**Localization with Few Distance Measurements**

Barak Ugav 318336229

Algorithmic Robotics and Motion Planning Project

**Introduction**

A depth sensor is placed inside a known polygonal workspace with vertices. Consider the depth mapping in which is the position of the sensor and (namely ) is the direction of the ray emanating from the sensor and along which the distance to the workspace boundary is measured. Thus the configuration space (C-space for short), namely the parametric space of sensor placements, is three-dimensional.

**Problem statement**

We wish to devise a data structure such that, after preprocessing the workspace, will efficiently answer the following queries: Given a distance measurement as query, report all the possible sensor configurations that could yield such a measurement. In other words, we aim to compute 's preimage.

**Our Solution**

the solution can be viewed in the *Robot\_Localization\_with\_Few\_Distance\_Measurements.pdf*

**The Software**

The Software consist of two main parts, the logic implemented in C++ on top of CGAL, and a GUI implemented in Python. We give an overview on how to use the code and its main components.

User manual:

* GUI manual:  
  The Python GUI application consists of two main components: the scene designer (*scene\_designer.py*) and the localizator GUI (*localizator\_gui.py*). One can use the scene designer to design a room, which must be a simple polygon room, that can be loaded later in the localizator GUI. The localizator GUI allows the user to load a scene from a JSON file and query it with different single measurement values. The GUI is simple and need no more explanation.  
  Launching scene designer: *python src/py/scene\_designer.py*   
  Launching Localizator GUI: *python src/py/localizator\_gui.py*

|  |  |
| --- | --- |
| scene designer | localizator GUI |
|  |  |

* CPP software API:  
  The API class is the *Localizator* class (*Localizator.h*), which has two API functions, an *init* function which accept a vector of points which represent the boundary of a simple polygon room, and a *query* method which calculate for a given measurement all the points in the room a sensor might be in to measure such value and return the answer as collection of simple polygon which approximate the result.

*class Localizator {*

*Localizator();*

*void init(const std::vector<Point> &points);*

*void query(const Kernel::FT &d, std::vector<Polygon> &res) const;*

*};*

Scene examples:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Output result |  |  | Output result |
| 0.3 |  |  | 1 |  |
| 4 |  |  | 2 |  |
| 5 |  |  | 8 |  |

It’s interesting to see the complex shapes these calculations result in. We can see in the lower values of the shapes are pretty trivial, but the greater values are very non-intuitive.

Developer manual (CPP):

The code is developed on top of *CGAL* and *boost*, and the logic is focused mainly in the following files:

* *trapezoider.cpp*  
  The *Trapezoider* class accept a simple polygon room description by vector of points, and calculate all the trapezoids that exists in the room in any angle. It does so with the following steps:
  + Arrangement building from the points:  
    simple code, validates the room requirements and create a *CGAL* 2D arrangement.  
    (function c*reate\_arrangement()*)
  + Vertical trapezoids calculation:   
    Using CGAL vertical decomposition, the vertical cells are created, which will be the starting trapezoids in the later step. There are some compilation, as the CGAL decomposition may return a vertex or a face as the object above or below a vertex, and we are only interested in edges.  
    (function *init\_trapezoids\_with\_regular\_vertical\_decomposition()*)
  + Parallel rotational sweep to calculate all trapezoids:  
    We calculate all the events of the parallel rotational sweep and sort them by their angle and by their distance if the angles are equal (prefer further events first). Given the starting vertical decomposition and the associated trapezoids, we iterate through the events, maintaining the imaginaries rays’ heaps and updating, creating and terminating trapezoids as described in the paper. There are a lot of edge cases in the handling. After handling all events, we union the trapezoids that we cut by our rotational starting angle (for example a trapezoid that exists in angle interval will be split into two by our starting point ).   
    (function *calc\_trapezoids\_with\_rotational\_sweep()*)
* *trapezoid.cpp*  
  the *Trapezoid* class represent a trapezoid that exists in a single angle interval, and has a top and bottom edges, and left and right vertices. It’s responsible to calculate its own results for a given query , and calculating it’s maximum and minimum opening. The class API functions are:
  + *get\_bounds\_2d()*:  
    Calculate the maximum bounds of the trapezoid. This is used during result calculation to ensure the result points are contained in the valid area. The bounds are represented as a simple polygon with 4 vertices, the top and bottom edges’ endpoints.
  + *calc\_result\_m1()*:  
    Calculate all the possible points a sensor might be in the trapezoid and measure to the top edge. The way of operation of this function is described in details in the paper, but in simplicity the function first breaks the angle interval into two smaller intervals, by the angle perpendicular to the top edge. For each angle interval, the function approximates the curves of the right and left limiting vertices and builds from the curves a simple polygon. The simple polygon is intersected with the trapezoid bounds which may result in multiple (constant) number of simple polygons with no holes, which are appended to the result.
  + *calc\_min\_max\_openings()*:  
    Calculate the maximum and minimum opening of the trapezoid. This is later used to maintain an output sensitive data structure in the room scope localization. There is probably an analytic way of calculating these values, but for now this is implemented as a binary search on the maximum value, by using the *calc\_result\_m1()* method. This is slow, but it only affects the preprocessing time, which we can accept. The minimum opening is set to zero, but is not used until two measurement queries will be supported.
* *localizator.cpp*The *Localizator* class is used as a “main” for all the other components, and support the following API functions:
  + *init()*:  
    Accept a list of points representing a simple polygon room, initialized the underlying *Trapezoider* to calculate all trapezoids, calculate for each trapezoid its maximum and minimum opening and construct two data structures that will be used to answer queries:
    - Array of trapezoids sorted by their maximum opening.
    - Interval tree of trapezoids where the intervals values are [*min\_opening, max\_opening*].
  + *query(d)* :  
    Calculate all the points a sensor might be after knowing it measured at some wall. This is done in an output sensitive manner using the array sorted by maximum opening by performing a binary search on the array, and calculating the result only for trapezoids we know will have non-empty output.
  + *query(d1, d2)* :  
    **not implemented yet** Calculate all the points a sensor might be after knowing it measured at some wall and and the exactly opposite orientation. This is done in an output sensitive manner using the interval tree, by querying the tree for trapezoids that have opening exactly at range , and calculating the result only for trapezoids we know will have non-empty output.

Additional files:

* *defs.h*:  
  Definitions files, used to choose CGAL kernel and other options. Can be used to speed up performance.
* *localizator\_daemon.cpp*:  
  Wrapper daemon for the Localizator that accept commands through JSON files. This is used to communicate with the Python GUI.
* *json\_utils.cpp*:  
  Some read/write function from/to JSON.

Additional notes:

Throughout the code we avoid using double, sqrt, sin, cos, ect. This is done to keep the best precision possible by using only +,-,\*,/. In some cases, we must use inaccurate functions, but we try to avoid it. This means for example to represent a trapezoid angle interval, instead of storing angle values, we store direction vectors, which we can calculate only by the accurate operations.

Developer Manual (Python):

The GUI is built on top of the GUI used in class. The scene designer is simple and trivial, also the GUI for the Localizator. The Localizator API (Python) is a bit trickier, as the communication is done via files and its async. This is done by few files, a command file used to pass commands to the CPP daemon, an ack file used to get an acknowledgment from the daemon it finished processing the last command, and an output file passed to the daemon for it to write the result JSON into. In addition, the daemon output is logged into a log file.

Disclaimers:

The software is not perfect, there is still a lot of work to put into it, especially when handling the degenerated cases, for example more than 2 collinear vertices of the room.

When testing this, please use simple scenes with no 3 collinear vertices, I think the other cases are covered, but its hard to tell, go easy on it. I will provide a poll of scenes the software works for but you can try your own of course.

In addition, the GUI is a last-minute addition, the heart of the software is the CPP, and therefore the GUI and its connection to the CPP through files is not perfect.

I put a **huge** effort to solve all of the math involved, writing the code and making this project work, please consider this when evaluating the flaws of this project.