of simulating a population of robots, sensors and objects, but does so in 3D
• Includes an accurate simulation of rigid-body physics and generates realistic sensor feedback
• Allows code designed to operate a physical robot to be executed in an artificial environment
• Gazebo is under active development at the OSRF (Open Source Robotics Foundation)
Gazebo

- Gazebo Demo
Gazebo

• ROS Indigo comes with Gazebo V2.2
• Gazebo home page - http://gazebosim.org/
• Gazebo tutorials - http://gazebosim.org/tutorials
Gazebo consists of two processes:

- **Server**: Runs the physics loop and generates sensor data
  - *Executable*: gzserver
  - *Libraries*: Physics, Sensors, Rendering, Transport

- **Client**: Provides user interaction and visualization of a simulation.
  - *Executable*: gzclient
  - *Libraries*: Transport, Rendering, GUI
ROS - Lecture 5

Mapping in ROS
rviz
ROS Services

Lecturer: Roi Yehoshua
roiyeho@gmail.com
Why Mapping?

• Building maps is one of the fundamental problems in mobile robotics
• Maps allow robots to efficiently carry out their tasks, such as localization, path planning, activity planning, etc.
• There are different ways to create a map of the environment
Cellular Decomposition

• Decompose free space for path planning
• Exact decomposition
  – Cover the free space exactly
  – Example: trapezoidal decomposition, meadow map
• Approximate decomposition
  – Represent part of the free space, needed for navigation
  – Example: grid maps, quadtrees, Voronoi graphs
Cellular Decomposition

Metric map of the environment

Exact cell decomposition

Rectangular cell decomposition

Regular cell decomposition

Quadtree decomposition
Occupancy Grid Map (OGM)

• Maps the environment as a grid of cells
  – Cell sizes typically range from 5 to 50 cm
• Each cell holds a probability value that the cell is occupied in the range [0,100]
• Unknown is indicated by -1
  – Usually unknown areas are areas that the robot sensors cannot detect (beyond obstacles)
Occupancy Grid Map

White pixels represent free cells
Black pixels represent occupied cells
Gray pixels are in unknown state
Occupancy Grid Maps

• Pros:
  – Simple representation
  – Speed

• Cons:
  – Not accurate - if an object falls inside a portion of a grid cell, the whole cell is marked occupied
  – Wasted space
Maps in ROS

- Map files are stored as images, with a variety of common formats being supported (such as PNG, JPG, and PGM)
- Although color images can be used, they are converted to grayscale images before being interpreted by ROS
- Associated with each map is a YAML file that holds additional information about the map
Editing Map Files

- Since maps are represented as image files, you can edit them in your favorite image editor.
- This allows you to tidy up any maps that you create from sensor data, removing things that shouldn’t be there, or adding in fake obstacles to influence path planning.
- For example, you can stop the robot from planning paths through certain areas of the map by drawing a line across a corridor you don’t want the robot to drive through.
Editing Map Files
SLAM

- Simultaneous localization and mapping (SLAM) is a technique used by robots to build up a map within an unknown environment while at the same time keeping track of their current location.
- A chicken or egg problem: An unbiased map is needed for localization while an accurate pose estimate is needed to build that map.
gmapping

- [http://wiki.ros.org/gmapping](http://wiki.ros.org/gmapping)
- The gmapping package provides laser-based SLAM as a ROS node called `slam_gmapping`
- Uses the FastSLAM algorithm
- It takes the laser scans and the odometry and builds a 2D occupancy grid map
- It updates the map state while the robot moves
- [ROS with gmapping video](http://wiki.ros.org/gmapping)
Install gmapping

• gmapping is not part of ROS Indigo installation
• To install gmapping run:

```bash
$ sudo apt-get install ros-indigo-slam-gmapping
```

– You may need to run `sudo apt-get update` before that to update package repositories list
Run gmapping

• Now move the robot using teleop

   $ roslaunch turtlebot_teleop keyboard_teleop.launch

• Check that the map is published to the topic /map

   $ rostopic echo /map -n1

• Message type is nav_msgs/OccupancyGrid

• Occupancy is represented as an integer with:
  – 0 meaning completely free
  – 100 meaning completely occupied
  – the special value -1 for completely unknown
map_server

- **map_server** allows you to load and save maps
- To install the package:

  ```
  $ sudo apt-get install ros-indigo-map-server
  ```

- To save dynamically generated maps to a file:

  ```
  $ rosrun map_server map_saver [-f mapname]
  ```

- **map_saver** generates the following files in the current directory:
  - **map.pgm** - the map itself
  - **map.yaml** - the map’s metadata
rviz

• **rviz** is a ROS 3D visualization tool that lets you see the world from a robot's perspective

```bash
$ rosrun rviz rviz
```
rviz Useful Commands

• Use right mouse button or scroll wheel to zoom in or out
• Use the left mouse button to pan (shift-click) or rotate (click)
rviz Displays

• The first time you open rviz you will see an empty 3D view
• On the left is the Displays area, which contains a list of different elements in the world, that appears in the middle
  – Right now it just contains global options and grid
• Below the Displays area, we have the Add button that allows the addition of more elements
# rviz Displays

<table>
<thead>
<tr>
<th>Display name</th>
<th>Description</th>
<th>Messages Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Axes</td>
<td>Displays a set of Axes</td>
<td></td>
</tr>
<tr>
<td>Effort</td>
<td>Shows the effort being put into each revolute joint of a robot</td>
<td><code>sensor_msgs/JointStates</code></td>
</tr>
<tr>
<td>Camera</td>
<td>Creates a new rendering window from the perspective of a camera, and overlays the image on top of it</td>
<td><code>sensor_msgs/Image</code>, <code>sensor_msgs/CameraInfo</code></td>
</tr>
<tr>
<td>Grid</td>
<td>Displays a 2D or 3D grid along a plane</td>
<td></td>
</tr>
<tr>
<td>Grid Cells</td>
<td>Draws cells from a grid, usually obstacles from a costmap from the navigation stack</td>
<td><code>nav_msgs/GridCells</code></td>
</tr>
<tr>
<td>Image</td>
<td>Creates a new rendering window with an Image</td>
<td><code>sensor_msgs/Image</code></td>
</tr>
<tr>
<td>LaserScan</td>
<td>Shows data from a laser scan, with different options for rendering modes, accumulation, etc</td>
<td><code>sensor_msgs/LaserScan</code></td>
</tr>
<tr>
<td>Map</td>
<td>Displays a map on the ground plane</td>
<td><code>nav_msgs/OccupancyGrid</code></td>
</tr>
</tbody>
</table>
## rviz Displays

<table>
<thead>
<tr>
<th>Display name</th>
<th>Description</th>
<th>Messages Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Markers</td>
<td>Allows programmers to display arbitrary primitive shapes through a topic</td>
<td><code>visualization_msgs/Marker</code> <code>visualization_msgs/MarkerArray</code></td>
</tr>
<tr>
<td>Path</td>
<td>Shows a path from the navigation stack</td>
<td><code>nav_msgs/Path</code></td>
</tr>
<tr>
<td>Pose</td>
<td>Draws a pose as either an arrow or axes</td>
<td><code>geometry_msgs/PoseStamped</code></td>
</tr>
<tr>
<td>Point Cloud(2)</td>
<td>Shows data from a point cloud, with different options for rendering modes, accumulation, etc</td>
<td><code>sensor_msgs/PointCloud2</code></td>
</tr>
<tr>
<td>Odometry</td>
<td>Accumulates odometry poses from over time</td>
<td><code>nav_msgs/Odometry</code></td>
</tr>
<tr>
<td>Range</td>
<td>Displays cones representing range measurements from sonar or IR range sensors</td>
<td><code>sensor_msgs/Range</code></td>
</tr>
<tr>
<td>RobotModel</td>
<td>Shows a visual representation of a robot in the correct pose (as defined by the current TF transforms)</td>
<td></td>
</tr>
<tr>
<td>TF</td>
<td>Displays the tf transform hierarchy</td>
<td></td>
</tr>
</tbody>
</table>
ROS Services

• The next step is to learn how to load the map into the memory in our own code
  – So we can use it to plan a path for the robot
• For that purpose we will use a ROS service called static_map provided by the map_server node
ROS Services

• Services are just synchronous remote procedure calls
  – They allow one node to call a function that executes in another node

• We define the inputs and outputs of this function similarly to the way we define new message types

• The server (which provides the service) specifies a callback to deal with the service request, and advertises the service.

• The client (which calls the service) then accesses this service through a local proxy
ROS - Lecture 6

ROS tf system
Get robot’s location on map

Lecturer: Roi Yehoshua
riyeho@gmail.com
What is tf?

• A robotic system typically has many coordinate frames that change over **time**, such as a world frame, base frame, gripper frame, head frame, etc.

• **tf** is a transformation system that allows making computations in one frame and then transforming them to another at any desired point in time

• tf allows you to ask questions like:
  – What is the current pose of the base frame of the robot in the map frame?
  – What is the pose of the object in my gripper relative to my base?
  – Where was the head frame relative to the world frame, 5 seconds ago?
ROS navigation stack
Costmaps
Localization
Sending goal commands (from rviz)

Lecturer: Roi Yehoshua
roiyeho@gmail.com
One of the most basic things that a robot can do is to move around the world. To do this effectively, the robot needs to know where it is and where it should be going. This is usually achieved by giving the robot a map of the world, a starting location, and a goal location. In the previous lesson, we saw how to build a map of the world from sensor data. Now, we’ll look at how to make your robot autonomously navigate from one part of the world to another, using this map and the ROS navigation packages.
ROS Navigation Stack

- [http://wiki.ros.org/navigation](http://wiki.ros.org/navigation)
- The goal of the navigation stack is to move a robot from one position to another position safely (without crashing or getting lost)
- It takes in information from the odometry and sensors, and a goal pose and outputs safe velocity commands that are sent to the robot
- [ROS Navigation Introductory Video](#)
# Navigation Stack Main Components

<table>
<thead>
<tr>
<th>Package/Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>map_server</td>
<td>offers map data as a ROS Service</td>
</tr>
<tr>
<td>gmapping</td>
<td>provides laser-based SLAM</td>
</tr>
<tr>
<td>amcl</td>
<td>a probabilistic localization system</td>
</tr>
<tr>
<td>global_planner</td>
<td>implementation of a fast global planner for navigation</td>
</tr>
<tr>
<td>local_planner</td>
<td>implementations of the Trajectory Rollout and Dynamic Window approaches to local robot navigation</td>
</tr>
<tr>
<td>move_base</td>
<td>links together the global and local planner to accomplish the navigation task</td>
</tr>
</tbody>
</table>
Install Navigation Stack

• The navigation stack is not part of the standard ROS Indigo installation
• To install the navigation stack type:

```bash
$ sudo apt-get install ros-indigo-navigation
```
Navigation Stack Requirements

Three main hardware requirements

- The navigation stack can only handle a differential drive and holonomic wheeled robots
  - It can also do certain things with biped robots, such as localization, as long as the robot does not move sideways

- A planar laser must be mounted on the mobile base of the robot to create the map and localization
  - Alternatively, you can generate something equivalent to laser scans from other sensors (Kinect for example)

- Its performance will be best on robots that are nearly square or circular
Navigation Planners

• Our robot will move through the map using two types of navigation—global and local

• The **global planner** is used to create paths for a goal in the map or a far-off distance

• The **local planner** is used to create paths in the nearby distances and avoid obstacles
Global Planner

- **NavFn** provides a fast interpolated navigation function that creates plans for a mobile base
- The global plan is computed before the robot starts moving toward the next destination
- The planner operates on a costmap to find a minimum cost plan from a start point to an end point in a grid, using Dijkstra’s algorithm
- The global planner generates a series of waypoints for the local planner to follow
Local Planner

• Chooses appropriate velocity commands for the robot to traverse the current segment of the global path
• Combines sensory and odometry data with both global and local cost maps
• Can recompute the robot's path on the fly to keep the robot from striking objects yet still allowing it to reach its destination
• Implements the Trajectory Rollout and Dynamic Window algorithm
Trajectory Rollout Algorithm

Taken from ROS Wiki [http://wiki.ros.org/base_local_planner](http://wiki.ros.org/base_local_planner)
Trajectory Rollout Algorithm

1. Discretely sample in the robot's control space \((dx,dy,d\theta)\)
2. For each sampled velocity, perform forward simulation from the robot's current state to predict what would happen if the sampled velocity were applied for some (short) period of time
3. Evaluate each trajectory resulting from the forward simulation, using a metric that incorporates characteristics such as: proximity to obstacles, proximity to the goal, proximity to the global path, and speed
4. Discard illegal trajectories (those that collide with obstacles)
5. Pick the highest-scoring trajectory and send the associated velocity to the mobile base
6. Rinse and repeat
OpenCV

- Open Source Computer Vision Library
- Contains efficient, well-tested implementations of many popular computer vision algorithms
- Created/Maintained by Intel
- Routines focused on real time image processing and 2D + 3D computer vision
- [http://docs.opencv.org/2.4/index.html](http://docs.opencv.org/2.4/index.html)
- [http://docs.opencv.org/3.1.0/examples.html](http://docs.opencv.org/3.1.0/examples.html)
- (examples)
ROS and OpenCV

- ROS passes images in its own `sensor_msgs/Image` message
- `cv_bridge` is a ROS package that provides functions to convert between ROS `sensor_msgs/Image` messages and the objects used by OpenCV
Acquiring Images

• Images in ROS are sent around the system using the sensor_msgs/Image message type
• To have images stream into our nodes, we need to subscribe to a topic where they are being published
• Each robot will have its own method for doing this, and names may vary
• Use rostopic list to find out what topics contain the robot’s camera data