

Polytechnique Montréal

**Rapport #19 – Version finale**

par

Lenz Myshael Kenley Joseph (2324171)

Marc-Antoine Pilon (2384987)

Thomas Sancho (2353746)

MEC1315: Technologies informationnelles en ingénierie

Prof. Richard Phat The Nguyen

24 octobre 2025

# Analyse critique du script Simulateur\_19.py

par ChatGPT-5 (forfait gratuit) – 16 octobre 2025

## Requis:

Lecture d'un fichier de paramètres (Robot.par), lecture d'un fichier de trajectoire articulaire (Trajet.trj), calcul des positions successives du robot 3R par cinématique directe, et génération d'une animation graphique (avec enregistrement GIF).

---

## Extrait(s) significatif(s) du code

Voir la fonction simuler\_robot dans le script Simulateur\_19.py

---

## Points positifs

### 1. Structure claire et logique

- Le code suit une séquence logique et intuitive :  
**Lecture → Calcul → Animation → Sauvegarde → Affichage.**
- La séparation en fonctions (lire\_fichier\_parametres, calculer\_points\_articulations, mise\_a\_jour) améliore la lisibilité.
- Les noms de variables sont explicites et en français, ce qui facilite la compréhension pour un contexte académique.

### 2. Utilisation efficace de NumPy et Matplotlib

- L'emploi de **NumPy** pour les calculs trigonométriques et vectoriels est optimal.
- L'animation **FuncAnimation** est utilisée correctement avec blit=True pour limiter les redessinages et améliorer les performances. Cela fluidifie le tout.
- Les listes path\_x et path\_y permettent de conserver la trace du bout du bras et d'afficher la trajectoire cartésienne.

### 3. Vérification visuelle dynamique

-   -> La couleur du bras (bleu = conforme / rouge = hors limites) fournit un **feedback immédiat** sur le respect des contraintes articulaires.

### 4. Lecture robuste des fichiers

- Le code ignore les lignes vides ou commentées (#) et valide le nombre d'entrées dans Robot.par. Il assigne aussi les paramètres à un dictionnaire, ce qui augmente la lisibilité.
  - L'usage de np.atleast\_2d() dans lire\_fichier\_trajet évite les erreurs de dimension pour un unique point de trajectoire.
-

## ⚠️ Problèmes / Risques

### 1. Gestion rigide du format des fichiers

- Le script suppose un format strict de **23 valeurs numériques** dans Robot.par. Si l'ordre ou le nombre change, la fonction échoue sans explication détaillée.

👉 *Solution* : afficher un message d'erreur contextualisé :

```
raise ValueError(f"Fichier {nom_fichier} incomplet : {len(valeurs)} valeurs lues (23 attendues)")
```

### 2. Accumulation illimitée de points dans la trajectoire

- Les listes path\_x et path\_y grandissent indéfiniment, ce qui peut ralentir les animations longues.

👉 *Solution possible* : tracer seulement les N derniers points :

```
N = 500  
ligne_trajet.set_data(path_x[-N:], path_y[-N:])
```

### 3. Redondance dans la mise à jour graphique

- path\_x et path\_y sont recalculés à chaque frame alors qu'ils pourraient être mis à jour plus efficacement.

👉 *Optimisation possible* : stocker la trajectoire complète avant l'animation.

### 4. Absence de gestion d'erreurs d'E/S

- Aucune protection contre les fichiers manquants (Robot.par, Trajet.trj) ou illisibles.

👉 *Ajout recommandé* :

```
try:  
    angles_trajet = lire_fichier_trajet(nom_trajet)  
except OSError as e:  
    print(f"Erreur : impossible de lire {nom_trajet} ({e})")  
    return
```

---

## 💡 Améliorations possibles

### 1. Paramétrage utilisateur

Permettre à l'utilisateur de facilement modifier certains paramètres de l'animation, comme:

- Le nombre d'images par seconde (fps) ;
- La couleur du bras ou de la trajectoire ;
- Le titre du graphique (possibilité d'annotation)

Exemple:

```
def simuler_robot(..., fps=30, couleur_robot="blue", couleur_trajet="red", titre=None)
```

## 2. Messages informatifs

Inclure des messages de progression:

```
print(f"Lecture du fichier {nom_par}... OK")
print(f"Simulation en cours ({len(angles_trajet)} étapes)..."")
```

## 3. Modularité accrue

Créer une classe:

```
class Robot3R:
    def __init__(self, fichier_par):
        self.params = lire_fichier_parametres(fichier_par)
    def animer(self, fichier_trajet, nom_gif=None):
        ...
```

→ Cela permettrait de simuler plusieurs robots simultanément ou de comparer différentes trajectoires.

## 4. Option “aperçu rapide”

Ajouter un mode de visualisation instantané (sans sauvegarde GIF) pour tester une configuration avant le rendu final.

## 5. ⏱️ Synchronisation réaliste (temps réel)

Actuellement, le paramètre interval=dt contrôle uniquement le **délai d'affichage graphique**, non la correspondance avec le temps physique.

→ Le simulateur **n'évolue pas à vitesse réelle** si le CPU ou le GPU est occupé

⚙️ En introduisant une synchronisation temporelle dynamique à la simulation avec une boucle basée sur le module *time*, cela garantirait que 1 seconde simulée = 1 seconde réelle, et permettrait:

- Une représentation encore plus fidèle à la réalité du mouvement du robot ;
- La validation temporelle des trajectoires

## 6. 📈 Journalisation analytique (enregistrement CSV)

Pour des analyses d'ingénierie (cinématique, vérification de contraintes, validation du générateur), il serait utile de **consigner les états articulaires et cartésiens à chaque frame** :

- angles  $\theta_1, \theta_2, \theta_3$  ;
- position (x, y) du bout ;
- statut “dans les limites?” ;
- éventuellement le temps réel écoulé.

L'enregistrement de ces données pourrait être fait dans un fichier CSV, facilement accessible après l'exécution du programme:

```

import csv

# Avant la boucle d'animation :
with open("journal_simulation.csv", "w", newline="", encoding="utf-8") as f:
    writer = csv.writer(f)
    writer.writerow(["t(ms)", "θ1(rad)", "θ2(rad)", "θ3(rad)", "x", "y", "dans_limite"])

    t = 0
    for i, angles in enumerate(angles_trajet):
        points = calculer_points_articulations(angles, L1, L2, L3, x_base, y_base)
        x, y = points[-1]
        dans_limite = np.all((angles >= limites_min) & (angles <= limites_max))
        writer.writerow([t, *angles, x, y, dans_limite])
        t += dt

```

---

## Conclusion

Le script **Simulateur\_19.py** est **fonctionnel, didactique et bien structuré**.

Il offre une animation claire du mouvement du bras 3R, en intégrant la vérification visuelle des limites articulaires.

Cependant :

- il repose sur un format de fichiers strict,
- et il manque de gestion d'erreurs et d'optimisation mémoire.

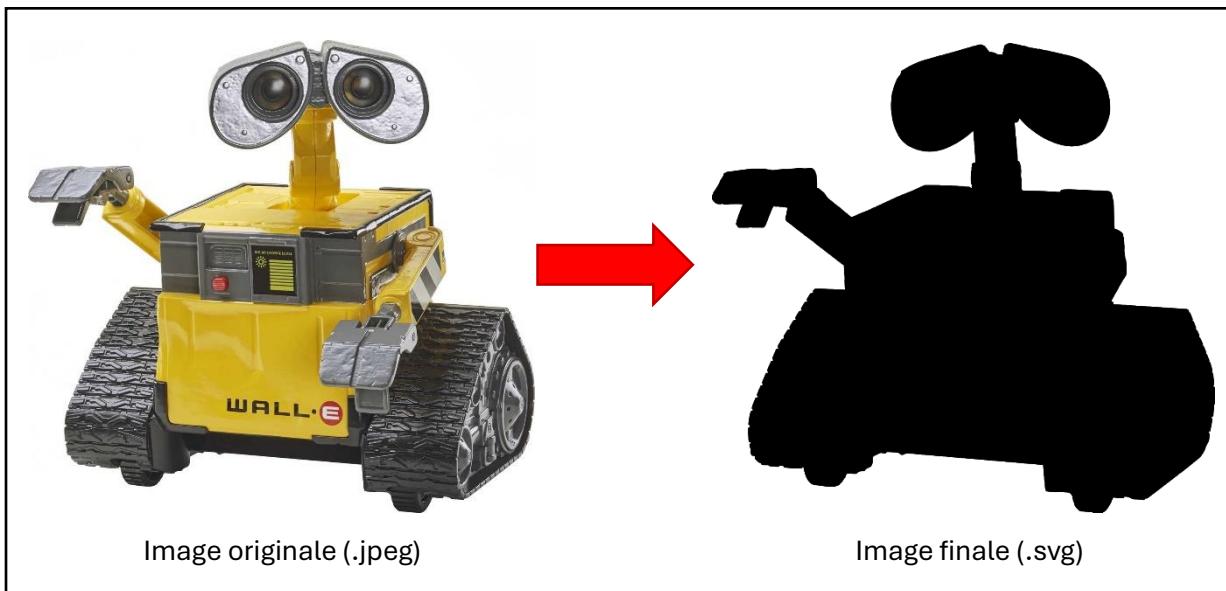
 **Dans son état actuel**, le simulateur répond pleinement aux exigences du projet TS1, mais un léger travail de **robustesse et de modularité** le rendrait prêt pour une utilisation plus large (comparaison de trajectoires, contrôle en boucle fermée, etc.).

## Présentation de l'exemple robot/trajectoire personnalisé

La trajectoire choisie comme exemple dessine le contour de notre petit robot préféré: Wall-E. Avant de pouvoir produire un tel tracé cependant, plusieurs manipulations ont dû être effectuées.

En effet, nous sommes initialement partis d'une image « .jpeg » de Wall-E. Cette image a ensuite été détournée, peinte en noir, puis éventuellement convertie en fichier « .svg » (vector graphics) à l'aide du logiciel *Affinity Photos*, une plateforme analogue à *Photoshop*. Ce procédé peut être observé à la figure 1.

Figure 1: Transformation de wall-e.jpeg en format graphique vectoriel



Ensuite, à l'aide d'un script produit par ChatGPT, le contour de cette image vectorielle a été converti en une suite de points avec des coordonnées x-y, puis le tout a été transféré dans le fichier « Trajet.xy ».

Voici le script utilisé pour faire cette conversion:

```
import sys
import numpy as np
import matplotlib.pyplot as plt
from svgpathtools import svg2paths2

# === Adjustable normalization box ===
X_MIN, X_MAX = 0.5, 1.5
Y_MIN, Y_MAX = 0.5, 1.5
# =====

def sample_path(path, num_points=100):
    """Return (x, y) coordinates sampled evenly along the path."""
    points = []
    for seg in path:
```

```

        for t in np.linspace(0, 1, num_points):
            pt = seg.point(t)
            points.append((pt.real, pt.imag))
    return points

def svg_to_points(svg_file, num_points=100):
    """Convert SVG file to a list of (x, y) coordinate points."""
    paths, attributes, svg_attributes = svg2paths2(svg_file)
    all_points = []
    for path in paths:
        all_points.extend(sample_path(path, num_points))
    return np.array(all_points)

def normalize_points(points, x_range=(X_MIN, X_MAX), y_range=(Y_MIN, Y_MAX)):
    """Normalize points to fit within the given box and flip Y-axis."""
    xs, ys = points[:,0], points[:,1]
    x_min, x_max = np.min(xs), np.max(xs)
    y_min, y_max = np.min(ys), np.max(ys)

    # Normalize to [0, 1]
    xs_norm = (xs - x_min) / (x_max - x_min)
    ys_norm = (ys - y_min) / (y_max - y_min)

    # Flip Y so image isn't upside down
    ys_norm = 1 - ys_norm

    # Scale to desired box
    xs_scaled = x_range[0] + xs_norm * (x_range[1] - x_range[0])
    ys_scaled = y_range[0] + ys_norm * (y_range[1] - y_range[0])

    return np.column_stack((xs_scaled, ys_scaled))

def plot_points(points, title="Preview of normalized SVG"):
    """Display a scatter plot of the sampled points."""
    plt.figure(figsize=(6,6))
    plt.scatter(points[:,0], points[:,1], s=1)
    plt.title(title)
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.axis("equal")
    plt.grid(True, linestyle='--', alpha=0.4)
    plt.show()

def save_points_xy(points, output_file="trajet.xy"):
    """Save points to a file in x y format (space separated)."""
    with open(output_file, "w") as f:
        for x, y in points:
            f.write(f"{x:.6f} {y:.6f}\n")
    print(f"Saved {len(points)} normalized points to {output_file}")

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Usage: python svg_to_points_xy_preview.py <file.svg> [num_points]")
    sys.exit(1)

```

```

svg_file = sys.argv[1]
num_points = int(sys.argv[2]) if len(sys.argv) > 2 else 100

# 1. Read and sample paths
points = svg_to_points(svg_file, num_points)

# 2. Normalize + flip vertically
points_norm = normalize_points(points)

# 3. Plot preview
plot_points(points_norm)

# 4. Save to .xy file
save_points_xy(points_norm)

```

Il est à noter que ce script emploi le module `svg2paths2` de la bibliothèque `svgpath`, bibliothèque qui sort quelque peu du cadre du cours. Cependant, puisque dans les consignes du travail, la méthode employée pour préparer le fichier « Trajet.xy » ne semble pas être explicitement évaluée, nous avons jugé qu'il était approprié d'utiliser cette bibliothèque.

Fait intéressant: le script est capable de normaliser les points de l'image « .svg » à l'intérieur d'un petit carré, ce qui nous permet de spécifier la zone dans laquelle nous voulons que le robot travaille.

De plus, avant de pondre le fichier « Trajet.xy », le script doit aussi s'assurer d'inverser les coordonnées en y, car le système de coordonnées pour les fichiers « .svg » utilise un axe des y qui croît vers le bas.

Pour ce qui est du fichier « Robot.par », ce dernier n'a subi que quelques petites modifications comparées à l'exemple fourni dans l'énoncé du TS1. Le changement majeur réside dans les angles de départ du robot. En effet, à l'aide d'un petit script de test, les angles de départ ont été manuellement ajustés afin que le robot puisse partir du premier point dans le fichier « Trajet.xy ». Enfin, les longueurs de bras du robot ont été légèrement modifiées.

Voici le script utilisé pour trouver les angles de départ:

```

from Generateur_19 import read_robot_par, reach, forward_kinematics
import numpy as np

# Charger le robot
robot = read_robot_par("Robot.par")

# Point XY cible
target_xy = np.array([1.085510, 1.498571])

# Angle de départ
theta_init = robot["theta_dep"].copy()

# Trouver les angles correspondants
theta_sol = reach(target_xy, theta_init, robot, avoidance=False, clamp=True)

print("Angles trouvés (rad):", theta_sol)
print("Angles trouvés (deg):", np.degrees(theta_sol))
print("Position obtenue :", forward_kinematics(theta_sol, robot))

```

# Explication de la méthode d'évitement des limites articulaires dans le script Generateur\_19.py

par ChatGPT-5 (forfait gratuit) – 16 octobre 2025

## 1. Structure générale de l'algorithme

L'ensemble du mécanisme d'évitement est contenu dans la fonction :

```
def reach(p_target, theta, robot, avoidance=True, clamp=True):
```

Cette fonction représente **le cœur du générateur**: elle calcule comment ajuster les angles articulaires theta pour que le bout atteigne la position cartésienne cible p\_target.

Deux éléments importants y interviennent :

- le **terme d'évitement**, qui suit la méthode décrite dans l'énoncé du TS1 (actif si avoidance=True),
- et le **clamp**, qui correspond à la saturation des angles aux articulations (actif si clamp=True).

Ce sont ces deux éléments qui régissent le comportement du robot lorsqu'il est aux abords de ses limites articulaires.

---

## 2. Calcul du déplacement articulaire principal

La première étape consiste à calculer le déplacement minimal nécessaire pour atteindre la position voulue :

```
J = jacobian(theta, robot)
Jp = pseudo_inverse(J)
dtheta = Jp @ (d * dp)
```

- J : la matrice Jacobienne du robot (décrise dans la fonction jacobian()),
- Jp : le pseudo-inverse gauche (calculé par pseudo\_inverse()),
- dp = p\_target - p : l'erreur cartésienne actuelle,
- dtheta : le déplacement articulaire minimal.

Ce calcul seul donne un **suivi précis de la trajectoire**, mais sans se soucier des limites articulaires.

---

## 3. Ajout du terme d'évitement (null-space correction)

Lorsque avoidance=True, le code ajoute un deuxième terme à dtheta :

```

if avoidance:
    h = W @ (theta_mid - theta)
    I = np.eye(3)
    dtheta_H = (I - Jp @ J) @ h
    dtheta = dtheta + alpha * dtheta_H

```

Explications :

- $\theta_{\text{mid}}$  est la **position médiane** entre les bornes min/max de chaque articulation.
- $W$  est une matrice diagonale donnant **plus d'importance aux articulations ayant une plage réduite**.
- $h$  est le **vecteur de direction** ramenant les articulations vers leur centre.
- $(I - J_p @ J)$  est le **projecteur sur le noyau du Jacobien** : il garantit que ce déplacement ne modifie pas la position cartésienne du robot.

Le facteur  $\alpha$  règle la force de cette correction (ici 0.2).

Ainsi, lorsque le bras approche d'une limite, cette correction crée un **léger biais** qui le ramène vers une configuration centrale, sans dévier de la trajectoire.

Il est à noter que le terme d'évitement n'est pas une contrainte dure, et correspond plutôt à une optimisation douce :

- tant que le mouvement principal ( $J^T \Delta p$ ) n'est pas trop exigeant, le terme d'évitement garde les angles à distance des bornes ;
- mais si la position cartésienne demandée **nécessite** de s'approcher (ou d'atteindre) une limite, le robot le fera.

Même avec l'évitement activé, il est donc toujours possible que le robot touche ses limites articulaires, or plus on augmente le  $\alpha$ , moins il est probable qu'un contact se produise, l'évitement devenant alors de plus en plus répulsif.

---

## 4. Mise à jour et clamp

Après chaque calcul de  $d\theta$ , les angles sont mis à jour :

```
theta = theta + d * dtheta
```

Puis, si le **clamp** est activé (clamp=True), les nouvelles valeurs sont **saturées** dans leurs bornes physiques :

```

if clamp:
    theta = np.minimum(np.maximum(theta, theta_min), theta_max)

```

Ce mécanisme agit comme une **barrière de sécurité** :

- empêche tout dépassement mécanique,
- protège le calcul contre des valeurs irréalistes (par ex. inversion de bras),

- stabilise le comportement numérique à proximité des limites.

Si clamp=False, ces lignes ne s'exécutent pas, et le robot peut **dépasser ses limites** (utile pour des comparaisons théoriques).

---

## 5. Paramétrage flexible

Grâce aux deux drapeaux, il est possible de facilement basculer entre quatre comportements :

Cas	AVOIDANCE	CLAMP	Comportement
 Mode nominal	True	True	L'évitement maintient les articulations loin des limites, et le clamp garantit qu'aucune ne les dépasse.
 Mode sans évitement	False	True	Le robot suit strictement la trajectoire minimale, mais reste confiné aux bornes grâce au clamp.
 Mode libre total	False	False	Aucune contrainte : le robot peut dépasser les limites. Utile pour des tests ou comparaisons théoriques.
 Mode rare	True	False	Le champ d'évitement agit encore, mais les limites ne sont plus strictement imposées. Peu utilisé en pratique.

---

## 6. Résultats et conclusion

Dans le contexte de notre exemple, l'évitement des limites articulaires provoque une légère déviation du trajet cartésien; déviation qui est particulièrement prononcée au niveau des chenilles de Wall-E. Cela est normal, et s'explique relativement facilement.

### Ce qui se passe réellement

Quand on ajoute l'évitement articulaire, la mise à jour de mouvement passe de :

$$\Delta\theta = J^+ \Delta p$$

à :

$$\Delta\theta = J^+ \Delta p + \alpha(I - J^+ J)h$$

où :

- Le premier terme  $J^+ \Delta p$  fait avancer l'effecteur selon la trajectoire cartésienne voulue.
- Le second terme  $(I - J^+ J)h$  pousse les articulations vers leurs positions moyennes, mais dans le **noyau de J** — donc, en théorie, sans affecter la position du bout du bras.

Cependant — et c'est la subtilité — cette nullité n'est **que théorique** : elle n'est exacte que pour des déplacements infinitésimaux et une linéarisation parfaite.

Dans la simulation numérique (avec des pas finis et un amortissement), cette propriété n'est qu'approximative.

Ainsi, à mesure que la simulation intègre des centaines de petits pas, des **minuscules écarts** s'accumulent, et la trajectoire de l'effecteur **dérive légèrement** du trajet parfait (un contour de Wall-E exacte).

C'est donc ce qu'on peut observer le long des chenilles de Wall-E : le terme d'évitement “s'oppose” légèrement au suivi cartésien. La déviation se produit spécifiquement au niveau des chenilles, simplement parce que c'est dans cette région de l'espace qu'une ou plusieurs articulations sont poussées près de leurs limites angulaires.

## Contribution des membres de l'équipe

Membre	Description de la tâche
Lenz	<ul style="list-style-type: none"><li>Écriture complète du script pour le simulateur (Simulateur_19.py) et ce, sans aucune utilisation de l'IA</li><li>Contribution partielle à l'écriture du rapport, particulièrement dans la section « Explication de la méthode d'évitement [...] »</li></ul>
Marc-Antoine	<ul style="list-style-type: none"><li>Écriture collaborative du script pour le générateur (Generateur_19.py) avec l'aide de l'IA</li><li>Contribution majeure à l'écriture du rapport final</li><li>Remise du travail sur Moodle</li></ul>
Thomas	<ul style="list-style-type: none"><li>Écriture collaborative du script pour le générateur (Generateur_19.py) avec l'aide de l'IA</li><li>Mise en page du rapport final et contribution partielle à son écriture</li><li>Écriture des exemples de fichiers robot/trajectoire personnalisés avec l'aide de l'IA</li></ul>

## Annexe

### Rapport de conversation de co-conception assisté par l'IA par ChatGPT-5 (forfait gratuit) – 16 octobre 2025

#### \*Note importante:

Lenz a effectué l'écriture du script pour le simulateur (Simulateur\_19.py) sans aucune assistance de l'intelligence artificielle. Il n'y a donc aucune conversation de co-conception avec l'IA disponible dans son cas.

Cependant, pendant que Lenz s'activait à finaliser son script, Marc-Antoine et Thomas ont, de leur côté, demandé à ChatGPT de créer le squelette d'un simulateur, afin de pouvoir visualiser la trajectoire obtenue par le script de leur générateur, déjà terminé à ce moment-là.

Voici donc cette conversation:

#### Q01 – Vous avez dit:

- 📁 Uploaded a file [TS1\_A2025.pdf]
- 📁 Uploaded a file [Robot.par]
- 📁 Uploaded a file [Trajet.xy]

Can you look at this and tell me what you think? The pdf file gives the context, and the “.par” and “.xy” files are the data you need

#### R01 – ChatGPT a dit:

Perfect — I've reviewed everything.

Here's a quick summary of what you've provided and what it all means for your project **MEC1315 – Travail Synthèse 1 (TS1)**:

---

#### 📘 PDF (TS1\_A2025.pdf)

This is the **assignment guide**. It defines two main deliverables:

#### 1. Simulateur graphique (30%)

You must:

- Read robot parameters from a .par file (e.g., Robot.par).
- Read a joint trajectory .trj file (generated later).
- Animate the 3R planar robot using Matplotlib.
- Display:

- The robot (blue if within limits, red if exceeding limits),
- The wall (from xmur, ymur values),
- The end-effector trajectory (red dashed line),
- Properly scaled and labeled axes.

## 2. Générateur de trajectoires (30%)

You must:

- Read the robot parameters (Robot.par).
- Read a Cartesian trajectory (Trajet.xy).
- Compute the corresponding joint trajectory (Trajet.trj) using:
  - The **velocity equation**:  
 $\dot{p} = J\theta' \cdot \dot{\theta}$
  - The **pseudoinverse** for inverse kinematics:  
 $\Delta\theta M = JT(JJT)^{-1} \Delta p$   
 $\Delta\theta M = JT(JJT)^{-1} \Delta p$
- Implement a **limit-avoidance strategy** using the null space component:  
 $\Delta\theta H = (I - JT(JJT)^{-1})h$   
 $\Delta\theta H = (I - JT(JJT)^{-1}J)h$   
with  $h = W(\theta^* - \theta)$   
 $h = W(\bar{\theta} - \theta)$

## 3. Deliverables

You'll need:

- Scripts: Generateur\_#.py, Simulateur\_#.py
- Example data: Robot\_#.par, Trajet\_#.xy, Trajet\_#.trj
- Rapport\_#.pdf
- Two GIFs (with/without avoidance).

### ⚙️ Robot.par

This defines the **3R robot's geometry and limits**.

It includes:

- Link lengths (L1, L2, L3)
- Graph limits (xmin, xmax, etc.)
- Base coordinates (x0, y0)
- Initial angles (t1\_dep, t2\_dep, t3\_dep)
- Joint limits ( $t^*_\text{min}$ ,  $t^*_\text{max}$ )
- Wall coordinates (xmur, ymur)
- Time step (dt)

This file is your robot's configuration — your code will parse it line by line to set all these variables.

---

### Trajet.xy

This defines a **Cartesian trajectory** for the robot's end-effector — i.e., the path in the XY-plane that the tip of the robot must follow.

Each line is a point:

x y

For example, the included data probably defines a **square path**, as in the assignment example.

---

### What you'll need to do next

To start:

1. **Write a parser** to load Robot.par into a dictionary or dataclass.
2. **Write a parser** to load Trajet.xy (list of (x, y) points).
3. Implement the **forward kinematics** (Eq. 1).
4. Implement the **Jacobian** (Eq. 9).
5. Implement the **inverse kinematics via pseudoinverse** (Eq. 3).
6. Discretize line segments between each Cartesian point and use Algorithm 1 from the PDF to step through the trajectory.
7. Save the resulting joint positions into a .trj file.

Once you have that, you can move on to the **simulator**, which will:

- Read .par and .trj
  - Compute forward kinematics for each frame
  - Animate with Matplotlib's FuncAnimation
- 

If you'd like, I can **start building you a working base version** of:

- Generateur.py (trajectory generator)  
or
- Simulateur.py (robot animation)

Which one would you like to tackle first?

## Q02 – Vous avez dit:

Can you create an extremely barebones simulator please? Only display the robot. I just want to see what the motion looks like when we use your code and obtain a traj file.

## R02 – ChatGPT a dit:

Perfect idea💡 — that's **exactly** what you should do next.

Before you build the full simulator with colors, walls, and end-effector traces, it's wise to start with a **minimal visualizer** that just shows whether your Generator.py produced a *sensible motion*.

Below is an **extremely barebones simulator** script.

It will:

- ✓ Read your Robot.par file
- ✓ Read your generated Trajet.trj file
- ✓ Plot the robot links for each frame using matplotlib.animation.FuncAnimation
- ✓ Use equal x–y scaling so geometry looks realistic

Nothing fancy — no colors, no labels, no walls, no trajectory line — just the robot moving.

---

### ✿ Simulateur\_Barebones.py

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
# -----
# Very barebones simulator for MEC1315 - Robot 3R planaire
# -----


def read_robot_par(filename):
    params = []
    with open(filename, "r") as f:
        for line in f:
            line = line.strip()
            if not line or line.startswith("#"):
                continue
            params.append(float(line.split()[0]))

    (
        L1, L2, L3,
        xmin, xmax, ymin, ymax,
        x0, y0,
        t1_dep, t2_dep, t3_dep,
        t1_min, t1_max, t2_min, t2_max, t3_min, t3_max,
        xmur0, xmur1, ymur0, ymur1,
        dt
    ) = params

    return {
```

```

    "L1": L1, "L2": L2, "L3": L3,
    "x0": x0, "y0": y0,
    "xmin": xmin, "xmax": xmax,
    "ymin": ymin, "ymax": ymax,
    "dt": dt
}

def read_trj(filename):
    return np.loadtxt(filename)

# Forward kinematics (returns joint coordinates)
def forward_kinematics(theta, robot):
    L1, L2, L3 = robot["L1"], robot["L2"], robot["L3"]
    t1, t2, t3 = theta
    x0, y0 = robot["x0"], robot["y0"]

    x1 = x0 + L1 * np.cos(t1)
    y1 = y0 + L1 * np.sin(t1)
    x2 = x1 + L2 * np.cos(t1 + t2)
    y2 = y1 + L2 * np.sin(t1 + t2)
    x3 = x2 + L3 * np.cos(t1 + t2 + t3)
    y3 = y2 + L3 * np.sin(t1 + t2 + t3)

    return np.array([[x0, x1, x2, x3],
                    [y0, y1, y2, y3]])

# -----
# MAIN
# -----


if __name__ == "__main__":
    robot = read_robot_par("Robot.par")
    traj = read_trj("Trajet.trj")

    fig, ax = plt.subplots()
    ax.set_aspect("equal")
    ax.set_xlim(robot["xmin"], robot["xmax"])
    ax.set_ylim(robot["ymin"], robot["ymax"])
    line, = ax.plot([], [], 'o-', lw=3)

    def init():
        line.set_data([], [])
        return line,

    def update(i):
        coords = forward_kinematics(traj[i], robot)
        line.set_data(coords[0, :], coords[1, :])
        return line,

    anim = FuncAnimation(
        fig, update, init_func=init,
        frames=len(traj), interval=robot["dt"], blit=True
    )

    plt.show()

```