COSC1076
Advanced Programming Techniques

Assignment 1
Particle Filter

Weight: 15% of the final course mark
Due Date: 11.59 pm, Friday 5 April 2019 (Week 5)
Learning Outcomes: This assignment contributes to CLOs: 1, 2, 3, 4, 6

Change Log

1.3
- Added submission instructions
- Updated code description requirements for Milestone 4
- Minor corrections in Background

1.2
- Updated references to classes
- Updated notes on Unit Tests to be clearer about the purpose of the sample unit tests.
- Updated parts of the Task descriptions to be clearer
- Added Section "Getting Started"

1.1
- Corrected positions on the worked example.
- Updated compilation command, removing `-c` from full code command.

1.0
- Initial Release

Contents

1 Introduction

1.1 Summary

In this assignment you will implement an algorithm known as a ***Particle Filter***, and use it with a simple simulated 2D robot moving around a room.

In this assignment you will:

- Practice the programming skills covered throughout this course, such as:
  - Pointers
  - Dynamic Memory Management
  - Provided API
- Correctly Implement a pre-defined API
- Implement a medium size C++ program
- Use a prescribed set of C++11/14 language features

This assignment is divided into four Milestones:

- **Milestone 1:** Writing Unit Tests
- **Milestone 2:** Minimum required component for implementing the simple particle filter
- **Milestone 3:** Optional component to extend the particle filter for map rotations
- **Milestone 4:** Optional component for efficient memory management

1.2 Relevant Lecture/Lab Material

To complete this assignment, you will requires skills and knowledge from lecture and lab material for Weeks 2 to 4 (inclusive). You may find that you will be unable to complete some of the activities until you have completed the relevant lab work. However, you will be able to commence work on some sections. Thus, do the work you can initially, and continue to build in new features as you learn the relevant skills.

1.3 Start-up Code

On Canvas you will find starter code to help you get running with the assignment. This code includes:

- Header files, of the API (C++ classes) that you will implement
- Dummy (empty) code files, for you to implement
- Unit Testing code, to help you write and run tests on your implementation
- Example Unit Test cases

1.4 Plagiarism

Plagiarism is a very serious offence.

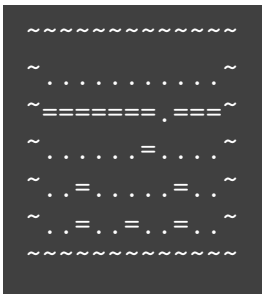
A core learning outcomes for this course is for you to: *Demonstrate and Adhere to the standards and practice of Professionalism and Ethics, such as described in the ACS Core Body of Knowledge (CBOK) for ICT Professionals.*

The penalty for plagiarised assignments include zero marks for that assignment, or failure for this course. Please keep in mind that RMIT University uses plagiarism detection software to detect plagiarism and that all assignments will be tested using this software. See the RMIT website for more information about the university policies on Plagiarism and Academic Misconduct.

2 Background

One challenge in robotics is called *localisation*. This is the process of the robot figuring out where it is within some environment. One algorithm for localising a robot is called a ***Particle Filter***. In this assignment you will implement a simple particle filter for a robot moving about a simple 2D maze.

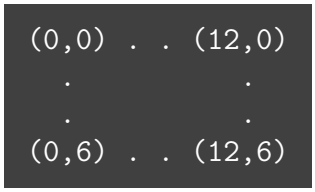
In this assignment, the simple 2D maze will be represented as a grid of ASCII characters. For example:



Aspects of the maze are represented by different symbols:

Symbol	Meaning
. (dot)	Empty/Open Space. The robot can enter any open space.
= (equal)	Wall or Obstacle within the maze. The robot cannot pass obstacles
~ (tilda)	The edge of the maze. Every maze is always bounded by the edge symbols

Each location in the maze (including the maze edges) is indexed by a cartesian `(x,y)` co-ordinate. The top-left corner of the maze is always the co-ordinate `(0,0)`, the x-coordinate increases right-wards, and the y-coordinate increases down-wards. For the above maze, the four corners have the following co-ordinates:



For the purposes of this assignment we will make two assumptions:

1. The robot know the map of the whole maze
2. While in the maze, the robot can only "see" one location about itself, represented as a 3x3 grid. For example, if the robot is at location `(2,1)` in the above maze, it can "see":

In the robot's view 3x3, the robot is always in the centre. The robot is also "facing" (or is orientated) in one of four directions This is represented by one of four characters:

Symbol	Meaning
<	Robot is facing LEFT
>	Robot is facing RIGHT
^	Robot is facing UP
v	Robot is facing DOWN

Thus, the position of the robot within the map is described as a 3-tuple, of its `(x,y)` co-ordinate and its *orientation*. For the above example, the robot's position is `(2,1,up)`.

For this assignment, the robot can move about the maze using one of three actions:

1. Rotate clockwise 90°
2. Rotate counter-clockwise 90°
3. Move one space in the direction it is facing

! Note that the robot can never remain stationary! It must always either rotate or move one space. This is important!

For example, if the robot started at position `(2,1,up)`, then rotated counter-clockwise and moved one space forward to be at position `(1,1,left)`, it would see:

However, if the robot had rotated clockwise and moved one space forward to be at position `(3,1,right)`, it would see almost exactly the same thing as when it started:

The goal of localisation is for the robot to figure out its position within the maze, using only the information that the robot can see. The robot can do this by moving about the maze, collecting more information, and narrowing down its position.

2.1 Particle Filter Algorithm

One algorithm for localisation is called a ***Particle Filter***. The robot estimates its position using a set of particles. A single *particle* represents one position that the robot could be at, that is `(x, y, orientation)`. As the robot moves about the maze, it updates the particles (that it, its possible locations), until the robot collapses its location to a single particle.

The overview of the particle filter algorithm is:

Pseudocode for the Particle Filter you will implement

```
1 Let M be the map of the environment
2 Let P be a list of particles (initially empty)
3 repeat
      // Assume the robot has taken one action (rotate or move)
4     Get new observation o
5     Generate new particles, P' using P for where the robot could be
6     Using M, remove all particles from P' that do not match o
7     Replace P with P', that is P = P'.
8 until Program Termination
```

You will implement this assignment[1].

2.2 Worked Example

A worked example of the particle filter algorithm is given using the above maze.

Initially, the robot knows the map, M, and the list particles, P, which is empty.

The robot gets it's first observation:

[1] This is a simplified algorithm for the purposes of this assignment. There are many variations in literature, so be careful!

Since this is the first observation, the robot could be at *any* position within the map, that is any `(x, y, orientation)`. There are 39 ($5 \times 11 - 16$) space that the robot could be at (taking into account walls). Since the robot is facing up, 39 particles should be generated and added to P', all with the same orientation (`up`).

Next these 39 particles are the observation, of which only 6 particles match the observation:

1. `(2,1,up)`
2. `(3,1,up)`
3. `(4,1,up)`
4. `(5,1,up)`
5. `(6,1,up)`
6. `(10,1,up)`

All but these 6 particles are removed from P'. Finally, P is replaced with P' to now contain 6 particles.

Next the robot takes an action and observes:

Thus, the robot make a clockwise rotation action. Thus, for each particle in P, the robot is rotated, to generated 6 new particles in P':

1. `(2,1,right)`
2. `(3,1,right)`
3. `(4,1,right)`
4. `(5,1,right)`
5. `(6,1,right)`
6. `(10,1,right)`

These particles are matched against the observation. Unfortunately, all of the generated particles match the observation, so none are removed. (Don't forget to set $P = P'$!). The robot takes another actions and observes:

This must have been a forward move action (since the robot did not rotate), so P' is:

1. `(3,1,right)`
2. `(4,1,right)`
3. `(5,1,right)`
4. `(6,1,right)`
5. `(7,1,right)`
6. `(11,1,right)`

Matching these against the observation removed 5 and 6, so that P is reduce to 4 particles.

Finally, the robot takes one last action and observes:

This was also a forward movement, so P' is:

1. `(4,1,right)`
2. `(5,1,right)`
3. `(6,1,right)`
4. `(7,1,right)`

Matching these against the observation removed all but one particle. Thus, the position of the robot is narrowed to `(7,1,right)`.

3 Task

This assignment is divided 4 Milestones. To receive a PS grade, you only need to complete Milestones 1 & 2. To receive higher grades, you will need to complete Milestones 3 & 4. You must complete these milestones in sequence. See the Marking Rubric on Canvas for more details.

3.1 Milestone 1: Unit Tests

Before starting out on implementation, you need to write a series of unit tests, so that you can test if your implementation is 100% correct.

A Unit test consists of three text files:

1. `<testname>.maze` - The complete maze
2. `<testname>.obs` - A series of 3x3 observations, provided in sequence separated by new lines.
3. `<testname>.pos` - The final position(s) that the robot could be at once all of the observations have been processed. Each position must be placed on a separate line

Your unit tests should be sufficient so that, if all of your unit tests pass, you are confident that your code in 100% correct.

The starter code provides sample unit tests to help you get started These example tests are not sufficient. ***You will need to write your own tests!***

> **!** The starter code provides a simple unit testing wrapper to help run your unit tests. The testing code is executed by: `./test <testname>`

3.2 Milestone 2: Particle Filter API

Your task is to implement the an API that facilitates the Particle Filter algorithm described in Section 2.1 using three C++ Classes:

- `Particle`
- `ParticleList`
- `ParticleFilter`

For each class, you are given the API that you must implement. This API is a set of pre-defined methods on the classes. You are able to add any of your own code, but you must not modify the provided API.

In addition to implementing the classes:

- You must provide some documentation about your implementation.
- Your code should be well formatted, following the style guide
- Your implementation of the API must comply with the a set of requirements and restrictions.

> **!** You are implementing an API (set of methods), not a full C++ program. To actually run your code, you will need the unit testing file provided in the starter code.
> Remember you cannot modify the pre-defined methods!

3.2.1 `Particle` Class

The `Particle` classes represents one possible position `(x,y,orientation)` of the robot within a maze. It has three methods that provide the components of the position of the particle.

```
// x-co-ordinate of the particle
int getX();

// y-co-ordinate of the particle
int getY();

// Orientation of the particle
Orientation getOrientation();
```

Notice that `getOrientation()` uses a custom type - `Orientation`. This type is represented as an Integer. The definition of the `Orientation` type provided in the started code, and is:

```
// Orientation codes
#define ORIEN_LEFT   0
#define ORIEN_UP     1
#define ORIEN_RIGHT  2
#define ORIEN_DOWN   3

typedef int Orientation
```

To help with using Particles, the `ParticlePtr` type is defined. This is a pointer to a `Particle` object.

```
// Pointer to a Particle
typedef Particle* ParticlePtr;
```

3.2.2 `ParticleList` Class

The `ParticleFilter` class provides a method for storing a list of particles, for the P and P' variables in the particle filter pseudocode (Section 2.1). It has one constructor, one de-constructor, and four methods:

```
// Create a New Empty List
ParticleList();

// Clean-up the particle list
~ParticleList();

// Number of particles in the ParticleList
int getNumberParticles();

// Get a pointer to the i-th particle in the list
ParticlePtr get(int i);

// Add a particle (as a pointer) to the list
//    This class now has control over the pointer
//    And should delete the pointer if the particle is removed from the list
void add_back(ParticlePtr particle);

// Remove all particles from the list
// Don't forget to clean-up the memory!
void clear();
```

To implement the `ParticleList` you **MUST** use an array of pointers to particles. To help you with this, the starter code gives an example way for storing the array of pointers. You are free to change this if you wish.

```
ParticlePtr    particles[100];
int            numParticles;
```

The ParticleList class has full control over all particle pointers that are stored in the list. Thus, if particles are removed from the list you must remember to "delete" the `Particle` object.

> ! Make sure you correctly clean-up the memory used by the `ParticleList`!

3.2.3 `ParticleFilter` Class

The `ParticleFilter` class provides an API (set of methods) to implementation of the particle filter algorithm in Section 2.1. It has three components:

1. Creating a particle filter, using a map of the maze
2. Providing observation updates
3. Retrieving the possible locations of the robot

The map of the maze is provided in the constructor of the `ParticleFilter`

```
// Initialise a new particle filter with a given maze of size (x,y)
ParticleFilter(Grid maze, int rows, int cols);
```

The maze is provided as a `Grid`, which is defined as a 2D array (see below). The top-left corner of the 2D array corresponds to the location `(0,0)`. The parameters `rows` and `cols` give the size of the maze.

It is very important to understand what the `Grid` type is. It is defined as a 2D array (given below). If you recall from lectures/labs, a 2D array is indexed by *rows* then *columns*. So if you want to look-up a position `(x,y)` in the maze, you find this by `maze[y][x]`, that is, first you look-up the `y` value, *then* you look-up the `x` value.

```
// A 2D array to represent the maze or observations
// REMEMBER: in a grid, the location (x,y) is found by grid[y][x]!
typedef char** Grid;
```

New observations are given to the `ParticleFilter` through the method

```
// A new observation of the robot, of size 3x3
void newObservation(Grid observation);
```

This also uses a grid. When this method is called, you may assume that:

- The grid is a 3x3 2D array, again indexed by rows *then* columns.
- The robot has taken precisely ONE action (either a 90° rotation, or a move forward) every time the `newObservation` method is called.

The `ParticleFilter` may be asked for all of the possible locations that the robot might be at using:

```
// Return a DEEP COPY of the ParticleList of all particles representing
//    the current possible locations of the robot
ParticleList* getParticles();
```

The `getParticles` method return an *deep copy* of the particles representing the possible locations of the robot. This means the returns a *copy* of all of the particles!

> ! A deep copy of a `ParticleList` means that a new list is made with a *copy* of all of the particles.

3.2.4 Code Description

At the top of your `ParticleFilter` implementation you must provide a description of the design of your implementation. Provide this in a comment-block. This block should:

- Describe (briefly) the approach you have taken in your implementation
- Describe (briefly) any issues you encountered
- Justify choices you made in your software design and implementation
- Analyse (briefly) the efficiency and quality of your implementation, identifying both positive and negative aspects of your software

3.2.5 Code Style

Your code style will be assessed. It should be well-formatting and comply with the course style guide.

3.2.6 Mandatory Requirements and Restrictions

! Take careful note. *If your submission does not comply with these requirements it will receive a NN grade.*

Your implementation **MUST**:

- Your `ParticleList` implmentation must use an ***Array of Pointers*** to store all `Particle` objects. The starter code provides an example to help you.

Your implementation may:

- Add additional methods to your classes that extend the `Particle`, `ParticleList` or `ParticleFilter`. However, your code will ONLY be assessed using the methods listed in this spec, and provided in the starter code.

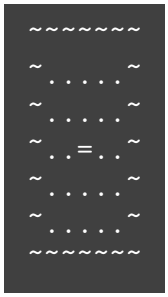
Your implementation **MUST NOT**:

- Alter the definition of the provided methods.
- Use the STL containers (such as `vector`, `list`, `deque`, `map`, or any of `utility`). For this assignment they are banned. If you are unsure if you can use an STL container, ask on the forum first.

3.3 Milestone 3: Unknown Orientation

! This is an *extension* Milestone. You only need to attempt this if seek a *DI* grade for this assignment.

In Milestone 2 the orientation of robot is given in an observation. However, in practical situations, the robot only knows that it is "looking forwards", that is, it ***does not*** know its orientation. Instead, the actual orientation of the robot in the maze must be deduced by the particle filter. Using the following maze.



Consider the following observation, where the symbol representing the robot is an asterisk (*). Here it is assumed that the robot is facing towards the top of the observation grid.

In this case, the robot can actually be at one of 4 positions in the maze:

- `(3,1,up)`
- `(4,3,right)`
- `(3,4,down)`
- `(2,3,left)`

For Milestone 2, you will need to modify your particle filter from milestone 1 to deduce the orientation of the robot if it is unknown. This will require you to think about how to handle different 90° rotations of the robot and maze. For observations for Milestone 2 you may assume that:

- The robot is represented using an asterisk (`*`).
- All observations assume that the direction the robot is facing towards the top of the grid.

3.4 Milestone 4: Efficient Memory Management

This is an *extension* Milestone. You only need to attempt this if seek a *HD* grade for this assignment.

For this Milestone, you need implement the `ParticleList` and `ParticleFilter` classes to use memory as efficiently as possible. Your code should only consume as much memory as is necessary.

To do this, you may wish to consider the following:

- Limiting the size of the array of pointers in `ParticleList` to only the number of cells needed to store all of the elements of the list.
- Storing the maze in `ParticleFilter` to use only the number of rows and columns given in the constructor
- Which class(es) have ownership over any memory, variable or parameter.

3.4.1 Code Description

Finally, for this Milestone, in your implementation of the `ParticleFilter` class, you must:

- Describe (briefly) the approach you have taken in your implementation.
- Justify (briefly) why your implementation is efficient.

4 Starter Code

The starter code comes with the following files:

File	Description
`Types.h`	Header file the typeedefs used in the assignment
`Particle.h`	Header file for the `Particle` class
`Particle.cpp`	Code file for the `Particle` class with dummy method implementations
`ParticleList.h`	Header file for the `ParticleList` class
`ParticleList.cpp`	Code file for the `ParticleList` class with dummy method implementations
`ParticleFilter.h`	Header file for the `ParticleFilter` class
`ParticleFilter.cpp`	Code file for the `ParticleFilter` class with dummy method implementations
`unit_tests.cpp`	Code file to run unit tests

To compile individual files, into object file for testing, you can use the command:

```
g++ -Wall -Werror -std=c++14 -O -c file.cpp
```

4.1 Running Unit Tests

To compile the unit test code, with your implementation into an executable, you can use the command:

```
g++ -Wall -Werror -std=c++14 -O -o unit_tests Particle.cpp ParticleFilter.cpp
                    ParticleList.cpp unit_tests.cpp
```

A unit test can be run with the command:

```
./unit_tests <testname>
```

For example, using the starter code

```
./unit_tests sampleTest/sample01
```

4.2 Getting Started

Part of the learning process of the skill of programming is devising how to solve problems. The best way to get started is to try just give some programming a go and try things out. Experimenting and practising are important to the process of learning the skills of programming and this course. Some suggestions of things you could think about are:

- Have you reviewed the Echo360 video on the assignment?
- Could you solve the particle filter problem by hand?
- Can you devise any simple tests, and test those by hand?
- Have a look at the header files and the methods you need to implement:
- Could you pick a class to start implementing?
- Could you pick a method that you could implement?

5 Submission Instructions

The assignment submission is divided into two ZIP files:

1. Your code solution (in `<student_id>.zip`)
2. Your unit tests (in `<student_id>_tests.zip`)

Upload both ZIP files to the Assignment 1 Module on Canvas.

You may re-submit multiple times, however, only your latest submission will be marked. Be careful, re-submissions *after* the due-date are subject to late penalties.

5.1 Submitting Code Files (.h/.cpp)

Place all of your code, header files (.h/.hpp) and code files (.cpp) into a single ZIP file. Do not use sub-directories.

The ZIP file must be named by your student number:

`<student_id>.zip`

For example, if your student number is `s123456`, your code file submission should be:

`s123456.zip`

5.2 Submitting Unit Tests (.maze/.obs/.pos)

Place all of your unit test files ***in a separate*** ZIP file. You may organise these tests in a clear manner to help us.

The ZIP file must be named by your student number + the `"tests"` label:

`<student_id>_tests.zip`

For example, if your student number is `s123456`, your unit tests submission should be:

`s123456_tests.zip`