UTSeaSim Documentation

November 6, 2011

Contents

1 Introduction

The UTSeaSim simulator is a custom-designed naval surface navigation simulator. It uses realistic $2D$ physical models of marine environments and sea vessels, and runs both in GUI and in non-GUI modes.

The simulator's core contains three main modules: a *Sea Environment* module, a *Ship* module, and a *Decision-Making* module. The sea environment module includes models of winds, water currents, waves, and obstacles. The ship module models all relevant aspects of a ship, including the ship's physical properties, sensing capabilities, and ship actuators. The decision making-module implements an agent that controls a ship autonomously. At each time step, the agent receives the perceptions sensed by the ship, processes them to update its current world state, and decides on control actions for the ship based on its current world state and its decision-making strategy. The following sections describe the functionality of the system and the main simulation modules.

2 Top Level requirements and functionality of the UTSeaSim

Here we briefly describe the high-level flow of the computation, the inputs and outputs of the simulator, and the command line flags used, including some examples.

2.1 Simulation Flow

The "main()" function of the simulator is the if statement at the bottom of the main.py file, which invokes the run() function (also in the main.py file) with some command line flags. The run() function implements the simulation high-level flow, which can be described as follows:

- Load environmental model from input configuration files.
- Load the Ships' initial positions and patrol paths from an input configuration file.
- Create ships.
- Create ship-controlling agents, based on command-line flags.
- Run simulation for n steps, where n can be determined in a command-line flag. Simulation can run both in GUI and in non-GUI modes, depending on a command-line flag.
- Write simulation results to file. In general, results could be any simulation data that is of interest to the user and is gathered during the simulation. The type of results to be written are determined by a command-line flag.

Each simulation step simulates the world change after one second. In general all the units in the simulator are standard: meters, seconds, Kg, and so on. **Note that the frame rate in the GUI is by default 60 frame-per-second, so simulation is displayed in a pace that is 60 times faster than real-time.**

2.2 Inputs and Outputs

A typical run of the simulator simulates autonomous ships navigating in the sea according to some plan, and writing some data that was gathered during the simulation. Below are the inputs and outputs of a simulation.

2.2.1 Inputs

The inputs to the simulator are the environmental conditions, ship positions, ship patrolling paths, and command-line flags:

- **Environmental conditions:** Currently assumed to be in the files ./input_files/wind.py, ./input_files/waterCurrents.py, ./input_files/waves.py, ./input_files/obstacles.py, where '.' is the src directory containing all the files. These are currently just place-holder python files creating objects from classes defined in the source file seaModels.py. In the future, these files could be replaced by text-based format files, like XML.
- **Ship Positions and Patrolling Paths:** Currently loaded from a python file. The python file is a text-file that should be relatively easy to understand and change. In the future, this could be changed to be loaded from a text-based Task Definition Language files. By default, the simulator assumes that the file is shipPathsAndPositions.py, but the filename can be changed using the command-line option -f. The file is always assumed to be under the directory ./input_files/
- **Command-Line Flags:** See section below.

2.2.2 Outputs

In general, the simulation can output any data that is gathered during the ship navigation simulation. Currently, it can output, based on command-line flag choice:

- **Traversal Times Graph Data:** Contains all the patrol nodes of the ships, along with the travel times between pairs of points travelled by some ships.
- **Point Visit-Frequencies:** This data is gathered during a multi-agent patrol, and maps each patrol node to the average frequency in which it was visited by the patrolling ships.

2.3 How To Run

In order to run the simulator, one needs to be in the src directory that contains all the source files and type:

```
python main.py [options]
```

Where options are the command line flags, which are described next.

2.3.1 Command-Line flags

The command line flags can be displayed in a usage message, when running (from inside the src directory):

```
python main.py -h
```

The following message is then displayed:

```
$$> python main.py -h
Usage:
  USAGE:      python main.py <options>
  EXAMPLE:    python main.py --option <value> # TODO complete with real values
                - #TODO explain what the example command do



Options:
  -h, --help            show this help message and exit
  -q, --quietTextGraphics
                        Generate minimal output and no graphics [Default:
                        False]
  -k NUMSTEPS, --numSteps=NUMSTEPS
                        Number of steps to simulate [Default: 10000]
  -d DIR, --inputFilesDir=DIR
                        the DIR in which input files are searched for
                        [Default: input_files]
  -f PATROLFILE, --patrolFile=PATROLFILE
                        A file defining patrol-points, ship initial positions,
                        and patrol paths. The file is assumed to be under the
                        input_files directory [Default:
                        shipPathsAndPositions.py]
  -r RESULTSTYPE, --resultsType=RESULTSTYPE
                        Type of results the simulator outputs. [Default:
                        edgeGraphData]
  -t SHIPTYPE, --shipType=SHIPTYPE
                        The ship's model [Default: basic]
  -w WORLDMODEL, --worldModel=WORLDMODEL
                        World model that the agent maintains [Default:
                        complete]
  -s STRATEGY, --strategy=STRATEGY
                        Agent's desicion making strategy [Default:
                        basicpatrol]
  -y, --rulesOfTheSea   Respect the rules of the sea and yield when needed
                        [Default: False]
```

Brief options descriptions:

- **-h:** Displays a usage message.
- **-q:** When chosen, the simulator would run in non-GUI mode (by default, a GUI mode is invoked)
- **-k:** Number of simulation step to run. One step simulates a second.
- **-d:** Change the directory in which input files are searched for. Usually, there is no need to use this flag.

- **-f:** The name of the file defining the ship positions and patrol paths. This file is assumed to be inside the input files dir, which is controlled by the -d flag and is by default ./input_files/ under the src directory.
- **-r:** The type of result to write. Currently takes one of two options: edgeGraphData and worstFreq (corresponds to edges travel times data and to the worst frequency output data, respectively). For further explanation about the different types of outputs, see the section describing the simulation outputs. The edgeGraphData option generates a edgeGraphData.py output file. The worstFreq option generates a worstFreq.txt output file.
- **-t:** Choose the type of a ship to be simulated. Currently there is only one type, so this option can be ignored.
- **-w:** The type of world model the agent maintains. Currently there is only one type, so this option can be ignored.
- **-s:** The agent's decision making strategy. The default strategy is basicpatrol (cyclic patrol along paths of (x,y) points, given in the input files).
- **-y:** Respect the rules of the sea and yield when needed (by default this is turned off)

2.3.2 Example Runs

Below are a few example runs with explanations.

- python main.py — Opens the GUI and loads the environmental conditions and ship positions and patrol paths from the files inside the input_files directory. The ship positions file is shipPathsAndPositions.py. To start press the "run" button.
- python main.py -q — Runs the same example, in a non-GUI mode. The output file that is written is edgeGraphData.py

3 Functionality of the Sea Environment Module

The sea environment implements different models of winds and currents that affects the ship's motion. We first refer to the HTML documentation of the code, and later describe the module's functionality.

In the HTML documentation, there are two relevant parts. For looking at the environmental model itself:

- Open index.html
- In the main frame, click seaModels.

A summary of all the related classes will be opened. Clicking on any one of them would take you to the corresponding class.

For looking at the ship response to the environmental conditions:

- Open index.html

- On the left frame click on shipModels.Ship
- In the main frame, click on the function: getOffsetByEnvConditions()

The documentation of the function will be opened. A link to the source code of the function is on the right.

The environment conditions model is currently encapsulated inside the Sea class. This class is nothing but a container for Wind, WaterCurrents, Waves and Obstacles classes. Each of the first three classes has only one function: getSpeedVectorInLocation(), which returns, for location (x,y), the speed vector of the wind, water, or the waves respectively. Currently a few simple models are implemented, in which the wind is static and constant, and also the currents are static and constant, but can be different in different areas of the sea. The fourth class, namely Obstacles, is a container of polygon obstacles.

A ship's response to the sea conditions is computed inside the ship itself, as different ships respond differently to the environment. Therefore, each ship has a function getOffsetByEnvConditions(), the documentation of which was mentioned above, that is responsible for computing the ship's offset due to the environment conditions in its (x,y) location. Currently the computation is done based on the ship's orientation, the wind direction, and the currents direction. The model is very simplistic: the wind and the current just offsets the ship in their direction, proportionally to their speed, with some small proportionally constant.

4 Functionality of the Ship Module

The ship module models the ships' physical properties, motion modelling and perception capabilities. We describe each of these next. For each of these, we first refer to the HTML documentation of the code, and later describe the module's functionality.

4.1 Ship's Physical Properties

In the HTML documentation:

- Open index.html
- In the main frame click on shipModels
- In the main frame click on Ship

The documentation of the Ship class will be opened. The Ship class has properties like mass, length and proportianlity constants that affect the computation of the forces and accelarations operating on the ship.

4.2 Ship's Perception Capabilities

In the HTML documentation:

- Open index.html
- On the left frame, click on shipModels.Ship

- In the main frame click on the method: getPercepts()

The documentation of the function will be opened. A link to the source code of the function is on the right.

Our sensing model for the environment is encapsulated inside a ship. The simulator sends its full world state to the ship, and the ship, depending on it's model, extracts from it only the data that it percepts can give it, and send it to the agent that controls the ship. Currently we use the complete state model as percepts, without any filtering, as a simple implementation. Later, this can easily be plugged out and replaced by a more sophisticated perception module that adhers the same interface. In general, any perception module can be plugged into any ship.

4.3 Ship's Motion Model

- Open index.html
- In the main frame, click on simulationEnvironment.simulationEnvironment
- In the main frame click on the method: updateShipExternalState()

The documentation of the function will be opened. A link to the source code of the function is on the right.

In this function, we compute the ship's state in the next time step, based on the current world state (the environment), the ship's engine and steering, and the time passed.

Currently, we approximate ship movement using the following model:

- For forward motion, we model forward force that operates on the ship by the engine, and a drag, which is quadratic in the ship's speed.
- For turning, there is an rotational torque that is applied by the rudder, and is proportional to the ship's speed, and to the projection of the rudder on the lateral direction. There is also a rotational drag force, that is quadratic in the ship's angular speed, (need to check about the accuracy of this modelling).
- Based on the above forces and the ship's mass, we compute the forward and angular accelerations.
- Then, based on the average forward and angular speed in a given time step, computed using the above accelerations, we infer the turn radius, and based on that, compute the ship position at the end of this time-step.

Some approximations we make:

- Although the angular acceleration depends on forward speed, which keeps being changed, we still assume constant forward speed (the avg. speed in this time step) during the computation of angular acceleration.
- Forward acceleration computation does not take into account the effects of turning which might slow it down.
- Forward acceleration depends on the drag, which is changing with speed change, but we approximate the drag based on the initial speed of a time step

5 Functionality of the Decision Making Module

The decision making module is encapsulated inside an Agent class. An agent repeatedly processes percepts, updates its belief about the world state and uses its decision making strategy to choose actions to take (usually steering and engine commands to the ship. In the HTML documentation:

- Open index.html
- In the main frame click on agents
- Optionally, click on the agent class

The agent is composed of two parts: its world model (class AgentWorldModel) and its decision making strategy (class AgentStrategy). In order to create a new agent, one needs to define two classes corresponding to these two parts. The first class should implement the interface of AgentWorldModel and the second class should implement the interface of AgentStrategy. For instance, the AgentBasicPatrolStrategy, implements for cyclic patrol given a list of way-points.

As an example for an agent world model, a world model that is currently implemented is the AgentWorldModelCompleteWorld, which exists mainly for testing purposes. This model assumes that the ship has perfect perceptions and that it receives the complete world state every cycle, so no belief maintenance is needed. In the future, we will add world models with partial observability and sensing of the environment.

5.1 Obstacle Avoidance

Obstacle avoidance behavior is implemented inside the top level class AgentBasicPatrolStrategy (see above). When AgentBasicPatrolStrategy starts to navigate to the next way point, it checks whether the path is blocked by obstacles. If the path is not blocked it navigates directly to the point, using a PID controller (`http://en.wikipedia.org/wiki/PID_controller`) for the steering and engine commands. If the path is blocked, an RRT algorithm (`http://msl.cs.uiuc.edu/rrt/`) computes a bypass and navigate through it.

5.1.1 Limitations

A few current limitations

- The ship only approximately follow the RRT path, as its controls are not fine tuned for very fine maneuvers. As a result, a ship might collide with an obstacle, when the computed bypass is close to the obstacle.

5.2 Rules of the sea

The decision making module has a basic implementation of obeying the rules of the sea. In brief, a ship computes whether there is a potential collision on its navigation path, and in case needed, the ship takes a preventive action. The preventive action is in general: if there are no ships on the right then turn right, otherwise stop.