FFR 125, Autonomous agents, 2016: Home problem 2

General instructions. READ CAREFULLY!

Home problem 2 consists of four parts. Problems 2.1 and 2.3 are mandatory, problems 2.2 and 2.4 are voluntary (but check the requirements for the various grades on the web page). **Note!** Before submitting your solutions and your report, make sure to download and *read carefully* the checklist for home problem submission, available at

`www.me.chalmers.se/~mwahde/courses/aa/2016/Checklist_HPSubmission.pdf`

When programming, you should use Matlab. Make sure to follow the Matlab coding standard, available at

`www.me.chalmers.se/~mwahde/courses/aa/2016/MatlabCodingStandard.pdf`

During correction of the home problems, it will be assumed that you have read and followed the information in the two documents listed above. If you have any questions regarding the coding standard or the submission procedure, please ask *before* submitting your solutions.

You should provide a single report (for the whole problem set) in the form of a PDF file. In the case of analytical problems, make sure to include *all the relevant steps of the calculation* in your report, so that the calculations can be followed. Providing only the answer is *not* sufficient. Whenever possible, use symbolical calculations as far as possible, and introduce numerical values only when needed. You should write the report on a computer, preferably using LaTeX (see `www.miktex.org`). Scanned (or photographed) handwritten pages are *not* allowed.

You may, of course, discuss the problems with other students. However, each student *must* hand in his or her *own* solution. In obvious cases of plagiarism, points will be deducted from all students involved.

Make sure to send the solutions and your report before the (strict) deadline. If the solutions are submitted after the deadline, only the mandatory problems (2.1 and 2.3) will be corrected. The submission time will be taken as the time when your last submission e-mail is *received*. Thus, make sure to include all solutions (as a single zip file, which should contain your report and the Matlab code) when sending the e-mail.

Make sure to have some (time) margin when submitting the solutions. Submitting a few seconds before midnight on the 7th *is not recommended.*

Deadline: 20160307, 23.59.59

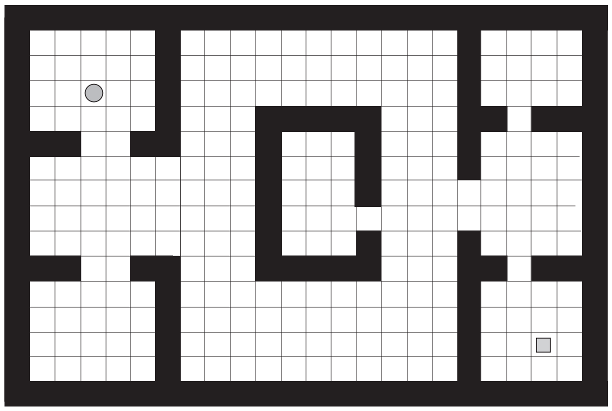

Figure 1: The arena used in Problem 2.1.

Problem 2.1, 4p, Path generation using Dijkstra's algorithm

Dijkstra's algorithm, described in Chapter 6 is an example of a grid-based navigation algorithm, which is guaranteed to find the shortest path to the navigation goal, albeit not very efficiently. In this problem, you will implement Dijkstra's algorithm (in Matlab), and apply it to find the shortest path between two rooms in an office arena, shown in Fig. 1.

Write a Matlab program that allows the user to set a start position and a navigation goal. The start node (grid cell) should be indicated in yellow, and the target node in blue. Obstacles should be shown in black, whereas cells that do not contain obstacles (and are neither start or target nodes) should (initially) be gray. When the program is running, the image should be updated for each considered node, such that the node under consideration is shown in white. Nodes that have been completed (i.e. considered during evaluation) should be shown in light green. When the robot reaches the navigation goal, it should trace (in green) the path from the goal to the starting point, as shown in Fig. 6.10 in Chapter 6, and in the video `Dijkstra.avi` available on the course web page. Thus, it is not sufficient only to find the navigation goal: Your program should also generate the shortest path, using the concept of predecessor nodes, described on p. 78 in Chapter 6.

Note: For any given current node, only the four neighbors in the coordinate directions should be considered. Diagonal movements are thus not allowed.

Hint: In order to display a grid, with various colors for different cells as described above, you should use the Matlab command `image`, using a customized color map containing the colors that you need (i.e. yellow, blue, white, black, gray, light green and green). A usage example can be found the file `ColorGrid.m` on the web page. The (integer) value assigned to any grid cell will depend on the cell's status (e.g. obstacle, target etc.).

Next, implement the arena (size 24×16 grid cells) shown in Fig. 1, and use the program to generate the shortest path between the starting node (shown as a disc in the figure) and the target node (shown as a gray square in the figure).

Problem 2.2, 2p, Simulation of E. Coli kinesis

Implement *E. Coli* kinesis as described in Chapter 4. Let each bacterium have two behaviors, namely *straight-line motion* (B1) in a given direction, and *tumbling* (B2). In B1, the bacterium should simply take a step (length = $0.005L$, where L is the side length of the arena) in the current direction of heading and, in B2, the bacterium should first determine a new, random heading (in the range $[0, 2\pi]$), and *then* take a step (of the same length as the steps taken in B1). The selection of behaviors should be based on the variable U given by $U = X - V$, where

$$\frac{\mathrm{d}V}{\mathrm{d}t} = -c_1 V + c_2 X, \tag{1}$$

where X is the concentration of the attractant, and c_1 and c_2 are non-negative constants. If $U > T$, B1 should be active, otherwise B2 should be active (T is a constant threshold parameter). You should start from the functions given in the zipped folder `EColiSimulator.zip`, available on the course web page. You need to implement two functions, namely

```
Bacteria = UpdateUtility(Bacteria,Arena,dt)
```

and

```
Bacteria = MoveBacteria(Bacteria,Arena,dt)
```

(see the `RunEColi.m` file). Put both functions in the `BacteriaFunctions` folder. Note that the Matlab `struct` concept is used in the program. Check the online documentation of Matlab for more information.

A time step `dt` of 0.01 s should be used when discretizing and integrating equation (1) (use a simple Euler integrator). When moving bacteria, use a step length of $0.005L$, see above. Apply periodic boundary conditions, so that a bacterium leaving the arena on one side appears on the opposite side. Assume further that the attractant concentration varies as

$$X = c_0 \mathrm{e}^{-k(x^2+y^2)}, \tag{2}$$

where $c_0 = 1$ and $k = 2$ are constants. Next, determine parameter values c_1, c_2 and T such that at least 95% of the bacteria will be located in a region of radius $r = 1.5$ after 1,000 time steps, assuming that the initial distribution is uniformly random (see `CreateBacteria`) in a square of side length $L = 6$. Note that the values of c_1 and c_2 are set in the function `CreateBacteria`, whereas T is set in `RunEColi.m`. Make sure to submit the entire program (all files) so that it can be tested without any editing.

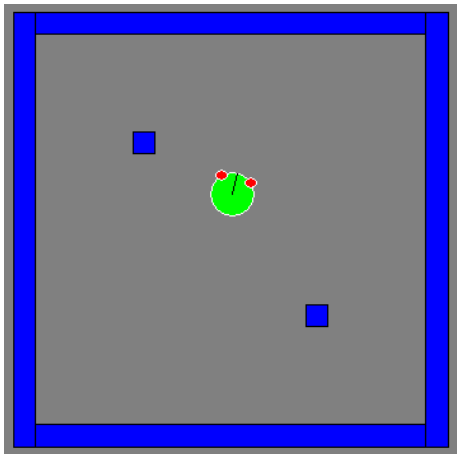

Figure 2: The arena considered in Problem 2.3.

Problem 2.3, 4p, Potential field navigation

In this problem, we will consider navigation by means of the potential field navigation method, assuming perfect odometry (i.e. no odometric drift). Consider the arena shown in Fig. 2, which is implemented in ARSim available at

`www.me.chalmers.se/~mwahde/courses/aa/2016/ARSim_HP2.3.zip`

Starting from this version of ARSim, generate a potential field for the arena, by introducing potentials (one or several) for each of the four walls and for the two obstacles in the arena. Make sure to set the parameters in such a way that the robot will avoid collisions. Also, introduce a goal potential, centered on the navigation goal $(x_{\mathrm{g}}, y_{\mathrm{g}})$. The potential field should be set up in the `CreateBrain` function. Thus, to modify the goal position, one must edit this file. Therefore, make sure that the two parameters defining the navigation goal are clearly and easily identifiable in `CreateBrain`. Do *not* use a grid to interpolate the potential field: Instead, compute the potential gradient exactly.

Next, in the `BrainStep` function, write a motor process (behavior) for potential field navigation, in which the potential field provides the desired direction of motion (φ_{ref}). The desired speed should be set as described on pp. 84-85 in Chapter 6. The parameters (e.g. V_{nav}) should be specified in `CreateBrain`. In addition to the normal navigation state, you should also introduce a rotation state (as described on pp. 84-85) which should be activated whenever the difference between the estimated and desired direction of heading exceeds a certain threshold $\Delta\varphi_{\max}$ (the value of which should also be set in `CreateBrain`). The estimated direction of heading (φ_{est}) should be obtained from the robot's odometer, which is here assumed to be noise free.

Note: Your program should be able to handle any pair of angles φ_{ref} and φ_{est} correctly, so that the robot never spins around unnecessarily. For example, if φ_{ref} is equal to -3 radians (say) and φ_{est} is equal to 3 radians, their difference is $2\pi - 3 - 3 \approx 0.283$, *not* 6!

Finally, use the implemented behavior to make the robot navigate from an *arbitrary* starting pose (specified in `RunRobot`) to an *arbitrary* goal position (set in `CreateBrain`). Note: (1) Your robot will be tested using a variety of different starting poses and goal positions; (2) You should only submit the files `RunRobot.m`, `CreateBrain.m` and `BrainStep.m`. All other submitted Matlab files will be ignored. In addition, you should, of course, give a clear description of the implemented brain process in your report.

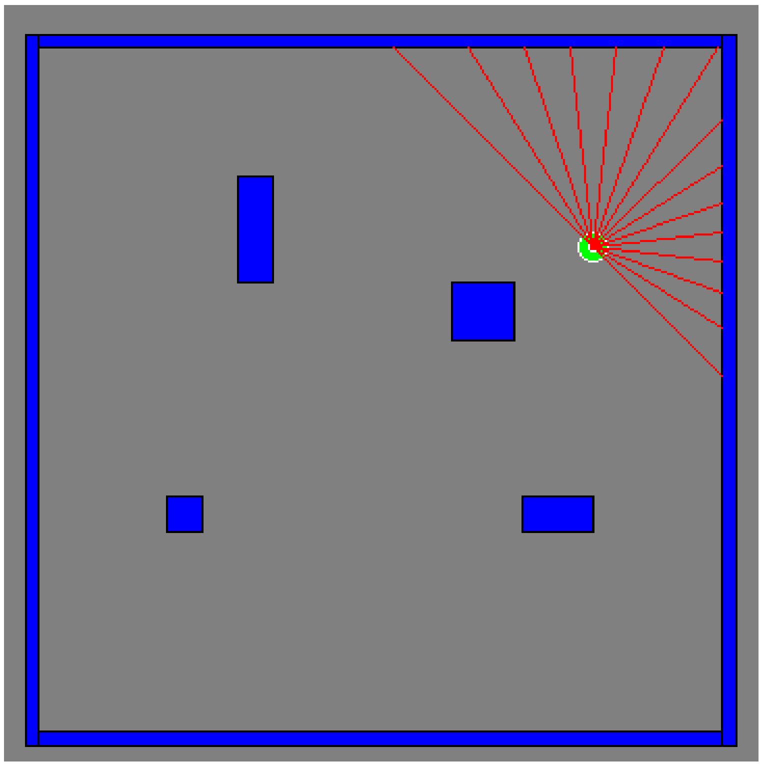

Figure 3: The arena considered in Problem 2.4. Note the 15 rays of the robot's LRF.

Problem 2.4, 5p, Robot localization

In this problem, we shall attempt to carry out robot localization using a simplified laser range finder (LRF) with only a few (15) rays. Starting from the version of ARSim available at

`www.me.chalmers.se/~mwahde/courses/aa/2016/ARSim_HP2.4.zip`

implement a suitable potential field for navigation in the arena. You may, of course, use your results from Problem 2.3. However, you should place the potential field navigation in its own struct (i.e. as a separate brain process). Similarly, you should implement a brain process called localization (described below), which should also be placed in its own struct. The idea will be to navigate (with the potential field method) using (noisy) odometry, which, from time to time, should be recalibrated by activating the localization brain process.

The two brain processes should contain a Boolean field `Active` that determines whether or not the brain process is active. Only one brain process should be active at a time.

The activation of the localization brain process can, for simplicity, be based on the time since its most recent de-activation, but you may use also other criteria for triggering localization.

You should write a decision-making structure (in `BrainStep`), which determines when to run the localization brain process and when to run potential field navigation. You may define utility functions for the brain processes, but this is not a requirement. However, regardless of which decision-making method you implement, you should give a *clear* description of that method in your report.

The localization brain process should work as follows: First the robot should stop. It may then rotate, to position itself in the most favorable position for localization. Next, it should obtain the readings of its LRF (see below), and match those to the virtual readings obtainable in a map (which you should also implement in the robotic brain) of the arena. For localization, the actual readings of the LRF should be compared with the expected readings at the given estimated pose (obtained from odometry), as described in Chapter 6. If the difference is too large (compared to a user-specified threshold; see Sect. 6.3), the robot should execute a search, as described on pp. 89-90 in Chapter 6). Once a new pose estimate has been obtained, the odometry should be recalibrated. (For simplicity, the LRF has been placed such that its coordinates coincide with those of the robot.) Then localization should be de-activated, and potential field navigation should resume.

Note that a somewhat ugly modification has been made to ARSim, in order to simulate the LRF, so make sure that you start from the version of ARSim given on the web link above. In this case (and *only* in this case), the `Reading` field of the LRF should not be used. Instead, the ray readings *should* be used since, for an LRF, they do represent an actual, physical quantity, namely the distance to the nearest obstacle along the ray. In order to obtain the reading of ray i, write

```
rayReading = testRobot.RaybasedSensors(1).RayLengths(i)
```

etc. Note that the robot has only one sensor in this problem. If the distance along a given ray exceeds the range of the LRF, the corresponding reading will be -1.

With this robotic brain, the robot should be capable of navigating from any initial pose to any goal point in the arena shown in Fig. 3. Note that the odometric drift (the level of which you may *not* modify!) is not very large so, in some cases, the robot may be able to reach its goal without carrying out localization. However, since the main purpose of this problem is to implement localization, your robot *must* make use of this brain process during its motion.

As usual, you should describe the robotic brain clearly in your report, and you should submit `CreateBrain.m`, `BrainStep.m` and `RunRobot.m`, as well as any added Matlab files (containing, for example, the brain processes).

Hint: The map of the arena may be represented as a sequence of lines. Each line, in turn, can be defined using four numbers (x_0, y_0, x_1, y_1). You need only include those lines that can be seen from inside the arena. In order to obtain the reading of the virtual LRF, you may wish to start from the function `GetDistanceToLineAlongRay` etc.