COS 120 – Computing that Matters

Project 1: Build a robotic insect

Spring 2017

For this project, you will write a program to make your robot behave like an insect, such as a cockroach, in a rather simplistic fashion. Your robot will:

- Explore an area by wandering around.
- Investigate interesting things it "sees" using its ultrasonic sensor.
- Turn and run from an approaching object.

The program that you write will be a simple *behavior-based controller*. You will write several *behaviors*, implemented as functions that will be called by the controller. Different behaviors will be responsible for different aspects of how the robot responds to its environment: wandering, avoiding an object that is approaching, etc.

Doing exercises 1–9 will prepare you for doing this project; in fact, some of the behaviors may be usable for the project with little or no change. The behaviors that your project should have are:

1. Avoid obstacle: when moving toward an object, stop, then turn and go around it
2. Wander: the robot will wander about the area
3. Investigate: when a stationary object is within some distance d, and closer than the background, then go toward it.
4. When a stationary object is in front of the robot at some other distance $d_1 < d$, report it to the user.
5. Turn and run: if the robot perceives something coming toward it, then it should turn and run for some distance in a direction that takes it away from the object.

Each of these will have access to the robot's *percepts*—its sensor data—and each will independently produce a description of the actions it thinks the robot should take in response. Both the percept format and the command format are described in Exercise 9, along with how to interface your controller to the robot using a robot interface that will be provided.

Your controller will call each of the robot's behaviors, collecting their action recommendations. It will then use a *subsumption* controller to decide which behavior's recommended actions are actually taken by the robot. This has the property that more important (in the current context) behaviors are favored over less important ones.

1 Background

- You should read the original journal article describing the subsumption architecture: R.A. Brooks, "A robust layered control system for a mobile robot," *IEEE Journal on Robotics and Autonomous Systems*, 1986. This is available on the module website.
- The original architecture has been used, modified, and extended to everything from the Roomba autonomous vacuum cleaners to autonomous underwater vehicle control (see, e.g., MIT's Ocean Odyssey AUV).

- The original architecture relied on a stack or layers of behaviors, such as shown in Figure 1. Each layer receives all input from the sensors, and each output what it wants the "actuators"—e.g., wheel motors, etc.—to do.

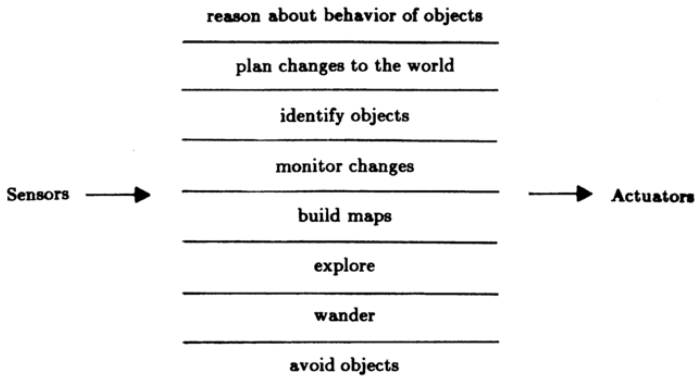

Figure 1: The original subsumption architecture. From Brooks [1986].

- The controller was very simple: when a higher-level wanted to control an actuator, it overrode, or *subsumed* anything any lower behavior wanted to do.
- Unfortunately, though very successful, there are some problems with this. For example, the "wander" behavior either (1) had to have its own obstacle avoidance built-in, since it overrode the "avoid objects" behavior; or (2) it had to output commands that weren't directly in competition with the "avoid objects" behavior so that the lower-level behavior would still be able to avoid objects. The first breaks the idea of separation of concerns, while the second violates the idea that the behaviors should be independent.
- This has led to different variations on the subsumption controller, including such things as activating/deactivating behaviors based the context, creating more complex control schemes than simple subsumption, and building the behaviors upside down so that, e.g., "avoid objects" would subsume "wander" to keep the vehicle from colliding with objects.

2 Requirements

In this project, you will implement two versions of a subsumption controller and try ordering your behaviors in two different ways for each. You will:

- Refactor your existing behaviors (from exercises 1–9) to use the robot object you created in the Exercise 9.
- Write a new behavior, `wander`, that will move the robot around its environment in a random manner.
- Write a new behavior, `run_away` (or refactor an existing one), that will turn and run away from an approaching object (rather than backing away, as in the exercises).
- Write a new behavior, `report_object`, that will tell the user when it has seen an object.
- Write a version of the subsumption controller that strictly overrides lower-level behaviors with higher-level ones; we'll refer to this as the *pure subsumption controller*.
- Write a version of the subsumption controller that blends the outputs of the behaviors, with a high-level behavior only overriding a lower-level one when their commands conflict—for example, if one wants to turn left and another wants to turn right; we'll call this the *blended subsumption controller*.

- Combine the behaviors and each controller to actually control the robot and test your behaviors.
  - For each controller, first order the behaviors with the obstacle avoidance behaviors at the "bottom" of the subsumption hierarchy
  - Now run the tests again, this time with the behaviors in the reverse order.
- You will demonstrate your robot's behavior in class, and you will submit a short report about the project and a video of your robot in action. Late work will be penalized up to 1/2 a letter grade/day.

3 Programming notes

- You will use the `robot.py` module introduced in Exercise 9 and any new classes you defined then.

3.1 Refactoring your existing behaviors

- Refactor each of your existing behaviors to work in the context of a behavior-based controller, which will call each behavior in the same manner and expect the kind of return value from each
- Each of your behaviors should take a single parameter, `percept`, which contain a Python dictionary holding the robot's percepts, as discussed in the previous exercise.
- Each should return a single Python dictionary that specifies the command(s) it wants the robot to do, as discussed in the previous exercise.

3.2 The `wander()` behavior

- Assume that when your robot has nothing better to do, it just wanders about the world.
- Write a `wander(percept)` behavior to do this.
- Speed, direction, etc,, can be random and should change over time.
- The robot should not hit objects, but you don't have to worry too much about that for your `wander()` behavior, since this is the purview of the avoidance behavior.
- Your behavior will accept as input a percept dictionary (see the previous exercise) from the robot interface and will return a command dictionary representing what it wants the robot to do.

3.3 The `run_away()` behavior

- When the robot encounters something moving toward it, it should turn and run away.
- The direction it runs is up to you:
  - It could be random (using the `random` package, in particular, the `randint()` function).
  - It could always run the same way.
  - It could turn until it sees no obstacle within some predefined distance, then go that way.
- The distance it runs is also up to you.

3.4 The `report_object()` behavior

- This behavior will only become active when an object is within some set distance of the front of the robot *and* the robot is stopped.
- It should report the location of the object relative to the user by printing to the Python console (e.g., with `print()`; e.g.,:

```
Sun Dec  4 16:27:23 2016: Object detected 11 cm at bearing 20 degrees.
```

- You will need to import `asctime()` from the `time` module.
- The bearing is just where the sonar is pointing.

3.5 The subsumption controller: Two versions

- Write a controller that calls all the behaviors and determines, and that determines which command(s) to try to carry out.
- The commands are then executed by calling either the robot's basic commands or the interface functions you created in a previous exercise.
- In the best case, different behaviors will all request mutually-compatible commands, for example, behavior 1 wants to go forward and behavior 2 wants to turn; in this case, we can merge the two command requests into one
- Most of the time, however, desired commands from different behaviors will conflict. This is where the controller comes in.
- A controller will take all the command requests from all the behavior commands and decide which command (or blended command) to actually give to the robot.

3.5.1 Simple controller

- First, build a very simple subsumption controller that gives complete control to the behavior whose output it selects.
- This controller doesn't need to worry about *what* the behaviors are, just the *order* in which it considers their outputs.
- You do not want to "hard code" your behaviors into the controller: that is, you don't want to have something in your controller that looks like:

```
# call the behaviors:
action = stop()
if avoid is not None:
   send_command(action)
else:
   action = avoid()
   if action is not None:
      send_command(action)
   else:
```

  If you were to do that, then not only does your code become rather ugly, but adding new behaviors is difficult.
- Instead, define a variable, say, `behaviors`, to contain the behaviors:

- It should be in the order you want the controller to call the behaviors.
- There are several ways you can populate this list, including:
  - Build it as a list of the actual functions, not their names:
    - E.g.,
      `behaviors = [stop, avoid, wander]`
    - Call behaviors in this list as:
      `behaviors[0](percept)`
      e.g., to call `stop()`
  - Build it as a list of the names of the behavior functions:
    - E.g.,
      `behaviors = ['stop', 'avoid', 'wander']`
    - Call these behaviors with something like:
      `eval(behaviors[0] + "(percept)")`
      e.g., to call `stop()`
- The simple controller can just loop through this list, stopping after the first behavior that returns a value that is not `None` (since it is not blending any actions), using, e.g.:

  ```
  for behavior in behaviors:
  ```

3.5.2 Blending controller

- Now, copy and modify the controller you just implemented so that instead of one behavior totally subsuming another (e.g., `avoid()` having complete control if it and `wander()` both output something other than `None`, or vice versa), it instead *blends* commands together where possible.
- First, decide what you are going to allow to be blended:
  - For example, it's unlikely that if one behavior says go forward and another says go backward that your controller should try to blend those.
  - However, if one behavior suggests going forward and another suggests turning left, or if one says go forward and another suggests turning the servo, then those *could* be blended.
- Now, either your controller will call *all* behavior functions and:
  - make a new list, say `behavior_commands`, of all return values that are not `None`, then go through that list and blend commands; or
  - build a blended command as it goes.
- In either case, you may want to start with a "blank" command that is a dictionary containing all possible command names as keys, each with `None` as the value.
- Then, for each command dictionary returned by a behavior function, your program would:
  - Loop for each command name in the command dictionary, e.g.:

    ```
    for command_name in behavior_command:
       if command[command_name] is None:
          command[command_name] = behavior_command[command_name]
    ```

    where `command` is the "blank" command your program created.
- When all behaviors are processed, then `command` contains the actual command (or set of commands) to be carried out.

4 Testing

1. Turn on the GoPiGo and place it on the floor.
2. Start Python and load your file(s).
3. Put some objects on the floor.
4. Run the simple controller with behaviors arranged so that the avoidance behaviors are able to subsume other behaviors' output if needed.
   (a) Let the robot run for a while, observing its behavior.
   (b) Note if the robot seems to have trouble navigating around objects.
   (c) Note if the robot runs into objects.
   (d) Note if the robot seems to be accurate when it scans objects.
   (e) In general, is it a convincing simulation of an insect?
5. Run the blending controller in the same environment with the same behavior order, noting the same kinds of things as for the simple controller.
6. Now do the tests again with each controller, but this time with the order of the behaviors reversed—i.e., with the behaviors ordered as in the original subsumption architecture.

5 Questions for thought

Address these questions in your project write-up:

1. Which kind of controller worked better? Why do you think that was the case? If it was the blended controller that worked better, do you think the extra programming effort needed to built it was worth it?
2. Which ordering of the behaviors worked better? Why do you suppose that was the case?
3. Can you think of anything you would do differently to improve the behaviors you have? Did you notice anything that would make you consider refactoring them again?
4. Did your robot ever stop and need intervention (i.e., moving an object, moving the robot) to begin moving again? If so, why?
5. Can you think of anything you would do differently to improve the controller?
6. What insights, if any, did your experience with this project give you about how insects might *actually* behave?
7. What are your thoughts about this kind of robot control (i.e. behavior-based control)?

6 Evaluation rubric

Grade	Criterion
A	Robot performs all behaviors satisfactorily and subsumption controller works correctly.
B	Robot performs most behaviors satisfactorily and subsumption controller works correctly.
C	Robot performs most behaviors satisfactorily and subsumption controller works mostly correctly.
D	Robot performs some behaviors satisfactorily and subsumption controller works somewhat correctly.
F	Robot does not perform any behaviors satisfactorily and/or subsumption controller does not work, or project was only partially completed or not turned in.

7 Stretch goals

- Add other behaviors and test them. You could look ahead to the next exercises, for example, and use the camera to take pictures of the objects found.
- The subsumption architecture is predicated on not keeping any *state* information around—that is, no memory. However, this has limitations, one of which is that sequences of actions cannot be carried out by a behavior without having the other behaviors have to wait. (Why? Can you explain?) Think about how you would add a behavior that has memory to the controller (which you will do in Project 2). Then, *without looking at Project 2*:
  - Add another behavior or modify `report_object()` so that when an object is detected, the robot scans the object with its sonar by moving it from one side to the other, getting sonar information at regular intervals.
  - This behavior will need to keep track of what is doing from one call to it to another.
  - The behavior should print an estimate of the size of the object based on the scan.
  - Integrate this behavior into your controllers and test it.