Software zur Berechnung von Jacobi- und Hessematrizen aus C und Fortran Code

D. Shiriaev* A. Griewank* J. Utke*

Zusammenfassung

Die hier beschriebenen Pakete ADOL–C and ADOL–F[1] erleichtern die Berechnung von Ableitungen ersten und höheren Grades von Vektorfunktionen, die durch Computerprogramme in C/C++ oder Fortran 77/90 definiert sind.

Die numerischen Werte von abgeleiteten Vektoren erhält man frei von Rundungsfehlern für ein geringes Vielfaches der Laufzeit und des Speicherbedarfs zur Berechnung der gegebenen Funktion. Jacobi–Matrizen kann man zeilen- oder spaltenweise erhalten. Für Lösungskurven, die durch gewöhnliche Differentialgleichungen gegeben sind, liegen spezielle Routinen vor, die Taylor–Koeffizientenvektoren und ihre Jacobi–Matrix bezüglich des aktuellen Vektors berechnen. Die Ableitungsberechnungen beziehen eine möglicherweise große Datenmenge ein, deren Umfang jedoch a priori berechenbar ist. Auf diese Daten wird strikt sequentiell zugegriffen und man kann sie deshalb automatisch auf externe Massenspeicher auslagern.

1 Einführung

Automatische Differentiation [3, 6] ist, wie symbolische Differentation, keine Näherungsmethode und erlaubt schnelle und exakte Berechnungen von Ableitungen beliebigen Grades. Die Genauigkeit der Ableitungen liegt dabei in derselben Größenordnung wie die Genauigkeit der durch das Computerprogramm berechneten Vektorfunktion. Besonders in Anwendungen die zweite und höhere Ableitungen benötigen ist die Vermeidung von Rundungsfehlern von entscheidender Bedeutung.

---

*Institut für Wissenschaftliches Rechnen, TU Dresden, D–01062 Dresden

[1]Neue und detaillierte Informationen zu der Software, Source Code und Manuals sind unter `http://www.math.tu-dresden.de/~adol-c` zu finden.

Im Vergleich zur Automatischen Differentiation werden bei Differenzenquotienten die Rundungsfehler im Zähler durch die kleinen Nenner verstärkt, was oft zu schlechten Näherungen für zweite Ableitungen führt und sie praktisch wertlos für dritte und höhere Ableitungen macht. Mittels Automatischem Differenzieren erhält man numerische Werte für höhere Ableitungen einer gegebenen Funktion, ohne eine explizite Formel dafür zu generieren, so daß der für die symbolische Differentiation typische "expression swell" und die Rundungsfehler der Differenzenquotienten vermieden werden.

Es gibt zwei grundlegende Modi der Automatischen Differentiation, die üblicherweise als Vorwärts- und Rückwärtsmodus bezeichnet werden. Der Hauptunterschied zwischen ihnen ist, daß die Komplexität des Vorwärtsmodus (der einfacher zu implementieren ist) im wesentlichen von der Anzahl n der unabhängigen Variablen abhängig ist, während die Komlexität des Rückwärtsmodus durch die Anzahl m der abhängigen Variablen bestimmt wird. Demzufolge liefert der Vorwärtsmodus den Gradienten einer gegebenen Funktion mit einem Zeitaufwand und Speicherbedarf der proportional zur Anzahl der unabhängigen Variablen ist. Das mag akzeptabel für kleine n sein, wird aber für große n problematisch.

Der Rückwärtsmodus hingegen liefert den Gradienten einer skalarwertigen Funktion ($m = 1$) im Prinzip mit höchstens dem Fünffachen der Anzahl der Operationen, die zur Berechnung der Funktion selbst ausgeführt werden. Damit ist der Aufwand für die Gradientenberechnung völlig unabhängig von der Anzahl der unabhängigen Variablen. Demzufolge kann man ebenfalls im Rückwärtsmodus den Rechenaufwand zur Bestimnmung der Jacobimatrix einer Vektorfunktion ($m \geq 1$) mit dem $5m$-fachen des Rechenaufwandes für die Vektorfunktion selbst abschätzen. Diese theoretische Abschätzung muß jedoch insofern korrigiert werden, als sie den Aufwand zur Vorbereitung des Rückwärtslaufes unberücksichtigt läßt. Diese Vorbereitung wird wird im Folgenden noch erwähnt werden. Das bedeutet für allgemeine Funktionen $F : \mathbb{R}^n \mapsto \mathbb{R}^m$, daß der Rückwärtsmodus nur für $n >> m$ in Bezug auf den Rechenaufwand vorteilhaft gegenüber dem Vorwärtmodus ist. Daher es ist für $n \approx m$ immer besser, dichtbesetzte Jacobimatrizen spaltenweise durch das Vorwärtsverfahren mit ungefähr dem n-fachen des Aufwandes für die Berechnung der Funktion zu bestimmen. In Anwendungsfällen werden aber häufig keine vollen Ableitungstensoren benötigt sondern nur gewisse Linearkombinationen oder sie sind dünnbesetzt. Der optimale Einsatz des Vorwärts- und Rückwärtmodus ist dann nicht mehr über eine einfache Regel definiert sondern muß für das konkrete Problem entschieden werden.

Die Abkürzung ADOL–C/F steht für **A**utomatic **D**ifferentiation by **O**ver-**L**oading in **C**/C++ or **F**ORTRAN (Automatische Differentiation durch

Überladen in C/C++ bzw. Fortran). ADOL–F [7], [8] ist ein Fortran 90 Interface zur ADOL–C [4] Bibliothek. Da jedoch die Funktionalität von Fortran 90 gegenüber C++ eingeschränkt ist, hat auch ADOL-F eine gegenüber ADOL-C ähnliche aber leicht eingeschränkte Funktionalität. ADOL–C/F ermöglicht die simultane Berechnung von Richtungsableitungen beliebig hohen Grades und der Gradienten dieser Taylor–Koeffizienten bezüglich aller unabhängigen Variablen. Der Berechnungsaufwand jedes solchen Skalar–Vektor Paares wächst wie das Quadrat des Grades der Ableitung relativ zum Aufwand der Berechnung der zugrundeliegenden Funktion, ist aber vollkommen unabhängig von der Anzahl der abhängigen Variablen der zugrundeliegenden Funktion.

2 Vorbereitung eines Codes für die Differentiation

ADOL–C/F ist so aufgebaut, daß der Benutzer nur minimale Veränderungen an seinem undifferenzierten Code vorzunehmen braucht. Die Hauptveränderungen betreffen Variablendeklarationen und Ein/Ausgabe–Operationen. Ein zentraler Begriff der Automatischen Differentiation ist das Konzept einer *aktiven Variablen.* Alle Variablen, die während der Programmausführung irgendwann als differenzierbar anzusehen sind, müssen als Variable aktiven Typs deklariert werden. ADOL–C/F benutzt einen Typ, dessen numerischer Wert doppelte Präzision hat. Durch diese Umdeklaration wird ein ursprüngliches C Programm zu einem C++ Programm[2], der Benutzer kommt jedoch nicht mit den eigentlichen C++ Bestandteilen von ADOL-C[3] in Berühung. Üblicherweise wird man die unabhängigen Variablen und alle von diesen direkt oder indirekt abhängenden Variablen als *aktiv* deklarieren. Andere Variablen, die nicht von den Unabhängigen abhängen, aber z.B. als Parameter eingehen, können ihren ursprünglichen *passiven* `float`, `double` oder `integer` Typ behalten.

Die eigentlichen Berechnungsanweisungen des Programmes müssen für die Automatische Differentiation nicht verändert werden. Legitimer C oder Fortran Code in aktiven Abschnitten kann vollständig unverändert bleiben, vorausgesetzt, die direkte Ausgabe aktiver Variablen wird vermieden. Alle Berechnungen, die aktive Variablen beinhalten und zwischen den Aufrufen

`trace_on(tag,keep)` und `trace_off(file)`

auftreten, werden in einer sequentielle Datenmenge, dem *Band*, protokolliert

[2] bzw. ein Fortran 77 zu einem Fortran 90 Programm

[3] bzw. den eigentlichen Fortran 90 Bestandteilen von ADOL-F

Das Band wird dann von den Routinen die den Vorwärts- bzw. Rückwärtsmodus ausführen gelesen. Es enthält eine vollständige Mitschrift aller zur Laufzeit ausgeführten Operationen, in denen `adouble`s involviert waren. D.h. das Band enthält pro Operation einen Code zur Identifikation der Operation und die internen Addressen der Argumente und des Ergebnisses bzw. die Werte konstanter Argumente. Wir werden die Folge von Anweisungen, die zwischen einem Aufruf von `trace_on` und dem folgenden Aufruf von `trace_off` auftreten, als *aktiven Abschnitt* des Programms bezeichnen. Die Mitschrift (das Band) eines aktiven Abschnitts kann wiederholt benutzt werden, und man kann verschiedene Mitschriften auf unterschiedlichen Bändern speichern, die durch die Bandnummer `tag` auseinander gehalten werden. Die Mitschrift enthält keine *vollständige* Repräsentation des Computerprogramms, insoweit der Ablauf des Programms (im Sinne eines Flußdiagramms) von den Werten der aktiven Variablen bestimmt wird. Sofern der Ablauf des Programms im aktiven Abschnitt von Werten aktiver Variablen abhängt, d.h. wenn aktive Variablen in Vergleichsoperationen auftreten, kann später in den Routinen für den Vorwärts-/Rückwärtsmodus die Gültigkeit des Bandes automatisch überprüft werden. Das bedeutet für den Fall der Ausführung des Vorwärts-/Rückwärtsmodus für Argumente, die nicht identisch sind mit den Argumenten zur Zeit der Erzeugung der Mitschrift, wird bei der Feststellung der Ungültigkeit des Bandes der Lauf abgebrochen und der Benutzer mit einem entsprechenden Returncode über die Notwendigkeit informiert, eine neue, gültige Mitschrift anzufertigen.

Nach der Ausführung eines großen aktiven Abschnittes werden Dateien die das Band enthalten in das aktuelle Arbeitsverzeichnis geschrieben. Durch den Aufruf von `trace_on` mit verschiedenen Bandnummern (`tag`), kann man verschiedene Bänder für unterschiedliche Funktionsberechnungen erstellen und Funktions– und Ableitungsberechnungen auf einem oder mehreren dieser Bänder ausführen. Das Band eines relativ kleinen aktiven Abschnittes kann in einem dynamischen Feld gehalten werden, falls nicht der Benutzer das Ausschreiben des Bandes auf Datei durch Aufruf von `trace_off` mit dem Parameter `file`, Wert `true`, erzwingt.

Aktive Abschnitte können rekursive Aufrufe zu benutzereigenen Funktionen enthalten. Selbstverständlich müssen die formalen und aktuellen Parameter typkonform sein. Insbesondere müssen benutzerdefinierte Funktionen mit korrekt zu "aktiv" umdeklarierten Variablen compiliert werden.

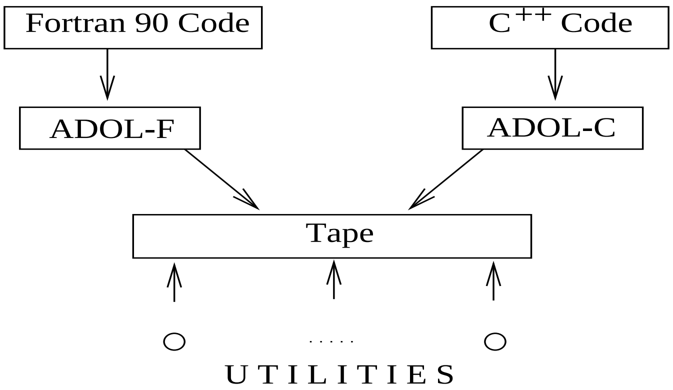
Für die Berechnung der Ableitungen müssen die n unabhängigen und die m abhängigen Variablen aus der Menge der aktiven Variablen ausgezeichnet werden. Diese Auszeichnung geschieht in ADOL-C mit Hilfe spezieller Zuweisungsoperatoren und in ADOL-F durch Aufruf der Funktionen `independent()` und `dependent()`.

Der folgende Abschnitt diskutiert einige Implementierungsfragen, deren Verstandnis für die reine Nutzung von ADOL-C/F nicht nötig ist.

3 Implementations–Gesichtspunkte

Im Unterschied zu einigen anderen Paketen zur Automatischen Differentiation z.B. [1, 5] wird hier das Prinzip überladener Operatoren anstelle des Prinzips der Codetransformation durch Precompiler verwendet. Dies bedingt zwar den weitgehenden Verlust compilerinterner Optimierung, ermöglicht aber die relativ überschaubare Implementierung der Automatischen Differentiation für Ableitungen beliebiger Ordnung mit einem vernünftigen Aufwand an Speicherplatz und Rechenzeit.

ADOL–C besteht softwaretechnisch aus zwei unterschiedlichen Teilen. Der erste benutzt die Überladungsfähigkeiten von C++, um die durch das C++ Programm (bzw. zum C++ Programm gewandelte C Programm) gegebene Funktion auf ein Band mitzuschreiben. Der zweite Teil enthält verschiedene Routinen und Treiber in C, welche das Band lesen und auf dessen Grundlage (Richtungs-)Ableitungen für verschiedene Argumentwerte berechnen. ADOL-F ersetzt im wesentlichen den ersten Teil durch ein entsprechendes Fortran 90 front end und ruft den zweiten Teil mit passenden Treibern in Fortran auf.



ADOL–C/F überladen alle arithmetischen Operationen wie auch Vergleiche, Zuweisungsoperatoren und die mathematischen Standardfunktionen.

Folgende Fortran 90 Spezifika sind von besonderer Bedeutung in ADOL-F: Nutzerdefinierte Operatoren, Funktionen und Operatoren mit beliebigem Ergebnistyp, Überladung von Prozeduren, Funktionen und Operatoren und das Modulkonzept. Für unsere Zwecke ist der Hauptnachteil von Fortran 90 im Vergleich zu C++ das Fehlen von Konstruktoren und Destruktoren für nutzerdefinierte Typen. Dieser Mangel führt dazu, daß im Unterschied zu ADOL–C, lokale aktive Variablen, beim Verlassen des Blocks in dem sie definiert sind, ihren 'Nummer'[4] nicht freigeben können. Dieser Effekt kann zu einer unverhältnismäßig großen Menge dieser 'Nummern' führen, was wahrscheinlich die Anzahl der page faults sowie den Speicherbedarf erhöht. Dieser Nachteil wiegt nicht besonders schwer, falls der Fortran 90 Code ohne rekursive Unterprogramme oder Funktionen geschrieben ist, so daß die Gesamtzahl der allokierten bezeichneten[5] Variablen begrenzt ist. Um mit der eventuell großen Anzahl compilergenerierter temporärer[6] Variablen umgehen zu können, wenden wir ein geeigneteres Prinzip zur Verwaltung der aktiven temporären Variablen als in ADOL–C an. Dazu benutzt ADOL–F einen separaten Speicherplatz für die Werte von temporären Variablen (Ergebnisse von Zwischenoperationen) und gibt diese 'Nummern' direkt in der Operation wieder frei. Zur Durchführung des Rückwärtsmodus müssen die Werte aller der Variablen im Rückwärtslauf bekannt sein[7], die als Argument in einer nichtlinearen Elementaroperation auftreten. Jede lineare Operation mit einem temporären Operanden speichert nur einen Operationscode auf dem Band der anzeigt, daß der Wert der temporären Variable nicht gebraucht und deshalb auch nicht gespeichert wird.

Jede nichtlineare Operation prüft die Operanden und speichert entweder den Wert eines Operanden, wenn sein interner Status 'temporär' ist zusammen mit einem Operationscode, der diesen Status signalisiert oder setzt nur den Status dieses Operanden auf 'nichtlinear'. Kommt man schließlich an die Zuweisungsoperation wird der Status der linken Seite auf 'nichtlinear' getestet und ggf. der Wert dieses Operanden abgespeichert.

Entsprechend gibt es zwei verschiedene Operationscodes für jeden Zuweisungsoperator, die anzeigen, ob der überschriebene Wert der linken Seite gespeichert wurde oder nicht. Dadurch wird auch eine bezeichnete aktive Variable nur dann abgespeichert, wenn sie Argument einer nichtlinearen Operation war.

[4]eine interne Adresse, die zur Identifikation von Zwischenwerten für den Rückwärtslauf benutzt wird

[5]d.h. vom Nutzer explizit deklarierten

[6]im Unterschied zu bezeichneten

[7]d.h. sie müssen also in einem vorbereitenden Vorwärtslauf separat in einer Datei abgespeichert werden

4 Aktive Vektor- und Matrixklassen

Um den übermäßigen Aufwand bei der Verarbeitung individueller skalarer Variablen und ihrer Operationen zu reduzieren, enthält ADOL-C Klassen für aktive Vektoren und Matrizen. Entsprechende aktive Datenstrukturen werden demnächst auch in ADOL-F eingeführt. Vektorelemente der Form `a[i]` können an Stelle jeder skalaren Variablen aktiven Typs verwendet werden.

Die Verwendung von aktiven Vektoroperationen kann die Länge des Bandes und die Laufzeit des Codes entscheidend reduzieren. Dieser Vorteil tritt speziell in den Operationen der linearen Algebra auf, wo in explizit komponentenweisen Berechnungen viele unnötige Zwischengrößen erzeugt werden. Die notwendigen linearen Operationen und Zuweisungsoperatoren sind zwischen aktiven Vektoren definiert. Der aktive Matrixtyp wird nur verwendet, um die automatische und zusammenhängende Allokation (und Deallokation) von Feldern, deren Elemente aktive Vektoren sind, zu erleichtern. Fortran ähnlicher Zugriff auf Spalten ist nicht möglich. Auch wenn keine Vektoroperationen durchgeführt werden sollen, ist die Deklaration mit den aktiven Vektor- und Matrixtypen der expliziten Erzeugung aktiver Felder mit `new` vorzuziehen.

5 Mathematische Beschreibung der berechneten Ableitungen

Berechnet werden Ableitungen beliebig hoher Ordnung von Vektorfunktion $F : \mathbb{R}^n \mapsto \mathbb{R}^m$, die durch den aktiven Abschnitt definiert ist. Innerhalb von ADOL-C/F werden Ableitungen immer in der Form von Taylorkoeffizienten berechnet. Durch geeignete Treiberroutinen können diese dann zu den gewohnten Vektoren, Matrizen und Tensoren zusammengestellt werden. Sei

$$x(t) \quad \equiv \quad \sum_{j=0}^{d} x_j t^j \quad : \quad \mathbb{R} \mapsto \mathbb{R}^n \tag{1}$$

ein Vektorpolynom der skalaren Variable $t \in \mathbb{R}$. Mit anderen Worten, $x(t)$ beschreibt eine Kurve in $\mathbb{R}^n$, die durch t parametrisiert ist. Die Taylorkoeffizientenvektoren

$$x_j \quad = \quad \frac{1}{j!} \frac{\partial^j}{\partial t^j} x(t) \bigg|_{t=0}$$

sind die skalierten Ableitungen von $x(t)$ am Ursprung $t = 0$. Der Vektor $x_1 \in \mathbb{R}^n$ kann als Tangente am Basispunkt x_0 betrachtet werden, der Vektor

$x_2 \in I\!R^n$ enthält Krümmungsinformation. Vorausgesetzt F ist d-mal stetig differenzierbar, so folgt aus der Kettenregel, daß die Bildkurve

$$y(t) \quad \equiv \quad F(x(t)) \quad : \quad I\!R \mapsto I\!R^m \tag{2}$$

ebenfalls glatt ist und $d+1$ Taylorkoeffizientenvektoren $y_j \in I\!R^m$ an der Stelle $t=0$ hat, so daß

$$y(t) \quad = \quad \sum_{j=0}^{d} y_j t^j + O(t^{d+1}) \quad . \tag{3}$$

Ebenfalls als eine Konsequenz der Kettenregel kann man überprüfen, daß jedes y_j eindeutig bestimmt ist durch die Koeffizientenvektoren x_i mit $i \leq j$. Speziell haben wir

$$y_0 = F(x_0) \quad , \quad y_1 = F'(x_0)\, x_1 \tag{4}$$

und

$$y_2 = F'(x_0)\, x_2 + \tfrac{1}{2} F''(x_0)\, x_1\, x_1 \quad . \tag{5}$$

Durch den letzten Term sind wir schon von der üblichen Matrix-Vektor-Notation abgewichen Im Gegensatz zu einer voll symbolischen Behandlung wachsen im Automatischen Differenzieren Speicherbedarf und Operationszahl zur Berechnung vom y_j nur quadratisch mit j.

Die Funktionen

$$y_j = y_j(x_0, x_1, \ldots, x_j) \in I\!R^m$$

besitzen für hinreichend glattes F partielle Ableitungsmatrizen mit der Invarianzeigenschaft.

$$\frac{\partial y_j}{\partial x_i} = \frac{\partial y_{j-i}}{\partial x_0} = A_{j-i}(x_0, x_1, \ldots, x_{j-i}) \tag{6}$$

die in [2] gezeigtt wurde. Die $m \times n$ Matrizen $A_k, k = 0, \ldots, d$ lassen sich auch als Taylorkoeffizienten der Jacobikurve $F'(x(t))$ interpretieren. Dieser Zusammenhang ist besonders für die Lösung von gewöhnlichen und Algebro–Differentialgleichungen von Interesse.

6 Die Funktion `forward`

Sei ein Band eines aktiven Abschnittes geschrieben und die Koeffizienten x_j durch den Nutzer vorgegeben. Dann können die resultierenden y_j durch geeignete Aufrufe der überladenen Funktion `forward` berechnet werden.

Für $F : I\!R^n \mapsto I\!R^m$ liefert die skalare Version von `forward`

```
forward(tag,m,n,d,keep,X,Y)
```

genau eine abgeschnittene Taylorreihe der $(y_j)_{j\leq d}$ für die gegebenen $(x_j)_{j\leq d}$. Die Zeilen der Matrix $X \in \mathbb{R}^{n\times(d+1)}$ müssen den unabhängigen Variablen in der Reihenfolge ihrer Unabhängigkeitsspezifikation im Programm entsprechen. Die Spalten von $X = \{x_j\}_{j=0..d}$ repräsentieren die Taylorkoeffizientenvektoren wie in Gleichung (1). Die Zeilen der Matrix Y entsprechen den abhängigen Variablen in der Reihenfolge ihrer Abhängigkeitsspezifikation im Programm.

Die Spalten von $Y = \{y_j\}_{j=0..d}$ repräsentieren Taylorkoeffizientenvektoren wie in (3). So enthält die erste Spalte von Y die Funktionswerte $F(x)$ selbst, die nächste Spalte repräsentiert den ersten Taylorkoeffizientenvektor von F, und die letzte Spalte den d-ten Taylorkoeffizientenvektor. Der ganzzahlige Schalter `keep` bestimmt, ob und wieviele Taylorkoeffizienten aller Zwischengrößen durch `forward` in eine gepufferte temporäre Datei in Vorbereitung für einen nachfolgenden Rückwärtslauf geschrieben werden. Der gegebene `tag` Wert wird von `forward` zur Bestimmung des Namens der Datei benutzt, in die das Band geschrieben wurde.

Die Routine `forward` kann zur Berechnung der Vektorfunktion F auch an einer anderen Stelle x als der, für die das Band generiert wurde, benutzt werden. Falls Programmverzweigungen im aktiven Abschnitt durch Vergleichsoperationen mit aktiven Variablen gesteuert wirden, so wird die Gültigkeit des Bandes automatisch überprüft. Falls es nicht mehr gültig ist, muß der aktive Abschnitt erneut ausgeführt werden und für das aktuelle Argument ein neues Band geschrieben werden. Anderenfalls können die numerischen Werte unkorrekt sein. Zur Zeit werden die folgenden Rückgabewerte benutzt.

+3	Die Funktion ist lokal analytisch.
+2	Die Funktion ist lokal analytisch, aber Werte von `max, min` oder `abs` liegen auf einem anderen Zweig als bei der Mitschrift.
+1	mindestens eine der Funktionen `min, max` oder `abs` wird an einer Gleichheits- oder Nullstelle berechnet. So ist zwar F Lipschitz-stetig, aber nicht notwendig differenzierbar.
0	Ein arithmetischer Vergleich zwischen aktiven Variablen ergibt Gleichheit. Deshalb kann F in einer Umgebung unstetig sein.
-1	Ein Vergleich zwischen aktiven Variablen führte zu anderen Ergebnissen als an der Stelle, für die das Band generiert wurde. Die resultierenden Ableitungswerte sind wahrscheinlich inkorrect.

Die Vektorversion von `forward` liefert eine Familie von $p \geq 1$ abgeschnit-

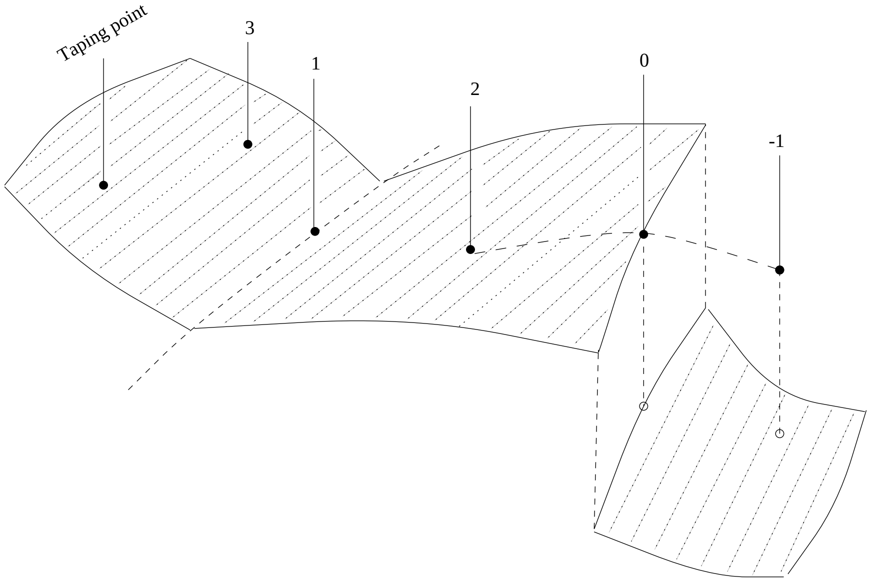


Abbildung 1: Rückgabewerte bezogen auf das Argument zur Zeit der Mitschrift des Bandes (taping point)

tenen Taylorreihen. Diese Funktion hat einen geringeren Aufwand als der wiederholte Aufruf der sklaren Vorwärtsroutine, da sie den Aufwand zur Interpretation des Bandes reduziert.

```
forward(tag,m,n,d,p,x,X,y,Y)
```

Hier enthalten X und Y die Taylorkoeffizienten der ersten und höheren Ordnung und x und y die Taylorkoeffizienten der Ordnung 0, d.h. die Werte der unabhängigen und abhängigen Variablen. Im Unterschied zur skalaren Version kann diese Routine *nicht* zur Vorbereitung eines nachfolgenden Rückwärtslaufs verwendet werden.

Da die Berechnung von Jacobi–Matrizen wahrscheinlich die wichtigste Aufgabe der Automatischen Differentiation ist, wird eine spezialisierte Vektorversion von `forward` für den Fall $d = 1$ zur Verfügung gestellt.

```
forward(tag,m,n,p,x,X,y,Y)
```

Wenn diese Routine mit $p = n$ und X als Einheitsmatrix aufgerufen wird, ist die resultierende Matrix Y einfach die Jacobi–Matrix $F'(x)$. Ganz allgemein erhält man die $m \times p$ Matrix $Y = F'(x)\,X$ für eine belibig gewählte Initialisierung der $n \times p$ Matrix X.

7 Die Funktion `reverse`

Für einen gegebenen Gewichtsvektor $u \in I\!R^m$, berechnet die Funktion `reverse` eine Menge von Zeilenvektoren

$$z_j \equiv u^T \frac{\partial y_j}{\partial x_0} = u^T A_j \in I\!R^n \tag{7}$$

für $j = 0, 1, \ldots, d$.

Nach der Ausführung eines aktiven Abschnittes mit `keep`$=$ *true* oder einem Aufruf von `forward` mit beliebigem `keep`$\leq d{+}1$ kann man die Funktion `reverse` entsprechend mit $d = 0$ oder mit $d =$`keep`-1 und derselben Bandnummer `tag` aufrufen. Falls u ein Vektor ist und Z eine $n \times (d+1)$ Matrix wie in (7), wird `reverse` im *skalaren Modus* durch den folgenden Aufruf ausgeführt:

```
reverse(tag,m,n,d,u,Z)
```

Eine Alternative wird durch die Vektorversion von `reverse` zur Verfügung gestellt, die eine Menge von Matrizen der Form

$$Z_j \equiv U \frac{\partial y_j}{\partial x_0} \in I\!R^{p \times m}, \tag{8}$$

liefert, wobei $U \in I\!R^{p \times m}$ eine *Gewichtsmatrix* repräsentiert. Falls $U = I_m$ mit $p = m$ liefert ein Aufruf von `reverse` die Menge der vollständigen Jacobi-Matrizen $\partial y_j / \partial x_0$. Diese Variante erfordert mehr Speicherplatz, reduziert aber wesentlich den Aufwand für die Bandinterpretation. Die Vektorversion von `reverse` wie folgt aufgerufen.

```
reverse(tag,m,n,d,p,U,Z)
```

Für Anwendungen in der Optimierung und bei der Lösung nichtlinearen Gleichungen kann man anstelle von `forward` und `reverse` bequemer die folgenden Treiber benutzen.

für $F : \mathbb{R}^n \mapsto \mathbb{R}$	
`evaluate(tag,1,n,x, y)`	$y = f(x)$
`gradient(tag,n,x, g)`	$g = (\nabla f)(x)$
`hessian(tag,n,x,H)`	$H = (\nabla^2 f)(x)$
`hess_vec(tag,n,x,t,h)`	$h = [(\nabla^2 f)(x)]\ t$
für $F : \mathbb{R}^n \mapsto \mathbb{R}^m$	
`evaluate(tag,m,n,x,y)`	$y = f(x)$
`jacobian(tag,m,n,x,J)`	$f'(x) = J = \left.\left[\frac{\partial f_i}{\partial x_j}\right]\right\|_x$
`jac_vec(tag,m,n,x,t,j)`	$j = f'(x)\ t$
`vec_jac(tag,m,n,r,x,l,j)`	$j = l^T\ f'(x)$
`lagra_hess_vec(tag,m,n,x,t,l, h)`	$h = l^T\ [(\nabla^2 f)(x)]\ t$

8 Beispiel

Im Folgenden wird die Anwendung von ADOL-F am Beispiel der Berechnung des Gradienten und eines Hessematrix-Vektor-Produkts für die Funktion

$$y \;=\; f(x) \;=\; \prod_{i=0}^{n-1} x_i.$$

erläutert. Ausführlichere Beispiele sind in [4] zu finden.

Der aktive Typ in ADOL–F heißt `a_real`. Der numerische Wert von `a_real` Variablen ist vom Typ `real(P_)`. In Fortran 90 bezeichnet `real(P_)` einen bezüglich der Präzision parametrisierten Typ. Dabei schreibt der Fortran 90 Standard nicht den Wert des Präzisionsparameters `p_` auf einem bestimmten System vor sondern man kann ihn z.B. für `double precision` wie folgt definieren.

```
integer, parameter :: p_ = kind(0.0D0)
```

Hiermit kann man portablen Fortran Code schreiben ohne den eigentlichen Wert für die Spezifikation der Präzision zu kennen. Die Änderung der Präzision kann durch eine einzige Anweisung bewirkt werden. Ändert sich die geforderte Präzision des Ergebnisses nicht, so erfordert die Portierung des Programms auf ein anderes System keine weiteren Änderungen unabhängig davon, ob sich die systemeigenen Typen unterscheiden oder nicht.

```
PROGRAM  Product
 USE ADOLF_I    ! Definiert u.a. integer, parameter ::  P_ = kind(0.0D0)
 USE ADOLF
 INTEGER, PARAMETER  ::  n = 5, tag = 1
 INTEGER :: i,j, status = 0
 REAL(P_) :: grad_error= 0.0_P_, errh= 0.0_P_
```

```
 REAL(P_) :: xp (N),  grad (N),  hess (N,N), yp = 0.0_P_
 TYPE(a_real), Dimension (N) :: x
 TYPE(a_real) :: y

 DO I=1, N
   xp(I) = I
 END DO
 CALL trace_on(tag, .TRUE.) ! Begin of the active section, KEEP == .TRUE.
 y = 1.0_P_
 DO I=1, N
   CALL independent (x(I), xp(I)) ! Specification of independents
   y = y * x(I)
 END DO
 CALL  dependent ( yp, y) ! Specification of dependents
 CALL trace_off(.TRUE.) ! End of the active section
 status = evaluate(tag,1,N, xp, grad) ! Evaluate the function
 status = gradient (tag,N, xp, grad) ! Compute the gradient
 status = hessian(tag, N, xp, hess) ! Compute the hessian
 DO I=1, N ! Compute the error in the gradient
   grad_error= grad_error + abs(grad(i)-yp/xp(i))
 END DO
 DO I=1, N ! Compute the error of the Hessian
   DO J=1, N
IF (i > j)   errh = errh + abs( hess(i,j)-grad(i)/xp(j))
   END DO
 END DO
 WRITE (*,*) 'Gradient  Error=', grad_error
 WRITE (*,*) 'Consistency check',  errh
END PROGRAM Product
```

Literatur

[1] C. H. Bischof, A. Carle, G. F. Corliss, A. Griewank, and P. Hovland. *ADIFOR: Generating derivative codes from Fortran programs.* Scientific Programming, 1 (1992), pp. 1–29.

[2] Bruce Christianson, *Reverse accumulation and accurate rounding error estimates for taylor series*, Optimization Methods and Software, 1 (1992), pp. 81–94.

[3] A. Griewank and G. F. Corliss, (eds.), *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, SIAM, Philadelphia, Penn., 1991.

[4] A. Griewank, A., D. Juedes, and J. Utke, *ADOL–C: A Package for the Automatic Differentiation of Algorithms Written in C/C++*, ACM TOMS, vol. 22(2), 1996, pp. 131–167.

[5] K. Kubota, *PADRE2, a Fortran precompiler yielding error estimates and second derivatives*, in [3], 1991, pp. 251–262.

[6] D. Shiriaev, *Fast automatic differentiation for vector processors and reduction of the spatial complexity in a source translation environment*, Dissertation, Mathematik, Universität Karlsruhe, 1993. http://www.math.tu-dresden.de/wir/staff/dima/research.html

[7] D. Shiriaev, A. Griewank, and J. Utke, *A User Guide to ADOL–F: Automatic Differentiation of Fortran Codes*, Technical Report, Institute of Scientific Computing, TU Dresden, 25 pp., 1995.

[8] D. Shiriaev and A. Griewank *ADOL–F: Automatic Differentiation of Fortran Codes*, in Computational Differentiation : Techniques, Applications, and Tools Martin Berz, Christian Bischof, George Corliss, Andreas Griewank (eds.), SIAM, Philadelphia, Penn., 1996.