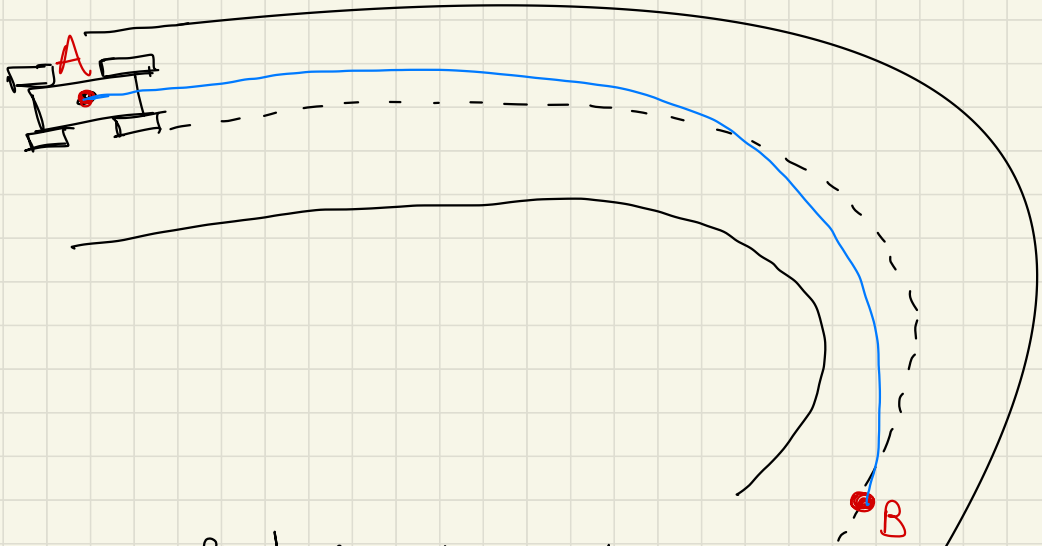
MOTION PLANNING WITH RRT* AND MOTION PRIMITIVES



TARGET → find the minimum time trajectory to move from A to B.
Static/dynamic obstacles can be part of the scenario

POSSIBLE USE CASES:

→ Real-time motion planning in the presence of static/dynamic obstacles

→ providing a trajectory guess to be used for on-line MPC

e.g. avoid solving 2 OCPs to decide whether to overtake an obstacle from the right or from the left

→ ...

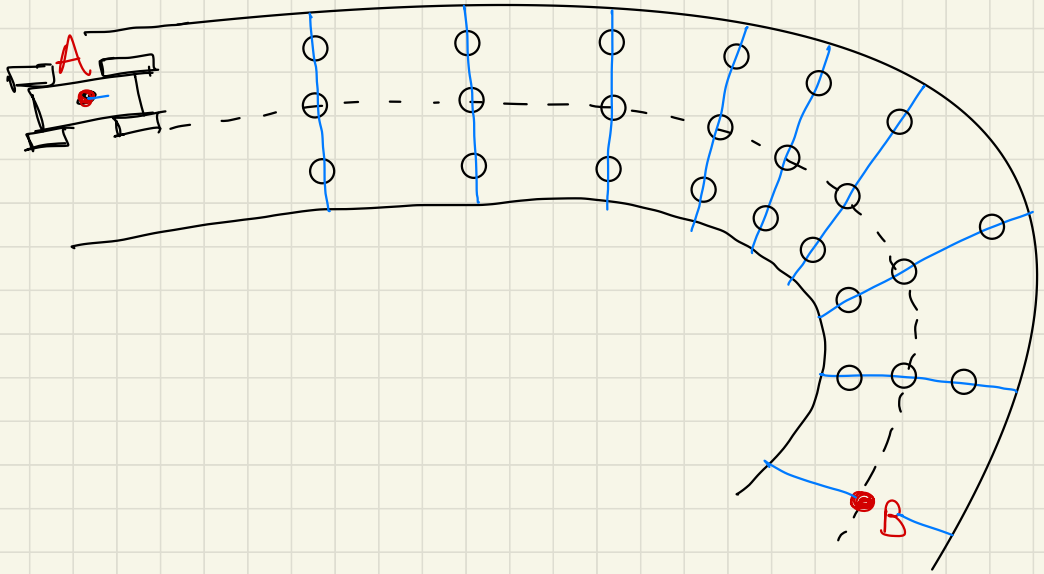
SKETCH OF METHOD 1

STEP 1 → define waylines and waypoints basing on:

✓ → local curvature of the road

✗ → local predicted vehicle speed (optional)

✗ → obstacles position (and velocity)



STEP 2 → HYBRID RRT* – FORWARD/BACKWARD

✓(1) Randomly sample one of the previously generated waypoints (point C)

✓(2) build a clothoid from A to C. (x_A, y_A, θ_A), (x_C, y_C, θ_C) are known. I can consider θ_C = local road orientation in C (relative yaw angle = 0)

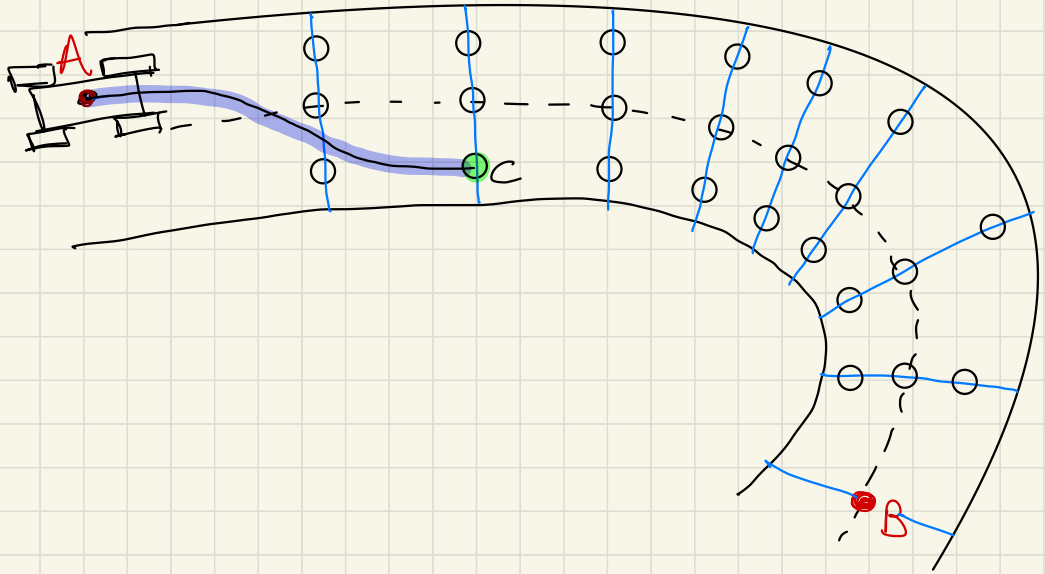
✓(3) compute the optim speed profile. I know v_A,

$v_{C_{MAX}} = \sqrt{\frac{a_{y max}}{k_C}}$, k_C = local clothoid curvature in C

↓ it's an upper bound

⇓

Apply the FW-BW and store the min time = cost(AC) and store the optimal v_C (it can be $< v_{C_{MAX}}$)



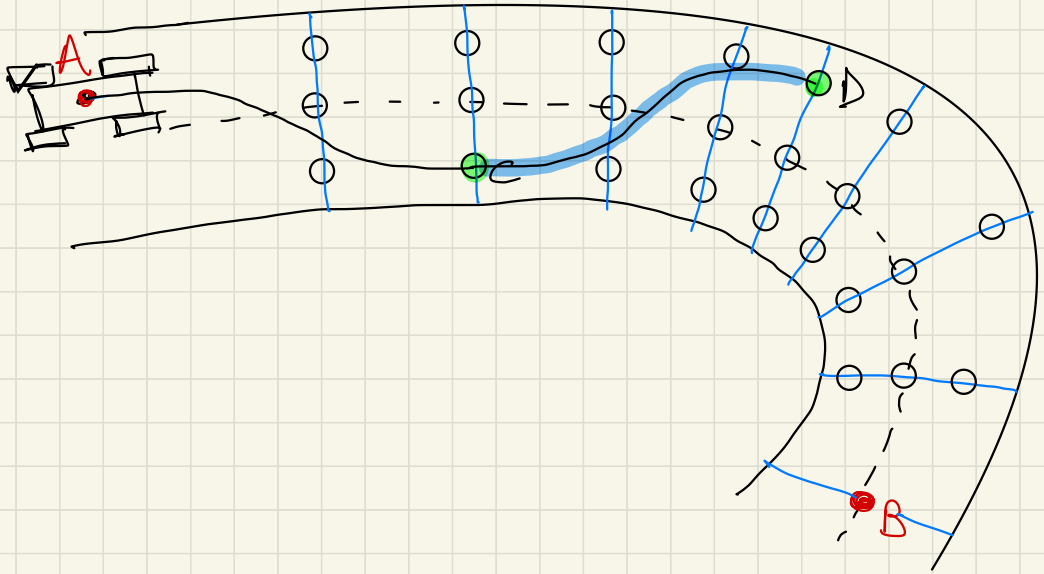
✓ (4) Sample a new point (D)

✓ (5) Apply the RRT* to expand the graph

✓ → (5a) → compute the best parent node for D, chosing from the nodes already explored lying e.g. in the previous 3 waylines

✓ → (5b) → try to rewire the graph → not possible in this case since there are no explored nodes ahead of D

The "cost" for a connection is the min time, i.e. the solution given by the FW-BW



Iterate until convergence or until the max computational time is reached

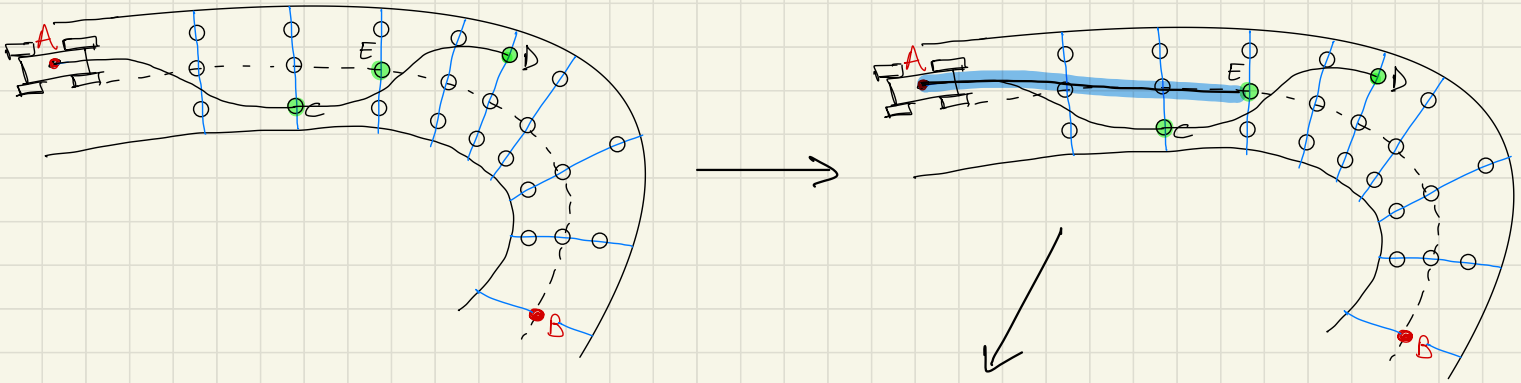
→ Sample a new point E → the best parent for E is to be chosen among A and C ✓

↙ Compare the FW-BW on $(\widehat{AC} + \widehat{CE})$ and on $(\widehat{AE})$ → $\widehat{AE}$ has a lower cost ⇒ create the $\widehat{AE}$ connection

↙ ✓ then Try to rewire the graph → the only option to be evaluated is if the connection $\widehat{AE} + \widehat{ED}$ has a lower cost w.r.t. $\widehat{AC} + \widehat{CD}$

↙ in this case the new connection $\widehat{ED}$ is advantageous

// ✓ create $\widehat{ED}$ and remove the old $\widehat{CD}$ (rewiring)



NOTES:

✓ → Like RRT* the algorithm is asymptotically optimal (w.r.t. the list of waypoints selected at the beginning)

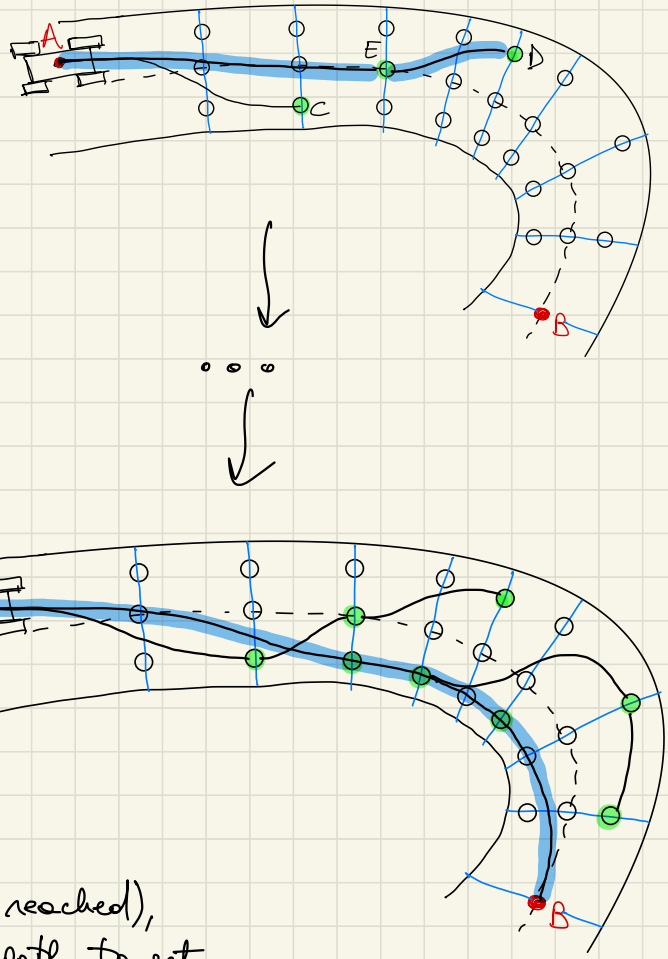
✓ → it should be more efficient w.r.t. the "brute force" comparison of all the possible combinations, especially if a max computational time is fixed

✓ → at each iteration, the min time and the final speed for the new connection(s) are stored

↓

but the FW-BW is applied, at each iteration, to individual clothoid segments

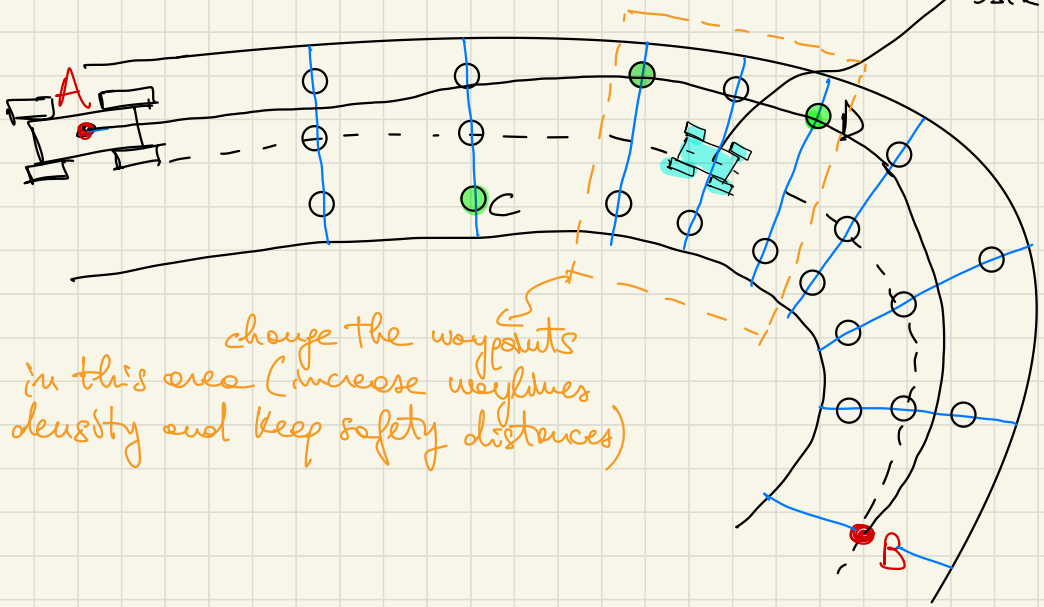
✗ → Once the algorithm converges (or max cpu time is reached), the FW-BW is applied to the entire final path, to get the full speed profile and min time



OBSTACLE AVOIDANCE

Obstacles (static/dynamic) are detected online ⇒ increase the density of waylines around the obstacle and ensure safety distances

if the obstacle is moving, use a prediction model (e.g. using analytical OC solutions) to improve the modification of the waylines/waypoints grid



I know position, v_x, a_x, n coord, Ωz

↓

calculate 1 manoeuvre for lane keeping and 1 for lane change, on a certain horizon, maximizing travel time

↓

calculate 2/3 manoeuvers, then compute FW/BW on them

X For the obstacle, I can assume to know its v_x, o_x, n coordinate, you note

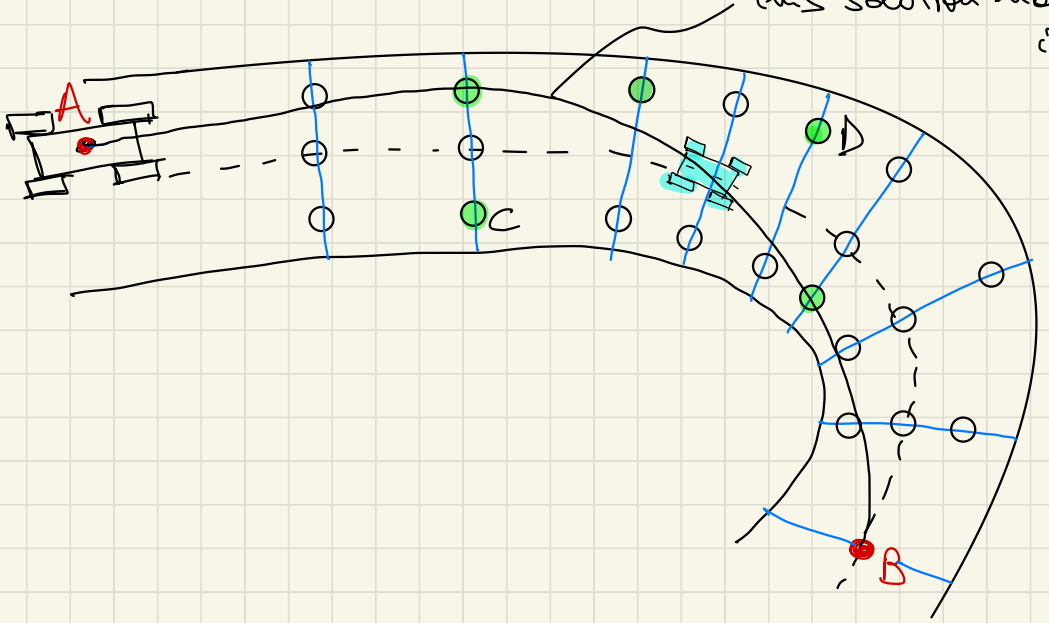
I can compute 2/3 possible maneuvers for the obstacle, on a certain horizon of around 50 m → because e.g. if I'm travelling at 60 m/s, then considering a time window of 1 sec. a horizon of 50 m can be ok

the obstacle path can be built with 1 or 2 clothoids and then apply the FW/BW to compute the speed profile for the obstacle

And then change waypoints and include check for collisions with clothoid library

exclude the clothoid segments that can collide with the predicted obstacle trajectory

X COLLISION CHECKING

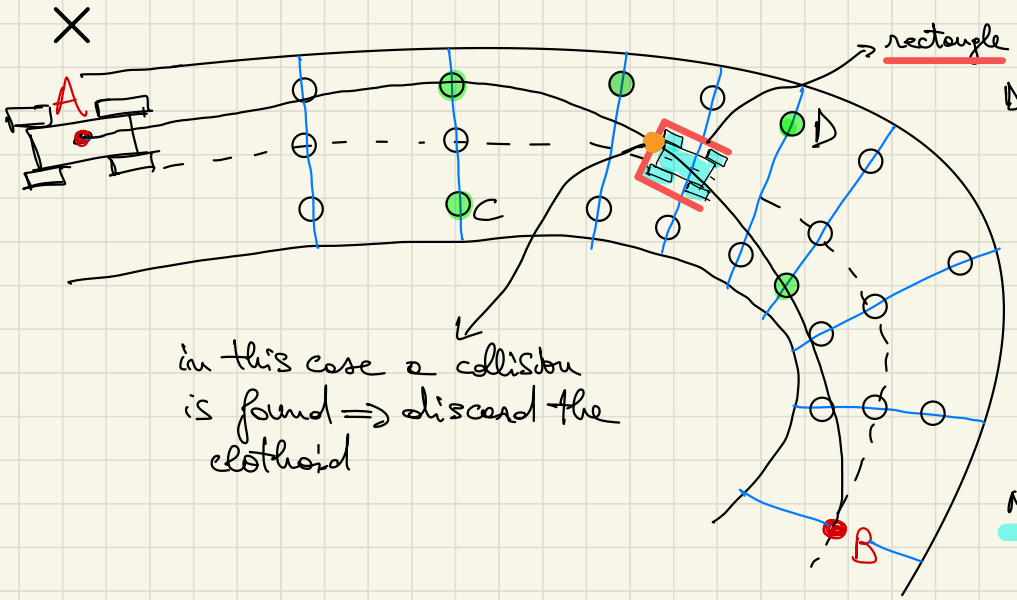


this solution has to be discarded since it collides with the obstacle

use the clothoid library to build a rectangular bounding box around the obstacle, and check if collisions occur between the trajectory (clothoids) and 3 of the edges of the bounding box

use the .intersect method of the clothoid library

refer to the file clothoids_intersect.m for more details



→ rectangle (3 edges) describing the bounding box. Define it with LineSegment objects

→ NOTE: check a collision only if an obstacle is detected in the current horizon, otherwise do not check for collisions with obstacles

in this case a collision is found ⇒ discard the clothoid

NOTE: collisions with obstacles should be checked every time a new connection is built or evaluated

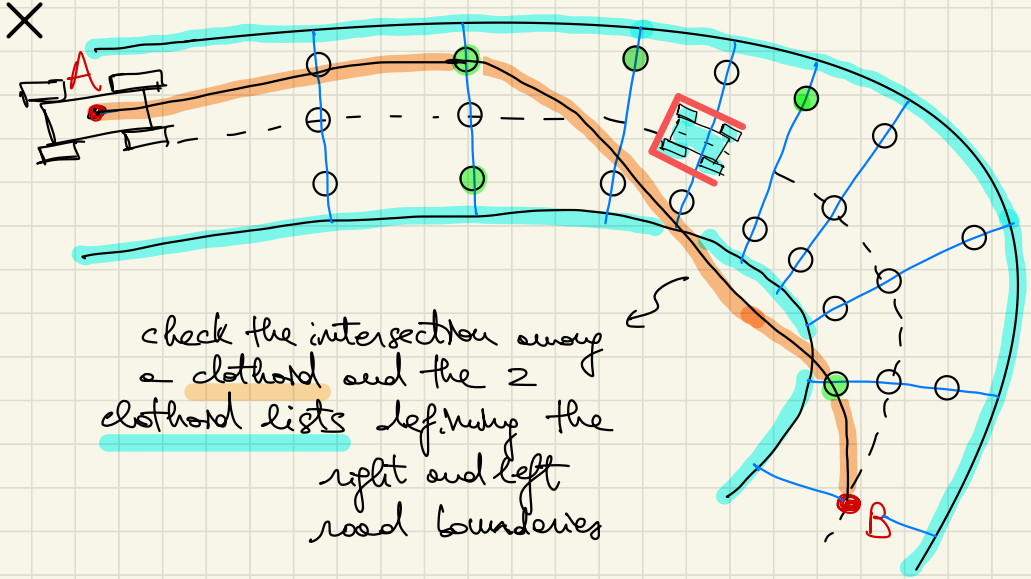
COLLISION CHECKING WITH ROAD BOUNDARIES → define the road boundaries as clothoid lists

↓

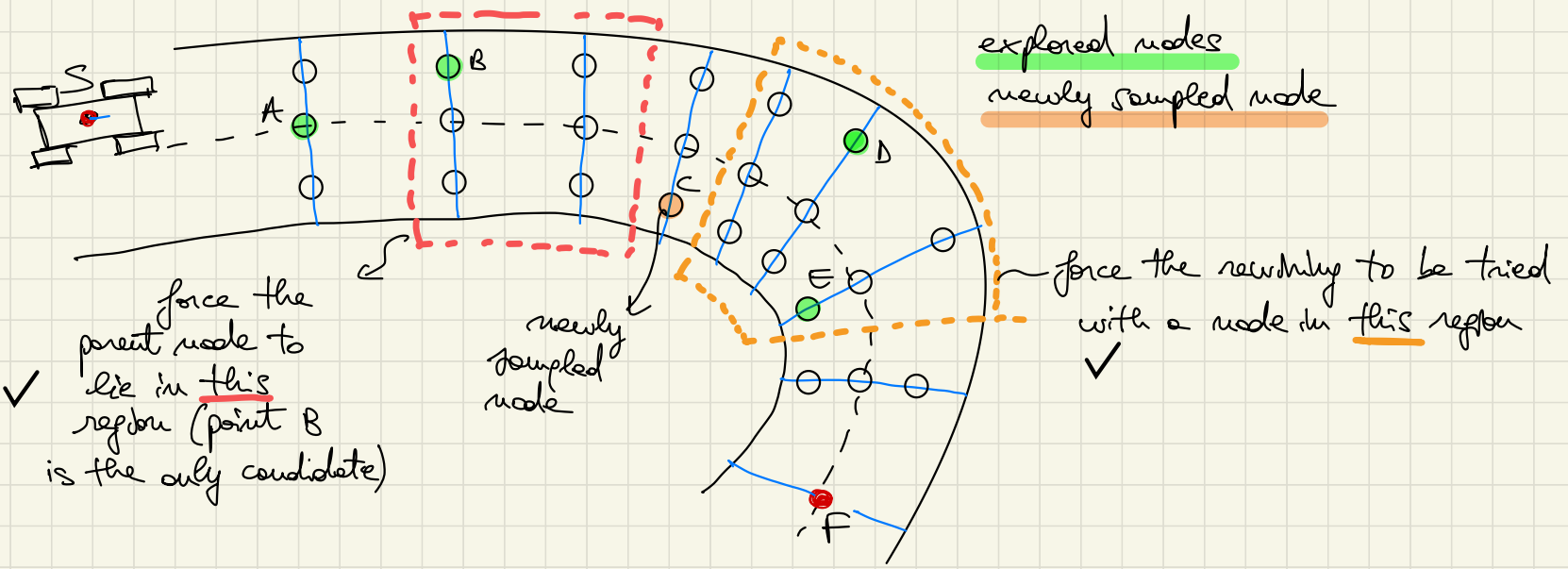
check the intersection between a generated clothoid and the road margins

↓

refer to clothoids_intersect.m for more details



NOTE: collisions with road margins can be avoided if the point node selection and rewiring are limited to the waylines lying in a certain region around the newly sampled point → this can allow avoiding the check for collisions with road boundaries



✓

✓

✓ => in this way the chance to violate the road margins is reduced

IMPROVEMENTS / VARIANTS

→ Elliptic G-G diagram in the FW-BW

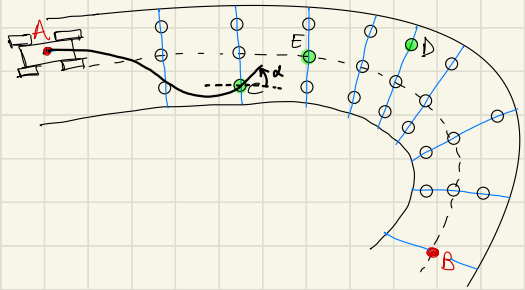
→ comparison with off-line min time OCP solution and use the optimal speed profile to generate the full set of waypoints (see the on-line motion planning section)

→ C++ implementation

→ min jerk (rather than min time) solutions can be obtained with analytical motion primitives (instead of the FW-BW)

→ allow a relative yaw angle $\alpha \neq 0$ between the vehicle trajectory and the orientation of the road in the waypoints positions

(e.g. define a finite set of α values and sample in that set)



→ improve the accuracy of the intermediate solutions

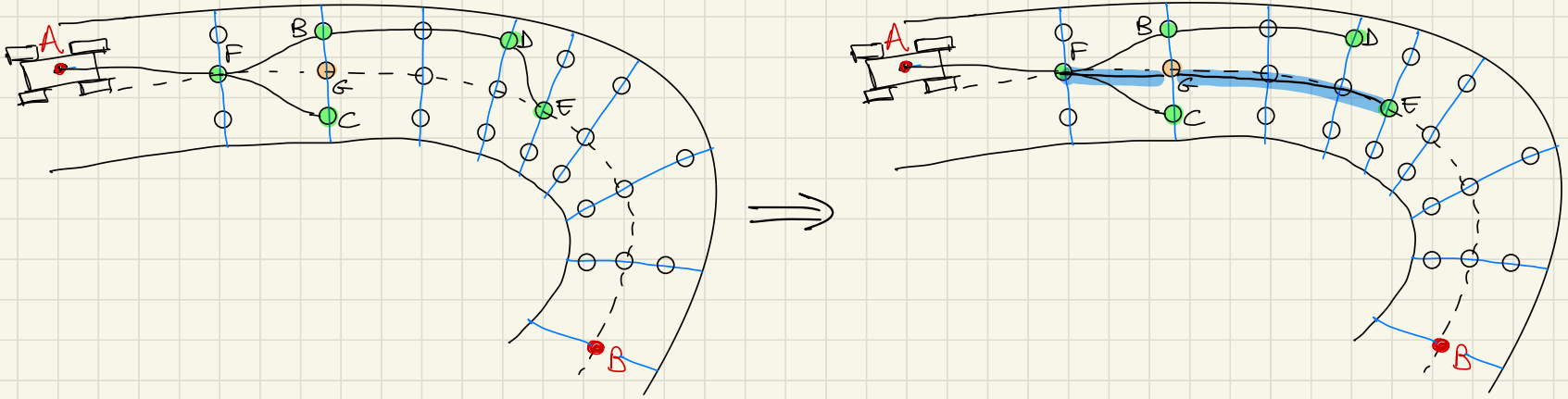
↓

METHOD 2

METHOD 1 vs METHOD 2 (method 1 is the one presented in the previous pages)

→ Referring to the sketch below, suppose that at a certain iteration the point G is sampled

→ the algorithm chooses F as the best parent, and rewiring is performed by connecting G to E and removing the $\overline{DE}$ connection



→ With method 1, the FW-BW is iteratively applied to simple clothoid segments. For example, to evaluate the rewiring, the algorithm computes the cost for the path $\overline{AFGE}$ as:

$$cost(AFGE) = cost(AF) + cost(FG) + cost(GE).$$

↓

→ cost(AF) is available from previous iterations, and it doesn't need to be recomputed

→ cost(FG) was computed when calculating the parent node for G, so it's available

→ cost(GE) is yet to be computed

BUT computing cost(AFGE) = cost(AF) + cost(FG) + cost(GE) is suboptimal

a more accurate solution is obtained if the Fw-Bw solution is applied directly to the full AFGE clothoid list, i.e. with method 2

METHOD 2

PROS:

→ even during the intermediate iterations, the Fw-Bw is applied to the entire clothoid lists

⇓

more accurate intermediate solutions

CONS :

→ it can be more expensive computationally

→ the clothoid lists should be iteratively stored → when the algorithm starts, I can initialize an empty set of clothoid lists, whose number is equal to the total n° of waypoints

↓

with the presented algorithm, at every iteration there exists at most 1 path connecting the origin A with another point (in the waypoints list), so it is possible to store a n° of clothoid lists equal to the n° of waypoints, and then update the clothoid lists during the iterations

↓

e.g. to compute cost(AFGE), start from the clothoid list AFG, that was stored, and do a push back of the point E

ON-LINE MOTION PLANNING

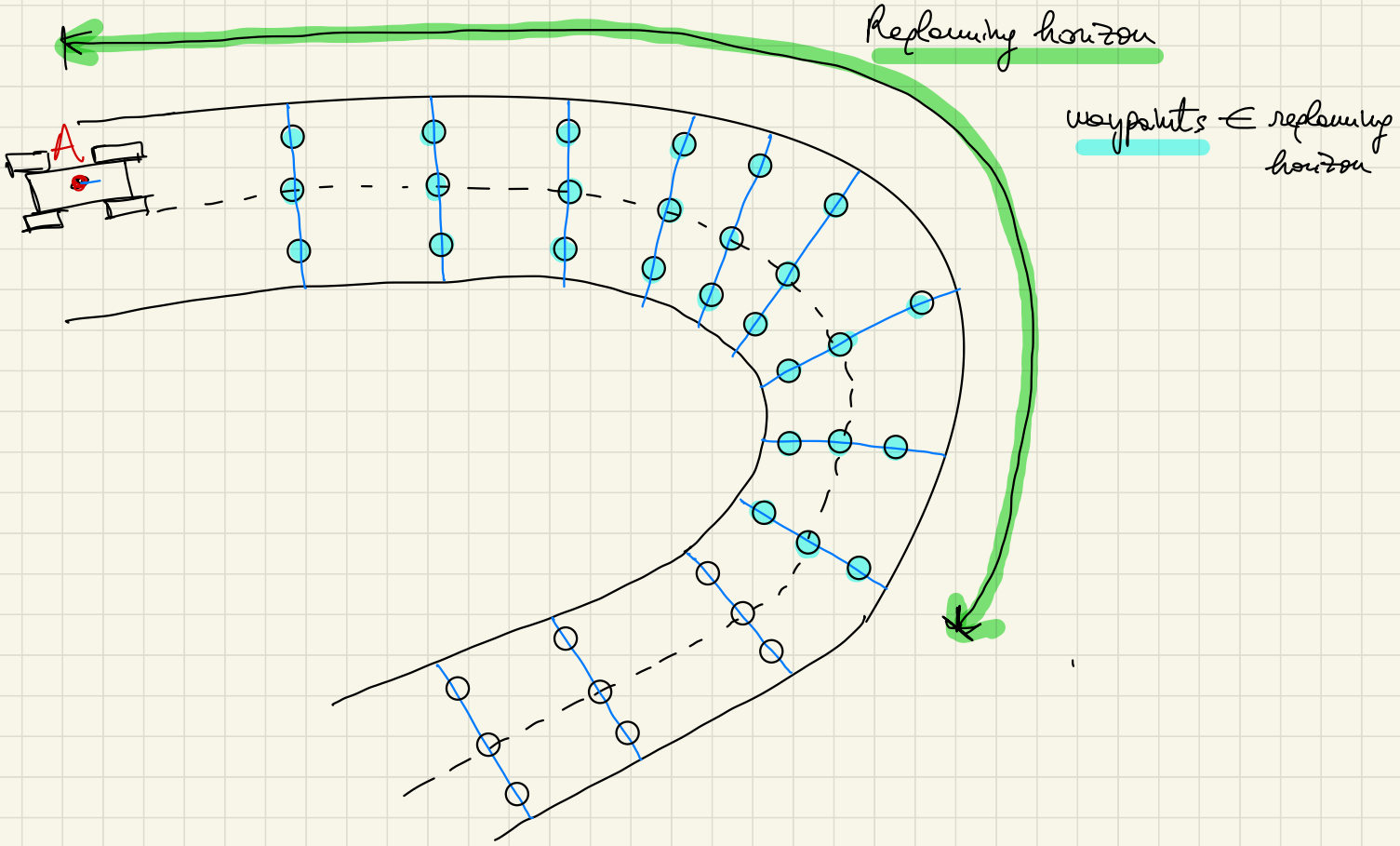
Possible ideas → (1) Define off-line the grid of waypoints, and the spacing among waylines is based on a (heuristic) function that depends on:

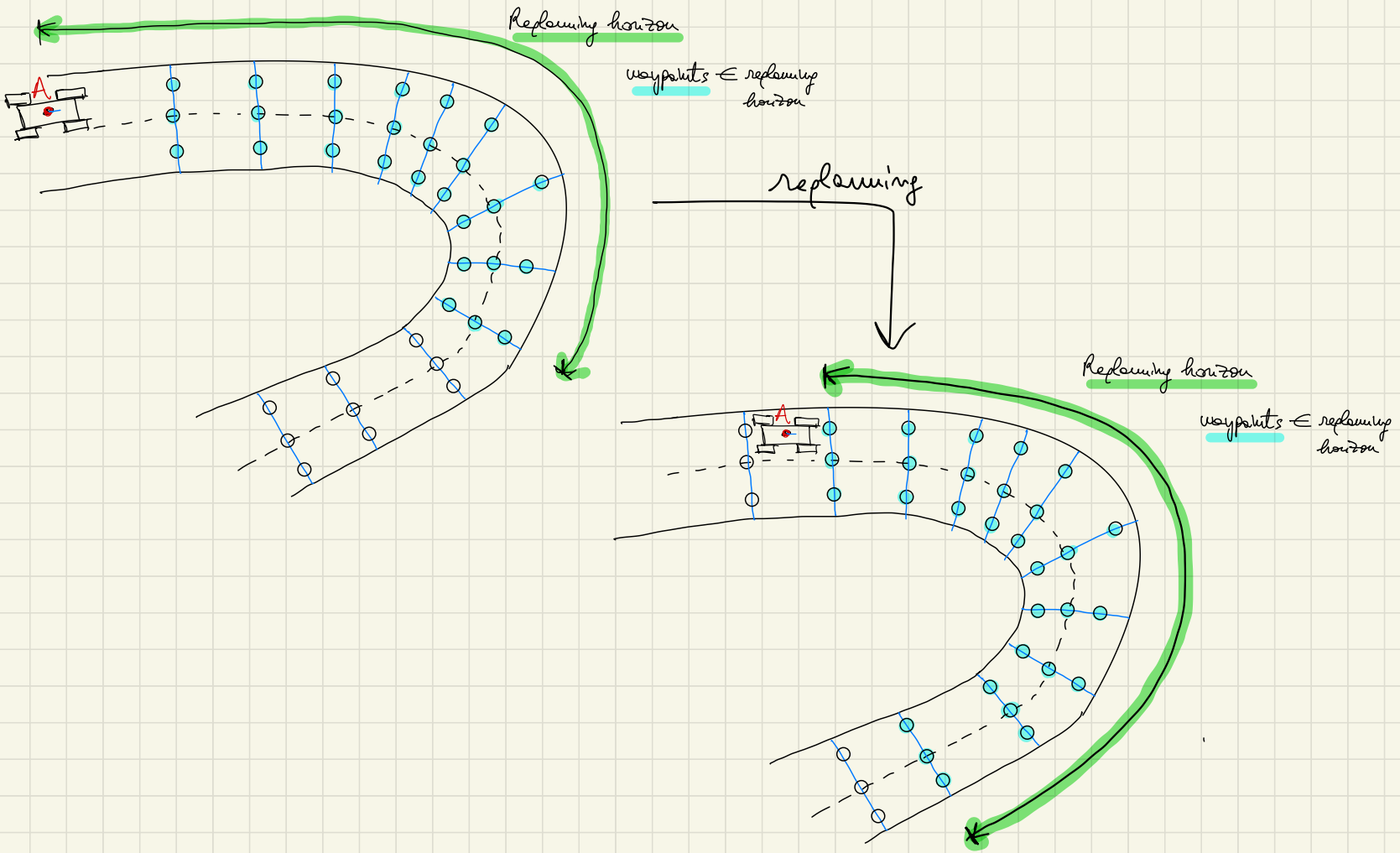
- → local curvature of the road
- → local speed estimate computed with an off-line min time OCP on the same circuit and with the same vehicle (optional)

and then on-line use a subset of the total n° of waypoints, i.e. the ones lying inside a prediction horizon, and change the position of some waypoints / waylines based on static/dynamic obstacles

→ (2) Define the waypoints on-line

SKETCH OF THE REPLANNING IDEA N°1



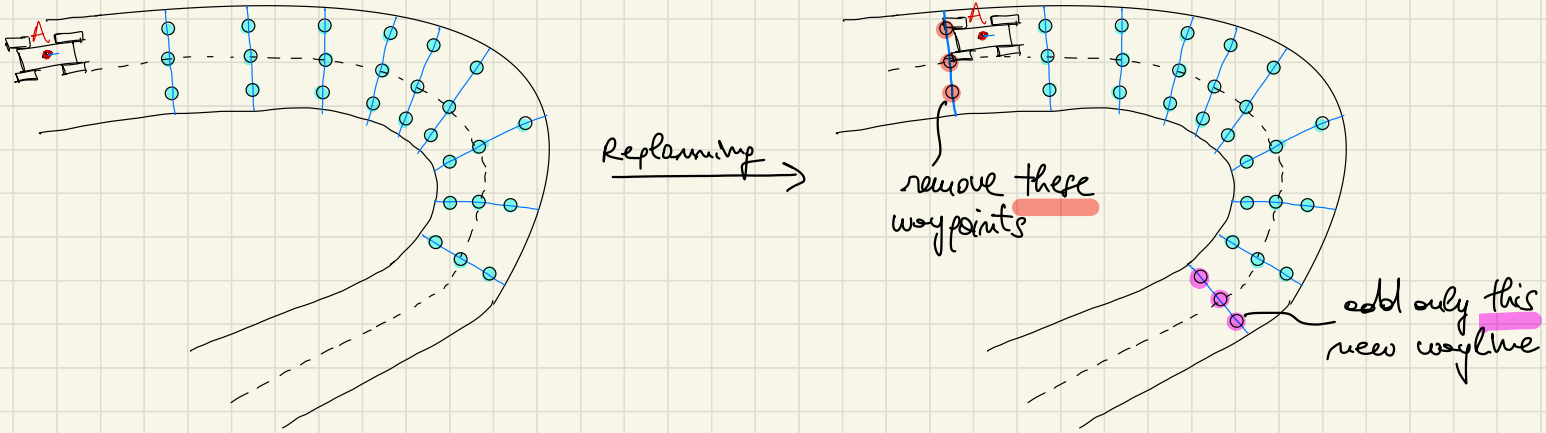

Replanning horizon
waypoints ∈ replanning horizon
A
replanning
Replanning horizon
waypoints ∈ replanning horizon
A

REPLANNING IDEA N°2

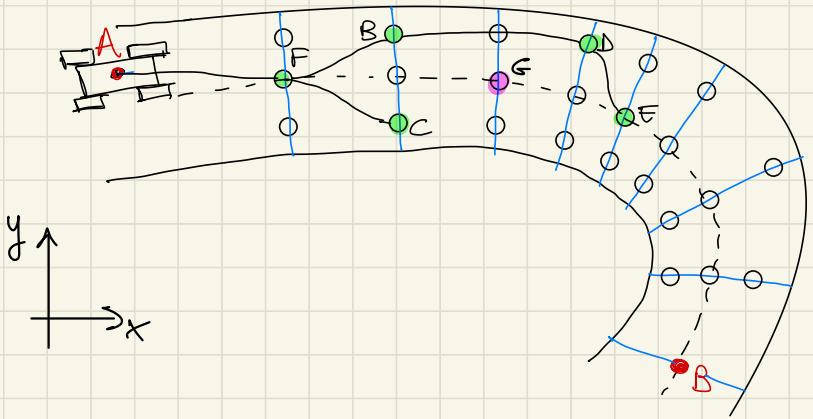
→ knowing $\{s, n\}$ of a waypoint I can get the corresponding $\{x, y, \theta\}$ with the clothoid library

or else get $\{x, y, \theta\}$ of the middle line, for a certain s, and then use rototranslation matrices to get $\{x, y\}$ of the waypoint

→ In the 1ST replanning step, I have to generate all the waypoints lying in the prediction horizon → but from the 2ND replanning step, I can keep the waylines previously generated and only add 1 or 2 new waylines



CHOICE OF THE PARENT NODE & REWIRING



In this case the already explored nodes are F, B, D, C, E and the newly sampled node is G

↓

the parent node is to be chosen among F, B, C (looking backwards w.r.t. point G)

↓

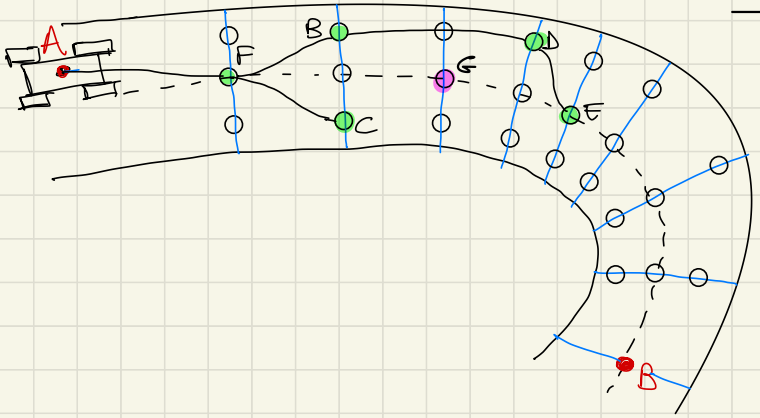
i.e. the points having a curvilinear abscissa s lower than the curvilinear abscissa of point G, named s_G (for example with respect to the point A)

CANDIDATE PARENTS — METHOD 1

So, to select the candidate parent nodes, the method (1) is to consider the explored waypoints belonging to waylines with curvil. abscissa $s < s_G$

CANDIDATE PARENTS – METHOD 2

Build an "exploration matrix" and use it to automatically select the candidate nodes for parent calculation and rewiring



⟶ in this case 3 waylines are generated, and each wayline contains 3 waypoints

↓

the exploration matrix E is 3×9 and it contains zeros for all the unexplored waypoints, and ones for the visited waypoints

↓

e.g. in this case

$$E = \begin{matrix} \\ F \\ \\ \\ \end{matrix} \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & - \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & - \end{bmatrix}$$

B D C G E

$$E = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & - \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & - \end{bmatrix}$$

candidate parents

candidate for rewiring

$\Rightarrow$ with the exploration matrix, the candidate nodes for parents / rewiring are trivially identified

To get the values of $\{x, y, \theta, k\}$ (curvature) of a waypoint, I can initialize off-line 4 matrices, named $X_{MAT}, Y_{MAT}, \theta_{MAT}, K_{MAT}$, initially filled e.g. with NaN values

And these matrices have the same size of the E matrix:

$$X_{MAT} = \text{ones}(3, 9) \cdot NaN, \quad Y_{MAT} = \theta_{MAT} = K_{MAT} = X_{MAT}$$

(initialize)

And also define a logic for the waypoints indexing:

$$I = \begin{bmatrix} 1 & 4 & 7 & & 22 & - \\ 2 & 5 & 8 & \cdots & 23 & 26 \\ 3 & 6 & 9 & & 24 & - \end{bmatrix}$$

$I = 0$ → starting point A

→ goal point B

matrix defining the indices for each waypoint

→ the matrices E, I, X_{MAT}, Y_{MAT}, θ_{MAT}, K_{MAT} have a dimension $3 \times 9 = n_r \times n_c$

And then on-line suppose that the waypoint n°8 is sampled → it corresponds to the element in position (2,3) in the matrices, then evaluate the $[x, y, \theta, K]$ values for that waypoint by doing: $[x, y, \theta, K] = roadClothoid.evaluate(S_{wayp})$

curvilinear abscissa of the waypoint (known in advance)

clothoid list providing the road center line

and then store these values in the matrices, so that they don't need to be computed again:

$x_{NAT}(2,3) = x$, $y_{NAT}(2,3) = y$, $\theta_{NAT}(2,3) = \theta$, $k_{NAT}(2,3) = k$

For rewiring it is useful to define another matrix, named P, to store the parent node for every sampled waypoint (note: each explored node has only 1 parent!)

P = ones(3, 9).NaN ← initialize

And then on-line, if the waypoint n°6 is sampled and the best parent node is the origin 0, then fill the matrix P with the value "0", in position "6" (i.e. in position (3, 2)):

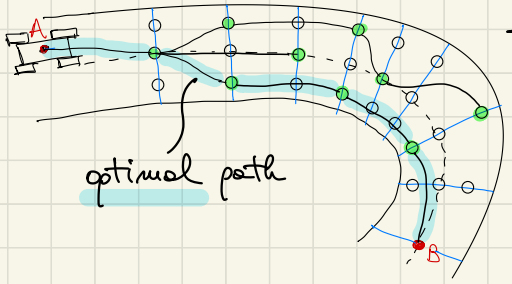
$$P = \begin{bmatrix} NaN & NaN & & NaN \\ NaN & NaN & \cdots & NaN \\ NaN & 0 & & NaN \end{bmatrix} NaN$$

→ then suppose that the node n°8 is sampled (index (2,3)) and the best parent for 8 is 6:

$$P = \begin{bmatrix} NaN & NaN & NaN & & NaN \\ NaN & NaN & 6 & \cdots & NaN \\ NaN & 0 & NaN & & NaN \end{bmatrix} NaN$$

So if I perform rewiring, i.e. I change the parent node for a certain waypoint (already explored), I simply need to change a value inside the P matrix (no for loops needed!)

And once the planning ends (either because all the nodes have been explored or because the max allowable computational time has elapsed) then I easily get the optimal path:



→ the corresponding P matrix is:

$$P = \begin{bmatrix} \text{NaN} & 2 & \text{NaN} & 4 & \text{NaN} & \text{NaN} & 14 & \text{NaN} \\ 0 & \text{NaN} & 2 & \text{NaN} & 10 & \text{NaN} & \text{NaN} & 21 \\ \text{NaN} & 2 & \text{NaN} & 6 & \text{NaN} & \text{NaN} & 12 & \text{NaN} \end{bmatrix}$$

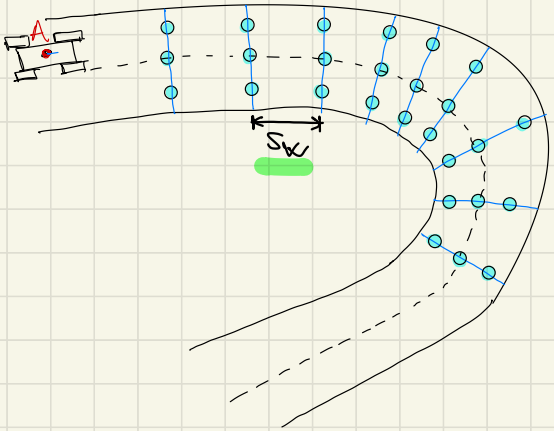
↓ the matrix contains NaNs if some waypoints have not been explored (due to the computational time limit)

to reconstruct the optimal path from the final P matrix, start from the goal point 25, whose parent is

21, and the parent of 21 is 12, and the parent of 12 of 6, whose parent is 2, whose parent is 0 => the optimal path is 0->2->6->12->21

METHODS FOR WAYPOINTS GENERATION

The method here presented applies to both on-line and off-line waypoints generation

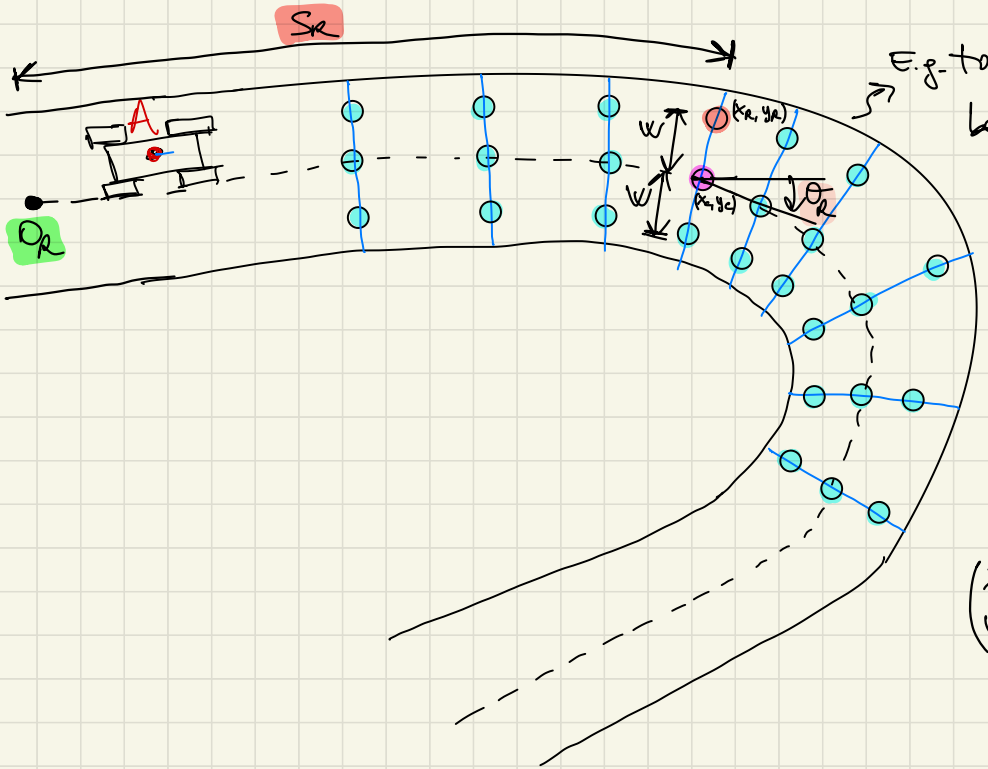


Use the following steps:

1. Define the road center line with a clothoid list
2. Decide a (heuristic) policy for the spacing s_w among waylines. One simple option is to compute s_w as a function of the local road curvature: if the clothoid list defining the road center line is named clothRoad, and a curvilinear abscissa vector is defined and named sVect (e.g. sVect = 0:0.1:clothRoad.length()), the corresponding curvature values can be computed with
   curvVect = clothRoad.kappa(sVect).
   A simple heuristic function for $s_w(k)$, with k being the local curvature, can be $s_w(k) = \min(10 - a \cdot k, 4)$, so that s_w decreases if k increases (the numbers may have to be chosen differently) and "a" is to be tuned

3. Generate the vector of curvilinear abscissae s_{Way} at which the waylines are located, using the chosen function $s_W(k)$

4. Build the waypoints for each wayline. One simple approach is to generate 3 waypoints for each wayline, with the same parameters for ∀ wayline (if there're no obstacles)



E.g. to generate the waypoint in red, which belongs to the wayline whose curvilinear abscissa is s_R (with respect to the road clothoid list, whose origin is in O_R):

the local heading of the road clothoid list in s_R is $\theta_R = clothRoad.theta(s_R)$, then the coords (x_R, y_R) of the red waypoint are:

$$\begin{pmatrix} x_R \\ y_R \end{pmatrix} = \begin{pmatrix} x_c \\ y_c \end{pmatrix} + \begin{bmatrix} \cos\theta_R & -\sin\theta_R \\ \sin\theta_R & \cos\theta_R \end{bmatrix} \begin{pmatrix} 0 \\ w \end{pmatrix},$$

where (x_c, y_c) are the coords of the waypoint on the road centerline belonging to the same waypline as the one in real:

$$[x_c, y_c] = \text{clothRoad.evaluate}(s_R)$$

Note: the methods .theta, .kappa, .evaluate accept as input also a vector of curvilinear abscisse sVect, and not just a single value. So the evaluation of θ_R, x_c, y_c can be done with only 2 lines of code, to compute the corresponding values for all the waypoints:

vectors

$$[x_c, y_c] = \text{clothRoad.evaluate}(sVect)$$

$$\theta_R = \text{clothRoad.theta}(sVect)$$

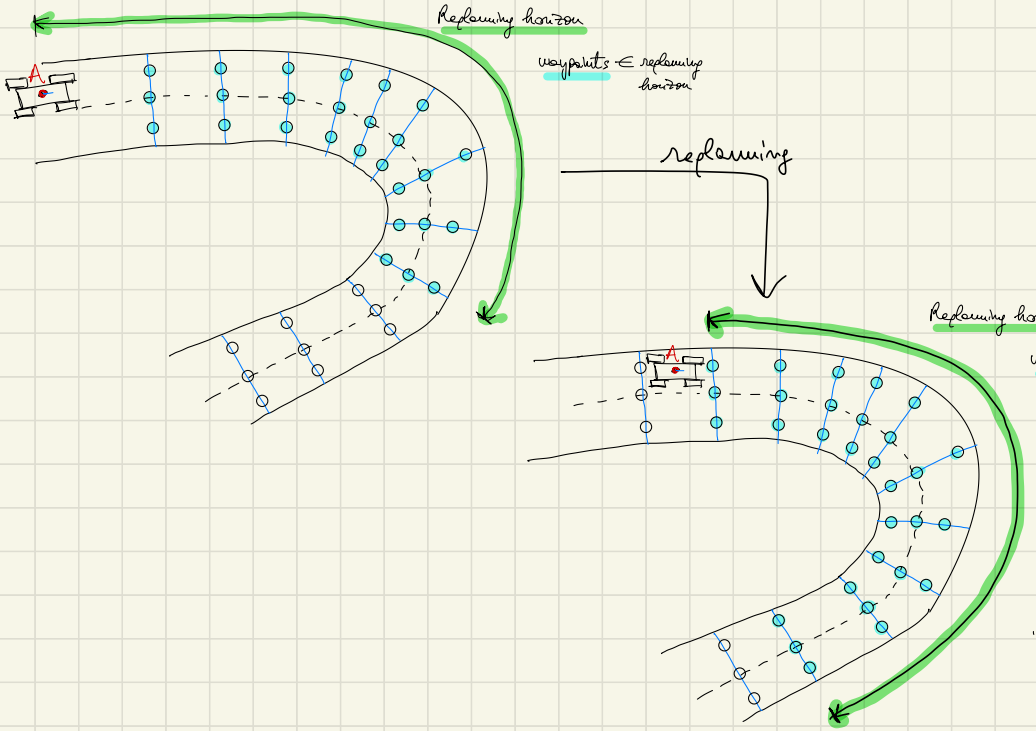
vector

→ NOTE: one easier way to get everything with 1 command is:

$$[x_c, y_c, \theta_R] = \text{clothRoad.evaluate}(sVect)$$

NOTE: Step (1) must be performed off-line only once.

Steps (3) – (4) can be performed off-line or on-line. If they're performed off-line, then on-line it is only necessary to select the portion of waypoints lying inside the chosen prediction horizon:



The off-line generation of waylines and waypoints should decrease the overall cpu time. If there're obstacles detected on-line, waylines and waypoints should be changed around the obstacle (as described in the previous sections).

A comparison of off-line and on-line waypoints generation can be interesting.

SAVING COMPUTATIONAL TIME

CLOTHOID LIBRARY

→ the object initializations "cost" a lot computationally

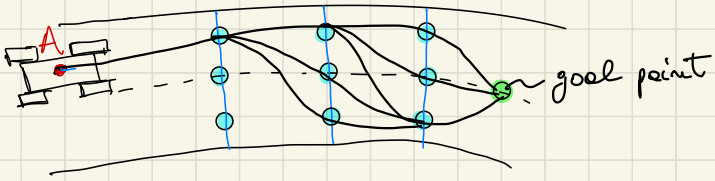
↳ initialize the clothoidCurve only once, at the beginning of the simulation, and then redefine the clothoid with the build_G1 command, without re-initializing it, every time a new clothoid is needed (for parent node selection / rewiring)

↓

see the matlab file save_cpu_time.m for more details

COMPARING WITH NAIVE SOLUTIONS

- One possible naive solution is to compute all the possible paths leading to the goal, calculate the minimum time solution for each of them, and finally select the path with the lowest cost



→ the total n° of possible solutions is $n_p^{n_e}$, with n_p = n° of waypoint
n_e = n° of waylines

↓

e.g. in this case $n_p = 3$, $n_e = 3$

↓

total n° of possible paths is $3^3 = 27$