FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Metoda pohyblivých vážených nejmenších čtverců v Julia
Student:	Tung Anh Vu
Vedoucí:	Ing. Tomáš Kalvoda, Ph.D.
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	Do konce letního semestru 2018/19

Pokyny pro vypracování

1) Seznamte se s problémem rekonstrukce funkcí definovaných v nerovnoměrně rozprostřených bodech (tedy umístěných mimo rovnoměrnou mřížku).
2) Nastudujte numerickou metodu pohyblivých vážených nejmenších čtverců (moving weighted least squares; MWLS).
3) Navrhněte uživatelsky přívětivé rozšíření prostředí Julia o metodu popsanou v bodě 2.
4) Rozšíření implementujte jako open source balíček v jazyce Julia. Kód vybavte jednotkovými testy a řádně dokumentujte.
5) Implementaci prakticky otestujte na problému odhadování derivací takto zadaných funkcí a dále ji porovnejte s dostupnými interpolačními metodami v případě rovnoměrné mřížky.

Seznam odborné literatury

Dodá vedoucí práce.

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 29. prosince 2017

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Metoda pohyblivých vážených nejmenších čtverců v Julia

Tung Anh Vu

Katedra teoretické informatiky
Vedoucí práce: Ing. Tomáš Kalvoda, Ph.D.

15. května 2018

Poděkování

Děkuji svému vedoucímu Ing. Tomáši Kalvodovi, Ph.D., za cenné rady, připomínky a pravidelné konzultace, které mi byly vždy cennou zpětnou vazbou.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 15. května 2018 ……

Odkaz na tuto práci

Vu, Tung Anh. *Metoda pohyblivých vážených nejmenších čtverců v Julia.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

V této práci byla implementována metoda pohyblivých vážených nejmenších čtverců v jazyce Julia. Metoda byla zkoumána jak po teoretické stránce, tak i experimentálně. Byl formulován problém hledání sousedů v daném rozsahu a předvedla se 2 řešení toho problému. Také byl vytvořen krátký popis programovacího jazyka Julia.

Klíčová slova metoda pohyblivých nejmenších čtverců, programovací jazyk Julia, hledání sousedů v dané oblasti, aproximace funkce

Abstract

In this work an implementation of moving weighted least squares method was created. The method was examined both theoretically and practically. The range search problem was formulated and 2 solutions were presented. A short description of Julia programming language can be found as well.

Keywords moving weighted least squares, Julia programming language, range search, function approximation

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Představme si, že je zadána sada dvojic vstupních a výstupních hodnot a naší úlohou je nalézt funkci f, která vztah mezi těmito hodnotami v jistém smyslu popisuje co nejlépe. Nejstarší metodou pro řešení této úlohy je metoda nejmenších čtverců z 19. století, jejíž autorem je A. M. Legendre, který v roce 1805 metodu popsal ve svém díle „Nouvelles Méthodes pour la Détermination des Orbites des Comètes“, ve kterém ji aplikoval na aproximaci tvaru Země [1, s. 12–14].

V práci se budeme věnovat vylepšení, které se nazývá metoda pohyblivých vážených nejmenších čtverců. Jednou z jejich podstatných vlastností je, že nevyžaduje, aby vzorová data byla umístěna na pravidelné mřížce. Metoda pro každý bod $\vec{x}$ vytvoří lokální aproximaci $f(\vec{x})$, která dokáže například interpolovat nad vstupními daty.

Častým problémem ve vědeckých výpočtech a programech vyžadujících vysoký výkon je „problém dvou jazyků“ – prototypování probíhá v dynamických vysokoúrovňových jazycích, jako jsou například Matlab, R nebo Python a výsledná implementace vznikne přepsáním prototypu do výkonnějších, staticky typovaných jazyků, kde častými volbami jsou Fortran nebo C. Autoři jazyku Julia si dávají za cíl vytvořit dynamicky typovaný jazyk, který bude výkonem schopen konkurovat nízkoúrovňovému Fortranu nebo C [2].

Výstupem této práce by měla být implementace metody pohyblivých vážených nejmenších čtverců v programovacím jazyce Julia.

Metoda pohyblivých vážených čtverců teoreticky i prakticky dosahuje velmi dobrých výsledků [3, 4]. V době tvorby práce však neexistovala žádná snadno dostupná implementace metody, a proto jsem se rozhodl práci realizovat.

V teoretické části si zformulujeme problém, který se dá řešit metodou nejmenších čtverců. Představíme metodu pohyblivých vážených nejmenších čtverců. Dále budeme rešerší datových struktur, jimiž lze řešit stěžejní podproblém metody pohyblivých vážených nejmenších čtverců a to *problém hledání dat v zadané oblasti*. V další části posíšeme realizaci a návrh naší knihovny.

V poslední části knihovnu předvedeme na ukázkách a srovnáme ji s metodou aproximace pracující na rovnoměrné mřížce.

Kapitola 1

Teorie

1.1 Základní pojmy a značení

Symbolem $\hat{n}$ značíme množinu čísel $\{1, 2, \ldots, n\}$. Vektory $\vec{v} \in \mathbb{R}^n$ značíme písmenem s šipkou. Implicitně se u vektorových prostorů předpokládá standardní báze. Matice $\mathbb{A}$ značíme zdvojeným písmenem. Chceme-li mluvit o i-tém řádku matice $\mathbb{A}$, pak používáme značení $\mathbb{A}_{i:}$. Podobně pro j-tý sloupec matice $\mathbb{A}$ budeme používat značení $\mathbb{A}_{:j}$. Normu značíme $\|\cdot\|_k$. Není-li spodní index k u normy specifikován, pak se myslí Euklidovská norma $\|\cdot\|_2$. Euklidovskou vzdálenost mezi dvojicí vektorů $\vec{u}$ a $\vec{v}$ tedy značíme $\|\vec{u} - \vec{v}\|$. V následujících kapitolách se používá porovnání mezi vektory $\vec{u} < \vec{v}$, kde $\vec{u}, \vec{v} \in \mathbb{R}^d$, které zavádíme takto: vektor $\vec{u}$ je menší než vektor $\vec{v}$, pokud $\forall i \in \hat{d} : \vec{u_i} < \vec{v_i}$, tedy je menší po složkách. Často budeme používat pro různé účely d-rozměrná pole a indexaci těchto polí budeme provádět vektory z $\mathbb{Z}^d$ nebo $\mathbb{N}_0^d$. Máme-li například 2-rozměrné pole, pak se indexem (1, 2) bude rozumět prvek na prvním řádku a druhém sloupci. Mluvíme-li o souřadnicích, pak zpravidla myslíme souřadnice v kartézském souřadnicovém systému.

1.2 Metoda nejmenších čtverců

Nechť je dána sada dat $\{\vec{x_i}, \vec{y_i}\}_{i=1}^k$, kde $\vec{x_i} \in \mathbb{R}^n$ a $\vec{y_i} \in \mathbb{R}^m$ a sada funkcí $f_1, \ldots, f_\ell$, kde $f_i : \mathbb{R}^n \to \mathbb{R}^m$. Úkolem je nalézt funkci

$$f_c = \sum_{i=1}^{\ell} c_i f_i,$$

s neznámými koeficienty $c_1, \ldots, c_\ell \in \mathbb{R}$, která aproximuje sadu dat co nejlépe ve smyslu nejmenších čtverců s chybovou funkcí

$$J(f_c) := \sum_{i=1}^{k} \|f_c(\vec{x_i}) - \vec{y_i}\|^2 = \sum_{i=1}^{k} \| \sum_{j=1}^{\ell} c_j f_j(\vec{x_i}) - (\vec{y_i})\|^2 = \\ = \sum_{i=1}^{k} \sum_{u=1}^{m} \sum_{j=1}^{\ell} (c_j f_j(\vec{x_i})_u - (\vec{y_i})_u)^2. \tag{1.1}$$

Naší úlohou bude tuto chybovou funkci $J(f_c)$ minimalizovat.

Minimalizační úlohu (1.1) lze řešit pomocí nutné podmínky pro existenci lokálního extrému. Pro obecné $v \in \hat{l}$ platí

$$\frac{\partial}{\partial c_v} J(f_c) = 2 \sum_{i=1}^{k} \sum_{u=1}^{m} \sum_{j=1}^{\ell} \left(c_j (f_j(\vec{x_i}))_u - (\vec{y_i})_u\right)(f_v(\vec{x_i}))_u.$$

Položme tyto derivace rovny nule a dostáváme pro jednotlivá $v \in \hat{\ell}$

$$2 \sum_{i=1}^{k} \sum_{u=1}^{m} \sum_{j=1}^{\ell} (c_j (f_j(\vec{x_i}))_u - (\vec{y_i})_u)(f_v(\vec{x_i}))_u = 0,$$

$$\sum_{i=1}^{k} \sum_{u=1}^{m} \sum_{j=1}^{\ell} c_j (f_j(\vec{x_i}))_u (f_v(\vec{x_i}))_u = \sum_{i=1}^{k} \sum_{u=1}^{m} (\vec{y_i})_u (f_v(\vec{x_i}))_u,$$

$$\sum_{j=1}^{\ell} \underbrace{\left(\sum_{i=1}^{k} \sum_{u=1}^{m} (f_j(\vec{x_i}))_u (f_v(\vec{x_i}))_u \right)}_{\mathbb{A}_{vj}} c_j = \sum_{i=1}^{k} \sum_{u=1}^{m} (\vec{y_i})_u (f_v(\vec{x_i}))_u. \tag{1.2}$$

Matice $\mathbb{A}$ má rozměry $\ell \times \ell$ a prvek na v-tém řádku a v j-tém sloupci je dán předpisem

$$\mathbb{A}_{vj} = \sum_{i=1}^{k} \sum_{u=1}^{m} (f_j(\vec{x_i}))_u (f_v(\vec{x_i}))_u. \tag{1.3}$$

Nyní lze levou stranu rovnice (1.2) zapsat jako $\mathbb{A}\vec{c}$, kde $\vec{c}$ je vektor neznámých koeficientů $\vec{c} = (c_1, \ldots, c_\ell)^T$. Dále zaveďme vektor $\vec{b} = (b_1, \ldots, b_\ell)^T$, kde pro jednotlivá $v \in \hat{\ell}$ platí

$$\vec{b}_v = \sum_{i=1}^{k} \sum_{u=1}^{m} (\vec{y_i})_u (f_v(\vec{x_i}))_u. \tag{1.4}$$

Dosaďme rovnice (1.3) a (1.4) do (1.2) a dostáváme soustavu lineárních rovnic s vektorem neznámých $\vec{c}$

$$\mathbb{A}\vec{c} = \vec{b}. \tag{1.5}$$

Regularitu matice $\mathbb{A}$ lze předpokládat, pokud jsou lineárně nezávislé funkce $f_1, \ldots, f_\ell$ a $\vec{x_1}, \ldots, \vec{x_n}$ jsou po dvojicích různé. Vyřešením soustavy (1.5) získáme hodnoty hledaných koeficientů. Minimalizovaná funkce $J(f_c)$ je součtem kvadrátů, z čehož plyne, že nalezený extrém bude odpovídat minimu.

1.3 Metoda pohyblivých vážených nejmenších čtverců

Metoda pohyblivých vážených nejmenších čtverců patří mezi metody pro rekonstrukci funkce mimo pravidelnou mřížku (angl. *meshfree*). Nevyžadují, aby vstupní (vzorová) data $\vec{x_i}$ byla umístěná na pravidelné mřížce. Také neexistuje žádné omezení pro hodnotu vektoru $\vec{x} \in \mathbb{R}^n$, k němuž chceme sestrojit aproximaci $f(\vec{x})$, což nemusí nutně platit pro metody, které vyžadují po vstupních datech, aby byla umístěná na pravidelné mřížce.

Nechť je dána sada dat $T := \{\vec{x_i}, \vec{y_i}\}_{i=1}^{k}$, kde $\vec{x_i} \in \mathbb{R}^n$ a $\vec{y_i} \in \mathbb{R}^m$, sada funkcí $f_1, \ldots, f_\ell$, kde $f_i : \mathbb{R}^n \to \mathbb{R}^m$ a váhová funkce $\theta : \mathbb{R}_0^+ \to \mathbb{R}_0^+$. Úkolem je nalézt funkci

$$f_c(\vec{x}) = \sum_{i=1}^{\ell} c_i(\vec{x}) f_i(\vec{x}),$$

s neznámými koeficienty $c_1, \ldots, c_\ell \in \mathbb{R}$. Výsledkem obyčejné metody nejmenších čtverců je jedna funkce, která provádí „globální“ aproximaci, zatímco u metody vážených nejmenších čtverců je výsledná funkce závislá na vektoru $\vec{x}$, který chceme aproximovat. Hlavním rozdílem oproti obyčejné metodě nejmenších čtverců je váhová funkce θ, díky níž do chybové funkce „vstupuje“ vektor $\vec{x}$, jehož funkční hodnotu (příp. derivaci) chceme aproximovat. Chybová funkce je dána vztahem

$$\begin{aligned} J(f_c(\vec{x})) &:= \sum_{i=1}^{k} \theta(\|\vec{x} - \vec{x_i}\|) \|f_i(\vec{x_i}) - \vec{y_i}\|^2 = \\ &= \sum_{i=1}^{k} \theta(\|\vec{x} - \vec{x_i}\|) \| \sum_{j=1}^{\ell} c_j(\vec{x}) f_j(\vec{x_i}) - \vec{y_i}\|^2 = \\ &= \sum_{i=1}^{k} \theta(\|\vec{x} - \vec{x_i}\|) \sum_{u=1}^{m} (\sum_{j=1}^{\ell} c_j(\vec{x}) \left(f_j(\vec{x_i})\right)_u - (\vec{y_i})_u)^2. \end{aligned} \tag{1.6}$$

Všimněme si, že pokud je váhová funkce θ konstantní, pak metoda „degraduje“ na obyčejnou metodu nejmenších čtverců. Minimalizační úlohu (1.6) pro libovolné $\vec{x} \in \mathbb{R}^n$ lze řešit pomocí nutné podmínky pro existenci lokálního extrému. Pro obecné $v \in \hat{\ell}$ platí

$$\frac{\partial}{\partial c_v(\vec{x})} J(f_c(\vec{x})) = 2 \sum_{i=1}^{k} \theta(\|\vec{x} - \vec{x_i}\|) \sum_{u=1}^{m} (\sum_{j=1}^{\ell} (c_j(\vec{x}) \left(f_j(\vec{x_i})\right)_u - (\vec{y_i})_u)(f_v(\vec{x_i})_u)).$$

Položme tyto derivace rovny nule a dostáváme pro jednotlivá $v \in \hat{\ell}$

$$2 \sum_{i=1}^{k} \theta(\|\vec{x} - \vec{x_i}\|) \sum_{u=1}^{m} (\sum_{j=1}^{\ell} (c_j(\vec{x}) (f_j(\vec{x_i}))_u - (\vec{y_i})_u)(f_v(\vec{x_i})_u)) = 0.$$

Roznásobením závorky $(c_j(\vec{x})(f_j(\vec{x_i}))_u - (\vec{y_i})_u)$ dostáváme

$$\sum_{i=1}^{k} \theta(\|\vec{x} - \vec{x_i}\|) \sum_{u=1}^{m} (\sum_{j=1}^{\ell} c_j(\vec{x})(f_j(\vec{x_i}))_u (f_v(\vec{x_i}))_u) - (\vec{y_i})_u (f_v(\vec{x_i}))_u = 0.$$

Menšitel $(\vec{y_i})_u (f_v(\vec{x_i}))_u$ nezávisí na sčítacím indexu j, převeďme jej na druhou stranu rovnice.

$$\sum_{i=1}^{k} \theta(\|\vec{x} - \vec{x_i}\|) \sum_{u=1}^{m} (\sum_{j=1}^{\ell} c_j(\vec{x})(f_j(\vec{x_i}))_u (f_v(\vec{x_i}))_u) = \\ = \sum_{i=1}^{k} \theta(\|\vec{x} - \vec{x_i}\|) \sum_{u=1}^{m} (\vec{y_i})_u (f_v(\vec{x_i}))_u.$$

Nyní stačí přeskládat levou stranu rovnice, čímž dostáváme

$$\sum_{j=1}^{\ell} \underbrace{\left(\sum_{i=1}^{k} \theta(\|\vec{x} - \vec{x_i}\|) \sum_{u=1}^{m} (f_j(\vec{x_i}))_u (f_v(\vec{x_i}))_u \right)}_{\mathbb{A}_{vj}} c_j(\vec{x}) = \\ = \sum_{i=1}^{k} \theta(\|\vec{x} - \vec{x_i}\|) \sum_{u=1}^{m} (\vec{y_i})_u (f_v(\vec{x_i}))_u. \tag{1.7}$$

Matice $\mathbb{A}$ má rozměry $\ell \times \ell$, kde prvek na v-tém řádku a v j-tém sloupci je dán vztahem

$$\mathbb{A}_{vj} = \sum_{i=1}^{k} \theta(\|\vec{x} - \vec{x_i}\|) \sum_{u=1}^{m} (f_j(\vec{x_i}))_u (f_v(\vec{x_i}))_u, \tag{1.8}$$

Nyní lze levou stranu rovnice (1.7) psát jako $\mathbb{A}\vec{c}$, kde $\vec{c}$ je vektor neznámých koeficientů $\vec{c} = (c_1, \ldots, c_\ell)^T$. Dále zaveďme vektor $\vec{b} = (b_1, \ldots, b_\ell)^T$, kde pro jednotlivá $v \in \hat{\ell}$ platí

$$\vec{b}_v = \sum_{i=1}^{k} \theta(\|\vec{x} - \vec{x_i}\|) \sum_{u=1}^{m} (\vec{y_i})_u (f_v(\vec{x_i}))_u. \tag{1.9}$$

Dosaďme rovnice (1.8) a (1.9) do (1.7) a dostáváme soustavu lineárních rovnic s vektorem neznámých $\vec{c}$

$$\mathbb{A}\vec{c} = \vec{b}. \tag{1.10}$$

Obecnost váhové funkce nám komplikuje rozhodování o regularitě matice $\mathbb{A}$. Případnou singularitu matice budeme muset řešit zvlášť v implementaci. Vyřešením soustavy (1.10) získáme hodnoty hledaných koeficientů. Minimalizovaná funkce $J(f(\vec{x}))$ je součtem kvadrátů, z čehož plyne, že nalezený extrém bude odpovídat minimu.

Častým případem je situace, kdy výstupní hodnota váhové funkce θ je 0 pro vstupní hodnoty, které jsou větší než pevný parametr $\varepsilon \in \mathbb{R}_0^+$, tzn. $\exists \varepsilon \in \mathbb{R}^+, \forall d > \varepsilon : \theta(d) = 0$. Pro tento případ pro aproximovanou funkci zavádíme speciální značení $f(\vec{x}, \varepsilon)$. V důsledku chybová funkce nabude tvaru

$$J(f(\vec{x}, \varepsilon)) := \sum_{\substack{i \in \hat{k} \\ \|\vec{x} - \vec{x_i}\| \leq \varepsilon}} \theta(\|\vec{x} - \vec{x_i}\|) \|f(\vec{x_i}, \varepsilon) - \vec{y_i}\|^2. \tag{1.11}$$

1.4 Problém hledání sousedů v daném rozsahu

V předchozí části jsme formulovali speciální variantu metody pohyblivých vážených nejmenších čtverců, kde váhová funkce θ nabývá nulových hodnot pro hodnoty větší než pevné ε, jejíž chybová funkce je dána vztahem (1.6). Hledání vektorů $\vec{x_i}$ z množiny vzorových dat, které jsou dostatečně „blízko“ k vektoru $\vec{x}$, jehož hodnotu chceme aproximovat, je podstatnou částí naší implementace a proto bychom chtěli tuto úlohu řešit rychle.

Nechť je dána množina vektorů $S := \{\vec{x_i}\}_{i=1}^n$, kde každé $\vec{x_i} \in \mathbb{R}^d$. Naším úkolem je nalézt pro vektor $\vec{q} \in \mathbb{R}^d$ a $\delta \in \mathbb{R}^+$ podmnožinu $M \subseteq S$ takovou, že pro všechny vektory $\vec{v} \in M$ platí, že $\|\vec{v} - \vec{q}\| \leq \delta$ a pro všechny vektory $\vec{w} \in S \setminus M$ platí, že $\|\vec{w} - \vec{q}\| > \delta$ a tedy $\vec{w} \notin M$.

Tuto úlohu lze řešit naivně algoritmem 1.1, který pro každý vektor $\vec{u} \in S$ zjistí vzdálenost $\|\vec{u} - \vec{q}\|$, a pokud je menší než δ, pak u přidá do výsledné množiny. Za předpokladu, že přidávání do výsledné množiny a počítání vzdálenosti mezi 2 vektory zvládneme v čase $O(1)$, tento algoritmus běží v čase $O(|S|)$ a zabírá $O(|S|)$ paměti.

Algoritmus 1.1 Naivní řešení problému hledání nejbližších sousedů

Vstup: $S \subset \mathbb{R}^n$, $\vec{q} \in \mathbb{R}^n$, $\delta \in \mathbb{R}^+$
Výstup: $M \subseteq S$

```
procedure NaiveSearch(S, q, δ)
    M ← ∅
    for all u ∈ S do
        if ||u − q|| ≤ δ then
            M ← M ∪ u
        end if
    end for
    return M
end procedure
```

V této sekci se budeme zabývat 2 řešeními tohoto problému, které byly použity v implementační části práce. Cílem je zejména snížit časovou složitost při hledání sousedů.

1.4.1 *Cell linked list*

Definice 1: Nechť je dána množina vektorů $S \subset \mathbb{R}^d$. Datová struktura *cell linked list* rozdělí prostor $V := \mathbb{R}^d$ na d-rozměrnou nekonečnou rovnoměrnou mřížku se stranami rovnoběžnými s osami kartézského systému souřadnic a s délkou hrany ε a v každé buňce si udržuje seznam vektorů z S, které se v ní nachází [5, s. 149–152].

Cell linked list lze implementovat pomocí d-rozměrného pole, které si v každé buňce ukládá spojový seznam vektorů, který se v příslušné buňce nachází. *Cell linked list* podporuje následující operace.

Operace	Komentář
CllBuild	Sestavení struktury.
CllSearch	Nalezení sousedů.

Pro *cell linked list* C zavádíme následující pomocné funkce: $\varepsilon(C)$ vrátí délku hrany mřížky, kterou je rozdělen prostor $\mathbb{R}^d$. Pro přistoupení k mřížce *cell linked listu* C používáme $G(C)$. *Cell linked list* budeme implementovat d-rozměrným polem, mluvíme-li o indexech nebo souřadnicích v *cell linked listu*, pak myslíme indexy resp. souřadnice tohoto d-rozměrného pole. Tyto indexy budeme značit pomocí vektorů $\vec{c} \in \mathbb{Z}^d$. Chceme-li tedy vybrat buňku, která se v C nachází na souřadnicích daných vektorem $\vec{c}$, pak použijeme značení $G(C, \vec{c})$. Množinu vektorů S uchovávánou v *cell linked listu* C značíme $S(C)$. Data, která si *cell linked list* uchovává v buňce na indexech $\vec{c}$, značíme $S(C, \vec{c})$.

Pro libovolný vektor $\vec{u} \in \mathbb{R}^d$ a *cell linked list* C se nabízí otázka, do které buňky vektor $\vec{u}$ vložit. Nejpřirozenější volbou je umístit jej do buňky na indexech $(\lfloor \vec{u}_1/\varepsilon(C) \rfloor, \ldots, \lfloor \vec{u}_d/\varepsilon(C) \rfloor)$. Tento výpočet budeme dále značit jako $I(C, \vec{u})$. Může se stát, že se vektor bude umisťovat do buňky se zápornými souřadnicemi. V této sekci povolíme indexaci pole zápornými čísly. Také budeme předpokládat, že pole jsou nekonečná, tzn. lze je indexovat libovolným číslem $i \in \mathbb{Z}$.

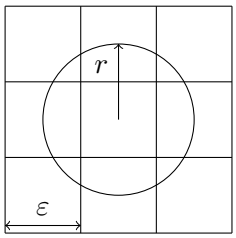
Operace CllBuild umístí zadanou sadu vektorů S do *cell linked listu* C. Pro každý vektor $\vec{u} \in S$ se spočte $\vec{c} = I(C, \vec{u})$ a umístí se do buňky na indexu $\vec{c}$. Tento algoritmus je doplněn pseudokódem 1.2.

Algoritmus 1.2 CllBuild

Vstup: $S \subset \mathbb{R}^d$, $\varepsilon \in \mathbb{R}^+$
Výstup: *cell linked list* C obsahující množinu S

```
C ← prázdný cell linked list
for all u ∈ S do
    c ← I(C, u)
    S(C, c) ← S(C, c) ∪ u
end for
return C
```

Operace CllSearch pro daný vektor $\vec{q} \in \mathbb{R}^d$ a kladnou hodnotu $\delta \in \mathbb{R}^+$ nalezne všechny vektory $\vec{u}$ v množině $S(C)$, jejichž Euklidovská vzdálenost $\|\vec{u} - \vec{q}\|$ je menší než dané δ. Nebude však naivně procházet celou množinu $S(C)$. Místo toho se podívá pouze na vektory v buňkách se souřadnicemi $\vec{a}$, jejichž maximová norma $\|I(C, \vec{q}) - \vec{a}\|_\infty$ je menší nebo rovna $r := \lceil \delta/\varepsilon(C) \rceil$. Intuice za volbou r je taková, že nechceme vybírat takové buňky, které určitě mají prázdný průnik s pomyslnou hypersférou se středem v $\vec{q}$ a poloměrem δ, protože nemohou obsahovat vektory, které jsou dostatečně „blízko“ k vektoru $\vec{q}$. Tento postup je zapsán do pseudokódu 1.3.



Obrázek 1.1: Vizualizace volby r operace CllSearch.

Algoritmus 1.3 CllSearch

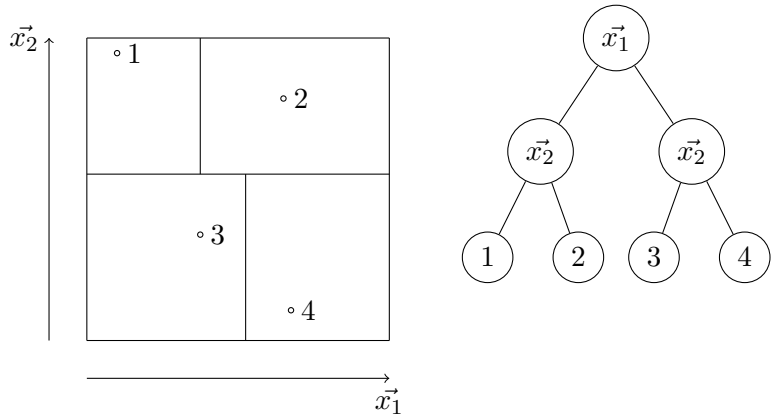
Vstup: $\delta \in \mathbb{R}$, $\vec{q} \in \mathbb{R}^d$, *cell linked list* C
Výstup: $M \subseteq S(G)$

```
r ← ⌈δ/ε(C)⌉
c⃗ ← I(C, q⃗)
M ← ∅
for all m⃗ ∈ ℤ^d : ‖m⃗ − c⃗‖_∞ < r do
    M ← NaiveSearch(S(C, m⃗))
end for
return M
```

Složitost algoritmu CllSearch není snadné určit bez znalosti rozdělení dat $S(C)$, a proto tuto analýzu vynecháváme. Předpokládejme, že chceme pro hledání sousedů v rozsahu je vždy použita vzdálenost $\delta \in \mathbb{R}^+$. Dále zaveďme algoritmus CllAllSearch, který zavolá algoritmus CllSearch z každého vektoru $\vec{u} \in S(C)$. Bentley, Stanat a Williams v [6] ukázali, že pokud je hrana buňky *cell linked listu* je rovna vzdálenosti δ a pokud je rozdělení dat $S(C)$ „řídké“, tzn. pro libovolný vektor $\vec{q} \in \mathbb{R}^d$ platí, že počet vektorů $\vec{x_i} \in S(C)$ jejichž norma je menší nebo rovna δ, je omezen nějakou konstantou $c \in \mathbb{N}$, pak složitost algoritmu CllAllSearch je $O(3^d d(1 + c)N)$, kde N je počet dat uchovávaný v datové struktuře. Odmyslí-li se podmínka řídkosti, pak může složitost algoritmu být i kvadratická vzhledem k počtu dat.

1.4.2 K-d strom

K-d strom[1] je binární vyhledávací strom, který si ve vrcholu V pamatuje diskriminátor $m \in \hat{k}$ a vektory $\vec{B_l} \in \mathbb{R}^k$ a $\vec{B_r} \in \mathbb{R}^k$. Podstrom zakořeněný ve vrcholu V si pamatuje pouze vektory $\vec{u} \in \mathbb{R}^k$, které nejsou menší než $\vec{B_l}$ a nejsou větší než $\vec{B_r}$, tzn. $\vec{B_l} \leq \vec{u} \leq \vec{B_r}$. Každý vrchol V k-d stromu obsahuje tyto 2 vektory a budeme je značit $\vec{B_l}(V)$, resp. $\vec{B_r}(V)$. Levého potomka vrcholu V značíme $L(V)$, pravého $R(V)$. Pokud budeme přistupovat k vektorům uloženým ve vrcholu V, pak píšeme $S(V)$. Diskriminátor vrcholu V budeme značit $m(V)$.



Obrázek 1.2: Ilustrace k-d stromu v prostoru $\mathbb{R}^2$. Ve vnitřních vrcholech jsou hodnoty diskriminátorů. Místo číselné hodnoty diskriminátoru je pro zřejmost uveden vektor „podél“ něhož se dělilo. Čísla v listech udávají označení bodu, který se v něm nachází.

V této části popíšeme pouze ty operace k-d stromu, které jsou v práci použity. Jmenovitě vypouštíme operace přidávání prvku, mazání prvku a hledání k nejbližších sousedů. Popisujeme k-d strom dle [7] a [8].

Konstrukce k-d stromu probíhá rekurzivně. Algoritmus je rozdělen do 2 funkcí, kde funkce KdBuild je pomocnou funkcí volajíc funkci KdBuildRec, která provádí samotné rekurzivní sestavování stromu. Pro konstrukci zcela nového stromu se zavolá funkce KdBuild s parametry S, h, které postupně udávají množinu ukládaných dat a maximální velikost listu. Rekurzivní funkce KdBuildRec pro sestavování stromu má navíc 2 parametry $\vec{B_l} \in \mathbb{R}^k$ a $\vec{B_r} \in \mathbb{R}^k$, které udávají rozmezí pro vektory, které se nachází v právě sestavovaném vrcholu. Funkce KdBuild pouze vrací výsledek volání funkce KdBuildRec, kde hodnotou $\vec{B_l}$ je vektor $(-\infty, \ldots, -\infty)^T$ a hodnotou $\vec{B_r}$ je vektor $(+\infty, \ldots, +\infty)^T$. Nyní lze přejít k popisu rekurzivní funkce KdBuildRec. Je-li velikost množiny S menší než parametr h, pak vracíme list, který

[1] „k“ v názvu značí dimenzi prostoru, v níž se pracuje

obsahuje množinu S a vektory $\vec{B_l}$ a $\vec{B_r}$. Jinak začneme určením diskriminátoru $p \in \hat{k}$ tak, aby se maximalizoval rozdíl vektoru z množiny vstupních dat S s nejmenší p. složkou a vektoru s největší p. složkou. To znamená, že hledáme $p \in \hat{k}$ tak, aby se maximalizovala hodnota $\| \min(S_p) - \max(S_p) \|$, kde S_i značí množinu všech i-tých složek všech záznamů k-d stromu pro $i \in \hat{k}$. Po nalezení p se hledá medián q množiny S_p. Vytvoříme vektory

$$\vec{b_l} = ((\vec{B_r})_1, \ldots, (\vec{B_r})_{p-1}, q, (\vec{B_r})_{p+1}, \ldots, (\vec{B_r})_k)$$

a

$$\vec{b_r} = ((\vec{B_l})_1, \ldots, (\vec{B_l})_{p-1}, q, (\vec{B_l})_{p+1}, \ldots, (\vec{B_l})_k).$$

Rozdělíme množinu S na množiny S_l a S_r, kde $S_l = \{\vec{u} \in S : \vec{u} \leq \vec{b_l}\}$ a $S_r = S \setminus S_l$. Vytvoří se vrchol k-d stromu V, kde $d(V) = p$, $\vec{B_l}(V) = \vec{B_l}$ a $\vec{B_l}(V) = \vec{B_r}$. Levým potomkem vrcholu V bude výsledek volání funkce KdBuildRec s parametry $S_l, h, \vec{B_l}, \vec{b_l}$. Pravým potomkem bude obdobně výsledek volání funkce KdBuildRec s parametry $S_r, h, \vec{b_r}, \vec{B_r}$. Návratovou hodnotou této rekurzivní funkce je vrchol V.

Z koncové podmínky rekurzivního sestavení stromu bude mít výsledný strom $\lceil |S|/h \rceil - 1$ (dále už budeme označovat pouze jako n) listů. Každý vrchol má buď oba potomky, nebo je listem. Kořen má stupeň 2, listy mají stupeň 1 a vnitřní vrcholy mají stupeň 3 (1 hrana z rodiče a 2 do potomků). Označme neznámý počet vnitřních vrcholů jako x. Z principu sudosti a vlastnosti, že počet vrcholů ve stromě je o 1 větší než počet hran, lze sestavit rovnici

$$\begin{aligned}
\sum_{v \in T} \deg(v) &= 2|E|, \\
n + 2 + 3x &= 2(|V| - 1), \\
n + 2 + 3x &= 2|V| - 2, \\
n + 2 + 3x &= 2(1 + n + x) - 2, \\
x &= n - 2,
\end{aligned}$$

z čehož plyne, že sestavený strom bude mít $2n - 1$ vrcholů a paměťová složitost je tedy $O(n)$. Pro lineární hledání mediánu v každém vytváření vrcholu stromu lze předzpracovat data seřazením přes každou složku, což zabere $O(k|S| \log(|S|))$ času. Pro sestavení každého vrcholu se musí projít předaná množina dat S pro nalezení diskriminátoru. Tento rekurentní vztah lze zapsat jako $T(|S|) = 2T(|S|/2) + k|S|$ a z Mistrovské věty lze určit celkový čas pro sestavení stromu jako $O(k|S| \log(|S|))$.

Algoritmus pro hledání všech záznamů v rozsahu také pracuje rekurzivně. Parametry pro hledání jsou vrchol stromu V, který algoritmus zrovna prohledává, vektor $\vec{q}$ a reálné číslo $\delta \in \mathbb{R}$ určující data, která chceme najít. Vhodnou představou je, že $\vec{q}$ a δ určují hyperkouli B a cílem je nalézt všechny záznamy

Algoritmus 1.4 Algoritmus sestavení k-d stromu

Vstup: množina vstupních dat S, maximální velikost listu k-d stromu $h \in \mathbb{N}$
Výstup: kořen k-d stromu T

```
procedure KdBuild(S, h)
    return KdBuildRec(S, h, (−∞, ..., −∞)^T, (∞, ..., ∞)^T)
end procedure
procedure KdBuildRec(S, h, B_l, B_r)
    if |S| < h then
        return KdNode(S, 0, ∅, ∅, (B_l), (B_r))     ▷ 0 značí prázdnou hodnotu
    end if
    p ← −∞
    for all u, v ∈ S, kde u ≠ v do
        for all i ∈ k̂ do                            ▷ dimenze jsou číslovány od 1
            p ← max(p, u_i − v_i)
        end for
    end for
    m ← median({u_p : ∀u ∈ S})
    b_l ← ((B_r)_1, ..., (B_r)_{p−1}, m, (B_r)_{p+1}, ..., (B_r)_k)
    b_r ← ((B_l)_1, ..., (B_l)_{p−1}, m, (B_l)_{p+1}, ..., (B_l)_k)
    s_l ← ∅
    s_r ← ∅
    for all u ∈ S do
        if u ≤ b_l then
            s_l ← s_l ∪ u
        else
            s_r ← s_r ∪ u
        end if
    end for
    return KdNode(S, p, kdBuild(s_l, h, B_l, b_l), kdBuild(s_r, h, b_r, B_r), B_l, B_r)
end procedure
```

v k-d stromu, které jsou „uvnitř“ B. Vektory $\vec{B}_l(V)$ a $\vec{B}_r(V)$ určují pomyslný hyperkvádr H v prostoru $\mathbb{R}^k$, který má hrany rovnoběžné s osami kartézského souřadnicového systému. Algoritmus zjistí, jestli má hyperkvádr H a B neprázdný průnik. Pokud $H \cap B = \emptyset$, pak neexistuje záznam, který by ležel v B a rekurze končí. Pokud $H \cap B \neq \emptyset$, pak v podstromě zakořeněném ve vrcholu V mohou existovat záznamy, které chceme nalézt a rekurze se musí zanořit do obou potomků. Zpravidla se nejprve zanořuje do potomka, jehož hyperkvádr má „větší“ průnik ve smyslu objemu s hyperkoulí B. Dojde-li algoritmus do listu, pak musí zkontrolovat platnost $\|\vec{q} - \vec{u}\| \leq \delta$ pro každý záznam $\vec{u}$, který se v listu nachází. Hledání průniku hyperkvádru a hyperkoule není konstantní operací, proto pokud $H \cap B = H$, pak hledání průniku přeskočíme pro všechny vrcholy podstromu zakořeněném ve vrcholu V a rovnou se po-

díváme na všechny listy tohoto podstromu a provádíme kontrolu z předchozí věty.

Algoritmus 1.5 Algoritmus pro hledání sousedů v daném rozsahu

Vstup: k-d strom T, vektor $\vec{q} \in \mathbb{R}^k$, reálné číslo $\delta \in \mathbb{R}$
Výstup: podmnožina vektorů $M \subseteq S(T)$ v k-d stromu T taková, že $\forall \vec{u} \in M : \|\vec{u} - \vec{q}\| \leq \delta$ a $\forall \vec{v} \in m(T) \setminus S : \|\vec{u} - \vec{q}\| > \delta$

```
procedure KdSearch(kořen k-d stromu V, q ∈ R^k, δ ∈ R)
    if L(V) = ∅ ∧ R(V) = ∅ then
        return NaiveSearch(S(V), q, δ)
    end if
    H_l ← hyperkvádr určený vektory L(B_l) a L(B_r)
    H_r ← hyperkvádr určený vektory R(B_l) a R(B_r)
    B ← hyperkoule se středem v q a poloměrem δ
    R ← ∅
    if H_l ∩ B ≠ ∅ then
        R ← R ∪ KdSearch(L(V), q, δ)
    end if
    if H_r ∩ B ≠ ∅ then
        R ← R ∪ KdSearch(R(V), q, δ)
    end if
    return R
end procedure
```

Složitost algoritmu 1.5 se obtížně určuje a proto tuto analýzu přeskočíme. Lee a Wong v [9] ukázali, že v nejhorším případě je tato složitost $O(kN^{1-1/k})$, kde N je počet záznamů ve stromě a k je dimenze, v níž se pracuje. Důsledkem tohoto výsledku je, že s rostoucí dimenzí se asymptotická složitost přibližuje k naivnímu vyhledávání přes celou množinu dat S.

1.4.3 Další možná řešení

Pro řešení problému hledání sousedů v daném rozsahu existuje řada dalších metod. Datová struktura *quadtree* funguje obdobně jako k-d strom, ale pouze v 2D prostoru. Každé vrchol stromu má 4 potomky, kteří odpovídají rozdělení roviny na 4 části pomocí 2 na sobě kolmých přímek tak, aby v každé části byla přibližně čtvrtina vstupu. Pro obecný k-dimenzionální prostor bude mít každý vrchol 2^k potomků, čímž exponenciálně vzroste počet vrcholů takového stromu. V praxi se například používá *octree*, který pracuje jako quadtree v 3D prostoru, pro práci s grafikou[10]. K-d strom je v podstatě binární vyhledávací strom. Vyhledávací stromy mohou však být i obecné a v jednotlivých vrcholech mohou mít více dat a tím pádem i více potomků. Příkladem takové struktury je *(a, b)-strom* [11, s. 190–198] (občas nazývaný *B-strom*, což je pouze (a, $2a - 1$) popř. (a, $2a$) strom), kde kořen může mít 2 až b potomků a vnitřní

vrcholy mají a až b potomků. Tento koncept se dá modifikovat i pro náš problém a příkladem takové modifikace je *R-strom* [12].

Operace přidávání, odebírání a modifikace prvků v k-d stromě mají slabinu v tom, že jejich provedením může vzniknout nevyvážený strom. Nevyváženosti v binárních vyhledávacích stromech většinou řešíme pomocí rotací vrcholů, čemuž může zabránit nerovnost diskriminátoru rotovaného vrcholu a jeho rodiče. Kvůli tomu by se musel podstrom zakořeněný v rodiči rotovaného vrcholu celý přestavět, což není příliš časově efektivní. Jednou z modifikací k-d stromu, která si s tímto problémem umí poradit je Bkd-strom [13], který mj. pravidelně strom přestavuje.

Kapitola 2

Realizace

2.1 Programovací jazyk Julia

Julia je relativně nový programovací jazyk (poprvé se objevil v roce 2012 [14]) a proto by bylo na místě jej představit a zvýraznit některé důležité, ne zcela obvyklé vlastnosti.

Julia je dynamicky typovaný *just in time* kompilovaný programovací jazyk. Oproti známějším, dynamicky typovaným jazykům jako je například Python[2] nebo Javascript lze u proměnných, třídních proměnných a parametrů funkcí specifikovat typ. Pro tento účel je existuje postfixový operátor `::<název typu>`. Například `a = 20` vytvoří proměnnou `a` s typem `Int64` (64 bitové celé číslo), zatímco `a::Int8 = 20` vytvoří proměnnou `a` s typem `Int8` (8 bitové celé číslo). Specifikace typu se však nikdy explicitně nevyžaduje a vždy je možné ji vynechat.

Nicméně v některých případech je výhodnější typ uvést. U třídních proměnných je vždy lepší typ uvést z výkonnostních důvodů. Pokud typ není specifikován, pak musí být schopen udržet v sobě libovolný objekt a z tohoto důvodu se třídní typy bez uvedeného typu musí alokovat na haldě. Správné používání typů (zejm. typová stabilita, tedy zachování typu proměnné při běhu programu) také může pomoct kompilátoru s optimalizací kódu. Například funkce `f(x) = x < 0 ? 0 : x` je příkladem typově nestabilní funkce, protože vrací `0::Integer`, když podmínka platí, zatímco `x` může mít libovolný typ. Typově stabilní podobou funkce `f` je např. `f(x) = x < 0 ? zero(x) : x`. Tento princip platí i u kontejnerů (pole, spojové seznamy, atd.) a vždy je lepší specifikovat typ ukládaných objektů [15].

Jako ve většině programovacích jazyků je i v Julii možné definovat si vlastní typy.

Složené typy jsou v Julii realizované strukturami, které jsou téměř totožné se strukturami z jazyka C. Tyto struktury a jejich členské proměnné

[2] pomineme-li typové anotace

jsou implicitně *immutable* (česky neměnné). Pokud jsou *immutable* členskými proměnnými složené typy (např. pole), pak lze měnit jejich obsahy, protože *immutable* jsou pouze reference na tyto objekty a ne objekty samotné. Pokud nechceme, aby byla deklarovaná struktura *immutable*, pak použijeme klíčové slovo `mutable` před deklarací struktury. Implicitní neměnnost nemusí být na škodu, protože poskytuje mnoho nezanedbatelných výhod, např. snadnější srozumitelnost kódu a výkonnost plynoucí z toho, že neměnné objekty v některých případech není potřeba alokovat na haldu. Tohoto se využívá v naší implementaci, objekty pro sestrojení aproximace jsou vždy neměnné, ačkoliv v sobě obsahují pole se vstupními daty. Všechny struktury mají implicitní konstruktor s totožným názvem a přijímají tolik argumentů, kolik má struktura členských proměnných a v takovém pořadí, v jakém jsou deklarovány jednotlivé členské proměnné. Konstruktory si lze samozřejmě dodefinovat dle potřeby.

Předávání argumentů funkcím v Julii funguje podobně jako v Javě, vše se předává hodnotou, ale hodnoty neprimitivních proměnných jsou reference. Proměnné samy o sobě žádný typ nemají, jsou pouze „pojmenováním" nějakého objektu. Tedy například posloupnost příkazů

```
a = 3 # typeof(a) -> Int64
a = "Ahoj" # typeof(a) -> String
a = 4.0 # typeof(a) -> Float64
```

žádnou chybu nevyvolá. Pokud se ovšem proměnné určíme typ operátorem `::`, pak už do ní nelze přiřadit hodnotu s jiným typem. Toto je znázorněno na příkladu 2.1. Globálním proměnným (tedy i proměnným v terminálovém prostředí) nelze přiřadit typ, proto se ukázka prováděla ve funkci. Mohlo by nás napadnout, proč se úspěšně podařilo definovat funkci `foo` (indikováno řádkem v příkladu), přestože v sobě obsahuje chybu. Důvodem je to, že definice funkce `foo` neprovádí příkazy v těle funkce. Chybné příkazy se provedly teprve při pokusu o překlad na řádku 7.

```
1  julia> function foo()
2            a::String = "Ahoj"
3            a = 4
4          end
5  foo (generic function with 1 method)
6
7  julia> foo()
8  ERROR: MethodError: Cannot `convert` an object of type
9  Int64 to an object of type String ...
```

Výpis kódu 2.1: Pokus o předefinování proměnné

Objektově orientované paradigma je v Julii realizováno poněkud netradičním způsobem. Abstraktní typy se deklarují klíčovým slovem `abstract`, ne-

mají žádné členské proměnné a slouží pouze jako „vrchol" v grafu typů Julie. Dědit se smí pouze z abstraktních typů. Dá se tedy říct, že v Julii se dědičnost vztahuje pouze na chování objektů a nikoliv na členské proměnné. Účelem je sémanticky svázat chování objektů, jež se mohou lišit implementací, ale vyjadřují tutéž „myšlenku". Vhodným příkladem jsou v Julii například matice. V Julii je mnoho různých objektů reprezentujících matici (všechny dědí ze třídy `AbstractMatrix`), například tridiagonální matice, hermitovská matice, atd. Funkce `transpose(A::AbstractMatrix)` dokáže transponovat libovolný druh matice, protože má pro každý typ matice vlastní metodu. Seznam těchto metod lze získat funkcí `methods(transpose)`. Výběr správné metody za běhu programu na základě typu je docílen mechanismem *multiple dispatch*. Termín *multiple* značí, že se při výběru použité metody uvažují typy všech předaných argumentů.

V práci se pro různé účely často používá pole, které se může chováním lišit od všemožných programovacích jazyků a proto si jej popíšeme. Pole v Julii se indexují od 1. Existuje typ vícerozměrné pole, které nelze zaměňovat s polem polí. Pole v Julii jsou v paměti uložena ve sloupcově majoritním pořadí (stejně jako ve Fortranu). Pragmaticky to například znamená, že pokud se iteruje přes dvourozměrné pole velikosti 3 řádky a 2 sloupce, pak by se mělo nejprve iterovat přes sloupec a poté teprve řádek, tzn. $[1,1], [2,1], [3,1], [1,2], [2,2], [3,2]$, kde první index značí řádek a druhý sloupec.

Julia má vestavěnou funkcionalitu pro vytváření dokumentace. Před definicemi funkcí a strukturami lze vytvořit řetězec, který bude obsahovat dokumentaci k definovanému objektu. Tuto dokumentaci lze v terminálovém prostředí otevřít pomocí příkazu `?<název objektu>`. Tuto vlastnost demonstrujeme na příkladu 2.2. Všimněme si, že stisknutí otazníku změní prompt z `julia>` na `help?>`.

```
1 julia> "This is sample documentation of Foo."struct Foo
     end
2
3 help?> Foo
4 search: ...
5
6   This is sample documentation of Foo.
```

Výpis kódu 2.2: Demonstrace vestavěné dokumentace v Julii.

2.2 Použité externí knihovny

Při implementaci byly použity následující externí knihovny:

- `DynamicPolynomials.jl` [16] a `MultivariatePolynomials.jl` [17] poskytují třídy pro práci s polynomy.

- `NearestNeighbors.jl` [18] poskytuje implementaci k-d stromu.
- `DataStructures.jl` [19] poskytuje spojový seznam.

Při implementaci a testování se používal `Jupyter notebook`[3], který v prohlížeči vytvoří interaktivní prostředí. Terminálové prostředí je nevhodné pro tvorbu rozsáhlejších programů, které se často pouštějí vícekrát, což by znamenalo ruční přepisování a kopírování příkazů při každém provádění. Na druhou stranu modifikovat skripty často znamená, že i při malých změnách se musí znovu provést interpretace celého skriptu. Navíc je obtížné nechat si k jednotlivým příkazům zobrazit jejich dílčí výstupy. `Jupyter` kombinuje oba přístupy a umožňuje snadnou modifikaci skriptů (zvané notebooky), zobrazování výstupu každého příkazu a možnost ukládat si posloupnosti příkazů.

2.3 Implementace *cell linked listu*

V teoretické části se pro reprezentaci mřížky *cell linked listu* použilo nekonečně velké pole, což může být v praxi překážkou. Nechť je dána sada vektorů $S := \{\vec{x_i}\}_{i=1}^{n}$, kde $\vec{x_i} \in \mathbb{R}^d$ a *cell linked list* C s pevně určeným parametrem $\varepsilon(C)$. Chceme určit minimální možnou velikost mřížky $G(C)$ tak, aby v ní bylo možné uchovat všechny vektory z množiny S. Zaveďme značení $\min(V)$ a $\max(V)$, čímž se myslí nejmenší resp. největší[4] vektor z nějaké množiny vektorů V. Je přirozené, aby vektor $\min(S)$ měl souřadnice $(1, \ldots, 1)^T$. Zvolili jsme vektor samých jedniček, protože v Julii se pole indexují od 1. Aby byla měla naše struktura minimální možnou velikost, pak stačí tak velké pole, aby do něj šel umístit vektor $\max(S)$. Operaci pro nalezení buňky k vektoru $\vec{u} \in \mathbb{R}^d$ v *cell linked listu* C jsme zavedli jako $I(C, \vec{u}) := (\lfloor \vec{u}_1/\varepsilon(C) \rfloor, \ldots, \lfloor \vec{u}_d/\varepsilon(C) \rfloor$. Aby souřadnice buňky $\min(S)$ skutečně měla jako souřadnicový vektor $(1, \ldots, 1)^T$, modifikujeme operaci indexace na

$$I_m(C, \vec{u}) = I(C, \vec{u}) - I(C, \min(S(C))) + (1, \ldots, 1)^T. \tag{2.1}$$

Výsledek operace $I_m(C, \max(S))$ je vektorem velikostí $\vec{a}$ vícerozměrného pole, kterým budeme implementovat *cell linked list*. Tzn. $(\vec{a})_i$ udává velikost i-tého rozměru vícerozměrného pole, kterým budeme strukturu implementovat.

Další implementační problém nastal při hledání sousedů. Velikost mřížky $G(C)$ je často o mnoho větší než počet buněk obsahující vektory, které chceme najít. Nechť je dán vektor $\vec{q}$, k němuž chceme nalézt všechny vektory *cell linked listu* C vzdálené maximálně o $\delta \in \mathbb{R}$. Opět si představíme, že vektor $\vec{q}$ určuje střed hyperkoule B s poloměrem δ. Budeme chtít prohledávat pouze buňky, které nemají prázdný průnik s hyperkoulí B. Určení buněk, které mají průnik s B není jednoduché, ale určení buněk s neprázdným průnikem s hyperkrychlí

[3] `https://jupyter.org/`

[4] připomeňme, že pro vektory $\vec{u}, \vec{v} \in \mathbb{R}^n$ platí $\vec{u} < \vec{v}$, pokud je vektor $\vec{u}$ menší po složkách

H opsané hyperkouli B je mnohem snadnější. Minimální souřadnice $\vec{H}_l$ hyperkrychle H určíme jako $\vec{q} - (\lceil \delta/\varepsilon(C) \rceil, \ldots, \lceil \delta/\varepsilon(C) \rceil)^T$ a maximální souřadnice $\vec{H}_r$ hyperkrychle H určíme jako $\vec{q} + (\lceil \delta/\varepsilon(C) \rceil, \ldots, \lceil \delta/\varepsilon(C) \rceil)^T$. Nyní stačí iterovat od $\vec{H}_l$ do $\vec{H}_r$ a pokud má aktuální buňka neprázdný průnik s B, pak v ní spustíme NaiveSearch a její výsledek přidáme do vektoru výsledků. Tento postup je popsán pseudokódem 2.1.

Algoritmus 2.1 Hledání sousedů v *cell linked listu*

Vstup: *cell linked list* C, $q \in \mathbb{R}^d, \delta \in \mathbb{R}^+$
Výstup: $M \subseteq S(C)$, kde $\forall \vec{u} \in M : \|\vec{u} - \vec{q}\| \leq \delta$ a $\forall \vec{v} \in S(C) \setminus M : \|\vec{v} - \vec{q}\| > \delta$

```
M ← ∅
r ← ⌈δ/ε(C)⌉
c⃗ ← I_m(C, q)
H⃗_l ← c⃗ − (r, ..., r)^T
H⃗_r ← c⃗ + (r, ..., r)^T
for i⃗ ← H⃗_l, H⃗_r do
    if ‖i⃗ − c⃗‖ ≤ r then
        M ← M ∪ NaiveSearch(S(G(C, i⃗)), q⃗, δ)
    end if
end for
return M
```

2.4 Implementace metody pohyblivých vážených nejmenších čtverců

V této části budeme používat totožné značení jako v části 1.3, kde se metoda popisovala teoreticky. Implementace je téměř doslovným přepsáním vzorců (1.8), (1.9) a (1.10). Proto se v popisu implementace zaměříme na ty části, které se liší výraznějším způsobem od teoretického popisu.

Těžištěm implementace je struktura `MwlsObject`, která si uchovává potřebné vstupy pro výpočet požadovaných aproximací. Mezi jeho členské proměnné patří

- `inputs` – sada vstupních dat $\{\vec{x_i}\}_{i=1}^k, \vec{x_i} \in \mathbb{R}^n$,
- `outputs` – sada výstupních dat $\{\vec{y_i}\}_{i=1}^k, \vec{y_i} \in \mathbb{R}^m$,
- `EPS` – reálná hodnota udávající implicitní velikost „okolí“ pro konstrukci aproximace
- `weightFunc` – váhová funkce θ pro konstrukci aproximace.

Členské proměnné `inputs` a `outputs` lze předat dvěma způsoby, buď jako dvě různá vícerozměrná pole, nebo jedním vícerozměrným polem a kladným

celočíselným parametrem, které udává dimenzi výstupních dat. Ačkoliv si Julia uchovává vícerozměrná pole ve sloupcové majoritě, tak se na vstupu očekávají data tak, že jednotlivé dvojice vstupů a výstupů jsou na jednom řádku.

Třída `MwlsObject` má 3 různé implementace, které se liší způsobem hledání sousedů v rozsahu. Struktura `MwlsNaiveObject` používá k hledání naivní iteraci přes celou množinu vstupních dat, `MwlsCllObject` používá *cell linked list* a `MwlsKdObject` používá k-d strom. Pro snadnější vytvoření jednotlivých struktur jsou připraveny pomocné funkce `mwlsNaive`, `mwlsCll` a `mwlsKd`.

Prvním rozdílem mezi implementací a teoretickým popisem jsou funkce použité k aproximaci. V teoretické části jsme pro aproximaci použili libovolné funkce $f_1, \ldots, f_\ell$. V implementaci se pro aproximaci vždy volí polynomy (přesněji řečeno monomy), což se zpravidla provádí i v praxi. Maximální povolený stupeň polynomu uživatel předává při tvorbě aproximačního objektu `MwlsObject`. Tento parametr lze vynechat, v tom případě se pro aproximaci volí polynomy stupně 2. Pro upřesnění uveďme několik příkladů:

- pro dimenzi prostoru vstupních hodnot 3, dimenzi prostoru výstupních hodnot 1 a maximální stupeň polynomu 1 se pro tvorbu aproximace zvolí polynomy $1, x_1, x_2, x_3$,
- pro dimenzi prostoru vstupních hodnot 2, dimenzi prostoru výstupních hodnot 1 a maximální stupeň polynomu 2 se pro tvorbu aproximace zvolí polynomy $1, x_1, x_2, x_1^2, x_1x_2, x_2^2$,
- pro dimenzi prostoru vstupních hodnot 1, dimenzi prostoru výstupních hodnot 2 a maximální stupeň polynomu 2 se pro tvorbu aproximace zvolí funkce $(1,0), (0,1), (x,0), (0,x), (x^2,0), (0,x^2)$.[5]

Algoritmus pro generování těchto polynomů je popsán pseudokódem 2.2. V obecném případě bude velikost výsledné množiny M algoritmu 2.2 $\binom{m+d}{m}$ (viz [20]).

Zaveďme matici $\mathbb{Y} \in \mathbb{R}^{k,m}$, kde $\mathbb{Y}_{ij} := (\vec{y_i})_k$, tzn. v i-tém řádku a j-tém sloupci matice je j-tá složka i-té výstupní hodnoty. Má-li množina vzorových dat vícerozměrné výstupní hodnoty, pak se množina M použije pro sestrojení aproximací pro v každé složce zvlášť, tedy zvlášť se aproximují jednotlivé vektory výstupních hodnot $\mathbb{Y}_{:1}, \ldots, \mathbb{Y}_{:m}$. To efektivně znamená, že funkce použité k aproximaci jsou $\vec{g}_{1,1}, \ldots, \vec{g}_{l,m}$, kde funkce $\vec{g}_{i,j} : \mathbb{R}^n \to \mathbb{R}^m$ je ve tvaru

$$\begin{cases} (\vec{g}_{i,j})_k = 0, & \text{pro } k \neq j \\ (\vec{g}_{i,j})_k = f_i, & \text{pro } k = j. \end{cases} \tag{2.2}$$

Nyní můžeme přejít k samotnému hledání neznámých koeficientů. Jednou možností je sestavit soustavu rovnic (1.7) pro jednotlivé jednorozměrné vektory výstupních hodnot $\mathbb{Y}_{:1}, \ldots, \mathbb{Y}_{:m}$. Ze vztahu (1.8) jde vidět, že výpočet levé strany závisí pouze na vzorových vstupních hodnotách. Proto vyřešíme

[5] Vysvětlení bude později.

Algoritmus 2.2 Algoritmus pro generování polynomů pro aproximaci maximálního stupně d

Vstup: dimenze prostoru výstupních hodnot m, stupeň generovaného polynomu d
Výstup: polynomy použité k aproximaci

```
M ← 1                                          ▷ konstantní polynom
for all v ∈ {x_1, ..., x_m} do
    R ← ∅
    for all T ∈ M do
        for all d' ∈ d̂ do
            p ← T · v^(d')
            if deg(p) > d then
                break                          ▷ polynom p má stupeň větší než d
            else
                R ← R ∪ p
            end if
        end for
    end for
    M ← M ∪ R
end for
return M
```

všechny soustavy najednou. Zavedeme matici $\mathbb{V}$, kde v j-tém sloupci chceme pravou stranu (1.9) pro vektor výstupních hodnot $\mathbb{Y}_{:j}$. Prvky matice $\mathbb{V}$ tedy napočítáme vztahem $\mathbb{V}_{vj} := \sum_{i=1}^{k} \mathbb{Y}_{ij} f_v(\vec{x_i})$. Jelikož výstupní hodnoty jsou nyní skaláry, tak kvadrát normy v původním vzorci jsme nahradili pouhým součinem. K matici pravé strany budeme přirozeně chtít i matici neznámých koeficientů. Zaveďme matici $\mathbb{U}$, kde $\mathbb{U}_{ij}$ je neznámý koeficient funkce f_i pro aproximaci výstupních dat $\mathbb{Y}_{:j}$. Vyřešením soustavy $\mathbb{A}\mathbb{U} = \mathbb{V}$ získáme hledané koeficienty. Připomeňme, že matice $\mathbb{A}$ je dána vztahem (1.8).

Samotné řešení soustavy se provádí v Julii operátorem \, který zavolá příslušnou metodu z knihovny `LAPACK` nebo `openBLAS`. Občas se však může stát z důvodu numerických nepřesností, že matice $\mathbb{A}$ bude „téměř singulární“ (v anglické literatuře se pro tento jev používá pojem *ill-conditioned matrix*). Proto bychom chtěli detekovat „singularitu“ takové matice dříve, než se vyhodí `SingularException`. Determinant matice není vhodným způsobem, jak numericky získat informaci o singularitě matice. Výpočet determinantu je také ovlivněn numerickými nepřesnostmi a rozhodnutí o singularitě bychom často prováděli na základě determinantu, jehož spočtená hodnota nemusí být přesná nula. Proto použijeme *podmíněnost* matice (angl. *condition number*). Tuto funkci poskytuje standardní knihovna jazyka Julia v podobě funkce `cond`. Usoudíme-li z podmíněnosti, že matice $\mathbb{A}$ je singulární, pak výsledné koeficienty určíme jako nulové.

Implementace poskytuje i funkci `mwlsDiff`, která umí aproximovat derivace funkcí. Argumentem funkce je bod, ve kterém chceme funkci aproximovat a n-tice[6], která značí, kolikrát se v jakém směru derivuje. Například pro $\vec{v} \in \mathbb{R}^2$ dvojice $(1, 2)$ značí, že chceme spočítat $\frac{\partial}{\partial x_1 \partial x_2^2} f(\vec{x})$. Aproximovaná funkce f je ve tvaru $\sum_{i=1}^{\ell} c_i f_i$, pro spočtení derivace akorát zderivujeme funkci f a tuto zderivovanou funkci necháme vyhodnotit v $\vec{v}$. Poznamenejme, že pokud maximální povolený stupeň polynomu použitého k aproximaci není větší nebo roven počtům derivací ve jednotlivých směrech, pak výsledek aproximace derivace bude vždy nulový. Například pokud chceme první derivaci podle $\vec{x_1}$, třetí derivaci podle $\vec{x_2}$ a pro aproximaci použijeme polynomy $f_1(x) = 1, f_2(x) = x$, pak výsledná aproximace derivace bude 0.

[6] připomeňme, že n je dimenze prostoru vstupních dat

Kapitola 3

Experimenty a testování

Všechny testy byly prováděny na procesoru `Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz`. Byla použita verze Julie 0.6. Pro vykreslování grafů se používá metabalíček `Plots.jl` [21], který poskytuje frontend pro velké množství backendů na vykreslování grafů. Jako backend pro `Plots.jl` jsme zvolili balíček `PyPlot.jl` [22].

3.1 Obecný postup experimentu

Pro každý experiment se zvolí funkce, jež se bude aproximovat. Volené aproximované funkce většinou nebudou mít žádný speciální význam a jsou voleny náhodně (ve smyslu libovolně). Budeme vytvářet 2 sady vzorových vstupních dat, jednu umístíme na rovnoměrnou mřížku a druhá bude vytvořena „vychýlením“ vstupních hodnot umístěných na rovnoměrné mřížce. Příkladem může být dvojice sad $\{-2, -1, 0, 1, 2\}$ a $\{-1.9, -1, 0.14, 0.98, 2.1\}$. Samotné vychýlení provedeme přičtením vhodně malých hodnot ke každé složce každého vektoru vstupních dat. Pro řešení problému hledání sousedů v daném rozsahu budeme střídavě používat k-d strom a *cell linked list*. Výsledky se nebudou lišit, střídání slouží pouze jako ukázka funkčnosti obou postupů. Vykreslíme si grafy absolutních chyb těchto výpočtů. Naši implementaci srovnáme s aproximační metodou pracující na rovnoměrné mřížce z balíčku `Interpolations.jl` [23]. Také bychom chtěli kvantifikovat chybu výsledných aproximací. Nechť je $f(\vec{x})$ funkce, kterou chceme aproximovat, a $g(\vec{x})$ je výsledná aproximace. Míru nepřesnosti zavedeme jako

$$\frac{1}{|D|} \int_D \|f(\vec{x}) - g(\vec{x})\| \, d\vec{x}_1 \dots d\vec{x}_k,$$

kde D je interval, na kterém byla prováděna aproximace. Tuto funkci lze interpretovat jako průměrnou chybu aproximace. Určitý integrál bude počítán numericky, k čemuž slouží balíček `Cubature.jl` [24]. Celý postup popsaný v tomto odstavci provedeme i pro aproximaci derivací funkcí.

3.2 Váhové funkce

V této části představíme několik váhových funkcí $\theta(d) : \mathbb{R}_0^+ \to \mathbb{R}_0^+$, které lze použít k aproximaci. Stále bude platit, že od nějakého $\varepsilon \in \mathbb{R}_0^+$ bude hodnotou těchto funkcí 0. Označme $d = \|\vec{x} - \vec{x_i}\|$ jako vzdálenost mezi vektorem $\vec{x}$, k němuž hledáme aproximaci, a libovolným vzorovým vstupním vektorem $\vec{x_i}$. Lancaster a Salkauskas v [3] volí ve svých experimentech funkce $\theta(d) = 1/d^2$ a $\theta(d) = 1/d^4$. Nealen v [20] rozšiřuje tyto funkce do podob $\theta(d) = 1/(d^2 + h^2)$ a $\theta(d) = 1/(d^4 + h^4)$ pro nějaké $h \in \mathbb{R}^+$. Nastavením h blízko k 0 při tvorbě aproximace v místě existujícího vzorového vstupního data vyjde téměř nekonečná váha a dojde k interpolaci v tomto místě. Levin ve svých experimentech v [4] došel k závěru, že se nejvíce osvědčila váhová funkce $\theta(d) = e^{-d^2/h^2}$, kde $h \in \mathbb{R}^+$ je průměrná vzdálenost mezi všemi dvojicemi vstupních vzorových hodnot. Tyto funkce mají společné to, že jsou klesající na intervalu $(0, +\infty)$. Tím je vyjádřena myšlenka, že čím vzdáleněji je vzorový vektor $\vec{x_i}$ od vektoru $\vec{x}$, tím méně ovlivňuje výslednou aproximovanou hodnotu $g(\vec{x})$. Parametry h slouží k „vyhlazení“ výsledných aproximací.

3.3 Aproximace funkce jedné proměnné s jednorozměrným výstupem

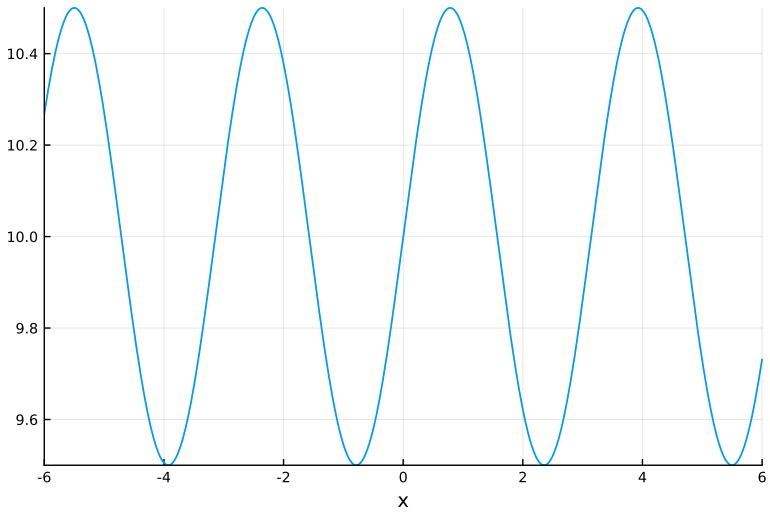
Nejprve si ukážeme funkčnost našeho balíčku na aproximaci funkce $g(x) = \sin(2x)/2 + 10$. Jako vzorovou množinu vstupních dat zvolíme hodnoty od -6 do 6 s rozestupem 0.2, tzn. $x = \{-6, -5.8, \ldots, 5.8, 6\}$. Pro určení aproximace volíme polynom jedné proměnné maximálního stupně 2, tedy funkce $f_1(x) = 1, f_2(x) = x, f_3(x) = x^2$. Maximální povolenou chybu pro výpočet míry nepřenosti nastavíme na 10^{-8}. Nastavíme parametr ε na 0.6 a jako váhovou funkci volíme

$$\theta(d) = \begin{cases} e^{\left(\frac{d^2}{0.2^2}\right)}, & \text{pro } d \leq \varepsilon \\ 0, & \text{pro } d > \varepsilon. \end{cases}$$

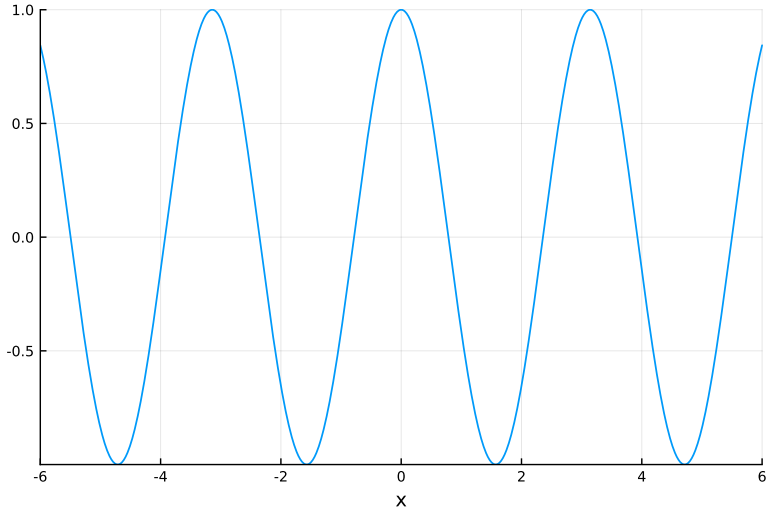
Graf funkce $g(x)$ je na obrázku 3.1, graf první derivace $g'(x)$ je na obrázku 3.2.

Graf absolutních chyb naší metody je na obrázku 3.3. Lze si všimnout, že naše metoda výrazněji chybuje v místech, kde má původní funkce lokální extrémy. Graf absolutních chyb metodou křívek B-splines je na obrázku 3.4. V tabulce 3.1 jsou vypočtené míry nepřesnosti jednotlivých metod. Obě metody výrazně chybovaly na krajích intervalu, na kterém se aproximace prováděla. Jsou-li vstupní data umístěna na rovnoměrné mřížce, pak aproximace pomocí B-spline křivek dosahuje lepších výsledků.

Nyní vychýlíme vstupní data. Ke každé vstupní hodnotě přičteme náhodné číslo v rozsahu $[-0.1, 0.1]$ a přepočítáme příslušné výstupní hodnoty. Graf absolutních chyb naší metody je na obrázku 3.5. Graf absolutních chyb metody



Obrázek 3.1: Graf funkce $g(x) = \sin(2x)/2 + 10$.

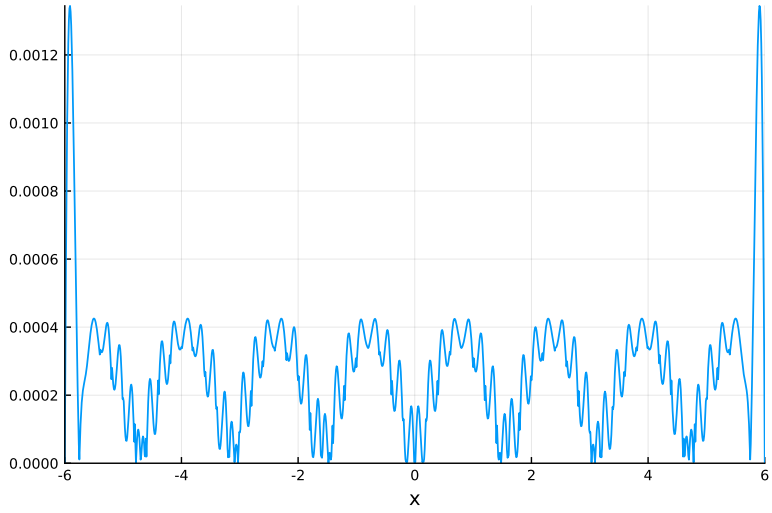


Obrázek 3.2: Graf derivace $g'(x) = \cos(2x)$.

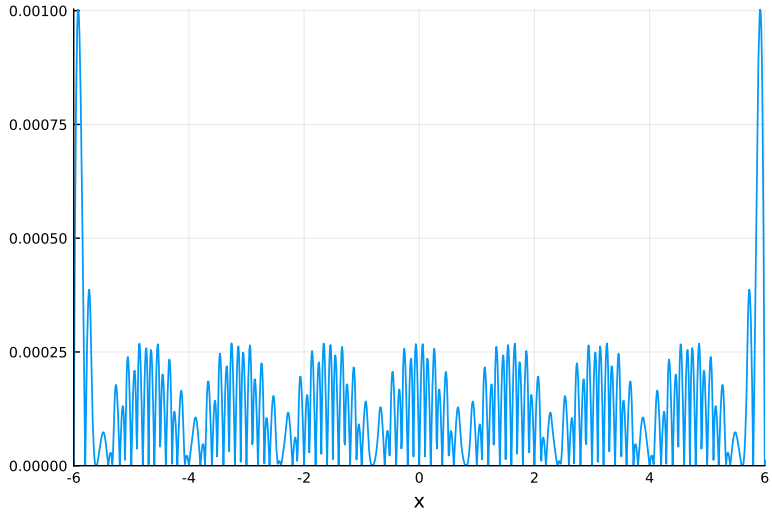
Metoda	Míra nepřesnosti
pohyblivé vážené nejmenší čtverce	0.0002739416389474902
křivky B-splines	0.0001339603334516833

Tabulka 3.1: Míry nepřesnosti aproximace funkce $g(x) = \sin(2x)/2 + 10$.

křivek B-splines je na obrázku 3.6. V tabulce 3.2 jsou vypočtené míry nepřesnosti jednotlivých metod. Přesnosti obou přístupů se zhoršily kvůli nerovnoměrnému rozdělení dat. Naše metoda sice pravidelné rozdělení nevyžaduje,



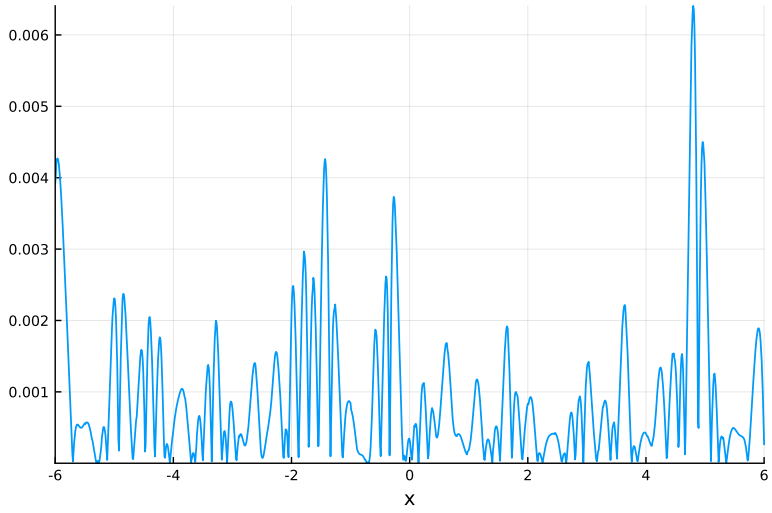
Obrázek 3.3: Absolutní chyby aproximace $g(x) = \sin(2x)/2 + 10$ pomocí k-d stromu.



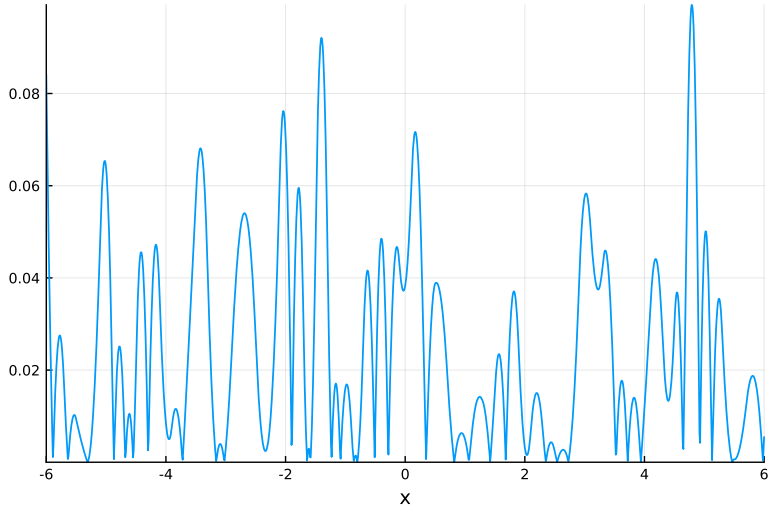
Obrázek 3.4: Absolutní chyby aproximace $g(x) = \sin(2x)/2+10$ pomocí křivek B-splines.

ale v místech s nižší přesností je důsledkem náhodného vychýlení méně vzorových dat v „okolí“ pro určení správné hodnoty aproximace. Jde vidět, že naše metoda dosahuje lepších výsledků v případě nepravidelného rozdělení dat.

Na stejných datech a se stejnými parametry metod proveďme experiment pro aproximace derivací. Graf absolutních chyb aproximace derivace naší metodou je na obrázku 3.7. Graf absolutních chyb aproximace metodou křivek B-splines je na obrázku 3.8. V tabulce 3.3 jsou vypočtené míry nepřesnosti



Obrázek 3.5: Absolutní chyby aproximace funkce $g(x) = \sin(2x)/ + 10$ s vychýlenými vstupními hodnotami pomocí *cell linked listu*.

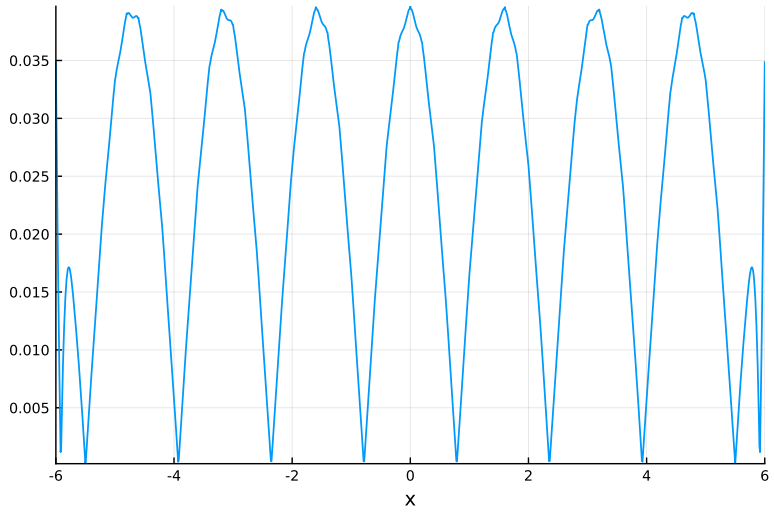


Obrázek 3.6: Absolutní chyby aproximace funkce $g(x) = \sin(2x)/ + 10$ s vychýlenými vstupními hodnotami pomocí křivek B-splines.

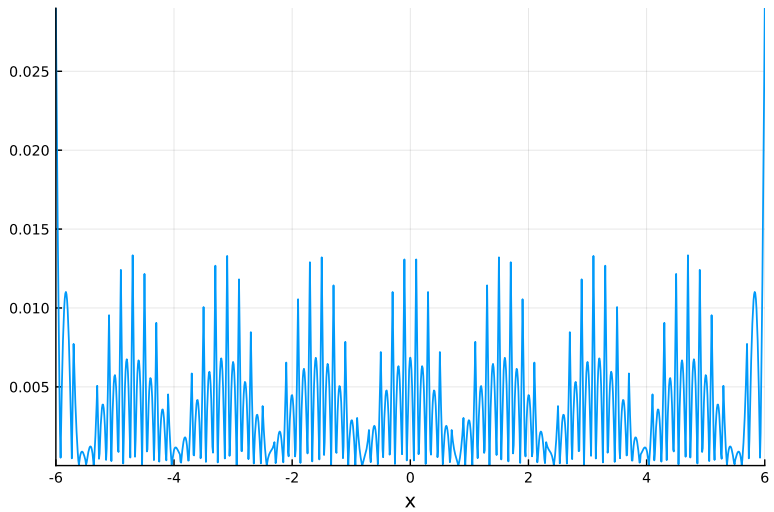
Metoda	Míra nepřesnosti
pohyblivé vážené nejmenší čtverce	0.0009406554285222674
křivky B-splines	0.024718920103864

Tabulka 3.2: Míry nepřesnosti aproximace funkce $g(x) = \sin(2x)/2 + 10$ s vychýlenými vstupními daty.

jednotlivých metod. Lze si opět všimnout, že obě metody výrazně chybují v lokálních extrémech aproximované derivace a největší chyby jsou na kraji aproximovaného intervalu. Jsou-li vstupní hodnoty pravidelně rozložené, pak i pro aproximaci derivace je lepší metoda křivek B-spline.



Obrázek 3.7: Absolutní chyby aproximace funkce $g'(x) = \cos(2x)$ pomocí *cell linked listu*.



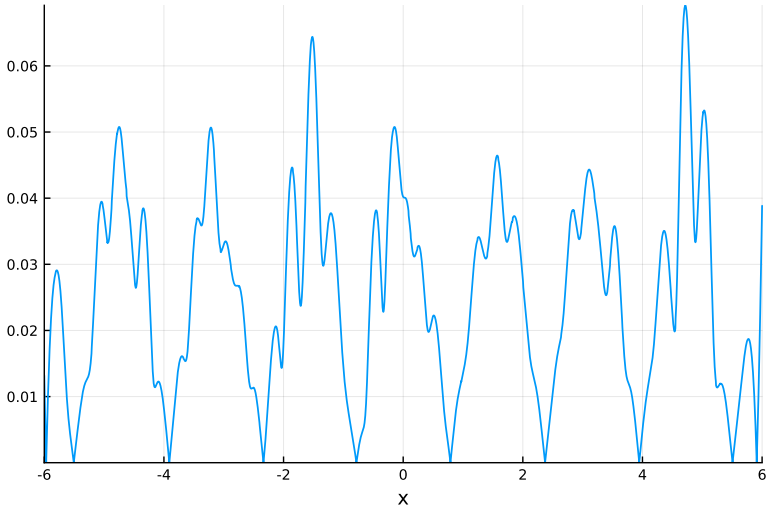
Obrázek 3.8: Absolutní chyby aproximace funkce $g'(x) = \cos(2x)$ pomocí křivek B-splines.

Nakonec proveďme aproximaci funkce $g'(x) = cos(2x)$ s vychýlenými vstupními daty. Graf absolutních chyb aproximace derivace naší metodou je na

Metoda	Míra nepřesnosti
pohyblivé vážené nejmenší čtverce	0.023780149818494 25
křivky B-splines	0.003587242922601426

Tabulka 3.3: Míry nepřesnosti aproximace funkce $g'(x) = \cos(2x)$.

obrázku 3.9. Graf absolutních chyb aproximace derivace metodou křivek B-splines na obrázku 3.10. V tabulce 3.4 jsou vypočtené míry nepřesnosti jednotlivých metod. U naší metody se přesnost nepatrně snížila, nepřesnost metody křivek B-splines se 100 krát zvětšila. V případě nerovnoměrného rozdělení dat opět vítězí naše metoda.



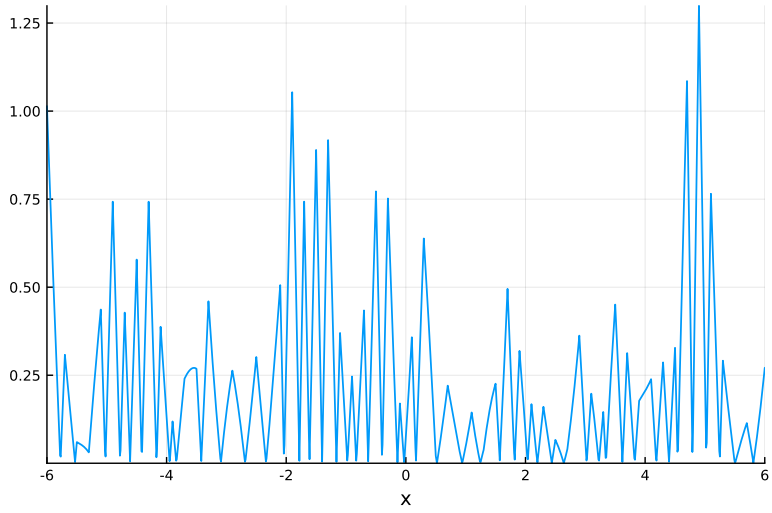
Obrázek 3.9: Absolutní chyby aproximace funkce $g'(x) = \cos(2x)$ s vychýlenými vstupními daty pomocí k-d stromu.

Metoda	Míra nepřesnosti
pohyblivé vážené nejmenší čtverce	0.025508261815647176
křivky B-splines	0.22589729352988716

Tabulka 3.4: Míry nepřesnosti aproximace funkce $g'(x) = \cos(2x)$ s vychýlenými vstupními daty.

3.3.1 Časové měření

Budeme měřit dobu sestrojení aproximace v každém vstupním bodu. Zvyšování velikosti vstupu budeme provádět zvyšováním hustoty vstupních dat na intervalu $(-6, 6)$. Se zvětšující se hustotou dat také budeme zmenšovat para-



Obrázek 3.10: Absolutní chyby aproximace funkce $g'(x) = \cos(2x)$ s vychýlenými vstupními daty pomocí křivek B-splines.

metr ε u váhové funkce, aby se pro sestavování aproximace vždy používalo stejné množství dat. Naměřené hodnoty jsou součtem časů 10 provedení aproximace v každé vstupní hodnotě. Všechny časové údaje jsou ve vteřinách. Doby běhu pro verzi s vychýlenými vstupními hodnotami jsou srovnatelné s dobami běhu pro pravidelně rozmístěné vstupní hodnoty a proto je vynecháváme. Naměřené doby běhu pro tvorbu aproximací jsou v tabulce 3.5 a doby běhu pro tvorbu aproximací derivací jsou v tabulce 3.6. Z pohledu časového je naše metoda vždy horší, než metoda křivek B-splines.

# dat	naivní prohledávání	k-d strom	*cell linked list*	křivka B-spline
11	0.082150	0.079543	0.116387	0.000024
51	0.518809	0.329571	0.545994	0.000045
101	1.603859	1.062111	1.412573	0.000049
501	19.140491	2.569747	5.102424	0.000195
1001	65.633024	4.872680	9.766312	0.000297
10001	> 600	52.379704	97.523626	0.004166

Tabulka 3.5: Součty 10 dob běhu aproximace funkce $g(x) = \sin(2x)/2 + 10$ s pevnou šířkou intervalu vzorových dat $(-6, 6)$.

# dat	naivní prohledávání	k-d strom	*cell linked list*	křivka B-spline
11	0.087109	0.062133	0.107071	0.000035
51	0.504414	0.335378	0.515064	0.000074
101	1.845466	0.968111	1.386275	0.000131
501	17.730011	2.621706	4.921474	0.000469
1001	65.762049	6.253431	9.892969	0.001092
10001	> 600	52.130899	98.982541	0.012220

Tabulka 3.6: Součty 10 dob běhu aproximace derivace $g'(x) = \cos(2x)/2$ s pevnou šířkou intervalu vzorových dat $(-6, 6)$.

3.4 Aproximace funkce dvou proměnných s jednorozměrným výstupem

V této části budeme aproximovat funkci $g(x, y) = e^{(-(x^2+y^2))}$. Nechť je $I := \{-1.95, -1.75, \ldots, 1.95\}$. Jako vzorovou množinu vstupních dat zvolíme množinu dat $I \times I$. Pro určení aproximace volíme polynomy dvou proměnných maximálního stupně stupně 4. Maximální povolenou chybu pro výpočet míry nepřesnosti nastavíme na 10^{-1}. Důvodem je to, že pro nižší tolerance se výpočet integrálu nezastavil[7]. Parametr ε nastavíme na 0.625. Jako váhovou funkci zvolíme

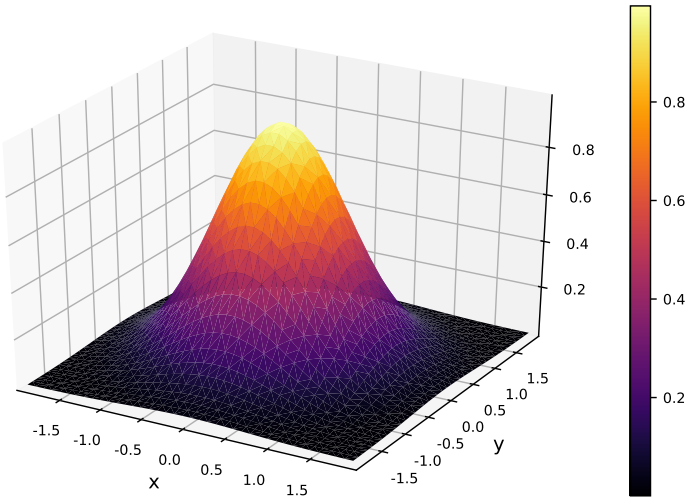
$$\theta(d) = \begin{cases} e^{\frac{d^2}{0.28^2}}, & \text{pro } d \leq \varepsilon \\ 0, & \text{pro } d > \varepsilon. \end{cases}$$

Graf funkce $g(x, y)$ je na obrázku 3.11, graf derivace $\frac{\partial}{\partial x \partial y} g(x, y) = 4xye^{(-x^2+y^2)}$ je na obrázku 3.12.

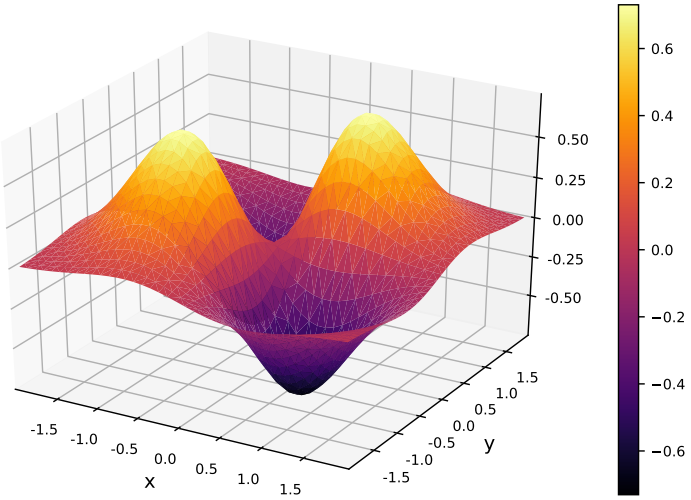
Graf absolutních chyb aproximace funkce $g(x, y)$ naší metodou je na obrázku 3.13. Nejvíce se chybovalo na „krajích“ intervalu. To bude nejspíše tím, že je v těchto místech menší množství dat pro sestrojení aproximace. Graf absolutních chyb aproximace funkce $g(x, y)$ metodou ploch B-splines je na obrázku 3.14. Největší chyby jsou ve stejných místech jako při aproximaci naší metodou. V místě lokálního extrému je oproti minulé sekci větší přesnost. Důvodem může být to, že pro tvorbu aproximace máme k dispozici více dat a navíc jsme použili pro aproximaci polynomy vyššího stupně. V tabulce 3.7 jsou vypočtené míry nepřesnosti pro obě metody. I ve vícerozměrném případě jsou plochy B-splines lepší.

Po zbytek této části už metody ploch B-splines nebudou použity. Prvním důvodem je, že knihovna `Interpolations.jl` má v době tvorby práce nesprávnou a nefunkční implementaci druhé derivace. Druhým důvodem je, že tato metoda dokáže pracovat s nepravidelně umístěnými vstupními hodnotami pouze v prostoru $\mathbb{R}$.

[7] do 10 minut



Obrázek 3.11: Graf funkce $g(x, y) = e^{-(x^2+y^2)}$.

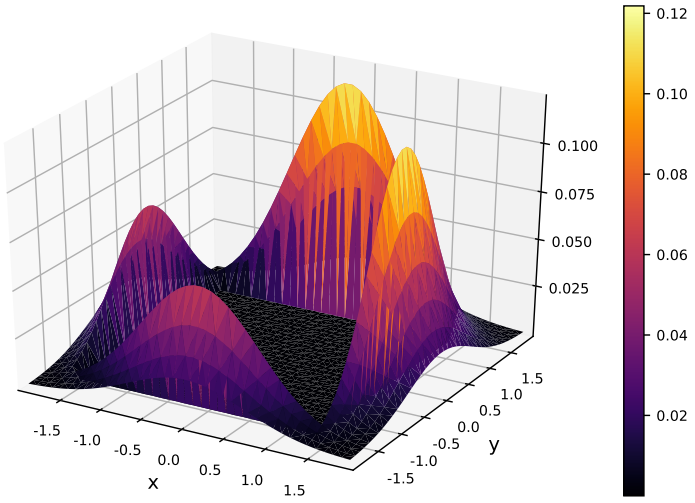


Obrázek 3.12: Graf funkce $\frac{\partial}{\partial x \partial y} g(x, y) = 4xye^{-(x^2+y^2)}$.

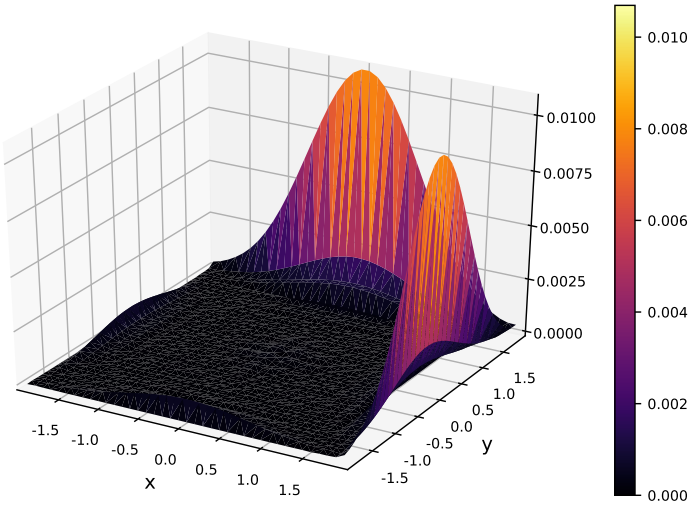
Metoda	Míra nepřesnosti
pohyblivé vážené nejmenší čtverce	0.0068291972547925035
křivky B-splines	0.00031195958155340304

Tabulka 3.7: Míry nepřesnosti aproximace funkce $g(x, y) = e^{-(x^2+y^2)}$.

Graf absolutních chyb aproximace derivace $\frac{\partial}{\partial x \partial y} g(x, y)$ je na obrázku 3.15. Metoda je velmi nepřesná na kraji intervalu, na kterém se provádí aproximace.



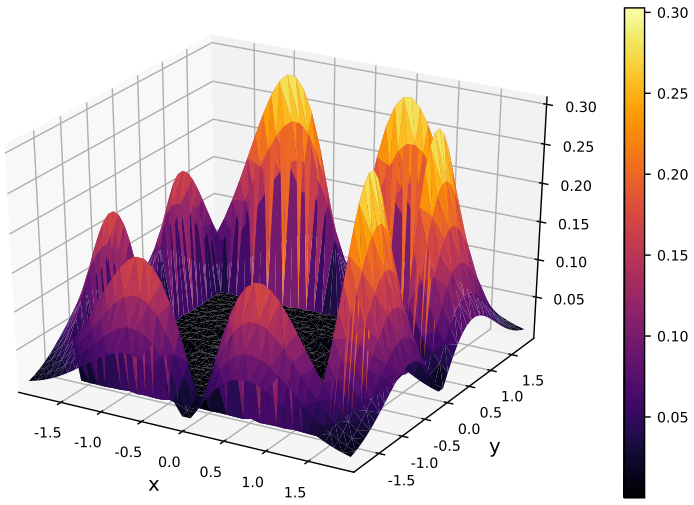
Obrázek 3.13: Absolutní chyby aproximace funkce $g(x, y) = e^{-(x^2+y^2)}$ pomocí *cell linked listu*.



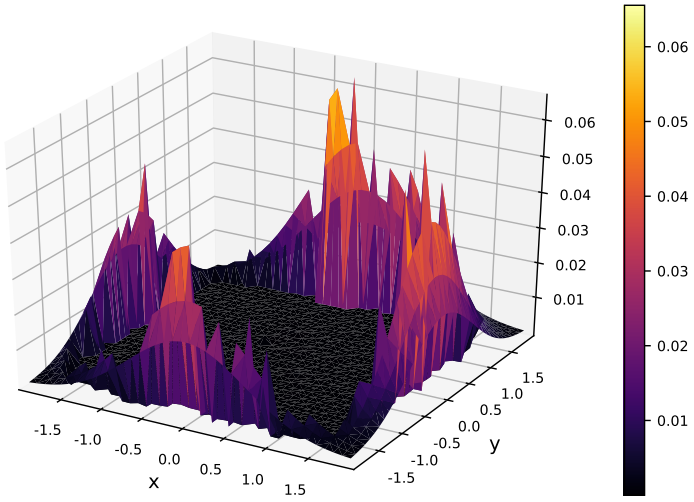
Obrázek 3.14: Absolutní chyby aproximace funkce $g(x, y) = e^{-(x^2+y^2)}$ pomocí ploch B-splines.

Míra nepřesnosti této aproximace je 0.0037921900900308417. Aproximace derivace má lepší míru nepřesnosti než aproximace funkce $g(x, y)$.

Nyní vychýlíme vstupní hodnoty přičtením náhodného čísla z intervalu $[-0.1, 0.1]$ k oběma složkám. Graf absolutních chyb aproximace funkce $g(x, y)$ je na obrázku 3.16. Míra nepřesnosti aproximace je 0.005204692719349843. Zdá se, že se oproti variantě s pravidelně rozmístěnými vstupními hodnotami přesnost mírně zlepšila.



Obrázek 3.15: Absolutní chyby aproximace derivace $\frac{\partial}{\partial x \partial y} g(x, y)$ pomocí k-d stromu.

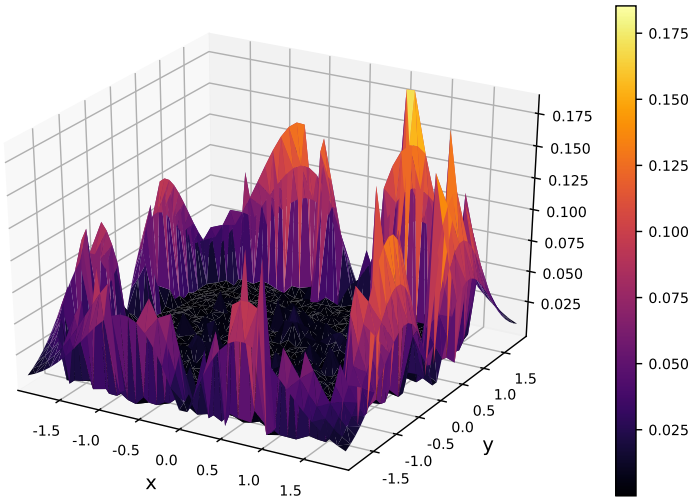


Obrázek 3.16: Absolutní chyby aproximace funkce $g(x, y)$ s vychýlenými vstupními hodnotami pomocí k-d stromu.

Na závěr si ukažme graf absolutních chyb aproximace derivace $\frac{\partial}{\partial x \partial y} g(x, y)$ na obrázku 3.17. Míra nepřesnosti této aproximace je 0.004243631005889305.

3.4.1 Časové měření

Budeme měřit dobu sestrojení aproximace v každém vstupním bodu. Zvyšování velikosti vstupu budeme provádět zvyšováním hustoty vstupních dat na



Obrázek 3.17: Absolutní chyby aproximace derivace $\frac{\partial}{\partial x \partial y} g(x, y)$ s vychýlenými vstupními hodnotami pomocí k-d stromu.

intervalu $(-1.95, 1.95) \times (-1.95, 1.95)$. Se zvětšující se hustotou dat budeme zmenšovat parametr ε váhové funkce, aby se pro sestavování aproximace vždy používalo stejné množství dat. Naměřené hodnoty jsou součtem časů 10 provedení aproximace v každé vstupní hodnotě. Všechny časové údaje jsou ve vteřinách. Mezi variantou s pravidelně rozmístěnými vstupními hodnotami a vychýlenými vstupními hodnotami nejsou výrazné časové rozdíly, proto budeme udávat čas pouze pro případ s pravidelně rozmístěnými vstupními hodnotami. Naměřené doby běhu pro tvorbu aproximací jsou v tabulce 3.8 a doby běhu pro tvorbu aproximací derivací jsou v tabulce 3.9. Naše metoda je časově mnohonásobně horší, než metody ploch B-splines.

# dat	naivní prohledávání	k-d storm	*cell linked list*	plochy B-splines
25	5.716852	4.469169	4.254995	0.000099
81	16.719839	16.442568	18.231763	0.000225
289	75.441396	69.446382	71.774254	0.001146
1089	419.341827	292.071572	295.283326	0.004251

Tabulka 3.8: Součty 10 dob běhu aproximace funkce $g(x, y) = e^{-(x^2+y^2)}$.

3.5 Testování

Ke knihovně jsou přiložené jednotkové testy kontrolující funkčnost a správnost. V době tvorby práce je pokrytí testy pouhých 54%. V knihovně je mnoho funkcí navíc, které se nepoužívají a nejsou tudíž testovány, např. operace při-

# dat	naivní prohledávání	k-d storm	*cell linked list*
25	4.619068	4.027047	4.210331
81	19.081255	18.261698	19.342711
289	82.639531	71.834943	72.359262
1089	434.343218	294.011937	297.926290

Tabulka 3.9: Součty 10 dob běhu aproximace derivace $\frac{\partial}{\partial x \partial y} g(x, y)$. Metoda křivek B-splines je vynechána, protože její implementace není funkční.

dávání, odebírání a modifikace prvku u *cell linked listu*. Správnost aproximací je ověřena podobným způsobem jako provedené experimenty. Pro úspěch v testech je potřeba, aby se absolutní odchylky aproximace a původních hodnot lišily maximálně o nějaké fixní číslo.

3.6 Poznámka k numerickým nepřesnostem

Při testování a experimentování knihovny se v některých případech lišily výsledky aproximace používající k-d strom a aproximace používající *cell linked list*. Jednou z možných příčin by může být to, že v některých aproximovaných bodech se použilo nesprávné řešení problému hledání sousedů v daném rozsahu. Tuto hypotézu otestujeme programem 3.1. Vytvoří se sada dat $\{-4, -3.8, \ldots, 4\}$ a vloží se do k-d stromu i *cell linked listu*, kde parametr pro okolí je 0.4. V každém vstupním bodu se udělá dotaz na všechny okolní body, které jsou vzdálené nejvýše o 0.4. Pokud se nerovná řešení *cell linked listu* a k-d stromu, tak vypíšeme vstup a tyto rozdílné výsledky. Jednotlivými výsledky jsou pole indexů do pole vstupních dat, které jsou v dostatečné blízkosti ke vstupní hodnotě.

```
1  using MovingWeightedLeastSquares
2
3  xs = collect(-4:0.2:4)
4  f = x -> 2sin(x) * cos(x) + 10
5  fs = [f(x) for x in xs]
6  w = (d, e) -> exp(-d^2)
7
8  kd = mwlsKd(xs, fs, 0.4, w)
9  cll = mwlsCll(xs, fs, 0.4, w)
10
11 for x in xs
12     cllRes = sort(cllInrange(cll.cll, [x, 0], cll.EPS))
13     kdRes = sort(inrange(kd.tree, [x, 0], kd.EPS))
14     if cllRes != kdRes
15       @show x, cllRes, kdres
```

```
16      end
17  end
```

Výpis kódu 3.1: Srovnání řešení k-d stromu a cell linked listu

Z výstupu programu 3.1 (viz 3.2) vidíme, že k-d strom v některých vstupních hodnotách vrací nepřesné výsledky. Jednoduchým vysvětlením je nesprávná práce s *floaty*. Lze si snadno rozmyslet, že délka výsledku pro každý nekrajní prvek by měla být 5 včetně sebe, přičemž nejvzdálenější hledaní sousedi jsou vzdálení o 0.4 a my chceme hledat všechny sousedy, kteří jsou nejvýše vzdáleni o 0.4. To může být např. tím, že tato implementace k-d stromu nepodporuje práci v 1-rozměrném prostoru a nemá uzpůsobené porovnávání *floatů* v tomto prostoru.

```
1  (x, cllRes, kdRes) = (-3.2, [3, 4, 5, 6, 7], [3, 4, 5,
       6])
2  (x, cllRes, kdRes) = (-2.8, [5, 6, 7, 8, 9], [6, 7, 8,
       9])
3  (x, cllRes, kdRes) = (-2.2, [8, 9, 10, 11, 12], [8, 9,
       10, 11])
4  ...
5  (x, cllRes, kdRes) = (-1.4, [12, 13, 14, 15, 16], [13,
       14, 15, 16])
6  ...
7  (x, cllRes, kdRes) = (2.2, [30, 31, 32, 33, 34], [31,
       32, 33, 34])
8  (x, cllRes, kdRes) = (2.8, [33, 34, 35, 36, 37], [33,
       34, 35, 36])
9  (x, cllRes, kdRes) = (3.2, [35, 36, 37, 38, 39], [36,
       37, 38, 39])
```

Výpis kódu 3.2: Výstup programu 3.1

Závěr

Metodu pohyblivých vážených nejmenších čtverců se podařilo úspěšně implementovat ve formě open source knihovny jazyka Julia. Metoda byla srovnána s interpolační metodou křivek B-splines pracující na rovnoměrné mřížce na problému aproximace derivací.

Knihovnu je možné vylepšit paralelizací výpočtů, zejména u sestavování soustavy lineárních rovnic pro vypočtení neznámých koeficientů aproximujících funkcí. V době tvorby práce bylo pokrytí testy pouhých 53%, tato statistika se určitě dá vylepšit přidáním testů. Dále chybí možnost modifikovat data zkonstruovaného aproximačního objektu.

Bibliografie

[1] Stephen M. Stigler. *The history of statistics: the measurement of uncertainty before 1900*. Belknap Press of Harvard University Press, 1998.

[2] Jeff Bezanson et al. „Julia: A Fresh Approach to Numerical Computing“. In: *SIAM Review* 59.1 (2017), s. 65–98.

[3] Peter Lancaster a Kes Salkauskas. „Surfaces generated by moving least squares methods“. In: *Mathematics of computation* 37.155 (1981), s. 141–158.

[4] David Levin. „The approximation power of moving least-squares“. In: *Mathematics of Computation of the American Mathematical Society* 67.224 (1998), s. 1517–1531.

[5] M. P. Allen. *Computer simulation of liquids*. Oxford: Oxford University Press, 2017. ISBN: 0198803206.

[6] Jon L Bentley, Donald F Stanat a E Hollins Williams Jr. „The complexity of finding fixed-radius near neighbors“. In: *Information processing letters* 6.6 (1977), s. 209–212.

[7] Jon Louis Bentley. „Multidimensional Binary Search Trees Used for Associative Searching“. In: *Commun. ACM* 18.9 (zář. 1975), s. 509–517. ISSN: 0001-0782. DOI: 10.1145/361002.361007. URL: http://doi.acm.org/10.1145/361002.361007.

[8] Jerome H Friedman, Jon Louis Bentley a Raphael Ari Finkel. „An algorithm for finding best matches in logarithmic expected time“. In: *ACM Transactions on Mathematical Software (TOMS)* 3.3 (1977), s. 209–226.

[9] Der-Tsai Lee a CK Wong. „Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees“. In: *Acta Informatica* 9.1 (1977), s. 23–29.

[10] Donald Meagher. „Geometric modeling using octree encoding“. In: *Computer Graphics and Image Processing* 19.2 (pros. 1982), s. 129–147. DOI: 10.1016/0146-664x(82)90104-6. URL: https://doi.org/10.1016/0146-664x(82)90104-6.

[11] Martin Mareš a Tomáš Valla. *Průvodce labyrintem algoritm*. Praha: CZ.NIC, z.s.p.o, 2017. ISBN: 9788088168195.

[12] Antonin Guttman. „R-trees: A Dynamic Index Structure for Spatial Searching“. In: *SIGMOD Rec.* 14.2 (červ. 1984), s. 47–57. ISSN: 0163-5808.

[13] Octavian Procopiuc et al. „Bkd-tree: A dynamic scalable kd-tree“. In: *International Symposium on Spatial and Temporal Databases*. Springer. 2003, s. 46–65.

[14] Jeff Bezanson et al. *Why We Created Julia*. [Online; Vytvořeno 2012-02-14, Viděno: 2018-04-30]. URL: https://julialang.org/blog/2012/02/why-we-created-julia.

[15] *Julia – performance tips*. [Online; Viděno: 2018-04-30]. URL: https://docs.julialang.org/en/stable/manual/performance-tips/.

[16] Benoît Legat a Sascha Timme. *JuliaAlgebra/DynamicPolynomials.jl v0.0.3*. Břez. 2018. DOI: 10.5281/zenodo.1203246. URL: https://doi.org/10.5281/zenodo.1203246.

[17] Benoît Legat et al. *JuliaAlgebra/MultivariatePolynomials.jl v0.1.4*. Břez. 2018. DOI: 10.5281/zenodo.1203247. URL: https://doi.org/10.5281/zenodo.1203247.

[18] Kristoffer Carlsson. *NearestNeighbors.jl*. Ún. 2018. URL: https://github.com/KristofferC/NearestNeighbors.jl.

[19] Vývojářský tým Julia. *DataStructures.jl*. Břez. 2018. URL: https://github.com/JuliaCollections/DataStructures.jl.

[20] Andrew Nealen. „An as-short-as-possible introduction to the least squares, weighted least squares and moving least squares methods for scattered data approximation and interpolation“. In: *URL: http://www. nealen. com/projects* 130.150 (2004), s. 25.

[21] Vývojářský tým Plots.jl. *Plots.jl*. Květ. 2018. URL: https://github.com/JuliaPlots/Plots.jl.

[22] Steven G. Johnson. *PyPlot.jl*. Květ. 2018. URL: https://github.com/JuliaPy/PyPlot.jl.

[23] Vývojářský tým Interpolations.jl. *Interpolations.jl*. Květ. 2018. URL: https://github.com/JuliaMath/Interpolations.jl.

[24] Steven G. Johnson. *Cubature.jl*. Květ. 2018. URL: https://github.com/stevengj/Cubature.jl.

Dodatek A

Uživatelská příručka

Pro použití balíčku je potřeba mít nainstalovanou Julii.

A.1 Instalace

Instalaci lze provést v terminálovém prostředí Julia příkazem

```
Pkg.clone(
"https://github.com/vutunganh/MovingWeightedLeastSquares.jl")
```

K použití knihovny ve skriptu či v terminálovém prostředí slouží příkaz

```
using MovingWeightedLeastSquares
```

A.2 Inicializace dat

K vytvoření inicializačního objektu slouží 3 pomocné funkce, které vytvoří „aproximační objekty“. Tyto objekty se budou lišit řešením problému hledání sousedů v daném rozsahu. Objekt vytvořený funkcí `mwlsNaive` bude problém řešit naivním prohledáváním, objekt vytvořený funkcí `mwlsKd` bude problém řešit k-d stromem a funkce objekt vytvořený funkcí `mwlsCll` bude problém řešit *cell linked listem*. Argumenty pro vytvoření aproximačního objektu jsou u všech 3 funkcí stejné. Signatury funkcí mají podobu

```
(inputs::Array{T, N}, outputs::Array{U}, EPS::Real,
 weightFunc::Function;
 maxDegree::Int = 2) where {T <: Real, U <: Real, N}.
```

Významy těchto argumentů jsou následující

- `inputs` – pole se vstupními hodnotami, kde každý vektor hodnot je na jednom řádku,

- `outputs` – pole s výstupními hodnotami, kde každý vektor hodnot je na jednom řádku,
- `EPS` – celé číslo s následujícími významy:
  - délka hrany *cell linked listu* (pokud se používá),
  - je-li norma aproximované hodnoty a vstupního data $\vec{x_i}$ větší než `EPS`, pak je výsledkem váhové funkce 0,
  - implicitní vzdálenost pro hledání sousedů v daném rozsahu,
- `weightFunc` – váhová funkce s 2 parametry, kde první argument je norma aproximované hodnoty a vstupního data a druhý argument je `EPS`,
- *keyword* argument `maxDegree` – maximální stupeň polynomů použitých k sestrojení aproximace, implicitní hodnota je 2.

Existuje ještě druhá podoba funkce se která se liší pouze způsobem předání vzorových dat. V této variantě data se místo `inputs` a `outputs` předávají v jednom poli a *keyword* argumentem `outputDim` se nastaví dimenze prostoru výstupních dat, které se počítají zprava. Tzn. `outputDim` s hodnotou jedna znamená, že poslední sloupec je sloupec výstupních dat, `outputDim` s hodnotou dva znamená, že poslední 2 sloupce jsou sloupce výstupních hodnot, atd.

A.3 Aproximace

Předpokládejme, že jsme si vytvořili jeden z aproximačních objektů `obj`. Příkazem `obj(inPt, dist)` lze získat hodnotu aproximace v bodě `inPt`, kde `inPt` je číslo nebo pole stejné dimenze jako dimenze prostoru vstupních hodnot. Parametr `dist` značí vzdálenost pro získání sousedů. Tato hodnota je *keyword* argument implicitně rovna `EPS`, které bylo předáno při vytvoření objektu. Chceme-li pouze znát koeficienty k polynomu výsledné aproximace, voláme funkci `calcMwlsObject`(obj, inPt, dist). Přibyl jenom jeden argument navíc a tím je aproximační objekt. K aproximaci derivace slouží funkce `mwlsDiff(obj, inPt, dirs; dist)`. Významy argumentů `obj`, `inPt` a `dist` se nezměnily od aproximace, přibyl pouze argument `dirs`. Tímto argumentem se předávají počty derivací v jednotlivých směrech. Například pro hodnotu `dist` $(1, 2)$ se bude počítat aproximace první derivace ve směru x_1 a druhé derivace ve směru x_2. Tento argument se očekává jako číslo nebo *tuple* stejné délky jako dimenze prostoru vstupních dat.

A.4 Příklady použití

Příklady použití se nachází v repozitáři na odkazu `https://github.com/vutunganh/MovingWeightedLeastSquares.jl` ve složce `test`. Lze v něm na-

lézt Jupyter notebooky `1d-testbench.ipynb` a `2d-testbench.ipynb`, které byly použity v experimentální části.

Dodatek B

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová kódy práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF