

Numerische Integration von Anfangswertproblemen, Teil 2 (Praxis)

Die Lösung von Aufgaben, die bereits im Teil 1 des Skripts mit Hilfe von MATLAB begonnen wurde, wird hier an typischen Beispielen erläutert. Weil dafür die im Teil 1 praktizierte Definition der Differenzialgleichungen als Inline-Functions nicht mehr praktikabel ist, soll zunächst eine Einführung in die Arbeit mit MATLAB-Functions gegeben werden.

Functions in MATLAB

MATLAB kennt zwei Arten von M-Files:

- ◆ **Script-Files** akzeptieren keine Eingabe-Argumente und liefern keine Ausgabe-Argumente ab (alle im Teil 1 behandelten Beispiele wurden mit Script-Files realisiert).
- ◆ **Function-Files** akzeptieren Eingabe-Argumente und können Ausgabewerte abliefern. Function-Files müssen mit dem Schlüsselwort **function** beginnen, Functions haben einen Namen. Functions können aus anderen M-files (Scripts oder Functions) aufgerufen werden, deshalb sollte der Name der Function mit dem Dateinamen (ohne die Extension .m) übereinstimmen.

In einem Function-File können mehrere Functions untergebracht sein, was gern ausgenutzt wird, um übersichtlich zu programmieren. Dann kann natürlich nur der Name einer Function mit dem Dateinamen übereinstimmen. Alle weiteren Besonderheiten werden nachfolgend an Beispielen erläutert.

Im MATLAB-Script **RKPendel.m** im Teil 1 wurde der Runge-Kutta-Algorithmus in einer Schleife folgendermaßen realisiert:

```
for i=1:n
    k1phi = phip (omega(i)) ;
    k1om = omegap (t(i),phi(i),omega(i),faktor) ;
    k2phi = phip (omega(i)+k1om*dt/2) ;
    k2om = omegap (t(i)+dt/2,phi(i)+k1phi*dt/2,omega(i)+k1om*dt/2,faktor) ;
    k3phi = phip (omega(i)+k2om*dt/2) ;
    k3om = omegap (t(i)+dt/2,phi(i)+k2phi*dt/2,omega(i)+k2om*dt/2,faktor) ;
    k4phi = phip (omega(i)+k3om*dt) ;
    k4om = omegap (t(i+1),phi(i)+k3phi*dt,omega(i)+k3om*dt,faktor) ;
    phi (i+1) = phi (i) + (k1phi + 2*k2phi + 2*k3phi + k4phi) * dt/6 ;
    omega(i+1) = omega(i) + (k1om + 2*k2om + 2*k3om + k4om) * dt/6 ;
end
```

Je viermal mussten die ersten Ableitungen aus unterschiedlichen Werten für die Zeit, den Winkel und die Winkelgeschwindigkeit berechnet werden. Dafür wurden jeweils die beiden Differenzialgleichungen, die die Bewegung des Stabpendels beschreiben

$$\dot{\varphi} = \omega \quad , \quad \dot{\omega} = -\frac{3g}{2l} \sin \varphi \quad ,$$

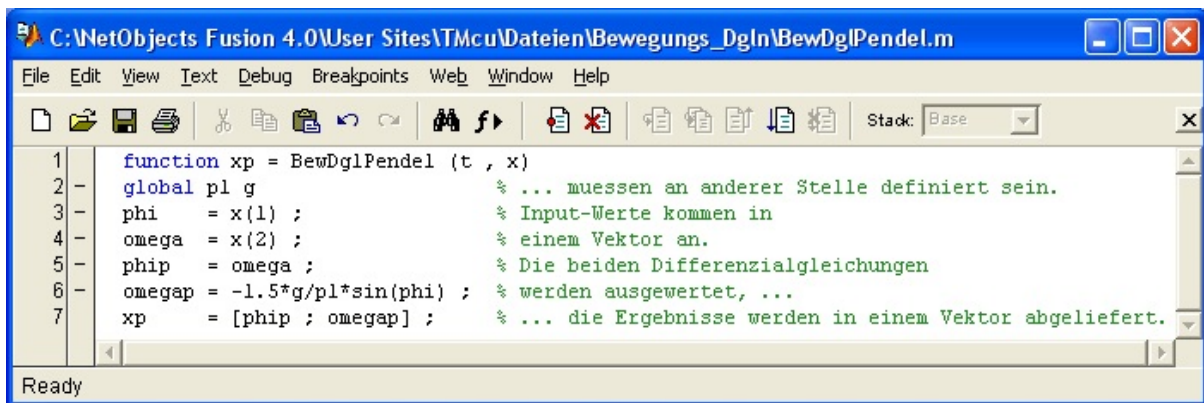
ausgewertet. Diese waren als Inline-Functions folgendermaßen definiert:

```
phip    = inline ('omega','omega') ;
omegap  = inline ('-faktor*sin(phi)','t','phi','omega','faktor') ;
```

Dies wird wie folgt geändert, wobei konsequent auf die Anpassung an die Strategie hingearbeitet wird, die für die Nutzung der in MATLAB für die numerische Integration von Anfangswertproblemen angebotenen Functions verwendet werden muss:

- ◆ Die Inline-Functions werden durch "normale" MATLAB-Functions ersetzt..
- ◆ Es werden keine Problemparameter an die Functions übergeben (wie der Parameter **faktor** in der Inline-Function zur Berechnung von **omegap**). Die Parameter werden trotzdem nur an einer Stelle definiert und als **global** deklariert, um sie mit den gleichen Werten in den Functions verfügbar zu haben.
- ◆ Es wird nur eine Function für alle Differenzialgleichungen definiert (und nicht wie oben für jede Differenzialgleichung eine). Diese liefert dann die Output-Werte (oben **phip** und **omegap**) in einem Vektor ab.
- ◆ Konsequenterweise werden deshalb auch die entsprechenden Inputwerte (im Beispiel oben **phi** und **omega**) in einem Vektor zusammengefasst.

Die MATLAB-Function, die die beiden oben gelisteten Inline-Functions ersetzt, sieht dann z. B. so aus:

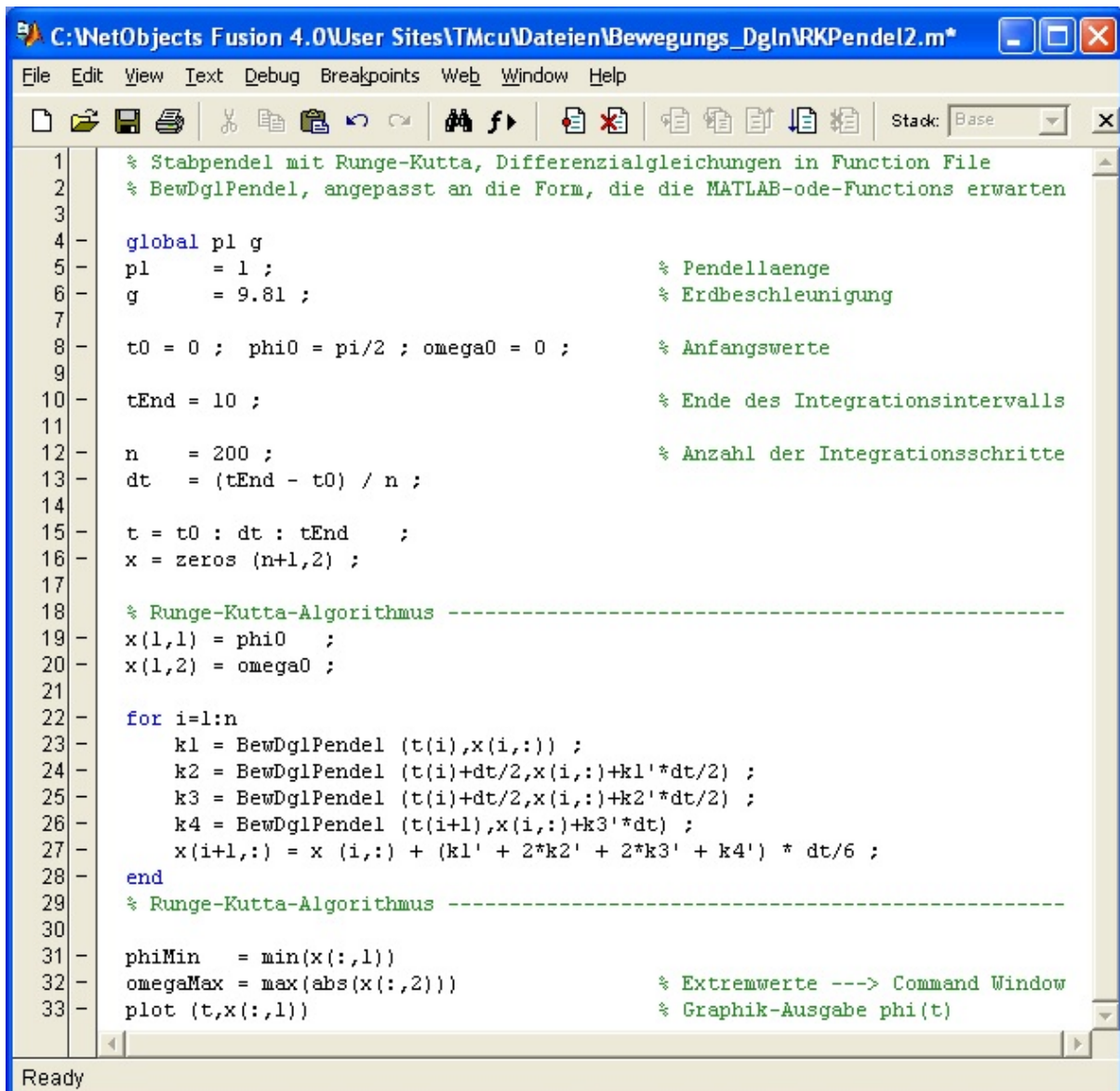


```
function xp = BewDglPendel (t , x)
2 - global pl g           % ... muessen an anderer Stelle definiert sein.
3 - phi    = x(1) ;      % Input-Werte kommen in
4 - omega  = x(2) ;      % einem Vektor an.
5 - phip   = omega ;     % Die beiden Differenzialgleichungen
6 - omegap = -1.5*g/pl*sin(phi) ; % werden ausgewertet, ...
7 - xp    = [phip ; omegap] ; % ... die Ergebnisse werden in einem Vektor abgeliefert.
```

Diese Function ließe sich auch kompakter schreiben, aber es wird vor allem im Hinblick auf kompliziertere Probleme dringend empfohlen, stets zunächst die Elemente des hier als **x** bezeichneten Inputvektors auf einfache "sprechende" Variablen zu übertragen und auch die Auswertung der Differenzialgleichungen zunächst einzeln vorzunehmen und dann die berechneten Werte zum Outputvektor zusammenzustellen.

Diese Function mit dem Namen **BewDglPendel** wird in einer Datei **BewDglPendel.m** abgespeichert. Dann kann sie aus einem MATLAB-Script so aufgerufen werden, wie es in dem auf der folgenden Seite zu sehenden Script **RKPendel2.m** zu sehen ist.

Der eigentliche Runge-Kutta-Algorithmus ist auf diesem Wege deutlich kompakter geworden, und er wird auch bei Aufgaben mit mehr als zwei Differenzialgleichungen exakt diese Form haben. Deshalb ist es naheliegend, auch diesen Algorithmus in eine Function zu verlagern. Diese müsste



```

C:\NetObjects Fusion 4.0\User Sites\TMcu\Dateien\Bewegungs_DglnVRKPendel2.m*
File Edit View Text Debug Breakpoints Web Window Help
Stack: Base
1 % Stabpendel mit Runge-Kutta, Differenzialgleichungen in Function File
2 % BewDglPendel, angepasst an die Form, die die MATLAB-ode-Functions erwarten
3
4 - global pl g
5 - pl = 1 ; % Pendellaenge
6 - g = 9.81 ; % Erdbeschleunigung
7
8 - t0 = 0 ; phi0 = pi/2 ; omega0 = 0 ; % Anfangswerte
9
10 - tEnd = 10 ; % Ende des Integrationsintervalls
11
12 - n = 200 ; % Anzahl der Integrationssschritte
13 - dt = (tEnd - t0) / n ;
14
15 - t = t0 : dt : tEnd ;
16 - x = zeros (n+1,2) ;
17
18 % Runge-Kutta-Algorithmus -----
19 - x(1,1) = phi0 ;
20 - x(1,2) = omega0 ;
21
22 - for i=1:n
23 - k1 = BewDglPendel (t(i),x(i,:)) ;
24 - k2 = BewDglPendel (t(i)+dt/2,x(i,:)+k1'*dt/2) ;
25 - k3 = BewDglPendel (t(i)+dt/2,x(i,:)+k2'*dt/2) ;
26 - k4 = BewDglPendel (t(i+1),x(i,:)+k3'*dt) ;
27 - x(i+1,:) = x (i,:) + (k1' + 2*k2' + 2*k3' + k4') * dt/6 ;
28 - end
29 % Runge-Kutta-Algorithmus -----
30
31 - phiMin = min(x(:,1))
32 - omegaMax = max(abs(x(:,2))) % Extremwerte ---> Command Window
33 - plot (t,x(:,1)) % Graphik-Ausgabe phi(t)
Ready

```

als Input die Information entgegennehmen, in welcher Function die Differenzialgleichungen definiert sind (hier: **BewDglPendel**) und benötigt außerdem die Anfangsbedingungen, das Zeitintervall, für das die Lösung erwartet wird, und die Anzahl der Abschnitte, in die dieses Intervall zu unterteilen ist.

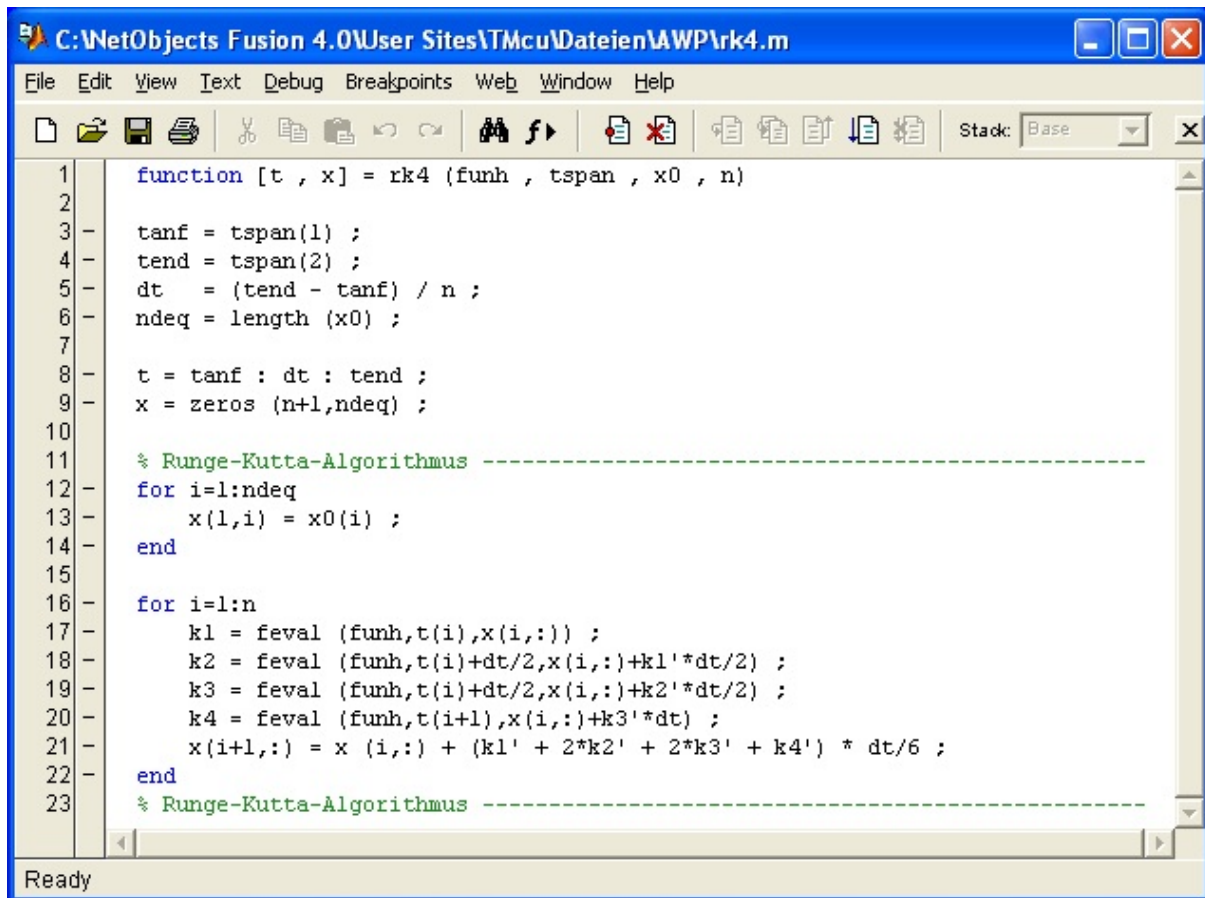
Der oben in den Zeilen 18 bis 29 zu sehende Runge-Kutta-Algorithmus soll einschließlich der vorbereitenden Zeilen 12 bis 17 durch folgenden Function-Aufruf ersetzt werden:

```

% Runge-Kutta-Algorithmus -----
[t , x] = rk4 ('BewDglPendel' , [t0 ; tEnd] , [phi0 ; omega0] , 200) ;
% Runge-Kutta-Algorithmus -----

```

Als Ergebnisse abgeliefert werden von dieser Function **rk4** ein Vektor **t**, der alle Zeitpunkte enthält, für die phi- und omega-Werte berechnet wurden (einschließlich des Anfangszeitpunktes 201 Werte) und in einer Matrix **x** in zwei Spalten die berechneten phi- bzw. omega-Werte. Die Function **rk4** in einer Datei **rk4.m** kann z. B. so aussehen:



```

C:\NetObjects Fusion 4.0\User Sites\TMcu\Dateien\AWP\rk4.m
File Edit View Text Debug Breakpoints Web Window Help
Stack: Base
1 function [t , x] = rk4 (funh , tspan , x0 , n)
2
3 - tanf = tspan(1) ;
4 - tend = tspan(2) ;
5 - dt = (tend - tanf) / n ;
6 - ndeq = length (x0) ;
7
8 - t = tanf : dt : tend ;
9 - x = zeros (n+1,ndeq) ;
10
11 % Runge-Kutta-Algorithmus -----
12 - for i=1:ndeq
13 -     x(1,i) = x0(i) ;
14 - end
15
16 - for i=1:n
17 -     k1 = feval (funh,t(i),x(i,:)) ;
18 -     k2 = feval (funh,t(i)+dt/2,x(i,:)+k1'*dt/2) ;
19 -     k3 = feval (funh,t(i)+dt/2,x(i,:)+k2'*dt/2) ;
20 -     k4 = feval (funh,t(i+1),x(i,:)+k3'*dt) ;
21 -     x(i+1,:) = x (i,:) + (k1' + 2*k2' + 2*k3' + k4') * dt/6 ;
22 - end
23 % Runge-Kutta-Algorithmus -----
Ready

```

Auf folgende Besonderheiten soll aufmerksam gemacht werden:

- ◆ Die Anzahl der Differentialgleichungen wird implizit aus der Länge des Vektors der Anfangsbedingungen $\mathbf{x0}$ entnommen (Zeile 6). Die Matrix, die die Ergebnisse aufnimmt, wird mit einer entsprechenden Anzahl von Spalten definiert, ihre erste Zeile übernimmt die Anfangsbedingungen.
- ◆ Der erste Input-Parameter ist der Name der Function, die die Differentialgleichungen repräsentiert und die in der Runge-Kutta-Schleife mehrmals zur Berechnung mit unterschiedlichen Parametern genutzt wird. Das wird mit der von MATLAB dafür vorgesehenen Function **feval** realisiert. Diese übernimmt als ersten Parameter den Funktionsnamen und anschließend genau die Parameter in der Reihenfolge, die die Funktion benötigt, deren Name ihr auf der ersten Position übergeben wird.

Um die Wirkung der Function **feval** zu illustrieren: Die beiden Zeilen

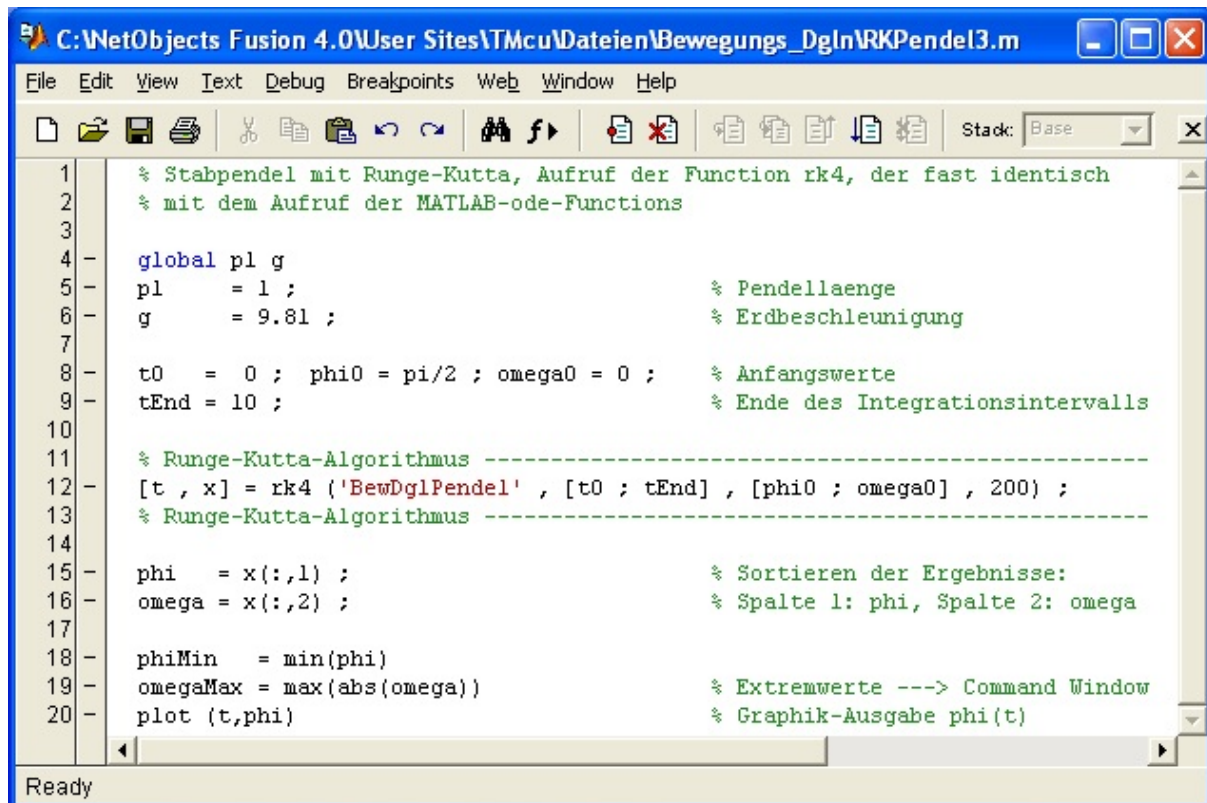
$$\mathbf{k1} = \mathbf{BewDglPendel} (\mathbf{t(i)} , \mathbf{x(i:)}) ;$$

(vgl. Script **RKPendel2** auf der vorigen Seite) und

$$\mathbf{k1} = \mathbf{feval} (\mathbf{funh} , \mathbf{t(i)} , \mathbf{x(i:)}) ;$$

sind äquivalent unter der Voraussetzung, dass **funh** im **feval**-Aufruf den String 'BewDglPendel' (oder ein entsprechendes "Function handle", dazu später) enthält.

Das Script **RKPendel3.m** arbeitet mit der Function **rk4** (www.DankertDankert.de/rk4.m) :



```

C:\NetObjects Fusion 4.0\User Sites\Tmcu\Dateien\Bewegungs_DglnVRKPendel3.m
File Edit View Text Debug Breakpoints Web Window Help
Stack: Base
1  % Stabpendel mit Runge-Kutta, Aufruf der Function rk4, der fast identisch
2  % mit dem Aufruf der MATLAB-ode-Functions
3
4  - global pl g
5  - pl      = 1 ;           % Pendellaenge
6  - g      = 9.81 ;       % Erdbeschleunigung
7
8  - t0     = 0 ; phi0 = pi/2 ; omega0 = 0 ;   % Anfangswerte
9  - tEnd   = 10 ;        % Ende des Integrationsintervalls
10
11  % Runge-Kutta-Algorithmus -----
12  - [t , x] = rk4 ('BewDglPendel' , [t0 ; tEnd] , [phi0 ; omega0] , 200) ;
13  % Runge-Kutta-Algorithmus -----
14
15  - phi    = x(:,1) ;     % Sortieren der Ergebnisse:
16  - omega  = x(:,2) ;     % Spalte 1: phi, Spalte 2: omega
17
18  - phiMin = min(phi)
19  - omegaMax = max(abs(omega)) % Extremwerte ---> Command Window
20  - plot (t,phi)         % Graphik-Ausgabe phi(t)
Ready

```

Man beachte, dass jetzt drei M-Files zusammenarbeiten: Aus **RKPendel3** wird die Function **rk4** aufgerufen, und über den Function-Aufruf (Zeile) 12 wird mitgeteilt, dass **rk4** die Function **BewDglPendel** (via **feval**) aufrufen soll. Der Benutzer von **rk4** muss sich jedoch auch bei der Behandlung anderer Probleme um diese Function nicht mehr kümmern.

Die MATLAB-ODE-Solver

Die gleiche Strategie, die oben mit der Function **rk4** verfolgt wurde, muss auch für die in MATLAB angebotenen "ODE-Solver" realisiert werden (ODE steht für "Ordinary Differential Equation"). Man kann z. B. in dem oben gelisteten Script die **rk4**-Aufruf-Zeile 12 einfach durch den Aufruf des MATLAB-Standard-Solvers **ode45** ersetzen:

```

% MATLAB-Solver ode45 -----
[t , x] = ode45 ('BewDglPendel' , [t0 ; tEnd] , [phi0 ; omega0]) ;
% MATLAB-Solver ode45 -----

```

Die Function **ode45** arbeitet mit einem noch besseren Runge-Kutta-Formelsatz, der für jeden Zeitschritt zwei Ergebnisse erzeugt: Eine Runge-Kutta-Approximation 4. Ordnung und eine Näherung 5. Ordnung werden miteinander verglichen, ihre Differenz wird als Maß für den lokalen Fehler benutzt, um damit die Schrittweite zu steuern.

Wegen dieser automatischen Schrittweitensteuerung entfällt der vierte Parameter (Anzahl der Zeitschritte), der für das Arbeiten mit der Function **rk4** erforderlich war: **ode45** legt selbst fest, in

wie viele Zeitschritte das Zeitintervall $t_0 \dots t_{End}$ unterteilt wird. Die Zeitschritte sind nicht mehr äquidistant, deshalb ist die Matrix der Ergebnisse x immer im Zusammenhang mit dem Ergebnisvektor t zu sehen. Die Länge des Ergebnisvektors kann gegebenenfalls mit der MATLAB-Funktion **length** erfragt werden.

Das so modifizierte Script **ODE45Pendel.m** liefert zunächst eher enttäuschende Ergebnisse: Man erkennt im nebenstehend zu sehenden Command Window, dass die beiden Extremwerte mit dem Runge-Kutta-Verfahren 4. Ordnung bei nur 200 Zeitschritten schon wesentlich besser genähert wurden.

Diese Erkenntnis wird sich als verallgemeinerungsfähig erweisen:

Die MATLAB-Standard-Einstellungen für die ode-Solver sind in der Regel nicht ausreichend. Man kommt trotz der automatischen Schrittweitensteuerung nicht umhin, Kontrollrechnungen mit unterschiedlichen Schrittweiten durchzuführen.

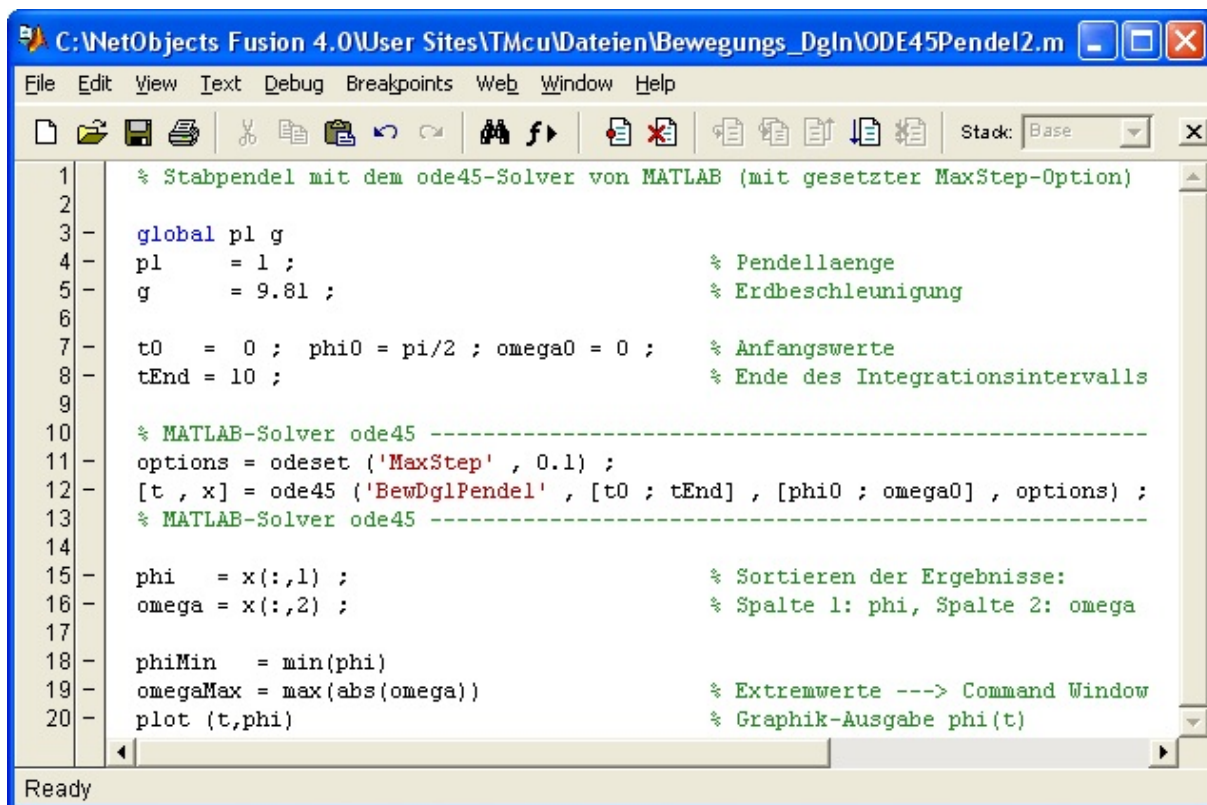


```

Command Window
File Edit View Web Window Help
>>
phiMin =
    -1.5690
omegaMax =
    5.4231
>> |
Ready

```

Für die MATLAB-ODE-Funktionen wird die Möglichkeit des Setzens geeigneter Optionen angeboten. Relativ einfach zu realisieren ist die "MaxStep"-Option (MATLAB-Help: "Positive scalar, an upper bound on the magnitude of the step size that the solvers uses. The default is one-tenth of the tspan interval."). Die Optionen werden mit der Funktion **odeset** gesetzt. Für das behandelte Problem wird das MATLAB-Script zu **ODE45Pendel2.m** so modifiziert:



```

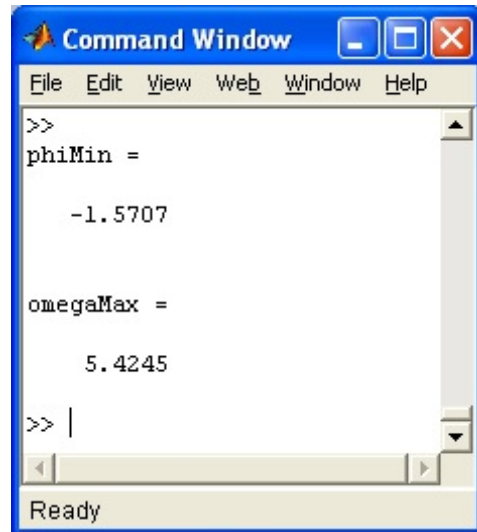
C:\NetObjects Fusion 4.0\User Sites\TMcu\Dateien\Bewegungs_Dgln\ODE45Pendel2.m
File Edit View Text Debug Breakpoints Web Window Help
Stack: Base
1 % Stabpendel mit dem ode45-Solver von MATLAB (mit gesetzter MaxStep-Option)
2
3 - global p1 g
4 - p1 = 1 ; % Pendellaenge
5 - g = 9.81 ; % Erdbeschleunigung
6
7 - t0 = 0 ; phi0 = pi/2 ; omega0 = 0 ; % Anfangswerte
8 - tEnd = 10 ; % Ende des Integrationsintervalls
9
10 % MATLAB-Solver ode45 -----
11 - options = odeset ('MaxStep' , 0.1) ;
12 - [t , x] = ode45 ('BewDglPendel' , [t0 ; tEnd] , [phi0 ; omega0] , options) ;
13 % MATLAB-Solver ode45 -----
14
15 - phi = x(:,1) ; % Sortieren der Ergebnisse:
16 - omega = x(:,2) ; % Spalte 1: phi, Spalte 2: omega
17
18 - phiMin = min(phi)
19 - omegaMax = max(abs(omega)) % Extremwerte ---> Command Window
20 - plot (t,phi) % Graphik-Ausgabe phi(t)
Ready

```

In der Scriptzeile 11 wird die "MaxStep"-Option auf den Wert 0,1 gesetzt, und die so definierte Struktur **options** wird der Funktion **ode45** als vierter Parameter angeboten. Dies hat zur Folge, dass sich das Ergebnis wesentlich verbessert (nebenstehendes "Command Window").

Generell kann folgende Empfehlung gegeben werden:

Man verwende immer die MaxStep-Option und führe mindestens zwei Berechnungen mit unterschiedlichen Werten aus (z. B. 0,1 und 0,01). Wenn die Ergebnisse ausreichend gut übereinstimmen, darf man davon ausgehen, dass die Rechnung "numerisch gesund" ist.



```

>>
phiMin =
    -1.5707

omegaMax =
    5.4245

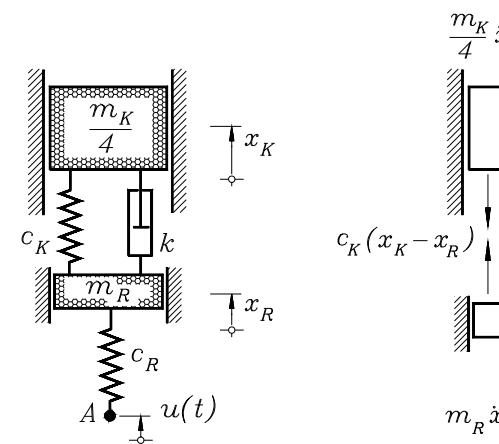
>> |
Ready
  
```

Systeme mit mehreren Freiheitsgraden, Kopplung in den Beschleunigungsgliedern

Die Bewegung von Systemen mit n Freiheitsgraden wird durch Differenzialgleichungssysteme mit n Differenzialgleichungen 2. Ordnung beschrieben. Diese können (wie im Teil 1 des Skripts für einen Freiheitsgrad beschrieben) durch jeweils eine zusätzliche Variable pro Freiheitsgrad (Geschwindigkeit oder Winkelgeschwindigkeit) in ein System von $2n$ Differenzialgleichungen 1. Ordnung überführt werden. Dann läuft die numerische Integration exakt so ab, wie sie in den vorangegangenen Abschnitten beschrieben wurde.

Beispiel:

Für die Analyse der Vertikalschwingungen eines Rades infolge der Bodenunebenheiten und der Übertragung der Schwingungen auf die Karosse dient das skizzierte Berechnungsmodell: Zwischen dem Rad (Masse m_R) und der Karosserie befinden sich eine Feder und ein Dämpfungsglied (Stoßdämpfer mit geschwindigkeitsproportionaler Dämpfung), auf denen näherungsweise ein Viertel der Karosseriemasse m_K lastet. Die Elastizität der Bereifung wird durch die Federzahl c_R erfasst. Dem Punkt A wird die Vertikalbewegung $u(t)$ aufgezwungen. Das System mit 2 Freiheitsgraden wird durch folgende Differenzialgleichungen beschrieben (vgl. Dankert/Dankert: Technische Mechanik, Seiten 691/692):



$$\begin{aligned} \frac{m_K}{4} \ddot{x}_K + c_K(x_K - x_R) + k(\dot{x}_K - \dot{x}_R) &= 0, \\ m_R \ddot{x}_R - c_K(x_K - x_R) - k(\dot{x}_K - \dot{x}_R) + c_R(x_R - u) &= 0. \end{aligned}$$

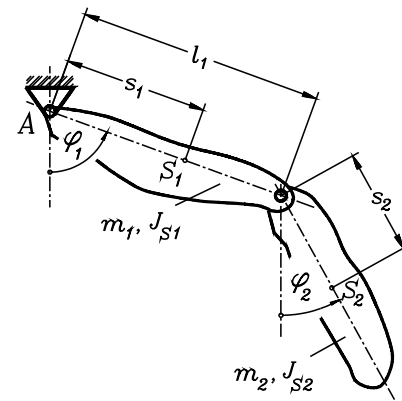
Als zusätzliche Variablen werden die Geschwindigkeiten v_R bzw. v_K eingeführt, und man erhält ein Differenzialgleichungssystem mit vier Differenzialgleichungen 1. Ordnung:

$$\begin{aligned}\dot{x}_K &= v_K & , \\ \dot{v}_K &= - \left[c_K (x_K - x_R) + k (\dot{x}_K - \dot{x}_R) \right] / (m_K/4) & , \\ \dot{x}_R &= v_R & , \\ \dot{v}_R &= \left[c_K (x_K - x_R) - c_R (x_R - u) + k (\dot{x}_K - \dot{x}_R) \right] / m_R & .\end{aligned}$$

Die Lösung dieses Systems mit MATLAB findet man unter www.DankertDankert.de als Ergänzung zur Seite 692.

Leider kann bei Systemen mit mehreren Freiheitsgraden nicht immer der Formelsatz für die Berechnung der ersten Ableitungen aller Variablen in dieser Form bereitgestellt werden (siehe folgendes Beispiel).

Beispiel: Ein Doppelpendel wird definiert durch die beiden Pendelmassen m_1 und m_2 , die auf die jeweiligen Schwerpunkte bezogenen Massenträgheitsmomente J_{S1} und J_{S2} , die Schwerpunktabstände von den Drehpunkten s_1 und s_2 und den Abstand l_1 der beiden Drehpunkte voneinander.



Der Weg zur Herleitung der nachfolgend angegebenen Bewegungs-Differenzialgleichungen für die freien Schwingungen dieses Systems mit zwei Freiheitsgraden wird in "Dankert/Dankert: Technische Mechanik" auf den Seiten 634/635 beschrieben:

$$\begin{aligned}\left[\left(\frac{s_1}{l_1} \right)^2 + \frac{J_{S1}}{m_1 l_1^2} + \frac{m_2}{m_1} \right] \ddot{\varphi}_1 + \left[\frac{m_2 s_2}{m_1 l_1} \cos(\varphi_1 - \varphi_2) \right] \ddot{\varphi}_2 \\ = - \frac{m_2 s_2}{m_1 l_1} \dot{\varphi}_2^2 \sin(\varphi_1 - \varphi_2) - \left(\frac{s_1}{l_1} + \frac{m_2}{m_1} \right) \frac{g}{l_1} \sin \varphi_1 \\ \left[\frac{m_2 s_2}{m_1 l_1} \cos(\varphi_1 - \varphi_2) \right] \ddot{\varphi}_1 + \left[\frac{m_2}{m_1} \left(\frac{s_2}{l_1} \right)^2 + \frac{J_{S2}}{m_1 l_1^2} \right] \ddot{\varphi}_2 \\ = \frac{m_2 s_2}{m_1 l_1} \dot{\varphi}_1^2 \sin(\varphi_1 - \varphi_2) - \frac{m_2 s_2}{m_1 l_1} \frac{g}{l_1} \sin \varphi_2\end{aligned}$$

Man erkennt die Besonderheit, die leider eher die Regel als die Ausnahme ist: In den beiden Differenzialgleichungen kommen jeweils beide Beschleunigungen vor. Durch Einführen der neuen Variablen

$$\begin{array}{l} \omega_1 = \dot{\phi}_1 \quad \text{und} \quad \omega_2 = \dot{\phi}_2 \\ \text{bzw.} \quad \dot{\omega}_1 = \ddot{\phi}_1 \quad \text{und} \quad \dot{\omega}_2 = \ddot{\phi}_2 \end{array}$$

wird aus den beiden Differenzialgleichungen 2. Ordnung auch hier ein System von 4 Differenzialgleichungen 1. Ordnung, von denen allerdings zwei in den Ableitungen gekoppelt sind. Das Differenzialgleichungssystem kann z. B. so formuliert werden:

$$\begin{array}{l} \dot{\phi}_1 = \omega_1 \\ \dot{\phi}_2 = \omega_2 \\ a_{11} \dot{\omega}_1 + a_{12} \dot{\omega}_2 = b_1 \\ a_{12} \dot{\omega}_1 + a_{22} \dot{\omega}_2 = b_2 \end{array}$$

mit den entsprechenden Ausdrücken für a_{11} , a_{12} , a_{22} , b_1 und b_2 , die aus den Differenzialgleichungen abzulesen sind. Auf Seite 694 in "Dankert/Dankert: Technische Mechanik" wird gezeigt, wie man auf bequeme Art die Gleichungen nach den ersten Ableitungen auflösen kann.

Besonders komfortabel arbeitet man allerdings mit Programmen, denen man die gekoppelten Gleichungen direkt anbieten kann. Die ODE-Solver von MATLAB sind z. B. auf die Lösung von Systemen der Art

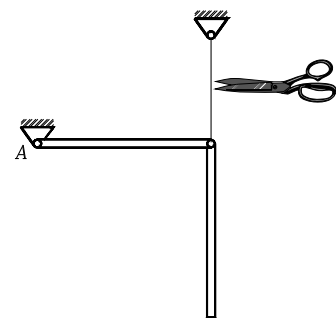
$$M(t,y) y' = f(t,y)$$

eingrichtet. Für das Doppelpendel kann das System so aufgeschrieben werden:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a_{11} & a_{12} \\ 0 & 0 & a_{12} & a_{22} \end{bmatrix} \begin{bmatrix} \dot{\phi}_1 \\ \dot{\phi}_2 \\ \dot{\omega}_1 \\ \dot{\omega}_2 \end{bmatrix} = \begin{bmatrix} \omega_1 \\ \omega_2 \\ b_1 \\ b_2 \end{bmatrix}$$

Für die MATLAB-ODE-Solver müssen dann jeweils eine Function, die die Matrix M definiert, und eine Function, die die "rechte Seite" definiert, bereitgestellt werden. Dies muss über die bereits in anderer Eigenschaft verwendete **odeset**-Function angekündigt werden. Auf der folgenden Seite ist das komplette MATLAB-Script zu sehen:

- Es wurde als "Function file" geschrieben, um auch die beiden zusätzlich erforderlichen Functions in dieser Datei unterbringen zu können.
- Mit **nargin** wird geprüft, ob die Funktion **Doppelpendel** mit den beiden Input-Parametern (Anfangswerte für die beiden Koordinaten) aufgerufen wurde, anderenfalls werden Standardwerte (nebenstehende Skizze) benutzt.
- Die Parameter (Massen, Längen, Trägheitsmomente), die das Doppelpendel charakterisieren, werden **global** deklariert, damit Änderungen der Zahlenwerte nur an einer Stelle nötig sind.



```

c:\netobjects fusion 4.0\user sites\vmcu\dateien\Doppelpendel.m
File Edit View Text Debug Breakpoints Web Window Help
Stack: Base
1 function DoppelPendel (philAnf , phi2Anf)
2
3 if nargin == 2 % Start mit vorgegebenen Anfangswerten?
4 x0 = [philAnf ; phi2Anf ; 0 ; 0] ; % Anfangswerte [phil ; phi2 ; omega1 ; omega2]
5 else
6 x0 = [pi/2 ; 0 ; 0 ; 0] ; % Default-Anfangswerte
7 end
8
9 % Parameter (global definiert, damit Wertaenderung nur an einer Stelle erforderlich):
10 global m dm J1 J2 s1 s2 gdl
11 m dm = 1 ; J1 = 1./12 ; J2 = 1./12 ; s1 = .5 ; s2 = .5 ; gdl = 9.81 ;
12
13 tspan = [0 10] ; % Zeitintervall
14
15 % Integration des Anfangswertproblems:
16 options = odeset('Mass', @Massenmatrix , 'MaxStep', 0.01) ;
17 [t x] = ode45 (@RechteSeite , tspan , x0 , options) ;
18
19 % Grafische Ausgabe von phil und phi2, zunaechst "Sortieren der Ergebnisse":
20 phili = [1 0 0 0] * x' ; % x' ist die transponierte Matrix der Ergebnisse,
21 phi2i = [0 1 0 0] * x' ; % phili und phi2i sind Vektoren
22
23 subplot(2,1,1) ; plot (t , phili) , grid on , title ('phil-t-Diagramm:')
24 subplot(2,1,2) ; plot (t , phi2i) , grid on , title ('phi2-t-Diagramm:')
25
26 % =====
27 % Funktion, die die "Massenmatrix" des Dgl.-Systems definiert:
28 function M = Massenmatrix (t , x)
29
30 global m dm J1 J2 s1 s2 gdl
31
32 c = cos(x{1}-x{2}) ;
33 M = [1 0 0 0 ; 0 1 0 0 ; 0 0 s1^2+J1+m dm*s2*c ; 0 0 dm*s2*c m dm*s2^2+J2] ;
34
35 % =====
36 % Funktion, die die "rechte Seite" des Dgl.-Systems definiert:
37 function f = RechteSeite (t , x)
38
39 global m dm J1 J2 s1 s2 gdl
40
41 s = sin(x{1}-x{2}) ;
42 f = [x{3} ; x{4} ; -m dm*s2*x{4}^2*s-(s1+m dm)*gdl*s*sin(x{1}) ; m dm*s2*x{3}^2*s-m dm*s2*gdl*s*sin(x{2})] ;
43
Ready

```

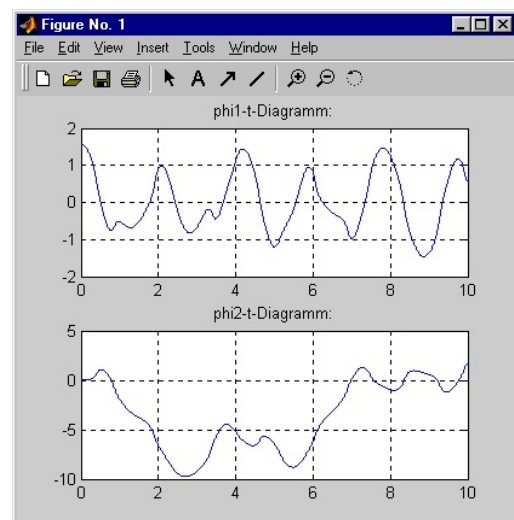
- Mit `odeset` wird (neben der in diesem Fall zwingend zu setzenden "MaxStep-Option") mit der "Mass-Option" angekündigt, dass ein System der Art

$$M(t,y) y' = f(t,y)$$

gelöst werden soll und dass die Matrix M in der Funktion `Massenmatrix` definiert wird. Diese Function wird mit ihrem "Handle" (Zeichen @, gefolgt vom Namen der Function) angekündigt.

- In Zeile 17 wird die Funktion `RechteSeite` (auch via "Handle" angekündigt, so dass die beiden ab Zeile 28 bzw. ab Zeile 37 definierten Funktionen gemeinsam das Differentialgleichungssystem definieren.

Nebenstehend sind die recht bizarren Funktionsverläufe zu sehen. Man erkennt, dass das untere Pendel mit einem "Salto" beginnt, dem nach wenigen Sekunden ein "Salto rückwärts" folgt.

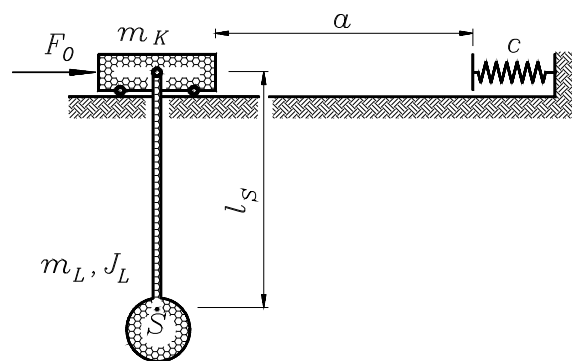


Ereignisse (“Events”)

Typische Ereignisse, die während der Bewegung eintreten können, sind zum Beispiel:

- ◆ Antriebskräfte oder -momente werden zugeschaltet oder abgeschaltet.
- ◆ Die Bewegung kommt (vorübergehend) zum Stillstand.
- ◆ Eine Masse schlägt an eine Feder an bzw. entfernt sich wieder.
- ◆ Eine Masse kommt in einen Bereich mit plötzlich geänderten Bewegungswiderstand (z. B. geänderten Gleitreibungskoeffizienten).

Es ist naheliegend, in der Function, in der die Differenzialgleichungen ausgewertet werden, die Parameter in Abhängigkeit von der Zeit, dem Ort bzw. der Geschwindigkeit (alle Informationen stehen in dieser Function bereit) jeweils an die aktuellen Werte anzupassen. Diese Strategie funktioniert in den meisten Fällen auch.

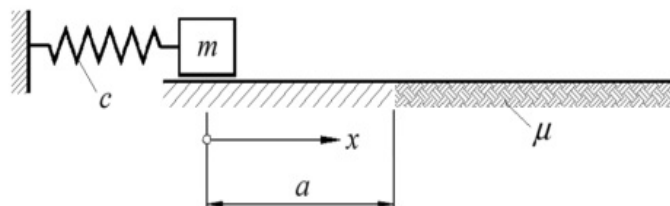


So kann die nebenstehend skizzierte Aufgabe (entnommen aus “Dankert/Dankert: Technische Mechanik”, Seite 697) durchaus so gelöst

werden, wobei die beiden Ereignisse (Antriebskraft F_0 wird nach 1 s abgeschaltet, Laufkatze stößt nach dem Zurücklegen der Strecke a an die Feder) wie oben beschrieben durch Abfrage von Zeit bzw. Weg in der Function “bemerkt” werden. Die Differenzialgleichungen sollten also unter Berücksichtigung von Antriebskraft und Feder formuliert werden, und F_0 bzw. c sollte jeweils in Abhängigkeit von der Zeit bzw. des Weges der Laufkatze der gegebene Wert oder der Wert 0 zugewiesen werden.

“Ganz sauber” ist diese Strategie nicht (allerdings sehr einfach zu realisieren und deshalb meistens “einen Versuch wert”). Probleme können dadurch auftauchen, dass die Lösungsalgorithmen in einem Zeitschritt mehrere Funktionswertberechnungen zu verschiedenen Zeitpunkten abfordern (beim Runge-Kutta-Verfahren 4. Ordnung z. B. am Anfang, in der Mitte und am Ende des Zeitschritts). Das Ereignis kann (und wird in der Regel) innerhalb eines Zeitschritts liegen, so dass die Funktionswertberechnungen mit unterschiedlichen Parametern ausgeführt werden. Insbesondere bei Algorithmen mit automatischer Schrittweitensteuerung (wie z. B. die MATLAB-ode-Functions) kann dies zu einem “Festfahren” der Rechnung führen. Die Problematik soll an folgendem kleinen Beispiel demonstriert werden.

Beispiel: Die skizzierte Masse m ist durch eine Feder gefesselt und kann sich im linken Bereich ($x < a$) reibungsfrei auf der Unterlage bewegen, im rechten Bereich ($x \geq a$) ist Gleitreibung mit dem Gleitreibungskoeffizienten μ zu berücksichtigen. In der skizzierten Lage ist die Feder entspannt.



Die Masse m wird um $x_0 = 2a$ ausgelenkt und ohne Anfangsgeschwindigkeit freigelassen. Man stelle die Weg-Zeit- und die Geschwindigkeits-Zeit-Funktion der Masse m für die ersten 10 Sekunden der Bewegung graphisch dar.

Geg.: $m = 20 \text{ kg}$; $c = 300 \text{ N/m}$; $\mu = 0,4$; $a = 0,5 \text{ m}$; $g = 9,81 \text{ m/s}^2$.

Das Problem kann durch eine Bewegungsdifferentialgleichung

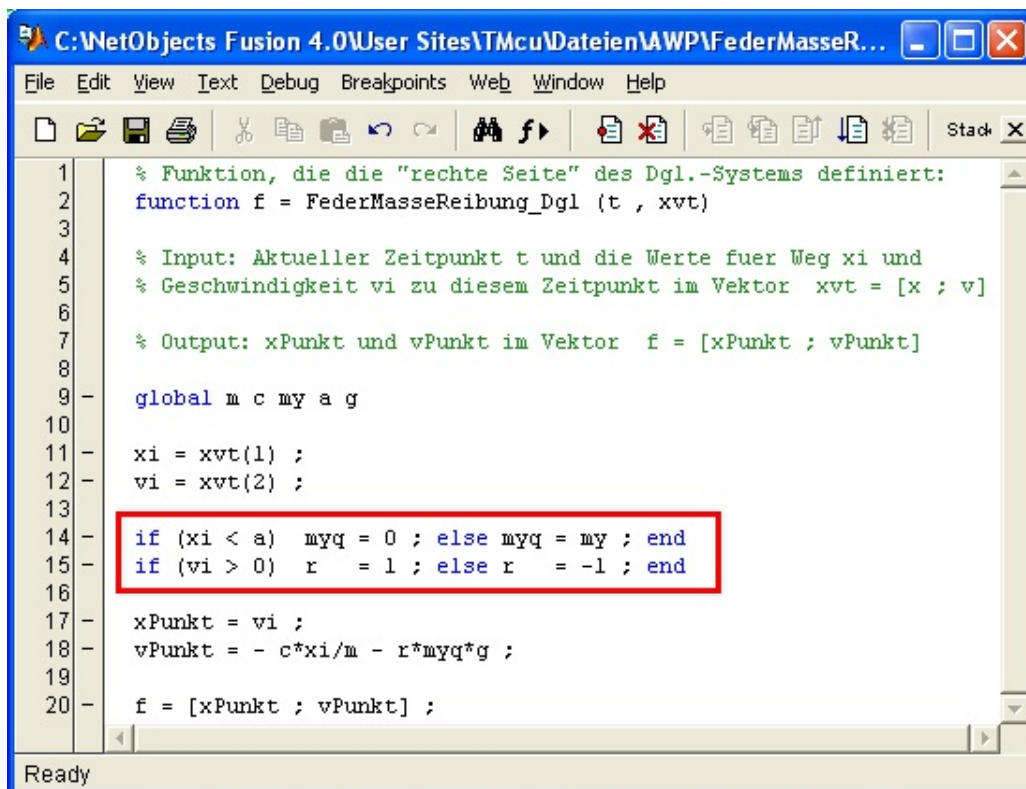
$$m\ddot{x} + cx + r\mu mg = 0$$

beschrieben werden, wenn für die Parameter μ und r folgende Abhängigkeiten berücksichtigt werden:

- $\mu = 0$ für $x < a$ und $\mu = 0,4$ für $x \geq a$,
- $r = 1$ bei Bewegung nach rechts (positive Geschwindigkeit) und $r = -1$ bei Bewegung nach links.

Unter www.juergendankert.de/TM3/Aufgabe_28-5/aufgabe_28-5.html wird die Lösung dieser Aufgabe mit MATLAB ausführlich behandelt. Hier soll das Fazit aus den verschiedenen Lösungsstrategien gezogen werden.

Die nachfolgend zu sehende Function realisiert die Auswertung der beiden Differentialgleichungen 1. Ordnung mit der "nicht ganz sauberen" Strategie:



```

1  % Funktion, die die "rechte Seite" des Dgl.-Systems definiert:
2  function f = FederMasseReibung_Dgl (t , xvt)
3
4  % Input: Aktueller Zeitpunkt t und die Werte fuer Weg xi und
5  % Geschwindigkeit vi zu diesem Zeitpunkt im Vektor  xvt = [x ; v]
6
7  % Output: xPunkt und vPunkt im Vektor  f = [xPunkt ; vPunkt]
8
9  global m c my a g
10
11  xi = xvt(1) ;
12  vi = xvt(2) ;
13
14  if (xi < a)  myq = 0 ; else myq = my ; end
15  if (vi > 0)  r   = 1 ; else r   = -1 ; end
16
17  xPunkt = vi ;
18  vPunkt = - c*xi/m - r*myq*g ;
19
20  f = [xPunkt ; vPunkt] ;

```

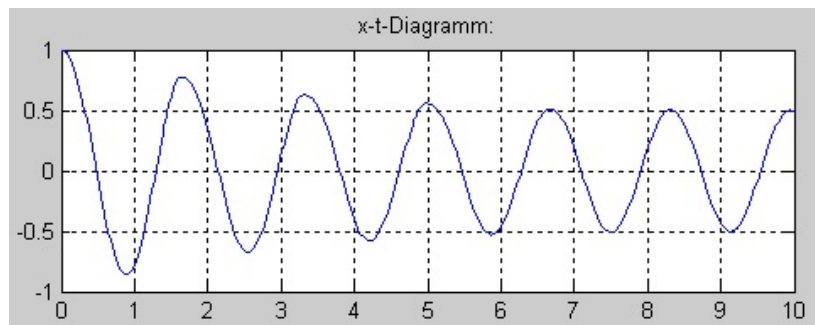
Die beiden umrahmten Zeilen sorgen dafür, dass für μ und r immer die passenden Werte verwendet werden, wenn in den nachfolgenden Zeilen 17 und 18 die Werte für die Ableitungen von Geschwindigkeit und Weg berechnet werden. Das nachfolgend zu sehende MATLAB-Script löst die Differentialgleichungen unter Verwendung der ode45-Funktion:

```

C:\NetObjects Fusion 4.0\User Sites\TMcu\Dateien\AWP\FederMasseReibun...
File Edit View Text Debug Breakpoints Web Window Help
[Icons] Stack: Base X
1 - clear all
2
3   % Parameter (global definiert, damit Wertaenderung nur an
4   % einer Stelle erforderlich):
5 - global m c my a g
6 - m = 20 ; c = 300 ; my = 0.4 ; a = 0.5 ; g = 9.81 ;
7
8 - x0    = [2*a ; 0] ;      % Anfangswerte
9 - tspan = [0 10] ;      % Zeitintervall
10
11 % Integration des Anfangswertproblems:
12 - options = odeset ('MaxStep' , 0.1) ;
13 - [t xv] = ode45 ('FederMasseReibung_Dgl' , tspan , x0 , options) ;
14
15 % Grafische Ausgabe, zunaechst "Sortieren der Ergebnisse":
16 - x = [1 0] * xv' ; % xv' ist die transponierte Matrix der Ergebnisse,
17 - v = [0 1] * xv' ;
18
19 - subplot(2,1,1) ; plot (t , x) , grid on , title ('x-t-Diagramm:')
20 - subplot(2,1,2) ; plot (t , v) , grid on , title ('v-t-Diagramm:')
21
22 - Zeitschritte = length (t)
23 - End_x = x(Zeitschritte)
24 - End_v = v(Zeitschritte)
Ready

```

Das Ergebnis der Rechnung entspricht den Erwartungen:



Das Weg-Zeit-Diagramm zeigt die (infolge der Reibung) kürzer werdenden Amplituden bis zu dem Zeitpunkt, zu dem die schwingende Masse den Bereich $x \geq a = 0,5 \text{ m}$ nicht mehr erreicht. Weil dann keine Reibung mehr zu berücksichtigen ist, bleiben ab diesem Zeitpunkt die Ausschläge konstant.

Aber schon kleine Änderungen der Parameter können das Verfahren überfordern. Die Masse kommt im Bereich $x \geq a$ bei jeder Bewegungsumkehr kurz zum Stillstand, und bei größerem Reibungskoeffizienten (oder geringerer Federsteifigkeit) kann die Situation eintreten, dass die Federkraft nicht ausreichend ist, um die Reibkraft zu überwinden (ganz korrekt müsste man im Moment des Stillstands sogar überprüfen, ob die Haftungskraft mit dem in der Regel größeren Haftungskoeffizienten überwunden werden kann). Aber dieser Zeitpunkt wird natürlich nie genau getroffen, was zur Folge hat, dass die MATLAB-ode45-Funktion die Schrittweite an dieser Stelle

ständig verfeinert, so dass man schließlich die Rechnung abbrechen muss (Ctrl-C im Command Window bricht eine MATLAB-Rechnung ab).

Für das betrachtete Beispiel tritt ein solcher Fall z. B. auf, wenn man nur den Reibungskoeffizienten auf den Wert $\mu = 0,8$ ändert. Während dies für die MATLAB-ode45-Funktion dann schon ein Fall für den Ctrl-C-Abbruch ist, führt ein Verfahren mit konstanter Schrittweite (z. B. die Funktion rk4) bei sehr feiner Schrittweite "nur zu einem Hin-und-Her-Wackeln auf der Stelle", was natürlich auch kein befriedigendes Ergebnis ist. Wirklich sauber kann man solche Situationen mit dem MATLAB-Event-Handling abfangen.

MATLAB-Event-Strategie

Die MATLAB-Event-Strategie wird an dem gerade behandelten Beispiel demonstriert. Unter www.juergendankert.de/TM3/Aufgabe_28-5/aufgabe_28-5.html findet man die komplette Lösung in dem Function-File FederMasseReibung2.m. Hier wird an Ausschnitten aus dieser Datei die Strategie erläutert:

Es werden drei Ereignisse ("Events") definiert, auf die die Rechnung besonders reagieren soll:

- ◆ Ereignis 1: "Geschwindigkeit Null",
- ◆ Ereignis 2: "Punkt $x = a$ wird passiert in Richtung $x > a$ ",
- ◆ Ereignis 3: "Punkt $x = a$ wird passiert in Richtung $x < a$ "

Die Ereignisse müssen in einer speziell dafür zu schreibenden Function (mit beliebigem Namen) definiert werden, und der MATLAB-ode-Function muss über die Optionen mitgeteilt werden, dass eine solche Ereignis-Function ausgewertet werden soll. Das Schlüsselwort für diese Option heißt "Events", so dass eine odeset-Zeile (hier auch mit der stets empfohlenen MaxStep-Option)

```
options = odeset ('MaxStep' , 0.1 , 'Events' , @Ereignis) ;
```

ankündigt, dass eine Function mit dem Namen "Ereignis" die auszuwertenden Ereignisse definiert. Diese kann z. B. so aussehen:

```
63 | % =====
64 | function [value,isterminal,direction] = Ereignis (t,xvt)
65 | - global m c my a g r myq
66 |
67 | - x = xvt(1) ;
68 | - v = xvt(2) ;
69 | - value(1)      = v ; % Ereignis 1: "Geschwindigkeit Null"
70 | - isterminal(1) = 1 ; % Abbruch ...
71 | - direction(1)  = 0 ; % ... in jedem Fall
72 |
73 | - value(2)      = x-a ; % Ereignis 2: "Punkt x=a erreicht"
74 | - isterminal(2) = 1 ; % Abbruch, ...
75 | - direction(2)  = 1 ; % ... wenn Bewegung nach rechts
76 |
77 | - value(3)      = x-a ; % Ereignis 3: "Punkt x=a erreicht"
78 | - isterminal(3) = 1 ; % Abbruch ...
79 | - direction(3)  = -1 ; % ... wenn Bewegung nach links
```

Die Function bekommt als Input die gleichen Werte (Zeit und Vektor der Bewegungsgrößen) wie die Function für die Definition der Differenzialgleichungen (hier: FederMasseReibung_Dgl). Sie

muss für jedes Ereignis 3 Werte abliefern: Eine “Nullfunktion” (hier auf Vektor **value**) definiert durch ihre Nullstelle das Ereignis, außerdem: Was soll beim Eintreten dieses Ereignisses geschehen (hier auf **isterminal**: 1 bedeutet Abbruch, 0 würde Weiterrechnung bedeuten), und soll dies in jedem Fall (0 auf **direction**) geschehen oder nur beim Nulldurchgang in positive Werte (1) oder nur beim Nulldurchgang in negative Werte (-1)?

Alle Ereignisse führen also in diesem Fall zum Abbruch der Rechnung. Die ode-Funktion liefert bei gesetzter Event-Option drei zusätzliche Ergebnisse:

```
[t xv te xve ereig] = ode45(@FederMasseReibung_Dgl , tspan , x0 , options) ;
```

Auf **te** werden die Zeiten abgeliefert, zu denen die Ereignisse eingetreten sind, auf **xve** findet man die Zustandsgrößen der Bewegung zu diesen Zeitpunkten (hier: Wegkoordinaten und Geschwindigkeiten), der Vektor **ereig** enthält die Ereignisnummern (hier 1, 2 oder 3).

Wenn die Ereignisse nicht zum Abbruch führen, kann man also nach Beendigung der Rechnung erfahren, welche Ereignisse wann aufgetreten sind und welche Werte die Bewegungsgrößen zu diesen Zeitpunkten hatten. Wenn aber (wie in diesem Fall) jedes Ereignis zum Abbruch führt, muss jeweils entschieden werden, ob und wie die Rechnung fortgesetzt werden soll. Dabei beschreiben immer die letzten Werte in den drei Output-Parametern **te**, **xve** und **ereig** das gerade eingetretene Ereignis. Diese Informationen werden ausgewertet, um die Parameter (hier **myq** und **r**) zu aktualisieren:

```
30 -   lent = length (t) ;
31 -   if (t(lent) >= tend)
32 -       cont = 0 ;           % Ende erreicht, Abbruch der while-Schleife
33 -   else                     % ... war ein "Ereignis" der Grund fuer den Abbruch
34 -       telen = length (te) ;
35 -       tspan = [te(telen) tend] ;           % ... weiter mit "Restintervall ..."
36 -       x0     = [xve(telen,1) xve(telen,2)] ; % ... und neuen Anfangsbedingungen
37 -       if ereig(telen) == 1           % Ereignis "Geschwindigkeit Null"
38 -           if abs(c*xve(telen,1)) < my0*m*g ...
39 -               & xve(telen,1) > a           % Federkraft kleiner Haftkraft, ...
40 -               cont = 0 ;                   % ... also Abbruch
41 -           tout = [tout ; tend] ;
42 -           xvout = [xvout ; [xv(lent,1) 0]] ;
43 -           elseif xve(telen,1) < 0 r=1 ;     % Es geht nach rechts weiter ...
44 -           else r=-1 ;                       % ... bzw. nach links
45 -           end
46 -       elseif ereig(telen) == 2 myq=my ;     % Ereignis "Ab sofort Reibung"
47 -       elseif ereig(telen) == 3 myq=0 ;     % Ereignis "Ab sofort reibungsfrei"
48 -       end
49 -   end
```

Weil die Rechnung mehrfach abgebrochen wird, ist die ode45-Funktion in eine while-Schleife eingebettet. Die Ergebnisse, die nach jedem ode45-Aufruf auf dem Vektor **t** und der Matrix **xv** abgeliefert werden, werden deshalb auf **tout** und **xvout** kumuliert:

Nach jedem Abbruch vor dem Ende des Integrationsbereichs wird die Rechnung innerhalb der while-Schleife neu gestartet (beim Erreichen des Endes wird über `cont = 0` - Zeile 40 im oben zu sehenden Ausschnitt - das Ende der while-Schleife initiiert). Dafür muss das Restintervall an ode45 übergeben, und als Anfangsbedingungen müssen die Werte zum Zeitpunkt des Abbruchs angesetzt werden. Dies wird in den Zeilen 35 und 36 des oben zu sehenden Ausschnitts realisiert. Der folgende Ausschnitt zeigt den Start der beschriebenen while-Schleife:

```

21 - tout = [] ; % Auf tout und xvout werden die Ergebnisse gesammelt
22 - xvout = [] ;
23 - cont = 1 ;
24
25 - while cont == 1
26 - [t xv te xve ereig] = ode45 (@FederMasseReibung_Dgl , tspan , x0 , options) ;
27 - tout = [tout ; t] ; % Die gerade abgelieferten Ergebnisse werden an ...
28 - xvout = [xvout ; xv] ; % ... die bereits gespeicherten Ergebnisse angehaengt
29
30 - lent = length (t) ;
31 - ...

```

Die Frage, ob die Bewegung nach dem Stillstand im Bereich $x \geq a$ fortgesetzt wird, kann in diesem Fall mit dem dafür zuständigen Haftungskoeffizienten gestellt werden.

Die komplette Datei ist unter der oben angegebenen Internet-Adresse mit ausführlichen Erläuterungen zu sehen und steht zum Download bereit. Mit den im vorigen Abschnitt gegebenen Werten der Aufgabenstellung (Aufgabe 28-5) ergeben sich die gleichen Ergebnisse wie mit dem Script mit der “nicht ganz sauberen Lösung”. Mit wesentlich vergrößertem Reibungskoeffizienten, für die das im vorigen Abschnitt angegebene Script versagte, liefert die Function `FederMasseReibung2.m` die zu erwartenden Ergebnisse.

Die Function kann aus dem Command Window mit zwei Parametern aufgerufen werden (Gleitreibungskoeffizient und Haftreibungskoeffizient). Im abgebildeten Command Window ist der Aufruf mit $\mu = 0,6$ und $\mu_0 = 0,8$ zu sehen, der zu dem rechts zu sehenden Ergebnis führt.

Die Masse kommt nach einigen Sekunden zur Ruhe, im Command Window sieht man den genauen Wert für die Endlage.

