OCTOMAP: Independent Study Report

Siva Pabbineedi (working with Nataliya)
gapa2065@colorado.edu

University of Colorado Boulder

1 Technical Description

This section explains some of the core aspects of the OctoMap library indicating the changes made to overcome its limitations. It serves to explain the practical logic behind the data structure, complementing the comments added to the code.

Some variables and their values necessary to understand the code are as follows:

- **resolution**: A resolution of 0.01 leads to the smallest voxel size of $1cm^3$.
- **Tree-depth**: The number of times we divide the space into 8 octants. Restricting this to some value (e.g. 16) leads to efficient calculations.
- **Tree-max-val**: Half the number of voxels present in the lowest level of the tree considering just one axis (32768).

All the explanations below use a single axis instead of three to employ easy diagrams for better visualization. The 3 axes are independent for all calculations in the code. Looking at the examples, the diagrams, and changing values in `getit.cpp` are the best ways to understand these.

- **coordToKey(coordinate, depth)**: This converts a given coordinate to the corresponding key at a given depth. Keys are values from 0 to 65536 that enable efficient binary operations. One can think of them as buckets that coordinate values fall into. Refer 1.

  `coordToKey(163.84) is 49152 at lowest depth.`

  `coordToKey(200, 2) is 57344 as 200 falls into bucket 57344 at depth 2.`

  `coordToKey(200, 3) is 53248 as 200 falls into bucket 53248 at depth 3.`

- **keyToCoord(key, depth)**: This converts a key at a given depth into a coordinate corresponding to the key's center.

  `keyToCoord(32768, 1) is 163.84`

  `keyToCoord(6000, 1) is -163.84`

- **adjustKeyAtDepth(key, depth)**: Adjusts a single key value from the lowest level to correspond to a higher depth. Refer 2.

  `adjustKeyAtDepth(39000, 1) falls into 49152 at depth 1`

- **Size-lookup-table**: This is an array of voxel sizes for all depth levels.

  `Values: 655.36, 327.68, ..... , 0.02, 0.01`

- **computeChildKey(pos, center-offset-key, parent-key)**: Computes the key of a child node while traversing the octree, given child index and current key.

  `center-offset-key = (tree_max_val >> (1 + 1)) which is 8192.`

  `computeChildKey(1, center_offset_key, 16384) is 24756 - right child key`

  `computeChildKey(0, center_offset_key, 16384) is 8192 - left child key`

  Note that center-offset-key is the value added or subtracted to the parent-key to get the child keys.

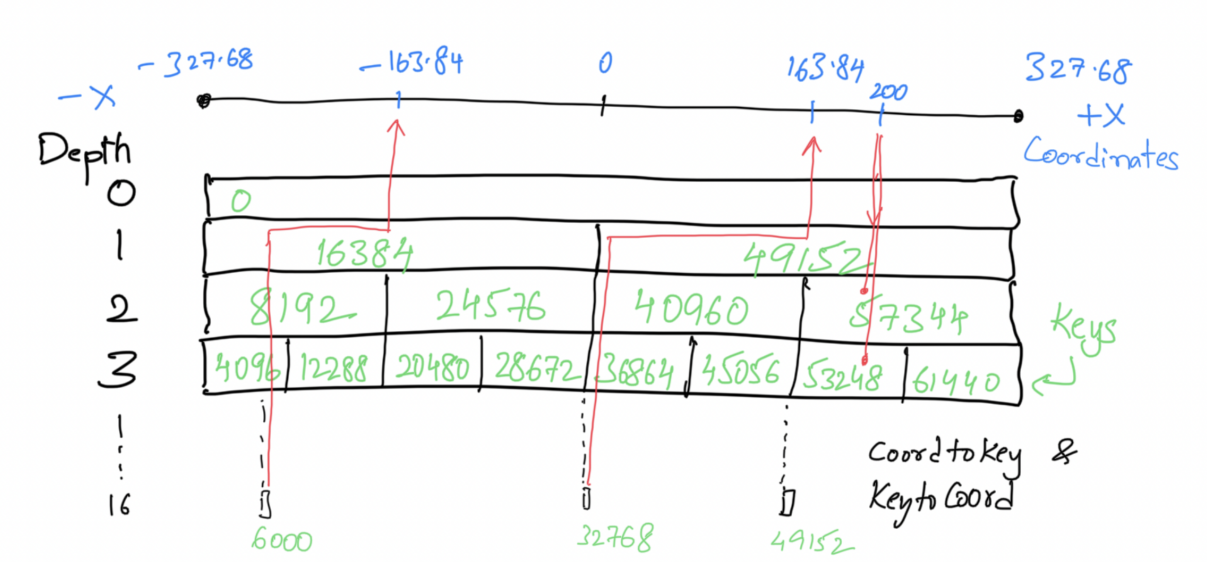

Figure 1: coordToKey and keyToCoord

- **computeChildIdx(key, depth)**: Generates child index (between 0 and 7) from key at given tree depth. Note that the depth here is actually (16 - depth) for the calculation.

  In simpler terms involving only one axis, given a key at the lowest level, this checks whether it falls in the left or right half of the bucket at a given depth.

  `computeChildIdx(24576, 16 - 1) is 0 as it falls on left at depth 1.`

  `computeChildIdx(24576, 16 - 2) is 1 as it falls on right at depth 2.`

  `computeChildIdx(24576, 16 - 3) is 1 as it falls at center which is included in right at depth 3`

  `computeChildIdx(24576, 16 - 4) is 0 as it falls on left at depth 4.`

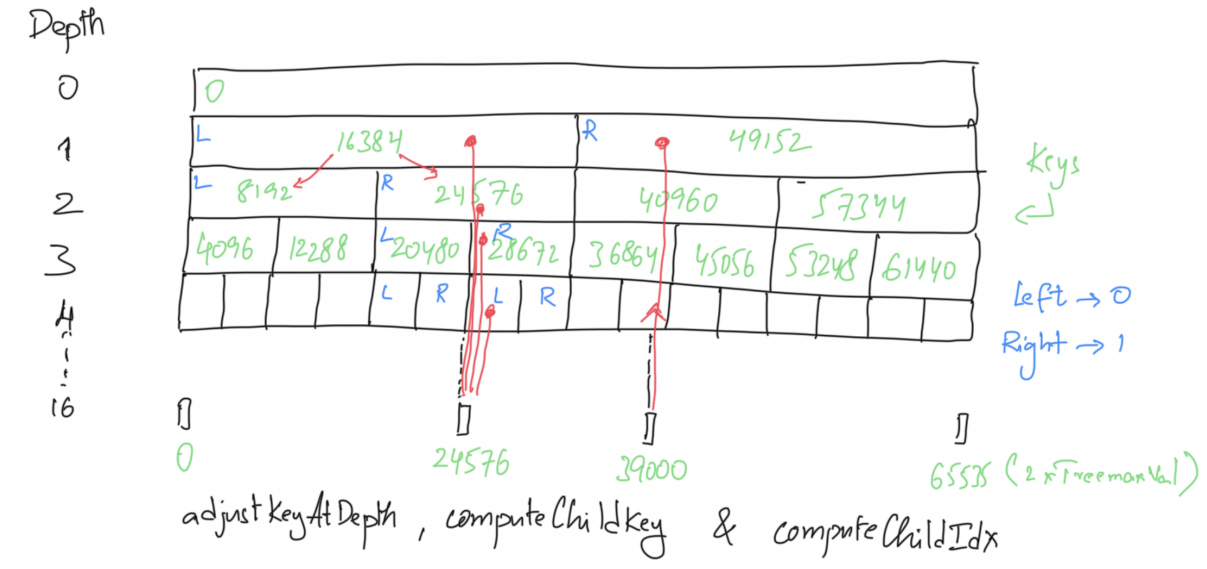

Figure 2: adjustKeyAtDepth, computeChildKey and computeChildIdx

- **Types of occupancies**: Occupancy can be free, occupied or unknown. Free and occupied nodes have occupancy probability lesser than or greater than some thresholds. Unknown space is not represented in the tree (NULL).

- **Leaves**: Nodes at the lowest level without children are leaves. Pruned nodes at upper levels are also leaves. Nodes with 8 identical children can be pruned to make all children NULL to save memory. This is only done when all the children have the same occupancy values and cost values. The occupancy of a pruned node is same as the occupancy of any of its children, while the cost is the sum of the cost of all children. In general, the cost of a parent is the sum of costs of all its children.

- **Datatype for Costs:** A cost value of 1 in each leaf node at the bottom-most level would add up to 65536^3 at the root. A double can easily hold this value.

- **Vector3 or Point3D**: Vector3 represent a 3D point (x, y, z) in a point cloud. Adding an optional cost to this class would be the best way to ensure backward compatibility. Note that this cost is assigned to a leaf node in the tree only when the node is considered occupied. Otherwise, the cost of the leaf node stays 0.

  NOTE: The recursive functions below are more complex than the ones above. You might need to refer to the comments I added to the code to understand them clearly.

- **pruneNode(node):** A node is collapsible when all its children have the same occupancy probabilities and cost values. In this case, we can delete all the children and put any child occupancy and the total children's costs into the pruned node.

- **Why is pruning necessary?**: The initial idea was to have a complete cost tree without concerning ourselves with pruning complexities. One can still query upper-levels to avoid time-consuming queries. But $2^{17} - 1$ leaf + inner nodes would not fit in addressable memory.

- **expandNode(node):** This is the opposite of pruneNode where we reform all the children because one of them has some update. We put pruned node occupancy in all children and divide the cost among eight children.

- **search(key, depth)**: Search a node at specified depth given an addressing key. This process starts from the root and navigates down to the given level (or the lowest) using `computeChildIdx` to find, at each level, which child the search key should be in. If we reach the tree-depth, we return the leaf. Otherwise, we return the pruned leaf node or NULL value for unknown space.

- **deleteNode(key, depth) and deleteNodeRecurs(...):** Delete a node (if exists) given an addressing key. If the search hits a pruned node, it has to be expanded to delete one particular child. If the deletion of the node causes its parent to have no children, it should be recursively deleted. Occupancy and cost updates need to be performed after deletion so that parent nodes have correct values.

- **insertPointCloud() and computeUpdate()**: Given a point in the point cloud, these methods compute keyrays to mark every key from sensor origin to the point as free and the point as occupied. After making the free and the occupied key sets for the whole point cloud, updateNode() is called on each of those keys.

- **updateNode() and updateNodeRecurs()**: These are the functions that integrate occupancy and cost measurements. Updates are made only at the tree-depth level. If we run into a pruned node, it is expanded as we have to update one of its descendants. Once the update is done, we prune ancestors if we can and make sure the occupancy and cost values of ancestors are updated correctly. The cost is set to the cost of the point in the point cloud only if the node is occupied.

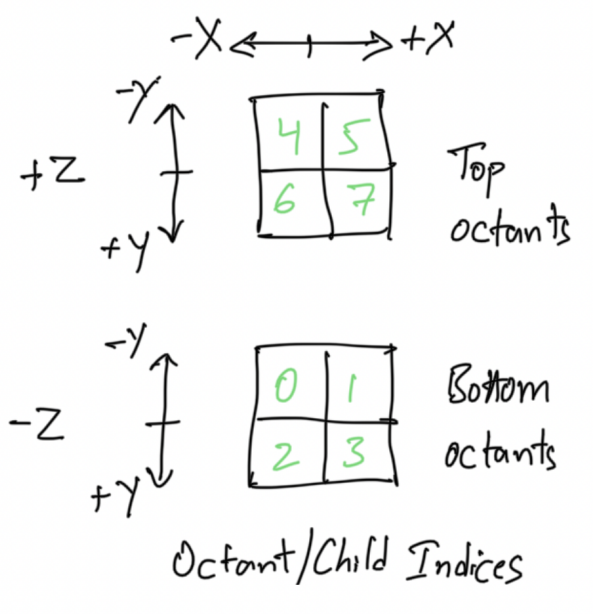

Figure 3: Child indices