

2I013 Groupe 5

Projet Robotique Cours 2 Géométrie Modélisation de l'arène - MVC¹

Nicolas Baskiotis

`nicolas.baskiotis@lip6.fr`

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

S2 (2018-2019)

¹Fortement inspiré du cours de V. Guigue

Etat des lieux

N'oubliez pas !!

Principe Agile : au plus simple et démo à chaque sprint !

- On veut voir le projet avancé, pas attendre 1 mois pour savoir où vous en êtes
 - Programmation itérative :
 - on développe des petites fonctionnalités
 - on ajoute au fur et à mesure du projet ce dont on a besoin
- ⇒ oui, cela revient à repasser n fois sur le même code mais seulement le bout de code qui le nécessite
- plus économe que de prévoir trop grand, trop complexe.
 - "Voir" le projet ⇒ le plus urgent ?

⇒ sortie texte (bof) ou graphique !

Etat des lieux

N'oubliez pas !!

Principe Agile : au plus simple et démo à chaque sprint !

- On veut voir le projet avancé, pas attendre 1 mois pour savoir où vous en êtes
 - Programmation itérative :
 - on développe des petites fonctionnalités
 - on ajoute au fur et à mesure du projet ce dont on a besoin
- ⇒ oui, cela revient à repasser n fois sur le même code mais seulement le bout de code qui le nécessite
- plus économe que de prévoir trop grand, trop complexe.

Etat des lieux

N'oubliez pas !!

Principe Agile : au plus simple et démo à chaque sprint !

- On veut voir le projet avancé, pas attendre 1 mois pour savoir où vous en êtes
 - Programmation itérative :
 - on développe des petites fonctionnalités
 - on ajoute au fur et à mesure du projet ce dont on a besoin
 - ⇒ oui, cela revient à repasser n fois sur le même code mais seulement le bout de code qui le nécessite
 - plus économe que de prévoir trop grand, trop complexe.
 - "Voir" le projet ⇒ le plus urgent ?
- ⇒ sortie texte (bof) ou graphique !

Modélisation arène, robot et physique

Robot

- A priori, le robot est simple mais quand même complexe à modéliser (formes, capteurs, roues, ...)
- Simplifier au maximum au début !

⇒ Rectangle qui se déplace

Arène

- Beaucoup d'éléments peuvent être intégrés, réfléchir au strict nécessaire !

⇒ Sol, murs, ...

Physique

- On peut simuler les frottements, les roues non alignées, ...
- Mais dans un premier temps ... faire simple !

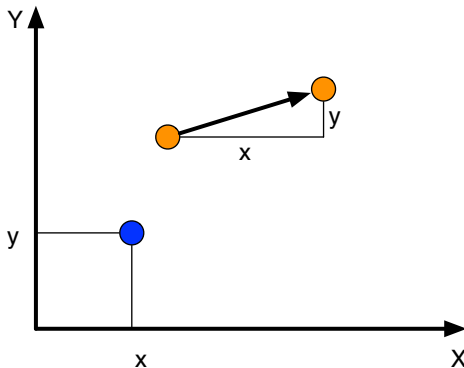
Plan

Géométrie

Exemple de senseur : senseur de distance

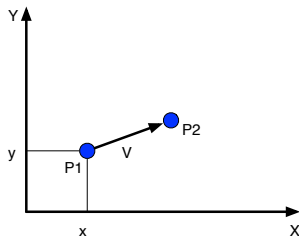
Model-View-Controller

Repérage dans l'espace 2D



- A l'aide de la classe `Point`
 - Attributs `double x` et `y`
- La même classe nous permet de gérer les points et les vecteurs

Gestion des déplacements

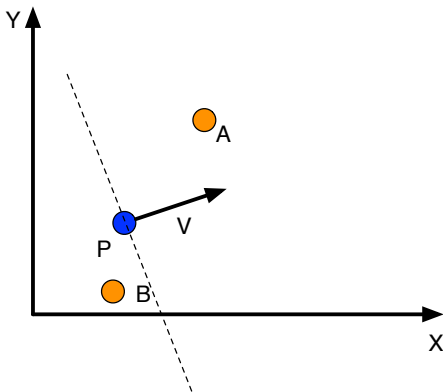


- Déplacements discrets (P: position, V: vitesse):

$$P_2 = P_1 + V, \quad \begin{cases} P_2.x = P_1.x + V.x \\ P_2.y = P_1.y + V.y \end{cases}$$

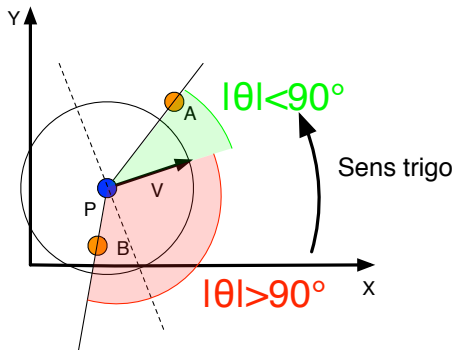
- En physique: $\vec{v} = \dot{\vec{x}} \approx \frac{\vec{x}_{t+1} - \vec{x}_t}{\delta_t}$, pour nous : $\delta_t = 1$ (unité arbitraire)
- En utilisant des vecteurs suffisamment petits: modélisation d'un déplacement continu

Se repérer dans l'espace



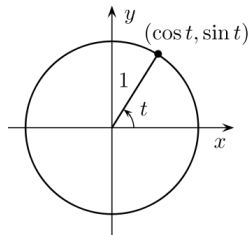
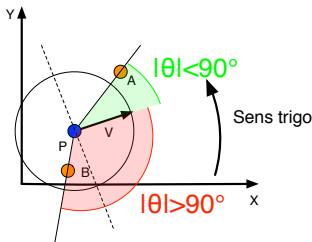
- Un objet est caractérisé par sa position P et sa vitesse V
- Qu'est ce qui est devant, qu'est ce qui est derrière l'objet?
- Qu'est ce qui est à droite, qu'est ce qui est à gauche?

Solution complète: calcul d'angle



- Devant/Derrière: calculer les angles $\widehat{V, PA}$ et $\widehat{V, PB}$
- Gauche/Droite: calculer les angles $\widehat{V, PA}$ et $\widehat{V, PB}$ avec le signe !

Détail du calcul d'angle

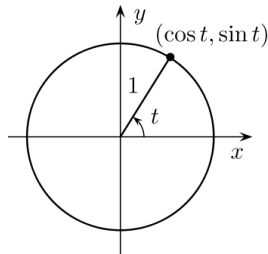
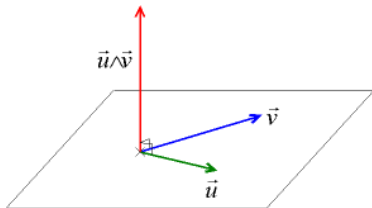


- Produit scalaire $U \cdot V = \langle U, V \rangle = \|U\| \|V\| \cos(\widehat{U, V}) = U_x V_x + U_y V_y$
- Corollaire:

$$\widehat{U, V} = \arccos \left(\frac{U \cdot V}{\|U\| \|V\|} \right)$$

- Attention: $\widehat{U, V} \in [0, \pi]$, pas de signe ici...
- Le signe de $U \cdot V$ permet de résoudre le pb devant/derrière

Détail du calcul d'angle (suite)



- Produit vectoriel: $\|U \wedge V\| = \|U\| \|V\| \sin(\widehat{U, V})$
- Corollaire:

$$\widehat{U, V} = \arcsin \left(\frac{\|U \wedge V\|}{\|U\| \|V\|} \right)$$

$$U \wedge V = \begin{bmatrix} u_2 v_3 - u_3 v_2 \\ u_3 v_1 - u_1 v_3 \\ u_1 v_2 - u_2 v_1 \end{bmatrix}$$

- L'étude de $u_1 v_2 - u_2 v_1$ permet de connaître le signe de l'angle...

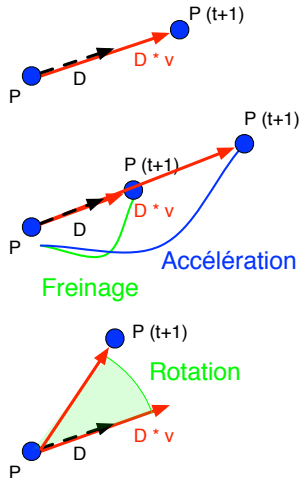
Pré-définition du robot

Le robot peut être défini géométriquement par:

- Sa position: P
- Sa direction (vecteur unitaire) : D
Conservation de la direction même à l'arrêt
- Sa vitesse (scalaire): $v \in [0, v_{max}]$

La commande du robot peut être sur 2 axes:

- Accélération/Freinage: modification de v
- Commande de direction: modification de D



Rotation sur un vecteur

Définition de la rotation d'un vecteur:

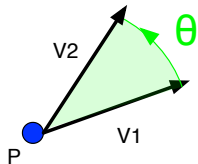
Soit un vecteur $V = \begin{bmatrix} v_x \\ v_y \end{bmatrix}$, la rotation d'angle θ est obtenu en utilisant la matrice de rotation:

$$R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}, \quad V' = RV$$

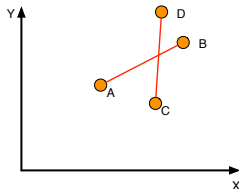
C'est à dire en utilisant la mise à jour:

- $v'_x = v_x \cos(\theta) - v_y \sin(\theta)$
- $v'_y = v_x \sin(\theta) + v_y \cos(\theta)$

ATTENTION à ne pas modifier v_x
avant la seconde ligne



Approfondissement: collisions



(Problématique de base
dans les cartes
graphiques/moteur
physique)

Comment détecter la collision de deux
vecteurs?

- Si C et D sont à gauche et à droite
de AB
- ET que A et B sont à gauche et à
droite de CD

Résultat:

S'ils sont de part et d'autre, l'un des produit vectoriel est positif,
l'autre négatif...

$$(AB \wedge AC)(AB \wedge AD) < 0 \text{ ET } (CD \wedge CA)(CD \wedge CB) < 0$$

Résumé des méthodes possibles de la classe Vecteur

- Addition, soustraction
 - génération d'un nouveau vecteur
 - auto-opérateur
- produit scalaire
- produit vectoriel (composante en z)
- multiplication par un scalaire
- rotation
- calcul de la norme
- clonage
- test d'égalité (structurelle)

Plan

Géométrie

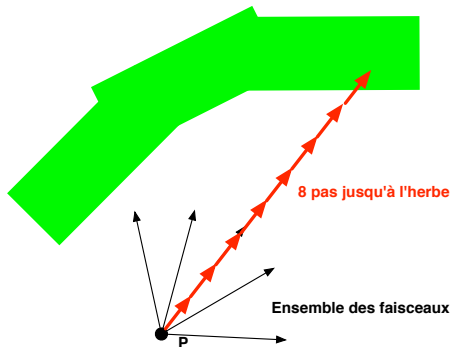
Exemple de senseur : senseur de distance

Model-View-Controller

Senseur de distance

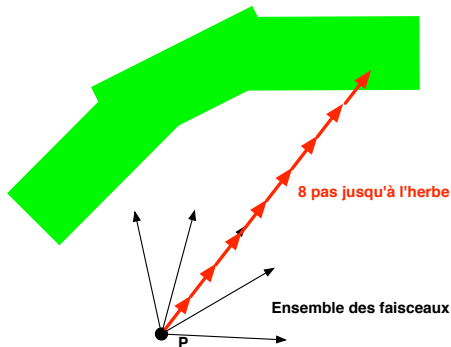
- Permet de savoir la distance (bruitée) d'un obstacle devant le robot.
- Comment le faire algorithmiquement ?

- Partir du robot
- Dans la **direction D** devant le robot
- Avancer d'un pas EPS
- Recommencer tant qu'il n'y a pas d'obstacle
- Renvoyer le nombre de pas effectués



Senseur de distance

- Permet de savoir la distance (bruitée) d'un obstacle devant le robot.
 - Comment le faire algorithmiquement ?
-
- Partir du robot
 - Dans la **direction D** devant le robot
 - Avancer d'un pas EPS
 - Recommencer tant qu'il n'y a pas d'obstacle
 - Renvoyer le nombre de pas effectués



Le problème du bruit

- Dans le modèle
- Dans les données des capteurs

⇒ modélisation aléatoire

2 classes intéressantes

- `random`
 - bruit uniforme, tirage d'entier, tirage gaussien
- `numpy.random`
 - Gestion de matrice aléatoire
 - Tirage selon la plupart des lois usuelles

Plan

Géométrie

Exemple de senseur : senseur de distance

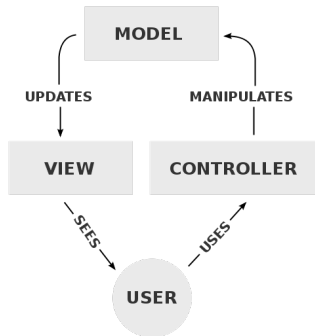
Model-View-Controller

Lasagne, spaghetti ou ravioli ?

- Spaghetti :
 - Code imbriqué, compliqué, pas (ou très peu) de classes
 - méthodes longues, et trop complexes.
 - ⇒ le pire ! à éviter
- Lasagne :
 - Code structuré, beaucoup de différentes classes bien réparties
 - mais changer une couche nécessite de changer toutes les couches (souvent le problème de trop de petites classes).
 - ⇒ Trop d'interdépendances, peu mieux faire
- Ravioli :
 - Bonne séparation des classes et des concepts, peut être réutilisable
 - chacun stand-alone, on peut changer indépendamment chaque classe
 - ⇒ Bien mais attention aux foisonnements de classes, il peut être difficile de comprendre les interactions.

- On veut un système **indépendant**
 - Pouvoir le **brancher ou le débrancher** facilement, sans intervenir dans le code de la simulation
 - Pouvoir changer de système de visualisation
- Il existe un modèle standard (assez lourd) pour gérer cette situation: **MVC Model View Controller**

- Une vue générique est élément capable de gérer la vue d'un objet.
- Pour chaque élément (robot, arène,...), un objet vue est créé.
- Lorsque le modèle change il informe le Contrôleur (événement)
- Le Contrôleur met à jour les informations d'affichage
- Ces dernières étapes peuvent se faire par le contrôleur.



credit: wikipedia

Principe général

Événement vs programmation linéaire

Différents éléments réagissent les uns par rapport aux autres: ils **émettent** des **événements** et **écoutent** ce qui se passe autour d'eux.

Principe: chaque composant doit être indépendant, le main fait le lien entre certains émetteurs et certains récepteurs.

Exemple:

1. Le robot bouge, le modèle doit envoyer un **signal: *update***
2. Si la vue est **branchée**, elle **reçoit le message** et procède à une mise à jour

Emetteur/récepteur

- **Émetteur:** le modèle émet un évènement lorsqu'une mise à jour de l'affichage est nécessaire
 - L'émission revient à un appel de méthode dans le récepteur
 - L'émetteur doit connaître/stocker les récepteurs pour pouvoir leur envoyer le message
 - Les récepteurs doivent répondre à un cahier des charges: on doit être sur que la méthode de réception est implémentée...
- **Récepteur:** le **Contrôleur** reçoit le message à travers un appel à une méthode. Il met à jour le modèle.
 - L'observateur fait le tampon entre le modèle et l'affichage, c'est le garant de l'indépendance

Séquence de communication

ACTE 0: Le main fait le lien entre la Vue, le Contrôleur et le Modèle physique

ACTE 1: Emission/Diffusion

- La simulation décide d'envoyer un message
appel à `void update()`

```
def update():  
    for listener in listeners:  
        listener.manageUpdate();
```

ACTE 2: Reception

Les méthodes `manageUpdate()` des récepteurs sont invoquées

ACTE 3: Mise à jour de la vue

- Le controleur/vue a une liste des observeurs :

```
def manageUpdate():  
    for observer in observers:  
        o.doSomething()
```