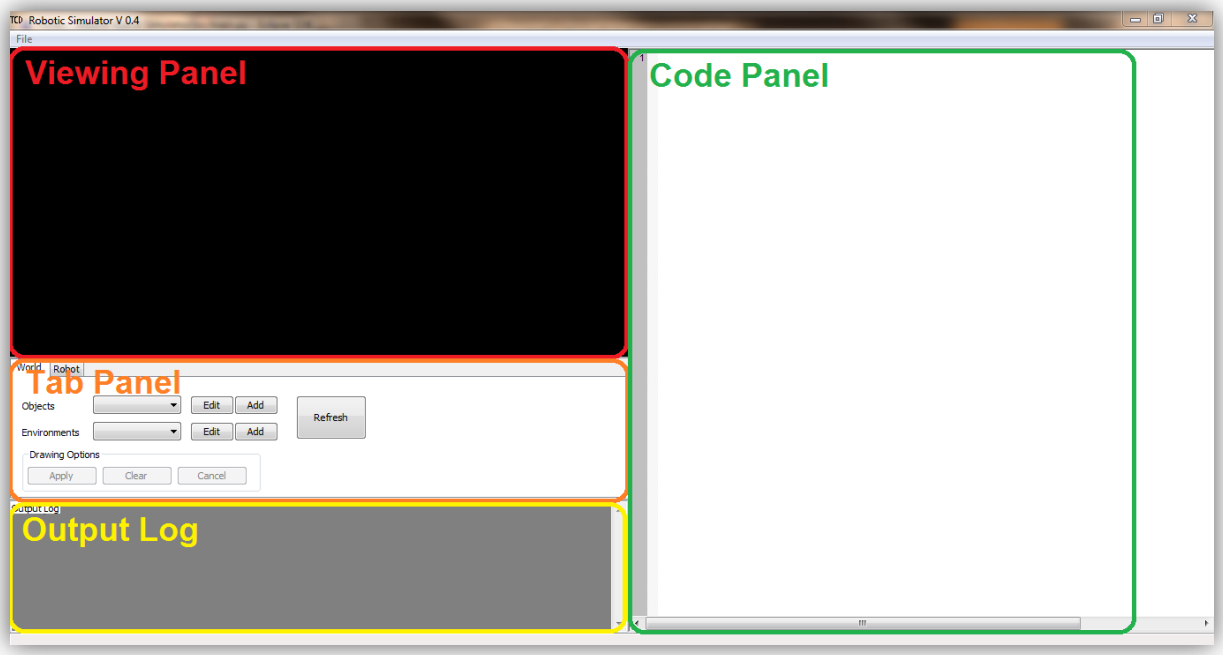
TCD ROBOTIC SIMULATOR V 0.4 USER MANUAL

CONTENTS

INTRODUCTION

This is a brief guide and reference for users wishing to set up worlds, perform simulations and develop robotic controllers using the TCD Robotic Simulator V 0.4. The simulator is still in the early stages of development so parts of this document may change or become irrelevant in the future. For more information please see the sample code provided.

INTERFACE



The major components of the simulators interface are:

- The Viewing panel: This is where the user may observe their simulation, belief space or map space. Left clicking and dragging will cause the view to pan, right clicking and dragging will zoom.
- The tab panel allows the user to switch the simulator's focus from a world editing view to a robot control view. The tab panel has specific functions for both of these tasks: the tab panel can be used to modify and edit objects while in "World Mode" and can be used to start/stop tests, lay tracks, and show the belief space when in "Robot Mode". Switching modes will also change the text being shown in the code panel on the write.
- The output log outputs status and error messages to the user. The user may write data out to this log via the python print command, in order to debug/understand their controllers.
- The code panel is where the user may edit their controller/world code.
- The user may use the file menu to start new projects, load controller or world files, save files, and import their own modules.

STARTING A NEW PROJECT

To start a new project click "New Project" in the file menu, and specify a project name and location. A "Project" consists of two files: the world file and the robot controller file. A world file is used to set up the test environment, robotic platform and map space by the user. This is done within the *init* function:

```
def init(sim_world, map_space):
    "Set up world, objects, robot & sensors"
```

Two variables are provided for the user: sim_world allows the user to edit a blank simulator world, map_space allows the user to populate a grid like representation of their world for use by their robot.

A controller file is used to control the robot(s) described in the world file, but can also control certain objects within then world (e.g. cause a door to move, simulating a potential obstacle for a robot). When a test is started the simulator will execute the *run* function from the control file, this function should contain a loop to keep it "alive":

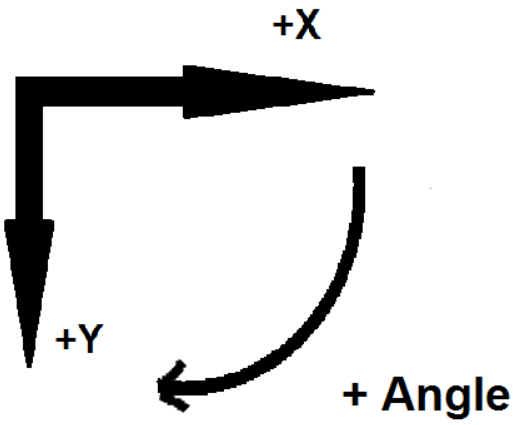
```
def run(robot, keep_going, belief_space, map_space, interact):
    "A simple control routine"

    while keep_going():
        """Keep_going will stop this loop if the user requests so.

        This is done by clicking "Stop" in the simulator."""
```

Several variables are provided in the *run* function: robot is used to address robot(s), keep_going allows the user to stop their controllers main while loop from the simulators interface, belief_space allows the user to update and query a belief space, map_space allows the user to query the map space, and interact allows the user to interact with world objects.

A WORD ON UNITS AND CONVENTIONS

Positions and shapes are described using Cartesian coordinates. The standard unit of distance is centimetre, i.e. the distance from [10, 0] to [20, 0] is 20 centimetres.

Angles are described in degrees. The acceptable range for input angles and coordinates is infinite for all practical purposes.



SETTING UP A WORLD

In the Trinity Robotic Simulator a world is a collection of objects, environments, robots and sensors. When setting up a world, we first describe the objects and then add then to the world. This is done in the world python file's *init* function.

ROBOT

```
Robot(name, position, orientation, colour, shape,
   kinematic_properties, kinematic_model)
```

Initializing a Robot:

- name: ***String***. Make sure you give each robot (if running multiple robots in one simulation) a unique name.
- position: ***List of integers [X,Y].*** Cartesian coordinates.
- orientation: ***Integer.*** Angle in degrees.
- colour: ***List or tuple of 3 integers from 0 to 255 in value (R, G, B).***
- shape: ***List of Lists of Integer pairs, each sub list is a Cartesian coordinate [[X1,Y1], [X2,Y2]….].*** This describes the shape of the robots body which will be based symmetrically about its position.
- kinematic_properties: ***A tuple of variables*****.** Depend on the specified kinematic model. See below.
- kinematic_model: a model that describes the robots chassis. If this is not specified then the standard model will be used, **TwoWheelKinematics**. The kinematic_properties of this model are (wheel_radius, axel_width) which are both described as **integers** of centimetres.

One you have finished describing the robot you may add it to the world:

```
sim_world.add_robot(robot)
```

- Your robot **must** be finalised before it is added. This means that you must also **attach sensors** before adding the robot to the simulator world.

SENSORS

3 Sensors are currently described in this simulator. The first and most simple sensor is a 1 dimensional **Laser**. Also available are the **LIDAR** and **Ultrasonic** sensors. A **LIDAR** sensor senses distance over a flat plane. The **Ultrasonic** Sensor sensing using sound, and can cover a wide area.

LASER

```
Laser(name, range, angle, probability_model)
```

- name: ***String.*** Make sure this is unique.
- range: ***Integer***
- angle: ***Optional Integer.*** This angle does not need to be specified, since the senor will snap to the orientation of the robot it is attached to.
- probability_model: ***Optional Probability Model.*** See technical documentation for more information on implementing a probability model.

LIDAR

```
LIDAR(name, range, angle_span, number_of_points)
```

- angle_span: ***Integer***. Defines the width of the LIDAR's sensing span.
- number_of_points: ***Integer***. Determines the number of points the LIDAR sensor will divide its angular span into.

ULTRASONIC SENSOR

```
UltraSonic(name, range, dead_zone_rane, minor_radius, angle, number_of_points)
```

- dead_zone_range: ***Integer***. The dead zone range is the minimum reading distance of the sensor.
- minor_radius: ***Integer.*** An Ultrasonic sensor is approximated as an ellipse. The minor radius describes how "wide" the sensor's reading area is.
- angle: ***Optional Integer.*** This angle does not need to be specified.
- number_of_points: ***Optional Integer.*** The number of points that the sensor will be approximated using.

ATTACHING A SENSOR TO A ROBOT

```
robot.add_sensor(sensor, offset, angle)
```

- sensor: ***Sensor {Laser, LIDAR, UltraSonic}.***
- offset: ***Optional List of Integers [X,Y].*** Describes where the sensor is placed on a robot relative to the robot's centre.
- angle: ***Optional Integer***. The angle that the senor makes relative to the forward angle of the robot.

ENVIRONMENT

An environment acts as the ultimate boundary of a simulation world, and in future implementations of the simulator will have conditions such as light and surfaces friction that may affect sensors and robot kinematics.

For the moment the environment object serves no real purpose (a simple object can be used as a surrounding/barrier more easily) but is still required in order for a simulation to work.

Creating an Environment:

```
Environment(shape,name)
```

- shape: ***List of Lists of Integer pairs, each sub list is a Cartesian coordinate [[X1,Y1], [X2,Y2]….].*** Unlike the shape of a Robot or ***BasicObject*** this is the shape of the environment in absolute terms.

Setting the world's environment:

```
sim_world.set_environment(Environment)
```

BASICOBJECTS

A basic object is a simple 2d object defined by a list of Cartesian coordinates. It may act as a physical barrier to a robot and is detectable by sensors.

```
BasicObject(position, orientation, shape, name, colour)
```

- position: ***List of integers [X,Y].*** Cartesian coordinates.
- orientation: ***Integer.*** Angle in degrees.
- shape: ***List of Lists of Integer pairs, each sub list is a Cartesian coordinate [[X1,Y1], [X2,Y2]….].*** This describes the shape of the robots body which will be based symmetrically about its position.
- name: ***String***. Make sure you give each object a unique name.
- colour: ***List or tuple of 3 integers from 0 to 255 in value (R, G, B).***

A ***BasicObject*** once specified may be added to a simulation world as follows:

```
sim_world.add_object(BasicObject)
```

MOVING OBJECTS

A moving object is simply a BasicObject which moves. A path is described for it Cartesian coordinates, and a speed in centimetres per second. A moving object may be used to act as a door, simulation moving objects such as people, vehicles or other robots, or generally just to introduce transient noise into the world.

```
MovingObject(starting_position, orientation, shape, name, colour, speed, path)
```

- speed: ***Integer.*** The speed of the object in centimetres per second.
- path: ***List of Lists of Integer pairs, each sub list is a Cartesian coordinate [[X1,Y1], [X2,Y2]….].*** These integers describe the points of a path that will be travelled by the object in **absolute Cartesian coordinates.**

Once described a ***MovingObject*** is added to the world in the same manner as a ***BasicObject***. A moving object may be triggered from the control program at any stage via the ***interact*** and ***execute*** functions.

```
interact(name).execute(repeating)
```

- name: ***String***. The same name as moving object you intent to interact with.
- repeating: ***Optional Boolean.*** If repeating is set to **True**, then the object will repeat its described movement for the duration of the simulation.

WRITING A CONTROLLER

Control programs are described using a controller file. The primary function *run* is executed when the user starts a test. A while loop should be situated within the run function so that it can act as a continuous controller. As well as interacting with a robots sensors and affecters, moving objects can also be executed.

INTERACTING WITH ROBOTS

The controller can interact with a robots sensors and affecters using the *robot* function (a supplied argument of the *run* function). Before the robot can be used it must be started.

```
robot(name).start()
```

- name: ***String***. The name of the robot the user wishes to control.

Controlling wheel speed:

```
robot(name).set_wheel(wheel, wheel_speed)
```

- wheel: ***String.*** Name of the wheel ("left" or "right" for the ***TwoWheelKinematics*** model).
- wheel_speed: **Integer.** The turning speed of the wheel in radians per second - can be negative or positive.

Obtaining sensor data:

```
reading = robot(name).sensor_reading(sensor_name)
```

- sensor_name: ***String.*** Name of the sensor that one wishes to receive data from.
- Sensors normally return either a single integer value (in the case of the ***Laser*** sensor and the ***UltraSonic*** sensor), or a list of points from left to right (for a ***LIDAR*** sensor, evenly distributed across the sensor width).

As part of a competent control system, the controller should be able to determine the position of the robot via a user defined localization algorithm. In the interest of aiding the user in the development of such an algorithm, and for comparison sake, the actual position and angle of the robot in the simulation space may be obtained as follows:

```
position = robot(name).position
```

```
angle = robot(name).orientation
```

INTERACTING WITH THE WORLD

As described previously moving objects can be trigged within the controllers run function using the *interact* function.

TIPS

- Since the control program is running on a virtual environment and many variables such as limited sensor bandwidth and sensor latency are not modelled the user should use the sleep function to apply appropriate pauses. I.e. a sleep function in a while loop that checks a sensor reading, causing the reading to be read every have a second, rather than continuously… which is not a realistic option.

PROBABILITY SPACES

Probability spaces are useful assets in autonomous mobile robotics as they help us to perform localisation and path planning strategies. In the interest of decreasing project complexity the simulator has inbuilt belief and map spaces which can be easily understood using the simulators visual capabilities.

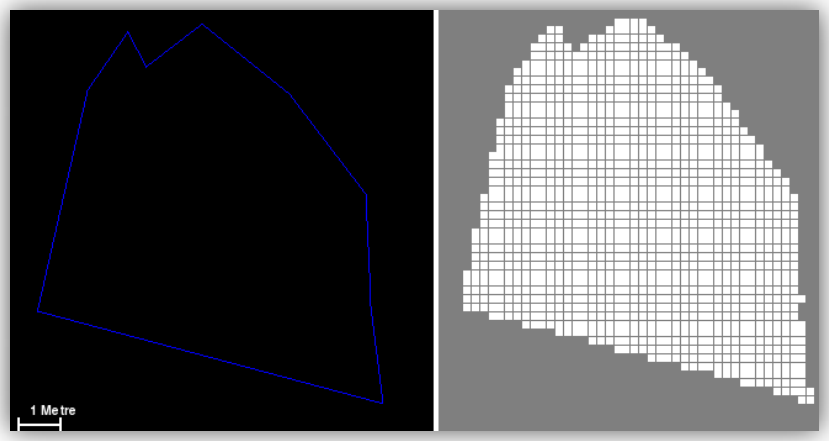


Figure 1. Left: a BasicObject in the simulator world. Right: The same object transferred to a map space.

The probability of 1 for a specified grid location means that there is definitely an object in that location. 0 would mean that there is definitely not an object there, and a probability of 0.5 means that nothing is know about the specific grid location.

BELIEF SPACE

The belief space provided by this simulator is a "sudo infinite, grid like" mapping of the world that is updated by the robots sensors readings on request. The user must manually update the belief space within the run function. Before we can use the belief space we must set it to our robot, so that it may know some information about the robots sensors:

```
belief_space.set_robot(robot(name))
```

We may then update the belief space with information gathered from sensors:

```
belief_space.update(robot_position, robot_angle,{sensor_name : reading})
```

- robot_position: ***Integer list [X,Y].*** This describes the perceived current location of the robot at the time that this sensor reading(s) was taken.
- robot_angle: ***Integer***. The forward angle of the robot at the time of reading.
- The sensors information is sent in a **Dictionary.**
  - sensor_name: ***String.*** This is the key of the dictionary.
  - reading: the information returns from robot(name).sensor_reading(sensor_name)
  - Example: {"Laser" : laser_reading, "LIDAR": LIDAR_reading}
- **Note:** multiple sensor readings can be sent to the belief space within the sensor dictionary.

In order to use the data we have discovered about the world by updating the belief space with sensor readings, we must query the belief space. This may be done as follows:

```
probability = belief_space.query(position)
```

MAP SPACE

The map space is not update in real time; instead it is information about an environment which is known before a robot enters it. Only probabilities of 1 and 0.5 exist in the map space –definite object or unknown. Objects may be copied directly from the world into the map space from within the world file *init* function.

```
map_space.add_object(BasicObject, solid)
```

- solid: ***Optional Bool.*** Describes whether the object should be drawn as a solid filled object in the map space.

The Map space can be queried for information from with a controller in the same way as the belief space. Unlike the belief space, map space has **No** ***set_robot*** function .

KNOWN ISSUES

Be wary of the following:

1. Creation of BasicObjects using the GUI names the objects as “anotherobject”. Please make sure to change this to something more specific.
2. When editing BasicObjects using the GUI sometimes changes may not appear to take effect. Please make sure to check that you do not have two objects with the same names, even if one of the objects code is commented.
3. When adding objects to the map space as “solid” drawing errors may occur. Use the map space view to check for these. If an objects perimeter is not complete when the object is not added as “solid” then this is the cause. Minor changes the shape may fix this.
4. Receiving the message “Please load world and robotic controller before starting test”. This can be remedied by fixing any errors stated in the output log, and then restarting the simulator and reloading the world and control files.
5. User module __init__ errors. Go file> View Modules, and edit the __init__ file, removing any missing modules.
6. Editing moving objects in via the GUI object editor will cause a **catastrophic error.**