Technische Universität Hamburg

Institut für Elektrische Energiesysteme und Automation
Prof. Dr.-Ing. T. T. Do

Parameterabhängige Schrittweitensteuerung für eine HIL-Simulation zur Prüfung automobiler Steuergeräte

MASTERARBEIT

Julius Harms B. Sc.

1. Prüfer: Prof. Dr.-Ing. G. Ackermann

2. Prüfer: Prof. Dr.-Ing. F. Thielecke

Betreuer: M.Sc. D. Kähler

11. Dezember 2018

Erkärung

Ich, Julius Harms (Student des internationalen Masters Mechatronics an der Technischen Universität Hamburg, Matrikelnummer 21267847), versichere an Eides Statt, dass ich die vorliegende Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe. Die Arbeit wurde in dieser oder ähnlicher Form noch keiner Prüfungskommission vorgelegt.

Hamburg, den 11. Dezember 2018

Julius Harms

Parameterabhängige Schrittweitensteuerung für eine HiL-Simulation zur Prüfung automobiler Steuergeräte

Parameter-dependent step size control for a HiL simulation for testing automotive control units

Aufgabenstellung

Durch die zunehmende Elektrifizierung des Automobils kommen immer mehr Steuergeräte (ECU) zum Einsatz. Diese Steuern verschiedene elektrische Verbraucher. Aus Komfort- oder Sicherheitsgründen, werden diese immer aufwendiger gesteuert, bzw. geregelt. Um die komplexen Steueralgorithmen der ECUs einfacher testen zu können, kommen vermehrt Hardware-in-the-loop Simulationen zum Einsatz, bei welchen der reale Verbraucher durch eine Echtzeitsimulation mit entsprechender Leistungselektronik ersetzt wird. Viele elektrische Verbraucher sind kleine Motoren, welche zum Teil mit hohen Drehzahlen betrieben werden. Außerdem werden einige Verbraucher durch Pulsweitenmodulation (PWM) angesteuert. Daraus resultierend ergeben sich hohe zeitliche Anforderungen an die Simulation, welche teilweise mit Zeitschrittweiten von unter 1 µs arbeiten muss. In diesem Zusammenhang wurde eine heterogene parallele Simulationsarchitektur entwickelt. Die von der ECU abgegebene PWM Spannung wird in ein nieder- und ein hochfrequentes Signal aufgeteilt. Der niederfrequente Anteil wird als Eingang für eine langsame Simulation mit komplexen Modellen genutzt und der hochfrequente Anteil für eine schnelle Simulation mit stark vereinfachten Modellen. Die Schrittweiten der beiden Simulationspfade können sich dabei um mehrere Größenordnungen unterscheiden. In der aktuellen Ausführung wird in beiden Pfaden die kleinstmögliche Schrittweite unter Berücksichtigung der Hardware verwendet.

In dieser Arbeit soll ein Verfahren zur automatischen parameterabhängigen Schrittweitenoptimierung erarbeitet werden. Dazu ist herauszuarbeiten, in welcher Weise die unterschiedlichen Parameter Einfluss auf das Verhalten des jeweiligen Simulationsmodells haben. Die Schrittweite soll automatisiert nach Eingabe der Modellparameter angepasst werden. Dieses Verfahren soll allgemein auf alle Modelle anwendbar sein, somit soll in dieser Arbeit auch die Umsetzung des Simulationsmodells im C++ Code standardisiert werden, damit dieselbe Optimierungsroutine auf alle Modelle gleichermaßen anwendbar ist. Damit einhergehend soll auch die Stabilität der Simulation untersucht werden. Aufgrund der Einschränkungen durch die Echtzeitfähigkeit und der Hardware kann unter Umständen keine geeignete Schrittweite gewählt werden. Ist dies der Fall kann der Start der Simulation verhindert werden. Weiterhin gilt es herauszufinden, ob eine Anpassung während der Laufzeit sinnvoll ist, beispielsweise bei dem Wechsel zwischen verschiedenen Betriebszuständen oder bei Veränderung von Parametern während der Simulation. Zu beachten sind auch Effekte, welche nicht von den Modellparametern abhängig sind.

Inhaltsverzeichnis

Symbolverzeichnis

Symbol	Bezeichnung
N	Stufen des Runge-Kutta Verfahrens
$\mathcal{O}$	Landau-Symbol
P	Parameter
u	Eingangsvariable
t	Zeit
y	Zustandsvariable
y_0	Anfangswert der Zustandsvariable
$h_{\mathbf{max}}$	Stabilitätsgrenze / größte stabile Schrittweite
h	Schrittweite
ω	Korrekturfaktor für die Stabilitätsgrenze
$F(y,u,t)$	Differentialgleichungssystem
$f(y,u,t)$	Differentialgleichungen
A	Systemmatrix
k	Stufe im Runge-Kutta Verfahren
z	Produkt aus Schrittweite h und Eigenwert λ
$R(z)$	Stabilitätsfunktion

Abkürzungen

AWP	Anfangswertproblem
BE-Verfahren	implizite Eulerverfahren - *backward Euler method*
DGL	Differentialgleichungen
FE-Verfahren	explizite Eulerverfahren - *forward Euler method*
HiL	Hardware in the loop
ODE	Gewöhnliche DGL - *Ordinary Differential Equation*
PDE	Partielle DGL - *Partial Differential Equation*
RK-Verfahren	Runge-Kutta Verfahren

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Quellcodes

1 Einleitung

Hardware in the loop (HiL) Simulationen bieten einen umfassenden und kostengünstigen Ansatz für den Systemtest von eingebetteten Systemen. Insbesondere in der Entwicklungsphase von komplexen Steuergeräten sind die Simulationsergebnisse daher von großer Bedeutung und Konsequenz. Aus diesem Grund werden hohe Anforderungen an die Simulationsgenauigkeit, insbesondere in sicherheitskritischen Systemen gestellt. Zur Simulationsoptimierung bieten gängige Simulationsprogramme (MatLab, Modelica, etc.) eine Vielzahl von Lösungsverfahren, mitunter Steuerungs-Algorithmen zur Verbesserung der Näherungsgenauigkeit. Die verschiedenen Verfahren sind jedoch hauptsächlich für reine Software-Simulationen entworfen und die Implementierungen für Echtzeitsimulationen erfüllen nicht immer die Zeitanforderungen der HiL-Umgebung. Die meisten Ansätze für Software-Simulationen optimieren die Genauigkeit im Verhältnis zur gesamten Simulationszeit und scheitern bei HiL-Simulationen in sehr kleinem Zeitbereich an der Einhaltung der Echtzeitfähigkeit. Um auf diesen systematischen Mangel zielführend eingehen zu können, ist eine Betrachtung der spezifischen Optimierungsmöglichkeiten von Lösungsverfahren in HiL-Umgebungen sinnvoll.

Durch die Echtzeitvorgabe erreicht eine HiL-Simulation von schnellen Systemen oder hochfrequenten Signalen früh die Grenze der Lösungsstabilität. Dies ist insbesondere bei CPU-basierten HiL-Umgebungen der Fall, welche aufgrund der hohen Latenz nur größenbegrenzte Schrittweiten realisieren können. Eine Möglichkeit zur Verbesserung des Simulationsergebnisses bietet daher z. B. die Wahl einer schnelleren, FPGA-basierten Simulation. Dennoch begünstigen die sehr hohe Verfügbarkeit von CPU-basierten Simulationen und die größere Flexibilität in der Gestaltung des Algorithmus deren bevorzugte Verwendung.

In der folgenden Arbeit werden verschiedene Optimierungsansätze für CPU-basierte HiL-Simulationen betrachtet. Das Ziel ist die Entwicklung eines in Bezug auf die Echtzeitfähigkeit optimierten Lösungsalgorithmus mit automatischer Schrittweitenwahl. Die Optimierung erfolgt modellorientiert und richtet sich nach den individuellen Modelleigenschaften. Das Optimierungsverfahren evaluiert das Modell bei jeder Änderung der Modellparameter und ist daher parameterabhängig gestaltet. Die Betrachtung des Zusammenhangs zwischen Modelleigenschaften und Lösungsstabilität ermöglicht dem Lösungsalgorithmus ein stabiles Lösungsverfahren und eine echtzeitfähige Schrittweite zu wählen. Dafür werden außerdem mögliche Fehlerquellen der Simulation miteinbezogen. Auch eine laufende Anpassung für verschiedene Betriebszustände wird betrachtet. Für die Modelldefinition gilt es, eine allgemeine Darstellungsform zu finden, mit welcher sich Modelle von unterschiedlicher Komplexität abbilden lassen. Für die Validierung des entwickelten Algorithmus wird die Umsetzung in C++ vorgestellt. Als Anwendungsbeispiel dient eine parallele HiL-Simulation zur Prüfung automobiler Steuergeräte. Die Simulation besteht aus einer parallelen Ausführung von Hardware- und Software-Simulation mit heterogenen Aufgaben in unterschiedlichen Zeitbereichen.

2 Numerische Integrationsalgorithmen für HiL-Simulationen

HiL-Simulationen verlangen besondere Berücksichtigung bei der Gestaltung der Lösungsverfahren und stellen etliche Anforderungen an die Simulationsumgebung. In diesem Kapitel werden HiL-Simulationen zunächst allgemein vorgestellt und auf die mit diesem Anwendungsfall verbunden Besonderheiten hingewiesen. Anschließend werden die Grundlagen der numerischen Simulation und deren Umsetzung in Computeralgorithmen dargestellt. Hierfür erfolgt zunächst eine kurze Betrachtung der für Simulationsmodelle relevanten Typen von Differentialgleichungen (DGL). Anschließend werden beispielhaft verschiedene numerische Integrationsverfahren vorgestellt, welche die Unterschiede zwischen Integrationsverfahren im Allgemeinen verdeutlichen. Die Begriffe Steifigkeit und Stabilität werden erläutert, welche für die Modell- und Verfahrensanalyse eine wichtige Rolle spielen. Eine kurze Betrachtung der Fehlerquellen von numerischen Integrationsalgorithmen ermöglicht die spätere Einschätzung der Verfahrensgenauigkeit. Durch die Vorstellung der Algorithmuskomplexität wird ein Bewertungsverfahren für die Einordnung der Effizienz von Algorithmen vorgestellt.

2.1 Hardware in the Loop Simulationen

Hardware in the Loop (HiL) Simulation bezeichnet ein Verfahren, bei welchem ein eingebettetes System mit seinen Ein- und Ausgängen an ein HiL-Simulator angeschlossen wird. Der HiL-Simulator bildet durch die numerische Simulation eines gegebenen Modells die realen Bedingungen für das eingebettete System. Diese Methode bietet durch seine Absicherung und Abgrenzung von der realen Umgebung, große Vorteile in der Entwicklung von eingebetteten Systemen, erfordert jedoch die Einhaltung zusätzlicher Anforderungen an die Simulation.

Abbildung 1: *Konzeptskizze einer HiL Simulation*

Ein HiL-Simulator besteht meist aus Hardware- und Softwarekomponenten, welche im Ablauf einer Simulation parallel ausgeführte Aufgaben übernehmen. Im Rahmen dieser Arbeit werden Simulatoren betrachtet, die ein gegebenes Modell CPU-basiert numerisch Lösen. Die Schnittstelle zwischen softwarebasierter Simulation und eingebettetem System

bildet eine Leistungselektronik als Hardwarekomponente des Simulators (s. Abb. 1). Für die Berechnung eines Simulationsschritts, liest die Hardwarekomponente die aktuellen Ausgangssignale des eingebetteten Systems ein und übergibt diese an die Simulationssoftware. Die Messdaten der Ausgangssignale dienen als Eingangswerte für die Simulation. Aus den Eingangswerten und den simulationsinternen Systemvariablen bestimmt die Simulationssoftware den nächsten diskreten Simulationsschritt. Die berechneten Werte werden an die Leistungselektronik übergeben, der die Eingangssignale für das eingebettete Systems steuert. Da der analoge Ausgang der Hardwarekomponente diskrete Simulationswerte erhält, wird meist eine Abtast-halte-Schaltung verwendet, um das Ausgangssignal zwischen den diskreten Stellen auf dem zuletzt berechneten Funktionswert zu halten. Abbildung 2 zeigt die Momentanwertabtastung eines Beispielsignals. Zu sehen sind die

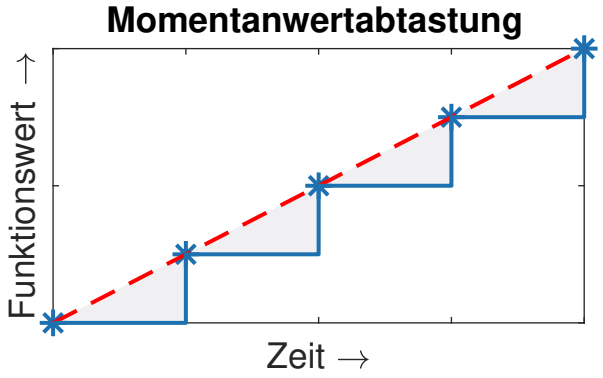

Abbildung 2: *Vergleich von Momentanwertabtastung (blau) eines Ausgangsignals mit dem eigentlichen Funktionsverlauf (rot).*

diskreten Simulationswerte als Punkte auf dem rot gestrichelten Funktionsverlauf. Der Funktionsverlauf ist nur zur Veranschaulichung eingezeichnet und lässt sich aus der Interpolation der diskreten Stellen bestimmen. Durch die Abtastung wird das dargestellte Stufenbild erzeugt, welches das Ausgangssignal der Hardware beschreibt. Die Fläche zwischen dem interpolierten Funktionsverlauf und der Momentanwertabtastung stellt daher die Abweichung von analogem Ausgangssignal zum theoretischen Funktionsverlauf dar.

Durch die Parallelisierung von eingebettetem System und Simulator ist die Echtzeitfähigkeit eine wichtige Anforderung an die HiL-Simulation. Das bedeutet im Falle der numerischen, zeitdiskreten Simulation, dass die Latenz eines Simulationsschrittes mit der Schrittweite h nicht länger sein darf, als die Schrittweite h selber. Die Latenz setzt sich aus der reinen Berechnungszeit des Schrittes und allen zusätzlichen Zeiten zum Einlesen und Schreiben der benötigten Informationen (im Weiteren kurz Restaufgaben) zusammen.

2.2 Differentialgleichungen von Simulationsmodellen

Simulationsmodelle aus ingenieurswissenschaftlichen Anwendungen, liegen meist als Kombination aus algebraischer Gleichung, gewöhnlicher Differentialgleichung und parti-

eller Differentialgleichung vor [20]. Grundlegend für eine Modellbeschreibung werden im Folgenden daher beide Arten von DGL sowie die Eigenschaft der Linearität vorgestellt.

2.2.1 Gewöhliche Differentialgleichungen

Eine Gewöhnliche DGL - *Ordinary Differential Equation* (ODE) ist eine Gleichung, deren gesuchte Funktion von nur einer Veränderlichen abhängt. Mit der gesuchten Funktion $y(t)$, auch als abhängige Variable bezeichnet, der unabhängigen Variable t und einer gegebenen Konstante μ ist im Folgenden ein Beispiel einer ODE gegeben [20]:

$$\frac{\mathrm{d}y}{\mathrm{d}t} = y'(t) = \mu \cdot y \tag{1}$$

Mit einem bekannten Anfangswert t_0, kann folgendes allgemeines Anfangswertproblem (AWP) formuliert werden:

$$\begin{aligned} y'(t) &= f(y,t) \\ y(t_0) &= y_0 \end{aligned} \tag{2}$$

Die exakte analytische Lösung des AWPs bezogen auf Gl. (1) ergibt sich mit:

$$y(t) = y_0 \cdot \mathrm{e}^{\mu_{max}(t-t_0)} \tag{3}$$

Die konstanten Faktoren einer DGL werden im Rahmen dieser Arbeit als Modellparameter bezeichnet. Die Parameter haben einen entscheidenden Einfluss auf das Funktionsverhalten. An dem Beispiel aus Gl. (1) und (3) wird deutlich, dass bei $\mu_{max} < 0$ eine exponentielle Annäherung vorliegt und die Funktion mit $\mu_{max} > 0$ ein exponentielles Wachstum beschreibt. [20]

Ist die gesuchte Funktion $y(t)$ eine Vektorfunktion n-ter Ordnung, spricht man von einem Differentialgleichungssystem (DGL-System) der n-ten Ordnung.

2.2.2 Partielle Differentialgleichungen

Eine Partielle DGL - *Partial Differential Equation* (PDE) besitzt im Gegensatz zur ODE partielle Ableitungen der abhängigen Variable. Mit den unabhängigen Variablen t und z, sowie der abhängigen Variable $u(t,z)$, bildet die Burger's Gleichung [1] ein Beispiel für eine (nichtlineare) PDE:

$$\frac{\partial u}{\partial t} = -u\frac{\partial u}{\partial z} + \mu\frac{\partial^2 u}{\partial z^2} \tag{4}$$

2.2.3 Linearität

Im Umgang mit DGL, sei es mit ODE oder PDE, kann zwischen linearen und nichtlinearen Gleichungen unterschieden werden. Eine lineare DGL zeichnet sich durch einen linearen Differentialoperator aus, d.h. die gesuchte Funktion und ihre Ableitungen werden ausschließlich mit bekannten Funktionen oder Zahlen und nicht mit sich selbst multipliziert [16]. Wie in den folgenden Abschnitten verdeutlicht wird, fällt der Umgang mit linearen DGL deutlich leichter als mit nichtlinearen. Dies wird auch bei Betrachtung der Zustandsraumdarstellung für lineare ODE deutlich. Selbst komplexe DGL-Systeme höherer Ordnung mit mehreren Ein- und Ausgangsgrößen lassen sich durch die Zustandsraumdarstellung vereinfacht darstellen. DGL-Systeme werden in der Zustandsraumdarstellung durch eine Matrix-DGL erster Ordnung dargestellt. Diese ist durch zwei Gleichungen gegeben:

$$\begin{aligned} y'(t) &= Ay(t) + Bu(t) \\ y_{out}(t) &= Cy(t) + Du(t) \end{aligned} \tag{5}$$

Die Gl. (5.a) ist die Systemgleichung mit dem Zustandsvektor y, dem Eingangsvektor u, der Systemmatrix A und der Eingangsmatrix B. Gl. (5.b) wird Ausgangsgleichung genannt. Hier ist y_{out} der Ausgangsvektor, C die Ausgangsmatrix und D die Durchgangsmatrix. Der Zustandsvektor y besteht aus den zur Systembeschreibung minimal notwendigen Zustandsvariablen. Für eine lineare zeitinvariante DGL kann die Systemmatrix A zur Systemanalyse verwendet werden. Durch die Analyse der Eigenwerte von A lassen sich Aussagen über die Steifigkeit und Stabilität der Gleichungen treffen, welche direkte Rückschlüsse auf benötigte Schrittweiten und Lösungsverfahren geben.

Nichtlineare DGL-Systeme lassen sich nicht in der Zustandsraumdarstellung beschreiben und werden daher meist in der Funktionsdarstellung als Vektorfunktion abgebildet:

$$y' = f(y,u,t) \tag{6}$$

Mit dem Zustandsvektor $y = [y_1(t),\ldots,y_k(t)]^\top$ für k abhängige Variablen und dem Vektor $u = [u_1(t),\ldots,u_m(t)]$ für m Eingangsgrößen bildet F eine Vektorfunktion der Größe k, mit $F(y,u,t) = [f_1(t,y),\ldots,f_k(t,y)]^\top$. Diese Darstellungsform ist im Gegensatz zur Zustandsraumdarstellung linearer Systeme deutlich unhandlicher, da z. B. für die Analyse nicht direkt auf die Eigenwerte einer Systemmatrix A zurück gegriffen werden kann. [16]

Es ist möglich nichtlineare DGL an einem Entwicklungspunkt zu linearisieren. In näherer Umgebung des Entwicklungspunktes beschreibt das lineare System annähernd das Verhalten des nichtlinearen Systems, sodass die Analyseverfahren für lineare Gleichungssysteme auch auf das linearisierte Problem angewendet werden können. Der gesamte Prozess der Linearisierung kann dabei in die folgenden drei Schritte aufgeteilt werden:

1. Linearisierung
2. Einfrieren der Koeffizienten
3. Diagonalisierung

Die Linearisierung des Systems erfolgt durch eine lineare Taylorentwicklung an einem Entwicklungspunkt. Hierfür kann die Jacobi-Matrix verwendet werden, welche alle partiellen Ableitungen eines DGL Systems enthält. Für ein gegebenes DGL System $F(t,y,u)$ mit k Zustandsvariablen y und dem Eingangsvektor u ist die Jacobi-Matrix J_F für y gegeben mit:

$$J_F(t,y,u) = \begin{pmatrix} \frac{\partial f_1}{\partial y_1} & \frac{\partial f_1}{\partial y_2} & \cdots & \frac{\partial f_1}{\partial y_k} \\ \frac{\partial f_2}{\partial y_1} & \frac{\partial f_2}{\partial y_2} & \cdots & \frac{\partial f_2}{\partial y_k} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_k}{\partial y_1} & \frac{\partial f_k}{\partial y_2} & \cdots & \frac{\partial f_k}{\partial y_k} \end{pmatrix} \tag{7}$$

Die partiellen Ableitungen sind lediglich für die Zustandsvariablen y gebildet. Da die Jacobi-Matrix eines nichtlinearen Systems weiterhin von der Zustandsvariablen y, sowie der Eingangsgröße u abhängig sein kann, erfolgt im nächsten Schritt das Einfrieren der Koeffizienten. Hierfür wird ein Wertepaar für y und u als Entwicklungspunkt in das linearisierte Problem eingesetzt. Durch das Einfrieren wird die lokale Betrachtung eines linearen Problems mit konstanten Koeffizienten ermöglicht [13].

$$y' = F(t,y,u) \tag{8}$$

$$\downarrow \quad \text{Linearisierung}$$

$$y' = J_F(t,y,u) \cdot y = A(t,y,u) \cdot y \tag{9}$$

$$\downarrow \quad \text{Einfrieren der Koeffizienten}$$

$$y' = A \cdot y \tag{10}$$

Das linearisierte Problem kann nun wie ein lineares Problem behandelt werden, wobei die eingefrorene Jacobi-Matrix die Systemmatrix A des lokalen Problems darstellt. Im nächsten Schritt folgt durch die Diagonalisierung von A die Bestimmung der Eigenwerte λ. [13]

$$y' = \lambda y \tag{11}$$

Nach Higham[13] können alle drei Schritte die Natur eines sehr komplexen Problems so stark verändern, dass wichtige Effekte verloren gehen können. Es wird für die in dieser

Arbeit betrachteten Modelle jedoch angenommen, dass die behandelten Probleme von einfacher Natur sind und sich lokal durch Linearisierungen, ausreichend genau approximieren lassen.

2.3 Numerische Integrationsverfahren

Für die Lösung eines Simulationsmodells, bestehend aus einem System aus DGL, können verschiedene Lösungsverfahren verwendet werden. Da eine analytische Lösung für die meisten Systeme schwer bis gar nicht zu finden ist, werden in der softwarebasierten Simulation numerische Lösungsverfahren zur Approximation der Lösung verwendet. Hierfür wird die exakte analytische Lösung $y(t)$ mit einer Näherung y_n an den diskreten Stützstellen $t_n = t_0 + i \cdot h$ mit $i = [0,\ldots,n]$ und der Schrittweite h approximiert. Ausgehend vom bekannten AWP aus Gl. (2) ergeben sich durch die Diskretisierung des AWP die Näherungswerte $[y_1,...,y_n]$ für die exakten, unbekannten Funktionswerte von $[y(t_1),...,y(t_n)]$. Dabei gibt es verschiedene numerische Integrationsverfahren für die Berechnung der Näherung. Einschritt- und Mehrschrittverfahren bilden zwei große Familien von numerischen Integrationsverfahren [14]. Beide lassen sich weiter in explizite und implizite Verfahren unterteilen. Dieser Abschnitt gibt eine Einführung in explizite und implizite Einschrittverfahren, sowie eine kurze Vorstellung von Mehrschrittverfahren.

2.3.1 Explizite Einschrittverfahren

Die einfachste numerische Integration ist durch das explizite Eulerverfahren - *forward Euler method* (FE-Verfahren) gegeben und verwendet zur Approximation den linearen Anteil der Taylorreihe. Das FE-Verfahren ist ein explizites Einschrittverfahren, welches die Näherung des nächsten Schrittes durch multiplikation der Funktionssteigung an der Stützstelle t_n mit der Schrittweite h bestimmt [4]. Die Funktionssteigung ergibt sich durch Einsetzen in die vorliegende DGL. Für eine gegebenen ODE mit dem Anfangswertproblem aus Gl. (2), kann der Folgeschritt y_{n+1} folgendermaßen genähert werden:

$$y(t_n + h) \approx y_{n+1} = y_n + h \cdot f(t_n, y_n) \tag{12}$$

Da alle Werte für die Berechnung von y_{n+1} bekannt sind, liegt Gl. (12) explizit vor. Wird das FE-Verfahren auf eine nichtlineare Differentialgleichung angewendet, wird das Problem durch die zeitliche Diskretisierung in eine lineare Gleichung umgewandelt, welche ohne weitere Schritte lösbar ist [16].

Das vorgestellte FE-Verfahren steht beispielhaft für das Grundprinzip von expliziten Lösungsverfahren. Es gibt verschiedene Ansätze, welche dieses Grundprinzip erweitern, um eine genauere Approximation zu bestimmen. So können z. B. bei der Berechnung des Folgeschrittes mehrere vorher berechnete Stützstellen zur Auswertung des Problems

hinzu gezogen werden (s. Abs. 2.3.3 Mehrschrittverfahren) oder, wie beim Runge-Kutta Verfahren (RK-Verfahren), weitere Funktionsauswertungen auf dem Intervall von $[t_n, t_{n+1}]$ durchgeführt werden. Die Familie der RK-Verfahren kann durch die mehrstufige Näherung eine höhere Genauigkeit gegenüber dem FE-Verfahren liefern, ohne dass weitere Ableitungen für eine höherrangige Taylorentwicklung berechnet werden müssen. Dies wird durch die gewichtete Linearkombination der zusätzlichen Funktionsauswertungen im Integrationsintervall erreicht. [18]

Mit den Gewichten $a_{i,j}$,b_i und c_i für die Stufen k_i ist die allgemeine Form für ein N-stufiges RK-Verfahren gegeben durch [17]:

$$y_{n+1} = y_n + h \cdot \sum_{i=1}^{N} b_i \cdot k_i \tag{13}$$

$$k_i = f(t_n + c_i \cdot h, y_n + h \cdot \sum_{j=1}^{N} a_{ij} \cdot k_j) \tag{14}$$

Es besteht ein ersichtlicher Zusammenhang zwischen der Anzahl der Stufen N und der Genauigkeit der Näherung. Inwieweit sich die Genauigkeit durch die Stufenzahl verbessert, kann durch die Konsistenzordnung (im weiteren auch kurz Ordnung) des Verfahrens ausgedrückt werden. Auf die allgemeine Beschreibung der Genauigkeit durch die Konsistenzordnung wird in Abs. 2.5 näher eingegangen. Die Koeffizienten des RK-Verfahren besitzen folgende Dimensionen:

$$\boldsymbol{c} = c_i \in \mathbb{R}^N, \quad \boldsymbol{b} = b_i \in \mathbb{R}^N \quad \text{und} \quad \boldsymbol{a} = a_{ij} \in \mathbb{R}^{N \times N} \tag{15}$$

und lassen sich übersichtlich im Butcher-Schema von J. Butcher [2] darstellen:

c	**a**
	b

Tabelle 3: *Allgemeines Butcher-Schema nach J. Butcher [2]*

Aus Gl. (13) und (14) wird ersichtlich, dass das RK-Verfahren der ersten Ordnung ($N = 1$), mit dem Gewicht $b = 1$ und $a = c = 0$, genau das FE-Verfahren darstellt. Das vierstufige ($N = 4$) *klassische Runge-Kutta Verfahren* lässt sich im Butscherschema wiefolgt ausdrücken:

0	0	0	0	0
1/2	1/2	0	0	0
1/2	0	1/2	0	0
1	0	0	1	0
	1/6	1/3	1/3	1/6

Tabelle 4: *Butcher-Schema vom klassischen Runge-Kutta Verfahren*

Angewendet auf Gl. (13) und (14) ergeben sich die expliziten Gleichungen des klassischen Runge-Kutta Verfahrens zu:

$$
\begin{aligned}
y_{n+1} &= y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \\
k_1 &= f(t_n, y_n) \\
k_2 &= f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1) \\
k_3 &= f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2) \\
k_4 &= f(t_n + h, y_n + h \cdot k_3)
\end{aligned}
\tag{16}
$$

2.3.2 Implizite Einschrittverfahren

Während in den FE-Verfahren alle Werte für die Näherung von y_{n+1} bekannt sind, bilden implizite Verfahren allgemeine nichtlineare Gleichungen mit Unbekannten. Analog zum FE-Verfahren bildet das implizite Eulerverfahren - *backward Euler method* (BE-Verfahren), das einfachste implizite Verfahren. Das BE-Verfahren wertet die Funktionssteigung an der Stelle t_{n+1} aus. Dadurch hängen die Funktionswerte von f, von dem unbekannten Wert y_{n+1} selbst ab. Ausgehend vom Anfangswertproblem aus Gl. (2), ergibt sich die Gleichung für den Folgeschritt y_{n+1} mit:

$$
y_{n+1} = y_n + h \cdot f(t_{n+1}, y_{n+1}) \tag{17}
$$

Gl. (17) liegt damit implizit vor und ist nicht direkt lösbar. Grundsätzlich sind für implizite Verfahren weitere Schritte zur vollständigen Berechnung der Lösung notwendig, sodass jeder Integrationsschritt mit einem größeren Aufwand verbunden ist. Die Lösung eines impliziten Schrittes kann beispielsweise durch ein iteratives Verfahren bestimmt werden. [18]

Die bereits vorgestellte Familie der RK-Verfahren kann sowohl explizit als auch implizit genutzt werden. Gl. (13) und (14) beschreiben ein explizites Verfahren falls $a_{ij} = 0$ für $j \geq i$, also **a** eine strikte untere Dreiecksmatrix ist. Die selben Gleichungen ergeben ein implizites Verfahren im anderen Fall [14].

2.3.3 Mehrschrittverfahren

Nach der Vorstellung der Einschrittverfahren wird in diesem Abschnitt kurz das Grundprinzip von Mehrschrittverfahren dargestellt. Mehrschrittverfahren greifen zur Berechnung

der Näherungswerte auf Informationen von mehreren, zuvor bereits verwendeten Stützstellen zurück. Ziel ist eine Approximation von höherer Ordnung mit weniger Funktionsauswertungen von f. Gleichzeitig bringen Mehrschrittverfahren auch Nachteile mit sich. Beispielsweise wird die Umsetzung einer Schrittweitenregelung durch die, für ein Mehrschrittverfahren scheinbar benötigten äquidistanten Stützstellen erschwert. Außerdem werden für den ersten Schritt in einem k-Schrittverfahren $k-1$ weitere Anfangswerte benötigt. Während fehlende Anfangswerte durch ein Einschrittverfahren berechnet werden können, bleibt die Umsetzung einer Schrittweitensteuerung für ein Mehrschrittverfahren deutlich komplizierter als bei den Einschrittverfahren. [14]

2.4 Modell- und Verfahrenseigenschaften

Soll ein vorliegendes System aus DGL numerisch gelöst werden, muss ein geeignetes Lösungsverfahren (Löser) mit passender Schrittweite gefunden werden. Die Wahl eines ungeeigneten Verfahrens kann zu ungenauen Lösungen oder sogar instabilen, falschen Werten führen. Vorbeugend sollten die Modelleigenschaften daher bekannt sein und das verwendete Verfahren darauf angepasst werden. Die folgenden Abschnitte erklären die zwei wichtigen Modell- und Verfahrenseigenschaften der Steifigkeit und Lösungsstabilität.

2.4.1 Steifigkeit

Die Steifigkeit eines Modells wird in der Literatur durch verschiedene Kriterien und Definitionen unterschieden. Dieser Abschnitt definiert Steifigkeit für die Verwendung in dieser Arbeit.

Steifigkeit tritt bei der numerischen Lösung eines AWPs auf wenn es Lösungsanteile gibt die sich verhältnismäßig schnell ändern, während sich andere Lösungsanteile langsam ändern [10]. Dies führt dazu, dass sich die Wahl der Schrittweite nach dem schnellen Lösungsanteil richten muss, während für den langsamen Lösungsanteil eine größere Schrittweite ausreichend wäre [14]. Die Steifigkeit von linearen DGL spiegelt sich in ihren Eigenwerten wieder. Besitzt eine DGL Eigenwerte die Betragsmäßig größer sind als die restlichen Eigenwertbeträge, so handelt es sich um eine steife DGL. Man spricht in diesem Zusammenhang auch von der Zeitkonstante eines Systems. Die Zeitkonstante verhält sich umgekehrt proportional zum Eigenwert, sodass ein System mit betragsmäßig großem Eigenwert eine kleine Zeitkonstante besitzt und umgekehrt. Ein System mit stark verschiedenen Zeitkonstanten führt zu einer benötigten Anpassung der Schrittweite h, wodurch für die Berechnung der Näherung viele Integrationsschritte benötigt werden und der Berechnungsaufwand deutlich größer ausfällt [20]. Eine steife DGL ist gerade für explizite Lösungsverfahren eine Schwierigkeit, da diese Verfahren beschränkte Stabilitätsgebiete haben (siehe Abs. 2.4.2).

2.4.2 Stabilität

In der Lösung von AWPs durch numerische Integration ist die Stabilität ein geläufiger Begriff. Allgemein ist ein System instabil, wenn kleine Änderungen in der Eingangsgröße zu großen oder selbstverstärkenden Änderungen in der Ausgangsgröße führen [18]. Meist wird auch eine Simulation, die durch arithmetisches Überlaufen zum Simulationsabbruch führt, als instabil bezeichnet. Analog bezeichnet die rein mathematische Definition die Lösung einer DGL als instabil, wenn y auf $0 < y < \infty$ unbeschränkt ist. Eine nicht instabile Lösung wird als stabil angesehen [10]. Im Folgenden wird zunächst weiter auf die mathematische Stabilitätsdefinition eingegangen.

In der Analyse von DGL und numerischen Näherungsverfahren kann zwischen der analytischen und der numerischen Stabilität unterschieden werden. Ist die exakte Lösung $y(t)$ eines AWPs für jedes $t \to \infty$ beschränkt, so ist die Lösung analytisch stabil. Die numerische Stabilität bezieht sich auf die äquivalente diskrete Lösung aus der numerischen Näherung [4]. Desweiteren ist die Näherung eines stabilen AWPs nur dann numerisch stabil, wenn sich die numerische Lösung der exakten Lösung nähert. Zur Analyse dieses Verhaltens, können die Stabilitätsgebiete der Verfahren betrachtet werden. Hierfür kann die Dahlquist Testgleichung als AWP herangezogen werden:

$$y' = \lambda \cdot y \tag{18}$$

Für ein komplexes λ ist eindeutig, dass die exakte Lösung $y(t)$ der Gl. (18) für $t \to \infty$ gegen null strebt, wenn $Re(\lambda) < 0$ ist (vgl. Gl. (3) und Abb. 3). Verhält sich die Lösung eines nume-

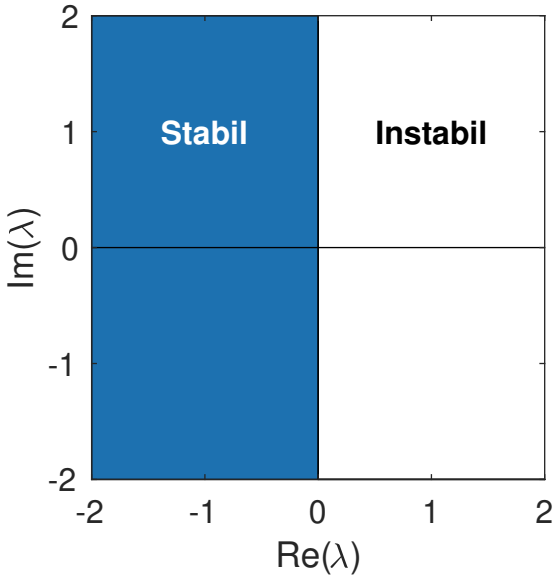

Abbildung 3: *Das analytische Stabilitätsgebiet der Gl. (18) wird auch A-stabil genannt.*

rischen Verfahrens mit der Schrittweite h gleich, dann befindet sich $h\lambda$ im Stabilitätsgebiet des Verfahrens. Das Stabilitätsgebiet ist definiert als die Menge der komplexen Zahlen

$z = h \cdot \lambda$, für die das numerische Verfahren mit fester Schrittweite h, bei der Lösung der Dahlquist Testgleichung (Gl. (18)), eine monoton fallende Folge von Näherungen liefert. [12]

Das in Abb. 3 dargestellte analytische Stabilitätsgebiet vom AWP aus Gl. (18) nennt sich auch A-Stabil, da das Stabilitätsgebiet die komplette linke Halbebene der komplexen Zahlenebene einnimmt [18]. Beispielhaft soll im Folgenden das Stabilitätsgebiet des FE-Verfahrens (siehe Abs. 2.3.2) hergeleitet werden. Wird das FE-Verfahren aus Gl. (12) auf die Dahlquist Testgleichung aus Gl. (18) angewendet, ergibt sich mit $z = h \cdot \lambda$ ausgehend von y_0 für den ersten Schritt y_1:

$$y_1 = (1 + z)\, y_0 \tag{19}$$

Wie bereits erläutert ist das Verfahren stabil, wenn die Lösung monoton fallend ist, sodass $|y_1| < |y_0|$ gilt und damit $|1 + z| < 1$ sein muss. Diese Bedingung wird auch durch die Stabilitätsfunktion $R(z) = 1 + z$ des FE-Verfahren ausgedrückt. Allgemein gilt

$$|R(z)| \leq 1 \tag{20}$$

mit

$$y_{n+1} = R(z) \cdot y_n \tag{21}$$

als Stabilitätsbedingung für die Lösung eines Verfahrens mit der Stabilitätsfunktion $R(z)$.[8] Das Stabilitätsgebiet des FE-Verfahrens beschreibt einen Kreis vom Radius 1 am Punkt -1 und ist in Abb. 4 dargestellt. Ein auf diese Weise hergeleitetes numerisches Stabilitäts-

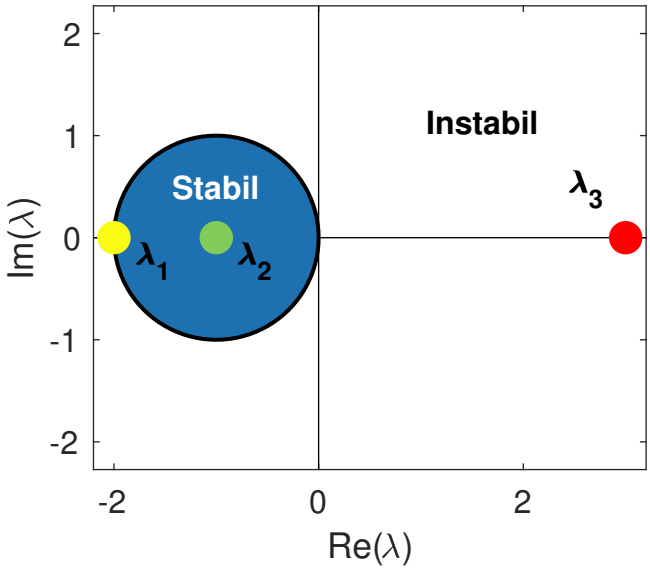

Abbildung 4: *Stabilitätsgebiet vom expliziten Eulerverfahren mit den eingezeichneten Eigenwerten* $\lambda_1 = -2$, $\lambda_2 = -1$ *und* $\lambda_3 = +3$

gebiet ist für das Integrationsverfahren allgemein gültig, sofern eine feste Schrittweite vorliegt und die Näherung eines linearen zeitinvarianten kontinuierlichen Systems durchgeführt wird [18]. Für eine linearisierte, ursprünglich nichtlineare Differentialgleichung

können daher nur Aussagen über die Stabilität im Bereich des Entwicklungspunktes der Linearisierung getroffen werden.

Die Bedeutung der Positionen der Eigenwerte eines Systems soll anhand der Dahlquist Testgleichung aus Gl. (18) näher erläutert werden. Hierfür werden drei Systeme mit unterschiedlichen Eigenwerten von $\lambda_1 = -2$, $\lambda_2 = -1$ und $\lambda_3 = +3$ betrachtet. Bei Anwendung des FE-Verfahrens mit einer festen Schrittweite von $h = 1{,}0$ kann die Stabilität des Verfahrens für die verschiedenen Systeme abgelesen werden.

Die Lösung für das System mit dem Eigenwert $\lambda_2 = -1$ (siehe Abb. 4) ist stabil, da der Eigenwert innerhalb des Stabilitätsgebiets liegt.

Der Eigenwert $\lambda_1 = -2$ liegt genau auf der Grenze des Stabilitätsgebiets, sodass die Lösung für dieses System als Grenzstabil bezeichnet wird. Die Approximation ist zwar beschränkt, jedoch findet keine numerische Näherung statt.

Instabil ist die Lösung des Systems mit $\lambda_3 = +3$, da der Eigenwert ausserhalb des Stabilitätsgebiets liegt. Damit ein Verfahren für ein System mit mehreren Eigenwerten stabil ist, müssen alle Eigenwerte im Stabilitätsgebiet liegen. [4]

Die Bedeutung des Lösungsverfahrens wird deutlich, wenn die selben drei Systeme durch das implizite Euler-Verfahren gelöst werden. Das Stabilitätsgebiet vom BE-Verfahren ist in Abb. 5 dargestellt [18]. Es ist schnell ersichtlich, dass nun alle Eigenwerte im stabilen

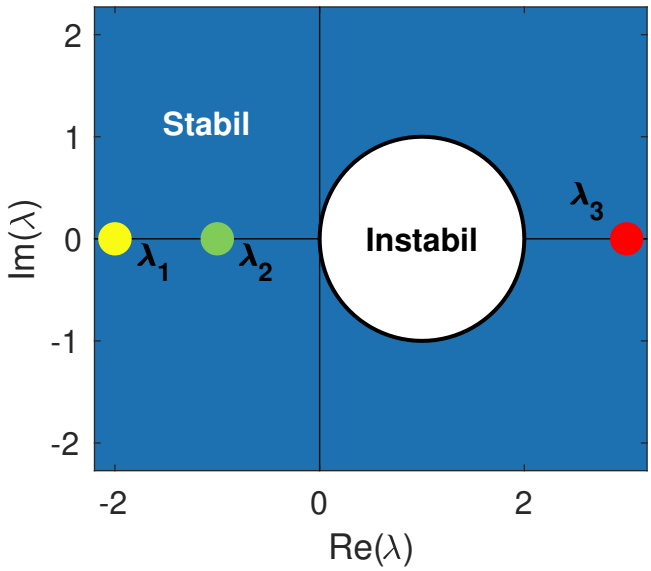

Abbildung 5: *Stabilitätsgebiet vom impliziten Eulerverfahren mit den eingezeichneten Eigenwerten* $\lambda_1 = -2$, $\lambda_2 = -1$ *und* $\lambda_3 = +3$

Bereich liegen. Der vorher grenzstabile Eigenwert $\lambda_1 = -2$ liegt nun vollständig im stabilen Bereich, da das Stabilitätsgebiet vom BE-Verfahren die gesamte linke Halbebene abdeckt. Tatsächlich ist dies bei allen impliziten Verfahren der Fall, sodass implizite Verfahren für steife Systeme, bessere Ergebnisse erzielen.[4]

Währen die Position des Eigenwerts $\lambda_3 = +3$, für das explizite Verfahren eine instabile Lösung andeutete, scheint die Lösung mit dem impliziten Verfahren stabil. Dies ist jedoch ein Trugschluss, da der Eigenwert zwar numerisch stabil, analytisch jedoch instabil ist. Dies ist ein bekanntes Problem bei impliziten Verfahren. Systeme die numerisch stabil scheinen, können sich analytisch als instabil herausstellen.

2.5 Fehler numerischer Verfahren

Der gesamte Fehler einer numerischen Lösung setzt sich aus verschiedenen Fehlerquellen zusammen. Auf die wichtigsten dieser Fehler wird im Folgenden weiter eingegangen.

2.5.1 Konsistezordnung und Diskretisierungsfehler

Numerische Integrationsverfahren sind Näherungsverfahren, welche ein gegebenes Problem, für welches keine explizite Lösung bekannt ist, durch ein Ersatzproblem annähern. Nach [7] muss das Ersatzproblem so formuliert sein, dass es numerisch gelöst werden kann und die Lösung nicht wesentlich von der expliziten Lösung abweicht. Fehler, die zur Abweichung der Näherung beitragen, werden als Verfahrensfehler bezeichnet. In der numerischen Integration von DGL stellt der Diskretisierungsfehler einen Verfahrensfehler dar. Der Diskretisierungsfehler beschreibt die Differenz zwischen der exakten Lösung an der Stelle $y(t_n)$ und der Näherung y_n eines Einschrittverfahrens:

$$\Delta_y = y(t_n) - y_n \tag{22}$$

Verkleinert sich der Fehler Δ_y mit Verkleinerung der Schrittweite h, wird von der *Konvergenz* des Verfahrens gesprochen. Gleichzeitig betrachtet die *Konsistenz* eines Einschrittverfahrens den lokalen Fehler, der pro Verfahrensschritt entsteht. Δ_y aus Gl. (22) wird daher auch als Konsistenzfehler bezeichnet. Die Konsistenzordnung ermöglicht eine Einschätzung der Verfahrensgüte, indem die Änderung des Konsistenzfehlers in Relation zur Änderung der Schrittweite h betrachtet wird. Allgemein gilt für die Konsistenzordnung $p \in \mathbb{N}$, beschrieben durch das Landau-Symbol $\mathcal{O}$:

$$\Delta_y = \mathcal{O}_p(h^p) \quad (h \to 0) \tag{23}$$

Die Konsistenzordnung ermöglicht eine Einschätzung der Verbesserung von der Approximation, bei Verkleinerung der Schrittweite h. Für ein Verfahren mit hoher Konsistenzordnung p sinkt der lokale Fehler Δ_y bei Verkleinerung der Schrittweite h schneller, als bei einem Verfahren mit geringerer Konsistenzordnung. Für die RK-Verfahren wurde bereits auf den Zusammenhang zwischen Stufenanzahl N und Konsistenzordnung hingewiesen. Um die Genauigkeit eines RK-Verfahren zu verbessern, gibt es daher die Möglichkeit die Schrittweite h zu verkleinern oder die Konsistenzordnung des Verfahrens zu erhöhen.

Die Erhöhung der Konvergenzordnung ist jedoch nur bis zu einer bestimmten Grenze sinnvoll, da die Stufenzahl N des expliziten RK-Verfahrens schneller wächst als die Konsistenzordnung. Für das bereits vorgestellte FE-Verfahren (RK-Verfahren erster Ordnung) ist die Konsistenzordnung $p = 1$ jedoch gleich der Stufenzahl $N = 1$. Dies gilt auch für das klassische Runge-Kutta Verfahren mit der Konsistenzordnung und Stufenzahl $N = p = 4$. [14]

2.5.2 Rundungsfehler

Bei der numerischen Näherung durch Computeralgorithmen treten in der Ausführung der Rechenoperationen lokale Rechnungsfehler auf. Diese resultieren aus dem endlichen Datenwort eines Computers, wodurch eine reelle Zahl nur zu einer endlichen Genauigkeit abgebildet werden kann. Übersteigt eine Zahl den Darstellungsbereich des Computers, wird die Zahl gerundet gespeichert. Der *Rundungsfehler* ist der Betrag der Differenz zwischen der eigentlichen Zahl z und der gerundeten Zahl. Das Ausmaß des Rundungsfehlers wird aus dem Beispiel von Cellier[4] deutlich. Betrachtet wird eine Taylorentwicklung der dritten Fakultät zur Integration von $y' = f(y,u,t)$. Es wird vereinfacht angenommen, dass Gleitkommazahlen mit einem Darstellungsbereich von ca. sechs signifikanten Stellen gerechnet werden.

$$y(t^* + h) \approx y(t^*) + f(t^*) \cdot h + \frac{\mathrm{d}f(t^*)}{\mathrm{d}t} \cdot \frac{h^2}{2!} + \frac{\mathrm{d}^2 f(t^*)}{\mathrm{d}t^2} \cdot \frac{h^3}{3!} \tag{24}$$

Nimmt man an, dass die Größenordnung von $\|y\|$ und seinen ersten drei Ableitungen $||f||$, $||\dot{f}||$, $||\ddot{f}||$ bei $\approx 1{,}0$ liegt und wird für die Lösung von Gl. (24) eine Schrittweite von $h = 10^{-3}$ gewählt, so ergibt sich für die Größenordnung der Summe:

$$\begin{aligned} \left\|y(t^* + h)\right\| \approx \quad & \left\|y(t^*)\right\| \quad + \quad \left\|f(t^*) \cdot h\right\| \quad + \quad \left\|\frac{\mathrm{d}f(t^*)}{\mathrm{d}t} \cdot \frac{h^2}{2!}\right\| \quad + \quad \left\|\frac{\mathrm{d}^2 f(t^*)}{\mathrm{d}t^2} \cdot \frac{h^3}{3!}\right\| \\ \approx \quad & 1{,}0 \quad + \quad 10^{-3} \quad + \quad 10^{-6} \quad + \quad 10^{-9} \end{aligned} \tag{25}$$

Die Größenordnung des Gesamtergebnisses bestimmt den Exponenten aller Summanden, sodass bei Summanden von kleinerer Größenordnung, signifikante Stellen *verloren* gehen. Dieser Fehler spielt vorallem dann eine Rolle, wenn sehr kleine Werte zu sehr großen Werten addiert werden. Abb. 6 aus dem Beispiel von Cellier[4] verdeutlicht dies, indem alle Summanden reihenweise dargestellt sind. Jedes Rechteck steht dabei für eine signifikante Dezimalstelle. Die gestrichelte Linie grenzt den Darstellungsbereich der Summe links von den wegfallenden Stellen auf der rechten Seite ab. Während der erste Summand aus Gl. (25) sechs signifikante Stellen beiträgt, kann der zweite Term nur noch drei signifikante Stellen und alle Verbleibenden garkeine beitragen. Daraus wird ersichtlich, dass die Berechnung der Glieder von zweiter und dritter Ordnung für dieses Beispiel irrelevant ist. Tatsächlich

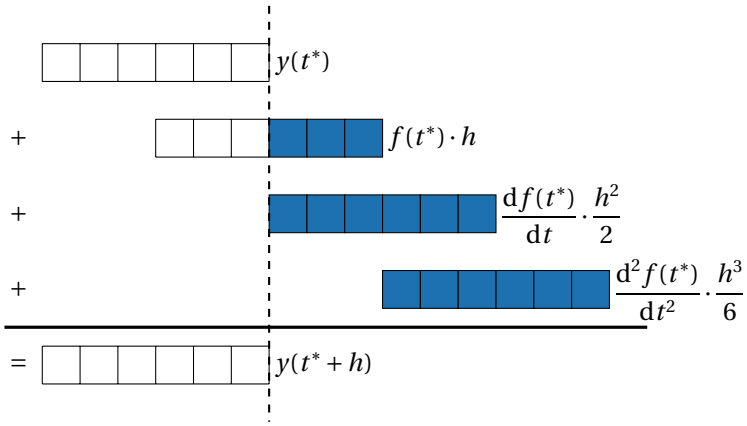

Abbildung 6: *Darstellung des Rundungsfehlers aus Gl. (25) nach Cellier[4]*

dramatisiert dieses Beispiel die beschriebenen Effekte, da bei einer Näherung dritter Ordnung normalerweise eine größere Schrittweite gewählt werden kann.[4]

2.6 Laufzeitkomplexität von Algorithmen

In diesem Abschnitt wird ein wichtiges Bewertungskriterium für Algorithmen vorgestellt. Da die Laufzeit von Algorithmen abhängig von dem ausführenden Rechner ist, wird ein Bewertungskriterium unabhängig von der verwendeten Hardware benötigt. Zum abstrakten Vergleich von Algorithmen wird daher die Komplexität eines Algorithmus betrachtet. Der Begriff Komplexität beschreibt in diesem Zusammenhang den Umgang des Algorithmus mit den zur Verfügung stehenden Betriebsmitteln, insbesondere mit Laufzeit und Speicherplatz [19]. Dies führt zu der Unterscheidung von Zeit- und Speicherkomplexität. Während die Speicherkomplexität den Speicherbedarf des Algorithmus bewertet, trifft die Zeitkomplexität eine Aussage über die Laufzeit des Programms. Die Bewertung der Laufzeitkomplexität wird im Folgenden anhand eines Beispiels von Solymosi[19] beschrieben. Hierfür wird die Ausführung der Prozeduren in Quellcode 1 betrachtet. Es ist

```
void proz0(int n) {
  proz1();
  for (int index1 = 1; index1 <= n; index1++) {
    proz2();
    for (int index2 = 1; index2 <= n; index2++) {
      proz3();
      proz3();
    };
  };
};
```

Quellcode 1: *Beispielcode für die Beschreibung der Zeitkomplexität eines Algorithmus*

ersichtlich, dass bei einmaligem Ausführen von `proz0()` die Anzahl der Ausführungen der restlichen Prozeduren vom Eingangsparameter n abhängt. Genauer wird `proz1()`

einmal, `proz2()` n-mal und `proz3()` $2 \cdot n^2$ mal ausgeführt. Gehen wir davon aus, dass die Ausführung der Schritte die meiste Zeit beansprucht und die Zeit für die Verwaltung der Schleifen vernachlässigbar ist, lässt sich die gesamte Ausführungsdauer von `proz0()` mit den bekannten Ausführungsdauern der restlichen Prozeduren und dem Eingangsparameter n bestimmen. Um von der absoluten Berechnung der Laufzeit zu einer abstrakten Aussage zu gelangen, wird ähnlich wie bei der Konsistenzordnung eines Lösungsverfahrens das asymptotische Verhalten des Algorithmus in Abhängigkeit von n durch das Landau-Symbol $\mathcal{O}(n)$ betrachtet. Es ist eindeutig, dass mit zunehmender Größe von n der Anteil der Laufzeit zunehmend von `proz3()` bestimmt wird. Der Laufzeitanteil von `proz2()` und `proz1()` wird mit zunehmendem n hingegen unbedeutend. Allgemein wird die Komplexität durch die höchste Potenz des Polynoms zur Beschreibung der Zeitabhängigkeit bestimmt. Die Komplexität des von `proz0()` dargestellten abstrakten Algorithmus beschreibt sich durch:

$$\mathcal{O}_{\texttt{proz0}}(n^2) = \mathcal{O}_{\texttt{proz3}}(2 \cdot n^2) + \mathcal{O}_{\texttt{proz2}}(n) + \mathcal{O}_{\texttt{proz1}}(1) \tag{26}$$

Für eine Vergrößerung der Problemgröße n um einen Faktor a bedeutet das eine Laufzeitvergrößerung um den Faktor a^2.

3 Lösungsherleitung

Ziel dieser Arbeit ist die Entwicklung eines Verfahrens zur automatischen parameterabhängigen Schrittweitenoptimierung einer HiL-Simulation. Dieses Kapitel beschreibt den Lösungsprozess, der bei der Entwicklung durchlaufen wurde. Hierfür werden im Folgenden die Anforderungen an die Analyse- und Lösungsverfahren definiert und durchgeführte Untersuchungen zum Einfluss der Simulationsparameter auf das Verhalten der Simulationsmodelle und deren Löser vorgestellt. Zunächst wird jedoch, zum besseren Verständnis der Folgeschritte, die allgemeine Ausgangsform eines Simulationsmodells vorgestellt.

3.1 Ausgangsmodell

Die hier behandelten Simulationsmodelle bestehen grundsätzlich aus AWPs von linearen und nichtlinearen ODE Systemen mit gegebenen Anfangswerten für alle Variablen. Bei der Entwicklung werden PDE zunächst nicht betrachtet, es wird sich jedoch herausstellen, dass die erarbeitete Optimierung mit leichten Anpassungen auch auf PDE anwendbar ist. Die betrachteten ODE Systeme beschreiben das Systemverhalten in Abhängigkeit von der Systemvariable y und der Eingangsvariable des Modells u. Ein praktisches Beispiel für die Eingangsvariable könnte z.B. eine angelegte Spannung an das Modell eines Elektromotors sein. Zusätzlich hängt das DGL System auch von den Parametern P ab. Diese Parameter sind während der Simulation meist konstant, können jedoch vor Simulationsstart angepasst werden um verschiedene Szenarien zu modellieren. Im Beispiel des zu simulierenden Elektromotors wären typische Parameter der elektrische Widerstand und die Induktivität des Motors. Da eine ausschließliche Systembeschreibung durch ein DGL System bei den meisten Modellen nicht ausreichend ist, werden Systemzustände eingeführt. Die Definition von Zuständen ermöglicht die Beschreibung des Systemverhaltens, mit unterschiedlichen DGL in verschiedenen Situationen. Im Fall des Elektromotors könnte ein Zustand durch die lastabhängige Rotation und ein zweiter durch die vollständige Blockierung, d.h. dem Festsetzen der Motorwelle beschrieben werden. Verbunden sind diese Zustände durch Schaltbedingungen, welche den Zustandsübergang in Abhängigkeit der Systemvariablen und Parameter beschreibt. Durch die Zustandswechsel sind auch unstetige Lösungsverläufe möglich, sowie Zustände mit konstanter Lösung. Zusätzlich zu der Systembeschreibung durch ODE können für den Ausgangsvektor noch algebraische Gleichungen benötigt werden. Diese sind vergleichbar mit den Aufgaben der Ausgangs- und Durchgangsmatrizen aus der Ausgangsgleichung der Zustandsraumdarstellung (s. Abs. 2.2.3).

3.2 Anforderungen

Um eine parameterabhängige Optimierung des Lösungsverfahrens zu ermöglichen, muss eine Optimierungsroutine entwickelt werden, welche auf eine allgemeine Modelldefinition anwendbar ist. Die Optimierungsroutine sollte in der Lage sein, auf Grundlage der parameterabhängen Modelleigenschaften, ein geeignetes Lösungsverfahren zu wählen und das Modell mit dem gewählten Verfahren stabil zu lösen. Gleichzeitig sollte die angewendete Schrittweite für eine möglichst genaue Lösung optimiert sein. Diese Grundanforderungen sind im Folgenden auf die drei Unterpunkte der Anforderungen an die Modelldarstellung, die Analyseverfahren und den Lösungsalgorithmus aufgeteilt.

3.2.1 Anforderungen an die Modelldarstellung

Für die Simulationsmodelle sollte eine allgemeine Darstellungsform existieren, welche eine Darstellung von Modellen verschiedener Größe und Komplexität ermöglicht und eine einfache Übergabe an die Optimierungsroutine erlaubt. Wie die Modellbeschreibung aus Abs. 3.1 zeigt, basiert die Modelldefinition auf den DGL der einzelnen Zustände. Es wird daher eine Darstellungsform benötigt, welche für verschiedene Zustände, unterschiedliche DGL in linearer und nichtlinearer Form beschreibt. Gleichzeitig müssen die zugehörigen Zustandsübergänge definiert werden, welche auch die Implementierung unstetiger Lösungsverläufe ermöglichen. Die allgemeine Darstellungsform eines Ausgangsmodells soll nach einmaliger Systemdefinition an das Lösungsverfahren übergeben werden. Hierfür wird eine geeignete Schittstelle zwischen Modell und Lösungsalgorithmus benötigt. Zur Realisierung der parameterabhängigen Optimierung muss außerdem eine Anpassung der Modellparameter nach der Definition möglich sein.

3.2.2 Anforderungen an die Analyseverfahren

Der Optimierungsroutine sollte es möglich sein gegebene Modelle zu analysieren um notwendige Informationen für das optimierte Lösungsverfahren zu erhalten. Hierfür werden Analyseverfahren benötigt, welche die allgemein definierten Modelle untersuchen. Die Analyseverfahren sollten Aufschluss über die parameterabhängigen Eigenschaften der DGL liefern. Vorallem sind Eigenschaften, welche Aufschluss über die Stabilität und die Steifigkeit des Modells, sowie über die Genauigkeit der Lösungsverfahren geben von Interesse. Diese Informationen können im Lösungsalgorithmus genutzt werden um ein geeignetes Lösungsverfahren mit passender Schrittweite zu finden. Da die DGL zustandsabhängig sind, sollten die Analysen alle möglichen Zustände beachten.

Neben der Analyse der Modelleigenschaften, sollte auch die Berechnungszeit eines Simulationsschrittes bekannt sein. Hierfür wird ein weiteres Analyseverfahren zur Laufzeitabschätzung benötigt. Die Durchführung der Analysen sollte sich in einem zeitlich

akzeptablen Rahmen halten und auf der zur Verfügung stehenden Hardware durchgeführt werden.

3.2.3 Anforderungen an den Lösungsalgorithmus

Der Lösungsalgorithmus soll die Informationen des Analyseverfahrens interpretieren und auf dieser Grundlage ein geeignetes Lösungsverfahren mit optimaler Schrittweite wählen. Die Optimierung soll hierbei drei Punkte beachten: die Genauigkeit und die Stabilität der Lösung, sowie die Echtzeitfähigkeit der Simulation. Die Genauigkeit der Lösung sollte maximiert werden, ohne dass die verbleibenden Anforderungen vernachlässigt werden. Für die HiL-Kompatibilität der Simulation ist die Echtzeitfähigkeit der Simulation eine wichtige Anforderung. Die parallele Ausführung von Hardwareschaltung und Simulation erfordert die Synchronisierung von Lösungsalgorithmus und Hardware Ein- und Ausgabe.

Neben der Verarbeitung der Analyseinformationen und der Wahl von Lösungsverfahren und Schrittweite, soll der Lösungsalgorithmus auch den Simulationsablauf steuern. Dies beinhaltet die Übergabe vom Modell an das Lösungsverfahren und die Ausführung der Simulationsschritte. In jedem Simulationsschritt sorgt der Lösungsalgorithmus für die Zustandsüberwachung der Simulation. Es löst die zustandsabhängigen ODE Systeme und führt die im Modell definierten Zustandsübergänge durch. Aufgrund der verschiedenen DGL Systeme für jeden Zustand, müssen auch Unstetigkeiten in der Lösung umgesetzt werden. Das bedeutet, dass die Lösung eines Integrationsschrittes nach der Berechnung unstetig veränderbar ist.

Der Lösungsalgorithmus greift für die Lösung einer DGL auf das gewählte Lösungsverfahren zurück. Die grundlegende Anforderung an das Lösungsverfahren ist die numerische Lösung linearer und nichtlinear Probleme mit angemessener Genauigkeit. Es sollen Lösungsverfahren existieren, die mit unterschiedlichen Eigenschaften, wie steifen und nichtsteifen DGL, optimal umgehen.

3.3 Untersuchung von Lösungsansätzen

In der Entwicklung der Optimierungsroutine werden verschiedene Ansätze betrachtet, welche maßgeblich für den Prozess der Lösungsfindung sind. Dieses Kapitel beschreibt den Weg der Lösungsfindung und die dabei gewonnen Erkenntnisse. Hierfür werden zunächst die notwendigen Schritte für die Lösung eines linearen und eines nichtlinearen Modells allgemein betrachtet. Anschließend folgt die Betrachtung verschiedener Optimierungsansätze für größere Lösungsgenauigkeit. Eine Untersuchung der Unterschiede in der Lösung durch implizite und explizite Verfahren führt dann zur Betrachtung eines

kombinierten Lösungsverfahren. Die Erkenntnisse der untersuchten Lösungsansätze werden schließlich genutzt um Analyseverfahren herzuleiten, die bei der Optimierung der Lösungsverfahren hilfreich sind.

3.3.1 Lösung linearer und nichtlinearer Modelle

Um ein Lösungsverfahren und eine allgemeine Darstellungsform zu finden, ist es sinnvoll zunächst die notwendigen Lösungsschritte verschiedener Modelle zu betrachten. Im Folgenden wird daher beispielhaft die numerische Lösung für ein lineares und ein nichtlineares Modell bestimmt. Hieraus werden Erkenntnisse für die Gestaltung der allgemeinen Darstellungsform gezogen und die Umsetzungsmöglichkeiten von numerischen Lösungsverfahren betrachtet.

Lineares Modell

Es wird der numerische Lösungsweg für eines linearen Systems mit zwei Zustandsvariablen y_1 und y_2 betrachtet. Das DGL System sei gegeben durch:

$$y_1'(t) = a \cdot y_1(t) + b \cdot y_2(t) + r \cdot u(t) \tag{27}$$

$$y_2'(t) = c \cdot y_1(t) + d \cdot y_2(t) \tag{28}$$

Das durch Gl. (27) und Gl. (28) gegebene Modell kann z. B. zur Beschreibung eines Gleichstrommotors mit Gleichungen für Ankerstrom und Drehzahl genutzt werden. Für eine Darstellung im Zustandsraummodell, wird der Zustandsvektor y definiert durch:

$$y = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \tag{29}$$

Das Zustandsraummodell mit y_2 als Ausgangsvariable (y_{out}) ist dann gegeben durch:

$$y'(t) = \underbrace{\begin{pmatrix} a & b \\ c & d \end{pmatrix}}_{=A} y'(t) + \underbrace{\begin{pmatrix} r \\ 0 \end{pmatrix}}_{=B} \cdot u(t) \tag{30}$$

$$y_{out}(t) = \underbrace{\begin{pmatrix} 0 & 1 \end{pmatrix}}_{=C} \cdot y(t) + \underbrace{0}_{=D} \cdot u(t) \tag{31}$$

Zur Systemanalyse können nun die Eigenwerte der Systemmatrix A betrachtet werden. Angenommen das System besitzt die reellen Eigenwerte $\lambda_1 = -100$ und $\lambda_2 = -1$, so ergeben

sich die Zeitkonstanten der Zustandsvariablen τ_1 und τ_2 zu:

$$\tau_1 = \frac{1}{|\lambda_1|} = 0{,}01s \tag{32}$$

$$\tau_2 = \frac{1}{|\lambda_2|} = 1s \tag{33}$$

Die Zeitkonstanten zeigen, dass y_1 deutlich schneller auf Änderungen reagiert als y_2, sodass das zugehörige DGL System als steif bezeichnet werden kann. Zur Lösung steifer Modelle eignen sich, wie in 2.4.1 beschrieben, besonders implizite Lösungsverfahren. Im Folgenden, soll daher eine Lösung mit dem impliziten Eulerverfahren gefunden werden:

$$f(y_{n+1}, u_{n+1}, t_{n+1}) = A \cdot y_{n+1} + B \cdot u_{n+1} \tag{34}$$

$$y_{n+1} = y_n + h \cdot f(y_{n+1}, u_{n+1}, t_{n+1}) \tag{35}$$

$$y_{out,n+1} = C \cdot y_{n+1} + D \cdot u_{n+1} \tag{36}$$

Durch Einsetzen von Gl. (34) in Gl. (35) und triviales Umformen mit einer 2x2 Einheitsmatrix E, ergibt sich für y_{n+1} die lösbare Gleichung:

$$y_{n+1} = \left(E - h \cdot A\right)^{-1} \cdot \left(y_n(t) + h \cdot B \cdot u_{n+1}\right) \tag{37}$$

Für dieses lineare Beispiel lässt sich das implizite Verfahren in eine direkt lösbare Gleichung umformen und es wird kein Iterationsverfahren zur Lösung benötigt. Dies ist für andere implizite Verfahren oder nichtlineare Systeme nicht möglich. Mit gegebenen Anfangswerten und einer Schrittweite h, ließe sich hier der numerische Lösungsverlauf bestimmen. Da die Eigenwerte λ_1 und λ_2 negativ reelle Zahlen sind und der Stabilitätsbereich des impliziten Eulerverfahrens die komplette linke Halbebene abdeckt, ist die Lösung von diesem Beispiel mit jeder möglichen Schrittweite h numerisch stabil.

Nichtlineares Modell

Im Folgenden soll die numerische Integration allgemein für ein nichtlineares System durchgeführt werden. Hierfür wird beispielhaft ein Modell zum Stromfluss in einer Halogenlampe mit temperaturabhängigem Widerstand verwendet. Die verallgemeinerte nichtlineare ODE mit den Zustandsvariablen y_1 und y_2 ist gegeben durch:

$$\begin{aligned} y_1'(t) &= a \cdot y_1^k(t) - b \cdot (y_1(t) - \gamma)^4 \\ y_2'(t) &= c \cdot y_1^k(t) \cdot y_2(t) - d \cdot u(t) \end{aligned} \tag{38}$$

Mit dem Zustandsvektor $y = [y_1, y_2]^\top$, lässt sich Gl. (38) in der Funktionsdarstellung aus

Gl. (6) darstellen:

$$f(t,y,u) = \begin{pmatrix} a \cdot y_1^k - b \cdot (y_1 - \gamma)^4 \\ c \cdot y_1^k \cdot y_2 - d \cdot u \end{pmatrix} \tag{39}$$

Zur Modellanalyse wird die Linearisierung der DGL an den Entwicklungspunkten $\bar{y}_1$ und $\bar{y}_2$ durchgeführt. Die hierfür benötigte Jacobi-Matrix aus Gl. (7) ergibt sich zu:

$$J = \begin{pmatrix} a \cdot k \cdot y_1^{k-1} - b \cdot 4 \cdot (y_1 - \gamma)^3 & 0 \\ c \cdot k \cdot y_1^{k-1} \cdot y_2 & c \cdot y_1^k \end{pmatrix} \tag{40}$$

Durch Einsetzen der Entwicklungspunkte $\bar{y}_1$ und $\bar{y}_2$ in die Jacobi-Matrix, erhält man die konstante Systemmatrix A für das linearisierten System in der Umgebung des Entwicklungspunkts. Die Eigenwerte λ_1 und λ_2 lassen sich wie im linearen Beispiel berechnen und können für eine Modellanalyse im näheren Bereich des Entwicklungspunkts verwendet werden. Wird das explizite Eulerverfahren aus Gl. (12) auf die Funktionsdarstellung angewendet, so ergibt sich das zeitdiskrete lineare Gleichungssystem:

$$\begin{pmatrix} y_{1,n+1} \\ y_{2,n+1} \end{pmatrix} = \begin{pmatrix} y_{1,n} \\ y_{2,n} \end{pmatrix} + h \cdot \begin{pmatrix} a \cdot y_{1,n}^k - b \cdot (y_{1,n} - \gamma)^4 \\ c \cdot y_{1,n}^k \cdot y_{2,n} - d \cdot u_n \end{pmatrix} \tag{41}$$

Gl. (41) kann mit gegebenen Anfangswerten y_0 und einer bekannten Eingangsgröße u_0 eindeutig gelöst werden.

3.3.2 Optimierungsansätze für Lösungsverfahren

Für steife oder sprunghafte Systeme gibt es verschiedene Optimierungsansätze, für eine stabile und genauere Näherung. Im Zusammenhang mit einer HiL-Simulation ergeben sich hier jedoch einige Einschränkungen. Um die Grundprinzipien dieser Optimierungsansätze und die gegebenen Einschränkungen zu verstehen, werden in diesem Abschnitt verschiedene Optimierungsansätze für Softwaresimulationen vorgestellt und die Schwierigkeit der Übertragung auf Echtzeitsimulationen beschrieben.

Eine weit verbreitete Möglichkeit steife Systeme effizient zu lösen ist die Schrittweitensteuerung. Im Gegensatz zu einem numerischen Integrationsverfahren mit äquidistanter Schrittweite, wird die Schrittweite bei der Schrittweitensteuerung laufend angepasst. So kann die Näherung eines Problems mit großer Zeitkonstante durch ein Verfahren mit großer Schrittweite schnell und genau gefunden werden. Ändert sich die Zeitkonstante während der Simulation, wird die Schrittweite dynamisch angepasst um stets ein Optimum zwischen Simulationsdauer und Genauigkeit zu finden. Die Schrittweitensteuerung greift hierfür auf eine Fehlerabschätzung für jeden Verfahrensschritt zurück. Die Umsetzung einer Schrittweitensteuerung für eine HiL-Simulation ist nur bedingt sinnvoll. Zum Einen führt eine solche Anpassung, für das hier betrachtete Beispiel, zu zusätzlichen Herausforderungen durch die notwendigen Synchronisierung von paralleler Hard- und

Softwaresimulation. Wird die Schrittweite laufend angepasst, muss die aktuelle Schrittweite stets an alle Komponenten der HiL-Simulation kommuniziert werden. Dies kann zu Einschränkungen in parallel ablaufenden Hardwareprozessen führen. Zum Anderen ist die kleinstmögliche Schrittweite für eine HiL-Simulation durch die Echtzeitanforderung begrenzt, sodass Zustände mit kleiner Zeitkonstante ohnehin nur mit der kleinstmöglichen Schrittweite gelöst werden können. Wird die Schrittweite für Zustände mit größerer Zeitkonstante vergrößert, kann kein direkter Nutzen aus dem Zeitgewinn der kürzeren Berechnungszeit gezogen werden, da jeder Folgeschritt erst mit dem Einlesen der neuen Eingangsgrößen gestartet wird. Durch die Abtast-halte-Schaltung (vgl. Abb. 2) würde eine größere Schrittweite sogar zum längeren Halten eines Wertes und damit zu einer größeren Abweichung zwischen Momentanwertabtastung und Funktionsverlauf führen. Aus diesem Grund scheint die Durchführung einer HiL-Simulation mit der kleinstmöglichen Schrittweite am sinnvollsten.

Ein weiterer Ansatz zur Optimierung ist bei der Verwendung von RK-Verfahren gegeben. Diese bieten die Möglichkeit die Ordnung des Lösungsverfahrens dynamisch anzupassen. Für Softwaresimulationen kann die Genauigkeit daher mit konstanter Schrittweite eingestellt werden. [3] Dafür wird auch hier eine Fehlerabschätzung in jedem Schritt durchgeführt. Wird ein festgelegter Toleranzwert nicht eingehalten, ist die bestimmte Näherung zu ungenau und es wird zu einem Verfahren höherer Konsistenzordnung gewechselt. Die Fehlerabschätzung kann hierbei durch eingebettete Runge-Kutta Verfahren durchgeführt werden [3]. Diese berechnen mithilfe unterschiedlicher Gewichtungsfaktoren für b, Näherungen von verschiedener Ordnung. Die Fehlerabschätzung wird dann aus der Differenz der beiden Näherungen gebildet. Da ein RK-Verfahren höherer Ordnung weitere Funktionsauswertungen bestimmt, wird mit der Erhöhung der Ordnung auch die Berechnungsdauer verlängert. Für eine HiL-Simulation bedeutet die Veränderungen der Berechnungszeiten jedoch auch eine Veränderung der Schrittweite. Für die Übertragung auf eine HiL Simulation treten daher die gleichen Probleme wie bei der Schrittweitensteuerung auf.

Beide Optimierungsansätze verdeutlichen, dass eine Optimierung für ein Lösungsverfahrens einer HiL-Simulation keinen großen Einfluss auf die Berechnungsdauer haben darf. Es muss daher ein Optimierungsansatz gefunden werden, welcher für verschiedene Systemzustände die jeweils größte Genauigkeit bei annähernd gleicher Berechnungszeit liefert. Wie in Abs. 2.3 bereits angedeutet wurde, können implizite und explizite Lösungsverfahren unterschiedlich gut mit steifen und sprunghaften DGL umgehen. Aus diesem Grund werden im Folgenden die Unterschiede der Näherungslösung beider Verfahren genauer untersucht.

3.3.3 Zu impliziten und expliziten Lösungsverfahren

In Abs. 2.4.2 wurde bereits der Zusammenhang von Lösungstabilität zu impliziten und expliziten Lösungsverfahren vorgestellt. Da eine der wichtigsten Anforderungen an das Lösungsverfahren das Finden einer stabilen Lösung ist, soll im Folgenden das Verhalten von impliziten und expliziten Lösern für HiL-Simulationen betrachtet werden.

Zunächst wird die Bedeutung der Stabilitätsgrenzen mit einem einfachen Testmodell, bestehend aus zwei gekoppelten linearen DGL, untersucht. Das Testmodell besitzt die konstanten Eigenwerte $\lambda_1 = -4$ und $\lambda_2 = -3$. Für die Lösung des AWPs wird das FE-Verfahren mit dem bekannten Stabilitätsgebiet aus Abs. 2.4.2 verwendet. Abb. 7 zeigt drei Szenarien, in denen die verwendete Schrittweite h so variiert wird, dass das Produkt $z = h \cdot \lambda$ stets innerhalb des Stabilitätsgebiets liegt und die Lösung damit stabil ist. Auf der linken Seite

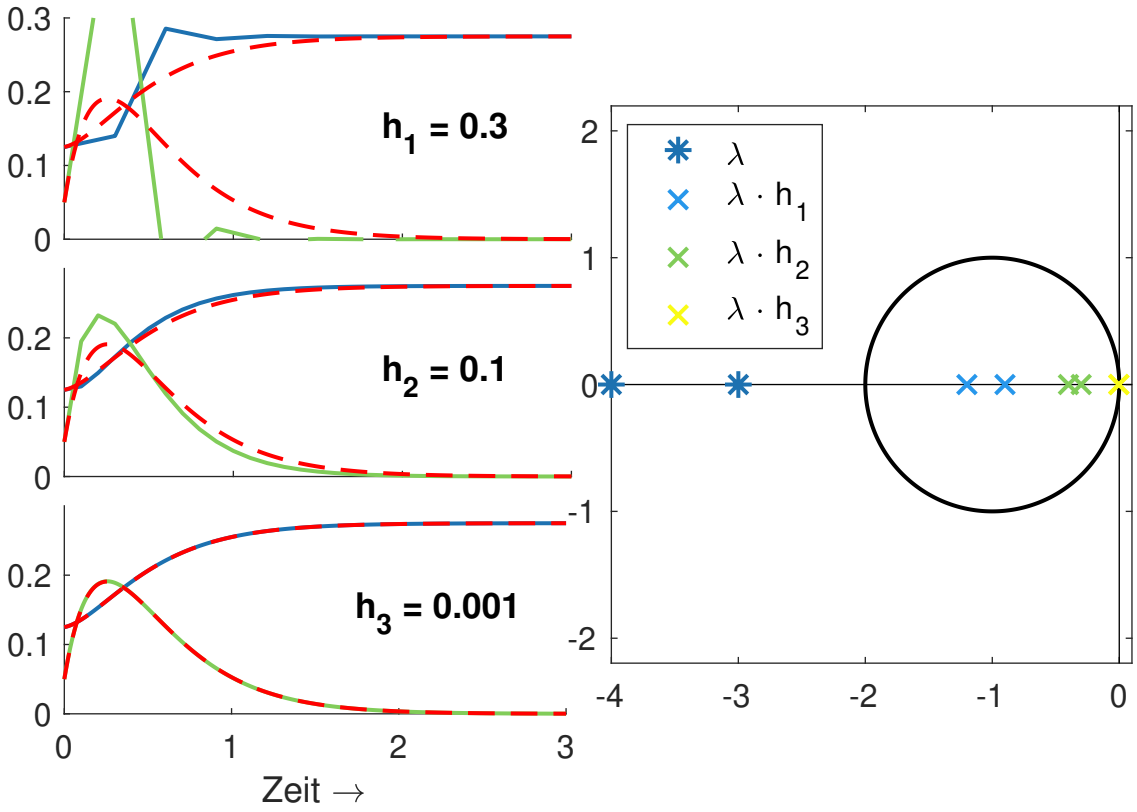

Abbildung 7: *Explizites Lösungsverfahren mit den Schrittweiten $h_1 = 0{,}3$, $h_2 = 0{,}1$, $h_3 = 0{,}001$, angewendet auf ein Testmodell. Der Lösungsverlauf und die exakte Lösung (rot) ist links dargestellt, rechts zu sehen ist das Stabilitätsgebiet des FE-Verfahrens und die zu den Schrittweiten gehörigen Eigenwerte.*

der Abb. 7, ist die exakte Lösung des Testmodells rot dargestellt. Die Lösungsvariationen mit den angewendeten Schrittweiten von h_1 bis h_3 sind untereinander dargestellt. Auf der rechten Seite sind die Eigenwerte des Testmodells und die zu den Schrittweiten gehörigen Produkte $\lambda \cdot h_i$ im Stabilitätsgebiet des Lösers dargestellt. Sofort ersichtlich ist, dass die Näherungen mit größerer Schrittweite eine geringere Genauigkeit aufweisen.

Dies lässt sich durch den in Abs. 2.5 beschriebenen Diskretisierungsfehler schnell begründen. Eine zusätzliche Erkenntnis ist, dass die Genauigkeit der Näherung vorallem in den dynamischen Lösungsverläufen geringer ist. Die große Schrittweite h_1 führt zu Beginn der Simulation zu einem Überschwingen der Lösung. Erst im weiteren Zeitverlauf kann sich die Näherung an die exakte Lösung angleichen und der Fehler verringert sich. Für dynamische Lösungsverläufe erzielen Schrittweiten an der betragsmäßig größeren Grenze deutlich schlechtere Ergebnisse. Im Folgenden der Arbeit wird die Grenze mit der größten Entfernung vom Ursprung, in der negativ reellen Halbebene, als Stabilitätsgrenze bezeichnet. Die Stabilitätsgrenze bezeichnet den Punkt, der in Richtung des Ursprung mindestens überschritten werden muss, um explizit eine stabile Näherung bestimmen zu können. Wird im weiteren Verlauf daher vom grenznahen Bereich gesprochen, ist eine Nähe zur Stabilitätsgrenze gemeint. Für ausgeglichene Lösungsverläufe reicht eine Schrittweite in der Nähe der Stabilitätsgrenze aus. Durch die gekoppelten DGL zeigt sich außerdem, dass sich der große Fehler der ersten Gleichung auch auf den Fehler in der zweiten Gleichung auswirkt. Wird die Schrittweite so gewählt, dass das Produkt z mit dem Eigenwert $\lambda_1 = -4$ gerade außerhalb des Stabilitätsgebiets liegt und mit dem Eigenwert $\lambda_2 = -3$ noch innerhalb, sind trotzdem beide Lösungen instabil. Bei gekoppelten Gleichungen ist stets die Gleichung mit dem betragsmäßig größten Eigenwert zu betrachten, um das Gesamtsystem stabil zu lösen.

Da die Schrittweite des Lösungsverfahrens nicht beliebig klein gewählt werden kann, besteht bei expliziten Lösungsverfahren die Gefahr in instabile Lösungsbereiche zu geraten. Wie bereits das vorherigen Beispiel gezeigt hat, führen die Oszillationen der expliziten Lösung in der Nähe der Stabilitätsgrenze zu Ungenauigkeiten der Näherung. Diesen Ungenauigkeiten kann durch die Verwendung eines impliziten Verfahrens entgegen getreten werden. Das Beispiel aus Abb. 8 veranschaulicht dies durch den Vergleich von expliziter und impliziter Lösung an einer sprunghaften Lösungsänderung. Der in Abb. 8 dargestellte Verlauf ist die Lösung eines AWPs, bei welchem der Eigenwert der zu lösenden DGL zum Zeitpunkt $t = 0{,}5$ sprungartig von $\lambda_1 = -5$ auf $\tilde{\lambda}_1 = -1000$ verändert wird. Die Schrittweite h ist dabei so gewählt, dass das Produkt von $z = \tilde{\lambda}_1 \cdot h$ ganz knapp im Stabilitätsbereich des expliziten Verfahrens liegt. Abb. 8 zeigt die Auswirkung, der sprunghaften Änderung auf das explizite und das implizite Verfahren. Obwohl beide Verfahren die gleiche Schrittweite zur Lösung verwenden, gerät die explizite Lösung in eine starke Schwingung und benötigt eine gewisse Zeit bis zum Einpendeln. Das implizite Verfahren löst die kritische Stelle ohne Oszillation und weist dadurch eine deutlich höhere Genauigkeit auf. Der Nachteil beim impliziten Verfahren ist jedoch, dass die Berechnungszeit für die implizite Lösung deutlich größer ist, als die Berechnungszeit der expliziten Lösung. Dies liegt an den zusätzlich notwendigen Funktionsauswertungen und der Iterationsschleife des Lösers.

Da eine größere Berechnungszeit in einer HiL-Simulation zur Nichteinhaltung der Echtzeitanforderung führen kann, müsste die Schrittweite für ein implizites echtzeitfähiges Verfahren vergrößert werden. Eine größere Schrittweite führt jedoch wieder zu einer größeren Diskretisierung und dies zu einer größeren Abweichung in der Momentanwertabtas-

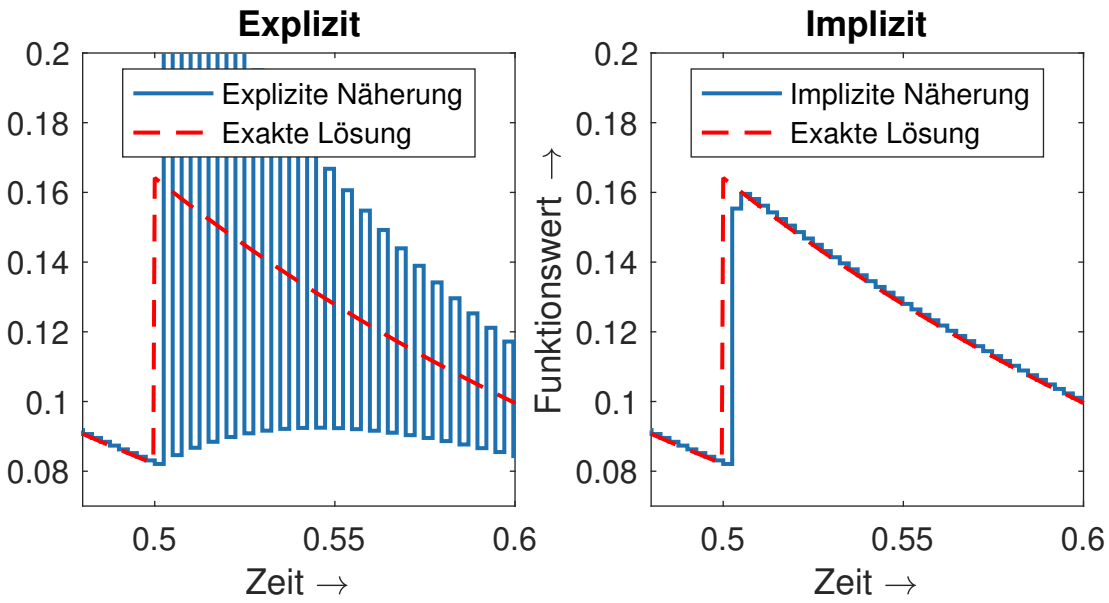

Abbildung 8: *Sprunghafte Änderung an der Stabilitätsgrenze führt zur Oszillation der expliziten Näherung (links). Ein implizites Verfahren findet eine stabile Lösung (rechts).*

tung der Hardwareausgabe. Im Gegensatz zu expliziten Verfahren bestimmt ein implizites Verfahren zwar eine akkurate Näherung in grenzstabilen Regionen, es muss zur Einhaltung der Echtzeitanforderungen aber auch eine größere Schrittweite wählen, sodass die Abweichung des Ausgangssignals größer wird. Ein optimaler Löser für eine HiL-Simulation mit Echtzeitanforderung muss daher die Schnelligkeit von expliziten Lösungsverfahren mit der Lösungsstabilität von impliziten Verfahren kombinieren.

3.3.4 Kombinierte Lösungsverfahren

Wie der vorherige Abschnitt verdeutlicht hat, wäre es nützlich die Eigenschaften von impliziten und expliziten Lösungsverfahren, für eine Echtzeitfähige HiL-Simulation zu kombinieren. Ziel ist eine möglichst hohe Genauigkeit bei gleichzeitig kurzer Berechnungszeit zu erlangen und gleichzeitig die Lösungsstabilität zu garantieren. Im Folgenden wird der Ansatz eines kombinierten Lösungsverfahrens verfolgt. Wie sich zeigen wird, ist das hier vorgestellte Verfahren für eine HiL-Simulation nicht direkt umsetzbar, es trägt jedoch zur Konzeptentwicklung maßgeblich bei und wird später in veränderter Form umgesetzt.

Das Grundkonzept des kombinierten Lösungsverfahrens ist, ungenaue explizite Lösungen z. B. an der Stabilitätsgrenze durch ein implizites Verfahren zu korrigieren. Hierfür wird in jedem Simulationsschritt eine Fehlerabschätzung für die explizite Näherung durchgeführt. Die Abschätzung kann z. B. durch das in Abs. 3.3.2 vorgestellte eingebettete RK-Verfahren bestimmt werden. Im kombinierten Verfahren korrigiert der implizite Löser die explizite

Näherung wenn die Fehlerabschätzung einen Toleranzwert überschreitet. Die berechnete explizite Näherung für diesen Zeitschritt wird dann als Initialwert für ein implizites Iterationsverfahren verwendet. Wird ein kombinierter Löser dieser Art auf das Beispiel aus Abb. 8 angewendet, erhält man den in Abb. 9 dargestellten Lösungsverlauf. In diesem Beispiel handelt es sich um eine reine Softwaresimulation, sodass die Simulation von der Echtzeitanforderung einer HiL-Simulation gelöst ist. Die gesamte Simulationsdauer bestimmt sich daher lediglich aus der Summe der Berechnungszeiten der einzelnen Schritte. Abb. 9 zeigt den tatsächlichen Funktionsverlauf des bekannten kritischen Schaltpunkts im

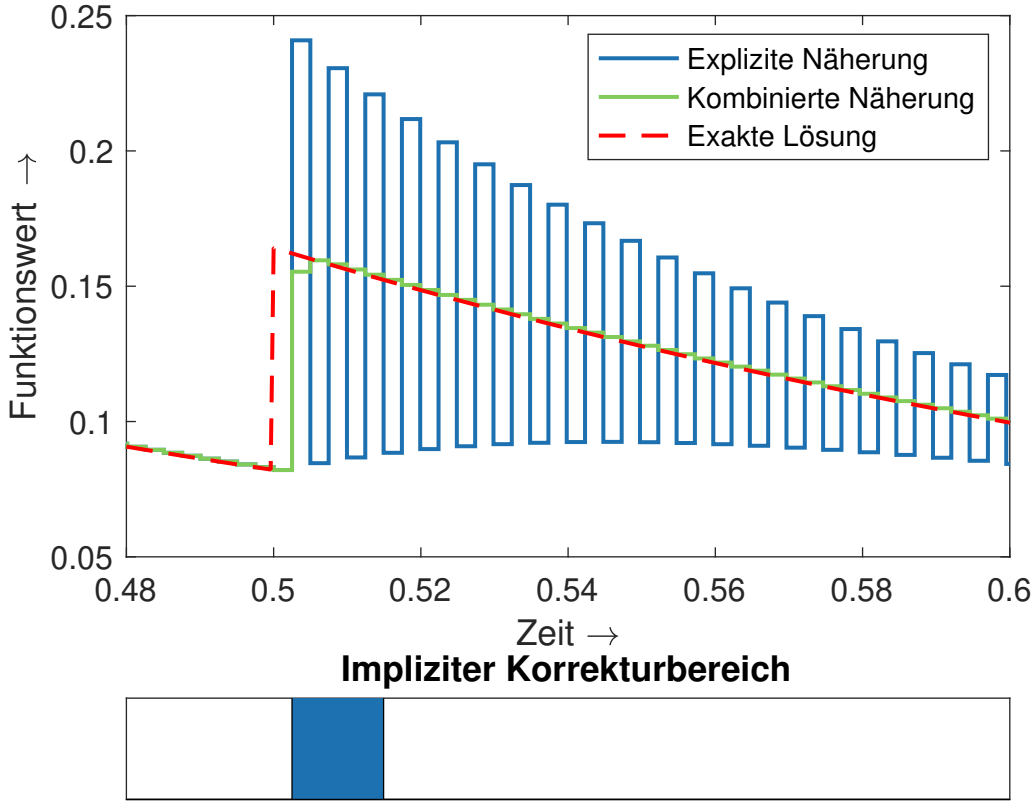

Abbildung 9: *Vergleich von kombinierter und expliziter Näherung an einer sprunghaften Änderung des Lösungsverlaufs. Der Bereich der impliziten Lösung im kombinierten Verfahren ist im unteren Diagramm markiert.*

oberen Diagramm als rote Linie. Zusätzlich sind die Näherungen durch ein rein explizites Verfahren und durch das vorgestellte kombinierte Verfahren eingezeichnet. Im unteren Diagramm ist der Zeitbereich, in welchem die expliziten Näherung durch implizite Iteration korrigiert wird, angezeigt. Es ist zu sehen, dass die implizite Korrektur zum Zeitpunkt der sprunghaften Änderung bei $t = 0.5$ einsetzt. Das bedeutet, dass die Fehlerabschätzung für die explizite Näherung zu diesem Zeitpunkt größer ist als der angegebene Toleranzwert. Die implizite Korrektur ist anschließend für fünf Schritte aktiv. Während das rein explizite Verfahren eine starke Oszillation aufweist, stabilisieren die wenigen impliziten Korrekturschritte das Verfahren und die Abweichung von der exakten Lösung fällt sehr gering aus. Nach den impliziten Korrekturschritten, kann die kombinierte Näherung wie-

der mit ausreichender Genauigkeit rein explizit bestimmt werden, während die explizite Näherung weiterhin stark oszilliert. Da die implizite Korrektur im kombinierten Verfahren zusätzliche Zeit benötigt, ist die Simulationsdauer etwas größer als die des rein expliziten Verfahrens. Die zusätzliche Zeit wird jedoch nur im Korrekturbereich benötigt und die Berechnungszeit der expliziten Schritte ist im kombinierten Verfahren sowie im rein expliziten Verfahren gleich. Vergleicht man die kombinierte Näherung aus Abb. 9 mit der rein impliziten Näherung aus Abb. 8, ist kein sichtbarer Unterschied festzustellen. Die gesamte Simulationsdauer des kombinierten Verfahrens liegt aufgrund der expliziten Anteile jedoch deutlich unter der Simulationsdauer des rein impliziten Verfahrens.

Um diese Art der Korrektur in einer HiL-Simulation umzusetzen, ist wieder eine Änderung der Schrittweite im Laufe der Simulation erforderlich. Ohne Schrittweitenanpassung müsste die Schrittweite vor der Simulation so gewählt werden, dass in jedem Schritt genug Zeit für eine explizite Näherung mit impliziter Korrektur ist. Die gewählte konstante Schrittweite wäre dann wieder genauso groß, wie die Schrittweite eines rein impliziten Verfahrens und der Vorteil der schnelleren expliziten Simulation geht verloren.

Im Folgenden wird daher ein kombiniertes Lösungsverfahren betrachtet, welches die Schrittweite im Falle einer impliziten Korrektur für nur den impliziten Schritt vergrößert und die expliziten Schritte weiterhin mit der kleineren Schrittweite bestimmt. Abb. 10 zeigt beispielhaft einen Ausschnitt einer Lösung eines solchen Verfahrens. Das gelöste Modell besitzt zum betrachteten Zeitpunkt eine sprunghafte Änderung mit leichtem Schwingverhalten. Die im Beispiel verwendeten impliziten und expliziten Schrittweiten wurden für die bessere Veranschaulichung stark vergrößert. Der theoretische Funktionsverlauf der

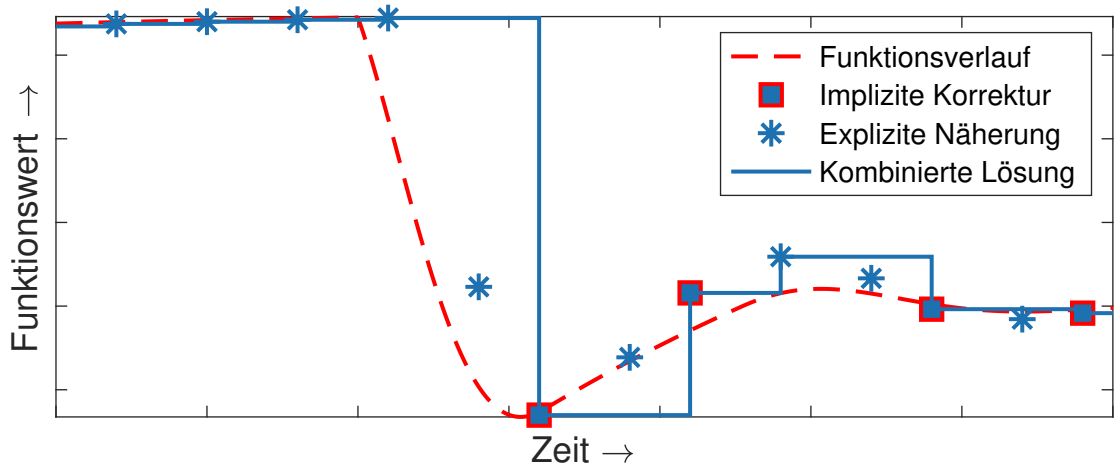

Abbildung 10: *Ein kombiniertes Lösungsverfahren mit Schrittweitenanpassung korrigiert ungenaue explizite Näherungen durch einen impliziten Schritt.*

Lösung ist in Abb. 10 wieder durch eine rote Linie dargestellt. Die blauen Sterne beschreiben alle explizit berechneten Näherungen und die roten Rechtecke zeigen die impliziten Korrekturschritte. Die blau dargestellte Stufenfunktion, verbindet alle verwendeten Näherungen und Korrekturschritte und zeigt dadurch die Momentanwertabtastung des

Ausgangs für das kombinierte Verfahren an. Unter Betrachtung der ersten vier expliziten Schritte ist zu sehen, dass die expliziten Näherungen eine ausreichend hohe Genauigkeit aufweisen. Der fünfte explizite Schritt ist der erste Schritt nach der sprunghaften Änderung des Lösungsverlaufs. Der deutlich sichtbare Fehler in diesem Schritt wird von der Fehlerabschätzung des Lösungsverfahrens erkannt und ein impliziter Korrekturschritt wird mit einer größeren Schrittweite berechnet. Der Wert des ersten Korrekturschrittes ist durch das erste rote Rechteck dargestellt. Die Korrektur scheint zunächst einen geringeren Berechnungsfehler als die explizite Näherung aufzuweisen, sodass dieser für die kombinierte Lösung verwendet wird. Der vorherige explizite Schritt wird verworfen. Da das Ausgangsignal zwischen dem verworfenem expliziten Schritt und der impliziten Korrektur, durch die Abtast-halte-Schaltung, auf dem expliziten Wert gehalten werden muss, ergibt sich eine große Abweichung zwischen der kombinierten Lösung und dem tatsächlichen Funktionsverlauf. Vergleicht man den Gewinn des genaueren impliziten Berechnungspunktes mit dem Verlust durch die Abweichung des Ausgangssignals wird deutlich, dass das Ausgangssignal mit dem verworfenen expliziten Wert durch die kleinere Schrittweite eine bessere Näherung beschreiben würde. Aus diesem Grund ist die implizite Korrektur mit Schrittweitenanpassung nur bedingt für eine HiL-Simulation geeignet.

Das dargestellte Beispiel aus Abb. 10 suggeriert, dass für eine Verbesserung des Verfahrens lediglich die verworfene explizite Näherung zusätzlich in die kombinierte Lösung aufgenommen werden muss. So würde die vorher verworfene Näherung eine bessere Verbindung zum impliziten Korrekturschritt herstellen und die kombinierte Lösung hätte eine geringe Abweichung zum Funktionsverlauf. Bei diesem Gedanken vernachlässigt man aber die Vorgaben einer HiL-Simulation. Wie bereits in Abs. 2.1 beschrieben, besteht die Latenz eines Schrittes aus der Berechnungszeit und der Zeit der Restaufgaben. Würde man den verworfenen Schritt in die kombinierte Lösung aufnehmen wollen, müsste dieser zunächst durch die Hardware gesetzt werden. Dadurch müsste zwischen dem expliziten Schritt und der Korrektur eine weitere Zeit für Restaufgaben eingeplant werden. Der Korrekturschritt würde sich dadurch verschieben und der Zeitgewinn der direkten impliziten Korrektur ginge verloren. Dadurch entstünde ein Lösungsverfahren, welches auf Grundlage des Vorschrittes zwischen implizitem und explizitem Löser wählen kann. Diese Art von Lösungsverfahren beschreibt die Grundidee des in dieser Arbeit umgesetzten Lösungsverfahrens und wird in Abs. 4 weiter ausgeführt.

3.4 Entwicklung von Analyseverfahren

Alle bisher betrachteten Optimierungsansätze basieren auf einer Fehlerabschätzung der berechneten Näherung. Die prinzipielle Funktionalität der Fehlerabschätzung für die Lösungsoptimierung von Softwaresimulationen wurde gezeigt. Optimierungsansätze auf dieser Grundlage erfordern jedoch immer unterschiedliche Berechnungsdauern und benötigen daher eine Anpassung der Schrittweite für HiL-Simulationen. Die negativen Konsequenzen für die Umsetzung eines solchen Ansatzes in einer HiL-Simulation wurde

in den vorherigen Beispielen veranschaulicht. Es zeigt sich daher, dass die Fehlerabschätzung nach der Berechnung eines Simulationsschrittes für die optimierung einer HiL-Simulation wenig nützlich ist. Hilfreicher wäre die Information über die Genauigkeit des Lösungsschrittes schon vor der Berechnung zu kennen. Auf diese Weise kann das Lösungsverfahren bereits vor der Berechnung angepasst werden, um die genauste Näherung für den nächsten Schritt zu finden. Außerdem könnten so unterschiedliche Berechnungsdauern verhindert werden und eine konstante Schrittweite gewählt werden. Eine absolute Abschätzung der Lösungsgenauigkeit vor der Berechnung lässt sich nur schwer umsetzen. Abs. 3.3.3 hat jedoch verdeutlicht, dass die Ungenauigkeit von expliziten Lösungsverfahren an der Stabilitätsgrenze und im instabilen Gebiet durch implizite Schritte korrigiert werden kann. Besitzt das Lösungsverfahren in jedem Simulationsschritt die Informationen zur Stabilitätsgrenze, könnte für diese Bereiche direkt ein implizites Verfahren gewählt werden, ohne dass eine vorherige explizite Näherung mit Fehlerabschätzung notwendig ist. Da die Berechnung der Stabilitätsgrenze während der Simulation mit einem großen Zeitaufwand verbunden ist, müssen diese Werte bereits vor der Simulation bestimmt werden. Dies ist die Aufgabe der Modellanalyse, dessen Konzept in Abs. 4.3.1 genauer beschrieben wird. Um stets eine annähernd konstante Berechnungszeit zu garantieren, sind außerdem Informationen zur Laufzeit der Lösungsalgorithmen notwendig. Hierfür wird eine Laufzeitanalyse entwickelt, welche für die Abschätzung der Berechnungsdauer verwendet wird.

4 Konzept

Auf Grundlage der definierten Anforderungen und der untersuchten Lösungsansätze wird in diesem Kapitel ein Konzept für ein optimiertes Lösungsverfahren aufgestellt. Das Grundkonzept basiert auf der in Abb. 11 dargestellten Optimierungsroutine. Die Opti-

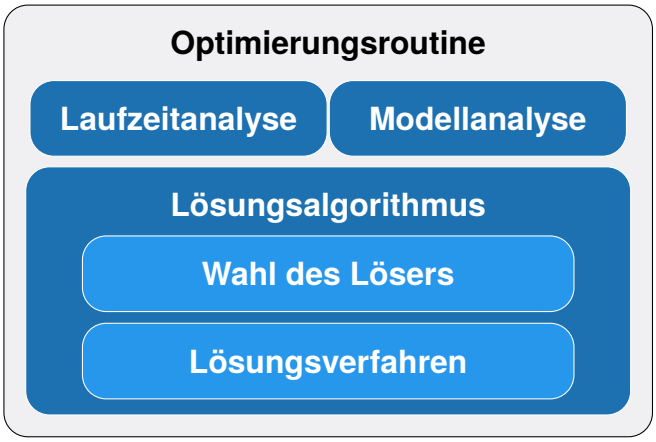

Abbildung 11: *Schematische Darstellung der Zusammensetzung der Optimierungsroutine.*

mierungsroutine ist ein allgemein formulierter Prozess, welcher die Analyse sowie die optimierte Lösung eines Modells übernimmt. Hierfür greift die Optimierungsroutine auf zwei Analyseverfahren zurück. Der Lösungsalgorithmus ist die Implementierung des optimierten Lösers, welcher auf Grundlage der Analyseinformationen ein Lösungsverfahren auswählt und mit diesem die Simulation durchführt. Im Folgenden werden die einzelnen Komponenten genauer beschrieben. Hierfür wird zunächst der Aufbau der Optimierungsroutine dargestellt und das Konzept für die allgemeine Darstellungsform eines Modells entwickelt. Anschließend folgt die Herleitung der Analyseverfahren, sowie des Lösungsalgorithmus.

4.1 Aufbau der Optimierungsroutine

Grundsätzlich besteht die Optimierungsroutine aus zwei Analyseschritten und dem Lösungsalgorithmus. Die erste Analyse ist die Modellanalyse, welche parameterabhängige Informationen über das Modell gewinnt. Die zweite Analyse bezieht sich auf den Umgang des Lösungsverfahrens mit dem Modell und ist von den Parametern unabhängig. Alle Analyseinformationen und Parameter werden zusammen mit dem Modell an den Lösungsalgorithmus übergeben, welcher ein Lösungsverfahren mit optimaler Schrittweite wählt. Abb. 12 beschreibt strukturell das Grundkonzept des Ablaufs. Wie auf Abb. 12 zu sehen, liegen die Simulationsparameter und das Modell für eine parameterabhängige Gestaltung getrennt vor. Bei jeder Anpassung wird das Modell zusammen mit den neuen Parametern an die Modellanalyse übergeben. Die Modellanalyse bestimmt dann parameterabhängige

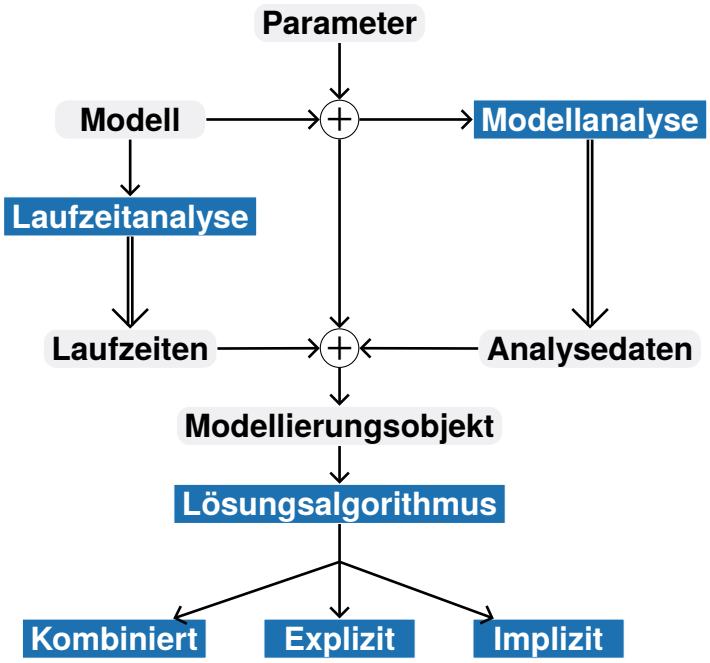

Abbildung 12: *Konzeptskizze des Ablaufs der Optimierungsroutine. Funktionen sind blau und Daten grau dargestellt.*

Modelleigenschaften, welche Aufschluss über die Steifigkeit des Systems geben. Diese Informationen werden zusammen mit dem Modell und seinen Parametern in einem Modellierungsobjekt gespeichert. Hinzu kommen die Daten aus der Laufzeitanalyse, welche die modellabhängigen Berechnungszeiten für die verschiedenen Lösungsverfahren enthalten. Auf Grundlage dieser Informationen kann der Lösungsalgorithmus eine geeignete Schrittweite für das jeweilige Lösungsverfahren wählen. Für die Simulation wird das Modellierungsobjekt mit seinen Informationen an den Lösungsalgorithmus übergeben. Hier werden die in den Analysen gesammelten Daten verarbeitet und ein für die Genauigkeit optimiertes Lösungsverfahren gewählt.

4.2 Modellgestaltung

Die Anforderungen an die Modellgestaltung fordert die allgemeine Darstellungsmöglichkeit von Systemen unterschiedlicher Komplexität und Größe. Eine vollständige Modelldefinition beinhaltet insgesamt folgende Informationen:

1. Systembeschreibende DGL
2. Jacobi-Matrizen des Systems
3. Zustandsdefinitionen und Schaltbedingungen
4. Algebraische Ausgangsgleichung

Für die Darstellungsform der DGL bietet sich die Funktionsdarstellung aus Gl. (6) an. Mit der Funktionsdarstellung können sowohl lineare als auch nichtlineare Gleichungen allgemein dargestellt werden. Die Funktionen liegen hierbei in Abhängigkeit der Systemvariablen y, der Eingangsgrößen u und der Parameter P vor. Auch wenn die Parameter während der Simulation konstant sind, ist diese Abhängigkeit für die parameterabhängige Modelldefinition notwendig. In der mathematischen Darstellung wird auf den Hinweis der Abhängigkeit von P jedoch verzichtet. Da die meisten Modelle aus DGL Systemen bestehen, werden Funktionsvektoren verwendet. Die Definition der Funktionsvektoren erfolgt für jeden Systemzustand und bildet so ein Zustandsfeld. Für ein Modell mit k Zustandsvariablen $y = [y_1, \cdots, y_k]^\top$, q Zuständen $z = [z_1, \ldots, z_q]^\top$ und der Eingangsgröße u ist die Definition des Zustandsfeldes $\mathscr{F}(z)$ gegeben durch:

$$\mathscr{F}(z) = \Big\{F_1(t,y,u), \cdots, F_q(t,y,u)\Big\} \tag{42}$$

Das Zustandsfeld wählt in Abhängigkeit des Zustandes z den jeweiligen Funktionsvektor F_j als aktives DGL System. Jedes DGL System ist dabei definiert durch:

$$F_j(t,y,u) = \begin{bmatrix} f_{1,j}(t,y,u) \\ \vdots \\ f_{k,j}(t,y,u) \end{bmatrix} \quad \text{mit} \quad j = [1,\ldots,q] \tag{43}$$

Jedes $f_{i,j}$ beschreibt die DGL der Zustandsvariablen y_i für $i = [1,\ldots,k]$.

In ähnlicher Weise ergeben sich die Definitionen der Jacobi-Matrizen. Für jeden Zustand wird die Jacobi-Matrix J_{Fj} des zugehörigen Funktionsvektors F_j definiert. Auch die Jacobi-Matrizen liegen in Abhängigkeit von y, u und P vor. So ist es möglich für jedes Wertepaar, eine ausgewertete Jacobi-Matrix zu erzeugen.

Die Grundlegenden Eigenschaften des Modells sind durch die Definitionen der DGL und Jacobi-Matrizen gegeben. Hinzu kommt die Zustandsüberwachung, eine Funktion zur Definition der Schaltbedingungen für die Zustandswechsel. Die Schaltbedingungen sind in Abhängigkeit der aktuellen Systemvariablen y und u definiert. Bei Erfüllung einer Schaltbedingung, führt die Zustandsüberwachung einen Zustandswechsel durch. Zum Beispiel wird bei dem Erreichen eines Schwellenwertes für y_1, der aktive Zustand von z_1 zu z_2 geändert. In der Zustandsüberwachung werden zusätzlich auch unstetige Funktionsverläufe gesetzt. So kann z. B. die Zustandsvariable y_1 bei Erreichen eines Schwellenwertes auf diesem Wert konstant festgehalten werden, obwohl die DGL in diesem Zustand einen anderen Funktionsverlauf vorsieht.

Die Darstellung der DGL sollte so einfach wie möglich gehalten werden, um den Rechenaufwand des numerischen Näherungsverfahrens zu verringern. Im Zusammenhang mit der Zustandsraumdarstellung aus Abs. 2.2.3 wurde das Prinzip der Ausgangsgleichung vorgestellt. Eine Ausgangsgleichung ermöglicht es, aus den berechneten Zustandsvariablen y, die Ausgangsvariablen y_{out} zu bestimmen. Dadurch können algebraische Gleichungen

direkt auf die Zustandsvariablen y angewendet werden. Eine unnötige Komplizierung durch die Implementierung der algebarischen Gleichungen in die Funktionsdarstellung der DGL wird so vermieden. Die Definition der Ausgangsgleichung erfolgt als Funktion in Abhängigkeit der Zustands- und Eingangsvariablen, sowie des aktuellen Zustandes.

4.3 Analysekonzept

Das Analysekonzept besteht aus zwei Analyseschritten. Die parameterhabhängige Modellanalyse bildet die Grundlage für die Optimierung des Lösungsverfahrens und untersucht die Steifigkeit des Systems in jedem Zustand. Die Laufzeitanalyse untersucht die Schnelligkeit des Lösungsalgorithmus angewendet auf das gewählte Modell. Beide Analyseverfahren werden in diesem Abschnitt nacheinander vorgestellt.

4.3.1 Modellanalyse

Wie in Abs. 2.2.3 bereits erläutert, geben die Eigenwerte des Systems Aufschluss über die Steifigkeit des Systems. Die Modellanalyse berechnet die Eigenwerte des Modells und bestimmt daraus die zustandsabhängigen Stabilitätsgrenzen. Die Modellanalyse lässt sich daher in zwei Schritte unterteilen, welche im Folgenden zur Vereinfachung getrennt betrachtet werden. Der erste Schritt der Analyse ist die Eigenwertbestimmung, im zweiten Schritt werden die Stabilitätsgrenzen mithilfe der Eigenwerte berechnet.

Eigenwertbestimmung

Die Systemeigenwerte von linearen Problemen sind durch die Eigenwerte der Systemmatrix A leicht zu berechnen. Die Bestimmung der Systemeigenwerte von nichtlinearen Problemen gestaltet sich jedoch etwas schwieriger. Da die Optimierungsroutine auf lineare und nichtlineare Probleme anzuwenden ist, wird eine allgemein anwendbare Methode für beide Problemarten benötigt. Im Folgenden wird daher die Eigenwertbestimmung nichtlinearer Probleme betrachtet.

Wie Abs. 2.2.3 beschreibt, lassen sich die Eigenwerte eines nichtlinearen Systems durch die Linearisierung an einem Entwicklungspunkt bestimmen. Das lokale Einfrieren der Koeffizienten bildet das Systemverhalten jedoch nur für den Moment des Entwicklungspunkts ab. Um daher ausreichend Informationen über ein nichtlineares System zu erhalten, wird für jedes System ein diskreter Arbeitsbereich betrachtet. Der Arbeitsbereich definiert sich als der mögliche Wertebereich aller Zustandsvariablen. Es wird vorausgesetzt, dass der mögliche Wertebereich der Zustandsvariablen vor der Lösung des Modells bekannt ist. Diese Annahme sollte bei der Simulation der meisten physikalischen Systeme gültig sein.

Angenommen es wird ein System mit zwei Zustandsvariablen y_1 und y_2 betrachtet. Mit einem bekannten Wertebereich ist der Arbeitsbereich gegeben durch:

$$y_{1,\min} \leq y_1 \leq y_{1,\max} \tag{44}$$

$$y_{2,\min} \leq y_2 \leq y_{2,\max} \tag{45}$$

Der Arbeitsbereich des Problems wird durch einen zweidimensionalen Raum aufgespannt. Für die Simulation eines kleinen Elektromotors könnte der Arbeitsbereich z. B. durch die Strom- und Drehzahlgrenzen definiert werden. Wird der Arbeitsbereich nun mit einer Auflösung r diskretisiert, ergibt sich für das System ein Feld von r^2 Arbeitspunkten. Allgemein ergibt sich für ein Problem mit k Systemvariablen ein k-dimensionaler Raum mit r^k Arbeitspunkten. In die k Systemvariablen müssen auch die veränderlichen Eingangsgrößen gezählt werden, welche mit nichtlinearem Zusammenhängen in den DGL dargestellt sind. Die Auflösung r ist dabei Ausschlaggebend für die Genauigkeit der Linearisierung. Durch das Einsetzen der Arbeitspunkte in die Jacobi-Matrix des Systems, lässt sich das Problem an jedem Arbeitspunkt einfrieren und jeweils eine linearisierte Systemmatrix A bestimmen. Für jede der r^k Systemmatrizen können dann die k Eigenwerte berechnet werden. Mit $k = 2$ und einem reellen Eigenwertpaar für jeden Arbeitspunkt, kann jeweils λ_1 und λ_2 in einem dreidimensionalen Raum dargestellt werden. Für komplexe Eigenwerte kommt eine weitere Dimension für die imaginäre Achse hinzu.

Um das beschriebene Verfahren zur Eigenwertbestimmung auch auf lineare Systeme anwenden zu können, muss lediglich die Auflösung $r = 1$ gesetzt werden. Da die Jacobi-Matrix eines linearen Problems konstant ist und genau die Systemmatrix A ergibt, wird eine Auswertung an mehreren Arbeitspunkten nicht benötigt.

Bestimmung der Stabilitätsgrenzen

Das Ziel der Modellanalyse ist die Untersuchung der Modellsteifigkeit durch die Betrachtung der Stabilitätsgrenzen. Dadurch soll es dem Lösungsalgorithmus ermöglicht werden, für jeden Simulationsschritt ein stabiles und genaues Lösungsverfahren zu wählen. Obwohl ein eindeutiger Zusammenhang zwischen den Systemeigenwerten und der Steifigkeit besteht (vgl. Abs. 2.4.1), ist es nicht möglich ein absolutes Kriterium für ein steifes System, ausschließlich auf Grundlage der Eigenwerte aufzustellen [13]. Die Bewertung der Steifigkeit erfolgt daher für jeden Arbeitspunkt unter Betrachtung der Stabilitätsgrenze des verwendeten Lösungsverfahrens. Die verwendete Schrittweite des Lösungsverfahrens wird mit der aktuellen Stabilitätsgrenze verglichen um eine Einschätzung der Effizienz des Lösungsverfahrens zu ermöglichen. Im Folgenden wird die Berechnung dieser Stabilitätsgrenzen genauer erläutert.

Für die Herleitung eines allgemeinengültigen Verfahrens zur Bestimmung der Stabilitätsgrenze, wird die in Abs. 2.4.2 beschriebene Stabilitätsfunktion eines Lösungsverfahrens

betrachtet. Zusammenfassend muss für ein stabiles Lösungsverfahren mit der Stabilitätsfunktion $R(z)$ gelten:

$$y_{n+1} = R(z) \cdot y_n$$

$$\text{mit} \quad |R(z)| \leq 1 \quad \text{und} \quad z = h \cdot \lambda \tag{46}$$

In Abs. 2.4.2 wurde aus dieser Anforderung bereits das Stabilitätsgebiet eines Lösungsverfahrens hergeleitet. Die Stabilitätsgrenze des Stabilitätsgebiets ergibt sich aus der Betrachtung der Grenzstabilität:

$$|R(z)| = 1 \quad \text{mit} \quad z = h \cdot \lambda \tag{47}$$

Die Stabilitätsgrenze eines gegebenen Eigenwerts λ, für ein Lösungsverfahren mit bekannter Stabilitätsfunktion $R(z)$, leitet sich durch das Einsetzen des Eigenwerts in Gl. (47) und Umstellen zur Schrittweite h her. Die Lösung für h beinhaltet die größtmögliche Schrittweite $h_{\max}$, mit der das Lösungsverfahren eine stabile Lösung finden kann. Anschaulicher wird dies mit der Betrachtung eines konkreten Beispiels. Hierfür sei die Stabilitätsfunktion des klassischen Runge-Kutta Verfahrens vierter Ordnung gegeben mit [8]:

$$R(z) = \frac{1}{24}z^4 + \frac{1}{6}z^3 + \frac{1}{2}z^2 + z + 1 \tag{48}$$

Das Stabilitätsgebiet dieser Funktion ist ist in Abb. 13 dargestellt und beschreibt den Wertebereich der Funktion aus Gl. (48), für den Gl. (46) gilt. Da eine korrekte Lösung für die Schrittweite h nur eine positive reelle Zahl sein kann, wirkt sich h wie ein Skalierungsfaktor aus, der die Eigenwerte auf einer Geraden in Richtung Nullpunkt verschieben kann. Betrachten wir den willkürlich gewählten komplex Eigenwert $\lambda_1 = -3 + i3$ und den realen Eigenwert $\lambda_2 = -4$ zeigt sich, dass der erste Schnittpunkt von Gerade und Stabilitätsgebiet die Lösung für die grenzstabile Schrittweite $h_{\max}$ ist. Einsetzen von $\lambda_1 = -3 + i3$ in Gl. (48) und Gl. (47) führt zu:

$$1 = \left| \frac{1}{24}(-4 \cdot h)^4 + \frac{1}{6}(-4 \cdot h)^3 + \frac{1}{2}(-4 \cdot h)^2 + (-4 \cdot h) + 1 \right| \tag{49}$$

$$\Leftrightarrow 1 = \left| -\frac{27}{2}h^4 + (9 + 9i) \cdot h^3 - 9i \cdot h^2 - (3 - 3i) \cdot h + 1 \right| \tag{50}$$

Die größtmögliche Schrittweite für eine stabile Lösung des Systems mit dem Eigenwert $\lambda_1 = -3 + i3$ ist die größte reale Lösung von Gl. (50) und gegeben durch $h_{1,\max} = 0{,}64$. Diese Schrittweite verschiebt, wie auf Abb. 13 zu sehen, den Eigenwert λ_1 auf den Punkt $z_1 = -1{,}9 + 1{,}9i$. Es zeigt sich, dass alle reellen Eigenwerte auf den gleichen Punkt z_2 der Stabilitätsgrenze bezogen werden, sodass die maximale Schrittweite für reelle Eigenwerte auch durch die Division mit einem konstanten Faktor berechnet werden kann. Für $\lambda_2 = -4$

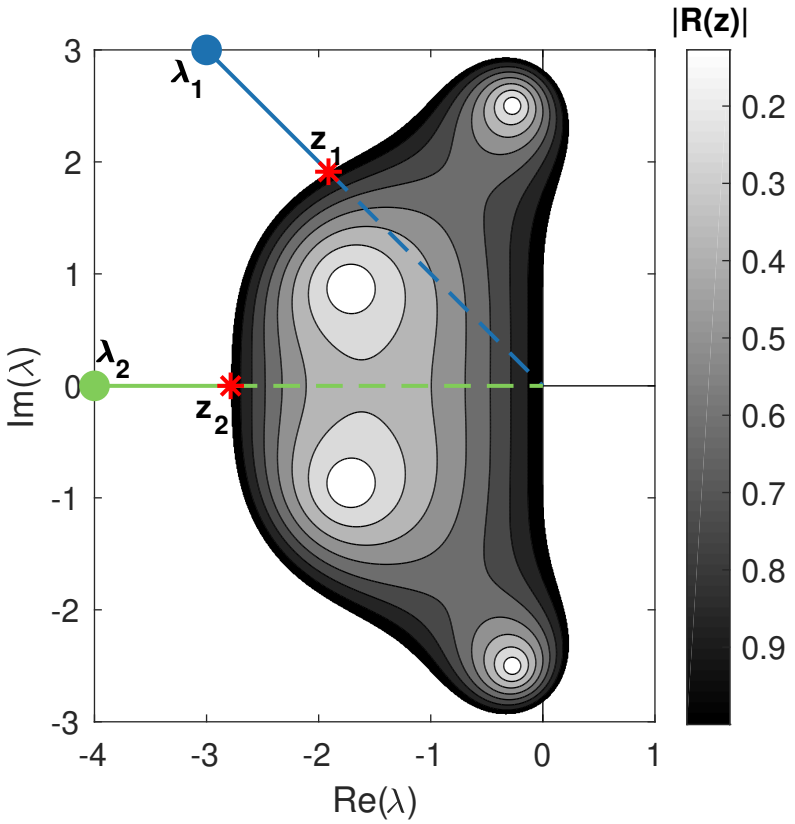

Abbildung 13: *Stabilitätsgebiet des klassischen Runge-Kutta Verfahrens. Die Eigenwerte λ_1 und λ_2 werden auf den Ursprung bezogen, um die zugehörigen Punkte z_1 und z_2 auf der Stabilitätsgrenze zu bestimmen.*

ergibt sich durch Einsetzen in Gl. (48) und Gl. (47)

$$1 = \left| \frac{32}{3} h^4 - \frac{32}{3} h^3 + 8 \cdot h^2 - 4 \cdot h + 1 \right| \tag{51}$$

$$\Rightarrow h_{2,\max} = 0{,}7 \tag{52}$$

Der Punkt z_2 auf der reellen Achse bestimmt sich dann aus:

$$z_2 = z_{real} = \lambda_2 \cdot h_{2,\max} \approx -2{,}7 \tag{53}$$

Zu jedem reellen Eigenwert λ_{real} ergibt sich dann die maximale Schrittweite aus:

$$h_{\max}(\lambda_{real}) = \frac{z_{real}}{\lambda_{real}} \tag{54}$$

Bei der Bestimmung der maximalen Schrittweite für komplexe Eigenwerte, muss die Gl. (47) jedes Mal gelöst werden. Da für gekoppelte DGL Systeme stets die kleinste Stabilitätsgrenze eingehalten werden muss, reicht es aus an jedem Arbeitspunkt nur die betragsmäßig größten Eigenwerte der Eigenwertpaare zu betrachten.

Wird die Bestimmung der Stabilitätsgrenze auf jeden Eigenwert im Arbeitsbereich angewendet, ergibt sich ein Feld von grenzstabilen Schrittweiten. Zur Veranschaulichung zeigt Abb. 14 das Ergebnis der Schrittweitenanalyse für das Modell einer einfachen Halogenlampe. Die X- und Y-Achse zeigen den Arbeitsbereich der Systemvariablen von Strom und Temperatur. Die Z-Achse zeigt die Größe der grenzstabilen Schrittweiten für jeden Arbeitspunkt. Die gezeigten Schrittweiten wurden aus den Eigenwerten für die DGL des Stroms, bei einer angelegten Spannung von 12V bestimmt. Aus der Darstellung der Stabilitätsgren-

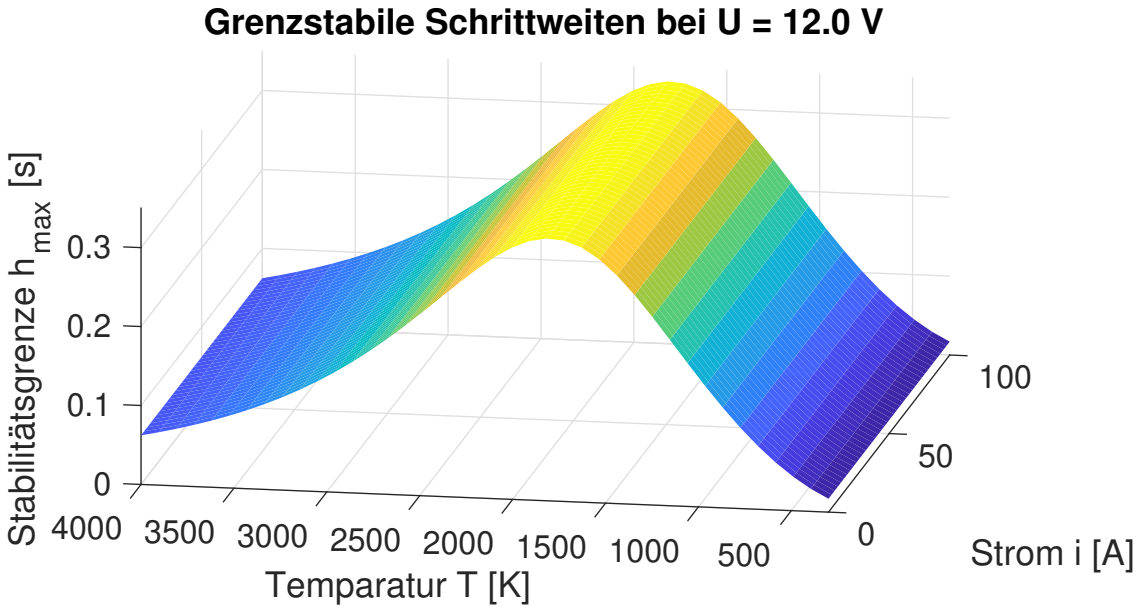

Abbildung 14: *Dreidimensionale Darstellung der Änderung der Stabilitätsgrenze für das Modell einer Halogenlampe. Während eine Temperaturänderung einen starken Einfluss auf die Stabilitätsgrenze hat, bleibt die Änderung des Stroms ohne Einfluss.*

zen wird deutlich, dass es einen Temperaturbereich gibt in dem die DGL des Stroms mit einer deutlich größeren Schrittweise gelöst werden kann als im restlichen Arbeitsbereich. Da der Betriebspunkt der Halogenlampe in diesem Bereich liegt, würde für die meisten Simulationen eine Schrittweite in der Nähe dieser Stabilitätsgrenze ausreichen. Für eine stabile, rein explizite Simulation muss jedoch der gesamte Arbeitsbereich betrachtet werden und die gewählte Schrittweite müsste daher unterhalb des kleinsten Wertes von $h_{\max}$ gewählt werden. Das gewählte Beispiel dient nur der Veranschaulichung des Problems, tatsächlich wird die Stabilitätsgrenze der Halogenlampe durch die gekoppelte DGL der Temperaturänderung bestimmt, welche eine deutlich kleinere Stabilitätsgrenze besitzt.

4.3.2 Laufzeitanalyse

Eine der wichtigsten Anforderungen in einer HiL-Simulation ist die Echtzeitfähigkeit der Simulation. Um die Echtzeitanforderung für jedes Modell und jedes Lösungsverfahren erfüllen zu können, muss die Latenz eines Schrittes kürzer sein als die Schrittweite h. Wie

anfangs bereits erwähnt, setzt sich die Latenz aus der Berechnungszeit und den zusätzlichen Hard- und Softwareaufgaben zusammen. Abbildung 15 zeigt die Zusammensetzung der Latenz Δt_{sim} für explizite und implizite Verfahren. Die reine Berechnungszeit Δt_{step}

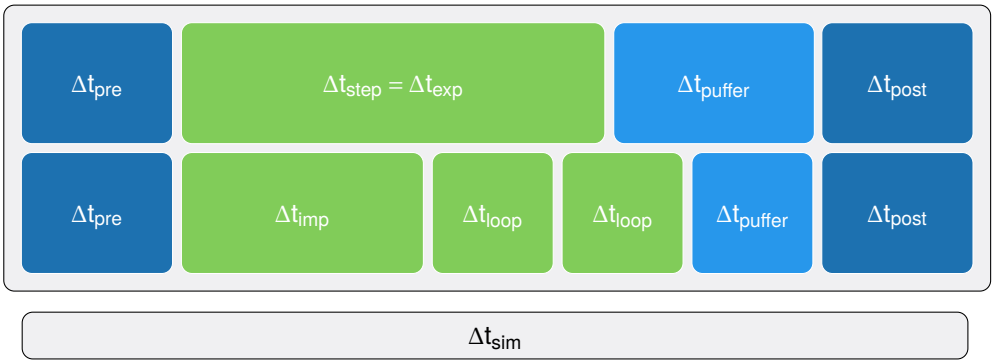

Abbildung 15: *Zusammensetzung der Latenz Δt_{sim}.*

ist die benötigte Zeit der Simulationssoftware zur Bestimmung der Werte eines Simulationsschrittes. Diese ist für ein explizites Verfahren durch Δt_{exp} gegeben. Ein implizites Verfahren besteht aus einem initialen Berechnungsschritt und möglicherweise mehreren Iterationsschleifen. Die Berechnungszeit des impliziten Schrittes setzt sich daher aus dem ersten Berechnungsschritt Δt_{imp} und der Summe der darauf folgenden Iterationsschleifen Δt_{loop} zusammen. Die Zeit der Restaufgaben Δt_{rest} setzt sich zusammen aus hardwarespezifische Aufgaben und Pufferzeiten, welche im Simulationsverlauf als annähernd konstant angenommen werden können. Vereinfacht kann Δt_{rest} bestimmt werden aus:

$$\Delta t_{\text{rest}} = \Delta t_{\text{pre}} + \Delta t_{\text{puffer}} + \Delta t_{\text{post}} \tag{55}$$

Δt_{pre} dient zur Vorbereitung der Berechnung und beinhaltet alle Aufgaben zum Lesen und Übertragen der Messwerte. Analog dazu beschreibt Δt_{post} die Nachbereitung der berechneten Werte, wie z. B. das Übertragen der Werte an die Leistungselektronik. Dazwischen liegt die Pausezeit Δt_{puffer}, welche die Einhaltung der Berechnungsdeadline absichert. Der Sicherheitspuffer ist nach dem Berechnungsschritt geplant und fängt unerwartete Verzögerungen in der Berechnung ab und sorgt so für eine konstante Latenz Δt_{sim}.

Ziel der Laufzeitanalyse ist die Abschätzung der gesamten Durchführungsdauer eines Simulationsschrittes Δt_{sim}. Ist die Berechnungszeit des Simulationsschrittes bekannt, kann eine echtzeitfähige Schrittweite bestimmt werden. Die Laufzeitanalyse ist unabhängig von den Parametern und analysiert das Zusammenspiel von Lösungsverfahren und Simulationsmodell vor dem Simulationsstart. Der reine Berechnungsschritt, mit der Berechnungsdauer Δt_{step}, ist abhängig von Modell und Lösungsverfahren und wird in der Laufzeitanalyse für jedes Modell abgeschätzt. Im Folgenden wird das Verfahren zur Abschätzung von Δt_{step} vorgestellt, um anschließend die Bestimmung der Schrittweite beschreiben zu können.

Das Vorgehen der Laufzeitanalyse ähnelt einem Benchmarking-Test.[6] Mit jedem mög-

lichen Lösungsverfahren wird eine feste Anzahl an Schritten ausgeführt und durch eine Zeitmessung die durchschnittliche Berechnungsdauer Δt_{step} für einen Schritt bestimmt. Die Berechnungsdauer ist zwar unabhängig von den Zahlenwerten der Zustandsvariablen, kann sich jedoch zwischen den Zuständen unterscheiden. Daher wird die Messungen von jedem Löser für jeden Zustand durchgeführt. Für die Bestimmung der Berechnungszeit eines Simulationsschrittes wird dann der schlechteste Zustand mit der größten Laufzeit angenommen. Für explizite Verfahren bildet diese Messung einen Richtwert für die Simulationsdauer eines Schrittes. Für ein implizites Verfahren muss zusätzlich die Laufzeit einer Iterationsschleife Δt_{loop} gemessen werden. Wird die Anzahl von Iterationsschritten im Lösungsverfahren durch I_{max} begrenzt, so bestimmt sich die Berechnungsdauer Δt_{step} für ein implizites Verfahren aus:

$$\Delta t_{\text{step}} = \mathrm{I}_{\text{max}} \cdot \Delta t_{\text{loop}} + \Delta t_{\text{imp}} \tag{56}$$

Die Schrittweite darf nicht kleiner sein, als die gesamte Simulationsdauer eines Schrittes, sodass für die Schrittweite h gilt:

$$h \quad \geq \quad \Delta t_{\text{sim}} \tag{57}$$

4.4 Lösungsalgorithmus

Der Lösungsalgorithmus der Optimierungsroutine implementiert die Wahl und Ausführung des Lösungsverfahrens. Hierfür wird das Simulationsmodell mit allen Daten aus der Modell- und Laufzeitanalyse an den Lösungsalgorithmus übergeben. Zur Aufstellung des Konzepts vom Lösungsalgorithmus werden zunächst die möglichen Lösungsverfahren und anschließend das Auswahlverfahren vorgestellt. Alle verwendeten Lösungsverfahren basieren auf der Familie der RK-Verfahren. Diese ermöglichen eine einfache Implementierung des implizit und explizit kombinierten Lösungsverfahrens. Die Verwendung von Mehrschrittverfahren wird aufgrund ihrer aufwändigeren Implementierung im Folgenden zunächst nicht weiter verfolgt. Die Familie der RK-Verfahren bildet jedoch eine gute Entwicklungsgrundlage, für welche die Verbesserungsmöglichkeiten durch Mehrschrittverfahren, in einer späteren Betrachtung geprüft werden können. Allgemein stehen dem Lösungsalgorithmus drei Lösungsverfahren zur Verfügung, die das Modell rein explizit, rein implizit und mit einem kombinierten Verfahren lösen können.

4.4.1 Schrittweitenwahl

Die Wahl einer geeigneten Schrittweite für eine HiL-Simulation gestaltet sich durch die speziellen Anforderungen anders als bei einer reinen Softwaresimulation. Während für Softwaresimulationen meist ein Optimum zwischen Genauigkeit und gesamter Simulationsdauer gefunden werden soll, ist die gesamte Simulationsdauer einer HiL-Simulation durch die Echtzeitanforderung vorgegeben. Die Optimierung der Genauigkeit bewegt sich

hier im stark limitierten Rahmen der Umsetzbaren Schrittweiten. Zur Wahl der optimalen Schrittweite betrachtet der Lösungsalgorithmus daher die geltenden Limitierungen. Im Folgenden wird dieser Prozess genauer dargestellt.

Die obere Grenze der Schrittweite ist durch die Genauigkeitsanforderungen an die Simulation gegeben. Bei expliziten Verfahren kommt zusätzlich die obere Begrenzung durch die kleinste Stabilitätsgrenze $\min(h_{\max})$ des Modells hinzu.

Wie Abb. 15 zeigt, setzt sich die Schrittweite einer HiL-Simulation aus der Berechnungsdauer Δt_{step} und den nicht direkt beeinflussbaren Zeiten für die Restaufgaben Δt_{rest} zusammen. Die kleinstmögliche Schrittweite h_{min} ist daher durch die Zeit der Restaufgaben und der Berechnungsdauer des Lösungsverfahrens vorgegeben. Die Wahl der optimalen Schrittweite bewegt sich daher im Rahmen der aufgezeigten Grenzen. Zur Maximierung der Genauigkeit gilt es nun, die möglichen Fehler bei der Wahl der Schrittweite zu betrachten. Hier spielt der Diskretisierungsfehler, der Rundungsfehler, sowie die Abweichung des Ausgangssignals durch die Momentanwertabtastung eine Rolle. Da der Diskretisierungsfehler mit kleinerer Schrittweite sinkt (vgl. Abs. 2.5.1) und die Abtastastung des Ausgangsignals eine besserer Kontinuität aufweist (vgl. Abs. 2.1), muss lediglich der Rundungsfehler für die kleinstmögliche Schrittweite betrachtet werden. Ist der Rundungsfehler für die kleinstmögliche Schrittweite vernachlässigbar klein, ist diese die Schrittweite mit der höchsten Genauigkeit. Für die Umsetzung der Optimierungsroutine muss daher eine Abschätzung des größten Rundungsfehlers erfolgen.

Abs. 2.5.2 zeigt, dass der Rundungsfehler für Verfahren mit kleiner Schrittweite größer wird. Da explizite Verfahren, durch ihre geringe Berechnungszeit die kleinsten Schrittweiten in HiL-Simulationen ermöglichen, wird der Rundungsfehler für das klassische RK-Verfahren vierter Ordnung betrachtet. Die Latenz der betrachteten HiL-Umgebung, für einen Berechnungsschritt durch das klassische gls RK-Verfahren verfahren, beträgt ungefähr $\Delta t_{\text{sim}} \approx 10\mu s$. Diese Zeit gibt auch die kleinste realisierbare Schrittweite $h_{min} = 10\mu s$ vor. Der Lösungsalgorithmus verwendet den Datentyp `double` (Gleitkommazahl mit doppelter Genauigkeit[5]) zur Bestimmung der Näherung. Dieser Datentyp belegt 64 Bit für eine Zahl und besitzt dadurch eine Genauigkeit von ca. 15 Stellen [5]. Da das in Gl. (16) dargestellte klassische RK-Verfahren vierter Ordnung, lediglich auf dem linearen Anteil der Taylorentwicklung beruht, beinhaltet die Näherung keine Potenz von h. Es kann angenommen werden, dass die Summe der Stufen $\sum k$, sowie die Zustandsvariablen y_n und y_{n+1} die gleiche Größenordnung besitzen. Daher gilt:

$$\|y_n\| \approx \|y_{n+1}\| \approx \left\|\sum k\right\| \tag{58}$$

Wird Gl. (58) auf Gl. (16.a) angewendet, ergibt sich für die Abschätzung des Rundungsfeh-

lers:

$$\begin{aligned} \|y_{n+1}\| &\approx \|y_n\| + \frac{h}{6} \cdot \left\| \sum k \right\| \\ &\approx 1.0 + \frac{10^{-5}}{6} \end{aligned} \tag{59}$$

Mit der Genauigkeit der Gleitkommazahl von 15 Stellen gehen bei der Addition aus Gl. (59), die letzten fünf Stellen des letzten Terms verloren. Die Näherung trägt daher zehn Stellen zur Lösung bei. Da die Hardwareausgabe aus der hier betrachteten Beispielumgebung eine Auflösung von $1mA$ besitzt (d.h. auf drei Nachkommastellen genau ist) ist eindeutig, dass der Rundungsfehler eine deutlich kleinere Größenordnung besitzt und nicht betrachtet werden muss. Die Schrittweite mit der größten Genauigkeit ergibt sich daher aus der kleinsten realisierbaren Schrittweite.

4.4.2 Explizites Lösungsverfahren

Da das explizite Lösungsverfahren einen geringeren Rechenaufwand besitzt als das implizite, lässt sich mit diesem Verfahren eine kleinere Berechnungsdauer Δt_{step} erzielen. Eine kürzere Berechnungsdauer lässt eine kleine Schrittweite h zu, sodass der Diskretisierungsfehler und die Momentanwertabweichung abnimmt und das explizite Verfahren eine höhere Genauigkeit bietet als das implizite. Dies gilt natürlich nur solange sich das explizite Verfahren im Stabilitätsgebiet befindet.

Da die Berechnungsdauer mit der Konsistenzordnung zunimmt, lässt sich die Schrittweite des expliziten Verfahrens, durch die Wahl eines Verfahrens erster Ordnung, weiter verkleinern. Die kleinstmögliche Schrittweite ist jedoch durch die Restaufgabenzeit Δt_{rest} begrenzt. Die Auswirkung der Reduzierung der Konsistenzordnung hat daher nicht unbedingt eine ausschlaggebende Auswirkung auf die Schrittweite. Für die im Rahmen dieser Arbeit betrachtete HiL Umgebung ist die Restaufgabenzeit so groß, dass die Latenz zwischen einem Verfahren vierter Ordnung zu einem Verfahren erster Ordnung einen verhältnismäßig kleinen Unterschied aufweist. Doch auch die Vergrößerung der Konsistenzordnung ist nur zu einem gewissen Grad sinnvoll. Da die Stufenzahl N eines expliziten RK-Verfahrens schneller wächst als die Konsistenzordnung p (vgl. Abs. 2.5.1), führt eine höhere Ordnung nicht automatisch zu einem besseren Ergebnis. Zum Beispiel benötigt ein RK-Verfahren fünfter Ordnung bereits eine Auswertung von sechs Stufen[14]. Eine gutes Kosten/Nutzen-Verhältnis bietet ein vierstufiges RK-Verfahren mit vier Funktionsauswertungen. Es hat sich gezeigt das die benötigte Berechnungszeit für das Verfahren vierter Ordnung in einem ähnlichen Zeitbereich liegt wie die benötigte Restaufgabenzeit. Eine Verringerung der Ordnung hätte daher nur einen geringen Einfluss auf die Schrittweite. Anhand eines Beispiels wurden die Berechnungszeiten vom RK-Verfahren mit dem

FE-Verfahren verglichen. Für das gewählte Beispiel ist mit dem RK-Verfahren vierter Ordnung eine Schrittweite von $10\mu s$ und mit dem FE-Verfahren eine Schrittweite von $7{,}5\mu s$ realisierbar. Für die in dieser Arbeit betrachteten Modelle hat sich daher ein explizites RK-Verfahren vierter Ordnung als optimal erwiesen. Der zeitliche mehraufwand von $2{,}5\mu s$ wird in Kauf genommen um eine Erhöhung der Konsistenzordnung von $p = 1$ auf $p = 4$ zu erreichen.

Das hier verwendete explizite Verfahren bietet ein schnelles, einfaches Verfahren, welches durch die Wahl einer kleinen Schrittweite, eine hohe Genauigkeit aufweist. Kommt das Lösungsverfahren in einen Grenznahen Bereich, sinkt die Genauigkeit und es können Oszillationen bei z. B. sprunghaften Lösungsänderungen auftreten. Unterschreitet die grenzstabile Schrittweite $h_{\max}$ die Simulationsschrittweite, sodass $h_{\max} < h$, wird die Lösung instabil und die Simulation führt zu einem unbrauchbaren Ergebnis. Aus diesem Grund hat der Lösungsalgorithmus die Möglichkeit einen impliziten oder den kombinierten Löser zu wählen.

4.4.3 Implizites Lösungsverfahren

Wie in Abs. 2.4.2 beschrieben besitzten implizite RK-Verfahren ein A-stabiles Stabilitätsgebiet. Dadurch eignen sich implizite Verfahren besonders für steife Systeme und Systeme mit kleinen Zeitkonstanten. Ein Modell, welches mit dem expliziten Lösungsverfahren nicht stabil gelöst werden kann, lässt sich durch das implizite Verfahren lösen. Der größte Nachteil der impliziten Lösung ist, dass die implizite Berechnungszeit deutlich größer ist und so eine größere Schrittweite gewählt werden muss.

Hinzu kommt, dass implizite Verfahren aufgrund der Echtzeitanforderungen nur bedingt für HiL-Simulationen geeignet sind. Durch die Iterationsschleifen kann die Berechnungsdauer eines Schrittes von sehr unterschiedlicher Länge sein. Durch das Begrenzen der Iterationen kann zwar die Echtzeitanforderung eingehalten werden, jedoch nur zum Leidwesen der Genauigkeit.

Der rein implizite Löser ist daher nur für Systeme sinnvoll, die sich im größten Teil der Simulation außerhalb der Stabilitätsgrenzen des expliziten Lösers befinden. Systeme die nur manchmal in einen instabilen Bereich geraten, lassen sich am besten durch das als nächstes vorgestellte kombinierte Lösungsverfahren lösen.

4.4.4 Kombiniertes Lösungsverfahren

Das kombinierte Lösungsverfahren ist eine Verknüpfung von expliziter und impliziter Näherung. Der Löser nutzt die Vorteile von beiden Lösungsverfahren, optimiert für die Anforderungen einer HiL-Simulation. Hierfür wird zu Beginn eines Simulationsschrittes, die aktuelle Stabilitätsgrenze mit der verwendeten Schrittweite verglichen. Ist die verwendete

Schrittweite kleiner als die Schrittweite der Stabilitätsgrenze, wird der Schritt mit einem expliziten Verfahren gelöst. Anderenfalls wird ein vereinfachtes implizites Verfahren verwendet. In Abs. 3.3 wurde das Lösungsverhalten von expliziten Verfahren in grenznahen Gebieten untersucht. Es hat sich gezeigt, dass gerade bei unstetigen Funktionsverläufen, starke Oszillationen auftreten können. Um dies zu vermeiden wird ein Korrekturfaktor ω eingeführt, welcher die Stabilitätsgrenze leicht nach unten verschieben kann. Der Wechsel von explizitem zu implizitem Verfahren erfolgt daher schon etwas früher als eigentlich notwendig.

Um die Schrittweite des kombinierten Verfahrens konstant zu halten, wird für das implizite Verfahren eine deutlich kleinere Konsistenzordnung als für das explizite Verfahren verwendet. Beide Verfahren haben dann eine ähnliche Berechnungszeit. Dadurch kann ein Schritt vom kombinierten Verfahren, unabhängig vom verwendeten Löser, mit einer konstanten Schrittweite ausgeführt werden. Die verwendeten Lösungsverfahren sind das klassische RK-Verfahren vierter Ordnung (wie im rein expliziten Verfahren), sowie das implizite Eulerverfahren erster Ordnung.

Die Daten zu den Stabilitätsgrenzen wurden vor Simulationsbeginn durch die Modellanalyse im Simulationsobjekt, für jeden Arbeitspunkt abgelegt. In jedem Simulationsschritt kann dann anhand der aktuellen Werte der Zustandsvariablen und Eingangsgrößen, die aktuelle grenzstabile Schrittweite $h_{\max}$ ausgelesen werden. Solange sich die Simulation im Stabilitätsbereich des expliziten Lösungsverfahrens befindet, ist die Lösung durch die hohe Konsistenzordnung und die kleine Schrittweite sehr genau. Droht die Lösung instabil zu werden, bestimmt das implizite Verfahren den Simulationsschritt. Die Lösung des impliziten Verfahrens weist durch die geringere Konsistenzordnung einen größeren Diskretisierungsfehler auf. Die Schrittweite vom expliziten Verfahren kann dafür jedoch eingehalten werden und Abweichungen durch grenzstabile Oszillationen werden vermieden.

Das kombinierte Verfahren wird auf Modelle angewendet, welche hauptsächlich im Stabilitätsgebiet des expliziten Verfahrens gelöst werden können jedoch auch instabile Punkte im Arbeitsbereich besitzen. Der größte Teil der Simulation kann dadurch mit einer hohen Genauigkeit durch das explizite Verfahren gelöst werden und muss nicht wie bei der Wahl eines impliziten Verfahrens mit größerer Schrittweite oder deutlich kleinerer Konsistenzordnung gelöst werden.

4.4.5 Wahl des Lösungsverfahrens

Der Lösungsalgorithmus wählt den geeigneten Löser auf Grundlage der gegebenen Analyseinformationen. Hierfür stehen folgende Lösungsverfahren zur Verfügung:

- rein explizites Lösungsverfahren vierter Ordnung
- rein implizites Lösungsverfahren vierter Ordnung

- kombiniertes Lösungsverfahren
- BE-Verfahren

Das rein explizite Verfahren wird gewählt, wenn die kleinste Stabilitätsgrenze über den gesamten Arbeitsbereich, in jedem Zustand größer ist als die kleinstmögliche explizite $h_{\text{min,exp}}$ Schrittweite. Durch den Korrekturfaktor ω kann auch ein Sicherheitsabstand zur kleinsten Stabilitätsgrenze eingehalten werden, sodass für die Wahl des expliziten Verfahrens gelten muss:

$$h_{\text{min,exp}} \leq \min(h_{\text{max}}) \cdot \omega \tag{60}$$

In diesem Fall ist die Stabilität für das explizite Verfahren über den gesamten Arbeitsbereich gesichert. Ist Gl. (60) nicht erfüllt, wird das kombinierte Verfahren gewählt, um stabile Bereiche explizit und instabile Bereiche impliziten zu lösen. Eine rein implizite Lösung wird gewählt wenn der größte Teil der Simulation in einem für das explizite Verfahren instabilen Bereich liegt. Hierfür steht das rein implizite Fahren mit hoher Ordnung oder das einfache implizite Eulerverfahren zur Verfügung. Eine Simulation bei der eine kleine Schrittweite wichtig ist, kann mit einem größeren Konsistenzfehler auf das implizite Eulerverfahren zurückgreifen. Wird eine größere Genauigkeit der diskreten Stellen gewünscht, kann das rein implizite Verfahren mit einer größeren Schrittweite und höherer Ordnung verwendet werden.

5 Umsetzung

Das in Abs. 4 dargestellte Konzept beschreibt eine Optimierungsroutine bestehend aus Analyse- und Lösungsverfahren für eine parameterabhängig optimierte HiL-Simulation. Im Folgenden wird die Umsetzung des Konzepts beschrieben. Dabei wird auf die Herausforderungen der Umsetzung des Algorithmus eingegangen und die Implementierung beispielhaft in der Programmiersprache C++ vorgestellt. Als Beispiel wird die Implementierung der Optimierungsroutine in eine bestehende HiL-Simulation zum Test automobiler Steuergeräte implementiert. Zur besseren Veranschaulichung wird dessen Aufbau zunächst beschrieben.

5.1 Vorstellung der Beispielumgebung

Das hier verwendete Beispiel basiert auf einem Projekt von Kähler und Ackermann [15], welches der Entwicklung und dem Test von automobilen Steuergeräten dient. Ein hybrides Simulationsmodell soll die am Steuergerät angeschlossenen mechatronischen Lasten simulieren. Die hierfür entwickelte HiL-Umgebung besteht aus dem Steuergerät als Eingebettetes System und zwei heterogenen, parallelen Simulationen für die niederfrequenten und die hochfrequenten Anteile der Signale. Während der niederfrequente Anteil durch eine CPU-basierte Simulation bestimmt wird, bestimmt eine analoge Simulation die Näherung der hochfrequenten Anteile. Beide Ergebnisse werden zusammen an eine Leistungselektronik übergeben, welche die Stromregelung und damit die Schnittstelle zum zu testenden Steuergerät bildet. Die Messung der Ausgangsgröße des Steuergeräts bildet die Eingangsgröße des Simulationsmodells. [15]

Die Implementierung der Optimierungsroutine erfolgt für die Softwaresimulation unter Berücksichtigung der HiL-Anforderungen. Beispiele für umgesetzte Modelle sind Gleichstrommotoren, Halogenlampen und Signalhörner.

5.2 Rahmengestaltung

Da die Implementierung der Optimierungsroutine in C++ erfolgt, bietet sich eine objektorientierte Umsetzung an. Diese hat den Vorteil, dass die unterschiedlichen Modelle durch eine einheitliche Darstellungsform als Objekt zusammengefasst werden können. Dies vereinfacht die Modelldefinition und ermöglicht eine kompakte Übergabe von Modell an Lösungsalgorithmus.

Ein großer Vorteil der Systemsprache C++ ist die Schnelligkeit des kompilierten Programms. Um diese Schnelligkeit weiter ausnutzen zu können wird in zeitkritischen Funktionsabschnitten, auf zeitintensive Datentypen verzichtet. Das bedeutet, dass von der

Nutzung des dynamischen Arrays `std::vector<>` weitestgehend abgesehen wird. Stattdessen werden statische Arrays oder Array-Zeiger verwendet, auf welche deutlich schneller zugegriffen werden kann. Da nicht jede Arraygröße bei der Kompilierung bekannt ist, werden dynamische Arrays durch Array-Zeiger ersetzt. Die Nutzung von Zeigern als Datenspeicher erfordert manuelle Speicherverwaltung. Vor der Nutzung muss im Programm daher die Allokation von Speicher erfolgen. Da eine manuelle Reservierung auch eine manuelle Freigabe erfordert, wird eine dafür vorgesehene Funktion jeweils im Konstruktor und Destruktor der Klasse aufgerufen.

5.3 Umsetzung des Algorithmus

Grundlegend werden für die Implementierung der Optimierungsroutine zwei Klassendefinitionen benötigt. Abb. 16 zeigt ein vereinfachtes Klassendiagramm, zur Vorstellung beider Klassen. Die Darstellung ist stark vereinfacht, da nur Attribute und Methoden vorgestellt werden, auf welche im Folgenden eingegangen wird. Die `Solver` Klasse ist

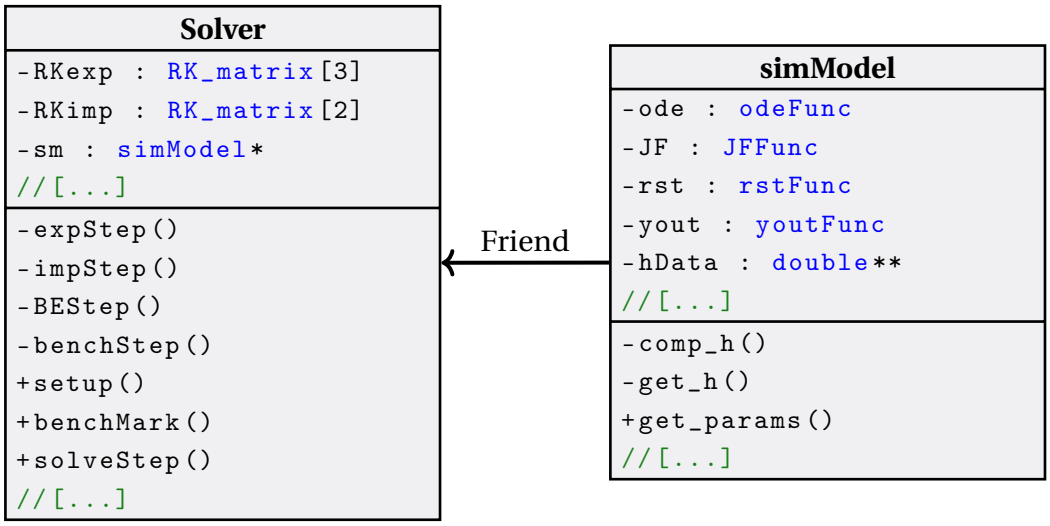

Abbildung 16: *Klassendiagramm der Optimierungsroutine. Die `simModel` Klasse deklariert die `Solver` Klasse als "Friend"*

die Umsetzung des Lösungsalgorithmus und enthält die Funktion zur Ausführung eines Lösungsschrittes als öffentliche (`public`) Funktion `solveStep()`. Die zweite Klasse ist die Modell beschreibende `simModel` Klasse. Diese deklariert `Solver` als "Friend" um ihre privaten Modelleigenschaften für den Löser zugänglich zu machen. Für eine detaillierte Beschreibung der Umsetzung werden die entwickelten Klassen im Folgenden vorgestellt und die Implementierung des Konzepts beschrieben.

5.3.1 Klassenbeschreibung des Modells

Die Klasse des Simulationsmodells ist in dieser Umsetzung die `simModel` Klasse. Während der Instanziierung eines Klassenobjekts wird durch den Konstruktor eine Modelldefinition geladen, sodass die Attribute der erzeugten Instanz, das Simulationsmodell beschreiben. Die Modelldefinition liegt, zur Erfüllung der Anforderung an die allgemeinen Darstellung, in allgemeiner Form vor. Eine Modelldefinition besteht aus sieben Modellfunktionen, deren Funktions-Zeiger durch eine `typedef` Deklaration vordefiniert ist. Die sieben Funktions-Zeiger werden in einem `Struct` kompakt zusammengefasst und können in dieser Form an den Konstruktor von `simModel` übergeben werden. Ein vollständiges Beispiel für eine Modelldefinition ist im Anhang A.2 zu finden.

Die sieben Typendefinitionen `odeFunc`, `JFFunc`, `rstFunc`, `youtFunc`, `prmSize`, `wrtFunc`, `RLFunc` beschreiben die DGL, die Jacobi-Matrizen, die Zustandsüberwachung, die Ausgabefunktion, die Funktion zur Übergabe der Arraygröße, die Funktion zur Parameterübergabe und die Übergabe des RL-Glieds. Die Funktion zur Übergabe des RL-Glieds ist in diesem Beispiel für die parallele Ausführung von Hardware und Softwaresimulation notwendig und wird daher nicht weiter behandelt. Die Funktionen werden im Folgenden genauer beschrieben.

Modellfunktion für DGL und Jacobi-Matrix

Für die Beschreibung der Implementierung der Modellfunktionen soll folgende DGL als Beispiel betrachtet werden:

$$\begin{aligned} y_1' &= \frac{u - y_1 \cdot R}{L} \\ y_2' &= 0 \end{aligned} \tag{61}$$

Die Parameter R und L werden in der Umsetzung durch den Parametervektor `p[]` dargestellt. Ebenso werden die beiden Zustandsvariablen y_1 und y_2 durch den Vektor für die Zustandsvariablen mit `y_n[0]` und `y_n[1]` beschrieben. Ein Ausschnitt der Modellfunktion zur Definition der Gl. (61) ist in Quellcode 2 durch die Funktion `ODE()` zu sehen. Als konstante Funktionsparameter erhält die Funktion den aktuellen Zeitwert `t_n`, den Zustandsvektor `y_n`, den Eingangswert `u_n`, den aktuellen Zustand `state_n`, sowie den Parametervektor `p`. Der schreibbare vektor `writeTo` ist ebenfalls ein Funktionsparameter und dient der Übergabe der Funktionsauswertung. Durch die `Switch`-Anweisung kann anhand des übergebenen Zustandswerts ausschließlich die Auswertung der DGL des aktuellen Zustands durchgeführt werden. Durch die Übergabewerte ist die Auswertung auch abhängig vom Parametervektor, wodurch die Anforderung an die Trennung von Modell und Parameter gewährleistet ist. Die Modellfunktion zur Definition der Jacobi-Matrix ist in

```
void ODE(const double t_n, const double y_n[], const double u_n,
         const int& state_n, double writeTo[], const double p[]){
    switch (state_n){
    case 0:
        writeTo[0] = (u_n - y_n[0] * p[0]) / p[1];
        writeTo[1] = 0;
        break;
    //[...] Weitere Case-Anweisungen
    default:
        // [...] Default Anweisung z.B. wie case 0
        break;
    }
}
```

Quellcode 2: *Programmausschnitt der Modellfunktion zur Definition der DGL*

gleicher Weise aufgebaut. Einziger Unterschied ist der hier zweidimensionale Array-Zeiger `writeTo`, welcher die Jacobi-Matrix des aktuellen Zustandes speichert.

Zustandsüberwachung und Ausgabefunktion

Die Umsetzung der Zustandsüberwachung ist Beispielhaft in Quellcode 3 dargestellt. Die Modellfunktion erhält ebenfalls die aktuellen Zustandswerte und Parameter, besitzt

```
void rst(const double& t_n, double y_n[], const double u_n,
         int& state_n, double p[]) {
    bool son;
    // [...] Setzen des Logikwerts son
    switch (state_n) {
    case 0:
    {
        if (!son) state_n = 1;
        else y_n[1] = 0;
        break;
    }
    // [...] Weitere Case-Anweisungen
    default:
        // [...] Default Anweisung z.B. wie case 0
        break;
    }
}
```

Quellcode 3: *Programmausschnitt der Modellfunktion zur Definition der Zustandsüberwachung*

jedoch die Möglichkeit den Zustandswert `state_n`, den Parametervektor `P`, sowie den Zustandsvektor `y_n` zu überschreiben. In der Zustandsüberwachung werden dann für den aktuellen Zustand, die Bedingungen für einen Zustandswechsel überprüft und der Zustandswert gegebenenfalls angepasst. Zur Realisierung von unstetigen Funktionsverläufen kann der Zustandsvektor `y_n`, wie in Quellcode 3 zu sehen, neu gesetzt werden. Das Überschreiben von Parametern ist dann sinnvoll, wenn durch den Parametervektor zusätzliche

Systembedingungen wie z. B. das Blockieren eines Motors gespeichert werden. Die Ausgabefunktion dient der Simulationsvereinfachung und kann berechnete Zustandsvariablen in Ausgangsvariablen umwandeln. So erhält die Funktion den aktuellen Zustandswert, sowie den Parametervektor und kann damit neue Ausgangswerte bestimmen.

Parameterübergabe

Die Parameterübergabe erfolgt durch zwei Funktionen zur Übergabe der Arraygröße und der Parameterwerte. Diese Übergabe wurde aufgeteilt, da der Konstruktor vor dem Einlesen der Parameter Informationen zur Größe des Modells benötigt. Durch die Verwendung von Array-Zeigern muss zuerst eine Allokation für den benötigten Speicher durchgeführt werden. Erst nachdem die Modelldimension bekannt ist und der Speicher reserviert wurde, wird die zweite Funktion zur Parameterübergabe aufgerufen. In dieser wird der Parametervektor, die Initialwerte für y_n und der Arbeitsbereich der Systemvariablen übergeben. Im Konstruktor wird anschließend die Funktion `get_params()` aufgerufen, welche den Parametervektor mit Werten aus dem Simulationsinterface überschreiben kann.

Die Funktions-Zeiger für die DGL, die Jacobi-Matrix, die Zustandsüberwachung und die Ausgabefunktion werden vom Konstruktor von `simModel` als Objektattribut gespeichert. Die anderen Modellfunktionen werden im Konstruktor lediglich zur Speicherreservierung und zur Übergabe von Modellinformationen aufgerufen und anschließend verworfen. Zusätzlich führt der Konstruktor nach dem Laden der Parameter die parameterabhängige Modellanalyse durch. Da die Lebensdauer eines Objekts für eine Simulation begrenzt ist, wird bei jedem Simulationsstart ein neues Objekt erzeugt und die Modellanalyse kann für eventuell geänderte Parameter neu durchgeführt werden. Die Umsetzung der Modellanalyse wird im Folgenden erläutert.

Modellanalyse

Die Modellanalyse wird über den in der Modelldefinition angegebenen Arbeitsbereich mit zugehöriger Auflösung durchgeführt. Wie im Konzept in Abs. 4.3 beschrieben, wird für jeden Zustand des Modells und für jeden Punkt im Arbeitsbereich, die Stabilitätsgrenze $h_{\max}$ bestimmt. Für `simModel` geschieht dies durch den Funktionsaufruf von `comp_h()` im Konstruktor. Die Bestimmung der Stabilitätsgrenzen wird in `comp_h()` nacheinander für jeden Systemzustand durchgeführt. Durch die vorliegenden Informationen über den Arbeitsbereich der Zustandsvariablen und Eingangsgrößen, sowie über die gewählte Auflösung (`res` für *resolution*), wird die Jacobi-Matrix für jeden Arbeitspunkt im Arbeitsbereich ausgewertet. Für k Zustandsvariablen und m Eingangsgrößen ergibt sich die Anzahl der Auswertungen pro Zustand aus:

$$\texttt{res}^{k+m} \tag{62}$$

Jede Auswertung der Jacobi-Matrix ergibt eine konstante $k \times k$ Matrix, für welche die Eigenwertpaare zu berechnen sind. Die Eigenwertbestimmung kann in C++ z. B. durch

die Template Library *Eigen*[11] übernommen werden. Da für die Bestimmung der Stabilitätsgrenze lediglich der betragsmäßig größte Eigenwert von Interesse ist, gilt es in jedem Zustand für `res`$^{k+m}$ Eigenwerte die Stabilitätsgrenze $h_{\max}$ zu bestimmen. In dem hier betrachteten Beispiel treten lediglich reelle Eigenwerte auf und die Berechnung der Stabilitätsgrenzen erfolgt zur Reduzierung der Laufzeit durch die Division mit dem in Abs. 4.3 beschriebenen Skalierungsfaktor z_{real}. In diesem Beispiel wird der Vektor mit den Stabilitätsgrenzen für jeden Zustand in die Matrix `hData` abgelegt. Für die Bestimmung der Stabilitätsgrenzen von komplexen Eigenwerten muss in C++ auf eine symbolische Template Library zurückgegriffen werden. Mit dieser kann Gl. (47) für jeden Eigenwert gelöst werden. Im letzten Schritt der Modellanalyse müssen die Stabilitätsgrenzen umsortiert werden, um den Gültigkeitsbereich zu erweitern. Da die bestimmten Stabilitätsgrenzen lediglich an den eingesetzten Arbeitspunkte gelten, jedoch auch auf den Arbeitsbereich zwischen den diskreten Punkten angewendet werden sollen, wird jede Stabilitätsgrenze durch den nächstkleineren Wert ersetzt. Soll die Stabilitätsgrenze für Systemvariablen zwischen zwei Arbeitspunkten geprüft werden ist sichergestellt, dass die nachgeschlagene Stabilitätsgrenze die nächstkleinere ist. Zum Nachschlagen bestimmt die Funktion `get_h()`, aus der bekannten Schreibreihenfolge der Werte, den Index der Matrix `hData`. Im Attribut `h_minAll` wird die kleinste Stabilitätsgrenze des gesamten Arbeitsbereichs gespeichert.

5.3.2 Klassenbeschreibung des Lösungsalgorithmus

Die Klasse `Solver` beschreibt in diesem Beispiel die Implementierung des Lösungsalgorithmus. Der Lösungsalgorithmus ist auf den Umgang mit Objekten der `simModel` Klasse zugeschnitten. Beim Erstellen einer `Solver`-Instanz wird daher der Zeiger zur `simModel`-Instanz `sm` übergeben. Durch die Friend-Deklaration der Klassen, kann die `Solver`-Instanz auf die Attribute des Simulationsmodells zugreifen und parameterabhängig ein optimales Lösungsverfahren wählen. Vor der Wahl des Lösungsverfahrens muss jedoch die Laufzeitanalyse für das übergebene Modell durchgeführt werden.

Implementierung der Laufzeitanalyse

Die Laufzeitanalyse wird durch den Funktionsaufruf `benchMark()` gestartet. Diese Funktion führt eine Laufzeitabschätzung für die Durchführung eines Simulationsschrittes aus. Da ein Simulationsschritt durch einen Funktionsaufruf von `solveStep()` ausgeführt wird, ist eine Laufzeitabschätzung für diese Funktion zu finden. Hierfür greift `benchMark()` auf die Funktion `benchStep()` zurück, welche den Aufbau der Lösungsalgorithmus-Funktion `solveStep()` nachahmt, aber speziell auf die Aufgaben der Laufzeitabschätzung zugeschnitten ist. So ist z. B. die Anzahl der Iterationsschleifen in der `benchStep()` Funktion fest gesetzt. `benchMark()` kann eine vorher festgelegte Anzahl an Testzyklen mit dem Funktionsaufruf von `benchStep()` in jedem möglichen Zustand und mit jedem Löser

ausführen. Aus den Zeitmessungen aller Zustände wird der Worst Case für jeden Löser in den Attributen der `simModel`-Instanz gespeichert. Zusätzlich wird separat die Laufzeit der Iterationsschleifen vom rein impliziten und vom BE-Verfahren bestimmt.

Nach der Laufzeitanalyse muss die `setup()` Funktion aufgerufen werden, welche aus den gegeben Informationen der Analysen ein Lösungsverfahren mit optimaler parameterabhängiger Schrittweite wählt.

Wahl des Lösungsverfahrens

Mit dem Aufruf der `setup()` Funktion werden Einstellungsmöglichkeiten übergeben, welche die Auswahl des Lösers beeinflussen, das Limit für die Iterationsschleifen festlegen und den Korrekturfaktor ω der Stabilitätsgrenze festlegen. Durch die Eingabe kann eine rein implizite oder eine rein explizite Ausführung erzwungen werden. Im anderen Fall wird geprüft ob die kleinstmögliche Schrittweite für das explizite Verfahren kleiner ist, als die kleinste Stabilitätsgrenze `h_minAll`. Für das explizite Verfahren berechnet sich die kleinstmögliche Schrittweite $h_{\text{exp,min}}$ aus der Worst Case Laufzeit `EXP_WC` addiert zur hardwarespezifischen Zeit für die Ausführung der Restaufgaben `simBuffer`:

$$h_{\text{exp,min}} = \texttt{EXP\_WC} + \texttt{simBuffer} \tag{63}$$

Die Zeit der Restaufgaben ist ein statisches, konstantes Klassenattribut der `Solver` Klasse. Bei der Wahl eines impliziten Verfahrens berechnet sich die kleinstmögliche Schrittweite $h_{\text{imp,min}}$ unter zusätzlicher Betrachtung des Iterationslimits `looplim` und der Worst Case Laufzeit eines Iterationsschrittes (`Ilp_WC` für das rein implizite und analog `Blp_WC` für implizite Euler-Verfahren):

$$h_{\text{imp,min}} = \texttt{IMP\_WC} + \texttt{simBuffer} + \texttt{looplim} \cdot \texttt{Ilp\_WC} \tag{64}$$

Die Wahl des Lösungsverfahrens wird als Objektattribut gespeichert, sodass bei der Ausführung des Lösungsschrittes `solveStep()` das richtige Verfahren verwendet wird.

Lösung des Modells

Nach der Berechnung der Schrittweite bestimmt jeder Aufruf der Funktion `solveStep()` (s. Abb. 17) die Näherung des Folgeschrittes mit dem gewählten Lösungsverfahren. Als Argument erhält die Funktion den Eingabevektor u_n, welcher in diesem Beispiel lediglich aus der skalaren Eingangspannung besteht. Das explizite und das implizite Runge-Kutta Verfahren ist jeweils in einer Löser-Funktion implementiert. Die implementierung ist allgemein vorgenommen, sodass die jeweilige Funktion Gl. (13) und (14) umsetzt. Für die Ausführung der speziellen Verfahren wird auf die `Struct`-Definition von `RK_matrix` zurückgegriffen. In dieser sind die Gewichtungsfaktoren a, b und c sowie die Stufenzahl N eines RK-Verfahrens definiert. Durch zwei `RK_matrix` `Struct`-Vektoren werden alle

verwendeten RK-Verfahren in jeweils einem Vektor für die impliziten und die expliziten Verfahren gespeichert. Durch die Indexierung lässt sich das gewählte Verfahren im `Struct`-Vektor gezielt auswählen. Das klassische RK-Verfahren befindet sich z. B. in dem expliziten `Struct`-Vektor und kann durch den Index `slv` = 2 verwendet werden. Eine dritte Löser-Funktion beschreibt das implizite Eulerverfahren als direkte Implementierung. Unabhängig vom gewählten Lösungsverfahren beginnt jeder Simulationsschritt mit der Bestimmung der Stabilitätsgrenze. Hierfür wird die `simModel` Methode `get_h()` aufgerufen. Der Rückgabewert wird mit dem Korrekturfaktor multipliziert, welcher als konstantes Klassenattribut von `Solver` vorliegt. Anschließend folgt eine `if`-Abfrage, welche je nach gewähltem Lösungsverfahren und aktueller Stabilitätsgrenze die jeweilige Löser-Funktion ausführt. Nach der Bestimmung der Näherung wird die Funktion zur Zustandsüberwachung `sm->rst()` aufgerufen.

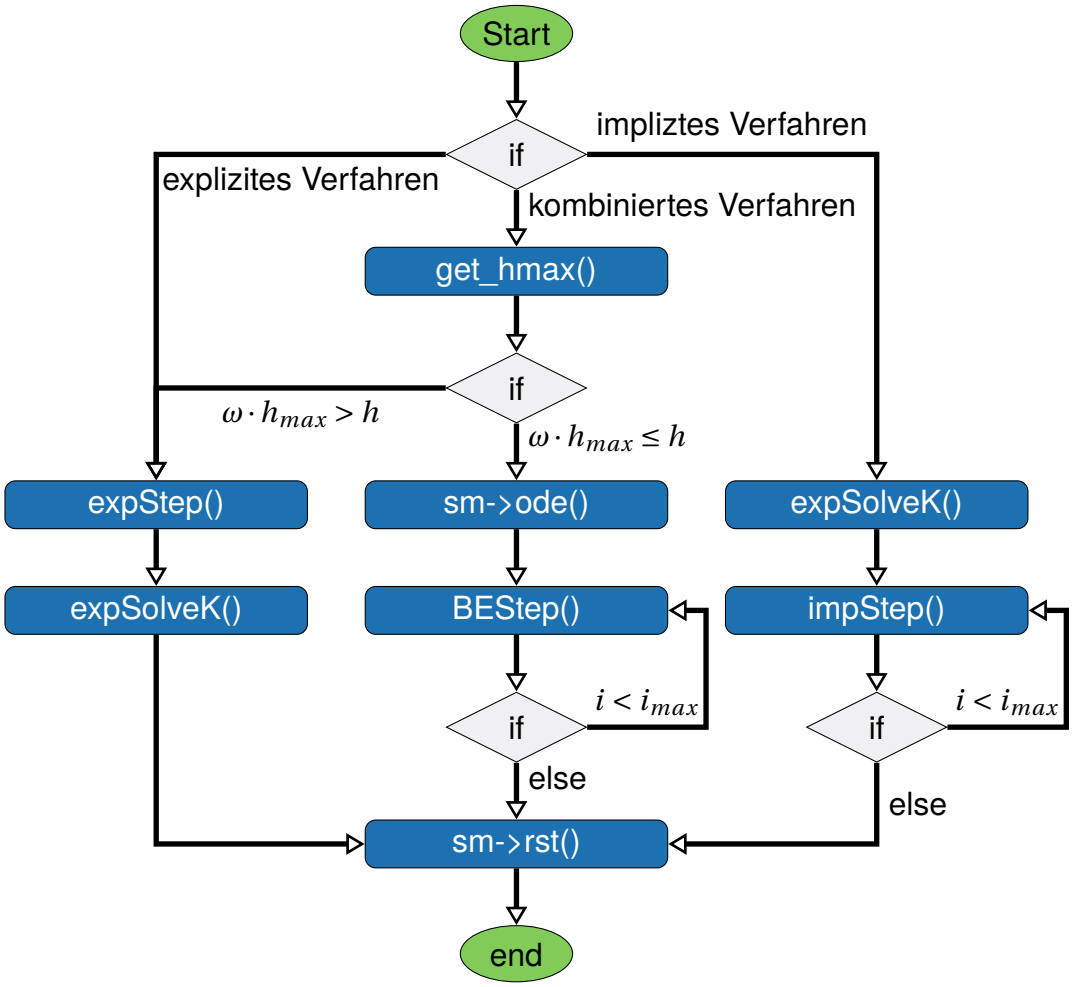

Abbildung 17: *Programmablaufplan der solveStep()-Funktion. $i < i_{max}$ steht für die Abbruchbedingung der Iterationsschleife bei Erreichen des Iterationslimits.*

Explizites Lösungsverfahren Das rein explizite Verfahren ist in zwei Schritte aufgeteilt, welche in Quellcode 4 dargestellt sind. Zunächst wird die Funktion `expStep()` mit

```
void Solver::expStep(double u_n, int slv) {
    expSolveK(u_n, slv);
    for (int i = 0; i < sm->yvrs; i++) {
        for (int j = 0; j < RKexp[slv].s; j++) {
            y_n[i] += k[j][i] * h_n * RKexp[slv].b[j];
        }
    }
};
void Solver::expSolveK(double u_n, int slv) {
    for (int i = 0; i < RKexp[slv].s; i++) {
        emptyVec(sumVec, sm->yvrs);
        for (int j = 0; j < sm->yvrs; j++) {
            for (int l = 0; l < i; l++) {
                sumVec[j] += RKexp[slv].a[i][l] * k[l][j];
            }
            sumVec[j] *= h_n;
            sumVec[j] += y_n[j];
        }
        sm->ode(...); // Auswertung der DGL
    }
};
```

Quellcode 4: *Allgemeine Implementierung des expliziten Runge-Kutta Verfahrens*

den Argumenten des aktuellen Eingabewertes `u_n` und dem Index `slv` des genutzten RK-Verfahrens im `RK_matrix Struct`-Vektor ausgeführt. Diese bestimmt durch den Aufruf der Funktion `expSolveK()` die Werte für die N (`RKexp[slv].N`) Stufen. Hierfür wertet `expSolveK()` die in `ODE()` umgesetzten DGL aus. In diesem Zusammenhang wird das Objektattribut `sumVec` für die Summation verwendet. Wie in Abs. 5.2 erläutert wurde, werden in zeitkritischen Funktionen Array-Zeiger als dynamische Speicher verwendet. Da ein lokaler Array eine Allokation und Deallokation für jeden zu berechnenden k-Wert verlangt, wird das Objektattribut `sumVec` wiederverwendet und zu Beginn durch `emptyVec()` auf Null gesetzt. Nach der Berechnung der Stufen k werden die neuen Zustandsvariablen y_n durch die Umsetzung von Gl. (13) bestimmt.

Implizites Lösungsverfahren Im rein impliziten Lösungsalgorithmus (s. Quellcode 5) ist das implizite Runge-Kutta Verfahren mit einer Newton-Raphson Iteration implementiert. Die Umsetzung orientierte sich dabei an dem von Granados [9] vorgestellten Lösungsweg. Für die Bestimmung der Startwerte des impliziten Lösungsalgorithmus werden die Stufen k durch die bereits vorgestellte Funktion `expSolveK()` bestimmt. Anschließend folgt eine `do-while`-Schleife für die Ausführung der impliziten Iteration. Die Abbruchbedingung ist durch das Erreichen der Genauigkeitsanforderung `abs_tol` oder des Iterationslimits `loopLim` gegeben. Im Anschluss an die Iterationsschleife werden aus den dort bestimmten Werten von k, die Zustandsvariablen y_{n+1} des nächsten Schrittes wie in Gl. (13) berechnet.

```
expSolveK(u_n, RKe);
do {
    impStep(u_n, RKi);
    loopCount += 1;
} while (errmax*0.9*h_n > abs_tol && loopCount <= loopLimit);
// [...] Berechnung von y_n+1 = y_n + h*b*k;
```

Quellcode 5: *Lösungsalgorithmus für einen Simulationsschritt mit dem rein imliziten Verfahren*

Im Folgenden wird die Umsetzung der Funktion `impStep()` auf dem mathematischen Weg erläutert. Die programmiertechnische Umsetzung kann dem dokumentierten Anhang A.1 entnommen werden. Wird die Gl. (14) auf eine implizite, N-stufige Runge-Kutta Matrix $\boldsymbol{a}$ angewendet, ergibt sich ein lineares Gleichungssystem mit den unbekannten Stufen k. Zur Lösung des Gleichungssystems empfiehlt Granados[9] das Netwot-Raphson Verfahren, welches gegenüber der Fixpunktiteration ein besseres Konvergenzverhalten aufweist[9]. Für die Netwot-Raphson Iteration, lässt sich Gl. (14) umformen zur Gl. (65).

$$g_i(k) = f(t_n + c_i \cdot h,\, y_n + h \cdot \sum_{j=1}^{N} a_{ij} \cdot k_j) - k_i = 0 \quad \text{für} \quad i = [1,...,N] \tag{65}$$

Da das Verfahren auf ein DGL System angewendet werden soll, wird Gl. (65) in Vektorschreibweise (**fett** gedruckt) dargestellt. Hierfür werden zunächst die Dimensionen der beteiligten Vektoren vorgestellt. Bei einem DGL System von m Zustandsvariablen $\boldsymbol{y} = [y_1,\ldots,y_m]^\top$ ergeben sich neben Gl. (15) folgende Dimensionen:

$$\boldsymbol{k} = \begin{bmatrix} \boldsymbol{k}_1 \\ \boldsymbol{k}_2 \\ \vdots \\ \boldsymbol{k}_N \end{bmatrix} \in \mathbb{R}^{N\cdot m} \quad \text{mit} \quad \boldsymbol{k}_i \in \mathbb{R}^m \quad \text{für} \quad i = [1,...,N]$$

$$\boldsymbol{g}(\boldsymbol{k}) \in \mathbb{R}^{N\cdot m}, \quad \boldsymbol{f}(t,\boldsymbol{y}) \in \mathbb{R}^m \quad \text{und} \quad \boldsymbol{y} \in \mathbb{R}^m \tag{66}$$

Nun ergibt sich die Vektorschreibweise von Gl. (65) für ein DGL System durch:

$$\boldsymbol{g}(\boldsymbol{k}) = \begin{bmatrix} \boldsymbol{f}(t_n + \boldsymbol{c}\cdot h,\, \boldsymbol{y}_n + h \cdot \sum_{j=1}^{N} a_{1,j} \cdot \boldsymbol{k}_j) - \boldsymbol{k}_1 \\ \boldsymbol{f}(t_n + \boldsymbol{c}\cdot h,\, \boldsymbol{y}_n + h \cdot \sum_{j=1}^{N} a_{2,j} \cdot \boldsymbol{k}_j) - \boldsymbol{k}_2 \\ \vdots \\ \boldsymbol{f}(t_n + \boldsymbol{c}\cdot h,\, \boldsymbol{y}_n + h \cdot \sum_{j=1}^{N} a_{N,j} \cdot \boldsymbol{k}_j) - \boldsymbol{k}_N \end{bmatrix} = 0 \tag{67}$$

Zur Findung der Lösungen von k mit den Newton-Raphson Verfahren wird die Jacobi-Matrix $\boldsymbol{J_g}$ von $\boldsymbol{g}(\boldsymbol{k})$ benötigt. Mit der Jacobi-Matrix $\boldsymbol{J_f}(t,\boldsymbol{y})$ der Vektorfunktion $\boldsymbol{f}(t,\boldsymbol{y})$, ist

$\boldsymbol{J_g}$ gegeben durch [9]:

$$\boldsymbol{J_g} = \begin{bmatrix} ha_{1,1}[\boldsymbol{J_f}(\boldsymbol{y}_n + h\sum_{j=1}^{N} a_{1,j}\boldsymbol{k}_j)] - \boldsymbol{I} & \dots & ha_{1,N}[\boldsymbol{J_f}(\boldsymbol{y}_n + h\sum_{j=1}^{N} a_{1,j}\boldsymbol{k}_j)] \\ \vdots & \ddots & \vdots \\ ha_{N,1}[\boldsymbol{J_f}(\boldsymbol{y}_n + h\sum_{j=1}^{N} a_{N,j}\boldsymbol{k}_j)] & \dots & ha_{N,N}[\boldsymbol{J_f}(\boldsymbol{y}_n + h\sum_{j=1}^{N} a_{N,j}\boldsymbol{k}_j)] - \boldsymbol{I} \end{bmatrix} \tag{68}$$

Mit $\boldsymbol{J_g}(\boldsymbol{k})$ aus Gl. (68) und $\boldsymbol{g}(\boldsymbol{k})$ aus Gl. (67) lässt sich dann folgendes Gleichungssystem aufstellen:

$$\boldsymbol{J_g}(\boldsymbol{k}_m) \cdot \Delta\boldsymbol{k}_m = -\boldsymbol{g}(\boldsymbol{k}_m) \tag{69}$$

Wird das Gleichungssystem für $\Delta\boldsymbol{k}_m$ gelöst, ergibt sich der Vektor $\boldsymbol{k}_{m+1}$ des nächsten Iterationsschrittes aus:

$$\boldsymbol{k}_{m+1} = \boldsymbol{k}_m + \gamma\Delta\boldsymbol{k}_m \tag{70}$$

γ entspricht hier einem Stellwert, mit welchem Einfluss auf die Konvergenzgeschwindigkeit genommen werden kann. Die initialen $\boldsymbol{k}_m$ Werte des ersten Iterationsschrittes stammen aus dem expliziten Vorschritt. Die Funktion `impStep()` ist so implementiert, dass der neu berechnete Wert $\boldsymbol{k}_{m+1}$ in der Variable des alten Werts $\boldsymbol{k}_m$ gespeichert wird. So kann durch jedes Aufrufen der Funktion ein weiterer Iterationsschritt durchgeführt werden. Die Fehlerabschätzung für die Abbruchbedingung der Iterationsschleife erfolgt durch die Betrachtung der Änderung von $\boldsymbol{k}$. Der maximale Fehler `errmax` ergibt sich aus:

$$\texttt{errmax} = \max(\gamma \cdot h \cdot \boldsymbol{c} \cdot \left\|\Delta\boldsymbol{k}_m\right\|) \tag{71}$$

Kombiniertes Lösungsverfahren Das kombinierte Lösungsverfahren wird durch den Vergleich der Stabilitätsgrenze mit der gewählten Schrittweite gesteuert. Liegt die gewählte Schrittweite unterhalb der Stabilitätsgrenze, wird das bereits beschriebene explizite Verfahren ausgeführt. Im anderen Fall wird auf eine vereinfachte Form des impliziten Verfahrens zurückgegriffen. Der implizite Schritt des kombinierten Verfahrens wird durch das BE-Verfahren bestimmt. Für eine effiziente Implementierung wird auf die allgemeine Darstellungsform, wie für das rein implizite Verfahren, verzichtet. Die Umsetzung erfolgt daher durch eine direkte Implementierung des Verfahrens. So wird der initiale k Wert für das Iterationsverfahren auch nicht durch die allgemein definierte Funktion `expStepK()` bestimmt, sondern besteht, wie in Quellcode 6 zu sehen, aus nur einer Funktionsauswertung der DGL. Die Funktion `BEStep()` führt ähnlich wie `impStep()` das vorgestellte Newton-Raphson Iterationsverfahren aus.

```
        sm->ode(t_n, y_n, u_n, state_n, k[0], sm->param);
        do {
            BEStep(u_n);
            loopCount += 1;
        } while (errmax*0.9*h_n > 1e-3 && loopCount <= loopLimit);
```

Quellcode 6: *Lösungsalgorithmus des kombinierten Verfahrens*

5.4 Simulationsablauf

Nachdem der Aufbau der Klassen für Löser und Modell beschrieben sind, soll im Folgenden auf den Simulationsablauf eingegangen werden.

5.4.1 Modellerstellung

Für die Modellerstellung müssen zunächst die in Abs. 5.3.1 beschriebenen sieben Modellfunktionen definiert werden. Zusätzlich wird für jede Modelldefinition eine Zugriffsfunktion benötigt, welche den `modelFcts`-Struct zur Verfügung stellt. Die Funktion `getModel()` in Quellcode 7 zeigt dies in allgemeiner Beschreibung.

```
int iniState = 0;
modelFcts getModel() {
    modelFcts modelDef = {  ODE, JF, rst, yout, ParamSize,//
                            WriteParams, RL, iniState };
    return modelDef;
};
```

Quellcode 7: *Modellerstellung durch die getModel() Funktion*

5.4.2 Simulation

Zur Beschreibung des Ablaufs einer Simulation zeigt Abb. 18 den Programmablauf der Main-Funktion. Zunächst wird der globale Zeiger `p_simMod` als `simModel` Instanz außerhalb der Main-Funktion erstellt. Innerhalb der Mainfunktion kann der Zeiger dann für das gewählte Modell instanziiert werden. Dafür wird abhängig vom Main-Funktionsargument die `getModel()` Funktion des jeweiligen Modells an den Konstruktor übergeben. Der Programmablauf in Abb. 18 zeigt dies für die verallgemeinerte Darstellung von ModellA. Nachdem das Simulationsmodell erstellt ist, wird eine `Solver`-Instanz mit dem Simulationsmodell als Konstruktor-Argument erzeugt. Durch den Aufruf der `benchMark()` Funktion kann eine Laufzeitanalyse gestartet werden. Die `setup()` Funktion setzt die richtigen Einstellungen und bestimmt die Schrittweite sowie das geeignete Lösungsverfahren für die Simulation. Vor dem Simulationsstart wird dann eine Initialisierungsfunktion ausgeführt, welche die Schrittweite an die parallele Hardwaresimulation übergibt und weitere hardwarespezifische Einstellungen übernimmt. Die Simulation wird durch das Ausführen der `while`-Schleife gestartet. Beim Aufrufen der Objektmethode `solveStep()` wird als Argument in diesem Beispiel der aktuelle Messwert der Spannung von der Hardware übergeben. Beim erreichen einer Abbruchbedingung wird die Simulation beendet und eine Exit-Routine für die Hardwarekomponenten ausgeführt.

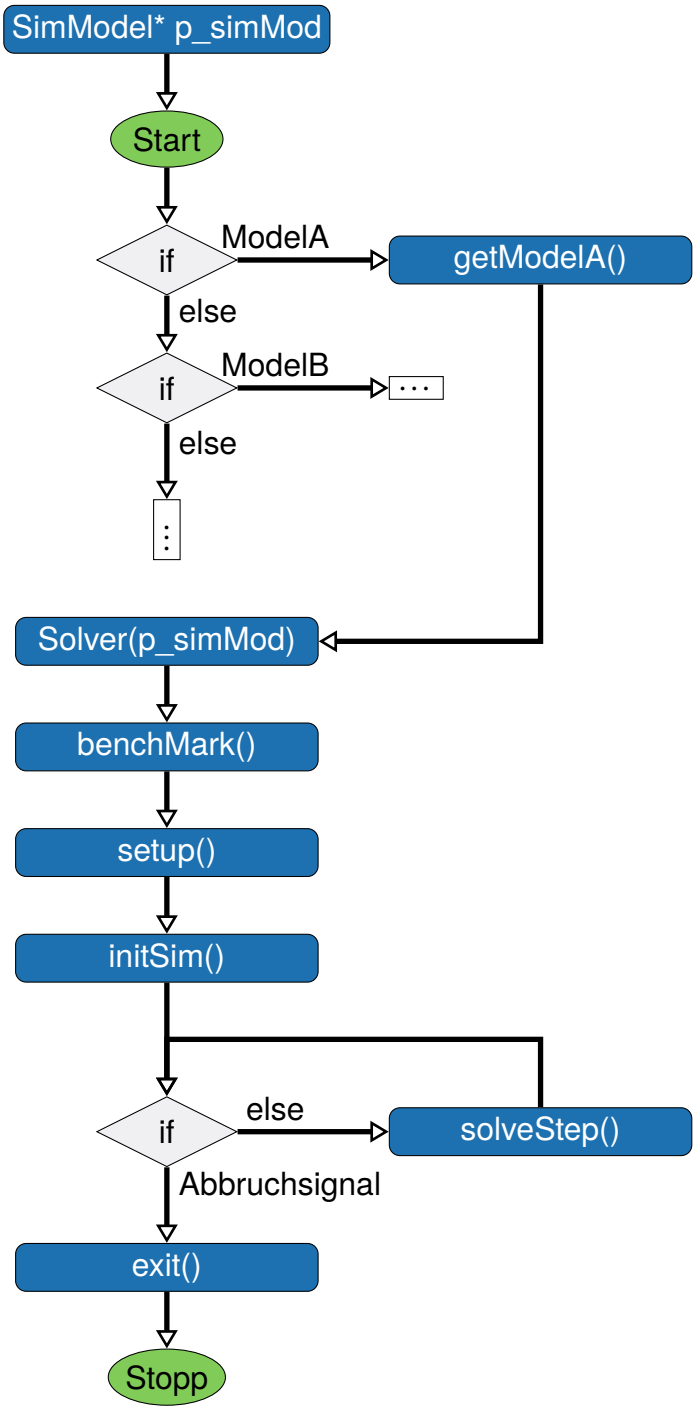

Abbildung 18: *Programmablaufplan der Main-Funktion*

6 Ergebnis

Im Rahmen dieser Arbeit wurde ein Lösungsalgorithmus mit automatischer, parameterabhängiger Schrittweitenoptimierung für eine HiL-Simulation entwickelt und für die in Abs. 5.1 vorgestellte Umgebung umgesetzt. In Kapitel 3 wurden die Anforderungen an die Optimierungsroutine hergeleitet. Darauf aufbauend folgte die Betrachtung verschiedener Lösungsansätze und der Entwurf eines Konzepts. Die Umsetzung dieses Entwurfs und die Implementierung in die bestehende HiL-Simulation wurden in Kapitel 5 beschrieben. In diesem Kapitel wird das Ergebnis der Umsetzung betrachtet. Es wird untersucht inwieweit die in Abs. 4 definierten Analyse- und Lösungsverfahren umgesetzt wurden und wie gut die Optimierungsroutine die Anforderung erfüllt. Zur Beurteilung der Leistungsfähigkeit wird die Genauigkeit der Lösung, die Komplexität des Algorithmus, sowie die Berechnungszeit der Lösungsverfahren als primäres Bewertungsmerkmal verwendet. Auf Grundlage der Ergebnisbewertung wird der Anwendungsbereich der entwickelten Optimierungsroutine eingegrenzt.

6.1 Bewertung der Analyseverfahren

Ziel der Analyseverfahren ist die parameterabhängige Optimierung des Lösungsalgorithmus. Durch die Betrachtung der Modelle werden Eigenschaften der DGL gesammelt und zur Berechnung der optimalen Schrittweite sowie zur Wahl des geeigneten Lösungsverfahrens hinzugezogen. Grundlage ist die Betrachtung der Modellsteifigkeit und der Lösungsstabilität für alle Zustände in verschiedenen Arbeitspunkten. Die Modellinformationen werden gespeichert, sodass der Lösungsalgorithmus während der Simulation auf diese zugreifen kann. Die Eigenwerte des Systems bestimmt eine Modellanalyse. Die variablen Eigenwerte von nichtlinearen Systemen werden für diskrete Arbeitspunkte in einem definierten Arbeitsbereich berechnet. Zur Nutzung der dadurch aufgezeigten Steifigkeitseigenschaften werden mithilfe der Stabilitätsfunktion des expliziten Verfahrens die Stabilitätsgrenzen der Schrittweiten berechnet. Diese beschreiben die größtmögliche Schrittweite für eine stabile Näherung mit dem expliziten Verfahren an jedem Arbeitspunkt. Die Kenntnis über die Stabilitätsgrenzen ermöglicht eine Optimierung durch die Wahl von Schrittweite und Lösungsverfahren. Die Grundanforderung an das Analyseverfahren ist damit erfüllt. Die Darstellungsform des Modells ermöglicht eine Anwendung auf allgemein definierte Modelle und erfüllt damit auch diese Anforderung. Durch die Durchführung der Analyse nach jeder Parametereingabe ist die Modellanalyse außerdem parameterabhängig gestaltet. Eine Laufzeitanalyse durch Laufzeittests bestimmt die Berechnungszeiten eines Simulationsschrittes für die optimale Wahl der Schrittweite.

Die Analyseverfahren stellen dem Lösungsalgorithmus hilfreiche Informationen zur Verfügung. Durch das Wissen über die größtmögliche Schrittweite ist es dem Algorithmus

möglich eine hohe Genauigkeit für das Verfahren zu erziehen. Die Stabilitätsgrenzen ermöglichen die parameterabhängige Wahl eines geeigneten Lösungsverfahrens und damit eine effiziente Näherung für jedes Simulationsmodell. Die Umsetzung der Analyseverfahren ist, durch eine optimierte Gestaltung für reelle Eigenwerte, schnell in der Berechnung der Stabilitätsgrenzen von kleinen Systemen. Auch der Zugriff auf die gespeicherten Daten gestaltet sich durch die Implementierung mit schnellen Array-Zeigern zeitsparend. Durch die Eigenwertbestimmung für jeden möglichen Arbeitspunkt, ist die Speicherkomplexität jedoch verbesserungswürdig. Für eine Auflösung des Arbeitsbereichs n ist die Speicherkomplexität O_{Speicher} eines Systems mit v Zustandsvariablen gegeben durch:

$$\mathcal{O}_{\text{Speicher}}(n^v) \tag{72}$$

Da der Speicherverbrauch mit der Potenz der Zustandsvariablen wächst, ist die Analyse von größeren Systemen schnell ineffizient. Natürlich spielt ab einer gewissen Größe auch der Zeitverbrauch für die Analyse eine wichtige Rolle.

6.2 Leistungsfähigkeit des Lösungsalgorithmus

Um die Leistungsfähigkeit des hier entwickelten Lösungsalgorithmus zu bewerten, wird die Komplexität und die Berechnungszeit im Zusammenhang zur Näherungsgenauigkeit betrachtet. Für eine Aussauge über die Effizienz wird die Komplexität der Verfahren allgemein verglichen und die unterschiedlichen Laufzeiten für ein Beispiel betrachtet. Um die Genauigkeit zu bewerten wird allgemein die Konsistenzordnung der Verfahren verwendet. Zusätzlich erfolgt ein beispielhafter Vergleich der Vor- und Nachteile der Lösungsverfahren.

6.2.1 Komplexität des Algorithmus

Die Betrachtung der Komplexität ermöglicht im Zusammenhang mit der implementierten Konsistenzordnung eine allgemeine Aussage über die Effizienz der jeweiligen Verfahren. Hierfür wird im Folgenden die Komplexität der in Kapitel 5 umgesetzten Lösungsverfahren bestimmt und verglichen.

Das rein explizite Lösungsverfahren ist allgemein für RK-Verfahren mit N Stufen implementiert. Unter der Annahme, dass die Funktionsauswertung der DGL der zeitaufwändigste Prozess ist und die zusätzlichen Berechnungen vernachlässigt werden können, bestimmt sich die Komplexität durch die Berechnung der N Stufen. Für die Berechnung jeder Stufe wird eine Funktionsauswertung für v Systemvariablen durchgeführt. Die Komplexität für das explizite Verfahren ergibt sich daher zu:

$$\mathcal{O}_{\text{exp}}(v \cdot N) \tag{73}$$

Der Algorithmus des rein impliziten Verfahrens besteht aus mehreren deutlich aufwändigeren Prozessen. Die Prozesse mit der größten Komplexität sind die Bestimmung der Jacobi Matrix J_g für die Newton-Raphson Iteration, die Bestimmung von $g(k)$, sowie die Lösung des linearen Gleichungssystems zur Bestimmung der Stufen k. Die Jacobi Matrix J_g wertet $N \times N$ mal die $v \times v$ große Jacobi-Matrix J_F des Modells aus. Die Zeitkomplexität vom Prozess zur Bestimmung von J_g ist daher durch $\mathcal{O}_{\mathrm{Jg}}(v^2 \cdot N^2)$ gegeben. Die Lösung des linearen Gleichungssystems greift auf einen Algorithmus zurück, welcher die gleiche Zeitkomplexität besitzt. Die Bestimmung von $g(k)$ basiert auf N Funktionsauswertung der v DGL, sodass die Zeitkomplexität mit $\mathcal{O}_{\mathrm{g(k)}}(v \cdot N)$ gegeben ist. Die Zeitkomplexität des gesamten impliziten Algorithmus wird daher von der Komplexität der Jacobi-Matrix für das Newton-Raphson Verfahren bestimmt. Mit der Anzahl der Iterationen l gilt für einen Schritt mit dem impliziten Verfahren:

$$\mathcal{O}_{\mathrm{imp}}(l \cdot v^2 \cdot N^2) \tag{74}$$

Das kombinierte Verfahren greift auf das rein explizite Verfahren oder das BE-Verfahren zurück. Da das BE-Verfahren eine vereinfachte Implementierung des impliziten Verfahrens für $N = 1$ ist, gilt für die Zeitkomplexität des impliziten Anteils:

$$\mathcal{O}_{\mathrm{komb,imp}}(l \cdot v^2) \tag{75}$$

Es wird deutlich, dass die Zeitkomplexität des rein expliziten Verfahrens linear von der Stufenzahl und der Systemgröße abhängt. Das implizite Verfahren ist von beiden Faktoren quadratisch und linear von der Anzahl der Iterationen abhängig. Der implizite Anteil des kombinierten Verfahrens besitzt eine quadratische Abhängigkeit von der Systemgröße und eine lineare von der Anzahl der Iteration. Die Effizienz des expliziten Verfahrens ist dadurch deutlich größer als die des impliziten. Für eine Vergrößerung der Konsistenzordnung durch die Vergrößerung der Stufenzahl N, wächst die Zeitkomplexität für das explizite Verfahren deutlich langsamer als für das implizite. Dem expliziten Verfahren ist es so möglich, größere Genauigkeiten mit kleinerem Zeitaufwand zu realisieren. Das gleiche gilt auch für die Systemvergrößerung durch v. Auch hier können große Systeme durch das explizite Verfahren schneller gelöst werden als durch das implizite. Der implizite Anteil des kombinierten Verfahrens besitzt im Gegensatz zum rein impliziten eine feste Ordnung durch die konstante Stufenzahl $N = 1$. So ist die Komplexität zwar geringer, das Verfahren jedoch auch weniger genau.

6.2.2 Vergleich der Lösungsverfahren

Der vorherige Abschnitt hat die Effizienz der Algorithmen allgemein eingeordnet. Im Folgenden soll die Effizienz und Genauigkeit der Lösungsverfahren anhand konkreter Beispiele verglichen werden. Zur Bewertung der Effizienz werden zunächst die Ergebnisse

einer Laufzeitanalyse eingeordnet. Anschließend folgt die Betrachtung der Ergebnisse einer Simulation mit verschiedenen Lösungsverfahren.

Für die in Abs. 5.1 vorgestellte HiL-Umgebung wird eine Laufzeitanalyse für das Modell einer Halogenlampe mit temperaturabhängigem Widerstand durchgeführt. Die Ausführende Hardware ist ein Raspberry Pi 3 Modell B+ mit 64-Bit ARMv8 Architektur, 1,4Ghz Taktung und 1GB Arbeitsspeicher. Die Laufzeitanalyse bestimmt die Worst Case Berechnungszeiten für einen Schritt des jeweiligen Lösungsverfahrens. Die Ergebnisse sind in Tab. 5 dargestellt. Die ermittelten Berechnungszeiten spiegeln die Erwartungen aus der Be-

	Berechnete Laufzeit
Expliziter Schritt	$3{,}9\mu s$
BE Schritt	$3{,}9\mu s$
BE Itertation	$1{,}1\mu s$
Impliziter Schritt	$11{,}3\mu s$
Implizite Iteration	$1{,}2\mu s$

Tabelle 5: *Ergebnisse der Laufzeitanalyse für alle Algorithmen, angewendet auf das Modell einer Halogenlampe.*

trachtung der Algorithmuskomplexität wider. Die kleinste Schrittweite kann durch das explizite Verfahren oder einen einzelnen Berechnungsschritt des BE-Verfahrens ermöglicht werden. Während das BE-Verfahren das Verfahren mit der kleinsten Konsistenzordnung $p = 1$ ist, ermöglicht das explizite Verfahren eine Näherung mit der Konsistenzordnung $p = 4$. Um die Genauigkeit des BE-Verfahrens zu erhöhen, können zusätzliche Iterationsschritte hinzugefügt werden, welche jedoch jeweils eine Verlängerung der Schrittweite um $1{,}1\mu s$ bedeutet. Das rein implizite Verfahren ist mit einer Berechnungszeit von $11{,}3\mu s$ deutlich zeitintensiver, bietet jedoch eine implizite Berechnung mit der Konsistenzordnung $p = 6$. Eine weitere Verbesserung der Genauigkeit ist auch hier durch weitere Iterationsschritte möglich, welche eine verhältnismäßig kleine Vergrößerung der Schrittweite von $1{,}2\mu s$ nach sich ziehen.

Nach der Betrachtung der Berechnungszeiten aus der Laufzeitanalyse werden nun die Ergebnisse eines Simulationsbeispiels betrachtet. Als Beispiel wird der Stromverlauf im Ein- und Ausschaltvorgang einer Halogenlampe simuliert. An die Lampe wird für $1ms$ ein Spannungsimpuls angelegt. Für das kombinierte, sowie für das rein explizite Verfahren wurde eine konstante Schrittweite von $10\mu s$ gewählt. Die Schrittweite wurde auf Grundlage der vorher durchgeführten Laufzeitanalyse berechnet, sodass das kombinierte Verfahren eine implizite oder explizite Berechnung mit konstanter Schrittweite durchführen kann.

Diese ergibt sich aus der Berechnungszeit eines Schrittes vom BE-Verfahren, der Zeit für eine zusätzliche Iteration und der Zeit für die Hardware abhängigen Restaufgaben:

$$h = 10\mu s = 3{,}9\mu s + 1{,}1\mu s + 5\mu s \tag{76}$$

Abbildung 19 zeigt die Näherungen durch das explizite und das kombinierte Verfahren im oberen Diagramm, sowie den zeitlichen Verlauf der Stabilitätsgrenzen im unteren. Die

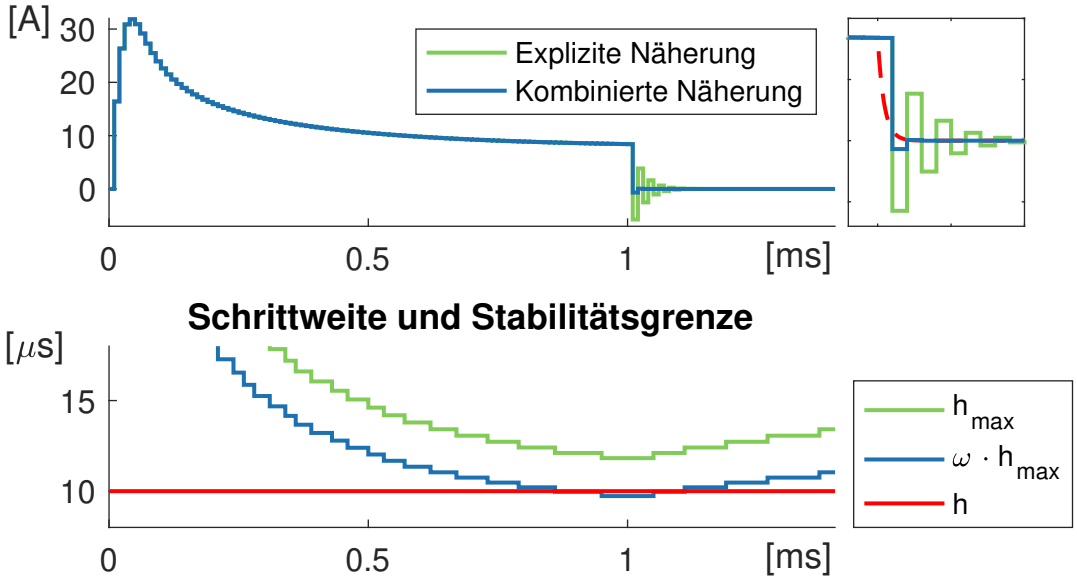

Abbildung 19: *Vergleich der rein expliziten Näherung (grün) mit der Lösung des kombinierten Verfahrens (blau) im oberen Diagramm. Eine Vergrößerung der Unstetigkeit, zusammen mit dem tatsächlichen Funktionsverlauf (rot), ist oben rechts gezeigt. Das untere Diagramm zeigt die Veränderung der Stabilitätsgrenze über die Simulationszeit. Die verwendete Schrittweite ist rot eingezeichnet. Das kombinierte Verfahren bestimmt die Näherung für den Teil $\omega \cdot h_{max} \leq h$, implizit.*

rot eingezeichnete Linie bei $10\mu s$ zeigt die Schrittweite beider Verfahren im Vergleich zur Stabilitätsgrenze. Die Stabilitätsgrenze ist für das das kombinierte Verfahren dieselbe wie für das rein explizite Verfahren, da beide auf derselben expliziten Näherung beruhen. Für das kombinierte Verfahren wurde die Stabilitätsgrenze jedoch durch einen Korrekturfaktor von ca. $\omega \approx 0{,}85$ leicht verkleinert, sodass der Wechsel von expliziter zu impliziter Näherung früher einsetzt. Das untere Diagramm in Abb. 19 zeigt daher zwei Verläufe für die eigentlich gleiche Stabilitätsgrenze. Der Verlauf für das kombinierte Verfahren ist leicht nach unten versetzt. Betrachtet man die Näherung des Stromverlaufs im oberen Diagramm zeigt sich das identische Ergebnis von kombiniertem und explizitem Verfahren für den größten Teil der Simulation. Lediglich in dem Bereich, in welchem die Stabilitätsgrenze die rot eingezeichnete Schrittweite unterschreitet, wird die Näherung durch das implizite BE-Verfahren bestimmt. Die in diesem Bereich initiierte sprunghafte Abschaltung führt zu einer starken Oszillation in der expliziten Näherung. Dies wird durch die implizite Berechnung im kombinierten Verfahren vermieden. Der beschriebene Bereich

ist in Abb. 19, am oberen rechten Rand, vergrößert dargestellt. Hier zeigen sich die Abweichungen von der rot gestrichelten exakten Lösung für beide Verfahren. Die Näherung durch das kombinierte Verfahren ist sichtbar genauer als die Oszillation des expliziten.

Da die implizite Näherung durch das BE-Verfahren im vorherigen Beispiel ein besseres Ergebnis erzielt hat, stellt sich die Frage ob dieses Verfahren vielleicht auch für die gesamte Simulation eine genauere Näherung bietet. Hierfür wird in einem weiteren Vergleich die Näherung desselben Beispiels zwischen dem kombinierten Verfahren und dem BE-Verfahren betrachtet. Abb. 20 zeigt einen vorher explizit gelösten Ausschnitt im linken

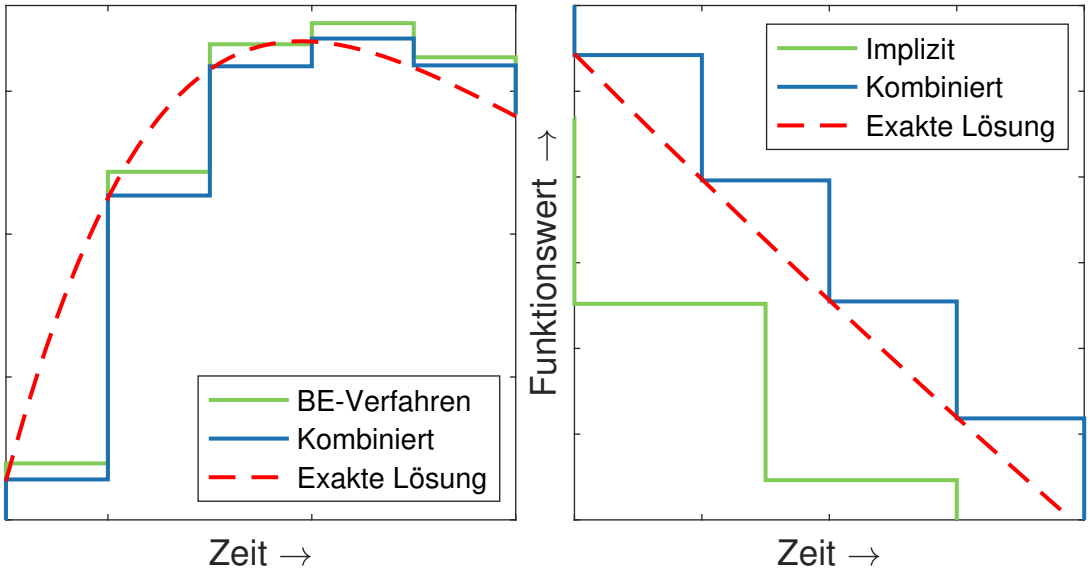

Abbildung 20: *Vergleich von Näherung des BE-Verfahrens mit der kombinierten Näherung links. Die Näherung des BE-Verfahrens weist durch die geringe Ordnung einen höheren Konsistenzfehler auf. Das rechte Diagramm zeigt den Verleich einer impliziten Näherung höherer Ordnung zum kombinierten Verfahren. Das implizite Verfahren besitzt durch die größere Schrittweite einen größeren Diskretisierungsfehler.*

Diagramm. Es wird deutlich, dass die explizite Berechnung durch das kombinierte Verfahren eine genauere Näherung erzielt, als das BE-Verfahren. Dies ist leicht durch die unterschiedlichen Konsistenzordnungen zu begründen. Die Verwendung des BE-Verfahrens für die gesamte Simulation würde daher zu einer schlechteren Näherung führen. Um die Genauigkeit des BE-Verfahrens zu erhöhen, kann das rein implizite Verfahren mit einer höheren Konsistenzordnung verwendet werden. Wie die vorherige Laufzeitanalyse gezeigt hat, muss für dieses Verfahren jedoch eine deutlich größere Schrittweite gewählt werden. Abbildung 20 zeigt im rechten Diagramm, dass die größere Schrittweite nicht nur zu einer größeren Abweichung der Momentanwertabtastung, sonder auch zu einem größeren Diskretisierungsfehler führt.

6.3 Anwendungsbereich

Der Vorteil des kombinierten Verfahrens wird in den vorgestellten Beispielen deutlich. Würde die gesamte Simulation ausschließlich explizit oder implizit gelöst werden, würde die Genauigkeit an unterschiedlichen Punkten sinken. Ein implizites Verfahren führt durch seinen höheren Berechnungsaufwand zu einem größeren Diskretisierungsfehler und ein explizites Verfahren durch das kleinere Stabilitätsgebiet zu Schwingungen in grenzstabilen Bereichen. Die Verwendung des kombinierten Verfahrens ist daher immer dann sinnvoll, wenn eine Simulation in der Nähe der Stabilitätsgrenze durchgeführt wird und es durch unstetige Lösungsverläufe zu Oszillationen in der Lösung kommen kann. Hier muss jedoch individuell beurteilt werden, ob Genauigkeit an den Unstetigkeiten wichtiger ist, als der Diskretisierungsfehler der gesamten Simulation. Für stetige Lösungsverläufe in grenzstabilen Regionen kann daher weiterhin eine rein explizite Näherung zu einem Ergebnis mit größerer Zufriedenstellung führen. Die Verwendung von rein impliziten Verfahren bietet sich hauptsächlich für sehr steife Modelle an, welche im größten Teil der Simulation, in für das explizite Verfahren instabilen Regionen, gelöst werden müssen. Die häufigen und schnellen Änderungen der Stabilitätsgrenze in steifen Modellen, kann auch zu Oszillationen des teilweise impliziten kombinierten Verfahren führen, sodass eine rein implizite Lösung bessere Ergebnisse erzielt.

Zusätzlich wird der Anwendungsbereich des kombinierten Verfahrens durch die notwendige Modellanalyse eingegrenzt. Durch die Laufzeit- und Speicherkomplexität des Analyseprozesses, ist die Analyse von großen nichtlinearen Modellen mit vielen Systemvariablen meist zu aufwändig. Auch Systeme mit sehr großem Arbeitsbereich, mit vielen schnellen Änderungen der Stabilitätsgrenzen benötigen eine zu große Auflösung der Linearisierungspunkte. Für sehr komplexe Modelle können die Systemeigenschaften nicht immer ausreichend durch die Linearisierungspunkte beschrieben werden [13]. Außerdem muss sich das zu simulierende Modell durch die allgemeine Darstellungsform nachbilden lassen. Für das implizite Verfahren bedeutet das auch, dass alle Jacobi-Matrizen der DGL Systeme bekannt sind. Es ist möglich, dass sich die Jacobi-Matrizen von sehr Aufwändigen DGL Systemen nicht in der vorgestellten Form darstellen lassen.

7 Fazit

Das Ziel dieser Arbeit ist die Entwicklung einer parameterabhängigen Schrittweitensteuerung für eine HiL-Simulation. Zum Erreichen dieses Ziels wurde eine Optimierungsroutine entworfen, welche auf Grundlage von Analyseverfahren, ein optimiertes Lösungsverfahren und eine passende Schrittweite wählt. Hierfür greift der entwickelte Lösungsalgorithmus auf bereits bekannte, sowie ein speziell entwickeltes Lösungsverfahren zurück. Die Entwicklung basiert auf der Betrachtung der Grundlagen, welche die Herausforderungen bei der Entwicklung eines Lösungsalgorithmus für eine HiL-Simulation veranschaulicht. Unter der Betrachtung der Herausforderungen wurden die Anforderungen an die Optimierungsroutine aufgestellt. Durch eine Untersuchung von verschiedenen Lösungsansätzen wurde ein allgemeines Konzept entworfen, welches anschließend in einer Beispielumgebung umgesetzt wurde.

Die größte Herausforderungen bei der Entwicklung eines Lösungsverfahrens für eine HiL-Simulation ist die Echtzeitanforderung. Das führt dazu, dass die numerische Näherung von jedem Simulationsschritt nicht mehr Zeit in Anspruch nehmen darf als die Schrittweite der Berechnung. Insbesondere im Zusammenhang mit steifen Modellen können durch große Schrittweiten Ungenauigkeiten in der Näherung entstehen. Diese entstehen durch einen größeren Diskretisierungsfehler oder durch die sinkende Kontinuität der Momentanwertabtastung des Hardwareausgangs. In normalen Softwaresimulationen gibt es verschiedene Möglichkeiten diesem Problem entgegenzutreten. Die Übertragung dieser Optimierungen auf eine HiL-Umgebung ist jedoch nicht ohne weiteres möglich. So kann für steife Systeme eine Schrittweitensteuerung verwendet werden, welche während der Simulation die Schrittweite zur Vergrößerung der Genauigkeit dynamisch anpasst. In einer HiL-Simulation ist die kleinstmögliche Schrittweite jedoch durch die Echtzeitanforderung begrenzt. Für eine hohe Genauigkeit und Kontinuität sollte daher ohnehin die kleinstmögliche Schrittweite gewählt werden, sofern der Rundungsfehler vernachlässigbar klein bleibt. Eine weitere Möglichkeit im Umgang mit steifen Systemen ist die Verwendung von impliziten Lösungsverfahren. Diese lösen durch ihre A-Stabilität auch steife Systeme mit großer Schrittweite, welche für explizite Verfahren nur durch die Wahl einer sehr kleinen Schrittweite lösbar wären. Der Nachteil der impliziten Verfahren ist durch die Komplexität des Lösungsalgorithmus gegeben. Während die Berechnungszeit eines expliziten Verfahrens linear mit der Systemgröße und Konsistenzordnung ansteigt, besitzen implizite Verfahren ein quadratisches Wachstum. Die Berechnung einer Lösung derselben Genauigkeit benötigt für ein implizites Verfahren daher deutlich mehr Zeit als für ein explizites. Die größere Berechnungszeit führt durch die Anforderungen der HiL-simulation zwangsweise zu einer größeren Schrittweite, sodass Diskretisierungsfehler und Abweichungen der Momentanwertabtastung zunehmen.

Um die Kosten für eine stabile implizite Lösung zu verringern, wurde ein Lösungsverfahren entwickelt, dass auf einer Kombination von implizitem und explizitem Verfahren

beruht. Hierfür wird im Rahmen der entwickelten Optimierungsroutine eine Modellanalyse durchgeführt, welche die Stabilitätsgrenzen des expliziten Lösungsverfahrens bezogen auf das Simulationsmodell bestimmt. Da die Stabilitätsgrenze von nichtlinearen Systemen, von dem Systemzustand abhängt, erfolgt die Modellanalyse über einen diskretisierten Arbeitsbereich. Nach einer Eigenwertbestimmung für jeden Arbeitspunkt, wird mithilfe der Stabilitätsfunktion die obere Grenze der Schrittweite für eine stabile Näherung bestimmt. Das kombinierte Lösungsverfahren nutzt die Kenntnis über die Stabilitätsgrenzen aus, um in jedem Simulationsschritt das Verfahren mit der größten Genauigkeit zu wählen. Hierfür wird in jedem Simulationsschritt, die aktuelle obere Grenze mit der Schrittweite der Simulation verglichen. Befindet sich die verwendete Schrittweite deutlich unterhalb der Stabilitätsgrenze, wird die Näherung mit einem expliziten Verfahrensschritt bestimmt. Ist die verwendete Schrittweite oberhalb oder in einem definierten Sicherheitsabstand von der oberen Grenze, wird der Simulationsschritt implizit gelöst. Damit die Schrittweite trotz Verfahrenswechsel konstant bleiben kann, wird ein implizites Verfahren mit kleinstmöglicher Konsistenzordnung verwendet. Dieses besitzt die gleiche Berechnungszeit wie das gewählte explizite Verfahren mit hoher Konsistenzordnung. Die Schrittweite wird auf Grundlage von Laufzeitanalysen für alle Lösungsverfahren berechnet. Für die Umsetzung in einer Optimierungsroutine wurde eine allgemeine, parameterabhängige Darstellungsform, geeignet für verschiedene Modelltypen entworfen. Vor der Simulation werden die Analysedaten durch die Modell- und Laufzeitanalyse bestimmt. Auf dieser Grundlage wählt ein Lösungsalgorithmus zwischen vier verschiedenen Lösungsverfahren. Ein rein explizites Verfahren wird verwendet, wenn die kleinste Stabilitätsgrenze einen ausreichenden Abstand zur verwendeten Schrittweite besitzt. Für den Fall von grenzstabilen oder instabilen Bereichen in Teilen der Simulation wird das kombinierte Verfahren verwendet. Zwei implizite Verfahren bieten die Möglichkeit, sehr steife Modelle vollständig implizit zu Lösen. Hier kann zwischen zwei verschiedenen Konsistenzordnungen mit unterschiedlicher Genauigkeit und Schrittweite gewählt werden.

Die entworfene Optimierungsroutine wurde an verschiedenen Beispielen getestet und erweist sich für ausgewählte Systeme als sinnvoll. Eine der größten Einschränkungen ist durch die große Speicherkomplexität der Modellanalyse gegeben. Die Anzahl der Arbeitspunkte im Arbeitsbereich nimmt mit der Anzahl der Variablen exponentiell zu. Dadurch ist eine Analyse für große Systeme nur bedingt möglich. Zusätzlich zeigen sich Verbesserungen in der Genauigkeit durch das kombinierte Verfahren hauptsächlich bei unstetigen oder schnellen Lösungsverläufen. Die Verwendung des impliziten Verfahrens in ruhigen, stetigen Lösungsverläufen kann zu einem größeren Diskretisierungsfehler und damit zu einem gegenteiligen Effekt führen. Für kleinere Systeme, welche schnelle Änderungen in der Nähe von instabilen Regionen aufweisen, stellt die Optimierungsroutine jedoch eine deutliche Verbesserung dar. Die explizite Näherung im stabilen Bereich weist eine hohe Genauigkeit auf, während die impliziten Schritte Oszillationen durch dynamische Verläufe verhindern. Außerdem ist die Optimierungsroutine für die meisten physikalischen Modelle geeignet und ist durch ihre Parameterabhängigkeit universal anwendbar.

8 Ausblick

Die Entwickelte Optimierungsroutine stellt eine gute Grundlage für die Optimierung von Lösungsverfahren für HiL-Simulationen dar. Diese Kapitel soll die Möglichkeiten zur Weiterentwicklung aufzeigen und Anstöße für Verbesserungen geben. Aus dem Vergleich der Berechnungszeiten in Tab. 5 wird deutlich, dass die Berechnungszeit für das implizite Verfahren deutlich größer ist, als für das explizite. Dies liegt zum einen an der Laufzeitkomplexität vom impliziten Runge-Kutta Verfahren. Zum anderen ist die Implementierung des Verfahrens verbesserungswürdig. Allgemein zeigt dies, dass ein effizienteres Lösungsverfahren mit ähnlichen Stabilitätseigenschaften gefunden werden muss. Eine effizientere Implementierung würde zu einer kleineren Schrittweite und damit zu einer höheren Genauigkeit führen. Durch die weitere Untersuchung von Alternativen zu impliziten RK-Verfahren könnten Mehrschrittverfahren eine Rolle spielen. Durch die Nutzung von bereits bestimmten Funktionswerten, sind weniger Funktionsauswertungen erforderlich. Gerade bei zeitintensiven Funktionsauswertungen könnte dies zu kürzeren Berechnungszeiten führen.

Ein weiterer Nachteil der entwickelten Optimierungsroutine ist die hohe Speicherkomplexität der Analyse. Statt einem äquidistanten Gitter von Arbeitspunkten könnte durch eine Anpassung der Auflösung eine dynamische Wahl der Arbeitspunkte gestaltet werden. Automatisch könnten kontinuierliche Bereiche mit großer Auflösung und Bereiche mit schnellen Änderungen mit kleiner Auflösung abgebildet werden. Außerdem wäre eine zustandsabhängige Definition des Arbeitsbereichs sinnvoll.

Für eine schnellere Simulation ist es immer möglich, auf eine Leistungsfähigere CPU zurück zu greifen. Doch auch für die bestehende Hardware lässt sich die Simulationsperformance erhöhen. So kann z. B. das Prinzip von *parallel computing* verwendet werden um Rechenopertationen auf verschiedene Prozessorkerne aufzuteilen. Durch die parallele Berechnung mit unterschiedlichen Lösungsverfahren, könnte mithilfe von Fehlerabschätzungen immer die genauste Näherung verwendet werden.

Eine weitere Verbesserung der Genauigkeit könnte durch eine zusätzliche Betrachtung der Zustandwechsel implementiert werden. Da die größten Ungenauigkeiten im Funktionsverlauf bei unstetigen Lösungsverläufen auftreten, wäre eine Vorhersage von Zustandswechseln sinnvoll. Dadurch könnte das Lösungsverfahren bereits vor dem Zustandswechsel an den neuen Zustand angepasst werden. Die Vorraussage könnte ebenfalls durch eine parallele Berechnung erfolgen, um die Performance der eigentlichen Simulation nicht zu verringern.

Ein großes Problem bei der Vergrößerung der Schrittweite ist die Abweichung der Momentanwertabtastung des Ausgangssingals vom theoretischen Funktionsverlauf. Zur Reduzierung der Abweichung könnte eine Alternative zur Abtast-halte-Schaltung verwendet werden. Durch die Übergabe der aktuellen Funktionssteigung, könnte das Ausgangssignal analog interpoliert werden und sich so auch zwischen den Schrittweiten anpassen.

Literatur

[1] BURGERS, Johannes M.: Mathematical examples illustrating relations occurring in the theory of turbulent fluid motion. In: *Selected Papers of JM Burgers*. Springer, 1995 (zitiert auf Seite 4)

[2] BUTCHER, J.C.: *The numerical analysis of ordinary differential equations: Runge-Kutta and general linear methods*. J. Wiley, 1987 (Wiley-Interscience publication) (zitiert auf Seiten vi und 8)

[3] CASH, J. R. ; KARP, Alan H.: A Variable Order Runge-Kutta Method for Initial Value Problems with Rapidly Varying Right-hand Sides. In: *ACM Trans. Math. Softw.* 16 (1990), September, Nr. 3, S. 201–222 (zitiert auf Seite 24)

[4] CELLIER, F.E. ; KOFMAN, E.: *Continuous System Simulation*. Springer US, 2006 (zitiert auf Seiten 7, 11, 13, 15 und 16)

[5] COMMITTEE, IEEE Computer Society. Microprocessor S. ; ELECTRICAL, Institute of ; ENGINEERS, Electronics ; BOARD, IEEE-SA S.: *IEEE Standard for Floating-Point Arithmetic*. Institute of Electrical and Electronics Engineers. (IEEE std) (zitiert auf Seite 42)

[6] DOLAN, Elizabeth D. ; MORÉ, Jorge J.: Benchmarking optimization software with performance profiles. In: *Mathematical programming* 91 (2002), Nr. 2, S. 201–213 (zitiert auf Seite 40)

[7] ENGELN-MÜLLGES, G. ; NIEDERDRENK, K. ; WODICKA, R.: *Numerik-Algorithmen: Verfahren, Beispiele, Anwendungen*. Springer Berlin Heidelberg, 2010 (Xpert.press) (zitiert auf Seite 14)

[8] FRANK, Jason: Stability of Runge-Kutta Methods. In: *Numerical Modelling of Dynamical Systems*. Utrecht University, 2008, Kap. 10, S. 53–64 (zitiert auf Seiten 12 und 37)

[9] GRANADOS, Andrés: *Implicit Runge-Kutta Algorithm Using Newton-Raphson Method*. 03 1998 (zitiert auf Seiten 55, 56 und 57)

[10] GREENSPAN, D.: *Numerical Solution of Ordinary Differential Equations: For Classical, Relativistic and Nano Systems*. Wiley, 2008 (Physics textbook) (zitiert auf Seiten 10 und 11)

[11] GUENNEBAUD, Gaël ; JACOB, Benoît u. a.: *Eigen v3*. http://eigen.tuxfamily.org. 2010 (zitiert auf Seite 52)

[12] HAIRER, E. ; NØRSETT, S.P. ; WANNER, G.: *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems*. Springer, 1993 (Lecture Notes in Economic and Mathematical Systems) (zitiert auf Seite 12)

[13] HIGHAM, Desmond J. ; TREFETHEN, Lloyd N.: Stiffness of ODEs. In: *BIT Numerical Mathematics* 33 (1993), Jun, Nr. 2, S. 285–303 (zitiert auf Seiten 6, 36 und 66)

[14] KARPFINGER, C.: *Höhere Mathematik in Rezepten: Begriffe, Sätze und zahlreiche Beispiele in kurzen Lerneinheiten.* Springer Berlin Heidelberg, 2015 (zitiert auf Seiten 7, 9, 10, 15 und 43)

[15] KÄHLER, D. ; PROF. DR. ACKERMANN, G.: Abschlussbericht. In: *Entwicklung generischer Modelle elektrischer Verbraucher mit einer neuartigen selbständigen Parametrierung für das Echtzeitprüfstandstool für automobile Steuergeräte (Automobile Toolsuite)* (zitiert auf Seite 47)

[16] LANGTANGEN, Hans P.: *Solving nonlinear ODE and PDE problems.* Department of Informatics, University of Oslo, 2016 (zitiert auf Seiten 5 und 7)

[17] LAPIDUS, L. ; SEINFELD, J.H.: *Numerical Solution of Ordinary Differential Equations.* Elsevier Science, 1971 (Mathematics in Science and Engineering) (zitiert auf Seite 8)

[18] SOETAERT, K. ; CASH, J. ; MAZZIA, F.: *Solving Differential Equations in R.* Springer Berlin Heidelberg, 2012 (Use R!) (zitiert auf Seiten 8, 9, 11, 12 und 13)

[19] SOLYMOSI, A. ; GRUDE, U.: *Grundkurs Algorithmen und Datenstrukturen in JAVA: Eine Einführung in die praktische Informatik.* Springer Fachmedien Wiesbaden, 2017 (SpringerLink : Bücher) (zitiert auf Seite 16)

[20] VANDE WOUWER, A. ; SAUCEZ, P. ; VILAS, C.: *Simulation of ODE/PDE Models with MATLAB®, OCTAVE and SCILAB: Scientific and Engineering Applications.* Springer International Publishing, 2014 (SpringerLink : Bücher) (zitiert auf Seiten 4 und 10)

A Anhang

A.1 Funktion impStep() zur Berechnung eines impliziten Schrittes

```
void Solver::impStep(double u_n, int slv) {
// Funktion zur Berechnung eines impliziten Steps
// Der Algorithm is basiert auf:
// Implicit Runge-Kutta Algorithm Using Newton-Raphson Method by Andres L. Granados M.
// researchgate.net/publication/275952224_Implicit_Runge-Kutta_Algorithm_Using_Newton-Raphson_Method
// Berechnung der k Summe aus den initalen ks des exp. Verfahrens: ksum = y_n + h * sumOf(k*a)
  for (int i = 0; i < RKimp[slv].N; i++) {
    emptyVec(ksum[i], sm->yvrs);    // Leeren von ksum
    for (int j = 0; j < RKimp[slv].N; j++) {
      vps_add(ksum[i], k[j], RKimp[slv].a[i][j], sm->yvrs);  // ksum = sumof(k*a)
    }
    vps(ksum[i], h_n, sm->yvrs);    // ksum = h * sumof(k*a)
    addvec(ksum[i], y_n, sm->yvrs);   // ksum = y_n + h * sumof(k*a)
  }
  // Berechnung der Jacobi-Matrix Jg aus Gleichung 42 in Granados Paper
  for (int i = 0; i < RKimp[slv].N; i++) {
    for (int j = 0; j < RKimp[slv].N; j++) {
      // Berechnung von Jf(t_n+h*c, y_n + h * sumof(k*a)) = Jf(tch_imp, ksum)
      // Speichern in der Matrix JG
      // Jedes Element von JG ist eine Jacobi Matrix der groesse [sm->yvrs,sm->yvrs]
           // Umsortieren in JG_temp fuer das loesen mit der LGS Funktion
      sm->JF(tch_imp[j], ksum[i], u_n, state_n, JG[i][j], sm->param);
      for (int m = 0; m < sm->yvrs; m++) {
        for (int n = 0; n < sm->yvrs; n++) {
          JG_temp[i * sm->yvrs + m][j * sm->yvrs + n] = JG[i][j][m][n] * ah_imp[i][j];
        }
      }
    }
  }
  // Subtrahieren der Einheitsmatrix von JG_temp um zu Gleichung 42 in Granados Paper zu gelangen
  subE(JG_temp, sm->yvrs * RKimp[slv].N);
  // Berechnung von g(k) (Gl. 41 in Granados Paper)
  for (int i = 0; i < RKimp[slv].N; i++) {
    sm->ode(tch_imp[i], ksum[i], u_n, state_n, gmat[i], sm->param); // gmat = f(ksum)
    subvec(gmat[i], k[i], sm->yvrs); // gmat = f(ksum)-k
  }
  // Umsortieren der Matrix gmat in den vektor-array gvec
  gmattogvec(RKimp[slv].N); // g wird hier ebfalls mit -1 multipliziert, zur Vorbereitung auf Gl. 43
       in Granados Paper
  // Loesen des LGS von JG*deltaK=-g (Gl. 43 in Granados Paper)
  solveLGS(gvec, JG_temp, RKimp[slv].N * sm->yvrs); // gvec = delta K
  // Transformationen in Matrix der groesse [RKorder x sm->yvrs]
  gvectogmat(RKimp[slv].N);
  // Berechnung von km+1 aus km und omega*deltaK wie in Gl. 43 in Granados Paper
  for (int i = 0; i < RKimp[slv].N; i++) {
    vps(gmat[i], omega, sm->yvrs);  // gmat = deltaK * omega
    addvec(k[i], gmat[i], sm->yvrs);  // gmat = km+1 = km + deltaK * omega
    // Berechnung des Fehlers wie Gl. 15 in Granados Paper
    myNorm(err[i], gmat[i], sm->yvrs); // err = normOf(deltaK)
  }
  vpv(err, RKimp[slv].c, RKimp[slv].N);  // err = normOf(deltaK) * c
  errmax = *std::max_element(std::begin(err), std::begin(err) + RKimp[slv].N); // errmax =
       max(normOf(deltaK))
};
```

A.2 Modelldefinition einer Halogenlampe

```
#include "simLamp.h"
#include <vector>
#include <cmath>
#include "simModel.h"

// Model Functions
void halrst(const double& t_n, double y_n[], const double u_n, int& state_n, double p[]) {
  switch (state_n) {
           // case 0: Zustandsueberwachung fuer Zustand 0
           // case 1: Zustandsueberwachung fuer Zustand 1
  default:
           // Nur ein Zustand => leere Funktion zur Zustandsueberwachung
    break;
  }
}
void halJF(const double& t_n, const double y_n[], const double u_n, const int& state_n, double
     **writeTo, const double p[]) {
  switch (state_n) {
  default:
       // Definition der Jacobi-Matrix
    writeTo[0][0] = -1.0*(p[0] * pow((y_n[1] / p[2]), p[5])) / p[1];
    writeTo[0][1] = -1.0*(p[0] * p[5] * y_n[0] * pow((y_n[1] / p[2]), (p[5] - 1))) / (p[1] * p[2]);
    writeTo[1][0] = 0;
    writeTo[1][1] = (4 * p[4] * pow((p[2] - y_n[1]), 3) - (p[5] * pow(u_n, 2)) / (p[0] * p[2] *
         pow((y_n[1] / p[2]), (p[5] + 1)))) / p[3];
    break;
  }
}
void halODE(const double t_n, const double y_n[], const double u_n, const int& state_n, double
     writeTo[], const double p[]) {
  switch (state_n)
  {
  default:
       // Definition der DGL
    double R_w = p[0] * pow(y_n[1] / p[2], p[5]);
    writeTo[0] = (u_n - (y_n[0] * R_w)) / (p[1]); // DGL des Stroms
    writeTo[1] = ((u_n * u_n / (R_w)) - p[4] * pow(y_n[1] - p[2], 4)) / p[3]; // DGL der Temperatur
    break;
  }
}
void halyout(const double y_n[], double y_out[], const double p[]) {
    // Ausgangsfunktion uebergibt lediglich die Zustandsvariablen an die Ausgangsvariable
  y_out[0] = y_n[0];
  y_out[1] = y_n[1];
}
void halParamSize(int& variable_sz, int& outvars_sz, int& param_sz, int& states_num, int& data_res) {
  // Anzahl der Parameter
    param_sz = 6;
  // Anzahl der Systemvariablen
  variable_sz = 2;
  // Anzahl der Ausgangsvariablen
  outvars_sz = 2;
  // Anzahl der States
  states_num = 1;
  // Aufloesung des Arbeitsbereichs
    data_res = 40;
}
void halWriteParams(double* y0, double *p, double* range[2]) {
  // Anfangswert der Zustandsvariablen
    y0[0] = 0;
    y0[1] = 300;
```

```
59
60   // Parameter
61   // Anfangs-Parameter der vereinfachten Halogenlampe
62   double R_c = 0.37; // Elektrischer Widerstand @ T_u in ohm
63   double L = 15e-6; // Induktivitaet in H
64   double T_u = 295; // Umgebungstemperatur in K
65   double c_w = 7e-3; // Waermekapazitaet in J / K
66   double b = 1.5e-12; // Koeffizient zur Berechnung der Abstrahlung in W / K ^ 4
67   double k = 0.9; // Exponent zur Berechnung des elektrischen Wiederstads bei grossen Temperaturen
68
69   // Uebertragung in allgemeine Schreibweise
70   p[0] = R_c;
71   p[1] = L;
72   p[2] = T_u;
73   p[3] = c_w;
74   p[4] = b;
75   p[5] = k;
76
77   // Definition des Arbeitsbereichs
78   // Strom wird definiert von 0A bis 100A
79   range[0][0] = 0;
80   range[1][0] = 100;
81     // Temperatur wird definiert von 295K bis 4000K
82   range[0][1] = 295;
83   range[1][1] = 4000;
84
85   // Eingangsspannung wird definiert von 0V bis 15V
86   range[0][2] = 0;
87   range[1][2] = 15;
88 }
89 void halRL(std::vector<double>& RL, const double *p) {
90   // Werte fuer Induktivitaet und Widerstand, zur Uebergabe an Hardwarekomponente
91   RL.at(0) = 5 * p[0];
92   RL.at(1) = p[1];
93 }
94
95 // Index des initialen Zustands
96 int iniSt_Hal = 0;
97
98 // Definition der Model-Funktion
99 modelFcts getlamp() {
100   modelFcts lampFcts = { halODE, halJF, halrst, halyout, halParamSize, halWriteParams, halRL,
        iniSt_Hal };
101   return lampFcts;
102 };
```