Introduction to ROS: the Robot Operating System

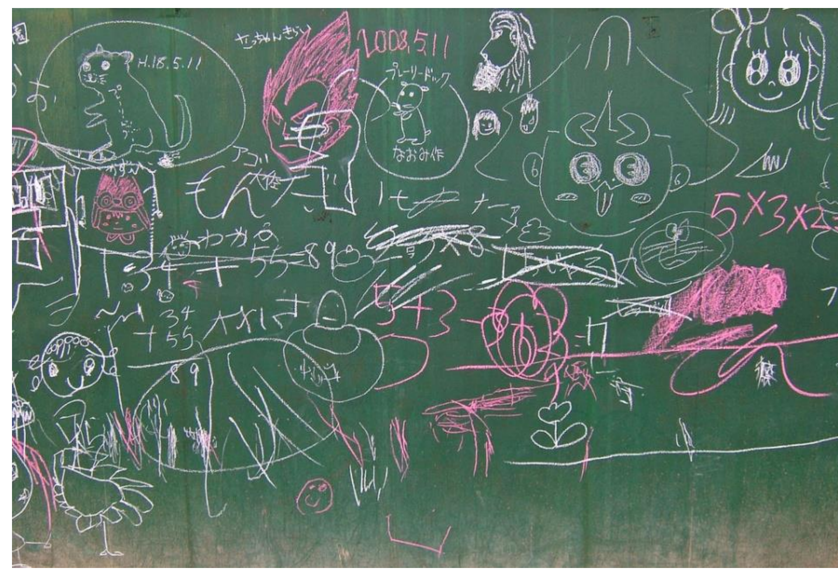
Arthur Richards

ROS?

- The ***Robot Operating System***
  - But not really an operating system

- Linux tools connecting robots and software
  - Messaging standards

- ***Blackboard architecture***



Reasons I like ROS

- It’s becoming a standard with lots of support
  - Drones – Rovers – Arms
- It helped me do something with vision
- It enables me to exploit others’ work
- It takes care of lots of housekeeping

Things I don't like about ROS

- It's Linux or nothing
- They keep upgrading it! Support lags a bit
- Asynchronous timing via callbacks
  - Can induce latency and timing issues
- Steep initial learning curve

Versions

- We’re going to use ROS “kinetic”
  – Latest is “lunar”
- Differences are mainly in build tools
  – We’ll stick with Python to avoid problems
  – Doing it in C++ is way more pain

Exercise 1

- Set up your ROS workspace
  - You’ll only have to do this once
- Learn to live with ***catkin***
  - ROS’s favourite package and build tool
  - Magic spells and potions here...

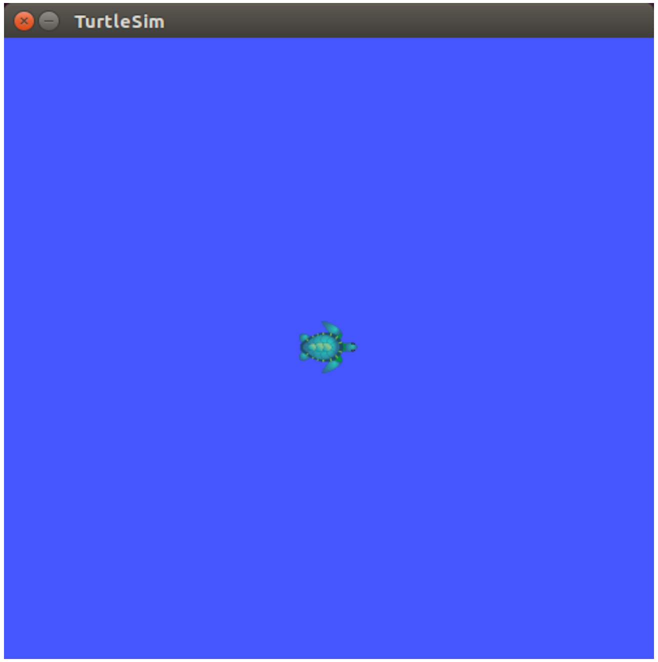
Nodes, Topics and Services

- A **node** is a programme that talks ROS
- A **topic** is a channel for sending messages between nodes
  - Nodes can **publish** to a topic or **subscribe** to a topic
- A **service** is a request/response interface between nodes
  - Nodes can **provide** services or **call** services

Topics versus Services

- Topic is asynchronous, one way data transfer
  - I can publish it when I like
  - Think of it as a letter or email
- Service is synchronous, two way data transfer
  - I call and you respond
  - Call and response can both contain data
  - Think of it as a phone call
- Topics are more commonly used
  - ***Actions***: non-blocking services via topics (hold on…)

Exercise 2: Turtle Driving



- Start a roscore:
  `roscore`

- Tip: Ctrl+Shift+T opens new tab in terminal

- Start a turtle simulator:
  `rosrun turtlesim turtlesim_node`
  – The “turtlesim_node” program lives in the package “turtlesim”

What’s available?

```
rostopic list

rostopic info /turtle1/cmd_vel

rosmsg show geometry_msgs/Twist

rostopic echo /turtle1/pose

rostopic hz /turtle1/pose
```

Send a command to the turtle

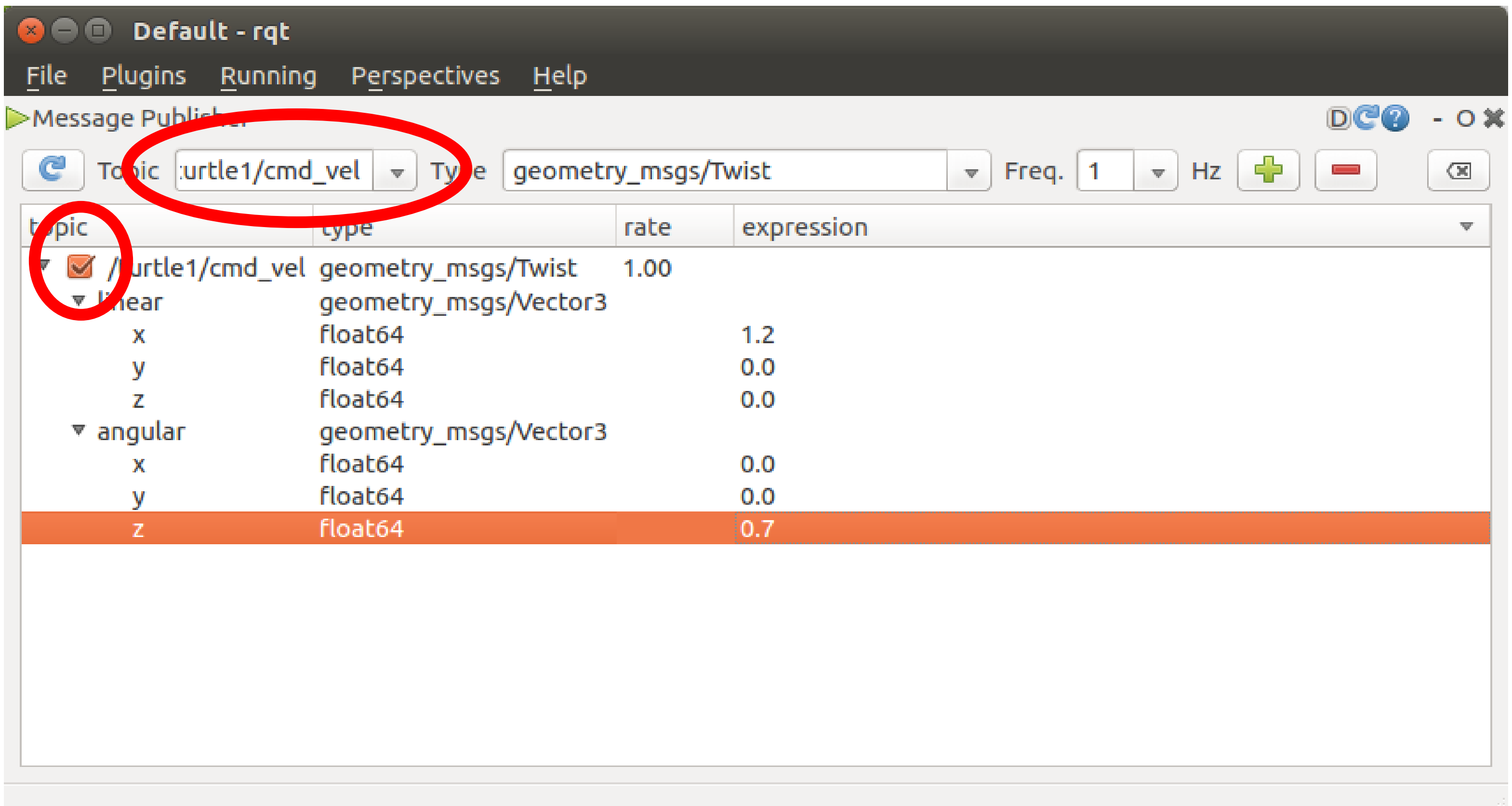
- Ugly!

```
rostopic pub /turtle1/cmd_vel
  geometry_msgs/Twist '{linear:  {x: 1.2, y:
  0.0, z: 0.0}, angular: {x: 0.0,y: 0.0,z:
  0.2}}'
```

  - *rostopic pub*: command to publish a message
  - */turtle1/cmd_vel*: topic where turtle gets commands
  - *geometry_msgs/Twist*: message type *Twist* from package *geometry_msgs*
  - ‘{linear...’: YAML syntax for parts of message

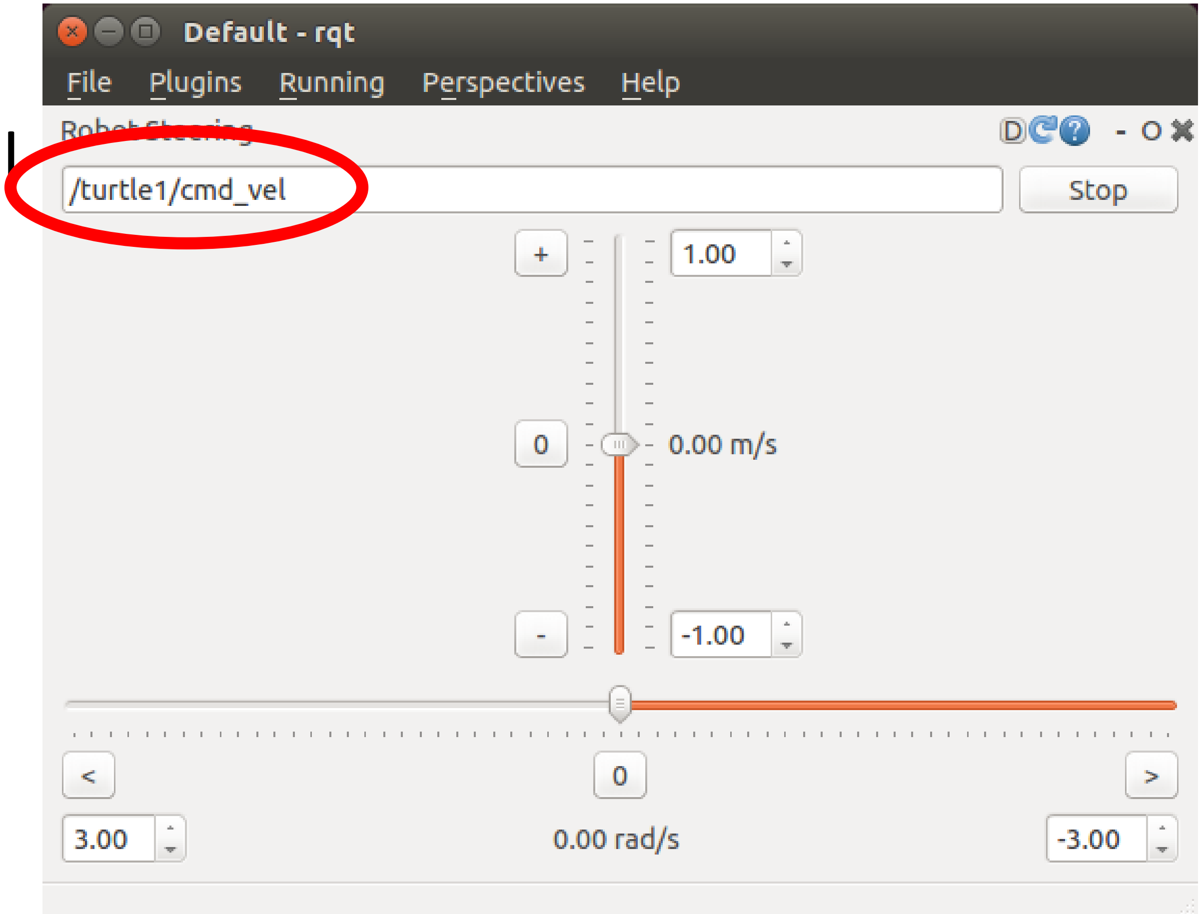
RQT - Publishing

- `rqt` – a very useful interface
  - ***Message publisher*** for all types of message



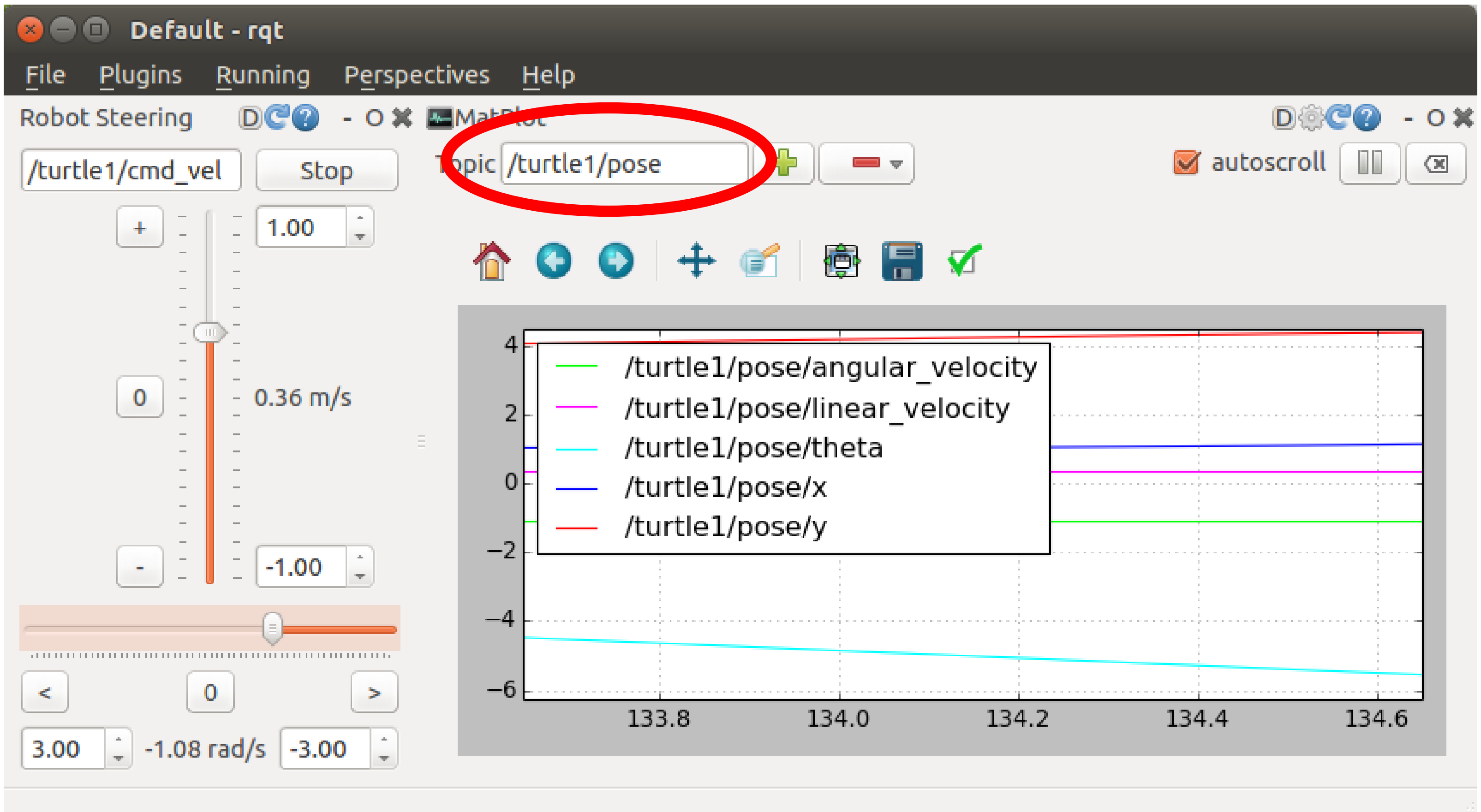
RQT - Steering

- `rqt` – a very useful interface
  - ***Steering*** tool
  - Publishes *Twist* from GUI



RQT - Plotting

- View real-time data plots



Exercise 3: Packages

- Can download packages (typically as source) from researchers' pages
- Make our own
  - `cd ~\catkin_ws\src`
  - `catkin_create_pkg ros_course std_msgs rospy turtlesim`
  - `cd ..`
  - `catkin_make`

Check ROS finds our package

`rospack list`

Terminal

```
qt_gui /opt/ros/indigo/share/qt_gui
qt_gui_cpp /opt/ros/indigo/share/qt_gui_cpp
qt_gui_py_common /opt/ros/indigo/share/qt_gui_py_common
random_numbers /opt/ros/indigo/share/random_numbers
resource_retriever /opt/ros/indigo/share/resource_retriever
robot_state_publisher /opt/ros/indigo/share/robot_state_publisher
ros_course /home/netlab/ag-richards/catkin_ws/src/ros_course
rosbag /opt/ros/indigo/share/rosbag
rosbag_migration_rule /opt/ros/indigo/share/rosbag_migration_rule
rosbag_storage /opt/ros/indigo/share/rosbag_storage
rosbash /opt/ros/indigo/share/rosbash
rosboost_cfg /opt/ros/indigo/share/rosboost_cfg
rosbuild /opt/ros/indigo/share/rosbuild
rosclean /opt/ros/indigo/share/rosclean
rosconsole /opt/ros/indigo/share/rosconsole
rosconsole_bridge /opt/ros/indigo/share/rosconsole_bridge
roscpp /opt/ros/indigo/share/roscpp
roscpp_serialization /opt/ros/indigo/share/roscpp_serialization
roscpp_traits /opt/ros/indigo/share/roscpp_traits
roscpp_tutorials /opt/ros/indigo/share/roscpp_tutorials
roscreate /opt/ros/indigo/share/roscreate
rosgraph /opt/ros/indigo/share/rosgraph
rosgraph_msgs /opt/ros/indigo/share/rosgraph_msgs
roslang /opt/ros/indigo/share/roslang
```

Exercise 4: A first publisher node

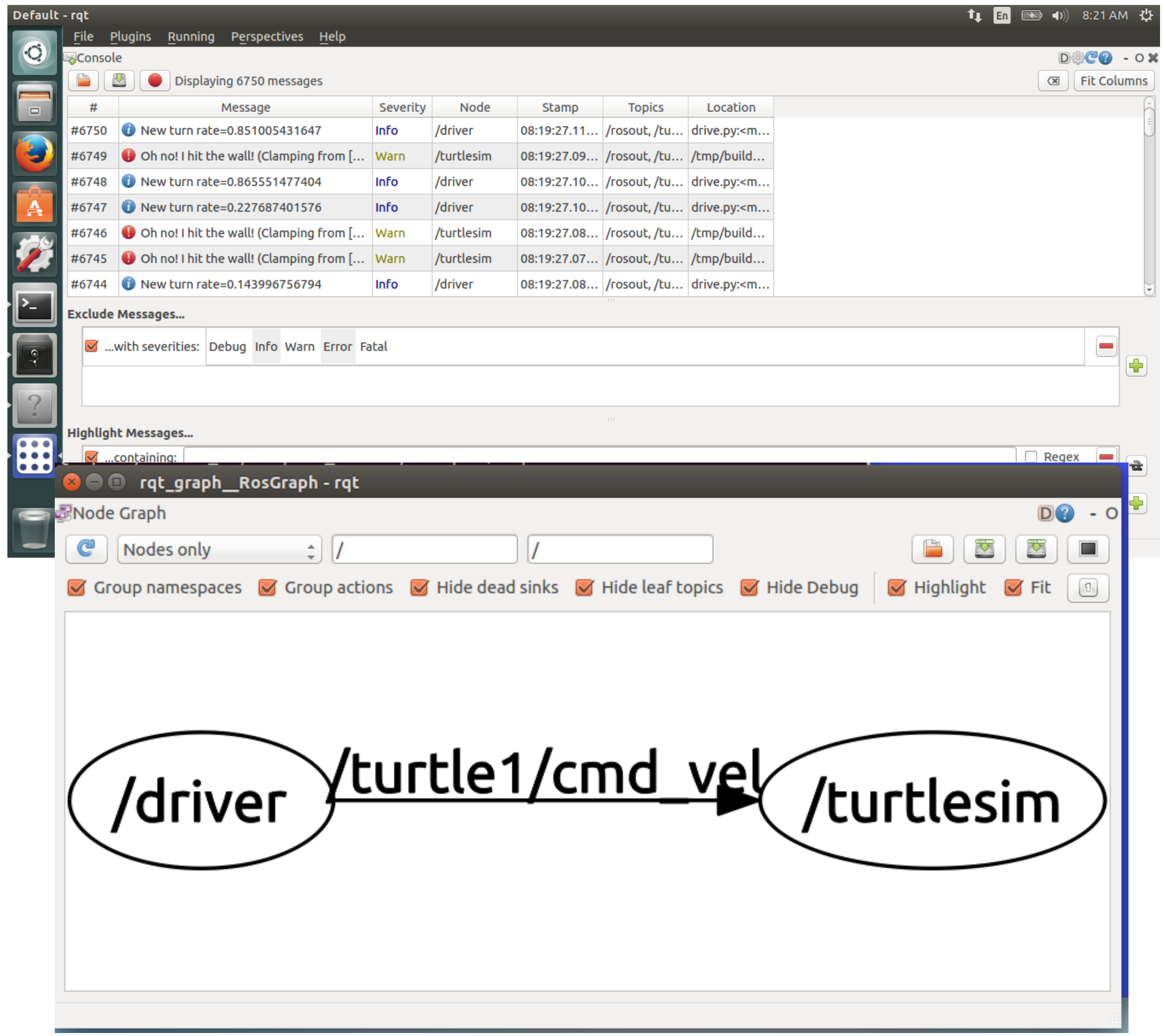
```
#!/usr/bin/python
import rospy
from geometry_msgs.msg import Twist
import random

# set up a publisher
pub = rospy.Publisher('turtle1/cmd_vel', Twist, queue_size=1)
# start the node
rospy.init_node('driver')
# will be updating at 2 Hz
r = rospy.Rate(2)

while not rospy.is_shutdown():
   # make a blank velocity message
   msg = Twist()
   # pick a random direction
   msg.angular.z = 2*(random.random() - 0.5)
   # constant speed
   msg.linear.x = turtle_speed
   # publish it
   pub.publish(msg)
   # show a message
   rospy.loginfo("New turn rate=%s"%msg.angular.z)
   # wait for next time
   r.sleep()
```

Running the publisher

- "`chmod +x`" your program so it's executable
- Start your roscore and turtle
- Run your node: `./drive.py`

More snooping with RQT

- View the ***console*** in rqt
- `rqt_graph`



Exercise 5: First subscriber node

```python
#!/usr/bin/python
import rospy
from geometry_msgs.msg import Twist
from turtlesim.msg import Pose
from math import sqrt

# start the node
rospy.init_node('listen')

# callback for pose does all the work
def pose_callback(data):
  rospy.loginfo("x is now %f" % data.x)

# and the subscriber
rospy.Subscriber("turtle1/pose",Pose,pose_callback)
rospy.spin()
```

Exercise 6: Altogether now...

```
#!/usr/bin/python
import rospy
from geometry_msgs.msg import Twist
from turtlesim.msg import Pose
from math import sqrt

# start the node
rospy.init_node('bounce')
# set up a publisher
pub = rospy.Publisher('turtle1/cmd_vel', Twist, queue_size=3)

# callback for pose does all the work
def poseCallback(data):
  radius = sqrt((data.x-5.0)**2 + (data.y-5.0)**2)
  rospy.loginfo("radius is now %f" % radius)
  turn_rate = 0.3*(radius - 4.0)
  msg = Twist()
  msg.linear.x = 0.5
  msg.angular.z = turn_rate
  pub.publish(msg)

# and the subscriber
rospy.Subscriber("turtle1/pose",Pose,poseCallback)
rospy.spin()
```

- Method 1

Exercise 6: Altogether now...

```python
#!/usr/bin/python
import rospy
from geometry_msgs.msg import Twist
from turtlesim.msg import Pose
from math import sqrt

# start the node
rospy.init_node('bounce')
# set up a publisher
pub = rospy.Publisher('turtle1/cmd_vel', Twist, queue_size=3)

# initialize global
radius = 4.0

# callback for pose does all the work
def poseCallback(data):
  global radius
  radius = sqrt((data.x-5.0)**2 + (data.y-5.0)**2)
  rospy.loginfo("radius is now %f" % radius)

# start the subscriber
rospy.Subscriber("turtle1/pose",Pose,poseCallback)

# main control loop
r = rospy.Rate(10)
while not rospy.is_shutdown():
  turn_rate = 0.3*(radius - 4.0)
  msg = Twist()
  msg.linear.x = 0.5
  msg.angular.z = turn_rate
  pub.publish(msg)
  r.sleep()
```

- Method 2

Exercise 6: Altogether now...

```python
#!/usr/bin/python
import rospy
from geometry_msgs.msg import Twist
from turtlesim.msg import Pose
from math import sqrt

class TurtleControlNode:

  def __init__(self):
    self.radius = 4.0
    # start the node
    rospy.init_node('loop_tidy')
    # set up a publisher
    self.pub = rospy.Publisher('turtle1/cmd_vel', Twist, queue_size=3)
    # rate and control
    self.rate = rospy.Rate(10)
    self.msg = Twist()

  def poseCallback(self,data):
    self.radius = sqrt((data.x-5.0)**2 + (data.y-5.0)**2)
    rospy.loginfo("radius is now %f" % self.radius)

  def run(self):
    # start the subscriber
    rospy.Subscriber("turtle1/pose",Pose,self.poseCallback)
    # main control loop
    while not rospy.is_shutdown():
      turn_rate = 0.3*(self.radius - 4.0)
      self.msg.linear.x = 0.5
      self.msg.angular.z = turn_rate
      self.pub.publish(self.msg)
      self.rate.sleep()

if __name__=='__main__':
  t = TurtleControlNode()
  t.run()
```

- Method 2.1

Observations

- We're being lazy – just running the nodes rather than using "rosrun"
  - Coming back to this later
- Moved from "print" to "rospy.loginfo"
  - This will help us out later when we run lots of nodes
- Notice the message is a structure
  - ROS enables you to define custom messages
  - For my money, it's a pain – next example just uses array of numbers

Exercise 7: Parameters

- ROS runs a parameter server
  - Nodes can get or set parameters from the server
    - Default values can be provided in the code
  - You can use a command line tool rosparam to interact with the server manually
  - Can set parameters in launch files – coming soon...

- Slower form of data transfer than topics

Parameter Example

- Use parameter to set turtle speed

```
#!/usr/bin/python
import rospy
from geometry_msgs.msg import Twist
import random

# set up a publisher
pub = rospy.Publisher('turtle1/cmd_vel', Twist, queue_size=1)
# start the node
rospy.init_node('driver')
# will be updating at 2 Hz
r = rospy.Rate(2)

# get name for topic
cmd_vel_name = rospy.resolve_name('turtle1/cmd_vel')
# show it
rospy.loginfo("Publishing to topic %s" % cmd_vel_name)

# get speed from parameter
turtle_speed = rospy.get_param('turtle1/turtle_speed',1.0)

while not rospy.is_shutdown():
    # make a blank velocity message
    msg = Twist()
    # pick a random direction
    msg.angular.z = 2*(random.random() - 0.5)
    # constant speed
    msg.linear.x = turtle_speed
    # publish it
    pub.publish(msg)
    # show a message
    rospy.loginfo("New turn rate=%s"%msg.angular.z)
    # wait for next time
    r.sleep()
```

Exercise 8: Task

- Program the turtle to reverse if it hits the wall

- Hints
  - Difficult to do anything faster than 10Hz
    - Use `rostopic hz` to check frequency
    - Down-sample if anything comes in too quick
  - Watch for immediate retrigger of reverse

So far...

- Learnt about ROS
  - Topics, packages and parameters
- Programmed ROS fundamentals in Python
  - Publishing and subscribing
- Used RQT to interact with ROS environment

ROS Names

- All topics, nodes, services and parameters are identified by their **names**
- Names can be either absolute or relative to the **current namespace**
  - Rather like a current directory
  - “/bob/topic” is **absolute** name
  - “bob/topic” is **relative** name

- Two ways to exploit:
  - Changing namespace great for multi-robot work
  - **Remapping** names good for node re-use

Current namespace

- `rosrun` *`blah`*
  - "/bob/topic" maps to "/bob/topic"
  - "bob/topic" also maps to "/bob/topic"
- `ROS_NAMESPACE=fred rosrun` *`blah`*
  - "/bob/topic" still maps to "/bob/topic"
  - "bob/topic" now maps to "/fred/bob/topic"
- Easily confused: use "rospy.resolve_name" to view where your node is looking
- Also useful if running duplicate nodes with same name

Remapping

- Namespaces are great for multi-robot stuff and housekeeping, but a little limited
  - Imagine if you had hardcoded filenames!
- Remapping enables significant flexibility
  - Re-use of nodes for different purposes
  - `rosrun` *`blah blah`* `/bob/topic:=/fred/topic`
- Now we have to be good and use `rosrun`

Why all the fuss?

- Because you can re-use code without having to change the topic pointers...
  - ... so ***no need to touch the source code*** at all!

- Get one turtle running, then:

```
ROS_NAMESPACE=/t2 rosrun turtlesim
  turtlesim_node
```

More remaps and namespaces

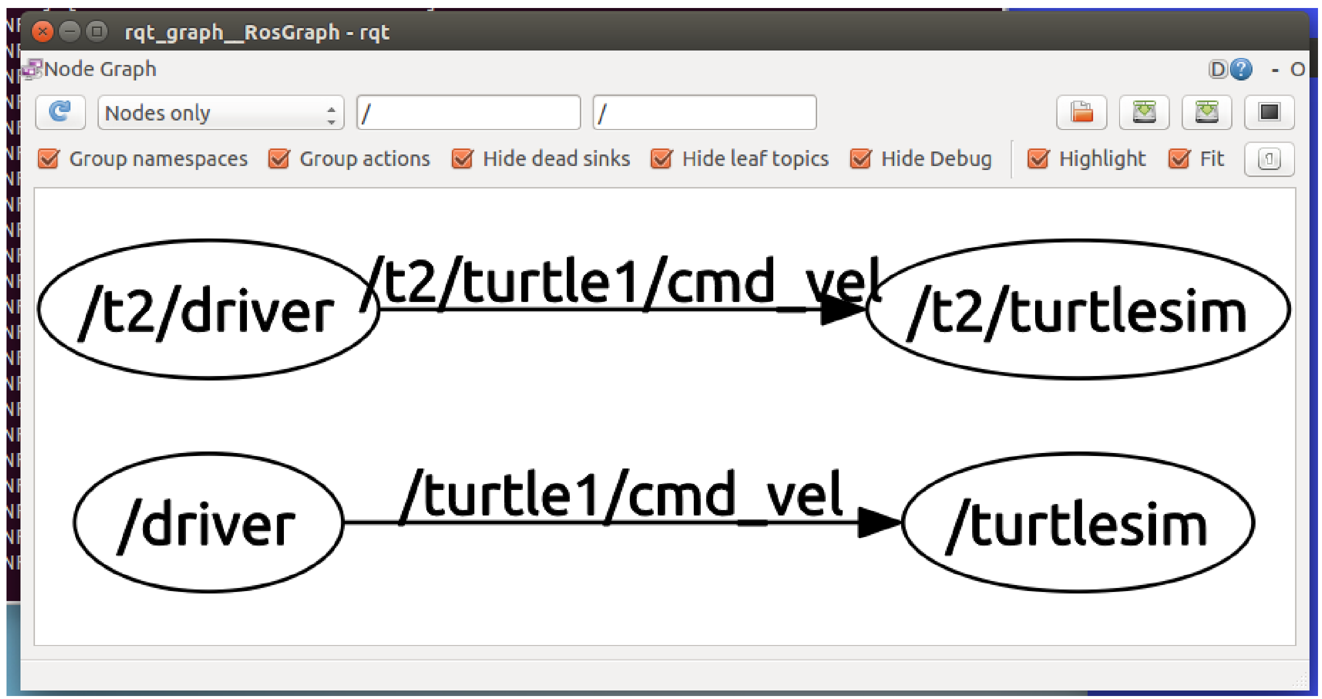
- Kill the second turtle and try this:

```
ROS_NAMESPACE=/t2 rosrun turtlesim
  turtlesim_node
  /t2/turtle1/cmd_vel:=/turtle1/cmd_
  vel
```

- What's going on here? Use rqt_graph...

Exercise 9: Two Turtles

- You should be able to use – unmodified – your turtle control code for each turtle
  - Do it by remap or by namespace – try both



Launch Files

- ROS quickly consumes many terminals
- Free your fingers with **launch files**!
- XML format enables:
  - Launching multiple nodes with one command
  - Specifying namespaces for nodes
  - Grouping nodes in common namespace
  - Remapping names
  - Including other launch files

Two nodes at once

```
<launch>
  <node name="turtle1" pkg="turtlesim" type="turtlesim_node" />
  <node name="control1" pkg="ros_course" type="“drive.py" />
</launch>
```

- Kill everything, including roscore, then run:

`roslaunch ros_course myturtle.launch`

- Observations
  - `roslaunch` will start a `roscore` if needed
  - Processes all in one window → print data not seen
  - Logging via rqt console now imporant

Launch with names

- Launch in a namespace

```
<launch>
   <group ns="bob">
       <node name="turtle1" pkg="turtlesim" type="turtlesim_node" />
       <node name="control1" pkg="ros_course" type="drive.py" />
   </group>
</launch>
```

- Launch with a remap

```
<launch>
   <group ns="bob">
       <node name="turtle1" pkg="turtlesim" type="turtlesim_node" />
   </group>
   <node name="control1" pkg="ros_course" type="drive.py">
       <remap from="/turtle1/cmd_vel" to="/bob/turtle1/cmd_vel" />
   </node>
</launch>
```

Task

- Write a launch file for the two turtles, each with your bounce-off-the-wall controller

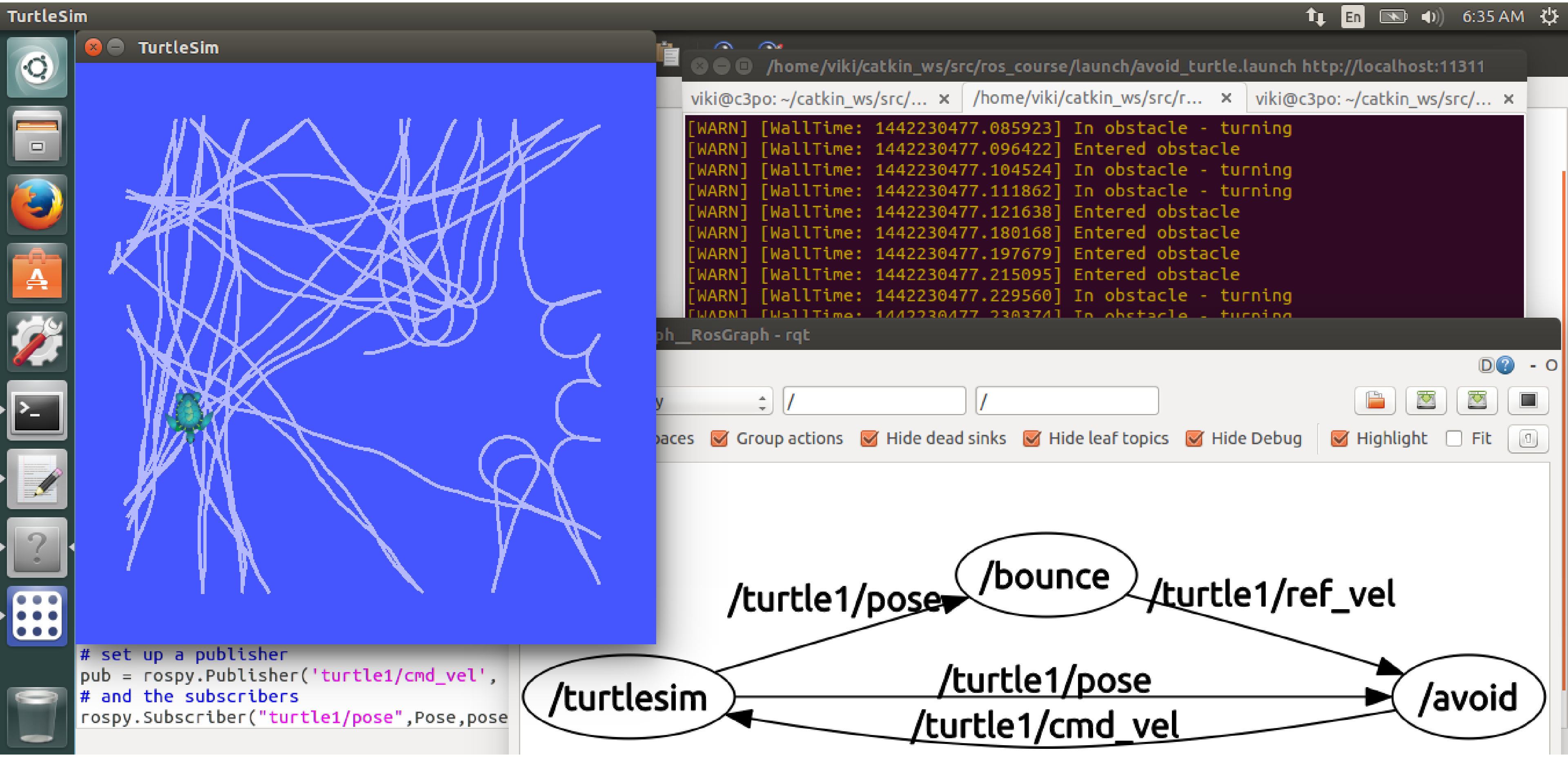
Exercise 15: Task

- Write a new node that intercepts the turtle command
  - If outside an obstacle at (5,5) with radius 2, command passed on un-changed
  - If inside the obstacle, modify the command to get out again

- Adding a behaviour to the system by adding nodes and intercepting topics
  ➔ a ***modular*** approach

Exercise 11: Avoidance

- Write a new node that intercepts the turtle



Exercise 12: Communications

- Exchange ROS messages across different computers on the network
  - Good for remote control

- Only one "master" runs roscore
  - *Every* PC needs its own IP address in ROS_IP
  - If not master, need master settings in ROS_MASTER_URI

Summary

- Core ROS ideas:
  - Packages; topics; nodes; parameters
    - We’ve skipped services
  - Publishing and subscribing
  - RQT; console; node graph; rostopic; rospack
- Use namespaces and remaps to use nodes for different purposes
- Use launch files to build integrated systems