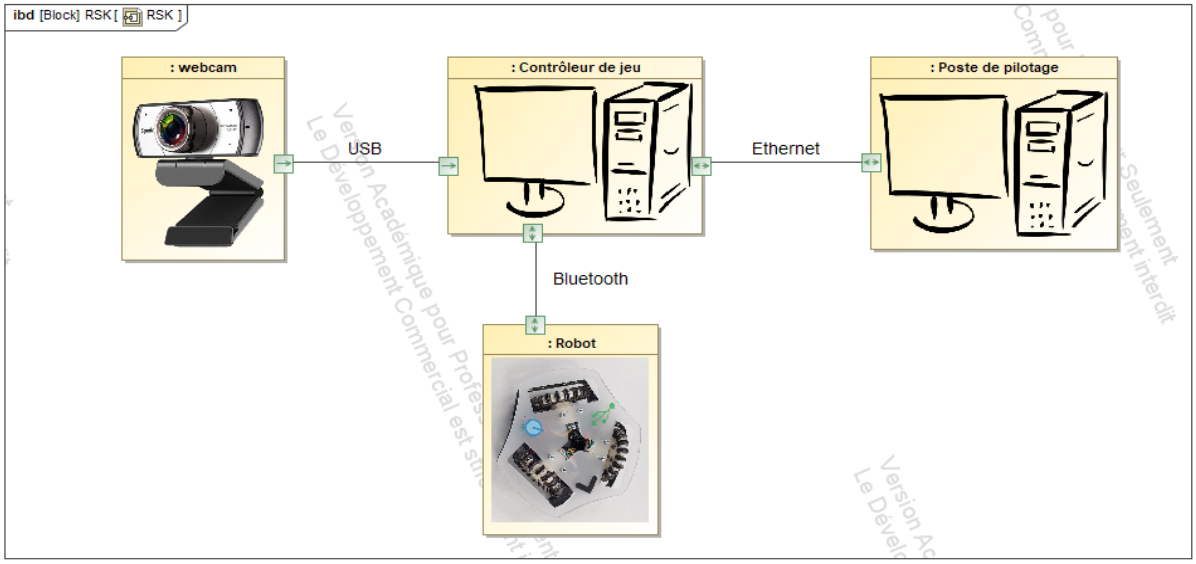
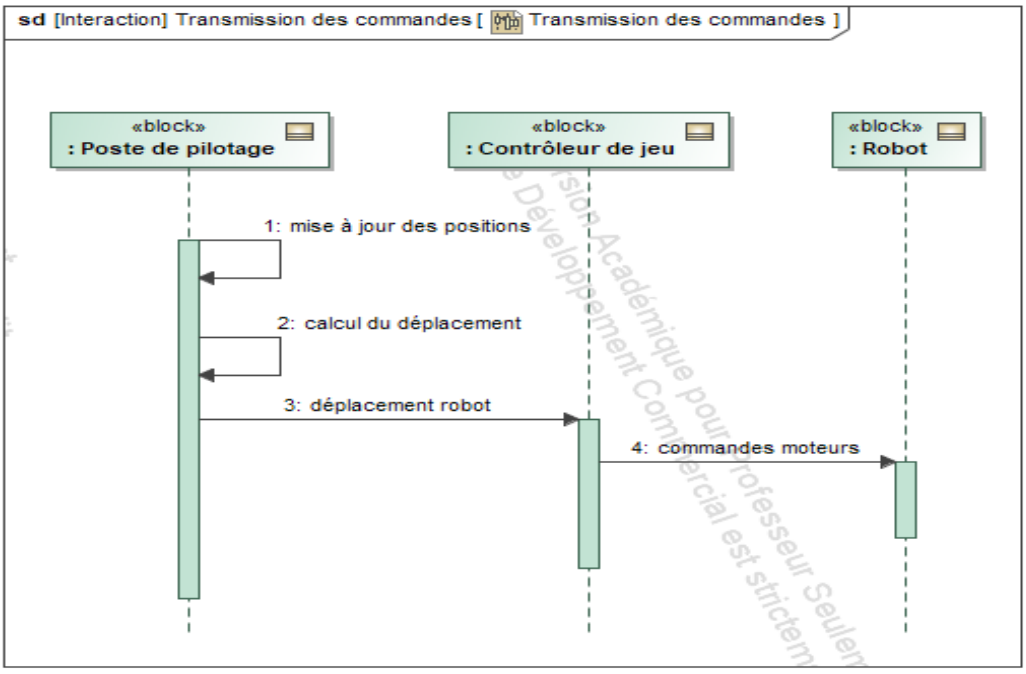
Robot Soccer Kit

Principe de fonctionnement

Dans cette compétition, les robots fonctionnent de façon autonome. Chaque robot est commandé par un programme qui s'exécute sur le poste de pilotage, un même programme pouvant contrôler plusieurs robots.



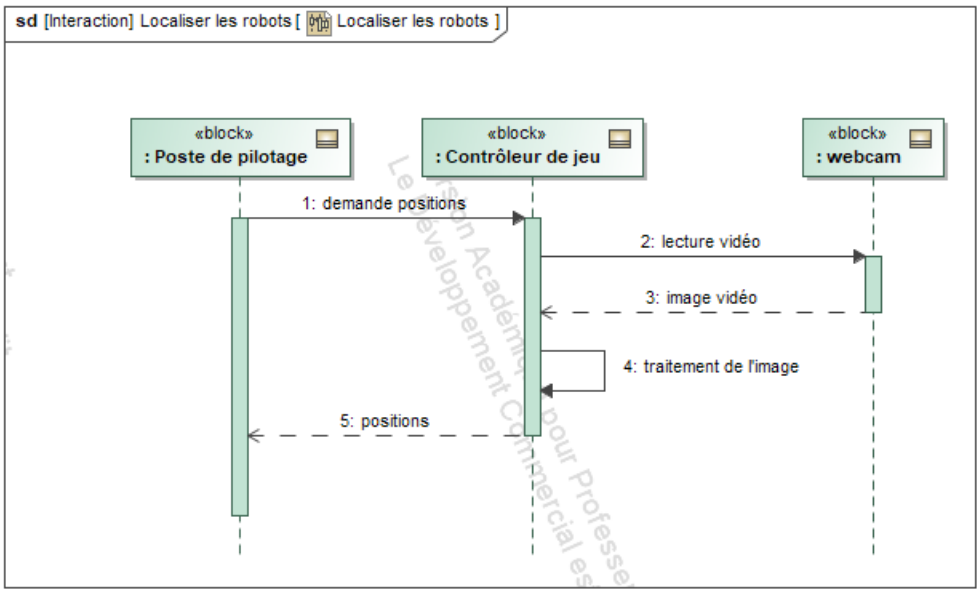
Une application appelée « Contrôleur de jeu » fonctionne comme serveur et reçoit *via* le réseau Ethernet les ordres de déplacement et les retransmet sous forme de commandes moteurs aux différents robots par transmission Bluetooth.



Localisation

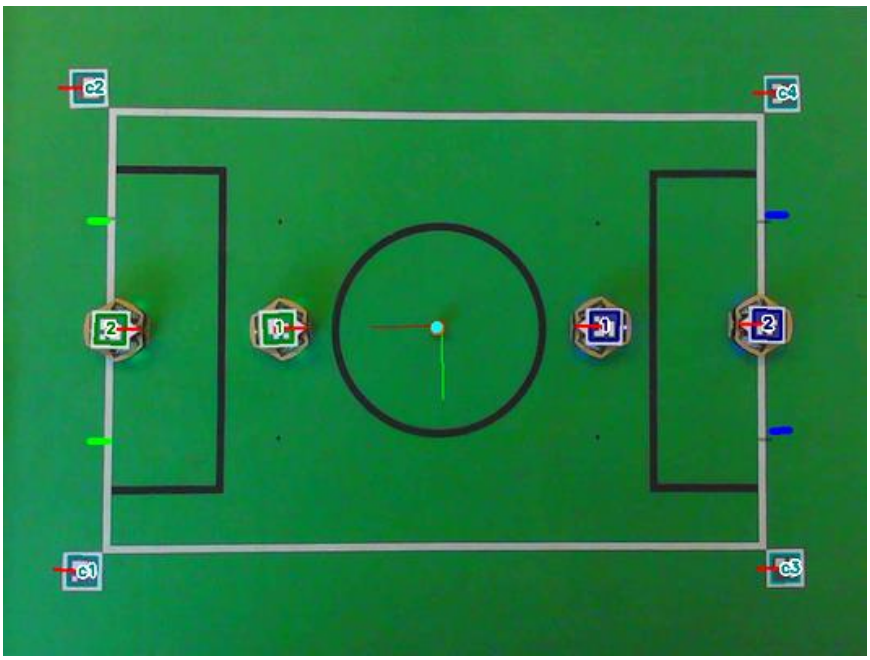
Grâce à la caméra placée au-dessus du terrain, le contrôleur de jeu peut localiser les différents robots sur le terrain, ainsi que la balle.

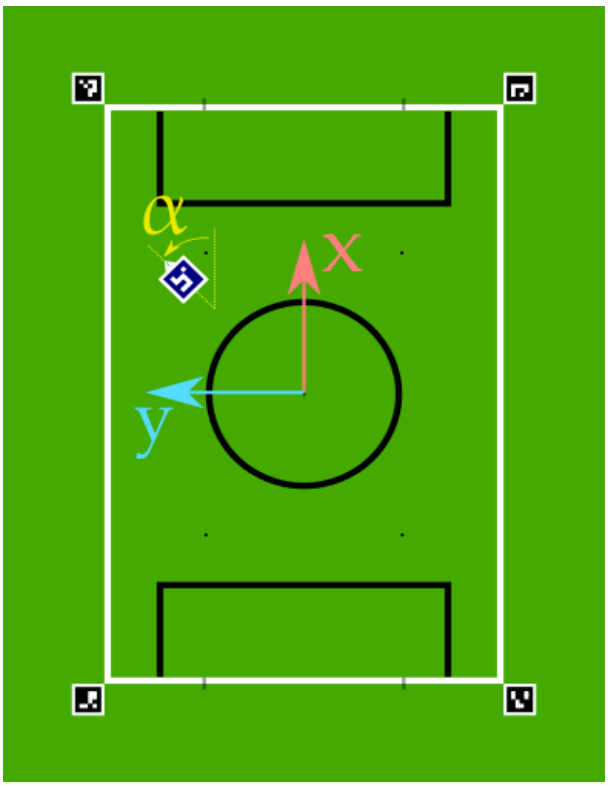
Le contrôleur de jeu reçoit les images de la webcam par USB



La localisation des robots est possible grâce aux étiquettes ArUco (habituellement utilisées pour les applications de réalité augmenté) dessinées sur le dessus du châssis.

De la même façon, des étiquettes permettent de connaître les délimitations du terrain.





La position de la balle est indiquée par un couple (x, y) et celle des robots par un triplet (x, y, α), où :

- x désigne la position (en m) sur l'axe des abscisses ;
- y la position (en m) sur l'axe des ordonnées ;
- α l'angle (en radian) par rapport à l'axe des abscisses.

A noter que l'origine du repère se trouve au centre du terrain.

Programmation

Premier exemple

Voici un premier exemple de la création d'un client à partir de l'API qui commande le premier robot vert de taper dans la balle.

```
import rsk

with rsk.Client() as client:
    robotVert1 = client.robots['green'][1]
    robotVert1.kick()
```

- `import rsk` permet d'importer le module nécessaire
- L'instruction `with` assure que le client s'arrête proprement en fin de connexion. En particulier, cela force l'arrêt des robots à la fin de l'exécution du programme.
- `robotVert1 = client.robots[green][1].kick()` permet de sélectionner le robot que l'on souhaite commander.
- `robotVert1.kick()` demande au robot de taper.

Quand on crée un client, on peut également utiliser les arguments suivants :

```
import rsk

with rsk.Client(host='127.0.0.1', key='') as client:
    robotVert1 = client. robots['green'][1]
    robotVert1.kick()
```

Avec

- `host` est l'adresse IP du contrôleur de jeu (l'adresse 127.0.0.1 est utilisée quand le programme de pilotage s'exécute sur le même machine que le contrôleur de jeu)
- `key` est la clé d'accès (vide par défaut) qui est renseignée dans le contrôleur de jeu et donnée à chaque équipe pour éviter qu'une équipe prenne le contrôle d'un robot adverse.

Informations de localisation

Un processus indépendant est chargé de récupérer de façon continue les données de localisation. Ces données sont ensuite accessibles au travers des attributs suivants :

```
# Robot position (x [m], y [m]):
client.robots['green'][1].position
# Robot orientation (theta [rad]):
client.robots['green'][1].orientation
# Position + orientation (x [m], y [m], theta [rad])
client.robots['green'][1].pose
# Ball's position (x [m], y [m])
client.ball
```

Dimensions du terrain

Les données suivantes sont disponibles :

```
from rsk import field_dimensions

# Field length (x axis)
field_dimensions.length
# Field width (y axis)
field_dimensions.width

# Goal width
field_dimensions.goal_width

# Side of the (green) border we should be able to see around the field
field_dimensions.border_size
```

Contrôler les robots

On peut utiliser les méthodes suivantes pour contrôler les robots au travers de leurs vitesses de déplacement selon les axes des x et y :

```
# Kicks, takes an optional power parameter between 0 and 1
robot.kick()

# Controls the robots in its own frame, arguments are x speed [m/s],
# y speed [m/s] and rotation speed [rad/s]

# Go forward, 0.25 m/s
robot.control(0.25, 0., 0.)

# Rotates 30 deg/s counter-clockwise
robot.control(0., 0., math.radians(30))
```

On peut également contrôler les robots pour qu'ils se déplacent vers des coordonnées de destination sur le terrain :

```
# Sending a robot to x=0.2m, y=0.5m, theta=1.2 rad
robot.goto((0.2, 0.5, 1.2))
```

Le code précédent permet d'envoyer le robot à la position `x=0.2`, `y=0.5` et `theta=1.2` sur le terrain.

On peut également utiliser un second argument (qui est le booléen `wait`) pour que la méthode `goto` ne bloque l'exécution du reste du programme. De cette façon, il est possible de réaliser une autre action en parallèle. Dans l'exemple ci-dessous on déplace deux robots à la fois :

```
# Placing two robots, the loop will block until both robots are placed
arrived = False
while not arrived:
    robot_1_arrived = client.robots[blue][1].goto((0.2, 0.3, 0.), wait=False)
    robot_2_arrived = client.robots[blue][2].goto((0.2, -0.3, 0.), wait=False)
    arrived = robot_1_arrived and robot_2_arrived
```

Quand le second argument `wait` de la méthode `goto` est à la valeur `True`, l'appel de la méthode `goto` bloque l'exécution du programme jusqu'à ce que le robot ait atteint sa destination.

Quand le second argument `wait` de la méthode `goto` est à la valeur `False`, l'appel de la méthode va renvoyer `True` si le robot a atteint sa destination, et `False` sinon.

Programmation avancée

Fonction de rappel

Si on souhaite exécuter une portion de code chaque fois qu'une information est mise à jour, on peut enregistrer une fonction de rappel avec l'attribut `on_update` de l'objet `client`:

```
def print_the_ball(client, dt):
    print(client.ball)

# This will print the ball everytime a new information is obtained from the
client
client.on_update = print_the_ball
```

Cela peut être utile si on souhaite enregistrer les données au fur et à mesure pour un traitement ultérieur, car on s'assure que les données sont enregistrées qu'une fois après chaque mise à jour.

Fonction de rappel

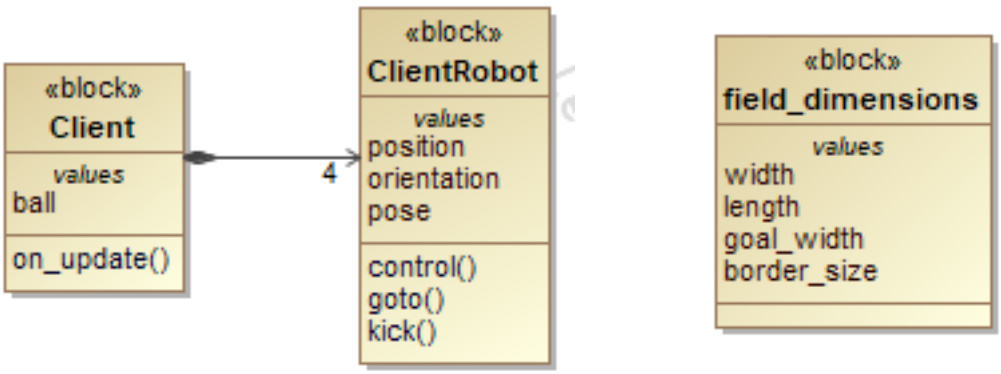
Si une commande échoue, une exception `rsk.ClientError` sera levée et mettra fin à l'exécution du programme. On peut intercepter cette exception de la façon suivante :

```
try :
    robot.kick()
    robot.control(0.1, 0, 0)
except rsk.ClientError:
    print('Error during a command!')
```

Une telle exception peut se produire si le robot que l'on souhaite commander est déjà controlé par le Contrôleur de jeu.

Diagramme de classes

Le schéma ci-dessus résume la composition des classes avec leurs attributs et leurs méthodes :



Fonctions mathématiques (module math)

faire `import math`

Fonctions mathématiques :

- `math.floor(-7.6)` partie entière, donne ici *-8.0*.
- `int(math.floor(4.5))` pour avoir l'entier *4*.
- `math.ceil(-7.6)` entier immédiatement supérieur, donne ici *-7*.
- `math.exp(2)` : exponentielle.
- `math.log(2)` : logarithme en base naturelle.
- `math.log10(2)` : logarithme en base 10.
- `math.log(8, 2)` : log de 8 en base 2.
- `math.sqrt` : racine carrée.
- `math.pow(4, 5)` : 4 puissance 5.
- `math.fmod(4.7, 1.5)` : modulo, ici *0.2*. Préférer cette fonction à `%` pour les flottants.
- `math.factorial(4)` : factorielle 4, donc *24* (uniquement pour les entiers positifs).
- `math.fsum(l)` : fait la somme des éléments d'une liste, à préférer à `sum` car moins d'erreurs d'arrondis (comparer `math.fsum([0.01 for i in range(100)])` et `sum([0.01 for i in range(100)])`)
- `math.isinf(x)` : teste si x est infini (`inf`) et renvoie True si c'est le cas.
- `math.isnan(x)` : teste si x est nan (Not a Number) et renvoie True si c'est le cas.
- `math.gamma` : la fonction gamma (pour n entier, gamma(n) = factorielle(n-1))
- `math.erf` : la fonction d'erreur (intégrale d'une gaussienne).

Fonctions trigonométriques :

- fonctions trigonométriques : `math.sin, math.cos, math.tan, math.asin, math.acos, math.atan` (l'argument est en radians).
- fonctions hyperboliques : `math.sinh, math.cosh, math.tanh, math.asinh, math.acosh, math.tanh`
- `math.degrees(x)` : convertit de radians en degrés (`math.radians(x)` pour l'inverse).
- `math.pi, math.e` : les constantes.