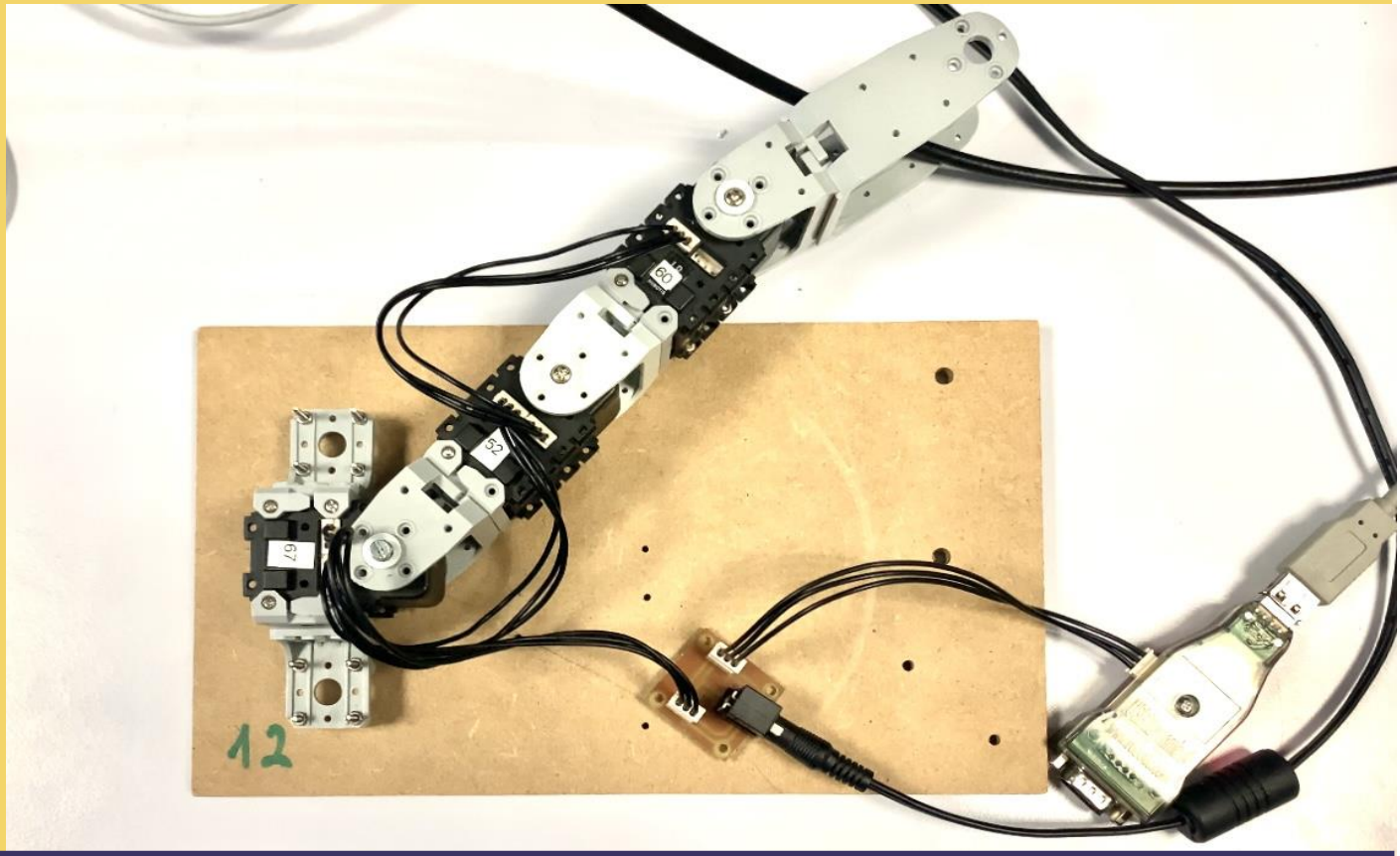


Projet de robotique LU3MEE01

Option Systèmes Robotiques, Groupe 1 MECA

Date limite du rendu : Mercredi 19 Avril 2023

Binôme :

Lara OUDJIT - 3801865

Samy HORCHANI - 28706765

Encadrants Projet et Responsables UE :

Brahim TAMADAZTE et Azad ARTINIAN

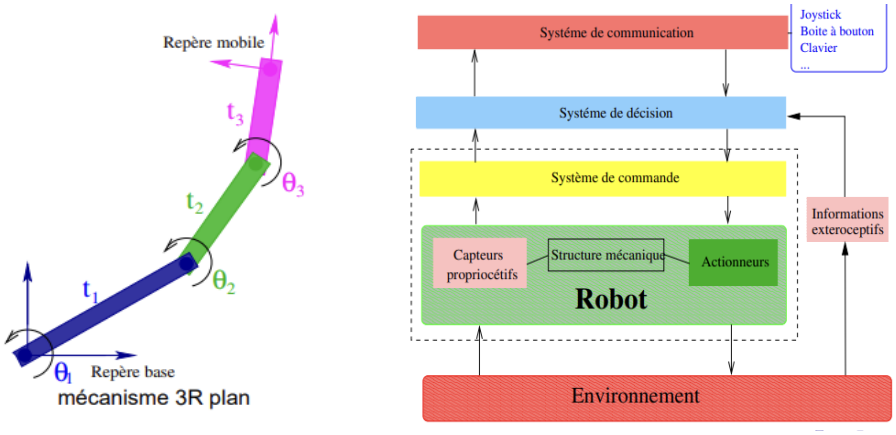
Sinan HALIYO et Sylvain ARGENTIERI

SOMMAIRE

PRESENTATION DU SYSTEME

Introduction

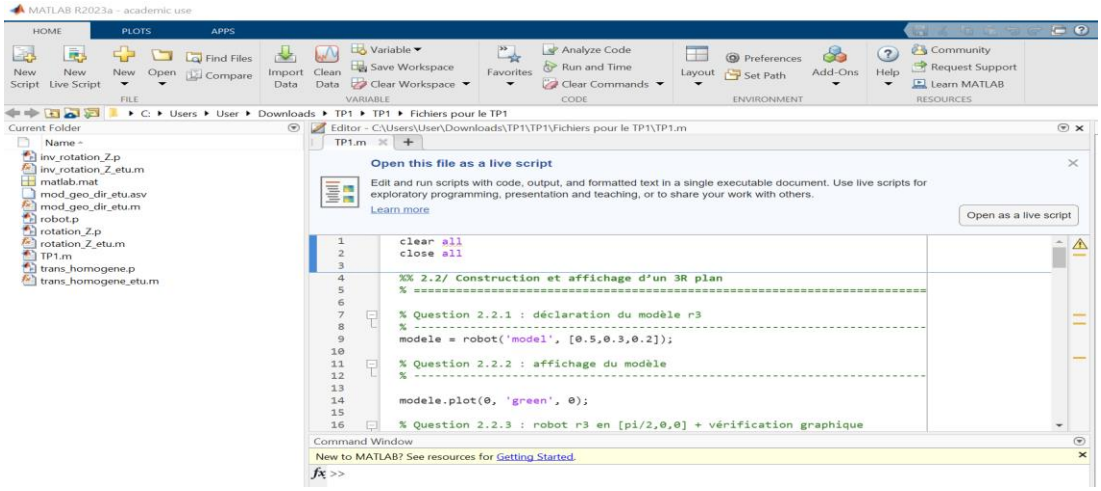
Au cours de l'année 2023, nous avons suivis les enseignements de l'option Systèmes Robotiques, visant à nous initier aux problématiques de la robotique. Nous avons observé une première approche plus théorique et mécanique, lors de cours magistraux. Ainsi, nous avons pu retracer l'histoire de cette science et apprendre les outils mathématiques de base, afin de construire des mouvements implantables à un système robotique :



Schémas récapitulatifs des structures robotiques (LU3ME001 CM1 2023, Sinan HALIYO)

Par la suite, nous avons suivi une seconde approche se rapportant plus au domaine de l'informatique et de l'électronique, lors des sessions de travaux pratiques. En effet, nous avions à disposition une maquette comportant un système robotique 3R associée à une connectique compatible avec nos ordinateurs.

Ces travaux pratiques nous ont permis de mettre en application les outils mathématiques, en les convertissant sous forme de fonctions et de commandes, dans un programme numérique en langage MATLAB :



Exemple d'interface MATLAB

Enfin, pour synthétiser notre apprentissage de la matière, nous avions pour objectif d'utiliser ces nouveaux outils pour la génération de mouvements et de commandes d'un robot sériel 3R. Ce rapport regroupe l'ensemble des résultats de notre réalisation expérimentale : en particulier, nous chercherons à détailler nos démarches pour le projet (les réponses au TP2, fournies dans une annexe, seront plus synthétiques). L'ensemble du programme sera restitué dans un fichier MATLAB, joint à ce compte-rendu.

Brève présentation de la maquette expérimentale

La maquette se présente comme un assemblage de trois moteurs DYNAMIXEL, fonctionnant au protocole TTL pour dialoguer entre eux, et reliés à une interface USB, dite USB2DYNAMIXEL :

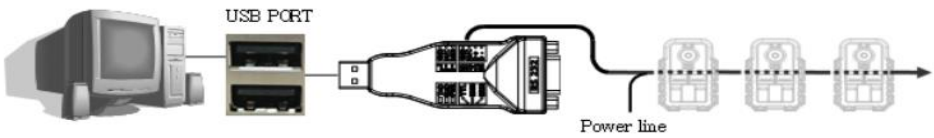


Schéma récapitulatif de l'interface UBS2DYNAMIXEL (énoncé TP2)

Les trois moteurs sont chacun repérés par un numéro, cet identifiant nous permet par la suite de leur donner des commandes transmises via l'USB :

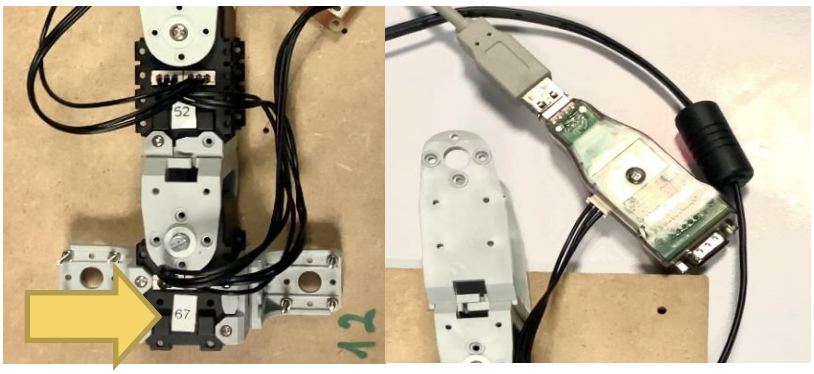


Photo de notre maquette réelle avec ID moteurs et USB

Il est également essentiel de connaitre l'identifiant du port USB correspondant à la maquette. Pour cela, nous pouvons le voir apparaître dans le gestionnaire de périphériques de l'ordinateur au moment où le câble USB est branché. Nous fonctionnerons avec le « USB Serial Port COMX », avec l'identifiant X :

Gestionnaire de périphérique pour identifier le port USB

PREMIERES ETAPES DE PROGRAMMATION

Schéma cinématique du robot (Q2.1)

A partir de notre maquette, nous pouvons repérer les données angulaires q1, q2 et q3, espacées des « bras » de longueurs l1, l2 et l3 et qui correspondent aux trois moteurs identifiés ID1, ID2 et ID3. Par ailleurs, le dernier moteur, qui est l'effecteur, est relié à une « pince » fixe, dans laquelle nous pouvons insérer un crayon.

Nous avons pour objectif final de faire faire au robot un « dessin ». Pour simplifier la compréhension, nous avons effectué un schéma de la cinématique du robot de manière manuscrite :

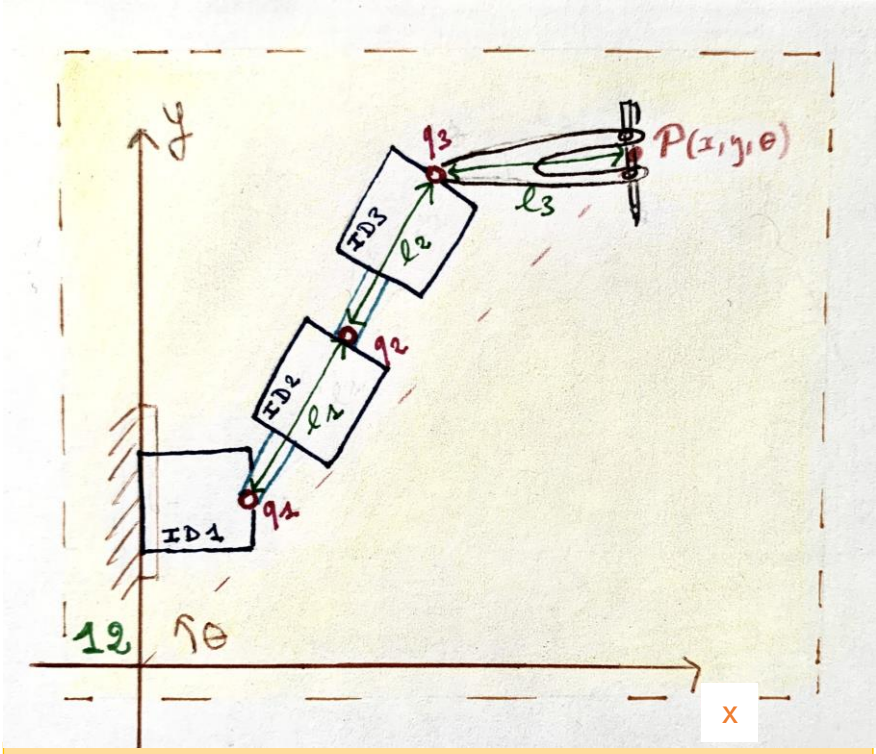


Schéma de la cinématique du robot 3R, dans un plan (X,Y,Z)

Nous prendrons l'habitude de ranger les données dans des vecteurs de 3 composantes (pour 3 moteurs) afin de commander le robot dans son ensemble et par soucis de relecture.

Nous mesurons les longueurs : l1 = 0.07m, l2 = 0.07m et l3 = 0.09m ; nous prenons en note les identifiants de notre modèle, par exemple : ID1 = 67, ID2 = 52 et ID3 = 60 ; même chose pour la connectique : PORT = 12 ; il faut préciser le BAUD = 1 000 000 ; les données angulaires seront regroupées dans un vecteur pour généraliser les commandes : q = [q1, q2, q3].

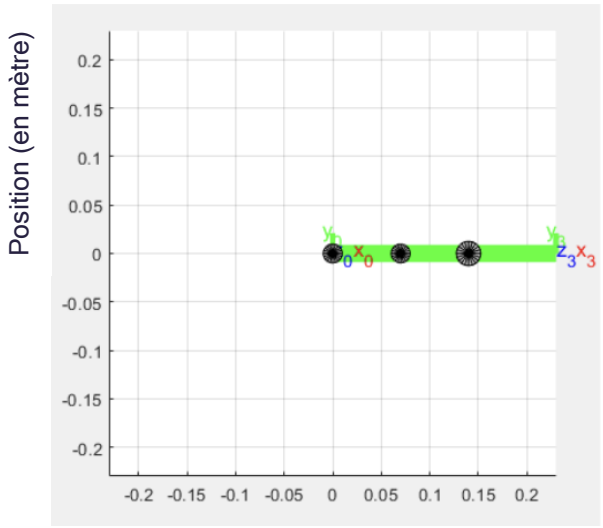
Initialisation de la maquette et simulation virtuelle (Q2.2)

Maintenant que la maquette est connectée et que nous avons les dimensions de notre bras robotisé, nous devons initialiser une maquette virtuelle dans notre programme MATLAB ; il s'agira de notre objet « **modele** ». Sachant que les moteurs fonctionnent avec un protocole API MATLAB, nous pouvons le faire à partir de la fonction « robot », issue du logiciel de programmation et qui nous permettra de faire le lien entre les actions des moteurs et nos commandes :

```
% 2.2 : Déclaration d'un modèle de robot
% ------------------------------------------------------------------------
modele = robot('model', [0.07,0.07,0.09]);
```

Code : déclaration du modèle virtuel

Nous pouvons également représenter le robot par une simulation graphique grâce à la commande « .plot ». La visualisation se fait dans un repère où l'on peut définir l'origine, ici nous prenons 0, et à titre comparatif avec les simulations suivantes, nous choisissons la couleur « green » :



Graphique : Bras 3R du modele dans un repère en mètres et radian (x(m), y(m), θ(rad))

Il faut également initialiser les dimensions pour la maquette réelle :

```
% Paramètres de la communications série
PORT = 12;       % A modifier avec le n° du port COM utilisé
BAUD = 1000000;

ID1 = 67;
ID2 = 52;
ID3 = 60;
POSITION = 512; % Correspond à la position zéros en steps ou le bras "vertical"
speed = 200;
maquette = robot('real',PORT,BAUD);  % initialisation maquette réelle
```

Code : initialisation de la maquette réelle

APPLICATION AU MGD (Q2.3)

Calcul des coordonnées opérationnelles

Afin de faire faire au robot un mouvement désiré, nous devons calculer les coordonnées du point atteint par son organe terminal.
Cela est possible à l'aide du **modèle géométrique direct** (MGD), une méthode de calcul mathématique pour les bras manipulateurs, qui, à la suite d'une configuration articulaire donnée (q1, q2, q3), nous permet de calculer les coordonnées du point (x, y, θ). A cet effet, nous définissons chaque angle des moteurs et nous les « rangeons » dans un vecteur q :

```
figure(2), title 'MGD'
q1 = 0;
q2 = pi/3;
q3 = 0;
q = [q1, q2, q3];
```

Code : choix d'une configuration articulaire

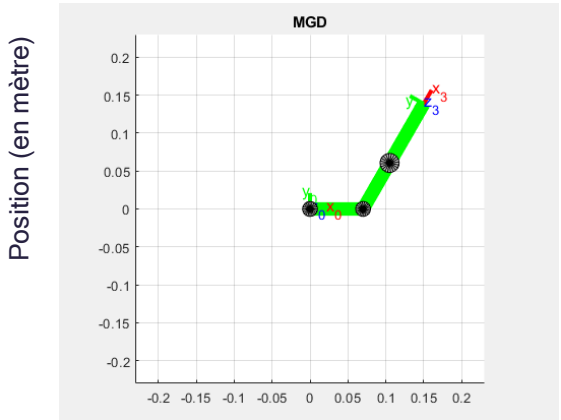
Nous devons ensuite implémenter ces nouvelles données angulaires au robot virtuel : cela est possible grâce à la fonction intrinsèque « .setAngularPosition ». De plus, dans le logiciel MATLAB, il existe déjà une fonction intrinsèque reprenant la méthode du MGD, qu'il suffit d'appliquer à notre modèle : « .mod_geo_dir() ».

```
modele.setAngularPosition(q);
mod_geo_dir_q = mod_geo_dir(modele);
```

Code : fonctions pour l'application du MGD

Nous pouvons observer le modèle virtuel dans la configuration q :

```
modele.plot(0, 'green', 0);
```



Graphique : Bras virtuel tracé à partir du MGD dans un repère en mètres et radian (x(m), y(m), θ(rad))

Concordance du modèle virtuel et du modèle réel

Nous pouvons vérifier la véracité de notre simulation graphique en cherchant à reproduire le mouvement sur notre maquette. Toutefois, l'application à notre bras robotique réel n'est pas immédiate car le robot fonctionne en « steps » ou « pas moteurs » (allant de 0 à 1024) : il faut donc convertir nos coordonnées angulaires en steps.

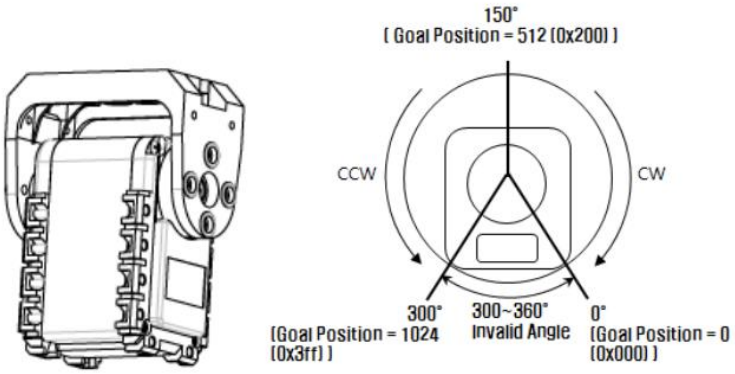


Schéma récapitulatif d'un servomoteur DYNAMIXEL *(énoncé TP2)*

L'idée est donc d'écrire une fonction, nommée « angle2step » qui effectue cette conversion en prenant en argument les angles (donnés en radian, sous la forme d'un vecteur q) et une variable appelée « offset », correspondant à la position initiale du robot.

Celle-ci est à définir dans le corps du programme, avant d'appeler la fonction. Nous choisissons comme position initiale celle du « bras vertical » à 150°. Cela correspond à un step de 512 environ.

```
function p = angle2step(q,offset)
% ANGLE2STEP convertit la valeur angulaire Q (rad) en une valeur de pas
% moteur P.
% OFFSET permet de spécifier à quelle valeur de pas moteur correspond la
% valeur angulaire 0.
%
% p = angle2step(q,offset);
thetaMax = 300*pi/180;

p = offset+(-q*1024/thetaMax);
```

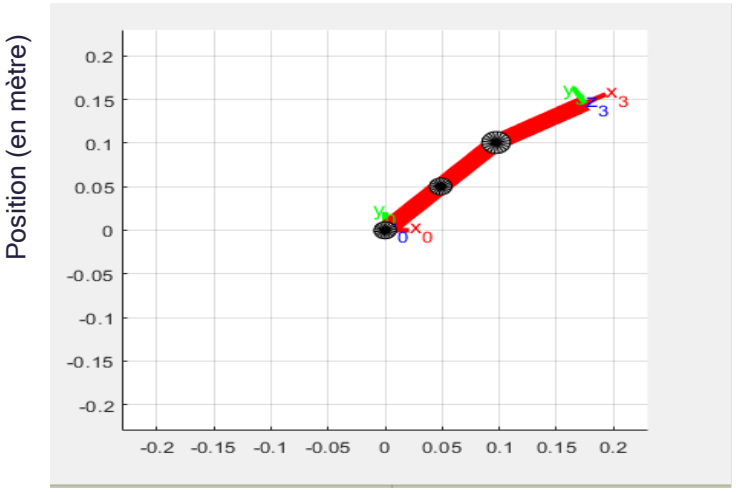
Code : fonction angle2step

Maintenant que nous pouvons communiquer à la maquette les coordonnées, nous pouvons lui « commander » un mouvement précis grâce à la fonction intrinsèque « .setStepPosition() » qui prend en argument les moteurs ID1, ID2 et ID3, ainsi que leur position associée, définie par angle2step :

```
% 2.1.6 : utilisation de la fonction setStepPosition
% ------------------------------------------------------------------------
POSITION = 512;
%pause(0.5)
maquette.setStepPosition(ID, [POSITION,POSITION,POSITION]);
%pause(0.5)
```

Code : Position du robot réel à partir du MGD

Nous pouvons choisir une configuration q au hasard ; après avoir exécuté le programme, le robot prend position et les modèles, virtuel et réel, semblent concorder :



Graphique : Bras virtuel par le MGD dans un repère en mètres et radian (x(m), y(m), θ(rad))

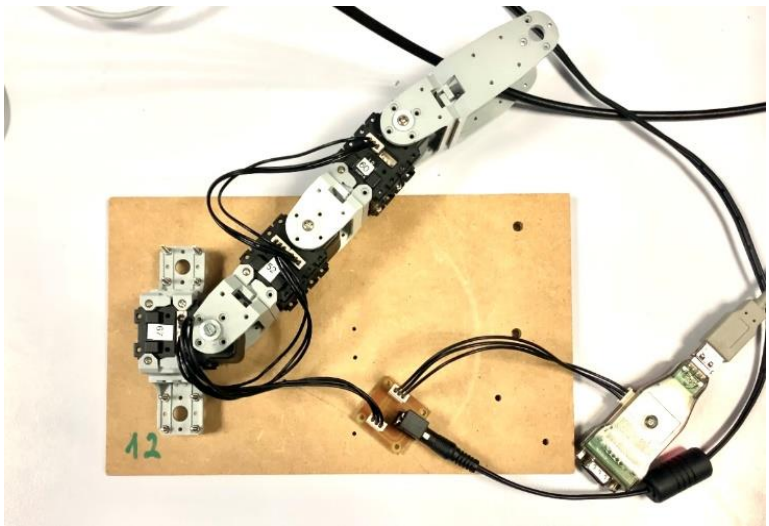
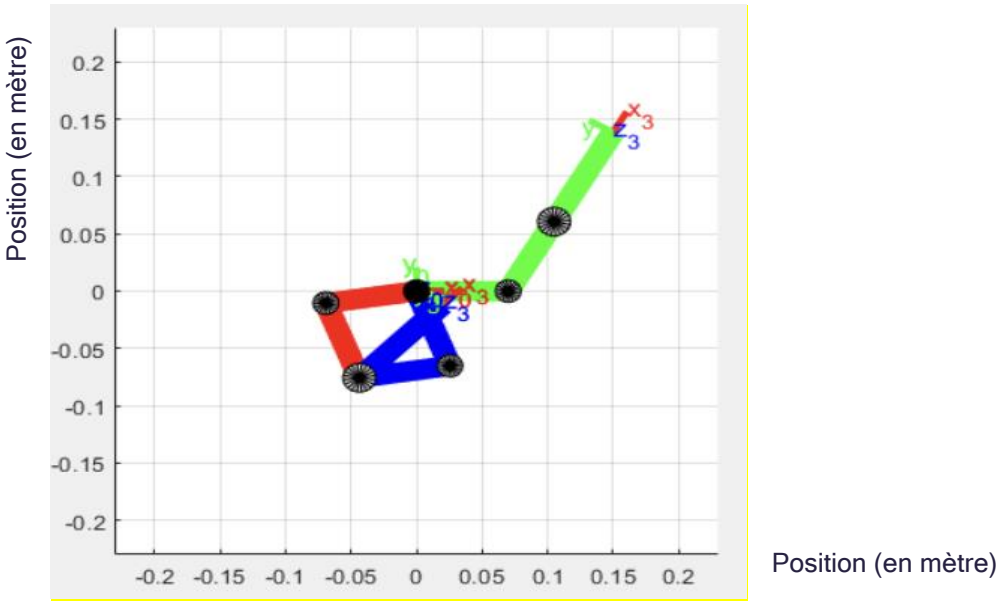


Photo : Bras réel par le MGD

Adaptation aux butées mécaniques et maîtrise de la trajectoire (Q2.4)

Lors des sessions de travaux pratiques, nous avons rencontrées des formes de bugs lors de l'exécution de mouvements donnés « au hasard ». En effet, alors que le modèle de robot virtuel le simule parfaitement, le moteur réel ne peut tourner à 360° :



Graphique : bras virtuel avec rotation irréaliste (à partir du MGI, voir Q2.5), dans un repère en mètres et radian (x(m), y(m), θ(rad))

Le robot réel est limité dans son mouvement de rotation par des obstacles comme la fixation au bâti de la maquette ou encore le bras du moteur précédent.

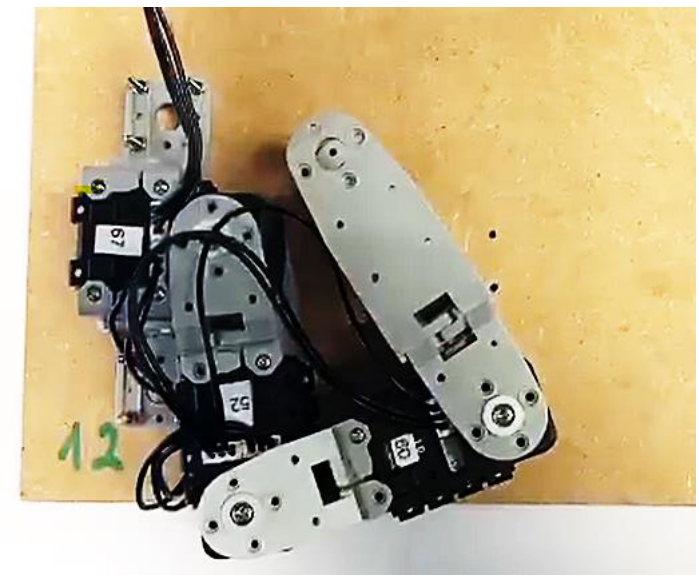


Photo : robot « contraint » dans son mouvement

En négligeant cet aspect, une anomalie se produit au cours de laquelle le moteur semble bloqué dans sa position, et souvent, contracte un protocole de sécurité anti-surchauffe (repérable par une diode rouge qui clignote à son exécution). Dès lors, il faut réinitialiser la maquette réelle : éteindre et rallumer l'interrupteur sur le branchement USB2DYNAMIXEL.

Photo : Procédure de sécurité de l'actionneur

Nous pouvons éviter ce problème en définissant les butées mécaniques, c'est-à-dire, les limites propres à chaque moteur : une valeur minimale (que l'on considèrera comme la limite à gauche) et une valeur maximale (limite à droite). *La procédure pour déterminer ces extremums est à répéter pour chaque extremum de chaque moteur.*

Pour cela, nous éteignons le bras robotisé, puis nous le forçons à tourner d'un côté. Quand le mouvement maximum est atteint, l'interrupteur peut être mis sur ON. Ensuite, dans l'invite de commande, nous utilisons la fonction « .getStepPosition(IDX) » en donnant en argument le moteur choisi X. Le logiciel renvoie la valeur limite atteinte.

```
>> positions
>> maquette.getStepPosition(ID)

ans =

   220   794   664
```

Code : invite de commande - renvoi des positions articulaires en temps réel

Pour les employer par la suite, il faut déclarer les extremums recensés pour chaque moteur :

```
%butée min de chaque moteur
step_min_ID1 = 220; %butée min moteur 1
step_min_ID2 = 159; %butée min moteur 2
step_min_ID3 = 165; %butée min moteur 3

%butée max de chaque moteur
step_max_ID1 = 796; %butée max moteur 1
step_max_ID2 = 862; %butée min moteur 2
step_max_ID3 = 866; %butée min moteur 3
```

Code : butées articulaires extrémales pour chaque moteur

Nous pouvons maintenant écrire une fonction dite « limit », prenant en argument une position donnée et les extremums associés au moteur donné :

```
function new_pos = limit(pos,val_min,val_max)
% LIMIT(POS,VAL_MIN,VAL_MAX) limite les valeurs de position dans le
% vecteur POS entre une valeur minimale VAL_MIN et maximale VAL_MAX
% autorisée par la geometrie du robot.

new_pos = zeros(1,length(pos)); %vecteur de taille 1*le nombre de position donné en argument

for i = 1:length(pos) %pour chaque position
    if pos(i) <= val_min
        new_pos(i) = val_min;

    elseif pos(i)>= val_max
            new_pos(i) = val_max;
    else
        new_pos(i) = pos;
    end
end
```

Code : fonction limit

Ensuite, il suffit de commander au robot de se déplacer avec la fonction « .setStepPosition() », sans oublier de prendre en argument la nouvelle fonction **limit**, et de définir la position donnée par la fonction **angle2step** :

```
maquette.setStepPosition(ID1,limit(angle2step(q1,offset),step_min_ID1, step_max_ID1 ));
maquette.setStepPosition(ID2,limit(angle2step(q2,offset),step_min_ID2, step_max_ID2 ));
maquette.setStepPosition(ID3,limit(angle2step(q3,offset),step_min_ID3, step_max_ID3 ));
```

Code : commande de déplacement au robot réel

APPLICATION AU MGI (Q2.5)

Nous avons vu que pour commander aux actionneurs un mouvement, nous pouvions utiliser le modèle géométrique direct (MGD), qui utilise une configuration angulaire (q1, q2, q3) en argument et renvoie les coordonnées du point (x, y, θ) associé. Pourtant, il existe également un modèle mathématique parallèle : le modèle géométrique inverse (MGI), qui utilise les coordonnées du point (x', y', θ') en argument, et renvoie ainsi la configuration articulaire (q1', q2', q3') associée.

Une fois de plus, la fonction intrinsèque faisant référence à ce modèle est déjà implémentable dans notre programme : « .mod_geo_inv() ». Dans le but d'effectuer un mouvement prenant en compte les limites physiques du robot, tout en atteignant sa destination finale, nous allons appliquer cette fonction à notre modèle :

Pose cible avec prise en compte des butées articulaires

Nous commençons par choisir une configuration cible à atteindre et l'insérons dans la fonction **mod_geo_inv**. Il est à noter que la fonction MGI du programme renvoie deux solutions : q_mdi1 et q_mdi2, étant donné la symétrie de l'espace opérationnel. Nous stockons les deux solutions, q1_mdi et q2_mdi, dans un vecteur. *Dans le cas où une seule solution serait obtenue, q_mdi1 et q_mdi2 seraient superposées.*

```
% 2.5 : Modele géométrique inverse
% ------------------------------------------------------------------------
figure(3), title 'MGI'
[q1_mdi, q2_mdi] = modele.mod_geo_inv(0.15,-0.05,0);
%stockage de nos deux solutions en appliquant le modèle géo. inverse dans
%q1_mdi et q2_mdi
```

Code : application du MGI au modèle virtuel

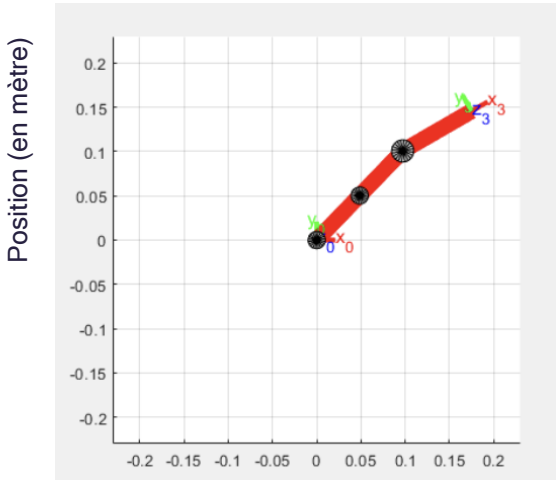
Nous pouvons maintenant demander au modèle d'effectuer la pose pour chaque solution. Pour cela, nous pouvons reprendre le corps de code du modèle précédent en modifiant seulement les arguments.

```
modele.setAngularPosition(q1_mdi);
modele.plot(1, 'red', 0);

modele.setAngularPosition(q2_mdi);
modele.plot(1, 'blue', 0);
```

Code : tracé de chaque solution issue du MGI

Nous devrions nous attendre à observer 2 bras virtuels, un rouge et un bleu.

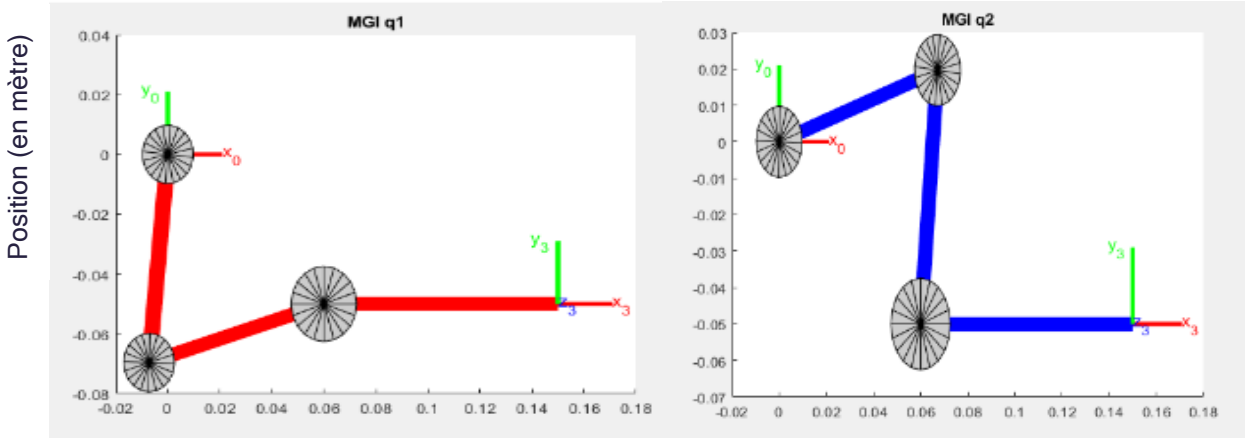
A l'exécution du programme, nous avons rencontré plusieurs difficultés : le modèle virtuel n'affichait qu'une seule solution, soit une seule couleur, alors que nous avions demandé l'affichage successif de q_mdi1 et q_mdi2. Autrement dit, la solution était toujours unique :



Graphique : Bras virtuel tracé à partir du MGI, solutions superposées, dans un repère en mètres et radian (x(m), y(m), θ(rad))

Cela était dû à la contrainte des limites angulaires du robot, qui étaient appliquées au modèle virtuel et restreignaient particulièrement l'ensemble des solutions. *Dans un premier temps, nous devions choisir une pose finale avec un faible écart à la position initiale.*

Par la suite, nous avons trouvé une autre façon d'observer les deux solutions : éteindre le moteur, configurer une pose finale « à la main », repérer manuellement cette pose finale (x', y', θ') et l'insérer dans la fonction MGI. Le moteur effectue un mouvement en prenant compte des butées articulaires, nous voyons bien deux solutions atteignables, MGI q1 et MGI q2 :



Graphique : solutions q1 et q2, dans un repère en mètres et radian (x(m), y(m), θ(rad))

Tracé de la trajectoire et animation du modèle (Q2.6)

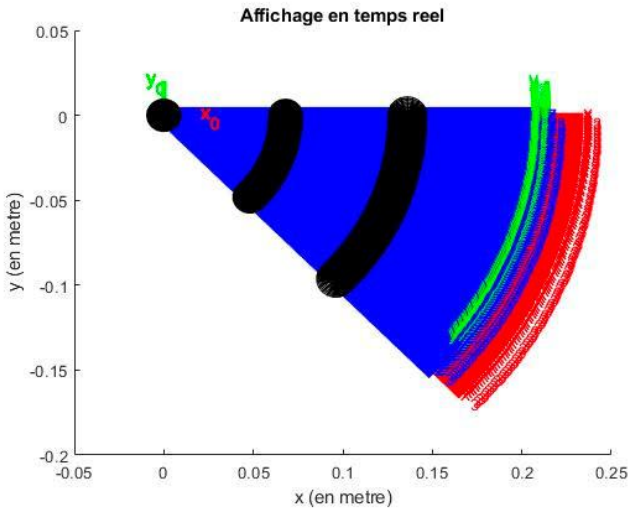
Sachant que nous pouvons analyser le robot pendant qu'il bouge et pour différentes positions, nous cherchons dans un premier temps à représenter sa trajectoire sur un graphique. Cela est possible en programmant une boucle sur la maquette en mouvement.

Nous avons besoin de la fonction tic (début du temps) et toc (arrêt du temps) pour agir comme chronomètre sur notre boucle temporelle. Grâce à la fonction « .isMoving(IDX) », nous pouvons effectuer une commande pendant le mouvement du robot : ici, l'objectif est de représenter graphiquement des positions demandées successivement :

```
maquette.setSpeed(ID2, speed)
x = [];
y = [];

maquette.setStepPosition(ID2, 0); % nous plaçons le moteur 2 à la position 0
pause(4) %afin de s'assurer que le robot ait le temps de se positionner
maquette.setStepPosition(ID2, 1023); % à la position 1023

tic;
timerVal = tic;
while (maquette.isMoving(ID2)) == 1 % tant que le moteur 2 bouge

    x = [x, toc]; %nous relevons le temps
    y = [y, maquette.getStepPosition(ID2)]; % nous relevons la position du moteur 2

end
```

Code : Animation du modèle en temps réel

A titre d'exemple graphique, voici le rendu d'une animation que nous avons fait pour la question 3 :



Graphique : exemple d'animation d'un mouvement en temps réel (voir Q3)

COMMANDE DE TRAJECTOIRES AU ROBOT (Q3)

Choix de la configuration :

Nous déterminons une position cible pour chaque moteur et commandons à la maquette de l'atteindre :

```
pos_init = [-pi/4, 0, 0]; %Nous choisissons une position de départ
modele.setAngularPosition(pos_init); %nous plaçons le modele virtuelle à la position de départ

step  = angle2step(pos_init, offset); % convertissement de la position initiale (step) en une valeur en angle

maquette.setStepPosition(ID, step); %nous placons la maquette du robot à la position choisi
modele.plot(0, 'blue', 0); %affichage du modele virtuel
pos_cible = [pi/4, 2*pi/3, -pi/6];
```

Code : configuration initiale et finale pour commande de trajectoire

Utilisation du modèle géométrique inverse :

Pour programmer au modèle la position cible, nous utilisons le modèle géométrique inverse pour chaque position intermédiaire :

```
pos_inter = modele.getAngularPosition();
[q1, q2] = modele.mod_geo_inv(pos_cible(1),pos_cible(2),pos_cible(3)); %determination du modele geométrique inverse
```

Code : application du MGI entre chaque position intermédiaire

Choix de la solution :

Sachant que le modèle géométrique inverse fournit deux solutions, q1 et q2, nous sommes amenés à en choisir un pour procéder aux étapes suivantes. Un premier tri se fait par la nécessité que la solution soit atteignable par la maquette réelle (butées articulaires). Un second critère de choix est la norme : la solution suscitant le moindre coût énergétique est à privilégier :

```
%Choix entre q1 et q2 => dq = qi - pos_inter avec la valeur absolue
q_min = min([abs(q1),abs(q2)]);

dq = q_min-pos_inter;
```

Code : critère de choix de la solution du MGI par la norme

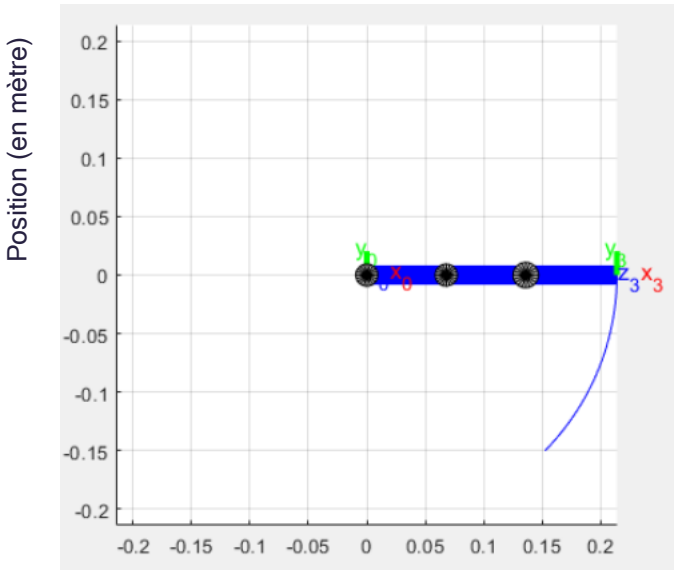
Programmation sur le modèle virtuel

Maintenant que nous avons défini les variables nécessaires, nous pouvons écrire une boucle effectuant l'interpolation linéaire afin de calculer la trajectoire à effectuer entre chaque position intermédiaire :

```
N = 100;
for i = 1:N % N fois
    pos_c = modele.getAngularPosition(); %recuperons la position du modele virtuelle
    modele.setAngularPosition(pos_c+(dq/N)); %Interpolation linéaire
    step2=angle2step( (pos_c+(dq/N)), offset);
    maquette.setStepPosition(ID, step2)
    modele.plot(0,'blue',1)
    pause(0.1)
end
```

Code : boucle effectuant l'interpolation linéaire de tous les points formant la trajectoire

A l'exécution de la cellule, le programme affiche la simulation du bras robot :



Graphique : bras virtuel traçant la trajectoire interpolée, dans un repère en mètres et radian (x(m), y(m), θ(rad))

Nombre de points intermédiaires N

Avoir plus de points intermédiaires nous permet de mieux contrôler la trajectoire du robot car il définit lui-même son parcours entre chaque point. Cependant, nous avons remarqué que **pour un nombre de points intermédiaires important, le robot réel comporte un mouvement saccadé** *(voir vidéo).* Ceci peut s'expliquer par un grand nombre de départs-arrêtés associés à son inertie.

Dessin de formes

Nous avons cherché à faire faire au robot des formes particulières, en insérant un crayon dans sa pince (organe terminal). Pour cela nous créons un fichier à part « bonus_letter.m ».

Nous devons dans un premier réécrire les paramètres d'initialisation du robot :

```
%INITIALISATION
clear all
clc

% Paramètres de la communications série
PORT = 9;       % A modifier avec le n° du port COM utilisé
BAUD = 1000000;

ID1 = 1; %ID du premier moteur
ID2 = 2; %ID du deuxieme moteur
ID3 = 3; %ID du troisieme moteur
ID = [ID1, ID2, ID3]; %Vecteur contenant les ID des moteurs


offset = 512; %position initiale coresspondant à notre 0 (pos verticale)

speed = 100; %vitesse moteur
maquette = robot('real',PORT,BAUD);  % Connexion aux moteurs
maquette.setSpeed(ID, [speed, speed, speed]); % setting des moteurs à la vitesse speed

modele = robot('model', [0.07,0.07,0.09]); %taille mesurée à la regle sur notre robot
```

Code : initialisation du robot pour le nouveau script « bonus_letter.m »

Ainsi que pour les butées articulaires :

```
%butée min de chaque moteur
step_min_ID1 = 220; %butée min moteur 1
step_min_ID2 = 159; %butée min moteur 2
step_min_ID3 = 165; %butée min moteur 3


%butée max de chaque moteur
step_max_ID1 = 796; %butée max moteur 1
step_max_ID2 = 862; %butée min moteur 2
step_max_ID3 = 866; %butée min moteur 3

%initialisation d'une matrice contenant les butées min et max de chaque
%moteur
mat = [[step_min_ID1,step_min_ID2,step_min_ID3]; [step_max_ID1,step_max_ID2, step_max_ID3]];
%reinitialisation de la position à notre position "0" offset
maquette.setStepPosition(ID, [offset, offset, offset]);
pause(3) %afin de s'assurer que les moteurs ait le temps de se placer
```

Code : déclaration des limites articulaires du robot et rangement dans une matrice

Une première façon de programmer une trajectoire « complexe » est de créer une fonction appelée « position() » nous permettant de saisir manuellement sur un graphique les points intermédiaires de la trajectoire :

```
%%
%Permet de tracer des points directement sur le graphe et de les relever
modele.plot(0, 'blue', 1);
positions(); %Nous tracons les points
```

Code : appel de la fonction « position » pour placer des points manuellement

La fonction **position()** se présente comme suit :

```
figure(1)
axis([-0.25 0.25 -0.25 0.25]) %on définit les limites des axes
XY = []; %vecteur qui va contenir les points placés
n = 100; %nombre de points à relever

for i = 1:n %pour chaque point
    xy = ginput(1); %nous placons le point sur la figure et les stockons dans xy
    XY = [XY; xy]; %nous rajoutons le point sélectionné à notre vecteur
    figure(1), hold on %la figure va retenir dans l'affichage le point précédent
    plot(XY(:,1), XY(:,2), 'ok')
end
```

*Code : fonction **position***

Il faut maintenant programmer la boucle pour que le robot effectue la suite de positions intermédiaires définies à la main :

```
for i = 1:length(XY) %pour chaque point
    [q1, q2] = modele.mod_geo_inv(XY(i,1), XY(i,2), 0); %stockage des solutions du mod. geo. inv.
    if(norm(q1-modele.getAngularPosition())<norm(q2-modele.getAngularPosition())) %détermination de la solution ayant la plus petite norme.
        q=q1;
    else
        q=q2;
    end
    modele.setAngularPosition(q); %positionnement du robot virtuel à la position q choisie

    modele.plot(0, 'blue', 1);
    for j = 1:3 %pour chaque moteur, on le positionne en tenant en compte les limites (butées)
        maquette.setStepPosition(ID(j), limit( angle2step(q(j), offset), mat(1, j),mat(2, j) ) );
    end
    pause(0.5)
end
```

Code : boucle pour traçage des positions successives à partir d'une entrée manuelle

Un problème persiste : le mouvement est saccadé et le robot choisit sa trajectoire entre chaque point intermédiaire. De plus, le crayon étant bloqué sur la feuille, le robot laisse sa trace de ses déplacements intermédiaires :

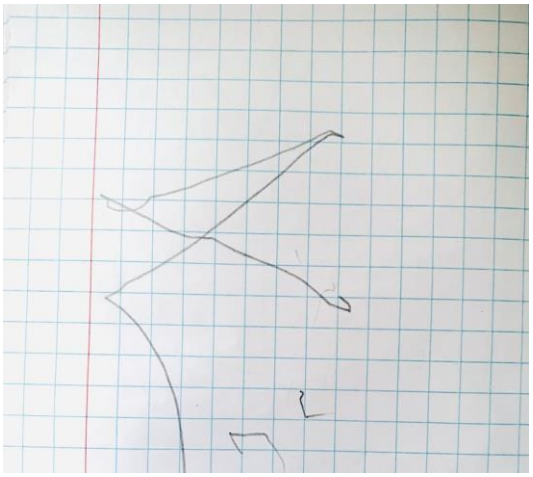


Photo : exemple de tracé final par le robot réel (objectif initial : tracer la lettre L …)

Pour tenter de résoudre ce problème, nous pouvons demander que le robot « attende » qu'on lui donne l'ordre de passer au point suivant. Nous ajoutons la commande avant la fin de la boucle :

```
% maquette.setStepPosition(ID(j), limit( angle2step(q(j), offset), mat(1, j),mat(2, j) ) );
 pause(); %afin que chaque point ce trace lorsque l'on appuie sur "entrée" dans la console
 %car nous observons qu'il est plus simple de controler les tracer du
 %robot ainsi.
```

Code : ajout d'une pause intermittente dans la boucle de mouvement

Nous pouvons tracer une forme de L avec ces améliorations :

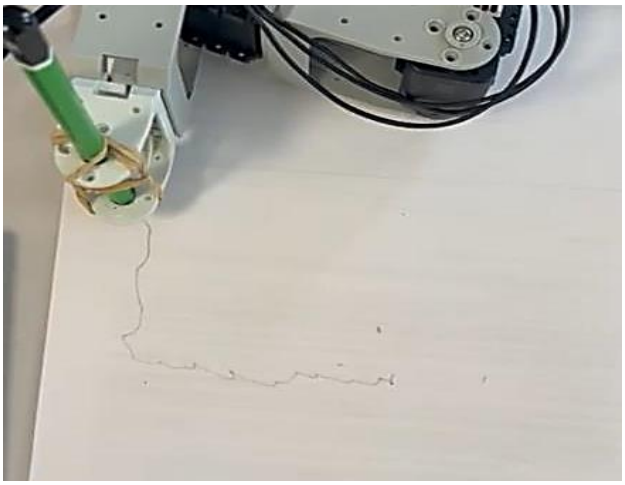


Photo : dessin de la lettre L par le robot réel

PISTES D'EXPLOITATION POUR EVITER LES SACCADES

Nous avons cherché quelques pistes pour éviter les saccades comme la commande en vitesse.

Ecriture de la matrice jacobienne :

$$J = \begin{bmatrix} \frac{\partial x}{\partial \theta_1} & \frac{\partial x}{\partial \theta_2} & \frac{\partial x}{\partial \theta_3} \\ \frac{\partial y}{\partial \theta_1} & \frac{\partial y}{\partial \theta_2} & \frac{\partial y}{\partial \theta_3} \\ \frac{\partial z}{\partial \theta_1} & \frac{\partial z}{\partial \theta_2} & \frac{\partial z}{\partial \theta_3} \end{bmatrix}$$

Ici nous prenons le cas où θ_3 est nul, soit un robot 2R :

$$J = \begin{bmatrix} \frac{\partial x}{\partial \theta_1} & \frac{\partial x}{\partial \theta_2} \\ \frac{\partial y}{\partial \theta_1} & \frac{\partial y}{\partial \theta_2} \end{bmatrix}$$

Nous pouvons calculer les positions x, y, θ avec le modèle géométrique direct ou MGD :

$$x = l1\cos(\theta_1) + l2\cos(\theta_1 + \theta_2)$$
$$y = l1\sin(\theta_1) + l2\sin(\theta_1 + \theta_2)$$
$$\theta = \theta_1 + \theta_2$$

Sachant que le modèle cinématique direct ou MCD correspond au MGD dérivé par rapport au temps, nous pouvons calculer les vitesses $\dot{x}$, $\dot{y}$, $\dot{\theta}$:

$$\dot{x} = -l_1\dot{\theta_1}\sin(\theta_1) - l_2\,\dot{\theta}_{12}\sin(\theta_{12})$$
$$\dot{y} = l_1\dot{\theta_1}\cos(\theta_1) + l_2\,\dot{\theta}_{12}\cos(\theta_{12})$$
$$\dot{\theta} = \dot{\theta_1} + \dot{\theta_2}$$

Maintenant que nous avons les expressions de (x, y) nous pouvons calculer la matrice jacobienne, sachant que θ_1 et θ_1 sont des scalaires :

$$J = \begin{bmatrix} -l_1\theta_1 sin(\theta_1) - l_2(\theta_1 + \theta_2)\sin(\theta_1 + \theta_2) & -l_2(\theta_1 + \theta_2)\sin(\theta_1 + \theta_2) \\ l_1\theta_1 cos(\theta_1) + l_2(\theta_1 + \theta_2)\cos(\theta_1 + \theta_2) & l_2(\theta_1 + \theta_2)\cos(\theta_1 + \theta_2) \end{bmatrix}$$

Cette écriture pourrait se simplifier par l'identité remarquable :

$$Cos(a + b) = cos(a)cos(b) - sin(a)sin(b)$$
$$Sin(a + b) = sin(a)cos(b) + sin(b)cos(a)$$

Nous obtenons un système matriciel à résoudre pour obtenir les vitesses de rotation $\dot{\theta}$ avec :

$$\dot{X} = J\,.\,\dot{\theta}$$

Nous obtenons un autre système matriciel à résoudre pour obtenir les vitesses $(\dot{x}, \dot{y})$ si nous prenons l'inverse de la matrice jacobienne :

$$J^{-1}\,.\,\dot{X} = \dot{\theta}$$

CONCLUSION

Au cours de l'année 2023, nous avons donc suivis les enseignements de l'option Systèmes Robotiques, visant à nous initier aux problématiques de la robotique : ce projet nous a permis de mettre en pratique les éléments acquis en cours magistral (Interpolation, MGD, MGI, ...) à travers une maquette de robot 3R et une modélisation virtuelle de celui-ci sur MATLAB.

Nous avons ainsi pu observer les limites du modèle virtuel en le comparant aux mouvements du robot. En effet, nous nous sommes heurtés à de nombreuses contraintes physique, comme la répétabilité d'une commande de position ou les butées articulaires qui peuvent contraindre le robot dans une position non choisie initialement.

Un second aspect dont la mise en œuvre fut difficile consistait à commander une trajectoire précise et complexe. Malgré une réalisation lisse du modèle virtuel, la maquette réelle comportait un mouvement saccadé ou imprédictible dans son choix de trajectoire intermédiaire d'un point à un autre.

Ce projet fut très enrichissant pédagogiquement : nous avons apprécié pouvoir observer nos réalisations dans le domaine de la robotique, créer de toute pièce notre programmation et pouvoir l'implémenter au fur et à mesure des séances de travaux pratiques.

Nous remercions nos encadrants de travaux pratiques : M. Azad ARTINIAN et M. Brahim TAMADAZTE ; les responsables de l'unité d'enseignement M. Sinan HALIYO et M. Sylvain ARGENTIERI ; ainsi que toute l'équipe pédagogique.

Samy HORCHANI et Lara Chloé OUDJIT.

ANNEXE : TP2

Dans cette section, nous nous intéressons aux questions de la 2nde séance de travaux pratique, au cours de laquelle nous avons cherché à utiliser les actionneurs de manière intelligente (appels de fonctions MATLAB pendant leur fonctionnement, test de vitesse, position …). La majorité du travail effectué dans cette séance nous a permis de comprendre et de concevoir au mieux nos réponses pour les séances de projet.

Utilisation du moteur en bout de chaîne cinématique

Fonction Step2Angle : formule (Q2.2.8)

L'objectif de la fonction est de convertir des coordonnées pas moteurs (steps de 0 à 1024) en coordonnées angulaires (q1, q2, 3). La fonction prend en argument q et la variable offset. Pour cela, nous convertissons l'angle de radian à degré par la formule suivante :

$$\theta_{max} = \frac{300\pi}{180}$$

Enfin, nous définissons « p », effectuant la conversion finale en utilisant le step maximal du robot et la variable offset :

$$p = -\frac{(offset - q)\theta_{max}}{1024}$$

Fonction Angle2Step : formule (Q2.2.10)

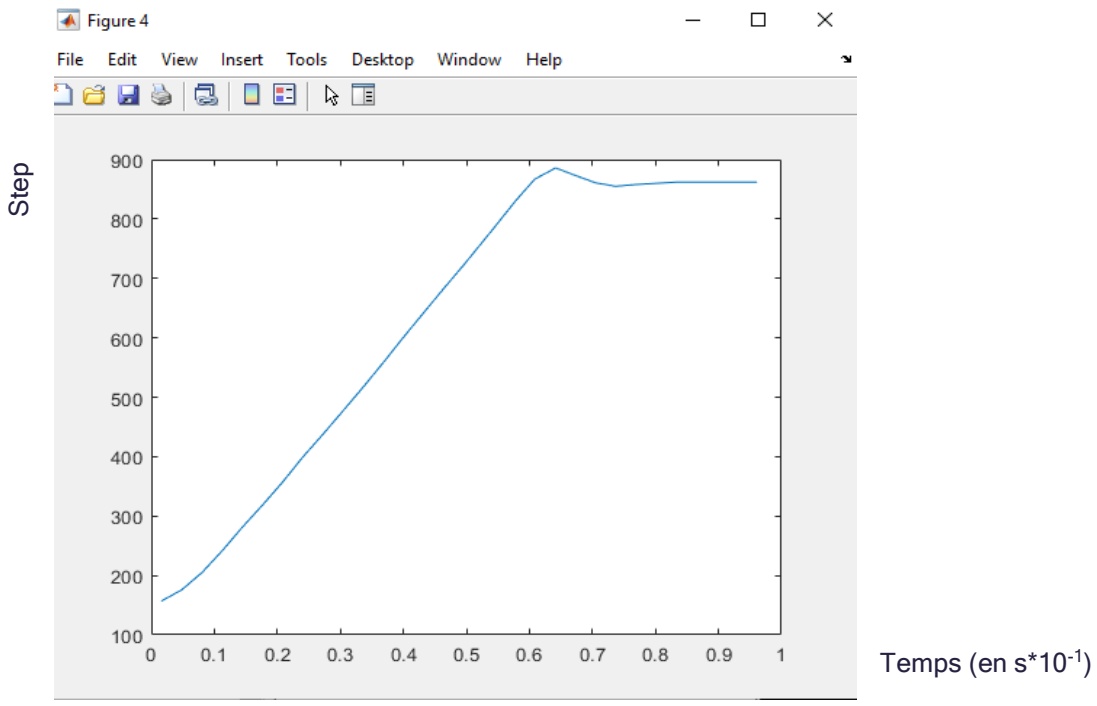
L'objectif de cette fonction est de convertir des données angulaires en pas moteurs. Pour cela, nous pouvons nous inspirer de la fonction précédente, Step2Angle :

$$p = offset + \frac{q1024}{\theta_{max}}$$

Tracé de la position en fonction du temps (Q2.2.12.a)

```
% 2.2.12 : tracé des positions en fonction du temps
% ------------------------------------------------------------------------
figure(1), title 'Position en fonction du temps'
speed = 100; %valeur de speed à changer à chaque fois
    maquette.setSpeed(ID2, speed)
    x = [];
    y = [];

    maquette.setStepPosition(ID2, 0); % nous plaçons le moteur 2 à la position 0
    pause(4) %afin de s'assurer que le robot ai le temps de se positionner
    maquette.setStepPosition(ID2, 1023); % nous essayons de mettre le robot à la position 1023

    tic;
    timerVal = tic;
    while (maquette.isMoving(ID2)) == 1 % tant que le moteur 2 bouge

        x = [x, toc]; %nous relevons le temps
        y = [y, maquette.getStepPosition(ID2)]; % nous relevons la position du moteur 2

    end
    plot(x,y);
    hold on
```

Code : tracé de la position en fonction du temps



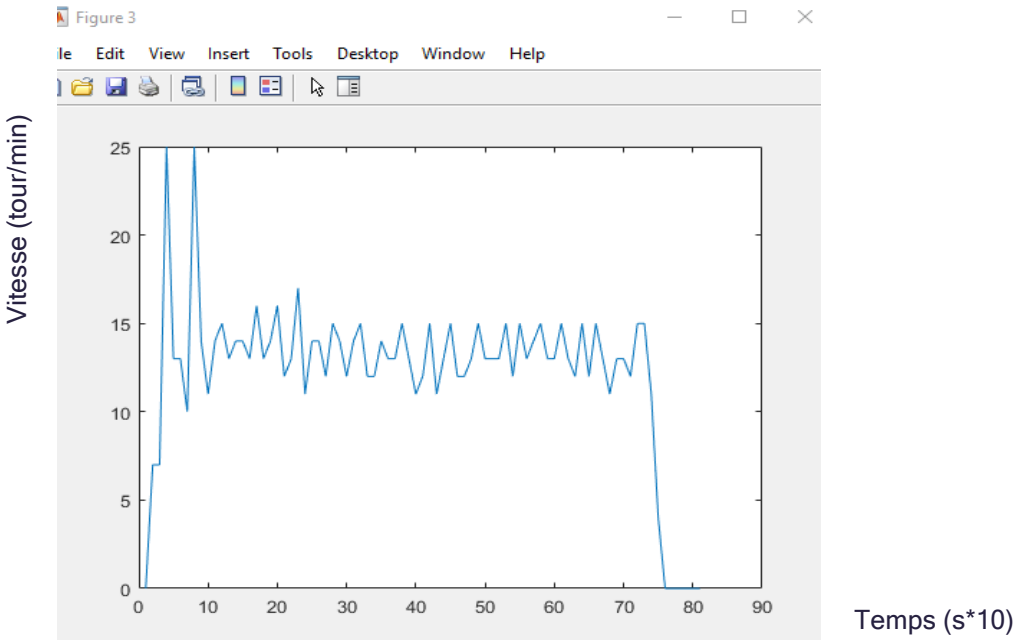
Graphique : tracé de la position en fonction du temps, dans un repère en temps et en steps

Tracé de la vitesse en fonction du temps (Q2.2.12.b/14)

```
% 2.2.12b/14 : tracé des vitesses en fonction du temps
% ------------------------------------------------------------------------
figure(2), title 'Vitesse en fonction du temps'

    maquette.setSpeed(ID2, speed)
    x2 = [];
    y2 = [];

    maquette.setStepPosition(ID2, 250);
    pause(5)
    maquette.setStepPosition(ID2, 800);

    timerVal = tic; %initialisation de la
    while (maquette.isMoving(ID2)) == 1 %tant que la maquette est enn mouvement
        x2 = [x2, toc]; %nous relevons le temps
        y2 = [y2, val2speed(maquette.getSpeed(ID2))];%nous relevons la vitesse en tour/min

    end
    plot(x2,y2);
```

Code : tracé de la vitesse en fonction du temps

Nous observons que la vitesse est initialement instable (grand pic, le robot passe d'un arrêt total à une vitesse très supérieure à 0), se stabilise à peu près, puis chute (arrêt du robot) :



Graphique : tracé de la vitesse en fonction du temps, dans un repère en temps (s) et vitesse (en tour/min)

Estimation de la vitesse de rotation par une méthode DF (Q2.2.15)

Calcul de la vitesse de rotation :

```
maquette.setStepPosition(ID, [POSITION,POSITION,POSITION]);
pause(3)
vitesse = [1000, 2000, 2040, 2060] * 2*pi/60; % On multiplie par 2*pi/60 pour avoir une vitesse de rotation
vitesse
```

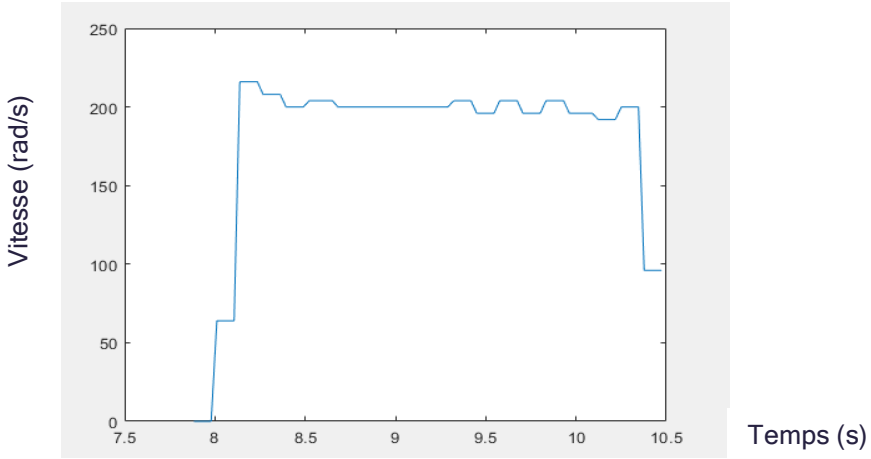
Code : formule pour le calcul de la vitesse de rotation

Application de la méthode des différences finies :

```
figure(3), title 'difference finie'
y3 = diff(y);
plot(y3)
%En utilisant une difference finie, nous trouvons la même allure de courbe
%que dans la question 12.
```

Code : application de la méthode des différences finies pour tracer la vitesse de rotation

Nous pouvons observer une allure similaire à la question 12, mais cette fois-ci « lissée » :

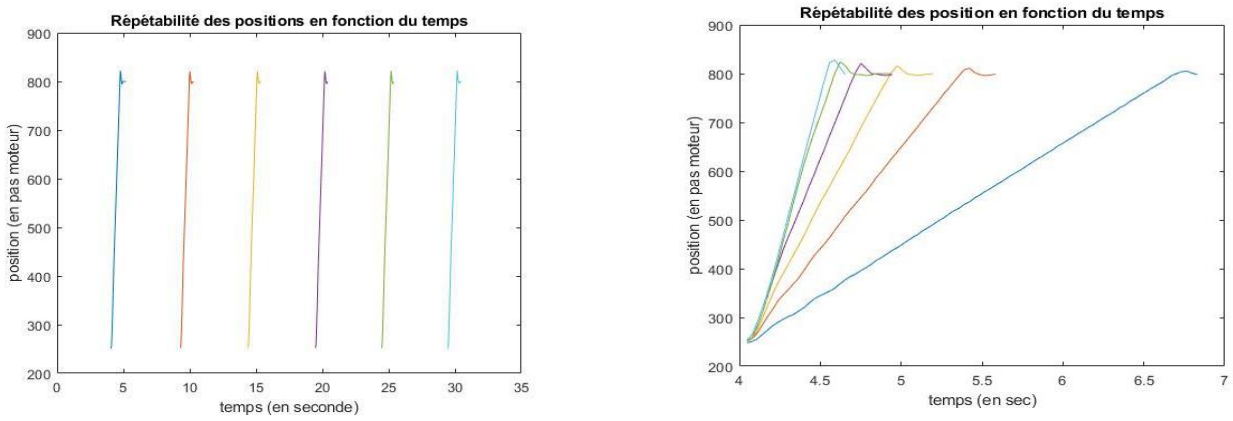


Graphique : Vitesse de rotation par une méthode DF, dans un repère en temps (s) et vitesse (rad/s)

Répétabilité du mouvement pour différentes vitesses (Q2.2.16.a)

```
% 2.2.16 : répétabilité des vitesses et positions
% ------------------------------------------------------------------------
maquette.setSpeed(ID, [100;100;100]);

maquette.setStepPosition(ID, [POSITION,POSITION,POSITION]);
pause(2)
figure(16), title 'Répétabilité des positions en fonction du temps'
speed = [100;200;300;400;500;600]; %valeur de speed à changer à chaque fois

for i = 1:6
    maquette.setSpeed(ID1, speed(i))

    x = []; %pour reinitialiser les valeurs pour chaque vitesse
    y = [];
    tic; %nous remettons le temps à 0
    timerVal = tic;
    maquette.setStepPosition(ID1, 250); % nous plaçons le moteur 1 à la position 250
    pause(4) %afin de s'assurer que le robot ai le temps de se positionner
    maquette.setStepPosition(ID1, 800); % nous essayons de mettre le robot à la position 800
     while (maquette.isMoving(ID1)) == 1 % tant que le moteur 1 bouge

        x = [x, toc]; %nous relevons le temps
        y = [y, maquette.getStepPosition(ID1)]; % nous relevons la position du moteur 2

     end
     plot(x,y); %nous affichons sur le graphiques les resultats
     hold on %on s'assure de conserver les resultats obtenues afin d'obtenir la courbe sur le même graphs
end
```

Code : Boucle pour la répétabilité des positions



Graphiques : répétabilité de la position pour une même vitesse/pour une vitesse qui accroît linéairement

Influence de la vitesse sur la précision (Q2.2.16.b)

Notre objectif était d'atteindre un pas moteur de 800. Pourtant, nous avons été au-delà avec les valeurs suivantes : 805, 811, 816, 821, 824, 828. Nous pouvons estimer l'erreur, qui augmente avec la vitesse : 5, 11, 16, 21, 28. Pour la même vitesse (400), nous obtenons une moyenne de 21,5 de pas d'erreur. Cela est dû à l'inertie du robot : il ne peut « s'arrêter » et dépasse la limite. Sur le graphique, cela se traduit par une légère bosse à la fin de chaque courbe.

Utilisation du moteur en début de chaîne cinématique

Fonction Limit (Q2.2.17)

La fonction limite permet de prendre en compte les limites physiques du robot. Elle peut être pratique pour tester si celui-ci « dépasse malgré lui » les limites imposées :

```
function new_pos = limit(pos,val_min,val_max)
% LIMIT(POS,VAL_MIN,VAL_MAX) limite les valeurs de position dans le
% vecteur POS entre une valeur minimale VAL_MIN et maximale VAL_MAX
% autorisée par la geometrie du robot.

new_pos = zeros(1,length(pos)); %vecteur de taille 1*le nombre de position donné en argument

for i = 1:length(pos) %pour chaque position
    if pos(i) <= val_min %si la pos. donné est <= à la pos. de butée minimum
        new_pos(i) = val_min; %on rend la position de butée minimum
    elseif pos(i)>= val_max %si la pos. donné est >= à la pos. de butée minimum
        new_pos(i) = val_max; %on rend la position de butée minimum
    else %sinon
        new_pos(i) = pos; %la position est atteignable : on rend la position demandée
    end
end
```

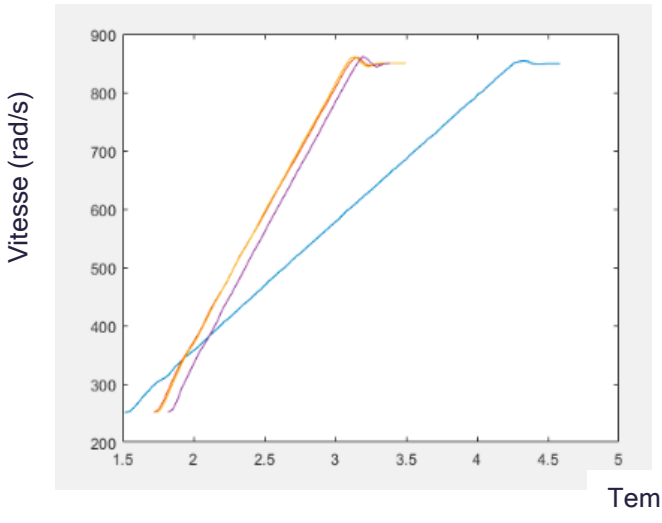
*Code : fonction **limit***

Nouvelle estimation de l'évolution de la position pour différentes vitesses et conséquences sur le tracé (Q2.2.19.a/b)

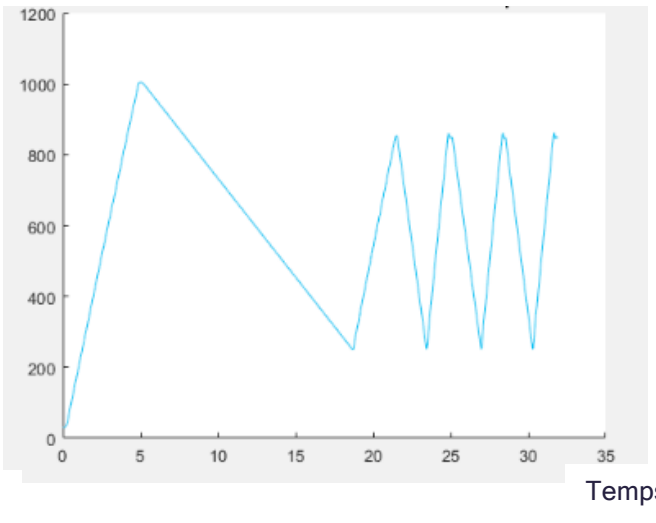
```
% 2.2.19 : tracé des positions en fonction du temps pour le 1er moteur
% --------------------------------------------------------------------------

maquette.setStepPosition(ID, [POSITION,POSITION,POSITION]);
pause(3)
vitesse = [1000, 2000, 2040, 2060] * 2*pi/60; % On multiplie par 2*pi/60 pour avoir une vitesse de rotation
vitesse

figure(4), title 'Question 19 : Position en fonction du temps'

for i = 1:length(vitesse)
    x = [];
    y = [];
    tic;
    timerVal = tic;
    maquette.setSpeed(ID1, vitesse(i));

    maquette.setStepPosition(ID1, 250); % nous plaçons le moteur 1 à la position initiale
    while (maquette.isMoving(ID1)) == 1
    pause(0.1)
    end
    %afin de s'assurer que le robot ai le temps de se positionner
    maquette.setStepPosition(ID1, 850); % nous essayons de mettre le robot à la position finale

    while (maquette.isMoving(ID1)) == 1 % tant que le moteur 1 bouge

        x = [x, toc]; %nous relevons le temps
        y = [y, maquette.getStepPosition(ID1)]; % nous relevons la position du moteur 1
    end
    plot(x,y);
    hold on
end
```

Code : boucle pour tracer la position pour différentes vitesses et le tracé de la vitesse pour différentes positions

Nous pouvons observer qu'il est difficile de retracer exactement la même trajectoire : certains facteurs extérieurs sont négligés dans notre système, comme les frottements avec le bâti de la maquette par exemple. Nous avons également observé le tracé de la vitesse : pour une vitesse de rotation très grande, la vitesse semble se stabiliser.



Graphique : tracé d'une même trajectoire avec vitesses différentes (repère en seconde et rad/s).



Graphique : tracé de l'accroissement de la vitesse (repère en seconde et rad/s).

BIBLIOGRAPHIE

Ce compte-rendu a été réalisé à l'aide de différentes ressources :

Documentation théorique

- Cours de LU3MEE01 de M. Sinan HALIYO
- Enoncés de travaux pratiques
- Documentation MATLAB avec Formulaire et Annexe

Images et outils

- Photos effectuées dans la salle 227 du bâtiment Esclangon de Sorbonne Université Jussieu
- Emprunt d'une maquette robot 3R avec moteurs DYNAMIXEL, protocole TTL et câble USB2DYNAMIXEL
- Logiciel MATLAB édition 2023 (sur ordinateurs personnels) et édition 2021 (sur ordinateurs du bâtiment Esclangon).
- Compte-rendu fait sur Word édition 2021 (ordinateur personnel)