Odvedljivost programskih jezikov $\mathcal{T}$-calculus

Žiga Sajovic, Martin Vuk

1. julij 2016

Povzetek

Dandanes matematične funkcije pogosto računamo z računalniki. Če je funkcija odvedljiva, nas včasih zanima tudi, koliko je njen odvod. Večina bralcev verjetno pozna simbolično odvajanje izrazov ali numerično odvajanje s končnimi diferencami. V tem prispevku bova predstavila še en način, kako lahko določimo odvode funkcije, ki jo računamo z računalniškim programom. *Avtomatsko odvajanje* je skupno ime za različne postopke, s katerimi program, ki računa vrednosti odvedljive funkcije, preoblikujemo v program, ki izračuna odvod. Večina algoritmov za avtomatsko odvajanje znajo to narediti brez dodatnega človekovega posega (odtod ime avtomatski). Za razliko od numeričnega odvajanja ne trpijo za izgubo natančnosti. Prednost pred simboličnim odvajanjem, pa je v tem, da jih lahko uporabimo tudi v primeru, ko ne poznamo eksplicitne formule, ampak vrednosti računamo s programom.

Avtomatsko odvajanje je uporabno povsod, kjer potrebujemo odvode funkcij, ki jih računamo s kompleksnimi algoritmi. Tipični primeri so optimizacija parametrov pri strojnem učenju, računanje odvoda preslikave, ki nima eksplicitne formulacije (npr. Poincarejeva preslikava za dinamične sisteme, lastne vrednosti matrik, ...).

1 Uvod

Za začetek si oglejmo preprost primer, na katerem ilustriramo osnovno idejo. Recimo, da računamo vrednosti funkcije $y = \sqrt{x}$ z babilonskim oziroma Haronovim obrazcem. Pri tem postopku, najprej izberemo začetni približek za kvadratni koren $y_0 = y_z$. Nato pa z rekurzivno formulo

$$y_n = \frac{1}{2}\left(y_{n-1} + \frac{x}{y_{n-1}}\right)$$

računamo člene zaporedja y_n. Zaporedje y_n konvergira zelo hitro k vrednosti $\sqrt{x}$, zato so vrednosti y_n že po nekaj korakih zelo blizu iskani vrednosti $\sqrt{x}$. Opisani postopek lahko prevedemo v računalniški program. Na primer v programskem jeziku Python, bi program izgledal takole

```
y = x/2
while abs(y*y-x) > 5e-11:
  y = (y+x/y)/2
```

Pri tem smo za začetni približek izbrali $x/2$, lahko pa bi izbrali tudi kaj drugega. Vrednost pa nas zanima na 10 decimalk natančno. Za funkcijo $f(x)$ želimo izračunati tudi njen odvod $f'(x)$. V našem primeru lahko odvod preprosto izrazimo s funkcijo $f'(x) = \frac{1}{2\sqrt{x}}$, ampak zamislimo si, da funkcije f sploh ne poznamo, ampak poznamo le računalniški program za njen izračun. Še vedno pa bi radi izračunali vrednosti odvoda $f'(x)$. Uporabimo lahko numerične približke za odvod, vendar pri tem izgubimo natančnost.

Osnovna ideja avtomatskega odvajanja je v tem, da odvajamo program. Računalniški program ni nič drugega kot zaporedje osnovnih računskih operacij in premetavanja vrednosti po pomnilniku. V našem primeru uporabljamo dve spremenljivki `x` in `y`, zato si lahko vsako vrstico programa, ki spreminja vrednosti `x` ali `y` predstavljamo kot funkcijo $\phi_i : \mathbb{R}^2 \to \mathbb{R}^2$. Označimo z (x_k, y_k) vrednosti spremenljivk `x` in `y`, po izvedbi k-te vrstice programa in pred izvedbo $k+1$-ve vrstice. Vsaka vrstica programa določa preslikavo

$$\begin{aligned} \phi_k &: \mathbb{R}^2 \to \mathbb{R}^2 \\ \phi_k &: (x_{k-1}, y_{k-1}) \mapsto (x_k, y_k). \end{aligned}$$

Celoten program predstavlja preslikavo $F : (x_0, y_0) \mapsto (x_n, y_n)$, ki jo lahko predstavimo kot kompozitum preslikav ϕ_k, defniranih z vsako vrstico programa:

$$F(x_0, y_0) = \phi_n \circ \phi_{n-1} \circ \ldots \phi_1(x_0, y_0) = \phi_n(\phi_{n-1}(\ldots \phi_1(x_0, y_0) \ldots))$$

Odvod preslikave F lahko po verižnem pravilu izrazimo kot produkt Jacobijevih matrik $D\phi_i$ preslikav ϕ_i. Ker nas zanima le odvod y po x, matriko DF pomnožimo z leve in desne z vektorjem $(0, 1)$

$$\frac{dy}{dx} = \begin{bmatrix} 0 & 1 \end{bmatrix} \cdot D\phi_n(x_{n-1}, y_{n-1}) \cdot D\phi_{n-1}(x_{n-2}, y_{n-2}) \ldots D\phi_1(x_0, y_0) \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad (1)$$

Program za računanje odvoda lahko povsem sledi originalnemu programu, le da si mora na vsakem koraku zapomniti vmesne vrednosti odvoda. Na vsakem koraku programa, si mora program za odvod zapomniti vektor

$$\begin{bmatrix} dx_k \\ dy_k \end{bmatrix} = D\phi_k(x_{k-1}, y_{k-1}) \ldots D\phi_1(x_0, y_0) \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Vrednost (dx_k, dy_k) lahko definiramo rekurzivno

$$\begin{bmatrix} dx_k \\ dy_k \end{bmatrix} = D\phi_k(x_{k-1}, y_{k-1}) \cdot \begin{bmatrix} dx_{k-1} \\ dy_{k-1} \end{bmatrix} \quad (2)$$

V našem primeru je

$$\phi_1(x, y) = (x, x/2) \text{ in } \phi_n(x, y) = \left(x, \frac{1}{2} \left(y + \frac{x}{y} \right) \right)$$

Vrednost spremenljivke x se ne spreminja tekom programa, zato je dovolj, da spremljamo le vrednosti odvoda po y

$$\frac{d}{dx}\phi_0(x, y(x))_2 = 1/2 \text{ in } \frac{d}{dx}\phi_k(x, y(x))_2 = \frac{1}{2}\left(\frac{dy}{dx} + \left(\frac{1}{y} - \frac{x}{y^2}\frac{dy}{dx}\right)\right)$$

Naslednji program poleg vrednosti funkcije (spremenljivka `y`) računa tudi njen odvod (spremenljivka `dy`)

```
dy = 0.5
y = x/2
while abs(y*y-x) > eps:
  dy = (dy + (1/y - x/(y*y)*dy)/2
  y = (y+x/y)/2
```

2 Avtomatsko odvajanje

Na uvodnem primeru smo videli, da si lahko računalniški program predstavljamo kot vektorsko funkcijo več spremenljivk. Vektorska funkcija, ki predstavlja program, je podana kot kompozitum preslikav, ki ustrezajo posameznim vrsticam programa. Program za odvod smo napisali tako, da smo vsaki vrstici programa dodali vrstico, ki je izračunala odvod. Ta postopek lahko namesto nas opravi računalnik (od tod ime avtomatski odvod). V nadaljevanju bomo opisali dva načina, kako je to izvedeno v obstoječih knjižnicah.

Oglejmo si matematični model, s katerim lahko utemeljimo postopke avtomatičnega odvajanja. V našem modelu bomo zajeli le spremenljivke predstavljene s števili s plavajočo vejico, s katerimi v računalniku predstavljamo realna števila. Ostale spremenljivke kot so števci v zankah, celoštevilske vrednosti, logične vrednosti in podobne ne bodo vključene v našem modelu, ampak jih upoštevamo zgolj kot parametre, ki sicer vplivajo na potek programa, vendar njihovega vpliva ne moremo infinitezimalno obravnavati. Če predpostavimo, da so vse spremenjivke, ki nas zanimajo tipa `float` (ali `double`)), si lahko stanje spomina predstavljamo kot n-razsežni realni vektor[1]. Vsaka spremenljivka(lokacija v spominu) predstavlja eno komponento tega vektorja. Množico vseh možnih stanj spomina, s katerim razpolaga program, lahko modeliramo z n-razsežnim vektorskim prostorom $V \simeq \mathbb{R}^n$. Vektorski prostor V bomo imenovali *virtualni pomnilnik programa*. Računalniški program, ki ima na voljo n-mest v virtualnem pomnilniku, si lahko predstavljamo kot preslikavo

$$P : V \to V, \tag{3}$$

kjer je V vektorski prostor dimenzije n. Množica vseh takih preslikav opremljena z operacijo kompozituma je *monoid*.

[1] omejitev, da je spremenljivka tipa `float`, smo vpeljali zgolj zaradi enostavnosti. Odvajamo lahko po poljubnem tipu, ki to dopušča. Če bi npr. definirali poseben tip, ki bi predstavljal funkcijo, bi na isti način lahko računali funkcijski odvod, kot ga poznamo v variacijskem računu.

Definicija 2.1. *Naj bo V Banachov vektroski prostor. Preslikava $P : V \to V$ je odvedljiva v točki $x \in V$, če obstaja linearen omejen operator $TP_x : V \to V$, za katerega je*

$$\lim_{h \to 0} \frac{\|P(x+h) - P(x) - TP_x(h)\|}{\|h\|} = 0. \tag{4}$$

Preslikavo TP_x imenujemo Fréchetov odvod *ali* linearizacija *preslikave P.*

Za preslikave $\mathbb{R}^n \to \mathbb{R}^m$ lahko Fréchetov odvod izrazimo kot množenje vektorja h z Jacobijevo matriko parcialnih odvodov komponent preslikave P

$$TP_x(h) = JP(x) \cdot h.$$

Preslikave, ki ustrezajo osnovnim ukazom in operacijam v določenem programskem jeziku, predstavljajo generatorje vseh možnih programov v monoidu. Označimo z $\mathcal{E}$ množico vseh elementarnih ukazov in operacij, ki so implementirane v programskem jeziku. Označimo z $\mathcal{T}$ prostor končnih programov, ki so generirani z elementarnimi operacijami iz $\mathcal{E}$

$$\mathcal{T} = \langle E \rangle$$

Predpostavimo za trenutek, da so vse elementarne operacije povsod odvedljive.

Izrek 2.1. *Naj bo $\tau : V \times T \to T$, $\tau : (x, P) \mapsto TP_x$ preslikava, ki vsakemu programu P priredi njegov odvod v dani točki x. Predpostavimo še, da sta med generatorji $\mathcal{E}$ tudi operaciji seštevanja in množenja. Za vsak $x \in V$ je preslikava $\tau_x : P \mapsto \tau(x, P)$ avtomorfizem monoida $(T, \circ)$.*

Če $\mathcal{E}$ vsebuje seštevanje in množenje, potem so v $\mathcal{T}$ tudi vse linearne preslikave na V. Dejstvo, da je τ_x homomorfizem za operacijo kompozituma, je pa neposredna posledica verižnega pravila.

2.1 Odvedljivost $\mathcal{T}$

Če želimo tekom programa sproti računati tudi odvode, moramo hraniti vmesne vrednosti odvodov. Zato moramo pomnilnik (vektorski prostor V) razširiti, da bo vseboval tudi odvode. Odvod je linearna preslikava $V \to V$, prostor vseh linearnih preslikav $\mathcal{L}(V)$ je izomorfen tenzorskemu produktu $V^* \otimes V$. Tako za izvedbo programa, ki računa odvode, potrebujemo virtualni pomnilnik izomorfen prostoru $V \oplus (V^* \otimes V)$. Poleg vsake spremenljivke x_j, ki jo uporabimo v programu, si moramo zapomniti še vrednosti odvodov po vseh ostalih spremenljivkah $\frac{\partial x_j}{\partial x_i}$.

Za višje odvode rekurzivno definiramo zaporedje vektorskih prostorov V_n, ki predstavljajo virtualni pomnilnik

$$V_0 = V \tag{5}$$
$$V_{n+1} = V \oplus (V^* \otimes V_n), \tag{6}$$

ki ga potrebuje program za izračun vrednosti n-tih odvodov. Izbira $V_n \simeq V \oplus (V \otimes V) \oplus (V \otimes V \otimes V) \ldots$ odraža dejstvo, da za izračun vrednosti n-tega odvoda v neki točki $x \in V$ potrebujemo vrednosti funkcije in vseh nižjih odvodov.

Naj bo P $\subset \mathcal{T}$ preslikava $V_{\mathbb{J}} \to V_{\mathbb{I}}$. Slika predstavlja mnogoterost $\mathcal{M} \subset V_{\mathbb{I}}$ (dejansko je unija zlepkov kot definrano v 21).

Spomnimo se popolnega diferenciala preslikave

$$df = \sum_{\forall_j} \frac{\partial f}{\partial x_j} dx_j, x_i \in V_{j \in \mathbb{J}}$$

in naj dx_j predstavljajo bazo duala $V_{\mathbb{J}}^*$. Poslužili se bomo tako povleka (Obratno), kot potiska (Direktno), odvisno od implementacije.

$$d_p \mathcal{T} : T_p V \to T_{\mathcal{T}x} V \tag{7}$$

$$d_p \mathcal{T}(\frac{\partial}{\partial u^a}) = \frac{\partial \mathcal{T}^b}{\partial u^a} \frac{\partial}{\partial v^b} \tag{8}$$

Potem lahko definiramo prostor, ki ga generiramo kot direktno vsoto $V \bigoplus (V^* \bigotimes V)$. Dotično nas zanima podprostor

$$\mathcal{V} = \mathcal{M} \bigoplus_{\forall_i} \partial_i \mathcal{M}$$

Definirajmo operator

$$\partial^{\oplus} : \mathcal{T} \to \mathcal{T} \bigoplus_{\forall_i} \partial_i \mathcal{T} \tag{9}$$

ki slika $\mathcal{E} \to \mathcal{E} + d\mathcal{E}$. Transformacija razširi operator monoida $\circ$, ki se pri $e_{i<1}$ izraža kot povlek/potisk.

Skupaj z definicijami prejšnjih sestavkov, lahko konstruiramo odvedljiv abstrakten stroj, ki z opisano mehaniko razširja množico Turing-izračunljivih strojev.

$$M = \langle \mathcal{T}, V, \partial^{\oplus} \rangle \tag{10}$$

- V služi kot množica stanj, domena ter "neskončen trak"
- $\mathcal{T}$ je monoid operacij nad V
- $\partial^{\oplus}$ razširja operand s svojim dualom

Karkoli operira pod temi specifikacijami, je odvedljiv Turingov stroj.

Ob konstrukciji M, je potrebna le enkratna aplikacija $\partial^{\oplus}$. Naj $\mathcal{T}$ označuje programski jezik generiran z $\langle \mathcal{E} \rangle$ in M stroj, ki ga implementira. Definirajmo stroj prvega reda

$$\partial^{\oplus} M = \langle \partial^{\oplus} \mathcal{T}, \partial^{\oplus} V, \partial^{\oplus} \rangle \tag{11}$$

oziroma v poznani, konceptualno ekvivalentni notaciji

$$M' = \langle \mathcal{T}', V', \partial^{\oplus} \rangle \tag{12}$$

Vsi programi konstruirani v M', skozi zaporedje akcij $\mathcal{T}'$ nad V', v vsaki točki izvajanja vsebujejo informacijo spremembe prvega reda. Tak program označimo s P'. Potem je $P'V'$ mnogoterost, na vsaki točki razširjena s kotangentnim svežnjem.

Z ustrezno izbiro projekcije $\mathcal{P}_{\mathbb{I}}$ programa, kot $\mathcal{M} \subset V$, omogočimo preučevanje njegovega učinka na V. Projeciramo lahko na enotske vektorje, ki predstavljajo konceptualni vhod programa in proučujemo spremembe akcije nad V. S tem merimo odzivnost programa na vhodne podatke. Diagram izvajanja programa lahko predstavimo z usmerjenim grafom;- potem lahko projeciramo na, in odvajamo po kateremkoli izmed njegvih oglišč.

Veliko algoritmov v uporabi, vsebuje hiper-parametre[2], katerih natančne vrednosti so odvisne od specifičnosti aplikacije. Skozi projekcijo na $\mathcal{H} \subset V$, kjer $\mathcal{H}$ predstavljaja hiper-parametre, algoritem interpretiramo, kot parametrično družino mnogoterosti. Konstrukcija algoritmov skozi M (10) omogoča nov pristop k rešitvi problema izbire hiper-parametrov, ki je danes še odprt. Njihovo uravnavanje, ki je navadno prepuščeno izkušnji programerja, v najini formulaciji postane ekvivalentno optimizaciji funkcij v prostoru parametrov. To pa je problem s številnimi učinkovitimi predlogi rešitve, t.j. ob uvedbi ustreznega energijskega funkcionala (mere napake), postanejo ti algoritmi naučljivi.

2.2 Višji odvodi

V nadaljnjem se ob odobravanju n-tega reda predpostavlja $\mathcal{E} \subset \mathcal{C}^n$, kar implicira obstoj $\partial^n \mathcal{T}$. Potenciranje operatorja popolnega diferenciala označimo z

$$d^n f = \sum_{\forall_{i,..,j}} \frac{\partial^n f}{\partial x_i ... \partial x_j} dx_{i,..,j}, x_i \in V_{i \in \mathbb{I}}$$

Operator lahko razumem kot n-to potenco divergence, kjer vsakemu členu vsote pripišemo enotski vektor.

Opazimo še, da je tudi slika operatorja $\partial^{\oplus}$ ob aplikaciji na $\mathcal{T}$ monoid, od tu dalje nazivan z $\partial^{\oplus}\mathcal{T}$. Potenca operatorja je rekurzivno definirana

$$\partial^n \oplus : \mathcal{T} \to \partial^n \mathcal{T} \bigoplus \partial^{(n-1)} \oplus \mathcal{T} \tag{13}$$

oziroma

$$\partial^n \oplus : \mathcal{T} \to \bigoplus_{0 \le i \le n} \partial^i \mathcal{T} \tag{14}$$

in slika $\mathcal{E} \to \sum_{i=0}^{n} d^i \mathcal{E}$. Rekurzivna definicija odraža strukturo prostora $\subseteq_{\forall_i} \partial^i \oplus \mathcal{T}$.

Izrek 2.2. *Dimenzija prostora $\partial^n \oplus \mathcal{T}$ je $\frac{1-k^{n+1}}{1-k}$, kjer je k dimenzija V.*

Dokaz je trivialen. Spominska kompleksnost modela je očitno eksponentna. Gre poudarit, da v praksi redko projeciramo na celoten V, pogosteje nas zanima

[2] v denimo Bayesianski statistiki, so hiper-parametri parametri a-priori distribucije.

manjša podmnožica komponent. Ob interpretacija programa skozi usmerjen graf, to pomeni, da merimo akcijo specifični oglišč v njem (ki eliminirajo vsa ostala oglišča, iz katerih obstaja pot do oglišča na katerega že projeciramo).

2.3 Prijemi standardne analize

V tej fomrulaciji je stroj M funkcijski prostor, na katerega elemente lahko apliciramo metode funkcionalne analize. Začnimo s preprostejšimi.

Analizo motiviramo s primerom. Predpostavimo časovno kompleksen algoritem A, za katerega iščemo približek. Z opisano mehaniko lahko program lineariziramo. Izberemo $v \in PV$ in preslikamo v $v' \in P'V'$ (kar je potrebno storiti le enkrat). Potem lahko program PV približamo z linearno aproksimacijo $v + (v' - v)(V)$, ki je linearne kompleksnost $\mathcal{O}(k)$, kjer je k dimenzija V. Slika linearizacije služi kot dober približek slike prvotnega programa.

Za nadaljevanje je prikladno vnesti nekaj notacije. Naj tenzor $\mathcal{V}^{ik}_{\mathbb{J}_k} \in \partial^n \oplus V = V_n(5)$ predstavlja element virtualnega prostora. $\mathbb{J}_k$ je nabor indexov, $|\mathbb{J}_k|$ je enak redu odvoda. Index $k = |\mathbb{J}_k|$, zato ga v zapisu izpuščamo.

$$\mathcal{V}^{ik}_{\mathbb{J}_k} = \bigoplus_{0 \leq k \leq n} \mathcal{V}^{ik}_{\mathbb{J}_k} \tag{15}$$

V tej notaciji linearizacija postane

$$\mathcal{V}^i + \mathcal{V}^i_j(v^j) \tag{16}$$

Z indukcijskim korakom, lahko program razvijemo v vrsto. Opazimo naravnost izraza, glede na prostor definiran v (5).

$$\mathcal{V}^i + \mathcal{V}^i_j(v^j) + \frac{1}{2!}\mathcal{V}^i_{jk}(v^{jk}) + \frac{1}{3!}\mathcal{V}^i_{jkl}(v^{jkl}) + ... + \frac{1}{|\mathbb{J}|!}\mathcal{V}^i_{\mathbb{J}}(v^{\mathbb{J}}) \tag{17}$$

kjer se razume, da $v^{ij..n}$ predstavlja $v^i v^j ... v^n$. Podobno lahko definiramo tenzor.

$$V^{\mathbb{J}_k}_k = \bigoplus_{0 \leq k \leq n} \frac{1}{|\mathbb{J}|!} V^{\mathbb{J}_k}_k \tag{18}$$

Potem je razvoj v odrezano vrsto moč izrazit kot srkčitev

$$\mathcal{V}^{ik}_{\mathbb{J}_k} V^{\mathbb{J}_k}_k \tag{19}$$

Neodvisnost (19) od koordinatnega sistema, se v programiranju prevede na neodvisnost v izvajanju. Tako je izraz (19) invarianten stopnji izvajanja programa.

2.4 Geometrija P⊂V

2.5 Optimizacija prostorske zahtevnosti

Ponavadi nas ne zanima odvod po vseh spremenljvkah, ki jih imamo v programu. Označimo z $e_i, \quad i = 1, 2, \ldots n$ standardne bazne vektorje v $\mathbb{R}^n$ in z

$\mathbb{I} \subset \{1, 2, \ldots n\}$ množico indeksov. Indeksi v $\mathbb{I}$ ustrezajo posameznim spremenljivkam (indeksi določajo lokacijo spremenljivke v spominu). Naj bo $V_\mathbb{I}$ vektorski podprostor napet na vektorje $\{e_i; \quad i \in \mathbb{I}\}$. Označimo z $\mathcal{P}_\mathbb{I}$ projekcijo na V_i, z $\mathcal{I}_\mathbb{I}$ pa vložitev $V_\mathbb{I}$ v $\mathbb{R}^n$. Vložitev $\mathcal{I}_\mathbb{I}$ ni mišljena kot linearna preslikava, ampak dopuščamo, da so vrednosti spremenljivk, ki niso v $\mathbb{I}$, poljubne. Recimo, da nas zanima nabor spremenljivk z indeksi $\mathbb{I}$, ki jih program P izračuna na podlagi nabora spremenljivk z indeksi $\mathbb{J}$. Preslikavo med $V_\mathbb{J} \to V_\mathbb{I}$, ki jo določa program P, lahko zapišemo kot

$$P_\mathbb{I}^\mathbb{J} = \mathcal{P}_\mathbb{I} \circ P \circ \mathcal{I}_\mathbb{J} \tag{20}$$

Odvod preslikave $P_\mathbb{I}^\mathbb{J}$ je enak produktu

$$P_\mathbb{I}^\mathbb{J} = P_\mathbb{I} \cdot DP \cdot I_\mathbb{J},$$

kjer sta $P_\mathbb{I}$ in $I_\mathbb{J}$ matriki, ki ustrezata projekciji na $V_\mathbb{I}$ oziroma vložitvi $V_\mathbb{J}$ v $\mathbb{R}^n$.

2.6 Kontrolne strukture

Do sedaj smo se omejili na operacije, ki spreminjajo vsebino spomina. Poleg prireditvenih ukazov, poznamo tudi kontrolne ukaze (npr. stavki `if`, `for`, `while`, ...). Kontrolni stavki ne vplivajo neposredno na vrednost spremenljivk, ampak spreminjajo potek programa. Seveda bo to vplivalo tudi na odvod. Ampak za določen nabor vhodnih spremenljivk, bo potek programa vedno enak. Zato si lahko kontrolne strukture predstavljamo kot definicijo zlepka. Vzemimo naprimer preprost program v programskem jeziku Python

```
def abs(x):
  if x<0:
   return -x
  else
   return x
```

Funkcija `abs(x)` je program, ki izračuna zlepek

$$|x| = \begin{cases} -x; & x < 0 \\ x; & x \geq 0 \end{cases} \tag{21}$$

Vsaka kontrolna struktura razdeli prostor parametrov na različna območja, znotraj katerih je potek programa enak. Celoten program torej razdeli prostor vseh možnih parametrov na končno množico območij $\{\Omega_i; \quad i = 1, \ldots k\}$, kjer je potek programa enak. Program lahko torej v splošnem definiramo kot zlepek. Za $\vec{x} \in \mathbb{R}^n$ je

$$P(\vec{x}) = \begin{cases} P_{n_1 1} \circ P_{(n_1-1)1} \circ \ldots P_{11}(\vec{x}); & \vec{x} \in \Omega_1 \\ P_{n_2 2} \circ P_{(n_2-1)2} \circ \ldots P_{12}(\vec{x}); & \vec{x} \in \Omega_2 \\ \vdots & \vdots \\ P_{n_k k} \circ P_{(n_k-1)k} \circ \ldots P_{1k}(\vec{x}); & \vec{x} \in \Omega_k \end{cases} \tag{22}$$

Linearnizacija TP programa P je seveda tudi odvisna od začetnih parametrov $\vec{x}$ in jo tudi lahko podamo kot zlepek a le v notranjosti deinicijskih območij Ω_i

$$TP_{\vec{x}} = \begin{cases} TP_{n_1 1} \cdot TP_{(n_1-1)1} \cdot \ldots TP_{11}; & \vec{x} \in \text{int}(\Omega_1) \\ TP_{n_2 2} \cdot TP_{(n_2-1)2} \cdot \ldots TP_{12}(\vec{x}); & \vec{x} \in \text{int}(\Omega_2) \\ \vdots & \vdots \\ TP_{n_k k} \cdot TP_{(n_k-1)k} \cdot \ldots TP_{1k}(\vec{x}); & \vec{x} \in \text{int}(\Omega_k) \end{cases} \tag{23}$$

Problem je seveda na robu območju $\partial\Omega_i$, kjer program P ni nujno odvedljiv.

2.7 Direktno odvajanje

2.8 Obratno odvajanje