Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE

Jan Hartman

Realizace metod pro automatické derivování

Katedra numerické matematiky

Vedoucí diplomové práce: Ing. Ladislav Lukšan, DrSc.

Studijní program: Matematika
Studijní obor: Výpočtová matematika
Studijní plán: Výpočtová matematika – algoritmy

Děkuji vedoucímu své diplomové práce Ing. Ladislavu Lukšanovi, DrSc.
za věnovaný čas, poskytnutou literaturu, nesčetné nápady a cenné připomínky,
které vedly k jejímu zlepšení.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně
s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 20. srpna 2001 Jan Hartman

Obsah

1 Úvod **5**
- 1.1 Co je to automatické derivování . . . 5
- 1.2 Malý motivační příklad . . . 6

2 Základní předpoklady **11**
- 2.1 Soupis operací . . . 11
- 2.2 Základní funkce a jejich vlastnosti . . . 13
- 2.3 Alokační funkce . . . 15
- 2.4 Model časové složitosti . . . 16

3 Výpočet první derivace **21**
- 3.1 Přímý mód automatického derivování . . . 21
  - 3.1.1 Odvození a vlastnosti . . . 21
  - 3.1.2 Složitost odvozeného programu . . . 25
- 3.2 Zpětný mód automatického derivování . . . 27
  - 3.2.1 Odvození a vlastnosti . . . 28
  - 3.2.2 Složitost odvozeného programu . . . 35
  - 3.2.3 Iterační přiřazení (nejen) ve zpětném módu . . . 37

4 Výpočet druhé derivace **39**
- 4.1 Odvození a vlastnosti . . . 40
- 4.2 Složitost odvozeného programu . . . 42

5 Způsoby implementace **46**
- 5.1 Přetížení operátorů a funkcí . . . 47
  - 5.1.1 Přímý mód . . . 48
  - 5.1.2 Zpětný mód . . . 52
  - 5.1.3 Druhé a vyšší derivace . . . 57
- 5.2 Použití preprocesoru . . . 59
  - 5.2.1 Přímý mód . . . 59
  - 5.2.2 Zpětný mód . . . 62
  - 5.2.3 Druhé a vyšší derivace . . . 67

6 Další vlastnosti, poznámky a zlepšení **69**
- 6.1 Omezení používání zásobníku . . . 69
- 6.2 Dvakrát provedený zpětný mód na lineární funkce . . . 71

6.3 Využití řídkosti Jacobiho a Hessovy matice při jejich výpočtu . . 72
6.3.1 Dynamické využití řídkosti 72
6.3.2 Statické využití řídkosti 76
6.4 Výpočet vyšších derivací 82
6.5 Diferencovatelnost základních funkcí 82
6.6 Poznámka o základních funkcích 83

7 Implementace automatického derivování v systému UFO 84
7.1 Stručný popis systému UFO 84
7.2 Návrh implementace 88

Několik slov závěrem 90

Literatura 91

Kapitola 1

Úvod

1.1 Co je to automatické derivování

Při řešení nejrůznějších úloh jsme postaveni před problém spočítat derivaci nějaké zadané funkce (případně celý gradient, Jacobiho matici, Hessovu matici, směrové derivace atd.). Taková situace nastává například při řešení optimalizačních úloh. Jedním z možných způsobů výpočtu hodnot derivací *pomocí počítače* se budeme zabývat v této práci.

Předpokládejme, že máme nějaký program, který počítá hodnotu funkce f v zadaném bodě. Připusťme, že takový program může obsahovat cykly, větvení, volání podprogramů apod. Chceme získat co nejpřesnější hodnotu derivace této funkce v tomtéž bodě.

Jednou z možností, jak toho dosáhnout, je počítat *poměrné diference*, například

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \quad \text{nebo} \quad f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

Takový postup není příliš vhodný, protože dostáváme pouze přibližné hodnoty derivace: pokud h je malé, projeví se zaokrouhlovací chyby způsobené počítačovou aritmetikou; pokud h není malé, dostanou se do popředí chyby metody (členy jako $hf''(x)/2$ apod.). Navíc, pro výpočet aproximace celého gradientu funkce n proměnných je potřeba zhruba $(n+1)$-krát tolik času, jako pro výpočet původní funkce.

Jinou metodou výpočtu derivace funkce (programu) je tzv. *symbolická derivace*. Ta aplikuje pravidlo řetězení (chain rule, vzorec pro derivaci složené funkce, viz věta 2.2) zpravidla na symbolickou reprezentaci funkce f v paměti počítače. Jedná se tedy o symbolické úpravy. Zadaný program však velmi často neobsahuje explicitní vyjádření vzorce pro výpočet f. Navíc tento program může být značně dlouhý a složitý.

Třetí možností, o které se zde zmíníme a kterou budeme popisovat ve zbytku této práce, je tzv. *automatické derivování*[1] (automatic differentiation, computati-

[1] Při vypracovávání této práce jsem vycházel výhradně z materiálů v angličtině a nepodařilo se mi najít jakoukoliv zmínku o automatickém derivování v češtině. Proto jsem byl nucen vymyslet české překlady pro používané pojmy. Pro úplnost uvádím i originální anglické termíny.

onal differentiation, algorithmic differentiation). Její aplikací na program, který počítá hodnoty funkce, získáme program, který vyhodnotí *navíc* i požadovanou derivaci této funkce. Tuto *transformaci programu* lze provést *automaticky*. Celý postup je opět založen na pravidle řetězení, které se aplikuje na *numerické hodnoty* získané vyhodnocením elementárních funkcí – narozdíl od symbolické derivace, kde se pravidlo řetězení používá pro symbolické úpravy. Výsledný program počítá hodnotu funkce a zároveň její derivace v jednom konkrétním zadaném bodě a výsledkem je číselná hodnota (nikoliv symbolický nebo explicitní výraz). Postup pro výpočet derivace, který automatickým derivováním dostaneme, má chybu metody rovnou nule. Neboli pokud by počítačová aritmetika byla přesná, dostali bychom odvozeným programem hodnotu derivace přesně. Asi nejzajímavější vlastnost automatického derivování je to, že vyhodnocení celého gradientu funkce n proměnných zabere při použití zpětného módu automatického derivování (reverse mode, adjoint) *nejvýše pětinásobek* času potřebného pro vyhodnocení derivované funkce. Jen pro úplnost dodejme, že pro výpočet 2. a vyšších derivací platí obdobné vlastnosti. O nich se zmíníme později.

1.2 Malý motivační příklad: oscilátor

Dříve než se ponoříme do detailního výkladu, ukážeme si automatické derivování pro lepší představu na velmi jednoduchém příkladu. Máme zadaný program, který počítá hodnotu funkce $f : \mathbb{R}^4 \to \mathbb{R}$ danou předpisem[2]

$$f(A, \omega, t, \varphi_0) = A \sin(\omega t + \varphi_0).$$

Chceme spočítat funkční hodnotu $y = f(1, 1.5, 3, \pi/4)$ a navíc i hodnotu derivace $\frac{\partial f}{\partial t}(1, 1.5, 3, \pi/4)$.

Program vyhodnocuje jednotlivé elementární funkce (např. sčítání, násobení, sinus, ...) v příslušných bodech a z těchto výsledků postupně „skládá“ výslednou funkční hodnotu. Označme v_i výsledek každé takové elementární operace. Pokud ještě přidáme přiřazení vstupních hodnot A, ω, t, φ_0 do proměnných v_{-3}, v_{-2}, v_{-1}, v_0 a výsledné hodnoty do y, dostáváme přesný přepis postupu výpočtu y, který budeme nazývat *soupis operací* (code-list). Můžeme si ho zhruba představit jako záznam numerických operací, které provádí procesor. Přesnou definici soupisu operací uvedeme v kapitole 2.1.

Pokud autor programu výpočet naprogramoval tak, že výraz, který se při výpočtu často vyskytuje, se spočte předem, a pak se jen používá na několika místech, je jeho hodnota označena také v_i a je použita přesně tam, kde má být použita. Pro jednoduchost zatím předpokládejme, že se jednotlivé v_i v paměti počítače nepřepisují, i když už nadále nejsou pro výpočet potřeba. Soupis operací pro naši úlohu včetně jeho numerického vyčíslení je uveden na obrázku 1.1.

[2] Jedná se vlastně o souřadnicový popis pohybu pružinového oscilátoru (například kámen houpající se na pružině při zanedbání určitých jevů), kde A je amplituda výchylky, ω úhlová rychlost, t čas a φ_0 je fázový posun v čase $t = 0$. Pro jednoduchost předpokládejme, že funkce f je definovaná na celém $\mathbb{R}^4$, i když to pro amplitudu a úhlovou rychlost nedává dobrý smysl.

v_{-3}	$= A$	$= 1.0000$	
v_{-2}	$= \omega$	$= 1.5000$	
v_{-1}	$= t$	$= 3.0000$	
v_0	$= \varphi_0$	$= 0.7854$	
v_1	$= v_{-2} * v_{-1}$	$= 1.5000 * 3.0000$	$= 4.5000$
v_2	$= v_1 + v_0$	$= 4.5000 + 0.7854$	$= 5.2854$
v_3	$= \sin v_2$	$= \sin(5.2854)$	$= -0.8403$
v_4	$= v_{-3} * v_3$	$= -0.8403 * 1.0000$	$= -0.8403$
y	$= v_4$	$= -0.8403$	

Obrázek 1.1: Soupis operací pro Malý motivační příklad

Přímý mód automatického derivování (forward mode, tangents)

Jak jsme již uvedli, chceme spočítat derivaci funkce $y = f(A, \omega, t, \varphi_0)$ podle t v bodě $(1, 1.5, 3, \pi/4)$. Uděláme to takovým způsobem, že pro každou proměnnou v_i, $i = -3, \ldots, 4$ určíme numerickou hodnotu její derivace podle t v bodě $(1, 1.5, 3, \pi/4)$. Tím vlastně spočteme i hledanou $\partial y/\partial t$. Ke každé proměnné v_i proto připojíme další proměnnou

$$\dot{v}_i = \frac{\partial v_i}{\partial t}(1, 1.5, 3, \pi/4).$$

Podle pravidla řetězení aplikovaného na každý řádek soupisu operací snadno odvodíme, jakou hodnotu každému $\dot{v}_i$ přiřadit. Zřejmě $\dot{v}_{-3} = 0.0000$, $\dot{v}_{-2} = 0.0000$, $\dot{v}_0 = 0.0000$, ale $\dot{v}_{-1} = 1.0000$. Dále například díky $v_4 = v_{-3} * v_3$ je jistě

$$\begin{aligned} \dot{v}_4 &= \frac{\partial v_4}{\partial v_{-3}} \cdot \dot{v}_{-3} + \frac{\partial v_4}{\partial v_3} \cdot \dot{v}_3 = v_3 \cdot \dot{v}_{-3} + v_{-3} \cdot \dot{v}_3 \\ &= -0.8403 \cdot 0.0000 + 1.0000 \cdot 0.8132 = 0.8132. \end{aligned}$$

Tímto způsobem odvodíme celý nový soupis operací, který tentokrát počítá i hodnotu derivace funkce f podle t v bodě $(1, 1.5, 3, \pi/4)$. Tento soupis operací (odvozený přímým módem automatického derivování) je na obrázku 1.2.

Vidíme, že počet přidaných operací je malý násobek (typicky mezi jednou a třemi) počtu operací pro vyhodnocení y. Pokud by program měl více výstupních proměnných y_i, pak právě naznačeným způsobem dostaneme všechny $\dot{y}_i = \partial y_i/\partial t$. Pokud bychom požadovali parciální derivaci podle jiné proměnné, např. ω, stačí ještě jednou provést tento odvozený soupis operací jen s tím rozdílem, že $\dot{t} = 0.0000$ a $\dot{\omega} = 1.0000$. Je samozřejmě možné předefinovat $\dot{v}_i$ na vektory parciálních derivací podle jednotlivých vstupních proměnných, abychom zmenšili režijní náklady, a tím spočetli třeba celý gradient najednou. Pro tento účel je ovšem mnohem efektivnější zpětný mód automatického derivování. Jak ale z původního programu dostat odvozený program, jehož soupis operací bude vypadat přesně podle našeho návrhu? Jsou možné dva přístupy – použít preprocesor nebo přetížení operátorů a funkcí. Oba způsoby rozebereme v kapitole 5. Přímým módem automatického derivování se budeme detailně zabývat v kapitole 3.1.

$v_{-3} = A$	$= 1.0000$	
$\dot{v}_{-3} = \dot{A}$	$= 0.0000$	
$v_{-2} = \omega$	$= 1.5000$	
$\dot{v}_{-2} = \dot{\omega}$	$= 0.0000$	
$v_{-1} = t$	$= 3.0000$	
$\dot{v}_{-1} = \dot{t}$	$= 1.0000$	
$v_0 = \varphi_0$	$= 0.7854$	
$\dot{v}_0 = \dot{\varphi}_0$	$= 0.0000$	
$v_1 = v_{-2} * v_{-1}$	$= 1.5000 * 3.0000$	$= 4.5000$
$\dot{v}_1 = v_{-1} * \dot{v}_{-2} + v_{-2} * \dot{v}_{-1}$	$= 3.0000 * 0.0000 + 1.5000 * 1.0000$	$= 1.5000$
$v_2 = v_1 + v_0$	$= 4.5000 + 0.7854$	$= 5.2854$
$\dot{v}_2 = \dot{v}_1 + \dot{v}_0$	$= 1.5000 + 0.0000$	$= 1.5000$
$v_3 = \sin v_2$	$= \sin(5.2854)$	$= -0.8403$
$\dot{v}_3 = \cos v_2 * \dot{v}_2$	$= \cos(5.2854) * 1.5000$	$= 0.8132$
$v_4 = v_{-3} * v_3$	$= -0.8403 * 1.0000$	$= -0.8403$
$\dot{v}_4 = v_3 * \dot{v}_{-3} + v_{-3} * \dot{v}_3$	$= -0.8403 * 0.0000 + 1.0000 * 0.8132$	$= 0.8132$
$y = v_4$	$= -0.8403$	
$\dot{y} = \dot{v}_4$	$= 0.8132$	

Obrázek 1.2: Přímým módem automatického derivování odvozený soupis operací pro Malý motivační příklad

Zpětný mód automatického derivování (reverse mode, adjoint, gradients)

V jistém smyslu zajímavější alternativa k přímému módu je mód zpětný. Nyní si nebudeme všímat závislosti proměnných v_i na vstupních parametrech, ale naopak závislosti výstupní proměnné y na každé proměnné v_i. Ke každému v_i tedy zavedeme novou proměnnou $\bar{v}_i = \partial y / \partial v_i$. Toto je názorné, ale nepřesné označení. To, co skutečně myslíme, je $\bar{v}_i = \partial y / \partial \delta_i|_{\delta_i=0}$, kde δ_i se přičítá k pravé straně rovnice definující v_i.

Zpětný mód automatického derivování je opět založen na pravidle řetězení. Tentokrát však při jeho aplikaci budeme postupovat „pozpátku“. Jistě platí $\bar{y} = 1$. Protože například v_2 se na pravé straně libovolného řádku soupisu operací vyskytuje jedině v přiřazení $v_3 = \sin v_2$, musí být $\bar{v}_2 = \bar{v}_3 \cos v_2$. (Pokud by se například v_2 objevilo na pravé straně v přiřazení $v_3 = \sin v_2$ a navíc ještě $v_7 = v_6 + v_2$, pak by muselo být $\bar{v}_2 = \bar{v}_3 \cos v_2 + \bar{v}_7$.) Podrobnosti budou uvedeny v kapitole 3.2, ještě zvídavější čtenáře odkazujeme na [1]. Celý soupis operací odvozený zpětným módem automatického derivování je uveden na obrázku 1.3.

Požadovaná derivace je rovna $\bar{v}_{-1} = 0.8133$. Přímým módem automatického derivování jsme získali hodnotu $\dot{y} = 0.8132$. Malý rozdíl je způsoben tím, že mezivýsledky v celém výpočtu jsme zaokrouhlovali na 4 desetinná místa.

Podobně jako u přímého módu vidíme, že počet přidaných operací je opět malý násobek (typicky mezi jednou a třemi) počtu operací pro výpočet y. V tomto

$v_{-3} = A$		$= 1.0000$	
$v_{-2} = \omega$		$= 1.5000$	
$v_{-1} = t$		$= 3.0000$	
$v_0 = \varphi_0$		$= 0.7854$	
v_1	$= v_{-2} * v_{-1}$	$= 1.5000 * 3.0000$	$= 4.5000$
v_2	$= v_1 + v_0$	$= 4.5000 + 0.7854$	$= 5.2854$
v_3	$= \sin v_2$	$= \sin(5.2854)$	$= -0.8403$
v_4	$= v_{-3} * v_3$	$= -0.8403 * 1.0000$	$= -0.8403$
y	$= v_4$	$= -0.8403$	
$\bar{y}$	$= 1.0000$		
$\bar{v}_4$	$= \bar{y}$	$= 1.0000$	
$\bar{v}_3$	$= \bar{v}_4 * v_{-3}$	$= 1.0000 * 1.0000$	$= 1.0000$
$\bar{v}_{-3}$	$= \bar{v}_4 * v_3$	$= 1.0000 * (-0.8403)$	$= -0.8403$
$\bar{v}_2$	$= \bar{v}_3 * \cos v_2$	$= 1.0000 * \cos(5.2854)$	$= 0.5422$
$\bar{v}_1$	$= \bar{v}_2$	$= 0.5422$	
$\bar{v}_0$	$= \bar{v}_2$	$= 0.5422$	
$\bar{v}_{-1}$	$= \bar{v}_1 * v_{-2}$	$= 0.5422 * 1.5000$	$= 0.8133$
$\bar{v}_{-2}$	$= \bar{v}_1 * v_{-1}$	$= 0.5422 * 3.0000$	$= 1.6266$

Obrázek 1.3: Zpětným módem automatického derivování odvozený soupis operací pro Malý motivační příklad

případě jsme ovšem spočítali celý gradient funkce f. To platí pro každou funkci n proměnných, kterou jsme zpracovali zpětným módem automatického derivování – celý její gradient jsme schopni spočítat za maximálně pětinásobek času potřebného pro jedno vyhodnocení derivované funkce. Kdybychom měli funkci o dvou výstupních hodnotách y_1 a y_2, pak bychom pro získání gradientu y_2 museli vyhodnotit tento odvozený soupis operací ještě jednou s pozměněnými vstupními hodnotami $\bar{y}_1 = 0.0000$ a $\bar{y}_2 = 1.0000$. Pro vyhodnocení několika gradientů najednou je možné předefinovat $\bar{v}_i$ na vektory. Program, který provádí odvozený soupis operací, získáme prekompilací nebo aplikací techniky přetížení operátorů a funkcí. Podrobnosti vyložíme v kapitole 5. Detailní popis zpětného módu automatického derivování uvedeme v kapitole 3.2.

Závěrem podotkněme, že tyto dvě základní techniky automatického derivování mají řadu rozšíření a vylepšení, která využívají například strukturu zadané funkce, řídkost Jacobiho matice aj. Směrové, druhé a vyšší derivace lze získat pomocí podobných „automatických“ technik. O nich se zmíníme v kapitolách 4 a 6.

Stručné historické poznámky

Vývoj a použití automatického derivování přímo souvisí s vlastnostmi a schopnostmi soudobých programovacích jazyků a jejich kompilátorů. Proto v poslední době zažívá automatické derivování rozmach zájmu. Jeho vážný začátek nastal

v 60. letech, kdy byl přímý mód popsán Wengertem (1964). Další zlepšení vypracovali Kedem (1980) a Rall (1981). Základy zpětného módu byly nezávisle na sobě představeny několika matematiky, například Linnainmaaem (1969), Spielpenningem (1980), Kimem (1984) a dalšími.

Literatura

Při vypracovávání této diplomové práce jsem vycházel zejména ze [4], [6], [7] a [10]. Další použité prameny jsou uvedeny přímo v textu práce.

Literatury, která se zabývá automatickým derivováním, je velké množství, zejména z poslední doby. Množství odkazů lze nalézt například v [4].

Kapitola 2

Základní předpoklady

Zabýváme se automatickým derivováním, které transformuje programy pro výpočet funkčních hodnot na programy, které navíc počítají derivace. V této kapitole uvedeme formální popis derivovaných programů, se kterým dále budeme pracovat, a uvalíme na ně jisté požadavky (které však nebudou příliš omezující). Potom nadefinujeme alokační funkci &, která popíše uložení proměnných $\boldsymbol{v}_i$ v paměti počítače. Na závěr této kapitoly představíme model složitosti, pomocí kterého budeme porovnávat efektivnost jednotlivých metod.

2.1 Soupis operací

Program, který chceme transformovat, vyhodnocuje funkci

$$\begin{aligned} \boldsymbol{F} &: D \subset \mathbb{R}^n \longrightarrow \mathbb{R}^m \\ \boldsymbol{F} &: \boldsymbol{x} \longmapsto \boldsymbol{y} = \boldsymbol{F}(\boldsymbol{x}), \end{aligned}$$

kde D je podmnožina[1] v $\mathbb{R}^n$. To, že se hodnoty funkce $\boldsymbol{F}$ počítají programem, je zárukou toho, že $\boldsymbol{F}$ je zadána *explicitně*. Neboli $\boldsymbol{F}$ je složení základních funkcí (například plus, krát, sinus apod.), jejichž hodnoty a *parciální derivace* musíme umět určit. Výše uvedené neznamená, že by vyhodnocení $\boldsymbol{F}$ muselo být popsáno pouze jedinou algebraickou formulí nebo že by program pro vyhodnocení $\boldsymbol{F}$ nemohl obsahovat cykly, podmíněná větvení, volání podprogramů aj.

Předpokládejme, že hodnoty všech parametrů a vstupních proměnných jsou již známy, tj. průběh programu (všechny prováděné operace) je již jednoznačně určen. Postupně se počítají výsledky jednotlivých základních operací a ty jsou používány jako argumenty dalších základních operací.[2] Každý takový výsledek základní funkce označme $\boldsymbol{v}_i$.

Pokud takto rozepíšeme celý algoritmus vyhodnocení $\boldsymbol{F}$ (pro konkrétní parametry a vstupní data) a přidáme přiřazení vstupních proměnných do $\boldsymbol{v}_i$, $i =$

[1] V kapitole 2.2 tento fakt upřesníme.

[2] Tím nemáme na mysli například čítač cyklu, ale skutečnou základní operaci, jejíž argumenty závisí na nezávislých proměnných $\boldsymbol{x}$ a jejíž výsledná hodnota ovlivňuje závislé proměnné $\boldsymbol{y}$. Viz dále.

$1-n, \ldots, 0$ a výstupních proměnných do $\boldsymbol{y}_i$, $i = 1, \ldots, m$, dostáváme *soupis operací* (code-list). Jinak řečeno, soupis operací je rozpis všech operací po rozvinutí cyklů[3] a podprogramů a po nalezení správných cest při větvení programu. Prozatím předpokládejme, že proměnné $\boldsymbol{v}_i$ se v soupisu operací vzájemně nepřepisují. To povolíme a osvětlíme v kapitole 3. Na obrázku 1.1 je uveden soupis operací pro Malý motivační příklad z kapitoly 1.

V obecném případě všechny číselné hodnoty spočtené při celém vyhodnocení funkce $\boldsymbol{F}$ v jednom konkrétním bodě označíme $\boldsymbol{v}_i$ a očíslujeme tímto způsobem.

$$\underbrace{\boldsymbol{v}_{1-n}, \ldots, \boldsymbol{v}_0}_{\boldsymbol{x}}, \boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_{l-m}, \underbrace{\boldsymbol{v}_{l-m+1}, \ldots, \boldsymbol{v}_l}_{\boldsymbol{y}}$$

Každou hodnotu $\boldsymbol{v}_i$, $i = 1, \ldots, l$ získáme vyhodnocením *základní funkce* $\boldsymbol{\varphi}_i$ (elemental function, library function) na příslušné $\boldsymbol{v}_j$ pro $j < i$. Neboli

$$\boldsymbol{v}_i = \boldsymbol{\varphi}_i(\boldsymbol{v}_j)_{j \prec i}.$$

Relace $j \prec i$ znamená, že $\boldsymbol{v}_i$ přímo závisí na $\boldsymbol{v}_j$. V Malém motivačním příkladu například platí $0 \prec 2$, $1 \prec 2$ a $2 \prec 3$, ale neplatí $0 \prec 3$. Zaveďme ještě relaci $\prec^*$. Pokud

$$a \prec b \prec c, \quad \text{budeme psát} \quad a \prec^* c.$$

Například tedy $0 \prec^* 3$, $1 \prec^* 3$, $-2 \prec^* 4$ atd.

Tyto relace (a tedy závislost mezi jednotlivými $\boldsymbol{v}_i$) můžeme názorně zobrazit pomocí orientovaného grafu, jehož vrcholy jsou právě $\boldsymbol{v}_i$, $i = 1-n, \ldots l$ a hrana vede z $\boldsymbol{v}_j$ do $\boldsymbol{v}_i$ právě když $j \prec i$. Tento graf se nazývá *graf výpočtu* (evaluation graph). Nebudeme ho však dále používat, protože veškeré výsledky lze snadno odvodit bez jeho představy. Na obrázku 2.1 je znázorněn graf výpočtu pro Malý motivační příklad.

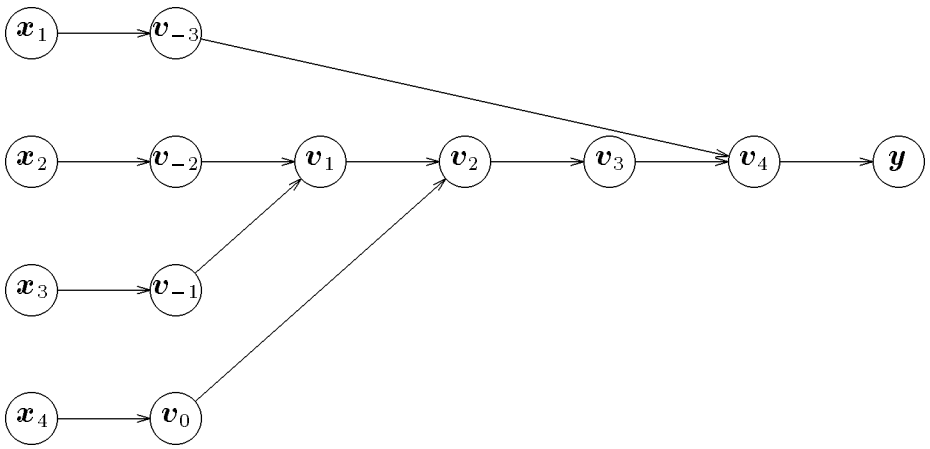
Na obrázku 2.2 je uveden obecný tvar soupisu operací, který budeme nadále používat. Je to postup vyhodnocení funkce $\boldsymbol{F}$ v jednom konkrétním bodě. Tento tvar soupisu operací je pro nás výhodný, protože se na něj pravidlo řetězení snadno a jasně aplikuje.

Základní funkce $\boldsymbol{\varphi}_i$ jsou zamýšleny hlavně jako „elementární“ funkce (např. binární sčítání, binární násobení, trigonometrické funkce atd.), ale v principu to mohou být i funkce složitější. K tomu se bude hodit následující zobecnění.

Jak už napovídá tučný tisk proměnných $\boldsymbol{v}_i$, dovolíme, aby pro $i = 1, \ldots, l-m$ byly $\boldsymbol{v}_i$ vektory délky $m_i \geq 1$.[4] Na přímém ani zpětném módu automatického derivování to nic nezmění, všechno lze odvodit úplně analogicky jako ve skalárním případě. Pro snadnější představu doporučuji, aby $\boldsymbol{v}_i$ byly zpočátku chápány jako skaláry. Toto zobecnění nám umožní pokládat například podprogramy řešící problémy lineární algebry za základní funkce. Také je možné chápat celý soupis operací jako základní funkci, která může být součástí soupisu operací na vyšší

[3] Viz předchozí poznámka pod čarou.

[4] Nezávislé proměnné $\boldsymbol{x}_i$ a závislé proměnné $\boldsymbol{y}_j$ mohou tedy být pouze skaláry.



Obrázek 2.1: Graf výpočtu pro Malý motivační příklad

$\boldsymbol{v}_{i-n}$	$= \boldsymbol{x}_i$	pro	$i = 1, \ldots, n$
$\boldsymbol{v}_i$	$= \boldsymbol{\varphi}_i(\boldsymbol{v}_j)_{j \prec i}$	pro	$i = 1, \ldots, l$
$\boldsymbol{y}_{m-i}$	$= \boldsymbol{v}_{l-i}$	pro	$i = m-1, \ldots, 0$

Obrázek 2.2: Obecný soupis operací

úrovni. Pak lze jednotlivé soupisy operací na různých úrovních zpracovat různými technikami automatického derivování. Označme dále počet složek vektoru z definičního oboru základní funkce $\boldsymbol{\varphi}_i$ jako

$$n_i = \sum_{j \prec i} m_j.$$

Všechny $\boldsymbol{v}_j$, na kterých $\boldsymbol{\varphi}_i$ přímo závisí, poskládejme do argumentu

$$\boldsymbol{u}_i = (\boldsymbol{v}_j)_{j \prec i} \in \mathbb{R}^{n_i}. \tag{2.1}$$

2.2 Základní funkce a jejich vlastnosti

Zopakujme dvě ze základních tvrzení o funkcích více proměnných.

Věta 2.1
Nechť funkce $f : \mathbb{R}^n \to \mathbb{R}$ má spojité všechny parciální derivace v bodě a. Pak existuje totální diferenciál $\mathrm{d}f(a)$ funkce f v bodě a. □

Věta 2.2 (pravidlo řetězení, chain rule)

Nechť funkce $y_1(x_1, \ldots, x_k), \ldots, y_n(x_1, \ldots, x_k)$ mají totální diferenciály $\mathrm{d}y_1(a)$, $\ldots$, $\mathrm{d}y_n(a)$ v bodě $a = (a_1, \ldots, a_k)$. Nechť funkce $z(y_1, \ldots, y_n)$ má totální diferenciál $\mathrm{d}z(b)$ v bodě $b = (y_1(a), \ldots, y_n(a))$. Pak složená funkce

$$z^*(x_1, \ldots, x_k) = z(y_1(x_1, \ldots, x_k), \ldots, y_n(x_1, \ldots, x_k))$$

má totální diferenciál $\mathrm{d}z^*(a)$ v bodě a a platí

$$\mathrm{d}z^*(a) = \frac{\partial z}{\partial y_1}(b) \cdot \mathrm{d}y_1(a) + \ldots + \frac{\partial z}{\partial y_n}(b) \cdot \mathrm{d}y_n(a).$$

Pro parciální derivace platí ($i = 1, \ldots, k$)

$$\frac{\partial z^*}{\partial x_i}(a) = \frac{\partial z}{\partial y_1}(b) \cdot \frac{\partial y_1}{\partial x_i}(a) + \ldots + \frac{\partial z}{\partial y_n}(b) \cdot \frac{\partial y_n}{\partial x_i}(a). \tag{2.2}$$

□

Předpoklad existence totálního diferenciálu na „vnější“ funkci z nelze zeslabit, viz například [2].

Vraťme se zpět k našemu problému. Označme následující vlastnost jako vlastnost 2.1.

Vlastnost 2.1 (Diferencovatelnost základních funkcí)

Všechny elementární funkce $\boldsymbol{\varphi}_i$, $i = 1, \ldots, l$ jsou d-krát spojitě diferencovatelné na svém definičním oboru D_i, kde D_i je otevřená v $\mathbb{R}^n$. Neboli

$$\boldsymbol{\varphi}_i \in C^d(D_i), \quad d \geq 1.$$

□

Její jasný důsledek je následující věta.

Věta 2.3

Nechť platí vlastnost 2.1. Pak

$D = \{\boldsymbol{x} \in \mathbb{R}^n;\ \boldsymbol{y} = \boldsymbol{F}(\boldsymbol{x})$ je dobře definované soupisem operací na obrázku 2.2$\}$

je otevřená množina v $\mathbb{R}^n$ a navíc $\boldsymbol{F} \in C^d(D)$, $d \geq 1$. □

Proto se zdá být logické požadovat následující: Pokud chceme počítat derivace funkce $\boldsymbol{F}$ až do řádu $d \geq 1$ na množině D, ať jsou základní funkce $\boldsymbol{\varphi}_i$ a jejich definiční obory (otevřené) D_i takové, že $\boldsymbol{\varphi}_i \in C^d(D_i)$ a že soupis operací dobře definuje funkci $\boldsymbol{F}$ pro všechna $\boldsymbol{x} \in D$.

Díky výše uvedenému budeme dále předpokládat, že základní funkce $\boldsymbol{\varphi}_i$ vybíráme z nějaké knihovny $\Phi = \{c, +, *, \exp, \sin, \ldots\}$, jejíž členy splňují vlastnost 2.1 pro nějaké $d \geq 1$ (pokud počítáme derivaci stupně d). Dále označme $\mathcal{F}$ všechny

funkce, které mohou ze základních funkcí z množiny Φ vzniknout jejich složením do soupisu operací. Pak tedy $\mathcal{F} \subset C^d$.

Tímto předpokladem jsme z množiny Φ vyloučili např. funkci absolutní hodnota definovanou na celém $\mathbb{R}$. Případem, kdy $\boldsymbol{\varphi}_i$ nemají spojité parciální derivace až do řádu d, se budeme zabývat k kapitole 6.5. Pro úplnost ještě dodejme, že předpoklad spojitosti parciálních derivací zaručuje jejich záměnnost.

Mezi základní funkce budou skoro určitě patřit funkce binární sčítání, binární násobení, unární minus, dělení (případně převrácená hodnota $1/v_j$). Dále Φ může obsahovat sinus, kosinus, exponenciálu atd.

Zdá se být praktické zařadit mezi základní funkce i funkci $\varphi(v) = |v|^c$ definovanou na $\mathbb{R}$ pro $c > 1$, protože například derivace funkce $f(x) = \sqrt{x^6} = |x|^3$ (jako složení odmocniny a mocniny anebo mocniny a absolutní hodnoty) není v bodě $x = 0$ spočítatelná pomocí pravidla řetězení kvůli neexistenci derivace funkce $|v|$ ani $\sqrt{v}$ v bodě $v = 0$. Ve skutečnosti však $f'(0) = 0$. Pokud bychom $\varphi(v) = |v|^c$ jako základní funkci skutečně zavedli, její derivace by byla $\varphi'(v) = c \cdot v \ \cdot |v|^{c-2}$. (Pro $v = 0$ a $c \in (1, 2)$ by bylo $\varphi'(0) = 0$.) Vyhodnocení $\varphi(v)$ a $\varphi'(v)$ jistě bude obsahovat nějaké větvení (např. podle znaménka v), ale to bude až uvnitř v jejich výpočtu. Navenek je φ „jednolitá“ základní funkce.

Předem odkazujeme čtenáře na kapitolu 6.6, ve které je uvedena poznámka o základních funkcích, které jsou definované po částech.

2.3 Alokační funkce

Velmi často se stane, že spočtená hodnota $\boldsymbol{v}_j$ už byla použita na všech pravých stranách výrazů $\boldsymbol{v}_i = \boldsymbol{\varphi}_i(\boldsymbol{v}_j)_{j \prec i}$, kde použita být měla. Proto je zbytečné, aby nadále zabírala paměť, a může být přepsána jinou $\boldsymbol{v}_k$. Umístění jednotlivých $\boldsymbol{v}_i$ v paměti popisuje alokační funkce &.

Až doposud jsme $\boldsymbol{v}_i = \boldsymbol{\varphi}_i(\boldsymbol{v}_j)_{j \prec i}$ považovali za přesnou matematickou definici. I když připustíme přepisování nepotřebných hodnot $\boldsymbol{v}_i$ v paměti, jejich označení $\boldsymbol{v}_i$, $i = 1 - n, \ldots, l$ a onen přesný matematický náhled na $\boldsymbol{v}_i = \boldsymbol{\varphi}_i(\boldsymbol{v}_j)_{j \prec i}$, $i = 1, \ldots, l$ si ponecháme i nadále. Vše bude jasné z následujícího odstavce.

Kompilátor zadaného programu (nebo preprocesor) vygeneruje alokační funkci (alokační schéma) &, která umísťuje hodnoty proměnných $\boldsymbol{v}_i$, $i = 1 - n, \ldots, l$ do místa v paměti, kde budou uchovávány. Pro jednoduchost budeme adresy paměti označovat přirozenými čísly. Neboli

$$\&\boldsymbol{v}_i \subset \mathcal{R} = \{1, 2, \ldots, r\}.$$

Je přirozené, že místo v paměti je znovu použito až tehdy, jestliže jeho současná hodnota $\boldsymbol{v}_i$ byla použita ve všech argumentech, kde použita být měla. Pro jednoduchost budeme dále předpokládat, že

$$\&\boldsymbol{v}_i \cap \&\boldsymbol{v}_j \neq \emptyset \quad \Longrightarrow \quad \&\boldsymbol{v}_i = \&\boldsymbol{v}_j.$$

Neboli, pokud se $\boldsymbol{v}_i$ a $\boldsymbol{v}_j$ překrývají, musí být jako vektory stejně dlouhé a musí se překrývat celé.

Abychom zajistili, že žádná adresa není znovu použita před využitím své hodnoty na všech místech, požadujeme splnění vlastnosti 2.2.

Vlastnost 2.2 (podmínka přímé kompatibility, forward compatibility)

$$j \prec i,\ j < k \leq i \quad \Longrightarrow \quad \&\boldsymbol{v}_k \neq \&\boldsymbol{v}_j$$

□

V případě sčítaného zpětného módu automatického derivování (nonincremental reverse mode, viz kapitola 3.2) budeme požadovat „duální“ vlastnost 2.3.

Vlastnost 2.3 (podmínka zpětné kompatibility, reverse compatibility)

$$j \prec i,\ j \leq k < i \quad \Longrightarrow \quad \&\boldsymbol{v}_i \neq \&\boldsymbol{v}_k$$

□

Podmínka přímé kompatibility zakazuje přepis $\boldsymbol{v}_i$ přes $\boldsymbol{v}_j$, pokud $j \prec i$. Tuto možnost přepisu promyslíme a povolíme v kapitole 3.2.3. Automatickým derivováním odvozené programy budou používat alokační funkci & také na nové proměnné v nich používané (například $\dot{\boldsymbol{v}}_i$, $\bar{\boldsymbol{v}}_i$), jak uvidíme později. Pro odvozování metod automatického derivování je však lepší přepisování neuvažovat.

2.4 Model časové složitosti

V této části představíme model, pomocí kterého budeme měřit práci a čas potřebný pro vyhodnocení jak původního, tak i odvozeného programu. Navíc, budeme schopni vyjádřit tyto náklady odvozeného programu pomocí nákladů původního programu. Upozorněme však na to, že uvedený model má svá slabá místa, například nerespektuje různou přístupovou dobu do paměti apod.

Označme $eval(\boldsymbol{F})$ jako vyhodnocení funkce $\boldsymbol{F}$ a $task(\boldsymbol{F})$ jako nějaký úkol na $\boldsymbol{F}$ (resp. vyhodnocení odvozeného programu, který tento úkol provádí), například vyhodnocení gradientu funkce $\boldsymbol{F}$.

Nějaký úkol (postup atd.) nazveme *aditivní*, pokud je proveden na každou základní funkci $\boldsymbol{\varphi}_i$ v příslušném bodě právě jednou. Např. vyhodnocení soupisu operací je aditivní úkol, protože elementární funkce se vyhodnocují jedna za druhou, jednou v příslušném bodě. Také odvozené programy budou v převážné většině aditivní. Tato aditivita nám přinese jisté výhody (nejen) v modelu složitosti. Příklad neaditivního úkolu, který bude počítat řídkou Jacobiho matici, je uveden v kapitole 6.3.1.

Aditivní úkol nazveme *homogenní*, jestliže jeho provedení na $\boldsymbol{F}$ je realizováno stejnými operacemi na základní funkce $\boldsymbol{\varphi}_i$. Typickým homogenním úkolem je vyhodnocení hodnoty funkce $\boldsymbol{F}$ v daném bodě, nehomogenním úkolem je například vyhodnocení gradientu pomocí zpětného módu automatického derivování.

V případě nehomogenního úkolu $task(\boldsymbol{F})$ budeme příslušný úkol prováděný na základní funkce $\boldsymbol{\varphi}_i$ značit $task'(\boldsymbol{\varphi}_i)$.

Pro jednoduché odhady budeme u aditivních úloh předpokládat splnění vlastnosti 2.4, která se zdá být velmi přirozenou.

Vlastnost 2.4 (aditivita práce)

$$WORK\{eval(\boldsymbol{F})\} = \sum_{i=1}^{l} WORK\{eval(\boldsymbol{\varphi}_i)\} \tag{2.3}$$

$$WORK\{task(\boldsymbol{F})\} \leq \sum_{i=1}^{l} WORK\{task'(\boldsymbol{\varphi}_i)\} \tag{2.4}$$

□

Člen $WORK\{task(\boldsymbol{F})\}$ je nějaký nezáporný skalár nebo vektor s nezápornými prvky, který měří složitost provedení úkolu $task(\boldsymbol{F})$ na funkci $\boldsymbol{F}$, například počet sčítání, násobení a operací s pamětí (v tomto případě ve třísložkovém vektoru).

My budeme používat čtyřsložkový vektor tvaru

$$WORK\{task\} = \begin{pmatrix} \text{počet čtení a zápisů v paměti} \\ \text{počet sčítání} \\ \text{počet násobení} \\ \text{počet nelineárních operací} \end{pmatrix} = \begin{pmatrix} MOVES \\ ADDS \\ MULTS \\ NLOPS \end{pmatrix}. \tag{2.5}$$

Navrženým postupem nerozlišujeme mezi čtením a zápisem do paměti, ani mezi paměťmi s různou přístupovou dobou. Mezi nelineární operace počítáme např. vyhodnocení exponenciály, trigonometrických funkcí nebo převrácené hodnoty (tím dostáváme dělení).

Například pro základní funkci $v_3 = \varphi(v_1, v_2) = v_1 * v_2$ potřebujeme přečíst (vyvolat) z paměti v_1 a v_2, vynásobit je a výsledek v_3 uložit do paměti. Proto

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

Pokud označíme c jako operaci přiřazení konstanty a ψ jako nelineární operaci, dostaneme takto tabulku nákladů pro pouhé vyhodnocení některých funkcí – viz tabulka 2.1.

Takto vzniklou matici označme jako W_{eval}. Jestliže vyhodnocení funkce $\boldsymbol{F}$ vyžaduje l_1 přiřazení konstant, l_2 sčítání, l_3 násobení a spočtení l_4 nelineárních operací, díky vlastnosti 2.4 a označení $|\boldsymbol{F}| = (l_1, l_2, l_3, l_4)^T$ platí

$$WORK\{eval(\boldsymbol{F})\} = W_{eval} \cdot |\boldsymbol{F}| = \begin{pmatrix} 1 & 3 & 3 & 2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} l_1 \\ l_2 \\ l_3 \\ l_4 \end{pmatrix} \tag{2.6}$$

$eval$	c	$+$	$*$	ψ
$MOVES$	1	3	3	2
$ADDS$	0	1	0	0
$MULTS$	0	0	1	0
$NLOPS$	0	0	0	1

Tabulka 2.1: Náklady pro vyhodnocení některých základních funkcí

V následujících kapitolách uvedeme analogické W_{task} pro jiné úkoly (například pro vyhodnocení gradientu), čímž dostaneme relativní porovnání s W_{eval}.

Kromě nákladů na vyhodnocení $WORK\{task(\boldsymbol{F})\}$ budeme ještě sledovat délku času potřebnou na vykonání úkolu $task(\boldsymbol{F})$. Tu můžeme určit jako

$$TIME\{task(\boldsymbol{F})\} = \boldsymbol{w}^T \cdot WORK\{task(\boldsymbol{F})\}, \tag{2.7}$$

kde $\boldsymbol{w}$ je vektor stejné délky jako $WORK$. Obsahuje kladná čísla – množství času potřebné na vykonání operace s pamětí, sečtení (resp. vynásobení) dvou čísel a provedení nelineární operace. Jako okamžitý důsledek vlastnosti 2.4 dostáváme

$$TIME\{eval(\boldsymbol{F})\} = \sum_{i=1}^{l} TIME\{eval(\boldsymbol{\varphi}_i)\} \tag{2.8}$$

$$TIME\{task(\boldsymbol{F})\} \leq \sum_{i=1}^{l} TIME\{task'(\boldsymbol{\varphi}_i)\} \tag{2.9}$$

Pro náš konkrétní model se čtyřsložkovým vektorem $WORK\{eval(\boldsymbol{F})\}$ budeme předpokládat

$$\boldsymbol{w} = \begin{pmatrix} \mu \\ 1 \\ \pi \\ \nu \end{pmatrix}, \tag{2.10}$$

pro který požadujeme

$$\mu \geq \max(1, \pi/2), \quad \pi \geq 1, \quad \nu \geq 2\pi. \tag{2.11}$$

Tyto předpoklady jsou docela přirozené – přístup do paměti trvá alespoň jako sčítání nebo polovina násobení; násobení trvá alespoň jako sčítání; nelineární operace trvá alespoň jako dvě násobení.

Vydělíme-li (2.9) a (2.8), elementárními úpravami dostaneme pro každé $\boldsymbol{F} \in \mathcal{F}$

$$\begin{aligned} \frac{TIME\{task(\boldsymbol{F})\}}{TIME\{eval(\boldsymbol{F})\}} &\leq \frac{\sum_{i=1}^{l} TIME\{task'(\boldsymbol{\varphi}_i)\}}{\sum_{i=1}^{l} TIME\{eval(\boldsymbol{\varphi}_i)\}} \\ &\leq \max_{i=1,\ldots,l} \frac{TIME\{task'(\boldsymbol{\varphi}_i)\}}{TIME\{eval(\boldsymbol{\varphi}_i)\}} \end{aligned}$$

$$\leq \sup_{\varphi \in \Phi} \frac{TIME\{task'(\varphi)\}}{TIME\{eval(\varphi)\}}$$

Tím získáváme stejnoměrnou omezenost pro dobu trvání výpočtu $task(\boldsymbol{F})$ vzhledem k $eval(\boldsymbol{F})$ na celém $\mathcal{F}$, pokud je Φ konečná. V dalším odstavci toto zjištění dále zobecníme.

Omezená složitost

Řekneme, že úkol $task(\boldsymbol{F})$ má *(lineárně) omezenou složitost na* $\mathcal{F}$ (linearly bounded complexity), jestliže existuje matice C_{task} taková, že pro každé $\boldsymbol{F} \in \mathcal{F}$ platí

$$WORK\{task(\boldsymbol{F})\} \leq C_{task} \cdot WORK\{eval(\boldsymbol{F})\} \quad \text{po složkách.} \tag{2.12}$$

Úkol, který nemá omezenou složitost, není ničím výjimečným nebo nebezpečným – může se jednat například o kvadratickou závislost, nikoli lineární, jak předpokládáme.

Například vyhodnocení p sloupců nebo q řádků Jacobiho matice má omezenou složitost. Narozdíl od vyhodnocení celé Jacobiho matice, jehož složitost je úměrná počtu nezávislých proměnných n nebo počtu závislých proměnných m.

Pro aditivní úlohy by omezená složitost na $\mathcal{F}$ byla splněna například tehdy, pokud by Φ byla konečná nebo pokud by množina dvojic

$$\{[WORK(eval(\varphi), WORK(task(\varphi)]; \varphi \in \Phi\}$$

byla konečná. My však použijeme ještě slabší předpoklad na složitost provedení úkolu na základní funkce.

Vlastnost 2.5 (Omezená složitost základních funkcí)
Existuje konečná množina matic $\mathcal{C}_{task}(\Phi)$ taková, že pro každou $\varphi \in \Phi$ existuje $C_{task}(\varphi) \in \mathcal{C}_{task}(\Phi)$, že platí

$$WORK\{task'(\varphi)\} \leq C_{task}(\varphi) \cdot WORK\{eval(\varphi)\} \tag{2.13}$$

□

Z této vlastnosti již snadno odvodíme omezenou složitost na celém $\mathcal{F}$. Označme však ještě

$$C_{task}(\Phi) = \max\{C_{task}(\varphi); \varphi \in \Phi\} \quad \text{po složkách.}$$

Věta 2.4 (Omezenost složitosti)
(i) Nechť pro úkol $task$ platí vlastnosti 2.4 a 2.5. Pak tento úkol má omezenou složitost na $\mathcal{F}$ ve smyslu (2.12), kde

$$C_{task} = C_{task}(\Phi).$$

(ii) Nechť pro úkol $task$ platí vlastnosti 2.4 a 2.5. Pak pro každý vektor $\boldsymbol{w}$ (viz (2.7)) a každou $\boldsymbol{F} \in \mathcal{F}$ platí

$$TIME\{task(\boldsymbol{F})\} \leq \omega_{task} TIME\{eval(\boldsymbol{F})\},$$

kde

$$\omega_{task} = \sup_{\varphi \in \Phi} \frac{\boldsymbol{w}^T C_{task} WORK\{eval(\boldsymbol{\varphi})\}}{\boldsymbol{w}^T WORK\{eval(\boldsymbol{\varphi})\}} \leq ||DC_{task}D^{-1}||_1$$

a kde $D = diag(\boldsymbol{w})$ a $||A||_1 = \max_{j=1,\ldots,n} \sum_{i=1}^{n} |a_{ij}|$ je 1-norma matice.

Důkaz:
(i) Díky elementárním nerovnostem a nezápornosti prvků snadno dostáváme:

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

(ii) Dále analogicky

$$\begin{aligned} \frac{TIME\{task(\boldsymbol{F})\}}{TIME\{eval(\boldsymbol{F})\}} &\leq \frac{\sum_{i=1}^{l} TIME\{task'(\boldsymbol{\varphi}_i)\}}{\sum_{i=1}^{l} TIME\{eval(\boldsymbol{\varphi}_i)\}} \\ &\leq \max_{i=1,\ldots,l} \frac{TIME\{task'(\boldsymbol{\varphi}_i)\}}{TIME\{eval(\boldsymbol{\varphi}_i)\}} \\ &\leq \max_{i=1,\ldots,l} \frac{\boldsymbol{w}^T C_{task}(\boldsymbol{\varphi}) WORK\{eval(\boldsymbol{\varphi}_i)\}}{\boldsymbol{w}^T WORK\{eval(\boldsymbol{\varphi}_i)\}} \\ &\leq \sup_{\varphi \in \Phi} \frac{\boldsymbol{w}^T C_{task}(\Phi) WORK\{eval(\boldsymbol{\varphi})\}}{\boldsymbol{w}^T WORK\{eval(\boldsymbol{\varphi})\}} \\ &\leq \max_{0 \neq z \in \mathbb{R}^{dim(w)}} \frac{\boldsymbol{w}^T C_{task}(\Phi) z}{\boldsymbol{w}^T z} \\ &= \max_{0 \neq z \in \mathbb{R}^{dim(w)}} \frac{||DC_{task}z||_1}{||Dz||_1} \\ &= ||DC_{task}D^{-1}||_1. \end{aligned}$$

Tím je důkaz proveden. □

Věta 2.4 vlastně říká, že čas potřebný pro výpočet odvozené úlohy je stejnoměrně omezený vzhledem k času pro vyhodnocení funkce $\boldsymbol{F}$, a to pro jakýkoliv vektor složitosti $WORK$ a pro jakýkoliv vektor časové náročnosti $\boldsymbol{w}$.

V našem případě čtyřsložkového vektoru složitosti dostáváme z věty 2.4, že

$$\omega_{task} = \max_{i=1,2,3,4} \frac{\boldsymbol{w}^T W_{task} \boldsymbol{e}_i}{\boldsymbol{w}^T W_{eval} \boldsymbol{e}_i}, \tag{2.14}$$

kde $\boldsymbol{e}_i$ jsou i-té vektory kartézské báze pro $\mathbb{R}^4$.

Kapitola 3

Výpočet první derivace

V předcházející kapitole jsme popsali funkci $\boldsymbol{F}$ jako složení základních funkcí $\boldsymbol{\varphi}_i$, $i = 1, \ldots, l$, které jsou alespoň jednou spojitě diferencovatelné. V této kapitole zmíněné předpoklady využijeme a ukážeme, jak zadaný program přetransformovat přímým nebo zpětným módem automatického derivování, aby počítal i požadované derivace. Uvedeme vlastnosti odvozených programů a zmíníme se o složitosti výpočtu.

Automatickým derivováním získáme program, který počítá derivace v zadaném směru (přímý mód automatického derivování) nebo derivace z váženého součtu závislých proměnných (zpětný mód automatického derivování). Tímto přístupem je samozřejmě možné počítat také celé Jacobiho matice atd.

Pro úplnost si zopakujme jedno známé tvrzení.

Věta 3.1

Nechť funkce $f : \mathbb{R}^n \to \mathbb{R}$ má v bodě a totální diferenciál $\mathrm{d}f(a)$. Pak pro derivaci funkce f v bodě a ve směru v, tj. $\partial_v f(a)$, platí

$$\partial_v f(a) = \sum_{i=1}^{n} \frac{\partial f}{\partial x_i}(a) \cdot v_i.$$

□

3.1 Přímý mód automatického derivování

Tento mód automatického derivování (forward mode, tangent) nám umožní počítat derivace všech výstupních proměnných najednou podle zadaného směru.

3.1.1 Odvození a vlastnosti

Pro odvození přímého módu automatického derivování dočasně zakážeme přepisování proměnných. Později jej povolíme.

Celý soupis operací (obrázek 2.2) s pomocí pravidla řetězení (věta 2.2) zderivujeme. Přesněji řečeno, ke každé základní operaci

$$\boldsymbol{v}_i = \boldsymbol{\varphi}_i(\boldsymbol{u}_i)$$

přidáme novou *sdruženou* operaci

$$\dot{\boldsymbol{v}}_i = \sum_{j \prec i} \frac{\partial \boldsymbol{\varphi}_i}{\partial \boldsymbol{v}_j}(\boldsymbol{u}_i) \cdot \dot{\boldsymbol{v}}_j, \tag{3.1}$$

která je derivací původní operace. Tím jsme ke každé proměnné $\boldsymbol{v}_i$ přiřadili novou *sdruženou* proměnnou $\dot{\boldsymbol{v}}_i$, která je její derivací[1]. Úvodní a závěrečnou část soupisu operací doplníme analogicky. Takto získáme *sdružený* soupis operací, který je na obrázku 3.1.

$$\left.\begin{array}{ll} \boldsymbol{v}_{i-n} & = \boldsymbol{x}_i \\ \dot{\boldsymbol{v}}_{i-n} & = \dot{\boldsymbol{x}}_i \end{array}\right\} \; i = 1, \ldots n$$

$$\left.\begin{array}{ll} \boldsymbol{v}_i & = \boldsymbol{\varphi}_i(\boldsymbol{u}_i) \\ \dot{\boldsymbol{v}}_i & = \sum_{j \prec i} \frac{\partial \boldsymbol{\varphi}_i}{\partial \boldsymbol{v}_j}(\boldsymbol{u}_i) \cdot \dot{\boldsymbol{v}}_j \end{array}\right\} \; i = 1, \ldots l$$

$$\left.\begin{array}{ll} \boldsymbol{y}_{m-i} & = \boldsymbol{v}_{l-i} \\ \dot{\boldsymbol{y}}_{m-i} & = \dot{\boldsymbol{v}}_{l-i} \end{array}\right\} \; i = m-1, \ldots 0$$

Obrázek 3.1: Přímým módem automatického derivování odvozený soupis operací

Jeho vstupními parametry jsou bod $\boldsymbol{x}$, ve kterém počítáme derivaci, a hodnota $\dot{\boldsymbol{x}}$, která znamená směr požadované derivace. Pokud chceme určit derivaci podle i-té nezávislé proměnné, tj. ve směru i-tého vektoru kartézské báze $\boldsymbol{e}_i$, vezmeme za $\dot{\boldsymbol{x}}$ právě $\boldsymbol{e}_i$. Pokud bychom požadovali derivaci ve směru $\boldsymbol{s}$, stačí provést tentýž soupis operací jen s tou změnou, že za $\dot{\boldsymbol{x}}$ dosadíme $\boldsymbol{s}$.

Každé $\dot{\boldsymbol{v}}_i$ je derivací $\boldsymbol{v}_i$ v bodě $\boldsymbol{x}$ ve směru $\dot{\boldsymbol{x}}$. Provedením celého sdruženého soupisu operací dostaneme derivace všech výstupních proměnných $\boldsymbol{y}_i$ v bodě $\boldsymbol{x}$ ve směru $\dot{\boldsymbol{x}}$, které budou uloženy v proměnných $\dot{\boldsymbol{y}}_i$, $i = 1, \ldots, m$.

Abychom přesněji nahlédli, proč tomu tak je, uvažujme na chvíli funkci

$$\boldsymbol{x}(\alpha) = \boldsymbol{z} + \alpha \boldsymbol{s} \quad \text{pro } \alpha \in \mathbb{R}.$$

Pak jistě platí

$$\boldsymbol{x}(0) = \boldsymbol{z} \quad \text{a} \quad \boldsymbol{x}'(0) = \boldsymbol{s}.$$

Chceme-li počítat hodnotu derivace $[\boldsymbol{F}(\boldsymbol{x}(\alpha))]'_{\alpha=0}$ odvozeným soupisem operací, musíme podle pravidla řetězení za $\boldsymbol{x}$ dosadit $\boldsymbol{z}$ a za $\dot{\boldsymbol{x}}$ dosadit $\boldsymbol{s}$. Výraz $[\boldsymbol{F}(\boldsymbol{x}(\alpha))]'_{\alpha=0}$ je podle definice derivace ve směru roven právě derivaci $\boldsymbol{F}$ v bodě $\boldsymbol{z}$ ve směru $\boldsymbol{s}$. Odvozeným soupisem operací spočteme

$$[\boldsymbol{F}(\boldsymbol{x}(\alpha))]'_{\alpha=0} = \boldsymbol{F}'(\boldsymbol{x}(0)) \cdot \boldsymbol{x}'(0) = \boldsymbol{F}'(\boldsymbol{z}) \cdot \boldsymbol{s},$$

[1] Během několika řádků bude zřejmé, derivaci podle *čeho* máme na mysli

což v našem značení je

$$\dot{\boldsymbol{y}} = \boldsymbol{F}'(\boldsymbol{x}) \cdot \dot{\boldsymbol{x}},$$

kde $\boldsymbol{F}'(\boldsymbol{x}) \in \mathbb{R}^{m \times n}$ je Jacobiho matice funkce $\boldsymbol{F}$ v bodě $\boldsymbol{x}$.

Hodnoty parciálních derivací

$$\boldsymbol{c}_{ij} = \frac{\partial \boldsymbol{\varphi}_i}{\partial \boldsymbol{v}_j}(\boldsymbol{u}_i)$$

mohou být vyhodnoceny jako číselné hodnoty ještě před tím, než vstoupí do násobení předepsaného pravidlem řetězení. To je základní rozdíl od symbolické derivace, kde $\partial \boldsymbol{\varphi}_i / \partial \boldsymbol{v}_j \, (\boldsymbol{u}_i)$ je symbolický výraz.

Analogicky jako v (2.1) zavedeme označení

$$\dot{\boldsymbol{u}}_i = (\dot{\boldsymbol{v}}_j)_{j \prec i} \in \mathbb{R}^{n_i}. \tag{3.2}$$

Potom už každou sdruženou operaci (3.1) můžeme psát jako

$$\dot{\boldsymbol{v}}_i = \dot{\boldsymbol{\varphi}}_i(\boldsymbol{u}_i, \dot{\boldsymbol{u}}_i). \tag{3.3}$$

Tuto funkci $\dot{\boldsymbol{\varphi}}_i : \mathbb{R}^{2n_i} \to \mathbb{R}^{m_i}$ budeme nazývat *sdružená funkce* k funkci $\boldsymbol{\varphi}_i$.

Výpočet sdružené operace (3.1) jsme zatím vždy prováděli těsně po spočtení $\boldsymbol{v}_i$. Takový přístup vypadá přirozeně, ale není úplně vhodný, pokud $\boldsymbol{v}_i$ v paměti přepíše některý ze svých argumentů $\boldsymbol{v}_j$. V tom případě se totiž $\dot{\boldsymbol{v}}_i = \dot{\boldsymbol{\varphi}}_i(\boldsymbol{u}_i, \dot{\boldsymbol{u}}_i)$ počítá z nesprávných hodnot. Naopak, pro některé základní funkce je výhodné spočíst nejprve $\boldsymbol{v}_i$ a tuto hodnotu pak použít při výpočtu sdružené proměnné $\dot{\boldsymbol{v}}_i$. Proto budeme od tohoto místa psát $\boldsymbol{\varphi}_i$ a $\dot{\boldsymbol{\varphi}}_i$ vedle sebe ve společných závorkách pro zdůraznění, že by měly být vyhodnocovány zároveň se společnými mezivýsledky. Na obrázku 3.2 je formální přepis soupisu operací z obrázku 3.1, do kterého jsme zahrnuli popsané myšlenky.

$[\boldsymbol{v}_{i-n}, \dot{\boldsymbol{v}}_{i-n}]$	$= [\boldsymbol{x}_i, \dot{\boldsymbol{x}}_i]$	$i = 1, \ldots, n$
$[\boldsymbol{v}_i, \dot{\boldsymbol{v}}_i]$	$= [\boldsymbol{\varphi}_i(\boldsymbol{u}_i), \dot{\boldsymbol{\varphi}}_i(\boldsymbol{u}_i, \dot{\boldsymbol{u}}_i)]$	$i = 1, \ldots, l$
$[\boldsymbol{y}_{m-i}, \dot{\boldsymbol{y}}_{m-i}]$	$= [\boldsymbol{v}_{l-i}, \dot{\boldsymbol{v}}_{l-i}]$	$i = m-1, \ldots, 0$

Obrázek 3.2: Přímým módem automatického derivování odvozený soupis operací, obecný tvar

Sdružené operace nejběžnějších základních funkcí jsou uvedeny v tabulce 3.1.

Analogicky s definicí sdružené funkce $\dot{\boldsymbol{\varphi}}_i$ zavedeme ještě „zastřešující“ sdruženou funkci $\dot{\boldsymbol{F}} : \mathbb{R}^{2n} \to \mathbb{R}^m$ k funkci $\boldsymbol{F}$ vztahem

$$\dot{\boldsymbol{y}} = \dot{\boldsymbol{F}}(\boldsymbol{x}, \dot{\boldsymbol{x}})$$

a pro výstupní hodnoty ze sdruženého soupisu operací budeme tedy psát

$$[\boldsymbol{y}, \dot{\boldsymbol{y}}] = \left[\boldsymbol{F}(\boldsymbol{x}), \dot{\boldsymbol{F}}(\boldsymbol{x}, \dot{\boldsymbol{x}})\right].$$

φ	$[\varphi, \dot{\varphi}]$
$v = c$	$v = c$ $\dot{v} = 0$
$v = u \pm w$	$v = u \pm w$ $\dot{v} = \dot{u} \pm \dot{w}$
$v = u * w$	$v = u * w$ $\dot{v} = \dot{u} * w + u * \dot{w}$
$v = 1/u$	$v = 1/u$ $\dot{v} = -v * v * \dot{u}$
$v = \sqrt{u}$	$v = \sqrt{u}$ $\dot{v} = 0.5 * \dot{u} / v$
$v = u^c$	$\dot{v} = \dot{u} / u$ $v = u^c$ $\dot{v} = c * v * \dot{v}$
$v = \exp(u)$	$v = \exp(u)$ $\dot{v} = v * \dot{u}$
$v = \ln(u)$	$v = \ln(u)$ $\dot{v} = \dot{u} / u$
$v = \sin(u)$	$v = \sin(u)$ $\dot{v} = \cos(u) * \dot{u}$

Tabulka 3.1: Některé základní funkce a jejich sdružené operace

Odvozený soupis operací počítá hodnoty směrových derivací $\boldsymbol{F}(\boldsymbol{x}) \cdot \dot{\boldsymbol{x}}$. Zdůrazněme, že během tohoto výpočtu nedojde explicitně ke spočtení celé Jacobiho matice $\boldsymbol{F}'(\boldsymbol{x})$. Analogická situace nastává ve všech metodách dále uvedených, nezmíníme-li jinak.

Jestliže chceme určit derivaci závislých proměnných podle $p \geq 1$ směrů, je velmi vhodné předefinovat směrové derivace $\dot{\boldsymbol{v}}_i \in \mathbb{R}$ (pro jednoduchost předpokládáme $\boldsymbol{v}_i$ skaláry) na vektory $\dot{\boldsymbol{V}}_i \in \mathbb{R}^p$. Vstupem soupisu operací tedy nebude vektor $\dot{\boldsymbol{x}} \in \mathbb{R}^n$, ale matice $\dot{\boldsymbol{X}} \in \mathbb{R}^{n \times p}$, v jejíchž sloupcích budou jednotlivé směry požadovaných derivací. Jinak zůstane soupis operací nezměněn. Jeho provedením dostaneme

$$\dot{\boldsymbol{Y}} = \boldsymbol{F}'(\boldsymbol{x}) \cdot \dot{\boldsymbol{X}} \in \mathbb{R}^{m \times p}.$$

V tom případě můžeme opět zavést $\dot{\boldsymbol{F}}(\boldsymbol{x}, \dot{\boldsymbol{X}}) = \boldsymbol{F}'(\boldsymbol{x}) \cdot \dot{\boldsymbol{X}}$ bez nebezpečí nedorozumění s definicí $\dot{\boldsymbol{F}}(\boldsymbol{x}, \dot{\boldsymbol{x}})$ – podle druhého parametru snadno určíme, jakou $\dot{\boldsymbol{F}}$ vlastně myslíme.

Pokud má základní funkce $\boldsymbol{\varphi}_i$ více argumentů ($n_i > 1$), výpočet celého jejího gradientu (případně Jacobiho matice) a následného násobení $\dot{\boldsymbol{u}}_i$, jak navrhujeme ve sdruženém soupisu operací na obrázku 3.1, nemusí být vždy efektivní. Lepším řešením je na funkci $\boldsymbol{\varphi}_i$ aplikovat automatické derivování lokálně a tak získat směrové derivace $\dot{\boldsymbol{v}}_i$ efektivněji. Tento navržený postup je vlastně rekurzí.

Zatím jsme nepřepisovali proměnné $\boldsymbol{v}_i$, které už nebyly pro výpočet potřebné.

Nyní však jejich přepis povolíme, ale pouze za splnění podmínky přímé kompatibility (vlastnost 2.2).[2] Sdružené proměnné $\dot{\boldsymbol{v}}_i$ budeme do paměti umísťovat právě tak jako $\boldsymbol{v}_i$, neboli budeme požadovat

$$\&\boldsymbol{v}_i = \&\boldsymbol{v}_j \quad \Longleftrightarrow \quad \&\dot{\boldsymbol{v}}_i = \&\dot{\boldsymbol{v}}_j,$$

což může být například splněno, pokud

$$\&\dot{\boldsymbol{v}}_i = \&\boldsymbol{v}_i + c.$$

pro vhodné c. Neboli $\dot{\boldsymbol{v}}_i$ přepíše $\dot{\boldsymbol{v}}_j$ právě tehdy, když $\boldsymbol{v}_i$ přepíše $\boldsymbol{v}_j$.

Poznamenejme ještě, že transformaci původního programu na nový pomocí přímého módu automatického derivování lze provést automaticky, bez zásahu uživatele. Při této transformaci zůstávají řídící příkazy (cykly, podmínky, volání podprogramů atd.) původního programu nezměněny.

Množství paměti potřebné na uložení proměnných odvozeného soupisu operací je a priori omezené. Doba běhu odvozeného soupisu operací vzhledem k trvání původního soupisu operací je také omezená, jak uvidíme v následujícím odstavci.

3.1.2 Složitost odvozeného programu

Úkol výpočtu směrové derivace podle obrázku 3.2 budeme dále označovat jako *tang* (tangent). Jak je zřejmé ze sdruženého soupisu operací, stejný úkol provádíme na $\boldsymbol{F}$ i na každou $\boldsymbol{\varphi}_i$, proto jej můžeme označovat jako homogenní.

Vlastnost 2.4 (aditivita práce) je zřejmě splněna a vlastnost 2.5 (omezená složitost základních funkcí) je snadno splnitelná. Podle věty 2.4 má tedy úkol *tang* omezenou složitost, tj. existuje matice C_{tang}, že pro každé $\boldsymbol{F} \in \mathcal{F}$

$$WORK\{tang(\boldsymbol{F})\} \leq C_{tang} \cdot WORK\{eval(\boldsymbol{F})\}.$$

Dále existuje $\omega_{tang} \in \mathbb{R}$, že pro každé $\boldsymbol{F} \in \mathcal{F}$

$$TIME\{tang(\boldsymbol{F})\} \leq \omega_{tang} TIME\{eval(\boldsymbol{F})\}.$$

Odvodíme přesnou hodnotu C_{tang} a ω_{tang} pro případ čtyřsložkového vektoru $WORK$, který jsme představili v kapitole 2.4. Za tím účelem sestavíme tabulku 3.2, do které uvedeme náklady provedení úkolu *tang* na operace c (přiřazení konstanty), $+$, $*$ a ψ (nelineární funkce). Součástí úkolu *tang* je také vyhodnocení základních funkcí. Tento fakt jsme zdůraznili rozdělením nákladů na dvě části – první číslo jsou náklady na pouhé vyhodnocení, druhé číslo jsou náklady pro výpočet první derivace.

Přiřazení konstanty $v = c$ a příslušná sdružená operace $\dot{v} = 0$ znamenají uložení dvou hodnot do paměti. Binární sčítání $v = u + w$ a sdružená operace $\dot{v} = \dot{u} + \dot{w}$ vyžadují vyvolání hodnot u, w a $\dot{u}$, $\dot{w}$ z paměti, provedení dvou sčítání a následně uložení v a $\dot{v}$ do paměti. Pro binární násobení $v = u * w$ se sdruženou operací $\dot{v} = \dot{u} * w + u * \dot{w}$ a pro nelineární operaci $v = \psi(u)$ se

[2] Iterační přiřazení, pro které $i \prec i$, vysvětlíme a povolíme v kapitole 3.2.3.

tang	c	$+$	$*$	ψ
$MOVES$	$1+1$	$3+3$	$3+3$	$2+2$
$ADDS$	0	$1+1$	$0+1$	0
$MULTS$	0	0	$1+2$	$0+1$
$NLOPS$	0	0	0	$1+1$

Tabulka 3.2: Náklady pro vyhodnocení některých základních funkcí, úkol *tang*

sdruženou operací $\dot{v} = \psi'(u) * \dot{u}$ dostáváme podobné výsledky (v tomto případě $\psi'(u)$ považujeme také za nelineární operaci). Tabulka 3.2 určuje hodnotu matice W_{tang} následujícím způsobem.

$$W_{tang} = \begin{pmatrix} 2 & 6 & 6 & 4 \\ 0 & 2 & 1 & 0 \\ 0 & 0 & 3 & 1 \\ 0 & 0 & 0 & 2 \end{pmatrix} \tag{3.4}$$

Matici C_{tang} můžeme zřejmě spočítat jako $W_{tang} \cdot W_{eval}^{-1}$, kde W_{eval} je definovaná ve vztahu (2.6). Proto

$$C_{tang} = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 1 & 0 \\ 0 & 0 & 3 & 1 \\ 0 & 0 & 0 & 2 \end{pmatrix}. \tag{3.5}$$

Předpokládejme, že doby trvání operace s pamětí, sečtení (resp. vynásobení) dvou čísel a provedení nelineární operace je dána vektorem $\boldsymbol{w}$ podle (2.10). Předpokládáme-li dále (2.11), s pomocí (2.14) dostáváme

$$\omega_{tang} = \max\left\{\frac{2\mu}{\mu}, \frac{6\mu+2}{3\mu+1}, \frac{6\mu+1+3\pi}{3\mu+\pi}, \frac{4\mu+\pi+2\nu}{2\mu+\nu}\right\} \in [2;3]. \tag{3.6}$$

Úkol, ve kterém vyhodnocujeme p směrových derivací najednou, označme jako *tangp*. Analogicky jako u úkolu *tang* odvodíme tabulku nákladů pro některé základní operace – viz tabulka 3.3.

tangp	c	$+$	$*$	ψ
$MOVES$	$1+p$	$3+3p$	$3+3p$	$2+2p$
$ADDS$	0	$1+p$	$0+p$	0
$MULTS$	0	0	$1+2p$	$0+p$
$NLOPS$	0	0	0	$1+1$

Tabulka 3.3: Náklady pro vyhodnocení některých základních funkcí, úkol *tangp*

Odtud určíme matici W_{tangp}.

$$W_{tangp} = \begin{pmatrix} 1+p & 3+3p & 3+3p & 2+2p \\ 0 & 1+p & p & 0 \\ 0 & 0 & 1+2p & p \\ 0 & 0 & 0 & 2 \end{pmatrix} \tag{3.7}$$

Matici C_{tangp} spočteme jako $W_{tangp} \cdot W_{eval}^{-1}$, čímž dostaneme

$$C_{tangp} = \begin{pmatrix} 1+p & 0 & 0 & 0 \\ 0 & 1+p & p & 0 \\ 0 & 0 & 1+2p & p \\ 0 & 0 & 0 & 2 \end{pmatrix}. \tag{3.8}$$

Podle (2.10), (2.11) a (2.14) pro poměr časů platí

$$\begin{aligned} \omega_{tangp} \quad = \quad & \max\left\{ \frac{(1+p)\mu}{\mu}, \frac{(1+p)(3\mu+1)}{3\mu+1}, \frac{3(1+p)\mu+p+(1+2p)\pi}{3\mu+\pi}, \right. \\ & \left. \frac{2(1+p)\mu+p\pi+2\nu}{2\mu+\nu} \right\} \in [1+p; 1+2p]. \end{aligned} \tag{3.9}$$

Podle posledního odhadu se zdá, že program pro výpočet Jacobiho matice odvozený přímým módem automatického derivování může trvat zhruba až dvakrát tak dlouho, jako spočtení aproximace Jacobiho matice poměrnými diferencemi. Jak je ale zmiňováno mnoha autory, tento odhad je nejhorší možný případ. Podle praktických výsledků je poměr časů mnohem menší.

3.2 Zpětný mód automatického derivování

Tak jako přímý mód automatického derivování výrazně těží z malého počtu nezávislých proměnných, zpětný mód automatického derivování (reverse mode, adjoint, gradients) může výrazně těžit z malého počtu závislých proměnných. Například, pro funkci $f : \mathbb{R}^n \to \mathbb{R}$ (se skalární výstupní hodnotou a libovolným n) dostaneme zpětným módem automatického derivování celý její gradient za nejvýše pětinásobek času potřebného pro vyhodnocení původní f. Totéž platí pro gradient váženého součtu jednotlivých složek vektorové funkce $\boldsymbol{F}$. Označme proto

$$\bar{\boldsymbol{x}} = \nabla(\bar{\boldsymbol{y}} \cdot \boldsymbol{F}(\boldsymbol{x})) = \bar{\boldsymbol{y}} \cdot \boldsymbol{F}'(\boldsymbol{x}) \in \mathbb{R}^n. \tag{3.10}$$

Vektor $\bar{\boldsymbol{y}} \in \mathbb{R}^m$ je pevně zadaný řádkový vektor vah, který hraje duální roli ke směrovému vektoru $\dot{\boldsymbol{x}}$ z přímého módu automatického derivování, kde jsme počítali $\dot{\boldsymbol{y}} = \boldsymbol{F}'(\boldsymbol{x}) \cdot \dot{\boldsymbol{x}}$ jako funkci $\boldsymbol{x}$ a $\dot{\boldsymbol{x}}$. Pomocí zpětného módu automatického derivování budeme vyhodnocovat $\bar{\boldsymbol{x}} = \bar{\boldsymbol{y}} \cdot \boldsymbol{F}'(\boldsymbol{x})$ jako funkci $\boldsymbol{x}$ a $\bar{\boldsymbol{y}}$. Zdůrazněme, že přitom nebudeme explicitně počítat matici $\boldsymbol{F}'(\boldsymbol{x})$, ale součin $\bar{\boldsymbol{y}} \cdot \boldsymbol{F}'(\boldsymbol{x})$ určíme přímo.

Pokud je $\boldsymbol{F}$ skalární funkce (tj. $m = 1$), pro $\bar{\boldsymbol{y}} = 1 \in \mathbb{R}$ dostaneme její gradient $\bar{\boldsymbol{x}} = \bar{\boldsymbol{y}} \cdot \boldsymbol{F}'(\boldsymbol{x}) = \nabla \boldsymbol{F}(x)$. Pokud je $\boldsymbol{F}$ vektorová funkce (tj. $m > 1$), vyhodnocením výrazu $\bar{\boldsymbol{x}} = \bar{\boldsymbol{y}} \cdot \boldsymbol{F}'(\boldsymbol{x})$ pro $\bar{\boldsymbol{y}} \in \mathbb{R}^m$ spočítáme lineární kombinaci

gradientů jednotlivých složek $\boldsymbol{F}_i$ s koeficienty $\bar{\boldsymbol{y}}_i$. Pro $\bar{\boldsymbol{y}} = \boldsymbol{e}_i$ tedy dostáváme gradient i-té komponenty funkce $\boldsymbol{F}$.

Poznamenejme ještě, že všechny vektory, které ve svém označení používají pruh, jsou řádkové vektory (zmíníme se i o zobecnění na matice) a že jejich vyhodnocení bude probíhat v opačném pořadí, než jsme zvyklí z přímého módu automatického derivování pro proměnné $\dot{\boldsymbol{v}}_i$. Odtud označení *zpětný* mód. Následující odstavce vše vysvětlí do nejmenšího detailu.

3.2.1 Odvození a vlastnosti

Pro odvození zpětného módu automatického derivování dočasně zakážeme přepisování proměnných. Později jej povolíme s určitými podmínkami pro zachování korektnosti celého výpočtu.

Existuje několik způsobů, jak transformaci soupisu operací (tj. zpětný mód automatického derivování) představit. My využijeme soupis operací na obrázku 3.1, který jsme získali přímým módem automatického derivování. Odvodíme tak výpočet $\bar{\boldsymbol{x}}$ definovaného v (3.10) pro zadané $\boldsymbol{x}$ a $\bar{\boldsymbol{y}}$.

Celý soupis operací na obrázku 3.1 přepíšeme jako součin matic a vektoru $\dot{\boldsymbol{x}}$. Ve shodě s výše uvedeným označme

$$\boldsymbol{c}_{ij} = \frac{\partial \boldsymbol{\varphi}_i}{\partial \boldsymbol{v}_j}(\boldsymbol{u}_i). \tag{3.11}$$

Pro $i = 1, \ldots, l$ značme dále

$$A_i = \begin{pmatrix} 1 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & 0 & 0 & \cdots & 0 \\ \boldsymbol{c}_{i,1-n} & \boldsymbol{c}_{i,2-n} & \cdots & \boldsymbol{c}_{i,i-1} & 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 1 \end{pmatrix} \in \mathbb{R}^{(n+l)\times(n+l)}, \tag{3.12}$$

kde $\boldsymbol{c}_{ij}$ jsou na $(n+i)$-tém řádku.

Označme I_n jednotkovou matici o rozměrech $n \times n$. Dále označme

$$P_n = (I_n, 0, \ldots, 0) \in \mathbb{R}^{n\times(n+l)} \tag{3.13}$$

$$Q_m = (0, \ldots, 0, I_m) \in \mathbb{R}^{m\times(n+l)} \tag{3.14}$$

matice, které zobrazují vektor délky $n+l$ na jeho prvních n složek, resp. posledních m složek.

Nyní již celý postup výpočtu směrové derivace $\dot{\boldsymbol{y}}$ ze směru $\dot{\boldsymbol{x}}$, který je popsán soupisem operací na obrázku 3.1, můžeme vyjádřit jako součin matic tvaru

$$\dot{\boldsymbol{y}} = Q_m A_l A_{l-1} \cdots A_2 A_1 P_n^T \dot{\boldsymbol{x}}, \tag{3.15}$$

protože násobení maticí $P_n^T \in \mathbb{R}^{(n+l)\times n}$ vnoří vektor $\dot{\boldsymbol{x}}$ do $\mathbb{R}^{n+l}$, což odpovídá první části zmíněného soupisu operací. Násobení maticí A_i dává vzniknout proměnné $\dot{\boldsymbol{v}}_i$, jak popisuje prostřední část soupisu operací. Závěrečné vynásobení maticí Q_m přiřadí posledních m složek vektoru $(\dot{\boldsymbol{v}}_{1-n}, \ldots, \dot{\boldsymbol{v}}_l)$ do vektoru $\dot{\boldsymbol{y}}$, jak předepisuje závěrečná část soupisu operací na obrázku 3.1.

Z vlastností přímého módu automatického derivování je jasné, že pro hodnotu $\dot{\boldsymbol{y}}$ spočtenou pomocí (3.15) platí také

$$\dot{\boldsymbol{y}} = \boldsymbol{F}'(\boldsymbol{x}) \cdot \dot{\boldsymbol{x}}. \tag{3.16}$$

Srovnáním vztahů (3.15) a (3.16) získáváme reprezentaci Jacobiho matice jako součin matic ve tvaru

$$\boldsymbol{F}'(\boldsymbol{x}) = Q_m A_l A_{l-1} \cdots A_2 A_1 P_n^T \in \mathbb{R}^{m\times n}. \tag{3.17}$$

Chceme odvodit postup pro výpočet

$$\bar{\boldsymbol{x}} = \bar{\boldsymbol{y}} \cdot \boldsymbol{F}'(\boldsymbol{x}). \tag{3.18}$$

Tento vztah ztransponujeme a za $(\boldsymbol{F}'(\boldsymbol{x}))^T$ dosadíme z rovnosti (3.17). Dostaneme tak

$$\bar{\boldsymbol{x}}^T = P_n A_1^T A_2^T \cdots A_{l-1}^T A_l^T Q_m^T \bar{\boldsymbol{y}}^T. \tag{3.19}$$

Vynásobení vektoru $\bar{\boldsymbol{y}}^T$ maticí Q_m^T zleva vnoří vektor $\bar{\boldsymbol{y}}^T$ do $\mathbb{R}^{n+l}$ s tím, že se jeho hodnota umístí na posledních m prvků. Takto vzniklý vektor budeme nadále značit $(\bar{\boldsymbol{v}}_{1-n}, \ldots, \bar{\boldsymbol{v}}_l)^T$ a jeho jednotlivé složky nazývat *přidružené proměnné* (adjoint variables). Vynásobíme-li jej zleva maticí A_i^T, všechny $\bar{\boldsymbol{v}}_j$ pro $j \not\prec i$ zůstávají nezměněny, zatímco všechny $\bar{\boldsymbol{v}}_j$ pro $j \prec i$ jsou zvětšeny o $\bar{\boldsymbol{v}}_i \cdot \boldsymbol{c}_{ij}$. Samotné $\bar{\boldsymbol{v}}_i$ je následně vynulováno. Nakonec po vynásobení maticí A_1^T je pouze prvních n hodnot vektoru $(\bar{\boldsymbol{v}}_{1-n}, \ldots, \bar{\boldsymbol{v}}_l)$ nenulových. Ty jsou díky vynásobení maticí P_n přiřazeny do $\bar{\boldsymbol{x}}$.

Poznámka k zápisu operací.

Velmi často budeme provádět operace, které k proměnné a přičtou hodnotu b a výsledek uloží do a. Takové operace budeme zapisovat

$$a = a + b,$$

a to i v případě matematických formulí. Bylo by zřejmě lepší používat zápis

$$a := a + b,$$

ale pro jednoduchost tak nebudeme činit. Pamatujme tedy, že výraz $a = a + b$ není ekvivalentní s výrazem $0 = b$. Vše bude jasné z dalšího výkladu, takže žádné nedorozumnění nehrozí.

Nyní již můžeme výše odvozený postup (3.19) přepsat do *přidruženého* soupisu operací[3], který je na obrázku 3.3. Oproti vzorci (3.19) jsme do něj zahrnuli malé

[3]Na první pohled toto není soupis operací přesně podle naší definice v kapitole 2.1. S trochou práce a fantazie je však možné tuto *vyhodnocovací proceduru* za soupis operací považovat. Proto ji (a podobné vyhodnocovací procedury) dále budeme soupisem operací nazývat.

$\bar{\boldsymbol{v}}_i = 0$		$i = 1-n, \ldots, l$
$\boldsymbol{v}_{i-n} = \boldsymbol{x}_i$		$i = 1, \ldots, n$
$\boldsymbol{v}_i = \boldsymbol{\varphi}_i(\boldsymbol{v}_j)_{j \prec i}$		$i = 1, \ldots, l$
$\boldsymbol{y}_{m-i} = \boldsymbol{v}_{l-i}$		$i = m-1, \ldots, 0$
$\bar{\boldsymbol{v}}_{l-i} = \bar{\boldsymbol{y}}_{m-i}$		$i = 0, \ldots, m-1$
$\bar{\boldsymbol{v}}_j = \bar{\boldsymbol{v}}_j + \bar{\boldsymbol{v}}_i \cdot \frac{\partial \boldsymbol{\varphi}_i}{\partial \boldsymbol{v}_j}(\boldsymbol{u}_i)$	pro $j \prec i$	$i = l, \ldots, 1$
$\bar{\boldsymbol{x}}_i = \bar{\boldsymbol{v}}_{i-n}$		$i = n, \ldots, 1$

Obrázek 3.3: Nasčítávaným zpětným módem automatického derivování odvozený soupis operací

úpravy: přidružené proměnné $\bar{\boldsymbol{v}}_i$ pro $i = 1-n, \ldots, l$ jsou na začátku soupisu operací vynulovány. Naopak, vynulování proměnných $\bar{\boldsymbol{v}}_i$ po všech jejich výskytech na pravých stranách je vynecháno.

Tento přidružený soupis operací se skládá ze dvou částí, které jsou odděleny dvojitou čarou. V první části probíhá výpočet funkční hodnoty $\boldsymbol{F}(\boldsymbol{x})$, ve druhé části akumulace hodnot do přidružených proměnných $\bar{\boldsymbol{v}}_j$. Tuto akumulaci provádíme „pozpátku“, jak můžeme pozorovat na indexech v předposledním řádku soupisu operací. Později uvidíme, že v první části budeme navíc provádět určité pomocné operace, kterých využijeme ve druhé části (například uschování hodnot některých proměnných atp.).

Uvedeme malou modifikaci právě odvozeného soupisu operací. Numerické výsledky obou verzí jsou naprosto identické. Rozdíly jsou pouze ve způsobu, jak výsledků dosáhnou: hodnotu přidružených proměnných $\bar{\boldsymbol{v}}_j$ můžeme spočítat také postupným přičítáním příspěvků přes všechny následovníky $i \succ j$, jak je naznačeno v soupisu operací na obrázku 3.4. Tento přístup je možná teoreticky přehlednější, ale jeho praktická realizace skrývá jisté obtíže: nemáme k dispozici seznam následníků $i \succ j$. Navíc z pouhého seznamu základních funkcí $\boldsymbol{\varphi}_i$ a jejich argumentů $\boldsymbol{u}_i$ je tato informace hůře zjistitelná než hledání předchůdců $j \prec i$, které potřebujeme pro původně představený soupis operací na obrázku 3.3.

V souvislosti s těmito úvahami upravme používanou terminologii. Původně představený soupis operací (obr. 3.3) jsme dostali *nasčítávaným zpětným módem automatického derivování* (incremental reverse mode), zatímco upravený soupis operací (obr. 3.4) jsme získali *sčítaným zpětným módem automatického derivování* (nonincremental reverse mode). Tyto názvy, byť jsou podobné a nepřehledné, dobře vystihují podstatu metody a rozdíly ve vlastnostech.

Poznamenejme ještě, že v Malém motivačním příkladu v kapitole 1 jsme předvedli nasčítávaný zpětný mód automatického derivování (obrázek 1.3): nejdříve se provede pouhé vyhodnocení f. Pak se za $\bar{y}$ dosadí jednička. Postupně uvažujeme jednotlivé operace z postupu vyhodnocení f v opačném pořadí a provádíme příslušné přidružené operace. Nakonec je v proměnné $\bar{v}_{-1}$ spočtena hodnota de-

$\boldsymbol{v}_{i-n} = \boldsymbol{x}_i$		$i = 1, \ldots, n$
$\boldsymbol{v}_i = \boldsymbol{\varphi}_i(\boldsymbol{v}_j)_{j \prec i}$		$i = 1, \ldots, l$
$\boldsymbol{y}_{m-i} = \boldsymbol{v}_{l-i}$		$i = m-1, \ldots, 0$
$\bar{\boldsymbol{v}}_{l-i} = \bar{\boldsymbol{y}}_{m-i}$		$i = 0, \ldots, m-1$
$\bar{\boldsymbol{v}}_j = \sum_{i \succ j} \bar{\boldsymbol{v}}_i \cdot \frac{\partial \boldsymbol{\varphi}_i}{\partial \boldsymbol{v}_j}(\boldsymbol{u}_i)$		$j = l-m, \ldots, 1-n$
$\bar{\boldsymbol{x}}_i = \bar{\boldsymbol{v}}_{i-n}$		$i = n, \ldots, 1$

Obrázek 3.4: Sčítaným zpětným módem automatického derivování odvozený soupis operací

rivace y podle t v příslušném bodě.

Zatím jsme na proměnné $\bar{\boldsymbol{v}}_i$ pohlíželi jako na pomocné proměnné. Lze ukázat, že jejich výsledné hodnoty charakterizují citlivost $\bar{\boldsymbol{y}} \cdot \boldsymbol{F}(\boldsymbol{x})$ na malou změnu $\boldsymbol{v}_i$. Přesně řečeno, změňme operaci $\boldsymbol{v}_i = \boldsymbol{\varphi}_i(\boldsymbol{u}_i)$ ze soupisu operací na obrázku 2.2 na operaci

$$\boldsymbol{v}_i = \boldsymbol{\varphi}_i(\boldsymbol{u}_i) + \delta_i, \tag{3.20}$$

kde δ_i chápeme jako novou nezávislou proměnnou[4]. Pak již lze ukázat, že

$$\bar{\boldsymbol{v}}_i = \frac{\partial}{\partial \delta_i}(\bar{\boldsymbol{y}} \cdot \boldsymbol{F}(\boldsymbol{x}))\bigg|_{\delta_i = 0}. \tag{3.21}$$

Podrobnosti lze najít například v [1].

Za těchto předpokladů je možné výslednou poruchu δ hodnoty $\bar{\boldsymbol{y}} \cdot \boldsymbol{F}(\boldsymbol{x})$ přibližně odhadnout jako

$$\delta \approx \sum_{i=1}^{l} \delta_i \bar{\boldsymbol{v}}_i. \tag{3.22}$$

Pokud budeme δ_i chápat jako zaokrouhlovací chybu vzniklou při výpočtu operace $\boldsymbol{v}_i = \boldsymbol{\varphi}_i(\boldsymbol{u}_i)$, pak má zřejmě tvar $\delta_i = \eta_i \boldsymbol{v}_i$ pro $|\eta_i| \leq \eta$. Odtud již snadno dostaneme přibližnou celkovou chybu pro $\bar{\boldsymbol{y}} \cdot \boldsymbol{F}(\boldsymbol{x})$ ve tvaru

$$|\delta| \lessapprox \left| \sum_{i=1}^{l} \eta_i \bar{\boldsymbol{v}}_i \boldsymbol{v}_i \right| \leq \eta \sum_{i=1}^{l} |\bar{\boldsymbol{v}}_i \boldsymbol{v}_i|. \tag{3.23}$$

Tento výraz bývá často používán jako odhad zaokrouhlovací chyby při výpočtu hodnoty $\bar{\boldsymbol{y}} \cdot \boldsymbol{F}(\boldsymbol{x})$.

Pokud bychom na $\bar{\boldsymbol{v}}_i$ pohlíželi právě ve smyslu (3.21), lze zpětný mód automatického derivování odvodit přímo z pravidla řetězení. Podrobnosti lze opět najít v [1].

Ve shodě s (2.1) a (3.2) označme

$$\bar{\boldsymbol{u}}_i = (\bar{\boldsymbol{v}}_j)_{j \prec i}. \tag{3.24}$$

[4]Pokud je $\boldsymbol{v}_i$ vektor, lze vše popsat analogicky.

Pak *přidruženou* operaci

$$\bar{\boldsymbol{v}}_j = \bar{\boldsymbol{v}}_j + \bar{\boldsymbol{v}}_i \cdot \boldsymbol{c}_{ij} \quad \text{pro } j \prec i \tag{3.25}$$

označujme jako

$$\bar{\boldsymbol{u}}_i = \bar{\boldsymbol{u}}_i + \bar{\boldsymbol{\varphi}}_i(\boldsymbol{u}_i, \bar{\boldsymbol{v}}_i). \tag{3.26}$$

Funkci $\bar{\boldsymbol{\varphi}}_i : \mathbb{R}^{n_i+m_i} \longrightarrow \mathbb{R}^{n_i}$ budeme nazývat funkce *přidružená* k funkci $\boldsymbol{\varphi}_i$.

Až doposud jsme nepřepisovali proměnné $\boldsymbol{v}_i$, i když už nebyly při výpočtu $\boldsymbol{F}(\boldsymbol{x})$ dále potřeba. Nyní přepis povolíme. Budeme však předpokládat splnění podmínky přímé kompatibility (vlastnost 2.2); iterace (neboli $i \prec i$) povolíme v kapitole 3.2.3. Ve sčítaném zpětném módu předpokládáme navíc splnění podmínky zpětné kompatibility (vlastnost 2.3). Narozdíl od přímého módu automatického derivování zde nebude situace tak jednoduchá, protože hodnoty $\boldsymbol{v}_i$ jsou potřeba jako součásti $\boldsymbol{u}_i$ při výpočtu $\boldsymbol{c}_{ij}$ ve druhé části přidruženého soupisu operací, tj. výpočtu $\bar{\boldsymbol{v}}_i$. Proto potřebujeme hodnoty $\boldsymbol{v}_i$ nějakým způsobem uchovat.

Jeden z možných přístupů k tomuto problému je při provedení operace $\boldsymbol{v}_i = \boldsymbol{\varphi}_i(\boldsymbol{u}_i)$ uschovat si celý vektor $\boldsymbol{u}_i$ a ve druhé části přidruženého soupisu operací z něj počítat příslušné $\boldsymbol{c}_{ij}$. Tento způsob však není velikostí potřebné paměti příliš vhodný. Bezesporu přijatelnější řešení je ukládat parciální derivace $\boldsymbol{c}_{ij}$. My se budeme věnovat ještě jinému přístupu, totiž před každou elementární operací $\boldsymbol{v}_i = \boldsymbol{\varphi}_i(\boldsymbol{u}_i)$ si uschováme hodnotu $\boldsymbol{v}_i$, aby již mohla být přepsána. Tuto uschovanou hodnotu obnovíme pro správný výpočet $\boldsymbol{c}_{ij}$ před příslušnou přidruženou operací. Hodnoty, které jsme uschovali, budeme tedy potřebovat přesně v opačném pořadí. Za účelem jejich uložení použijeme zásobník, který budeme značit $\mathcal{S}$. Symbol $\boldsymbol{v} \rightarrow \mathcal{S}$ ať znamená uložení hodnoty $\boldsymbol{v}$ na zásobník, symbol $\boldsymbol{v} \leftarrow \mathcal{S}$ vyzvednutí hodnoty $\boldsymbol{v}$ ze zásobníku. Proto v první části přidruženého soupisu operací provedeme pro každou $\boldsymbol{\varphi}_i$ operace

$$\begin{aligned} \boldsymbol{v}_i &\rightarrow \mathcal{S} \\ \boldsymbol{v}_i &= \boldsymbol{\varphi}_i(\boldsymbol{v}_j)_{j \prec i}, \end{aligned}$$

zatímco v jeho druhé části provedeme

$$\begin{aligned} \boldsymbol{v}_i &\leftarrow \mathcal{S} \\ \bar{\boldsymbol{v}}_j &= \bar{\boldsymbol{v}}_j + \bar{\boldsymbol{v}}_i \cdot \boldsymbol{c}_{ij} \quad \text{pro } j \prec i \\ \bar{\boldsymbol{v}}_i &= 0. \end{aligned}$$

Všimněme si nulování $\bar{\boldsymbol{v}}_i$, které je v případě přepisování proměnných nutností.

Stejně jako v přímém módu, při výpočtu přidružených operací lze v některých případech výhodně použít již spočtenou hodnotu základní funkce $\boldsymbol{\varphi}_i$. Proto je někdy lepší nejdříve provést přidruženou operaci a teprve potom vyzvednout hodnoty $\boldsymbol{v}_i$ ze zásobníku. Toto můžeme pozorovat na příkladech přidružených operací pro některé základní funkce v tabulce 3.4. Zde operace $\vec{\boldsymbol{\varphi}}_i$ (resp. $\overleftarrow{\boldsymbol{\varphi}}_i$) znamenají kombinaci $\boldsymbol{\varphi}_i$ (resp. $\bar{\boldsymbol{\varphi}}_i$) s příslušnou zásobníkovou operací.

$\vec{\varphi}$	$\overleftarrow{\varphi}$
$v \to \mathcal{S}$	$v \leftarrow \mathcal{S}$
$v = c$	$\bar{v} = 0$
$v \to \mathcal{S}$	$v \leftarrow \mathcal{S}$
$v = u + w$	$\bar{u} = \bar{u} + \bar{v}$
	$\bar{w} = \bar{w} + \bar{v}$
	$\bar{v} = 0$
$v \to \mathcal{S}$	$v \leftarrow \mathcal{S}$
$v = u * w$	$\bar{u} = \bar{u} + \bar{v} * w$
	$\bar{w} = \bar{w} + \bar{v} * u$
	$\bar{v} = 0$
$v \to \mathcal{S}$	$\bar{u} = \bar{u} - \bar{v} * v * v$
$v = 1/u$	$v \leftarrow \mathcal{S}$
	$\bar{v} = 0$
$v \to \mathcal{S}$	$\bar{u} = \bar{u} + 0.5 * \bar{v}/v$
$v = \sqrt{u}$	$v \leftarrow \mathcal{S}$
	$\bar{v} = 0$
$v \to \mathcal{S}$	$\bar{u} = \bar{u} + \bar{v} * c * v/u$
$v = u^c$	$v \leftarrow \mathcal{S}$
	$\bar{v} = 0$
$v \to \mathcal{S}$	$\bar{u} = \bar{u} + \bar{v} * v$
$v = \exp(u)$	$v \leftarrow \mathcal{S}$
	$\bar{v} = 0$
$v \to \mathcal{S}$	$v \leftarrow \mathcal{S}$
$v = \ln(u)$	$\bar{u} = \bar{u} + \bar{v}/u$
	$\bar{v} = 0$
$v \to \mathcal{S}$	$v \leftarrow \mathcal{S}$
$v = \sin(u)$	$\bar{u} = \bar{u} + \bar{v} * \cos(u)$
	$\bar{v} = 0$

Tabulka 3.4: Některé základní funkce a jejich přidružené operace

Přidružený soupis operací z obrázku 3.3 nyní můžeme přepsat do poněkud stručnější formy – viz obrázek 3.5. Mimo jiné úpravy jsme také umožnili, aby do $\bar{\boldsymbol{v}}_{i-n}$, $i = 1, \ldots, n$ byly přiřazeny nenulové počáteční hodnoty $\bar{\boldsymbol{x}}_i$. Provedením tohoto soupisu operací dostáváme (mimo jiné) výsledek

$$\bar{\boldsymbol{x}} = \bar{\boldsymbol{x}} + \bar{\boldsymbol{F}}(\boldsymbol{x}, \bar{\boldsymbol{y}}), \tag{3.27}$$

kde

$$\begin{aligned} \bar{\boldsymbol{F}} &: \mathbb{R}^{n+m} \longrightarrow \mathbb{R}^n \\ \bar{\boldsymbol{F}} &: (\boldsymbol{x}, \bar{\boldsymbol{y}}) \longmapsto \bar{\boldsymbol{y}} \cdot \boldsymbol{F}'(\boldsymbol{x}). \end{aligned}$$

$\bar{\boldsymbol{v}}_i = 0$		$\}\ i = 1, \ldots, l$
$\boldsymbol{v}_{i-n} = \boldsymbol{x}_i$ $\bar{\boldsymbol{v}}_{i-n} = \bar{\boldsymbol{x}}_i$		$\left.\right\}\ i = 1, \ldots, n$
$\boldsymbol{v}_i \to \mathcal{S}$ $\boldsymbol{v}_i = \boldsymbol{\varphi}_i(\boldsymbol{u}_i)$		$\left.\right\}\ i = 1, \ldots, l$
$\boldsymbol{y}_{m-i} = \boldsymbol{v}_{l-i}$		$\}\ i = m-1, \ldots, 0$
$\bar{\boldsymbol{v}}_{l-i} = \bar{\boldsymbol{y}}_{m-i}$		$\}\ i = 0, \ldots m-1$
$\boldsymbol{v}_i \leftarrow \mathcal{S}$ $\bar{\boldsymbol{u}}_i = \bar{\boldsymbol{u}}_i + \bar{\boldsymbol{\varphi}}_i(\boldsymbol{u}_i, \bar{\boldsymbol{v}}_i)$ $\bar{\boldsymbol{v}}_i = 0$		$\left.\right\}\ i = l, \ldots, 1$
$\bar{\boldsymbol{x}}_i = \bar{\boldsymbol{v}}_{i-n}$		$\}\ i = n, \ldots, 1$

Obrázek 3.5: Nasčítávaným zpětným módem automatického derivování odvozený soupis operací, obecný tvar

Pokud chceme vyhodnotit více gradientů s různými vahami najednou, můžeme podobně jako v přímém módu předefinovat vektor $\bar{\boldsymbol{y}} \in \mathbb{R}^m$ na q-řádkovou matici $\bar{\boldsymbol{Y}} \in \mathbb{R}^{q \times m}$. V jednotlivých řádcích uvedeme váhové vektory. V přidruženém soupisu operací je potřeba nahradit (pro jednoduchost) skalární proměnné $\bar{\boldsymbol{v}}_i \in \mathbb{R}$ vektory $\bar{\boldsymbol{V}}_i \in \mathbb{R}^q$. Tak dostaneme[5]

$$\bar{\boldsymbol{X}} = \bar{\boldsymbol{X}} + \bar{\boldsymbol{F}}(\boldsymbol{x}, \bar{\boldsymbol{Y}}) = \bar{\boldsymbol{X}} + \bar{\boldsymbol{Y}} \cdot \boldsymbol{F}'(\boldsymbol{x}), \tag{3.28}$$

kde $\bar{\boldsymbol{F}} : \mathbb{R}^n \times \mathbb{R}^{q \times m} \to \mathbb{R}^{q \times n}$.

Zamysleme se nad tím, v jaké fázi průběhu soupisu operací získaného nasčítávaným zpětným módem automatického derivování musíme mít přístupnou proměnnou $\bar{\boldsymbol{v}}_i$. Pokud jí příslušející proměnná $\boldsymbol{v}_i$ existuje v první části přidruženého soupisu operací v „intervalu indexů základních funkcí“ $[i, k]$ ($\boldsymbol{v}_k$ je poslední proměnná, pro jejíž výpočet je $\boldsymbol{v}_i$ potřeba), pak $\bar{\boldsymbol{v}}_i$ musí existovat v tomtéž intervalu, ale ve druhé části přidruženého soupisu operací, tedy vlastně v „otočeném intervalu“ $[k, i]$. Díky tomu, že proměnná $\boldsymbol{v}_i$ obsahuje správné hodnoty i ve druhé části soupisu operací v tomtéž intervalu $[k, i]$, můžeme paměť pro $\bar{\boldsymbol{v}}_i$ alokovat způsobem

$$\&\bar{\boldsymbol{v}}_i = \&\boldsymbol{v}_i + \tilde{c}. \tag{3.29}$$

pro vhodné $\tilde{c}$. Obecněji řečeno, paměť lze alokovat za splnění podmínky

$$\&\boldsymbol{v}_i = \&\boldsymbol{v}_j \quad \Longleftrightarrow \quad \&\bar{\boldsymbol{v}}_i = \&\bar{\boldsymbol{v}}_j.$$

Podmínka přímé kompatibility (vlastnost 2.2), kterou předpokládáme splněnou, implikuje

$$j \prec i,\ j < k\ \leq i \quad \Longrightarrow \quad \&\bar{\boldsymbol{v}}_k \neq \&\bar{\boldsymbol{v}}_j. \tag{3.30}$$

[5] Zde platí analogická poznámka jako u přímého módu, která se týká definice funkce $\dot{\boldsymbol{F}}(\boldsymbol{x}, \dot{\boldsymbol{x}})$, resp. $\dot{\boldsymbol{F}}(\boldsymbol{x}, \dot{\boldsymbol{X}})$ na straně 24.

Bez této vlastnosti by přidružený soupis operací na obrázku 3.5 nebyl korektní.

Pro případ sčítaného zpětného módu připomeňme, že pro jeho korektní průběh je nutné navíc požadovat splnění podmínky zpětné kompatibility (vlastnost 2.3), a to jak pro proměnné $\boldsymbol{v}_i$, tak pro proměnné $\bar{\boldsymbol{v}}_i$.

Poznamenejme ještě, že transformaci původního programu na nový pomocí zpětného módu automatického derivování lze provést automaticky, bez zásahu uživatele. Tato tranformace ale nebude tak triviální jako u přímého módu.

Množství paměti potřebné na uložení proměnných odvozeného soupisu operací je a priori omezené.[6] Doba běhu odvozeného soupisu operací vzhledem k trvání původního soupisu operací je také omezená, jak uvidíme z následujícího odstavce.

3.2.2 Složitost odvozeného programu

Úkol provedení soupisu operací získaného nasčítávaným zpětným módem automatického derivování na obrázku 3.5, tj. výpočet hodnot $\boldsymbol{F}(\boldsymbol{x})$ a $\bar{\boldsymbol{X}} + \bar{\boldsymbol{F}}(\boldsymbol{x}, \bar{\boldsymbol{y}})$, budeme značit $grad(\boldsymbol{F})$. Pro každou základní funkci $\boldsymbol{\varphi}_i$ musíme pro tento úkol provést následující operace: v první části soupisu operací uložit hodnotu $\boldsymbol{v}_i$ na zásobník, vyhodnotit $\boldsymbol{\varphi}_i(\boldsymbol{u}_i)$ a uložit ji do $\boldsymbol{v}_i$; ve druhé části soupisu operací obnovit hodnotu $\boldsymbol{v}_i$ ze zásobníku, spočíst $\bar{\boldsymbol{\varphi}}_i(\boldsymbol{u}_i, \bar{\boldsymbol{v}}_i)$ a přičíst ji do $\bar{\boldsymbol{u}}_i$ a nakonec vynulovat $\bar{\boldsymbol{v}}_i$. Tyto činnosti budeme dohromady značit jako $grad'(\boldsymbol{\varphi}_i)$, proto

$$\begin{aligned} WORK\{grad'(\boldsymbol{\varphi}_i)\} \quad = \quad & WORK\{\boldsymbol{v}_i = \boldsymbol{\varphi}_i(\boldsymbol{u}_i)\} + WORK\{\bar{\boldsymbol{u}}_i = \bar{\boldsymbol{u}}_i + \bar{\boldsymbol{\varphi}}_i(\boldsymbol{u}_i, \bar{\boldsymbol{v}}_i)\} + \\ & WORK\{\boldsymbol{v}_i \rightarrow \mathcal{S}\} + WORK\{\boldsymbol{v}_i \leftarrow \mathcal{S}\} + \\ & WORK\{\boldsymbol{v}_i = 0\} \end{aligned} \tag{3.31}$$

Výraz $WORK\{\boldsymbol{v}_i \rightarrow \mathcal{S}\} + WORK\{\boldsymbol{v}_i \leftarrow \mathcal{S}\}$ je nulový, pokud nepřepisujeme proměnné. Pro jednoduchost našich úvah ho nebudeme v dalších odhadech složitosti uvažovat, i když se tím dopustíme jisté nepřesnosti.[7]

Na úkol $grad(\boldsymbol{F})$ můžeme při splnění vlastností 2.4 a 2.5 aplikovat větu 2.4, která zaručí existenci matice C_{grad}, že pro všechna $\boldsymbol{F} \in \mathcal{F}$

$$WORK\{grad(\boldsymbol{F})\} \leq C_{grad} \cdot WORK\{eval(\boldsymbol{F})\}.$$

Dále existuje $\omega_{grad} \in \mathbb{R}$, že pro každé $\boldsymbol{F} \in \mathcal{F}$

$$TIME\{grad(\boldsymbol{F})\} \leq \omega_{grad} TIME\{eval(\boldsymbol{F})\}.$$

Abychom získali hodnoty C_{grad} a ω_{grad}, uvažujme opět čtyřsložkový vektor složitosti představený v kapitole 2.4. Budeme vyšetřovat operace c (přiřazení konstanty), $+$, $*$ a ψ (nelineární funkce).

Přiřazení konstanty $v = c$ spolu s příslušnou přidruženou operací $\bar{v} = 0$ znamená uložení dvou hodnot do paměti. Binární sčítání $v = u + w$ a přidružené

[6] Při implementaci automatického derivování však nebude situace tak jasná, viz kapitola 5.

[7] Uvědomme si, že pokud se proměnné $\boldsymbol{v}_i$ nepřepisují, nemusíme zásobník $\mathcal{S}$ vůbec používat. Navíc, pokud už ho budeme používat, pak zdaleka ne všechny hodnoty $\boldsymbol{v}_i$ na něj musíme ukládat, jak uvidíme v kapitole 6.1.

operace $\bar{u} \;= \bar{u} \;+ \bar{v}$, $\bar{w} = \bar{w} + \bar{v}$ a $\bar{v} \;= 0$ vyžadují vyvolání hodnot u, w a $\bar{u}$, $\bar{w}$, $\bar{v}$ z paměti, provedení tří sčítání a následně uložení v, $\bar{u}$, $\bar{w}$ a $\bar{v}$ do paměti. Pro binární násobení $v = u \;* w$ s přidruženými operacemi $\bar{u} \;= \bar{u} \;+ \bar{v} \;* w$, $\bar{w} = \bar{w} + \bar{v} \;* u$, $\bar{v} \;= 0$ a pro nelineární operaci $v = \psi(u)$ s přidruženými operacemi $\bar{u} \;= \psi'(u) * \bar{v}$, $\bar{v} \;= 0$ dostáváme podobné výsledky (v tomto případě $\psi'(u)$ považujeme také za nelineární operaci). Uvedené hodnoty jsme shrnuli do tabulky 3.5.

$grad'$	c	$+$	$*$	ψ
$MOVES$	$1+1$	$3+6$	$3+8$	$2+5$
$ADDS$	0	$1+2$	$0+2$	$0+1$
$MULTS$	0	0	$1+2$	$0+1$
$NLOPS$	0	0	0	$1+1$

Tabulka 3.5: Náklady pro vyhodnocení některých základních funkcí, úkol $grad'$

Tyto hodnoty určují matici $W_{grad'}$.

$$W_{grad'} = \begin{pmatrix} 2 & 9 & 11 & 7 \\ 0 & 3 & 2 & 1 \\ 0 & 0 & 3 & 1 \\ 0 & 0 & 0 & 2 \end{pmatrix} \tag{3.32}$$

Matici C_{grad} můžeme zřejmě spočítat jako $W_{grad'} \cdot W_{eval}^{-1}$, kde W_{eval} je definovaná ve vztahu (2.6). Proto

$$C_{grad} = \begin{pmatrix} 2 & 3 & 5 & 3 \\ 0 & 3 & 2 & 1 \\ 0 & 0 & 3 & 1 \\ 0 & 0 & 0 & 2 \end{pmatrix}. \tag{3.33}$$

Předpokládejme, že doby trvání operace s pamětí, sečtení (resp. vynásobení) dvou čísel a provedení nelineární operace je dána vektorem $\boldsymbol{w}$ podle (2.10). Předpokládáme-li dále (2.11), s pomocí (2.14) dostáváme

$$\omega_{grad} = \max\left\{\frac{2\mu}{\mu}, \frac{9\mu+3}{3\mu+1}, \frac{11\mu+2+3\pi}{3\mu+\pi}, \frac{7\mu+1+\pi+2\nu}{2\mu+\nu}\right\} \in [3;4]. \tag{3.34}$$

V případě sčítaného zpětného módu automatického derivování bychom potřebovali o něco méně operací – ušetřili bychom nulování $\bar{\boldsymbol{v}}_i$ a první přičítání do $\bar{\boldsymbol{v}}_i$ by bylo přiřazení.

Úkol, ve kterém vyhodnocujeme q gradientů s různými vahami najednou, označme jako $gradq$. Analogicky jako u úkolu $grad$ odvodíme tabulku nákladů pro úkol $gradq'$ pro některé základní operace – viz tabulka 3.6.

Odtud určíme matici $W_{gradq'}$.

$gradq'$	c	$+$	$*$	ψ
$MOVES$	$1+q$	$3+6q$	$5+6q$	$3+4q$
$ADDS$	0	$1+2q$	$0+2q$	$0+q$
$MULTS$	0	0	$1+2q$	$0+q$
$NLOPS$	0	0	0	2

Tabulka 3.6: Náklady pro vyhodnocení některých základních funkcí, úkol $gradq'$

$$W_{gradq'} = \begin{pmatrix} 1+q & 3+6q & 5+6q & 3+4q \\ 0 & 1+2q & 2q & q \\ 0 & 0 & 1+2q & q \\ 0 & 0 & 0 & 2 \end{pmatrix} \tag{3.35}$$

Matici C_{gradq} spočteme jako $W_{gradq'} \cdot W_{eval}^{-1}$, čímž dostaneme

$$C_{gradq} = \begin{pmatrix} 1+q & 3q & 2+3q & 1+2q \\ 0 & 1+2q & 2q & q \\ 0 & 0 & 1+2q & q \\ 0 & 0 & 0 & 2 \end{pmatrix}. \tag{3.36}$$

Podle (2.10), (2.11) a (2.14) pro poměr časů platí

$$\begin{aligned} \omega_{gradq} \quad = \quad & \max\left\{ \frac{(1+q)\mu}{\mu}, \frac{(3+6q)\mu+1+2q}{3\mu+1}, \frac{(5+6q)\mu+2q+(1+2q)\pi}{3\mu+\pi}, \right. \\ & \left. \frac{(3+4q)\mu+q(1+\pi)+2\nu}{2\mu+\nu} \right\} \in [1+2q; 1.5+2.5q]. \end{aligned} \tag{3.37}$$

Odtud je zřetelné, že pokud ve vyhodnocení funkce $\boldsymbol{F}$ výrazně převažuje počet sčítání nad počtem násobení, trvá vyhodnocení q gradientů postupem $gradq$ zhruba $2q$-krát tak dlouho, jako vyhodnocení derivované funkce. V kapitole 1 jsme hovořili o tom, že vyhodnocení celého gradientu funkce f bude stát maximálně pětinásobek nákladů na f. Odvodili jsme hranice mnohem menší. Důvodem jsou různé možné předpoklady na časovou náročnost jednotlivých operací (vektor $\boldsymbol{w}$). Pokud jsou tyto předpoklady reálné, obvykle se pod pětinásobek dostaneme.

Pro úplnost ještě poznamenejme, že automatické derivování není metodou, která by našla úplně nejrychlejší způsob výpočtu derivace. Existují programy (resp. funkce), které lze ručně upravit, aby počítaly i svůj gradient; ve srovnání s programy získanými zpětným módem automatické derivace běží ručně upravené programy rychleji.

3.2.3 Iterační přiřazení (nejen) ve zpětném módu

Pro zobecnění tvaru vstupního soupisu operací a pro umožnění derivování již zderivovaných soupisů operací povolíme „iterační“ tvar základních funkcí a vysvětlíme, jak ho automatickým derivováním ztransformovat.

Předpokládejme tedy základní funkci $\boldsymbol{\varphi}_i : \mathbb{R}^{n_i} \to \mathbb{R}^{m_i}$ tvaru

$$\boldsymbol{v}_i = \boldsymbol{\varphi}_i(\boldsymbol{u}_i) = \boldsymbol{\varphi}_i(\boldsymbol{v}_i, \tilde{\boldsymbol{u}}_i),$$

která splňuje vlastnost 2.1 pro nějaké $d \geq 1$ (pokud počítáme derivaci stupně d). Její argument $\boldsymbol{u}_i$ rozdělme na složky

$$\boldsymbol{u}_i = (\boldsymbol{v}_i, \tilde{\boldsymbol{u}}_i), \quad \text{kde} \quad \tilde{\boldsymbol{u}}_i = (\boldsymbol{v}_j)_{i \neq j \prec i}.$$

Abychom odvodili její přidruženou operaci, odstraníme iterační charakter přepsáním na dvojici základních operací

$$\begin{aligned} \boldsymbol{w}_i &= \boldsymbol{v}_i \\ \boldsymbol{v}_i &= \boldsymbol{\varphi}_i(\boldsymbol{w}_i, \tilde{\boldsymbol{u}}_i). \end{aligned}$$

Proměnná $\boldsymbol{w}_i$ je nová pomocná proměnná, která se mimo tyto dvě přiřazení nikde nepoužívá. Postupem odvozeným v kapitole 3.2 dostaneme jejich přidružené operace.

$$\begin{aligned} \bar{\boldsymbol{w}}_i &= \bar{\boldsymbol{w}}_i + \bar{\boldsymbol{v}}_i \cdot \boldsymbol{c}_{ii} \\ \bar{\boldsymbol{v}}_j &= \bar{\boldsymbol{v}}_j + \bar{\boldsymbol{v}}_i \cdot \boldsymbol{c}_{ij} \quad \text{pro } i \neq j \prec i \\ \bar{\boldsymbol{v}}_i &= 0 \\ \bar{\boldsymbol{v}}_i &= \bar{\boldsymbol{v}}_i + \bar{\boldsymbol{w}}_i \\ \bar{\boldsymbol{w}}_i &= 0 \end{aligned}$$

Proměnnou $\boldsymbol{w}_i$ snadno vyeliminujeme (nezapomeňme, že na začátku byla vynulovaná) a tak odvodíme, jaké operace se mají provádět ve druhé části přidruženého soupisu operací.

$$\begin{aligned} \boldsymbol{v}_i &\leftarrow \mathcal{S} \\ \bar{\boldsymbol{v}}_j &= \bar{\boldsymbol{v}}_j + \bar{\boldsymbol{v}}_i \cdot \boldsymbol{c}_{ij} \quad \text{pro } i \neq j \prec i \\ \bar{\boldsymbol{v}}_i &= \bar{\boldsymbol{v}}_i * \boldsymbol{c}_{ii}. \end{aligned}$$

Uvědomme si, že operace $\bar{\boldsymbol{v}}_i = \bar{\boldsymbol{v}}_i \cdot \boldsymbol{c}_{ii}$ musí být provedena jako poslední. Pokud by základní funkce $\boldsymbol{\varphi}_i$ nebyla iterační, místo operace $\bar{\boldsymbol{v}}_i = \bar{\boldsymbol{v}}_i \cdot \boldsymbol{c}_{ii}$ by se provedlo vynulování $\bar{\boldsymbol{v}}_i$.

V přímém módu lze totéž zobecnění a odvození provést naprosto analogickým způsobem, proto zde uvedeme pouze výsledek. Sdruženou operací k základní operaci $\boldsymbol{v}_i = \boldsymbol{\varphi}_i(\boldsymbol{v}_i, \tilde{\boldsymbol{u}}_i)$ je

$$\dot{\boldsymbol{v}}_i = \sum_{i \neq j \prec i} \frac{\partial \boldsymbol{\varphi}_i}{\partial \boldsymbol{v}_j} \cdot \dot{\boldsymbol{v}}_j + \frac{\partial \boldsymbol{\varphi}_i}{\partial \boldsymbol{v}_i} \cdot \dot{\boldsymbol{v}}_i = \sum_{j \prec i} \frac{\partial \boldsymbol{\varphi}_i}{\partial \boldsymbol{v}_j} \cdot \dot{\boldsymbol{v}}_j.$$

Nadále budeme automaticky předpokládat, že funkce $\boldsymbol{\varphi}_i(\boldsymbol{u}_i)$ nezávisí na $\boldsymbol{v}_i$, pokud to explicitně neřekneme, nebo pokud to nebude zřejmé. V každém případě už víme, jak s iteračním přiřazením naložit.

Kapitola 4

Výpočet druhé derivace

V této kapitole odvodíme transformaci programu, který počítá hodnoty funkce $\boldsymbol{F}$, aby počítal také druhé derivace této funkce. Zmíníme se i o složitosti získaného programu.

Pro odvození výpočtu druhé derivace použijeme již popsané metody pro výpočet první derivace. Před tím, než se pustíme do vlastního odvozování, ukážeme zajímavou vlastnost. Pokud na zadaný program (soupis operací) aplikujeme dvakrát za sebou zpětný mód automatického derivování, dostaneme program, který dává stejné výsledky jako původní program (soupis operací), na který jsme aplikovali nejdříve zpětný mód a na vzniklý soupis operací ještě přímý mód automatického derivování. Tím odstraníme nepříjemný a nepřehledný nárůst velikosti soupisu operací při opakované aplikaci zpětného módu automatického derivování.

Uvažujme tedy dvakrát spojitě diferencovatelnou funkci $\boldsymbol{F} : \mathbb{R}^n \to \mathbb{R}^m$. Pro zadaný vektor vah $\bar{\boldsymbol{y}} \in \mathbb{R}^m$ jsme provedli přidružený soupis operací získaný zpětným módem, takže jsme spočetli hodnoty

$$\begin{aligned} \boldsymbol{y} &= \boldsymbol{F}(\boldsymbol{x}) && (4.1)\\ \bar{\boldsymbol{x}} &= \bar{\boldsymbol{x}} + \bar{\boldsymbol{y}} \cdot \boldsymbol{F}'(\boldsymbol{x}). && (4.2) \end{aligned}$$

Abychom na tento výsledek aplikovali přímý, resp. zpětný mód automatického derivování, považujme za jeho vstupní proměnné hodnoty $(\boldsymbol{x}, \bar{\boldsymbol{y}})$, za jeho výstupní proměnné hodnoty $(\boldsymbol{y}, \bar{\boldsymbol{x}})$.

Chceme-li aplikovat přímý mód, derivujme ve směru $(\dot{\boldsymbol{x}}, \dot{\bar{\boldsymbol{y}}})$, kde $\dot{\boldsymbol{x}} \in \mathbb{R}^n$ je sloupcový vektor a $\dot{\bar{\boldsymbol{y}}} \in \mathbb{R}^m$ je řádkový vektor.[1] Podle definice derivace ve směru dostaneme

$$\begin{aligned} \dot{\boldsymbol{y}} &= \boldsymbol{F}'(\boldsymbol{x}) \cdot \dot{\boldsymbol{x}}\\ \dot{\bar{\boldsymbol{x}}} &= \dot{\bar{\boldsymbol{x}}} + \bar{\boldsymbol{y}} \cdot \boldsymbol{F}''(\boldsymbol{x}) \cdot \dot{\boldsymbol{x}} + \dot{\bar{\boldsymbol{y}}} \cdot \boldsymbol{F}'(\boldsymbol{x}), \end{aligned}$$

kde $\dot{\boldsymbol{y}} \in \mathbb{R}^n$ je sloupcový vektor a $\dot{\bar{\boldsymbol{x}}} \in \mathbb{R}^m$ řádkový vektor. Výraz $\bar{\boldsymbol{y}} \cdot \boldsymbol{F}''(\boldsymbol{x}) \cdot \dot{\boldsymbol{x}}$ chápejme jako

$$\bar{\boldsymbol{y}} \cdot \boldsymbol{F}''(\boldsymbol{x}) \cdot \dot{\boldsymbol{x}} = \frac{\partial}{\partial \alpha}\, \bar{\boldsymbol{y}} \cdot \boldsymbol{F}'(\boldsymbol{x} + \alpha \dot{\boldsymbol{x}}) \Big|_{\alpha=0} \in \mathbb{R}^n \qquad (4.3)$$

[1] Směr není vždy jen sloupcový vektor – pro správnou definici $\bar{\boldsymbol{x}}$ ve vztahu (4.2) je v tomto případě korektní řádkový vektor $\dot{\bar{\boldsymbol{y}}}$.

Chceme-li na (4.1) a (4.2) aplikovat ještě jednou zpětný mód automatického derivování, derivujme podle $(\bar{\boldsymbol{y}}, \boldsymbol{x})$ s vahami $(\tilde{\boldsymbol{y}}, \tilde{\bar{\boldsymbol{x}}})$, kde $\tilde{\boldsymbol{y}} \in \mathbb{R}^m$ je řádkový vektor a $\tilde{\bar{\boldsymbol{x}}} \in \mathbb{R}^n$ je sloupcový vektor.

Tím dostaneme[2]

$$\begin{aligned} \tilde{\bar{\boldsymbol{y}}} &= \tilde{\bar{\boldsymbol{y}}} + \boldsymbol{F}'(\boldsymbol{x}) \cdot \tilde{\bar{\boldsymbol{x}}} \\ \tilde{\boldsymbol{x}} &= \tilde{\boldsymbol{x}} + \bar{\boldsymbol{y}} \cdot \boldsymbol{F}''(\boldsymbol{x}) \cdot \tilde{\bar{\boldsymbol{x}}} + \tilde{\boldsymbol{y}} \cdot \boldsymbol{F}'(\boldsymbol{x}), \end{aligned}$$

kde $\tilde{\bar{\boldsymbol{y}}} \in \mathbb{R}^n$ je sloupcový vektor a $\tilde{\boldsymbol{x}} \in \mathbb{R}^m$ řádkový vektor.

Ihned vidíme, že

$$\dot{\boldsymbol{x}} = \dot{\bar{\boldsymbol{x}}},\ \dot{\tilde{\boldsymbol{y}}} = \dot{\tilde{\boldsymbol{y}}} \quad \Longrightarrow \quad \dot{\bar{\boldsymbol{y}}} = \dot{\boldsymbol{y}},\ \dot{\tilde{\boldsymbol{x}}} = \dot{\tilde{\boldsymbol{x}}},$$

neboli výsledky získané přímým a zpětným módem na výstupní soupis operací ze zpětného módu jsou shodné. Přitom jsme předpokládali, že proměnné $\dot{\bar{\boldsymbol{y}}}$, $\dot{\tilde{\boldsymbol{x}}}$ a $\dot{\tilde{\bar{\boldsymbol{x}}}}$ byly na počátku vynulovány, takže provedené přičítání bylo vlastně přiřazování.

4.1 Odvození a vlastnosti

Pro toto odvození zakážeme přepis již nepotřebných proměnných, později ho však opět umožníme.

Máme program, který počítá hodnotu funkce $\boldsymbol{F}$ v bodě $\boldsymbol{x}$. Odvodíme z něj program, který bude navíc počítat i hodnoty druhé derivace ve tvaru

$$\bar{\boldsymbol{y}} \cdot \boldsymbol{F}''(\boldsymbol{x}) \cdot \dot{\boldsymbol{x}} \in \mathbb{R}^n, \tag{4.4}$$

který chápeme stejně jako v (4.3). Tuto hodnotu spočteme jako derivaci výrazu $\bar{\boldsymbol{x}} = \bar{\boldsymbol{y}} \cdot \boldsymbol{F}'(\boldsymbol{x})$ ve směru $\dot{\boldsymbol{x}}$. Jinak řečeno, zpětným módem automatického derivování získáme hodnotu $\bar{\boldsymbol{x}} = \bar{\boldsymbol{y}} \cdot \boldsymbol{F}'(\boldsymbol{x})$ a na ni budeme aplikovat přímý mód. Obecně tedy vlastně spočteme

$$\bar{\boldsymbol{y}} \cdot \boldsymbol{F}''(\boldsymbol{x}) \cdot \dot{\boldsymbol{x}} + \dot{\bar{\boldsymbol{y}}} \cdot \boldsymbol{F}'(\boldsymbol{x}).$$

Pokud bychom na začátku výpočtu přiřadili $\dot{\bar{\boldsymbol{y}}} = 0 \in \mathbb{R}^m$, člen $\dot{\bar{\boldsymbol{y}}} \cdot \boldsymbol{F}'(\boldsymbol{x})$ by byl nulový a dostali bychom tak žádaný výsledek (4.4).

Využijeme sčítaný zpětný mód automatického derivování na obr. 3.4. Výraz

$$\bar{\boldsymbol{v}}_j = \sum_{i \succ j} \bar{\boldsymbol{v}}_i \cdot \frac{\partial \boldsymbol{\varphi}_i}{\partial \boldsymbol{v}_j}(\boldsymbol{u}_i) = \sum_{i \succ j} \bar{\boldsymbol{v}}_i \cdot \boldsymbol{c}_{ij}$$

zderivujeme pro každé $j = 1-n, \ldots, l-m$, čímž dostaneme

$$\dot{\bar{\boldsymbol{v}}}_j = \sum_{i \succ j} \left(\dot{\bar{\boldsymbol{v}}}_i \cdot \frac{\partial \boldsymbol{\varphi}_i}{\partial \boldsymbol{v}_j} + \bar{\boldsymbol{v}}_i \cdot \frac{\partial \dot{\boldsymbol{\varphi}}_i}{\partial \boldsymbol{v}_j} \right),$$

[2] Návod: Počítejme derivace

$$\frac{\partial}{\partial \bar{\boldsymbol{y}}}(\tilde{\boldsymbol{y}}\boldsymbol{y} + \bar{\boldsymbol{x}}\tilde{\bar{\boldsymbol{x}}}) \quad \text{a} \quad \frac{\partial}{\partial \boldsymbol{x}}(\tilde{\boldsymbol{y}}\boldsymbol{y} + \bar{\boldsymbol{x}}\tilde{\bar{\boldsymbol{x}}}.)$$

kde

$$\frac{\partial \boldsymbol{\varphi}_i}{\partial \boldsymbol{v}_j} = \frac{\partial \boldsymbol{\varphi}_i}{\partial \boldsymbol{v}_j}(\boldsymbol{u}_i) = \boldsymbol{c}_{ij}$$

a

$$\frac{\partial \dot{\boldsymbol{\varphi}}_i}{\partial \boldsymbol{v}_j} = \sum_{k \prec i} \frac{\partial^2 \boldsymbol{\varphi}_i}{\partial \boldsymbol{v}_j \partial \boldsymbol{v}_k}(\boldsymbol{u}_i) \cdot \dot{\boldsymbol{v}}_k.$$

Ve shodě s označením v kapitole 3 označíme

$$\boldsymbol{c}_{ijk} = \frac{\partial^2 \boldsymbol{\varphi}_i}{\partial \boldsymbol{v}_j \partial \boldsymbol{v}_k}(\boldsymbol{u}_i)$$

a také

$$\dot{\boldsymbol{u}}_i = (\dot{\boldsymbol{v}}_j)_{j \prec i}.$$

Pokud jsou proměnné $\dot{\boldsymbol{v}}_i$ pro $i = 1, \ldots, l-m$ předem vynulovány, můžeme jejich hodnoty počítat postupným přičítáním jednotlivých členů, tak jako se děje pro $\bar{\boldsymbol{v}}_i$ při nasčítávaném zpětném módu automatického derivování na obr. 3.3 (ve srovnání se sčítaným zpětným módem na obr 3.4). Proto zavedeme vektorovou funkci $\dot{\bar{\boldsymbol{\varphi}}}_i$, která bude vyhodnocovat jednotlivé příspěvky do $\dot{\bar{\boldsymbol{u}}}_i$ pro $i = 1, \ldots, l$

$$\begin{aligned}
\dot{\bar{\boldsymbol{\varphi}}}_i(\boldsymbol{u}_i, \dot{\boldsymbol{u}}_i, \bar{\boldsymbol{v}}_i, \dot{\bar{\boldsymbol{v}}}_i) &= \bar{\boldsymbol{\varphi}}_i(\boldsymbol{u}_i, \dot{\bar{\boldsymbol{v}}}_i) + \bar{\boldsymbol{v}}_i \cdot \left(\frac{\partial \dot{\boldsymbol{\varphi}}_i}{\partial \boldsymbol{v}_j}\right)_{j \prec i} \\
&= \bar{\boldsymbol{\varphi}}_i(\boldsymbol{u}_i, \dot{\bar{\boldsymbol{v}}}_i) + \bar{\boldsymbol{v}}_i \cdot \left(\sum_{k \prec i} \boldsymbol{c}_{ijk} \cdot \dot{\boldsymbol{v}}_k\right)_{j \prec i}
\end{aligned}$$

Všimněme si, že druhý parametr funkce $\bar{\boldsymbol{\varphi}}_i$ je $\dot{\bar{\boldsymbol{v}}}_i$ – v nasčítávaném zpětném módu je tento parametr $\bar{\boldsymbol{v}}_i$. Nyní povolíme přepis již nepotřebných proměnných[3], takže budeme používat zásobník $\mathcal{S}$ na uschování hodnot do druhé části soupisu operací. Tak dostaneme obecný postup pro výpočet hodnot $\bar{\boldsymbol{y}} \cdot \boldsymbol{F}''(\boldsymbol{x}) \cdot \dot{\boldsymbol{x}}$. Je uveden na obrázku 4.1.

Tento soupis operací jsme také mohli odvodit aplikací přímého módu na soupis operací získaný nasčítávaným zpětným módem automatického derivování, pokud by ovšem přímý mód byl schopen pracovat se zásobníkovými operacemi. Všimněme si, že první část odvozeného soupisu operací je až na odkládání dat na zásobník stejná jako soupis operací získaný přímým módem automatického derivování. Dále, vstupní hodnoty vah $\bar{\boldsymbol{y}}_i$ a $\dot{\bar{\boldsymbol{y}}}_i$ (pro $i = 1, \ldots, m$) potřebujeme znát až na počátku druhé části odvozeného soupisu operací.

Analogicky jako při výpočtu prvních derivací označme funkci

$$\dot{\bar{\boldsymbol{F}}}(\boldsymbol{x}, \dot{\boldsymbol{x}}, \bar{\boldsymbol{y}}, \dot{\bar{\boldsymbol{y}}}) = \bar{\boldsymbol{y}} \cdot \boldsymbol{F}''(\boldsymbol{x}) \cdot \dot{\boldsymbol{x}} + \dot{\bar{\boldsymbol{y}}} \cdot \boldsymbol{F}'(\boldsymbol{x}).$$

Provedením soupisu operací na obrázku 4.1 tak vlastně spočítáme

$$\dot{\boldsymbol{x}} = \dot{\boldsymbol{x}} + \dot{\bar{\boldsymbol{F}}}(\boldsymbol{x}, \dot{\boldsymbol{x}}, \bar{\boldsymbol{y}}, \dot{\bar{\boldsymbol{y}}}).$$

[3]Podmínka přímé kompatibility nechť je ale splněna.

$[\bar{\boldsymbol{v}}_i, \dot{\bar{\boldsymbol{v}}}_i]$	$= [0,0]$	$\}$	$i = 1, \ldots, l$
$[\boldsymbol{v}_{i-n}, \dot{\boldsymbol{v}}_{i-n}]$	$= [\boldsymbol{x}_i, \dot{\boldsymbol{x}}_i]$	$\Big\}$	$i = 1, \ldots, n$
$[\bar{\boldsymbol{v}}_{i-n}, \dot{\bar{\boldsymbol{v}}}_{i-n}]$	$= [\bar{\boldsymbol{x}}_i, \dot{\bar{\boldsymbol{x}}}_i]$		
$[\boldsymbol{v}_i, \dot{\boldsymbol{v}}_i]$	$\to \mathcal{S}$	$\Big\}$	$i = 1, \ldots, l$
$[\boldsymbol{v}_i, \dot{\boldsymbol{v}}_i]$	$= [\boldsymbol{\varphi}_i(\boldsymbol{u}_i), \dot{\boldsymbol{\varphi}}_i(\boldsymbol{u}_i, \dot{\boldsymbol{u}}_i)]$		
$[\boldsymbol{y}_{m-i}, \dot{\boldsymbol{y}}_{m-i}]$	$= [\boldsymbol{v}_{l-i}, \dot{\boldsymbol{v}}_{l-i}]$	$\}$	$i = m-1, \ldots, 0$
$[\bar{\boldsymbol{v}}_{l-i}, \dot{\bar{\boldsymbol{v}}}_{l-i}]$	$= [\bar{\boldsymbol{y}}_{m-i}, \dot{\bar{\boldsymbol{y}}}_{m-i}]$	$\}$	$i = 0, \ldots, m-1$
$[\boldsymbol{v}_i, \dot{\boldsymbol{v}}_i]$	$\leftarrow \mathcal{S}$	$\Big\}$	$i = l, \ldots, 1$
$[\bar{\boldsymbol{u}}_i, \dot{\bar{\boldsymbol{u}}}_i]$	$= [\bar{\boldsymbol{u}}_i, \dot{\bar{\boldsymbol{u}}}_i] + [\bar{\boldsymbol{\varphi}}_i(\boldsymbol{u}_i, \bar{\boldsymbol{v}}_i), \dot{\bar{\boldsymbol{\varphi}}}_i(\boldsymbol{u}_i, \dot{\boldsymbol{u}}_i, \bar{\boldsymbol{v}}_i, \dot{\bar{\boldsymbol{v}}}_i)]$		
$[\bar{\boldsymbol{v}}_i, \dot{\bar{\boldsymbol{v}}}_i]$	$= [0,0]$		
$[\bar{\boldsymbol{x}}_i, \dot{\bar{\boldsymbol{x}}}_i]$	$= [\bar{\boldsymbol{v}}_{i-n}, \dot{\bar{\boldsymbol{v}}}_{i-n}]$	$\}$	$i = n, \ldots, 1$

Obrázek 4.1: Soupis operací odvozený zpětným a následně přímým módem, obecný tvar

Zabývejme se otázkou, jakým způsobem alokovat proměnné $\dot{\boldsymbol{v}}_i$, $\bar{\boldsymbol{v}}_i$ a $\dot{\bar{\boldsymbol{v}}}_i$, aby výpočet byl korektní. Odvozený soupis operací je vlastně kombinací soupisů operací z přímého a zpětného módu, takže se můžeme nechat inspirovat jejich způsobem alokace. Použijme tedy přístup

$$\&\boldsymbol{v}_i = \&\boldsymbol{v}_j \quad \Longleftrightarrow \quad \&\dot{\boldsymbol{v}}_i = \&\dot{\boldsymbol{v}}_j, \quad \&\bar{\boldsymbol{v}}_i = \&\bar{\boldsymbol{v}}_j, \quad \&\dot{\bar{\boldsymbol{v}}}_i = \&\dot{\bar{\boldsymbol{v}}}_j.$$

Toto je speciálně splněno pro

$$\&\dot{\boldsymbol{v}}_i = \&\boldsymbol{v}_i + c, \quad \&\bar{\boldsymbol{v}}_i = \&\boldsymbol{v}_i + \tilde{c}, \quad \&\dot{\bar{\boldsymbol{v}}}_i = \&\boldsymbol{v}_i + \tilde{\tilde{c}}$$

za předpokladu, že se bloky paměti pro všechny původní $\boldsymbol{v}_i$, sdružené $\dot{\boldsymbol{v}}_i$, přidružené $\bar{\boldsymbol{v}}_i$ a sdružené přidružené $\dot{\bar{\boldsymbol{v}}}_i$ vzájemně nepřekrývají.

Stejně jako při výpočtu první derivace, i zde můžeme snížit náklady na výpočet, pokud požadujeme derivace podle více směrů $\dot{\boldsymbol{x}}$. Nahradíme-li vektor $\dot{\boldsymbol{x}}$ maticí $\dot{\boldsymbol{X}} \in \mathbb{R}^{n \times p}$ s jednotlivými směrovými vektory, budeme počítat matici

$$\dot{\bar{\boldsymbol{X}}} = \bar{\boldsymbol{y}} \cdot \boldsymbol{F}''(\boldsymbol{x}) \cdot \dot{\boldsymbol{X}} \in \mathbb{R}^{n \times p}.$$

Je zřejmé, jak bychom pro tuto situaci upravili soupis operací. Uvědomme si jen, že výpočet s maticí $\dot{\boldsymbol{X}}$ nemusí pro velmi malá p nutně znamenat zrychlení výpočtu, protože musíme navíc provádět cykly přes všechny sloupce, které výpočet zpomalují. Analogická poznámka platí i pro úkoly *tangp* a *gradq*.

4.2 Složitost odvozeného programu

Úkol výpočtu čtveřice

$$[\boldsymbol{y}, \dot{\boldsymbol{y}}, \bar{\boldsymbol{x}}, \dot{\bar{\boldsymbol{x}}}] = [\boldsymbol{F}(\boldsymbol{x}), \dot{\boldsymbol{F}}(\boldsymbol{x}, \dot{\boldsymbol{x}}), \bar{\boldsymbol{F}}(\boldsymbol{x}, \bar{\boldsymbol{y}}), \dot{\bar{\boldsymbol{F}}}(\boldsymbol{x}, \dot{\boldsymbol{x}}, \bar{\boldsymbol{y}}, \dot{\bar{\boldsymbol{y}}})]$$

označíme jako $soad(\boldsymbol{F})$ (second order adjoint). Je to opět aditivní úkol. Jeho provedení vyžaduje po každé základní funkci $\boldsymbol{\varphi}_i(\boldsymbol{v}_i)$ vyhodnocení páru $[\boldsymbol{\varphi}_i(\boldsymbol{u}_i), \dot{\boldsymbol{\varphi}}_i(\boldsymbol{u}_i, \dot{\boldsymbol{u}}_i)]$ a přiřazení této hodnoty do $[\boldsymbol{v}_i, \dot{\boldsymbol{v}}_i]$ v první části soupisu operací, dále vyhodnocení $[\bar{\boldsymbol{\varphi}}_i(\boldsymbol{u}_i, \bar{\boldsymbol{v}}_i), \dot{\bar{\boldsymbol{\varphi}}}_i(\boldsymbol{u}_i, \dot{\boldsymbol{u}}_i, \bar{\boldsymbol{v}}_i, \dot{\bar{\boldsymbol{v}}}_i)]$ a přičtení této hodnoty k $[\bar{\boldsymbol{u}}_i, \dot{\bar{\boldsymbol{u}}}_i]$ ve druhé části soupisu operací, uložení hodnot $[\boldsymbol{v}_i, \dot{\boldsymbol{v}}_i]$ na zásobník a jejich vyzvednutí ze zásobníku a také vynulování $[\bar{\boldsymbol{v}}_i, \dot{\bar{\boldsymbol{v}}}_i]$. Tyto úkoly budeme dohromady označovat jako $soad'(\boldsymbol{\varphi})$, takže

$$\begin{aligned} WORK\{soad'(\boldsymbol{\varphi}_i)\} \;=\; & WORK\{\boldsymbol{v}_i = \boldsymbol{\varphi}_i(\boldsymbol{u}_i)\} + WORK\{\dot{\boldsymbol{v}}_i = \dot{\boldsymbol{\varphi}}_i(\boldsymbol{u}_i, \dot{\boldsymbol{u}}_i)\} + \\ & WORK\{\bar{\boldsymbol{u}}_i = \bar{\boldsymbol{u}}_i + \bar{\boldsymbol{\varphi}}_i(\boldsymbol{u}_i, \bar{\boldsymbol{v}}_i)\} + \\ & WORK\{\dot{\bar{\boldsymbol{u}}}_i = \dot{\bar{\boldsymbol{u}}}_i + \dot{\bar{\boldsymbol{\varphi}}}_i(\boldsymbol{u}_i, \dot{\boldsymbol{u}}_i, \bar{\boldsymbol{v}}_i, \dot{\bar{\boldsymbol{v}}}_i)\} \\ & WORK\{[\boldsymbol{v}_i, \dot{\boldsymbol{v}}_i] \rightarrow \mathcal{S}\} + WORK\{[\boldsymbol{v}_i, \dot{\boldsymbol{v}}_i] \leftarrow \mathcal{S}\} + \\ & WORK\{\bar{\boldsymbol{v}}_i = 0\} + WORK\{\dot{\bar{\boldsymbol{v}}}_i = 0\}. \end{aligned} \tag{4.5}$$

Výraz $WORK\{[\boldsymbol{v}_i, \dot{\boldsymbol{v}}_i] \rightarrow \mathcal{S}\} + WORK\{[\boldsymbol{v}_i, \dot{\boldsymbol{v}}_i] \leftarrow \mathcal{S}\}$ je nulový, pokud nepřepisujeme proměnné. Pro jednoduchost našich úvah ho nebudeme v dalších odhadech složitosti uvažovat, i když se tím dopustíme určité nepřesnosti.

Na úkol $soad(\boldsymbol{F})$ můžeme za splnění vlastností 2.4 a 2.5 aplikovat větu 2.4, která zaručí existenci matice C_{soad}, že pro všechna $\boldsymbol{F} \in \mathcal{F}$

$$WORK\{soad(\boldsymbol{F})\} \leq C_{soad} \cdot WORK\{eval(\boldsymbol{F})\}.$$

Dále existuje $\omega_{soad} \in \mathbb{R}$, že pro každé $\boldsymbol{F} \in \mathcal{F}$

$$TIME\{soad(\boldsymbol{F})\} \leq \omega_{soad} TIME\{eval(\boldsymbol{F})\}.$$

Abychom získali jejich hodnotu, uvažujme opět čtyřsložkový vektor složitosti představený v kapitole 2.4 a operace c (přiřazení konstanty), $+$, $*$ a ψ (nelineární funkce).

V tabulce 4.1 jsou uvedeny tyto základní operace $\boldsymbol{\varphi}$ spolu s příslušnými sdruženými operacemi $\dot{\boldsymbol{\varphi}}$, přidruženými operacemi $\bar{\boldsymbol{\varphi}}$ a s operacemi $\dot{\bar{\boldsymbol{\varphi}}}$ zavedenými v této kapitole. Dále uvádíme složitost provedení úkolu $soad'(\boldsymbol{\varphi})$ na základní operace $\boldsymbol{\varphi}$ s tím, že operace na jednom řádku v obrázku 4.1 (například $\boldsymbol{\varphi}_i$, $\dot{\boldsymbol{\varphi}}_i$) chápeme jako společně vyhodnocované operace. Takže získání stejné hodnoty proměnné z paměti, které by prováděly obě funkce, započítáme pouze jednou.

Zjištěné hodnoty jako obvykle zapíšeme do matice $W_{soad'}$.

$$W_{soad'} = \begin{pmatrix} 4 & 18 & 22 & 14 \\ 0 & 6 & 7 & 3 \\ 0 & 0 & 9 & 5 \\ 0 & 0 & 0 & 3 \end{pmatrix}$$

Matici C_{soad} snadno určíme jako $W_{soad'} \cdot W_{eval}^{-1}$, kde W_{eval} je definovaná vztahem (2.6). Proto

$$C_{soad} = \begin{pmatrix} 4 & 6 & 10 & 6 \\ 0 & 6 & 7 & 3 \\ 0 & 0 & 9 & 5 \\ 0 & 0 & 0 & 3 \end{pmatrix}.$$

$\boldsymbol{\varphi}$	$\dot{\boldsymbol{\varphi}}$	složitost
$\bar{\boldsymbol{\varphi}}$	$\dot{\bar{\boldsymbol{\varphi}}}$	pro $soad'(\boldsymbol{\varphi})$
$v = c$	$\dot{v} = 0$	$(4,0,0,0)$
$\bar{v} = 0$	$\dot{\bar{v}} = 0$	
$v = u + w$	$\dot{v} = \dot{u} + \dot{w}$	$(18,6,0,0)$
$\bar{u} = \bar{u} + \bar{v}$	$\dot{\bar{u}} = \dot{\bar{u}} + \dot{\bar{v}}$	
$\bar{w} = \bar{w} + \bar{v}$	$\dot{\bar{w}} = \dot{\bar{w}} + \dot{\bar{v}}$	
$\bar{v} = 0$	$\dot{\bar{v}} = 0$	
$v = u * w$	$\dot{v} = \dot{u} * w + u * \dot{w}$	$(22,7,9,0)$
$\bar{u} = \bar{u} + \bar{v} * w$	$\dot{\bar{u}} = \dot{\bar{u}} + \dot{\bar{v}} * w + \bar{v} * \dot{w}$	
$\bar{w} = \bar{w} + \bar{v} * u$	$\dot{\bar{w}} = \dot{\bar{w}} + \dot{\bar{v}} * u + \bar{v} * \dot{u}$	
$\bar{v} = 0$	$\dot{\bar{v}} = 0$	
$v = \psi(u)$	$\dot{v} = \psi'(u) * \dot{u}$	$(14,3,5,3)$
$\bar{u} = \bar{u} + \bar{v} * \psi'(u)$	$\dot{\bar{u}} = \dot{\bar{u}} + \dot{\bar{v}} * \psi'(u) + \bar{v} * \psi''(u) * \dot{u}$	
$\bar{v} = 0$	$\dot{\bar{v}} = 0$	

Tabulka 4.1: Některé základní funkce $\boldsymbol{\varphi}$ a jejich příslušné $\dot{\boldsymbol{\varphi}}$, $\bar{\boldsymbol{\varphi}}$, $\dot{\bar{\boldsymbol{\varphi}}}$ a složitost úkolu $soad'(\boldsymbol{\varphi})$ (nepřesně řečeno – pro upřesnění viz text)

Předpokládejme, že doby trvání operace s pamětí, sečtení (resp. vynásobení) dvou čísel a provedení nelineární operace je dána vektorem $\boldsymbol{w}$ ve vztahu (2.10). Předpokládáme-li dále (2.11), s pomocí (2.14) dostáváme

$$\omega_{soad} = \max\left\{\frac{4\mu}{\mu}, \frac{18\mu+6}{3\mu+1}, \frac{22\mu+7+9\pi}{3\mu+\pi}, \frac{14\mu+3+5\pi+3\nu}{2\mu+\nu}\right\} \in [7;10].$$

Úkol, ve kterém budeme vyhodnocovat matici $\dot{\bar{\boldsymbol{X}}} = \dot{\bar{\boldsymbol{X}}} + \bar{\boldsymbol{y}} \cdot \boldsymbol{F}''(\boldsymbol{x}) \cdot \dot{\boldsymbol{X}} \in \mathbb{R}^{n\times p}$ pro $\dot{\boldsymbol{X}} \in \mathbb{R}^{n\times p}$ (p směrů najednou), označíme $soadp(\boldsymbol{F})$, resp. $soadp'(\boldsymbol{\varphi})$ pro analogii s (4.5) pro základní funkci $\boldsymbol{\varphi}$. Podobnými úvahami jako výše určíme náklady pro operace přiřazení konstanty (c), binární sčítání, binární násobení a nelineární funkci (ψ) pro provedení úkolu $soadp'$. Tak dostaneme tabulku 4.2.

$soadp'$	c	$+$	$*$	ψ
$MOVES$	$2+2p$	$9+9p$	$11+11p$	$7+7p$
$ADDS$	0	$3+3p$	$2+5p$	$1+2p$
$MULTS$	0	0	$3+6p$	$2+3p$
$NLOPS$	0	0	0	3

Tabulka 4.2: Náklady pro vyhodnocení některých základních funkcí, úkol $soadp'$

Přepsáním vznikne matice $W_{soadp'}$.

$$W_{soadp'} = \begin{pmatrix} 2+2p & 9+9p & 11+11p & 7+7p \\ 0 & 3+3p & 2+5p & 1+2p \\ 0 & 0 & 3+6p & 2+3p \\ 0 & 0 & 0 & 3 \end{pmatrix}$$

Matici C_{soadp} můžeme spočíst jako $W_{soadp'} \cdot W_{eval}^{-1}$. Tak dostaneme

$$C_{soadp} = \begin{pmatrix} 2+2p & 3+3p & 5+5p & 3+3p \\ 0 & 3+3p & 2+5p & 1+2p \\ 0 & 0 & 3+6p & 2+3p \\ 0 & 0 & 0 & 3 \end{pmatrix}.$$

Podle vztahů (2.10), (2.11) a (2.14) pro poměr časů platí

$$\begin{aligned} \omega_{soadp} \quad = \quad \max \Bigg\{ & \frac{(2+2p)\mu}{\mu}, \frac{(9+9p)\mu + (3+3p)}{3\mu+1}, \\ & \frac{(11+11p)\mu + (2+5p) + (3+6p)\pi}{3\mu+\pi}, \\ & \frac{(7+7p)\mu + (1+2p) + (2+3p)\pi + 3\nu}{2\mu+\nu} \Bigg\} \in [3+3p; 6+6p]. \end{aligned}$$

Zdůrazněme, že metodou odvozenou v této kapitole spočítáme například vektor derivací

$$\left(\frac{\partial^2 \boldsymbol{F}}{\partial \boldsymbol{x}_1^2}(\boldsymbol{x}), \ldots, \frac{\partial^2 \boldsymbol{F}}{\partial \boldsymbol{x}_1 \partial \boldsymbol{x}_n}(\boldsymbol{x}) \right)^T \in \mathbb{R}^n$$

za nejvýše desetinásobek času pro vyhodnocení původní funkce $\boldsymbol{F}(\boldsymbol{x})$. Vyhodnocení celé Hessovy matice však bude stát nejvýše zhruba $6n$-krát více než vyhodnocení původní funkce.

Kapitola 5

Způsoby implementace

V předcházejících kapitolách jsme popsali některé metody automatického derivování. V této kapitole prakticky ukážeme, jak je aplikovat na zadaný program, aby počítal i požadované derivace. Stručně řečeno, popíšeme implementaci uvedených metod.

Tato kapitola si neklade za cíl (a ani nemůže) být popisem implementace automatického derivování do posledního detailu. Spíše ukážeme základní techniky a praktické postřehy. Konkrétní implementace je ostatně závislá na použitém systému. Na základě zde uvedených principů je snadné ji připravit.

V praktických případech máme zadaný *program* (který obsahuje cykly, podmínky, volání podprogramů ad.), který chceme ztransformovat do jiného programu. Implementace metod automatického derivování je tedy vlastně *transformace programu*, který vyhodnocuje hodnoty funkce $\boldsymbol{F}$, na program, který počítá i její derivace. Při této transformaci nadefinujeme (naalokujeme) nové proměnné, přidáme numerické operace atd. Popsali jsme transformaci soupisu operací, ale vstupem je program. Tento rozpor snadno odstraníme vysvětlením principů transformace a nepředstavuje tedy žádný problém.

Dva základní přístupy, jak *automaticky* přejít od zadaného k odvozenému programu, jsou buď *přetížení operátorů a funkcí* nebo použití *preprocesoru*. Předcházející tvrzení upřesníme a o každé z těchto metod se zmíníme v následujících kapitolách.

Ať už použijeme přetížení operátorů a funkcí nebo preprocesor, vždy bude potřeba upravit program pro výpočet funkce $\boldsymbol{F}$. Toto lze udělat buď ručně, nebo za použití nějakého vhodného programu. V tom případě je nutné předem říci, jaké jsou nezávislé a závislé proměnné, jaké derivace požadujeme a případně také mód automatického derivování, který se má použít. Tyto informace je vhodné dodat v externím (parametrickém) souboru.

Kromě transformace zadaného programu je nutné (při přetěžování i prekompilaci) připravit soubor (balík) pomocných procedur a funkcí, které budou využívány přetransformovaným programem. Jedná se například o deklarace nových typů proměnných, jejich inicializace, pozměněné definice základních funkcí (aby například obsahovaly i jejich sdružené operace) atd. Podrobnosti ukáže další výklad.

Transformace vstupního programu může samozřejmě být provedena „v ruce“, tj. jeho ruční editací (modifikací). Takový postup má své výhody i nevýhody. Je sice pracný, ale můžeme využít znalosti struktury derivovaného programu. Dosažené výsledky tedy mohou být mnohem lepší než při slepém použití nějakého programu, který by transformaci provedl. Při ruční přípravě máme snahu o co nejmenší změny, které musíme do vstupního programu zanést. Použití přetížení operátorů a funkcí tento požadavek do značné míry splní.

Spolu s transformací zadaného programu je nutné provést i transformaci používaných podprogramů. Z níže uvedeného vyplyne, jakým způsobem.

Načtení nezávislých proměnných $\boldsymbol{x}$, směrů $\dot{\boldsymbol{x}}$ (resp. vah $\bar{\boldsymbol{y}}$) a výstup závislých proměnných $\boldsymbol{y}$ a jejich derivací se řídí konkrétní situací a systémem. Těmito otázkami se zde proto nebudeme zabývat.

Poznamenejme ještě, že zmiňovaná transformace může proběhnout bez znalostí globálních vlastností zadaného programu. Jinak řečeno, lokální znalosti jsou dostatečné. Je ale zřejmé, že znalost globálních vlastností by byla užitečná pro zefektivnění odvozeného programu, například díky lepšímu opakovanému využívání již jednou spočtených hodnot či kvůli optimalizaci běhu (toku) programu.

Dále popíšeme nejen podobu odvozených programů, ale zejména postup, jakým odvozený program z původního programu získat. Popíšeme tedy, jak má automatické derivování vlastně pracovat. Budeme se držet spíše na obecnější rovině, konkrétní podmínky jsou totiž závislé na systému, ve kterém automatické derivování implementujeme.

5.1 Přetížení operátorů a funkcí

Technika přetížení operátorů (například $+$, $*$ nebo $=$, operator overloading) nebo funkcí (například funkcí definovaných uživatelem, procedure overloading) je ve srovnání s použitím preprocesoru jednodušší a přehlednější. Bývá podporována moderními programovacími jazyky, například C++, Adou nebo Fortranem 90. Je založena na jejich schopnosti rozlišit při volání operátorů (funkcí) *počet* a *typ* parametrů a díky tomu vybrat mezi operátory (funkcemi) se stejným jménem tu definici operátoru (funkce), která odpovídá zadaným parametrům. Například, kdybychom měli dva různé typy proměnných `typA` a `typB` a dvě definice podprogramu `PP` (jednu pro případ, kdy vstupní parametr je typu `typA`, a druhou pro případ, kdy vstupní parametr je typu `typB`), při volání `PP(ProměnnáTypuA)`, resp. `PP(ProměnnáTypuB)` překladač podle typu parametru rozliší, jakou definici procedury `PP` má zavolat.

V následujících odstavcích předvedeme jednoduchý způsob implementace automatického derivování pomocí přetížení operátorů a funkcí v programovacím jazyce C++. Všechny ukázky jsou jednoduché a navíc je okomentujeme, takže je lze v případě potřeby snadno převést do jiného programovacího jazyka. Pokud se chce čtenář více poučit o jazyce C++, mohu doporučit například [3].

5.1.1 Přímý mód

Popíšeme jednoduchý postup, jak pomocí přetížení operátorů a funkcí implementovat přímý mód automatického derivování, který jsme odvodili v kapitole 3.1. Tímto způsobem vyhodnotíme derivaci v zadaném směru.

Zavedeme nový typ proměnné, který označíme `doublet`. Tato struktura (třída) obsahuje dvě proměnné typu `double` (reálné číslo[1]). Tyto dvě složky slouží pro uložení hodnot proměnné $\boldsymbol{v}_i$ a její sdružené proměnné $\dot{\boldsymbol{v}}_i$.[2] Neboli

```
class doublet
{
 public:
   double v;
   double vdot;
};
```

Nyní můžeme definovat nové funkce, jejichž vstupní nebo výstupní hodnoty jsou typu `doublet`. Díky tomu přetížíme elementární matematické funkce. Budou provádět nejen obvyklé vyčíslení své hodnoty, ale také spočtení příslušné sdružené operace. Například funkci sinus můžeme přetížit následujícím způsobem.

```
doublet sin (doublet a)
{
 doublet b;
 b.v = sin (a.v);
 b.vdot = cos (a.v) * a.vdot;
 return b;
};
```

Pokud proměnné `x` a `y` jsou typu `doublet` a zavoláme funkci `y = sin(x)`, pak se tímto provede nejen přiřazení `y.v = sin (x.v)`, ale i příslušná sdružená operace `y.vdot = cos (x.v) * x.vdot`. Pokud bychom zavolali funkci `sin` na proměnnou typu `double`, provedla by se funkce sinus, jak je definovaná výrobcem překladače. Výsledkem by byla proměnná typu `double`.

Analogicky přetížíme operátor `*`.

```
doublet operator* (doublet a, doublet b)
{
  doublet c;
  c.v = a.v * b.v;
  c.vdot = a.vdot * b.v + a.v * b.vdot;
  return c;
};
```

Pokud proměnné `x`, `y` a `z` jsou typu `doublet`, pak výraz `z = x * y` způsobí nejen spočtení součinu, ale také provedení příslušné sdružené operace. Dále je pro

[1] Přesněji řečeno, aproximace reálného čísla.

[2] V celé kapitole 5 budeme pro jednoduchost předpokládat, že všechny $\boldsymbol{v}_i$ jsou skaláry.

operátor * (a vůbec binární operátory) potřeba nadefinovat smíšené operace pro konstanty (například typu `double`)[3] a `doublet`. Podobným způsobem přetížíme další aritmetické operace a používané funkce. Poznamenejme ještě, že přetížené operátory a funkce mají v jazyce C++ stejnou prioritu jako jejich nepřetížení jmenovci.

Zmíněné definice typů proměnných a funkcí budou uvedeny v souboru (balíku) s pomocnými definicemi a funkcemi, nikoli v hlavním programu. Dále musíme upravit zadaný program, aby pracoval s proměnnými typu `doublet` a zajistit jejich korektní vstup a výstup.

Zamysleme se, co všechno musí balík s pomocnými definicemi a funkcemi vlastně obsahovat.

- Zavedení nového typu proměnné (nazvali jsme ho `doublet`), který bude obsahovat nejen hodnotu $\boldsymbol{v}_i$, ale i její derivaci $\dot{\boldsymbol{v}}_i$.

- Přetížení obvyklých operátorů (funkcí) prováděných na běžné typy proměnných, aby pracovaly i s parametry typu `doublet`. Tyto nové operace budou navíc provádět i sdružené operace podle pravidla řetězení s využitím složky `vdot`, kterou chápeme jako $\dot{\boldsymbol{v}}_i$.

- Zajištění správné inicializace obou složek proměnné typu `doublet` a naopak vyexportování hodnoty `v` a její derivace `vdot`.

- Případně také načtení a výstup proměnných typu `doublet`. Tyto činnosti může ale také dělat odvozený program bez pomocných procedur.

Je jasné, že musíme změnit i derivovaný program. Logicky se snažíme o to, aby nutných změn bylo co nejméně. Nevyhnutelně však musíme provést následující úpravy.

- Všechny proměnné, jejichž derivace nás zajímají, musí být předeklarovány (například z typu `double`) na typ `doublet`. Toto se týká všech nezávislých proměnných, závislých proměnných a proměnných, které jsou závislé na nezávislých a ovlivňují závislé.

- Při přiřazení numerické hodnoty do nezávislé proměnné musí být také přiřazena odpovídající hodnota její derivace (tj. směru).

- Kromě tisku (obecněji exportu) závislých proměnných musí být také hodnota příslušné derivace vypsána (vyexportována).

Je zřejmé, že řídící příkazy (podmínky, cykly...) se nemění.[4]

Je vhodné (nikoli však bezpodmínečně nutné) mít funkci, která provede přiřazení hodnoty nezávislé proměnné (resp. konstanty) a její derivace (resp. nulové

[3] Zde konstantou myslíme hodnotu, která nezávisí na nezávislých proměnných.

[4] Toto je nepřesné tvrzení. Pokud například v podmínce testujeme hodnotu proměnné `z`, která závisí na $\boldsymbol{x}$, v přetransformovaném programu musíme testovat hodnotu `z.v`. Jinak se ale řídící příkazy opravdu nemění.

hodnoty místo derivace) do proměnné typu `doublet`. Tyto hodnoty bychom mohli do proměnných dosazovat i přímo, ale z hlediska čistoty programování a následných dalších zlepšení to není vhodné. Proto nadefinujeme funkci `makedoublet`, která přiřadí zadané hodnoty do složek proměnné typu `doublet`.

```
doublet makedoublet (double val, double dotval)
{
  doublet c;
  c.v = val;
  c.vdot = dotval;
  return c;
};
```

Nadefinujeme další variantu této funkce, která provádí téměř to samé.

```
doublet makedoublet (double val)
{
  doublet c;
  c.v = val;
  c.vdot = 0.0;
  return c;
};
```

Tyto funkce pak můžeme volat například jako `y = makedoublet(a,b);` nebo `z = makedoublet(2.0);`.

Jisté zjednodušení úprav derivovaného programu přináší přetížení operátoru přiřazení. Pak budeme moci na jeho pravé straně psát proměnnou typu `double`, zatímco na levé straně proměnnou typu `doublet` s významem, jako by byla volána funkce `makedoublet` s proměnnou typu `double`.

```
doublet& operator= (doublet& a, double b)
{
  a = makedoublet (b);
  return a;
};
```

Jako „opačnou“ k funkci `makedoublet` je vhodné zavést funkci `value`, resp. `dotvalue`, která ze zadané proměnné typu `doublet` předá hodnotu její složky `v`, resp. `vdot`. Vytvoření těchto funkcí je snadné, proto je zde neuvádíme. Transformaci zadaného programu ukážeme na jednoduchém příkladě.

Příklad

Chceme spočítat derivaci funkce

$$z = \ln \sum_{i=1}^{n} \left[(x - x_i)^2 + (y - y_i)^2 \right]$$

v bodě (x, y) ve směru $s = (\sin \alpha, \cos \alpha)$. Hodnoty x_i, y_i a α jsou konstanty. Tato funkce může být vyhodnocena pomocí následující ukázky zapsané v C++.

```
double x, y, z;
double xi, yi;
cin >> x;
cin >> y;
z = 0.0;
for (int i=0; i<n; i++)
{
  cin >> xi;
  cin >> yi;
  z = z + (x - xi)**2 + (y - yi)**2;
};
z = ln (z);
cout << z;
```

Na okraj poznamenejme, že příkazy `cin >> x;`, resp. `cout << z;` jsou operace s proudy (streamy), pomocí kterých přečteme, resp. zapíšeme hodnoty proměnných ze standardního vstupu, resp. na standardní výstup.

Tuto část programu ztransformujeme, aby počítal i žádanou derivaci. Tak dostaneme

```
doublet x, y, z;                          *
double xi, yi;
double alpha;                             *
cin >> x;
cin >> y;
cin >> alpha;                             *
x = makedoublet (x, sin (alpha));         *
y = makedoublet (y, cos (alpha));         *
z = 0.0;
for (int i=0; i<n; i++)
{
  cin >> xi;
  cin >> yi;
  z = z + (x - xi)**2 + (y - yi)**2;
};
z = ln (z);
cout << value (z);                        *
cout << valuedot (z);                     *
```

Veškeré změny, které jsme provedli, jsou předefinování příslušných proměnných z typu `double` na `doublet`, inicializace proměnných `x` a `y` a výpis hodnot složek proměnné `z`. Navíc jsme ještě přetížili operátor `>>`, aby pracoval s proměnnými typu `doublet`. Při jeho použití je hodnota složky `vdot` nastavena jako nulová. Všechny řádky, které jsou nové, nebo na kterých došlo k nějaké změně, jsme označili hvězdičkou. Jádro programu, ve kterém probíhá vlastní výpočet, zůstalo nezměněno. K úspěšnému běhu (resp. slinkování výsledného programu) potřebu-

jeme mít připravený soubor (balík) s pomocnými funkcemi atd., o kterém jsme se zmínili výše v této kapitole.

Pokud derivovaný program volá nějaké podprogramy, je nutné provést analogické úpravy i v těchto podprogramech.

Snadnými úpravami navrženého postupu bychom dosáhli současného vyhodnocení derivace podle p směrů najednou. Museli bychom nadefinovat nový typ proměnné (analogii `doublet`), znovu přetížit numerické operace a obstarat správnou inicializaci, vstup a výstup sdružených proměnných.

Zdaleka jsme se nezmínili o všech aspektech implementace přímého módu automatického derivování pomocí přetížení operátorů a funkcí. Některé další postřehy vyplynou z kapitoly 5.2, resp. 5.2.1.

5.1.2 Zpětný mód

V této kapitole popíšeme jednoduchou implementaci nasčítávaného zpětného módu automatického derivování, který jsme představili v kapitole 3.2. Tak budeme schopni spočítat gradient libovolné složky zadané funkce $\boldsymbol{F}$ nebo gradient lineární kombinace $\bar{\boldsymbol{y}} \cdot \boldsymbol{F}$ těchto složek.

V průběhu výpočtu hodnoty funkce $\boldsymbol{F}$ v bodě $\boldsymbol{x}$ budeme zaznamenávat každou základní operaci, kterou provedeme, spolu s jejími argumenty. Tyto údaje budeme sekvenčně ukládat do dlouhého pole. Po dokončení vyhodnocení $\boldsymbol{F}(\boldsymbol{x})$ projdeme polem zpět a pro každou v něm zaznamenanou operaci provedeme příslušné přidružené operace. Tím splníme požadavek na průchod soupisem operací v opačném pořadí, jak vyžaduje zpětný mód automatického derivování. Je zřejmé, že tento přístup není úplně vhodný pro velké úlohy. Proto je nutné uplatnit nějaká úsporná opatření, například použít „super-základní funkce“, o nichž se zmíníme na závěr této kapitoly.

Balík s pomocnými definicemi a funkcemi by měl obsahovat následující.

- Zavedení typu `element`, ve kterém budeme uchovávat informace o provedené základní operaci (hodnota $\boldsymbol{v}_i$ a $\bar{\boldsymbol{v}}_i$, kód operace, která $\boldsymbol{v}_i$ spočetla, a odkazy (ukazatele) na své argumenty). Dále zavedeme pole `trace` těchto `element`ů pro záznam všech provedených základních operací. K těmto typům zavedeme ještě další pomocné typy `redouble` (třída s ukazatelem na `element`) a `traceptr` (ukazatel na prvek v poli).

- Přetížení základních operací, aby pracovaly s parametry typu `redouble`. Takto nadefinované operace nejen počítají hodnotu $\boldsymbol{v}_i$ (jako jejich nepřetížení jmenovci), ale také zaznamenají své zavolání včetně svých parametrů do zmíněného pole.

- Zavedení funkce `reverse_sweep`, která po vyhodnocení $\boldsymbol{F}(\boldsymbol{x})$ projde polem `trace` nazpět a spočítá všechny přidružené proměnné $\bar{\boldsymbol{v}}_i$, jak určuje pravidlo řetězení.

- Zavedení funkcí pro korektní přiřazení hodnot nezávislých proměnných $\boldsymbol{x}_i$ a

vah $\bar{\boldsymbol{y}}_i$ do složek `*redouble` (tedy `elementu`) a naopak pro výstup spočtených derivací $\bar{\boldsymbol{x}}_i$.

- Případně také načtení a výstup proměnných typu `element`. Tyto činnosti může ale také dělat ztransformovaný program bez pomocných procedur.

Musíme také provést změny přímo v programu, který počítá hodnoty funkce $\boldsymbol{F}(\boldsymbol{x})$. Máme snahu, aby jich bylo co nejméně, nicméně následující jsou bezpodmínečně nutné.

- Všechny nezávislé proměnné, závislé proměnné a proměnné, které závisí na nezávislých a ovlivňují závislé, musí být předeklarovány na typ `redouble`.
- Hodnoty nezávislých proměnných musí být na začátku přiřazeny do správných složek proměnných typu `redouble`. Po vyhodnocení závislých proměnných musí být do jejich příslušných složek přiřazeny váhy $\bar{\boldsymbol{y}}_i$. Až pak může být zavolána funkce `reverse_sweep`, po jejímž dokončení je vyexportována hodnota derivace.
- Funkce `reverse_sweep` (druhá část soupisu operací) musí být zavolána, aby se spočetly hodnoty $\bar{\boldsymbol{v}}_i$ a tím i požadované derivace. Toto zavolání může být provedeno z funkce, která vypisuje hodnoty derivací.

Je zřejmé, že řídící příkazy (podmínky, cykly,...) se nemění.[5]

Nyní uvedeme, jak mohou vypadat definice nových typů a funkcí, které jsme výše popsali. Základní datové struktury jsou `element`, `trace` a `redouble`.

```
class element
{
  double v;
  double vbar;
  int opcode;
  element *arg1;
  element *arg2;
};
class redouble;
{
  element *ref;
};
element trace [VelkéČíslo];
element *traceptr = trace;
```

Tyto typy a proměnné jsme zavedli globálně, tj. jsou přístupné ze všech funkcí celého programu. Pokud by velikost pole `trace` nestačila (chtěli bychom provádět více základních operací než `VelkéČíslo`), je potřeba tuto hodnotu zvětšit

[5] Toto je nepřesné tvrzení. Pokud například v podmínce testujeme hodnotu proměnné `z`, která závisí na $\boldsymbol{x}$, v přetransformovaném programu musíme testovat hodnotu *složky* proměnné `z`, která hodnotu původní proměnné `z` obsahuje. Jinak se ale řídící příkazy opravdu nemění.

a spustit odvozený program znovu, případně použít super-základní funkce nebo jiné spořící techniky.

Dále potřebujeme definovat celočíselné konstanty pro jednoznačné určení základní funkce, která byla zavolána. Tyto konstanty budeme ukládat do složky `opcode` každého prvku pole `trace`, který charakterizuje provedenou operaci. Díky tomu budeme schopni ve druhé části soupisu operací provádět příslušné přidružené operace ve správném pořadí.

```
const int
  emptyv = 0, constv = 1, indepv = 2, bplusv = 3,
  bminusv = 4, bmultv = 5, recipv = 6, ...
  expv = 11, lnv = 12, sinv = 13, ... ;
```

Přetížené funkce můžeme definovat například takto.

```
redouble sin (redouble a)
{
  redouble b;
  b.ref =traceptr;
  traceptr->v = sin (a.ref->v);
  traceptr->vbar = 0.0;
  traceptr->opcode = sinv;
  traceptr->arg1 = a.ref;
  traceptr++;
  return b;
};
redouble operator* (redouble a, redouble b)
{
  redouble c;
  c.ref =traceptr;
  traceptr->v = (a.ref->v) * (b.ref->v);
  traceptr->vbar = 0.0;
  traceptr->opcode = bmultv;
  traceptr->arg1 = a.ref;
  traceptr->arg2 = b.ref;
  traceptr++;
  return c;
};
```

Každá taková přetížená funkce vyčíslí hodnotu složky `v`, vynuluje hodnotu `vbar`, zaznamená se (co je zač) a nakonec ještě uloží ukazatele na své argumenty.

Pro operátor `*` (a další binární operátory) je dále potřeba nadefinovat smíšené operace pro konstanty (například typu `double`)[6] a proměnné typu `redouble`. Podobným způsobem přetížíme další aritmetické operace a používané funkce.

Je vhodné mít funkci, která přiřadí hodnotu nezávislé proměnné na správné místo do proměnné typu `redouble` a doplní odpovídající operace. Nazvěme ji `makeindepvar`.

[6]Zde konstantou myslíme hodnotu, která nezávisí na nezávislých proměnných.

```
  redouble makeindepvar (double a)
  {
    redouble b;
    b.ref =traceptr;
    traceptr->v = a;
    traceptr->vbar = 0.0;
    traceptr->opcode = indepv;
    traceptr++;
    return b;
  };
```

Potom již můžeme psát například `x = makeindepvar(a);`. Stejně jako v předchozí kapitole můžeme přetížit operátor přiřazení, což přinese příjemné výhody.

Dále musíme napsat funkci `reverse_sweep`, která projde polem `trace` nazpět a pro každou operaci provede příslušné přidružené operace. Tato procedura je vlastně jen cyklus (polem odzadu), ve kterém příkaz `switch` třídí prováděné funkce.

```
void reverse_sweep
{
 double deriv;
 while (traceptr-- > trace)
  {
   switch(traceptr->opcode)
    {
     ...
     case sinv:
      deriv = cos(traceptr->agr1->v);
      traceptr->arg1->vbar += traceptr->vbar * deriv;
     break;
     ...
     case bmultv:
      traceptr->arg1->vbar += traceptr->vbar * traceptr->arg2->v;
      traceptr->arg2->vbar += traceptr->vbar * traceptr->arg1->v;
     break;
     ...
    }
  }
};
```

Po proběhnutí této procedury jsou ve složkách `vbar` těch proměnných, do kterých jsme přiřazovali nezávislé hodnoty, žádané hodnoty gradientu.

Čtení spočtených hodnot závislých proměnných bude provádět funkce `value` a přiřazení příslušných přidružených hodnot (tj. vah) funkce `setbarvalue`. Tato funkce musí být provedena před zavoláním funkce `reverse_sweep`. Hodnoty derivací zjistíme po proběhnutí funkce `reverse_sweep` pomocí funkce `barvalue`. Tyto funkce si jsou velmi podobné, proto ukážeme pouze například `barvalue`.

```
double barvalue (redouble a)
{
  double b;
  b = a.ref->vbar;
  return b;
};
```

Je užitečné zavést binární proměnnou (příznak), která bude vypovídat, zda už proběhla funkce `reverse_sweep`. Ve funkci `barvalue` pak budeme tuto proměnnou testovat a v případě potřeby funkci `reverse_sweep` zavoláme odtud. Tento přístup má i svou další výhodu – nebudeme muset vkládat volání funkce `reverse_sweep` přímo do transformovaného programu.

Zopakujme, že v zadaném programu nemusíme dělat žádné jiné změny než předeklarování příslušných proměnných na typ `redouble`, správné přiřazení hodnot nezávislých proměnných (lze provést přetížením operátoru přiřazení), po spočtení hodnot $\boldsymbol{F}(\boldsymbol{x})$ musíme přiřadit váhy $\bar{\boldsymbol{y}}_i$ a nakonec přečíst vypočtené hodnoty derivací (zavolání funkce `reverse_sweep` jsme zautomatizovali). Jinak řečeno, vlastní jádro programu, kde probíhá výpočet, je stejně jako v případě přímého módu nezměněno. Proto zde neukážeme příklad jako v minulé kapitole, byl by téměř shodný.

Pokud derivovaný program volá nějaké podprogramy, je nutné provést analogické úpravy i v těchto podprogramech.

Snadnou modifikací navržených postupů můžeme vyhodnocovat q gradientů najednou. Za tímto cílem musíme změnit definici typu `element`, dále funkce `setbarvalue`, `barvalue` a také `reverse_sweep`. Všimněme si, že definice přetěžovaných operací volaných v první části soupisu operací upravovat nemusíme, pokud přetížíme přiřazení $\bar{\boldsymbol{v}}_i = 0$, resp. `traceptr->var = 0.0`, aby na jeho levé straně bylo pole s q prvky.

Poznamenejme také, že hodnoty vah $\bar{\boldsymbol{y}}$ stačí zadat až po dokončení vyhodnocení $\boldsymbol{F}(\boldsymbol{x})$. To znamená, že pokud váhy $\bar{\boldsymbol{y}}$ závisí na vstupních datech $\boldsymbol{x}$ a jejich hodnotu zjistíme během výpočtu $\boldsymbol{F}(\boldsymbol{x})$, nedojde ve výpočtu k žádnému zdržení nebo opakovanému výpočtu[7].

Po proběhnutí funkce `reverse_sweep` je možné pole `trace` znovu použít pro další vyhodnocení derivace. K tomuto opětovnému použití poslouží přiřazení `traceptr = trace;`, které zajistí, že údaje se do něj budou zapisovat znovu od začátku.

Jinou možností, jak naimplementovat zpětný mód automatického derivování, je počítat derivace $\boldsymbol{c}_{ij}$ ihned v první části přidruženého soupisu operací a ukládat je pro použití v druhé části. V tomto případě se ve druhé části přidruženého soupisu operací obejdeme bez hodnoty $\boldsymbol{v}_i$ a bez charakterizace operace `opcode`. Přesněji řečeno, využíváme graf, jehož vrcholy jsou $\bar{\boldsymbol{v}}_i$ a s každou hranou je spojena hodnota derivace $\boldsymbol{c}_{ij}$. Struktura tohoto grafu se vytvoří v první části soupisu operací. Funkce `reverse_sweep` jen tímto grafem projde a vyhodnotí příslušné

[7] Pokud by v přímém módu automatického derivování směry závisely na vstupních datech $\boldsymbol{x}$ a počítaly by se při výpočtu $\boldsymbol{F}(\boldsymbol{x})$, pak by se (nepřesně řečeno) příslušná část výpočtu musela opakovat dvakrát.

přidružené operace, čímž určí hodnoty $\bar{\boldsymbol{v}}_i$. Tento přístup má výhodu v menší náročnosti na paměť, ale po dokončení vyhodnocení $\boldsymbol{F}(\boldsymbol{x})$ nelze tento postup vyhodnocení v plné míře zrekonstruovat, takže je nemožné například počítat další derivace.

Jak jsme slíbili, popíšeme jeden ze způsobů, jak ušetřit část pole `trace` pomocí *super-základních funkcí*. Zatím jsme za základní funkce $\boldsymbol{\varphi}_i$ považovali elementární operace (plus, krát, sinus atd.). Pokud by však vyhodnocení funkce $\boldsymbol{F}$ v bodě $\boldsymbol{x}$ používalo nějaké postupy (několik základních funkcí za sebou), které se opakují a které používají mnoho pomocných proměnných, které nejsou později potřeba,[8] je účelné tento postup označit jako novou základní funkci $\boldsymbol{\varphi}_j$ a nazývat ji super-základní funkcí. Jednotlivé elementární operace prováděné během jejího vyhodnocování by se nezaznamenávaly do pole `trace`, pouze výsledek celé super-základní funkce. Při zpětném průchodu polem `trace` (tj. kumulaci přidružených hodnot) by se hodnota derivací super-základní funkce spočítala pomocí *lokálního automatického derivování* přímo na místě. Tímto postupem tedy omezíme využívání pole `trace`.

Nezmínili jsme některé aspekty transformace programu při zpětném módu automatického derivování. Tento prohřešek alespoň zčásti napravíme v kapitole 5.2, resp. 5.2.2.

5.1.3 Druhé a vyšší derivace

Naznačíme, jak naimplementovat výpočet hodnoty $\boldsymbol{F}''(\boldsymbol{x}) \cdot \boldsymbol{s}$ pro skalární funkci $\boldsymbol{F} : \mathbb{R}^n \longrightarrow \mathbb{R}$, kdy $\boldsymbol{s}$ nezávisí na $\boldsymbol{x}$. K tomu použijeme metodu automatického derivování, kterou jsme odvodili v kapitole 4.1. Pokud $\boldsymbol{s}$ bude i-tý vektor kartézské báze, spočítáme tak i-tý sloupec Hessovy matice funkce $\boldsymbol{F}$ v bodě $\boldsymbol{x}$.

Použijeme implementaci zpětného módu z předcházející kapitoly, ale pozměníme definici typu `element` takto.

```
class element
{
  doublet v;
  doublet vbar;
  int opcode;
  element *arg1;
  element *arg2;
};
```

Složky `v` a `vbar` nyní nejsou typu `double`, ale `doublet`. Proto musíme předefinovat některé inicializační a naopak přístupové (exportovací) funkce, například `value` nebo `barvalue`, které nyní budou vracet hodnotu typu `doublet`. Do balíku s pomocnými definicemi a funkcemi přidáme také přetížené funkce pro typ `doublet`.

Jednotlivé složky proměnné `v` typu `element` lze interpretovat (po dokončení výpočtu příslušné hodnoty) takto.

[8] Jedná se například o řešení soustavy lineárních rovnic.

`value(value(v))`	$= v$
`dotvalue(value(v))`	$= \dot{v}$
`value(barvalue(v))`	$= \bar{v}$
`dotvalue(barvalue(v))`	$= \dot{\bar{v}}$

Díky přetížení operátorů a funkcí bude funkce `reverse_sweep` například pro funkci sinus obsahovat

```
case sinv:
  deriv = cos (traceptr->arg1->v);
  traceptr->arg1->vbar += traceptr->vbar * deriv;
break;
```

Složky `v`, `vbar` a `deriv` jsou typu `doublet`. Operace s nimi jsou přetížené a je snadné si rozmyslet, že díky tomu provádí `reverse_sweep` přesně to, co požadujeme v kapitole 4.1.

Pokud by složka `vdot` proměnné typu `doublet` byla vektor (pole) délky p, lze tak vyhodnotit p sloupců Hessovy matice zároveň, tedy speciálně například celou Hessovu matici. Je jasné, že bychom při této modifikaci museli přepsat definice přetěžujících funkcí pro typ `doublet`, stejně jako inicializační, přístupové, vstupní a výstupní funkce. Vše je ale snadné.

Tak jako lze vnořovat a na sebe aplikovat jednotlivé transformace přímého a zpětného módu (typicky například zpětný, na něj přímý a ještě jednou přímý pro výpočet třetí derivace), lze i výše uvedené typy podobně vnořovat. Pokud správně přetížíme jednotlivé operace, dostáváme korektní výsledky. Například pro zmíněnou třetí derivaci můžeme definovat typ `triplet` jako

```
class triplet
{
  double v;
  double vdot;
  double vdotdot;
};
```

nebo

```
class triplet
{
  double v;
  doublet vdot;
};
```

Je vhodné tento typ `triplet` (a také ještě obecnější typy) použít v definici typu `element`, protože pak díky zpětnému módu získáme hodnoty třetích (případně vyšších) derivací. Příslušné definice přetížených operací je nutné nadefinovat obvyklým způsobem.

5.2 Použití preprocesoru

Implementaci metod automatického derivování můžeme kromě techniky přetížení operátorů a funkcí dosáhnout i použitím preprocesoru, o kterém se zmíníme v této kapitole.

Preprocesor je program, jehož vstupem je program v určitém programovacím jazyce a výstupem je program ve stejném programovacím jazyce, který byl nějakým způsobem pozměněn. V našem případě bude program na spočtení hodnot funkce $\boldsymbol{F}$ v bodě $\boldsymbol{x}$ změněn na program, který bude navíc počítat i derivace funkce $\boldsymbol{F}$ v bodě $\boldsymbol{x}$. Nyní není úplně jasný rozdíl mezi technikami přetížení a preprocesoru – v obou případech přece měníme původní program. *Preprocesor* analyzuje zdrojový program řádek po řádku, přiřazovací příkazy rozloží na jednotlivé základní funkce a podle nich vkládá příslušné operace, aby nový program počítal také derivace (v přímém módu například vkládá sdružené operace). Tento výsledek uloží do vznikajícího souboru (programu). Jinak řečeno, výpočet derivace je fyzicky vložen preprocesorem do původního programu. Programovací jazyk, ve kterém je program na vyhodnocení funkce $\boldsymbol{F}$ napsaný, může tedy být jakýkoliv běžný programovací jazyk, který nemusí podporovat přetížení operátorů a funkcí.[9]

Připomeňme, že ztransformovaný program bude používat některé definice a funkce uložené v externím souboru, podobně jako při implementaci přetížením operátorů a funkcí.

Preprocesoru je nutné při jeho zavolání sdělit označení nezávislých a závislých proměnných, jaké derivace požadujeme a pokud možno i mód automatického derivování, který má použít. Tyto údaje je možné zadat nějakým dohodnutým způsobem přímo v těle programu nebo v jiném (parametrickém) souboru (čímž nemusíme zasahovat do původního programu).

V následujících odstavcích popíšeme pro přímý i zpětný mód automatického derivování způsob implementace, který není úplně doslova použitím preprocesoru. Je přímou modifikací implementace přetížením, odkud jsme ale přetěžování odstranili. Pak naznačíme i „klasickou“ implementaci přetížením.

5.2.1 Přímý mód

Na půl cesty mezi přetížením a preprocesorem

Vyjdeme z implementace, kterou jsme navrhli v kapitole 5.1.1 pomocí přetížení operátorů a funkcí. Toto řešení zůstane prakticky zachováno, nahradíme však přetížení.

Preprocesor projde programem pro vyhodnocení funkce $\boldsymbol{F}$ (zdrojový soubor). Pro každé přiřazení, které nalezne, provede rozklad jeho pravé strany na jednotlivé základní funkce. Volání každé této základní funkce nahradí voláním jiné

[9] Může nám přijít zbytečné důrazně rozlišovat dva způsoby implementace automatického derivování, i když si jsou velmi podobné a vzájemně se „prolínají“. Zdá se ale být praktické a nanejvýš názorné tyto techniky oddělovat, proto tak budeme činit.

funkce (s jiným názvem), která bude provádět i sdružené operace. Například volání funkce `sin (x)` nahradíme voláním funkce `sint (x)`,[10] která bude kromě vyhodnocení sinu provádět i příslušnou sdruženou operaci. U funkcí (operátorů), které se zapisují v infixové notaci, budeme postupovat podobně – binární násobení `x * y` nahradíme zavoláním funkce `bmultt (x, y)`. Složíme-li zpátky pozměněné volání funkcí do výrazu ve stejném pořadí (do stejné struktury), jak jsme ho rozebrali, dostaneme výraz, který bude provádět vyhodnocení nejen své původní hodnoty, ale také své derivace. K tomu je samozřejmě nutné, abychom nadefinovali nový typ proměnné, se kterým budou tyto funkce pracovat. Její definice může být naprosto shodná s definicí typu `doublet` v kapitole 5.1.1., tj. dvojice reálných hodnot. Pokud je volána funkce, jejíž jeden argument je konstanta (tj. nezávisí na x), máme na výběr dva přístupy. Buď z konstanty uděláme proměnnou typu `doublet` s nulovou složkou `vdot` a zavoláme funkci pro dva `doublety`, nebo zavedeme novou funkci, například `bmulttconst (x,c)`, kdy druhý argument je konstanta. Definice nových funkcí spolu s definicemi nových typů proměnných budou uvedeny v externím souboru (balíku). Například tedy

```
doublet sint (doublet a)
{
  doublet b;
  b.v = sin (a.v);
  b.vdot = cos (a.v) * a.vdot;
  return b;
};
doublet bmultt (doublet a, doublet b)
{
  doublet c;
  c.v = a.v * b.v;
  c.vdot = a.vdot * b.v + a.v * b.vdot;
  return c;
};
```

Další úpravy jsou stejné jako u implementace přetížením, takže je popíšeme jen stručně. Preprocesor musí v původním programu zaměnit typy proměnných na `doublet` u všech nezávislých proměnných, závislých proměnných a proměnných, které závisí na nezávislých a ovlivňují závislé.[11] Dále musí zajistit přiřazení směrů

[10] Zde `t` znamená *tangent*.

[11] Zjištění, kterou proměnnou předefinovat a kterou ne, je snadné. Stačí, aby si preprocesor při analýze výrazů všímal, které proměnné jsou závislé na nezávislých. Ty je třeba předefinovat. Tímto způsobem bychom měli vybrat i závislé proměnné. Pokud se v programu používají ukazatele na proměnné (nebo při složitější práci s poli) nemusí být vždy zřejmé, které proměnné se vzájemně ovlivňují. V tom případě je potřeba postupovat „co nejopatrněji“, tj. předefinovat každou proměnnou, která by snad mohla být závislá na nezávislých proměnných. Bohužel, tímto přístupem možná předefinujeme i proměnné, které jsme předefinovat nemuseli, čímž budeme provádět zbytečné aritmetické operace. I za této situace získaný výsledek bude správný. Podotkněme ještě, že toto hledání závislostí proměnných se bude díky prováděné analýze výrazů provádět mnohem lépe v případě použití preprocesoru než v případě přetěžování.

do sdružených proměnných (například funkcí `makedoublet`) na začátku programu a naopak výstup spočtených derivací na konci programu. Řídící příkazy zůstávají zachovány (až na případ testování hodnoty $\boldsymbol{v}_i$ – viz poznámka pod čarou na straně 49).

Soubor (balík) s pomocnými definicemi typů proměnných a funkcí musí tedy obsahovat definici typu `doublet`, definice základních funkcí (s pozměněnými jmény) pro typ `doublet` (případně jeho kombinace s dalšími typy), inicializaci proměnných a výstup hodnot pro proměnné typu `doublet`.

Pokud program pro výpočet hodnot funkce $\boldsymbol{F}$ používá nějaké podprogramy, je nutné i v těchto podprogramech provést změnu typů proměnných a změnu jmen volaných funkcí. Nesmíme zapomenout na změnu v parametrech těchto podprogramů.

Pokud bychom chtěli vyhodnotit derivaci v p směrech najednou, snadno zmodifikujeme navržené postupy – viz kapitola 5.1.1.

Další implementace na půl cesty mezi přetížením a preprocesorem (Implementace přímého módu ve Fortranu 77)

Některé programovací jazyky (například Fortran 77) neumožňují definování nových typů proměnných. Pro tyto jazyky jsou tedy dosud popsané implementace nepoužitelné. Jedno z celé řady možných řešení je velmi podobné implementaci zpětného módu pro Fortran 77, které je velmi podrobně popsáno na stranách 63 až 66. Proto implementaci přímého módu automatického derivování pro Fortran 77 uvedeme až na straně 67.

Klasické použití preprocesoru

Další z řady možných implementací pro Fortran 77 naznačíme v tomto odstavci. Protože však bude v mnoha rysech podobné již navrženým implementacím, budeme struční.

Ke každé proměnné, která je buď nezávislá, nebo závislá, nebo ovlivněná nezávislými a ovlivňuje závislé, nadeklarujeme novou proměnnou (zpravidla stejného typu), která bude mít význam její derivace. Její jméno vznikne přidáním písmena `D` na začátek jména původní proměnné. Například, pro proměnnou `X` vznikne nová proměnná `DX`. Preprocesor bude nyní procházet zdrojový program a pro každý přiřazovací příkaz, který nalezne, rozloží pravou stranu na základní operace. Podle toho určí vzorec pro výpočet derivace pravé strany. Do vznikajícího (výsledného) programu vloží jak původní přiřazení (například `Y=...`), tak i nové pro výpočet derivace (`DY=...`). Podmínky, cykly atp. preprocesor měnit nebude. Pokud se při výpočtu používá nějaký podprogram, je nutné ho upravit stejným způsobem, aby počítal derivace, a zajistit jeho volání. Dále je nutné připravit vstup a výstup proměnných, které uchovávají hodnoty derivací. Zobecnění na vektory je již snadné.

5.2.2 Zpětný mód

Znovu na půl cesty mezi přetížením a preprocesorem

Tato implementace zpětného módu automatického derivování pomocí preprocesoru vyjde z implementace přetížením operátorů a funkcí popsané v kapitole 5.1.2, přičemž použijeme stejný nápad jako v předcházející kapitole 5.2.1. Veškeré přetěžování obejdeme pomocí funkcí s různými jmény, jinak vše zůstane zachováno. Proto budeme poněkud struční.

Preprocesor bude při průchodu programem, který počítá hodnoty funkce $\boldsymbol{F}$ v bodě $\boldsymbol{x}$, rozkládat pravé strany přiřazovacích příkazů a volání jednotlivých základních funkcí bude nahrazovat voláním jiných funkcí, které budou kromě vyhodnocování hodnoty nahrazované funkce také zaznamenávat své zavolání včetně argumentů. Například, funkci `sin (x)` nahradíme funkcí `sing (x)` a operaci `x * y` nahradíme funkcí `bmultg (x, y)`.[12] Toto provedeme pro celé vyhodnocení $\boldsymbol{F}(\boldsymbol{x})$. Argumenty a výsledné hodnoty těchto pozměněných operací jsou typu `redouble`, stejně jako při implementaci přetížením v kapitole 5.1.2. Pokud argument nějaké binární funkce bude konstanta, pak máme dvě možnosti postupu. Buď vytvoříme novou funkci, jejímž jedním argumentem bude konstanta, nebo z konstanty vytvoříme proměnnou typu `redouble` a na ni zavoláme funkci pro situaci, kdy oba argumenty jsou typu `redouble`. Definice všech nových typů proměnných (`element`, `redouble`, `trace`, `traceptr`) spolu s definicemi funkcí budou uvedeny v externím souboru (balíku). Například tedy

```
redouble sing (redouble a)
{
  redouble b;
  b.ref = traceptr;
  traceptr->v = sin (a.ref->v);
  traceptr->vbar = 0.0;
  traceptr->opcode = sinv;
  traceptr->arg1 = a.ref;
  traceptr++;
  return b;
};
redouble bmultg (redouble a, redouble b)
{
  redouble c;
  c.ref =traceptr;
  traceptr->v = (a.ref->v) * (b.ref->v);
  traceptr->vbar = 0.0;
  traceptr->opcode = bmultv;
  traceptr->arg1 = a.ref;
  traceptr->arg2 = b.ref;
  traceptr++;
```

[12] Písmenko `g` na konci jména volané funkce znamená *gradient*.

```
    return c;
  };
```

Preprocesor dále musí předeklarovat na typ `redouble` všechny nezávislé proměnné, závislé proměnné a také ty proměnné, které závisí na nezávislých a ovlivňují závislé. Je nutné přiřadit vstupní hodnoty nezávislým proměnným před vyhodnocením $\boldsymbol{F}(\boldsymbol{x})$. Dále je nutné dosadit správné údaje do přidružených proměnných závislých hodnot (tj. $\bar{\boldsymbol{y}}$), ale to je možné až po dokončení vyhodnocení $\boldsymbol{F}(\boldsymbol{x})$. Pak se musí zavolat funkce `reverse_sweep` a po jejím proběhnutí vyexportovat hodnoty derivací. Řídící příkazy zůstávají zachovány (až na případ testování hodnoty $\boldsymbol{v}_i$ – viz poznámka pod čarou na straně 49).

Soubor (balík) s pomocnými definicemi typů proměnných a funkcí musí obsahovat definici typů `redouble`, `element`, `trace` a `traceptr`, dále definice základních funkcí (s pozměněnými jmény) pro typ `redouble`, inicializaci a výstup hodnot pro proměnné typu `redouble` a nakonec i funkci `reverse_sweep`. Tato funkce je naprosto stejná jako pro implementaci přetížením v kapitole 5.1.2.

Pokud program pro výpočet hodnot funkce $\boldsymbol{F}$ používá nějaké podprogramy, je nutné i v těchto podprogramech provést změnu typů proměnných a změnu jmen volaných funkcí. Nesmíme zapomenout na změnu v parametrech těchto podprogramů.

Implementace zpětného módu pomocí grafu, kterou jsme uvedli na konci kapitoly 5.1.2, lze poměrně snadno naimplementovat také bez přetížení operátorů a funkcí.

Pokud bychom chtěli vyhodnotit q gradientů najednou, snadno zmodifikujeme navržené postupy – viz kapitola 5.1.2.

Ještě jednou na půl cesty mezi přetížením a preprocesorem (Implementace zpětného módu ve Fortranu 77)

Jak jsme již zmínili výše, některé programovací jazyky neumožňují pracovat s ukazateli nebo definovat nové typy proměnných. Proto popíšeme modifikaci implementace automatického derivování z předchozího odstavce, která tyto vlastnosti obejde. Opět tedy budeme zaznamenávat jednotlivé prováděné základní operace a informace o nich ukládat do dlouhého pole, kterým na závěr projdeme odzadu a provedeme příslušné přidružené operace. Uvedené ukázky jsou napsané v programovacím jazyce Fortran 77.

Třída `element` obsahuje pět složek, a to `v`, `vbar`, `opcode`, `arg1` a `arg2`. Pole `trace` těchto `element`ů nahradíme pěti samostatnými poli `V`, `VBAR`, `OPCODE`, `ARG1` a `ARG2` typu `REAL`, `REAL`, `INTEGER`, `INTEGER` a `INTEGER` (ve stejném pořadí), ve kterých budeme uchovávat stejné hodnoty jako v jednotlivých složkách proměnné typu `element` v poli `trace`. Neboli, k-tý prvek v poli `trace` má stejné hodnoty složek jako k-té prvky polí `V`, `VBAR`, `OPCODE`, `ARG1` a `ARG2`.

Pro odkaz na konkrétní provedenou základní funkci $\boldsymbol{\varphi}_k$ (a s ní související údaje) budeme namísto proměnné typu `redouble` používat *index polí*[13]. Proto

[13] Všechny hodnoty v polích, které jsou příslušné jedné operaci, mají stejný index.

v polích ARG1 a ARG2 budeme uchovávat právě index na prvky polí s argumenty příslušné základní funkce.

Pole budeme deklarovat takto.

```
  PARAMETER (LNGARR = 1000)
  REAL V(LNGARR), VBAR(LNGARR)
  INTEGER OPCODE(LNGARR), ARG1(LNGARR), AGR2(LNGARR)
  INTEGER INDARR
```

Konstanta LNGARR (délka polí) je nějaké velké číslo, které má stejný význam jako hodnota VelkéČíslo na straně 53. Pokud bychom prováděli více základních operací než LNGARR, je potřeba tuto hodnotu zvětšit a spustit odvozený program znovu. Proměnná INDARR znamená index prvku v polích, kam budeme zapisovat následující prováděnou základní operaci.

Zavedeme konstanty, podle kterých rozeznáme, jakou základní funkci jsme volali.

```
  PARAMETER (EMPV = 0, CONV = 1, INDV = 2, BPLUSV = 3,
 *           BMINUV = 4, BMULTV = 5, RECIPV = 6, ...
 *           EXPV = 11, LNV = 12, SINV = 13, ...)
```

Předefinujeme všechny elementární operace, aby jejich argumenty byly indexy polí (kde jsou uloženy hodnoty argumentů atd.) a aby vracely index polí, pod kterým zaznamenaly své zavolání včetně výsledné funkční hodnoty. Takto pozměněné funkce přejmenujeme, například funkce SIN bude nově SING. Pro jistotu budeme pole se záznamy o provedených základních operacích spolu s proměnnou INDARR předávat pomocí parametrů.

```
  INTEGER FUNCTION SING (IARG1,
 *                       V, VBAR, OPCODE, ARG1, ARG2, INDARR)
    REAL V(LNGARR), VBAR(LNGARR)
    INTEGER OPCODE(LNGARR), ARG1(LNGARR), AGR2(LNGARR)
      V(INDARR) = SIN (V(IARG1))
      VBAR(INDARR) = 0.0
      OPCODE(INDARR) = SINV
      ARG1(INDARR) = IARG1
      SING = INDARR
      INDARR = INDARR + 1
    END

  INTEGER FUNCTION BMULTG (IARG1, IARG2,
 *                         V, VBAR, OPCODE, ARG1, ARG2, INDARR)
    REAL V(LNGARR), VBAR(LNGARR)
    INTEGER OPCODE(LNGARR), ARG1(LNGARR), AGR2(LNGARR)
      V(INDARR) = V(IARG1) * V(IARG2)
      VBAR(INDARR) = 0.0
      OPCODE(INDARR) = BMULTV
```

```
      ARG1(INDARR) = IARG1
      ARG2(INDARR) = IARG2
      BMULTG = INDARR
      INDARR = INDARR + 1
    END
```

Pro binární operace nadefinujeme ještě funkce, kdy jeden z argumentů je konstanta nezávislá na nezávislých proměnných.[14]

Nezapomeňme, že všechna volání základních funkcí musí být v původním programu nahrazena voláním modifikovaných funkcí, které jsme právě zmínili.

Všechny nezávislé proměnné, závislé proměnné a proměnné, které závisí na nezávislých a ovlivňují závislé, předeklarujeme na typ `INTEGER`, neboli index prvku polí, ve kterých o nich budou uschovávány informace.

Hodnoty nezávislých proměnných je nutné před začátkem výpočtu dosadit do polí, spolu se všemi příslušnými údaji. K tomu se hodí funkce `MKINDP`.

```
  INTEGER FUNCTION MKINDP (VALUE,
 *                         V, VBAR, OPCODE, ARG1, ARG2, INDARR)
    REAL V(LNGARR), VBAR(LNGARR)
    INTEGER OPCODE(LNGARR), ARG1(LNGARR), AGR2(LNGARR)
      V(INDARR) = VALUE
      VBAR(INDARR) = 0.0
      OPCODE(INDARR) = INDV
      MKINDP = INDARR
      INDARR = INDARR + 1
    END
```

Nejpozději po dokončení vyhodnocení funkce $\boldsymbol{F}$ v bodě $\boldsymbol{x}$ je potřeba přiřadit hodnoty vah $\bar{\boldsymbol{y}}_i$ na správná místa v poli `VBAR`, aby se mohla spustit subrutina (podprogram) `RVRSWP` pro zpětný průchod poli a vypočtení všech přidružených hodnot. Vytvoření subrutiny pro přiřazení vah je snadné, proto se tím zde nebudeme zabývat. Subrutina `RVRSWP` může mít například následující podobu.

```
  SUBROUTINE RVRSWP (V, VBAR, OPCODE, ARG1, ARG2, INDARR)
    REAL V(LNGARR), VBAR(LNGARR)
    INTEGER OPCODE(LNGARR), ARG1(LNGARR), AGR2(LNGARR)
    REAL DERIV
    INTEGER I
    DO 999, I = INDARR-1, 1, -1
        IF(OPCODE(I) .EQ. CONV) THEN
          .
          .
          .
        ELSE IF(OPCODE(I) .EQ. BMULTV) THEN
```

[14] Můžeme postupovat i jinak. Konstantu zaznamenáme do polí jako konstantu a pak provedeme „funkci pro dva indexy“.

```
          VBAR(ARG1(I)) = VBAR(ARG1(I)) + VBAR(I) * V(ARG2(I))
          VBAR(ARG2(I)) = VBAR(ARG2(I)) + VBAR(I) * V(ARG1(I))
        ELSE IF(OPCODE(I) .EQ. SINV) THEN
          DERIV = COS (V(ARG1(I)))
          VBAR(ARG1(I)) = VBAR(ARG1(I)) + VBAR(I) * DERIV
        ELSE IF(OPCODE(I) .EQ. COSV) THEN
          .
          .
          .
        END IF
999   CONTINUE
    END
```

Po jejím provedení jsou již v prvcích pole `VBAR` příslušejících nezávislým proměnným spočtené hodnoty požadovaných derivací.

Řídící sekvence (podmínky, cykly atd.) se právě popsanou transformací nemění, s výjimkou změny názvu proměnných, například `ABC` je potřeba změnit na `V(ABC)`. Pokud se v programu pro vyhodnocení funkce $\boldsymbol{F}$ používají nějaké podprogramy, je nutné je upravit analogickým způsobem. Přitom bychom neměli zapomenout na změnu v jejich parametrech. Kdybychom požadovali vyčíslení q gradientů najednou, změníme pole `VBAR` na dvourozměrné a zřejmým způsobem upravíme související funkce, zejména `RVRSWP`.

Klasické použití preprocesoru

Naznačíme jinou implementaci zpětného módu, ve které se vyhneme polím, zato bude velmi náročná na preprocesor. Nápad je snad vůbec nejjednodušší, jaký může být. Naznačíme myšlenky, technické detaily je snadné domyslet.

Pro každou nezávislou proměnnou, závislou proměnnou a proměnnou, která závisí na nezávislých a ovlivňuje závislé, nadeklarujeme novou proměnnou, která bude mít význam přidružené proměnné. Nazveme ji jako původní proměnnou, jen na začátek jména vložíme písmeno `G`. Preprocesor bude číst řádek po řádku původní program a pro každou operaci bude generovat příslušnou přidruženou operaci. Ty je však potřeba vkládat do nového programu odzadu. Je třeba řádně ošetřit podmínky a cykly – ve stejné podobě jako jsou ve vstupním programu, musí být i ve výstupním programu, jen (přidružené) operace se budou provádět v opačném pořadí. Díky přepisování proměnných není zaručené, že při průchodu nazpět by výsledek testů v podmínce nebo hranice cyklů byly stejné, jako při původním vyhodnocení funkce $\boldsymbol{F}$ v bodě $\boldsymbol{x}$. Proto je vhodné výsledky testů a hranice cyklů uschovat a použít je v průchodu nazpět. Je třeba řádně ošetřit vstup a výstup nezávislých, závislých a přidružených proměnných. Pokud se ve vyhodnocení funkce $\boldsymbol{F}$ v bodě $\boldsymbol{x}$ volají podprogramy, je nutné i tyto podprogramy upravit pro výpočet derivace.

Velké nároky na preprocesor jsou vyváženy velmi malými režijními náklady při běhu odvozeného programu, narozdíl od jiných způsobů implementace.

Implementace přímého módu pro Fortran 77

V tomto odstavci stručně naznačíme implementaci *přímého* módu automatického derivování, které je velmi podobné předchozí implementaci zpětného módu, ve které jsme použili pět polí.

I zde budeme používat pole pro uchování vypočtených hodnot základních funkcí a jejich sdružených proměnných. Nazveme je `V` a `VDOT`. Předefinujeme (a přejmenujme) základní funkce, aby jejich argumenty byly index prvku pole (kde jsou uloženy argumenty) a vracely index prvku pole, kam uložily spočtené hodnoty. Protože není nutné uchovávat výsledné hodnoty všech základních operací (ale jen ty, které při výpočtu hodnoty funkce $\boldsymbol{F}$ v bodě $\boldsymbol{x}$ ukládáme do proměnných), zavedeme ještě další dvě pole `VTMP` a `VDOTTM`, do kterých budeme ukládat hodnoty spočtené uvnitř aritmetického výrazu a tudíž nepotřebné po dokončení jeho vyhodnocení. Při zpracování každého výrazu budeme zapisovat do těchto polí od jejich začátku. Do polí `V` a `VDOT` budeme tedy ukládat pouze ty hodnoty, které se ve vyhodnocení funkce $\boldsymbol{F}$ v bodě $\boldsymbol{x}$ ukládají do proměnných. I zde je nutné předefinovat nezávislé proměnné, závislé proměnné a proměnné, které závisí na nezávislých a ovlivňují závislé, na typ `INTEGER` jako index pole. Tento návrh implementace je nutné doplnit o další technické podrobnosti[15], které však nejsou složité, proto je zde neuvádíme.

5.2.3 Druhé a vyšší derivace

V kapitole 5.1.3 jsme uvedli implementaci výpočtu druhých a vyšších derivací pomocí přetížení operátorů a funkcí. Tato vlastnost přetěžování se používala významným způsobem. Pokud chceme výpočet druhých a vyšších derivací implementovat pomocí preprocesoru, nahradíme veškerá přetížená volání funkcí voláním funkcí s různými názvy. Jejich těla budou (až na změněná případná volání) stejná jako při implementaci přetěžováním. Definice typů proměnných se také nezmění.

Závěrem poznamenejme, že lze vytvářet různé kombinace navržených postupů pro výpočet druhých i vyšších derivací. Na základě zmíněných vlastností není těžké takové kombinace promyslet a připravit.

Například, popsanou implementaci zpětného módu ve Fortranu 77 na stranách 63 až 66 lze velmi snadno upravit, aby počítala i druhé derivace, pokud se inspirujeme postupem navrženým v kapitole 4.1. Stačí totiž pouze zavést dvě nová pole `VDOT` (pole sdružených proměnných $\dot{\boldsymbol{v}}_i$) a `VBARDT` (pole sdružených proměnných k přidruženým proměnným, $\dot{\bar{\boldsymbol{v}}}_i$) a pozměnit již jednou pozměněné základní funkce (aby nyní kromě spočtení své hodnoty a zaznamenání se provedly také spočtení sdružených hodnot) a také subrutinu `RVRSWP` (aby vyhodnocovala i sdružené přidružené proměnné).

[15] Například je třeba zařídit, aby základní funkce věděla, s jakými poli má pracovat – zda s `V`, `VTMP`, `VDOT` nebo `VDOTTM`.

Shrnutí kapitoly 5

Naznačili jsme několik základních myšlenek a nápadů k implementaci automatického derivování. V žádném případě tento výčet není úplný – přesvědčili jsme se, že lze neustále nacházet nové způsoby a cesty vedoucí k různým modifikacím a zlepšením. Implementace automatického derivování je ostatně vždy závislá na vlastnostech konkrétního systému a použitém programovacím jazyce. Ani proto nemůže být výklad úplný.

Kapitola 6

Další vlastnosti, poznámky a zlepšení

V této kapitole uvedeme několik zajímavých postřehů a dalších rozšíření metod automatického derivování. V kapitole 6.1 omezíme používání zásobníku pro uchovávání informací o provedených základních operacích ve zpětném módu. V kapitole 6.3 odvodíme metody automatického derivování pro řídké Jacobiho a Hessovy matice. Závěrem se zmíníme o výpočtu třetích a vyšších derivací a také o požadavku spojitosti základních funkcí až do řádu d.

6.1 Omezení používání zásobníku

Ve zpětném módu automatického derivování je hodnota $\boldsymbol{v}_i$ před spočtením základní funkce $\boldsymbol{v}_i = \boldsymbol{\varphi}_i(\boldsymbol{u}_i)$ uložena na zásobník a později je z něj vyzvednuta, dokonce i pro velmi jednoduché operace. Ve srovnání s rychlým provedením některých základních operací mohou operace se zásobníkem trvat velmi dlouho.[1] Zamysleme se nyní nad tím, jak a kdy je možné snížit počet odkládaných (a obnovených) proměnných na zásobníku.

Za tímto účelem nám velmi dobře poslouží následující příklad. Uvažujme funkci

$$f(\boldsymbol{x}) = \exp(x_1 * x_n) + \sum_{i=2}^{n} x_i * x_{n-i+1}. \tag{6.1}$$

Její vyhodnocovací procedura a druhá část přidružené vyhodnocovací procedury získané nasčítávaným zpětným módem automatického derivování[2] jsou uvedeny na obrázku 6.1.

Přitom předpokládáme, že $\&v_1 = \&v_i$ a $\&\bar{v}_1 = \&\bar{v}_i$ pro $i = 2, \ldots n+1$, neboli všechna v_i, resp. všechna $\bar{v}_i$ pro $i = 1, \ldots, n+1$ sdílí společné umístění v paměti. Bez toho by výše uvedené vyhodnocovací procedury nedávaly správné výsledky. Tvrdíme, že uvedená přidružená vyhodnocovací procedura, i když neobsahuje žádnou práci se zásobníkem, dává stejné výsledky v_i a $\bar{v}_i$, $i = 1-n, \ldots, n+1$ jako

[1] Záleží ale na implementaci zásobníku a na konkrétní architektuře počítače.

[2] Přidružená vyhodnocovací procedura je přidružený soupis operací, který jsme přepsali například pomocí cyklů do přehlednějšího tvaru.

originální vyhodnocovací procedura

for $i = 1 \ldots n$
  $v_{i-n} = x_i$
$v_1 = v_0 * v_{1-n}$
$v_2 = \exp(v_1)$
for $i = 2 \ldots n$
  $v_{i+1} = v_{i+1} + v_{1-i} * v_{i-n}$
$y = v_{n+1}$

přidružená vyhodnocovací procedura (2. část)

$\bar{v}_{n+1} = \bar{y}$
for $i = n \ldots 2$
  $\bar{v}_{1-i} = \bar{v}_{1-i} + \bar{v}_{i+1} * v_{i-n}$
  $\bar{v}_{i-n} = \bar{v}_{i-n} + \bar{v}_{i+1} * v_{1-i}$
  $v_{i+1} = v_{i+1} - v_{1-i} * v_{i-n}$
$\bar{v}_1 = \bar{v}_2 * v_2$
$\bar{v}_0 = \bar{v}_0 + \bar{v}_1 * v_{1-n}$
$\bar{v}_{1-n} = \bar{v}_{1-n} + \bar{v}_1 * v_0$
for $i = n \ldots 1$
  $\bar{x}_i = \bar{v}_{i-n}$

Obrázek 6.1: Vyhodnocovací procedura a druhá část přidružené vyhodnocovací procedury pro funkci $f(\boldsymbol{x})$ z kapitoly 6.1

přidružený soupis operací, který bychom získali nasčítávaným zpětným módem automatického derivování při použití zásobníku. Důvodem je to, že v_i, $i \geq 2$ jsme spočetli na cestě zpět postupným odečítáním součinů od celkového součtu y a v_1 není pro výpočet přidruženého soupisu operací potřeba.

Obecněji lze říci, že pro některé základní funkce (např. exponenciálu $v_2 = \exp(v_1)$) nepotřebujeme v přidružené operaci znát hodnotu argumentu (v_1), ale výslednou hodnotu (v_2). Naopak pro některé jiné základní funkce, například násobení $v_1 = v_0 * v_{1-n}$, musíme v jejich přidružené operaci znát hodnoty argumentů (v_0, v_{1-n}), kdežto výsledná hodnota této základní operace je pro nás zbytečná (v_1). Totéž platí pro operaci $v = v + u * w$, kde ani hodnota starého v (před přepsáním), ani hodnota nového v (po přepsání) nemusí být uschována, protože její závislost je pouze lineární.

Uvedené poznatky a jednoduché úvahy můžeme shrnout do pravidla, že proměnné $\boldsymbol{v}_i$ jsou potřebné ve druhé části přidruženého soupisu operací, pokud jsou výsledkem například základní funkce převrácená hodnota, mocnina nebo exponenciála; nebo pokud se vyskytují jako nelineární argument v některé základní funkci.

Jednoduchý způsob, jak popsaný výběr naprogramovat, je připojit ke každému místu v paměti pro $\boldsymbol{v}_i$ příznak, který charakterizuje nutnost uložení příslušné hodnoty na zásobník, pokud by měla být přepsána. Tento příznak může být nastaven při spočtení hodnoty základní funkce nebo později při jejím použití jako (nelineární) argument v jiné základní funkci.

Ve zmíněné přidružené vyhodnocovací proceduře jsme potřebnou hodnotu v_2 spočetli postupným odčítáním příslušných součinů od celkového součtu. Vlastně jsme prováděli inverzní výpočet. Bohužel, při použití tohoto přístupu (nejen v tomto případě) mohou vzniknout významné zaokrouhlovací chyby.

Tyto chyby nemohou vzniknout, pokud se pro výpočet hodnot $\boldsymbol{v}_i$ ve druhé

části přidruženého soupisu operací použije stejný postup jako v původním výpočtu, tj. v první části přidruženého soupisu operací. V našem příkladě tak nemusíme v cyklu provádět operaci $v_{i+1} = v_{i+1} - v_{1-i} * v_{i-n}$, ale těsně po jeho skončení spočteme $v_1 = v_0 * v_{1-n}$ a $v_2 = \exp(v_1)$. Tím jsme eliminovali možná zaokrouhlování a zrychlili průběh, pokud n je velké.

Proto je nutné, abychom pro každou proměnnou $\boldsymbol{v}_i$ určili, jak s ní nakládat. Máme čtyři možnosti:

1. Proměnná není potřeba ve druhé části přidruženého soupisu operací.
2. Proměnná je uložena na zásobník.
3. Proměnná je ve druhé části spočtena pomocí inverzní funkce.
4. Proměnná je ve druhé části spočtena stejně jako v první části.

Pro výběr nejlepší strategie je nutná globální znalost celého soupisu operací. Je otázkou, nakolik se vyplatí navrhované přístupy implementovat a používat kvůli nákladům, které tím vzniknou navíc. Rozumnější přístup se zdá být uložit na zásobník všechno to, co se při zpětném průchodu může jen trochu hodit.

Další způsoby, kterými je možné snížit množství hodnot ukládaných na zásobník, lze nalézt například v [4].

6.2 Dvakrát provedený zpětný mód na lineární funkce

Uvažujme základní operace

$$\boldsymbol{v} = \boldsymbol{v} * c, \quad \text{resp.} \quad \boldsymbol{v} = \boldsymbol{v} + c \cdot \boldsymbol{u},$$

kde c je konstanta nezávislá na vstupních datech. Příslušné přidružené operace při nasčítávaném zpětném módu jsou

$$\bar{\boldsymbol{v}} = \bar{\boldsymbol{v}} * c, \quad \text{resp.} \quad \bar{\boldsymbol{u}} = \bar{\boldsymbol{u}} + c \cdot \bar{\boldsymbol{v}}.$$

Vidíme, že v prvním případě jsme dostali přesně stejnou operaci, jen s proměnnou $\bar{\boldsymbol{v}}$. Ve druhém případě jsme dostali operaci stejného tvaru, která by po ještě jednom provedení nasčítávaného zpětného módu a po substituci $\boldsymbol{v} = \bar{\bar{\boldsymbol{v}}}$ a $\boldsymbol{u} = \bar{\bar{\boldsymbol{u}}}$ byla stejná jako původní operace. Pro funkci složenou pouze z těchto základních funkcí budou platit analogické vlastnosti.

Předpokládejme, že máme nějakou metodu pro řešení soustavy lineárních rovnic

$$A\boldsymbol{y} = \boldsymbol{x}$$

s regulární maticí $A \in \mathbb{R}^{n\times n}$. Tedy $\boldsymbol{y} = \boldsymbol{F}(\boldsymbol{x}) = A^{-1}\boldsymbol{x}$. Aplikací nasčítávaného zpětného módu dostáváme

$$\bar{\boldsymbol{x}} = \bar{\boldsymbol{x}} + \bar{\boldsymbol{F}}(\boldsymbol{x}, \bar{\boldsymbol{y}}) = \bar{\boldsymbol{x}} + \bar{\boldsymbol{y}}A^{-1},$$

kde $\bar{\boldsymbol{x}}$ a $\bar{\boldsymbol{y}}$ jsou řádkové vektory délky n. Pokud byl na počátku vektor $\bar{\boldsymbol{x}}$ vynulován, vyřešili jsme tím systém lineárních rovnic

$$A^T \bar{\boldsymbol{x}}^T = \bar{\boldsymbol{y}}^T.$$

Uvědomme si, že jsme přitom nepočítali ani transpozici ani jakoukoli inverzní matici. Pokud bychom aplikovali nasčítávaný mód ještě jednou, dostaneme opět výsledek $A^{-1}\bar{\bar{\boldsymbol{x}}}$, v tomto případě však přičítaný do proměnné $\bar{\bar{\boldsymbol{y}}}$. Neboli $\bar{\bar{\boldsymbol{y}}} = \bar{\bar{\boldsymbol{y}}} + A^{-1}\bar{\bar{\boldsymbol{x}}}$.

6.3 Využití řídkosti Jacobiho a Hessovy matice při jejich výpočtu

V kapitole 3 a 4 jsme odvodili základní metody automatického derivování. Pro výpočet celé Jacobiho matice je vhodné použít přímý mód (pokud $n \leq m$) nebo zpětný mód (pokud $n > m$). Poměr časů pro výpočet $\boldsymbol{F}'(\boldsymbol{x})$ a $\boldsymbol{F}(\boldsymbol{x})$ je za této situace zhruba roven nejvýše $c \min(n, m)$.

Existuje celá řada modifikací, které snižují náklady pro výpočet derivace. Bývají založeny na využití řídkosti Jacobiho nebo Hessovy matice nebo na struktuře (postupu výpočtu) derivované funkce. V této kapitole uvedeme dvě z těchto modifikací. První z nich bude *dynamická*, která bude strukturu výpočtu $\boldsymbol{F}(\boldsymbol{x})$ (resp. řídkost $\boldsymbol{F}'(\boldsymbol{x})$) analyzovat přímo při vyhodnocování derivace a podle toho provádět další operace. Druhá modifikace bude *statická*, jejíž průběh bude pevně určen již při kompilaci. Následující odstavce si z důvodu značné rozsáhlosti tématu nekladou za cíl být detailním výkladem. Další podrobnosti lze najít například v [4].

6.3.1 Dynamické využití řídkosti

Nejprve zavedeme potřebné označení. Množinu indexů těch nezávislých proměnných $\boldsymbol{x}_j$, které netriviálně ovlivňují hodnotu $\boldsymbol{v}_i$, budeme značit $\mathcal{X}_i$. Tedy

$$\mathcal{X}_i = \{j \leq n;\ j - n \prec^* i\} \quad \text{pro } i = 1 - n, \ldots, l. \tag{6.2}$$

Je zřejmé, že

$$\mathcal{X}_{j-n} = \{j\} \quad \text{pro } j = 1, \ldots, n$$

a také

$$\mathcal{X}_k = \bigcup_{j \prec k} \mathcal{X}_j \quad \text{pro } k = 1, \ldots, l.$$

Jistě platí

$$\{j \leq n;\ \frac{\partial \boldsymbol{v}_k}{\partial \boldsymbol{x}_j} \neq 0\} \subseteq \mathcal{X}_k.$$

Snadno lze také ukázat, že pokud všechny základní funkce $\boldsymbol{\varphi}_j$ jsou unární nebo binární, pak

$$|\mathcal{X}_i| \leq i + 1 \quad \text{pro } i = 1, \ldots, l.$$

Výraz $|\mathcal{X}_i|$ znamená počet prvků množiny $\mathcal{X}_i$. Analogicky definujme $\mathcal{Y}_k$ jako množinu indexů závislých proměnných $\boldsymbol{y}_i$, které jsou netriviálně ovlivňovány proměnnou $\boldsymbol{v}_k$. Neboli

$$\mathcal{Y}_k = \{i \leq m;\ k \prec^* l - m + i\} \quad \text{pro } k = 1 - n, \ldots, l. \tag{6.3}$$

Analogicky i zde platí, že

$$\mathcal{Y}_j = \{j - l + m\} \quad \text{pro } j = l - m + 1, \ldots, l$$

a také

$$\mathcal{Y}_k = \bigcup_{j \succ k} \mathcal{Y}_j \quad \text{pro } k = 1 - n, \ldots, l - m.$$

Je rozumné předpokládat, že každá proměnná $\boldsymbol{v}_i$ závisí na nějaké nezávislé a ovlivňuje nějakou závislou proměnnou. Neboli pro $i = 1 - n, \ldots, l$ ať platí

$$\mathcal{X}_i \neq \emptyset \quad \text{a} \quad \mathcal{Y}_i \neq \emptyset.$$

Z definice $\mathcal{X}_i$ a $\mathcal{Y}_i$ je snadné odvodit, že

$$j \prec^* i \quad \Longrightarrow \quad \mathcal{X}_j \subseteq \mathcal{X}_i \quad \text{a} \quad \mathcal{Y}_j \supseteq \mathcal{Y}_i.$$

Jinak řečeno, každé $\mathcal{X}_k$ je obsaženo v nějaké $\mathcal{X}_{l-m+i}$ a každé $\mathcal{Y}_k$ je obsaženo v nějaké $\mathcal{Y}_{j-n}$. Můžeme tedy korektně psát a zároveň označit

$$\hat{n} = \max_{1 \leq i \leq m} p_i = \max_{1 \leq i \leq l} |\mathcal{X}_i| \tag{6.4}$$

a také

$$\hat{m} = \max_{1 \leq i \leq n} q_i = \max_{1-n \leq i \leq l-m} |\mathcal{Y}_i|, \tag{6.5}$$

kde $p_i = |\mathcal{X}_{l-m+i}|$ a $q_i = |\mathcal{Y}_{i-n}|$. Tyto hodnoty $\hat{n}$, resp. $\hat{m}$ lze interpretovat jako maximální počet nenulových prvků v každém řádku, resp. sloupci Jacobiho matice $\boldsymbol{F}'(\boldsymbol{x})$. Dále nadefinujeme hodnoty $\bar{n}$ a $\bar{m}$, které se spolu s hodnotami $\hat{n}$ a $\hat{m}$ budou objevovat v odhadech složitosti výpočtu Jacobiho matice.

$$\bar{n} = \frac{\sum_{k=1}^{l} |\mathcal{X}_k| TIME\{eval(\boldsymbol{\varphi}_k)\}}{\sum_{k=1}^{l} TIME\{eval(\boldsymbol{\varphi}_k)\}}. \tag{6.6}$$

Toto číslo $\bar{n}$ (average domain size) je vlastně vážený průměr hodnot $|\mathcal{X}_k|$ (velikostí (dimenzí) definičního oboru[3] funkce $\boldsymbol{v}_k(\boldsymbol{x})$) s vahami $TIME\{eval(\boldsymbol{\varphi}_k)\}$ a které obecně není přirozené číslo. Lze ho také interpretovat jako převrácenou hodnotu míry vzájemné nezávislosti závislých proměnných. Je ihned vidět, že také platí

$$\bar{n} \leq \hat{n} \leq n.$$

Analogicky definujme vážený průměr hodnot $|\mathcal{Y}_k|$ jako

$$\bar{m} = \frac{\sum_{k=1}^{l} |\mathcal{Y}_k| TIME\{eval(\boldsymbol{\varphi}_k)\}}{\sum_{k=1}^{l} TIME\{eval(\boldsymbol{\varphi}_k)\}}. \tag{6.7}$$

[3]Nepřesná formulace, ale pro názornou představu užitečná.

Tuto hodnotu $\bar{m}$ (average range size) lze interpretovat jako míra provázanosti závislých proměnných vzhledem k nezávislým proměnným. Dále platí

$$\bar{m} \leq \hat{m} \leq m.$$

Nyní již můžeme předvést algoritmy pro výpočet prvních a druhých derivací za použití množin $\mathcal{X}_i$ a $\mathcal{Y}_i$.

Přímý mód

Počítejme Jacobiho matici $\dot{\boldsymbol{Y}} = \dot{\boldsymbol{F}}(\boldsymbol{x}, \dot{\boldsymbol{X}}) = \boldsymbol{F}'(\boldsymbol{x}) \cdot \dot{\boldsymbol{X}}$ pro $\dot{\boldsymbol{X}} = \boldsymbol{I}_n$ pomocí soupisu operací na obrázku 3.1, který mírně upravíme. Skaláry $\dot{\boldsymbol{v}}_i$ nahradíme vektory $\dot{\boldsymbol{V}}_i$ délky n. Pokud $j \in \mathcal{X}_i$, pak j-tá komponenta $\dot{\boldsymbol{V}}_{i,j}$ vektoru $\dot{\boldsymbol{V}}_i$ může být nenulová. Pokud $j \notin \mathcal{X}_i$, nutně $\dot{\boldsymbol{V}}_{i,j} = 0$. Tak dostáváme soupis operací pro výpočet řídké Jacobiho matice, který je uveden na obrázku 6.2.

$$\left.\begin{array}{lll} \boldsymbol{v}_{i-n} & = \boldsymbol{x}_i & \\ \mathcal{X}_{i-n} & = \{i\} & \\ \dot{\boldsymbol{V}}_{i-n,i} & = 1 & \end{array}\right\} \; i = 1, \ldots, n$$

$$\left.\begin{array}{lll} \boldsymbol{v}_i & = \boldsymbol{\varphi}_i(\boldsymbol{u}_i) & \\ \mathcal{X}_i & = \cup_{k \prec i} \; \mathcal{X}_k & \\ \dot{\boldsymbol{V}}_{i,j} & = \sum_{k \prec i} \; \boldsymbol{c}_{ik} \cdot \dot{\boldsymbol{V}}_{k,j} \quad \text{pro } j \in \mathcal{X}_i & \end{array}\right\} \; i = 1, \ldots, l$$

$$\left.\begin{array}{lll} \boldsymbol{y}_{m-i} & = \boldsymbol{v}_{l-i} & \\ \dot{\boldsymbol{Y}}_{m-i} & = \dot{\boldsymbol{V}}_{l-i} & \end{array}\right\} \; i = m-1, \ldots, 0$$

Obrázek 6.2: Přímým módem automatického derivování odvozený soupis operací pro výpočet řídké Jacobiho matice

Poznamenejme, že toto není aditivní úkol. Lze ukázat, že pro vyhodnocení Jacobiho matice $\boldsymbol{F}'(\boldsymbol{x})$ podle soupisu operací na obrázku 6.2 platí

$$TIME\{eval(\boldsymbol{F}')\} \leq (1 + 2\bar{n}) TIME\{eval(\boldsymbol{F})\}.$$

Pokud se již nepotřebné proměnné $\boldsymbol{v}_i$ vzájemně přepisují, totéž mohou provádět příslušné vektory $\dot{\boldsymbol{V}}_i$ a množiny $\mathcal{X}_i$. (Podmínku přímé kompatibility předpokládáme ale splněnou.)

Zpětný mód

Pozměníme soupis operací na obrázku 3.5, aby efektivně vyhodnocoval řídké Jacobiho matice – množiny $\mathcal{Y}_i$ využijeme pro indikaci, kterou hodnotou se zabývat. Tak spočítáme $\bar{\boldsymbol{X}} = \bar{\boldsymbol{F}}(\boldsymbol{x}, \bar{\boldsymbol{Y}}) = \bar{\boldsymbol{Y}} \cdot \boldsymbol{F}'(\boldsymbol{x})$ pro $\bar{\boldsymbol{Y}} = \boldsymbol{I}_m$. Skaláry $\bar{\boldsymbol{v}}_i$ nahradíme vektory $\bar{\boldsymbol{V}}_i$. Jeho j-tá komponenta $\bar{\boldsymbol{V}}_{i,j}$ může být nenulová pouze tehdy, když $j \in \mathcal{Y}_i$.

$\mathcal{Y}_i$	$= \emptyset$	$\}$	$i = 1-n, \ldots, l$
$\bar{\boldsymbol{V}}_i$	$= 0$		
$\boldsymbol{v}_{i-n}$	$= \boldsymbol{x}_i$	$\}$	$i = 1, \ldots, n$
$\boldsymbol{v}_i$	$\rightarrow \mathcal{S}$	$\}$	$i = 1, \ldots, l$
$\boldsymbol{v}_i$	$= \boldsymbol{\varphi}_i(\boldsymbol{u}_i)$		
$\boldsymbol{y}_{m-i}$	$= \boldsymbol{v}_{l-i}$	$\}$	$i = m-1, \ldots, 0$
$\mathcal{Y}_{l-i}$	$= \{m-i\}$	$\}$	$i = 0, \ldots, m-1$
$\bar{\boldsymbol{V}}_{l-i,m-i}$	$= 1$		
$\boldsymbol{v}_i$	$\leftarrow \mathcal{S}$	$\}$ $j \prec i$	$i = l, \ldots, 1$
$\mathcal{Y}_j$	$= \mathcal{Y}_j \cup \mathcal{Y}_i$		
$\bar{\boldsymbol{V}}_{j,k}$	$= \bar{\boldsymbol{V}}_{j,k} + \boldsymbol{c}_{ij} \cdot \bar{\boldsymbol{V}}_{i,k}$ pro $k \in \mathcal{Y}_i$		
$\mathcal{Y}_i$	$= \emptyset$		
$\bar{\boldsymbol{V}}_i$	$= 0$		
$\bar{\boldsymbol{X}}_i$	$= \bar{\boldsymbol{V}}_{i-n}$	$\}$	$i = 1, \ldots, n$

Obrázek 6.3: Nasčítávaným zpětným módem automatického derivování odvozený soupis operací pro výpočet řídké Jacobiho matice

Tak dostáváme soupis operací pro výpočet řídké Jacobiho matice, který je uveden na obrázku 6.3.

Lze ukázat, že pro tento úkol platí

$$TIME\{eval(\boldsymbol{F}')\} \leq (1 + 3\bar{m})TIME\{eval(\boldsymbol{F})\}.$$

Upozorněme však na to, že do tohoto odhadu (ani do odhadu pro přímý mód v předcházejícím odstavci) nejsou zahrnuty náklady operací s množinami $\mathcal{X}_i$ a $\mathcal{Y}_i$, které mohou výsledek negativně ovlivnit. Pokud program pro vyhodnocení $\boldsymbol{F}(\boldsymbol{x})$ neobsahuje žádné větvení, lze množiny $\mathcal{X}_i$ a $\mathcal{Y}_i$ určit při kompilaci a tak napevno naprogramovat úpravy pro vyhodnocení řídké Jacobiho matice pro jeden konkrétní případ. Výsledky mohou být velmi zajímavé. Pokud se již nepotřebné proměnné $\boldsymbol{v}_i$ vzájemně přepisují, totéž mohou provádět příslušné vektory $\bar{\boldsymbol{V}}_i$ a množiny $\mathcal{Y}_i$. (Podmínku přímé kompatibility předpokládáme ale splněnou.)

Druhé derivace

Soupis operací pro výpočet druhých derivací na obrázku 4.1 lze upravit podobným způsobem, jaký jsme naznačili v předcházejících dvou odstavcích. Dostaneme tak soupis operací, jehož první část je totožná se soupisem operací na obrázku 6.2. Ve druhé části sestavujeme množiny $\bar{\mathcal{X}}_i$, které charakterizují, jaké prvky vektoru $(\partial \bar{\boldsymbol{v}}_i / \partial \boldsymbol{x}_j)_{j=1,\ldots,n}$ jsou nenulové a je s nimi třeba počítat. Přitom je zvláštní pozor-

nost upřena na operace násobení a nelineární operace, pro které je druhá derivace nenulová. Ukázky soupisů operací a další podrobnosti lze najít v [4].

Popsali jsme, jak lze snížit poměr času výpočtu Jacobiho matice $\boldsymbol{F}'(\boldsymbol{x})$ vzhledem k času výpočtu $\boldsymbol{F}(\boldsymbol{x})$ až na zhruba $c\min(\bar{n},\bar{m})$. Díky časté manipulaci s indexovými množinami $\mathcal{X}_i$ a $\mathcal{Y}_i$ je však na místě otázka, jestli se to vůbec vyplatí.

6.3.2 Statické využití řídkosti

V této kapitole popíšeme modifikaci automatického derivování, pomocí které lze velmi rychle spočíst řídkou Jacobiho nebo Hessovu matici. Tento způsob budeme nazývat *maticovou kompresí*. Narozdíl od předcházející kapitoly však potřebujeme předem znát množiny $\mathcal{X}_{l-m+i}$ pro $i = 1,\ldots,m$ nebo $\mathcal{Y}_{j-n}$ pro $j = 1,\ldots,n$ (v závislosti na druhu komprese). To není příliš omezující, protože strukturu Jacobiho matice (čili množiny $\mathcal{X}_{l+m-i}$ a $\mathcal{Y}_{j-n}$) alespoň tušíme.[4] Nejprve popíšeme *řádkovou maticovou kompresi*, později naznačíme její přesnou analogii, *sloupcovou maticovou kompresi*. Dále ukážeme *kombinovanou* řádkovou a sloupcovou maticovou kompresi. Nakonec se zmíníme o maticové kompresi při výpočtu *druhých* derivací.

Řádková maticová komprese

Uvažujme matici $S \in \mathbb{R}^{n\times p}$. Dosadíme-li $\dot{\boldsymbol{X}} = S$ (tedy matice směrů), přímým módem automatického derivování spočteme

$$B = \boldsymbol{F}'(\boldsymbol{x}) \cdot S \ \in \mathbb{R}^{m\times p}.$$

Označíme-li $\boldsymbol{e}_i$ jako i-tý vektor kartézské báze v $\mathbb{R}^m$, pro i-tý řádek matice B platí

$$\boldsymbol{b}_i^T = \boldsymbol{e}_i^T \cdot B = \boldsymbol{e}_i^T \cdot \boldsymbol{F}'(\boldsymbol{x}) \cdot S \ = \nabla \boldsymbol{F}_i(\boldsymbol{x}) \cdot S \ \in \mathbb{R}^p. \tag{6.8}$$

Odvodíme podmínky, za kterých lze (pro každé $i = 1,\ldots,n$) ze spočteného vektoru $\boldsymbol{b}_i^T$ zrekonstruovat i-tý řádek Jacobiho matice $\boldsymbol{F}'(\boldsymbol{x})$, tj. gradient i-té složky funkce $\boldsymbol{F}$ v bodě $\boldsymbol{x}$.

Nutnou podmínkou je, aby délka p vektoru $\boldsymbol{b}_i$ nebyla menší než počet neznámých hodnot[5] v i-tém řádku Jacobiho matice $\boldsymbol{F}'(\boldsymbol{x})$. Požadujeme tedy

$$p \geq p_i = |\mathcal{X}_{l-m+i}|. \tag{6.9}$$

Chceme zrekonstruovat celou matici $\boldsymbol{F}'(\boldsymbol{x})$ pomocí stejné matice S, takže vztah (6.9) požadujeme splněný pro každé $i = 1,\ldots,m$. Neboli chceme, aby platilo

$$p \geq \hat{n},$$

kde $\hat{n} = \max_{1\leq i\leq m} |\mathcal{X}_{l-m+i}|$ je definováno v (6.4).

[4] Je rozumné přistupovat k problému struktury Jacobiho matice opatrně. Pokud si nejsme jistí nulovostí nebo nenulovostí určitého prvku, považujme ho raději za nenulový.

[5] Nepřesně můžeme říci „nenulových hodnot“.

Pro jednoduchost budeme dále značit matici $\boldsymbol{F}'(\boldsymbol{x})$ jako A. Strukturu i-tého řádku Jacobiho matice $\boldsymbol{F}'(\boldsymbol{x})$ charakterizuje indexová množina $\mathcal{X}_{l-m+i}$. Přepíšeme vztah (6.8) – vynecháme členy, které nutně musí být kvůli řídkosti nulové. Pro $i = 1, \ldots, m$ tak dostáváme

$$\boldsymbol{b}_i^T = \boldsymbol{e}_i^T \cdot A \cdot S \ = \boldsymbol{a}_i^T \cdot S_i, \tag{6.10}$$

kde

$$\boldsymbol{a}_i = (\boldsymbol{e}_i^T \cdot A \ \cdot \boldsymbol{e}_j)_{j \in \mathcal{X}_{l-m+i}} \in \mathbb{R}^{p_i} \tag{6.11}$$

a

$$S_i = (\boldsymbol{e}_j^T \cdot S)_{j \in \mathcal{X}_{l-m+i}} \in \mathbb{R}^{p_i \times p} \tag{6.12}$$

a kde $\boldsymbol{e}_i$ je i-tý vektor kartézské báze[6].

Tím jsme dostali soustavu p lineárních rovnic

$$S_i^T \cdot \boldsymbol{a}_i = \boldsymbol{b}_i \tag{6.13}$$

pro p_i neznámých $\boldsymbol{a}_i$. Protože $p_i \leq p$, je tato soustava přeurčená; díky (6.8) je ale jistě konzistentní. Hodnoty $\boldsymbol{a}_i$ budeme moci jednoznačně spočíst, bude-li všech p_i sloupců matice S_i^T lineárně nezávislých. Budeme tedy volit takovou S, aby byl tento požadavek splněn. Podotkněme, že pro rekonstrukci celé Jacobiho matice musíme vyřešit m takových soustav lineárních rovnic.

Nyní uvedeme několik možností, jak volit matici S.

CPR metoda pro volbu matice S (Curtis, Powell, Reid)
Označíme zobrazení

$$c: \ \{1, \ldots, n\} \longrightarrow \{1, \ldots, p\}$$

takové, že pro každé j, $k \leq n$ platí

$$\mathcal{Y}_{j-n} \cap \mathcal{Y}_{k-n} \neq \emptyset \quad \Longrightarrow \quad c(j) \neq c(k). \tag{6.14}$$

Toto zobrazení rozděluje sloupce Jacobiho matice $\boldsymbol{F}'(\boldsymbol{x})$ do p *skupin* podle příslušnosti do množiny $c^{-1}(i)$ pro jednotlivá $i = 1, \ldots, p$. Každé skupině přiřadíme *barvu* i. Požadavek (6.14) mimo jiné znamená, že pokud j a k patří do stejné skupiny (mají stejnou barvu), pak v žádném řádku matice $\boldsymbol{F}'(\boldsymbol{x})$ nemůže být nenulový prvek v j-tém a zároveň v k-tém sloupci.

Označme $\mathcal{G}_c = (\mathcal{V}_c, \mathcal{E}_c)$ neorientovaný graf, jehož vrcholy jsou

$$\mathcal{V}_c = \{1, \ldots, n\}$$

a pro jehož hrany platí

$$(j,k) \in \mathcal{E}_c \quad \Longleftrightarrow \quad \mathcal{Y}_{j-n} \cap \mathcal{Y}_{k-n} \neq \emptyset.$$

Lze ukázat, že nalezení zobrazení c je ekvivalentní úloze obarvit vrcholy grafu $\mathcal{G}_c$ tak, aby žádné dva vrcholy spojené hranou neměly stejnou barvu. Chceme,

[6] Velikosti vektorů $\boldsymbol{e}_i$ jsou takové, aby všechny vztahy výše uvedené dávaly smysl. Pro přehlednost je raději neuvádíme.

aby počet barev byl co nejmenší (vzápětí uvidíme proč). Minimální možný počet barev p, kterými lze graf obarvit, se nazývá *chromatické číslo* $\chi(\mathcal{G}_c)$ grafu $\mathcal{G}_c$ (chromatic number) a jeho určení je velmi těžké. Existují heuristické algoritmy pro určení nepatrně vyšších hodnot než $\chi(\mathcal{G}_c)$. Tyto jsou pro naše účely dostatečné.

Volme matici S způsobem

$$S = \left(\boldsymbol{e}_{c(j)}^T\right)_{j=1,\ldots,n} \in \mathbb{R}^{n\times p}, \tag{6.15}$$

kde $\boldsymbol{e}_i$ je i-tý vektor kartézské báze v $\mathbb{R}^p$. Submatice S_i definovaná vztahem (6.12) má tvar

$$S_i = \left(\boldsymbol{e}_{c(j)}^T\right)_{j\in\mathcal{X}_{m-l+i}} \in \mathbb{R}^{p_i\times p}. \tag{6.16}$$

Každý řádek matice S_i obsahuje právě jednu jedničku, ostatní hodnoty jsou nulové. Sporem se vztahem (6.14) lze snadno ukázat, že každý sloupec matice S_i obsahuje nejvýše jednu jedničku, ostatní hodnoty jsou opět nuly. Proto má soustava (6.13) jediné řešení

$$\boldsymbol{a}_{i,j} = \boldsymbol{e}_i^T \cdot A \cdot \boldsymbol{e}_j = \boldsymbol{e}_i^T \cdot B \cdot \boldsymbol{e}_{c(j)} = \boldsymbol{b}_{i,c(j)}. \tag{6.17}$$

Neboli každý nenulový prvek Jacobiho matice $\boldsymbol{F}'(\boldsymbol{x}) = A$ lze najít mezi nenulovými prvky matice B.

NR metoda pro volbu matice S (Newsam, Ramsdell)
Volme matici S způsobem

$$S = \left(\lambda_j^{k-1}\right)_{j=1,\ldots,n}^{k=1,\ldots,p} \in \mathbb{R}^{n\times p} \tag{6.18}$$

pro $p = \hat{n}$. Odtud

$$S_i = \left(\lambda_j^{k-1}\right)_{j\in\mathcal{X}_{l-m+i}}^{k=1,\ldots,p} \in \mathbb{R}^{p_i\times p}. \tag{6.19}$$

Abychom dostali jediné řešení $\boldsymbol{a}_i$, musí mít matice S_i maximální hodnost. K tomu stačí, aby (pro každé $i = 1,\ldots,m$) všech p_i hodnot λ_j (pro $j \in \mathcal{X}_{l-m+i}$) bylo navzájem různých. Lze volit například

$$\lambda_j = 2\frac{j-1}{n-1} - 1 \quad \text{pro } j = 1,\ldots,n.$$

Pokud je n příliš velké, mohou být hodnoty λ_j příliš blízké. Proto je rozumné volit $\lambda_j = \mu_{c(j)}$ pro navzájem různé hodnoty μ_k, $k = 1,\ldots,\chi(\mathcal{G}_c)$, a pro „obarvující“ funkci c definovanou vztahem (6.14). Potom má matice S_i opět maximální hodnost.

K vyřešení soustavy (6.13) s volbou matice S podle (6.18) je zapotřebí pouze $2.5p_i^2$ operací, jak je ukázáno v [8]. Bohužel, v tomto případě číslo podmíněnosti matice $\hat{S}_i$ roste exponenciálně s její velikostí.[7]

[7] Matici S_i rozdělíme: prvních p_i sloupců označíme jako matici $\hat{S}_i$ a zbývajících $p-p_i$ sloupců označíme jako matici $\tilde{S}_i$. Neboli $S_i = (\hat{S}_i, \tilde{S}_i)$.

Označme T_k Čebyševův polynom stupně k na intervalu $[-1;1]$. Pak lze matici S položit také jako

$$S = \left(T_{k-1}(\mu_{c(j)})\right)_{j=1,\dots,n}^{k=1,\dots,p} \in \mathbb{R}^{n\times p}. \tag{6.20}$$

Při této volbě je číslo podmíněnosti mnohem menší ve srovnání s (6.18), ale soustavu (6.13) již nelze řešit tak levně.

Sloupcová maticová komprese

Pomocí zpětného módu automatického derivování umíme spočítat matici

$$C^T = W^T \cdot \boldsymbol{F}'(\boldsymbol{x}) \in \mathbb{R}^{q\times n} \quad \text{pro} \quad W \in \mathbb{R}^{m\times q}.$$

Všechny vztahy odvozené v předcházejících odstavcích pro řádkovou maticovou kompresi můžeme tedy aplikovat na matici $(\boldsymbol{F}'(\boldsymbol{x}))^T = A^T \in \mathbb{R}^{n\times m}$.

Analogicky tedy definujme zobrazení

$$d:\ \{1,\dots,m\} \longrightarrow \{1,\dots,q\},$$

aby pro každé j, $k \le m$ platilo

$$\mathcal{X}_{l-m+j} \cap \mathcal{X}_{l-m+k} \neq \emptyset \quad \Longrightarrow \quad d(j) \neq d(k).$$

V tomto případě bychom barvili graf $\mathcal{G}_r = (\mathcal{V}_r, \mathcal{E}_r)$ s vrcholy $\mathcal{V}_r = \{1,\dots,m\}$ a hranami $\mathcal{E}_r$, pro které platí

$$(j,k) \in \mathcal{E}_r \quad \Longleftrightarrow \quad \mathcal{X}_{l-m+j} \cap \mathcal{X}_{l-m+k} \neq \emptyset.$$

Označme $\boldsymbol{e}_i$ jako i-tý vektor kartézské báze pro $\mathbb{R}^q$. Pak CPR metoda dává

$$W = \left(\boldsymbol{e}_{d(j)}^T\right)_{j=1,\dots,m} \in \mathbb{R}^{m\times q},$$

zatímco NR metodou dostáváme pro $q = \hat{m}$

$$W = \left(\mu_{d(j)}^{k-1}\right)_{j=1,\dots,m}^{k=1,\dots,q},$$

respektive

$$W = \left(T_{k-1}(\mu_{d(j)})\right)_{j=1,\dots,m}^{k=1,\dots,q}.$$

Soustavy lineárních rovnic, které je nutné řešit, mají tvar

$$W_j^T \cdot \boldsymbol{a}_j = \boldsymbol{c}_j = C^T \cdot \boldsymbol{e}_j \in \mathbb{R}^q,$$

kde

$$\boldsymbol{a}_j = (\boldsymbol{e}_k^T \cdot A \ \cdot \boldsymbol{e}_j)_{k\in\mathcal{Y}_{j-n}} \in \mathbb{R}^{q_i}$$

a

$$W_j = (\boldsymbol{e}_k^T \cdot W)_{k\in\mathcal{Y}_{j-n}} \in \mathbb{R}^{q_i\times q}.$$

Další úvahy a vlastnosti platí analogicky jako při řádkové kompresi.

Kombinovaná řádková a sloupcová maticová komprese

Ani řádková ani sloupcová maticová komprese nepomůže v případě Jacobiho matice tvaru

$$\boldsymbol{F}'(\boldsymbol{x}) = A = \begin{pmatrix} \delta_1 & 0 & & 0 & \alpha_1 \\ 0 & \delta_2 & & 0 & \alpha_2 \\ & & \ddots & & \vdots \\ 0 & 0 & & \delta_{n-1} & \alpha_{n-1} \\ \beta_1 & \beta_2 & \dots & \beta_{n-1} & \delta_n \end{pmatrix},$$

protože $n = \hat{n} = \chi(\mathcal{G}_c) = \chi(\mathcal{G}_r) = \hat{m} = m$. Naproti tomu však

$$A \cdot \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ \vdots & \vdots \\ 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} \delta_1 & \alpha_1 \\ \delta_2 & \alpha_2 \\ \vdots & \vdots \\ \delta_{n-1} & \alpha_{n-1} \\ \sum_{i=1}^{n-1} \beta_i & \delta_n \end{pmatrix}$$

a zároveň

$$\begin{pmatrix} 0 & 0 & \dots & 0 & 1 \end{pmatrix} \cdot A = \begin{pmatrix} \beta_1 & \beta_2 & \dots & \beta_{n-1} & \delta_n \end{pmatrix}.$$

Jak vidíme, ve třech vektorech derivací[8] jsou uloženy všechny nenulové prvky Jacobiho matice $\boldsymbol{F}'(\boldsymbol{x})$.

Chtěli bychom tedy obecně najít postup, jak sestrojit matice $S \in \mathbb{R}^{n\times p}$ a $W \in \mathbb{R}^{m\times q}$, aby se z matic $B = A \cdot S \in \mathbb{R}^{m\times p}$ a $C = A^T \cdot W \in \mathbb{R}^{n\times q}$ daly zrekonstruovat nenulové prvky matice A. Navíc tak, aby součet $p + q$ byl co nejmenší.

Stručně naznačíme jeden postup pro získání matic S a W a hodnot p a q. Podrobnosti lze najít například v [4], [7] nebo [9]. Uvažujme Jacobiho matici $A = \boldsymbol{F}'(\boldsymbol{x})$, respektive její permutaci $A = Q \cdot \boldsymbol{F}'(\boldsymbol{x}) \cdot P$. Matici A rozdělíme na blokově dolní trojúhelníkovou matici A_r a blokově horní trojúhelníkovou matici $A_c = A - A_r$ tak, aby matice A_r obsahovala poslední blokový řádek matice A. Poznamenejme, že bloky těchto matic nemusí být nutně čtvercové. Matici S zvolíme jako matici pro kompresi a následnou rekonstrukci matice A_r, matici W pro kompresi a následnou rekonstrukci matice A_c.[9] Automatickým derivováním spočteme matice $B = A \cdot S$ a $C = A^T \cdot W$. Protože poslední blokový řádek matice B nezávisí na matici A_c, lze z něj zrekonstruovat poslední blokový řádek matice A_r, tedy i A. Potom opravíme hodnotu matice C (zhruba řečeno, od matice C odečteme ztransponovaný poslední blokový řádek matice A vynásobený zprava W) a vyhodnotíme poslední blokový sloupec matice A_c. Dál postupujeme analogicky – upravíme hodnotu B a spočteme předposlední blokový řádek A_r, upravíme C a spočteme předposlední blokový sloupec A_c atd. Tímto způsobem spočteme celou matici A. Podle použité metody komprese, hodnota p (resp. q)

[8] Dva vektory derivací jsme získali přímým módem a jeden vektor zpětným módem automatického derivování.

[9] Lze například použít CPR nebo NR metodu.

může být buď chromatické číslo grafu $\mathcal{G}_{c,A_r}$ (resp. $\mathcal{G}_{r,A_c}$) příslušného matici A_r (resp. A_c) nebo maximální počet nenulových číslic ve všech řádcích matice A_r (resp. sloupcích matice A_c).

Ve [4] a [9] je popsán algoritmus pro nalezení matic A_r a A_c a hodnoty q pro zadané p, resp. nalezení hodnoty p pro zadané q. Dále jsou v těchto knihách uvedeny rady a postupy pro optimální výběr kombinace hodnot p a q.

Výpočet druhé derivace pomocí maticové komprese

Řídkost (struktura) Jacobiho matice $\boldsymbol{F}'(\boldsymbol{x})$ ovlivňuje i strukturu (řídkost) tenzoru druhých derivací $\boldsymbol{F}''(\boldsymbol{x})$. Proto matice S a W, pomocí kterých počítáme kompresi $\boldsymbol{F}'(\boldsymbol{x})$, lze použít i pro výpočet komprese $\boldsymbol{F}''(\boldsymbol{x})$ a následně celé $\boldsymbol{F}''(\boldsymbol{x})$.

Například, i-tá složka $\boldsymbol{F}_i$ funkce $\boldsymbol{F}$ závisí pouze na těch $\boldsymbol{x}_j$, pro které $j \in \mathcal{X}_{l-m+i}$. Proto všechny nenulové prvky Hessovy matice $\boldsymbol{F}_i''(\boldsymbol{x})$ jsou obsaženy v matici

$$\hat{\boldsymbol{F}}_i''(\boldsymbol{x}) = \left(\frac{\partial^2 \boldsymbol{F}_i}{\partial \boldsymbol{x}_k \partial \boldsymbol{x}_j}(\boldsymbol{x}) \right)_{k \in \mathcal{X}_{l-m+i}}^{j \in \mathcal{X}_{l-m+i}} \in \mathbb{R}^{p_i \times p_i}.$$

Definujme S_i stejně jako v předchozích odstavcích, tj. vztahem (6.12). Výraz

$$S^T \cdot \boldsymbol{F}_i''(\boldsymbol{x}) \cdot S \ = S_i^T \cdot \hat{\boldsymbol{F}}_i''(\boldsymbol{x}) \cdot S_i$$

lze spočíst pomocí automatického derivování popsaného v kapitole 4. Z tohoto výrazu je již možné určit matici $\hat{\boldsymbol{F}}_i''(\boldsymbol{x})$, resp. celou $\boldsymbol{F}_i''(\boldsymbol{x})$.

Naznačíme ještě jeden způsob výpočtu $\boldsymbol{F}''(\boldsymbol{x})$ maticovou kompresí. Pro $j = 1, \ldots, n$ označme

$$\boldsymbol{F}^{(j)} = \frac{\partial \boldsymbol{F}'}{\partial \boldsymbol{x}_j}(\boldsymbol{x}) \in \mathbb{R}^{m \times n}.$$

Automatickým derivováním můžeme pro $i = 1, \ldots, q$ a $k = 1, \ldots, p$ spočíst vektory

$$\boldsymbol{w}_i^T \cdot \boldsymbol{F}''(\boldsymbol{x}) \cdot \boldsymbol{s}_k,$$

ze kterých sestavíme

$$W^T \cdot \boldsymbol{F}^{(j)}(\boldsymbol{x}) \cdot S \quad \text{pro } j = 1, \ldots, n.$$

Z těchto hodnot můžeme díky struktuře $\boldsymbol{F}^{(j)}(\boldsymbol{x})$ zrekonstruovat matice $\boldsymbol{F}_i''(\boldsymbol{x}) \cdot S$ a odtud již $\boldsymbol{F}_i''(\boldsymbol{x})$ pro $i = 1, \ldots, m$. Podrobnosti o těchto metodách lze nalézt například v [4].

Poznamenejme, že určení hodnoty $\hat{n}$, resp. $\hat{m}$ (pro NR metodu) je mnohem snazší než určení chromatického čísla $\chi(\mathcal{G}_c)$, resp. $\chi(\mathcal{G}_r)$ nebo jeho aproximace (pro CPR metodu). Na druhou stranu, při dekompresi v případě NR metody musíme řešit soustavy lineárních rovnic pro rekonstrukci matice A.

Jak se zmiňuje mnoho autorů, náklady pro dekompresi spočtené matice jsou vzhledem k nákladům na její spočtení velmi malé.

Dynamické využití řídkosti Jacobiho matice, které jsme popsali v kapitole 6.3.1, se zdá být efektivnější než statické, neboť $\bar{n} \leq \hat{n} \leq \chi(\mathcal{G}_c)$ a $\bar{m} \leq \hat{m} \leq \chi(\mathcal{G}_r)$. Kvůli značné manipulaci s množinami $\mathcal{X}_i$, resp. $\mathcal{Y}_j$ bude však tato lepší efektivita zřetelná až pro velké hodnoty $\hat{n} - \bar{n}$, resp. $\hat{m} - \bar{m}$.

6.4 Výpočet vyšších derivací

V kapitole 3 jsme odvodili transformaci soupisu operací pro výpočet hodnot funkce $\boldsymbol{F}$ v bodě $\boldsymbol{x}$ na soupis operací, který bude počítat navíc i její první derivace. Toho jsme využili v kapitole 4, kde jsme popsali transformaci soupisu operací až na výpočet druhých derivací. Stejným způsobem lze postupovat dále k vyšším derivacím. Přitom můžeme libovolně kombinovat přímý i zpětný mód automatického derivování. Pro přehlednost je vhodné provést zpětný mód pouze jednou a na vzniklý soupis operací aplikovat několikrát za sebou přímý mód.

Implementaci automatického derivování pro výpočet vyšších derivací lze provést podobným vnořováním, k čemuž se výborně hodí technika přetížení operátorů a funkcí. Buďme přitom ale nanejvýš pozorní a dobře si rozmysleme, co a jak vlastně počítáme.

Závěrem konstatujme, že výpočet třetích a vyšších derivací se v praktických úlohách téměř neprovádí. Přesto lze o tomto tématu v literatuře najít velké množství zajímavých úvah.

6.5 Diferencovatelnost základních funkcí

Připomeňme ještě jednu vlastnost funkcí více proměnných.

Věta 6.1
Nechť funkce $f : \mathbb{R}^n \to \mathbb{R}$ má totální diferenciál v bodě a. Pak existují všechny parciální derivace funkce f v bodě a a jsou konečné. □

V kapitole 2.2 jsme uvalili předpoklady na základní funkce $\boldsymbol{\varphi}_i$. Požadovali jsme, aby byly d-krát spojitě diferencovatelné na otevřené množině D_i v $\mathbb{R}^n$, tj. splnění vlastnosti 2.1.

Tento předpoklad lze zeslabit na požadavek spojitosti derivací základních funkcí $\boldsymbol{\varphi}_i$ až do řádu $n-1$ a současně existence totálního diferenciálu pro $(n-1)$-ní derivaci základních funkcí $\boldsymbol{\varphi}_i$, oboje na otevřené množině D_i v $\mathbb{R}^n$ (stejné jako za původních předpokladů). Pravidlo řetězení (věta 2.2) platí totiž i za těchto slabších předpokladů a všechna odvození automatického derivování lze provést naprosto analogicky. Dostáváme tak stejné výsledky, jaké jsme získali za původních předpokladů.

Pokud nejsou splněny ani tyto slabší předpoklady na základní funkce $\boldsymbol{\varphi}_i$, nelze obecně zaručit platnost pravidla řetězení a nelze tedy automatické derivování odvodit. V praktických úlohách ostatně tato situace není pravděpodobná, protože základní funkce bývají na D_i spojité až do řádu d.

Je možné zavést zobecněné derivace, pro jejichž výpočet lze také použít jisté techniky automatického derivování. Viz například [4].

6.6 Poznámka o základních funkcích

Předpokládejme, že derivovaný program provádí (mimo jiné) vyhodnocení funkce

$$g(y) = \begin{cases} \exp(y)\frac{1-\cos(y)}{y^2} & \text{pro } y \neq 0, \\ 1/2 & \text{pro } y = 0. \end{cases}$$

Pro $y = 0$ bychom automatickým derivováním kvůli přiřazení $g(0) = 1/2$ dostali *nesprávný* výsledek $g'(0) = 0$. Správná hodnota je přitom $g'(0) = 1/2$. Tomuto problému se lze vyhnout, budeme-li definovat funkci g jako základní funkci $\boldsymbol{\varphi}$. Ve výpočtu její derivace bychom opět testovali, zda $y = 0$. V tom případě bychom přiřadili $g'(0) = \boldsymbol{\varphi}'(0) = 1/2$.

Kapitola 7

Implementace automatického derivování v systému UFO

V této kapitole stručně popíšeme interaktivní systém UFO (Universal Functional Optimization system) pro optimalizaci funkcí. Dále navrhneme implementaci automatického derivování do tohoto systému s ohledem na jeho specifické vlastnosti a použití.

7.1 Stručný popis systému UFO

Systém UFO byl vyvinut v Ústavu informatiky Akademie věd České republiky. Bližší informace o tomto systému lze získat na internetové adrese `http://www.cs.cas.cz/vvvvedci/luksan/ufo.html`, jeho podrobný popis je uveden v [5]. Pro operační systémy MS-DOS a Microsoft Windows je možné jej stáhnout z adresy `http://www.cs.cas.cz/vvvvedci/luksan/ufowin.zip`.

Systém UFO je možné použít pro

- formulaci a řešení konkrétních optimalizačních problémů (úloh);
- přípravu speciálních optimalizačních metod založených na metodách již naimplementovaných;
- návrh a testování nových optimalizačních metod.

Nejobecnější problém, který lze systémem UFO řešit, je (lokální i globální) minimalizace funkce $F : \mathbb{R}^n \to \mathbb{R}$ na množině $X \subseteq \mathbb{R}^n$. Tvar této funkce je možné blíže specifikovat, například jako součet mocnin hodnot $|f_k(x)|$, maximum z hodnot $|f_k(x)|$ atd. Množinu X lze zvolit jako $X = \mathbb{R}^n$ nebo je možné ji charakterizovat jako podmnožinu $\mathbb{R}^n$ pomocí vazbových podmínek (rovnic a nerovnic) s lineárními i nelineárními (hladkými) funkcemi.

Díky modulární struktuře systému UFO je možné použít celou řadu optimalizačních metod, které lze sestavit na základě zadaných požadavků. Základní třídy implementovaných metod jsou například metody s proměnnou metrikou nebo Newtonovy metody. Konkrétní metoda je určena specifikací dalších parametrů (například směr a délka kroku).

Řešení optimalizačního problému probíhá ve čtyřech fázích.

1. Specifikace optimalizačního problému a výběr metody. Tyto údaje se popíší pomocí řídícího jazyka systému UFO (UFO CL, UFO control language) ve vstupním souboru. O tomto jazyce se zmíníme později. Vstupní soubor (má příponu `UFO`) buď napíše uživatel, nebo je vytvořen pomocí dialogového módu, tj. na základě otázek systému a odpovědí uživatele (v tom případě uživatel nemusí vůbec tušit, že UFO CL a vstupní soubor existuje). Asi nejlepší variantou vzniku vstupního souboru je kombinovaný mód, kdy vstupní soubor je předpřipraven uživatelem a následně doplněn v dialogovém módu.

2. Vstupní soubor zpracujeme UFO preprocesorem. Podle uvedených požadavků vzniká řídící program ve Fortranu 77, který bude řešit zadaný problém pomocí specifikovaných metod.

3. Tento vygenerovaný program je přeložen a slinkován s knihovními podprogramy. Pro překlad se pod operačními systémy MS-DOS a Microsoft Windows používá Microsoft Fortran Power Station version 1.0, který není součástí balíku vystaveného na internetu. Lze ale použít jiný překladač Fortranu 77.

4. Připravený program je spuštěn a jeho průběhem získáváme řešení našeho optimalizačního problému.

Toto zpracování je možné provést zavoláním několika dávkových souborů anebo přímo z integrovaného vývojového prostředí, které bylo pro systém UFO (pod systémy MS-DOS a Microsoft Windows) vytvořeno.

Protože systém UFO je naprogramován v jazyce Fortran 77, lze ho přeložit a používat například i pod operačními systémy typu UNIX.

Máme možnost zjistit množství zajímavých údajů a hodnot získaných během řešení zadaného optimalizačního problému (například hodnoty nezávislých proměnných, funkčních hodnot, gradientu, směru atd.) – nejenom konečné výsledky, ale i hodnoty získané během jednotlivých iterací. Tato data lze znázornit na monitoru v textové nebo grafické podobě nebo je lze uložit do textového souboru (standardní přípona je `OUT`). Specifikace požadovaného výstupu je uvedena ve vstupním souboru.

Pro ilustraci ukážeme jednoduchý příklad. Chceme lokálně minimalizovat Rosenbrockovu funkci

$$F(x) = 100(x_1^2 - x_2)^2 + (x_1 - 1)^2,$$

přičemž počáteční bod pro iterace nechť je $x = (-1.2, 1.0)$. Připravíme vstupní soubor `P.UFO`, který bude obsahovat

```
$SET(INPUT)
  X(1)=-1.2D0; X(2)=1.0D0
$ENDSET
$SET(FMODELF)
```

```
  FF = 1.0D2*(X(1)**2-X(2))**2+(X(1)-1.0D0)**2
$ENDSET
$NF=2
$NOUT=1
$BATCH
$STANDARD
```

Tím jsme nastavili počáteční bod a minimalizovanou funkci a specifikovali jsme tvar výstupu. Zpracujeme-li tento soubor systémem UFO, dostáváme soubor `P.OUT` s požadovanými výsledky.

```
  0  NIT= 43 NFV=147 NFG=  0 NDC=  0 NCG=  0 F= .233D-15 G= .164D-07
FF=    .2333078060D-15
X=     .9999999847D+00    .9999999694D+00
TIME= 0:00:00.66
```

V bodě `X` bylo nalezeno lokální minimum s funkční hodnotou `FF`. Volbu metody, jakou byl náš problém vyřešen, provedl systém automaticky, protože jsme žádnou ve vstupním souboru nespecifikovali.

Velké množství variant implementovaných metod je možné díky struktuře systému UFO založené na modulech. Tyto moduly se obvykle skládají ze dvou částí – ze šablony napojení (interface template) a vlastního kódu ve Fortranu 77. Šablonu používá pouze UFO preprocesor, který pomocí ní generuje výsledný program[1] řešící zadaný optimalizační problém. Šablona zajistí správné volání vlastního kódu (druhá součást modulu), kterým modul provede to, co má. Mimo tyto šablony, které jsou součástí modulů, existují i speciální šablony, které řídí běh UFO preprocesoru. Jednou z nich je například soubor `UZDCLP.I` nebo vstupní soubor s popisem řešeného optimalizačního problému. V šablonách jsou také uloženy informace pro automatický výběr optimalizační metody.

Nyní blíže popíšeme vstupní soubor, řídící jazyk systému UFO (ve kterém je vstupní soubor napsán), a zpracování vstupního souboru UFO preprocesorem. Řídící jazyk systému UFO obsahuje čtyři typy instrukcí: příkazy Fortranu 77, příkazy Fortranu 77 obsahující makroproměnné, instrukce řídící UFO preprocesor (mohou obsahovat makroproměnné) a speciální makroinstrukce (substituce).

Makroproměnné jsou v systému UFO velmi podstatné. Jsou typu textový řetězec a jejich jméno vždy začíná znakem `$`. Hlavní význam makroproměnných je v substituci jejich hodnot v příkazech Fortranu nebo v řídících příkazech UFO CL. Přiřazení hodnoty do makroproměnné provedeme příkazem `$JMÉNOMAKROPROMĚNNÉ='hodnota'`. Pokud hodnota je číslo, není nutné používat uvozovky. Například přiřadíme-li `$HESF='D'`, `$TYPE='L'`, `$DECOMP='G'` a `$NUMBER=1`, pak se z výrazu `CALL UD$HESF$TYPE$DECOMP$NUMBER` po zpracování UFO preprocesorem díky substituci stane `CALL UDDLG1`. Jiný způsob, jak nastavit hodnotu makroproměnné, je

```
  $SET(JMÉNOMAKROPROMĚNNÉ)
```

[1] Přesněji řečeno, část programu, která odpovídá volanému modulu.

```
  hodnota
$ENDSET
```

Hodnota `hodnota` nemusí být právě jeden řádek, může jich obsahovat libovolně mnoho. Tento způsob se hodí zejména při vkládání několika příkazů Fortranu. Hodnoty makroproměnných lze měnit pomocí instrukcí jazyka UFO CL.

Obecně řečeno, pomocí instrukcí jazyka UFO CL je možné například definovat a měnit hodnoty makroproměnných, vkládat další soubory (se současným provedením nebo neprovedením jejich zpracování UFO preprocesorem), větvit průběh zpracování a provádět cykly (i zde se například jako řídící proměnná cyklu používá makroproměnná). Dalšími instrukcemi jsou speciální substituce, které ovlivňují běh UFO preprocesoru. Jedná se například o instrukce `$BATCH`, `$METHOD`, `$STANDARD` atd. Například uvedení `$METHOD` způsobí vygenerování optimalizační metody (vlastně složení z jednotlivých modulů) podle předem specifikovaných požadavků. Instrukce `$STANDARD` provede (zhruba řečeno) deklaraci a inicializaci proměnných, sestavení metody a výpis jejích výsledků. Pro úplnost dodejme, že každá instrukce začíná znakem `$`.

UFO preprocesor je založen na interpretru BEL (Batch Editor Language), který byl vyvinut jako součást systému UFO. BEL je určen hlavně pro dávkové zpracování textů (programů, tiskových výstupů atd.). Jeho hlavní příkazy jsou buď řídící instrukce (podmínka, cyklus, operace s makroproměnnými atd.) nebo substituce hodnot makroproměnných. Vidíme tedy, že některé řídící příkazy UFO CL jsou vlastně řídícími příkazy BELu. Jeho podrobnější popis je uveden v [5].

Vrátíme-li se nyní k příkladu na straně 85, je již téměř celý jasný. Makroproměnnou `$INPUT` specifikujeme počáteční hodnoty proměnné x, makroproměnnou `$FMODELF` postup pro výpočet optimalizované funkce. Kdybychom specifikovali hodnoty makroproměnných `$GMODELF`, a `$HMODELF` jako postup výpočtu gradientu a Hessovy matice optimalizované funkce, použil by se pro jejich výpočet tento předpis. Tyto makroproměnné jsme ale nezadali, takže se gradient i Hessova matice spočte numericky (pomocí poměrných diferencí). Makroproměnná `$NF` určuje počet nezávislých proměnných a `$NOUT` podrobnosti výstupu spočtených hodnot. Instrukce `$BATCH` způsobí, že se nebude volat dialogový mód, a `$STANDARD` vytvoří program ve Fortranu, který bude řešit náš optimalizační problém.

V tomto vstupním souboru jsme neurčili, jaká metoda se má použít. Byla systémem vybrána automaticky podle údajů o metodách, které jsou uloženy v šablonách. Metodu, kterou chceme použít, specifikujeme volbou hodnot makroproměnných `$CLASS`, `$TYPE`, `$DECOMP`, `$NUMBER` a `$UPDATE`.

Všechny funkce, které souvisí s řešeným optimalizačním problémem[2], se zadávají pomocí makroproměnných, jejichž jméno se skládá ze tří částí. První část je některé z písmen `F`, `G`, `D`, `H` nebo jejich kombinace (`FG`, `FD`, `GD`, `FGD` nebo `FGH`). Písmeno `F` znamená hodnotu funkce, `G` gradient vzhledem k základním proměnných, `D` gradient vzhledem k stavovým proměnným a `H` Hessovu matici vzhledem

[2] Jedná se například o minimalizovanou (maximalizovanou) funkci, funkce určující vazbové podmínky atd.

k základním proměnným. Kombinace těchto písmen znamená, že jsou zadány příslušné funkce zároveň. Druhá část jména makroproměnné je vždy `MODEL`. Třetí část je jedno z písmen `F`, `A`, `C`, `E`, `Y`, navíc každé z nich (kromě `F`) může být následováno písmenem `S`. Podle tohoto jednoho (dvou) písmen specifikujeme, jakou funkci (případně její derivace) zadáváme. `F` znamená minimalizovanou (modelovou) funkci, `C` znamená funkci z vazbových podmínek atd. Pokud na konci jména makroproměnné není písmeno `S`, zadáváme příslušnou funkci pro konkrétní index, jinak zadáváme funkce pro všechny indexy najednou. Libovolné kombinace první, druhé a třetí části jména makroproměnné nejsou možné. Jména makroproměnných, které lze z těchto tří části složit, včetně dalších podrobností jsou uvedeny v [5]. Jménem makroproměnné se řídí také označení závislých a nezávislých proměnných funkce, kterou makroproměnná definuje.

7.2 Návrh implementace

Jak již víme, systém UFO používá ke svým výpočtům také hodnoty derivací funkcí. Ty je možné zadat analyticky pomocí makroproměnných, které jsme popsali na konci předchozího odstavce, jinak jsou jejich aproximace počítány pomocí poměrných diferencí (viz makroproměnná `$MCG`). V této kapitole popíšeme návrh implementace automatického derivování v systému UFO, pomocí kterého se budou požadované derivace vyhodnocovat přesně a rychle. Automatickou transformaci výpočtu hodnot funkce na výpočet hodnoty funkce a její derivace bude provádět BEL preprocesor. V následujících odstavcích popíšeme detaily této implementace.

Zavedeme nové makroproměnné `$AMODELF`, `$AMODELA`, `$AMODELC`, `$AMODELE` a `$AMODELY`, které budou obsahovat postup výpočtu hodnot funkce, který chceme automaticky derivovat.[3] Tento výpočet je zároveň postupem, jak počítat hodnoty funkcí specifikovaných posledním písmenem ve jméně této makroproměnné. Proto UFO preprocesor dosadí hodnotu těchto nových makroproměnných do makroproměnných, kde jsme počáteční písmeno `A` vyměnili za `F`, tedy `$FMODELF`, `$FMODELA`, `$FMODELC`, `$FMODELE` a `$FMODELY`. Požadujeme, aby zadaný postup byl napsán ve Fortranu 77, a to nikoli v jeho volném formátu. Makroproměnné mohou být v postupu výpočtu použity, ve výsledném programu budou nahrazeny svou hodnotou.

Podle posledního písmena ve jménu nové makroproměnné víme, jakou funkci tato makroproměnná definuje. Proto také víme, jaké jsou její nezávislé proměnné a jak je označena závislá proměnná (viz [5]). Výsledkem výpočtu v nově zavedených makroproměnných je skalární hodnota, spočtená pro zadané parametry a nezávislé proměnné. Podle zvolené optimalizační metody, případně podle makroproměnné `$KDS`, UFO preprocesor sám určí, kolikáté derivace až potřebuje.[4] Protože v makroproměnných `$GMODELF`, `$HMODELF` atd. se počítají celé gradienty,

[3] Pokud těmto makroproměnným nepřiřadíme hodnotu, nebudou se derivace generovat automatickým derivováním. Je samozřejmě možné využít automatické derivování jen pro některou nebo některé z uvedených pěti skupin funkcí.

[4] Nejvyšší derivace, používané v systému UFO, jsou druhé derivace.

respektive celé Hessovy matice, budeme tak automatickým derivováním činit i my, pokud budou potřeba.

Zamysleme se nyní, jaký mód automatického derivování je výhodnější v naší situaci použít. Počítáme gradienty pouze skalárních funkcí, proto budeme pro výpočet první derivace používat mód zpětný. Druhé derivace budeme počítat kombinací přímého a zpětného módu, kterou jsme popsali v kapitole 4.

Vlastní transformaci původního postupu výpočtu na nový (který bude navíc počítat i derivace) provede BEL preprocesor při svém druhém průchodu zdrojovým souborem (který je napsán v řídícím jazyce systému UFO). Volíme raději druhý průchod, aby všechny makroproměnné použité v derivovaném kódu již byly známé. Konkrétní postup transformace může být shodný s postupem, který jsme navrhli a podrobně popsali v kapitole 5.2.2 na stranách 63 až 66. Délku polí, které se ve zmíněné implementaci používají, můžeme předem specifikovat v makroproměnné `$DIMCL`. Ve stručnosti zopakujme, že tato transformace předeklaruje nezávislé a závislé proměnné a proměnné, které závisí na nezávislých a ovlivňují závislé. Dále nahradí volání základních funkcí jinými funkcemi, které se budou navíc zaznamenávat do dlouhých polí. Nakonec je nutné projít těmito poli odzadu, aby se nastřádaly všechny hodnoty do přidružených proměnných a tím se spočetla hodnota hledaného gradientu.

Je-li potřeba postup výpočtu ztransformovat pro výpočet Hessovy matice, BEL preprocesor použije postup navržený v kapitole 5.2.3, který zároveň spočte i gradient.

Tímto způsobem jsme odvodili postup pro výpočet prvních, resp. druhých derivací zadaných funkcí. Přistupujme k němu podobně, jako kdyby byl zadaný analyticky pomocí makroproměnných `$GMODELF`, `$HMODELF` atd.

Systém UFO dokáže podstatným způsobem využívat řídkosti Hessových matic (viz například makroproměnná `$HESF`). Uživatel přitom zadá pomocí makroproměnných umístění jejich nenulových prvků. Této struktury můžeme využít při automatickém derivování, jak jsme popsali v kapitole 6.3.2. BEL preprocesor ze zadané struktury vygeneruje matice S a W a s nimi provede výpočet zkomprimované Hessovy matice. Nakonec ještě zajistí její dekompresi. V případě velkých úloh můžeme tímto způsobem dosáhnout značného zrychlení výpočtu derivací.

Několik slov závěrem

V této práci jsme popsali transformaci programu pro výpočet hodnot funkce $\boldsymbol{F}$ v bodě $\boldsymbol{x}$ na program, který bude přesně a velmi rychle počítat také hodnoty jejich derivací. Mimo jiné jsme se zmínili o využití řídkosti Jacobiho a Hessovy matice při jejich výpočtu. Dále jsme podrobně popsali implementaci automatického derivování v systému UFO.

V literatuře je možné se dočíst o dalších vylepšeních automatického derivování. Při jejich implementaci však rostou režijní náklady a je otázkou, zda se je vyplatí pro běžné úlohy používat.

Doufejme, že se postupem času technika automatického derivování rozšíří a zařadí po bok klasických metod pro výpočet hodnot derivací (resp. jejich aproximací), totiž symbolického derivování a poměrných diferencí. Má k tomu totiž veškeré potřebné předpoklady.

Literatura

[1] Kim K. V., Nesterov I. E., Skokov V. A., Cherkasskii B. V. (1984): An Efficient Algorithm for Computing Derivatives and Extremal Problems. Anglický překlad z „Effektivnyi algoritm vychisleniia proizvodnykh i ekstremal'nye zaduchi", *Ekonomika i matematicheskie metody*, **2**, 309–318.

[2] Jarník V. (1984): Diferenciální počet II. Academia, Praha.

[3] Virius M. (1999): Programování v C++. ČVUT, Praha.

[4] Griewank A. (2000): Evaluation Derivatives: Principles and Techniques of Algorithmic Differentiation. SIAM, Philadelphia.

[5] Lukšan L., Tůma M., Šiška M., Vlček J., Ramešová N. (2000): UFO 2000, Interactive System for Universal Functional Optimization, Technical Report 826. Institute of Computer Science, Academy of Sciences of the Czech Republic. Viz též `http://www.cs.cas.cz/ics/reports/v826-00.ps`

[6] Automatic Differentiation of Algorithms: Theory, Implementation, and Application. Sborník. Ed. Griewank A., Corliss G. F. SIAM, Philadelphia, 1992.

[7] Verma A. (1998): Structured Automatic Differentiation. Disertační práce, Cornell University.

[8] Golub G. H., Van Loan C. F.(1996): Matrix computations. Johns Hopkins University Press, Baltimore.

[9] Computational differentiation – techniques, applications, and tools. Sborník. Ed. Berz M., Bischof C. H., Corliss G. F., Griewank A. SIAM, Philadelphia, 1996.

[10] Griewank, A. (1988): On Automatic Differentiation. Preprint ANL/MCS-P10-1088, Argonne National Laboratory, Illinois, USA.