Projet de recherche de stage de seconde 2026: Le Python Gourmand, introduction à la cinématique inverse

Nicolas Pronost, Nathan Salazar
Laboratoire LIRIS, équipe SAARA

1 Introduction

1.1 Objectif

Le problème de la cinématique inverse est un problème du domaine de la robotique et de l'animation par ordinateur. Pour le comprendre définissons d'abord celui de la cinématique dite directe. Lorsque l'on veut définir la pose d'un robot ou d'un personnage virtuel nous devons donner des angles à ses articulations. Ces angles définissent la pose et grâce à ces angles et les distances entre les articulations nous pouvons calculer la position de chacune des articulations. Ce procédé consistant à calculer les positions à partir des angles s'appelle la cinématique directe. La cinématique inverse est le procédé inverse, c'est-à-dire celui qui consiste à calculer les angles en fonction des positions des articulations. Le problème devient intéressant lorsque nous voulons résoudre le problème quand nous n'avons pas toutes les positions des articulations, mais typiquement qu'une seule: la dernière. Le problème consiste alors à trouver la pose du robot ou du personnage de telle façon qu'il puisse atteindre une cible donnée (par exemple pour attraper ou toucher un objet).

Dans ce stage, vous étudierez, implémenterez et comparerez différentes méthodes de calcul pour le problème de la cinématique inverse, et ce pour plusieurs configurations d'articulations. Nous nous limiterons à des configurations à une, deux et trois articulations possédant un dégrée de liberté chacun (une seule rotation le long d'un seul axe) et dans un environnement à deux dimensions. Le personnage sera un serpent (un python, comme le langage ;-)) qui devra atteindre une pomme avec sa bouche (voir Figure 1).

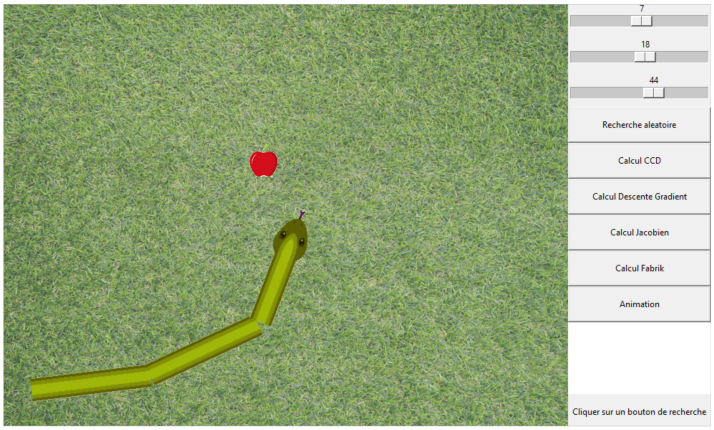


Figure 1: Visualisation d'une configuration pour le problème de cinématique inverse : un serpent composé de plusieurs parties doit manger une pomme.

1.2 Environnement de développement

L'implémentation informatique des méthodes de cinématique inverse sera réalisée en Python. Une base de code est fournie avec le sujet. L'interface graphique est réalisée avec Tkinter qui est normalement installé avec toute distribution Python (choisir d'installer tcl/tk lors de l'installation de Python). La gestion des images est par contre réalisée avec le module pillow et peut s'installer grâce à la commande: `pip install pillow`.

Télécharger la base de code du stage. Vous y trouverez les images nécessaires à l'exécution de l'application, au format png (pomme, herbe et différentes parties du serpent), ainsi que les scripts Python. Le script IKImage.py contient les fonctionnalités de base pour gérer les images, placer les parties du serpent, changer les angles etc. (regarder juste les fonctionnalités présentes, vous n'avez pas besoin de comprendre tout le code). Les scripts Serpent1DDL.py, Serpent2DDL.py et Serpent3DDL.py sont les scripts à compléter. Vous pouvez changer les valeurs des angles grâce aux sliders sur la droite de la fenêtre de l'application. Vous pouvez lancer les différentes méthodes de calcul grâce aux boutons en dessous. Vous pouvez lancer l'animation grâce au bouton encore en dessous. Et vous pouvez visualiser les temps d'exécution dans l'encadré tout en bas.

- Pour des tutoriels sur le langage Python: https://courspython.com/
- La documentation officielle de Python: https://docs.python.org/3/
- La documentation du module TKinter : https://docs.python.org/3/library/tkinter.html

2 Serpent à un degré de liberté (1DDL)

2.1 Résolution analytique

Considérons la configuration la plus simple en premier. Le ≪robot≫ ou ≪personnage≫ dont on veut déterminer la pose est un serpent qui peut tourner uniquement autour de son extrémité gauche (centre de rotation). La tête du serpent (sa bouche), à droite, doit se positionner, par rotation autour de son centre de rotation à gauche, le plus près possible d'une cible donnée, symbolisée par une pomme (voir Figure 2).

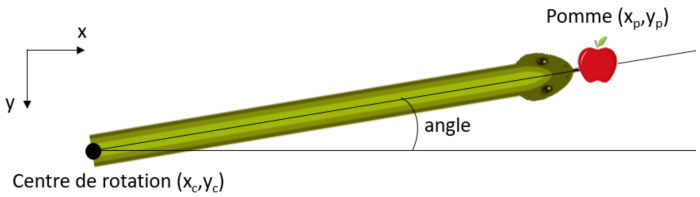


Figure 2: Problème de cinématique inverse avec 1 degré de liberté

✎ Résoudre ce problème sur papier, c'est-à-dire répondre à la question: quel angle permet au serpent de se diriger dans la bonne direction (celle de la pomme)?
Indication: essayer de visualiser un cercle trigonométrique sur la Figure 2.

⌨ Implémenter la formule de résolution de ce problème dans la fonction `calcul_analytique` de la classe Serpent1DDL et la tester (appuyer sur le bouton correspondant et regarder la position de la tête du serpent et le temps d'exécution).

2.2 Résolution aléatoire

Une méthode très simple ne faisant pas intervenir de formule trigonométrique consiste à simplement tirer aléatoirement un angle pour le centre de rotation, l'appliquer, et de regarder si la tête est suff-

isamment proche de la pomme. Si oui c'est fini et on a trouvé une solution au problème et sinon on recommence.

⌨ Implémenter cette méthode dans la fonction `cherche_aleatoire` de la classe Serpent1DDL et la tester.

✎ Quels sont les avantages et inconvénients de ces deux méthodes de résolution (aléatoire et analytique) ? Quelle est celle qui obtient la solution la plus précise ? Quelle est la plus rapide à exécuter ?

3 Serpent à deux degrés de liberté (2DDL)

Considérons maintenant une configuration un peu plus compliquée. Le serpent est constitué de deux parties de taille égale et de deux degrés de liberté, un en position P_1 (ne bougeant jamais) et un en position P_2. Positionner correctement le serpent, pour qu'il atteigne la pomme avec sa bouche, consiste alors à calculer deux angles α_1 et α_2 (voir Figure 3).

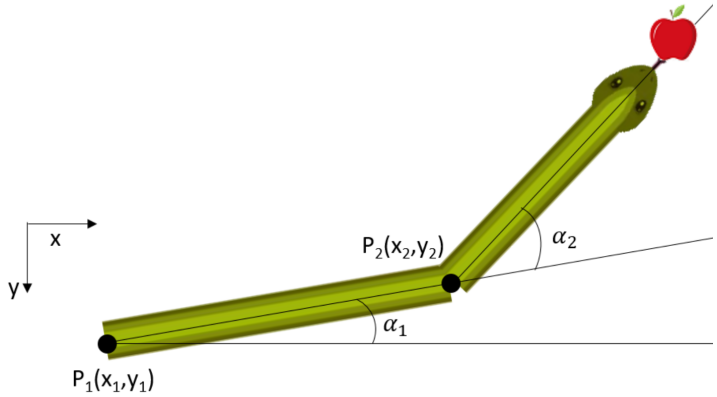


Figure 3: Problème de cinématique inverse avec 2 degrés de liberté

3.1 Résolution aléatoire

De la même manière que pour le serpent à 1 degré de liberté, il est tout à fait possible de tirer aléatoirement deux angles différents et de les essayer.

⌨ Implémenter cette méthode dans la fonction `cherche_aleatoire` de la classe Serpent2DDL et la tester.

✎ Est-elle plus ou est-elle moins rapide en moyenne que sur le serpent à 1 degré de liberté? De beaucoup? A votre avis pourquoi?

3.2 Résolution analytique

Nous allons maintenant essayer de calculer, directement, les valeurs des angles α_1 et α_2 en fonction des données d'entrée à notre disposition c'est-à-dire la position de la pomme, la position du centre

de rotation P_1 et la longueur des deux segments du serpent que nous nommerons L_1 et L_2 (250 pixels pour les deux dans notre exemple).

✎ Ce problème a combien de solutions ? Est-ce qu'il y a des configurations pour lesquelles le nombre de solutions est différent ? Faire des schémas simplifiés (points et lignes) qui illustrent les différentes configurations et les différentes solutions (ou absence de solution).

Consulter l'annexe A pour obtenir les formules de calcul des angles (essayer de comprendre l'idée générale, les détails des calculs sont un peu complexes et vous n'avez pas besoin de les retenir).

⌨ Implémenter ces formules dans la fonction `calcul_analytique` de la classe Serpent2DDL et la tester, en particulier avec différentes positions de la pomme et différentes solutions au problème lorsqu'il y en a plusieurs.

✎ La solution est-elle toujours ≪exacte≫? Noter les temps de calcul et comparer les avec les configurations et méthodes précédentes.

4 Serpent à trois degrés de liberté (3DDL)

Nous allons maintenant nous intéresser à la troisième et dernière configuration, un serpent en trois parties avec donc trois degrés de liberté. Nous noterons P_1 (qui ne bouge jamais), P_2 et P_3 les positions des trois centres de rotation du serpent. Positionner correctement le serpent, pour qu'il atteigne la pomme avec sa bouche, consiste à calculer les trois angles α_1, α_2 et α_3 (voir Figure 4).

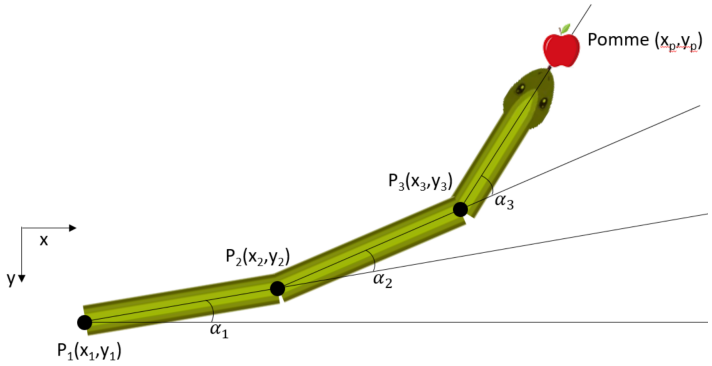


Figure 4: Problème de cinématique inverse avec 3 degrés de liberté

4.1 Résolution aléatoire

Toujours de la même manière, commençons par essayer de tirer aléatoirement trois angles différents et essayons les.

⌨ Implémenter cette méthode dans la fonction `cherche_aleatoire` de la classe Serpent3DDL et la tester.

✎ Reporter les temps de calculs nécessaires à l'obtention d'une solution. Que pouvez-vous en conclure sur cette méthode, en particulier alors que le nombre de degrés de liberté pourrait encore augmenter ?

4.2 Résolution itérative

Puisque nous avons 3 inconnues (α_1, α_2 et α_3) et uniquement 2 contraintes/équations (la position de la pomme x_p et y_p), il existe une infinité de solutions (lorsque la position est atteignable) et donc nous ne pouvons pas (facilement) calculer cet ensemble de solutions de manière analytique. Nous allons devoir trouver une solution approximative, en ≪tâtonnant≫, par essais successifs, c'est-à-dire par itération, d'où le nom de ces méthodes. Puisqu'il y a une infinité de solutions et que nous devons en choisir une seule, il y a un très grand nombre de méthodes différentes possibles qui trouvent des solutions différentes. Nous allons voir quatre méthodes itératives différentes.

4.2.1 Méthode de Cyclic Coordinate Descent (CCD)

Le principe de cette méthode consiste à essayer, successivement, de rapprocher la bouche du serpent le plus possible de la pomme pour chaque centre de rotation du serpent. Comme une rotation entraine un changement des positions des centres de rotation suivants, nous réalisons ces rotations une par une en commençant par le premier centre (le plus à gauche). Nous calculons l'angle $\Delta\alpha$ entre la direction du centre de rotation vers la bouche du serpent et la direction du centre de rotation vers la pomme. Cet angle est appliqué pour tourner le serpent vers la pomme (sans changer les autres angles): $\alpha \leftarrow \alpha + \Delta\alpha$. Puis on fait la même chose pour le deuxième centre de rotation (application de l'angle entre la bouche et la pomme) et finalement pour le troisième centre de rotation (voir Figure 5). Ensuite, nous recommençons à nouveau à partir du premier centre de rotation jusqu'à ce que la bouche soit suffisamment proche de la pomme (ou que l'on a fait trop d'itérations).

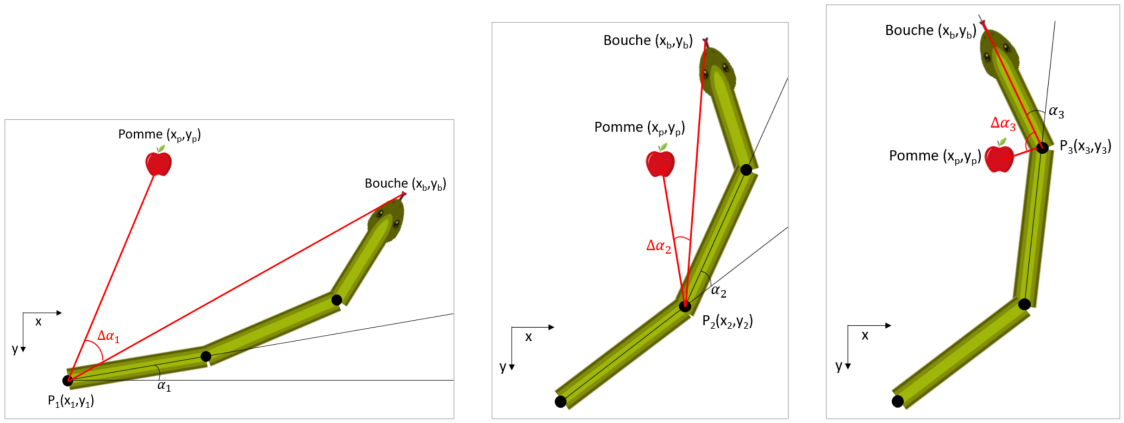


Figure 5: Illustration d'une itération de la méthode de CCD. Les différences d'orientations $\Delta\alpha_1$, $\Delta\alpha_2$ et $\Delta\alpha_3$ sont calculées et appliquées respectivement sur les angles α_1, α_2 et α_3 jusqu'à obtention d'une solution.

⌨ Implémenter cette méthode dans la fonction `cherche_CCD` de la classe Serpent3DDL et la tester avec différentes positions de la pomme.

✎ Reporter les temps de calculs nécessaires à l'obtention d'une solution. La méthode arrive-t-elle toujours à trouver une solution, même lorsqu'elle existe ? Est-ce qu'il y a des positions de la pomme plus difficiles à atteindre que d'autres ? Est-ce que la méthode converge rapidement et de manière ≪fluide≫ à une solution ? La pose obtenue est-elle ≪naturelle≫ ?

⌨ *[Optionnel]* Il est aussi possible d'itérer dans l'autre sens, en commençant par le troisième centre de rotation puis le deuxième puis le premier. Implémenter cette variante et la comparer à la précédente.

4.2.2 Méthode de descente de gradient

Le principe de cette méthode repose sur l'évaluation d'essais successifs de petites rotations autour de la pose actuelle. Si appliquer une petite variation dans la pose améliore la pose, c'est-à-dire approche la bouche de la pomme, alors cette variation est adoptée et à partir de cette nouvelle pose nous essayons de trouver une autre petite variation qui améliore encore plus. Ainsi, de proche en proche, nous pourrons converger vers une solution (bouche suffisamment proche de la pomme). Cette méthode s'appelle la méthode du gradient car c'est le nom que l'on donne à l'ensemble de ces variations des données (angles) qui permettent d'améliorer la solution (voir Figure 6).

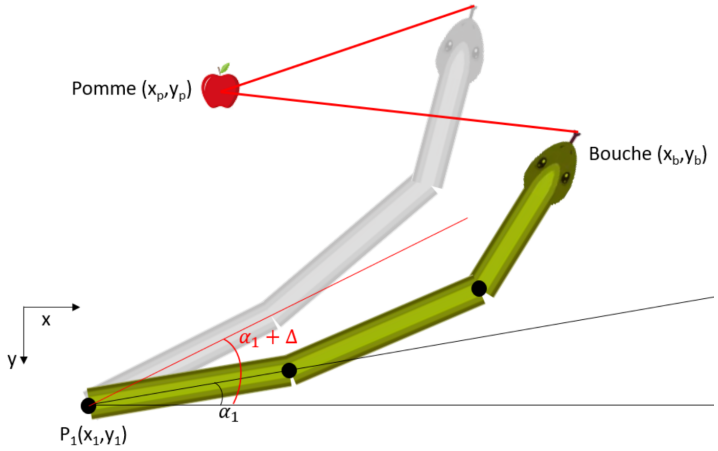


Figure 6: Illustration du principe de la descente de gradient. Si la bouche du serpent après application de l'angle $\alpha_1 + \Delta$ est plus proche de la pomme alors on garde cette solution. On fait de même pour les autres centres de rotation. Sinon, on tourne dans le sens inverse (si ça n'améliore pas ça empire et donc l'inverse améliorera).

Le pseudocode de cette méthode est le suivant:

Algorithm 1 Calcul des angles du serpent par la méthode de descente de gradient

```
Tant que la bouche n'est pas suffisamment proche de la pomme et le nombre d'itérations max n'est pas atteint faire
    incrément du nombre d'itérations
    Pour chaque centre de rotation i faire
        distanceInitiale ← distance entre la bouche et la pomme
        ajout de Δ à l'angle du centre de rotation i
        distanceDelta ← distance entre la bouche et la pomme
        retour à l'angle initial
        gradient_i ← (distanceDelta − distanceInitiale) / Δ
    Fin Pour
    Pour chaque centre de rotation i faire
        angle_i ← angle_i − β * gradient_i
        application de l'angle_i
    Fin Pour
    affichage de la pose
Fin Tant que
```

Vous pouvez vous apercevoir que tous les gradients sont d'abord calculés indépendamment les uns des autres, puis que tous les angles sont calculés de sorte qu'ils se dirigent dans la direction

opposée au gradient. En effet si le gradient est positif cela veut dire que la distance à la pomme est plus grande. Il faut donc changer l'angle dans l'autre sens (opposé). Vous remarquez aussi que l'on multiplie le gradient par une valeur $\beta \; \epsilon \; [0, 1]$. En effet, nous voulons réaliser de ≪petites≫ variations d'angle afin de ne pas corriger l'angle trop et dépasser notre cible et aussi afin que l'influence de la rotation d'un des centres de rotation n'influe pas trop sur le calcul des autres.

⌨ Implémenter ce pseudocode dans la fonction `cherche_Gradient` de la classe Serpent3DDL et la tester avec différentes positions de la pomme.

✎ Reporter les temps de calculs nécessaires à l'obtention d'une solution. La méthode arrive-t-elle toujours à trouver une solution, même lorsqu'elle existe ? Est-ce qu'il y a des positions de la pomme plus difficiles à atteindre que d'autres ? Est-ce qu'elle converge rapidement et de manière ≪fluide≫ à une solution ? La pose obtenue est-elle ≪naturelle≫ ?

✎ Faire des expérimentations avec différentes valeurs de Δ et β. Vous pouvez reporter les résultats dans un tableau (par exemple sur les lignes les différentes valeurs Δ testées, sur les colonnes les valeurs de β, et dans les cases les temps de calculs).

Pour améliorer la convergence de cette méthode, nous pouvons, à la fin de chaque itération, modifier les valeurs de Δ et β. En effet si lorsque les itérations augmentent alors les deux valeurs diminuent, nous pourrons converger plus rapidement vers une solution.

✎ *[Optionnel]* Ajouter l'évolution des valeurs de Δ et β avec le nombre d'itérations et étudier si la convergence est différente.

4.2.3 Méthode du Jacobien

Dans la méthode précédente nous tournions autour des centres de rotation afin de constater l'amélioration, ou non, de la distance de la bouche du serpent à la pomme. Dans la méthode dite du Jacobien, nous allons calculer la relation entre la rotation autour d'un des centres de rotation et le déplacement de la bouche du serpent. Si la bouche, grâce à cette rotation, permet de déplacer la bouche dans la direction de la pomme alors nous appliquons cette rotation, sinon non. La quantité de rotation appliquée va directement dépendre de la quantité d'amélioration de la pose. Nous allons favoriser davantage les rotations qui améliorent le plus la pose. La méthode est dite du Jacobien car c'est le nom de l'entité mathématique qui décrit comment chaque rotation individuelle autour des centres de rotation va faire se déplacer notre point d'intérêt, ici la bouche du serpent. Consulter l'annexe B décrivant la construction du Jacobien de notre serpent (essayer de comprendre l'idée générale, les détails des calculs ne sont pas à retenir). Identifier les formules (en rouge) qui seront nécessaires à l'implémentation de cette méthode.

⌨ Implémenter le pseudocode de l'annexe B dans la fonction `cherche_Jacobien` de la classe Serpent3DDL et la tester avec différentes positions de la pomme et valeurs de γ.

✎ Reporter les temps de calculs nécessaires à l'obtention d'une solution. La méthode arrive-t-elle toujours à trouver une solution, même lorsqu'elle existe ? Est-ce qu'il y a des positions de la pomme plus difficiles à atteindre que d'autres ? Est-ce qu'elle converge rapidement et de manière ≪fluide≫ à une solution ? La pose obtenue est-elle ≪naturelle≫ ?

4.2.4 Méthode FABRIK

Dans les méthodes précédentes nous effectuions tous les calculs grâce à des rotations autour des centres. Dans la méthode FABRIK, les positions des centres sont modifiées afin de résoudre le problème (faire approcher la bouche du serpent le plus près possible de la pomme) tout en respectant les contraintes de distance fixe entre les centres de rotation. La méthode est toujours une méthode itérative et donc se répète jusqu'à ce que l'on ait trouvé une solution correcte (ou bien que le nombre d'itérations maximales ait été atteint). A chaque itération nous allons parcourir le serpent deux fois: une première fois depuis la bouche jusqu'au premier centre de rotation puis une deuxième fois dans l'autre sens du premier centre jusqu'à la bouche. Pour chaque centre de rotation nous allons le déplacer afin de se diriger vers la pomme, tout en respectant les distances entre les centres (voir Annexe C). Le pseudocode de cette méthode est le suivant:

Algorithm 2 Calcul des angles du serpent par la méthode FABRIK

Tant que la bouche n'est pas suffisamment proche de la pomme et qu'on n'a pas atteint le nombre d'itérations max faire
    incrément du nombre d'itérations
    $\overrightarrow{Pomme\ P_3} = (x_{P_3} - x_p,\ y_{P_3} - y_p)$
    $P_3 = Pomme$ déplacé dans la direction $\overrightarrow{Pomme\ P_3}$ de la taille de la dernière partie du serpent
    $\overrightarrow{P_3P_2} = (x_{P_2} - x_{P_3},\ y_{P_2} - y_{P_3})$
    $P_2 = P_3$ déplacé dans la direction $\overrightarrow{P_3P_2}$ de la taille de la deuxième partie du serpent
    $\overrightarrow{P_1P_2} = (x_{P_2} - x_{P_1},\ y_{P_2} - y_{P_1})$
    $P_2 = P_1$ déplacé dans la direction $\overrightarrow{P_1P_2}$ de la taille de la première partie du serpent
    $\overrightarrow{P_2P_3} = (x_{P_3} - x_{P_2},\ y_{P_3} - y_{P_2})$
    $P_3 = P_2$ déplacé dans la direction $\overrightarrow{P_2P_3}$ de la taille de la deuxième partie du serpent
    $\overrightarrow{P_3\ Pomme} = (x_{P_3} - x_p,\ y_{P_3} - y_p)$
    $Bouche = P_3$ déplacé dans la direction $\overrightarrow{P_3\ Pomme}$ de la taille de la troisième partie du serpent
    calcul de l'angle α_1 entre les directions $\vec{x}$ et $\overrightarrow{P_1P_2}$
    calcul de l'angle α_2 entre les directions $\overrightarrow{P_1P_2}$ et $\overrightarrow{P_2P_3}$
    calcul de l'angle α_3 entre les directions $\overrightarrow{P_2P_3}$ et $\overrightarrow{P_3\ Bouche}$
    affichage de la pose
Fin Tant que

⌨ Implémenter ce pseudocode dans la fonction `cherche_FABRIK` de la classe Serpent3DDL et la tester avec différentes positions de la pomme.

✎ Reporter les temps de calculs nécessaires à l'obtention d'une solution. La méthode arrive-t-elle toujours à trouver une solution, même lorsqu'elle existe ? Est-ce qu'il y a des positions de la pomme plus difficiles à atteindre que d'autres ? Est-ce qu'elle converge rapidement et de manière ≪fluide≫ à une solution ? La pose obtenue est-elle ≪naturelle≫ ?

5 Application pour l'animation

Nous avons maintenant plusieurs méthodes de résolution du problème de cinématique inverse. Ces méthodes sont très utilisées dans les domaines de l'animation et de la robotique pour calculer des poses permettant de placer des extrémités de chaînes articulées (mains, pieds, pinces et autres instruments et outils) à des positions données. Nous allons tester les différentes méthodes implémentées dans un problème simple d'animation: nous allons déplacer la pomme dans le plan 2D et étudier les mouvements (poses successives) du serpent poursuivant cette pomme mouvante.

✎ Pour chacune des méthodes de résolution, étudier le mouvement du serpent lorsque vous placer vous-même la pomme à des positions successives proches. Quelle(s) méthode(s) donne(nt) les mouvements les plus ≪fluides≫, les plus ≪réalistes≫, les plus ≪précis≫ ?

Tester des positions successives de la cible à la main est un peu fastidieux. Nous allons donc programmer une animation de la pomme qui sera jouée automatiquement. Pour tester différentes configurations nous allons programmer une trajectoire qui part de la position de la bouche quand tous les angles sont à zéro puis qui suit de manière générale l'arc de cercle montant vers le coin en haut à gauche de la fenêtre mais tout en s'approchant et en s'éloignant de cet arc de cercle (voir Figure 7). Ainsi nous aurons régulièrement des positions inatteignables, des positions nécessitant un serpent tout droit et des positions demandant au serpent de se ≪ recroqueviller ≫ sur lui-même.

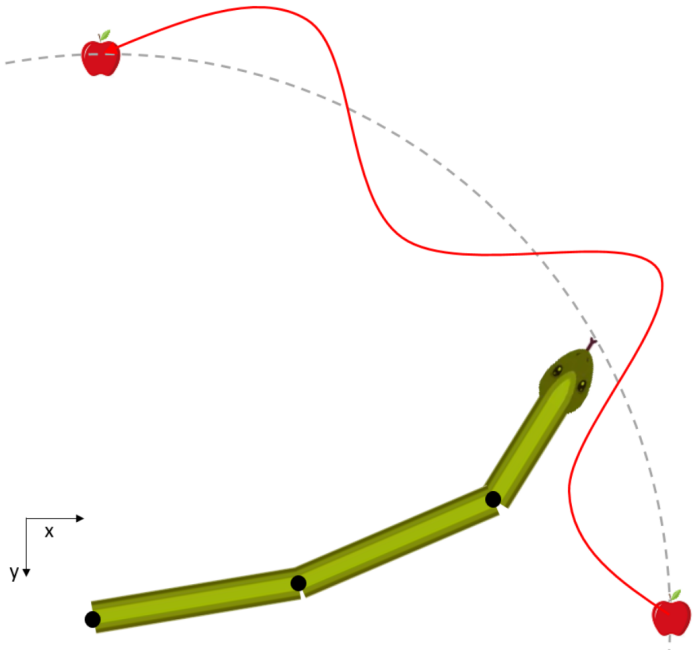


Figure 7: Illustration de la trajectoire de la pomme dans les tests d'animation. La trajectoire générale (en gris) suit un arc de cercle et la déviation à cette trajectoire générale oscille, grâce à une sinusoïde, autour de cet arc de cercle (en rouge).

⌨ Consulter la fonction `animation` des classes Serpent1DDL, Serpent2DDL et Serpent3DDL et la tester. La fonction est appelée en cliquant sur le bouton correspondant.

✎ Pour la version à 3DDL, tester les différentes méthodes de résolution, et reporter les temps de calculs, succès/échecs de la résolution, la qualité du mouvement et toutes autres mesures que vous jugerez intéressantes.

6 Ouverture vers d'autres méthodes

Il existe encore beaucoup d'autres méthodes de résolution du problème de la cinématique inverse. En fonction du temps restant à votre disposition, vous pouvez rechercher, implémenter et tester d'autres méthodes. Vous pouvez par exemple consulter ces articles scientifiques (en anglais) présentant un grand nombre de méthodes:

- Inverse kinematics techniques in computer graphics : a survey
- A comparative study of human inverse kinematics techniques for lower limbs

Les plus faciles à implémenter sont probablement les autres méthodes basées sur le Jacobien (pseudo-inverse, moindre carrés et décomposition SVD) ou bien les approches basées sur des données utilisant par exemple de l'apprentissage (Intelligence Artificielle).

Pour ceux intéressés par une méthode basée sur de l'apprentissage automatique, nous fournissons le script Python `Serpent1DDL_LR.py`. Dans ce script reprenant l'exercice 1, vous trouverez une base de code pour effectuer une régression linéaire et un modèle de machine learning apprenant automatiquement des paramètres pour prédire une fonction linéaire décrivant les données à partir d'exemples. Pour l'utiliser, il faudra installer les modules numpy et sklearn avec les commandes: `pip install numpy` et `pip install scikit-learn`.

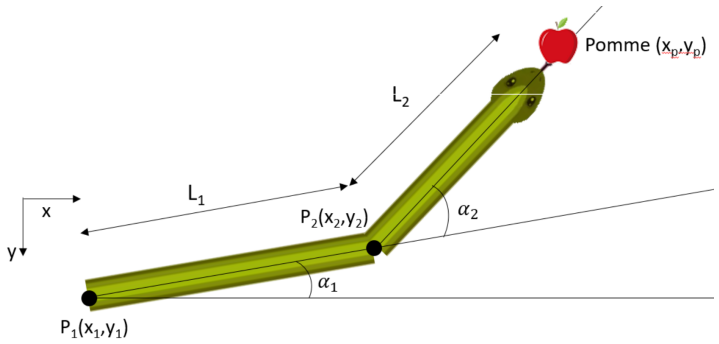
⌨ *[Optionnel]* Implémenter la régression linéaire dans la classe Serpent1DDL_LR. La fonction `generateDB` permet de générer une base de données d'angles valides pour atteindre la pomme et la fonction `inference` permet d'utiliser la régression linéaire pour prédire l'angle à appliquer pour atteindre la pomme. Les fonctions sont appelées en cliquant sur les boutons correspondants.

✎ *[Optionnel]* Tester différentes tailles de bases de données, reporter les temps de calculs, et succès/échecs de la résolution.

✎ *[Optionnel]* Tester et étudier les résultats de différentes modèles d'apprentissage disponibles dans le module scikit-learn, par exemple NearestNeighbors, SVR, et MLPRegressor.

A Solution analytique pour 2DDL

A.1 Définition du problème



On a les relations entre positions, distances et angles suivantes :

$$\begin{cases} x_2 = x_1 + L_1 \times \cos(\alpha_1) \\ y_2 = y_1 - L_1 \times \sin(\alpha_1) \end{cases} \quad \text{et} \quad \begin{cases} x_p = x_2 + L_2 \times \cos(\alpha_1 + \alpha_2) \\ y_p = y_2 - L_2 \times \sin(\alpha_1 + \alpha_2) \end{cases}$$

On connaît x_1, y_1, L_1, L_2, x_3, y_3 et on veut calculer α_1, α_2. On a un système à 2 équations et 2 inconnues.

A.2 Mise sous forme réduite

On considère le système équivalent suivant (substitution de x_2 et y_2) :

$$\begin{cases} x_p = x_1 + L_1 \times \cos(\alpha_1) + L_2 \times \cos(\alpha_1 + \alpha_2) \\ y_p = y_1 - L_1 \times \sin(\alpha_1) - L_2 \times \sin(\alpha_1 + \alpha_2) \end{cases}$$

On pose $E = x_p - x_1$ et $F = -(y_p - y_1)$ Le système devient :

$$\begin{cases} L_1 \times \cos(\alpha_1) + L_2 \times \cos(\alpha_1 + \alpha_2) = E \\ L_1 \times \sin(\alpha_1) + L_2 \times \sin(\alpha_1 + \alpha_2) = F \end{cases}$$

A.3 Interprétation vectorielle

On reconnaît des vecteurs du plan :

$$L_1\big(\cos(\alpha_1),\, \sin(\alpha_1)\big) + L_2\big(\cos(\alpha_1 + \alpha_2),\, \sin(\alpha_1 + \alpha_2)\big) = (E,\, F)$$

Qui sont deux vecteurs de normes L_1 et L_2, faisant un angle α_2.

A.4 Calcul du deuxième angle

On prend la norme du vecteur au carré :

$$E^2 + F^2 = L_1^2 + L_2^2 + 2L_1L_2\cos(\alpha_2)$$

Et donc :

$$\cos(\alpha_2) = \frac{E^2 + F^2 - L_1^2 - L_2^2}{2L_1L_2}$$

Si $\frac{E^2+F^2-L_1^2-L_2^2}{2L_1L_2}$ n'est pas entre -1 et 1 alors il n'y a pas de solution au problème. Le point (x_p, y_p) ne pourra pas être atteint (hors de portée).

α_2 est donc calculé par la formule :

$$\alpha_2 = \pm \arccos\left(\frac{E^2 + F^2 - L_1^2 - L_2^2}{2L_1L_2}\right)$$

A.5 Calcul du premier angle

On introduit $R = \sqrt{E^2 + F^2}$ et l'angle global θ tel que :

$$(E, F) = R(\cos\theta, \sin\theta)$$

Donc :

$$\theta = \text{atan2}(F, E)$$

A.6 Décomposition angulaire

On peut décrire la relation entre les angles, positions et longueurs grâce à des nombres complexes sous forme exponentielle :

$$L_1 e^{i\alpha_1} + L_2 e^{i(\alpha_1+\alpha_2)} = e^{i\alpha_1}\left(L_1 + L_2 e^{i\alpha_2}\right)$$

Donc :

$$e^{i\alpha_1} = \frac{E + iF}{L_1 + L_2 e^{i\alpha_2}}$$

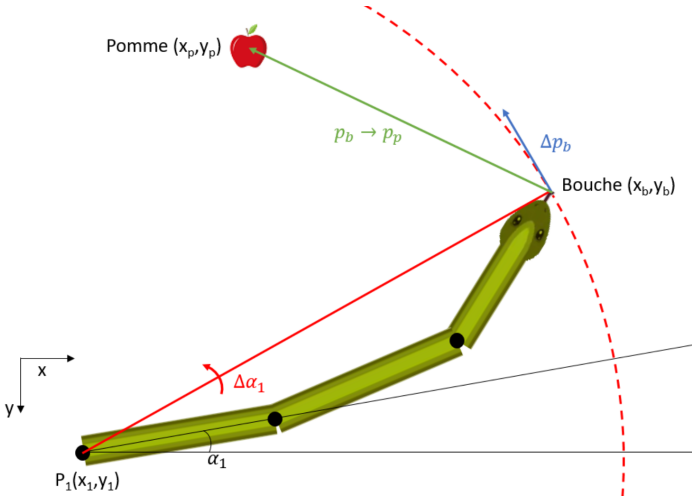
Finalement, on a la valeur de l'angle α_1 :

$$\alpha_1 = \text{atan2}(F, E) - \text{atan2}\left(L_2 \sin\alpha_2,\ L_1 + L_2\cos\alpha_2\right)$$

B Construction du Jacobien

B.1 Présentation du problème

Nous souhaitons pouvoir décrire le Jacobien du système du serpent, c'est-à-dire de construire les relations qui existent entre les rotations autour des centres de rotation et le déplacement de l'extrémité (la bouche) du serpent. Lorsque l'on tourne de $\Delta\alpha_1$ autour du point P1 (centre de rotation le plus à gauche) la bouche du serpent (p_b) se déplace le long du cercle de centre P1 et de rayon la distance entre P1 et la bouche. Notons cette direction de déplacement Δp_b (flèche bleue dans le schéma ci-dessous).



Si cette direction est alignée avec la direction entre la bouche et la pomme (notée $p_b \rightarrow p_p$ et dessinée en vert) alors tourner autour de P_1 sera très bénéfique à la solution. Si elle est dans le sens opposé alors il faut tourner dans l'autre sens ($-\Delta\alpha_1$) pour que ça soit bénéfique. Si par contre elle est dans la direction orthogonale, alors tourner (dans un sens ou dans l'autre) n'apportera aucun bénéfice, on ne tournera donc pas. Entre ces configurations, nous tournerons de la quantité correspondante au bénéfice.

B.2 Calcul de la direction de déplacement

Mathématiquement, nous pouvons calculer cette quantité de la manière suivante. Soient $P_i(x_i, y_i)$ la position du centre de rotation i (c'est-à-dire soit P_1, P_2 ou P_3), $p_b(x_b, y_b)$ la position de l'extrémité (la bouche) et $p_p(x_p, y_p)$ la position de la pomme.

La direction (le vecteur) du centre de rotation vers la bouche est :

$$\overrightarrow{P_i p_b} = (x_b - x_i,\ y_b - y_i)$$

La direction de déplacement $\overrightarrow{\Delta p_b}$ de la bouche par une rotation de centre P est :

$$\overrightarrow{\Delta p_b} = \overrightarrow{P_i p_b} \times \vec{z}$$

En deux dimensions ce produit vectoriel ($\times$) avec l'axe z (orthogonal au plan (x, y)) revient à faire une rotation de 90 degrés. Nous avons donc simplement :

$$\overrightarrow{\Delta p_b} = (y_{P_i p_b},\ -x_{P_i p_b})$$

B.3 Calcul de la quantité de rotation

Afin de représenter la quantité de rotation q_i pouvant participer à rapprocher la bouche de la pomme, nous allons projeter la direction de déplacement de la bouche sur la direction vers la pomme ($p_b \rightarrow p_p$) notée $\overrightarrow{p_b p_p}$. Plus ce projeté est grand plus nous tournerons. Le projeté se calcule grâce au produit scalaire entre ces deux vecteurs.

$$q_i = \overrightarrow{\Delta p_b} \cdot \overrightarrow{p_b p_p} = (y_{P_i p_b},\ -x_{P_i p_b}) \cdot (x_p - x_b,\ y_p - y_b)$$

$$q_i = y_{P_i p_b} * (x_p - x_b) + (-x_{P_i p_b}) * (y_p - y_b)$$

B.4 Mise à jour de l'angle

Finalement il suffit de mettre à jour l'angle autour de P_i en l'incrémentant de q_i. Cet incrément est mis à l'échelle par une valeur $\gamma \in [0, 1]$ permettant de contrôler la quantité de changement d'angle appliquée à chaque itération. En effet nous ne devons pas corriger chaque angle trop et dépasser notre cible et nous devons aussi limiter l'influence de la rotation d'un des centres sur le calcul des autres.

B.5 Pseudocode de la méthode

Voici le pseudocode complet de la méthode de résolution par Jacobien.

```
Tant que la bouche n'est pas suffisamment proche de la pomme et qu'on n'a pas atteint le nombre d'itérations max faire
    incrément du nombre d'itérations
    Pour chaque centre de rotation i faire
        calcul du vecteur P_i p_b
        calcul du déplacement Δp_b
        calcul de la quantité q_i
        calcul du nouvel angle α_i ← α_i + γ * q_i
    Fin Pour
    Pour chaque centre de rotation i faire
        application de l'angle α_i
    Fin Pour
    affichage de la pose
Fin Tant que
```

C Étapes détaillées d'une itération de la méthode FABRIK

1. La pomme est placée dans l'environnement 2D.
2. La direction entre la bouche du serpent et la pomme est calculée.
3. La bouche est déplacée sur la pomme et la direction de $\boldsymbol{P_3}$ à la pomme est calculée.
4. $\boldsymbol{P_3}$ est déplacé vers la bouche (pomme) pour respecter la taille de la dernière partie du serpent, et la direction de $\boldsymbol{P_3}$ à $\boldsymbol{P_2}$ est calculée.
5. $\boldsymbol{P_2}$ est déplacé vers $\boldsymbol{P_3}$ pour respecter la taille de la deuxième partie du serpent, et la direction de $\boldsymbol{P_2}$ à $\boldsymbol{P_1}$ est calculée.
6. $\boldsymbol{P_1}$ reste à sa position initiale, qui est fixe.
7. $\boldsymbol{P_2}$ est déplacé vers $\boldsymbol{P_1}$ pour respecter la taille de la première partie du serpent, et la direction de $\boldsymbol{P_2}$ à $\boldsymbol{P_3}$ est calculée.
8. $\boldsymbol{P_3}$ est déplacé vers $\boldsymbol{P_2}$ pour respecter la taille de la deuxième partie du serpent, et la direction de $\boldsymbol{P_3}$ à la pomme est calculée.