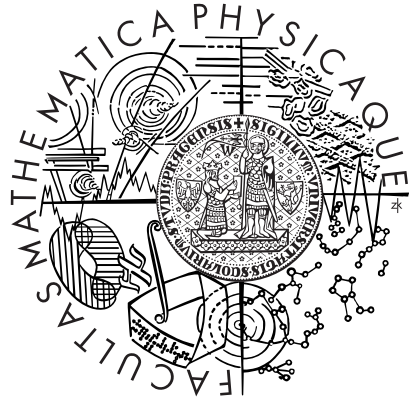
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta



Disertační práce

Jan Hartman

Automatické derivování

Katedra numerické matematiky

Školitel: Doc. RNDr. Jan Zítko, CSc.
Konzultant: Prof. Ing. Ladislav Lukšan, DrSc.

This work is a part of the research project MSM 0021620839 financed by MSMT.

Děkuji vedoucímu své disertační práce doc. RNDr. Janu Zítkovi, CSc.
a konzultantovi prof. Ing. Ladislavu Lukšanovi, DrSc. za věnovaný čas,
poskytnutou literaturu, nesčetné nápady a cenné připomínky, které vedly
k jejímu zlepšení.

Obsah

Předmluva

Při řešení nejrůznějších úloh jsme postaveni před úkol spočítat derivaci zadané funkce nebo zobrazení (případně celý gradient, Jacobiho matici, Hessovu matici, směrové derivace atd.). Taková situace nastává například při řešení optimalizačních úloh, v meteorologii, fyzice, chemii atd., kdy se mohou počítat derivace různých funkcí v mnoha různých bodech. Jedním z možných způsobů efektivního výpočtu hodnot derivací *pomocí počítače* se budeme zabývat v této práci.

Předpokládejme, že máme program, který počítá hodnotu reálné 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 uvažovaném bodě.

Jednou z možností, jak toho dosáhnout, je počítat tzv. *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 typu $hf''(x)/2$ apod.). Navíc, pro výpočet aproximace celého gradientu funkce n proměnných je potřeba zhruba $(n+1)$-krát tolik času, jako pro výpočet původní funkce. Při výpočtu vyšších derivací pomocí poměrných diferencí je přesnost spočtených hodnot ještě menší a výpočet ještě náročnější.

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

Třetí možností výpočtu derivace pomocí počítače je *automatické derivování* (algoritmické derivování, automatic differentiation, computational differentiation, algorithmic differentiation). Aplikací automatického derivování na program, který počítá hodnoty funkce, získáme *program*, který vyhodnotí *navíc* i požadovanou derivaci této funkce. Hlavní výhody automatického derivování jsou:

- transformaci programu lze provést automaticky,
- dostáváme přesné hodnoty derivace (chyba metody je nula),
- vyhodnocení celého gradientu skalární funkce n proměnných zabere při použití zpětného módu automatického derivování *nejvýše čtyřnásobek* času potřebného pro výpočet hodnoty derivované funkce.

Automatickým derivováním, jeho vlastnostmi a aplikacemi se budeme zabývat v této práci.

Z praktického hlediska je automatické derivování užitečná metoda, která nachází uplatnění ve všech oblastech, kde je potřeba počítat derivace pomocí počítače – v matematice, různých oblastech fyziky, průmyslu atd. Existuje také mnoho implementací automatického derivování, ze známých jmenujme alespoň ADIFOR nebo ADOL-C, viz [2] nebo [26].

Zároveň probíhá další výzkum teoretických vlastností a hledání vylepšení používaných metod, například v oblasti využití struktury derivovaných funkcí nebo řídkosti matic derivací. Dále jsou zkoumána možná vylepšení implementací automatického derivování, například v oblasti optimalizace počtu numerických operací a efektivního využití paměti počítače.

Vývoj a použití automatického derivování přímo souvisí s vlastnostmi a schopnostmi soudobých programovacích jazyků a jejich kompilátorů. Proto v posledních letech zažívá automatické derivování rozmach zájmu. První zmínky o automatickém derivování lze vysledovat v 60. letech, kdy byl přímý mód popsán Wengertem (1964). Další zlepšení vypracovali Kedem (1980), Rall [23] nebo Kagiwada [15]. Základy zpětného módu byly nezávisle na sobě představeny několika matematiky, například Linnainmaaem (1969), Spielpenningem (1980) a Kimem (1984), viz například [16] nebo [7].

Literatury, která se zabývá automatickým derivováním, je velké množství, zejména z poslední doby. Množství odkazů lze nalézt například v [5]. Pravidelně jsou konány konference o automatickém derivování, viz například [1], [3], [6]. Zajímavý rozcestník s množstvím odkazů a literatury je na internetové adrese `http://www.autodiff.org`.

Obsah a cíle

V první kapitole nejprve popíšeme formální tvar programu, na který budeme aplikovat automatické derivování, a uvedeme základní předpoklady. Odvodíme program pro výpočet první, druhé a vyšších derivací a jejich základní vlastnosti. Dále popíšeme základní možnosti implementace programů odvozených automatickým derivováním a odhad složitosti odvozeného výpočtu derivace.

Jedním z hlavních cílů této disertační práce byla implementace technik automatického derivování do systému UFO. Proto ve druhé kapitole popíšeme systém UFO – jeho principy a použití.

Ve třetí kapitole uvedeme, jakým způsobem je automatické derivování v systému UFO možné vyvolat a použít. Podrobně ukážeme, jak jsme automatické derivování do tohoto systému implementovali. Uvedené principy neplatí pouze v systému UFO, lze je totiž použít obecně i v jiných systémech nebo aplikacích.

Jedním z předpokladů automatického derivování je, aby všechny elementární funkce byly spojité až do toho řádu, do jakého požadujeme spočíst derivace. Ve čtvrté kapitole rozšíříme množinu elementárních funkcí o některé nehladké funkce, čímž bude možno pomocí automatického derivování spočítat nějaký subgradient zadané funkce (gradient nemusí existovat). Díky tomu se (nejen v systému UFO) mohou optimalizovat nehladké funkce pomocí metod nehladké optimalizace s efektivním výpočtem derivace, respektive subgradientu, pomocí metod automatického derivování.

Uvedené metody budeme ve všech kapitolách demonstrovat na několika příkladech.

Kapitola 1

Automatické derivování

V první kapitole se budeme zabývat hlavními principy automatického derivování a jeho vlastnostmi. Formálně popíšeme tvar programu, na který budeme aplikovat automatické derivování, a uvedeme základní předpoklady. Odvodíme postupy pro výpočet první, druhé a vyšších derivací a jejich základní vlastnosti. Dále se zmíníme o implemetaci programů odvozených automatickým derivováním a o odhadu složitosti výpočtu. Uvedeme několik (záměrně jednoduchých) příkladů, které budou principy automatického derivování jasně demonstrovat. Nejprve však popíšeme základní vlastnosti automatického derivování.

1.1 Vlastnosti automatického derivování

Jak jsme již uvedli v předmluvě, aplikací automatického derivování na program, který počítá hodnoty funkce, získáme *program*, který vyhodnotí *navíc* i derivaci této funkce. Tuto *transformaci programu* lze provést *automaticky*, jak uvidíme v kapitole 1.6. Celý postup je založen na pravidle řetězení (Věta 1.2), které se aplikuje na *numerické hodnoty* získané vyhodnocením elementárních funkcí – na rozdíl 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. Pokud by tedy počítačová aritmetika byla přesná, dostali bychom odvozeným programem hodnotu derivace přesně. Asi nejzajímavější vlastnost automatického derivování je to, že vyhodnocení celého gradientu skalární funkce n proměnných zabere při použití zpětného módu automatického derivování *nejvýše čtyřnásobek* času potřebného pro vyhodnocení derivované funkce. Pro výpočet druhých a vyšších derivací platí obdobné vlastnosti.

Další vlastnosti automatického derivování vyplynou z následujících kapitol.

1.2 Základní předpoklady

Abychom mohli popsat, jak automatické derivování funguje, uvedeme nejprve formální popis obecné struktury programu, na který budeme automatické derivování aplikovat. Dále položíme základní předpoklady, které musí být pro aplikaci automatického derivování splněny.

1.2.1 Soupis operací

Předpokládejme, že program, který chceme transformovat, počítá hodnotu funkce

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

Hodnoty funkce f se počítají programem, funkce f je tedy zadána *explicitně*. Neboli f je složením *základních funkcí* (elementárních funkcí, elementárních operací) – například sčítání, násobení, sinus apod. Abychom mohli automatické derivování použít, musíme být schopni určit *parciální derivace* základních funkcí.

Výše uvedené neznamená, že by výpočet hodnoty $f(x)$ musel být popsán pouze jedinou algebraickou formulí nebo že by program pro výpočet hodnoty $f(x)$ 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. Během programu se postupně počítají výsledky jednotlivých základních operací a ty jsou používány jako argumenty dalších základních operací. Každý takový výsledek základní funkce označme v_i.

Pokud takto rozepíšeme celý algoritmus vyhodnocení $f(x)$ (pro konkrétní parametry a vstupní data) a připojíme-li přiřazení vstupních proměnných $x = (x_1, \dots x_n)$ do v_i, $i = 1-n, \dots, 0$ a přiřazení výstupních proměnných do $y = (y_1, \dots y_m)$, dostáváme *soupis operací* (code-list). Jinak řečeno, soupis operací je rozpis všech operací po *rozvinutí* cyklů a podprogramů a po nalezení správných cest při větvení programu.

Všechny číselné hodnoty v_i spočtené během vyhodnocení funkce f v jednom konkrétním bodě označme následujícím způsobem:

$$\underbrace{v_{1-n}, \dots, v_0}_{x}, v_1, v_2, \dots, v_{l-m}, \underbrace{v_{l-m+1}, \dots, v_l}_{y}.$$

Dále zavedeme relaci $\prec$. Řekneme, že $j \prec i$ právě tehdy, když hodnota proměnné v_i závisí *přímo* na proměnné v_j.

Každou hodnotu v_i, $i = 1, \dots, l$ získáme vyhodnocením *základní funkce* φ_i (elementární funkce, elemental function, library function), neboli

$$v_i = \varphi_i(v_j)_{j \prec i}, \tag{1.1}$$

kde $(v_j)_{j \prec i}$ jsou všechny proměnné, které přímo ovlivňují proměnnou v_i, tj. ty proměnné, ze kterých se počítá hodnota v_i. Navíc budeme předpokládat, že indexy

proměnných v_i splňují vztah

$$j \prec i \quad \Longrightarrow \quad j < i,$$

neboli hodnota v_i se počítá z hodnot v_j s indexy $j < i$. Dále budeme požadovat, aby pokud je definovaná hodnota v_i, pak aby byly definované i všechny předchozí hodnoty $v_{1-n}, \ldots, v_{i-1}$, neboli "posloupnost proměnných v_i je v každém okamžiku souvislá". Tyto požadavky lze snadno splnit vhodným očíslováním proměnných v_i.

Program, který chceme transformovat, lze tedy formálně přepsat do soupisu operací na obrázku 1.1. V první části jsou proměnným $v_{1-n}, \ldots, v_0$ přiřazeny vstupní hodnoty $x_1, \ldots, x_n$, ve druhé části probíhá výpočet hodnoty $f(x)$ a ve třetí části jsou naplněny výstupní hodnoty $(y_1, \ldots, y_m) = f(x)$.

v_{i-n}	$=$	x_i	pro	$i = 1, \ldots, n$
v_i	$=$	$\varphi_i(v_j)_{j \prec i}$	pro	$i = 1, \ldots, l$
y_{m-i}	$=$	v_{l-i}	pro	$i = m-1, \ldots, 0$

Obrázek 1.1: Soupis operací.

Proces výpočtu hodnoty $f(x)$ je možné popsat pomocí orientovaného *grafu výpočtu*. Jeho vrcholy odpovídají proměnným x_i, v_j a y_k. Orientovaná hrana vede z vrcholu v_i do vrcholu v_j, právě když proměnná proměnná v_j přímo závisí na proměnné v_i, neboli $i \prec v$. Graf výpočtu lze využít například při hledání minimálního počtu prováděných numerických operací při výpočtu derivace, tj. při využití struktury derivované funkce. Viz například [24].

Příklad I. Popisované principy budeme v celé práci demonstrovat na několika ilustrativních příkladech. Předpokládejme nyní, že máme program pro výpočet hodnot funkce $f \subseteq \mathbb{R}^2 : M \to \mathbb{R}$ definované předpisem

$$f(x_1, x_2) = \frac{\sin x_1}{x_1 x_2} + x_1 x_2,$$

kde $M = \{[x_1, x_2] \in \mathbb{R}^2; x_1 \neq 0 \text{ a } x_2 \neq 0\}$. Chceme spočítat hodnotu $y = f(\pi/4, 1)$ a také její gradient v tomtéž bodě.

Podle formálního postupu uvedeného výše předpokládejme, že výpočet hodnoty $f(x)$ probíhá podle obrázku 1.2.

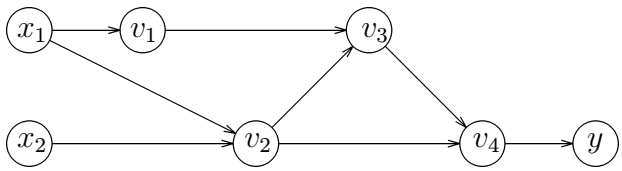
Graf výpočtu odpovídající tomuto soupisu operací je uveden na obrázku 1.3. □

Základní funkce φ_i jsou zamýšleny hlavně jako "elementární" funkce (např. sčítání, násobení, trigonometrické funkce atd.), ale v principu to mohou být i funkce složitější, například výpočet inverní matice atp.

Proměnné $v_i, i = 1, \ldots, l - m$ je možné chápat také jako vektorové, ne nutně jako skalární. Na přímém ani zpětném módu automatického derivování tato úprava nic nezmění, všechny postupy a vlastnosti lze odvodit úplně analogicky jako ve skalárním případě.

v_{-1}	=	x_1	=	$\frac{\pi}{4} = 0.7854$
v_0	=	x_2	=	1.0000
v_1	=	$\sin v_{-1}$	=	0.7071
v_2	=	$v_{-1} * v_0$	=	0.7854
v_3	=	v_1/v_2	=	0.9003
v_4	=	$v_2 + v_3$	=	1.6857
y	=	v_4	=	1.6857

Obrázek 1.2: Soupis operací pro Příklad I.



Obrázek 1.3: Graf výpočtu pro Příklad I.

1.2.2 Totální diferenciál složené funkce

Pro úplnost výkladu připomeneme dvě základní tvrzení o funkcích více proměnných, která jsme převzali z [14] a která uvedeme bez důkazu.

Věta 1.1
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 1.2 (pravidlo řetězení, chain rule)
Nechť funkce $y_1(x_1, \ldots, x_k)$, …, $y_n(x_1, \ldots, x_k)$ mají totální diferenciály $\mathrm{d}y_1(a)$, …, $\mathrm{d}y_n(a)$ 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

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

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

$$\frac{\partial z^*}{\partial x_i}(a) = \frac{\partial z}{\partial y_1}(b) \cdot \frac{\partial y_1}{\partial x_i}(a) + \cdots + \frac{\partial z}{\partial y_n}(b) \cdot \frac{\partial y_n}{\partial x_i}(a). \tag{1.2}$$

□

Předpoklad existence totálního diferenciálu na "vnější" funkci z nelze zeslabit, viz například [14].

Věta 1.2 je základní větou, na které je automatické derivování založeno. V následujícím odstavci položíme požadavky na základní funkce, aby byly splněny teoretické předpoklady pro automatické derivování a aby ho tedy bylo možné použít.

1.2.3 Základní funkce a jejich vlastnosti

S ohledem na Věty 1.1 a 1.2, budeme v dalším textu předpokládat splnění následujícího předpokladu.

Předpoklad 1.1 (Diferencovatelnost základních funkcí)
Nechť d je přirozené číslo. Všechny elementární funkce φ_i, $i = 1, \ldots, l$ jsou d-krát spojitě diferencovatelné na svém definičním oboru $D_i \subset \mathbb{R}^n$, kde D_i je otevřená v $\mathbb{R}^n$. □

Pro praktické účely jsou dostačující případy $d = 1$ (pro výpočet první derivace) nebo $d = 2$ (pro výpočet druhé derivace). Jasným důsledkem předpokladu 1.1 je následující věta.

Věta 1.3
Nechť platí předpoklad 1.1. Pak

$$D = \{x \in \mathbb{R}^n; \text{hodnota } f(x) \text{ je definovaná soupisem operací na obrázku 1.1}\}$$

je otevřená množina v $\mathbb{R}^n$ a navíc $f \in C^d(D)$. □

Další ze známých vět o funkcích více proměnných tvrdí následující.

Věta 1.4
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. □

Předpoklad spojité diferencovatelnosti elementárních funkcí až do řádu d na otevřené množině D_i v $\mathbb{R}^n$ lze zeslabit na požadavek spojitosti derivací základních funkcí φ_i až do řádu $n-1$ a současně existence totálního diferenciálu pro $(n-1)$-ní derivaci základních funkcí φ_i, oboje na otevřené množině D_i v $\mathbb{R}^n$ (stejné jako za původních předpokladů). Pravidlo řetězení (věta 1.2) platí totiž i za těchto slabších předpokladů a všechna odvození automatického derivování lze provést naprosto analogicky. Dostaneme tak stejné výsledky, jaké jsme získali za původních předpokladů.

V kapitole 4 se budeme zabývat výpočtem subgradientu pomocí automatického derivování. Výše uvedené předpoklady o základních funkcích tam budou ještě zeslabeny.

1.3 Výpočet první derivace - přímý mód

Přímý mód automatického derivování (forward mode, tangent) nám umožní počítat derivace všech výstupních proměnných $y_1, \dots y_m$ najednou podle zadaného směru.

Pro úplnost si zopakujme jedno známé tvrzení z [14].

Věta 1.5

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 $v = (v_1, \dots, v_n)^T$, tj. $\partial_v f(a)$, platí

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

1.3.1 Vlastnosti a odvození přímého módu

Soupis operací na obrázku 1.1 zderivujeme s pomocí pravidla řetězení (Věta 1.2). Přesněji řečeno, ke každé základní operaci

$$v_i = \varphi_i(v_k)_{k \prec i}$$

přidáme novou *sdruženou* operaci

$$\dot{v}_i = \sum_{j \prec i} \frac{\partial \varphi_i}{\partial v_j}(v_k)_{k \prec i} \cdot \dot{v}_j, \tag{1.3}$$

která je derivací původní operace. Tím jsme ke každé proměnné v_i přiřadili novou *sdruženou* proměnnou $\dot{v}_i$, která je její derivací[1]. Úvodní a závěrečnou část soupisu operací doplníme analogicky. Takto získáme *sdružený* soupis operací, který je na obrázku 1.4.

$$\left.\begin{array}{lcl} v_{i-n} & = & x_i \\ \dot{v}_{i-n} & = & \dot{x}_i \end{array}\right\} \; i = 1, \dots n$$

$$\left.\begin{array}{lcl} v_i & = & \varphi_i(v_k)_{k \prec i} \\ \dot{v}_i & = & \sum_{j \prec i} \frac{\partial \varphi_i}{\partial v_j}(v_k)_{k \prec i} \cdot \dot{v}_j \end{array}\right\} \; i = 1, \dots l$$

$$\left.\begin{array}{lcl} y_{m-i} & = & v_{l-i} \\ \dot{y}_{m-i} & = & \dot{v}_{l-i} \end{array}\right\} \; i = m-1, \dots 0$$

Obrázek 1.4: Soupis operací odvozený přímým módem automatického derivování.

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

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

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

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

Je možné také spočíst derivaci podle p směrů najednou, pokud proměnnou $\dot{x}$ budeme uvažovat jako matici, v jejíž sloupcích jsou uloženy jednotlivé směry. V tom případě chápeme proměnné $\dot{x}_i$ jako vektory.

Demonstrace na příkladu I. Navážeme na výše uvedený příklad I ze strany 11. Chceme spočíst gradient funkce f v bodě $(\pi/4, 1)$, tj. $\nabla f(\pi/4, 1)$.

Nejdříve spočítáme derivaci podle x_1. Na každý řádek soupisu operací z minulého příkladu aplikujeme pravidlo řetězení. Tím dostáváme soupis operací, který je uveden na obrázku 1.5.

x_1	$=$	$\frac{\pi}{4}$	$=$	0.7854
$\dot{x}_1$	$=$	1	$=$	1.0000
x_2	$=$	1	$=$	1.0000
$\dot{x}_2$	$=$	0	$=$	0.0000
v_1	$=$	$\sin x_1$	$=$	0.7071
$\dot{v}_1$	$=$	$\cos x_1 * \dot{x}_1$	$=$	0.7071
v_2	$=$	$x_1 * x_2$	$=$	0.7854
$\dot{v}_2$	$=$	$x_1 * \dot{x}_2 + x_2 * \dot{x}_1$	$=$	1.0000
v_3	$=$	v_1 / v_2	$=$	0.9003
$\dot{v}_3$	$=$	$(\dot{v}_1 - v_3 * \dot{v}_2)/v_2$	$=$	-0.2460
v_4	$=$	$v_2 + v_3$	$=$	1.6857
$\dot{v}_4$	$=$	$\dot{v}_2 + \dot{v}_3$	$=$	0.7540
y	$=$	v_4	$=$	1.6857
$\dot{y}$	$=$	$\dot{v}_4$	$=$	0.7540

Obrázek 1.5: Soupis operací odvozený přímým módem automatického derivování pro Příklad I.

Všimněme si, že kromě hodnoty derivace $\partial f / \partial x_1(\pi/4, 1)$ jsme také spočetli hodnotu $f(\pi/4, 1)$. Pokud bychom chtěli vypočítat ještě derivaci podle druhé proměnné, totiž $\partial f / \partial x_2(\pi/4, 1)$, provedeme tento soupis operací znovu, ale s počátečními hodnotami $(\dot{x}_1, \dot{x}_2) = (0, 1)$. □

1.4 Výpočet první derivace - zpětný mód

Až do této chvíle jsme se zabývali přímým módem automatického derivování. V tomto odstavci odvodíme zpětný mód automatického derivování, který je v mnoha ohledech zajímavější a poskytuje větší praktické využití.

Pokud je f skalární funkce, tj. $m = 1$, pomocí *zpětného* módu automatického derivování (reverse mode, adjoint, gradients) spočítáme její gradient. Pokud je f vektorová funkce, tj. $m > 1$, spočítáme lineární kombinaci gradientů jednotlivých složek f_i, respektive gradient i-té komponenty funkce f při vhodné volbě parametrů.

1.4.1 Vlastnosti a odvození zpětného módu

Odvození zpětného módu automatického derivování je možné několika způsoby. Zde provedeme odvození přímo z pravidla řetězení a potom z přímého módu. Oba získané postupy budou až na malé detaily stejné.

Odvození zpětného módu z pravidla řetězení

Pro jednoduchost výkladu budeme nejprve předpokládat, že $m = 1$, neboli výstupní hodnota $v_l = y \in \mathbb{R}$ je skalár. Numerické hodnoty získané během výpočtu splňují tedy označení

$$\underbrace{v_{1-n}, \ldots, v_0}_{=x}, v_1, v_2, \ldots, v_{l-1}, \underbrace{v_l}_{=y}.$$

Obecnějším případem, kdy $y \in \mathbb{R}^m, m > 1$, se budeme zabývat později.

Zaveďme nové proměnné

$$\bar{v}_i = \frac{\partial v_l}{\partial v_i}, \qquad i = 1 - n, \ldots, l.$$

Věta 1.6
Platí následující rovnosti

(a) $\bar{v}_l = 1$,

(b) pro $j = 1 - n, \ldots, l - 1$ platí

$$\bar{v}_j = \sum_{i \ ; \ i \succ j} \bar{v}_i \cdot \frac{\partial \varphi_i}{\partial v_j}. \tag{1.4}$$

Důkaz.

(a) Zřejmé.

(b) Tuto rovnost ukážeme ve speciálním případě, kdy právě tři proměnné v_{i_1}, v_{i_2} a v_{i_3} přímo závisí na proměnné v_j, tj. $j \prec i_1$, $j \prec i_2$ a $j \prec i_3$. V jiném případě je důkaz naprosto analogický.

Hodnotu $v_l = y$ je možné vyjádřit jako funkci proměnné v_j a dalších proměnných $v_{p_1}, \ldots, v_{p_r}$, které na proměnné v_j vůbec nezávisí. Neboli

$$v_l = v_l(v_j, v_{p_1}, \ldots, v_{p_r}).$$

Protože však proměnná v_j ovlivňuje proměnné v_{i_1}, v_{i_2} a v_{i_3}, je možné proměnnou $v_l = y$ vyjádřit jako funkci proměnných v_{i_1}, v_{i_2} a v_{i_3} a dále $v_{p_1}, \ldots, v_{p_r}$, tj.

$$v_l = v_l(v_{i_1}, v_{i_2}, v_{i_3}, v_{p_1}, \ldots, v_{p_r}).$$

Tuto funkci bychom zde měli označit spíše jako v_l^*, ale pro jednoduchost tak nebudeme činit. Protože v_{i_1}, v_{i_2} a v_{i_3} jsou funkcí v_j (a možná i některých dalších proměnných), lze psát

$$\begin{aligned} v_{i_1} &= v_{i_1}(v_j, \ldots), \\ v_{i_2} &= v_{i_2}(v_j, \ldots), \\ v_{i_3} &= v_{i_3}(v_j, \ldots). \end{aligned}$$

Použitím výše uvedených vztahů můžeme psát

$$v_l = v_l(v_{i_1}(v_j, \ldots), v_{i_2}(v_j, \ldots), v_{i_3}(v_j, \ldots), v_{p_1}, \ldots, v_{p_r}).$$

Aplikací pravidla řetězení na tuto rovnost dostáváme

$$\begin{aligned} \frac{\partial v_l}{\partial v_j} &= \frac{\partial v_l}{\partial v_{i_1}} \cdot \frac{\partial v_{i_1}}{\partial v_j} + \frac{\partial v_l}{\partial v_{i_2}} \cdot \frac{\partial v_{i_2}}{\partial v_j} + \frac{\partial v_l}{\partial v_{i_3}} \cdot \frac{\partial v_{i_3}}{\partial v_j} + \\ &\quad \frac{\partial v_l}{\partial v_{p_1}} \cdot \frac{\partial v_{p_1}}{\partial v_j} + \cdots + \frac{\partial v_l}{\partial v_{p_r}} \cdot \frac{\partial v_{p_r}}{\partial v_j}. \end{aligned}$$

Protože však $v_{p_1}, \ldots, v_{p_r}$ nezávisí na v_j, je

$$\frac{\partial v_{p_1}}{\partial v_j} = \cdots = \frac{\partial v_{p_r}}{\partial v_j} = 0.$$

Odtud již

$$\frac{\partial v_l}{\partial v_j} = \frac{\partial v_l}{\partial v_{i_1}} \cdot \frac{\partial v_{i_1}}{\partial v_j} + \frac{\partial v_l}{\partial v_{i_2}} \cdot \frac{\partial v_{i_2}}{\partial v_j} + \frac{\partial v_l}{\partial v_{i_3}} \cdot \frac{\partial v_{i_3}}{\partial v_j}.$$

V obecném případě

$$\frac{\partial v_l}{\partial v_j} = \sum_{i \ ; \ i \succ j} \frac{\partial v_l}{\partial v_i} \cdot \frac{\partial v_i}{\partial v_j},$$

neboli

$$\bar{v}_j = \sum_{i \ ; \ i \succ j} \bar{v}_i \cdot \frac{\partial \varphi_i}{\partial v_j}.$$

Tím je důkaz proveden. □

Z právě dokázaného vztahu (1.4) vyplývá, že pro výpočet hodnoty $\bar{v}_j$ stačí znát (mimo hodnot příslušných parciálních derivací) hodnoty proměnných $\bar{v}_i$ pro $i \succ j$, neboli $\bar{v}_l$, $\bar{v}_{l-1}$, …, $\bar{v}_{j+1}$.

Tohoto pozorování využívá algoritmus výpočtu hodnot derivací pomocí zpětného módu: nejdříve se přiřadí $\bar{v}_l = 1$. Dále se postupně počítají hodnoty proměnných $\bar{v}_{l-1}$, $\bar{v}_{l-2}$, …, $\bar{v}_1$, $\bar{v}_0$, $\bar{v}_{-1}$, …, $\bar{v}_{1-n}$ z již spočtených hodnot. Protože však

$$\bar{v}_0 = \frac{\partial v_l}{\partial v_0} = \frac{\partial f(x)}{\partial x_n}, \quad \ldots, \quad \bar{v}_{1-n} = \frac{\partial v_l}{\partial v_{1-n}} = \frac{\partial f(x)}{\partial x_1},$$

v_{i-n}	$= x_i$	$i = 1, \ldots, n$
v_i	$= \varphi_i(v_k)_{k \prec i}$	$i = 1, \ldots, l$
y	$= v_l$	
$\bar{v}_l$	$= 1$	
for $j = l-1, \ldots, 1-n$ do		
$\bar{v}_j$	$= \sum_{i; i \succ j} \bar{v}_i \cdot \frac{\partial \varphi_i}{\partial v_j}(v_k)_{k \prec i}$	
end for		
$\frac{\partial f(x)}{\partial x_i}$	$= \bar{v}_{i-n}$	$i = n, \ldots, 1$

Obrázek 1.6: Soupis operací odvozený sčítaným zpětným módem automatického derivování.

spočítali jsme tak celý gradient $\nabla f(x) = (\bar{v}_{1-n}, \ldots, \bar{v}_0)$. Celý algoritmus pro výpočet derivace pomocí zpětného módu je uveden na obrázku 1.6.

Tento postup výpočtu derivace nazveme *sčítaným* zpětným módem. V jeho první části probíhá výpočet hodnot v_{1-n}, …, v_l, tedy $f(x) = v_l$, ve druhé části probíhá výpočet hodnot $\bar{v}_l$, …, $\bar{v}_{1-n}$, tedy $\nabla f(x) = (\bar{v}_{1-n}, \ldots, \bar{v}_0)$. Výpočet hodnot probíhá v pořadí $\bar{v}_l$, $\bar{v}_{l-1}$, …, $\bar{v}_{1-n}$, proto se tento mód automatického derivování nazývá zpětný.

Praktická realizace tohoto postupu není jednoduchá, protože obvykle neznáme ty indexy i, pro které $i \succ j$, neboli indexy těch proměnných v_i, při jejichž výpočtu se použila hodnota v_j. Tuto nevýhodu lze poměrně jednoduše odstranit přeformulací sumace, jak je ukázáno v soupisu operací na obrázku 1.7. Tento postup nazveme *nasčítávaným* zpětným módem. Jeho numerické výsledky jsou stejné jako numerické výsledky získané sčítaným zpětným módem.

v_{i-n}	$= x_i$	$i = 1, \ldots, n$
v_i	$= \varphi_i(v_k)_{k \prec i}$	$i = 1, \ldots, l$
y	$= v_l$	
$\bar{v}_l$	$= 1$	
$\bar{v}_i$	$= 0$	$i = l-1, \ldots, 1-n$
for $i = l, \ldots, 1$ do		
for $j \prec i$ do		
$\bar{v}_j = \bar{v}_j + \bar{v}_i \cdot \frac{\partial \varphi_i}{\partial v_j}(v_k)_{k \prec i}$		
end for		
end for		
$\frac{\partial f(x)}{\partial x_i}$	$= \bar{v}_{i-n}$	$i = n, \ldots, 1$

Obrázek 1.7: Soupis operací odvozený nasčítávaným zpětným módem automatického derivování.

Poznámka. Předposlední krok v právě uvedeném nasčítávaném zpětném módu (obrázek 1.7), totiž

for $i = l, \ldots, 1$ do
  for $j \prec i$ do
    $\bar{v}_j = \bar{v}_j + \bar{v}_i \cdot \frac{\partial \varphi_i}{\partial v_j}(v_k)_{k \prec i}$
  end for
end for,

ve kterém dochází k vlastnímu výpočtu hodnot $\bar{v}_s$, lze popsat jako:

- Ber postupně jednu operaci za druhou v opačném pořadí, než když se počítala hodnota $f(x)$, tj. vezmi nejdříve $v_l = \varphi_l(v_k)_{k \prec l}$, pak $v_{l-1} = \varphi_{l-1}(v_k)_{k \prec l-1}$, ..., a nakonec $v_1 = \varphi_1(v_k)_{k \prec 1}$. Pro každou z těchto operací udělej:
  - Pro každou z proměnných na pravé straně, tj. pro každou $v_j, j \prec i$ (zpracováváme-li i-tou operaci $v_i = \varphi_i(v_k)_{k \prec i}$), udělej:
    * K proměnné $\bar{v}_j$ přičti $\bar{v}_i \cdot \frac{\partial \varphi_i}{\partial v_j}(v_k)_{k \prec i}$.

□

Tuto operaci

$$\bar{v}_j = \bar{v}_j + \bar{v}_i \cdot \frac{\partial \varphi_i}{\partial v_j}(v_k)_{k \prec i}$$

nazývejme *přidruženou* operací k operaci

$$v_i = \varphi_i(v_k)_{k \prec i}$$

a proměnné $\bar{v}_i$ nazývejme *přidružené* proměnné.

Demonstrace na příkladu I. Navážeme na výše uvedený příklad I ze strany 15. Chceme spočíst gradient funkce f v bodě $(\pi/4, 1)$.

Podle výše uvedeného postupu dostáváme nasčítávaný zpětný mód, který je uveden na obrázku 1.8. Tím jsme spočetli hodnotu $y = f(\pi/4, 1) = v_4 = 1.6857$ a gradient $\nabla f(\pi/4, 1) = (\bar{v}_{-1}, \bar{v}_0) = (0.7540, -0.1149)$. □

Zatím jsme se omezili na situaci, kdy $y \in \mathbb{R}$ byl skalár. Jak uvidíme v dalším odstavci, je možné použít zpětný mód automatického derivování i pro funkce $y = f(x) \in \mathbb{R}^m$, $m > 1$. Spočteme však pouze gradient jedné složky vektoru y, resp. gradient nějaké lineární kombinace složek vektoru y.

Obecně zpětný mód počítá hodnotu součinu

$$(\bar{v}_{l-m+1}, \bar{v}_{l-m+2}, \ldots, \bar{v}_l) \cdot f'(x),$$

kde $f'(x)$ značí Jacobiho matici funkce $f(x)$ a $\bar{v}_{l-m+1}, \bar{v}_{l-m+2}, \ldots, \bar{v}_l$ jsou předem zadané koeficienty. Zpětný mód tedy počítá lineární kombinaci gradientů jednotlivých složek funkce $f(x)$, neboli gradient součinu$(\bar{v}_{l-m+1}, \bar{v}_{l-m+2}, \ldots, \bar{v}_l) \cdot f(x)$. Pokud chceme spočítat gradient například předposlední složky vektoru y, pak na začátku zpětného módu přiřadíme $\bar{v}_l = 0$, $\bar{v}_{l-1} = 1$, $\bar{v}_{l-2} = 0$, ..., $\bar{v}_{l-m+1} = 0$.

v_{-1}	$=$	x_1	$=$	$\frac{\pi}{4} = 0.7854$
v_0	$=$	x_2	$=$	1.0000
v_1	$=$	$\sin v_{-1}$	$=$	0.7071
v_2	$=$	$v_{-1} * v_0$	$=$	0.7854
v_3	$=$	v_1/v_2	$=$	0.9003
v_4	$=$	$v_2 + v_3$	$=$	1.6857
y	$=$	v_4	$=$	1.6857
$\bar{v}_4$	$=$	1.0000		
$\bar{v}_i$	$=$	0		$i = 3, \ldots, -1$
$\bar{v}_3$	$=$	$\bar{v}_3 + \bar{v}_4$	$=$	1.0000
$\bar{v}_2$	$=$	$\bar{v}_2 + \bar{v}_4$	$=$	1.0000
$\bar{v}_1$	$=$	$\bar{v}_1 + \bar{v}_3/v_2$	$=$	1.2732
$\bar{v}_2$	$=$	$\bar{v}_2 + \bar{v}_3 * (-v_1/v_2^2)$	$=$	
	$=$	$\bar{v}_2 - \bar{v}_3 * v_3/v_2$	$=$	-0.1463
$\bar{v}_0$	$=$	$\bar{v}_0 + \bar{v}_2 * v_{-1}$	$=$	-0.1149
$\bar{v}_{-1}$	$=$	$\bar{v}_{-1} + \bar{v}_2 * v_0$	$=$	-0.1463
$\bar{v}_{-1}$	$=$	$\bar{v}_{-1} + \bar{v}_1 * \cos v_{-1}$	$=$	0.7540
$\frac{\partial f}{\partial x_1}$	$=$	$\bar{v}_{-1}$	$=$	0.7540
$\frac{\partial f}{\partial x_2}$	$=$	$\bar{v}_0$	$=$	-0.1149

Obrázek 1.8: Soupis operací odvozený nasčítávaným zpětným módem automatického derivování pro Příklad I.

Poznámka. V následujícím odstavci odvodíme zpětný mód automatického derivování z přímého módu automatického derivování pomocí maticového vyjádření Jacobiho matice $f'(x)$ jako součinu matic, jejichž prvky jsou parciální derivace elementárních funkcí $\varphi_i(x)$. Na toto maticové vyjádření Jacobiho matice $f'(x)$ lze pohlížet jako na součin matic, jaký bychom dostali pomocí pravidla řetězení (neboli derivace složené funkce) pro vektorové funkce.

Odvození zpětného módu z přímého módu

Přímý mód automatického derivování, jak jsme ho popsali výše na obrázku 1.4, lze přepsat jako součin matic a vektoru $\dot{x}$. Pro $i = 1, \ldots, l$ a pro $j \prec i$ označme

$$c_{ij} = \frac{\partial \varphi_i}{\partial v_j}(v_k)_{k \prec i}.$$

Pro $i = 1, \ldots, l$ značme dále

$$A_i = \left(\begin{array}{cccc|c|ccc} 1 & 0 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & 0 & 0 & \cdots & 0 \\ \hline c_{i,1-n} & c_{i,2-n} & \cdots & c_{i,i-1} & 0 & 0 & \cdots & 0 \\ \hline 0 & 0 & \cdots & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & 0 & 0 & \cdots & 1 \end{array}\right) \in \mathbb{R}^{(n+l)\times(n+l)}, \tag{1.5}$$

kde c_{ij} jsou na $(n+i)$-tém řádku.

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

$$P_n = (I_n, 0, \ldots, 0) \in \mathbb{R}^{n\times(n+l)} \tag{1.6}$$

$$Q_m = (0, \ldots, 0, I_m) \in \mathbb{R}^{m\times(n+l)} \tag{1.7}$$

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

Z přímého módu automatického derivování vyplývá, že hodnotu derivace

$$\dot{y} = f'(x) \cdot \dot{x}$$

můžeme vyjádřit jako součin matic tvaru

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

Srovnáním těchto vztahů získáváme reprezentaci Jacobiho matice $f'(x)$ jako součin matic ve tvaru

$$f'(x) = Q_m A_l A_{l-1} \cdots A_2 A_1 P_n^T \in \mathbb{R}^{m\times n}. \tag{1.8}$$

Jak jsme již naznačili, zpětným módem chceme dostat derivace ve tvaru

$$\bar{x} = \bar{y} \cdot f'(x). \tag{1.9}$$

Zde jsme pro jednoduchost zavedli nové proměnné, kde $\bar{y}$ je pouze zkratka za proměnné $(\bar{v}_{l-m+1}, \bar{v}_{l-m+2}, \ldots, \bar{v}_l)$. Obě strany výrazu (1.9) transponujme. S využitím vyjádření vztahu (1.8) tak vlastně chceme spočíst hodnotu součinu

$$\bar{x}^T = P_n A_1^T A_2^T \cdots A_{l-1}^T A_l^T Q_m^T \bar{y}^T. \tag{1.10}$$

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

v_{i-n}	$= x_i$	$i = 1, \ldots, n$
v_i	$= \varphi_i(v_k)_{k \prec i}$	$i = 1, \ldots, l$
y_{m-i}	$= v_{l-i}$	$i = m-1, \ldots, 0$
$\bar{v}_{l-i}$	$= \bar{y}_{m-i}$	$i = 0, \ldots, m-1$
$\bar{v}_i$	$= 0$	$i = l-m, \ldots, 1-n$

for $i = l, \ldots, 1$ do
  for $j \prec i$ do
  $\bar{v}_j = \bar{v}_j + \bar{v}_i \cdot \frac{\partial \varphi_i}{\partial v_j}(v_k)_{k \prec i}$
  end for
end for		

$\bar{x}_i$	$= \bar{v}_{i-n}$	$i = n, \ldots, 1$

Obrázek 1.9: Soupis operací odvozený nasčítávaným zpětným módem automatického derivování.

Odvozený vzorec je možné přepsat do postupu na obrázku 1.9. Tento postup je v podstatě shodný s postupem, který jsme odvodili pomocí pravidla řetězení (obrázek 1.7) – nyní však přímo již pro situaci $y \in \mathbb{R}^m$, $m > 1$.

Poznamenejme, že jsme spočítali hodnotu součinu $\bar{y} \cdot f'(x)$ a také funkční hodnotu $f(x)$, ale nespočetli jsme hodnotu $f'(x)$ explicitně.

Demonstrace na příkladu I. Navážeme na výše uvedený příklad I ze strany 19. Platí, že $n = 2$, $m = 1$ a $l = 4$. Matice (1.6) a (1.7) jsou tedy rovny

$$P_2 = \left(\begin{array}{cc|cccc} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{array}\right) \in \mathbb{R}^{2\times 6}$$

a

$$Q_1 = \left(\begin{array}{ccccc|c} 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right) \in \mathbb{R}^{1\times 6}.$$

Dále, pro matice (1.5) platí

$$A_1 = \left(\begin{array}{cc|c|ccc} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ \hline \frac{\partial \varphi_1}{\partial v_{-1}} & \frac{\partial \varphi_1}{\partial v_0} & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right) = \left(\begin{array}{cc|c|ccc} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ \hline \cos v_{-1} & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right)$$

$$= \left(\begin{array}{cc|c|ccc} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ \hline 0.7071 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right) \in \mathbb{R}^{6\times 6}$$

a dále

$$A_2 = \left(\begin{array}{ccc|c|cc} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ \hline \frac{\partial\varphi_2}{\partial v_{-1}} & \frac{\partial\varphi_2}{\partial v_0} & \frac{\partial\varphi_2}{\partial v_1} & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right) = \left(\begin{array}{ccc|c|cc} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ \hline v_0 & v_{-1} & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right)$$

$$= \left(\begin{array}{ccc|c|cc} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ \hline 1 & 0.7854 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right) \in \mathbb{R}^{6\times 6},$$

$$A_3 = \left(\begin{array}{cccc|c|c} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ \hline \frac{\partial\varphi_3}{\partial v_{-1}} & \frac{\partial\varphi_3}{\partial v_0} & \frac{\partial\varphi_3}{\partial v_1} & \frac{\partial\varphi_3}{\partial v_2} & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right) = \left(\begin{array}{cccc|c|c} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 1/v_2 & -v_1/v_2^2 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right)$$

$$= \left(\begin{array}{cccc|c|c} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 1.2732 & -1.1463 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 1 \end{array}\right) \in \mathbb{R}^{6\times 6}$$

a konečně

$$A_4 = \left(\begin{array}{ccccc|c} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ \hline \frac{\partial\varphi_4}{\partial v_{-1}} & \frac{\partial\varphi_4}{\partial v_0} & \frac{\partial\varphi_4}{\partial v_1} & \frac{\partial\varphi_4}{\partial v_2} & \frac{\partial\varphi_4}{\partial v_3} & 0 \end{array}\right) = \left(\begin{array}{ccccc|c} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 & 1 & 0 \end{array}\right) \in \mathbb{R}^{6\times 6}.$$

Dosadíme-li nyní do (1.10) ještě $\bar{y} = 1$, dostáváme

$$\bar{x}^T = P_n A_1^T A_2^T \cdots A_{l-1}^T A_l^T Q_m^T \bar{y}^T = \begin{pmatrix} 0.7540 \\ -0.1149 \end{pmatrix},$$

a tím jsme tedy spočetli $(\bar{x}_1, \bar{x}_2) = \nabla f(\pi/4, 1) = (0.7540, -0.1149)$. □

Zatím jsme na proměnné $\bar{v}_i$ pohlíželi jako na pomocné proměnné. Lze ukázat, že jejich výsledné hodnoty charakterizují citlivost $\bar{y} \cdot f(x)$ na malou změnu v_i. Přesně řečeno, změňme operaci $v_i = \varphi_i(v_k)_{k\prec i}$ ze soupisu operací na obrázku 1.1 na operaci

$$v_i = \varphi_i(v_k)_{k\prec i} + \delta_i, \tag{1.11}$$

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

$$\bar{v}_i = \frac{\partial}{\partial \delta_i}(\bar{y} \cdot f(x))\bigg|_{\delta_i=0}. \tag{1.12}$$

Podrobnosti lze najít například v [5].

Je také možné spočíst gradienty s q různými váhami najednou, pokud proměnnou $\bar{y}$ budeme uvažovat jako matici, v jejíž řádcích jsou uloženy jednotlivé váhové vektory. V tom případě chápeme proměnné $\bar{y}_i$ jako vektory.

1.5 Výpočet druhých a vyšších derivací

V tomto odstavci popíšeme, jak použít automatické derivování pro výpočet druhých nebo vyšších derivací pomocí aplikace zpětného a přímého módu.

1.5.1 Vlastnosti a odvození výpočtu druhých a vyšších derivací

Pro přehlednost zopakujme, že nezávislé a závislé proměnné při výpočtu hodnot $y = f(x)$ jsou označeny takto:

$$\underbrace{v_{1-n}, \ldots, v_0}_{=x}, v_1, v_2, \ldots, v_{l-m}, \underbrace{v_{l-m+1}, \ldots, v_l}_{=y}.$$

Program pro výpočet druhých (nebo vyšších) derivací je možné odvodit kombinací přímého a zpětného módu. Nejdříve na zadaný program aplikujeme zpětný mód, odkud získáme program pro výpočet gradientu funkce f (nebo gradientu váženého součtu $\bar{y} \cdot f(x)$). Na tento program aplikujeme přímý mód a získáme tak program pro vyhodnocení druhých derivací ve tvaru

$$\bar{y} \cdot f''(x) \cdot \dot{x} + \dot{\bar{y}} \cdot f'(x), \tag{1.13}$$

kde $\dot{\bar{y}}$ je vstupní parametr, podobně jako $\dot{x}$ nebo $\bar{y}$. Výraz (1.13) označme jako $\dot{\bar{x}}$. Abychom spočetli pouze druhou derivaci tvaru $\bar{y} \cdot f''(x) \cdot \dot{x}$, je potřeba předem přiřadit $\dot{\bar{y}} = 0$. Výraz $\bar{y} \cdot f''(x) \cdot \dot{x}$ chápeme v této souvislosti jako

$$\bar{y} \cdot f''(x) \cdot \dot{x} = \frac{\partial}{\partial \alpha}\ \bar{y} \cdot f'(x + \alpha \dot{x})\bigg|_{\alpha=0} \in \mathbb{R}^n. \tag{1.14}$$

Příslušný soupis operací získáme například tím způsobem, že aplikujeme přímý mód na soupis operací získaný nasčítávaným zpětným módem z obrázku 1.9.

Z výrazu

$$\bar{v}_j = \bar{v}_j + \bar{v}_i \cdot \frac{\partial \varphi_i}{\partial v_j}(v_k)_{k \prec i}$$

přitom odvodíme

$$\dot{\bar{v}}_j = \dot{\bar{v}}_j + \dot{\bar{v}}_i \cdot \frac{\partial \varphi_i}{\partial v_j}(v_k)_{k \prec i} + \bar{v}_i \left(\cdot \frac{\partial \varphi_i}{\partial v_j}(v_k)_{k \prec i} \right)', \tag{1.15}$$

kde

$$\left(\frac{\partial \varphi_i}{\partial v_j}(v_k)_{k\prec i}\right)' = \sum_{l\prec i} \frac{\partial^2 \varphi_i}{\partial v_j \partial v_l}(v_k)_{k\prec i} \cdot \dot{v}_l.$$

Výsledný soupis operací je na obrázku 1.10. Po provedení tohoto soupisu operací jsou požadované hodnoty druhých derivací $\bar{y} \cdot f''(x) \cdot \dot{x}$ uloženy v proměnných

$$\dot{\bar{x}} = (\dot{\bar{x}}_1, \ldots, \dot{\bar{x}}_m).$$

$$\left.\begin{array}{lcl} v_{i-n} & = & x_i \\ \dot{v}_{i-n} & = & \dot{x}_i \end{array}\right\} \quad i = 1, \ldots, n$$

$$\left.\begin{array}{lcl} v_i & = & \varphi_i(v_k)_{k\prec i} \\ \dot{v}_i & = & \sum_{j\prec i} \frac{\partial \varphi_i}{\partial v_j}(v_k)_{k\prec i} \cdot \dot{v}_j \end{array}\right\} \quad i = 1, \ldots l$$

$$\left.\begin{array}{lcl} y_{m-i} & = & v_{l-i} \\ \dot{y}_{m-i} & = & \dot{v}_{l-i} \end{array}\right\} \quad i = m-1, \ldots 0$$

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

$$\left.\begin{array}{lcl} \bar{v}_i & = & 0 \\ \dot{\bar{v}}_i & = & 0 \end{array}\right\} \quad i = 1-n, \ldots, l-m$$

for $j = l, \ldots, 1$ do
  for $j \prec i$
    $\bar{v}_j = \bar{v}_j + \bar{v}_i \cdot \frac{\partial \varphi_i}{\partial v_j}(v_k)_{k\prec i}$
    $\dot{\bar{v}}_j = \dot{\bar{v}}_j + \dot{\bar{v}}_i \cdot \frac{\partial \varphi_i}{\partial v_j}(v_k)_{k\prec i} + \bar{v}_i \cdot \left(\sum_{l\prec i} \frac{\partial^2 \varphi_i}{\partial v_j \partial v_l}(v_k)_{k\prec i} \cdot \dot{v}_l\right)$
  end for
end for

$$\left.\begin{array}{lcl} \bar{x}_i & = & \bar{v}_{i-n} \\ \dot{\bar{x}}_i & = & \dot{\bar{v}}_{i-n} \end{array}\right\} \quad i = n, \ldots, 1$$

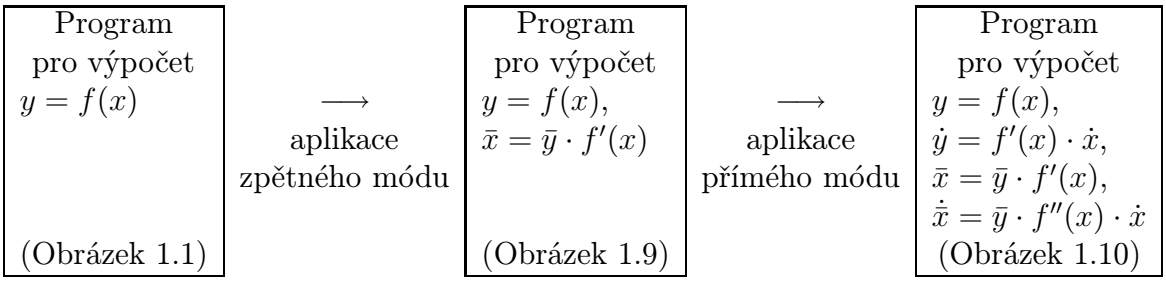
Obrázek 1.10: Soupis operací pro výpočet druhých derivací, odvozený nasčítávaným zpětným módem a následnou aplikací přímého módu.

Při vyhodnocení druhých derivací jsme "jako vedlejší produkt" také spočetli hodnotu prvních derivací tvaru $f'(x) \cdot \dot{x}$ (jako kdybychom provedli přímý mód) a také $\bar{y} \cdot f'(x)$ (jako kdybychom provedli zpětný mód).

Názorné schéma odvození programů pro výpočet druhé derivace je znázorněno na obrázku 1.11.

Podobně jako v přímém módu, je možné spočíst derivace podle více směrů zároveň, pokud budeme uvažovat proměnnou $\dot{x}$ jako maticovou.

Vyšší než druhé derivace mohou být získány analogicky jednou aplikací zpětného módu a následně několikanásobnou aplikací přímých módů.



Obrázek 1.11: Schématické znázornění odvození programu pro výpočet druhé derivace.

Demonstrace na příkladu I. Navážeme na výše uvedený příklad I ze strany 22. Chceme spočíst hodnotu druhých parciálních derivací funkce f v bodě $(\pi/4, 1)$.

Nejdříve na soupis operací pro vyhodnocení funkce f na obrázku 1.2 aplikujeme zpětný mód s hodnotou $\bar{y} = \bar{v}_4 = 1$. Dostaneme tak soupis operací na obrázku 1.8, na který aplikujeme přímý mód s hodnotami $\dot{x}_1 = 1$, $\dot{x}_2 = 0$ a $\dot{\bar{y}} = 0$. Získáme tak soupis operací na obrázku 1.12, který spočte hodnoty $\dot{\bar{x}}_1 = \partial^2/\partial x_1^2 \; f(\pi/4, 1)$ a $\dot{\bar{x}}_2 = \partial^2/(\partial x_2 \partial x_1) \; f(\pi/4, 1)$.

Pro výpočet zbývajících dvou hodnot parciálních derivací $\partial^2/(\partial x_1 \partial x_2) \; f(\pi/4, 1)$ a $\partial^2/\partial x_2^2 \; f(\pi/4, 1)$ stačí provést tentýž soupis operací (obrázek 1.12), ale s počátečním nastavením hodnot $\bar{y} = \bar{v}_4 = 1$, $\dot{x}_1 = 0$, $\dot{x}_2 = 1$ a $\dot{\bar{y}} = 0$.

Jak jsme již poznamenali výše, kromě hodnot druhých derivací jsme spočetli také hodnoty prvních derivací ve tvaru $f'(\pi/4, 1) \cdot \dot{x}$ a $\bar{y} \cdot f'(\pi/4, 1)$, tj. hodnoty derivací, které jsme původně získali jen přímým a jen zpětným módem. □

1.6 Způsoby implementace automatického derivování

V předchozích odstavcích jsme popsali základní principy automatického derivování. V tomto odstavci ukážeme, jak tyto metody prakticky aplikovat na zadaný program, aby počítal i požadované derivace.

Tato kapitola si neklade za cíl a ani nemůže být detailním popisem implementace automatického derivování. Spíše ukážeme základní techniky a praktické postřehy. Konkrétní implementace je ostatně závislá na použitém systému.

Automatické derivování je *transformace programu*: program pro vyhodnocení funkčních hodnot $f(x)$ je transformován na program pro vyhodnocení funkčních hodnot $f(x)$ a jejich derivací, za použití přímého nebo zpětného módu popsaného výše. Tato transformace programu lze provést ručně nebo (v lepším případě) jiným programem. Tato transformace může být provedena *automaticky* bez zásahu uživatele, odkud také pochází pojmenování *automatické* derivování. To je jedna z velkých výhod popisovaných metod.

V předcházejících odstavcích jsme popsali transformaci soupisu operací, ale vstupem je program. Tento rozpor snadno odstraníme vysvětlením principů transformace

v_{-1}	=	x_1	=	$\frac{\pi}{4} = 0.7854$
$\dot{v}_{-1}$	=	$\dot{x}_1$	=	1.0000
v_0	=	x_2	=	1.0000
$\dot{v}_0$	=	$\dot{x}_2$	=	0.0000
v_1	=	$\sin v_{-1}$	=	0.7071
$\dot{v}_1$	=	$\cos v_{-1} * \dot{v}_{-1}$	=	0.7071
v_2	=	$v_{-1} * v_0$	=	0.7854
$\dot{v}_2$	=	$v_{-1} * \dot{v}_0 + v_0 * \dot{v}_{-1}$	=	1.0000
v_3	=	v_1 / v_2	=	0.9003
$\dot{v}_3$	=	$(\dot{v}_1 - v_3 * \dot{v}_2)/v_2$	=	-0.2460
v_4	=	$v_2 + v_3$	=	1.6857
$\dot{v}_4$	=	$\dot{v}_2 + \dot{v}_3$	=	0.7540
y	=	v_4	=	1.6857
$\dot{y}$	=	$\dot{v}_4$	=	0.7540
$\bar{v}_4$	=	$\bar{y}$	=	1.0000
$\dot{\bar{v}}_4$	=	$\dot{\bar{y}}$	=	0.0000
$\bar{v}_i$	=	0		$i = 3, \ldots, -1$
$\dot{\bar{v}}_i$	=	0		$i = 3, \ldots, -1$
$\bar{v}_3$	=	$\bar{v}_3 + \bar{v}_4$	=	1.0000
$\dot{\bar{v}}_3$	=	$\dot{\bar{v}}_3 + \dot{\bar{v}}_4$	=	0.0000
$\bar{v}_2$	=	$\bar{v}_2 + \bar{v}_4$	=	1.0000
$\dot{\bar{v}}_2$	=	$\dot{\bar{v}}_2 + \dot{\bar{v}}_4$	=	0.0000
$\bar{v}_1$	=	$\bar{v}_1 + \bar{v}_3 / v_2$	=	1.2732
$\dot{\bar{v}}_1$	=	$\dot{\bar{v}}_1 + \dot{\bar{v}}_3 / v_2 - \bar{v}_3 * \dot{v}_2 / v_2^2$	=	-1.6211
$\bar{v}_2$	=	$\bar{v}_2 - \bar{v}_3 * v_3 / v_2$	=	-0.1463
$\dot{\bar{v}}_2$	=	$\dot{\bar{v}}_2 - \dot{\bar{v}}_3 * v_3 / v_2 - \bar{v}_3 * (\dot{v}_1 - 2 v_3 * \dot{v}_2)/v_2^2$	=	1.7727
$\bar{v}_0$	=	$\bar{v}_0 + \bar{v}_2 * v_{-1}$	=	-0.1149
$\dot{\bar{v}}_0$	=	$\dot{\bar{v}}_0 + \dot{\bar{v}}_2 * v_{-1} + \bar{v}_2 * \dot{v}_{-1}$	=	1.2460
$\bar{v}_{-1}$	=	$\bar{v}_{-1} + \bar{v}_2 * v_0$	=	-0.1463
$\dot{\bar{v}}_{-1}$	=	$\dot{\bar{v}}_{-1} + \dot{\bar{v}}_2 * v_0 + \bar{v}_2 * \dot{v}_0$	=	1.7727
$\bar{v}_{-1}$	=	$\bar{v}_{-1} + \bar{v}_1 * \cos v_{-1}$	=	0.7540
$\dot{\bar{v}}_{-1}$	=	$\dot{\bar{v}}_{-1} + \dot{\bar{v}}_1 * \cos v_{-1} - \bar{v}_1 * \sin v_{-1} * \dot{v}_{-1}$	=	-0.2739
$\partial^2 f / \partial x_1^2$	=	$\dot{\bar{v}}_{-1}$	=	1.2460
$\partial^2 f / \partial x_2 \partial x_1$	=	$\dot{\bar{v}}_0$	=	-0.2739

Obrázek 1.12: Soupis operací pro výpočet druhých derivací, odvozený zpětným módem a následnou aplikací přímého módu pro Příklad I.

a nepředstavuje žádný problém.

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.

Kromě vlastní transformace zadaného programu je nutné ke ztransformovanému programu připojit soubor (balík) pomocných procedur a funkcí, které budou tímto programem využívány. Jedná se například o deklarace nových typů proměnných, jejich inicializace, pomocné procedury nebo pozměněné definice základních funkcí (aby například obsahovaly i jejich sdružené operace). Podrobnosti budou ukázany v dalších odstavcích.

Základními metodami implementace automatického derivování jsou *přetížení operátorů a funkcí* nebo *použitím preprocesoru*, které v následujících odstavcích popíšeme.

1.6.1 Přetížení operátorů a funkcí

Technika přetížení operátorů (například operátorů $+$, $*$ nebo $=$) nebo funkcí (například funkce sinus nebo funkcí definovaných uživatelem) 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 podprogramu `PP(ProměnnáTypuA)`, resp. `PP(ProměnnáTypuB)` překladač podle typu parametru rozliší, jakou definici procedury `PP` má zavolat.

Výhoda použití technik přetížení je, že téměř celý zadaný program může být při aplikaci automatického derivování ponechán beze změny.

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. Více informací o jazyce C++ je možné nalézt například v [25].

První derivace - přímý mód

Popíšeme jednoduchý postup, jak pomocí přetížení operátorů a funkcí implementovat přímý mód automatického derivování, který jsme odvodili v kapitole 1.3. Tímto způsobem vyhodnotíme derivaci v zadaném směru.

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

```
class doublet       // definice třídy doublet
{                   //  obsahující funkční hodnotu
                    //  a hodnotu derivace
 public:
   double v;        // funkční hodnota v_i
   double vdot;     // hodnota derivace v_i s tečkou
```

[2] V celé kapitole 1.6 budeme pro jednoduchost předpokládat, že všechny v_i jsou skaláry.

```
};
```

Provedeme-li nyní deklaraci

```
doublet a;
```

nadefinujeme tím proměnnou `a` typu `doublet`. Na její složku `v` se pak odvoláme pomocí výrazu `a.v` a bude v ní uložena hodnota v_i. Na složku `vdot` se odvoláme pomocí `a.vdot` a bude v ní uložena hodnota derivace $\dot{v}_i$.

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

```
doublet sin (doublet a)        // definice funkce sinus, jejímž
                               //  vstupem je proměnná typu doublet
{                              //  a výstupem proměnná typu doublet
 doublet b;
 b.v = sin (a.v);              // výpočet funkční hodnoty
 b.vdot = cos (a.v) * a.vdot;  // výpočet derivace
 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`.

Operátor `*` můžeme přetížit analogickým způsobem:

```
doublet operator* (doublet a, doublet b) // definice operátoru násobení, jehož
                                         //  vstupy jsou proměnné typu doublet
{                                        //  a výstupem proměnné typu doublet
  doublet c;
  c.v = a.v * b.v;                      // výpočet funkční hodnoty
  c.vdot = a.vdot * b.v + a.v * b.vdot; // výpočet derivace
  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 operátor `*` (a vůbec binární operátory) potřeba nadefinovat smíšené operace pro konstanty[3] (například typu `double`) a typ `doublet`.

Operátor dělení `/` přetížíme například následujícím způsobem:

```
doublet operator/ (doublet a, doublet b) // definice operátoru dělení, jehož
                                         //  vstupy jsou proměnné typu doublet
{                                        //  a výstupem proměnné typu doublet
  doublet c;
```

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

```
    c.v = a.v / b.v;                          // výpočet funkční hodnoty
    c.vdot = (a.vdot * b.v - a.v * b.vdot)/(b.v * b.v);
                                              // výpočet derivace
    return c;
  };
```

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

- Zavedení nového typu proměnné (nazvali jsme ho `doublet`), který obsahuje nejen hodnotu v_i, ale i její derivaci $\dot{v}_i$.
- Přetížení obvyklých operátorů (funkcí) prováděných na běžné typy proměnných, aby pracovaly i s parametry typu `doublet`. Tyto nové operace provádějí navíc i sdružené operace podle pravidla řetězení s využitím složky `vdot`, která reprezentuje $\dot{v}_i$.
- Zajištění správné inicializace obou složek proměnné typu `doublet` a naopak vrácení (vyexportování) hodnoty `v` a její derivace `vdot`.

Je jasné, že musíme změnit i derivovaný program. Přirozeně 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 vypsána (vyexportována) také hodnota příslušné derivace.

Řídící příkazy (podmínky, cykly…) se přitom nemění. Pouze v případech, kdy například v podmínce testujeme hodnotu proměnné `z`, která závisí na x, potom v přetransformovaném programu musíme testovat hodnotu `z.v`.

Je vhodné (nikoli však bezpodmínečně nutné) mít funkci, která provede přiřazení hodnoty nezávislé proměnné a její 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)
                    // definice funkce makedoublet, jejímiž
                    //  vstupy jsou proměnné typu double
                    //  a výstupem proměnná typu doublet
{
  doublet c;
  c.v = val;       // přiřazení hodnoty nezávislé proměnné
  c.vdot = dotval; // přiřazení její derivace (směru)
  return c;
};
```

Jako "opačnou" k funkci `makedoublet` je užitečné zavést funkci `value`, resp. `dotvalue`, která ze zadané proměnné typu `doublet` vrátí hodnotu její složky `v`, resp. `vdot`.

```
double value (doublet a)  // definice funkce value, jejímž
                          //  vstupem je proměnná typu doublet
{                         //  a výstupem proměnná typu double
  double d;
  d = a.v;                // vrátí funkční hodnotu v_i
  return d;
};

double dotvalue (doublet a) // definice funkce dotvalue, jejímž
                            //  vstupem je proměnná typu doublet
  {                         //  a výstupem proměnná typu double
  double d;
  d = a.vdot;  // vrátí hodnotu derivace v_i s tečkou
  return d;
};
```

Popsanou transformaci zadaného programu ukážeme opět na příkladu I.

Demonstrace na příkladu I. Navážeme na výše uvedený příklad I ze strany 26. Chceme spočítat derivaci funkce

$$f(x_1, x_2) = \frac{\sin x_1}{x_1 x_2} + x_1 x_2$$

v bodě $(\pi/4, 1)$ a také derivaci $\partial f/\partial x_1$ v tomtéž bodě. Tato funkce může být vyhodnocena pomocí následující ukázky zapsané v C++:

```
//deklarace a vstup proměnných
   double x1, x2, y;
   double temp;
   x1 = pi/4;
   x2 = 1;
//výpočet
   temp = x1 * x2;
   y = sin(x1) / temp + temp;
//výstup proměnných
   cout << y;
```

Poznamenejme, že příkaz `cout << y;` je operace, která zapíše hodnoty proměnných na standardní výstup.

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

```
//deklarace a vstup proměnných
   doublet x1, x2, y;                          *
   doublet temp;                               *
   x1 = makedoublet (pi/4, 1.0);               *
   x2 = makedoublet (1.0,  0.0);               *
//výpočet
   temp = x1 * x2;
   y = sin(x1) / temp + temp;
//výstup proměnných
   cout << value(y);                           *
   cout << dotvalue(y);                        *
```

Veškeré změny, které jsme provedli, jsou předefinování příslušných proměnných z typu `double` na `doublet`, inicializace proměnných `x1` a `x2` a výpis hodnot složek proměnné `y`. Všechny řádky, které jsou nové, nebo na kterých došlo k nějaké změně, jsme označili hvězdičkou. Jádro programu, ve kterém probíhá vlastní výpočet, zůstalo nezměněno. To je důležité, protože to bývá nejobsáhlejší a nejméně přehledná část derivovaného programu.

Aby bylo možné tento program přeložit a spustit, je nutné k němu připojit balík s definicemi pomocných procedur a funkcí a deklaracemi nových typů proměnných, jak jsme se o tom již zmínili výše v tomto odstavci. □

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 následujících kapitol.

První derivace - zpětný mód

V tomto odstavci popíšeme jednoduchou implementaci nasčítávaného zpětného módu automatického derivování, který jsme uvedli v odstavci 1.4 na obrázku 1.7, resp. 1.9. Tak budeme schopni spočítat gradient libovolné složky zadané funkce f nebo gradient lineární kombinace $\bar{y} \cdot f$ těchto složek. Nejdříve implementaci zpětného módu automatického derivování obecně popíšeme a pak ji budeme aplikovat na příklad I.

V průběhu výpočtu hodnoty funkce f v bodě x budeme zaznamenávat každou základní operaci, kterou provedeme, spolu s jejími argumenty. Tyto údaje budeme sekvenčně ukládat do dlouhého pole. Po dokončení vyhodnocení $f(x)$ projdeme tímto

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. Tento přístup není úplně vhodný pro velké úlohy, proto je možné uplatnit nějaká úsporná opatření, viz například [5] nebo [8].

Podobně jako při implementaci přímého módu, i nyní připravíme balík s pomocnými definicemi a funkcemi. Ten by měl by obsahovat následující deklarace:

- Zavedení třídy `element`, která bude uchovávat informaci o provedené elementární operaci. Bude tedy obsahovat
  - v proměnné `v` hodnotu $v_i = \varphi_i(v_j)_{j \prec i}$, viz vztah (1.1),
  - v proměnné `vbar` hodnotu $\bar{v}_i = \sum_{j \succ i} \bar{v}_j \cdot \frac{\partial \varphi_j}{\partial v_i}$, viz vztah (1.4)
  - v proměnné `opcode` označení elementární operace (například sčítání odpovídá 10, násobení 30, viz dále)
  - v proměnné `arg1` odkaz na první argument této elementární operace
  - v proměnné `arg2` odkaz na druhý argument této elementární operace

  Dále zavedeme pole `trace` těchto `element`ů pro záznam všech provedených základních operací.

  K těmto typům ještě z důvodu lepší manipulace zavedeme 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 v_i (jako jejich nepřetížení jmenovci), ale také zaznamenají své zavolání včetně svých parametrů do zmíněného pole `trace`.

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

- Zavedení funkcí pro korektní přiřazení hodnot nezávislých proměnných x_i a vah $\bar{y}_i$ do složek `*redouble` (tedy `element`u) a naopak pro výstup spočtených derivací $\bar{x}_i$.

Musíme také provést změny přímo v programu, který počítá hodnoty funkce $f(x)$. Máme snahu, aby jich bylo co nejméně, nicméně následující jsou 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 $\bar{y}_i$. Až pak může být zavolána funkce `reverse_sweep`, která spočte a vrátí (vyexportuje) hodnotu derivace.

- Musí být zavolána funkce `reverse_sweep` (druhá část soupisu operací), aby se spočetly hodnoty $\bar{v}_i$ a tím i požadované derivace.

Řídící příkazy (podmínky, cykly,...) se automatickým derivováním nemění. Pouze v případech, kdy například v podmínce testujeme hodnotu proměnné `z`, která závisí na x, v přetransformovaném programu musíme testovat hodnotu *složky* proměnné `z`, která hodnotu původní proměnné `z` obsahuje.

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

```
class element          // definice třídy element
{                      // obsahující informace o elementární operaci
  public:
    double v;          // funkční hodnota
    double vbar;       // hodnota derivace
    int opcode;        // typ elementární operace
    element *arg1;     // odkaz na první argument
    element *arg2;     // odkaz na druhý argument
};

class redouble        // definice třídy redouble
{                     // obsahující ukazatel na element
  public:
    element *ref;
};

element trace [VelkéČíslo];  // pole elementů pro uchování
                             // všech elementárních operací

element *traceptr = trace;   // ukazatel do pole na aktuální
                             // elementární operaci při jeho vyplňování
```

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

Dále potřebujeme definovat celočíselné konstanty pro jednoznačnou identifikaci 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.

Definici těchto konstant můžeme provést například takto:

```
const int
  emptyv = 0, constv = 1, indepv = 2, bplusv = 10,
  bminusv = 20, bmultv = 30, recipv = 40, ...
  expv = 50, lnv = 51, sinv = 60, ... ;
```

Přetížené elementární funkce nejen spočtou svoji hodnotu, ale také provedou zaznamenání svého provedení do pole `trace`. Po funkci sinus, násobení a dělení je můžeme definovat například tímto způsobem.

```
redouble sin (redouble a)        // definice funkce sin, jejímž
                                 //  vstupem je proměnná typu redouble
{                                //  výstupem je proměnná typu redouble
  redouble b;
  b.ref =traceptr;               // ukazatel do pole na aktuální operaci
  traceptr->v = sin (a.ref->v);  // výpočet hodnoty elementární funkce
  traceptr->vbar = 0.0;          // vynulování složky vbar, kam se bude
                                 //  nasčítávat derivace
  traceptr->opcode = sinv;       // uložení identifikace této elementární operace
  traceptr->arg1 = a.ref;        // uložení argumentu této elementární operace
  traceptr++;                    // posun na další prvek v poli
  return b;
};

redouble operator* (redouble a, redouble b) // definice funkce násobení, jejímiž
                                            //  vstupy jsou proměnné typu redouble
{                                           //  výstupem je proměnná typu redouble
  redouble c;
  c.ref =traceptr;                          // ukazatel do pole na aktuální operaci
  traceptr->v = (a.ref->v) * (b.ref->v);    // výpočet hodnoty elementární funkce
  traceptr->vbar = 0.0;                     // vynulování složky vbar, kam se bude
                                            //  nasčítávat derivace
  traceptr->opcode = bmultv;                // uložení identifikace
                                            //  této elementární operace
  traceptr->arg1 = a.ref;                   // zaznamenání prvního argumentu
                                            //  této elementární operace
  traceptr->arg2 = b.ref;                   // zaznamenání druhého argumentu
                                            //  této elementární operace
  traceptr++;                               // posun na další prvek v poli
  return c;
};

redouble operator/ (redouble a, redouble b) // definice funkce dělení, jejímiž
                                            //  vstupy jsou proměnné typu redouble
{                                           //  výstupem je proměnná typu redouble
  redouble c;
  c.ref =traceptr;                          // ukazatel do pole na aktuální operaci
  traceptr->v = (a.ref->v) / (b.ref->v);    // výpočet hodnoty elementární funkce
  traceptr->vbar = 0.0;                     // vynulování složky vbar, kam se bude
                                            //  nasčítávat derivace
  traceptr->opcode = bdivv;                 // uložení identifikace
                                            //  této elementární operace
  traceptr->arg1 = a.ref;                   // zaznamenání prvního argumentu
                                            //  této elementární operace
  traceptr->arg2 = b.ref;                   // zaznamenání druhého argumentu
                                            //  této elementární operace
  traceptr++;                               // posun na další prvek v poli
  return c;
};
```

Každá taková přetížená funkce vyčíslí hodnotu složky `v`, vynuluje hodnotu `vbar`, zaznamená se (co je to za operaci podle číselníku a hodnota se vloží do `opcode`) a nakonec ještě uloží ukazatele na své argumenty.

Pro operátory `*`, `/` (a další binární operátory) je dále potřeba nadefinovat smí-

šené operace pro konstanty[4] (například typu `double`) a proměnné typu `redouble`. Podobným způsobem přetížíme další aritmetické operace a používané funkce.

Je užitečné mít funkci, která přiřadí hodnotu nezávislé proměnné na správné místo do proměnné typu `redouble` a označí ji, že vznikla z nezávislé proměnné. Takovou funkci nazvěme `makeindepvar`.

```
redouble makeindepvar (double a)  // definice funkce makeindepvar, jejímž
                                  //  vstupem je proměnné typu double
{                                 //  výstupem je proměnná typu redouble
  redouble b;
  b.ref =traceptr;                // ukazatel do pole na aktuální operaci
  traceptr->v = a;                // přiřazení hodnoty nezávislé proměnné
  traceptr->vbar = 0.0;           // vynulování pro nasčítávání derivace
  traceptr->opcode = indepv;      // identifikace této operace
  traceptr++;                     // posun na další prvek v poli
  return b;
};
```

Dále musíme zavést 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 původně prováděné elementární funkce a zajišťuje vlastní napočítávání derivací.

```
void reverse_sweep()          // definice funkce reverse_sweep, která
{                             //  nemá žádný přímý vstup ani výstup
 double deriv;
 while (traceptr-- > trace)   // cyklus přes všechny provedené
                              //  elementární operace
                              //  od poslední k první
                              //  (polem trace odzadu dopředu)
 {
  switch(traceptr->opcode)    // podle provedené elementární operace
                              //  rozlišíme výpočet derivace
  {
   ...
   case sinv:                 // elementární operace byla sinus
    deriv = cos(traceptr->arg1->v);
    traceptr->arg1->vbar = traceptr->arg1->vbar +    // postupné načítání
                           traceptr->vbar * deriv;  // hodnot derivací
   break;
   ...
   case bmultv:               // elementární operace byla násobení
    traceptr->arg1->vbar = traceptr->arg1->vbar +               // postupné načítání
                           traceptr->vbar * traceptr->arg2->v; //  hodnot derivací
    traceptr->arg2->vbar = traceptr->arg2->vbar +               // postupné načítání
                           traceptr->vbar * traceptr->arg1->v; //  hodnot derivací
   break;
   ...
   case bdivv:                // elementární operace byla dělení
    traceptr->arg1->vbar = traceptr->arg1->vbar +                // postupné načítání
                           traceptr->vbar / traceptr->arg2->v;  //  hodnot derivací
```

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

```
      traceptr->arg2->vbar = traceptr->arg2->vbar -                // postupné načítání
                             traceptr->vbar * traceptr->arg1->v / //  hodnot derivací
                             (traceptr->arg1->v)**2;
     break;
     ...
    }
  }
};
```

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

Čtení spočtených hodnot v_i závislých proměnných bude provádět funkce `value`.

```
  double value (redouble a)   // definice funkce value, jejímž
                              //  vstupem je proměnné typu redouble
  {                           //  výstupem je proměnná typu double
    double b;
    b = a.ref->v;             // vrátí funkční hodnotu v_i
    return b;                 //  v_i = \varphi_i(v_j)
  };
```

Přiřazení příslušných přidružených hodnot $\bar{y}_i$ (tj. vah) bude zajištěno funkcí `setbarvalue`.

```
void setbarvalue(redouble a, double b) // definice funkce setbarvalue, jejímž
                                       //  vstupem jsou proměnné typu double a redouble
  {                                    //  na výstupu není žádná proměnná
    a.ref->vbar = b;                   // nastavení vah y_i s pruhem
  };
```

Tato funkce musí být provedena před zavoláním funkce `reverse_sweep`.

Hodnoty derivací $\bar{x}_i$ zjistíme po proběhnutí funkce `reverse_sweep` pomocí funkce `barvalue`.

```
  double barvalue (redouble a)     // definice funkce barvalue, jejímž
                                   //  vstupem je proměnná typu redouble
  {                                //  výstupem je proměnná typu double
    double b;
    b = a.ref->vbar;               // vrátí hodnotu derivace \bar x_i
    return b;
  };
```

Zopakujme, že v zadaném programu nemusíme dělat žádné jiné změny než předeklarování příslušných proměnných na typ `redouble`, správné přiřazení hodnot nezávislých proměnných, po spočtení hodnot $f(x)$ musíme přiřadit váhy $\bar{y}_i$, zavolat funkci `reverse_sweep` a nakonec přečíst vypočtené hodnoty derivací. Jinak řečeno, vlastní jádro programu, kde probíhá výpočet, je stejně jako v případě přímého módu nezměněno.

Demonstrace na příkladu I. Navážeme na výše uvedený příklad I ze strany 31. Chceme spočítat derivaci funkce

$$f(x_1, x_2) = \frac{\sin x_1}{x_1 x_2} + x_1 x_2$$

v bodě $(\pi/4, 1)$ a také gradient funkce f v tomtéž bodě.

Nyní, na základě předchozích odstavců, ukážeme, jak by mohla vypadat transformace programu pomocí zpětného módu – transformace deklarace proměnných i výpočtu. Funkce f může být vyhodnocena pomocí následující ukázky zapsané v C++:

```
//deklarace a vstup proměnných
   double x1, x2, y;
   double temp;
   x1 = pi/4;
   x2 = 1;
//výpočet
   temp = x1 * x2;
   y = sin(x1) / temp + temp;
//výstup proměnných
   cout << y;
```

Tuto část programu ztransformujeme pomocí zpětného módu, aby počítal i žádaný gradient. Tak dostaneme

```
//deklarace a vstup promenných
   redouble x1, x2, y;                                    *
   redouble temp;                                         *
   x1 = makeindepvar(pi/4);                               *
   x2 = makeindepvar(1.0);                                *
//výpočet
   temp = x1 * x2;
   y = sin(x1) / temp + temp;
//výstup proměnných - hodnota funkce f(x)
   cout << value(y);                                      *
//výpočet derivace
   setbarvalue(y,1.0);                                    *
   reverse_sweep();                                       *
//výstup proměnných - hodnota derivace: grad(f(x))
   cout << "derivace podle x1: " << barvalue(x1) << endl;  *
   cout << "derivace podle x2: " << barvalue(x2) << endl;  *
```

Veškeré změny, které jsme provedli, jsou předefinování příslušných proměnných z typu `double` na `redouble`, inicializace proměnných `x1` a `x2`, výpis hodnoty proměnné `y`, přiřazení váhy do $\bar{y}$, zavolání funkce `reverse_sweep` a výpis spočtených derivací $\bar{x}_1$ a $\bar{x}_2$. Všechny řádky, které jsou nové, nebo na kterých došlo k nějaké změně, jsme označili hvězdičkou. Jádro programu, ve kterém probíhá vlastní výpočet, zůstalo nezměněno. To je důležité, protože to bývá nejobsáhlejší a nejméně přehledná část derivovaného programu.

pořadí prováděné operace	1	2	3	4	5	6	
prováděná operace	definice nezávislé proměnné x_1	definice nezávislé proměnné x_2	$x_1 \cdot x_2$	$\sin(x1)$	$\sin(x1)/temp$	$\sin(x1)/temp+$ $+temp$	
funkce nebo operátor, odkud záznam v polích vznikl	`make-indep-var (pi/4)`	`make-indep-var (1.0)`	`x1 * x2`	`sin(x1)`	`sin(x1)/temp`	`sin(x1)/temp + + temp`	
index v poli (C++ čísluje od 0)	0	1	2	3	4	5	

pole `trace`:

– složka `v`	hodnota x_1	hodnota x_2	hodnota $x_1 \cdot x_2$	hodnota $\sin(x1)$	hodnota $\sin(x1)/temp$	hodnota $\sin(x1)/temp$ $+temp$	
– složka `vbar`	0	0	0	0	0	0	
– složka `opcode`	2	2	30	60	40	10	
– složka `arg1`	nezadáno	nezadáno	odkaz na 0.prvek v poli	odkaz na 0.prvek v poli	odkaz na 3.prvek v poli	odkaz na 4.prvek v poli	
– složka `arg2`	nezadáno	nezadáno	odkaz na 1.prvek v poli	nezadáno	odkaz na 2.prvek v poli	odkaz na 2.prvek v poli	

Obrázek 1.13: Uložení dat v polích pro Příklad I.

Na obrázku 1.13 je ukázáno, jak jsou data v poli `trace` uložena v okamžiku před zavoláním funkce `reverse_sweep`. (Funkce `reverse_sweep` by pouze naplnila složky `vbar`.)

Aby bylo možné tento program přeložit a spustit, je nutné k němu připojit balík s definicemi pomocných procedur a funkcí a deklaracemi nových typů proměnných, jak jsme se o tom již zmínili výše v tomto odstavci.

Další názorné příklady použití a efektivity automatického derivování jsou uvedeny například v [10], [11], [12] nebo [13] .

□

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`.

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.

Nezmínili jsme některé aspekty transformace programu při zpětném módu auto-

matického derivování. Některé z nich však vyplynou z následujících kapitol.

Druhé derivace

Implementaci automatického derivování pro druhé nebo vyšší derivace je možné získat kombinací uvedených technik implementace pro přímý a zpětný mód automatického derivování.

V tomto odstavci naznačíme, jak naprogramovat výpočet hodnoty $\bar{y} \cdot f''(x) \cdot \dot{x}$ pro funkci $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ a pro vektory $\dot{x} \in \mathbb{R}^n$ a $\bar{y} \in \mathbb{R}^m$. Výraz $\bar{y} \cdot f''(x) \cdot \dot{x}$ přitom chápeme stejně ve vztahu 1.14. Pokud $\dot{x}$ bude i-tý svislý vektor kartézské báze a pokud $\bar{y}$ bude j-tý vodorovný vektor kartézské báze, spočítáme tak i-tý řádek Hessovy matice j-té složky funkce f v bodě x.

K výpočtu $\bar{y} \cdot f''(x) \cdot \dot{x}$ použijeme metodu automatického derivování, kterou jsme odvodili v kapitole 1.5, tedy kombinaci zpětného a přímého módu automatického derivování.

Pro odvození implementace výpočtu druhých derivací využijeme implementace přímého a zpětného módu z předchozích odstavců. Definici třídy `doublet` ponecháme beze změny.

```
class doublet       // definice třídy doublet
{                   //  obsahující funkční hodnotu
                    //  a hodnotu směrové derivace
 public:
   double v;        // funkční hodnota v_i
   double vdot;     // hodnota derivace v_i s tečkou
};
```

Pozměníme ale definici třídy `element`, a to takto:

```
class element          // definice třídy element
{                      // obsahující informace o elementární operaci
  public:
    doublet v;         // funkční hodnota  - typ doublet!
    doublet vbar;      // hodnota derivace - typ doublet!
    int opcode;        // typ elementární operace
    element *arg1;     // odkaz na první argument
    element *arg2;     // odkaz na druhý argument
};
```

Složky `v` a `vbar` nejsou nyní typu `double`, ale `doublet`, obsahují tedy kromě funkčních hodnot také derivaci ve směru $\dot{x}$. Přesněji řečeno,

- složka `v.v` obsahuje funkční hodnotu v,
- složka `v.vdot` obsahuje hodnotu směrové derivace $\dot{v}$,
- složka `vbar.v` obsahuje hodnotu derivace $\bar{v}$,

- složka `vbar.vdot` obsahuje hodnotu druhé derivace $\dot{\bar{v}}$.

Proto je nutné 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 po dokončení výpočtu příslušné hodnoty interpretovat takto.

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

Funkce `reverse_sweep` bude díky přetížení operátorů a funkcí obsahovat například pro funkci sinus stejné příkazy jako v implementaci zpětného módu, tedy

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

Složky `v`, `vbar` a `deriv` jsou zde ovšem typu `doublet`. Operace s nimi jsou přetížené a díky tomu provádí procedura `reverse_sweep` výpočet druhé derivace, jak je popsáno v kapitole 1.5.

Pokud by složka `vdot` proměnné typu `doublet` byla vektor (pole) délky p, lze tak vyhodnotit p řádků Hessovy matice zároveň, tedy například celou Hessovu matici. Při této modifikaci bychom museli přepsat definice přetěžujících funkcí pro typ `doublet`, stejně jako inicializační, přístupové, vstupní a výstupní funkce.

Třetí a vyšší derivace

Automatické derivování je možné použít i pro výpočet třetích nebo vyšších derivací. Letmo se proto o nich zmíníme, i když se tyto vyšší derivace v praxi příliš nepoužívají.

Například, třetí derivaci lze získat aplikaci zpětného, na něj přímého a ještě jednou přímého módu na zadaný program. Podobně lze vnořovat i výše uvedené datové typy.

Například pro zmíněnou třetí derivaci můžeme definovat typ `triplet` jako

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

Použijeme-li tento typ `triplet` (anebo ještě obecnější typy) v definici typu `element`

```
class element          // definice třídy element
{                      // obsahující informace o elementární operaci
  public:
    triplet v;         // funkční hodnota  - typ triplet!
    triplet vbar;      // hodnota derivace - typ triplet!
    int opcode;        // typ elementární operace
    element *arg1;     // odkaz na první argument
    element *arg2;     // odkaz na druhý argument
};
```

pak díky zpětnému módu získáme hodnoty třetích (případně vyšších) derivací. Příslušné definice přetížených operací je přitom nutné nadefinovat podobným způsobem jako výše.

1.6.2 Použití preprocesoru

Implementaci automatického derivování je kromě techniky přetížení operátorů a funkcí možné provést i jiným způsobem, například použitím *preprocesoru* ("preprocessing", "předzpracování"). Této možnosti je výhodné použít, pokud programovací jazyk transfromovaného programu nepodporuje přetížení operátorů a funkcí, například Fortran 77. O této možnosti se zmíníme v následujících odstavcích.

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 $y = f(x)$ změněn na program, který bude navíc počítat i derivace funkce f v bodě x. Preprocesor analyzuje zdrojový program řádek po řádku, přiřazovací příkazy rozloží na jednotlivé elementární 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 f napsaný, může tedy být jakýkoliv běžný programovací jazyk.

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 (a nemusíme tedy zasahovat do původního programu).

V následujícím odstavci popíšeme způsob implementace zpětného módu automatického derivování, který používá preprocesor, nicméně je podobný implementaci pomocí přetížení, odkud jsme ale přetěžování odstranili. Tato technika je použita při implementaci automatického derivování do systému UFO, jak bude popsáno v kapitole 3. Systém UFO je naprogramován v jazyce Fortran 77, proto pro popis implementace použijeme tento jazyk. Navíc je Fortran 77 v numerické matematice velmi oblíbený a rošířený.

1.6.3 Implementace zpětného módu ve Fortranu 77

Některé programovací jazyky neumožňují pracovat s ukazateli nebo definovat nové typy proměnných. Proto popíšeme modifikaci implementace automatického derivování z předchozího odstavce 1.6.1, 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. Jak jsme již zmínili, 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 `elementů` nahradíme pěti samostatnými poli `V`, `VBAR`, `OPCODE`, `ARG1` a `ARG2` typu `REAL`, `REAL`, `INTEGER`, `INTEGER` a `INTEGER` (ve tomtéž 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 φ_k (a s ní související údaje) budeme namísto proměnné typu `redouble` používat *index polí*[5]. Proto 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 34. 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 = 10,
*           BMINUV = 20, BMULTV = 30, RECIPV = 40, ...
*           EXPV = 50, LNV = 51, SINV = 60, ...)
```

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`. Pole `V, VBAR, OPCODE, ARG1, ARG2` se záznamy o provedených základních operacích, spolu s proměnnou `INDARR`, budeme těmto funkcím předávat pomocí parametrů, případně je lze také předávat pomocí tzv. `COMMON` proměnných, jak bude ukázáno v kapitole 3.

Předefinované operace sinus a násobení mohou vypadat například tímto způsobem:

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

```
 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))   ! výpočet hodnoty elementární funkce
     VBAR(INDARR) = 0.0           ! vynulování složky vbar, kam se bude
                                  !  nasčítávat derivace
     OPCODE(INDARR) = SINV        ! uložení identifikace této elementární operace
     ARG1(INDARR) = IARG1         ! uložení argumentu této elementární operace
     SING = INDARR                ! index této elementární operace
     INDARR = INDARR + 1          ! posun na další prvek v poli
   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) ! výpočet hodnoty elementární funkce
     VBAR(INDARR) = 0.0           ! vynulování složky vbar, kam se bude
                                  !  nasčítávat derivace
     OPCODE(INDARR) = BMULTV      ! uložení identifikace této elementární operace
     ARG1(INDARR) = IARG1         ! uložení prvního argumentu této elem. operace
     ARG2(INDARR) = IARG2         ! uložení druhého argumentu této elem. operace
     BMULTG = INDARR              ! index této elementární operace
     INDARR = INDARR + 1          ! posun na další prvek v poli
   END
```

Pro binární operace nadefinujeme ještě funkce, kdy jeden z argumentů je konstanta nezávislá na nezávislých proměnných. Mohli bychom však postupovat i jinak, kdy bychom konstantu zaznamenali do polí jako konstantu a pak provedli „funkci pro dva indexy polí“, například `BMULTG`.

Všechna volání základních funkcí tedy 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 prvků 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          ! přiřazení hodnoty nezávislé proměnné
     VBAR(INDARR) = 0.0         ! vynulování složky vbar, kam se bude
                                !  nasčítávat derivace
     OPCODE(INDARR) = INDV      ! uložení identifikace této elementární operace
     MKINDP = INDARR            ! index této elementární operace
     INDARR = INDARR + 1        ! posun na další prvek v poli
```

```
    END
```

Nejpozději po dokončení vyhodnocení funkce f v bodě x je potřeba přiřadit hodnoty vah y_i na správná místa v poli `VBAR`, aby se mohla spustit subrutina (podprogram) `RVRSWP` pro zpětný průchod poli a vypočtení všech přidružených hodnot. 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           // cyklus přes všechny provedené
                                          //  elementární operace
                                          //  od poslední k první
                                          //  (polem trace odzadu dopředu)
        IF(OPCODE(I) .EQ. CONV) THEN      // podle provedené elementární operace
          .                               //  rozlišíme výpočet derivace
          .
          .
        ELSE IF(OPCODE(I) .EQ. BMULTV) THEN // elementární operace byla násobení
          VBAR(ARG1(I)) = VBAR(ARG1(I)) + VBAR(I) * V(ARG2(I))
                                          // postupné načítání hodnot derivací
          VBAR(ARG2(I)) = VBAR(ARG2(I)) + VBAR(I) * V(ARG1(I))
                                          // postupné načítání hodnot derivací
          .
          .
          .
        ELSE IF(OPCODE(I) .EQ. SINV) THEN   // elementární operace byla sinus
          DERIV = COS (V(ARG1(I)))
          VBAR(ARG1(I)) = VBAR(ARG1(I)) + VBAR(I) * DERIV  // postupné načítání
                                                           // hodnot derivací
        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 f používají nějaké podprogramy, je nutné je upravit analogickým způsobem, včetně jejich parametrů. Kdybychom požadovali vyčíslení q gradientů najednou, změníme pole `VBAR` na q-rozměrné a zřejmým způsobem upravíme související funkce, zejména `RVRSWP`.

Implementaci výpočtu druhých derivací pomocí Fortranu 77 podrobně popíšeme v kapitole 3, která se zabývá implementací automatického derivování do systému UFO.

1.7 Složitost odvozeného programu

Čas potřebný pro vyhodnocení derivací pomocí programů vygenerovaných automatickým derivováním lze shora odhadnout malým násobkem času potřebného pro výpočet hodnoty derivované funkce $f(x)$. Pro tyto odhady je potřebné uvažovat několik předpokladů, které však nejsou příliš omezující. Například je nutné předpokládat, že práce potřebná na vykonání ztransformované operace je stejnoměrně omezená násobkem práce potřebné na vykonání původní (neztransformované) operace atd.

V tabulce 1.1 jsou uvedeny odhady, které za těchto předpokladů platí pro funkci $f : \mathbb{R}^n \to \mathbb{R}^m$. Numerické hodnoty uvedené v tabulce jsou odvozené například v [5].

Chceme-li si zde přiblížit anebo alespoň zhruba odvodit hodnoty z této tabulky 1.1, uvažujme pro přímý mód například operaci $\sin(x)$. Ta se ztransformuje na $\cos(x) \cdot \dot{x}$. K jedné operaci $\sin(x)$ jsme přidali dvě nové operace, a to $\cos(x)$ a násobení, což odpovídá $\omega_F = 1 + 2$. Ve zpětném módu se $y = \sin(x)$ ztransfomuje na $\bar{x} = \bar{x} + \bar{y} \cdot \cos(x)$, které přibližně odpovídá $\omega_R = 1.5 + 2.5$.

Pokud je Jacobiho nebo Hessova matice řídká, lze odvodit efektivnější techniky výpočtu derivací a tedy výrazně lepší odhady složitosti výpočtu derivací.

První derivace – přímý mód (směrové derivace)			
$\frac{TIME(f'(x)\cdot\dot{x})}{TIME(f(x))} \leq \omega_F$	kde	pro jeden směr $\dot{x}$ je	$\omega_F = 3$
		pro p směrů $\dot{x}$ zároveň je	$\omega_F = 1 + 2p$
První derivace – zpětný mód (gradienty)			
$\frac{TIME(\bar{y}\cdot f'(x))}{TIME(f(x))} \leq \omega_R$	kde	pro jeden váhový vektor $\bar{y}$ je	$\omega_R = 4$
		pro q váhových vektorů $\bar{y}$ zároveň je	$\omega_R = 1.5 + 2.5q$
Druhé derivace – zpětný a přímý mód (směrové derivace gradientů)			
$\frac{TIME(\bar{y}\cdot f''(x)\cdot\dot{x})}{TIME(f(x))} \leq \omega_{RF}$	kde	pro jeden směr $\dot{x}$ je	$\omega_{RF} = 12$
		pro p směrů $\dot{x}$ zároveň je	$\omega_{RF} = 6 + 6p$

Tabulka 1.1: Odhady složitosti výpočtu derivací pomocí programů získaných automatickým derivováním.

Kapitola 2

Systém UFO

Jedním z úkolů této disertační práce bylo implementovat automatické derivování do systému UFO, který slouží pro optimalizaci funkcí. V této kapitole tedy systém UFO stručně popíšeme.

2.1 Stručný popis systému UFO

Část 2.1 této kapitoly, ve které uvedeme stručný popis systému UFO, vychází z technické zprávy [18], která systém UFO podrobně dokumentuje.

2.1.1 Základní vlastnosti systému UFO

Interaktivní systém UFO (Universal Functional Optimization system), který slouží pro optimalizaci funkcí, byl vyvinut v Ústavu informatiky Akademie věd České republiky. Bližší informace o tomto systému lze získat na internetové adrese `http://www.cs.cas.cz/~luksan/ufo.html`. Z této adresy je také možné stáhnout volně šiřitelnou verzi tohoto systému. Podrobný popis tohoto systému je uveden v [18].

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 metody Newtonova typu. 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).

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

3. Tento vygenerovaný program je pomocí překladače Fortranu 77 přeložen a slinkován (spojen) s knihovními podprogramy.

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 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., a to 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.

2.1.2 Příklad

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 hledání řešení ("nultá 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 (INPUT), minimalizovanou funkci (FMODELF) a specifikovali jsme tvar výstupu (NOUT).[1] 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. Celkový počet provedených iterací je NIT, počet vyhodnocení funkční hodnoty FF je NFV, počet vyhodnocení její derivace je NFG atd. Volbu metody, jakou byl náš problém vyřešen, provedl systém automaticky, protože jsme žádnou ve vstupním souboru nespecifikovali. Podrobnější popis výstupů ze systému UFO je uveden v [18]. □

2.1.3 Šablony a moduly

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 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.

2.1.4 Řídící jazyk systému UFO, makroproměnné

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'.

[1] Podrobnější popis vstupního souboru systému UFO je uveden na straně 50.

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 následující posloupnost příkazů

```
$SET(JMÉNOMAKROPROMĚNNÉ)
  hodnota_makroproměnné
$ENDSET
```

Hodnota `hodnota_makroproměnné` nemusí být pouze 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`, `$STANDARD` atd. Například uvedení instrukce `$METHOD` způsobí vygenerování optimalizační metody (vlastně složení z jednotlivých modulů) podle předem specifikovaných požadavků. Instrukce `$STANDARD` provede sestavení programu se (zhruba řečeno) deklaracemi a inicializací proměnných, vlastní výpočet sestavenou metodou a vypsáním získaných výsledků. Pro úplnost dodejme, že každá instrukce začíná znakem `$`.

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 [18].

Vrátíme-li se nyní k příkladu na straně 48, je již téměř celý jasný. Makroproměnnou `$INPUT` specifikujeme počáteční hodnoty proměnné x, makroproměnná `$FMODELF` definuje 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 tedy 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`.

2.1.5 Označení funkcí

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 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 [18]. Například, obsah makroproměnné `FGMODELF` určuje výpočet hodnoty optimalizované funkce a její první derivace. Ze jmen makroproměnných také vyplývá, jaké je označení závislých a nezávislých proměnných funkcí, které jsou těmito makroproměnnými definovány.

Jak již víme, systém UFO používá ke svým výpočtům také hodnoty derivací funkcí. Ty je možné zadat analyticky pomocí makroproměnných, které jsme popsali v předchozím odstavci (například makroproměnnou `FGMODELF`), nebo jsou jejich aproximace počítány pomocí poměrných diferencí (viz makroproměnná `$MCG`).

Jeden z úkolů této disertační práce bylo implementovat automatické derivování do systému UFO jako další způsob výpočtu derivací funkcí. Automatickou transformaci výpočtu hodnot funkce na výpočet hodnoty funkce a její derivace bude provádět BEL preprocesor. V další kapitole tuto implementaci detailně popíšeme.

[2] Jedná se například o minimalizovanou (maximalizovanou) funkci, funkce určující vazbové podmínky atd.

Kapitola 3

Automatické derivování v systému UFO

Jedním z úkolů této disertační práce byla implementace automatického derivování do systému UFO jako další způsob výpočtu derivací funkcí. V následujících odstavcích popíšeme detaily této implementace.

3.1 Vyvolání automatického derivování v systému UFO

Pomocí automatického derivování je možné v systému UFO počítat hodnoty prvních nebo druhých derivací funkcí, které jsou zadány makroproměnnými `FMODELF`, `FMODELA` nebo `FMODELC`.

Zadáním jedné z následujících hodnot do proměnné `$IADF` se určuje, zda se má použít automatické derivování pro výpočet hodnoty derivace funkce `FF` v proměnné `FMODELF`:

- `$IADF`=0 (defaultní hodnota)
  Derivace funkce `FF` zadané v `FMODELF` se nepočítají pomocí automatického derivování.

- `$IADF`=1
  První derivace funkce `FF` zadané proměnnou `FMODELF` se počítají pomocí zpětného módu automatického derivování. Je vytvořena proměnná `FGMODELF` s výpočtem hodnoty funkce `FF` a jejího gradientu `GF`. Proměnná `FMODELF` je poté zrušena.

- `$IADF`=2
  Druhé derivace funkce `FF` zadané proměnnou `FMODELF` se počítají pomocí zpětného a přímého módu automatického derivování. Jsou vytvořeny proměnné `FGMODELF` (s výpočtem hodnoty funkce `FF` a jejího gradientu `GF` – pomocí zpětného módu) a proměnná `HMODELF` (s výpočtem hodnoty funkce `FF`, jejího gradientu `GF` a její Hessovy matice `HF` – pomocí zpětného módu a následnou aplikací přímého módu). Pak je proměnná `FMODELF` zrušena.

Vyvolání automatického derivování pro funkce definované proměnnými FMODELA, resp. FMODELC se provádí analogicky jako pro proměnnou FMODELF, liší se však jméno proměnné určující požadovanou transformaci: místo proměnné $IADF se používá $IADA, resp. $IADC.

V rámci jednoho vstupního souboru je možné nezávisle kombinovat hodnoty proměnných $IADF, $IADA a $IADC, například $IADA=2, $IADC=1 atp.

3.2 Omezení při použití automatického derivování v systému UFO

Jak již víme z předchozích kapitol, transformace proměnné FMODELF, FMODELA nebo FMODELC pomocí automatického derivování je transformace programu, nebo jeho části. Aby implementace automatického derivování v systému UFO nebyla příliš komplikovaná, položili jsme jistá omezení na obsah těchto proměnných FMODELF, FMODELA a FMODELC, tj. na výpočet hodnot FF, FA a FC:

- ve výpočtu je zakázáno volání subroutin a funkcí, kromě standardních funkcí Fortranu. (Toto omezení neplatí, máme-li k dispozici navíc i subroutinu nebo funkci s vypočtem derivace podle konvencí popsaných v dalším odstavci. V tom případě je možné uživatelsky definované subroutiny nebo funkce ve výpočtu použít.)
- pokud se ve výpočtu používá pole (definované například REAL*8 W(100)) a jeho hodnoty jsou závislé na nezávislých proměnných X(), pak je třeba "ručně" deklarovat pole INTEGER IAD_W(100), tj. pole, jehož jméno je složením řetězce IAD_ a jména původního pole. Toto nově definované pole je typu INTEGER a má stejnou velikost jako původní pole. Toto nové pole bude obsahovat indexy do pole, kde jsou zaznamenávány všechny prováděné operace.
- ve jménu proměnné nemůže být obsažena číslice
- v příkazu IF a ELSEIF:
  - lze porovnávat pouze čísla a proměnné, žádné výrazy nelze v podmínkách použít,
  - není možné používat mezery (například, není možné použít podmínku IF (A .LT. B) C=D+1 , lze však použít podmínku IF(A.LT.B)C=D+1 ,
  - indexy polí a argumenty funkcí se netransformují, jsou pouze překopírovány. Toto omezení se týká pouze podmínky, nikoliv příkazu prováděného, když je podmínka splněna.
- v příkazu přiřazení:
  - na rozdíl od příkazu IF a ELSEIF, mezery v příkazu přiřazení mohou být obsaženy,

– "iterační přiřazení" (například `F=F*X(I)`) s proměnnými, které jsou závislé na nezávislých proměnných `X()`, je nutné přeformulovat za použití cyklu a nově nadefinovaného pole, například `W(I+1)=W(I)*X(I)`.

- nelze použít tzv. "computed goto statement", tj. příkaz
  `GO TO(`*label1, label2, ... labelN*`)` *INTEGER-EXPRESSION*

Jak vidíme, tyto předpoklady na zápis výpočtu funkcí `FF`, `FA` a `FC` v makroproměnných `FMODELF`, `FMODELA` a `FMODELC` nejsou příliš omezující.

3.3 Základní principy implementace automatického derivování v systému UFO

V tomto a následujícím odstavci popíšeme základní principy implementace automatického derivování pro proměnnou `FMODELF`. Pro proměnné `FMODELA` a `FMODELC` je postup implementace úplně analogický.

Všechny popsané transformace automatickým derivováním probíhají na úplném začátku zpracování vstupního souboru `*.UFO` systémem UFO.

Jak již víme z odstavce 3.1, při zadání hodnoty 1 nebo 2 do proměnné `$IADF` dojde k vytvoření proměnné `FGMODELF` (a pro `$IADF`=2 také k vytvoření proměnné `HMODELF`) a smazání proměnné `FMODELF`.

Do proměnné `FGMODELF` se přitom uloží postup výpočtu hodnoty funkce `FF` a jejího gradientu `GF` pomocí zpětného módu automatického derivování. Používáme zpětný mód, protože v proměnné `FMODELF` se počítá skalární hodnota `FF` a pro výpočet jejího gradientu je zpětný mód výhodnější.

Proměnnou `FGMODELF` získáme z proměnné `FMODELF` následujícím způsobem. Stručně řečeno, všechny výpočty a přiřazení ztransformujeme, aby se kromě provedení tohoto výpočtu a přiřazení zaznamenala provedená operace společně s argumenty do polí, kterými ve druhé části výpočtu projdeme odzadu a vykonáme patřičné operace pro výpočet derivace. Přednastavené délky těchto polí jsou `$NADARR=1000`. Je-li potřeba tato hodnota zvětšit, vložíme do vstupního souboru řádek, na kterém přiřadíme potřebnou hodnotu, například `$NADARR=2000`.

Pro proměnnou `HMODELF` platí výše uvedená fakta analogicky jako pro proměnnou `FGMODELF`, ale výpočet derivace probíhá pomocí aplikace nejprve zpětného módu a následně přímého módu.

Konkrétní ukázky příslušných definic proměnných a subrutin a další podrobnosti implementace uvedeme v následujícím odstavci 3.4.

3.4 Technické detaily implementace automatického derivování v systému UFO

V tomto odstavci uvedeme technické podrobnosti o implementaci automatického derivování v systému UFO. Více detailů je možné najít ve výzkumné zprávě [9].

3.4.1 První derivace

Jak jsme již naznačili, při výpočtu hodnot derivace pomocí automatického derivování (`$IADF`=1 nebo 2) jsou všechny nezávislé proměnné a prováděné operace (jejich typ, argumenty a numerické hodnoty výsledků) postupně zaznamenávány do polí o velikosti `$NADARR`. Pro `$IADF`=1 nebo 2 se jedná o pole:

- `REAL*8 V($NADARR)` – je pole, ve kterém jsou uloženy hodnoty $v_i = \varphi_i(v_j)_{j \prec i}$, viz vztah (1.1),
- `REAL*8 VBAR($NADARR)` – je pole, ve kterém jsou uloženy hodnoty $\bar{v}_i = \sum_{j \succ i} \bar{v}_j \cdot \frac{\partial \varphi_j}{\partial v_i}$, viz vztah (1.4),
- `INTEGER OPCODE($NADARR)` – je pole, ve kterém je uložena identifikace operace prováděné v i-tém kroku (například, sčítání odpovídá 10, násobení odpovídá 30, sinu odpovídá 60 atp.),
- `INTEGER ARG1($NADARR)` – je pole odkazů do těchto polí pro první argument právě prováděné operace,
- `INTEGER ARG2($NADARR)` – je pole odkazů do těchto polí pro druhý argument právě prováděné operace.

Při výpočtu druhých derivací (`$IADF` = 2) se používají dvě další pole, která uvedeme v následujícím odstavci.

Každá elementární funkce $\varphi_i(v_j)_{j \prec i}$ (například násobení, sinus atd.) v postupu výpočtu hodnoty $y = f(x)$ je v průběhu transformace nahrazena subrutinou, která kromě původní operace $\varphi_i(v_j)_{j \prec i}$ provede i zaznamenání svého zavolání do polí `V`, `VDOT`, `OPCODE`, `ARG1`, `ARG2`, případně také `VDOT`, `VBARDT`. Například, elementární funkce násobení nebo sinus jsou nahrazeny subrutinami `BMULTG`, resp. `SING`:

```
!--- ztransformovaná operace násobení ---
INTEGER FUNCTION BMULTG(IARG1, IARG2)   !vstupní parametry: pořadí (index)
                                        !numerických hodnot v polích, které
                                        !máme vynásobit
                                        !výstupní hodnota: pořadí (index)
                                        !v polích pro právě tuto operaci
  COMMON /AD_F1/ V, VBAR, OPCODE, ARG1, ARG2, INDARR
    REAL*8 V($NADARR), VBAR($NADARR)
    INTEGER OPCODE($NADARR), ARG1($NADARR), ARG2($NADARR)
    INTEGER INDARR
  INTEGER IARG1, IARG2

  V(INDARR)=V(IARG1)*V(IARG2)           !spočtení elementární operace násobení
  VBAR(INDARR)=0.0D0                    !vynulování proměnné v_i s pruhem
                                        !(bude se do ní postupně přičítat)
  OPCODE(INDARR)=30                     !zaznamenání kódu této operace (30=násobení)
  ARG1(INDARR)=IARG1                    !zaznamenání pořadí (indexu) v polích
                                        !pro první (levý) argument
  ARG2(INDARR)=IARG2                    !zaznamenání pořadí (indexu) v polích
                                        !pro druhý (pravý) argument
```

```
  BMULTG=INDARR                          !pořadí této operace
  INDARR=INDARR+1                        !příprava na další elementární operaci -
                                         !- posunutí indexu ("ukazovátka") do polí,
                                         !kam až jsou pole vyplněny
END


!--- ztransformovaná operace sinus ---
INTEGER FUNCTION SING(IARG1)
  COMMON /AD_F1/ V, VBAR, OPCODE, ARG1, ARG2, INDARR
    REAL*8 V($NADARR), VBAR($NADARR)
    INTEGER OPCODE($NADARR), ARG1($NADARR), ARG2($NADARR)
    INTEGER INDARR
  INTEGER IARG1

  V(INDARR)=SIN(V(IARG1))
  VBAR(INDARR)=0.0D0
  OPCODE(INDARR)=60                      !zaznamenání kódu této operace (60=sinus)
  ARG1(INDARR)=IARG1                     !sinus má pouze jeden argument
  SING=INDARR
  INDARR=INDARR+1
END
```

V průběhu ztransformovaného výpočtu se na všechny nezávislé proměnné a na proměnné, které na nich závisí, a také na konstanty, odkazujeme pomocí pořadového čísla prováděné operace, ve které vznikly. Toto pořadí odpovídá indexu v polích, kde jsou informace o této operaci uloženy. Jména proměnných jsou pro tento účel transformována – jsou složením řetězce `IAD_` a jména příslušné proměnné, například `IAD_X(*)` pro nezávislou proměnnou `X(*)` nebo `IAD_PROM` pro proměnnou `PROM`. Na straně 58 uvedeme názorný příklad s vysvětlením, jak se operace a hodnoty do polí ukládají.

Poslední fáze výpočtu hodnoty derivace je zavolání subrutiny `RVRSWP`, která projde poly `V`, `VDOT`, `OPCODE`, `ARG1`, `ARG2` odzadu dopředu a provede tak vlastní výpočet hodnot (1.4), tj.

$$\bar{v}_i = \sum_{j \succ i} \bar{v}_j \cdot \frac{\partial \varphi_j}{\partial v_i}$$

neboli

$$\bar{v}_i = \bar{v}_i + \bar{v}_j \cdot \frac{\partial \varphi_j}{\partial v_i}(v_k)_{k \prec j} \quad \text{pro} \;\; i \prec j \qquad j = l, \ldots, 1.$$

Procedura `RVRSWP`:

```
SUBROUTINE RVRSWP()
  COMMON /AD_F1/ V, VBAR, OPCODE, ARG1, ARG2, INDARR
    REAL*8 V($NADARR), VBAR($NADARR)
    INTEGER OPCODE($NADARR), ARG1($NADARR), ARG2($NADARR)
    INTEGER INDARR
  REAL*8 DERIV
  INTEGER I

  DO 999, I=INDARR-1, 1, -1              !cyklus přes jednotlivé operace pozpátku,
```

```
                                        !tj. průchod polemi odzadu dopředu
    IF(OPCODE(I).EQ.30) THEN            !operace násobení
      VBAR(ARG1(I))=VBAR(ARG1(I))+VBAR(I)*V(ARG2(I))   !postupné přičítání hodnot
                                                       !do proměnné v_. s pruhem
      VBAR(ARG2(I))=VBAR(ARG2(I))+VBAR(I)*V(ARG1(I))   !postupné přičítání hodnot
                                                       !do proměnné v_. s pruhem
   .
   .
    ELSEIF(OPCODE(I).EQ.60) THEN        !operace sinus
      DERIV=COS(V(ARG1(I)))                            !pomocná proměnná
      VBAR(ARG1(I))=VBAR(ARG1(I))+VBAR(I)*DERIV        !postupné přičítání hodnot
                                                       !do proměnné v_. s pruhem
   .
   .
    ELSEIF(OPCODE(I).EQ.2) THEN         !operace nezávislá proměnná
      CONTINUE
   .
   .
    ENDIF
999 CONTINUE
    END
```

Po provedení této subrutiny jsou již spočteny hodnoty derivací v příslušných prvcích pole `VBAR`, odkud je přiřadíme do proměnných `GF(*)`, jak se předpokládá pro výstup z proměnné `FGMODELF` – například pro $\mathtt{X=(X_1,X_2)}$:

```
   DO 86 IADCOUNT=1,2
   GF(IADCOUNT)=VBAR(IAD_X(IADCOUNT))
86 CONTINUE
```

Pro úplnost bychom měli dodat, že na začátku proměnné `FGMODELF` je zavolán cyklus pro správnou definici a označení nezávislých proměnných `X(*)` – například pro $\mathtt{X=(X_1,X_2)}$:

```
   DO 85 IADCOUNT=1,2
   IAD_X(IADCOUNT)=MKINDP(X(IADCOUNT))
85 CONTINUE
```

kde `MKINDP(X(.))` je subrutina, která uloží hodnotu proměnné `X(.)` do polí a označí ji jako nezávislou proměnnou:

```
! --- operace definování nezávislé proměnné ---
INTEGER FUNCTION MKINDP(CISLO)
  COMMON /AD_F1/ V, VBAR, OPCODE, ARG1, ARG2, INDARR
    REAL*8 V($NADARR), VBAR($NADARR)
    INTEGER OPCODE($NADARR), ARG1($NADARR), ARG2($NADARR)
    INTEGER INDARR
  REAL*8 CISLO

  V(INDARR)=CISLO
  VBAR(INDARR)=0.0D0
  OPCODE(INDARR)=2
  MKINDP=INDARR
  INDARR=INDARR+1
END
```

Protože víme, jaké jsou závislé a nezávislé proměnné pro funkce zadané makroproměnnými FMODELF, FMODELA a FMODELC (například pro FMODELF jsou nezávislé X a závislé FF), je označení těchto proměnných jako nezávislé (a závislé) pomocí procedur MKINDP snadné.

Konstanty, které se ve výpočtu používají, jsou také uloženy do polí pomocí subroutiny MKCNST(hodnota_konstanty)

```
! --- operace definování konstanty ---
INTEGER FUNCTION MKCNST(CISLO)
  COMMON /AD_F1/ V, VBAR, OPCODE, ARG1, ARG2, INDARR
    REAL*8 V($NADARR), VBAR($NADARR)
    INTEGER OPCODE($NADARR), ARG1($NADARR), ARG2($NADARR)
    INTEGER INDARR
  REAL*8 CISLO

  V(INDARR)=CISLO
  VBAR(INDARR)=0.0D0
  OPCODE(INDARR)=1
  MKCNST=INDARR
  INDARR=INDARR+1
END
```

Příklad

Na jednoduchém příkladu ukážeme, jak jsou data v polích V, VBAR, OPCODE, ARG1 a ARG2 uložena.

Mějme funkci $f(x_1, x_2) = x_1 \cdot x_2 + 1$. Ve zdrojovém programu, na který budeme aplikovat automatické derivování, se počítá její hodnota výrazem

```
X(1) * X(2) + 1.
```

Tento výraz se transformuje automatickým derivováním na výraz

```
BPLUSG(BMULTG(IAD_X(1),IAD_X(2)),MKCNST(DBLE(1))),
```

kde IAD_X(1) a IAD_X(2) jsou odkazy (indexy) do polí, kde jsou uloženy nezávislé proměnné X(1) a X(2) (pomocí procedur MKCNST(X(1)) a MKCNST(X(2))).

V tabulce 3.1 je ukázáno, jak jsou data v polích V, VBAR, OPCODE, ARG1 a ARG2 těsně před zavoláním procedury RVRSWP uložena. □

3.4.2 Druhé derivace

Při výpočtu druhých derivací ($IADF=2) se kromě pěti polí se zaznamenanými elementárními operacemi (kapitola 3.4.1) používají dvě pole navíc:

- REAL*8 VDOT($NADARR) – je pole, ve kterém jsou uloženy hodnoty $\dot{v}_i = \sum_{j \prec i} \frac{\partial \varphi_i}{\partial v_j}(v_k)_{k \prec i} \cdot \dot{v}_j$, viz vztah (1.3),
- REAL*8 VBARDT($NADARR) – je pole, ve kterém jsou uloženy hodnoty $\dot{\bar{v}}_i$, viz vztah 1.15.

pořadí prováděné operace	1	2	3	4	5	...
prováděná operace	definice nezávislé proměnné $X(1)$	definice nezávislé proměnné $X(2)$	$x_1 \cdot x_2$	definice konstanty 1	$x_1 \cdot x_2 + 1$	... ...
subrutina, odkud záznam operace v polích vznikl	MKINDP (X(1))	MKINDP (X(2))	BMULTG (.,.)	MKCNST (DBLE(1))	BPLUSG (.,.)	... ...
index v polích	1	2	3	4	5	...
pole `V`	hodnota x_1	hodnota x_2	hodnota $x_1 \cdot x_2$	hodnota 1	hodnota $x_1 \cdot x_2 + 1$	... ...
pole `VBAR`	0	0	0	0	0	...
pole `OPCODE`	2	2	30	1	10	...
pole `ARG1`	0	0	1	0	3	...
pole `ARG2`	0	0	2	0	4	...

Obrázek 3.1: Uložení dat v polích pro příklad implementace automatického derivování.

Ve výpočtu druhých derivací, kdy se aplikuje nejprve zpětný mód a pak přímý mód, jsou výše uvedené podprogramy složitější. Odvodíme je například tak, že na podprogramy z odstavce 3.4.1 aplikujeme přímý mód automatického derivování.

```
!--- ztransformovaná operace násobení (výpočet druhých derivací) ---
INTEGER FUNCTION BMULTH(IARG1, IARG2)              !vstupní parametry: pořadí (index)
                                                   !numerických hodnot v polích, které
                                                   !máme vynásobit
                                                   !výstupní hodnota: pořadí (index)
                                                   !v polích pro právě tuto operaci
COMMON /AD_F2/ V, VBAR, VDOT, VBARDT, OPCODE, ARG1, ARG2, INDARR
 REAL*8 V($NADARR), VBAR($NADARR), VDOT($NADARR),VBARDT($NADARR)
  INTEGER OPCODE($NADARR), ARG1($NADARR), ARG2($NADARR)
  INTEGER INDARR
  INTEGER IARG1, IARG2

  V(INDARR)=V(IARG1)*V(IARG2)          !spočtení elementární operace násobení
  VDOT(INDARR)=V(IARG1)*VDOT(IARG2)+VDOT(IARG1)*V(IARG2)  !aplikace přímého módu
                                                          !na operaci násobení
  VBAR(INDARR)=0.0D0                   !vynulování proměnné v_i s pruhem
                                       !(bude se do ní postupně přičítat)
  VBARDT(INDARR)=0.0D0                 !vynulování proměnné v_i s pruhem a tečkou
                                       !(bude se do ní postupně přičítat)
  OPCODE(INDARR)=30                    !zaznamenání kódu této operace (30=násobení)
  ARG1(INDARR)=IARG1                   !zaznamenání pořadí (indexu) v polích
                                       !pro první (levý) argument
  ARG2(INDARR)=IARG2                   !zaznamenání pořadí (indexu) v polích
                                       !pro druhý (pravý) argument
  BMULTH=INDARR                        !pořadí této operace
  INDARR=INDARR+1                      !příprava na další elementární operaci -
```

```
                                        !- posunutí indexu ("ukazovátka") do polí,
                                        !kam až jsou pole vyplněny
END


!--- ztransformovaná operace sinus (výpočet druhých derivací) ---
INTEGER FUNCTION SING2(IARG1)
COMMON /AD_F2/ V, VBAR, VDOT, VBARDT, OPCODE, ARG1, ARG2, INDARR
 REAL*8 V($NADARR), VBAR($NADARR), VDOT($NADARR),VBARDT($NADARR)
  INTEGER OPCODE($NADARR), ARG1($NADARR), ARG2($NADARR)
  INTEGER INDARR
  INTEGER IARG1

  V(INDARR)=SIN(V(IARG1))
  VDOT(INDARR)=COS(V(IARG1))*VDOT(IARG1)
  VBAR(INDARR)=0.0D0
  VBARDT(INDARR)=0.0D0
  OPCODE(INDARR)=60                     !zaznamenání kódu této operace (60=sinus)
  ARG1(INDARR)=IARG1                    !sinus má pouze jeden argument
  SING2=INDARR
  INDARR=INDARR+1
END
```

Subrutina, která prochází poli odzadu dopředu a počítá vlastní derivace:

```
 SUBROUTINE RVRSWPH()
 COMMON /AD_F2/ V, VBAR, VDOT, VBARDT, OPCODE, ARG1, ARG2, INDARR
  REAL*8 V($NADARR), VBAR($NADARR), VDOT($NADARR),VBARDT($NADARR)
   INTEGER OPCODE($NADARR), ARG1($NADARR), ARG2($NADARR)
   INTEGER INDARR
   REAL*8 DERIV
   INTEGER I

   DO 998, I=INDARR-1, 1, -1            !cyklus přes jednotlivé operace pozpátku,
                                        !tj. průchod polemi odzadu dopředu
     IF(OPCODE(I).EQ.30) THEN           !operace násobení
       VBAR(ARG1(I))=VBAR(ARG1(I))+VBAR(I)*V(ARG2(I))  !postupné přičítání hodnot
                                                       !do proměnné v_. s pruhem
       VBARDT(ARG1(I))=VBARDT(ARG1(I))+                !postupné přičítání hodnot
&                      VBARDT(I)*V(ARG2(I))+           !do proměnné v_. s pruhem a tečkou
&                      VBAR(I)*VDOT(ARG2(I))
       VBAR(ARG2(I))=VBAR(ARG2(I))+VBAR(I)*V(ARG1(I))  !postupné přičítání hodnot
                                                       !do proměnné v_. s pruhem
       VBARDT(ARG2(I))=VBARDT(ARG2(I))+                !postupné přičítání hodnot
&                      VBARDT(I)*V(ARG1(I))+           !do proměnné v_. s pruhem a tečkou
&                      VBAR(I)*VDOT(ARG1(I))
    .
    .
     ELSEIF(OPCODE(I).EQ.60) THEN       !operace sinus
       DERIV=COS(V(ARG1(I)))                           !pomocná hodnota
       VBAR(ARG1(I))=VBAR(ARG1(I))+VBAR(I)*DERIV       !postupné přičítání hodnot
                                                       !do proměnné v_. s pruhem
       VBARDT(ARG1(I))=VBARDT(ARG1(I))+                !postupné přičítání hodnot
&                      VBARDT(I)*DERIV-                !do proměnné v_. s pruhem a tečkou
&                      VBAR(I)*SIN(V(ARG1(I)))*VDOT(ARG1(I))
```

```
    .
    .
     ELSEIF(OPCODE(I).EQ.2) THEN        !operace nezávislá proměnná
       CONTINUE
    .
    .
     ENDIF
998 CONTINUE
    END
```

Protože při výpočtu druhých derivací pro proměnnou HMODELF chceme spočíst celou Hessovu matici, musí být uvnitř proměnné HMODELF cyklus, ve kterém se počítají jednotlivé řádky Hessovy matice. Například pro X=(X_1,X_2):

```
   DO 97 IADN=1,2
   ...
   výpočet IADN-tého řádku Hessovy matice
   ...
97 CONTINUE
```

Dále, cyklus pro přiřazení hodnot spočtených derivací do proměnných HF – například pro X=(X_1,X_2):

```
   DO 99 IADCOUNT=1,IADN
     HF(IADN1+IADCOUNT)=VBARDT(IAD_X(IADCOUNT))   ! spočtená hodnota derivace
99 CONTINUE
   IADN1=IADN1+IADN
```

Cyklus pro přiřazení hodnot nezávislých proměnných do polí na začátku výpočtu proměnné HMODELF, kde se navíc určuje, podle kterých proměnných se bude derivovat v přímém módu:

```
   DO 98 IADCOUNT=1,2
     IF(IADCOUNT.EQ.IADN) THEN
       IAD_X(IADCOUNT)=MKINDPH(X(IADCOUNT),1.0D0) !definice nezávislé proměnné
                                                  ! a směrové derivace
                         ELSE
       IAD_X(IADCOUNT)=MKINDPH(X(IADCOUNT),0.0D0) !definice nezávislé proměnné
                                                  ! a směrové derivace
     ENDIF
98 CONTINUE
```

Definice nezávislé proměnné:

```
! --- operace definování nezávislé proměnné (výpočet druhých derivací) ---
INTEGER FUNCTION MKINDPH(CISLO1,CISLO2)
COMMON /AD_F2/ V, VBAR, VDOT, VBARDT, OPCODE, ARG1, ARG2, INDARR
 REAL*8 V($NADARR), VBAR($NADARR), VDOT($NADARR),VBARDT($NADARR)
  INTEGER OPCODE($NADARR), ARG1($NADARR), ARG2($NADARR)
  INTEGER INDARR
  REAL*8 CISLO1, CISLO2

  V(INDARR)=CISLO1               !definice nezávislé proměnné
```

```
  VBAR(INDARR)=0.0D0
  VDOT(INDARR)=CISLO2              !směrový vektor
  VBARDT(INDARR)=0.0D0
  OPCODE(INDARR)=2
  MKINDPH=INDARR
  INDARR=INDARR+1
END
```

Definice konstanty:

```
! --- operace definování konstanty (výpočet druhých derivací) ---
INTEGER FUNCTION MKCNSTH(CISLO)
COMMON /AD_F2/ V, VBAR, VDOT, VBARDT, OPCODE, ARG1, ARG2, INDARR
 REAL*8 V($NADARR), VBAR($NADARR), VDOT($NADARR),VBARDT($NADARR)
  INTEGER OPCODE($NADARR), ARG1($NADARR), ARG2($NADARR)
  INTEGER INDARR
  REAL*8 CISLO

  V(INDARR)=CISLO                  !definice konstanty
  VBAR(INDARR)=0.0D0
  VDOT(INDARR)=0.0D0
  VBARDT(INDARR)=0.0D0
  OPCODE(INDARR)=1
  MKCNSTH=INDARR
  INDARR=INDARR+1
END
```

Shrněme nyní pro přehlednost ještě jednou všechny změny, které při transformaci automatickým derivováním nastávaji:

- definice procedur `BPLUSG, BSING, RVRSWP` atd., sloužící pro výpočet a zaznamenání elementárních operací atd. a jejich slinkování s původním programem,
- transformace výpočtu (tak, aby se kromě spočtení elementárních operací také zaznamenalo jejich pořadí, parametry a výsledné hodnoty) a s tím související přejmenování proměnných (nezávislých proměnných a všech proměnných, které na nich závisí),
- zavolání procedury `RVRSWP` pro `$IADF`=1 (resp. `RVRSWPH` pro `$IADF`=2), která projde seznamem prováděných operací „pozpátku“ a provede vlastní výpočet derivací.

3.5 Modifikace šablon systému UFO pro automatické derivování

Systém UFO je založen na textovém preprocesoru BEL, který byl vyvinut jako součást systému UFO. Zajišťuje vytváření výsledného programu, který řeší zadaný optimalizační problém. Preprocesor BEL zpracovává šablony a podle specifikovaných požadavků na optimalizační problém provádí dosazování jednotlivých částí programu. Při implementaci automatického derivování do systému UFO bylo nutné

změnit některé šablony i samotný textový preprocesor BEL. Zde popíšeme několik základních myšlenek, detaily jsou popsané ve výzkumné zprávě [9].

Šablona `UZDECL.I` slouží k deklaraci proměnných, používaných ve výsledném programu `P.UFO`. Proto jsme na její konec přidali část kódu, kterým se, pomocí šablon `UZADD1.I`, resp. `UZADD.I`, deklarují proměnné používané při automatickém derivování, popsané v odstavci 3.4.1 na straně 55, respektive v odstavci 3.4.2 na straně 58.

Šablona `UZINIT.I` se používá pro inicializaci numerických metod. V případě automatického derivování zde, anebo ve vnořených šablonách, probíhají následující kroky:

- Přiřazení `$NADARR=1000`, pokud hodnota `$NADARR` není v souboru `P.UFO` definovaná jinak. Toto přiřazení probíhá v šablonách `UZADS1`, resp. `UZADS2.I`.
- Deklarace procedur, které se používají při volání automatického derivování. Jedná se o procedury z odstavce 3.4.1, resp. z odstavce 3.4.2. Tyto procedury jsou vloženy do makroproměnné `SUBROUTINES` a do výsledného programu jsou vloženy pomocí šablon `UZADS1.I`, resp. `UZADS2.I`.
- Vlastní transformace makroproměnné `FMODELF` (resp. `FMODELA` nebo `FMODELC`) na makroproměnnou `FGMODELF` (resp. `FGMODELA` nebo `FGMODELC`) a případně také `HMODELF` (resp. `HMODELA` nebo `HMODELC`), aby tyto upravené makroproměnné počítaly derivace automatickým derivováním. Tato transformace je zajištěna šablonami `UZADF1.I`, resp. `UZADF2.I`.
- Vymazání makroproměnné `FMODELF` (resp. `FMODELA` nebo `FMODELC`), provedené ze šablony `UZADF1.I`, resp. `UZADF2.I`.

3.6 Příklad

Použití automatického derivování v systému UFO a jeho výhody budeme demonstrovat na následujícím příkladu. Nechť $x \in \mathbb{R}^N$ a nechť funkce $F : \mathbb{R}^N \to \mathbb{R}$ je definována jako

$$F(x) = \sum_{i=1}^{N} \left(N + i - P_i\right)^2, \tag{3.1}$$

kde

$$P_i = \sum_{j=1}^{N} \left(5\left(1 + mod(i,5) + mod(j,5)\right)\sin(x_j) + \frac{i+j}{10}\cos(x_j) \right),$$

kde $mod(a,b)$ je zbytek po vydělení "a děleno b". Hledáme lokální minimum funkce F s počáteční iterací

$$x_0 = \left(1, \frac{1}{2}, \ldots, \frac{1}{N}\right).$$

Vstupní soubor pro systém UFO `P.UFO` je na obrázku 3.2.

```
INTEGER IAD_W($$NF+1)                                    !01
REAL*8 W($$NF+1)                                         !02
$SET(INPUT)                                              !03
      DO 80 I=1, $$NF                                    !04
      X(I)=1.0D0/I                                       !05
   80 CONTINUE                                           !06
$ENDSET                                                  !07
$SET(FMODELA)                                            !08
      W(1)=0.0D0                                         !09
      DO 81 I=1, $$NF                                    !10
        A=5.0D 0*(1.0D 0+MOD(I,5)+MOD(KA,5))             !11
        B=DBLE(I+KA)/1.0D1                               !12
        W(I+1)=W(I)+A*SIN(X(I))+B*COS(X(I))              !13
   81 CONTINUE                                           !14
      FA=(DBLE($$NF+KA)-W($$NF+1))**2                    !15
$ENDSET                                                  !16
$MODEL='AF'                                              !17
$NF=50                                                   !18
$NA=50                                                   !19
$NOUT=1                                                  !20
$IADA=1                                                  !21
$BATCH                                                   !22
$STANDARD                                                !23
```

Obrázek 3.2: Vstupní soubor `P.UFO` pro příklad – automatické derivování v systému UFO.

Při výpočtu chceme použít automatické derivování pro funkce definované makroproměnnými `FMODELA`, proto přiřadíme `$IADA=1` (řádek 21). Makroproměnná `$NF` (řádek 18) určuje počet nezávislých proměnných `X(*)`, makroproměnné `MODEL` (řádek 17) a `$NA` (řádek 19) specifikují typ optimalizované úlohy (viz [18]). Výpočet hodnoty

$$(N + i - P_i)^2 \tag{3.2}$$

(viz vztah 3.1) se definuje proměnnou `FA` v makroproměnné `FMODELA` (řádky 8 až 16). Hodnoty počáteční iterace x_0 se zadávají makroproměnnou `INPUT` (řádky 3 až 7).

Protože makroproměnná `$IADA` je nastavena na hodnotu 1 (řádek 21), provede se na začátku zpracování vstupního souboru `P.UFO` vymazání makroproměnné `FMODELA` a vytvoření makroproměnné `FGMODELA`, která je zobrazena na obrázku 3.3. Zde jsou na řádcích 2 až 4 označeny proměnné `X(.)` jako nezávislé proměnné. Na řádcích 9 až 10 je ztransformovaný příkaz přiřazení `W(I+1)=W(I)+A*SIN(X(I))+B*COS(X(I))` z řádku 13 (na obrázku 3.2) a na řádcích 12 a 13 (na obrázku 3.3) je ztransformovaný řádek 15 (na obrázku 3.2), totiž `FA=(DBLE($$NF+KA)-W($$NF+1))**2`. Z těchto ztransformovaných řádků jsou volány subroutiny, které provádí nejen původní elementární aritmetické operace, ale také zaznamenání těchto operací do polí (viz kapitola 1.6). Parametry těchto subroutin jsou odkazy do polí (indexy polí) – jméno proměnné je vždy spojení řetězce `IAD_` a jména původní proměnné. [1]

[1] Tyto proměnné – odkazy do polí není třeba zvlášť deklarovat, s výjimkou případu, kdy původní proměnná je pole, jak je popsáno v kapitole 3.2. Proto na řádku 1 na obrázku 3.2 deklarujeme pole

```
   INDARR=1                                                          !01
   DO 85 IADCOUNT=1,50                                               !02
   IAD_X(IADCOUNT)=MKINDP(X(IADCOUNT))                               !03
85 CONTINUE                                                          !04
   IAD_W(1)=MKCNST(DBLE(0.0D0))                                      !05
   DO 81 I=1, 50                                                     !06
   A=5.0D 0*(1.0D 0+MOD(I,5)+MOD(KA,5))                              !07
   B=DBLE(I+KA)/1.0D1                                                !08
   IAD_W(I+1)=BPLUSG(BPLUSG(IAD_W(I),BMULTG(MKCNST(DBLE(A)),SING(IAD_ !09
  &    X(I)))),BMULTG(MKCNST(DBLE(B)),COSG(IAD_X(I))))               !10
81 CONTINUE                                                          !11
   IAD_FA=BEXPG(BMINUG(MKCNST(DBLE(DBLE(50+KA))),IAD_W(50+1)),MKCNST( !12
  &    DBLE(2)))                                                     !13
   FA=V(IAD_FA)                                                      !14
   VBAR(IAD_FA)=1.0D0                                                !15
   CALL RVRSWP()                                                     !16
   DO 86 IADCOUNT=1,50                                               !17
   GA(IADCOUNT)=VBAR(IAD_X(IADCOUNT))                                !18
86 CONTINUE                                                          !19
```

Obrázek 3.3: Hodnota proměnné `FGMODELF` pro příklad – automatické derivování v systému UFO

Na řádku 14 je přiřazena spočtená hodnota `FA`, tj. hodnota

$$(N + i - P_i)^2 .$$

Na řádku 15 se provádí

$$\bar{v}_l = 1.$$

Průchod poly nazpět se zaznamenanými operacemi se vyvolá z řádku 16 subroutinou `RVRSWP()`. Po jejím provedení jsou spočtené hodnoty derivací přiřazeny do proměnných `GA` (řádky 17 až 19).

Řádky 5 až 8 (na obrázku 3.3) jsou ponechány beze změny, protože nejsou závislé na nezávislých proměnných `X(.)`. Na řádku 1 (na obrázku 3.3) je počáteční nastavení indexu do polí na ukládání provedených operací.

Pro porovnání doby výpočtu jsme tuto úlohu provedli také s výpočtem derivací pomocí poměrných diferencí (ve vstupním souboru `P.UFO` jsme přiřazení `$IADA=1` nahradili přiřazením `$IADA=0`). Doby trvání výpočtů jsou uvedeny v tabulce 3.1. Při výpočtu pomocí automatického derivování spočtené hodnoty konvergují k řešení rychleji a výpočet trvá kratší dobu. □

`INTEGER IAD_W($$NF+1).`

Počet proměnných	automatické derivování	poměrné diference
$N = 10$	0.11 s	0.16 s
$N = 20$	0.49 s	0.94 s
$N = 50$	1.37 s	6.97 s
$N = 100$	3.36 s	44.93 s

Tabulka 3.1: Porovnání doby trvání výpočtu – Příklad UFO.

Kapitola 4

Automatické derivování a subgradient

V této kapitole automatické derivování zobecníme – zeslabíme požadavek spojitosti derivací elementárních funkcí řádu n na lipschitzovskost, případně konvexitu. V bodech, kde derivace nebude spojitá, budeme pro funkce absolutní hodnota a maximum schopni spočíst nějaký subgradient. Tato hodnota je dostatečná pro použití v nehladkých optimalizačních metodách, jak budeme demonstrovat v závěru této kapitoly.

4.1 Lipschitzovské funkce a subgradient

Nejprve uvedeme některé vlastnosti lipschitzovských funkcí. Dále zavedeme pojem subgradient a subdiferenciál a uvedeme jejich vlastnosti pro lipschitzovské funkce. Nedokázaná tvrzení jsou převzata z [4], [17] nebo [20].

Definice 4.1
Řekneme, že funkce $f : \mathbb{R}^n \to \mathbb{R}$ je *lipschitzovská* v okolí bodu $x \in \mathbb{R}^n$ (s konstantou L), jestliže existuje $\varepsilon > 0$ tak, že platí

$$|f(x_2) - f(x_1)| \leq L\|x_2 - x_1\|,$$

pokud $x_1 \in B(x, \varepsilon)$ a $x_2 \in B(x, \varepsilon)$.

Řekneme, že funkce $f : \mathbb{R}^n \to \mathbb{R}$ je *lokálně lipschitzovská* v oblasti Ω, je-li lipschitzovská v okolí každého bodu $x \in \Omega$. □

O existenci derivací lipschitzovských funkcí vypovídá následující tvrzení:

Věta 4.1 (Rademacher)
Nechť funkce $f : \mathbb{R}^n \to \mathbb{R}$ je lokálně lipschitzovská v oblasti $\Omega \in \mathbb{R}^n$. Pak f je diferencovatelná skoro všude, tj. množina $\{x \in \mathbb{R}^n : \nabla f(x) \text{ neexistuje}\}$ má Lebesgueovu míru nula.

Definice 4.2
Řekneme, že funkce $f : \mathbb{R}^n \to \mathbb{R}$ má v bodě $x \in \mathbb{R}^n$ *směrovou derivaci* ve směru

$h \in \mathbb{R}^n$, existuje-li konečná limita

$$f'(x,h) = \lim_{t \to 0+} \frac{f(x+th) - f(x)}{t}.$$

□

Protože v některých bodech nemusí pro lipschitzovské funkce derivace existovat, pojem derivace zobecníme a pojem zavedeme subdiferenciál a subgradient.

Definice 4.3
Zobecněnou (Clarkovu) směrovou derivaci funkce $f : \mathbb{R}^n \to \mathbb{R}$ v bodě $x \in \mathbb{R}^n$ ve směru $h \in \mathbb{R}^n$ definujeme předpisem

$$f^0(x,h) = \limsup_{\substack{y \to x \\ t \to 0+}} \frac{f(y+th) - f(y)}{t}.$$

□

Definice 4.4
Nechť funkce $f : \mathbb{R}^n \to \mathbb{R}$ je lipschitzovská v okolí bodu $x \in \mathbb{R}^n$. Pak množinu

$$\partial f(x) = \{g \in \mathbb{R}^n : f^0(x,h) \geq g^T h \quad \forall h \in \mathbb{R}^n\}$$

nazveme *subdiferenciálem* funkce f v bodě x. Elementy $g \in \partial f(x)$ budeme nazývat *subgradienty* funkce f v bodě x. □

V bodech, kde neexistuje derivace, je pro numerické optimalizační metody postačující namísto derivace použít subdiferenciál, respektive nějaký subgradient. Základní vlastnosti subgradientu popisují následující tvrzení.

Věta 4.2
Nechť funkce $f : \mathbb{R}^n \to \mathbb{R}$ je spojitě diferencovatelná v bodě $x \in \mathbb{R}^n$. Pak f je lipschitzovská v okolí bodu x a platí

$$\partial f(x) = \{\nabla f(x)\}. \tag{4.1}$$

□

Věta 4.3
Je-li funkce $f : \mathbb{R}^n \to \mathbb{R}$ lipschitzovská v okolí bodu $x \in \mathbb{R}^n$ a diferencovatelná v tomto bodě, platí

$$\nabla f(x) \in \partial f(x).$$

Rovnost (4.1) lze dokázat pouze v případě spojité diferencovatelnosti. □

Věta 4.4
Nechť funkce $f : \mathbb{R}^n \to \mathbb{R}$ je lokálně lipschitzovská v okolí bodu $x \in \mathbb{R}^n$, který je jejím lokálním extrémem (minimem nebo maximem). Pak platí

$$0 \in \partial f(x).$$

□

Pro zdůvodnění výpočtu subgradientu funkcí absolutní hodnota nebo maximum pomocí automatického derivování je zapotřebí dokázat následující tvrzení.

Věta 4.5
Nechť funkce $f_i : \mathbb{R}^n \to \mathbb{R}$, $i \in \{1, \ldots, k\}$ jsou lipschitzovské v okolí bodu $\bar{x} \in \mathbb{R}^n$. Definujme funkci $f : \mathbb{R}^n \to \mathbb{R}$ předpisem

$$f(x) = \max_{i=1,\ldots,k} f_i(x).$$

Nechť $l \in \{1, \ldots, k\}$ je takové, že $f(\bar{x}) = f_l(\bar{x})$, a nechť $\nabla f_l(\bar{x})$ existuje. Pak

$$\nabla f_l(\bar{x}) \in \partial f(\bar{x}).$$

Důkaz:
Potřebujeme ukázat, že pro všechna $h \in R^n$ platí

$$f^0(\bar{x}, h) \geq \nabla f_l(\bar{x})^T h.$$

Protože $f(z) \geq f_l(z)$ pro z v okolí bodu $\bar{x}$, platí

$$f^0(\bar{x}, h) = \limsup_{\substack{y \to \bar{x} \\ t \to 0+}} \frac{f(y + th) - f(y)}{t} \geq \limsup_{\substack{y \to \bar{x} \\ t \to 0+}} \frac{f_l(y + th) - f(y)}{t}.$$

Omezíme-li se na $y = \bar{x}$, dostáváme

$$\limsup_{\substack{y \to \bar{x} \\ t \to 0+}} \frac{f_l(y + th) - f(y)}{t} \geq \limsup_{t \to 0+} \frac{f_l(\bar{x} + th) - f(\bar{x})}{t}$$

a protože $f(\bar{x}) = f_l(\bar{x})$, je dále

$$\limsup_{t \to 0+} \frac{f_l(\bar{x} + th) - f(\bar{x})}{t} = \lim_{t \to 0+} \frac{f_l(\bar{x} + th) - f_l(\bar{x})}{t} = \nabla f_l(\bar{x})^T h.$$

Tím je důkaz proveden. □

4.2 Funkce maximum a absolutní hodnota, výpočet subgradientu

Funkce *maximum* (resp. *minimum*) a *absolutní hodnota* nemají derivace na celém svém definičním oboru. Přitom v praxi bývají tyto funkce, jako elementární funkce, součástí optimalizovaných funkcí. Z toho důvodu byly vyvinuty optimalizační metody, kterým v bodě s neexistující derivací postačí hodnota nějakého subgradientu. Proto je užitečné rozšířit automatické derivování i pro výpočet subgradientu.

Implemetaci výpočtu subgradientu pomocí automatického derivování pro funkci maximum lze podle Věty 4.5 provést takovým způsobem, že se spočte gradient té funkce, která maximum v bodě určuje. Neboli, označme

$$f(x) = \max_{i=1,\ldots,k} f_i(x)$$

a nechť například v bodě $\bar{x}$ je $l \in \{1, \ldots, k\}$ takové, že $f(\bar{x}) = f_l(\bar{x})$, a nechť existuje $\nabla f_l(\bar{x})$, pak automatické derivování spočte hodnotu $\nabla f_l(\bar{x})$.

Je-li v tomto bodě $\bar{x}$ funkce f je hladká, reprezentuje spočtená hodnota $\nabla f_l(\bar{x})$ gradient funkce f v bodě $\bar{x}$.

Jestliže v bodě $\bar{x}$ funkce f není hladká, podle Věty 4.5 (a za jejích předpokladů) se tedy spočetl nějaký subgradient funkce f v bodě $\bar{x}$.

Poznamenejme, že pro funkci minimum platí analogická situace. Pro funkci absolutní hodnota

$$f(x) = |g(x)|$$

platí také analogická tvrzení, neboť

$$|a| = \max(a, -a).$$

4.3 Implementace funkce maximum a absolutní hodnota

V tomto odstavci stručně ukážeme, jak lze automatické derivování pro funkci maximum implemetovat. Doplníme tak vlastně odstavce 1.6.3 a 3.4.1 o další funkce a detaily.

Jak již víme, každá elementární funkce $\varphi_i(v_j)_{j \prec i}$ v postupu výpočtu hodnoty $y = f(x)$ (kterou chceme derivovat) je v průběhu transformace nahrazena subrutinou, která kromě původní operace $\varphi_i(v_j)_{j \prec i}$ provede i zaznamenání svého zavolání do polí `V`, `VDOT`, `OPCODE`, `ARG1`, `ARG2`, případně také `VDOT`, `VBARDT`. Elementární funkce maximum je takto nahrazena subrutinou `MAXG`:

```
!--- ztransformovaná operace maximum ---
INTEGER FUNCTION MAXG(IARG1, IARG2)     !vstupní parametry: pořadí (index)
                                        !numerických hodnot v polích, ze
                                        !kterých určíme maximum
                                        !výstupní hodnota: pořadí (index)
                                        !v polích pro právě tuto operaci
  COMMON /AD_F1/ V, VBAR, OPCODE, ARG1, ARG2, INDARR
    REAL*8 V($NADARR), VBAR($NADARR)
    INTEGER OPCODE($NADARR), ARG1($NADARR), ARG2($NADARR)
    INTEGER INDARR
  INTEGER IARG1, IARG2

  IF(V(IARG1).GT.V(IARG2)) THEN         !spočtení elementární operace maximum
    V(INDARR)=V(IARG1)
                           ELSE
    V(INDARR)=V(IARG2)
```

```
  ENDIF
  VBAR(INDARR)=0.0D0                    !vynulování proměnné v_i s pruhem
                                        !(bude se do ní postupně přičítat)
  OPCODE(INDARR)=100                    !zaznamenání kódu této operace (100=maximum)
  ARG1(INDARR)=IARG1                    !zaznamenání pořadí (indexu) v polích
                                        !pro první (levý) argument
  ARG2(INDARR)=IARG2                    !zaznamenání pořadí (indexu) v polích
                                        !pro druhý (pravý) argument
  MAXG=INDARR                           !pořadí této operace
  INDARR=INDARR+1                       !příprava na další elementární operaci -
                                        !- posunutí indexu ("ukazovátka") do polí,
                                        !kam až jsou pole vyplněny
END
```

Do procedury RVRSWP, která prochází poli se zaznamenanými hodnotami a počítá derivace, jsme doplnili funkci maximum:

```
SUBROUTINE RVRSWP()
  COMMON /AD_F1/ V, VBAR, OPCODE, ARG1, ARG2, INDARR
    REAL*8 V($NADARR), VBAR($NADARR)
    INTEGER OPCODE($NADARR), ARG1($NADARR), ARG2($NADARR)
    INTEGER INDARR
  INTEGER I

  DO 999, I=INDARR-1, 1, -1             !cyklus přes jednotlivé operace pozpátku,
                                        !tj. průchod polemi odzadu dopředu
   .
   .
    ELSEIF(OPCODE(I).EQ.100) THEN       !operace maximum
      IF(V(ARG1(I)).GT.V(ARG2(I))) THEN
        VBAR(ARG1(I))=VBAR(ARG1(I))+VBAR(I)  !postupné přičítání hodnot
                                             !do proměnné v_. s pruhem
        VBAR(ARG2(I))=VBAR(ARG2(I))          !postupné přičítání hodnot
                                             !do proměnné v_. s pruhem
                                             !tento řádek lze samozřejmě vynechat
                                   ELSE
        VBAR(ARG1(I))=VBAR(ARG1(I))          !postupné přičítání hodnot
                                             !do proměnné v_. s pruhem
                                             !tento řádek lze samozřejmě vynechat
        VBAR(ARG2(I))=VBAR(ARG2(I))+VBAR(I)  !postupné přičítání hodnot
                                             !do proměnné v_. s pruhem
      ENDIF
   .
   .
    ENDIF
999 CONTINUE
    END
```

Jak jsme již zmínili výše, pro funkce minimum a absolutní hodnota platí tvrzení analogická. Tvar jednotlivých subrutin při výpočtu druhých derivací lze odvodit stejně jako v kapitole 3.4 – aplikací přímého módu na zde uvedené subrutiny. Elementární funkce maximum je tedy nahrazena subrutinou MAXH:

```
!--- ztransformovaná operace maximum ---
```

```
INTEGER FUNCTION MAXH(IARG1, IARG2)     !vstupní parametry: pořadí (index)
                                        !numerických hodnot v polích, ze
                                        !kterých určíme maximum
                                        !výstupní hodnota: pořadí (index)
                                        !v polích pro právě tuto operaci
  COMMON /AD_F2/ V, VBAR, VDOT, VBARDT, OPCODE, ARG1, ARG2, INDARR
    REAL*8 V($NADARR), VBAR($NADARR), VDOT($NADARR),VBARDT($NADARR)
    INTEGER OPCODE($NADARR), ARG1($NADARR), ARG2($NADARR)
    INTEGER INDARR
  INTEGER IARG1, IARG2

  IF(V(IARG1).GT.V(IARG2)) THEN         !spočtení elementární operace maximum
    V(INDARR)=V(IARG1)
    VDOT(INDARR)=VDOT(IARG1)            !aplikace přímého módu
                           ELSE
    V(INDARR)=V(IARG2)
    VDOT(INDARR)=VDOT(IARG2)            !aplikace přímého módu
  ENDIF
  VBAR(INDARR)=0.0D0                    !vynulování proměnné v_i s pruhem
                                        !(bude se do ní postupně přičítat)
  VBARDT(INDARR)=0.0D0                  !vynulování promenné v_i s pruhem a teckou
                                        !(bude se do ní postupne pricítat)
  OPCODE(INDARR)=100                    !zaznamenání kódu této operace (100=maximum)
  ARG1(INDARR)=IARG1                    !zaznamenání pořadí (indexu) v polích
                                        !pro první (levý) argument
  ARG2(INDARR)=IARG2                    !zaznamenání pořadí (indexu) v polích
                                        !pro druhý (pravý) argument
  MAXH=INDARR                           !pořadí této operace
  INDARR=INDARR+1                       !příprava na další elementární operaci -
                                        !- posunutí indexu ("ukazovátka") do polí,
                                        !kam až jsou pole vyplněny
END
```

A dále, do procedury RVRSWPH, která prochází poli se zaznamenanými hodnotami a počítá derivace, jsme doplnili funkci maximum:

```
SUBROUTINE RVRSWPH()
  COMMON /AD_F2/ V, VBAR, VDOT, VBARDT, OPCODE, ARG1, ARG2, INDARR
    REAL*8 V($NADARR), VBAR($NADARR), VDOT($NADARR),VBARDT($NADARR)
    INTEGER OPCODE($NADARR), ARG1($NADARR), ARG2($NADARR)
    INTEGER INDARR
  INTEGER I

  DO 998, I=INDARR-1, 1, -1              !cyklus přes jednotlivé operace pozpátku,
                                         !tj. průchod polemi odzadu dopředu
   .
   .
    ELSEIF(OPCODE(I).EQ.100) THEN        !operace maximum
      IF(V(ARG1(I)).GT.V(ARG2(I))) THEN
        VBAR(ARG1(I))=VBAR(ARG1(I))+VBAR(I)  !postupné přičítání hodnot
                                             !do proměnné v_. s pruhem
        VBARDT(ARG1(I))=VBARDT(ARG1(I))+     !postupné přičítání hodnot
&                       VBARDT(I)            !do proměnné v_. s pruhem a tečkou
                                  ELSE
```

```
        VBAR(ARG2(I))=VBAR(ARG2(I))+VBAR(I)  !postupné přičítání hodnot
                                             !do proměnné v_. s pruhem
        VBARDT(ARG2(I))=VBARDT(ARG2(I))+     !postupné přičítání hodnot
&                       VBARDT(I)            !do proměnné v_. s pruhem a tečkou
      ENDIF
   .
   .
    ENDIF
998 CONTINUE
    END
```

4.4 Příklad

Výpočet subgradientu, jak jsme ho popsali v předcházejícím odstavci, je možné využít například při nehladké optimalizaci. V tomto odstavci budeme demonstrovat optimalizaci nehladké funkce pomocí svazkové metody, kde v případě neexistující derivace spočteme nějaký subgradient pomocí automatického derivování.

Nejdříve však ještě uvedeme dvě tvrzení o subgradientu součtu funkcí a zmíníme se také o svazkových metodách.

4.4.1 Subgradient součtu funkcí

Důkazy následujících dvou tvrzení o subgradientu součtu funkcí jsou uvedeny například ve [20].

Věta 4.6
Nechť funkce $f_1 : \mathbb{R}^n \to \mathbb{R}$ a $f_2 : \mathbb{R}^n \to \mathbb{R}$ jsou lipschitzovské v okolí bodu $x \in \mathbb{R}^n$. Pak

$$\partial(f_1 + f_2)(x) \subset \partial f_1(x) + \partial f_2(x). \tag{4.2}$$

Je-li alespoň jedna z nich spojitě diferencovatelná v bodě x, nastává ve vztahu (4.2) rovnost. □

Věta 4.7
Nechť funkce $f_i : \mathbb{R}^n \to \mathbb{R}$, $i = 1, \dots m$, jsou lipschitzovské v okolí bodu $x \in \mathbb{R}^n$. Pak

$$\partial \left(\sum_{i=1}^{m} f_i \right)(x) \subset \sum_{i=1}^{m} \partial f_i(x). \tag{4.3}$$

Rovnost ve vztahu 4.3 nastane, jsou-li všechny funkce f_i až na jednu spojitě diferencovatelné. □

4.4.2 Optimalizace nehladkých funkcí – svazkové metody

V tomto odstavci stručně popíšeme *svazkovou metodu*, tj. jednu z metod pro nehladkou optimalizaci. Její podrobný popis je uveden například v [22].

Budeme minimalizovat funkci $f : \mathbb{R}^n \to \mathbb{R}$, která je lokálně lipschitzovská. Protože lokálně lipschitzovská funkce je podle Rademacherovy věty 4.1 diferencovatelná skoro všude, existuje gradient skoro všude a označme ho $g(x) = \nabla f(x)$. V bodech, kde derivace funkce f neexistuje, je možné spočítat nějaký subgradient $g(x) \in \partial f(x)$, kde $\partial f(x)$ je subdiferenciál funkce $f(x)$.

Protože hodnota subgradientu nemusí dostatečně popisovat chování funkce f v okolí bude x, používá se raději "svazek hodnot" $y_j \in \mathbb{R}^n$, $j \in J_k \subset \{1, 2, ...k\}$, který toto chování lépe popisuje.

Namísto funkce f se používá její "kvadratický model"

$$f_Q^k(x) = \frac{1}{2}(x - x_k)^T G_k (x - x_k) + \max_{j \in J_k} (f(x_k) + g(y_j)^T (x - x_k) - \alpha_j^k),$$

kde $\alpha_j^k = f(x_k) - f(y_j) - g(y_j)^T (x_k - y_j)$. Její minimalizací je vybrán směrový vektor d_k. Délka kroku t_k se volí například tak, aby $x_{k+1} = x_k + t_k d_k$ byl buď spádový nebo nulový krok. Více podrobností je uvedeno například v [22]. Nová hodnota y_{k+1} je také přidána, respektive nahradí jinou, ve svazku hodnot y_j, $j \in J_{k+1} \subset \{1, 2, ...k + 1\}$.

4.4.3 Příklad

Výpočet nějakého subgradientu pomocí automatického derivování a jeho aplikaci demonstrujme na již zmíněné svazkové metodě (viz také například [20] nebo [21]) a na testovací úloze 3.15 z balíku testovacích úloh [19].

Označme zobrazení $F : \mathbb{R}^6 \to \mathbb{R}$:

$$F(x) = \sum_{i=1}^{50} \left| x_1 e^{-x_2 t_i} \cos(x_3 t_i + x_4) + x_5 e^{-x_6 t_i} - y_i \right|, \tag{4.4}$$

kde

$$\begin{aligned} y_i &= 0.5e^{-t_i} - e^{-2t_i} + 0.5e^{-3t_i} + 1.5e^{-1.5t_i} \sin 7t_i + e^{-2.5t_i} \sin 5t_i \\ t_i &= 0.1(i-1), \quad 1 \le i \le 51. \end{aligned}$$

Hledáme lokální minimum zobrazení F s počáteční iterací $x_0 = (0, 2, 7, 0, -2, 1)$.

Výpočet jsme provedli pomocí svazkové metody z odstavce 4.4.2, viz též [21]. Porovnali jsme tři varianty, kdy derivace (resp. subgradient) byla vypočítána analyticky, pomocí automatického derivování a pomocí poměrných diferencí.

V bodech, kde neexistuje derivace výrazu

$$\left| x_1 e^{-x_2 t_i} \cos(x_3 t_i + x_4) + x_5 e^{-x_6 t_i} - y_i \right|, \tag{4.5}$$

se vyhodnocoval nějaký jeho subgradient. Věta 4.7 nám poskytuje zdůvodnění faktu, že sečtením gradientů (příp. nějakého subgradientu) výrazů (4.5) dostaneme gradient (příp. subgradient) funkce (4.4). Předpoklad spojitosti derivací funkcí f_i, až na jednu z nich, byl ověřen v průběhu výpočtu nenulovostí argumentu absolutní hodnoty.

Doby trvání výpočtů jsou uvedené v tabulce 4.1. Nejkratší dobu trval výpočet s derivací spočtenou analyticky, jen o zlomek sekundy více potřeboval výpočet s derivací spočtenou automaticky. Nejdelší dobu, přibližně 2,5krát více, trval výpočet s derivací spočtenou pomocí poměrných diferencí. Získané hodnoty lokálního minima jsou pro všechny způsoby výpočtu derivace shodné.

derivace spočtená pomocí:	doba výpočtu
analyticky "v ruce"	0,0605 s
automatickým derivováním	0,0635 s
poměrných diferencí	0,1547 s

Tabulka 4.1: Doba výpočtu při nehladké optimalizace, pro různé druhy výpočtu derivací pro příklad 4.4.3.

Poznámka. Programy pro výše uvedený příklad vznikaly tak, že nebylo přihlíženo ke struktuře úlohy a programu. Pokud však využijeme toho, že automatické derivování spočítá navíc hodnotu funkce při výpočtu její derivace, lze dobu trvání programu s výpočtem derivace pomocí automatického derivování snížit na 0,0575 sekundy, což je dokonce méně než pro analytický výpočet derivace. Důvodem je analytický postup výpočtu derivace, který je náročný a který nepřihlíží ke struktuře derivované funkce, zatímco automatické derivování využívá již spočtených a uložených hodnot.

Závěr

Automatické derivování je numerická metoda pro výpočet derivací funkce, která je zadaná programem. Jeji půvab spočívá v jednoduchosti použití a efektivnosti výpočtu derivace. Cílem této disertační práce byla implementace automatického derivování do systému UFO, aby bylo možné využít efektivní výpočet derivací při optimalizaci funkcí. V první kapitole jsme tedy popsali základní principy automatického derivování, jeho vlastnosti a možnosti implementace. Ve druhé kapitole jsme uvedli stručný popis systému UFO. Ve třetí kapitole jsme detailně popsali implementaci automatického derivování do systému UFO. Tato implementace byla navržena s ohledem na jeho specifické vlastnosti, požadavky a použití. Dále byla tato implementace zrealizována a v současné době je automatické derivování součástí programového balíku UFO. Uvedené postupy implementace jsou univerzální a lze je použít i pro realizaci automatického derivování v jiných systémech nebo aplikacích.

Zároveň bylo ve čtvrté kapitole automatické derivování rozšířeno o efektivní výpočet subgradientu v případě, že mezi elementárními funkcemi jsou i nehladké funkce maximum, minimum nebo absolutní hodnota, a gradient tedy nemusí existovat. Hodnotu subgradientu je možné použít namísto gradientu například v nehladkých optimalizačních metodách.

Literatura

[1] *Computational differentiation – techniques, applications, and tools.* Sborník. Ed. Berz M., Bischof C. H., Corliss G. F., Griewank A. SIAM, Philadelphia, 1996.

[2] Bischof C., Carle A., Corliss G., Griewank A., Hovland P.: *ADIFOR: Fortran Source Translation for Efficient Derivatives.* Mathematics and Computer Science Division, Argonne National Laboratory, 1991.

[3] *Automatic Differentiation: Applications, Theory, and Implementations.* Ed. Bucker H. M., Corliss G. F., Hovland P. D., Naumann U., Norris B. Springer, 2005.

[4] Fletcher, R.: *Practical Methods of Optimization.* John Wiley & Sons, 1993.

[5] Griewank A.: *Evaluation Derivatives: Principles and Techniques of Algorithmic Differentiation.* SIAM, Philadelphia, 2000.

[6] *Automatic Differentiation of Algorithms: Theory, Implementation, and Application.* Sborník. Ed. Griewank A., Corliss G. F. SIAM, Philadelphia, 1992.

[7] Griewank A.: *On Automatic Differentiation.* Preprint ANL/MCS-P10-1088, Argonne National Laboratory, Illinois, USA, 1988.

[8] Hartman J.: *Realizace metod pro automatické derivování.* Diplomová práce. Matematicko-fyzikální fakulta, Univerzita Karlova, Praha, 2001.

[9] Hartman J., Lukšan L.: *Automatické derivování v systému UFO.* Technická zpráva V-1002. ICS AS CR, Praha, 2007.

[10] Hartman J., Zítko J.: *Principy automatického derivování.* Výzkumná zpráva. Katedra numerické matematiky, Matematicko-fyzikální fakulta, Univerzita Karlova, Praha, 2006.

[11] Hartman J., Lukšan L. Zítko J.: Automatic differentiation and its program realization. Zasláno do Kybernetiky.

[12] Hartman J.: *Automatic Differentiation and Nonsmooth Optimization*, In: Sborník zimní školy Seminar on Numerical Analysis, ICS AS CR, Praha, 2006.

[13] Hartman J.: *Principy automatického derivování.* In: Proceedings of the 11th Annual Conference of Doctoral Students - WDS 2002, Matfyz Press, Prague, 2002.

[14] Jarník V.: *Diferenciální počet II.* Academia, Praha, 1984.

[15] Kagiwada H., Kalaba R., Rasakhoo N., Spingarn K.: *Numerical derivatives and nonlinear analysis*, Mathematical Concepts and Methods in Science and Engineering, Plenun Press, New York - London, 1986.

[16] 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.

[17] Kiwiel K. C.: *Methods of Descent for Nondifferentiable Optimization.* Lecture Notes in Mathematics, Vol. 1133, Springer Verlag, Berlin, 1985.

[18] Lukšan L., Tůma M., Hartman J., Vlček J., Ramešová N., Šiška M., Matonoha C.: *UFO 2006 - Interactive system for universal functional optimization.* Technická zpráva V-977. ICS AS CR, Praha, 2006.

[19] Lukšan L., Vlček J.: *Test Problems for Nonsmooth Unconstrained and Linearly Constrained Optimization.* Technická zpráva V-798, ICS AS CR, Praha, 2000.

[20] Lukšan L., Vlček J.: *Základy nehladké analýzy.* Učební text.
`http://www.cs.cas.cz/~ luksan/lekce3.ps`

[21] Lukšan L., Vlček J.: *NDA: Algorithms for Nondifferentiable Optimization.* Technická zpráva V-797, ICS AS CR, Praha, 2000.

[22] Lukšan L.: *Numerické optimalizační metody.* Technická zpráva V-923, ICS AS CR, Praha, 2005.

[23] Rall L. B.: *Automatic differentiation: Techniques and applications.* Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1981.

[24] Verma A.: *Structured Automatic Differentiation.* Disertační práce, Cornell University, 1998.

[25] Virius M.: *Programování v C++.* ČVUT, Praha, 1999.

[26] Walther A. , Griewank A. and Vogel O.: *ADOL-C: Automatic differentiation using operator overloading in C++.* In PAMM 2:41 – 44 (2003).