ME 599/EE546: Biology-Inspired Robotics

University of Washington, Autumn 2017, Prof. Sawyer B. Fuller

Problem Set 1

The idea of this problem set is to promote good software development practices for simulation code, and to explore dynamics and control simulations. The skeleton code provided here is written in MATLAB (see below for how to download and a basic tutorial). You may also write software in another programming language, e.g. Python, but that will require re-implementing the skeleton code that is provided with this problem set. You are encouraged to work together to work out solutions, but you must submit your own work.

1. Please read Braitenburg1984, chapters 1–4, available under the `papers` section of the course website. You will be simulating the behavior of some of the vehicles he describes. For our case, imagine your vehicle is moving on flat ground, such as a tabletop.

2. You will first simulate vehicle 1, which follows a straight line. Let's suppose its sensor is responding in proportion to brightness of the light as detected by the sensor (rather than heat as suggested by the book). To implement this in simulation, we introduce the notion of state, which are all of the parameters of the robot that evolve in time. For this robot, we will use a state consisting of its position and orientation, $\boldsymbol{q} = [x,\, y,\, \theta]^T$.

The wheels are connected in such a way that they spin with a speed that is proportional to the sensor reading. If the robot's speed is given by v, then the state evolves according to

$$\dot{\boldsymbol{q}} = \begin{bmatrix} v\sin\theta \\ v\cos\theta \\ 0 \end{bmatrix}. \tag{1}$$

This is a specific example of a more general dynamical system formulation, $\dot{\boldsymbol{q}} = \boldsymbol{f}(\boldsymbol{q}, \boldsymbol{u})$, where $\boldsymbol{u}$ is some input to the system (such as wheel velocity). Numerical simulation consists of iteratively computing, for each time instant k, how the state has changed over the time increment T according to

$$\boldsymbol{q}_k = \boldsymbol{q}_{k-1} + T\dot{\boldsymbol{q}}_k.$$

Skeleton code to simulate a robot performing these dynamics is provided online with the problem set in `braitenburg_1.m`. The matlab function `robot_dynamics` in that file computes $\dot{\boldsymbol{q}}$ (Equation 1). Please update it to include these dynamics. Next, update the matlab function `compute_light_response` so that the vehicle receives feedback in proportion to the distance to the light source. Submit your updated software code and an image of the resulting motion of the vehicle.

3. Next, you will implement vehicle 2. For these robots, you must compute distances to the light source for two sensors, each placed in a different location on the robot. Note that your code must compute where the light sensor is by computing where the vehicle is and how it is oriented (by using its state). You have some flexibility to decide where on the vehicle the light sensors are (front/middle/back) - but the should be on both sides somewhere. Skeleton code is in `braitenburg_2.m`. Submit your software code and plots showing light following and light avoiding maneuvers (2a and 2b).

4. Last, we will make a robot that uses *memory* about previous sensor readings to perform maneuvers. Add two additional states to your robot's state vector $\boldsymbol{q}$ that encode the time integral of light intensity of the two light sensors. Use this to create a robot that stops once it has spent enough time in the vicinity of a light source by detecting when this time integral has reached a threshold. Submit your code and a representative figure.

Notes on MATLAB

MATLAB is a programming language designed to make numerical computation as simple as possible. It is available through the Mechanical Engineering Department's remote desktop server (more information at `https://www.me.washington.edu/computing.html`) or for purchase for a fee at U. Washington's U-Ware site. The following tutorial will get you up to speed or serve as a refresher: `http://faculty.washington.edu/minster/bio_inspired_robotics/files/matlab_tutorial.pdf`

This problem set uses two additional aspects of MATLAB that are not covered in the tutorial:

Structures These are convenient for combining number of different values (such as the parameters of our vehicle) into a single variable. They are different from arrays because each field has a name. To create a structure, assign to one of its fields:

```
mystruct.field1 = 22;
mystruct.asdf = [1, 2, 3];
```

You can use it in an expression like any other variable:

```
>> mystruct.field1 * 3

ans =
66
```

Functions These allow you to encapsulate functionality, making them good programming practice. They are defined as follows:

```
function return_val = myfunction(argument1, argument2)
   blah = 22;
   % return_val must be set somewhere inside the function:
   return_val = blah * argument1 * argument2;
end
```

There can be any number of arguments in your definition.

Calling a function you write is like calling any function:

```
>> myval = myfunction(1, 5)

ans =
110
```

You can get more help on these and other MATLAB features by using the "help" command:

```
>> help function
```

Note that MATLAB distinguishes between two types of `.m` files: scripts and function files. A file is a function file if its first line is a MATLAB function definition. If you want to define additional utility functions within a file, as is the case for the skeleton code provided with this problem set, you can only do so within function files. Scripts are therefore typically used to set initial conditions and then call MATLAB functions.

One additional point: after the code inside a function is finished executing, the variables defined inside the function are no longer available in the workspace. If you want to look at variables inside the function, click on the dash beside a line number to create a red circle "break point". Next time you execute, MATLAB will stop at that line number 24 ● ➪ and give you a prompt `K>>` that you can use to run simple commands like evaluating a variable. When you are done, stop the debugger by clicking ■ in the toolbar.