TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta mechatroniky, informatiky a mezioborových studií

BAKALÁŘSKÁ PRÁCE

Liberec 2011 **Pavel Exner**

TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta mechatroniky, informatiky a mezioborových studií

Studijní program: B2612 - Elektrotechnika a informatika

Studijní obor: 2612R011 - Elektronické informační a řídicí systémy

Interpolační třídy pro funkce reálné proměnné s aplikací v hydrologii

Interpolation classes for functions of real variable with application in hydrology

Bakalářská práce

Autor: **Pavel Exner**

Vedoucí práce: Mgr. Jan Březina, Ph.D.

V Liberci 19. května 2011

Zadání

Prohlášení

Byl jsem seznámen s tím, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 o právu autorském, zejména § 60 (školní dílo).

Beru na vědomí, že TUL má právo na uzavření licenční smlouvy o užití mé bakalářské práce a prohlašuji, že **s o u h l a s í m** s případným užitím mé bakalářské práce (prodej, zapůjčení apod.).

Jsem si vědom toho, že užít své bakalářské práce či poskytnout licenci k jejímu využití mohu jen se souhlasem TUL, která má právo ode mne požadovat přiměřený příspěvek na úhradu nákladů, vynaložených univerzitou na vytvoření díla (až do jejich skutečné výše).

Bakalářskou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím bakalářské práce a konzultantem.

Datum

Podpis

Abstrakt

V oboru hydrologie se při modelování nenasyceného proudění vody v porézním prostředí opakovaně počítají složité funkce, kde je proměnnou veličinou tlak. Tyto funkce lze aproximovat s určitou chybou a získat tak jejich jednodušší popis. Výhodou aproximací je snazší vyhodnocování funkční hodnoty v bodě, což by mělo výrazně urychlit běh programu.

Tato bakalářská práce se zabývá popisem funkcí jedné proměnné, výpočtem jejich funkčních hodnot, derivací a především jejich interpolací. Jsou zde popsány vybrané interpolační metody a zároveň navrženy možné způsoby algoritmizace. Základní myšlenkou je tvorba interpolace pomocí spline funkce. V této práci se rozlišuje spline Lagrangeova typu a spline Hermitova typu. Důležitou součástí je možnost aplikace adaptivního přístupu na tyto metody, tzn. umožnění adaptivní volby uzlů při minimalizaci chyby interpolace.

Získané teoretické poznatky se pak implementují v jazyce C++. Mezi hlavní charakteristiky programového řešení patří použití šablon pro funktory k popisu funkce jako takové. Výpočet derivací je prováděn metodou automatické diferenciace. Důraz je kladen na obecnost návrhu a na rychlost především těch částí programu, které obsluhují vytvořenou interpolaci.

Navržený systém má sloužit pro tvorbu interpolací hydrologických funkcí s velmi rychlým vyhodnocováním.

Klíčová slova:

interpolace, spline, šablony, C++

Abstract

Very complicated functions in which the only variable is pressure are repeatedly evaluated during the modeling of the unsaturated flowing of water in the porous material. These functions can be approximated with a specified tolerance. The main advantage of using the approximation is making the evaluation simpler and faster which should speed up the main modeling algorithm.

This bachelor thesis is aimed at the description of the functions depending on one variable, computing their values and derivates and especially their interpolations. Selected methods of interpolation are described and the algorithms of these methods are suggested. The main idea is using the interpolations with spline functions. There are two types of spline functions in the thesis – Lagrange and Hermit spline. The possibility of making these two methods adaptive is very important. Adaption means that one can change the nodes to minimize the error of the interpolation.

The theoretical knowledge is then implemented in the C++ language. One of the main code features is the description of functions using the templates on functors. The automatic differentiation algorithm is used to compute the derivates of functions. Emphasis is placed on the complexity of the code and the speed of evaluating the interpolations.

The system should be used for creation interpolations of hydrological functions with very fast evaluations.

Keywords:

interpolation, spline, templates, C++

Obsah

Zadání 2

Prohlášení 3

Abstrakt 4

Obsah 6

Seznam obrázků 8

Seznam kódů 8

Seznam tabulek 8

1 Úvod 9

1.1 Motivace 9

1.2 Struktura práce 10

2 Matematický popis interpolace 12

2.1 Interpolace Lagrangeova typu 12

2.2 Interpolace Hermitova typu 17

3 Třídy pro popis funkce 19

3.1 Funktor 19

3.2 Struktura funktorových tříd 20

3.3 Derivace a knihovna FADBAD++ 22

3.3.1 Získání derivace zpětnou metodou diferenciace . 23

3.3.2 Získání derivace z Taylorova rozvoje 24

4 Třídy pro tvorbu interpolace 26

4.1 Struktura interpolačních tříd 26

4.1.1 Třídy popisující výstupní objekt – interpolant . 27

4.1.2 Třída `Polynomial` – polynom 28

4.1.3 Třída `BCondition` – okrajové podmínky 30

4.2 Třída `Lagrange` - Lagrangeova interpolace 30

4.3 Knihovna CLAPACK 32

4.4 Vyhodnocování chyby interpolace 33

4.5 Třída `Adaptive` – adaptivní interpolace 35

5 Testování systému 38

5.1 Odlaďování 38

5.2 Test funkcí FK a FQ 39

6 Závěr 44

Použitá literatura 46

A Zdrojové kódy vybraných tříd 48

A.1 `FunctorDiffBase` 48

A.2 `InterpolantBase` 48

A.3 `InterpolantEq` a `InterpolantAdapt` 49

A.4 `Polynomial` 49

A.5 `IInterpolation` 50

A.6 `Lagrange` 51

A.7 `Adaptive` 52

A.8 `BCondition` 52

Seznam obrázků

1 Příklad použití knihovny v programu 10
2 Příklad sestavení matice 16
3 Struktura systému funktorů 20
4 Struktura interpolačních tříd 26
5 Struktura výpočtu norem 34
6 Algoritmus adaptivní interpolace 36

Seznam kódů

1 Ukázka jednoduchého funktoru pro funkci x^3 19
2 Třída `FunctorValue` 21
3 Možnosti tvorby funktoru 22
4 Metoda vracející 1. derivaci 23
5 Metoda vracející n-tou derivaci 25
6 Výpočet chyby v metodě `ComputeError` 36
7 Vypis koeficientů polynomů 38
8 Parametry testu 39

Seznam tabulek

1 Tvorba interpolace – ekvidistantní dělení 40
2 Tvorba interpolace – adaptivní dělení 41
3 Čas [s] výpočtu `n` hodnot – ekvidistantní dělení 42
4 Čas [s] výpočtu `n` hodnot – adaptivní dělení 42

1 Úvod

Bakalářská práce si klade za cíl vytvořit nástroj pro tvorbu interpolací funkcí jedné reálné proměnné. Vybrané matematické metody k určení interpolace převádí do programové implementace v jazyce C++. Používá některé zajímavé programátorské přístupy a vytváří ucelený systém tříd a metod.

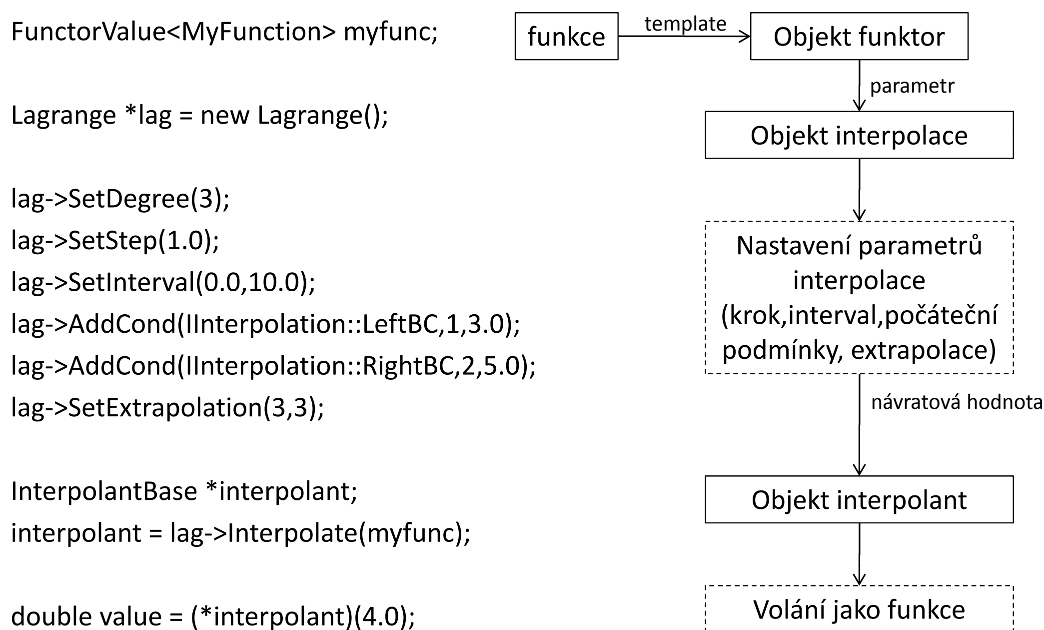
1.1 Motivace

Složitost funkcí v oboru hydrologie vede k velké časové náročnosti výpočtů. Zejména při modelování nenasyceného proudění vody v porézním prostředí se vyhodnocují funkce ve mnoha bodech a právě rychlost získání funkčních hodnot je zde omezující. Počítané hydrologické funkce závisí na jedné proměnné – na tlaku, což je jedním z předpokladů této práce. Dalším předpokladem je znalost předpisů těchto funkcí a možnost tyto funkce aproximovat. Dvě vybrané funkce – hydraulická vodivost a vodní saturace – byly nejprve přepsány z jazyka Fortran do C++, aby poté na nich mohla být vyzkoušena interpolace.

Výsledná knihovna by měla najít uplatnění všude, kde je potřeba vyhodnocovat velmi složité funkce s důrazem na rychlost výpočtu a kde je možné danou funkci nahradit jejím přiblížením. Neopomenutelným cílem je i dostatečná obecnost a možnost jejího dalšího rozšiřování, či úprav, a to především matematického aparátu, tj. interpolačních nebo obecně aproximačních metod bez nutnosti zásahu do navrženého systému.

1.2 Struktura práce

Abychom získali představu, co by nám knihovna měla ve výsledku umožnit, uvedeme si již na tomto místě stručný příklad tvorby interpolace pomocí knihovních tříd (viz obr. 1). Konkrétní zápis příkazů zde není důležitý, podstatnější je nyní schéma po pravé straně obr. 1, které naznačuje, o co v jakém kroku jde. Na konci tohoto textu již budeme vědět, co se skrývá pod jednotlivými bloky schématu.



obr. 1: Příklad použití knihovny v programu

V první části si popíšeme jaké matematické postupy budeme implementovat do našeho systému. To bude zahrnovat použité interpolační metody splinu Lagrangeova a Hermitova typu. Dílčí matematické metody např. pro vyhodnocování chyby řešení (výpočet norem), evaluaci polynomu v bodě a jiné uvedeme na příslušném místě. V dalších kapitolách se potom budeme zabývat implementací jednotlivých metod a popisem tříd.

Postup (implementační část) můžeme rozdělit do několika etap, ve kterých jsme řešili dílčí problémy. Základem bylo navrhnout objekt, který bude popisovat matematickou funkci, bude vhodně vyhodnocovat její hodnotu a derivaci v nějakém bodě (využití knihovny FADBAD++). Takový objekt potom bude vstupem do systému zobecněných funktorů, který si přiblížíme v kapitole 3.

V další etapě se zaměříme na návrh tříd, které budou vyhodnocovat vlastní interpolaci. Vstupní i výstupní objekt bude potomkem obecného funktoru. Výstupní objekt nazveme interpolant, popíšeme jeho metody a to především vyhodnocování hodnoty, derivace a integrálu, jeho extrapolaci a další. Popíšeme i pomocné třídy definující okrajové podmínky, polynomy a také rozhraní s knihovnou CLAPACK, kterou využíváme při interpolaci k výpočtu vzniklé pásové matice.

Po návrhu interpolačních tříd, které braly v úvahu výpočet pouze pro síť ekvidistantních uzlů, se přesuneme k úkolu obtížnějšímu, a to k implementaci adaptivity. V kapitolách věnovaných této etapě rozebereme, jakým způsobem k adaptivní interpolaci přistupujeme, jak vyhodnocujeme chybu interpolace a dosažení zadané tolerance a jak následně zjemňujeme síť uzlů.

V poslední kapitole chceme ukázat příklad použití na vybraných funkcích z oboru hydrologie a zhodnotit výsledky, které jsme s interpolačním systémem dosáhli. Porovnání provedeme z hlediska časové náročnosti evaluace původní funkce oproti interpolantu s uvedením dosažené přesnosti – chyby, které jsme se interpolací dopustili.

Všechny třídy a metody jsou okomentovány stylem, umožňujícím automatické generování dokumentace systémem Doxygen.

2 Matematický popis interpolace

V této kapitole si přiblížíme problematiku interpolace z matematického hlediska a definujeme základní pojmy. Nejprve zavedeme obecný pojem aproximace funkce, kterou rozumíme hledání funkce jiné, která je v nějakém smyslu jednodušší a kterou je možné funkci původní za určitých podmínek nahradit. Aproximovanou funkci můžeme znát přesně a hledáme pouze jednodušší způsob, kterým bychom ji mohli s určitou tolerancí vyjádřit, nebo známe jen hodnoty v některých bodech a prokládáme jimi jinou funkci takovým způsobem, abychom minimalizovali odchylku od funkce původní. Aproximace a interpolaci definuje např. E. Vitáska (viz publikace [2], kapitola 32, str. 620):

Definice. Zvolíme předem konečnou množinu bodů v definičním oboru aproximované funkce f a žádáme, aby hodnoty aproximace a eventuálně i hodnoty některých jejích derivací v této množině souhlasily s příslušnými hodnotami funkce f. V tomto případě mluvíme o interpolační aproximaci nebo stručně o *interpolaci*.

2.1 Interpolace Lagrangeova typu

Uvedeme definici interpolace Lagrangeova typu klasickou spline funkcí (viz [3], kapitola 5.3, str.85-89)

Definice. Nechť jsou dány body $a = x_0 < x_1 < \ldots < x_n = b$ (nazýváme je uzly). *Spline* funkce řádu r s uzly $x_0, \ldots, x_n$ nazveme funkci S(x), která splňuje následující podmínky:

(1) V každém intervalu $\langle x_i, x_{i+1} \rangle$ je polynomem nejvýše řádu r.

(2) Funkce S má spojité derivace až do řádu $r - 1$.

Spline funkce 1. řádu je po částech lineární, jejímž grafem je lomená čára. V uzlech splňuje podmínku (1), tj. má spojitou 0. derivaci, resp. spline funkce je spojitá. Konstantní aproximace této definici již nevyhovuje úplně (derivace -1. řádu pro nás nebude mít smysl), ale uplatňujeme podmínku (2). Nejčastěji používané jsou *kubické spliny* (spline funkce 3.řádu). V této práci jsme místo několika tříd navrhli jedinou obecnou třídu Lagrange, která implementuje Lagrangeovu metodu pro libovolný řád. Tímto krokem jsme obětovali některé výhody (především jednoduchost konstrukce interpolantu pro konstantní, lineární a kubickou interpolaci), ale získali nové možnosti, zcela obecný přístup a uspořili řádky kódu.

Uveďme nyní z čeho vycházíme při implementaci. Abychom teorii propojili přímo s kódem, budeme do následujícího textu vpisovat i názvy metod třídy `Lagrange`, ve kterých je daný problém zpracován. Mějme síť $n+1$ uzlů

$$x_0,\, x_1,\, \ldots,\, x_i,\, \ldots,\, x_n, \tag{1}$$

ve kterých známe (můžeme vypočítat - metoda `SetFunctionValues`) $n+1$ funkčních hodnot

$$f_0,\, f_1,\, \ldots,\, f_i,\, \ldots,\, f_n. \tag{2}$$

Mezi uzly hledáme n polynomů řádu r

$$P_1,\, P_2,\, \ldots,\, P_i,\, \ldots,\, P_n, \tag{3}$$

které mají derivace až do řádu $r-1$

$$
\begin{aligned}
P_i(x) &= \alpha_i^0 + \alpha_i^1 x + \alpha_i^2 x^2 + \ldots + \alpha_i^r x^r \\
P_i'(x) &= \alpha_i^1 + 2\alpha_i^2 x + \ldots + r\alpha_i^r x^{r-1} \\
P_i''(x) &= 2\alpha_i^2 + 6\alpha_i^3 x + \ldots + r(r-1)\alpha_i^r x^{r-2} \\
&\vdots \\
P_i^{(k)}(x) &= \frac{!k}{!0}\alpha_i^k + \frac{!k+1}{!1}\alpha_i^{k+1} x + \ldots + \frac{!r}{!(r-k)}\alpha_i^r x^{r-k} \\
&\vdots \\
P_i^{(r-1)}(x) &= (!r-1)\alpha_i^{r-1} + \frac{!r}{!1}\alpha_i^r x. \qquad (4)
\end{aligned}
$$

Koeficienty polynomů α_i^j jsou neznámé, kterých je $n(r+1)$. Z podmínky spojitosti (nulté derivace) neboli rovnosti funkce a interpolantu v uzlových bodech vyplývá $2n$ následujících rovnic

$$
\begin{aligned}
P_1(x_0) &= f_0 \\
P_2(x_1) = P_1(x_1) &= f_1 \\
&\vdots \\
P_i(x_i) = P_{i-1}(x_i) &= f_i \\
&\vdots \\
P_n(x_{n-1}) = P_{n-1}(x_{n-1}) &= f_{n-1} \\
P_n(x_n) &= f_n. \qquad (5)
\end{aligned}
$$

Z podmínky spojitosti $r-1$ derivací splinu v $n-1$ uzlových bodech (bez krajních) vyplývá $(n-1)(r-1)$ rovnic

$$
\begin{aligned}
P_i'(x_i) &= P_{i-1}'(x_i) \\
&\vdots \\
P_i^{(r-1)}(x_i) &= P_{i-1}^{(r-1)}(x_i) \qquad \text{pro } i = 2, \ldots, n. \qquad (6)
\end{aligned}
$$

Zatím uvedené rovnice plníme do matice v metodě `PutEquations`. Nutno ještě zmínit, že v rámci spline funkce omezujeme definiční obor polynomu P_i na interval $\langle x[i-1], x[i] \rangle$ a ten zároveň o $-x[i-1]$ posouváme (tzn. že jeho interval musíme zapsat $\langle 0, x[i] - x[i-1] \rangle$). To má samozřejmě vliv na hodnoty koeficientů a takový tvar polynomu nazýváme normalizový.

Nyní nám zbývá ještě $r-1$ rovnic, abychom mohli určit všechny koeficienty. Tyto rovnice nám dají počáteční podmínky. Na levé straně straně i na pravé straně intervalu máme $r-1$ nedefinovaných derivací. Můžeme si zvolit libovolný počet podmínek na libovolné straně, ale dohromady jich musíme mít definováno přesně $r-1$. Na levé máme tyto možnosti:

$$\begin{aligned} P_1'(x_0) &= u_1 \\ &\vdots \\ P_1^{(r-1)}(x_0) &= u_{r-1}. \end{aligned} \tag{7}$$

Podobně na pravé straně můžeme definovat podmínky:

$$\begin{aligned} P_n'(x_n) &= v_1 \\ &\vdots \\ P_n^{(r-1)}(x_n) &= v_{r-1}. \end{aligned} \tag{8}$$

Pokud je definován menší počet podmínek, zavolá se metoda `AutoAdd`, která určí počet potřebných podmínek, rozdělí je rovnoměrně na levou a pravou stranu a od nejvyšších derivací je vyplní nulami. Na počtu okrajových podmínek závisí rozměr matice, takže metoda `AutoAdd` se volá ještě před alokováním matice, ale metoda `PutBC`, která plní podmínky do matice, se volá až při plnění matice.

Na schematickém zázornění matice (obr. 2) je případ pro interpolaci splinem 3. řádu na intervalu $\langle x[0], x[3] \rangle$. Čárkovanou čarou jsou odděleny řádky obsahující okrajové podmínky (z nich musíme vybrat jen 2, jinak by soustava byla přeurčená). Pravou stranu tvoří funkční hodnoty a hodnoty okrajových podmínek. Za zmínění stojí také fakt, že submatice pro jednotlivé polynomy (v příkladu o rozměrech 4x4), se podél diagonály opakují, pouze krajní submatice se liší dle okrajových podmínek.

x_0				x_1				x_2				x_3
	P_1				P_2				P_3			
α_0^1	α_1^1	α_2^1	α_3^1	α_0^2	α_1^2	α_2^2	α_3^2	α_0^3	α_1^3	α_2^3	α_3^3	
0	1											u_1
0	0	2										u_2
1	0	0	0									f_0
1	1	1	1	0								f_1
	1	2	3	0	-1							0
		2	6	0	0	-2						0
			0	1	0	0	0					f_1
				1	1	1	1	0				f_2
					1	2	3	0	-1			0
						2	6	0	0	-2		0
							0	1	0	0	0	f_2
								1	1	1	1	f_3
									1	0	0	v_1
										2	0	v_2

obr. 2: Příklad sestavení matice

2.2 Interpolace Hermitova typu

Ačkoli v systému zatím nebyla implementována, popíšeme zde i metodu interpolace Hermitovým splinem. Je jednou z těch, která pro sestavení interpolantu předpokládá znalost derivace funkce, což vedlo k zavedení možnosti určovat derivaci do tohoto systému pomocí knihovny FADBAD++. Uveďme si definici E. Vitáska z publikace [2], kapitola 32.9, definice (3):

Definice. Hermitovým splinem stupně $2k-1$ ($k \geq 1$ celé) pro uzly

$$a = x_0 < x_1 < \ldots < x_n = b$$

rozumíme funkci, která je v každém intervalu

$$\langle x_i, x_{i+1} \rangle, i = 0, \ldots, n-1$$

rovna polynomu stupně nejvýše $2k-1$ a která má v celém intervalu $\langle a, b \rangle$ spojité derivace až do řádu $k-1$ včetně.

Uvažujme pro zjednodušení kubický hermitovský spline, na intervalu s ekvidistantním rozdělením uzlových bodů. Můžeme pak odvodit konkrétní vztahy pro algoritmizaci. Budeme tvořit polynomy 3. řádu ($k=2$) a v uzlových bodech budeme požadovat rovnost 1. derivace polynomů ($k-1=1$). Interval budeme dělit konstantním krokem h a podintervaly budeme uvažovat normalizované (viz minulá kapitola)

$$\langle x_i, x_i + h \rangle \rightarrow \langle 0, h \rangle .$$

Označíme funkční hodnotu f_i a derivaci g_i funkce v i-tém uzlu. Uvažujeme polynom (obdobně jako v (4) pro řád $r=3$)

$$P_i(x) = \alpha_i^0 + \alpha_i^1 x + \alpha_i^2 x^2 + \alpha_i^3 x^3. \tag{9}$$

Potom

$$P_i(0) = f_i, \qquad P_i(h) = f_{i+1}, \qquad P_i'(0) = g_i, \qquad P_i'(h) = g_{i+1}. \quad (10)$$

Dosazením (10) do (9) dostaneme

$$\begin{aligned} &\alpha_i^0 = f_i, \qquad \alpha_i^1 = f_{i+1}, \\ &\alpha_i^3 h^3 + \alpha_i^2 h^2 + \alpha_i^1 h + \alpha_i^0 = f_{i+1}, \\ &3\alpha_i^3 h^2 + 2\alpha_i^2 h + \alpha_i^1 = g_{i+1}. \end{aligned} \quad (11)$$

Ze soustavy (11) vypočítáme vztahy pro všechny koeficienty (12), které poté použijeme do algoritmu.

$$\begin{aligned} \alpha_i^3 &= 2\frac{f_i - f_{i+1}}{h^3} + \frac{g_i + g_{i+1}}{h^2}, \\ \alpha_i^2 &= 3\frac{f_{i+1} - f_i}{h^2} + \frac{2g_i + g_{i+1}}{h}, \\ \alpha_i^1 &= f_{i+1}, \\ \alpha_i^0 &= f_i. \end{aligned} \quad (12)$$

Tyto zjednodušené vztahy lze odvodit i pro proměnný krok h, což by umožnilo použití adaptivní metody se zjemňováním rozložení uzlových bodů. Složitější by bylo úplné zobecnění pro libovolný řád (tedy lichý, protože $r = 2k - 1$). Potom bychom dosáhli úplné univerzálnosti, je však otázkou, zda by vůbec spliny vyšších řádů měly smysl pro reálné použití.

3 Třídy pro popis funkce

3.1 Funktor

Nejprve je potřeba uvést, jak budeme v programu přistupovat k matematické funkci. V objektovém programování je velice vhodným prostředkem tzv. funktor. To může být jakýkoli objekt (jakkoli rozsáhlý, s jakýmikoli metodami a členy) ale s přetíženým operátorem (), který vrací hodnotu. Jednoduchá funkce x^3 pak může vypadat obdobně, jak je uvedeno v kódu 1.

Kód 1: Ukázka jednoduchého funktoru pro funkci x^3

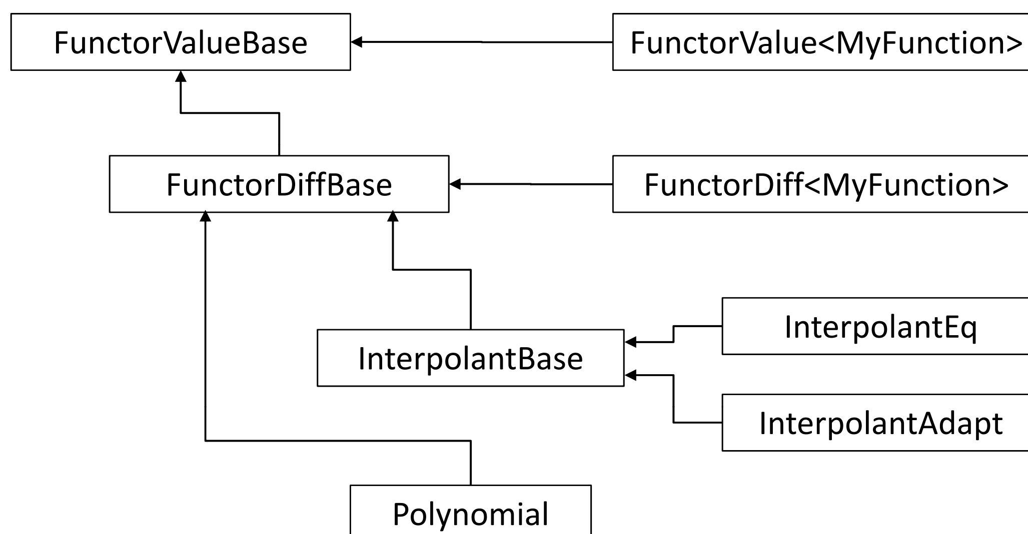
```
1  class MyFunction
2  {
3    public:
4      template <class T>
5      T operator()(const T& x)
6      {
7        T z = x*x*x;
8        return z;
9      }
10 };
```

Pro výpočet derivací používáme knihovnu FADBAD++, která definuje své vlastní numerické typy (numerický datový typ je takový typ, který má definované matematické operace např. +, -, * atd.). Při výpočtu derivace se právě tyto typy předvávají jako parametry (viz kap. 3.3). Funktor proto musíme ještě upravit tím způsobem, že na metodu `operator()` použijeme šablonu (template, viz manuál [8]). Šablona označuje všeobecný typ, jehož funkcionalita může být adaptována na více než jeden typ (třídu), aniž bychom opakovali celý kód pro každý typ zvlášť.

Příkladem může být právě i třída `MyFunction` v kódu 1. Za třídu `class T` můžeme dosadit jakýkoli numerický typ např. `double`, `int`, `float`, nebo právě z knihovny FADBAD++ typ `B<double>`. Takto definovaný objekt bude vstupem do našeho systému.

3.2 Struktura funktorových tříd

Abychom mohli použít jakékoli obecné templatované funktory, definujeme nad nimi rozhraní, které bude tvořit systém funktorů. Uvedeme na tomto místě strukturu funktorových tříd (viz obr. 3, plné šipky označují dědičnost).



obr. 3: Struktura systému funktorů

Nejobecnějším rozhraním je abstraktní třída `FuntorValueBase`, která definuje pouze to, že potomek této třídy musí mít přetížený operátor `operator()`. To pro nás znamená, že za tímto rozhraním může být schován jakýkoli objekt, který umí vrátit nějakou hodnotu `double` přes tento operátor.

Kód 2 ukazuje, jak je implementována třída `FunctorValue`, která je veřejným potomkem `FuntorValueBase`. Není již abstraktní – slovo `virtual` pouze říká, že je zde implementována virtuální metoda z předka. Z prvního řádku vidíme, že jde opět o šablonu třídy, `class Func` může představovat jakýkoli funktor (například výše zmíněnou funkci x^3 – kód 1).

Kód 2: Třída `FunctorValue`

```
1  template<class Func>
2  class FunctorValue : public FunctorValueBase
3  {
4    private:
5      Func fce;
6
7    public:
8      FunctorValue(void) {}   //constructor
9      ~FunctorValue(void) {}  //destructor
10
11     //overloaded operator() returns value of the function
12     virtual double operator() ( const double &x )
13     {
14       return fce(x);
15     }
16 };
```

Obdobně jsou navrženy třídy `FunctorDiffBase` (viz příloha A.1) a `FunctorDiff`, které vychází ze tříd založených na vracení hodnoty. Je v nich ale navíc implementován výpočet derivace, který rozebereme podrobněji v kapitole 3.3. Na tomto místě pouze zmíníme, že se derivace vrací strukturou `der`, která obsahuje jak hodnotu derivace, tak i funkční hodnotu.

Ostatní třídy zakreslené do schématu na obr. 3 ukazují, jak jsou zařazeny do struktury funktorů. Třídy `Polynomial`, `InterpolantEq`, `InterpolandtAdapt` a `InterpolantBase` popíšeme blíže v kapitole 4.1.

Tyto třídy nám umožňují v kódu deklarovat nebo konstruovat objekt například jedním ze zápisů v kódu 3. Konstrukce se může zdát složitá, ale umožňuje používání obecně definovaných funktorů a odstraňuje také komplikaci virtuální metody, která nemůže být zároveň šablonou.

Kód 3: Možnosti tvorby funktoru

```
1 FunctorValue<MyFunction> f1;
2 FunctorValue<MyFunction> *f2
3                   = new FunctorValue<MyFunction>();
4 //using abstract class as interface
5 FunctorValueBase *f3 = new FunctorValue<MyFunction>;
6 FunctorDiffBase *f4 = new InterpolantAdapt();
```

3.3 Derivace a knihovna FADBAD++

Pro získání derivace použijeme volně dostupnou knihovnu s názvem FADBAD++ [6] (Flexible Automatic Differentiation). Tato knihovna implementuje techniku automatické diferenciace pomocí šablon a přetížených operátorů. Nabízí metody dopředné a zpětné diferenciace a Taylorův rozvoj. Jednotlivé metody jsme postupně vyzkoušeli a protože potřebujeme zatím pouze první derivaci, rozhodli jsme se použít zpětnou metodu. V případě, že bychom potřebovali derivace vyšších řádů, lze použít Taylorův rozvoj nebo kombinaci více metod alespoň pro druhou derivaci.

Knihovna používá templatovanou automatickou diferenci na funkce implementované v C++. Abychom mohli použít automatickou diferenci, přepíšeme aritmetické typy v kódu (např. `double`) za šablonové typy (tzv. AD-typy) `F<double>`, `B<double>`, `T<double>`. Místo manuálního přepsání zmíněných typů je použita právě template, která

umožňuje zavolání `operator()` jak s parametrem typu `double`, tak např. s parametrem `B<double>`.

Uživatel pak musí zadat vstupní a výstupní proměnné, vzhledem ke kterým se bude derivovat. Při výpočtu se vytvoří orientovaný acyklický graf, který popisuje výpočet funkce a ze kterého potom knihovna určí jednou z metod derivaci. V následujících podkapitolách popíšeme dva přístupy – zpětnou diferenci a Taylorův rozvoj.

3.3.1 Získání derivace zpětnou metodou diferenciace

V kódu 4 je zobrazena metoda `Diff` z třídy `FunctorDiff`. Nejprve je definována struktura `der`, která je návratovou hodnotou. Obsahuje funkční hodnotu `f` a derivaci `dfdx`. Toho využíváme při interpolaci, kde většinou s derivací potřebujeme i funkční hodnotu a zároveň tím dodržujeme shodu s knihovnou FADBAD++, která také vrací obě hodnoty najednou.

Kód 4: Metoda vracející 1. derivaci

```
1  struct der
2  {
3    double f;
4    double dfdx;
5  };
6
7  virtual der Diff ( const double &i_x )
8  {
9    B<double> x(i_x);    // Initialize arguments
10   Func func;           // Instantiate functor
11   B<double> f(func(x)); // Evaluate function and record DAG
12   f.diff(0,1);         // Differentiate
13
14   der d;               //structure der
15   d.f = f.x();         // Value of function
16   d.dfdx = x.d(0);     // Value of df/dx
17   return d;            // Return function value
18 }
```

Ačkoli by se opět hodilo použít `template`, používáme na pevno daný typ `double`, protože naše metoda je virtuální a nemůže být tudíž zároveň templatovaná (template kód je generován v čase kompilace, zatímco výběr žádané virtuální funkce je záležitostí run-time).

Při výpočtu derivace se nejdříve inicializuje proměnná x (viz kód 4, řádek 9). Poté vytvoříme instanci funkce a zavoláme ji. Tím se zkonstruuje DAG diagram. Příkazem `f.diff(0,1)` se určí diference vzhledem k dané proměnné – (0,1) vyjadřuje proměnnou na pozici 0 v jednoprvkovém poli, knihovna FADBAD++ umí totiž derivovat i podle více proměnných. Na dalších řádcích se jen získané hodnoty vloží do struktury `der`, která je návratovou hodnotou.

3.3.2 Získání derivace z Taylorova rozvoje

Tato metoda byla vyzkoušena, ale zde ji uvádíme pouze jako obecnou ukázku, v naší práci by bez úprav nemohla být použita ze dvou důvodů. Prvním je template parametr `class U`, který bychom museli odstranit a dopředu počítat s typem `double`, protože metoda je virtuální (důvod uveden výše). A za druhé bychom museli vracet hodnotu přes strukturu `der` a vložit do ní za funkční hodnotu první prvek Taylorova rozvoje.

Implementaci, která používá Taylorův rozvoj k získání derivace ukazuje kód 5. V této metodě přibývá vstupní paramter `n`, kterým říkáme, jaký řád derivace žádáme.

Nejprve deklarujeme proměnné AD-typu `T<U>`, kde U může být typicky double. Na řádku 5 inicializujeme proměnnou. Poté inicializujeme první stupeň Taylorova rozvoje, abychom určili vzhledem ke

které proměnné budeme funkci rozvíjet. Na dalším řádku se napočítají funkční hodnoty a vytvoří se DAG. Řádek $f.eval(N)$; napočítá do pole $f[0] \dots f[N]$ členy Taylorova rozvoje do řádu N. Matematicky je Taylorův rozvoj definován takto:

$$T_n = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \cdots + \frac{f^{(n)}(a)}{n!}(x-a)^n \quad (13)$$

Abychom získali derivaci, musíme vynásobit příslušným faktoriálem jednotlivé sčítance (13), které odpovídají $f[0] \dots f[N]$. To provádíme ve `for` cyklu a následně už vracíme žádanou n-tou derivaci.

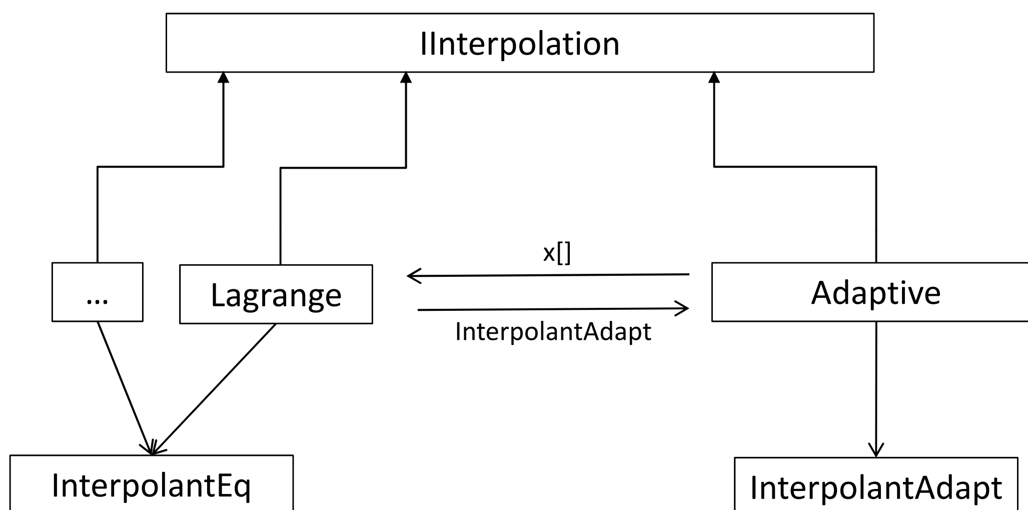
Kód 5: Metoda vracející n-tou derivaci

```
1  template <class U>
2  U Der(U i_x, int n)
3  {
4    T<U> x,f;
5    x = i_x;
6    x[1] = 1;
7    f = fce(x);
8    f.eval(N);
9
10   for(int i = 0; i < N; i++)
11   {
12     f[i] = f[i] * this->Factorial(i);
13   }
14   return f[n];
15 }
```

4 Třídy pro tvorbu interpolace

4.1 Struktura interpolačních tříd

Schéma tříd ukazuje stavbu interpolační části knihovny (plné šipky značí dědičnost, prázdné šipky tok dat – viz obr. 4).



obr. 4: Struktura interpolačních tříd

Rozhraní pro všechny interpolace poskytuje abstraktní třída `IInterpolation` (viz příloha A.5). Každý objekt tvořící interpolaci musí obsahovat řád interpolace `M`, pole uzlových bodů (`vector<double> x`), interval (`double a,b`) a okrajové podmínky (`BCondition lefcond, rightcond`). Proměnnou `double step` definujeme krok mezi uzlovými body v případě ekvidistantního dělení intervalu, u adaptivního přístupu můžeme jeho prostřednictvím určit počáteční krok. V případě určení uzlových bodů z vnějšku (příznak `bool x_defined`) je krok potlačen – ve schématu je pole, ačkoli se jedná o vektor, symbolicky zapsáno `x[]`. Extrapolace je definována řádem extrapolujícího polynomu na každé straně proměnnou `left_degree` a `rightt_degree`.

Většina veřejných metod umožňuje nastavení a správu výše zmíněných proměnných. Interpolace se počítá ve virtuální metodě `Interpolate`, která je implementovaná v potomcích (`Lagrange` a `Adaptive`). Jejím parametrem je reference na interpolovanou funkci.

Rozhraní také obsahuje metodu `ComputeError`, kterou dokážeme určit chybu interpolace, neboli rozdíl mezi původní funkcí a interpolantem (viz kapitola 4.4). Metoda `GetPolynomialErrors` vrací pointer na tzv. prioritní frontu `pq`, která obsahuje chyby jednotlivých polynomů (blíže opět v kapitole týkající se chyby 4.4).

Na levé větvi diagramu (obr. 4) jsou třídy, které implementují konkrétní metody interpolace. Vyhodnocování probíhá dle zadaných parametrů, a to buď s ekvidistantním dělením nebo s definovaným rozložením uzlových bodů – tomu odpovídá i výstupní objekt `InterpolantEq` nebo `InterpolantAdapt` (popsány v následujících podkapitolách). Lagrangeova metoda (`Lagrange`) je navržena pro obecný řád splinu a bude popsána v kapitole 4.2. Na stejné úrovni by se implementovala i Hermitova metoda popsaná v kapitole 2.2.

Třída `Adaptive` využívá již navržených implementací a pouze obsluhuje adaptivitu (neimplementuje žádnou interpolační metodu). Podrobněji bude popsána v samostané kapitole 4.5.

4.1.1 Třídy popisující výstupní objekt – interpolant

Nyní popíšeme třídy definující typ výstupního objektu interpolace. Základní prvky a metody obsahuje třída `InterpolantBase` (viz příloha A.2), která je předkem konkrétních interpolantů. Drží vektor polynomů `polynomials` vytvořené interpolací, dále polynomy extrapo-

lace pro levou a pravou stranu a interval, pro který byla interpolace počítána. Tato třída je potomkem `FunctorDiffBase` (viz obr. 3), a proto implementuje metody `operator()` a `Diff`. Dále umožňuje počítat integrál a určit extrapolaci z krajních polynomů.

Při vyhodnocování hodnoty v nějakém bodě probíhá vyhledávání příslušného intervalu, což zařizuje metoda `FindPolynomial`, která vrací pozici polynomu v poli (vektoru). Virtuální je z důvodu různého vyhledávání ve dvou případech. V prvním můžeme mít interpolaci ekvidistantní, kdy je pozice polynomu určena krokem a hledání tak trvá konstaní dobu (třída `InterpolantEq`, viz příloha A.3). Ve druhém případě se může jednat o interpolaci adaptivní – `InterpolantAdapt` (viz příloha A.3), čili s obecným rozložením uzlových bodů, a polynom se pak hledá metodou půlení intervalu (přístupy vyhledávání v poli blíže uvedeny v Numerical Recipes [4], kapitola 3.4 How to Search an Ordered Table).

4.1.2 Třída `Polynomial` – polynom

Třída `Polynomial` popisuje polynom, který je základní stavební jednotkou interpolantu (deklarace tříd uvedeny v příloze A.4). Člen `degree` nese informaci o řádu polynomu. Velikosti vektoru `coefs`, který obsahuje koeficienty polynomu, odpovídá číslo `degree+1`. Každý polynom má také pomocí `a,b` definovaný interval, na kterém uvažujeme jeho platnost v rámci interpolace.

Metody polynomu jsou víceméně zřejmé z dědičnosti – předkem je `FunctorDiffBase`. Operátor `operator()` vrací hodnotu, metoda `Diff` derivaci v bodě. `Integral` spočítá integrál pro daný interval, nebo bez

parametrů na intervalu definovaném polynomem.

Výpočty hodnot polynomu provádíme tzv. Hornerovým schématem, které vyhodnocení polynomu (s mocninami) převede na posloupnost násobení. V následujících vztazích uvádíme výpočet pro polynom r-tého řádu, viz rovnice (14) pro hodnotu, rovnice (15) pro integrál a rovnice (16) pro derivaci.

$$\begin{aligned} p(x) &= \alpha_r x^r + \alpha_{r-1} x^{r-1} + \ldots + \alpha_1 x + \alpha_0 = \\ &= (\ldots(\alpha_r x + \alpha_{r-1})x + \alpha_{r-2})\ldots + \alpha_1)x + \alpha_0 \end{aligned} \tag{14}$$

$$\begin{aligned} \int_0^x p(t)\,\mathrm{d}t &= \frac{\alpha_r}{r+1}x^{r+1} + \frac{\alpha_r}{r}x^r + \ldots + \frac{\alpha_1}{2}x^2 + \frac{\alpha_0}{1}x = \\ &= \left(\ldots\left(\frac{\alpha_{r+1}}{r+1}x + \frac{\alpha_r}{r}\right)x + \frac{\alpha_{r-1}}{r-1}\right)\ldots + \frac{\alpha_1}{2}\right)x + \alpha_0\right)x \end{aligned} \tag{15}$$

$$\begin{aligned} \frac{\mathrm{d}}{\mathrm{d}x}p(x) &= r\alpha_r x^{r-1} + (r-1)\alpha_{r-1}x^{r-2} + \ldots + 2\alpha_2 x + \alpha_1 = \\ &= (\ldots(r\alpha_r x + (r-1)\alpha_{r-1})x + (r-2)\alpha_{r-2})\ldots \\ &\ldots + 2\alpha_2)x + \alpha_1 \end{aligned} \tag{16}$$

Z těchto předpisů dostáváme jednoduché rekurentní předpisy:

$$\begin{aligned} p_0 &= \alpha_r \\ p_j &= p_{j-1}x + \alpha_t \qquad \text{pro} \quad j = 1, \ldots, r; \quad t = r - j \end{aligned} \tag{17}$$

$$\begin{aligned} I_0 &= \frac{\alpha_r}{r+1}x \\ I_j &= \left(I_{j-1} + \frac{\alpha_t}{t+1}\right)x \qquad \text{pro} \quad j = 1, \ldots, r; \quad t = r - j \end{aligned} \tag{18}$$

$$\begin{aligned} D_0 &= r\alpha_r \\ D_j &= D_{j-1}x + t\alpha_t \qquad \text{pro} \quad j = 1, \ldots, r-1 \quad t = r - j \end{aligned} \tag{19}$$

4.1.3 Třída `BCondition` – okrajové podmínky

Okrajové podmínky splinů byly navrženy co možná nejuniverzálněji a nakonec si vyžádaly vlastní samostatnou třídu `BCondition` (viz příloha A.8). Ta má metody pro přidání, odstranění hodnot a také pro automatické naplnění. Samotné podmínky jsou ukládány do vektoru a to ve struktuře `defvalue`, která mimo hodnotu derivace obsahuje ještě `bool` proměnnou udávající, zdali je daná derivace definována nebo ne.

Metoda automatického doplnění okrajových podmínek `AutoAdd` byla implementována přímo do třídy `BCondition`. Jejími parametry jsou počet podmínek k doplnění a nejvyšší řád derivace v krajním bodě – ty jsou zadány z venku a jsou odvozené od řádu interpolace. Metoda prochází vektor podmínek od nejvyššího řádu derivace a doplňuje nedefinované nulami. Při nezadání okrajových podmínek vede tedy interpolace na tzv. přirozený spline, který má například konkrétně pro kubický druhé derivace nulové.

4.2 Třída `Lagrange` - Lagrangeova interpolace

V této kapitole popíšeme algoritmus tvorby interpolace Lagrangeovou metodou (základní pojmy a byly ukázány v kap. 2.1). Třída `Lagrange` (viz příloha A.6) je potomkem `IInterpolation`, takže dědí všechny členy, nastavovací a další metody. Interpolace je implementována v metodě `Interpolate`, jejímž parametrem je interpolovaný funktor.

Nejprve probíhá kontrolní metoda `Check`, která oznámí chybu, pokud nebyly definovány všechny potřebné parametry. Následně se provede `SetFunctionValues` – napočítají se uzlové body (pokud nejsou

definovány a je k dispozici pouze krok dělení) a v nich funkční hodnoty. Posledním uzlovým bodem je vždy konec intervalu `b`.

Ze znalosti řádu splinu, počtu uzlů a okrajových podmínek můžeme určit parametry matice, a proto se může zavolat `CreateBandMatrix`. Pokud není vložen dostatečný počet okrajových podmínek, doplní se nulami symetricky na obě strany. Vytvoříme objekt `BandMatrixSolve` (rozhraní s knihovnou CLAPACK, viz kap.4.3) s parametry `n` – rozměr čtvercové matice a rozměrem pásu `ku` – počet superdiagonál, `kl` – počet subdiagonál.

$$\begin{aligned}\texttt{n} &= (\text{řád} + 1) \cdot (\text{počet uzlů} - 1)\\ \texttt{ku} &= \text{řád} - \text{počet levých okr. podmínek}\\ \texttt{kl} &= 1 + \text{počet levých okr. podmínek}\end{aligned}$$

Výjimkou je řád 0 (konstantní), kdy je `n` rovno počtu uzlů a parametry `ku`=`kl`=0. Příklad matice byl uveden na obr. 2

Nyní začínáme plnit matici. Metodou `PutBC` se vyplní řádky týkající se okrajových podmínek, což jsou rovnice (7) a (8) v kapitole 2.1. Metodou `PutEquations` se naplňují rovnice (5) a (6) v téže kapitole. Postupuje se po jednotlivých submaticích a zároveň se také plní pravá strana.

Když je matice hotova, objekt `band` zavolá metodu `Solve` a vyřeší pásovou matici. Výsledkem jsou napočítané koefienty polynomů α v poli `bandres`.

Pro vytvoření objektu interpolantu se zavolá metoda `CreateInterpolant`, která nejprve vytvoří vektor polynomů `polynomials` s odpovídajícími intervaly a koeficienty, a pak je předá

konstruktoru interpolantu – buď `InterpolantEq` pro konstantní krok, nebo `InterpolantAdapt` v opačném případě.

Pak zbývá dle zadaných hodnot zavolat u interpolantu metodu `SetExtrapolation` pro extrapolaci a vrátit pointer na interpolant. Extrapolace se určí z krajních polynomů, a to dle zadaného řádu nebo řádu okrajových podmínek, maximálně však řádu splinu.

Pokud si nyní vzpomeneme na motivační příklad uvedený v úvodu (obr. 1), měli bychom mít na tomto místě jasnou představu o tom, jak zapsaný kód funguje.

4.3 Knihovna CLAPACK

Při výpočtu interpolace splinem, jak už jsme naznačili výše, počítáme pásovou matici. Ta je vytvořena ze soustavy rovnic, které závisí na podmínkách dané interpolační metody a jejichž počet se odvíjí od počtu uzlů. Rozhodli jsme se použít volně dostupný balík funkcí lineární algebry CLAPACK (Linear Algebra PACKage doplněný o konverzi z jazyka Fortran do C). Využíváme jeho dvou metod – `dgbtrf_`, která provádí nejprve faktorizaci (LU rozklad a pivotaci), a `dgbtrs_`, která poté aplikuje zpětnou Gaussovu eliminaci pro libovolný počet pravých stran.

Pozn.: Vyšší počet pravých stran by šel výhodně využít i v naší práci, a to v případě, že bychom počítali interpolaci pro více funkcí najednou při neměnném intervalu, počátečních podmínkách a řádu interpolace a jediné, co by zůstalo proměnné, by byla funkce, jejíž hodnoty v uzlech tvoří právě pravou stranu matice.

Při implementaci plnění matice a využití CLAPACKu jsme se po-

týkali s několika nepříjemnostmi v zadávání parametrů zmíněných funkcí. Bylo potřeba nastudovat ukládání prvků pásové matice do jednorozměrného pole, abychom mohli navrhnout rozhraní pro přístup typickými indexy A_{ij}. Přitom jsme museli brát v úvahu, že CLAPACK vychází z jazyka Fortran, kde je pole indexováno od 1, na rozdíl od C/C++, kde je pole indexováno od 0. Výsledkem se stala třída `BandMatrixSolve`, která tvoří rozhraní nad knihovnou. Jsou v ní zabudované výše zmíněné funkce, které zároveň omezujeme pro naše potřeby – uvažujeme jen matici čtvercovou ($m = n$), netransponovanou.

Knihovna je umístěna do adresáře CLAPACK-3.2.1, ze kterého se v makefile linkuje. Dovolím si poznámku, že záleží na pořadí těchto knihoven při linkování, protože na sobě postupně závisí.

4.4 Vyhodnocování chyby interpolace

Chybu interpolace počítáme jako vzdálenost dvou funkcí v normě $L^2(a,b)$, v případě, že chceme znát i chybu v derivaci počítáme normu $W^{1,2}(a,b)$. Vzdálenost dvou funkcí je definována pomocí normy $L^2(a,b)$ takto (viz [1], kapitola 16.1, definice (2)):

Definice. Vzdáleností dvou funkcí $f, g \in L^2(a,b)$ rozumíme (nezáporné) číslo

$$\|h\|_{L^2(a,b)} = \|f - g\|_{L^2(a,b)}\,, \tag{20}$$

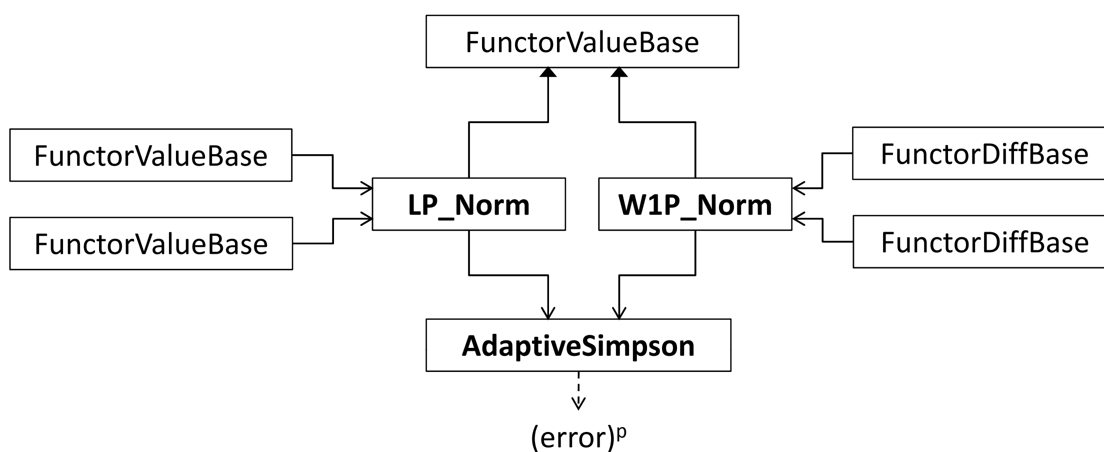
kde pro $h \in L^2(a,b)$ platí

$$\|h\|_{L^2(a,b)} = \left(\int_a^b (f-g)^2 \,\mathrm{d}x \right)^{\frac{1}{2}}. \tag{21}$$

Součet vzdáleností funkcí a jejich derivací je dán v normě $W^{1,2}(a,b)$ vztahem

$$\|h\|_{W^{1,2}(a,b)} = \|h\|_{L^2(a,b)} + \|h'\|_{L^2(a,b)} \, . \tag{22}$$

Strukturu výpočtu dokumentuje schéma na obr. 5. Dvě porovnávané funkce (funktory `FunctorValueBase` resp. `FunctorDiffBase`) jsou parametry konstruktoru objektů norem `LP_Norm` resp. `W1P_Norm`, které počítají druhou mocninu rozdílu funkcí, resp. funkcí a jejich derivací. Třídy jsou navrženy obecně pro p-tou mocninu.



obr. 5: Struktura výpočtu norem

Integrál spočítáme statickou třídou `AdaptiveSimpson`, která implementuje adaptivní Simpsonovu metodu. Jako parametr vkládáme funktor odvozený od `FunctorValueBase` – jeho potomky jsou i třídy norem. Pro výpočet užíváme tříbodového Simpsonova pravidla a rekurzivní dělení intervalu až do splnění požadované přesnosti.

Simpsonovo pravidlo (kapitola 4 v [4])*:*

$$S^{(m)} = \int_{x1}^{x3} f(x)\,\mathrm{d}x = h\left[\frac{1}{3}f_1 + \frac{4}{3}f_2 + \frac{1}{3}f_3\right] + O(h^5 f^{(4)}) \tag{23}$$

Kritérium ukončení adaptivní metody S. pravidla (blíže v [5])*:*

$$\left| S^{(m)}f - S^{(2m)} \right| \leq 15\epsilon, \tag{24}$$

Přičemž platí, že $S^{(m)}f$ je hodnota Simpsonova pravidla (pro m bodů) před rozdělením intervalu a $S^{(2m)}f$ (pro $2m$ bodů) po rozdělení intervalu. Chyba ϵ odpovídá zadané maximální požadované chybě – ve třídě `IInterpolation` je definována makrem `SIMPSON_TOLERANCE`.

Tříbodová formule je přesná pro polynomy do řádu 3 včetně. Po integraci p-té mocniny nesmíme zapomenout výsledek odmocnit. Tím dostáváme informaci o chybě interpolace – vzdálenosti funkce a interpolantu.

V implementaci výpočtu chyby mezi funkcí a interpolantem v metodě `ComputeError` třídy `IInterpolation` vstupují do normy ve `for` cyklu jednotlivé polynomy (viz kód 6, kde `f` je funkce, `g` je interpolant). Počítají se postupně mezi uzlovými body dílčí chyby `p_err`, které jsou přepočítávány na relativní (děleny délkou intervalu, zatímco absolutní jsou přičítány k celkové chybě `tot_err`) a spolu s indexem polynomu vkládány do tzv. prioritní fronty. Ta je zároveň řadí dle zvolené podmínky – sestupně dle velikosti chyby, a tak máme přehled o tom, který polynom přispívá k celkové chybě nejvíc. Prioritní fronty pak využíváme při adaptivitě.

4.5 Třída `Adaptive` – adaptivní interpolace

Třída `Adaptive` (deklarace metod viz příloha A.7) je stejně jako každá interpolační třída potomkem rozhraní `IInterpolation`. Využívá ale již navržených tříd k výpočtu interpolace a obsluhuje hlavně výpočet

Kód 6: Výpočet chyby v metodě `ComputeError`

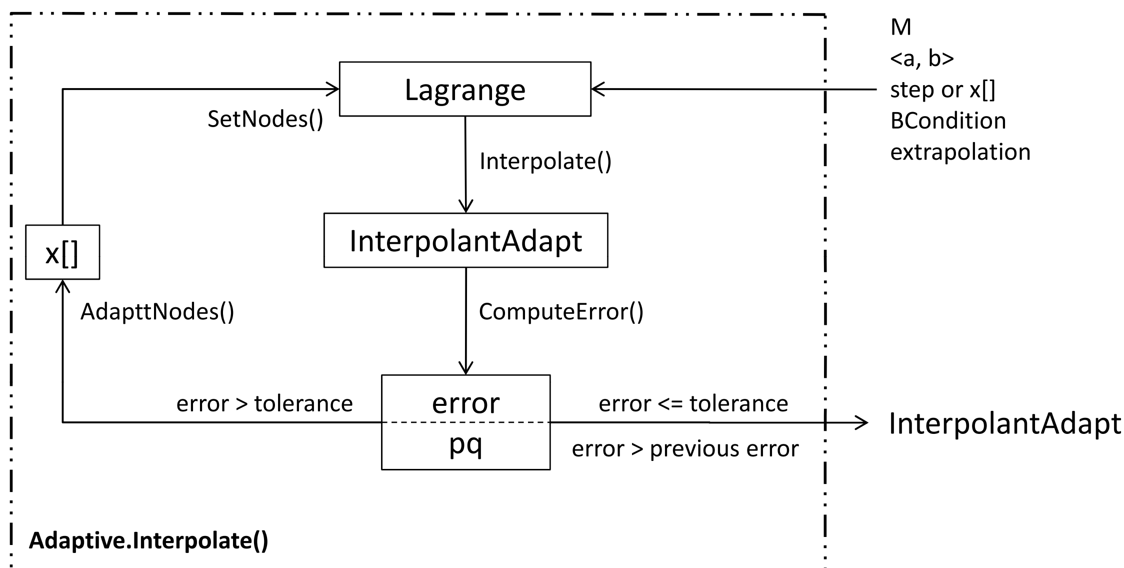
```
for(unsigned long i = 0; i < g->GetCount(); i++)
{
  LP_Norm norm(f,g->GetPol(i),2);
  p_err.i = i;

  //absolute polynomial error
  p_err.err = sqrt(AdaptiveSimpson::AdaptSimpson( norm,
                   g->GetPol(i)->GetA(),
                   g->GetPol(i)->GetB(),
                   SIMPSON_TOLERANCE) );
  //increase the absolute total error
  tot_err += p_err.err;

  //p_err.err convertion absolute -> relative
  p_err.err /= (g->GetPol(i)->GetB()-g->GetPol(i)->GetA());

  pq.push(p_err); //puts in priority queue
}
```

chyby a úpravu rozložení uzlových bodů. Umožňuje nastavit proměnnou `tolerance`, kterou zadáváme maximální absolutní chybu interpolace a která je měřítkem pro zjemňování intervalů. Na obr. 6 je znázorněn algoritmus adaptivity.



obr. 6: Algoritmus adaptivní interpolace

Na začátku vytvoří objekt příslušné interpolační metody (v konkrétním případě `Lagrange`) se všemi potřebnými parametry (řád `M`, interval `<a,b>`, atd.). Pokud jsou definovány počáteční uzlové body `x[]`, jsou předány, jinak se předává počáteční krok `step`. Počáteční podmínky jsou předány objektem `BCondition` a extrapolace řádem extrapolujícího polynomu.

Poté začíná adaptivní cyklus. Interpolační třída vrátí vypočítaný interpolant (objekt `InterpolantAdapt`). Vyhodnotí se, zdali byla dosažena požadovaná přesnost pomocí metody `ComputeError` a v pozitivním případě vrátí interpolant. V opačném případě se metodou `AdaptNodes` přidají uzlové body a interpolace se opakuje odesláním adaptovaných uzlů příslušné třídě, která znovu vypočítá interpolaci. Kontroluje se také rozdíl s předchozí chybou, aby se v případě nárůstu iterace ukončila.

Uzlové body se doplňují do intervalů, kde je největší relativní chyba (chyba přepočtená na jednotkový interval). Tuto informaci získáme z prioritní fronty `pq`, která vrací polynomy (čili i intervaly) sestupně podle vypočtené chyby. Direktivou `#define P 0.05` je definovaný počet intervalů, který se bude při iteraci dělit. Hodnota 0.05 říká, že 5% intervalů, kde je chyba největší, se rozdělí na polovinu. Konstantu `P` lze v hlavičkovém souboru zvětšit respektive zmenšit a tato změna způsobí urychlení resp. zpomalení hledání interpolace pro zadanou toleranci chyby.

5 Testování systému

5.1 Odlaďování

Nejprve pár poznámek k poznatkům získaných při odlaďování, které můžeme považovat částečně za testování systému. Byly použity jednoduché známé funkce x^3 a x^5. Výsledky interpolací byly očekávány přesné, koeficienty polynomů splinu rovné celým číslům (za předpokladu dělení na intervaly o délce 1.0). Bohužel tomu tak není (jak ukazují výsledky – kód 7) zřejmě v důsledku zaokrouhlovacích chyb ve výpočtu pásové matice. Vypsané koeficienty náleží prvním 7 polynomům nahrazujícím funkci x^3 na intervalech délky 1.0 (interpolace pokračovala až do 30.0). Při velkém množství dělených intervalů (to vede ke zvětšování matice) se chyba dále šíří a narůstá. To by pak svědčilo o nestabilitě algoritmu nebo špatné podmíněnosti matice.

Následující test má za cíl především porovnat rychlosti vyhodnocování různých interpolantů a zmíněná numerická nestabilita nemá na měření časů významný vliv.

Kód 7: Vypis koeficientů polynomů

```
1 p[0]={0 | 5.55111512312578e-16 | 0 | 0.999999999999999}
2 p[1]={1 | 3 | 3 | 1}
3 p[2]={8 | 12 | 6 | 1}
4 p[3]={27 | 27 | 9 | 0.999999999999995}
5 p[4]={64 | 48 | 12 | 1.00000000000002}
6 p[5]={125 | 75 | 15 | 0.999999999999921}
7 p[6]={216 | 108 | 17.9999999999998 | 1.00000000000033}
8 p[7]={343 | 147 | 21.0000000000008 | 0.999999999998756}
```

5.2 Test funkcí FK a FQ

Testování funkce systému, jsme provedli na vybraných hydrologických funkcích `FK` (hydraulická vodivost) a `FQ` (vodní saturace), které byly přepsány z fortranového kódu do C++. Výsledkem testů je porovnání chyb interpolací, velikostí vektorů s polynomy a doby trvání evaluace `n` hodnot (`n` je volená proměnná větší než 10^7).

Před samotným testem bylo potřeba zajistit, aby chyba interpolací byla srovnatelná. Proto byla pro interpolace s ekvidistantním dělením napsána funkce `FindStep`, která opakuje interpolaci se zmenšujícím se krokem, dokud chyba neklesne pod zadanou toleranci. Takto se vytvořily objekty konstatní (spline 0.řádu), lineární (spline 1.řádu) a kubické (spline 3.řádu) interpolace s konstantním krokem. Stejná tolerance pak byla parametrem adaptivní interpolace.

Pro měření času výpočtu hodnot funkcí a interpolantů byla napsána funkce `MeasureTime`, jejímž vstupem je funktor `FunctorValueBase` a počet volání výpočtu `n`.

Kód 8: Parametry testu

```
1 int n[] = {5e+7, 7e+7, 9e+7, 1e+8, 3e+8, 5e+8};
2 double a = -10.0;
3 double b = -0.126500012501607;
4 double step = 1.0;
5 double tolerance = 1.0e-7;
6 double tolerance_0 = 5.0e-4; //for constant interpolation
7 //boundary conditions
8 lag->AddCond(IInterpolation::LeftBC,2,fk.Diffn(a,2));
9 lag->AddCond(IInterpolation::RightBC,2,fk.Diffn(b,2));
```

V kódu 8 jsou uvedeny parametry testu. Interval končí v bodě `b`,ve kterém končí hladkost interpolovaných funkcí (od tohoto bodu jsou funkce nahrazeny konstantou, a proto je nevhodné zatěžovat inter-

polaci tímto nehladkým bodem). Všechny testované interpolace vycházejí z Lagrangeovy metody. Pro konstatní interpolaci byla zvolena větší tolerance, protože pak by test trval neúměrně dlouho a pole konstant by pravděpodobně při nějaké hodnotě překročilo možnosti alokace paměti. Okrajové podmínky pro kubický spline tvoří druhé derivace v krajních bodech spočítané knihovnou FADBAD++.

řád splinu	chyba interpolace	počet iterací	počet polynomů
FK			
0 (konstantní)	$4.54 \cdot 10^{-4}$	13	40442
1 (lineární)	$9.30 \cdot 10^{-8}$	13	40442
3 (kubický)	$1.07 \cdot 10^{-8}$	9	2528
FQ			
0 (konstantní)	$3.94 \cdot 10^{-4}$	15	161768
1 (lineární)	$6.16 \cdot 10^{-8}$	11	10111
3 (kubický)	$2.66 \cdot 10^{-8}$	10	5056

Tabulka 1: Tvorba interpolace – ekvidistantní dělení

V tabulce 1 jsou výsledky tvorby interpolace s ekvidistantním dělením, kdy se krok hledal metodou `FindStep`. V levém sloupci je název interpolované funkce a řád splinu (0, 1 a 3). Ke každému typu je uvedena chyba, počet iterací, kterými jsme dosáhli zadané tolerance a velikost vektoru polynomů, což odpovídá počtu intervalů.

Konstantní interpolace je zde uvedena, aby ukázala, že s ní rozhodně nelze vystačit – s chybou o 4 řády větší než ostatní interpolace musí mít i tak vektor konstant mnohonásobně více položek. Srovnat můžeme lépe interpolaci lineární a kubickou. Velikostí počtu polynomů vychází kubická lépe a dostatečně jemné rozdělení intervalu je

také nalezeno v menším počtu iterací.

Tabulka 2 nám podává obdobné informace jako tabulka 1, zde ovšem byla použita adaptivní metoda interpolace. Je nutné připomenout, že se během iterace v tomto případě dělí na polovinu pouze 5% intervalů. Výrazně se tak projevuje efektivnost dělení intervalů pouze v místech, která přispívají největší chybou. Adaptivní přístup má obzvláště pro kubický spline dobré výsledky. Počet polynomů potřebných k docílení chyby menší než 10^{-7} je oproti lineární interpolaci minimální.

řád splinu	chyba interpolace	počet iterací	počet polynomů
FK			
0 (konstantní)	$4.89 \cdot 10^{-4}$	223	975206
1 (lineární)	$9.81 \cdot 10^{-8}$	156	37095
3 (kubický)	$8.95 \cdot 10^{-8}$	37	102
FQ			
0 (konstantní)	$4.89 \cdot 10^{-4}$	196	261201
1 (lineární)	$9.69 \cdot 10^{-8}$	130	10426
3 (kubický)	$9.33 \cdot 10^{-8}$	26	56

Tabulka 2: Tvorba interpolace – adaptivní dělení

V tabulkách 3 a 4 jsou shrnuta měření časů výpočtu hodnot funkcí a jejich interpolantů v `n` bodech. Jsou zde vždy uvedeny časy evaluace původní funkce a pod nimi časy jednotlivých interpolantů.

Výsledky v tabulce 3 patří interpolacím s ekvidistantním dělením. Zde stojí za zmínku, že čas náležící kubické interpolaci je větší než v případě lineárním, ačkoli počet polynomů je několikrát menší.

n	$5 \cdot 10^7$	$7 \cdot 10^7$	$9 \cdot 10^7$	10^8	$3 \cdot 10^8$	$5 \cdot 10^8$
FK	80.02	111.51	124.10	136.53	449.67	672.79
FK 0 (konstantní)	2.46	2.58	3.60	3.76	13.07	18.49
FK 1 (lineární)	2.77	2.91	3.78	4.25	13.32	20.92
FK 3 (kubický)	3.30	3.51	4.71	6.06	15.56	25.11
FQ	17.26	18.10	24.36	28.29	85.57	129.43
FQ 0 (konstantní)	2.35	2.58	3.35	4.16	13.71	18.62
FQ 1 (lineární)	2.77	2.90	4.01	4.86	12.94	20.85
FQ 3 (kubický)	3.35	3.51	4.80	5.70	16.74	25.12

Tabulka 3: Čas [s] výpočtu `n` hodnot – ekvidistantní dělení

n	$5 \cdot 10^7$	$7 \cdot 10^7$	$9 \cdot 10^7$	10^8	$3 \cdot 10^8$	$5 \cdot 10^8$
FK	66.7	107.92	131.74	140.29	412.93	701.13
FK 0 (konstantní)	8.75	15.95	15.50	17.36	54.29	102.23
FK 1 (lineární)	6.87	12.65	12.55	14.92	44.33	69.54
FK 3 (kubický)	3.90	7.06	7.09	7.74	23.80	44.82
FQ	13.91	24.13	26.17	28.14	79.55	145.06
FQ 0 (konstantní)	8.29	14.44	15.87	16.88	47.55	77.23
FQ 1 (lineární)	6.23	10.95	11.25	13.69	36.66	59.44
FQ 3 (kubický)	3.76	6.15	6.29	7.30	21.23	35.40

Tabulka 4: Čas [s] výpočtu `n` hodnot – adaptivní dělení

Tabulka 4 týkající se adaptivních interpolací je již zajímavější. Časy evaluace kubických splinů jsou kratší než u lineárních, což spolu s mnohonásobně menšími rozměry pole polynomů potvrzuje jejich význam. Na druhou stranu při porovnání obou tabulek mezi sebou vidíme u adaptivních kubických splinů, že i přes menší rozměr pole polynomů jsou časy delší. To je způsobeno složitějším vyhledáváním v poli narozdíl od ekvidistantního přístupu, kde stačí dosazení kroku do jediného

vzorce a čas hledání je tak konstantní pro pole libovolného rozměru. Rozdíl časů bychom mohli ještě snížit použitím sofistikovanější metody vyhledávání.

Bude tedy hodně záležet na požadavcích a obtížnosti dosáhnout zadané tolerance pro danou funkci. Můžeme klást důraz na minimalizaci nároků na paměť a potom zajisté vyhraje volba adaptivity i za cenu ztráty času potřebného na procházení polem polynomů. Význam minimalizace velikosti pole se projeví, vezmeme-li při reálném výpočtu v úvahu i velikost cache paměti. Především pokud bude interpolace použita v rámci jiného výpočtu, kdy se bude načítat opakovaně celé pole polynomů do paměti, což může zpomalit vyhledavání polynomů a omezit i hlavní výpočet.

Pokud si můžeme dovolit plýtvat paměťovým prostorem, pak může být výhodnější rozdělení na menší, ale ekvidistantní intervaly a minimalizovat tak čas přístupu k žádanému polynomu v poli. To by se projevilo především u volání interpolace v náhodných bodech.

Správný výběr vhodné interpolace se bude lišit i podle vlastností interpolované funkce.

6 Závěr

Výsledkem práce je knihovna tříd a metod umožňujících práci s funkcemi a s jejich interpolací. Ve struktuře funktorových tříd slouží abstraktní třídy jako rozhraní nad jednotlivými objekty a dávají tak při psaní dalšího kódu široké možnosti. Stejným způsobem můžeme přistupovat jak k výpočtu hodnoty původní funkce, tak jejího interpolantu, či jediného polynomu splinu. To samé platí i pro získání derivace, ačkoli se v jednotlivých případech počítá různými způsoby.

Ve struktuře interpolačních tříd se podařilo oddělit třídy, které přímo implementují konkrétní matematický postup, od tříd definujících obecné vlastnosti a funkce interpolace. Výpočet chyby (rozdílu funkcí) je univerzální a lze ho použít pro libovolné dva funktory. Adaptivita (změna rozložení uzlových bodů v místech největší chyby) je nezávislá na použité interpolační metodě. Tyto důležité vlastnosti umožňují pod navržené rozhraní přidat jakoukoli třídu, která bude umět vypočítat interpolaci (nebo obecně aproximaci), a zajistit tak její okamžitou funkčnost v rámci celého systému.

Aby bylo možné pracovat s konkrétní interpolací byla do systému vybrána a vyzkoušena jedna interpolační metoda – Lagrangeova. Byla navržena pro libovolný řád polynomů, libolvolné rozložení uzlových bodů a volitelné okrajové podmínky.

Nedosáhla však očekávaných výsledků u interpolací vyššího řádu. Jedním z problémů je numerická nestabilita v řešení vzniklé pásové matice. Dokumentuje ji test interpolace jednoduchých jednočlenů x^3 a x^5, které by samozřejmě spline měl interpolovat přesně. Tuto chybu

lze odstranit zlepšením podmíněnosti matice, čehož by šlo pravděpodobně dosáhnout už přepočítáním intervalů mezi uzlovými body na jednotkové, čímž by se omezila velikost koeficientů polynomu a velikost šíření chyby v rámci řešení matice.

Druhý problém se týká samotného principu zvolené metody. Pro použití interpolace byl problém nahrazení funkce zjednodušen na proložení bodů splinem. Je možné, že lepších výsledků u obecných funkcí by bylo dosaženo metodou, ve které by byla povolena i odchylka v uzlových bodech. Podmínkou by pak nebylo přesné dodržení rovnosti v uzlových bodech, ale minimalizace chyby na jednotlivých intervalech.

Z testů výpočtu velkého množství hodnot vyplývá, že hledání zjednodušeného popisu funkce smysl má. Čas, který je potřeba k výpočtu hodnot splinu, je o 70 až 95% kratší, než při evaluaci původní funkce. Je možné používat konstantní a lineární interpolaci. Adaptivitu lze aplikovat na libovolnou interpolační (i obecně aproximační) metodu. U interpolace splinem vyššího řádu jsme objevili několik úskalí a poukázali na pár kritických míst. Jejich ošetření či zavedení jiných složitějších matematických metod již přesahuje rámec této práce a může být tématem dalších projektů. Jak však ukázal závěrečný test, jsou spliny vyššího řádu použitelné a za vhodných podmínek mohou dosáhnout dobrých výsledků. Nejlepší metodou se podle testu jeví lineární ekvidistantní interpolace. Při nalezení jiného algoritmu, lze vytvořit novou aproximační třídu a jednoduše ji zahrnout do systému.

Použitá literatura

[1] Rektorys Karel a spolupracovníci. *Přehled užité matematiky I.* 7. vydání Praha: Prometheus, 2000, ISBN: 80-7196-180-9

[2] Rektorys Karel a spolupracovníci. *Přehled užité matematiky II.* 7. vydání Praha: Prometheus, 2000, ISBN: 80-7196-181-7

[3] Miroslav Brzezina a kol. *Matematika IV.* Liberec, 1996

[4] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing.* Second Edition. Cambridge University Press 1992. ISBN: 0-521-43108-5

[5] Rowland J.H., Y.L.Varol. *Exit Criteria for Simpson's Compound Rule* [online]. Mathematics of Computation, volume 26, number 119, July 1972. URL: http://www.ams.org/journals/mcom/1972-26-119/S0025-5718-1972-0341823-8/S0025-5718-1972-0341823-8.pdf

[6] Ole Stauning and Claus Bendtsen. *FADBAD++* [online], 1996-2007. URL: http://www.fadbad.com/fadbad.html

[7] Anderson, E. and Bai, Z. and Bischof, C. and Blackford, S. and Demmel, J. and Dongarra, J. and Du Croz, J. and Greenbaum, A. and Hammarling, S. and McKenney, A. and Sorensen, D. *LAPACK Users' Guide. Third Edition.*

Philadelphia: Society for Industrial and
Applied Mathematics 1999. ISBN: 0-89871-447-8

[8] The C++ Resource Network. *C++ Language Tutorial* [online], listopad 2007.
URL: http://www.cplusplus.com/doc/tutorial/templates/

A Zdrojové kódy vybraných tříd

V následujících přílohách nalezneme deklarace nejdůležitějších tříd systému, které nebyly zmíněny přímo v textu. Metody jsou pouze deklarovány vzhledem k rozsahu zdrojového kódu a bez komentářů. Pro konkrétní implementace a popisy je nutné nahlédnout do zdrojových souborů.

A.1 FunctorDiffBase

```
1  struct der
2  {
3    double f;
4    double dfdx;
5  };
6  class FunctorDiffBase : public FunctorValueBase
7  {
8    public:
9      virtual der Diff(const double &x) = 0;
10     virtual double operator() ( const double &x ) = 0;
11 };
```

A.2 InterpolantBase

```
1  class InterpolantBase : public FunctorDiffBase
2  {
3    protected:
4      std::vector<Polynomial> polynomials;
5      Polynomial left;
6      Polynomial right;
7      unsigned long polynomialcount;
8      double a,b;
9
10     virtual unsigned long FindPolynomial(
11         const double &x) = 0;
12
13   public:
14     InterpolantBase(
15         std::vector<Polynomial> &polynomials );
16     ~InterpolantBase( void );
17
18     inline double GetA();
19     inline double GetB();
20     inline unsigned long GetCount();
21     inline Polynomial* GetPol( unsigned long i );
```

```
22
23     virtual double operator() ( const double &x );
24     virtual der Diff( const double &x );
25     double Integral( const double &u,
26         const double &v );
27
28
29     void SetExtrapolation(
30         const Polynomial &left,
31         const Polynomial &right );
32     void SetExtrapolation (
33         const unsigned char &left_degree,
34         const unsigned char &right_degree );
35 };
```

A.3 InterpolantEq a InterpolantAdapt

```
1  class InterpolantAdapt : public InterpolantBase
2  {
3    private:
4      virtual unsigned long FindPolynomial(
5          const double& x);
6    public:
7      InterpolantAdapt(
8          std::vector<Polynomial> &polynomials);
9  };
10
11 class InterpolantEq : public InterpolantBase
12 {
13   private:
14     double step;
15     virtual unsigned long FindPolynomial(
16         const double& x);
17   public:
18     InterpolantEq(
19         std::vector<Polynomial> &polynomials,
20         const double &step);
21 };
```

A.4 Polynomial

```
1  class Polynomial : public FunctorDiffBase
2  {
3    private:
4      unsigned char degree;
5      double a,b;
6      std::vector<double> coefs;
7    public:
8      Polynomial();
9      Polynomial( const unsigned char &degree );
```

```
10      Polynomial( const double &a, const double &b,
11                  const std::vector<double> &coefs );
12      Polynomial( const Polynomial &pol );
13      ~Polynomial( void );
14
15      inline double GetA();
16      inline double GetB();
17      inline std::vector<double> *GetCoefs();
18      void WriteCoef( void );
19
20      void SetInterval( const double &a, const double &b );
21      void SetCoeficients( double *coeficients,
22          const unsigned int &size );
23
24      virtual double operator() ( const double &x );
25      virtual der Diff( const double &x );
26      double Integral( const double &u, const double &v );
27      double Integral( void );
28  };
```

A.5 IInterpolation

```
1   struct ErrorNum {
2       double err;
3       unsigned long i;
4   };
5   class CompareErrorNum {
6   public: bool operator()(ErrorNum& err1, ErrorNum& err2);
7   };
8   class IInterpolation
9   {
10    protected:
11      std::vector<double> x;
12      bool x_defined;
13      double a,b;
14      double step;
15      unsigned int M;
16      bool leftcond_defined, rightcond_defined;
17      BCondition *leftcond;
18      BCondition *rightcond;
19      bool extrapolation_defined;
20      unsigned char left_degree;
21      unsigned char right_degree;
22      std::vector<bool> checks;
23      std::priority_queue< ErrorNum,
24          std::vector<ErrorNum>,
25          CompareErrorNum> pq;
26    public:
27      enum BCkind {LeftBC, RightBC};
28      IInterpolation( void );
29      ~IInterpolation( void );
30      inline unsigned int GetDegree();
31      inline std::vector<double> *GetNodes();
32      inline double GetA();
```

```
33      inline double GetB();
34      inline double GetStep();
35      inline std::priority_queue< ErrorNum,
36          std::vector<ErrorNum>, CompareErrorNum>
37          *GetPolynomialErrors();
38      void ClearPolynomialErrors();
39      void SetDegree( const unsigned int& M);
40      void SetNodes ( const std::vector<double> &x );
41      void SetInterval ( const double &a, const double &b );
42      void SetStep ( const double& step );
43      void AddCond ( BCkind cond,
44          const unsigned int& derivate,
45          const double& value );
46      void AddCond ( BCkind cond,
47          const BCondition& condition );
48      void ClearCondition( BCkind cond );
49      void SetExtrapolation(
50          const unsigned char& left_degree,
51          const unsigned char& right_degree );
52      double ComputeError( FunctorValueBase* f,
53          InterpolantBase* g);
54      virtual bool Check() = 0;
55      virtual InterpolantBase* Interpolate(
56          FunctorValueBase& f) = 0;
57  };
```

A.6 Lagrange

```
1   class Lagrange  : public IInterpolation
2   {
3     private:
4       std::vector<double> f;
5       BandMatrixSolve *band;
6
7       virtual bool Check();
8       void SetFunctionvalues ( FunctorValueBase &func );
9       void CreateBandMatrix ();
10      void PutBC ();
11      void PutEquations ();
12      InterpolantBase* CreateInterpolant ( double *bandres);
13    public:
14      Lagrange ( void );
15      ~Lagrange ( void );
16      virtual InterpolantBase* Interpolate(
17          FunctorValueBase& func);
18  };
```

A.7 Adaptive

```
1  class Adaptive : public IInterpolation
2  {
3    private:
4      double tolerance;
5
6      virtual bool Check( void );
7      void FillNodes();
8      void AdaptNodes();
9    public:
10     Adaptive( void );
11     ~Adaptive( void );
12     void SetTolerance( const double& tolerance );
13     virtual InterpolantBase* Interpolate (
14         FunctorValueBase &func );
15 };
```

A.8 BCondition

```
1  struct defvalue
2  {
3    double value;
4    bool defined;
5  };
6  class BCondition
7  {
8    private:
9      std::vector<defvalue> condition;
10     unsigned int count;
11   public:
12     BCondition( void );
13     BCondition( const BCondition& source );
14     ~BCondition( void );
15     std::vector<defvalue> *GetCond();
16     unsigned int GetCount();
17     void AddCond( const unsigned int &derivate,
18            const double &value );
19     void ClearCondition();
20     void AutoAdd( const unsigned int &number_of_conditions,
21       const unsigned int &highest_order_of_derivate );
22 };
```