Einführung in die Automatische Differentiation

im Rahmen des Seminars „Automatische Differentiation“
von Dr. Johannes Schlöder und Dr. Ekaterina Kostina,
Sommersemester 2004, Universität Heidelberg

Hans Joachim Ferreau

10.05.2004

Inhaltsverzeichnis

1 Grundidee der Automatischen Differentiation **2**
1.1 Motivation . . . 2
1.2 Kettenregel und ein Beispiel . . . 2
1.3 Vergleich mit anderen Differentiationstechniken . . . 4

2 Konzepte rund um die Funktionsauswertung **5**
2.1 Programmebenen . . . 5
2.2 Dreiteilige Funktionsauswertung und Datenabhängigkeitsrelation . . . 5
2.3 Erweiterungen . . . 7
2.4 Elementarfunktionen und ihre Differenzierbarkeit . . . 8
2.5 Speicherverwaltung . . . 10
2.6 Ein zeitbezogenes Komplexitätsmodell . . . 11

3 Schlussbemerkungen, Literatur **14**

1 Grundidee der Automatischen Differentiation

1.1 Motivation

Die meisten Algorithmen der Numerischen Mathematik und der Optimierung benötigen möglichst genaue Berechnungen von Ableitungswerten. Diese werden entweder direkt, zur Fehlerabschätzung oder zur adaptiven Programmsteuerung verwendet. In manchen Bereichen – etwa bei der numerischen Integration von gewöhnlichen Differentialgleichungen – werden Ableitungen dritter oder noch höherer Ordnung eingesetzt. Entsprechend groß ist der Bedarf nach effizienten Methoden, die beliebige Funktionen nahezu exakt ableiten können. Beliebig heißt hier, dass die verwendeten Funktionen nicht zwangsläufig in Form geschlossener, algebraischer Ausdrücke vorliegen, sondern oft aus umfangreichem Programmcode in diversen Programmiersprachen bestehen. Symbolische Differentiation und Approximation mithilfe finiter Differenzen stoßen in diesem Falle schnell an praktische Grenzen. Die *Automatische Differentiation*[1] hat sich in dieser Situation als vielversprechende Möglichkeit erwiesen, (auch höhere) Ableitungen im Rahmen der Maschinengenauigkeit exakt auszuwerten. Und das mit einem Aufwand, der einem geringen Vielfachen einer zugrundeliegenden Funktionsauswertung entspricht. Die langjährige Annahme, dass Ableitungsinformationen teuer und ungenau seien, gerät damit ins Wanken.

1.2 Kettenregel und ein Beispiel

Die Automatische Differentiation basiert auf der Beobachtung, dass Programmcode zur Funktionsauswertung in elementare, arithmetische Operationen zerlegt werden kann (genau dies ist die Hauptaufgabe von Compilern). Diese Elementarfunktionen (elementals)[2] – z. B. $+$, $-$, $\cdot$, exp, sin – können einfach differenziert und durch die Kettenregel verknüpft werden:

Satz 1.1 (Kettenregel)
Seien X, Y und Z normierte $\mathbb{K}$-Vektorräume und Abbildungen

$$U \xrightarrow{g} V \xrightarrow{f} Z$$

gegeben, wobei U offen in X und V offen Y ist. Sei ferner g differenzierbar in $x \in U$ und f differenzierbar in $y := g(x)$. Dann ist $f \circ g$ differenzierbar in x und es gilt

$$\mathrm{d}(f \circ g)(x) = \mathrm{d}f(y) \circ \mathrm{d}g(x)$$

Für die Ableitungen besagt das

$$(f \circ g)'(x) = f'(y) \cdot g'(x)$$

Sind f und g stetig differenzierbar, dann ist es auch $f \circ g$.
Beweis: siehe [4], Seite 93 □

Korollar 1.2
Gilt im Falle von Satz 1.1

$$\begin{aligned} f:\ \mathbb{R}^m &\to \mathbb{R} \\ g:\ \mathbb{R} &\to \mathbb{R}^m \end{aligned}$$

so folgt

$$\frac{\partial f \circ g}{\partial x}(x) = \frac{\partial f\left(g_1(x), \ldots, g_m(x)\right)}{\partial x} = \left(\frac{\partial f}{\partial y_1}(y), \ldots, \frac{\partial f}{\partial y_m}(y)\right) \cdot \begin{pmatrix} g_1'(x) \\ \vdots \\ g_m'(x) \end{pmatrix} = \sum_{i=1}^{m} \frac{\partial f}{\partial y_i}(y) \cdot g_i'(x)$$

[1] In [1] wird die Bezeichnung *Algorithmische Differentiation* benutzt.
[2] Es wurde durchgängig versucht, deutsche Bezeichnungen für die englischen Fachbegriffe zu finden. Um die Recherche in der Originalliteratur zu vereinfachen, wird der englische Ausdruck beim ersten Auftreten in Klammern angegeben.

Dazu ein einfaches

Beispiel:

Wir wollen die Funktion

$$y = F(x_1, x_2) = (\sin(x_1/x_2) + x_1/x_2 - \exp(x_2))\,(x_1/x_2 - \exp(x_2))$$

und ihre Ableitung im Punkt $(1.5, 0.5)^T$ auswerten. Als eine mögliche Vorgehensweise ergibt sich[3]:

v_{-1}	=	x_1	=	1.5000		
v_0	=	x_2	=	0.5000		
v_1	=	v_{-1}/v_0	=	1.5000/0.5000	=	3.0000
v_2	=	$\sin(v_1)$	=	$\sin(3.0000)$	=	0.1411
v_3	=	$\exp(v_0)$	=	$\exp(0.5000)$	=	1.6487
v_4	=	$v_1 - v_3$	=	$3.0000 - 1.6487$	=	1.3513
v_5	=	$v_2 + v_4$	=	$0.1411 + 1.3513$	=	1.4924
v_6	=	$v_5 \cdot v_4$	=	$1.4924 \cdot 1.3513$	=	2.0167
y	=	v_6	=	2.0167		

Tabelle 1: Auswertung der Funktion F im Punkt $(1.5, 0.5)^T$

Die Funktion wurde in mehrere Elementarfunktionen zerlegt. Nach dem Initialisieren der Eingabevariablen, werden sukzessive die sog. Zwischenvariablen (intermediate variables) v_i, $i > 0$, berechnet und das Resultat schließlich der Ergebnisvariable zugewiesen. Um nun die Funktion beispielsweise nach x_2 zu differenzieren, genügt es jede einzelne Variable v_i (welche genaugenommen auch Funktionen sind) gemäß der Kettenregel nach x_2 abzuleiten:

$$\begin{aligned}
v_{-1} = x_1 \quad &\Longrightarrow \quad \dot{v}_{-1} = 0 \\
v_0 = x_2 \quad &\Longrightarrow \quad \dot{v}_0 = 1 \\
v_1 = \frac{v_{-1}}{v_0} \quad &\Longrightarrow \quad \dot{v}_1 = \frac{\partial v_1}{\partial v_{-1}}\dot{v}_{-1} + \frac{\partial v_1}{\partial v_0}\dot{v}_0 = \frac{1}{v_0}\dot{v}_{-1} + \frac{-v_{-1}}{v_0^2}\dot{v}_0 = \frac{\dot{v}_{-1} - v_1\dot{v}_0}{v_0} \\
&\quad \dots
\end{aligned}$$

(Um nach x_1 abzuleiten genügt es, $\dot{v}_{-1} = 1$ und $\dot{v}_0 = 0$ zu setzen.)

Man erhält folgendes Resultat[3]:

v_{-1}	=	x_1	=	1.5000		
$\dot{v}_{-1}$	=	$\dot{x}_1$	=	0.0000		
v_0	=	x_2	=	0.5000		
$\dot{v}_0$	=	$\dot{x}_2$	=	1.0000		
v_1	=	v_{-1}/v_0	=	1.5000/0.5000	=	3.0000
$\dot{v}_1$	=	$(\dot{v}_{-1} - v_1 \cdot \dot{v}_0)/v_0$	=	$(0.0000 - 3.0000 \cdot 1.0000)/0.5000$	=	−6.0000
v_2	=	$\sin(v_1)$	=	$\sin(3.0000)$	=	0.1411
$\dot{v}_2$	=	$\cos(v_1)\dot{v}_1$	=	$-0.9900 \cdot (-6.0000)$	=	5.9400
v_3	=	$\exp(v_0)$	=	$\exp(0.5000)$	=	1.6487
$\dot{v}_3$	=	$v_3 \cdot \dot{v}_0$	=	$1.6487 \cdot 1.0000$	=	1.6487
v_4	=	$v_1 - v_3$	=	$3.0000 - 1.6487$	=	1.3513
$\dot{v}_4$	=	$\dot{v}_1 - \dot{v}_3$	=	$-6.0000 - 1.6487$	=	−7.6487
v_5	=	$v_2 + v_4$	=	$0.1411 + 1.3513$	=	1.4924
$\dot{v}_5$	=	$\dot{v}_2 + \dot{v}_4$	=	$5.9400 + (-7.6487)$	=	−1.7088
v_6	=	$v_5 \cdot v_4$	=	$1.4924 \cdot 1.3513$	=	2.0167
$\dot{v}_6$	=	$\dot{v}_5 \cdot v_4 + v_5 \cdot \dot{v}_4$	=	$-1.7088 \cdot 1.3513 + 1.4924 \cdot (-7.6487)$	=	−13.7240
y	=	v_6	=	2.0167		
$\dot{y}$	=	$\dot{v}_6$	=	−13.7240		

Tabelle 2: Berechnung der ersten Ableitung (nach x_2) der Funktion F im Punkt $(1.5, 0.5)^T$

Hierbei wurde der sog. Vorwärtsmodus (forward mode) angewandt. Für eine eingehende Beschreibung des Vorwärtsmodus sowie für die Beschreibung des Rückwärtsmodus (reverse mode) verweise ich auf die im Rahmen dieses Seminars noch zu erwartenden Vorträge.

[3] Es wurde mit 5 signifikanten Stellen gerechnet.

1.3 Vergleich mit anderen Differentiationstechniken

Wie das obige Beispiel zeigt, ist die Automatische Differentiation mit dem symbolischen Differenzieren verwandt. Der entscheidende Unterschied ist jedoch, dass die Zwischenvariablen bei der Automatischen Differentiation stets *numerisch* ausgewertet werden. Kommen in einer Funktion bestimmte Teilausdrücke mehrfach vor, so wird ihr Wert nur *einmal* berechnet. Beim symbolischen Differenzieren – wie etwa aus Computeralgebrasystemen bekannt – wird dagegen jeder dieser Teilausdrücke separat behandelt. Dies führt oft zu sehr umfangreichen Darstellungen der Ableitung (und entsprechend hohem Speicherbedarf), was auch durch nachträgliches Vereinfachen nur teilweise behoben werden kann.

Ein besonderer Vorteil der Automatischen Differentiation liegt darin, dass sie ausführbaren Code zum Berechnen der Ableitung liefert. Dieser kann, genau wie die Funktion selbst, im Rahmen der Maschinengenauigkeit ε_{RND} *exakt* ausgewertet werden. Anders sieht dies bei finiten Differenzen aus: Die *einseitige Differenz (one-sided-difference)* nutzt die Definition der (partiellen) Ableitung

$$\frac{\partial f}{\partial x_i}(x) = \lim_{\eta \to \infty} \frac{f(x + \eta e_i) - f(x)}{\eta} \quad = \quad \frac{f(x + \eta e_i) - f(x)}{\eta} + \mathcal{O}(\eta) \quad \text{(nach TAYLOR)}$$

Die richtige Wahl von η ist hierbei entscheidend: Wird es zu klein gewählt, so führt dies zu Rundungsfehlern, wird es zu groß gewählt, ist die Approximation ungenau. In [5] wird als optimale Wahl

$$\eta_{opt} = \sqrt{\varepsilon_{RND}}$$

vorgeschlagen und damit ein Fehler von

$$\left| \frac{\partial f}{\partial x_i}(x) - \frac{f(x + \eta_{opt} e_i) - f(x)}{\eta_{opt}} \right| \approx \sqrt{\varepsilon_{RND}}$$

gezeigt. Das bedeutet, dass dabei die Hälfte der signifikanten Stellen verlorengeht! Auch bei der mit höherem Rechenaufwand verbundenen *zentralen Differenz*

$$\frac{\partial f}{\partial x_i}(x) = \frac{f(x + \eta e_i) - f(e - \eta e_i)}{2\eta} + \mathcal{O}(\eta^2)$$

bleiben nur zwei Drittel der signifikanten Stellen übrig. Derartiges tritt bei der Automatischen Differentiation nicht auf; wir formulieren daher die

Regel 0: *Automatische Differentiation ist nicht von Abbruchfehlern (truncation errors) betroffen.*

Dennoch ist die Automatische Differentiation kein „Allheilmittel“:

Regel 1: *In manchen Anwendungen ist der Einsatz von Differenzenquotienten sinnvoll.*

2 Konzepte rund um die Funktionsauswertung

Wir wollen in diesem Kapitel mehrere Konzepte vorstellen, die den Prozess der Funktionsauswertung konkretisieren und ihn damit weiteren theoretischen Betrachtungen zugänglich macht. Als erstes ist die Frage zu klären, in welcher Form unsere zu untersuchende Funktion

$$F: \ D \subseteq \mathbb{R}^n \ \rightarrow \ \mathbb{R}^m$$

vorliegen muss.

2.1 Programmebenen

Wie im ersten Teil erläutert, ist die Forderung nach einem geschlossenen, algebraischen Ausdruck zu speziell. Es soll im Folgenden genügen, wenn F als Computerprogramm vorliegt (natürlich lässt sich ein algebraischer Ausdruck auch als solches formulieren). Dabei unterscheiden wir hinsichtlich der Programmsteuerung drei Programmebenen:

1. Ein *Auswertungsprogramm (evaluation program)* unterliegt keinen Einschränkungen. In ihm dürfen alle Programmiermethoden verwendet werden.

2. Eine *Auswertungsprozedur (evaluation procedure)* zeichnet sich durch eine geradlinige Programmausführung aus. In ihr sind Verzweigungen, Rekursionen und Schleifen nicht zulässig. Die Prozedur kann Parameter übergeben bekommen und es dürfen Unterprozeduren mit Parameterübergabe (auch merhfach, aber nicht rekursiv) aufgerufen werden. Alle sonstigen Befehle, wie arithmetische Operationen oder Variablenmanipulation, sind als essentielle Bestandteile natürlich möglich.

   Jedes Auswertungsprogramm lässt sich in eine Auswertungsprozedur umwandeln: Sobald die für die Programmsteuerung relevanten Eingabedaten feststehen, kann man durch *bedingtes Compilieren (instantiation)* die Verzweigungsbedingungen auswerten und die entsprechenden Programmteile an die passenden Stellen des Auswertungsprogrammes schreiben. Rekursionen werden aufgelöst und Schleifen durch explizites Wiederholen der im Inneren befindlichen Befehle ersetzt (loop unrolling).

3. Schließlich entsteht der *Auswertungspfad (evaluation trace)* aus einer Auswertungsprozedur dadurch, dass nun allen Variablen Werte zugewiesen und die Unterprozeduren direkt eingebettet werden (inlinig). Dieser Vorgang wird auch *Expansion (expansion)* genannt.

Wir erhalten also folgende Beziehung:

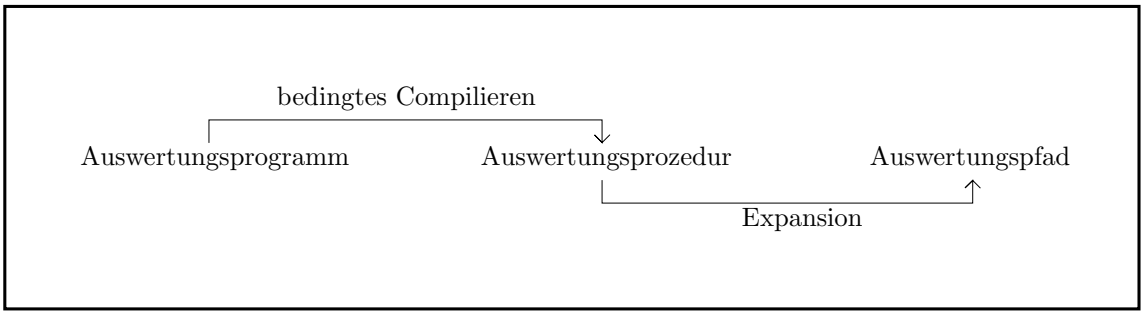

Abbildung 1: Die drei Programmebenen

2.2 Dreiteilige Funktionsauswertung und Datenabhängigkeitsrelation

In dem Beispiel auf Seite 3 wurde die Funktion F (z. B. von einem Compiler) in Elementarfunktionen zerlegt. Die Funktionsauswertung ist dreigeteilt:

1. Teil: Die Eingabevariablen x_i werden in interne Variablen v_{i-n}, $1 \leq i \leq n$, kopiert.

2. Teil: Die Zwischenvariablen v_i, $1 \leq i \leq l$, werden berechnet.

3. Teil: Das Ergebnis v_{l-m+i} wird in die Ergebnisvariablen y_i, $1 \leq i \leq m$, kopiert.

Die Variablen v_i werden dabei stets so nummeriert, dass sie folgendem Schema entsprechen:

$$v = (\underbrace{v_{1-n}, v_{2-n}, \ldots, v_{-1}, v_0}_{x \in \mathbb{R}^n}, v_1, v_2, \ldots, v_{l-m-1}, v_{l-m}, \underbrace{v_{l-m+1}, v_{l-m+2}, \ldots, v_{l-1}, v_l}_{y \in \mathbb{R}^m}) \in \mathbb{R}^{n+l}$$

Jedem v_i, $1 \leq i \leq l$, wird dabei das Ergebnis einer Elementarfunktion $\varphi_i \in \Phi$ zugewiesen. Dabei bezeichnet Φ die Menge aller Elementarfunktionen, deren Inhalt wir in Abschnitt 2.4 diskutieren werden. φ_i kann dabei von jeder Variable v_j, $j < i$, abhängen:

Definition 2.1 (Datenabhängigkeitsrelation)
Auf der Indexmenge $I = \{1-n, \ldots, l\}$ des Variablenvektors v definieren wir die antisymmetrische Relation

$$j \prec i \quad :\Longleftrightarrow \quad v_i = \varphi_i(v_j) \text{ hängt direkt von } v_j \text{ ab}$$

Sie heißt Datenabhängigkeitsrelation (data dependence relation)*. Ihr* transitiver Abschluss

$$j \prec^* i \quad :\Longleftrightarrow \quad \exists\, p_1, \ldots, p_r \in I : \; j \prec p_1 \;\wedge\; p_k \prec p_{k+1} \; \forall\, k \in \{1, \ldots, r-1\} \;\wedge\; p_r \prec i$$

stellt laut [1], S. 19, eine partielle Ordnung auf I dar[4]*. $i \prec i$ entspricht dabei einer Iteration, welche vorerst nicht zulässig ist.*

Die Datenabhängigkeiten können durch einen gerichteten, zusammenhängenden, azyklischen Graphen, dem sog. *Berechnungsgraphen (computational graph)*, veranschaulicht werden. Darin werden die Variablen v_i durch Knoten (Wurzeln, falls $i \leq 0$ und Blätter, falls $i > l - m$) dargestellt. Es läuft genau dann eine Kante von Knoten v_j zu Knoten v_i, wenn $j \prec i$ gilt. In unserem Beispiel gilt u. a. $1 \prec 4$ und $3 \prec 4$ sowie $0 \not\prec 4$ und $0 \prec^* 4$:

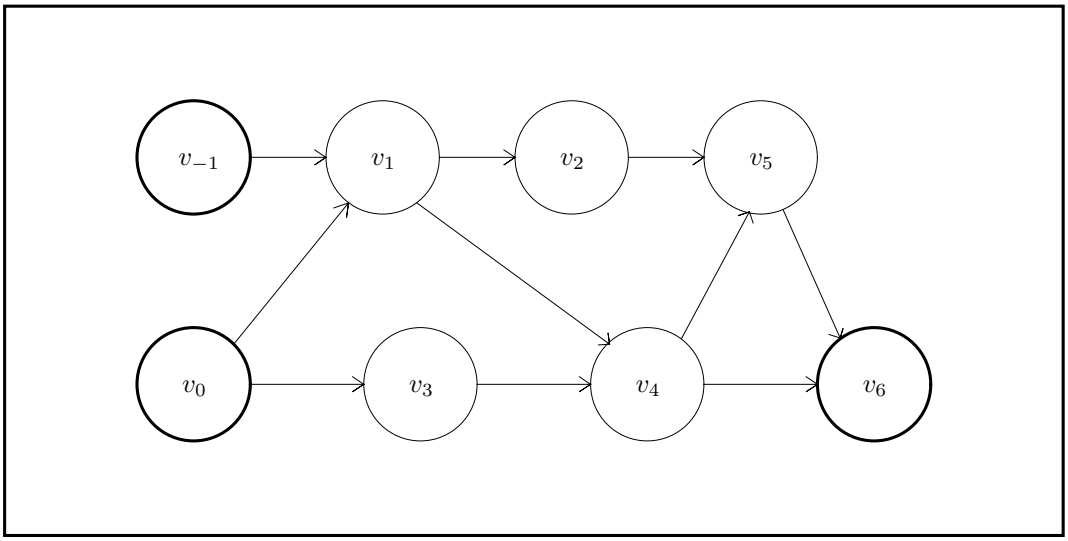

Abbildung 2: Berechnungsgraph des Beispiels von Seite 3

Auswertungsprozeduren können auf Grund von Definiton 2.1 formalisiert werden:

v_{i-n}	$=$	x_i	$\forall\, i \in \{1, \ldots, n\}$
v_i	$=$	$\varphi_i(v_j)_{j \prec i}$	$\forall\, i \in \{1, \ldots, l\}$
y_i	$=$	v_{l-m+i}	$\forall\, i \in \{1, \ldots, m\}$

Tabelle 3: Formale Beschreibung einer Auswertungsprozedur

Dies liefert uns die Funktion F als Komposition von Elementarfunktionen

$$\begin{aligned} & F(x_1, \ldots, x_n) &&= (y_1, \ldots, y_m) \\ \Longleftrightarrow \quad & F(v_{1-n}, \ldots, v_0) &&= (v_{l-m+1}, \ldots, v_l) = (\varphi_i \circ \varphi_{i,1} \circ \ldots \circ \varphi_{i,n_i}(v_{j_i}))_{i=l-m+1,\ldots,l} \\ & && \forall\, i \in \{l-m+1, \ldots, l\} : \; \varphi_{i,k} \in \{\varphi_{1-n}, \ldots, \varphi_{i-1}\} \; \forall\, k \in \{1, \ldots, n_i\}, \\ & && \qquad 0 \leq n_i, \; 1-n \leq j_i \leq 0 \end{aligned} \tag{2.1}$$

auf welche wir die Kettenregel (Satz 1.1) anwenden können.

[4] Dagegen fordert z. B. [3], S. 27, von einer solchen auch *Reflexivität*, welche hier i. Allg. nicht erfüllt ist.

Die obige Darstellung der Funktion F ist keineswegs eindeutig: Zwei unterschiedliche Auswertungsprozeduren können dieselbe Funktion beschreiben. Schon einfache symbolische Umformungen können zu unterschiedlicher Effizienz und numerischer Stabilität führen. Diese Beobachtung liegt jedoch nicht in der Automatischen Differentiation begründet, sondern ist vielmehr eine allgemeiner Effekt. Wir wollen uns daher vorläufig von dieser Problematik befreien:

Regel 2: *Ein* abgeleitetes *Auswertungsprogramm ist genau so gut oder schlecht wie das ursprüngliche Auswertungsprogramm.*

2.3 Erweiterungen

Wie in Tabelle 3 beschrieben, lassen sich die Zwischenvariablen v_i als Funktionen $v_i(v_j), 1 \leq j < i \leq l-m$, und letztlich als Funktionen $v_i(x)$, $x \in \mathbb{R}^n$, auffassen. Diesen Zwischen*funktionen*, die bisher nur Skalare als Ergebnis haben, wollen wir nun erlauben *vektorwertig* zu sein. Wir definieren

$$m_i := \dim(v_i) \geq 1 \quad \forall\, i \in \{1, \ldots, l-m\}$$

und da die Komponenten x_i und y_i von x und y per definitionem Skalare sind, setzen wir noch

$$m_i := 1 \quad \forall\, i \in \{1-n, \ldots, 0\} \cup \{l-m+1, \ldots, l\}$$

m_i ist also die Dimension des Bildraumes von φ_i. Mit dieser Verallgemeinerung ist es z. B. möglich, einfache Funktionen der linearen Algebra als Elementarfunktionen hinzuzufügen. Man kann auf diese Weise sogar die gesamte Auswertungsprozedur der Funktion F als eine (Super-)Elementarfunktion definieren. Diese kann dann in hierarchisch höheren Auswertungsprozeduren/-programmen benutzt werden. Allgemein kann so einer Menge von Elementarfunktionen jede beliebige Komposition ihrer Elemente hinzugefügt werden.
Zur Vereinfachung der Notation wollen wir – wie bereits stillschweigend praktiziert – bei *mehrstelligen* Zwischenfunktionen die verschiedenen Argumente zu *einem* Argument zusammenfassen. Dazu verschmelzen wir alle Argumente zu einem Argumentvektor

$$u_i := (v_j)_{j \prec i} \in \mathbb{R}^{n_i} \quad \forall\, i \in \{1, \ldots, l\}$$

wobei n_i die Dimension des Definitionsbereiches von φ_i ist:

$$n_i := \sum_{j \prec i} m_j \quad \forall\, i \in \{1, \ldots, l\}$$

Bei dieser Vorgehensweise wird immer der gesamte Vektor v_j als Teil von u_i an v_i übergeben, auch wenn v_i nur von einer einzigen Komponente von v_j abhängt. Daher dient diese Konvention vorrangig theoretischen Betrachtungen, da sie bei der praktischen Umsetzung zu Effizienzeinbußen führen kann.

Eine andere Erweiterung besteht darin, Tabelle 3 als nichtlineares Gleichungssystem zu formulieren:

$$0 = E(x; v) := (\varphi_i(u_i) - v_i)_{i=1-n,\ldots,l}$$

Die ersten n Gleichungen entsprechen der Initialisierung mit dem Eingabevektor x. Weiter wollen wir o. B. d. A. annehmen, dass die Komponenten des Ergebnisvektors y paarweise unabhängig voneinander sind. Definieren wir nun die sog. *partiellen Elementarableitungen (elemental partials)* durch

$$c_{ij} := c_{ij}(u_i) := \frac{\partial \varphi_i}{\partial v_j}(u_i) \quad \forall\, i, j \in \{1-n, \ldots, l\}$$

so hat die (erweiteterte) JACOBI-Matrix von $E(x; v)$ die Form

$$E'(x; v) = (c_{ij} - \delta_{ij})_{i,j=1-n,\ldots,l} = C - Id_{n+l}$$

$$= \left(\begin{array}{cccc|cccccccc} -1 & 0 & \cdots & 0 & 0 & \cdots & \cdots & 0 & 0 & \cdots & \cdots & 0 \\ 0 & \ddots & \ddots & \vdots & \vdots & \ddots & & \vdots & \vdots & \ddots & & \vdots \\ \vdots & \ddots & \ddots & 0 & \vdots & & \ddots & \vdots & \vdots & & \ddots & \vdots \\ 0 & \cdots & 0 & -1 & 0 & \cdots & \cdots & 0 & 0 & \cdots & \cdots & 0 \\ \hline * & \cdots & \cdots & * & -1 & 0 & \cdots & 0 & 0 & \cdots & \cdots & 0 \\ \vdots & \ddots & & \vdots & * & \ddots & \ddots & \vdots & \vdots & \ddots & & \vdots \\ \vdots & & \ddots & \vdots & \vdots & \ddots & \ddots & 0 & \vdots & & \ddots & \vdots \\ * & \cdots & \cdots & * & * & \cdots & * & -1 & 0 & \cdots & \cdots & 0 \\ * & \cdots & \cdots & * & * & \cdots & \cdots & * & -1 & 0 & \cdots & 0 \\ \vdots & \ddots & & \vdots & \vdots & \ddots & & \vdots & 0 & \ddots & \ddots & 0 \\ \vdots & & \ddots & \vdots & \vdots & & \ddots & \vdots & \vdots & \ddots & \ddots & 0 \\ * & \cdots & \cdots & * & * & \cdots & \cdots & * & 0 & \cdots & 0 & -1 \end{array}\right) = \left(\begin{array}{ccc} -Id_n & 0 & 0 \\ B & L - Id_{l-m} & 0 \\ R & T & -Id_m \end{array}\right)$$

Sie ist eine normierte, untere Dreiecksmatrix, da nach Vorraussetzung

$$\left(v_i = \varphi_i\left((v_j)_{j \prec i}\right) \;\Rightarrow\; j < i\right) \quad \Longrightarrow \quad c_{ij} = \frac{\partial \varphi_i}{\partial v_j}(u_i) = 0, \text{ falls } j \geq i$$

gilt. Insbesondere ist sie – und die rechte, untere Teilmatrix – regulär. Deshalb gibt es nach dem Satz über implizite Funktionen (siehe [4], Seite 113f.) eine Funktion $g : \mathbb{R}^n \to \mathbb{R}^l$, so dass

$$E(x; v) = 0 \quad \Longleftrightarrow \quad v_i = g_i(x) \quad \forall\, i \in \{1, \ldots, l\}$$

gilt (in geeigneten Umgebungen). Insbesondere sind dadurch die m Komponenten des Ergebnisvektors y implizit durch Funktionen in x bestimmt. Aus dieser „mathematischeren“ Formulierung des Problems kann man einige Beziehungen und Algorithmen direkt herleiten. Sie wird uns im Vortrag über den Rückwärtsmodus wiederbegegnen.

2.4 Elementarfunktionen und ihre Differenzierbarkeit

Nun klären wir welche Funktionen in der Menge der Elemtarfunktionen Φ enthalten sein sollten. Darauf gibt es natürlich keine eindeutige Antwort. Die grundlegenden aritmetischen Operationen Addition, Multiplikation und der unäre Vorzeichenwechsel sind dabei ebenso unverzichtbar, wie die Initialisierung mit einer Konstanten c. Diese minimale Menge

$$\Phi_{min} = \{+, \cdot, -, c\}$$

heißt *polynomialer Kern (polynomial core)*, da man mit ihr sämtliche Polynome auswerten kann. Darüber hinaus sind weitere Funktionen wie $\exp(a)$, $\log(a)$, $\sin(a)$, $\cos(a)$ und $1/a$ wünschenswert. Da deren Berechnung meist durch table look-ups (also letztlich Verzweigungen) und anschließenden iterative Verfahren realisiert wird und wir Derartiges in unseren Auswertungsprozeduren nicht zugelassen haben, sind sie für unsere Betrachtungen sogar zwingend erforderlich.

	glatt	Lipschitz-stetig	sonstige
essentiell	$a+b,\ a\cdot b,\ -a,\ c,$ $1/a,\ \exp(a),\ \log(a),$ $\sin(a),\ \cos(a),\ \lvert a\rvert^c\,(c>1),\ \ldots$	$\lvert a\rvert,\ \lvert(a,b)\rvert = \sqrt{a^2+b^2}$	heaviside(a)
optional	$a-b,\ a/b,\ c\cdot a,\ c\pm a,$ $a^k,\ a^c,\ \tan(a),\ \arcsin(a),\ \ldots$	$\max(a,b),\ \min(a,b)$	sign(a), if-then-else
vektorwertig	$\sum_k a_k\cdot b_k,\ \sum_k c_k\cdot a_k$	$\max\{\lvert a_k\rvert\}_k,\ \sum_k \lvert a_k\rvert,\ \left(\sum_k a_k^2\right)^{1/2}$	–

$$k \in \mathbb{N}, \quad a, b, a_k, b_k \in \mathbb{R} \text{ (Variablen)}, \quad c, c_k \in \mathbb{R} \text{ (Konstanten)}$$

Tabelle 4: Mögliche Wahl von Φ, der Menge der Elementarfunktionen

Die Heaviside-Funktion dient dazu, Verzweigungen dem Funktionsformalismus zu unterwerfen. Allerdings hat sie – genauso wie etwa der Betrag – den Nachteil, dass sie nicht differenzierbar ist. Da dies eine notwendige Voraussetzung zur Anwendung der Kettenregel ist, wollen wir zunächst nur glatte Funktionen verwenden.

Für unsere weiteren Betrachtungen machen wir folgende

Annahme ED (Elementare Differenzierbarkeit): *Alle Elementarfunktionen $\varphi_i \in \Phi$ sind in ihrem* offenen *Definitionsbereich $D_i \in \mathbb{R}^n$ (mindestens) d-mal stetig-differenzierbar, $0 \le d \le \infty$.*

Die Funktionen des polynomialen Kerns erfüllen die Annahme offensichtlich auf ganz $\mathbb{R}$ bzw. $\mathbb{R}^2$. Bei der Division $\varphi_i(a,b) = a/b$ muss die Achse $b = 0$ herausgenommen werden.

Satz 2.2
Falls die Annahme ED gilt, ist die Menge

$$D := \{x \in \mathbb{R}^n \mid y = F(x) \text{ ist durch die Auswertungsprozedur in Tabelle 3 wohldefiniert}\}$$

eine offene *Teilmenge des $\mathbb{R}^n$ und es gilt: $F \in \mathcal{C}^d(D)$, $0 \le d \le \infty$.*

Beweis: Ist F in x wohldefiniert, so muss $u_i = u_i(x) \in D_i$ für alle $i \in \{1, \ldots, l\}$ gelten. Da alle D_i nach Voraussetzung offen sind, folgt aus Satz 1.1, dass alle v_i und insbesondere auch alle $y_i = F_i(x) = v_{l-m+i}$ in einer vollen Umgebung von x (mindestens) d-mal stetig-differenzierbar sind. □

Korollar 2.3
Werden von der Vektorfunktion $F : D \subseteq \mathbb{R}^n \to \mathbb{R}^m$ Ableitungen bis zur Ordnung $d > 0$ benötigt, müssen die Elementarfunktionen $\varphi_i \in \Phi$ so gewählt werden, dass sie auf ihren offenen Definitionsbereichen d-mal stetig-differenzierbar sind und die Auswertungsprozedur für alle $x \in D$ wohldefiniert ist.

Es kann vorkommen, dass der maximale Definitionsbereich D leer ist, weil für jedes $x \in \mathbb{R}^n$ eine Zwischenfunktionen ein Argument außerhalb ihrer Definitionsbereiches übergeben bekommt.
Die Zahl d kann durch verschiedene Umstände beschränkt sein. Falls ein Programm eine Funktion durch Splines interpoliert, so kann man typerscherweise nicht mehr als $d = 2$ erwarten. Auch wenn Potenzen der Betragsfunktion auftauchen, ist d endlich. Nehmen wir als Beispiel die Funktion

$$F(x) = \sqrt{x^6} = |x|^3 \quad \Longrightarrow \quad F'(x) = 3x\,|x|$$

Sie kann problemlos mit

$$v_0 := x,\ v_1 := v_0^6,\ y = v_2 := \sqrt{v_1} \quad \text{oder} \quad \widetilde{v}_0 := x,\ \widetilde{v}_1 := |\widetilde{v}_0|\,,\ y = \widetilde{v}_2 := \widetilde{v}_1^3$$

ausgewertet werden. Ihre Ableitung hat im Nullpunkt den Wert 0, aber weder $\dot{v}_2(0)$ noch $\dot{\widetilde{v}}_1(0)$ ist endlich. Man umgeht diese Schwierigkeit, in dem man

$$\varphi(x) = |x|^c\,, \quad \varphi'(x) = cx\,|x|^{c-2} \in \mathcal{C}^{\lfloor c-1 \rfloor}(\mathbb{R})$$

als Elementarfunktion einführt und ihre Ableitung als „Mikrocode“ mitliefert.

Im Allgemeinen muss bei der Funktion $a^c := \exp(c \cdot \log(a))$ mit $c \in \mathbb{R} \setminus \mathbb{Z}$, $a > 0$ vorausgesetzt werden. Dies gilt auch für die Quadratwurzel ($c = 0.5$). Die Hinzunahme des Nullpunktes würde der Voraussetzung eines offenen Definitionsbereiches widersprechen. Da bereits durch die Zahldarstellung im Rechner kleine Störungen auftreten, ist die Forderung nach einem offenen Definitionsbereich durchaus sinnvoll.

Nach diesen Überlegungen wählen wir die Menge der Elementarfunktionen

$$\Phi := \left\{c,\ \pm,\ \cdot,\ \exp,\ \log,\ \sin,\ \cos,\ ^{-1},\ \ldots\right\}$$

so, dass deren Elemente die Annahme ED für ein gemeinsames $d \ge 1$ erfüllen. Außerdem bezeichnen wir mit

$$\mathcal{F} := \operatorname{span}(\Phi) \subseteq \mathcal{C}^d(\mathbb{R}^n)$$

die Menge aller Funktionen, die als Auswertungsprozedur mit Elementen aus Φ definiert werden können.

2.5 Speicherverwaltung

Bisher haben wir die Zwischenvariablen nur aus der mathematischen Sicht betrachtet: Allen v_i wurde genau einmal ein Wert zugewiesen und diese Zuweisung hatte keinen Effekt auf die anderen Zwischenvariablen. Die Reihenfolge der Auswertung war beliebig, solange sichergestellt war, dass alle v_j definiert wurden, bevor sie das erste Mal als Teil eines Arguments $u_i = (v_j)_{j \prec i}$ auftauchen.
In realen Programmen ist diese Form der Speicherverwaltung nicht zufriedenstellend. Wir wollen den Zwischenvariablen v_i, $i > 0$, daher erlauben, sich Speicherplätze (locations) zu teilen. Dazu folgende

Definition 2.4 (Adressierungsfunktion)
Eine Adressierungsfunktion (adressing scheme) *ist eine Abbildung*

$$\& : \ \{1, \ldots, l\} \ \rightarrow \ \mathcal{P}(\mathcal{R}) \setminus \emptyset, \quad \mathcal{R} := \{1, \ldots, r\}, \ r \in \mathbb{N}$$

die jedem v_i, $i > 0$, eine nichtleere Teilmenge $\&v_i = \&i$ der Speicherplätze $\mathcal{R}$ zuordnet.

Damit die Speicherplätze nicht wiederverwendet werden, bevor ihr Inhalt das letzte Mal gebraucht wird, fordern wir

$$j \prec i, \ j < k \leq i \quad \Longrightarrow \quad \&v_k \neq \&v_j \ \left(\overset{(*)}{\Longleftrightarrow} \ \&v_k \cap \&v_j = \emptyset \right) \tag{2.2}$$

und nennen die Adressierungsfunktion & in diesem Fall *vorwärtskompatibel (forward compatible)*. Die Äquivalenz $(*)$ dient hierbei zur Vereinfachung der Notation: Sie besagt, dass sich verschiedene Adressbereiche nicht überlappen dürfen.
Beim Rückwärtsmodus wird analog eine *Rückwärtskompatibilität (reverse compatibility)*

$$j \prec i, \ j \leq k < i \quad \Longrightarrow \quad \&v_i \neq \&v_k \tag{2.3}$$

gefordert. Eine Adressierungsfunktion &, welche beide Bedingungen erfüllt, heißt *zweifach kompatibel (two-way compatible)*.
Die Bedingungen (2.2) und (2.3) kann man dahingehend abschwächen, dass man zusätzlich eine Zuweisung

$$j \prec i, \quad \&v_i = \&v_j$$

zulässt. Eine Zwischenvariable darf also ihr eigenes Argument überschreiben, falls dieses nicht mehr benötigt wird. Man spricht in diesem Falle von *schwacher* Vorwärts- bzw. Rückwärtskompatibilität.
Wir wollen das Eingangsbeispiel von Seite 3 entsprechend umformulieren:

v_1	$=$	$\frac{v_{-1}}{v_0}$	v_1	$=$	$\frac{v_{-1}}{v_0}$
v_2	$=$	$\sin(v_1)$	v_2	$=$	$\exp(v_0)$
v_3	$=$	$\exp(v_0)$	v_3	$=$	$v_1 - v_2$
v_4	$=$	$v_1 - v_3$	v_1	$=$	$\sin(v_1)$
v_1	$=$	$v_2 + v_4$	v_2	$=$	$v_1 + v_3$
v_2	$=$	$v_1 \cdot v_4$	v_3	$=$	$v_3 \cdot v_2$
y	$=$	v_2	y	$=$	v_3

Tabelle 5: Speichereffizientere Formulierung des Eingangsbeispiels mit starker (links) und schwacher (rechts) Vorwärtskompatibilität

Wir sehen, dass die Anzahl der benötigten Speicherplätze von der verwendeten Adressierungsfunktion abhängt. Wir führen in Abhängigkeit von der auszuwertenden Funktion F die Bezeichnung

$$RAM(F) := r := \max \{k \in \mathbb{N} \mid k \in \&v_i, \ 1 \leq i \leq l\}$$

ein. Sie deutet an, dass der Zugriff auf die Speicherplätze in der Regel keinem speziellen Muster genügt, sondern zufällig erfolgt.

Wir wollen nun eine weitere Bedingung an unsere Auswertungsprozedur stellen, die sog. *Alias-Sicherheit (alias-safety)*. Auch wenn die Vorwärtskompatibilität – z. B. durch Programmierfehler – nicht erfüllt ist, wird eine Funktion $\widetilde{F}(x)$ ausgewertet. Wir möchten sicherstellen, dass auch in diesem Falle die falsche

Funktion $\widetilde{F}$ korrekt abgeleitet wird, auch wenn dies einen höheren Aufwand bedeutet. Gilt z. B. $\&v_0 = \&v_1$ beim Auswerten der beiden ersten Zeilen

$$\begin{aligned} v_1 &= v_{-1}/v_0 \\ \dot{v}_1 &= \frac{\dot{v}_{-1}v_0 - v_{-1}\dot{v}_0}{v_0^2} = \frac{\dot{v}_{-1} - v_1\dot{v}_0}{v_0} \end{aligned}$$

des Mittelteils von Tabelle 2, so wird $\dot{v}_1$ mit einem falschen Nenner berechnet. Beim Vorwärtsmodus kann dies immer mit dem Vertauschen von Funktionswert- und Ableitungsberechnung korrigiert werden. Im Allgemeinen ist dies etwas aufwändiger.

2.6 Ein zeitbezogenes Komplexitätsmodell

Eine besondere Stärke der Automatischen Differentiation ist es, dass der Aufwand zur Berechnung der Ableitung einer Funktion sicher vorhergesagt werden kann: Er beträgt meist höchstens das Vier- bis Fünffache des Aufwandes, der nötig ist, um die zugrundeliegende Funktion auszuwerten. Um dies formal beweisen zu können, wollen wir in diesem Abschnitt ein zeitbezogenes Komplexitätsmodell einführen, welches die Anzahl verschiedener arithmetischer Operationen und die Speicherzugriffe zählt. Dabei soll jedoch nicht verschwiegen werden, dass dieses Vorgehen für heutige Rechnern oft unzureichend ist. Da der Speicher in der Regel nicht uniform ist, werden die realen Werte stark durch Zugriffszeiten und Größe des Caches und des Hauptspeichers beeinflusst. Bei Parallelrechnern muss das Programm oft völlig umstrukturiert werden, um gute Werte für Speedup/Effizienz zu erreichen.
Wir geben daher an dieser Stelle nur ein Grundkonzept für die Bewertung der sequentiellen Ausführung einer Auswertungsprozedur, welches bei Bedarf erweitert werden kann. Zunächst bezeichnen wir mit

$$eval(F) \quad \text{bzw.} \quad task(F)$$

die Auswertung einer (aus Elementarfunktionen zusammengesetzten) Funktion F bzw. eine andere additive Aufgabe – etwa die Auswertung der Ableitung. Additiv meint in diesem Zusammenhang, dass sich die Aufgabe in viele Elementaraufgaben zerlegen lässt, die Schritt für Schritt unabhängig voneinander ausgeführt werden können. Entsprechend bezeichnen wir mit

$$eval'(\varphi) \quad \text{bzw.} \quad task'(\varphi)$$

die Auswertung einer Elementarfunktion bzw. einen auf eine Elementarfunktion bezogenen Teil der Aufgabe. Bei der Funktionsauswertung nehmen wir an, dass sich der Aufwand der Elementarfunktionen zum Gesamtaufwand der Funktionsauswertung summiert (also $eval'(F) = eval(F)$ gilt). Dagegen gilt oft $task(\varphi) < task'(\varphi)$: Ein auf eine Elementarfunktion angewandter Teil der Aufgabe braucht separat ausgeführt oft etwas weniger Rechenaufwand als im Kontext der Gesamtaufgabe. Wir machen daher folgende

Annahme TA (Zeitliche Additivität (temporal additivity)):

$$\begin{aligned} WORK\,\{task(F)\} &\leq \sum_{i=1}^{l} WORK\,\{task'(\varphi_i)\} \\ WORK\,\{eval(F)\} &= \sum_{i=1}^{l} WORK\,\{eval(\varphi_i)\} \end{aligned}$$

Dabei sind $WORK\,\{task(F)\}$ und $WORK\,\{eval(F)\}$ Vektoren des $\mathbb{R}^{\dim(WORK)}$ mit nichtnegativen Komponenten. Die Ungleichung ist komponentenweise zu verstehen.

Wir wählen $\dim(WORK) = 4$ und lassen unser Komplexitätsmodell folgende Operationen zählen:

$$WORK\,\{task(F)\} := \begin{pmatrix} MOVES(F) \\ ADDS(F) \\ MULTS(F) \\ NLOPS(F) \end{pmatrix} := \begin{pmatrix} \text{Zahl der Speicherzugriffe (Lesen/Schreiben) von } F \\ \text{Zahl der Additionen/Subtraktionen von } F \\ \text{Zahl der Multiplikationen von } F \\ \text{Zahl der nichtlinearen Operationen von } F \end{pmatrix}$$

Wie bereits angedeutet, machen wir bei den Speicherzugriffen keinerlei Unterscheidung zwischen Registern und Festplatte, obwohl letztere teilweise um den Faktor 10000 langsamer ist. Da eine Division oft deutlich länger als eine Multiplikation dauert, behandelt wir sie wie eine Multiplikation und eine nichtlineare Operation. Allgemein benötigt etwa die Addition (als Elementaroperation) 3 Speicherzugriffe – 2 Argumente lesen, Ergebnis zurückschreiben – und eine Addition (als Kostenmaß), also

$$WORK\{eval(+)\} = (3,1,0,0)^T$$

Bezeichnet ψ eine nichtlineare Operation ($1/a$, $\exp(a)$, $\log(a)$, $\sin(a)$ etc.) so ergeben sich folgende Werte:

	c	$\pm$	$\cdot$	ψ
$MOVES$	1	3	3	2
$ADDS$	0	1	0	0
$MULTS$	0	0	1	0
$NLOPS$	0	0	0	1

Tabelle 6: Komplexität der Elementarfunktionen

Wir nennen die Kostenmatrix für die Funktionsauswertung (Tabelle 6) W_{eval}. Für eine Funktion F bezeichne $|F| \in \mathbb{R}^{dim(WORK)}$ einen Vektor (elemental frequency vector of F), dessen Komponenten l_i die Anzahl der Initialisierungen, der Additionen, der Multiplikationen und der nichtlinearen Operationen enthalten. Dann ergibt sich

$$WORK\{eval(F)\} = W_{eval}\,|F| = \begin{pmatrix} 1 & 3 & 3 & 2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} l_1 \\ l_2 \\ l_3 \\ l_4 \end{pmatrix}$$

Jetzt wollen wir die tatsächliche *Laufzeit (runtime)* $TIME$ bestimmen. Dafür definieren wir das *Laufzeit-Funktional*

$$TIME\{task(F)\} := w^T WORK\{task(F)\}, \quad w \in \mathbb{R}^{dim(WORK)} \tag{2.4}$$

Die Komponenten von w sind positiv und enthalten etwa die Anzahl der CPU-Zyklen für die einzelnen Operationen. Unser Komplexitätsmodell wird dadurch plattformunabhängiger, da es für einfache Abschätzungen ausreicht, den Vektor w passend für das verwendete System zu wählen. Wir benutzen die Zeit für eine Addition als Einheit und setzen

$$w^T = (\mu, 1, \pi, \nu)^T \quad \text{mit} \quad \mu \geq \max(1, \pi/2),\ \pi \geq 1,\ \nu \geq 2\pi$$

Da alle Komponenten von w positiv sind, können wir $TIME$ als gewichtete L_1-Norm von $WORK$ auffassen. Mit Hilfe der induzierten Matrixnorm erhalten wir daher ein skalares Maß für die Komplexität im Verhältnis zur reinen Funktionsauswertung.
Unter der Annahme TA leiten wir nun eine wichtige Schranke her:

$$\begin{aligned} \frac{TIME\{task(F)\}}{TIME\{eval(F)\}} &= \frac{w^T WORK\{task(F)\}}{w^T WORK\{eval(F)\}} \leq \frac{w^T \sum_{i=1}^{l} WORK\{task'(\varphi_i)\}}{w^T \sum_{i=1}^{l} WORK\{eval(\varphi_i)\}} \\ &= \frac{\sum_{i=1}^{l} TIME\{task'(\varphi_i)\}}{\sum_{i=1}^{l} TIME\{eval(\varphi_i)\}} \leq^{5} \max_{1\leq i\leq l} \frac{TIME\{task'(\varphi_i)\}}{TIME\{eval(\varphi_i)\}} \\ &\leq \sup_{\varphi\in\Phi} \frac{TIME\{task'(\varphi)\}}{TIME\{eval(\varphi)\}} \end{aligned} \tag{2.5}$$

Wenn Φ nur endlich viele Elementarfunktionen enthält, haben wir damit eine einheitliche Schranke für das Laufzeitverhältnis von $task(F)$ zu $eval(F)$ bewiesen. Doch genaugenommen enthält bereits der polynomiale Kern unendlich viele Elemente, da wir (theoretisch) beliebig viele verschiedene Werte c initialisieren können. Bevor wir eine Bedingung angeben, um auch im Falle $|\Phi| = \infty$ eine derartige Schranke abzuleiten, formulieren wir noch eine

[5] *Beweis:* Seien $a, b, c, d > 0$ und o. B. d. A. $\frac{a}{b} \geq \frac{c}{d}$: $\Leftrightarrow\ \frac{a}{b}d \geq c \ \Leftrightarrow\ a + \frac{a}{b}d = \frac{a}{b}b + \frac{a}{b}d \geq a + c \ \Leftrightarrow\ \frac{a}{b}(b+d) \geq a + c$ $\Leftrightarrow\ \frac{a}{b} \geq \frac{a+c}{b+d}$ Daraus folgt die Behauptung per Induktion, da der Fall $l = 1$ trivial ist. □

Definition 2.5 (linear beschränkte Komplexität)
Wir sagen eine Aufgabe task(F) hat linear beschränkte Komplexität (linearly bounded complexity) über $\mathcal{F}$*, falls eine quadratische Matrix* C_{task} *existiert, so dass*

$$WORK\{task(F)\} \leq C_{task} WORK\{eval(F)\} \quad \forall F \in \mathcal{F}$$

Wir nennen eine solche Matrix Komplexitätsschranke der Aufgabe.

Offensichtlich ist die Funktionsauswertung selbst per definitionem von linear beschränkter Komplexität ($C_{eval} = Id_{\dim(WORK)}$). Neben der linear beschränkten Komplexität kann es aber auch durchaus Aufgaben geben, die *polynomial beschränkte Komplexität* haben. So ist beispielsweise die Auswertung einer vollbesetzten JACOBI-Matrix keine Aufgabe mit linear beschränkter Komplexität, selbst wenn Φ nur den polynomialen Kern umfasst. Man kann nämlich zeigen, dass die Auswertung von p Zeilen *oder* q Spalten der JACOBI-Matrix linear beschränkte Komplexität besitzt. Daher ist die Komplexität der Auswertung einer JACOBI-Matrix proportional zur Dimension des Definitionsbereiches n oder zur Dimension des Wertebereiches m. Unter der vernünftigen Annahme, dass die Komponenten von $WORK\{eval(F)\}$ für alle $F \in \mathcal{F}$ durch ein Vielfaches von n oder m beschränkt sind, schließen wir daher

$$WORK\{eval(F')\} = \mathcal{O}\left(WORK\{eval(F)\}^2\right)$$

Da bereits der polynomiale Kern Φ_{min} nicht endlich ist, machen wir eine weitere
Annahme EB (lineare Beschränktheit der elementaren Aufgaben): *Es gibt eine Matrix* $C_{task}(\Phi)$*, so dass*

$$WORK\{task'(\varphi)\} \leq C_{task}(\Phi) \cdot WORK\{eval(\varphi)\}$$

für alle $\varphi \in \Phi$ *gilt.*

Wir fordert also, dass jeder auf eine Elementarfunktion bezogene Teil einer Aufgabe linear beschränkte Komplexität über Φ besitzt. Dies ist insbesondere dann erfüllt, wenn die Menge der Vektorpaare

$$\left\{\left(WORK\{task'(\varphi)\}, WORK\{eval(\varphi)\}\right) \;\middle|\; \varphi \in \Phi\right\}$$

endlich ist. Da alle Initialisierungen mit verschiedenen c die gleiche Komplexität haben, gilt die Annahme EB auch für Φ_{min}. Damit können wir für additive Aufgaben folgenden Satz beweisen:

Satz 2.6 (lineare Beschränktheit von additiven Aufgaben)
Die Annahmen TA und EB implizieren linear beschränkte Komplexität über $\mathcal{F}$ *im Sinne der Definition 2.5 mit* $C_{task} = C_{task}(\Phi)$*. Außerdem gilt für jedes Laufzeit-Funktional (2.4) die Abschätzung*

$$TIME\{task(F)\} \leq \omega_{task} TIME\{eval(F)\} \quad \forall F \in \mathcal{F}$$

$$mit \quad \omega_{task} := \sup_{\varphi \in \Phi} \frac{w^T WORK\{task'(\varphi)\}}{w^T WORK\{eval(\varphi)\}} \leq \left\|DC_{task}D^{-1}\right\|_1 < \infty, \quad D := diag(w)$$

Beweis: Aus den beiden Annahmen folgt direkt

$$\begin{aligned} WORK\{task(F)\} &\leq \sum_{i=1}^{l} WORK\{task'(\varphi_i)\} \\ &\leq \sum_{i=1}^{l} C_{task}(\Phi) \cdot WORK\{eval(\varphi_i)\} \\ &= C_{task}(\Phi) \cdot WORK\{eval(F)\} \end{aligned}$$

Damit ist der erste Teil der Aussage bewiesen. Wie bei der Herleitung von Gleichung (2.5) folgern wir

$$\begin{aligned} \frac{TIME\{task(F)\}}{TIME\{eval(F)\}} &\leq \max_{1 \leq i \leq l} \frac{TIME\{task'(\varphi_i)\}}{TIME\{eval(\varphi_i)\}} \\ &\leq \sup_{\varphi \in \Phi} \frac{w^T WORK\{task'(\varphi)\}}{w^T WORK\{eval(\varphi)\}} \\ &\leq \sup_{\varphi \in \Phi} \frac{w^T C_{task} WORK\{eval(\varphi)\}}{w^T WORK\{eval(\varphi)\}} \\ &\leq \sup_{0 \neq z \in \mathbb{R}^{\dim(w)}} \frac{w^T C_{task} z}{w^T z} = \sup_{0 \neq z \in \mathbb{R}^{\dim(w)}} \frac{\|DC_{task}z\|_1}{\|Dz\|_1} = \left\|DC_{task}D^{-1}\right\|_1 < \infty \end{aligned}$$

□

Wir formulieren die Kernaussage von Satz 2.6 als

Regel 3: *Additive Aufgaben, die über der Menge der Elementarfunktionen linear beschränkte Komplexität besitzen, haben auch über der Menge der aus diesen Elementarfunktionen zusammengesetzten Funktionen linear beschränkte Komplexität.*

Für einen beliebigen Komplexitätsvektor mit entsprechendem Laufzeit-Funktional ist also das Verhältnis von $TIME\{task(F)\}$ zu $TIME\{eval(F)\}$ durch ω_{task} beschränkt. Für unseren vier-elementigen Komplexitätsvektor ergibt sich

$$\omega_{task} = \max_{1 \le i \le 4} \frac{w^T W_{task} e_i}{w^T W_{eval} e_i}$$

Dabei ist $e_i \in \mathbb{R}^4$ der i-te Einheitsvektor.

3 Schlussbemerkungen

Der Inhalt dieses Handouts orientiert sich in großen Teilen an den Kapiteln 1 und 2 von [1]. Daher wurde darauf verzichtet, sämtliche Stellen, die sinngemäß übernommen wurden, besonders zu kennzeichnen. Insbesondere wurden sämtliche Sätze, Annahmen und Regeln aufgenommen, damit nachfolgende Vortragende besser Bezug auf sie nehmen können.

Das Handout sowie die Präsentationsfolien sind auch unter

`http://www.mathi.uni-heidelberg.de/~ferreau/ad/`

zum Download verfügbar.

Literatur

[1] Griewank, A.: *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* SIAM, Philadelphia (2000).

[2] Griewank, A.: `http://www.math.tu-dresden.de/wir/staff/griewank`.

[3] Heun, V.: *Grundlegende Algorithmen.* Vieweg, Wiesbaden (2003).

[4] Königsberger, K.: *Analysis 2.* Springer, Heidelberg (2000).

[5] Nocedal, J.; Wright, S. J.: *Numerical Optimization.* Springer, New York (1999).