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

APLIKACE PRO DEMONSTROVÁNÍ PRINCIPŮ HYBRIDNÍCH SYNTÉZ ZVUKU

APPLICATION FOR DEMONSTRATING PRINCIPLES OF HYBRID SOUND SYNTHESIS

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

AUTOR PRÁCE — Bc. Šimon Prokop

AUTHOR

VEDOUCÍ PRÁCE — doc. Ing. Jiří Schimmel, Ph.D.

SUPERVISOR

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní program **Audio inženýrství**
specializace Zvuková produkce a nahrávání
Ústav telekomunikací

Student: Bc. Šimon Prokop ***ID:*** 221488
Ročník: 2 ***Akademický rok:*** 2023/24

NÁZEV TÉMATU:

Aplikace pro demonstrování principů hybridních syntéz zvuku

POKYNY PRO VYPRACOVÁNÍ:

Prostudujte metody hybridní syntézy zvukových signálů, tj. syntézy, kde oscilátory využívají digitálních obvodů a pamětí, a následné zpracování zvukového signálu je analogové. V prostředí Matlab realizujte funkce demonstrující principy zvukových generátorů používaných v hybridních syntezátorech, jako jsou generátory horní oktávy, číslicové generátory využívající čítače, Walshovy funkce, multiplexery a paměti ROM a to včetně jejich elementárních stavebních bloků. Vytvořené funkce budou umožňovat zobrazení průběhu signálu ve všech důležitých bodech generátoru. U generátorů s pamětí ROM demonstrujte také syntézy single cycle, multi cycle a wavetable. Vytvořené funkce využijte pro realizaci jednoduchého polyfonního hybridního syntezátoru v Audio Toolboxu prostředí Matlab nebo v prostředí JUCE s pevným a proměnným vzorkovacím kmitočtem a jednoduchým modulem filtrů modifikátoru subtraktivní syntézy.

DOPORUČENÁ LITERATURA:

[1] RUSS, Martin. Sound synthesis and sampling. 2nd ed. Amsterdam: Elsevier, 2004. ISBN 0-240-51692-3.
[2] REISS, Joshua D. a MCPHERSON, Andrew P. Audio effects: theory, implementation and application. Boca Raton: CRC Press, 2014. ISBN 978-1-4665-6028-4.

Termín zadání: 5.2.2024 ***Termín odevzdání:*** 21.5.2024

Vedoucí práce: doc. Ing. Jiří Schimmel, Ph.D.

doc. Ing. Jiří Schimmel, Ph.D.
předseda rady studijního programu

ABSTRAKT

Diplomová práce se zabývá popisem a použitím hlavních principů generátorů zvukových signálů obsažených v hybridních syntezátorech. Jednotlivé metody a dílčí způsoby generování zvukových signálů byly nejprve nastudovány a na základě teorie byly vytvořeny skripty v programu Matlab, které principiální funkce jednotlivých generátorů demonstrovaly. Na základě logiky těchto skriptů byl vytvořen polyfonní hybridní syntezátor ve formátu VST3 v jazyce C++ v prostředí JUCE. Syntezátor umožňuje uživateli volit mezi základními způsoby vyčítání signálů Wavecycle a Wavetable, filtraci zvukového signálu nebo převzorkování celkového signálového procesu.

KLÍČOVÁ SLOVA

Matlab, JUCE, syntezátor, hybridní syntéza, zvukové generátory, klopné obvody, čítače, multiplexery, tabulka, Wavetable, Wavecycle, filtrace, převzorkování

ABSTRACT

The master thesis deals with the description and application of the main principles of sound signal generators contained in hybrid synthesizers. The individual methods and ways of generating audio signals were first studied and based on the theory, scripts were then created in Matlab to demonstrate the principle functions of the individual generators. Based on the logic of these scripts, a polyphonic hybrid synthesizer was created in VST3 format in the C++ language with the JUCE framework. The synthesizer allows the user to choose between basic readout methods Wavecycle and Wavetable, audio signal filtering or oversampling of the overall signal process.

KEYWORDS

Matlab, JUCE, synthesizer, hybrid synthesis, sound generators, flip-flop circuits, counters, multiplexers, look-up table, Wavetable, Wavecycle, filtering, oversampling

PROKOP, Šimon. *Aplikace pro demonstrování principů hybridních syntéz zvuku*. Diplomová práce. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2024. Vedoucí práce: doc. Ing. Jiří Schimmel, Ph.D.

Prohlášení autora o původnosti díla

Jméno a příjmení autora:	Bc. Šimon Prokop
VUT ID autora:	221488
Typ práce:	Diplomová práce
Akademický rok:	2023/24
Téma závěrečné práce:	Aplikace pro demonstrování principů hybridních syntéz zvuku

Prohlašuji, že svou závěrečnou práci jsem vypracoval samostatně pod vedením vedoucí/ho závěrečné 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 závěrečné práce dále prohlašuji, že v souvislosti s vytvořením této závěrečné práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo 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*

*Autor podepisuje pouze v tištěné verzi.

PODĚKOVÁNÍ

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

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů

Úvod

Syntéza zvuku je již od dob analogových syntezátorů velmi oblíbenou metodou při tvorbě umělých zvuků, ale i zvuků reálných nástrojů. Na přelomu digitální a analogové doby se v oblasti zvukové syntézy, jako i v mnoha dalších odvětvích, rozvíjela fúze těchto dvou světů a dostávala se do popředí hudebního průmyslu za pomoci hybridních syntezátorů. Hybridní syntéza se tak na moment stala nedílnou součástí hudební scény prostřednictvím písní od George Michaela, A-ha nebo Eurythmics.

Cílem diplomové práce bylo demonstrovat a implementovat hlavní principy generátorů zvukových signálů využitých právě v hybridních syntezátorech. Konkrétně jsou v práci popsány číslicové generátory s využitím čítačů a generátory využívající vyčítání dat z paměti, včetně vyčítání pomocí nízkofrekvenčního oscilátoru. Skripty demonstrující logiku fungování jednotlivých generátorů byly napsány v programu Matlab a do výsledné aplikace ve formátu VST plug-in modulu byla logika těchto skriptů převedena pomocí prostředí JUCE v jazyce C++.

Teoretická část práce se zaměřuje na logiku klopných obvodů, čítačů a ostatních elementárních prvků v obvodech reálných hybridních syntezátorů. Dále pak na základní funkční celky generátorů včetně jejich implementace logiky a rozdělení dle typů a možností konkrétního systému. V této části jsou vysvětleny rozdíly mezi jednotlivými způsoby generování a vyčítání signálů, jejich omezení a případná rizika použití.

Na základě této teorie jsou v dalších kapitolách práce popsány vytvořené skripty a jejich možnosti nastavení uživatelem. V této části se práce zabývá především implementací jednotlivých funkčních celků generátorů a rozboru jejich kódů, které simulují logiku vytváření základních tvarových průběhů zvukových signálů, včetně jejich složitějších variant. V neposlední řadě jsou zde zobrazeny i grafické výstupy těchto skriptů, které jsou úzce spjaty s teoretickou částí práce.

Po vytvoření skriptů v prostředí Matlab bylo přistoupeno ke konkrétní realizaci polyfonního hybridního syntezátoru. Celková realizace VST plug-in modulu za pomoci frameworku JUCE je vysvětlena ve třetí kapitole, kde je stručně popsáno prostředí JUCE, základní šablona práce a organizační aplikace Projucer. Hlavní náplní kapitoly je posléze popsání architektury celkového programu, úkoly jednotlivých tříd a metod včetně jejich navázání. Nedílnou součástí práce bylo i testování syntezátoru v prostředí externího DAW a demonstrace funkčnosti implementovaných způsobů generování zvukových signálů.

1 Teoretická část

Hybridní syntezátory jsou formou přechodu mezi digitálními a analogovými syntezátory. Jedním z charakteristických prvků těchto syntezátorů je jejich rozložení analogových a digitálních prvků v signálovém procesu generování tvarových průběhů. Tři nejčastější rozložení obvodů v hybridních syntezátorech jsou následující:

1. Generování pilového, obdélníkové nebo impulsového průběhu pomocí logických obvodů a následné zpracování ostatních průběhů pomocí filtračních metod,
2. vyčtením hodnot z paměti (číslicové tabulkové generátory) a následná úprava pomocí analogových obvodů,
3. vytvoření průběhu z části digitální a z části analogovou metodou. [1]

Další uváděná klasifikace hybridních syntezátorů je podle typu metody, která je použita pro tvorbu výsledného zvuku. Jedná se o dělení na:

1. Wavecycle,
2. Wavetable a
3. digitálně řízený oscilátor (dále jen DCO),

kde metody *Wavecycle* a *Wavetable* používají čtení signálů z ROM nebo PROM. *DCO* je jejich nejjednodušší formou, která vyčítá pouze základní průběhy (sinusový, obdélníkový, pilový a impulsní). [2]

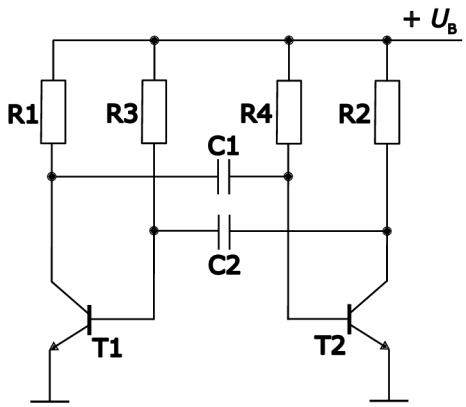
V hybridních syntezátorech se vyskytuje celá řada generátorů signálů, které se v 80. letech minulého století staly stěžejním bodem u většiny předních modelů syntezátorů (ve zkratce např. PPG série Wave 2, Roland Juno-60 nebo Roland D-50 [1]) a v 90. letech se jejich popularita přenesla do digitálních metod generování zvukových signálů. V následujících kapitolách jsou představeny samotné generátory včetně jejich elementárních součástek a metody tvorby výsledného zvuku. [2]

1.1 Číslicové obvody

Základní číslicové obvody používané v generátorech jsou typicky klopné obvody, čítače, multiplexery a demultiplexery.

1.1.1 Klopné obvody a čítače

Klopné obvody (dále jen KO) vytváří přechod mezi dvěma nebo více diskrétními stavy (často mezi 0 a 1), kde přechod mezi těmito stavy není pozvolný, ale skokový. Pro vytvoření skokového přechodu je v obvodu použít setrvačný prvek (nejčastěji kondenzátor), který je vhodně zapojen do aktivního nelineárního obvodu. Samotné skokové stavy jsou následně vyvolány kladnou zpětnou vazbou v obvodu, která je dosažena buď pomocí zapojení, nebo použitím aktivního členu s vnitřní kladnou



Obr. 1.1: Jednoduchý astabilní KO se dvěma tranzistory. [2]

zpětnou vazbou (např. dvoubázová dioda). Často se v obvodech KO objevují i záporné zpětné vazby, které zaručují splnění podmínky pro rozkmitání. [3]

KO dělíme dle stability stavů na astabilní, monostabilní a bistabilní (viz tab. 1.1). Astabilní KO je obvod bez stabilního stavu a jedná se o zapojení s obousměrnou kladnou zpětnou vazbou. Perioda kmitání tohoto KO je dána součtem časových konstant $t_1 = \ln(2) \cdot R_2 \cdot C_1$ a $t_2 = \ln(2) \cdot R_3 \cdot C_2$[1], kde t_1 je doba trvání vysoké úrovně a t_2 doba nízké úrovně (viz obr. 1.1). Nejčastěji se tento typ KO používá pro realizaci oscilátorů obdélníkového průběhu.
Monostabilní KO jsou již nastavitelné a mají jeden stabilní stav, ze kterého lze KO přepnout do nestabilního stavu pomocí spouštěcího impulsu. V tomto stavu KO setrvají po dobu $t_1 = \ln(2) \cdot R_2 \cdot C_1$. Monostabilní KO se nejčastěji používají jako tvarovače nebo čítače impulsů.

Tab. 1.1: Rozdělení klopných obvodů.

Rozdělení klopných obvodů podle stability stavů

typ KO	stav
astabilní	žádný stabilní stav, kmitá z jednoho stavu do dalšího
monostabilní	jeden stabilní stav
bistabilní	dva stabilní stavy
Schmittův	bistabilní KO s hysterezí

Bistabilní KO (dále jen BKO) mají dva stabilní stavy, které lze volitelně přepínat za pomoci vstupních signálů. BKO lze použít v generátorech jako čítače modulu signálu nebo děličky kmitočtů. Samotný obvod má mnoho variant, dále budou uvedeny alespoň nejzákladnější provedení a jejich logika.
Prvním zapojením BKO je ve formě asynchronního obvodu Reset-Set (označení **RS**),

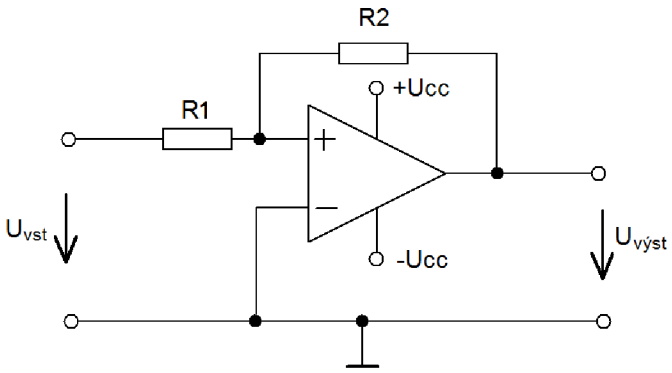
[1] viz [3]. Vycházející úpravou ze vzorce (61) na str. 111

které pracuje na základě logické 1 nebo 0, které přicházejí na Reset/Set vstupy. Při sepnutí logickou 1 na vstupu Reset vyšle obvod na výstupu logickou 0. V opačném případě, když logická 1 dojde na vstup Set, tak na výstupu je nastavena logická 1.

Dalším zapojením BKO je obvod se zkratkou **JK** podle amerického vynálezce Jacka Kilbyho, který tento obvod vytvořil. Jedná se o systém synchronního vstupu, který funguje velmi podobně jako v případě BKO typu **RS**, avšak při přivedení logické 1 na oba vstupy zároveň se výsledná vnitřní hodnota neguje. Dalšími typy mohou být například BKO **D** a **T** (z anglického *Delay* a *Toggle*). Obvod typu **D** je charakteristickou jednobitovou pamětí, přičemž jeho výstup kopíruje vstup pouze v okamžiku náběžné hrany hodinového impulsu na vstupu C (logická 1). Tento stav setrvává na výstupu po dobu trvání logické 0 na vstupu C a je přepsán při přivedení další náběžné hrany (logické 1) na vstupu C. BKO typu **T** je funkcí velmi podobný obvodu typu **D**, avšak s náběžnou hranou dojde na jeho výstupu k inverzi předchozího vstupu. Systém tak mění svůj stav při každé změně hodinového impulsu. Jeho použití je například jako dělička frekvence.[2]

Speciálním typem KO jsou Schmittovi KO (dále jen SKO), které fungují na principu hystereze, tedy pro překlopení do určitého stavu musí být rozdíl přiveden-ého a stávajícího napětí roven alespoň minimální prahové hodnotě $\pm H$. Tím vzniká závislost výstupního signálu nejen na vstupním signálu, ale i na jeho předchozí hodnotě. Hraniční napětí, při kterém se SKO překlopí, lze určit z dílčích rezistorů R_1, R_2 a saturačního napětí $\pm U_S$ následujícím vztahem [5]

$$\pm H = U_S \cdot \frac{R_1}{R_1 + R_2}. \tag{1.1}$$



Obr. 1.2: Schmittův KO s OZ. [6]

Kombinací KO, lze vytvořit čítací obvody, které jsou schopné počítat vstupní hodinový impuls. Základní realizací je kaskáda, kde je výstup KO zapojen do vstupu následujícího KO, tímto je vytvořen například asynchronní binární čítač. Vzhledem

[2]Kapitola 7.5 Sekvenční logické systémy z [4]

k nebezpečí čítání v asynchronním čítači (doba nutná k vyčtení jednoho impulsu se vlivem počtu KO může prodloužit až na dobu jedné periody hodinového impulsu)[3] je možné tento obvod předělat do podoby synchronního čítače za pomoci KO typu T. Typická pravdivostní tabulka výstupů z asynchronního čítače je uvedena v tab. 1.2.

Tab. 1.2: Pravdivostní tabulka výstupů asynchronního čítače se dvěma KO

číslo výstupního signálu	Q_1	Q_2
0	0	0
1	1	0
2	0	1
3	1	1

1.1.2 Multiplexery a demultiplexery

Multiplexer, resp. demultiplexer funguje na principu rozšíření vstupů, resp. výstupů. Multiplexer je adresovací jednotka nebo též přepínač, který podle adresovacích vstupů směřuje signál z jednotlivých vstupů k jednomu konkrétnímu výstupnímu signálu. Počet adresovatelných vstupů k na výstup je dán počtem adresových vstupů a_n a to $k = 2^n$. Výstupem multiplexeru je tedy jeden výstupní signál vybraný ze vstupních signálů pomocí adresových vstupů. Demultiplexer pracuje na opačném principu, kdy na vstup přichází pouze jediný vstupní signál a podle adresových vstupů je mu udělen daný počet výstupů. [1]

1.2 Číslicové generátory

Z výše zmíněných elementárních prvků lze následně sestavit několik základních generátorů zvukových signálu. Generátory jsou nejčastěji ve formě *DCO*, který byl zmíněn na začátku této kapitoly (viz kapitola 1), tedy přehrávají pouze jeden periodický signál s jedním neproměnným tvarem (např. pilového, obdélníkového nebo sinusového průběhu). Další skupina generátorů používá již čtení z paměti ROM a umožňuje tvarování těchto průběhů. Výstup je variabilnější a z uměleckého hlediska tonálně zajímavější.

[3] Kapitola 7.7.2 Synchronní čítače z [4]

1.2.1 Generátory horní oktávy

Nejbližším vyobrazením klasického *DCO* jsou právě generátory horní oktávy. Součástí tohoto systému je 12 (může však být méně nebo více) impulsových oscilátorů v temperovaném ladění, které jsou součástí 12 oktáv. Tyto oscilátory jsou generovány tzv. *Master oscillator* tvořeného krystalovým oscilátorem s vysokým kmitočtem. Posléze je těchto 12 oscilátorů děleno resp. násobeno na děličkách a násobičkách pro vhodnou interpretaci požadovaného tónu. Druhou možností je dělení a násobení samotného *Master* oscilátoru. Tímto vzniká na výstupu obdélníkový signál, který lze posléze filtrovat pro dosažení ostatních průběhů a pomocí klaviatury spouštět požadované tóny.

Děličky a násobičky jsou však ve svém základu binární KO a pracují tak pouze s celými čísly. Je nutné tedy zaokrouhlovat dělící poměry na celá čísla, čímž vzniká aproximační chyba ve výsledné frekvenci výstupního signálu oproti požadované frekvenci z temperovaného ladění. Tuto chybu však lze zanedbat při dostatečně vysoké frekvenci *Master* oscilátoru, která zaručí menší odchylku od požadované frekvence. Pro příklad mějme *Master* oscilátor o frekvenci 500 kHz. Pro vytvoření výstupního signálu tónu D6 (1174,66 Hz) je nutné dělit 425,65. Po zaokrouhlení na 426 je výsledná frekvence 1173,71 Hz. Absolutní procentuální rozdíl mezi požadovanou frekvencí a reálnou frekvencí je 0,08 %, tedy 0,95 Hz. Chybovost samozřejmě bude záviset na výšce požadovaného tónu. Proto například pro nízkofrekvenční signály bude potřeba nižší chybovosti. Pokud by však byl vysokofrekvenční oscilátor pouze o frekvenci např. 50 kHz, tak se dělící poměr zmenší na hodnotu 42,57 (po zaokrouhlení 43) a výsledná frekvence bude pouhých 1162,79 Hz. Tímto vzniká rozdíl již 11,87 Hz resp. $1,01\,\%$. [2]

1.2.2 Generátory číslicových funkcí

Následným duplikováním a zpracováním obdélníkových impulsů lze dosáhnout různých tvarových průběhů. Generátory číslicových funkcí pracují právě s tímto principem a po vygenerování základního obdélníkového signálu tento signál modifikují a sčítají pro dosažení požadovaného průběhu. Běžnými funkcemi demonstrující tento postup tvarování signálu jsou např. Walshovy funkce, generátory pilového průběhu nebo pokročilejší čítání signálů z paměti ROM (někdy též číslicové tabulkové generátory), které navíc využívá adresování vstupů ROM (skrze multiplexer) pro generování složitějšího, popřípadě pseudonáhodného výstupního signálu.[2][3]

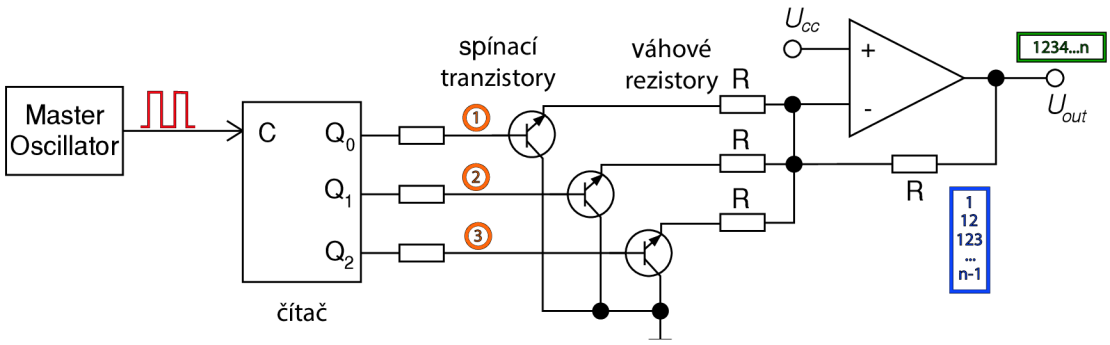
Walshovy funkce

Základním principem Walshových funkcí je již zmíněné sčítání několika obdélníkových signálů do výsledného signálu.[4] V první řadě vstupuje hodinový impuls o frekvenci f_{clock} dané podle následujícího vzorce do čítače

$$f_{\text{clock}} = 2^n \cdot f, \tag{1.2}$$

kde n je počet klopných obvodů nebo též bitů čítače a f je frekvence požadovaného výstupního signálu. Vstupní signál je dále dělen děličkou kmitočtů číslem $k = 2^p$ v každém kroku čítače, kde p je pořadí aktuálního KO v čítači (například 4bitový čítač bude ve svých čtyřech krocích dělit 2, 4, 8 a 16, čímž vzniknou 4 výstupní obdélníkové signály o frekvencích $\frac{f}{2}, \frac{f}{4}, \frac{f}{8}, \frac{f}{16}$). Tyto výstupní signály jsou přivedeny na bázi spínacích tranzistorů, které následně přivádí daný signál na váhové rezistory R o stejné hodnotě odporu. Součty signálů jsou prováděny invertujícím operačním zesilovačem a zpětnou vazbou přivedeny k opakovanému sčítání.

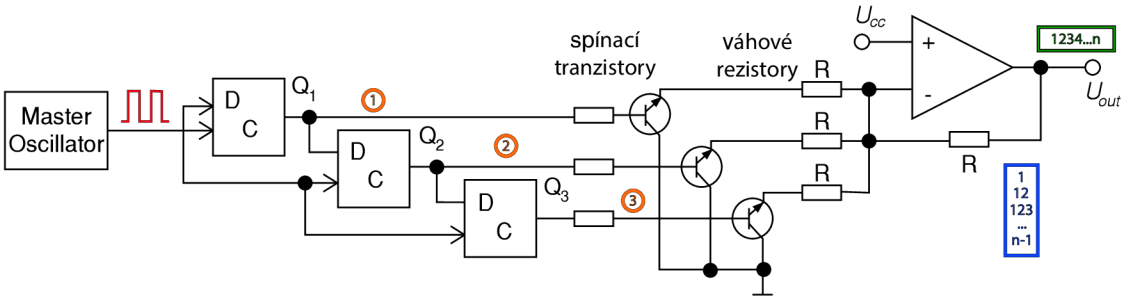
Výstupní signál tohoto zapojení je zobrazen v grafu 2.3 zelenou barvou a jednotlivé mezikroky sčítání čítaných signálů barvou modrou. Dílčí signály čítače jsou zobrazeny v grafu 2.2. [2]



Obr. 1.3: Princip zapojení generátoru Walshových funkcí.

Nahrazením čítače z předchozího zapojení kaskádovým spojením BKO typu **D** je docíleno posunutí jednotlivých výstupních signálů. Každý výstupní signál je zpožděn právě o hodnotu počtu BKO v kaskádovém zapojení nad ním (viz schéma zapojení 1.4). Posléze se opět dílčí signály sčítají na OZ v invertujícím zapojení a stejně jako v předchozím případě jsou zpětnou vazbou přivedeny k opakovanému sčítání pro vytvoření výstupního signálu. Tento obvod může aproximovat v závislosti na množství použité filtrace a počtu děliček sinusový nebo trojúhelníkový průběh. [1]

[4] viz princip zapojení na obr. 1.3, který barevně koresponduje s grafy 2.2 a 2.3



Obr. 1.4: Princip zapojení generátoru Walshových funkcí s KO typu D.

Generátory pilového průběhu

Generátory pilového průběhu fungují na velmi podobné principu jako Walshovy funkce, avšak jejich hlavním rozdílem jsou různé hodnoty váhových rezistorů za spínacími tranzistory (podobně jako tomu je na obr. 2.7). Hodnoty rezistorů se odvíjejí od váhového rezistoru R, na který přichází signál s frekvencí výstupního signálu. Aby bylo možné vytvořit schodovitý průběh, který aproximuje pilový, je nutné, aby poměry váhových rezistorů $R : R_1 : R_2 : \cdots : R_n$ byly v poměru $1 : 2 : 4 : \cdots : 2^n$, kde n je počet bitů čítače. Tímto je zaručeno, že výchylky výstupních signálů přicházející na OZ budou v převráceném poměru, tedy $1 : \frac{1}{2} : \frac{1}{4} : \cdots : \frac{1}{2^n}$. Výsledný signál bude mít schodovitý tvar aproximující pilový průběh v 2^n úrovních. Z tohoto výsledku vyplývá fakt, že čím větší bude počet bitů čítače, tím lépe bude pilový signál aproximovaný.

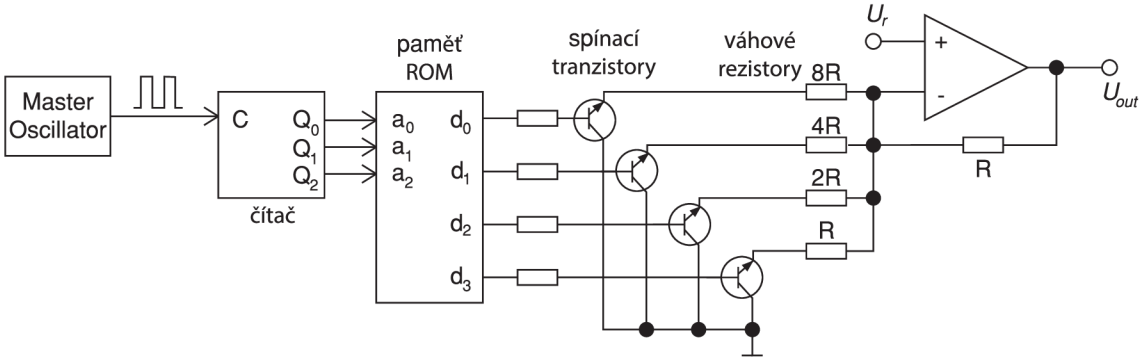
Generátory funkcí s pamětí

Generování signálů z pamětí ROM je složitější variantou dříve zmíněných generátorů. Jejich funkce se odráží do většiny metod generování zvuku, které byly naznačeny na začátku kapitoly 1. Nejdůležitějším prvkem v obvodu je paměť (ROM nebo PROM), která má v sobě uložená data průběhů jednotlivých signálů. Do pamětí přichází impulsy z adresového čítače $a_0, \cdots, a_{n-1}$ (v případě obr. 1.5 se jedná o 3bitový čítač), čímž určí danou $n$bitovou adresu. Na základě zaslané adresy se na výstupu ROM $d_0 \cdots, d_{n-1}$ vyčtou $n$bitová dvojková data, která interpretují výstupní hodnotu signálu v konkrétním kroku (jehož časová délka je určena periodou hodinového impulsu).[5] Rozlišení aproximace dané funkce je určena počtem bitů vyjadřujících dvojková data (pro 3bitové číslo je maximální možné rozlišení $2^3 = 8$ úrovní). Počet dvojkových čísel v posloupnosti udává počet možných harmonických složek, které lze pomocí konkrétní posloupnosti generovat. Pokud by byla posloupnost například

[5] převzato z [3], str. 139

16číselná, lze generovat touto posloupností 1. až 16. harmonickou složku aproximovaného signálu. Všechny ostatní složky by byly brány jako parazitní.

Výstupní signály z paměti pokračují na spínací tranzistory, přes které signál projde na váhové rezistory a dále na OZ. Pokud je na všech spínacích tranzistorech logická 0, jedná se o stav bez přivedeného signálu a na výstupu OZ bude napětí U_r. Pokud však sepne jeden z tranzistorů, například tranzistor na výstupu z d_0, vznikne smyčka s rezistorem $8R$ a výstup nyní bude vykazovat napětí $U_r + \frac{1}{8} \cdot U_r$. Analogicky bude vzrůstat napětí při sepnutí zbylých tranzistorů. Tento mechanizmus generuje různá napětí na výstupu OZ, která mají příslušnou hodnotu dvojkového čísla v analogové podobě. [3]



Obr. 1.5: Princip generátoru funkcí s pamětí ROM. [1]

V dnešní době je podobná logika vyčítání z paměti nedílnou součástí digitálních i softwarových syntezátorů a pracuje na ní celá řada VST nebo AAX plug-in modulů (například Xfer Serum nebo Native Instruments Massive).

1.3 Metody čtení dat z paměti

Pokročilejší metody generování zvukových signálů běžně staví na generátorech s pamětí, přičemž v různých krocích čítají určitá data a tvarují přehrávaný průběh popřípadě pročítají paměť náhodně nebo za přispění LFO. Nejběžnější rozdělení metod je na *Wavecycle*, který se dále dělí na *Single-cycle* a *Multi-cycle*, *Wavetable* a *DCO*.

1.3.1 Wavecycle

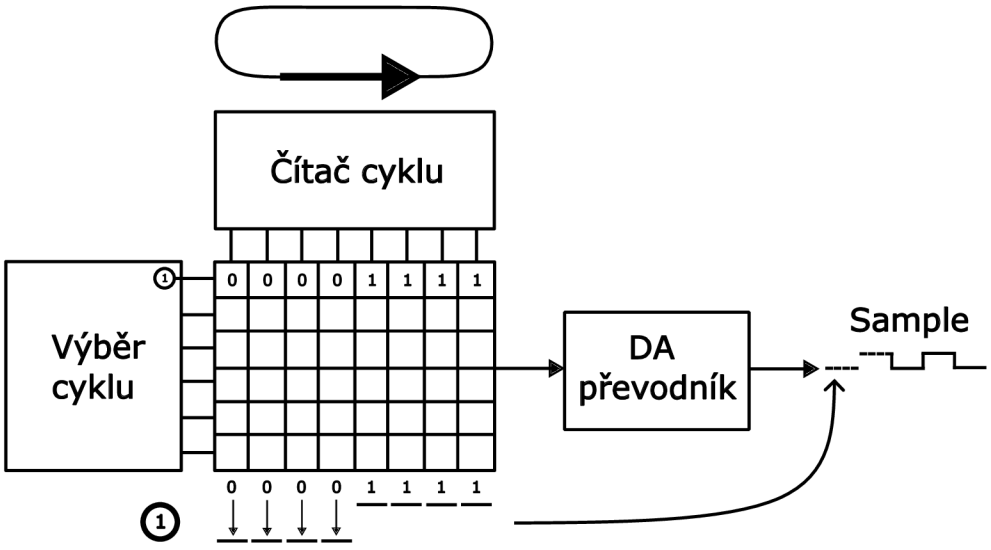
Metoda využívající jednoduchého čtení z tzv. *Look-up table* (dále tabulka), která po dobu jednoho hodinového impulsu posílá zvolený signál z paměti na DA převodník. Tímto lze dosáhnout rychlé změny tvaru zvoleného průběhu a tím pádem i různých spektrálních složek přehrávaného zvuku. Přehrávané úseky zvolené paměti se nazývají cykly (*cycle*) a po přehrání několika těchto cyklů se výstupní průběh nazývá

sample. Samotné vyčítání z paměti může fungovat na dopředném, zpětném nebo náhodném čítání cyklů a volba může být determinována například dynamikou síly úhozu na snímač, LFO modulátorem nebo může být čistě fixní. Nejdůležitější rozdíly v jednotlivých metodách typu Wavecycle (*Single-cycle* a *Multi-cycle*) je délka přehrávaného průběhu a volba opakování. [2]

Single-cycle

Oscilátory používající metodu *Single-cycle* vytváří pevný tvar průběhu výsledného signálu. V porovnání s tvary průběhů generovanými analogovými oscilátory (dále jen VCO) je tato metoda totožná, avšak lze generovat velký počet tvarových průběhů (analogicky by bylo potřeba velký počet volitelných VCO). Z počátku se pro tvarování vlny používal přístup zahrnující modulaci střídou (PWM) signálu, která se měnila pozicí potenciometru. Vzhledem k 2stupňové charakteristice impulsního signálu se prvotní syntezátory vyráběly ze 2 úrovní a samotná variabilita tvarů byla omezená.

První víceúrovňový mechanizmus byl představen v tzv. *Slider scanning* syntezátoru. Výsledný signál je tvarován z n výstupů čítače, které jsou přivedeny na multiplexer. Multiplexer následně pročítá (angl. *scanning*) posuvné potenciometry (angl. *sliders*). Po přečtení všech potenciometrů vzniká jeden cyklus signálu a multiplexer přiřazuje daným čítaným signálům jejich úroveň podle zvolené hodnoty na potenciometru. Pokud by se například nastavila první polovina potenciometrů do maximální polohy a druhá polovina do minimální polohy, vytvořil by se výstupní signál o fixním obdélníkovém tvaru průběhu s 50 % střídou. Tato metoda je však silně omezena počtem potenciometrů. [2]

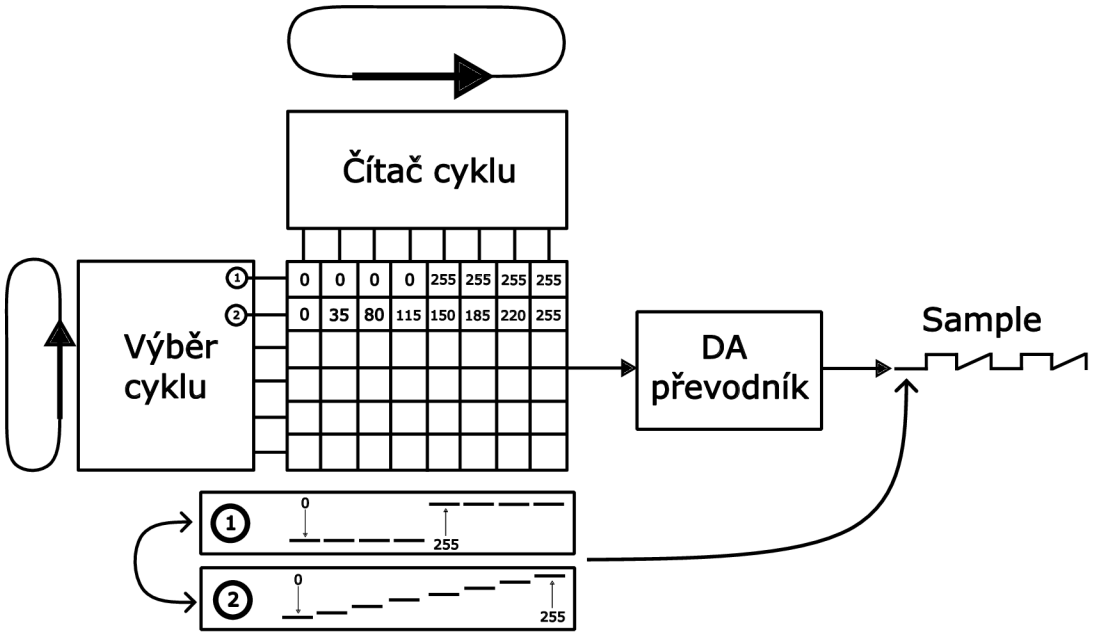


Obr. 1.6: Metoda *Wavecycle* způsobem *Single-cycle*.

Multi-cycle

Hlavní rozdíl oproti metodě *Single-cycle* je dynamická změna vyčítání cyklů z paměti. Pokud je k čítači přidána dynamická změna adresy při každém průchodu cyklu, vznikne tak nový tvarový průběh a výstupní signál o poloviční frekvenci. Pokud by čítač zvolil v prvním cyklu průběh na adrese 1 (např. sinusový průběh) a v druhém cyklu by změnil pozici adresy na 2. (např. s obdélníkovým průběhem), výsledný signál by měl dvojnásobnou periodu, která by obsahovala v první polovině nejprve sinusový průběh a následně v druhé polovině obdélníkový průběhu.

Zde nastává problém s omezením počtu možných zvolených průběhů za sebou. V případě, že by se celý signál skládal například z 16 různých cyklů, potom by měl výsledný signál kmitočet o 4 oktávy níže. Dalším problémem je délka jedné periody, která musí být vždy mocnina 2, jinak by výstupní signál nekorespondoval s požadovaným tónem ani na úrovni oktáv (pokud by chtěl hráč zahrát například komorní A4 a zvolil by 1. cyklus pilový průběh a další dva cykly ticho, ve výsledku by dostal tón o oktávu a kvintu nižší, tedy tón C5). [1] [2]



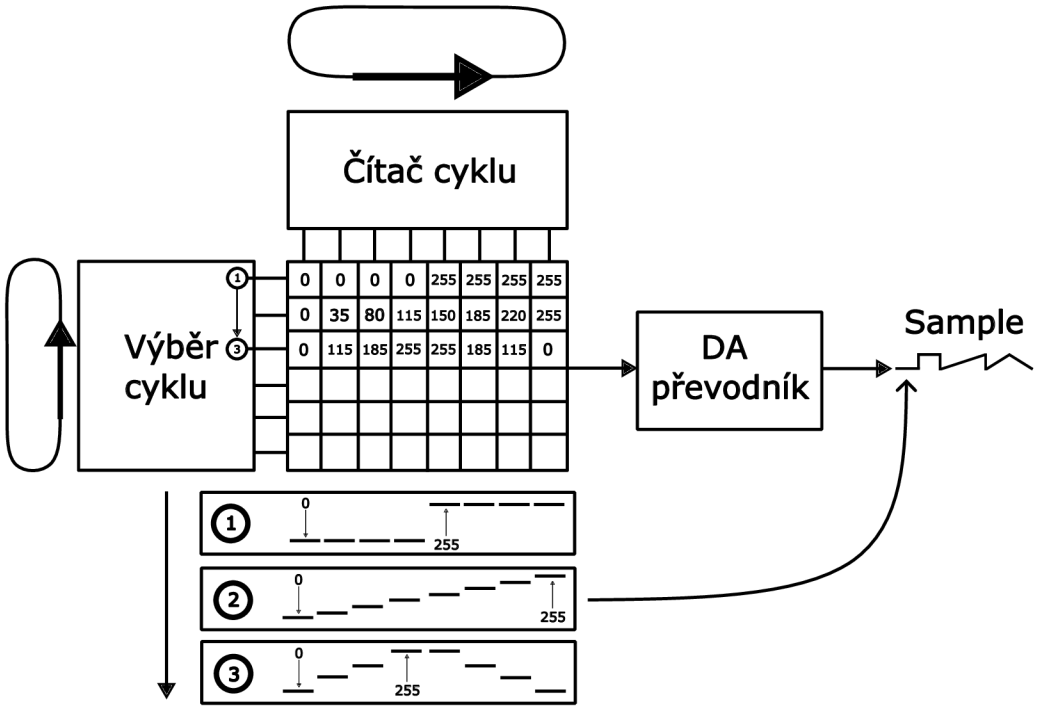
Obr. 1.7: Metoda *Wavecycle* způsobem *Multi-cycle*.

1.3.2 Wavetable

Wavetable metoda velice úzce připomíná dříve zmíněnou metodu *Multi-cycle*, avšak jejich hlavní rozdíl je v možnosti vyčítání jednotlivých cyklů po sobě. Zatímco v metodě *Multi-cycle* se sekvence cyklů opakovala nepřetržitě a nešlo měnit adresy jednotlivých opakujících se cyklů (vždy se opakovalo n cyklů podle pořadí, které zvolil uživatel), *Wavetable* tuto možnost obsahuje avšak umožňuje volit jiné adresy cyklus po cyklu. Dva obvyklé způsoby procházení skrz tabulku cyklů jsou *Swept* a *Random-access*. [2]

Swept

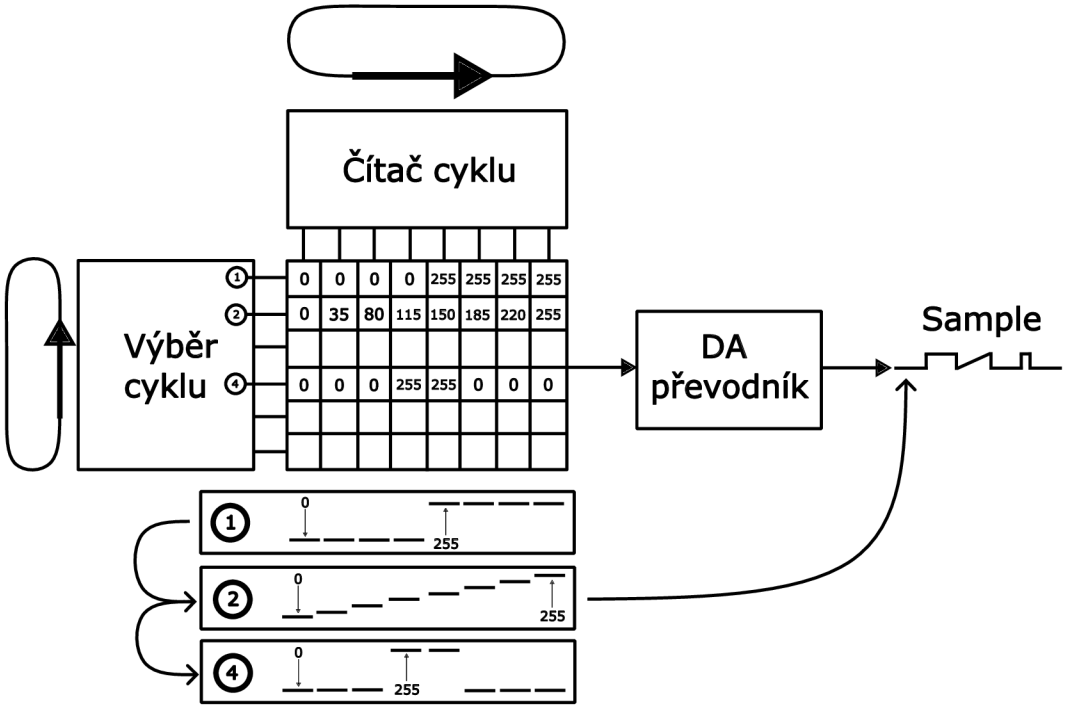
Způsob procházení *Swept* je založený na průchodu ukazatele hodnotami několika po sobě jdoucích cyklů. Po průchodu ukazatele všemi hodnotami se vrací na počáteční zvolený cyklus a proces se opakuje pokud uživatel nezadá jinak. Pro tento způsob čtení se obvykle vytváří speciální *Look-up table*, který má ve svém repertoáru sadu cyklů, které jsou svojí barvou/spektrálním zastoupením podobné nebo návazné, a zařídí tak hladší průběh výstupního signálu.[2]



Obr. 1.8: Vyčítání z pamětí metodou *Wavetable* způsobem *Swept*.

Random-access

Tento způsob procházení tabulkou cyklů je v podstatě pouze speciálním případem *Swept* procházení, kdy je přiřazen ukazatel na adresu libovolného cyklu v tabulce. Zadáním více ukazatelů lze vyčíst vybraný počet cyklů po sobě. Pokud by ukazatelé zůstali fixně na jedné adrese po celou dobu čtení, tak *Wavetable* vyčítání bude mít charakter *Multi-cycle*. Uživatel však může ukazatelům libovolně měnit adresy cyklů v průběhu přehrávání, čímž vzniká právě popisovaný způsob vyčítání *Random-access* typický pro *Wavetable* metodu. [2]



Obr. 1.9: Vyčítání z pamětí metodou *Wavetable* způsobem *Random-access*.

2 Demonstrační skripty hybridních syntéz

V následující kapitole jsou představeny skripty vytvořené v prostředí Matlab demonstrující základní typy generátorů zvukových signálů v hybridních syntezátorech společně s jednotlivými mezikroky při tvorbě výsledných signálů. V prvních kapitolách jsou představeny generátory využívající hodinového impulsu. Dále jsou rozebrány skripty generující signál tabulkovými metodami. Všechny skripty byly napsány v prostředí Matlab ve verzi R2023b s Update 3.

2.1 Generátor hodinových impulsů

Základem vytvořených generátorů byl právě generátor obdélníkového průběhu, který simuluje hodinový impuls tvořený *Master Oscillatorem*. Samotná aplikace zajišťuje jednoduché generování a ukládání 0 a 1 do výstupního vektoru, který lze dále použít pro generování ostatních tvarových průběhů. Výstupní hodinový impuls je generován na základě volby počtu bitů čítače (simulace počtu KO v obvodu). Počet vzorků je determinován počtem bitů a to podle vzorce

$$\texttt{vzorky} = 2^{\texttt{bity}}. \tag{2.1}$$

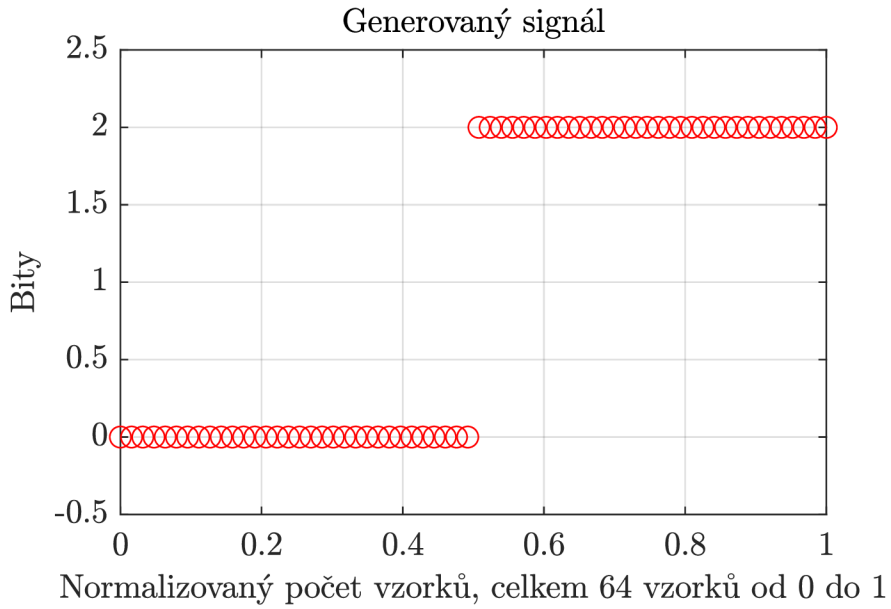
Tímto je zajištěna výpočetní rychlost a především korektní velikost výstupního vektoru hodinového impulsu pro výpočty v dalších funkčních skriptech. Z tohoto vzorce lze odvodit i potřebný minimální kmitočet f_{clock} generátoru hodinového impulsu pro zvolený kmitočet výsledného signálu f

$$f_{clock} = 2^{\texttt{bity}} \cdot f. \tag{2.2}$$

Skript dále poskytuje uživateli možnost měnit velikost amplitudy (maximální hodnoty bitů) a přidávat zpoždění hodinového impulsu o požadovaný počet vzorků pomocí Matlab funkce `circshift`. Zpoždění o x vzorků lze realizovat jednoduchým matematickým výpočtem $\texttt{zpoždění} = \frac{x}{\texttt{vzorky}}$. Hlavní logika kódu je zobrazena ve výpisu 2.1.

2.2 Dělička kmitočtů

Po realizaci hodinového impulsu bylo zapotřebí vytvořit děličku kmitočtu, která by hodinový impuls dělila a umožnila tak vzniknu dílčích signálů uzpůsobených k součtu na OZ. Dělička se skládá ze stejného počtu čísel, jako je množství bitů čítače. Hodnoty čísel děličky jsou zapisovány pro jednotlivé dělené signály tak, aby se



Obr. 2.1: Generovaný hodinový impuls z 6 bitů o 64 vzorcích.

Výpis 2.1: Hlavní programová logika MG_2.m.

```
%% Ulozeni hodnot signalu do vektoru
% prochazi cely vektor signalu az do posledniho vzorku
for k = 1:length(vzorky)
    if k>length(vzorky)/2
        signal(:,k) = A*1;
    elseif k<length(vzorky)/2
        signal(:,k) = 0;
    else
        signal(:,k) = 0;
    end
end

%% Posunuti signalu o zpozdeni
if zpozdeni ~= 0
    signal = circshift(signal, ...
    [0, round(zpozdeni*length(vzorky))]);
    % kruhove posunuti o dany pocet vzorku (zpozdeni)
end
```

poslední vygenerovaný signál svojí periodou rovnal periodě výsledného signálu generátoru `MG_2.m`. To lze zařídit postupným krokováním všech bitů podle stejné rovnice, která determinuje počet vzorků v generátoru (viz rovnice 2.1). Pokud má generátor např. 5bitový čítač, bude dělička generovat 5 dělících hodnot s poslední hodnotou $2^5 = 32$, celkový vektor tak bude vypadat následovně `[1 2 4 8 16 32]`. Dělící hodnoty jsou uloženy ve vektoru `div`, který se následně používá v dalších skriptech.[1] Ideální demonstrací výstupních signálů z děličky je čítač generátoru Walshových funkcí (viz graf 2.2).

2.2.1 Generátor Walshových funkcí

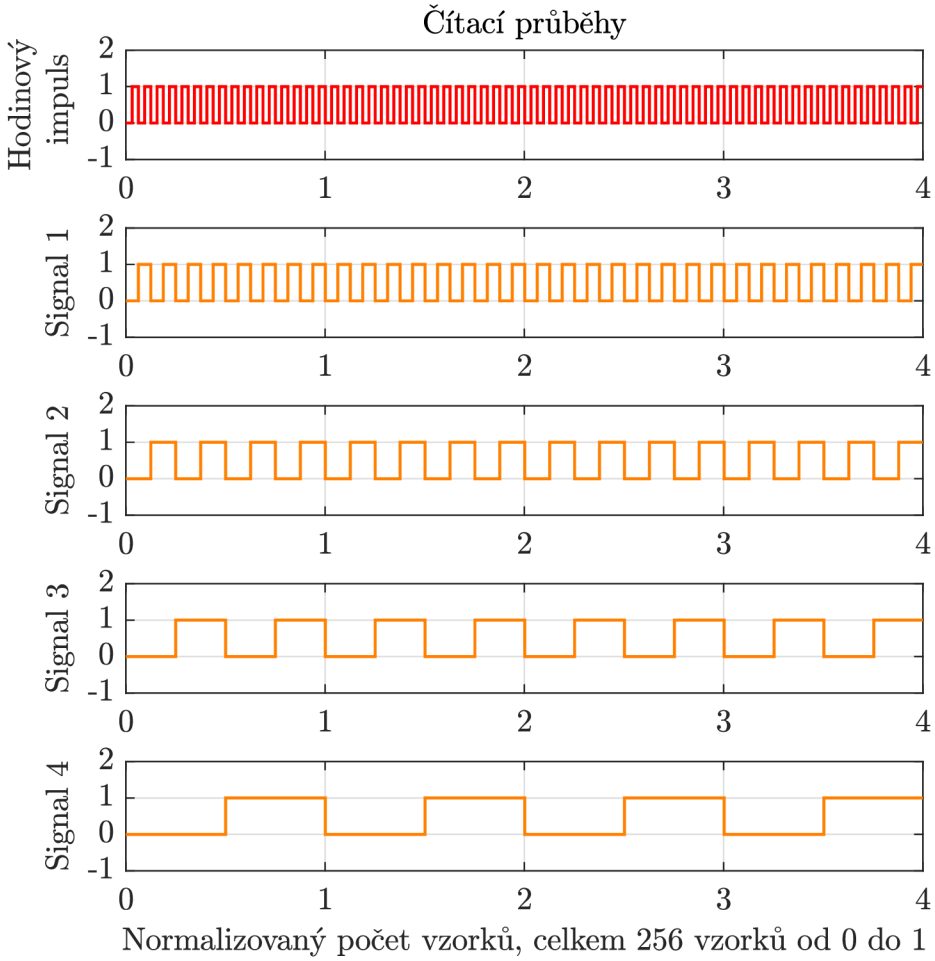
Generátor pracuje na součtu jednoduchých čítacích signálů bez zpoždění po průchodu děličkou. Výsledný signál tvoří tvarový průběh podobající se pilovému průběhu. Skript pracuje na principu vygenerování čítacích signálů pomocí generátoru `MG_2.m`. Do generátoru je poslán vektor lineárně rozložených vzorků `vz_perioda` (normalizovaných od 0 do 1) o celkovém počtu vzorků, který vychází z aktuální hodnoty děličky a počtu vzorků na periodu hodinového impulsu (takže např. signál vycházející ze 3. děličky bude mít třikrát více vzorků než perioda hodinového impulsu).

Po vygenerování se signály prodlužují na celkovou délku jedné periody výsledného signálu, což je zaručeno inverzní hodnotou děličky (`inv_div`), kterou se signál na vstupu dělil. Generované dílčí signály před sumací jsou zobrazeny v grafu 2.2. Posléze se signály ukládají do matice `Signaly` odkud jsou vyčítány pro sumaci. Výsledné signály po sumaci jsou uloženy v matici `sumSignaly` a replikovány na zvolený počet period uživatelem. Výsledné generované průběhy v každém kroku sumace lze vidět v grafu 2.3.

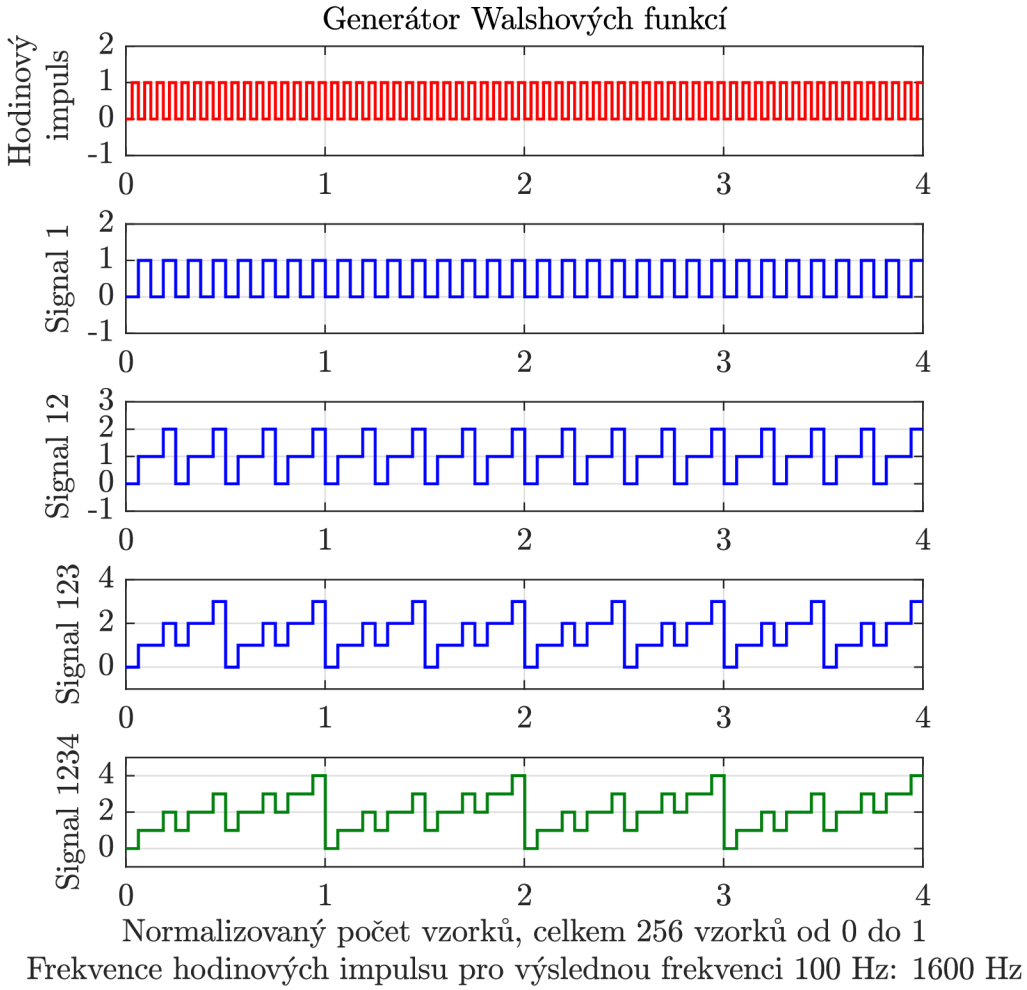
2.2.2 Generátor Walshových funkcí s KO typu D

Tento generátor je logikou podobný klasickému generátoru Walshových funkcí, jak byl popsán výše. Jeho hlavní rozdíl je však ve zpoždění jednotlivých součtových signálů. V demonstračním kódu 2.4 je zobrazena logika zpožďovacích prvků, které se následně posílají na vstup do generátoru `MG_2.m`. Rovnice pro proměnnou `krokVzorku` vyplývá z rozmezí mezi dvěma dílčími vzorky generovaného signálu stanoveným zlomkem, kde v čitateli je celkový počet bitů čítače a ve jmenovateli celkový počet vzorků (např. pro 5bitový čítač je krok mezi dvěma sousedními vzorky $\frac{5}{2^5} = 0{,}15625$).

[1] Kód děličky (viz výpis 2.2) je vždy součástí konkrétního skriptu. Nemá vlastní Matlab skript.



Obr. 2.2: Čítací průběhy po průchodu děličkou při generování Walshových funkcí.



Obr. 2.3: Výsledné kroky sumace Walshových funkcí.

Výpis 2.2: Vytvoření vektoru děličky kmitočtu.

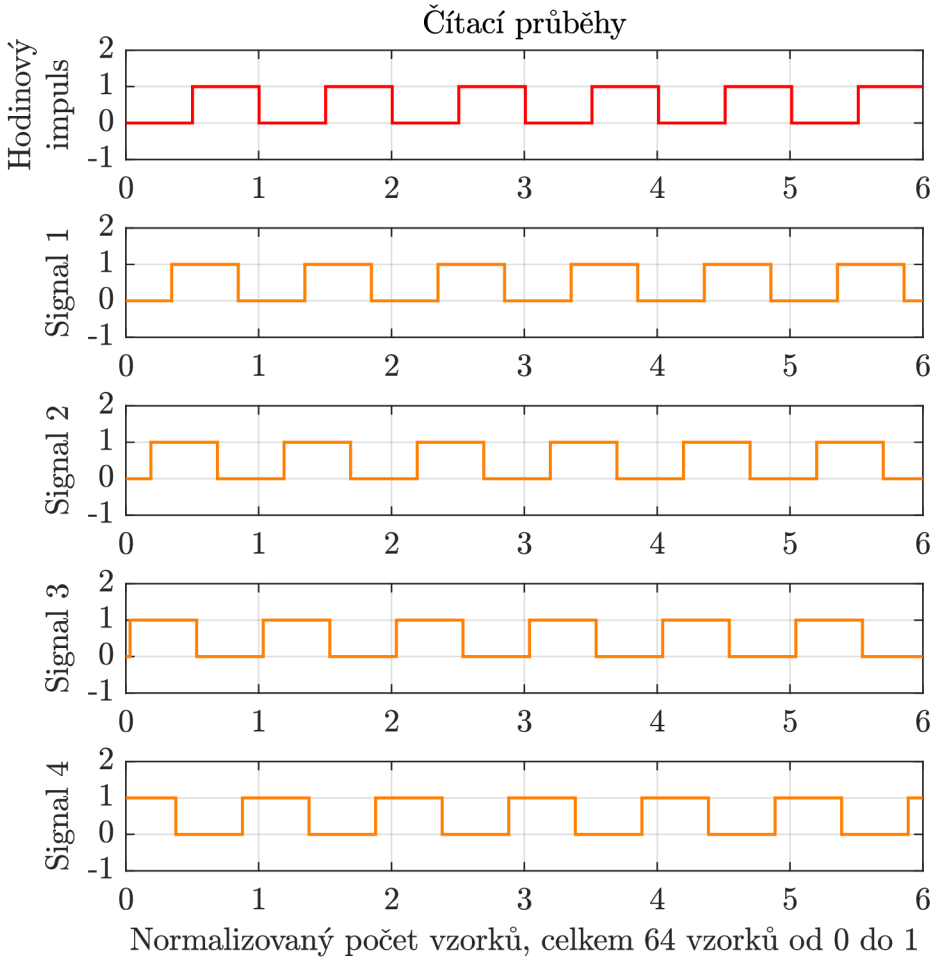
```
%% Vytvoreni vektoru div pro pouziti na delicce
div = zeros(1,pocet_bitu_citace+1); % initializace
   vektoru delicky
for p = 1:pocet_bitu_citace  % vycet pro vsechny bity
    if p == 1
            div(p) = 1;
            div(p+1) = 2^p; % zaruceni ulozeni 1 na
               pozici (1:1) ve vektoru div
            continue;
    end
    div(p+1) = 2^p;
end
```

Následně je tato hodnota vynásobena aktuálním indexem zpožďovací linky (simulace průchodu signálu skrz několik KO typu D) a 10 pro ideální zpoždění o deset vzorků s každým dalším krokem.

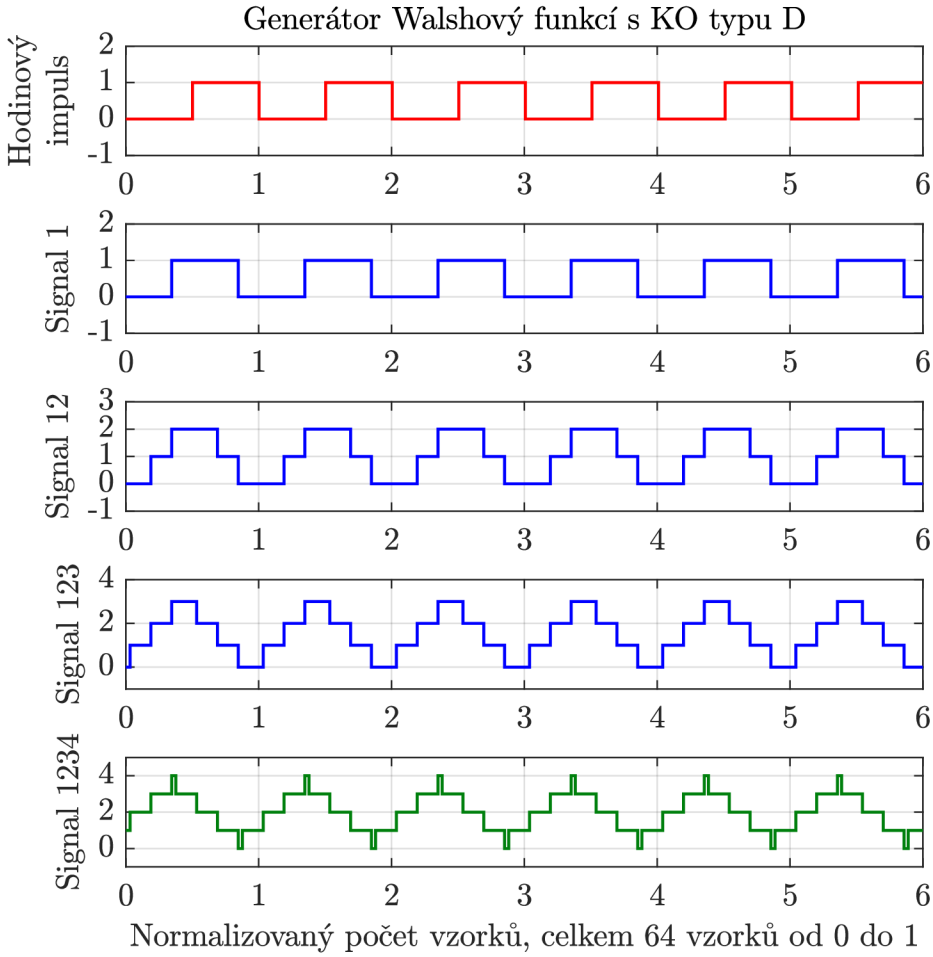
Výsledkem generování je signál aproximující trojúhelníkový tvar průběhu zobrazený v grafu 2.5. Skript zpoždění a jeho zasílání do generátoru `MG_2.m` je zobrazeno ve výpisech 2.4 a 2.5.

2.2.3 Generátor pilového průběhu

Pro lepší aproximaci pilového průběhu vznikl na základě Walshových funkcí bez zpoždění generátor pilového průběhu. V něm se změnila hodnota výchylky jednotlivých generovaných signálů tak, aby korespondovala v každém čítacím cyklu s aktuálním krokem děličky. Tedy pokud by měl čítač 5 bitů, budou výchylky postupně zvětšovány s krokem 2^n (n je index aktuální hodnoty děličky) a maximální hodnota výchylky posledního čítacího signálu bude $2^5 = 32$. Tento přístup zobrazený v grafu 2.6 zaručí ve sčítacím obvodu postupný lineární inkrement výchylky výsledného signálu, který je patrný z jednotlivých součtových signálů v grafu 2.7. Implementace této logiky úpravou skriptu Walshových funkcí je zobrazena ve výpisu 2.6.



Obr. 2.4: Čítací průběhy Walshových funkcí se zpožděním.



Obr. 2.5: Výsledné průběhy Walshových funkcí se zpožděním.

Výpis 2.3: Generování Walshových funkcí.

```
%% Generovani signalu Walshovych funkci

for k = 1:length(div)
    vz_perioda = linspace(0,1,vzorky_perioda*((vzorky/
       vzorky_perioda)*div(k)));
    % vektoru poctu vzorku pro dany signal (napr.:
       hodinovy impuls = 8 vzorku, 1. signal = 9*2 = 16,
       2. signal = 18*2 = 32...)

    signal = MG_2(1,vz_perioda,0);
    % generuje signal
    inv_div = div(length(div)-k+1);
    % invertovani vektoru delicky pro vektor duplikace
       signalu
    signal = repmat(signal,1,inv_div);
    Signaly(k,:) = signal;

    if k == 1
        sumSignaly(k,:) = Signaly(k,:);
        % ulozeni hodinoveho impulsu
    elseif k == 2
        sumSignaly(k,:) = Signaly(k,:);
    elseif k == length(div)
        sumSignaly(k,:) = signal + sumSignaly(k-1,:);
        % sumace vsech signalu do vysledneho vektoru
        sumSignaly = repmat(sumSignaly,1,pocet_period);
        % replikace pro dany pocet period
        Signaly = repmat(Signaly,1,pocet_period);
        % replikace pro dany pocet period
        vz_perioda = linspace(0,pocet_period,length(
           sumSignaly));
        % vytvoreni vysledneho vektoru poctu vzorku podle
            delky vyslednych signalu
    else
        sumSignaly(k,:) = signal + sumSignaly(k-1,:);
    end
end
```

Výpis 2.4: Zpoždění signálu.

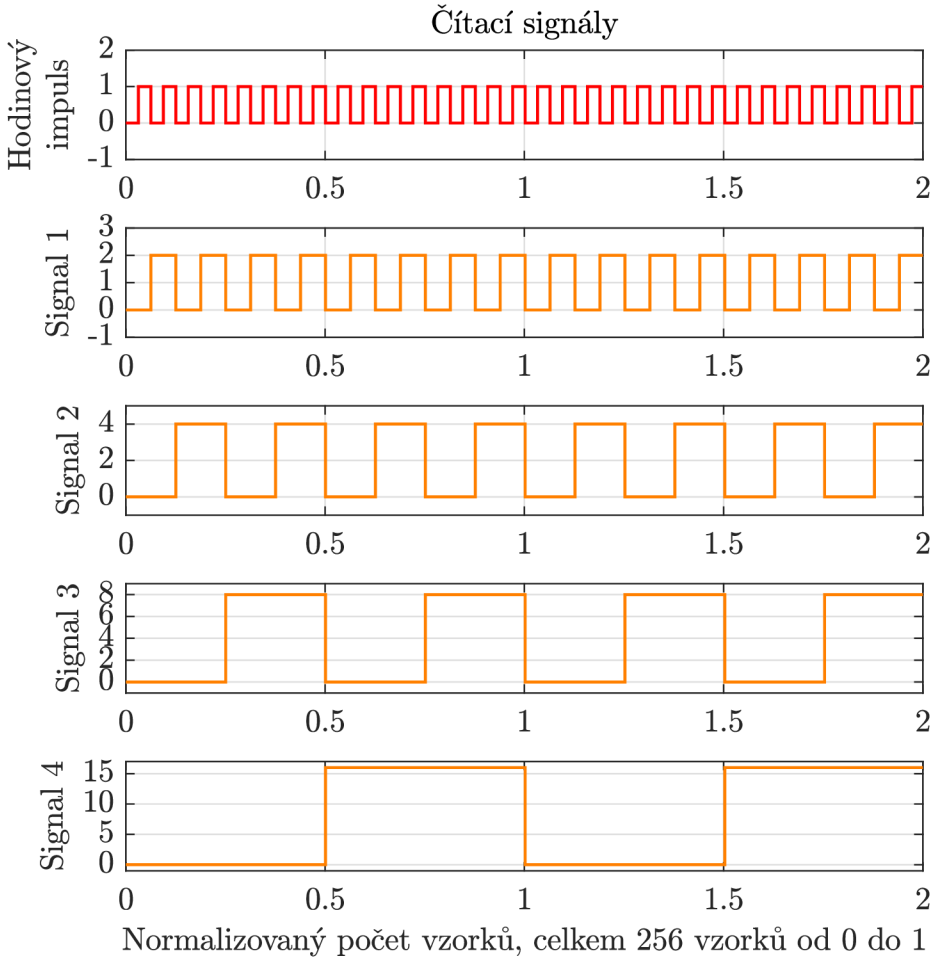
```
%% Zpozdeni
zpozdeni = zeros(1,pocet_zpozdeni);
for l = 0:pocet_zpozdeni % vytvoreni vektoru zpozdeni
    krokVzorku = l*10*pocet_bitu/vzorky; % krok zpozdeni
    zpozdeni(:,l+1) = krokVzorku; % ulozeni zpozdeni do
       vektoru zpozdeni
end
```

Výpis 2.5: Generování Walshových funkcí se zpožděním.

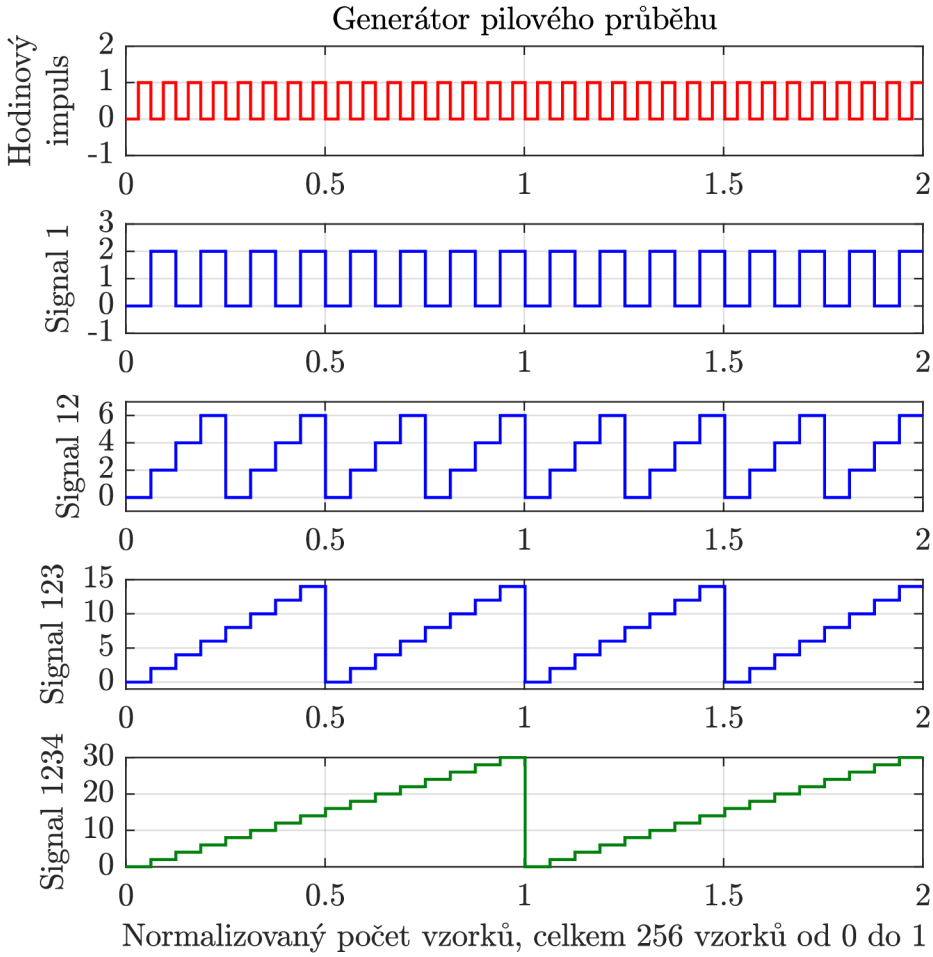
```
%% Generovani signalu
for k = 1:pocet_zpozdeni
    vz_perioda = linspace(0,1,vzorky);
    % vektoru poctu vzorku periody pro dany signal (napr
       .: hodinovy impuls = 9 vzorku, 1. signal = 9*2 =
       18, 2. signal = 18*2 = 32, 3. signal = 32*2 = 64,
       ...)

    signal = MG_2(1,vz_perioda,zpozdeni(k)); % generuje
       signal
    Signaly(k,:) = signal; % ulozeni signalu do matice
       Signaly
```

Výpis 2.6: Úprava Walshových funkcí pro generování pilového průběhu.

```
%% Generovani piloveho prubehu
    signal = MG_2(div(k),vz_perioda,zpozdeni); %
       generator signalu s inkrementem vychylky
    inv_div = div(length(div)-k+1); % inverzni index
       delicky pro prodlouzeni signalu na pozadovanou
       delku
    signal = repmat(signal,1,inv_div); % prodlouzeni
       signalu
    Signaly(k,:) = signal; % ulozeni signalu do matice
       Signaly
```



Obr. 2.6: Čítací signály generátoru pilového průběhu.



Obr. 2.7: Výsledné signály generátoru pilového průběhu.

2.3 Čtení z paměti

Další variantou generování zvukových signálů je vyčítání předem uložených dat z paměti, které jsou popsány v kap. 1.3. Tyto generátory využívají tzv. tabulek s daty určitých tvarových průběhů, které různým způsobem vyčítají, čímž vytváří nový tvarový průběh výsledného signálu. Pro demonstraci byly vytvořený dvě základní metody – *Wavecycle* a *Wavetable*. V neposlední řadě byl vytvořen skript `ROM_LFO.m`, který vytváří experimentálnější tvary průběhů za použití LFO.

Prvně byl vytvořen skript `Tabulka_1.m` obsahující několik generátorů datových tabulek (matic signálů), ze kterých jsou vybrány a následně vyčítány referenční signály. Tyto signály jsou generovány interními funkcemi Matlabu textttsin, `sawtooth` a `square`. Tento skript přijímá na vstupu index zvolené tabulky, vektor počtu bitů a vektor vzorků.

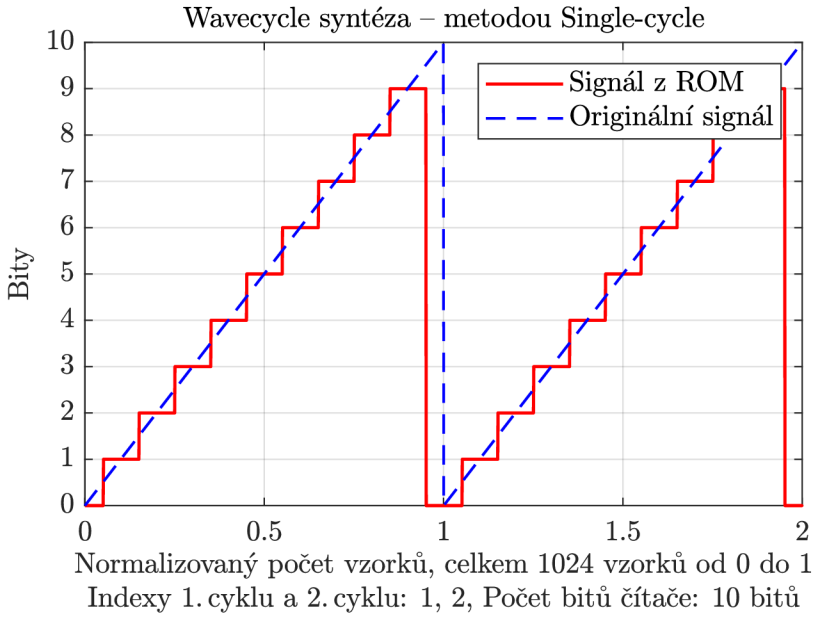
Pro zobrazení diskrétních hodnot dat, jak tomu je u hybridních syntezátorů, na místo plynulých průběhů, které generuje Matlab, byl vytvořen algoritmus pro vyčítání úrovní překročených bitů. Při překročení hodnoty daného bitu v aktuálním vzorku referenčním signálem se do výstupního vektoru vypíše hodnota překročeného bitu. Pokud však referenční signál v aktuálním vzorku nepřekročil větší resp. menší hodnotu bitu než byla poslední překročená hodnota, ukládá se do výstupního vektoru poslední překročená hodnota bitu. Tento algoritmus je zobrazen ve výpisu 2.7.

2.3.1 Wavecycle

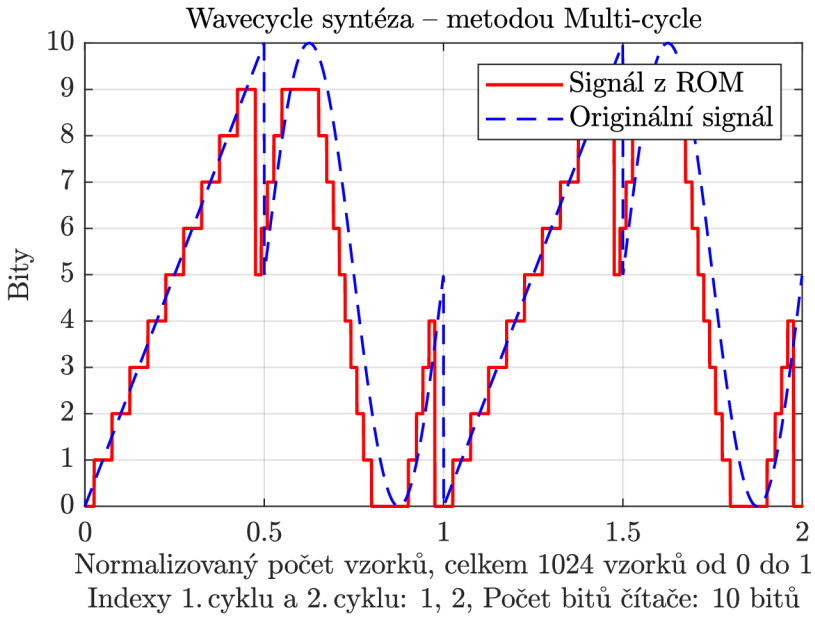
Demonstrování syntézy *Wavecycle* bylo rozděleno na dva základní způsoby vyčítání dat a to na *Single-cycle* a *Multi-cycle*. V první zmíněné metodě se vypisuje pouze první vybraný cyklus ze zvolené tabulky (princip *DCO*). Druhý způsob vyčítání využívá integrované funkce `cat`, která první a druhý vektor vybraných cyklů spojí za sebe do společného výstupního vektoru. Implementace skriptu je zobrazena ve výpisu 2.8. Výsledkem jsou v obou případech vektory referenčního signálu, které jsou poslány do předchozího algoritmu zápisu dat. Výstupní průběhy pro jednotlivé způsoby vyčítání jsou zobrazeny v grafech 2.8 a 2.9.

Výpis 2.7: Algoritmus simulující vyčítání z paměti v hybridních syntezátorech.

```
%% Algoritmus zapisovani
y = zeros(size(vektor_signalu)); % inicializace
   vystupniho vektoru
posledni_prekroceny_bit = vektor_bitu(1); % pocatecni
   hodnota posledniho prekroceneho bitu (0)

for i = 1:length(vektor_signalu)
    data = vektor_signalu(i); % hodnota signalu pro dany
       vzorek

    % porovnani signalu s hodnotami vektoru bitu
    index_prekroceneho_bitu = find(data >= vektor_bitu,
       1, 'last'); % nalezeni posledniho prekroceneho
       bitu

    if isempty(index_prekroceneho_bitu) % pokud index
       nenasel novy prekroceny bit, zapise se na pozici
       posledni prekroceny bit
        y(i) = posledni_prekroceny_bit;
    else
        y(i) = vektor_bitu(index_prekroceneho_bitu);
        posledni_prekroceny_bit = vektor_bitu(
           index_prekroceneho_bitu);
    end
end
signal = y; % vysledny signal
```



Obr. 2.8: Zobrazení 2 period syntézy *Wavecycle* – způsobem *Single-cycle*.



Obr. 2.9: Zobrazení 2 period syntézy *Wavecycle* – způsobem *Multi-cycle*.

Výpis 2.8: Ukládání referenčního signálu metodami *Single-cycle* a *Multi-cycle*.

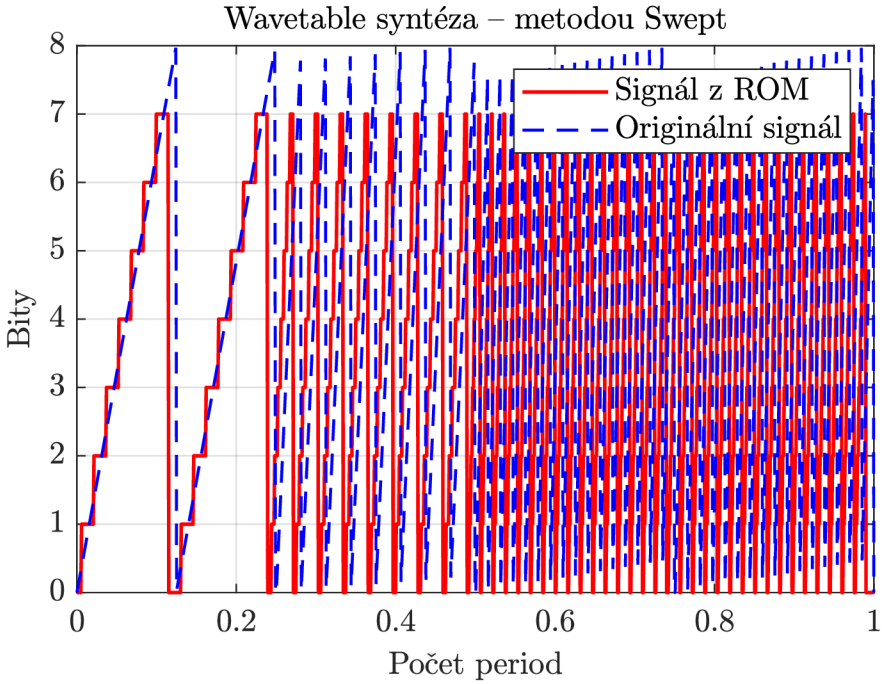
```
%% Generovani a vyber cyklu (Tabulka)
matice_signalu = Tabulka_1(tabulka, vektor_bitu,
   vektor_vzorku);
if strcmp(typ,'Single-cycle')
    vektor_signalu = matice_signalu(cyklus1,:);
elseif strcmp(typ,'Multi-cycle')
    vektor_signalu = matice_signalu(cyklus1,:);
    vektor_signalu = cat(2,vektor_signalu,matice_signalu(
       cyklus2,:));
end
```

2.3.2 Wavetable

Čtení metodou *Wavetable* je demonstrováno způsoby *Swept* a *Random-access*. *Swept* způsob funguje na principu vyčtení posloupnosti po sobě jdoucích cyklů z tabulky, zatímco způsob *Random-access* pracuje s vybraným počtem cyklů poskládaných v tabulce náhodně. Výstupem je referenční signál pokračující do algoritmu pro demonstraci hybridní syntézy. Skript je připraven tak, aby si uživatel mohl zvolit, kolik cyklů bude z tabulky vyčteno, a po spuštění je proveden výčet na základě zvoleného způsobu. Oba způsoby přitom využívají dříve zmíněnou funkci `cat`, která spojuje zadané vektory za sebe. Konkrétní implementace je zobrazena ve výpisu 2.9.

Swept

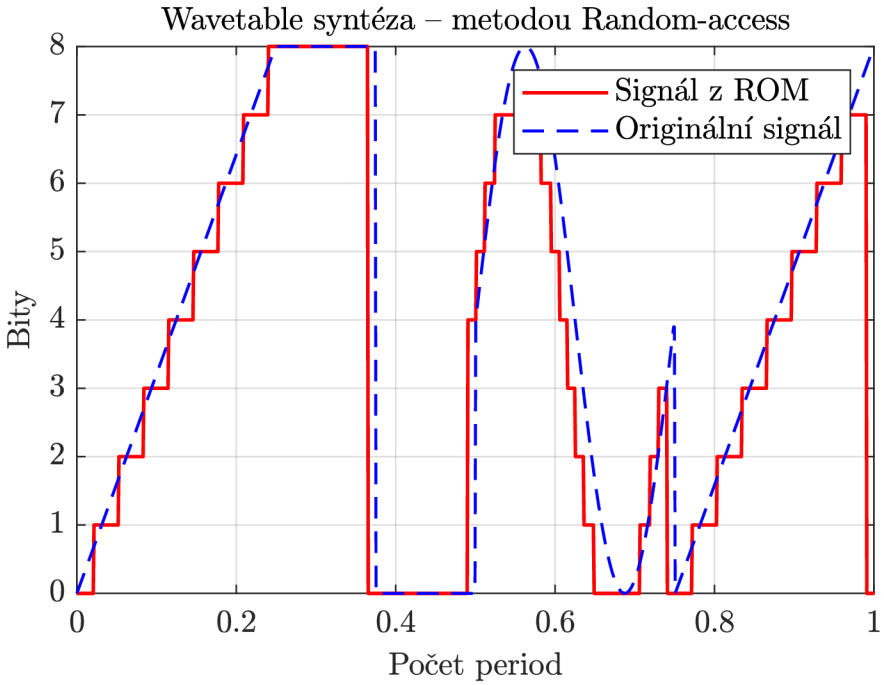
Způsob *Swept* zapisuje do výsledného vektoru signálu s každou iterací for cyklu referenční průběh vyčtený ze zvolené tabulky. V první iteraci vloží do výstupního vektoru první cyklus dané tabulky. Následně pokračuje s přidáváním jednotlivých referenčních dat navazujících cyklů za první cyklus až do indexu posledního cyklu, který je určen proměnnou `pocet_cyklu`. Výstupní vektor signálu je poté použit jako referenční signál v algoritmu simulace hybridní syntézy. Výsledek je zobrazen v grafu 2.10 (v tomto případě byly vybrány 4 cykly z tabulky).



Obr. 2.10: Syntéza *Wavetable* – způsobem *Swept*.

Random-access

Pro vyčítání náhodných cyklů z vybrané tabulky je použit způsob *Random-access*. První cyklus je opět uložen na začátek vektoru referenčního signálu. Následně jsou za tato data v každé iteraci for cyklu přidávána data cyklů s náhodným indexem. Výsledný počet cyklů, který obsahuje referenční signál, je dán proměnnou `pocet_cyklu`. Po průchodu referenčního signálu algoritmem simulace hybridního syntezátoru je výsledkem např. graf s průběhy 2.11.

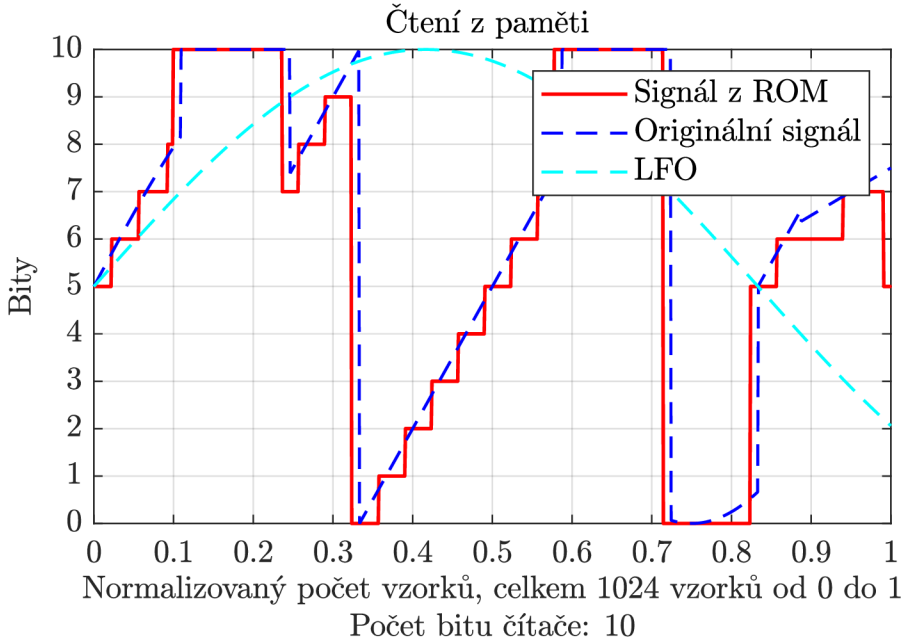


Obr. 2.11: Syntéza *Wavetable* – způsobem *Random-access*.

2.3.3 Čtení z paměti za pomoci LFO

Pro variabilnější změnu tvaru čteného průběhu z tabulky byl vytvořen skript, který umožňuje vyčítání dat bez ohledu na aktuální pozici ve čtecím cyklu. Pokud jsou například vyčítána data cyklu A na 10. pozici ze 32 a okamžitá hodnota LFO se v tento moment dostane nad, resp. pod vybranou prahovou mez, skokově se změní čtený cyklus na cyklus B v dané pozici, tedy 10. pozice ze 32 v cyklu B. Tímto lze měnit barvu výsledného signálu i mimo omezení vyčítání vždy celého cyklu.

Ve skriptu lze volit mezi dvěma průběhy LFO (sinusový a pilový skrze proměnnou `typ_metody`) a délku periody skrze proměnnou `perLF`. Tím lze ovlivnit rychlost změny mezi jednotlivými čítanými cykly. Dále je implementována logika přepínání mezi cykly podle okamžité hodnoty LFO, která definuje různé množiny hodnot v rozsahu výchylky LFO, a to od -1 do 1, které může signál LFO nabývat. Použití této logiky, včetně generování LFO, je možné vidět ve výpisu 2.10. Výsledné průběhy jsou zobrazeny v grafu 2.12.



Obr. 2.12: Výsledný signál závislý na okamžitých hodnotách LFO.

Výpis 2.9: Ukládání referenčního signálu metodami *Swept* a *Random-access*.

```
%% Vycitani z tabulky zpusobem Swept nebo Random-access
if strcmp(typ_metody,'Swept')
    for p = 1:pocet_cyklu
        if p == 1
            vektor_signalu = matice_signalu(p,:);
        elseif p == pocet_cyklu
            vektor_signalu = cat(2,vektor_signalu,
               matice_signalu(p,:));
        else
            vektor_signalu = cat(2,vektor_signalu,
               matice_signalu(p+1,:));
        end
    end
elseif strcmp(typ_metody,'Random-access')
    for p = 1:pocet_cyklu
        if p == 1
            vektor_signalu = matice_signalu(p,:);
        else
            vektor_signalu = cat(2,vektor_signalu,
               matice_signalu(randi([1,10],1),:)); %
               pridani vektoru signalu za stavajici
               vektor
        end
    end
end
```

Výpis 2.10: Implementace logiky změny čteného cyklu pomocí LFO.

```
%% LFO generovani
if typeLF == 'sin'
    LFO(1,:) = sin(2*pi*vzorky*perLF);
elseif typeLF == 'saw'
    LFO(1,:) = sawtooth(2*pi*vzorky*perLF);
else
    LFO(1,:) = sin(2*pi*vzorky*perLF);
end

%% Prochazeni Wavetable pomoci LFO
for l = 1:length(LFO) % skok skrz wavetable pole podle
   hodnoty prekroceni LFO
    if LFO(1,l) >= 0 && LFO(1,l) < 0.2
        vektor_signalu(1,l) = Signaly(1,l);
    elseif LFO(1,l) >= 0.2 && LFO(1,l) < 0.4
        vektor_signalu(1,l) = Signaly(2,l);
        % atd.
    elseif LFO(1,l) <= -0.8 && LFO(1,l) >= -1
        vektor_signalu(1,l) = Signaly(5,l);
    end
```

3 Realizace hybridního syntezátoru v prostředí JUCE

Výsledný syntezátor byl vytvořen v jazyce C++ uvnitř vývojového prostředí Microsoft Visual Studio 2022 za pomoci frameworku JUCE v7.0.9. V této kapitole je rozebráno základní rozložení prostředí JUCE, tříd a metod obsažených v šabloně tohoto frameworku a jejich hlavní užití. Větší část této sekce je věnována vytvořenému kódu, který přepisuje předchozí skripty generátorů z kapitoly 2 do jazyka C++ a využívá interní třídy frameworku JUCE. Na začátku kapitoly je širší popis architektury programu následující konkrétnějším popisem individuálních tříd. Je důležité upozornit, že v textu se často objevuje záměna mezi českou a anglickou verzí slova *buffer*, jehož českým ekvivalentem je vyrovnávací paměť. V kódu lze také často narazit na třídu *juce::dsp::AudioBlock*, tato třída realizuje odkaz na základní *buffer* a lze ji vnímat jako alias této paměti.

3.1 Prostředí JUCE

Jules' Utility Class Extensions (zkráceně JUCE) je volně dostupný a editovatelný (*open-source*) aplikační framework poprvé vydaný v roce 2003 autorem Julianem Storerem. Jednalo se původně o část zpracovávající grafickou a zvukovou část v DAW s názvem Traktion, které se do dnešní doby zachovalo pod názvem Waveform. JUCE má podporu pro multiplatformní zvuková zařízení (například CoreAudio, ASIO, ALSA, JACK, WASAPI, DirectSound) včetně MIDI, vestavěnou kompatibilitu pro běžné formáty zvukových souborů (například WAV, AIFF, FLAC, MP3 a Vorbis) a také tzv.*wrappery*, tedy zařízení sloužící pro vytváření různých typů zvukových zásuvných modulů, například efektů a nástrojů VST. To vedlo k jeho širokému využití právě v komunitě vývojářů zvukových aplikací. [7]

3.1.1 Projucer

Pro organizaci projektů a nastavení základních pomocných tříd a struktur před samotným vývojem aplikace využívá JUCE podpůrný program Projucer. V tomto uživatelském prostředí si může programátor nastavit požadovaný typ vyvíjeného softwaru, hlavní používané moduly z JUCE, výstupní formát samotného programu (VST3, AAX apod.) a vývojové prostředí (např. Apple XCode, Microsoft Visual Studio a další). Projucer následně vytvoří základní šablonu pro dané vývojové prostředí s vybranými moduly. [8]

3.1.2 Základní rozložení šablony

Šablona se skládá ze dvou základních tříd – *PluginProcessor.cpp* a *PluginEditor.cpp*. Jak již název tříd může napovědět, v třídě *PluginProcessor.cpp* je řešený problém zpracování dat, včetně procesů generování, ukládání a vyčítání. Data jsou zde ve formě vzorků uchována do vyrovnávací paměti (tzv. *bufferu*), který je dále zpracováván. Třída *PluginEditor.cpp* oproti tomu řeší vzhled grafického uživatelského rozhraní (též *Graphical User Interface* – *GUI*) a spojení části procesoru s uživatelským prostředím, kde je uživatel schopen měnit parametry dané aplikace. Níže jsou bodově popsány základní vlastnosti využitých metod v jednotlivých třídách.

PluginProcessor.cpp

- **prepareToPlay()** – metoda zajišťující předání parametrů vzorkovacího kmitočtu, počtu vzorků ve vyrovnávací paměti a dalších parametrů (např. počet zpracovaných kanálů daného zvukového signálu) před spuštěním aplikace.
- **ProcessBlock()** – *callback* metoda probíhající průběžně při chodu aplikace, jsou zde zpracovávána data z vyrovnávací paměti.
- **createEditor()** – metoda generující uživatelské rozhraní.
- **setStateInformation** – ukládání hodnot parametrů do paměti, zabránění jejich smazání při uzavření plug-in modulu v hostitelské aplikaci.
- **getStateInformation** – načtení hodnot parametrů z paměti.

PluginEditor.cpp

- **paint()** – metoda sloužící k vykreslení *GUI* na obrazovku.
- **resized()** – metoda k rozmístění uživatelských prvků v okně.

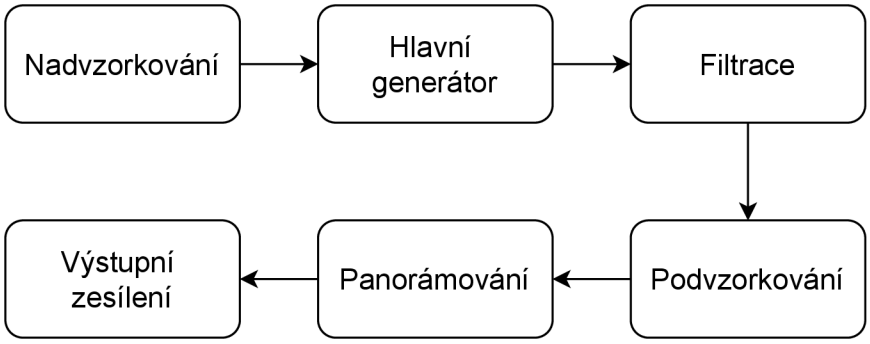
Pro účely práce byla použita především třída *PluginProcessor.cpp*, kde byly vytvořeny všechny důležité podtřídy a metody pracující s vygenerovanými vzorky. Samotné *GUI* je vytvořeno pomocí volání metody `createEditor()` s návratovou hodnotou `new juce::GenericAudioProcessorEditor(*this)`, která vygeneruje základní list po sobě jdoucích parametrů definovaných v metodě `createParams()` skrze třídu *AudioProcesorValueTreeState* (viz kapitola 3.2.8) [9].

Výpis 3.1: Vytvoření základního uživatelského rozhraní.

```
1 juce::AudioProcessorEditor* Hybrid_synthAudioProcessor::createEditor()
2 {
3     //return new Hybrid_synthAudioProcessorEditor (*this);
4     return new juce::GenericAudioProcessorEditor(*this);
5 }
```

3.2 Architektura signálového procesu

Pro realizaci autentické logiky hybridních syntezátorů, tedy čtení vzorků z paměti o určité pevné délce a následné cyklické opakování tohoto průběhu, byly navrženy tři základní třídy řešící problém generování výstupního signálu. Konkrétně se jedná o třídy *Waveform.cpp*, *Synth.cpp* a *Oscillator.cpp*, které společně v diagramu 3.1 tvoří blok *Hlavní generátor*. Detailnější propojení mezi těmito třídami je patrné z diagramu 3.2 a je rozebráno v navazujících kapitolách 3.2.3, 3.2.4 a 3.2.5. Dále stojí v cestě signálu filtrace, která probíhá v oddělené třídě *Filters* a její základní funkce a metody jsou popsány v kapitole 3.2.7. Signál posléze prochází zpětným podvzorkováním, kterému na začátku celého procesu oponovalo nadvzorkování vstupní vyrovnávací paměti. Těsně před výstupem audio signálu je signál panorámován a zesílen (viz kapitola 3.2.2).



Obr. 3.1: Blokové schéma celkového průběhu signálu.

3.2.1 Převzorkování

Na začátku zpracování je možnost převzorkování vstupní vyrovnávací paměti. Převzorkování je realizováno ve dvou fázích, a to nadvzorkováním před generováním a filtrováním signálu a následným zpětným podvzorkováním na hodnotu vzorkovacího kmitočtu hostitelské aplikace. Pro převzorkování byla použita třída *Oversampling* z knihoven JUCE. Třída podporuje několik stupňů nadvzorkování, a to konkrétně 2, 4, 8 a 16násobné nadvzorkování oproti vstupní vzorkovací frekvenci. Všechny tyto možnosti byly vloženy do konstruktoru třídy *PluginProcessor* s počtem kanálů pro zpracování a antialiasingovým filtrem typu IIR (v tomto případě se jednalo o typ filterHalfBandPolyphaseIIR), který zaručuje menší latenci oproti jiným nabízeným možnostem (viz dokumentace [10]) za cenu vyššího fázového zkreslení. Následná příprava převzorkování je provedena v metodě `prepareToPlay()` skrz volání metodou `initProcessing()`, které je zaslán počet vzorků na jednotku vyrovnávací paměti (angl. SamplesPerBlock). Aplikace převzorkování probíhá opakovaně pro

každý *buffer* v metodě `ProcessBlock()` na základě zvolené hodnoty parametru oversamplingFactor a je tak možné měnit převzorkování v rámci uživatelského rozhraní pomocí metody `processSamplesUp()`. Po průchodu *Hlavním generátorem* a *Filtrací* je následně provedeno podvzorkování pomocí metody `processSamplesUp()`.

3.2.2 Panorámování a zesílení

Po zpětném podvzorkování je provedeno panorámování za pomoci interní třídy *Panner*, která umožňuje několik variant pro tzv. *Panning law*, tedy možností pro nastavení přechodů panorámování z jednoho kanálu do druhého ve stereofonní bázi. V tomto případě se zvolila možnost balanced, která pracuje na principu zesilování jednoho ze stereofonních kanálu s úměrným zeslabování druhého kanálu a obráceně [11]. Inicializace panorámy probíhá opět v metodě `prepareToPlay()` za pomoci metod `prepare()` a `setRule()`, která je nastavena na již zmíněnou možnost balanced. Výstupní panoráma je ovlivňována v metodě `ProcessBlock()` voláním metody `setPan()` [12]. Posléze bylo aplikováno výstupní zesílení signálu, které je realizované pomocí metody samotného *bufferu* – `applyGainRamp()`. Této metodě je zaslána informace o indexu prvního a posledního vzorku v aktuálně zpracovávaném *bufferu*, hodnota parametru zesílení před začátkem zpracování, tedy hodnota zesílení před změnou parametru a konečná hodnota zesílení (aktuálně zvolená hodnota). Tímto je vytvořeno náběžné zesílí v rámci jednoho *bufferu* a je tak možné vytvářet jednoduché přechody hlasitostí bez problému se skokovou změnou hlasitosti [13].

3.2.3 Třída *Oscillator*

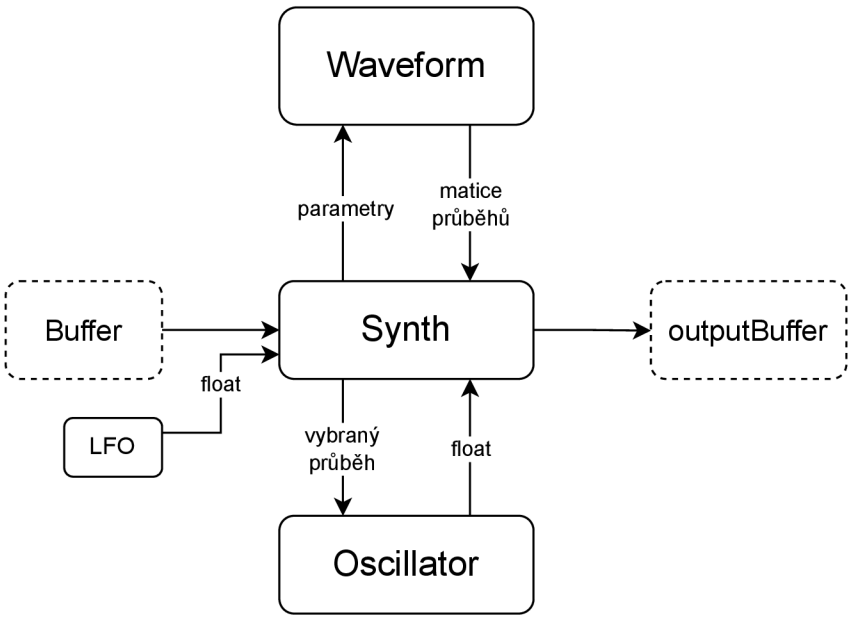
Prvním prvkem v řadě pro popis struktury *Hlavního generátoru*[1] zvukového signálu je třída *Oscillator*. Logika třídy je postavena na vyčítání vzorků z vektoru a byla založená na kódu z článku [14]. Vzhledem k propracovanosti kódu byla této třídě přidána pouze další metoda, která mění vyčítaný průběh zvolený uživatelem. Nejdůležitější funkce této třídy je implementována v metodě `loopingOverWaveshape()` (zobrazené ve výpisu 3.2), kde se pomocí operace *modulo* získává aktuální index vzorku.

Díky práci v digitální doméně lze výpočet indexu aktuálního vzorku založit na vzorkovací frekvenci hostitelské aplikace. Pokud máme v reálné doméně spojitý sinusový signál definovaný jako

$$s(t) = A \cdot \sin(2\pi f t + \phi), \tag{3.1}$$

lze odvodit, že pozice aktuálního vzorku v reálném čase bude závislá na čase t. Vzorkovací kmitočet udává počet vzorků za sekundu, tedy aktuální vzorek bude

[1] viz konkrétní propojení dílčích tříd *hlavního generátoru* ve schématu 3.2



Obr. 3.2: Blokové schéma *Hlavního generátoru.*

odpovídat indexu $n = f_s \cdot t$. Pokud dosadíme tuto rovnici do rovnice 3.1, dostaneme závislost sinusového průběhu v digitální doméně ve tvaru

$$s[n] = A \cdot \sin(2\pi f \frac{n}{f_s}). \tag{3.2}$$

Pokud bychom na fázi určitého periodického signálu hleděli jako na periodicky se opakující inkrement (např. pro sinusový signál by se jednalo o rozsah hodnot $<0;2\pi)$), poté lze tento rozsah dát do poměru s aktuální fází v daném čase, což je ekvivalentní s aktuálním vzorkem v poměru k délce celého vektoru průběhu. Taková rovnice by měla tvar

$$\frac{n}{L} = \frac{\phi}{2\pi}. \tag{3.3}$$

Po převedení celkové délky vektoru L dostaneme rovnici pro aktuální index vzorku

$$n = \frac{\phi L}{2\pi}. \tag{3.4}$$

Rovnice 3.4 počítá však pouze s periodickými signály v intervalu od 0 do 2π. Pro aplikaci postupného opakovaného vyčítání je nutné ještě resetovat hodnoty aktuální fáze při dosažení maxima jedné periody. To lze zařídit periodickým odečítáním (resp. přičítáním v případě opačné fáze signálu) maximální hodnoty periody k hodnotě aktuální fáze při dosažení tohoto maxima (použito ve třídě *LFO* v kapitole 3.2.6), čímž by se fáze dostala zpět do rozsahu zapsaných hodnot v paměti. Alternativně by se vzorek na poslední pozici daného průběhu dostal na začátek tohoto průběhu. Tento přístup využívá funkce `fmod`, která daný problém řeší zbytkem po dělení neceločíselných čísel. Ve výpisu 3.2 je tento kód zobrazen na řádku 4. [15]

Výpis 3.2: Cyklické čtení vzorku z vybraného průběhu.

```
float Oscillator::loopingOverWaveshape()
{
  jassert(isPlaying());
  index = std::fmod(index, static_cast<float>(waveTable.size()));
  const auto sample = interpolateLinearly();
  index += indexIncrement;
  return sample;
};
```

Po provedení výpočtu indexu vzorku je nutné provést lineární interpolaci pro případ, že by se uživatel hrou na klaviaturu netrefil do konkrétní celočíselné hodnoty indexu vzorku, což řeší implementovaná metoda `interpolateLinearly()`. Následně je k indexu vzorku přičten jednotkový inkrement, tedy vzdálenost dvou vzorků od sebe, kterou lze spočítat z počtu vzorků na periodu periodického signálu, vzorkovacího kmitočtu a požadované frekvence přehrávání [15].

$$\texttt{inkrement} = f \cdot \frac{L}{f_s}. \tag{3.5}$$

Tento inkrement se počítá v metodě `setFrequency()`. Interní metody `stop()` a `isPlaying()` zajišťují případy puštění MIDI klávesy (MIDI zpráva NoteOff) a kontrolu, zda daná instance oscilátoru aktálně existuje, tedy její hodnota inkrementu je nenulová. Pokud přijde MIDI zpráva o puštění klávesy (NoteOff), oscilátoru se zavolá metoda `stop()` a jeho inkrement i aktuální index jsou resetovány do původního stavu. Pro vytvoření polyfonního nástroje je před začátkem spuštění zavolána metoda `prepareToPlay()` ve třídě *Synth*. V té se vytvoří přesně 128 instancí třídy *Oscillator*, čímž se zaručí, že při stlačení jakékoliv klávesy v rozsahu MIDI not bude připravená jedna instance *Oscillator* pro přehrání zvuku. Výstupem této třídy je hodnota jediného vzorku vybraného průběhu uživatelem, který je zaslaný zpět do třídy *Synth()*.

3.2.4 Třída *Waveform*

Třída *Oscillator* by však ztrácela na smyslu, pokud by neměla nějaký vektor o specifických hodnotách, které by vyčítala. Tuto práci zastřešuje třída *Waveform*. Uvnitř třídy se nachází několik metod pro generování průběhů vytvořených na základě již zmíněných skriptů z Matlab popsaných v kapitole 2. Tyto metody byly předělány do jazyka C++ včetně funkcí jako `repmat` nebo `circshift`, které jsou implementovány ve třídě *HelpFuntions*, která je krátce zmíněna v navazující podkapitole.

Princip třídy spočívá ve vygenerování matice o velikosti 6 vektorů, který každý obsahuje 64 vzorků (6 řádku na 64 sloupců). Pomyslnou sběrnicí těchto vektorů je

metoda `generateWaveform()`, která přijímá vstupní parametry potřebné pro generování průběhů (například počet děliček – numDividers nebo počet bitů – numBits) a volá dílčí metody generující jednotlivé vektory průběhů. Ty jsou ukládány do výstupní matice waveformList vytvořené třídou *juce::dsp::Matrix*. Při běhu aplikace se matice aktualizuje spojením třídy *Synth* a *Waveform* metodou `updateList()`.

Tři základní průběhy kopírují logiku Walshových funkcí bez zpoždění, se zpožděním a generování pilového průběhu. Jejich implementace je rozdělena na metody `generateSawOrWalsh()` a `generateWalshD()`. Základní hodinový impuls je opět generovaný metodou `MG()` (*Master Generator*). Další vygenerované průběhy byly implementovány na základě nejčastěji dostupných tvarů průběhů v komerčních syntezátorech – sinusový, trojúhelníkový a pilový. Vzhledem k tomu, že je pilový průběh aproximován již v metodě `generateSawOrWalsh()`, byl zvolen jiný způsob generování, a to na základě matematické rovnice

$$s_{saw}[n] = 2 \cdot (\phi_{\text{norm}} - \texttt{floor}(\phi_{\texttt{norm}})), \tag{3.6}$$

kde funkce `floor` zaokrouhluje na nejbližší celé nižší číslo normalizovanou fázi $\phi_{\text{norm}} = \frac{n}{L}$ [2]. Rozdílem zaokrouhlené a nezaokrouhlené fáze vznikne nový inkrement funkce, který je zdvojnásoben a tvoří tak matematickou rovnici přímky se směrnicí $k = 2$. Po průchodu všemi vzorky vznikne průběh pily, který je generován metodou `generateSawtooth()`. [16] [17] Dalším krokem bylo vytvoření trojúhelníkového tvaru, který byl podobně jako tvar pilový spočítán na základě matematické rovnice v metodě `generateTriangular()`, která má následující tvar

$$s_{tri}[n] = 1 - 4 \cdot |(\texttt{fmod}(\phi_{\text{norm}}, 1) - 0{,}5)|. \tag{3.7}$$

Jako poslední byl vygenerován sinusový průběh pomocí známé matematické rovnice

$$s_{sin}[n] = 2\pi \cdot \phi_{\text{norm}}. \tag{3.8}$$

Průběhy jsou zkontrolovány a normalizovány pomocí metody `changeNumSamples()` ze třídy *HelpFunctions* na celkový počet 64 vzorků. Po vygenerování posledního průběhu metodou `generateSine()` jsou všechny průběhy uloženy do již zmíněné matice waveformList a zaslány do třídy *Synth* pro další zpracování.

Třída *HelpFunctions*

Třída *HelpFunctions* zastřešuje základní matematické a logické operace používané v ostatních třídách. Tato třída byla nejprve zamýšlena pouze na replikaci funkcí používaných prostředím Matlab, avšak postupem času se ukázala jako ideální místo pro vytvoření dalších pomocných metod. Vnitřní metody replikují funkce `circshift()` a `repmat()` z Matlabu, které byly použité pro tvorbu demonstračních skriptů.

[2] viz vzorec 3.3

3.2.5 Třída *Synth*

Pokud byly třídy *Oscillator* a *Waveform* pomyslné nohy a ruce systému, poté je třída *Synth* bez pochyby mozkem celého systému. Třída se stará o veškeré fungování dílčích tříd od zvolení typu generovaného průběhu až po výstup tohoto průběhu skrze vyrovnávací paměť. Je zde řešena reakce na MIDI data, aktualizace jednotlivých tříd i propojení dílčích komponentů grafického rozhraní s vnitřními parametry procesoru. Před spuštěním syntezátoru je třídou inicializováno v metodě `prepareToPlay()` všech 128 oscilátorů, všechny instance třídy *LFO* a ADSR obálka. Při běhu plug-in modulu se třída aktualizuje průběžně pomocí metody `updateSynth()`, která aktualizuje parametry změněné uživatelem a zasílá tyto informace ostatním dílčím třídám.

Hlavním nástrojem této třídy je metoda `updateWaveform()`, která podle zvoleného přístupu vyčítání (Wavecycle, Wavetable) vykonává operaci vyčítání z příchozí matice waveformList ze třídy *Waveform*. Výsledný vektor je v případě zvolení metody vyčítání *Multicycle* ještě zredukován na pevný počet 64 vzorků. Následně je tento výstupní vektor přiřazen všem instancím oscilátorů. V této metodě jsou aplikovány i všechny zesilovací prvky, aplikace tremolo efektu nebo LFO. Uživatel může díky této metodě volit různé způsoby vyčítání dat, a to konkrétně metody *Single-cycle*, *Multi-cycle*, *Swept* a *Random-access*.

Zpracování MIDI dat je prováděno v metodě `handleMidiEvent()`, kde jsou nastaveny tři základní zprávy, které syntezátor může přijmout, a to konkrétně nota zapnuta, vypnuta a možnost práce se stavem, kdy jsou všechny noty vypnuty. Při stisku klávesy jsou brány v potaz i možnosti, kdy je použito převzorkování a musí se tak pro korektní přehrávání požadované frekvence vynásobit obrácenou hodnotou korekční konstanty samplingCorrection, která udává druhou mocninu aktuálně zvoleného indexu převzorkování oversamplingFactor. Jinými slovy se musí frekvence podělit zvoleným násobkem převzorkování. Ještě před voláním metody pro zjištění frekvence oscilátoru je index oscilátoru, resp. hodnota MIDI noty, sečtena s parametrem půltónového přeladění. Frekvence oscilátoru se počítá v metodě `midiNoteNumberToFrequency()` skrz rovnici pro temperované ladění, kde nota A4 je rovna 440Hz a její index v MIDI zápisu je roven 69

$$f(n) = 440 \cdot 2^{\frac{n-69}{12}}, \tag{3.9}$$

kde parametr n udává hráčem zahranou MIDI notu. [18] Po zjištění frekvence je danému oscilátoru frekvence přiřazena a je na něm aktivována ADSR obálka pomocí metody `noteOn()`, přičemž konkrétní parametry ADSR obálky jsou nastaveny v metodě `updateADSR()` a obálka jako taková se aplikuje na celý *buffer* ve třídě

PluginProcessor [19]. Při ukončení přehrávání (uvolnění klávesy) je danému oscilátoru pomocí metody `stop()` nastaven inkrement a aktuální index na nulové hodnoty (viz kapitola 3.2.3). V případě, že žádný z oscilátorů není aktivní, vyčistí se výstupní vektor s průběhem a zastaví se všechny oscilátory.

Proces přijímání MIDI zprávy a ukládání dat do výstupní vyrovnávací paměti je uložen v metodě `processBlock()`, která vykonává velmi podobnou práci jako její sesterská metoda ve třídě *PluginProcessor*. Jako příchozí data si metoda bere celkový *buffer* a *buffer* MIDI zpráv. Pokud hráč zahraje na klávesu, metoda z MIDI zprávy vyčte informaci o typu zprávy a časový moment, kdy byla zpráva vytvořena. Následně zprávu pošle do metody `render()`, která spojuje dříve zmíněnou cirkulaci v metodě `loopingOverWaveshape()` z kapitoly 3.2.3 s výstupním *buffer*. Metoda `render()` kontroluje všechny oscilátory a pokud najde aktuálně přehrávaný oscilátor, projde ho pomocí `loopingOverWaveshape()` a uloží jednotlivé hodnoty do výstupního *bufferu*.

3.2.6 Třída *LFO*

Třída tvořící nízkofrekvenční oscilátory má na starost modulaci a vyčítání signálu, především potom zastřešuje způsob vyčítání způsobem Wavetable metodou *Swept* a *Random-access*. Při vyčítání způsobem *Swept* je v metodě `updateWaveform()` nejprve získána aktuální hodnota LFO a pomocí následující rovnice získán index požadovaného průběhu v matici v závislosti na této hodnotě

$$n = n_a \cdot \texttt{floor}(|(k-1) \cdot \texttt{lfo}|), \tag{3.10}$$

kde n_a určuje aktuálně zvolený index uživatelem, k je celkový počet uložených průběhů v matici a `lfo` je aktuální hodnota LFO.

Metoda *Random-access* je založena na segmentaci hodnot LFO signálu, kdy je pomocí metody `decidedIndexByLFO()` vybrán přírůstek přičítaný k aktuálnímu indexu ukazatele na určitý průběh z matice waveformList. Tento index je zaslán zpět do `updateWaveform()` pro vygenerování nového výstupního vektoru.

Další instancí této třídy je objekt tremolo, který moduluje signál přímo v metodě `updateWaveform()`. Implementace generování vzorků je podobná generování průběhů ve třídě *Oscillator*, avšak nastalo zde několik změn. Jednotlivé hodnoty se generují v podstatě na stejném principu fázového posunu (inkrementu), nyní je však použito klasické odčítání maximální hodnoty periody při dosažení tohoto maxima fází oproti dřívějšímu použití funkce `fmod`. Konkrétní hodnoty jsou generovány na základě logických obvodů pomocí `while` cyklů a matematické funkce sinus podle vzoru [20].

3.2.7 Třída *Filters*

Po naplnění *bufferu* vzorky z *hlavního generátoru* je vyrovnávací paměť zaslána do třídy *Filters*. Zde je možné vzorky filtrovat základními typy filtrů, a to dolní propustí, horní propustí a filtrem typu Peak (angl. Low pass, High pass a Peak). Pro zjednodušení a zpřehlednění kódu byla třída *Filters* inspirována návodem od Charlese Schiermeyera [21]. Filtrům je před spuštěním nejprve předána informace o vzorkovacím kmitočtu hostitelské aplikace a jsou inicializovány pro práci s dvěma mono kanály. Zpracování signálu probíhá v kaskádě, kdy signál nejprve projde dolní propustí, poté peak filtrem a naposledy horní propustí. Důležitou částí této třídy je metoda `getChainSettings()`, která načítá hodnoty parametrů z GUI za pomoci `AudioProcessorValueTreeState`, který bude popsán později v kapitole 3.2.8, a ukládá je do struktury ChainFilterSettings. Tato struktura zahrnuje parametry pro Peak, HighPass a LowPass filtry, což umožňuje centralizovanou správu všech filtrů v rámci jednoho objektu, podobně jako tomu je u hlavičkového souboru `SynthParameters.h`. Aktualizace všech koeficientů je řešena metodou `updateCoefficients()`, která přijímá staré a nové koeficienty filtrů a aktualizuje je na základě volání z metody `update()`, což je nezbytné pro správnou funkci filtru při změně parametrů. Následují realizace jednotlivých metod filtrů, které mají podobné rozložení.

Výpis 3.3: Tvorba koeficientů filtru typu horní propust pomocí třídy *FilterDesign*.

```
auto highPassCoeff =
  juce::dsp::FilterDesign<float>::
  designIIRHighpassHighOrderButterworthMethod
(
  chainSettings.highPassFreq,
  sampleRate,
  2 * (chainSettings.highPassSlope + 1)
);
// vytvoreni koeficientu High pass filtru
```

Na začátku metod `updateLowPassFilter()` a `updateHighPassFilter()` je vždy zavolána jedna z metod pro výpočet koeficientů filtrů typu IIR v závislosti na mezním, nebo též středním kmitočtu filtru f_c, vzorkovací frekvenci f_s a strmosti filtru (angl. *Slope*). Realizace horní a dolní propusti nabízí čtyři možnosti strmostí, a to 12, 24, 36 a 48 dB/okt. V případě filtru typu Peak v metodě `updatePeakFilter()` jsou nastaveny kromě středního kmitočtu i parametry činitele jakosti filtru Q a Gain filtru, které ovlivňují šířku pásma filtru a zesílení, resp. zeslabení výstupního signálu v určitém pásmu. Rozsah činitele jakosti Q Peak filtru dosahuje hodnot od $Q = 0{,}1$ do $Q = 24$, zesílení lze upravovat v rozsahu Gain $= \pm 24\,\mathrm{dB}$. Pro vytvoření

koeficientů jsou použity metody `makePeakFilter()` a metody tvořící koeficienty IIR filtrů podle Butterwortha obsažené třídou *FilterDesign* (viz dokumentace [22]). Po nastavení koeficientů filtrů následuje v metodách kontrola parametru bypass, který umožňuje konkrétní filtr vypnout. Posléze je proveden samotný proces zpracování signálu pro pravý i levý kanál v metodě `updateFilter()`, ve které lze nalézt spádový `switch`, který podle zvolené strmosti filtru nastavuje přírůstek výsledné strmosti zvoleného filtru.

3.2.8 Komunikace procesoru s uživatelským rozhraním

Komunikaci procesoru a uživatelského rozhraní řeší interní třída frameworku JUCE – *AudioParameterValueTreeState*, zkráceně *APVTS*. *APVTS* spojuje reaktivní prvky GUI s parametry procesoru, které posléze ovlivňují chování procesoru (například změna počtu děliček při generování průběhů Walshových funkcí – numDividers). Základní deklarace třídy *APVTS* a metody přidávající parametry `createParams()` jsou zobrazeny ve výpisu 3.4. Hodnoty parametrů jako takové jsou aktualizovány v metodě `getSynthParams()` uvnitř třídy *Synth*. Ta se odkazuje na strukturu parametrů v hlavičkovém souboru SynthParameters.h, kde jsou pro jednoduchost uloženy i jednotlivé ID těchto parametrů. Pro filtry jsou parametry aktualizovány metodou `getChainSettings()`. Ve třídě *PluginEditor* se k těmto parametrům přistupuje opět skrze *APVTS* a vytváří se spojení pomocí tzv. *Attachments*. [23]

Výpis 3.4: Deklarace třídy a metody řešící propojení s uživatelským rozhraním.

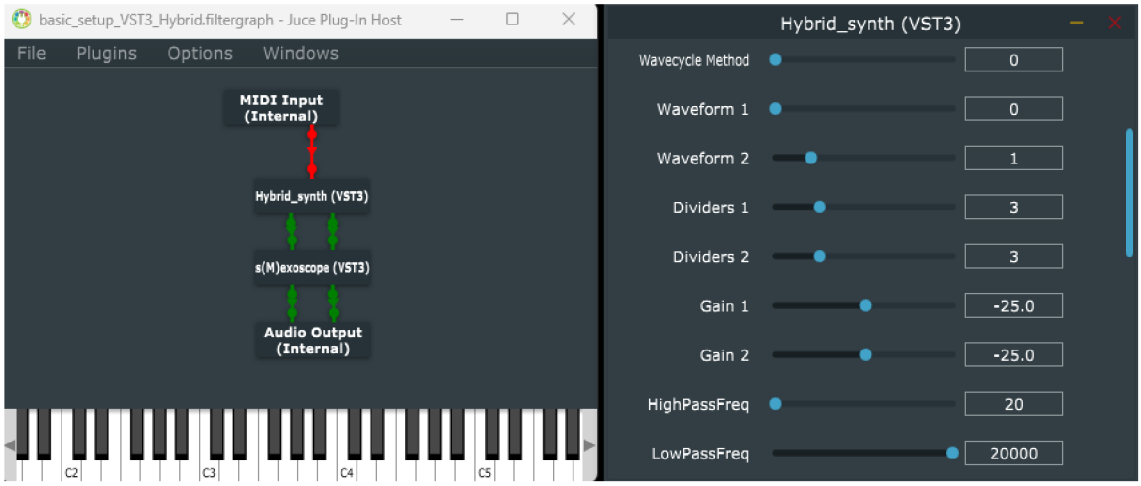
```
1    juce::AudioProcessorValueTreeState apvts
2    { *this, nullptr, "Params", createParams() };
3    //Objekt, ktery se stara o spravu a propojeni procesoru s GUI
4
5    juce::AudioProcessorValueTreeState::ParameterLayout createParams();
6    //deklarace funkce ParameterLayout: uklada nazvy parametru
7    //a jejich aktualni hodnoty, ktere dale posila do procesoru
```

Výpis 3.5: Ukázka přidání parametru skrz metodu *createParams()*.

```
    layout.add(std::make_unique<juce::AudioParameterFloat>(
        //Metoda add() pridava parametr do komunikacniho kanalu
        //a nastavuje zakladni parametry
        //jako nazev, rozsah hodnot atd.
        synth.synthParamsID.gainID,
        synth.synthParamsID.gainID,
        juce::NormalisableRange<float>(-60.0f, 10.0f, 0.01f),
        0.0f));
```

3.3 Demonstrace funkcí syntezátoru

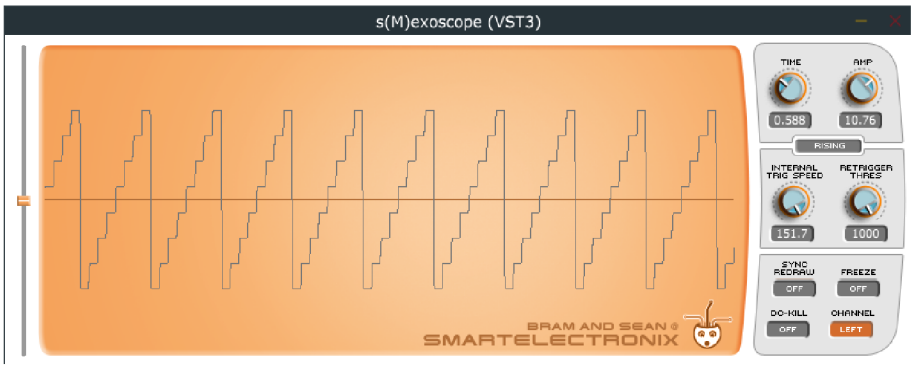
Kontrola funkčnosti programu probíhala ve dvou po sobě jdoucích částech. Nejprve byl program testován v prostředí Juce Plug-in Host, který umožňuje generování MIDI zpráv a průchod kaskádou zapojených procesorů. V tomto případě byla MIDI zpráva zaslána do vytvořeného syntezátoru, načež následovalo zobrazení vygenerovaného signálu v osciloskopu. Pro zobrazení průběhu v čase byl použit volně dostupný plug-in *S(M)exoscope)* od vývojáře Bram de Jong (pod záštitou bram @ smartelectronix plugins). [24] V této části byla testována funkčnost jednotlivých prvků, které ovlivňují výstupní signál. Po ověření funkčnosti byl program ve formátu VST3 testován v DAW Ableton Live v12.0.2.



Obr. 3.3: Zobrazení testovacího prostředí *Plugin Host*.

Hlavním bodem byla funkčnost generování jednotlivých průběhů obsažených v hybridních syntezátorech včetně jejich dílčích úprav a možných způsobů vyčítání. Ve výsledné verzi jsou zahrnuty metody *Wavecycle* způsobem *Single-cycle*, *Multi-cycle* a metoda *Wavetable* způsobem čtení *Swept* a *Random-access*. Kritický moment nastal při vyčítání průběhů právě pomocí metody *Random-access*. Metoda vybírá čtené průběhy na základě prahových hodnot LFO, proto je nutné při každém překročení prahové hodnoty aktualizovat celou matici průběhů `waveformList` v krátkém časovém úseku. To je datově i časově náročné a na prahových hodnotách při změně indexu průběhu vzniká problém, který znehodnocuje výstupní data uložená v *bufferu*.

Další zásadní komponentou syntezátoru byl modul subtraktivní syntézy, který obsahuje filtry popsané v kapitole 3.2.7. Další komponentou je efekt tremolo, který byl implementován za pomoci třídy *LFO*. V neposlední řadě byla testována možnost převzorkování celého systému.



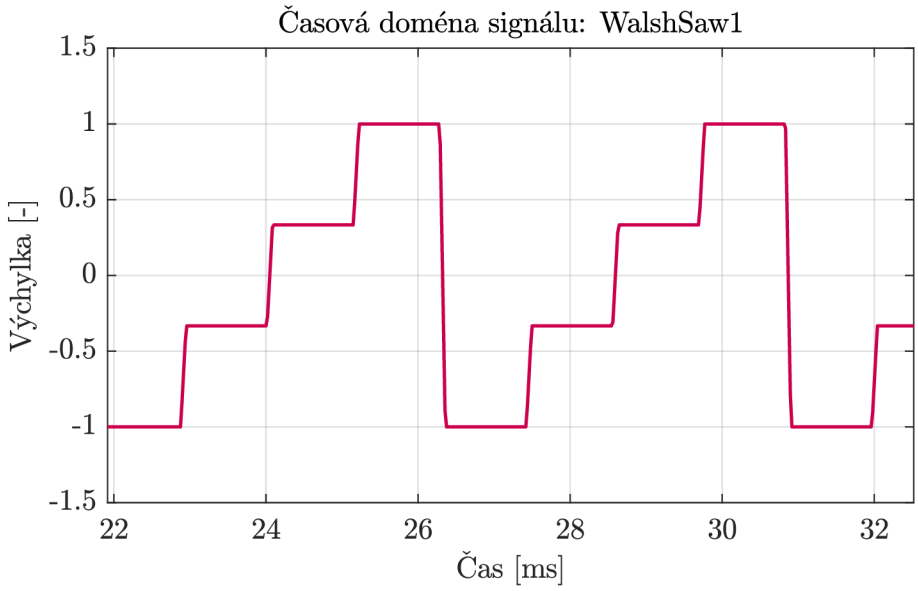
Obr. 3.4: Zobrazení osciloskopu.

3.3.1 Generované průběhy a metoda vyčítání *Single-cycle*

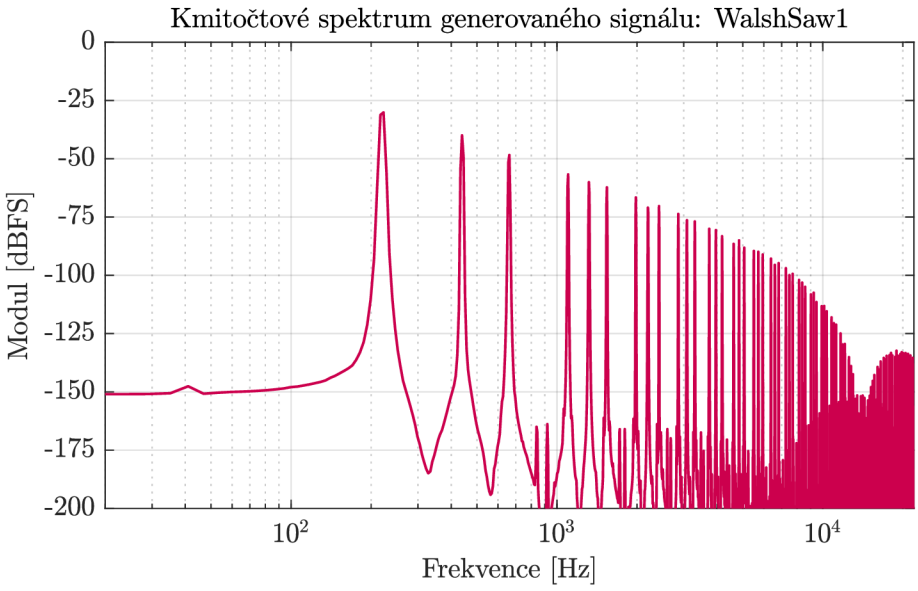
Všechny průběhy založené na metodách demonstrovaných Matlab skripty je možné ovlivňovat parametrem udávající počet děliček s názvem numDividers. Ten umožňuje uživateli měnit tvar průběhu a tím i zároveň demonstruje dílčí součty signálů vycházející z jednotlivých děliček (viz například graf 2.7). Všechny testované signály byly generovány pro frekvenci $f = 440\,\mathrm{Hz}$, tedy komorní a. Ostatní parametry syntezátoru zůstaly nezměněny kromě obálky ADSR, která byla omezena pouze na maximální hodnotu Sustain. Ostatní parametry byly nastaveny na nulové hodnoty. Tímto testováním bylo pokryto i testování metody *Single-cycle*, která vyčítá právě jeden výstupní průběh. Jednotlivé demonstrační zvukové soubory patřící k jednotlivým grafům jsou umístěny v příloze na konci této práce.

Jako první byl testován pilový průběh, který je zobrazen v několika konfiguracích s proměnným počtem děliček. Z grafu 3.6 je patrné, že na vyšších kmitočtech vzniká určitý útlum (v pásmu za 10 kHz) a zhuštění spektrálních složek. To je zapříčiněno absencí antialiasingového filtru, který nebyl do syntezátoru implementován, a vzniká tak součet se zrcadlenými složkami. Tento jev je částečně potlačen vlivem možného převzorkování signálu. Se vzrůstajícím počtem děliček se průběh více přibližuje matematickému popisu tvaru pily, což potvrzuje i teorie z kapitoly 1.

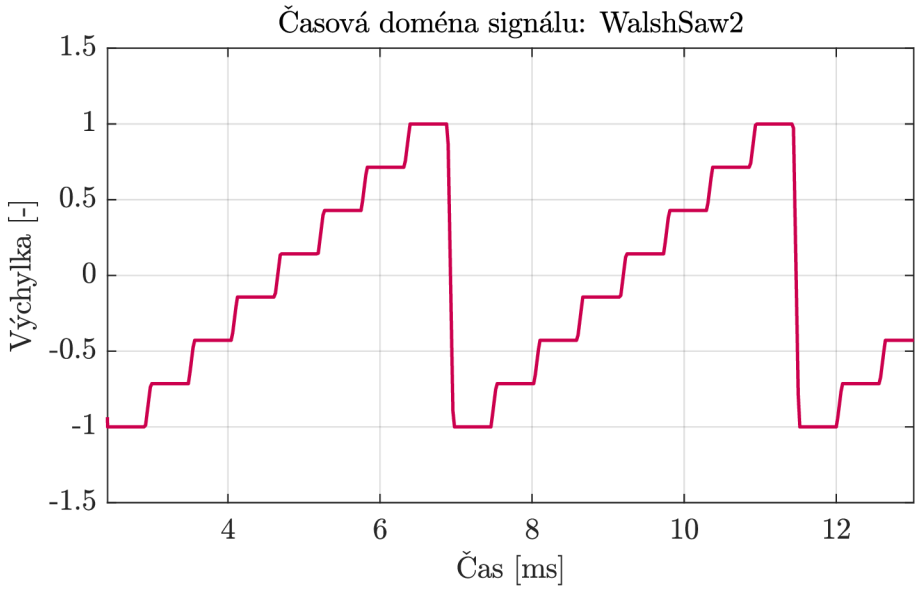
Následovalo testování průběhu Walshových funkcí bez zpoždění. Průběh byl testován pro všechny varianty počtu děliček. Po vygenerování Walshových funkcí bez zpoždění byla otestována i varianta se zpožďovací linkou. Plug-in neumožňuje změnu zpoždění mezi sčítanými signály generujícími výstupní signál, avšak je záměrem tuto možnost doplnit. Na následujících stránkách jsou graficky zobrazeny v časové i frekvenční oblasti jednotlivé průběhy v závislosti na počtu použitých děliček. Vzhledem k podobnosti většiny frekvenčních spekter a zájmu spíše o tvary generovaných průběhů jsou uvedeny vždy pouze spektra signálů generovaných 2 a 3 děličkami. Zvukové soubory, frekvenční spektra a dodatečné průběhy lze nalézt v příloze této práce.



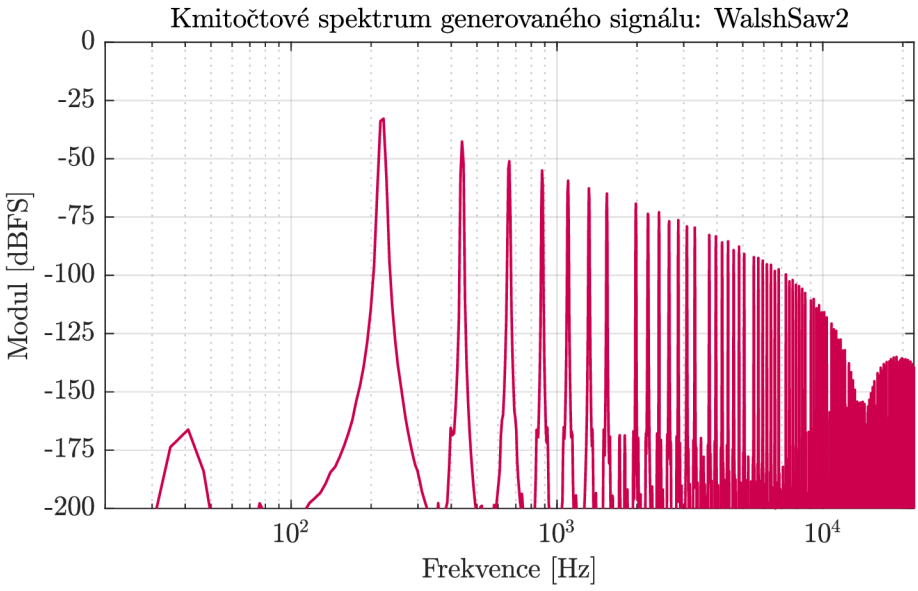
Obr. 3.5: Pilový průběh generovaný 2 děličkami.



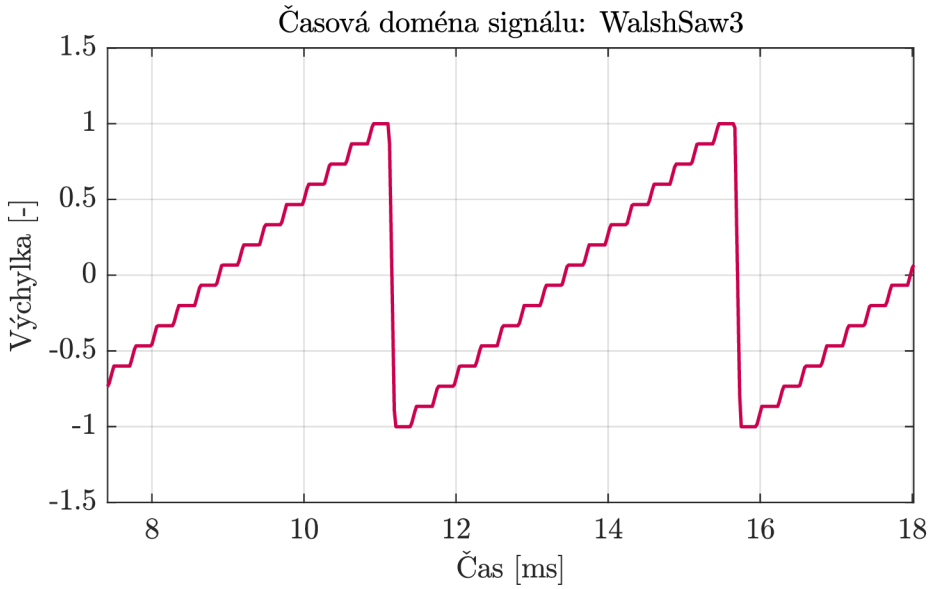
Obr. 3.6: Frekvenční spektrum signálu *WalshSaw1*.



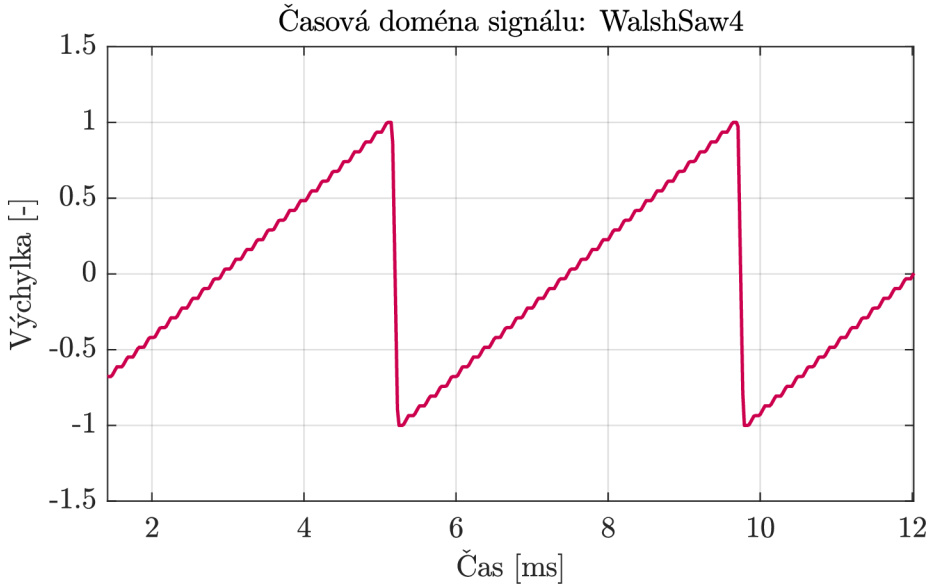
Obr. 3.7: Pilový průběh generovaný 3 děličkami.



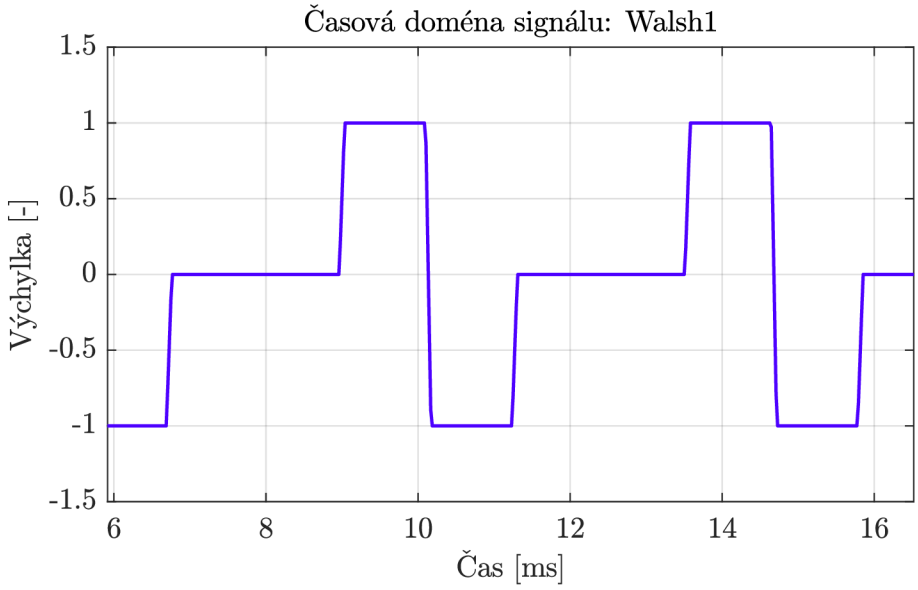
Obr. 3.8: Frekvenční spektrum signálu *WalshSaw2*.



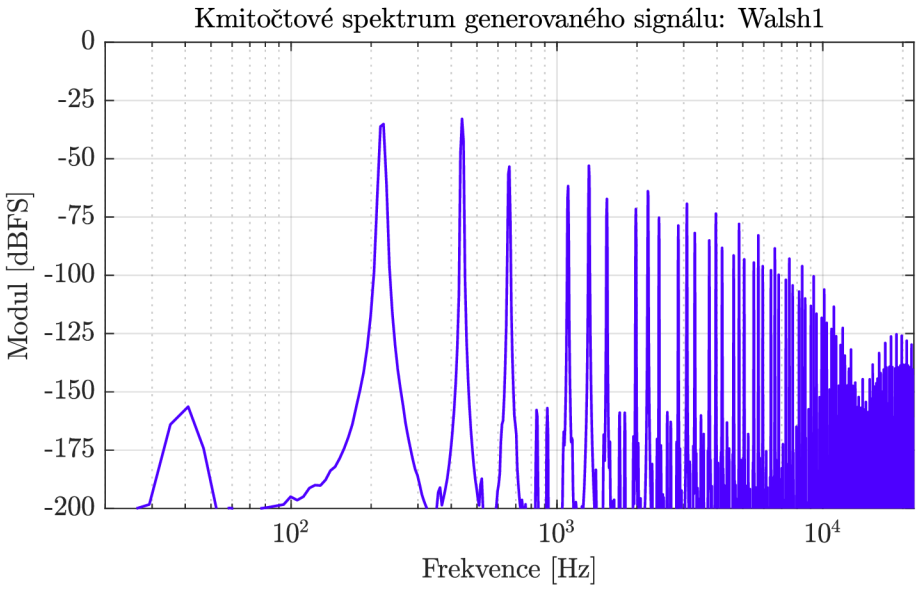
Obr. 3.9: Pilový průběh generovaný 4 děličkami.



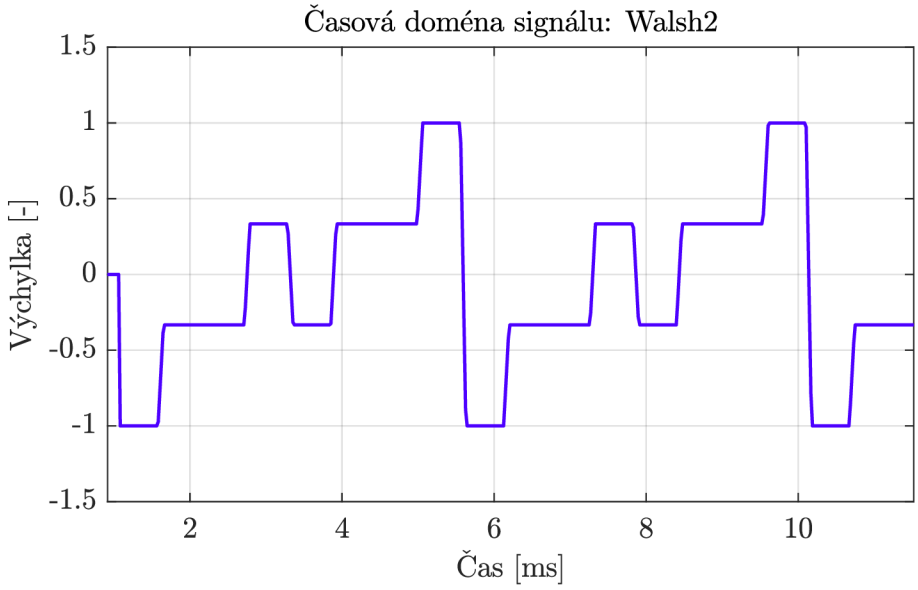
Obr. 3.10: Pilový průběh generovaný 5 děličkami.



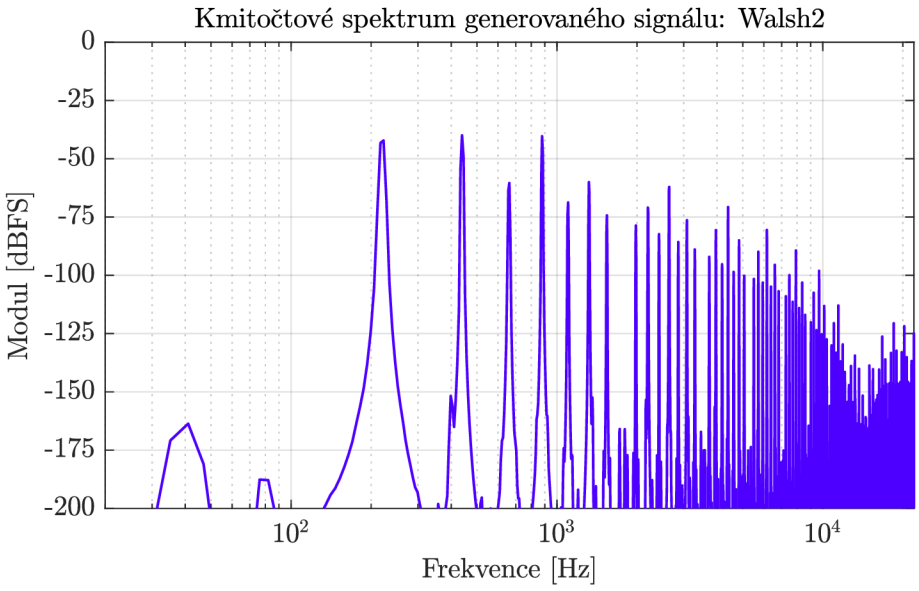
Obr. 3.11: Průběh Walshových funkcí generovaný 2 děličkami.



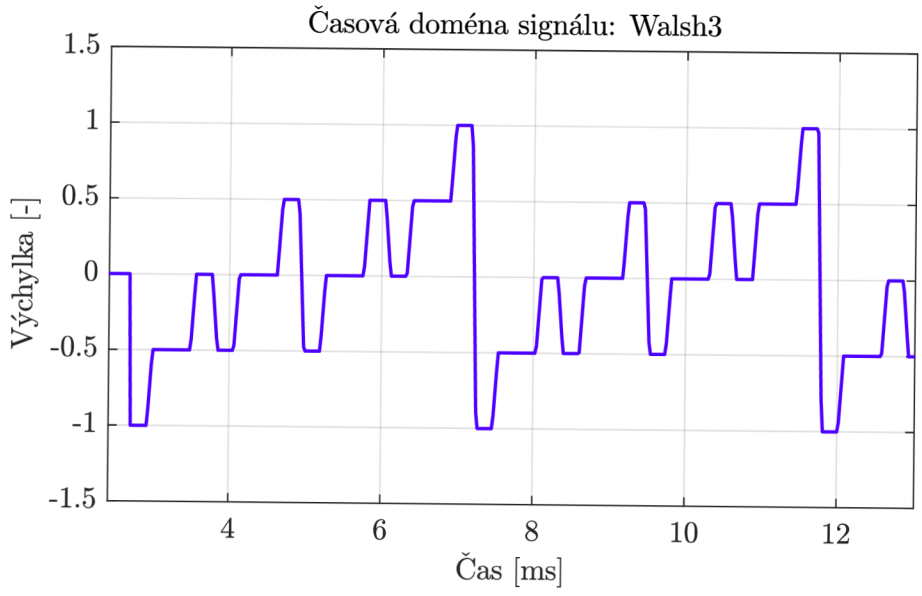
Obr. 3.12: Frekvenční spektrum signálu *Walsh1*.



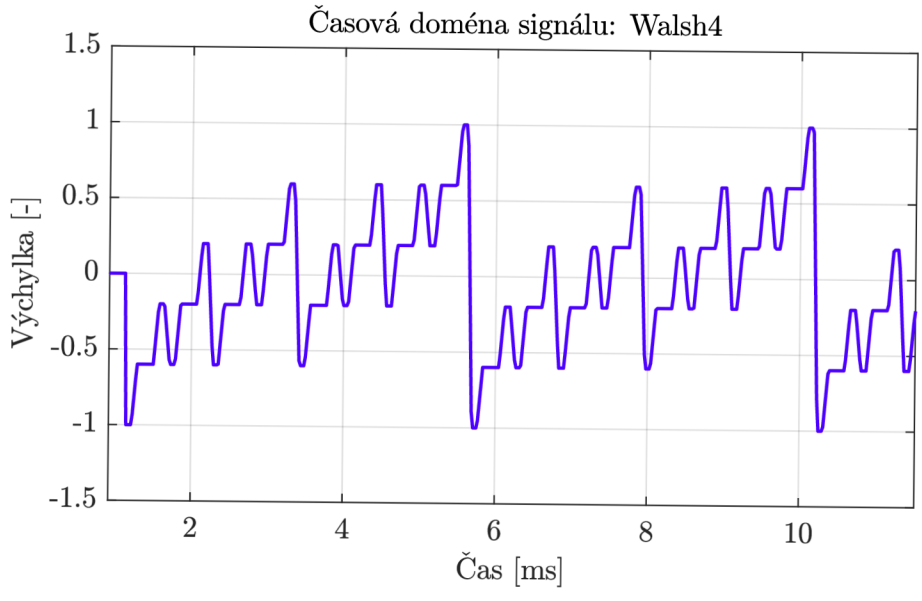
Obr. 3.13: Průběh Walshových funkcí generovaný 3 děličkami.



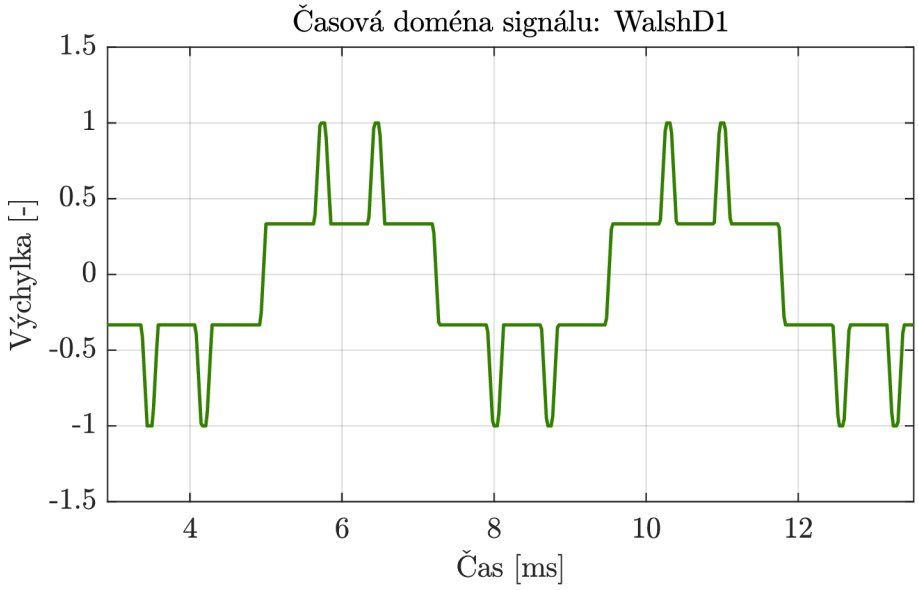
Obr. 3.14: Frekvenční spektrum signálu *Walsh2*.



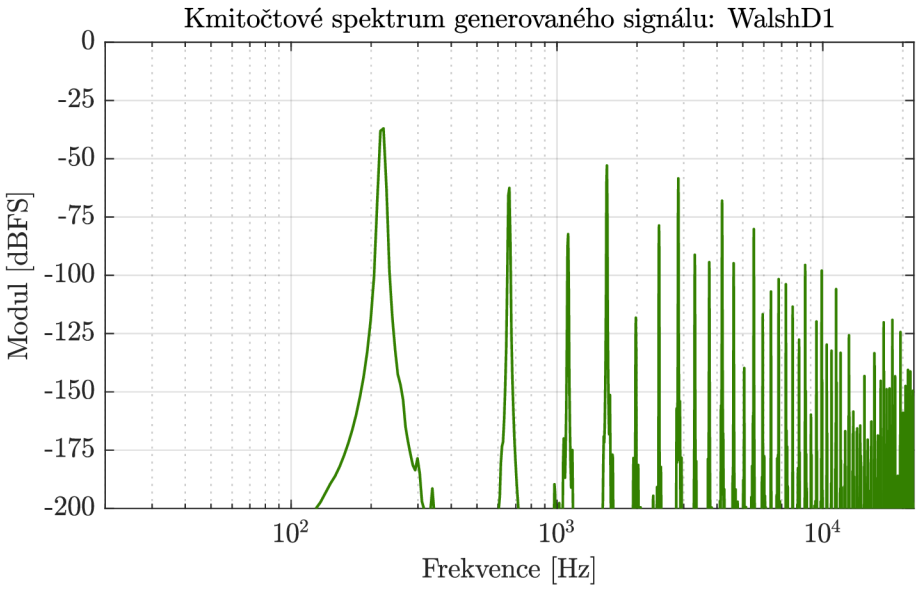
Obr. 3.15: Průběh Walshových funkcí generovaný 4 děličkami.



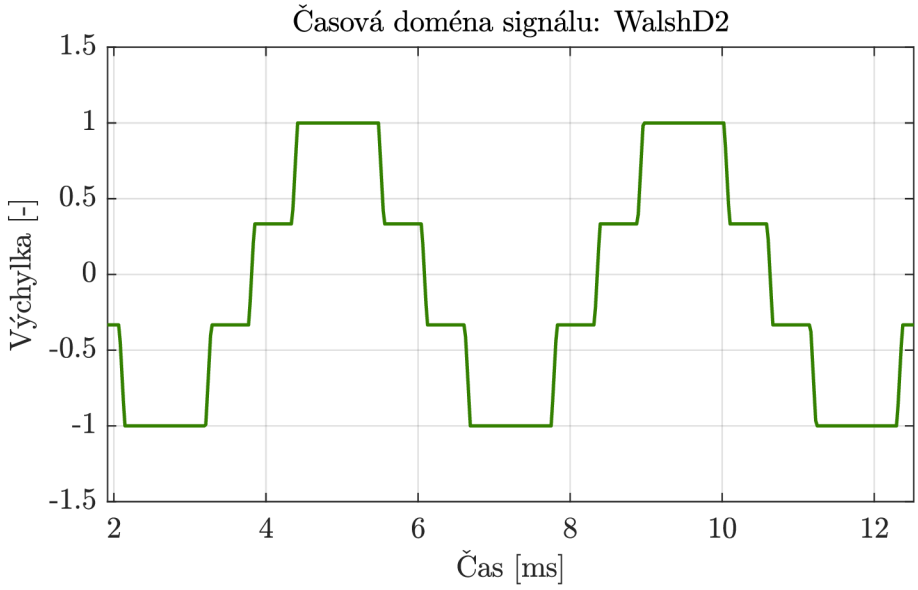
Obr. 3.16: Průběh Walshových funkcí generovaný 5 děličkami.



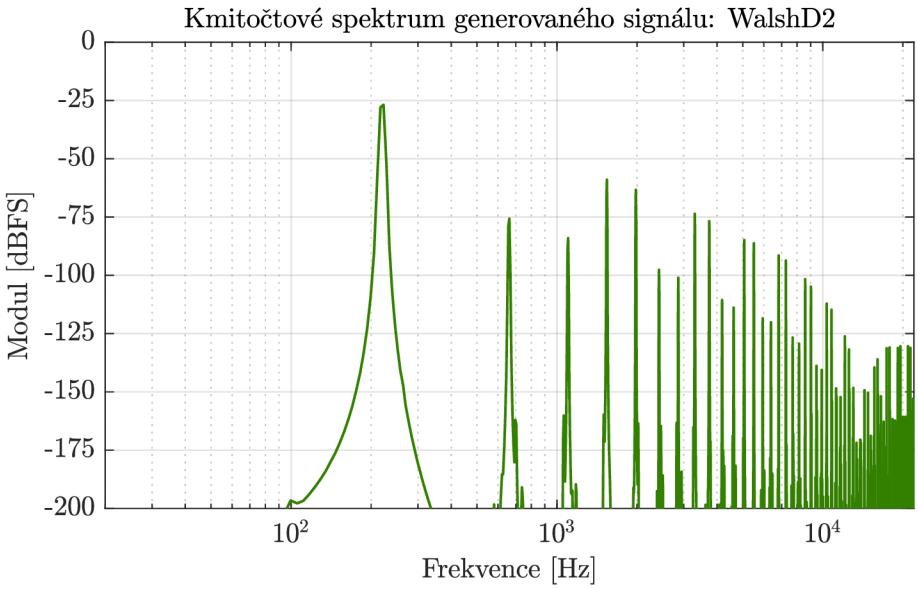
Obr. 3.17: Průběh Walshových funkcí se zpožděním generovaný 2 děličkami.



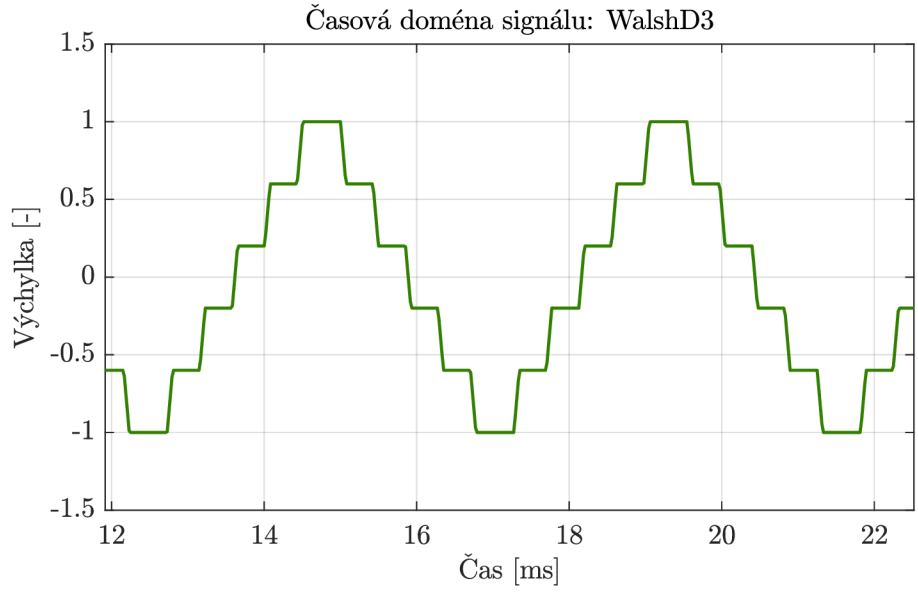
Obr. 3.18: Frekvenční spektrum signálu *WalshD1*.



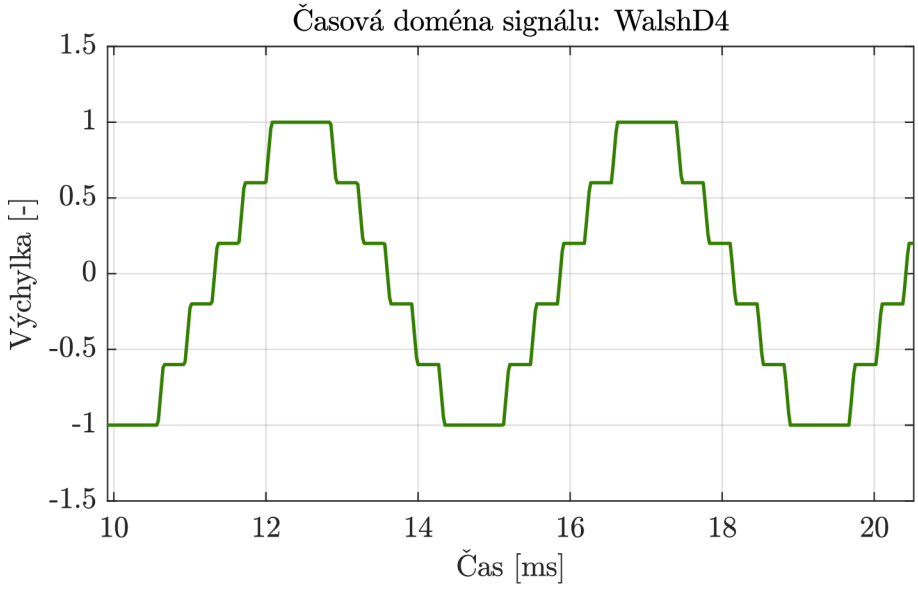
Obr. 3.19: Průběh Walshových funkcí se zpožděním generovaný 3 děličkami.



Obr. 3.20: Frekvenční spektrum signálu *WalshD2*.



Obr. 3.21: Průběh Walshových funkcí se zpožděním generovaný 4 děličkami.

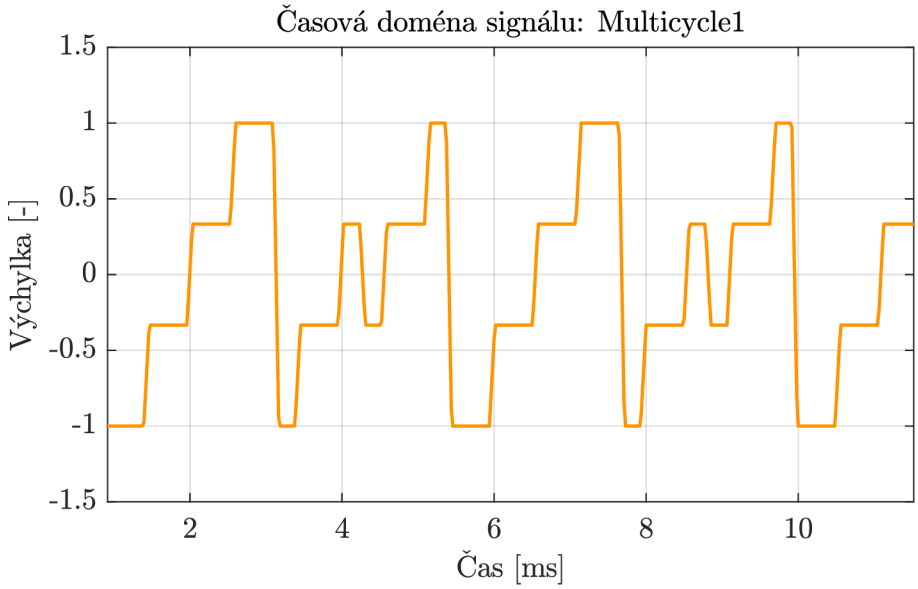


Obr. 3.22: Průběh Walshových funkcí se zpožděním generovaný 5 děličkami.

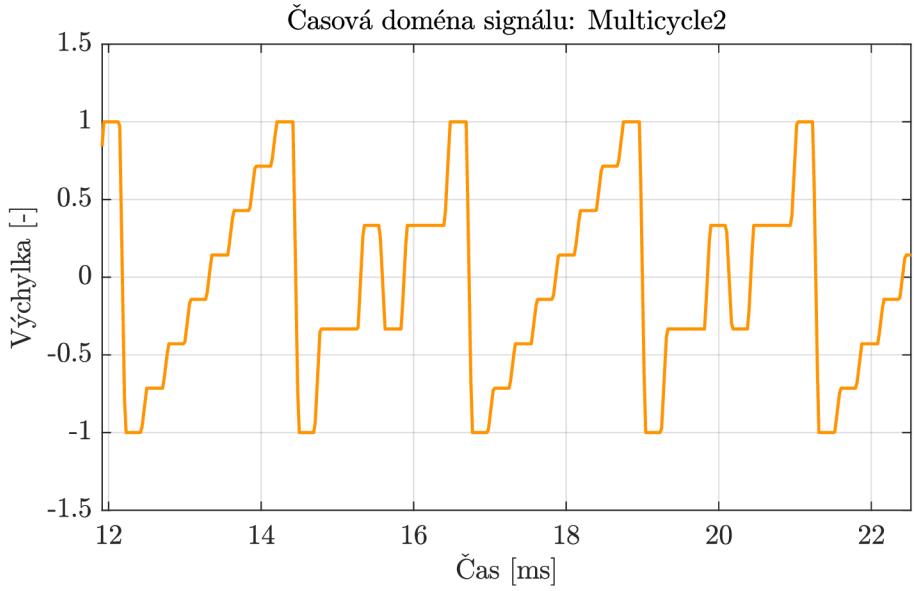
3.3.2 Metoda vyčítání *Multi-cycle*

Uživatel má celkem 30 možností kombinací ($m = (6 \cdot 6) - 6$, pokud nepočítáme 6 možností kombinací stejných průběhů) jednotlivých průběhů metodou *Multi-cycle*. Signály generované pomocí metod hybridní syntézy mají však dalších 5 možností rozdílných tvarů v závislosti na počtu děliček (celkem 5 možností nastavení děličky pro každý ze 3 generovaných průběhů). Tím počet kombinací vzroste na číslo 306 ($m = ((5 \cdot 3) + 3) \cdot 17)$).

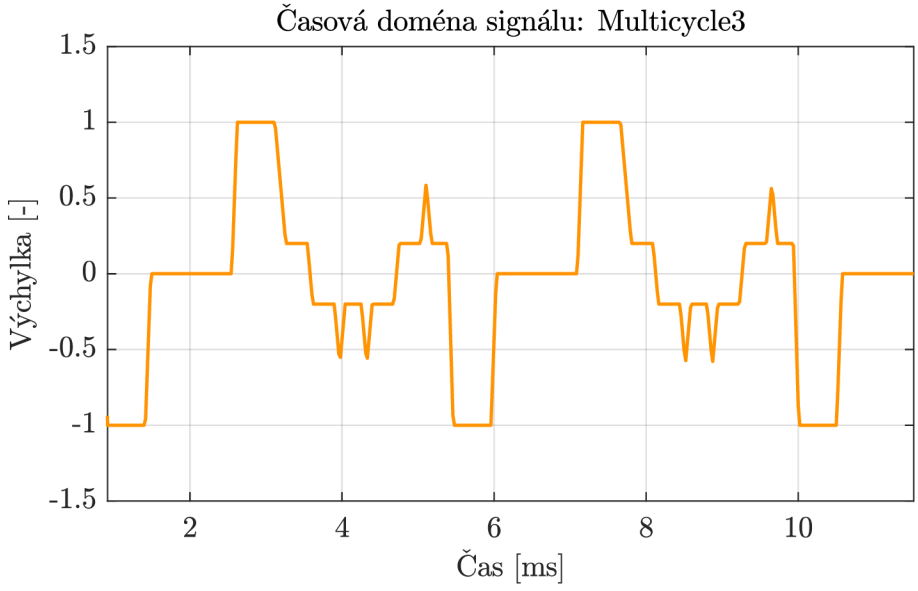
Pro zachování rozumné délky práce bylo pro demonstraci zvoleno 9 kombinací průběhů, které byly náhodně vybrány z matice průběhů. Níže je zobrazeno pět z nich, zbylé průběhy jsou obsaženy v příloze včetně frekvenčních spekter a zvukových souborů.



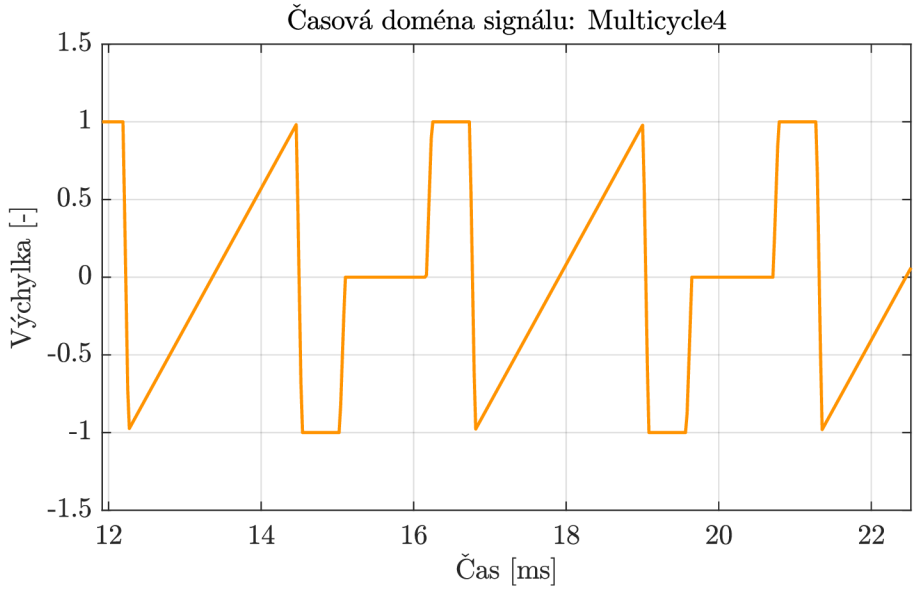
Obr. 3.23: Průběh kombinace – Pila-Walsh.



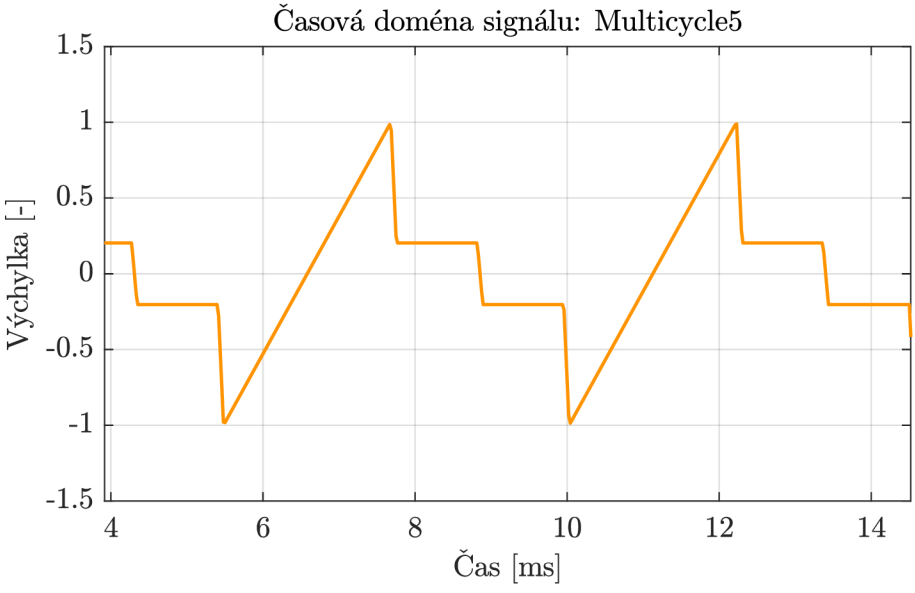
Obr. 3.24: Průběh kombinace – Pila-Walsh.



Obr. 3.25: Průběh kombinace – Walsh-WalshD.



Obr. 3.26: Průběh kombinace – Sawtooth-Walsh.

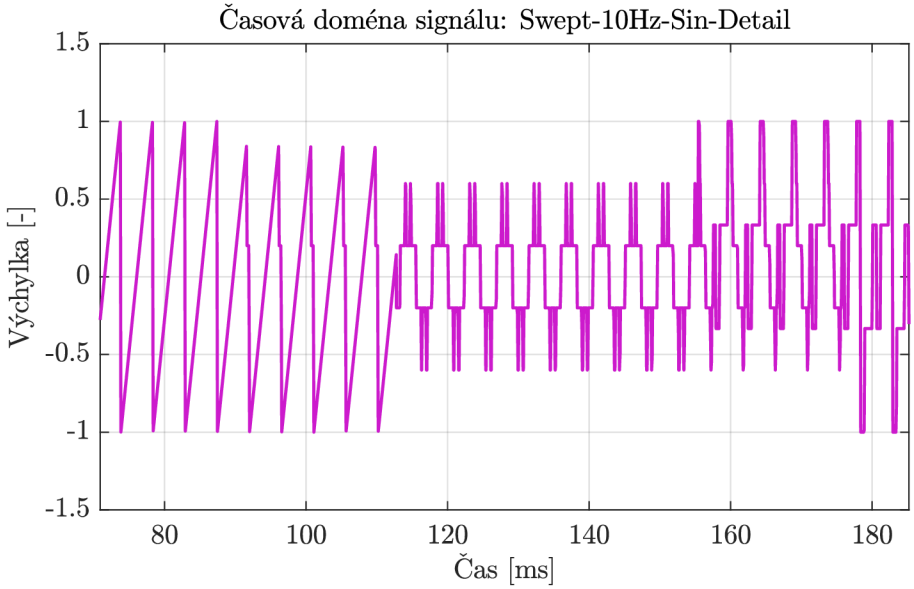


Obr. 3.27: Průběh kombinace – Sawtooth-WalshD.

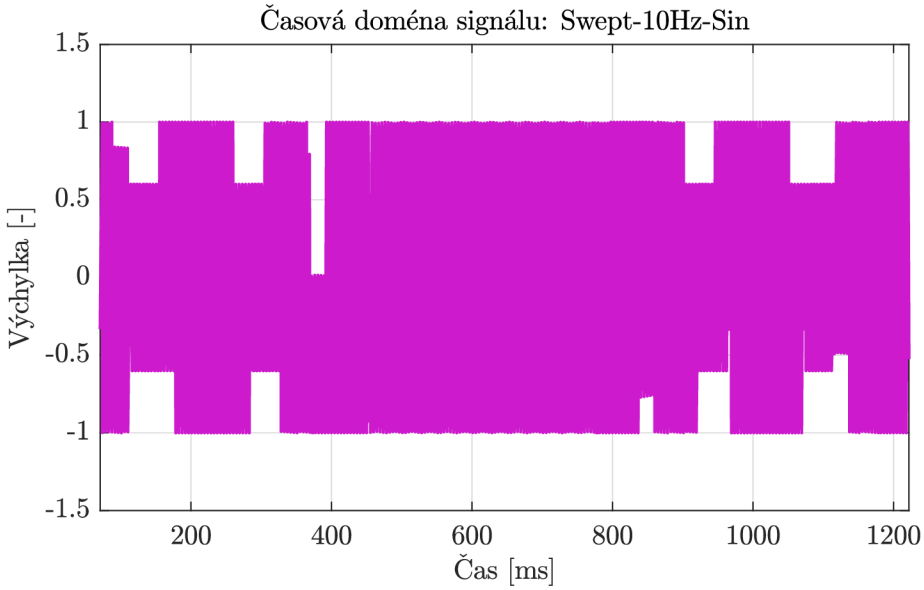
3.3.3 Metoda vyčítání *Swept*

Postupné procházení maticí průběhů pomocí aktuální hodnoty LFO bylo řešeno v rámci metody `updateWaveform()` jednoduchou matematickou operací popsanou v kapitole 3.2.6. Pro demonstraci byly vybrány průběhy sinusového a trojúhelníkového tvaru LFO. Parametry LFO pro testování byly následující

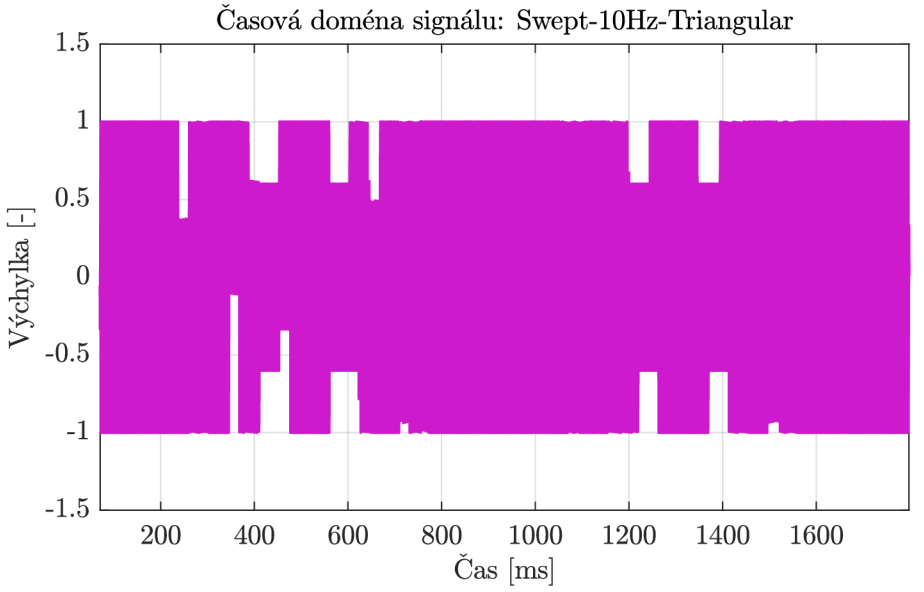
- frekvence: $f_{\texttt{LFO}} = 5\,\text{Hz}$ a $10\,\text{Hz}$,
- hloubka modulace *Depth*: $D = 1$,
- prvotní signál: Walshovy funkce generované 3 děličkami.



Obr. 3.28: Detail přechodu vyčítání metodou *Swept* (LFO-sinusový signál).



Obr. 3.29: Průběh vyčítaný metodou *Swept* (LFO-sinusový signál).

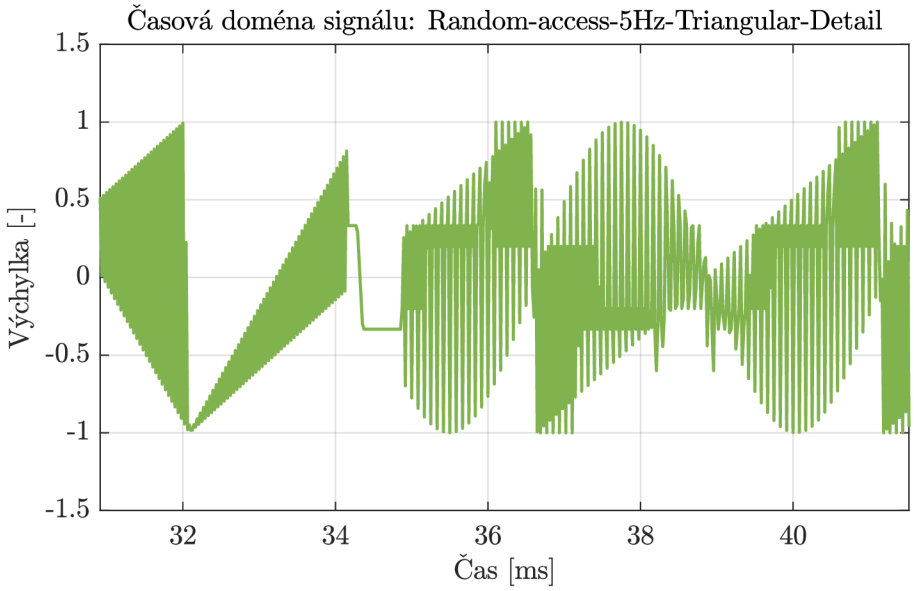


Obr. 3.30: Průběh vyčítaný metodou *Swept* (LFO-trojúhelníkový signál).

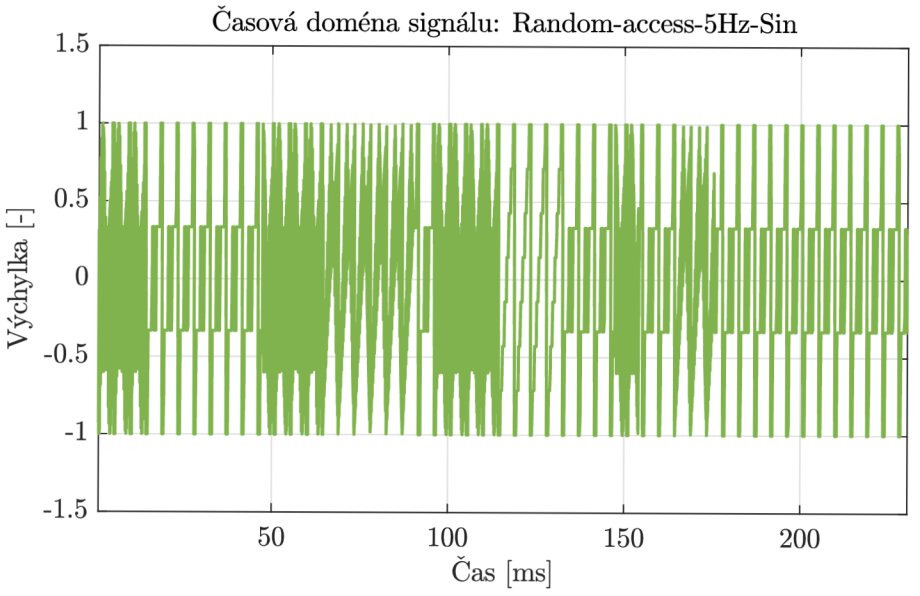
3.3.4 Metoda vyčítání *Random-access*

Náhodné vyčítání z matice pomocí nízkofrekvenčního oscilátoru bylo jedním z problémů implementace plug-in modulu. Zásadní problém netvořila ani tak implementace LFO, ale její funkční propojení s *hlavním generátorem*. Tento problém nebyl ve výsledku odladěn do ideální podoby, a tak vyčítání způsobuje degradaci dat při jeho zpracování ve výstupním *bufferu*. Detailní ukázka problému je zobrazena v grafu 3.31. Stejně jako pro metodu *Swept* byly vybrány demonstrační průběhy sinusového a trojúhelníkového tvaru LFO. Testovací parametry všech LFO byly stanoveny takto

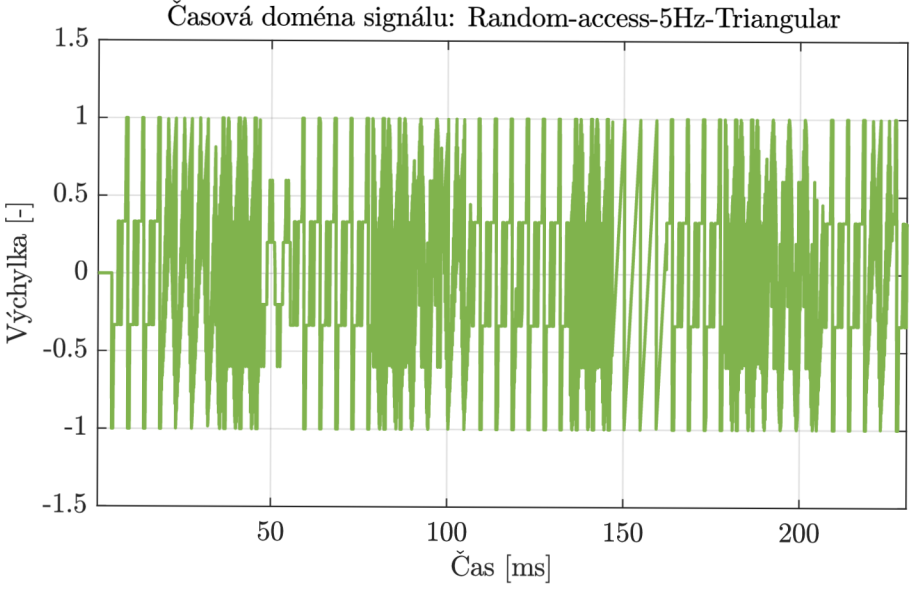
- frekvence: $f_{\texttt{LFO}} = 5\,\text{Hz}$,
- hloubka modulace *Depth*: $D = 1$,
- prvotní signál: Walshovy funkce generované 3 děličkami.



Obr. 3.31: Detailní pohled na problém s degradací dat v metodě *Random-access*.



Obr. 3.32: Průběh vyčítaný metodou *Random-access* (LFO-sinusový signál).

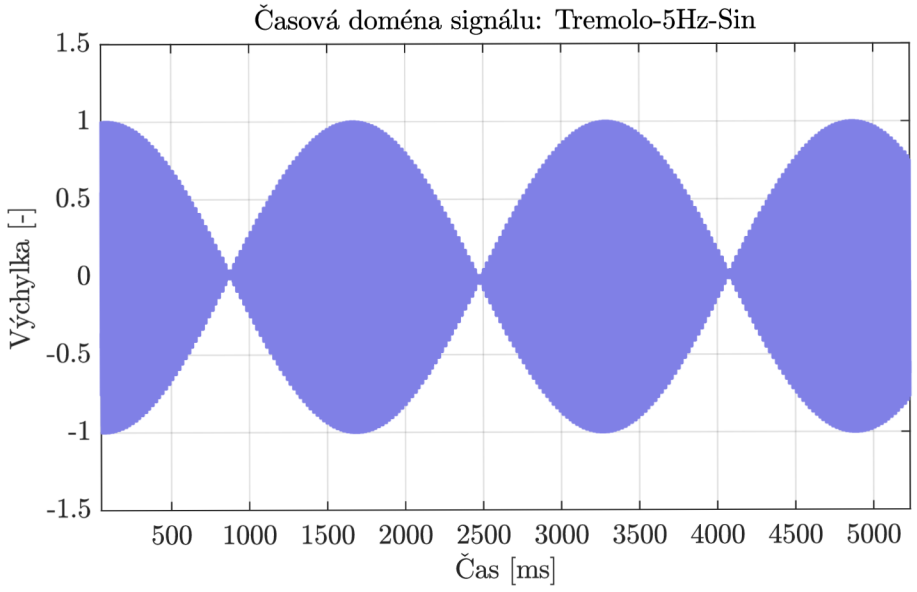


Obr. 3.33: Průběh vyčítaný metodou *Random-access* (LFO-trojúhelníkový signál).

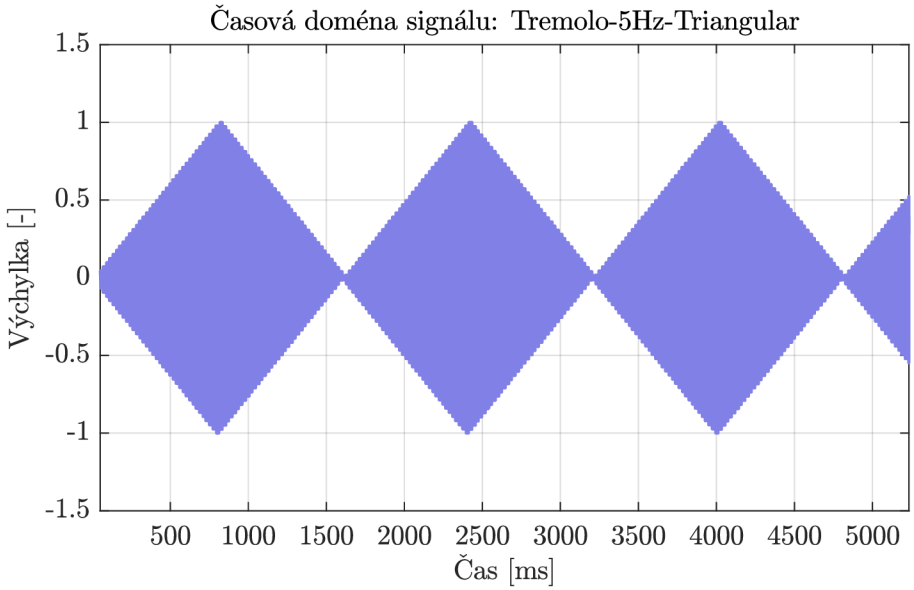
3.3.5 Efekt tremolo

Uživatel má přístup k jednoduchému efektu tremolo, který byl testován ve stejném uspořádání parametrů LFO jako předchozí testy metody *Random-access*. Pro rekapitulaci

- Frekvence: $f_{\texttt{LFO}} = 5\,\text{Hz}$,
- hloubka modulace *Depth*: $D = 1$.



Obr. 3.34: Zobrazení modulace signálu efektem tremolo-sinusový signál.

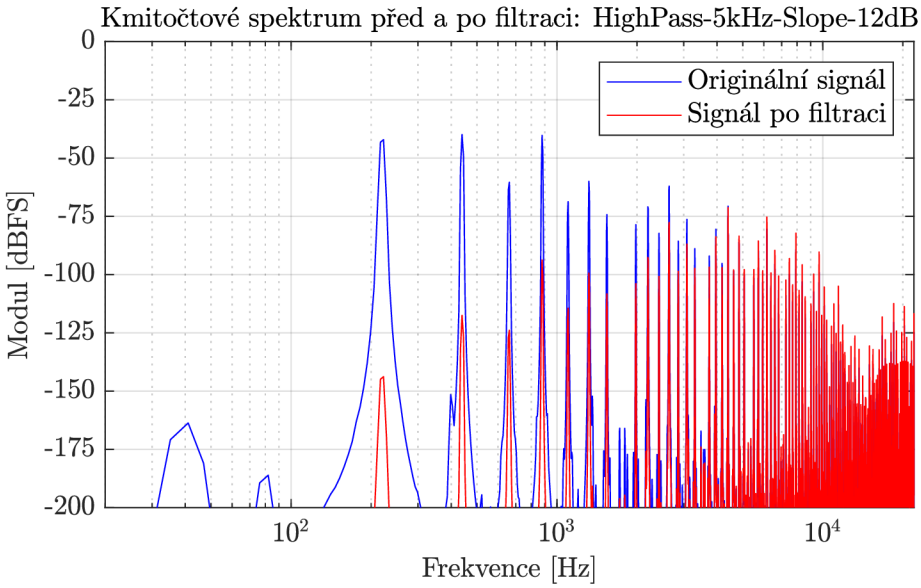


Obr. 3.35: Zobrazení modulace signálu efektem tremolo-trojúhelníkový signál.

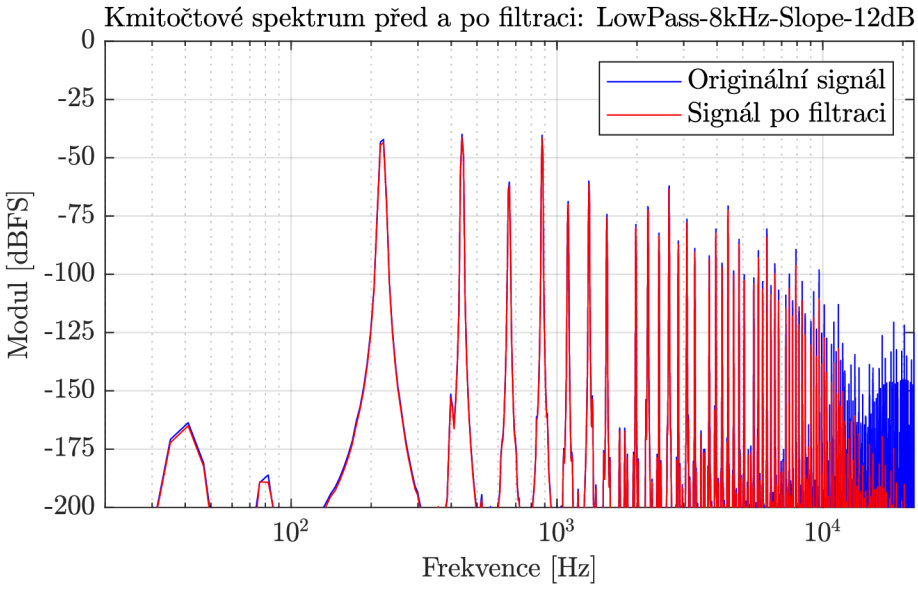
3.3.6 Filtrace

Modul subtraktivní syntézy byl testován na referenčním signálu s průběhem Walshových funkcí generovaném třemi děličkami (viz obr.3.7) s frekvencí $f = 440\,\mathrm{Hz}$. Níže jsou zobrazeny nejreprezentativnější příklady výsledků testování. Jednotlivé hodnoty parametrů filtrů pro testování byly následující

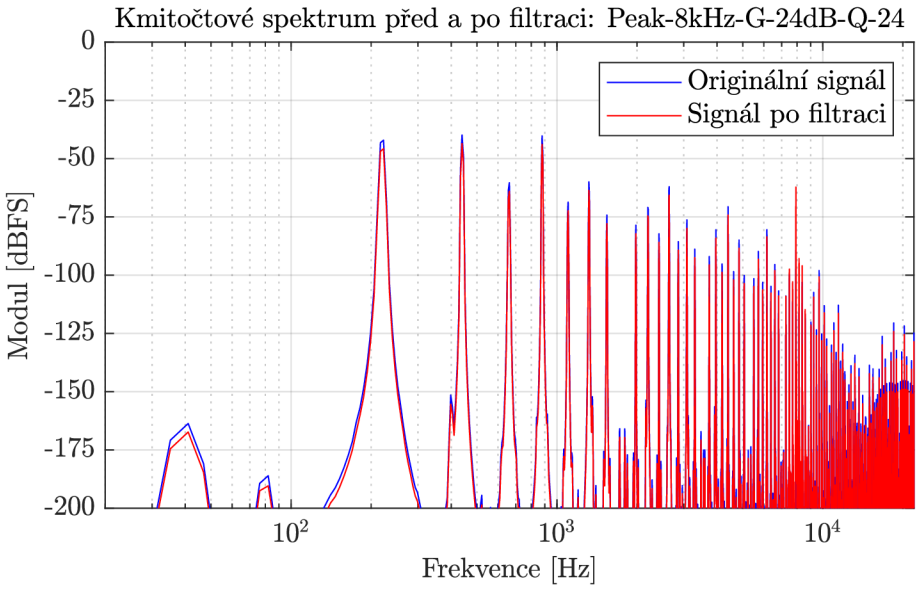
- filtr typu horní propust (High Pass)
  - mezní kmitočty: $f_c = 1\,\mathrm{kHz}$ a $f_c = 5\,\mathrm{kHz}$,
  - strmost filtru *Slope*: 12 dB/okt.
- filtr typu dolní propust (Low Pass)
  - mezní kmitočty: $f_c = 3\,\mathrm{kHz}$ a $f_c = 8\,\mathrm{kHz}$,
  - strmost filtru *Slope*: 12 dB/okt.
- filtr typu Peak
  - mezní kmitočty: $f_c = 2\,\mathrm{kHz}$, $f_c = 4\,\mathrm{kHz}$ a $f_c = 8\,\mathrm{kHz}$,
  - činitel jakosti: $Q = 24$,
  - zesílení filtru: $G = +24\,\mathrm{dB}$.



Obr. 3.36: Filtrace horní propustí s mezním kmitočtem 5 kHz.



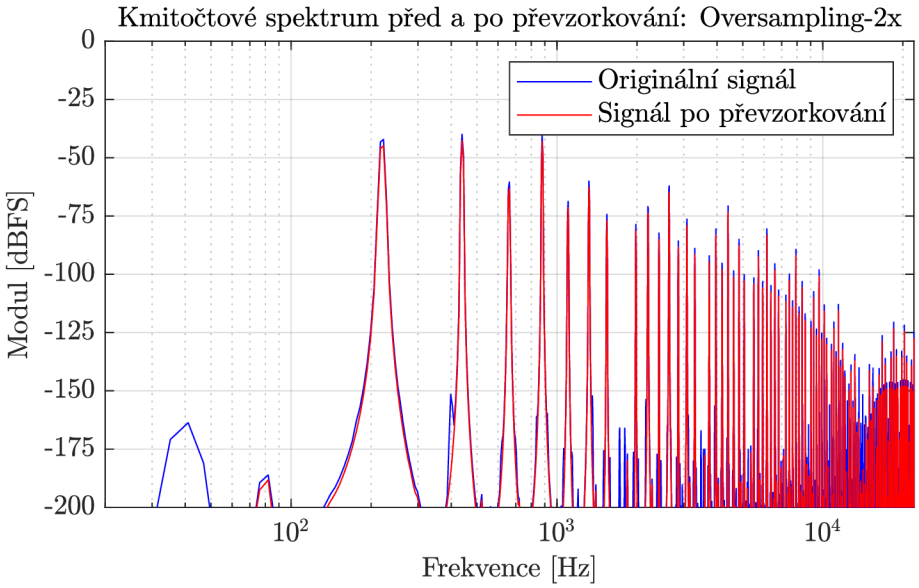
Obr. 3.37: Filtrace dolní propustí s mezním kmitočtem 8 kHz.



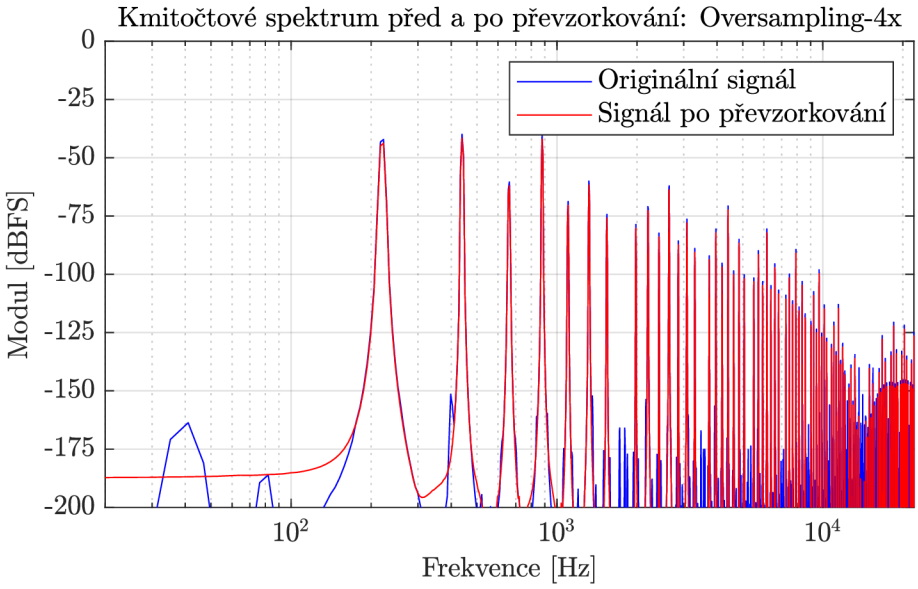
Obr. 3.38: Filtrace filtrem typu Peak s mezním kmitočtem 8 kHz.

3.3.7 Převzorkování

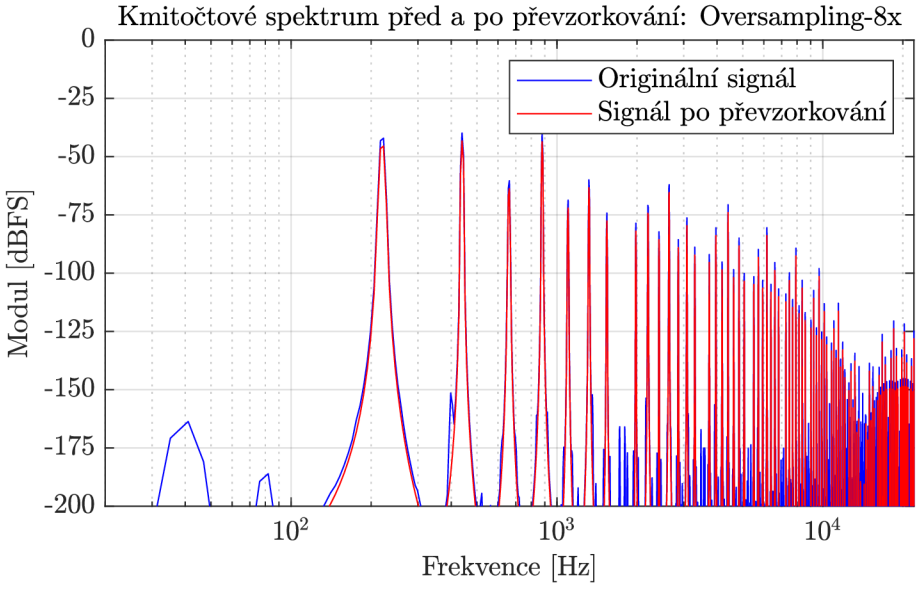
V neposlední řadě bylo testováno převzorkování signálového procesu. Jak již bylo popsáno v kapitole 3.2.1 implementace podporuje převzorkování až do 16násobku původní vzorkovací frekvence. Při aplikaci převzorkování je patrný úbytek aliasingových spektrálních složek. Testování proběhlo pro stejný referenční signál jako předchozí testování filtrace a nastavení metody *Random-access*.



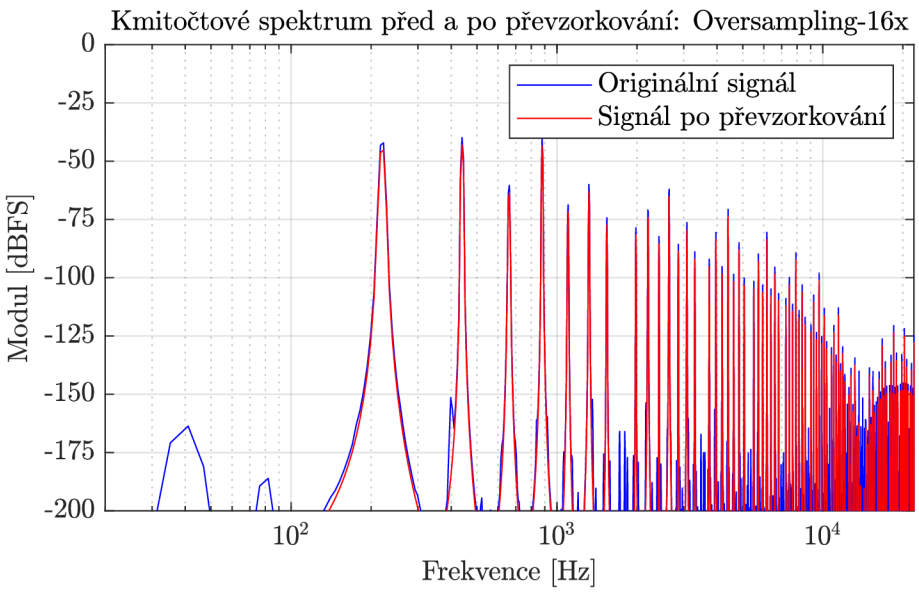
Obr. 3.39: Dvojnásobné převzorkování syntezátoru.



Obr. 3.40: Čtyřnásobné převzorkování syntezátoru.



Obr. 3.41: Osminásobné převzorkování syntezátoru.



Obr. 3.42: Šestnáctinásobné převzorkování syntezátoru.

Závěr

V této diplomové práci byly popsány základní principy generátorů zvukových signálů použitých v hybridních syntezátorech, a to od elementárních prvků až po celkovou funkci jednotlivých metod a způsobů generování signálů včetně rozsahu jejich možností, omezení a rizik použití. Na základě této teorie byly vytvořeny skripty v prostředí `Matlab`, které zobrazují výsledné i průběžné stavy generovaných signálů v reálných obvodech. Na základě těchto skriptů byl následně naprogramován hybridní syntezátor v prostředí JUCE v jazyce C++.

V první části práce byla rozebrána teorie různých typů klopných obvodů, čítačů, multiplexerů a demultiplexerů. Na základě logiky těchto obvodů byly individuálně popsány generátory zvukových signálů jako jsou generátory horní oktávy, Walshových funkcí, pilového průběhu nebo generátory využívající čtení z paměti. Následně se přešlo k popisu vytvořených realizací demonstrativních skriptů, které byly napsány v prostředí `Matlab`.

V části realizaci skriptů bylo vytvořeno několik prvotních stavebních kódů pro stanovení základny generátorů pracujících na čítačích (generátor hodinového impulsu) a generátorů využívající výčtu dat z paměti (vytvoření tabulky dat). Posléze byly vytvořeny skripty pro demonstrování funkčních celků jednotlivých generátorů zvukového signálu. V těchto skriptech může uživatel volit z několika možných proměnných parametrů v závislosti na vybraném typu generátoru. V neposlední řadě byl vytvořen skript generující signály na základě čtení z paměti za použití LFO, který umožňuje atypické vyčítání oproti klasickým principům generátorů využívajících čtení z paměti.

V navazující části práce byla vytvořena samostatně fungující aplikace ve formátu VST3 v prostředí JUCE, která implementuje individuální logiku generátorů do jedné kompaktní verze digitálního hybridního syntezátoru včetně základních nástrojů přeladění, převzorkování a zesílení, resp. zeslabení signálového procesu. V aplikaci má uživatel možnost měnit počet děliček, který má dopad na tvar průběhů generovaných signálů Walshových funkcí a pilového signálu. Dále může volit mezi způsoby vyčítání *Wavecycle* a *Wavetable*, a to konkrétně metodami *Single-cycle*, *Multi-cycle*, *Swept* a *Random-access*. V neposlední řadě má uživatel možnost aktivovat modul subtraktivní syntézy, která zahrnuje několik základních filtrů a nebo efekt tremolo. V rámci realizace aplikace proběhlo i její testování, které je zahrnuto v poslední kapitole této práce.

Literatura

[1] SCHIMMEL, Jiři. *Hybridní syntezátory a sekvencery.* kurz: Hudební elektronika. Brno: Vysoké učení technické v Brně, 2022.

[2] RUSS, Martin. *Sound synthesis and sampling.* Druhé vydání. Amsterdam: Elsevier, 2004. ISBN 0-240-51692-3.

[3] SÝKORA, Rudolf; KRUTÍLEK, František a VČELAŘ, Jaroslav. *Elektronické hudební nástroje a jejich obvody.* Praha: SNTL-Nakladatelství technické literatury, 1981.

[4] TICHÝ, Milan a ŠÍCHA, Miloš. *Elektronické zpracování signálů.* [online]. Praha: Karolinum, nakladatelství UK, 1998. Dostupné z URL:<`https://physics.mff.cuni.cz/kfpp/skripta/elektronika/`>. [cit. 4. 12. 2023].

[5] MORAVČÍK, Lukáš. *Úlohy pro fyzikální praktikum - Operační zesilovače.* [online]. Brno: Masarykova univerzita, Pedagogická fakulta, 2007. Dostupné z URL: <`https://is.muni.cz/th/hmgje/`>. [cit. 4. 12. 2023].

[6] *Napěťový komparátor.* [online]. Elektronická učebnice. Olomouc: ELUC, 2015. Dostupné z URL: <`https://eluc.kr-olomoucky.cz/verejne/lekce/695`>. [cit. 4. 12. 2023].

[7] *About JUCE.* [online]. Internet Archive, 24. května 2014. Dostupné z URL: <`https://web.archive.org/web/20130807012255/http://www.juce.com/about-juce`>. [cit. 15. 5. 2024].

[8] LEITGEB, David. *Realizace zvukového efektu Waveshaper.* Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2022. [cit. 15. 5. 2024].

[9] *Dokumentace JUCE k třídě GenericAudioProcessorEditor.* [online] Dostupné z URL: <`https://docs.juce.com/master/classGenericAudioProcessorEditor.html`>.[cit. 15. 5. 2024].

[10] *Dokumentace JUCE k třídě dsp::Oversampling.* [online] Dostupné z URL: <`https://docs.juce.com/master/classdsp_1_1Oversampling.html`>.[cit. 15. 5. 2024].

[11] SCHIMMEL, Jiří. *Studiová a hudební elektronika.* Druhé vydání. Brno: Vysoké učení technické v Brně, 2015. ISBN: 978-80-214-4452-2. [cit. 15. 5. 2024].

[12] *Dokumentace JUCE k třídě dsp::Panner*. [online] Dostupné z URL: <https://docs.juce.com/master/classdsp_1_1Panner.html>.[cit. 15. 5. 2024].

[13] *Dokumentace JUCE k metodě* ***applyGainRamp()*** . [online] Dostupné z URL: <https://docs.juce.com/master/classAudioBuffer.html#ab0542e5b626b36087f0054e795695682>.[cit. 15. 5. 2024].

[14] WILCZEK, Jan. *Wavetable Synth Plugin in JUCE C++ Framework Tutorial.* [online]. nahráno 24. prosince, 2021. Dostupné z URL: <https://thewolfsound.com/sound-synthesis/wavetable-synth-plugin-in-juce/>. [cit. 15. 5. 2024].

[15] WILCZEK, Jan. *Wavetable Synthesis Algorithm Explained.* [online]. nahráno 13. července, 2021. Dostupné z URL: <https://thewolfsound.com/sound-synthesis/wavetable-synthesis-algorithm/#computing-a-waveform-value-from-the-wave-table>. [cit. 15. 5. 2024].

[16] WILCZEK, Jan. *Sine, Saw, Square, Triangle, Pulse: Basic Waveforms in Synthesis and Their Properties.* [online]. nahráno 26. června, 2022. Dostupné z URL: <https://thewolfsound.com/sine-saw-square-triangle-pulse-basic-waveforms-in-synthesis/>. [cit. 15. 5. 2024].

[17] BARANIUK, Richard et al. *Common Fourier Series* [online]. Kalifornie: Rice University, the UC Davis Library, 2024. Dostupné z URL: <https://eng.libretexts.org/Bookshelves/Electrical_Engineering/Signal_Processing_and_Modeling/Signals_and_Systems_(Baraniuk_et_al.)/06%3A_Continuous_Time_Fourier_Series_(CTFS)/6.03%3A_Common_Fourier_Series>.[cit. 16. 5. 2024].

[18] SCAVONE, Gary. *Midi/Frequency Conversion.* [online] Kanada: McGill University, Department of Music Research Dostupné z URL: <https://www.music.mcgill.ca/~gary/307/week1/node28.html>.[cit. 16. 5. 2024].

[19] *Dokumentace JUCE k třídě ADSR.* [online] Dostupné z URL: <https://docs.juce.com/master/classADSR.html>.[cit. 16. 5. 2024].

[20] WOLF, Alan. *DIY Synth 2: Common Wave Forms* [online]. nahráno 19. května, 2012. Dostupné z URL: <https://blog.demofox.org/2012/05/19/diy-synthesizer-chapter-2-common-wave-forms/>. [cit. 16. 5. 2024].

[21] SCHIERMEYER, Charles. *Learn Modern C++ by Building an Audio Plugin (w/ JUCE Framework) - Full Course.* YouTube [online].

20. 5. 2021 Dostupné z URL: <https://www.youtube.com/watch?v=i_Iq4_Kd7Rc>.[cit. 17. 5. 2024].

[22] *Dokumentace JUCE k třídě FilterDesign*. [online] Dostupné z URL: <https://docs.juce.com/master/structdsp_1_1FilterDesign.html#aa8d0302bc17b600465203fe06398ce76>.[cit. 17. 5. 2024].

[23] *Dokumentace JUCE k třídě AudioProcessorValueTreeState*. [online] Dostupné z URL: <https://docs.juce.com/master/classAudioProcessorValueTreeState.html>.[cit. 16. 5. 2024].

[24] DE JONG, Bram. *plug-in S(m)exoscope*. [online]. aktualizováno 26. 6. 2023, Dostupné z URL: <https://github.com/bdejong/smartelectronix>. [cit. 18. 5. 2024].

Seznam symbolů a zkratek

KO klopné obvody

BKO bistabilní klopné obvody

SKO Schmittův klopný obvod

LFO nízkofrekvenční oscilátor – Low Frequency Oscilator

VCO analogový oscilátor – Voltage Controlled Oscillator

DCO digitálně řízený oscilátor – Digital Controlled Oscilator

GUI grafické uživatelské rozhraní – Graphical User Interface

VST Virtual Studio Technology

AAX Avid Audio eXtension

IIR nekonečná impulsní odezva – Infinite Impulse Response

DAW hudební software – Digital Audio Workstation

JUCE Jules' Utility Class Extensions

A Obsah elektronické přílohy

- `Diplomová práce - Šimon Prokop` ..... kořenový adresář přiloženého archívu
  - `JUCE` ..... adresář obsahující soubory související s hybridním syntezátorem
    - `Generované signály` ..... adresář s testovanými zvukovými ukázkami
      - `Filtry`
        - `Demonstrace_HP_1kHz_12dB.wav` ..... Dílčí zvukové ukázky
        - `Demonstrace_HP_5kHz_12dB.wav`
        - `Demonstrace_LP_3kHz_12dB.wav`
        - `Demonstrace_LP_8kHz_12dB.wav`
        - `Demonstrace_Peak_2kHz_G24dB_Q24.wav`
        - `Demonstrace_Peak_4kHz_G24dB_Q24.wav`
        - `Demonstrace_Peak_8kHz_G24dB_Q24.wav`
        - `Demonstrace_Referencni_signal.wav`
      - `Multicycle`
        - `Demonstrace_Multicycle_1.wav`
        - `Demonstrace_Multicycle_2.wav`
        - `Demonstrace_Multicycle_3.wav`
        - `Demonstrace_Multicycle_4.wav`
        - `Demonstrace_Multicycle_5.wav`
        - `Demonstrace_Multicycle_6.wav`
        - `Demonstrace_Multicycle_7.wav`
        - `Demonstrace_Multicycle_8.wav`
        - `Demonstrace_Multicycle_9.wav`
      - `Oversampling`
        - `Demonstrace_Oversampling_16x.wav`
        - `Demonstrace_Oversampling_2x.wav`
        - `Demonstrace_Oversampling_4x.wav`
        - `Demonstrace_Oversampling_8x.wav`
      - `Random-access`
        - `Demonstrace_Random-access_10Hz_Saw.wav`
        - `Demonstrace_Random-access_10Hz_Sin.wav`
        - `Demonstrace_Random-access_10Hz_Square.wav`
        - `Demonstrace_Random-access_10Hz_Triangular.wav`
      - `Swept`
        - `Demonstrace_Swept_10Hz_Saw.wav`
        - `Demonstrace_Swept_10Hz_Sin.wav`
        - `Demonstrace_Swept_10Hz_Triangular.wav`
        - `Demonstrace_Swept_5Hz_Saw.wav`
        - `Demonstrace_Swept_5Hz_Sin.wav`
        - `Demonstrace_Swept_5Hz_Triangular.wav`
      - `Tremolo`
        - `Demonstrace_Tremolo_5Hz_Pila.wav`
        - `Demonstrace_Tremolo_5Hz_Sinus.wav`
        - `Demonstrace_Tremolo_5Hz_Triangular.wav`
      - `Walsh`

- Demonstrace_Walsh_1.wav
- Demonstrace_Walsh_2.wav
- Demonstrace_Walsh_3.wav
- Demonstrace_Walsh_4.wav
- WalshD
  - Demonstrace_WalshD_1.wav
  - Demonstrace_WalshD_2.wav
  - Demonstrace_WalshD_3.wav
  - Demonstrace_WalshD_4.wav
- WalshSaw
  - Demonstrace_WalshSaw.wav
  - Demonstrace_WalshSaw_1.wav
  - Demonstrace_WalshSaw_2.wav
  - Demonstrace_WalshSaw_3.wav
  - Demonstrace_WalshSaw_4.wav
  - Demonstrace_WalshSaw_5.wav

- Grafické ukázky ..... adresář grafů testovaných zvukových ukázek
  - Filtry
  - Multicycle
  - Oversampling
  - Random-access
    - Detail
  - Swept
    - Detail
  - Tremolo
  - Walsh
  - WalshD
  - WalshSaw
- Zdrojový kód ..... adresář zdojového kódu JUCE
  - Hybrid_synth.jucer
  - Hybrid_synth.vst3 ..... soubor s VST3 plug-in modulem
  - JuceLibraryCode ..... použité moduly z prostředí JUCE
  - Source ..... zdrojový kód hybridního syntezátoru
    - Filters.cpp
    - Filters.h
    - HelpFunctions.cpp
    - HelpFunctions.h
    - LFO.cpp
    - LFO.h
    - Oscillator.cpp
    - Oscillator.h
    - PluginEditor.cpp
    - PluginEditor.h
    - PluginProcessor.cpp
    - PluginProcessor.h
    - Synth.cpp

- Synth.h
- SynthParameters.h
- Waveform.cpp
- Waveform.h

- Matlab .... adresář obsahující soubory související s demonstračními skripty
  - Grafy ..... adresář s grafy generovanými Matlab skripty
    - Generator obdelnikoveho prubehu (MG_2.m)
    - Generator piloveho prubehu
    - ROM
    - Walshovy funkce
    - Wavecycle_Wavetable
  - Zdrojový kód ..... adresář obsahující jednotlivé demonstrační skripty
    - Basic_Wavecycle.m
    - Basic_Wavetable.m
    - Generator_piloveho_prubehu.m
    - GRAPHICS.m ..... skript upravující grafický vzhled
    - MG_2.m
    - ROM_LFO.m
    - Tabulka_1.m
    - Walshovy_funkce.m
    - Walshovy_funkce_typ_D.m

Příloha ke stažení v souboru .ZIP je dostupná na stránce

<https://drive.google.com/drive/folders/1oJo4H-7w_A0P7B4xksf5TbwA7b7RvXS9?usp=drive_link>