Robotics

programming and simulation

Rico A. R. Picone
Department of Mechanical Engineering
Saint Martin's University

06 June 2020

Contents

Part I

Introduction to Robotics

01

Introduction

Lecture 01.01 Defining robots

What is a robot? Due to the wide variety of existing robots, it can be challenging to identify the gist of the term, but here are some I claim are essential:

mechanical
presence
environment
sensing
acting
intelligence
autonomous

mechanicality A robot has a *mechanical presence* in an *environment*.
sensitivity A robot can partially *sense* its environment.
potency A robot can *act* on its environment.
intelligence A robot can act *intelligently*.
artificiality A robot is designed by humans.[1]
autonomy A robot is *autonomous*, acting at least partially without direct human intervention.

These are each *necessary* conditions for a device to be a robot. However, I claim they collectively are *sufficient* conditions. In other words, a device must have all these qualities to be a robot, and if it is missing any, it is not a robot. Although this definition of a robot may be flawed, or may change in the future, it gives us a useful device for discerning if a given device is a robot or not. Furthermore, it allows us to determine which qualities the device would need to have to be considered a robot.

Example 01.01-1 Is it a robot?

Classify each of the following as robot or not-robot. If it is not a robot, list the missing qualities. Comment on ambiguities.

a. A Roomba vacuum cleaner.
b. A desktop computer.
c. A home heating/cooling system.
d. A toilet tank.
e. A car.
f. A cell phone.
g. A simulation of a robot.
h. A 3D printer.

[1] Or, at least, if it was designed by a robot, which was designed by a robot, etc.—the original robot must have been designed by a human.

We will explore the meaning of each of the qualities of a robot in the following lectures. For now, let's pause a moment to consider the "why" of

robotics.

01.01.1 Why robots

Why do we make robots? Reasons include:

robots are cool One motivating factor seems to be the awe we experience when creating something that appears life-like. In fact, creating *artificial life* has been one of the explicit goals of some roboticists. Other roboticists have been inspired by (biological) life to create more effective robots; this field is called *biomimicry*.

artificial life

biomimicry

biology is cool A related reason to robot is that while we're creating artificial life and biomimetic robots, we frequently "reverse engineer" biological life, which yields a deeper grokking of biology. We can develop robots in service of biology.

robots help make things we like Robots help manufacture cars, airplanes, food, computers, cell phones, and many other things we like.

robots do dangerous work Instead of humans doing certain dangerous work, like cleaning up a toxic chemical spill, robots can take our places.

robots do boring work Especially in manufacturing, but also in household chores, robots can replace humans in work that is repetitive and boring. This has the potential to free up human time for activity we find more meaningful.

robots can do precise work Robot-assisted surgery, for instance, allows a human surgeon to guide a robot through a delicate procedure that requires mechanical precision beyond that of human capability.

we like money Robots give us economic advantages, which give us money. Under our estrangement in capitalism, we fetishize money not for its exchange value, but for itself.

We must address something here: despite its advantages, does not robot labor necessarily lead to the *loss of human jobs* and the further *concentration of wealth*? Yes and no. Yes, robot labor has in fact reduced human jobs and concentrated wealth, and it will continue to do so under the currect world economic system (capitalism). No, this loss of human jobs is not necessary. Under a different economic system, robot labor could have a positive impact on human being.

loss of human jobs

concentration of wealth

Lecture 01.02 Robot mechanicality

For a device to be considered a robot, it must have a mechanical presence in an environment. One immediate conclusion from this is that a *simulation* is not a robot. This does not mean we cannot simulate robots. In fact, we *must* simulate a robot to design one of any value, which is part of why we spend several chapters on just that, later in this text.

simulation

So what does it mean that a simulation is not a robot? There are two points here being emphasized.

1. Reality is much more complicated than can be simulated, and therefore even good robot simulations cannot account for everything. Reality's tough, kid!
2. Simulations of robots are great, but they can do no mechanical work.

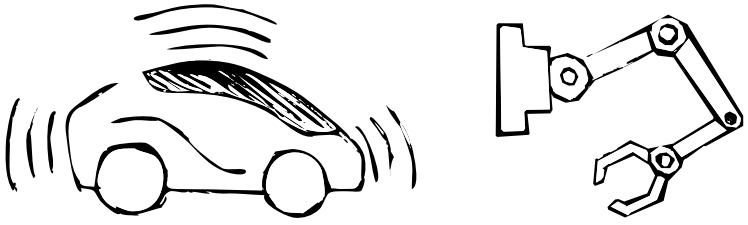
Or, put simply: a simulation doth not a robot make.

Another implication of the mechanicality of a robot is that it has *space* and therefore *matter* and *form*. What is the stuff (matter) of a robot? Most robots are made of the usual materials found in machines: metals, plastics, rubbers, and ceramics. And, of course, silocon. The form one takes depends on its function. A robot that must change its location requires a means of *locomotion*. One that must manipulate objects in the world must change its own *orientation* relative to the world.

space
matter
form
locomotion
orientation

These last two are more than simple examples. They divide the two primary types of robots: *mobile robots* and *manipulation robots*. The paradigmatic case of the former is the *self-driving car* and of the latter is the manufacturing *robot arm*. There's no reason a self-driving car can't have a robot arm (can't be both a mobile and a manipulation robot), but that's just showing off.

mobile robots
manipulation robots
self-driving car
robot arm



(a) self-driving car (b) robot arm

Figure 01.1: examples of two types of robot, (a) mobile and (b) manipulation. (PR)

01.02.1 Locomotion

locomotion

Mobile robots must do something basic to animal life: move about in an environment. Moving about, or *locomotion*, is a fascinating topic with novelty everywhere. Something that makes it challenging is that it depends on both the robot and its environment. For instance, a robot that locomotes with wheels might not be effective at navigating the terrain of a rocky hillside, and a lake even less-so.

Locomotion, then is a robot-environment problem. Some types of environments commonly considered are: on-ground (i.e. terrestrial), underground (i.e. fossorial), in-liquid (e.g. aqueous), in-gas (e.g. aerial), and space. Most robots effectively move about in only one of these. Usually, there is enough variation in each type of environment (e.g. calm versus stormy air) to render robots effective in just a subset of the types of environment listed above.

locomotion methods

Examples of *methods of locomotion* include:
• rolling • walking • jumping • stick-slipping • slithering • undulating • jet-propelling • rotary-propelling • flapping • gliding • soaring • swimming • ballooning.

locomotion devices

Examples of robotic *locomotion devices* include:
• wheels • tracks • legs • arms • tails • rockets • propellers • sails • wings • fins • magnets • cilia.

biomimicry

Locomotion is one of the fields of robotics that relies most heavily on *biomimicry*. Animals have developed incredible and unique methods of locomotion, and the study of them has been a gold mine for robotics.

actuators effectors behaviors

It is worth considering here a three-fold distinction made among *actuators*, *effectors*, and *behaviors*. Consider the aerial robot of Figure 01.2 that flies by flapping its wings. The motor actuates the wings (effectors) which produces the behavior of flapping or flying.

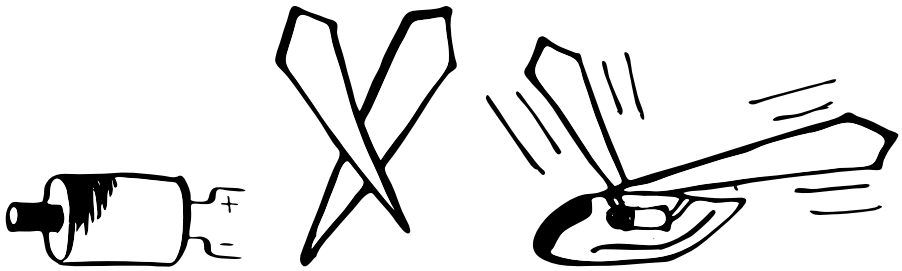
navigation

This brings us to another important consideration in mobile robotics, *navigation*. This involves several of the qualities of a robot we'll consider in the text, but the mechanical facet of navigation is that of describing spatial *location* and *orientation* through time, and the forces involved. We'll return to these considerations, which constitute the study of *mechanics*, at the end of this lecture.

location orientation mechanics

01.02.2 Manipulation

Manipulation robots move around objects in the world. Although it is not a requirement, most of the time they are themselves stationary, attached to



(a) actuator: motor (b) effector: wings (c) behavior: flapping or flying

Figure 01.2: example of how actuators, effectors, and behaviors are related. (PR)

something relatively fixed. This helps the robot move things by providing "somewhere to stand," as it were.

Manipulation robots also use actuators, effectors, and exhibit behaviors. The behavior of *grasping* is especially important for manipulation robots: by grasping an object (typically with an effector called a "gripper"), it becomes rigidly attached to the effector, the position and orientation of which is presumably known to the robot, and the robot can then manipulate the object by changing its own position and orientation.

grasping

It's hard to think a manipulation robot without an arm, a fact that jives with a survey of primates, animals known for cognition and an ability to manipulate tools. This does not mean there aren't superior ways, but that arms are, dare I say, close at hand to a human designer.

Let's consider another way of understanding the advantages of an arm for a manipulation robot. The concept of *degrees of freedom* (DOF) will help us here. Later we will consider the world's three-dimensional space in greater detail, but for now consider that an object in this space can potentially *translate* in three independent directions and *rotate* about three independent axes. Speaking a somewhat simplified mode, we can say that a robot has a degree of freedom for each independent axis along with it can translate and about which it can rotate. Returning, then, to the arm, we see it has several *joints* that allow it to increase its DOF. The jointedness of arms are the key to their excellence in manipulation: the more degrees of freedom it has to move, the more complex can its movements be.

DOF

translate
rotate

joints

There are systematic ways of classifying joints and arms in terms of DOF, which we will later consider. For now, we simply want to understand the motivation of going into a detailed analysis. The goal of analyzing joints and arms is to describe an arm's position and orientation, how to make it

move from one to another, and understanding the forces there-involved. As in the conclusion of the preceding section on locomotion, we have found ourselves concerned with matters of *mechanics*.

mechanics

01.02.3 Mechanics

Mechanics is the study of the motion of matter and the causes and effects thereof. We call the cause of motion *force*, which is typically understood to potentially produce the motion of matter. As mechanical engineers, we are interested in several sub-fields of mechanics, including fluid mechanics, solid mechanics, and rigid-body mechanics. Most of these specialized fields of study are focused on the motion and forces that cause it in specific types of material.

force

It is convenient to differentiate between two primary considerations in mechanics: *kinematics*, which mathematically describes the motion of matter and *kinetics*, which mathematically describes the forces that cause motion.

kinematics
kinetics

A famously challenging aspect of mechanics in robotics is called *inverse kinematics*, which is the study of how to "back out" the positions and orientations of a robot's parts that yield some desirable overall configuration. The quintessential example here is a robot arm: if we want the gripper to be located in a certain position and orientation, where should each of the individual joints be?

inverse kinematics

There are frequently *multiple* solutions for a given gripper configuration. This problem is exacerbated by the fact that frequently there are additional constraints on variables, yielding a system of equations and inequalities. Even worse, these equations are usually *nonlinear*. Good analytic and numerical techniques for inverse kinematics have been developed, and we will consider some later in the text.

Lecture 01.03 Robot sensitivity

A necessary part of a robot's intelligence is its ability to sense its environment. The old joke that

> *to a hammer*
> *everything looks like a nail*

is apropos and pairs well with the gibe

> *dummer than a bag of hammers*

to suggest that intelligence requires a sensitivity utterly lacking in a hammer, which, of course, makes a poor robot. But what does it mean to be sensitive? Fundamentally, it involves an interaction between a robot and itself or its environment. This interaction is called *perception* or *measurement*, which is another fascinating field of study. This is not the place to delve into the theory of measurement, but I do highly recommend doing so at some point.

perception
measurement

01.03.1 Measurement and perception

So a robot measures itself (*proprioception*) and its environment (*exteroception*). Clearly, for intelligent behavior in an environment, it must act in accordance with this measurement. In Lecture 01.05 we will consider the details of acting in accordance with a measurement, but for now, we can just acknowledge that it must be so.

proprioception
exteroception

What about itself and its environment does a robot measure? Given the mechanicality discussed in the preceding lecture, it certainly has to measure aspects of space and time: length, position, duration, velocity, acceleration, force, torque, etc. But additional quantities will be important in many applications: voltage, current, pressure, flowrate, temperature, heat, sound, light, etc.

How does a robot measure? Consider Figure 01.3. The device at the point of measurement is called a *sensor*. A sensor output is almost always a electronic *signal*: a low-power, information-bearing voltage through time. A sensor is frequently supported by electronics that provide power to the sensor, amplify the signal, or filter the signal. The output signal of the support electronics is usually connected to more electronics or a computing device that decides what to do with the measurement.

sensor
signal

Measurements are never perfectly accurate; in fact, it is a fundamental quality of measurement that

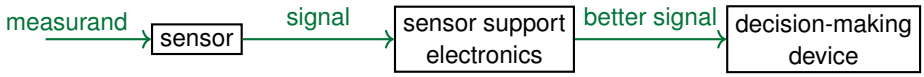


Figure 01.3: diagram of a measurement.

every measurement affects the measurand.

thermal noise

That is, measuring changes the state of the thing measured. And beyond this fundamental limit, virtually all measurements include *thermal noise* a random signal that is introduced through the microscopic motion of matter at nonzero temperature.

01.03.2 Sensors

transducer

A sensor, then, is a type of energy *transducer*, converting one form of energy into another. Since the output energy domain is normally electronic, sensors are electro-mechanical/photo/thermo/etc. transducers. This is one reason there has been such interest in materials and processes that exhibit this type of transduction. For instance, piezo-electric materials convert mechanical stress into a flow of charge (current). Research into transducers like this have been combined with *nanotechnology* to build *micro-electro-mechanical systems* (MEMS) sensors that fit on a microchip.

nanotechnology
MEMS

01.03.2.1 Types of sensors

proprioceptive
exteroceptive

Sensors that measure quantities related to the robot's own state are called *proprioceptive*. Those that measure quantities related to its environment are called *exteroceptive*.

passive
detector
active
emitter

Passive sensors measure by means of a *detector* alone. Conversely, *active sensors* use an *emitter* with a detector.

simple

A *simple sensor* is one that provides a signal that requires relatively little post-processing by the sensor support circuitry or decision-making device. Examples of simple sensors include the following.

Switches are sensors that have only two states, typically instantiated as a circuit with contacts that close or break the circuit.

Pressure or force sensors are touch sensors that are sensitive to pressure on or force through the sensor by piezoelectric transduction or resistance-based strain gauge.

Photocells are electronically resistive sensors, the resistance of which varies with light exposure; these are typically slow.

Polarizing filters polarize light such that the light not parallel to the polarizing plane is filtered.

Reflective optosensors are sensors that detect light from an *emitter* (usually an LED) that is collected by a *photodiode* or *phototransistor*. These are much faster detectors than photocells. There are two types.

Reflectance sensors require the light to reflect off an object and return to the detector.

Break beam sensors have their emitter and detector pointed at one another such that an object may interfere with the beam.

Shaft encoders encoders, are used to measure the angular position of a shaft. These are typically *optical* and *quadrature* encoders that also indicate the direction of rotation. Basically, an emitter bounces two lasers off a spinning wheel with stripes offset 90 degrees from each other.

Potentiometers (i.e. "pots" or "rheostats") are variable resistors. They often have a knob on them, the angular positions of which correspond to varying electronic resistance.

Example 01.03-1 a human body's sensors

Thinking about the human body as a very advanced robot can help us better design robots. Identify which of the above types of sensors one could say, by analogy, a human typically has.

For actionable information from a *complex sensor*, more support and potentially computation is required than for simple sensors. The following are important types. **complex**

Gyroscopes and accelerometers can be used to detect the motion and especially the orientation of a robot. Gyroscopes used to be built as macroscopic flywheels, the angular momentum of which would maintain its orientation when mounted in gimbals. Today, MEMS mimic this behavior so that gyroscopes can be inexpensively and conveniently placed on a printed circuit board (PCB).

Ultrasound (i.e. "sonar") sensors allow us to use *echolocation* in robotics. Sonars emit a chirp and measure the time-of-flight for the chirp to

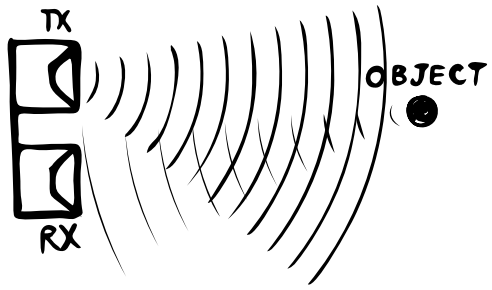


Figure 01.4: a sonar emitter (TX) transmitting a sound wave that reflects of an object and is detected by a receiver (RX). (PR)

specular reflection

return. It's great. *Specular reflection* is the reflection from the surface of an object being detected by sonar. Smooth surfaces are hard to detect because the waves can be completely reflected away from the detector. Rough surfaces are better. More sensors (in configurations called *phased arrays*) improve accuracy of sonar systems. For an illustration of sonar, see Figure 01.4.

Lasers emit coherent beams of light, some visible, others not. Laser sensors can use the same time-of-flight principles as sonar to measure distance, but must use phase-shift information for short distances (because light's pretty fast). Advantages of lasers are that they are faster than sonar, have good resolution, and have fewer specularity issues. Disadvantages include that they are much more expensive than sonar, bulky, and provide limited information (small beams!).

Cameras capture an *image* of a *scene*. Processing these images is a huge challenge.

Edge detection processing attempts to find the edges in an image.

Segmentation is the process of organizing an image into sections that correspond to an object in the image.

Model-based vision uses stored *models* to compare with features in images.

Stereo vision gives two views of one scene, adding depth and three-dimensionality to images.

01.03.3 Sensor fusion

sensor fusion

Sensor fusion is the process of combining information for several sensors. An example of sensor fusion is the fusion of gyroscopic and accelerometer data to yield an accurate estimation of the orientation of an object. Gyro-

scopes can be used to measure the three-axis angular velocity of an object very quickly (their response is fast), but in order to determine the angle, the angular velocity must be integrated to get position. Unfortunately, this is plagued by an accumulation of error through time. However, the angular position can be measured quite accurately from a three-axis accelerometer by tracking the gravitational acceleration direction. The drawback is the accelerometer is slower to respond. In short, for this application, gyroscopes are fast but lose accuracy over time and accelerometers are slow but accurate.

Enter sensor fusion. A quick response and accurate estimation of the angular orientation can be found by techniques such as the venerable *Kalman filtering*.

Kalman filtering

Lecture 01.04 Robot potency

acting A robot must be able to *act* on its environment. Common acts are in service of locomotion and manipulation, but there many others, like cleaning (e.g. vacuuming), cutting (e.g. CNC milling), and delivering material (e.g. 3D printing).

As is often the case when deepening our understanding of a device, following the flow of energy through it, a robot in this case, will help us better understand it. We start with where the robot gets its energy and follow this through its application to the environment in an action, as shown in Figure 01.5.

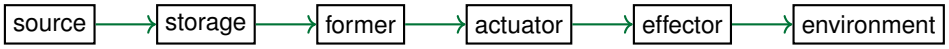


Figure 01.5: power flow through a robot.

01.04.1 Energy, its source and storage

Actions require energy, which is typically delivered on-demand from the electrical grid for stationary robots (typically manipulation arms) and delivered from an on-board battery for mobile robots. Some mobile robots can harvest energy (e.g. via solar panels), but the rate of harvest is typically much slower than is required for peak performance. Therefore, mobile **battery** robots tend to have energy limitations and *battery* technology is crucial for mobile robot development.

Box 01.1 self-driving cars and batteries

Self-driving cars are typically electric and rely on large, rechargeable batteries, typically of the lithium-ion variety. Considerations here include energy storage capacity (vehicle range), power rating (vehicle power), recharge rate (driver waiting for recharge), self-discharge rate (when chilling), specific energy (J/kg), and lifespan.

01.04.2 Electrical power forming

Before a robot applies power to the environment, it must first first transform it into the appropriate form for its actuators (considered next). The two **DC** primary forms of electrical power are *direct current* (DC) and *alternating* **AC**

current (AC). Batteries delivers DC power and the grid delivers AC. Some actuators take DC and others AC power.

In mobile robots, due to a reliance on (DC) battery power and the efficiency cost of *DC-to-AC* conversion, DC actuators are preferred. In stationary robots, like those for manipulation, AC power is plentiful, so both AC actuators and DC actuators (using *AC-to-DC* conversion) are used.

DC-to-AC

AC-to-DC

Further sub-forms of DC and AC power can also be identified, as outlined in Figure 01.6. These sub-forms come about because most of the time actuators will need to be delivered variable quantities of power. For instance, *switched DC* power or *pulse-width modulation* (PWM) can be used to deliver variable average power to an actuator: by rapidly switching DC power on and off, the actuator average power is varied. This digital electronics technique is usually less expensive than the analog *amplifiers* required for truly continuously varying DC power. A drawback of switched power is that it introduces significant high-frequency noise, which can negatively impact sensors.

switched DC
PWM

amplifiers

AC power also has sub-forms. Some high-power AC actuators require multiple *phases*, frequently three. In this case, three different signals with proper phase differences must be delivered. Other AC actuators operate at one steady-state speed per *frequency* of AC power. For changing speeds, this frequency must be varied—something achievable with a *variable-frequency drive*.

phases

VFD

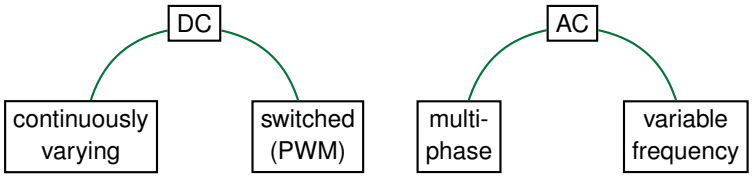


Figure 01.6: forms of electrical power for actuation.

01.04.3 Actuation

actuation

Actuation is a robot's transduction of electrical power to the proper energy domain for its corresponding effector (considered next). Usually, this device is *electromechanical*. In fact, the paradigmatic actuator is the *motor* in its 31 flavors.

electromechanical
motor

Most motors convert electrical power to *rotational* mechanical power, but some, called *linear motors*, convert to *translational* power. The fundamental mechanism in a motor is the *Lorentz force* acting on a wire sur-

linear motors
Lorentz force

rounded by a magnetic field and through which electrical current is flowing. The varieties of motors are essentially different methods of arranging for this mechanism's unfolding.

The following list describes some types of motors important for robotics.

DC motors are those that require DC power in its various forms (e.g. switched).

- **BDC** or brushed DC motors have a mechanical contact that reverses current flow. These are inexpensive but are less efficient than other types and require more maintenance due to brush wearing.
  - **PMDC** or permanent-magnet DC motors are brushed and have a background magnetic field generated by permanent magnets. These are relatively easy to model, but require feedback to control.
  - **Wound-stator** DC motors are brushed and have a background magnetic field provided by an electromagnetic coil.
- **BLDC** or brushless DC motors are actually AC motors with complex built-in electronics that allow it to act like a DC motor. These are more-expensive and require expensive electronic controllers, but require less maintenance than BDC motors.
- **Stepper** motors are DC motors that (usually) rotate a predictable amount for each DC pulse applied. The "usually" qualifier here means that for accurate position control, feedback is still required. These are fine for some position control, but are not great for continuous rotation or high speeds.

AC motors are those that require AC power in its various forms (e.g. multiphase).

- **Induction/asynchronous** AC motors generate a rotating electromagnetic field that induces current to flow through an electrical loop in the rotor (the part that spins with the shaft). Again, the Lorentz force kicks in. In steady-state, there's a difference called *slip* between the rotation rate of the electromagnetic field and that of the rotor. The speed of these motors is varied by changing the frequency of the AC power with a VFD. These are very common for large-load industrial applications.

slip

- **Synchronous** AC motors generate a rotating magnetic field on both the stator (the part that doesn't turn) via electromagnetic coils

and on the rotor via either permanent magnets or DC electromagnets. Since they do not rely on magnetic induction, the rotor field and stator have the same angular velocity in steady-state (i.e. there's no slip). If the rotor uses an electromagnet, brushes are required, with all their baggage. These can be more expensive than induction motors, but they can be also be driven by a VFD.

servomotors

Another term frequently encountered here is *servomotors*, which has two common meanings. The first is simply most any of the motors above[2] of suitable quality for precise feedback control. The second is really a package: a motor, a speed measurement device (usually an *encoder*), and a *feedback controller*. Sometimes the controller is included and other times it is sold separately.

encoder
feedback controller

mechanisms

Since the desired effect of the actuator on the effector is frequently not simply rotation in the range of speed and torque of the motor, *mechanisms* frequently comprise the final stage of the actuator. This is the stuff of mechanical engineers' dreams: gears, spools, pulleys, linkages, belts, tracks, power screws, etc.

01.04.4 Affection

It also means "to affect"—the robot's environment in this case! Most of the time, this is a mechanical interaction. And *interaction* it is: let's not forget Newton's third law here; the robot's structure and, if applicable, base, will experience reaction forces.

effectors

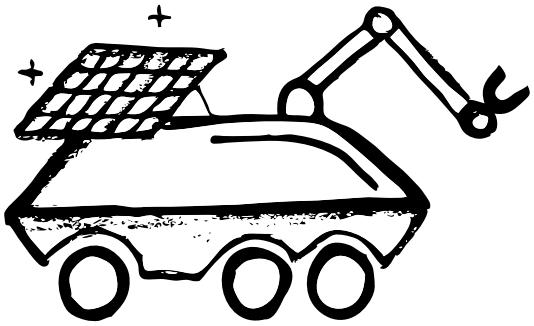
Devices actuators use to yield effects in the environment are called *effectors*,[3] which have direct interaction with the environment. Common effectors include pretty much all the locomotion devices considered previously and grippers, claws, feet, wipers, water jets, high-power lasers, sanding pads, and suction cups.

[2] A possible exception here is a stepper servomotor.

[3] Affect is usually a verb and effect usually a noun, so you might be offended by the term "effector." However, effect *can* be a verb. I looked it up.

Example 01.04-1 power flow through a solar rover bot

Consider a solar-powered rover bot for exploring the surface of Mars, illustrated below (PR). For the behaviors of (a) driving and (b) collecting a soil sample with an arm, trace the flow of power through the robot, and along the way identify energy storage elements, energy transducers, actuators, and effectors.



Lecture 01.05 Robot intelligence

Intelligence is famously challenging to define, but that won't stop us.

Definition 01.05.1: intelligence

Intelligence is the ability to map perceptions to performant actions.

Let's consider this definition specifically with regard to *robot intelligence*, a form of *artificial intelligence*. Here, *perceptions* are measurements, discussed in Lecture 01.03. *Actions* are with the robot's environment (Lecture 01.04). The remaining qualifier here is *performant*: to perform well with respect to a *metric*.

robot intelligence
artificial intelligence
perceptions
actions
performant
metric

Let's consider this last requirement in more detail, with reference to Figure 01.7. If a robot action map did not have a metric, it would offend our

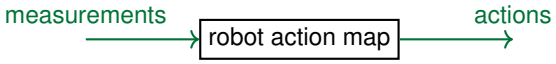


Figure 01.7: robot action map.

sense of the meaning of the term "intelligence." For instance, if a vehicle in locomotion perceived an obstacle, was mechanically capable of avoiding a collision, but did not, it would not seem intelligent. If one of its metrics is collision-avoidance, it performed poorly and looks dumb. If it didn't have such a metric in the first place, it looks not-even-dumb. Stupidly, it was lacking an obvious aspect of locomotion.

The term *goal* or *objective* is frequently used here, in the sense that for something to be intelligent it must have one. But there is something more implied in the term "goal": *intention*. It was the study of biology that first alerted us to the limitations of requiring intelligence to involve intention. Many biological systems exhibit what we would call intelligence, such as the flocking of birds as they fly together, without any sort of coordinated effort. In fact, flocking behavior can be replicated in groups of robots based on simple rules given to each robot. That is, a complex behavior that could be assigned a metric (e.g. flocking could be measured by the efficiency of group flying) can emerge without intention. This is called *emergence*, a topic that is perhaps more mysterious than is necessary.

goal
objective
intention
emergence

01.05.1 Robotic applications of artificial intelligence

Before we introduce some of the methods of artificial intelligence, it is worth considering some of its applications in robotics.

Machine vision systems process image sensor data streams to construct actionable information. This includes object detection and internal model construction.

SLAM or simultaneous localization (of the robot) and mapping (of its environment) must fuse (sensor fusion!) vision and other sensor data. AI can contribute to this process.

Path planning is the process of planning the route the robot should take through an environment. For instance, self-driving cars use map data and GPS to plan its route.

Natural language processing can make the robot responsive to human speech and allow it to construct human-understandable speech.

01.05.2 Methods of robot intelligence

The development of artificial intelligence (AI) has been so varied in method that summaries invariably and unjustly minimize entire fields of research. Therefore, instead of attempting a hierarchical organization, we will highlight a few ideas of particular interest.

symbolic manipulation

The first is that of *symbolic manipulation*. Here intelligence is understood as the use of symbolic representations and rules of inference. For instance,

> *if there is a parking spot available on the street, park;*
> *otherwise, circle the block.*

There are several symbols here, e.g. a parking spot, the street, parking, block, circling. This approach has encountered formidable challenges, such as the *signal-to-symbol problem* of going from measurement signals to appropriate symbolic representations. In the example, it is challenging to determine from sensor measurements what is a parking spot. This approach also encounters the vast ambiguities in natural language; consider the multiple potential meanings of the sentence, "There's a spot." *Natural language processing* is now an entire field of AI.

signal-to-symbol problem

natural language processing

Logical approaches to AI understand intelligence to be comprised of *reasoning*. Reasoning requires a *knowledge base* with *semantic* (i.e. meaningful) relations, so *knowledge representations* have been a significant aspect of this work. This approach features, for instance, the ability to make *plans*. In reality, nothing is certain, so logical methods have come to use *probabilistic*

logical reasoning

knowledge base

semantics

knowledge representations

planning

probabilistic reasoning

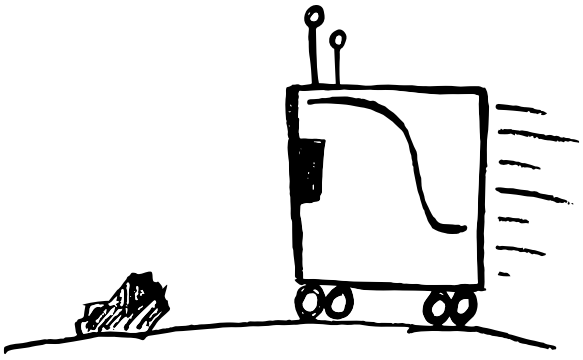


Figure 01.8: a robotic vehicle hurtling toward an obstacle. (PR)

reasoning with, for instance, *Bayesian networks*. Another approach to uncertain reasoning is *fuzzy logic* in which elements can be in a set to a degree.

Bayesian networks
fuzzy logic

Beyond the minimum requirement set forth in our definition of intelligence, it is certainly a mark of intelligence to *adapt* or *learning*: to increase intelligence from experience. So, consider again the vehicle in locomotion and perceiving an obstacle it is mechanically capable of avoiding, now illustrated in Figure 01.8. If it has as a metric to avoid obstacles, it might still hit the obstacle if its action map is insufficiently intelligent. Now consider if this map is *trained* on many obstacles such that it tweaks the map in accordance with its obstacle-avoidance metric. This would yield an action map the performance of which improves in intelligence. Taken broadly, this program is called *machine learning*, a leading strategy for developing artificial intelligence.

adaptation
learn
training
machine learning

One method of machine learning uses large amounts of data to make inferences about what should be done, making it inherently *statistical*. A simple example here is the technique of *regression*: fitting a function to data by optimizing the function's parameters.

statistics
regression

Another machine learning approach is to train an *artificial neural network*, a mathematical operation that approximates the function of biological neural networks that are key to brain activity. Typically, a set of *training data* that includes *labels* (the "correct" answers, usually assigned by humans) is processed by the network, which adapts its parameters such that its output approaches the "correct" answers. Then labeled *testing data*, on which the network has *not* trained, tests to see how well the network performs on new data.

neural network
training data
labels
testing data

Relatively recently, a combination of advances in computing power

and new approaches to using neural networks has been highly successful, surpassing previous performance metrics for many important applications (e.g. vision, natural language processing, game playing, etc.). The new techniques use multiple network *layers* and are collectively called *deep learning* (deep in layers).

layers
deep learning

01.05.3 Control theory

A *control system*, which *control theory* studies, is comprised of a system to be controlled called the *plant*, a *feedback* measurement of the plant's output, and a *controller* that determines the plant's input given the feedback and some goal output. This is frequently represented in the block diagram of Figure 01.9.

control system
control theory
plant
feedback
controller

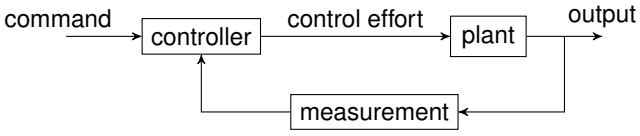


Figure 01.9: a feedback control system block diagram.

If it is successful at meeting performance goals, such a system meets all the requirements of artificial intelligence in our definition. Although control theory developed as a separate field from AI, it is in fact another form and is now generally recognized as such.

Control systems appear all over most robots as subsystems. For instance, a robot arm joint usually includes a motor (actuator), an encoder (sensor), and a controller. The plant, then, might be the motor-link assembly. A process in some higher-level controller that perhaps has planned the arm's motion and performed an inverse kinematics calculation would send a command to the joint to rotate to a specific angle.

01.05.4 General artificial intelligence

There may be objections at this point about our threshold for intelligence being too low. Some find it to so because they conceive of intelligence as being what is now called *general artificial intelligence*: a single AI system applicable to *any* problem. For instance, it would be capable of navigating a robot, having a conversation, and playing the cello. This is sometimes called *strong AI*, in contrast with the version considered above, now termed *weak AI*.

general AI
strong AI
weak AI

There is no strong AI, at present. However, if it does emerge, it is possible that it will develop beyond human intelligence rather quickly, into a state called *superintelligence*. There are ethical concerns here, with some potential for superintelligence becoming an existential threat to the human species.

superintelligence

Lecture 01.06 Robot artificiality and artificial life

artificial

We have claimed a necessary quality of a robot is that it is *artificial*, i.e. designed by humans. We include robots designed by robots designed by robots ... designed by robots designed by humans. That is, as long as the original robot that begot further generations was designed by humans, these offspring are also considered artificial and therefore robots (if they also meet our other criteria for a robot). Robots designing robots—what does this mean?

artificial life

A field of interest to many roboticists is that of *artificial life*, which is the interdisciplinary (biology, chemistry, computer science, robotics, etc.) study of life itself and its artificial creation. The term "life" has no universally agreed upon definition, but certain features have been suggested as necessary, such as the following.

Appearing life-like is a tacit requirement for many people. Despite its apparent banality, this is perhaps not to be overlooked, for it is plausible that "the meaning of a word is its use in the language" (Wittgenstein and Anscombe, 2001). This has been taken to heart by artists such as Theo Jansen, who has created *strandbeests*: wind-powered kinetic sculptures that walk on the beech and appear life-like. This perspective is limited, however, since it doesn't elucidate the meaning of "life" to say that it is that which appears to be "life."

Self-organization is the process of local interactions of a disordered system yielding global order. Emergence in robotics is a form of self-organization. Examples of self-organization include ant colonies, crystal growth, and lasers.

Self-replication is the process of an entity creating a copy of itself (perfect or not). Of course, these copies would (usually) also be capable of reproducing.

Natural selection is the evolutionary process that requires both variation in self-replication and the natural survival and reproduction of those offspring that are better-adapted to the environment.

Autopoiesis is the (fundamentally cellular) logical loop that posits its own constitutive self-environment boundary as being caused by itself (Varela, 1996). Or, "life emerges when the external limitation (of an entity by its environs) turns into self-limitation" (Žižek, 2012).

Others suggest no such general qualities of life can be established (Wolfram, 2002, 2017) because our definitions are always relative to our own human

perspective. One wonders, however, what more one could hope for from any definition.

01.06.1 Cellular automata

Early researchers constructed abstract models of life from small sets of basic rules.

One such model is the *cellular automaton*, which is a set of (abstract) cells in a grid (of any finite dimension) such that each cell has *neighbors*: is adjacent to others. Each cell can be in one of a finite number of states. A set of rules determine the new state of each cell at each (discrete) time step from its previous state and the previous state of its neighbors. Therefore, in non-stochastic models, a given initial state or *initial condition*, together with the set of rules, results in a deterministic process.

cellular automaton
neighbors
initial condition

In *Conway's Game of Life*, a two-dimensional cellular automaton, the two states are taken to be simply "populated" or "empty". The game has the following, simple rules:

Conway's Game of Life

> *For a space that is "populated":*
> *Each cell with one or no neighbors dies, as if by solitude.*
> *Each cell with four or more neighbors dies, as if by overpopulation.*
> *Each cell with two or three neighbors survives.*
> *For a space that is "empty" or "unpopulated":*
> *Each cell with three neighbors becomes populated.*

Somewhat surprisingly, very complex patterns emerge in this simple game. An example of what a game can look like is shown in Figure 01.10. Play it yourself, here

`www.conwaylife.com`.

Or download the app Golly here:

`golly.sourceforge.net`.

One such cellular automaton is *John von Neumann's universal constructor* self-replicating "machine," which works as follows (Von Neumann and Burks, 1966). Consider an automaton system containing the following elements.

John von Neumann's universal constructor

A description ϕ_1 of this system sans the description itself (for it cannot contain both itself and other automatons).

A universal constructor that can read a description of an automaton and construct it.

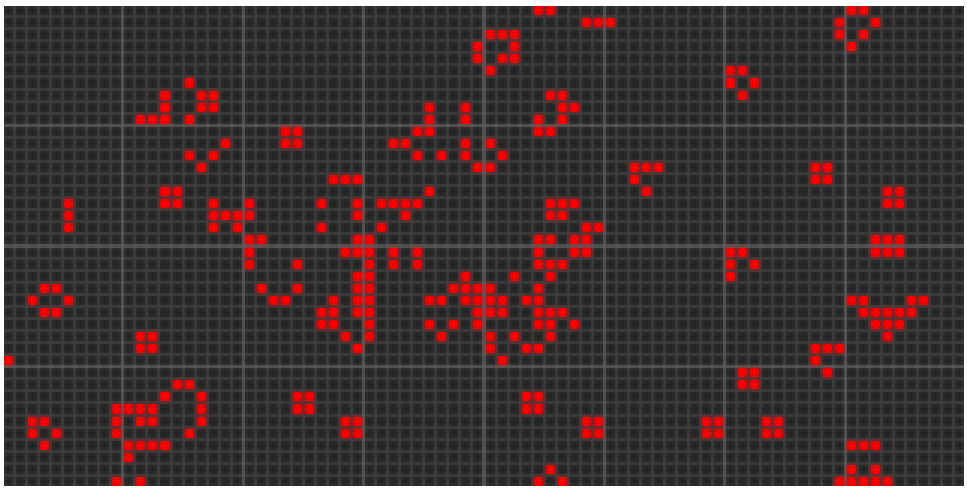


Figure 01.10: a game of life state at one moment in time. Red cells are "populated" and gray are not.

A universal copier that can copy any description of an automaton.
A controller that applies the constructor and copier.

Let the collection of the constructor, copier, and controller be called X_1. Then the original machine is (X_1, ϕ_1). The controller commands as follows. It

1. commands the copier to make two copies ϕ_2 and ϕ_3 of the instructions ϕ_1;
2. commands the constructor to read ϕ_3 (thereby destroying it) and construct a new machine (sans instructions) X_2; and
3. ties together X_2 with the undestroyed copy of the instructions ϕ_2.

Now there are two machines, the original (X_1, ϕ_1) and its descendant (X_2, ϕ_2).

01.06.2 Living robots?

Lest we seem to be too far afield from robotics, let's return to robots, proper, with their mechanical presences. We have examined how a robot might be considered intelligent—but alive? Some researchers not only think it is possible, they plan to make them.

For instance, the "Autonomous Robot Evolution" (ARE) project is designing an ecosystem for robot evolution (Hale et al., 2019). One of the key aspects of natural selection is competitive survival, which requires an arena. This project includes the creation of such an subsystem, along with several others, such as an ecosystem manager, a virtual environment, and a training environment.

This is one among several projects with artificial life as a goal. At this point, few have short-term ambitions to become mechanical, but the foundations are being laid.

Lecture 01.07 Robot autonomy and human-robot collaboration

Autonomy is our final essential condition for robots. As with some of the others, it is challenging to draw a line between devices that are and aren't autonomous. Even if we were "hardliners," there would be ambiguity: does autonomy include independence of human influence in *all* things? Consider the following aspects of robot behavior, considered as the behavior of a group or an individual robot:

who the choice of robot(s) acting,
what the action,
when the timing,
where the location,
why the goal, and
how the method.

Any of these six aspects could be autonomous, but it seems too strong to require all these to be autonomous; after all, isn't one of our motivations for making robots having some influence on them? And there are more considerations, such as the programming, construction, and even design of the robot. Perhaps the only fully autonomous robot is an ideal robot approached in an evolutionary process of "alive" (*à la* artificial life) robots.[4] Our perspective is that if any of the above is autonomous, that is sufficient to satisfy the condition of robot autonomy.

It turns out we would like to inhabit the same spaces as robots. In fact, one of our primary motivations for building robots is to have them *help us*. If you've ever been helped by someone over whom you have no influence, you'll start to see the trouble with "fully autonomous" robots. What has proven more valuable in virtually every field of robotics is work that contributes to the better integration of human and robot activities. A way to consider the breadth of this field is to give it two categories.

Human-robot interaction (HRI) is the broad and interdisciplinary study of the interaction of robots with humans, including communication, socialization, and design.

Human-robot collaboration (HRC) is the study of human-robot teams working together to achieve goals. It is sometimes considered a subcategory of HRI.

[4] And even here, one may object that the evolutionary process was started by humans, the conclusion being that a truly autonomous robot is, in fact, impossible.

We will here explore HRC in more detail.

01.07.1 Human-robot collaboration (HRC)

It has been observed that what is hard for a human is easy for a robot and what is easy for a human is hard for a robot. This is often understood as a challenge to human-robot interaction: humans tend to expect robots to be able to perform tasks simple to humans easily, so robots frequently seem downright inept. However, we can also turn this observation around: humans and robots are actually *complementary*.

complementary

The problem, then, is to find ways for robots to work *collaboratively* with humans—not necessarily replace them. There are, of course, several challenges, most of which have been revealed by attempts to design, build, and deploy collaborative robots. Other challenges were predicted by studying *human collaboration* to discern some essential qualities of collaboration that will likely remain true in successful human-robot collaboration. Before we consider some important ideas that have emerged from these studies, it is worth noting that, since humans find collaboration with humans on many tasks, like moving furniture, simple, we should expect that robots won't. And this turns out to be very much true.

collaboration

human collaboration

shared goal Collaboration requires team members share a common goal.

commitment Collaboration requires members of the team to be committed to the achievement of the common goal.

knowledge Collaboration requires members of a team to internally represent knowledge of the states of the environment.

sensitivity Collaboration requires team members be sensitive to the environment, each other, and themselves.

communication Collaboration requires team members to be able to communicate effectively, updating each other about the states of the environment and themselves.

planning Collaboration requires team members be able to plan; that is, to reason through which actions are required to achieve a common goal.

Relatively simple models of practical reasoning (e.g. belief-desire-intention or BDI, Müller (1999)) and relatively detailed cognitive architectures (e.g. Soar, Laird (2012)) have been used to better design robots that can better collaborate with humans.

Several robot control architectures have been developed with insights gained from this work. We will review some important examples in Chapter 04.

01.08 Exercises for Chapter 01

Exercise 01.1

Classify each of the following as robot or not-robot. If it is not a robot, list the missing qualities. Comment on ambiguities.

a. A conveyor belt that maintains a constant speed regardless of load.
b. A remotely controlled unmanned aerial vehicle (UAV).
c. A simulated spaceship.
d. A raven.
e. A high-speed train.

Exercise 01.2

For each of the types of robots described below, list at least one potential actuator, effector, and behavior.

a. A self-driving truck.
b. A small, insect-like aerial robot.
c. A manufacturing robot that sands parts.

Exercise 01.3

For each of the robot behaviors below, list three useful sensors.

a. Flying through air.
b. Driving across bumpy terrain.
c. Swimming through water.
d. Pouring a glass of whiskey.
e. Folding a shirt.

Exercise 01.4

For each of the robot-action pairs described below, describe a specific potential actuator and effector.

a. A mobile robot lifting a crate from the ground to a shelf at a height of 3 m.
b. A stationary manipulator robot painting a car.
c. A small mobile robot hopping up stairs.

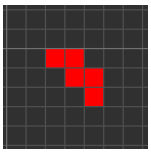
Exercise 01.5

Write a one- or two-sentence response to each of the following questions and imperatives.

a. Why must robot action maps be performant to be considered intelligent?
b. What is machine learning?
c. Apply the definition of intelligence to the following system to determine if it is intelligent: a control system that rotates a link at a constant rate, regardless of load.

Exercise 01.6

Play a standard Conway's Game of Life on paper, starting with the configuration shown. How many generations does it take to develop a static pattern, and what is that static pattern?



Exercise 01.7

Write a one- or two-sentence response to each of the following questions and imperatives.

a. What are some aspects of a self-driving car that are autonomous? Explain why!
b. Classify each of the following as human-robot collaboration or not. Explain why!
    i. A manufacturing process in which a human builds a robot performs a task, then, separately, the robot performs a task.
    ii. A manufacturing process in which a human and a robot perform parts of the same tasks, simultaneously, in the same space.
    iii. A drone with autopilot and a remote human actuating cameras aboard the drone.
    iv. A team of five drones and two humans searching a disaster area for survivors, with the drones giving the humans additional views of hard-to reach areas.
    v. A robot crawling through a tight space with a human operator that has some control, but who is overridden when the robot deems a command from the operator to be dangerous.

c. Why is it that robots are probably better when they aren't "fully" autonomous?

02

Embodiment

03

Robot mechanics

04

Robot control architectures

control architectures

Robot control architectures are conceptual structures for organizing robot control such that we can design controllers systematically. All such architectures include maps of measurements to actions, a process that was central to our definition of intelligence (Lecture 01.05). We call this process *sense-decide-act* (SDA). With reference to Figure 04.1, sensing (measurement) provides the robot with information about the state of itself and the environment; from this, a decision is made about how the robot should act; finally, the robot acts. The differences among robot control architectures lie almost entirely in the *decide* step—that is, in the *controller*.

SDA

controller

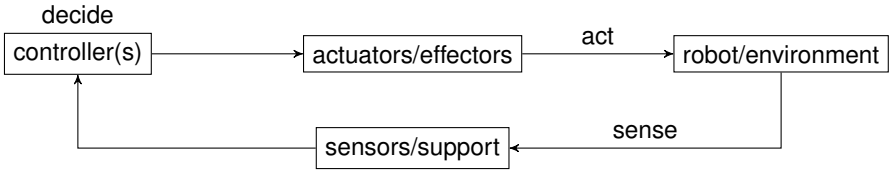


Figure 04.1: a block diagram showing the sense-decide-act structure common to all robot control architectures.

The "controller" here is not necessarily a single device, although it can be. Control devices are frequently *microcontrollers* that include *microprocessors*, *memory*, and input/output interfaces. However, some control logic is so simple, it can be instantiated in analog- or digital-circuits alone. It is also notable that the diagram of Figure 04.1 encompasses processes that can be happening asynchronously and in parallel. For instance, measurements may be made at different times, controller decisions may take different times for different situations, etc.

microcontrollers
microprocessors
memory

From our understanding of feedback control theory,[1] we can conceive of how we might control simple robot actions, such as turning by some angle or raising an effector to some height. While feedback control systems of complex systems (like a robot arm) can be very complicated, they typically require *low-level commands*, i.e. a goal state through time.

low-level commands

Actions, tasks, and behaviors

As necessary as feedback control is, it is inadequate to command the robot to perform complex actions, such as finding an object or exploring an environment—i.e. *high-level commands*. But just such high-level commands are what a designer would like to give a robot. Sometimes, we say there are *mid-level commands* as well, those that require more than low-level commands, but are probably lower-level than a robot designer would like to give. In fact, we can categorize actions by command complexity.

high-level commands

mid-level commands

Simple actions are those that require only low-level commands. For instance, moving an effector to a given state is a simple action.

Tasks are actions that require only mid-level commands. For instance, grasping an object in a gripper is a task.

Behaviors are actions that require only high-level commands. For instance, following walls is a behavior.

These categorizations are helpful, as we'll see, despite their ambiguity.

Models and their representation

models

Some robot control architectures use internal *models* to help the controller to decide what to do. These models are typically mathematical models, maps of the environment, and mechanical solid models. Models, of course, need *representations* that can be stored in computer memory. However, models useful in many robot control applications take a lot of memory (i.e. they are *memory-intensive*), which is only the first of three major drawbacks. The second is that using the models is *processing-intensive*, which costs power, money, complexity, and most importantly *time*. The third drawback is that these internal models don't age well and usually require constant updates in a dynamic environment.

representations

memory-intensive

processing-intensive

time-intensive

Despite the drawbacks, however, models are very helpful, especially when the robot is to be designed to exhibit a behavior that requires multiple

[1] We assume the reader has at least a cursory understanding of feedback control theory. If not, please review Chapter 01 of our *Control: an introduction*.

steps to be effective. For instance, it's not hard to go from location A to location B when there are no obstructions: just go toward B. However, if there are obstacles, it is more-difficult, and if there is a labyrinth—a map would surely help!

The architectures

There are four common robot control architectures.

deliberative control Deliberative control makes extensive use of stored information and models to predict what might happen if different actions are taken, attempting to optimally choose a course of actions. This allows the robot to *plan* a sequence of actions to achieve complex goals (exhibit a behavior), thereby allowing a designer to give high-level commands that are interpreted in terms of extensive models. This paradigm is often called *sense-plan-act*, thereby substituting "plan" for "decide" in our usual scheme. In essence, deliberative control decides actions through careful planning. Deliberation is costly in terms of the hardware required, the energy used by computation, and, most importantly *time*. Even with seemingly ever-increasing processing power, time remains the bottleneck for deliberative control. "Pure" deliberative control is rarely used, as we'll see, but it is nonetheless indispensable for some behaviors.

planning

sense-plan-act

reactive control Reactive control is rather elegant in its simplicity: simple rules map sense data to simple actions, but in combination these rules interact to generate task-level actions. Or perhaps it's better to say a designer arranges these simple rules to achieve modular task-level actions. The most common variety of this architecture is the *subsumption architecture*, which uses the concept of *layers*, which can affect (subsume) each other in limited ways we'll explore. Layers can frequently be constructed to yield task-level actions, but usually more is required to exhibit full-blown behaviors (again, these categories are fuzzy).

subsumption architecture

hybrid control In hybrid control, a wedding is held for deliberative and reactive control in the hopes that each's positive qualities will be brought forth and negative qualities will be left behind. This is probably the most popular approach, but it is very challenging to arbitrate between or mix the two approaches in such a way that it doesn't comprise an unhappy union. Popular tasks for reactive control are danger-zone shutdowns, obstacle-avoidance, and

pretty much any activity that requires a quick … reaction. Left to deliberative control are the high-level decisions that aren't too time-sensitive, such as path-planning, object recognition, and task coordination.

behavior-based control Behavior-based control tries to extend reactive control beyond tasks to behaviors. This is really an attempt to design emergence, a behavior that is not explicitly commanded, but is comprised of simple actions running more-or-less in parallel. As we will see, reactive and behavior-based control rely heavily on lessons learned from biology, especially evolution and emergence.

Each of these robot control architectures is explored in this chapter. Later, we will consider how to instantiate these in software and hardware, simulated and mechanical.

Lecture 04.01 Deliberative robot control architecture

Lecture 04.02 Reactive robot control architecture

reactive control architecture

Robot control that is characterized by sense data being simply mapped to simple actions that work together to achieve tasks is said to have a *reactive control architecture*. By "simply mapped," we mean a long calculation is not required to determine the appropriate action. Frequently, the maps are simple rules like, "If s then a." For instance, "if a dropoff is detected ahead, stop." This structure is called a *finite state machine* (FSM). A FSM models a robot-environment "world" as consisting of a finite number of states, exactly one of which exists at each moment. *State transitions* occur from one state to another when some conditions are met. In the case of "If s_1 then a_1," we define a *state transition function* (map) f_1 that maps (sense) event s_1 to action a_1, which presumably will change the actual state to some (usually) new state s_2.

finite state machines

state transitions

state transition function

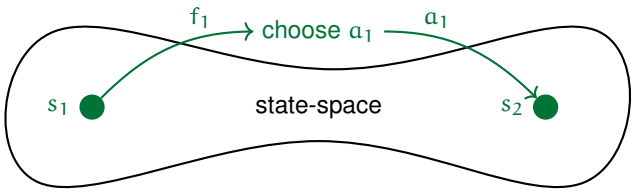


Figure 04.2: a state transition from s_1 to s_2 via the state transition function f_1 and action a_1.

For simple actions, it is easy to see how these maps work. For more-complicated actions, especially those involving long sequences of simple actions, it is not so clear how to go about designing such maps. This is especially true when we consider the frequently large number of possible states in which the robot could be: for every position, orientation, speed, distance from objects, etc., actions must be specified. In other words, the state-space is usually large and if we imagine, as designers, assigning an action to each state ... we see the trouble: there are too many possible states to choose an action for each. In other words, the problem is usually *intractable*.

intractable

One approach is to break the state-space into subspaces and assign actions to these, instead of individual states. But a further complication here arises: what if the subspace domains of these maps aren't mutually exclusive? Consider Figure 04.3. In the region of overlap $S_1 \cap S_2$, both f_1 and f_2 apply, leading to different actions a_1 and a_2. Sometimes there is no conflict and both actions are desirable (and non-conflicting); other

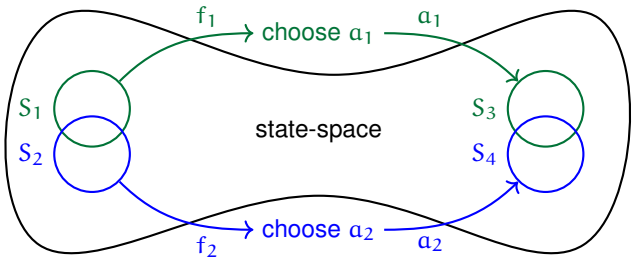


Figure 04.3: two overlapping subspaces with corresponding state transitions.

times, only one or the other is desirable, so *arbitration* is necessary; finally, sometimes a *fusion* is desirable in which the original actions are mixed in some way. For instance, perhaps $S_1 = \{\text{an object is on the left}\}$ and $S_2 = \{\text{an object is on the right}\}$. In, for instance, a corner of a room, both will be true, so the state $s \in S_1 \cap S_2$ obtains. If a_1 is "continue and angle right" and a_2 is "continue and angle left," which seems reasonable, something must be be done because there is clearly a conflict here. If we proceed by arbitration, either a_1 or a_2 is chosen, but neither is probably desirable. We could proceed by a simple fusion in which we simply "add" the two actions (programmatically and not electro-mechanically, which would waste of power and could cause damage to the robot): the robot would just continue forward. No, instead, we probably want what could be considered a new subspace-function-action or a more complex fusion, something like "stop, rotate by some angle, and continue."

arbitration
fusion

But even if a designer could go through each subspace and assign it an action in a reasonable amount of (design) time, which actions (and arbitrations) should they choose in each state to consistently achieve desired tasks?

To even further complicate things, the state of the robot must be estimated from measurements, from which it is not always possible to completely or accurately reconstruct the state. And even when it is possible, the estimation process can be model-dependent and therefore it may take (run) time—something generally discouraged in a reactive control architecture.

These challenges indicate a systematic design approach. This is provided by the *subsumption [reactive] architecture* (SA), to which we now turn. But before we describe its structure, it is worth considering some of its fundamental design principles.

subsumption architecture (SA)

04.02.1 The world is its own best model

The motto, (Brooks, 1999)

> *The world is its own best model*

is one of the fundamental principles of the SA and other reactive architectures. The idea here is that it is better to get information about the world from itself than from models thereof—that is: measure it, and now! This means the SA relies very little on models and computation. For instance, consider a robot performing the three-action task T_1: (a_1) pick up an object; (a_2) open the hatch; and (a_3) place the object inside. We could reason as follows: when we pick up an object, open the hatch, then place the object inside. That is, $a_1 \Rightarrow a_2 \Rightarrow a_3$. The implicit assumption, here, is that we know how things will go. But the world is a fickle place, my friend. The object was slippery and part way through being picked up (a_1), it was dropped, and nothing was placed inside! Or the hatch got stuck (a_2) and our manipulator crashed into it (a_3)! However, using the principle that the world is its own best model, we would not rely on such (FSM) logic. Instead, we might use realtime sensor information, interpreted as events, say

- $s_1 = \{$sensed an object to pick up$\}$,
- $s_2 = \{$sensed an object near the hatch$\}$, and
- $s_3 = \{$sensed the hatch is open$\}$.

Then events would proceed as follows:

- if s_1 then a_1 (should cause s_2),
- if s_2 then a_2 (should cause s_3), and
- if s_3 then a_3.

This is much more resilient; if, for instance, the hatch gets stuck, then $\neg s_3$ and therefore $\neg a_3$—that is, the manipulator would not crash into the hatch.[2]

communication

At this point, we can see another way of thinking about this design principle: it is as if *communication* among the modules that act is channeled *through the world itself*. Instead of communicating among modules through

[2] This simple example ignores the obvious fact that even a non-reactive control architecture would probably make more extensive use of sensor data than imagined here. Similarly, the "model" here is a simplistic FSM logic: if action a_i, then the event I expect will certainly follow. Most models would be more nuanced and be updated from sensor data; however, added model detail leads to slower responses.

software or hardware signals, the results of each one's action in the world (environment-robot) are simply *there* and need no other "model."

Although this principle was originally developed by the founders of the reactive control architecture, it has really become a general design principle in all robotic control. And let's not kid ourselves: it has its limitations. The most significant limitation is *temporal*: sometimes the past and the (modeled) future are relevant to what actions we would be best taken now. Furthermore, sensor data is imperfect and incomplete: although we have said the world is "simply there," this is actually a fantasy, and we always have to *estimate* what is going on from measurements. It is more honest to say "it is easier to measure most things than to model them."

temporal limitations

estimate

04.02.2 Evolution and emergence

The next design principle of the subsumption architecture is

> *Start with the simplest actions. I.e.—design bottom-up!*

The apparent banality of this is deceiving: it is easy to get stuck thinking about "high-level" behaviors when we begin designing. While we cannot forget that these are the goal, in a subsumption architecture (and beyond), the simplest actions are first. The next principle is related:

> *Iteratively include more actions, debugging along the way.*

The idea is to try to form more-complex tasks by including more actions. How might these actions combine? The following design principle begins to answer this question:

> *Higher actions can override lower ones.*

We mean "higher" in a sense already alluded to, but which will become more precise in the next section. Given our bottom-up approach, lower-levels are designed early and higher-levels are designed later. In this sense, the subsumption architecture design process follows biological *evolution*, which starts simply, builds incrementally, and overrides selectively.

evolution

Finally, consider the final design principle:

> *Complex tasks emerge from combinations of simpler actions.*

This is a sort of "promise" that complexity can be achieved by following these design principles: it is reasonable to expect *emergence*. Given the success of this architecture in many robot applications, it seems well-founded.

emergence

04.02.3 The subsumption architecture

The subsumption architecture uses a type of finite state machine (FSM) model.[3] Transition functions map subsets of the state-space among each other.

module **layer** Design proceeds incrementally by *module* aka *layer*, each of which contains one or (usually) several state transition function definitions. A layer is designed to achieve a *task* like "stand up" or "drive forward" or "wander." Layers are *stacked* "up," with the higher layers having two privileged capabilities over lower layers:

task **stacked**

suppression A higher layer can *suppress* (turn off) one or more of a lower layer's input(s).

inhibition A higher layer can *inhibit* (turn off) one or more of a lower layer's output(s).

This provides a great deal of flexibility in the design process. For instance, consider mobile robot with two layers: a layer L_1: *wander* and a higher layer L_2: *avoid obstacles*. Most likely, it will be necessary for L_2 to inhibit at least some of the outputs of L_1, with L_2, doing its best impersonation of Jesus, "taking the wheel," if you will.

04.02.4 Similar reactive architectures

It is worth mentioning that many reactive architectures have been developed from some or all of the principles of the subsumption architecture. In particular, the behavior-based architecture of Lecture 04.04 is a direct extension thereof. Others, such as SMACH (`wiki.ros.org/smach`) from the Robot Operating System (ROS – we will introduce ROS in Part II). SMACH uses what is called a *hierarchical state machine* that has several advantages.

hierarchical state machine

[3] Brooks uses some nonstandard terminology here that can cause confusion. He calls the fundamental building unit of the subsumption architecture an "augmented finite state machine" or AFSM. By "state machine," he seems to mean what we have called a state transition function and action. By "augmented," he seems to mean the inclusion of a regular transition-action, registers, timers, and connections thereamong (Brooks, 1999, p. 30).

Lecture 04.03 Hybrid robot control architecture

Lecture 04.04 Behavior-based robot control architecture

04.05 Exercises for Chapter 04

Exercise 04.1

Respond to the following questions and imperatives with a sentence or two and, if needed, equations or a diagram.

a. For a mobile land robot, give an example of a simple action, a task, and a behavior.
b. Why is it usually best to avoid models, when possible?
c. Which of the robot control architectures considered in the chapter tends to have the shortest "decide" step in the sense-decide-act paradigm? Explain why.
d. Explain why a robot control architecture is necessary, especially in light of the fact that we have feedback control.

Exercise 04.2

Consider a mobile robot we would like to exhibit the behavior of wandering about an indoor environment with the usual walls, halls, obstacles, etc., covering as much of it as possible. It has three "bump" sensors, one in front and one on each side to detect when it hits a wall. The robot can drive forward, stop, and pivot. What are the possible states of the robot-environment system? Design a simple *reactive* controller that allows it to cover as much of the environment as possible move about without getting stuck. Be sure to specify the states that, when sensed, would be mapped each action, and the new states expected as the outcomes of each action. Furthermore, be sure to clarify any suppressions and/or inhibitions.

Part II

Introduction to ROS

05

Introducing ROS

All the high-level considerations of Part I have to be instantiated *somehow*. How do we keep track of the state of robot? Implement a controller? Communicate among robots? Interface with a user? The answer is almost always: (with difficulty and) with *computers*. And we know what that means: *software*.

computers

software

As we saw in Part I, *robots are complicated.* Can you imagine the amount of software required to run a given robot? A *ton*. Not to mention the expertise in several sub-fields within robotics. In the late 2000s, roboticists started the difficult but important task of collaboratively developing an open-source software framework that can be used to program many different types of robots: the *Robot Operating System* (ROS) (Quigley et al., 2009).

ROS

ROS is a framework in that it brings together code libraries, code tools, and development conventions to create a system in which individual applications can be developed. Many robotics researches share their expertise and development work via this framework, which means (among other things) cutting-edge libraries are available to everyone.

We adopt this platform, which is now ubiquitous.

Lecture 05.01 ROS methodology

ROS has several key aspects to its methodology that are worth considering at this point.

05.01.1 Distributed computing

nodes **messages** **P2P**

ROS *nodes* are software modules running on potentially different computers. Nodes communicate by sending *messages* over a network *peer-to-peer* (P2P) – that is, directly to each other. This lack of centralization is very flexible and scalable. Nodes, messages, and related concepts are described further in Lecture 06.01.

05.01.2 Use with other programs

ROS systems can easily interact with software tools for visualization, navigation, data logging, etc. This strength allows ROS to remain focused on its core tasks.

05.01.3 Multilinguality

ROS programs can be written in several languages, including Python, C++, and Matlab. The most popular are Python and C++, and we will use the former.

client libraries

The development of ROS programs with a specific language is enabled by a language-specific *client library*. All but the Python (`rospy`), C++ (`roscpp`), and LISP (`roslisp`) client libraries are considered experimental.[1]

05.01.4 Modularity

ROS developers (you!) are encouraged to write their programs in a modular manner such that each module performs some limited task, then *composing* several modules to perform more-complex tasks. This makes debugging, maintenance, and collaboration much easier.

packages

Previously developed ROS programs are available in the default ROS installation and in the form of additional *packages*. We will discuss packages more in Chapter 06.

[1] For more client libraries, see `wiki.ros.org/Client Libraries`.

05.01.5 Open sourceness

ROS is open-source! The licensing is such that commercial, proprietary software can include it, making it a good choice for research and industry.

Resource R1 Setting up the development environment

A development environment for ROS can be installed on many personal computers and operating systems. In this text, we use the following stack of software for our development environment.

Resource R1.6 VirtualBox

virtual machines

VirtualBox, by Oracle, is a free virtualizer that can install a *virtual machine* with a variety of operating systems installed. VirtualBox is cross-platform and can be installed on most modern operating systems (e.g. Windows, MacOS, Linux). This allows a host computer (your PC) with (say) Windows to run a virtual machine with (say) Linux—simultaneously!

Box 05.1 if your computer is resource-challenged

ROS can be resource-intensive, especially when running simulations. If your personal computer is resource-challenged, especially in RAM and processing, consider forgoing VirtualBox and installing Ubuntu in dual-boot mode. More on that in a moment.

Download and install the latest VirtualBox app for your host computer:
`virtualbox.org/wiki/Downloads`.
For greater functionality, consider installing the Extension Pack from the same downloads page.

Resource R1.7 Ubuntu Bionic (18.04.4) LTS

This popular Linux distribution is fully compatible with the version of ROS we will use, has a long-term maintenance schedule (LTS), is lightweight, and is free. Download the Desktop (64-bit) version here:
`releases.ubuntu.com/18.04.4`.
We will install this operating system as a virtual machine with VirtualBox.

Box 05.2 if your computer is resource-challenged: dual-boot

If you decide you need to dual-boot, skip Resource Lecture R1.7.1 and see the installation guide:
`help.ubuntu.com/lts/installation-guide/amd64`.
The official installation guide above may not some Windows 10-specific instructions; as a supplement, see:
`tecmint.com/install-ubuntu-alongside-with-windows-dual-boot`.

Resource R1.7.1 Preparing a virtual machine

First, we must prepare a virtual machine with VirtualBox. Use the following steps:

1. In the VirtualBox app, create a New virtual machine. Name it (say) `ubuntu18`, select Type » Linux and Version » Ubuntu (64-bit).
2. Allocate RAM of at least `4096` MB.
3. Select "Create a virtual hard disk now".
4. Select "VDI (VirtualBox Disk Image)".
5. Select "Dynamically allocated".
6. Allocate a maximum of `15-20` GB of virtual hard disk.
7. Select the new virtual machine `ubuntu18` and Start.
8. When prompted, select as the "virtual optical drive" the downloaded Ubuntu `.iso` file and Start.

Resource R1.7.2 Install Ubuntu

Pay special attention to the following during the Ubuntu installation process.

Install Ubuntu.

☑ Download updates while installing Ubuntu.
☑ Install third-party software for graphics and Wi-Fi hardware, Flash, MP3 and other media.
☑ Erase disk and install Ubuntu.[2]

The Ubuntu Bionic documentation is available here:
`help.ubuntu.com/lts/ubuntu-help`.

Resource R1.8 Ubuntu VirtualBox Guest Additions

Installing VirtualBox Guest Additions *in Ubuntu* should improve the performance of your virtual machine. I recommend the following setup process.

1. Open a Terminal window.
2. Update your package manager.

```
sudo apt-get update
```

[2] If you're *not* installing to a virtual machine, be cautious here!

3. Install Ubuntu VirtualBox Guest Additions.

```
sudo apt-get install virtualbox-guest-additions-iso
```

4. Install VirtualBox Guest Utilities.

```
sudo apt-get install virtualbox-guest-utils
```

5. Restart the virtual machine.

Resource R1.8.1 Add a shared folder

Your virtual machine can mount a folder on your host machine such that files can be easily shared between them. In VirtualBox, in your virtual machine's Settings > Shared Folders, click the button that adds a new shared folder. Give it the path of the shared folder on the host machine and do auto-mount. This directory should now be available in the virtual machine as a mounted drive.

Resource R1.9 ROS Melodic Morenia

We will install the recent version ROS Melodic Morenia to our Ubuntu Bionic virtual machine. Follow the official instructions:

`wiki.ros.org/melodic/Installation/Ubuntu`.

Terminal It assumes you will enter the given commands in the *Terminal* in Ubuntu, which can be opened through the GUI or with the keyboard shortcut Ctrl + Alt + T.

Follow the recommended options and be sure to:

- install the "Desktop-Full" version;
- under "Environment setup," follow the instructions to source your environment variables in your dotfile `~/.bashrc`; and
- under "Dependencies for building packages," install the recommended tools, listed in the provided command.

Consider working through the first four ROS tutorials:

`wiki.ros.org/ROS/Tutorials`.

Use the recommended `catkin` option.

Resource R1.10 Python

Python 2 We will write most of our ROS code in *Python 2*. It's best practice not to mess with Ubuntu's Python installation and instead install our own.

The package *pyenv*[3] will help us manage what will be multiple Python versions. Installing `pyenv` is easy in a Terminal. **pyenv**

```
curl https://pyenv.run | bash
```

Be sure to open your `~/.bashrc` file (e.g. `gedit ~/.bashrc`) and add the following lines.

```
export PATH="/home/picone/.pyenv/bin:$PATH"
eval "$(pyenv init -)"
eval "$(pyenv virtualenv-init -)"
```

To finalize the installation, either open a new Terminal or `exec $SHELL` (this will reload your `.bashrc` so that `pyenv` is available to the bash shell). Now create a fresh Python 2 installation using `pyenv`.

```
pyenv install 2.7.17
```

Now we can list installed Python versions with the following command.

```
pyenv versions
```

```
system
2.7.17
```

To set the global default Python version, use the following.

```
pyenv global 2.7.17
```

You can also set local Python environments using `pyenv local`, which sets the environment in the current and sub-directories.

[3] See `github.com/pyenv/pyenv` for documentation.

06

ROS basics

In this chapter, we will explore some of the basic concepts and tools used to create ROS systems. Understanding the structure and organization of these systems is a prerequisite for developing ROS systems of our own.

Box 06.1 Install first

Before continuing, install ROS. See Resource 1.

Lecture 06.01 ROS graphs

ROS graphs **graph theory** **nodes** **edges**

A *ROS graph* is a graph (*à la graph theory*) representation of a ROS systems, such as that of Figure 06.1. Graph *nodes* represent ROS programs running on potentially different machines. Graph *edges* represent the peer-to-peer communication of messages among nodes.

Figure 06.1: a ROS graph with nodes in green and edges in black.

06.01.1 Big Other `roscore`

Jacques Lacan introduced the concept of the "big Other," who is, among other things, the virtual repository of all knowledge, the invisible guarantor that the network of symbols is consistent, and the registrar of the symbolic.[1]

roscore

The big Other of ROS is the communication service *`roscore`*: all nodes of a ROS system register the message streams they provide and those to which they would like to subscribe.

topics

These message streams are organized by *topics*: a node that publishes information for other nodes does so by registering a topic with `roscore`. `roscore` maintains lists of these topics and subscribers thereto and provides these details to any node upon request. If node `foo` publishes to a topic `x`, subscribed-to by node `bar`, `foo` would find out that `bar` has subscribed, then would *directly* transmit messages to `bar` on topic `x`, as shown in Figure 06.2.

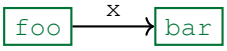


Figure 06.2: a ROS graph of nodes `foo` and `bar` showing the peer-to-peer transmission of messages on topic `x`.

Thus, big Other `roscore` is virtually in all ROS graphs, required for its existence, but we don't include it explicitly. We must launch a `roscore` service for every ROS system. Doing so is trivial in a Terminal window, as follows.

[1] See, for instance, *How to Read Lacan* (Žižek, 2006).

```
roscore
```

Fortunately, we won't have to remember to do this manually every time, as we'll see when we discuss `roslaunch` in Lecture 06.03.2.

Lecture 06.02 ROS packages

packages ROS code is arranged into *packages*. Before we can describe packages, though, we need two apparatus: the ROS build system `catkin` and the ROS *workspace*.
catkin
workspaces

06.02.1 The ROS build system `catkin`

source code Most software is written by programmers as *source code* in some programming language. In this text, we write source code in Python. When the software is ready to be used, it is converted from source code into (binary) machine code and packaged up for distribution. A piece of software that
build automation utility controls this process is called a *build automation utility*. Examples include Make, Qbs, and Cabal.

catkin ROS has its own build system `catkin` built atop CMake, which is itself built on Make. It shares a name with the cluster of flowers such as that of the willow, pictured in Figure 06.3. Because we are developing in Python, we will use only a few of `catkin`'s features, some of which are introduced in the following sections.

Figure 06.3: the catkin of a willow (Didier Descouens).

06.02.2 ROS workspaces

workspaces

Workspaces are directories in which you can develop ROS code. Each project should have its own workspace, and workspaces cannot interact.

06.02.2.1 Setting up a workspace

We will now set up a workspace. Open a bash terminal.[2] Change (`cd`) to a convenient directly like your user home directory `~`. Make a new directory for your code like `code` as follows.

```
cd ~ # change directory to user home
mkdir -p code # -p creates dir only if it doesn't exist
cd code # change directory into code
```

Now make a directory `ros_ws_01` for your new workspace.

```
mkdir -p ros_ws_01
cd ros_ws_01
```

Every workspace needs a source directory `src`.

```
mkdir -p src
cd src
```

Let's inspect the tree we've made.

```
pwd # print current directory
```

```
/home/picone/code/ros_ws_01/src
```

06.02.2.2 Initializing the workspace

Now that we're in the `src` directory, we can initialize a workspace.

```
catkin_init_workspace
```

[2] We assume you have sourced the ROS distribution `setup.bash` in your `.bashrc` file so it will load when you open a new bash terminal.

```
Copying file from
↪  "/opt/ros/melodic/share/catkin/cmake/toplevel.cmake" to
↪  "/home/picone/code/ros_ws_01/src/CMakeLists.txt"
```

As we can see, this created a file `CMakeLists.txt`.

```
ls # list files and folders in current dir
```

```
CMakeLists.txt
```

Now we can finalize our new workspace using the `catkin_make` command from the workspace root.

```
cd .. # up a level to ros_ws
catkin_make
```

We have made a workspace!

06.02.2.3 Sourcing the workspace

Let's investigate the new directories in our workspace.

```
ls
```

```
build  devel  src
```

So `build` and `devel` are new! We will not make much use of the former, but the latter will include the `setup.bash` file, which we will source in order to make available to our shell the new workspace.

```
source devel/setup.bash
```

Note that this must be sourced whenever a new terminal (bash shell) is opened. Of course, you can make this automatically be sourced in your `~/.bashrc` file, but this assumes you will only be using this workspace.

06.02.3 ROS packages

packages

ROS *packages* are code directories containing certain files and organized in a certain way. Packages are usually written for specific applications, but could be applied to many others. The ROS community tends to share packages and develop them cooperatively, but there are privately held packages as well (the ROS license permits this).

06.02.3.1 Creating a new package

In this section, we will create a new package. Packages are developed in a workspace's `src` directory. Let's `cd` to that of the workspace created in the preceding section.

```
cd ~/code/ros_ws_01/src
```

We can create a new package as follows.

```
catkin_create_pkg sweet_package rospy
```

```
Created file sweet_package/package.xml
Created file sweet_package/CMakeLists.txt
Created folder sweet_package/src
Successfully created files in
↪  /home/picone/code/ros_ws_01/src/sweet_package. Please adjust
↪  the values in package.xml.
```

This created the directory `sweet_package` and populated it with `CMakeLists.txt`, `package.xml`, and the directory `src`.

```
cd sweet_package
ls
```

```
CMakeLists.txt  package.xml  src
```

The first of these has information for `catkin` and the directory `src` is initially empty – it will contain the package source code we will write. The `package.xml` file contains package metadata and should be edited.

```
cat package.xml
```

The following is an abbreviated version of the `package.xml` file contents with some editing.[3]

[3] A built-in text editor `gedit` can be used (e.g. `gedit package.xml`). However, consider installing the friendlier app Sublime Text via the Ubuntu Software app store. It will give you the command `subl` (e.g. `subl package.xml`) which you can use to easily edit many text-based files such as `xml` files.

```
<?xml version="1.0"?>
<package format="2">
<name>sweet_package</name>
<version>0.0.0</version>
<description>The sweet_package package</description>

<!-- One maintainer tag required, one per tag -->
<maintainer email="rpicone@stmartin.edu">Rico Picone</maintainer>

<!-- One license tag required, multiple allowed, one per tag -->
<license>BSD</license>

<!-- Url tags are optional, multiple allowed, one per tag -->
<url type="website">http://wiki.ros.org/sweet_package</url>

<!-- Author tags are optional, multiple allowed, one per tag -->
<author email="rpicone@stmartin.edu">Rico Picone</author>

<!-- The *depend tags are used to specify dependencies -->
<buildtool_depend>catkin</buildtool_depend>
<build_depend>rospy</build_depend>
<build_export_depend>rospy</build_export_depend>
<exec_depend>rospy</exec_depend>

<!-- The export tag contains other, unspecified, tags -->
<export>
</export>
</package>
```

depend tags

I have filled in some of this information as an example. Of special importance are the *depend tags*. When we called `catkin_create_package`, the first argument was the name of our new package `sweet_package` and the second argument was a dependency `rospy`, a ROS package which is the dependency that is required for writing nodes in Python. Note that this dependency appears in `package.xml` under multiple types of depend tags; the differences among these tags will be discussed, later. For now, note that we could have added more dependencies when we created the package by listing them after `rospy`. But we can always add more dependencies later by directly editing `package.xml`.

Now that we have a package, we can add Python code files (`.py`) that will become ROS graph nodes to the `sweet_package/src` directory. Before we do this for our own package, however, let's first learn how to run some nodes that come from pre-existing packages.

Lecture 06.03 Running and launching ROS nodes

Let's fire up some ROS nodes! Technically, we could `cd` around our filesystem, find packages, and start nodes with[4]

```
python <filename>.py
```

However, this is highly inconvenient. The `rosbash` package includes several utilities to improve this experience. Install it with the following.

```
sudo apt install rosbash
```

Reload your shell with `exec $SHELL`.

First, we might want to list files in an installed ROS package by simply executing, in any directory, *`rosls`* as follows. **rosls**

```
rosls <package_name>
```

Second, we might want to change to the directory of an installed ROS package with, in any directory, *`roscd`* as follows. **roscd**

```
roscd <package_name>
```

Third, there's tab completion. Terminal itself has *tab completion*: in any directory with a subdirectory named `foo`, type `cd fo<tab>`. It's a sort of autocompletion. ROS itself has this for its commands like `roscd`. Try starting to to type `roscd rospy_tutorials` and hit tab. If there's more than one matching package, double-tap tab to get a list. **tab completion**

There are a couple others that we'll explore in the following sections: `rosrun` and `roslaunch`.

06.03.1 Running ROS nodes

In this section, we'll start a few nodes, mostly from the `rospy_tutorials` package, installed with the following command.

```
sudo apt install ros-melodic-ros-tutorials
```

[4]For the curious, some nodes we'll be starting in a second could be started by navigating to `/opt/ros/melodic/share/rospy_tutorials/001_talker_listener` and executing, say, `python talker.py`.

As usual, after installation, `exec $SHELL` Before we start any nodes, we need a `roscore` service started.

```
roscore
```

Now open a fresh terminal. We'll start our first "real" node with the **rosrun** *rosrun* command.

```
rosrun rospy_tutorials talker
```

In general, the syntax is as follows.

```
rosrun <package_name> <program_filename> [args]
```

So `talker.py` is run and should start printing something like the following every ten milliseconds.

```
[INFO] [1585538656.490473]: hello world 1585538656.49
[INFO] [1585538656.591393]: hello world 1585538656.59
[INFO] [1585538656.691669]: hello world 1585538656.69
```

This `talker` node is publishing `hello world <time>` on topic `chatter`. In a new terminal window, let's start a node to listen to the topic `chatter`: the `listener` node.

```
rosrun rospy_tutorials listener
```

This should give us something like the following.

```
[INFO] [1585542073.580711]: /listener_6552_1585542070720I heard
↪  hello world 1585542073.58
[INFO] [1585542073.682800]: /listener_6552_1585542070720I heard
↪  hello world 1585542073.68
[INFO] [1585542073.780337]: /listener_6552_1585542070720I heard
↪  hello world 1585542073.78
```

The ROS graph we just built is considered the "hello world" of ROS and is depicted in Figure 06.4.

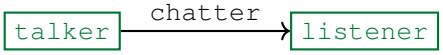


Figure 06.4: the `talker-listener` ROS graph with topic `chatter`.

You can generate similar ROS graph representations with the following, in a new Terminal.

```
rqt_graph
```

When you're satisfied, *stop* each node with ctrl+C. **stop a node**

06.03.2 Launching ROS nodes

It is inconvenient to manually `rosrun` every node for larger (i.e. typical) ROS graphs. *Launch files* have extension `.launch` and are collections of node information that the command *`roslaunch`* operates on. The example `talker-listener` graph from above has a launch file `talker_listener.launch`. **launch files** **roslaunch**

Let's first find the launch file.

```
roscd rospy_tutorials/001_talker_listener
ls
```

```
listener  listener.py  README  talker  talker_listener.launch
↪  talker.py
```

Now let's print its contents.

```
cat talker_listener.launch
```

```
<launch>
  <node name="listener" pkg="rospy_tutorials" type="listener.py"
  ↪  output="screen"/>
  <node name="talker" pkg="rospy_tutorials" type="talker.py"
  ↪  output="screen"/>
</launch>
```

The `pkg` parameter for each `node` tag specifies the package from which the node comes; the `type` tag, the Python file; the `output` tag is often `"screen"` so that the node outputs to the console (instead of just a log file). The `name` tag may at first seem superfluous. However, it is very important: distinct `name`s can be given to the same node `type`. For instance, two `listener.py` nodes can be launched with distinct `name`s. This is one way of separating what is called the *namespace* of a ROS graph. **namespace**

From any directory, the `talker-listener` graph can be launched with the following call to the launch file.

```
roslaunch rospy_tutorials talker_listener.launch
```

We should get the same results as our manual (`rosrun`) method above.

Lecture 06.04 Coordinate frame transformations

position
orientation

Robotics requires we keep track of the positions and orientations of many objects in three-space. A rigid body's state can be expressed as a three-dimensional *position* and *orientation* (angular position). In a coordinate system, this takes a minimum of $3 + 3 = 6$ coordinates.

body-fixed

Different objects have different convenient coordinate systems. For instance, a mobile robot might have a *body-fixed coordinate system* with origin at its geometric centroid, x-axis pointing forward, y-axis pointing leftward, and z-axis pointing upward. Locating an object in this coordinate system would be different than that of, say, a base station. Consider for a mobile robot a two-dimensional body-fixed coordinate system o, world coordinate system w, and a pseudo body-fixed coordinate system p that is merely a translation of the world coordinate system to the p origin—see Figure 06.5. Let a point in space in $w/p/o$-coordinates is represented by the position vector $\mathbf{r}_w/\mathbf{r}_p/\mathbf{r}_o$. Let $\mathbf{t}$, be a vector from the w-origin to the p-origin.

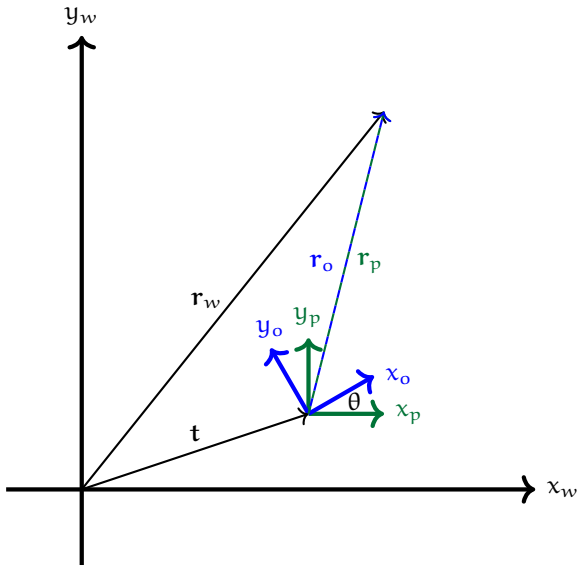


Figure 06.5: a two-dimensional body-fixed coordinate system o, world coordinate system w, and a pseudo body-fixed coordinate system p.

06.04.1 Translation

Suppose the robot can only translate and not rotate. The w and p coordinate transformations are sufficient to describe its motion. The transformation is

$$\mathbf{r}_w = \mathbf{r}_\mathrm{p} + \mathbf{t} \tag{06.1a}$$
$$\mathbf{r}_\mathrm{p} = \mathbf{r}_w - \mathbf{t}. \tag{06.1b}$$

As we will see in a moment, rotation of a vector is described by a matrix operation on a vector. It is therefore convenient to write translation as a matrix operation in one extra dimension:

$$\mathbf{r}_w = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \mathbf{r}_\mathrm{p} \tag{06.2a}$$
$$= \underbrace{\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}}_{\mathsf{T}} \begin{bmatrix} r_x^\mathrm{p} \\ r_y^\mathrm{p} \\ 1 \end{bmatrix} \tag{06.2b}$$
$$= \begin{bmatrix} r_x^\mathrm{p} + t_x \\ r_y^\mathrm{p} + t_y \\ 1 \end{bmatrix}. \tag{06.2c}$$

The last component, then, becomes an accounting tool for writing the translation operation in this form—called a *homogeneous representation* (Bullo and Lewis, 2005). The transformation matrix T translates but does not rotate.

homogeneous representation

Exercise 06.1

Show that $\mathbf{r}_\mathrm{p} = \mathsf{T}^{-1}\mathbf{r}_w$ by showing it to be equivalent to Equation 06.1b.

06.04.2 Rigid body transformation

Transformation to and from a body-fixed coordinate system is usually a *rigid body transformation*: one that changes coordinate frame origin position and orientation, but preserves the Euclidean distance between any two points. Transformations between the w and o coordinate systems, above, are rigid body transformations. These could be represented as a *rotation matrix* R transformation followed by a translation by $\mathbf{t}$:

rigid body transformation

rotation matrix

$$\mathbf{r}_w = \mathsf{R}\mathbf{r}_\mathrm{o} + \mathbf{t}. \tag{06.3}$$

Here, R rotates counter-clockwise by θ with matrix

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}. \tag{06.4}$$

However, we frequently like to write this in a homogeneous representation, as well, again adding a component to the vectors such that R becomes

$$R = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{06.5}$$

and the rigid body transformation becomes

$$\mathbf{r}_w = TR\mathbf{r}_o. \tag{06.6}$$

06.04.3 Rotation transformations

Rotation transformations, such as R above, come in a variety of flavors.

Euler angles These rotations are described by the sequential rotation about a (typically) body-fixed coordinate system. The order matters because rotating about one axis changes the direction of the others! Not one, but several conventions exist for Euler angle rotation.

Fixed angles Similarly, rotations can be described about axes the origin of which remains fixed to the body, but the orientation of which remains *fixed to the world frame*.

Axis-angles Axis-angle representations describe a rotation as a unit vector and an angle of rotation about that vector.

Quaternions Quaterions are complex numbers with a real part and *three* (instead of the usual one) imaginary parts. They can describe rotations in a manner that avoids certain problems (e.g. gimbal lock and ill-conditioned quatities) of other representations and is more computationally efficient.

The non-quaternion rotation transformations use matrix multiplication and can therefore have homogeneous forms that include translation. Quaternions cannot represent translations, so vector-addition must supplement (multiplicative) quaternion transformations.

06.04.4 The ROS package `tf2`

At this point, some things should be clear:

1. for a three-dimensional robot with six degrees of freedom, keeping track of even two coordinate systems (e.g. world and body-fixed) can be complicated;
2. adding more coorinate systems for arms, sensors, moving objects in the environment, etc.—as most real robots require—vastly complicates coordinate transformations;
3. coordinate transformations change with time as body-fixed coordinate systems move; and finally
4. keeping track of all this in an *ad hoc* way would be disastrous, so a systematic approach is required.

For these reason, ROS provides just such a systematic approach via its *`tf2`* package.[5,6] **tf2**

The `tf2` package has conventions for coordinate transformation data, organized into a tree structure and *buffered in time*. Time-buffering is important: frequently, we need not just the latest data, but recent data as well. As with all ROS dataflow, `tf2` communicates via publishing and subscribing to topics. **time-buffering**

ROS `tf2` uses quaternions to apply and store rotation information. However, it is usually easier for us to think in terms of Euler angles. The older `tf` package provides a nice conversion from Euler angles to quaternions:

```
from tf.transformations import quaternion_from_euler
q = quaternion_from_euler(ax,ay,az) # usage
```

In the usage example, above, rotation angles ("`a`") are applied sequentially to body-fixed `x`, `y`, and `z` axes.

06.04.5 Try out `tf2`

In a Terminal window, enter the following to get and compile a turtle `tf2` demo.

[5]The `tf2` package documentation can be found here:
`wiki.ros.org/tf2`.
The `tf2_ros` package provides Python bindings:
`wiki.ros.org/tf2_ros`.

[6]The `tf2` package replaces the older `tf` package. For information about migrating, see
`wiki.ros.org/tf2/Migration`.

```
sudo apt-get install \
  ros-$ROS_DISTRO-turtle-tf2 \
  ros-$ROS_DISTRO-tf2-tools \
  ros-$ROS_DISTRO-tf
```

Now launch the demo with the following command.

```
roslaunch turtle_tf2 turtle_tf2_demo.launch
```

A separate screen should load with two turtles. Select the Terminal window and use the arrow keys to direct one of the turtles about. The other turtle will follow, as shown in Figure 06.6.

For the full demo, see

`wiki.ros.org/tf2/Tutorials/Introduction to tf2`.

Figure 06.6: a turtle-follow-turtle graphic using `tf2`.

07

ROS topics

In this chapter, we will learn the details of how to publish to, subscribe to, and create topics.

Box 07.1 Get the textbook code

Make sure to start with Resource 2, below. It will explain how to make the code accompanying this text available to your system.

Resource R2 Getting the textbook code

The code we will explore and write together in the following chapters is available in the following code repository:

`github.com/ricopicone/robotics-book-code`.

Follow the instructions there for downloading and making it available to your ROS installation.

Box 07.2 It's going to change

Due to the fact that the code repository is under development and will likely be updated throughout the term, I recommend using the `git`-based method of obtaining the repository and keeping it up-to-date. Resource 3 is a crash-course on how to get and configure `git`.

Resource R3 Installing and configuring `git`

This resource will help you install and configure `git` on your machine. It assumes you are using Ubuntu or some similar OS. It also takes you through setting up your own `git` repository for your ROS packages!

Resource R3.1 Installing `git`

Open a Terminal window. Update `apt`itude.

```
sudo apt update
```

Install `git`.

```
sudo apt install git
```

Check that it is correctly installed.

```
git --version
```

```
git version 2.26.0
```

The specific version of `git` isn't important.

Resource R3.2 Configuring `git`

Set your name and email.

```
git config --global user.name "Your Name"
git config --global user.email "youremail@yourdomain.com"
```

These are stored in `~/.gitconfig`. Change them there, as desired.

Resource R3.3 Setting up GitHub

GitHub (`github.com`) is a place to remotely host a `git` repository. A remote host such as this is important for backup, sharing, and collaboration with `git`. Create a GitHub account here:

`github.com/join`.

To avoid having to re-enter your GitHub username and password frequently, consider setting up an SSH key from the following article:

`git.io/Jeo3f`.

Resource R3.4 Create your own package repo

You'll want a repository for your ROS repositories. Create a new repository in GitHub:

`git.io/JeDDp`.

Consider using the following options.

- ☑ Initialize this repository with a README[1]
- Select[2] Add .gitignore: ROS

Now you can `clone` this repo by navigating to it in the web interface and copying the URL provided by the green Clone or download button.[3]

Open a Terminal window and `cd` to a ROS workspace's `src` directory. Clone your remote repo with the following.

```
git clone <copied repo URL>
```

If this is successful, you should now have local copy of your repository in the current directory.

Now set up your package repository with `catkin_create_pkg`, as we learned in Lecture 06.02.3.1. Once your package is created, stage your changes for commit. First, see which files have changed.

```
git status
```

Stage all changes for commit.

```
git add -A # careful with this
```

Commit changes.

```
git commit -m 'created a package'
```

Now we can push these changes up to the remote repo.

```
git push
```

[1] It is a good idea to have a README in every `git` repository. GitHub makes this easy and even renders it in a nice format on the website.

[2] The `.gitignore` file includes a list of extensions for `git` to ignore in your repository. It is convenient that there is a default one for ROS.

[3] If you have set up SSH, use the SSH URL. Otherwise, use the HTTPS URL.

If it fails, it will probably suggest you set `remote` as `upstream` with the `set-upstream` option. If so, just copy/paste the suggestion and try it.

You now have a `git` repository for your ROS packages!

Resource R3.5 Forking the book code repository

Go to this book's GitHub code repository:

`github.com/ricopicone/robotics-book-code`.

On the upper-right, click the Fork button. This will give you a copy of the repository in *your* GitHub account. Now you can `clone` this fork by navigating to it in the web interface and copying the URL provided by the green Clone or download button.[4]

Open a Terminal window and `cd` to a ROS workspace's `src` directory. Clone your remote repo with the following.

```
git clone <copied repo URL>
```

Follow the directions in the README to use `catkin_make` to make the repo packages available to ROS.

Resource R3.6 Updating from the original repo

If you'd like to bring updates to the book's GitHub repo into your fork, first add it as an upstream.

```
git remote add upstream \
  https://github.com/ricopicone/robotics-book-code.git # or ssh
```

Fetch remote branches.

```
git fetch upstream
```

Make sure you're working on your `master` branch.

```
git checkout master
```

Now merge your and my `master` branches.

```
git rebase upstream/master
```

[4] If you have set up SSH, use the SSH URL. Otherwise, use the HTTPS URL.

Lecture 07.01 Publishing to topics

advertise

New topics must first be registered with big Other `roscore`, which will thereafter *advertise* this topic. In `rospy`, the syntax is as follows.

```
pub = rospy.Publisher(<topic name string>,<message_type>)
```

message type

The first argument is the name of the topic and the second is the *message type* (all messages on a topic have the same type). This registers the topic name.

Later, we will learn to create our own message types, but for now we'll stick to the standard message types defined by the ROS package *std_msgs*. For a list of available message types in `std_msgs`, see

std_msgs

wiki.ros.org/std_msgs.

07.01.1 Creating a simple publisher node

The code accompanying the text has a simple publisher node in the `rico_topics` package. You should use `catkin_create_pkg` to create a parallel package in your own code repository, as follows.

```
catkin_create_pkg my_topics \
rospy std_msgs message_runtime message_generation
```

We'll need the dependencies listed above. Create a new Python file in `my_topics/src` with the following.

```
touch my_topics/src/topic_publisher.py
```

Open the empty `topic_publisher.py` in a text editor. You'll want to enter here the same code as appears in the sample `topic_publisher.py` from `robotics-book-code/rico_topics/src`, which is listed in Figure 07.1.

shebang

executable

Since this is the first `rospy` node we've written, it's worth considering it in detail. The first line is called a *shebang* and indicates the file is *executable* and the relevant interpreter (in this case, `python`). One more step is actually required to make your new file executable in Ubuntu: you must change its *permissions* to be executable, as follows.

permissions

```
chmod u+x my_topics/src/topic_publisher.py
```

```python
#!/usr/bin/env python
import rospy
from std_msgs.msg import Int32 # standard int

# Setup: initialize node, register topic, set rate
rospy.init_node( # initialize node
  'topic_publisher' # node default name
)
pub = rospy.Publisher( # register topic w/roscore
  'counter', # topic name
  Int32, # topic type
  queue_size=5 # queue size
)
rate = rospy.Rate(2) # adaptive rate in Hz

# Loop: publish, count, sleep
count = 0
while not rospy.is_shutdown(): # until ctrl-c
    pub.publish(count) # publish count
    count += 1 # increment
    rate.sleep() # set by rospy.Rate above
```

Figure 07.1: `rico_topics/src/topic_publisher.py` listing.

07.01.2 Setting up the node

Back to Figure 07.1, following the shebang, there's the loading of packages via Python's package **`import`** mechanism. Note that we're using both `rospy` and `std_msgs`, which we included in our `package.xml` when we used `catkin_create_pkg`. Then follows the initalization of a ROS node via *`rospy.init_node`*. For more details on initializing nodes, see

`wiki.ros.org/rospy/Overview/Initialization and Shutdown`.

`rospy.init_node`

We then register a topic `counter` of type `Int32` (from `std_msgs`) and *queue size* of `5` via *`rospy.Publisher`*. Queue size is how many buffered messages should be stored on the topic. The general guidance is: use more than you need. For more on selecting queue size, see

`wiki.ros.org/rospy/Overview/Publishers and Subscribers`.

queue size

`rospy.Publisher`

Finally, we use `rospy.Rate` to specify our desired loop timing. This powerful mechanism will be used in a moment to adaptively maintain a looping rate.

07.01.3 Publishing to the topic

The **`while`** loop in Figure 07.1 is pretty simple: while the node isn't shut down,

1. publish the count to topic `counter` via the `publish` method of the object `pub` created by `rospy.Publisher`,
2. increment the count, and
3. wait until the `sleep` method says to iterate.

The `Rate` object `rate` can use its `sleep` method to adaptively attempt to keep the loop running at the specified rate. This timing mechanism is quite convenient.

07.01.4 Running and verifying the node

First, we need to `catkin_make` the workspace to make our new package available. Navigate (`cd`) in Terminal to your workspace root directory.

```
catkin_make
```

If you have an error involving the Python packages `em`, `yaml`, or `catkin_pkg`, try installing them with the following.

```
pip install empy pyyaml catkin_pkg
```

Once your `catkin_make` finishes successfully, source the workspace.

```
source devel/setup.bash
```

Now open a new Terminal and start a `roscore` service. Now we can `rosrun` the new node!

```
rosrun my_topics topic_publisher.py
```

Our node is running! Let's check the current topics to see if `counter` is being advertised. A nice tool for this is `rostopic`.

```
rostopic list
```

```
/counter
/rosout
/rosout_agg
```

So it is. We can ignore the other topics, which always appear. Let's see what is being published to the topic.

```
rostopic echo counter -n 3
```

```
data: 17
---
data: 18
---
data: 19
---
```

The `-n 3` option/value shuts down `rostopic` after three messages. Otherwise it would continue until we Ctrl+C.

We can also see how the successful our `sleep` method is at maintaining our desired loop rate. (We have to Ctrl+C to stop this one.)

```
rostopic hz counter
```

```
subscribed to [/counter]
average rate: 2.001
    min: 0.500s max: 0.500s std dev: 0.00000s window: 2
average rate: 1.999
    min: 0.500s max: 0.501s std dev: 0.00051s window: 4
average rate: 2.000
    min: 0.498s max: 0.501s std dev: 0.00095s window: 6
average rate: 2.000
    min: 0.498s max: 0.501s std dev: 0.00088s window: 7
```

Not too bad!

Lecture 07.02 Subscribing to topics

Subscribing to topics with `rospy` involves two steps:

callback

1. defining a *callback* function that is called every time a message arrives (on the topics specified in a moment) and
2. registering the subscription with `roscore`.

The name of the callback function can be anything—say, `callback`, but its argument should be handled as a message of the correct type (i.e. the message type of the topic to which we are subscribing). Registering the subscription with `roscore` is accomplished with the `Subscriber` method as follows.

```
rospy.Subscriber(
  <topic name string>,        # e.g. 'cool_topic_bro'
  <message_type>,             # e.g. Int32 from std_msgs
  <callback function handle>  # e.g. callback
)
```

The first two arguments are the same as those of `rospy.Publisher`. The final argument is simply the name of the callback function from above.

07.02.1 Creating a simple subscriber node

The code accompanying the text has a simple subscriber node in the `rico_topics` package. You should use have used `catkin_create_pkg` in Lecture 07.01 to create a parallel package in your own code repository—we'll call it `my_topics`. Create a new Python file in `my_topics/src` with the following.

```
touch my_topics/src/topic_subscriber.py     # create file
chmod u+x my_topics/src/topic_subscriber.py # make executable
```

Open the empty `topic_subscriber.py` in a text editor. You'll want to enter here the same code as appears in the sample `topic_subscriber.py` from `robotics-book-code/rico_topics/src`, which is listed in Figure 07.2.

We see that the callback function definition `def callback(msg)` simply `print`s the message's data to the Terminal running the node. The call to to `rospy.Subscriber` register's (with `roscore`) this node's

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import Int32

def callback(msg):   # callback for receiving messages
  print(msg.data)    # print to Terminal

rospy.init_node('topic_subscriber') # initialize node

sub = rospy.Subscriber('counter', Int32, callback) # subscribe

rospy.spin() # wait for node to be shut down
```

Figure 07.2: `rico_topics/src/topic_subscriber.py` listing.

subscription to the topic `'counter'`, with its message type `Int32`, and directs messages to the callback function `callback`, just defined.

Finally, there's a call to `rospy.spin`. This function here acts to keep the node running (so it can receive messages) until it is explicitly shut down. It's doing something like the following.

```
while not rospy.core.is_shutdown():
  rospy.rostime.wallsleep(0.5) # seconds
```

07.02.2 Running and verifying the node

Now that we have created `my_topics/src/topic_subscriber.py`, we need to `catkin_make` and `source` our workspace.

```
cd ros_ws_01 # if needed
```

```
catkin_make
```

Now we can source our workspace.

```
source devel/setup.bash
```

Now, make sure you've started a `roscore` service running. If not, start it with the following.

```
roscore
```

Also make sure you still have the `topic_publisher.py` node running. If not, start it with the following.

```
rosrun my_topics topic_publisher.py
```

And now we're ready to launch the new `topic_subscriber.py` node.

```
rosrun my_topics topic_subscriber.py
```

```
100
101
102
```

The terminal prints the counter, as expected. To see who's publishing and subscribing to `counter`, we can use `rostopic` as follows.

```
rostopic info counter
```

```
Type: std_msgs/Int32

Publishers:
 * /topic_publisher (http://socrates:35309/)

Subscribers:
 * /topic_subscriber (http://socrates:40387/)
```

Just as we expected: `topic_publisher` is publishing to and `topic_subscriber` is subscribed to the topic `counter`.

07.02.3 Latched topics

latched topic

Sometimes a topic will have messages published so infrequently that it could be problematic if a subscriber misses a message because it was not-yet subscribed to the topic. In this case, we can publish a *latched topic*, which makes it so that every new subscriber gets the last message published to the topic. Latched topics are created with the `rospy.Publisher` named argument `latched=True`, which is by default `False`.

Table 07.1: built-in ROS field- and constant-types for messages.

type	serialization	C++	Python 2/3
`bool`	unsigned 8-bit int	**`uint8_t`**	`bool`
`int8`	signed 8-bit int	**`int8_t`**	`int`
`uint8`	unsigned 8-bit int	**`uint8_t`**	`int`
`int16`	signed 16-bit int	**`int16_t`**	`int`
`uint16`	unsigned 16-bit int	**`uint16_t`**	`int`
`int32`	signed 32-bit int	**`int32_t`**	`int`
`uint32`	unsigned 32-bit int	**`uint32_t`**	`int`
`int64`	signed 64-bit int	**`int64_t`**	`long`/`int`
`uint64`	unsigned 64-bit int	**`uint64_t`**	`long`/`int`
`float32`	32-bit IEEE float	**`float`**	`float`
`float64`	64-bit IEEE float	**`double`**	`float`
`string`	ascii string	`std::string`	`str`/`bytes`
`time`	sec/nsec unsigned 32-bit int	`ros::Time`	`rospy.Time`
`duration`	sec/nsec signed 32-bit int	`ros::Duration`	`rospy.Duration`

Lecture 07.03 Custom messages

The messages that come in the `std_msgs` should be exhausted before considering the specification of a new *message description*: a line-separated list of *field* type-name pairs and *constant* type-name-value triples. For example, the following is a message description with two fields and a constant.

message description
field
constant

```
int32 x      # field type: int32, name: x
float32 y    # field type: float32, name: y
int32 Z      # constant type: int32, name: Z
```

The field- and constant-types are usually ROS built-in types, which are shown in Table 07.1. Other field- and constant-types are possible, as described in the documentation:

`wiki.ros.org/msg`.

Of particular interest are arrays of built-in types, like the variable-length array of integers `int32[] foo`, which is interpreted as a Python `tuple`.

To use a custom message description, create a *`.msg` file* in the subdirectory `<package>/msg/` (you may need to create the subdirectory) and enter your message description.

`.msg` file

07.03.1 An example message description

In this section, we develop a custom message description `Complex` in `msg/Complex.msg` for messages with a real and an imaginary floating-point number. We continue to build on the package we've been creating in this chapter, `my_topics`, which shadows the package included with the book, `rico_topics`.

The first thing when creating a custom message description is to create the message description file.

07.03.1.1 Creating a message description

From your package root, create it with the following.

```
mkdir msg
touch msg/Complex.msg
```

Now we can edit the contents of `Complex.msg` to include the following.

```
float32 real
float32 imaginary
```

Both field types are `float32` and have field names `real` and `imaginary`.

We are now ready to update the build-system

07.03.1.2 Updating the build-system configuration

The package we've been working on in this chapter, `my_topics`, was created with a bit of forethought: we included as dependencies in our `catkin_create_pkg` call the packages `message_runtime` and `message_generation`. If we hadn't had such foresight, we would have to make several changes in our package's `package.xml` and `CMakeLists.txt` files before proceeding to create our own message description. As it stands, we still need to make a few changes to them.

How we need to change `package.xml`

Including `message_runtime` and `message_generation` in our `catkin_create_pkg` call yielded the following lines in our `package.xml`, which would otherwise need to be added manually.

```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```

However, we still need to add `message_runtime` as a **`<build_depend>`**.

```
<build_depend>message_runtime</build_depend>
```

How we need to change `CMakeLists.txt`

Including `message_runtime` and `message_generation` in our `catkin_create_pkg` call yielded the following lines in our `CMakeLists.txt`, which would otherwise need to be added manually. As an additional line in the `find_package(...)` block, we would need the following.

```
message_generation
```

The rest of the changes we do need to make manually. As an additional line in the `catkin_package(...)` block, we need the following.

```
CATKIN_DEPENDS message_runtime
```

The `add_message_files(...)` block needs uncommented and edited to appear as follows.

```
add_message_files(
  FILES
  Complex.msg
)
```

We have already created the `Complex.msg` file.

Finally, the `generate_messages(...)` block needs to be uncommented such that it appears as follows.

```
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

Now our package is set up to use the message type `Complex`—or, it will be once we `catkin_make` our workspace. First, let's write a simple publisher and subscriber to try it out.

```python
#!/usr/bin/env python
import rospy
from rico_topics.msg import Complex # custom message type
from random import random # for random numbers!

rospy.init_node('message_publisher') # initialize node

pub = rospy.Publisher(    # register topic
  'complex',              # topic name
  Complex,                # custom message type
  queue_size=3            # queue size
)

rate = rospy.Rate(2) # set rate

while not rospy.is_shutdown(): # loop
  msg = Complex()            # declare type
  msg.real = random()        # assign value
  msg.imaginary = random()   # assign value

  pub.publish(msg)  # publish!
  rate.sleep()      # sleep to keep rate
```

Figure 07.3: `rico_topics/src/message_publisher.py` listing.

07.03.1.3 Writing a publisher and subscriber

We can now write a publisher and subscriber that publish and subscribe to messages with type `Complex`. Create (`touch`) a Python node file `my_topics/src/message_publisher.py`, change its permissions to user-executable (`chmod u+x`), and edit it to have the same contents as the `rico_topics/src/message_publisher.py` file shown in Figure 07.3.

Repeat a similar process to create a `my_topics/src/message_subscriber.py` with the same contents as the `rico_topics/src/message_subscriber.py` file shown in Figure 07.4.

07.03.1.4 Running and verifying these nodes

Let's try it out. Navigate to your workspace root and build your workspace.

```
catkin_make
```

```
#!/usr/bin/env python
import rospy
from rico_topics.msg import Complex

def callback(msg):
    print 'Real:', msg.real           # print real part
    print 'Imaginary:', msg.imaginary # print imag part
    print                             # blank line

rospy.init_node('message_subscriber') # initialize node

sub = rospy.Subscriber( # register subscription
  'complex',            # topic
  Complex,              # custom type
  callback              # callback function
)

rospy.spin() # keep node running until shut down
```

Figure 07.4: `rico_topics/src/message_subscriber.py` listing.

Fire up a `roscore`. In a new Terminal, in your workspace root, `source devel/setup.bash` then run the publisher node.

```
rosrun my_topics message_publisher.py
```

In another new Terminal, in your workspace root, again `source devel/setup.bash` then run the subscriber node.

```
rosrun my_topics message_subscriber.py
```

```
Real: 0.308157861233
Imaginary: 0.229206711054

Real: 0.121079094708
Imaginary: 0.568501293659

Real: 0.807860195637
Imaginary: 0.486804276705
```

It works! Random complex numbers are being printed by the `message_subscriber.py` node.

Lecture 07.04 Other considerations

ROS topics have hardly been exhausted, and this will remain true even after we consider a few more aspects of special note.

07.04.1 The `rosmsg` command

The `rosmsg` command comes with the `rosbash` package already installed. It allows us to explore which messages are described and their descriptions.

As always, we need to navigate to our workspace.

```
cd ros_ws_01
```

Then source it!

```
source devel/setup.bash
```

The `show` option lists message descriptions. Even our `Complex` custom definition can be listed in this way.

```
rosmsg show Complex
```

```
[rico_topics/Complex]:
float32 real
float32 imaginary

[my_topics/Complex]:
float32 real
float32 imaginary
```

This is how we could see the message description of a `geometry_msgs` message `Point`.

```
rosmsg show geometry_msgs/Point
```

```
float64 x
float64 y
float64 z
```

The `package` option lets us list those messages defined in a given package.

```
rosmsg package my_topics
```

```
my_topics/Complex
```

For the `tf2_msgs` package, which groups the Error and Transform messages for `tf2_ros`, several message definitions are provided.

```
rosmsg package tf2_msgs
```

```
tf2_msgs/LookupTransformAction
tf2_msgs/LookupTransformActionFeedback
tf2_msgs/LookupTransformActionGoal
tf2_msgs/LookupTransformActionResult
tf2_msgs/LookupTransformFeedback
tf2_msgs/LookupTransformGoal
tf2_msgs/LookupTransformResult
tf2_msgs/TF2Error
tf2_msgs/TFMessage
```

The `list` option lists all messages available to ROS.

```
rosmsg list
```

We have suppressed the output, which is long.

07.04.2 Publishing and subscribing in the same node

Why not? This is actually rather common. Consider the example node `robotics-book-code/rico_topics/doubler.py`, listed in Figure 07.5. This node subscribes to topic `number`, multiplies the received `msg.data` (an `Int32`) by two, and publishes the result (an `Int32`) to topic `doubled`.

Perhaps the most interesting aspect of this is that, instead of publishing at some set rate, the publishing happens *inside the callback*. This means a new message will be published to `doubled` right after a new message is published to topic `number`. This is frequently the most desirable behavior.

```python
#!/usr/bin/env python
import rospy
from std_msgs.msg import Int32

rospy.init_node('doubler') # initialize node

def callback(msg):
    doubled = Int32()            # declare
    doubled.data = msg.data * 2 # double
    pub.publish(doubled)         # publish in callback!

sub = rospy.Subscriber('number', Int32, callback)
pub = rospy.Publisher('doubled', Int32, queue_size=3)

rospy.spin() # keep node running until shut down
```

Figure 07.5: `rico_topics/src/doubler.py` listing.

08

ROS services

Lecture 08.01 Introducing ROS services

services
server
clients

A ROS *service* is effectively a function one node (the *server*) provides to other nodes (the *clients*).

Box 08.1 *i can haz service?* a script

Pretty much, if we have Node A [server] and Node B [client]:
Node A: "Yo I can do X service [registers a service],"
Node B: "Node A, do X for me plz? [requests service]" and waits
Node A: does X [service occurs]
Node A: sends Node B the result of doing X [server returns values]
Node B: "thnks fam" [I just assume this happens].

synchronous

A key aspect to services is that they are *synchronous*: a client *waits*, doing nothing else, while the server "services" it. So obviously this only works well for tasks that take a limited amount of time, such as:

1. getting a sensor value,
2. setting a parameter, or
3. performing a computation.

08.01.1 An example service type definition

In this section, we develop a custom service type definition `WordCount` in `srv/WordCount.srv` for a service that has as input a string and as output the number of words in that string. We create a new package for this chapter, `my_services`, which shadows the package included with the book, `rico_services`. So use, in your workspace's `src` directory, use `catkin_create_pkg` to create a package, as follows.

```
catkin_create_pkg my_services \
  roscpp rospy message_generation message_runtime
```

The first thing when creating a custom service definition is to create the service definition file.

08.01.1.1 Creating a service definition

From your package root, create it with the following.

```
mkdir srv # traditional directory for service definitions
touch srv/WordCount.srv
```

Now we can edit the contents of `WordCount.srv` to include the following.

```
string words
---
uint32 count
```

Above the delimiter "`---`" are *input field* types and names and below the delimiter are *output field* types and names.

input field
output field

We are now ready to update the build-system.

08.01.1.2 Updating the build-system configuration

The package we're creating in this chapter, `my_services`, was created with a bit of forethought: we included as dependencies in our `catkin_create_pkg` call the packages `message_runtime` and `message_generation`. If we hadn't had such foresight, we would have to make several changes in our package's `package.xml` and `CMakeLists.txt` files before proceeding to create our own message description. As it stands, we still need to make a few changes to them.

How we need to change `package.xml`

Including `message_runtime` and `message_generation` in our `catkin_create_pkg` call yielded the following lines in our `package.xml`, which would otherwise need to be added manually.

```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```

However, we still need to add `message_runtime` as a **`<build_depend>`**.

```
<build_depend>message_runtime</build_depend>
```

How we need to change `CMakeLists.txt`

Including `message_runtime` and `message_generation` in our `catkin_create_pkg` call yielded the following lines in our `CMakeLists.txt`, which would otherwise need to be added manually.

As an additional line in the `find_package(...)` block, we would need the following.

```
message_generation
```

The rest of the changes we do need to make manually. The `add_service_files(...)` block needs uncommented and edited to appear as follows.

```
add_service_files(
  FILES
  WordCount.srv
)
```

We have already created the `WordCount.srv` file.

Finally, the `generate_messages(...)` block needs to be uncommented such that it appears as follows.

```
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

Now our package is set up to use the service type `WordCount`—or, it will be once we `catkin_make` our workspace. (Go ahead and do so now.)

08.01.1.3 See it with `rossrv`

The package `rosmsg` (already installed) includes the command `rossrv`, which gives information about services.

```
rossrv
```

```
rossrv is a command-line tool for displaying information about ROS
↪  Service types.
Commands:
  rossrv show     Show service description
  rossrv info     Alias for rossrv show
  rossrv list     List all services
  rossrv md5      Display service md5sum
  rossrv package  List services in a package
  rossrv packages List packages that contain services
```

We could try it on our new service type `WordCount` as follows.

```
rossrv show WordCount
```

```
[my_services/WordCount]:
string words
---
uint32 count

[rico_services/WordCount]:
string words
---
uint32 count
```

So the `WordCount` service type is available in both packages `rico_services` and `my_services`. We have successfully created a service type! In Lecture 08.02, we'll learn to serve and call this service type.

Lecture 08.02 Serving and calling a ROS service

server node Creating a *server node* is our first consideration.

08.02.1 Creating a server node

Here are some key aspects of a `rospy` server, listed below as instructions for creating such a node.

1. Import the service type and its `Response` function:
   `from <pkg>.srv import <srv_type> <srv_type>Response`.
2. Define a function to serve:
   `def fun(request)`.
3. Register a service:
   `rospy.Service(<srv_name>,<srv_type>,<fun>)`.
4. Wait for service requests: `rospy.spin()`.

The service function can return:

1. a `<srv_type>Response` object:
   `return <srv_type>Response(<value1>,<value2>,...)` or
2. a single value (matching a single service output type):
   `return <value>` or
3. a `list` of values (matching the output types):
   `return [<value1>,<value2>,...]` or
4. a `dict`ionary of values (matching the output names and types):
   `return {'name1':<value1>,'name2':<value2>}`.

08.02.2 An example server node

Let's implement our new service `word_count`, created in Lecture 08.01. We need a server node to do so. Create (`touch`) a Python node file `my_services/src/service_server.py`, change its permissions to user-executable (`chmod u+x`), and edit it to have the same contents as the `rico_services/src/service_server.py` file shown in Figure 08.1.

08.02.3 Creating a client node

client node The key elements to creating a *client node* are:

1. Import the service:
   `from <pkg>.srv import <srv_type>`.

```python
#!/usr/bin/env python
import rospy
from rico_services.srv import WordCount, WordCountResponse

def count_words(request):
  return len(request.words.split()) # num of words

rospy.init_node('service_server')

service = rospy.Service( # register service
  'word_count', # service name
  WordCount,    # service type
  count_words   # function service provides
)

rospy.spin()
```

Figure 08.1: `rico_services/src/service_server.py` listing.

2. Wait for a service:
   `rospy.wait_for_service('service_name')`.
3. Set up a proxy server for communication:
   `rospy.ServiceProxy(<srv_name>,<srv_type>)`.
4. Use the service: `fun(...)`.

08.02.4 An example client node

Let's create a client for our new service `word_count`. We need a client node to do so. Create (`touch`) a Python node file `my_services/src/service_client.py`, change its permissions to user-executable (`chmod u+x`), and edit it to have the same contents as the `rico_services/src/service_client.py` file shown in Figure 08.2.

The only thing that may surprise us here is the line `words = ' '.join(sys.argv[1:])`. The inner statement `sys.argv[1:]` returns a `list` of command-line arguments supplied to the node. Then `' '.join(...)` concatenates the (`string`) elements of the list with a space character between each pair. This is one of many ways we could *parse* command-line arguments.

argument parsing

```
#!/usr/bin/env python
import rospy
from rico_services.srv import WordCount
import sys

rospy.init_node('service_client')

rospy.wait_for_service('word_count') # wait for registration
word_counter = rospy.ServiceProxy( # set up proxy
  'word_count', # service name
  WordCount     # service type
)
words = ' '.join(sys.argv[1:]) # parse args
word_count = word_counter(words) # use service

print(words+'--> has '+str(word_count.count)+' words')
```

Figure 08.2: `rico_services/src/service_client.py` listing.

08.02.5 Running and verifying the server and client nodes

Navigate to your workspace root and build the workspace.

```
catkin_make
```

Run a `roscore`. In a new Terminal, in your workspace root, `source devel/setup.bash`, then run the server node.

```
rosrun my_services service_server.py
```

In a new Terminal, in your workspace root, `source devel/setup.bash`, then run the client node with command line arguments passed.

```
rosrun my_services service_client.py hello world sweet world
```

```
hello world sweet world--> has 4 words
```

It works!

09

ROS actions

Lecture 09.01 Introducing ROS actions

actions
action server
asynchronicity
clients
goals
results
feedback

A ROS *action* is effectively a function one node (the *action server*) *asynchronously* provides to other nodes (the *clients*). Note this is just like *service*, but with the asynchronicity of a *topic*. Like a service, an action has a *goal* and a *result*; but unlike a service, an action also provides *feedback* during execution. This makes actions more suitable for goal-oriented tasks that take time, such as:

1. navigating to a location,
2. performing a complex manipulation, or
3. performing a long calculation.

09.01.1 An example action type definition

In this section, we develop a custom action type definition `Timer` in `action/Timer.action` for an action that has as

input a `duration` to wait `time_to_wait`;
output a total actual `duration` waited `time_elapsed` and a total `uint32` number of feedback updates sent `updates_sent`; and
feedback a `duration` waited so far `time_elapsed` and a `duration` left to wait `time_remaining`.

Box 09.1 why a timer though

The `Timer` action is for demonstration purposes only and shouldn't be used to implement timing in a ROS graph. For timing, use `rospy.sleep()`.

We create a new package for this chapter, `my_actions`, which shadows the package included with the book, `rico_actions`. So, in your workspace's `src` directory, use `catkin_create_pkg` to create a package, as follows.

```
catkin_create_pkg my_actions roscpp rospy actionlib_msgs
```

action definition file

The first thing when creating a custom action definition is to create the *action definition file*.

09.01.1.1 Creating an action definition

From your package root, create it with the following.

```
mkdir action
touch action/Timer.action
```

Now we can edit the contents of `Timer.action` to include the following.

```
# inputs
duration time_to_wait
---
# outputs
duration time_elapsed
uint32 updates_sent
---
# feedback
duration time_elapsed
duration time_remaining
```

Above the first delimiter "`---`" are *input field* types and names; between the delimiters are *output field* types and names; and after the second delimiter are *feedback field* types and names.

input field
output field
feedback field

We are now ready to update the build-system.

09.01.1.2 Updating the build-system configuration

The package we're creating in this chapter, `my_actions`, was created with a bit of forethought: we included as dependencies in our `catkin_create_pkg` call the package `actionlib_msgs` for creating actions. If we hadn't had such foresight, we would have to make several changes in our package's `package.xml` and `CMakeLists.txt` files before proceeding to create our own message description. As it stands, we still need to make a few changes to them.

How we *would have* had to change `package.txt`

Including `actionlib_msgs` in our `catkin_create_pkg` call yielded the following lines in our `package.xml`, which would otherwise need to be added manually.

```
<build_depend>actionlib_msgs</build_depend>
<build_exec_depend>actionlib_msgs</build_exec_depend>
<exec_depend>actionlib_msgs</exec_depend>
```

How we need to change `CMakeLists.txt`

Including `actionlib_msgs` in our `catkin_create_pkg` call yielded the following lines in our `CMakeLists.txt`, which would otherwise need to be added manually. As an additional line in the `find_package(...)` block, we would need the following.

```
actionlib_msgs
```

The rest of the changes we do need to make manually. The `add_action_files(...)` block needs uncommented and edited to appear as follows.

```
add_action_files(
  DIRECTORY action
  FILES Timer.action
)
```

We have already created the `Timer.action` file.

The `generate_messages(...)` block needs to be uncommented and `actionlib_msgs` added such that it appears as follows.

```
generate_messages(
  DEPENDENCIES
  actionlib_msgs
  std_msgs
)
```

Finally, the `catkin_package` block also needs uncommented and `actionlib_msgs` added such that it appears as follows.

```
catkin_package(
  CATKIN_DEPENDS
  actionlib_msgs
)
```

Now our package is set up to use the action type `Timer`—or, it will be once we `catkin_make` our workspace. (Go ahead and do so now.) As before with services, `catkin_make` will take our action definition and create several message definition `.msg` files. This highlights the fact that an action communicates via services.

We have successfully created an action type! In Lecture 09.02, we'll learn to serve and call this action type.

Lecture 09.02 Serving and calling a ROS action

Creating an *action server node* is our first consideration.

action server node

09.02.1 Creating an action server node

Here are some key aspects of a `rospy` action server, listed below as instructions for creating such a node.

1. Import the Python package `actionlib`:

```
import actionlib
```

2. Import the action's generated message types:

```
from <pkg>.msg import <action_type>Action, \
  <action_type>Goal, <action_type>Result, <action_type>Feedback
```

3. Define an action function to serve:

```
def do_action(goal):
```

   a) Check for errors in client request and abort with result if necessary:

```
result = <action_type>Result() # create result object
result.<field_name> = <field_value> # set result(s)
server.set_aborted(result,"An abort message")
```

   b) While `goal` isn't yet met, do the following in a timed loop.

      i. Check for client request to preemptively abort goal and abort with result if necessary:

```
if server.is_preempt_requested():
  result = <action_type>Result() # create result obj
  server.set_preempted(result,"Preempted abort msg")
  return
```

      ii. Set and publish feedback:

```
feedback = <action_type>Feedback() # create fdbck obj
feedback.<field_name> = <field_value> # set feedback
server.publish_feedback(feedback) # publish feedback
```

   c) When `goal` is met, publish the result:

```
result = <action_type>Result() # create result object
result.<field_name> = <field_value> # set result(s)
server.set_succeeded(result,"A success message")
```

4. Register an action:

```
server = actionlib.SimpleActionServer(
  'server_name',          # server name string
  <action_type>Action,    # action Action message type
```

```
    do_action,              # action function
    False # autostart server? always set to False
)
```

5. Start the action and wait for requests:

```
server.start()
rospy.spin()
```

09.02.2 An example action server node

Let's implement our new action `Timer`, created in Lecture 09.01. We need an action server node to do so. Create (`touch`) a Python node file `my_actions/src/fancy_action_server.py`, change its permissions to user-executable (`chmod u+x`), and edit it to have the same contents as the `rico_services/src/fancy_action_server.py` file shown in Figure 09.1 (be sure to change `rico_actions` to `my_actions`).

```python
#! /usr/bin/env python
import rospy
import time       # for regular Python timing
import actionlib  # for actions!
from rico_actions.msg import \
    TimerAction, TimerGoal, TimerResult, TimerFeedback

def do_timer(goal): # action function
    start_time = time.time()
    update_count = 0
    if goal.time_to_wait.to_sec() > 60.0: # check req duration
        result = TimerResult()
        result.time_elapsed = rospy.Duration.from_sec(
            time.time() - start_time)
        result.updates_sent = update_count
        server.set_aborted(result, "Aborted: too long to wait")
        return # too long of a requested wait
    while (time.time()-start_time) < goal.time_to_wait.to_sec():
        # waiting to meet goal duration
        if server.is_preempt_requested(): # check preemption
            result = TimerResult()
            result.time_elapsed = rospy.Duration.from_sec(
                time.time() - start_time)
            result.updates_sent = update_count
            server.set_preempted(result, "Timer preempted")
            return
        feedback = TimerFeedback()
        feedback.time_elapsed = rospy.Duration.from_sec(
            time.time() - start_time)
        feedback.time_remaining = \
            goal.time_to_wait - feedback.time_elapsed
        server.publish_feedback(feedback)
        update_count += 1
        time.sleep(1.0)
    result = TimerResult()
    result.time_elapsed = rospy.Duration.from_sec(
        time.time() - start_time)
    result.updates_sent = update_count
    server.set_succeeded(result, "Timer completed successfully")

rospy.init_node('timer_action_server') # initialize node
server = actionlib.SimpleActionServer(
    'timer', TimerAction, do_timer, False
)
server.start()
rospy.spin()
```

Figure 09.1: `rico_actions/src/fancy_action_server.py` listing.

09.02.3 Creating an action client node

action client node

The key elements to creating an *action client node* are:

1. Import the Python package `actionlib`:
   ```
   import actionlib
   ```
2. Import the action's generated message types:
   ```
   from <pkg>.msg import <action_type>Action, \
     <action_type>Goal, \
     <action_type>Result, \
     <action_type>Feedback
   ```
3. Define a feedback callback function:
   ```
   def feedback_cb(feedback):
   ```
   Do whatever you like with the feedback in here.
4. Register an action client and wait for server connection:
   ```
   client = actionlib.SimpleActionClient(
     'server_name',      # action server name
     <action_type>Action # action Action message
   )
   client.wait_for_server()
   ```
5. Define and send `goal` to server:
   ```
   goal = <action_type>Goal()
   goal.<field_name> = <field_value> # set goal field(s)
   client.send_goal(goal, feedback_cb=feedback_cb) # send to
   ```
6. Wait for results, then do what you like with them:
   ```
   client.wait_for_result()
   ```

09.02.4 An example action client node

Let's create a client for our new action `Timer`. We need a client node to do so. Create (`touch`) a Python node file `my_actions/src/fancy_action_client.py`, change its permissions to user-executable (`chmod u+x`), and edit it to have the same contents as the `rico_actions/src/fancy_action_client.py` file shown in Figure 09.2 (be sure to change `rico_actions` to `my_actions`).

09.02.5 Launching and verifying the server and client nodes

launch file

Let's make a *launch file* for our action server and client nodes. Navigate to your `my_actions` directory and create (`touch`) a file `fancy_action.launch`. Now edit it to include the contents shown in Figure 09.3.

```python
#! /usr/bin/env python
import rospy
import time          # for regular Python timing
import actionlib     # for actions!
from rico_actions.msg import \
    TimerAction, TimerGoal, TimerResult, TimerFeedback

def the_feedback_cb(feedback): # feedback callback function
    print('[Feedback] Time elapsed: %f' %
      (feedback.time_elapsed.to_sec()))
    print('[Feedback] Time remaining: %f' %
      (feedback.time_remaining.to_sec()))

rospy.init_node('timer_action_client') # initialize node
client = actionlib.SimpleActionClient( # register client
    'timer',         # action server name
    TimerAction    # action Action message
)
client.wait_for_server()     # wait for action server
goal = TimerGoal()            # create goal object
goal.time_to_wait = rospy.Duration.from_sec(5.0) # set field
# Uncomment this line to test server-side abort:
# goal.time_to_wait = rospy.Duration.from_sec(500.0)

client.send_goal(goal, feedback_cb=the_feedback_cb ) # send goal
# Uncomment these lines to test goal preemption:
# time.sleep(3.0)
# client.cancel_goal()

client.wait_for_result() # wait for action server to finish
# print results:
print('[Result] State: %d' % (client.get_state()))
print('[Result] Status: %s' % (client.get_goal_status_text()))
if client.get_result():
    print('[Result] Time elapsed: %f' %
        (client.get_result().time_elapsed.to_sec()))
    print('[Result] Updates sent: %d' %
        (client.get_result().updates_sent))
```

Figure 09.2: `rico_actions/src/fancy_action_client.py` listing.

```
<launch>
  <node name="server" pkg="my_actions"
  ↪  type="fancy_action_server.py" output="screen"/>
  <node name="client" pkg="my_actions"
  ↪  type="fancy_action_client.py" output="screen"/>
</launch>
```

Figure 09.3: code listing for launch file `fancy_action.launch`.

Navigate to your workspace root and build the workspace.

```
catkin_make
```

Source your workspace: `source devel/setup.bash`. Now launch the ROS graph with the following.

```
roslaunch my_actions fancy_action.launch
```

```
... logging to /home/socrates/.ros/log/....log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.
started roslaunch server http://socrates:44495/
SUMMARY
========
PARAMETERS
 * /rosdistro: melodic
 * /rosversion: 1.14.5
NODES
  /
    client (rico_actions/fancy_action_client.py)
    server (rico_actions/fancy_action_server.py)
auto-starting new master
process[master]: started with pid [9827]
ROS_MASTER_URI=http://localhost:11311
setting /run_id to 9f2db0a0-8c0f-11ea-a1ae-0800272f9db4
process[rosout-1]: started with pid [9838]
started core service [/rosout]
process[server-2]: started with pid [9845]
process[client-3]: started with pid [9846]
[Feedback] Time elapsed: 0.000013
[Feedback] Time remaining: 4.999987
[Feedback] Time elapsed: 1.002952
[Feedback] Time remaining: 3.997048
```

```
[Feedback] Time elapsed: 2.008340
[Feedback] Time remaining: 2.991660
[Feedback] Time elapsed: 3.017035
[Feedback] Time remaining: 1.982965
[Feedback] Time elapsed: 4.022336
[Feedback] Time remaining: 0.977664
[Result] State: 3
[Result] Status: Timer completed successfully
[Result] Time elapsed: 5.028862
[Result] Updates sent: 5
[client-3] process has finished cleanly
log file: /home/socrates/.ros/log/....log
```

We see the feedback printing as expected, along with the final results. The resulting goal status, accessed by `client.get_goal_status_text()`, is `'Timer completed successfully'`.

09.02.6 More `actionlib` documentation

The core of what we have used to construct our action server and client is the `actionlib` ROS package:

`wiki.ros.org/actionlib`.

The `rospy` (Python) library API is here for the `SimpleActionServer` class:

`ricopic.one/redirect/SimpleActionServer`.

And here for the `SimpleActionClient` class:

`ricopic.one/redirect/SimpleActionClient`.

Part III

Open-loop control with ROS

Part IV

Closed-loop control with ROS

Part V

Control architectures with ROS

Start with (Koubaa, 2017, pp. 124-5). Koubaa (2020, 2018, 2017, 2016)

10

Bibliography

A. Agarwal and J. Lang. *Foundations of Analog and Digital Electronic Circuits*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2005. ISBN 9780080506814. URL https://books.google.com/books?id=lGgP7FDEv3AC.

R.C. Arkin, R.P.D.M.R.L.R.C. Arkin, M.I.T. Press, and R.C. Arkin. *Behavior-based Robotics*. Bradford book. MIT Press, 1998. ISBN 9780262011655. URL https://books.google.com/books?id=mRWT6alZt9oC.

Richard C. Booton and Simon Ramo. The development of systems engineering. *IEEE Transactions on Aerospace and Electronic Systems*, AES–20: 306–9, July 1984.

William L Brogan. *Modern Control Theory*. Prentice Hall, third edition, 1991.

Rodney A. Brooks. *Cambrian Intelligence: The Early History of the New AI*. A Bradford book. BRADFORD BOOK, 1999. ISBN 9780262522632. URL https://books.google.com/books?id=btvRZ5rj51EC.

Francesco Bullo and Andrew D. Lewis. *Geometric control of mechanical systems: modeling, analysis, and design for simple mechanical control systems*. Springer, 2005.

E. Charniak. *Introduction to Deep Learning*. Mit Press. MIT Press, 2019. ISBN 9780262039512.

Matthew F. Hale, Edgar Buchanan, Alan F. Winfield, Jon Timmis, Emma Hart, Agoston E. Eiben, Mike Angus, Frank Veenstra, Wei Li, Robert

Woolley, Matteo De Carlo, and Andy M. Tyrrell. The are robot fabricator: How to (re)produce robots that can evolve in the real world. *Artificial Life Conference Proceedings*, (31):95–102, 2019. doi: 10.1162/isal_a_00147. URL https://www.mitpressjournals.org/doi/abs/10.1162/isal_a_00147.

Guy Hoffman and Cynthia Breazeal. *Collaboration in Human-Robot Teams*. American Institute of Aeronautics and Astronautics, 2004. doi: 10.2514/6.2004-6434. URL https://arc.aiaa.org/doi/abs/10.2514/6.2004-6434.

P Horowitz and W Hill. *The Art of Electronics*. Cambridge University Press, 2015. ISBN 9780521809269. URL https://books.google.com/books?id=LAiWPwAACAAJ.

Anis Koubaa, editor. *Robot Operating System (ROS): The Complete Reference (Volume 1)*. Studies in Computational Intelligence 625. Springer International Publishing, 1 edition, 2016. ISBN 978-3-319-26052-5,978-3-319-26054-9. URL http://gen.lib.rus.ec/book/index.php?md5=f914a41247978eb9baf42bfce070cca4.

Anis Koubaa, editor. *Robot Operating System (ROS): The Complete Reference (Volume 2)*. Studies in Computational Intelligence 707. Springer International Publishing, 1 edition, 2017. ISBN 978-3-319-54926-2, 978-3-319-54927-9. URL http://gen.lib.rus.ec/book/index.php?md5=4e5042943712194093c578a7c8ea9679.

Anis Koubaa, editor. *Robot Operating System (ROS): The Complete Reference (Volume 3)*, volume 3 of *Studies in Computational Intelligence 778*. Springer, 2018. ISBN 978-3-319-91590-6. URL http://gen.lib.rus.ec/book/index.php?md5=b98448ecef80c88786fc9703dcb5ffd5.

Anis Koubaa, editor. *Robot Operating System (ROS): The Complete Reference (Volume 4)*. Studies in Computational Intelligence 831. Springer International Publishing, 1st ed. edition, 2020. ISBN 978-3-030-20189-0;978-3-030-20190-6. URL http://gen.lib.rus.ec/book/index.php?md5=11871d8ea203df5ddea8fca33c407658.

J. Laird. *The Soar Cognitive Architecture*. Mit Press. MIT Press, 2012. ISBN 9780262122962. URL https://books.google.com/books?id=X4gtrFSlbosC.

Maja J. Matarić and François Michaud. *Behavior-Based Systems*, pages 891–909. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-30301-5. doi: 10.1007/978-3-540-30301-5_39. URL https://doi.org/10.1007/978-3-540-30301-5_39.

Anand S.; Singh Munindar P. Müller, Jörg P.; Rao. *[Lecture Notes in Computer Science] Intelligent Agents V: Agents Theories, Architectures, and Languages Volume 1555 || The Belief-Desire-Intention Model of Agency*, volume 10.1007/3-540-49057-4. 1999. ISBN 978-3-540-65713-2,978-3-540-49057-9. doi: 10.1007/3-540-49057-4_1. URL http://gen.lib.rus.ec/scimag/index.php?s=10.1007/3-540-49057-4_1.

Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. Adaptive Computation and Machine Learning. The MIT Press, 1 edition, 2012. ISBN 0262018020,9780262018029.

N.S. Nise. *Control Systems Engineering, 7th Edition*. Wiley, 2015. ISBN 9781118800829. URL https://books.google.com/books?id=BwTYBgAAQBAJ.

R. Pfeifer and J. Bongard. *How the Body Shapes the Way We Think: A New View of Intelligence*. MIT Press, 2006. ISBN 9780262288521. URL https://books.google.com/books?id=EHPMv9MfgWwC.

Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

Derek Rowell and David N. Wormley. *System Dynamics: An Introduction*. Prentice Hall, 1997.

S.J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall series in artificial intelligence. Prentice Hall, 2010. ISBN 9780136042594. URL https://books.google.com/books?id=8jZBksh-bUMC.

Francisco Varela. The emergent self. Web, May 1996. URL https://www.edge.org/conversation/francisco_varela-chapter-12-the-emergent-self.

John Von Neumann and Arthur W. Burks. Theory of self-reproducing automata, 1966. URL https://archive.org/details/theoryofselfrepr00vonn_0.

Slavoj Žižek. *How to Read Lacan*. W W Norton & Company Inc, 2006.

Ludwig Wittgenstein and G.E.M. Anscombe. *Philosophical Investigations*. Blackwell Publishing, 2001.

Stephen Wolfram. *A New Kind of Science*. Wolfram Media, 1 edition, 2002. ISBN 1-57955-008-8,9781579550080.

Stephen Wolfram. A new kind of science: A 15-year review. Web, May 2017. URL https://writings.stephenwolfram.com/2017/05/a-new-kind-of-science-a-15-year-view/.

Slavoj Žižek. *Less Than Nothing: Hegel and the Shadow of Dialectical Materialism*. Verso, 2012. ISBN 9781844678976.