6.01 Infrastructure Guide

This guide outlines the 6.01 hardware and software infrastructure. We use ActivMedia Pioneer 3-DX robots, together with some hardware of our own invention, to support a variety of experiments in robot sensing and control and circuit design and construction. In addition, we have a program that simulates the robot's sensing and actions as it moves through the world.

Our robots have an on-board computer that does the low-level control and manages the sensors; this computer interacts with a laptop over a serial cable.

We control the real or simulated robot using a Python program called *soar*, which runs on a laptop. Soar lets you manage whether you're talking to a real robot or a simulator, and lets you drive with a joystick, or with a software "brain". The brain controls the robot using the transducer model (as described in Chapter 1 of the readings), which means the brain program has the job of mapping the current sensory inputs into the next action; it will be executed several times per second. You can run the soar simulator on your own computer or on an Athena* machine, but to control the robot you need to run on one of the lab laptops. The descriptions below apply to the lab laptops, but use of the soar simulator is similar on your laptop or on Athena*.

1 Robot sensors and effectors

Our robots have several different sensors (ways of gaining information about the external world) and effectors (ways of changing the state of the outside world).

Wheels

The robots have two drive wheels, one on each side, and a caster wheel in the back. You can command the robots by specifying a forward velocity in meters per second and rotational velocity in radians per second. The control software will convert that into signed velocities for each wheel.

Sonars

The robots have eight ultrasonic transducers (familiarly known as sonars); these are the gold-colored disks on the front. They send out pulses of sound and listen for reflections. The raw sensors return the time-of-flight values, which is how long it took the sound to bounce back. The robot's processor converts these into distance estimates.

When the sonars send out a sound pulse, there is a period known as the "ring-down period" where the sensor is still vibrating from producing the pulse, and can't listen effectively for echos. If an object is so close that the echo returns during the ring-down period, the sensor will not detect the object. Similarly, objects that are too far away will not return an echo. The sensors are generally fairly reliable if the closest object is between 10 centimeters and a meter away from the robot. If no echo is detected by the sensor, a distance that is well outside the range of the sensor is returned.

Depending on the material of the object in front of the sonar sensor, the sound pulse may be reflected off in some other direction, rather than back toward the sensor. This will also cause the sensor to fail to detect the object. To combat this problem we have wallpapered all of our "walls"

*Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

with bubble wrap, which causes the pulses to bounce off the wall in many directions, making it likely that the sensor will detect reflections.

Odometry

The robot has *shaft encoders* on each of its drive wheels that count (fractions of) rotations of the wheels. The robot processor uses these encoder counts to update an estimated *pose* (a term that means both position and orientation) of the robot in a global reference frame.

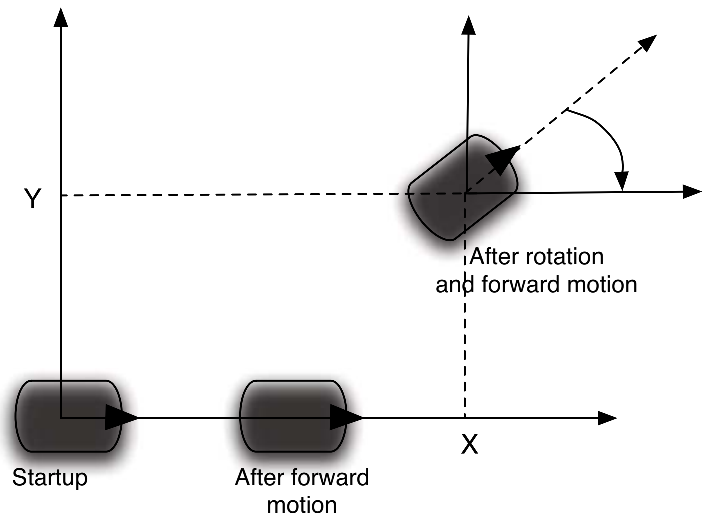


Figure 1 Global odometry reference frame

Figure 1 shows the reference frame in which the robot's odometry is reported. When the robot is turned on, the frame is initialized to have its origin at the robot's center and the positive x axis pointing out the front of the robot. You can now think of that frame as being painted on the ground; as you move or rotate the robot, it keeps reporting its pose in that original frame.

Note that the pose reading doesn't change if you pick up the real robot and move it, without the wheels turning. It only knows that it has moved because of the turning of the wheels. And remember that the odometry is far from perfect: the wheels slip and the error (especially rotational error) compounds over time.

Analog-to-Digital interface

We can connect additional sensors and effectors to the robot via an analog-to-digital interface. There is a cable coming out of the robot that supplies connections to four analog-to-digital converters (these allow us to convert voltages on input wires to values that can be read via software) and one digital-to-analog converter (this allows us to convert a number written in software to an output voltage). See Section 7.4 and Section 7.5 for details on the hardware connections for these inputs.

2 Brains

A soar brain has the following structure:

```
def setup():
    print "Loading brain"
def brainStart():
    print "Starting"
def step():
    print "Hello robot!"
def brainStop():
    print "Stopping"
```

The `setup` procedure is called once, when the brain file is read by the interpreter. The `brainStart` procedure is called when you start running the brain. The `step` procedure is called over and over, several times per second. The above brain, if run, will first print `Loading brain` when you load (or reload) the file, and will then print `Starting` when soar's **Start** button is pushed, and then just keep printing `Hello robot!` in the soar message window until you stop it, at which point it will print out `Stopping`.

We will generally use state machines to control our robots. Here is a very simple brain with a state-machine controller:

```
import lib601.sm as sm
from soar.io import io

class MySMClass(sm.SM):
    def getNextValues(self, state, inp):
        return (None, io.Action(fvel = 0.1, rvel = 0))

def setup():
    robot.behavior = MySMClass()
    robot.behavior.start(verbose = True)

def step():
    robot.behavior.step(io.SensorInput()).execute()
```

The program starts by loading some support files. Then, we have shown the definition of a trivial state machine, which always generates as output an action that sets the robot's forward velocity to 0.1 m/s and its rotational velocity to 0. We will call any state machine that consumes a stream of instances of the `SensorInput` class and produces a stream of instances of the `Action` class a *behavior*. Note that both `SensorInput` and `Action` are defined in a soar module called `io`, and so we have to prefix them with their module name.[1]

The `setup` function instantiates the `MySMClass()` behavior (if you want to use a different behavior, you instantiate it here) and starts it. The argument `verbose` to `start` causes the input, output, and state of the SM to be printed; you can suppress printing by setting that argument to `False` (or by deleting the optional `verbose` argument from the call).

Then, finally, in the `step` function, we call `io.SensorInput()` which creates an instance of the `io.SensorInput` class, containing the most recent available sensor data. That is passed as input to the state machine's `step` method, which calls its `generateOutput` method, which returns an

[1] Alternatively, we could have put the line `from io import *` at the top of the file, which would then have allowed us to say `Action` instead of `io.Action`. As a matter of policy, we try not to do this, because in a complicated software system it can become difficult to figure out what module a particular class or procedure is defined in, and sometimes name clashes (where two modules both define `getBest`, for example) ensue.

instance of the `io.Action` class. Finally, we ask that action to execute, which actually sends commands to the robot. If the way it is written seems confusing, here is an alternative version that makes it clearer what is happening

```
def step():
    inp = io.SensorInput()
    act = robot.behavior.step(inp)
    act.execute()
\stoptypine

The attributes of an {\tt io.Action} instance are:
\startitemize
\item {\tt fvel}: a single real number specifying forward velocity in
  meters per second.  A positive value moves the robot forward, and a
  negative value moves the robot backward.
\item {\tt rvel}: a single real number specifying rotational velocity
  in radians per second.  A positive number causes the robot to turn
  left and a negative number causes the robot to turn right.
\item {\tt voltage}: a single number in the range 0 to 10, specifying
  the voltage to be transmitted to the digital-to-analog converter
\stopitemize
Passing those three values into the initializer will make a new
instance.  Calling the {\tt execute()} method on an {\tt Action}
instance will cause the robot to execute that action.

The attributes of a {\tt io.SensorInput} instance are
\startitemize
\item {\tt sonars}:  a list of 8 values from the sonar sensors;
  distances measured in meters; with the leftmost sensor being number
  0
\item {\tt odometry}:  an instance of the {\tt util.Pose} class, which is
  described in more detail in the online software documentation
\item {\tt analogInputs}:  a list of 4 values from the
  analog-to-digital encoders, in the range 0 to 10.
\stopitemize
In the simulator only, it can sometimes be useful to ‘‘cheat’’ and
make reference to the robot’s {\em actual} coordinates in the global
coordinate frame of the simulator (rather than in the odometry frame
which is where the robot was when it was started).  If you make an
instance of {\tt io.SensorInput} this way:
\starttyping
s = io.SensorInput(cheat = True)
```

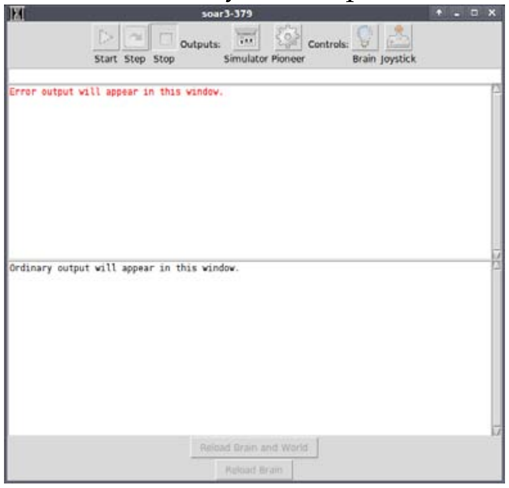
then the odometry values will represent actual coordinates in the simulator. Executing this on the robot will cause an error. The situation in which using the cheat pose makes a real difference is when you drag the simulated robot around on the screen: the regular odometry doesn't know that the robot has moved (because the wheels haven't turned), but the cheat pose will still know exactly where it has ended up. Use this for special debugging only, because the real robots can't do anything like this!

3 Using the Soar simulator

To use a brain to control the simulated robot, do the following:

- Type

  ```
  > soar
  ```

  to the prompt in a terminal window on your computer. This should bring up the soar window, which looks like this: 

  ```
  Error output will appear in this window.
  ```

  ```
  Ordinary output will appear in this window.
  ```

- Click on the **Simulator** button, and pick the `tutorial.py` world.
- Click on the **Brain** button, use the file dialog to navigate to `Desktop/6.01/designLab02/smBrain.py` and select it.
- Click the **Start** button.

The robot will drive forward and print some output in the window. To stop the robot, click the **Stop** button. Don't worry if it keeps printing for a while. All that text has been stored up somewhere waiting to be printed.

In general, you define a soar brain in a file and load it into soar. If there is an error upon loading the brain, you will see an error message (in red). If you need to edit the brain file, you can edit in the editor and then tell soar to reload the brain by pressing the "reload brain" button.[2] The **Reload Brain and World** button will also reload the brain, and in the simulator, it will place the robot back in its initial pose. On the Pioneer, reloading the world will reset the serial connection to the robot, which makes the robot reset its odometry to the origin.

For the sonars on the simulator, you'll see lines protruding from the robot at various points: these represent the signal paths of the sonar sensors. These simulated sensors follow an ideal model, in that they always measure the distance (with a small amount of added random noise) down that ray to the first surface they contact. The line is drawn in red when the sonar does not detect any reflection. In the simulator, you can pick up and drag the robot with the mouse while the brain is running and watch how the readings change.

You may notice that the simulated robot occasionally turns red. That means it is near or in collision with the walls. If it is stuck and you send it commands to move forward, nothing will happen. You can still back up or turn it, though.

[2] If you use idle as the editor, remember that you *don't* run the brain with "run module" from the run menu. You can, however, check the syntax of your code with "check module," which could be a useful thing to do before loading it into soar.

4 Using the Pioneer robot

You can connect a robot to a lab laptop by (1) connecting a short blue USB adapter cable to your laptop, (2) connecting the USB adapter to a one end of a serial cable, and (3) connecting the other end of the serial cable to the serial connector on the robot, which is near the panel with the power switch and the indicator lights on the robot's side.

Be sure to tie the serial cable in a simple knot around the handle on the back of the robot before you plug it in. This provides strain relief, preventing the serial cable from being pulled off the robot or bending the connector if the cable is tugged.

Start by turning the robot on. To drive the robot around using the joystick, click **Pioneer** and **Joystick**, and then **Start**. Clicking and draging in the resulting Joystick window will control the robot's forward and rotational velocity.

Hints

for running the robot:

- When you're going to run the robot, make sure that the robot is turned on and the serial cable is plugged in at both ends.
- The robots are pretty sturdy but (a) dropping them or (b) crashing them into a wall can damage them. Please be careful. If the robot starts to get away from you, **grab it and pick it up** and turn the power off.
- Sometimes you can do basic debugging by tilting the robot on its back end, with its wheels in the air, so you can see the wheels turning (and in what direction) without the robot actually moving.
- If the robot looks completely dead, that is, there are no indicator lights lit when the power is on, it may have blown a fuse. Talk to a staff member.
- If you're not having luck, try this process:
  - Quit soar and turn the robot off;
  - Restart soar and turn the robot on;
  - *After that*, pick the robot icon in soar; then
  - Listen for the "tickety-tick" sound of the sonars.
- If the serial cable comes unplugged for some reason, perform the procedure above.

5 Tracing and graphing

Soar has facilities for doing various kinds of tracing and graphing for debugging your programs and documenting your results. Here we outline each of these facilities.

5.1 Seeing what the robot sees

Often it is important to have a good understanding of what the robot's sensors are telling it, either to understand why the robot is behaving the way it is, or to document the result of an experiment. There are a few useful tools for doing this. To use many of these tools, you must create a `RobotGraphics` instance in your setup function. This will typically look like this:

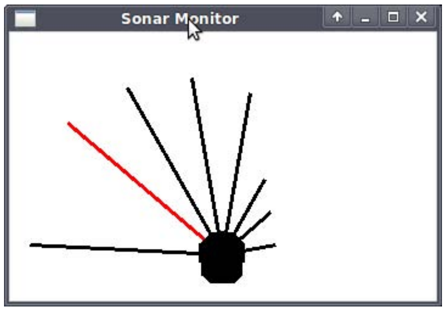
```
import gfx
```

```
def setup():
    robot.gfx = gfx.RobotGraphics(sonarMonitor = True,
                                  drawSlimeTrail = True)
```

The slime trails and static plots (described below) will be drawn when the brain is stopped. This is always done when you push the soar **Stop** button, but if you'd like the brain to stop automatically when your controlling state machine is done, you can add the following line to your `step` function:

```
    io.done(robot.behavior.isDone())
```

5.1.1 Sonar monitor



It can be useful to see the robot's sonar readings in real time, either to detect a problem with the hardware, or as a debugging tool for better understanding your brain's behavior. The sonar monitor, seen on the left, provides a graphical display of the robot's sonar readings, whether soar is connected to a simulator robot or a Pioneer. Just like the sonars in the simulator, the sonars are represented by a black line which turns red when the sonar hears no echo at all.

You can control whether or not the sonar monitor is displayed by changing the `sonarMonitor` parameter of the `RobotGraphics` constructor, which is a Boolean. The default is `False`.

5.1.2 Slime trails

It is often useful to know where the robot was, during execution. We can't know this exactly on a real robot, but we can get some idea of what is happening by looking at the pose reported by the odometry. We do this by drawing a "slime trail" of the robot's path, as if it were a snail. The color of the trail is varied continuously along it, to give you a sense of the time at which particular locations were occupied. **Figure 2** shows an example slime trail.

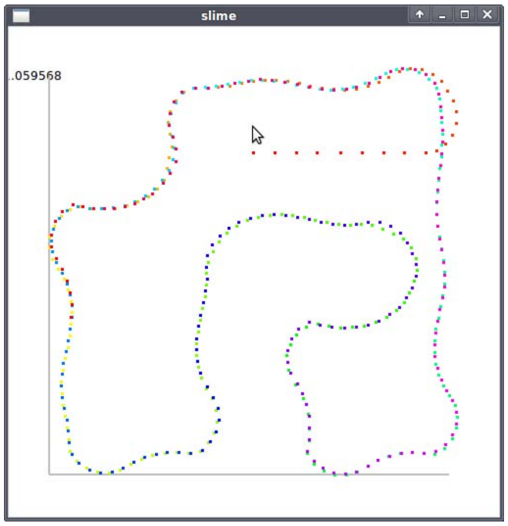
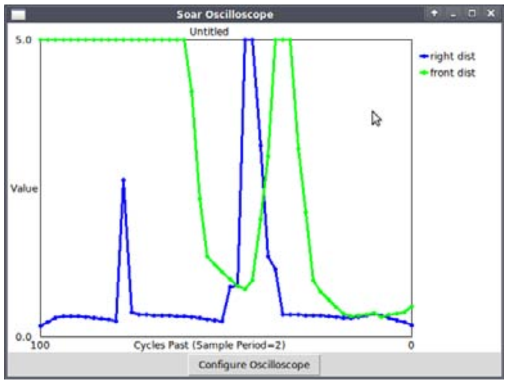


Figure 2 Slime trail of a wall-following robot in the `tutorial` simulator world.

If the `drawSlimeTrail` argument of the `RobotGraphics` constructor is set to `True` a slime trail window will pop up when the brain stops running. If you are running on the simulator, you can set `drawSlimeTrail` to be `'Cheat'`, and this will use the true position of the robot rather than the robot's internal odometry measurements so that when you drag the robot, the slime trail will show the motion.

5.1.3 Dynamic plotting

Another useful tool, the virtual oscilloscope, provides a real-time plot of some arbitrary function of the robot's sensor values. The `addDynamicPlotFunction` takes a `(name, function)` tuple, where `name` is a string to display in the legend, and `function` is a function that takes a `SensorInput` instance and returns a number.



The figure on the right shows the oscilloscope window plotting the minimum distance reported by the two right-most sonar sensors ('right dist'), and the average distance reported by the two front sonars ('front dist'). We generate this plot by adding the following lines to the `setup` function.

```
robot.gfx.addDynamicPlotFunction(('right dist',
                                  lambda inp: min(inp.sonars[6:])))
robot.gfx.addDynamicPlotFunction(('front dist',
                                  lambda inp: sum(inp.sonars[3:5])/2.0))
```

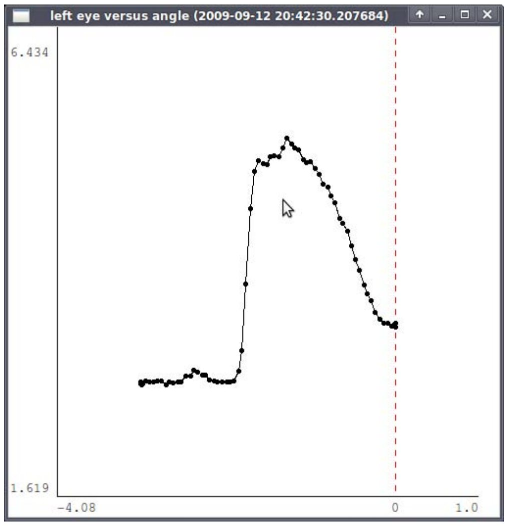
The **Configure Oscilloscope** button allows you to adjust the axis bounds and labels on the plot. By default, the y-axis bounds are the minimum and maximum y-value seen so far.

It is also possible to plot multiple signals with one call to `addDynamicPlotFunction` by providing a list of names and a function which returns a list of values. For example, this code will plot the first three analogInput signals, assigning each signal a name that corresponds to the device it will be connected to in the circuits labs:

```
robot.gfx.addDynamicPlotFunction((['neck', 'left', 'right'],
                                  lambda inp. inp.anaglogInputs[0:3]))
```

While the dynamic plotting can be a very useful debugging tool, you will sometimes need to turn it off to get optimal performance from your brain, as the graphics tasks can eat up a lot of compute cycles.

5.1.4 Static plotting



Sometimes you may want to plot a signal as a function of another signal, rather than as a function of time. Also, you may want to generate plots of multiple signals on separate axes if their magnitudes are significantly different. Or you may find that the real-time plotting takes too much time at each step, and causes your brain to behave strangely. In these cases, you can use static plotting instead, which will simply store the values you want to plot, and then generate a plot window for each function when the brain stops.

The `RobotGraphics` class provides an `addStaticPlotFunction`, which is similar to the `addDynamicPlotFunction`, except that it can take two different signals, so it is important to specify which is the x-axis signal and which is the y-axis signal. If no `x` argument is provided, the default is to plot the signal as a function of the number of steps.

The code below, placed in the `setup` function, will produce a plot like the one in the figure: the intensity of light in the robot's left eye as a function of the robot's angle relative to its initial orientation. The `connectPoints` argument is a Boolean specifying whether or not to draw a line between each point in the plot.

```
startTheta = io.SensorInput().odometry.theta
robot.gfx.addStaticPlotFunction(
    x=('angle', lambda inp: startTheta-inp.odometry.theta),
    y=('left eye', lambda inp: inp.analogInputs[1]),
    connectPoints=True)
```

5.2 State machine information

Another important source of information for debugging your program is the internal state of the program itself. Since all of our robot controllers are composed of state machine instances, the state machine class provides a number of techniques for printing and plotting important values.

5.2.1 Plotting state machine variables

We can produce static or dynamic plots of state machine inputs, states, and outputs in much the same way that we produce plots of sensor values. We will first illustrate this process with an example, and then explain the steps in detail. The example shows how to make a static plot of the forward velocity of the output of a wall-follower state machine.

1. Make sure the state machine of interest has a name (more naming techniques are below):

```
myMachine = WallFollowerSM()
myMachine.name = 'wallFollow'
def setup():
    robot.behavior = myMachine   # could be combination of myMachine with others
```

2. Add a static state machine "probe" to the `RobotGraphics` instance:

```
    robot.gfx.addStaticPlotSMProbe(y=('forward vel', 'wallFollow', 'output',
                                      lambda outp: outp.fvel))
```

3. Start the state machine with a list of graphics tasks (this will typically be done by default):

```
    robot.behavior.start(robot.gfx.tasks())
```

Once you have done those three steps, you should get a static or dynamic plot of your state machine quantity when you run your brain. The first step requires you to **name** the machine you want to monitor. There are several ways to do this:

- Make an instance of a state machine, and then add a `name` attribute.

```
myMachine = Increment()
myMachine.name = 'myFavoriteMachine'
```

- Make a machine using one of the `SM` combinators, and supply a `name` at initialization:

```
myMachine = sm.Parallel(Increment(),Increment(), name = 'incrementInParallel')
```

- Define your own new `SM` subclass that takes a `name` parameter in the initializer and sets the `name` attribute of the object:

```
class GoForwardAtSpeed(sm.SM):
    def __init__(self, name = None):
        self.name = name
    def getNextValues(self, state, inp):
        return (None, io.Action(fvel = inp, rvel = 0))
myMachine = GoForwardAtSpeed(name = 'speedy')
```

The second step involves making a **state machine probe** to give to the `RobotGraphics` instance. The state machine probe is a tuple: `(signalName, machineName, mode, plotFunction)`.

- `signalName` is the name that will be associated with the signal in the plot.
- `machineName` is the name of the machine you want to probe.
- `mode` is one of `'input'`, `'state'`, `'output'`, and indicates which of these quantities of the machine you are interested in.
- `plotFunction` is a function of the input, state, or output of the machine that produces the value to be plotted.

You can pass such a probe to either `addStaticSMProbe` or `addDynamicSMProbe` as appropriate.

5.2.2 Verbosity

Sometimes the value you are interested in is better observed by printing the value rather than plotting it. The simplest way to print lots of debugging information is to set the `verbose` parameter to `True` (it is `False` by default) and the `compact` parameter to `False` (it is `True` by default), when you call `start` on your behavior. This will print the input, output, and state of each primitive machine, and the state of the combinator machines, on each step. Each machine's `name` will also be printed out. By default, names are assigned by the system, as the class of the machine instance, and a number for uniqueness. The printout will look something like this:

```
Step: 0
 Cascade_1
     upDownSM Counter = 0
         Sequence_2 Counter = 0
             CountUntil_3 In: Sonar: [5, 5, 1.491, 0.627, 0.330, 0.230, 0.200, 0.265]; Odo:
pose:(-1.633, -1.341, 4.067); Analog: (0.000, 0.000, 0.000, 0.000) Out: -0.400 Next State:
-0.400
Output of upDownSM = -0.4
     forwardSpeedSM In: -0.400 Out: Act: [-0.400, 0, 5.000] Next State: Act: [-0.400, 0,
5.000]
Step: 1
 Cascade_1
     upDownSM Counter = 0
         Sequence_2 Counter = 0
             CountUntil_3 In: Sonar: [5, 5, 5, 0.669, 0.356, 0.251, 0.219, 0.289]; Odo:
pose:(-1.614, -1.316, 4.067); Analog: (0.000, 0.000, 0.000, 0.000) Out: -0.300 Next State:
-0.300
Output of upDownSM = -0.3
     forwardSpeedSM In: -0.300 Out: Act: [-0.300, 0, 5.000] Next State: Act: [-0.300, 0,
5.000]
```

If this mode is too verbose, there are two things you can do:

1. Set the `compact` parameter to `True` (it is `True` by default) when you call `start`. In this case, the output will look more like this. You can see the whole state of the machine, but it's hard to tell what pieces of it belong to which constituent SMs:

```
In: Sonar: [2.465, 1.352, 1.350, 2.465, 1.427, 1.139, 1.060, 0.785]; Odo: pose:(-0.987,
1.493, 2.443); Analog: (0.000, 0.000, 0.000, 0.000) Out: Act: [-0.400, 0, 5.000] Next
State: ((0, (0, -0.5)), None)
In: Sonar: [2.465, 1.381, 1.386, 2.465, 1.466, 1.165, 1.087, 0.739]; Odo: pose:(-0.957,
1.468, 2.443); Analog: (0.000, 0.000, 0.000, 0.000) Out: Act: [-0.300, 0, 5.000] Next
State: ((0, (0, -0.40000000000000002)), None)
```

2. Set the `printInput` parameter to `False` when you call `start`. This will skip printing the input, which is sometimes useful if the sensory data is getting in the way. In this case, the output will look more like this:

```
Out: Act: [-0.400, 0, 5.000] Next State: ((0, (0, -0.5)), None)
Out: Act: [-0.300, 0, 5.000] Next State: ((0, (0, -0.40000000000000002)), None)
```

5.2.3 Printing state machine values

If all of that printing is too much, and you just want to see a particular value, you can create your own **trace task** and give it to the state machine when it is started. These tasks are simply added to the list of tasks created by the `RobotGraphics` instance. This example shows how to print out the state of two state machines that are part of the main behavior machine, and have been named `'sm1'` and `'sm2'`:

```
robot.behavior.start(traceTasks = robot.gfx.tasks() +
                     [('sm1', 'state', util.prettyPrint),
                      ('sm2', 'state', util.prettyPrint)])
```

The state machine trace task is much like the state machine probe described in **Section 5.2.1**. It is a tuple: `(machineName, mode, traceFunction)`.

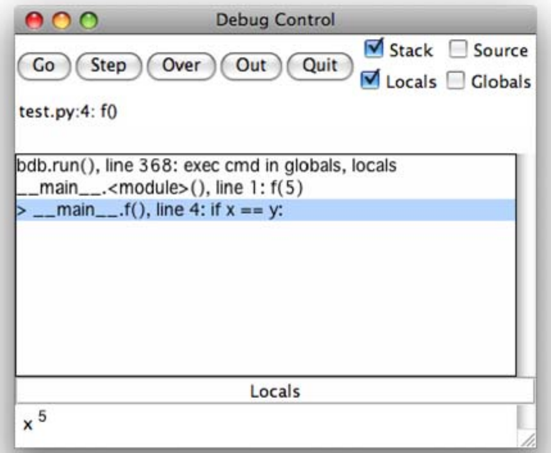
- `machineName` is the name of the machine you want to trace.
- `mode` is one of `'input'`, `'state'`, `'output'`, and indicates which of these quantities of the machine you are interested in.
- `traceFunction` is a function of the input, state, or output of the machine that prints out whatever you want to be printed. The function `util.prettyPrint` is a utility function we have provided that will print out many things in nice, readable format. You are welcome to use your own function here if `prettyPrint` doesn't do what you want.

6 Debugging in Idle

Idle has a built in debugger. There are two styles of debugging: (a) steping through your program, or (b) waiting for an error and look to see what happened.

6.1 Stepping through your program

Click on the Debug menu and choose Debugger. You will see `[DEBUG ON]` in the Python Shell and a window will pop up. Now type the expression that you want to evaluate in the Python Shell. You will see it appear in the window. Try clicking Step repeatedly to see the execution of the program. Click Out to finish execution without further debugging. Click Quit to exit the debugger.



Stack ✓ Source ☐ Locals ✓ Globals ☐

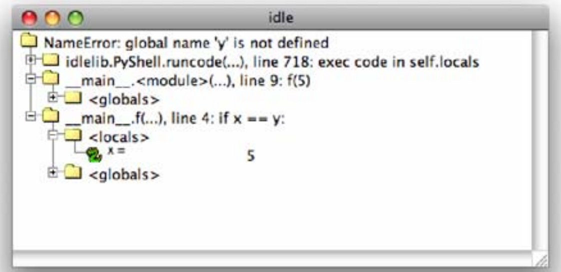
```
test.py:4: f()
```

```
bdb.run(), line 368: exec cmd in globals, locals
__main__.<module>(), line 1: f(5)
> __main__.f(), line 4: if x == y:
```

```
x 5
```

6.2 Post Mortem

If you get an error in your code during normal execution, click on the Debug menu and choose Stack Viewer. This will open a window that will show you a line for each of the procedures that is currently active. If you click on the + signs at the left of each line, you can see entries for the `globals` and `locals`, these are the variable bindings that are active at each procedure call. This should make sense to you after reading Chapter 3.



```
idle
NameError: global name 'y' is not defined
  idlelib.PyShell.runcode(...), line 718: exec code in self.locals
  __main__.<module>(...), line 9: f(5)
    <globals>
  __main__.f(...), line 4: if x == y:
    <locals>
      x =                 5
    <globals>
```

7 Equipment for circuit labs

This section provides a list of parts that are available in the lab.

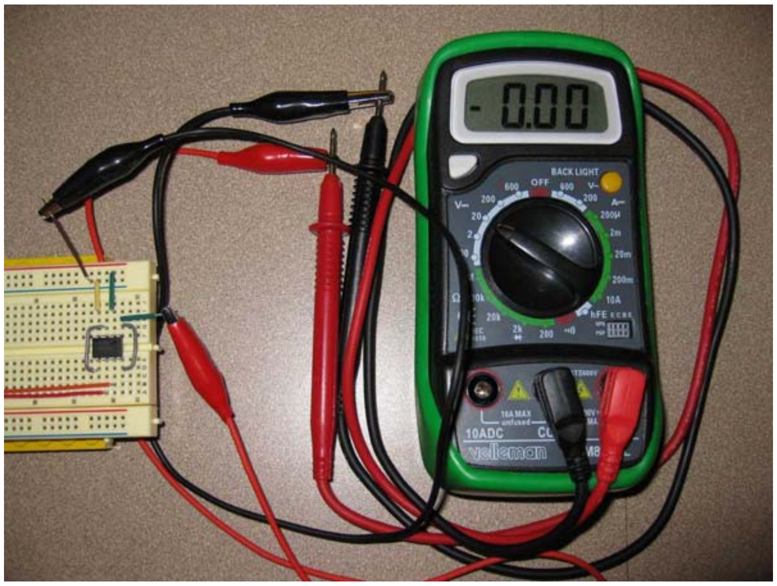
7.1 Power supply

The power supply has four terminals (one black and three red). To connect a wire to a terminal, use a wire with alligator clips on both ends. Then there is no need to unscrew the terminal knobs.

It is a voltage source. We will use the black terminal as negative ('ground') terminal, and the red terminal marked +15V as the postive terminal. The knob below the +15V terminal actually controls the voltage of this source, from 0V to +15V.

7.2 Multimeter

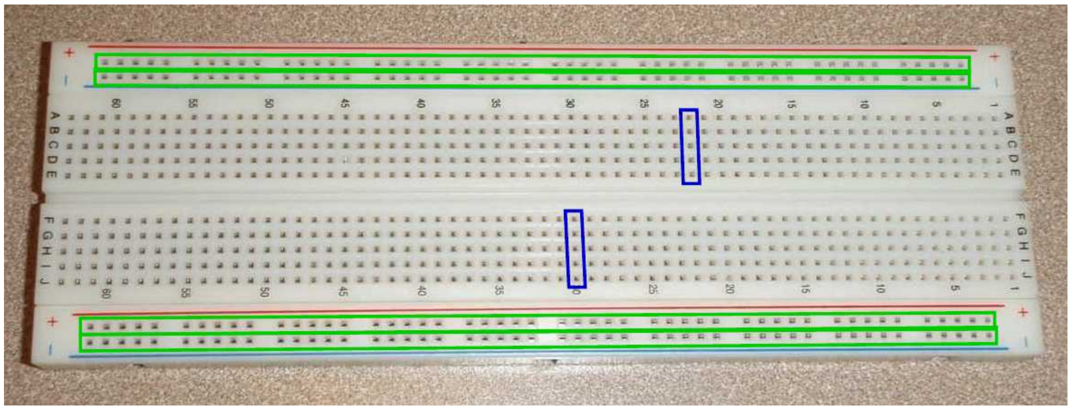
Our multimeters allow you to measure current, voltage, and resistance. You connect the multimeter to a circuit using two leads. You can use The black lead should be plugged into the ground (common) jack. The red lead should be plugged into a jack labeled "V-Ω-mA," except when measuring currents that are larger than 200 mA, when the "10 ADC" jack should be used instead.



Because the meter probes are large, they can bridge, and thereby make unwanted electrical connections ("short circuits") between adjacent pins of small components. Such short circuits can damage your circuit. To avoid this, you can measure the resistance or voltage across points in your breadboard by using short wires that are connected to the meter probes with alligator cables, as shown above. Alligator leads are colorful shielded wires with an 'alligator' clip on each end. You can squeeze the alligator clip to clamp onto another wire or a meter probe or a terminal of the power supply.

7.3 Breadboard

You will build your circuits on a "breadboard" like the one shown below.

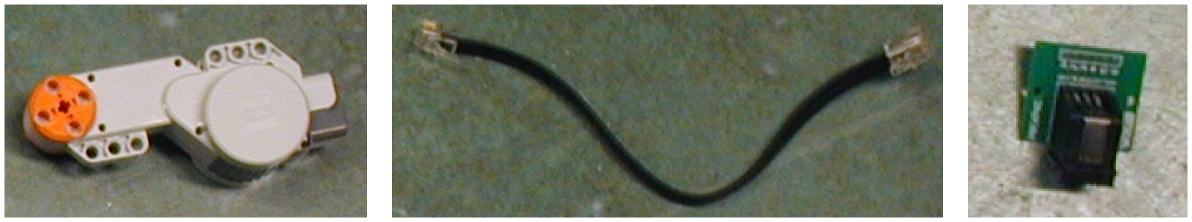


The breadboards have holes into which wires and components can be inserted. Holes in the long top row (labeled +) are connected internally (as are those in the second row, bottom row and next-to-bottom row), as indicated by the horizontal (green) boxes (above). These rows are convenient for distributing power (+10 V) and ground. Each column of 5 holes in the center areas is connected internally, as indicated by two representative vertical (blue) boxes (above). Note that the columns of 5 are *not* connected across the central divide.

7.4 Connection to Motor

Note: the connectors to the motor and to the robot are different. The motor connector has 6 pins and the robot connector has 8.

Electrical connections to a motor are provided by a a 6-pin cable with an RJ11 6P6C connector, which can be interfaced to the breadboard with an adaptor, shown below.

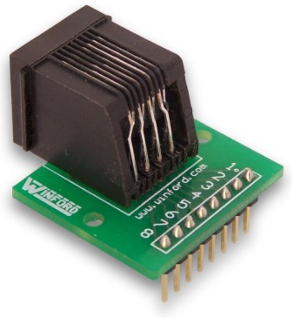


We only ever use pins 5 and 6 of this connector. Connecting them to +12V and GND will make the motor spin.

7.5 Connection to Robot

*Note: the connectors for the robot and the head are the **same connector**, so you need to take care that you do not get them mixed up when plugging in the cables. The connectors to the motor and to the robot are different. The motor connector has 6 pins and the robot connector has 8.*

Electrical connections to the robot are provided through an 8-pin cable with an RJ45 8P8C connector, which can be interfaced to the breadboard with an adaptor, shown below.



pin 1:	V_{i1}	analog input #1
pin 2:	+10 V	power (limited to 0.5 A)
pin 3:	V_{i2}	analog input #2
pin 4:	ground	
pin 5:	V_{i3}	analog input #3
pin 6:	V_o	analog output
pin 7:	V_{i4}	analog input #4

The pins provide power as well as four input channels and one output channel. Normally, one would connect pin 2 (+10 V) to the positive power rail of the breadboard and pin 4 (ground) to the negative (ground) rail.

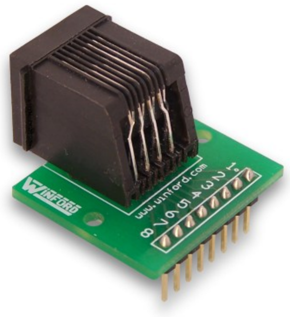
The analog inputs must be between 0 and +10 V and are converted to a digital form with 12-bit resolution. They can be accessed in `soar` by looking at the `analogInputs` attribute of a `io.SensorInput` instnace, which is a list of the form $[V_{i1}, V_{i2}, V_{i3}, V_{i4}]$. Analog inputs are updated just once per `step` in a `soar` brain.

The analog output signal from the robot (V_o) is between 0 and +10 V and is produced with 8 bits of resolution. It can be set by calling `analogOutput`, which takes a number between 0 and 10 as input and returns `None`. The analog output is set just once per `step` in a `soar` brain, using the value set by the most recently executed `io.Action` instance.

7.6 Connection to Head

*Note: the connectors for the robot and the head are the **same connector**, so you need to take care that you do not get them mixed up when plugging in the cables. The connectors to the motor and to the robot are different. The motor connector has 6 pins and the robot connector has 8.*

The pinout for the head connector is shown here.



pin 1:		neck pot (top)
pin 2:		neck pot (center)
pin 3:		neck pot (bottom)
pin 4:		photoresistor (left)
pin 5:		photoresistor (common)
pin 6:		photoresistor (right)
pin 7:	V_{M+}	Motor drive +
pin 8:	V_{M-}	Motor drive −

The first three pins are used for the potentiometer, which is described in **Section 7.11**. Both the left and the right photoresistors are connected on one end to pin 5, with the other end at pin 4 (the left eye) and pin 6 (the right eye). The photoresistors are described in **Section 7.12**.

7.7 Wire

We have a lot of wire kits that contained wires of different lengths that are pre-cut and pre-stripped. Use these if you can. Try to select wires that are just the right length, so they can lie flat on the board. Messes of loopy wires are harder to debug and more likely to fall apart. Note that the wire colors in CMax match the colors of the actual wires with the same length, which should help you choose wires cleverly. If you need a longer wire, cut what you need from a spool. Use one of the pre-stripped wires for guidance on how much to strip: too little and it won't go deep enough into the breadboard; too much, and you'll have a lot of bare wire showing, risking shorts against other wires and components.

7.8 Resistors

We use quarter-watt resistors, which means that they can dissipate as much as 250 mW under normal circumstances. Dissipating more than 250 mW will cause the resistor to overheat and destroy itself.

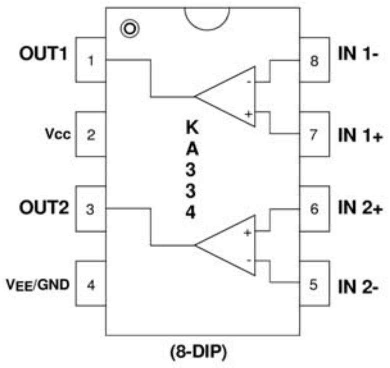
0:	black	5:	green
1:	brown	6:	blue
2:	red	7:	violet
3:	orange	8:	grey
4:	yellow	9:	white

The value of the resistance is indicated by color bands, The first two bands indicate the first two significant digits of the resistance value. The third band represents a power of 10. Thus the color bands of the resistor illustrated above (red, yellow, green) represent a resistance of 24×10^5 ohms. The fourth band represents the tolerance (accuracy) of the value, where gold represents $\pm 5\,\%$.

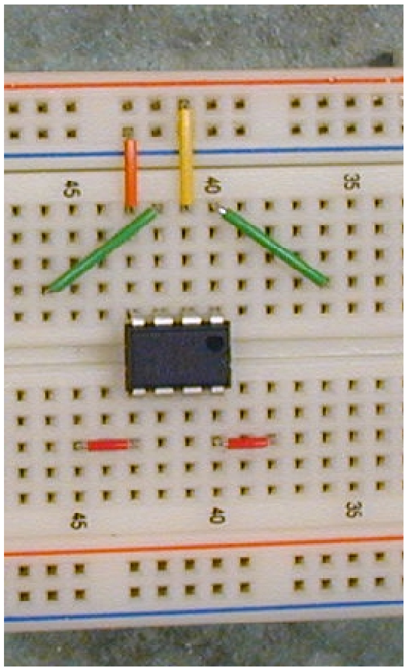
If you have trouble reading the colors, you can always measure the resistance with a meter.

7.9 Op Amps

We use op amps (KA334) that are packaged so that two op amps fit in an 8-pin (dual inline) package, as shown below.



KA334 internal block diagram (middle image) © Fairchild Semiconductor. All rights reserved. This content is excluded from our Creative Commons license. For more information, see http://ocw.mit.edu/fairuse.



The spacing of the pins is such that the package can be conveniently inserted into a breadboard as shown above (notice that a dot marks pin 1). The top center (yellow) wire in the picture above shows the connection of the op amp to the positive power supply (+10 V). The shorter (orange) wire to the left of center shows the connection to ground. The diagonal (green) wires indicate connections to the outputs of the two amplifiers, and the short horizontal (red) wires indicate connections to the two inverting (−) inputs.

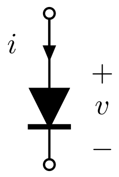
The KA334 op amps are capable of driving output currents as high as 700 mA. However, the total power dissipated by the two op amps in one package should be limited to 1 W. Exceeding the current or power limits will destroy the op amp.

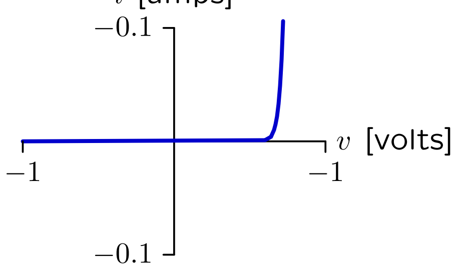
7.10 Diodes

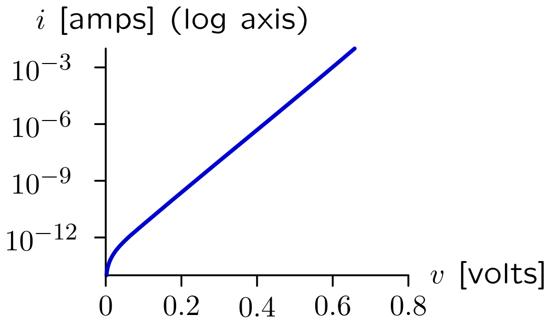
Diodes are non-linear devices: the current through a diode increases exponentially with voltage,

$$i = I_0\left(e^{\frac{v}{v_T}} - 1\right)$$

where v_T is approximately 26 mV and I_0 is approximately 0.1 pA for the diodes that we use in lab (1N914). The black bar corresponds to the negative terminal of the diode.



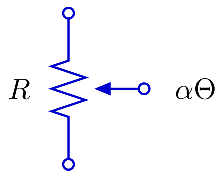


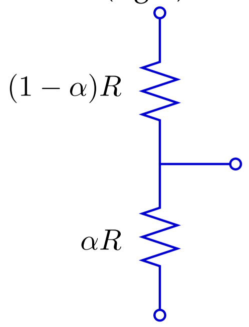


The plots show the exponential relation between voltage and current on linear axes and on semi-logarithmic axes.

7.11 Potentiometer

A potentiometer (or pot) is a three terminal device whose electrical properties depend on the angle of its mechanical shaft. The following figure shows a picture of the pot that we will use in lab (left), the electrical symbol used for a pot (center), and an equivalent circuit (right).





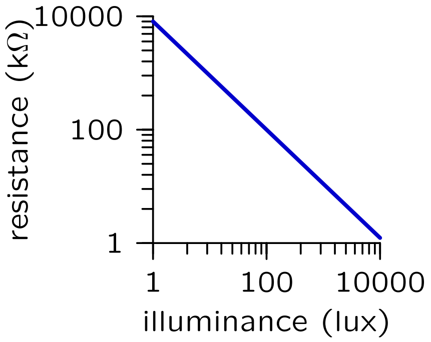
The resistance between the bottom and middle terminals increases in proportion to the angle of the input shaft (θ) and the resistance between the middle and top terminal decreases, so that the sum of the top and bottom resistors is constant. We define a proportionality constant α, which varies between 0 and 1 as the angle of the potentiometer shaft turns from 0 to its maximum angle Θ, which is approximately 270° for the potentiometers that we use in lab.

By connecting a pot as a variable resistor (using top and middle terminals), the resistance across those terminals is proportional to the angle of the shaft.

By connecting a pot as a voltage divider (top terminal to a voltage source and bottom terminal to ground), the voltage at the middle terminal is made proportional to the angle of the shaft.

7.12 Photoresistor

A photoresistor is a two terminal device whose electrical resistance depends on the intensity of light incident on its surface. A photoresistor is made from a high resistance material (e.g., CdS). Incident photons excite the electrons – liberating them from the atoms to which they are normally held tightly – so that the electrons can move freely through the material and thereby conduct current. The net effect can be characterized by plotting electrical resistance as a function of incident light intensity, as in the following plot (notice that the axes are logarithmically scaled).



Normal room lighting is between 10 and 100 lux. Illuminance near a 60 watt light bulb (as we will use in lab) can be greater than 10,000 lux.

7.13 Phototransistor

A phototransistor is a two terminal device whose current depends on the intensity of light incident on its surface. A phototransistor is a semiconductor device in which incident photons create holes and electrons. Our phototransistors are operated with their shorter lead (the “collector”) more positive than their longer lead (the “emitter”), so that the photo-generated current is multiplied by an integrated transistor (of type “NPN”).

The plastic lens that covers the phototransistor collects light from a limited cone of approximately 30° arc. The photocurrent peaks at approximately 20 mA for a very bright source. The phototransistor should be connected in series with a resistor across a constant voltage power supply, and the resistor should be chosen so that no more than 0.1 watts are dissipated in the phototransistor.

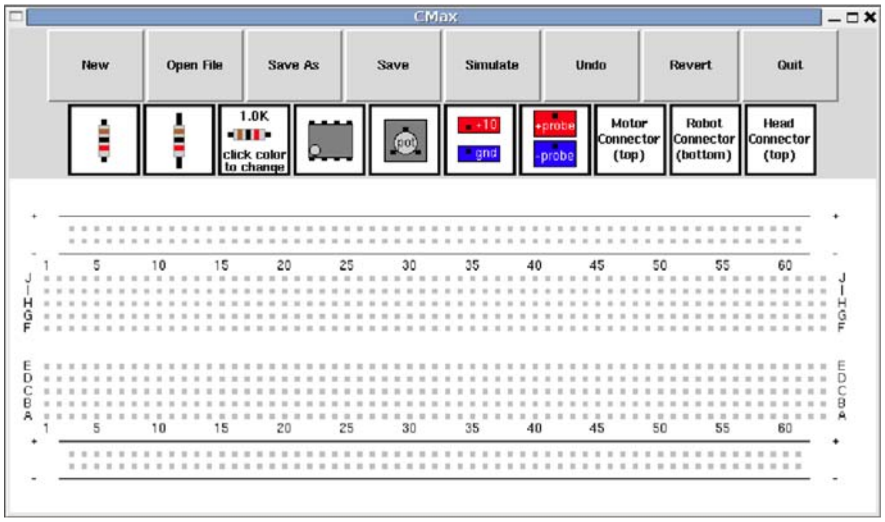
8 CMax

We will be using a simple layout and simulation tool, called *Circuits Maximus*, or “CMax”, to its friends, to design and test circuits before constructing them. You can run CMax by navigating to the directory containing the file `CMax.py` and typing

```
> python CMax.py
```

8.1 Creating and simulating layouts

When you start CMax, you'll see a window containing a row of buttons and an image of a breadboard, shown below. You can add components and wires to the board, and simulate the resulting circuit.



Adding components and wires

- You can place components on your board (resistors, op-amps, etc.) by clicking the associated button. They will appear in the lower left corner of the board, and then you can drag them to where you want them to appear on the circuit. Note that CMax allows you to place components in locations on the board that don't have any holes; this is to give you room to maneuver, but be careful not to leave any components disconnected.
- To obtain a component in a different orientation, hold down the **Shift** key when you select it from the menu.
- Resistors are rectangles with three color bands. You can change the value of the next resistor you place by clicking on the color stripes of the *prototype* resistor icon (the one with the text in it); it will cycle through the colors. The meaning of the colors is described in Section 7.8. **Shift**-clicking a resistor band will cycle through the values in the other direction.
- You can connect any two locations on the board with a wire by clicking on the first spot and dragging to the second.
- Wire ends and component leads must be at one of the gray dots that repesent a hole. Only one wire or component lead can occupy a hole.

- You must connect your board to power and ground by adding the `+10` and `gnd` components to it. You must have exactly one of each.
- You can add connections to a **motor** using pins 5 and 6 on a *motor connector* as described **Section 7.4**.
- A *head connector* connects to a motor on pins 7 and 8. The neck potentiometer, also on the head connector, is on pins 1, 2, and 3 (with pin 2 being the wiper).
- A *robot connector* supplies the connections described **Section 7.5**. If you are using a robot connector for power, do not use separate power and ground connections.

Modifying your circuit

- You can delete a component or wire by holding down the control key (you see a skull/crossbones cursor) and then clicking on the body of the component or on a wire.
- You can move the endpoint of a wire by clicking and dragging it; you can move a whole wire by dragging the middle.
- Moves of components and component creation can be undone using **Undo**. The undo operation is only one level deep, so hitting **Undo** again will re-do the operation.
- To read the value of a resistor in your layout (in case you forget what the color bands mean), **shift**-click the resistor. The value of that resistor will be shown in the prototype resistor button.

File management

- The **Quit**, **Save**, **Save As**, **New** and **Open File** commands should do what you expect. Make sure that the files you create to save your circuits have a `.txt` extension. The **Revert** button will erase the changes you have made since the last time you saved the file.

Running tests

There are several ways to see what happens when you run your circuit. The **Simulate** button will run your circuit; it needs to use an input file that specifies time sequences of inputs to the potentiometers in your circuit. You won't ever need to write an input file; we will specify them for you, to run particular tests.

When you click **Simulate** the first time, you pick a test file. It will use the same test file thereafter. If you **Shift**-click **Simulate**, it will re-prompt you for a test file, so you can select a different one.

- You can measure the voltage between two points in your circuit by placing a `+probe` and a `-probe` (exactly one of each) in your circuit, hitting the **Simulate** button, and selecting the file `noInput.py`; it will print the voltage across the probed locations in the window from which you started Python. If there is a component with temporal dynamics (a potentiometer or a motor) in your circuit, then when you simulate, it will also pop up a window showing the signal generated at the probe.
- If there is a *motor* in your circuit, when you hit **Simulate**, a window will pop up that shows a time sequence of the motor's speed, in radians per second.
- If you want to see how your circuit behaves as a function of an input signal, you can add a *potentiometer*. If there is a potentiometer in your circuit, when you hit **Simulate**, a window will pop up that shows a time sequence of the potentiometer alpha values, so you can see what the input is that your circuit is reacting to.

8.2 Keyboard shortcuts

There are several keyboard shortcuts:

- ′n′: Clear button
- ′o′: Open file button
- ′p′: Revert file button
- ′q′: Quit button
- ′r′: Simulate (run) button
- ′s′: Save button
- ′u′: Undo button

8.3 Debugging

Here are some common problems:

- `Failed to solve equations! Check for short circuit or redundant wire` This can be caused by connecting power to ground, for example. Examine your wiring. Maybe you inadvertently used the same column of holes for two purposes. At worst, you can systematically remove wires until the problem goes away, and that will tell you what the problem was.
- `Element ['Wire', 'b47', 'b41'] not connected to anything at node b41` The name `'b41'` stands for the **b**ottom group of five holes, in column 41. If you get a message like this, check to see what that element should have been connected to. You know that there should be something else plugged into the bottom section of column 41, in this case.
- `Illegal pin` means that you have a wire or component that has an end or a pin in position on the board that does not have a hole.