Software-unterstützter Steuerungs- und Reglerentwurf eines instabilen mechanischen Starrkörpersystems

Carsten Knoll*, Kurt Reinschke

Institut für Regelungs- und Steuerungstheorie, TU Dresden

1. Oktober 2014

In diesem Dokument wird die im Paket `MSRM` („**M**odellbasierte **S**teuerung und **R**egelung eines **m**echanischen Systems") zusammengefasste Sammlung von Python-Routinen beschrieben. Diese Routinen ermöglichen das Aufstellen der Euler-Lagrange-Bewegungsgleichungen, ihre Linearisierung und Überführung in den Frequenzbereich (Polynom-Matrix-Beschreibung), die Festlegung geeigneter Basisgrößen, die Vorgabe von hinreichend glatten Wunschtrajektorien für die Basissignale, den Entwurf von stabilisierenden Reglern, die Simulation des geschlossenen Regelkreises auf Basis der nichtlinearen Bewegungsgleichungen, sowie eine geeignete Visualisierung (Diagramme und Animation). Die implementierte Funktionalität wird am Beispiel eines ebenen unteraktuierten mechanischen Systems diskutiert. Das methodische Vorgehen orientiert sich dabei am Buch [6].

1 Einführung

Der Zweck dieses Berichts ist es, die Anwendung der in [6] beschriebenen Vorgehensweisen zum Entwurf von Steuerungs- und Regelungs-Einrichtungen anhand eines Beispiels mittlerer Komplexität zu illustrieren. Informationen zur Inbetriebnahme der Software finden sich in der dem Quelltext beiliegenden `README.TXT`-Datei.

Anders als die meisten im Buch diskutierten Beispiele führen reale technische Problemstellungen oft auf mathematische Modelle, für deren Handhabung aufgrund des Rechenaufwandes eine Software-Unterstützung praktisch unverzichtbar ist. Sowohl in der Fallstudie [6, Kap. 10] als auch hier wird das Modell eines zweirädrigen Schienenfahrzeuges zum Transport von Schüttgut (kurz: „das Gefährt") betrachet, siehe Abb. 1.

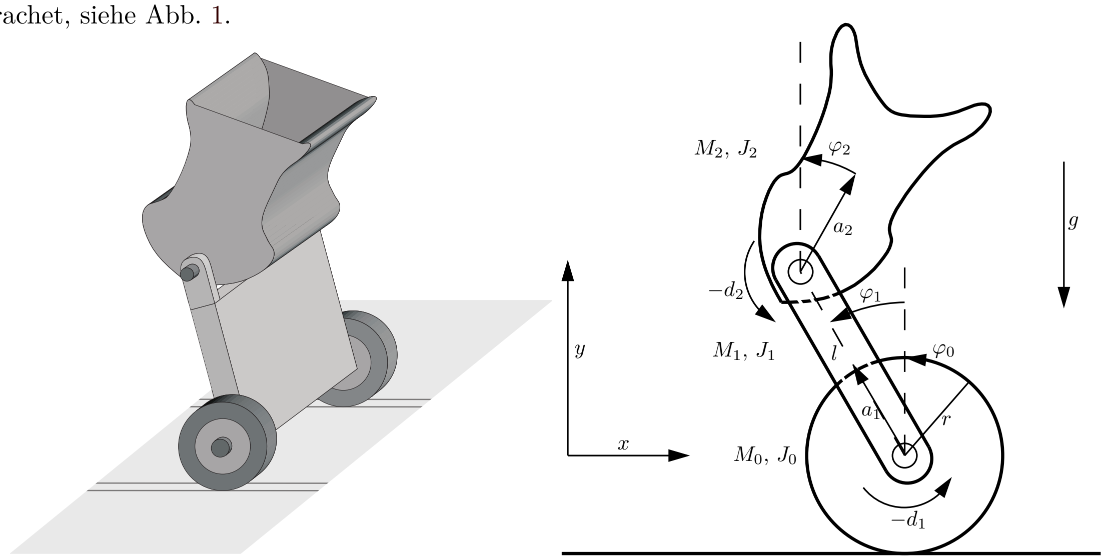

Abb. 1: Schematische Darstellung des Gefährts (3D- und Seitenansicht).

*Kontakt: Carsten.Knoll@tu-dresden.de

Die wesentliche Herausforderung dabei ist, dass dieses Gefährt, bzw. jede seiner physikalisch sinnvollen Ruhelagen, instabil ist. Ohne eine geeignete Vorgabe der Drehkräfte in den aktiven Gelenken würde das Gefährt umkippen. Die betrachtete Steuerungs- und Regelungsaufgabe ist die Überführung zwischen zwei Ruhelagen des einachsigen Fahrzeugs Mit anderen Worten: es ist eine seitliche Translation des Gefährts durchzuführen.

Aus der Vielzahl der verfügbaren Möglichkeiten zur Implementierung regelungstechnischer Algorithmen wurde die Programmiersprache Python in Verbindung den Bibliotheken Numpy, Scipy, Sympy und Matplotlib gewählt. Ausschlaggebend dafür waren die freie Verfügbarkeit und die sehr gute Integration von symbolischen und numerischen Berechnungen, ansprechender Visualisierung und klassischer Programmierung.

Die Dokumentation der Entwurfsschritte erfolgt anhand einer verbalen Beschreibung und der auszugsweisen Auflistung des damit zusammenhängenden Quelltextes.

Für ein vertieftes Verständnis der Implementierungsdetails (und damit der zugrunde liegenden Methodik) eignen sich besonders folgende kombinierbare Herangehensweisen:

(A): Änderungen an Parameterwerten oder an algorithmischen Code-Teilen und Interpretation der Veränderung der Ergebnisse

(B): Interaktive Inspektion relevanter Code-Abschnitte, vgl. Abschnitt 7.1.

Um dabei effektiv vorgehen zu können, ist die Kenntnis der Struktur des Programmpaketes und des Gesamtablaufs sehr hilfreich.

1.1 Struktur des Software-Pakets

Das vorgestellte Software-Paket `MSRM` besteht aus folgenden Modulen (Python-Dateien):

Modulname	Erläuterung
`model`	Modelle des Gefährts (nichtlineares und linearisiertes im Bildbereich)
`open_loop`	Steuerungsentwurf
`closed_loop`	Reglerentwurf
`simulation`	Simulation des geregelten nichtlinearen Modells
`visualization`	Visualisierung und Animation des Bewegungsablaufs

Tabelle 1: Struktur des Software-Pakets.

Des Weiteren ist das Funktionieren des Codes von einer Reihe zusätzlicher Pakete (Sammlung von Modulen) abhängig[1]:

Paketname	Erläuterung
`Sympy`	symbolisches Rechnen [7]
`Numpy`	grundlegende numerische Operationen, [5]
`Scipy`	komplexere numerische Operationen [5]
`Matplotlib`	Visualisierung [4]
`control_aux`	Hilfsfunktionen mit regelungstechnischem Bezug (eigene Entwicklung)

Tabelle 2: Externe Abhängigkeiten des Software-Pakets `MSRM`.

[1] Pakete der Python-Standardbibliothek sind nicht aufgeführt.

1.2 Gesamtablauf (Überblick)

Das Programmpaket umfasst die folgenden Aspekte zur Lösung der regelungstechnischen Entwurfsaufgabe:

- Festlegung der Werte der Modellparameter
- Herleitung eines mathematischen Modells für die betrachtete Regelstrecke (drei nichtlineare DGLn zweiter Ordnung)
- Linearisierung der Modellgleichungen um eine instabile Ruhelage und anschließende Laplace-Transformation ($\triangleq$ Entwurfsmodell)
- Festlegung geeigneter Basisgrößen
- Vorgabe der Wunschtrajektorien für die Basissignale
- Berechnung aller weiteren relevanten Verläufe (Winkel, Winkelgeschwindigkeiten, Stellsignale)
- Festlegung eines geeigneten (dynamischen) Reglers zur Stabilisierung der Wunschtrajektorien
- Erzeugung eines Simulationsmodells aus dem nichtlinearen Modell der Regelstrecke und dem Regelungsalgorithmus
- Optional: Exemplarische Berücksichtigung von Unbestimmtheiten durch kleine Abänderung der Modellparameterwerte
- Durchführung der Simulation mit vorgebbaren Anfangswerten
- Visualisierung der Simulationsergebnisse (Zeitverläufe, Animation)

Diese Schritte sind auf die in Tabelle 1 aufgelisteten Module verteilt. Zwischenergebnisse werden jeweils in separate Dateien geschrieben, wodurch unabhängig einzelne Teile des gesamten Prozesses durchlaufen werden können. In Abbildung 1.2 ist die Programmstruktur dargestellt.

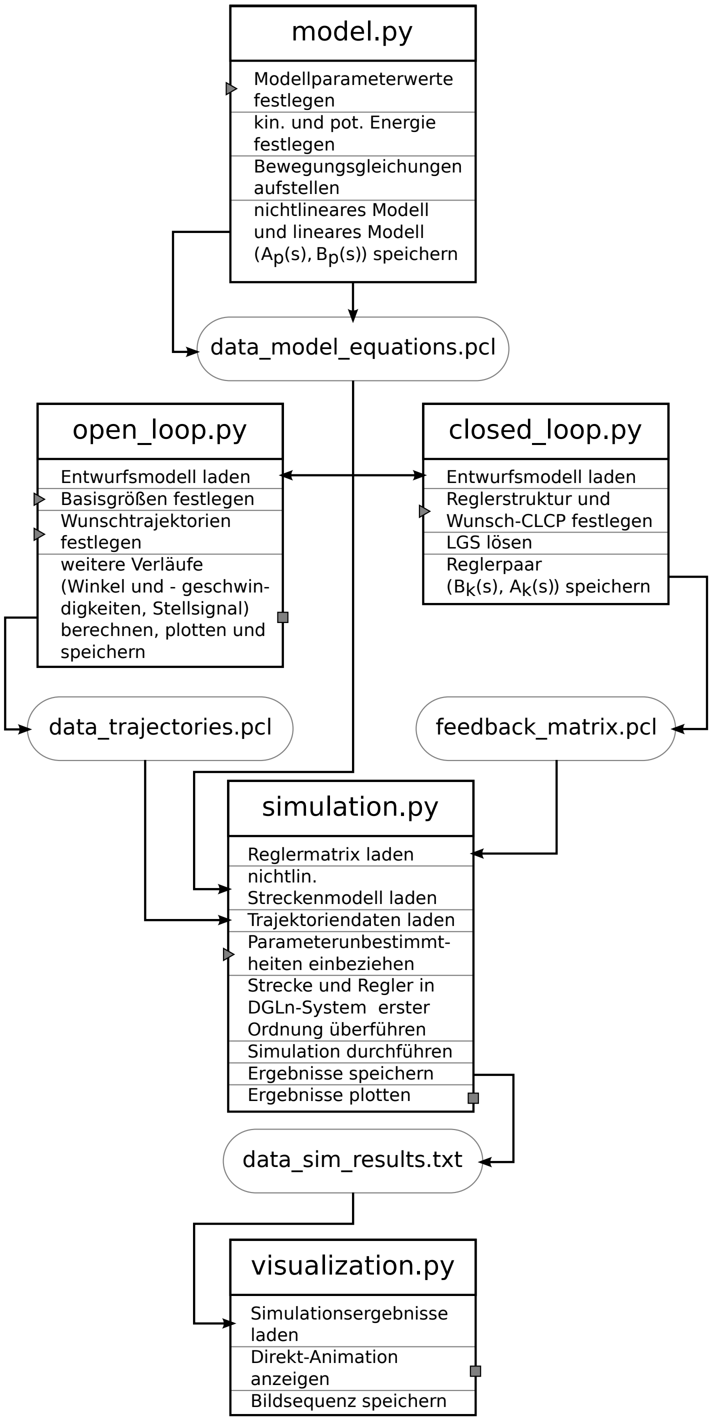

Abb. 2: Struktur des Programmpaketes.

Jedes Modul (Rechteck) lässt sich separat ausführen. Dabei werden die jeweils aufgelisteten Schritte abgearbeitet.
Die Ergebnisdaten der Ausführung eines Moduls werden jeweils in speziellen Dateien (abgerundete Kästen) gespeichert, und stehen den folgenden Modulen zur Verfügung. Für die numerischen Simulationsergebnisse wird das Text-Format verwendet. Alle anderen Daten-Dateien werden mittels des Serialisierungsmoduls der Python-Standardbibliothek (`pickle`) verarbeitet.
Durch das Symbol ▷ sind Stellen markiert, an denen sich eine Veränderung des Quellcodes zu Verständniszwecken besonders anbietet, beispielsweise zur Modifikation von Parameterwerten oder der alternativen Nutzung von Entwurfsfreiheitsgraden. In den Quelltext-Dateien sind diese Stellen mittels der Zeichenkette `##->` gekennzeichnet und andere vorbereitete Entwurfsvarianten sind dort per Fallunterscheidung einfach auswählbar.
Das Symbol □ kennzeichnet Abschnitte mit grafischer Ausgabe.

2 Modellbildung (Aufstellen der Bewegungsgleichungen)

Das in Abb. 1 dargestellte Gefährt wird als ebenes Starrkörpersystem mit drei mechanischen Freiheitsgraden und zwei Steuersignalen modelliert.

Auf Basis der symbolischen Ausdrücke für die kinetischen Energie T und die potentielle Energie V werden die Euler-Lagrange-Gleichungen, siehe z.B. [6, Abschnitt 2.5.3, Gl. 2.53] aufgestellt:

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial T}{\partial \dot{q}_\nu} - \frac{\partial T}{\partial q_\nu} = F_\nu - \frac{\partial V}{\partial q_\nu}, \quad \nu = 0, ..., 2. \tag{1}$$

Dabei bezeichnet F_ν die im jeweiligen Gelenk eingeprägten (Dreh-)Kräfte (Antriebskräfte und Reibungskräfte). Die Bildung der partiellen Ableitungen und das Aufstellen der Bewegungsgleichungen geschieht durch die Funktion `control_aux.model_tools.generate_model(...)`:

```
sys_model = generate_model(T, V, q, F)  # *+-
```

Quellcode-Auszug 1: model.py

Das Ergebnis sind drei verkoppelte Differentialgleichungen (DGLn) 2. Ordnung. Diese werden anschließend linearisiert und als Polynom-Matrix-Darstellung in der Datenstruktur abgespeichert:

```
116 # Gleichgewichtslage festlegen: (Winkel und Geschwindigkeiten = 0) #*+
117 x0 = zip0(qs) + zip0(qds)
118
119 sys_model.M0 = M.subs(x0)
120 sys_rest = sys_model.eq_list.subs(zip0(sys_model.qdds))
121 sys_model.K0 = sys_rest.jacobian(qs).subs(x0)
122
123
124 # Matrix der Dissipationsterme:
125 sys_model.D0 = sys_model.eq_list.jacobian(sys_model.qds).subs(x0)
126
127 # Eingangsmatrix (hier noch allgemein)
128 sys_model.B0 = sys_model.eq_list.jacobian(F_rel)
129
130 sys_model.params = params
131
132 # lineares Polynom-Matrix-Modell:
133 s = sp.Symbol('s')
134 sys_model.s = s
135 sys_model.Ap =\
136     (sys_model.M0 * s ** 2 + sys_model.D0 * s + sys_model.K0).subs(numparams)
137
138 # Unteraktuiertes System F0 = 0 -> nur die letzten 2 Spalten von B nutzen
139 sys_model.Bp = sys_model.B0[:, 1:]
140 #sp.pprint((sys_model.Ap.row_join(sys_model.Bp))) #*-
```

Quellcode-Auszug 2: model.py

Zur Überprüfung der Zwischenergebnisse kann die nach dem Einsetzen der numerischen Systemparameter resultierende Polynommatrix $(A_\mathrm{P}(s), B_\mathrm{P})$ ausgeben werden (Zeile 140):

```
|     2                2                      2                  |
|3.1*s  + 0.1*s     3*s  - 0.1*s              s            -1  0 |
|                                                                |
|    2                2                    2                     |
| 3*s  - 0.1*s    5.1*s  + 0.2*s - 30    2*s  - 0.1*s      1   -1|
|                                                                |
|      2              2                   2                       |
|      s           2*s  - 0.1*s      1.1*s  + 0.1*s - 10   0   1 |
```

Zu beachten ist dabei, dass die Drehkräfte d_1, d_2 hier entgegen der Koordinatenrichtung eingeführt wurden.

Im Programmablauf werden danach die *nichtlinearen* Modellgleichungen nach den Beschleunigungen aufgelöst und die rechten Seiten zum Zweck der späteren Simulation gespeichert. Abschließend werden alle relevanten Objekte mittels der Serialisierungsfunktion `pickle.dump(...)` in eine Datei gespeichert.

3 Steuerungsentwurf

Um das System, wie gewünscht steuern und regeln zu können, ist die Linksteilerfreiheit[2] des Matrizenpaars $(A_\mathrm{P}(s), B_\mathrm{P})$ vorauszusetzen. Diese ist leicht nachzuweisen, in dem man zwei belibige Minore 3. Ordnung

[2] Siehe dazu auch [6, Abschnitte 8.1, 8.2].

findet, die keine gemeinsamen Nullstellen haben.

```
# Linksteilerfreihheit überprüfen (Spaltennummerierung beginnt bei 0) #*+
S1 = set(st.roots(st.col_minor(ABp, 0,1,2))) # Nullstellen des OLCP
S2 = set(st.roots(st.col_minor(ABp, 2,3,4)))
assert len(S1.intersection(S2)) == 0 #*-
```

Quellcode-Auszug 3: open_loop.py

Bei der Umsetzung der in Abschnitt 1 formulierten Steuerungsaufgabe (Überführung zwisch zwei Ruhelagen) bestehen Freiheitsgrade im konkreten Bewegungsablauf. Diese werden hier exemplarisch durch drei *Varianten* abgebildet, die sich in unterschiedlicher Wahl der Basisgrößen niederschlagen.

Ausgangspunkt ist die polynomiale Systembeschreibung

$$\begin{pmatrix} A_p(s) & B_p \end{pmatrix} \begin{pmatrix} \mathbf{X}(\mathbf{s}) \\ \mathbf{U}(\mathbf{s}) \end{pmatrix} = \mathbf{0}, \tag{2}$$

siehe auch Gleichung [6, Gl. (10.4)]. Zur Bestimmung der Basisgrößen wird die $(n \times (n+m))$-Systemmatrix $(A_p(s), B_p(s))$ derart um m Zeilen ergänzt, dass die resultierende Matrix unimodular wird, das heißt, ihre Determinante soll unabhängig von s sein:

$$\det \begin{pmatrix} A_p(s) & B_p \\ Z & 0 \end{pmatrix} \stackrel{!}{=} 1. \tag{3}$$

Die Ergänzungszeilen definieren dabei die Basisgrößen:

$$\Xi(s) := \begin{pmatrix} Z & 0 \end{pmatrix} \begin{pmatrix} \mathbf{X}(\mathbf{s}) \\ \mathbf{U}(\mathbf{s}) \end{pmatrix}. \tag{4}$$

Für eine möglichst einfache Bewegungsplanung sind Basisgrößen erwünscht, die ausschließlich aus Linearkombinationen von $\mathbf{X}$ bestehen. Diese Forderung wird im Ansatz berücksichtigt, indem $Z \in \mathbb{R}^{m \times n}$ (Polynome 0. Ordnung) und für den hinteren Block die $\mathbb{R}^{m \times m}$-Nullmatrix gewählt wird. Im konkreten Fall gilt $n = 3$ (drei Winkel) und $m = 2$ (zwei Drehmomente). Der folgende Code-Auszug zeigt, wie freie Parameter in Z angesetzt und bestimmt werden.

```
##-> Festlegung der Basisgrößen: #*+
# Ergänzung der System-Matrix, sodass diese quadratisch und unimodular wird

k1, k2 = sp.symbols('k1, k2')

# Betrachtung von drei Varianten:
# Ansatz1: xi1 := k1*phi0 + k2*phi1,     xi2 := phi2
# Ansatz2: xi1 := k1*phi0 + k2*phi2,     xi2 := phi1
# Ansatz3: xi1 := -phi0,                 xi2 := k1*phi1 + k2*phi2

# Randbedingungen am Anfang
xa = Matrix( [0, 0, 0])
# RB Ende:
xb = Matrix( [-4, 0, 0])

variant = 1
if variant == 1:
    Z = Matrix([[k1, k2, 0, 0, 0], [0, 0, 1, 0 ,0]])
    # res = {k1: 0.0333333333333333, k2: 0.0474178403755869}
    T_end = 4.25
elif variant == 2:
    Z = Matrix([[k1, 0, k2, 0, 0], [0, 1, 0, 0 ,0]])
    # res = {k1: -0.100000000000000, k2: -0.0577464788732394}
    T_end = 9.5
elif variant == 3:
    Z = Matrix([[-1, 0, 0, 0, 0], [0, k1, k2, 0 ,0]])
```

```python
    # res = {k1: -0.100000000000000, k2: -0.0577464788732394}
    T_end = 17
    xb = Matrix( [2, 0, 0])
else:
    raise ValueError, "Unerwartete Variante"

M = st.row_stack(ABp, Z) # Hyper-Zeilen zusammenfügen #*-
```

Quellcode-Auszug 4: open_loop.py

Die so erhaltene unimodulare Matrix kann im Ring der Polynommatrizen invertiert werden

$$\begin{pmatrix} A_p(s) & B_p \\ Z & 0 \end{pmatrix}^{-1} =: U^R(s) =: \left(\begin{array}{c|c} U^R_{11}(s) & U^R_{12}(s) \\ \hline U^R_{21}(s) & U^R_{22}(s) \end{array}\right), \tag{5}$$

wobei die zweite Hyperspalte zur Berechnung der Systemgrößen $\mathbf{X}$ und der Eingänge $\mathbf{U}$ aus den Basisgrößen $\Xi(s)$ dient:

$$\mathbf{X}(\mathbf{s}) = \mathbf{U_{12}}(\mathbf{s})\boldsymbol{\Xi}(\mathbf{s}) \quad \text{und} \quad \mathbf{U}(\mathbf{s}) = \mathbf{U_{22}}(\mathbf{s})\boldsymbol{\Xi}(\mathbf{s}). \tag{6}$$

```python
U_R = M.adjugate()  # Hier inv == adjugate (weil det == 1) #*+
U_12R = U_R[:3, 3:]
U_22R = U_R[3:, 3:] #*-
```

Quellcode-Auszug 5: open_loop.py

Um die gewünschten Trajektorien $t \mapsto (\xi_1(t), \xi_2(t))$ festlegen zu können, werden zunächst die auf Ebene der Systemgrößen $\mathbf{x}$ angegebenen Randbedingungen in die entsprechenden Basissignale umgerechnet. Anschließend werden die Wunschtrajektorien für ξ_1 und ξ_2 als polynomiale Übergänge von den Anfangs auf die Endwerte mit Hilfe der Funktion `control_aux.symb_tools.trans_poly(...)` bestimmt.

```python
##-> Wunschtrajektorien im Zeitbereich festlegen #*+

Z1 = M[-2:, :3]

# Definition der Basisgrößen Xi aus Systemgrößen X:
# Xi := Z1 * X

xi_a = Z1 * xa
xi_b = Z1 * xb

# Übergangspolynome (Trajektorien der Basissignale (Zeitbereich))
t = sp.Symbol('t')
xi_polys = []
##-> Glattheitsanforderung (>=3) ist ein Entwurfsfreiheitsgrad
cn = 3 # Glattheitsforderung (legt Anzahl der Randbed. fest)

for i in range(2):

    # Randbedingungen:
    left = (0,xi_a[i,0]) + (0,)*cn
    right = (T_end,xi_b[i,0]) + (0,)*cn

    poly = st.trans_poly(t, cn, left, right) # Polynome bestimmen
    print "xi_{0}(t) = ".format(i), poly.evalf()

    # Stückweise definierte Funktion für konstante Teile am Anfang und Ende:
    pw = sp.Piecewise((left[1], t<left[0]), (poly, t<T_end), (right[1], True))
    xi_polys.append(pw)

# Liste in Matrix umwandeln
```

```
xi_traj = sp.Matrix(xi_polys) #*-
```

Quellcode-Auszug 6: open_loop.py

Nun werden die Matrizen $\mathbf{U}_{12}$ und $\mathbf{U}_{22}$ aus Gleichung (6) als Differentialoperatoren aufgefasst,

$$\mathbf{x}(t) = \mathbf{U}_{12}\left(\frac{\mathrm{d}}{\mathrm{d}t}\right) \circ \xi(t), \qquad \mathbf{u}(t) = \mathbf{U}_{22}\left(\frac{\mathrm{d}}{\mathrm{d}t}\right) \circ \xi(t), \tag{7}$$

mit denen sich die Zeitverläufe von $\mathbf{x}(t)$ und $\mathbf{u}(t)$ aus den Basissignalen bestimmen lassen. Programmtechnisch wird das mit Hilfe der Funktion `control_aux.symb_tool.do_laplace_deriv(...)` umgesetzt.

```
# Trajektorien der Winkel (Systemgrößen): #*+
# ("Gemischte Darstellung": Laplace-Bereich und Zeitbereich)
PHI = U_12R*xi_traj

# Laplace-Variable als Ableitungsoperator anwenden:
phi_traj = st.do_laplace_deriv(PHI, s, t)

#  Ausdrücke in ausführbare Funktion umwandeln
xi_func = st.expr_to_func(t, list(xi_traj), eltw_vectorize=True)
phi_func = st.expr_to_func(t, list(phi_traj), eltw_vectorize=True)# *-
```

Quellcode-Auszug 7: open_loop.py

Für die Überführungsvariante 1 sind die resultierenden Verläufe (Basissignale, Winkel, Eingänge) in Abb. 3 dargestellt. Durch die Vorgabe $\xi_2 \equiv \varphi_2$ (Variante 1), zusammen mit den hier gewählten Randbedingungen ergibt sich automatisch $\varphi_2(t) \equiv 0$. Der obere Körper bleibt folglich immer exakt aufrecht, d. h.$\varphi_2(t) \equiv 0$ (vgl. auch Abb. 4)

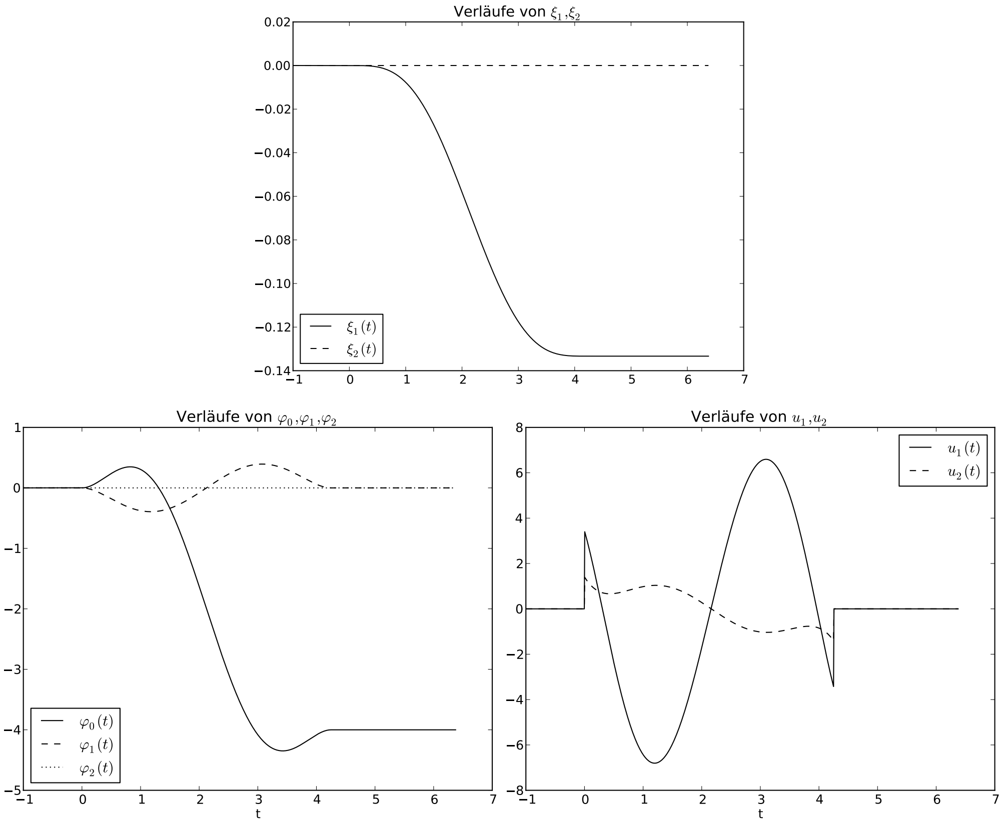

Abb. 3: Trajektorien für die Basissignale, die Winkelverläufe und die Eingänge.

Abschließend werden diese Verläufe für die spätere Verwendung im Regelkreis unter Nutzung des Serialisierungsmoduls `pickle` gespeichert.

4 Reglerentwurf

Wie in [6, Abschnitte 7.5 u. 10.4] beschrieben, wird der Reglerentwurf für das untersuchte mechanische System durchgeführt. Ausgangspunkt ist die Gleichung

$$\underbrace{\begin{pmatrix} A_\mathrm{P} & -B_\mathrm{P} \\ B_\mathrm{K} & A_\mathrm{K} \end{pmatrix}}_{M_\mathrm{CL}(s)} \begin{pmatrix} \mathbf{E} \\ \mathbf{U} \end{pmatrix} = \begin{pmatrix} A_\mathrm{P} & 0 \\ 0 & A_\mathrm{K} \end{pmatrix} \begin{pmatrix} \mathbf{R} \\ \mathbf{Z} \end{pmatrix} \tag{8}$$

welche den Zusammenhang zwischen Referenz-Größen, Regelabweichung und Eingangsgrößen beschreibt, vgl. [6, Abb. 7.12]. Zum Reglerentwurf wird die Matrix $(A_\mathrm{P}, -B_\mathrm{P})$ zunächst zeilenweise ergänzt, so dass die Linksteilerfreiheit erhalten bleibt. Die letzte Zeile enthält dann ausreichend freie Parameter, so dass die Determinante der Gesamtmatrix durch das Lösen eines linearen algebraischen Gleichungssystems einem gewünschten Polynom in s entspricht. Dieses Polynom ist dann das charakteristische Polynom des geschlossenen Regelkreises (CLCP) und der Regler wird als LAG[3]-Regler bezeichnet. Die Bestimmung der Reglermatrizen A_K und B_K ist im Modul `closed_loop.py` in der Funktion `calc_controller` implementiert. Im konkreten Fall hat die Matrix $(B_\mathrm{K}, A_\mathrm{K})$ zwei Zeilen. Als Entwurfsfreiheitsgrad sind verschiedene Varianten vorgesehen, die sich in den Zahlenwerten für vorgegebene Zeile und der Polynom-Struktur beider Zeilen (und damit in der dynamischen Ordnung des Reglers) unterscheiden. Der folgende Code-Auszug stellt eine dieser Varianten dar.

```
def calc_controller(Ap, Bp, desired_clcp, controller_variant = 1): #*+-

    ApBp = Ap.row_join(-Bp)#*+
    p, m = Bp.shape

    # -> Entwurfsfreiheitsgrad

    if controller_variant == 1:

        # erste Ergänzungszeile (unten):
        # konkrete Wahl der Zahlen ist ein Entwurfsfreiheitsgrad
        Z3Z4_1 = sp.Matrix([0, 10, -10, 1, 0]).T

        # Symbole für Ansatz:
        N_symbols = 2*p + 1
        p_symbols = sp.symbols( 'p1:%i' % (N_symbols + 1) )

        P0 = sp.Matrix(p_symbols[:p]).T # für 0. Ordnung
        P1 = sp.Matrix(p_symbols[p:2*p]).T # für 1. Ordnung

        # zweite Zeile von Z3
        Z3_2 = (P0+P1*s)

        # zweite Zeile (nach hinten) ergänzen
        Z3Z4_2 = Z3_2.row_join(sp.Matrix([0, p_symbols[-1]+s]).T)
        hc_flag = True  #*-
```

Quellcode-Auszug 8: closed_loop.py

Ist die von freien Parametern k_i abhängige Ergänzungszeile festgelegt, wird die Determinante gebildet und ein Koeffizientenvergleich mit dem gewünschten CLCP durchgeführt.

[3] LAG: lineares algebraisches Gleichungssystem

```python
    # Systemmatrix des geschlossenen Kreises (CLSM, Schritt 1): #*+
    CLSM1 = ApBp.col_join(Z3Z4_1)
    assert st.is_left_coprime(CLSM1)

    CLSM = CLSM1.col_join(Z3Z4_2)
    Z3Z4 = CLSM[p:, :] # = (Bk, Ak) (Zusatz-Zeilen)


    det = CLSM.berkowitz_det().expand()
    det = st.trunc_small_values(det)

    highest_coeff = st.poly_coeffs(det, s)[0]

    desired_clcp = sp.Poly(desired_clcp, s, domain = "EX")

    if hc_flag:
        # höchsten Koeffizienten angleichen (nicht immer notwendig)
        assert desired_clcp.is_monic
        desired_clcp *= highest_coeff

    # Determinante ist ein Polynom (s) mit den f-Param. in den Koeffizienten
    poly_det = sp.Poly(det, s, domain = "EX")
    deg = poly_det.degree()

    # Überprüfen, ob die Ordnung des Wunsch-CLCP mit der Ordnung
    # von CLSM.det() übereinstimmt
    assert desired_clcp.degree() == deg


    # Differenz soll identisch verschwinden
    # Koeff des Differenzpolynoms sollen all 0 sein
    diff_poly = st.trunc_small_values(poly_det - desired_clcp)


    # Koeff.Vergleich -> Gleichungssystem aufstellen
    # (durch den speziellen Ansatz linear in den Parametern)
    eqns = st.poly_coeffs(diff_poly, s)

    # Gleichungen (linke Seiten) nach 0 Auflösen
    sol = sp.solve(eqns, p_symbols)

    # sicherstellen, dass eine (eindeutige) Lösung gefunden wurde

    assert len(sol) == len(p_symbols)

    res = Z3Z4.subs(sol) # Endergebnis #*-
```

Quellcode-Auszug 9: closed_loop.py

Optional kann mittels der Funktion `verify_properness(...)` die Erfüllung der Properness-Bedingungen (vgl. [6, Abschnitt 7.5.2]) überprüft werden. Dabei ist zu beachten, dass man die Ableitungen $\varphi_0, \varphi_1, \varphi_2$ als physikalisch messbare Größen auffassen kann und damit auch Regler sinnvoll sind, welche die Properness-Bedingung verletzen (vgl. [6, Abschnitt 10.4]). Dies wird in Variante 4 der Reglerstruktur ausgenutzt.

Zusätzlich zur Struktur des Reglers, die mittels des Arguments `controller_variant` ausgewählt wird, kann selbstverständlich das CLCP vorgegeben werden. Dessen Ordnung muss mit jener von $\det M_{\mathrm{CL}}$ übereinstimmen. Die CLCP-Optionen im Code sind daher kompatibel zu den Strukturvarianten vorgegeben.

```python
clcp1b = roots_to_rpoly_expr(s, -1, -1.5+.5j, -2+1j, -2.5+2j)
```

```
clcp2a = roots_to_rpoly_expr(s, -.25, -2, -3, -3.1, -3.2, -10.3, -10.4, -10.5)
clcp2b = roots_to_rpoly_expr(s, -5.0, -2+1j, -2+1j, -2+1j, -10)
clcp2c = roots_to_rpoly_expr(s, -.25, -2, -20, -21, -22, -23, -24, -25) #[RL07]

clcp3a = roots_to_rpoly_expr(s, -.25, -2, -3, -3.1, -3.2, -10.3, -10.4, -10.5)
clcp3b = roots_to_rpoly_expr(s, -1.+1j, -2, -3+2j, -8+5j, -10)

clcp4a = (s+3)**6
clcp4b = (s+1)*(s+2)*(s+3)*(s+4)*(s+5)*(s+6) #*-
```

Quellcode-Auszug 10: closed_loop.py

Beispielhaft sei das Ergebnis des Aufrufes:

angeben (vergleiche auch [6, S. 574], dort in anderer Sortierung):

```
|      0                    -10                    10              1  0|
|                                                                      |
| -8.9917s - 3.5838  -27.9920s - 90.5973  -9.7569s + 6.7896  0  1|
```

5 Simulation

Das Modul `simulation.py` stellt die notwendige Funktionalität zur Verfügung, um den geschlossenen Regelkreis zu simulieren: Die zugehörende Anfangswertaufgabe wird durch numerische Integration gelöst. Der (via `scipy.integrate.odeint`) verwendete Integrationsalgorithmus basiert dabei auf einer Zustandsdarstellung des zu lösenden Differentialgleichungssystems[4] in der Form

$$\dot{\boldsymbol{y}} = \boldsymbol{f}_{\mathrm{rhs}}(\boldsymbol{y}, t). \tag{9}$$

Der Index „rhs" steht dabei für „right hand side", einer in der Simulationstechnik üblichen Bezeichnung für die rechte Seite der zu integrierenden Differentialgleichung. Das zweite Argument erlaubt aus Sicht des Lösungsalgorithmus eine Zeitabhängigkeit der DGL, welche allerdings für die vorliegende Aufgabenstellung irrelevant ist.

Das Modul löst im wesentlichen zwei Aufgaben: einerseits werden die *nichtlinearen* Bewegungsgleichungen des Gefährts (DGLn zweiter Ordnung) und der als Polynommatrix vorliegende Regler in eine gemeinsame Zustandsdarstellung überführt. Andererseits werden die numerischen Werte der Modell-Parameter durch (konstante) Zufallsgrößen gestört um damit aus der Systemidentifikation resultierende Fehler exemplarisch abzubilden. Beide Aufgaben wurden objektorientiert gelöst, d.h., die Funktionalität ist in der Klasse `SimModel` gekapselt. Die Methode `create_simfunction` dient als „Fabrik-Funktion" zur Erstellung der `rhs`-Funktion auf Basis der als Argumente übergebenen Regler-Matrizen $A_{\mathrm{K}}, B_{\mathrm{K}}$ und der Instanzvariablen (Modellparameter):

```

    def create_simfunction(self, Ak, Bk, symb):#**+
        """
        Erzeugt rechte Seite des Zustands-Systems
        (für Strecke und Beobachter)
        """

        n = self.state_dim

        # F0 kann nicht beeinflusst werden. -> ersten Einrtrag ignorieren
        input_symbs = list(pdict_eqn['extforce_list'])[1:]

        args = list(pdict_eqn['qs'])+list(pdict_eqn['qds'])+ input_symbs

```

[4] Obwohl allgemeinere Lösungsalgorithmen, zum Beispiel für differentialalgebraische Systeme, existieren, sind nur die Zustandsraummethoden in den gängigen Simulations-Werkzeugen direkt verfügbar

```
        # F0 = 0 im Ausdruck setzen
        self.param_values.update({'F0': 0})

        qdd_expr = pdict_eqn['rhs'].subs(self.param_values)

        assert st.matrix_atoms(qdd_expr, sp.Symbol).issubset( set(args) )

        # Funktion, die die Beschleunigung in Abhängigkeit von q, qd, u
        # berechnet (nichtlineare Bewegungsgleichung)
        qdd_fnc = sp.lambdify(args, list(qdd_expr), modules="numpy")

        # Regler in Zustandsdarstellung bringen
        controller_rhs, orig_controller_output = \
                                    poly_matr_to_state_funcs(Ak, Bk, symb)
```

Quellcode-Auszug 11: simulation.py

Nach diesen vorbereitenden Festlegungen wird *innerhalb des Namensraums von* `create_simfunction` die Funktion `rhs` definiert. Dabei handelt es sich um ein aufrufbares Funktionsobjekt, welches Zugriff auf den umgebenden Namensraum, und damit insbesondere auf die Objekte `qdd_fnc`, `controller_rhs`, etc hat. Das `rhs`-Objekt wird schließlich am Ende der Methode zurückgegeben. Es stellt die Funktion $f_{\text{rhs}}(\boldsymbol{y}, t)$ aus (9) zur Verfügung, welche dem Integrationsalgorithmus übergeben wird.

```

        def rhs(state, time): #*+

            # Zustände der Strecke
            q = state[:n/2].T
            qd = state[n/2:n].T

            plant_state = np.concatenate([q, qd])

            # Differenz zum Sollzustand:

            # Soll-Zustand zur aktuellen Zeit:
            des_state = st.to_np(state_func(time)).squeeze()

            # Differenz (e = r - x)
            e01 = des_state - plant_state

            # Zustände des Reglers
            w = state[n:].T
            wd = controller_rhs(w, e01)

            # Ausgang des Reglers (Eingang der Strecke)
            v = orig_controller_output(w, e01)


            u = st.to_np(u_func(time)).squeeze() +   v

            args = np.concatenate([q, qd, u])
            qdd = qdd_fnc(*args.T)
```

Quellcode-Auszug 12: simulation.py

Zur Berechnung der tatsächlich wirksamen Stellgrößen ist eine weitere Funktion notwendig: `final_input_calculation(...)`. Diese wird ebenfalls innerhalb des Namensraums von `create_simfunction` definiert und schließlich dem `rhs`-Objekt als Attribut hinzugefügt. Dadurch wird erreicht, dass nur das `rhs`-Objekt zurückgegeben werden muss.

```

        def final_input_calculation(state, time):#*+
```

```
            """
            Funktion um nachträglich die wirksamen Stellgrößen zu berechnen
            (gleicher Code (Teilmenge) wie rhs, aber anderer Rückgabewert)
            """

            # Zustände der Strecke
            q = state[:n/2].T
            qd = state[n/2:n].T
            plant_state = np.concatenate([q, qd])

            # Soll-Zustand zur aktuellen Zeit:
            des_state = st.to_np(state_func(time)).squeeze()

            # Differenz (e = r - x)
            e01 = des_state - plant_state

            # Zustände des Reglers
            w = state[n:].T
            wd = controller_rhs(w, e01)

            # Ausgang des Reglers (Eingang der Strecke)
            v = orig_controller_output(w, e01)
            u = st.to_np(u_func(time)).squeeze() +   v
            return u

        # diese Funktion wird der rhs-Funktion als Attribut mitgegeben
        # => Fabrik-Funktion (create_simfunction) hat nur einen Rückgabewert:
        rhs.final_input_calculation = final_input_calculation
```

Quellcode-Auszug 13: simulation.py

Die Methode `apply_uncertainity` dient zur Modifikation der Modellparameter, und erlaubt damit die Auswirkungen eines Unterschieds zwischen den nominellen Parameterwerten des Entwurfsmodells und denen des Simulationsmodells zu untersuchen. Dazu wird jeder Parameterwert mit einem zufälligen relativen Fehler in einem vorgegebenen Intervall beaufschlagt.

```

        np.random.seed(seed) #*+
        Np = len(self.param_values)
        noise = np.random.rand(Np)*2-1 # zwischen -1 und 1
        rel_noise = 1+noise*bound # zwischen 1-bound und 1+bound

        keys, values = zip(*self.param_values.items())
        new_values = np.array(values)*rel_noise
```

Quellcode-Auszug 14: simulation.py

Im Hauptteil des Skripts dann die beschriebene Funktionalität genutzt um die Simulation wie gewünscht durchzuführen. Dabei besteht die Möglichkeit die Modellungenauigkeit und den Anfangsfehler explizit anzugeben.

```
sim_mod = SimModel()

##-> Unbestimmtheiten berücksichtigen
#sim_mod.apply_uncertainty(bound = 0.05)

# rhs-Objekt auf Basis der Reglermatrizen und der veränderten Modell-Parameter
rhs = sim_mod.create_simfunction(Ak, Bk, s)

# Anfangswerte (laden und Anpassung an Zustandsdarstellung):
xa = list(pdict_ol['xa']) + [0,0,0] + [0]*sim_mod.number_of_controller_states
```

```
xa = st.to_np(xa).squeeze() # -> numpy array

##-> Anfangsfehler der Simulation vorgeben
#xa[0]+=.5

# Durchführung der eigentlichen Simulation
print "\n", u"Simulation des geschlossenen Regelkreises", "\n"
res = odeint(rhs, xa, tt, rtol = tol, atol = tol)#*-
```

Quellcode-Auszug 15: simulation.py

Abschließend werden die Simulationsergebnisse grafisch dargestellt und zur weiteren Verarbeitung gespeichert. Weil die Simulationsdaten sich als zweidimensionles Array darstellen lassen, kann dafür[5] die Funktion `np.savetxt` benutzt werden.

6 Visualisierung

Für eine anschauliche Interpretation der Bewegung lädt das Modul `visualization.py` die Simulationsergebnisse und stellt das Gefährt in der zur jeweiligen Zeit aktuellen Konfiguration schematisch dar. Dabei stehen zwei Ausgabe-Optionen zur Verfügung: Voreingestellt ist die Anzeige des Gefährts auf dem Bildschirm, wobei das Bild zeitabhängig erneuert wird, sodass eine vom Python-Skript in Echtzeit erzeugte Animation resultiert. Die zweite Möglichkeit ist die Ausgabe der einzelnen Bilder (siehe Abb. 4) in Dateien, die dann zum Beispiel zu einem Video zusammengefasst werden können.

7 Nutzungs- und Installationshinweise

7.1 Nutzungshinweise

Erfahrungsgemäß ist das Nachvollziehen von fremdem Quelltext oft eine Hürde. Die Autoren sind jedoch der Meinung, dass das Verständnis der hier genutzten Methodik zum Entwurf der Steuerungs und Regelungsentwurf und ihre Anwendung auf eigene Probleme von einer „spielerischen“ Beschäftigung mit dem Quelltext profitieren.

Um diese Art der Auseinandersetzung zu erleichtern sind im Quelltext Stellen mit `##->` hervorgehoben, die sich für ein unmittelbares Abändern besonders eignen, zum Beispiel um eine andere Überführungsvariante auszuwählen. Selbstverständlich muss nach der Änderung das entsprechende Python-Skript erneut ausgeführt werden, sowie ggf. weitere Skripte, welche die Zwischenergebnisse weiterverarbeiten (siehe Abb. 1.2).

Für ein tieferes Verständnis der Algorithmen empfiehlt es sich, den Inhalt von (Hilfs-)Variablen zu betrachten. Am einfachsten geschieht dies durch das Einfügen von `print`-Anweisungen. Allerdings weist dieser Ansatz zwei Nachteile auf: Zum einen wird der Ausgaben-Bereich (z.B. die System-Shell) oft schon durch relativ wenige `print`-Anweisungen unübersichtlich. Zum anderen ergeben sich durch das Wissen um den Inhalt einer Variable oft Anschlussfragen, den Inhalt anderer Variable betreffend. Dafür wäre das Hinzufügen weiterer Ausgabe-Anweisungen und ein Neustart des Skripts notwendig.

Eine sehr hilfreiches Werkzeug für solche Situationen ist durch die „IPython Embedded Shell“ (`IPS`) gegeben. Durch einen Aufruf von

```
IPS()
```

an beliebiger Stelle im Code[6] wird im lokalen Kontext eine interaktive Python-Eingabeaufforderung gestartet. Darin sind alle bis dahin importierten und definierten Variablen, Funktionen und sonstige Objekte verfügbar. Außerdem erlaubt `IPython` eine automatische Vervollständigung der Attributnamen und direkten Zugriff auf die Dokumentation. Durch das interaktive Untersuchen der wichtigen Quelltext-Stellen kann mit vergleichsweise wenig Aufwand ein vertieftes Verständnis der dem Bericht zugrunde liegenden Software-Routinen und damit auch des theoretischen Hintergrunds erreicht werden.

[5] Das bisher verwendete Modul `pickle` ist hauptsächlich für heterogene Datenstrukturen geeignet

[6] Voraussetzung dafür ist die Import-Anweisung `from IPython import embed as IPS` welche in allen genannten Modulen bereits vorhanden ist.

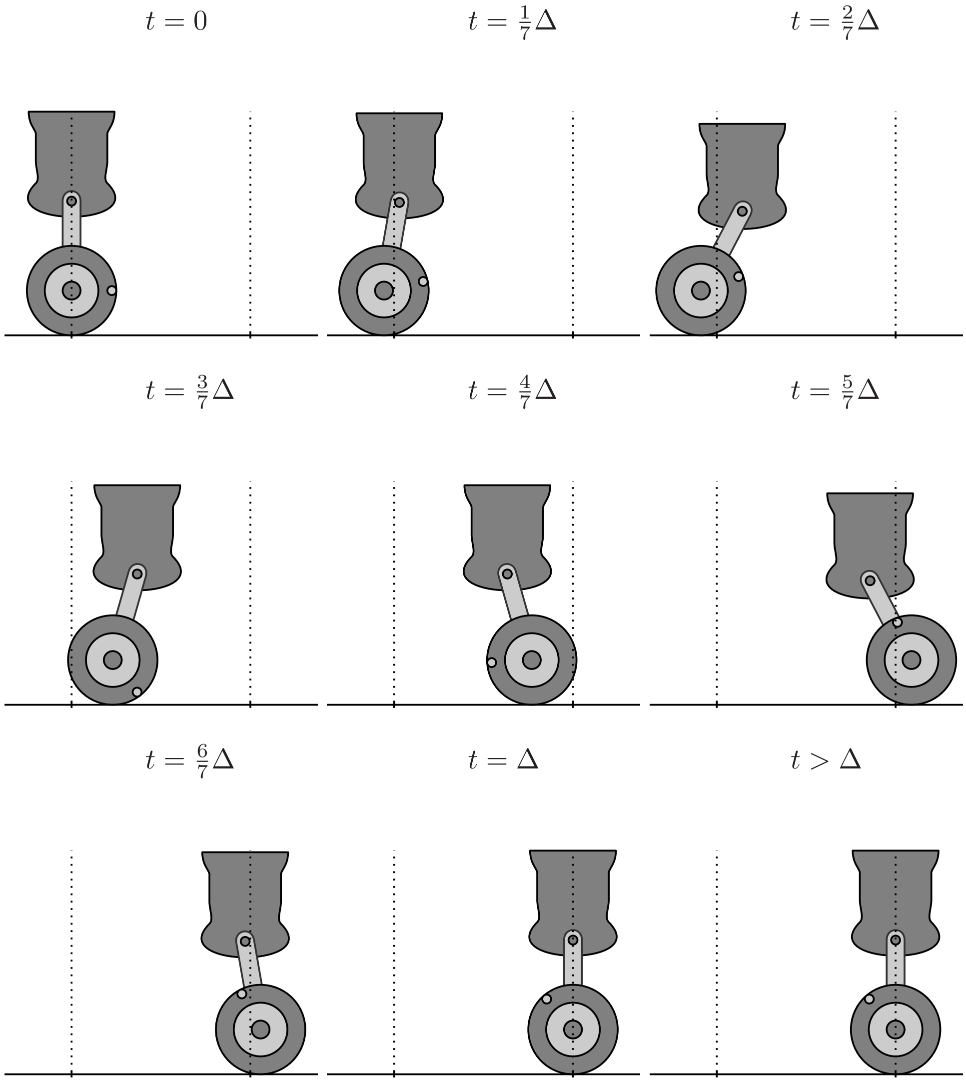

Abb. 4: Schematische Visualisierung des Gefährts während der Ruhelagenüberführung nach Variante 1.

7.2 Installationshinweise

Das im vorliegenden Bericht vorgestellte Softwarepaket `MSRM` wird unter [1] der Allgemeinheit zur freien Verfügung gestellt.

Alle zur Nutzung von `MSRM` notwendigen Software-Komponenten stehen unter einer freien Lizenz und sind im Internet in jeweils aktuellen Versionen verfügbar. Prinzipiell können der Python-Interpreter (inkl. der Python-Standardbibliothek), sowie alle Abhängigkeiten (`numpy, scipy, sympy, matplotlib`) separat installiert werden. Auf Unix-basierten Betriebsystemen mit einem Paket-Verwaltungssystem ist dieser Weg unkompliziert möglich.

Auf Windows-Plattformen empfiehlt sich die Nutzung einer entsprechenden Python-Distribution wie zum Beispiel „Python(x,y)“[2] oder „WinPython“ [3].

Weitere Informationen zur Inbetriebnahme der Software finden sich in der dem Quelltext beiliegenden `README.TXT`-Datei.

Literatur

[1] http://tu-dresden.de/rst/software; Zugriff 2014-10-01.

[2] http://code.google.com/p/pythonxy/; Zugriff 2014-08-14.

[3] http://code.google.com/p/winpython/; Zugriff 2014-08-14.

[4] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[5] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. http://www.scipy.org/; Zugriff 2014-08-14.

[6] K. Reinschke. *Lineare Regelungs- und Steuerungstheorie, 2. vollständig überarbeitete und erweiterte Auflage*. Springer, Heidelberg, 2014.

[7] SymPy Development Team. Sympy: Python library for symbolic mathematics, 2014. http://www.sympy.org/; Zugriff 2014-08-14.