Prof. Dr. Maren Bennewitz
M.Sc. Padmaja Kulkarni

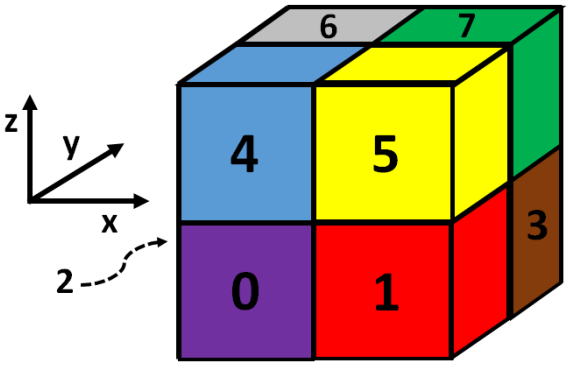
Bonn, 9 May 2019

Humanoid Robots
Exercise Sheet 5 - 3D Representations

Exercise 6 (10 points)

Assume that we have a point cloud of a 3D environment, for example retrieved from a laser scanner or a 3D camera. Now we would like to represent this data as an octree.

The octree algorithm starts with a complete cube and subdivides it recursively into eight smaller cubes. It only splits cubes that are neither fully occupied nor fully empty until all cubes are either occupied or free. When inserting a new point, you keep splitting the cubes until you reach a pre-defined minimum size of a 3D cube, i.e., a certain depth within the octree. The order of the subcubes is given in the figure on the right.



In the source code for this exercise, we represent a 3D cube as an instance of class `Node`. Each node has a pointer to the parent node and 8 pointers to child nodes for representing the tree structure.

Exercise steps:

a) The method `Octree::insertPoint(const Eigen::Vector3d& point)` is already given in the code and contains the general algorithm for inserting a new point into an octree. Inspect the code to find out how the algorithm works.

b) Implement `Node::findIndex(const Eigen::Vector3d& point)`, which is a member function of class `Node`, and is responsible for returning the index (0–7) of the child cube that will contain the given input `point` if we split that node.

c) Implement `Octree::findNode(const Eigen::Vector3d& point)`, which is a member of class `Octree` and traverses the tree from the root to find the existing leaf `Node` that contains a given point.

d) Implement `Node::split(const Eigen::Vector3d& point)` which is a member function of class `Node`, and splits the node into eight child nodes. It returns the child node containing the given point, and sets its status temporarily to `OCCUPIED`. The status of the other empty child nodes are set to `FREE`. The status of the parent node is set to `MIXED`, i.e. neither completely occupied nor completely free.

e) Implement `Node::merge()` which tries to merge the current node in case all children are either occupied or free. It sets the status of the parent to the common status of the children and deletes all the children nodes.

The `scripts` folder contains a Gnuplot script for rendering an interactive 3D view of the octree that your program generates. You can move the 3D view with your mouse. Alternatively, you will find a top-down view and a rotating 3D animation in the Wiki.

Exercise 7 (10 points)

In this exercise you will implement the signed distance function to estimate a 2D map from a series of laser range measurements recorded by a robot while travelling through an environment.

Each measurement consists of the 2D robot position and a list of 2D laser end points indicating where the laser beams hit obstacles. The existing code already loads the data into the following struct:

```
struct Measurement {
    Eigen::Vector2d robotPosition;
    std::vector<Eigen::Vector2d> laserPoints;
};
```

All input data and constants in this exercise have already been converted to grid cells in the map coordinate system, so there is no need to convert between units or coordinate systems.

a) Implement the Euclidean distance between two points in `calculateDistance()`.

b) Implement the truncated distance function in `truncateDistance()` according to Fig. 1(a).

c) Implement the weighting function in `calculateWeight()` according to Fig. 1(b).

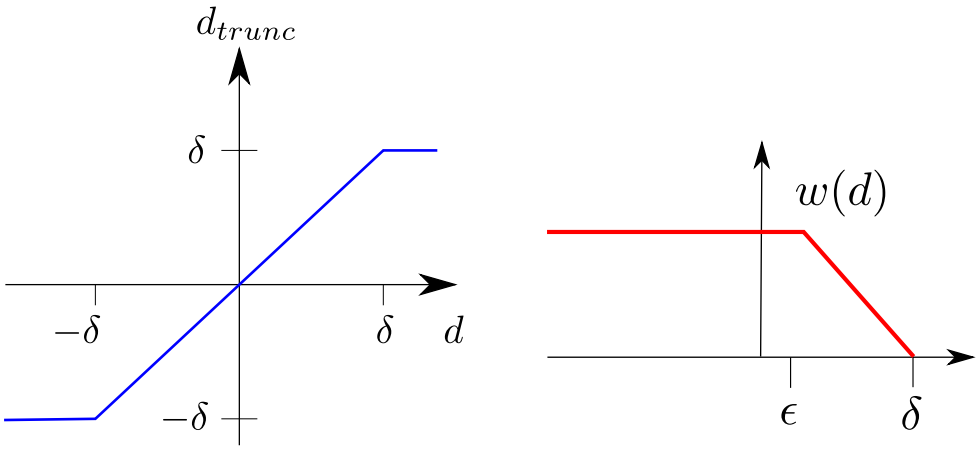
d) Update the signed distance value of a grid cell in `updateMap()` according to the rule

$$D = \frac{WD + wd}{W + w} \tag{1}$$

W is the weight of the current cell before the update, w is the weight of the current measurement. D is the signed distance function value of the grid cell before the update and d is the signed distance function value from the current measurement.

e) Update the weight value of a grid cell in `updateWeight()` according to the rule

$$W = W + w \tag{2}$$



(a) truncated signed distance function (b) weights as a function of the signed distance

Figure 1: Functions for this exercise. The constants δ and ϵ are given in the code.

f) Implement the method `integrateLaserScan` that integrates a new laser measurement into the map and updates the weights associated with the grid cells.

1) Iterate over all laser points in the current measurement.
2) Use the existing method `bresenham(A, B)` to get a list of cells that lay on the straight line through the points A and B. This modified Bresenham algorithm gives you the cells not only between the points A and B, but also beyond B up to $B + (B - A)$. The remaining two arguments `numRows`, `numCols` are the number of rows and columns of the map matrix.
3) For each point returned by the Bresenham algorithm compute the signed distance function and weight with the methods defined above and update the grid cell and the associated weight. Be careful with the following implementation issues:
   - Following common conventions, the x axis corresponds to the columns of the map matrix and the y axis corresponds to the rows of the map matrix.
   - Update a map cell only if the weight of the current measurement is greater than 0, otherwise Equation (1) will cause division by zero errors.
   - Make sure that you cast row and column indices from `double` to an integer type when accessing elements of the map and weight matrices.

If you have Gnuplot installed on your computer, then you can get an interactive 3D view of the resulting signed distance map as well as a 2D view of the walls and robot trajectory by executing the scripts `scripts/plot_3d.gp` and `scripts/plot_map.gp`.

Deadline: 16 May 2019, 11:59 am