

Jürgen Dankert:

Numerische Integration von Anfangswertproblemen

Teil 1: Grundlagen

Dieses Skript gehört zu den Internet-Ergänzungen des
Lehrbuchs „Dankert/Dankert: Technische Mechanik“

Inhaltsverzeichnis

1	Anfangswertprobleme	3
2	Das Verfahren von Euler-Cauchy	3
3	Differenzialgleichungen höherer Ordnung, Differenzialgleichungssysteme	8
4	Verifizieren der Berechnungsergebnisse	10
5	Verbesserte Integrationsformeln	13
5.1	Prädiktor-Korrektor-Verfahren, das Verfahren von Heun	13
5.2	Klassisches Runge-Kutta-Verfahren	15
5.3	Verlagerung des Algorithmus in eine Matlab-Function	18
5.4	Schrittweitenproblem, Richardson-Extrapolation	23
6	Einschrittverfahren	25
6.1	Runge-Kutta-Familie, Butcher-Array	26
6.2	Eingebettete Verfahren	27
6.3	Dormand-Prince-Verfahren, Standardsolver in Matlab	30
7	Mehrschrittverfahren	35
7.1	Methoden von Adams-Bashforth (explizite Verfahren)	35
7.2	Adams-Moulton- und Adams-Bashforth-Moulton-Verfahren (implizite Verfahren)	36
7.3	Verfahren von Gear	37
8	Verfahrenswahl	38

1 Anfangswertprobleme

Bei einer **gewöhnlichen Differentialgleichung** n -ter Ordnung

$$y^{(n)} = f(x, y, y', y'', \dots, y^{(n-1)}) \quad (1.1)$$

enthält die allgemeine Lösung n Integrationskonstanten. Es können n zusätzliche Bedingungen formuliert werden, mit denen diese Integrationskonstanten zu bestimmen sind, so dass sich die spezielle Lösung des durch Differentialgleichung und Zusatzbedingungen beschriebenen Problems ergibt.

Wenn alle Zusatzbedingungen für die gleichen Stelle x_0 gegeben sind (der Funktionswert y und die Ableitungen bis zur $(n - 1)$ -ten Ordnung sind an dieser Stelle vorgeschrieben), dann spricht man von einem **Anfangswertproblem** (im Gegensatz zum **Randwertproblem**, bei dem diese Bedingungen für unterschiedliche x -Werte gegeben sind). Die mit dem **Differenzenverfahren** zu lösenden Aufgaben der Biegetheorie sind z. B. typische lineare Randwertaufgaben (Differentialgleichungen und Randbedingungen sind linear), für die eine geschlossene Lösung prinzipiell möglich ist, auch wenn komplizierte praxisnahe Probleme eine numerische Lösung nahe legen.

Für **nichtlineare Differentialgleichungen** ist eine geschlossene Lösung nur in ganz seltenen Ausnahmefällen möglich. Auch Näherungsmethoden wie das Differenzenverfahren sind nicht praktikabel, weil sich sehr große nichtlineare Gleichungssysteme ergeben würden. Wenn die Aufgabe jedoch als Anfangswertproblem formuliert ist, lassen sich auf numerischem Wege brauchbare Näherungslösungen gewinnen. Glücklicherweise sind gerade viele nichtlineare Probleme der Ingenieur-Mathematik Anfangswertprobleme, für die hier geeignete Näherungsverfahren behandelt werden.

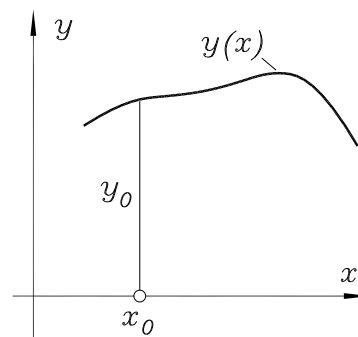
2 Das Verfahren von Euler-Cauchy

Die Idee der numerischen Integration soll zunächst am einfachsten Anfangswertproblem mit dem einfachsten Verfahren vorgestellt werden. Für das Anfangswertproblem 1. Ordnung

$$y' = f(x, y) \quad , \quad y(x_0) = y_0 \quad (2.1)$$

(eine Differentialgleichung 1. Ordnung und die dazugehörige Anfangsbedingung) ist die Funktion $y(x)$ gesucht, die die Differentialgleichung und die Anfangsbedingung erfüllt (nebenstehende Skizze).

Ausgehend vom einzigen x -Wert, für den der gesuchte y -Wert bekannt ist, dem „Anfangspunkt“ x_0 mit dem Wert y_0 , sucht man einen Wert y_1 für die Stelle $x_1 = x_0 + h$ (h ist die „Schrittweite“), um anschließend auf gleiche Weise zum nächsten Punkt zu kommen usw.



Anfangswertproblem 1. Ordnung: Ein Punkt der Lösungsfunktion ist gegeben

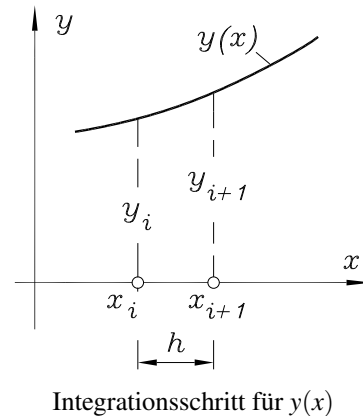
Dieser Prozess sei bis zur Stelle x_i abgelaufen, y_i ist also bekannt. Dann ist das Berechnen von y_{i+1} für $x_{i+1} = x_i + h$ der typische Integrationsschritt des Verfahrens.

Beide Seiten der Differentialgleichung des Anfangswertproblems werden über das Intervall h integriert:

$$\int_{x=x_i}^{x_{i+1}} y'(x) dx = \int_{x=x_i}^{x_{i+1}} f(x,y) dx \quad ,$$

$$[y(x)]_{x_i}^{x_{i+1}} = y_{i+1} - y_i = \int_{x=x_i}^{x_{i+1}} f(x,y) dx \quad ,$$

$$y_{i+1} = y_i + \int_{x=x_i}^{x_{i+1}} f(x,y) dx \quad .$$



Das verbleibende Integral auf der rechten Seite, das den Zuwachs des Funktionswertes vom Punkt i zum Punkt $i + 1$ repräsentiert, muss näherungsweise gelöst werden, weil die im Integranden enthaltene Funktion $y(x)$ nicht bekannt ist. Die verschiedenen Verfahren der numerischen Integration von Anfangswertproblemen unterscheiden sich im Wesentlichen in der Art und Qualität, wie dieses Integral angenähert wird.

Die grösste Näherung für das Integral ist die Annahme, der Integrand $f(x,y)$ sei im gesamten Integrationsintervall $x_i \leq x \leq x_{i+1}$ konstant und kann durch den Wert $f(x_i, y_i)$ am linken Rand des Integrationsintervalls ersetzt werden (x_i und y_i sind bekannt). Mit

$$\int_{x=x_i}^{x_{i+1}} f(x,y) dx \approx \int_{x=x_i}^{x_{i+1}} f(x_i, y_i) dx = [x f(x_i, y_i)]_{x_i}^{x_{i+1}} = f(x_i, y_i)(x_{i+1} - x_i) = y'_i h \quad (2.2)$$

erhält man die

Integrationsformel von Euler-Cauchy:

$$y_{i+1} = y_i + y'_i h \quad , \quad x_{i+1} = x_i + h \quad . \quad (2.3)$$

Dies ist die einfachste Näherungsformel für die numerische Integration eines Anfangswertproblems, die Lösung $y(x)$ wird durch einen Polygonzug approximiert. Die einfache Berechnungsvorschrift verdeutlicht in besonderer Schärfe das Problem aller Integrationsformeln für Anfangswertprobleme: Die Näherungslösung für das Integral erzeugt einen Fehler („Quadratfehler“), der in die Berechnung von y' für den nächsten Integrationsschritt eingeht und dabei einen weiteren Fehler (Steigungsfehler) erzeugt.

Andererseits wird auch die Stärke dieser Verfahren deutlich: Eine einfache, immer wieder auf die gerade berechneten Werte angewendete Formel kommt der Programmierung in hohem Maße entgegen. Weil jeder Schritt nur die Ergebnisse seines Vorgängers kennen muss, ist der Speicherplatzbedarf außerordentlich gering, wenn nicht alle berechneten Werte für eine nachfolgende Auswertung gespeichert werden müssen.

Beispiel:

Das Problem, die Funktion $y(x)$ zu bestimmen, die einen Spiegel definiert, der paralleles Licht so ablenkt, dass sich alle Lichtstrahlen in einem Punkt (Fokus) treffen, kann ohne Einschränkung der Allgemeinheit für Lichtstrahlen parallel zur x -Achse und mit dem Nullpunkt als Fokus formuliert werden.

Einige einfache geometrische Überlegungen (nachfolgende Skizze) führen auf die Differentialgleichung

$$y' = \frac{y}{x + \sqrt{x^2 + y^2}} ,$$

für die (mit einiger Mühe) die allgemeine Lösung berechnet werden kann, so dass sich die nichtlineare Differentialgleichung vorzüglich für eine Abschätzung der Genauigkeit einer numerischen Lösung eignet.

Auf der Internet-Seite ["Differentialgleichungen 1. Ordnung, Beispiel: Parabolspiegel"](#) wird die allgemeine Lösung hergeleitet:

$$y^2 = C^2 + 2Cx$$

beschreibt eine Schar quadratischer Parabeln.

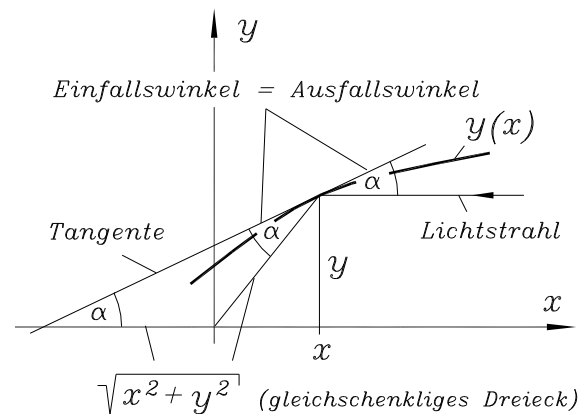
Die Integrationskonstante C kann ziemlich willkürlich festgelegt werden. Wählt man zum Beispiel als Anfangsbedingung $y(0) = 1$, dann erhält man mit $C = 1$ als spezielle Lösung die quadratische Parabel

$$y = \pm\sqrt{1 + 2x} .$$

Die nebenstehende Tabelle zeigt die numerische Lösung nach Euler-Cauchy für das nichtlineare Anfangswertproblem

$$y' = \frac{y}{x + \sqrt{x^2 + y^2}} , \quad y(0) = 1$$

mit der sehr groben Schrittweite $h = 0,1$ im Vergleich mit der exakten Lösung.



Zur x -Achse paralleler Lichtstrahl wird an der Funktion $y(x)$ so gespiegelt, dass er durch den Nullpunkt verläuft

i	x_i	y_i	y'_i	$y_{i,exakt}$
0	0,0	1,0000	1,0000	1,0000
1	0,1	1,1000	0,9132	1,0954
2	0,2	1,1913	0,8461	1,1832
3	0,3	1,2759	0,7921	1,2649
...
10	1,0	1,7560		1,7321

Schon am Ende des sehr kurzen Integrationsintervalls $x = 0 \dots 1$ zeigt sich eine sichtbare Abweichung, die bei größeren Integrationsintervallen stärker wird, sich durch kleinere Schrittweiten jedoch verringern lässt. Die nachfolgende Tabelle zeigt die Ergebnisse, die sich bei verschiedenen Schrittweiten am Ende des Integrationsintervalls $x = 0 \dots 5$ ergeben:

Schrittweite	$h =$	1,0	0,1	0,01	0,001	Exakte Lösung
Integrationsschritte	$n =$	5	50	500	5000	
$y(5)$		3,9163	3,3723	3,3221	3,3172	3,3166

Natürlich kann man die Anzahl der Integrationsschritte nicht beliebig erhöhen, weil mit der Anzahl der Rechenoperationen auch der mit jeder Operation unvermeidlich verknüpfte Rundungsfehler das Ergebnis verfälschen wird.

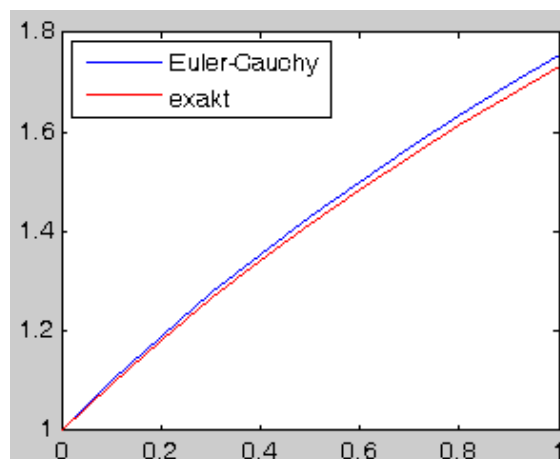
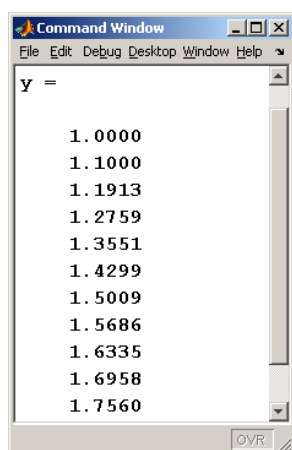
Das nachfolgend angegebene kleine Matlab-Script (als [ECSpiegel.m](#) zum Download verfügbar), mit dem diese Berechnung realisiert werden kann, zeigt, wie einfach das Verfahren zu programmieren ist:

```

1  % Euler-Cauchy-Verfahren fuer Differenzialgleichung y'=f(x,y),
2  % hier "Spiegel-Problem":
3
4  f = inline ('y/(x+sqrt(x^2+y^2))','x','y') ;
5
6  x0 = 0 ; y0 = 1 ;           % Anfangswert
7  xEnd = 1 ;                 % Ende des Integrationsintervalls
8  n = 10 ;                  % Anzahl der Integrationssschritte
9  h = (xEnd - x0) / n ;     % Schrittweite
10
11 x = x0 : h : xEnd ;
12 y = zeros (n+1,1) ;
13
14 % Euler-Cauchy-Algorithmus -----
15 y(1) = y0 ;
16
17 for i=1:n
18     y(i+1) = y(i) + h * f (x(i),y(i)) ;
19 end
20 % Euler-Cauchy-Algorithmus -----
21
22 % Exakte Lösung zum Vergleich:
23 C = -x0 + sqrt(x0^2+y0^2) ;
24 yexakt = sqrt (C^2+2*C*x) ;
25
26 y % Ausgabe der Ergebnisse in das Command Window und
27 % gemeinsam mit der exakten Loesung in ein Graphik-Fenster:
28 plot (x,y,x,yexakt,'r') ; legend ('Euler-Cauchy','exakt',2) ;

```

- Der gesamte Euler-Cauchy-Algorithmus besteht aus wenigen Zeilen. In Zeile 15 wird der Anfangswert gesetzt, die übrigen y-Werte werden in der Schleife von Zeile 17 bis 19 berechnet.
- Die spezielle Differenzialgleichung, die gelöst wird, wurde am Anfang (Zeile 4) als „Inline-Funktion“ definiert, die gegebenenfalls für die Lösung eines anderen Problems ersetzt werden kann. Bei komplizierteren Differenzialgleichungen sollte jedoch die Definition mit einer Matlab-Funktion realisiert werden.
- In den Zeilen 11 und 12 werden die beiden Vektoren x und y entsprechend der Anzahl der Integrationsschritte n mit $n+1$ Elementen definiert. In Zeile 11 wird x mit äquidistanten Werten bestückt, der Vektor y wird in Zeile 12 mit Nullen vorbelegt.
- Die Ergebnisse (y -Werte) werden in das Command Window ausgegeben (Zeile 26). Außerdem wird die Funktion $y(x)$ gemeinsam mit den Werten der exakten Lösung in ein Graphik-Fenster gezeichnet (Zeile 28):



Dargestellt sind die Ergebnisse für die Rechnung mit der sehr groben Schrittweite $h = 0,1$. Bei kleinerer Schrittweite sind die Abweichungen von der exakten Lösung wesentlich geringer.

An diesem kleinen Beispiel kann man außerdem ein Problem verdeutlichen, auf das im Kapitel „Kinetik des Massenpunktes“ hingewiesen wird: Bei Berechnung dieses Anfangswertproblems mit negativer Schrittweite (um die Spiegelform auch links vom Fokus zu bestimmen), ist für die Stelle $x = -0,5$ ein Versagen der Rechnung zu erwarten, weil dort y' unendlich wird. Durch die unvermeidlichen Rundungsfehler bei der Rechnung äußert sich dies unter Umständen „nur“ durch unsinnige Ergebnisse.

Dies ist ein generelles Problem bei der Lösung von Differentialgleichungen in Bereichen, in denen $y' = f(x,y)$ sich „mit y sehr stark ändert“. Für die Eindeutigkeit der Lösung einer Differentialgleichung muss gefordert werden, dass in dem betrachteten Bereich die partielle Ableitung von $f(x,y)$ nach y entsprechend

$$\left| \frac{\partial f(x,y)}{\partial y} \right| \leq K \quad (2.4)$$

begrenzt ist (so genannte „Lipschitz-Bedingung“).

Im Scheitelpunkt der betrachteten Lösungsfunktion ist diese Bedingung nicht erfüllt. Die exakte Lösung verzweigt sich dort auch in einen oberen und einen unteren Zweig.

Das Verfahren von Euler-Cauchy wurde nur aus didaktischen Gründen hier so ausführlich behandelt. Für die weitaus meisten praktischen Probleme gibt es keinen Grund, nicht eines der noch zu behandelnden genaueren Verfahren zu verwenden. Im folgenden Abschnitt wird allerdings das speziell für die Technische Mechanik wichtige Problem der Behandlung von Differentialgleichungen höherer Ordnung und Differentialgleichungssystemen auch mit dem Euler-Cauchy-Verfahren demonstriert.

3 Differenzialgleichungen höherer Ordnung, Differenzialgleichungssysteme

Die Euler-Cauchy-Formel ist problemlos auf Differenzialgleichungssysteme 1. Ordnung übertragbar, indem sie für jede der zu berechnenden Funktionen aufgeschrieben wird. Für ein Anfangswertproblem mit zwei Differenzialgleichungen 1. Ordnung

$$\begin{aligned} y_1' &= f_1(x, y_1, y_2) & , & & y_1(x_0) &= y_{1,0} & , \\ y_2' &= f_2(x, y_1, y_2) & , & & y_2(x_0) &= y_{2,0} \end{aligned} \quad (3.1)$$

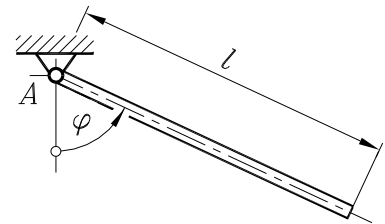
wird die Euler-Cauchy-Formel in jedem Integrationsschritt zweimal verwendet, so dass $y_{1,i+1}$ und $y_{2,i+1}$ aus $y_{1,i}$ und $y_{2,i}$ berechnet werden können.

Damit ist auch eine Möglichkeit der numerischen Integration von Differenzialgleichungen (und Differenzialgleichungssystemen) höherer Ordnung gegeben: Durch Einführen von zusätzlichen Variablen für die ersten Ableitungen werden Differenzialgleichungen höherer Ordnung in ein Differenzialgleichungssystem 1. Ordnung überführt.

Dies soll am Beispiel einer einfachen Bewegungs-Differenzialgleichung demonstriert werden. Bewegungs-Differenzialgleichungen der Technischen Mechanik sind stets Differenzialgleichungen 2. Ordnung (Beschleunigung ist 2. Ableitung der Wegkoordinate). Für die numerische Integration bedeutet das, dass neben der Wegkoordinate (z. B.: s oder bei Drehbewegungen φ) auch noch die Geschwindigkeit (z. B.: $v = \dot{s}$ bzw. $\omega = \dot{\varphi}$) als Variable auftritt, so dass das Beschleunigungsglied durch die erste Ableitung der Geschwindigkeit ersetzt wird. Die unabhängige Variable in Bewegungs-Differenzialgleichungen ist die Zeit t .

Beispiel:

Ein dünner Stab der Länge l mit konstantem Querschnitt ist an einem Ende reibungsfrei gelagert. Er wird aus der vertikalen Lage um den Winkel φ_0 ausgelenkt und ohne Anfangsgeschwindigkeit freigegeben. Die freie Schwingung wird durch das Anfangswertproblem (Herleitung findet man im Kapitel „Kinetik starrer Körper“)



$$\ddot{\varphi} = -\frac{3g}{2l} \sin \varphi \quad , \quad \varphi(t=0) = \varphi_0 \quad , \quad \dot{\varphi}(t=0) = 0$$

beschrieben (die unabhängige Variable t taucht in der Differenzialgleichung gar nicht auf, dies ist sehr häufig bei Bewegungs-Differenzialgleichungen). Durch Einführen einer zusätzlichen abhängigen Variablen $\omega = \dot{\varphi}$ wird aus der Differenzialgleichung 2. Ordnung ein Differenzialgleichungssystem 1. Ordnung:

$$\begin{aligned} \dot{\varphi} &= \omega & ; & & \varphi(t=0) &= \varphi_0 & ; \\ \dot{\omega} &= -\frac{3g}{2l} \sin \varphi & ; & & \omega(t=0) &= 0 & . \end{aligned}$$

Dieses Anfangswertproblem kann zum Beispiel mit dem folgenden Euler-Cauchy-Formelsatz berechnet werden (zum Problem passend ist die Zeit t die unabhängige Variable, an Stelle der Schrittweite h wird Δt geschrieben, und für y_1 und y_2 werden φ bzw. ω verwendet):

$$\begin{aligned}\varphi_{i+1} &= \varphi_i + \Delta t \dot{\varphi}_i \quad , \\ \omega_{i+1} &= \omega_i + \Delta t \dot{\omega}_i \quad , \\ t_{i+1} &= t_i + \Delta t \quad .\end{aligned}$$

Die Programmierung ist im Vergleich zu einem Problem mit nur einer Differentialgleichung nicht nennenswert aufwendiger, wie man an dem folgenden MATLAB-Script sieht (steht als [ECPendel.m](#) zum Download zur Verfügung):

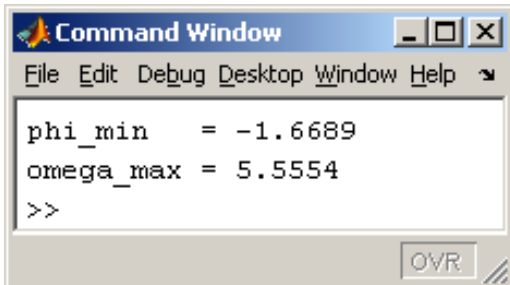
```

1  % Euler-Cauchy-Verfahren fuer Stab-Pendel mit großen Ausschlaegen :
2
3  phip   = inline ('omega','omega') ;
4  omegap = inline ('-faktor*sin(phi)','t','phi','omega','faktor') ;
5
6  pl     = 1 ; g = 9.81 ;                % Pendellaenge , Erdbeschleunigung
7  faktor = 1.5*g/pl ;
8
9  t0 = 0 ; phi0 = pi/2 ; omega0 = 0 ;    % Anfangswerte
10 tEnd = 1 ;                               % Ende des Integrationsintervalls
11 n   = 100 ;                               % Anzahl der Integrations Schritte
12 dt  = (tEnd - t0) / n ;
13
14 t = t0 : dt : tEnd ;
15 phi = zeros (n+1,1) ;
16 omega = zeros (n+1,1) ;
17
18 % Euler-Cauchy-Algorithmus -----
19 phi(1) = phi0 ; omega(1) = omega0 ;
20
21 for i=1:n
22     phi (i+1) = phi (i) + dt * phip (omega(i)) ;
23     omega(i+1) = omega(i) + dt * omegap (t(i),phi(i),omega(i),faktor) ;
24 end
25 % Euler-Cauchy-Algorithmus -----
26
27 disp ([ 'phi_min_==_' , num2str(min(phi))]) ;           % Extremwerte ---->
28 disp ([ 'omega_max_==_' , num2str(max(abs(omega)))] ) ; % Command Window
29 plot (t , phi) , title ('\phi(t)') ;                   % Graphik phi(t)

```

Nachfolgend sieht man die Ergebnisse der Rechnung mit diesem Script. Dabei wurden (wie im Listing zu sehen) eine Stablänge $l = 1$ m und eine Anfangsauslenkung $\varphi_0 = \pi/2$ angenommen. Es wurde über ein Zeitintervall von nur 1 s gerechnet (es ergibt sich etwa eine halbe Pendelschwingung).

Man sieht in diesem Fall dem Ergebnis sofort an, dass es sehr ungenau ist (immerhin wurde das kurze Zeitintervall in 100 Abschnitte unterteilt), denn natürlich kann das Pendel zur anderen Seite nicht über $\varphi = -\pi/2$ hinaus ausschlagen. In der Regel sind solche Ungenauigkeiten dem Ergebnis allerdings nicht sofort anzusehen. Die hier mit dem primitiven Verfahren zu gewinnende Erkenntnis ist allgemeinerungsfähig (gilt auch für beliebig leistungsfähige Verfahren): Ergebnisse der numerischen Integration von Anfangswertproblemen müssen immer verifiziert werden.

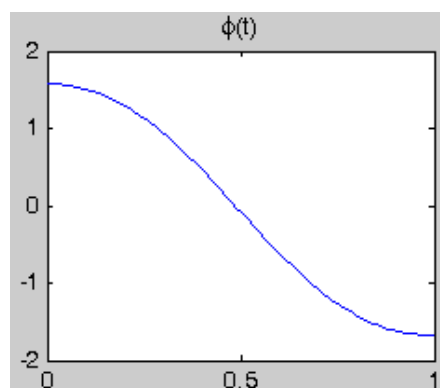


```

Command Window
File Edit Debug Desktop Window Help
phi_min   = -1.6689
omega_max = 5.5554
>>
OVR

```

Selbst mit 100 Schritten kann das Euler-Cauchy-Verfahren eine halbe Pendelschwingung nicht ausreichend genau berechnen.



4 Verifizieren der Berechnungsergebnisse

Im Kapitel „Verifizieren von Computerrechnungen“ wird ein ganzer Katalog von Möglichkeiten zusammengestellt und an Beispielen demonstriert, wie man solche Berechnungen verifizieren kann. Dies ist für kaum einen anderen Problemkreis so wichtig wie für die numerische Integration von Anfangswertproblemen. Einige der dort empfohlenen Möglichkeiten können auf das gerade berechnete Pendelproblem angewendet werden:

- ⇒ Die **Berechnung mit verschiedenen Verfahren** kann besonders empfohlen werden. In den nachfolgenden Abschnitten werden weitere (wesentlich leistungsfähigere) Integrationsverfahren vorgestellt, und die Berechnung des Pendelproblems wird noch mehrfach bei Vergleich mit den hier gewonnenen Ergebnissen durchgeführt.
- ⇒ Bei Diskretisierungsverfahren ist es beinahe zwingend, **mehrere Rechnungen mit unterschiedlichen Schrittweiten** durchzuführen. Wenn sich die Ergebnisse am Ende des Integrationsintervalls bei halbiertem Schrittweite nicht wesentlich ändern, kann man der Rechnung vertrauen. Bei dem gerade behandelten Beispiel kann man den zunächst sehr verdächtigen Wert φ_{min} beobachten.
- ⇒ Die Frage, ob sich **Teilergebnisse mit erträglichem Aufwand kontrollieren lassen**, führt hier zu folgender Überlegung: Während sich für das einfache Schwingungsproblem die Bewegungsgesetze $\varphi(t)$ und $\omega(t)$ nur numerisch berechnen lassen, kann das Bewegungsgesetz $\omega(\varphi)$ auch für beliebig große Anfangsauslenkungen mit Hilfe des Energiesatzes direkt formuliert werden. Speziell liefert diese Betrachtung für die (maximale) Winkelgeschwindigkeit bei Durchgang durch die tiefste Lage des Pendels ($\varphi = 0$) aus

$$mg \frac{l}{2} (1 - \cos \varphi_0) = \frac{1}{2} \left(\frac{1}{3} ml^2 \right) \omega_{max}^2$$

den zu erwartenden Wert

$$\omega_{max} = \sqrt{\frac{3g}{l} (1 - \cos \varphi_0)} \quad .$$

Mit den verwendeten Zahlenwerten $l = 1 \text{ m}$ und $g = 9,81 \text{ m/s}^2$ müsste sich bei einer Anfangsauslenkung $\varphi_0 = \pi/2$ also eine maximale Winkelgeschwindigkeit $\omega_{max} = 5,4249 \text{ s}^{-1}$ ergeben.

- ⇨ Schließlich kann sogar noch eine zusätzlich **Kontrollfunktion** erzeugt werden. Weil keine Energie während der Bewegung zu- oder abgeführt wird, muss die zum Startzeitpunkt vorhandene Energie konstant bleiben. Für eine beliebige Lage φ kann die Summe aus potenzieller und kinetischer Energie aufgeschrieben werden. Wenn man das Null-Potenzial auf die Höhe des Aufhängepunktes A legt, gilt:

$$T_{ges} = -mg \frac{l}{2} \cos \varphi + \frac{1}{2} J_A \omega^2 = ml \left(-\frac{l}{2} \cos \varphi + \frac{l}{6} \omega^2 \right) .$$

Darin ist $J_A = ml^2/3$ das Massenträgheitsmoment des dünnen Stabs bezüglich des Punktes A (siehe Kapitel „Kinetik starrer Körper“). Wenn man die Funktion $T_{ges}(t)$ mit den berechneten Werten $\varphi(t)$ und $\omega(t)$ zeichnet, müsste sie bei korrekten Werten eine horizontale Gerade liefern.

- ⇨ Der Empfehlung, eine **Vergleichsrechnung mit einem vereinfachten Berechnungsmodell durchzuführen, für das die Lösung bekannt ist**, kann man hier folgen, indem man eine Rechnung mit kleiner Anfangsauslenkung φ_0 durchführt. Für kleine Anfangsauslenkung kann die Differenzialgleichung wegen $\sin \varphi \approx \varphi$ linearisiert werden:

$$\ddot{\varphi} = -\frac{3g}{2l} \sin \varphi \quad \Rightarrow \quad \ddot{\varphi} + \frac{3g}{2l} \varphi = 0 .$$

Dies ist eine lineare Differenzialgleichung und kann geschlossen gelöst werden. Man findet ihre Lösung, aus der man die Dauer T einer vollen Schwingung ablesen kann, zum Beispiel im Bereich [Mathematik für die Technische Mechanik](#):

$$\varphi = \varphi_0 \cos \left(\sqrt{\frac{3g}{2l}} t \right) \quad \Rightarrow \quad T = \frac{2\pi}{\sqrt{\frac{3g}{2l}}} = 2\pi \sqrt{\frac{2l}{3g}} .$$

Für die verwendeten Zahlenwerte $l = 1 \text{ m}$ und $g = 9,81 \text{ m/s}^2$ ist also eine Schwingungsdauer $T = 1,638 \text{ s}$ zu erwarten bzw. $T/2 = 0,819 \text{ s}$ als Zeit für eine halbe Schwingung.

Alle Kontrollempfehlungen sind realisiert in dem nur geringfügig erweiterten Matlab-Script [ECPendel2.m](#), das sich von dem auf Seite 9 gelisteten Script nur in dem Auswertungsbereich ab Zeile 27 unterscheidet:

```

26
27 % Erweiterte Auswertung, Verifizieren der Ergebnisse:
28 Tges = - g/2*cos(phi) + pl*omega.^2/6 ; % Kontrollfunktion "Gesamtenergie"
29
30 [phimin , imin] = min(phi) ;
31 disp ([ 'Anzahl_der_Zeitschritte_n_=_', num2str(n)] ) ;
32 disp ([ 'phi_0_=_', num2str(phi0)] ) ;
33 disp ([ 'phi_min_=_', num2str(phimin) , '_bei_t_=_', num2str(t(imin))]) ;
34 disp ([ 'omega_max_=_', num2str(max(abs(omega)))] ) ;
35 subplot (2,1,1) ; plot ( t , phi) , title ('\phi(t)') ;
36 subplot (2,1,2) ; plot ( t , Tges) , title ('T_{ges}(t)') ;

```

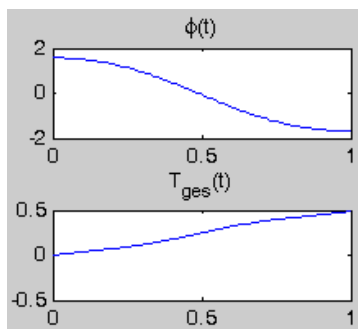
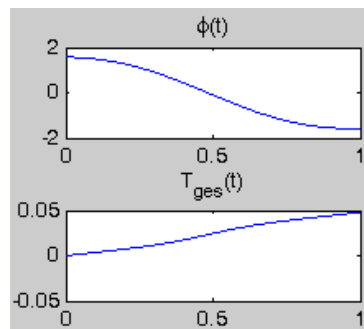
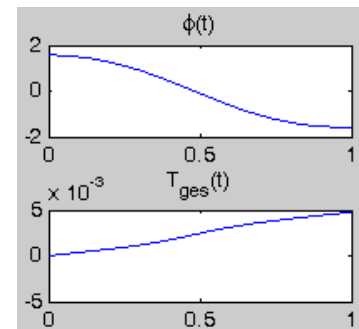
Die nebenstehende Kopie des Command Windows zeigt die Ergebnisse von drei Rechnungen mit $n = 100$, $n = 1000$ und $n = 10000$ Schritten. Man sieht, dass sich der Maximalausschlag nach links dem zu erwartenden Wert $\varphi_{min} = -\pi/2$ nähert. Der Wert für die maximale Winkelgeschwindigkeit verbessert sich auch erheblich.

Nachfolgend sieht man die zugehörigen Graphik-Ausgaben. Bemerkenswert ist die Kontrollfunktion T_{ges} , die scheinbar für alle Rechnungen gleich schlechte Ergebnisse attestiert. Der Blick auf die Werte, die auf den vertikalen Skalen angezeigt werden, zeigt aber doch wesentliche Verbesserungen.

```

Command Window
File Edit Debug Desktop Window Help
Anzahl der Zeitschritte n = 100
phi_0      = 1.5708
phi_min    = -1.6689 bei t = 0.98
omega_max  = 5.5554
>>
Anzahl der Zeitschritte n = 1000
phi_0      = 1.5708
phi_min    = -1.5803 bei t = 0.968
omega_max  = 5.4378
>>
Anzahl der Zeitschritte n = 10000
phi_0      = 1.5708
phi_min    = -1.5717 bei t = 0.9668
omega_max  = 5.4262
>>
OVR

```

 $n = 100$  $n = 1000$  $n = 10000$

Es wird noch die Kontrolle mit einem vereinfachten Berechnungsmodell gezeigt: Für sehr kleine Anfangsauslenkungen ist, wie oben theoretisch gezeigt wurde, die für eine halbe Schwingung (erstmaliges Erreichen von φ_{min}) benötigte Zeit gleich $T/2 = 0,819$ s.

Das nebenstehende Command Window zeigt die Ergebnisse von zwei Rechnungen mit den Anfangsauslenkungen $\varphi_0 = 5^\circ$ bzw. $\varphi_0 = 1^\circ$. Der zu erwartende Wert wird recht gut genähert.

```

Command Window
File Edit Debug Desktop Window Help
Anzahl der Zeitschritte n = 1000
phi_0      = 0.087266
phi_min    = -0.087794 bei t = 0.82
omega_max  = 0.33566
>>
Anzahl der Zeitschritte n = 1000
phi_0      = 0.017453
phi_min    = -0.017559 bei t = 0.819
omega_max  = 0.067152
>>
OVR

```

Fazit: Das Verfahren von Euler-Cauchy ist selbst bei einer so einfachen Aufgabe sehr schnell überfordert. Es eignet sich aber sehr gut, um die Probleme der numerischen Integration von Anfangswertproblemen zu demonstrieren, die zwar von verbesserten Integrationsformeln abgemildert werden, aber nicht verschwinden.

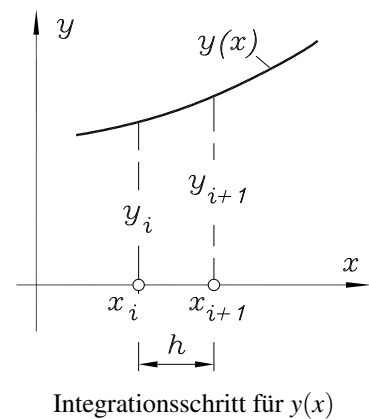
5 Verbesserte Integrationsformeln

5.1 Prädiktor-Korrektor-Verfahren, das Verfahren von Heun

Eine Verbesserung der Näherung für das Integral in

$$y_{i+1} = y_i + \int_{x_i}^{x_{i+1}} f(x, y) dx \quad (5.1)$$

kann durch das Einbeziehen weiterer Punkte des Integrationsintervalls h erreicht werden. Wenn der Integrand nicht nur durch einen Funktionswert (am linken Rand des Intervalls wie beim Verfahren von Euler-Cauchy) ersetzt wird, sondern z. B. auch der Funktionswert am rechten Rand $f(x_{i+1}, y_{i+1})$ in die Näherung einbezogen wird, kann der Integrand als linear veränderliche Größe angenähert werden („Rechteck“-Näherung wird zur deutlich besseren „Trapez“-Näherung). Mit



$$\int_{x_i}^{x_{i+1}} f(x, y) dx \approx \frac{f(x_i, y_i) + f(x_{i+1}, y_{i+1})}{2} (x_{i+1} - x_i) = (y'_i + y'_{i+1}) \frac{h}{2} \quad (5.2)$$

gelangt man zur wesentlich besseren Integrationsformel

$$y_{i+1} = y_i + (y'_i + y'_{i+1}) \frac{h}{2} \quad (5.3)$$

Diese Formel ist allerdings nicht ohne Vorleistung anwendbar, denn auf der rechten Seite geht über $y'_{i+1} = f(x_{i+1}, y_{i+1})$ der Wert y_{i+1} ein, der mit dieser Formel erst ermittelt werden soll. Man berechnet deshalb einen vorläufigen Näherungswert (*Prädiktor*) nach der Euler-Cauchy-Formel 2.3, der dann eine (gegebenenfalls mehrfache) Verbesserung nach der verbesserten Formel 5.3 erfährt (*Korrektor*-Schritte). Dies ist das

Verfahren von Heun:

$$\begin{array}{rcl}
 \text{Prädiktor:} & p_{i+1} = y_i + y'_i h & ; \quad x_{i+1} = x_i + h \\
 & \downarrow & \\
 p_{i+1} = y_{i+1} & \Rightarrow & y'_{i+1} = f(x_{i+1}, p_{i+1}) \\
 & \uparrow & \downarrow \\
 & \Leftarrow \Leftarrow \Leftarrow \Leftarrow & y_{i+1} = y_i + (y'_i + y'_{i+1}) \frac{h}{2} \quad (\text{Korrektor})
 \end{array} \quad (5.4)$$

Das Verfahren kann mit einer festen Anzahl von Korrektorschritten arbeiten oder aber einen Integrationsschritt erst dann beenden, wenn sich y_{i+1} nicht mehr ändert. Wie das Euler-Cauchy-Verfahren kann auch das Verfahren von Heun auf ein System von Differentialgleichungen angewendet werden, indem für jede Differentialgleichung der angegebene Algorithmus ausgeführt wird.

Die Programmierung ist nicht wesentlich aufwendiger als beim Verfahren von Euler-Cauchy. Im Listing des Euler-Cauchy-Verfahrens (Seite 9) müssen nur die Zeilen 18 bis 25 (Euler-Cauchy-Algorithmus) durch den Heun-Algorithmus ersetzt werden:

```

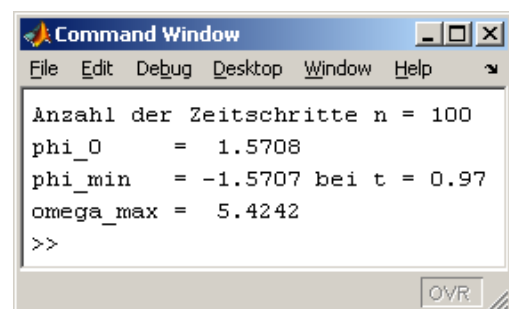
18 % Heun-Algorithmus -----
19 phi(1) = phi0 ; omega(1) = omega0 ;
20
21 for i=1:n
22     phipi = phip (omega(i)) ;
23     ompi = omegap (t(i), phi(i), omega(i), faktor) ;
24     % * Prädiktorschritt nach Euler-Cauchy: *****
25     prphi = phi (i) + dt * phipi ;
26     prom = omega(i) + dt * ompi ;
27     for j=1:3 % * Drei Korrektorschritte: *****
28         prphi = phi (i) + (phipi + phip(prom))*dt/2 ;
29         prom = omega(i) + (ompi + omegap(t(i+1), prphi ,prom , faktor))*dt/2 ;
30     end
31     phi (i+1) = prphi ; % ... die endgueltigen Werte des Schritts
32     omega(i+1) = prom ;
33 end
34 % Heun-Algorithmus -----

```

Das komplette Script, das das im vorigen Abschnitt behandelte Pendelproblem (2 Differentialgleichungen) löst und auch die für die Kontrolle der Ergebnisse ergänzten Zeilen enthält (siehe Listing auf Seite 11), ist als [HeunPendel.m](#) zum Download verfügbar. In jedem Integrationsschritt werden nach dem Prädiktorschritt genau 3 Korrektorschritte ausgeführt.

Wenn man mit dem auf diese Weise modifizierten Programm die gleichen Testrechnungen ausführt, die mit dem Programm nach Euler-Cauchy durchgeführt wurden, stellt man fest, dass schon bei wesentlich groberer Schrittweite die Genauigkeit erreicht wird, die nach Euler-Cauchy eine sehr feine Intervalleinteilung erfordert.

Bei einer Rechnung mit 100 Integrationschritten über ein Integrationsintervall von 1 s, die nach Euler-Cauchy äußerst ungenaue Ergebnisse lieferte, zeigt das nebenstehend zu sehende „Command Window“ etwa die theoretisch zu erwartenden Ergebnisse.



```

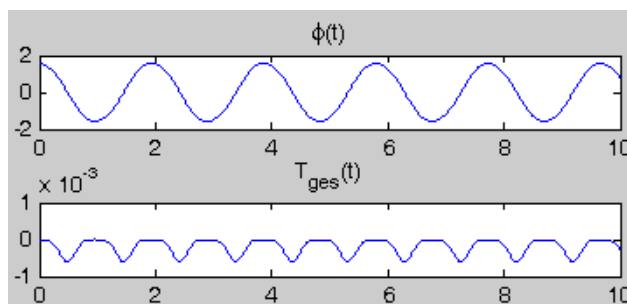
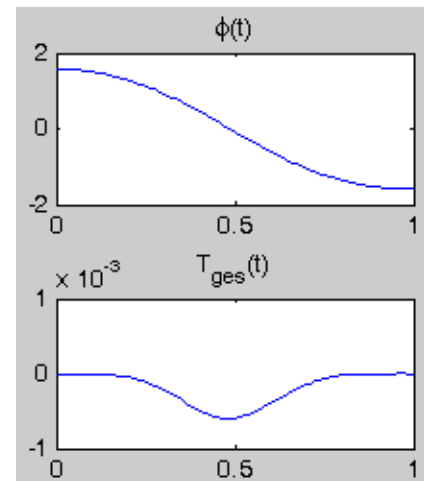
Command Window
File Edit Debug Desktop Window Help
Anzahl der Zeitschritte n = 100
phi_0      = 1.5708
phi_min    = -1.5707 bei t = 0.97
omega_max  = 5.4242
>>

```

Allerdings mahnt die nachfolgend unter der Funktion $\varphi(t)$ zu sehende Kontrollfunktion $T_{ges}(t)$ (Gesamtenergie während der Bewegung) nach wie vor zur Vorsicht.

Die $T_{ges}(t)$ -Werte auf der Abszisse sind immerhin sehr klein, und offensichtlich ist am Ende des Integrationsintervalls die Energie wieder auf dem Ausgangsniveau. Das bedeutet, dass im Integrationsbereich ϕ - ω -Wertepaare entstehen, die nicht exakt das Energiekriterium erfüllen. Weil aber das Ende des Integrationsintervalls immer auch der Startpunkt für die weitere Integration ist, stimmt das Energie-Kriterium in diesem Fall eher optimistisch für eine Integration über einen größeren Zeitbereich.

Unten sieht man eine Rechnung mit 1000 Integrationschritten über den wesentlich größeren Integrationsbereich von 10 s. Alle Kriterien sprechen dafür, dass diese Rechnung „gesund“ ist.



```

Command Window
File Edit Debug Desktop Window Help
Anzahl der Zeitschritte n = 1000
phi_0      = 1.5708
phi_min    = -1.5708 bei t = 2.9
omega_max  = 5.4246
>>
  
```

5.2 Klassisches Runge-Kutta-Verfahren

Mit den Verfahren von Euler-Cauchy und Heun wurden zwei einfache Vertreter einer kaum zu überblickenden Anzahl von Integrationsverfahren vorgestellt, an denen aber die typischen Probleme sichtbar werden, die der Anwender beachten muss. Die verschiedenen Verfahren unterscheiden sich im Wesentlichen in der Anzahl der Funktionswertberechnungen für den Integranden in einem Integrationsschritt (beeinflusst die Qualität der Näherung des Integrals), in der Strategie der Berechnung von y_{i+1} (feste oder variable Anzahl von Operationen) und in der Festlegung der Schrittweiten h für die Integrationsschritte (feste oder variable Schrittweite).

Nachfolgend werden die Formeln eines besonders häufig verwendeten Verfahrens einer ganzen Verfahrensklasse angegeben. Es ist ein Vertreter der Runge-Kutta-Algorithmen, die aus Funktionswerten für den Integranden in

$$y_{i+1} = y_i + \int_{x=x_i}^{x_{i+1}} f(x, y) dx \quad (5.5)$$

an verschiedenen Punkten des Integrationsintervalls h den Wert für y_{i+1} am rechten Intervallrand so ermitteln, dass die Genauigkeit der Berücksichtigung möglichst vieler Glieder der

Taylorreihen-Entwicklung der Lösung entspricht. Der folgende Formelsatz, der für einen Integrationsschritt vier Funktionswerte des Integranden berechnet, ist ein

Runge-Kutta-Verfahren 4. Ordnung:

$$\begin{aligned}
 y_{i+1} &= y_i + \frac{h}{6} (k_1 + 2k_2 + 2k_3 + k_4) \quad , \quad x_{i+1} = x_i + h \\
 \text{mit} \quad k_1 &= f(x_i, y_i) \quad , \\
 k_2 &= f\left(x_i + \frac{h}{2}, y_i + \frac{h}{2}k_1\right) \quad , \\
 k_3 &= f\left(x_i + \frac{h}{2}, y_i + \frac{h}{2}k_2\right) \quad , \\
 k_4 &= f(x_i + h, y_i + hk_3) \quad .
 \end{aligned} \tag{5.6}$$

Bei einem Verfahren 4. Ordnung (Übereinstimmung mit den ersten 5 Gliedern der Taylorreihen-Entwicklung) entsteht in jedem Integrationsschritt ein Fehler in der Größenordnung h^5 , der bei genügend kleiner Schrittweite sehr klein ist, andererseits reagiert das Verfahren bei zu großer Schrittweite sehr empfindlich. Gerade für die Runge-Kutta-Algorithmen gibt es eine recht ausgefeilte Theorie zur geeigneten Schrittweitenwahl (und damit auch der Schrittweitenänderung während der Rechnung), die allerdings den gravierenden Mangel hat, dass sie sichere Werte nur dann liefert, wenn die Lösung bekannt ist.

Weil hier nur Beispiele aus der Technischen Mechanik behandelt werden (das „Spiegel“-Beispiel im Abschnitt 2 war eine Ausnahme, weil die Technische Mechanik keine geeignete Differenzialgleichung 1. Ordnung liefern kann), soll der Runge-Kutta-Formelsatz 4. Ordnung für das typische Anfangswertproblem mit einer Bewegungs-Differenzialgleichung formuliert werden. Das typische Anfangswertproblem 2. Ordnung

$$\ddot{x} = f(x, \dot{x}, t) \quad , \quad x(t = t_0) = x_0 \quad , \quad \dot{x}(t = t_0) = v_0 \tag{5.7}$$

(x ist die Bewegungskordinate, \dot{x} die Geschwindigkeit, \ddot{x} die Beschleunigung, t ist die Zeit) wird durch Einführen ein neuen Variablen $v = \dot{x}$ zu einem Differenzialgleichungssystem 1. Ordnung:

$$\begin{aligned}
 \dot{x} &= v \quad , \quad x(t = t_0) = x_0 \quad , \\
 \dot{v} &= f(x, v, t) \quad , \quad v(t = t_0) = v_0 \quad .
 \end{aligned} \tag{5.8}$$

Dafür gilt (siehe auch Kapitel „Kinetik des Massenpunktes“) das

Runge-Kutta-Verfahren 4. Ordnung für eine Bewegungs-Differenzialgleichung:

$$\begin{aligned}
 x_{i+1} &= x_i + \frac{\Delta t}{6} (k_{1x} + 2k_{2x} + 2k_{3x} + k_{4x}) \quad , \quad t_{i+1} = t_i + \Delta t \quad , \\
 v_{i+1} &= v_i + \frac{\Delta t}{6} (k_{1v} + 2k_{2v} + 2k_{3v} + k_{4v}) \quad , \\
 k_{1x} &= f_1(t_i, x_i, v_i) \quad , \quad k_{1v} = f_2(t_i, x_i, v_i) \quad , \\
 k_{2x} &= f_1\left(t_i + \frac{\Delta t}{2}, x_i + \frac{\Delta t}{2}k_{1x}, v_i + \frac{\Delta t}{2}k_{1v}\right) \quad , \quad k_{2v} = f_2\left(t_i + \frac{\Delta t}{2}, x_i + \frac{\Delta t}{2}k_{1x}, v_i + \frac{\Delta t}{2}k_{1v}\right) \quad , \\
 k_{3x} &= f_1\left(t_i + \frac{\Delta t}{2}, x_i + \frac{\Delta t}{2}k_{2x}, v_i + \frac{\Delta t}{2}k_{2v}\right) \quad , \quad k_{3v} = f_2\left(t_i + \frac{\Delta t}{2}, x_i + \frac{\Delta t}{2}k_{2x}, v_i + \frac{\Delta t}{2}k_{2v}\right) \quad , \\
 k_{4x} &= f_1(t_i + \Delta t, x_i + \Delta t k_{3x}, v_i + \Delta t k_{3v}) \quad , \quad k_{4v} = f_2(t_i + \Delta t, x_i + \Delta t k_{3x}, v_i + \Delta t k_{3v}) \quad .
 \end{aligned} \tag{5.9}$$

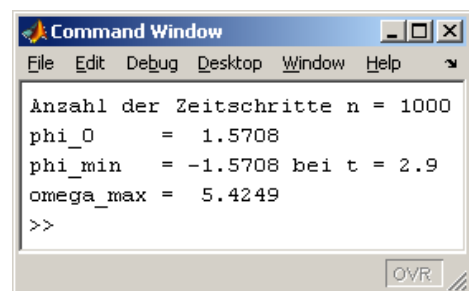
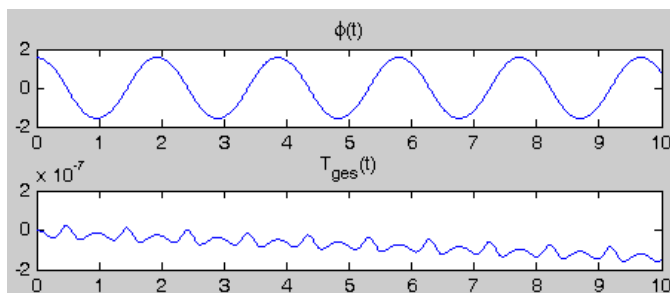
Dass dieser Formelsatz etwas unhandlich aussieht, ist unbedeutend, denn eine Handrechnung verbietet sich in der Regel von selbst, und die Programmierung muss einmal ausgeführt werden. Im Listing des Euler-Cauchy-Verfahrens (Seite 9) für die Berechnung des Stab-Pendels müssen nur die Zeilen 18 bis 25 (Euler-Cauchy-Algorithmus) durch den Runge-Kutta-Algorithmus 4. Ordnung ersetzt werden. Weil eine Drehbewegung berechnet wird, tritt an die Stelle der Koordinate x die Winkelkoordinate φ , an die Stelle der Geschwindigkeit v die Winkelgeschwindigkeit ω :

```

18 % Runge-Kutta-Algorithmus -----
19 phi(1) = phi0 ; omega(1) = omega0 ;
20
21 for i=1:n
22     k1phi = phip (omega(i)) ;
23     k1om = omegap (t(i), phi(i), omega(i), faktor) ;
24     k2phi = phip (omega(i)+k1om*dt/2) ;
25     k2om = omegap (t(i)+dt/2, phi(i)+k1phi*dt/2, omega(i)+k1om*dt/2, faktor) ;
26     k3phi = phip (omega(i)+k2om*dt/2) ;
27     k3om = omegap (t(i)+dt/2, phi(i)+k2phi*dt/2, omega(i)+k2om*dt/2, faktor) ;
28     k4phi = phip (omega(i)+k3om*dt) ;
29     k4om = omegap (t(i+1), phi(i)+k3phi*dt, omega(i)+k3om*dt, faktor) ;
30     phi (i+1) = phi (i) + (k1phi + 2*k2phi + 2*k3phi + k4phi) * dt/6 ;
31     omega(i+1) = omega(i) + (k1om + 2*k2om + 2*k3om + k4om) * dt/6 ;
32 end
33 % Runge-Kutta-Algorithmus -----

```

Das komplette Matlab-Script steht als [RKPendel.m](#) zum Download zur Verfügung. Mit ihm liefert eine Rechnung mit $n = 1000$ Integrationschritten die nachfolgend zu sehenden Ergebnisse.



Die Zahlenwerte im Command Window stimmen in allen ausgegebenen Stellen mit den erwarteten theoretischen Werten überein, sind also besser als die auf Seite 15 mit gleicher Schrittanzahl mit dem Verfahren von Heun erzeugten Ergebnisse.

Die unter der Funktion $\varphi(t)$ zu sehende Kontrollfunktion $T_{ges}(t)$ (Gesamtenergie während der Bewegung) sieht auf den ersten Blick allerdings schlechter aus als die Kontrollfunktion der entsprechenden Heun-Rechnung. Daran muss man sich gewöhnen, wenn man Matlab die freie Wahl eines geeigneten Maßstabs lässt. Tatsächlich sind die Abweichungen vom Sollwert bei der Runge-Kutta-Rechnung deutlich kleiner, wie ein Blick auf die Skala auf der Abszissenachse bestätigt. Den hier ermittelten Ergebnissen darf man vertrauen.

5.3 Verlagerung des Algorithmus in eine Matlab-Function

Für die Lösung unterschiedlicher Aufgabenstellungen bleibt der eigentliche Algorithmus der numerischen Integration (Euler-Cauchy, Heun, Runge-Kutta, ...) unverändert. Es ist deshalb ratsam, diesen in eine (unverändert immer wieder verwendbare) Function auszulagern. Weil darüber hinaus für kompliziertere Probleme die bisher praktizierte Definition der Differenzialgleichungen als Inline-Functions nicht mehr praktikabel ist, soll zunächst eine Einführung in die Arbeit mit Matlab-Functions gegeben werden.

Matlab kennt zwei Arten von M-Files:

- **Script-Files** akzeptieren keine Eingabe-Argumente und liefern keine Ausgabe-Argumente an andere Matlab-Files ab (alle bisher behandelten Beispiele wurden mit Script-Files realisiert).
- **Function-Files** akzeptieren Eingabe-Argumente und können Ausgabewerte an andere Matlab-Files 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. Zunächst werden die Inline-Functions durch „normale“ Matlab-Functions ersetzt.

In dem auf Seite 17 gelisteten Runge-Kutta-Algorithmus wurden die ersten Ableitungen, die die Differenzialgleichungen definieren, aus unterschiedlichen Werten für die Zeit, den Winkel, die Winkelgeschwindigkeit und den in der Variablen *faktor* steckenden Problemparametern je viermal berechnet. Dafür waren die beiden Inline-Functions *phip* und *omegap* zuständig:

```

phip = inline ('omega','omega') ;
omegap = inline ('-faktor*sin(phi)','t','phi','omega','faktor') ;
...
% Runge-Kutta-Algorithmus
phi(1) = phi0 ; omega(1) = omega0 ;

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) ;
% Runge-Kutta-Algorithmus

```

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:

```

1 function xvp = BewDglPendel ( t , xv )
2 global pl g % ... muessen an anderer Stelle definiert sein.
3 phi = xv(1) ; % Input-Werte kommen in ...
4 omega = xv(2) ; % ... einem Vektor an.
5 phip = omega ; % Die beiden Differenzialgleichungen
6 omegap = -1.5*g/pl*sin(phi) ; % werden ausgewertet, ...
7 xvp = [phip ; omegap] ; % ... die Ergebnisse werden im Vektor xvp 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 *xv* bezeichneten Inputvektors auf einfache „sprechende“ Variablen zu übertragen und auch die Auswertung der Differenzialgleichungen zunächst einzeln vorzunehmen und dann die berechneten Werte zum Outputvektor (hier: *xvp*) zusammenzustellen.

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 (natürlich dann mit mehr Anfangswerten):

```

% Runge-Kutta-Algorithmus -----
x(1,1) = phi0 ;
x(1,2) = omega0 ;

for i=1:n
    k1 = BewDglPendel ( t(i),x(i,:) ) ;
    k2 = BewDglPendel ( t(i)+dt/2,x(i,:)+k1'*dt/2 ) ;
    k3 = BewDglPendel ( t(i)+dt/2,x(i,:)+k2'*dt/2 ) ;
    k4 = BewDglPendel ( t(i+1),x(i,:)+k3'*dt ) ;
    x(i+1,:) = x ( i , : ) + ( k1' + 2*k2' + 2*k3' + k4' ) * dt/6 ;
end
% Runge-Kutta-Algorithmus -----

```

Deshalb ist es naheliegend, auch diesen Algorithmus in eine Function zu verlagern. Diese müsste als Input vier Information entgegennehmen: Anfangsbedingungen, das Zeitintervall,

für das die Lösung erwartet wird, die Anzahl der Abschnitte, in die dieses Intervall zu unterteilen ist und in welcher Function die Differenzialgleichungen definiert sind (hier die oben gelistete Function *BewDglPendel*). Diese Function könnte zum Beispiel so aussehen:

```

1 function [t , xv] = rk4 (funh , tspan , xv0 , n)
2
3 tanf = tspan(1) ;
4 tend = tspan(2) ;
5 dt   = (tend - tanf) / n ;
6 ndeq = length (xv0) ;
7
8 t    = tanf : dt : tend ;
9 xv   = zeros (n+1,ndeq) ;
10
11 % Runge-Kutta-Algorithmus -----
12 for i=1:ndeq
13     xv(1,i) = xv0(i) ;                               % Anfangsbedingungen
14 end
15
16 for i=1:n
17     k1 = feval (funh , t(i) , xv(i ,:)) ;
18     k2 = feval (funh , t(i)+dt/2 , xv(i ,:)+k1'*dt/2) ;
19     k3 = feval (funh , t(i)+dt/2 , xv(i ,:)+k2'*dt/2) ;
20     k4 = feval (funh , t(i+1) , xv(i ,:)+k3'*dt) ;
21     xv(i+1,:) = xv (i ,:) + (k1' + 2*k2' + 2*k3' + k4') * dt/6 ;
22 end
23 % Runge-Kutta-Algorithmus -----

```

Erläuterungen zu dieser Function *rk4*, die als [rk4.m](#) zum Download verfügbar ist:

- Der Zeitbereich, über den integriert werden soll, wird als Vektor *tspan* mit den beiden Werten t_{anf} und t_{end} angeliefert (siehe Zeilen 3 und 4).
- Im Vektor *xv0* übernimmt die Function *rk4* die Anfangsbedingungen. Die Länge des Vektors ist außerdem die Information über die Anzahl der Differenzialgleichungen *ndeq* (Zeile 6). Die Function *rk4* kann ein Differenzialgleichungssystem mit beliebig vielen Differenzialgleichungen 1. Ordnung bearbeiten (deshalb ist das Übernehmen der Anfangsbedingungen in die Ergebnismatrix in eine Schleife eingebettet, Zeilen 12 bis 14).
- Die Anzahl der Zeitschritte *n*, in die das Integrationsintervall unterteilt werden soll, wird in Zeile 5 für die Berechnung der Schrittweite *dt* genutzt.
- Der erste Input-Parameter ist der Name der Function, die die Differenzialgleichungen repräsentiert und die in der Runge-Kutta-Schleife mehrmals zur Berechnung mit unterschiedlichen Parametern genutzt wird. Das wird mit der von Matlab dafür vorgesehene 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 des *feval*-Aufrufs übergeben wird.

Der letzte Punkt bedarf noch einer Erläuterung: Der direkte Aufruf einer Function wie zum Beispiel

```
k1 = BewDglPendel ( t(i) , x(i ,:)) ;
```

ist mit dem indirekten Aufruf

```
k1 = feval ( funh , t(i) , x(i ,:)) ;
```

gleichwertig, wenn an *feval* auf der ersten Position der Parameterliste die Information übergeben wird, dass die Funktion *BewDglPendel* mit den nachfolgend angegebenen Parametern aufgerufen werden soll. Diese Information kann der Name des Function-Files sein, in dem *BewDglPendel* zu finden ist, oder ein „Function Handle“ (wird weiter unten erläutert).

Der Grund, warum in *rk4* die Funktion *BewDglPendel* nicht direkt aufgerufen wird, ist plausibel: *rk4* soll nicht nur für beliebige Integrationsintervalle und Anfangsbedingungen bei beliebig feiner Diskretisierung (Anzahl der Integrationsschritte) verwendbar sein, sondern auch für beliebige Differenzialgleichungssysteme 1. Ordnung, die in Functions definiert sind, die nicht *BewDglPendel* heißen müssen.

Die Funktion *rk4* liefert zwei Ergebnisse ab: In einem Vektor *t* befinden sich alle Zeitpunkte, für die Ergebnisse berechnet wurden (einschließlich Startzeitpunkt), bei *n* Integrationsschritten also *n+1* Werte. Dies sind bei *rk4* (konstante Schrittweite) äquidistante Werte. Die Matrix *xv* enthält in *ndeq* Spalten (*ndeq* ist die Anzahl der Differenzialgleichungen) die berechneten Ergebnisse, für eine Rechnung mit der Funktion *BewDglPendel* also eine Spalte mit den *n+1* φ -Werten und eine zweite Spalte mit den ω -Werten.

Nebstehend sieht man die Ergebnisse, die bei einer Rechnung über die Zeitspanne $t = 0 \dots 10$ und $n = 1000$ Integrationsschritten mit den Anfangswerten $\varphi_0 = \pi/2$ und $\omega_0 = 0$ abgeliefert werden.

<i>i</i>	Vektor <i>t</i>	Matrix <i>xv</i>	
0	$t_{anf} = 0$	$\varphi_0 = 1,5708$	$\omega_0 = 0$
1	$t_1 = 0,01$	$\varphi_1 = 1,5701$	$\omega_1 = -0,1471$
2	$t_2 = 0,02$	$\varphi_2 = 1,5679$	$\omega_2 = -0,2943$
3	$t_3 = 0,03$	$\varphi_3 = 1,5642$	$\omega_3 = -0,4414$
...
...
1001	$t_{end} = 10$	$\varphi_{end} = 0,7710$	$\omega_{end} = -4,5944$

Der Aufruf der Funktion *rk4* muss u. a. auf der ersten Position die Information über die zu verwendende Function für die Definition der Differenzialgleichungen enthalten, also z. B. den Verweis auf die Function *BewDglPendel*. Wenn diese Function in einer separaten Datei mit dem Namen *BewDglPendel.m* zu finden ist, muss der Dateiname angegeben werden:

$$[t, xv] = rk4 ('BewDglPendel', [t0; tEnd], [\varphi_0; \omega_0], n);$$

Im Allgemeinen ist es übersichtlicher, die Function *BewDglPendel* in der gleichen Datei (z. B.: *RKPendel4.m*) unterzubringen, die den gesamten Berechnungsablauf steuert. Dafür sind zwei Bedingungen zu erfüllen: Weil Functions in Matlab nur in Function-Files stehen dürfen, muss auch das Steuerprogramm als Function deklariert werden, indem man einfach

function RKPendel4

als erste Programmzeile schreibt. Der Aufruf der Funktion *rk4* erfolgt dann mit einem so genannten „Handle“ auf die Function *BewDglPendel*:

$$[t, xv] = rk4 (@BewDglPendel, [t0; tEnd], [\varphi_0; \omega_0], n);$$

Nachfolgend ist die komplette Datei gelistet (als [RKPendel4.m](#) zum Download verfügbar):

```

1 % Runge-Kutta-Verfahren 4. Ordnung fuer Stab-Pendel mit groeBen Ausschlaegen :
2
3 function RKPendel4 (nsteps)
4
5 if ( nargin < 1 ) n = 1000 ; % Anzahl der Integrationssschritte
6 else n = nsteps ;
7 end
8
9 global pl g
10 pl = 1 ; g = 9.81 ; % Pendellaenge , Erdbeschleunigung
11
12 t0 = 0 ; phi0 = pi/2 ; omega0 = 0 ; % Anfangswerte
13 tEnd = 10 ; % Ende des Integrationsintervalls
14
15 % Runge-Kutta-Algorithmus -----
16 [ t , xv ] = rk4 ( @BewDglPendel , [ t0 ; tEnd ] , [ phi0 ; omega0 ] , n ) ;
17 % Runge-Kutta-Algorithmus -----
18
19 phi = xv (:,1) ; % Sortieren der Ergebnisse:
20 omega = xv (:,2) ; % Spalte 1: phi , Spalte 2: omega
21
22 % Erweiterte Auswertung , Verifizieren der Ergebnisse:
23 Tges = - g/2*cos(phi) + pl*omega.^2/6 ; % Kontrollfunktion "Gesamtenergie"
24
25 [ phimin , imin ] = min(phi) ;
26 disp ( [ 'Anzahl der Zeitschritte_n=' , num2str(n) ] ) ;
27 disp ( [ 'phi_0=' , num2str(phi0) ] ) ;
28 disp ( [ 'phi_end=' , num2str(phi(end)) ] ) ;
29 disp ( [ 'omega_max=' , num2str(max(abs(omega))) ] ) ;
30 subplot ( 2,1,1 ) ; plot ( t , phi ) , title ( '\phi(t)' ) ;
31 subplot ( 2,1,2 ) ; plot ( t , Tges ) , title ( 'T_{ges}(t)' ) ;
32
33 %%%%%%%%%%% Funktion , die die Differenzialgleichungen definiert: %%%%%%%%%%%
34 function xvp = BewDglPendel ( t , xv )
35 global pl g % ... sind in der Function RKPendel4 (oben) definiert.
36 phi = xv(1) ; % Input-Werte kommen in ...
37 omega = xv(2) ; % ... einem Vektor an.
38 phip = omega ; % Die beiden Differenzialgleichungen
39 omegap = -1.5*g/pl*sin(phi) ; % werden ausgewertet , ...
40 xvp = [ phip ; omegap ] ; % ... die Ergebnisse werden im Vektor xvp abgeliefert.

```

- Es arbeiten also drei Functions zusammen, die in zwei Dateien untergebracht sind: In der Datei *RKPendel4.m* befinden sich die Functions *RKPendel4* und *BewDglPendel*. Diese enthalten alle Informationen über das aktuelle Problem einschließlich der Problemparame-ter. Bei einer Modifizierung der Parameter oder bei Lösung eines ganz anderen Problems müssen die Änderungen ausschließlich in dieser Datei vorgenommen werden.

Der Algorithmus zur Realisierung der numerischen Integration (hier: *rk4*) befindet sich in der separaten Datei *rk4.m*, die nicht wieder angefasst werden muss. Sie muss nur verfügbar sein, indem sie z. B. im gleichen Verzeichnis wie *RKPendel4.m* steht.

- Weil das Steuerprogramm ohnehin als Function deklariert werden muss, wenn man auch die Function *BewDglPendel* in der gleichen Datei unterbringen will, kann man noch einen Schritt weiter gehen: Functions können auch direkt aus dem Command Window aufgerufen und von dort mit Parametern versorgt werden. Das wird in der oben geliste-ten Function *RKPendel4* mit dem Parameter *nsteps* demonstriert (Zeile 3), mit dem die Anzahl der Integrationssschritte gesteuert werden kann.

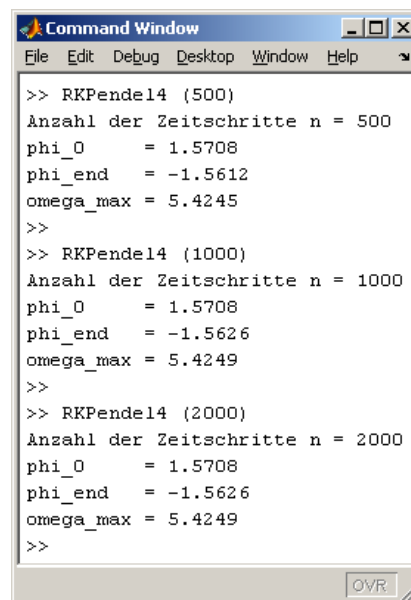
Damit trotzdem die Rechnung auch ohne Parameter gestartet werden kann, wurde die Abfrage in Zeile 5 eingefügt. Wenn die Matlab-Variable *nargin* (**n** Argumente im **Input**) den Wert Null hat (Aufruf ohne Parameter) wird mit dem Standardwert $n = 1000$ für die Anzahl der Integrationsschritte gearbeitet, ansonsten wird die als Parameter übergebene Anzahl übernommen.

Die Function kann also aus dem Command Window zum Beispiel mit

```
RKPendel4 (500)
```

gestartet werden. Nebenstehend sieht man die Ergebnisse von drei Rechnungen mit unterschiedlicher Schrittzahl, die nacheinander aus dem Command Window gestartet wurden.

Trotz aller nachfolgend beschriebenen Möglichkeiten der Verfahrens-Verbesserung, Schrittweitensteuerung und nachträglicher Ergebnis-Verbesserung bleibt dem Praktiker letztendlich doch kaum eine andere Möglichkeit als die Mehrfachrechnung. Die Steuerung (hier: Schrittzahl n) über einen Parameter erspart dann den Eingriff in den Function-Code.



```

Command Window
File Edit Debug Desktop Window Help
>> RKPendel4 (500)
Anzahl der Zeitschritte n = 500
phi_0 = 1.5708
phi_end = -1.5612
omega_max = 5.4245
>>
>> RKPendel4 (1000)
Anzahl der Zeitschritte n = 1000
phi_0 = 1.5708
phi_end = -1.5626
omega_max = 5.4249
>>
>> RKPendel4 (2000)
Anzahl der Zeitschritte n = 2000
phi_0 = 1.5708
phi_end = -1.5626
omega_max = 5.4249
>>
  
```

5.4 Schrittweitenproblem, Richardson-Extrapolation

Bei der numerischen Integration eines Anfangswertproblems ist die Wahl einer geeigneten Schrittweite h (bzw. Δt) ebenso wichtig wie schwierig. Dem Praktiker kann deshalb als effektives Verfahren zur Beurteilung der Qualität einer Rechnung nur empfohlen werden, eine zusätzliche Kontrollrechnung mit halber Schrittweite (und doppelter Anzahl von Integrationschritten) auszuführen. Die Übereinstimmung beider Lösungen (bei einer vertretbaren Toleranz) ist das sicherste Kriterium für die Bestätigung der Schrittweitenwahl.

Diese Aussage gilt, obwohl nachfolgend verschiedene Möglichkeiten der Beurteilung der Schrittweite und der Steuerung der Schrittweite während des Integrationsprozesses beschrieben werden, die allerdings unbedingt verwendet werden sollten, auch wenn die absolute Sicherheit, eine „gesunde“ Rechnung zu erreichen, nicht gewährleistet ist. Hier wird zunächst eine einfache Idee vorgestellt, die in ähnlicher (zum Teil deutlich verbesserter) Form den Strategien zur Schrittweitensteuerung und Ergebnisverbesserung zugrunde liegt.

Das Runge-Kutta-Verfahren 4. Ordnung, dessen Formelsatz 5.6 auf der Seite 16 vorgestellt wurde, erzeugt in jedem Schritt einen Fehler, der bei der Schrittweite h in der Größenordnung h^5 ist, entsprechend dem ersten nicht mehr berücksichtigten Glied der Taylor-Reihe also:

$$\delta y^{(h)} \approx \frac{y^{(5)}(\bar{x})}{5!} h^5 = C_1 h^5 \quad . \quad (5.10)$$

Bei einer Rechnung mit der doppelten Schrittweite $2h$ liegt der Fehler in der Größenordnung $\delta y^{(2h)} \approx C_2 (2h)^5 = 32C_2 h^5$. Unter der Annahme, dass sich die 5. Ableitung von y im betrachteten Bereich nicht nennenswert ändert, darf man $C_1 \approx C_2$ setzen und kommt zu der Aussage, dass der Fehler der „Grobrechnung“ $\delta y^{(2h)}$ (ein Schritt mit der Schrittweite $2h$) das 16-fache des Fehlers einer „Feinrechnung“ (2 Schritte, jeweils mit der Schrittweite h) beträgt.

Das Ergebnis y_{grob} der „Grobrechnung“ ist also mit dem Fehler $\delta y^{(2h)}$ behaftet, das Ergebnis y_{fein} der „Feinrechnung“ mit dem Fehler $\delta y^{(h,h)}$. Der (nicht bekannte) exakte Wert am Ende des Integrationsintervalls y_{exakt} kann mit den beiden numerisch ermittelten Werten und ihren jeweiligen Fehlern also so aufgeschrieben werden:

$$y_{exakt} = y_{fein} + \delta y^{(h,h)} = y_{grob} + \delta y^{(2h)} \approx y_{grob} + 16 \delta y^{(h,h)} \quad . \quad (5.11)$$

Diese Beziehung liefert eine Abschätzung des Fehlers der Feinrechnung:

$$\delta y^{(h,h)} \approx \frac{1}{15} (y_{fein} - y_{grob}) \quad . \quad (5.12)$$

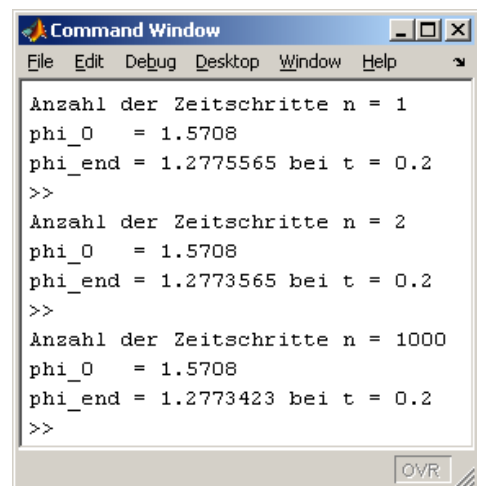
Dieser Wert kann doppelt genutzt werden:

- Man kann eine Grenze für den Fehler vorgeben, bei deren Überschreitung die Schrittweite verkleinert wird (*Schrittweitensteuerung*).
- Wenn für den Fehler der Feinrechnung eine Abschätzung bekannt ist, wird man natürlich den berechneten Wert verbessern (*Ergebniskorrektur*):

$$y_{verbessert} = y_{fein} + \delta y^{(h,h)} \quad . \quad (5.13)$$

Mit dem Matlab-Script RKPendel4.m (siehe Seite 22) kann diese so genannte Richardson-Extrapolation sehr schön demonstriert werden (das Script wurde im Bereich der Ausgabe leicht modifiziert, um die gewünschten Ergebnisse zu sehen).

Es werden zwei Rechnungen ausgeführt, jeweils über den Integrationsbereich $t = 0 \dots 0,2$ s, einmal mit nur einem Integrationsschritt (Grobrechnung) und einmal mit zwei Integrationsschritten. Das nebenstehend zu sehende Command Window zeigt die beiden Werte für den Winkel φ am Ende des Integrationsintervalls. Aus den beiden Ergebnissen berechnet man:



```

Command Window
File Edit Debug Desktop Window Help
Anzahl der Zeitschritte n = 1
phi_0 = 1.5708
phi_end = 1.2775565 bei t = 0.2
>>
Anzahl der Zeitschritte n = 2
phi_0 = 1.5708
phi_end = 1.2773565 bei t = 0.2
>>
Anzahl der Zeitschritte n = 1000
phi_0 = 1.5708
phi_end = 1.2773423 bei t = 0.2
>>
OVR

```

$$\delta \varphi^{(h,h)} \approx \frac{1}{15} (\varphi_{fein} - \varphi_{grob}) = \frac{1}{15} (1,2773565 - 1,2775565) = -0,0000133 \quad .$$

Damit kann das Ergebnis der Feinrechnung korrigiert werden:

$$\varphi_{verbessert} = \varphi_{fein} + \delta \varphi^{(h,h)} = 1,2773565 - 0,0000133 = 1,2773432 \quad .$$

Dass damit eine drastische Verbesserung erzielt wurde, zeigt das Ergebnis der Kontrollrechnung mit 1000 Zeitschriften (im oben zu sehenden Command Window ist dies die letzte

Rechnung). Der ausgewiesene Wert darf als exakt angesehen werden: $\varphi_{end} = 1,2773423$ (die ausgewiesenen Stellen sind tatsächlich alle korrekt):

- Der Fehler der Grobrechnung beträgt 0,0168%, der Fehler der Feinrechnung 0,00112%. Es bestätigt sich die oben gewonnene Erkenntnis, dass der Fehler der Grobrechnung etwa das 16-fache des Fehlers der Feinrechnung ist.
- Die Abweichung des verbesserten Wertes vom exakten Wert mit nur 0,0000705% ist also etwa noch einmal um den gleichen Faktor 16 verkleinert worden.

Die Abweichungen vom exakten Wert erscheinen so minimal, dass der Aufwand einer Ergebniskorrektur übertrieben erscheint, aber man beachte, dass es der Fehler eines einzigen Integrationsschrittes bzw. Doppelschrittes ist, der sich heftig fortpflanzen würde, denn das Ergebnis am Ende eines Integrationsschrittes ist immer der Startwert für den folgenden Schritt. Im behandelten Beispiel müsste man sogar so schlussfolgern: Der für den Wert der Feinrechnung geschätzte Fehler $\delta\varphi^{(h,h)} \approx 0,0000133$ ist zu groß, um mit der gewählten Schrittweite weiter zu rechnen.

Mit der besonders einfachen Strategie der Richardson-Extrapolation konnte sehr schön gezeigt werden, wie man einerseits ein Kriterium gewinnt, um die Schrittweite zu steuern, andererseits damit das Ergebnis der Rechnung noch erheblich verbessern kann. Grundsätzlich braucht man dafür zwei Ergebnisse mit unterschiedlicher Ordnung der Genauigkeit, ermittelt entweder mit einem Verfahren bei unterschiedlicher Schrittweite (wie oben gezeigt) oder mit zwei verschiedenen Verfahren unterschiedlicher Ordnung.

Allerdings ist der Preis dafür ein nicht unerheblicher Mehraufwand. Weil der Aufwand bei der numerischen Integration von Anfangswertproblemen im Wesentlichen von der Anzahl der Funktionswertberechnungen bestimmt wird, wurden Verfahren entwickelt, die einerseits unterschiedlicher Ordnung sind, andererseits aber möglichst viele Funktionswertberechnungen gemeinsam nutzen können. Gegen diese so genannten „Eingebetteten Verfahren“ (siehe Seite 27) ist die hier aus didaktischen Gründen behandelte Richardson-Extrapolation nicht konkurrenzfähig.

6 Einschrittverfahren

Einige Begriffs-Definitionen müssen vorangestellt werden:

- Ein **Integrationsschritt** (gelegentlich auch nur als **Schritt** bezeichnet) ist der Übergang vom Punkt i zum Punkt $i + 1$. In einem **Einschrittverfahren** wird dabei nur auf die bekannten Werte am Punkt i zurückgegriffen (und nicht zusätzlich auf die Werte an weiter zurückliegenden Punkten).

Alle bisher behandelten Verfahren sind Einschrittverfahren.

- Ein Verfahren wird **s-stufig** genannt, wenn für die Berechnung des Näherungswertes am Punkt $i + 1$ genau s Funktionswertberechnungen $f(x_j, y_j)$ erforderlich sind. Das Verfahren von Euler-Cauchy (Seite 4) ist ein 1-stufiges Verfahren (nur eine Funktionswertberechnung am linken Rand des Integrationsintervalls), das klassische Runge-Kutta-

Verfahren 5.6 (Seite 16) ist ein 4-stufiges Verfahren (vier Funktionswertberechnungen für k_1 bis k_4).

- Ein Verfahren wird konsistent von der **Ordnung p** genannt, wenn für einen Schritt über die Schrittweite h der lokale Fehler $\leq O(h^p)$ ist. Das bedeutet eine Übereinstimmung mit den ersten $p + 1$ Gliedern der Taylorreihen-Entwicklung. Das Verfahren von Euler-Cauchy ist von der Ordnung $p = 1$, das klassische Runge-Kutta-Verfahren ist ein Verfahren 4. Ordnung.
- Ein **explizites Verfahren** berechnet die Funktionswerte $f(x_j, y_j)$ nur mit y_j -Werten, die durch vorher berechnete Werte bekannt sind. Das klassische Runge-Kutta-Verfahren ist dafür ein Beispiel, denn jede k_j -Wert-Berechnung verwendet nur einen vorab bereits berechneten anderen k_j -Wert.
- Ein **implizites Verfahren** berechnet die Funktionswerte auch unter Verwendung von Werten, die gerade erst berechnet werden sollen, so dass iterativ vorgegangen werden muss. Das Verfahren von Heun 5.4 (Seite 13) ist ein Beispiel dafür.

Die Ordnung p ist also für die Genauigkeit verantwortlich, die durch den Aufwand erkauft wird, der für ein s -stufiges Verfahren erforderlich ist. Allgemein gilt, dass p nicht größer als s sein kann.

6.1 Runge-Kutta-Familie, Butcher-Array

Das im Abschnitt 5.2 behandelte klassische Runge-Kutta-Verfahren ist ein Sonderfall einer großen Verfahrensfamilie mit einheitlicher Berechnungsvorschrift, alle bezeichnet als

Runge-Kutta-Verfahren:

$$y_{i+1} = y_i + h \sum_{j=1}^s b_j k_j \quad , \quad x_{i+1} = x_i + h \quad (6.1)$$

mit

$$k_j = f(x_j, y_j) \quad , \quad x_j = x_i + h c_j \quad , \quad y_j = y_i + h \sum_{l=1}^s a_{jl} k_l \quad .$$

Es soll zunächst gezeigt werden, wie das klassische Runge-Kutta-Verfahren in dieses allgemeine Schema hineinpasst. Es ist ein 4-stufiges Verfahren, das y_{i+1} wie folgt berechnet:

$$y_{i+1} = y_i + h \sum_{j=1}^4 b_j k_j = y_i + h \left(\frac{1}{6} k_1 + \frac{1}{3} k_2 + \frac{1}{3} k_3 + \frac{1}{6} k_4 \right) \quad .$$

Es gilt also: $s = 4$ und $b_1 = \frac{1}{6}, b_2 = \frac{1}{3}, b_3 = \frac{1}{3}, b_4 = \frac{1}{6}$.

Die c_j -Werte und die a_{jl} -Werte findet man durch Vergleich der Berechnungsvorschrift 5.6 für k_1 bis k_4 von Seite 16 mit den oben angegebenen Formeln 6.1:

$$\begin{aligned} k_1 = f(x_i, y_i) &\Rightarrow c_1 = 0 \quad ; \quad a_{11}, \dots, a_{14} = 0, 0, 0, 0 \quad ; \\ k_2 = f\left(x_i + \frac{h}{2}, y_i + \frac{h}{2} k_1\right) &\Rightarrow c_2 = \frac{1}{2} \quad ; \quad a_{21}, \dots, a_{24} = \frac{1}{2}, 0, 0, 0 \quad ; \\ k_3 = f\left(x_i + \frac{h}{2}, y_i + \frac{h}{2} k_2\right) &\Rightarrow c_3 = \frac{1}{2} \quad ; \quad a_{31}, \dots, a_{34} = 0, \frac{1}{2}, 0, 0 \quad ; \\ k_4 = f(x_i + h, y_i + h k_3) &\Rightarrow c_4 = 1 \quad ; \quad a_{41}, \dots, a_{44} = 0, 0, 1, 0 \quad . \end{aligned}$$

Die c_j, b_j und a_{jl} können in zwei Vektoren und einer Matrix zusammengefasst werden:

$$\mathbf{c} = \begin{bmatrix} 0 \\ \frac{1}{2} \\ \frac{1}{2} \\ 1 \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} \frac{1}{6} \\ \frac{1}{3} \\ \frac{1}{3} \\ \frac{1}{6} \end{bmatrix}, \quad \mathbf{A} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}. \quad (6.2)$$

Die Elemente der Matrix \mathbf{A} sind die Faktoren, mit denen die k_j -Werte zu multiplizieren sind. Bei einem expliziten Verfahren (es wird nur auf vorab berechnete k_j -Werte zugegriffen) sind deshalb nur unterhalb der Hauptdiagonalen von Null verschiedene Elemente.

Die Elemente der beiden Vektoren \mathbf{c} und \mathbf{b} und der Matrix \mathbf{A} werden üblicherweise kompakt im so genannten „Butcher-Array“ zusammengestellt, mit dem das Verfahren eindeutig charakterisiert wird:

<p>Butcher-Array für das klassische Runge-Kutta-Verfahren 4. Ordnung:</p>	$\mathbf{c} \mid \frac{\mathbf{A}}{\mathbf{b}^T} \Rightarrow$	<table style="border-collapse: collapse; margin-left: auto; margin-right: auto;"> <tr><td style="border-right: 1px solid black; padding: 5px;">0</td><td style="padding: 5px;"></td><td style="padding: 5px;"></td><td style="padding: 5px;"></td><td style="padding: 5px;"></td></tr> <tr><td style="border-right: 1px solid black; padding: 5px;">$\frac{1}{2}$</td><td style="border-bottom: 1px solid black; padding: 5px;">$\frac{1}{2}$</td><td style="padding: 5px;"></td><td style="padding: 5px;"></td><td style="padding: 5px;"></td></tr> <tr><td style="border-right: 1px solid black; padding: 5px;">$\frac{1}{2}$</td><td style="border-bottom: 1px solid black; padding: 5px;">0</td><td style="border-bottom: 1px solid black; padding: 5px;">$\frac{1}{2}$</td><td style="padding: 5px;"></td><td style="padding: 5px;"></td></tr> <tr><td style="border-right: 1px solid black; padding: 5px;">1</td><td style="border-bottom: 1px solid black; padding: 5px;">0</td><td style="border-bottom: 1px solid black; padding: 5px;">0</td><td style="border-bottom: 1px solid black; padding: 5px;">1</td><td style="padding: 5px;"></td></tr> <tr><td style="border-right: 1px solid black; padding: 5px;"></td><td style="padding: 5px;">$\frac{1}{6}$</td><td style="padding: 5px;">$\frac{1}{3}$</td><td style="padding: 5px;">$\frac{1}{3}$</td><td style="padding: 5px;">$\frac{1}{6}$</td></tr> </table>	0					$\frac{1}{2}$	$\frac{1}{2}$				$\frac{1}{2}$	0	$\frac{1}{2}$			1	0	0	1			$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{6}$	(6.3)
0																												
$\frac{1}{2}$	$\frac{1}{2}$																											
$\frac{1}{2}$	0	$\frac{1}{2}$																										
1	0	0	1																									
	$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{6}$																								

Die Null-Elemente auf und oberhalb der Hauptdiagonalen der Matrix \mathbf{A} werden im Butcher-Array weggelassen. Es ist leicht zu bestätigen, dass für das Euler-Cauchy-Verfahren 2.3 (Seite 4) das nebenstehend zu sehende besonders einfache Butcher-Array gilt.

$$\begin{array}{c|c} 0 & \\ \hline & 1 \end{array} \quad (6.4)$$

Euler-Cauchy-Verfahren

6.2 Eingebettete Verfahren

Bei der Herleitung von Runge-Kutta-Verfahren kann man zum Beispiel die gewünschte Ordnung p und die Zahl der Stufen $s \geq p$ vorgeben. Für die dann zu erfüllenden Bedingungen stehen mit den Parametern, die mit dem Butcher-Array das Verfahren definieren, im Allgemeinen wesentlich mehr Freiwerte als benötigt zur Verfügung, so dass bei Vorgabe von p und s mehrere Verfahren kreiert werden können.

Während früher (für die Handrechnung) möglichst Formeln mit einfachen Zahlen angestrebt wurden (das klassische Runge-Kutta-Verfahren ist ein typisches Beispiel), spielen für die Computerrechnung andere Kriterien eine Rolle: Neben dem Ziel, für gewünschte Genauigkeit und erforderlichen Aufwand ein möglichst optimales Verhältnis zu finden, ist besonders die Möglichkeit der Fehlerabschätzung und damit der Schrittweitensteuerung das Ziel der Verfahrenssuche gewesen.

Wie im Abschnitt 5.4 demonstriert, benötigt man für dieses Ziel zwei Ergebnisse mit unterschiedlicher Genauigkeit. Diese wurden im Abschnitt 5.4 aus Rechnungen mit verschiedenen

Schrittweiten mit dem klassischen Runge-Kutta-Verfahren gewonnen. Alternativ dazu kann man zwei Rechnungen mit zwei Verfahren unterschiedlicher Ordnung p ausführen. Das Ziel der Suche nach neuen Verfahren bestand im Wesentlichen darin, zwei solche Verfahren zu finden, bei denen möglichst viele der berechneten Funktionswerte $f(x_j, y_j)$ für beide Rechnungen verwendet werden können. Ideal sind Verfahren, die sich nur in den Koeffizienten des Vektors \mathbf{b} unterscheiden, weil dann alle Funktionswertberechnungen $f(x_j, y_j)$ für beide Verfahren verwendet werden können. Verfahren, die diese Bedingung erfüllen, nennt man „Eingebettete Verfahren“.

Der deutsche Mathematiker E. Fehlberg hat ein 6-stufiges Verfahren vorgeschlagen, das diese Bedingung erfüllt und mit zwei unterschiedlichen Sätzen von b_j -Werten eine Lösung 5. Ordnung und eine Lösung 4. Ordnung erzeugt. Dies ist das

Butcher-Array für das Verfahren von Fehlberg:	0						
	$\frac{1}{4}$	$\frac{1}{4}$					
	$\frac{3}{8}$	$\frac{3}{32}$	$\frac{9}{32}$				
	$\frac{12}{13}$	$\frac{1932}{2197}$	$-\frac{7200}{2197}$	$\frac{7296}{2197}$			
	1	$\frac{439}{216}$	-8	$\frac{3680}{513}$	$-\frac{845}{4104}$		
	$\frac{1}{2}$	$-\frac{8}{27}$	2	$-\frac{3544}{2565}$	$\frac{1859}{4104}$	$-\frac{11}{40}$	
	$p = 4$	$\frac{25}{216}$	0	$\frac{1408}{2565}$	$\frac{2197}{4104}$	$-\frac{1}{5}$	0
	$p = 5$	$\frac{16}{135}$	0	$\frac{6656}{12825}$	$\frac{28561}{56430}$	$-\frac{9}{50}$	$\frac{2}{55}$

(6.5)

Mit den b_j -Werten der vorletzten Zeile wird ein Ergebnis der Ordnung $p = 4$ erzeugt, mit den b_j -Werten der letzten Zeile ein Ergebnis der Ordnung $p = 5$.

Die beiden amerikanischen Mathematiker J. R. Dormand und P. J. Prince haben ein 7-stufiges Verfahrens-Paar entwickelt, das wie das Verfahren von Fehlberg eine Lösung der Ordnung $p = 5$ und eine eingebettete Lösung der Ordnung $p = 4$ erzeugt. Obwohl 7-stufig, sind auch beim Dormand-Price-Verfahren nur 6 Funktionswertberechnungen für einen Integrations-schritt erforderlich, weil die letzte Funktionswertberechnung eines Schrittes gleichzeitig als erste für den nächsten Schritt genutzt wird. Die Methode wird definiert durch das

Butcher-Array für das Verfahren von Dormand-Prince:	0							
	$\frac{1}{5}$	$\frac{1}{5}$						
	$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$					
	$\frac{4}{5}$	$\frac{44}{45}$	$-\frac{56}{15}$	$\frac{32}{9}$				
	$\frac{8}{9}$	$\frac{19372}{6561}$	$-\frac{25360}{2187}$	$\frac{64448}{6561}$	$-\frac{212}{729}$			
	1	$\frac{9017}{3168}$	$-\frac{355}{33}$	$\frac{46732}{5247}$	$\frac{49}{176}$	$-\frac{5103}{18656}$		
	1	$\frac{35}{384}$	0	$\frac{500}{1113}$	$\frac{125}{192}$	$-\frac{2187}{6784}$	$\frac{11}{84}$	
	$p = 4$	$\frac{5179}{57600}$	0	$\frac{7571}{16695}$	$\frac{393}{640}$	$-\frac{92097}{339200}$	$\frac{187}{2100}$	$\frac{1}{40}$
	$p = 5$	$\frac{35}{384}$	0	$\frac{500}{1113}$	$\frac{125}{192}$	$-\frac{2187}{6784}$	$\frac{11}{84}$	0

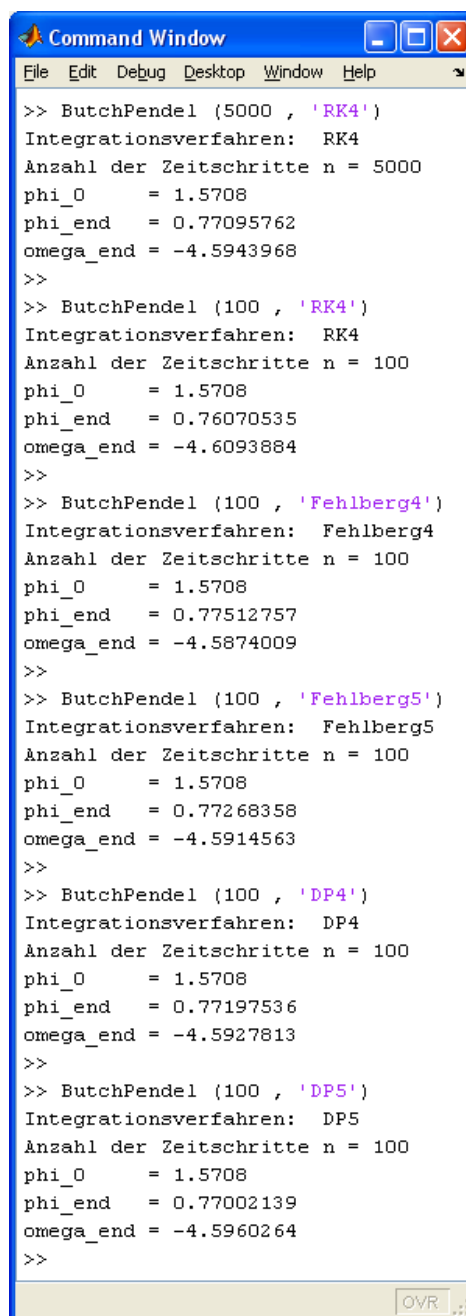
(6.6)

Das Butcher-Array für das Fehlberg-Verfahren 6.5 und das Butcher-Array für das Dormand-Prince-Verfahren 6.6 beschreiben jeweils zwei Integrationsvorschriften. Diese wurden gemeinsam mit dem Verfahren von Euler-Cauchy 6.4 und dem klassischen Runge-Kutta-Verfahren 4. Ordnung 6.3 in einer Matlab-Function realisiert, die als `butch.m` zum Download verfügbar ist. Ihr Aufruf entspricht dem der Function `rk4.m` (siehe Listing auf Seite 22), ergänzt um einen String, der das gewünschte Verfahren kennzeichnet, nach dem integriert werden soll.

Aus dem Matlab-Script *ButchPendel* (Download als `ButchPendel.m`) wird diese Function aufgerufen, um die bereits mehrfach gelöste Aufgabe „Stab-Pendel mit großen Ausschlägen“ (siehe Seite 8) zu berechnen. Nebenstehend sieht man die Ergebnisse einiger Rechnungen im Command Window.

Es wurde wieder der schon mehrfach behandelte Fall mit einer Anfangsauslenkung $\varphi = \pi/2$ über einen Zeitbereich $t = 0 \dots 10$ s berechnet. Die erste Rechnung (nach dem Runge-Kutta-Verfahren 4. Ordnung) mit der außerordentlich großen Schrittanzahl $n = 5000$ kann als Referenzrechnung für die anderen Rechnungen gelten: Die für das Ende des Integrationsintervalls ausgewiesenen Werte φ_{end} und ω_{end} dürfen auch mit der ausgegebenen Stellenanzahl als exakt angesehen werden.

Alle weiteren Rechnungen mit der deutlich geringeren Schrittanzahl $n = 100$ weichen von den Referenzwerten ab, sind aber durchaus noch im akzeptablen Bereich.



```

Command Window
File Edit Debug Desktop Window Help
>> ButchPendel (5000 , 'RK4')
Integrationsverfahren: RK4
Anzahl der Zeitschritte n = 5000
phi_0      = 1.5708
phi_end    = 0.77095762
omega_end  = -4.5943968
>>
>> ButchPendel (100 , 'RK4')
Integrationsverfahren: RK4
Anzahl der Zeitschritte n = 100
phi_0      = 1.5708
phi_end    = 0.76070535
omega_end  = -4.6093884
>>
>> ButchPendel (100 , 'Fehlberg4')
Integrationsverfahren: Fehlberg4
Anzahl der Zeitschritte n = 100
phi_0      = 1.5708
phi_end    = 0.77512757
omega_end  = -4.5874009
>>
>> ButchPendel (100 , 'Fehlberg5')
Integrationsverfahren: Fehlberg5
Anzahl der Zeitschritte n = 100
phi_0      = 1.5708
phi_end    = 0.77268358
omega_end  = -4.5914563
>>
>> ButchPendel (100 , 'DP4')
Integrationsverfahren: DP4
Anzahl der Zeitschritte n = 100
phi_0      = 1.5708
phi_end    = 0.77197536
omega_end  = -4.5927813
>>
>> ButchPendel (100 , 'DP5')
Integrationsverfahren: DP5
Anzahl der Zeitschritte n = 100
phi_0      = 1.5708
phi_end    = 0.77002139
omega_end  = -4.5960264
>>

```

Man sieht, dass die verschiedenen Verfahren zwar unterschiedliche Abweichungen liefern, dass aber kein Verfahren die anderen deutlich überragt, weil nur die einzelnen Verfahren getestet wurden. Die Vorteile, die ein „eingebettetes Verfahrenspaar“ liefert, wurden noch nicht genutzt. Erst die Verbesserung der Ergebnisse in jedem Integrationsschritt, die mit zwei Verfahren unterschiedlicher Ordnung möglich sind, liefert auch bessere Endergebnisse.

Der Standardsolver in Matlab arbeitet mit dem Dormand-Prince-Algorithmus bei Ausnutzung der Verbesserungsmöglichkeiten in jedem Schritt und Steuerung der Schrittweite bei Abweichungen der Ergebnisse der beiden Verfahren, die eine gewisse Grenze überschreiten.

6.3 Dormand-Prince-Verfahren, Standardsolver in Matlab

Matlab bietet eine ganze Reihe von numerischen Integrationsverfahren für Anfangswertprobleme an. Allen gemeinsam ist, dass eine automatische Schrittweitensteuerung eingebaut ist (Matlab-Help, Question and Answers, Frage: „Can I integrate with fixed step sizes?“ Antwort: „No.“) Als Standardverfahren wird der so genannte ode45-Solver („ordinary differential equations 4. and 5. order“) empfohlen (Matlab-Help: „In general, ode45 ist the best function to apply as a first try for most problems.“) Es ist das im vorigen Abschnitt vorgestellte Verfahren von Dormand-Prince 6.6.

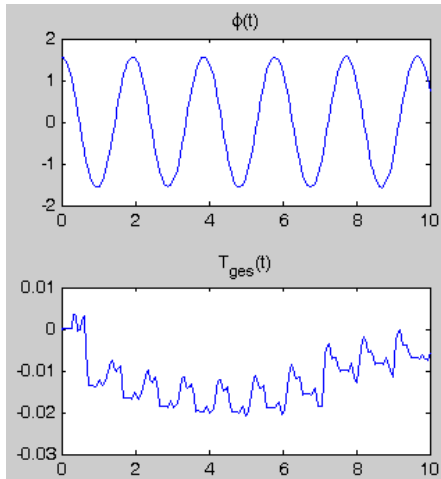
Alle Verfahren arbeiten mit dem gleichen Aufruf (bis auf den Namen der Function natürlich). Darauf wurde in den Beispielen der vorangegangenen Abschnitte bereits hingearbeitet: Der Aufruf des klassischen Runge-Kutta-Verfahrens *rk4* im Beispiel auf Seite 22 kann einfach durch den Aufruf von *ode45* ersetzt werden. Der letzte Parameter im *rk4*-Aufruf (Anzahl der auszuführende Integrationsschritte) muss dabei entweder weggelassen (*ode45* bestimmt die Schrittweite und damit auch die Anzahl der Schritte selbst) oder durch den „options-Parameter“ ersetzt werden, mit dem man verschiedene Wünsche anmelden kann. Das Matlab-Script von Seite 22 lässt sich also wie folgt vereinfachen ([DP45Pendel.m](#)):

```

1 % Stab-Pendel mit großen Ausschlägen, Lösung mit Matlab-Standard-Solver ode45:
2
3 function DP45Pendel
4
5 global pl      g
6     pl = 1 ; g = 9.81 ;           % Pendellaenge, Erdbeschleunigung
7
8 t0 = 0 ; phi0 = pi/2 ; omega0 = 0 ; % Anfangswerte
9 tEnd = 10 ;                               % Ende des Integrationsintervalls
10
11 % Runge-Kutta-Algorithmus
12 [t , xv] = ode45 (@BewDglPendel , [t0 ; tEnd] , [phi0 ; omega0]) ;
13 % Runge-Kutta-Algorithmus
14
15 phi = xv(:,1) ;                               % Sortieren der Ergebnisse:
16 omega = xv(:,2) ;                             % Spalte 1: phi, Spalte 2: omega
17
18 % Erweiterte Auswertung, Verifizieren der Ergebnisse:
19 Tges = - g/2*cos(phi) + pl*omega.^2/6 ; % Kontrollfunktion "Gesamtenergie"
20
21 [phimin , imin] = min(phi) ;
22 disp(['Anzahl der Zeitschritte = ', num2str(length(t))]) ;
23 disp(['phi_0 = ', num2str(phi0)]) ;
24 disp(['phi_end = ', num2str(phi(end) , 8)]) ;
25 disp(['omega_end = ', num2str(omega(end) , 8)]) ;
26 subplot(2,1,1) ; plot(t , phi) , title ('\phi(t)') ;
27 subplot(2,1,2) ; plot(t , Tges) , title ('T_{ges}(t)') ;
28
29 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
30 function xvp = BewDglPendel (t , xv)
31 global pl g           % ... sind in der Function DP45Pendel (oben) definiert.
32 phi = xv(1) ;       % Input-Werte kommen in ...
33 omega = xv(2) ;     % ... einem Vektor an.
34 phip = omega ;     % Die beiden Differenzialgleichungen
35 omegap = -1.5*g/pl*sin(phi) ; % werden ausgewertet, ...
36 xvp = [phip ; omegap] ; % ... die Ergebnisse werden im Vektor xvp abgeliefert.

```

Damit scheint der Anwender vom schwierigsten Problem, der Wahl einer geeigneten Schrittweite, befreit zu sein. Die Ergebnisse sprechen eine andere Sprache. Es wurde wieder der schon mehrfach behandelte Fall mit einer Anfangsauslenkung $\varphi = \pi/2$ über einen Zeitbereich $t = 0 \dots 10$ s berechnet:



```

Command Window
File Edit Debug Desktop Window Help
Anzahl der Zeitschritte n = 189
phi_0      = 1.5708
phi_end    = 0.72884515
omega_end  = -4.6819218
>>
  
```

Standard-Einstellungen liefern inakzeptable Ergebnisse. Erwartet werden am Ende des Integrationsintervalls die Werte $\varphi_{end} = 0,77096$ und $\omega_{end} = -4,5944$.

Die Kontrollfunktion $T_{ges}(t)$ (Gesamtenergie während der Bewegung, vgl. Abschnitt 4) zeigt deutliche Abweichungen vom Sollwert in nicht zu akzeptierender Größenordnung (linkes Bild). Die als Kontrollen ins Command Window geschriebenen Endwerte der berechneten Variablen (rechtes Bild) sind weit von den zu erwartenden Werten entfernt.

Die Ursachen dafür sind weder das verwendete Verfahren (Dormand-Prince) noch die Berechnungsstrategie (automatische Schrittweitensteuerung auf der Basis der Ergebnisse des eingebetteten Verfahrens). Verantwortlich für die schlechten Ergebnisse sind die Werte, die für die Steuerung der Schrittweite eingestellt sind, denn natürlich müssen irgendwo Zahlenwerte für die Beantwortung der Frage vorgesehen sein, ob und wie die Schrittweite geändert werden soll. Es ist müßig, darüber zu streiten, ob die Standard-Einstellungen in Matlab sinnvoll sind oder nicht. Es ist schlicht unmöglich, Standard-Einstellungen zu wählen, die für alle Probleme passend sind. Generell gilt:

Die Aussage der Anbieter kommerzieller Software, dass „leistungsfähige Verfahren“ zur numerischen Integration von Anfangswertproblemen eingesetzt werden, die mit „automatischer Wahl der Schrittweite“ arbeiten, ist in der Regel richtig (trifft z. B. für Matlab in vollem Umfang zu).

Diese Aussage darf allerdings auf keinen Fall dazu verleiten, auf die Ergebnisse ohne genaue Prüfung zu vertrauen. Mindestens muss eine Überprüfung durchgeführt werden, ob mit einer geeigneten Schrittweite gerechnet wurde.

- Aus historischen Gründen sind die Standard-Einstellungen bei kommerzieller Software häufig zu optimistisch: Weil kleine Schrittweiten natürlich größere Rechenzeiten erzeugen, konnten diese noch am Ende des 20. Jahrhunderts zu lästigem Warten auf das Ergebnis führen (kann natürlich bei aufwendigen Problemen auch heute noch passieren). Die Computer sind inzwischen drastisch schneller geworden, die Standard-Einstellungen sind häufig geblieben.

- Der Anwender sollte in jedem Fall die Information abfragen, wie viele Zeitschritte tatsächlich berechnet wurden. Im oben gelisteten Script *DP45Pendel* wird dies mit der Länge des Vektors t der als Ausgabe von *ode45* erzeugten Zeitschritte realisiert. Der im Command Window zu findende Wert $n = 189$ weicht allerdings von der Anzahl der Integrationsschritte ab, weil Matlab für jeden Integrationsschritt (nur bei *ode45*) standardmäßig 3 zusätzliche Zwischenwerte durch Interpolation erzeugt. Tatsächlich wurden also nur $(189 - 1)/4 = 47$ Dormand-Prince-Integrationsschritte ausgeführt, und das ist für das behandelte Problem eindeutig zu wenig.
- Leider können auch die besten Verfahren mit den feinsinnigsten Strategien nicht für richtige Ergebnisse garantieren. Das Problem der Verifizierung der Ergebnisse bleibt beim Anwender, der natürlich wissen muss, an „welchen Schraubchen er drehen kann“, um die Schrittweite auch bei automatischer Schrittweitensteuerung zu beeinflussen.

Es gibt mehrere sinnvolle Möglichkeiten, mit denen die Schrittweite auch bei automatischer Schrittweitensteuerung zu beeinflussen ist. Weil Matlab sie alle anbietet, werden die Matlab-Optionen nachfolgend gelistet:

- Nach jedem Integrationsschritt eines eingebetteten Verfahrenspaares kann der Schrittfehler δy_i für jede Komponente y_i des Lösungsvektors geschätzt werden. Aus dem Vektor der Fehler der einzelnen Komponenten kann eine Vektornorm berechnet und mit einem Toleranzwert verglichen werden, es kann aber auch schärfer kontrolliert werden, indem jede einzelne Komponente mit einem Toleranzwert verglichen wird. Matlab bietet beide Möglichkeiten, wobei die Kontrolle der einzelnen Komponenten die Standardstrategie ist. Die beiden Werte ϵ_{rel} und ϵ_{abs} dienen als Fehlertoleranzen. Es muss am Ende eines jeden Schritts gelten:

$$|\delta y_i| \leq \max(\epsilon_{rel} \cdot |y_i|, \epsilon_{abs}) \quad . \quad (6.7)$$

Wenn diese Bedingung nicht erfüllt ist, wird die Schrittweite verkleinert. Voreingestellt sind in Matlab die Werte $\epsilon_{rel} = 10^{-3}$ bzw. $\epsilon_{abs} = 10^{-6}$.

- „Der 1. Schritt“ kann noch nicht auf einer Fehlerabschätzung basieren. Man kann dem Verfahren gegebenenfalls helfen, hier einen sinnvollen (und nicht zu großen) Wert zu verwenden. Auf die Gesamtrechnung (speziell die insgesamt ausgeführten Integrationschritte) wirkt sich dies im Allgemeinen nur geringfügig aus.
- Man kann die Größe der Integrationsschritte generell begrenzen (kein Integrationsschritt darf größer sein als ein vorzugebender Wert). Dies führt in der Regel zu einer allgemeinen Verbesserung der Ergebnisse, wenn man eine geeignete Grenze gefunden hat. In Matlab ist dafür der so genannte *MaxStep*-Parameter vorgesehen, dessen Standardwert (ein Zehntel des gesamten Integrationsintervalls) in der Regel deutlich zu groß ist, um eine Begrenzung der Schrittweite zu veranlassen.

Man beachte, dass mit den eingestellten Fehlertoleranzen nur der Fehler eines einzigen Integrationsschrittes begrenzt wird. Sie geben (leider) keinerlei Garantie dafür, dass der Fehler der Gesamtrechnung innerhalb dieser Grenzen oder überhaupt innerhalb vernünftiger Grenzen liegt. Dem Praktiker bleibt keine Wahl: Es muss mehrfach gerechnet werden, wobei jeweils „an geeigneten Schraubchen gedreht“ werden muss, um die zu erreichende Genauigkeit zu verbessern, was auf geeignete Weise kontrolliert werden muss.

Empfehlenswert ist das Variieren der Schranke ε_{rel} , weil diese sich an der Größenordnung der errechneten Funktionswerte orientiert. Mit ε_{abs} ist viel schwieriger zu steuern, weil man dafür die Größenordnung der zu berechnenden Funktionswerte in die Überlegungen einbeziehen muss, die ja für die einzelnen Komponenten stark unterschiedlich sein können.

Weniger feinsinnig ist die Steuerung über den *MaxStep*-Parameter, mit dem man im Wesentlichen eine Mindestanzahl von Integrationsschritten erzwingen kann. Es ist allerdings nicht zu bestreiten, dass dies für den Praktiker, der weiß, dass er ohnehin mehrfach rechnen muss, in der Regel die einfachste (und deshalb wohl auch beliebteste) Steuerungsmöglichkeit ist.

Die nachfolgend gelistete Matlab-Funktion *DP45oPendel* ist eine Modifikation des oben gelisteten Scripts, mit dem zwei Optionen für die *ode45*-Rechnung vorgegeben werden:

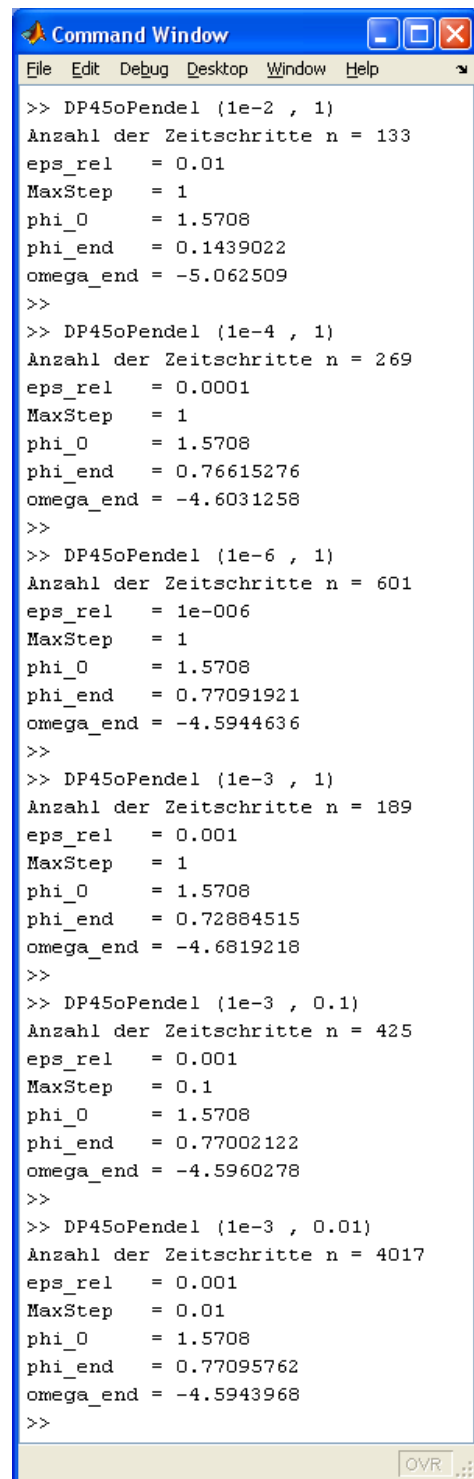
```

1 % Stab-Pendel mit großen Ausschlägen, Lösung mit Matlab-Standard-Solver ode45:
2
3 function DP45oPendel (epstol , mstep)
4
5 if (nargin < 1) reltol = 1e-3 ; maxstep = 1 ;
6 else
7     reltol = epstol ;
8     if (nargin < 2) maxstep = 1 ;
9     else
10        maxstep = mstep ;
11    end
12 end
13
14 global pl      g
15 pl = 1 ; g = 9.81 ;           % Pendellaenge, Erdbeschleunigung
16
17 t0 = 0 ; phi0 = pi/2 ; omega0 = 0 ; % Anfangswerte
18 tEnd = 10 ;                   % Ende des Integrationsintervalls
19
20 % Dormand-Prince-Algorithmus
21 options = odeset ('RelTol' , reltol , 'MaxStep' , maxstep) ;
22 [t , xv] = ode45 (@BewDglPendel , [t0 ; tEnd] , [phi0 ; omega0] , options) ;
23 % Dormand-Prince-Algorithmus
24
25 phi = xv(:,1) ;                % Sortieren der Ergebnisse:
26 omega = xv(:,2) ;             % Spalte 1: phi, Spalte 2: omega
27
28 % Erweiterte Auswertung, Verifizieren der Ergebnisse:
29 Tges = - g/2*cos(phi) + pl*omega.^2/6 ; % Kontrollfunktion "Gesamtenergie"
30
31 [phimin , imin] = min(phi) ;
32 disp ([ 'Anzahl der Zeitschritte_n=' , num2str(length(t))]) ;
33 disp ([ 'eps_rel=' , num2str(reltol)]) ;
34 disp ([ 'MaxStep=' , num2str(maxstep)]) ;
35 disp ([ 'phi_0=' , num2str(phi0)]) ;
36 disp ([ 'phi_end=' , num2str(phi(end) , 8)]) ;
37 disp ([ 'omega_end=' , num2str(omega(end) , 8)]) ;
38 subplot (2,1,1) ; plot (t , phi) , title ('\phi(t)') ;
39 subplot (2,1,2) ; plot (t , Tges) , title ('T_{ges}(t)') ;
40
41 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
42 function xvp = BewDglPendel (t , xv)
43 global pl g
44 phi = xv(1) ; % ... sind in der Funktion RKPendel4 (oben) definiert.
45 omega = xv(2) ; % Input-Werte kommen in ...
46 phip = omega ; % ... einem Vektor an.
47 omegap = -1.5*g/pl*sin(phi) ; % Die beiden Differenzialgleichungen
48 xvp = [phip ; omegap] ; % werden ausgewertet, ...
49 % ... die Ergebnisse werden im Vektor xvp abgeliefert.

```

- Die Optionen für die *ode*-Functions werden in einer Struktur zusammengestellt, die mit der Function *odeset* bestückt wird (Zeile 19). Dieser muss jeweils über ein Schlüsselwort (hier 'RelTol' bzw. 'MaxStep') angekündigt werden, wie der nachfolgende Zahlenwert zu interpretieren ist. Die auf diese Weise gefüllte Struktur (hier *options* genannt) muss der *ode*-Function als 4. Parameter übergeben werden.
- Als Kontrollwerte für die erreichte Genauigkeit dienen (neben der Kontrollfunktion T_{ges}) hier wieder die Endwerte φ_{end} und ω_{end} der berechneten Funktionen (Zeilen 34 und 35). In diesem Fall sind deren korrekte Werte durch die in den vorigen Abschnitten durchgeführten Rechnungen mit sehr feiner Diskretisierung bekannt ($\varphi_{end} = 0,77095762$ und $\omega_{end} = -4,5943968$). Normalerweise muss man natürlich mehrfach rechnen, bis sich diese Werte im Rahmen der gewünschten Genauigkeit nicht mehr ändern.
- *DP45oPendel* ist als Function geschrieben, die mit zwei Parametern aus dem Command Window aufgerufen werden kann (ϵ_{rel} und Wert für MaxStep-Option, Zeile 3). Werden keine Parameter angegeben (oder bei Aufruf aus dem Matlab-Editor), werden die Standardwerte eingestellt (Zeilen 5 bis 10).

Nebstehend sieht man die Ergebnisse von 6 Rechnungen. Bei den ersten drei Rechnungen wird ϵ_{rel} variiert. Man sieht, dass sich bei kleineren Toleranzwerten die Schrittzahl und die Qualität der Ergebnisse erhöhen. Bei einem noch kleineren Toleranzwert als $\epsilon_{rel} = 10^{-6}$ (dritte Rechnung) verbessern sich die Ergebnisse kaum noch, weil dann entsprechend 6.7 (Seite 32) der Toleranzwert ϵ_{abs} greift. Für eine weitere Verbesserung müsste auch dieser noch abgesenkt werden.



```

Command Window
File Edit Debug Desktop Window Help
>> DP45oPendel (1e-2 , 1)
Anzahl der Zeitschritte n = 133
eps_rel = 0.01
MaxStep = 1
phi_0 = 1.5708
phi_end = 0.1439022
omega_end = -5.062509
>>
>> DP45oPendel (1e-4 , 1)
Anzahl der Zeitschritte n = 269
eps_rel = 0.0001
MaxStep = 1
phi_0 = 1.5708
phi_end = 0.76615276
omega_end = -4.6031258
>>
>> DP45oPendel (1e-6 , 1)
Anzahl der Zeitschritte n = 601
eps_rel = 1e-006
MaxStep = 1
phi_0 = 1.5708
phi_end = 0.77091921
omega_end = -4.5944636
>>
>> DP45oPendel (1e-3 , 1)
Anzahl der Zeitschritte n = 189
eps_rel = 0.001
MaxStep = 1
phi_0 = 1.5708
phi_end = 0.72884515
omega_end = -4.6819218
>>
>> DP45oPendel (1e-3 , 0.1)
Anzahl der Zeitschritte n = 425
eps_rel = 0.001
MaxStep = 0.1
phi_0 = 1.5708
phi_end = 0.77002122
omega_end = -4.5960278
>>
>> DP45oPendel (1e-3 , 0.01)
Anzahl der Zeitschritte n = 4017
eps_rel = 0.001
MaxStep = 0.01
phi_0 = 1.5708
phi_end = 0.77095762
omega_end = -4.5943968
>>

```

Bei den letzten drei Rechnungen wird der Wert für die MaxStep-Option variiert. Man sieht, dass man damit sehr schnell zu sehr genauen Ergebnissen kommt, aber der Preis dafür ist hoch: Die letzte Rechnung gibt Ergebnisse für 4017 Zeitpunkte aus (das entspricht 1004 Dormand-Prince-Integrationsschritten).

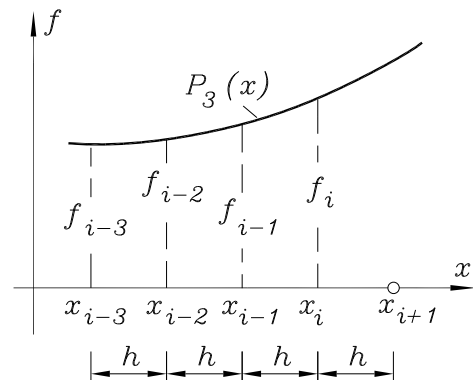
7 Mehrschrittverfahren

Während die *Einschrittverfahren* (Abschnitt 6) den Näherungswert y_{i+1} an der Stelle $x_{i+1} = x_i + h$ ausschließlich auf der Basis des für den Punkt i ermittelten Wert berechnen, beziehen die so genannten *Mehrschrittverfahren* auch weiter zurück liegende Punkte mit ein.

Ausgangspunkt der Überlegung ist (wie bereits beim Euler-Cauchy-Verfahren, siehe Abschnitt 2) die Frage, wie das Integral in

$$y_{i+1} = y_i + \int_{x=x_i}^{x_{i+1}} f(x,y) dx \quad (7.1)$$

(siehe Formel 2) genähert wird. Für alle bereits berechneten Punkte kann $f_i(x_i, y_i)$ berechnet werden. Die Funktion $f(x, y)$ kann damit als Polynom angenähert werden. Die nebenstehende Skizze zeigt dies für das Einbeziehen von vier Punkten x_{i-3} bis x_i , mit denen ein Polynom 3. Grades $P_3(x)$ definiert wird.



Dieses Polynom wird nun als Näherung für $f(x, y)$ auch über den Punkt x_i hinaus angenommen und für die Lösung des Integrals in 7.1 verwendet. Wenn für die Bestimmung des Polynoms nur die Punkte mit bereits bekannten Funktionswerten genutzt werden, spricht man von einem *expliziten Verfahren*, wird auch der unbekannte Funktionswert an der Stelle x_{i+1} einbezogen, ist es ein *implizites Verfahren*.

7.1 Methoden von Adams-Bashforth (explizite Verfahren)

Wenn für die Bestimmung des Polynoms, mit dem die Funktion $f(x, y)$ in 7.1 genähert wird, m bereits bekannte Funktionswerte herangezogen werden, spricht man von einem *expliziten m-Schrittverfahren*. Für das Aufschreiben der Polynome verwendet man zweckmäßig die Strategie der Lagrangeschen Interpolationspolynome. Die Polynome können geschlossen integriert werden.

Dieser Weg, auf dem man zu den Näherungsformeln für einen Integrationsschritt kommt, ist nicht schwierig, aber etwas mühsam. Die sich ergebenden Integrationsformeln sind die

Methoden von Adams und Bashforth,

4-Schrittverfahren:

$$y_{i+1} = y_i + \frac{h}{24} (55 f_i - 59 f_{i-1} + 37 f_{i-2} - 9 f_{i-3}) \quad , \quad (7.2)$$

5-Schrittverfahren:

$$y_{i+1} = y_i + \frac{h}{720} (1901 f_i - 2774 f_{i-1} + 2616 f_{i-2} - 1274 f_{i-3} + 251 f_{i-4}) \quad . \quad (7.3)$$

Es können Formeln für beliebige m -Schrittverfahren hergeleitet werden, die alle erhebliche Vorteile gegenüber den Einschrittverfahren haben, allerdings auch nicht zu übersehende Nachteile:

- Auch wenn ein Integrationsschritt einer Adams-Bashforth-Methode m Funktionswerte $f_i(x_i, y_i)$ einbezieht, muss in jedem Schritt nur ein Funktionswert neu berechnet werden (alle übrigen sind aus den vorangegangenen Schritten bekannt). Deshalb wird hier auch der Begriff des m -Schrittverfahrens verwendet und nicht von einem s -stufigen Verfahren wie bei den Einschrittverfahren (siehe Definition auf Seite 25) geredet. Der Aufwand für die Funktionswertberechnungen $f_i(x_i, y_i)$ (im Allgemeinen der wesentliche Aufwand) ist bei den Mehrschrittverfahren in der Regel deutlich geringer als bei Einschrittverfahren vergleichbarer Fehlerordnung.
- Ein m -Schrittverfahren nach Adams-Bashforth ist von der Fehlerordnung p (siehe Definition auf der Seite 26). Man kann deshalb mit ähnlich minimalen Aufwand wie mit den eingebetteten Einschrittverfahren (Abschnitt 6) zwei Ergebnisse unterschiedlicher Ordnung erzeugen und daraus eine Näherung des Schrittfehlers bestimmen.
- Schrittweitensteuerung (Änderung der Schrittweite durch Auswertung der genäherten Schrittfehler) ist allerdings deutlich schwieriger zu realisieren als bei den Einschrittverfahren, weil dann die Funktionswerte $f_i(x_i, y_i)$ nicht mehr an äquidistanten Punkten vorliegen. Dies ist eine wesentliche Schwäche der Mehrschrittverfahren. Die fehlenden Funktionswerte zurückliegender Punkte müssen gegebenenfalls durch Interpolation berechnet werden.
- Der Start eines Mehrschrittverfahrens erfordert eine Vorleistung: Weil keine zurückliegenden Punkte zur Verfügung stehen, müssen hier in der Regel Einschrittverfahren einspringen. Man kann zum Beispiel mit einem Runge-Kutta-Verfahren einen Satz von Startwerten erzeugen, möglichst mit kleinerer Schrittweite, mindestens sollten die Fehler der Startwerte der Ordnung des verwendeten Mehrschrittverfahrens entsprechen.

7.2 Adams-Moulton- und Adams-Bashforth-Moulton-Verfahren (implizite Verfahren)

Wenn in die Bestimmung des Polynoms, das den Integranden in 7.1 annähern soll, neben den bereits berechneten Funktionswerten $f_i(x_i, y_i)$ auch noch der Wert $f_{i+1}(x_{i+1}, y_{i+1})$ am Punkt x_{i+1} einbezogen wird, in den der noch unbekannte Wert y_{i+1} eingeht, spricht man von einem *impliziten Verfahren*. Die Herleitung der Formeln ist nicht schwieriger als die Herleitung der Adams-Bashforth-Formeln, aber durchaus auch etwas mühsam. Man erhält zum Beispiel bei Einbeziehung der 4 Punkte x_{i-2} bis x_{i+1} die Formel für das so genannte

Adams-Moulton-Verfahren 4. Ordnung:

$$y_{i+1} = y_i + \frac{h}{24} (9 f_{i+1} + 19 f_i - 5 f_{i-1} + f_{i-2}) \quad . \quad (7.4)$$

Hierfür ist (neben der Startrechnung) sogar noch in jedem einzelnen Schritt eine Vorleistung zu erbringen, denn $f_{i+1}(x_{i+1}, y_{i+1})$ ist natürlich nicht ohne zumindest einen auf anderem Wege erzeugten Näherungswert für y_{i+1} zu berechnen. Es ist das gleiche Problem wie

beim Verfahren von Heun (Abschnitt 5.1). Dort wurde die Strategie des *Prädiktor-Korrektor-Verfahrens* ausführlich behandelt.

Es bietet sich natürlich an, als Prädiktor ein (in der Ordnung passendes) Adams-Moulton-Verfahren zu verwenden. Man kommt so zum Beispiel zum

Prädiktor-Korrektor-Verfahren 4. Ordnung nach Adams-Bashforth-Moulton:

$$\begin{aligned} y_{i+1}^{(P)} &= y_i + \frac{h}{24} (55 f_i - 59 f_{i-1} + 37 f_{i-2} - 9 f_{i-3}) \quad (\text{Prädiktor}) , \\ y_{i+1} &= y_i + \frac{h}{24} (9 f_{i+1} + 19 f_i - 5 f_{i-1} + f_{i-2}) \quad . \end{aligned} \quad (7.5)$$

Wie beim Verfahren von Heun, kann man auch beim Adams-Bashforth-Moulton-Verfahren mit mehreren Korrektorschritten arbeiten, wobei jeweils der verbesserte Wert für y_{i+1} verwendet wird.

Auch Adams-Bashforth-Moulton-Formelsätze können für beliebige Ordnung erzeugt werden.

7.3 Verfahren von Gear

Im Gegensatz zu der Strategie, mit der die Formelsätze für das Adams-Bashforth- und das Adams-Moulton-Verfahren hergeleitet werden, dem Ersetzen der Funktion $f(x, y)$ durch ein Polynom auf der Basis der $f_i(x_i, y_i)$ -Werte, werden die Formeln für das Verfahren von Gear auf der Basis eines Polynoms entwickelt, das die Werte $y_i(x_i)$ als Stützstellen hat. Die sich ergebenden Formelsätze haben deshalb einen etwas anderen Charakter: Es geht neben den (bereits bekannten) y_i -Werten nur ein Funktionswert $f_{i+1}(x_{i+1}, y_{i+1})$ ein, dieser allerdings an dem Punkt, für den der Wert y_{i+1} gerade berechnet werden soll.

Verfahren von Gear,

3. Ordnung:

$$y_{i+1} = \frac{1}{11} (6h f_{i+1} + 18y_i - 9y_{i-1} + 2y_{i-2}) \quad , \quad (7.6)$$

4. Ordnung:

$$y_{i+1} = \frac{1}{25} (12h f_{i+1} + 48y_i - 36y_{i-1} + 16y_{i-2} - 3y_{i-3}) \quad . \quad (7.7)$$

Die Gear-Verfahren sind also implizite Mehrschrittverfahren, die eine iterative Lösung in jedem Schritt erfordern, weil im f_{i+1} das gerade zu berechnende y_{i+1} steckt. Dieser Nachteil wird dann in Kauf genommen, wenn die Eigenschaft, dass das Gear-Verfahren auch für die Behandlung so genannter „steifer Differentialgleichungen“ (vgl. Abschnitt 8) stabil arbeitet, genutzt werden soll.

8 Verfahrenswahl

Es gibt unendlich viele Verfahren zur numerischen Integration von Anfangswertproblemen (einfach deshalb, weil sowohl Einschritt- als auch Mehrschrittverfahren beliebiger Ordnung kreiert werden können). Dem Praktiker ist sicher schon die in diesem Skript vorgestellte Menge zu groß.

Aber es ist leider so: Es gibt weder das ideale Verfahren noch eindeutige Empfehlungen, welches Verfahren man für welche Probleme nutzen sollte. Bezeichnend für dieses Dilemma ist das recht große Verfahrensangebot, das Matlab für die numerische Integration von Anfangswertproblemen bereitstellt, andererseits mit Empfehlungen sehr vage bleibt (Beispiel aus Matlab-Help: „When to use ode45? Most of the time.“).

Der Grund für diesen unbefriedigenden Zustand ist nicht etwa die Vernachlässigung der Forschung auf diesem wichtigen und schwierigen Gebiet, im Gegenteil. Die Anzahl der zu diesem Thema veröffentlichten Bücher und Fachartikel ist nicht mehr zu überblicken, die Suche im Internet liefert Riesenmengen an einschlägigen Seiten. Die Untersuchungen der Mathematiker konzentrieren sich auf die beiden wichtigen Fragen: „Welches Verfahren ist *konstistent* (mit welcher Ordnung p), arbeitet *stabil* (in welchem Bereich) und führt damit zu *Konvergenz* (mit welcher Ordnung)?“ und „Für welche Differenzialgleichungen ist welches Verfahren besonders geeignet (bzw. ungeeignet)?“.

Die Ergebnisse der aufwendigen und schwierigen Untersuchungen zur ersten Frage darf der Ingenieur als „ausgewertet und in die Verfahrensauswahl der gängigen Softwareprodukte eingeflossen“ ansehen. Auch die Auswahl der in diesem Skript vorgestellten Verfahren wurde von diesen Erkenntnissen bestimmt.

Die Beantwortung der ebenso schwierigen zweiten Frage führt zwangsläufig auf den Begriff der „steifen Differenzialgleichung“. Und damit ergibt sich das nächste große Dilemma: Einerseits lässt sich ziemlich klar nachweisen, dass einige Lösungsverfahren für steife Differenzialgleichungen ungeeignet sind (und es lassen sich Empfehlungen für besser geeignete Verfahren formulieren), andererseits ist die Beantwortung der Frage, ob eine Differenzialgleichung diese Eigenschaft besitzt oder nicht, ausgesprochen schwierig. Zwei Aussagen, die beliebig durch weitere mit ähnlichem Tenor ergänzt werden können, verdeutlichen das Problem:

H. R. Schwarz schreibt in seinem Klassiker „Numerische Mathematik“ (Verlag B. G. Teubner 1998): „Das Problem der Steifheit existiert ausgeprägt bei nichtlinearen Differenzialgleichungssystemen und wird vermittels einer Linearisierung zu erfassen versucht, indem man das lokale Verhalten der exakten Lösung $y(x)$ in einer kleinen Umgebung von x_i studiert unter der Anfangsbedingung $y(x_i) = y_i$, wo y_i die berechnete Näherungslösung an der Stelle x_i bedeutet.“

M. Eiermann schreibt in seinem Skript „Numerik gewöhnlicher Differenzialgleichungen“ (zu finden unter <http://www.mathe.tu-freiberg.de/eiermann/Vorlesungen/ODE>, sehr zu empfehlen): „Es gibt keine zufriedenstellende Definition der Bauart 'eine Differenzialgleichung heißt steif, wenn ...'“.

Auch wenn das, was Schwarz als „vermittels einer Linearisierung zu erfassen versucht“ beschreibt, durchaus gelingt, letztendlich kommt man zu einem quantitativ anzugebenden Wert für den Begriff „steif“, für den Anwender ist es kaum von Nutzen, zumal in der Bemerkung von Schwarz auch die Aussage steckt, dass der Begriff nicht für eine Differentialgleichung allgemein gelten muss, sondern sich auf bestimmte Bereiche der Lösung beschränken kann.

Die typischen Anfangswertprobleme der Technischen Mechanik sind nichtlineare Anfangswertprobleme. Es bleibt schließlich als Empfehlung für den Praktiker nur das übrig, was enttäuschend klingen mag:

- Man beginne mit einem eingebetteten Einschrittverfahren mit automatischer Schrittweitensteuerung (entspricht auch der Empfehlung der Matlab-Hilfe: *ode45*).
- Es sollte stets (mindestens) eine weitere Rechnung folgen, in der man eine höhere Genauigkeit fordert.
- Erst bei unbefriedigender Konvergenz der Rechnungen mit dem Einschrittverfahren sollte man auf ein Mehrschrittverfahren ausweichen, wobei man wegen der dann zu vermutenden „Steifheit“ ein für diese Eigenschaft geeignetes Verfahren wählen sollte (zum Beispiel das Gear-Verfahren, Abschnitt 7.3).
- Eine Kontrollrechnung mit einem anderen Verfahren ist in jedem Fall sinnvoll. Um möglichst auch eine andere Strategie zu realisieren, kann dies durchaus ein Verfahren mit konstanter Schrittweite sein: Das klassische Runge-Kutta-Verfahren 4. Ordnung ist dafür immer eine gute Empfehlung.

Dringend zu empfehlen ist, mit einer einschlägigen Software zunächst einige Rechnungen mit Problemen zu starten, deren Ergebnisse bekannt sind. Die Praxis im Umgang mit den Verfahren ist durch keine theoretische Untersuchung zu ersetzen. Hilfe in dieser Hinsicht findet man im zweiten Teil dieses Skripts ["Numerische Integration von Anfangswertproblemen - Teil 2: Praxis"](#) und in zahlreichen Beispielen, die man über die Internetseite ["Numerische Integration von Anfangswertproblemen"](#) erreicht.