MComp - Obstacle Avoidance

Generated by Doxygen 1.9.4

Chapter 1

MComp Lawn Mower Obstacle Avoidance

Obstacle avoidance using LiDAR testing and simulation. Development and testing of methods for eventual use in the automatic lawn mower. To use input a perimeter, a list of nogo zones, a start point, and an end point. These points should be in GPS format as they will then be converted to UTM. The nogo zones and perimeter can typed literally, but the current method (and would be easier) to use a .out file created using np.savetxt.

UTM has been chosen as the distance the lawn mower will travel is small enough such that the curvature of the Earth is thought to have negligible affect on accuracy. UTM also provides an easy way to traverse the space in metres, using the compass on the robot.

A perimeter is needed but to test random nogos within the perimeter use the `-t` arg, followed by the number of runs. This will output the final image, total distance, coverage, and the number of skipped checkpoints when finished. To output printing messages the user can pass `-v`, and to output images for each movement (to create GIFs similar to those shown in this README) the user can use `-p`.

Note: Images can cause a long wait time due to the number of detected points being plotted.

If the user does not wish to use GPS points for the perimeter or nogo zones then generic x, y points should be fine but the conversion to UTM needs to be omitted.

To generate a path for the robot to follow within the given perimeter one can use my `mapping` repo.

This repo is intended to be used to learn about methods, and test said methods to determine efficiency and find issues before implementing on the robot designed for this MCOMP project.

1.1 Changelog

```
Some updates do not come with a changelog update or
descriptive commit, this is due to the repo being a
place to test, develop, and keep backups of this work.
Only major changes which are considered to be long
term or substantial will be described in this section.
This repo is primarily for learning, testing, and
theoretical development, it is not necessarily
intended to be used in actual projects,
but it is hoped that this repo can at least help
those understand and visualise methods required for
path traversal and obstacle detection.
```

- 17/02/2023: Added methods to stop the robot seeing through obstacles. The method is more relatively exhaustive but wouldn't be needed in real-life.

  - Testing with a smaller LiDAR range. Fixed getting stuck on a wall. Allowing for more direct travel.

- 18/02/2023: Tested on coverage map generated without known obstacles.
- 08/03/2023: Integrated the route traversal methods from the `Mapping` repo
  - Fixed incorrect back-on-track point
  - Removed section instructing robot to move within the line detection method
- 09/03/2023: Fixed doubling back if off course.
  - If the robot becomes 'off-course' but an obstacle is between it and the ideal path then it continues as normal until past the obstacle
- 10/03/2023: Fixed issues with robot seeing through walls
  - Added method to prevent movement that results in moving through walls
  - Changed logic for determining which direction to move when an object is found
- 11/03/2023: Added methods to produce randomly generated nogos within scene, output the shapes, and output coverage, total distance, and number of points skipped
  - Now the robot has N number of tries (relating to distance from origin) to reach a point, if this fails then A$*$ is used along with the robot's knowledge of the scene from detecting points to determine if the point is accessible or not. **Needs more testing**
    - If the point is determined inaccessible then the robot will skip this checkpoint.
- 12/03/2023: Instead of N amount of tries, A$*$ is used when the robot makes minimal progress within N amount of moves.
- 15/03/2023: Fixed issue with A$*$ not providing a correct path
  - Testing more *realistic* values based on RTK inaccuracy and LiDAR ranges
- 04/04/2023: Added method to reduce number of detected points to minimal set required for A$*$. Method needs more testing, but appears stable.

1.2 TODO

- [x] Handle multiple objects
  - Both close together and far apart (allowing movement between)
- [x] Stop the robot seeing through walls
- [x] Handle case if point is inaccessible due to an obstacle covering, surrounding or blocking
- [x] Combine with route mapping - if off course and no obstacles, move back to route.
- [ ] ~~Move robot to end of detected end point~~
- [x] Test scenes with randomly generated nogos
- [x] Reduce number of detected points to minimal set
- [ ] Consider dimensions of robot - not just centroid
- [ ] More user friendly input, options, and output
- [ ] User guide
- [ ] Full documentation of methods

1.3 Known Issues

- Max tries methods have different logic and so the name and return need to be changed.
- Some methods have grown too large and less succinct, some refactoring is needed.

1.4 State Diagrams

Work in progress - states are split to make them more readable and easier to understand (links only work when viewing raw image in browser)

- Main State - four main states, moving to point, avoiding obstacles, A$*$ path, and back on track.
- A$*$ Path State - Similar to main state but only initialised when the robot cannot reach a point within N amount of moves, in this state the robot does not become off course as this allows path (if not perfect) to pull the robot towards the desired point. If the robot makes no progress in this state then it skips a point, otherwise a recurisve loop of A$*$ could occur.
- Back on Track State - Similar to main state but only initialised when the robot is off course **and not avoiding obstacles**, and has only one point. A$*$ can be used to find a path to this point, but as this point moves as the robot moves this should not occur often. Ultimately, the back-on-track point is inaccessable and A$*$ cannot find this point, then it is likely the robot cannot reached the original point and should therefore skip it.

1.5 Current Pipeline

- Generate a 'perfect' route with some amount of overlap, in the below case it is 25% the width of the robot
- Reduce the number of points by removing those along the same line
  - This reduces the memory requirements but also reduces the chances a point is within an unknown object
- Follow this route with unknown obstacles within
- Without map matching to the route this gives the following result
- Applying map matching is the next step and should improve the total coverage
- Using the robot's location and points detected from the LiDAR unknown objects can be detected
  - These points can be sent back to the server for a more optimal coverage map
  - This method could also be applied to improve the digital map's accuracy
- Applying the map matching to improve the overall coverage

```
When the robot is considered off course a new target point is determined.
This point lies on the line between the origin and current destination,
initially it is the closest point on the line to the robot. However,
this creates problems when an obstacle is between it and this new point -
the robot is most often off course when avoiding obstacles. In an attempt to
avoid obstacles and move in the correct direction the new point is offset
by some value. It appears, the greater the value the less chance of getting stuck,
but the lower total coverage.

Overall coverage can be limited to RTK accuracy, route, and unknown obstacles.
These inaccuracies can be hard, or in some cases not possible due to the limitations
of the hardware in use. To overcome this one method is to provide more than one
route. This provides more work for the robot and is in-efficient however, for
an automated robot the time to finish requirement is not necessarily the primary
factor - though battery usage may be. In this project total coverage is the primary
focus and as such this is a convinient way to produce desired results.
```

- Using two overlapping routes to increase overall coverage

1.6 Tasks

- What level of route overlap do we need?
- What balance of accuracy to memory efficiency should we have?
  - More points means a higher coverage percentage, the above routes are 907 and 118 points respectively.
  - Can we find a middle ground or should we perform multiple different routes to account for the innacuracy.
- What else, other than RTK inaccuracy, will affect our route?
- What is an acceptable time to compute, total route distance, and time to complete route.
- Testing different off course offset values.
- A$*$ is used to find a new path dependant on the known boundaries, when to ignore this route and skip to the next checkpoint - deciding between coverage and time to find next point.

1.7 Progress

1.8 References

The algorithm in its current state is based primarily on the work found in:

Peng, Y., Qu, D., Zhong, Y., Xie, S., Luo, J., & Gu, J. (2015, August). The obstacle detection and obstacle avoidance algorithm based on 2-d lidar. In 2015 IEEE international conference on information and automation (pp. 1648-1653). IEEE.

Further reading has been and will continue to be conducted therefore, this section will be updated when ever the implementation draws from those sources.

Chapter 2

Namespace Index

2.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 4

Data Structure Index

4.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 5

Namespace Documentation

5.1 consts Namespace Reference

Variables

- int **ON_COURSE** = 0
- int **OFF_COURSE** = 1
- int **OBJECT_TO_LEFT** = -1
- int **OBJECT_TO_RIGHT** = 1
- int **OBJECT_TO_CENTRE** = 0

5.1.1 Detailed Description

```
Small set of constants used amongst modules
```

5.2 lidar_sim Namespace Reference

Data Structures

- class NoPath
- class NoTurns

Functions

- def **printable_detected_ends** (ends)
- def **print_graph** (mower, test_shape, nogos, path, current, target, img, detected)
- def collinear (p0, p1, p2)
- def **get_line** (a, b)
- def perpen_point (p, a, b)
- def avoidance (mower, path, target, nogos, centre_line, test_shape, current, img, right_bear, left_bear, centre_bear)
- def **signal_handler** (sig, frame)
- def **main** (to_test, run_num)

Variables

- int **lidar_range** = 60
- int **lidar_dist** = 1
- float **move_dist** = 0.3
- int **lidar_width** = 15
- float **obj_gap** = 0.5
- int **try_count** = 5
- int **Q_SIZE** = 10
- float **MAX_SPEED** = 0.6
- bool **PRINT** = False
- int **OFF_COURSE_TARGET** = -1
- int **SIGINT** = 0
- bool **DIRECT** = False
- bool **REAL_TIME** = False
- **profiler** = cProfile.Profile()
- int **runs** = 1
- bool **to_test** = True
- **stdout**
- **stats** = pstats.Stats(profiler).sort_stats('tottime')

5.2.1 Detailed Description

```
 LiDAR SIM is intended to test methods and parameters for the eventual use
on a robotic mower. The intention is to better understand and experiment with
methods before implementation, hopefully allowing issues to be found and
visualised.

The methods are primarily based on the work found in:

Peng, Y., Qu, D., Zhong, Y., Xie, S., Luo, J., & Gu, J. (2015, August). The obstacle detection and obstacle av
```

5.2.2 Function Documentation

5.2.2.1 avoidance()

```
def lidar_sim.avoidance (
              mower,
              path,
              target,
              nogos,
              centre_line,
              test_shape,
              current,
              img,
              right_bear,
              left_bear,
              centre_bear )
```

```
The main driver function to avoid obstacles.

Args:
    mower: The lawn mower object.
    path: The traversal route.
    target: The current target within the route.
    nogos: The list of nogo-zones in the scene.
    centre_line: The centre line of the robot's LiDAR.
    test_shape: The outer perimeter of the scene.
    current: The current (previous target) within the route.
    img: The image number used for exporting images.
    right_bear: The list of bearings to try when turning right.
    left_bear: The list of bearings to try when turning left.
    centre_bear: The list of bearings to try when iteratively
        turning left and right.
```

5.2.2.2 collinear()

```
def lidar_sim.collinear (
              p0,
              p1,
              p2 )


Determine if three points are collinear using the slope method
```

5.2.2.3 perpen_point()

```
def lidar_sim.perpen_point (
              p,
              a,
              b )


Get a point on a line perpendicular to a 3rd point.

This function is used to determine the back-on-track point when
the robot is off-course and not avoiding obstacles.

Args:
    p: The given point - the robot's location
    a: The start point of the line
    b: The end point of the line
```

5.3 mower Namespace Reference

Data Structures

- class Robot

Functions

- def shift_float (num)
- def shift_int (num)

5.3.1 Detailed Description

```
 Robot is the main class for controlling the robotic mower and determining
detected objects with the LiDAR - the LiDAR may be seperated out in future
development.

Note: The LiDAR values do not have to be the maximum values of the sensor
used, a subset can be considered i.e. one can consider only the front section
when using a 360 LiDAR.
```

5.3.2 Function Documentation

5.3.2.1 shift_float()

```
def mower.shift_float (
              num )

Shifts a UTM value from standard accuracy to desired accuracy.

Shifting by 10 produces cm level scale. More than this and the
robot is often engulfed by nearby points, too little and A* thinks
the robot can fit through small gaps (A* doesn't account for the
robot's width)

Args:
    num: The UTM number
```

5.3.2.2 shift_int()

```
def mower.shift_int (
              num )

Reverse the shift done by 'shift_float'

Args:
    num: The shifted UTM value
```

5.4 utm_func Namespace Reference

Functions

- def utm_bearing (p1, p2)
- def utm_bearing_dist (p1, p2)
- def to_utm (points)
- def utm_dist (p1, p2)
- def utm_coords (point, heading, distance)

5.4.1 Detailed Description

```
 Small collection of functions to reduce code size when using
the surveytoolbox UTM functions.

Functions are not uniform, some use dicts, some return raw values.
# TODO make functions return predictable values
```

5.4.2 Function Documentation

5.4.2.1 to_utm()

```
def utm_func.to_utm (
              points )
```

```
Converts lat/long points to UTM equivalents.

During this conversion it is necessary for the grids to be the
same to reduce error. It is unlikely in our application for them
to be different due to the size of each region but it is checked
and a warning given nonetheless.
```

5.4.2.2 utm_bearing()

```
def utm_func.utm_bearing (
              p1,
              p2 )
```

```
Short wrapper function to reduce lines of code.
Returns the bearing of two points.
```

5.4.2.3 utm_bearing_dist()

```
def utm_func.utm_bearing_dist (
              p1,
              p2 )
```

```
Short wrapper function to reduce lines of code.
Returns the bearing and distance of two points.
```

5.4.2.4 utm_coords()

```
def utm_func.utm_coords (
              point,
              heading,
              distance )
```

Short wrapper function to reduce lines of code.
Returns the coordinates of a point from the current
given a heading and distance.

5.4.2.5 utm_dist()

```
def utm_func.utm_dist (
              p1,
              p2 )
```

Short wrapper function to reduce lines of code.
Returns the distance from two points.

Chapter 6

Data Structure Documentation

6.1 Ends Class Reference

Public Member Functions

- def **__init__** (self, p1, p2)
- def **line** (self)
- def **array** (self)
- def **points** (self)
- def **close_to** (self, other)
- def **overlaps** (self, other)
- def **touches** (self, other)
- def **collinear** (self, other)
- def **is_same_line** (self, other)

Data Fields

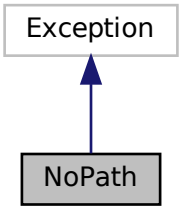
- **p1**
- **p2**

The documentation for this class was generated from the following file:

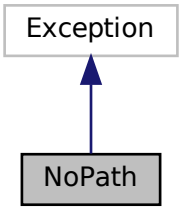
- ends.py

6.2 NoPath Class Reference

Inheritance diagram for NoPath:



Collaboration diagram for NoPath:

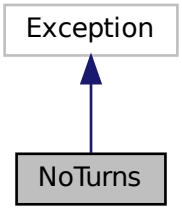


The documentation for this class was generated from the following file:

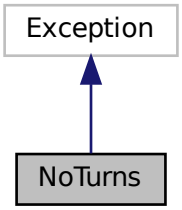
- lidar_sim.py

6.3 NoTurns Class Reference

Inheritance diagram for NoTurns:



Collaboration diagram for NoTurns:



The documentation for this class was generated from the following file:

- lidar_sim.py

6.4 Robot Class Reference

Public Member Functions

- def **__init__** (self, x, y, target, Q_SIZE, lidar_range, lidar_width, lidar_dist, move_dist, try_count, obj_gap, REAL_TIME, DIRECT)
- def enqueue (self, val)
- def per_on_course (self)
- def is_off_course (self)
- def clear_q (self)
- def seperate_lidar_ranges (self)
- def is_outside_buffer (self, path, current, target)

- def max_tries (self, path, target)
- def print_finder_graph (self, target, b, matrix)
- def order_line (self, line)
- def detected_points_to_lines (self)
- def clean_apath (self, path, b)
- def is_accessible (self, target)
- def lidar_lines (self)
- def find_nearest (self, points)
- def remove_hidden (self, points)
- def detect_objects (self, nogos, target)
- def closer_point (self, p1, p2)
- def move (self, metres, target)
- def find_target (self, nogos, target)
- def get_detected_line (self, target, nogos, heading, centre_line, test_shape, current, img, path)
- def get_closest_intersect (self, detected_line, lidar)
- def lidar_intersecting (self, centre, left, right, detected_line)

Data Fields

- **x**
- **y**
- **visited**
- **heading**
- **target**
- **inside**
- **Q_SIZE**
- **nogo_polys**
- **per_poly**
- **dist_travelled**
- **detected_ends**
- **detected_points**
- **tries**
- **lidar_range**
- **lidar_dist**
- **lidar_width**
- **move_dist**
- **try_count**
- **obj_gap**
- **REAL_TIME**
- **DIRECT**

6.4.1 Detailed Description

```
The Robot class is the simulated version of a lawn mower
used for testing obstacle detection, mapping, and
area coverage.
```

6.4.2 Member Function Documentation

6.4.2.1 clean_apath()

```
def clean_apath (
              self,
              path,
              b )
```

Takes an A* path and removes any points that violates a known boundary.

6.4.2.2 clear_q()

```
def clear_q (
              self )
```

Resets the queue. Used when the robot confirms reaching a checkpoint - may not be used in real application.

6.4.2.3 closer_point()

```
def closer_point (
              self,
              p1,
              p2 )
```

Finds the closer of two points on a UTM grid

Returns:
    The closest point.

6.4.2.4 detect_objects()

```
def detect_objects (
              self,
              nogos,
              target )
```

Find all objects within range of the robot's LiDAR.

This method does require the objects to be 'known' but will only return those visible to the robot.

Args:
    nogos: The list of nogos in the scene.
    target: The robot's current coordinate target, typically
        the next in the given route.

Returns:
    All objects within range of the robot.

6.4.2.5 detected_points_to_lines()

```
def detected_points_to_lines (
              self )
```

Reduce the detected points to simpler linestrings.

Using all detected points quickly gets out of hand, so lines
are used instead. The end points of these lines can be used to
replace the current detected points to save memory and
processing time.

6.4.2.6 enqueue()

```
def enqueue (
              self,
              val )
```

Keep track of the n amount of on-course and off-course
during traversal

This queue is used to keep a n amount of on and off course
readings to confirm whether the robot has become off course.
Only one value may be an error and so a number are used to
confirm.

Args:
    val: Whether the robot was on-course or off-course

6.4.2.7 find_nearest()

```
def find_nearest (
              self,
              points )
```

Given a set of detected objects, find the points nearest to
the robot. The robot will handle the closest object first.

Args:
    points: All detected points.

Return:
    The nearest object, or merged objects.

6.4.2.8 find_target()

```
def find_target (
              self,
              nogos,
              target )
```

```
A helper function to re-orientate the robot to the next
node in the route.

Args:
    target: The robot's current target.
```

6.4.2.9 get_closest_intersect()

```
def get_closest_intersect (
              self,
              detected_line,
              lidar )
```

```
Considering all LiDAR lines in this region, which point is closest
to the robot.

Args:
    mower: When determining which way to move the robot should
        ideally move away from the closest point as this will
        result in faster and more efficient obstacle avoidance.
        All points must be considered due to irregular shapes, but
        the closest is the primary concern.
    detected_line: The current detected line on a shape defined by
        the two end points.
    lidar: The set of LiDAR lines for a given region (left-side,
        right-side, centre)

Returns:
    The closest distance from robot to detected point, and whether
    the region intersects at all.
```

6.4.2.10 get_detected_line()

```
def get_detected_line (
              self,
              target,
              nogos,
              heading,
              centre_line,
              test_shape,
              current,
              img,
              path )
```

```
Get the points detected by the LiDAR.

Depending on the number of points return either a polygon,
line, or single point. Whilst there are no detected obstacles,
move forward - try in the direction of the target first, if
obstacles then return to original given heading.

Args:
    target: The current target node in the route.
    nogos: The list of bounds - nogos and perimeter
    heading: The given heading to try.
    test_shape: The bounds of the area.
    current: The current positon in the route - for printing
        the graph
    img: The current image number - for printing the graph
    path: The given coverage route.

Returns:
    The detected points as a shape, line, or single point.
```

6.4.2.11 is_accessible()

```
def is_accessible (
              self,
              target )

When the robot is unable to make progress it may be stuck,
using it's knowledge of the scene an A* path is generated. If
no path can be generated then the point is unreachable and
shall be skipped.
```

6.4.2.12 is_off_course()

```
def is_off_course (
              self )

Based on the percentage of on-course values, decide if the
robot is considered off-course or not
```

6.4.2.13 is_outside_buffer()

```
def is_outside_buffer (
              self,
              path,
              current,
              target )

Based on the robots distance from the line between its
origin and target whilst traversing a path, it is outside the
region of inaccuracy (the buffer).

Args:
    path:
    current:
    target:
```

6.4.2.14 lidar_intersecting()

```
def lidar_intersecting (
              self,
              centre,
              left,
              right,
              detected_line )
```

```
Determine if the LiDAR is detecting an object and if so, where
on the LiDAR range.

Depending on where the object is in the LiDAR's range the robot
will want to move differently i.e. if the object is to the right
then the robot will want to move left.

Args:
    centre: The centre line of the LiDAR
    left: The left-most line of the LiDAR
    right: The right-most line of the LiDAR
    mower: The lawn mower.
    detected_line: The line detected by the robot.

Returns:
    If a line has been detected and in which direction.
```

6.4.2.15 lidar_lines()

```
def lidar_lines (
              self )
```

```
Generate an array of lines at different angles to simulate
the robot's LiDAR

Returns:
    inter in index 0 is the LiDAR array used for detection.
    The polygon in index 1 is simply used for the visual
    output.
```

6.4.2.16 max_tries()

```
def max_tries (
              self,
              path,
              target )
```

```
Determine if the robot has exceeded the maximum number of
tries when traversing to a point

If the robot has no progress in the last N moves then it may
be stuck. This function with determine this, and in turn cause
A* to be called to find a new path.
```

6.4.2.17 move()

```
def move (
              self,
              metres,
              target )
```

```
Move the robot given a distance and bearing.

Args:
    metres: The distance the robot to move in metres.
    target: The next node in the route. It's current target
        coordinates.
```

6.4.2.18 order_line()

```
def order_line (
              self,
              line )
```

```
Takes a LineString and orders the given points from the
starting boundary.

Ordering the line is needed to reduce the number of points to
the resolution needed for the A* grid. Only points which are
considered one object (less than a given distance away) are
kept as one line, points which are too far away are added to
'skipped', the process is repeated for these skipped points
until are all in a given line.

Args:
    line: The LineString needing to be sorted and segmented.

Returns:
    If no points are found None is returned, this needs to be
    checked as not all points are necessarily part of a line.
```

6.4.2.19 per_on_course()

```
def per_on_course (
              self )
```

```
Determines the percentage of values in the queue to be
on-course
```

6.4.2.20 print_finder_graph()

```
def print_finder_graph (
              self,
              target,
              b,
              matrix )
```

```
Prints the graph used for A*, mostly used to debug issues,
is mostly useful when changing the grid scale.

Args:
    target: The robot's intended target
    b: The min values used to map UTM to (0, 0) grid coordinates
    matrix: The A* matrix initialised with path weights and obstacles
```

6.4.2.21 remove_hidden()

```
def remove_hidden (
              self,
              points )
```

```
Remove any detected points that would not be visible in a
real application.

A point is considered hidden if another point is collinear and
closer to the robot.

Args:
    points: All points detected by the robot that are in range
        and being considered at this time.
```

6.4.2.22 seperate_lidar_ranges()

```
def seperate_lidar_ranges (
              self )
```

```
Movement depends on where the obstacle is detected by the
LiDAR i.e. if detected on left the robot will favour right
movement.

The LiDAR is split into ranges to give a balance of direction
favouring. Favouring a side too much may result in quicker
movement decisions, but often results in the mower getting
stuck more often.
```

The documentation for this class was generated from the following file:

- mower.py

Index