Achse	a_{i-1}	α_{i-1}	d_i	θ_i	Art
1	0	0	0	$\alpha - pi/2$	Rotation
2	$l_1 = 0.16$ m	0	0	β	Translation and Rotation
3	$l_2 = 0.128$ m	0	0	0	Translation

Tabelle 1: DH-Parameter des Knickarmroboters

1 Weihnachtsprojekt: Simulation und Regelung eines Knickarmroboters

Aufgabe 1 - Bestimmung der Kinematik und DH-Parameter

Zur Bestimmung der Kinematik ist es notwendig herauszufinden, wie die einzelnen Gelenke des Roboters miteinander verbunden sind. Die Denavit-Hartenberg-Notation wird verwendet, um die 3D-Transformation zum nächsten Gelenk mithilfe von vier Parameter zu beschreiben. Im Fall des Knickarmroboters ist dies sehr einfach, da es sich lediglich um eine Kette von drei Gelenken handelt. Die Koordinatentransformation ist jeweils eine Rotation um die z-Achse mit θ_i und eine Translation in der xy-Ebene, um die Länge $a_i = l_i$. Wir können somit das erste Gelenk in den Ursprung legen und mithilfe von zwei Transformationen T_{12} und T_{23} alles beschreiben. Um mit einem nicht gedrehten Koordinatensystem anzufangen, benötigen wir zusätzlich T_{01}. Die DH-Parameter sind in Tabelle 1 aufgeführt.

Daraus resultieren die Transformationsmatrizen

$$T_{01}(\alpha) = \begin{pmatrix} \sin(\alpha) & \cos(\alpha) & 0 & 0 \\ -\cos(\alpha) & \sin(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$T_{12}(\beta, l_1) = \begin{pmatrix} \cos(\beta) & -\sin(\beta) & 0 & l_1 \\ \sin(\beta) & \cos(\beta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

und

$$T_{23}(l_2) = \begin{pmatrix} 0 & 0 & 0 & l_2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Die Gesamttransformation ergibt sich aus der Multiplikation der beiden Matrizen:

$$T_{03} = T_{01} \cdot T_{12} \cdot T_{23}$$

M. Denkinger, S. Eyes, J. Schnitzler
18. Januar 2024

Aufgabe 2 - Bestimmung der Bewegungsgleichung

Generalisierte Koordinaten

Die Wahl fällt zu $y(1) = \alpha$ und $y(2) = \beta$. Die generalisierten Koordinaten sind somit die Winkel der Gelenke 1 und 2.

Lagrang'sche Gleichung 2. Art

Die Bewegungsgleichung des Arms kann mit folgender Gleichung beschrieben werden:

$$M(y) \cdot \ddot{y} + D(y, \dot{y}) \cdot \dot{y} + g(y) = \tau_{Reib} + \tau_{Aktormoment} \tag{1}$$

Dabei berechnet sich die Massenmatrix M_i mit $M = \sum_i M_i$ von Arm i nach:

$$M_i(y) = \left[m_i J_{TiS}^T(y) \cdot J_{TiS}(y) + J_{RiS}^T(y) \cdot S_{0,iS}(y) \cdot I_{iS,iS} \cdot S_{0,is}^T(y) J_{RiS}(y) \right] \tag{2}$$

Dabei sind $J_{TiS}(y)$ und $J_{RiS}(y)$ die Jakobimatrizen für den Schwerpunkt für Körper i der Translation beziehungsweise Rotation und $I_{iS,iS}$ der Trägheitstensor von Körper i dargestellt in dessen Schwerpunkt.

Mithilfe der Transformationsmatrix $S_{0,iS}(y)$ kann der Trägheitstensor im Inertialsystem dargestellt werden:

$$I_{iS,0} = S_{0,iS}(y) \cdot I_{iS,iS} \cdot S_{0,is}^T(y) \tag{3}$$

Für $D(y, \dot{y})$ gilt:

$$D_{kj} = \sum_{i=1}^{n} h_{ijk}(y) \dot{y}_i \tag{4}$$

mit den Christoffel-Symbolen

$$h_{ijk} = \frac{1}{2} \left(\frac{\partial M_{kj}}{\partial y_i} + \frac{\partial M_{ki}}{\partial y_j} - \frac{\partial M_{ij}}{\partial y_k} \right) \tag{5}$$

Der Graviationsvektor $g(y)$ ist definiert als

$$g(y) = \left[\frac{\partial V}{\partial y_1}, \frac{\partial V}{\partial y_2} \right]^T \tag{6}$$

mit der potentiellen Energie V:

$$V(y) = \sum_i V_i = \sum_i m_i g \cdot p_{iS,0}(y) \tag{7}$$

nach 2.1.11 Lagrange'sche Gleichungen zweiter Art in [**?**]. Die Umsetzung der genannten Gleichungen wurde mit der Symbolic Math Toolbox in Matlab implementiert. Gleichung (1) liefert dann jeweils eine Differentialgeichung pro generalisierte Koordinate y_i.

M. Denkinger, S. Eyes, J. Schnitzler
18. Januar 2024

Reibungsterm

In Gleichung (1) geht das Reibmoment der Lagerung der Arme τ_{Reib} mit ein, welches aus der Überlagerung der viskosen mit der statischen Reibung modelliert wird:

```
%% Reibmoment:
Fs1 = 8.5e-04;                                          %statische Reibung in Nm
Fs2 = 3.2e-04;                                          %statische Reibung in Nm
Mreib_1 = 3.843e-06*y_punkt(1) + -1*sign(y_punkt(1))*Fs1;       %viskose + statische Reibung
Mreib_2 = 3.887e-06*y_punkt(2) + -1*sign(y_punkt(2))*Fs2;       %viskose + statische Reibung
```

Abbildung 1: Modellierung der Lagerreibung

Aufgabe 3 - Trajektorienplanung

Die vorliegende Trajektorienplanung beschäftigt sich mit der Definition und Berechnung von Trajektorien für einen Robotermanipulator. Das Ziel der Trajektorie wird dabei in globalen Koordinaten angegeben und ist durch den Vektor $y_{\text{end\_glob}}$ definiert. Dieser Vektor repräsentiert die gewünschte Endposition des Endeffektors des Roboters.

```
Ziel in globalen Koordinaten
y_end_glob = [4*sqrt(6)-4*sqrt(2)-10; ...
              -4*sqrt(6)-4*sqrt(2)-10*sqrt(3); 0]/125;
```

Um die Trajektorie zu planen, müssen zuerst die globalen Koordinaten in Gelenkwinkel umgerechnet werden. Aus der Kinematik ergibt sich folgende Funktion zum Bestimmen der Gelenkwinkel. Dabei ist $q(1)$ der Winkel α und $q(2)$ der Winkel β. Die Funktion `pos_endeff` gibt die Endposition in globalen Koordinaten abhängig von den Gelenkwinkeln α und β an. Weiterhin berechnet die Funktion `to_endeff` den Abstand zwischen der Endeffektorposition und der Startposition, abhängig von den Gelenkwinkeln. Sobald `to_endeff` Null ist, befindet sich der Endeffektor an der gewünschten Position. Die Funktion `q_end` löst dies mit den Anfangsbedingungen q_0 und gibt die Gelenkwinkel für die Endposition zurück.

```
pos_endeff = @(q) 4/25*[sin(q(1)) + 4/5*sin(q(1)+q(2)); ...
                        - cos(q(1)) - 4/5*cos(q(1)+q(2)); 0];
to_endeff = @(q) pos_endeff(q) - y_end_glob;
q_0 = [y_0(1); y_0(3)];
q_end = fsolve(to_endeff, q_0);
```

Mit den berechneten Winkeln kann die Trajektorienplanung beginnen. Wir haben uns für einen quintic spline entschieden, da dort sowohl Lage, Geschwindigkeit und Beschleunigung der Start- und Endlage vorgegeben werden können. Es müssen die Koeffizienten der Polynome 5.ten Grades berechnet werden.

```
a_poly = @(t)  a_coef(1) + a_coef(2)*(t-t_0) + a_coef(3)*(t-t_0).^2
             + a_coef(4)*(t-t_0).^3 + a_coef(5)*(t-t_0).^4 + a_coef(6)*(t-t_0).^5;

b_poly = @(t)  b_coef(1) + b_coef(2)*(t-t_0) + b_coef(3)*(t-t_0).^2
             + b_coef(4)*(t-t_0).^3 + b_coef(5)*(t-t_0).^4 + b_coef(6)*(t-t_0).^5;
```

Die Anfangs- und Endbedingungen werden in `a_init` und `b_init` definiert. Dabei sind die Lage, die Geschwindigkeit und die Beschleunigung für Anfangs- und Endlage beschrieben.

```
t_0 = tspan(1);
t_end = tspan(2);
a_init = [ q_0(1); y_0(2); 0;  q_end(1); 0; 0];
b_init = [ q_0(2); y_0(4); 0;  q_end(2); 0; 0];
```

Die A-Matrix wird durch das Ableiten des Polynoms und Einsetzen des Startzeitpunktes und des Endzeitpunktes für t gebildet.

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & (t_{\text{end}} - t_0) & (t_{\text{end}} - t_0)^2 & (t_{\text{end}} - t_0)^3 & (t_{\text{end}} - t_0)^4 & (t_{\text{end}} - t_0)^5 \\ 0 & 1 & 2(t_{\text{end}} - t_0) & 3(t_{\text{end}} - t_0)^2 & 4(t_{\text{end}} - t_0)^3 & 5(t_{\text{end}} - t_0)^4 \\ 0 & 0 & 2 & 6(t_{\text{end}} - t_0) & 12(t_{\text{end}} - t_0)^2 & 20(t_{\text{end}} - t_0)^3 \end{bmatrix}$$

Um die Koeffizienten zu bestimmen, lösen wir die Gleichung $A \cdot \text{coeff} = \text{init}$ und setzen die gefundenen Koeffizienten in die Gleichung des Polynoms ein.

```
a_poly(t) =  1.6982*t^3 - 2.5473*t^4 + 1.0189*t^5;
b_poly(t) =  -7.8540*t^3 + 11.7810*t^4 - 4.7124*t^5;
```

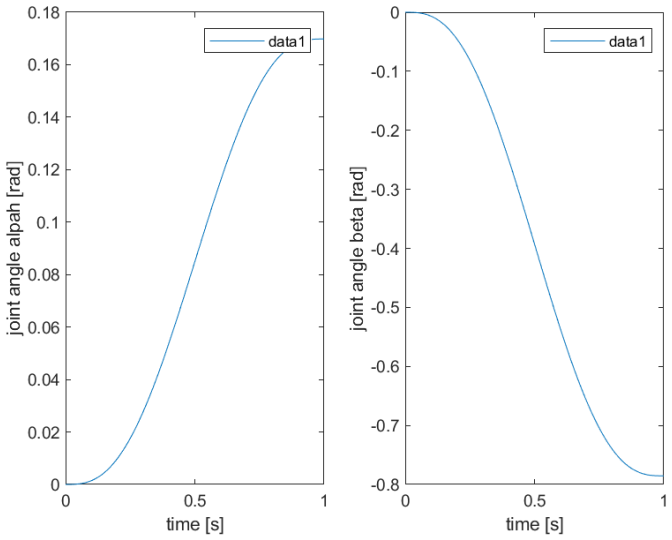

Abbildung 2: Trajektorie für α und β

Aufgabe 4 - Bewegungsgleichung in Matlab

Unsere Gruppe hat sich dafür entschieden die Bewegungsgleichung und ihre Lösung direkt in MatLab zu implementieren. Wie bereits in Aufgabenteil 2 erläutert wurde, haben wir die `Symbolic Toolbox von Matlab` benutzt, um die Bewegungsgleichung aufzustellen.

Unser Ziel ist es nun gewesen die aufgestellte Ordinary Differential Equations (ODE) mithilfe eines Solvers wie Euler-Vorwärts zu lösen. Dafür haben wir die Funktion `ode45` benutzt, welche die ODE numerisch löst. Die Funktion `ode45` benötigt als Eingabe die ODE, die Anfangsbedingungen und den Zeitbereich, in dem die ODE gelöst werden soll. Als Ausgabe liefert die Funktion die Lösung der ODE.

Wie vorgegeben haben wir als Optionen des Solvers `ode45` die `RelTol` auf 10^{-4} und `AbsTol` auf 10^{-7} gesetzt. Die `RelTol` gibt die relative Toleranz an, die die Lösung der ODE haben darf. Die `AbsTol` gibt die absolute Toleranz an, die die Lösung der ODE haben darf. Die Toleranzen sind wichtig, da die Lösung der ODE numerisch berechnet wird und somit nicht exakt ist. Die Toleranzen geben an, wie genau die Lösung der ODE sein muss. Außerdem haben wir eine maximale Schrittweite gesetzt mithilfe von `MaxStep` $= 3 \cdot 10^{-3}$. Das setzen dieser initiailen Werte geschieht in Listing 1.

```
% Syntax: y_0 = [alpha; alpha_dot; beta; beta_dot, err_alpha, err_beta]
y_0 = [pi/2; 0.5; -pi/5; -0.1; 0; 0];
tspan = [0, 1];
opts = odeset('RelTol', 1e-4, ...
      'AbsTol', 1e-7, ...
      'MaxStep', 3*1e3);

[t, y] = ode45(odefun, tspan, y0, opts);
```

Listing 1: Aufruf der Funktion `ode45`

Aufstellen der rechten Seite

Nun benötigen wir noch eine rechte Seite in der Form einer `odefun`, die abhängig ist von der Zeit und der Lösung der ODE. Die rechte Seite der ODE ist die Ableitung der Lösung der ODE. Haben wir diese können wir die ODE numerisch lösen und visualisieren.

Vorerst beschäftigen wir uns nicht mit der Implementierung des Reglers, welcher die Trajektorienplanung umsetzt, sondern schauen ganz grundlegend auf das Problem. Die Bewegungsgleichung erhalten wir als Symbolic Equation aus der von uns implementierten Funktion `bewegungsgl` und schreiben mithilfe des Befehls diese als Matlab-M-Funktion in den workspace. Aus Effizienzgründen machen wir das nur, wenn die Funktion noch nicht existiert oder wir explizit wollen, dass sie neu berechnet wird, siehe Listing: 2. Zuerst hatten wir es mit `subs` probiert, aber dies ist langsamer und weniger elegant.

```
%% Bewegungsgleichung
if ~exist('func_y_ddot.m', 'file') || berechne_doppelt
    symbolic_y_ddot = bewegungsgl();
    matlabFunction(symbolic_y_ddot, 'file', 'func_y_ddot.m');
end
```

Listing 2: Aufruf der Funktion `ode45`

Nun schreiben wir einen *Wrapper* für diese Funktion, da wir aus einer ODE 2. Ordnung eine ODE 1. Ordnung machen müssen. Dies geschieht in Listing 3.

```
function dy = assemble_odefun(t, y, func, reg)
% assuming y -> [alpha; alpha_dot; beta; beta_dot; err_alpha; err_beta]
    dy = zeros(6,1);
```

```
    u = reg.pid(t, y);

    % limit u values
    u_min = -1;
    u_max = 1;
    u = min(u_max, max(u_min, u));

    % Noise in friction can also be added to u
    noise = reg.noise_amp *2*(rand(2,1)-0.5);
    u = u + noise;

    % Bewegungsgleichung from symbolic toolbox
    y_ddot = func(y(1), y(2), y(3), y(4), u(1), u(2));

    dy(1) = y(2);
    dy(2) = double(y_ddot(1));
    dy(3) = y(4);
    dy(4) = double(y_ddot(2));
    % error from soll-value
    dy(5) = y(1) - reg.r_alpha(t);
    dy(6) = y(2) - reg.r_beta(t);
end
```

Listing 3: Definition der rechten Seite

Das wichtigste was hier passiert ist das die Ableitung des Winkels einfach die Winkelgeschwindigkeit ist, welches ein Eingabewert ist. Erst die rechte Seite der Winkelgeschwindigkeit ist dann die Bewegungsgleichung. Hier fügen wir unseren Regler hinzu und das Rauschen, welches wir in der Aufgabenstellung vorgegeben bekommen haben.

Aufgabe 5 - Vorsteuerung und PID-Regler

Um die Bewegungen des Roboters zur Endposition durchführen zu können, wurde eine Vorsteuerung und zwei PID-Regler in Matlab entworfen.

Die Vorsteuerung nutzt die in Aufgabe 3 vordefinierten Trajektorien $(a_{\text{poly}}, b_{\text{poly}})$, um die erwarteten Gelenkwinkel q(t) als Vorgabe für das System bereitzustellen. Durch die Vorsteuerung wird die geplante Trajektorie aktiv in das Regelungssystem integriert. Gleichzeitig kompensiert der PID-Regler den entstehenden Fehler zwischen der geplanten Trajektorie und der tatsächlichen Gelenkposition aufgrund von Störungen im Reibterm.

Die Störungen im Reibterm führen zu falschen Drehmomenten an den Gelenken, was wiederum zu unerwünschten Beschleunigungen führt. Die Vorsteuerung agiert, indem sie die erwarteten Gelenkwinkel liefert und somit die gewünschte Fahrtrichtung vorgibt. Der PID-Regler wird dann verwendet, um aktiv auf Abweichungen zwischen der geplanten Trajektorie und der realen Bewegung des Systems zu reagieren.

Der Regelkreis ist somit in der Lage, die Auswirkungen der Störungen im Reibterm zu minimieren, indem er das Regelungssystem in Echtzeit anpasst. Die Vorsteuerung spielt dabei eine Schlüsselrolle bei der Vorgabe der idealen Bewegung, während der PID-Regler auf tatsächliche Abweichungen reagiert und diese korrigiert. Dieses koordinierte Zusammenspiel ermöglicht eine präzise und robuste Regelung, auch in Anwesenheit von Störungen, und stellt sicher, dass die geplante Trajektorie so genau wie möglich verfolgt wird.

Der Regler ist sowohl für α als auch β implementiert. Die Eingangswerte umfassen die Lage, die Geschwindigkeit und den Regelabweichungsterm beider Gelenkwinkel. Reg.Kp ist der Proportional-

gewichtungsfaktor, der die Stärke des proportionalen Regelterms steuert. Er beeinflusst die Regelabweichung ($x(1) - \text{reg.r\_alpha}(t)$), also die Abweichung der aktuellen Position unter der Trajektorie. Reg.Ki ist der Integralgewichtungsfaktor, der die Stärke des integralen Regelterms steuert. Dabei repräsentiert $x(5)$ den kumulierten Fehler im Gelenkwinkel. Reg.Kd ist der Derivativgewichtungsfaktor, der die Stärke des derivativen Regelterms steuert. Dieser Term reagiert auf die Änderungsrate des Fehlers und glättet die Regelung.

```
assuming x -> [alpha; alpha_dot; beta; beta_dot; err_alpha; err_beta]
reg.pid = @(t, x) [-reg.Kp*(x(1)-reg.r_alpha(t))-reg.Ki*x(5)-reg.Kd*x(2);...
                   -reg.Kp*(x(3)- reg.r_beta(t))-reg.Ki*x(6)-reg.Kd*x(4)];
```

Der gesamte Ausdruck gewährleistet eine präzise und stabile Verfolgung der vorgegebenen Sollwerte $\text{reg.r\_alpha}(t)$ und $\text{reg.r\_beta}(t)$ durch die Gelenke. Dabei tragen die verschiedenen Terme des PID-Reglers dazu bei, die Regelungseigenschaften anzupassen und auf unterschiedliche Aspekte des Regelproblems zu reagieren.

Für den PID-Regler werden folgende Werte verwendet:

```
reg.Kp = 150;
reg.Ki = 1;
reg.Kd = 10;
```

Aufgabe 6 - Störsignal

Zur Modellierung eines Störsignals wird eine zufällige Zahl im Bereich von -1 bis 1 generiert. Diese Zufallszahl wird anschließend mit dem Faktor $\text{reg.noise\_amp} = 5 \times 10^{-3}$ multipliziert. Dieser Vorgang wird für beide Reibungsterme unabhängig durchgeführt, da im realen Aufbau davon ausgegangen wird, dass sie nicht denselben Fehler aufweisen.

$$\text{noise} = \text{reg.noise\_amp} \times 2 \times (\text{rand}(2,1) - 0.5)$$

Dieser Ausdruck berechnet das Störsignal (noise) durch Multiplikation der zufälligen Zahlen im Bereich von -1 bis 1 mit dem angegebenen Faktor.

Das Störsignal wird nun auf die Stellgröße des Reglers u addiert, da es den gleichen Einfluss auf die Bewegungsgleichung wie der Reibterm Q hat.

$$M(Q + u - D \cdot \dot{y}_{\text{punkt}} - G)$$

Die Stellgröße u wird durch die Addition des Störsignals korrigiert:

$$u = u + \text{noise}$$

Der PID-Regler muss dieses Störsignal ausgleichen, da die tatsächlich gefahrene Strecke von der geplanten Trajektorie abweichen wird.

Aufgabe 7 - Resultat und Visualisierung

Animiert wird die Bewegung des Knickarmroboters mithilfe des `plot` Befehls. Hierfür werden die bereits in Aufgabe 1 erläuterten homogenen Transformationsmatrizen genutzt, um die generalisierten Koordinaten α und β auf kartesische Koordinaten zu übersetzen. Das erste Gelenk ist hierbei gelb, das zweite grün eingefärbt.

Die Zielposition ist mit einem orangenem Kreis markiert.

Ein einzelnes Frame ist in Abbildung 3 zu sehen. Erzeugt wird diese Animation mit dem in Listing 4 dargestellten Code. Welche `drawnow` ausnutzt und für jedes Frame eine Pause einlegt, um die adaptive Schrittweite des Runge-Kutta solvers auszugleichen. Die Funktion `plot_robot` erzeugt die eingefärbten Gelenke.

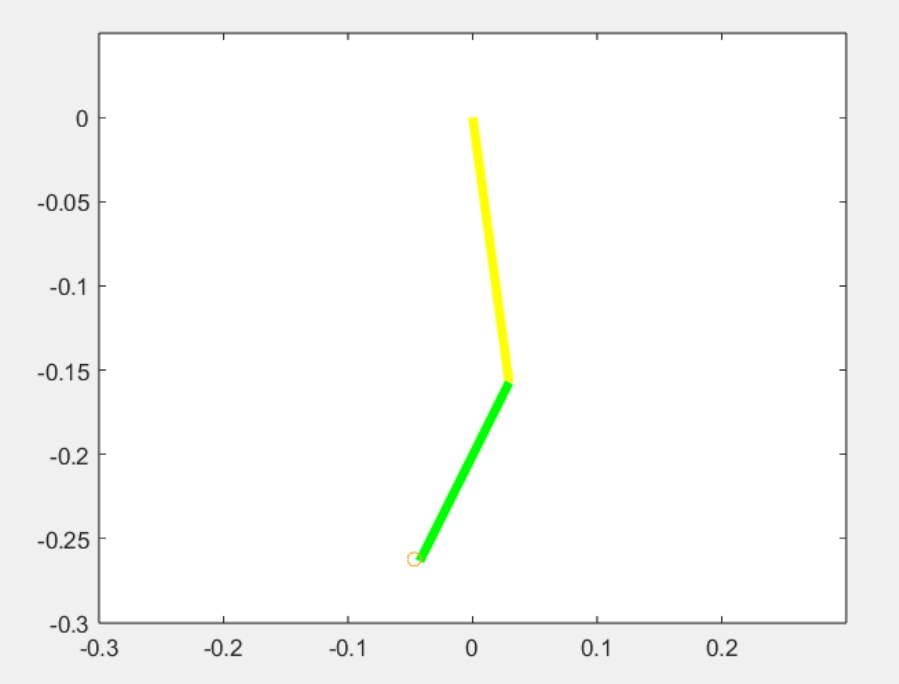

Abbildung 3: Visualisierung der Simulation

```
for frame=1:n_frame
    i = floor(frame/n_frame .* numel(t));
    G2 = T_02(alphas(i), betas(i))*orig;
    G3 = T_03(alphas(i), betas(i))*orig;
    plot_robot(G2, G3)
    hold on
    plot(y_end_glob(1), y_end_glob(2), "o")
    hold off
    drawnow;
    %wait such that the animation is as long as the simulated time
    pause(time(i));
end
```

Listing 4: Definition der rechten Seite