Introduction

Welcome! This packet gives a detailed sample curriculum for teaching introductory programming and robotics using the AERobot.

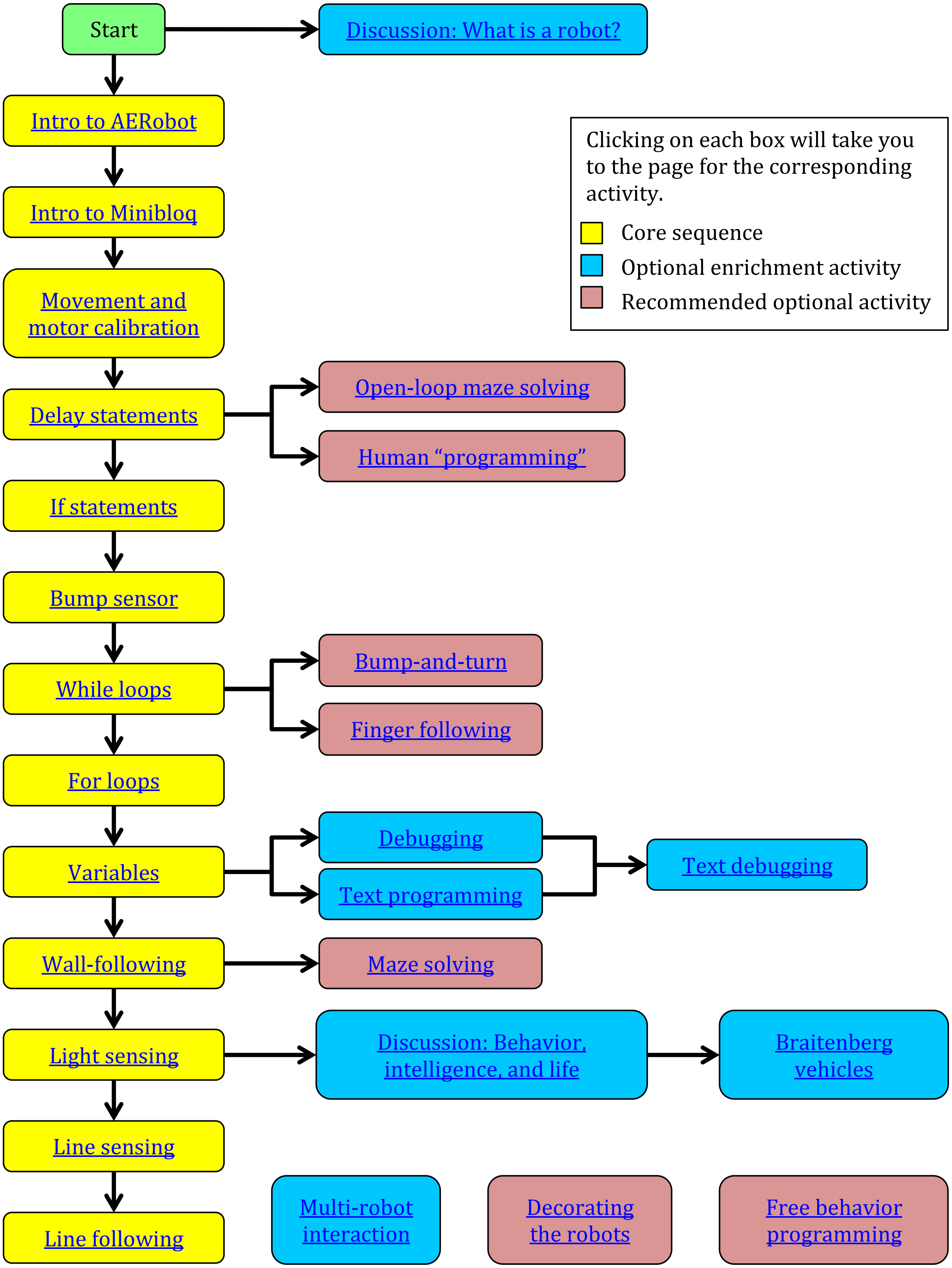
Before you begin, read about the robot on its website. Get an overview of the hardware and software. Be aware of technical issues you may encounter with the robot, and in particular be sure to understand calibrating it.

The next page gives an overview of the activities in the curriculum, showing a suggested core sequence and optional enrichment activities. The overview links to detailed descriptions in the rest of the document. The activities can be reordered, excerpted, or adapted according to class needs.

Materials and setting: Students will need access to computers—ideally one per student, but sharing can work—as well as space for their robots to operate (on a table or floor). Because the robot's sensors can be affected by bright light, especially sunlight, it's best if the room allows closing shades on any windows and lowering artificial lights. With larger groups, standalone multi-port USB chargers are useful for charging several robots at once between activities. Some activities call for additional materials, as noted on their own pages.

A course based on this material was popular and effective in pilot sessions with rising fifth through eighth graders in STEM summer camps run by i2 Camp, which supported this initiative.

This material was developed by Justin Werfel at the Wyss Institute at Harvard, with contributions from Mike Rubenstein, Bo Cimino, Julián da Silva Gillig, and Jerry Shaw, and is distributed under a Creative Commons Attribution-NonCommercial (CC BY-NC 4.0) license.



Discussion: What is a robot?

Goal:	Explore the idea of what makes something a robot, classic and unusual examples of robots, and boundaries of robot-ness
Prerequisites:	None
Optional materials:	Whiteboard or similar (for recording notes on the discussion); computer, LCD projector, and internet connection (for showing videos of robots)
Estimated time:	30 minutes

Ask students to brainstorm about what the important characteristics are that make something a robot. Make a list of these features. Talk about a few examples of robots—maybe ask the students what their favorite robots are—and how they compare to this list. Talk about what you would get if each feature from the list were missing, and give an example of such a machine (some of which are still legitimately called robots; there are gray areas). Pictures or videos of example robots could be useful (see this suggested list).

Our list of key features for something to be a robot, and examples for things lacking each feature, is:

- *Embodiment*—it must exist physically in the real world. Software agents, like the Google webcrawler that automatically visits web pages to help build the search index, are sometimes called "robots" despite not being embodied.
- *Actuation*—it must be able to move in some way on its own, or otherwise have an impact on the physical world. A computer, on its own, isn't a robot.
- *Sensing*—it must be able to get some kind of input from the world to decide how to act. Industrial factory robots actually often lack this feature—they do exactly the same thing over and over again blindly, not actually reacting to what's in front of them.
- *Autonomy*—it needs to be able to take the information from its sensing, and decide what to do with it in choosing how to act, on its own. Some robots also lack this feature—e.g., the iRobot military PackBot is not autonomous, it's controlled directly by a human. Such robots are called *teleoperated*.

These exact terms aren't important if students come up with other ways of describing the same ideas, but at minimum those four concepts should be discussed. From this perspective, a Roomba is a great example of a robot, and a good one for them to think about—both for its familiarity and so that when the Aerobot is introduced they're thinking along the lines of Roomba rather than Optimus Prime.

Some features may be suggested but not be important characteristics for something to be considered a robot, e.g.:

- *Humanoid form*—R2-D2 is just as much a robot as is C-3PO.
- *Stiff movement*—many modern robots are built to move very fluidly. Videos of robots built by Boston Dynamics, like BigDog and Petman, are especially good examples.

Introduction to AERobot

Goal:	Learn about AERobot and its sensors and actuators
Prerequisites:	None
Materials:	AERobots; hardware handout
Estimated time:	30 minutes

Give each student an AERobot kit and a hardware handout.

> *Note:* With younger students, it can be helpful to insert the battery before giving the robot to the student.

Have the students assemble their robot kits.

Using the handout as a reference, point out the robot's features and how each is like something else familiar:

- *Movement*: the robot moves using two cell-phone vibration motors. This is like the way a cell phone set to vibrate can move across a table; having two motors, and turning on one or both at a time, lets the robot move straight or turn.
- *Bump sensors*: these are infrared emitters (like a television remote control) and receivers, that can tell the distance to an object according to the strength of the infrared signal when it bounces back.
- *Light sensors*: these can be used to measure the brightness of light in the room. One example of a light sensor in everyday life that may be familiar to students is the one on a laptop that adjusts the brightness of the display according to how dark it is in the room.
- *Light-emitting diode*: can turn on in many different colors. These should be familiar from most electronic devices (though the multi-color aspect may be new to students).

If the class has had the discussion about what makes something a robot (previous page), go through the list of robot characteristics the class decided on and talk about how the AERobot satisfies each one.

Introduction to Minibloq

Goal: Become familiar with the Minibloq interface and using it to program the robot (turning on its LED)
Prerequisites: Introduction to AERobot
Materials: Computers with Minibloq software installed; (recommended) teacher's computer with LCD projector to demonstrate
Estimated time: 30 minutes

Show the students the process of (1) creating a Minibloq program that turns on the robot's LED, (2) connecting the robot to the computer, (3) downloading the program to the robot, (4) unplugging the robot and running the program. Show how the LED lights up, and how the program can be changed to light up a different color.

★ Minibloq has a sample program for this activity. Go to the File menu and choose Examples; then, in the window that opens, double-click on the AERobot folder, then on 10.ledOn, and finally on ledOn.mbqc. ★

Give students computers running Minibloq and have them write their own programs to do the same thing with their own robots.

An optional activity, to give students who've gotten this working a further challenge while you help other students who are still having trouble with the initial steps, is to introduce the delay block and have them make their robot blink a message in Morse code. This can be extended by having the students decode each other's messages.

★ Minibloq has a sample program using the LED and delay blocks. Go to the File menu and choose Examples; then, in the window that opens, double-click on the AERobot folder, then on 20.delay, and finally on delay.mbqc. ★

Note: If the class is going to do the text-based programming activity later in the lesson sequence, it's helpful to have the panel open in Minibloq showing the C code corresponding to the blocks, so they get used to it from the start. If the class is going to stick with graphical programming only, it can be simpler to keep that panel hidden.

Movement and motor calibration

Goal:	Learn about making the robot move, and calibrating its movement
Prerequisites:	Introduction to Minibloq
Materials:	Computers, robots
Estimated time:	30 minutes

Introduce the motor control block in Minibloq.

Have students write a program to move straight ahead.

Observe that the robot is (probably) not moving very effectively straight ahead, and that any given pair of robots (and even the same robot over time) doesn't move in the same way.

Introduce the idea of calibrating the motors (setting the speed at which they spin in order to get the result you want for how the robot moves), demonstrate the calibration routine, and have students perform it.

Repeat for the other movement directions (backward, turn left, turn right).

Note 1: Why don't the robots come pre-calibrated? Because the speeds the motors turn at to get the most effective movement will be different for each robot, and for each different surface they move on. You may be able to demonstrate this by taking a robot calibrated to move on a tabletop, and showing that it doesn't work as well on another surface (like the floor or a textbook).

Note 2: The variability in how the robot moves highlights the importance of using feedback ("closed-loop" control—using sensing to help decide on action—as opposed to "open-loop" control, when the robot just does something completely preprogrammed). For instance, the robot might have trouble going straight on its own, but if there's a wall or a line on the ground to use as a reference, it can follow that.

Delay statements

Goal:	Learn about the delay block
Prerequisites:	Introduction to Minibloq
Materials:	Computers, robots
Estimated time:	10 minutes

Introduce the delay block in Minibloq.

Have students program their robots to move forward for a certain time, then stop.

★ Minibloq has a sample program for this activity. Go to the File menu and choose Examples; then, in the window that opens, double-click on the AERobot folder, then on 30.moveForward, and finally on moveForward.mbqc. (The next example, 40.basicMovements, extends this to include all the basic movement commands.) ★

See if they can find the right length of time to wait to get their robot to move a given distance. Next, the right length of time to wait to get a turning robot to turn ninety degrees.

Have them program their robots to move in a square, using forward and turn movement blocks, and the delay block to control the length of the side and the angle of the turn.

Open-loop maze solving

Goal:	Program the robot to move through a simple maze
Prerequisites:	Movement and motor calibration, Delay statements
Materials:	Maze construction elements (can draw a maze on paper with markers, or build a physical one out of cardboard or small free-standing objects); computers, robots
Estimated time:	1 hour

Have students create a maze for their robot (or for a partner's!). It should be simple: not many turns, wide passages, not too long (so it doesn't take the slow-moving robot forever to get through it).

Have students program their robot to move through the maze. Doing this without feedback from the sensors will be hard and require a lot of trial and error.

Note: The difficulty in getting the robot to get through the maze successfully will vividly illustrate the importance of using feedback ("closed-loop" control—using sensing to help decide on action—as opposed to "open-loop" control, when the robot just does something completely preprogrammed). Returning to the maze-solving activity later, when the students have learned how to use the robot's sensors to help it figure out for itself what to do, will be much easier.

Human “programming”

Goal:	Students “program” each other to move through a human-sized maze, getting a more direct sense of the challenges for the robot
Prerequisites:	Movement and motor calibration, Delay statements
Materials:	Open space; blindfold; masking tape
Estimated time:	45 minutes

Have students create a life-size (but simple) maze by putting masking tape “walls” down on the floor. (If space where tape can be used isn’t available, another possibility is to have some students act as the maze by holding out their arms. But the student who’ll act as the robot may be able to tell when they’re near someone, even if blindfolded, so that may help them get through the maze more easily than the robot could.)

It’s best if the students who’ll be acting as the robot don’t get to see the maze ahead of time, so they don’t know in advance how they should move.

Pick a student to act as the robot. Have the other students write a “program” for them to follow—the same way they would for the robot, with commands for movement (start moving forward, start turning in a direction, stop) and delays.

Pick a student to act as the “program”. Blindfold the “robot” and put them at the start of the maze. Have the “program” face away from the maze (so they can’t see the robot’s progress to help influence what they’re saying) and tell the “robot” what to do according to the list of instructions the class came up with. (The “program” can use a clock or watch to get the length of delay commands about right.)

Have the class edit the program and reset the “robot” to the starting position until they’re able to get through the maze successfully.

Then try running the same program with a different student as the “robot”. It probably won’t work on the first try because they’ll probably move at a different speed from the first “robot”—they weren’t “calibrated” the same way!

If statements

Goal:	Learn about the idea and use of conditional statements
Prerequisites:	Delay statements
Materials:	Computers, robots
Estimated time:	30 minutes

The actions the students have had their robots perform up until now have been completely predetermined (open-loop), with no ability for the robots to respond to the world around them. This activity is the first step in giving the robots more flexibility, letting them react in different ways to different situations.

Introduce the idea of *conditional* behavior—*if* a certain thing happens, *then* do something in response.

Show students how to use the if block in Minibloq. Set the condition to be something obviously true or false (e.g., “if 1=1”, “if 1=5”, “if 3>2”), and the result to be turning the LED on (so they can see that it goes on if the condition is true, or doesn’t if it’s false).

Show how the logical operators “and” and “or” can be used to combine two tests. (The logic of “and” and “or” works the same way as the familiar English meaning, so it should be straightforward to explain and understand.)

Note: These examples are conditional but don’t actually involve reacting to the world. Next we’ll introduce a sensor and use that, together with if statements, to create a *reactive* behavior.

Bump sensors

Goal:	Learn about the idea and use of the robot's bump sensors
Prerequisites:	If statements
Materials:	Computers, robots; hardware handout
Estimated time:	30 minutes

With the handout (or with an LCD projector and the diagram on the Hardware page online), point out AERobot's three pairs of infrared transmitters and receivers along its front edge. When a program turns them on, the transmitters send out infrared light. When this light reflects off an object and returns to the robot, a receiver can look at how bright the reflected light is to determine the distance to the object. The closer the object, the less light is lost and the more light is reflected back.

The bump-sensor block in Minibloq returns True or False according to whether the light measured by the receiver (whichever one the program specifies—left, right, or center) is bright enough to indicate an object very close by. There's a threshold: True if the light is brighter than that threshold, False if it's dimmer.

Note 1: The brightness of the light in the room (again, especially sunlight) will affect how much light the receivers sense, so that the distance at which a bump sensor registers the presence of an object can change accordingly.

Have students write a program using the bump sensor block. For instance, the robot might light up its LED in one color if there's an obstacle near one sensor when it's turned on, and in another color if there isn't.

Next, use an "or" block to turn the LED one color if there's an obstacle near any of the bump sensors, and another color if there's no obstacle near any of them.

Note 2: The while block, covered on the next page, will let students write programs that run continuously instead of once. (For instance, without a while command, the program might have the robot turn on the light in one color or the other according to whether there's an object there when it's turned on, and stay that way regardless of whether the object is moved later. With a while command, it can switch the light back and forth between different colors as the object is taken away and brought back, without needing to run the program more than once.)

Note 3: There's another block that treats these transmitter-receiver pairs as distance sensors. Rather than returning True or False according to whether the measured light is brighter than a threshold, it returns the numerical value of the brightness directly. (Because it's measuring the brightness of the reflected light, it does the opposite of what you'd intuitively expect distance sensors to do—closer objects give a higher value, not a lower one.) If some students finish this activity early while others are still working, an advanced activity can be to have them use the distance sensor in a program to make the robot do something more complicated and interesting (e.g., turn the LED green when there are no objects nearby, yellow when an object is moderately close, red when an object is very close).

While loops

Goal:	Learn about while loops
Prerequisites:	If statements
Materials:	Computers, robots
Estimated time:	20 minutes

Introduce the while block in Minibloq.

The simplest use is for infinite loops. At first, a program was limited to being a linear sequence of commands—do each of these things in turn, and then stop. The if block introduced the possibility of *conditional* behavior, meaning that different sections of code could be run (or ignored) depending on what happened—but program flow was still in one direction, from start to finish, stopping at the end. Now a "while true" loop lets a section of code be repeated forever.

Have students use a "while true" loop to make the robot flash a sequence of LED colors indefinitely.

★ Minibloq has a sample program for this activity. Go to the File menu and choose Examples; then, in the window that opens, double-click on the AERobot folder, then on 50.blinkColors, and finally on blinkColors.mbqc. (The next example, 60.mixingColors, shows how to set the LED color to anything the programmer wants, not limited to the basic nine choices.) ★

Next, use a "while true" loop with an if statement inside it, to have the robot turn its LED one color whenever there's an obstacle near one of its sensors, and another color whenever there isn't. (This is like the program they wrote in the activity on bump sensors; but in that case, they had to run the program from the start every time they wanted the robot to check for objects nearby, while this one updates automatically.)

★ Minibloq has a sample program for this activity. Go to the File menu and choose Examples; then, in the window that opens, double-click on the AERobot folder, then on 80.bumpCenter, and finally on bumpCenter.mbqc. (The next example, 90.bumpAndColors, has a more complicated version that turns the LED a different color for objects near each of the three bump sensors. The one after that, 100.bumpThreshold, shows how to use the distance-sensor block to get a more fine-grained result from these sensors than the bump-sensor block gives—see Note 3 on the previous page.) ★

Finally, write a program that uses a condition for the while loop that's more complicated than just "true". For instance, while the bump sensors detect no obstacle, turn the LED red and then blue. Such a program will keep the light flashing between those two colors until an object gets close, and then the light will stop changing color.

Bump-and-turn

Goal:	Write a program to make the robot move and avoid obstacles
Prerequisites:	While statements
Materials:	Computers, robots
Estimated time:	1 hour

Now students have all the pieces they need to program the robots to move forward and turn away from any obstacles they run into. The bump sensor tells them when they've run into something; the if statement lets them do different things according to whether or not they sense an obstacle; and the "while true" loop lets them keep doing that forever.

Students who get this program working quickly can develop it into more elaborate forms. For instance, if an obstacle sensed on the right, turn left; if an obstacle is sensed on the left, turn right; if an obstacle is sensed in front, stop (or turn in either direction, however they want their robot to behave).

Students can also play with their robots by giving them this program and then steering them around the table, using a finger as an obstacle to make the robot turn where and when they want.

★ Minibloq has a sample program for this activity. Go to the File menu and choose Examples; then, in the window that opens, double-click on the AERobot folder, then on 160.wallFollow1, and finally on wallFollow1.mbqc. (This example, and the next one (170.wallFollow2), use the distance-sensor form of the programming block, rather than the bump-sensor form—see Note 3 on the bump sensor activity page.) ★

Finger following

Goal:	Write a program to make the robot follow a finger around the table
Prerequisites:	While statements
Materials:	Computers, robots
Estimated time:	45 minutes

The bump-and-turn activity had the robot trying to avoid obstacles, so that students could steer it around using their finger as an obstacle. Turning that behavior on its head lets them steer the robot using their finger to attract rather than repel it.

Have students write a program—or, if they've just written the bump-and-turn program, change that one slightly—so that the robot moves toward obstacles: if an obstacle is detected in front, move forward; if an obstacle is detected at left, turn left; if at right, turn right; otherwise, stay still.

Students can again develop this idea however they like: for instance, turning on the LED in different colors according to where the robot senses an object, or reacting in different ways according to where the obstacle is sensed (so that putting a finger in front of different sensors is just another way of interacting with the robot to tell it what to do while it's running: for instance, a finger on the left could mean "spin around", in the middle "go forward", and on the right "stop").

For loops

Goal:	Learn about for loops, to repeat a section of code a number of times
Prerequisites:	While statements
Materials:	Computers, robots
Estimated time:	30 minutes

Introduce the for block in Minibloq.

Have students write a program to blink a sequence of LED colors 3 times, then stop.

★ Minibloq has a sample program for this activity. Go to the File menu and choose Examples; then, in the window that opens, double-click on the AERobot folder, then on 70.blink3Times, and finally on blink3Times.mbqc. ★

Recall the earlier activity where students wrote a program to have the robot move in a square. Have them rewrite that program, but instead of writing it in sixteen lines (explicitly repeating “forward, delay, turn, delay” four times), use a for loop to write it in five lines (using the loop to make the robot repeat that sequence).

Next, try making a shape with more sides!

Variables

Goal:	Understand and use program variables
Prerequisites:	For loops
Materials:	Computers, robots
Estimated time:	1 hour

So far, the robot behaviors have been based on reacting to something the robot encounters right then at that moment. But robots can also remember something they've done or encountered in the past, and use that to help them decide what to do.

A variable is a way of having a robot remember information. Show students how to use variables in Minibloq, using as an example putting a value into a variable and using that to set the length of time the LED stays on.

Have students create a new behavior that uses variables in some way. Example ideas:

- bump-and-turn but alternating turns left and right, switching off between whatever it did last;
- cycle through LED colors according to how many obstacles have been detected;
- turn further each time a new obstacle is detected.

Students can show off their favorite ideas in a class discussion.

Debugging

Goal:	Get practice with debugging, by looking at example programs, figuring out why they don't work as intended, and correcting them
Prerequisites:	Variables
Materials:	Computers; programs for debugging exercises
Estimated time:	1 hour

There are (at least) two kinds of program bugs: the kind that keep the program from running in the first place, and the kind that make the program (or robot) have unintended behavior. Minibloq prevents the first kind by only letting you put together blocks in ways that are allowed. The second kind is unfortunately unavoidable.

Give students the files for the first six debugging exercises (Ex1.mbqc—Ex6.mbqc), downloadable from the link above, and the descriptions of their intended behavior below. Have them work through (in groups or alone) these examples of programs with stated goals, that don't work quite as intended. By tracing through the program flow and considering what will happen in different situations the robot might be in, students should be able to figure out what's wrong with these programs and how to fix them. (In many cases, multiple solutions are possible.)

Note: The question may come up of why computer programming errors are called bugs. There's a story that the term came about after an error in an early computer was traced to a moth trapped inside. That did actually happen, and probably contributed to the popularity of the term for computer software errors, but it wasn't actually the origin—the term was being used in engineering many decades earlier. See, e.g., http://en.wikipedia.org/wiki/Software_bug#Etymology .)

Ex1.mbqc

Intended behavior: The robot should travel in a square.
Error: The programmer forgot to increment the loop variable *index*.
Faulty behavior: The robot keeps moving and doesn't stop after it's finished the square.
Corrections: Increment the loop variable inside the loop; or use a *for* loop instead of a *while* loop.

Ex2.mbqc

Intended behavior: Bump-and-turn: move forward; if any sensor detects an object, turn.
Error: The programmer used an *and* block instead of an *or* one.
Faulty behavior: The robot keeps moving forward unless all three sensors detect an object at once, which won't typically happen.
Correction: Replace the *and* block with an *or* one.

Ex3.mbqc
Intended behavior: "Frustration": move forward until an object is detected, and then turn; each successive time an obstacle is encountered, turn for a shorter and shorter period, until after the tenth obstacle, give up and stop moving altogether.
Error: The programmer mixed up the variables *x* and *y* in one place.
Faulty behavior: The robot moves for a longer period each time instead of a shorter one.
Corrections: In the delay statement, use *x* instead of *y*; or eliminate the variable *y* altogether, and delay for 10-*x* instead.
Bonus lesson: Using descriptive variable names is a great practice! It makes it much harder to mix them up.

Ex4.mbqc
Intended behavior: The robot should turn toward a light source.
Faulty behavior: The robot turns away from the light instead of towards it.
Corrections: Replace the < with a > ; or switch the left and right turn directions; or switch the left and right sensor readings.

Ex5.mbqc
Intended behavior: The robot should turn on its LED in a color according to the average distance measured over time by its front distance sensor: red if an obstacle has been close on average since the program started running, yellow for intermediate distance, green for far-away or no obstacle. (The longer the program runs, the harder it will be to influence the light color, because more samples will have been taken and the most recent sample carries correspondingly less weight.)
Error: The program makes a 0/0 division error right at the start, since it tries to divide *total_measured* by *num_samples* before updating either of them to nonzero values.
Corrections: Reorder the code to first take the sensor reading and only then perform the division and set the LED color; or initialize *total_measured* as the distance sensor reading and *num_samples* as 1.

Ex6.mbqc
Intended behavior: The robot should move forward if the average of the three distance sensor readings indicates no obstacles. (That is, the averaging in this case is being performed over space (the three sensors sampled at the same time), not over time (one sensor sampled at different times).)
Error: The programmer used the wrong operator in the first block (- instead of /).
Faulty behavior: The robot stops much more sensitively than intended (or may not move at all), since the "average" reading it gets is much larger than the programmer meant, giving the impression of an obstacle much closer than there really is.
Corrections: Replace the - block with a / one.

Text programming

Goal:	Start to get comfortable writing simple programs directly in C
Prerequisites:	Variables
Materials:	Computers
Estimated time:	1.5 hours

Minibloq has a text panel that shows C code corresponding to the graphical program.

Have students create an extremely simple program (any single statement) and look at the text panel to get a sense for the shell of code that goes around a program. What it does is not critical to understand; the important thing is that it sets things up and leaves space in the middle for the specific program code to go in.

Have students go through each of the Minibloq blocks they've used and put it into the graphical program to see how it corresponds to text program code.

Have students try writing C programs corresponding to previous activities, directly in the text window. The easiest way to do this is to load one of Minibloq's sample programs: go to the File menu and choose Examples; then, in the window that opens, double-click on the AERobot folder, then on 210.textCoding1, and finally on textCoding1.mbqc. This example already has some code in place; you can use this as a starting point for exploration, or remove everything except the following five lines:

```
#include <mbq.h>
void go()
{
    initBoard();
}
```

(Be aware that if you save a program you edit in this way, it'll overwrite Minibloq's example program!)

With some additional setup, it's possible to create a new program from scratch that lets you type C commands directly into the text window. Instructions for doing that are here.

Text debugging (ADVANCED)

Goal:	Get practice with text debugging, by looking at example programs in C, figuring out why they don't work as intended, and correcting them
Prerequisites:	Debugging, Text programming
Materials:	Computers; programs for debugging exercises
Estimated time:	1 hour

The Debugging activity mentioned two kinds of bugs. In addition to the kind that leads to unintended behavior, C programs can have *syntax errors*, where the program won't run at all because something was written wrong (like a typo).

Give students the files for the text debugging exercises (bump_and_turn_ex.c, Ex7.mbqc—Ex10.mbqc), downloadable from the link above, and the descriptions of their intended behavior on the following pages. Have them work through (in groups or alone) these examples of programs with bugs: first one with simple syntax errors to find, and later others that run but don't work as intended. By tracing through the program flow and considering what will happen in different situations the robot might be in, students will be able to figure out what's wrong with these programs and how to fix them.

1. Syntax errors

This exercise is based on a program written directly in C and meant to encode bump-and-turn behavior. The program logic is fine, but several syntax errors have been made. Students can look through an electronic or printed version of the faulty code and try to find all the errors, using the side-by-side graphical/text display in Minibloq for previous programs they've created as a guide for how things should look, or trying to reconstruct this one if they need to. The correct Minibloq program is given by bump_and_turn_correct.mbqc, and the corresponding correct C program by bump_and_turn_correct.c; these programs should not be given to students. The version with syntax errors is provided by bump_and_turn_ex.c. The errors are highlighted below:

```
void setup()
{
	initBoard();
	//Bump-and-turn
* missing variable declarations and initialization (float left_bumper = (bumpSens(LEFT));, etc.)
	while(true)
	{
		mov(FORWARD);  ← command move misspelled
		left_bumper = (bumpSens(LEFT));
		right_bumper = (bumpSens(RIGHT));
		front_bumper = (bumpSens(CENTER));
		if((left_bumper!=0))
		{
			move(TURN_RIGHT);
			delay(1);  ← capitalization wrong on command Delay
		* missing }
		else
		{
			if right_bumper!=0  ← missing parentheses around condition
			{
				move(TURN_LEFT)  ← missing semicolon
				Delay(1)  ← missing semicolon
			}
			else
			{
				if((front-bumper!=0))  ← hyphen used instead of underscore
				{
					move(BACK);  ← the name of the constant is BACKWARD
					Delay(1);
				}
				else
				{
				}
			}
		}
	}
}
```

2. Logical errors

These exercises, like the first set above, will run successfully but not produce the behavior intended. Unlike the first set, they are provided only as C programs. The second two in particular are tricky, involving errors that Minibloq prevents, and that will probably be difficult for inexperienced programmers to identify, but important for C programmers to be familiar with; trying to recreate the program in Minibloq and comparing the generated code could help students identify the errors if they can't find them directly.

It's also possible to run these C programs (or any others you write from scratch) directly on the robot through Minibloq—though it's a bit roundabout to make that work. To do so:
1. Create a new program in Minibloq. In the main window with the graphical program, click on the drop-down menu on the starting block, where it says "initBoard", and then click on "go".
2. Go to File → Save All, and save this program wherever you want to store it.
3. Go to Component → Build. (There will be errors; don't worry about them.)
4. Go to Component → Open Folder, which will open a new folder in the file manager (e.g., Windows Explorer or File Explorer) where you saved the program. You'll see the program (an .mbqc file) and a folder with the same name.
5. Open the .mbqc file in any text editor. In the second line, where it says "<files/>", change it to "<files> <f name="userCode.cpp"/> </files>". Save the file.
6. Go back to Minibloq, and with File → Open, reload your program. In the text window, you'll see two tabs: the first with userCode.cpp, the second with (your program name).cpp. Now you can enter any code into the userCode.cpp window. You can copy-and-paste any of the C programs corresponding to these exercises into the window, or write new code from scratch. *Important:* if you do paste a program from elsewhere into userCode.cpp, the line that normally says "void setup()" should be changed to "void go()"; and the last three lines that say "void loop() { }" need to be deleted. Otherwise you'll get errors when you try to compile your code.

Ex7.c

Intended behavior: The robot should travel in a square, then travel in a square back the other way, and then repeat both squares again.
Faulty behavior: The first square has 5 sides instead of 4, and the robot stops after the second square instead of repeating both.
Error: Two different ways of making a loop error. (The differences between the different loop statements should make it easier at least to find the issues, via comparison.) The outer loop initializes the loop variable to 1 instead of 0; because the termination condition is i<2, it will only run once instead of the intended twice. The first inner loop has a termination condition using <= instead of <, and so it will run for an extra time. The second inner loop has both substitutions and so works as intended!
Corrections: As the third loop shows, there are multiple ways to get the loop behavior to be what you want, but the best practice is to always do things the same way: initialize at 0, use < in the termination condition.

Ex8.c

Intended behavior: The robot should turn on its LED red when something is too close to its front distance sensor, and turn its LED off otherwise.
Faulty behavior: The LED goes on when no object is present and off when one is.
Error: Recall that the distance sensor returns higher values when things are closer.
Correction: Replace the < with >.

Ex9.c
Intended behavior: The robot should move forward whenever there's nothing in front of it, and stop when something gets too close to its front distance sensor.
Faulty behavior: The robot never moves.
Error: Semicolons are used to indicate the ends of statements (it's a little bit like English that way; at least there's some intuition available). If you leave out a semicolon at the end of a statement (as in the syntax error exercise above), the C compiler will tell you you've made a syntax error; it's like not ending a sentence. But if you put in a semicolon where you don't mean to, you can end a statement early and change its meaning. (It's like the difference between "I helped you fix your program." and "I helped you. Fix your program.") With a conditional statement (*if* or *while*), the structure goes like "If (such and such is true), then do this thing." If you put the semicolon early, then that indicates the end of the statement; you're not telling it to do anything if the conditional is true, and the statement after the semicolon becomes a separate command that's no longer related to the conditional. So in this case, the semicolon after *while(true)* turns the meaning into "While TRUE (i.e., forever), do nothing." And the program just gets stuck there, and never moves on to the next line.
Corrections: Take out the semicolon after *while(true)*.

Ex10 (HARD!)
Intended behavior: The robot should count the number of times an object appears in front of its front sensor. After 5 times, the LED should turn on orange. After 10 times, the LED should turn white.
Faulty behavior: The LED turns on white right away.
Error: There are actually two separate errors here, one logical, one super-sneaky. The first is that the robot is sampling the front sensor as fast as it can, so if you put an object there, it will count up to a high number very quickly. That would account for the LED turning white as soon as you put an object in front of the robot.
But there's a more insidious problem. Note the difference between the two conditions for setting the LED control. The first has a double-equals sign; the second has a single-equals. (That's why there are two separate LED-setting conditionals in this example, so that at least an observant student can notice the difference, even if they haven't learned what it means before.) The first two cases in the program with a single-equals are assignment statements, when you're telling the program you want to assign a new value to the variable x. The one with the double-equals is a comparison, where you're asking whether two things are currently equal. You might think that anything inside a conditional statement *if (*)* is recognized as being meant to be a comparison, not an assignment; but nope, C only pays attention to the number of equals characters in the expression. So *(x=10)* is also an assignment: x gets assigned the value 10, and then because of the way conditionals work, it counts as TRUE and the LED gets set to white.
This could be considered an unfair question to give to students with no previous exposure to C, but it's important because it's such a common error; even experienced programmers make this mistake, and it's very hard to find, so it's important to be aware of. (And if there are students with previous programming experience in C, this is a good exercise for them—in fact it might even be better to give them a version without the middle block of code that sets the LED to orange after 5 objects, so they don't have that extra cue of a correct == to tip them off.)
Corrections: (1) Add a delay of a second or so inside the *if bumpSens(CENTER)* block, along with the line incrementing *x*, to give a chance for the object to be taken back away. (2) Replace the single-equals in the last conditional with a double-equals.

Wall-following

Goal:	Write a program to make the robot move around the edge of a large obstacle
Prerequisites:	Variables
Materials:	Computers, robots; free-standing obstacles or materials to build walls
Estimated time:	1 hour

Brainstorm as a full class or in small groups about how to use the bump sensor to get the robot to move around the outside of a large obstacle. (A combination of moving forward and turning should work: e.g., move forward while bearing right; any time an obstacle is detected by the right bump sensor, turn left.)

Have students write programs to give the robot this behavior. If it doesn't work well on the first try, figure out where it falls short and how to improve it.

Maze-solving

Goal:	Write a program to let the robot find its way through simple mazes by itself
Prerequisites:	Wall-following
Materials:	Computers, robots; free-standing obstacles or materials to build walls
Estimated time:	1 hour

Start with a classroom discussion and brainstorming session: if you were trapped in a maze, what are some simple strategies you could try following to find your way out? (Ideas might include: keep your hand on one wall and just keep following it; take every right turn you come to; take turns at random.)

Have students write programs for their robots to let them find their way out of a simple maze, using the bump sensor to recognize walls.

Have students create a simple maze for their robot (or a partner's). (If the class did the open-loop maze-solving activity, and the mazes from that are still around, use those!) See how well the robot does; try to use the cases where it has trouble to understand where the program falls short, and to improve it.

Light sensing

Goal:	Learn about the idea and use of the robot's light sensors
Prerequisites:	If statements
Materials:	Computers, robots; penlights or other small flashlights
Estimated time:	30 minutes

If the class hasn't done the bump sensors activity, use the hardware handout or the diagram on the Hardware page online to point out the robot's three infrared receivers along its front edge. If the class has done that activity, remind them of these sensors.

These sensors can act as bump sensors or distance sensors, by having the infrared transmitters send out light, and the receivers measure the brightness of the light reflected from objects nearby. But they can be used in another way: the receivers can measure the brightness of visible light directly.

Have students write a program using the light sensor block, for instance to turn on the robot's LED if a light is detected. Test it in a dark room, using a flashlight to provide the light for the robot to sense. Figure out using trial and error how high a number the sensor reports when the flashlight is close.

Next, write a program that turns the LED a different color according to which sensor the light is shining brightest on.

Then, change this program to make the robot turn toward the direction of brightest light, and move toward it. Lead the robot around using the flashlight.

★ Minibloq has a sample program for this activity. Go to the File menu and choose Examples; then, in the window that opens, double-click on the AERobot folder, then on 140.lightSensors1, and finally on lightSensors1.mbqc. (This example and the next one, 150.lightSensors2, get around the problem of needing to know how large are the numbers the light-sensor block returns, by having the robot turn in one direction or the other according to which side the light is brighter on.) ★

Discussion: Behavior, intelligence, and life

Goal:	Discuss the lines between robots and animals, and think about what makes for intelligence
Prerequisites:	Light sensing
Optional materials:	Hexbug Nanos and habitat (example)
Estimated time:	45 minutes

Lead students through a discussion along the lines of the following outline:

1. What's the difference between a robot and an animal?
2. Focus first on the issue of preprogrammed, stereotyped behavior vs. intelligent thought and action, rather than metal and circuits vs. cells.
3. Think about increasingly simple animals. A human doesn't seem much like a robot, nor does a dog, but what about a goldfish? beetle? gnat? amoeba?

4. A famous experiment (first done by Henri Fabre in 1879) on digger wasps makes the issue more concrete. A certain kind of wasp digs a burrow, goes and gets a cricket, and lays her eggs in the burrow along with the cricket for them to eat when they hatch. But between digging the burrow and coming back with the cricket, something could happen to the burrow—for instance, another insect might move in, and eat both cricket and eggs. So when she returns with the cricket, the wasp first drops it on the threshold and goes into the burrow to check on it. She then emerges, drags the cricket in, lays her eggs, seals the burrow, and leaves. So far this seems like intelligent, reasoned behavior. But if, while she's inside the burrow during her initial inspection, the cricket is moved a few inches away, then when the wasp comes back out, she drags it up to the burrow entrance but not inside—instead she drops it again and goes back in to check again. If the cricket is moved again, she'll repeat the same routine again; this can go on dozens of times, until the experimenter gets bored repeating it. Now this seems like a robotic routine—if certain conditions are met, go on to the next step, otherwise go back to a previous step!
5. That makes intelligence appear to be more in the eye of the beholder. We attribute intelligence to the wasp because what it's doing makes sense for its survival and reproduction, in its normal environment. But in situations it doesn't normally encounter, suddenly its behavior looks very mechanical.
6. Could other apparently intelligent activity be equally mechanistic, especially with particularly simple animals? Conversely, could a robot's programmed behavior be considered intelligent?

7. Introduce a simple robot like a Hexbug Nano. It's essentially a vibration motor on a toothbrush head—it moves in the same sort of way as the Aerobot, but has no sensing, no way to react to the environment, and only one action: move straight ahead all of the time (if it bumps into something, it may turn just by bouncing off it). It would be a stretch to call it a robot (this connects back to the earlier discussion of robot characteristics).
8. Watch one or multiple hexbugs running in the same habitat or field of obstacles (either live or in a video, e.g., https://youtu.be/4fmNshvteUk?t=90). It looks like their behavior is complicated, and they give the sense of watching a bunch of insects, but all of the apparent complexity in what they're doing comes just out of mechanical influences from the environment.

9. That shows also that an agent’s activity in a given situation can vary from one instance to the next just through randomness in the world. A robot can also act differently in the same situation because of random “choices” it makes, like flipping a coin inside its head.
10. How could you decide if an insect or robot is acting only mechanically, or if there’s something more? What does intelligence mean?

Braitenberg vehicles

Goal:	Learn about and discuss simple Braitenberg vehicles
Prerequisites:	Discussion: Behavior, intelligence, and life
Materials:	Computer, robots; whiteboard or similar
Estimated time:	1 hour

Lead students through a discussion along the lines of the following outline, with sketches on the whiteboard:

1. A scientist named Valentino Braitenberg described several kinds of hypothetical "vehicles" in which sensory input was connected to motor output in various ways.
2. One of the simplest vehicles has one wheel and one light sensor on the front. If the light is brighter, the wheel turns faster. What does this vehicle do if you put a light in front of it? It goes faster and faster as it gets closer to the light, until it smashes into it.
3. If the wheel instead turns slower as the light gets brighter, and you put a light in front of the vehicle, it will approach but slow down as it approaches.
4. That's very simple and not very interesting. But now make it only a little more complicated: say there are two light sensors, on the front at left and right (like a car's turn signals), and two wheels. Say the right sensor is connected to the right wheel, and left sensor to left wheel, and the brighter the light is, the faster the connected wheel turns. Now what happens if you put a light in front of the vehicle and off to one side? The wheel on that side turns faster than the other (because the sensor on that side is closer to the light), and the vehicle goes forward but turns away from the light.
5. And now what if you make it so the wheel turns slower as the light gets brighter? The vehicle will turn toward the light and drive up to it, slowing down as it gets closer!
6. Suppose instead you have each sensor control the speed of the wheel on the opposite side. If the brighter light makes the wheel turn faster, it will turn toward the light and smash into it (or, depending on the turning speed and distance, you may be able to set up the vehicle to do something like orbit the light). If the brighter light makes the wheel turn slower, it will turn away and slow down.
7. A human observer watching these vehicles might attribute emotional state or personality to them. How would you interpret the first two-wheeled vehicle, that turns away from lights, running away faster if the light is brought closer and slowing down as it moves off into the dark? (Perhaps shy, or scared?) The second, which approaches the light and slows down? (Curious?) The third, which approaches the light and speeds up? (Aggressive?)
8. And again, all of these vehicles are incredibly simple and mechanical; everything else is in the eye of the observer. It could be that things like insects are also following simple rules, or that the attribution of intention and mental state to an agent could be equally valid for robots as for insects.

Try programming the robot to act like one of these Braitenberg vehicles, using the light sensors. (The AERobots don't let you control their speed directly, so students will need to either come up with a workaround or ignore the issue of speed here.)

Line sensing

Goal:	Learn about the idea and use of the robot's line sensors
Prerequisites:	If statements
Materials:	Computers, robots; hardware handout; paper and black markers
Estimated time:	30 minutes

With the handout (or with an LCD projector and the diagram on the Hardware page online), point out AERobot's line sensors on its bottom face: an infrared transmitter in the center, and two infrared receivers at left and right. These look down at the surface underneath the robot. The line-sensor block in Minibloq returns a value indicating whether there's a black line underneath the robot to its left, in the center, or to its right.

The light sensor needs to be calibrated, to work correctly for a given surface with a given amount of light in the room, similar to the way the motors need to be calibrated. Using a 1-cm-thick line drawn on white paper with a black marker, demonstrate the calibration routine, and have students perform it.

Have students write a program to turn the robot's LED a different color according to where the robot senses a line below it.

★ Minibloq has a sample program for this activity. Go to the File menu and choose Examples; then, in the window that opens, double-click on the AERobot folder, then on 180.lineAndColors, and finally on lineAndColors.mbqc. ★

Line following

Goal:	Program the robot to follow a black line drawn on the surface beneath it
Prerequisites:	Line sensing
Materials:	Computer, robots; white paper and black markers
Estimated time:	1 hour

Have students use the paper and markers to create a dark line on the surface underneath the robot. The line should be about the same thickness as the robot's USB plug, and wavy (straight is boring) but not too wiggly (or the robot won't be able to follow it).

Discuss how you could write a program to make the robot follow a line on the ground. What would it look like to the robot if it was following that behavior correctly? (Our answer: If the sensor says the line is in the center, move forward. If it says the line is to the left, turn left. If the line is to the right, turn right.)

Have students program the robot with this behavior.

Note: It can help to calibrate the robot's motion so that "left turn" and "right turn" include some forward motion, rather than just rotating in place.

Students with extra time should, as always, feel free to build on the core behavior. For instance, if the robot loses track of the line completely, it could stop and turn the LED red to indicate to the user to pick it up and put it back on the line.

★ Minibloq has a sample program for this activity. Go to the File menu and choose Examples; then, in the window that opens, double-click on the AERobot folder, then on 190.lineFollower, and finally on lineFollower.mbqc. ★

Multi-robot interaction

Goal:	Explore ways of having multiple AERobots interact with one another
Prerequisites:	As many activities as students want to have tools available!
Materials:	Computers, robots
Estimated time:	1 hour

Start with a short discussion about how, in general, animals act not completely in isolation but interact with others of their kind. (Even solitary animals communicate with others, even if it's only to mark and enforce their territory.) How could these robots interact?

Some starting ideas:

1. Could the bump or distance sensors could be used to detect the presence of another robot, by treating it as an obstacle?
2. Remember that those sensors work by emitting an infrared light and measuring its strength when it bounces back off an object. So one robot using its bump or distance sensors might emit a light that another one can detect.
3. The LED also emits light, and the light sensors can detect that. In a darkened room, can one robot turn on its LED white and move at random, and another one do light-following behavior to follow it? Could you chain a whole string of robots that way?

If students come up with particularly interesting behaviors and interactions, we're very interested to hear about them—please let us know!

Decorating the robots

Goal:	Personalize the appearance of the robots
Prerequisites:	None
Materials:	Robots; arts-and-crafts supplies (see below)
Estimated time:	1 hour

Here's the students' chance to decorate their robots to match their own vision! They can decorate it to look like a machine, an insect, a vehicle, or anything they like.

We recommend getting sheets of sticker paper; having students trace the shape of the robot's circuit board onto the sticker paper, and cutting it out; and decorating that paper and then sticking it onto the robot, rather than decorating the robot's body directly. This can help to prevent accidents like dripping glue that could cause problems for the robot; it makes it easier to remove and replace decorative "skins"; and in settings where it's not possible for each student to have and keep their own robot, it makes some personalization still possible although robots are reused.

In addition to the sticker paper and scissors, effective arts-and-crafts materials can include googly eyes, pipe cleaners, colored markers, glitter or glitter glue, sequins, craft feathers, craft beads, origami paper, etc.

Note 1: This was the most popular activity in our pilot sessions of this course, with universal appeal.

Note 2: Students should be aware (or may discover!) that the way they decorate the robot has the potential to affect its performance. Knowing where the robot's sensors and motors are, they should be aware about adding decorations that might interfere with those elements. Adding too much mass or changing its balance could change the way the robot moves (slow it down, give it difficulty turning, potentially even make it unable to move, etc.)

Free behavior programming

Goal:	Free exploration of programming robot behavior
Prerequisites:	As many activities as students want to have tools available!
Materials:	Computers, robots
Estimated time:	1 hour

With everything the students have learned about programming, behaviors, hardware, and intelligence, here’s their chance to come up with their own behaviors and program their robots to act that way.

This activity can be followed by a show-and-tell of favorite results, and/or a discussion of how things went, which could include questions like:

- What went right?
- What went wrong?
- How did you feel when things didn’t go as well as you had hoped?
- Were there problems that you anticipated?
- If so, what kept you from solving those problems?
- Was there something unexpected that you discovered?
- What was the best part about working with a partner/in a group?
- What was the hardest part?
- What was the most challenging part of the behavior to program?

Suggested optional video list

- Industrial factory robots on a car assembly line:
  https://www.youtube.com/watch?v=sjAZGUcjrP8
- iRobot PackBot (promotional video):
  https://www.youtube.com/watch?v=u5W6uwgQV20
- iRobot Roomba moving around a small room, showing it reacting to obstacles it encounters:
  https://www.youtube.com/watch?v=xTnaqED-5b4

Natural-looking, fluid motion:

- Boston Dynamics BigDog:
  https://www.youtube.com/watch?v=cNZPRsrwumQ
- BigDog parody:
  https://www.youtube.com/watch?v=VXJZVZFRFJc
- Boston Dynamics PETMAN:
  https://www.youtube.com/watch?v=mclbVTIYG8E
  https://www.youtube.com/watch?v=tFrjrgBV8K0

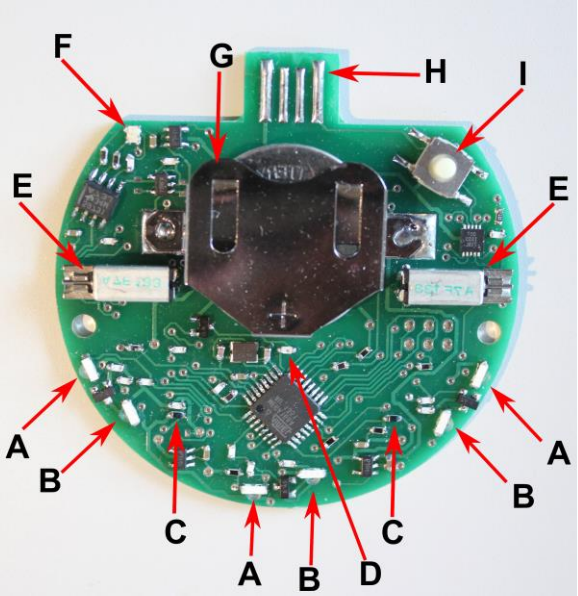
Different ways of getting around:

- above videos show walking (BigDog, PETMAN) and rolling with wheels (Roomba) or treads (PackBot)
- Somewhere between walking and rolling: RHex:
  https://www.youtube.com/watch?v=ISznqY3kESI
- Another form of a similar idea, closer to wheels, called “whegs”:
  https://www.youtube.com/watch?v=CdmwqdhKbhw
- Climbing: Stickybot:
  https://www.youtube.com/watch?v=EKDCv0l5ndY
- Jumping: Sand Flea:
  https://www.youtube.com/watch?v=6b4ZZQkcNEo
- Snake-like forms of movement:
  https://www.youtube.com/watch?v=T62E-_pQt3c
- Swimming:
  https://www.youtube.com/watch?v=1-iyPHse8uk
- Flying:
  https://www.youtube.com/watch?v=n_qRuHkD5lc#t=0m29s
- Sliding (exactly the same way as the AERobots):
  https://www.youtube.com/watch?v=cHbgrnv_8Nk
- and many others!

Robots inspired by insects:

- Termites (building):
  https://www.youtube.com/watch?v=LFwk303p0zY
- Ants (carrying large objects together):
  https://www.youtube.com/watch?v=qAjWL6AyIeE
- Flies/bees (flying):
  https://www.youtube.com/watch?v=b9FDkJZCMuE
- and many others!

AERobot Hardware



A Light/distance/bump sensors (3).
These *phototransistors* sense light falling on them.
They face outward, into the world.
- If used to measure the brightness of visible light, they act as light sensors.
- If used to measure the brightness of infrared light (emitted by **B**, reflected by objects in the world), they act as distance sensors.
- If used like distance sensors that only look at whether an object is closer than a certain distance, they act as bump sensors.

B Emitters of infrared light (3).
They face outward, into the world.
Infrared light emitted from them can reflect off nearby objects and be detected by the light sensors **A**. The closer the object, the brighter the reflected light.

C Line sensors (2).
These phototransistors face downward, toward the table, and measure the brightness of light emitted by **D** and reflected from the surface beneath.

D Emitter of infrared light (1).
It faces downward, toward the table.

E Vibration motors (2).

F Light-emitting diode (LED).
It can light up in any color.

G Battery (lithium-ion, rechargeable).

H USB connector.

I Power switch.