mardi 27 janvier 2026 22:05

Lecture fichier de code env dans src, début 22h10

Fonction get_observation:

1. Partie I

```
assert self.world is not None, "World is None"

    kart: pystk2.Kart = self.world.karts[kart_ix]

    def kartview(x):
        """Returns a vector in the kart frame

        X right, Y up, Z forwards
        """
        return rotate(x - kart.location, kart.rotation)
```

Cette partie du code permet de modifier les positions de telle sorte que au départ ce sont des positions relatives au plan, et après modifiation ce sont des positions relatives au kart.

Crée un nouveau repère (x,y,z) tel que: x est droite/gauche, y en haut/en bas, et z devant/derrière, coordonnées relatives au kart

kart = self.world.karts[kart_ix] , permet de récuperer les infos sur le kart ix
Noter que self.world donne les infos sur le jeu en entier, et self.word.kart est la "liste" de tous les karts, chacun contenant ses propres informations (par ex la vitesse, son vecteur d'emplacement etc..)

La fonction kartview prend en paramètre x qui est une position dans le plan (qui peut correspondre à un objet ou à un kart adversaire) et renvoie le vecteur reliant le kart courant à cet position

x-kart.location : kart.location -> coordonnées du kart donc faire x - kart.location donne les coordonnées qui sépare les deux points, un peu comme faire yb - ya; xb - xa pour calculer un vecteur comme on l'a appris de base

Rotate(x-kart.location, kart.rotation) permet d'incurver le vecteur selon la rotation du kart, parce que le vecteur est exprimé dans le repère monde et pas dans le repère du kart

2. Partie II

```
def list_permute(list, sort_ix):
    list[:] = (list[ix] for ix in sort_ix)
```

```
def sort_closest(positions, *lists):
   # z axis is front
   distances = [np.linalg.norm(p) * np.sign(p[2]) for p in positions]

   # Change distances: if d < 0, d <- -d+max_d+1
   # so that negative distances are after positives ones
   max_d = max(distances)
   distances_2 = [d if d >= 0 else -d + max_d + 1 for d in distances]

   sorted_ix = np.argsort(distances_2)

   list_permute(positions, sorted_ix)
   for list in lists:
      list_permute(list, sorted_ix)



 def list_permute(list, sort_ix):
   list[:] = (list[ix] for ix in sort_ix)
```

Cette fonction change les positions des éléments d'une liste ,
List[:] dit remplace list par ce qui est après le =
Sort_ix est une liste d'indice qui represente l'ordre des éléments que je souhaite après la permutation
Imaginons : list=[1, 2, 3] et sort_ix = [2, 0, 1] alors on aura list=[3, 1, 2]


```
def sort_closest(positions, *lists):
   # z axis is front
   distances = [np.linalg.norm(p) * np.sign(p[2]) for p in positions]

   # Change distances: if d < 0, d <- -d+max_d+1
   # so that negative distances are after positives ones
   max_d = max(distances)
   distances_2 = [d if d >= 0 else -d + max_d + 1 for d in distances]

   sorted_ix = np.argsort(distances_2)

   list_permute(positions, sorted_ix)
   for list in lists:
      list_permute(list, sorted_ix)
```


Sort_closest permet de trier les différents objets observés du plus proche du kart au plus loin
Position est une liste de positions x, y, z correspondant au items dans la liste lists , ex : on a trois vecteurs x, y, z qui correspondent au kart 3 devant moi, à une banane et à un nitro, si je change l'ordre des positions des vecteurs dans la liste de sorte à avoir du plus proche au plus loin, il faut aussi réorganiser de la même manière la liste d'items, afin que chaque coordonnées correspond respectivement à son item.

distances = [np.linalg.norm(p) * np.sign(p[2]) for p in positions] ; découpons :

np.linalg.norm(p) donne la longueur du vecteur p , distance entre l'objet et le kart

np.sign(p[2]) donne le signe de la coordonnée z du vecteur p , 1 si positif, -1 si negatif, 0 si nul, p = [x, y, z]

Ainsi pour chaque vecteur position p, distances = [np.linalg.norm(p) * np.sign(p[2]) for p in positions] donne la norme du vecteur et indique si l'objet de position p se trouve devant (distance positive) ou s'il est derrière (distance négative)

Distances[] est donc une liste de norme de vecteurs positives ou négatives correspondant aux objets autour de nous

(pause de 15 min pour men souvenir pour le temps)

distances_2 = [d if d >= 0 else -d + max_d + 1 for d in distances] :

Cette ligne modifie les distances telles que : les objets devant (norme positive) soient petits, et les objets derrières (norme négative) deviennent très grand, pour que lorsque l'on trie les distances par ordre croissant, on est d'abord les objets devant le kart, puis les objets derrière le kart

Pour les valeurs négatives, on prend -d (qui sera donc positif) et on l'ajoute à la valeur max de la liste distances (qui est positive) et 1 de sorte à ce qu'il soit plus grand que toutes les autres valeurs de la liste. Si d est positif, on n'y touche pas

sorted_ix = np.argsort(distances_2) :

Np.argsort() trie dans l'ordre croissant les éléments de la liste distances_2, on a donc en premier élément la position de l'objet le plus proche du kart dans la liste sorted_ix

list_permute(positions, sorted_ix) : réorganise la liste de positions selon l'ordre sorted_ix

```
for list in lists:
  list_permute(list, sorted_ix)
```

Cette boucle permet de réorganiser toutes les listes qu'on met dans lists de sorte à trier les positions, les noms, les types etc..

3. Partie III

```
# Sort items and karts by decreasing
    karts_position = [
       kartview(other_kart.location)
       for ix, other_kart in enumerate(self.world.karts)
       if ix != kart_ix
    ]
    sort_closest(karts_position)

    items_position = [kartview(item.location) for item in self.world.items]
    items_type = [item.type.value for item in self.world.items]
    sort_closest(items_position, items_type)
```

```
# Distance from center of track
start, end = kartview(self.track.path_nodes[path_ix][0]), kartview(
   self.track.path_nodes[path_ix][1]
)

s_e = start - end
x_orth = np.dot(s_e, start) * s_e / np.linalg.norm(s_e) ** 2 - start

center_path_distance = np.linalg.norm(x_orth) * np.sign(x_orth[0])
```

L'objectif de ce bloc est de calculer à quelle distance est le kart du centre de la piste et de quel côté, gauche ou droite ?
A la fin, on obtient center_path_distance qui :
s'il est positif -> kart a droite du centre
s'il est négatif -> kart a gauche du centre
Proche de zero -> centré sur la piste

```
karts_position = [
   kartview(other_kart.location)
   for ix, other_kart in enumerate(self.world.karts)
   if ix != kart_ix
]
```

Karts_position : on crée une liste de position ,
Self.word.kart : est la liste de tous les karts participants à la course ex [team1, team2 etc..]

Enumerate(self.world.kart) ajoute l'index de ce kart dans la liste ex : (0, team1), (1, team2) etc..

Et donc ix->index , other_kart -> nom du kart
If ix != kart_ix rappel ! Kart_ix est l'index de mon kart, donc on regarde tous les kart et on les numérote sauf celui sur lequel on travaille

Au final la liste kart_position est composée de kartview(other_kart.location) qui je rappelle transforme les coordonnées d'un point de vue "map" à un point de vue depuis le kart, other_kart.location est telle que c'est appelé, juste les coordronnées des autres karts :)

sort_closest(karts_position) trie les positions des karts du plus proche devant au plus loin derrière

```
items_position = [kartview(item.location) for item in self.world.items]
items_type = [item.type.value for item in self.world.items]
sort_closest(items_position, items_type)
```

items_position = [kartview(item.location) for item in self.world.items] :
Récupère la position de chaque item, l'exprime en coordonnée kart

items_type = [item.type.value for item in self.world.items] :
Récupère le type de l'item pour chaque item , à noter que le type d'item est un int d'après pystk2.Item, ex 1 nitro ,2 chewing etc ..
sort_closest(items_position, items_type) :

On trie les positions et le type d'item de la même manière, c'est-à-dire du plus proche au plus loin ne gardant bien le lien quel item? -> quelle position ?

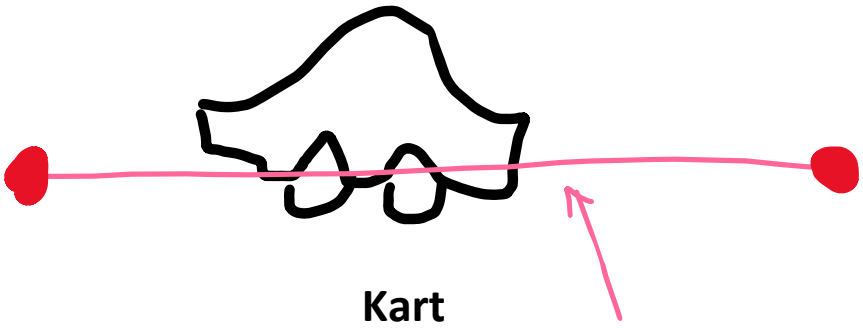
self.track.path_nodes contient l'ensemble des "segments" formés par les nodes du chemin la piste ressemble à quelque chose comme ça : (noter que un node est une sorte de "segment" comprenant un point de départ start et un point de fin de segment end)

. _ . _ . _ . Où chaque point est un point et chaque trait est un segment,
On fait donc s_e = start - end où start est le premier point du segment et end le deuxième :

Start ---------- end

Et donc self.track.path_node contient une liste de tous les segments qui sont notés :
[
[point0, point1], segment 0
[point1, point2], segment 1
[point2, point3], segment 2
etc..
]

path_ix = kart.node ; path_ix contient alors le nœud sur lequel se trouve le kart ie le segment de track sur lequel on est , et donc faire self.track.path_node[path_ix][1] donne alors le point end du segment sur lequel se trouve le kart, qui correspond au point le plus proche de notre kart
schéma :

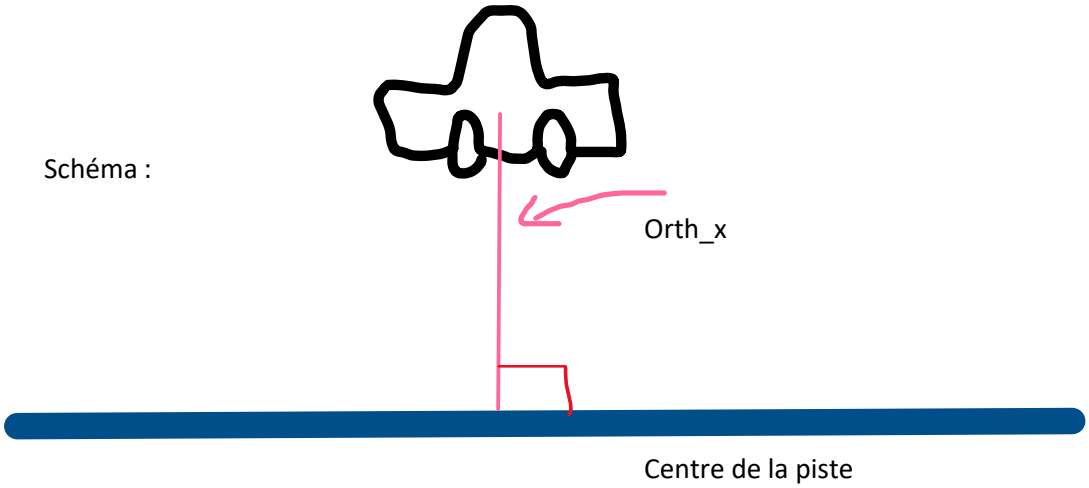


start, end = kartview(self.track.path_nodes[path_ix][0]), kartview(self.track.path_nodes[path_ix][1])

Rappel : kartview fait la modif coordonnées monde -> coordonnées kart
Donc start -> premier point de self.track.path_node et end -> deuxieme point

x_orth = np.dot(s_e, start) * s_e / np.linalg.norm(s_e) ** 2 - start:
x_orth : vecteur qui correspond à la flèche perpendiculaire à la piste qui va de ton kart vers le centre de la piste

Schéma :



center_path_distance = np.linalg.norm(x_orth) * np.sign(x_orth[0]):
Ici center_path_distance est donc la norme de ce vecteur, on multiplie par np.sign(x_orth[o]) car:
x_orth[0] est la coordonnées sur x, qui indique si on est à droite ou à gauche
Np.sign(x_orth[0]) donne donc le signe de cette coordonnées afin que la norme soit négative si on est à droite ou positive si on est à gauche
Noter que si la norme vaut zéro c'est qu'on est au centre de la piste !

Pause 15 min

1. Partie IV

```
# Add action if using AI bot
    # (this corresponds to the action before the observation)
    obs = {}
    if use_ai:
        # Adds actions
        action = self._process.get_kart_action(kart_ix)
        obs = {
            "action": {
                "acceleration": np.array([action.acceleration], dtype=np.float32),
```

```
                "brake": action.brake,
                "drift": action.drift,
                "fire": action.fire,
                "nitro": action.nitro,
                "rescue": action.rescue,
                "steer": np.array([action.steer], dtype=np.float32),
            }
        }


    # --- Sets up the list of nodes to output
    # (1) Set the maximum number of nodes
    if self.max_paths is not None:
        path_size = min(len(self.track.path_distance), self.max_paths)
    else:
        path_size = len(self.track.path_distance)

    # Iterate by using the successor and path
    # distance to break ties

    _self = self
```

Cette partie ajoute l'action actuelle du kart dans le dictionnaire d'observation, en plus des informations sur l'environnement : le bot regarde → décide → agit → le monde change → et regarde à nouveau

Obs = {} crée un dictionnaire vide

If use_ai impose de n'appliquer cette partie de code seulement si c'est un bot qui pilote

action = self._process.get_kart_action(kart_ix):
Demande au moteur du jeu quelles sont les commandes actuellement appliquées à ce kart, il accélère de combien? il a une rotation de combien? Il dérape? Il lance un item? Etc.. Puis les ajoute à obs

```
obs = {
  "action": {
    "acceleration": np.array([action.acceleration], dtype=np.float32),
    "brake": action.brake,
    "drift": action.drift,
    "fire": action.fire,
    "nitro": action.nitro,
    "rescue": action.rescue,
    "steer": np.array([action.steer], dtype=np.float32),
  }
}
```

Cette partie sert à pouvoir "réflechir" comme un bot, dans le sens où les agent ia ne font pas actions au hasard, ils observent et agisse, cette partie doit donc être utilisée si par exemple on voit que notre agent1 perd les courses contre les bots, ce qui veut dire qu'ils ont implementé de meilleurs "réflexes" observation<->action

```
    # --- Sets up the list of nodes to output
    # (1) Set the maximum number of nodes
    if self.max_paths is not None:
        path_size = min(len(self.track.path_distance), self.max_paths)
    else:
        path_size = len(self.track.path_distance)

    # Iterate by using the successor and path
    # distance to break ties

    _self = self
```

if self.max_paths is not None: si on a défini un maximum de nœuds

path_size = min(len(self.track.path_distance), self.max_paths) :
Prend : le nombre total de nœuds sur la piste, le maximum autorisé, et choisi le plus petit des deux , afin de ne jamais demande plus de nœuds que ce qu'il existe

Traduction
Ils prennent :

Le nombre total de nœuds sur la piste

La limite max autorisée

Et ils prennent le plus petit des deux

Donc :

On ne demande jamais plus de nœuds que ce qui existe vraiment

else:
path_size = len(self.track.path_distance)
Ça veut dire :

Si aucune limite, on prend tous les nœuds

1. Partie V

```
class PathComponent:
        def __init__(self, ix):
            self.ix = ix
            # Distance from the kart node start
            start: float = _self.track.path_distance[self.ix, 1]
            self.distance = np.maximum(
                np.abs(start - _self.track.path_distance[path_ix, 1]),
                _self.track.length / 2,
            )

        def __lt__(self, other: "PathComponent"):
            return self.distance < other.distance

    path_indices: list[int] = []
    path_heap: list[int] = [PathComponent(path_ix)]
```

```
    for _ in range(path_size):
        path = heapq.heappop(path_heap)
        path_indices.append(path.ix)

        for succ_ix in self.track.successors[path.ix]:
            heapq.heappush(path_heap, PathComponent(succ_ix))

    return {
        **obs,
        # World properties
        "phase": Phase.from_stk(self.world.phase).value,
        "aux_ticks": np.array([self.world.aux_ticks], dtype=np.float32),
        # Kart properties
        "powerup": kart.powerup.num,
        "attachment": kart.attachment.type.value,
        "attachment_time_left": np.array(
            [kart.attachment.time_left], dtype=np.float32
        ),
        "max_steer_angle": np.array([kart.max_steer_angle], dtype=np.float32),
        "energy": np.array([kart.energy], dtype=np.float32),
        "skeed_factor": np.array([kart.skeed_factor], dtype=np.float32),
        "shield_time": np.array([kart.shield_time], dtype=np.float32),
        "jumping": 1 if kart.jumping else 0,
        # Kart physics (from the kart point view)
        "distance_down_track": np.array(
            [kart.distance_down_track], dtype=np.float32
        ),
        "velocity": kart.velocity_lc,
        "front": kartview(kart.front),
        # path center
        "center_path_distance": np.array([center_path_distance], dtype=np.float32),
        "center_path": np.array(x_orth),
        # Items (kart point of view)
        "items_position": tuple(items_position),
        "items_type": tuple(items_type),
        # Other karts (kart point of view)
        "karts_position": tuple(karts_position),
        # Paths
        "paths_distance": tuple(
            self.track.path_distance[ix] for ix in path_indices
        ),
        "paths_width": tuple(self.track.path_width[ix] for ix in path_indices),
        "paths_start": tuple(
            kartview(self.track.path_nodes[ix][0]) for ix in path_indices
        ),
        "paths_end": tuple(
            kartview(self.track.path_nodes[ix][1]) for ix in path_indices
        ),
    }
```

```
class PathComponent:
```

```
def __init__(self, ix):
    self.ix = ix
    # Distance from the kart node start
    start: float = _self.track.path_distance[self.ix, 1]
    self.distance = np.maximum(
        np.abs(start - _self.track.path_distance[path_ix, 1]),
        _self.track.length / 2,
    )
```

self.track.path_distance[ix, 1] :

Path distance est un tableau qui fait correspondre un index et une distance
On utilise [ix, 1]car c'est dans la colonne 1 qu'est stocké la distance au point de départ, la colonne 0 stocke une autre information sur ix
self.track.path_distance[ix, 1] = la position du nœud ix le long de la piste

self.distance = np.maximum(np.abs(start - _self.track.path_distance[path_ix, 1]), self.track.length / 2) :

Cette partie du code sert à savoir si un nœud est devant ou derrière le kart sur une piste circulaire
Ils prennent la position du kart sur la piste et la position du nœud sur la piste, puis ils calculent la distance qu'il faudrait parcourir en avançant pour atteindre ce nœud
Comme la piste est un cercle, il existe toujours deux chemins entre le kart et un nœud : un chemin court (le nœud est devant) et un chemin long qui fait presque un tour complet (le nœud est derrière), du coup ils coupent mentalement la piste en deux moitiés : tout ce qui peut être atteint en moins d'une moitié de circuit est considéré comme "devant", et tout ce qui nécessite plus d'une moitié de circuit est considéré comme "derrière"
Dans ce cas précis, quand la distance calculée dépasse la moitié de la longueur du circuit, le code donne au nœud une grande valeur de distance. Résultat : les nœuds devant le kart gardent une petite distance et les nœuds derrière le kart passent à la fin du tri.

```
def __lt__(self, other: "PathComponent"):
    return self.distance < other.distance
```

-> Un PathComponent est plus petit qu'un autre si sa distance est plus petite, Parce que heapq a besoin de savoir **comment comparer deux objets** pour décider lequel sortir en premier

path_indices: list[int] = []
On crée une liste vide
Dedans, on va stocker les indices des nœuds de la piste dans l'ordre choisi

path_heap: list[int] = [PathComponent(path_ix)]
On crée une liste qui va servir de tas de tri (heap).
On met dedans un seul élément :

Le nœud où se trouve actuellement le kart (path_ix), à l'interieur d'un PathComponent pour qu'il ait une distance associée

for _ in range(path_size):
On va répéter l'opération path_size fois, sans se preocupper de la variable d'où le _

path = heapq.heappop(path_heap)
heappop :

Regarde tous les éléments dans path_heap, compare leurs distances grâce à __lt__, retire et retourne celui qui a la plus petite distance

path_indices.append(path.ix)

On prend l'index du nœud (path.ix)
Et on l'ajoute dans la liste finale

for succ_ix in self.track.successors[path.ix]:
successors[path.ix] :

La liste des nœuds qui viennent juste après ce nœud sur la piste

Donc on les parcourt un par un

heapq.heappush(path_heap, PathComponent(succ_ix)) :
Pour chaque nœud suivant :

On le transforme en PathComponent
On l'ajoute dans le tas
Le tas se réorganise automatiquement selon la distance

A la fin -> On récupère le nœud le plus proche devant le kart

Cette partie du code sert à sélectionner quels nœuds de la piste le kart va "voir" devant lui et dans quel ordre
On prend le nœud actuel du kart sur la piste et la liste des nœuds suivants reliés à celui-ci
On ajoute ces nœuds dans une structure qui les classe par distance devant le kart, on récupère toujours le nœud le plus proche devant le kart, et on l'ajoute à la liste finale, et ses nœuds suivants dans la structure de tri
On repete ce processus un certain nombre de fois (path_size)
On obtient une liste de nœuds de piste triés du plus proche au plus loin, situés devant le kart, l'agent ne reçoit que la partie de la piste qu'il va rencontrer ensuite

Fin à 2h15