Exponentielle Rosenbrock-Verfahren

Master-Arbeit

Georg Jansing

Mathematisches Institut
der
Heinrich-Heine-Universität Düsseldorf

Lehrstuhl für Angewandte Mathematik

10.06.2009

Betreuung: Prof. Dr. Marlis Hochbruck

Inhaltsverzeichnis

Kapitel 1
Einleitung

1.1 Motivation und Problemstellung

In dieser Arbeit geht es darum, eine gewöhnliche Differentialgleichung der Form

$$u'(t) = F(t, u(t)), \qquad u(t_0) = u_0 \tag{1.1}$$

numerisch zu lösen. Dabei soll die rechte Seite F sehr steife Komponenten enthalten dürfen. Solche treten etwa bei Raumdiskretisierungen partieller Differentialgleichungen auf. Die verwendete Lösungsverfahrensklasse nennt sich *exponentielle Rosenbrock-Verfahren*. Diese Klasse wird zunächst erklärt und dann eine Konvergenz- und Stabilitätsanalyse durchgeführt. Weiterhin wird eine Implementierung diskutiert. Ziel dieser Arbeit ist es, eine Implementierung in `MATLAB` zu entwickeln, die in Syntax und Verhalten zu den `MATLAB` Standard-Integratoren soweit wie möglich kompatibel ist. Speziell sollen möglichst viele der in der `odeset`-Methode verfügbaren Optionen unterstützt werden. Weiterhin soll die Möglichkeit bestehen, *dense output* zu erzeugen.

1.2 Struktur der Arbeit

Die Arbeit ist in zwei Teile unterteilt. Im ersten Teil geht es um die Herleitung und Analyse exponentieller Rosenbrock-Verfahren. Der zweite diskutiert eine `MATLAB` Implementierung dieser Verfahrensklasse.

Der Anfang des ersten Teils befasst sich zunächst mit exponentiellen Runge-Kutta Verfahren, anhand derer die Idee exponentieller Integratoren erklärt wird. Darauf aufbauend werden exponentielle Rosenbrock-Verfahren hergeleitet. Es wird eine Umformulierung angegeben, die es ermöglicht, dass die spätere Implementierung auch für hochdimensionale Systeme effizient ist.

Das nächste Kapitel behandelt die Fehleranalyse der Verfahrensklasse. Zunächst werden einige Voraussetzungen an die Differentialgleichung gestellt. In den nächsten Abschnitten werden zuerst Defekte und dann Fehler der Verfahren untersucht. Bei der Diskussion der Defekte treten Bedingungen an die Koeffizienren des Verfahrens auf, die sich später als hinreichend für die Ordnung des Verfahrens herausstellen werden. Es folgt eine Stabilitätsanalyse des im Verfahren auftretenden Exponentialoperators. Der letzte Abschnitt enthält das Hauptresultat über Konvergenz und Stabilität exponentieller Rosenbrock-Verfahren.

Das vierte Kapitel bildet den Übergang zwischen dem theoretischen und dem Implementierungsteil der Arbeit. Es stellt einige mathematische Grundlagen für die spätere Implementierung zur Verfügung. Als erstes werden zwei konkrete Verfahren angegeben, die den

Ordnungsbedingungen des vorigen Kapitels - einmal bis Ordnung drei und einmal bis Ordnung vier - genügen. Als nächstes wird die verwendete Schrittweitenkontrolle diskutiert. Folgend wird eine Verallgemeinerung der Verfahrensklasse auf nichtautonome Systeme angegeben, denn zunächst wurden exponentielle Rosenbrock-Verfahren nur für autonome Systeme hergeleitet. Anschließend werden Krylov-Verfahren als Näherungsmethode für die auftretenden Produkte von Matrixfunktionen mit Vektoren vorgestellt und ein Abbruchkriterium für diesen Prozess erklärt. Zuletzt wird die gewählte *dense output* Implementierung erläutert.

Das fünfte Kapitel befasst sich schließlich mit einer konkreten `MATLAB` Implementierung der exponentiellen Rosenbrock-Verfahren. Zunächst wird die Benutzung des Integrators erklärt, im Folgenden das Setzen von Optionen. Der nächste Abschnitt enthält eine Auflistung aller vom Integrator unterstützten Optionen und detaillierte Informationen zu deren Auswirkung auf den Integrationsprozess. Anschließend wird erläutert, wie der Benutzer selbst Funktionen schreiben kann, welche die Produkte von Matrixfunktionen mit Vektoren auswerten. Der letzte Abschnitt behandelt die Struktur des Softwarepackets und was dazu nötig ist, um es zu erweitern.

1.3 Danksagung

An dieser Stelle möchte ich mich sehr herzlich bei allen Personen bedanken, die am Entstehen dieser Arbeit mitgewirkt haben.

Bei Prof. Dr. Marlis Hochbruck für die Vergabe eines sehr interessanten Themas, für viele fruchtbringende Diskussionen und eine hervorragende Betreuung.

Bei Prof. Dr. Florian Jarre, der sich als Zweitgutachter zur Verfügung gestellt hat.

Bei Dr. Julia Schweitzer und PD Dr. Volker Grimm für viele nützliche Tipps und Hinweise und viel konstruktive Kritik.

Bei meiner Freundin und meiner Familie, die mich über die ganze Zeit hinweg unterstützt und gestärkt haben.

Und bei allen Korrekturlesern, die die Qualität dieser Arbeit deutlich verbessern konnten.

Kapitel 2

Exponentielle Rosenbrock-Verfahren

In diesem Kapitel wird die Idee exponentieller Rosenbrock-Verfahren erklärt. Diese ist eine Modifizierung exponentieller Runge-Kutta Verfahren, welche hier auch kurz beschrieben werden.

Weiterhin wird erläutert, wie das Verfahren effizient implementiert werden kann. Hierbei wird speziell auf die Auswertung der auftretenden Matrixfunktionen mithilfe von Krylov Verfahren eingegangen.

2.1 Idee exponentieller Integratoren

In diesem Abschnitt werden exponentielle Integratoren im Allgemeinen motiviert und hergeleitet. Ausgehend von semilinearen Problemen wird mithilfe der Variation-der-Konstanten-Formel der Ursprung des exponentiellen Ansatzes gezeigt. Danach werden vorbereitend auf den nächsten Abschnitt exponentielle Runge-Kutta-Verfahren hergeleitet.

Es sei eine semilineare gewöhnliche Differentialgleichung gegeben, das heißt eine Differentialgleichung der Form

$$u'(t) = Au(t) + g(t, u(t)), \qquad u(t_0) = u_0, \tag{2.1}$$

mit konstanter Matrix A und einem nichtlinearen Anteil g. Die Anwendung der Variation-der-Konstanten Formel liefert die Lösung

$$u(t_n + h) = e^{hA}u(t_n) + h\int_0^1 e^{(1-\tau)hA}g(t_n + h\tau, u(t_n + h\tau))\,d\tau. \tag{2.2}$$

Hier tritt zum ersten Mal die Exponentialfunktion zu Tage, der die ganze Integratorenklasse ihren Namen verdankt. Nun ergibt sich das Problem, das Integral auf der rechten Seite zu berechnen. Da u unbekannt ist, ist dies im Allgemeinen nicht exakt möglich. Das Ziel sind jedoch numerische Methoden, gute Näherungen sind also ausreichend. Hier wird der Ansatz gewählt, g durch ein Interpolationspolynom anzunähern. Hierzu sei die Lagrange-Darstellung

$$g(t_n + h\tau) \approx \sum_{i=1}^{s} g(t_n + c_i h, u(t_n + c_i h))l_i(\tau), \qquad \tau \in [0, 1] \tag{2.3}$$

mit den Lagrange-Polynomen

$$l_i(\tau) = \prod_{j \neq i} \frac{\tau - c_j}{c_i - c_j}$$

an gewissen Interpolationsknoten c_i verteilt auf dem Intervall $[0, 1]$ gewählt. Dabei tritt die Lösung ausgewertet auf den Zeiten $t_n + c_i h$ auf. Da diese nicht bekannt ist, muss man sich mit Näherungen $U_{ni} \approx u(t_n + c_i h)$ begnügen, die – analog zu klassischen Runge-Kutta-Verfahren – als innere Stufen bezeichnet werden. Man definiert weiterhin $G_{ni} := g(t_n + c_i h, U_{ni})$. Setzt man nun das Interpolationspolynom in (2.2) ein, erhält man die Darstellung

$$\begin{aligned}\int_0^1 e^{(1-\tau)hA} g(t_n + h\tau, u(t_n + h\tau))\, d\tau &\approx \int_0^1 e^{(1-\tau)hA} \sum_{i=1}^{s} G_{ni} l_i(\tau)\, d\tau \\ &= \sum_{i=1}^{s} G_{ni} \int_0^1 e^{(1-\tau)hA} l_i(\tau)\, d\tau\end{aligned}$$

für das Integral. Bezeichnet man nun mit $b_i(z) = \int_0^1 e^{(1-\tau)z} l_i(\tau)\, d\tau$ die noch übrig gebliebenen Integrale, ergibt sich insgesamt ein exponentielles Runge-Kutta-Schema

$$\begin{aligned}U_{ni} &= e^{c_i hA} u_n + h \sum_{j=1}^{s} a_{ij}(hA) G_{nj}, \qquad 1 \leq i \leq s \\ u_{n+1} &= e^{hA} u_n + h \sum_{i=1}^{s} b_i(hA) G_{ni}.\end{aligned}$$

Die inneren Stufen ergeben sich dabei in analoger Weise über die Approximation von

$$u(t_n + c_i h) = e^{c_i hA} u(t_n) + h \int_0^{c_i} e^{(c_i - \tau)hA} g(t_n + h\tau, u(t_n + h\tau))\, d\tau,$$

indem auch hier g durch das Interpolationspolynom aus (2.3) genähert wird. Die Koeffizientenfunktionen sind hier $a_{ij}(z) = \int_0^{c_i} e^{(c_i - \tau)z} l_j(\tau)\, d\tau = c_i \int_0^1 e^{(1-\tau)c_i z} l_j(c_i \tau)\, d\tau$.

Die l_i haben Grad kleiner oder gleich $s - 1$ – falls die c_i paarweise verschieden sind, sogar genau Grad $s-1$. Daher lassen sich diese als Linearkombination der skalierten Monome $\frac{x^{k-1}}{(k-1)!}$ für $1 \leq k \leq s$ schreiben. Dadurch sind die Koeffizientenfunktionen b_i Linearkombinationen der ganzen Funktionen

$$\varphi_k(z) = \int_0^1 e^{(1-\tau)z} \frac{\sigma^{k-1}}{(k-1)!}\, d\tau, \qquad 1 \leq k \leq s. \tag{2.4}$$

Die a_{ij} sind entsprechend Linearkombinationen von $c_i \varphi_k(c_i z)$. Die ganzen Funktionen erfül-

len, wie sich elementar nachrechnen lässt, die Rekursionsformel

$$\varphi_k(z) = \frac{\varphi_{k-1}(z) - \varphi_{k-1}(0)}{z}, \qquad \varphi_0(z) = e^z. \tag{2.5}$$

2.2 Exponentielle Rosenbrock-Verfahren

Im Folgenden werden die in dieser Arbeit betrachteten exponentiellen Rosenbrock-Verfahren hergeleitet. Zur Vereinfachung der Notation und einiger Rechnungen, wird die Verfahrenklasse mit konstanter Schrittweite eingeführt. Eine Diskussion mit variabler Schrittweite findet sich in [6]. Im Folgenden sei also immer

$$t_n = t_0 + hn. \tag{2.6}$$

Betrachtet wird eine nicht notwendigerweise semilineare Differentialgleichung

$$u'(t) = F(u(t)), \qquad u(t_0) = u_0 \tag{2.7}$$

in autonomer Form. Nicht-autonome Probleme werden später in Abschnitt 4.3 diskutiert.

Die Idee der exponentiellen Rosenbrock-Verfahren ist es nun, F in jedem Zeitschritt zu linearisieren, um die Differentialgleichung auf eine semilineare Gestalt (2.1) zu bringen. Für festes u_n definiert man dazu

$$J_n := \frac{\partial F}{\partial u}(u_n) \text{ und } g_n(u) := F(u) - J_n u. \tag{2.8}$$

Damit erhält die Differentialgleichung die Form

$$u'(t) = J_n u(t) + g_n(u(t)). \tag{2.9}$$

Würde man an Stelle der Linearisierung in jedem Zeitschritt eine a-priori-Linearisierung benutzen, etwa $J_n = \frac{\partial F}{\partial u}(u_0)$ oder allgemeiner $J_n = A$ für alle n, so erhielte man ein gewöhnliches exponentielles Runge-Kutta-Verfahren.

Man betrachtet nun u_n als Näherung an die exakte Lösung $u(t)$ zur Zeit t_n, wobei $u_0 = u(t_0)$ der (exakte) Anfangswert des gestellten Anfangswertproblems sei. Wendet man auf die neue Form der Differentialgeichung nun ein explizites exponentielles Runge-Kutta-Schema (2.1) an, so erhält man das exponentielle Rosenbrock-Schema

$$U_{ni} = e^{c_i h J_n} u_n + h \sum_{j=1}^{i-1} a_{ij}(hJ_n) g_n(U_{nj}), \qquad 1 \le i \le s \tag{2.10a}$$

$$u_{n+1} = e^{hJ_n} u_n + h \sum_{i=1}^{s} b_i(hJ_n) g_n(U_{ni}). \tag{2.10b}$$

Die Forderung, dass das Runge-Kutta-Verfahren explizit ist, bedeutet, dass für alle i die Summe in der i-ten inneren Stufe bis $i-1$ läuft, die inneren Koeffizienten $a_{ij}(z)$ also konstant Null sind für $j \ge i$. Dies hat zur Folge, dass man U_{ni} ohne die Lösung nicht-linearer Gleichungssysteme explizit aus den vorher schon bekannten U_{nj}, $1 \le j < i$ berechnen kann.

Die Koeffizientenfunktionen b_i seien dabei noch zu bestimmende Linearkombinationen von den ganzen Funktionen (2.4) und die a_{ij} entsprechend von $c_i \varphi_k(c_i z)$.

Es wird folgende vereinfachende Forderung gestellt:

$$\sum_{i=1}^{s} b_i(z) = \varphi_1(z), \qquad \sum_{j=1}^{i-1} a_{ij}(z) = c_i \varphi_1(c_i z), \qquad 1 \le i \le s \tag{2.11}$$

Diese Forderung liefert, wie später zu sehen sein wird, kleine Defekte und die Methode lässt sich dadurch effizient auf nicht-autonome Probleme übertragen. Wegen

$$\sum_{j=1}^{0} a_{1j}(z) = 0 = c_1 \varphi_1(c_1 z)$$

für alle z gilt $c_1 = 0$ und damit $U_{n1} = e^{c_1 h J_n} u_n = e^0 u_n = I u_n = u_n$.

Weiterhin erhalten Verfahren, die die Forderung (2.11) erfüllen, Gleichgewichtspunkte der Differentialgleichung:

Angenommen, es gilt $F(u_0) = 0$. Dann ist die Lösung mit Anfangswert $u(t_0) = u_0$ durch $u(t) \equiv u_0$ für alle t gegeben. Für die numerische Lösung erhält man induktiv:

$$\begin{aligned} U_{ni} &= e^{c_i h J_0} u_0 + h \sum_{j=1}^{i-1} a_{ij}(hJ_0) g_0(U_{nj}) \\ &= e^{c_i h J_0} u_0 + h \sum_{j=1}^{i-1} a_{ij}(hJ_0) g_0(u_0) \qquad \text{(Induktion nach } i\text{)} \\ &= e^{c_i h J_0} u_0 + h c_i \varphi_1(c_i h J_0) g_0(u_0) \qquad \text{wegen (2.11)} \end{aligned}$$

Es gilt $F(u_0) = 0 \Longleftrightarrow g_0(u_0) = -J_0 u_0$. Wegen

$$e^z - z\varphi_1(z) = e^z - z\left(\frac{e^z - 1}{z}\right) = e^z - e^z + z = z \tag{2.12}$$

folgt weiterhin

$$e^{c_i h J_0} u_0 + h c_i \varphi_1(c_i h J_0) g_0(u_0) = u_0.$$

Für die numerische Lösung zum nächsten Zeitschritt ergibt sich analog:

$$\begin{aligned} u_{n+1} &= e^{hJ_0} u_0 + h \sum_{i=1}^{s} b_i(hJ_0) g_0(U_{ni}) \\ &= e^{hJ_0} u_0 + h \sum_{i=1}^{s} b_i(hJ_0) g_0(u_0) \\ &= e^{hJ_0} u_0 + h\varphi_1(hJ_0) g_0(u_0) \\ &= u_0. \end{aligned}$$

Also bleibt die numerische Lösung beim ersten Zeitschritt konstant. Daher bleibt sie dann auch bei allen weiteren Zeitschritten fest auf dem Anfangswert.

2.3 Umformulierung

Das im vorigen Abschnitt hergeleitete Rosenbrock-Verfahren (2.10) enthält mehrere Produkte von Matrixfunktionen mit Vektoren. Es gibt verschiedene Möglichkeiten diese auszurechnen. Zum einen besteht die Möglichkeit, die Matrix zu diagonalisieren und die Matrixfunktionen nur auf der Diagonalmatrix der Eigenwerte zu berechnen. Als Alternative bieten sich hierzu Krylov-Verfahren an, falls die Matrix zum Beispiel sehr groß und dünn besetzt ist – in diesem Fall sind Produkte der Matrix mit Vektoren, die zentrale Operation bei Krylov-Verfahren, billig zu berechnen. Dann werden die Produke mit den Matrixfunktion allerdings nur genähert.

Um diese Art der Berechnung zu optimieren, führe neue Vektoren $D_{ni} := g_n(U_{ni}) - g_n(u_n)$ ein, also gilt $g_n(U_{ni}) = g_n(u_n) + D_{ni}$. Mit diesen erhält man:

$$\begin{aligned} U_{ni} &= e^{c_i h J_n} u_n + h \sum_{j=1}^{i-1} a_{ij}(hJ_n)(g_n(u_n) + D_{nj}) \\ &= e^{c_i h J_n} u_n + h c_i \varphi_1(c_i h J_n) g_n(u_n) + h \sum_{j=1}^{i-1} a_{ij}(hJ_n) D_{nj} \end{aligned}$$

$$= e^{c_i h J_n} u_n + h c_i \varphi_1(c_i h J_n)(F(u_n) - J_n u_n) + h \sum_{j=1}^{i-1} a_{ij}(hJ_n) D_{nj}$$

$$= u_n + c_i h \varphi_1(c_i h J_n) F(u_n) + h \sum_{j=2}^{i-1} a_{ij}(hJ_n) D_{nj},$$

wobei die letzte Gleichheit aus (2.12) folgt. Der erste Summand fällt weg, da $U_{n1} = u_n$. Analog erhält man:

$$u_{n+1} = u_n + h\varphi_1(hJ_n)F(u_n) + h \sum_{i=2}^{s} b_i(hJ_n) D_{ni}$$

Bei hinreichend glatter rechter Seite F und ebenfalls hinreichend glatter Lösung u sind die D_{ni} bei kleinen Schrittweiten in der Größenordnung von h^2: Es ist mit der Taylorformel

$$\begin{aligned}\|D_{ni}\| &= \|g_n(U_{ni}) - g_n(u_n)\| \\ &\leq \|g_n(u_n) - g_n(u_n)\| + \|U_{ni} - u_n\| \left\| \frac{\partial g}{\partial u}(u_n) \right\| + \|U_{ni} - u_n\|^2 \left\| \frac{\partial^2 g}{\partial u^2}(\xi) \right\|\end{aligned}$$

mit einer Zwischenstelle ξ zwischen U_{ni} und u_n. Es ist $\frac{\partial g}{\partial u} = \frac{\partial F}{\partial u}(u_n) - J_n = 0$. Im nächsten Kapitel werden einige Voraussetzungen gestellt werden, die unter anderem garantieren, dass $\frac{\partial^2 g}{\partial u^2}$ gleichmäßig beschränkt und die Lösung Lipschitz-stetig ist. Mit den Konvergenzaussagen des nächsten Kapitels – Lemma 3.4 und Theorem 3.11 – erhält man dann

$$\begin{aligned}\|U_{ni} - u_n\| &\leq \|U_{ni} - u(t_n + c_i h)\| + \|u(t_n + c_i h) - u(t_n)\| + \|u(t_n) - u_n\| \\ &\leq C\|u(t_{n+1}) - u_{n+1}\| + Ch^3 + Cc_i h + \|u(t_n) - u_n\| \\ &\leq Ch^p + Ch^3 + Ch + Ch^p \\ &\leq Ch.\end{aligned}$$

Wenn man Krylov-Verfahren benutzt, um diese Produkte zu nähern, braucht man daher nur kleine Krylov-Raum-Dimensionen bei der Berechnung von $b_i(hJ_n)D_{ni}$ und $a_{ij}(hJ_n)D_{nj}$, um auf dieselbe Genauigkeit zu kommen, wie bei der des Produktes $\varphi_1(hJ_n)F(u_n)$. Man kommt also von $s+1$ Krylov-Raum-Berechnungen mit hoher Dimension – eine je Vektor $F(u_n)$ beziehungsweise $g_n(U_{ni})$ – auf eine mit hoher Dimension, mit dem Vektor $F(u_n)$, und $s-1$ mit deutlich kleinerer Dimension, mit den Vektoren D_{ni}. Mehr Informationen dazu finden sich in den Abschnitten 4.4 und 4.5.

Eine konkrete Implementierung von Rosenbrock-Verfahren der Ordnungen drei und vier wird später in Kapitel 5 vorgestellt. Diese arbeitet mit der hier beschriebenen Umformulierung der Methode und Krylov-Verfahren zur Berechnung der Matrixfunktionen.

Kapitel 3

Fehleranalyse

In diesem Kapitel werden die Fehler der exponentiellen Rosenbrock-Verfahren analysiert. Dazu werden zunächst Voraussetzungen an die rechte Seite der Differentialgleichung und die Lösung gestellt und deren Plausibilität erläutert. Diese sind wesentlich bei der Herleitung der Fehlerschranke.

Im Weiteren werden als erstes Defekte, dann Fehler betrachtet. Im Abschnitt der Analyse ersterer werden Bedingungen erarbeitet, die sich später als Ordnungsbedingungen an das Verfahren herausstellen werden.

Bevor die letztendliche Konvergenzaussage gezeigt werden kann, ist eine Analyse der im Verfahren auftretenden Exponentialoperatoren vonnöten, der sich ein weiterer Abschnitt widmet.

Den Abschluss des Kapitels bildet die Formulierung und der Beweis des Hauptresultats über Konvergenz und Stabilität exponentieller Rosenbrock-Verfaren.

3.1 Analytischer Rahmen

Um zu zeigen, dass das Verfahren in vernünftiger Weise funktioniert, müssen nun dessen Fehler untersucht werden. Dazu werden zunächst einige Voraussetzungen aufgestellt.

Mit

$$J := J(u) := DF(u) = \frac{\partial F}{\partial u}(u) : X \longrightarrow X$$

sei die Fréchet-Ableitung von F in einer Umgebung der exakten Lösung bezeichnet. Dabei handelt es sich um eine Verallgemeinerung der totalen Ableitung auf normierte Räume. Hier ist bereits eine erste Voraussetzung enthalten: Kommt die betrachtete gewöhnliche Differentialgleichung etwa von einer parabolischen partiellen Differentialgleichung, deren Lösung gewissen Randbedingungen unterworfen ist, so sind diese im Raum X kodiert und es wird daher verlangt, dass der Operator J diese Randbedingung erhält.

Vorstellung: Sind an die Differentialgleichung etwa homogene Dirichlet-Randbedingungen gestellt, zu lösen auf einem Gebiet Ω, so wäre $X = H_0^1(\Omega)$.

Die erste Voraussetzung ist funktionalanalytischer Natur und die Erklärung der darin enthaltenen Begriffe würde einen eigenen Abschnitt erfordern. Daher wird in diesem Rahmen darauf verzichtet. Nähere Informationen zu Halbgruppentheorie findet man beispielsweise in [4], [11]. Die Voraussetzung lautet:

Voraussetzung 1. *Der Operator J erzeugt eine gleichmäßig stetige Halbgruppe e^{tJ} auf dem Banach-Raum X.*

Aus der Voraussetzung folgt die für die Analyse wichtige Eigenschaft, dass es Konstanten C und ω gibt, so dass

$$\left\|e^{tJ}\right\|_{X\leftarrow X} \leq Ce^{\omega t} \qquad \forall\, t \geq 0 \tag{3.1}$$

gleichmäßig in einem Streifen entlang der exakten Lösung der Differntialgleichung gilt. Die Konstanten sind also unabhängig vom Argument des Operators, solange es nahe genug bei der exakten Lösung liegt.

Die dabei auftretende Norm $\|\cdot\|_{X\leftarrow X}$ ist die Operatornorm, wobei die Argumente als Operatoren von X nach Xaufgefasst werden. Ist im Weiteren von der Norm $\|\cdot\|$ die Rede, so handelt es sich um die Norm auf X.

Mit Voraussetzung 1 erhält man unter anderem eine Abschätzung der Koeffizientenfunktionen b_i und a_{ij}. Die b_i sind Linearkombinationen der ganzen Funktionen φ_k. Mit (2.4) gilt:

$$b_i(hJ_n) = \sum_k \lambda_k \varphi_k(hJ_n) = \sum_k \lambda_k \int_0^1 e^{(1-\sigma)(hJ_n)} \frac{\sigma^{k-1}}{(k-1)!}\, d\sigma,$$

wobei die λ_k beliebige aber feste Koeffizienten seien. Schätzt man die Operatornormen mit (3.1) ab, erhält man

$$\begin{aligned}\left\|b_i(hJ_n)\right\|_{X\leftarrow X} &\leq \sum_k |\lambda_k| \int_0^1 \left\|e^{(1-\sigma)(hJ_n)}\right\|_{X\leftarrow X} \left|\frac{\sigma^{k-1}}{(k-1)!}\right| d\sigma, \\ &\leq \sum_k |\lambda_k| \int_0^1 e^{(1-\sigma)\omega h} \left|\frac{\sigma^{k-1}}{(k-1)!}\right| d\sigma.\end{aligned}$$

Die Lösung wird immer über ein festes, beschränktes Zeitintervall $[t_0, T]$ gesucht. Wegen $t_0 \leq h \leq T$ ist dann $h \leq C_H$ und $e^h \leq e^{C_H} \leq \widetilde{C}$, eine Schranke, die unabhängig von der Schrittweite ist. Dann gelten:

$$\left\|b_i(hJ_n)\right\|_{X\leftarrow X} \leq C \qquad \text{und} \qquad \left\|a_{ij}(hJ_n)\right\|_{X\leftarrow X} \leq C, \tag{3.2}$$

wobei die Schranke für die a_{ij} auf analoge Weise gezeigt wird.

Für die zweite Voraussetzung wenden wir nun auf F eine a-priori-Linearisierung an. Man schreibt dazu das Anfangswertproblem als

$$F(u) := Au + f(u), \qquad u(t_0) = u_0. \tag{3.3}$$

Damit gilt $J_n = A + \frac{\partial f}{\partial u}(u_n)$ und $g_n(u) = Au + f(u) - Au - \frac{\partial f}{\partial u}(u_n)u = f(u) - \frac{\partial f}{\partial u}(u_n)u$.
Damit lässt sich nun eine weitere Voraussetzung formulieren:

Voraussetzung 2. *$f : X \longrightarrow X$ sei hinreichend oft Fréchet-differenzierbar und die Lösung u habe hinreichend viele Ableitungen in X. Alle auftretenden Ableitungen seien gleichmäßig beschränkt.*

Unter Anderem bedeutet das, dass J die Lipschitz-Bedingung

$$\|J(u) - J(v)\|_{X \leftarrow X} = \left\| \frac{\partial f}{\partial u}(u) - \frac{\partial f}{\partial u}(v) \right\|_{X \leftarrow X} \leq C\|u - v\| \tag{3.4}$$

erfüllt.

3.2 Defekte

Mithilfe der im vorigen Abschnitt aufgestellten Voraussetzungen kann nun damit begonnen werden, Fehler zu analysieren. Zuerst betrachten wir die Defekte. Zunächst führen wir die abkürzenden Formeln

$$G_n(t) := g_n(u(t)),$$

sowie

$$E_{ni} := U_{ni} - u(t_n + c_i h) \text{ und } e_n := u_n - u(t_n) \tag{3.5}$$

für die Fehler in den inneren Stufen beziehungsweise in den vollen Zeitschritten ein. Dann setzen wir die exakte Lösung in das exponentielle Rosenbrock-Schema (2.10) ein

$$u(t_n + c_i h) = e^{c_i h J_n} u(t_n) + h \sum_{j=1}^{i-1} a_{ij}(hJ_n) G_n(t_n + c_j h) + \Delta_{ni}, \qquad 1 \leq i \leq s \tag{3.6a}$$

$$u(t_{n+1}) = e^{hJ_n} u(t_n) + h \sum_{i=1}^{s} b_i(hJ_n) G_n(t_n + c_i h) + \delta_{n+1}, \tag{3.6b}$$

um die Defekte Δ_{ni} und δ_{n+1} zu erhalten.

Im folgenden Lemma leiten wir Schranken für δ_{n+1} und die Δ_{ni} her, die zum einen den Fehler e_n des Verfahrens zur letzten Zeit t_n und zum anderen Terme, die im wesentlichen aus verschiedenen Potenzen der Schrittweite h multipliziert mit verfahrensabhängigen Koeffizienten bestehen, enthalten. Die Defekte werden also klein, wenn diese Koeffizienten bis zu einer gegebenen Ordnung p Null sind. Daraus ergeben sich Bedingungen, die in Tabelle (3.1) zusammengefasst sind. Wie sich später herausstellen wird, sind diese Bedingungen hinreichend für die entsprechende Ordnung des Verfahrens.

p	Ordnungsbedingung		entsprechendes ψ
1	$\sum_{i=1}^{s} b_i(z) = \varphi_1(z)$		$\psi_1(z) \equiv 0$
2	$\sum_{j=1}^{i-1} a_{ij}(z) = c_i\varphi_1(c_i z),$	$2 \le i \le s$	$\psi_{1i}(z) \equiv 0$
3	$\sum_{i=2}^{s} b_i(z)c_i^2 = 2\varphi_3(z)$		$\psi_3(z) \equiv 0$
4	$\sum_{i=2}^{s} b_i(z)c_i^3 = 6\varphi_4(z)$		$\psi_4(z) \equiv 0$

Tabelle 3.1: Ordnungsbedingungen

Lemma 3.1. *Für $1 \le p \le 4$ gilt*

$$\|\Delta_{ni}\| \le Ch^2\|e_n\| + Ch^3, \tag{3.7a}$$

$$\|\delta_{n+1}\| \le Ch^2\|e_n\| + Ch^{p+1}, \tag{3.7b}$$

falls die Bedingungen aus Tabelle (3.1) *für alle $1 \le q \le p$ erfüllt sind.*

Beweis. Ausdrücken von $u(t_{n+1}) = u(t_n + h)$ in (2.9) durch die Variation-der-Konstanten Formel liefert

$$u(t_n + h) = e^{hJ_n}u(t_n) + h\int_0^1 e^{(1-\tau)hJ_n}G_n(t_n + h\tau)\,d\tau$$

und durch Subtrahieren von (3.6b) ergibt sich weiterhin

$$\delta_{n+1} = h\int_0^1 e^{(1-\tau)hJ_n}G_n(t_n + h\tau)\,d\tau - h\sum_{i=1}^{s} b_i(hJ_n)G_n(t_n + c_i h).$$

Taylorentwickeln von G_n um t_n bis zum Grad $p-1$ ergibt:

$$= \underbrace{h\left[\underbrace{\int_0^1 e^{(1-\tau)hJ_n}\,d\tau}_{=\varphi_1(hJ_n)} - \sum_{i=1}^{s} b_i(hJ_n)\right]}_{:=\psi_1(hJ_n)} G_n(t_n)$$

$$
\begin{aligned}
&+ h^2 \underbrace{\left[\underbrace{\int_0^1 e^{(1-\tau)hJ_n}\tau\, d\tau}_{=\varphi_2(hJ_n)} - \sum_{i=1}^{s} b_i(hJ_n)c_i \right]}_{:=\psi_2(hJ_n)} G_n'(t_n) \\
&+ \dots \\
&+ h^p \underbrace{\left[\underbrace{\int_0^1 e^{(1-\tau)hJ_n}\frac{\tau^{p-1}}{(p-1)!}\, d\tau}_{=\varphi_p(hJ_n)} - \sum_{i=1}^{s} b_i(hJ_n)\frac{c_i^{p-1}}{(p-1)!} \right]}_{:=\psi_p(hJ_n)} G_n^{(p)}(t_n) \\
&+ h^{p+1} \underbrace{\left[\int_0^1 e^{(1-\tau)hJ_n}\frac{\tau^p}{p!}\, d\tau G^{(p+1)}(\xi) - \sum_{i=1}^{s} b_i(hJ_n)\frac{c_i^p}{p!}G^{(p+1)}(\xi_i) \right]}_{:=\delta_{n+1}^{[p]}}
\end{aligned}
$$

mit $\xi \in [t_n, t_{n+1}]$, $\xi_i \in [t_n, t_n + c_i h]$, $1 \le i \le s$ und

$$\psi_k(z) := \varphi_k(z) - \sum_{i=2}^{s} b_i(z)\frac{c_i^{k-1}}{(k-1)!},$$

wobei wegen $c_1 = 0$ für $k \ge 2$ der erste Summand entfällt.

Die Bedingungen aus Tabelle (3.1) bedeuten, dass die ψ_i für $i = 1$ und $3 \le i \le p$ verschwinden. Es bleibt also zu zeigen, dass

$$\|h^2\psi_2(hJ_n)G_n'(t_n)\| \le Ch^2\|e_n\|$$

und

$$\left\|\delta_{n+1}^{[p]}\right\| \le Ch^{p+1} \text{ für } p \ge 3.$$

(3.2) zeigt die Beschränktheit der b_i und im Beweis der Aussage wurde auch eine Schranke für die φ_k gezeigt, also ist ganz ψ_2 unter Kontrolle. Es genügt daher $\|G_n'(t_n)\|$ durch $\|e_n\|$

zu beschränken. Dies gelingt durch

$$\begin{aligned} G_n'(t_n) &= (g_n(u))'(t_n) \\ &= \frac{\partial g_n}{\partial u}(u(t_n))u'(t_n) \\ &= \left(\frac{\partial f}{\partial u}(u(t_n)) - \frac{\partial f}{\partial u}(u_n)\right) u'(t_n), \end{aligned}$$

denn somit gilt

$$\begin{aligned} \|G_n'(t_n)\| &\leq \widetilde{C}\|u(t_n) - u_n\|\|u'(t_n)\| \\ &\leq C\|e_n\| \end{aligned}$$

nach der Lipschitz-Bedingung (3.4) und da nach Voraussetzung 2 die Ableitungen der Lösung u gleichmäßig beschränkt sind.

Die Schranke für die $\delta_{n+1}^{[p]}$ folgt aus den Schranken für die ganzen Funktionen φ_k und den Koeffizientenfunktionen und Voraussetzung 2.

Der Beweis für die Δ_{ni} funktioniert analog. Dabei treten die Funktionen

$$\psi_{ji}(z) := \varphi_j(z)c_i^j - \sum_{k=1}^{i-1} a_{ik}(z)\frac{c_k^{j-1}}{(j-1)!}$$

auf, die in der Bedingung für Ordnung zwei in Form von ψ_{1i} vorkommen. □

Bemerkung 3.2. Die vereinfachenden Bedingungen (2.11) entsprechen genau den ersten beiden Ordnungsbedingungen aus Tabelle (3.1). Daher haben Verfahren, die die vereinfachenden Bedingungen erfüllen, automatisch mindestens Ordnung zwei.

3.3 Fehler

Als nächstes müssen die Fehler e_n und E_{ni} untersucht werden. Dieser Abschnitt befasst sich mit letzteren, den Fehlern der inneren Stufen. Um diese zu kontrollieren werden als Hilfsmittel zunächst einige der auftretenden Terme untersucht. Dazu betrachtet man als

erstes die Differenz von (3.6) und dem exponentiellen Rosenbrock-Schema (2.10):

$$E_{ni} = e^{c_i h J_n} e_n + h \sum_{j=1}^{i-1} a_{ij}(hJ_n)\left(g_n(U_{nj}) - G_n(t_n + c_j h)\right) - \Delta_{ni}, \quad 1 \le i \le s \tag{3.8a}$$

$$e_{n+1} = e^{hJ_n} e_n + h \sum_{i=1}^{s} b_i(hJ_n)\left(g_n(U_{ni}) - G_n(t_n + c_i h)\right) - \delta_{n+1}. \tag{3.8b}$$

In dem folgenden Lemma wird nun die Differenz von Auswertungen von g_n und G_n untersucht.

Lemma 3.3. *Unter Voraussetzung 2 gilt:*

$$\|g_n(U_{ni}) - G_n(t_n + c_i h)\| \le C\left(h + \|e_n\| + \|E_{ni}\|\right)\|E_{ni}\|, \tag{3.9}$$

solange die Fehler E_{ni} und e_n in einer hinreichend kleinen Umgebung von 0 bleiben, die Fehler im n-ten Schritt also nicht zu groß werden.

Beweis. Mit Taylorentwicklung von g_n um $u(t_n + c_i h)$ erhält man:

$$\begin{aligned} g_n(U_{ni}) - G_n(t_n + c_i h) = &\underbrace{g_n(u(t_n + c_i h))}_{=G_n(t_n + c_i h)} - G_n(t_n + c_i h) \\ &+ \frac{\partial g_n}{\partial u}(u(t_n + c_i h))(\underbrace{U_{ni} - u(t_n + c_i h)}_{=E_{ni}}) \\ &+ \int_0^1 (1-\tau)\frac{\partial^2 g_n}{\partial u^2}(u(t_n + c_i h) + \tau E_{ni})(E_{ni}, E_{ni})\, d\tau. \end{aligned}$$

Wegen

$$\begin{aligned} &\left\| \frac{\partial^2 g_n}{\partial u^2}(u(t_n + c_i h) + \tau E_{ni})(E_{ni}, E_{ni}) \right\| \\ &\quad \le \left\| \frac{\partial^2 g_n}{\partial u^2}(u(t_n + c_i h) + \tau E_{ni}) \right\| \cdot \|E_{ni}\| \cdot \|E_{ni}\| \\ &\quad \le \; C \cdot \|E_{ni}\|^2 \end{aligned}$$

ist der Integralterm beschränkt. Es muss also nur noch der Term mit der ersten Ableitung von g_n untersucht werden. Entwickelt man diesen um $u(t_n)$, ergibt sich

$$\begin{aligned} \frac{\partial g_n}{\partial u}(u(t_n + c_i h))(E_{ni}) = &\frac{\partial g_n}{\partial u}(u(t_n))(E_{ni}) \\ &+ \frac{\partial^2 g_n}{\partial u^2}(\xi)(u(t_n) - u(t_n + c_i h), E_{ni}), \qquad \xi \in [u(t_n), u(t_n + c_i h)]. \end{aligned}$$

Weiterhin gilt

$$-\int_0^1 \frac{\partial^2 g_n}{\partial u^2}(u(t_n)+\tau e_n)(e_n)\,d\tau = -\left[\frac{\partial g_n}{\partial u}(u(t_n)+\tau e_n))\right|_{\tau=0}^{\tau=1}$$
$$= -\frac{\partial g_n}{\partial u}(u_n) + \frac{\partial g_n}{\partial u}(u(t_n)).$$

Nach Konstruktion ist

$$\frac{\partial g_n}{\partial u}(u_n) = \frac{\partial f}{\partial u}(u_n)\cdot u_n - \frac{\partial f}{\partial u}(u_n)\cdot u_n = 0,$$

also

$$\frac{\partial g_n}{\partial u}(u(t_n)) = -\int_0^1 \frac{\partial^2 g_n}{\partial u^2}(u(t_n)+\tau e_n)(e_n)\,d\tau.$$

Anwenden der Norm auf diese Gleichung ergibt dann $\|\frac{\partial g_n}{\partial u}(u(t_n))\| \le C\|e_n\|$, daher gilt $\|\frac{\partial g_n}{\partial u}(u(t_n))(E_{ni})\| \le C\|e_n\|\|E_{ni}\|$. Als letztes haben wir noch

$$\left\|\frac{\partial^2 g_n}{\partial u^2}(\xi)(u(t_n)-u(t_n+c_ih),E_{ni})\right\| \le \widetilde{C}\|u(t_n)-u(t_n+c_ih)\|\|E_{ni}\|$$
$$\le \overline{C}\|t_n - t_n + c_ih\|\|E_{ni}\|$$
$$\le Ch\|E_{ni}\|,$$

und die Behauptung ist bewiesen. □

Nachdem dieses Hilfsresultat nun zur Verfügung steht, kann damit der Fehler in den inneren Stufen durch die Schrittweite und den Fehler des vorigen vollen Zeitschritts wie folgt beschränkt werden:

Lemma 3.4. *Unter den Voraussetzungen 1 und 2 gilt:*

$$\|E_{ni}\| \le C\|e_n\| + Ch^3, \tag{3.10}$$

solange die Fehler e_n in einer Umgebung von 0 bleiben.

Beweis. Die Aussage folgt induktiv nach (3.8), (3.7a) und Lemma 3.3, denn für $i = 1$ haben wir

$$\|E_{n_1}\| \le C\|e_n\| + \|\Delta_{n1}\|$$
$$\le C\|e_n\| + Ch^2\|e_n\| + Ch^3$$

und für $i > 1$

$$\begin{aligned}\|E_{ni}\| &\leq C\|e_n\| + h\sum_{j=1}^{i-1} C_j\|g_n(U_{nj}) - G_n(t_n + c_j h)\| + \|\Delta_{ni}\| \\ &\leq C\|e_n\| + h\sum_{j=1}^{i-1} C_j\left(h + \|e_n\| + \|E_{nj}\|\right)\|E_{nj}\| + Ch^2\|e_n\| + Ch^3 \\ &\leq C\|e_n\| + h\sum_{j=1}^{i-1} C_j\left(h + \|e_n\| + C\|e_n\| + Ch^3\right)\left(C\|e_n\| + Ch^3\right) + Ch^2\|e_n\| + Ch^3.\end{aligned}$$

Der Beweis ist damit erbracht, denn alle Summanden enthalten mindestens einen Faktor $\|e_n\|$ oder h^3. □

3.4 Stabilitätsresultate

Dieser Abschnitt befasst sich mit dem Verhalten der Exponentialoperatoren des Schemas, zunächst entlang der exakten Lösung, dann entlang der numerischen. Um ersteres Verhalten zu studieren, definieren wir

$$\widehat{J}_n := DF(u(t_n)) = J(u(t_n)) \tag{3.11}$$

analog zu J_n in (2.8) als die Fréchet-Ableitung von F an der exakten Lösung zum Zeitpunkt t_n. Zunächst zeigen wir einige Hilfsresultate.

Lemma 3.5. *Es seien Voraussetzungen 1 und 2 erfüllt. Dann gibt es für jedes $\widetilde{\omega} > \omega$ eine Konstante C_L, die unabhängig von h ist, so dass*

$$\left\|e^{t\widehat{J}_n} - e^{t\widehat{J}_{n-1}}\right\|_{X\leftarrow X} \leq C_L h e^{\widetilde{\omega}t}, \qquad t \geq 0 \tag{3.12}$$

gilt.

Beweis. Betrachte das folgende Anfangswertproblem

$$v'(t) = \widehat{J}_n v(t) = \widehat{J}_{n-1}v(t) + (\widehat{J}_n - \widehat{J}_{n-1})v(t), \qquad v(0) = v_0,$$

$v : \mathbb{R} \longrightarrow X$. Diese Gleichung wird gelöst von $v(t) = e^{t\widehat{J}_n} v_0$, denn $v'(t) = \widehat{J}_n e^{t\widehat{J}_n} v_0 = \widehat{J}_n v(t)$. Anwenden der Variation-der-Konstanten-Formel liefert

$$v(t) = e^{t\widehat{J}_{n-1}} v(0) + t \int_0^1 e^{(1-\tau)t\widehat{J}_{n-1}} (\widehat{J}_n - \widehat{J}_{n-1}) v(t\tau)\, d\tau.$$

Subtrahiert man $e^{t\widehat{J}_{n-1}} v(0)$ auf beiden Seiten und setzt die Lösung ein, so ergibt sich

$$(e^{t\widehat{J}_n} - e^{t\widehat{J}_{n-1}}) v_0 = \int_0^1 t e^{(1-\tau)t\widehat{J}_{n-1}} (\widehat{J}_n - \widehat{J}_{n-1}) e^{t\tau\widehat{J}_n}\, d\tau.$$

Nach (3.4) ist $\left\| \widehat{J}_n - \widehat{J}_{n-1} \right\|_{X \leftarrow X} \leq \widetilde{C} \|u(t_n) - u(t_{n-1})\|$ und nach Voraussetzung 2

$$\|u(t_n) - u(t_{n-1})\| \leq \widehat{C} |t_n - t_{n-1}| = Ch,$$

denn die erste Ableitung von u ist beschränkt, also u selbst Lipschitz-stetig. Dann ergibt sich durch Anwendung der Norm

$$\begin{aligned}
\left\| e^{t\widehat{J}_n} - e^{t\widehat{J}_{n-1}} \right\|_{X \leftarrow X} &\leq \int_0^1 t \left\| e^{(1-\tau)t\widehat{J}_{n-1}} \right\|_{X \leftarrow X} Ch \left\| e^{t\tau\widehat{J}_n} \right\|_{X \leftarrow X} d\tau \\
&\leq tCh \int_0^1 e^{\omega(1-\tau)t} e^{t\tau\omega}\, d\tau \qquad \text{nach (3.1)} \\
&\leq C_L h \int_0^1 e^{t\widetilde{\omega}(1-\tau+\tau)}\, d\tau \\
&= C_L h e^{t\widetilde{\omega}}.
\end{aligned}$$

Für jedes $\epsilon > 0$ gibt es ein $\overline{C}$ mit $t \leq \overline{C} e^{\epsilon t}$ für alle $t \geq 0$. Daher kann für $\widetilde{\omega} := \omega + \epsilon$ die Konstante $C_L = \overline{C} \cdot C$ gewählt werden. □

Das nächste Resultat definiert eine Skala von Normen, die vom Zeitschritt abhängen und gegeneinander abgeschätzt werden können. Dies macht es später möglich, die Entwicklung entlang mehrerer Zeitschritte zu verstehen.

Lemma 3.6. *Unter den Voraussetzungen 1 und 2 wird durch*

$$|||\, x \,|||_n := \sup_{t \geq 0} e^{-\widetilde{\omega}t} \left\| e^{t\widehat{J}_n} x \right\|, \qquad x \in X \tag{3.13}$$

für $n = 0, 1, 2, \ldots$ eine Norm auf X definiert, die äquivalent zur Norm $\|\cdot\|$ ist. Es gilt dabei sogar $\|x\| \leq |||\, x \,|||_n$, also mit Konstante $C = 1$. Weiterhin ist die Abschätzung

$$|||\, x \,|||_n \leq (1 + C_L h)\, |||\, x \,|||_{n-1}\,, \qquad n \geq 1 \tag{3.14}$$

erfüllt.

Beweis. Nach Voraussetzung 1 gilt:

$$\begin{aligned} |||\, x \,|||_n &\leq \sup_{t \geq 0} C e^{-\widetilde{\omega} t} e^{\omega t} \|x\| \\ &= \sup_{t \geq 0} C e^{(\omega - \widetilde{\omega}) t} \|x\| \\ &\leq C \|x\|, \end{aligned}$$

da $\widetilde{\omega} > \omega$ und daher $e^{(\omega - \widetilde{\omega})t}$ monoton fällt. Die Ungleichung

$$\|x\| \leq |||\, x \,|||_n$$

ist offensichtlich erfüllt, denn für $t = 0$ ist $\|x\| = e^{-\widetilde{\omega} t} \left\| e^{t \widehat{J}_n} x \right\|$ und das Supremum genommen über alle Wahlen von t kann dann nicht mehr kleiner werden. Weiterhin gilt für jedes $x \in X$:

$$\begin{aligned} |||\, x \,|||_n &= \sup_{t \geq 0} e^{-\widetilde{\omega} t} \left\| \left(e^{t \widehat{J}_n} - e^{t \widehat{J}_{n-1}} + e^{t \widehat{J}_{n-1}} \right) x \right\| \\ &\leq |||\, x \,|||_{n-1} + \sup_{t \geq 0} e^{-\widetilde{\omega} t} \left\| e^{t \widehat{J}_n} - e^{t \widehat{J}_{n-1}} \right\|_{X \leftarrow X} \|x\| \\ &\leq |||\, x \,|||_{n-1} + \sup_{t \geq 0} e^{-\widetilde{\omega} t} C_L h e^{\widetilde{\omega} t} \|x\| \\ &= |||\, x \,|||_{n-1} + C_L h \|x\| \sup_{t \geq 0} e^{-\widetilde{\omega} t + \widetilde{\omega} t} \\ &\leq (1 + C_L h)\, |||\, x \,|||_{n-1} \end{aligned}$$

nach dem vorigen Lemma und der bereits gezeigten Ungleichung $\|\cdot\| \leq |||\cdot|||_{n-1}$. □

Als nächstes wird die Entwicklung der Exponentialoperatoren über mehrere Schritte entlang der exakten Lösung untersucht. Das folgende Lemma zeigt eine Schranke für die Norm der Hintereinanderausführung von $n - m + 1$ dieser Operatoren.

Lemma 3.7. *Unter den Voraussetzungen 1 und 2 gibt es eine Konstante C, so dass*

$$\left\| e^{h \widehat{J}_n} \cdots e^{h \widehat{J}_m} \right\|_{X \leftarrow X} \leq C e^{\Omega (t_{n+1} - t_m)}, \qquad 0 \leq m \leq n, \Omega := C_L + \widetilde{\omega}. \tag{3.15}$$

Beweis. Es ist

$$\left\| e^{h\widehat{J}_n} \cdots e^{h\widehat{J}_0} \right\|_{X \leftarrow X} = \sup_{x \in X} \frac{\left\| e^{h\widehat{J}_n} \cdots e^{h\widehat{J}_0} x \right\|}{\|x\|}$$

nach Definition der Operatornorm. Mithilfe von Lemma 3.6 lässt sich das Resultat nun wie folgt zeigen:

$$\begin{aligned}
\left\| e^{h\widehat{J}_n} \cdots e^{h\widehat{J}_0} x \right\| &\leq \left\| \left| e^{h\widehat{J}_n} \cdots e^{h\widehat{J}_m} x \right| \right\|_n \qquad \text{nach Lemma 3.6} \\
&= \sup_{t \geq 0} e^{-\widetilde{\omega} t} \left\| e^{t\widehat{J}_n} e^{h\widehat{J}_n} \cdots e^{h\widehat{J}_m} x \right\| \qquad \text{nach (3.13)} \\
&= \sup_{t \geq 0} e^{-\widetilde{\omega} t} \left\| e^{(t+h)\widehat{J}_n} e^{h\widehat{J}_{n-1}} \cdots e^{h\widehat{J}_m} x \right\| \\
&\leq \sup_{t \geq 0} e^{-\widetilde{\omega} t} \underbrace{\left\| e^{h\widehat{J}_n} \right\|_{X \leftarrow X}}_{\leq C e^{\omega h}} \left\| e^{t\widehat{J}_n} e^{h\widehat{J}_{n-1}} \cdots e^{h\widehat{J}_m} x \right\| \\
&\leq e^{\widetilde{\omega} h} \left\| \left| e^{h\widehat{J}_{n-1}} \cdots e^{h\widehat{J}_m} x \right| \right\|_n \qquad (3.16) \\
&\leq e^{\widetilde{\omega} h} (1 + C_L h) \left\| \left| e^{h\widehat{J}_{n-1}} \cdots e^{h\widehat{J}_m} x \right| \right\|_{n-1} \qquad \text{nach (3.14)} \\
&\leq \cdots \leq e^{\widetilde{\omega} h (n-m)} (1 + C_L h)^{n-m} \left\| \left| e^{h\widehat{J}_m} x \right| \right\|_m \qquad \text{Induktion.}
\end{aligned}$$

Ist $m > 0$, so geht die Ungleichungskette weiter mit

$$\begin{aligned}
&\leq e^{\widetilde{\omega} h (n-m+1)} (1 + C_L h)^{n-m+1} \|x\|_{m-1} \\
&\leq C e^{\widetilde{\omega} h (n-m+1)} (1 + C_L h)^{n-m+1} \|x\| \qquad \text{nach Lemma 3.6} \\
&\leq C e^{(\widetilde{\omega} + C_L) h (n+1)} \|x\| \qquad \text{da } 1 + C_L h \leq e^{C_L h} \\
&= C e^{\Omega (t_{n+1} - t_0)} \|x\|.
\end{aligned}$$

Im Fall $m = 0$, muss man an der Stelle, an der oben unterbrochen wurde, etwas anders argumentieren, denn eine $\left\| \left| \cdot \right| \right\|_{-1}$-Norm gibt es nicht. Hier geht es weiter mit

$$\begin{aligned}
&\leq C e^{\widetilde{\omega} h n} (1 + C_L h)^n \left\| e^{h\widehat{J}_0} x \right\| \qquad \text{nach Lemma 3.6} \\
&\leq C e^{\widetilde{\omega} h n} (1 + C_L h)^n (1 + C_L h) \underbrace{\left\| e^{h\widehat{J}_0} \right\|_{X \leftarrow X}}_{\leq C e^{\omega h} \leq e^{\widetilde{\omega} h}} \|x\| \qquad (3.17) \\
&\leq C e^{(\widetilde{\omega} + C_L) h (n+1)} \|x\| \qquad \text{da } 1 + C_L h \leq e^{C_L h} \\
&= C e^{\Omega (t_{n+1} - t_0)} \|x\|.
\end{aligned}$$

Dabei muss in (3.16) $\widetilde{\omega}$ so gewählt werden, dass $Ce^{\omega h} \leq e^{\widetilde{\omega} h}$ gilt. Mit $c := \log(C)$ erhält man $e^c e^{\omega h} = e^{c+\omega h} = e^{(c/h+\omega)h} \leq e^{\widetilde{\omega} h}$, also $\widetilde{\omega} \geq \frac{c}{h} + \omega$.

In (3.17) wurde ein Faktor $(1 + C_L h)$ eingefügt. C_L besteht aus den beiden Konstanten $\overline{C}$ aus dem Beweis von Lemma 3.5 und C aus Voraussetzung 1, die beide positiv sein müssen. Daher gilt $C_L \geq 0$ und folglich $(1 + C_L h) \geq 1$.

$1+C_L h \leq e^{C_L h}$ erkennt man dadurch, dass man die zweite Ableitung von $1+x-e^x$ betrachtet. Sie ist $-e^x$ und daher überall echt negativ, die Funktion selbst also streng konkav. Daher hat sie ein eindeutiges Maximum, welches bei 0 liegt und den Wert 0 hat. Somit ist sie überall nicht-positiv und die Ungleichung gezeigt. □

Damit ist nun das Verhalten der exakten Lösung hinreichend gut verstanden, um die benötigten Ergebnisse für die numerische Lösung zu erhalten. Betrachten wir dazu die Differenz von zwei Zeitschritten der numerischen Lösung:

$$\begin{aligned} \|u_n - u_{n-1}\| &= \|(u_n - u(t_n)) - (u_{n-1} - u(t_{n-1})) + (u(t_n) - u(t_{n-1}))\| \\ &\leq \|e_n\| + \|e_{n-1}\| + C \underbrace{|t_n - t_{n-1}|}_{=h} \end{aligned} \tag{3.18}$$

Mit dieser Abschätzung kann nun der Exponentialoperator entlang der numerischen Lösung untersucht werden. Das erste Resultat auf dem Weg dahin ist ein Analogon zu Lemma 3.5.

Lemma 3.8. *Es seien Voraussetzungen 1 und 2 erfüllt. Dann gibt es für jedes $\widetilde{\omega} > \omega$ eine Konstante C_L unabhängig von h, so dass*

$$\left\| e^{tJ_n} - e^{tJ_{n-1}} \right\|_{X \leftarrow X} \leq C_L(h + \|e_n\| + \|e_{n-1}\|) e^{\widetilde{\omega} t}, \qquad t \geq 0, \tag{3.19}$$

solange die numerische Lösung in einer hinreichend kleinen Umgebung der exakten Lösung bleibt.

Auch der Beweis läuft analog zu dem von Lemma 3.5 ab. Der einzige Unterschied ist an der Stelle, an der die Lipschitz-Stetigkeit von u ausgenutzt wird. Da hier die numerische anstelle der exakten Lösung auftritt, kommen hier die beiden zusätzlichen Summanden $\|e_n\|$ und $\|e_{n-1}\|$ aus (3.18) ins Spiel.

Auch die zeitschrittabhängigen Normen aus Lemma 3.6 werden benötigt. Diesmal werden sie mit J_n anstelle von $\widehat{J}_n$ definiert.

Lemma 3.9. *Unter den Voraussetzungen 1 und 2 wird durch*

$$|\!|\!| x |\!|\!|_n := \sup_{t \geq 0} e^{-\widetilde{\omega} t} \left\| e^{tJ_n} x \right\|, \qquad x \in X \tag{3.20}$$

für $n = 0, 1, 2, \ldots$ eine Norm auf X definiert, die äquivalent zur Norm $\|\cdot\|$ ist. Es gilt dabei sogar $\|x\| \leq |||x|||_n$, also mit Konstante $C = 1$. Weiterhin erfüllt sie die Abschätzung

$$|||x|||_n \leq (1 + C_L(h + \|e_n\| + \|e_{n-1}\|))\, |||x|||_{n-1}, \qquad n \geq 1. \tag{3.21}$$

Dies gilt, solange die numerische Lösung in einer hinreichend kleinen Umgebung der exakten Lösung bleibt.

Der Beweis der Aussage stimmt fast mit der seines Analogons, Lemma 3.6 überein. Die Äquivalenz der $|||\cdot|||_n$- mit der $\|\cdot\|$-Norm lässt sich komplett übernehmen. Die Abschätzung von $|||\cdot|||_n$ gegen $|||\cdot|||_{n-1}$ benutzt allerdings Lemma 3.8 anstelle von Lemma 3.5 und enthält daher die beiden Fehlernormen als zusätzliche Summanden.

Damit sind alle benötigten Hilfsmittel zusammengetragen, die für das nun folgende Stabilitätsresultat benötigt werden. Es ist ein Analogon zu Lemma 3.7, übertragen auf die numerische Lösung.

Theorem 3.10. *Unter den Voraussetzungen 1 und 2 gibt es Konstanten C und C_E, so dass*

$$\left\|e^{hJ_n} \cdots e^{hJ_m}\right\|_{X \leftarrow X} \leq C e^{\Omega(t_{n+1}-t_m)+C_E \sum_{j=m}^{n} \|e_j\|}, \qquad 0 \leq m \leq n, \Omega := C_L + \widetilde{\omega}, \tag{3.22}$$

solange die numerische Lösung in einer hinreichend kleinen Umgebung der exakten Lösung bleibt.

Beweis. Es werden nur die Schritte angegeben, die aus dem Beweis von Lemma 3.7 angepasst werden müssen. Wie dort, genügt es auch hier zu zeigen, dass die Aussage auf Elementen von X richtig ist und auf der linken Seite mit der $|||\cdot|||_n$-Norm begonnen werden kann:

$$\left|\left|\left| e^{hJ_n} \cdots e^{hJ_m} x \right|\right|\right|_n \leq C e^{\Omega(t_{n+1}-t_m)+C_E \sum_{j=m}^{n} \|e_j\|} \|x\|.$$

Weiter analog vorgehend erhält man

$$\begin{aligned}
\left|\left|\left| e^{hJ_n} \cdots e^{hJ_m} x \right|\right|\right|_n &\leq e^{\widetilde{\omega}h} \left|\left|\left| e^{hJ_{n-1}} \cdots e^{hJ_m} x \right|\right|\right|_n \\
&\leq e^{\widetilde{\omega}h}(1 + C_L(h + \|e_n\| + \|e_{n-1}\|)) \left|\left|\left| e^{hJ_{n-1}} \cdots e^{hJ_m} x \right|\right|\right|_{n-1} \quad \text{L. 3.9} \\
&\leq e^{2\widetilde{\omega}h}(1 + C_L(h + \|e_n\| + \|e_{n-1}\|)) \\
&\qquad (1 + C_L(h + \|e_{n-1}\| + \|e_{n-2}\|)) \left|\left|\left| e^{hJ_{n-1}} \cdots e^{hJ_m} x \right|\right|\right|_{n-2} \\
&\leq e^{(n-m)\widetilde{\omega}h} \prod_{j=m+1}^{n} \underbrace{(1 + C_L(h + \|e_j\| + \|e_{j-1}\|))}_{\leq e^{C_L(h+\|e_j\|+\|e_{j-1}\|)}} \left|\left|\left| e^{hJ_m} x \right|\right|\right|_m \quad \text{Induktion.}
\end{aligned}$$

Auch hier muss man wieder die Fälle $m = 0$ und $m > 0$ unterscheiden. Die Methode dies aufzulösen ist dieselbe wie bereits gesehen und der Schritt wird daher weggelassen. In beiden Fällen kommt man mit $\widetilde{m} = m$, falls $m > 0$ und $\widetilde{m} = 1$, falls $m = 0$ zu der Abschätzung

$$\begin{aligned}
\left|\!\left|\!\left| e^{hJ_n} \cdots e^{hJ_m} x \right|\!\right|\!\right|_n &\leq C e^{(n-m+1)\widetilde{\omega}h + \sum_{j=\widetilde{m}}^{n} (C_L(h+\|e_j\|+\|e_{j-1}\|))+C_L} \|x\| \qquad \text{da } C_L > 0 \\
&= C e^{(\widetilde{\omega}+C_L)h(n-m+1)+\sum_{j=\widetilde{m}}^{n}(C_L(\|e_j\|+\|e_{j-1}\|))} \|x\| \\
&\leq C e^{(\widetilde{\omega}+C_L)h(n-m+1)+2C_L\sum_{j=m}^{n}\|e_j\|} \|x\|.
\end{aligned}$$

Der Beweis ist dann mit $C_E := 2C_L$ erbracht. □

3.5 Fehlerakkumulation

Mit der Stabilitätsanalyse aus dem letzten Abschnitt stehen nun die wesentlichen erforderlichen Werkzeuge zur Verfügung, um das Hauptresultat über die Konvergenz und Stabilität von exponentiellen Rosenbrock-Verfahren beweisen zu können. Es fehlt nur noch eine allgemeine Aussage über Schranken für gewisse endliche Summen. Dieses sogenannte diskrete Lemma von Gronwall wird häufig beim Aufsummieren von Fehlern in Konvergenzbeweisen benutzt und wird daher hier nur angegeben und nicht bewiesen.

Der Rest dieses Abschnitts wird sich dann detailliert mit dem Beweis des Hauptresultats beschäftigen. Dieses lautet wie folgt:

Theorem 3.11. *Für das Anfangswertproblem* (3.3) *seien Voraussetzungen 1 und 2 erfüllt. Es werde darauf ein exponentielles Rosenbrock-Schema* (2.10) *angewendet. Für* $2 \leq p \leq 4$ *seien die Ordnungsbedingungen aus Tabelle* (3.1) *bis zum Grad p erfüllt. Dann konvergiert die numerische Methode mit Ordnung p. Das bedeutet: Es gibt eine Konstante* C*, so dass*

$$\|u_n - u(t_n)\| \leq Ch^p \tag{3.23}$$

gleichmäßig für $t_0 \leq t_n \leq T$ *gilt und* h *hinreichend klein.*

Der Beweis folgt nach Angabe des diskreten Lemmas von Gronwall, welches, wie oben angekündigt, den Werkzeugkasten für den Beweis der Fehleranalyse vervollständigt.

Lemma 3.12 (Diskretes Lemma von Gronwall)**.** *Seien* $\eta_0, \ldots, \eta_{n-1} > 0$*,* $\alpha, \beta \in \mathbb{R}$ *und* $\epsilon_0, \ldots \epsilon_n \geq 0$*. Es gelte*

$$\epsilon_0 \leq \alpha \qquad \text{und} \qquad \epsilon_k \leq \alpha + \beta \sum_{m=0}^{k-1} \eta_m \epsilon_m, \quad 1 \leq k \leq n,$$

dann gilt

$$\epsilon_k \leq \alpha e^{\beta \sum_{m=0}^{k-1} \eta_m}, \qquad \text{für alle } k \in \{0, 1, \ldots, n\}.$$

Kommen wir nun zum

Beweis des Theorems. Aus (3.8b) erhalten wir die folgende Rekursionsdarstellung der Fehler:

$$e_{n+1} = e^{hJ_n} e_n + h\varrho_n - \delta_{n+1},$$

wobei ϱ_n eine Abkürzung für

$$\varrho_n := \sum_{i=1}^{s} b_i(hJ_n)(g_n(U_{ni}) - G_n(t_n + c_i h))$$

ist. Da das Verfahren auf der exakten Lösung startet, ist der Startfehler $e_0 = 0$. Man kann die Lösung dieser Rekursionsgleichung direkt angeben, denn mit

$$e_n = h \sum_{m=0}^{n-1} e^{hJ_{n-1}} \cdots e^{hJ_{m+1}} (\varrho_m - h^{-1}\delta_{m+1}) \tag{3.24}$$

erhält man

$$e_1 = h(\varrho_0 - h^{-1}\delta_1) = he^{hJ_0} e_0 + h\varrho_0 - \delta_1$$

und für $n > 1$ mit Induktion

$$\begin{aligned} e_n &= h \sum_{m=0}^{n-1} e^{hJ_{n-1}} \cdots e^{hJ_{m+1}} (\varrho_m - h^{-1}\delta_{m+1}) \\ &= e^{hJ_{n-1}} h \sum_{m=0}^{n-2} e^{hJ_{n-2}} \cdots e^{hJ_{m+1}} (\varrho_m - h^{-1}\delta_{m+1}) + h(\varrho_{n-1} - h^{-1}\delta_n) \\ &= e^{hJ_{n-1}} e_{n-1} + h\varrho_{n-1} - \delta_n. \end{aligned}$$

Zunächst wird der hintere Differenzterm dieser Lösung mit den beiden Abschnitten über Defekte und Fehler untersucht. Mit (3.2) und Lemma 3.3 erhalten wir

$$\|\varrho_m\| \leq C \sum_{i=1}^{s} (h + \|e_m\| + \|E_{mi}\|)\|E_{mi}\|.$$

Lemma 3.4 zeigt

$$\|E_{mi}\| \leq C\|e_m\| + Ch^3.$$

Damit erhält man

$$\begin{aligned}\|\varrho_m\| &\leq C\sum_{i=1}^{s}(h + \|e_m\| + \|E_{mi}\|)\|E_{mi}\| \\ &\leq C\sum_{i=1}^{s}(h + \|e_m\| + C\|e_m\| + Ch^3)(C\|e_m\| + Ch^3) \\ &\leq C(h\|e_m\| + \|e_m\|^2 + h^p),\end{aligned}$$

Lemma 3.1 sagt aus, dass

$$\|\delta_{m+1}\| \leq Ch^2\|e_m\| + Ch^{p+1},$$

wir erhalten also insgesamt die Ungleichung

$$\|\varrho_m\| + h^{-1}\|\delta_{m+1}\| \leq C(h\|e_m\| + \|e_m\|^2 + h^p).$$

Nun wird der Abschnitt über die Stabilität des Exponentialoperators herangezogen, um die Produkte der e^{hJ_k} in der Lösung zu beschränken. Mit den vorigen Überlegungen und Theorem 3.10 erhalte

$$\|e_n\| \leq Ch\sum_{m=0}^{n-1} e^{\Omega(t_n - t_{m+1}) + C_E\sum_{j=m+1}^{n-1}\|e_j\|}(h\|e_m\| + \|e_m\|^2 + h^p), \tag{3.25}$$

solange die Fehler in einer hinreichend kleinen Umgebung von 0 bleiben.

Zwischenbehauptung. Es gilt

$$\|e_n\| \leq Mh,$$

mit einer Konstante M gleichmäßig in $t_0 \leq t_n \leq T$, das numerische Verfahren hat also Ordnung eins. Der

Beweis der Zwischenbehauptung läuft induktiv. Für $0 = \|e_0\| \leq Mh$ ist die Aussage klar. Im Induktionsschritt sei die Aussage für alle Fehler einschließlich des $n-1$-ten Schrittes

gezeigt. Es gilt für den darauffolgenden Schritt nach (3.25):

$$\begin{aligned}
\|e_n\| &\leq Ch \sum_{m=0}^{n-1} e^{\Omega(t_n - t_{m+1}) + C_E \sum_{j=m+1}^{n-1} \|e_j\|} (h\|e_m\| + \|e_m\|^2 + h^p) \\
&\leq Ch \sum_{m=0}^{n-1} e^{\Omega(t_n - t_{m+1}) + C_E h(n-1-m)M} (h^2 M + h^2 M^2 + h^p) \qquad \text{I.A.} \\
&\leq Ch \sum_{m=0}^{n-1} e^{(\Omega + C_E M)(T - t_0)} h^2 (M + M^2 + 1) \qquad p \geq 2 \\
&\leq Chne^{(\Omega + C_E M)(T - t_0)} h^2 (M + M^2 + 1) \\
&\leq h^2 C(T - t_0) e^{(\Omega + C_E M)(T - t_0)} (M + M^2 + 1).
\end{aligned}$$

Die Aussage gilt, solange

$$\begin{aligned}
& Mh \geq h^2 C(T - t_0) e^{(\Omega + C_E M)(T - t_0)} (M + M^2 + 1) \\
\iff & h \leq \frac{M}{C(T - t_0) e^{(\Omega + C_E M)(T - t_0)} (M + M^2 + 1)},
\end{aligned}$$

h also hinreichend klein ist, denn der Ausdruck auf der rechten Seite ist eine Konstante, gleichmäßig in $t_0 \leq t_n \leq T$. Damit ist die Zwischenbehauptung gezeigt. ■

Damit lässt sich die in (3.25) auftretende Summe der $\|e_k\|$ abschätzen durch

$$\sum_{k=1}^{n-1} \|e_k\| \leq \sum_{j=1}^{n-1} Mh \leq Mnh \leq C_A.$$

Daraus ergibt sich, dass $\Omega(t_n - t_{m+1}) + C_E \sum_{j=m+1}^{n-1} \|e_j\| \leq \widetilde{C}_A$. Dies zeigt mit $\overline{C} := e^{\widetilde{C}_A}$ wiederum $e^{\Omega(t_n - t_{m+1}) + C_E \sum_{j=m+1}^{n-1} \|e_j\|} \leq \overline{C}$, wobei die auftretende Konstante immer noch gleichmäßig in $t_0 \leq t_n \leq T$ ist. Durch Einsetzen dieser Ungleichung und der Zwischenbehauptung in (3.25), erhält man

$$\begin{aligned}
\|e_n\| &\leq Ch \sum_{m=0}^{n-1} \overline{C}(h\|e_m\| + \|e_m\|^2 + h^p) \\
&\leq C\overline{C}h^{p+1} \sum_{m=0}^{n-1} 1 + C\overline{C} \sum_{m=0}^{n-1} (h\|e_m\| + Mh\|e_m\|) \\
&\leq C\overline{C}hnh^p + C\overline{C}(1 + M)h \sum_{m=0}^{n-1} \|e_m\|
\end{aligned}$$

$$\leq C_1 h^p + C_2 h \sum_{m=0}^{n-1} \|e_m\|$$

mit neuen Konstanten $C_1 := C\overline{C}(T - t_0)$ und $C_2 := C\overline{C}(1 + M)$ gleichmäßig in $t_0 \leq t_n \leq T$. Damit sind die Voraussetzungen des diskreten Lemmas von Gronwall mit $\alpha = C_1 h^p$, $\beta = C_2 h$ und $\eta_0 = \cdots = \eta_{n-1} = 1$ erfüllt. Demnach ist also

$$\begin{aligned} \|e_n\| &\leq C_1 h^p e^{C_2 h \sum_{m=0}^{n-1} 1} \\ &\leq C_1 h^p e^{C_2 h (T - t_0)} \\ &\leq C h^p \end{aligned}$$

mit einer neuen Konstante C, die wieder gleichmäßig in $t_0 \leq t_n \leq T$ ist. Damit ist die gesuchte Schranke gefunden und die Behauptung gezeigt. □

Das gerade gezeigte Hauptresultat bildet den Höhepunkt des theoretischen Teils dieser Arbeit und schließt die Fehleranalyse ab. Die weiteren Kapitel widmen sich nun einer Implementierung der Rosenbrock-Verfahren.

Kapitel 4

Mathematische Grundlagen der Implementierung

In diesem Kapitel werden einige der für die im nächsten Kapitel beschriebene Implementierung notwendigen mathematischen Grundlagen erklärt. Den Anfang machen zwei konkrete Verfahren, die den Ordnungsbedingungen aus Tabelle (3.1) genügen und die Ordnung drei beziehungsweise vier haben. Nachfolgend wird die in der Implementierung verwendete Schrittweitenkontrolle erläutert. Es folgt eine Erweiterung des Verfahrens auf nichtautonome Systeme. Die letzten beiden Abschnitte befassen sich mit Krylov-Verfahren und damit, wie man diese effizient einsetzt, um Produkte von Matrixfunktionen mit Vektoren zu berechnen.

4.1 Konkrete Verfahren

Dieser Abschnitt befasst sich mit zwei konkreten exponentiellen Rosenbrock Integrationsschemata. Dazu gehört jeweils ein eingebetteter Fehlerschätzer. Dabei handelt es sich ebenfalls um ein exponentielles Rosenbrock-Schema einer Ordnung kleiner als das eigentliche Verfahren, das jedoch dieselben inneren Stufen benutzt. Die Idee dabei ist die folgende: Berechne zusätzlich zu der numerischen Lösung u_n auch die Lösung mit dem Fehlerschätzerverfahren $\widehat{u}_n$. Subtrahiere beide und erhalte

$$Ch^p \geq \|e_n\| \approx \|u_n - \widehat{u}_n\| =: \|\widehat{e}_n\|.$$

Dabei wird eigentlich der Fehler des eingebetteten Verfahrens abgeschätzt, wenn man die Lösung höherer Ordnung als „exakt“ annimmt. Um den Fehler des eigentlichen Verfahrens zu schätzen, bräuchte man allerdings ein Verfahren, dessen Ordnung echt höher wäre. Dieser Aufwand wäre viel zu hoch, um „nur“ den Fehler zu schätzen.

Die beiden Verfahren sind einmal von Ordnung drei mit einem Fehlerschätzer von Ordnung zwei (`exprb32`) und einmal von Ordnung vier mit Fehlerschätzer der Ordnung drei (`exprb43`).

Das `exprb32`-Schema ergibt sich zu

$$\begin{array}{c|cc} c_1 & & \\ c_2 & a_{12} & \\ \hline & b_1 & b_2 \\ & \widehat{b}_1 & \widehat{b}_2 \end{array} = \begin{array}{c|cc} 0 & & \\ 1 & \varphi_1 & \\ \hline & \varphi_1 - 2\varphi_2 & 2\varphi_2 \\ & \varphi_1 & 0 \end{array}$$

Dass dieses Verfahren tatsächlich die Bedingungen von Ordnungstabelle (3.1) bis zur Ordnung drei erfüllt, lässt sich elementar nachrechnen.

Das `exprb43`-Schema hat die folgende Form:

$$
\begin{array}{c|ccc}
c_1 & & & \\
c_2 & a_{12} & & \\
c_3 & a_{13} & a_{23} & \\
\hline
 & b_1 & b_2 & b_3 \\
 & \widehat{b}_1 & \widehat{b}_2 & \widehat{b}_3
\end{array}
=
$$

$$
\begin{array}{c|ccc}
0 & & & \\
\frac{1}{2} & \frac{1}{2}\varphi_1(\frac{1}{2}\cdot) & & \\
1 & \multicolumn{3}{c}{\varphi_1} \\
\hline
 & \varphi_1 - 14\varphi_3 + 36\varphi_4 & 16\varphi_3 - 48\varphi_4 & -2\varphi_3 + 12\varphi_4 \\
 & (3\alpha+1)\varphi_1 + (-8+3\beta)\varphi_3 & -4\alpha\varphi_1 + (8-4\beta)\varphi_3 & \alpha\varphi_1 + \beta\varphi_3
\end{array}
$$

Der Fehlerschätzer hat hier zwei Parameter α und β. Diese liefern für alle Wahlen ein Integrationsschema der Ordnung drei. Wählt man dabei $\beta = 1 - 6\alpha$, erhält das Verfahren sogar schwache Ordnung vier. Das bedeutet, dass das Schema bei $J_n \longrightarrow 0$ Ordnung vier erhält. Dies ist nicht erwünscht, denn dann haben Verfahren und Fehlerschätzer dieselbe Ordnung. Dies kann zu Auslöschungen im Fehler führen.

4.2 Schrittweitenkontrolle

In diesem Abschnitt wird die Schrittweitenkontrolle für den `exprb`-Integrator erläutert. Diese berechnet - falls aktiviert - mithilfe der Schrittweite des aktuellen und des letzten Schrittes und des Fehlerschätzers die Schrittweite für den nächsten Schritt.

Der Fehlerschätzer ist dabei die Differenz der beiden Iterierten des eingebetteten und des eigentlichen Verfahrens

$$\widehat{e}_n := u_n - \widehat{u}_n.$$

Um den Fehler in der Größenordnung von vorgegebenen absoluten und relativen Toleranzen *ATol* und *RTol* zu erhalten, fordern wir, dass folgende skalierte Norm des Fehlers durch Eins beschränkt ist:

$$\|\widehat{e}_n\|_{sc} = \left(\frac{1}{d} \sum_{i=1}^{d} \left| \frac{\widehat{e}_n(i)}{sc(i)} \right|^2 \right)^{1/2} \leq 1.$$

Dabei sei $\widehat{e}_n(i)$ die i-te Komponente von $\widehat{e}_n$, $sc(i)$ die i-te des Vektors sc, der durch

$$sc = ATol + \max\{|u_n|, |u_{n-1}|\} \cdot RTol$$

gegeben ist. *ATol* darf ein Skalar oder, falls für verschiedene Komponenten verschiedene

Größenordnungen zu erwarten sind, ein Vektor sein. *RTol* ist immer ein Skalar. Der Absolutbetrag ist komponentenweise zu verstehen.

Wird diese Forderung nicht erfüllt, so wird der aktuelle Schritt verworfen und die Schrittweite verkleinert.

Es wird erwartet, dass $\|\widehat{e}_n\| = \mathcal{O}(h^p)$. Setzt man voraus, dass der Fehler sich im nächsten Schritt etwa gleich verhält, so würde man $h_{new} = h \cdot f_C$ mit

$$f_C = \|\widehat{e}_n\|^{1/p}$$

wählen um eine Fehlernorm von etwa eins zu erreichen. Dabei spielt es keine Rolle, ob der aktuelle Schritt verworfen wurde oder nicht.

Zusätzlich wurde noch eine zweite Methode zu Rate gezogen, die so genannte Gustafsson-Methode (vgl [9]), deren Anpassungsfaktor sich als

$$f_G = \frac{h}{h_{old}} \left(\frac{\|\widehat{e}_{n-1}\|_{sc}}{\|e_n\|_{sc}^2} \right)^{1/p}$$

errechnet. h_{old} ist dabei die Schrittweite des vorangegangenen Zeitschritts.

Die endgültige Schrittweite errechnet sich mit dem Sicherheitsfaktor $f_s = 0.9$ und den beiden Schranken $f_{min} = 0.2$ und $f_{max} = 5$ zu

$$h_{new} = h \cdot \max\left\{\min\left\{f_S f_C, f_S f_G, f_{max}\right\}, f_{min}\right\}.$$

4.3 Nichtautonome Systeme

Dieser Abschnitt beschäftigt sich mit nichtautonomen Systemen

$$u'(t) = F(t, u(t)), \qquad u(t_0) = u_0,$$

also solchen, bei denen die rechte Seite explizit von der Zeitvariablen t abhängt. Diese lassen sich über den Umweg

$$\widetilde{F}\left(\begin{bmatrix} t \\ u \end{bmatrix}\right) := \begin{bmatrix} 1 \\ F(t,u) \end{bmatrix}, \qquad \widetilde{u}(t) := \begin{bmatrix} t \\ u(t) \end{bmatrix} \tag{4.1}$$

als autonome Differentialgleichungen schreiben. Dann hat die Jacobi-Matrix die Form

$$\widetilde{J}_n = \begin{pmatrix} 0 & 0 \\ v_n & J_n \end{pmatrix} \text{ mit } v_n = \frac{\partial \widetilde{F}}{\partial t}(t_n, u_n) \text{ und wie bisher } J_n = \frac{\partial \widetilde{F}}{\partial u}(t_n, u_n).$$

Daraus ergibt sich g_n dann zu

$$g_n(t,u) = F(t,u) - \widetilde{J}_n \begin{bmatrix} t \\ u \end{bmatrix} = F(t,u) - J_n u - v_n t.$$

Um die Matrixfunktionen mit dem neuen $\widetilde{J}_n$ zu berechnen, benutzen wir

$$\varphi(h\widetilde{J}_n) = \begin{pmatrix} \varphi(0) & 0 \\ h\widehat{\varphi}(hJ_n)v_n & \varphi(hJ_n) \end{pmatrix},$$

mit $\widehat{\varphi}(z) = \frac{\varphi(z)-\varphi(0)}{z}$, also $\widehat{\varphi}_k(z) = \varphi_{k+1}(z)$, was sich aus der Cauchy-Integralformel und der Struktur von $\widetilde{J}_n$ ergibt.

Man betrachtet wieder die Vektoren $D_{ni} = g(t_n + c_i h, u(t_n + c_i h)) - g(t_n, u_n)$ und wendet das autonome Schema aus Abschnitt 2.3 auf (4.1) an. Daraus kann man das exponentielle Rosenbrock-Schema für nichtautonome Differentialgleichungen

$$U_{ni} = u_n + c_i h\varphi_1(c_i hJ_n)F(t_n,u_n) + h\sum_{j=2}^{i-1} a_{ij}(hJ_n)D_{nj} + h^2 c_i^2 \varphi_2(c_i hJ_n)v_n, \qquad 1 \le j \le s \tag{4.2a}$$

$$u_{n+1} = u_n + h\varphi_1(hJ_n)F(t_n,u_n) + h\sum_{i=1}^{s} b_i(hJ_n)D_{ni} + h^2\varphi_2(hJ_n)v_n. \tag{4.2b}$$

herleiten.

4.4 Krylov-Verfahren und Matrixfunktionen

Dieser Abschnitt behandelt so genannte Krylov-Verfahren und erläutert, wie man diese benutzen kann, um Produkte von Matrixfunktionen mit Vektoren zu nähern. Dies wird beispielsweise notwendig, wenn die Dimension der Jacobi-Matrix der rechten Seite der Differentialgleichung (2.7) sehr groß ist. In diesem Fall ist es zu teuer, die Matrixfunktionen aus dem Rosenbrock-Schema (2.10) direkt zu berechnen.

Zunächst wird die Frage geklärt, was man sich überhaupt darunter vorstellen soll, eine Matrix (oder einen linearen Operator) in eine skalare Funktion einzusetzen. Eine Antwort liefert die auf Matrizen verallgemeinerte Cauchy-Integralformel

$$\varphi(A) = \frac{1}{2\pi\,\mathrm{i}} \int_\Gamma \varphi(\lambda)(\lambda I - A)^{-1}\,d\lambda \tag{4.3}$$

mit einer Kurve $\Gamma \subseteq \mathbb{C}$, die die Eigenwerte von A umschließt (vgl [10]).

Will man nun $\varphi(A)b$ berechnen, treten durch Vorbeischieben des Vektors an $d\lambda$ lineare Gleichungssysteme $x(\lambda) = (\lambda I - A)^{-1}b$, also

$$(\lambda I - A)x(\lambda) = b, \tag{4.4}$$

für jedes λ auf der Kurve auf. Diese Gleichungssysteme können mit Krylov-Verfahren genähert werden. Dazu sei zunächst die Definition von Krylov-Räumen angegeben:

Definition 4.1. *Sei $A \in \mathbb{C}^{n\times n}$ eine Matrix, $b \in \mathbb{C}^n$ ein Vektor, dann ist der m-te Krylov-Raum zu A und b durch*

$$\mathcal{K}_m(A, b) = \operatorname{span}\left\{b, Ab, A^2b, \cdots, A^{m-1}b\right\}$$

gegeben.

Es ist nötig, von diesen Räumen Basen V_m zu berechnen. Diese werden iterativ so konstruiert, dass in jedem Schritt von V_{m-1} zu V_m nur ein neuer Basisvektor v_m hinzu genommen wird. Dieser muss dann automatisch Komponenten in Richtung $A^{m-1}b$ enthalten. Dies macht man sich iterativ zunutze, denn v_{m-1} hat dann Komponenten in Richtung $A^{m-2}b$. Daher genügt es, die neue Richtung Av_{m-1} anstatt von $A^{m-1}b$ zu benutzen, welche deutlich stabiler zu berechnen ist.

Es ergibt sich für jede solche iterierte Basis V_m von $\mathcal{K}_m(A, b)$ die folgende Beziehung:

$$AV_m = V_{m+1}\widetilde{H}_m$$

mit einer so genannten oberen Hessenberg-Matrix $\widetilde{H}_m \in \mathbb{C}^{m+1\times m}$. Eine obere Hessenberg-Matrix ist eine obere Dreiecksmatrix, die zusätzlich noch auf der ersten unteren Nebendiagonale nicht-Null Einträge haben darf. Es kann vorkommen, dass Av_m in $\mathcal{K}_m(A, b)$ liegt. In diesem Fall hat der Krylov Raum seine maximale Dimension erreicht und wird stationär. Lässt sich Av_m als Linearkombination aus Vektoren von V_m schreiben, so ist $Av_{m+1} = A^2v_m \in \mathcal{K}_{m+1}(A, b) \subseteq \mathcal{K}_m(A, b)$. In diesem Fall bricht der Algorithmus ab. Ist dies nicht der Fall, ist die Hessenberg-Matrix sogar nicht-reduziert. Das bedeutet, dass die erste untere Nebendiagonale überhaupt keine Nullen enthält. Äquivalent zu obiger Relation kann man auch

$$AV_m = V_{m+1}H_m + h_{m+1,m}v_{m+1}e_m^T \tag{4.5}$$

schreiben, mit

$$\widetilde{H}_m = \begin{bmatrix} & H_m & \\ 0 & \cdots & 0\ h_{m+1,m} \end{bmatrix}.$$

Aus Stabilitätsgründen ist es immer günstig, die Vektoren zu orthonormieren. Daher ist eine Möglichkeit, eine solche Basis zu berechnen, das so genannte Arnoldi-Verfahren, eine Modifikation des Gram-Schmidt-Orthogonalisierungsverfahrens, angepasst an Av_{m-1} statt $A^m b$:

Algorithmus 4.1 Arnoldi-Verfahren**.**

geg. $A \in \mathbb{C}^{n\times n}$, $b \in \mathbb{C}^n$
$\beta := \|b\|$, $v_1 := b/\beta$
for $m = 0, 1, \ldots$ do
  for $j = 1, \ldots, m$ do
    $h_{j,m} := v_j^H A v_m$
  end for
  $\widetilde{v}_{m+1} := Av_m - \sum_{j=1}^{m} h_{j,m} v_j$
  $h_{m+1,m} = \|\widetilde{v}_{m+1}\|$
  $v_{m+1} = \widetilde{v}_{m+1}/h_{m+1,m}$
end for

Will man nun Gleichungssysteme $Ax = b$ mit einem Krylov-Verfahren lösen, gibt es verschiedene Ansätze dazu. Der hier interessante ist der so genannte Galerkin-Ansatz. Dazu sei zu einer noch zu bestimmenden Iterierten $x_m \in K_m(A, b)$ das Residuum $r_m := Ax_m - b$ definiert. $x_m = V_m y_m$ wird nun so bestimmt, dass r_m orthogonal auf $\mathcal{K}_m(A, b)$ steht:

$$
\begin{aligned}
& r_m \perp V_m \\
\iff \quad & 0 = V_m^H r_m \\
& = V_m^H (A V_m y_m - b) \\
& = V_m^H (V_m H_m y_m + h_{m+1,h} v_{m+1} e_m^T - \beta V_m e_1) \\
& = \underbrace{V_m^H V_m}_{=I} H_m y_m + h_{m+1,h} \underbrace{V_m^H v_{m+1}}_{=0} e_m^T - \beta \underbrace{V_m^H V_m}_{=I} e_1 \\
\iff & H_m y_m = \beta e_1 \\
\iff & y_m = \beta {H_m}^{-1} e_1 \\
\iff & x_m = \beta V_m {H_m}^{-1} e_1.
\end{aligned}
$$

Die Gleichung $b = \beta V_m e_1$ folgt aus der zweiten Zeile des Arnoldi-Verfahrens, denn $v_1 = b/\beta$. Die Invertierbarkeit der Matrix H_m ist, wie sich gleich zeigen wird, ist hier immer erfüllt.

Ziel ist es nun, damit für eine – zunächst skalare – Funktion $\varphi : \mathbb{C} \longrightarrow \mathbb{C}$ den Ausdruck $\varphi(A)b$ zu berechnen. In Anwendung des binomischen Lehrsatzes ergibt sich, dass

$$(\lambda I - A)^k = \sum_{j=0}^{n} \binom{k}{j} \lambda^j A^{k-j}$$

und damit $(\lambda I - A)^k b \in \text{span}\left\{b, Ab, \cdots, A^k b\right\}$ gilt. Daher ist $\mathcal{K}_m(A, b) = \mathcal{K}_m(\lambda I - A, b)$ für alle $\lambda \in \mathbb{C}$. Also kann man für alle Gleichungssysteme der Form (4.4) denselben Krylov-Raum aufbauen. Analog zur Relation (4.5) folgt

$$(\lambda I - A)V_m = V_{m+1}(\lambda I - H_m) + h_{m+1,m} v_{m+1} e_m^T.$$

Die Galerkin-Iterierten sind dann $x_m(\lambda) = \beta V_m(\lambda I - H_m)^{-1} e_1$. Da der Wertebereich von H_m eine Teilmenge dessen von A ist, umschließt die Kurve Γ die Eigenwerte von H_m und für jedes $\lambda \in \Gamma$ ist $\lambda I - H_m$ daher invertierbar. Die gesuchte Approximation an das Produkt der Matrixfunktion mit b ergibt sich dann mit der Cauchy-Integralformel (4.3) durch

$$\begin{aligned}
\varphi(A)b &= \frac{1}{2\pi \,\mathbb{i}} \int_\Gamma \varphi(\lambda)(\lambda I - A)^{-1}\, d\lambda \; b \\
&= \frac{1}{2\pi \,\mathbb{i}} \int_\Gamma \varphi(\lambda) x(\lambda)\, d\lambda \\
&\approx \frac{1}{2\pi \,\mathbb{i}} \int_\Gamma \varphi(\lambda) x_m(\lambda)\, d\lambda \\
&= \frac{1}{2\pi \,\mathbb{i}} \int_\Gamma \varphi(\lambda) \beta V_m(\lambda I - H_m)^{-1} e_1\, d\lambda \\
&= \beta V_m \frac{1}{2\pi \,\mathbb{i}} \int_\Gamma \varphi(\lambda)(\lambda I - H_m)^{-1}\, d\lambda e_1 \\
&= \beta V_m \varphi(H_m) e_1.
\end{aligned}$$

Es genügt also, die Matrixfunktion nur auf der kleinen, $m \times m$-dimensionalen Matrix H_m direkt zu berechnen. Dies kann beispielsweise durch Diagonalisierung oder Padé-Approximationen geschehen.

Weiterführende Informationen zu Matrixfunktionen, die mit Krylov-Verfahren genähert werden, finden sich zum Beispiel in [3].

4.5 Abbruchkriterium für den Krylov-Prozess

Dieser Abschnitt beschäftigt sich mit einem Abbruchkriterium für den Krylov-Prozess, falls die Matrixfunktionen des Schemas mit dem Arnoldi-Algorithmus approximiert werden.

Um ein Abbruchkriterium zu erhalten, wird das so genannte verallgemeinerte Residuum

$$res_m = h\|b\| h_{m+1,m} (\varphi(hH_m))_{m,1} v_{m+1}$$

verwendet (vgl. [8]). Der Fehler der Approximation in der äußeren Stufe ist

$$p_{n+1} = \left(\|hF(u_n)\| V^{(1)}_{m^{(1)}} \varphi_1(hH^{(1)}_{m^{(1)}}) e_1 - h\varphi_1(hJ_n)F(u_n) \right) + \sum_{i=2}^{s} \left(\|hD_{ni}\| V^{(i)}_{m^{(i)}} b_i(hH^{(i)}_{m^{(i)}}) e_1 - hb_i(hJ_n)D_{ni} \right).$$

Dies sieht man, indem man obige Approximation von der Umformulierung aus Abschnitt 2.3 abzieht. Dabei sind $V^{(1)}_{m^{(1)}}$ und $H^{(1)}_{m^{(1)}}$ die Matrizen aus dem Krylov-Prozess mit J_n und $F(u_n)$ und $V^{(i)}_{m^{(i)}}$ und $H^{(i)}_{m^{(i)}}$ die aus denen mit J_n und D_{ni}.

Um zu erkennen, wie gut die Krylov-Approximation sein muss, um keine Ordnung zu verlieren, sei aus [5] der folgende Satz zitiert:

Theorem 4.2. *Unter den Voraussetzungen von Theorem 3.11 erfüllt die Lösung von*

$$\widetilde{U}_{ni} = e^{c_i hJ_n}\widetilde{u}_n + h\sum_{j=1}^{i-1} a_{ij}(hJ_n)g_n(\widetilde{U}_{nj}) + P_{ni}, \qquad 1 \le i \le s$$

$$\widetilde{u}_{n+1} = e^{hJ_n}\widetilde{u}_n + h\sum_{i=1}^{s} b_i(hJ_n)g_n(\widetilde{U}_{ni}) + p_{n+1}$$

die Fehlerschranke

$$\|\widetilde{u}_{n+1} - u(t_{n+1})\| \le C\sum_{j=0}^{n}\left(h^2 \sum_{i=1}^{s} \|P_{ji}\| + \|p_{j+1}\| \right) + Ch^p$$

gleichmäßig für $t_0 \le t_n \le T$ in einem Streifen entlang der exakten Lösung.

Die Fehler P_{ni} der inneren Stufen haben bereits einen Faktor h^2, so dass nach dem Aufsummieren über alle Zeitschritte noch ein Faktor h übrig bleibt. Es muss also garantiert werden, dass die Summe der $\|p_{j+1}\|$ in der gleichen Größenordnung ist wie der Fehler des Zeitschritts. Letzterer wird durch die Schrittweitenkontrolle in der $\|\cdot\|_{sc}$-Norm auf Eins gebracht. Lässt man den Krylov-Prozess also laufen, bis das Residuum

$$\|res_m\|_{sc} \le h$$

erfüllt, ist die gewünschte Genauigkeit erreicht.

Günstig daran ist, dass in der Norm des verallgemeinerten Residuums die Norm des Vektors v als Faktor enthalten ist. Daher ist für kleine $\|v\|$, wie es bei den D_{ni} der Fall ist, nur eine geringe Krylov-Raum-Dimension nötig.

4.6 Dense Output

In diesem Abschnitt geht es um so genanntem *dense output*. Damit bezeichnet man eine stetige Fortsetzung der numerischen Lösung auf das gesamte Integrationsintervall, die eine möglichst hohe Ordnung hat. Diese Fortsetzung soll mit möglichst geringem Mehraufwand zu berechnen und zu jeder Zeit $t \in [t_0, T]$ auszuwerten sein. Die hier verwendete Methode erzeugt sogar eine stetig-differenzierbare Fortsetzung. Es wird vermutet, dass ihr Fehler dieselbe Größenordnung wie das Verfahren selbst hat. Für einen Beweis dieser Aussage fehlt die Eigenschaft, dass

$$\|F(y_n) - F(y(t_n))\| \leq Ch^{p-1}, \qquad n \geq 0 \tag{4.6}$$

gilt. Ein Beweis dieser Eigenschaft geht über den Rahmen dieser Arbeit hinaus und sei daher als Vermutung in diesem Abschnitt vorausgesetzt.

In Anpassung an die `MATLAB` Notation wird die Lösung der Differentialgleichung von nun an mit y bezeichnet. Da wir uns in einem Implementierungskontext befinden, wird weiterhin $\dim(X) < \infty$ angenommen. Das Ziel dieses Abschnittes ist es also eine Funktion $y_\theta : [t_0, T] \longrightarrow X$ zu konstruieren, so dass $\|y_\theta(t) - y(t)\| \leq Ch^p$ für alle $t \in [t_0, T]$ und $y_\theta(t_n) = y_n$ für alle n gilt.

Es wird Hermite-Interpolation verwendet, um diese Lösung zusammen zu setzen. Dabei wird auf jedem Intervall eines Zeitschritts $[t_n, t_{n+1}]$ ein kubisches Polynom konstruiert, so dass die Interpolationsbedingungen $y_\theta(t_n) = y_n$, $y_\theta(t_{n+1}) = y_{n+1}$, $y'_\theta(t_n) = F(y_n) =: F_n$ und $y'_\theta(t_{n+1}) = F(y_{n+1}) =: F_{n+1}$ erfüllt sind. Das Ergebnis ist dann automatisch differenzierbar, da es an den Verklebungspunkten differenzierbar konstruiert und dazwischen ein Polynom ist. Definiert man

$$s(\theta) := (1-\theta)y_n + \theta y_{n+1} + \theta(\theta-1)\left((1-2\theta)(y_{n+1}-y_n) + (\theta-1)hF_n + \theta hF_{n+1}\right), \tag{4.7}$$

so ist die Ableitung von s gegeben durch

$$\begin{aligned} s'(\theta) = -y_n + y_{n+1} &+ (2\theta-1)\left((1-2\theta)(y_{n+1}-y_n) + (\theta-1)hF_n + \theta hF_{n+1}\right) \\ &+ \theta(\theta-1)\left(-2(y_{n+1}-y_n) - hF_n + hF_{n+1}\right). \end{aligned}$$

Setzt man nun für alle n und $t_n < \theta \leq t_{n+1}$

$$y_\theta(\theta) := s\left(\frac{\theta - t_n}{t_{n+1} - t_n}\right),$$

so sind die geforderten Interpolationseigenschaften tatsächlich erfüllt, denn

$$y_\theta(t_n) = s\left(\frac{t_n - t_n}{h}\right) = s(0) = y_n, \quad y_\theta(t_{n+1}) = s\left(\frac{t_{n+1} - t_n}{h}\right) = s(1) = y_{n+1},$$

$$y'_\theta(t_n) = \frac{\mathrm{d}}{\mathrm{d}\theta} s\left(\frac{\theta - t_n}{h}\right)\Big|_{\theta=t_n} = \frac{s'(0)}{h} = \frac{-y_n + y_{n+1} - (y_{n+1} - y_n - hF_n)}{h} = F_n \text{ und}$$
$$y'_\theta(t_{n+1}) = \frac{\mathrm{d}}{\mathrm{d}\theta} s\left(\frac{\theta - t_n}{h}\right)\Big|_{\theta=t_{n+1}} = \frac{s'(1)}{h} = \frac{-y_n + y_{n+1} + (-y_{n+1} + y_n + hF_{n+1})}{h} = F_{n+1}.$$

Um den Fehler des *dense output* zu verstehen, muss eine Hilfsfunktion $\widetilde{y}_\theta$ eingeführt werden. Sie entsteht die auf dieselbe Weise wie y_θ. Diesmal wird jedoch die exakte Lösung interpoliert: $\widetilde{y}_\theta(t_n) = y(t_n)$, $\widetilde{y}_\theta(t_{n+1}) = y(t_{n+1})$, $\widetilde{y}'_\theta(t_n) = F(y(t_n))$ und $\widetilde{y}'_\theta(t_{n+1}) = F(y(t_{n+1}))$. Nach Voraussetzung 2 ist die Lösung hinreichend oft differenzierbar, also $y \in C^4([t_0, T])$ und $y^{(4)}$ gleichmäßig beschränkt. Satz 2.21 aus [2] besagt für jede Komponente i und $t_n < \theta \leq t_{n+1}$:

$$|((y - \widetilde{y}_\theta)(\theta))_i| \leq Ch^4 \max_{\xi \in [t_n, t_{n+1}]} |(y^{(4)}(\xi))_i| \leq Ch^4$$
$$\Rightarrow \|(y - \widetilde{y}_\theta)(\theta)\| \leq Ch^4$$

Würde also die exakte statt der numerischen Lösung interpoliert, so wäre die richtige Ordnung bereits gezeigt. Um die Fehlerschranke auch für die numerische Lösung zu erhalten, muss nun die Differenz von y_θ und $\widetilde{y}_\theta$ untersucht werden. Schreibe

$$\widetilde{y}_\theta(\theta) = \widetilde{s}\left(\frac{\theta - t_n}{t_{n+1} - t_n}\right)$$

mit einem Polynom $\widetilde{s}$ in der Form wie s. Es müssen dabei nur y_n durch $y(t_n)$, y_{n+1} durch $y(t_{n+1})$, F_n durch $F(y(t_n))$ und F_{n+1} durch $F(y(t_{n+1}))$ ersetzt werden. Kürzt man das Argument der Polynome s und $\widetilde{s}$ mit t ab, so ist $\|y_\theta(\theta) - \widetilde{y}_\theta(\theta)\|$ durch

$$|(1-t)|Ch^p + |t|Ch^p + |t(t-1)(1-2t)|2Ch^p +$$
$$|t(t-1)(t-1)|h\|F_n - F(y(t_n))\| + |t(t-1)t|h\|(F_{n+1} - F(y(t_{n+1}))\|$$

beschränkt, mit $0 \leq t \leq 1$. Nach Vermutung (4.6) ist $\|F_n - F(y(t_n))\|$ durch Ch^{p-1} beschränk. Die Polynome in den Beträgen sind alle stetig und nehmen daher auf dem kompakten Intervall $[0, 1]$ ihr Minimum und Maximum an. Schätzt man ihren Wert durch das betragliche Maximum aller auftretenden Polynome über das ganze Intervall ab, ist damit dann

$$\|y_\theta(\theta) - \widetilde{y}_\theta(\theta)\| \leq Ch^p$$

gezeigt. Der *dense output*-Fehler verhält sich also unter Vermutung (4.6) wie angekündigt:

$$\|y(t) - y_\theta(t)\| \leq \|y(t) - \widetilde{y}_\theta(t)\| + \|\widetilde{y}_\theta(t) - y_\theta(t)\| \leq Ch^4 + Ch^p \leq Ch^p.$$

Kapitel 5

Implementierung

In diesem Kapitel wird eine MATLAB-Implementierung der exponentiellen Rosenbrock-Verfahren (2.10), der `exprb` Integrator, diskutiert. Ziel dieser Implementierung ist eine möglichst gute Kompatibilität zu den entsprechenden Standardtools von MATLAB, wie etwa `ode45`, `ode23` oder `ode23s`. Möglichst viele der von diesen unterstützten Optionen und die Möglichkeit zum Erzeugen des im vorigen Kapitel erklärten *dense output* sollten verfügbar sein. Eine vollständige Unterstützung aller Optionen ist nicht möglich, da einige auf spezielle Lösungsverfahren zugeschnitten sind.

Weiterhin ist der Code so geschrieben worden, dass eine Erweiterung auf andere exponentielle Integratoren leicht möglich ist. Ein Langzeitziel ist die Schaffung einer ganzen exponentiellen Toolbox, genannt `EXPODE`. Eine ähnliche Sammlung exponentieller Integratoren ist das `EXPINT` Paket [1]. Als Hilfsmittel dazu wurde eine Erweiterung der internen MATLAB-Methode `odeset` geschaffen, die eine möglichst hohe Flexibilität aufweist und dem Benutzer gute Hilfestellung leistet.

Zunächst wird der Integrator diskutiert, danach das Setzen von Optionen. Anschließend wird eine vollständige Liste aller von `exprb` unterstützen Optionen angegeben und detailliert erklärt. Als nächstes folgt eine Erklärung einiger interessanter Implementierungsdetails, die für Autoren weiterer `EXPODE` Integratoren hilfreich sind.

Dieser Teil der Arbeit wird als Dokumentation des Integrators auf der Webseite zur Verfügung gestellt werden. Aus diesem Grund ist der Rest dieses Kapitels in englischer Sprache verfasst.

To simplify the notation, optional function arguments (the ones that can be omitted) will be printed in braces. Do not confuse this with MATLAB's cell objects, these will not appear in this context. Example:

```
>> res = fun(neccesary, {optional});
```

Remark: In this documentation we use the name *Jacobian* for the derivative of the right hand side of the differential equation with respect to the phase space variable. Strictly spoken this is only correct for autonomous problems $u' = F(u)$. Nevertheless, for non-autonomous problems $u' = F(t, u)$ we also call $\frac{\partial F}{\partial u}$ the *Jacobian* of F even though the true Jacobian would additionally contain the derivative with respect to t.

5.1 The `exprb` Integrator

The `exprb` integrator solves a given ordinary differential equation with an exponential Rosenbrock-type method (2.10). After downloading and extracting the code package, you will get a directory called `expode`. This directory contains the `exprb` integrator, the options

helper `exprbset`, the information command `exprbinfo`, an analogon to `MATLAB`'s `deval`, `devalexp` and the function `initPaths` to setup `MATLAB`'s searchpath. Additionally there are some subdirectories. One of these is the `example` directory. The programs in that directory can be run directly and be used as a reference how to call the integrator.

To use `exprb` you either get to move the package's content to your `MATLAB` working directory or use

```
>> addpath /path/to/expode
```

to tell `MATLAB` where to find the integrator.

`exprb`'s user interface is adapted to `MATLAB`'s internal integrators. A call with all available arguments is of the following form:

```
>> [t, y] = exprb(@ode, {@jac}, {tspan, y0}, {opts}, {varargin});
```

To keep the explanation well-arranged, the possible call combinations will be explained step by step. The simplest way to invoke the integrator is

```
>> [t, y] = exprb(@ode, tspan, y0);
```

where `ode` is a function to evaluate the differential equation (and its first derivative) at given data `y` and `t`. `tspan = [t0, T]` defines the integration interval. You can also use `tspan = [t0, ..., tN]` instead. This will result in a solution evaluated at the given times instead of the ones chosen by the integrator internally. In that case, clearly we have `t = tspan`. This kind of solution is called *dense output* because one would typically use `tspan = linspace(t0, T, N + 1)` with more output data than the step selection would produce. `y0` is the initial condition to solve the equation with.

The `ode` function has to be callable in the following way:

```
>> res = ode(t, y, {flag});
```

`t` and `y` are the time and phase space variables respectively. `flag` can either be omitted or given as `flag = 'jacobian'` or `flag = 'df_dt'`. In the first case `ode` should return the evaluation of the right hand side of the equation at `t` and `y`, in the second case the derivative of the right hand side with respect to `y` – here called Jacobian, see the introduction of this chapter for explaination – has to be evaluated for the same data. The last case is only needed if the differential equation is non-autonomous. Then `ode` should return the derivative of the right hand side with respect to the time variable `t`. If the differential equation is autonomous, you have to set the option `NonAutonomous` to `'off'`, see the options section. A typical function body for `ode` is presented below.

```
function res = ode(t, y, flag)
    if nargin == 2 || isempty(flag)
       flag = '';
    end

    switch flag
       case ''
          res = evaluation of the right hand side;
       case 'jacobian'
          res = evaluation of the Jacobian;
       case 'df_dt'
          res = evaluation of the derivative of the ...
                right hand side w.r.t. t;
       otherwise
          error('Unknown flag: %s.', flag);
    end

    return res;
```

All function handles used can be replaced by inlines or the functions' names. All of these can be evaluated by the `EXPODE` integrators.

It is possible to source out the evaluation of the Jacobian into an additional function. In that case `exprb` has to be called this way:

```
>> [t, y] = exprb(@ode, @jac, tspan, y0);
```

where `function J = jac(t, y)` is the new function for the Jacobian. The handle to the Jacobian can alternatively be given as an integrator option. See the `Jacobian` option in section 5.3 for details. If this option is switched `'off'` and `JacobianV` is `'on'` instead, `jac` will be interpreted as a function to evaluate the product of the Jacobian with a vector. In that case `function res = jac(t, y, v)` has to be callable.

To use integrator options, invoke `exprb` in the following way:

```
>> [t, y] = exprb(@ode, {@jac}, tspan, y0, opts);
```

The construction of such an options object will be discussed in the next section.

If you want to hand over additional parameters to the `ode` function – stiffness parameters for springs or some dimensions for example – these parameters can be passed to the integrator as follows:

```
>> [t, y] = exprb(@ode, {@jac}, tspan, y0, {opts}, varargin);
```

varargin will be passed to the ode function directly via

```
>> res = ode(t, y, flag, varargin);
```

Please *do not* use a struct object as the first of the varargin arguments if you do not use integrator options. The integrator would confuse this argument with an options structure.

If you still want to do so, you have to provide an empty vector ([]) as the options parameter.

```
function res = ode(t, y, flag, varargin)
    if nargin == 2 || isempty(flag) || ~ischar(flag)
       if ~ischar(flag)
          varargin = {flag, varargin{:}};
       end
       flag = '';
    end

    % The rest can be the same as before.
```

If varargin is used, it will also be passed to the jac function if one was provided. It will be called with J = jac(t, y, varargin) in that case.

It is also possible to omit the integration interval, the initial condition, and the integrator options if the ode function is able to provide them. Calling exprb with

```
>> [t, y] = exprb(@ode, {@jac}, {varargin});
```

will invoke ode with the 'init' flag:

```
>> [tspan, y0, opts] = ode([], [], 'init', {varargin});
```

The example code for the ode function has to be extended by another case for 'init', as can be seen here.

```
      case 'init'
          res{1} = [t0, T];
          res{2} = y0;
          opts   = exprbset(option1, value1, option2, value2, ...);
                 % see section 5.2, exprbset
          res{3} = opts;
```

To be able to evaluate the solution at an arbitrary time after the integration is finished, the call

```
sol = exprb(@ode, {@jac}, tspan, y0, {opts}, varargin);
```

will generate a variable `sol` which can be used with the `devalexp` function via

```
>> [ y, dydt ] = devalexp(sol, t);
```

The argument `t` then is a vector of times in the integration interval. Then `y` contains the evaluation of the numerical solution and `dydt` its first derivative ot the times in `t`.

5.2 Integrator Options

`exprbset` is used to set options for the `exprb` integrator. It is an extension to `MATLAB`'s `odeset`. It would have been preferable to use `odeset` directly, but it has a fixed set of available options and is not extensible to support a larger set.

Both `odeset` and `exprbset` create a so called *options structure* or *options object*, which contains the options set by the user. These structures can be passed to the integrators. It has been explained in the previous section how to do that for `exprb`.

`exprbset` is compatible with the option objects created by `odeset`, so one can create an options structure with `odeset` and then extend it with `exprbset`. Unfortunately, it is not possible the other way around, since `odeset` removes all options it does not know about. The first sequence is more important though. Usually, a program will be written using `MATLAB`'s standard tools and then be extended to use other integrators.

The `exprbset` function can be called analogicalty to `odeset`. The simplest way to use it is

```
>> opts = exprbset(option, value);
```

where `option` is an option's name and `value` the user's choice. The latter one can be scalar, vector valued, logical, a string, or a function, depending on the option chosen. All supported option types are listed later in this section. The `exprbinfo` command provided by the `EXPODE` package can print options supported by the `exprb` integrator.

It is also possible to pass more option-value-pairs at once and extend already existing option objects:

```
>> opts = exprbset(option1, value1, option2, value2, ...);
>> opts = exprbset(opts, option3, value3, ...);
```

For a more comfortable usage, `exprbset` checks if the provided options are valid.

In the remaining part of this section, the supported value types for options will be discussed. All of the listed types can be checked for validity by `exprbset`. Which type is allowed for

which exprb option is described in the next section. The exprbinfo command in MATLAB can give this information.

```
>> exprbinfo({optname});
```

will display a list of all options supported by the integrator if optname is omitted. If it is given, more detailed help on the specific option will be printed. Using optname = '-' will give this detailed help on all options. Warning: the output is quite long, it should be used with

```
>> more on; exprbinfo('-'); more off;
```

The output of exprbinfo contains the option's type in the first line in squared brackets. For the AbsTol option this line reads

```
AbsTol - Absolute error tolerance [ positive scalar | positive vector {1e-06} ].
```

The vertical bar seperates alternative types, so the type is positive scalar or positive vector. The default value is set in braces after the type. The two types are each a combination of two other types: positive and scalar or vector respectively. Which types are compatible will be explained below. Another example for types is given by the JacobianV option. Here the exprbinfo command prints

```
JacobianV - ...  [ function_handle | {'off'} | 'on' ].
```

The allowed types in this case are boolean or function_handle. For types boolean and its generalisation list a list of accepted values will be stated. For a boolean this list is fixed and consists of the values 'on' (true) and 'off' (off), for the list type the list depends on the option. The listed values are always quoted strings. Now all option types will be discussed in detail.

Type scalar. This type can be any numerical type that is supported by MATLAB. scalar can be combined with integer, positive and index.

Type vector. This type can be a vector of any numerical type that is supported by MATLAB. scalar is the special case of this with length one, so any scalar is a vector as well. Vector can be combined with integer, positive and index.

Type integer. This type can be any integer number. This does not enforce the use of MATLAB's int8, int16 or similar types. value is accepted as an integer as long as the expression (value - round(value)) == 0 ist true. integer can be combined with vector, scalar and positive.

Type positive. This type can be any numerical type that is supported by MATLAB (scalar or vector) where all components are strictly positive. positive can be combined with vector, scalar and integer.

Type `non-negative`**.** This type allows any numerical `MATLAB` type (scalar or vector) where all components are not-negative (positive or zero). `non-negative` can be combined with `vector`, `scalar` and `integer`.

Type `negative`**.** This type can be any numerical type that is supported by `MATLAB` (scalar or vector) where all components are strictly negative. `negative` can be combined with `vector`, `scalar` and `integer`.

Type `non-positive`**.** This type allows any numerical `MATLAB` type (scalar or vector) where all components are not-positive (negative or zero). `non-positive` can be combined with `vector`, `scalar` and `integer`.

Type `index`**.** This type is an abbreviation for `positive integer scalar`. `index` can therefore not be combined with any other type.

Type `indices`**.** This type is an abbreviation for `positive integer vector`. `indices` can therefore not combined with any other type.

Type `boolean`**.** This type can be given in different forms. Either use `'on'` and `'direct'` or use the numerical values `true` and `false`. `boolean` is not combinable with any other type.

Type `list`**.** This type can be given in different forms. Either use one of the available values printed by the `exprbinfo` command or use the number of the list entry, start counting from `0`. Example: The `MatrixFunction` option's info states (ignore `function_handle` here, it will be discussed later)

```
MatrixFunction - ...  [ {'direct'} | 'arnoldi' | function_handle ]
```

which permits to set the values `'direct'` and `'arnoldi'`, in that order. Therefore `'off'` has number `0` and `'arnoldi'` has number `1`. Thus, it is possible to set value for the option `MatrixFunction` to use Arnoldi's method in the following ways:

```
>> opts = exprbset('MatrixFunction', 'arnoldi'); % or equivalently
>> opts = exprbset('MatrixFunction', 1);
```

`list` is cannot be combined with any other type.

Type `text`**.** This type can be any `MATLAB` `char` array (string). `text` cannot be combined with any other type.

Type `function_handle`**.** This type can be either a `MATLAB` function handle, a `MATLAB` inline function or a `char` array (string). A function handle to a function `func` is generated by `@func`. An inline function is generated by `MATLAB`'s `inline` command. If the argument is a string, it has to represent a function's name: use `'func'` for the function `func`. `function_handle` cannot be combined with any other type.

5.3 exprb's options

In this section, the available options for the `exprb` integrator will be discussed. There are several types of options: Options directly affecting the integration, a couple concerning the evaluation of the ODE functions, some for properties of the differential equation. Furthermore, a few options dealing with output specific settings and the last set controlling the verbosity of the integrator. These types will be grouped together to provide a better overview.

The first set of options deals with the integrator itself.

Option `Order`**.** *Integrator order to use*
Permitted values: `'three'` or `{'four'}`

Order to use for the integrator. The `exprb32` method is the order three method with two inner stages using the second order exponential Euler method as error estimator. For order four the `exprb43` method will be used. This one has three stages and uses a three staged order three error estimator.

Option `hConstant`**.** *Use constant step size*
Permitted values: `{'off'}` or `'on'`

Uses constant step size in the integrator. In case `hConstant` is set to `'on'`, consider setting the `InitialStep` option to specify the stepsize.

See also: `InitialStep`

Option `MaxStep`**.** *Maximal step size to use*
Permitted type: `non-negative scalar {0}`

Set the maximal step size to use in the integration process here. Setting `MaxStep` to 0 (default) will result in using $(\texttt{tfinal} - \texttt{t}_0)/100$, so using at least ten integration steps until the integrator is finished. This option will be ignored when using constant step size.

See also: `MinStep`, `InitialStep` and `hConstant`

Option `MinStep`**.** *Minimal step size to use*
Permitted type: `non-negative scalar {0}`

Set the minimal step size to use in the integration process here. Setting `MinStep` to 0 (default) will result in using `eps(t)` at integration time `t`, so that `t + h` is at least different from `t`. This option will be ignored when using constant step size.

See also: `MaxStep`, `InitialStep` and `hConstant`

Option `ErrorEstimate`. *Error Estimator Parameters*
Permitted type: `vector {[ 0 -2 ]}`

The order four integrator uses an error estimator that can be tweaked with two parameters `a` and `b`. These two parameters can be specified here with `ErrorEstimate` = [a, b]. Choose `b` $\sim=$ 1 − 6`a`, otherwise the error estimator gets close to weak order four instead of three.

Option `AbsTol`. *Absolute error tolerance*
Permitted type: `positive scalar` or `positive vector {1e-06}`

A scalar tolerance applies to all components of the solution vector. Elements of a vector of tolerances apply to corresponding components of the solution vector. `AbsTol` defaults to 10^{-6} in all solvers.

See also: `RelTol`

Option `RelTol`. *Relative error tolerance*
Permitted type: `positive scalar {0.001}`

This scalar applies to all components of the solution vector, and it defaults to 10^{-3} (0.1% accuracy) in all solvers. The estimated error in each integration step satisfies $\|err\| \leq 1$ in a norm scaled with $RelTol \cdot \max(\text{abs}(\texttt{y}(i), \text{abs}(\texttt{y}_2(i))) + AbsTol(i)$ in each component, where `y` is the numerical solution at the current, $\texttt{y}_2$ the one at the previous time step.

See also: `AbsTol`

Option `MatrixFunctions`. *Evaluation method for the matrix functions*
Permitted values: `{'direct'}`, `'arnoldi'` or `function_handle`

Set the method to evaluate the product of the matrix function with vectors. There are two builtin methods. The default `'direct'` method uses diagonalisation of the Jacobian to do this. If the Jacobian is too large, this will be too expensive computionally – use `'arnoldi'` in this case. All matrix functions will then be approximated in Krylov subspaces, the Arnoldi method is then used to compute a nested orthonormal basis of that space. Another alternative is to provide a custom function to compute these results. See the next section how this method has to work.

See also: `JacobianV`

Option `KrylovTestIndex`. *Dimensions of the Krylov subspaces to test the residual*
Permitted type: `vector of indicies {[ 1 2 3 4 6 8 11 15 20 27 36 ]}`

Sequence of ascending integers which indicate the dimensions of the Krylov subspaces where the residual is tested. The largest number is the maximum dimension. If the Krylov process does not terminate within this range, the time step size is reduced. This option only applies if MatrixFunctions is set to `'arnoldi'`.

See also: `MatrixFunctions`

Now we look at options concerning properties of the differential equation itself.

Option `NonAutonomous`. *Specifies whether the ODE is autonomous or not*
Permitted values: `'off'` or `{'on'}`

Set to `'on'` if the differential equation is not autonomous. The exponential Rosenbrock–scheme was designed for autonomous problems, so this has to be handled specially internally.

Option `Complex`. *Solution is complex*
Permitted values: `'off'` or `{'on'}`

Set to `'false'` if the solution does not have complex components. This will set all possibly numerically generated imaginary parts to zero.

Option `Symmetry`. *Structure of the Jacobian*
Permitted values: `{'symmetric'}` or `'skewsymmetric'`

Set to `'skewsymmetric'` if the Jacobian is skew–symmetric, leave it `'symmetric'` otherwise.

Option `Semilin`. *Specifies whether the ODE is semilinear*
Permitted values: `{'off'}` or `'on'`

Set to `'on'`, if the differential equation has the form

$$y' = A \cdot y + g(t, y)$$

Then add the flags `'gfun'` and `'dg_dy'` to your ODE file to evaluate the non–linear part g and its derivative with respect to y. Alternatively, you can use the `GFun` and `GJacobian` options to set these. If you use Krylov approximations to the matrix exponentials, you can also add the flag `'dg_dy_v'` to evaluate the Jacobian of g times a vector or use the `GJacobianV` option to do so.

See also: `GFcn`, `GJacobian`, `GJacobianV`, `MatrixFunctions` and `JacobianV`

The following options define ways of evaluating the Jacobian and other possibly required parts of the differential equation.

Option `Jacobian`. *Evaluation function for the ODE's right hand side's Jacobian*
Permitted type and values: `function_handle`, `'off'`, `{'on'}` or `matrix`

The `exprb` integrator needs to compute matrix functions with the Jacobian of the right hand side of the ODE. Therefore, either the Jacobian has to be evaluated, or the evaluation of Jacobian times vector has to be provided – this only works when using Krylov approximations to the matrix functions. If this option is set to `'on'` (default), the `ode` function will be called with

```
>> j=ode(t,y,'jacobian',varargin);
```

If it is a function handle, the call

```
>> jac=opts.Jacobian; j=jac(t,y,varargin);
```

will be executed.

See also: `JacobianV` and `MatrixFunctions`

Option `JacobianV`. *Evaluation function for the ODE's right hand side's Jacobian times vector*
Permitted type and values: `function_handle`, `{'off'}` or `'on'`

When using Krylov approximations to the matrix functions of the Jacobian of the right hand side times vectors, this option can be used to provide the result of the computation of the Jacobian multiplied by a vector. Sometimes it is faster to compute this directly instead of calculating the whole Jacobian. If this option is set to `'on'` (default), the ODE function will be called with

```
>> res=ode(t,y,'jacobian_v',v,varargin);
```

If it is a function handle, the call

```
>> jacv=opts.JacobianV; res=jacv(t,y,v,varargin);
```

will be executed.

See also: `JacobianV` and `MatrixFunctions`

Option `GFcn`. *Evaluation of the nonlinear part of the ODE, for semilinear problems*
Permitted type and values: `function_handle`, `'off'` or `{'on'}`

This option will only be used,if `Semilin` is `'on'`. Set this to `'on'` if the ODE file can evaluate the nonlinear part when used with flag `'gfun'`. Set to a function handle if an external function can evaluate it. Setting to `'off'` will result in reverting to the standard solving method, ignoring the special structure of the ODE.

See also: `Semilin`

Option `GJacobian`. *Evaluation of the Jacobian of the nonlinear part of the ODE, for semilinear problems*
Permitted type and values: `function_handle`, `'off'` or `{'on'}`

This option will only be used if `Semilin` is `'on'`. Set this to `'on'` if the ODE file can evaluate the Jacobian of the nonlinear part when used with flag `'dg_dy'`. Set to a function handle if an external function can evaluate it. Setting this to `'off'`, you will need to use Krylov approximations to the matrix functions and set the `GJacobianV` to something else than `'off'`.

See also: `Semilin`, `GJacobianV` and `MatrixFunctions`

Option `GJacobianV`. *Evaluation of the Jacobian of the nonlinear part of the ODE times vector, for semilinear problems*
Permitted type and values: `function_handle`, `{'off'}` or `'on'`

This option will only be used if `Semilin` is `'on'` and MatrixFunctions is not `'direct'`. Set this to `'on'` if the ODE file can evaluate the product of the Jacobian of the nonlinear part with a vector when used with flag `'dg_dy_v'`. Set to a function handle if an external function can evaluate it. Setting to `'off'` will use `GJacobian`.

See also: `Semilin`, `GJacobian` and `MatrixFunctions`

The next four options control the output of the `exprb` solver.

Option `Refine`. *Output refinement factor*
Permitted type: `positive integer scalar {1}`

This property increases the number of output points by the specified factor producing smoother output. Refine defaults to 1. Refine does not apply, if length(`tspan`) > 2.

Option `OutputFcn`. *Output function called after each time step*
Permitted type and value: `function_handle` or `{'off'}`

If a function handle is given, it is called after each time step. The output function has to understand the following calls:

```
>> outputFunction([t0,tfinal],y,'init',varargin)
>> outputFunction(t,y,'',varargin)
```

The first call should initialize the output function. The second call is executed at every time step – either the ones selected by the solver's error estimator, at the refined steps or at the steps definded by `tspan` argument at the solver's call. t_0, `tfinal` and `t` are scalar times. t_0 and `tfinal` limit the integration interval and `t` is the time of the current step. `y` is the

solution at t_0 on the 'init' call and the solution at t on init=''. varargin is an argument which will be forwarded from the solver call.

See also: OutputSel

Option OutputSel. *Indices of the solution given to the output function*
Permitted type: vector of indices {[]}

Only used, if an output function is used. If this argument is an empty vector, all indices of the solution are passed to the output function. Otherwise, y(OutputSel) is passed. $OutputSel = 1 : \text{length}(y)$ will give the same result as OutputSel = [].

See also: OutputFcn

This last set of options covers statistical and progress output, verbosity and the activation of a progressbar.

Option Stats. *Display status messages*
Permitted values: 'silent', {'off'}, 'on' or 'verbose'

Set the level of status messages printed by the integrator. If JacobianStats, StepStats and MatrixFunctionLog are set to 'auto', they will be activated only when Stats is set to 'verbose' and be deactivated on all. other settings. Setting this option to 'silent' will also disable warnings.

See also: JacobianStats, StepStats and MatrixFunctionStats

Option JacobianStats. *Display status messages for the evaluation of the Jacobian*
Permitted values: 'off', 'on' or {'auto'}

Control whether or not to display status messages for the evaluation of the Jacobian. When set to auto, this will be activated when Stats is set to 'verbose'. If enabled, this options prints

```
>> Evaluating Jacobian ...  done
```

each time the Jacobian is evaluated. If option Stats set to 'verbose' the Eigenvalues of the Jacobian will be plotted additionally.

See also: Stats

Option StepStats. *Display status messages for step size related events*
Permitted values: 'off', 'on' or {'auto'}

Control whether or not to display step size related status messages. When set to auto, this will be activated when Stats is set to 'verbose'. You will get informed about step size reductions and step rejections if this option is activated.

See also: `Stats`

Option `MatrixFunctionStats`. *Display status messages for matrix function evaluations*
Permitted values: `'off'`, `'on'` or `{'auto'}`

Control whether or not to display status messages for the matrix function evaluations. When set to auto, this will be activated when `Stats` is set to `'verbose'`.

See also: `Stats`

Option `Waitbar`. *Show a waitbar to display progress*
Permitted values: `{'off'}`, `'on'`, `'text'` or `'both'`

Set to `'on'` to display the current progress after each time step. Set to `'text'` if you want to display the progess on the MATLAB prompt. Set to `'both'` if you want both kind of displays.

See also: `Stats`

Option `ClearInternalData`. *Clear internal integrator data*
Permitted values: `'off'` or `{'on'}`

Set to `'off'` if you want to keep the integrator's internal data. This data is available in the global `eD` variable.

5.4 Matrix Functions

This section deals with custom evaluation functions for the product of matrix functions with vectors.

`exprb` has two built-in methods to compute these evaluations: directly by diagonalisation and using a Krylov subspace method. In some situations, it can be useful to provide a custom function that does this job. If the Jacobian has a special structure that makes it possible to compute matrix exponentials exactly even in large dimensions, this would be such a case. For this reason, it is possible to hook an arbitrary function into the integration process. See the `MatrixFunctions` option on how to accomplish that.

The two internal functions are implemented the same way as an external method would have to work. They can be used for reference and can be found in the `EXPODE` distribution at `matfun/matfunDirect.m` and `matfun/matfunKrylov.m`. The former one is much easier to understand due to its size and complexity.

It will be neccessary to use the global `eD` varibale which contains information shared between the various `EXPODE` functions. See subsection 5.5.4 for details on this variable. The parts important for this context will be explained when they appear.

All logging output of the function should be printed with the `eD.log.matFunLog` function. This way it can easily be redirected or disabled by the integrator.

The custom function will be called `matfun` in the remainder of this section. It will be invoked in different ways. The function head should look like

```
function [ h, varargout ] = matfun(job, t, y, h, flag, v, reusable, reuse)
    global eD;
    o = eD.int.o;
    funs = eD.functions;
```

The flag variable has two different roles. It can either request a specific action or it can inform the method about the meaning of the currently evaluated matrix function.

There are several different actions that have to be provided:

```
>> matfun([], [], [], [], 'init')
```

should initialize the function globally. It has to state, whether it requires to explicitly evaluate the Jacobian of the right hand side and – if used – of the non-linear part g for semilinear problems (see option `Semilin`). This is done by setting

```
    eD.matfun.needJacExplicit = true;
    eD.matfun.needGJacExplicit = true;
```

or `false` respectively. If the first variable is `true`, then `eD.jac.J` contains the evaluation of the Jacobian at the current step. `eD.jac.Jg` contains the evaluation of g's Jacobian.

Additionally, all statistical data fields have to be initialized. They are used later with the `'statistics'` flag. Example: the direct evaluator sets

```
    eD.stats.matfun.NofDiag = 0;
    eD.stats.matfun.NofMFEv = 0;
```

These two variables count the number of matrix function evaluations and the number of diagonalisations of the Jacobian.

The next flag is the initstep flag:

```
>> matfun(job, t, y, h, 'initstep')
```

When this flag is provided, the function should prepare itself for several matrix function evaluations with the same `t`, `y` and `h`. The `job` variable will be a vector containing a

one for each matrix function which has to be calculated and a zero for all that will not be needed. The indices correspond to the ones of the `eD.int.jobFunctions` cell object. `eD.int.jobFunctions` contains the scalar, vectorized versions of the matrix functions. For `exprb` it is constructed the following way:

```
eD.int.jobFunctions = {
    @phim
    @psim
    @phi3m
    @phi4m
    @(A, h) phim(A, h/2)
    @(A, h) psim(A, h/2)
    @(A, h) exp(h * A);
};
```

This way, `matfun` knows the functions it has to evaluate. The functions `phim`, `psim`, `phi3m` and `phi4m` are part of the `EXP4` package [7] and are included in `EXPODE`.

At the end of the integration process, `matfun` will be called with flag `'cleanup'`:

```
>> matfun([], [], [], [], 'cleanup')
```

Then it should clean up all persistent variables it uses. The `matFun` field of the global `eD` variable will automatically be cleared by the integrator.

The call

```
>> desc = matfun([], [], [], [], 'description')
```

should return a basic information string, naming the method used to calculate the results. The result will be printed at the beginning of the integration phase by

```
>> sprintf('Matrix functions evaluated %s.\n', ...
>>       matfun([], [], [], [], 'description'));
```

The last flag, which is not for any calculation purpose is

```
>> matfun([], [], [], [], 'statistics')
```

Then `matfun` is expected to print some statistics. See the `'init'` flag above for an example on what the direct method logs. If there are no interesting statistical data collected, this flag should be ignored.

All other flags which are provided should be used by `matfun` to recognize the use of the executed job. Then `flag` can be an arbitrary string. In that case, `matfun` needs to actually compute the result of the matrix functions specified by `job` evaluated at `h * J`$_n$ multiplied by the vector `v`.

The `job` variable is a cell-array of vectors. Each vector contains the coefficients for a linear combination of the matrix functions in `eD.int.jobFunctions`. So for vector number `k` in the `job` cell, `matfun` should return the following in `k`th element of the `varargout` variable (written in a suggestive notation):

$$\texttt{varargout\{k\}} = \sum_{\texttt{m}=1}^{\texttt{length(job\{k\})}} \texttt{job\{k\}(m)} * (\texttt{eD.int.jobFunctions\{m\}}(J_n, \texttt{h}) * \texttt{v})$$

There are two additional parameters: `reusable` and `reuse`. The first one indicates whether the matrix function evaluated at the *same* arguments will have to be used again if the current step was rejected by the error estimator. The second one will be set `true` if the previous step was actually rejected and the values calculated there can be reused. In case of an exact evaluation the diagonalization computed the last time can be reused. When approximating the results (e.g. in the Krylov version) it may be required to calculate up to a higher precision. In case of Kylov approximizations, the old Krylov subspaces can be extended. To save the reusable data, use the structure object

```
eD.matFun.save.(flag)
```

This is one of the reasons to provide those `flag`s. The other reason is to save statistical data separately for each matrix function call in the integration scheme. Use

```
eD.stats.matFun.(flag)
```

as storage here.

The implementation of custom matrix functions is quite q complex task. The steps needed were described as simple as possible even though they are still quite hard to understand only by reading this section. Therefore, it is highly recommended to look at the two existing implementations as a reference.

5.5 Writing an expode Integrator

This section contains information for integrator authors. It overviews the EXPODE helper routines and how and where to use them. It is split up into several subsections.

The first subsection describes the fundamentals of the EXPODE package. It names the code files needed, explains the package structure and sets coding style standards to make the EXPODE code as a whole consistent and well readable.

The second subsection deals with the integrators themselves. It states the calling conventions which will make the usage consistent. It will introduce the helper functions that will ease the author's life. These will let him concentrate on the integrator's details instead of having to deal with memory management or syntactic correctness of options. Semantic correctness still has to be considered by the author.

The third subsection deals with integrator options. Here we will explain how to create the option description structure.

The last subsection explains the global `eD` variable. This variable contains all global information shared by `EXPODE` functions.

5.5.1 `expode` basics

The `EXPODE` package consists of the following parts:

- the integrators (`exprb`, etc),
- the integrator info functions (`exprbinfo`, etc),
- the options helper functions (`exprbset`, etc),
- the `initPaths` routine to setup the appropriate paths,
- common helper functions in the `expode` directory,
- matrix-function evaluation helpers in the `matFun` directory,
- some files from the `exp4` package [7] in the `3rdparty/exp4` directory,
- a set of helper functions for each integrator in folders with the names of the integrators and
- some examples in the `examples` directory.

Each integrator has to provide at least the integrator M-file, the options helper function and a function that creates the option description structure. The latter one has to be placed in the helper directory specific for the integrator. As an example see the files `exprb.m`, `exprbset.m` and `exprb/exprbopts.m`. The integrator specific helper directory can also contain additional supportive functions which will be accessible from the integrator.

For readability reasons, the code files should obey the following conventions:

- All M-files should not contain any warnings (and of course no errors) displayed by `MATLAB`'s editor. Usually warnings either mark bad coding style or optimization possibilities. Rarely it is required to disable single warnings like unused function parameters.

- All names of functions, that are direcly visible to the user should be completely lower case. These are mainly the functions in the top level directory except `initPaths` which is used internally to setup paths. Example `exprbinfo`.
- All names of variables and internal functions should begin with a lower case letter. Each new word in the names should start with an upper case letter. Example: `checkValidOptions`, `quitIntegrator`
- Each M-file should contain a help text in its header. Longer functions that do non-trivial work should have comments in the code as well.
- Indentation is done via four spaces, no tabs.
- All commas should be followed by a space. Insert spaces after an open squared bracket or brace and before a closing one. Empty brackets can be written without spaces. Use no spaces around parentheses. Equal signs should be surrounded by spaces. Example:

```
>> result = str2func([ 'exprb', 'opts', [] ]);
```

- All lines except the ones that are part of control strucures like `if`s or `while`s should be terminated by a semicolon.
- Lines should be maximally around eighty characters log. „Around“ means that a line should be split up after the first word that exeeds the eightieth column. `MATLAB` can highlight this column. This option can be activated in the preferences dialog under „Editor/Debugger“ → „Display“. The `MATLAB` syntax supports split lines via three dots at their ends. Strings can be split up, too:

```
>> text = [ 'This is a very very very ' ...
>>          'very long text' ];
```

  Note the space after the last „very“ in the first line, because the variable `text` will not contain a newline chacter where the text is split. The remainder of the split lines has to be indented eight more spaces than its parent.
- One-line-`if` constructions should be avoided as well as nested expressions. Generally it is dicouraged to put more than one command in a single line. Do not use constructions like:

```
>> if condition, expression, end
```

  or

```
>> var1 = func1(func2(var2, func3(var3)), func4(var4));
```

  The former should be written in three lines, the latter at least expanded into

```
>> temp1 = func2(var2, func3(var3));
>> temp2 = func4(var4);
>> var1 = func1(temp1, temp2);
```

- Function code should always be indented one level.
- Nested functions should not be indented. A file `majorFunction.m` should look like:

```
function majorFunction
    majorFunctionCode

function nestedFunction
    nestedFunctionCode
```

- There is *only one* global variable: `eD`. `EXPODE` functions should never define additional ones. Generally, it is discouraged to use this variable. Use it only if really necessary or the information stored are of common interest doublessly.

5.5.2 Integrators

This section describes how new `EXPODE` integrators should work. The most important helper functions will be explained. In distinction to the existing `exprb` integrator we will use a fictitious integrator called `expnew` here.

`EXPODE` integrators should be callable the same way `MATLAB` integrators are. At least the following call should be possible:

```
>> [t, y] = expnew(@ode, {@jac}, {tspan, y0}, {opts}, {varargin});
```

There are several supportive methods to help integrator authors. The code snipplets presented are quite minimal. They are only presented to show the general structure. To get a more detailed insight, use the `exprb` source code as a reference. We start with some initialisation routines. First we have to initialize the `EXPODE` internal paths.

```
intName = 'expnew';
initPaths(intName);
```

Then initialize the integrator.

```
% Initialize the integrator
initIntegrator;
if quitIntegrator
```

```
    % quit, if initIntegrator says that we should
    return;
end
```

Now we have to handle the integrator arguments:

```
[ ode, odeName ] = getFunctionFromArg(ode);
```

retrieves a function handle from the `ode` argument. It can handle inlines, function names (given as strings) or function handles and will always return a function handle.

Next we handle the `jac` argument:

```
[ jac, jacName ] = getFunctionFromArg(jac);
if isempty(jacName)
    % We do not have a jac Argument, shift the later arguments
    % (varargin = {opts, varargin{:}}, opts = y0, etc.)
end
```

The other arguments have to be checked as well, but there are no helpers available to support this. If we need to retrieve the `tspan`, y_0 and `opts` information from the `ode` function, do this with

```
[tspan, y0, opts] = ode ([], [], 'init', varargin);
```

We now can get the integration interval from the `tspan` variable:

```
t0 = tspan(1);
T = tspan(end);
% Globally announce ODE Data
setODEData(t0, y0, T, options);
```

This will save the initial data in the global `eD` variable. See section 5.5.4. Next, we can evaluate the options:

```
[ o, nv ] = evaluateOptions(opts);
```

This will assure that all options are set, even the ones not specified by the user. The latter ones will be set to their default values. See section 5.5.3 for details on options. Options of type `boolean` will be evaluated to `0` for `false` and `1` for `true`. If there exists an option `Waitbar` of type `boolean`, you can use

```
if o.Waitbar
    % Waitbar was set 'on'
else
    % Waitbar was set 'off'
end
```

to handle these options. The same works for `list` type options. You can use the `nv` variable that contains the numeric values corresponding to the list's entries here. If `Waitbar` is a `list` type option with the allowed values `'on'`, `'off'`, `'maybe'`, you can handle its setting by

```
switch o.Waitbar
    case nv.Waitbar.on
       % Waitbar was set 'on'
    case nv.Waitbar.off
       % Waitbar was set 'off'
    case nv.Waitbar.maybe
       % Waitbar was set 'maybe'
end
```

Note that these comparisons are integer comparisons and not string comparisons. This will make it much faster when used inside the integration process where it needs to be done on every iteration.

There is a helper routine for options which can either be a function handle or indicate to use the `ode` function with a certain flag.

```
jac = getFunctionFromOpt(o.Jacobian, ode, haveVarargin, type);
```

The `haveVarargin` variable needs to be a boolean, and be set to `true` if the `varargin` argument was used at the integrator call. The type variable specifies the flag to set when using the `ode` function. `getFunctionFromOpt` will either return a function handle or `0` if the option was set to `'off'`. The following type/flag combinations are available: `-1`/`[]`, `0`/`'jacobian'`, `1`/`'jacobian_v'`, `2`/`'df_dt'`, `3`/`'gfun'`, `4`/`'dg_dy'` and `5`/`'dg_dy_v'`.

All functions which need to be evaluated to solve the differential equation have to be saved in the global `eD` variable as function handles. These are usually the evaluation of the right hand side and the Jacobian, but can possibly include more, see the `JacobianV` option for `exprb`. See `eD.functions` in section 5.5.4.

Next, we need to set the job functions used by the integrator. These are the matrix functions which need to be evaluated and then multiplied by vectors involved in the integrator scheme. In the `eD.int.jobFunctions` save scalar, vectorized versions of the matrix functions. A set of typically used funtions is available in the `3rdparty/exp4` directory of the `EXPODE` package. The order of these functions is important, see section 5.4 on matrix functions for details.

Example: Assuming `expnew` will only need to evaluate the entire function φ_1 and φ_2, we can set

```
eD.int.jobFunctions = {
    @phim % evaluates φ1
    @psim % evaluates φ2
};
```

To announce the matrix function evaluator and the `varargin` needed for the `ode` evaluation we finally need to set

```
matFunV = @matFunDirect;
funs.matFunV = matFunV;
funs.varargin = varargin;
```

An important part for user feedback is logging. Therefore `EXPODE` provides a number of different logging levels (see the appropriate options of `exprb`). They need to be set with

```
setLoggingFunctions(verbose, status, jacLog, stepLog, matFunLog, ...
        warn, err);
```

each of the arguments above is either a function handle to `fprintf` (set with `status = @fprintf`) or to `warning`, `error` or `nullfunction`. The latter one ignores all input, so all messages sent to this function will be silenced. `verbose` and `status` are generic logging functions, `jacLog` will be used when dealing with the Jacobian, `stepLog` when dealing with step size related information and `matFunLog` will be used by the matrix function evaluators (see section 5.4). With the last two arguments it is possible – though discouraged – to supress warnings and errors. This can be useful in debugging and testing cases. Use

```
eD.log.status('This is a status message');
```

to print a status message.

When all preliminary option handling is done, we can prepare to start the integration process. The matrix function evaluator needs to be initialized:

```
matFunV([], [], [], [], 'init');
```

The waitbar needs to be initialized as well – if it is used:

```
progressbar([t0 T], y0, 'init', intName, style);
```

And finally, we prepare the output matrices with

```
initResults(h, 1, false);
```

Now we can start the integration process. Our current time and phase space variables will be called `t` and `y`. If you need to evaluate the `ode` function or the Jacobian, use the following functions:

```
evaluateF(t, y);
evaluateJac(t, y);
```

To prepare the matrix function evaluation call

```
matFunV(job, t, y, h, 'initstep', [], reuseable, reuse);
```

and see 5.4 for details. Now the integrator has to do its actual work. After a successful step, call

```
addStep(tOut, yOut);
```

with all new time steps calculated in this integration step. Note that this can be more than one, if using a *dense output* technique. In that case `tOut` will be a vector, and `yOut` will be a matrix. If the integration process took an unsuccessful step, `addStep` should still be called to update integration statistics on the failed step. Then use `tOut = []; yOut = [];`.

If you did a successful step and use the `EXPODE progressbar` function to display a waitbar, you have to update its data now as well.

```
progressbar(t, y, '');
```

where `t` and `y` are the time and the numerical solution at the next full time step. Do not provide the data calculated by a *dense output* function here.

After the integration process is finished, we need to clean up a little. Shut down the waitbar with

```
progressbar(T, eD.ydata(end,:), 'done');
```

`eD.ydata` now contains the full integration data generated by the integrator thanks to the `addStep` function. `T` should be the end time of the integration. All data not interesting for the user should be purged from the `eD` variable.

Now we have seen a rough outline of the technical part of the integration process. To get a better understanding one should take a closer look at the `exprb` code.

5.5.3 Options

This section deals with integrator options. EXPODE has advanced option handling subroutines which allow automatic checking of options set by the user. First, each integrator needs an options helper function as a user interface. This function should be called the same as the integrator with a „set“ suffix. The exprb integrator is contained in the exprb.m M-file and its options helper is named „exprbset.m“ for instance. The options helper should simply be a wrapper around the expset function which only adds the integrator name as the first argument:

```
function opt = expnewset(varargin)
    intname = 'expnew';
    initPaths(intname);

    if isempty(varargin)
       help expnewset;
       return;
    end

    opt = expset(intname, varargin{:});
```

The initPaths routine is needed to initialize the EXPODE internal paths as seen in the previous section.

This expset method needs a so-called *option description object*. This has to be provided by a function called expnewopts and has to be located in the expnew directory. This object contains all information about the options for expnew. It is a MATLAB struct object and has to have the following strutcure:

```
>> expnewset
ans =
         integrator:  'expnew'
     integratorname:  'new great exponential integrator'
     integratordesc:  [1x223 char]
              usage:  [1x522 char]
               opts:  [1x1 struct]
```

The first four fields are all strings, the first one containing the integrator name, the second a longer name of the integration method, the third a short description, and the last one the help text for the integrator, displayed when calling the integrator without arguments. This is the reason the initIntegrator method can set the quitIntegrator variable to true. It checks if no arguments were provided and prints this help message.

The last entry, opts, is again a structure object. It contains the actual options. An option is a field of opts and can be created via

```
opts.NewOption = createOptDesc( ...
    'short description', ...
    'type', ...
    'default', ...
    { 'value1', 'value2' }, { ...
    'Very long and detailed description for the very nice'
    'and new option NewOption.'
    }, {
    'OptionA', 'OptionB'
});
```

The first argument is a short description that will be displayed when using expnewinfo, see below. The next one contains the option's type. Available option types are listed in section 5.2. Next, we have the default value for the option. This can be a numeric value or string, depending on the option type. In case of boolean or list it can either be the numeric or string value of the option. Argument four of the createOptDesc function contains the allowed values for list. For all other types, provide the empty cell object {}. The next argument contains the long and detailed description for the option. This can either be a string or a cell of strings as used in the example. The last argument contains a list of referenced options. This list will be printed in the form See also: OptionA, OptionB when calling expnewinfo NewOption.

For reference and a lot of examples look at the file exprb/exprbopts.m in the EXPODE package.

Another function that needs to be provided for each integrator is the info command. This is a wrapper around optInfo similar to the options setter. It will print information on the integrator. The information will be extracted from the option description object described above. Here is its basic structure:

```
function expnewinfo(optName)
    if nargin < 1
       optname = '';
    end
    optInfo('expnew', optName);
```

5.5.4 The global eD Variable

There is only one global variable used throughout EXPODE (see the coding conventions above), namely eD. It is a structure object containing several different types of information. After an exprb call it contains the following fields if the ClearInternalData is set to 'off':

```
>> eD
eD =
  integrator:  'exprb'
          ODE:  [1x1 struct]
          int:  [1x1 struct]
          log:  [1x1 struct]
    functions:  [1x1 struct]
        stats:  [1x1 struct]
       matFun:  [1x1 struct]
        tdata:  [451x1 double]
        ydata:  [451x19 double]
        evals:  [1x1 struct]
```

The first field is `integrator` and it holds the name of the integrator. The second, `ODE`, contains information about the differential equation and the solver request:

```
>> eD.ODE
ans =
          t0:  0
           T:  45
    duration:  45
        tDir:  1
          y0:  [19x1 double]
        ylen:  19
   transpose:  1
     options:  [1x1 struct]
```

It contains the endpoints of the integration interval, `t0` and `T`, their difference `duration`, the integration direction (`tDir`) – with values `1` for ascending or `-1` for descending time steps, the initial value `y0` and its length `ylen` as well as the `options` provided at the integrator's call. `transpose` is a boolean that indicates, whether the solution vectors are rows or columns. In the latter case, they have to be transposed before being stored in the result matrix.

The `int` field of `eD` contains three entries: `jobFunctions` contains function handles to scalar, vectorized versions of the matrix functions involved in the integrator's scheme. These will be used by matrix function evaluators, see sections 5.4 and and 5.5.2. Additionally, there exist the fields `o` containing a version of the options structure but with all non-set options evaluated to their default values and all `list`-type options evaluated to the index of the selected value in the list. `boolean`-type options are also converted to `1` for value `true` and `0` for `false`. The last field, `nv`, contains the numeric values of the list-type options, see the previous section.

Next in the eD structure we have the log field containing logging functions for several purposes. They all have the same syntax as the fprintf command, but may be directed to a file or simply ignored depending on what the user wants to see.

```
>> eD.log
ans =
       verbose:  @fprintf
           log:  [function_handle]
        status:  @fprintf
        jacLog:  @nullfunction
       stepLog:  @fprintf
     matFunLog:  @nullfunction
       warning:  @warning
         error:  @error
```

The stats field in eD contains statistical data collected over the whole integration process. It has a subfield matFun which contains data gathered by the matrix function evaluators.

The fields tdata and ydata contain the solution data. This is what the integrator returns to the user.

The last field, evals contains evaluations of the right hand side of the differential equation, its Jacobian and the Jacobian of the non-linear part – if Semilin is 'on'.

Most of this data will be cleaned before the integration ends. If you want to preserve the data for testing, set the option ClearInternalData to 'off'.

Literaturverzeichnis

[1] Håvard Berland, Bård Skaflestad, and Will M. Wright. Expint—a matlab package for exponential integrators. *ACM Trans. Math. Softw.*, 33(1):4, 2007.

[2] M. Hochbruck, Mathematisches Instititut, Heinrich-Heine-Universität Düsseldorf. Vorlesungsskript zur Numerik, 2005–2009.

[3] M. Hochbruck and Ch. Lubich. On Krylov subspace approximations to the matrix exponential operator. 34(5):1911–1925, 1997.

[4] A. Pazy. *Semigroups of Linear Operators and Applications to Partial Differential Equations*. Springer, 1983.

[5] J. Schweitzer. *Numerical Simulation of Relativistic Laser-Plasma Interaction*. PhD thesis, Heinrich-Heine Universität Düsseldorf, 2008.

[6] M. Hochbruck und A. Ostermann und J. Schweitzer. Exponential Rosenbrock-type methods. *SIAM J. Numer. Anal.*, 47:786–803, 2009.

[7] M. Hochbruck und Ch. Lubich und H. Selhofer. `exp4` – exponential integrator of order 4. `http://na.uni-tuebingen.de/projects.shtml`, 1998.

[8] M. Hochbruck und Ch. Lubich und H. Selhofer. Exponential integrators for large systems of differential equations. *SIAM J. Sci. Comp.*, 19:1552–1574, 1998.

[9] E. Hairer und G. Wanner. *Solving Ordinary Differential Equations II. Stiff and Differential-Algebraic Problems*. Vol. 14 of Springer Series in Computational Mathematics. Springer, 2nd edition, 1996.

[10] N. Dunford und J.T. Schwartz. *Linear Operators. Part I: General Theory. Reprint of the 1958 original*. John Wiley & Sons, New York, 1988.

[11] K.-J. Engel und R. Nagel. *One-Parameter Semigroups for Linear Evolution Equations*. Springer, 2000.

Erklärung

Hiermit versichere ich, die vorliegende Master-Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Düsseldorf, im 10.06.2009

(Georg Jansing)