2I013 Groupe 1

Projet Foot

Maxime Sangnier – Nicolas Baskiotis
maxime.sangnier@lip6.fr

2019

Laboratoire d'Informatique de Paris 6 (LIP6)
Sorbonne Université

Outline

Description de l'UE

Description de l'UE

Objectifs du cours

Apprendre :

- à réaliser un projet informatique personnel ;
- quelques outils (interface graphique, motifs de conception) ;
- ce que sont l'apprentissage statistique et l'intelligence artificielle ;
- à préparer un rapport et une soutenance.

Ce n'est pas

- un cours approfondi de Python,
- que du code.

Pré-requis

- des notions d'algorithmique et de structure ;
- les bases de Python ;
- de la motivation !

Déroulement de l'UE

En pratique

- 1h45 de cours le lundi 10h45-12h30 en 23-24.203 (Maxime Sangnier) ;
- 3h30 de TME le lundi 16h-19h45 en 14-15.303 (Elina Thibeau-Sutre & Maxime Sangnier) ;
- supports de cours et code sur **github** :
  `https://github.com/sangnier/Soccer`

Évaluation

- contrôle continu : 70%
  - participation (seul(e), tout le temps)
  - partiel écrit (seul(e), à mi-parcours)
  - rapport, soutenance et code (en binôme, à la fin)
- examen : 30%
  - examen sur machine (seul(e), à la fin).

Introduction au projet

A propos du projet

Objectif

- développer des IAs (plus ou moins intelligentes) de joueurs de football

Code fourni : le simulateur

- les règles du jeu
- la gestion des matchs
- une interface graphique simple

Code demandé : implémentation des joueurs

- pour commencer, des joueurs simples
- puis des joueurs plus intelligents (notions de plan expérimental, d'apprentissage statistique)
- bonus : apprentissage avancé

Championnat

Organisation :

- a partir de la 2 ou 3ème semaine (selon l'avancement), chaque semaine une série de rencontres, tous les groupes rencontrent tous les groupes
- catégories : 1 contre 1, 2 contre 2 (et plus tard 4 contre 4)
- des challenges points d'étapes pour vérifier la qualité de vos joueurs (tirer un but, récupérer la balle, . . .)

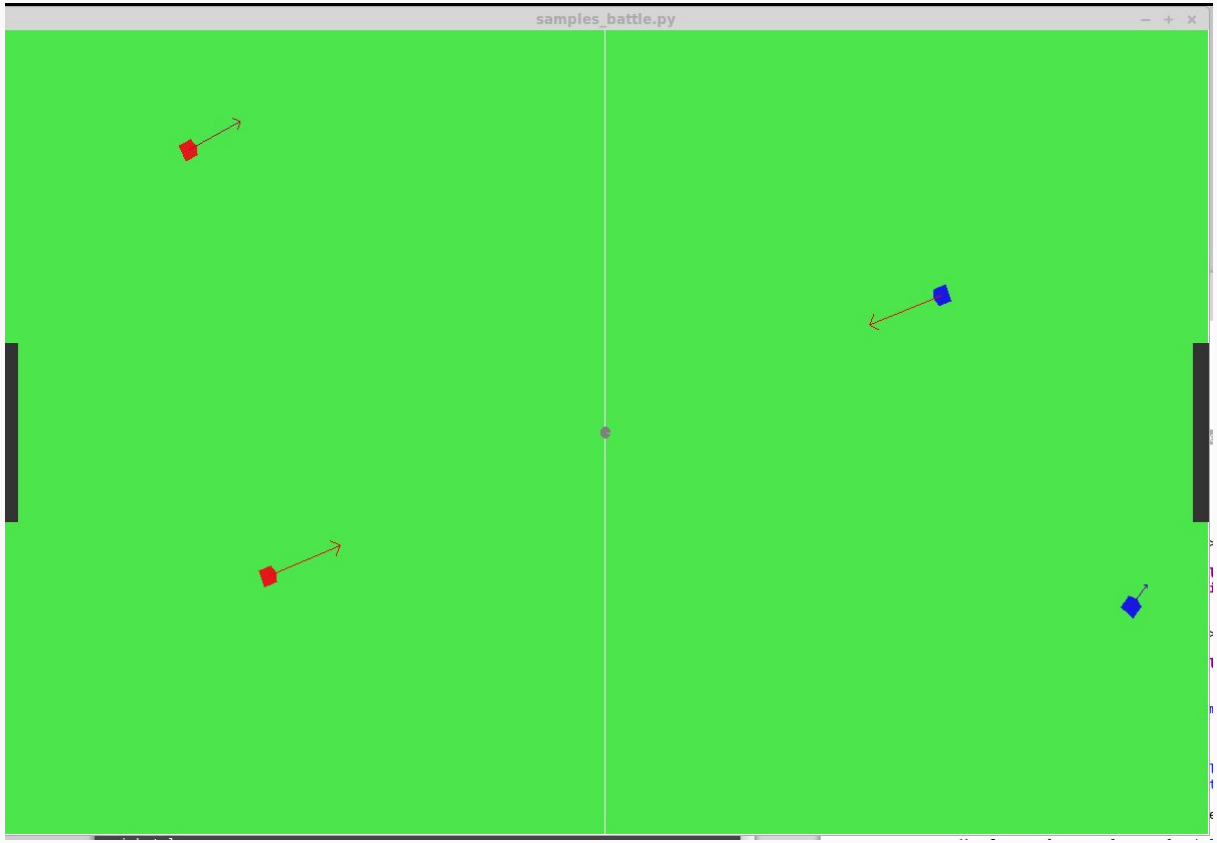
Evaluation du controle continu

- classement dans le championnat, mais il ne suffit pas de gagner !
- prime aux joueurs les mieux pensés, justifiés,
- progression d'une semaine à l'autre,
- participation.

Plateforme de simulation

Besoins

Concept de :

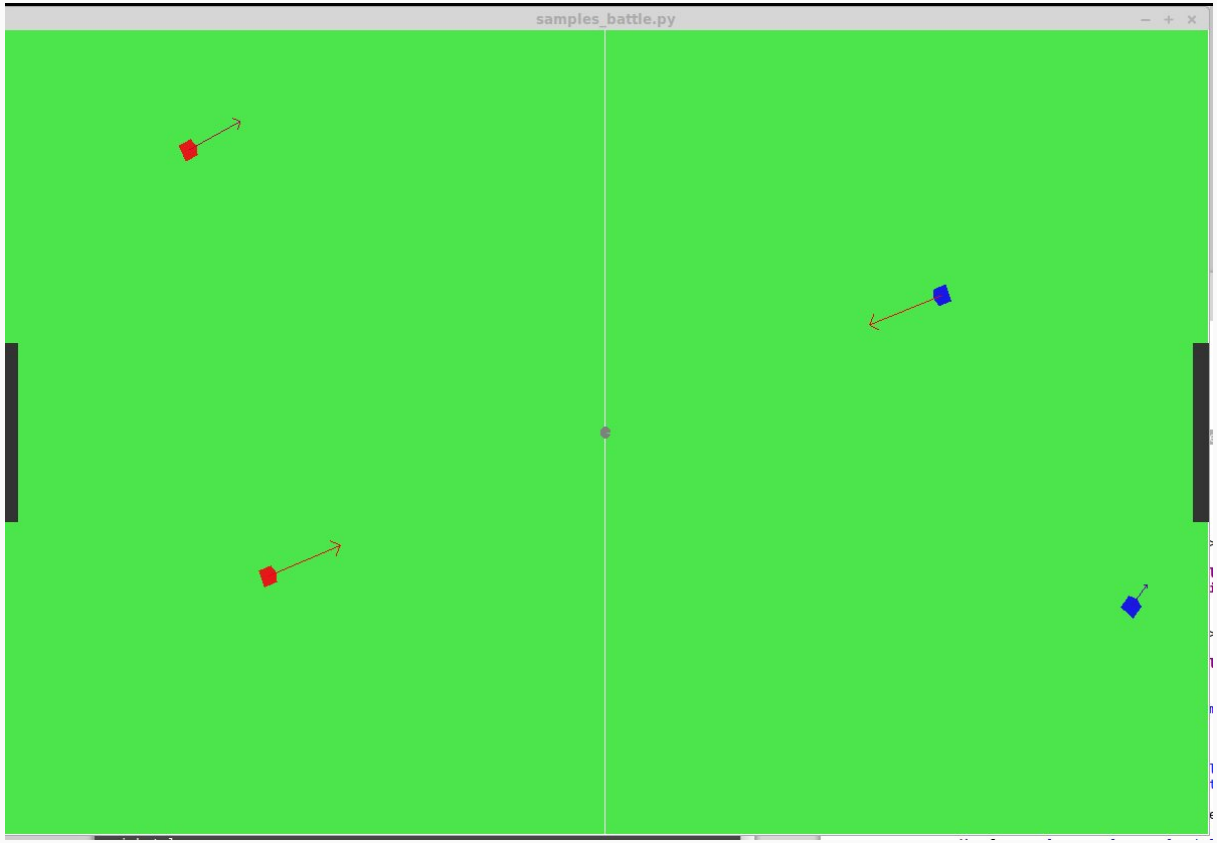


Plateforme de simulation

Besoins

Concept de :

- terrain
- ballon
- joueur
- équipe
- tournoi
- c'est tout ?

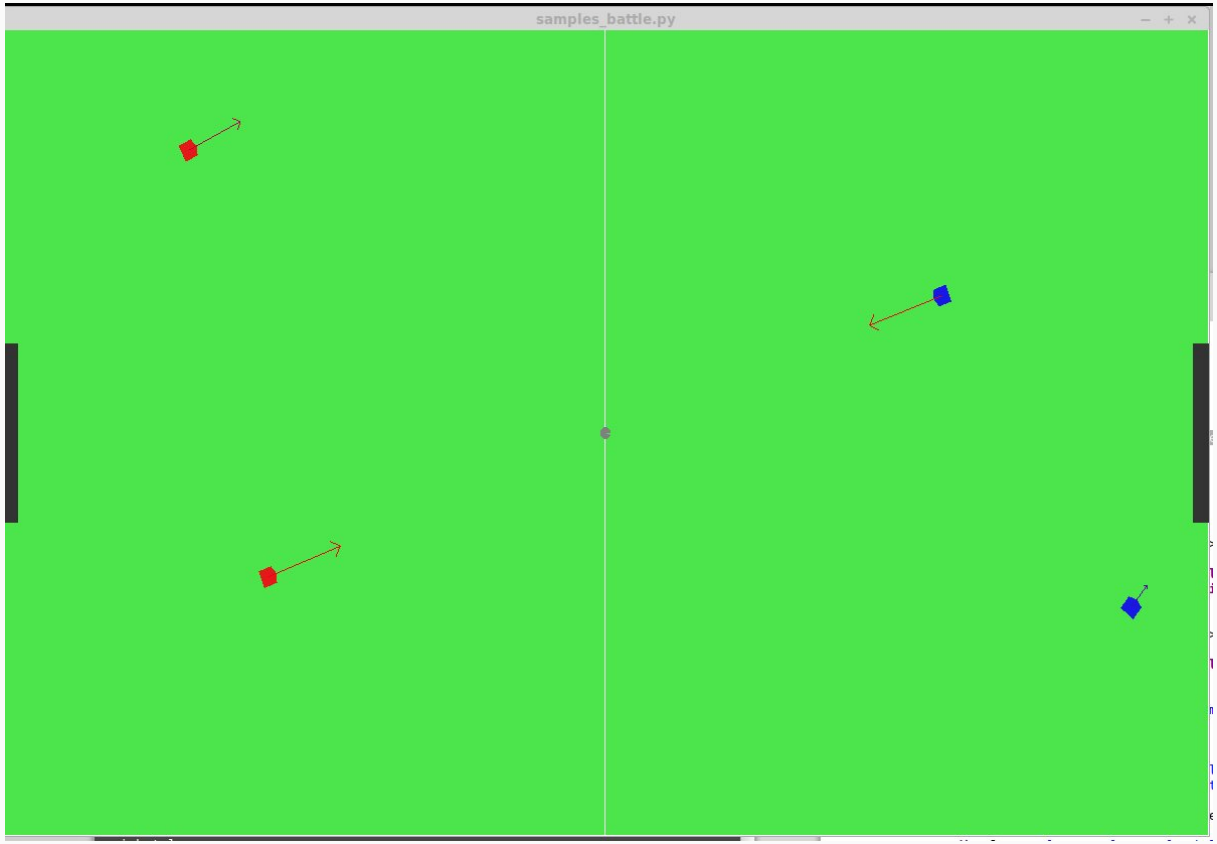


Plateforme de simulation

Besoins

Concept de :

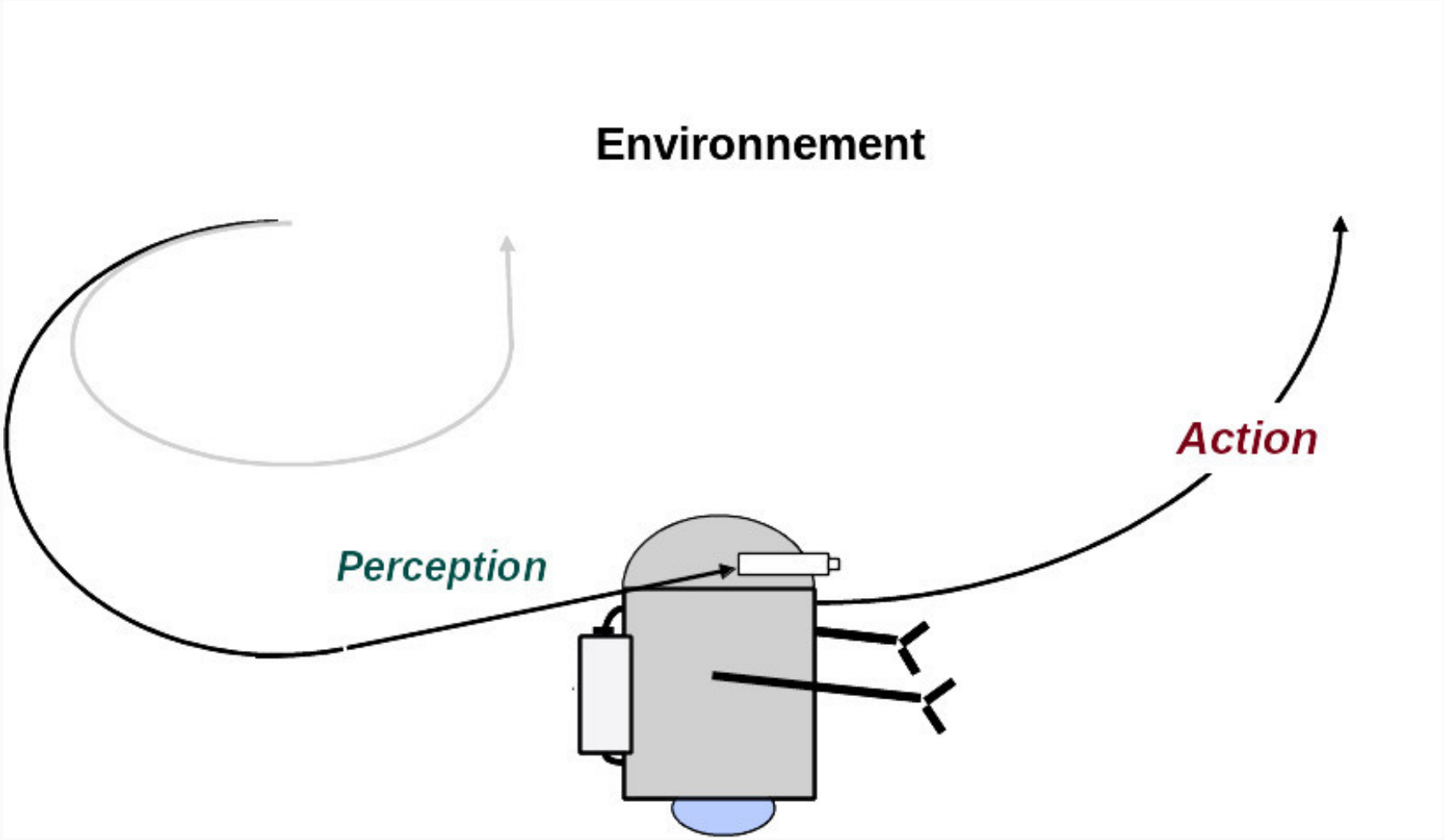
- terrain
- ballon
- joueur
- équipe
- tournoi
- c'est tout ?



Un joueur est une coquille vide !

$\Rightarrow$ il faut lui donner les moyens de réagir

Un joueur = un agent



Modélisation agent

Principe

- Environnement
  - tout ce qui est extérieur à l'agent
- État
  - ce que perçoit l'agent
- Action
  - ce que peut décider l'agent

Exemples

- Jeu d'échecs
- Tetris
- Sudoku
- flappy bird
- ... et le foot.

Modélisation Agent : Foot

Agent = joueur

- Environnement = plateforme de simulation
- État
  - Terrain
  - Position/vitesse de la balle
  - Position/vitesse des joueurs
- Action :
  - Déplacement
  - Tir

Plateforme de la simulation

Plateforme : les objets en présence

- `Vector2D` : x, y et toutes les opérations vectorielles
- `MobileMixin` : base pour tous les objets mobiles, contient `position` et `vitesse`
- `SoccerAction` : contient `acceleration` et `shoot`, deux `Vector2D`
- `PlayerState` : état d'un joueur (vitesse, accélération, position, shoot)
- `SoccerState` :
  - `player_state(self,id_team,id_player)` : renvoie l'état du joueur
  - `ball` : `ball.position, ball.vitesse`
  - `score_team1,score_team2,get_score_team(self,i)`
  - `step` : numéro de l'état
- `Player` : joueur, contient un nom (`name`) et une stratégie (`strategy`)
- `Strategy` : modèle de stratégie, toute stratégie doit implémenter la méthode `compute_strategy(...)`
- `SoccerTeam` : liste des joueurs
- `Simulation` : permet de lancer un bout de match entre deux équipes.
- `SoccerTournament` : tournoi.

Boucle d'action et stratégie

Boucle d'action

- Pour `step` de 0 à `MAX_STEP`
  - calcul pour chaque jouer l'action selon l'état présent : méthode `compute_strategy(self,state,id_team,id_player)`
  - cette méthode doit renvoyer un objet `SoccerAction` correspondant à l'action
  - calcul du prochain état en fonction des actions des joueurs.

Stratégie constante

```
class Strategy:
    def __init__(self, name):
        self.name = name
    def compute_strategy(self, state, id_team, id_player):
        return SoccerAction()
```

Boucle d'action et stratégie

Boucle d'action

- Pour `step` de 0 à `MAX_STEP`
  - calcul pour chaque jouer l'action selon l'état présent : méthode `compute_strategy(self,state,id_team,id_player`
  - cette méthode doit renvoyer un objet `SoccerAction` correspondant à l'action
  - calcul du prochain état en fonction des actions des joueurs.

Stratégie aléatoire

```
class RandomStrategy(Strategy):
 def __init__(self):
     Strategy.__init__(self, "Random")
 def compute_strategy(self, state,id_team, id_player):
     return SoccerAction(Vector2D.create_random(),
                         Vector2D.create_random())
```

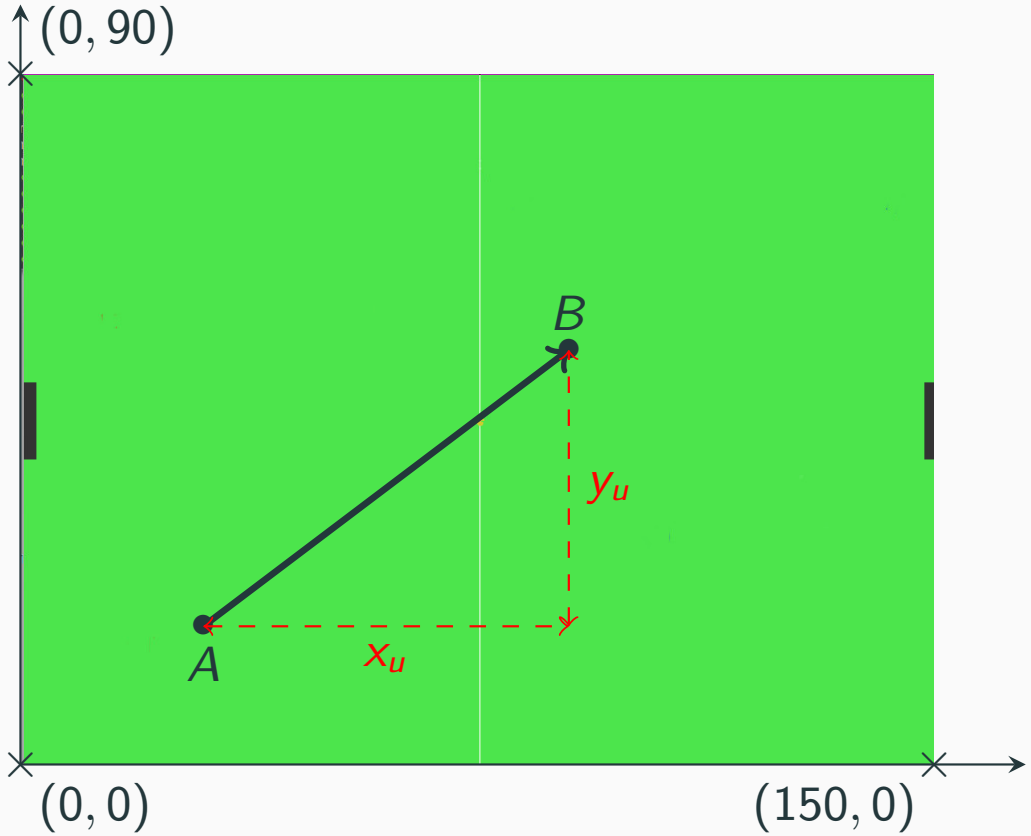
Lancer une partie

```
from soccersimulator import Strategy, SoccerAction, Vector2D
from soccersimulator import SoccerTeam,Simulation,Player,show_simu

class RandomStrategy(Strategy):
    def __init__(self):
        Strategy.__init__(self, "Random")
    def compute_strategy(self, state, id_team, id_player):
        return SoccerAction(Vector2D.create_random(),
                            Vector2D.create_random())

## Creation d'une equipe
pyteam = SoccerTeam("PyTeam")
## Ajout d'un joueur
pyteam.add("PyPlayer",RandomStrategy())
## Creation d'une deuxieme equipe et ajout d'un joueur (autre possibilite p
thon=SoccerTeam("ThonTeam",[Player("ThonPlayer",RandomStrategy())])

## Creation d'une simulation de 2000 pas de temps
match = Simulation(pyteam, thonteam,max_steps = 2000)
match.start() # ou
show_simu(match) # pour l'affichage graphique
```

Géométrie vectorielle

Géométrie 2D

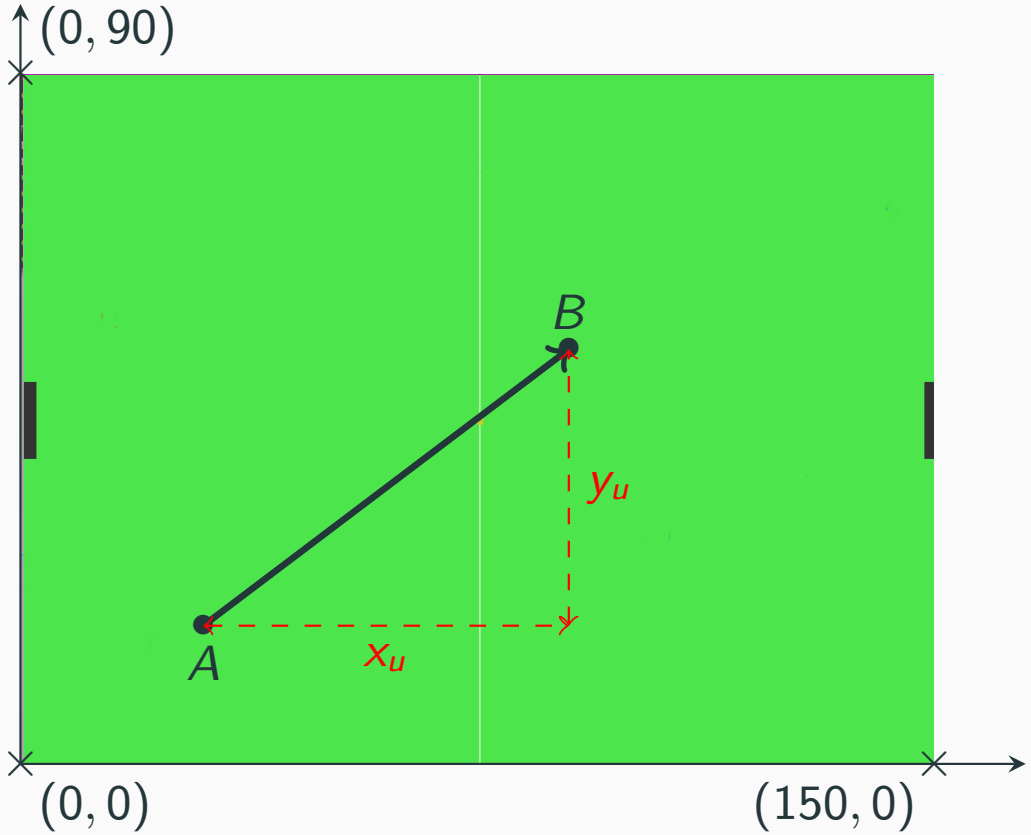
- Un point :
  $A : (x_A, y_A) \in \mathbb{R}^2$
- Un vecteur :
  $\boldsymbol{u} = (x_u, y_u) \in \mathbb{R}^2$
- Vecteur entre 2 points :
  $\overrightarrow{AB} = (x_B - x_A, y_B - y_A)$



Géométrie 2D

- Un point :
  $A : (x_A, y_A) \in \mathbb{R}^2$
- Un vecteur :
  $\boldsymbol{u} = (x_u, y_u) \in \mathbb{R}^2$
- Vecteur entre 2 points :
  $\overrightarrow{AB} = (x_B - x_A, y_B - y_A)$

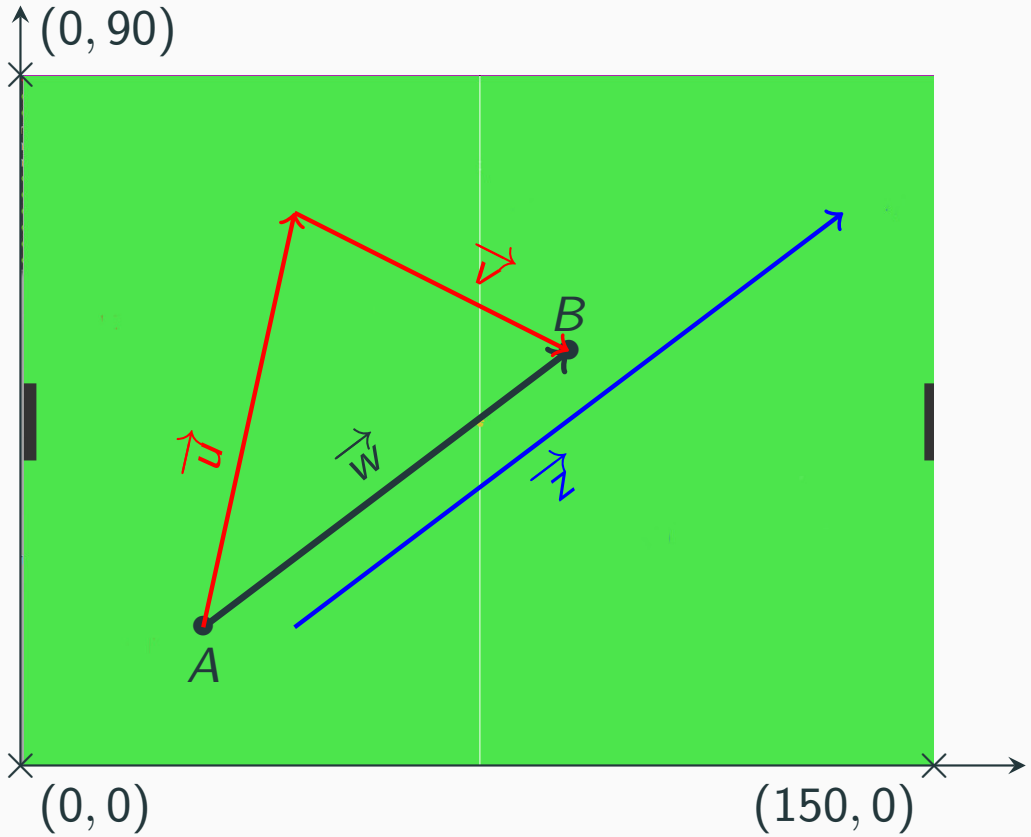
Un vecteur caractérise un déplacement, une vitesse, une accélération : une *norme* (puissance, force) et un *angle* (direction).



Opérations algébriques

$$
\begin{aligned}
\overrightarrow{w} &= \overrightarrow{u} + \overrightarrow{v} \\
\begin{pmatrix} x_w \\ y_w \end{pmatrix} &= \begin{pmatrix} x_u \\ y_u \end{pmatrix} + \begin{pmatrix} x_v \\ y_v \end{pmatrix} \\
&= \begin{pmatrix} x_u + x_v \\ y_u + y_v \end{pmatrix}
\end{aligned}
$$

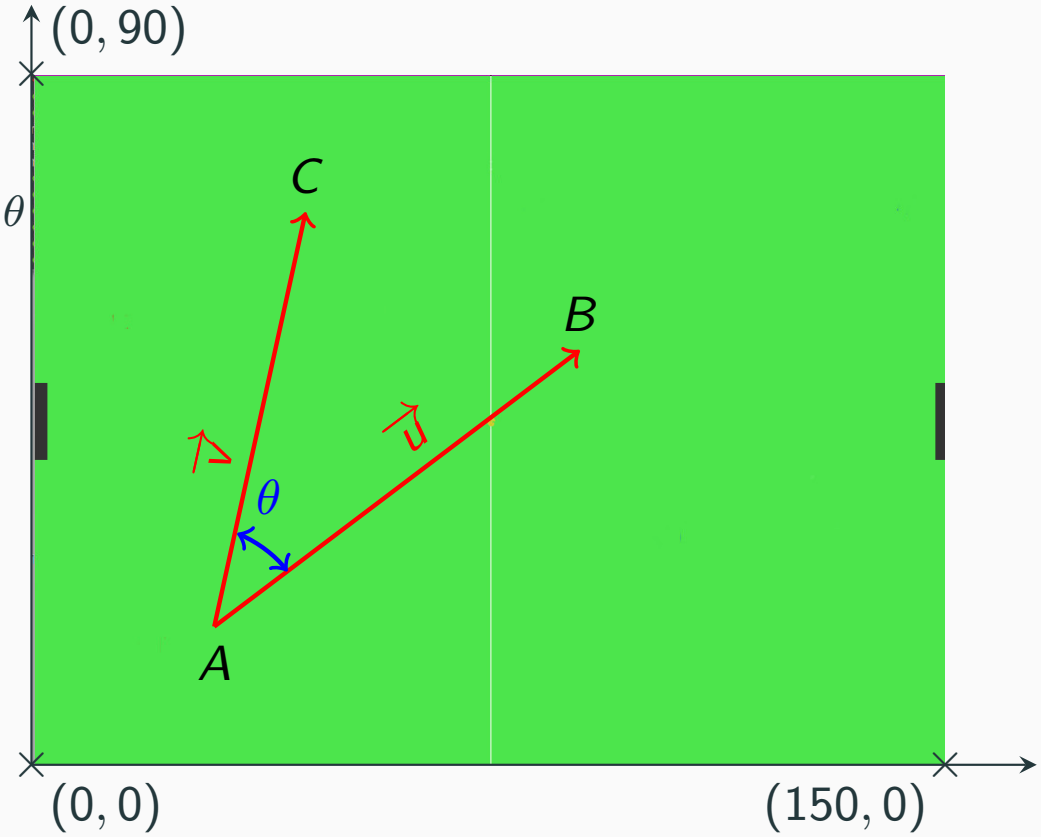
$$
\begin{aligned}
\overrightarrow{z} &= a\overrightarrow{w} \\
&= a \begin{pmatrix} x_w \\ y_w \end{pmatrix} \\
&= \begin{pmatrix} ax_w \\ ay_w \end{pmatrix}
\end{aligned}
$$



Produit scalaire

Propriétés

$$\begin{aligned}
\vec{u}.\vec{v} &= x_u x_v + y_u y_v \\
&= \|\vec{u}\|_{\ell_2} \|\vec{v}\|_{\ell_2} \cos\theta \\
(\vec{u} + \alpha\vec{v}).\vec{w} &= \vec{u}.\vec{w} + \alpha\vec{v}.\vec{w} \\
\\
\|\vec{u}\|_{\ell_2} &= \sqrt{\vec{u}.\vec{u}} \\
&= \sqrt{x_u^2 + y_u^2} \\
\|\alpha\vec{u}\|_{\ell_2} &= \alpha \|\vec{u}\|_{\ell_2}
\end{aligned}$$



- $\vec{u}$ et $\vec{v}$ colinéaire

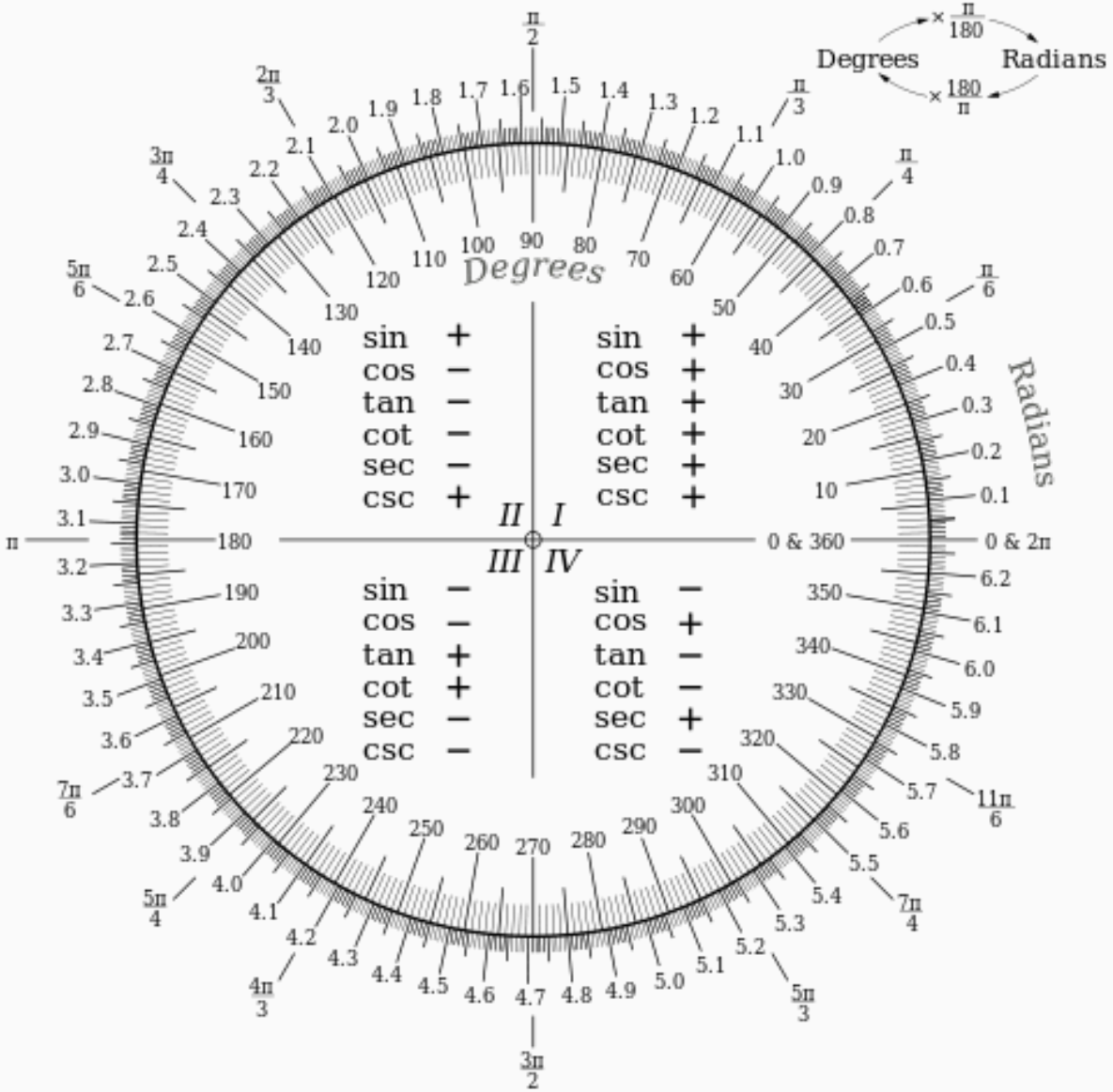
$\Leftrightarrow \vec{u} = \alpha\vec{v}$

$\Leftrightarrow \theta = 0,\ \vec{u}.\vec{v} = \|u\|_{\ell_2} \|v\|_{\ell_2}$

- $\vec{u}$ orthogonal à $\vec{v}$

$\Leftrightarrow \vec{u}.\vec{v} = 0,\ \theta = \pm\pi/2$

Les angles


Degrees
Radians
Degrees
Radians
sin +
cos −
tan −
cot −
sec −
csc +
sin +
cos +
tan +
cot +
sec +
csc +
sin −
cos −
tan +
cot +
sec −
csc −
sin −
cos +
tan −
cot −
sec +
csc −
II
I
III
IV
0 & 360
0 & 2π
180
π

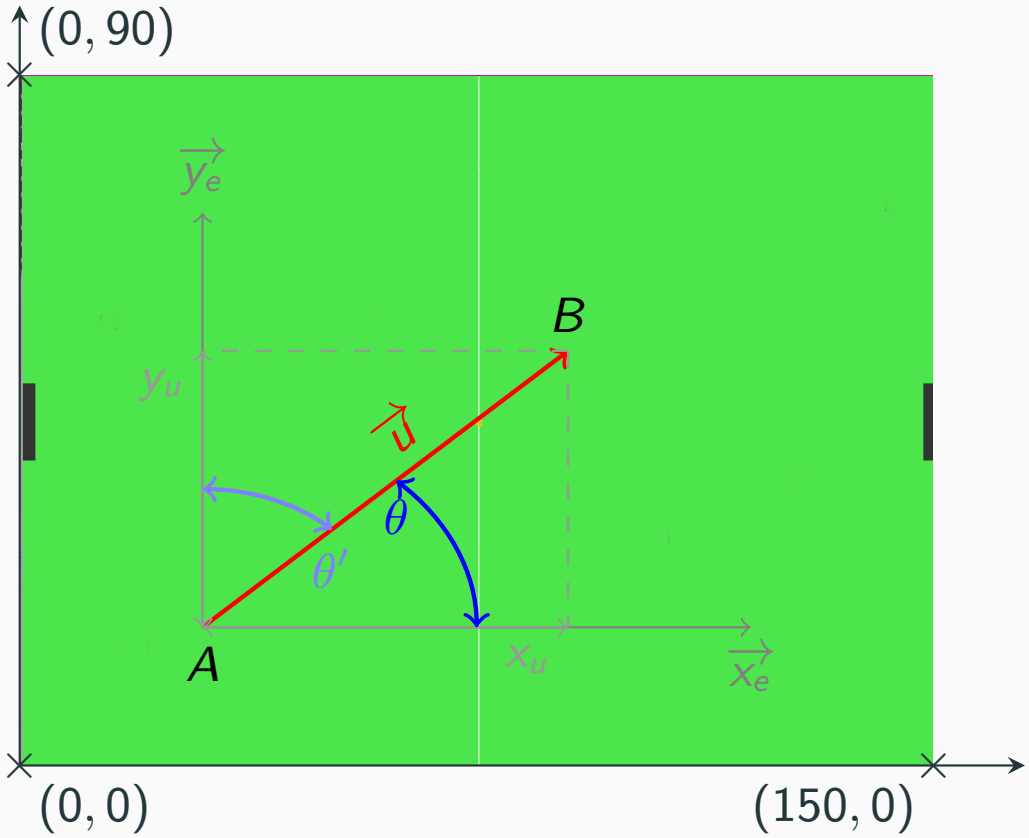
Décomposition dans la base normale

Coordonnées polaires

Rayon (norme) et angle à e_x

$$\begin{aligned}
\vec{u}.\vec{e_x} &= x_u \\
&= \|\vec{u}\|_{\ell_2} \cos\theta \\
&= \|\vec{u}\|_{\ell_2} \sin\theta' \\
\vec{u}.\vec{e_y} &= y_u \\
&= \|\vec{u}\|_{\ell_2} \cos\theta' \\
&= \|\vec{u}\|_{\ell_2} \sin\theta
\end{aligned}$$

cartésiennes	polaires
$\begin{pmatrix} x_u \\ y_u \end{pmatrix}$	$\begin{pmatrix} u_r = \|\vec{u}\|_{\ell_2} \\ u_\theta = \theta \end{pmatrix}$



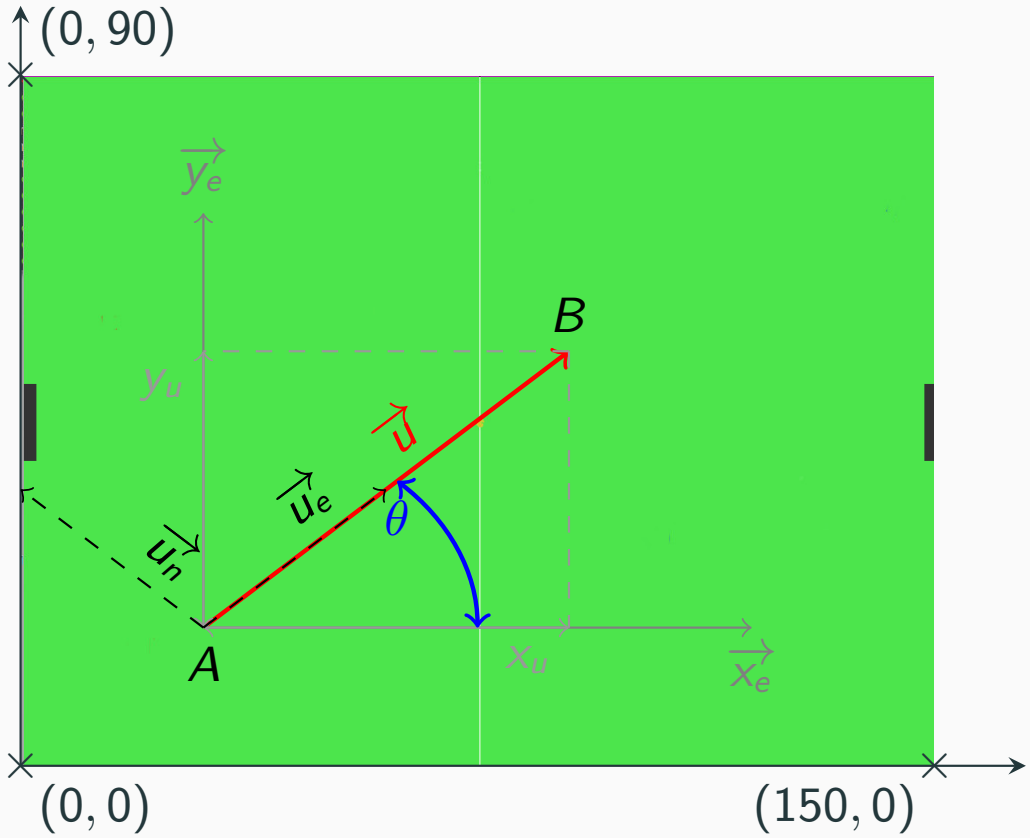
Changement de base

Engendré par un vecteur $\overrightarrow{u}$

Trouver $\overrightarrow{u_e}$ et $\overrightarrow{u_n}$, de norme 1 :

- $\overrightarrow{u_e}$ colinéaire à $\overrightarrow{u}$
- $\overrightarrow{u_n}$ normal à $\overrightarrow{u}$

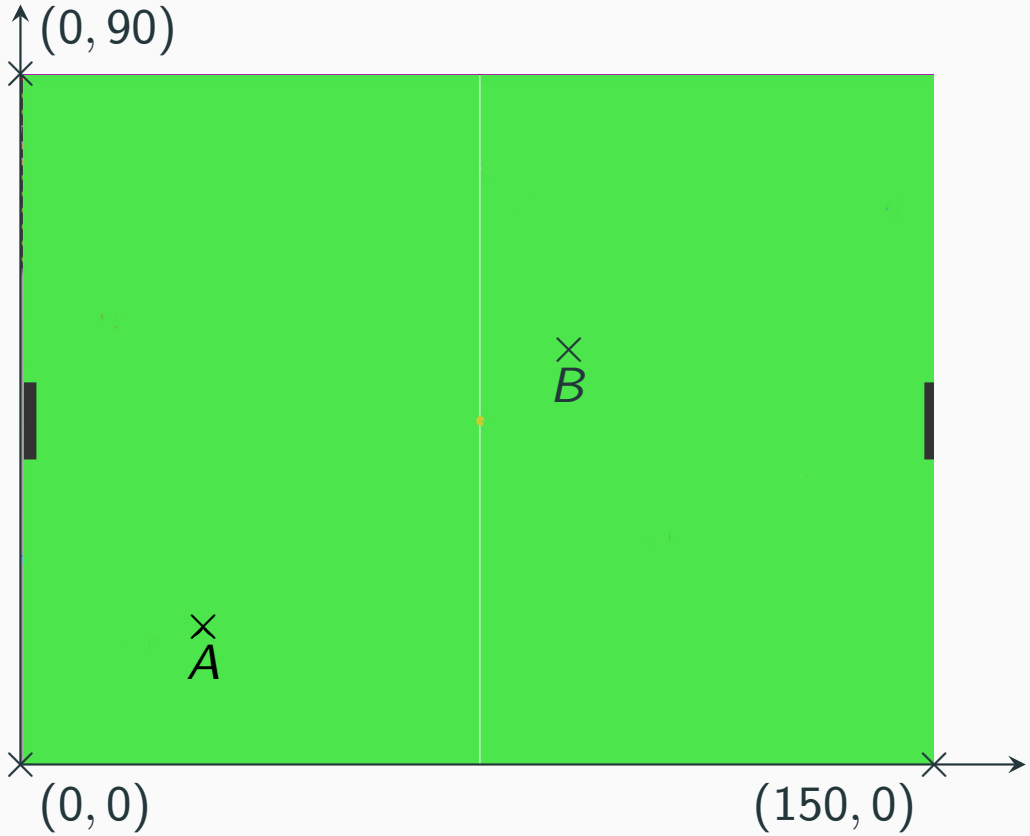
$$
\begin{aligned}
\overrightarrow{u_e} &= \frac{\overrightarrow{u}}{\|\overrightarrow{u}\|_{\ell_2}} \\
\text{(cart.)} &= \left(\frac{x_u}{\sqrt{x_u^2+y_u^2}}, \frac{y_u}{\sqrt{x_u^2+y_u^2}}\right) \\
\text{(polaires)} &= (1, \theta)
\end{aligned}
$$



$$
\begin{aligned}
\overrightarrow{u_n} &= \begin{pmatrix} x_u \cos \pi/2 - y_u \sin \pi/2 \\ x_u \sin \pi/2 + y_u \cos \pi/2 \end{pmatrix} \\
\text{(cart.)} &= \begin{pmatrix} -y_u \\ x_u \end{pmatrix} \\
\text{(polaires)} &= (1, \theta + \pi/2)
\end{aligned}
$$

Aller vers un point

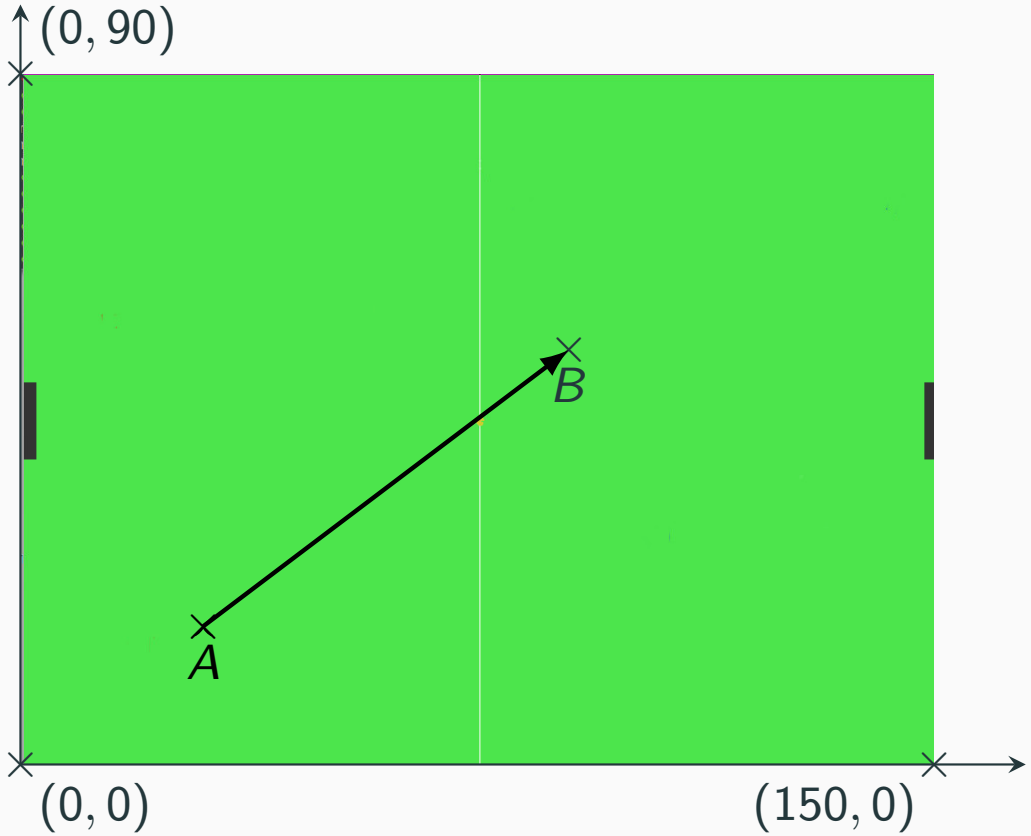
- A : position courante
- B : destination
- Comment quantifier le déplacement : $B = A + \overrightarrow{??}$?



Aller vers un point

- A : position courante
- B : destination
- Comment quantifier le déplacement : $B = A + \overrightarrow{??}$?
- Déplacement :

$$\begin{aligned}\overrightarrow{AB} &= \begin{pmatrix} x_{AB} \\ y_{AB} \end{pmatrix} \\ &= \begin{pmatrix} x_B \\ y_B \end{pmatrix} - \begin{pmatrix} x_A \\ y_A \end{pmatrix} \\ &= \begin{pmatrix} x_B - x_A \\ y_B - y_A \end{pmatrix}\end{aligned}$$



Objectifs du TME

Objectifs du TME

- prendre en main **git** ;
- créer un compte **github** ;
- installer la plateforme de simulation ;
- prendre en main Python et l'environnement ;
- programmer un joueur aléatoire ;
- programmation un joueur fonceur.

Dépôt git

`https://github.com/sangnier/Soccer`

Pour paramétrer le proxy

Ajouter dans `.bashrc` : `export https_proxy=proxy:3128`

Pour installer un module python manquant

```
pip install [MODULE] --user
```

Pour installer un module python stocké dans le repertoire courant

```
pip install . -e --user
```

Petite intro python

Python : un langage interprété

Peut être exécuté

- en console : interaction direct avec l'interpréteur
- par exécution de l'interpréteur sur un fichier script : `python fichier.py`

Opération élémentaire

```
# Affectation d'une variable
a = 3
# operations usuelles
(1 + 2. - 3.5), (3 * 4 /2 ),  4**2
# Attention ! reels et entiers
1/2, 1./2
# Operations logiques
True and False or (not False) == 2>1
# chaines de caracteres
s = "abcde"
s = s + s # concatenation
# afficher un resultat
print(1+1-2,s+s)
```

Structures : N-uplets et ensembles

Liste d'éléments ordonnes, de longueur fixe, non mutable : aucun élément ne peut être changé après la création du n-uplet

```
c = (1,2,3) # creation d'un n-uplet
c[0],c[1]  # acces aux elements d'un couple,
c + c  # concatenation de deux n-uplet
len(c) # nombre d'element du n-uplet
a, b, c = 1, 2, 3 # affectation d'un n-uplet de variables

s = set()  # creation d'un ensemble
s = {1 ,2 ,1}
print(len(s)) #taille d'un ensemble
s.add('s')   # ajout d'un element
s.remove('s') # enlever un element
s.intersection({1,2,3,4})
s.union({1,2,3,4})
```

Structures : Listes

Structure très importante en python. Il n'y a pas de tableau, que des listes (et des dictionnaires)

```
l = list() # creation liste vide
l1 = [ 1, 2 ,3 ] # creation d'une liste avec elements
l = l + [4, 5] #concatenation
zip(l1,l2) : liste des couples
len(l) #longueur
l.append(6)      # ajout d'un element
l[3]      #acces au 4-eme element
l[1:4]    # sous-liste des elements 1,2,3
l[-1],l[-2] # dernier element, avant-dernier element
sum(l)      # somme des elements d'une liste
sorted(l)      #trier la liste
l = [1, "deux", 3] # une liste compposee
sub_list1 = [ x for x in l1 if x < 2] # liste comprehension
sub_list2 = [ x + 1 for x in l1 ] # liste comprehension 2
sub_list3 = [x+y for x,y in zip(l1,l1)] # liste comprehension 3
```

Structures : Dictionnaires

Dictionnaires : listes indexées par des objets (hashmap), très utilisées également. Ils permettent de stocker des couples (clé,valeur), et d'accéder aux valeurs a partir des clés.

```
d = dict() # creation d'un dictionnaire
d['a']=1   # presque tout type d'objet peut etre
d['b']=2   # utilise comme cle, tout type d'objet
d[2]= 'c'  # comme valeur
d.keys()   # liste des cles du dictionnaire
d.values() # liste des valeurs contenues dans le dictionnaire
d.items() # liste des couples (cle,valeur)
len(d)    #nombre d'elements d'un dictionnaire
d = dict([ ('a',1), ('b',2), (2, 'c')])  # autre methode pour creer un dict
d = { 'a':1, 'b':2, 2:'c'}  # ou bien...
d = dict( zip(['a','b',2],[1,2,'c']))  #et egalement...
d.update({'d':4,'e':5}) # "concatenation" de deux dictionnaires
```

Boucles, conditions

Attention, en python toute la syntaxe est dans l'indentation : un bloc est formé d'un ensemble d'instructions ayant la même indentation (même nombre d'espaces précédent le premier caractère).

```
i=0
s=0
while i<10:   # boucle while
    i+=1      #indentation pour marquer ce qui fait parti de la boucle
    s+=i
s=0
for i in [1, 2, 3]: #boucle for
    j = 0           # indentation pour le for
    while j<i:       # boucle while
        j+=1         # deuxieme indentation pour le bloc while
        s = i + j
    s = s + s # retour a la premiere indentation, instruction du bloc for
```

Fonctions

```
def increment(x):    # definition d'une fonction par le mot-cle def
    return x+1       # retour de la fonction

y=increment(5)       # appel de la fonction

def somme_soustraction(x,y=2):
    # possibilite de donner une valeur par defaut aux parametres
    return x+y,x-y               #  possibilite de retourner
                                 #un n-uplet de valeurs,
                                 # equivalent a (x+y,x-y)
xsom,xsub = somme_soustraction(10,5) #ou
res = somme_soustraction(10,5)
xsom == res[0],res[1]
```

Fichiers

```
##Lire
f=open("/dev/null","r")
print(f.readline())
f.close()

#ou plus simplement
with open("/dev/null","r") as f :
    for l in f:
        print l

## Ecrire
f=open("/dev/null","w")
f.write("toto\n")
f.close()

#ou
with open("/dev/null","w") as f:
    for i in range(10):
        f.write(str(i))
```

Modules

- Un module groupe des objets pouvant être réutilisés
  - module `math` : `cos, sin, tan, log, exp, ...`
  - module `string` : manipulation de chaîne de caractères
  - module `numpy` : librairie scientifique
  - modules `sys`, `os` : manipulation de fichiers et du système
  - module `pdb`, `cProfile` : debuggage, profiling
- importer un module : `import module [as surnom]` et accès au module par `module.fonction` (ou `surnom.fonction`)
- importer un sous-module ou une fonction : `from module import sousmodule`
- tout répertoire dans le chemin d'accès qui comporte un fichier `__init__.py` est considéré comme un module !
- tout fichier python dans le répertoire courant est considéré comme module : `import fichier` si le fichier est `fichier.py` (ou plus souvent `from fichier import *`)

Les objets : très grossièrement

- c'est une structure : contient des variables stockant des informations
- contient des *méthodes* (fonctions) qui agissent sur ses variables,
- contient *un constructeur*, fonction spécifique qui sert à l'initialiser.
- le . sert à indiquer l'appartenance d'un objet/fonction à un autre objet : `obj.fun` est l'appel de la fonction `fun` de l'objet `obj`
- *self* indique l'objet lui-même

Un objet `Agent` pourrait être ainsi le suivant :

```
class Agent(object):
      def __init__(self,nom):
          self.nom = nom
          self.x = 0
          self.y = 0
      def agir(self,etat):
          action = None
          return action
      def get_position(self):
          return self.x, self.y
      def safficher(self):
          print("Je␣suis␣%s␣en␣%d,%d"
                %(self.nom,self.x,self.y))
```

```
a = Agent("John") # creation
a.x, a.y = 1, 1 # deplacement
a.safficher() #equivalent a
Agent.safficher(a)
a.mavar = 4 #ajout d'une variable
```