Projet Foot
2I013

Nicolas Baskiotis
nicolas.baskiotis@lip6.fr

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

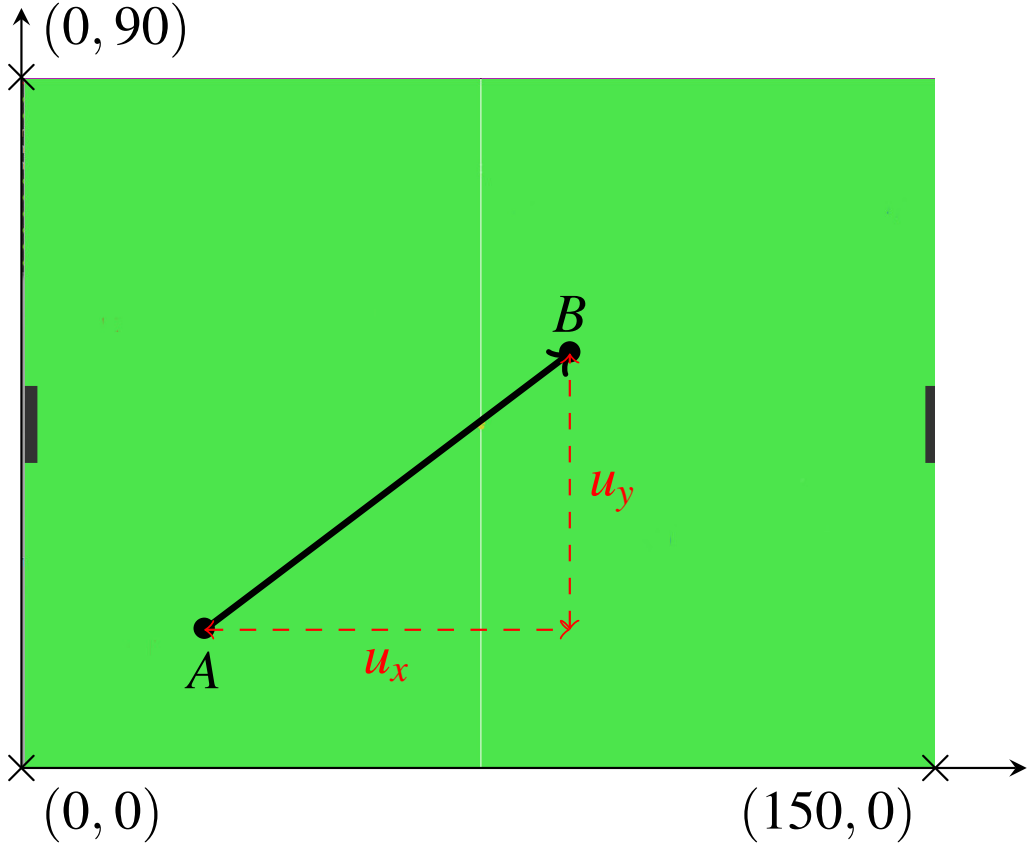
S2 (2015-2016)

Plan

Quelques rappels

Géométrie 2D

- Un point :
  $A : (A_x, A_y) \in \mathbb{R}^2$
- Un vecteur :
  $\vec{u} = (u_x, u_y) \in \mathbb{R}^2$
- Vecteur entre 2 points :
  $\overrightarrow{AB} = (B_x - A_x, B_y - Ay)$

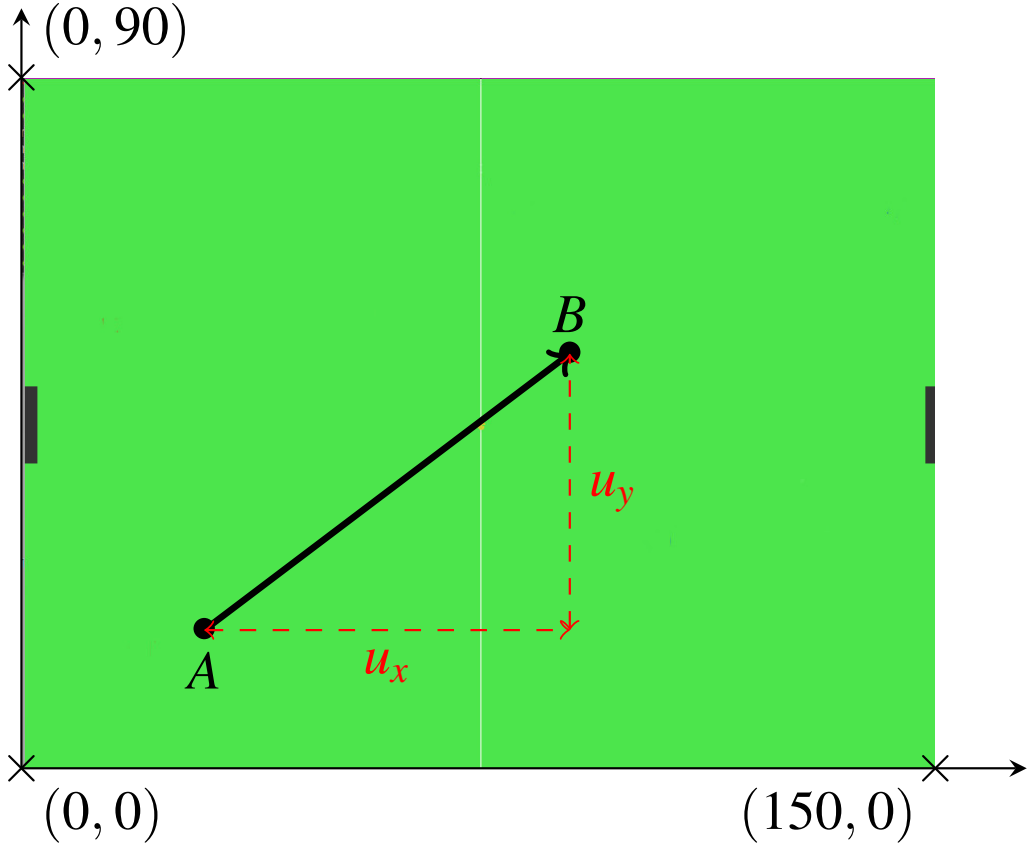


Quelques rappels

Géométrie 2D

- Un point :
  $A : (A_x, A_y) \in \mathbb{R}^2$
- Un vecteur :
  $\vec{u} = (u_x, u_y) \in \mathbb{R}^2$
- Vecteur entre 2 points :
  $\overrightarrow{AB} = (B_x - A_x, B_y - Ay)$

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

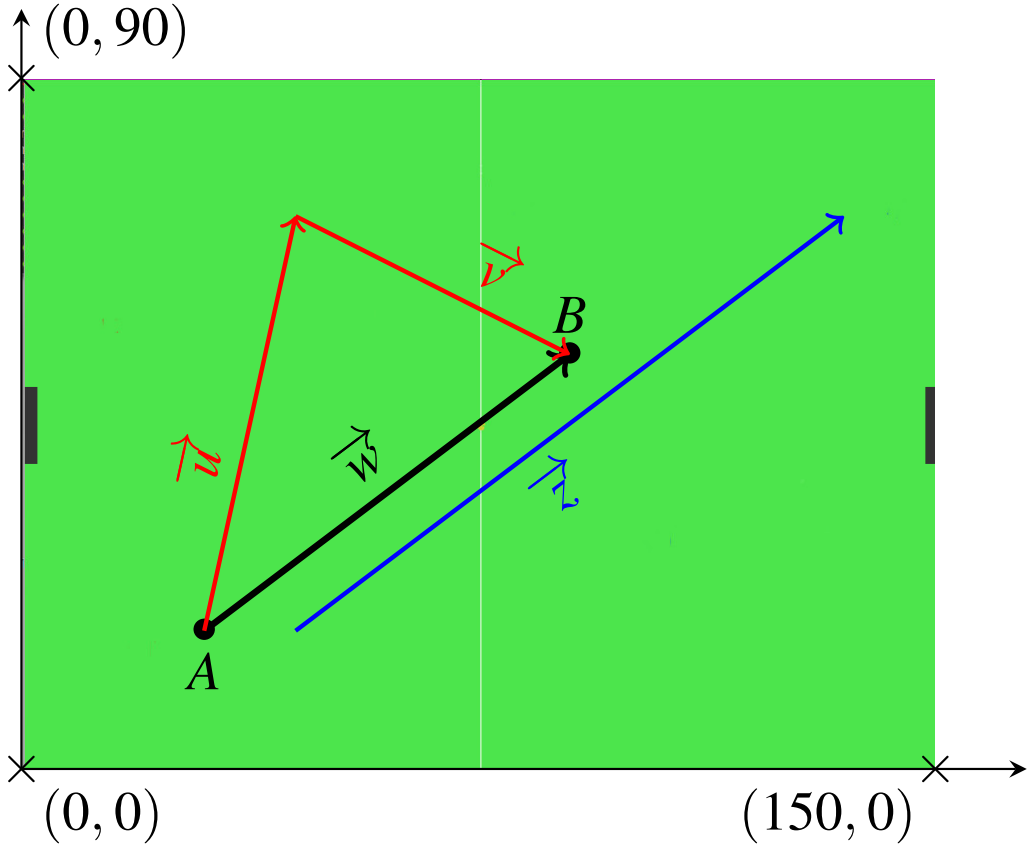


Quelques rappels

Opérations algébriques

$$\begin{aligned}\overrightarrow{w} &= \overrightarrow{u} + \overrightarrow{v}\\ \begin{pmatrix} w_x \\ w_y \end{pmatrix} &= \begin{pmatrix} u_x \\ u_y \end{pmatrix} + \begin{pmatrix} v_x \\ v_y \end{pmatrix}\\ &= \begin{pmatrix} u_x + v_x \\ u_y + v_y \end{pmatrix}\end{aligned}$$

$$\begin{aligned}\overrightarrow{z} &= a\overrightarrow{w}\\ &= a\begin{pmatrix} w_x \\ w_y \end{pmatrix}\\ &= \begin{pmatrix} aw_x \\ aw_y \end{pmatrix}\end{aligned}$$



Produit scalaire

Propriétés

$$\begin{aligned}
\vec{u}.\vec{v} &= u_x v_x + u_y v_y \\
&= \|\vec{u}\| \, \|\vec{v}\| \cos\theta \\
(\vec{u} + \alpha\vec{v}).\vec{w} &= \vec{u}.\vec{w} + \alpha\,\vec{v}.\vec{w}
\end{aligned}$$

$$\begin{aligned}
\|\vec{u}\| &= \sqrt{\vec{u}.\vec{u}} \\
&= \sqrt{u_x^2 + u_y^2} \\
\|\overrightarrow{\alpha u}\| &= \alpha \|\vec{u}\|
\end{aligned}$$

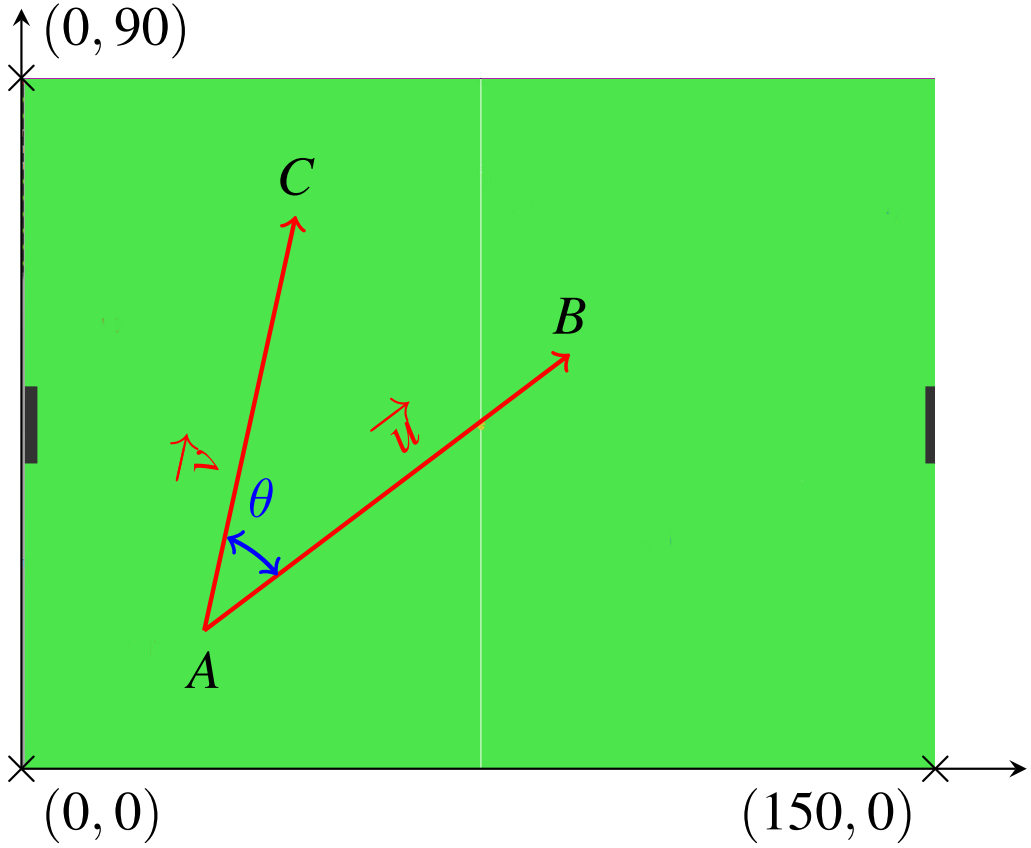
- $\vec{u}$ et $\vec{v}$ colinéaire

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

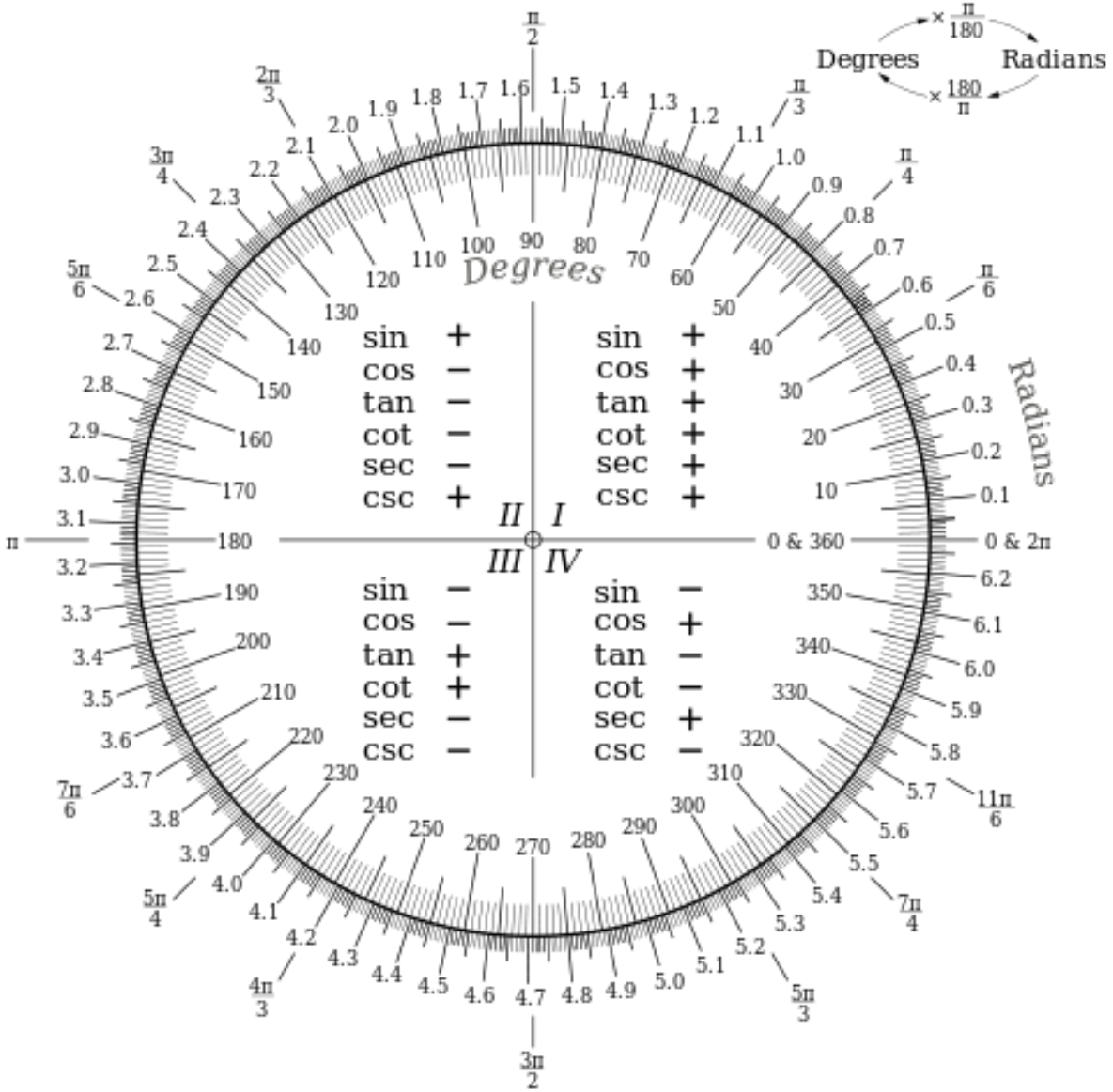
$\Leftrightarrow \theta = 0,\ \vec{u}.\vec{v} = \|u\| \, \|v\|$

- $\vec{u}$ orthogonal à $\vec{v}$

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



Les angles


Degrees
Radians
× π/180
× 180/π
Degrees
Radians
II
I
III
IV
sin + cos − tan − cot − sec − csc +
sin + cos + tan + cot + sec + csc +
sin − cos − tan + cot + sec − csc −
sin − cos + tan − cot − sec + csc −
0 & 360
0 & 2π
π
π/2
π/3
π/4
π/6
2π/3
3π/4
5π/6
7π/6
5π/4
4π/3
3π/2
5π/3
7π/4
11π/6

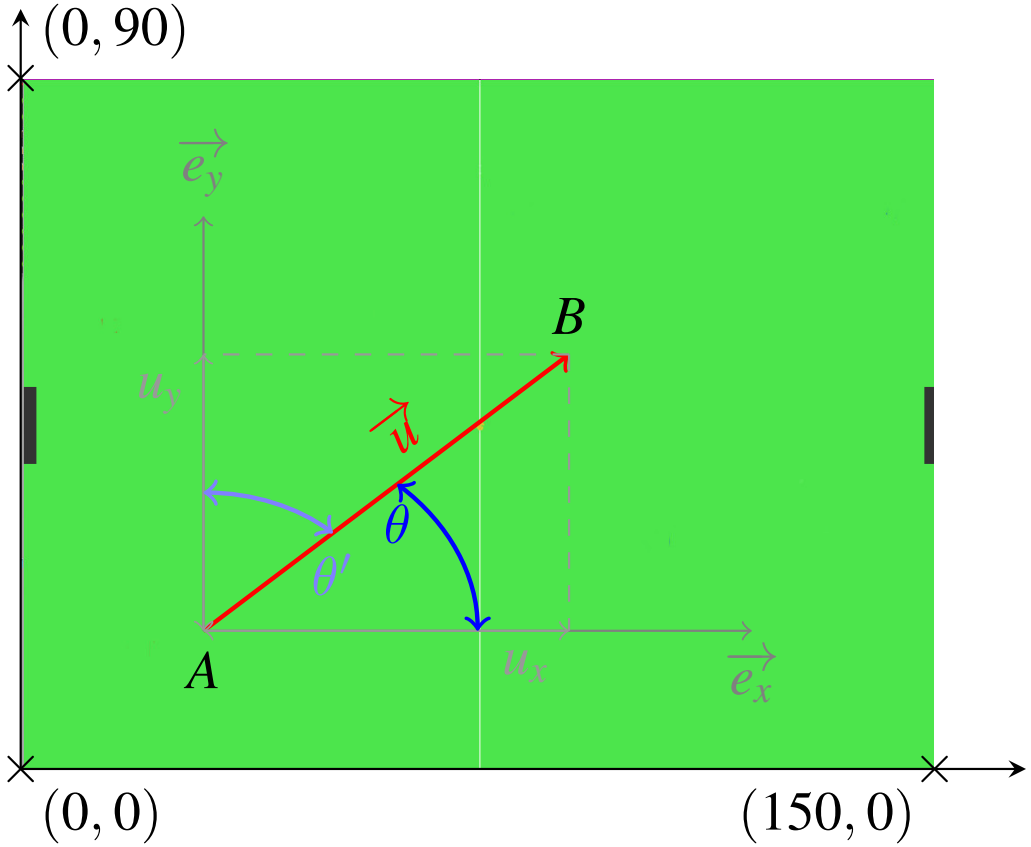
Décomposition dans la base normale

Coordonnées polaires

Rayon (norme) et angle à e_x

$$\begin{aligned}
\vec{u}\,.\vec{e_x} &= u_x \\
&= \|\vec{u}\| \cos\theta \\
&= \|\vec{u}\| \sin\theta' \\
\vec{u}\,.\vec{e_y} &= u_y \\
&= \|\vec{u}\| \cos\theta' \\
&= \|\vec{u}\| \sin\theta
\end{aligned}$$

cartésiennes	polaires
$\begin{pmatrix} u_x \\ u_y \end{pmatrix}$	$\begin{pmatrix} u_r = \|\vec{u}\| \\ u_\theta = \theta \end{pmatrix}$



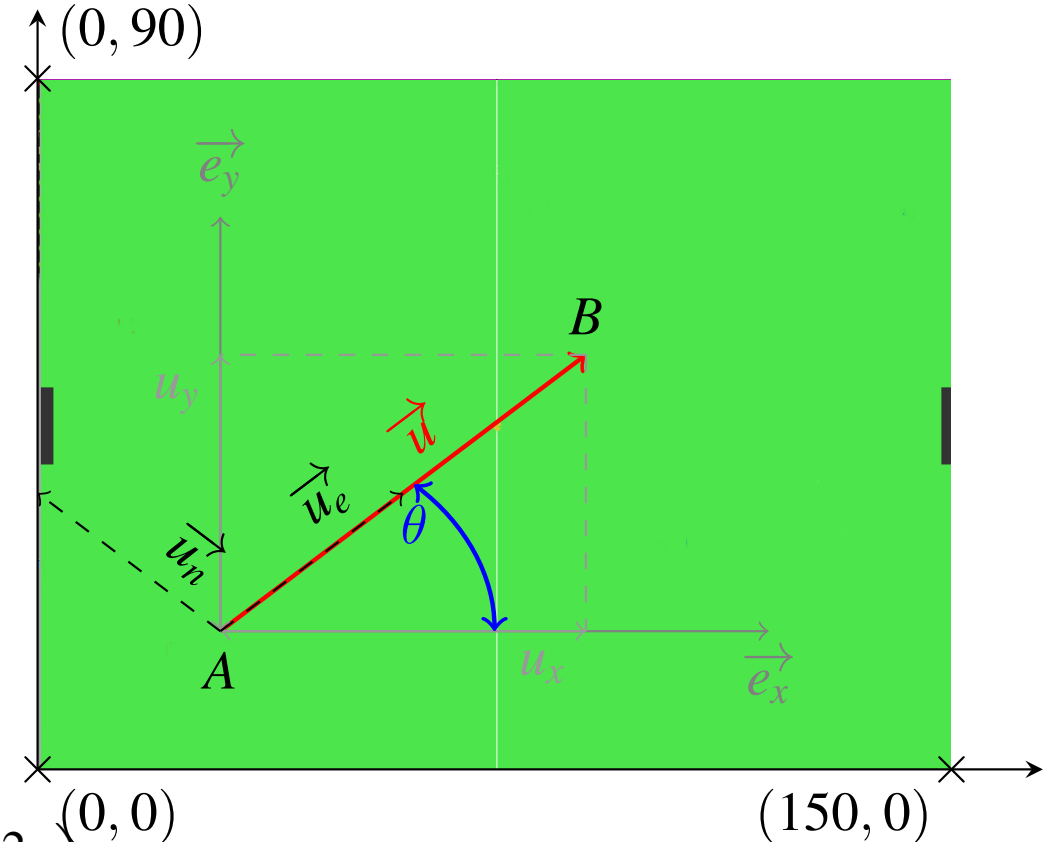
Changement de base

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 $\overrightarrow{u}$

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



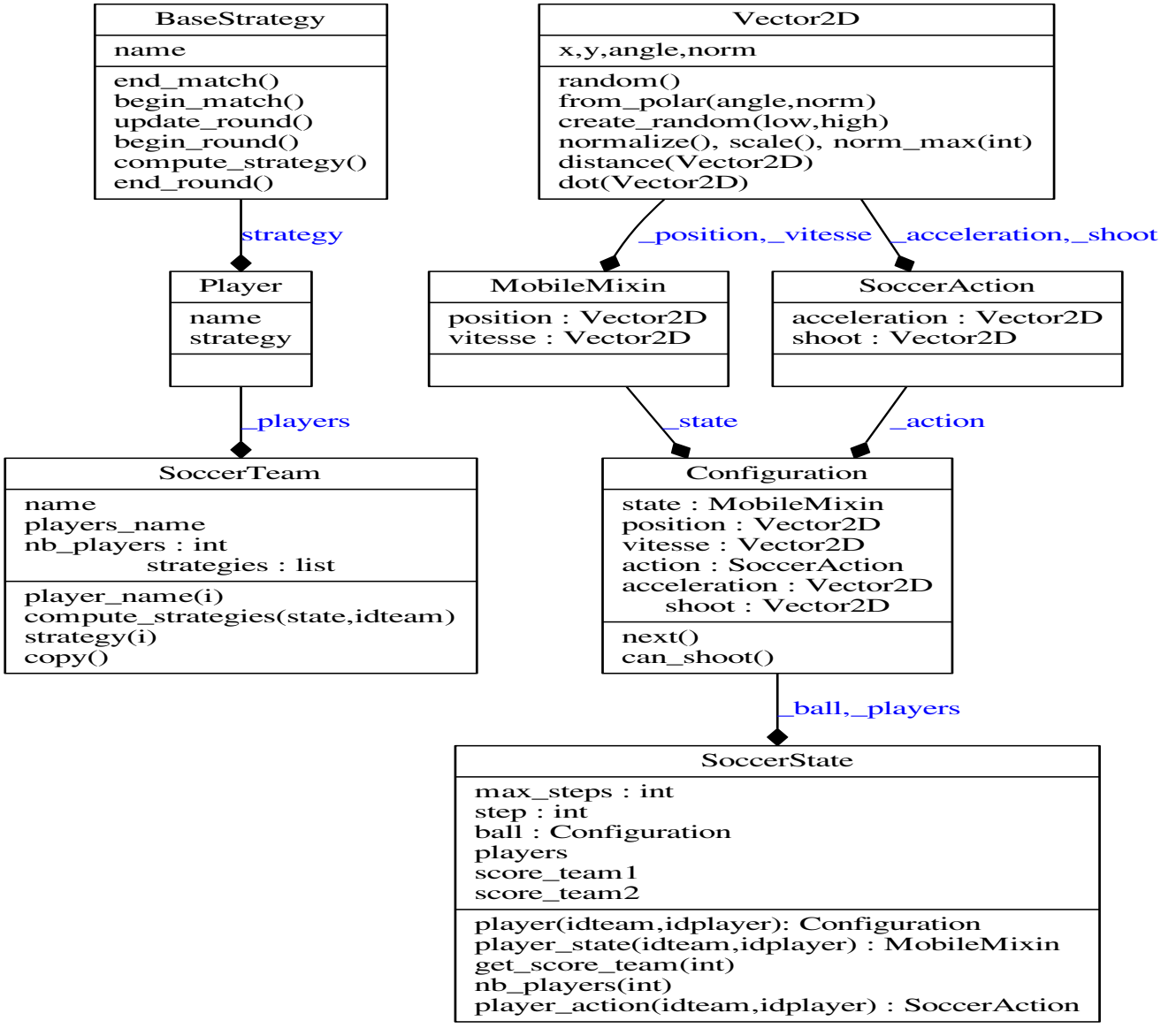
$$\begin{aligned} \overrightarrow{u_n} &= \begin{pmatrix} u_x \cos \pi/2 - u_y \sin \pi/2 \\ u_x \sin \pi/2 + u_y \cos \pi/2 \end{pmatrix} \\ \text{(cart.)} &= \begin{pmatrix} -u_y \\ u_x \end{pmatrix} \\ \text{(polaires)} &= (1, \theta + \pi/2) \end{aligned}$$

Plan

Les objets en présence (et leurs attributs)

- `Vector2D` : représente un point ou un vecteur (`x,y`);
- `MobileMixin` : représente un objet mobile (`position=Vector2D(),vitesse=Vector2D()`);
- `SoccerAction` : représente l'action d'un joueur (`acceleration=Vector2D(),shoot=Vector2D()`)
- `Player` : représente un joueur (`name="",strategy`);
- `Strategy` : représente une stratégie (fonction `compute_strategy(id_iteam,id_player)`)
- `SoccerState` :représente un état (`ball`,`player_state(id_team,id_player)`)
- `SoccerTeam` : liste de joueurs
- `SoccerMatch` : un match

Les objets en présence (et leurs attributs)

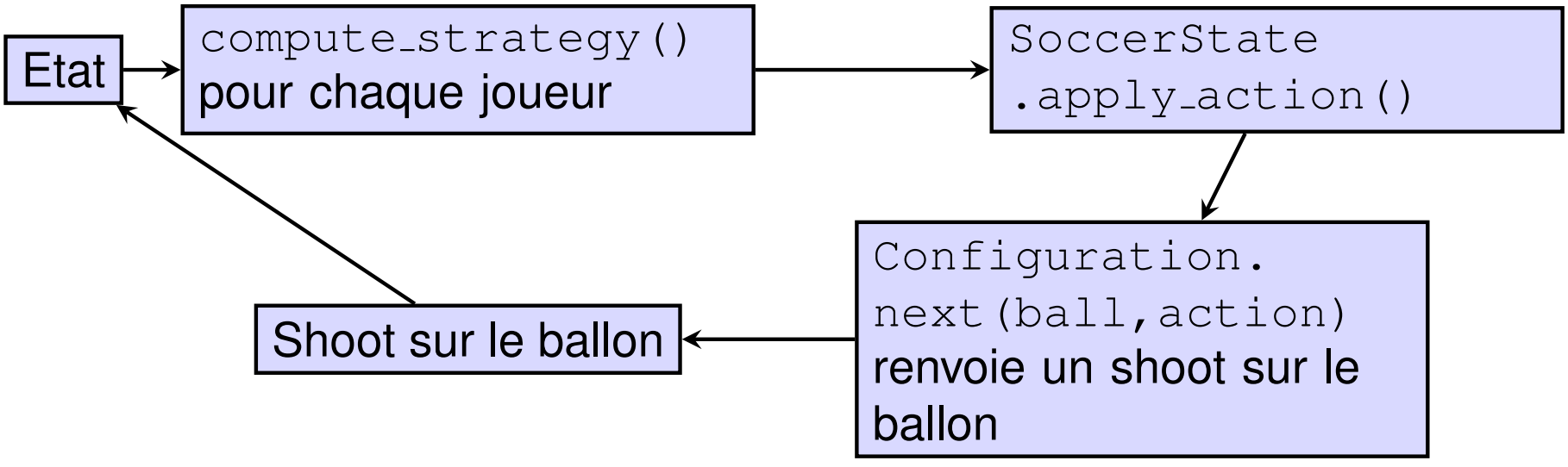


Les commandes utiles

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):
    # Calculer la nouvelle vitesse, freinage + acceleration
    self._state.vitesse *= (1 - settings.playerBrackConstant)
    self._state.vitesse += self.acceleration
    self._state.vitesse=self._state.vitesse.norm_max(maxPlayerSpeed)
    #Maj de la position
    self._state.position += self.vitesse
    if self._state.position.x < 0 or self.position.x > settings.GAME_WID
    or self.position.y < 0 or self.position.y > settings.GAME_HEIGHT:
    #repositionner le joueur
    #si le joueur ne shoot pas ou ne peut pas shooter, le shoot est nul
    if self._action.shoot.norm == 0 or not self.can_shoot():
        return Vector2D()
    if self._state.position.distance(ball.position)
            > (settings.PLAYER_RADIUS + settings.BALL_RADIUS):
        return Vector2D()
    # on calcule une penalite en fonction de l'angle vitesse joueur
    #/vitesse ballon et la distance au ballon
    angle_factor = 1.-abs(math.cos((self.vitesse.angle-self.shoot.angle)
    dist_factor = 1. - self._state.position.distance(ball.position) / (
        settings.PLAYER_RADIUS + settings.BALL_RADIUS)
    shoot = self.shoot * (1 - angle_factor * 0.25 - dist_factor * 0.25)
    shoot.angle += (2*random.random()-1.)*(
        angle_factor+dist_factor)/2.*shootRandomAngle*math.pi/2.
    return shoot
```

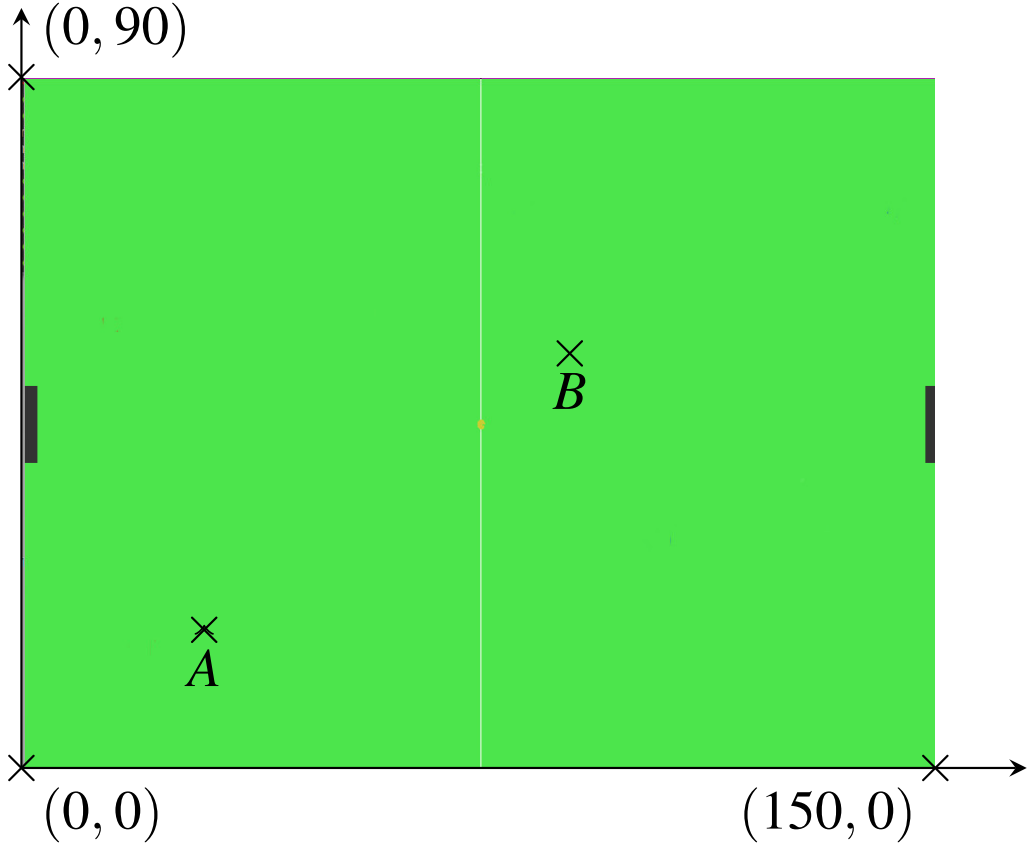
Le moteur du ballon

```
def apply_actions(self, actions=None):
  sum_of_shoots = Vector2D()
  for k, c in self._configs.items():
      if k in actions:
  sum_of_shoots += c.next(self.ball, actions[k])
  self.ball.vitesse.norm += -ballBrakeSquare*self.ball.vitesse.norm**2
                            - ballBrakeConstant * self.ball.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(self.ball.vitesse.dot(u_s))
      speed_ortho = self.ball.vitesse.dot(u_t)
      speed = Vector2D(speed_abs * u_s.x - speed_ortho * u_s.y,
                       speed_abs * u_s.y + speed_ortho * u_s.x)
      speed += sum_of_shoots
      self.ball.vitesse = speed
  self.ball.vitesse = self.ball.vitesse.norm_max(maxBallAcceleration)
  self.ball.position += self.ball.vitesse
  #corriger la position de la balle (rebondit contre les murs)
```

Plan

Aller vers un point ?

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

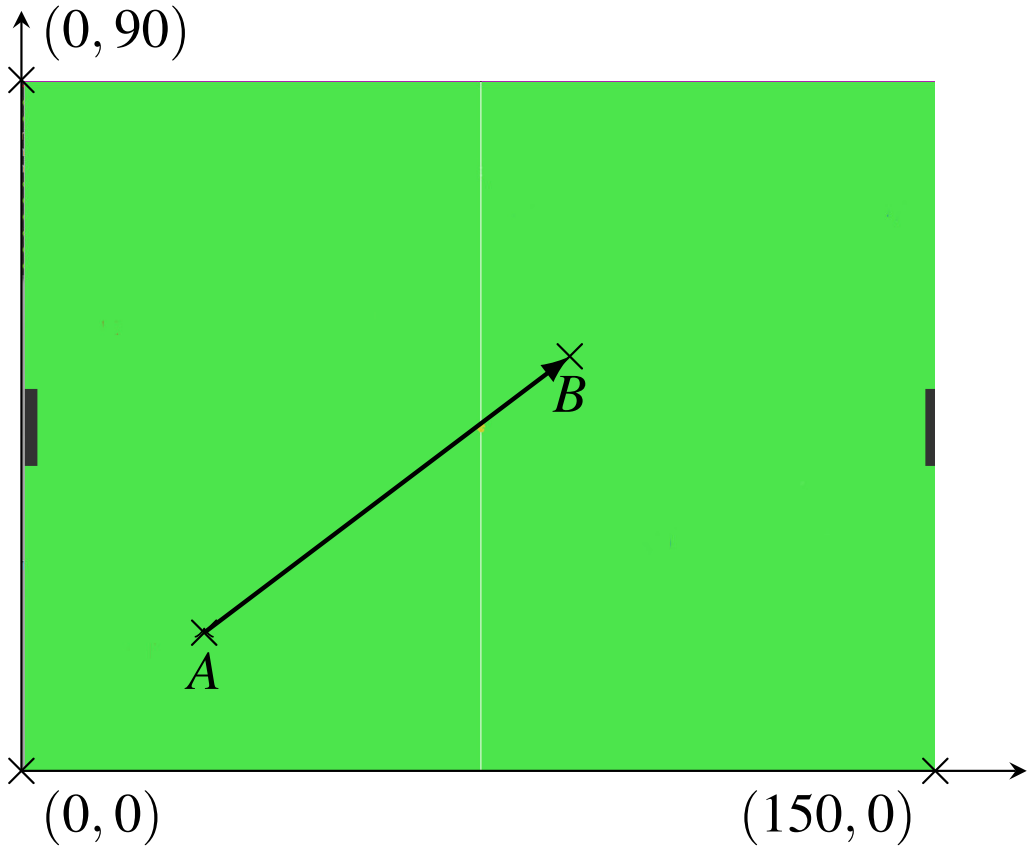


Aller vers un point ?

- 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 ?

$\Rightarrow$ 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.state.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.state.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(BaseStrategy):
      def __init__(self):
           BaseStrategy.__init__(self,"Ma strat")
      def compute_strategy(self,state,idteamn,idplayer):
          return MyState(state,idteam,idplayer).compute_strategy()
#ou
class SousStrat(BaseStrategy):
    def __init__(self,sous_strat):
        BaseStrategy.__init__(self,sous_strat.__name__)
        self.strat=sous_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)
FonceurStrat = SousStrat(fonceur)
```