Computational Engineering und Robotik

TECHNISCHE UNIVERSITÄT DARMSTADT

Sommersemester 2022, Homework 1

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

Total points: 15 + 2 bonus

Abgabefrist: 11:59, Montag, 23 Mai 2022

1.1 Schreibkurs für Roboter [15 Points + 2 Bonus]

In dieser Übung ist es unser Ziel, mit einem zweigelenkigen Roboterarm (siehe Abb. 2) Buchstaben auf eine Leinwand zu zeichnen (siehe Abb. 1). Der Roboter kann beide Gelenke frei bewegen und zeichnet mit einem Druckkopf an seinem Endeffektor, den er nach Bedarf an- und ausschalten kann. Bevor wir jedoch Buchstaben zeichnen können, müssen wir uns den Herausforderungen der Kinematik stellen.

Dazu werden wir zunächst ein kinematisches Modell des Roboters aufstellen. Dies ermöglicht uns zum einen, diesen in Simulation zu visualisieren. Zum anderen können wir dieses Modell nutzen, um die inverse Kinematik zu berechnen. Mittels der inversen Kinematik sind wir in der Lage, den Druckkopf des Roboters an Punkten unserer Wahl auf der Leinwand zu platzieren. Allerdings werden wir sehen, dass dies alleine noch nicht ausreichend ist, um Buchstaben zu zeichnen, da wir auch sicherstellen müssen, dass der Roboter gerade Linien zwischen diesen Punkten zeichnen kann. Dieses Problem werden wir dann mittels Zwischenzielen (oder engl. Via-Points) beheben. Für Interessierte gibt es am Ende noch eine Bonusaufgabe, bei der wir uns eine Alternative zu den Zwischenzielen ansehen werden.

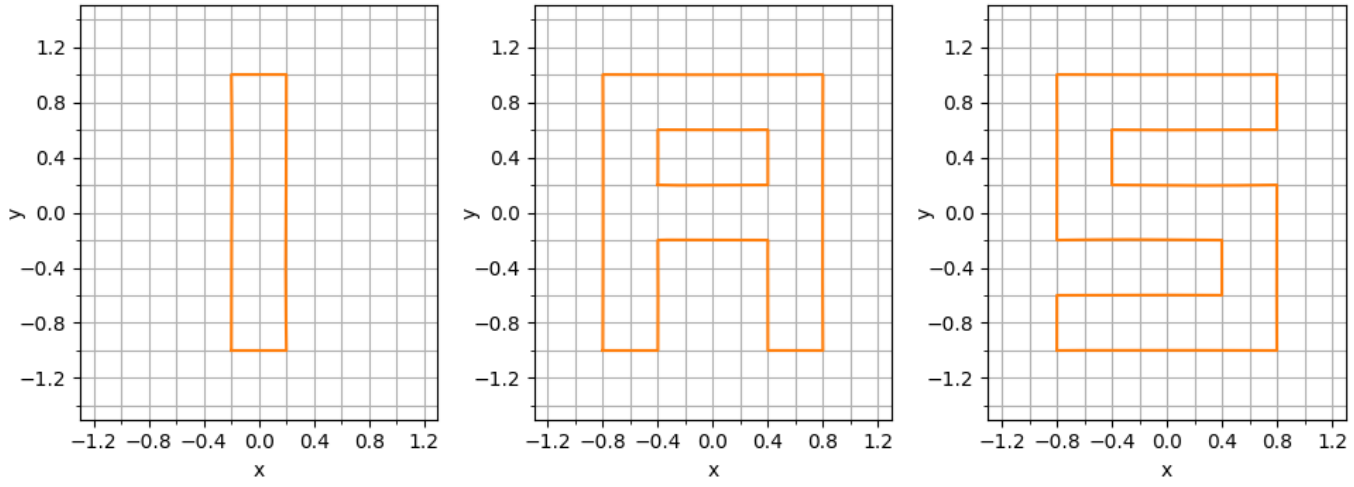

Figure 1: Ziel dieser Übung ist es diese Buchstaben auf die Leinwand zu malen.

Aufsetzen der Programmierumgebung

Setze dir zunächst eine *conda/virtualenv* Umgebung auf, in der *python 3.6* oder höher installiert ist. Wenn du nicht weißt, wie das geht, oder du noch nicht mit *Jupyter*-Notebooks gearbeitet hast, folge dem Tutorial unter

```
https://git.ias.informatik.tu-darmstadt.de/2021sscerpublic/pythontutorial
```

für eine kurze Einführung.

Falls du das nicht schon gemacht hast, aktiviere die *conda* Umgebung mittels

```
conda activate name_of_your_env
```

Lade dir nun das Archiv *cer_pex1.zip* aus Moodle herunter und entpacke es auf deinem PC. Navigiere anschließend in einem Terminal in das entpackte Verzeichnis mittels

```
cd /path/to/unpacked/dir
```

In dem Verzeichnis befindet sich ein *Jupyter*-Notebook *cer-pex1.ipynb* und das zu dieser Programmierübung gehörende *python* Paket *cer_pex1_lib*. Installiere dies mittels

```
pip install ./cer_pex1_lib
```

Bei der Installation dieses Pakets werden automatisch alle für diese Übungen benötigten Abhängigkeiten mitinstalliert.

Starte nun in dem Verzeichnis *Jupyter*

```
jupyter-notebook
```

und öffne das Notebook *cer-pex1.ipynb*.

Wie du siehst, steht in jeder Aufgabe an einer oder mehreren Stellen

```
# Implement this...
```

Im Verlauf dieser Übung wirst du diese Stellen mit Code füllen, um die erforderte Funktionalität umzusetzen. Bitte editiere hierfür nur Zellen, die mit dem obigen Kommentar markiert sind, da Änderungen an anderen Zellen bei der Korrektur nicht berücksichtigt werden. Es ist erlaubt neue Funktionen hinzuzufügen, solange dies ausschließlich in den zu bearbeitenden Zellen geschieht. Weiterhin dürfen die zu implementierenden Funktionen nicht umbenannt werden, da die Tests bei der Korrektur sonst scheitern.

Für fast jeder Funktion existiert eine Zelle mit öffentlichen Tests, die ihr nutzen könnt, um zu überprüfen, ob eure Implementierung korrekt funktioniert. Jedoch decken diese Tests immer nur einen kleinen Teil der möglichen Eingaben ab. Du bist also eingeladen, weitere Tests hinzuzufügen, um sicherzustellen, dass deine Implementierung auch in anderen Fällen funktioniert.

Abgabe und Bewertung

Lade zur Abgabe bitte lediglich das bearbeitete *Jupyter*-Notebook (*cer-pex1.ipynb*) in Moodle hoch. Es ist ausreichend, wenn **ein** Gruppenmitglied die Lösung einsendet, alle angegebenen Gruppenmitglieder einer Abgabe erhalten dann dieselbe Bewertung.

Die Bewertung setzt sich aus zwei Komponenten zusammen: öffentliche und geheime Tests. Für alle Teilaufgaben, für die öffentliche Tests existieren, erhältst du die Hälfte der Punkte, wenn deine Implementierung **alle** öffentlichen Tests besteht. Werden nicht alle öffentlichen Tests bestanden, so erhältst du für die entsprechende Teilaufgabe keine Punkte. Voraussetzung für die Bewertung einer Teilaufgabe ist jedoch, dass die Implementierung gewissenhaft und vollständig durchgeführt wurde. So gibt es z.B. keine Punkte, wenn man eine Lookup-Table baut, die gerade für die öffentlichen Tests die korrekten Ergebnisse zurückgibt, aber für keine anderen Eingaben funktioniert.

Die geheimen Tests sind – wie der Name vermuten lässt – geheim und werden von uns im Anschluss an die Abgabe zusätzlich zu den öffentlichen Tests durchgeführt. Anhand der geheimen Tests werden die verbleibenden Punkte vergeben.

1. **Analytische Lösung der Vorwärtskinematik [2 Points]**

   Da das von uns betrachtete System vergleichsweise übersichtlich ist, können wir die Vorwärtskinematik, also die Transformation von der Roboterbasis p_0 zum Druckkopf p_2, mit klassischen Methoden der Trigonometrie berechnen. Stelle dazu zunächst ein mathematisches Modell der Vorwärtskinematik in geschlossener Form auf

   $$\boldsymbol{p}^1 = f^{\boldsymbol{p}_1}(\boldsymbol{q}, \boldsymbol{l})$$
   $$\boldsymbol{p}_2 = f^{\boldsymbol{p}_2}(\boldsymbol{q}, \boldsymbol{l})$$

   wobei $\boldsymbol{q} \in \mathbb{R}^2$ die Gelenkwinkel angibt, $\boldsymbol{l} \in \mathbb{R}^2$ die Längen der Verbindungselemente, $\boldsymbol{p}^1 \in \mathbb{R}^2$ die Position des Ellenbogens und $\boldsymbol{p}_2 \in \mathbb{R}^2$ die Position des Druckkopfs (siehe Abb. 2 für eine Übersicht).

   Implementiere anschließend die Funktion

   ```
   p1, p2 = compute_forward_kinematics_analytical(q, l)
   ```

   welche die Gelenkwinkel `q` und Verbindungselementlängen `l` als Eingabe bekommt und einen Tupel aus Ellenbogenposition `p1` und Druckkopfposition `p2` zurückgibt. **Wichtig:** um die geheimen Tests zu bestehen, muss die Funktion mit verschiedenen Verbindungselementlängen $\boldsymbol{l}$ klar kommen und nicht nur mit der einen aus den öffentlichen Tests. Erstelle hierfür am besten einige zusätzliche Tests um dies sicherzustellen.

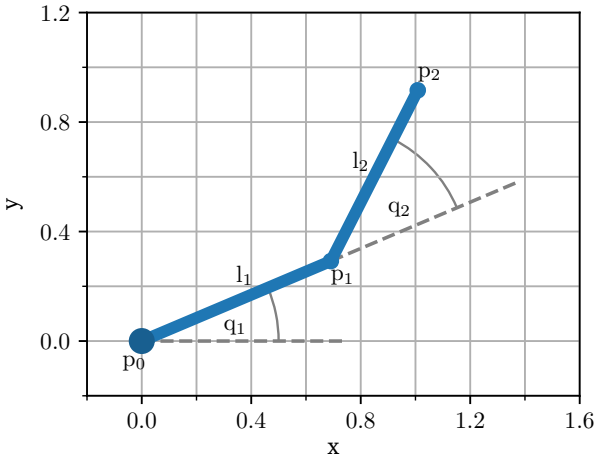

Figure 2: Visualisierung des Roboterarms. p_0 ist die Basis des Arms, p_1 das Ellenbogengelenk und p_2 der Druckkopf.

2. **Aufstellen einer Transformationsmatrix aus DH-Parametern [1 Points]**

   Der Nachteil der obigen Methode zur Bestimmung der Vorwärtskinematik ist, dass diese bei komplexeren Systemen oft nicht mehr praktikabel ist. Um unseren Code so erweiterbar wie möglich zu gestalten, wollen wir also zusätzlich eine Methode implementieren, welche die Vorwärtskinematik aus DH-Parametern berechnen kann. Der Vorteil dieser Methode ist, dass sie problemlos auf komplexere Systeme angewendet werden kann, da wir lediglich die DH-Parameter der einzelnen Glieder bestimmen müssen.

   Bevor wir die Vorwärtskinematik aus den DH-Parametern eines Systems berechnen können, müssen wir zunächst die Transformationsmatrix jedes einzelnen Glieds bestimmen können. Implementiere dazu die Funktion

   ```
   transformation_matrix = compute_transformation_dh(theta, d, a, alpha)
   ```

   welche als Eingabe die DH-Parameter eines i-ten Gliedes also $\boldsymbol{theta}_i$, $\boldsymbol{d}_i$, $\boldsymbol{a}_i$ und $\boldsymbol{alpha}_i$ bekommt und die entsprechende homogene Transformationsmatrix ${}^{i-1}\boldsymbol{T}_i$ berechnet. Hierbei ist `transformation_matrix` eine 4×4-Matrix, die die Transformation vom Koordinatensystem des $(i-1)$-Glieds zum Koordinatensystem des i-ten Glieds repräsentiert.

3. **Berechnung der Vorwärtskinematik mittels DH-Transformationen [2 Points]**

   Wir wollen nun die in der vorherigen Aufgabe implementierte Funktion einsetzen, um die Vorwärtskinematik zu berechnen. Bestimme dazu zunächst die einzelnen Glieder des Roboters und die zugehörigen DH-Parameter. Da unser Roboter nur zweidimensional ist, kannst du für die Bestimmung der DH-Parameter davon ausgehen, dass sich dieser ausschließlich in der XY-Ebene auf der Höhe Z=0 bewegt. Oder anders gesagt, wir erweitern jede Koordinate des Roboters um eine Z Komponente, die jedoch immer 0 ist. Bitte gib jedoch am Ende die Punkte wieder als 2D Vektoren aus X und Y zurück, wie in Aufgabe 1.

   Implementiere anschließend die Funktion

   ```
   p1, p2 = compute_forward_kinematics_dh(q, l)
   ```

   die `compute_transformation_dh` und die von dir bestimmten DH-Parameter nutzt um die Vorwärtskinematik zu berechnen. `q`, `l`, `p1` und `p2` sind definiert wie in Aufgabe 1.

   Verbindliche Anforderung: *Für die Berechnung der zwischen Transformationen muss die Funktion* `COMPUTE_TRANSFORMATION_DH` *verwendet werden.*

4. **Berechnung der inversen Kinematik mittels einfacher Fixpunktiteration [3 Points]**

   Für viele Anwendungen in der Robotik müssen wir in der Lage sein, mit dem Endeffektor (in unserem Fall der Druckkopf) bestimmte, vordefinierte Punkte im Arbeitsbereich anzufahren. Dafür ist es notwendig, gegeben einer gewünschten Position des Endeffektors $\boldsymbol{p}_2^*$, Gelenkwinkel $\boldsymbol{q}^*$ zu so zu bestimmen, dass der Endeffektors diese Position erreicht. Mittels der Vorwärtskinematik können wir bisher jedoch nur die andere Richtung berechnen, also gegeben der Gelenkwinkel die Position des Endeffektors bestimmen. Dem wollen wir jetzt Abhilfe verschaffen und mittels Fixpunktiteration die inverse Kinematik berechnen.

   Transformiere dazu zunächst das inverse Kinematikproblem

   $$\textit{Finde } \boldsymbol{q}^*\textit{, so dass } \boldsymbol{p}_2^* = f^{\boldsymbol{p}_2}(\boldsymbol{q}^*)$$

   in ein Nullstellenproblem

   $$\textit{Finde } \boldsymbol{q}^*\textit{, so dass } g(\boldsymbol{q}^*) = 0$$

   Implementiere anschließend die Funktion

   ```
   q_star = compute_inverse_kinematics_fpi(target_ee_pos, l, q0)
   ```

   die als Argumente eine Zielposition `target_ee_pos`, die Verbindungselementlängen `l` und eine Startlösung `q0` bekommt und die errechnete Optimallösung `q_star` zurückgibt. Implementiere die Fixpunktiteration so, dass sie terminiert wenn entweder 10,000 Iterationen durchgeführt wurden oder die euklidische Distanz des Endeffektors zur Zielposition kleiner ist als 10^{-5}. Nutze als Startlösung den übergebenen Wert `q0` und als Relaxationsmatrix $A \coloneqq 0.01I$.

5. **Berechnung der Jacobi-Matrix der Vorwärtskinematik [2 Points]**

   Wie wir an den Beispielen im *Jupyter* Notebook gesehen haben, konvergiert die Fixpunktiteration leider nicht überall. Um diesem Problem zu begegnen, wollen wir stattdessen das Newtonverfahren implementieren.

   Für das Newtonverfahren benötigen wir die Ableitung der Vorwärtskinematik nach den Gelenkwinkeln

$$J_{f^{p_2}}(\boldsymbol{q}) = \frac{\partial}{\partial \boldsymbol{q}} f^{p_2}(\boldsymbol{q}) = \begin{pmatrix} \frac{\partial}{\partial q_1} f_1^{p_2}(\boldsymbol{q}) & \frac{\partial}{\partial q_2} f_1^{\boldsymbol{p}_2}(\boldsymbol{q}) \\ \frac{\partial}{\partial q_1} f_2^{\boldsymbol{p}_2}(\boldsymbol{q}) & \frac{\partial}{\partial q_2} f_2^{\boldsymbol{p}_2}(\boldsymbol{q}) \end{pmatrix}$$

   welche man auch als *Jacobi-Matrix* des Roboters bezeichnet. **Wichtig**: es handelt sich hierbei noch nicht um die Jacobimatrix $J_g(\boldsymbol{q})$ von $g(\boldsymbol{q})$. Wir benötigen $J_{f^{p_2}}(\boldsymbol{q})$ jedoch, um $J_g(\boldsymbol{q})$ zu berechnen.

   Implementiere die Funktion

```
jac = compute_jacobian(q, l)
```

   die die 2×2 Jacobi-Matrix der Vorwärtskinematik `jac` für die Gelenkwinkel `q` zurückgibt.

6. **Berechnung inversen Kinematik mittels Newtonverfahren [3 Points]**

   Nun wollen wir die in der vorherigen Aufgabe berechnete Jacobi-Matrix nutzen, um das Newtonverfahren für die inverse Kinematik zu implementieren.

   Implementiere dazu die Funktion

```
q_star = compute_inverse_kinematics_newton(target_ee_pos, l, q0)
```

   die wie die Fixpunktiteration als Argumente eine Zielposition `target_ee_pos`, die Verbindungselementlängen `l` und eine Startlösung `q0` bekommt und die errechnete Optimallösung `q_star` zurückgibt. Implementiere das Newtonverfahren so, dass es terminiert wenn entweder 10,000 Iterationen durchgeführt wurden oder die euklidische Distanz des Endeffektors zur Zielposition kleiner ist als 10^{-5}. Nutze als Startlösung den übergebenen Wert `q0` und als Schrittweite $\alpha \coloneqq 0.01$.

   Hinweis: Die Verwendung von `np.linalg.inv` kann zu numerischer Instabilität führen (was das ist lernen wir in Vorlesung 10). Es bietet sich daher meistens an, stattdessen `np.linalg.solve` zu verwenden.

7. **Bestimmung der Eckkoordinaten der Buchstaben [1 Points]**

   Bevor wir mit dem Schreiben von Buchstaben beginnen können, müssen wir zunächst definieren wie diese auszusehen haben. Nimm dir dazu Abb. 1 als Referenz und implementiere die fehlenden Teile von

```
path = get_letter_print_path(letter)
```

   wobei `letter` entweder "I", "A" oder "S" ist und `path` eine Liste von Wegpunkten ist, die angibt wie der Buchstabe gezeichnet werden muss. Jeder Eintrag von `path` hat die Form $((x, y), e)$, wobei (x, y) die Koordinate ist zu der der Druckkopf als Nächstes (gradlinig) fahren soll und die boolesche Variable e angibt ob der Roboter auf dem Weg zu dieser Koordinate drucken soll oder nicht. Angefangen von der ersten Koordinate arbeitet der Roboter dann diese Liste Wegpunkt für Wegpunkt ab, bis er das Ende erreicht. Als Beispiel haben wir den Buchstaben "I" in der Vorlage bereits implementiert.

   Achte bei der Implementierung bitte darauf, dass nur Eckpunkte zurückgegeben werden, d.h. z.B. für das "I" besteht die Liste nur aus den Punkten $(-0.2, -1.0)$, $(-0.2, 1.0)$, $(0.2, 1.0)$ und $(0.2, -1.0)$. Stelle weiterhin sicher, dass die von dir zurückgegebenen Wegpunkte exakt mit den Buchstaben in Abb. 1 übereinstimmen.

8. **Begradigung der Teilstrecken durch Zwischenziele [1 Points]**

   Wie wir in der vorherigen Aufgabe gesehen haben, ist der naive Ansatz, einfach die Eckpunkte der Reihe nach anzufahren, nicht ausreichend um lesbare Buchstaben zu zeichnen. Das Problem ist hier, dass lineare Bewegung der Gelenke keine lineare Bewegung des Druckkopfs hervorruft und dieser stattdessen Bögen fährt. Ein Weg dieses Problem zu beheben, ist die Strecken in kleine Teilstrecken zu unterteilen und diese der Reihe nach abzufahren. Zwar wird der Roboter auf diesen Teilstrecken auch Bögen fahren, machen wir diese aber klein genug, so fällt dies nicht auf.

   Implementiere dazu die Funktion

   ```
   via_points = compute_via_points(start_pos, target_pos, max_step_size)
   ```

   die eine Startposition `start_pos`, eine Zielposition `target_pos`, sowie eine maximale Abschnittslänge `max_step_size` bekommt und ein $n \times 2$ Array aus Zwischenzielen zurückgibt.

   Hierbei müssen alle Zwischenziele auf grader Linie zwischen `start_pos` und `target_pos` liegen und zwei aufeinanderfolgende Zwischenziele dürfen nicht weiter als `max_step_size` auseinander liegen. Der erste Eintrag der zurückgegebenen Liste an Zwischenzielen muss `start_pos` sein und der letzte Eintrag `target_pos`. Weiterhin soll die Anzahl der Zwischenziele so klein wie möglich gewählt werden, ohne dass die obigen Bedingungen verletzt werden.

9. **Bonus: Eine elegantere Lösung [2 Bonus Points]**

   Wie dir wahrscheinlich aufgefallen ist, ist das Problem an der obigen Methode, dass der Druckkopf sich an unterschiedlichen Stellen unterschiedlich schnell bewegt. Stellen wir uns nun vor, dieser würde Tinte ausgeben, so würde sich die Linienstärke dadurch ständig ändern und wir würden kein ansehnliches Bild bekommen. Also wollen wir es in dieser Aufgabe mit einer etwas eleganteren Lösung versuchen.

   Die Idee die wir dabei verfolgen ist die folgende: An unterschiedlichen Stellen benötigen wir offensichtlich unterschiedliche Gelenkgeschwindigkeiten, um die selbe Druckkopf-Geschwindigkeit zu erzielen. Anstatt mit konstanter Gelenkgeschwindigkeit die Ziele anzufahren, könnten wir auch versuchen an jedem Punkt die Gelenkgeschwindigkeiten so einzustellen, dass wir exakt mit der richtigen Druckkopf-Geschwindigkeit fahren. Bisher haben wir jedoch noch keine Relation zwischen Gelenkgeschwindigkeiten $\dot{\boldsymbol{q}}(t)$ und Druckkopf-Geschwindigkeiten $\dot{\boldsymbol{p}}_2(t)$ hergestellt. Allerdings haben wir durch die Berechnung der Vorwärtskinematik bereits eine Relation zwischen den Integralen der beiden Terme hergestellt:

   $$\int \dot{\boldsymbol{p}}_2(t)\, dt = \boldsymbol{p}_2(t) = f^{\boldsymbol{p}_2}(\boldsymbol{q}) = f^{\boldsymbol{p}_2}\left(\int \dot{\boldsymbol{q}}(t)\right)$$

   Können wir diese Gleichung nutzen, um $\dot{\boldsymbol{q}}(t)$ in Abhängigkeit von $\dot{\boldsymbol{p}}_2(t)$ zu bestimmen?

   Implementiere die Funktion

   ```
   q_vel = compute_joint_vel(q, l, target_vel)
   ```

   die gegeben der Gelenkpositionen `q`, der Verbindungselementlängen `l` und einer Druckkopf-Zielgeschwindigkeit `target_vel`, die zugehörige Winkelgeschwindigkeit `q_vel` berechnet.

   Hinweis: Die Funktion `compute_jacobian(q, l)` aus Aufgabe 5 darf verwendet werden.

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.