

COMPTE RENDU SEMAINE DU 02/01/2026:

fonction `_calcul_angle(self,obs):`

Notre fonction permet de calculer l'angle entre la trajectoire du nœud (milieu de segment avec `path_start` et `path_end`) vers le kart et la trajectoire du kart.

```
def choose_action(self, obs):
    angle = self._calcul_angle(obs)
    print(f"Angle actuel: {angle:.3f} rad, {np.degrees(angle):.1f}°")

    acceleration = 0
    steering = -1.0
    action = {
        "acceleration": acceleration,
        "steer": False, #np.clip(angle, -0.5, 0.5),
        "brake": False, #abs(angle) > 1.0, # bool(random.getrandbits(1)),
        "drift": False,
        "nitro": False,
        "rescue": False,
        "fire": False,
    }
    return action
```

Pour afficher les valeurs de l'angle a chaque frame (une superposition d'images affichées à l'écran). Ainsi à chaque frame, on obtient un ensemble d'angle qui précise dans quel coté nous sommes , par exemple si nous sommes à gauche l'angle en radian sera ainsi négatif et dans le cas où le kart est à droite , les angles en radian seront positives.

Algorithme suivi:

[#pour avoir le nb de noeud on fait `len(obs.path_start)`]

#vérifier si on est au début et qu'il y a bien y a des noeuds disponibles (si env existe)(obs ou `self.env.track`)

 #si vérifié alors prend des vecteurs de positions depuis obs voir en bas (obs ou `self.env.world.karts[idx]`)

`#obs = ("center_path", "path_nodes", "paths_start", "paths_end")`

`#chercher indice sécurisé (position d' un nœud qui existe vrm !)`

`#quand trouvé commencer le calcul d'angle calcule $dx = node.x - kart.x$ et $dz = node.z - kart.z$, puis $angle = atan2(dx, dz)$.`

`#node = nodes[idx][:3]`

`#dx = node[0] - pos[0]`

`#dz = node[2] - pos[2]`

`#ang_rad = np.arctan2(dx, dz) # convention : 0 = devant (z), >0 = à droite`

 #a chaque noeud atteint on reset le tableau des obs (pour ne pas avoir les mêmes observations des noeuds de la position précédente)

[#boucle possible , reset pour optimisée]

Dans un premier temps nous avons suivi, cet algorithme cependant nous nous sommes rendus compte qu'il n'y avait pas de nœuds. Ainsi depuis les segments (ce sont des paires de nœuds) de la trajectoire, nous avons donc calculé les coordonnées. La variable "cible" permet de viser le milieu de chaque segment afin que la direction change progressivement et ainsi avoir moins de zigzag, ainsi avoir une conduite du kart plus fluide. Cependant, nous avons remarqué que notre fonction d'angle renvoyait des valeurs aléatoires qui ne suivaient pas réellement la trajectoire de notre kart, nous avons par exemple un angle qui peut aller du positif vers le négatif même si notre kart est à la droite de la piste.

À l'affichage du terminal :

Angle actuel: 0.776 rad, 44.4°

Angle actuel: 1.338 rad, 76.6°

Angle actuel: 1.432 rad, 82.1°

Angle actuel: 1.264 rad, 72.4°

Angle actuel: 1.041 rad, 59.7°

Angle actuel: -2.644 rad, -151.5° ← incohérent avec les observations qu'on attendait

Angle actuel: 0.550 rad, 31.5°

.
. .
. .
. .

Angle actuel: 1.384 rad, 79.3°

On se donne donc comme objectif de régler ce problème pour la semaine suivante. Et nous discuterons des éventuelles améliorations.

var en rad entre 6 et 0 (ou en degré) essayer de faire en sorte que la trajectoire du kart soit au milieu de la piste .

Comment marche le python objet. ←

préciser les commit et les modifications

faire une branche pour merger dans le main → une branche dev

Améliorations possibles:

- L'attribut "lookahead" pour choisir un nombre de nœuds visible pour être le plus stratégique et rapide en fonction de la situation (obstacle, virage, saut..) rencontrés de l'observation du kart.

Réflexion faite:

- Plus il y a de nœud visible plus la conduite de l'agent sera fluide (mais peut être problème de vitesse -> la vitesse sera minimale) et le calcul de distance (= nb de nœuds) => choix de direction plus rapide, en cas de piste avec plusieurs chemins possible.

début d'une implémentation d'un code qui réagit à un obstacle:

```
#if target_item_distance== 10:
    if target_item_type in bad_type:
        if target_item_angle >5: # obstacle a droite
            action["steer"]= - 0.5 # on tourne a gauche
            action["nitro"]=False
            action["acceleration"]= action["acceleration"] - 0.5 #val a
determiner

        elif target_item_angle ==0 : # obstacle a droite
            action["steer"]= - 0.5 # on tourne a gauche ou droite
            action["nitro"]=False
            action["acceleration"]= action["acceleration"] - 0.5 #val a
determiner

        elif target_item_angle <=-5: # obstacle a gauche
            action["steer"]= 0.5 # on tourne a droite
            action["nitro"]=False
            action["acceleration"]= action["acceleration"] - 0.5 #val a
determiner

    elif target_item_type in good_type:
        if target_item_angle <=-5: # a gauche
            action["steer"]= - 0.5 # on tourne a gauche
            action["nitro"]=True
            action["acceleration"]= action["acceleration"] + 0.5 #val a
determiner

        elif target_item_angle >5: #a droite
            action["steer"]= 0.5 # on tourne a droite
            action["nitro"]=True
            action["acceleration"]= action["acceleration"] + 0.5 #val a
determiner

(ajout d'un code qui reagis aux obstacles dans choose_action)
return action
```

Raisonnement :

La fonction choose action reçoit des observations (en provenance de l'environnement) et adapte sa réaction pour réagir aux obstacles de manière réfléchie.

Comment sont classés les obstacles et quels sont-ils dans ce jeu?

Les obstacles (ou item) sont au nombre de 10 et sont classés dans des listes selon s'ils sont bénéfiques à l'agent ou non. En effet, dans le fichier ItemObservationWrappers, les bons items sont rangés dans la liste good_types[] (ex: bonus_nitro) et les mauvais dans bad_types[] (ex: banane).

```
good_types = [0, 2, 3, 6]
bad_types = [1, 4]
# Assume a maximum of 10 different item types.
```

Quand est- ce qu'on prend en compte un item ?

Lorsqu'il est à une distance <10.

```
valid_mask = ahead_mask & (distances < 10) & (np.abs(angles) < 30)
```

Comment faire face à un bon et un mauvais temps?

Il faut éviter les mauvais et aller dans la direction des bons.

Nous avons à disposition des keys : target_item_position, target_item_distance, target_item_angle, target_item_type. Celles-ci sont utiles dans l'implémentation de la réaction de l'agent. Cependant, lorsqu'on lance la partie avec les modifications ci-dessus, aucun changement n'est constaté sur l'agent. Ainsi on peut supposer que le problème peut provenir d'une mauvaise utilisation des keys.

Les valeurs utilisées dans les if seront a adapté lorsque les keys auront été correctement utilisés. Ainsi nous pourrons tester différentes valeurs en fonction de nos observations du comportement de l'agent durant les courses.

Questions encore en suspens:

Est- ce qu'il faut ralentir et tourner face à un mauvais obstacle ou seulement tourner?

Comment modifier la vitesse du kart? avec la variable velocity? la clé "accélération"?

Comment utiliser Kart_obs_space dans l'implémentation de choose_action()? Est-ce nécessaire?

Leon:

1. Lorsqu'on vise le path qui se situe trop près de soi, elle provoque une **oscillation** de kart.
2. **Norme de vitesse** : Calcul de la scalaire du vecteur vitesse $v = (v_x, v_y, v_z)$. Il s'agit de la « vitesse absolue » indépendamment de la direction.
 - 2.1. `speed = np.linalg.norm(velocity)`
3. **obs["paths_start"]** : Coordonnées relatives des nœuds basées sur la position et l'orientation actuelles du kart. Ce n'est pas un système de coordonnées mondiales absolues, mais un système de coordonnées locales où le kart est toujours au centre (0, 0, 0) et l'avant correspond à l'axe z positif.
4. Type de donnée **velocity** : Un tableau de nombres Float de taille 3.
5. **stuck_steps** : Un dispositif qui surveille si le kart est bloqué et déclenche une recule.
6. **np.array** : La conversion en np.array permet d'effectuer des opérations mathématiques complexes (addition de vecteurs, soustraction, produit scalaire, calcul de norme).
7. **Phase**
 - 7.1. `READY_PHASE`
 - 7.2. `SET_PHASE`
 - 7.3. `GO_PHASE` : Juste après l'affichage de "Go" et le début de la course.
 - 7.4. `RACE_PHASE` : Course normale en cours.
8. **paths_start** : Liste des nœuds situés au centre de la piste.
9. `shuffle` : Les voitures sont fixées sur la piste, et **np.random.shuffle(agents)** simule un tirage au sort juste avant la course pour déterminer quel pilote monte dans quelle voiture.
10. **np.clip** : Limite les valeurs pour qu'elles ne dépassent pas l'intervalle $[-1, 1]$.
11. Si l'on continue de freiner alors que le véhicule est à l'arrêt, alors elle passe en marche arrière.

de ayse et mariam

```
import numpy as np
import random
```

```
from utils.track_utils import compute_curvature, compute_slope
from agents.kart_agent import KartAgent
```

message.txt

5 Ko

Mariam a commencé un appel qui a duré 13 minutes. — 02/02/2026 20:19

a_ksu a commencé un appel qui a duré 2 heures. — 02/02/2026 20:31

```
import numpy as np
import random
```

```
from utils.track_utils import compute_curvature, compute_slope
from agents.kart_agent import KartAgent
```

```
class Agent2(KartAgent):
```

```
    def __init__(self, env, path_lookahead=3):
        super().__init__(env)
        self.path_lookahead = path_lookahead
        self.agent_positions = []
        self.obs = None
        self.isEnd = False
        self.name = "Team2" # replace with your chosen name
```

```
    def reset(self):
        self.obs, _ = self.env.reset()
        self.agent_positions = []
```

```
    def endOfTrack(self):
        return self.isEnd
```

```
#implementation du code calcul d'angle
```

```
def _calcul_angle(self, obs):
    if len(obs) < 1 : #permet de verifier si obs existe
        return 0.0
    # il est mieux de retourner une valeur telle que 0.0 pour renvoyer un
    comportement stable, donc pas de plantage , donc parfait pour un controleur
    # le fait de return None c'est possible moins optimal -> ça oblige à tester une
    condition supplementaire telle que if angle is None :
    # De plus il peut casser les calculs -> ex : np.arctan2
    # pour avoir le nombre de noeuds on peut faire len(obs.get["path_start"])
    elif len(obs["paths_start"]) < 1:
        return 0.0
    elif len(obs["paths_end"]) < 1: # paths_end renvoie un segment de fin
        return 0.0
    elif len(obs["karts_position"]) < 1 :
        return 0.0
    elif len(obs["velocity"]) < 3 :
        return 0.0
    else :
```

```

paths_start = obs["paths_start"] # x y z
paths_end = obs["paths_end"] # x2 y2 z2
karts_position = obs["karts_position"]
#nodes = obs["path_nodes"] n existe oas

```

```

idx = min(self.path_lookahead, len(paths_start) - 1)

```

```

seg_start = np.array(paths_start[idx])
seg_end = np.array(paths_end[idx])
pos = np.array(karts_position[0])

```

```

cible = (seg_start+seg_end)/2
vecteur_traj = cible - pos

```

```

vecteur_vit = np.array(obs["velocity"][0:3]) #[0:3] -> on prend les
coordonnées x y z

```

```

produit_vect = vecteur_traj[0]*vecteur_vit[2] - vecteur_traj[2]*vecteur_vit[0]
produit_scal = vecteur_traj[0]*vecteur_vit[0] + vecteur_traj[2]*vecteur_vit[2]
angle = np.arctan2(produit_vect,produit_scal)
return angle

```

TME SOLOOOOOO !!!!!!!!!!!

tme solo : - manipuler git → clone , savoir le code de notre équipe , creer un petit document (avec petit dessin (savoir utiliser des outils numeriques) , description de ce qu'on a fait) , soit ordinateur de la ppti ou pc portable