**6.4 Reach Set Approximation**

**Motivation:** *Reach set* is a strong tool for *Obstacle Avoidance* because it contains all possible *avoidance maneuvers*. The current implementations (sec. **??**) have the following flaws:

1. *Realistic approximation* - *nonlinear systems* or *heavily constrained systems* cannot be approximated well by *linear continuous-time Reach Sets*.
2. *Finite count of possibilities* - continuous-time *Reach Set* contains infinite possibilities for *avoidance maneuvers*; the SAA system demands conflict resolution infinite time.
3. *Computationally feasible data structures* - binding related properties seem problematic because *continuous- time reach sets* do not have a unique identifier of maneuver, trajectory nor segment.

**Proposed Solution Features:** Our Reach set Estimation method will provide the following features:

1. *System Control Interface* - implemented via *Movement Automaton*, requiring only a *discrete command chain* to approximate system behavior.
2. *Finite count of possibilities* - finite number of elements in *Reach set* will enable *scalable* calculation.
3. *Computationally feasible data structures* - approximation of Reach set as a set of trajectories, each trajectory can be split into a finite number of segments. Each element will have a unique identifier enabling both-side property binding.
4. *Computationally feasible data-structures* - some specific behavior, like horizontal/vertical separation, or maneuver shape can be encoded into different types of reach set approximation algorithms.

# 6.4.1 Trajectory Set Approximation of Reach Set

**Discretization of Reach set:** There is a need for a discrete finite *Reach Set approximation* to enable *Avoidance Strategy Evaluation* infinite time. Replacing *Continuous Control Set Inputs(t)* by *Movement Automaton* is feasible:

**Definition 1** (Reach set Approximation by Movement Automaton)**.** A trajectory *(def.* **??***) for system state*˙ = *f*(*time,state,input*) *under control of the movement automaton*

MA *is given as execution of movement buffer (def.* **??***) with an initial state of system state*0*.*

*Therefore notation Trajectory*(*state*0*,buffer*) *is used.*

**The Complete Reach Set** *(6.1) for system with initial state state*0 *with existing* control strategy *control*(*time*) ∈ *Controls*(*time*)*. for time τ > time*0*.*

*ReachSet*(*τ,time*0*,state*0) = [{*state*(*s*) : *control*(*s*) ∈ *Controls*(*s*)*,s* ∈ (*time*0*,τ*]}

(6.1)

**The Reach Set Approximation by Movement Automaton** *(6.2) of the system under the control of the movement automation* MA *consist from the set of trajectories Trajectory* (*state*0*, Buffer*)*, which are executed in constrained time τ > time*0*.*

  *duration*(*buffer*)

 

*ReachSet*(*τ,time*0*,state*0) = *Trajectory*(*state*0*,buffer*) : ≤ (6.2)

 (*time*0 − *τ*) 

*Note. Reach Set Approximation* (def. 1) is a subset of *Full Reach Set* (def. **??**) in continuous space R*n* it inherits all important properties, like *Invariance* [1].

*Discretization* of *Reach Set* have been achieved leaving us with a *finite count* of *Trajectories*, instead of *Infinite subspace or* R*N*

**Approximated Reach Set Containment:** The *Approximated Reach Set* introduced in (def. 1) is constrained only by *future expansion time τ*. UAS makes space assessment in *Avoidance Grid*. There is no point to consider Trajectories outside of *Avoidance Grid*

**Definition 2** (Contained Approximated Reach Set)**.** *For a pair (state*0*, AvoidanceGrid*0*) at time time*0 *and* prediction horizon *τ* = ∞ *there is* Contained Reduced Reach Set:

    *time*0*,* *Trajectory*(*...*)

   ∀*segment* ∈ *AvoidanceGrid*0*,* *ReachSet* *state*0*,*  = ∈ :

   *ReachSet*(6*.*2) *segment* ∈ *Trajectory*(*...*) 

*AvoidanceGrid*0

(6.3)

**Properties:** Container Approximated Reach Set *contains only trajectories where all segments belong to* Avoidance Grid*, there are following functions:*

1. Membership function *for any* Trajectory *in* Constrained Reduced Reach set *returns* Ordered Set *of* Passing Cells*.*
2. The cost function *for any* Trajectory Portion *in* Constrained Reduced Reach Set *return* Cost of Execution

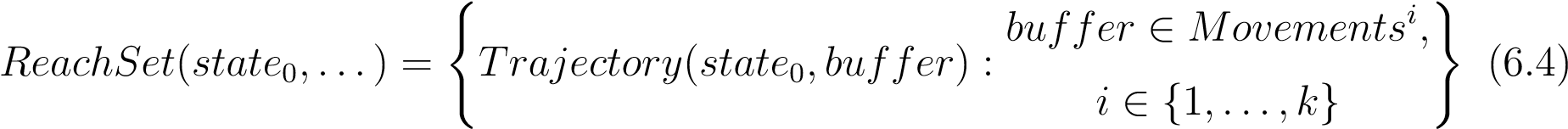
**Passing cell:** Cell *of* Avoidance Grid *which has some intersection with Trajectory.*

*Note. It contained Reduced Reach Set* (eq. 6.3) which is contained in the *Avoidance Grid* and have a *Membership Function* enable Property transition between Reach set and *Avoidance grid*.

*Example:* Visibility from cells along *Trajectory* can be gathered to calculate *Trajectory‘s* feasibility.

**Reach Set Pruning:** There is a need to implement *Set Difference* between *Reach Set* and *Constraint Set*. Constraint Set can be an *Obstacle Set* from *Known World* (sec. **??**) and other different constraints.

**Reach Set Trajectory Tree:** (6.4) *Any Reach Set* where *Control Strategy Constraint* is implemented as *Movement Automaton*, with defined *Movements* set and for single initial *state*0. The *Reach Set* is given as discrete tree with root *Trajectory*(*state*0*,*∅).



For each *Trajectory Segment*, there exists *intersection function* which evaluates as true if there exists at least one point in *Segment* which belongs to *Constraint Set*. Formally:

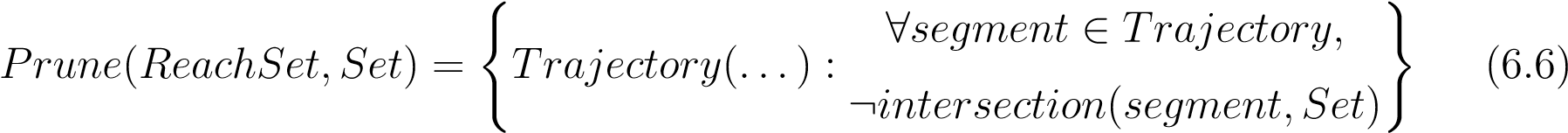


∃*point* ∈ *segment,*

 : *true*

*intersection*(*segment,Set*) : *point* ∈ *Set* (6.5) *Otherwise* : *false*

**Definition 3** (Pruned Reach Set)**.** *For* Reach set *represented as* Trajectory Tree *(eq. 6.4) and some constraint set (Set) where exist* intersection function *(eq. 6.5). The* Pruned Reach set *is given as follows:*

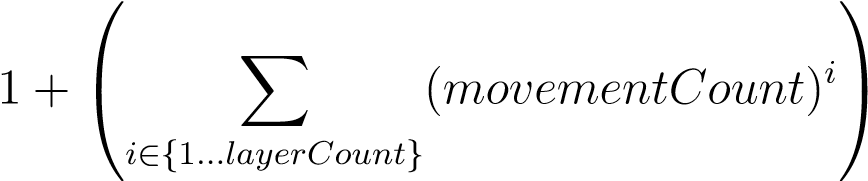


*Note.* Pruning(def. 3) [2] is applied multiple times for various *Constraints Set*.

Example of *Approximated Reach set Calculation* (def. 1), *Reach Set Containment* (def. 2), and, *Pruning* is given in [3].

# 6.4.2 Distinctive Properties of the Trajectories

**Motivation:** The need to Make *Reach Set* scalable approach. This may be a problem due to the *Expansion rate*. *Reach set* represented as a *Trajectory Tree* (eq. 6.4) for Avoidance Grid with *layer count* and Movement automaton with *movement count*, the *Node count* is given as:

 (6.7)

*This scaling* is not feasible for *Avoidance Grid* with many layers (*<* 10) or *Movement Set* with many movements (*<* 9). There is a need for *Reduced Reach set calculation*.

**Performance Criteria:** The scaling factor (eq. 6.7) shows that there are going to be many trajectories. The main point is that not every trajectory in *Reach Set* is giving us *maneuverability advantage*. Our expectations lie in following *Performance Requirements*:

1. *Reach set* must *Cover* maximum of the *possible unique maneuvers* in *Avoidance*

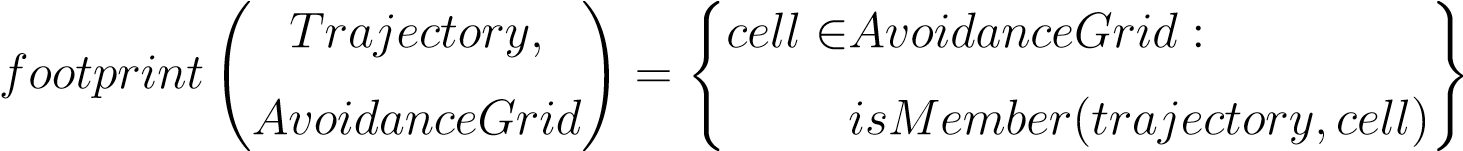
*Grid*.

1. *Trajectories* in *Reach Set* should be smoothest possible to prevent cargo damage / UAS wear.

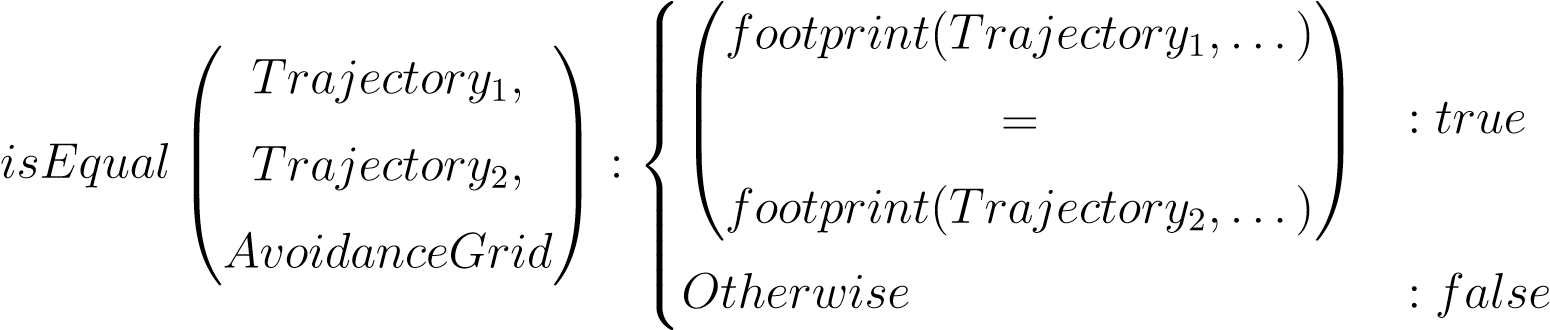
**Trajectory footprint:** Discrete space of *Avoidance Grid* is organized in cells. *The cell* is a minimal space portion accessible by *property binding*. There is a need to know if two trajectories contribution to *Maneuverability* in this environment.

Each trajectory passes through space in *Avoidance Grid*. If there exists a method to extract unique identifier for each *trajectory passed cells*, we can compare two trajectories *Coverage* in *Avoidance Grid*.

**Definition 4** (Trajectory footprint)**.** *For* Trajectory *from* Reach set *(def. 2) defined for* Avoidance Grid *has membership function.* Membership Function *returns* ordered set of passing cells*:*

 (6.8)

*Then we can define equality function for Trajectory*1 *and Trajectory*2*, as the comparison of their footprints in common* Avoidance Grid *as follow:*

 (6.9)

*Note.* Depending on *Movement Automaton‘s* movement set and *Avoidance Grid* parameters, there can be multiple *trajectories* which are equal.

**Coverage set:** Now it is possible to create a set of unique *trajectory footprints* due to *footprint function* (eq. 6.8). Similarly, there is a possibility to create *Reach set skeleton* containing unique trajectories, by using *equality function* (eq. 6.9). *Coverage set* is sufficient for now.

**Definition 5** (Coverage Set)**.** Coverage set *(6.10) is defined for* Avoidance Grid *and* Reach Set *pair as a set of unique* Trajectory footprints*:*

! ( ! )

*AvoidanceGrid, Trajectory,* ∀*Trajectory*

*coverages* = *footprint* :

*ReachSet AvoidanceGrid* ∈ *ReachSet*

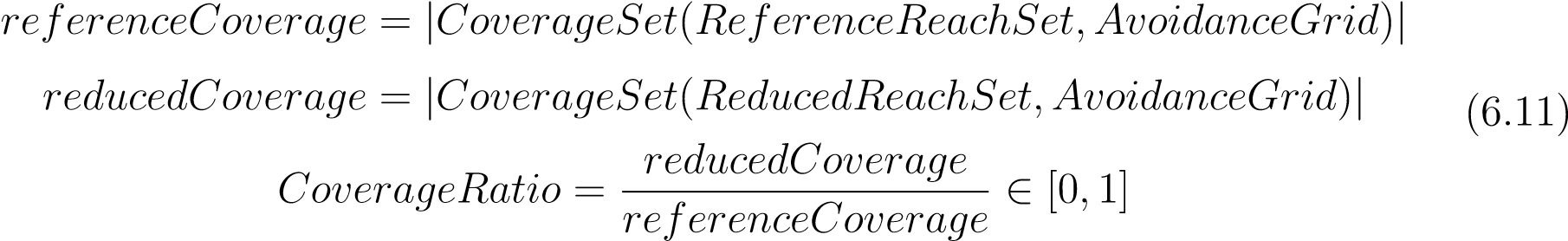
(6.10)

**Coverage set properties:** Trajectory footprint (eq. 6.8)is not a *bijection*, neither *injection* for *ReachSet* → *CoverageSet*. This implies the following properties:

1. Equal *Reach Sets* in same *Avoidance Grid* have equal *Coverage Sets*.
2. Equal *Coverage Sets* does not imply *Reach Set* equality.
3. For two Coverage Sets, there is a possibility to compare their member count to create coverage ratio.

The second *Property* gives us a proposition that there is a possibility of *Reach Set Reduction* without losing *Coverage*.

**Definition 6** (Coverage Ratio)**.** Coverage Ratio *is a ratio of* Coverage Set Member Count *between two* Reach Sets*. Reach set with a* lesser count of unique Trajectories *is considered as* Reduced Reach Set*. Reach set with* greater Count of unique Trajectories *is considered as* Reference Reach Set*.*



*Note. Reference Reach Set* is usually *Full Reach Set* containing all possible trajectories in space contained by *Avoidance Grid*. In case *Full Reach Set* cannot be computed, Avoidance Grid is too large, most complex *Reach Set* is used as *Reference Reach Set*.

**Trajectory smoothness:** Trajectory other than straight line have some changes in *UAS* heading.

The goal is to minimize *Maneuvering* of UAS, because:

1. *Every Heading Change* needs to be reported to *UTM*.
2. *Sharp Maneuvering* can damage cargo/wear UAS.
3. *Often course changes* make *Intruder prediction* harder for other Civil General Aviation.

For this purpose, *Smoothness Metric* needs to be applied for *Reach Set* or *Trajectory*. In the case of *Movement Automaton Control,* two distinguish *Movement Sets* are introduced: *Smooth* and *Chaotic* movements set with the following properties:

*MovementSet* = *SmoothMovements* ∪ *ChaoticMovements*

*SmoothMovements* ∩ *ChaoticMovements* = ∅ (6.12)

|*SmoothMovements*| *>* 0*,* |*ChaoticMovements*| *>* 0

Then *Smoothnes clasificator* for *Trajectory*(*initialState,buffer*) can be defined as *isSmooth* and *Smooth Movement Counter* function as *smoothCount* like follow:



*movement* ∈ *SmoothMovements* : 1

*isSmooth*(*movement*) =

*movement* ∈ *ChaoticMovements* : 0

(6.13)

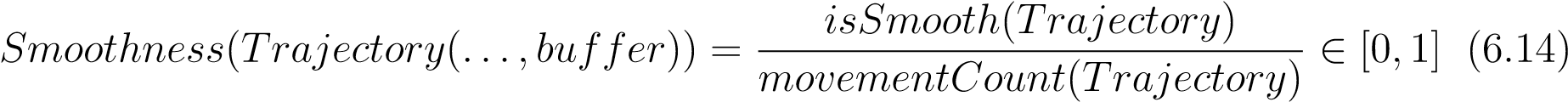
X

*isSmooth*(*movement*)*,*

*smoothCount*(*Trajectory*(*...,buffer*)) =

∀*movement* ∈ *Buffer*

**Definition 7** (Smoothness Rating for Trajectory)**.** Smoothness *for trajectory generated by* Movement Automaton *for some* Initial State *with some* Movement Buffer*, under the assumption of* Smooth and Chaotic Movement Set *split (eq. 6.12), with existing* classification *and* counter *functionals (eq. 6.13) is given as follows:*



*For* Trajectory *with buffer* = ∅ Smoothness *is given as 1.*

# 6.4.3 Heuristic Trajectory Tree Building

**Motivation:** *Purpose* of *Navigation* is to move forward to *Goal Waypoint* in *Mission*.

*Structure* of *Avoidance Grid* is designed to enable *forward* and *turning* maneuvers. The

*Avoidance Grid* is organized in *Layers* characteristic by the same distance from *Avoidance Grid Origin*.

Survey of motion planning algorithm was given in [4]. The ideal candidate for propagation algorithm is *Wave-front* algorithm propagating *Trajectory tree* through Layers. Due to the *Avoidance Grid* onion-like layers, there is a possibility to implement turn maneuver through layers iterative and effectively.

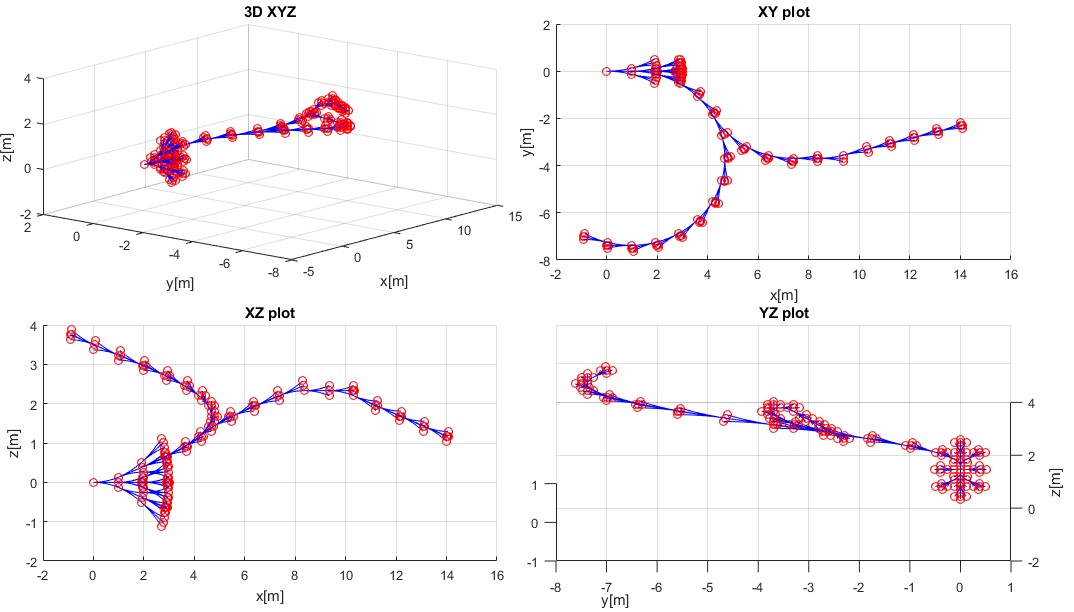


Figure 6.1: *Rapid Exploration tree as a result of* Constrained trajectory expansion.

**Rapid Exploration Tree** (fig. 6.1) was selected because it enables *Movement Automaton Utilization* and *Property Binding*. A similar approach was used for space exploration [5].

**The example** (fig. 6.1) shows a *Rapid Exploration Tree* in *Free Space* containing *Waypoint Navigation Path* and *Turn Away Path*. Both paths are starting in same *Root Node* (red circle) which was expanded with simple *Movement Automaton* (a bunch of nodes originating from one node is showing the way of expansion). The connection (blue line) between two nodes (red circles) represents *Trajectory portion* for *Executed Movement*.

**Rapid Exploration Tree Node** will contain the following information:

1. *Initial state* - root entry point, used in state evolution calculation.
2. *Trajectory (state evolution)* - trajectory passing through *state space* in the local coordinate frame of *Avoidance Grid*.
3. *Buffer* (applied movements) - ordered list of *executed movements* applied on the *initial state* to obtain *state evolution*.
4. *Cost* - calculated for *state evolution* based on a *predefined cost function*.
5. *Footprint* - ordered set of *passing cells* in *Avoidance Grid*.
6. *Parent Node Reference* - tree reference for the parent node, not in case of the *root node*.
7. *Other Bounded Properties* - value list of other properties, depending on *Expansion Constraints* and *Reachability* evaluation algorithm.

**Wave-front propagation of Rapid Exploration Tree** is given in (alg. 6.1).

The *Avoidance Grid* have UAS with *position* ∈ *Initial State* at the *origin*. The *Grid Layer* is a column ordered the set of cells with same *Mean distance* from the origin. *Grid Layers* are indexed from origin starting with 1; there is a maximum of *i* ≥ *1* layers.

**Step: Initialization** contains base structure preparation like follows:

1. *Avoidance Grid* - Space was containing *Reach set* (def. 2).
2. *Movement Automaton* - Used as *Predictor*, consuming *buffer* containing *Movements* to generate *Trajectory*(*initialState, buffer*).
3. *Reach Set* - tree consisting from *Wave-frontNodes* representing the endpoint of *Trajectory*(*initialState, buffer*) where each *Edge* represents *one Movement application*. The root is set as a node containing *Initial State*.

Function *initializeReachSet(root, stack, grid, automaton)* will take root and enforces *full wavefront propagation* to *First Layer*.

**Step: Wave-front Propagation** is forced propagation of trajectories from layer *i* to layer *i* + 1. The process goes as follows:

1. *Selection of Feasible candidates* - function *[candidates,leftovers] = ExpansionConstraints.select(stack)* for working layer, row and cell selects *feasible trajectory nodes* ordered by *Cost function*. The *Example of Cost Function* can be *Trajectory Smoothness* (def. 7).
2. *Expansion of Candidates* - for each *candidate* function *candidate.expandNode (automaton)* is invoked. This function will expand *Candidate Node structure* by appending *Full Trajectory Tree Evolution* until each *Leaf Trajectory* reaches *Next Layer*. Simply put *Par rent Node Node(initialState, buffer, cost, footprint )* buffer is appended by movements until the next layer is reached.
3. *Leftovers purge* - function *reachSet.purge(leftovers)* removes unexpanded *Nodes* leading to cell, effectively removing trajectories which do not lead to the *next layer*.
4. *Append Reach Set* - function *reachSet.append(leafs)* puts newly created *Nodes (Trees)* into *Reach Set* structure. The *Wave-front Propagation* for one cell is finished.

**Step: After Layer Propagation Purge** is covered by function *reachSet.purgeSameFootprint()* which takes trajectories with the same footprint and keeps some of them based on *Selection criteria*, more in (sec. 6.4.4, 6.4.5). *Pruning methods* over *Large Decision Trees* are *fast* and *viable* [6].

*Note. Reach Set* is usually computed *Prior the Flight* for *some Initial State* in *Local Coordinate Frame* in *right had coordinate frame* with *X*+ used as *main axis*.

**Algorithm 6.1:** *Wave-front propagation* of *Rapid Exploration Tree* to form

*Reach Set*.

**Input :** Node(initialState,buffer=∅,cost=0,footprint=∅), AvoidanceGrid,

ExpansionConstraints, MovementAutomaton(movementSet) **Output:** ReachSet(AvoidanceGrid)

# Initialization Sequence; grid=AvoidanceGrid, automaton=MovementAutomaton, root = Node; reachSet = initializeReachSet(root,stack,grid,automaton);

# Main Expansion through, layers (i), rows (j), cells(k);

**for**

*layer(*

1

*...i*

*)*

in

*grid*

**do**

**for**

*row(*

1

*...j*

in

*layer)*

**do**

**for**

*cell(*

1

*...k*

*)*

in

*row*

**do**

#applyselectioncriteria;

[

candidates,leftovers]=ExpansionConstraints.select(stack);

#collectexpansions;

leafs=[];

**for**

*candidate*

in

*Candidates*

**do**

leafs=[leafs,candidate.expandNode(automaton)];

**end**

reachSet.purge(leftovers);

reachSet.append(leafs);

**end**

**end**

reachSet.purgeSameFootprint();

**end**

# 6.4.4 Coverage-Maximizing Reach Set Aproximation

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

**Background:** There is *Coverage Ratio* property of *Reach Set* (def. 6). It has been shown that creating *Reach Set* via *greedy approach* is not feasible due the *Scaling Factor*. *Contracted Expansion* (sec. 6.4.3) is enabling to apply selection criteria while building *Reach Set* in given *Cell*.

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

**Expansion Constraint Function Implementation** (alg. 6.2) is based on the simple principle: *Select candidate Nodes which are closest to outer walls of Cell, with a unique footprint*.

**Tuning Parameters**: *Proximity to Cell outer wall* gives good chances to break into other rows or columns in the *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 nonredundant trajectories use ∞, it will consider full footprint.
2. *Spread Limit* - the 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 are 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 the *trajectory is constrained in the* cell*,j,k*.

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

1. *Best Performance Map* is created with a *footprint* as a key set element to ensure footprint uniqueness.
2. *Wall distance* for the *test node* is calculated as a closest trajectory portion distance to the *top, bottom, left, right* wall of cell *cell,j,k*
3. *The footprint* for the *test node* is created with the maximal length given by *Footprint Length* tuning parameter.
4. *Existence and Performance Test* is executed to ensure that the best performing node is selected. If there is no key entry in the *Best Performance Map*, then a new entry for *Test Node* is created. If there is a 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 6.2:** Expansion Constraint function for *Coverage-Maximizing 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**

**end**

bestPerformanceMap.setByKey(footprint,test);

**end**

# Select best performing nodes up to *spreadLimit* count; candidates = bestPerformanceMap.select(count =

spreadLimit).orderBy(’wallDistance’,’Ascending’);

leftovers = passing - candidates; **return** *[candidates,leftovers]*

**Example:** for *Avoidance Grid* with *Distance 10 m*, *Layer count 10*, *Horizontal range*

], *Horizontal Cell Count 7*, *Vertical range* ], and *Vertical Cell Count 5*. Is given in (fig. 6.2). The UAS is at *Back-side* of *Figure* (initial state is at all *Trajectory Origins*). The *black dashed line* marks *Avoidance Grid* space boundary. Each trajectory has its own color and ends at *Front-side* of *Avoidance Grid Boundary*.

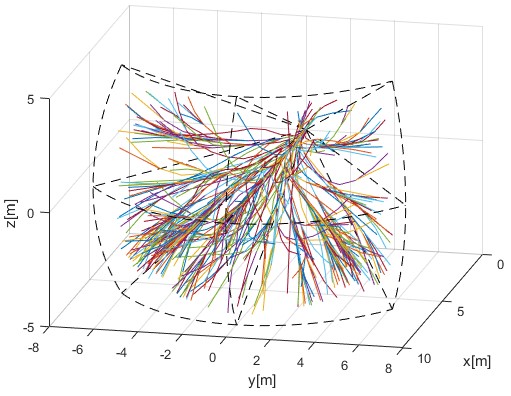


Figure 6.2: *Coverage-Maximizing* reach set *approximation*.

**Pros and Cons:** It can be seen from example (fig. 6.2) that *Coverage-Maximizing Reach Set Approximation Method* (alg. 6.2) generates much *turning* and *shaky trajectories*.

*High Coverage Ratio* (∼ 0*.*9) is provided while keeping *medium node count*. The calculation complexity scales linearly with grid size. The *upper limit of trajectories* is given as follow:

*countTrajectories*(*ReachSet*) ≤ *layerCellCount* × *spreadLimit*

× *size*(*Movements*) (6.15)

The *upper limit of nodes* is given as follow:

*countNodes*(*ReachSet*) ≤ *layerCount* × *layerCellCount*

× *size*(*Movements*) × *spread limit* (6.16)

*The absence* of *Smooth Trajectories* disqualifies *Coverage Maximizing -RSA* to be used for *Navigation*. This type of reach set is feasible for *Avoidance* because it contains a variety of maneuvers.

# 6.4.5 Turn-Minimizing Reach Set Approximation

**Motivation:** Imagine having an *Avoidance Grid* like (fig. **??**). There is a need of

*Reach Set Approximation* which will have *Smooth Trajectories* (def. 7) going nearby *cell* centers.

**Background:** The *Smoothness Rating for Trajectory* (def. 7) uses two distinct sets *Smooth Movements* and *Chaotic Movements* (eq. 6.12) which are defined for our *Movement Automaton* (sec. **??**) like following:

*SmoothMovements* = {*Straight*}

(6.17)

*ChaoticMovements* = *Movements* − *SmoothMovements*

*Smooth Movements* contains only *Straight* movement because others are considered as extreme turning movements. *Smooth Movements* should contain only direct flight movements or slight heading correction. *Chaotic Movements* set is a supplement of *Movement Automaton‘s Movement Set*.

The *Avoidance Grid* (fig. **??**) cell centers for fixed indexes *jfix*, *kfix* are linearly aligned with the *initial state*. That means that cell centers of cells *cell*1*,jfix,kfix,...,celli,jfix,kfix*, where *i* is a count of *layers* lie on one line. If the trajectory can achieve *cell center* on some *layer,* only minor trajectory corrections are required to stay on the given line. This type of trajectory gives us the following advantages:

1. *Minimal steering at the beginning* - the minimal steering is advantageous in *Controlled Airspace* because is diminishing the amount of communication to *UTM Service*.
2. *Additional safe space in the Linear segment* - once the *center of the cell* is reached, *Trajectory* sticks to the line between cell centers. Each point on this line has a *maximal distance* to outer walls of the cell. This gives us extra space given as minimum of distance between *UAS position* and *Outer cell walls*.

**Expansion Constraint Function Implementation** (alg. 6.3) is based on the simple principle: *Select candidate Nodes which are closest to Cell center, with a unique footprint*.

*Note. Cell center* can be closely reached by *smooth movement* from a previous cell or *chaotic movement* from a neighbouring cell from the current or previous layer. These trajectories are usually equivalent in *Smoothness*.

**Tuning Parameter:** *Proximity to Cell Center* gives a good chance to keep trajectory smooth or *smooth after one correction maneuver*. It has been mentioned that *Cell Center* can be reached by various trajectories. In this method full footprint length is always considered; therefore only one tuning parameter can be offered:

1. *Spread Limit* - the upper limit of candidates which are going to be selected for further expansion, minimal value 1, default value *Count of unique Moves in Movement set*, maximal value ∞. If the maximal value ∞ is selected, the algorithm will generate the skeleton of *Reach Set* with full Coverage and with the smoothest *Trajectories*.

**Step: Initialization** sets candidate *Nodes* as empty set, leftover *Nodes* as empty set. and selects all *Nodes* from *Stack* which represents *Finishing Trajectories* in working cell

*celli,j,k*.

**Algorithm 6.3:** Expansion Constraint function for *Turn-Minimizing Reach Set*

*Approximation*

**Input :** Node[] stack, Cell cell*i,j,k*

**Tuning Parameters:** int+ spreadLimit

**Output :** Node[] candidates, Node[] leftovers

# Initialize structures;

Node[] candidates = [], Node[] leftovers=[];

Node[] passing = cell*i,j,k*.getFinishingTrajectories(stack);

# Select unique smoothest trajectories; Map*<*Buffer,Node*>* bestPerformanceMap; **for** *Node test* ∈ *passing* **do**

centerDistance= test.getPerformance(cell*i,j,k*)]; footPrint = test.getFootprint(); **if** *bestPerformanceMap.contains(footPrint)* **then** old = bestPerformanceMap.getByKey(footprint); oldPerformance= old.getPerformance(cell*i,j,k*); **if** *oldPerformance > centerDistance* **then** bestPerformanceMap.setByKey(footprint,test); **end**

**else**

**end**

bestPerformanceMap.setByKey(footprint,test);

**end**

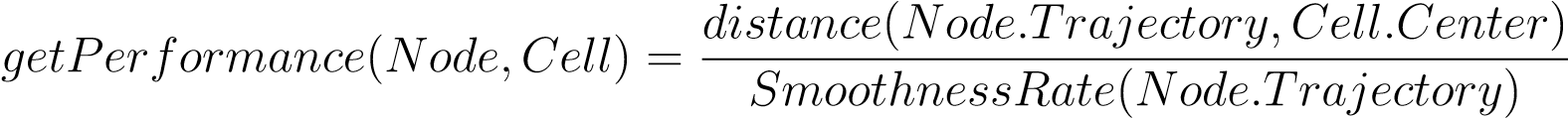
# Select best performing nodes up to *spreadLimit* count; candidates = bestPerformanceMap.select(count =

spreadLimit).orderBy(’cellCenterDistance’,’Ascending’);

leftovers = passing - candidates; **return** *[candidates,leftovers]*

**Step: Evaluate smoothest trajectories with unique Footprints** is implemented as *multi-criteria filtration*.

*The first criterion* is the *distance to Cell Center* which is penalized by trajectory *smoothness rate* implemented in method *Node.getPerformance(Cell celli,j,k)* defined as follow.

 (6.18) Distance of *Trajectory* is *enumerator* because its considered as the *base value* and is defined in the interval [0*,maximalWallDistance*]. The *Smoothness Rate* is in the denominator, because it is a penalization coefficient defined in the interval [0*,*1].

*The second criterion* is *trajectory uniqueness.* This is provided by *Best Performance Map*, where best performing *Node* belongs to one unique *trajectory footprint*. The implementation is identical to *coverage-maximizing set expansion* (alg. 6.2).

**Step: Select candidates** is executed on *Best Performance Map* records using *Penalized Cell Center Distance* as pivot parameter, ordered in ascending order and limited by *Spread Limit* tuning parameter. The *Leftovers* are difference set between *Passing Nodes* and *Candidate Nodes*.

**Example:** for *Avoidance Grid* with *Distance 10 m*, *Layer count 10*, *Horizontal range*

], *Horizontal Cell Count 7*, *Vertical range* ], and *Vertical Cell Count 5*. Is given in (fig. 6.3). The UAS is at *Back-side* of *Figure* (the initial state is at all *Trajectory Origins*). The *black dashed line* marks *Avoidance Grid* space boundary. Each trajectory has its color and ends at *Front-side* of *Avoidance Grid Boundary*. The *Spread Limit, in this case,* was set to 9 which is *Size of the Movement Set*.

*Note.* Please note *Trajectories* are organized in bundles going around *Cell Centers smoothly*. Most of the steering maneuvers are executed at the *beginning* of the *Avoidance Grid*.

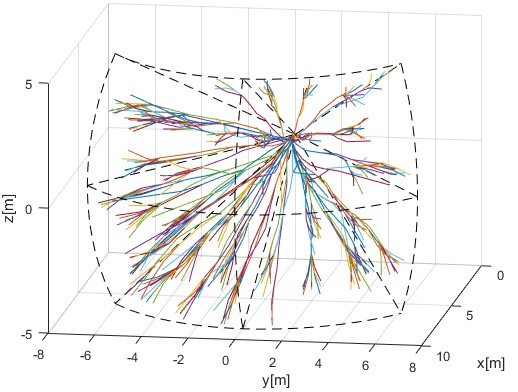


Figure 6.3: *Turn-minimizing* reach set *approximation*.

**Pros and Cons:** It can be seen from example (fig. 6.3) that *Turn-Minimizing Reach Set Approximation Method* (alg. 6.3) generates *smooth evenly spread trajectories*.

High smoothness ratio (≥ 0*.*9) is provided while keeping low node count for UAS systems. The calculation complexity scales linearly with grid size. The upper limit of trajectories is given as follow:

*countTrajectories*(*ReachSet*) ≤ *layerCellCount* × *spreadLimit*

× *size*(*Movements*) (6.19)

The *upper limit of nodes* is given as follow:

*countNodes*(*ReachSet*) ≤ *layerCount* × *layerCellCount* × *spread limit* (6.20)

The absence of *High Coverage Ratio* disqualifies *Turn-Minimizing Reach Set Approximation* to be used for *Emergency Avoidance*. This type of *Reach Set* is feasible for *Open Space Navigation* or *Controlled Airspace Navigation*. Its low turning rate in contained *Trajectories* are desired for such tasks.

# 6.4.6 ACAS-X like Reach Set Approximation

**Motivation:** The implementation of *ACAS-Xu* behavior in DAA system will be mandatory for *National Airspace System Integration* in United spaces [7].

Implementation of ACAS-Xu like behavior increase usability of approach, if it can be achieved without major concept changes.

**Background:** The *ACAS-Xu* system on the operational level has been described in [8]. The *Policy for Collision Avoidance* proposal has been given in [9].

Some behavioral patterns can be encoded into *Reach Set*. ACAS-Xu navigation part is basically *Look-up table of Maneuvers for Allowed Separations*.

The *Evasive Maneuver* selection process in ACAS-Xu is similar to our approach: *Select most energy efficient maneuver in compliance with space-time constraints*. ACASXu intruder model is similar to our *Body Volume Intersection Model* (app. **??**). The *ACAS-Xu* defines following base separations:

1. *Horizontal* - movements on a *Horizontal Plane* in *Global Coordinate System*.
2. *Vertical* - movements on *Vertical Plane* in *Global Coordinate System*.

There are allowed custom separations which can be used, for further experimentation:

1. *Slash* - movement on +45◦ *Tilted Plane to Horizontal Plane* in *Global Coordinate System*.
2. *Backslash* - movement on −45◦ *Tilted Plane to Horizontal Plane* in *Global Coordinate System*.

For given *Movement Automaton* implementation (sec. **??**) the separations are given as follow:

*Horizontal* = {*Straight,Left,Right*}

*V ertical* = {*Straight,Up,Down*}

(6.21) *Slash* = {*Straight,UpLeft,DownRight*}

*Backslash* = {*Straight,UpRight,DownLeft*}

For each *Node*(*...,buffer*) and each *separation* there is a evaluation function *isSeparation* which decides, if *Trajectory* defined by node buffer is made up only from *Separation* movements. The function *isSeparation*(*...*) is defined like:



∀*movement* ∈ *buffer,*

 : *true*

*isSeparation*(*buffer,separation*) = *movement* ∈ *separation* (6.22)

*otherwise* : *false*

Following *Separation Modes* can be defined with given *separations*:

1. *Horizontal* (ACAS-X defined mode) containing *horizontal* separation.
2. *Vertical* (ACAS-X defined mode) containing *vertical* separation.
3. *Horizontal-Vertical* (ACAS-X defined mode) containing *horizontal, vertical* separations.
4. *Full* (custom defined mode) containing all *Separation Modes*.

*Note.* Every separation modes generate 2D trajectories set on *Respective plane*. There is no need for *Tuning parameters* for further *Expansion Constraint*.

**Expansion Constraint Function Implementation** (alg. 6.4) is based on the simple principle: *Select only candidate Nodes which Trajectories have at least one desired Separation Mode*.

**Step: Initialization** sets candidate *Nodes* as the empty set, leftover *Nodes* as the empty set, and, select all nodes to form a *stack* which represents *Finishing Trajectories* in working *celli,j,k*,

**Step: Candidate Selection Process** is evaluated for each *test Node* from *passing Node Set*.

For each *applicable separation*, given as input parameter *separations*, The test function *isSeparation* (eq. 6.22) is applied:

1. If *test Node* trajectory belongs to at least one allowed separation it is added to candidates set.
2. Else is added to *Leftovers*.

*Note. Separation sets* (eq. 6.21) are not *exclusive sets* in *Movement Automaton* domain.

One *Trajectory* contained by Node can belong to multiple *Separations*.

**Algorithm 6.4:** Expansion Constraint function for *ACAS-like Reach Set Approximation*

**Input :** Node[] stack, Cell cell*i,j,k*, Separation[] separations

**Tuning Parameters:** *None* : ∅

**Output :** Node[] candidates, Node[] leftovers

# Initialize structures;

Node[] candidates = [], Node[] leftovers=[];

Node[] passing = cell*i,j,k*.getFinishingTrajectories(stack);

# Select nodes containing trajectories with usable separations; **for** *Node test* ∈ *passing* **do for** *separation* ∈ *separations* **do**

**end**

**end**

# Get separations for Node;

Separations[] nodeSeparations = test.getSeparations();

# If trajectory given by buffer is on Separation plane; **if** *isIn(isSeparation(test.buffer,separation)(6.22)* **then** candidates.append(test); **end**

# If there was no applicable separation, throw Node away; **if** *test* 6∈ *candidates* **then** leftovers.append(test);

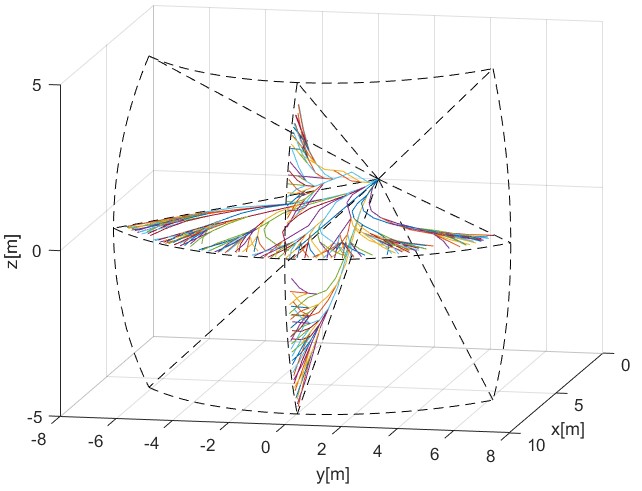
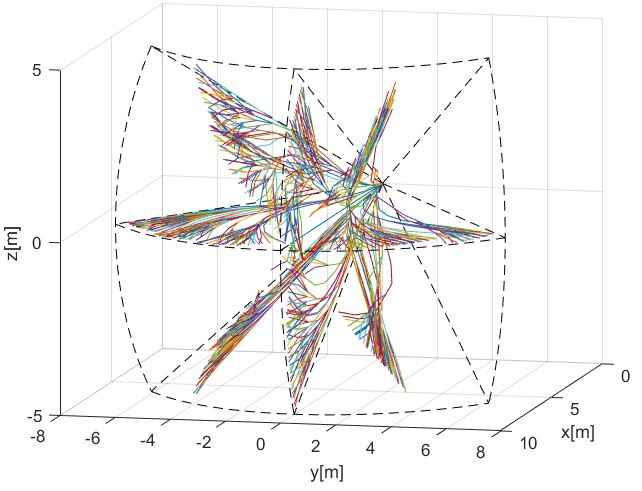
**end**

# Return results; **return** *[candidates,leftovers]*

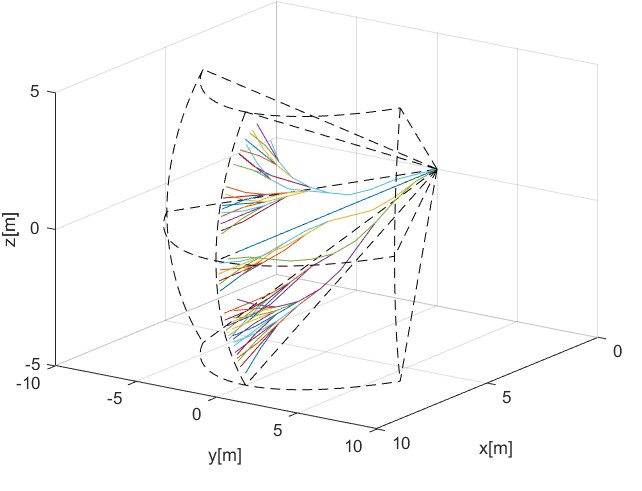
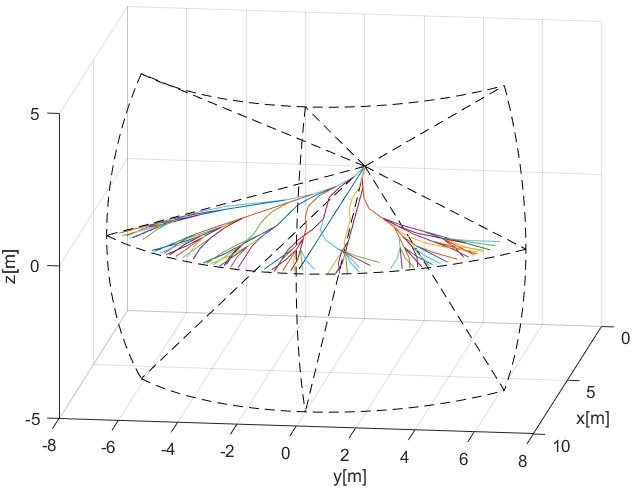
**Example:** for *Avoidance Grid* with *Distance 10 m*, *Layer count 10*, *Horizontal range*

], *Horizontal Cell Count 7*, *Vertical range* ], and *Vertical Cell Count 5*. Is given in (fig. 6.4). The UAS is at *Back-side* of *Figure* (initial state is at all *Trajectory Origins*). The *black dashed line* marks *Avoidance Grid* space boundary. Each trajectory has its own color and ends at *Front-side* of *Avoidance Grid Boundary*.

*Full* separation mode is given in (fig. 6.4a). *Horizontal-Vertical* separation mode, used in original *ACAS-Xu* testing [8], given in (fig. 6.4b). *Horizontal* separation mode given in (fig. 6.4c) is usually used by planes. *Vertical* separation mode given in (fig. 6.4d) is usually used by copters.



(a) Full. (b) Horizontal-Vertical.



(c) Horizontal. (d) Vertical.

Figure 6.4: ACAS-X imitation *reach set* approximation for various *separation modes*.

**Pros and Cons:** It can be seen from examples (fig. 6.4) that *ACAS-like Reach Set Approximation Method* (alg. 6.4) generates a full reach set for 2D plane located in 3D space.

The *Reach Set* contains trajectories with *high coverage ratio* and *high smoothness rating* for selected 2D separation plane. Overall performance compared to full 3D reach sets (sec. 6.4.4, 6.4.5 6.4.7) is poor.

The *node* and *trajectory* count boundary was not implemented. It is common knowledge that *2D* avoidance sets do not require scaling [8]. Otherwise, trajectory footprint mechanism like in *Turn-Minimizing Reach Set Approximation* (alg. 6.3) can be introduced.

This reach set implements *Planar-Separation* as a native feature, it can be used for both *navigation* and *avoidance* tasks in *Controlled Airspace*. For *Non-controlled Airspace,* there are far more superior *Combined Reach Set* (sec. 6.4.7).

# 6.4.7 Combined Reach Set Approximation - Tree Merge

**Motivation:** Turn-Minimizing Reach Set Approximation (sec. 6.4.5) is *efficient* for *Navigation in* Controlled Airspace. Coverage-Maximizing Reach Set Approximation (sec. 6.4.4) is good for *Emergency avoidance*. The need for the differentiation between *Navigation* and *Emergency Avoidance* mode is necessary for *Controlled Airspace*. But not in *Noncontrolled Airspace*. The combination of *Turning-Minimizing* and *Coverage-Maximizing* reach set approximations is an obvious solution.

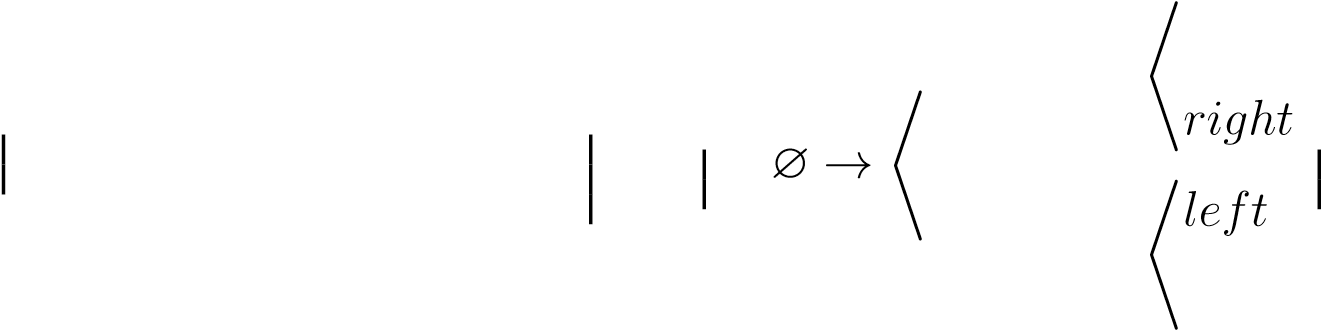
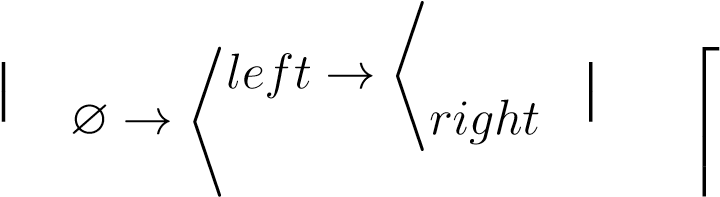
*Automatic mode switch* can be provided by a combination of *Navigation Reach Set* and *Avoidance Reach Set* with an elevated cost function. Overall having a method to merge multiple trees would be beneficial.

**Background:** If two *Reach Set Approximation* were calculated for the same *Avoidance Grid* and *Initial State*, using same *Movement Automaton* and *UAS model* are possible to merge.

The *Reach Set Approximation* is a *tree* with *Root Node* in *initial state* with movement buffer = ∅. The *movement buffer* in each node can be used as *route trace* during the merging procedure. The example two reach set merge can be given as follow, where only the *latest* applied movement is taken into account.

 

First Reach Set

  

 *left* 

Combined Reach Set 

  

  *left*    →  *left* →  (6.23)

 ∅



Second Reach Set

  

  → 

 \*∅   *right right*



 ∅ → \**left* 



 *right* → 

*right*

*First Reach Set* contains two trajectories given by buffers {*left,left*} and {*left,right*}.

*Second Reach Set* contains two trajectories given by buffers {*right,left*} and {*right,right*}. The *Combined Reach Set* contains all four trajectories.

*Note.* The combined tree [10] does not need to have a combined amount of original *Reach Sets* trajectories. There can be *Duplicity* which means that any bounded property like *Cost* must be *calculated* again.

**Combined Reach Set Calculation Function** (alg. 6.5) is implemented as function *NodecombinedReachSet*(*...*) which takes root Node with *initial State*, *Avoidance Grid* and respective parameters for each calculation method. *turn-minimizing spread* for *TurnMinimizing Reach set calculation* and *coverage spread*, *Footprint Length* for *CoverageMaximizing Reach Set Approximation*.

*Separate Reach Sets* are calculated using *Wave-front propagation* (alg. 6.1) using respective *Constrained Expansion* functions for *Turn-Minimizing* (alg. 6.3) and *CoverageMaximizing* (alg. 6.2) reach sets.

*Combined Reach Set* is created using *Node merge tree(...)* function because different cost function or *Bounded Parameters Calculation* may be applied on *Original Reach Sets*.

*Cost* for *each node* needs to be recalculated due to original reach sets disparity. Function *combined.applyCostFunction()* will recalculate the new cost for each node.

The Goal is to have a penalization for *Chaotic behavior*, implementation of *Automatic Mode Switch* can be done like follows:

1. *Calculate Normal Cost* for Node *Cost*(*Node*) for the associated trajectory:

*Cost*(*Node.Trajectory*).

1. *Calculate Penalization for* additional maneuvering, calculate *Smoothness Rating for Trajectory* (def. 7) in the interval [0*,*1], introduce penalization with base 100%.

The final *Cost*(*Node*) function is applied to each *Combined Reach Set Node* and look like follows:

*Cost*(*Node*) = *Cost*(*Node.Trajectory*) × *...*

··· × (1 + (1 − *SmoothnessRate*(*Node.Trajectory*))) (6.24)

**Tree Merge Function** *mergeTree*(*...*) implements *Outer Join* operation on two trees. Example was given in (eq. 6.23). Function is applied on *root Node* iterating over *Movements in Movement Set*, because *Movement is pivot*.

**Algorithm 6.5:** Reach Set Merge Function and Combined Reach Set calculation

# Tree merge function;

Node mergeTree(*Node firstNode, Node secondNode*)

# Try to copy reference node or return null;

Node referenceNode = (firstNode?:(secondNode?: return null)); Node merged = new Node(referenceNode); merged.leafs= [];

**end**

# Try to fetch movement nodes if exist in any sub tree; **for** *movement* ∈ *Movements* **do**

firstLeaf = firstNode.getLeafFor(movement); secondLeaf = secondNode.getLeafFor(movement); newLeaf = mergeTree(firstLeaf,secondLeaf); **if** *newLeaf* ∼= *null* **then**

merged.leafs.append(newLeaf); **end**

**return** *merged*

# Combined Reach Set calculation function;

Node combinedReachSet(*Node root, AvoidanceGrid grid,int*+ *coverageSpread, int*+ *turnSpread, int*+ *footprintLength*)

Node cmrsa = chaoticReachSet(root,grid, footprintLength,coverageSpread);

Node tmrsa = harmonicReachSet(root,grid, turnSpread); Node combined = mergeTree(cmrsa,tmrsa); combined.applyCostFunction(); **return** *combined*

**Example:** for *Avoidance Grid* with *Distance 10 m*, *Layer count 10*, *Horizontal range*

], *Horizontal Cell Count 7*, *Vertical range* ], and *Vertical Cell Count 5*. Is given in (fig. 6.5). The UAS is at *Back-side* of *Figure* (initial state is at all *Trajectory Origins*). The *black dashed line* marks *Avoidance Grid* space boundary. Each trajectory has its own color and ends at *Front-side* of *Avoidance Grid Boundary*. The *Coverage-Maximizing Spread* was set to 8, *Footprint Length* to 3 and *Turn-Minimizing Spread* to 1.

*Note.* Notice there are typical trajectories from both *Turn-Minimizing* (fig. 6.3) and *Coverage-Maximizing* (fig. 6.2) *Reach Set Approximations*.

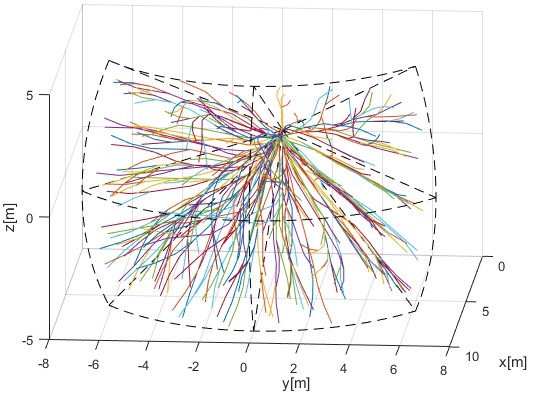


Figure 6.5: *Combined* reach set *approximation*.

**Pros and Cons:** It can bee is seen from example (fig. 6.5) that *Combined Reach Set Approximation* (alg. 6.5) contains both types of maneuvers. *Cheaper turn-minimizing* for navigation and *More Expensive Coverage-Maximizing* for *Emergency Avoidance*. The upper limit of trajectories is given as follow:

*countTrajectories*(*ReachSet*) ≤ *countTrajectories*(*CM* − *RSA*)

+ *countTrajectories*(*TM* − *RSA*) (6.25)

The *upper limit of nodes* is given as follow:

*countNodes*(*ReachSet*) ≤ *countNodes*(*CM* −*RSA*)+*countNodes*(*TM* −*RSA*) (6.26)

*Turn-Minimizing Reach Set* is ideal for *Non-controlled Airspace* missions, because it contains *Automatic Mode Switch* between *Navigation* and *Emergency Avoidance*.

24

# Bibliography

1. Franco Blanchini. Set invariance in control. *Automatica*, 35(11):1747–1767, 1999.
2. John Birmingham and Peter Kent. Tree-searching and tree-pruning techniques. In *Computer chess compendium*, pages 123–128. Springer, 1988.
3. Alojz Gomola, Joa˜o Borges de Sousa, Fernando Lobo Pereira, and Pavel Klang. Obstacle avoidance framework based on reach sets. In *Iberian Robotics conference*, pages 768–779. Springer, 2017.
4. Chad Goerzen, Zhaodan Kong, and Bernard Mettler. A survey of motion planning algorithms from the perspective of autonomous uav guidance. In *Selected papers from the 2nd International Symposium on UAVs, Reno, Nevada, USA June 8–10, 2009*, pages 65–100. Springer, 2009.
5. Catherine Plaisant, Jesse Grosjean, and Benjamin B Bederson. Spacetree: Supporting exploration in large node link tree, design evolution and empirical evaluation. In *Information Visualization, 2002. INFOVIS 2002. IEEE Symposium on*, pages 57–64. IEEE, 2002.
6. John Mingers. An empirical comparison of pruning methods for decision tree induction. *Machine learning*, 4(2):227–243, 1989.
7. Jay Shively. Uas integration in the nas: Detect and avoid. 2018.
8. Mike Marston and Gabe Baca. Acas-xu initial self-separation flight tests, 2015.
9. Kyle D Julian, Jessica Lopez, Jeffrey S Brush, Michael P Owen, and Mykel J Kochenderfer. Policy compression for aircraft collision avoidance systems. In *Digital*

*Avionics Systems Conference (DASC), 2016 IEEE/AIAA 35th*, pages 1–10. IEEE, 2016.

1. Patrick O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O’Neil. The logstructured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

25