Multi-Robot Hybrid Motion Planning Using Satisfiability Methods

Rui Pedro Caramujo de Sá
ruipedrodesa@tecnico.ulisboa.pt

Instituto Superior Técnico
October 2021

***Abstract*—Automation is an ever-growing need, especially when it comes to space exploration. In this thesis, the problem of planning for the stowage of cargo by a fleet of robots, in micro-gravity conditions, is tackled. This requires working with a hybrid and non-linear environment, since deliberation on both discrete and continuous-time characteristics of the system have to be made. Most techniques focus on the discretization of the continuous part, or the separation of both discrete and continuous and are not capable of handling non-linearity. The methods presented here consider all of them at the same time by using hybrid automata to describe the system and then, dReach ([1]), a tool which encodes the hybrid δ-reachability problem into a logical language, so that a Boolean Satisfiability (SAT), more specifically SAT Modulo Theory (SMT) solver, dReal ([2]), can find a model which validates it (a solution). dReal is the tool which enables taking into account the non-linearity of the system. This work, inspired by the advancements in SMT technology and tools which widen the range of problems they can handle, serves as a proof of concept, showing the potential for the use of the aforementioned techs and tools to solve these types of non-linear hybrid problems, which are historically very complex and hard to solve by conventional planning methods.**

1. Introduction

Space exploration has been a human interest since we first looked up. A method for motion planning for the transportation of cargo is presented, using hybrid planning in combination with satisfiability methods.

The scenario considered is a space station, where a fleet of robots is tasked with the unloading of supplies and other packages that arrive. The fleet should take the cargo from its initial position to a final one. The only available information is the map of the station, the previously mentioned positions and the maximum number of robots available.

Now, it's important to take into account that these operations will be subject to micro or even zero-gravity conditions. As such, part of the process of solving this problem was to decide how the robots would transport the cargo. Two options were considered:

- each robot attaches itself to the cargo and uses its propellers to guide the movement;
- each robot attaches itself to the cargo and an anchor point in the wall and pulls the cargo towards that anchor point.

Due to simplicity of operation and control, the second option was the one considered as means of transportation.

The problem is now defined as computing a motion plan for a given input with the following information:

- A map;
- Its connection points;
- An initial point;
- A final point;
- the pulling speed;
- the maximum number of robots aloud to be used at any given moment.

Thus, the robot should be able to plan which anchor points to connect to, in what order and how much time it should remain connected to that point.

Since finding a solution to this problem involves dealing with continuous change to the cargo's coordinates, as well as discrete change of the anchor points a robot is connected to, a hybrid model is needed, which considers both types of changes.

After creating a model, a satisfiability modulo theory solver is used to create a plan.

In Section 2, a theoretical background is given, in which every previous knowledge needed in order to understand the methods developed is given. Section 3 gives a brief overview of the current landscape for solving problems of the same type, while the approach developed in this work is explained in full detail in Section 4. Some results are analysed in Section 5. Finally, Section 6 provides a few concluding remarks as well as some suggested future ideas.

2. Background

This Section explains the planning and modelling tools used in solving the motion planning problem.

2.1. Hybrid Model

The context of this problem forces the use of a model which takes into account both discrete changes, when the robot changes the anchor point it's connected to, for instance, and continuous changes, like the evolving position of the cargo through the path from the initial to the final point. This type of model is called a ***Hybrid Model***.

This model integrates a discrete state space in which the continuous changes are represented for each state by continuous variables, and changes to the discrete state variables correspond to the discontinuous changes of the problem.

A hybrid model is formally described by the ***Hybrid Automaton***

Hybrid Automaton. According to [3], a **hybrid automaton** is a structure which contains

- $\mathbf{X} = \{x_1, x_2, ..., x_n\}$, a finite set of continuous variables;
- and Q, a finite directed graph, called ***Control Graph***, with each node being called a ***Control Mode*** and each edge a ***Control Switch***.

Each **control mode**, q, is composed of

- a set of invariant conditions, $\boldsymbol{inv}_q$, over the variables of $\mathbf{X}$, which must be true when in mode q;
- a set of flow conditions, $\boldsymbol{flow}_q$, in the form of differential equations over the variables of $\mathbf{X}$ and their first derivatives, $\dot{\mathbf{X}}$, which represent the dynamics of each variable in $\mathbf{X}$ when in mode q;
- and a set of initial values of $\mathbf{X}$ in q, $\boldsymbol{init}_q$.

Each **control switch**, $\text{jump}_{q \to q'}$, is associated to a pair of control modes, q and q', and a set of jump conditions over the variables of X, which, when true, allow switching from q to q'.

A hybrid automaton can be defined as a structure

$$H = \langle \mathbf{X}, Q, \text{init}, \text{inv}, \text{flow}, \text{jump} \rangle \tag{1}$$

where init, inv and flow contain init_q, inv_q and flow_q for each mode q and jump contains every $\text{jump}_{q \to q'}$ from any mode q to any other mode it is connected to, q'.

According to this definition, discrete change is modeled by the control switches of the automaton, that is, each control switch corresponds to a discrete change in the system, and continuous change is modeled by the flow conditions associated to each control mode.

With this tool, one can represent a problem as an automaton, where the control graph corresponds to the state-space, which represents every possible discrete scenario of said problem, with respect to the possible values and combinations between each discrete state variable, and within each control mode (state) the continuous variables evolve according to certain conditions.

2.2. $\mathcal{L}_{\mathcal{F}}$-Representation of Hybrid Automata

$\mathcal{L}_{\mathcal{F}}$ is the first-order signature of the set $\mathcal{F}$, of Type 2 computable functions, over $\mathbb{R}$. In [4] a representation of hybrid automata in $\mathcal{L}_{\mathcal{F}}$ is defined.

The work in [4] also gives the definition of bounded $\mathcal{L}_{\mathcal{F}}$-formulae ($\Sigma_1$-sentences in $\mathcal{L}_{\mathcal{F}}$). It is defined as

$$\varphi : Q^{I_1} x_1 \dots Q^{I_n} x_n . \psi(x_1, \dots, x_n) \tag{2}$$

where Q can be either existential or universal quantifiers ($\exists$ and $\forall$). This definition means that ψ is true for at least one or all (depending on the used quantifier) x_i, $1 \leq i \leq n$. For instance, the sentence $\forall^{I_i} x_i . \phi$, in which ϕ is any applicable logic formula, corresponds to $\forall x_i . (x_i \in I_i \implies \phi)$.

A hybrid automaton, such as the one presented before, in $\mathcal{L}_{\mathcal{F}}$-representation is

$$\begin{aligned} H = \langle &\mathbf{X}, Q, \{flow_q(\mathbf{x}, \mathbf{y}, t) : q \in Q\}, \\ &\{inv_q(\mathbf{x}) : q \in Q\}, \{jump_{q \to q'}(\mathbf{x}, \mathbf{y}) : q, q' \in Q\}, \\ &\{init_q(\mathbf{x}, \mathbf{y}, t) : q \in Q\} \rangle \end{aligned} \tag{3}$$

in which $\mathbf{X}$ and Q are as before and the remaining elements are finite sets of quantifier-free $\mathcal{L}_{\mathcal{F}}$-formulas. For any hybrid automaton, H, $X(H)$, $Q(H)$, $\text{flow}(H)$, $\text{inv}(H)$, $\text{jump}(H)$ and $\text{init}(H)$ denote its components.

2.3. Reachability in Hybrid systems

Consider an n-dimensional hybrid automaton H and a subset of its state space $U \subseteq Q(H) \times \mathbf{X}(H)$. If there is a trajectory $\xi \in [\![H]\!]$ that is able to be mapped into a sequence of modes of H along with a time duration for each, then it is said that U is reachable by H.

When restricting the continuous time duration to a bounded interval and the number of discrete switches to a finite number, one gets a ***bounded reachability problem for hybrid systems***. Considering, now, that $X(H)$ is a bounded subset of $\mathbb{R}^n$ and there is some $k \in \mathbb{N}$ and $M \in \mathbb{R}^{\geq 0}$, the (k, M)-bounded reachability problem asks for the same requirements as the unbounded version, with the addition of the conditions that $i \leq k$ and $t = \sum_{i=0}^{k} t_i$ where $t_i \leq M$.

For simplification, a discrete jump in a trajectory is referred to as a step.

2.4. dReach

dReach solves bounded reachability problems of hybrid systems by encoding them as first-order logic formulas over the real numbers. It, then, uses an SMT solver, dReal ([2]), to find a solution to the resulting formula.

Specifically, it provides a δ-complete reachability analysis, which reduces model checking problems to δ-decision problems of formulas over the reals.

Standard bounded reachability problems for simple hybrid systems are highly undecidable [5], thus, dReach decides the δ-reachability of hybrid systems.

The analysis of the hybrid system tolerates numerical errors within the bounds specified by the user via an arbitrary positive rational number, δ. For this, it considers the ***δ-bounded overapproximations*** of both the hybrid system H and U.

Bounded δ-reachability. Consider a hybrid system defined as in Section 2.1, $H = \langle X, Q, \text{flow}, \text{jump}, \text{inv}, \text{init}\rangle$, with Q being its control graph flow, jump, inv, init its flow, jump, invariant and initial conditions, respectively. Given a chosen non-negative rational error bound, δ, the δ-perturbation of H is defined as

$$H^\delta = \langle X, Q, \varphi^\delta_{flow}, \varphi^\delta_{jump}, \varphi^\delta_{inv}, \varphi^\delta_{init}\rangle \tag{4}$$

where φ_{flow}, φ_{jump}, φ_{inv} and φ_{init} are the Σ_1-sentences that encode the conditions in flow, jump, inv and init, and $\varphi^\delta_{...}$ their corresponding δ-weakened formulae.

Now, let there be a bound in the number of steps of a plan, $n \in \mathbb{N}$, and an upper bound on time duration, $T \in \mathbb{R}^+$. `unsafe` is defined by a first-order formula and denotes a subset of the state space of H. A bounded δ-reachability problem reaches one the two conclusions:

- `safe` - H cannot reach `unsafe` in n steps within time T;
- δ-`unsafe` - H^δ can reach `unsafe`$^\delta$ in n steps and within time T, here `unsafe`$^\delta$ is the δ-weakening of `unsafe`.

Note that, when the tool reaches the first option, it's certain that H does not reach the unsafe region, in case of δ-`unsafe` as an answer, there exists a δ-bounded perturbation of the system which can render it unsafe. This last answer discovers robustness problems in the system, which should be regarded as unsafe.

If the `unsafe` region is made to be the goal region of the cargo, then, dReach is able to decide if that region (more specifically, its δ-weakening) is reachable by the hybrid automaton H when starting at a specific initial point, if so, the returned "witness" can be used to generate a motion plan from that initial point to the final region.

The framework defined in [4], where the decidability of bounded δ-reachability is proven to be possible for a wide range of nonlinear hybrid systems, provides the formal correctness guaranties of dReach.

3. Related Work

In this section a few writings are introduced which explore similar problems or apply similar techniques.

3.1. Task and Motion Planning

Task and Motion Planning (TMP) problems are ones in which, for a robot to generate a plan, it must be able to decide on tasks (discrete decisions) and motion (continuous decisions). They are, by definition, hybrid planning problems.

In [6] the authors solve a planning problem for a hybrid system. They consider a problem in which a mobile robot with arms has to pick up bottles from various positions and move them to their goal regions, a pick-and-place domain. The robot can move its base, torso, arm of gripper to a target joint configuration as its primitive actions.

Since the robot should be able to reason about discrete actions as well as continuous configurations, this is a hybrid problem. In order to handle this, the authors use sampling to discretize the configuration space and then use a Hierarchical Task Network (HTN) to find a solution. For this they present an algorithm, the State-Abstracted HTN algorithm, which, given a description of the domain and a function which specifies the relevant state-variables for doing an action from some state, outputs a hierarchical optimal solution.

The work in [7] also tackles planning problems which involve both discrete and continuous decisions. An algorithm is presented which uses incremental SMT solvers (SMT solver which can perform repeated satisfiability checks while constraints are pushed and popped from a stack maintained by the solver) to generate task plans and then a sampling-based motion planner, RRT-Connect, to generate a motion plan according to the previously generated task plan. If the motion planner fails to find a solution for a given task plan, then new task constraints are added to the task planner in order to produce a new task plan, if, however, the task planner fails to produce a new task plan, the main algorithm increments the task planning step horizon (maximum number of steps in the task plan) and the motion planning sampling horizon (timeout of the motion plan) and starts over without the added constraints to the task planner.

As a final example, another approach for solving TMP problems is presented in [8], named Robosynth. This approach receives as input a scene description, plan outline and a set of requirements.

A scene description is composed of a domain, where the user describes the information which remains unchanged throughout every instance of the problem, and a scene, the opposite, information which may change between instances. A plan outline consists of a program in which the user gives a high-level description of what a successful plan should look like. Lastly, the user should provide Robosynth with a goal and a set of invariants which must hold for the entirety of the planned paths.

Robosynth uses a variation of a manipulation graph, called a placement graph, as a sort of database of possible, feasible paths, then it uses an SMT solver to find a solution to the logical representation of the problem, querying the placement graph whenever it decides on a path as to verify it, thus solving both discrete and continuous parts of the problem.

3.2. Planning with Hybrid Automata

The work in [9] introduces a way of representing a hybrid planning problem using hybrid automata as well as its subsequent logical encoding as to allow an **SMT!** (**SMT!**)

solver to find a solution. This is the work which is closest to this thesis, and it is also the first of the presented works which takes into account the possibility of the presence of non-linearity in the system.

4. Approach

In this section an overview of the steps necessary to retrieve a motion plan given the input data described in Section 1 are explained.

4.1. Convex Decomposition

The first thing that is done with the input data in a convex decomposition of the map. The method used here is the one briefly described in [10].

The map's convex decomposition is done by recurrently calling an algorithm which receives a simple polygon with no holes, P, and tries to find a convex polygon within P, represented by a list of vertex $v_1, \dots, v_n$ in clockwise order.

Note that for a polygon of size n, when referring to its possible indexes, the reader should consider the remainder of the division by n, for instance, index i should be read as $i\%n$, $\%$ representing the remainder. This allows looping through the list of vertices which represents a polygon even when the starting point doesn't coincide with the one of said list.

Algorithm 1 has the corresponding pseudocode, with Algorithm 2 being an auxiliary function used in the process.

Algorithm 1: MP3

```
Data: P - simple polygon with no holes
s - starting index
Result: L - extracted convex polygon
P\L - P without L
the index (in P\L) of the last element in L
if P is convex then
    return L = [], P\L = P and 0;
end
Add P(s) and P(s+1) to L;
Initialise i at s+2;
repeat
    Add P(i) to L;
    increment i;
until P(i) creates a notch in L;
Remove lastly added P(i) from L
repeat
    Check_L(L);
until L returns unchanged;
Initialise i at s-1;
repeat
    Add P(i) to L;
    decrease i by 1;
until P(i) creates a notch in L;
Remove lastly added P(i) from L;
repeat
    Check_L(L);
until L returns unchanged;
if L is of length 2 or any vertex of L is a notch then

else
    return L = [], P\L = P and 0;
end
return L, P\L, and i;
```

Starting at a given index, s, of the polygon, given as an input along with $P = \{v_1, \dots, v_n\}$, the algorithm appends vertices v_s and v_{s+1} into a list L, initialized as empty, which holds the vertices of the convex polygon. After L consists of two vertices, the algorithm only adds a vertex v_i after checking three angles. These are $\text{ang}(v_i, l_1, l_2)$, $\text{ang}(l_k, v_i, l_1)$ and $\text{ang}(l_{k-1}, l_k, v_i)$, where l_i is the vertex at position i of L, and $\text{ang}(a, b, c)$ is the angle from vector $\vec{ab}$ to $\vec{bc}$ in the counter-clockwise direction. Since the vertex are in clockwise order in L (because they are so in P), the three angles checked give the internal angle in v_i and each of its two adjacent neighbors, as such, if any of these are greater than 180º, the addition of v_i fails and the algorithm goes to the next phase. If this is not the case, v_i is successfully added to L and removed from $P \setminus L$ (which was initialized as a copy of P), the remaining polygon when L is subtracted from P.

Afterwards, in case L has more than two vertices, the algorithm checks its validity. First it checks if there is any notch, v, of $P \setminus L$ in R, the smallest rectangle containing L with sides parallel to the x and y axis. A vertex of a polygon is a notch if its internal angle is greater than 180º. In case this is the situation, it then checks if that notch is inside L itself. This being the case, L needs to be changed, otherwise L is valid and the algorithm can move on to the next phase.

Correcting L is a simple case of removing its last vertex, l_k, as well as every vertex contained in the half-plane defined by l_1 and l_k containing the notch that triggered the correction, v.

This verification and correction step is done until either L converges to a valid convex polygon (i.e., L remains unchanged after this step) or L becomes composed by less than three vertices (stops being a polygon).

The algorithm now starts adding vertices to $L = \{l_1, \dots, l_k\}$ once again, except this time it starts adding in counter-clockwise order. Let v_s be the equivalent in $P \setminus L$ of l_1, $\{v_{s-1}, vs - 2, \dots\}$ is the sequence of vertices whose addition will be considered. When adding a new vertex, the internal angles it creates in itself, l_k and the lastly added vertex (l_1 if no new vertex has been added in this stage yet) are tested as in the first vertex addition stage. Note that if a, b, c are in clockwise order, using $\text{ang}(a, b, c)$ always gives the internal angle at vertex b.

Another verification is done in the same conditions as previously mentioned. Finally, if L is composed of more than two vertices and if either its first or last vertex is a notch of P, then L is accepted as a convex component of P.

When iterated, this algorithm is capable of decomposing

```
Algorithm 2: Check_L
  Data: L - extracted convex polygon
  P\L - P without L
  Result: changed - boolean variable True if L
          changed during this algorithm and False
          otherwise
  if any notch of P\L in R (the smallest possible
    rectangle which contains L) then
      if any notch of P\L in L then
          remove last vertex from L;
          remove vertices from L that are in the
            half-plane created by the first and last
            vertices of L which also contains the notch
            which triggered the condition;
          return changed = True
      else
          return changed = False
      end
  else
      return changed = False
  end
```

a simple polygon P with no holes into a set D of convex polygons. It starts by receiving P and the index of the first element in P and outputting a convex polygon L, which is promptly added to D, $P \setminus L$ and the index, in $P \setminus L$, of the last element of L, f. At each subsequent iteration, it takes $P \setminus L$ as a polygon to divide and f as the starting index, repeating this until L is returned empty, meaning that $P \setminus L$ was already convex at the end of the previous iteration and it is then added to D, completing the decomposition.

As a result of this decomposition, the original map is transformed into a set of non-overlapping convex polygons, which are connected through shared borders.

4.2. Polygon Extension

After the convex decomposition of the map, the resulting non-overlapping polygons are extended.

Consider the map in Figure 1 and its given convex decomposition in Figure 2. The method described henceforth is going to expand polygon A into polygon B and vice-versa.

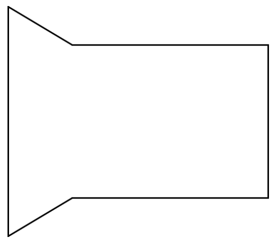

Figure 1: Example of map for Polygon Extension algorithm.

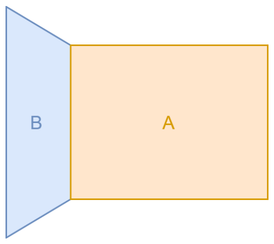

Figure 2: Decomposition of map if Figure 1.

In order to do this, the algorithm considers the two vertex points which define the border line between these polygons. When computing the expansion of polygon A into polygon B, pictured in Figure 3, the border lines neighbouring the one being expanded (the one between the polygons A and B) are extended into in the direction of polygon B. In Figure 3 these would be the lines $\overline{{}^AP_n{}^AP_1}$ and $\overline{{}^AP_3{}^AP_2}$. Then, there are three possible cases:

- The intersection point of $\overline{{}^AP_n{}^AP_1}$ with $\overline{{}^AP_3{}^AP_2}$, ${}^AP_i$ exists and is **inside** polygon B;
- ${}^AP_i$ exists and is **outside** polygon B;
- ${}^AP_i$ does not exist.

In the first case, ${}^AP_i$, is added to polygon A in between ${}^AP_1$ and ${}^AP_2$, in the correct, clockwise order.

In both of the other two cases, the intersection points of $\overline{{}^AP_n{}^AP_1}$ and $\overline{{}^AP_3{}^AP_2}$ with polygon B's border are computed, ${}^AP_i, 1$ and ${}^AP_i, 2$ respectively. Then, they are added to a copy of polygon B, in the correct, clockwise order. Finally, starting from ${}^AP_i, 1$, each point, in clockwise order, which belongs to polygon B, is added to polygon A, between ${}^AP_1$ and ${}^AP_2$, in the correct, clockwise order.

In Figure 3, one can see that it is the last case which happens when extending polygon A into B, while in the inverse extension, polygon B into A, it is the first.

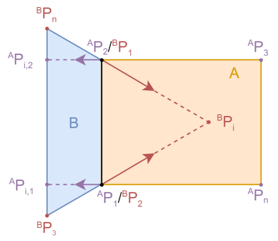

Figure 3: Example of Polygon Extension algorithm.

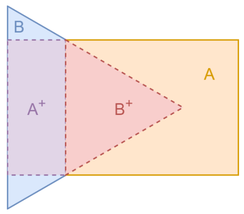

Figure 4: Overlapping zones resulting from the algorithm in Figure 3.

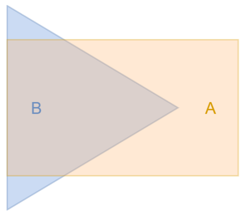

Figure 5: Final convex decomposition for the map in Figure 1.

Figure 4 shows the computed overlap zones to be added to each polygon and Figure 5 shows the final convex decomposition for the map in Figure 1, after expanding the polygons in the initial decomposition.

Consider the slightly altered map, represented in Figure 6 already decomposed, and the application of the extension algorithm to its composing polygons, A and B, also presented in Figure 6. When expanding polygon B into A, an example of the second case can be witnessed, in which the intersection point between lines $\overline{{}^BP_n{}^BP_1}$ and $\overline{{}^BP_3{}^BP_2}$, ${}^BP_i$, is outside of polygon A.

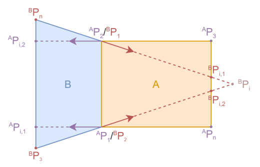

Figure 6: Example of Polygon Extension algorithm.

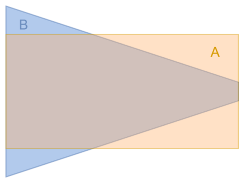

Figure 7: Final convex decomposition resulting from the algorithm application in6

This is the last step in the transformation of the original map into a set of convex, overlapping polygons. This transformation is very important to the model, for it enables the division of the problem into smaller, simpler sub-problems, a crucial step in its resolution, further explored in subsection 4.4.

4.3. Creating the Hybrid Automaton

The first step in creating a hybrid automaton is to define the variables. The only continuous variables considered are the coordinates of the cargo's position (x_c, y_c). Defining the discrete variables implies defining the different possible modes of the system. Since these modes serve to describe the dynamics of the continuous variables, defining them is simply a case of finding the different ways these vary. Given that the method of transportation considered is through the attachment of each active robot to an anchor point followed by the pulling of the cargo to each of those anchor points and since the map is divided into overlapping convex regions, there should be a mode for each possible combination of anchor points inside of each region.

Flow Conditions. The method of transportation chosen for consideration can be represented by a vector, v_i, that "pulls" from the cargo to the corresponding anchor point at a certain speed w (Figure 8 illustrates this with vectors for the case where two robots are "pulling" the cargo). The flow equations for a mode where r robots are each connected to a distinct anchor point a_i, $1 \leq i \leq r$, can thus be written as follows.

$$\begin{cases} \dot{x} = w \sum_{i=1}^{r} \frac{x_{v_{a_i}}}{||v_{a_i}||} = w \sum_{i=1}^{r} \frac{x_{a_i} - x}{\sqrt{(x_{a_i}-x)^2 + (y_{a_i}-y)^2}} \\ \dot{y} = w \sum_{i=1}^{r} \frac{y_{v_{a_i}}}{||v_{a_i}||} = w \sum_{i=1}^{r} \frac{y_{a_i} - y}{\sqrt{(x_{a_i}-x)^2 + (y_{a_i}-y)^2}} \end{cases} \quad (5)$$

That is, the sum of every vector generated by each acting robot's pull. With $v_{a_i} = (x_{v_{a_i}}, y_{v_{a_i}}) = (x_{a_i} - x, y_{a_i} - y)$ being the vector generated by the pulling of the cargo in position (x, y) towards the anchor point $a_i = (x_{a_i}, y_{a_i})$.

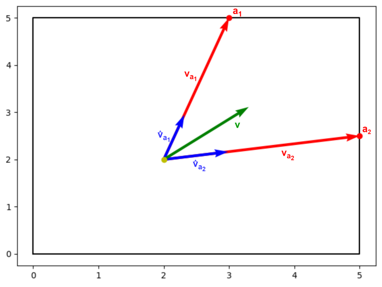

Figure 8: Example of set of vectors which represent the force generated by the pulling motion of the two active anchor points (red dots) and cargo position represented by the yellow dot. The red vectors represent the ones that "pull" the cargo, in blue their unit vectors, in green is the vector resulting from the addition of the two (blue) unit vectors.

A visual representation is given in Figure 8, where red vectors represent the pull of each robot, v_{a_i} attached to each anchor point a_i (red dots), blue vectors the corresponding unit vector $\hat{v}_i$, and in green the resulting motion vector v.

Invariant Conditions. The only thing that should always be true inside each mode is that the cargo (x, y) should remain inside the region associated to the activated mode at each time unit. This means that the invariant conditions of each mode can be derived from the border lines' equations. As an example, consider the region in Figure 9. Any mode associated to it would have as invariant conditions the following set.

$$\{(x > 0); (y > 0); (x < 5); (y < 15)\} \quad (6)$$

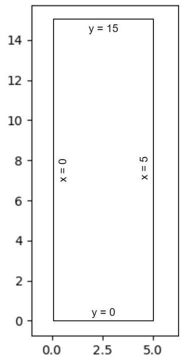

Figure 9: Example of a region and border lines' equations.

Jump Conditions. Since this decomposition of the map consists of overlapping regions, modes q_i from a region c_i can only switch to modes q_j from region c_j, $i \neq j$, when the cargo (x, y) is in both c_i and c_j.

Jumping from one mode to another can only be done if the preconditions are all true and there may be effects of such transition on the continuous variables of the problem. As mentioned, the preconditions to any jumps which require moving regions is for the cargo to be in the overlap shared by those regions. However, in case the modes are of the same region and only the connection point changes there are no preconditions, because the region is convex, it is always possible for this lastly mentioned type of transitions to happen. The effects on the continuous variables, cargo coordinates (x, y), are non-existent at switching time (beside the change in flow equations which naturally occurs when switching modes), the initial values of the next mode for these variables are passed in the effects of a jump to that mode.

Consider the regions in Figure 10. Going from the blue region to the orange one is defined by the jump whose preconditions and effects are in Equation 7, with the preconditions being derived from the overlap area's border lines' equations. Modes q_1 and q_2 are the ones associated to the blue and orange regions, respectively.

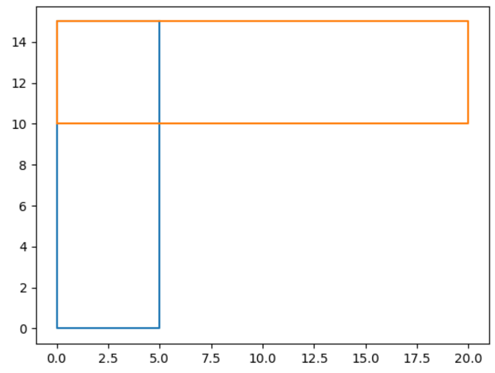

Figure 10: Example of two regions with overlap.

$$\begin{aligned} \text{pre}_{q_1 \to q_2} &: \begin{cases} x > 0.0 \\ y < 15.0 \\ x < 5.0 \\ y > 10.0 \end{cases} \\ \text{eff}_{q_1 \to q_2} &: \begin{cases} x_2^0 = x_1^t \\ y_2^0 = y_1^t \end{cases} \end{aligned} \tag{7}$$

Initial and Final conditions. The `.drh` file, which encodes the problem to the dReach input language, requires the initial and final modes to be explicitly indicated. Due to this, two modes were created to act only as initial and final modes. Otherwise, there would have to be an additional step in this whole process in which the modes with associated regions which contained the initial point would be considered as initial modes.

These modes have invariants which coincide with the problem's initial and final conditions and null flow conditions (the cargo should be stopped in these modes). The initial mode contains jumps to every mode associated with the region the initial point is in, the final mode contains no jumps since it will always be the last step in the plan.

With this the hybrid Automaton is fully described, although when solving a problem by considering the full automaton, dReach takes greater than desirable amounts of time (13 minutes for a simple problem and not even solving a problem a bit more complex even when left overnight).

4.4. Divide and Conquer

The method used to tackle the solving time issue mentioned above is to split the full problem into smaller, simpler sub-problems.

In order to do this, a graph is created, using the set of overlapping regions which result from the convex decomposition of the map followed by the expansion of the derived polygons, where each node is a region and each edge connects two nodes if it is possible to travel from one region to the other. Then, a path finding algorithm is used to find a sequence of nodes from the initial region, corresponding to the one in which the initial point is located, to the final, corresponding to the region in which the final point is located. This sequence is used to divide the problem into the smaller problems, which are then solved sequentially and solutions concatenated to get the motion plan of the full problem.

To create a sub-problem, each region is considered individually, along with the anchor points in it located, and the same methods as before are used to generate the modes of the hybrid automaton. A `.drh` file is created and passed to dReach, in the very same way as previously mentioned, the solution is then used to define the next sub-problem's initial point. If no solution is found, then the path (from the initial to the final region) being currently evaluated is considered unfeasible and the next one is evaluated. If no other path exists, then the algorithm informs the user it wasn't able to find a solution to the problem.

The only mode which is defined differently from what has already been presented is the final mode, which, for the first $n-1$ sub-problems in a path with n steps, instead of being defined by a box centred on the final point, is defined by the overlap zone between the regions it is traversing, the transitions zone (the n-th sub-problem's final mode is the whole problem's final mode, thus, defined as before).

Consider a region i, with $0 < i < n$, a sub-problem ${}^{s}p_i$'s final mode is defined by the overlap zone between region i and region $i+1$, i.e., a solution to ${}^{s}p_i$ is one which takes (x, y) from the initial point (given by the solution of the previous sub-problem ${}^{s}p_{i-1}$) to this zone, the actual point in the zone where the cargo ends up is given in the model computed by dReach, in case there is a solution. This final point becomes the initial point of the next sub-problem (${}^{s}p_{i+1}$).

After having done all these steps, a `.drh` file is generated according to the specifications in [1] and passed to dReach in order for it to try and find a solution.

5. Results and Analysis

The final methods were tested with two files, `simple.prb` and `complex.prb`, the former describes the problem in Figure 11 and the latter the one in Figure 12, in both Figures red dots are the available anchor points and yellow and green dots the initial and final points, respectively.

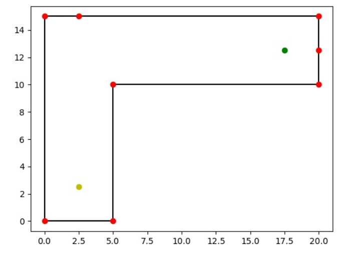

Figure 11: Figure defining simple case (`simple.drh`).

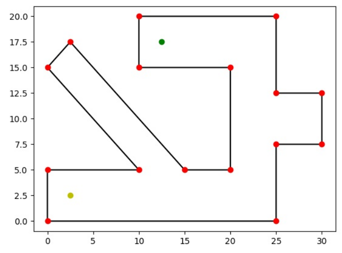

Figure 12: Figure defining complex case (`complex.drh`).

Both these problems were correctly solved by these methods, their solutions can be seen in Figures 13 and 14 and corresponding tables with information on the time spent in each step, as well as what anchor points are being used can be verified in Tables 1 and 2.

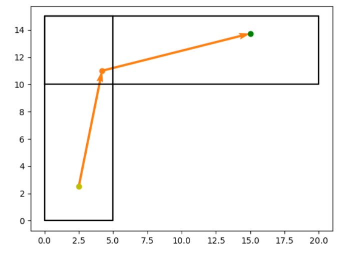

Figure 13: Solution to simple problem.

q_i	t_i	$Anchors$
q_1	1s	$(5, 15)$
q_2	2s	$(20, 15)$

Table 1: Table associated with the solution of `simple.prb` depicted in 13

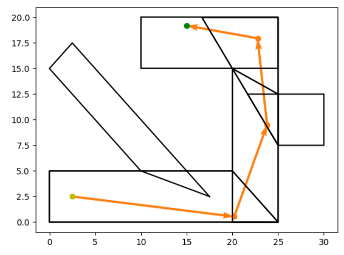

Figure 14: Solution to complex problem.

q_i	t_i	$Anchors$
q_1	17s	$(25, 0)$
q_2	9s	$(25, 12.5)$
q_3	8s	$(22.5, 20)$
q_4	7s	$(10, 20)$

Table 2: Table associated with the solution of `complex.prb` depicted in 14

The amount of time taken to solve the problems was 5 seconds for the simple case and 7 for the complex. This is a great improvement when compared to the 13 minutes it took to solve the simple problem before the divide and conquer method (the complex was not even solved when left running overnight).

6. Conclusion

This work tries to start the development of methods for enabling a fleet of robots to successfully plan the transportation of cargo, from its drop off point to its designated place. It hypothesises that this can be achieved using a hybrid representation and a δ-reachability problem solver, dReach, both of which are explained in Section 2. As such, the main problem can be seen as a motion planning problem where the object whose motion is being planned is the cargo, the robots are the means through which the cargo moves, thus, deciding how the cargo moves implies deciding what each robot should do (to which anchor points each should be connected) in order to force that to happen. This problem formulation is thoroughly described in Section 4.

6.1. Contribution

The problem of motion planning for micro-gravity conditions is, at the time of writing this thesis, relatively unexplored. As such, the work here presented is meant to be seen as closer to a proof of concept than to an actual solution for how to handle these types of problems.

The results presented and analysed in Section 5 show the current tools, when allied with the right models, may be ready for these types of challenges. Particularly, the combination of the ”divide and conquer” method (see Section 4) in combination with convex decomposition of the map with overlapping of polygons shows much promise to anyone who cares to follow this work, having been able to reduce the solving time very significantly already, in some cases this being critical to the decidability of the problem.

However, the reader should take note of the simplicity of the model, it does not consider such things as inertia, nor obstacle handling, both very important in this field of research. This is a start, but there's still a long way to go.

Further work is needed, both in the model and solver sides, before this type of planners can be used. As such continuing to try and use methods like the ones presented here and refine them to become more robust and consider a wider range of scenarios would be effort well spent in this steep uphill hike for knowledge.

6.2. Future Work

There are a lot of other spins to solving this problem which can be derived from this work and can potentially refine it.

Firstly, the decomposition of the map in regions could be made in a different way, in which the visibility polygons for each anchor point are computed and the map is divided into zones according to which anchor points are visible where, or even according to combinations of anchor points, given the maximum number of robots. For instance, a map of a problem in which a maximum of 2 robots may be used, can be divided into each visibility polygon, for modes with 1 robot, and every intersection between every pair of polygons, for modes in which two anchor points are used. The divisions will be overlapping and this may cause an explosion in the state space of the hybrid automaton but the methods in section 4 should be able to counter the effects of this explosion.

The solver could be improved in order to better handle these kinds of flow equations, reducing the amount of time it take to solve a full problem without requiring the divide and conquer method.

There is also a heuristics which could be used to filter through the possible modes, enabled by using the "divide and conquer" method. Since a sub-problem takes place in a small, convex polygon, the path from the initial point to the goal area, will most likely already start with a direction pointing, to a certain angle, to such area. Thus, the initial vectors of movement for each mode can be checked and the modes filtered based on this condition.

Additionally, the robots could save each solution and compare each new problem with its database. This could save time in a situation where some initial part of the plan for the new problem coincides with a part of an old problem. This part of the old plan can be "reused" in the new plan, and the decision making process for those steps skipped, saving a portion of time.

Finally, subsequent models should work in an effort to recreate the real conditions the system will be subject to, something this work does not take into account, since it is an early sketch of a possible tool. As such, it is crucial that some things are taken in consideration, the flow equations should account for mass, drag, inertia and other properties of motion in micro-gravity and the model should evolve to three-dimensional space. The mechanics and logistics of switching between anchor points, whichever way it is decided to happen, as well as how much time this switch takes are also important aspects to consider.

Acknowledgments

The author would like to acknowledge his dissertation supervisors Prof. Rodrigo Ventura and Prof. Mikoláš Janota for their knowledge and support and for giving me this idea to explore.

References

[1] S. Kong, S. Gao, W. Chen, and E. M. Clarke, "dReach: δ-Reachability Analysis for Hybrid Systems," in *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, ser. Lecture Notes in Computer Science, C. Baier and C. Tinelli, Eds., vol. 9035. Springer, 2015, pp. 200–205.

[2] S. Gao, S. Kong, and E. M. Clarke, "dReal: An SMT Solver for Nonlinear Theories over the Reals," in *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, ser. Lecture Notes in Computer Science, M. P. Bonacina, Ed., vol. 7898. Springer, 2013, pp. 208–214.

[3] M. Ghallab, D. S. Nau, and P. Traverso, *Automated Planning and Acting*. Cambridge University Press, 2016.

[4] S. Gao, S. Kong, W. Chen, and E. M. Clarke, "δ-Complete Analysis for Bounded Reachability of Hybrid Systems," *CoRR*, vol. abs / 1404.7171, 2014.

[5] R. Alur, C. Courcoubetis, T. A. Henzinger, and P. . H. Ho, "Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems," in *Hybrid Systems*, ser. Lecture Notes in Computer Science, R. L. Grossman, A. Nerode, A. P. Ravn, and H. Rischel, Eds., vol. 736. Springer, 1992, pp. 209–229.

[6] J. A. Wolfe, B. Marthi, and S. J. Russell, "Combined Task and Motion Planning for Mobile Manipulation," in *Proceedings of the 20th International Conference on Automated Planning and Scheduling, ICAPS 2010, Toronto, Ontario, Canada, May 12-16, 2010*, R. I. Brafman, H. Geffner, J. Hoffmann, and H. A. Kautz, Eds. AAAI, 2010, pp. 254–258.

[7] N. T. Dantam, Z. K. Kingston, S. Chaudhuri, and L. E. Kavraki, "Incremental Task and Motion Planning: A Constraint-Based Approach," in *Robotics: Science and Systems XII, University of Michigan, Ann Arbor, Michigan, USA, June 18 - June 22, 2016*, D. Hsu, N. M. Amato, S. Berman, and S. A. Jacobs, Eds., 2016.

[8] S. Nedunuri, S. Prabhu, M. Moll, S. Chaudhuri, and L. E. Kavraki, "SMT-based synthesis of integrated task and motion plans from plan outlines," in *2014 IEEE International Conference on Robotics and Automation, ICRA 2014, Hong Kong, China, May 31 - June 7, 2014*. IEEE, 2014, pp. 655–662.

[9] D. Bryce, S. Gao, D. J. Musliner, and R. P. Goldman, "SMT-Based Nonlinear PDDL+ Planning," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, B. Bonet and S. Koenig, Eds. AAAI Press, 2015, pp. 3247–3253.

[10] J. Fernández, B. Tóth, L. Cánovas, and B. Pelegrín, "A Practical Algorithm for Decomposing Polygonal Domains Into Convex Polygons by Diagonals," *Top*, vol. 16, no. 2, p. 367–387, 2008.