Contents

1 Planner

This section describes the planner algorithm implemented in the project.

A planner's purpose is, given a set of trajectories that share the same dimension, to generate a sequence of points, belonging (or not) to those trajectories.

In order to describe its main objectives more accurately, we need to elaborate a bit on the notion of trajectory.

1.1 Trajectory

Definition : a trajectory t is a continuous application from a segment of R in R^n :

$t : [a, b] -> R^n$, a and b two real numbers, n a positive integer, t continuous.

Naming convention :

- n : the dimension of the trajectory;
- x in [a, b] : an index of the trajectory;
- t(x), x in [a, b] : a point of the trajectory;
- a : the minimal index, the initial index;
- t(a) : the initial point;
- b : the maximal index, the final index;
- t(b) : the final point;

A trajectory t represents a path in a particular space from its initial point t_i to its final point t_f.
Its aim is to fully determine the path that some system must follow in this space;

1.2 Hook mode

Let us consider now, two trajectories, u and v that share the same dimension, and must be followed one after the other, u_i, u_f, v_i, v_f initial and final points of u and v. If both trajectories are joint, ie $u_f = v_i$, the path between u_i and v_f is fully determined. If trajectories are disjoint, the path is simply undetermined, as there is no trajectory defined between u_f and $v_i$to do so.

A planner can accept only a sequence of consecutively joint trajectories, or can accept any sequence of trajectories. In the second case, it must contain a hook algorithm, that is in charge of determining an arbitrary path from u_f to v_i. The planer has to work modes, trajectory mode, where it processes a trajectory, and hook mode, where it attempts to reach a trajectory's initial point;

1.3 Transition point

A transition point is a point where a brutal change of direction occurs.

As a planer can be used to control physical systems, that may suffer of this kind of changes, it may be important to monitor the existence of those points. A planner can offer this monitoring, so that actions can be taken by another piece of code.

The presence of transition points inside trajectories is uncertain, and must be determined manually; A planner can also offer this functionality, but it involves traversing the trajectory entirely, and it can be heavy for some low-performance systems;

However if we cannot easily detect transition points inside a trajectory, there are two immediate transition points, at extremal indices.

When completing a trajectory (reaching its final point), the planner may, enter into hook mode. The hook algorithm is used to determine a path to the next trajectory's initial point. This is the first transition point, as the direction the planner takes may not be the direction at the end of the trajectory;

When the hook algorithm manages to reach the next trajectory's initial point, the planner goes into trajectory mode, and starts searching the trajectory for an adequate point. This is the second transition point, as the direction the next trajectory takes may not be the direction the planner took to reach its initial point.

1.4 Managing transition points

The planner maintains a list referencing transition points in their order of appearance in the global path;

This list verifies the following conditions :

- when the planner has processed all trajectories, the list is empty;
- when the planner is processing a trajectory, the first transition point in the list corresponds to the current-trajectory's final point;
- when the planner is hooking the next trajectory, the first transition point in the list corresponds to the initial point of the trajectory to hook;

This list is updated the following way :

- When adding a trajectory to the planner's list, the planner determines if the trajectory is joint to its predecessor. If so, it modifies the last transition point, to reflect the direction change due to the trajectory, and marks the transition to trigger trajectory mode. If not, it modifies the last transition point, to reflect the direction change due to the hook algorithm, and mark it to trigger hook mode; it then adds a transition point to monitor, at the initial point of the new trajectory, to reflect the direction change between the hook aglorithm and the trajectory; Then, it attempts to find transition points in the trajectory. If it finds some, it adds them in their order of appearance; As stated earlier, this operation is time-consuming, and can be skipped. Finally, it creates another transition point to monitor, representing the stop at the end of the new trajectory. This transition will be modified, when a new trajectory will be added, as described just before, to reflect a direction change instead of a stop.
- When processing a trajectory, when a transition point is reached inside the trajectory, the planner removes the first transition point of the list;
- At trajectory mode exit, when the planner reaches the last point of a trajectory, or at hook mode exit, when the planner has hooked the first point of a trajectory, it removes the first transition point, and depending on its type, it enters into hook mode, or trajectory mode;

1.5 Trajectory mode

Following a trajectory is a primary objective for a control system. It is the base objective. The planner's objective is to decide of a target position. After this decision, multiple corrections or error sources, or perturbations may intervein, that the planner will not be aware of. This will result in a final position that may not be the one the planner previously picked. Because of this, to choose a target point, the planner must take the current known position into account.

The planner will comprise an algorithm whose purpose is, given a position P (that may not belong to the trajectory),to select a point of the trajectory that verifies a distance condition. For example, the planner may have to pick a point of the trajectory whose maximal or minimal or average distance from the provided position is within a certain range $[d_{min}, d_{max}]$.

More verbosely, the planner disposes of a distance function $d : R^n \times R^n \to R^+$, that takes two points in arguments, and returns a real number, that we will call the distance. It also posseses a target interval $I = [d_{min}, d_{max}]$, and a distance target d_{target}, that verify $d_{min} < d_{target} < d_{max}$. The planner's goal will be to find a point Q, as $d(P, Q) \in I$, as close as d_{target} as possible.

Given t a trajectory to process, P the position, d the distance function, we can define D the reduced distance function : $D : R \to R^+, i- > d(P, t(i))$ that gives the distance from the position of the point of t at index i. This function will indicate the planner if a point is close enough to the current position to be considered as already visited.

The objective of the planner is, given a position P, a trajectory t, a distance function d, a probable index i_e (e for estimate), and a probable index delta δ_i to determine, in a minimal number of computations, an index i_f (f for final) as $f(i_v)) \in [d_{min}, d_{max}]$. i_e and δ_i are two important parameters. The first will determine the first index whose distance should be computed. If its distance is not in the target interval, δ_i will be used, to determine the next point to be evaluated.

This problem as it is a classical analysis problem, and can be solved with many methods; Though, an aditional rule comes to complexify the resolution. Indeed, as the planner's goal is to traverse the whole trajectory, it should prioritise points with high indices. A simpler rule that is not equivalent but easier to work with, is that the cost should increase when increasing the index, at the choosen point.

A fact to remember, is that the index where the search begins is purely speculative. In case of inacurate prediction, different cases can occur.

1.6 Prediction

We develop here the first stage of the determination of a point. Depending on the exactitude of i_e, several cases are encountered.

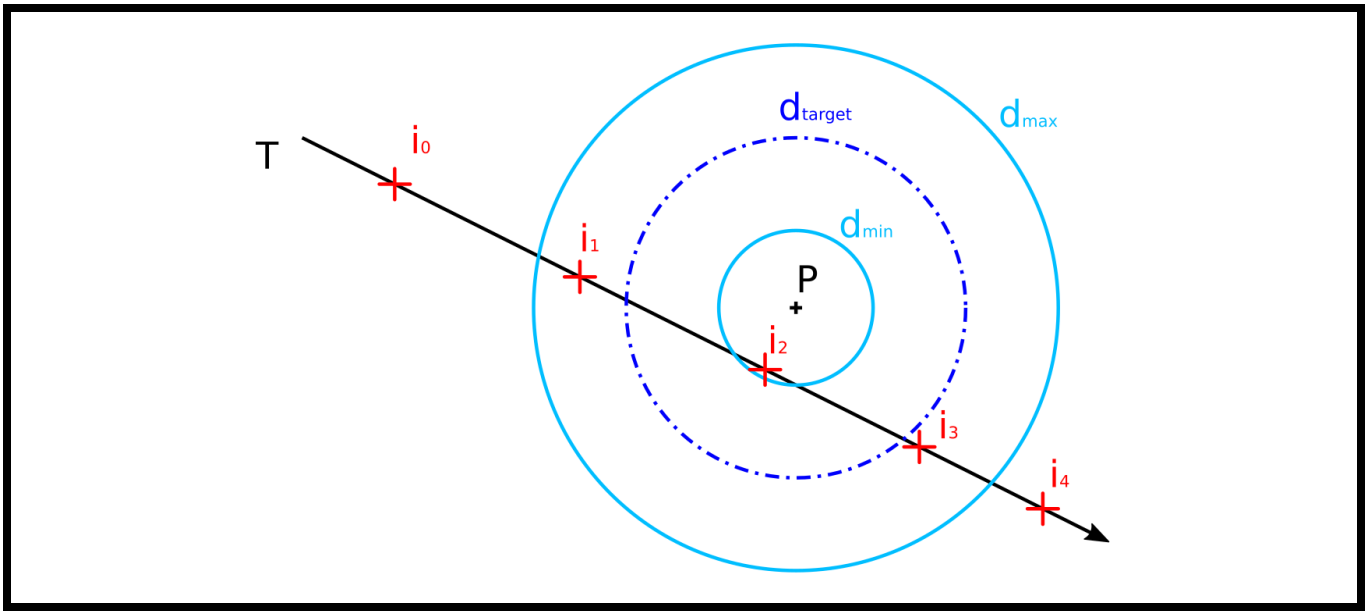

Figure 1: Different cases, depending on the index prediction

The figure behind shows different cases that may occur, for different values of i_e, with a simple example : the trajectory T to follow is a simple 2D line, and the distance considered is the geometric distance. P the current position, and circles delimitating the acceptable positions set are drawn accordingly.
Different cases are :

- i_0, critical traction : i_e is highly underestimated, the distance to the posion being superior to the maximal bound, incrementing the index making this distance decrease. In this situation, simply increasing the index until a valid one is found is not enough, because it would lead to choose a point where the previously enounced rule doesn't apply. In this case, the planner must increase the index, until the distance starts increasing and consider points beyond that index only.

- i_1, ghost traction : same case as before, but this time, the estimation gives a valid distance. This case is a little bit problematic, as the estimation seems to be valid, but is a matter of fact, counterveins to our decision rule. In this case again, the index must be increased until the distance increases, and only consider points beyond that index.

- $_i2$, traction : the prediction lead to a point too close to the current position. This point is already visited, and can't be selected. Once again, the index must be increased until the distance increases , and only indices beyond must be considered during the search.

- i_3, correct prediction : the prediction lead to an acceptable point.

- i_4, resistance : i_e is overestimated, the distance to the position being superior to the maximal bound, incrementing the index making this distance increase. A suitable point must be seatched before the estimated index.

1.7 Search procedures

In the list behind, two procedures were implicitely refered :

- the slide. Given two indices and their associated distances, the trajectory is traversed in the local descending direction, until a local minimum is found, or until the distance decreases over a certain target value, or the slide arrives to an extremal index;
- the raise. Given two points, and their associated distances, the trajectory is traversed in the local descending direction, until a point at a sufficient distance is found, or the raise reach an extremal index;
- the windowing. Given two points, and their associated distances d_0 and d_1, the trajectory is traversed until a point at a position belonging to an interval $D \in [d_0, d_1]$ is found.

1.8 Algorithm

At the beginning stage of the algorithm, only the point at i_e and its distance are computed. No indication is available on how the distance will vary when the index does. As you may have seen, this information alone does not suffice to determine in which case we are. We can only determine that we are in i_1 or i_3, or that we are in i_2, or that we are in i_0 or i_4. To go further, a second point will be determined, its index being given by δ_i, and different actions will be taken.

Different cases are :

- i_1 or i_3 : within the acceptable bounds : the second point is used to determine the current case. If the distance increases when the index increases (positive derivative), we are in case i_3 and the point is acceptable. If not, we are in case i_1, and a slide must be executed. At the local minimal point, we are either in case i_3, which gives us an acceptable point, or in case i_2.
- i_2 : below the minimal distance : the estimated point is considered as already visited. A raise must be executed, to the minimal distance. If a point below the maximal distance is found, we are in i_3, which gives us an acceptable point. If a point beyond the maximal distance is found, the windowing gives us an acceptable point. If no point is found, the trajectory is finished.
- i_0 or i_4 : beyond the maximal distance : the second point is used to determine the current case. If the distances increases with the index increasing, we can suppose we are in i_4. If not, we are in i_1. A slide is executed, targetting the maximal value, if we are in i_1. If in i_4, we need to reach the minimum. Here comes a problem : there is no guarantee that the local minimum is in the distance bounds, as the direction to take is extrapolated from evaluated points, which has no guarantee to be accurate. There is either no guarantee that a point in the distance bounds exists, as the current position could be too far from the trajetory. If we are in i_4, if a local minimum was detected, and the maximal distance has never been reached, the planner has failed to compute the point. If the maximal distance has been reached, we either have determined a valid point, or can use the windowing to compute one. If we are in i_1, and the local minimum is not lower than the maxmal distance, the planner has failed to compute the point. If not, we end up in i_3, which gives us a valid point, or in i_2, which leads us to a valid point.

2 Planner data structures

This section discusses the format of data structures in the planner implementation.

The planner implementation comprises five main data structures : the trajectory, the panner element, the transition, the transition factory and the planner;

2.1 Trajectory

A trajectory structure, as introduced in the section about the planner's algorithm, is define like below :

```
1
2  struct trajectory {
3
4    //The dimension of the trajectory;
5    const size_t t_dim;
6
7    //The start index, minimal value of the index;
8    const float t_imin;
9
10   //The stop index, maximal value of the index;
11   const float t_imax;
12
13   //An index increment that can cause significant increments on point coordinates;
14   const float t_dincr;
15
16   //The trajectory function;
17   void (*const t_function)(float index, float *dst_array, size_t dst_size);
18
19 };
```

This is the root structure for all trajectories.

A particular trajectory will use composition to allow pointer cast, like below :

```
1
2  struct rdm_trajectory{
3
4    //Composition, with a trajectory struct;
5    struct trajectory traj_data;
6
7    //rdm_trajectory struct data;
8    ...
9
10 };
```

2.2 A trajectory's lifetime in the planner

A trajectory is not created by the planner. An external program is in charge of creating the trajectory of the right type, and to transfer it to the planner. The planner will store it in a list, process it entirely, providing chosen coordinates. During its processing, the trajectory can be consulted by another piece of code, for other computation. When it its processing is complete, the trajectory can be deleted. As the planner did not create it, it does not delete it. This task is delegated to another piece of code.

This transfer of the trajectory to the planner can by done directly by reference : the algorithm creates an instance of appropriate type of trajectory, and passes a reference to the planner.

This solution raises a problem, when applied to a real use case. Indeed, the trajectory itself is only a part of what will define the movement. Once the current point of a trajectory and its index have been chosen by the planner, some other piece of code may use this information to complete the target. For example, if we want to draw a line on a 2D machine, and to increase the speed linearly from one speed to another, the speed target must be evaluated after the target trajectory position is determined, from the chosen index.

In reality, the trajectory is a part of some more complex object that is meant to determine the movement. This kind of struct cannot contain a trajectory struct directly (composition) because the trajectory struct only has a meaning when it is composed to form an actual trajectory.

2.3 Planner element

This problem is solved by introducing the planner object struct :

```
1
2  struct planner_elmt {
3
4    //A list head, for list storage;
5    struct list_head pe_head;
6
7    //The trajectory that the planner should process;
8    struct trajectory *pe_traj;
9
10 };
```

This structure contains a list head for storage in the planner, and a trajectory struct ref.

It can be used freely in composition.

2.4 Transition

A transition references a point where some brutal direction change occurs. Its is formated like below :

```
1
2  /**
3   * A transition represents a sudden change of direction, from dir0 to dir1, defined by their
       coordinates, a at
4   *  certain position;
5   *
6   *  The transition struct only provides base data : point and directions coordinates.
7   *
8   *  A regulation algorithm will use this data as a base and compute its proprietary data from it.
9   */
10
11 struct transition {
12
13   //A list head, for storage in the planner;
14   struct list_head t_head;
15
16     //The position of the transition point;
17     float t_pos[];
18
19     //Movement direction before the direction change;
20     float t_dir0[];
21
22     //Movement direction, after the direction change;
23     float t_dir1[];
24
25 };
26
27 The transition struct aims to store basic transition data, as stated previously, but also, to
       support
28 composition, so that a regulation algorithm can define its own type of transition structure.
29 Indeed, the transition struct only provided point and direction coordinates, which is not enough
       for
30 a proper regulation algorithm. Proprietary (implementation dependent) data are to be computed for
       each
31 transition, and those need a place to be stored in.
32 \newline
```

2.5 Transition factory

Let us examine the lifecycle of a transition struct in the planner :

- A transition struct is created at the will of the planner.
- When the transition is updated (at its creation or its modification), proprietary data must be updated accordingly.
- When the transition is outdated, it must be deleted, at the planner's will again. Though, as stated before, the planner doesn't know the exact format of the struct that it manipulates, only that it is composed by a transition struct. It can't by himself ensure the proper deletion. The same fact applies to the creation.

These actions will be taken by another piece of code, the transition factory, that is, as stated, in charge of creating, updating, and deleting proprietary transition structs. The transition factory is formated like below :

```
1
2  struct transiton_factory {
3
4    //Create a transition.
5    struct transition *(*create_transition)(struct transition_factory *factory, size_t dimension);
6
7    //Update proprietary data; The factory, the transition to update,
8    // and the ref of the list's first transition are provided
9    void (*update_transition)(
10     struct transition_factory *factory,
11     struct transition *trans,
12     const struct transition *first_trans
13   );
14
15   //Delete a transition;
16   void (*delete_transition)(struct transition_factory *factory, struct transition *trans);
17
18 };
```

At any moment, the transitions list may be queried by any external piece of code, that is aware of the proprietary struct format, for regulation purposes.

2.6 Planner modes

As described in the section about planner algorithm, the planner has two modes :

- hook mode : the planner must hook the next transition point;
- trajectory mode : the planner must process the current trajectory;

The planner_mode struct is with no surprise, formatted like below :

```
1
2  enum planner_mode {
3
4    PLANNER_HOOK_MODE,
5
6    PLANNER_TRAJECTORY_MODE,
7
8  };
```

2.7 Planner

At last, we introduce the planner data structure, that keeps all things together.

```
1
2  struct planner {
3
4    //The dimension of the planner.
5    const size_t p_size;
6
```

```
  //The list of planner elements;
  struct planner_elmt *p_elmts;

  //The current transition;
  struct transition *p_trans;

  //The transition factory;
  struct transiton_factory *p_tfactory;

  //The current index in the current trajectory;
  float p_index;

  //The current increment in the current trajectory;
  float p_incr;

  //The minimal distance bound;
  const float p_dmin;

  //The target distance;
  const float p_dtarget;

  //The maximal distance bound;
  const float p_dmax;

};
```

The planner stores references to all previously described objects, and some other fields (p_index, p_incr, p_dimn, p_dtarget, and p_dmax) that are used when processing a particular trajectory.