

Univerzita Pavla Jozefa Šafárika v Košiciach
Prírodovedecká fakulta

ZRÝCHLENIE VÝPOČTU SPLAJN POVRCHOV

DIPLOMOVÁ PRÁCA

Študijný odbor:	Informatika
Školiace pracovisko:	Ústav informatiky
Vedúci záverečnej práce:	doc. RNDr. Csaba Török, CSc.
Konzultant:	RNDr. Lukáš Miňo

Košice 2016

Bc. Viliam Kačala

Podakovanie

Rád by som poďakoval vedúcemu záverečnej práce doc. RNDr. Csabovi Törökovi, CSc. za cenné pripomienky, odborné vedenie a obetavosť počas tvorby práce. Taktiež by som sa rád poďakoval RNDr. Lukášovi Miňovi za cenné rady a pomoc počas tvorby aplikačnej časti tejto práce.

Zadanie práce

Abstrakt

Splajny sú dôležitá súčasť počítačovej grafiky. Jedná sa o matematický model plochy, ktorá slúži na čo „najlepšie spojenie“ konečnej množiny bodov. Termín „najlepšie spojenie“ v našom prípade znamená hladká, matematicky ľahko vyjadriteľná plocha s čo najmenším zakrivením. Využitie splajnov v grafike je veľmi široké od rôznych CAD aplikácií, v štatistike, alebo v analýze dát. Splajny existujú v mnohých formách, či už vo forme krivky v rovine, rôznych trojrozmerných telies, atď.. Táto práca si dáva za cieľ navrhnúť, analyzovať a implementovať nový algoritmus pre bikubickú interpoláciu v trojrozmernom priestore.

Abstract

Splines are important part of computer graphics. It is a mathematical model of the surface, which is for the "best connection" of any finite set of points. The term "best connection" in this case means smooth, easily calculable mathematical surface with minimal curvature. Use of splines in graphics varies from large variety of CAD applications, statistics or in data analysing. Splines exist in many forms, whether in the form of curves in the plane, a variety of three-dimensional bodies, etc. .. This work aims to design, analyze and implement new algorithm for counting and generating splines bicubic clamped interpolation in three-dimensional space.

Obsah

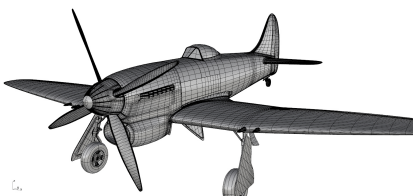
1	De Boorov algoritmus a návrh zrýchlenia	8
1.1	Polynómy	8
1.2	Splajny	9
1.3	Trojdiagonálna LU dekompozícia	13
1.4	De Boorova interpolácia	16
1.5	Počítanie derivácií redukovanou sústavou	18
1.6	Konštrukcia splajnu	22
2	Zrýchlenie	24
2.1	Procesorová architektúra	24
2.1.1	Kešovanie	25
2.1.2	Inštrukčný paralelizmus	27
2.1.3	Rýchlosť aritmetických operácií	29
2.2	Teoretické zrýchlenie	31
2.2.1	Počty pre plný algoritmus	32
2.2.2	Počty pre redukovaný algoritmus	33
2.2.3	Pamäťové nároky	35
2.2.4	Zhrnutie	36
2.3	Zrýchlenie v praxi	37
3	Implementácia a užívateľská príručka	40
3.0.1	Vizualizácia	42
3.0.2	Výkonnostný tester	44

Úvod

Témou mojej diplomovej práce sú rovinné krivkové splajny a priestorové splajn povrchy, pričom naším cieľom je preskúmať nové poznatky o splajnoch, na ktorých istý čas pracuje vedúci tejto práce doc. RNDr. Csaba Török, CSc.

Výsledok tejto práce je návrh novej metódy výpočtu derivácií splajnu v jeho uzloch, jej porovnanie s de Boorovým postupom z ktorého vychádzame, vysvetlenie zrýchlenia a implementácia aplikácií na vizuálne a výkonnostne porovnávanie oboch metód.

Prínos zrýchlenia výpočtu splajnov spočíva v lepších možnostiach modelovania ľubovoľných trojrozmerných útvarov ako po častiach polynomických funkcií. Akékoľvek zrýchlenie totiž znamená možnosť v reálnom čase modelovať zložitejšie objekty, čo je užitočné nielen vzhľadom na vizuálnu reprezentáciu, ale aj na skúmanie fyzikálnych vlastností ako napríklad aerodynamické vlastnosti lietadiel a podobne.



Obr. 1: Splajnový model lietadla.

Technológia v ktorej tieto poznatky implementujeme je Microsoft Silverlight. Jedná sa o veľmi schopný nástroj na tvorbu webových aplikácií s plnou podporou hardvérovej akcelerácie užívateľského prostredia a možnosťou tvorby rýchlej trojrozmernej grafiky. Výhodou tohto frameworku je, keďže beží na platforme Microsoft .NET, možnosť jednoduchšej portácie na desktopovú prípadne mobilnú aplikáciu. Na výkonnostné porovnanie nášho algoritmu s existujúcim však implementujeme samostatnú aplikáciu nad C++.

Štruktúra práce je nasledovná.

- **De Boorov algoritmus a návrh zrýchlenia**
 - **Polynómy**
Základné pojmy o polynómoch.
 - **Splajny**
Definícia krivkových a povrchových splajn.
 - **LU dekompozícia**
Spôsob počítania trojdiagonálnych sústav.
 - **De Boorova interpolácia**
V tejto časti si vysvetlíme splajn interpoláciu podľa Carla de Boora.
 - **Počítanie derivácií redukovanou sústavou**
Ukážka modifikovaného postupu pre kubické splajny a jeho rozšírenie pre bikubické splajny.
 - **Konštrukcia splajnu**
Zostrojenie splajnu ako zloženej funkcie segmentov.
- **Zrýchlenie**
Očakávané zrýchlenie výpočtov novým algoritmom.
 - **Procesorová architektúra**
Tu si objasníme vplyv moderných inštrukčných sád na zrýchlenie redukovaného algoritmu.
 - **Teoretické zrýchlenie**
Spočítanie časovej zložitosti vysvetlenie vplyvu základných aritmetických a pamäťových operácií.
 - **Zrýchlenie v praxi**
Výsledky z našej implementácie.
- **Implementácia a užívateľská príručka**
Podrobnosti implementácie v MS Silverlight a C++. Stručný návod na použitie programov.
- **Záver**

Formálne tézy diplomovej práce sú:

- Analýza modelov interpolačných splajnov.
- Redukovaný algoritmus výpočtu koeficientov splajn povrchov.

- Testovanie faktorov vplývajúcich na rýchlosť výpočtu splajn koeficientov
 - Klasický a Hermitov splajn
 - De Boorov a redukovaný algoritmus
 - Vplyv latencie a priepustnosti procesorových inštrukcií a vyrovnávacích pamätí.

Kapitola 1

De Boorov algoritmus a návrh zrýchlenia

V tejto kapitole si postupne zdefinujeme pojem polynómu a tento postupne rozšírime na pojem splajnu. Následne si popíšeme dve metódy výpočtu derivácií splajnu a konštrukciu celej splajnovej krivky alebo plochy.

1.1 Polynómy

Pre úplnosť skôr než si povieme o splajnoch, predstavíme si pojem polynomickej funkcie ktorý je neoddeliteľnou súčasťou pri definícii splajnu.

Definícia 1.1 Nech n je z $\mathbb{N} \cup \{0\}$ a pre každé i z $\{0, \dots, n\}$ je a_i z \mathbb{R} , pričom $a_n \neq 0$. Funkciu $p: \mathbb{R} \rightarrow \mathbb{R}$ tvaru

$$p(x) = \sum_{i=0}^n a_i x^i$$

nazveme *polynomickeá funkcia jednej premennej stupňa n* .

Označenie 1.2 Polynomickeé funkcie jednej premennej budeme skrátene nazývať *polynomickeé funkcie*. Špeciálne polynomickeé funkcie stupňa 3 nazveme *kubické funkcie*.

V práci však budeme pracovať najmä s funkciami dvoch premenných. Analogicky si pre dve premenné zdefinujeme aj polynomickeé funkcie.

Definícia 1.3 Nech n a m sú z $\mathbb{N} \cup \{0\}$ a pre každé i z $\{0, \dots, n\}$ a j z $\{0, \dots, m\}$ je a_{ij} z \mathbb{R} , pričom $a_{nm} \neq 0$. Funkciu $p: \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ tvaru

$$p(x, y) = \sum_{i=0}^n \sum_{j=0}^m a_{ij} x^i y^j$$

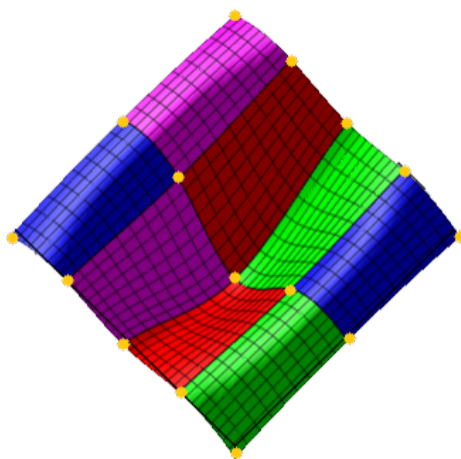
nazveme *polynomická funkcia dvoch premenných stupňov n a m*

Označenie 1.4 Polynomické funkcie dvoch premenných budeme skrátené nazývať *bipolynomické funkcie*. Bipolynomické funkcie stupňov 3 a 3 nazveme *bikubické funkcie*.

1.2 Splajny

V našej práci pracujeme s hermitovskými splajnami, ktoré sú štandardne triedy C^1 , teda splajnami, ktorých prvé derivácie v uzloch sa rovnajú. Pre naše potreby budeme uvažovať iba hermitovské splajny triedy C^2 , teda tie pri ktorých v uzloch máme zaručenú rovnosť aj derivácií druhého rádu. Všeobecne sa splajny triedy C^2 nazývajú *naturálne splajny*[6].

Keďže naším cieľom je zrýchlený algoritmus počítania povrchových splajnov, ktorý vznikol zovšeobecnením algoritmu pre krivkové splajny tak si zaviedieme pojem splajnu pre krivky aj plochy. Iste by bolo možno elegantnejšie zadať splajn všeobecne, ale rozdelenie definície špeciálne pre krivkové splajny v rovine a povrchové v priestore bude čitateľnejšie.



Obr. 1.2: Funkcia $\sin(\sqrt{x^2 + y^2})$ aproximovaná bikubickým splajnom.

Formálnejšie si popíšeme základné pojmy s ktorými budem v tomto článku pracovať.

Označenie 1.5 Postupnosť (a_0, a_1, \dots) nazveme *rovnomerne rastúcou* ak pre ľubovoľné i z $\{0, 1, \dots\}$ platí $a_{i+1} > a_i$ a pre ľubovoľné i, j z $\{0, 1, \dots\}$ platí $a_{j+1} - a_j = a_{i+1} - a_i$.

Formálna definícia krivkového splajnu v rovine je nasledovná.

Definícia 1.6 Nech $I \geq 0$ je prirodzené číslo, (u_0, \dots, u_{I-1}) je rovnomerne rastúca postupnosť. Nech pre každé i z $\{0, 1, \dots, I-2\}$ funkcie S_i sú polynomicke funkcie, ktoré spĺňajú

- $S_i(u_{i+1}) = S_{i+1}(u_{i+1})$,
- $\frac{dS_i}{dx}(u_{i+1}) = \frac{dS_{i+1}}{dx}(u_{i+1})$

Funkciu S z intervalu $[u_0, u_{I-1}]$ do \mathbb{R} pre ktorú platí:

$$S(x) = \begin{cases} S_0(x, y) & \text{pre } x \in [u_0, u_1], \\ S_1(x, y) & \text{pre } x \in [u_1, u_2], \\ \vdots & \\ S_{I-1}(x, y) & \text{pre } x \in [u_{I-2}, u_{I-1}], \end{cases} \quad (1.1)$$

nazveme *krivkový splajn*.

Označenie 1.7 Pri označeniach z predchádzajúcej definície označme:

- Funkcie S_i nazveme *segmenty*.
- Hodnoty u_i nazveme *uzly*.
- Uzly u_0 a u_{I-1} nazveme *krajné uzly*.
- $z_i = S(u_i)$ nazveme *funkčné hodnoty splajnu v uzloch*.
- $d_i^x = \frac{dS(u_i)}{dx}$ nazveme *derivácie v uzloch*. $h = u_i - u_{i-1}$ pre ľubovoľné $i \in \{1, \dots, I-1\}$.

Definíciu krivkového splajnu následne rozšírime na povrchový splajn.

Definícia 1.8 Nech $I \geq 0$ a $J \geq 0$ sú prirodzené čísla, (u_0, \dots, u_{I-1}) a (v_0, \dots, v_{J-1}) sú rovnomerne rastúce postupnosti. Nech pre každé i z $\{0, 1, \dots, I-2\}$ a j z $\{0, 1, \dots, J-2\}$ funkcie $S_{i,j}$ sú bipolynomicke funkcie, ktoré spĺňajú

- $S_{i,j}(u_{i+1}, y) = S_{i+1,j}(u_{i+1}, y)$,
- $S_{i,j+1}(x, v_{j+1}) = S_{i,j+1}(x, v_{j+1})$,

- $\frac{\partial S_{i,j}}{\partial x}(u_{i+1}, y) = \frac{\partial S_{i+1,j}}{\partial x}(u_{i+1}, y)$
- $\frac{\partial S_{i,j+1}}{\partial y}(x, v_{j+1}) = \frac{\partial S_{i,j}}{\partial y}(x, v_{j+1}),$

Funkciu S z intervalu $[u_0, u_{I-1}] \times [v_0, v_{J-1}]$ do \mathbb{R} pre ktorú platí:

$$S(x, y) = \begin{cases} S_{0,0}(x, y) & \text{pre } (x, y) \in [u_0, u_1] \times [v_0, v_1], \\ S_{0,1}(x, y) & \text{pre } (x, y) \in [u_0, u_1] \times [v_1, v_2], \\ \vdots & \\ S_{0,J-1}(x, y) & \text{pre } (x, y) \in [u_0, u_1] \times [v_{J-2}, v_{J-1}], \\ S_{1,0}(x, y) & \text{pre } (x, y) \in [u_1, u_2] \times [v_0, v_1], \\ \vdots & \\ S_{1,J-1}(x, y) & \text{pre } (x, y) \in [u_1, u_2] \times [v_{J-2}, v_{J-1}], \\ \vdots & \\ S_{I-1,0}(x, y) & \text{pre } (x, y) \in [u_{I-2}, u_{I-1}] \times [v_0, v_1], \\ \vdots & \\ S_{I-1,J-1}(x, y) & \text{pre } (x, y) \in [u_{I-2}, u_{I-1}] \times [v_{J-2}, v_{J-1}], \end{cases} \quad (1.2)$$

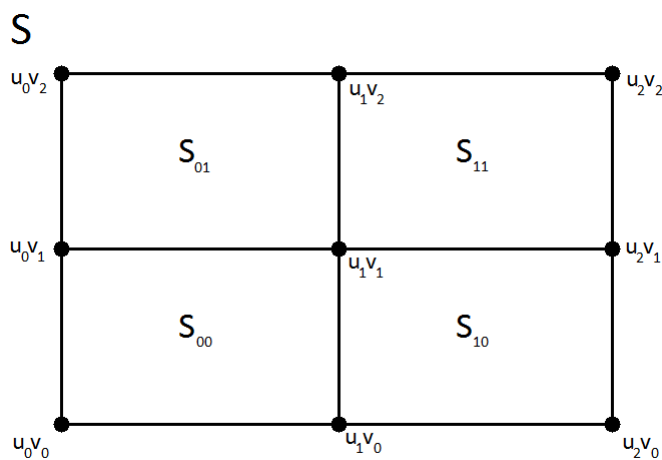
nazveme *splajn*.

Označenie 1.9 Pre úplnosť analogicky ako pri krivkách označme:

- Funkcie S_{ij} nazveme *segmenty splajnu*.
- Dvojice (u_i, v_j) nazveme *uzly*.
- Uzly (u_0, v_0) , (u_{I-1}, v_0) , (u_0, v_{J-1}) a (u_{I-1}, v_{J-1}) nazveme *rohové uzly*.
- $z_{i,j} = S(u_i, v_j)$ nazveme *funkčné hodnoty splajnu v uzloch*.
- $d_{i,j}^x = \frac{\partial S(u_i, v_j)}{\partial x}$ nazveme *x-ové derivácie v uzloch*.
- $d_{i,j}^y = \frac{\partial S(u_i, v_j)}{\partial y}$ nazveme *y-ové derivácie v uzloch*.
- $d_{i,j}^{xy} = \frac{\partial^2 S(u_i, v_j)}{\partial x \partial y}$ nazveme *zmiešané derivácie v uzloch*.
- $h_x = u_i - u_{i-1}$ a $h_y = v_j - v_{j-1}$ pre ľubovoľné $i \in \{1, \dots, I-1\}$, resp. $j \in \{1, \dots, J-1\}$.

Prirodzene vidieť, že takto definovaný krivkový a povrchový splajn je triedy C^1 . Pri vhodne zvolených hodnotách derivácií v uzloch vieme ale zaručiť rovnosť aj druhých derivácií, čím dostaneme splajn z triedy C^2 čo nám zaručí hladkosť „spojenia“ jednotlivých segmentov. Ak uvažujeme ľubovoľnú funkciu tak pomocou nej vieme nájsť žiadané hodnoty derivácií. Navyše takto získaný splajn bude danú funkciu interpolovať, teda ju „napodobní“. To je žiadané napríklad v grafickom modelovaní, pretože pre počítač je jednoduchšie a rýchlejšie pracovať s polynómom interpolujúcim napríklad goniometrickú funkciu ako priamo s ňou. Navyše sa dá ukázať, že interpolácia funkcie splajnom je jednoznačná. Témou tejto práce sú práve metódy výpočtu prvých derivácií, ktorými je možné dosiahnuť práve spomenutý cieľ.

Témou následovných častí práce je úloha na základe vstupných uzlov a funkcie $d : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ nájsť *hladkú*, po častiach definovanú funkciu $S : [u_0, u_{I+1}] \times [v_0, v_{J+1}] \rightarrow \mathbb{R}$ z so spojitými deriváciami prvého aj druhého rádu takú, že pre každé $i \in 0, \dots, I-1$ a $j \in 0, \dots, J-1$ platí $z_{i,j} = f(u_i, v_j)$ a hodnoty derivácií S v uzloch sú čo najbližšie hodnotám derivácií f . Funkciu S nazývame splajn (konkrétne povrchový splajn), pričom jednotlivé časti nazveme *segmenty*.



Obr. 1.3: Ukážka uzlov pre štvorsegmentový splajn.

Ak i je z $\{0, \dots, I-2\}$ a j je z $\{0, \dots, J-2\}$, tak štvorica uzlov (u_i, v_j) , (u_i, v_{j+1}) , (u_{i+1}, v_j) a (u_{i+1}, v_{j+1}) tvoria obdĺžnikový úsek nad ktorým sa nachádza splajnový segment. Každý segment $S_{i,j}$ je bikubická funkcia z $[u_i, u_{i+1}] \times [v_j, v_{j+1}]$ do \mathbb{R} . Výsledná funkcia S teda vznikne zjednotením segmentov $S_{i,j}$. Na vypočítanie každého segmentu potrebujeme štyri uzly, a pre každý uzol príslušne hodnoty z , d^x , d^y a d^{xy} . Tieto hodnoty vieme získať buď priamo z

interpolovanej funkcie f , alebo ich vypočítať iným spôsobom, napríklad De Boorovou interpoláciou.

1.3 Trojdiagonálna LU dekompozícia

Definícia 1.10 Nech $n \geq 3$ je z $\mathbb{N} \cup \{0\}$. Sústavu rovníc tvaru

$$\begin{pmatrix} b_1 & a_1 & 0 & \cdots & 0 & 0 \\ c_1 & b_2 & a_2 & \cdots & 0 & 0 \\ 0 & c_2 & b & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & b & a_{n-1} \\ 0 & 0 & 0 & \cdots & c_{n-1} & b_n \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} r_1 \\ r_2 \\ r_3 \\ \vdots \\ r_{n-1} \\ r_n \end{pmatrix} \quad (1.3)$$

nazveme *trojdiagonálna sústava lineárnych rovníc*.

Základ oboch postupoch tkvie v opakovanom počítaní systémov trojdiagonálnych lineárnych rovníc. Jeden zo spôsobov riešenia týchto rovníc spočíva v LU dekompozícii $A\mathbf{x} = L \underbrace{U\mathbf{x}}_{\mathbf{y}} = \mathbf{r}$:

$$A = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & 0 \\ \lambda_2 & 1 & 0 & \cdots & 0 & 0 \\ 0 & \lambda_3 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & 0 & \cdots & \lambda_n & 1 \end{pmatrix} \cdot \begin{pmatrix} v_1 & 1 & 0 & \cdots & 0 & 0 \\ 0 & v_1 & 1 & \cdots & 0 & 0 \\ 0 & 0 & v_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & v_{n-1} & 0 \\ 0 & 0 & 0 & \cdots & 0 & v_n \end{pmatrix}, \quad (1.4)$$

Pre k z $\{2, \dots, n\}$ sú hodnoty v_k a λ_k určené takto:

$$v_i = b, \left\{ \lambda_i = \frac{1}{v_{i-1}}, v = b - \lambda_i \right\}, i \in \{2, \dots, n\}. \quad (1.5)$$

Pre priamy a spätný chod máme

$$\text{Priamy: } L\mathbf{y} = \mathbf{r}, y_1 = r_1, \{y_i = r_i - \lambda_i\}, i \in \{2, \dots, n\}, \quad (1.6)$$

$$\text{Spätný: } U\mathbf{d} = \mathbf{y}, d_i = \frac{y_i}{u_i}, \left\{ d_i = \frac{1}{u_i}(y_i - d_{i+1}) \right\}, i \in \{n-1, \dots, 1\}. \quad (1.7)$$

LU dekompozíciou sa rieši ako de Boorova sústava (1.12), tak aj naša redukovaná (1.23). Nižšie si ukážeme pseudokód tohto algoritmu. Procedúra algoritmu je priestorovo optimalizovaná kde používame LU rozklad robíme

Algoritmus 1 Všeobecná LU dekompozícia

```
1: procedúra VYRIEŠLU
2:   vstup:
3:      $c[0..n-2]$  spodná diagonála
4:      $b[0..n-1]$  hlavná diagonála
5:      $a[0..n-2]$  horná diagonála, po skončení sa nezachová
6:      $r[0..n-1]$  pravá strana
7:   výstup:  $r[0..n-1]$  pravá strana obsahujúca výsledok
8:
9:      $a[0] \leftarrow a[0]/b[0]$ 
10:     $r[0] \leftarrow r[0]/b[0]$ 
11:    pre  $i$  od 1 až  $n-1$ 
12:       $m \leftarrow 1/b[i] - c[i] \cdot a[i-1]$ 
13:       $a[i] \leftarrow m \cdot a[i]$ 
14:       $r[i] \leftarrow m \cdot (r[i] - c[i] \cdot r[i-1])$ 
15:    pre  $i$  od  $n-2$  až 0
16:       $r[i] \leftarrow r[i] - a[i] \cdot r[i+1]$ 
```

priamo na vstupných vektoroch predstavujúce diagonály a výsledok ukladáme do vektora obsahujúci pravú stranu.

V prípade počítania derivácií je sústava rovníc v špeciálnom tvare podľa 1.12 pre plný alebo 1.23 pre redukovaný algoritmus. V oboch prípadoch všetky čísla na hornej aj spodnej diagonále majú hodnotu 1. Na hlavnej diagonále sú to hodnoty 4 pri plnom algoritme a v prípade redukovaného postupu sú to hodnoty -14 . Posledný prvok diagonály tiež môže obsahovať hodnotu -15 ak počet rovníc je párny. Teda pri implementácii si nemusíme pamätať vektory, stačí nám zapamätať si iba štyri hodnoty. Upravený algoritmus, ktorý rieši LU dekompozíciu špeciálne pre plný a redukovaný algoritmus je nižšie.

Algoritmus 2 Špeciálna LU dekompozícia

```
1: procedúra VYRIEŠLU
2:   vstup:
3:
4:    $b$  hodnota prvkov hlavnej diagonály
5:    $b_l$  hodnota posledného prvku hlavnej diagonály
6:    $r[0..n-1]$  pravá strana
7:   výstup:  $r[0..n-1]$  pravá strana obsahujúca výsledok
8:
9:    $u \leftarrow [0..n-2]$   $\triangleright$  Pomocný vektor nahradzujúci hornú diagonálu
10:   $m \leftarrow 1/b$ 
11:   $p[0] \leftarrow q$ 
12:   $r[0] \leftarrow q \cdot r[0]$ 
13:  pre  $i$  od 1 až  $n-2$ 
14:     $q \leftarrow 1/b - p[i-1]$ 
15:     $p[i] \leftarrow m$ 
16:     $r[i] \leftarrow m \cdot (r[i] - r[i-1])$ 
17:   $m \leftarrow 1/b_l - p[n-1]$ 
18:   $p[i] \leftarrow m$ 
19:   $r[i] \leftarrow m \cdot (r[n-1] - r[n-2])$ 
20:  pre  $i$  od  $n-2$  až 0
21:     $r[i] \leftarrow r[i] - p[i] \cdot r[i+1]$ 
```

1.4 De Boorova interpolácia

Nech sú dané hodnoty (u_0, \dots, u_{I-1}) a (v_0, \dots, v_{J-1}) , kde I, J sú z $\mathbb{N} \cup \{0\}$, pričom chceme interpolovať funkciu f na $[u_0, u_{I-1}] \times [v_0, v_{J-1}]$. Výsledný splajn bude teda tvorený $(I-1) \cdot (J-1)$ segmentami. Ako bolo spomenuté, každý segment splajnu potrebuje štyri uzly a hodnoty z , d^x , d^y a d^{xy} , ktoré potrebujeme získať z funkcie f . Získanie derivácií priamo z f môže byť v praxi na počítači nákladné najmä v prípade, keď pracujeme so symbolicky zapísanou funkciou vo forme textového reťazca, ktorú je potrebné interpretovať.

Algoritmus nájdený Carlom de Boorom[4] umožňuje s malou odchýlkou odhadnúť hodnoty derivácií v uzloch na základe týchto vstupných hodnôt, ktoré potrebujeme získať vyhodnotením funkcie f :

- $z_{i,j} = f(u_i, v_j)$ pre $i \in \{0, \dots, I-1\}$, $j \in \{0, \dots, J-1\}$.
- $d_{i,j}^x = \frac{\partial f}{\partial x}(u_i, v_j)$ pre $i \in \{0, I-1\}$, $j \in \{0, \dots, J-1\}$.
- $d_{i,j}^y = \frac{\partial f}{\partial y}(u_i, v_j)$ pre $i \in \{0, \dots, I-1\}$, $j \in \{0, J-1\}$.
- $d_{i,j}^{xy} = \frac{\partial^2 f}{\partial x \partial y}(u_i, v_j)$ pre $i \in \{0, I-1\}$, $j \in \{0, J-1\}$.

Príklad 1.11 Na príklade uzlov z obrázka 1.3 potrebujeme poznať hodnoty nasledovne:

- $z_{0,2}, z_{1,2}, z_{2,2},$
 $z_{0,1}, z_{1,1}, z_{2,1},$
 $z_{0,0}, z_{1,0}, z_{2,0},$
- $d_{0,2}^x, \quad , d_{2,2}^x,$
 $d_{0,1}^x, \quad , d_{2,1}^x,$
 $d_{0,0}^x, \quad , d_{2,0}^x,$
- $d_{0,2}^y, d_{1,2}^y, d_{2,2}^y,$
 $d_{0,0}^y, d_{1,0}^y, d_{2,0}^y,$
- $d_{0,2}^{xy}, \quad , d_{2,2}^{xy},$
 $d_{0,0}^{xy}, \quad , d_{2,0}^{xy},$

Zvyšné derivácie d^x , d^y a d^{xy} vieme jednoznačne vypočítať pomocou $2(I) + J + 5$ lineárnych sústav s celkovo $3IJ + I + J + 2$ rovnicami: Nižšie uvádzame modelové rovnice, pomocou ktorých sú zostrojené tieto sústavy lineárnych rovníc.

Pre $j \in \{0, \dots, J-1\}$, teda pre každý stĺpec j vypočítame parciálne derivácie d^x

$$d_{i+1,j}^x + 4d_{i,j}^x + d_{i-1,j}^x = \frac{3}{h_x}(z_{i+1,j} - z_{i-1,j}), \quad (1.8)$$

$$i \in \{1, \dots, I-2\}$$

Pre $j \in \{0, J-1\}$, teda pre prvý a posledný stĺpec vypočítame parciálne derivácie $d^{x,y}$

$$d_{i+1,j}^{xy} + 4d_{i,j}^{xy} + d_{i-1,j}^{xy} = \frac{3}{h_x}(d_{i+1,j}^y - d_{i-1,j}^y), \quad (1.9)$$

$$i \in \{1, \dots, I-2\}$$

Pre $i \in \{0, \dots, I-1\}$, teda pre každý riadok i vypočítame parciálne derivácie d^y

$$d_{i,j+1}^y + 4d_{i,j}^y + d_{i,j-1}^y = \frac{3}{h_y}(z_{i,j+1} - z_{i,j-1}), \quad (1.10)$$

$$j \in \{1, \dots, J-2\}$$

Pre $i \in \{0, \dots, I-1\}$, teda pre každý riadok j dopočítame parciálne derivácie $d^{x,y}$

$$d_{i,j+1}^{xy} + 4d_{i,j}^{xy} + d_{i,j-1}^{xy} = \frac{3}{h_y}(d_{i,j+1}^x - d_{i,j-1}^x), \quad (1.11)$$

$$j \in \{1, \dots, J-2\}$$

Každá z týchto sústav má takýto maticový tvar:

$$\begin{pmatrix} 4 & 1 & 0 & \cdots & 0 & 0 \\ 1 & 4 & 1 & \cdots & 0 & 0 \\ 0 & 1 & 4 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 4 & 1 \\ 0 & 0 & 0 & \cdots & 1 & 4 \end{pmatrix} \cdot \begin{pmatrix} D_1 \\ D_2 \\ D_3 \\ \vdots \\ D_{N-3} \\ D_{N-2} \end{pmatrix} = \begin{pmatrix} \frac{3}{\Delta}(Y_2 - Y_0) - D_0 \\ \frac{3}{\Delta}(Y_3 - Y_1) \\ \frac{3}{\Delta}(Y_4 - Y_2) \\ \vdots \\ \frac{3}{\Delta}(Y_{N-2} - Y_{N-4}) \\ \frac{3}{\Delta}(Y_{N-1} - Y_{N-3}) - D_{N-1} \end{pmatrix}, \quad (1.12)$$

kde podľa toho o ktorú z modelových rovníc sa jedná, hodnoty N , D a Y zadávame následovne. Nech k z $1, \dots, K-1$:

- $N = I$, $h = h_x$, $D_k = d_{k,j}^x$ a $Y_k = z_{k,j}$, pre rovnicu 1.8.
- $N = I$, $h = h_x$, $D_k = d_{k,j}^{xy}$ a $Y_k = d_{k,j}^y$, pre rovnicu 1.9.
- $N = J$, $h = h_y$, $D_k = d_{i,k}^y$ a $Y_k = z_{i,k}$, pre rovnicu 1.10.
- $N = J$, $h = h_y$, $D_k = d_{i,k}^{xy}$ a $Y_k = d_{i,k}^x$, pre rovnicu 1.10.

Po vypočítaní všetkých derivácií môžeme funkciu jednoznačne interpolovať splajnom.

Poznámka 1.12 V aplikácií na vizualizáciu používame práve Hermitove splajny.

Poznámka 1.13 De Boorova interpolácia vo všeobecnosti nepredpokladá len rovnomerne rastúce postupnosti uzlov (u_0, \dots, u_I) a (v_0, \dots, v_J) . Náš postup v ďalšej časti článku ale funguje len s takýmito postupnosťami. Preto sme v tejto časti popísali len špeciálny prípad interpolácie pre takto určené uzly.

1.5 Počítanie derivácií redukovanou sústavou

V rámci svojej bakalárskej práce som popisoval nový postup pre kubické splajnové krivky triedy C2, teda splajny, kde interpolovaná funkcia f je typu $\mathbb{R} \rightarrow \mathbb{R}$. Cieľom tejto práce je postup zovšeobecniť pre bikubické splajny, teda pre splajny, kde interpolovaná funkcia f je typu $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$.

Označenie 1.14 Teraz popíšeme tento rozšírený algoritmus, ktorý pracovne označíme ako *redukovaný algoritmus*. Ďalej budeme pôvodný de Boorov postup, označovať pojmom *plný algoritmus*.

Vstupné hodnoty sú identické ako pri pôvodnom algoritme. Máme dané (u_0, \dots, u_{I-1}) a (v_0, \dots, v_{J-1}) , kde I, J sú z $\mathbb{N} \cup \{0\}$, pričom chceme interpolovať funkciu f na $[u_0, u_{I-1}] \times [v_0, v_{J-1}]$. Pre pripomenutie potrebujeme z funkcie f priamo získať tieto hodnoty.

- $z_{i,j} = f(u_i, v_j)$, pre $i \in \{0, \dots, I-1\}$, $j \in \{0, \dots, J-1\}$.
- $d_{i,j}^x$ pre $i \in \{0, I-1\}$, $j \in \{0, \dots, J-1\}$.
- $d_{i,j}^y$ pre $i \in \{0, \dots, I-1\}$, $j \in \{0, J-1\}$.
- $d_{i,j}^{xy}$ pre $i \in \{0, I-1\}$, $j \in \{0, J-1\}$.

Zvyšné derivácie d^x , d^y a d^{xy} vieme jednoznačne vypočítať pomocou $2(I) + J + 5$ lineárnych sústav s celkovo $3IJ + I + J + 2$ rovnicami: Nižšie uvádzame modelové rovnice, pomocou ktorých sú zostrojené tieto sústavy lineárnych rovníc. Označme I_l a J_l indexy po ktoré budeme iterovať. Platí:

$$I_l = \begin{cases} I - 2 & \text{ak } I \text{ je nepárne,} \\ I - 3 & \text{ak } I \text{ je párne,} \end{cases}$$

$$J_l = \begin{cases} J - 2 & \text{ak } J \text{ je nepárne,} \\ J - 3 & \text{ak } J \text{ je párne,} \end{cases}$$

Pre $j \in \{0, \dots, J - 1\}$, teda pre každý stĺpec j vypočítame parciálne derivácie d^x

$$d_{i+2,j}^x - 14d_{i,j}^x + d_{i-2,j}^x = \frac{3}{h_x}(z_{i+2,j} - z_{i-2,j}) - \frac{12}{h_x}(z_{i+1,j} - z_{i-1,j}), \quad (1.13)$$

$$i \in \{2, 4, \dots, I_l\}$$

Rovnica je podobná ako plnom algoritme 1.8. Všimnime si, že sústavu rovníc teraz budujeme len pre párne indexy i , teda vyriešením tejto sústavy získame iba polovicu žiadaných hodnôt d^x . Pre $i \in \{1, 3, \dots, I_l\}$ a $j \in \{0, \dots, J - 1\}$ zvyšné derivácie d^x vypočítame ako

$$d_{i,j}^x = \frac{3}{4h_x}(z_{i+1,j} - z_{i-1,j}) - \frac{1}{4}(d_{i+1,j}^x - d_{i-1,j}^x) \quad (1.14)$$

Pre $i \in \{0, \dots, I - 1\}$, teda pre každý riadok i analogicky vypočítame parciálne derivácie d^y

$$d_{i,j+2}^y - 14d_{i,j}^y + d_{i,j-2}^y = \frac{3}{h_y}(z_{i,j+2} - z_{i,j-2}) - \frac{12}{h_y}(z_{i,j+1} - z_{i,j-1}), \quad (1.15)$$

$$i \in \{2, 4, \dots, I_l\}$$

Následne analogicky pre $i \in \{1, 2, \dots, I - 1\}$ a $j \in \{1, 3, \dots, J_l\}$ zvyšné derivácie d^y vypočítame ako

$$d_{i,j}^y = \frac{3}{4h_y}(z_{i,j+1} - z_{i,j-1}) - \frac{1}{4}(d_{i,j+1}^y - d_{i,j-1}^y) \quad (1.16)$$

Pre $j \in \{0, J - 1\}$, teda pre prvý a posledný stĺpec vypočítame parciálne derivácie $d^{x,y}$ rovnako ako pri plnom algoritme.

$$d_{i+1,j}^{xy} + 4d_{i,j}^{xy} + d_{i-1,j}^{xy} = \frac{3}{h_x}(d_{i+1,j}^y - d_{i-1,j}^y), \quad (1.17)$$

$$i \in \{1, \dots, I - 2\}$$

Pre $i \in \{0, I-1\}$, teda pre prvý a posledný riadok analogicky vypočítame parciálne derivácie $d^{x,y}$

$$d_{i,j+1}^{xy} + 4d_{i,j}^{xy} + d_{i,j-1}^{xy} = \frac{3}{h_y}(d_{i,j-1}^x - d_{i-1,j}^x), \quad (1.18)$$

$$j \in \{1, \dots, J-2\}$$

Pre $i \in \{2, 4, \dots, I_l\}$, teda pre každý stĺpec i dopočítame parciálne derivácie $d^{x,y}$

$$\begin{aligned} d_{i,j+2}^{xy} + 4d_{i,j}^{xy} + d_{i,j-2}^{xy} = & \\ & \frac{1}{7}(d_{i-2,j+2}^{xy} - d_{i-2,j-2}^{xy}) - 2d_{i-2,j}^{xy} \\ & + \frac{3}{7h_x}(d_{i-2,j+2}^y - d_{i-2,j-2}^y) + \frac{3}{7h_y}(-d_{i-2,j+2}^x - d_{i-2,j-2}^x) \\ & + \frac{9}{7h_x}(d_{i,j+2}^y - d_{i,j-2}^y) + \frac{9}{7h_x h_y}(-z_{i-2,j+2} + z_{i-2,j-2}) \\ & + \frac{12}{7h_x}(-d_{i-1,j+2}^y - d_{i-1,j-2}^y) + \frac{12}{7h_y}(d_{i-2,j+1}^x - d_{i-2,j-1}^x) \\ & + \frac{3}{7h_y}(d_{i,j+2}^x - d_{i,j-2}^x) + \frac{27}{7h_x h_y}(-z_{i,j+2} + z_{i,j-2}) \\ & + \frac{36}{7h_x h_y}(z_{i-1,j+2} - z_{i-1,j-2} + z_{i-2,j+1} - z_{i-2,j-1}) \\ & - \frac{6}{h_x}d_{i-2,j}^y + \frac{144}{7h_x h_y}(-z_{i-1,j+1} + z_{i-1,j-1}) + \frac{24}{h_x}d_{i-1,j}^y, \end{aligned} \quad (1.19)$$

$$j \in \{4, 6, \dots, J_l - 2\}$$

Následne vypočítame zvyšné derivácie $d^{x,y}$. Najprv pre $i \in \{1, 3, \dots, I_l\}$ a $j \in \{1, 3, \dots, J_l\}$ platí

$$\begin{aligned} d_{i,j}^{xy} = & \frac{1}{16}(d_{i+1,j+1}^{xy} + d_{i+1,j-1}^{xy} + d_{i-1,j+1}^{xy} + d_{i-1,j-1}^{xy}) \\ & - \frac{3}{16h_y}(d_{i+1,j+1}^x - d_{i+1,j-1}^x + d_{i-1,j+1}^x - d_{i-1,j-1}^x) \\ & - \frac{3}{16h_x}(d_{i+1,j+1}^y + d_{i+1,j-1}^y - d_{i-1,j+1}^y - d_{i-1,j-1}^y) \\ & + \frac{9}{16h_x h_y}(z_{i+1,j+1} - z_{i+1,j-1} - z_{i-1,j+1} + z_{i-1,j-1}). \end{aligned} \quad (1.20)$$

Nakoniec pre $i \in \{1, 3, \dots, I_l + 1\}$ a $j \in \{2, 4, \dots, J_l\}$

$$d_{i,j}^{xy} = \frac{3}{4h_y}(d_{i,j+1}^x - z_{i,j-1}^x) - \frac{1}{4}(d_{i,j+1}^{xy} - d_{i,j-1}^{xy}) \quad (1.21)$$

a pre $i \in \{2, 4, \dots, I_l\}$ a $j \in \{1, 3, \dots, J_l + 1\}$

$$d_{i,j}^{xy} = \frac{3}{4h_y}(d_{i,j+1}^x - z d_{i,j-1}^x) - \frac{1}{4}(d_{i,j+1}^{xy} - d_{i,j-1}^{xy}) \quad (1.22)$$

Označenie 1.15 Zaveďme dve označenia.

- Rovnice 1.14, 1.16 budeme súhrne označovať pojmom *resty*.
- Rovnice 1.20, 1.21, 1.22 budeme súhrne označovať pojmom *zmiešané resty*.

Modelové sústavy rovníc 1.13 a 1.15 majú takýto maticový tvar:

$$\begin{pmatrix} -14 & 1 & 0 & \cdots & 0 & 0 \\ 1 & -14 & 1 & \cdots & 0 & 0 \\ 0 & 1 & -14 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & -14 & 1 \\ 0 & 0 & 0 & \cdots & 1 & \mu \end{pmatrix} \cdot \begin{pmatrix} D_2 \\ D_4 \\ D_6 \\ \vdots \\ D_{v-2} \\ D_v \end{pmatrix} = \begin{pmatrix} \frac{3}{h}(Y_4 - Y_0) - \frac{12}{h}(Y_3 - Y_1) - D_0 \\ \frac{3}{h}(Y_6 - Y_2) - \frac{12}{h}(Y_5 - Y_3) \\ \frac{3}{h}(Y_8 - Y_4) - \frac{12}{h}(Y_7 - Y_5) \\ \vdots \\ \frac{3}{h}(Y_\nu - Y_{\nu-4}) - \frac{12}{h}(Y_{\nu-3} - Y_{\nu-5}) \\ \frac{3}{h}(Y_{\nu+\tau} - Y_{\nu-2}) - \frac{12}{h}(Y_{\nu-1} - Y_{\nu-3} - \theta D_{K+1}) \end{pmatrix}, \quad (1.23)$$

kde

$$\begin{aligned} \mu &= -15, \tau = 0, \theta = -4, \text{ a } \nu = N, & \text{ak } K \text{ je párne,} \\ \mu &= -14, \tau = 2, \theta = 1, \text{ a } \nu = N - 1, & \text{ak } N \text{ je nepárne,} \end{aligned} \quad (1.24)$$

a podľa toho o ktorú z modelových rovníc sa jedná, hodnoty N , D a Y zadávame následovne. Nech k z $1, \dots, K - 1$:

- $N = I$, $h = h_x$, $D_k = d_{k,j}^x$ a $Y_k = z_{k,j}$, pre rovnicu 1.13..
- $N = J$, $h = h_y$, $D_k = d_{i,k}^y$ a $Y_k = z_{i,k}$, pre rovnicu 1.15.

Analogicky vieme zostrojiť maticový tvar aj pre modelovú sústavu pre derivácie d^{xy} podľa 1.19.

Označenie 1.16 Špeciálne zaveďme označenia pre tieto hodnoty:

- Parciálne derivácie d^x a d^y počítané rovnicami 1.14 a 1.16 budeme nazývať *zbytkové derivácie*.
- Zmiešané parciálne derivácie d^{xy} počítané rovnicami 1.20, 1.21 a 1.22 budeme nazývať *zmiešané zbytkové derivácie*.

Lemma 1.17 *Splajn S s takto definovanými segmentami a deriváciami v uzloch vypočítaných podľa plného alebo podľa redukovaného algoritmu je triedy C^2 a interpoluje funkciu f .*

1.6 Konštrukcia splajnu

Teraz si ukážeme ako podľa zadaných uzlov a vypočítaných derivácií zostrojiť segmenty splajnu ako funkcie dvoch premenných v priestore. Predpokladajme $I \geq 3$ a $J \geq 3$ z $\mathbb{N} \cup \{0\}$, rovnomerné uzly u_0, \dots, u_{I-1} a v_0, \dots, v_{J-1} , funkciu dvoch premenných $f : [u_0, u_{I-1}] \times [v_0, v_{J-1}] \rightarrow \mathbb{R}$. Úlohou je zostrojiť splajn S triedy C^2 interpolujúci funkciu f . Položme

- funkčné hodnoty splajnu $z_{0,0}, \dots, z_{I-1,0}, \dots, z_{0,J-1}, \dots, z_{I-1,J-1}$,
- x -ové derivácie splajnu $d_{0,0}^x, \dots, d_{I-1,0}^x, \dots, d_{0,J-1}^x, \dots, d_{I-1,J-1}^x$,
- y -ové derivácie splajnu $d_{0,0}^y, \dots, d_{I-1,0}^y, \dots, d_{0,J-1}^y, \dots, d_{I-1,J-1}^y$,
- x -ové derivácie splajnu $d_{0,0}^{xy}, \dots, d_{I-1,0}^{xy}, \dots, d_{0,J-1}^{xy}, \dots, d_{I-1,J-1}^{xy}$,

a inicializujeme ich podľa 1.4.

Ďalším krokom je vypočítanie zvyšných hodnôt derivácií buď plným alebo redukovaným algoritmom. Výstupom oboch sú rovnaké hodnoty derivácií, ktoré nám zaručia spojitosť a hladkosť medzi jednotlivými segmentami budovaného splajnu, čím podľa definície bude patriť práve do triedy C^2 .

Teraz už máme všetky potrebné hodnoty aby sme mohli definovať segmenty splajnu S podľa definície 1.8. Pre každé i z $\{0, \dots, I-2\}$ a j z $\{0, \dots, J-2\}$ položíme segment $S_{i,j} : [u_i, v_j] \times [u_{i+1}, v_{j+1}] \rightarrow \mathbb{R}$ vzťahom:

$$S_{i,j}(x, y) = \lambda^T(x, u_i, u_{i+1}) \cdot \Phi(u_i, u_{i+1}, v_j, v_{j+1}) \cdot \lambda(y, v_j, v_{j+1}), \quad (1.25)$$

kde

$$\lambda(t, t_0, t_1) = \left(\frac{(t-t_1)^2(1-2\frac{t-t_0}{t_0-t_1})}{(t_0-t_1)^2}, \frac{(t-t_0)^2(1-2\frac{t-t_1}{t_1-t_0})}{(t_1-t_0)^2}, \frac{(t-t_1)^2(t-t_0)}{(t_0-t_1)^2}, \frac{(t-t_0)^2(t-t_1)}{(t_1-t_0)^2} \right)^T, \quad (1.26)$$

$$\begin{aligned} \Phi(t_0, t_1, s_0, s_1) = \\ \left(\frac{(t-t_1)^2(1-2\frac{t-t_0}{t_0-t_1})}{(t_0-t_1)^2}, \frac{(t-t_0)^2(1-2\frac{t-t_1}{t_1-t_0})}{(t_1-t_0)^2}, \frac{(t-t_1)^2(t-t_0)}{(t_0-t_1)^2}, \frac{(t-t_0)^2(t-t_1)}{(t_1-t_0)^2} \right)^T, \end{aligned} \quad (1.27)$$

Kapitola 2

Zrýchlenie

Časová zložitosť oboch algoritmov je rovnaká a síce $O(I \cdot J)$. Redukovaný algoritmus je však teoreticky, a aj prakticky (2.3) rýchlejší, navyše s menšími pamäťovými nárokmi. Keďže porovnávame asymptoticky rovnako rýchle algoritmy, musíme brať do úvahy vplyv „nízkoúrovňových“ operácií, konkrétne rýchlosti výpočtu sčítania, násobenia, delenia a na koniec aj samotný vplyv pamäťových operácií čítania a zapisovania. V tejto kapitole si spočítame počty týchto operácií.

Na moderných procesoroch v prípade matematických operácií s plávajúcou desatinnou čiarkou platí, že sčítanie a násobenie sú podobne rýchle, pričom delenie je niekoľkonásobne pomalšie. Pamäťové operácie sú pritom v závislosti od konkrétneho typu operačnej pamäte približne dvadsaťnásobne pomalšie ako sčítanie, resp. násobenie.¹

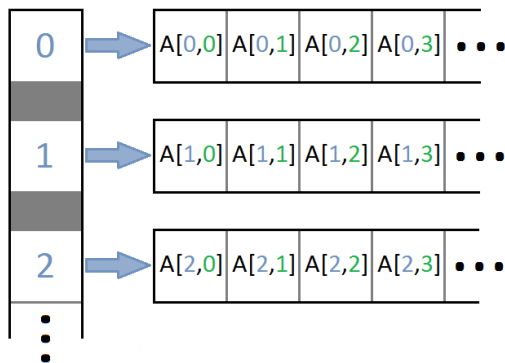
2.1 Procesorová architektúra

Predtým ako začneme s počítaním operácií je nutné objasniť si ako moderné procesory narábajú s dátami a ako funguje aplikovanie výpočtov. Architektúry CPU prešli za posledné desaťročia značným vývojom. V dnešnej dobe už nemožno zvyšovať výpočtovú rýchlosť hrubou silou zvyšovaním frekvencie. Moderné procesorové mikroarchitektúry často používajú rôzne optimalizačné „triky a finty“ ako dosiahnuť zlepšenie výkonu a ktoré majú značný vplyv na reálne rýchlosti algoritmov. Dvomi takými optimalizáciami sú takzvané kešovanie a inštrukčný paralelizmus.

¹V praxi sa aj výpočty rádovo tisícov uzlov často zmestia do vyrovnávacej pamäte procesora (*cache*). V tom prípade sú pamäťové operácie len 1×-10× pomalšie ako sčítanie.

2.1.1 Kešovanie

Pri návrhu algoritmu je vhodné si uvedomiť tvar dátových štruktúr do ktorých pri reálnej implementácii budeme ukladať funkčné hodnoty, uzly a derivácie. Naša implementácia v prípade kriviek používa klasické vektory, teda polia. povrchoch sa hodnoty reprezentujú maticami. Maticu je možné interpretovať buď ako „pole polí“ respektíve „vektor vektorov“ (anglicky „jagged array“), alebo ako jedno spojitú pole, kde prvok na i -tom riadku a j -tom stĺpci má v takejto reprezentácii index $n \cdot i + j$, kde n je počet stĺpcov. My budeme v meraniach a výpočtoch uvažovať práve prvú reprezentáciu znázornenú na obrázku nižšie.



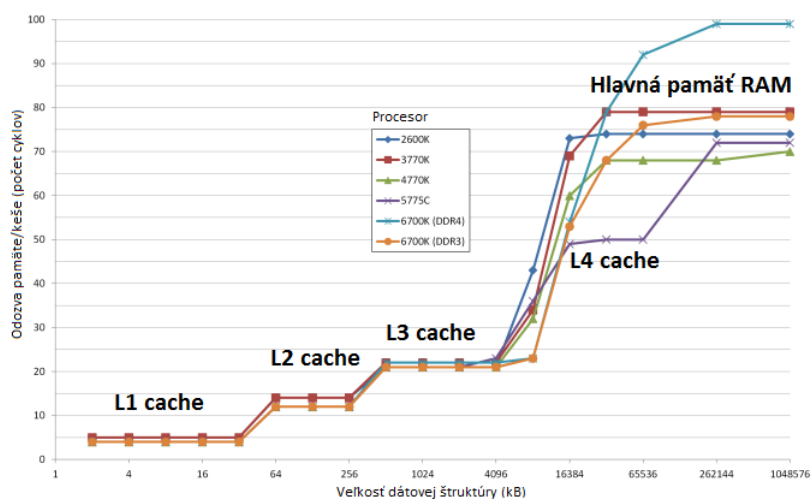
Obr. 2.4: Vizuálna reprezentácia dátovej štruktúry „pole polí“.

Operačná pamäť v dnešných počítačoch je rádovo pomalšia ako procesor, ktorý musí čakať desiatky až stovky strojových cyklov ak chce načítať hodnotu z pamäte. Moderné procesory preto používajú systém malých, ale rýchlych vyrovnávacích pamätí tzv. *cache* do ktorých sa prednačítavajú dáta a strojové inštrukcie programu z operačnej pamäte, v ideálnom prípade skôr ako ich procesor bude potrebovať. Toto kešovanie je plne automatický proces riadený samotným CPU. Programátor má len nepriame možnosti ovplyvnenia tohto procesu, napríklad vhodným výberom dátových štruktúr a podobne.

Uvažujme maticu $m \times n$ typu pole polí ako z obrázka vyššie. Ak z matice načítame prvok $a_{0,0}$ tak je vysoko pravdepodobné, že procesor následne bude mať nakešované aj prvky $a_{0,1}$, $a_{0,2}$ pretože sú z jedného poľa a teda sú v pamäti „vedľa seba“ (CPU kešuje pamäť po spojitých blokoch), ale už nemusí mať kešovaný prvok $a_{1,0}$ lebo tento je už z iného poľa a teda z úplne inej oblasti pamäte. Z toho v praxi vyplýva, že výraz $a_{0,0} + a_{0,1}$ procesor vyhodnotí rýchlejšie ako výraz $a_{0,0} + a_{1,0}$.

Pozorný čitateľ si môže všimnúť, že v prípade použitia druhej reprezentácie matice, teda jedného veľkého spojitého poľa dĺžky $m \cdot n$, pravdepodobnosť kešovania aj prvku $a_{1,0}$ by sa zvýšila, keďže je v tomto prípade sa bude nachádzať v tom istom bloku pamäte ako prvok $a_{0,0}$ iba o n prvkov (t.j. jeden celý stĺpec) ďalej. Toto je pravda a všeobecne sa druhá menovaná reprezentácia považuje za efektívnejšiu vzhľadom na procesorový čas a pamäť počítača, ale to platí len pre malé matice najmä s malým počtom stĺpcov n .

V aplikácii sme totiž vyskúšali obe varianty reprezentácie a pre matice s rádovo 1000×1000 prvkami sa ako efektívnejšia ukazuje práve prvá reprezentácia, teda „pole polí“. Totiž ak jeden prvok matice má 8 bajtov (veľkosť číselného typu *double* vo väčšine jazykoch), tak matica zaberá v pamäti približne 7,63 megabajtov. Pri matici typu pole polí máme 1000 polí každé zaberajúce práve 7,81 kilobajtov ($1 \text{ kB} = 1024 \text{ B}$, $1 \text{ MB} = 1024 \text{ kB}$, atď.). Pre CPU je kešovanie takýchto polí podstatne jednoduchšie ako v prípade matice reprezentovanej jedným poľom, ktoré samotné má veľkosť práve 7,63 MB, čo presahuje kapacitu väčšiny vyrovnávacích pamätí. Obrázok nižšie názorne ukazuje vzťah medzi veľkosťou dátovej štruktúry v kB a počtom strojových cyklov, ktoré procesor musí čakať pre prístup k jej prvkom. Ako vidieť do pri dosiahnutí veľkosti 32 kB, dáta sú namiesto veľmi rýchlej vyrovnávacej pamäte L1 kešované do trojnásobne pomalšej vyrovnávacej pamäte L2 atď..



Obr. 2.5: Graf rýchlosti prístupu k dátam rôznych veľkostí na moderných CPU[7].

2.1.2 Inštrukčný paralelizmus

V posledných rokoch sa v počítačovej vede stále viac spomína pojem paralelizmu. Dnešné procesory architektúry x86 využívajú až štyri úrovne paralelizácie výpočtov. Na najvyššej úrovni hovoríme o návrhu kedy je procesor zložený z niekoľkých autonómne pracujúcich „podprocesorov“ nazývaných *jadrá*. Tie zdieľajú systémové zbernice a pamäť pričom majú väčšinou vlastnú L1 alebo L2 cache. Každé jadro môže spracovávať na sebe nezávislé procesy prípadne jeden proces môže byť rozdelený do takzvaných vlákien, kde každé môže byť spracovávané iným jadrom. V tomto prípade hovoríme o takzvanom *vláknovom paralelizme* s ktorým sa stretávame najmä pri programovaní vo vyšších jazykoch. V našej implementácii testujeme vláknovo paralelizované aj klasické sériové verzie algoritmov.

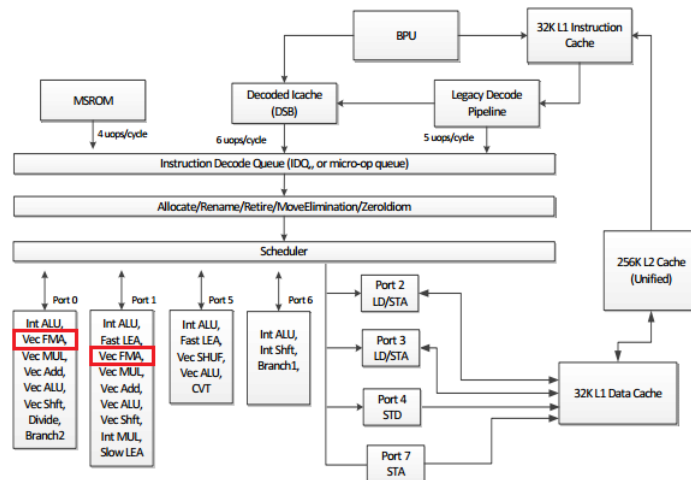
Ďalšie úrovne paralelizácie majú spoločné označenie *inštrukčný paralelizmus*. Pod týmto pojmom rozumieme *superskalárnosť*, *pipelining* a *vektORIZÁCIA*. Pri tejto úrovni paralelizmu má programátor len obmedzené možnosti jeho ovplyvnenia, všetku „ťažkú“ prácu obstará prekladač a pri behu aplikácie zasa inštrukčný plánovač v samotnom procesorovom jadre.

Hlavný vplyv na výkon plného a najmä redukovaného algoritmu má superskalárnosť. Najprv si vysvetlíme čo tento pojem znamená. Jadro je tvorené niekoľkými až desiatkami špecializovanými jednotkami ako sú aritmeticko logické jednotky (ALU), bitové posuvníky (Shift), numerické koprocessory (FPU) a podobne. Narozdiel od celých jadier pri vláknovom paralelizme, tieto jednotky vo všeobecnosti nedokážu fungovať samostatne a simultánne spracovávať viacero vlákien alebo procesov². Vedia iba v rámci jedného vlákna, za splnenia určitých podmienok, spracovávať niekoľko inštrukcií naraz. Nutnou podmienkou je napríklad vzájomná nezávislosť niekoľkých po sebe idúcich inštrukcií, teda keď výsledok jednej inštrukcie nezávisí na výsledku predchádzajúcej.

Tu ale paralelizácia nekončí. Aj samotné jednotky totiž dokážu simultánne spracovávať viac než jeden dátový vstup. Na tejto úrovni rozlišujeme medzi vektorizáciou, a pipeliningom. Prvá menovaná technika umožňuje jednu inštrukciu aplikovať na celé vektory, resp. polia. Z povahy výpočtov v našej práci nás ale vektorizácia nemusí zaujímať čo si vysvetlíme neskôr. Pipelining je založený na myšlienke princípu fungovania výpočtových jednotiek podobne ako výrobná linka vo fabrike, kde nový výrobok na linku vstúpi skôr ako je predchádzajúci dokončený. Teda aj väčšina výpočtových jednotiek dokáže s novým výpočtom začať ešte pred dokončením práve prebiehajúcej operácie.

Spomenuli sme majoritný vplyv superskalárnosti. Na obrázku 2.6 vidíme

²Existujú technológie ako napríklad Hyper-threading umožňujúce za určitých podmienok uplatniť vláknový paralelizmus aj v rámci jediného superskalárneho jadra.



Obr. 2.6: Schéma výpočtového jadra mikroarchitektúry Intel Skylake[8].

schému jedného jadra procesorovej mikroarchitektúry Intel Skylake, ktorá je ku dňu písania práce najmodernejšou bežne dostupnou procesorovou generáciou a na ktorej testujeme náš zrýchlenie redukovaného algoritmu. Jadrá (nielen) tejto architektúry sú vybavené dvomi sčítačko-násobičkami čísiel s plávajúcou desatinnou čiarkou, ktoré sú na obrázku zvýraznené červenou farbou. Pointa superskalárnosti v tomto prípade spočíva v možnosti výpočet výrazov typu $a_0 \circ a_1 \circ \dots \circ a_n$, kde $\circ \in \{+, \cdot\}$ rozložiť medzi dve jednotky a dosiahnuť dvojnásobného zrýchlenia. Toto má značný vplyv na rýchlosť najmä redukovaného algoritmu pri počítaní pravých strán trojdiagonálnych rovníc, ktoré sú na prvý pohľad zložitejšie ako v prípade plného algoritmu. Keďže procesor má ale iba jednu deličku, pre výrazy typu $a_0/a_1/\dots/a_n$ takýto trik fungovať nebude.

Procesorové jadro na obrázku vyššie má svojich 29 jednotiek prístupných cez osem zberníc (porty 0 až 7). To konkrétne v prípade procesorov Skylake znamená, že dokážu naraz využívať maximálne osem svojich výpočtových jednotiek za splnenia určitých podmienok.

Málokedy ale proces alebo vlákno obsahuje vždy osmice vzájomne nezávislých inštrukcií. V situácií kedy jeden proces (vlákno) nemôže využiť všetky jednotky jadra, niektoré procesory umožňujú nevyužívané porty priradiť inému procesu (vláknu). Táto technika sa nazýva *Simultaneous Multi-Threading* (ďalej len SMT). V literatúre sa, v prípade procesoroch Intelu, často môžeme stretnúť s názvom *Hyper-Threading*. Procesory AMD majú implementovaný odlišný variant tejto techniky zvaný *Clustered Multi-Threading*, kde

namiesto klasických jadier je procesor vybavený tzv. modulmi, kde každý modul má dve samostatné „oklieštené“ jadrá schopné spracovávať iba celočíselné operácie a jedinú FPU jednotku spoločnú pre oba jadrá. V praxi teda SMT predstavuje akýsi medzistupeň medzi vláknovým a inštrukčným paralelizmom, kedy procesor operačnému systému a bežiacim procesom hlási väčší počet jadier akým je v skutočnosti vybavený, pričom interne na úrovni mikroarchitektúry sa jedná iba o ďalšiu techniku superskalárnosti.

SMT má výrazne pozitívny vplyv na výkon procesora ak procesy (vlákna) zdieľajúce jedno jadro nepožadujú prístup k tým istým jednotkám. V opačnom prípade dochádza k javu kedy jeden proces (vlákno) musí čakať na uvoľnenie jednotky druhým procesom (vláknom) a vtedy môže dôjsť k zníženiu výkonu oproti identickému procesor s neaktívnym SMT. V praktických meraniach má SMT na implementované algoritmy negatívny vplyv.

2.1.3 Rýchlosť aritmetických operácií

Procesory majú mnoho aritmetických jednotiek špecializovaných na určitý typ operácie. Je prirodzené predpokladať, že tieto jednotky budú pracovať navzájom rozličnými rýchlosťami. V prípade výpočtových algoritmov má pochopiteľne okrem výberu vhodných dátových štruktúr vplyv najmä rýchlosť vykonania základných matematických operácií a síce sčítania, odčítania, násobenia a delenia.

Najlepším zdrojom ako zistiť rýchlosť týchto operácií je dokumentácia inštrukčných sád procesorov. Inštrukčná sada x86 sa od svojho prvotného uvedenia v roku 1978 dočkala mnohých rozšírení. Moderné procesory obsahujú mnoho spôsobov ako napríklad vynásobiť dve čísla. Keďže nie je v silách otestovať všetky možné rozšírenia sady, zvolili sme si jedno konkrétne rozšírenie a síce Streaming SIMD Extensions (skrátene SSE), konkrétne vo verzii 4 (SSE4). Sady z rodiny SSE sú v čase písania práce najpoužívanejšími sadami (najmä SSE2) ktoré sú podporované prakticky všetkými nemožeálnymi procesormi.

V súčasnosti už existuje modernejšia náhrada tejto sady zvaná Advanced Vector Extensions (skratka AVX), ktorej hlavný prínos spočíva vo vylepšených vektorových operáciach. Vektorové operácie vyžadujú aby výpočty využívajúce jednotlivé prvky vektora boli navzájom nezávislé. To nie je prípad ani jedného z testovaných algoritmov. Nakonfigurovaním prekladača na generovanie týchto inštrukcií by sme iba znemožnili beh aplikácie na polovici z testovaných procesorov. Praktické zrýchlenie sme testovali na šiestich procesoroch architektúry x86, pričom sme pokryli väčšinu mikroarchitektúr v rozmedzí rokov 2007 až 2015.

V nasledujúcej tabuľke[9] uvidíme rýchlosti štyroch základných mate-

matických operácií v rámci šiestich testovaných mikroarchitektúr. Tabuľka obašhuje tieto stĺpce.

- **Architektúra (rok)**

Testovaná mikroarchitektúra a rok jej uvedenia na trh. Architektúry sú zoradené abecedne podľa výrobcu a následne podľa roku vydania.

- **Odozva**

Počet strojových cyklov potrebných na vykonanie inštrukcie.

- **Inverzný prietok**

Počet strojových cyklov ktoré je nutné čakať kým je daná výpočtová jednotka schopná zopakovať inštrukciu. V prípade operácií sčítania, odčítania a násobenia je tento počet menší ako odozva. To znamená, že aritmetické sčítačky a násobičky sú schopné vďaka technike pipelining-u začať nový výpočet ešte pred dokončením aktuálneho výpočtu.

Architektúra (rok)	Odozva				Inverzný prietok			
	+	−	·	÷	+	−	·	÷
AMD Piledriver (2012)	5-6	5-6	5-6	9-27	0,5	0,5	0,5	5-10
Intel Penryn (2007)	3	3	5	6-21 ²	1	1	1	5-20 ²
Intel Nehalem (2008)	3	3	5	7-22	1	1	1	7-22
Intel Sandy Bridge (2011)	3	3	5	10-22	1	1	1	10-22
Intel Haswell (2013)	3	3	5	10-20	1	1	0,5	8-14
Intel Skylake (2015)	4	4	4	13-14	0,5	0,5	0,5	4

Podľa tabuliek vidno, že operácie sčítania a odčítania sú rovnako rýchle čo sa pochopiteľne dalo očakávať. Tieto dve operácie preto budeme spoločne označovať symbolom \pm . Od tejto chvíle ak spomenieme operáciu sčítania tak tým budeme súčasne myslieť aj operáciu odčítania. Ako vidieť zďaleka najpomalšie je práve delenie.

Pre zaujímavosť si môžeme ukázať praktické výsledky z testovacej aplikácie. Merali sme operácie na dvoch 512 prvkových poliach, pričom aby sme dostali „rozumne“ dlhé časy výpočty boli opakované 500000 krát. V tabuľke sú namiesto mikroarchitektúr uvedené konkrétne modely procesorov, pričom ich poradie zodpovedá poradiu z minulej tabuľky. Všetky údaje sú v milisekundách.

²Odozva je nadobúda menšie hodnoty ak je deliteľ celý.

Procesor	\pm	\cdot	\div
AMD FX-6300	176	173	673
Intel Core 2 Duo E8200	namerat	u	c. toroka
Intel Core i5 650	198	224	1758
Intel Core i3 2350M	233	231	2462
Intel Core i5 4440	120	119	1087
Intel Core i7 6700K	99	99	257

V praxi vidno, že sčítavanie a násobenie môžeme považovať za podobne rýchle operácie. Delenie býva až desať násobne pomalšie. To je spôsobné veľmi vysokou hodnotou inverzného prietoku v prípade opakovane vykonávanom delení čo nastáva napríklad ak chceme vydeliť dva vektory. V prípade našich algoritmov ale dochádza len k jednému deleniu na mnoho sčítaní a násobení. Z toho dôvodu môžeme inverzný prietok v prípade delenia zanedbať a v kontexte implementovaných algoritmov delenie považovať podľa jeho odozvy za približne **trojnásobne** pomalšie ako násobenie.

2.2 Teoretické zrýchlenie

Dôvod, prečo je náš redukovaný algoritmus rýchlejší je práve fakt, že pri ňom dochádza k značne menšiemu počtu delení, ktoré je oproti ostatným trom operáciám výrazne pomalšie. V ďalšej sekcii sy spočítame jednotlivé operácie a vypočítame teoretické zrýchlenie redukovaného algoritmu.

Spočítajme počet elementárnych matematických operácií sčítania, násobenia a delenia pre oba algoritmy. To budeme počítat tak, že si zdefinujeme „procedúry“ ktoré predstavujú jednotlivé časti algoritmu. Potom z nich zostavíme pseudokód pre oba algoritmy. Začnime teda počítat počty operácií. Keďže všetky procesory použité pri meraní obsahujú dve jednotky pre operácie s plávajúcou desatinnou čiarkou tak pri počítaní budeme uvažovať, že pri výrazoch s viacerými operáciami vezmeme do úvahy len polovičný počet operácií. Najprv položme procedúry predstavujúce LU dekompozíciu spoločné pre oba postupy.

- Procedúra *InicalizujLU* inicializuje hodnoty r_0, \dots, r_{K-1} a b z LU dekompozície 1.5 podľa 1.12 v prípade plného algoritmu, respektíve podľa 1.23 v prípade redukovaného algoritmu.
- Procedúra *VyriešLU* vypočíta sústavu rovníc na základe hodnôt pravej strany poskytnutými *InicalizujLU*.

2.2.1 Počty pre plný algoritmus

Počty základných aritmetických a pamäťových operácií pre vyššie uvedené procedúry môžeme zhrnúť do tabuľky, pričom uvažujeme počty v našej ukážkovej implementácii³. Riadky tabuľky predstavujú jednotlivé procedúry a stĺpce udávajú počet vykonaných operácií, pre základné matematické operácie, kde K je počet neznámych. Podotýkam, že operáciu odčítania budeme uvažovať ako sčítanie. Operácie MEM predstavuje kopírovanie vypočítaných derivácií do výslednej matice uzlov. Ďalej musíme odlíšiť operácie vykonané na spojitých a nespojitých dátach podľa sekcie 2.1.1 o kešovaní. V prípade nespojitých dát má značný vplyv na meranie práve prístup do nekešovaných častí pamäte. To či matematické operácie v nasledujúcich procedúrach budú kešované alebo nie závisí od kontextu v akom sú volané. Zatiaľ budeme predpokladať, že všetky operácie sú vykonávané na spojitých dátach.

Procedúra	\pm	\cdot	\div	M
<i>InicializujLU</i>	K	K	0	0
<i>VyriešLU</i>	$4K$	$3K$	K	K

Uvažujme procedúry predstavujúce implementáciu plného algoritmu:

- Procedúra *VypočítajDx* vypočíta parciálne derivácie d^x pomocou procedúry *VyriešLU*. Vstupné hodnoty vezmeme z 1.8. Jedno volanie *VyriešLU* vypočíta derivácie d^x pre jeden stĺpec. Teda procedúra musí počítať LU pre každý stĺpec, ktorých je J .
- Procedúra *VypočítajDxy* vypočíta parciálne derivácie d^{xy} pomocou procedúry *VyriešLU*. Vstupné hodnoty vezmeme z 1.9. Jedno volanie *VyriešLU* vypočíta derivácie d^{xy} pre jeden stĺpec. Procedúra musí počítať LU pre prvý a posledný stĺpec.
- Procedúra *VypočítajDy* vypočíta parciálne derivácie d^y pomocou procedúry *VyriešLU*. Vstupné hodnoty vezmeme z 1.10. Jedno volanie *VyriešLU* vypočíta derivácie d^y pre jeden riadok. Teda procedúra musí počítať LU pre každý riadok, ktorých je I .
- Procedúra *VypočítajDyx* vypočíta parciálne derivácie d^{xy} pomocou procedúry *VyriešLU*. Vstupné hodnoty vezmeme z 1.11. Jedno volanie *VyriešLU* vypočíta derivácie d^{xy} pre jeden riadok. Teda procedúra musí počítať LU pre každý riadok, ktorých je I .

³Repozitár so zdrojovými kódmi k aplikáciám možno nájsť na adrese <https://github.com/vildibald/VKDiplom-master>

- Procedúra *VypočítajPlný* vypočíta na základe vstupných hodnôt pre de Boorovu interpoláciu 1.4 zovolaním procedúr *VypočítajDx*, *VypočítajDxy*, *VypočítajDy* a *VypočítajDyx*.

Nech I značí počet uzlov na osi x a J značí počet uzlov na osi y . Všetky počty sú v tvare $a \cdot IJ + b \cdot I + c \cdot J + d$. Pre zjednodušenie budeme počty, tam kde je koeficient a nenulový, uvádzať v tvare $a \cdot IJ$. Keďže počty rastú kvadraticky tak pre veľké I a J bude odchýlka zanedbateľná. Pre plný algoritmus teda dostaneme tieto počty

Procedúra	\pm	\cdot	\div	M
<i>VypočítajDx</i>	$4IJ$	$3IJ$	IJ	IJ
<i>VypočítajDxy</i>	$8I$	$6I$	$2I$	I
<i>VypočítajDy</i>	$4IJ$	$3IJ$	IJ	IJ
<i>VypočítajDyx</i>	$4IJ$	$3IJ$	IJ	IJ
<i>VypočítajPlný</i>	$12IJ$	$9IJ$	$3IJ$	$3IJ$

2.2.2 Počty pre redukovaný algoritmus

Redukovaný spôsob na počítanie derivácií využíva iný tvar trojdiagonálnej sústavy rovníc. Zmeňme počty operácií procedúr *VyriešLU* a *InicalizujLU*. Ďalej položíme dve nové pomocné procedúry *InicalizujZmiešLU*, ktorá inicializuje hodnoty r_0, \dots, r_{K-1} a b z LU dekompozície 1.5 pre zmiešané derivácie d^{xy} a k nej príslušnú *VyriešZmiešLU*.

Procedúra	\pm	\cdot	\div	M
<i>InicalizujLU</i>	$1,5K$	K	0	0
<i>VyriešLU</i>	$4,5K$	$3K$	K	K

V prípade počtov operácií v procedúrach pre zmiešané derivácie nepristupujeme k niektorým dátam spojitým, navyše čítame dáta z niekoľkých vektorov naraz čo má malý negatívny vplyv na výkon. Ten sa pri testovaní ukázal byť malý v rozsahu 5% až 20% v závislosti od konkrétneho procesora. Aby sa nám to v nasledujúcej časti nemýlilo tak počty uvedieme v samostatných tabuľkách kde v prvej budú uvedené len počty operácií na spojitých dátach a následne na nespojitých dátach. Pre spojitá dáta máme

Procedúra	\pm	\cdot	\div	M
<i>InicalizujZmiešLU</i>	0	0	0	0
<i>VyriešZmiešLU</i>	$3K$	$3K$	K	0

a pre operácie na nespojitých dátach dostaneme

Procedúra	+	·	÷	M
<i>InicalizujZmiešLU</i>	16,5K	8,5K	0	0
<i>VyriešZmiešLU</i>	16,5K	8,5K	0	K

Analogicky položíme procedúry predstavujúce implementáciu redukovaného algoritmu:

- Procedúra *VypočítajDxResty* vypočíta zvyšné parciálne derivácie d^x podľa 1.14.
- Procedúra *VypočítajDx* vypočíta parciálne derivácie d^x pomocou procedúry *VyriešLU*. Vstupné hodnoty vezmeme z 1.13. Jedno volanie *VyriešLU* vypočíta derivácie na párnych riadkoch d^x pre jeden stĺpec. Teda procedúra musí počítať LU pre každý stĺpec, ktorých je J . Zvyšné derivácie dopočítame procedúrou *VypočítajResty*.
- Procedúra *VypočítajDyResty* vypočíta zvyšné parciálne derivácie d^y podľa 1.16.
- Procedúra *VypočítajDy* vypočíta parciálne derivácie d^y pomocou procedúry *VyriešLU*. Vstupné hodnoty vezmeme z 1.15. Jedno volanie *VyriešLU* vypočíta derivácie na párnych stĺpcoch d^y pre jeden riadok. Teda procedúra musí počítať LU pre každý riadok, ktorých je I . Zvyšné derivácie dopočítame procedúrou *VypočítajResty*.
- Procedúra *VypočítajDxy* vypočíta parciálne derivácie d^{xy} pomocou procedúry *VyriešLU*. Vstupné hodnoty vezmeme z 1.17 a 1.18. Jedno volanie *VyriešLU* vypočíta na párnych riadkoch(stĺpcoch) derivácie d^{xy} pre jeden stĺpec(riadok). Procedúra musí počítať LU pre prvý a posledný stĺpec a pre prvý a posledný riadok.
- Procedúra *VypočítajZmiešResty* vypočíta zvyšné parciálne derivácie d^{xy} podľa 1.20, 1.21 a 1.22.
- Procedúra *VypočítajDyx* vypočíta parciálne derivácie d^{xy} pomocou procedúry *VyriešLU*. Vstupné hodnoty vezmeme z 1.19, pričom na ich inicializovanie do LU dekompozície použijeme procedúru *InicalizujZmiešLU*. Jedno volanie *VyriešLU* vypočíta na párnych stĺpcoch d^{xy} pre jeden riadok. Procedúra musí počítať LU pre každý párny riadok, ktorých je I . Zvyšné derivácie dopočítame procedúrou *VypočítajZmiešResty*.

- Procedúra *VypočítajRedukovaný* vypočíta na základe vstupných hodnôt pre de Boorovu interpoláciu 1.4 postupným vykonaním predchádzajúcich siedmich procedúr.

Najprv si spočítajme procedúry pre resty.

Procedúra	\pm	\cdot	\div	M
<i>VypočítajDxResty</i>	$1,5IJ$	IJ	0	0
<i>VypočítajDyResty</i>	$1,5IJ$	IJ	0	0
<i>VypočítajZmiešResty</i>	$7IJ$	$2,5IJ$	0	0

Opäť pre zjednodušenie budeme počty v tvare $a \cdot IJ + b \cdot I + c \cdot J + d$ zanedbávať na $a \cdot IJ$. Ak hodnota I značí počet uzlov na osi x a hodnota J značí počet uzlov na osi y tak pre redukovaný algoritmus dostaneme nasledujúce počty nad kešovanými dátami

Procedúra	\pm	\cdot	\div	M
<i>VypočítajDx</i>	$3IJ$	$2IJ$	$0,5IJ$	0
<i>VypočítajDy</i>	$3IJ$	$2IJ$	$0,5IJ$	0
<i>VypočítajDxy</i>	$6I + 8J$	$7I + 9J$	$2I$	$I + J$
<i>VypočítajDyx</i>	$2,875IJ$	$1,625IJ$	$0IJ$	0
<i>VypočítajRedukovaný</i>	$8,875IJ$	$5,625IJ$	$1,25IJ$	0

V prípade *VypočítajDyx* ešte musíme započítať operácie nad nespojitými dátami

Procedúra	\pm	\cdot	\div	M
<i>VypočítajDx</i>	0	0	0	$0,5IJ$
<i>VypočítajDy</i>	0	0	0	$0,5IJ$
<i>VypočítajDyx</i>	$4,125IJ$	$2,125IJ$	0	$0,25IJ$
<i>VypočítajRedukovaný</i>	$4,125IJ$	$2,125IJ$	0	$1,25IJ$

2.2.3 Pamäťové nároky

Redukovaný algoritmus by nemal byť lepší len v počte aritmetických operácií, ale taktiež priniesť menšie pamäťové nároky vyplývajúce z polovičnej veľkosti sústav rovníc riešených LU dekompozíciou. Uvažujme maticu uzlov veľkosti I a J . Potrebujeme si pamätať $I \cdot J$ funkčných hodnôt v uzloch, derivácií podľa x , y a zmiešaných derivácií. Samotné uzly si pamätať nemusíme lebo uvažujeme len uniformné splajny. Teda stačí si nám uložiť iba uzly u_0, u_{I-1} ,

v_0, v_{J-1} . Zvyšné uzly si vieme vypočítať vzorcom $u_i = u_0 + |u_{I-1} - u_0| \cdot \frac{i}{I}$ pre ľubovoľné i z $\{1, \dots, I-2\}$, respektíve $v_j = v_0 + |v_{J-1} - v_0| \cdot \frac{j}{J}$ pre ľubovoľné j z $\{1, \dots, J-2\}$. Potrebujeme si teda pamätať aspoň $4 \cdot I \cdot J + 4$ čísel pre funkčné hodnoty uzlov.

Počítanie LU dekompozície všeobecne podľa algoritmu 1 potrebuje päť vektorov, kde tri predstavujú trojdiagonálu, jeden pravú stranu, v ktorom na konci bude výsledok. Nám ale stačí uvažovať iba procedúru pre náš špeciálny tvar trojdiagonálnej sústavy podľa algoritmu 2. Pri použití tejto priestorovo optimalizovanej varianty výpočtu LU dekompozície potrebujeme na vypočítanie $I \cdot J$ uzlov práve $I \cdot J + 2I + 2J$ pamäte.

2.2.4 Zhrnutie

Teraz máme spočítané jednotlivé matematické a pamäťové operácie.

V záujme prehľadnosti v rámci tejto podsekcie zavedme tieto označenia

Označenie 2.1 Označme dve triedy dátových štruktúr symbolmi:

- \succ označuje spojité dátové štruktúry ako napríklad vektory.
- $\dot{\cdot}$ označuje nespojité dátové štruktúry ako napríklad spájané zoznamy alebo vektory referencií na konkrétne dáta.

Nech $t_{\text{operácia}}^{\text{dáta}}$ udáva dobu trvania nejakej operácie na nejakých dátach. Napríklad $t_+^{\text{spojité}}$ udáva čas sčítania dvoch rovnako veľkých spojitých kešovaných vektorov. Na začiatku sekcie sme zvolili predpoklad, že pri číselných typoch s pohyblivou čiarkou sú operácie sčítania (odčítania) a násobenia rovnako rýchle. Na všetkých testovaných procesoroch je totožne rýchla aj pamäťová operácia kopírovania pri spojitých dátach. Teda $t_+ = t_x = t_{\text{MEMc}}$. Položme si teraz pomery udávajúce pomery dôb trvania jednotlivých operácií.

- $\gamma = \frac{t_+^{\dot{\cdot}}}{t_+^{\succ}}$
- $\eta = \frac{t_M^{\dot{\cdot}}}{t_M^{\succ}}$
- $\iota = \frac{t_+^{\dot{\cdot}}}{t_+^{\dot{\cdot}}}$

Celková cena algoritmu je teda súčet počtu všetkých operácií z tabuliek 2.2.1 pre plný algoritmus, respektíve 2.2.2 a 2.2.2 pre redukovaný algoritmus z predchádzajúcich podsekcí. Počty operácií delenia na kešovaných dátach vynásobíme γ , počty nekešovaných pamäťových operácií vynásobíme η , počty operácií sčítania a násobenia na nespojitých dátach vynásobíme ι .

Pre plný algoritmus teda dostávame

$$25,5IJ + 2\gamma IJ + \eta IJ + 2\iota IJ + \lambda \iota IJ, \quad (2.1)$$

a pre redukovaný máme

$$14,375IJ + 0,75\gamma IJ + 0,5\eta IJ + 5,84\iota IJ + 0,5\lambda \iota IJ. \quad (2.2)$$

Ak chceme určiť zrýchlenie redukovaného algoritmu stačí vydeliť 2.1 a 2.2 čím dostaneme

$$\frac{25,5IJ + 2\gamma IJ + \eta IJ + 2\iota IJ + \lambda \iota IJ}{14,375IJ + 0,75\gamma IJ + 0,5\eta IJ + 5,84\iota IJ + 0,5\lambda \iota IJ}. \quad (2.3)$$

Definícia 2.2 Výraz 2.3 nazveme *pomer rýchlostí*.

Príklad 2.3 Ak budeme predpokladať $\gamma = 3$, to jest operácia delenia je trojnásobne pomalšia ako operácia sčítania, ďalej $\eta = 4$ (podľa obrázka 2.5), $\iota = 1$, $\lambda = 1$ a budeme uvažovať maticu uzlov s počtom $I \times J = 1000 \times 1000$ tak pomer rýchlostí bude 1,18, teda plný algoritmus je 1,18 krát pomalší ako redukovaný.

2.3 Zrýchlenie v praxi

V kapitole Zrýchlenie sme určili teoretické zrýchlenie počítania uzlov, dosiahnuteľné zredukovaním veľkosti trojdiagonálnych sústav. V tejto časti si ukážeme reálne výsledky dosiahnuté v ukážkovej implementácii.

Ako bolo spomenuté, testovacia aplikácia bola implementovaná v jazyku C++. Program obsahuje testy pre sekvenčné aj paralelné počítanie derivácií pre oba predmetné algoritmy. Použitý prekladač bol Intel C++ Compiler v 64 bitovej verzii nastavený na generovanie agresívne optimalizovaného binárneho kódu (-O2), pričom paralelný kód bol implementovaný pomocou rozhrania OpenMP.

Testy boli vykonané na šiestich rôznych počítačových zostavách, všetky so systémom Windows 7 a 10. Testovacie stroje obsahujú rozličné multivláknové procesory od starého Penryn z roku 2007 až po najmodernejší Skylake z roku 2015 so vzájomne odlišnými architektúrami a hlavne spôsobmi vykonávania paralelizovaných procesov.

Stĺpec 1 obsahuje modely procesorov zoradených podľa mikroarchitektúry ako v tabuľke 2.1.3. Vedľa názvu modelu CPU sa nachádza údaj tvaru nC/mT , kde n je počet fyzických jadier procesora a m počet logických jadier. V prípade $m > n$ je procesor vybavený určitou formou SMT. Stĺpce 2

a 3 predstavujú časy behov sekvenčných implementácií plného a redukovaného algoritmu v prípade krivkových splajnov. Stĺpce 4 a 5 predstavujú časy behov sériovej verzie pre povrchové splajnov. Ostávajúce stĺpce analogicky znázorňujú časy a pomer paralelných verzií. Všetky uvedené časové údaje sú v milisekundách.

Procesor	Krivka		Sériovo		Paralelne	
	Plný	Reduk.	Plný	Reduk.	Plný	Reduk.
FX-6300 _{3C/6T}						
C2D E8200 _{2C/2T}						
Ci5 650 _{2C/4T}						
Ci3 2350M _{2C/4T}	66	53	153	131	82	81
Ci5 4440 _{4C/4T}						
Ci7 6700K _{4C/8T}	29	25	55	49	28	30

Sériová verzia redukovaného algoritmu je o približne 12% rýchlejšia ako pôvodný de Boorov postup čo viac-menej súhlasí s odhadnutým zlepšením na konci kapitoly Zrýchlenie v príklade 2.3.

Môžeme všimnúť malé škálovanie výkonu paralelizovaného algoritmu najmä v prípade procesorov obsahujúcich nejakú formu SMT kde je redukovaný algoritmus dokonca o niečo pomalší ako plný. Prvou zmysľanou príčinou môže byť fakt, že v plnom algoritme všetky parciálne derivácie podľa nejakej premennej vypočítame trojdiagonálnymi sústavami naraz v jednom cykle. V redukovanom ale cez trojdiagonálne sústavy vyrátame len polovicu derivácií, pričom zvyšok dopočítame práve restami. Toto rozdelenie výpočtu na dve subprocedúry môže hypoteticky implikovať vysokú réžiu plánovača vlákien (tzv. *thread scheduler*) OpenMP alebo operačného systému. V implementácií sa nám síce podarilo spojiť výpočet restov spolu s LU dekompozíciou do jedného kroku v prípade derivácií podľa premenných x a y . Teda resty môžeme počítateľ súčasne s deriváciami počítanými cez LU. Táto optimalizácia však momentálne nie je možná pri zmiešaných deriváciách. Síce máme rozpracované zefektívnenie výpočtu zmiešaných restov, ktoré sa nám ale ešte nepodarilo do práce dokončiť.

V prípade SMT procesorov sa dá pomalosť, aj keď to nemáme overené, vysvetliť ešte superskalárnosťou podľa podsekcie 2.1.2. Väčšina moderných mikroarchitektúr má jadrá vybavené práve dvomi matematickými jednotkami pre operácie s plávajúcou desatinnou čiarkou (označujú sa skratkou FPU). V tomto prípade SMT môže teoreticky každú jednotku prideliť inému vláknu. Pri redukovanom algoritme ale máme veľa operácií v tvare $a_0 a_1 \circ \dots \circ a_n$, kde $\circ \in \{+, \cdot\}$ kde optimalizujúci prekladač môže takýto výpočet rozložiť do viacerých strojových inštrukcií. V tomto prípade dôjde k využitiu oboch

FPU jadra v rámci jediného vlákna ale napriek tomu SMT na toto jadro priradí aj druhé vlákno ktoré počíta práve na FPU. Dôjde tak k zbytočnej réžii plánovača vlákien či už na úrovni OS alebo inštrukčného plánovača samotného procesora, čo spôsobí pokles výkonu.

Ak je táto hypotéza korektná tak by bolo potrebné upraviť resty, najmä pri zmiešaných deriváciách, aby sa ich výpočet dal vykonať paralelne v jednom kroku spolu s LU dekompozíciou

Kapitola 3

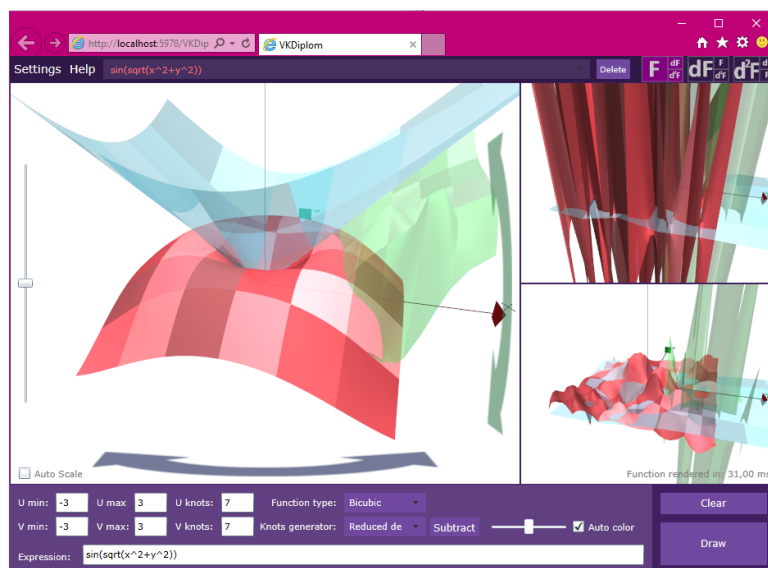
Implementácia a užívateľská príručka

Jedným z našich cieľom je vytvorenie aplikácií na vizuálne a výkonnostné porovnávanie splajnov, ktorých uzly sú počítané De Boorovim a našim redukovaným algoritmom. Pôvodne sme vizuálnu časť implementovali v Microsoft Silverlight, čo je bezplatný nástroj na tvorbu webových aplikácií spustiteľných priamo vo webovom prehliadači. Samotný vývoj aplikácii môže prebiehať v ľubovoľnom programovacom jazyku bežiacom pod virtuálnym strojom Common Language Runtime. V našom prípade sme siahli po jazyku C#.



Silverlight je kompatibilný s väčšinou moderných webových prehliadačov a operačných systémov vrátane Microsoft Windows, Apple OS X a vďaka technológii Moonlight – open-source implementácii Silverlight-u aj na väčšine Linuxových distribúcií. Framework je možné použiť aj na vývoj off-line aplikácií v operačných systémoch Windows Phone, Windows 8 a Windows RT.

Poznámka 3.1 Naša aplikácia používa technológie, ktoré nie sú s Moonlight kompatibilné. V súčasnej dobe teda podporujeme iba MS Windows a Apple OS X. Prípadná portácia na GNU/Linux by musela byť vo forme off-line desktopovej aplikácie.



Obr. 3.7: Ukážka z aplikácie.

Pretože CLR znemožňuje korektne porovnávať procesorový čas a pamäťové nároky implementovaných algoritmov, rozhodli sme sa výkonnostný tester oddeliť do samostatnej aplikácie implementovanej v natívnom jazyku, konkrétne v jazyku C++.

```

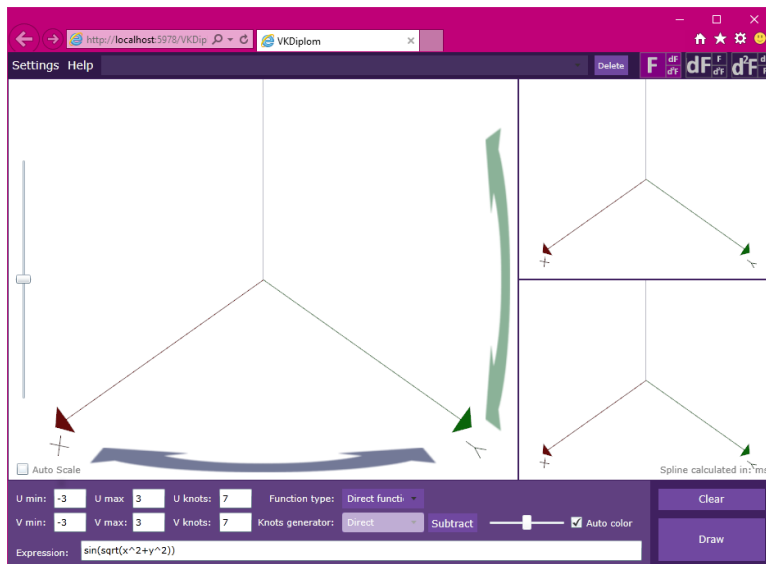
SplineKnots Benchmark 2
1: Multiplication vs division benchmark.
2: Spline curve benchmark.
3: Spline surface benchmark.
4: Spline surface benchmark (in parallel).
5: Compare Csaba T. vs. Vilo K. LU decomposition.
B: Disable/enable buffering in benchmarks.
Q: End program

```

Obr. 3.8: Výkonnostný tester.

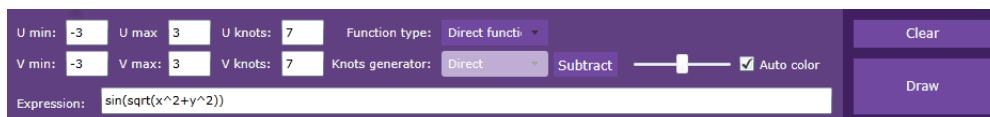
3.0.1 Vizualizácia

V tejto kapitole si ukážeme stručný návod na používanie a testovanie implementovaných aplikácií. Začnime aplikáciou na grafickú vizualizáciu.



Obr. 3.9: Aplikácia po spustení.

Okno aplikácie sa skladá z troch hlavných častí, ktoré si trochu netradične prejdeme zdola nahor. Na obrázku 3.10 vidíme panel, pomocou ktorého mô-

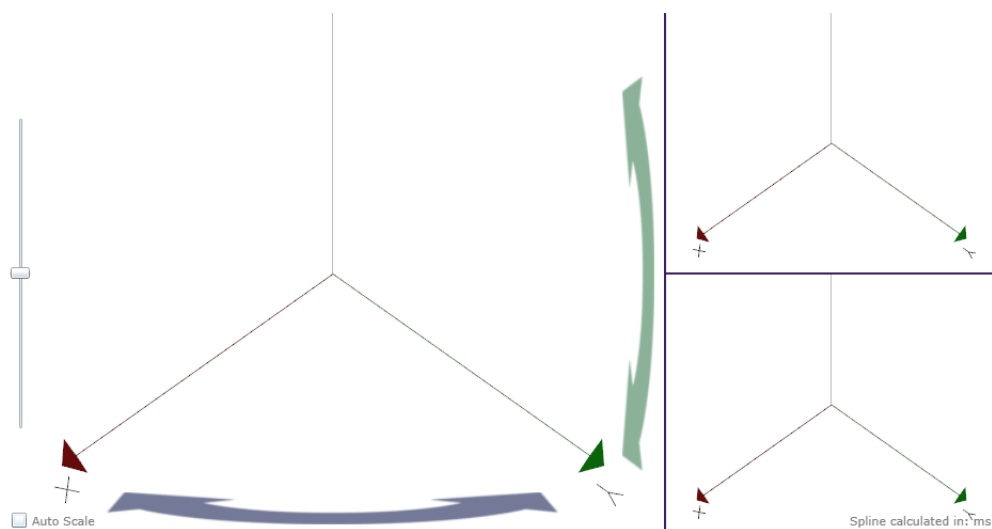


Obr. 3.10: Panel zadávania splajnu.

žeme na ľubovoľnom intervale a s ľubovoľným počtom uzlov (v rozumných medziach) interpolovať zapísanú funkciu dvoch premenných x a y . Popíšme si jednotlivé ovládacie prvky. Textové polia označené $U min$, $U max$ a $U knots$ ($V min$, $V max$ a $V knots$) značia interval a počty uzlov na osi x (osi y). Do spodného textového políčka označené *Expression* môžeme napísať takmer ľubovoľnú matematickú funkciu premenných x a y ktorú si želáme vykresliť. Vysúvateľný zoznam označený *Function type* umožňuje zvoliť typ interpolácie. Na výber máme ponúknuté tri možnosti akým spôsobom si želáme interpretovať matematický výraz v políčku *Expression*.

- *Direct function* vypočíta funkčné hodnoty potrebné pre grafickú vizualizáciu priamo z funkcie.
- *Bicubic* vypočíta z funkcie iba funkčné hodnoty v uzloch. Zvyšné potrebné body sú získané interpolačným bikubickým splajnom.
- *Biquartic* funguje analogicky ako predchádzajúca položka. Použije sa ale bikvartický splajn.

Vysúvateľný zoznam označený *Knots generator* nám, v prípade výberu vykreslenia funkcie splajnom, umožňuje zvoliť plný (*De Boor*) alebo redukovaný (*Reduced de Boor*) algoritmus. Tlačítko *Subtract* umožní vykresliť rozdiel medzi aktuálne zadanými hodnotami a nejakou už vykreslenou funkciou. Pomocou posuvníka označený *Auto color* môžeme namiesto automaticky vybranej farby vykresliť novú plochu vlastnou farbou. Nakoniec v pravej časti okna sa nachádzajú tlačidlá *Clear* a *Draw*, pričom prvé odstráni všetky vykreslené funkcie a druhé nám vypočíta a zobrazí funkciu podľa aktuálne zadaných hodnôt.



Obr. 3.11: Hlavné zobrazovacie okná.

Na obrázku 3.11 vidíme dominantnú časť aplikácie a síce zobrazovanie funkciových plôch. Obrazovka je rozdelená na tri časti, kde v najväčšia časť implicitne zobrazuje interpretovanú funkciu a zvyšné dve zobrazujú prvú, respektíve druhú zmiešanú deriváciu funkcie. V ľavej časti okna môžeme nájsť posuvník na škálovanie osi z . Na spodnej a pravej strane hlavnej časti okna sú tlačidlá reprezentované modrou horizontálnou, respektíve zelenou vertikálnou šípkou. Pri ich držaní myšou je možné rotovať obrazom v danej osi.

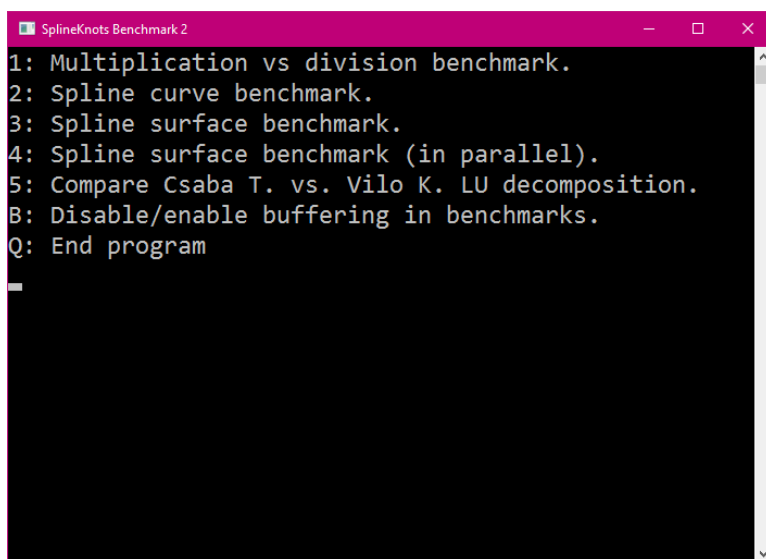


Obr. 3.12: Lišta nástrojov.

Položka *Settings* umožňuje prispôbiť grafické vykresľovanie splajnov. Tieto nastavenia je vhodné upraviť najmä pri starších počítačoch na dosiahnutie optimálnej plynulosti vykresľovania. V strednej časti panela máme k dispozícii vysúvateľný zoznam vykresľovaných plôch, ktorým ich môžeme zvýrazňovať. V pravej časti nájdeme tri tlačidlá, ktoré vycentrujú zobrazenie splajnu, prípadne jeho prvej alebo druhej zmiešanej derivácie.

3.0.2 Výkonnostný tester

Teraz si popíšme druhú implementovanú aplikáciu zameranú nie na vykresľovanie, ale na ukážku či skutočne je redukovaný algoritmus rýchlejší. Na



Obr. 3.13: Výkonnostný tester.

obrázku 3.13 môžeme vidieť úvodné okno aplikácie, ktoré obsahuje sedem položiek. Ich význam si teraz vyjasníme.

1. *Multiplication vs division benchmark* spustí niekoľko variant testovania pomerov rýchlosti operácií sčítania, násobenia a delenia.
2. *Spline curve benchmark* pomerá časy výpočtov derivácií pre krivkový splajn interpolujúci preddefinovanú funkciu plným a redukovaným al-

```
SplineKnots Benchmark 2
1: Multiplication vs division benchmark.
2: Spline curve benchmark.
3: Spline surface benchmark.
4: Spline surface benchmark (in parallel).
5: Compare Csaba T. vs. Vilo K. LU decomposition.
B: Disable/enable buffering in benchmarks.
Q: End program
3

-----
Spline surface benchmark

Enter number of iterations:
10
Enter number of knots:
999
```

Obr. 3.14: Zadanie údajov pre testovanie.

```
SplineKnots Benchmark 2
Q: End program
3

-----
Spline surface benchmark

Enter number of iterations:
10
Enter number of knots:
999
Ignore -1457.33
Full : 68
Reduced : 53
Difference : 1.28302
=====
Press any key to continue . . .
```

Obr. 3.15: Výsledky testu.

goritmom. Pred spustením aplikácie vyžaduje postupne zadanie dvoj údajov. Prvý udáva počet opakovaní testu pre každý algoritmus. Výsledný pomer vznikne vydelením aritmetických priemerov nameraných časov pre každý algoritmus. Čím je zadané číslo väčšie, tým je meranie menej náchylné na odchýlky. Odporúča sa zadať dvojcifernú hodnotu

a potvrdiť klávesou **Enter**. Druhé zadané číslo znamená počet uzlov splajnu. Odporúča sa zadať rádovo 10^5 až 10^6 uzlov. Menší počet dokáže väčšina moderných počítačov vypočítať príliš rýchlo aby ten čas dokázala aplikácia rozumne zmerať.

3. *Spline surface benchmark* analogicky ako predchádzajúci test testuje pomer časov výpočtu plného a redukovaného algoritmu pre plochy. Druhý zadany údaj tentoraz znamená počet uzlov pre jeden rozmer. Celkový počet uzlov je teda štvorec zadanej hodnoty. Pre väčšinu počítačov sa odporúča zadať rádovo 10^2 až 10^3 uzlov.
4. *Spline surface benchmark (in parallel)* paralelná verzia testu pre plochy využívajúca na výpočet jednej matice uzlov všetky dostupné logické procesory.
- B. *Disable/enable buffering* prepne medzi naivnou a optimalizovanou variantou algoritmu pre počítanie LU dekompozície v testoch 2 a 3. Test 4 toto nastavenie neovplyvňuje z dôvodu značných pamäťových nárokov naivnej verzie LU dekompozície.
- Q. *End program* ako názov napovedá, pri stlačení tlačidla **Q** dôjde k vypnutiu aplikácie.

Záver

Podarilo sa nám teda pomerne výrazne urýchliť výpočet derivácií splajnov v ich uzloch a tiež zmenšiť pamäťové nároky algoritmu. Samozrejme stále existujú oblasti v ktorých by sa dal zlepšiť. Prioritou teraz ostáva jednak zovšeobecniť redukovaný algoritmus aj pre neuniformné splajny a upraviť zmiešané zbytkové derivácie aby sme mohli zefektívniť paralelizáciu. Tá má momentálne prínos najmä v polovičných pamäťových nárokoch samotného algoritmu.

Zoznam použitej literatúry

- [1] David Salomon, Curves and Surfaces for Computer Graphics, Springer, 2006
- [2] I. Szabó, L. Miño, C. Török, Biquartic polynomials in bicubicspline construction, PF UPJŠ, 2014
- [3] L. Miño, Parametrické modelovanie dát komplexnej štruktúry, PF UPJŠ, 2014
- [4] C. de Boor, Bicubic spline interpolation, Journal of Mathematics and Physics, 41(3),1962, 212-218.
- [5] J. Albahari, B.Albahari, C# 5.0 in a Nutshell, O'Reilly, 2012
- [6] E. Süli, D. Mayers, An Introduction to Numerical Analysis, Cambridge University Press, 2003, ISBN 0 521 00794 1
- [7] <http://www.anandtech.com/show/9483/intel-skylake-review-6700k-6600k-ddr4-ddr3-ipc-6th-generation/9>
- [8] <http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf>
- [9] http://www.agner.org/optimize/instruction_tables.pdf
- [10] <http://www.lighterra.com/papers/modernmicroprocessors/>
- [11] <http://techreport.com/review/23750/amd-fx-8350-processor-reviewed/3>