Computational Engineering und Robotik

TECHNISCHE UNIVERSITÄT DARMSTADT

Sommersemester 2022, Homework 3

Prof. J. Peters, K. Ploeger, K.Hansel, D. Palenicek, F. Al-Hafez und T. Schneider

Total points: 15

Abgabefrist: 11:59, Montag, 4 Juli 2022

3.1 Steuerung und Regelung [15 Points]

In der letzten Übung haben wir uns mit der physikalischen Simulation unseres Roboterarms auseinandergesetzt. Nun wollen wir uns damit befassen, den Roboter so anzusteueren, dass dieser sich verhält wie wir das wünschen. Konkret sind wir daran interessiert, den Roboter auf Kommando an bestimmte Gelenkpositionen fahren zu lassen oder sogar bestimmten Trajektorien folgen zu lassen. Wir werden im Verlauf dieser Übung sehen, dass wir dabei einige Herausforderungen überwinden müssen, die der Trägheit des Roboters und der Gravitation geschuldet sind.

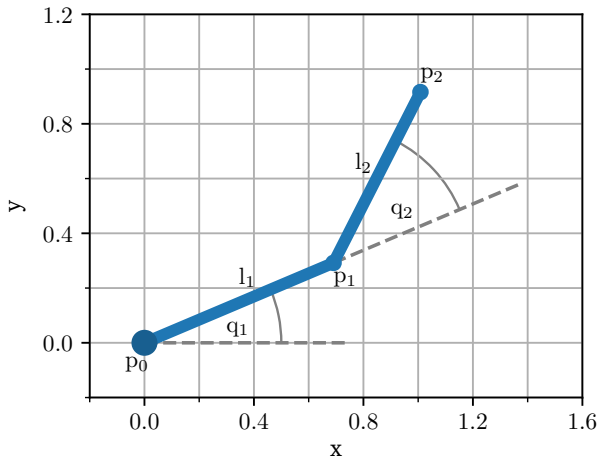

Figure 1: Visualisierung des Roboterarms. p_0 ist die Basis des Arms, p_1 das Ellenbogengelenk und p_2 der Endeffektor.

Aufsetzen der Programmierumgebung
Falls du das nicht bereits schon gemacht hast, folge den Anweisungen auf Übungsblatt 1, um eine Programmierumgebung aufzusetzen und zu aktivieren. Installiere anschließend die benötigten Pakete mittels

```
pip install /path/to/cer_pex3_lib
```

und starte wie gewohnt deinen *Jupyter*-Server. Es gelten die selben Hinweise wie auf Übungsblatt 1.

1. **Implementierung eines PD-Reglers [2 Points]**

   Wir wollen zunächst versuchen, den Roboter mittels eines PD-Reglers an beliebigen Gelenkpositionen zu stabilisieren. Implementiere dazu die Klasse

   ```
   PDController(p, d)
   ```

   welche als Eingabeargumente die Reglerkoeffizienten $p \in \mathbb{R}^2$ und $d \in \mathbb{R}^2$ erhält. Diese sind so zu verstehen, dass p_1, d_1 die Reglerkoeffizienten für Gelenkwinkel q_1 und p_2, d_2 the Reglerkoeffizienten für Gelenkwinkel q_2 sind.
   Innerhalb der Klasse `PDController(p, d)` ist lediglich die Funktion

   ```
   M = PDController.__call__(q, dq, q_target, dq_target, dt)
   ```

   zu implementieren. Diese erhält als Eingabe die momentanen Gelenkwinkel $q \in \mathbb{R}^2$ und Gelenkgeschwindigkeiten $\dot{q} \in \mathbb{R}^2$, sowie die Zielgelenkwingel $q^* \in \mathbb{R}^2$ und Zielgelenkgeschwindigkeiten $\dot{q}^* \in \mathbb{R}^2$. Der Systemzeitschritt δt kann hier ignoriert werden, dieser ist lediglich für die Implementierung des PID-Reglers in Aufgabe 3 erforderlich. Ausgegeben werden von der Funktion die Drehmomente $M \in \mathbb{R}^2$, die der Roboter an seinen Motoren aufbringen soll.
   Beachte, dass wir in der Vorlesung bei der Einführung des PD-Reglers angenommen haben, dass die Zielgeschwindigkeit $\dot{q}^*$ immer Null ist. Wollen wir jedoch mit unserem Regler einer Trajektorie folgen anstatt einen statischen Punkt anzufahren, so ist diese Annahme nicht erfüllt. Modifiziere also den in der Vorlesung vorgestellten PD-Regler so, dass dieser Zielgeschwindigkeiten ungleich Null unterstützt.

2. **Einstellen der Parameter des PD-Reglers [1 Points]**

   Bevor wir unseren PD-Regler nutzen können, müssen wir noch Reglerkoeffizienten p und d finden, die gut mit unserem Roboter funktionieren. Implementiere dazu die Funktion

   ```
   p, d = get_pd_controller_coefficients()
   ```

   die (manuell) optimierte Koeffizienten $p, d \in \mathbb{R}^2$ für den gegebenen Roboter zurückgibt. Um das Tuning der Koeffizienten zu effizient wie möglich zu gestalten, rufe dir nochmal ins Gedächtnis, was die Aufgabe der beiden Koeffizienten jeweils ist.
   Beachte bei der Wahl dieser Koeffizienten, dass der Roboter eine Drehmomentsbegrenzung hat und Drehmomente über 50 Nm abschneidet. In realen Systemen dient dies häufig zum Schutz des Roboters und der Umgebung, in unserem Fall ist es nötig weil die Simulation nicht in der Lage ist beliebig hohe Drehmomente akkurat zu simulieren.

3. **Implementierung eines PID-Reglers [2 Points]**

   Wie wir in der vorherigen Aufgabe gesehen haben, ist der PD-Regler nicht in der Lage mit konstant wirkenden Kräften, wie Gravitation, umzugehen. Folglich wollen wir unseren PD-Regler um einen Integralteil erweitern. Implementiere dazu die Klasse

   ```
   PDController(p, i, d)
   ```

   welche als Eingabeargumente die Reglerkoeffizienten $p \in \mathbb{R}^2$, $i \in \mathbb{R}^2$ und $d \in \mathbb{R}^2$ erhält.
   Auch hier ist wieder die Funktion

   ```
   M = PDController.__call__(q, dq, q_target, dq_target, dt)
   ```

   zu implementieren, welche die selbe Signatur hat wie die Funktion des PD-Reglers.
   Da der PID-Regler im Gegensatz zum PD-Regler nicht zustandslos ist, ist es unter Umständen nötig, neue Felder im Konstruktor hinzuzufügen. Bitte stelle sicher, dass diese in der Funktion

   ```
   M = PDController.reset()
   ```

   wieder auf die Initialwerte zurückgesetzt werden.
   Verwende zur Approximation des Fehlerintegrals die Obersumme.

4. **Einstellen der Parameter des PID-Reglers [1 Points]**

   Ähnlich wie beim PD-Regler, müssen wir noch gute Reglerkoeffizienten p, i und d bestimmen. Implementiere dazu die Funktion

   ```
   p, i, d = get_pid_controller_coefficients()
   ```

   die (manuell) optimierte Koeffizienten $p, i, d \in \mathbb{R}^2$ für den gegebenen Roboter zurückgibt. Um das Tuning der Koeffizienten zu effizient wie möglich zu gestalten, rufe dir auch hier ins Gedächtnis, was die Aufgabe der drei Koeffizienten jeweils ist. Beachte auch hier, dass der Roboter eine Drehmomentsbegrenzung hat und Drehmomente über 50 Nm abschneidet.

5. **Trajektoriengenerierung [5 Points]**

   Wie wir in der letzten Aufgabe gesehen haben, ist ein PID-Regler alleine noch nicht ausreichend, um beliebige Gelenkpositionen anzufahren. In dieser Aufgabe wollen wir diesem Problem abhilfe verschaffen, indem wir für gegebene Start- und Zielpositionen, Trajektorien mit vielen Zwischenzielen berechnen. Diesen Trajektorien können wir mit unserem Regler dann folgen, um somit beliebige Gelenkpositionen erreichen zu können.
   Implementiere dazu die Funktion

   ```
   q, dq = generate_trajectory(start_pos, end_pos, v_max, a_max, dt)
   ```

   welche für eine gegebene Startposition $q_{\text{start}} \in \mathbb{R}$ und Zielposition $q_{\text{end}} \in \mathbb{R}$, sowie Geschwindigkeitslimit $\dot{q}_{\text{lim}} \in \mathbb{R}$, Beschleunigungslimit $\ddot{q}_{\text{lim}} \in \mathbb{R}$ und Zeitschritt $\delta t \in \mathbb{R}$, eine Trajektorie $(\mathbf{q}, \dot{\mathbf{q}})$ zurückgibt. Hierbei ist $\mathbf{q} = (q_1, \ldots, q_N)$ so zu verstehen, dass $q_i \in \mathbb{R}$ die Gelenkposition und $\dot{q}_i \in \mathbb{R}$ die Gelenkgeschwindigkeit ist, die der Roboter zum Zeitpunkt $i\delta t$ nach Start der Trajektorie annehmen soll. Es gilt also grundsätzlich $q_1 = q_{\text{start}}$, $q_N = q_{\text{end}}$ und $\dot{q}_1 = \dot{q}_N = 0$. Beachte, dass wir erstmal nur von einem Gelenk ausgehen, und diese Funktion dann individuell für beide Gelenke nacheinander aufrufen werden.

Die Positionen und Geschwindigkeiten der generierten Trajektorie sollen einem trapezoiden Bewegungsprofil folgen. Das heißt, der Roboter soll erst maximal beschleunigen, bis er die Maximalgeschwindigkeit erreicht hat, dann auf dieser Geschwindigkeit verweilen, und schließlich mit maximaler Beschleunigung verzögern, um genau am Ziel zur Ruhe zu kommen. Mehr Informationen zur Berechnung dieses Bewegungsprofils findest du hier.

Beachte, dass es vorkommen kann, dass der Roboter die Maximalgeschwindigkeit nicht erreicht, bevor er anfangen muss zu verzögern. In diesem Fall wird das Profil zu einem Dreieck, wie in Abbildung 2 dargestellt. Stelle sicher, dass deine Implementierung diesen Fall behandelt.

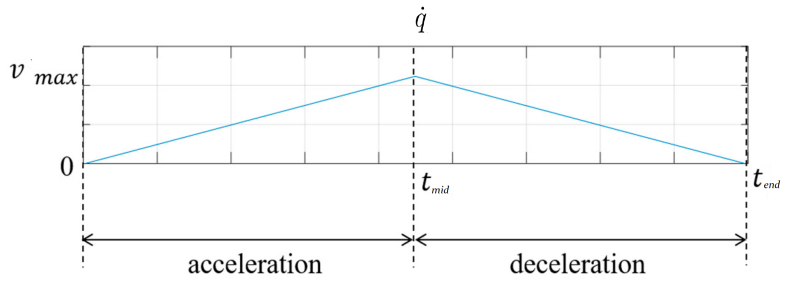

Figure 2: Qualitative Darstellung des Trajektorienverlaufs für Geschwindigkeit ($\dot{q}$). In diesem Fall wird die Zielgeschwindigkeit nicht erreicht, wodurch das triangulare Bewegungsprofil entsteht.

Hinweis: Es kann sein, dass die für die Trajektorie benötigte Zeit nicht exakt durch δt teilbar ist. In dem Fall kommt der Roboter an seinem Ziel an, bevor der letzte Zeitschritt vorbei ist. Stelle hier bitte einfach sicher, dass die Endgeschwindigkeit 0 ist und die Endposition mit der Zielposition übereinstimmt.

6. **Trajektoriengenerierung im Toolspace [2 Points]**

Wir stehen nun vor einem ähnlichen Problem wie in Programmierübung 1: Wir können zwar beliebige Punkte anfahren, allerdings nicht gradlinig. Wollen wir aber z.B. Buchstaben zeichnen, so ist gradliniges Fahren eine Notwendigkeit. In dieser und der folgenden Aufgabe wollen wir uns mit diesem Problem auseinandersetzen.

Um gradlinig fahren zu können, müssen wir zunächst in der Lage sein, eine Trajektorie zu generieren, die gradlinig im Toolspace verläuft. Da unser Toolspace zweidimensional ist, müssen wir also zunächst unsere Trajektoriengenerierung so anpassen, dass diese mit zweidimensionalen Start- und Endkoordinaten zurecht kommt. Implementiere dazu die Funktion

```
x, dx = generate_trajectory(start_pos, end_pos, v_max, a_max, dt)
```

welche für eine gegebene Startposition $\mathbf{x}_{\text{start}} \in \mathbb{R}^2$ und Zielposition $x_{\text{end}} \in \mathbb{R}^2$, sowie Geschwindigkeitslimit $\dot{x}_{\text{lim}} \in \mathbb{R}$, Beschleunigungslimit $\ddot{x}_{\text{lim}} \in \mathbb{R}$ und Zeitschritt $\delta t \in \mathbb{R}$, eine Trajektorie $(\mathbf{x}, \dot{\mathbf{x}})$ zurückgibt. Hierbei ist $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_N)$ so zu verstehen, dass $\mathbf{x}_i \in \mathbb{R}^2$ die (kartesische) Position und $\dot{\mathbf{x}}_i \in \mathbb{R}^2$ die (kartesische) Geschwindigkeit ist, die der Roboter zum Zeitpunkt $i\delta t$ nach Start der Trajektorie annehmen soll. Es gilt also grundsätzlich $\mathbf{x}_1 = \mathbf{x}_{\text{start}}$, $\mathbf{x}_N = \mathbf{x}_{\text{end}}$ und $\dot{\mathbf{x}}_1 = \dot{\mathbf{x}}_N = 0$. Beachte, dass alle $(\mathbf{x}_1, \ldots, \mathbf{x}_N)$ auf einer gradlinigen Strecke zwischen $\mathbf{x}_{\text{start}}$ und $\mathbf{x}_{\text{end}}$ liegen müssen.

Hinweis: Die in der vorherigen Aufgabe implementierte Funktion zur Trajektoriengenerierung darf von dieser Funktion gerne aufgerufen werden.

7. **Trajektorientransformation [2 Points]**

   Als letzten Schritt müssen wir nun noch die soeben generierte Trajektorie im kartesischen Raum $(\mathbf{x}, \dot{\mathbf{x}})$ in eine Trajektorie im Gelenkwinkelraum $(\mathbf{q}, \dot{\mathbf{q}})$ transformieren. D.h. für jede kartesische Endeffektorposition $\mathbf{x}_i$ und Endeffektorgeschwindigkeit $\dot{\mathbf{x}}_i$, wollen wir nun Gelenkpositionen $\mathbf{q}_i$ und Gelenkgeschwindigkeiten $\dot{\mathbf{q}}_i$ finden, die diese realisieren.

   Implementiere dazu die Funktion

   ```
   q, dq = transform_trajectory(x, dx, arm, q0)
   ```

   welche eine Endeffektor-Trajektorie im kartesischen Raum $\left(\mathbf{x} \in \mathbb{R}^{N\times 2}, \dot{\mathbf{x}} \in \mathbb{R}^{N\times 2}\right)$, sowie eine Instanz des zu nutzenden Roboterarms und eine initiale Winkelposition $q_0 \in \mathbb{R}^2$ erhält und die transformierte Trajektorie $\left(\mathbf{q} \in \mathbb{R}^{N\times 2}, \dot{\mathbf{q}} \in \mathbb{R}^{N\times 2}\right)$ zurückgibt.

   Den Zusammenhang zwischen Endeffektorpositionen bzw. -geschwindigkeiten und Gelenkpositionen bzw. -geschwindigkeiten haben wir in der ersten Programmierübung bereits behandelt. Sollte dir also unklar sein, was hier zu tun ist, lohnt es sich, erneut einen Blick in diese Übung zu werfen. Um euch etwas Mühe zu ersparen, stellen wir euch die zuvor implementierten Funktionen zur Berechnung der inversen Kinematik

   ```
   q = arm.inverse_kinematics(x, q0)
   ```

   und der Jakobimatrix

   ```
   jac = arm.compute_jacobian(q)
   ```

   zur Verfügung.

   Hinweis: Da es für unseren Roboter in fast allen Fällen mehrere Lösungen für die inverse Kinematik gibt, ist es wichtig die Startlösung q_0 vorsichtig zu wählen. Ansonsten kann es passieren, dass der Roboter in zwei aufeinander folgenden Schritten der Trajektorie sehr unterschiedliche Lösungen im Gelenkraum findet, auch wenn die Endeffektorpositionen sehr nah beieinander sind. Nutze also immer die Gelenkwinkel des vorherigen Schritts als Initialisierung für die inverse Kinematik des folgenden Schritts. Für den ersten Schritt nutze bitte die initiale Winkelposition q_0.

Hinweis zu wissenschaftlichem Arbeiten

Der Fachbereich Informatik misst der Einhaltung der Grundregeln der wissenschaftlichen Ethik großen Wert bei. Mit der Abgabe einer Lösung für eine schriftliche Aufgabe oder eine Programmieraufgabe bestätigen Sie, dass Sie/Ihre Gruppe die alleinigen Autoren des gesamten Materials sind. Falls die Verwendung von Fremdmaterial gestattet ist, so müssen Quellen korrekt zitiert werden.

Weiterführende Informationen finden Sie auf der Internetseite des Fachbereichs Informatik:

`https://www.informatik.tu-darmstadt.de/studium_fb20/im_studium/studienbuero/plagiarismus/index.de.jsp`

Es ist nicht gestattet, Lösungen anderer Personen als die der Gruppenmitglieder als Lösung der Aufgabe abzugeben. Des Weiteren müssen alle zur Lösungsfindung verwendeten, darüber hinausgehenden, relevanten Quellen explizit angegeben werden. Dem widersprechendes Handeln ist Plagiarismus und ist ein ernster Verstoß gegen die Grundlagen des wissenschaftlichen Arbeitens, das ernsthafte Konsequenzen bis hin zur Exmatrikulation haben kann.