7.0.1 (W) Chaotic Reach set

Motivation: Design of calculation method for *Reach Set Approximation* guarantying high *Maneuverability.*

Background: There is *Coverage Ratio* property of *Reach Set* (def. **??**). It has been shown that creating *Reach Set* via *greedy approach* is not feasible due the *Scaling Factor. Contracted Expansion* (sec. **??**) is enabling to apply selection criteria while building *Reach Set* in given *Cell.*

The *Cell* $cell_{i,j,k}$ has a center and walls from UAS viewpoint: front wall , back wall (for $layer > 1$), top wall, left wall, right wall, bottom wall. It is expected that trajectory leading close to one cell walls will continue to different cell, increasing chance to obtain more *Unique Footprints.*

Expansion Constraint Function Implementation (alg. 7.1) is based on simple principle: *Select candidate Nodes which are closest to outer walls of Cell, with unique footprint.*

Tuning Parameters : *Proximity to Cell outer wall* gives good chances to break into other rows or columns in *Avoidance Grid. Unique footprint* guarantees future *Unique Footprint* after appending Trajectory by *Movement application.*

1. *Considered Footprint Length* - how much last cells in footprint should be considered in unique path track, minimal value 1, default value 3, maximal value ∞. If you want to generate non redundant trajectories use ∞, it will consider full footprint.

2. *Spread Limit* - upper limit of candidates which are going to be select for further expansion, minimal value 1, default value *Count of unique Moves in Movement set*, maximal value ∞. If more than default values is selected the algorithm will generate *redundant trajectories.* If less is selected then some trajectories are omitted and *Coverage Rate* decreases sharply.

Step: Initialization initialization of *candidate* array (return value), *leftovers* array (return Value). Node array *passing* is populated with *Nodes* which represents *end node of Trajectory* and the tip of *trajectory is constrained in* $\text{cell}_{i,j,k}$.

Step: Evaluate best trajectories with unique Footprints following steps are executed:

1. *Best Performance Map* is created with *footprint* as key set element to ensure footprint uniqueness.

2. *Wall distance* for *test node* is calculated as a closest trajectory portion distance to *top, bottom, left, right* wall of cell $cell_{i,j,k}$

3. *Footprint* for *test node* is created with maximal length given by *Footprint Length* tuning parameter.

4. *Existence and Performance Test* is executed to ensure that best performing node is selected. If there is not key entry in the *Best Performance Map*, then new entry for *Test Node* is created. If there is key entry, the performance of *Old Node* and *Test Node* is compared and better is stored.

Step: Select candidates is executed on *Best Performance Map* records using *Wall distance* as pivot parameter, ordering by closest proximity and limited by *Search Limit* tuning parameter. The *Leftovers* are difference set between *Passing Nodes* and *Candidate Nodes.*

Algorithm 7.1: Expansion Constraint function for *Chaotic Reach Set Approximation*

```
Input             : Node[] stack, Cell cell_{i,j,k}
Tuning Parameters: int+ footprintLength, int+ spreadLimit
Output            : Node[] candidates, Node[] leftovers

# Initialize structures;
Node[] candidates = [], Node[] leftovers=[];
Node[] passing = cell_{i,j,k}.getFinishingTrajectories(stack);

# Select best performing trajectories with unique footprint;
Map<Footprint,Node> bestPerformanceMap;
for Node test ∈ passing do
    wallDistance= test.minimalDistanceToWall(cell_{i,j,k})];
    footPrint = test.getFootprint(lastCells = footprintLength);
    if bestPerformanceMap.contains(footPrint) then
        old = bestPerformanceMap.getByKey(footprint);
        oldPerformance= old.minimalDistanceToWall(cell_{i,j,k});
        if oldPerformance > wallDistance then
            bestPerformanceMap.setByKey(footprint,test);
        end
    else
        bestPerformanceMap.setByKey(footprint,test);
    end
end

# Select best performing nodes up to spreadLimit count;
candidates = bestPerformanceMap.select(count =
 spreadLimit).orderBy('wallDistance','Ascending');
leftovers = passing - candidates;
return [candidates,leftovers]
```

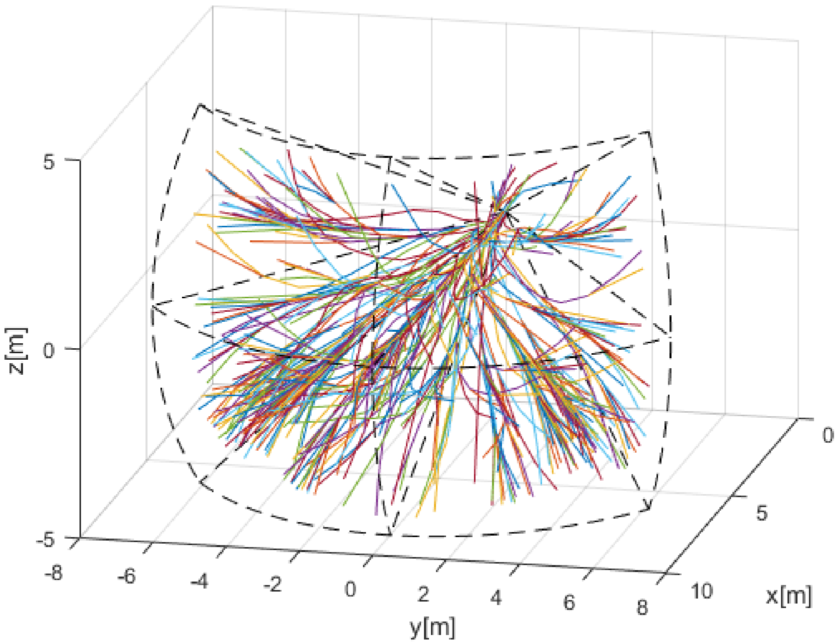

Figure 7.1: *Chaotic* reach set *approximation*.

Example:

Pros and Cons: It can be seen from example (fig. 7.1) that *Chaotic Reach Set Approximation Method* (alg. 7.1) generates a lot of *turning* and *shaky trajectories*.

High Coverage Ratio (~ 0.9) is provided, while keeping *low node count* ($\sim 30\%$). The calculation complexity scales linearly with grid size.

Absence of *Smooth Trajectories* disqualifies *Chaotic Reach Set Approximation* to be used for *Navigation*. This type of reach set is feasible for *Avoidance*, because it contains variety of manuevers.

7.0.2 (W) Harmonic Reach set

- Introduction of smoothness performance criterion
- Smoothness formula for our *movement set* - Gaussian spread
- Harmonic reach set example

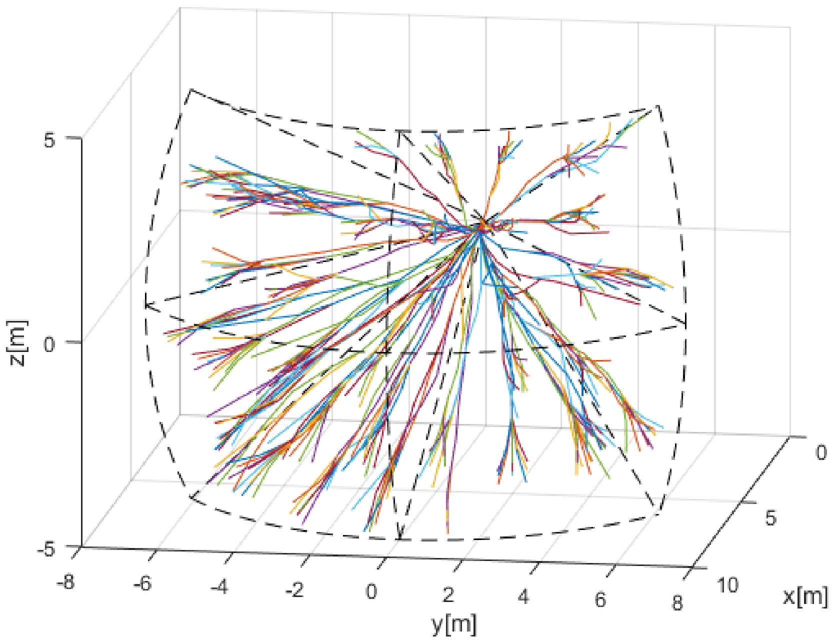

Figure 7.2: *Harmonic* reach set *approximation*.

7.0.3 (W) Combined Reach set

- Used as *Avoidance Grid* for emergency avoidance
- Tree merge of chaotic and harmonic reach set
- Combined Reach set example

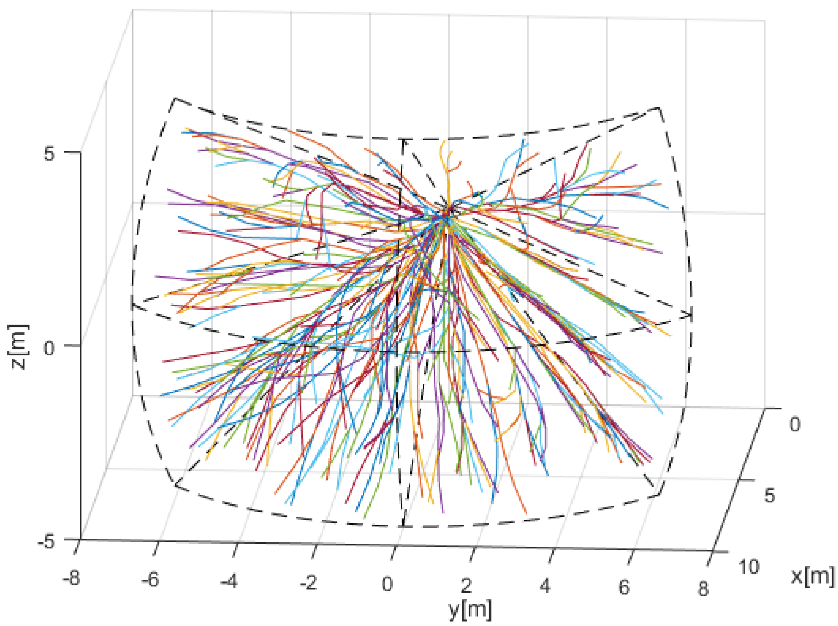

Figure 7.3: *Combined* reach set *approximation*.

7.0.4 (W) ACAS-X imitation Reach set

- Used as *Navigation Grid* because trajectories are compliant with separation mode in controlled airspace
- Criterion function for separation introduction
- Explain ACAS separation modes
- Picture for separation modes

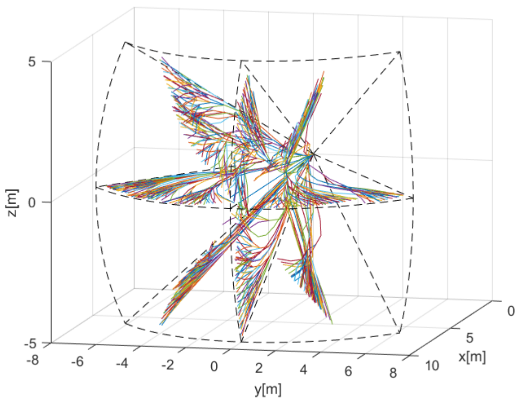

Figure 7.4: Full.

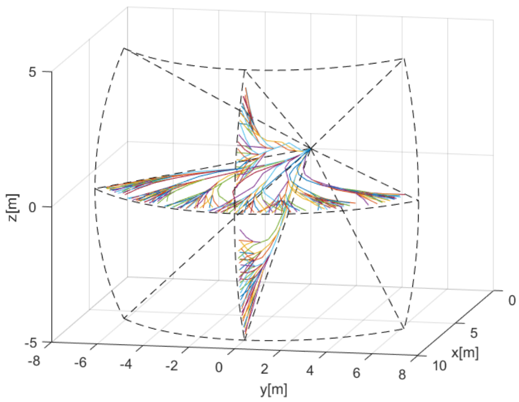

Figure 7.5: Horizontal-Vertical.

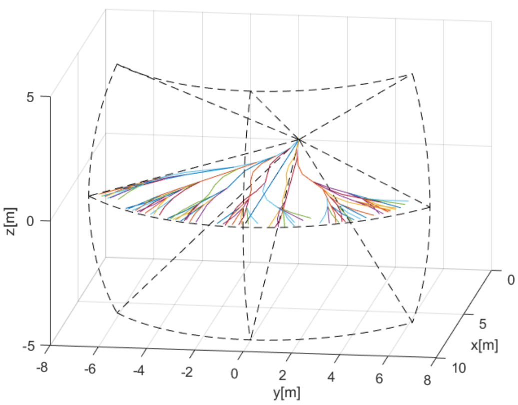

Figure 7.6: Horizontal-only.

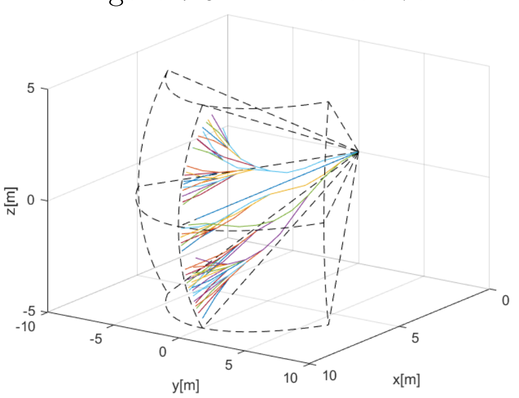

Figure 7.7: Vertical.

Figure 7.8: ACAS-X imitation *reach set* approximation for various *separation modes*.