VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ
DEPARTMENT OF TELECOMMUNICATIONS

IMPLEMENTACE ČASOVĚ-VARIANTNÍHO KONVOLUČNÍHO PROCESORU
IMPLEMENTATION OF TIME-VARIANT CONVOLUTION PROCESSOR

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. Tomáš Brhel**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. Miroslav Balík, Ph.D.**
SUPERVISOR

BRNO 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní obor **Audio inženýrství**
Ústav telekomunikací

Student: Bc. Tomáš Brhel ***ID:*** 164590
Ročník: 2 ***Akademický rok:*** 2017/18

NÁZEV TÉMATU:

Implementace časově-variantního konvolučního procesoru

POKYNY PRO VYPRACOVÁNÍ:

Popište problematiku provádění rychlé konvoluce v reálném čase bez procesního zpoždění. Nastudujte teorii tzv. živé konvoluce, která umožňuje provádění konvoluce dvou signálů prakticky neomezené délky v reálném čase a bez procesního zpoždění. Podrobně a výstižně dokumentujte algoritmus živé konvoluce bez procesního zpoždění, následně celý algoritmus implementujte v Matlabu. Algoritmus musí být připraven pro jeho pozdější implementaci v reálném čase a bez procesního zpoždění. Popište možnosti optimalizace a snížení výpočetní náročnosti tohoto algoritmu. Nastudujte a popište možnosti knihoven, které umožňují multiplatformní vícevláknové zpracování signálu v reálném čase s využitím zásuvných modulů VST. V návaznosti pak realizujte celý algoritmus časově-variantního konvolučního procesoru jako zásuvný modul VST s grafickým uživatelským rozhraním.

DOPORUČENÁ LITERATURA:

[1] Wefers, F. Partitioned convolution algorithms for realtime auralization [online], Logos Verlag Berlin GmbH, 2015 [cit. 10.9.2017]. Dostupné z: http://goo.gl/QDoxQT

[2] Brandtsegg, Ø., Saue, S. Live convolution with time-variant impulse response. Proceedings of the 19th International Conference on Digital Audio Effects (DAFx-16), Brno, 2016, ISSN 2413-6700

Termín zadání: 5.2.2018 ***Termín odevzdání:*** 21.5.2018

Vedoucí práce: Ing. Miroslav Balík, Ph.D.
Konzultant:

prof. Ing. Jiří Mišurec, CSc.
předseda oborové rady

ABSTRAKT

V této práci je řešen návrh univerzálního časově-variantního konvolučního FIR procesoru. Návrh je připraven pro zpracování velmi dlouhých posloupností s potlačením procesního zpoždění. V práci jsou dále navrženy metody pro plynulý přechod mezi různými impulsovými odezvami. Algoritmus je nejdříve realizován v prostředí Matlab a následně jako VST zásuvný modul.

KLÍČOVÁ SLOVA

FIR systém, časově-variantní systém, živá konvoluce, prolnutí, impulsová odezva, kmitočtová charakteristika, FFT, VST, C++.

ABSTRACT

In this thesis there is a design of universal time-variant convolution FIR processor. Proposed algorithm is ready to process long signals with reduced process delay. Furthermore, there are proposed methods for smooth crossfade between different impulse responses. Algorithm is implemented in Matlab environment and later as a VST plugin.

KEYWORDS

FIR system, time-variant system, live convolution, crossfade, impulse response, frequency response, FFT, VST, C++.

BRHEL, Tomáš *Implementace časově-variantního konvolučního procesoru*: diplomová práce. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2018. 67 s. Vedoucí práce byl Ing. Miroslav Balík, Ph.D.

PROHLÁŠENÍ

Prohlašuji, že svou diplomovou práci na téma „Implementace časově-variantního konvolučního procesoru“ jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu diplomové práce panu Ing. Miroslavu Balíkovi, Ph.D. za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci.

Brno ....

podpis autora

SIX research centre
sensor, information and communication systems

Faculty of Electrical Engineering
and Communication

Brno University of Technology
Technicka 12, CZ-61600 Brno, Czechia

http://www.six.feec.vutbr.cz

Výzkum popsaný v této diplomové práci byl realizovaný v laboratořích podpořených projektem Centrum senzorických, informačních a komunikačních systémů (SIX); registrační číslo CZ.1.05/2.1.00/03.0072, operačního programu Výzkum a vývoj pro inovace.

MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

EVROPSKÁ UNIE
EVROPSKÝ FOND PRO REGIONÁLNÍ ROZVOJ
INVESTICE DO VAŠÍ BUDOUCNOSTI

2007–13
OP Výzkum a vývoj pro inovace

OBSAH

SEZNAM OBRÁZKŮ

ÚVOD

Tato práce se zabývá implementací univerzálního časově-variantního konvolučního FIR procesoru. Jedná se o systém zpracovávající velmi dlouhé vstupní posloupnosti časově-variantními impulsovými odezvami. Princip je založen na tzv. živé konvoluci, která bude představena v teoretické části práce. Pro funkční systém využitelný například v hudebních aplikacích je však nezbytné vyřešit dva zásadní problémy.

Jedním z nich je potlačení nežádoucího procesního zpoždění, které doprovází zpracování velmi dlouhých posloupností impulsovou odezvou téměř nekonečné délky. V teoretické části práce je navrženo řešení vycházející z vhodného rozdělení dlouhé impulsové odezvy. Algoritmus je za určitých podmínek schopen zpracovávat posloupnosti impulsovou odezvu délky např. 500 000 vzorků bez slyšitelného procesního zpoždění. Funkční algoritmus však předpokládá zpracování úloh na více vláknech procesoru.

Druhý zásadní problém se vyskytuje při změně impulsových odezev v reálném čase. Při skokové změně dochází ke vzniku slyšitelných přechodových jevů, které se na poslech projevují jako zvukové artefakty. V práci budou navrženy efektivní metody pro potlačení těchto jevů, které lze později snadno implementovat pro funkční procesor.

Výše uvedená řešení lze využít k realizaci univerzálního časově-variantního konvolučního procesoru, který může být později realizován např. jako *VST* zásuvný modul pro využití v hudebních aplikacích. Pomocí takového procesoru lze zpracovávat hudební signál dlouhou impulsovou odezvou nebo dokonce jiným hudebním signálem a vytvářet tak zcela nové zvuky.

V praktické části práce je připravena simulace časově-variantního konvolučního procesoru s potlačením přechodového jevu. Simulace je implementována v prostředí Matlab. Na základě této simulace jsou v jazyce C++ a s využitím frameworku *JUCE* realizovány dva zásuvné moduly pro ověření funkčnosti algoritmu v reálném čase. V poslední kapitole jsou diskutovány dosažené výsledky a zákonitosti vycházející z praktické realizace algoritmu.

1 IMPLEMENTACE RYCHLÉ KONVOLUCE V REÁLNÉM ČASE

Implementace rychlé konvoluce je založena na výpočtu diskrétní Fourierovy transformace (DFT) v kmitočtové oblasti. Tato teorie je již popsána v mé bakalářské práci [1], proto z ní budu do jisté míry čerpat.

Následující text a rovnice vychází z [6] a popisují přechod od obecné diferenční rovnice diskrétní konvoluce 1.1 po výpočet pomocí diskrétní Fourierovy transformace v kmitočtové oblasti pro zpracování vstupní posloupnosti $x(n)$ systémem s impulsovou odezvou $h(n)$

$$y(n) = \sum_{m=-\infty}^{\infty} h(m)x(n-m) \quad \leftrightarrow \quad y(n) = h(n) * x(n), \tag{1.1}$$

kde $*$ značí přímou konvoluci. Pro konvoluci periodických posloupností s periodou M lze rovnici přepsat do tvaru diskrétní kruhové konvoluce

$$y_M(n) = \sum_{m=-\infty}^{M-1} h_M(m)x_M(n-m) \quad \leftrightarrow \quad y_M(n) = h_M(n) *_M x_M(n), \tag{1.2}$$

kde M představuje konečnou délku posloupnosti a $*_M$ značí kruhovou konvoluci.

Pro zpracování neomezeně dlouhého signálu FIR systémem s konečnou délkou impulsové odezvy M se zavádí výpočet modifikované kruhové konvoluce

$$y(n) = \sum_{m=-\infty}^{M-1} h_M(m)x_M(n-m) \qquad h_M(n) = b_i\Big|_{\mathrm{FIR},M=N+1}, \tag{1.3}$$

kde M značí délku konečné posloupnosti impulsové odezvy $h_M(n)$ a $b_0, b_1, b_2, \ldots b_N$ jsou koeficienty čitatele přenosové funkce s celkovým počtem $M = N+1$. Výstupem výpočtu kruhové diskrétní konvoluce je diskrétní konvoluce.

Optimálním řešením pro realizaci FIR systémů vysokého řádu je však realizace výpočtu diskrétní konvoluce v kmitočtové oblasti pomocí diskrétní Fourierovy transformace (DFT). DFT lze spočítat pomocí rychlé Fourierovy transformace (FFT) s výpočetní náročností lineárně narůstající s rostoucím řádem. Pro výpočet v kmitočtové oblasti je nutné nejdříve získat obraz diskrétní Fourierovy transformace vstupního signálu a kmitočtové charakteristiky.

Následující rovnice a text jsou převzaty z [6], kde rovnice

$$\boldsymbol{X}(k) = \sum_{n=0}^{N-1} x(n)\mathrm{e}^{-jk\frac{2\pi}{N}n} \tag{1.4}$$

představuje zjednodušený zápis obrazu diskrétní Fourierovy transformace definující diskrétní složky spektra. Symbol $\boldsymbol{X}$ značí obraz transformace.

Pomocí inverzní diskrétní Fourierovy transformace lze poté definovat vzor takového obrazu

$$x(n) = \frac{1}{N} \sum_{n=0}^{N-1} \boldsymbol{X}(k) \mathrm{e}^{+jk\frac{2\pi}{N}n} = \frac{1}{N} \sum_{n=0}^{N-1} \boldsymbol{X}^*(k) \mathrm{e}^{-jk\frac{2\pi}{N}n}, \tag{1.5}$$

kde $*$ nyní označuje komplexně sdružený obraz dopředné Fourierovy transformace. Zpětná (inverzní) Fourierova transformace z výše uvedené rovnice se liší pouze ve znaménku jádra transformace a váhovací konstantě $1/N$.

Obraz $\boldsymbol{H}$ kmitočtové charakteristiky lze poté vypočítat podle rovnice

$$\boldsymbol{H}(k) = \sum_{n=0}^{N-1} h(n) \mathrm{e}^{-jk\frac{2\pi}{N}n}. \tag{1.6}$$

Princip konvoluce v kmitočtové oblasti spočívá v Hadamardově součinu[1] kmitočtového spektra segmentu $\boldsymbol{X}$ a kmitočtové charakteristiky $\boldsymbol{H}$. Následující rovnice je převzata z [6] a představuje Hadamardův součin

$$\boldsymbol{Y}(k) = \boldsymbol{X}(k) \circ \boldsymbol{H}(k). \tag{1.7}$$

Výsledkem součinu je spektrum konvoluce. Inverzní transformací a následným váhováním $1/(2N)$ lze získat výsledek konvoluce y v časové oblasti.

Pomocí algoritmu rychlé Fourierovy transformace v kmitočtové oblasti lze realizovat systém zpracovávající vstupní posloupnost (signál), avšak s procesním zpožděním, které transformace přináší. V následujících podkapitolách bude vysvětlen princip zpracování signálu takovým systémem a popsán způsob potlačení procesního zpoždění.

1.1 Segmentální zpracování signálu

Princip zpracování vstupního signálu velmi dlouhé nebo nekonečné délky signálem jiným spočívá v rozdělení signálu na kratší délky - segmenty, které poté vstupují do výpočtu konvoluce. Výpočet konvoluce může být dále realizován pomocí rychlé Fourierovy transformace (FFT). Obecně jsou známé dva algoritmy vhodné pro segmentální zpracování signálu jak uvádí [2].

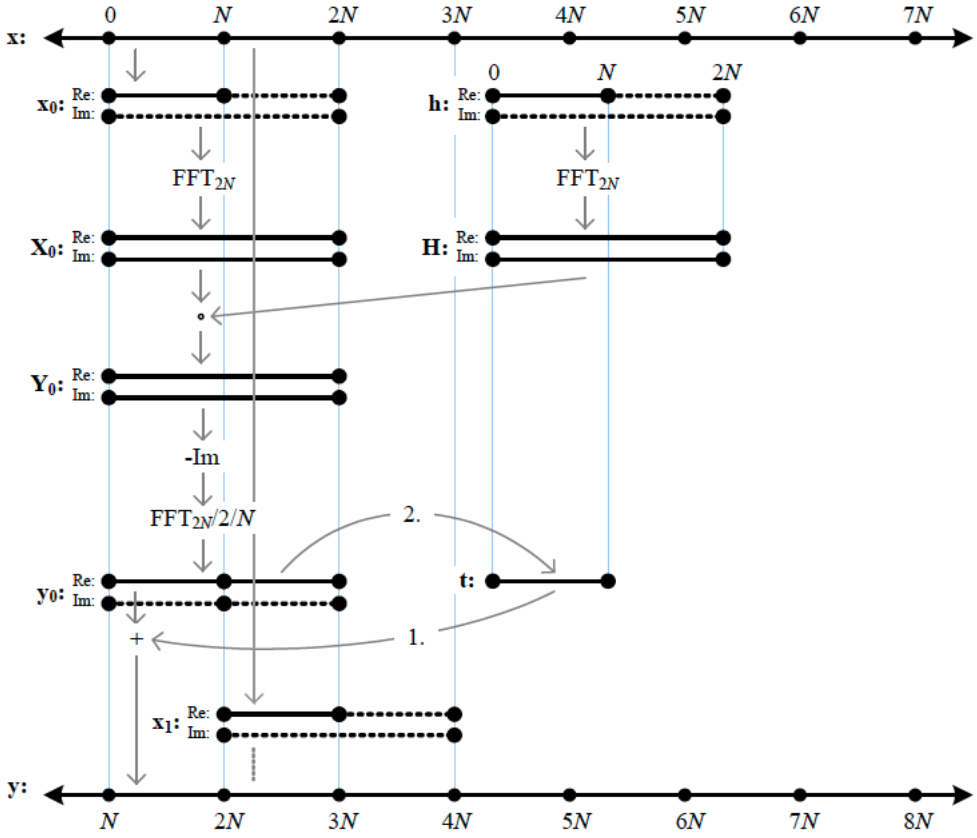
- Rychlá konvoluce s přičtením přesahu (*Overlap-Add OLA*)
- Rychlá konvoluce s uložením přesahu (*Overlap-Save OLS*)

Zpracování vstupního signálu bude demonstrováno na algoritmu jednokanálové rychlé konvoluce s přičtením přesahu (*OLA - Overlap-Add*).

[1] Jedná se násobení dvou vektorů nebo matic `A` a `B` o stejném rozměru po složkách.

1.1.1 Algoritmus rychlé konvoluce s přičtením přesahu

Následující algoritmus rychlé konvoluce s přičtením přesahu je převzat z [6] a [1]. Algoritmus je realizován pomocí dvou dvojitých zásobníků. Jeden ze zásobníků obstarává segmentaci dat vstupního signálu a druhý desegmentaci dat výstupních ve vnějším a vnitřním cyklu algoritmu.



Obr. 1.1: Realizace jednokanálové rychlé konvoluce s přičtením přesahu (OLA). Převzato z [6].

Na obrázku 1.1 vstupuje do algoritmu signál $\mathbf{x}$ o celkové délce $7N$, kde N představuje délku nejkratšího segmentu. S časovým intervalem NT_{vz} je ve vnější části algoritmu vybrán nový segment $\mathbf{x}_i$ o délce N, který vstupuje do vnitřního cyklu algoritmu.

Ve vnitřním cyklu je segment $\mathbf{x}_i$ doplněn nulami na délku $2N$. Algoritmus FFT předpokládá komplexní vstupní posloupnost, proto je segment načten do komplexního zásobníku. Při dvoukanálové realizaci je levý kanál signálu načten do reálné části komplexního zásobníku a pravý kanál do části imaginární. Při realizaci jednokanálové je imaginární část naplněna nulami.

Následně je spočítáno kmitočtové spektrum $\mathbf{X}_i$ signálu délky $2N$ pomocí FFT. Kmitočtovou charakteristiku $\mathbf{H}$ lze spočítat obdobným způsobem.

Pro výpočet obrazu výstupní posloupnosti aktuálního segmentu $\mathbf{Y}_i$ se využije Hadamardův součin z rovnice 1.7. Výsledek je dále podroben inverzní transformaci a váhování.

K první polovině výstupního segmentu $\mathbf{y}_i$ je následně přičten obsah dočasného zásobníku $\mathbf{t}$, který je v prvním kroku algoritmu naplněn nulami. Druhá polovina výstupního segmentu je poté načtena do dočasného zásobníku pro výpočet v dalším kroku algoritmu. Tímto způsobem lze získat aktuální výstupní segment $\mathbf{y}_i$, který přechází do vnějšího cyklu algoritmu a je zařazen do výstupní posloupnosti $\mathbf{y}$.

Celý proces je doprovázen procesním zpožděním $2NT_{\mathrm{vz}}$. Jedná se o čas naplnění vstupního dvojitého zásobníku a odezvu na počáteční podmínky.

1.2 Potlačení procesního zpoždění

Podle zdrojů [6] a [1] je procesní zpoždění závislé na délce impulsové odezvy. Při zpracování vstupního signálu impulsovými odezvami nebo jiným signálem o délkách např. 100 000 vzorků a více může zpoždění nabývat vysokých hodnot, které je nežádoucí například při živé reprodukci hudby.

Potlačení procesního zpoždění je založeno na efektivní segmentaci impulsové odezvy. Efektivní metoda je již detailně popsána v mé bakalářské práci [1] a v této kapitole bude metoda popsána jen stručně, jelikož není hlavním tématem této práce.

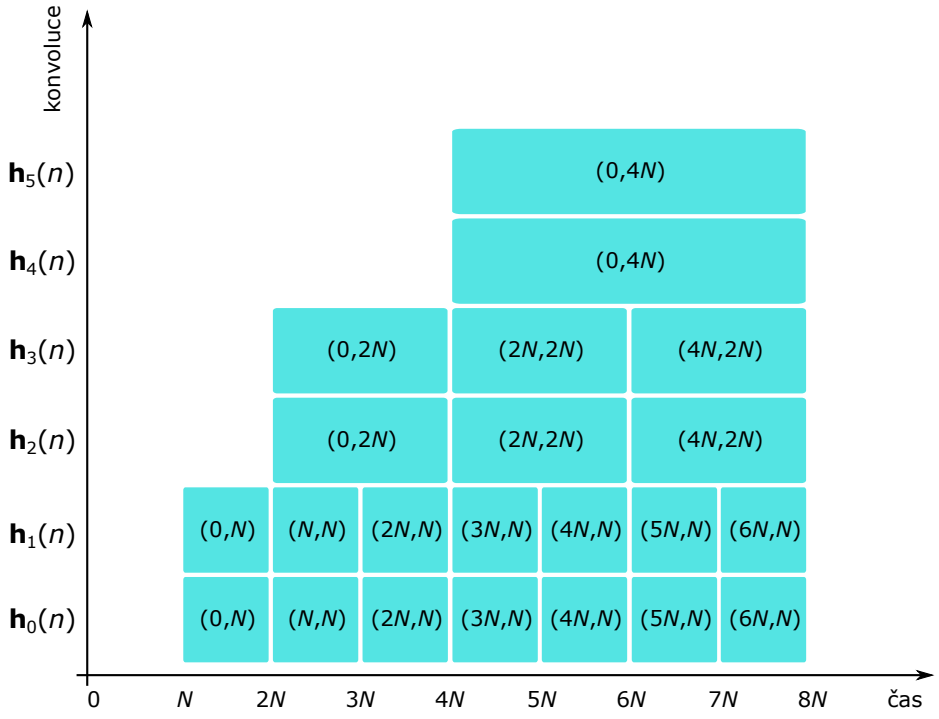
Algoritmus potlačení procesního zpoždění vychází z [7]. Potlačení spočívá v rozdělení impulsové odezvy do různých délek. Díky lineární vlastnosti konvoluce lze konvoluci každého tohoto segmentu impulsové odezvy se segmentem vstupní posloupnosti spočítat jednotlivě a poté se výsledky konvolucí sčítají a synchronizují. Doba procesního zpoždění je tak zkrácena pouze na dobu čekání na výpočet prvního segmentu délky N. Následující tabulka 1.1 použita z [7] zobrazuje vhodné rozdělení impulsové odezvy. Efektivní výpočet konvoluce pomocí algoritmu FFT očekává vstup o délkách mocniny čísla 2. S výše popsaným rozdělením je možné docílit toho,

Tab. 1.1: Vhodná segmentace impulsové odezvy pro algoritmus FFT. Použito z [7].

	N	N	$2N$	$2N$	$4N$	$4N$
$\mathbf{h}$	$\mathbf{h}_0$	$\mathbf{h}_1$	$\mathbf{h}_2$	$\mathbf{h}_3$	$\mathbf{h}_4$	$\mathbf{h}_5$

že odezva segmentu délky N začne $2N$ vzorků před začátkem impulsove odezvy dalšího segmentu. Procesor počítače má poté dostatek času pro výpočet konvoluce a nemusí čekat na načtení dat.

Rozdělení impulsové odezvy v čase a doba výpočtu konvolucí jsou zobrazeny na obrázku 1.2. Text vychází z [1].



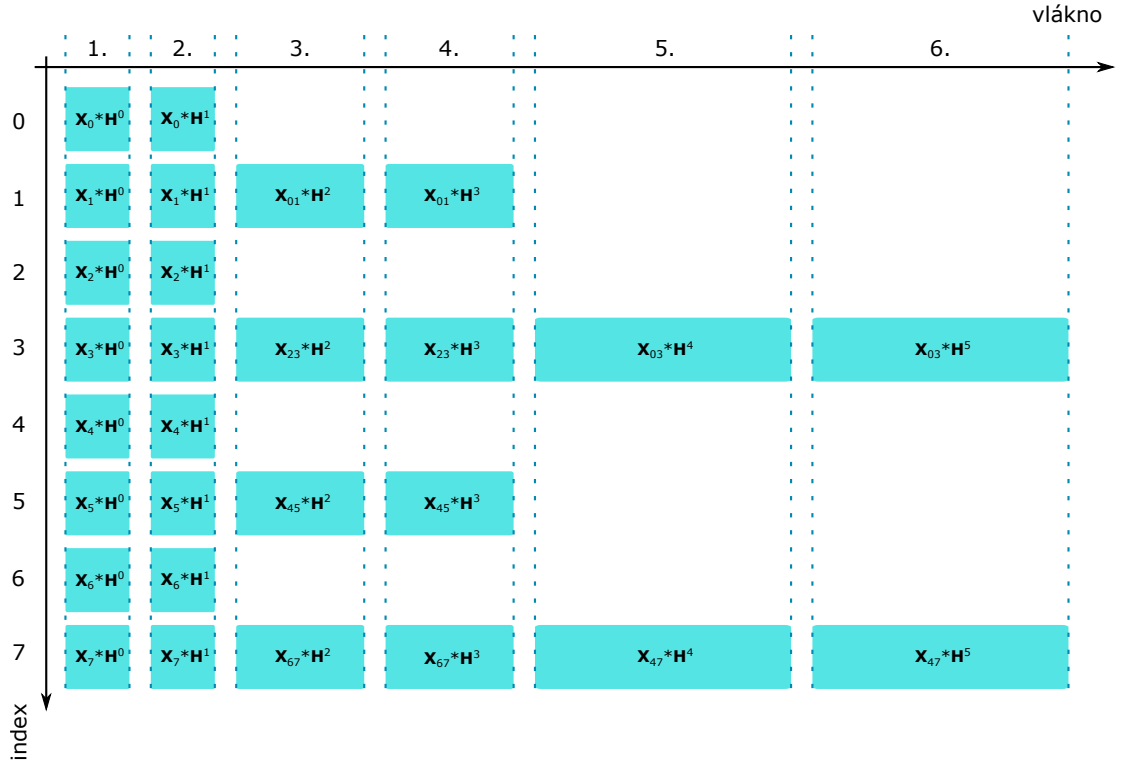
Obr. 1.2: Rozdělení impulsové odezvy a doba výpočtů konvolucí v časové oblasti. Použito z [1].

Načtení dat pro výpočet konvoluce začíná v čase 0. V čase N začíná samotný výpočet konvoluce prvního segmentu impulsové odezvy $\mathbf{h}_0$ délky N se segmentem vstupní posloupnosti délky N. Tento úsek lze označit jako první krok výpočtu. Během stejného kroku probíhá současně výpočet konvoluce segmentu $\mathbf{h}_1$ se stejným segmentem vstupní posloupnosti. Po výpočtu z prvního kroku se načte další segment impulsové odezvy, respektive vstupní posloupnosti o velikosti N pro výpočet konvoluce v druhém kroku.

Výpočet druhého kroku začína v čase $2N$. Ve stejném čase tak probíhá výpočet konvoluce vstupního segmentu délky N se segmenty impulsové odezvy $\mathbf{h}_0$, $\mathbf{h}_1$ a také výpočet konvoluce vstupního segmentu délky $2N$ (začínajícího v čase 0) se segmenty impulsové odezvy $\mathbf{h}_2$, $\mathbf{h}_3$ rovněž délky $2N$.

Obdobným způsobem probíhá načítání a výpočet konvoluce pro všechny zbývající segmenty impulsové odezvy. Uvedená segmentace impulsové odezvy, respektive výpočty konvolucí odezvy se vstupním signálem poté umožnují vypočítat segmenty menších délek během výpočtu segmentu délky větší. Pokud bude tato úloha rozdělena do více paralelních větví a poté naprogramována jako výpočet na více vláknech

(jádrech) procesoru, je možné během výpočtu segmentu délky $4N$ současně spočítat 4-krát segment délky N.



Obr. 1.3: Výpočty dílčích konvolucí jednotlivých kroků algoritmu v kmitočtové oblasti. Použito z [1].

Obrázek 1.3 je použit ze zdroje [1] a zobrazuje implementaci dílčích výpočtů konvoluce rozdělenou do více paralelních vláken. Pojem *vlákno* představuje výpočet vstupního segmentu s různými segmenty impulsové odezvy, nejedná se tedy ještě o fyzické vlákno (jádro) procesoru. Indexem je označek krok algoritmu. Ve schématu je použito nové značení pro jednotlivé výpočty konvolucí v kmitočtové oblasti a je ponecháno podle zdroje [1].

Dolní index identifikátoru značí pořadí vstupního segmentu a horní index označuje pořadí kmitočtové charakteristiky. Víceciferný dolní index označuje složení dvou a více segmentů vstupní posloupnosti. Zápis $\mathbf{X}_{23}$ odpovídá segmentu o délce $2N$ složeného z druhého a třetího segmentu vstupní posloupnosti. Tento segment vstupuje do výpočtu konvoluce se segmentem impulsové odezvy s označením $\mathbf{H}^2$. Další výpočty a značení segmentů jsou řízeny již analogicky.

2 ČASOVĚ-VARIANTNÍ SYSTÉMY S KONEČNOU IMPULSOVOU ODEZVOU

Pro popis takového systému je nutné nejdříve vymezit dva pojmy – *časově-variantní systém* a *systém s konečnou impulsovou odezvou*. Podle [6] je *časově-variantní* systém takový systém, jehož parametry (koeficienty) se s časem mění.

Pro systémy s *konečnou impulsovou odezvou* (systémy FIR) platí, že jsou definovány přenosovou funkcí, kde je jmenovatel roven jedné. Rovnice 2.1 popisuje přenosovou funkci obecného FIR systému a je převzata z [6].

$$\boldsymbol{H(z)} = \boldsymbol{P(z)} = \sum_{i=0}^{N} b_i \boldsymbol{z}^{-i} = b_0 + b_1 \boldsymbol{z}^{-1} + b_2 \boldsymbol{z}^{-2} + \cdots + b_N \boldsymbol{z}^{-N} = k \prod_{i=1}^{N} (\boldsymbol{z} - \boldsymbol{n}_i). \quad (2.1)$$

Tento systém je definován koeficienty $b_0, b_1, b_2 \ldots b_N$ polynomu $\boldsymbol{P(z)}$. Nulové body $\boldsymbol{n}_i$ jsou řešením charakteristické rovnice. Následuje rovnice 2.2 impulsové odezvy $h(n)$ definovaná koeficienty polynomu $\boldsymbol{P(z)}$. Rovnice je převzata z [6].

$$h(n) = [b_0 \quad b_1 \quad b_2 \quad \ldots \quad b_N]_M, \quad (2.2)$$

kde M označuje celkový počet koeficientů systému nebo délku jeho impulsové odezvy v počtu vzorků. FIR systém je tedy definován pouze svou impulsovou odezvou. Výše definovaným systémem lze realizovat zpracování nekonečně dlouhé vstupní posloupnosti impulsovou odezvou téměř nekonečné délky.

Doposud bylo uvažováno zpracování vstupní posloupnosti impulsovou odezvou předem známé délky. Následující text objasní zpracování vstupní posloupnosti impulsovou odezvou časově-variantní. V praxi to znamená, že lze realizovat zpracování nekonečně dlouhé vstupní posloupnosti impulsovou odezvou měnící se v čase a pokud využijeme algoritmu z předešlé kapitoly, může tato impulsová odezva nabývat téměř nekonečné délky.

2.1 Impulsová odezva časově-variantního systému

Následující text a rovnice vychází z [3]. Pokud se impulsová odezva $h(n)$ mění v čase, lze hovořit o tzv. impulsové odezvě časově-variantního systému. Zpracování vstupní posloupnosti poté probíhá následovně. Do výpočtu konvoluce nejdříve vstupuje impulsová odezva $h_\mathrm{A}(n)$, která se v čase n_T změní na impulsovou odezvu $h_\mathrm{B}(n)$.

$$y_i(n) = \begin{cases} x(n) * h_\mathrm{A}(n), & \text{pokud} \quad n < n_\mathrm{T}, \\ x(n) * h_\mathrm{B}(n), & \text{pokud} \quad n \geq n_\mathrm{T}, \end{cases} \quad (2.3)$$

kde index i značí výsledek konvoluce pro odezvu h_A a h_B.

Rovnice 2.4 a 2.5 popisují konvoluci vstupní posloupnosti se změněnou impulsovou odezvou.

$$y_\mathrm{A}(n) = \sum_{k=0}^{N-1} x(n-k)h_\mathrm{A}(k) \tag{2.4}$$

$$y_\mathrm{B}(n) = \sum_{k=0}^{N-1} x(n-k)h_\mathrm{B}(k) \tag{2.5}$$

Výše uvedené rovnice popisují změnu pouze mezi dvěma různými odezvami. Impulsová odezva se však s časem může měnit libovolně a do výpočtů konvoluce může vstoupit vždy zcela nová impulsová odezva h_i. Podle [4] se jedná o tzv. „živou“ konvoluci.

Pokud je taková změna odezvy provedena skokově, může dojít k nežádoucím přechodovým jevům, které se na poslech projevují jako nepříjemné zvukové artefakty.[1] Vznik slyšitelných přechodových jevů závisí na vlastnostech impulsové odezvy a také na tom, v jakém místě je změna provedena. Při změnách v místech kde impulsová odezva nabývá svého maxima, je vznik slyšitelných přechodových jevů daleko pravděpodobnější než při změnách v okolí nuly.

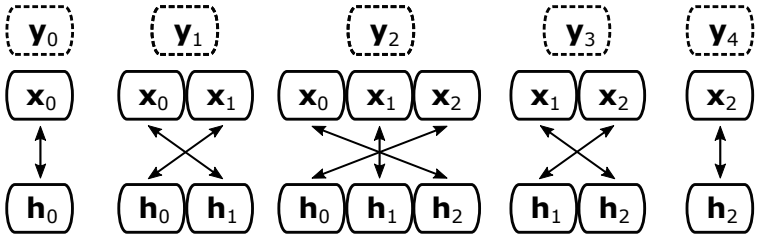
V následujících podkapitolách budou popsány obecné poznatky o impulsové odezvě časově-variantního systému a řešení pro potlačení slyšitelných přechodových jevů.

2.1.1 Minimální doba trvání vstupního segmentu a impulsové odezvy

Následující text a obrázky vychází z [3] a jsou základem k pozdějšímu návrhu potlačení přechodových jevů. Na obrázku 2.1 je zobrazeno segmentální zpracování vstupní posloupnosti $\mathbf{x}_n$ impulsovou odezvou $\mathbf{h}_n$, kde impulsová odezva nabývá délku $N = 3$. Algoritmus uvažuje dělení impulsové odezvy pouze na segmenty délky N a pro jednoduchost provedení a snadnější pochopení bude stejné dělení zachováno i pro pozdější úvahy. Dále text popisuje algoritmus pouze v časové oblasti. Kmitočtová oblast je z důvodu snadnějšího popisu neuvažována.

V prvním kroku algoritmu probíhá konvoluce mezi segmentem vstupní posloupnosti $\mathbf{x}_0$ a impulsové odezvy $\mathbf{h}_0$. Výsledkem je výstupní segment $\mathbf{y}_0$. V druhém kroku algoritmu je výsledkem výstupní segment $\mathbf{y}_1$, který je dán součtem konvolucí segmentů $\mathbf{x}_0$ s $\mathbf{h}_1$ a segmentů $\mathbf{x}_1$ s $\mathbf{h}_0$. Tímto způsobem probíhá dále segmentální zpracování vstupní posloupnosti impulsovou odezvou o celkové délce $N = 3$, při rozdělení impulsové odezvy na segmenty délky N.

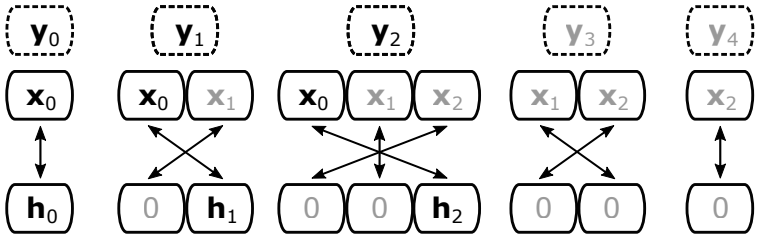
[1] Artefakty se projevují jako rázy nebo lupnutí.



Obr. 2.1: Segmentální zpracování vstupní posloupnosti impulsovou odezvou délky $3N$. Převzato z [3].

Z teorie segmentálního zpracování signálu resp. z obrázku 2.1 lze určit několik poznatků. Pro výpočet prvního výstupního segmentu $\mathbf{y}_0$ postačují segmenty $\mathbf{x}_0$ a $\mathbf{h}_0$. Výstupní segment lze spočítat ihned po načtení segmentů vstupujících do výpočtu konvoluce. Tyto segmenty lze navíc do paměti načíst paralelně, což podstatně urychluje výpočet.

Z následujícího obrázku lze určit, jak dlouho je nutné segment vstupní posloupnosti $\mathbf{x}_0$ uchovávat v paměti pro výpočet konvoluce. Pro výpočet výstupního segmentu $\mathbf{y}_3$ již segment $\mathbf{x}_0$ není potřeba, proto jej není nutné dále uchovávat v paměti.



Obr. 2.2: Minimální doba trvání vstupního segmentu a impulsové odezvy. Převzato z [3].

Vstupní segment $\mathbf{x}_n$ je tedy nutné uchovávat právě tolik kroků algoritmu, na kolik segmentů je rozdělena impulsová odezva tak, aby vstupní segment vstoupil do výpočtu se všemi vzorky impulsové odezvy. V ukázkovém příkladě je to tedy $N = 3$.

Pro segmenty impulsové odezvy platí, že v rámci výpočtů se segmentem $\mathbf{x}_0$ se segmenty impulsové odezvy $\mathbf{h}_n$ mění s každým krokem.

2.2 Principy časově-variantního konvolučního procesoru

Na základě předešlé teorie lze implementovat univerzální, časově-variantní konvoluční procesor. Jedná se tedy o systém zpracovávající nekonečně dlouhou vstupní

posloupnost impulsovou odezvou téměř nekonečné délky, která může navíc měnit své koeficienty v čase.

Používaný pojem impulsová odezva může představovat jednak odezvu systému na jednotkový impuls tak, jak uvádí [8], ale také vstupní posloupnost, která reprezentuje například zvukový signál. Zdroj [3] popisuje zpracování nahrávky barokní hudby pomocí krátkých řečových nahrávek. Jedná se tedy o určitý druh modulace zvukové nahrávky jinou zvukovou nahrávkou pomocí konvoluce.

Zdroje [3] a [4] dále uvádí způsob segmentace, respektive změny impulsové odezvy (zvukového signálu) na základě transientní analýzy. Odezva se tedy může měnit například v určité rytmické formě apod.

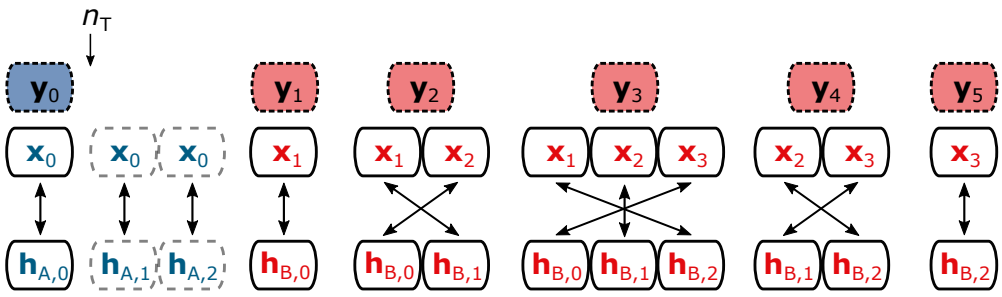
Pro funkční procesor je však zcela zásadní potlačení přechodových jevů při změnách impulsové odezvy. Spojením teorie potlačení procesního zpoždění a eliminace přechodových jevů lze implementovat funkční, výše popsaný systém pro využití v hudebních aplikacích.

2.3 Principy plynulého přechodu mezi odezvami

Pro popis bude v této části uvažována změna mezi impulsovou odezvou $\mathbf{h}_\mathrm{A}$ a $\mathbf{h}_\mathrm{B}$. Obě odezvy jsou délky $3N$. Index ve značení $\mathbf{h}_{\mathrm{A},i}$ představuje pořadí N-dlouhého segmentu impulsové odezvy. Text vychází ze [3].

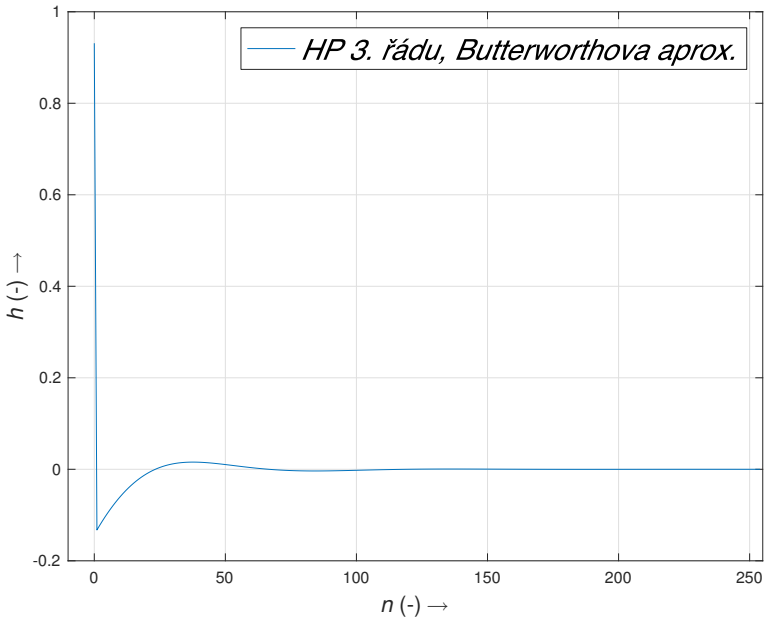
Nejdříve bude přiblíženo kritické místo přechodu mezi impulsovými odezvami. Pokud bude ponecháno značení z kapitoly 2.1, je kritickým místem časový okamžik změny impulsové odezvy n_T. Obrázek 2.3 zobrazuje vznik přechodových jevů při změně odezvy v okamžiku n_T.

V tomto místě dochází k přerušení výpočtu konvoluce zbývajících segmentů impulsové odezvy $\mathbf{h}_{\mathrm{A},1}$ a $\mathbf{h}_{\mathrm{A},2}$ se segmentem $\mathbf{x}_0$. Tyto segmenty se tedy nijak neprojeví na výstupním segmentu $\mathbf{y}_\mathrm{n}$ a dochází k jejich „zahození“. V závislosti na průběhu impulsové odezvy tak může nastat situace, že vzorky výstupního segmentu končí výrazně vzdálené od nulových hodnot.



Obr. 2.3: Neplynulá změna impulsové odezvy. Vychází z [3].

Ve stejném čase n_T zároveň dochází k načtení nového segmentu, respektive vzorků impulsové odezvy $\mathbf{h}_\mathrm{B}$, které vstupují do výpočtu konvoluce se vstupní posloupností. Pokud je impulsová odezva $\mathbf{h}_\mathrm{B}$ charakteristická výrazným zákmitem v časové oblasti např. filtr typu horní propust, na přelomu výstupního segmentu $\mathbf{y}_0$, ovlivněného odezvou $\mathbf{h}_\mathrm{A}$ a segmentu $\mathbf{y}_1$, ovlivněného odezvou $\mathbf{h}_\mathrm{B}$ vzniká výrazný, nekontinuální průběh vzorků způsobující slyšitelný přechodový jev.



Obr. 2.4: Horní propust 3. řádu s Butterworthovou aproximací.

Pro plynulý přechod mezi odezvami a k potlačení přechodových dějů lze využít několik způsobů. V následujících podkapitolách budou představeny dva způsoby.

2.3.1 Přechod mezi odezvami v kmitočtové oblasti

První způsob přechodu a jeho podrobný popis jsou uvedeny v [2]. V této části bude uveden pouze popis zjednodušený. Přechod mezi odezvami v kmitočtové oblasti vychází z přechodu v oblasti časové. Hlavní výhodou realizace v kmitočtové oblasti je ušetřený výpočet zpětné Fourierovy transformace.

Princip spočívá ve vynásobení dílčích výstupních segmentů lineární nebo harmonickou funkcí (oknem), která poté determinuje obálku signálu výstupních segmentů. Následující rovnice představují obálku lineární a harmonické funkce. Písmeno N představuje délku nejkratšího segmentu. Převzato z [2].

$$\text{lineární} \qquad f_{out}(n) = 1 - \frac{N}{n} \qquad f_{in}(n) = \frac{N}{n}. \tag{2.6}$$

$$\cos^2 \qquad f_{out}(n) = cos^2\left(\frac{\pi n}{2N}\right) \qquad f_{in}(n) = sin^2\left(\frac{\pi n}{2N}\right). \tag{2.7}$$

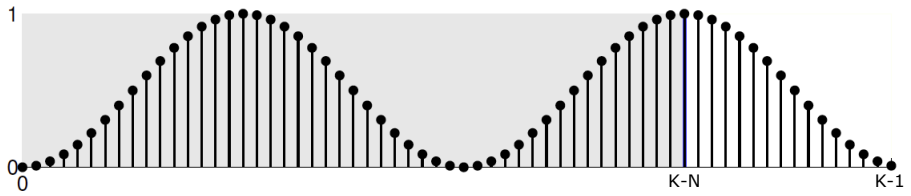
Jakmile je známa obálka lineární, respektive harmonické funkce pro nástupní (f_{in}) a sestupnou (f_{out}) hranu, je poté spočítán obraz (spektrum) obálky $\boldsymbol{F}_{in}$, $\boldsymbol{F}_{out}$. Dále dochází k vynásobení spekter výstupních segmentů $\boldsymbol{Y}_0$, $\boldsymbol{Y}_1$ se spektry obálek $\boldsymbol{F}_{in}$, $\boldsymbol{F}_{out}$ podle rovnice 2.9, kde $*$ značí Hadamardův součin.

$$y(n) = y_0(n) \cdot f_{out}(n) + y_1(n) \cdot f_{in}(n). \tag{2.8}$$

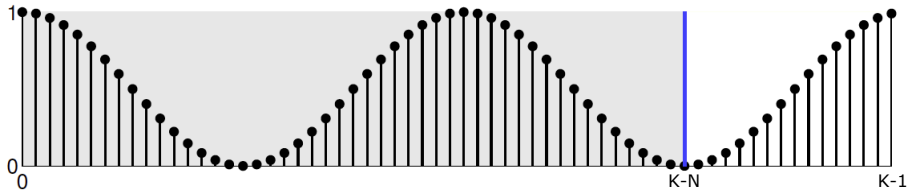
$$\boldsymbol{Y}(n) = \boldsymbol{Y}_0(k) * \boldsymbol{F}_{out}(k) + \boldsymbol{Y}_1(k) * \boldsymbol{F}_{in}(k). \tag{2.9}$$

Pokud bude předcházející rovnice aplikována na přechod mezi dvěma různými odezvami, pak bude aktuální výstupní segment $\boldsymbol{Y}_n$ délky N dán součtem dílčích, superponovaných segmentů první $\boldsymbol{Y}_0$ a druhé $\boldsymbol{Y}_1$ odezvy.

Na obrázcích 2.5, 2.6 lze vidět způsob získání obálky harmonického průběhu tak, jak uvádí [2]. Obálka harmonického signálu je prodloužena na dvě periody. Písmenem K je označena celková délka obálky. Část průběhu signálu do bodu $K-N$ je zahozena (vyznačeno šedě). Pro plynulý přechod nástupné a sestupné hrany je použita pouze poslední, N-dlouhá část harmonického signálu.



Obr. 2.5: Implementace plynulého přechodu - Fade out. Převzato z [2].



Obr. 2.6: Implementace plynulého přechodu - Fade in. Převzato z [2].

2.3.2 Přechod mezi odezvami v časové oblasti

Další způsob plynulého přechodu mezi odezvami je modifikací předešlé podkapitoly. Princip je založen na zpracování výstupních segmentů váhovacím oknem v časové oblasti. Poznatky o váhovacích oknech vychází z [6]. Mějme nekauzální neperiodické (symetrické) Hannovo okno se sudou délkou popsané následujícími rovnicemi. Převzato z [6]. Pro Hannovo okno platí

$$w_{\mathrm{han}}(n) = 0,5\left(1 - \cos\left(\frac{2\pi n}{N-1}\right)\right), \quad n = 0, 1, \ldots, N-1, \tag{2.10}$$

kde N značí délku okna. Pro pozdější využití lze tento zápis rozdělit pro první a druhou polovinu Hannova okna následujícím způsobem

$$w_{\mathrm{han1}}(n) = 0,5\left(1 - \cos\left(\frac{2\pi n}{N-1}\right)\right), \quad n = 0, 1, \ldots, \frac{N-1}{2}, \tag{2.11}$$

$$w_{\mathrm{han2}}(n) = 0,5\left(1 - \cos\left(\frac{2\pi n}{N-1}\right)\right), \quad n = \frac{N-1}{2}, \ldots, N-1, \tag{2.12}$$

kde $w_{han1}(n)$ značí první polovinu a $w_{han2}(n)$ druhou polovinu Hannova okna.

Obdobným způsobem a značením je definováno také Barlettovo (trojúhelníkové) okno. Použito z [6].

$$w_{\mathrm{Bar1}}(n) = \frac{2n}{N-1}, \quad n = 0, 1, \ldots, \frac{N-1}{2}, \tag{2.13}$$

$$w_{\mathrm{Bar2}}(n) = 2 - \frac{2n}{N-1}, \quad n = \frac{N-1}{2}, \ldots, N-1. \tag{2.14}$$

Hannovo a Barlettovo okno je graficky znázorněno na obrázku 2.7.

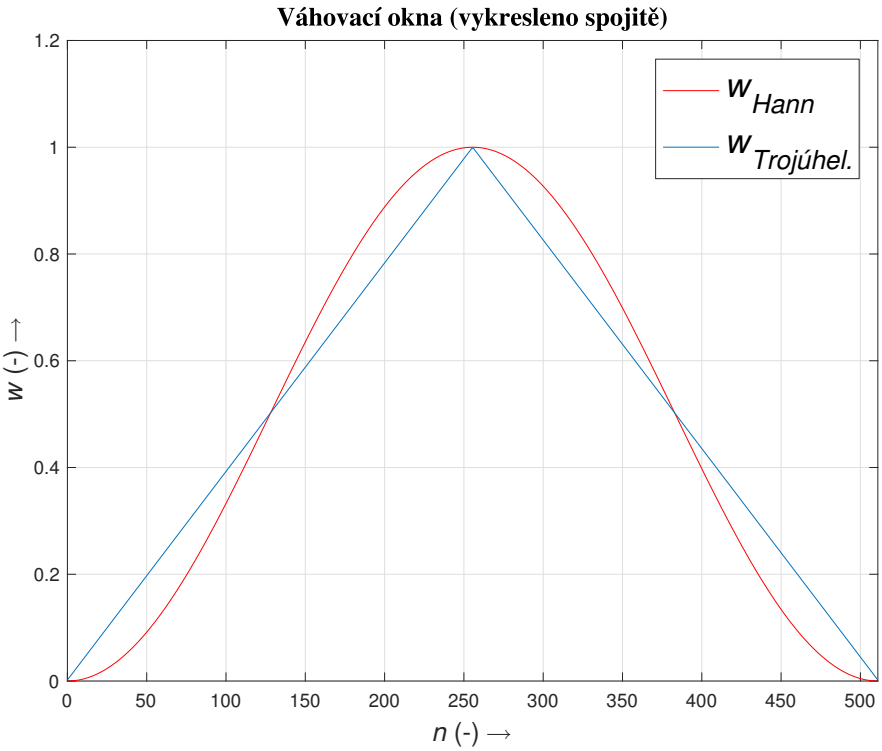
Na základě definice se vytvoří posloupnost reprezentující dané okno. V algoritmu eliminujícím přechodové jevy se poté využívá druhá polovina symetrického okna pro vytvoření plynule sestupující posloupnosti odezvy $\mathbf{h}_{\mathrm{A}}$ a první polovina okna pro vytvoření plynule narůstající posloupnosti odezvy $\mathbf{h}_{\mathrm{B}}$. Obě váhované posloupnosti se poté v rámci jednoho zpracovaného přechodového segmentu v časové oblasti vnějšího cyklu algoritmu překrývají.

S využitím výše uvedených rovnic, rovnic 2.3, 2.4 a 2.5 a jejich zápisu lze zapsat funkční přechod následovně.

$$y_i(n) = \begin{cases} y_{\mathrm{A}}(n), & \text{pro} \quad n < n_{\mathrm{T}}, \\ y_{\mathrm{A}}(n) w_{\mathrm{han2}}(n) + y_{\mathrm{B}}(n) w_{\mathrm{han1}}(n), & \text{pro} \quad n_{\mathrm{T}} \leq n < n_{\mathrm{T}} + N, \\ y_{\mathrm{B}}(n), & \text{pro} \quad n \geq n_{\mathrm{T}} + N, \end{cases} \tag{2.15}$$

kde $y_i(n)$ představuje aktuální vzorek výstupu po konvoluci v čase před změnou, v přechodovém segmentu po změně a po přechodovém segmentu.

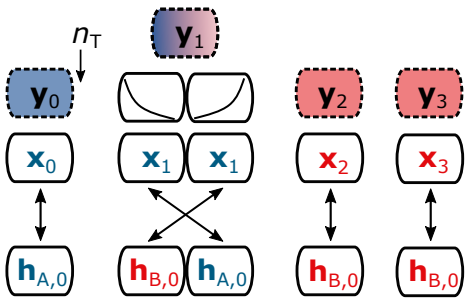
Nutno zmínit, že pro funkční plynulý přechod je nezbytné, aby se délka poloviny váhovacího okna rovnala délce zpracovávaného segmentu. Pokud je tedy délka nejkratšího zpracovávaného segmentu rovna N, musí být váhovací okno zadefinováno pro délku $2N$.



Obr. 2.7: Barlettovo (trojúhelníkové) a Hannovo okno.

Na obrázku 2.8 je zobrazen princip funkčního algoritmu potlačující přechodový jev. V čase n_T dochází ke změně impulsové odezvy $\mathbf{h}_\mathrm{A}$ na $\mathbf{h}_\mathrm{B}$. Změna je provedena postupně a může k ní dojít pouze v k-násobku segmentu délky N podle podmínky

$$n_\mathrm{T} = k * N \quad \text{, kde} \quad k = 0, 1, 2 \ldots . \tag{2.16}$$

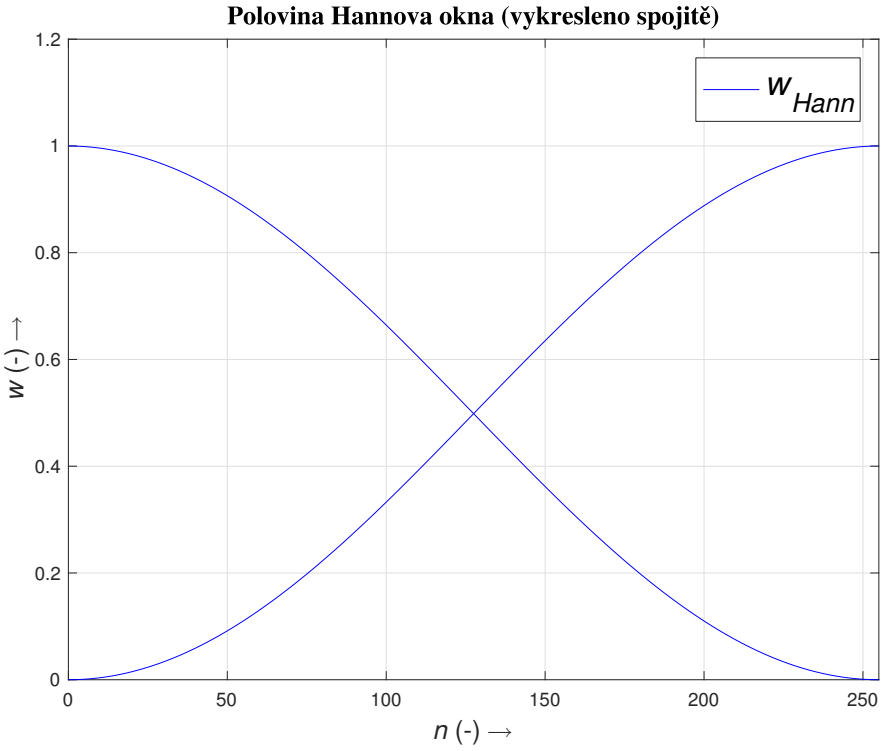


Obr. 2.8: Realizace plynulého přechodu mezi odezvami.

Pro jednoduchost je změna demonstrována pouze na odezvách délky N. V rámci jednoho výstupního tzv. přechodového segmentu $\mathbf{y}_1$ délky N se nejdříve spočítá konvoluce mezi vstupním segmentem $\mathbf{x}_1$ a první impulsovou odezvou $\mathbf{h}_\mathrm{A}$ a zároveň konvoluce mezi tímtež vstupním segmentem s impulsovou odezvou $\mathbf{h}_\mathrm{B}$. Oba výsledky konvolucí jsou dále váhovány příslušnou polovinou Hannova okna. Výstupní přechodový segment $\mathbf{y}_1$ je tak dán váhovaným výsledkem konvolucí segmentů $\mathbf{x}_1$ s $\mathbf{h}_{\mathrm{h},0}$ plus $\mathbf{x}_1$ s $\mathbf{h}_{\mathrm{B},0}$. V přechodovém segmentu tak dochází k dvojnásobnému výpočtu konvoluce.

V případě dlouhých impulsových odezev se dílčí segmenty impulsové odezvy $\mathbf{h}_\mathrm{A}$ po změně neuvažují, protože nevstupují dále do výpočtu konvoluce. Při implementaci je tedy nutné vyprázdnit zásobníky obsahující vzorky odezvy $\mathbf{h}_\mathrm{A}$ před výpočtem konvoluce $\mathbf{x}_1$ s $\mathbf{h}_\mathrm{B}$.

Hannovo okno ve srovnání s oknem Barlettovým vytváří jemnější přechod. Pro plynulý přechod jsou však vhodná obě okna. Obrazek 2.9 představuje překryv polovin Hannova okna.



Obr. 2.9: Vytvoření překryvu pomocí Hannova okna - crossfade.

3 IMPLEMENTACE VÍCEVLÁKNOVÝCH ÚLOH V C++

V této kapitole bude přiblížena problematika vícevláknového programování v jazyce C++. Nejdříve bude vysvětlena podstata vícevláknového programování a nastíněna situace, při které je vhodné vícevláknové programování využít. Následující podkapitoly představí způsob vytvoření výpočetního vlákna v jazyce C++, správný přístup vlákna do paměti, způsoby synchronizace vláken a možné problémy při práci s vlákny.

Následující text vychází z [13]. Implementace vícevláknového programování má smysl na vícejádrových výpočetních jednotkách - CPU[1]. U procesoru se rozlišuje mezi fyzickým jádrem a virtuálním jádrem. O virtuálním jádru hovoříme tehdy, je-li v procesoru obsažena technologie *Hyper-threading* [2] Nápříklad dnešní běžné procesory se čtyřmi fyzickými jádry a s technologií *Hyper-threading* tak disponují až osmi jádry.

Vícevláknová implementace je obvykle závislá na použité platformě. Standard C++11 však přináší mimo jiné také standardní knihovny pro podporu vícevláknového programování, které se tak stává multiplatformním. Dále uvažované implementace budou tedy založeny právě na standardizovaných knihovnách.

Technologický pokrok dnes nabízí také využití výpočetního výkonu výkonných grafických karet pro zpracování náročných aplikací. Pro implementace na GPU[3] však zatím neexistují standardizované knihovny jazyka C++.

3.1 Paralelní procesy

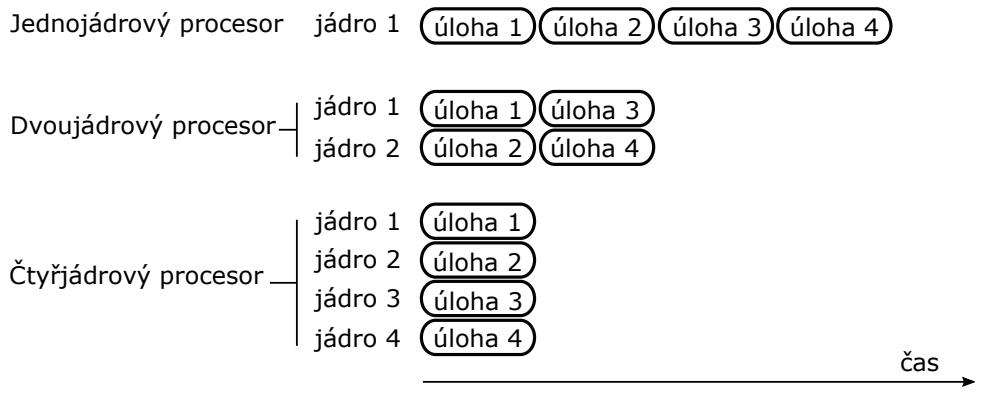
Některé výpočetní úlohy lze značně urychlit, pokud budou implementovány ve více paralelních vláknech. Pro jiné úlohy je vícevláknová implementace nutností např. 1.2. Jen tak se dá zaručit efektivní funkcionalita, rychlá odezva, včasně ukončený výpočet apod. Pokud se však jedná o jednoduchou úlohu, je lepší vícevláknovou implementaci neuvažovat z důvodu šetření výkonu, který je potřebný pro vytvoření a sychnronizaci vláken.

Na následujícím obrázku lze demonstrovat rozdíl mezi sekvenčním a paralelním zpracováním úloh podle dostupných jader procesoru. Pokud procesor obsahuje více jader, je zřejmé, že je schopen zpracování úloh dokončit rychleji než jednojádrový procesor. Jednotlivé úlohy můžou například představovat výpočet náročné operace

[1] Central Processing Unit - centrální procesorová (výpočetní) jednotka

[2] Technologie a termín pochází od výrobce Intel.

[3] Graphics Processing Unit - grafická výpočetní jednotka



Obr. 3.1: Princip vícevláknového zpracování úloh. Vychází z [13].

nebo paralelní chod GUI a audio procesoru zásuvného modulu.

3.1.1 Přístupy do paměti

Text čerpá z [12]. S vícevláknovým návrhem algorimu souvisí dva typy přístupu vláken do paměti. Přístupy lze rozdělit podle toho, na jaká data vlákna přistupují.

- **Nesdílený paměťový prostor** – Jednotlivá vlákna nesdílí společný paměťový prostor (místo v paměti) a při svém běhu nezasahují do sousedních vláken. Nedochází tedy k situaci, kdy ve stejný okamžik dvě nebo více vláken potřebují přístup ke stejné paměti. Příkladem vícevláknové úlohy s nesdíleným paměťovým prostorem může být paralelní načtení několika souborů více vlákny, kde načtení jednotlivého souboru obstarává právě jedno vlákno a soubor ukládá do specifické proměnné pro dané vlákno.
- **Sdílený paměťový prostor** – Jednotlivá vlákna sdílí společný paměťový prostor a vlákna při svém běhu zasahují do sousedních vláken. Nastává situace, kdy více vláken vyžaduje přístup do stejného paměťového prostoru ve stejný okamžik. Zdroj [13] popisuje tento jev pojmem *race condition*. Pořadí přístupu vláken obstarává operační systém a při opakovaném běhu programu se pořadí náhodně mění. Tento jev se nazývá *undefined behavior* neboli nedefinované chování. Pro funkční úlohu je tedy nutné zajistit správnou synchronizaci mezi vlákny. Jednoduchým příkladem pro úlohu se sdílenou pamětí bez synchronizace vláken může být volání funkce z více vláken, ve které se inkrementuje proměnná.

Přístup do paměťového prostoru lze dále rozlišovat podle toho, bude-li se z paměti pouze číst nebo se bude do paměti zapisovat. Při čtení nedochází k výrazným komplikacím. Při zápisu však nastává výše zmíněná situace a bez synchronizace nelze kontrolovat přístup vláken do paměti.

3.1.2 Vytvoření vlákna

Nasledující text vychází z [13]. Pro vytvoření vlákna je nutné vložit hlavičkový soubor `<thread>` ze standardní C++ knihovny. Dále je nutné vytvořit instanci typu `std::thread` a do konstruktoru třídy zadat potřebné parametry. Konstruktor třídy `std::thread` lze inicializovat libovolným počtem parametrů. Následující výpis představuje vytvoření dvou vláken a volání globální funkce s parametry.

Výpis kódu 3.1: Příklad vytvoření vlákna a volání funkce s parametry

```
#include <iostream>
#include <thread>

void pocitadlo(int id, int pocetIteraci)
{
    for (int i = 0; i < pocetIteraci; ++i)
    {
      std::cout << "pocitadlo " << id << "hodnota: " << i << std::endl;
    }
}

int main()
{
    std::thread vlakno1(pocitadlo, 1, 10);
    std::thread vlakno2(pocitadlo, 2, 15);
    vlakno1.join();
    vlakno2.join();
return 0;
}
```

Vlákna je nutné po spuštění připojik k hlavnímu vláknu (main) pomocí funkce `join()`. Tím je zaručeno, že hlavní vlákno nebude ukončeno dříve než postranní vlákna.

Do konstruktoru vlákna lze také předat členskou funkci s parametry. Následující řádek zobrazuje příklad volání členské funkce.

```
std::thread vlakno1(&Trida::pocitadlo, 1, 10);
```

Zde opět platí fakt, že pokud více vláken zasahuje na stejná data dané instance, je nutné přístup vláken ošetřit patřičnou synchronizací.

3.2 Obecné principy synchronizace

Způsobů synchronizace mezi více vlákny je několik, ovšem zde budou představeny ty nejčastější. Zvolený způsob synchronizace se vždy odvíjí od složitosti a stavby řešené úlohy. Text vychází z [13].

3.2.1 Atomický přístup do paměti

Chceme-li současně inkrementovat či dekrementovat proměnnou např. typu `int`, potřebuje procesor pro dokončení úlohy vykonat několik instrukcí jako je načtení proměnné do registru, přičtení čísla a uložení výsledku zpět do registru či paměti. Pokud má být inkrementace prováděna více vlákny, nelze zaručit, které vlákno bude mít přístup k proměné nejdříve. Nastává již zmiňovaný *race condition*. Díky standardu C++11 lze proměnnou zadefinovat jako typ `std::atomic`, u níž je zaručen atomický přistup. To znamená, že během instrukce procesoru je zajištěna synchronizace a vlákna bezpečně (thread-safe) přistupují k datům.

Standardní knihovna však podporuje jen některé jednoduché typy, které lze zadefinovat jako `std::atomic`. Chybí zde podpora např. pro typ `float`, `double` a další složitější struktury.

3.2.2 Mutex

Dalším hojně využívaným způsobem synchronizace je použití knihovny `std::mutex`, která opět přichází s C++11 standardem. Mutex je zkratka pro termín *mutual exclusion*. Zdroj [13] definuje použití mutexu jako explicitní synchronizaci.

Pro použití mutexu je nutné přiložit hlavičkový soubor `<mutex>`. Princip spočívá ve vytvoření mutexu (zámku), který se uzamče před vykonávanou operací a odemče se až po dokončení operace. Tedy až ve chvíli, kdy došlo k zápisu dat do paměti. Každé vlákno před spuštěním operace uzamče mutex. V tu chvíli má unikátní právo přistupovat k datům a ostatní vlákna v danou chvíli nemůžou s „uzamčenými“ daty pracovat, dokud nebude zámek uvolněn.

Standard rozlišuje třídy mutex s časovou závislostí a bez časové závislosti. Vlákna, která používají zámky s časovou závislostí mají možnost získat unikátní přístup k datům jen po zadaný čas nebo se můžou pokusit o odemčení zámku v určitém čase. Ve srovnání s tím uzamčení mutexu bez časové závislosti trvá až do jeho explicitního odemčení.

Při práci se zámky mutex můžou nastat následující problémy.

- **Neodemčení mutexu** – tato situace nastane při neodemčení uzamčeného mutexu. Další vlákna tak nedostanou možnost přistoupit k uzamčeným datům, což vede k selhání programu. Problém lze ošetřit správným odemykáním

mutexu nebo použitím třídy `std::lock_guard`. Instance této třídy se vytvoří jen jednou a uzamče požadovanou operaci. Při zániku instance se automaticky zámek otevře.

- **Deadlocks** – při špatném návrhu algoritmu s více zámky může nastat situace, kdy první zámek čeká na otevření druhého zámku, který však čeká na otevření prvního zámku. Běh programu se tedy zasekne a nemůže dále pokračovat. Řešením takové situace je správné navržení algoritmu a zámků.

3.2.3 Semafor

Synchronizace pomocí semaforů je obdobná synchronizaci pomocí mutexu. Na mutex lze pohlížet jako na zámek, který se nachází ve stavu *zamčeno* nebo *odemčeno*. Semafor ve srovnání s mutexem může nabývat vyšších hodnot. Jedná se tedy o proměnnou, která může být inkrementována nebo dekrementována a podle aktuální hodnoty semaforu logika algoritmu provede synchronizaci. Semafor může být definován například jako atomická proměnná.

3.2.4 Condition Variable

Dalším synchronizačním principem je tzv. *Condition Variable*. Jedná se o knihovnu `std::condition_variable`. Princip je založený na blokování vláken do doby, dokud nebude změněna společná proměnná. Vlákno, které ukončilo úlohu oznámí informaci o ukončení ostatním vláknům pomocí funkce `notify_all()`.

3.3 Thread Pools

Jiný pohled na zpracování úloh ve více vláknech přináší princip známý jako *Thread Pool*. Na začátku programu je vytvořen požadovaný počet vláken, kterým se podle potřeby přerozdělují úlohy ke zpracování. Hlavní výhodou tohoto principu je, že během programu se v kritických místech nevytváří nová vlákna a tím se nezpomaluje běh programu. Stále zde však platí nutnost synchronizace úloh.

Tento princip bude využit v praktické části práce, kde bude princip více přiblížen a ukázán na konkrétním příkladu.

4 REALIZACE ČASOVĚ-VARIANTNÍHO KONVOLUČNÍHO PROCESORU V MATLABU

V následující kapitole bude představena realizace konvolučního procesoru v prostředí *Matlab*®. Konkrétně byla použita verze R2016a (9.0.0.341360). Realizace představuje algoritmus, který zpracovává dvoukanálový signál, řeší eliminaci přechodových jevů a potlačuje procesní zpoždění. Verze algoritmu je přichystaná k paralelním výpočtovým operacím pro aplikace v reálném čase.

Konvoluční procesor je zde implementován jako časově-variantní filtrace harmonického signálu. Bude využito segmentální zpracování signálu tak, jak je uvedeno v teoretické části s tím, že délka nejkratšího segmentu je 256 vzorků. Během zpracování se v čase postupně změní filtrace z dolní propusti na horní. V místě přechodu vzniká slyšitelný přechodový jev, který je potlačen způsobem popsaným v teoretické části 2.3.2. V příloze je dodán skript zpracovávající dvoukanálový signál s řešením přechodových jevů `TVConvHann.m` a skript bez řešení `TVConv.m`. Dále lze v příloze nalézt funkci `fzerofft2ch.m` volanou z obou skriptů. V kapitole 6.1 budou porovnány výsledky realizace obou algoritmů.

Pro proměnné a funkce bude použit text psaný strojopisem. Znak `*` značí násobení a `.*` značí Hadamardův součin. Stěžejní části kódu budou vypsány v rámečku. Z technických důvodů budou komentáře k řádkům programu psané bez diakritiky.

Seznam důležitých proměnných:

- `bufferSize` délka nejkratšího segmentu
- `hd` impulsová odezva dolní propusti
- `Hd` obraz impulsové odezvy dolní propusti
- `hh` impulsová odezva horní propusti
- `Hh` obraz impulsové odezvy horní propusti
- `fade` Hannovo okno pro realizaci plynulého přechodu
- `tL, tL2` dočasné zásobníky pro kanál 1
- `tR, tR2` dočasné zásobníky pro kanál 2
- `PocetVlaken` počet vláken hlavní smyčky
- `x_max` vstupní dvoukanálový komplexní zásobník
- `y_maxL, y_maxR` výstupní zásobníky maximální délky
- `X_n` obraz vstupního segmentu
- `Y_n` výsledek po Hadamardově součinu
- `y_n` aktuální výstupní segment v časové oblasti
- `yL_n` aktuální výstupní segment prvního kanálu
- `yR_n` aktuální výstupní segment druhého kanálu
- `yL,xR` výstupní posloupnosti

4.1 Vstupní proměnné algoritmu

Na začátku skriptu jsou zadefinované důležité proměnné využívané v algoritmu. Vzorkovací kmitočet 44100 Hz je uložen do proměnné `Fvz`. Délka nejkratšího segmentu je uložena do proměnné `bufferSize`.

4.1.1 Vstupní signál

Jako vstupní signál je použit harmonický signál složen ze signálů o kmitočtech 500 Hz a 1500 Hz. Tento signál je převeden do stereofoního signálu o délce 4 sekund. Vstupní data jsou uložena do proměnné `xL` respektive `xR`. Následně je nutné upravit délku vstupního signálu na délku násobku nejkratšího segmentu `bufferSize` pro funkční segmentální zpracování. Délka vstupního signálu v násobcích nejkratšího segmentu je uložena v `inputSizeSeg` a je využita k úpravě vstupního signálu na novou vhodnou délku.

Výpis kódu 4.1: Zarovnání vstupního signálu na vhodnou délku

```
inputSize = length(xL); % delka signalu
inputSizeSeg = fix(inputSize/N) + 1; % pocet zpracovanych
   segmentu delky N
xL = [xL, zeros(1,(inputSizeSeg*N-inputSize))]; % kanal 1
   upraveny na spravnou delku
xR = [xR, zeros(1,(inputSizeSeg*N-inputSize))]; % kanal 2
   upraveny na spravnou delku
```

4.1.2 Příprava impulsové odezvy

Pro algoritmus jsou připraveny impulsové odezvy dvě. Odezva dolní a horní propusti třetího řádu s Butterworthovou aproximací. Mezní kmitočet dolní a horní propusti je nastaven na 500 Hz. Koeficienty filtru jsou vytvořeny pomocí funkce `butter`. Pomocí funkce `impz` je spočítána odezva filtru podle zadaných koeficientů. Proměnná `hd` a `hh` slouží k uložení hodnot impulsové odezvy dolní, respektive horní propusti. Impulsová odezva horní propusti zadefinovaná tímto způsobem zaručuje nekontinuální průběh při změnách odezev a vznik slyšitelných přechodových jevů.

4.2 Hlavní smyčka zpracování signálu

Před zpracováním hlavní smyčkou algoritmu je nutné připravit několik dalších proměnných. Jedná se o vynulování dočasných zásobníků `tL, tL2` a `tR, tR2`. Dále je potřeba vypočítat kmitočtové spektrum dolní a horní propusti pomocí funkce `fft`. Dvoukanálový vstupní signál je nutné převést na komplexní posloupnost, kterou funkce `fft` očekává na svém vstupu. Podle délky impulsové odezvy je dynamicky aktualizována proměnná `PocetVlaken`, která reprezentuje počet výpočetních vláken a slouží k vyhodnocování podmíněných cyklů v hlavní smyčce.

V hlavní smyčce algoritmu se opakovaně volá funkce `fzerofft2ch` 4.3, která v návratových hodnotách vrací výsledek po konvoluci aktualního vstupního segmentu s příslušným segmentem impulsové odezvy a obsah dočasných zásobníků. Funkce `fzerofft2ch` představuje realizaci vnitřní části algoritmu popsané v 1.1.1. Podle velikosti aktuálně zpracovávaného segmentu se volá výše uvedená funkce s příslušnými vstupními parametry.

4.2.1 Realizace segmentálního zpracování

Realizace segmentálního zpracování je provedena následovně. Ve výpisu 4.2 je smyčka prováděna s krokem od 1 až do počtu segmentů vstupního signálu `inputSizeSeg`. S každým novým krokem se v proměnných `x_maxL` a `x_maxR` nachází vždy nové vzorky vstupního signálu. Tímto způsobem se zaručí zpracování všech vstupních vzorků po segmentech délky `bufferSize`. Na této úrovni se dá také hovořit o realizaci vnější části algoritmu 1.1.1.

Zpracování výstupní posloupnosti představuje několik posledních řádků v hlavní smyčce algoritmu. S každým novým krokem je do výstupní posloupnosti `yL`, `yR` uložen aktuální výsledek z výstupních zásobníků `yL_max`, `yR_max`. Pozice pro uložení se s každým krokem posune právě o délku nejkratšího segmentu. S každým krokem je taky nezbytné posunout obsah vstupních a výstupních zásobníku. Výstupní zásobníky je potřeba dále vhodně vynulovat tak, aby došlo ke správnému posunu obsahu zásobníků.

Níže zobrazený výpis 4.2 kódu představuje realizaci časově variantního konvolučního procesoru se změnou impulsové odezvy a bez potlačení přechodového jevu.

Výpis kódu 4.2: Hlavní smyčka zpracování - vnější část algoritmu

```
%%% hlavni smycka %%%
for i=1:inputSizeSeg
    % segmentace vstupu
    x_maxL(xMax - Ni(1) + 1 : xMax) = xL((i-1) * N + 1 : i * N);
    x_maxR(xMax - Ni(1) + 1 : xMax) = xR((i-1) * N + 1 : i * N);
    x_max = complex(x_maxL, x_maxR);

    if (i >= 172)   % HP
            if PocetVlaken ~= 1
                        [y_maxL, y_maxR, tL2{1}, tR2{1}] = fzerofft2ch(x_max, y_maxL, y_maxR,
                            N, Ni(1), tL2{1}, tR2{1}, H2{1}, PocetVlaken, 1, Ni);     % 1
                           vlakno
                        [y_maxL, y_maxR, tL2{2}, tR2{2}] = fzerofft2ch(x_max, y_maxL, y_maxR,
                            N, Ni(2), tL2{2}, tR2{2}, H2{2}, PocetVlaken, 2, Ni);     % 2
                           vlakno
                    % n-te vlakno
                if Nhmax ~= 512
                    for kk=1: expoend    % expoend se meni podle posledniho nejvetsiho
                       segmentu rozdelene impulsove odezvy
                        n = kk * 2;
                        k = 2^kk;
                        if mod(i, k) == 0
                            [y_maxL, y_maxR, tL2{n+1}, tR2{n+1}] = fzerofft2ch(x_max, y_maxL,
                                y_maxR, N, Ni(n+1), tL2{n+1}, tR2{n+1}, H2{n+1}, PocetVlaken,
                                n+1, Ni);
```

```
                        [y_maxL, y_maxR, tL2{n+2}, tR2{n+2}] = fzerofft2ch(x_max, y_maxL,
                            y_maxR, N, Ni(n+2), tL2{n+2}, tR2{n+2}, H2{n+2}, PocetVlaken,
                            n+2, Ni);
                    end
                end
            end
        else
            [y_maxL, y_maxR, tL2{1}, tR2{1}] = fzerofft2ch(x_max, y_maxL, y_maxR, N, Ni(1),
                tL2{1}, tR2{1}, H2{1}, PocetVlaken, 1, Ni);     % 1 vlakno
        end
else  % DP
        if PocetVlaken ~= 1
                    [y_maxL, y_maxR, tL{1}, tR{1}] = fzerofft2ch(x_max, y_maxL, y_maxR, N
                        , Ni(1), tL{1}, tR{1}, H{1}, PocetVlaken, 1, Ni);     % 1 vlakno
                    [y_maxL, y_maxR, tL{2}, tR{2}] = fzerofft2ch(x_max, y_maxL, y_maxR, N
                        , Ni(2), tL{2}, tR{2}, H{2}, PocetVlaken, 2, Ni);     % 2 vlakno
            % n-te vlakno
            if Nhmax ~= 512
                for kk=1: expoend    % expoend se meni podle posledniho nejvetsiho segmentu
                    rozdelene impulsove odezvy
                    n = kk * 2;
                    k = 2^kk;
                    if mod(i, k) == 0
                        [y_maxL, y_maxR, tL{n+1}, tR{n+1}] = fzerofft2ch(x_max, y_maxL,
                            y_maxR, N, Ni(n+1), tL{n+1}, tR{n+1}, H{n+1}, PocetVlaken, n+1, Ni
                            );
```

```
                        [y_maxL, y_maxR, tL{n+2}, tR{n+2}] = fzerofft2ch(x_max, y_maxL,
                            y_maxR, N, Ni(n+2), tL{n+2}, tR{n+2}, H{n+2}, PocetVlaken, n+2, Ni
                            );
                    end
                end
            end
        else
            [y_maxL, y_maxR, tL{1}, tR{1}] = fzerofft2ch(x_max, y_maxL, y_maxR, N, Ni(1), tL
                {1}, tR{1}, H{1}, PocetVlaken, 1, Ni);     % 1 vlakno
        end
    end
    %%% vysledny synchronizovany vystup - vybiram N vzorku z prodlouzeneho bufferu maximalni
        delky
    yL((i-1) * N + 1 : i * N) = y_maxL(1 : N);
    yR((i-1) * N + 1 : i * N) = y_maxR(1 : N);
    %%% posunuti vstupnich a vystupnich zasobniku o N vzorku s kazdym i
    x_maxL(1 : xMax - N) = x_maxL(N + 1 : xMax);
    x_maxR(1 : xMax - N) = x_maxR(N + 1 : xMax);
    y_maxL(1 : yMax - N) = y_maxL(N + 1 : yMax);
    y_maxR(1 : yMax - N) = y_maxR(N + 1 : yMax);
    y_maxL(yMax - N + 1 : yMax) = zeros(1, N);          % zamezeni pricitani predesleho segmentu
    y_maxR(yMax - N + 1 : yMax) = zeros(1, N);          % zamezeni pricitani predesleho segmentu
end;
```

Výpis kódu 4.3: Funkce fzerofft2ch - realizace vnitřní části algoritmu

```
function [y_maxL, y_maxR, tL_n, tR_n] = fzerofft2ch(x_max, y_maxL, y_maxR, N, Nk, tL_n, tR_n, Hk,
    PocetVlaken, PoradiVlakna, Ni)
xMax = length(x_max);
yMax = length(y_maxL);
Nn = yMax/N;                        % delka vystupniho bufferu v nasobcich N
X_n = fft((x_max(xMax - Nk + 1 : xMax)), 2 * Nk);
Y_n = X_n .* Hk;
y_n = fft(conj(Y_n), 2 * Nk)/2/Nk;
yL_n = real(y_n(1 : Nk)) + tL_n;
yR_n = imag(y_n(1 : Nk)) + tR_n;
tL_n = real(y_n(Nk + 1 : 2 * Nk));
tR_n = imag(y_n(Nk + 1 : 2 * Nk));
B = 1;
segmentvector = Ni;
for j=1:PocetVlaken
    % synchronizace
    if j == PoradiVlakna
        y_maxL(yMax - (Nn - (B - Nk/N)) * N + 1 : yMax - (Nn - B) * N) = y_maxL(yMax - (Nn - (B -
            Nk/N)) * N + 1 : yMax - (Nn - B) * N) + yL_n;
        y_maxR(yMax - (Nn - (B - Nk/N)) * N + 1 : yMax - (Nn - B) * N) = y_maxR(yMax - (Nn - (B -
            Nk/N)) * N + 1 : yMax - (Nn - B) * N) - yR_n;
    end
    % priprav pozici ulozeni
    B = B + (segmentvector(j)/N);
end
```

4.2.2 Vnitřní část algoritmu

Vniřní část algoritmu představuje funkce `fzerofft2ch`. Funkce vrací následující proměnné.

- `y_maxL` výstupní zásobník `L` naplněn aktuálním výsledkem konvoluce
- `y_maxR` výstupní zásobník `R` naplněn aktuálním výsledkem konvoluce
- `tL_n` dočasný zásobník `L` naplněn druhou polovinou výsledku konvoluce
- `tR_n` dočasný zásobník `R` naplněn druhou polovinou výsledku konvoluce

Funkce očekává tyto vstupní parametry.

- `x_max` aktuální vstupní segment vstupující do konvoluce
- `y_maxL` výstupní zásobník `L` naplněn aktuálním výsledkem konvoluce
- `y_maxR` výstupní zásobník `R` naplněn aktuálním výsledkem konvoluce
- `N` délka nejmenšího segmentu
- `Nk` délka aktuálního segmentu výpočtu
- `tL_n` dočasný zásobník `L` naplněn druhou polovinou výsledku konvoluce
- `tR_n` dočasný zásobník `R` naplněn druhou polovinou výsledku konvoluce
- `Hk` aktuální segment kmitočtové charakteristiky impulsové odezvy
- `PocetVlaken` počet paralelních výpočtů
- `PoradiVlakna` aktuální pořadí vlákna
- `Ni` vektor délek segmentů impulsové odezvy

4.2.3 Změna impulsové odezvy

Změna z první impulsové odezvy na druhou je realizována podmíněným příkazem, na základě kterého se poté vykonává jiný blok kódu, který volá funkci `fzerofft2ch` s jinou impulsovou odezvou. Ve výše zobrazeném výpisu kódu je změna explicitně nastavena při 171. průchodu hlavní smyčkou z první impulsové odezvy `H` na druhou `H2`. Příkaz představuje řádek
`if (i >= 172) %HP`.

Tímto způsobem je změna impulsových odezev prováděna skokově. Kritické místo přechodu v okolí 171. zpracovávaného vstupního segmentu bez potlačení přechodového jevu bude analyzováno v kapitole 6.1.

4.3 Realizace plynulého přechodu mezi odezvami

Plynulý přechod mezi odezvami je realizován na základně teorie popsané v kap. 2.3.2. Přechod je realizován v časové oblasti, tedy ve vnější části algoritmu. Realizaci lze popsat na následujícím zjednodušeném výpisu kódu 4.4. Vnitřní část (cyklus) algoritmu bude nahrazena červeně zabarveným textem.

Segmentace vstupu v hlavní smyčce probíhá stále stejným způsobem. Podmíněný příkaz nejdříve zajistí vykonávání bloku pro výpočet konvoluce vstupního segmentu s první impulsovou odezvou (zde značeno `H1`).

Po splnění podmínky `(i == 172)` se spustí blok pro výpočet tzv. přechodového segmentu. Zde se počítá konvoluce vstupního segmentu s odezvou `H1`. Výsledek je váhován plynule sestupující funkcí pro vytvoření „fade out“ posloupnosti a uložen do výstupní posloupnosti. Následně je nezbytné zcela vynulovat výstupní zásobníky pro pravý a levý kanál `y_maxL, y_maxR` tak, aby nedocházelo k používání dříve vypočtených vzorků v dalším kroku.

Dále se spustí pro tentýž vstupní segment vnitřní cyklus s odezvou `H2`. Ve skutečnosti se jedná o konvoluci vstupního segmentu s prvním segmentem odezvy `H2`. Výsledek je váhován plynule vzrůstající funkcí pro vytvoření „fade in“ posloupnosti, sečten se sestupující posloupností z výše popsaného bloku a uložen do výstupní posloupnosti `yR, yL`.

Následující řádky kódu jsou již totožné s řádky ve výpisu 4.2. Jediný rozdíl je v desegmentaci výstupní posloupnosti z výstupních zásobníku. Jelikož přechodový segment (zde 172. krok) má svou vlastní desegmentaci, je nezbytné běžnou desegmentaci ošetřit tak, aby nedocházelo k duplikaci vzorků ve výstupních zásobnících. Ošetření je provedeno příkazem `if (i ~= 172)`.

Výpis kódu 4.4: Realizace plynulého přechodu, `TVConvHann.m`

```
for i=1:inputSizeSeg
    % segmentace vstupu
    if (i > 172)
            --VNITŘNÍ CYKLUS, impulsova odezva H2--
    elseif (i == 172) % prechodovy segment delky N
            --VNITŘNÍ CYKLUS zpracovávající poslední segment H1--
            % realizace sestupne posloupnosti
            yL((i-1) * N + 1 : i * N) = y_maxL(1 : N).*fade(N+1:end);   % fade out
            yR((i-1) * N + 1 : i * N) = y_maxR(1 : N).*fade(N+1:end);   % fade out
            y_maxL(1:end) = zeros(1, length(y_maxL)); % vynulovani vystupniho zasobniku
            y_maxR(1:end) = zeros(1, length(y_maxL)); % vynulovani vystupniho zasobniku

            --VNITŘNÍ CYKLUS zpracovávající první segment H2--
            yL((i-1) * N + 1 : i * N) = yL((i-1) * N + 1 : i * N) + y_maxL(1 : N).*fade(1:N); %
               fade out + fade in
            yR((i-1) * N + 1 : i * N) = yR((i-1) * N + 1 : i * N) + y_maxR(1 : N).*fade(1:N); %
               fade out + fade in
    else
           --VNITŘNÍ CYKLUS, impulsová odezva H1--
    %%% vysledny synchronizovany vystup - desegmentace - vybiram N vzorku z prodlouzeneho bufferu
        maximalni delky
    if (i ~= 172) % 172-ty prechodovy segment ma vlastni vystupni desegmentaci
        yL((i-1) * N + 1 : i * N) = y_maxL(1 : N);
        yR((i-1) * N + 1 : i * N) = y_maxR(1 : N);
    end
```

```
    %%% posunuti vstupnich a vystupnich zasobniku o N vzorku s kazdym i
    x_maxL(1 : xMax - N) = x_maxL(N + 1 : xMax);
    x_maxR(1 : xMax - N) = x_maxR(N + 1 : xMax);
    y_maxL(1 : yMax - N) = y_maxL(N + 1 : yMax);
    y_maxR(1 : yMax - N) = y_maxR(N + 1 : yMax);
    y_maxL(yMax - N + 1 : yMax) = zeros(1, N);          % zamezeni pricitani predesleho segmentu
    y_maxR(yMax - N + 1 : yMax) = zeros(1, N);          % zamezeni pricitani predesleho segmentu

end
```

5 REALIZACE KONVOLUČNÍHO PROCESORU JAKO VST ZÁSUVNÝ MODUL

V této kapitole budou realizovány dva VST zásuvné moduly na základě předešlé teorie a simulace v *Matlabu*. Pro ověření změny impulsové odezvy v reálném čase bez zvukových artefaktů slouží zásuvný modul `FastFir2H`. Druhý zásuvný modul `FastFirMultiThread` slouží k ověření výpočtu rychlé konvoluce pro dlouhé impulsové odezvy.

VST (Virtual Studio Technology) je jedním z formátů zásuvných modulů, které se používájí pro práci v hostitelské aplikaci. Pro zpracování audio signálu je hostitelskou aplikací tzv. DAW (Digital Audio Workstation). Pro vývoj zásuvného modulu ve formátu VST je nutné stáhnout volně dostupný standardní vývojový kit poskytovaný firmou *Steinberg*. Formát VST je spustitelný jak na operačním systému *OS X*, tak na operačním systému *Windows*.

Zásuvné moduly budou realizovány pomocí jazyka C++. Při vývoji budou používány knihovny a výhody standardu C++11. Výsledné moduly `FastFirMultiThread` a `FastFir2H` ve formátu VST jsou součástí přilohy a jsou spustitelné na operačním systému *OS X* a *Windows*.

5.1 Problematika vývoje zásuvných modulů

Před samotnou realizací je vhodné zrekapitulovat úskalí spojená s vývojem zásuvných modulů a princip zpracování audio signálu v jazyce C++.

5.1.1 Segmentální zpracování v jazyce C++

Aby mohl být audio signál zpracováván na výpočetním prostředku v reálném čase, je nutné tento signál zpracovávat po segmentech. Načtení vstupních vzorků signálu ke zpracování a jejich zpětné uložení nebo přehrání systémem je tedy realizováno po segmentech určité délky. Segment bude nadále značen pojmem *audio buffer*[1].

Segmentální zpracování v jazyce C++ představuje typicky funkce, která s pravidelným intervalem očekává nový segment vzorků. Tato funkce obstarává získ nových vzorků a uložení či přehrání vzorků zpracovaných. Příkladem takové funkce může být funkce `void processBlock()` z výpisu 5.1. Zde jsou nové a zpracované vzorky uloženy v objektu `buffer`, který je parametrem funkce a představuje vstupní

[1]Délku segmentu může typicky přenastavit uživatel v hostitelské aplikaci.

Výpis kódu 5.1: Příklad funkce pro segmentální zpracování ve frameworku JUCE

```
void AudioProcessor::processBlock(AudioSampleBuffer& buffer)
{
    const int totalNumInputChannels = getTotalNumInputChannels();
    const int totalNumOutputChannels = getTotalNumOutputChannels();

    for (int i = totalNumInputChannels; i < totalOutputChannels; i++)
        buffer.clear(i, 0, buffer.getNumSamples());

    float* ld = buffer.getWritePointer(0);
    float* rd = buffer.getWritePointer(1);

    for (int i = 0; i < buffer.getNumsamples(); i++)
    {
        ld[i] = 0.0f;   // zpracovane vzorky
        rd[i] = 0.0f;   // zpracovane vzorky
    }
}
```

a výstupní zásobník zpracování. Funkce nejdříve zjistí celkový počet vstupních a výstupních kanálů a poté přistupuje přes ukazatele `ld`, `rd` na nová data z objektu `buffer`, jenž představuje *audio buffer*.

Zpracování audio dat je obstaráváno jedním výpočetním vláknem – *audio vlákno.* Funkce `void processBlock()` je z tohoto vlákna volána pravidelně s určitým intervalem a na nic nečeká. Při zpracování vstupního signálu je nutné zajistit výstupní zpracovaná data (vzorky) včas tak, aby je bylo možné uložit do výstupního zásobníku `buffer`. Pokud se tak nestane, vzorky nebudou uloženy a při zpracování či přehrání v reálném čase se chybějící vzorky projeví jako zvukové artefakty. Níže je uvedeno několik doporučení pro správný chod algoritmu zpracovávající audio data. Body jsou vztaženy k audio vláknu.

- Nealokovat a dealokovat paměť
- Nepoužívat operátor `new` a `delete`
- Nevolat objekty, které `new` a `delete` používají interně
- GUI je obstaráváno vlastním vláknem
- Data alokovat do zásobníků typu *stack*
- Data alokovat dopředu
- Mít kontrolu nad životností objektu

5.1.2 Reprezentace audio dat

Audio data jsou v zásuvném modulu reprezentovány jako typ `float` s hodnotami v rozmezí od -1 do 1. Při překročení těchto hodnot pro daný vzorek dochází k digitálnímu ořezu signálu (*digital clipping*). Při konvoluci dvou signálů dochází k akumulaci několika vzorků a k tomuto jevu může snadno dojít. Zejména pak při zpracování signálu dlouhými impulsovými odezvami. Ovládáním hlasitosti v hostitelské aplikaci nelze přistupovat na vniřní data zásuvného modulu. Proto je nutné kontrolovat aktuální hodnotu vzorku, který má být přehrán uvnitř zásuvného modulu a adekvátně jej váhovat.

5.2 Použití frameworku JUCE

Vývoj zásuvného modulu probíhal ve frameworku *JUCE*. Jedná se o multiplatformní framework spolupracující s běžnými IDE[2] Pro nekomerční účely je volně stažitelný a lze jej získat z odkazu `https://juce.com`.

Tento framework byl zvolen z důvodů snadno dostupných knihoven, jednoduché realizace grafického uživatelského prostředí a výborné dokumentace. Navíc, kód

[2]Integrated Development Environment - Integrované vývojové prostředí např. MS Visual Studio, XCode, Code::Blocks atd.

napsaný v tomto frameworku je snadno přeložitelný na všech běžně známých operačních systémech a zásuvný modul lze přeložit do všech podporovaných formátů zásuvných modulů jako *VST, AU, RTAS* a *AAX*. Pro práci s *VST* formátem je nutné do frameworku vložit patřičné knihovny firmy Steinberg. Zásuvný modul byl vyvíjen ve frameworku verze 5.3.1.

5.2.1 Třídy a hlavní funkce frameworku

Základním kamenem zásuvného modulu jsou dvě hlavní třídy `FastFirMultiThreadAudioProcessor` a `FastFirMultiThreadAudioProcessorEditor`[3]. Třídy jsou deklarovány v souborech `PluginProcessor.h` a `PluginEditor.h`. Audio data jsou zpracovávána ve třídě `FastFirMultiThreadAudioProcessor`. Pro zpracování grafiky slouží třída `FastFirMultiThreadAudioProcessorEditor`. Třídy dědí veřejné členské funkce a proměnné z tříd `AudioProcessor` a `AudioProcessorEditor`, které jsou součástí frameworku.

Stěžejní třídy poskytované frameworkem a hojně využívané v implementaci jsou `AudioSampleBuffer` a `juce::dsp::Complex<Type>`. První z nich představuje zásobník pro zpracování audio dat. Druhá třída reprezentuje complexní číslo nezbytné pro výpočet rychlé Fourierovy transformace.

Jak již bylo zmíněno dříve, hlavní funkcí zpracovávající audio data je funkce `processBlock()`. V této funkci se postupně využívají členské proměnné a funkce třídy `FastFirMultiThreadAudioProcessor` představující samotný algoritmus rychlé konvoluce s přičtením přesahu.

5.2.2 Externí knihovna `ctpl.h`

Pro realizaci vícevláknové implementace byla použita externí knihovna `ctpl.h`[4]. Jedná se o knihovnu, která představuje zpracování úloh na více vláknech způsobem *Thread Pool*, který je popsán v teoretické části práce. Ve třídě `FastFirMultiThreadAudioProcessor` je vytvořen objekt `m_threadPool` jako instance externí knihovny.

[3]Názvy tříd se pro druhý zásuvný modul liší. Lze je dohledat v příloze práce

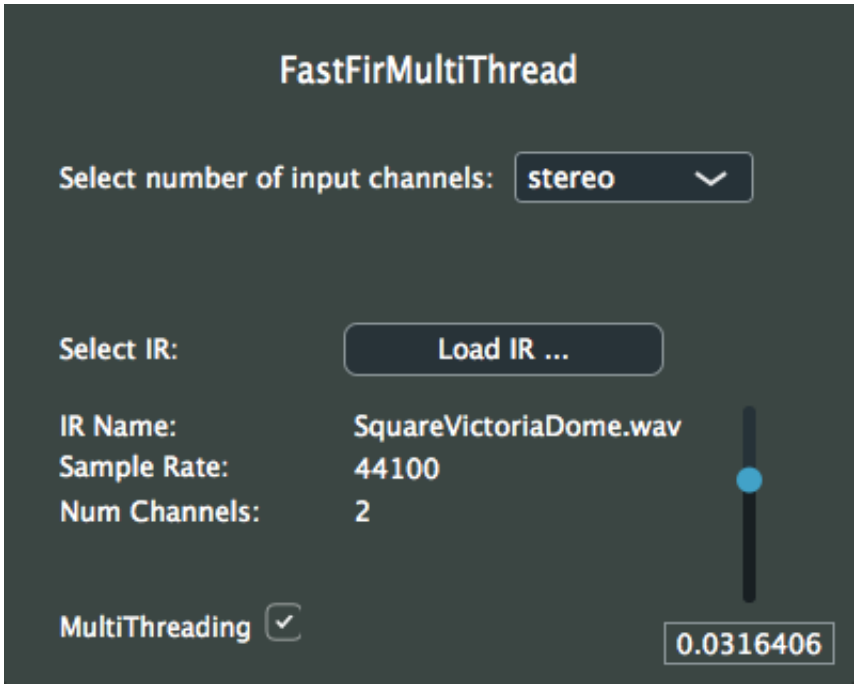
[4]Knihovna je dostupná z `https://github.com/vit-vit/CTPL` pod licencí Apache 2.0. Autorem je Vitaliy Vitsentiy.

5.3 Vlastnosti zasuvného modulu a jeho ovládání

Jelikož byly zásuvné moduly vytvořeny dva, bude v nadcházející podkapitole popsáno ovladání pro oba moduly zvlášť. Bohužel nejsou moduly zcela optimalizované pro všechny podmínky zadané uživatelem, avšak pro ověření teorie postačují. Pro správnou funkčnost je nutné nastavit některé parametry manuálně a dodržet jisté podmínky. Tyto parametry a podmínky budou popsány v kapitole 6.

5.3.1 Grafické uživatelské rozhraní

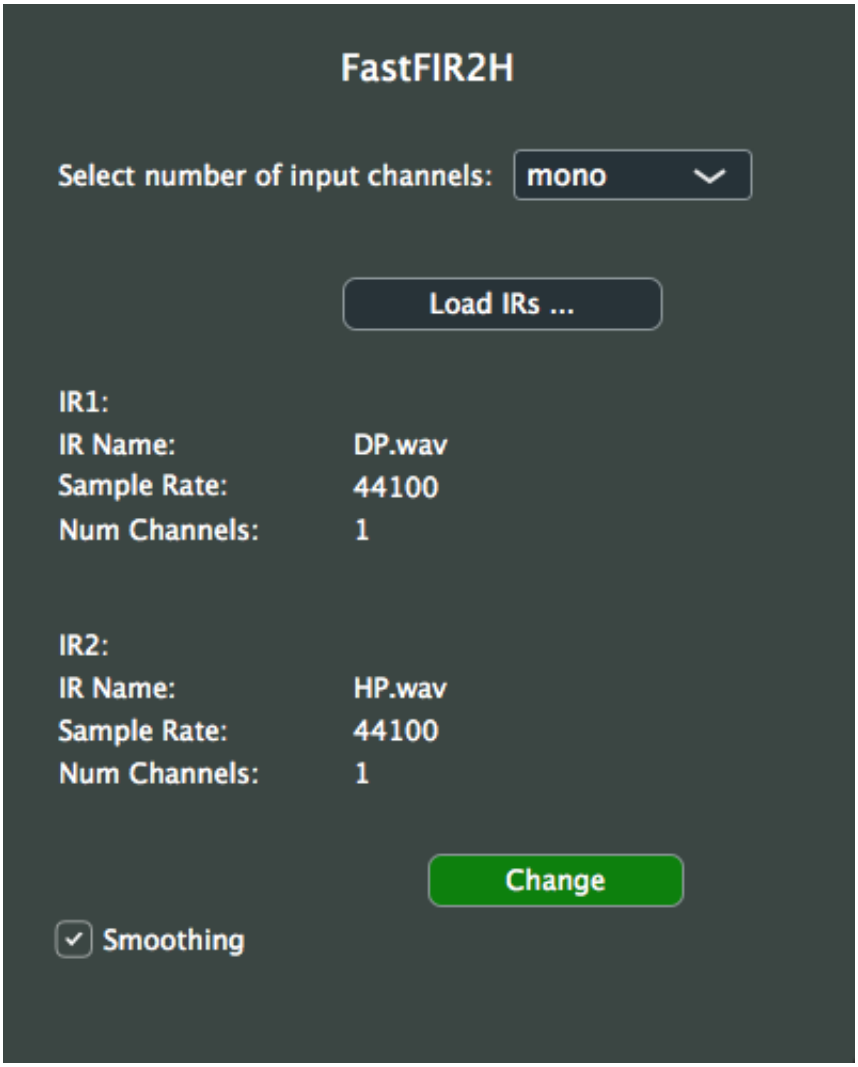
Uživatelské rozhraní modulu `FastFirMultiThread` se skládá z několika komponent. Jako první lze pomocí rozklikávacího tlačítka nastavit počet kanálů vstupního signálu. Primárně je počet nastaven pro mono signál.



Obr. 5.1: Uživatelské rozhraní zásuvného modulu FastFirMultiThread.

Dále lze pomocí tlačítka *Load IR...* vybrat požadovanou odezvu ze souboru ve formátu WAV. Do modulu lze nahrat jednokanálové i dvoukanálové odezvy. Uživateli jsou zobrazeny základní údaje o nahrané impulsové odezvě. Posuvným tlačítkem vpravo lze kontrolovat vnitřní úroveň signálu 5.1.2. Defaultně je hodnota nastavena na nízkou úroveň tak, aby nedocházelo k nežádoucímu digitálnímu zkreslení při načtení dlouhé odezvy. Posledním prvkem je zaklikávací tlačítko pro spuštění více vláken.

Prostředí modulu `FastFir2H` se nepatrně liší. Zde jsou impulsové odezvy ze souborů WAV načteny automaticky po stisknutí tlačítka *Load IRs...*. Rovněž jsou automaticky načteny vzorky pro vytvoření plynulého přechodu. Primárně dochází k filtraci signálu dolní propustí a přepíná se na horní propust. Přechodový jev se sleduje právě v tomto směru. Přepnutí lze vyvolat tlačítkem *Change*.



Obr. 5.2: Uživatelské rozhraní zásuvného modulu FastFir2H.

V levém dolním rohu je zaklikávací tlačítko pro vypnutí a zapnutí potlačení přechodového jevu. Testování a nutné nastavení parametrů proběhne v kapitole 6.

5.3.2 Příprava impulsové odezvy a zásobníků

Po načtení impulsové odezvy z GUI je zavolána funkce `setWav()`, ve které se provede několik operací. Podle reálné délky impulsové odezvy se pomocí funkce `calcNMax()` zjistí délka nová, maximální délky. Tato délka je uložena v proměnné `m_NMax`. Podle

nové délky je vytvořen komplexní zásobník `m_H` pro uložení vzorků obrazu kmitočtové charakteristiky a zásobník `m_T` pro uložení dočasných výpočtů.

Dále jsou vytvořeny vstupní a výstupní zásobníky `m_X` a `m_Y` pomocí funkce `createInputOutputBuffers()`. Po převedení vzorků impulsové odezvy do komplexního zásobníku `h` funkcí `ASBuffer2Complex()` je vypočítána kmitočtová charakteristika funkcí `calcH()`. Tímto způsobem jsou dopředu přichystány vzorky kmitočtové charakteristiky a nezbytné zásobníky algoritmu.

5.4 Rychlá konvoluce s přičtením přesahu v jazyce C++

Jakmile jsou přichystány a naplněny zásobníky `m_X` a `m_H` datami, může proběhnout vnitřní cyklus algoritmu rychlé konvoluce. Princip bude popsán na následujícím výpisu.

Výpis kódu 5.2: Funkce vnitřního cyklu rychlé konvoluce

```
void FastFirMultiThreadAudioProcessor::CalcSingleThread(
   AudioSampleBuffer& buffer)
{
    int modCounter = m_counter % m_xSize;
    int startIdx = modCounter * m_sizeBlock;

    ASBuffer2Complex(buffer, m_isInputMono, m_X, startIdx);

    int offset = 0;
    int idxX = 0;
    int idxH = 0;
    int idxY = m_counter * m_sizeBlock;

    for (int i = 0; i <= m_lastSliceIdx; ++i)
    {
        int k = pow(2, i);

        if (((modCounter + 1) % k) == 0)
        {
            int sliceSize = k * m_sizeBlock;
            int buffSize = 2 * sliceSize;
            std::vector<juce::dsp::Complex<float>> out1(
               buffSize, juce::dsp::Complex<float>(0.0, 0.0))
```

```
                ;
            idxX = (modCounter*m_sizeBlock + m_sizeBlock) -
                sliceSize;
            calc_FFT_X(idxX, k, &out1[0]);
            auto out2(out1); // kopie bufferu out1 do out2
            idxH = offset;
            calc_MULT_FFT_X_H(out1, idxH);
            calc_FFT_X_H(idxY, idxH, k, &out1[0]);
            offset += buffSize;
            idxH = offset;
            idxY += sliceSize;
            calc_MULT_FFT_X_H(out2, idxH);
            calc_FFT_X_H(idxY, idxH, k, &out2[0]);
            offset += buffSize;
        }
    }
}
```

Funkce začíná nastavením počítadla `modCounter`, které slouží k rozlišení spouštění paralelních výpočtů. V této funkci jsou však paralelní výpočty počítány seriově jedním vláknem. Paralelní výpočty na více vláknech jsou popsány v podkapitole 5.5. Funkce dále pokračuje nastavením indexu `startIdx`, který slouží ve funkci `ASBuffer2Complex()` k převodu vstupních dat na komplexní hodnoty uložené v zásobníku `m_X`. Následuje inicializace indexů do vstupních a výstupních zásobníků a vynulování ofsetu, o který se indexy posunují.

V cyklu se poté nastavuje proměnná `k` na hodnoty začínající jedničkou a pokračující mocninami čísla 2. Vždy podle délky nahrané impulsové odezvy. Podmínka `(((modCounter + 1) % k) == 0)` rozhoduje, které výpočty, respektive jak dlouhé segmenty se mají počítat.

Po vytvoření pomocného vstupního zásobníku `out1` a aktualizaci indexu `idxX` dochází k výpočtu spektra pomocí členské funkce `calc_FFT_X()` 5.3. Definice všech dílčích funkcí zde nejsou uvedeny, jsou však dostupné v souboru `PluginProcessor.cpp` v příloze práce. Po úspěšném provedení se v zásobníku `out1` nachází vzorky spektra. Dále je vytvořena kopie vzorků spektra vstupního segmentu v zásobníku `out2`.

Následuje volání funkce `calc_MULT_FFT_X_H()` reprezentující Hadamardův součin vzorků spekter. Funkce `calc_FFT_X_H()` je poslední funkcí, která obstarává výpočet zpětné Fourierovy transformace. První polovinu vzorků aktuálně vypočítaného segmentu ukládá do zásobníku `m_Y` a druhou polovinu do dočasného zásobníku `m_T`. Vzorky v dočasném zásobníku budou použity v dalším volání funkce `pro-`

cessBlock(). S přenastaveným ofsetem a indexy do zásobníků se poté znovu volají funkce calc_MULT_FFT_X_H() a calc_FFT_X_H(). Nyní však pro nový segment odezvy m_H.

Pro impulsové odezvy do délky $2N$ je proměnná k vždy rovna 1. Pro odezvu délky $14N$ bude proměnná cyklicky nabývat hodnot 1, 2, 4. Pro tuto odezvu se bude podle proměné k řídit výpočet delších segmentů.

Výpis kódu 5.3: Výpočet FFT ze vstupního segmentu v jazyce C++

```
void FastFirMultiThreadAudioProcessor::calc_FFT_X(const int
   idxX, const int k, juce::dsp::Complex<float>* out)
{
    int sliceSize = k * m_sizeBlock;
    int buffSize = 2 * sliceSize;
    int fftOrder = log2(buffSize);
    juce::dsp::FFT fft(fftOrder);
    std::vector<juce::dsp::Complex<float>> inp(buffSize, juce
       ::dsp::Complex<float>(0.0, 0.0));

    for (int j = 0; j < sliceSize; ++j)
    {
        inp[j] = m_X[j + idxX];
    }
    fft.perform(&inp[0], out, false);
}
```

5.4.1 Optimalizace algoritmu

Algoritmus je optimalizován tak, jak navrhuje zdroj [7]. Jedná se tedy o opětovné použití již vypočítaného spektra vstupního signálu m_X. Ve vypisu 5.2 se jedná o řádek auto out2(out1), kde již vypočítané spektrum vstupního segmentu je použito pro Hadamardův součin s druhým segmentem odezvy m_H. Tímto krokem je ušetřen jeden výpočet FFT. Popsaná optimalizace je zároveň optimalizací oproti realizaci v Matlabu.

Optimalizace algoritmu založená na výpočtech ve více vláknech procesoru je řešena v podkapitole 5.5.

Návrhy pro další optimalizace jsou například použití jiných efektivnějších knihoven pro výpočet FFT či použití specializovaných kontejnerů pro rychlou práci s daty. V této realizaci je pro výpočet FFT použita knihovna frameworku *JUCE*.

5.5 Implementace algoritmu pomocí více vláken

Vícevláknová implementace byla nejdříve založena na standardních knihovnách `std::thread`. Postupem času se však implementace přesunula k použití externí knihovny `ctpl.h`.

Pro funkční algoritmus rychlé konvoluce s přičtením přesahu zpracovávaný více vlákny je nejdůležitější částí synchronizace. Jedná se o uložení zpracovaných vzorků do výstupních zásobníků ve správném pořadí. Navíc výpočty z jednotlivých vláken sdílí stejný paměťový prostor a při spuštění více vláken najednou může dojít k *undefined behavior*. Tuto problematiku je nutné rovněž ošetřit patřičnou synchronizací.

Inicializace proměnných nezbytných pro synchronizaci se provádí ve funkci `setWav()`. Zde dochází k nastavení počtu výpočtových vláken v `m_threadPool` podle délky impulsové odezvy. Dále je zde zadefinován vektor stavů vláken v proměnné `m_threadRunning`. Jedná se o uchování informace o běžícím nebo dokončeném vlákně z `m_threadPool`. V dalším vektoru `m_threadWhen` je uložena informace o tom, kdy je potřebný výpočet daného vlákna. Pro popis samotné vícevláknové implementace bude použit výpis 5.4.

5.5.1 Předání výpočetní úlohy vláknu

Funkce `CalcMultiThread()` obsahuje pomocné proměnné a funkce stejné jako funkce `CalcSingleThread()`. Jediný rozdíl je v přerozdělení úloh k výpočtu jednotlivým vláknům. Pomocí lamda funkce `push([] (){})` jsou pro každé vlákno v `m_threadPool` předány zásobníky a aktuální ukazatele do zásobníků pro jednotlivé výpočty. Pořadí vlákna je uloženo v proměnné `threadId`. Při spuštění vlákna se uloží informace o aktuálně běžícím vlákně v `m_threadRunning[m_threadId]`. Dále je pro toto vlákno zjištěno, kdy je potřeba výsledek z vlákna uložit do výstupního zásobníku. Po ukončení funkce `calc_FFT_X_H()` je aktuální vlákno považováno za ukončené a tuto informací předá synchronizačnímu objektu pomocí funkce `notify_all()`.

Pro výpočet vstupního segmentu s druhým segmentem impulsové odezvy je zavoláno další vlákno z `m_threadPool`. Stejným způsobem je poté během iterací vnějšího cyklu algoritmu voláno tolik vláken, kolik je pro danou impulsovou odezvu potřeba. Pro odezvu délky $14N$ je to 12 vláken.

Výpis kódu 5.4: Funkce vnitřního cyklu rychlé konvoluce s vícevláknovým výpočtem

```
void FastFirMultiThreadAudioProcessor::CalcMultiThread(
   AudioSampleBuffer& buffer)
{
    int modCounter = m_counter % m_xSize;
```

```
int startIdx = modCounter * m_sizeBlock;
ASBuffer2Complex(buffer, m_isInputMono, m_X, startIdx);
int offset = 0;
int idxX = 0;
int idxH = 0;
int idxY = m_counter * m_sizeBlock;
for (int i = 0; i <= m_lastSliceIdx; ++i)
{
    int k = pow(2, i);
    if (((modCounter + 1) % k) == 0)
    {
        int sliceSize = k * m_sizeBlock;
        int buffSize = 2 * sliceSize;
        std::vector<juce::dsp::Complex<float>> out1(
           buffSize, juce::dsp::Complex<float>(0.0, 0.0))
           ;
        idxX = (modCounter*m_sizeBlock + m_sizeBlock) -
           sliceSize;
        calc_FFT_X(idxX, k, &out1[0]);
        auto out2(out1); // kopie bufferu out1 do out2
        idxH = offset;
        calc_MULT_FFT_X_H(out1, idxH);
        m_threadPool.push([this, out1, idxY, idxH, k] (
           int threadId) {
            m_threadRunning[threadId] = true;
            m_threadWhen[threadId] = m_counter + (k - 1);
            calc_FFT_X_H(idxY, idxH, k, &out1[0]);
            m_threadRunning[threadId] = false;
            m_cv.notify_all();});

        offset += buffSize;
        idxH = offset;
        idxY += sliceSize;

        calc_MULT_FFT_X_H(out2, idxH);
        m_threadPool.push([this, out2, idxY, idxH, k] (
           int threadId) {
            m_threadRunning[threadId] = true;
            m_threadWhen[threadId] = m_counter + k;
            calc_FFT_X_H(idxY, idxH, k, &out2[0]);
            m_threadRunning[threadId] = false;
```

```
                m_cv.notify_all();});
            offset += buffSize;
        }
    }
    {
        std::unique_lock<std::mutex> lock(m_mtx);
        m_cv.wait(lock, [this] { return IsReadyForPlay(); });
    }
}
```

5.5.2 Synchronizace mezi vlákny

Po ukončení výpočtů pro danou iteraci je pomocí funkcí `std::unique_lock <std::mutex>::lock` a `std::condition_variable::wait()` provedena synchronizace jednotlivých vláken. Ve funkci `wait()` se kontroluje návratová hodnota funkce `IsReadyForPlay()`. Pokud je `true` může funkce `processBlock()` pokračovat k přehrání vzorků na výstup. V opačném případě se čeká na výsledky těch vláken, které musí nutně přispět do výstupního zásobníku. Výpis 5.5 zobrazuje definici funkce `IsReadyForPlay()`. Ve funkci se kontroluje informace o ukončení vlákna. Pokud vlákno, které má být ukončeno a přispět tak výsledkem do výstupního zásobníku stále běží, je nutné na něj počkat. V takové případě je návratová hodnota `false`.

Výpis kódu 5.5: Funkce pro synchronizaci jednotlivých vláken

```
bool FastFirMultiThreadAudioProcessor::IsReadyForPlay() const
{
    for (int i = 0; i < m_threadWhen.size(); i++)
    {
        if(m_threadWhen[i] <= m_counter && m_threadRunning[i]
            == true)
        {
            return false;
        }
    }
    return true;
}
```

Poslední synchronizační princip je použit ve funkci `calc_FFT_X_H()`, kde dochází k zápisu dat do společného výstupního zásobníku `m_Y` a dočasného zásobníku `m_T`. Před zápisem si dané vlákno uzamče zámek `m_mtxOutput` a v tu chvíli ma výhradní přístup pro zápis dat. Po zápisu dat je zámek automaticky uvolněn.

5.6 Realizace plynulého přechodu mezi odezvami v jazyce C++

Realizace plynulého přechodu mezi odezvami je implementována v modulu `FastFir2H`. Tento modul byl vytvářen dříve. Část kódu se může lišit a některé funkce nejsou optimalizované jako u druhého modulu. Pro vysvětlení podstaty plynulého přechodu však slouží lépe.

Algoritmus je založen na zdvojeném výpočtu konvoluce s první a druhou odezvou a součtu jejich váhovaných výsledků. Při načtení odezev jsou data uložena ve zvojených zásobnících. Všechny kroky potřebné k načtení a uložení dat se tedy opakují. Výpočet konvoluce v modulu obstarává nyní funkce `calcOutput()`. Funkce počíta spektrum vstupního segmentu, Hadamardův součin a následnou zpětnou Fourierovu transformaci. Zmíněná funkce primárně počítá s první impulsovou odezvou, tedy s dolní propustí o délce 256 vzorků. Po přepnutí se počítá modifikovaná funkce `calcOutputCross()`, ve které se počítá konvoluce dvakrát se starou a novou odezvou. Výsledky se poté váhují a překrývají. Ukazka překryvu je na následujícím výpisu.

Výpis kódu 5.6: Část kódu funkce počítající přechodový segment

```
for (int i = 0; i < sliceSize; i++)
{
    m_out[i] = (m_out[i] * m_fadeOut[i]) + (m_out2[i] *
       m_fadeIn[i]);
}
for (int i = 0; i < sliceSize; ++i)
{
    // 1st half of output is Y
    int idx = ((idxY%m_ySize)+i)%m_ySize;
    memcpy(m_Y+idx, m_out+i, sizeof(juce::dsp::Complex<
       float>));
}
```

Dále je vytvořena poslední funkce `calcOutput2()`, která počítá konvoluci s horní propustí. V hlavní funkci `processBlock` se poté volají tyto tři funkce podle aktuálně nastavených přepínačů, které jsou ovládány z GUI.

Výpis kódu 5.7: Část kódu volání funkcí pro potlačení přechodového jevu

```
if (((modCounter+1) % k) == 0)
        {
            idxX = (modCounter*m_sizeBlock+m_sizeBlock)-
               sliceSize;
            idxH = offset;
            if (m_smoothingOn) // soft
            {
                if (m_playHflag == 1 && m_playH2flag == 0)
                {
                    calcOutput(idxX, idxH, idxY, k);
                }
                else if(m_crossSegmentOn)
                {
                    calcOutputCross(idxX, idxH, idxY, k);
                    m_crossSegmentOn = false;
                }
                else if(m_playHflag==0 && m_playH2flag==1)
                {
                    calcOutput2(idxX, idxH, idxY, k);
                }
                offset += 2*sliceSize;
            }
            else // hard
            {
                if (m_playHflag == 1 && m_playH2flag == 0)
                {
                    calcOutput(idxX, idxH, idxY, k);
                }
                else if(m_playHflag==0 && m_playH2flag==1)
                {
                    calcOutput2(idxX, idxH, idxY, k);
                }
                offset += 2*sliceSize;
            }
```

6 OVĚŘENÍ VÝSTUPU ALGORITMU

V těchto podkapitolách bude ověřena funkčnost algoritmu. Nejdříve bude algoritmus testován v prostředí Matlab, kde bude analyzováno zejména potlačení přechodového jevu. Vícevláknovou implementaci v prostředí Matlab nelze jednoduše analyzovat, neboť výpočty a přehrání výstupních vzorků neprobíhají v reálném čase, ale běží na pozadí.

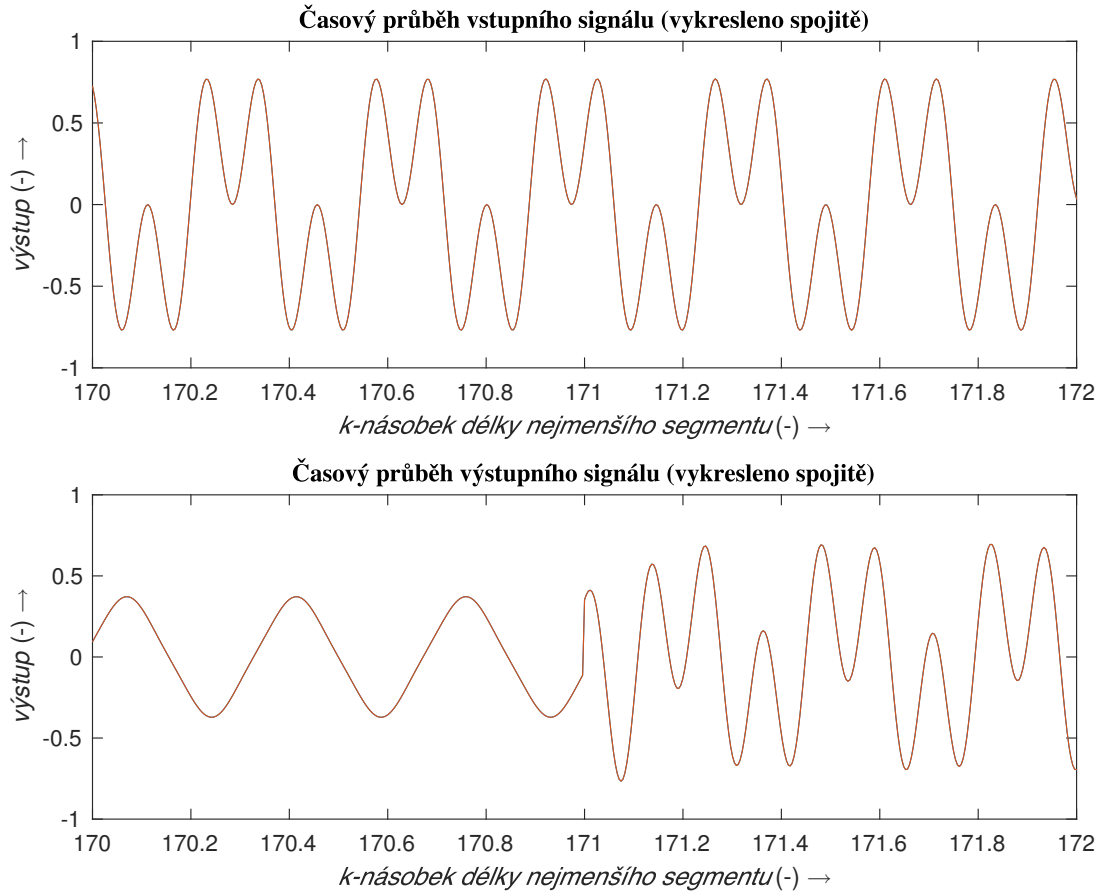
Testování vytvořeného zásuvného modulu probíhá v DAW *Reaper* (ver. 5.75) na platformě *OS X*. Konkrétně byl použit tento výpočetní prostředek.

- operační systém – macOS High Sierra
- verze – 10.13.3
- model – Mac mini (Late 2012)
- procesor – 2,3 GHz Intel Core i7
- pameť – 16 GB 1600 MHz DDR3
- grafika – Intel HD Graphics 4000 1536 MB

6.1 Analýza přechodových jevů v Matlabu

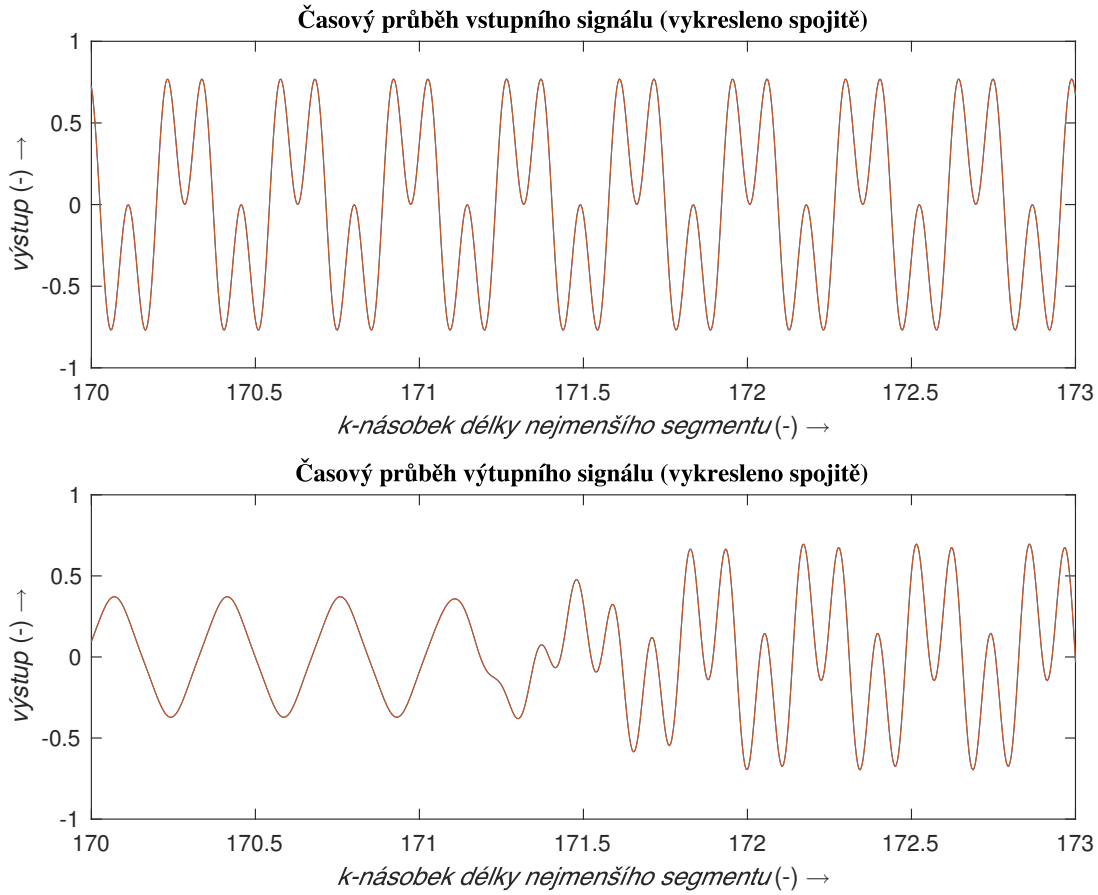
Přiložené skripty reprezentující obě realizace algoritmu jsou testovány stejným vstupním signálem. Vstupní signál je zadefinován tak, jako v kapitole 4.1.1. V prvním případě jde o testování s impulsovými odezvami dolní a horní propusti zadefinované ve stejné kapitole. Dále bude algoritmus testován dlouhou impulsovou odezvou prostoru. Velikost nejmenšího segmentu je nastavena na 256 vzorků. Vzorkovací kmitočet je 44100 Hz.

Výstupy algoritmu lze sledovat na následujících grafech. Komentář k jednotlivým výstupům bude uveden vždy pod příslušným grafem. Pro zvukovou analýzu algoritmu je nutné odkomentovat příslušný řádek ve skriptu a tím zavolat funkci `sound()`.



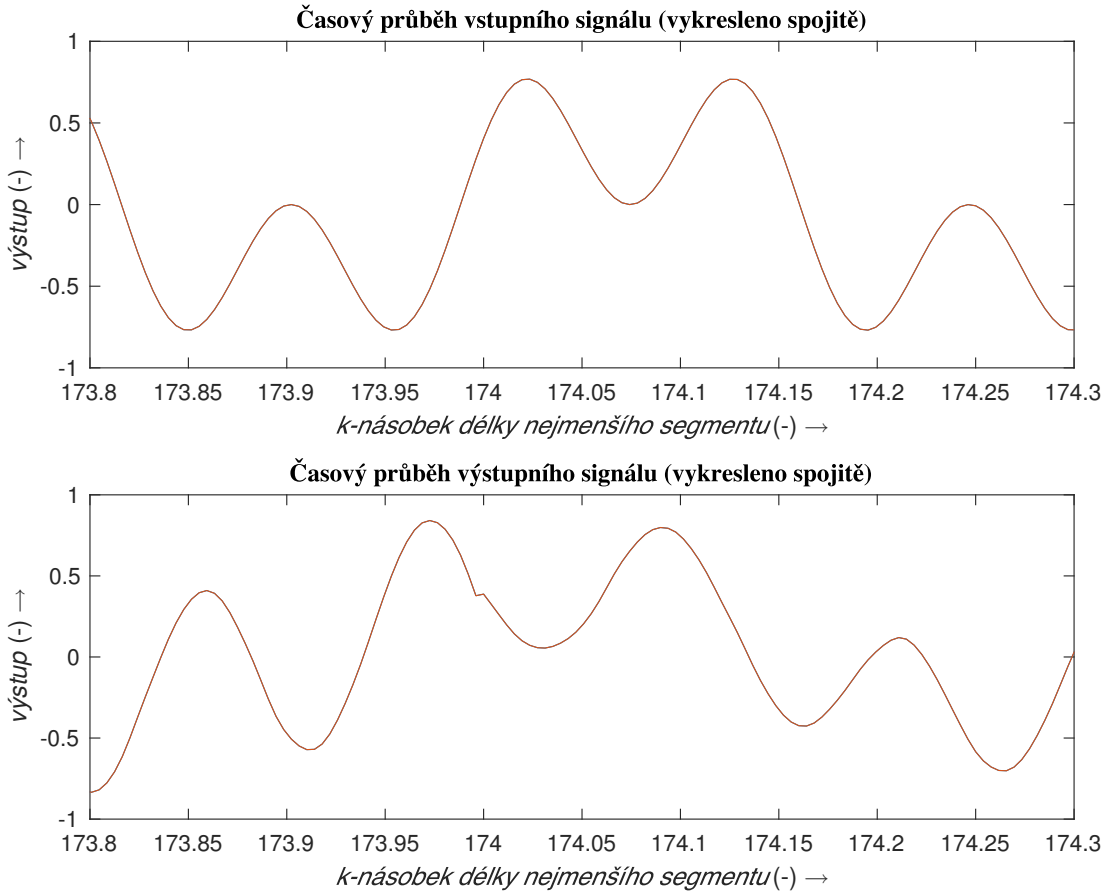
Obr. 6.1: Vstupní a výstupní časový průběh signálu se skokovou změnou filtrace z dolní na horní propust.

První graf je výstupem skriptu `TVConv.m`. V grafu lze pozorovat nespojitost ve výstupním signálu způsobenou skokovou změnou impulsových odezev při filtraci z dolní na horní propust. Nespojitost se nachází na 171. násobku délky nejmenšího segmentu. Tato nespojitost se při přehrání projeví jako zvukový artefakt.



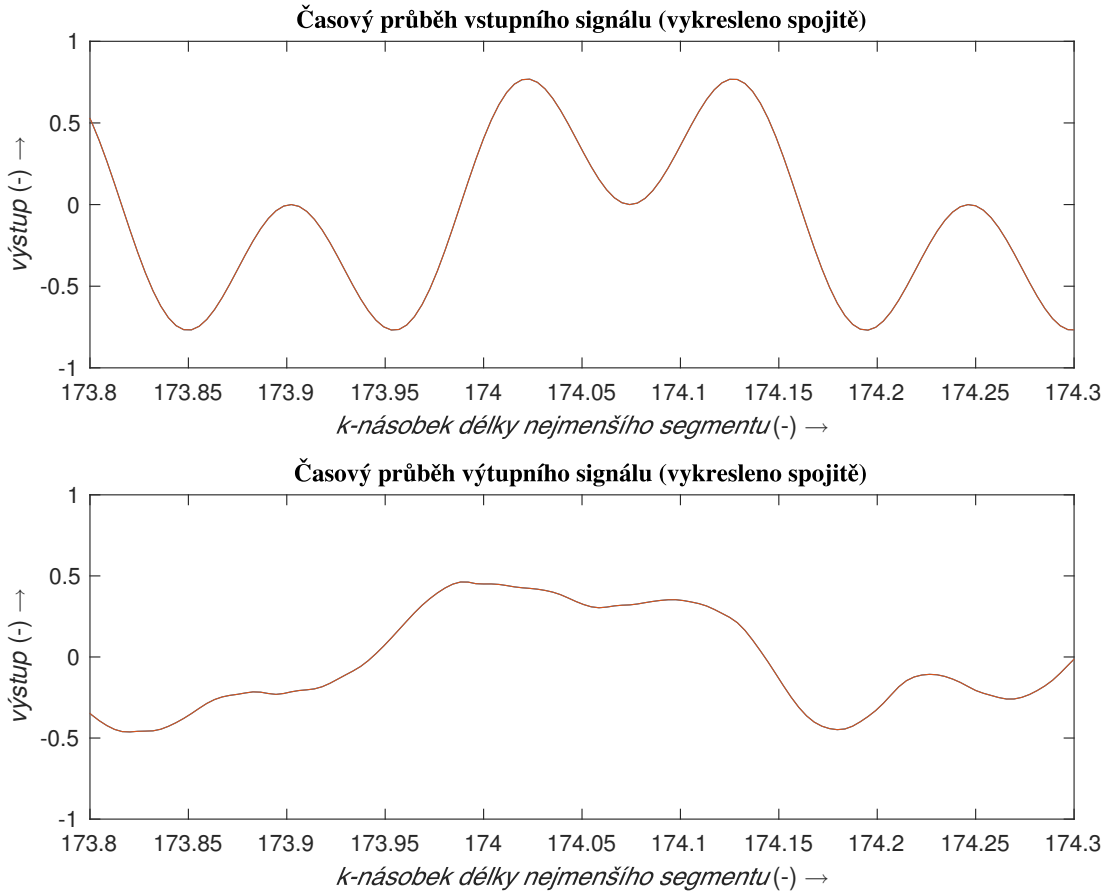
Obr. 6.2: Vstupní a výstupní časový průběh signálu s plynulou změnou filtrace z dolní na horní propust.

Druhý graf je výstupem skriptu `TVConvHann.m`. Jedná se o stejnou filtraci jako v předešlém případě, ovšem s potlačením přechodového jevu. Nespojitost ve výstupním signálu je plynule odstraněna. Plynulou změnu představuje přechodový segment začínající na 171. násobku nejmenšího segmentu a končící na násobku 172. Při přehrání je změna odezev bez zvukového artefaktu.



Obr. 6.3: Vstupní a výstupní časový průběh signálu se skokovou změnou dlouhé impulsové odezvy.

Graf výše je výstupem skriptu `TVConvLong.m`. Zde se jedná o přepnutí dlouhé impulsové odezvy prostoru (22 438 vzorků) načtené ze souboru WAV. Odezva je načtena dvakrát a přepnutí z první odezvy na druhou představuje přepnutí na tutéž odezvu. Cílem tohoto testu byla analýza přechodového jevu při dlouhých odezvách. Při přepnutí dochází ke vzniku nežádoucího přechodového jevu projevujícího se na poslech jako nepříjemné lupnutí. V grafu je kritické místo přechodu přiblíženo k 174. násobku délky nejmenšího segmentu.



Obr. 6.4: Vstupní a výstupní časový průběh signálu s plynulou změnou dlouhé impulsové odezvy.

Poslední graf je výstupem skriptu `TVConvHannLong.m`. Graf zobrazuje plynulý přechod během přepnutí odezvy. Opět se jedná o stejnou impulsovou odezvu jako v předešlém případě. Potlačení přechodového jevu se odehrává na segmentu délky 256 vzorků. Při přehrání je slyšitelný výrazný nástup druhé impulsové odezvy, avšak nepříjemný zvukový artefakt byl odstraněn. Výrazný nástup druhé odezvy by mohl být potlačen přechodovým segmentem s nastavenou větší délkou. Ten však není ve skriptu uvažován.

6.2 Testování zásuvného modulu

Zásuvný modul `FastFirMultiThread` byl testován v hostitelské aplikaci *Reaper*. Zde byl vložen postupně na jednokanálové a dvoukanálové signály. Pro ověření funkčnosti zásuvného modulu je potřeba nastavit velikost *bufferu* v DAW na 1024. Vzorkovací kmitočet aplikace musí být nastaven na hodnotu 44 100 Hz. Dále pro ověření vícevláknové implementace je nutné vybrat velmi dlouhou odezvu např. `Square-VictoriaDome.wav`, která je dostupná v příloze. Pro krátké odezvy není rozdíl mezi jednotlivými implementacemi patrný.

Při testování s dlouhou odezvou na jednovláknové implementaci dochází k „výpadkům“ audio dat. Procesor nestihne včas vypočítat požadované úlohy a při přehrání dochází k zásekům. Po přepnutí na vícevláknovou implementaci je chod a přehrání vzorků plynulé.

Funkčnost lze dále ověřit měřením výkonu CPU v aplikaci *Performance Meter* v DAW. Zde je zobrazeno využití CPU jednotlivými stopami. Testování modulu proběhlo na jedné stopě s jednou instancí modulu pro výše popsaný výpočetní prostředek. Pro jednovláknovou implementaci činilo využití procesoru okolo 13,52 %. Pro vícevláknovou implementaci byla hodnota snížena na 5,47 %. Obrázky z tohoto testu jsou k dispozici v příloze B.

Při testování vícevláknové implementace s nastaveným nižším *bufferem* v hostitelské aplikaci dochází rovněž ke zvukovým artefaktů a zásekům. Vysvětlení pro tento jev pravděpodobně spočívá ve výpočtu FFT z použité knihovny frameworku *JUCE*. Dokumentace frameworku představuje tuto knihovnu jako základní a ne zcela vhodnou pro využití v náročných aplikacích. Pokud by tedy modul obsahoval výpočet FFT na základě jiných knihoven, jeho funkčnost při nastaveném malém *bufferu* může být lepší.

Testování druhého modulu `FastFir2H` probíhá následovně. Je nutné nastavit velikost *bufferu* v hostitelské aplikaci na 256 vzorků, jelikož modul interně pracuje s funkcemi pro potlačení přechodového jevu, které jsou načteny z WAV souborů. Tyto soubory je nutné vložit do následujících adresářů.

- OS X: ~/Users/FastFir2H/wav-files/
- Windows: /Users/FastFir2H/wav-files/

Při vypnutém tlačítku *Smoothing* je patrný přechodový jev při přepnutí z dolní propusti na horní. Aktivací tlačítka se v modulu přenastaví algoritmus a pro tento směr je přechodový jev potlačen.

7 ZÁVĚR

V práci byla představena implementace univerzálního časově-variantního konvolučního FIR procesoru. Teoretická část popisuje princip takového procesoru pro zpracování velmi dlouhých posloupností v reálném čase s potlačeným procesním zpožděním.

Potlačení procesního zpoždění je založeno na vhodné segmentaci dlouhé impulsové odezvy a výpočtu konvoluce pomocí algoritmu rychlé Fourierovy transformace (FFT). Výsledky konvolucí je nutné správně synchronizovat pro správnost výpočtu. Procesní zpoždění je tak sníženo na čas $2NF_{\mathrm{vz}}$, kde N představuje délku nejkratšího segmentu. Funkční algoritmus pro aplikace v reálném čase však předpokládá výpočet na více vláknech procesoru. V práci jsou představeny obecné poznatky při vývoji vícevláknových aplikací.

Dále byl v teoretické části popsán vznik přechodových jevů vznikajících při změně z jedné impulsové odezvy na druhou. Neplynulé přechody mezi impulsovými odezvami se můžou na poslech projevit jako nežádoucí zvukové artefakty. Ty můžou v praxi představovat problem při přehrávání signálu o vysokých úrovních nebo při zpracování zvuku v reálném čase. V práci jsou navrženy efektivní metody pro potlačení slyšitelných přechodových jevů. Jedna z metod realizuje plynulý přechod dvojitým výpočtem konvoluce pro původní a novou odezvu. Výsledky konvolucí jsou poté váhovány a sečteny.

V praktické části práce je vytvořena simulace časově-variantního konvolučního FIR procesoru jako filtrace harmonického signálu odezvami filtru typu dolní a horní propust. Při přepnutí dochází k přechodovému jevu, který je potlačen algoritmem popsaným v teoretické části. Výsledky jsou poté zhodnoceny v poslední kapitole.

Následně byly podle simulace v Matlabu vytvořeny dva VST zásuvné moduly pro ověření teorie. Jeden z modulů představuje simulaci přepnutí mezi odezvami v reálném čase s potlačeným přechodovým jevem. Druhý modul využívá vícevláknové zpracování signálu, které je nezbytné pro zpracování dlouhých impulsových odezev. Modul je schopen za zmíněných podmínek zpracovávat odezvu o délce 128 551 vzorků bez výrazného zpoždění. Výsledky a testování modulů jsou řešeny v poslední kapitole.

LITERATURA

[1] BRHEL, T. *Implementace zvukového procesoru FIR s minimálním procesním zpožděním: bakalářská práce*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2016. 60 stran. Vedoucí práce Ing. M. Balík, Ph.D.

[2] WEFERS, F. *Partitioned convolution of algorithms for realtime auralization*. [online], Logos Verlag Berlin GmbH, 2015. Dostupné z: `http://goo.gl/QDoxQT`.

[3] BRANDTSEGG, Ø., SAUE, S. *Live convolution with time-variant impulse response*. Proceedings of the 19th International Conference on Digital Audio Effects (DAFx-16), Brno, 2016, ISSN 2413-6700.

[4] MYHRE, L. E., BARDOZ, A. H., SAUHE, S., BRANDTSEGG, Ø., TRO, J. *Cross convolution of live audio signals for musical applications*. [online], Dostupné z: `http://oeyvind.teks.no/ftp/Projects/Projects/writings/2013/convBros/LarsAntoineCrossConvolutionFinal.pdf`

[5] WEFERS, F., VORLÄNDER M. *Efficient time-varying FIR filtering using crossfading implemented in the DFT domain*. [online], Forum Acusticum Krakow, 2014. Dostupné z: odkaz.

[6] BALÍK, M. *Diskrétní systémy s konečnou impulsovou charakteristikou. Číslicové zpracování akustických signálů*. Brno: 2010. 21 s.

[7] GARDNER, W. G. *Efficient Convolution without Input/Output Delay*. J. Audio Eng. Soc., 1995. [cit. 20. 11. 2017]. Dostupné z: `http://www.cs.ust.hk/mjg_lib/bibs/DPSu/DPSu.Files/Ga95.PDF`.

[8] MIŠUREC, J., SMÉKAL, Z. *Číslicové zpracování signálů*. Brno: Vysoké učení technické v Brně Fakulta elektrotechniky a komunikačních technologií Ústav telekomunikací, 2012. 187 s. ISBN 978-80-214-4448-5.

[9] ZOLZER, U. *DAFX - Digital Audio Effects*. John Willey & Sons Ltd., 2005, ISBN 0-470-84604-6.

[10] INGLE, K, V., PROAKIS, G, J. *Digital Signal Processing Using MATLAB®*. 3rd ed. Stamford, Conn.: Cengage Learning, 2012, xv, 652 s. ISBN 1111427372.

[11] Zwicker, E., Fastl, H. *Psychoacoustics, Facts and Models*. 2nd edition. Springer-Verlag Berlin, Heidelberg, New York, 1999. ISBN 3-540-65063-6.

[12] Williams, A, *C++ Concurrency in Action - Practical Multithreading*. 1st edition. Manning Publications Co., Shelter Island, New York 11964, 2012, 530 s. ISBN 9781933988771.

[13] Gregoire, M, *Professional C++*. 3rd edition. John Wiley & Sons, Inc., Indianapolis, IN 46256, 2014, 987 s. ISBN 978-1-118-85805-9.

SEZNAM SYMBOLŮ, VELIČIN A ZKRATEK

N	délka nejkratšího segmentu
FIR	Finite Impulse Response – konečná impulsová odezva
DFT	Discrete Fourier Transform – diskrétní Fourierova transformace
FFT	Fast Fourier Transform – rychlá Fourierova transformace
f_{vz}	vzorkovací kmitočet
CPU	Central Processing Unit – centrální procesorová jednotka
GPU	Graphics Processing Unit – grafická procesorová jednotka
VST	Virtual Studio Technology – formát zásuvných modulů
IR	Impulse Response – impulsová odezva
DAW	Digital Audio Workstation – aplikace pro zpracování zvuku

PŘÍLOHA A

Obsah přiloženého CD

`DPTomasBrhel.pdf` - text práce

Adresář `Matlab`:

`fzerofft2ch.m` - funkce algoritmu

`TVConv.m` - skript se skokovou změnou odezvy

`TVConvHann.m` - skript s plynulou změnou odezvy

`TVConvLong.m` - skript se skokovou změnou dlouhé odezvy

`TVConvHannLong.m` - skript s plynulou změnou dlouhé odezvy

`TheSlot.wav` - dlouhá odezva k otestování

Adresář VST:

`FastFirMultiThread.vst` - modul pro OS X

`FastFir2H.vst` - modul pro OS X

`FastFirMultiThread.dll` - modul pro Windows

`FastFir2H.dll` - modul pro Windows

`SquareVictoriaDome.wav` - odezva k testování (128 551 vzorků)- stereo

`TheSlot.wav` - odezva k testování (22 438 vzorků) - stereo

`SteinmanHall.wav` - odezva k testování (52 446 vzorků) - stereo

`FastFirMultiThreadsource` - adresář se zdrojovými soubory

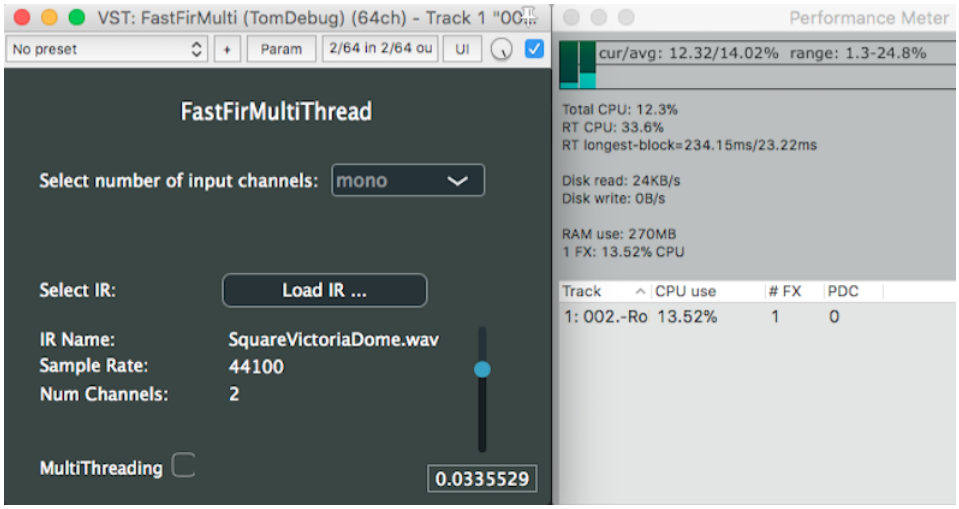
- `PluginEditor.cpp` - zdrojový soubor se třídami pro grafiku
- `PluginEditor.h` - hlavičkový soubor se třídami pro grafiku
- `PluginProcessor.cpp` - zdrojový soubor se třídami audio vlákna
- `PluginProcessor.h` - hlavičkový soubor se třídami audio vlákna
- `ctpl_stl.h` - externí třída pro ThreadPool

`FastFir2Hsource` - adresář se zdrojovými soubory

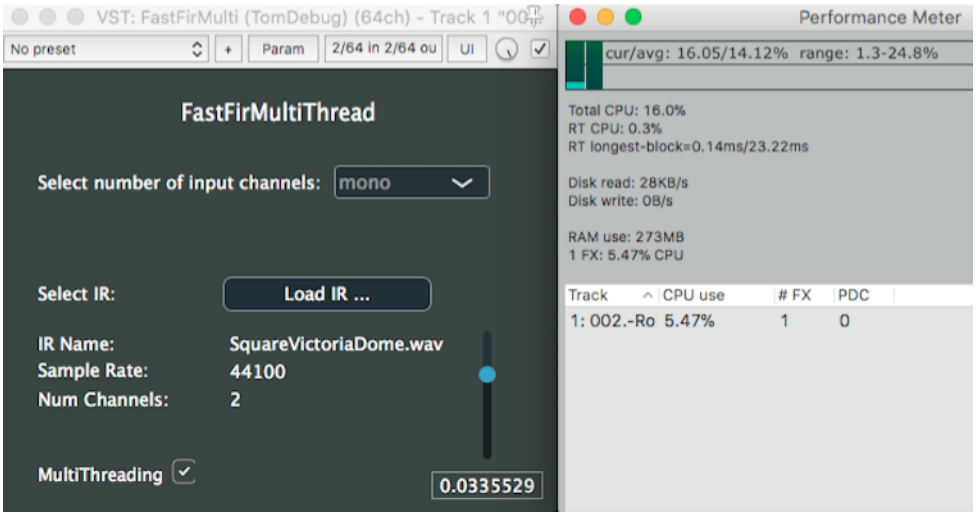
- `PluginEditor.cpp` - zdrojový soubor se třídami pro grafiku
- `PluginEditor.h` - hlavičkový soubor se třídami pro grafiku
- `PluginProcessor.cpp` - zdrojový soubor se třídami audio vlákna
- `PluginProcessor.h` - hlavičkový soubor se třídami audio vlákna
- `fadeIn.wav` - nutno nakopírovat do požadované cesty
- `fadeOut.wav` - nutno nakopírovat do požadované cesty
- `HP.wav` - nutno nakopírovat do požadované cesty
- `DP.wav` - nutno nakopírovat do požadované cesty

PŘÍLOHA B

Ukázka výkonostního testu zásuvného modulu



Obr. 7.1: Výkonostní test jednovláknové implementace modulu FastFirMultiThread v programu Reaper.



Obr. 7.2: Výkonostní test vícevláknové implementace modulu FastFirMultiThread v programu Reaper.