CS100M Fall 2005 Project 2 due Thursday 9/22 at 6pm

Submit your files on-line in CMS before the project deadline. See the CMS link on the course web page for instructions on using CMS. Both correctness and good programming style contribute to your project score.

You must work either on your own or with one partner. You may discuss background issues and general solution strategies with others, but the project you submit must be the work of just you (and your partner). If you work with a partner, you and your partner must register as a group in CMS and submit your work as a group.

Objectives

Completing this project will help you learn about loops, selection statements, and developing algorithms. The first part applies the "brute-force" algorithm for finding a function minimum to an environmental modeling problem. The second part is a game that requires you to develop an algorithm for traversing a maze. Compared to Project 1, this project is more open-ended and requires you to sort through given information that may or may not be directly used in your programs. Any constants that you use should be named (as variables) and defined in your programs, and any assumptions you make should be stated using comments.

1 Dissolved Oxygen in Rivers

One indicator of the well being of rivers and streams is the concentration of dissolved oxygen (DO) in the water. The US Environmental Protection Agency determines DO guidelines that are appropriate for different uses of the streams. For example, recreational fisheries require DO concentrations above 8 mg/l. Rivers have a "cleaning capacity" in that they consume pollutants through biological processes. These biological processes deplete DO, but fortunately for all living things the level of DO recovers naturally in a process called reaeration!

The depletion and reaeration of DO can be modeled by the "DO sag equation" of Streeter and Phelps:

$$c(t) = c_s - \frac{L_0 K_d}{K_a - K_r}[\exp(-K_r t) - \exp(-K_a t)] - (c_s - c_0)\exp(-K_a t) \quad (1)$$

where
- t is **t**ime (independent variable),
- $c(t)$ is the DO **c**oncentration at time t,
- c_s is the DO **s**aturation concentration (maximum possible DO given temperature and pressure),
- c_0 is the initial DO concentration,
- L_0 is the pollutant **L**oad into a river,
- K_d is the rate of **d**ecay of pollutants through biological processes,
- K_r is the rate of **r**emoval of pollutants through biological processes and settling, and
- K_a is the rate of re**a**eration.

Write a MATLAB script `oxygen.m` to determine (and print)

1. the minimum DO concentration and when it occurs
2. how many days it will take for the river's DO level to recover to its initial state given a pollutant load L_0.

The river data and model parameter values are as follows: $L_0 = 30$ mg/l, $c_s = 8$ mg/l, $c_0 = 7$ mg/l, $K_d = 0.35$/day, $K_r = 0.35$/day, $K_a = 0.8$/day.

Figure 1 shows the shape of a typical DO sag curve. From the curve and from its equation, you can tell that you can actually determine its minimum value analytically using Calculus! *However*, you will use the "brute-force" algorithm (discussed in class) for finding the function minimum in this project. Similarly, use the brute-force framework to find the DO recovery time—do not solve for the solution analytically. What should be the bounds for the independent variable t? The left bound could reasonably be set to zero; the right bound you can estimate by trial-and-error. (Remember that you need to find the recovery time (in days) as well as the minimum DO level (in mg/l).) How about the step size Δt? Reasonable values for such a water quality model would range from 1 to 5 hours. You also need to ask yourself how close is close enough in finding the DO recovery time. Use your best (scientific) judgment.

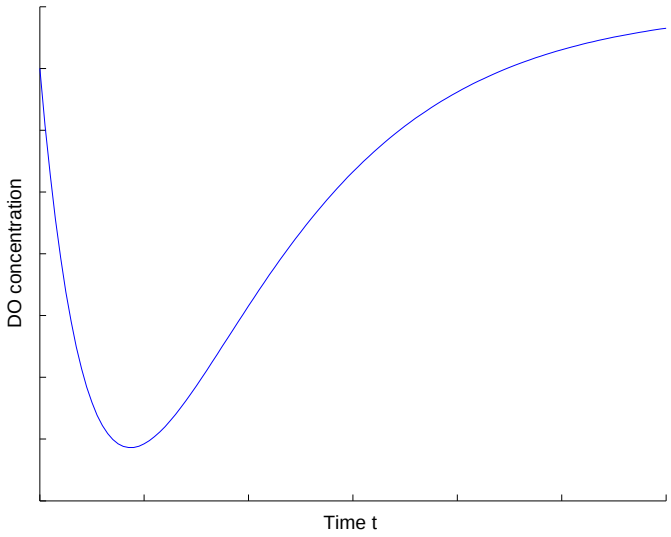


Figure 1: DO sag curve

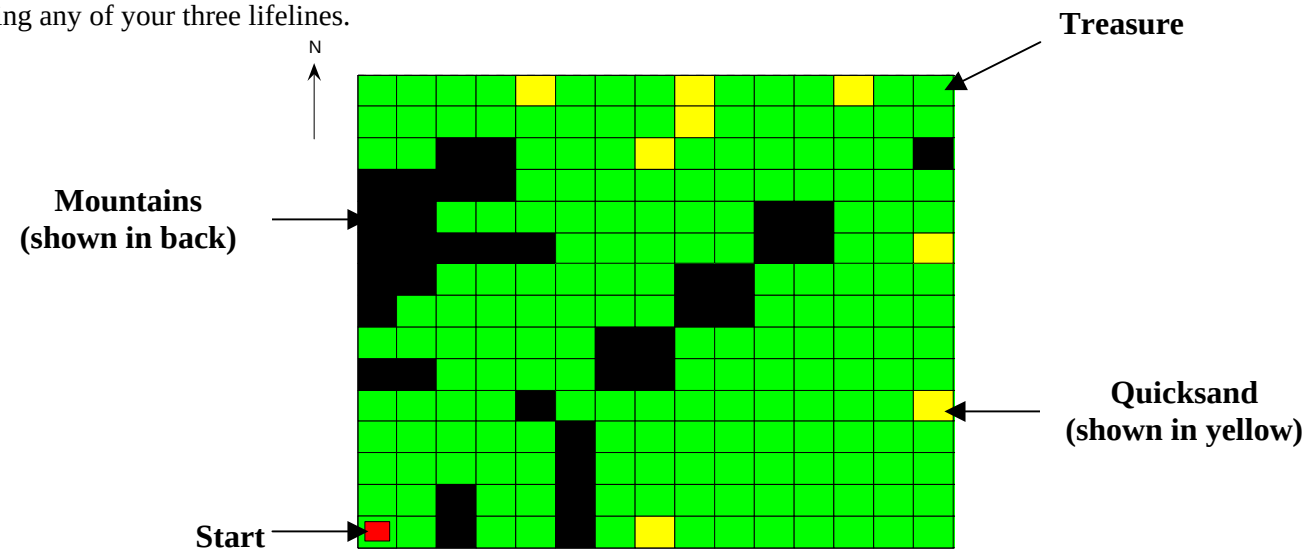
Aside

In the U.S., elaborate (and realistic?) water quality simulation models that take into consideration the complex geography, biochemical and physical processes, and time-variable nature of pollution have been developed by government agencies, universities, and engineering consultants for many major waterways. While the Streeter-Phelps equation is central to many of these models, it is almost always just one component of a much larger model that answers the two questions (minimum water quality, recovery time) we ask above. So when is a direct application of the Streeter-Phelps equation, as we have done above, useful? Think about the tragic aftermath of hurricane Katrina. In the field, engineers do not have the time or the capability to take all the measurements that would feed a complex simulation model. Furthermore, some established models no longer apply because flooding has caused extreme changes to the flow characteristics of the waterways. Then there are the new waterways created by the flood for which no prior model exists. In the field, engineers and scientists do quick "back of the envelope" calculations based on estimated parameter values, which may amount to a short program using the Streeter-Phelps equation as you have developed.

One last thought: did you find the use of *time* as the independent variable curious? Don't people usually worry about *where* the worst water quality will occur instead of when? We often use time for quick calculations because one can then translate "x days" into "y miles downstream" given the flow velocities [distance/time] of the different waterways of concern.

2 Treasure Hunt

On a secluded island in the Caribbean lies a treasure far beyond your imagination. You will write a MATLAB script **treasureHunt.m** to walk through the maze of mountains and quicksand to find the treasure. The island is surrounded by water on all the sides. You will start at position (1,1) and will try to get to position (15,15) as shown on the diagram below. There are mountains that you will have to walk around (cannot climb or walk through) and quicksand pits that you will want to avoid. You are given three lifelines that you can use. Any time you step into a quicksand pit, you will automatically lose a lifeline. The objective of this game is to reach the treasure without losing any of your three lifelines.



You can use only three sensors and three kinds of motions in this treasure hunt, and you cannot swim! The sensors are coded in the following functions:

- **obstacleInFront()**
  - Returns 1 if there is a mountain, 2 if there is quicksand, or 3 if there is water in front of you; returns 0 if it is an open space in front.
- **obstacleOnLeft()**
  - Returns 1 if there is a mountain, 2 if there is quicksand, or 3 if there is water immediately to your left; returns 0 if there is an open space to the left.
- **inQuicksand(xTemp, yTemp)**
  - Returns 1 if position (**xTemp**, **yTemp**) is a quicksand pit. Otherwise returns 0.

Your possible motions are coded in the following functions:

- **moveForward ()**
  - Moves you forward one-step (square)—your position (stored in variables **xCoordinate** and **yCoordinate**) is updated after this function executes. This function reduces your number of lifelines (variable **lifelines**) by 1 if you step into quicksand by moving forward.
- **turnRight()**
  - You turn to your right—the direction in which you face (stored in the variable **direction**) is updated by this function.
- **turnLeft()**
  - You turn to your left—the direction in which you face (stored in the variable **direction**) is updated by this function.

Read the comments and code for the above functions *except* for function **moveForward()**. You should understand the *selection statements* in these functions, but do not worry about the other code in the functions.

Design an algorithm to traverse the island and implement your algorithm in the script **treasureHunt.m**. Remember that you have only three lifelines—the game is over once you use up your lifelines. Write your code only in the marked area of the file.

You may use the above functions in your code, although you may not need all of them. *Do not use a trial-and-error algorithm* where at each step (square) you simply iterate through the possible step/directions in some (random) order. Although a trial-and-error approach will eventually take you to the treasure, it is inefficient and shows poor design. Your algorithm is the *decision logic* to be followed at each step. Your algorithm must be general, i.e., you should be able to traverse a different map that begins at position (1,1) and ends at position (15,15). (We will test your programs on other maps that are traversable.) Use comments as appropriate to clarify your algorithm! Be concise!

Download the file **p2files.zip**, which contains all the required M-files (**treasureHunt.m**, **obstacleInFront.m**, **obstacleOnLeft.m**, **inQuicksand.m**, **moveForward.m**, **turnRight.m**, **turnLeft.m**, **loadMap1.m, loadMap2.m, drawMap.m**. You will write code in and submit the file **treasureHunt.m**. *Read the hints below before you start your work!*

Hints:

- Read the skeleton code that we have provided in **treasureHunt.m**. We have provided the code to draw the map and the moves that you make—at this time you do not have to learn how to draw the diagrams. Note that *we have given you several statements to demonstrate some of the sensor and motion functions* that you may need to use. Read this demonstration code and run the program to see how they work. Next, add more code to our demonstration code to make an "illegal" move, e.g., walk into the mountain. You will then see that the program terminates in error and displays an error message. Remember to delete our demonstration code when you write your own.
- Can't think of an algorithm? On paper, draw the first few moves that you must make. At each move, ask yourself *what are the conditions that lead you to make that particular move*.
- Guard against infinite loops! (If it happens, go to the Command Window and type **<Ctrl>c** to stop the program.)
- Make sure that you are not making the same (bad) move over and over again.
- If you want to speed up the graphics, change the argument to the function **pause** in **treasureHunt.m**. Type **help pause** in the Command Window to find out how **pause** works.
- You are given two different mazes. Test your algorithm on both of these. The **loadMap1** function is called within **treasureHunt.m**; you can change it to **loadMap2** to use the second maze instead. Your project will be graded on other maps where a solution is possible.

Final Words:

In this question, we have done a lot of work for you, including the very important task of ***decomposing*** the problem! For example, each move that the treasure hunter makes is a subtask that must be accomplished, and accomplished repeatedly, in order to solve the problem. We have completed these subtasks for you so that you don't have to worry about the detail. The overall maze problem is big and complicated, but by decomposing it we get individual subtasks that are small. The result? You, who are relatively new to programming, get to work on a (part of a) large (and we hope fun) project! The lesson? ***Decomposition rules!*** The benefit of decomposition is augmented by making the entire program *modular*: each subtask is separated out as a module (function) that can be written by a different programmer in a project team, for example, making collaboration easy. Our next topic in class is about developing modular programs, i.e., writing *user-defined functions*.