

# 2I013 Groupe 1

## Projet Foot

### cours 2

Nicolas Baskiotis - Maxime Sangnier

`nicolas.baskiotis@lip6.fr`

`maxime.sangnier@upmc.fr`

Laboratoire d'Informatique de Paris 6 (LIP6)  
Université Pierre et Marie Curie (UPMC) - Sorbonne Universités

S2 (2017-2018)

# Plan

**Géométrie vectorielle**

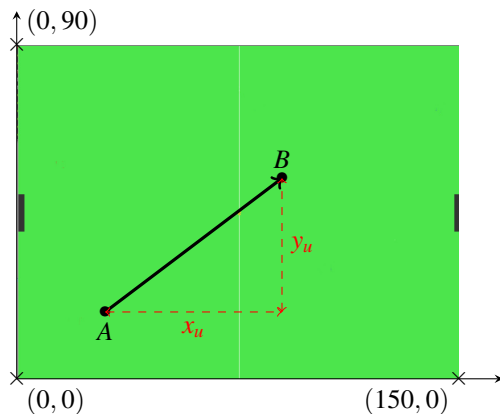
Le simulateur décortiqué

Quelques problèmes géométriques

# Quelques rappels

## Géométrie 2D

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

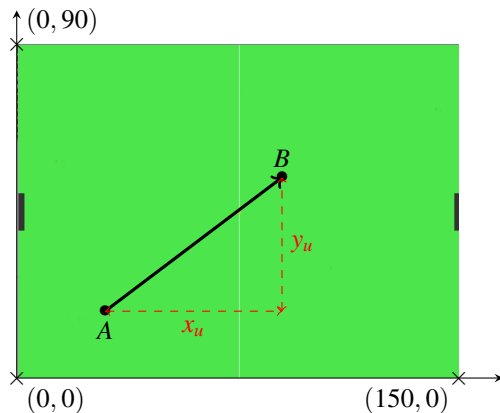


# Quelques rappels

## Géométrie 2D

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

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

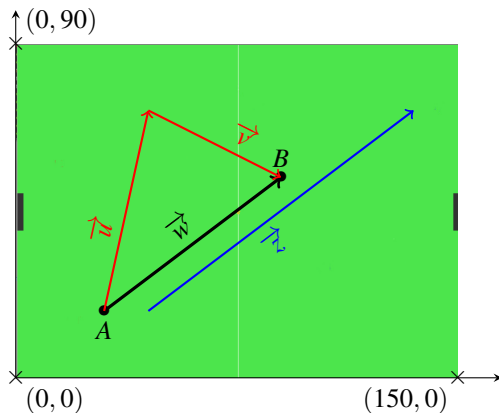


# Quelques rappels

## Opérations algébriques

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

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



# Produit scalaire

## Propriétés

$$\begin{aligned}\vec{u} \cdot \vec{v} &= x_u x_v + y_u y_v \\ &= \|\vec{u}\| \|\vec{v}\| \cos \theta \\ (\vec{u} + \alpha \vec{v}) \cdot \vec{w} &= \vec{u} \cdot \vec{w} + \alpha \vec{v} \cdot \vec{w}\end{aligned}$$

$$\begin{aligned}\|\vec{u}\| &= \sqrt{\vec{u} \cdot \vec{u}} \\ &= \sqrt{x_u^2 + y_u^2} \\ \|\alpha \vec{u}\| &= \alpha \|\vec{u}\|\end{aligned}$$

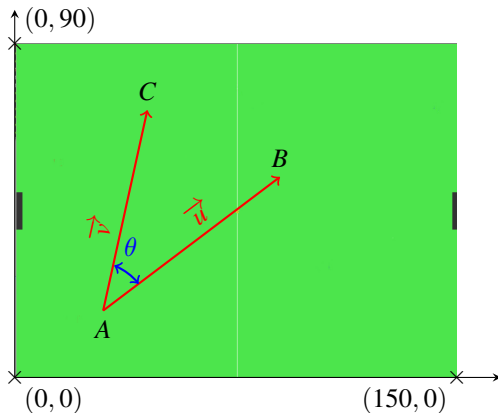
- $\vec{u}$  et  $\vec{v}$  colinéaire

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

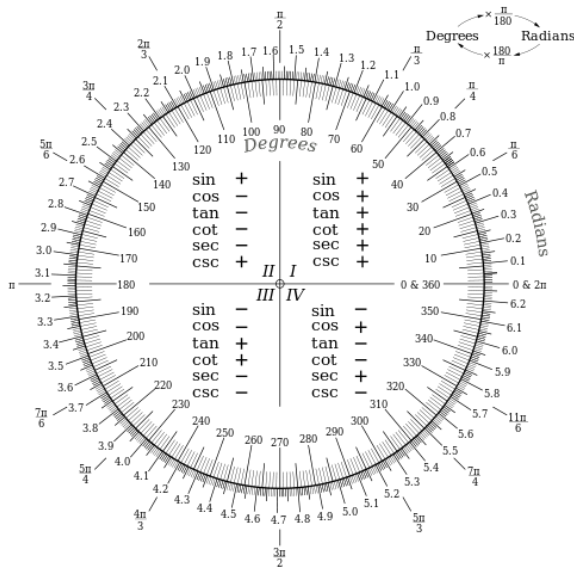
$$\Leftrightarrow \theta = 0, \vec{u} \cdot \vec{v} = \|u\| \|v\|$$

- $\vec{u}$  orthogonal à  $\vec{v}$

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



# Les angles



# Décomposition dans la base normale

## Coordonnées polaires

Rayon (norme) et angle à  $e_x$

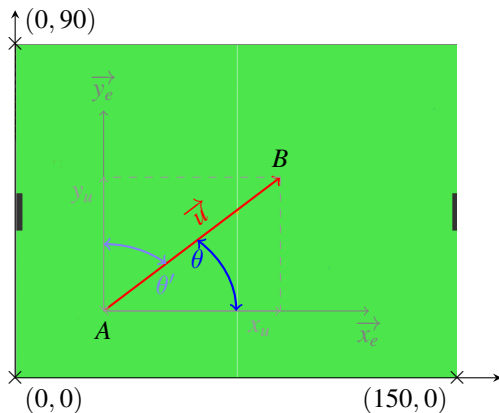
$$\begin{aligned}\vec{u} \cdot \vec{e}_x &= x_u \\ &= \|\vec{u}\| \cos \theta \\ &= \|\vec{u}\| \sin \theta'\end{aligned}$$

$$\begin{aligned}\vec{u} \cdot \vec{e}_y &= y_u \\ &= \|\vec{u}\| \cos \theta' \\ &= \|\vec{u}\| \sin \theta\end{aligned}$$

cartésiennes

polaires

$$\begin{pmatrix} x_u \\ y_u \end{pmatrix} \quad \left| \quad \begin{pmatrix} u_r = \|\vec{u}\| \\ u_\theta = \theta \end{pmatrix}\right.$$





# Changement de base

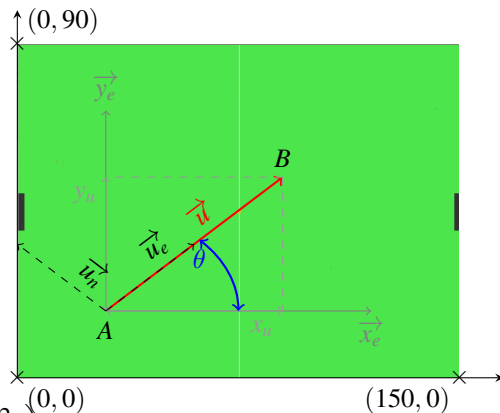
## Engendré par un vecteur $\vec{u}$

Trouver  $\vec{u}_e$  et  $\vec{u}_n$ , de norme 1 :

- $\vec{u}_e$  colinéaire à  $\vec{u}$
- $\vec{u}_n$  normal à  $\vec{u}$

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

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



# Plan

Géométrie vectorielle

**Le simulateur décortiqué**

Quelques problèmes géométriques

# Les objets en présence (et leurs attributs )

- `Vector2D` : représente un point ou un vecteur;
- `MobileMixin` : représente un objet mobile (position, vitesse);
- `SoccerAction` : représente l'action d'un joueur (accélération, shoot);
- `Player` : représente un joueur (nom, stratégie);
- `Strategy` : représente une stratégie;
- `PlayerState` : représente un état d'un joueur (position, vitesse)
- `SoccerState` : représente un état du jeu (balle, joueurs, score)
- `SoccerTeam` : liste de joueurs et de leur stratégie
- `Simulation` : une simulation de match

# Les objets en présence (et leurs attributs)

SoccerAction
acceleration: Vector2D shoot: Vector2D
copy()

Ball
vitesse: double position: double
inside_goal() next(sum_of_shoots)

Strategy
name: string
compute_strategy(state,id_team,id_player)

Vector2D
angle: double norm: double x: double y: double
copy() static create_random(low,high) distance(Vector2D) dot(Vector2D) from_polar(angle,norm) norm_max(norm) normalize() random(low,high) scale() set(Vector2D)

SoccerState
ball: Ball goal: int max_steps: int states: dict((id_team,id_player) -> PlayerState) players: [string] score_team1: int score_team2: int step: int strategies: dict((id_player,id_team)->string)
player_state(id_team,id_player) static create_initial_state(nb_players_1,nb_players_2) get_score_team(id_team) nb_players(id_team)

Player
name: string strategy: SoccerStrategy

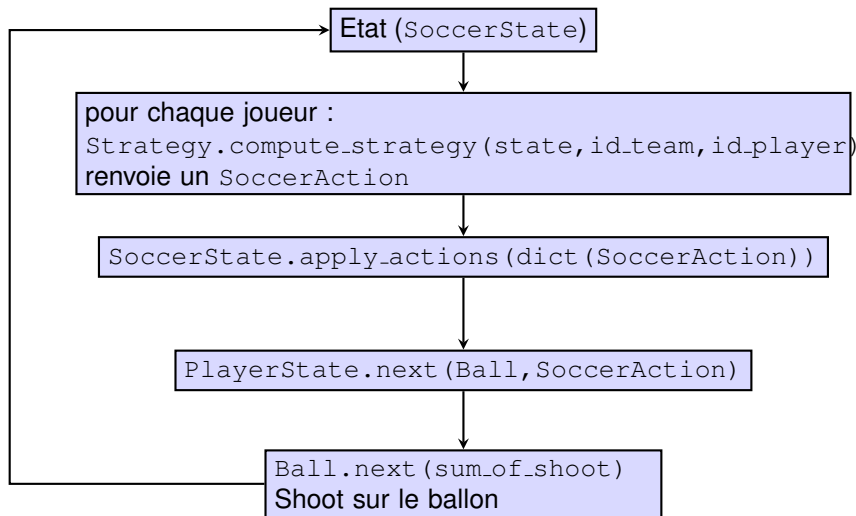
PlayerState
position: Vector2D vitesse: Vector2D action: SoccerAction --acceleration: Vector2D --shoot: Vector2D
can_shoot() copy() next(ball,action)

# Les commandes utiles

## Etant donné un état `state`

- `state.ball` : `MobileMixin` de la balle
- `state.ball.position` : la position de la balle  
(`state.ball.position.x`, `state.ball.position.y`)
- `state.ball.vitesse` : la vitesse de la balle
- `state.player(idteam, idplayer)` : la configuration d'un joueur
- `state.player_state(idteam, idplayer)` : `MobileMixin` du joueur
- `state.player_state(idteam, idplayer).position` : position du joueur
- `state.player_state(idteam, idplayer).vitesse` : vitesse du joueur
- `state.players` : liste des clés (`idteam, idplayer`) de tous les joueurs

# Le cœur du simulateur



# Le moteur d'un joueur

```
def next(self, ball, action=None):
    if not (hasattr(action, "acceleration") and hasattr(action, "shoot")):
        action = SoccerAction()
        self.action = action.copy()
        self.vitesse *= (1 - settings.playerBrackConstant)
        self.vitesse = (self.vitesse + \
            self.acceleration).norm_max(settings.maxPlayerSpeed)
        self.position += self.vitesse
    if self.position.x < 0 or self.position.x > settings.GAME_WIDTH \
        or self.position.y < 0 \
        or self.position.y > settings.GAME_HEIGHT:
        self.position.x = max(0, min(settings.GAME_WIDTH, self.position.x))
        self.position.y = max(0, min(settings.GAME_HEIGHT, self.position.y))
        self.vitesse = Vector2D()
    if self.shoot.norm == 0 or not self.can_shoot():
        self._dec_shoot()
        return Vector2D()
    self._reset_shoot()
    if self.position.distance(ball.position) \
        > (settings.PLAYER_RADIUS + settings.BALL_RADIUS):
        return Vector2D()
    return self._rd_angle(self.shoot, (self.vitesse.angle - self.shoot.angle,
        self.position.distance(ball.position) / (settings.PLAYER_RADIUS +
```

# Le moteur du ballon

```
def next(self, sum_of_shoots):
    vitesse = self.vitesse.copy()
    vitesse.norm = self.vitesse.norm - \
        settings.ballBrakeSquare * \
        self.vitesse.norm ** 2 - \
        settings.ballBrakeConstant * self.vitesse.norm
    ## decomposition selon le vecteur unitaire de ball.speed
    snorm = sum_of_shoots.norm
    if snorm > 0:
        u_s = sum_of_shoots.copy()
        u_s.normalize()
        u_t = Vector2D(-u_s.y, u_s.x)
        speed_abs = abs(vitesse.dot(u_s))
        speed_ortho = vitesse.dot(u_t)
        speed_tmp = Vector2D(speed_abs * u_s.x \
            - speed_ortho * u_s.y, \
            speed_abs * u_s.y + speed_ortho * u_s.x)
        speed_tmp += sum_of_shoots
        vitesse = speed_tmp
    self.vitesse = vitesse.norm_max(settings.maxBallAcceleration)
    self.position += self.vitesse
```



# Plan

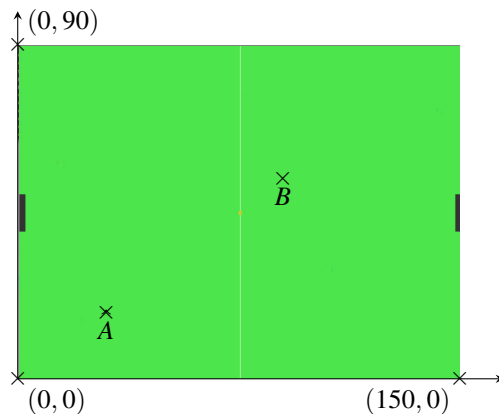
Géométrie vectorielle

Le simulateur décortiqué

**Quelques problèmes géométriques**

# Aller vers un point ?

- A position courante
- P : destination
- Quelle action ?

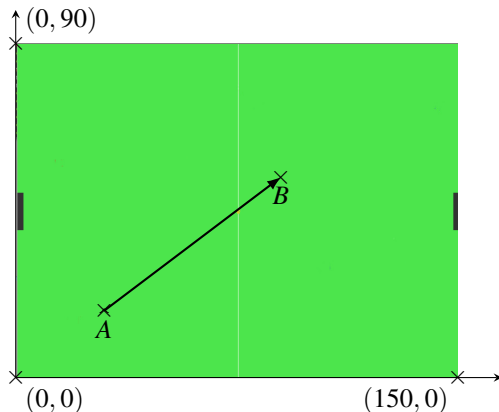


# Aller vers un point ?

- A position courante
- P : destination
- Quelle action ?
- Vecteur vitesse:

$$\vec{v} = \overrightarrow{AB}$$

- Importance de la norme ?



```
def compute_strategy(self, state, id_team, id_player):  
    return SoccerAction(state.ball.position  
                        -state.player_state(id_team,id_player).position,  
                        Vector2D())
```

# Proposition de stratégie

## Stratégie naïve

- Fonceur
- Goal
- ...

# Proposition de stratégie

## Stratégie naïve

- Fonceur
  - Goal
  - ...
  - Comment choisir entre les différentes stratégies ?
  - Comment le faire de manière élégante ?
- ⇒ Coder des petites fonctions légères et génériques !

# Création d'une Toolbox

## Dans un fichier séparé (par exemple `tools.py`)

Inclure les fonctions usuelles pour :

- aller vers un point
- shooter vers le but
- trouver l'adversaire le plus proche
- ... (toutes les petites fonctions récurrentes dont vous aurez besoin)

## Puis dans votre fichier

```
from tools import *  
...
```

Réfléchissez à la structure de vos fonctions :

- Elles doivent être générique (situation miroir selon l'identifiant de l'équipe)
- Facile à manier.
- Possibilité d'encapsuler l'objet `SoccerState`.

# Encapsuler un objet

## Il s'agit

- d'enrichir un objet de nouvelles fonctionnalités;
- de *traduire* certaines de ses propriétés (par exemple objet miroir)
- d'en faciliter l'utilisation.

## Exemple

```
class MyState(object):
    def __init__(self, state, idteam, idplayer):
        self.state = state
        self.key = (idteam, idplayer)
    def my_position(self):
        return self.state.player_state(*pkey).position
        #equivalent a self.player_state(self.key[0], self.key[1])
    def ball_position(self):
        return self.state.ball.position
    def aller(self, p):
        return SoccerAction(p-self.my_position(), Vector2D())
    def shoot(self, p):
        return SoccerAction(Vector2D(), p-self.state.my_position())
    def compute_strategy(self):
        return self.aller(p)=self.shoot(p)
```

# Encapsuler un objet

## Il s'agit

- d'enrichir un objet de nouvelles fonctionnalités;
- de *traduire* certaines de ses propriétés (par exemple objet miroir)
- d'en faciliter l'utilisation.

## Exemple

```
class MyStrategy (Strategy) :
    def __init__(self):
        Strategy.__init__(self, "Ma_strat")
    def compute_strategy(self, state, idteamn, idplayer) :
        return MyState(state, idteam, idplayer).compute_strategy()

#ou
class SousStrat (Strategy) :
    def __init__(self, sous_strat):
        Strategy.__init__(self, sous_strat.name)
        self.strat=sous_strat
    def compute_strategy(self, state, idteam, idplayer) :
        return self.strat(MyState(state, idteam, idplayer))

def fonceur(me) :
    return me.aller(me.ball_position)+me.shoot(me.but_adv)
fonceur.name = "Fonceur"; FonceurStrat = SousStrat(fonceur)
```