TDT4113 (PLAB2)
Project 6: Robot

— Assignment Sheet —

Purpose:

- Learn a simple, but powerful, general strategy for robot control.
- Gain hands-on experience with robotics programming from the lowest levels of sensors and actuators to the highest levels of behavior control.

Practical Information

- This project will be done in groups of size four. Each group will be issued a robot for the three weeks of the project and will return it immediately after their demonstration.
- The due date for the demonstration is
  Wednesday, November 6, 2019 (kl 10:00)
  — we strongly recommend to prepare for presenting your solutions significantly earlier.
- Your code must be uploaded to BLACKBOARD prior to 8:00 (in the morning) on November 6.
- Your code needs to have a pylint level of at least 8.0

1 Introduction

In the early days of Artificial Intelligence (AI), the standard robot control architecture was a strict hierarchy whose lowest levels, sensors and actuators, were cleanly separated from high-level decision-making modules. As depicted in Figure 1, a typical timestep began with the input and interpretation of data from a multitude of sensors.

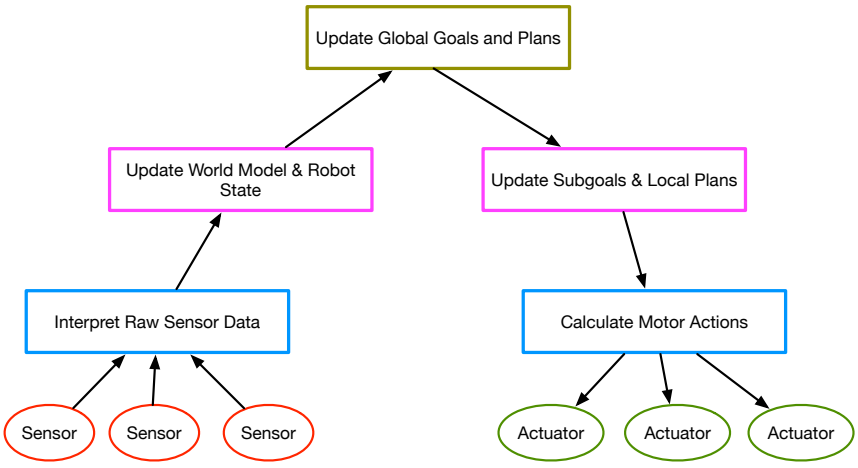

Figure 1: A typical example of the classic AI robotic control hierarchy in which activity flows from primitive activity at the bottom left, up through higher-level activities such as model updating and planning, and then back down to individual signals sent to actuators.

This information governed the updating of a *world model* that the robot maintained. With an updated model, the robot could then **plan** a **coarse sequence** of activities to achieve it goals. This, in turn, could be broken into a finer sequence of actions to achieve intermediate subgoals, and these low-level action plans could translate into simple commands to the motor actuators.

The problem with this classic, hierarchical approach is that world models are notoriously inaccurate due to the combination of nearly unavoidable sensor noise and the ever-changing nature of our world. Actuators are no less noisy; for example, it is very hard to make a robot turn exactly 90 degrees due to the friction of a contact surface, small discrepancies in the strengths of the wheel motors, etc. So even the best-laid plans will not produce the exact goal- and sub-goal-satisfying states that the world-model-based AI planning system may have predicted.

Behavior-Based Robotics (BBR) follows an entirely different philosophy: one motivated by the motto *the world is its own best model*. Instead of wasting valuable computing resources building inaccurate models and crafting unachievable plans, BBR manifests a more reactive approach, inspired by the psychological theory of *behaviorism*, in which the current state of the environment is believed to convey enough information to determine appropriate actions.

As shown in Figure 2, a behavior-based robot has a much flatter hierarchy in which a set of behaviors compete for *attention*, with the winner having the greatest influence in choosing the next action. Each such behavior has a direct connection to sensors and actuators, but only to those of particular relevance to itself.

As a simple (mildly hypothetical) example, a robot fielder (in baseball or cricket) might have a *track ball* behavior whose primary sensory input would be from a camera that tracked the flight, roll or bounce of a ball. However, it would not use data from other cameras or distance sensors (that might allow it to avoid running into other players, or stadium walls). Those would be the province of *collision avoidance* behaviors.

On the actuator side, ball tracking might only involve motors controlling the rotations of the camera mount, while collision avoidance would control the wheels or artificial legs of the fielder.

In BBR, there is no point at which all of the sensory input becomes integrated into a world model. The closest thing to integration is the *arbitration* stage, wherein an *arbitrator* module receives the motor requests of each behavior and determines which can actually be performed during the current timestep.

This preponderance of semi-independent modules with only weak integration supports a very versatile robotic design strategy. Fully working robots can be built with only a few behaviors, but as more complex functionality is required, additional behaviors can be coded up and combined into the existing architecture. In most cases, neither the older behaviors, nor the arbitrator, require any modifications. Modularity yields extensibility.

For example, consider a (relatively dumb) robot soccer player that knows how to move about in a restricted area (marked by white field lines), via behaviors such as *wander-randomly*, *avoid-collisions* and *avoid-crossing-white-lines*, and to visually follow the ball, via a *track-ball* behavior. It can be modfied to play defense with the addition of a new behavior to stay on a line between the ball and the center of the goal, and to play goalkeeper with another behavior to keep it inside the goal box (whose color would need to be something other than white to keep the other players from congregating there). The original behaviors can remain intact, although the priorities associated with each behavior (and used by the arbitrator) might need to change.

2 Implementing a Behavior-Based Robotic Controller

The robotics literature contains many alternatives for BBR controllers; we will not review them here. The fundamental features are

1. the modularity of behaviors (and the extensibility it supports),
2. the flat hierarchy in which behaviors have direct access to sensory information (either raw or pre-processed) and can make direct requests for motor actions,
3. an arbitrator to resolve the motor requests coming from the active behaviors.

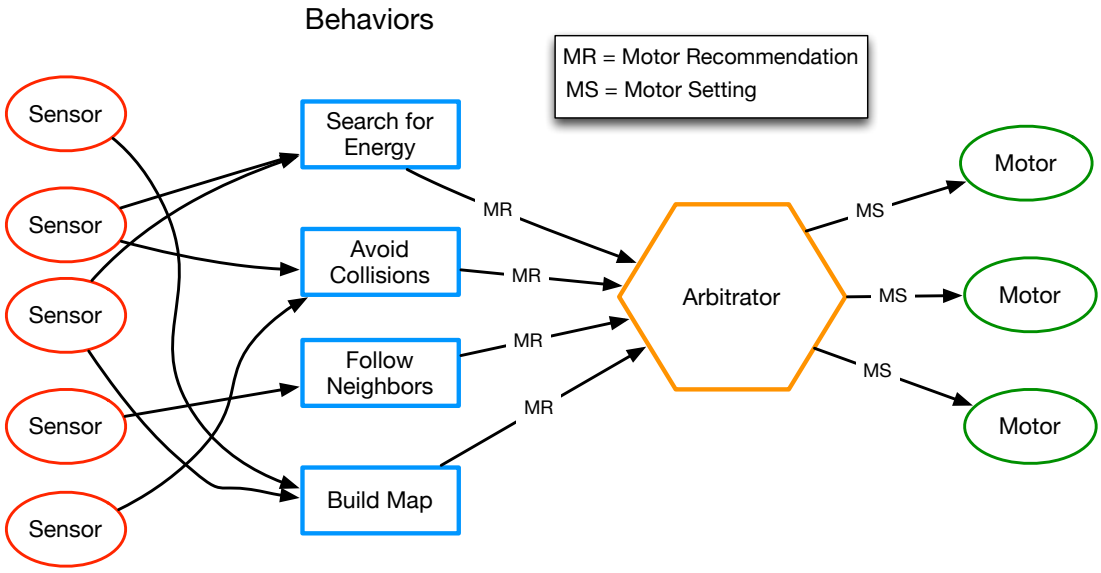

Figure 2: A typical example of behavior-based robotic control, wherein the core decision-making modules are behaviors, all of which have direct access to relevant sensor data and can formulate recommendations for motor/actuator settings. The arbitrator receives all of these recommendations and decides on the final motor recommendations based on various static and dynamic priorities among the behaviors. This figure serves only as a rough overview, while Figures 3 and 5 provide more precise details of pre- and post-arbitration, respectively.

What follows is a detailed description of a generic BBR controller, which you must implement for this project. EACH of the classes described below must be defined in your own code (though you are free to use different class names), and you may wish to include additional subclasses.

2.1 Class BBCON

The highest-level class, BBCON (Behavior-Based Controller) should only require one instance (per robot). At each timestep, the robot should call its bbcon to determine its next move. A bbcon should contain (at least) the following instance variables:

1. behaviors - a list of all the behavior objects used by the bbcon
2. active-behaviors - a list of all behaviors that are currently active.
3. sensobs - a list of all sensory objects used by the bbcon
4. motobs - a list of all motor objects used by the bbcon
5. arbitrator - the arbitrator object that will resolve actuator requests produced by the behaviors.

Other instance variables, such as the current timestep, the inactive behaviors, and the controlled agent/robot may also prove useful in your implementation.

The method set for BBCON should include the following simple procedures:

1. add_behavior - append a newly-created behavior onto the behaviors list.
2. add_sensob - append a newly-created sensob onto the sensobs list.
3. activate_behavior - add an existing behavior onto the active-behaviors list.
4. deactive_behavior - remove an existing behavior from the active_behaviors list.

In addition, BBCON must include a method named **run_one_timestep**, which constitutes the core BBCON activity. It should perform (at least) the following actions on each call:

1. Update all sensobs - These updates will involve querying the relevant sensors for their values, along with any pre-processing of those values (as described below)
2. Update all behaviors - These updates involve reading relevant sensob values and producing a motor recommendation.
3. Invoke the arbitrator by calling **arbitrator.choose_action**, which will choose a winning behavior and return that behavior's motor recommendations and *halt_request* flag.
4. Update the motobs based on these motor recommendations. The motobs will then update the settings of all motors.
5. Wait - This pause (in code execution) will allow the motor settings to remain active for a short period of time, e.g., one half second, thus producing activity in the robot, such as moving forward or turning.
6. Reset the sensobs - Each sensob may need to reset itself, or its associated sensor(s), in some way.

2.2 Class Sensob

A sensob serves as an interface between (one or more) sensors (of the agent) and the bbcon's behaviors. Serving as an overview image, Figure 2 portrays only sensors and behaviors, but Figure 3 provides a more detailed and accurate account. Note that a single sensor may be shared by several sensobs, and a sensob may employ several sensors.

As an example of sensor sharing among sensobs, assume that the sensor is a camera and the sensobs are different views or interpretations of the camera image: one sensob considers only the green component of each red-green-blue (RGB) pixel, while another looks only at the red, and a third does edge-detecting preprocessing such that its value (used by one or more behaviors) is a line drawing of the contours of objects.

As an example of multiple sensors aggregated into one sensobs, imagine a row of infrared sensors on the bottom of a robot (as is used by the Zumo robot for line following). The intensity readings from these K sensors may be bundled into a short array by a single sensob that functions as a line detector.[1]

As shown in Figure 3, the sensob houses the pre-processed sensory data that behaviors use to determine their actuator recommendations. Though not necessarily mentioned in brief descriptions of behavior-based robotics, this intermediate level (the sensob) serves a useful purpose by allowing behaviors to work with something other than the most primitive sensor data.

The main instance variables of a sensob are a) its associated sensor(s) and b) its value. So in the case of the line-detecting sensob mentioned above, its sensors are the infrared sensors on the robot's underside, while its value variable might actually house a pair of values such as (0, 2), indicating that the strongest indication of a line is from the sensor in position 0 to that in position 2. A line-following behavior could then easily convert that information into a recommendation to turn slightly to the left (assuming that the infrared sensors are labeled 0-5 from left to right).

The main method for a sensob is **update**, which should force the sensob to fetch the relevant sensor value(s) and convert them into the pre-processed sensob value. This should only need to be done once each timestep. So even if several behaviors share the same sensob, S, there should be no need for S to update more than once each timestep.

[1] In this project, the bottom sensors of the Zumo come pre-bundled into an infrared sensor array (as described below).

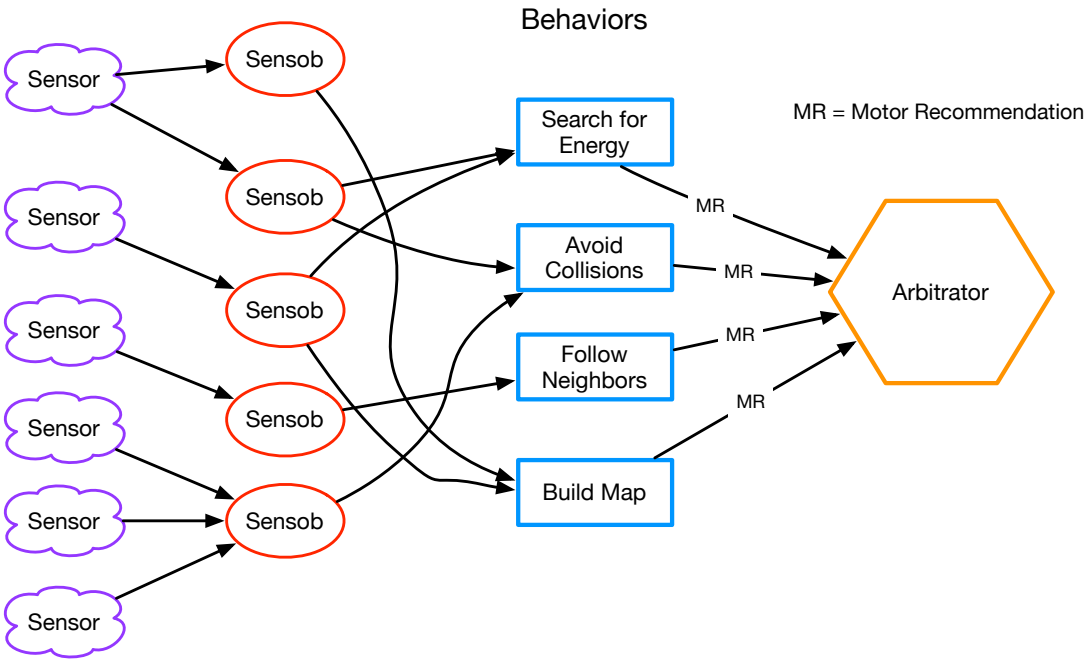

Figure 3: Overview of the basic relationships between sensors, sensobs, behaviors and the arbitrator in a behavior-based controller.

2.3 Sensors

Your Zumo robot comes equiped with several types of sensors, including a camera. We provide you with Python code to access these physical sensors, so you need not deal with those low-level details (of which there are many). This code is also object-oriented, with each physical sensor having a corresponding Python object, known as its **sensor wrapper** (SW). Figure 4 shows this relationship between a sensob and the actual physical sensor.

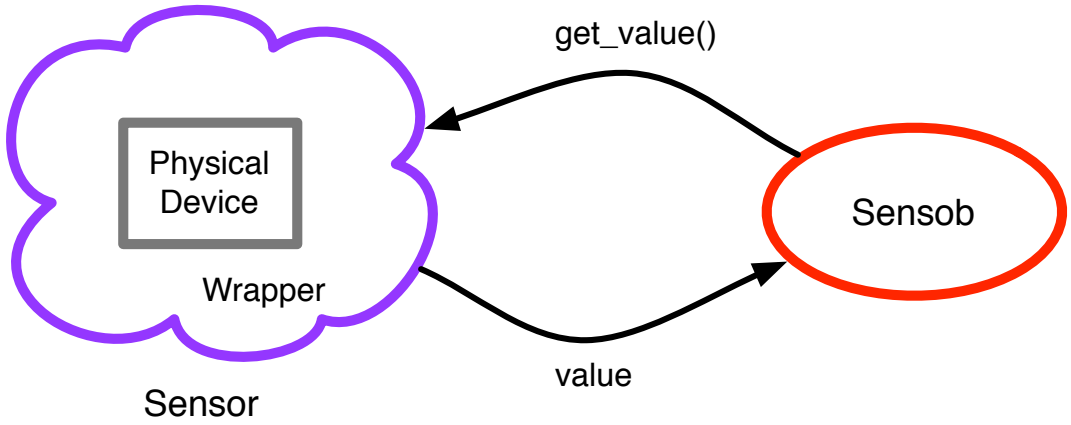

Figure 4: The key components of a sensor: a physical device and a Python code wrapper, which serves as the interface to the sensob.

You will write code for sensobs, and that code will call the SWs. Each SW provides a very simple interface for sensobs, with just a few (standard) methods such as **update** and **get_value**. Each SW does a small amount of preprocessing of the (completely) raw sensory data, but the information that each sensob receives (by calling a wrapper's get_value method) is also fairly basic, and open for many additional forms of preprocessing. For example, the camera SW returns a complete array of RGB pixels (encapsulated within an "Image" object), which your sensobs can manipulate using the full power of the Python Imaging Library (PIL, PILLOW).

The Zumo that you receive has 3 types of sensors: camera, infrared and ultrasonic. Each comes with a fully-coded sensor wrapper. You are free to explore other sensor types, and our PLAB engineers will supply you with those that we have in stock, but you will then have to write your own sensor wrappers for them. These vary in difficulty.

Below is a brief description of the four sensors and wrappers that accompany your Zumo. Note that the wrapper classes do not contain the word "wrapper". Rather, the names are simply those of the physical sensor types that they encapsulate. As far as your sensobs are concerned, they are just calling a sensor and getting a value back.

2.3.1 Ultrasonic Sensor

This consists of a sender-receiver pair, mounted vertically on the front of your Zumo. It is used to detect obstacles in the vicinity of the robot. The sensor wrapper causes the sensor to emit an ultrasonic pulse and then wait to receive a reflected pulse. Time differences between the sending and receiving are then used to calculate the distance to the nearest obstacle.

The wrapper class named **Ultrasonic** (in file ultrasonic.py) handles this type of sensor. Its three essential methods are:

- **update** - this invokes the sending-receiving actions, followed by calculation of the actual distance, which is placed in the wrapper's *value* slot and used as the method's return value.
- **get_value** - this simply returns that distance (in centimeters) as a floating-point value.
- **reset** - this sets the *value* slot to None.

This sensor is useful for behaviors such as collision avoidance, approach to a target object, and following of a moving object. It is surprisingly accurate for an inexpensive sensor.

2.3.2 Infrared (IR) Sensor Array

The bottom of the Zumo contains a row of 6 infrared (reflectance) sensors, which are primarily designed for line following and edge-detecting behaviors. These employ 6 capacitors that are initially charged to capacity and then lose charge as a function of the amount of ambient infrared light that is reflected from the underlying surface (e.g. floor or table top) onto the bottom of the robot.

These six sensors are packaged together by a single wrapper whose value is an array of 6 floating-point numbers between 0 and 1. These values indicate the amount of reflectance measured by each of the 6 sensors, with high values indicating a lighter (whiter) color, whereas low values signal darker colors.

Following lines then involves comparing the 6 array values to determine whether the robot is currently: centered on a line, off to the left or right, or not in contact at all with a line.

The wrapper class named **ReflectanceSensors** (in file reflectance_sensors.py) handles this type of sensor. Its three essential methods are:

- **update** - this loads all 6 capacitors and then gives them 5 milliseconds to decay. At that time, their voltage levels are scaled to produce reflectance values between 0 and 1. This also returns the array of values.
- **get_value** - returns the array of 6 real-valued reflectances.
- **reset** - set the value slot to an array containing 6 copies of -1.

Calibration of this sensor array is normally performed when your Python code first creates a ReflectanceSensors object. The whole point of calibration is for the wrapper to figure out (approximately) what the minimum and maximum reflectance values are in the current environment.

Without this information, the capacitor levels may be improperly scaled, thus making it difficult for your sensobs to correctly detect lines.

See the **__init__** method for ReflectanceSensors for the details of calibration. It only occurs when the *auto_calibrate* argument (to __init__) is True. Otherwise, the constructor uses min and max readings that you can send as additional arguments to that __init__ method.

During auto-calibration, the wrapper prints a message (e.g. "calibrating") to the screen. This indicates that the 5-second calibration process has begun. In those 5 seconds, you should quickly move the robot around in an area with some dark and some light surface coloring/shading so that the sensors experience realistic minimum and maximum reflectance values. To perform auto-calibration when the Zumo is not connected to a laptop (and thus has no screen for printing messages), you will need to remember that the process begins at approximately the same time as when the robot initializes your controller. So right after turning the robot on, start moving it around to light and dark surfaces for 5 or 10 seconds. Many types of sensors are extremely sensitive to the ambient conditions, so auto-calibration often proves more effective than simply assuming min and max values ahead of time. However, the default values for min (100) and max (1000) reflectance (used by the __init__ method) tend to work pretty well in a normal indoor lab settings.

2.3.3 Camera

The PLAB-2 version of Zumo includes a Raspicam (Raspberry Pi Camera) as its most information-rich sensor. This camera provides an adjustable resolution that can exceed 10,000 RGB pixels (e.g. 128 x 96).

The wrapper for the Raspicam is the class named **Camera** (in file camera.py). Its three essential methods are:

- **update** - This instructs the Raspicam to take a picture and store it in a file (image.png) on the Raspberry Pi. It then calls Image.open(image.png), which creates a PIL Image object and loads it with data from the file image.png. Finally, it sets the value slot of the Camera object to that Image object, which serves as the return value.
- **get_value** - returns the Image object, which is ready to be analyzed and modified using the wide range of PIL methods.
- **reset** - set the value slot to None.

The default resolution for Raspicam images is 128 (width) x 96 (height) pixels; but as shown in the file camera.py, this resolution can be modified via the arguments to Camera.__init__(). Taking high-resolution pictures can be a resource-draining operation for a small robot, so if your sensobs are only interested in low-resolution pictures, then modify the resolution parameters for the Camera object (as opposed to taking the high-resolution pictures but then resizing them in the sensob).

Furthermore, any bbcon behaviors that require camera data as input should be carefully implemented such that the behaviors become active only when necessary. For example, if your robot's task is to first follow a line and then (when, for example, the line runs into a wall) the robot needs to determine the color of the wall, there is no need to have the camera on (and snapping pictures) until line-following has ended. The camera and its resource demands are key motivations for the use of active and inactive states for behaviors and their sensobs.

To handle the images returned by the camera, you will need to understand the Python Imaging Library (PIL), which is explained in a separate document (pil-intro.pdf).

2.3.4 Zumo Button

Just to the right of the on/off switch on the back of the Zumo, there is a small button. The file zumo_button.py houses a very simple wrapper class (ZumoButton) for this button (i.e., touch sensor). The only method for a ZumoButton is wait_for_press, which simply puts the Zumo in a waiting loop until the button is pressed.

This button is useful for the start of demonstrations (and is used by the three main demos in robodemo.py), since you can send the command to start a demo to the robot from your keyboard and then disconnect the ethernet cable, relocate the robot to a desired location, and start the robot running when you are ready.

2.4 Class Motob

The motor object (motob) manifests an interface between a behavior and one or more motors (a.k.a. actuators). It contains (at least) the following instance variables:

1. motors - a list of the motors whose settings will be determined by the motob.
2. value - a holder of the most recent motor recommendation sent to the motob.

Its primary methods are:

1. update - receive a new motor recommendation, load it into the *value* slot, and operationalize it.
2. operationalize - convert a motor recommendation into one or more motor settings, which are sent to the corresponding motor(s).

Motobs allow behaviors to make motor recommendations at a relatively high level, such as (L, 30) (i.e., turn 30 degrees to the left), which can then be translated into lower-level motor settings for individual actuators, such as the wheel speeds and directions (i.e., forward or backward) of a two-wheeled robot. [2] In this case, a single motob would be associated with two motors, one for each wheel.

Figure 5 depicts the main connections along the output path from arbitrator to motors. For many simple robots, such as the Zumo, the standard arrangement consists of one motob and two (wheel) motors, with motor recommendations such as (L, 45) and motor settings such as +0.5: run the given wheel at 50% of maximum speed in the forward (+) direction for one timestep.

2.5 Motors

The Zumo only comes with two controllable moving parts: the two wheels, each of which is driven by a small physical motor. The class **Motors** (in file motors.py) provides a simple wrapper around both of these wheel motors. The basic relationship between physical parts, wrappers and motobs is shown in Figure 6.

The two essential methods for a Motors wrapper are:

- **set_value (v)** - sets speed and direction (forward or backward) of the left and right wheel motors via v, a vector of size two. Each of these two elements has a range of [-1, 1] with positive (negative) values indicating forward (backward) rotation of the corresponding wheel. The absolute value of each element determines the rotational velocity, with 1 indicating the maximum motor speed that the Zumo can handle.

[2] This is actually a difficult problem, since determining the motor speeds and durations needed to rotate exactly 30 degrees is extremely sensitive to ambient conditions, such as the friction between the floor and wheels. You will understand this issue once you begin experimenting with your robot.

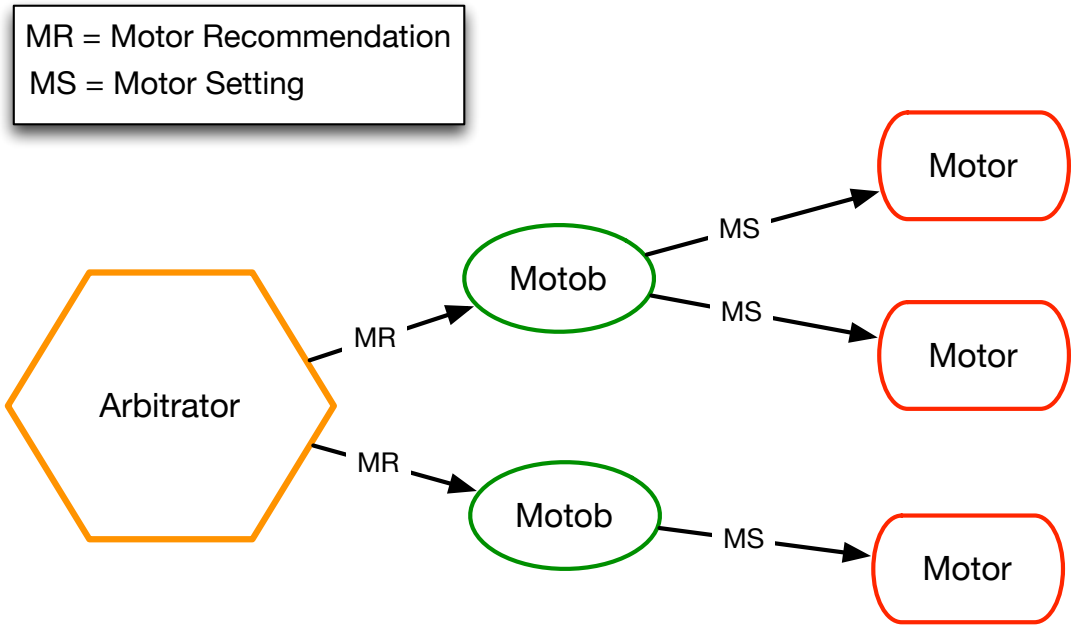

Figure 5: Overview of the basic relationships between the arbitrator, motor recommendations, motobs, motor settings and motors in a behavior-based controller.

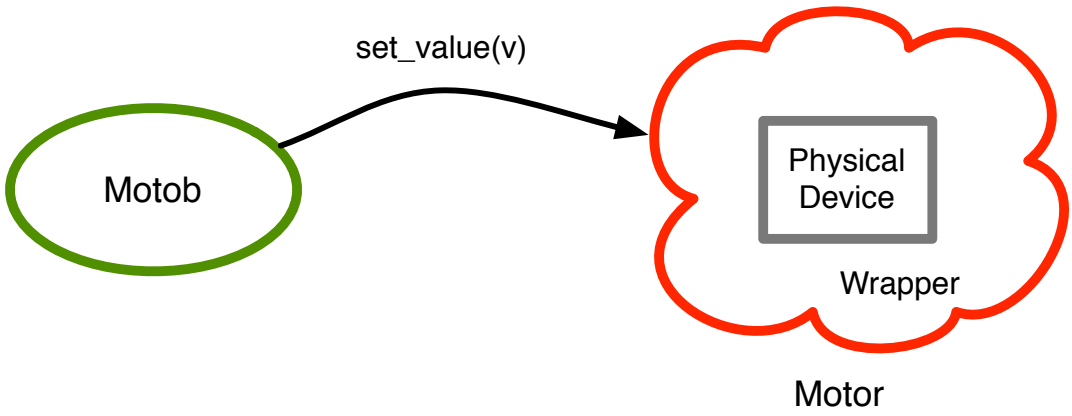

Figure 6: The key components of a motor: a physical device (or several devices) and a Python code wrapper, which serves as the interface for the motob, whose main action involves sending motor settings to the wrapper.

- **stop** - turns off both wheel motors.

Note that once wheel speeds are set, the wheels will continue to run at those speeds and directions until new settings (or a stop command) are given.

2.6 Class Behavior

The core of BBR are the behaviors themselves, each a modular unit designed to analyze a subset of the sensory information as the basis for determining a motor request. Behaviors operate in a vacuum in the sense that they have no knowledge of or direct connection to other behaviors.

It violates the fundamental principles of BBR to design behaviors that communicate directly with one another. All interaction occurs indirectly via either the arbitrator or via information posted by one behavior (in the bbcon) and read by a second behavior (from the bbcon) [3] **One important condition for receiving a passing mark on this project is that your group's code obey's this simple, yet extremely important, principle.**

The primary instance variables for a behavior object are the following:

1. bbcon - pointer to the controller that uses this behavior.
2. sensobs - a list of all sensobs that this behavior uses.
3. motor_recommendations - a list of recommendations, one per motob, that this behavior provides to the arbitrator. In this assignment, we assume that ALL motobs (and there will only be one or a small few) are used by all behaviors.
4. active_flag - boolean variable indicating that the behavior is currently active or inactive.
5. halt_request - some behaviors can request the robot to completely halt activity (and thus end the run).
6. priority - a static, pre-defined value indicating the importance of this behavior.
7. match_degree - a real number in the range [0, 1] indicating the degree to which current conditions warrant the performance of this behavior.
8. weight - the product of the priority and the match_degree, which the arbitrator uses as the basis for selecting the winning behavior for a timestep.

As a brief review of these variables, the pointer up to the bbcon allows each behavior to check the bbcon for any important posts (by other behaviors) in cases where limited interaction between behaviors occurs. This should **not** be a dominant factor in your implementation, as a behavior should base its motor recommendations primarily on sensobs, but occasionally it can save a lot of work if one behavior posts information to the bbcon that another behavior can read. The bbcon pointer also enables easy coordination in cases where a behavior activates or deactives (based on sensory input) and needs to inform the bbcon (in order to be added or removed from bbcon.active_behaviors).

As indicated by the behavior's active_flag, a behavior may shift between phases of activity and inactivity. Active behaviors are those that actually analyze their sensory information and MAKE a motor recommendation – though it may not be the MR chosen by the arbitrator. An inactive behavior essentially does not run on a timestep (except to check whether it can become active). This is explained in more detail below.

A behavior may make a halt_request based on its sensory input or other information, such as state variables of the behavior or information posted to the bbcon. If such a request is made, and this behavior wins during arbitration, then the agent's current run will halt completely. Ideally, only

[3] The bbcon thus serves as a *blackboard* for posting and reading information, an approach made popular by various Blackboard Control Architectures used in Artificial Intelligence.

a few behaviors will make halt requests, such as those that detect that the agent has reached its final goal, or those that keep track of total run time and declare that a run has exceeded its time limit.

The arbitrator bases its selection of the winning behavior (for a timestep) on the weight property of each behavior, where weight = match_degree x priority. The behavior priorities are determined by the user prior to the run, while the match degrees are calculated by every active behavior. The match degree indicates a combination of the urgency and appropriateness of performing the given behavior at the given time. The terminology stems from AI expert systems, in which each rule (similar to a behavior) has preconditions and an action, with the degree to which the preconditions match the current state of the world affecting the chances that the rule is chosen, and its action performed.

For example, a line-following behavior might give a high match degree to a sensory situation in which the line is still visible, but only barely (i.e., only one bottom sensor is detecting a line), since although the sensory information is rather weak (with only one hot sensor), the urgency is very high (since one more timestep spent moving away from the line could leave the agent lost, without any immediate sensory clues as to the line's location). Conversely, a collision-avoidance behavior would have a high match degree when an impending obstacle was very near, as indicated by the robot's ultraviolet distance sensors.

The main methods for a behavior are:

1. consider_deactivation - whenever a behavior is active, it should test whether it should deactivate.
2. consider_activation - whenever a behavior is inactive, it should test whether it should activate.
3. update - the main interface between the bbcon and the behavior (detailed below).
4. sense_and_act - the core computations performed by the behavior that use sensob readings to produce motor recommendations (and halt requests).

The call to **update** will initiate calls to these other methods, since an update will involve the following activities:

- Update the activity status - Each behavior will have its own tests for becoming active or inactive. Some behaviors may be active all of the time, so the tests are trivial, whereas other behaviors may be computationally expensive to run and thus the bbcon can spare resources if it shuts them off in cases where they are clearly not needed. For example, behaviors that require camera images, particularly images that are preprocessed by sensobs, are expensive. If all behaviors that use a particular camera-based sensob are inactive, then the expensive image-processing computations can be avoided for that period of inactivity. Hence, when a behavior becomes active or inactive, its sensobs should be informed of the status change so that they too may activate or deactivate. Of course, for sensobs that are used by two or more behaviors, some simple extra bookkeeping is required: if sensob S is used by both behaviors A and B, then S can only deactivate when both A and B are inactive.
- Call sense_and_act
- Update the behavior's weight - Use the match_degree (calculated by sense_and_act) multiplied by the behavior's user-defined priority.

The central computation of a behavior occurs in its **sense_and_act** method. The activity in this method will be highly specialized for each behavior but will typically involve gathering the values of its sensobs (and possibly checking for relevant posts on the bbcon). Using that information, the behavior will then determine motor recommendations, and possibly a halt request. It must also set the match_degree slot to a real value in the range [0, 1].

In some cases, a behavior may require instance variables that maintain some memory of previous states of the sensobs. For instance, a red-object tracking behavior might record the fraction of red in the previous camera image and then compare it to the fraction in the current image to determine the movement of the red object relative to the agent. Similarly, a line-following behavior may want to record a series of recent line readings to indicate whether the agent is successfully staying on the line or gradually losing touch with it.

In general, behaviors can perform many operations, but they MUST:

- consider activation or deactivation
- produce motor recommendations
- update the match_degree

and they MUST NOT communicate directly with other behaviors.

2.7 Class Arbitrator

The arbitrator is a fairly simple class that makes a very important decision at each timestep: which behavior *wins* and thus gets its motor recommendations transferred to the agent's motobs, which will then determine the overt action(s) of the agent.

The arbitrator may include an instance variable housing a pointer to the bbcon, such that the arbitrator can easily fetch all of the bbcon's active_behaviors. The primary method of the arbitrator is *choose_action*, which should check all of the active behaviors and pick a winner.

This choice can either be very simple: pick the behavior with the highest weight; or it can include an element of stochasticity. In this latter case, the arbitrator makes a random, but biased, choice among the behaviors, with bias stemming from the behavior weights. For example, assume that there are 3 active behaviors with weights in parentheses: B1(0.8), B4 (0.5), and B6 (0.7). To make a stochastic choice among these behaviors, use the weights to generate a range for each behavior:

B1: [0, 0.8)

B4: [0.8, 1.3)

B6: [1.3, 2.0)

Next, generate a random real number, R, between 0 and 2.0 (= 0.8 + 0.7 + 0.5). Whichever of the 3 ranges that R falls within, the corresponding behavior wins. Clearly, behaviors with higher weights have a larger chance (but are not guaranteed) of winning.

You are free to choose a deterministic or a stochastic solution, or to implement both and let the user decide by setting the value of a simple instance variable (e.g., named *stochastic*) in the arbitrator. Experience shows that many robotics problems are more easily solved with some element of stochasticity. [4]

Regardless of the selection strategy, choose_action should return a tuple containing:

1. motor recommendations (one per motob) to move the robot, and
2. a boolean indicating whether or not the run should be halted.

In the cases of the simple deterministic and the stochastic arbitration strategies, both of these values should come directly from the winning behavior.

[4] Another option is to combine the motor recommendations of several behaviors, possibly scaled by the weights, but this is a bit more difficult and risky. You are welcome to try it, but the results may not be too satisfying. For example, the robot may be one step from the goal (that one of its behaviors recognizes), but due to the combination with other motor recommendations, the final move may be tangential to or even opposite the goal.

2.8 Trial and Error

The class descriptions above give the mandatory general structure for this project. However, the detailed code for your behaviors, sensobs, and motobs will determine the success or failure of your system.

You will need to carefully consider the choices for the static priorities of behaviors, along with the logic behind the calculation of the behavior's match_degree at each timestep. Although behaviors cannot directly communicate with other behaviors, YOU the designer should carefully consider the potential interactions between the behaviors and set their priorities accordingly. For example, if one behavior involves moving toward a goal, and if the goal can be seen with the camera, then presumably there is no obstacle in the way (otherwise the camera's view of the goal would be blocked). Hence, the priority of this goal-directed behavior should be set high enough so that when it's match degree is also high (e.g. when the goal is in plain sight), no other behavior should be able to beat it out[5].

The whole process of designing a robot controller (behavior-based or otherwise) is rarely done in one pass. A LOT of trial and error is involved, so plan to spend several days playing with parameters once your basic code is debugged. There is no simple, single answer to the proper settings for these values, nor the proper collection of behaviors. Experiment until you find something that works.

3 Demonstration

To receive a passing mark for this project, your group must:

1. Build a bbcon (in object-oriented Python) to run your Zumo robot, which will perform some interesting, multi-step task.
2. Include the camera and two other types of sensors in the bbcon (e.g. infrared belly sensors and an ultraviolet light sensor) , and use each of these sensors in at least one aspect of the multi-step task. The Zumo Button does NOT count as one of the 2 additional sensors.
3. Show a working demonstration of your system in which EACH of your bbcon behaviors can be clearly seen to have some effect upon the robot's activity.

4 Appendix: Getting Started with Zumo-Pi

Your Zumo robot is now driven by a Raspberry Pi processor, which provides considerably more power and programming possibilities than the Arduino. It comes fully loaded with Python (versions 2 and 3) and all of the important packages for connecting Python to your robot's sensors and motors. You should be able to take these connections for granted and focus most of your effort on standard Python programming.

However, there are a few preliminary activities that you will need to perform:

1. Connect your robot to a power source, insert batteries and turn it on. Connect your robot to the internet via an ethernet cable.
2. In order to use SSH and SFTP (described below) you need to get your robot's IP address. When you are using the ethernet cables in the A4 rooms, you can simply use a DNS name instead of an actual IP address. The format is robotxx.idi.ntnu.no where xx is the number printed on the ethernet port on your robot. It is also usually the same as your group name,

[5] Of course, this simple example assumes that all obstacles are above ground. It does not account for holes or ravines that the robot would probably benefit from avoiding.

but check that they match. For example if your number on the ethernet port is 34 then you will use the DNS name robot34.idi.ntnu.no. **Note:** If your group number is 1-9, you need an additional 0 in front, for example robot02.idi.ntnu.no.

3. (Optional) If you want to connect to the robot at home, you need to connect your robot to a screen and keyboard. Then login with the username and password (described below) and type:

   hostname -I

   The first number in the output will be your robot's IP address.

From your laptop, you can access the robot using the 'ssh' command from a terminal window. Simply type:

ssh plab@<your R-Pi's IP address>

So with the IP address above, this becomes:

ssh plab@robot34.idi.ntnu.no

Here, use the default password: **piberryrasp**. To change the password, use the unix "passwd" command and follow the instructions.

It is important to note that you will now need to prefix calls to "python3" and "pip3" with "sudo". So you need to type "sudo python3" and "sudo pip3".

To transfer files back and forth between robot and laptop, there are several options, but many will require you to install additional software on the robot's Raspberry Pi. A simple approach involves using FTP (actually SFTP). The process is quite straightforward (and fully supported by your robot already):

1. Open a terminal window on your laptop and navigate to it's robot directory, e.g. mylaptop/robot.
2. From that directory, type "sftp plab@<your R-Pi's IP address>". Once you have entered the password, use UNIX commands such as **ls** and **pwd** to figure out where you are in the robot's file tree, and then use **cd** to navigate to the robot directory (e.g. /home/plab).
3. Once in the proper robot directory ('/home/plab/') on the Raspberry Pi, you have a direct connection between it and the corresponding directory on your laptop (e.g. 'mylaptop/robot'). So **put** and **get** commands to SFTP will transfer files from laptop to robot, and robot to laptop, respectively.

   For instance, to transfer my_controller.py from the laptop to the robot, type "put my_controller.py". And to transfer the "image.png" file from the robot to your laptop, type "get image.png".

When you have transfered all relevant files to the robot, navigate to /home/plab and type "sudo python3" on the command line. Now you can enter Python commands at the terminal window, just like in a PyCharm terminal window. To run any of the demos in the file robodemo.py, such as **dancer**, simply type:

```
>>> from robodemo import *
>>>  dancer()
```