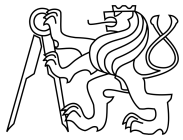
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Multiplatformní zobrazovací software pro elektroencefalografii
Student:	Bc. Martin Bárta
Vedoucí:	Ing. Petr Ježdík, Ph.D.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2017/18

Pokyny pro vypracování

Cílem diplomové práce je rozšíření a doladění prototypu multiplatformní aplikace pro vizualizaci křivek elektroencefalografie, který byl vytvořen v rámci BP Martinem Bártou v roce 2015.
Funkční požadavky:
Zobecnění filtrace signálu: uživatel má možnost definovat změnu amplitudy v určitých frekvenčních pásmech.
Integrace spike detektoru a vizualizace výsledků této analýzy v EEG zobrazení.
Synchronizace časové pozice a vzorků na centimetr mezi více instancemi aplikace na jednom počítači a na počítačích v lokální síti.
Výkonnostní požadavky na aplikaci:
Cílem je zvládat zobrazení až 10 sekund signálu o 8 kHz s 2048 kanály.
Typický objem dat 10 s na stránku, 128 kanálů, 1 kHz.
Je vyžadována podpora operačních systémů Microsoft Windows 7, 8, 10 a Linux Ubuntu 14 a 16. Je žádoucí možnost použití programu ve virtuálním prostředí. Řešení řádně otestujte a zdokumentujte.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 2. ledna 2017

Prohlášení

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

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

V Praze dne 16. února 2018 .....

Odkaz na tuto práci

Bárta, Martin. *Multiplatformní zobrazovací software pro elektroencefalografii.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Tato práce popisuje rozšíření programu pro zobrazování a analýzu EEG dat se zaměřením na epilepsii. Mezi podporované funkce patří: pokročilá filtrace, automatické detekční analýzy, *montáže* – programovatelné kombinace kanálů, atd. Zpracování signálu a vykreslování využívá hardwarovou akceleraci. Program podporuje soubory typu GDF, EDF a MAT, což umožňuje přímé použití ve výpočetním prostředí Matlab.

Klíčová slova EEG, epilepsie, hardwarová akcelerace, GDF, EDF, biologické signály, zpracování digitálních signálů, Matlab

Abstract

This thesis describes the extension of a program for visualization and analysis of EEG data with focus on epilepsy. Among the supported functions belong: advanced filtering, automatic detection analyzes, *montages* – programmable combinations of channels, etc. Signal processing and rendering use HW acceleration. The program supports file formats GDF, EDF and MAT, which enables direct usage in Matlab.

Keywords EEG, epilepsy, hardware acceleration, GDF, EDF, biological signals, digital signal processing, Matlab

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů

Úvod

Tato práce patří do oblasti lékařství, konkrétně potom do odvětví neurologie. Neurologie se zabývá fungováním lidské nervové soustavy a mozku. Mozek představuje nejen nejdůležitější, ale také nejsložitější orgán. Po celém světě probíhá intenzivní výzkum v této oblasti s cílem rozkrýt záhady této pozoruhodné části lidského těla.

Jednou z možností jak lépe něčemu porozumět je sledování a kvantifikování pomocí měření. V neurologii se měří mozková aktivita pomocí tzv. EEG (Elektroencefalogramu).

EEG

EEG[36] je záznam změny elektrického napětí způsobeného mozkovou aktivitou. Tento záznam je pořízen pomocí elektroencefalografu, přístroje snímajícího napětí na připojených elektrodách. Elektroencefalograf může zaznamenávat data pro pozdější detailnější analýzu, nebo jsou hodnoty měření zobrazovány přímo na obrazovce.

Tradičně se EEG zaznamenává na papír. S rozmachem informačních technologií se začala EEG měření zpracovávat jako digitální signál a ukládat do datových souborů. Tento přístup přináší spoustu výhod:

- k zobrazování není potřeba specializované zařízení – stačí obyčejný osobní počítač
- zobrazení lze interaktivně ovládat
- signální data lze programově manipulovat
- signál lze filtrovat a tím odstranit artefakty (nežádoucí signál vzniklý vnějšími rušivými vlivy jako jsou např. hluk střídavého napětí, tzv. brum, nebo svalová aktivita)

EEG dnes patří mezi běžná vyšetření. Používá se k monitorování a diagnostice různých chorob a stavů, např.

- epilepsie,
- poruch spánku nebo
- kóma.

EEG se používá i k nelékařským účelům. Jedním z příkladů je tzv. neuromarketing[39], který zkoumá reakce mozku na určitý druh reklamy.

Zadavatel

Zadavatelem (někdy v textu označován také jako zákazník) této práce je ISARG[8], výzkumná skupina zaměřená na analýzu EEG signálů pacientů trpících epilepsií. Členy této skupiny jsou výzkumní pracovníci ČVUT (včetně vedoucího práce) a Univerzity Karlovi. ISARG spolupracuje s Klinikou dětské neurologie FN Motol.

Kapitola 1

Cíl práce

Cílem práce je rozšíření a vylepšení aplikace vyvinuté v předešlé BP[21].

Tato aplikace si získala kódové označení „Alenka“. Já budu toto jméno používat, když se budu odkazovat na aplikaci, která je předmětem obou prací, bakalářské i diplomové.

1.1 Stav projektu před začátkem práce na DP

Prototyp aplikace z BP nabízí tyto základní funkce:

- hardwarově akcelerované EEG zobrazení s podporou zobrazení anotací
- podpora jednoho datového formátu – GDF
- správa metadat uložených v hlavním i rozšiřujících souborech
- základní 3 typy filtrace
- mechanizmus montáží: možnost definovat vzorce pro vytvoření nového signálu kombinací kanálů původního signálu uloženém v souboru

Během práce s programem bylo objeveno několik nedostatků a chyb. Některé problémy byly vyřešeny už před začátkem práce na DP, některé budu řešit nyní.

Pro příklad uvedu problém špatného řešení alokace grafické paměti. Paměť se alokovala najednou při inicializaci, což vedlo k pádu v případě špatného nastavení limitu. Zároveň to zabraňovalo spuštění několika paralelních oken najednou.

Jedním z cílů této DP by také mělo být zjednodušit práci s projektem, aby ostatní – jiní studenti, nebo členové open-source komunity – mohli pokračovat ve vývoji aplikace. Toho mohu dosáhnout např. zjednodušením procesu překladu a distribuce, který byl slabou stránkou prototypu.

1.2 Rozbor zadání

Zadání obsahuje tři funkční požadavky:

- Zobecnění filtrace signálu (řešeno v sekci 2.3)
- Integrace spike-detektoru (řešeno v sekci 2.2)
- Synchronizace časové pozice (řešeno v sekci 2.1)

Tyto požadavky jsou nezávislé a budu se jim věnovat jednotlivě a postupně.

Dále zadání uvádí jeden konkrétní nefunkční požadavek na výkon. Tento požadavek stanovuje spodní limit na objem dat, který musí aplikace zvládat na všech podporovaných platformách. Zhodnocení výkonu relevantní pro tento požadavek je prezentováno v sekci 3.4.

Těmto požadavkům se věnuji detailně a do hloubky. V kapitola 2 o analýze, návrhu a realizaci ale také zmíním některé ostatní změny a nové funkce, které jsem implementoval nad rámec zadání.

Nakonec zmíním, že Alenka využívá multiplatformní GUI knihovnu Qt[33], což dovoluje aby běžela na všech operačních systémech vyžadovaných v zadání.

Kapitola 2

Analýza, návrh a realizace

2.1 Synchronizace zobrazení

V této sekci popisuji naplnění následujícího funkčního požadavku ze zadání: *Synchronizace časové pozice a vzorků na centimetr mezi více instancemi aplikace na jednom počítači a na počítačích v lokální síti.*

Od schválení zadání vyšlo najevo, že synchronizace vzorků na centimetr se už nepovažuje za potřebnou funkci. Tudíž jsem implementoval pouze synchronizaci časové pozice.

Hlavním úkolem Alenky je zobrazovat záznamy signálu. Aplikace nabízí spoustu způsobů, jak zobrazení přizpůsobit, např.:

- změnou přiblížení (zoomu)
- zapnutím filtrace
- zapnutím montáže
- vypnutí části kanálu v případě, že jich soubor obsahuje příliš

Zajímavé je sledovat více těchto zobrazení paralelně. Srovnání mezi různými pohledy na tu samou informaci může odhalit více, než prozkoumání jednoho po druhém. Navíc můžeme chtít využít více monitorů (potenciálně i na různých počítačích) pro maximální efektivitu práce.

Zásadní je ale aby byla zobrazení synchronizovaná, tedy aby jsme věděli, že se díváme na stejný moment v záznamu. Bez podpory ze strany aplikace by musel uživatel pracně ovládat více oken, případně i více počítačů simultánně aby tohoto docílil. Navíc neustálé kontrolování, zda je na správném místě, by bylo značně rozptylující.

Nová funkce navržená v této části se snaží tento problém řešit propojením libovolného počtu běžících instancí programu. Toto by mělo odstranit jakou-

koliv manuální režii ze strany uživatele v průběhu inspekce záznamu spojenou s více paralelními zobrazeními.

2.1.1 Analýza

Tato funkce vyžaduje komunikaci mezi více instancemi běžícího programu. Nejedná se pouze o více paralelních procesů. V tomto případě by stačilo otevření systémového soketu. Jelikož je vyžadována komunikace i na lokální síti, je potřeba využití některého ze síťových protokolů.

Zde máme několik variant. Já jsem se rozhodl pro použití protokolu WebSocket[24]. WebSocket je rozšíření TCP spojení. Stejně jako TCP umožňuje obousměrnou komunikaci a zaručuje konzistentnost a správnost posílaných dat. Navíc ale umožňuje posílat data v „balíčcích“ s danou délkou, se kterými se dá pracovat jako se zprávami. Toto zjednodušuje interpretaci dat na straně příjemce, kdy už nemusíme pracovat pouze s proudem bytů.

Navíc Qt implementuje WebSocket, což znamená žádné další závislosti na knihovnách třetí strany.

2.1.2 Návrh

Síťová komunikace vyžaduje definování dvou rolí:

- server a
- klient

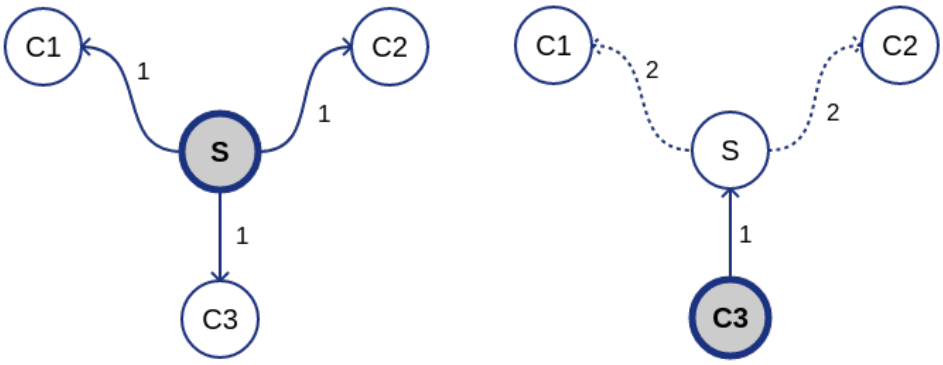
To v našem případě znamená, že jedna aplikace se zhostí role serveru a bude naslouchat a čekat na připojení od dalších instancí aplikace, které budou představovat klienty.

Server může být v jeden okamžik pouze jeden. Identita serveru v systému je určena portem, na kterém server naslouchá. Když se pokusí druhý server otevřít soket se stejným portem, doje k chybě. To za nás zařídí systém, respektive implementace protokolu WebSocket v Qt. Je dovoleno spustit více paralelních serverů, každý s jiným portem.

Server a klient jsou téměř ve všem rovnocenní: oba poskytují zobrazení a vyvolávají synchronizační události. Klient a server se liší pouze v tom, jak komunikují se svým okolím.

Na obrázku 2.1 je znázorněno, jak tato komunikace probíhá. Jsou dovoleny pouze dva případy:

- Synchronizační událost vzniká na serveru. Ten vysílá zprávu všem klientům. Ti zprávu zpracují a komunikace končí. Čekáme na další událost.
- Synchronizace začíná u některého z klientů. Ten vysílá zprávu na server. Server přepošle zprávu všem ostatním klientům. Komunikace končí a čeká se na další událost.



Obrázek 2.1: Dva případy dovolené komunikace mezi serverem (S) a třemi klienty (C1, C2 a C3)

Všechny ostatní varianty (jako například odeslání zprávy serverem zpět klientovi) jsou zakázány, aby se předešlo zacyklení.

2.1.2.1 Návrh uživatelského rozhraní

Uživatelské rozhraní musí uživateli dovolovat výběr role aplikace – tedy server, nebo klient – a zabránit mu ve změně jakmile je komunikace navázána.

Klient potřebuje k navázání komunikace dvě informace:

- IP adresu počítače, na kterém běží serverová aplikace, a
- výše zmíněný port

Uživatel klienta musí být schopný jednoduše zjistit tyto dvě informace, nejlépe přímo v rozhraní ovládajícím synchronizaci. Dále musí být možnost specifikovat port v serverovém módu.

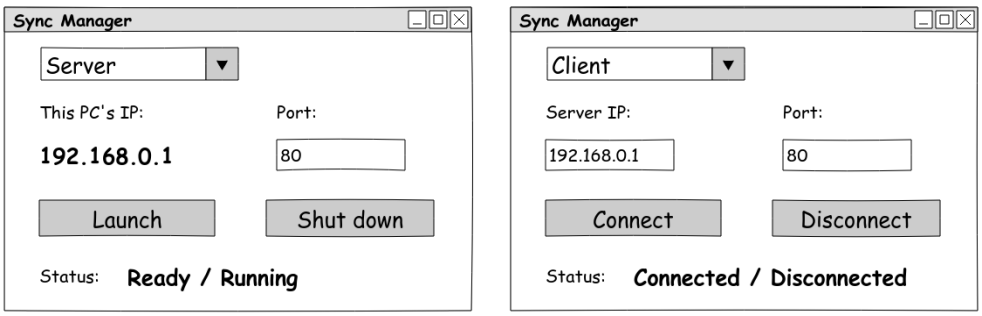
Obrázek 2.2 ukazuje návrh uživatelského rozhraní. Jedná se o jednoduché dialogové okno. Prvním ovládacím prvkem, lze přepínat mezi server a klient módem. Rozhodl jsem se pro jediné okno, které se mění podle potřeby, což nedovoluje uživateli spustit server a klienta najednou.

Textová pole pro zadávání IP adresy a portu se stanou neaktivní ve chvíli, kdy úspěšně navážeme komunikaci.

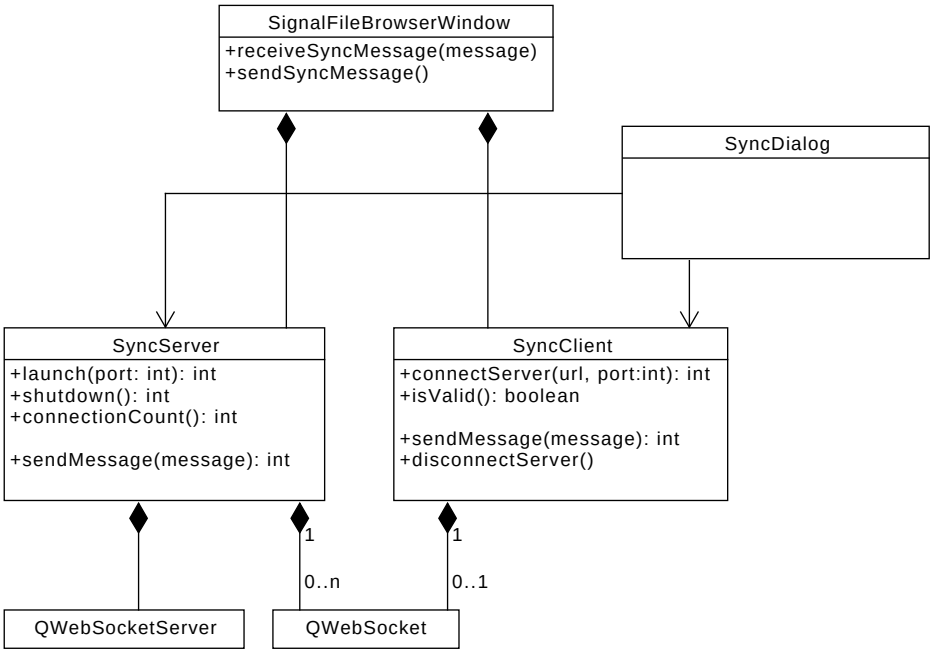
Nakonec dialog informuje uživatele o stavu a případném vynuceném odpojení druhou stranou pomocí textu statusu.

2.1.2.2 Objektový návrh

Vytvořil jsem dvě třídy pro správu komunikace pro server a klienta. Tyto dvě třídy odstiňují zbytek programu od logiky a konkrétní implementace komuni-



Obrázek 2.2: Návrh uživatelského rozhraní dialogu pro synchronizaci



Obrázek 2.3: UML class diagram pro synchronizaci zobrazení

kace. Samy přitom s obsahem posílaných zpráv nijak nepracují. Synchronizační dialog používá tyto dvě třídy přímo, ale nevlastní je.

Třída `SignalFileBrowserWindow` vlastní (ať už přímo, či nepřímo) všechny ovládací prvky. Proto je to dobré místo pro přidání kódování a interpretace synchronizačních zpráv.

Obrázek 2.3 ukazuje tento návrh graficky.

2.1.3 Realizace

Synchronizace byla implementována přesně podle návrhu ze sekce 2.1.2.1.

Během realizace byl odhalen jeden nepředvídaný implementační problém a také jeden nedostatek uživatelského rozhraní.

2.1.3.1 Problém se zacyklením

Tento problém se projevuje tak, že zobrazení u obou synchronizovaných aplikacích někdy z ničeho nic začíná „poskakovat“, nebo „odjíždět na stranu“. To je způsobeno opakováním synchronizačních zpráv v nekonečném cyklu.

Protokol komunikace, který jsem definoval v návrhu, je správný. Problém je v tom, že synchronizace vyvolává Qt signál pro nastavení aktuální pozice, na který pak synchronizace sama reaguje.

Signál, který nastavuje hodnotu na stejnou, ale nemá žádný efekt, protože hodnota by se nezměnila, je ignorován. Může ale nastat, že při převodu pozice z celočíselné hodnoty vzorku na desetinné číslo času a poté zpět, dojde k zaokrouhlovací chybě. To způsobí, že hodnota není totožná, a tudíž signál ignorován není.

Navíc, tento problém je závislý na hodnotách, které se účastní převodů: vzorkovací frekvence, zoom zobrazení, pozice *Time Line* atd. To znamená, že problém je zákeřný tím, že je datově závislý. Jako takový byl odhalen až při testování u zákazníka.

Jedním z možných řešení je vyhnout se převodům na desetinná čísla. Jako synchronizační zprávu nemůžeme použít přímo časovou pozici, protože ta je závislá na dalších hodnotách, které definují jak vypadá zobrazení. Jednou z požadovaných vlastností synchronizace je aby jsme mohli použít různá zobrazení v paralelních oknech. Takže nemůžeme použít pouze tuto hodnotu.

Změnit jak se aplikace chová vzhledem ke všem těmto hodnotám, aby se předešlo zaokrouhlování, by bylo značně pracné. Takže se zdá, že tato linie uvažování nikam nevede.

Další možností je nějakým způsobem detekovat, že právě vyvolaný signál změnou pozice má zdroj v poslední zprávě přijaté od serveru. Tudíž se nejedná o událost vyvolanou uživatelem a v tomto případě nemá smysl odesílat synchronizační zprávu.

Bohužel v Qt nelze jednoduše rozlišovat zdroj signálu. Tento „ozvěnový“ signál má ale jinou vlastnost, která ho odlišuje od ostatních: předaná hodnota je velmi blízká hodnotě, kterou jsme právě obdrželi v synchronizační zprávě. Pokud si uložíme poslední přijatou hodnotu ve zprávě, můžeme před odesláním nové zprávy otestovat, zda nová hodnota je velmi blízká té právě obdržené. Pokud ano, můžeme tento signál bezpečně ignorovat, protože požadovaná hodnota je už s vysokou pravděpodobností sdílena mezi všemi synchronizovanými instancemi.

Toto je řešení, které jsem zvolil. Řešení se zdá být efektivní a žádné další problémy v synchronizaci už nevyšli najevo.

2.1.3.2 Synchronizace na *Time Line*

Původně bylo zobrazení synchronizováno podle levého okraje okna. To se ukázalo být značně nešikovné, neboť právě okolí synchronizovaného bodu je tím nejzajímavějším místem zobrazení.

Využil jsem proto už implementovaného nástroje *Time Line*, svislé modré čáry, která slouží jako značka pro indikaci aktuální časové pozice. Tato značka je nastavitelná klávesou `T` a slouží například pro funkci *Go to Event*.

Tato značka se nyní používá jako referenční bod pro synchronizaci.

2.2 Integrace spike-detektoru

V této části se zaměřím na tento požadavek ze zadání: *Integrace spike detektoru a vizualizace výsledků této analýzy v EEG zobrazení.*

Manuální analýza záznamů EEG je velmi pracná záležitost. Jak pokračuje současný trend, kdy se nahrávací technika zlevňuje a zdokonaluje, objem dat se stále zvyšuje. Pracuje se s delšími záznamy, více kanály a vyššími vzorkovacími frekvencemi. Disciplína *high-density* EEG toto dovádí do extrému: je zcela běžné pracovat se stovkami i tisíci kanálů najednou. Toto je už pro člověka zcela neřešitelná situace.

Kde lidé selhávají, počítače přicházejí a zachraňují den. V současné době je nejen v ISARG[8] předmětem intenzivního výzkumu vývoj automatických detekčních algoritmů pro identifikaci vzorů v záznamu EEG charakteristických pro epileptická ložiska.

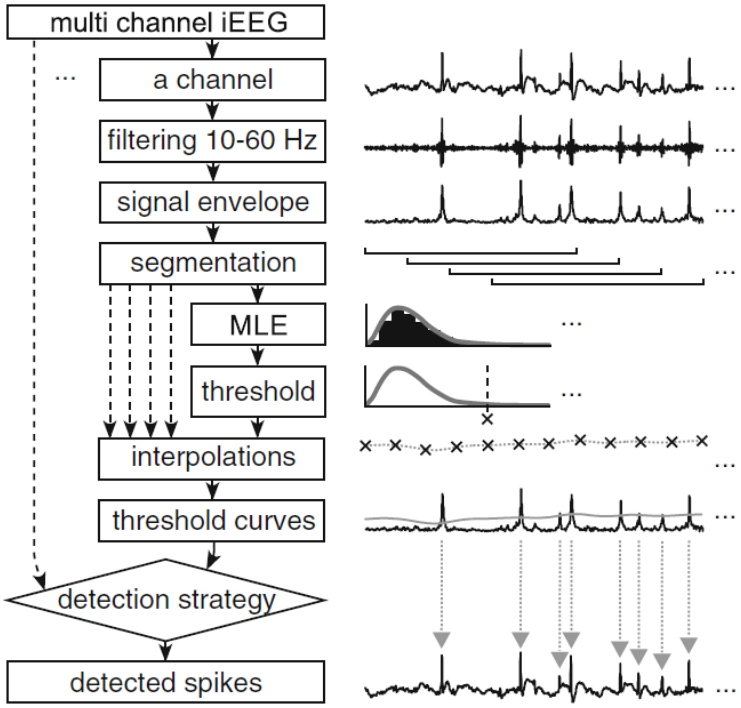
Jedním z takových algoritmů je spike-detektor. Původní podoba spike-detektoru byla představena ve vědecké práci[25], jejíž součástí je i Matlab[12] skript implementující tento algoritmus.

Na obrázku 2.4 můžeme vidět jakým způsobem probíhá analýza spike-detektoru. (Obrázek uvádím pouze pro zajímavost, rozbor algoritmu není předmětem této práce.) Pro příklad typické detekce si všimněte posledního kroku a šipek ukazující na místa, která budou ve výstupu algoritmu.

Na tuto práci navazuje BP[23], ve které bylo úkolem vytvořit samostatnou knihovnu nezávislou na prostředí Matlab. Důvodem bylo to, že skript není jednoduché integrovat do jiného softwaru (jako je v tomto případě Alenka).

2.2.1 Analýza

Vstupem spike-detektoru je čistý signál a jeho frekvence. Výstupem analýzy jsou dvě datové struktury, které jsou popsané v tabulkách 2.1 a 2.2.



Obrázek 2.4: Průběh spike-detektoru (převzato z [25])

Název	**Popis**
pos	Pozice výboje v kanálu. V sekundách od začátku signálu.
dur	Doba trvání s fixní hodnotou $\frac{1}{fs}$.
chan	Kanál ve kterém byl výboj detekován.
con	Typ výboje: 1 - zřejmý, 0,5 - nejednoznačný.
weight	Statistická významnost "CDF".
pdf	Statistická významnost "PDF".

Tabulka 2.1: Návratová struktura *out* (převzato z [23])

Implementace algoritmu z BP[23] dělá přesně to co potřebuji. Hlavním úkolem je tedy co nejpřirozeněji integrovat relevantní kód – části spojené s demonstrační aplikací lze vynechat – do projektu. Pokud bude potřeba bylo by dobré navrhnout abstrakční vrstvu, která by odstínila Alenku od spike-detektoru.

Název	Popis
MV	Matice, obsahující hodnoty, které signalizují: 1 ... zřejmý výboj, $\frac{1}{2}$ ... nejednoznačný výboj.
MA	Matice obsahující max. amplitudu obálky nad pozadím.
MP	Počáteční pozice multikanálové události.
MD	Doba trvání události.
MW	Hodnota kumulativní distribuční funkce výboje z lognormálního modelu.
MPDF	Hodnota hustoty pravděpodobnosti.

Tabulka 2.2: Návratová struktura *discharges* (převzato z [23])

2.2.2 Návrh

2.2.2.1 Návrh vizualizace výsledků

Jednotlivé detekce budu vizualizovat jako jednokanálové anotace. Pro jejich generování využiji hodnot ve struktuře *out* (viz tabulka 2.1) následujícím způsobem:

- *pos* pro určení začátku anotace. (Jako logičtější by se mohlo zdát anotaci „vycentrovat“ na pozici detekce. To by ale přineslo problémy při exportu výsledků z aplikace, kdy by se musely výsledky korigovat.)
- *chan* pro kanál anotace
- *con* pro typ anotace

Položka *dur* není vhodná pro určení délky anotace. Pro vysoké vzorkovací frekvence by anotace s touto délkou byly „neviditelné“. Proto jsem se rozhodl rozšířit nastavení spike-detektoru o nastavení délky anotací (parametr `--sed` viz příloha C).

Pro každý běh analýzy se vytvoří tři nové typy anotací. Každý typ sdružuje detekce ze stejné kategorie. Kategorie se dělí podle jistoty, která je vyjádřena hodnotou *con*. To pomůže rozeznat anotace představující detekce vytvořené při posledním spuštění analýzy od ostatních. Zdá se, že spike-detektor podporuje pouze dva typy, ale nabízí nastavení tří hladin tolerance. Pro jistotu ale ponechám tři typy, protože v budoucnu se toto může změnit.

2.2.2.2 Návrh uživatelského rozhraní

Rozhodl jsem se napodobit uživatelské rozhraní demonstračního programu spike-detektoru v těchto ohledech:

- Implementuji dialogové okno pro změnu nastavení. Výchozí hodnoty budou nastavitelné skrze přepínače a konfigurační soubory.
- Zachovám podporu sledování průběhu za použití ovládacího prvku *progress bar* a možnost analýzu předčasně ukončit.

Dále implementuji možnost spuštění analýzy z terminálu. Uživatel bude moci zadat cestu k souboru, jméno souboru s výsledkem (v `.mat` formátu) a změnit parametry. Detaily o tom jak se Alenka používá z příkazové řádky a výčet všech možností nastavení můžete najít v příloze C.

2.2.2.3 Objektový návrh

Na obrázku 2.5 můžeme vidět vztahy mezi třídami spojenými se spike-detektorem.

Navrhl jsem jednoduché rozhraní, které zapouzdřuje složitý kód detektoru. Toto rozhraní sestává ze dvou tříd:

- `Spikedet`
- `AbstractSpikedetLoader`

Třídu `Spikedet` lze parametrizovat pomocí struktury s nastavením. Poté analýzu spustíme funkcí `runAnalysis()`. Tato třída podporuje sledování procentuálního postupu výpočtu. Pokud toto chceme použít, musíme analýzu spustit asynchronně v novém vlákně. To znamená, že funkce `progressPercentage()` a `cancel()` by měli být *thread-safe*.

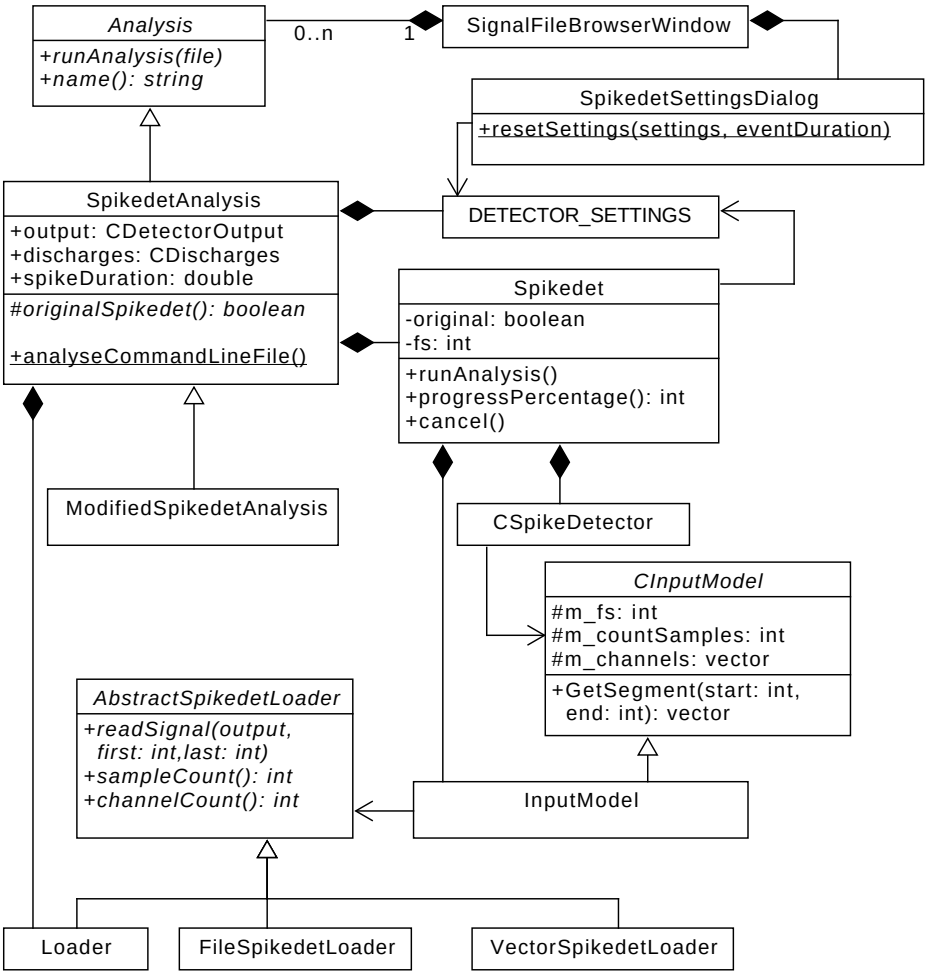
Abstraktní třída `AbstractSpikedetLoader` dovoluje definovat novou logiku načítání vstupních dat. Toto je rozšíření poněkud omezené třídy `CInputModel`, která počítá s načítáním dat ze souboru. Toto nové rozhraní je dostatečně obecná, aby dovolilo např. následující použití:

- načítání z nestandardního souborového typu (`FileSpikedetLoader`, který se používá při spuštění analýzy z terminálu)
- čtení z pole v paměti (`VectorSpikedetLoader` použitý při testování)
- online konstrukci vstupního signálu za použití mechanizmu montáží (`Loader`)

Třída `CSpikeDetector` je vstupním bodem do implementace spike-detektoru.

Obecná abstraktní třída `Analysis` dovoluje budoucí rozšíření menu *Tools* o další položky představující další analýzy. Třídy `SpikedetAnalysis` a `ModifiedSpikedetAnalysis` představují analýzy spike-detektoru; druhá z nich potom odpovídá upravenému detektoru s optimalizovanou decimací (viz sekce 2.2.3.3).

`SpikedetSettingsDialog` implementuje rozhraní pro změnu výchozích hodnot nastavení analýzy.



Obrázek 2.5: UML class diagram pro spike-detektor

2.2.3 Realizace

Spike-detektor jsem integroval podle návrhu.

Během této práce jsem narazil na několik nedostatků v převzaté implementaci. Ve zbytku této sekce popisuji, jak jsem tyto problémy řešil.

2.2.3.1 Odstranění závislosti na wxWidgets

Implementace spike-detektoru z BP má jednu nešťastnou vlastnost: závisí na knihovně wxWidgets[18], která se používá především pro multiplatformní programy s uživatelským rozhraním.

To platí i pro ten kód, který zajímá mě a který nemá s uživatelským

Výpis 2.1: Definice `wxVector` a `wxString`

```
template<class T>
using wxVector = std::vector<T>;

#define wxT(a_) wxString(a_)

class wxString : public std::string
{
public:
    wxString() : std::string() {}
    wxString(const char* str) : std::string(str) {}

    template<class... T>
    int Printf(const wxString& format, T... args)
    {
        int size1 = snprintf(nullptr, 0, format.c_str(), args...);
        std::unique_ptr<char[]> buffer(new char[size1 + 1]);

        int size2 = sprintf(buffer.get(), format.c_str(), args...);
        assert(size1 == size2); (void)size2;

        assign(buffer.get(), buffer.get() + size1);
        return size1;
    }

    template<class... T>
    static wxString Format(const wxString& format, T... args)
    {
        wxString str;
        str.Printf(format, args...);
        return str;
    }
};
```

rozhraním nic společného. Rozhodl jsem se proto implementovat minimální požadovanou sadu funkci a tříd z knihovny wxWidgets používanou v kódu spike-detektoru. Následují příklady implementace některých z mnou implementovaných rozhraní.

Spike-detektor využívá `wxVector` a `wxString` namísto standardních variant. Oba nabízí téměř stejnou funkcionalitu a téměř totožné rozhraní jako jejich alternativy ze standardní knihovny. Řešením je tedy využití standardních implementací s drobným rozšířením (viz výpis 2.1).

`wxVector` je alias šablonové třídy. `wxString` dědí z `std::string`, který navíc rozšiřuje o dvě formátovací funkce. Všimněte si využití variadických šablon pro předání proměněného počtu parametrů.

Výpis 2.2: Definice `wxThread`

```
class wxThread
{
public:
    typedef void* ExitCode;

    wxThread(int kind = wxTHREAD_DETACHED) : kind(kind) {}
    virtual ~wxThread() {}

    int Run()
    {
        thread = std::thread([this] {
            entryReturn = Entry();
        });

        if (kind == wxTHREAD_DETACHED)
            Wait();

        return 0;
    }

    ExitCode Wait()
    {
        thread.join();
        return entryReturn;
    }

    virtual bool TestDestroy() { return stop; }
    void Stop() { stop = true; }

protected:
    virtual ExitCode Entry() = 0;

private:
    std::thread thread;
    int kind;
    ExitCode entryReturn;
    std::atomic<bool> stop{false};
};
```

Dále jsem implementoval `wxThread` (viz výpis 2.2). Použil jsem znovu standardní implementaci. Dále jsem implementoval několik pomocných struktur, které zde ale nebudu uvádět.

Tento kód simuluje přesně to co by dělala knihovna wxWidgets a nahrazuje dva hlavičkové soubory `wx/wx.h` a `wx/thread.h`. Takto jsem schopný bez většího zásahu do původního kódu nahradit potřebnou funkcionalitu svojí vlastní

Výpis 2.3: Řešení úniku paměti (v souboru `CSpikeDetector.h`)

```
~oneChannelDetectRet()
{
    // These thwo vectors are allocated in localMaximaDetection().
    delete m_markersHigh;

    if (m_markersHigh != m_markersLow) // Sometimes they point to
        the same vector.
        delete m_markersLow;
}
```

implementací a zbavit se tím závislosti na obrovské knihovně třetí strany.

2.2.3.2 Opravené chyby

Během práce se spike-detektorem jsem narazil na tři chybu v kódu. V této sekci popíšu, jak jsem je vyřešil.

Uvádím poněkud velmi detailní popis a konkrétní řešení problémů v kódu, aby čtenář mohl použít tento text k opravě chyb v původní implementaci. Alternativně se dá použít implementace, která je součástí Alenky a která všechny tyto chyby řeší.

Ve funkci `COneChannelDetect::localMaximaDetection()` se na druhém řádku vytváří vektor na haldě, který se posléze vrací ven z funkce. Tento vektor se nikdy nesmaže, a tudíž dochází k úniku paměti. Únik reportují nástroje pro detekci úniků paměti Valgrind[16] i Address Sanitizer[1].

Řešením je vektory vytvořené v této funkci uvolnit v destruktoru struktury `oneChannelDetectRet`, když už nebudou k ničemu potřeba. Moje řešení lze vidět ve výpisu 2.3.

Další problém představuje chybný výpočet indexů, který pro některé počty kanálů způsobuje pád se *segmentatio fault*.

Kód z BP velice těsně sleduje svojí předlohu, kterou je implementace v Matlabu. V tomto případě se autor ale poněkud odchýlil. Ve výpisu 2.4 můžeme vidět původní kód v Matlabu[12]. Přiřazení operátoru hranatých závorek (`[]`) do elementu pole (nebo přesněji do jedno dimenzionální matice) vymaže prvek s daným indexem. Výsledné pole má délku o jedna menší.

Ve výpisu 2.5 je původní (chybná) implementace. Moje řešení je ve výpisu 2.6.

Třetí a poslední chybou je špatné zpracování výjimky.

Funkce `alglib::spline1dconvcubic()` může vrátit výjimku. Jedním z důvodům, proč se tak může stát je, že vstupní pole hodnot obsahuje samé

Výpis 2.4: Původní Matlab kód podmínky

```
if index_stop(end)-index_start(end)<T_seg*fs
    index_start(end)=[];
    index_stop(end-1)=[];
end
```

Výpis 2.5: Implementace podmínky z BP (`CSpikeDetector.cpp`)

```
if (indexStop.back() - indexStart.back() < T_seg * fs)
{
    indexStart.pop_back();
    indexStop.back() = cntElemInCh;
}
```

Výpis 2.6: Moje opravená verze kódu podmínky (`CSpikeDetector.cpp`)

```
if (indexStop.back() - indexStart.back() < T_seg * fs)
{
    indexStart.pop_back();
    auto tmp = indexStop.back();
    indexStop.pop_back();
    indexStop.back() = tmp;
}
```

nuly. To způsobuje, že se v mezi výpočtu objevují nekonečné hodnoty. To tato funkce není schopná zvládnout a vrací výjimku.

V *catch* klauzuli se vrací nulový ukazatel, což způsobuje problémy dále v kódu, kde se očekává, že tato operace proběhla v pořádku.

Moje řešení (výpis 2.7) je vyplnit výsledek nulovými hodnotami. Toto se mi zdá jako dobrý kompromis pro výsledek operace, která v tomto případě (s konstantním signálem na vstupu) zřejmě není správně definovaná.

2.2.3.3 Optimalizace

Také jsem identifikoval a vyřešil dva výkonnostní problémy. V této sekci prezentuji jejich řešení a výsledky měření demonstrující přínos těchto změn.

V souboru `CSpikeDetector.cpp`, ve funkci `COneChannelDetect::Entry()` můžeme najít kód z výpisu 2.8. Konstantní hodnoty se vkládají jedna po druhé ve smyčce na začátek vektoru. Tato operace má sama o sobě lineární složitost, protože před přiřazením prvku se musí nejprve překopírovat všechny elementy o jeden index dozadu. Navíc vektor může růst a to znamená velice

Výpis 2.7: Špatně vyřešená výjimka pro nulový kanál (`CSpikeDetector.cpp`)

```
try
{
    // interpolation Median and Std
    wxVector<double> phatMedVecDoub(phatMedian.begin(),
        phatMedian.end());
    wxVector<double> phatStdVecDoub(phatStd.begin(), phatStd.end());

    xreal.setcontent(x.size(), &x[0]);
    yrealMedian.setcontent(phatMedVecDoub.size(), &phatMedVecDoub[0]);
    yrealStd.setcontent(phatStdVecDoub.size(), &phatStdVecDoub[0]);
    x2real.setcontent(y.size(), &y[0]);

    alglib::spline1dconvcubic(xreal, yrealMedian, x2real, retMedian);
    alglib::spline1dconvcubic(xreal, yrealStd, x2real, retStd);
}
catch(alglib::ap_error e)
{
    //return NULL;
    std::vector<double> allZeroes(yrealMedian.length());
    retMedian.setcontent(allZeroes.size(), allZeroes.data());
    retStd.setcontent(allZeroes.size(), allZeroes.data());
}
```

Výpis 2.8: Původní verze vkládání elementů do vektoru

```
for (i = 0; i < floor(m_settings->m_winsize * m_fs / 2); i++)
{
    phat_int[0].insert(phat_int[0].begin(), temp_elem0);
    phat_int[1].insert(phat_int[1].begin(), temp_elem1);
}
```

drahou alokaci nové paměti. Celá tato smyčka má složitost řádově $O(n^2)$, tedy kvadratickou.

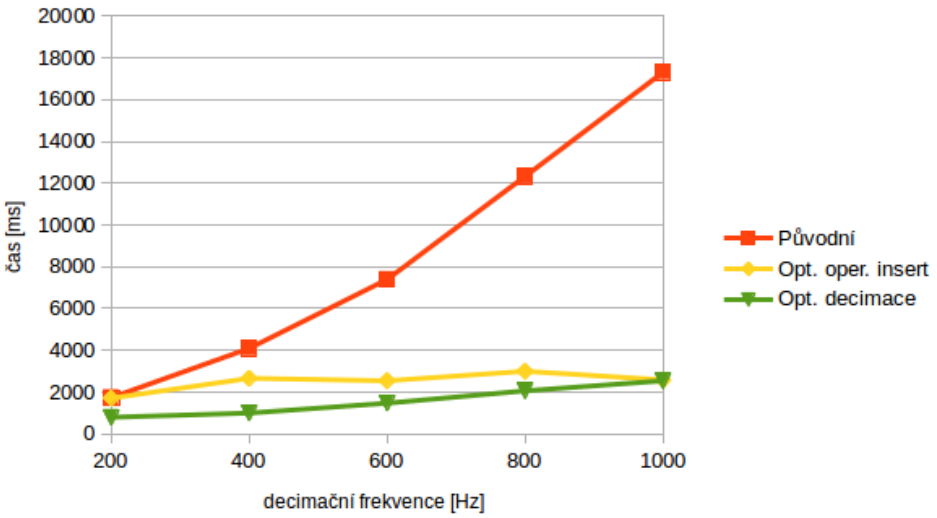
Pokud vkládáme konstantní prvek a jsme schopni vypočítat kolik prvků potřebujeme vložit, je mnohem efektivnější vložit všechny prvky najednou. Tím zaručíme, že budeme alokovat a kopírovat pouze jednou. Řešení můžeme vidět ve výpisu 2.9.

Jenom bych ještě chtěl podotknout, že `wxVector` použité přetížení funkce `insert()` nepodporuje. Takže je zapotřebí použít standardní vektor.

Na obrázku 2.6 můžeme vidět jaký dopad má tato optimalizace. Graf ukazuje čas běhu analýzy v závislosti na decimační frekvenci (parametr `--dec` viz příloha C). Pro původní řešení je zřejmá lineární závislost na frekvenci. Pro optimalizovanou verzi je vliv zvyšující se frekvence pouze nepatrný.

Výpis 2.9: Ekvivalentní, optimalizovaný kód vkládání elementů do vektoru

```
const int n = floor(m_settings->m_winsize * m_fs / 2);
phat_int[0].insert(phat_int[0].begin(), n, temp_elem0);
phat_int[1].insert(phat_int[1].begin(), n, temp_elem1);
```



Obrázek 2.6: Výsledek vektorové optimalizace spike-detektoru

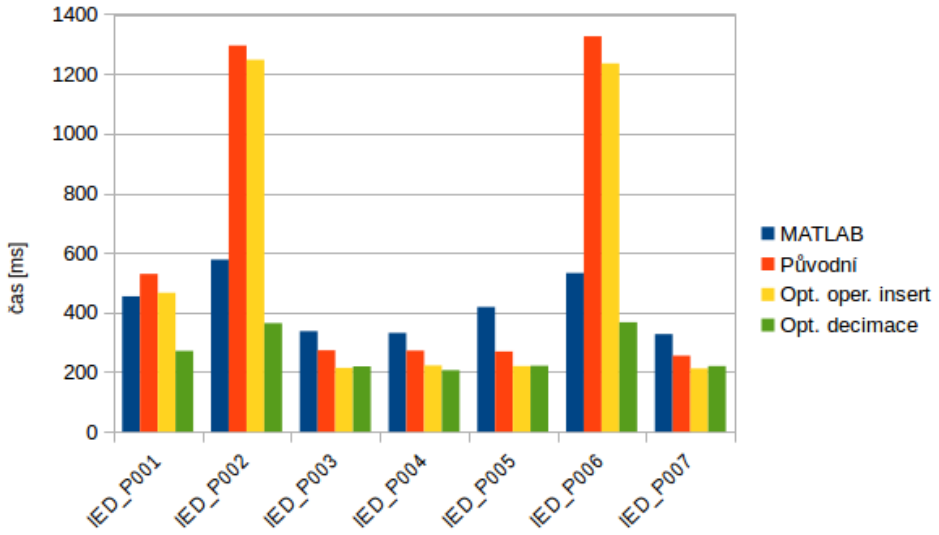
Pro nižší decimační frekvence dominuje příprava signálu – filtrace a decimace – nad samotným algoritmem provádějícím analýzu. Jak se frekvence zvyšuje, zvyšuje se i rozsah dat, který musí algoritmus zpracovat. Tento kód se nachází v nejkritičtější cestě výpočtu a má, jak je z obrázku vidět, velice významný vliv na výkonnost algoritmu.

Pokusil jsem se také vyřešit problém s výkonem decimace, o kterém se autor zmiňuje v závěru BP.

U testovacích souborů s vyšší vzorkovací frekvencí je zřejmý výrazný pokles výkonu oproti referenční implementaci v Matlabu. Jak můžeme vidět z obrázku 2.7, rozdíl je více jak dvojnásobný. Tento problém se jenom stupňuje jak roste frekvence. Pro 8 kHz testovací soubor je rozdíl už šestinásobný.

Problém je způsoben metodou zvolenou pro změnu vzorkovací frekvence signálu při přípravě pro analýzu. V původní verzi byla použita knihovna *libsamplerate*[29], která je určena pro převzorkování zvuku. Jako taková klade důraz na vysokou přesnost výsledku, ale bohužel není vhodná pro toto použití.

Já jsem zvolil knihovnu *SignalResampler*[19], což je adaptace funkce



Obrázek 2.7: Výsledky optimalizace spike-detektoru

`resample()` z Matlabu. Tuto funkci používá také referenční Matlab skript.

Graf na obrázku 2.7 ukazuje výsledky testu na počítači popsaném v tabulce 3.2. Je zřejmé, že optimalizovaná metoda decimace řeší problém u dvou souborů s vyšší vzorkovací frekvencí.

Optimalizovanou verzi decimace jsem otestoval na původní sadě souborů a přesnost analýzy není dramaticky ovlivněna. Výsledek se liší pouze pro jeden soubor `IED_P006`, kde dostáváme o 2 z 648 detekcí méně. To je stále pohodlně v toleranci 1% vyžadované zadáním BP.

Metodu decimace lze zvolit z příkazového řádku (parametr `--osd` viz příloha C) a v hlavním menu aplikace.

2.3 Zobecnění filtrace signálu

Tato sekce popisuje naplnění následujícího požadavku: *Zobecnění filtrace signálu: uživatel má možnost definovat změnu amplitudy v určitých frekvenčních pásmech.*

Motivací pro tuto novou funkčnost je to, že ISARG[8] se v poslední době snaží zkoumat jistý fenomén: teorie říká, že typickému spiku předchází vysokofrekvenční aktivita. Tato vlastnost se dá použít k přesnější klasifikaci nálezů v záznamu a k vyfiltrování falešných detekcí.

Problém ale nastává v okamžiku, kdy se tento fenomén snažíme vizualizovat pro kontrolu člověkem. Tato aktivita je v normálním EEG zobrazení zastíněna vlnami s nižší frekvencí, které mají z pravidla mnohonásobně vyšší

amplitudu. Typické spektrum EEG vypadá jako nepřímá úměra, tedy přibližně jako graf funkce $f(x) = \frac{1}{x}$.

Řešením tohoto problému je zesílení amplitudy v určitém frekvenčním pásmu. To spolu s potlačením nejnižších frekvencí dovoluje zvýraznit zajímavou vysokofrekvenční aktivitu.

2.3.1 Analýza

Problém popsaný v úvodu této sekce lze vyřešit rozšířením podpory pro filtraci.

Je potřeba implementovat nové ovládací prvky pro specifikování parametrů filtrace. Dále je žádoucí dát uživateli zpětnou vazbu – tedy nějakým způsobem vizualizovat vlastnosti aplikovaného filtru.

Mnou zvolená metoda výpočtu filtrace je nezávislá na koeficientech FIR filtru, které se pro daný výpočet používají. To přesouvá řešení problému do domény designu samotného filtru. Veškerá infrastruktura zpracování signálu proto nebude vyžadovat žádné změny.

Metoda pro design FIR filtru *frequency-sampling*[30], kterou jsem zvolil v předchozí BP, podporuje v podstatě libovolnou frekvenční odezvu. Stačí jako vstup zadat požadované hodnoty pro patřičné frekvence. Tyto hodnoty nejsou omezené pouze na 0 a 1, tedy na vypnuto a zapnuto. Můžeme zadat např. 10, čímž dosáhneme desetinásobného zesílení frekvencí patřících do daného frekvenčního pásma.

Volba této metody se opět vyplácí. Předtím nám dovolila zredukovat počet filtračních průchodů pro různě typy filtrů (low-pass, high-pass atd.) na jeden. Teď můžeme provést i mnohem složitější filtraci a pořád nám na to stačí jeden filtr.

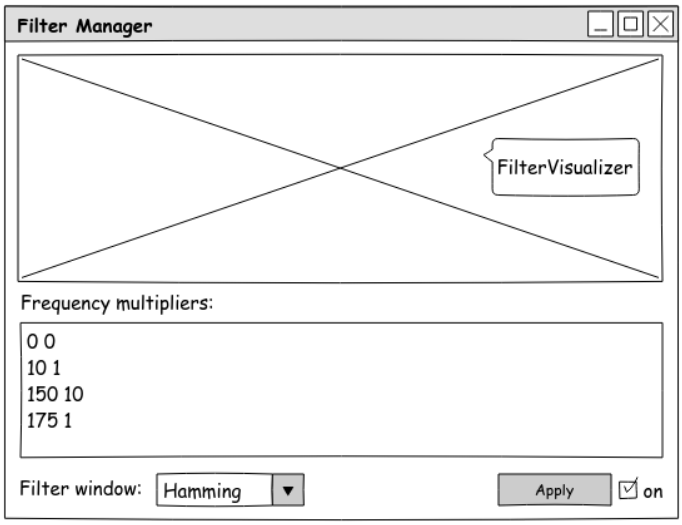
2.3.2 Návrh

Rozhodl jsem se, že nové ovládání filtrace bude pouze rozšiřovat stávající rozhraní. Tři základní typy filtrů jsou stále pro většinu situací dostačující a nemá cenu tuto funkčnost měnit, nebo odstraňovat. To znamená, že vizualizace by měla fungovat dobře se stávající implementací filtrace.

2.3.2.1 Návrh uživatelského rozhraní

Navrhl jsem nové pod-okno, které bude zodpovědné za veškerou novou funkcionalitu spojenou s filtrací. Návrh je ilustrován na obrázku 2.8.

Hodnoty pro definici obecné filtrace jsem nazval „frekvenční multiplikátory“. Zadávají se do textového pole jako textový řetězec. Toto je na první pohled celkem primitivní řešení, ale má také několik výhod oproti sofistikovanému/interaktivnímu rozhraní:



Obrázek 2.8: Návrh uživatelského rozhraní manažeru filtrace

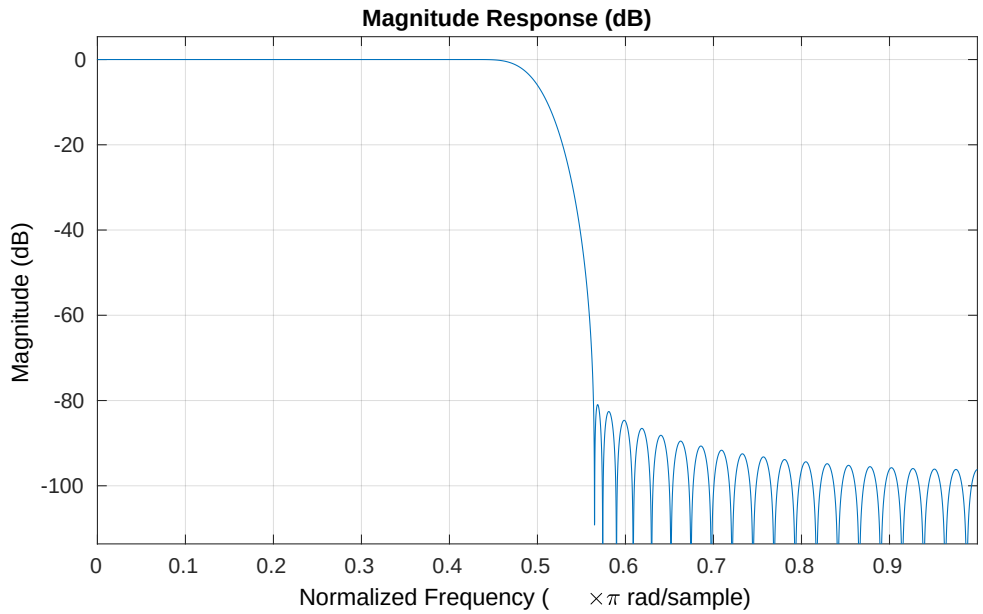
- multiplikátory si uživatel může vygenerovat v nějakém externím programu (jednoduchým příkazem v Matlabu, Pythonu, nebo dokonce i v tabulkovém procesoru)
- ukládání stavu multiplikátorů je triviální – stačí uložit jen jeden řetězec

Syntaxe pro definici multiplikátorů se řídí následujícími pravidly:

1. řetězec se čte po řádcích
2. každý řádek může obsahovat jedno až dvě desetinná čísla (f, m) oddělená bílými znaky
3. f určuje frekvenci
4. m je volitelné (výchozí hodnota je 1) a udává hodnotu amplitudy pro frekvence $\geq f$
5. řádky nevyhovující této definici se ignorují

To tedy znamená, že záznamy musíme zadávat ve vzestupném pořadí podle frekvence. Řetězec z příkladu na obrázku 2.8 tedy znamená: odfiltruj 0 až 10 Hz, zvyš desetkrát amplitudu pro 150 až 175 Hz a zbytek frekvencí bude mít amplitudu 1.

Okno dále nabízí vizualizaci frekvenční odezvy aktivního filtru: `FilterVisualizer`. Toto zobrazení bude využívat stejný formát jako de facto standardní nástroj `freqz()` z prostředí Matlab.



Obrázek 2.9: Ukázka výstupu funkce `freqz()` v Matlabu

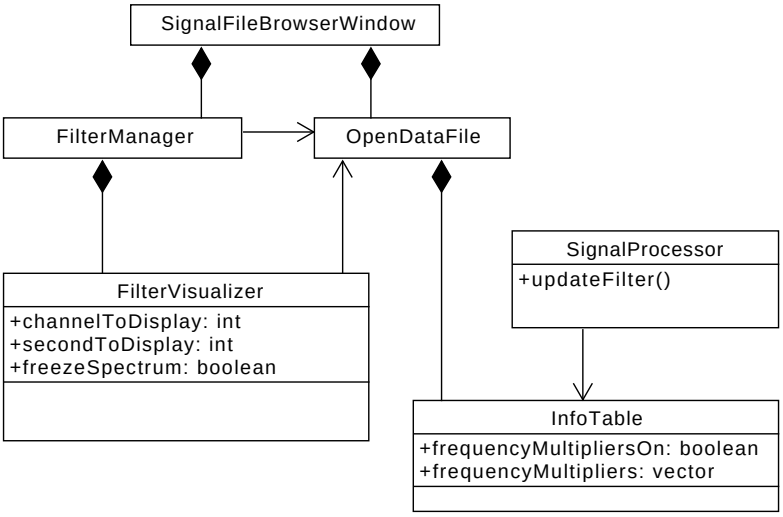
Příklad výstupu této funkce můžeme vidět na obrázku 2.9. Low-pass filtr v ukázce má přechod na $\frac{1}{4}$ vzorkovací frekvence. Na svislé ose je použita logaritmická škála. To je proto, aby se daly poznat malé změny kolem jedničky na hraně přechodu. U utlumení nás přesná hodnota nezajímá, stačí pouze řádový odhad.

Nakonec okno obsahuje ovládací prvek pro zvolení oknovací funkce. Možnosti budou zahrnovat několik populárních funkcí včetně hranatého okna (tedy žádné/vypnuto). Toto může uživatel použít pro zvýšení útlumu filtru za cenu delšího přechodového pásma.

2.3.2.2 Objektový návrh

Na obrázku 2.10 můžeme vidět UML diagram pro třídy zodpovědné za implementaci nové funkčnosti.

`FilterManager` představuje nové pod-okno, jehož součástí je `FilterVisualizer`, prvek pro vizualizaci filtru. Obě třídy přistupují ke globálnímu stavu aplikace, který je přístupný skrze objekt `OpenDataFile`. Samotné aplikování filtru je provedeno funkcí `updateFilter()` třídy `SignalProcessor`, která vygeneruje nový filtr při každé změně parametrů filtrace. K tomu použije multiplikátory, které byly předzpracovány v manažeru.



Obrázek 2.10: UML class diagram pro filtraci

2.3.3 Realizace

Na obrázku 2.11 můžeme vidět konečnou podobu pod-okna z návrhu uživatelského rozhraní (obrázek 2.8).

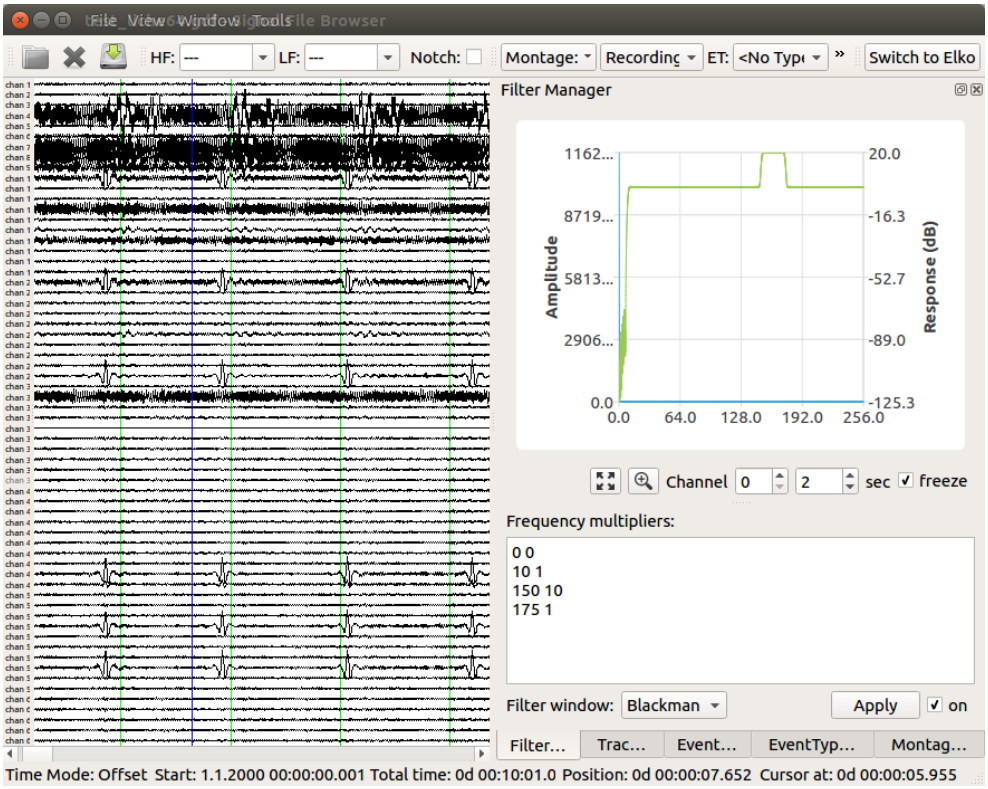
K implementaci vizualizace jsem využil ovládací prvek `QChartView`[34], který je více než dostačující pro vykreslení jednoduchého grafu. Navíc jako bonus dovoluje interaktivní přibližování.

Došlo k několika změnám oproti návrhu:

- rozšířil jsem vizualizaci o spektrum čistého signálu (signálu načteném přímo ze souboru)
- přidal jsem ovládací prvky pro přizpůsobení vizualizace (výběr kanálu pro spektrum, interval pro spektrum atd.)

Online vykreslování spektra může být na některých platformách značně drahé (např. na virtuálním počítači). Proto jsem přidal zaškrtávací pole pro „zmrazení“ zobrazení, což deaktivuje obnovu spektra při změně pozice v záznamu.

Pro výpočet spektra jsem použil knihovnu Eigen[4] (původně použité ve spike-detektoru), která nabízí extrémně jednoduché rozhraní, ale také extrémně slabý výkon. Jelikož vizualizace spektra je velice okrajová funkčnost (která je navíc ve výchozím stavu vypnutá), rozhodl jsem se zůstat u této varianty. Alternativou by bylo využít výpočtu FFT na GPU stejně jako při filtraci. Toto by znamenalo zvýšenou složitost kódu, ale předešlo by se paradoxní situaci, kdy zobrazení spektra pro několik málo sekund z jednoho kanálu



Obrázek 2.11: Ukázka okna Filter Manager

zpomaluje hlavní zobrazení, které zpracovává nerovnatelně (řádově stokrát i tisíckrát) větší objem dat.

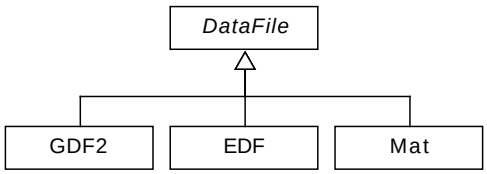
2.4 Rozšíření podpory datových formátů

Původní verze Alenky podporovala pouze jeden datový formát: GDF[31]. Pro tento formát však existuje spousta alternativ. Po dohodě se zákazníkem jsem rozšířil podporu o dva nové formáty:

- EDF[5] a
- MAT[10]

Pro výběr každého z těchto dvou formátů byly různé důvody. EDF je velice podobný GDF, ale má širší využití a aplikační podporu. MAT je formát do, kterého lze exportovat přímo z Matlabu[12]. To – jak uvidíme dále – zredukuje počet kroků nutných na zobrazení dat, která vzniknou v Matlabu, na minimum.

Všechny tři formáty sdílejí jednotné rozhraní: abstraktní třídu `DataFile` (viz obrázek 2.12). Ta byla navržena v BP přesně za účelem jednoduchého



Obrázek 2.12: UML class diagram pro `DataFile`

budoucího rozšíření o nové formáty. Toto nám mimo jiné umožní využít jednotný mechanizmus sekundárních souborů pro rozšíření ukládání metadat, která nelze přímo uložit v některých ze tří standardních formátech.

2.4.1 EDF

EDF[5] neboli *European Data Format* je jednoduchý a flexibilní formát pro ukládání biologických signálů. Používá se v medicíně i ve výzkumu a je nejběžnějším formátem pro nahrávání EEG záznamů.

K implementaci jsem použil knihovnu EDFlib[22], která umožňuje čtení i zápis EDF a EDF+ (novější verze EDF) souborů.

Tento formát je natolik užitečný, že jsem implementoval speciální funkci, která exportuje aktuálně otevřený soubor do EDF+. Pro tuto funkci jsem využil toho, že EDFlib podporuje zápis a že `DataFile` umožňuje čtení přes jednotné rozhraní. Na první pohled nenápadná funkce export dále rozšiřuje použitelnost aplikace: Alenka se stává převodníkem formátů, případně generátorem souborů z výstupu z Matlabu.

2.4.2 MAT

MAT[10] je souborový formát používaný výpočetním prostředím Matlab[12] pro export proměnných. Exportovat lze skalární hodnoty, matice i datové struktury.

Existuje několik verzí formátu MAT. Je velmi důležité se ujistit, že používáte právě poslední verzi 7.3 z roku 2006. Tato verze nabízí následující vlastnosti:

- podpora komprese
- maximální velikost jedné proměnné $\geq$ 2 GB (nově ve verzi 7.3)
- načítání pouze části dat proměnné (nově ve verzi 7.3)

První dvě vlastnosti jsou nutností pro ukládání dlouhých záznamů EEG s více kanály a vyšší vzorkovací frekvencí. Komprese může ušetřit až 50% prostoru na disku.

Jednou z velkých výhod Alenky je to, že dokáže načítat datové soubory po částech/blocích. Tím odpadá zdlouhavé předzpracování signálu pro zobrazení a snižuje se paměťová náročnost. Toto ale vyžaduje podporu ze strany datových formátů a případně také od knihoven, které Alenka používá pro jejich čtení. To je hlavní důvod, proč je nutné používat MAT ve verzi 7.3. Starší verze totiž čtení po částech nepodporují – především potom verze 7.0, která navíc zavádí kompresi, což ještě více zpomaluje čtení. Tento problém se projevuje tím, že aplikace je zpomalena a „zamrzává“, jak je hlavní vlákno blokováno při čtení ze souboru.

Řešení tohoto problému by vyžadovalo zásah do chování aplikace při otevírání souboru. Mohli bychom např. využít vyrovnávací *cache* paměti, kterou lze nyní zapnout parametrem `--fileCacheSize` (viz příloha C). Do této paměti by bylo možné při startu načíst celý soubor, pokud bude dostatečně malý, a předejít dynamickému načítání během pohybu v záznamu. Tento problém však není prioritou a dá se mu předejít přechodem na novější verzi formátu, což by neměl být problém – dnes už je tomu více jak deset let co je verze 7.3 dostupná – a proto jsem tento problém ponechal nevyřešený.

Moje implementace čtení tohoto formátu využívá knihovnu MATIO[11]. Ta podporuje všechny verze MAT formátu. Pro verzi 7.3 MATIO vyžaduje knihovnu HDF5[7], která implementuje čtení ze stejnojmenného formátu, na kterém je formát MAT 7.3 založen.

2.4.2.1 Propojení s Matlabem

Motivací pro rozšíření podpory o formát MAT bylo hlavně to, že jsem chtěl co nejvíce snížit režii spojenou s vizualizací dat vygenerovaných přímo v Matlabu.

Typické využití vizualizačního softwaru jako je Alenka může probíhat např. takto:

1. zpracování signálu, nějakou nestandardní metodou
2. uložení dat do externího formátu, který podporuje požadovaná aplikace
3. spuštění aplikace
4. vyhledání souboru na disku a otevření v dané externí aplikaci

Tento postup se může opakovat mnohokrát za sebou v iteracích. Jak si může čtenář lehce představit, tento postup vyžaduje poměrně hodně manuálních operacích, které uživatel musí otrocky opakovat po každé znovu.

Jednou z variant jak se této rutinní práci vyhnout je použít některý z vizualizačních nástrojů přímo v Matlabu. Můžeme mít ale důvod, proč preferovat některou z externích aplikací. V případě Alenky by to byl např. výkon vykreslování, nebo speciální funkce jako filtrace a montáže.

Rozšířil jsem rozhraní Alenky o možnost použití parametrů a přepínačů z příkazové řádky. Kompletní seznam můžete najít v příloze C. Toto ve spojení

Výpis 2.10: Příklad použití Alenky pro vizualizaci dat přímo z Matlabu

```
save('export.mat', 'data', 'fs'); !./Alenka-Linux/Alenka --matData
    data export.mat
```

s podporou formátu MAT znamená, že výše popsané kroky 2 až 4 můžeme zvládnout jednořádkovým příkazem přímo z Matlabu.

Příklad takového příkazu můžete vidět na výpisu 2.10. Chceme vizualizovat signál uložený ve sloupcové matici – každý sloupec je jeden kanál – která je uložená v proměnné s názvem `data`. Vzorkovací frekvence signálu je uložena v proměnné `fs`. Jedním příkazem `save()` exportujeme data do souboru. Dále použijeme operátor vykřičník pro spuštění příkazu operačního systému (alternativně můžeme použít funkci `system()`) pro spuštění Alenky. Předáme parametr se jménem souboru a pomocí přepínače zvolíme proměnnou, která obsahuje signál (alternativně můžeme použít výchozí hodnotu d). To je vše co je potřeba pro přenos dat z pracovního prostoru Matlabu do Alenky.

Pro spuštění musíme použít skript, který je součástí distribučního balíčku, nikoliv přímo spustitelný/binární soubor. Toto je způsobeno tím, že Matlab přesměrovává na systémové sdílené knihovny, které jsou součástí jeho distribuce. Ty jsou nekompatibilní a způsobují pád Alenky.

Specifikování jmen proměnných podporuje jednu úroveň zanoření do struktury. Můžeme tedy např. uložit vzorkovací frekvenci jako součást struktury s názvem `header`. Potom bychom museli použít přepínač `--matFs header.fs` při spuštění Alenky.

Alenka také podporuje více vstupních MAT souborů a lze specifikovat více jmen proměnných pro signál. Toto nám umožňuje zobrazení jednoho signálu, který je spojením několika signálů dohromady.

Zákazník používá komplexní skripty pro analyzování záznamů EEG, které občas přerušují svou práce a vyžadují verifikaci „pohledem“ od technika. Toto rozšíření aplikace dovoluje přirozené použití z takovýchto skriptů, což dále rozšiřuje použitelnost programu.

2.5 Změny zaměřené na údržbu a kvalitu

U *open-source* projektů, které spoléhají převážně na přispění dobrovolníků, je velice důležité aby byly snadno přístupné. Aspirující dobrovolník vývojář by měl být schopný jednoduše „rozchodit“ projekt na svém lokálním počítači. Toto je zásadní požadavek úspěšnosti projektu. Další důležité aspekty mohou být čitelnost a organizace kódu, dostupnost testů atd.

V této sekci popíšu, jaké změny jsem učinil abych můj projekt lépe zpřístupnil potenciálním budoucím vývojářům.

2.5.1 Build

Snažil jsem se maximálně zjednodušit a zautomatizovat proces překladu programu. V příloze D jsou aktuální instrukce.

Přešel jsem z pro Qt specifický systém *qmake* na mnohem typičtější a dnes de facto standardní *cmake*[3]. Tento nástroj se používá ke generování projektů v různých formátech a na různých platformách. Pomocí konfiguračního souboru lze celkem jednoduše specifikovat kroky pro přeložení a sestavení programu. Zároveň lze ale také konfigurační soubor parametrizovat a přizpůsobit pro specifické požadavky některých platforem.

Přechodem na *cmake* jsem se byl schopný zbavit komplikovaných konfiguračních skriptů z předchozí verze. Navíc už není potřeba používat nestandardní nástroje, nebo dokonce IDE dodávané s Qt pro provedení samotného překladu. Stále lze ale *Qt Creator*[14] (výše zmíněné IDE) použít, protože *Qt Creator* *cmake* podporuje. *Cmake* generuje plně funkční projekt pro Visual Studio, takže toto je další alternativa pro vývojáře.

Všechny závislé knihovny třetí strany kromě Qt lze automaticky stáhnout pomocí přiloženého skriptu `download-libraries.sh`. Podobný skript je k dispozici pro stažení testovacích dat.

2.5.2 Release

Pro usnadnění distribuce jsou k dispozici dva skripty pro vytvoření samostatných balíčků. Tyto skripty lze najít v podadresáři `misc/deploy/`. Tyto balíčky nevyžadují instalaci žádných dodatečných knihoven a softwaru. Jedinou výjimkou jsou ovladače GPU a OpenCL *runtime*.

První skript je určen pro systém Windows. Vytvoří se dva balíčky pro 32-bitové a 64-bitové OS. Verze pro Windows vyžaduje navíc instalaci Visual C++ *runtime*.

Druhý skript je univerzální pro Ubuntu 14 a 16 (poslední dvě dlouhodobě podporované verze). Proto aby Alenka fungovala na obou verzích, staticky linkuji některé z vyžadovaných systémových knihoven. Toto způsobuje, že balíčky pro Linux jsou podstatně větší (cca 100 MB) než na Windows (cca 50 MB). Z důvodů výhodnosti jedné univerzální verze, jsem se rozhodl tento kompromis akceptovat.

Nakonec bych se ještě zmínil, že jsem se pokusil také o rozšíření podpory pro systém OS X[38]. Podařilo se mi úspěšně program přeložit a spustit, ale neměl jsem dostatek času a hardwaru pro testování, abych mohl podporu dotáhnout na stejnou úroveň jakou mají ostatní dvě platformy.

2.5.3 Přepoužitelnost kódu

Provedl jsem také změnu ve struktuře kódu projektu. Vyčlenil jsem dva moduly, které lze nyní použít nezávisle na hlavní aplikaci:

- Alenka-File: modul pro načítání/ukládání datových souborů
- Alenka-Signal: modul pro zpracování signálu

Pokud by někdo v budoucnu chtěl navázat na mojí práci a chtěl zpracovávat výstupní soubory rozšiřující primární datové soubory, může využít souborového modulu a ušetřit si práci. Další potenciální využití tento modul může mít pro své simplistické rozhraní pro dynamické načítání dat ze tří různých datových formátů (viz sekce 2.4).

Druhý modul implementuje verzatilní metodu filtrace, kombinování kanálů přes koncept montáží a nově také automatický detekční algoritmus spike-detektor (viz sekce 2.2).

Oba tyto moduly jsou plně nezávislé na knihovně Qt. K tomu abych tohoto dosáhl jsem musel změnit implementaci čtení XML formátu. Nyní používám knihovnu pugixml[26] na místo Qt.

2.5.4 Code style

Pro zlepšení čitelnosti kódu jsem se rozhodl zvolit jednotný styl a formát pro celý projekt.

Začal jsem používat automatický nástroj pro formátování kódu *clang-format*[2]. *Clang-format* velice efektivně formátuje i velmi složitý kód a je prioritně navržen pro C a C++.

Používám výchozí styl podporovaný tímto nástrojem s názvem „LLVM“.

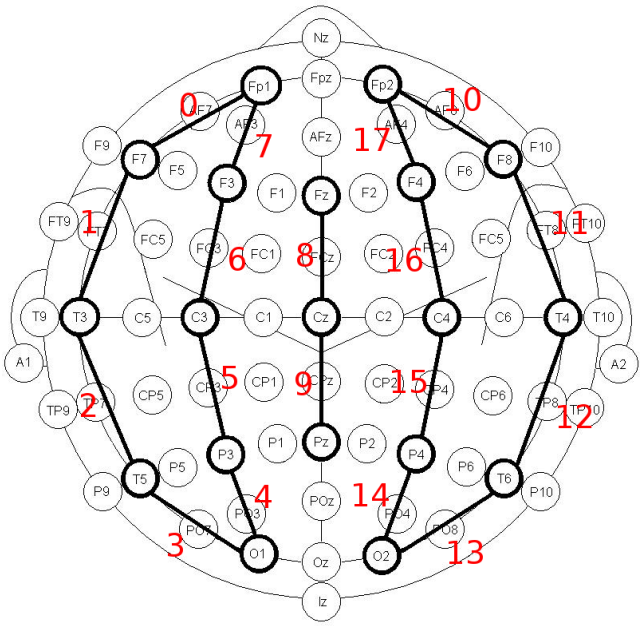
2.6 Změny v montážích

V této sekci popíšu, jak jsem rozšířil koncept montáží. Montáž je sada vzorců, které definují kanály nového signálu za použití dat z originálního signálu uloženém v datovém souboru.

2.6.1 Analýza

Motivací pro práci popsané v této sekci byl následující příklad použití při analýze EEG záznamu. Na obrázku 2.13 můžete vidět příklad jedné z typických montáží. Pro záznam se používá čepice s elektrodami označenými stejně jako je vidět na schématu na obrázku. Některé čepice mají pouze elektrody, které jsou zde zvýrazněné, jiné mají všechny. Klíčové je, že elektrody na určité pozici mají vždy stejné jméno.

Pro každou zapojenou elektrodu se vytvoří jeden kanál v záznamu. My ale nyní potřebujeme definovat nový signál s 18 kanály. Každý nový kanál představuje rozdíl napětí na elektrodách podle schématu. Uživatel nyní musí definovat 18 vzorců a zaručit, že správně namapuje jména ze schématu na indexy kanálů v záznamu. Jak jednou tuto montáž definuje, může ji přepoužít i pro jiné soubory se záznamem. Problém ale nastává, když se změní pořadí



Obrázek 2.13: Zapojení elektrod pro montáž *double-banana*

kanálů v záznamu, nebo když použijeme novou čepici s větší hustotou pokrytí elektrod.

Ve stávající formě jsou montáže dostatečně obecné, aby dovolili definovat takřka libovolnou kombinaci kanálů skrze kód v jazyce OpenCL C. Avšak práce s tímto rozhraním je krkolomná a náročná. Navíc kód montáží je specifický pro každý soubor. Cílem je tedy zjednodušení práce s montážemi. Toho docílím několika způsoby:

- rozšířením syntaxe o podporu symbolických jmen kanálů
- přidáním automatických programovatelných montáží
- implementací nových ovládacích prvků pro správu přepoužitelných šablon

2.6.2 Návrh

2.6.2.1 Rozšíření syntaxe

Nahrávací elektrody jsou často označovány standardními značkami. Tyto značky lze importovat spolu se signálem z datových souborů v podobě popisek (*labels*), nebo jmen kanálů. Toto jsou krátké řetězce znaků, které identifikují elektrody odpovídající daným kanálům. Pokud datový soubor tyto popisky neobsahuje, může je uživatel skrze rozhraní Alenky přidat. Tato informace se potom uchovává v sekundárním souboru.

Aby se docílilo přepoužití formulí montáží, je potřeba přerušit přímé spojení mezi kódem a indexem kanálu v souboru. Toho docílím rozšířením syntaxe formulí o možnost specifikovat vstupní kanál pomocí symbolického jména na místo indexu.

Jednou z možností, jak tohoto docílit, je předání řetězců se značkami kernelu pro výpočet montáže. Potom bych mohl přetížit funkci `in()`, aby akceptovala řetězce a pro danou značku přiřadila správný kanál. Tento výpočet by se musel provádět při každém spuštění kernelu a vracel by vždy stejný výsledek. Navíc vyhledávání ve velké tabulce symbolů by mohlo mít neblahý vliv na výkon – GPU jsou optimalizována pro numerické operace a složité větvení v kódu jim nesvědčí.

Druhým řešením je neměnit implementaci kernelu, ale transformovat formule montáží: nahradit symbolická jména ve vstupu uživatele patřičným indexem. Tento přístup netrpí výše zmíněnými potenciálními problémy s výkonem. Jediná nevýhoda je, že se nevyhneme kompilaci kernelu při změnách jmen kanálů. Toto je řešení, které jsem zvolil.

2.6.2.2 Programovatelné montáže

Další novou funkcí bude automatické generování montáží pomocí logiky definované v kódu. Generátor dostane seznam jmen kanálů a v závislosti na vybraném druhu automatické montáže vytvoří novou montáž se vzorci vytvořených podle určitého pravidla.

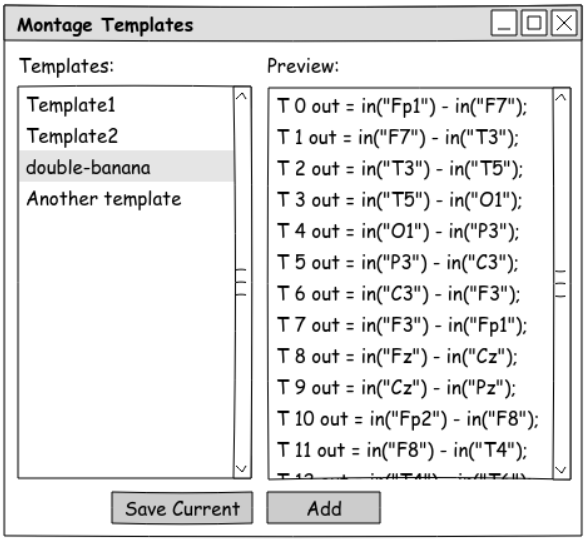
Příkladem takové montáže může být tzv. bipolární montáž. Ta obsahuje kanály, které jsou rozdíly sousedních kanálů z původního záznamu. Díky jmenné konvenci použité pro elektrody lze toto pravidlo implementovat poměrně jednoduše. Bipolární montáž bude součástí výchozí implementace. Je ale žádoucí, aby bylo snadné definovat nové druhy programovatelných montáží.

2.6.2.3 Návrh uživatelského rozhraní

Pro správu šablon montáží jsem navrhl jednoduché okno. Návrh můžete vidět na obrázku 2.14. Okno umožňuje

- exportovat aktuální montáž a
- přidat vybranou šablonu do aktuálně otevřeného souboru

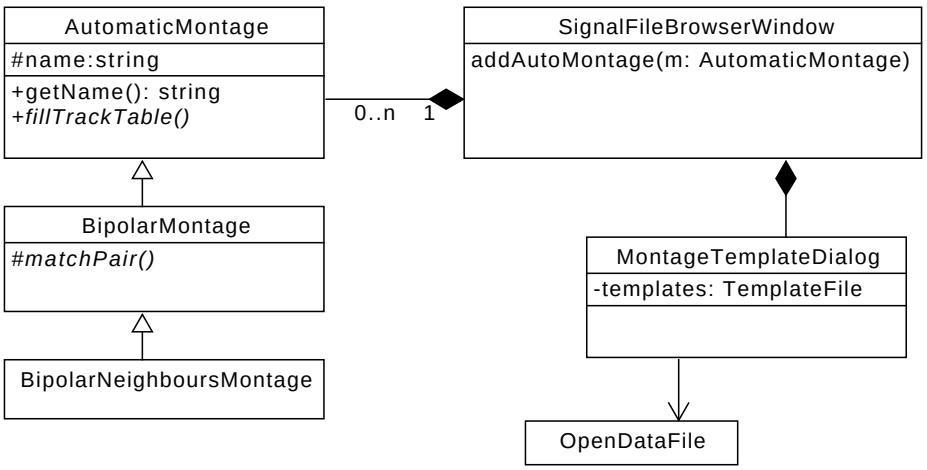
Seznam šablon v levém sloupci je vytvářen při otevření okna. Každý záznam v seznamu odpovídá jednomu souboru v adresáři `montageTemplates` v instalačním adresáři Alenky. Očekávaný formát šablon je ilustrován ve výpisu 2.11.



Obrázek 2.14: Návrh okna pro správu šablon montáží

Výpis 2.11: Příklad souboru pro uložení šablony montáže: *double-banana*

```
<?xml version="1.0"?>
<document>
    <row label="T 0">
        <code>out = in("Fp1") - in("F7");</code>
    </row>
    <row label="T 1">
        <code>out = in("F7") - in("T3");</code>
    </row>
    <row label="T 2">
        <code>out = in("T3") - in("T5");</code>
    </row>
    <row label="T 3">
        <code>out = in("T5") - in("O1");</code>
    </row>
    <row label="T 4">
        <code>out = in("O1") - in("P3");</code>
    </row>
    <!-- ... -->
    <row label="T 16">
        <code>out = in("C4") - in("F4");</code>
    </row>
    <row label="T 17">
        <code>out = in("F4") - in("Fp2");</code>
    </row>
</document>
```



Obrázek 2.15: UML class diagram pro novou funkcionalitu montáží

2.6.2.4 Objektový návrh

Obrázek 2.15 ukazuje návrh nových prvků potřebných pro implementaci výše navržené funkcionality. Změny týkající se syntaxe budou přidány do stávající implementace zpracování montáží.

`AutomaticMontage` je základní třídou pro specifikaci logiky generování programovatelných montáží. Sama tato třída generuje prázdnou montáž (s 0 kanály), což je triviální případ generované montáže. Zděděním této třídy a přidáním potomka do seznamu generátorů pomocí `addAutoMontage()` může vývojář rozšířit rozhraní Alenky o novou funkčnost.

`MontageTemplateDialog` implementuje okno pro správu šablon včetně exportu a importu souborů. Okno mění globální stav programu, proto musí mít odkaz na objekt, který patřičný stav reprezentuje (tedy `OpenDataFile`).

2.6.3 Realizace

V předešlé verzi se všechny kanály montáže počítaly v jednom výpočetním kernelu. V nové verzi má každý kanál vlastní kernel. Důvodem bylo, že jsem chtěl kód pro jednotlivé kanály izolovat, aby chyba kompilace pro jeden neznamenala selhání všech ostatních. Toto ale přineslo nečekané problémy s výkonem (více v sekci 2.6.3.2).

Aktuální verze kódu kernelu lze vidět ve výpisu 2.12. Kód nadále využívá stejného mechanizmu pro specifikaci vstupu a výstupu:

- funkce `in()` – jejíž parametry jsou zastíněny pomocí stejnojmenného makra – pro vstup a

Výpis 2.12: Kód kernelu pro výpočet montáže

```
__kernel void montage(__global float *_input_,
       __global float *_output_, int _inputRowLength_,
       int _inputRowOffset_, int IN_COUNT, int _outputRowLength_,
       int INDEX, int _outputCopyCount_, __global float *_xyz_) {
  float out = 0;

  {
    out = in(0) - in(10); // Example of a formula entered by the user
  }

  int outputIndex = _outputCopyCount_ *
                  (_outputRowLength_ * INDEX + get_global_id(0));
  for (int i = 0; i < _outputCopyCount_; ++i) {
    _output_[outputIndex + i] = out;
  }
}
```

- proměnná `out` pro výstup

Kernel dále podporuje zapsání více kopií stejné hodnoty do výstupu. Toto je nutné pro vykreslení anotací, protože potřebujeme dva body pro vytvoření polygonu z lomené čáry (viz BP[20]). Také jsou předávány dvě konstanty, které lze využít ve formulích:

- `IN_COUNT` počet kanálů v souboru
- `INDEX` index kanálu montáže

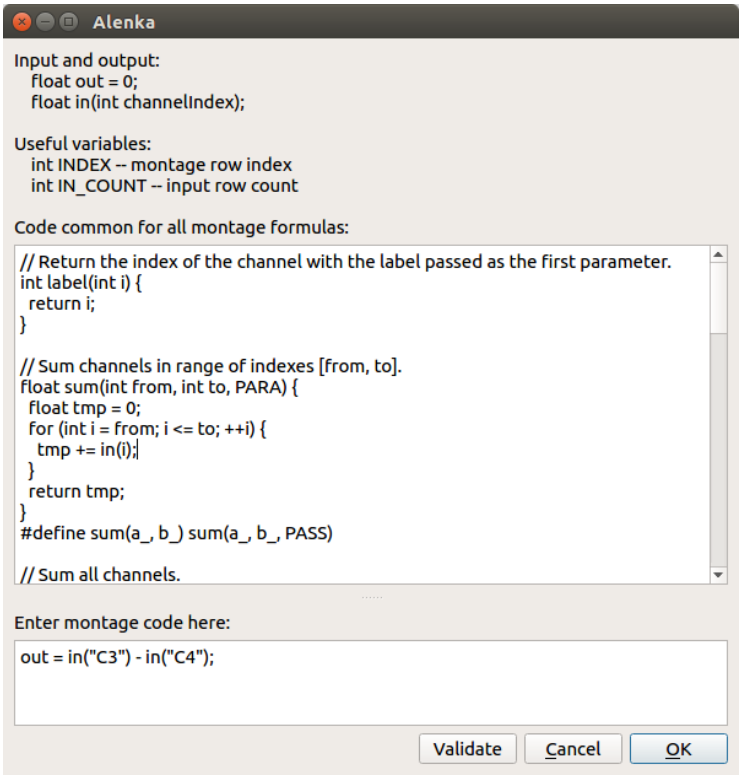
Každý kernel má výchozí hlavičku s definicemi (výpis 2.13). Hlavička byla rozšířena o funkce `x()`, `y()` a `z()`, které lze nyní použít pro získání souřadnic elektrod. Souřadnice lze zadávat v pod-okně *Track Manager* pro montáž s indexem 0. To znamená, že uživatel nyní může při výpočtech využít např. vzdálenosti mezi elektrodami.

Uživatel může také definovat vlastní funkce. Dříve se uživatelské definice načítaly přímo ze souboru při startu aplikace. Nyní lze za běhu tento kód editovat. To umožní rychlejší prototypování při vývoji nových funkcí. Tímto se také předejde problémům v případě, že hlavička obsahuje nekorektní kód. Dialog pro editaci kódu montáže, který je pro ilustraci na obrázku 2.16, nyní implementuje editaci a validaci kódu.

Ve výpisu 2.14 můžete vidět příklad definice vlastních funkcí a využití rozšíření o podporu souřadnic elektrod. Funkce `distAverage()` počítá vážený průměr z rozdílů elektrod v závislosti na vzdálenosti od centrální elektrody. K výpočtu vzdálenosti se používá pomocná funkce `dist()`, která počítá de

Výpis 2.13: Fixní hlavička kernelu pro výpočet montáže

```
#define PARA __global float *_input_, int _inputRowLength_, \
  int _inputRowOffset_, int IN_COUNT, __global float *_xyz_, int INDEX
#define PASS _input_, _inputRowLength_, _inputRowOffset_, \
  IN_COUNT, _xyz_, INDEX

float in(int i, PARA) {
  if (0 <= i && i < IN_COUNT)
    return _input_[_inputRowLength_ * i + _inputRowOffset_ +
        get_global_id(0)];
  else
    return 0;
}
#define in(a_) in(a_, PASS)

float x(int i, PARA) {
  if (0 <= i && i < IN_COUNT)
    return _xyz_[3 * i];
  else
    return 0;
}
#define x(a_) x(a_, PASS)

float y(int i, PARA) {
  if (0 <= i && i < IN_COUNT)
    return _xyz_[3 * i + 1];
  else
    return 0;
}
#define y(a_) y(a_, PASS)

float z(int i, PARA) {
  if (0 <= i && i < IN_COUNT)
    return _xyz_[3 * i + 2];
  else
    return 0;
}
#define z(a_) z(a_, PASS)

// Programmable header (montageHeader.cl) goes here
```



Obrázek 2.16: Okno pro zadávání kódu montáže

facto Pythagorovu větu. Použití funkce `distAverage()` ve formuli montáže bude potom vypadalo takto:

```
out = distAverage(INDEX, 1);
```

Tento vzorec – stejný pro všechny řádky montáže – nám dá průměr rozdílů pro všechny elektrody.

2.6.3.1 Implementace transformace formulí

Podpora symbolických jmen pro identifikaci vstupních kanálů byla implementována pomocí regulárních výrazů. Využil jsem implementaci ze standardní knihovny C++11.

Regulární výraz hledá volání funkce o jednom nebo dvou parametrech v podobě řetězcového literálu. Na místo řetězce se poté substituuje index kanálu, který odpovídá danému jménu. Pokud řetězec neodpovídá žádnému kanálu, použije se nevalidní index -1, pro který funkce `in()` vrátí hodnotu 0.

Tato transformace se nevztahuje pouze na volání funkce `in()`, ale na volání všech funkcí. To je korektní, protože řetězce nemají pro výpočet montáže

Výpis 2.14: Příklad definice vlastní funkce pro kód montáže

```
// Euclidean distance between channels i and j.
float dist(int i, int j, PARA) {
  return sqrt(pown(x(i) - x(j), 2) + pown(y(i) - y(j), 2) +
              pown(z(i) - z(j), 2));
}
#define dist(a_, b_) dist(a_, b_, PASS)

// Weigted average by distance d where weights = 1/(d*coeff + 1).
// For ceoff = 1 the weights for distances 0, 1, 2, ... are 1, 1/2,...
float distAverage(int i, int coeff, PARA) {
  float tmp = 0;
  for (int j = 0; j < IN_COUNT; ++j) {
    if (i != j) {
      float d = dist(i, j) * coeff + 1;
      tmp += (in(i) - in(j)) * 1 / d;
    }
  }
  return tmp;
}
#define distAverage(a_, b_) distAverage(a_, b_, PASS)
```

žádný význam. Takže se dá očekávat, že nebudou použity v žádném jiném kontextu.

2.6.3.2 Problémy s výkonem kompilace

Přechodem z kompilace všech formulí na jednou vznikl problém, kdy na slabším hardwaru, nebo při velkém počtu kanálů dochází ke značným prodlevám při inicializaci montáží.

Problém se dále komplikuje tím, že různé ovladače se chovají v tomto ohledu jinak. Při testování aplikace u zákazníka se projevilo, že při každém přepnutí montáže nastává značná prodleva. Jak se ukázalo nové verze ovladačů Nvidia[13] na mém počítači *cachují* zkompilovaný kód kernelů. Navíc tato *cache* je perzistentní. To znamená, že se drahá kompilace pro určitý zdrojový kód provede pouze jednou. Staré verze ovladačů toto neprovádí, takže se kompilace musí provádět opakovaně.

Pokusil jsem se napodobit tento přístup a implementoval jsem vlastní perzistentní úložiště kernelů v binární formě. Tato funkce se dá zapnout přepínačem `--kernelCacheSize` (viz příloha C), která určuje limit uložených záznamů. Výchozí hodnota je 0, což znamená, že tato funkce je vypnutá.

Dále jsem optimalizoval dva nejčastější případy montáží:

- kopírující montáž – montáž, která překopíruje jednu hodnotu ze vstupu

do výstupu s libovolným indexem

- identickou montáž – výchozí montáž pro po prvé otevřené soubory, která kopíruje vstupní hodnotu na stejnou pozici do výstupu

Před kompilací kódu montáže se testuje zda nejde o některý z výše uvedených speciálních typů montáže pomocí regulárních výrazů. Pokud ano, tato montáž využije jeden společný kernel, který stačí zkompilovat pouze jednou pro každý speciální typ.

Tento problém jsem se také snažil řešit pomocí paralelizace kompilace. Toto řešení ale nebylo úspěšné, protože při paralelních voláních skrze jeden OpenCL kontext ovladač kompilaci synchronizoval a nedošlo k očekávanému zrychlení. Tuto snahu jsem tudíž opustil.

2.7 Další provedené změny

Nakonec se ještě stručně zmíním o některých dalších změnách a vylepšeních provedených na projektu během práce na DP.

2.7.1 Integrace klastrovače

Klastrovač je další automatický algoritmus pro analýzu EEG dat vyvinutý ISARG[8]. Využívá výstupu spike-detektoru popsaném v sekci 2.2: rozděluje spiky a jejich skupiny/shluky do charakteristických kategorií.

Tento algoritmus je opět implementován jako skript pro Matlab. Využil jsem nástroj v Matlabu pro konverzi skriptu do C++ kódu, který jsem poté integroval do Alenky (do samostatného modulu pro zpracování signálu).

Před spuštěním klastrovače musíme nejdříve spustit spike-detektor – klastrovač využívá výsledků z poslední provedené analýzy. Výsledkem jsou vše kanálové anotace, které označují místa nalezených kategorií. Samotné kategorie jsou indikovány typy anotací.

2.7.2 Integrace rozšiřující mobilní aplikace

Jako rozšíření Alenky byla v rámci BP[32] Lenky Svobodové vyvinuta mobilní aplikace, která transformuje výstup spike-detektoru na vizualizaci četnosti detekcí na jednotlivých snímacích elektrodách. Tato technika se používá k lokalizaci zdroje spiků v mozku při krátkém, předoperačním EEG vyšetření.

Role Alenky je automatizovat tento proces propojením několika funkčních prvků – záznamu EEG, detekčního algoritmu a mobilní aplikace pro prezentaci.

Do mobilní aplikace se lze z Alenky přepnout z panelu nástrojů tlačítkem *Switch to Elko*. Přepínačem `--mode` (viz příloha C) lze zapnout tzv. tablet mód, který mění výchozí vzhled a chování některých ovládacích prvků pro snazší použití na zařízení s dotykovým displejem.

Já jsem pomohl s integrací aplikace a odladěním pro mobilní zařízení. Aplikace byla otestována na tabletu s 32-bitovým operačním systémem Windows 8.

2.7.3 Změny v zobrazování

Změnil jsem také algoritmus zobrazování. Hlavním cílem bylo zvýšení spolehlivosti a kompatibility. Vyžadován je hardware s podporou OpenGL 2.0, což je velké zlepšení oproti 4.1 z verze vyvinuté v BP.

Hlavní změny proběhly v práci s grafickou pamětí. *Cachování* signálu jsem posunul do poslední fáze zpracování, do *buffru* OpenGL. Toto pomůže na platformách, které nepodporují přímé sdílení dat mezi OpenGL a OpenCL (jako je např. virtuální počítač), kdy se vyhneme drahé synchronizaci mezi těmito dvěma technologiemi. Alokace této paměti nově probíhá *on demand*, což snižuje paměťové nároky pro typické použití.

Odstranil jsem také asynchronní načítání ze souboru. Tento kód byl příliš složitý a nepřinášel velké výkonnostní přínosy. Nakonec jsem přidal optimalizaci, která přeskakuje fázi filtrace, když jsou všechny filtry vypnuté.

2.7.4 Refaktorizace

Většina kódu zaznamenala poměrně rozsáhlou refaktorizaci. Cílem bylo zjednodušit kód a snížit duplicitu za účelem zlepšení čitelnosti a ulehčení údržby.

Jako příklad uvedu předělání kódu pod-oken manažerů. Implementace jednotlivých pod-oken sdílí funkcionalitu. Toho ale předtím nebylo využito a vznikala tím nešťastná duplicita kódu pro implementaci sloupců. Také jsem oddělil kód datových struktur pro reprezentaci metadat načtených ze souborů od logiky ovládání manažerů.

2.7.5 Undo a auto-save funkcionalita

V rámci zvýšení spolehlivosti aplikace jsem se rozhodl implementovat dva funkční prvky pro ochranu uživatelských dat.

Změny metadat se ukládají přes mechanizmus undo/redo (toto je implementace návrhového vzoru *command*[35]) podporovaný knihovnou Qt[15]. Tyto změny je možné vrátit zpět, nebo znovu aplikovat.

Tento mechanizmus podporuje také detekovat, kdy se aplikace nachází v „nečistém“ stavu. Tento stav periodicky testuji a vytvářím zálohu metadat v dočasném sekundárním souboru `.mont`. Při otevírání souboru se tento soubor hledá a aplikace vyzve uživatele k obnově dat, pokud soubor existuje.

Kapitola 3

Testování

V této kapitole popíšu, jak jsem aplikaci testoval.

Testoval jsem se čtyřmi grafickými kartami pro vyzkoušení různých ovladačů a implementací OpenCL[27]:

- Nvidia GeForce GTX 960
- Nvidia GeForce GT 540M
- AMD/ATI Radeon HD 5670
- Intel HD Graphics, integrované GPU na čipu

Tabulka 3.1 ukazuje které kombinace hardwaru a operačních systémů jsem otestoval. Otestoval jsem různé operační systémy na dvou stolních počítačích a jednom laptopu. Všechno to jsou 64-bitové počítače. Na tabletu, pro který bylo vyvinuto rozšíření Elko[32], běží 32-bitová verze Windows 8.

3.1 Jednotkové testy

Dva samostatné moduly – modul pro zpracování signálu a parsování souborů – mají vlastní jednotkové testy, které testují základní funkčnost.

Tyto testy nejsou kompletní, a proto by bylo vhodné je v budoucnosti rozšířit. Toto pomůže při potenciálním přepoužití těchto dvou modulů.

Tabulka 3.1: Testovací prostředí

	Nvidia 960	Nvidia 540M	AMD 5670	Intel HD
Ubuntu 16.04	x		x	x
Ubuntu 14.04	x		x	x
Windows 10 x64	x		x	x
Windows 8 x86				x
Windows 7 x64		x	x	x

3.2 Testovací scénář

Při testování uživatelského rozhraní jsem následoval kroky v testovacím scénáři uvedeném dále v této sekci.

Při vytváření tohoto scénáře jsem používal nástroje gcov[6] a lcov[9] pro měření pokrytí kódu. Hlavní důvod, proč jsem tyto nástroje využil je to, že jsem se chtěl ujistit, že scénář testuje všechny významné funkční prvky a že se nezapomene na rozsáhlejší části kódu.

Scénář dosahuje přibližně 90% pokrytí řádků. Výstupní zpráva o pokrytí je součástí přiloženého DVD.

1. Test zobrazování
   a) Otevření libovolného souboru
   b) Vyzkoušení všech prvků ovládajících zobrazování a kontrola, zda dostáváme očekávaný výsledek
      i. Zoom
      ii. Cross (klávesa `C`) a time line (klávesa `T`)
   c) Přidání anotace pro jeden i všechny kanály
2. Test souborů (zopakovat pro všechny 3 podporované formáty)
   a) Otevření čistého souboru (bez `.info` a `.mont` souborů)
   b) Změna stavu zobrazení, uzavření bez uložení a kontrola správného načtení stavu z `.info` souboru
   c) Změna hodnot v plovoucích pod-oknech (manažerech)
   d) Uložení (včetně uložení do primárního souboru pomocí „Save“ pro montáž)
   e) Test zálohového souboru
   f) Test správného uložení stavu aplikace a sekundárních dat
   g) Přesunutí souboru a test uložení do primárního souboru (kromě `.mat` formátu)
3. Export GDF/MAT do EDF
4. Test auto-save funkce
   a) Další změna a čekání 2 minuty na auto-save (dokud se do konzoly nevypíše zpráva o uložení)
   b) Přerušení aplikace (simulace pádu)
   c) Otevření znovu a ignorování auto-save souboru

   d) Zopakování prvních dvou bodů a akceptace auto-save souboru. Kontrola načteného stavu a okamžité uložení, což uloží stav permanentně do `.mont` souboru
   e) Znovu otevření souboru
5. Test filtrace
   a) Intuitivní test efektivity filtru vyzkoušením různých parametrů
   b) Test vizualizace filtru a nastavení specializovaného filtru
6. Test datového modelu a undo funkcionality
   a) Během změn pravidelně vraťte a poté znovu aplikujte změny pomocí undo. Kontrolujte korektní replikaci stavu
   b) Použití goto akce z Event manažeru
   c) Kopírování buněk přes menu i klávesovou zkratku
7. Test spike detektoru a klastrovače
   a) Spuštění analýzy s různými parametry
      i. Obě metody decimace
      ii. Různé nastavení filtrování a vzorkovací frekvence
   b) Porovnání výsledků analýzy na připravené sadě testovacích souborů
   c) Spuštění klastrovače
8. Test synchronizace
   a) Spusťte server a připojte lokálně klienta
   b) Přípojení druhého klienta po lokální síti
   c) Vyzkoušejte různé ovládací prvky a kontrolujte, zda se děje to co má
9. Test montáží
   a) Test různých vzorců (včetně špatných/nesmyslných) a použití dialogu pro editování formulí
   b) přidání automatické montáže
   c) přidání šablony montáže
   d) export montáže do
10. Přepnutí do Elka[32]
11. Test parametrů

a) `--help`, `--clInfo`, `--glInfo`, `--version`

b) `--kernelCacheSize`

c) `--config`

d) `--gl20`

e) `--gpuMemorySize`

f) `--fileCacheSize`

3.3 Problémy odhalené při testování

Při finálním testování podle scénáře jsem odhalil následující problémy:

- Aplikace občas padá při mazání aktuálně vybrané montáže v tabulce pod-okna *Montage Manager*.
- Po obnovení zálohy z auto-save souboru je aplikace v „čistém“ stavu a dovoluje zavření bez výzvy k uložení změn. Toto může vést k nechtěné ztrátě dat.
- Export do EDF formátu může pro velké soubory trvat velmi dlouho. Aplikace by během této operace měla uživatele informovat o průběhu pomocí ovládacího prvku *progress bar*.
- Dialog pro sledování průběhu analýzy spike-detektoru nefunguje správně pro krátké soubory.

Pouze první problém je kritický a pokusím se ho odstranit co nejdříve v příští verzi. Ostatní zkusím zapracovat do některých z budoucích verzí.

3.4 Test výkonu zobrazení

V této sekci otestuji následující nefunkční požadavek ze zadání:

Výkonnostní požadavky na aplikaci:

- *Cílem je zvládat zobrazení až 10 sekund signálu o 8 kHz s 2048 kanály.*
- *Typický objem dat [je] 10 s na stránku, 128 kanálů, 1 kHz.*

Ověření tohoto požadavku provedu tak, že porovnám čas potřebný pro překreslení hlavního zobrazení pro různá prostředí. Simuluji typické chování uživatele za použití testovacího souboru `sample1kHz128c.mat`. `Typický objem dat` znamená že aplikace musí zpracovat přibližně $10 \times 128 \times 1000 \times 4 \approx 5MB$ dat.

Alenka podporuje přepínač `--printTiming`, který zapíná výpis časových intervalů potřebných pro obnovení ovládacího prvku `Canvas`. Při měření jsem provedl následující kroky:

Tabulka 3.2: Konfigurace testovacího HW s vysokým výkonem

Operační systém	Ubuntu 16.04
Procesor	Intel Core i7-2600K; 4 jádra; 8 vláken; 4,6 GHz
Paměť	32 GB
GPU	Nvidia GeForce GTX 960; 4 GB; 2365 GFLOPS[37]

Tabulka 3.3: Konfigurace testovacího HW s nízkým výkonem

Operační systém	Windows 10
Procesor	Intel Celeron G1840; 2 jádra; 2,8 GHz
Paměť	8 GB
GPU	AMD/ATI Radeon HD 5670; 1 GB; 620 GFLOPS[40]

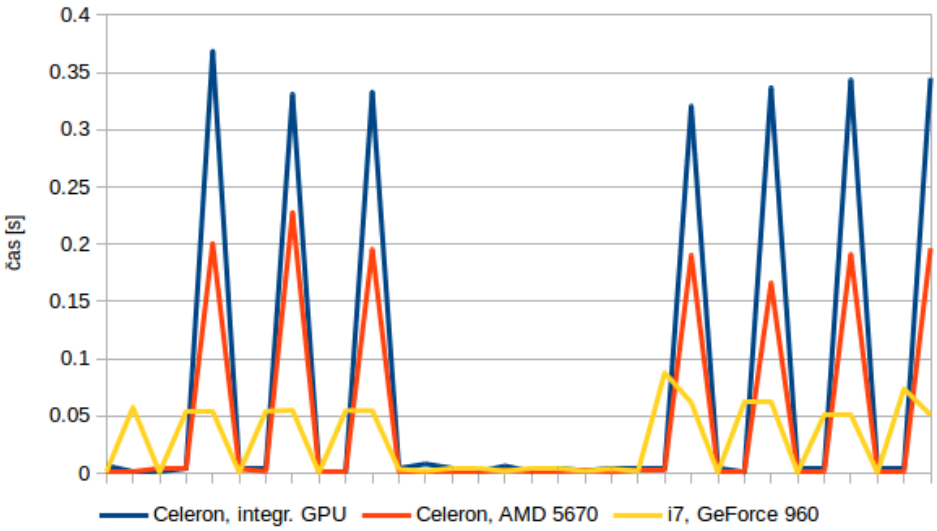
1. Spuštění Alenky s výchozím nastavením z terminálu. Zobrazení je nastaveno na 10 sekund a filtr je vypnutý.
2. 10 posunů pomocí klávesy *PageDown* doprava
3. 10 posunů pomocí klávesy *PageUp* zpátky doleva
4. Zapnutí filtru zaškrtnutím *Notch* v panelu nástrojů
5. Znovu 10 posunů pomocí klávesy *PageDown* doprava
6. Násilné ukončení pomocí signálu (`Ctrl + C`) z terminálu

Testoval jsem na dvou počítačích. První je vysoko výkonnostní pracovní stanice (viz tabulka 3.2). Toto je počítač, který můžeme očekávat pouze v profesionálním prostředí.

Jako druhý (viz tabulka 3.3) jsem zvolil obyčejný a laciný – cenu srovnatelného, nového počítače odhaduji pod 10 tisíc korun – stolní počítač. Jeho výkon (hlavně kvůli mizernému procesoru) a cena jsou nesrovnatelné se strojem popsaným výše. Tento hardware jsem zvolil přesně proto, abych demonstroval, že aplikaci lze pohodlně používat i na běžném počítači, nebo laptopu.

Na obrázku 3.1 je srovnání výkonu různých grafických karet. Můžeme vidět, že měření prochází třemi fázemi v každé z nichž se provádí různé výpočetní operace:

1. Při prvním posouvání doprava se čte signál z disku a kopíruje do paměti GPU.
2. Při posouvání zpět doleva už je signál v paměti na zařízení a tak se měří pouze vykreslování pomocí OpenGL[28].
3. Při zapnutí filtru se musí paměť na GPU vyčistit. Nyní při posouvání doprava musí aplikace navíc provést filtraci.



Obrázek 3.1: Měření výkonu vykreslování: různá GPU

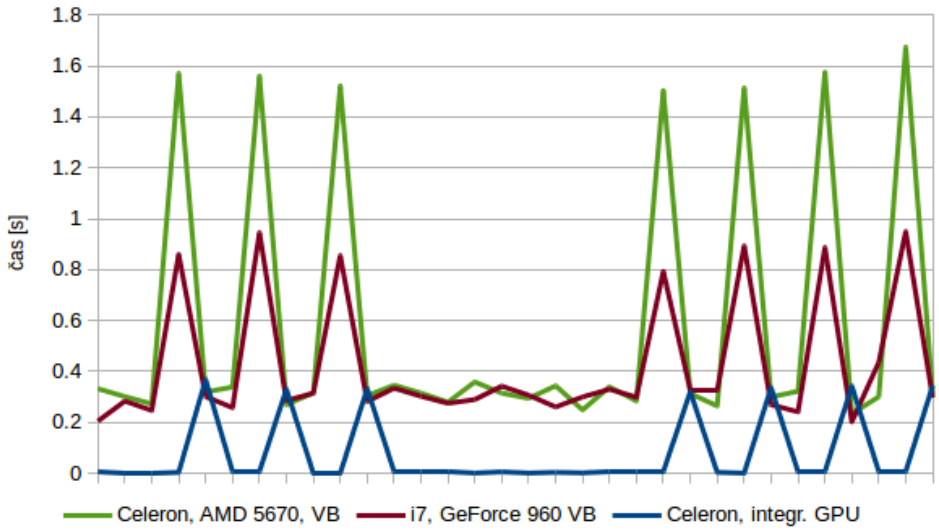
Z obrázku 3.1 můžeme učinit několik závěrů:

- Pro výkonou grafickou kartu zpoždění nikdy nepřesáhne desetinu sekundy, což se dá považovat za dostatečně malou prodlevu pro interaktivní aplikace.
- Pro méně výkoné GPU začíná být tento objem dat problematický, ale stále je aplikace dobře použitelná.
- Samotné vykreslování křivky signálu má téměř zanedbatelnou složitost.
- Jak se zdá, filtrování nemá výrazný vliv na odezvu zobrazení.

Na obrázku 3.2 je srovnání nejpomalejší grafické karty – integrované na CPU – s virtuálním počítačem běžícím na obou testovacích strojích. Použil jsem virtualizační software VirtualBox[17]. Nainstaloval jsem *Přídavky pro hosta* a nastavil jsem všechny výkonnostní parametry na maximum (kromě akcelerace 2D a 3D grafiky, s kterou mám špatné zkušenosti).

Zde můžeme vidět, že virtuální prostředí má poměrně velký vliv na výkon aplikace. Zejména na slabším hardwaru dochází k už poměrně nepříjemným prodlevám. Pro využití ve virtuálním počítači bych tedy doporučil používat soubory s menší datovou zátěží.

Pokusil jsem se také otestovat v zadání zmíněný cílový objem dat (soubor `sample8kHz2048c.mat`). Ten je 128 krát větší než typický objem, tedy minimálně $10 \times 2048 \times 8000 \times 4 \approx 655MB$ pro jednu obrazovku. Kvůli vykreslování anotací se navíc musí signál v paměti GPU duplikovat. To



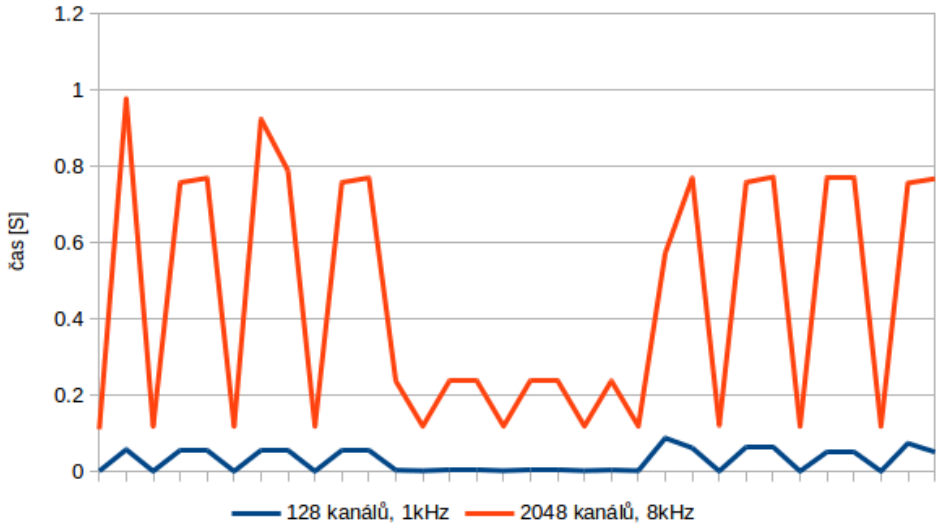
Obrázek 3.2: Měření výkonu vykreslování: virtuální prostředí

nám dává poněkud extrémní paměťové nároky. Přesto se mi povedlo soubor s těmito parametry otevřít na výkoném počítači popsaném v tabulce 3.2.

Pokud se data pro celé zobrazení nevejdou do paměti, dochází k podobnému chování jako při nedostatku RAM (tzv. *thrashing*), kdy zpoždění způsobené odkládáním na disk řádově sníží výkon. Pokud toto nastane pro naší aplikaci, výkon je velice negativně ovlivněn. Tomuto se tedy chceme za každou cenu vyhnout. To také znamená, že velikost paměti GPU je jedním z největších limitujících faktorů aplikace.

Na obrázku 3.3 vidíme srovnání měření typické a cílové datové zátěže. Při posunutí zobrazení dochází přibližně k sekundové prodlevě. To je asi dvacetinásobné zhoršení při 128 násobném nárůstu dat. Zde se projevuje úžasná paralelní síla grafických akcelerátorů. Je zcela realistické očekávat, že podobná aplikace provádějící celý výpočet na CPU zvládne typický objem dat; cílová zátěž (bez redukce dat. např. decimací) je ale pro běžné CPU neřešitelný problém.

Pokud pomineme fakt, že zobrazení s 2048 kanály je pouze černá obrazovka – musíme si uvědomit, že máme dvakrát více kanálů než je rozlišení na běžném monitoru – mohu prohlásit, že je aplikace i v tomto extrémním případě použitelná. Považuji tedy nefunkční požadavek na výkon za splněný.



Obrázek 3.3: Měření výkonu vykreslování: cílový objem dat

Závěr

Splnil jsem všechny funkční i nefunkční požadavky obsažené v zadání. Navíc jsem implementoval mnoho dalších funkcí a vylepšení nad rámec původního zadání. Dovoluji si tedy prohlásit cíl práce za splněný.

Tato diplomová práce pro mě byla cennou zkušeností a vřele doufám, že její produkt bude v budoucnu k užitku.

Literatura

[1] AddressSanitizer. *Clang 7 documentation* [online]. 2018 [cit. 2018-01-31]. Dostupné z: `https://clang.llvm.org/docs/AddressSanitizer.html`

[2] ClangFormat. *Clang 7 documentation* [online]. 2018 [cit. 2018-02-04]. Dostupné z: `https://clang.llvm.org/docs/ClangFormat.html`

[3] *CMake* [online]. 2018 [cit. 2018-02-04]. Dostupné z: `https://cmake.org/`

[4] *Eigen* [online]. 2018 [cit. 2018-02-01]. Dostupné z: `https://eigen.tuxfamily.org/`

[5] *European Data Format* [online]. [cit. 2018-02-02]. Dostupné z: `https://www.edfplus.info/`

[6] Gcov: a Test Coverage Program. *GCC: the GNU Compiler Collection* [online]. 2018 [cit. 2018-02-12]. Dostupné z: `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`

[7] HDF5. *The HDF Group* [online]. 2017 [cit. 2018-02-03]. Dostupné z: `https://support.hdfgroup.org/HDF5/`

[8] *ISARG: Intracranial Signal Analysis Research Group* [online]. [2013] [cit. 2018-02-01]. Dostupné z: `https://isarg.feld.cvut.cz/`

[9] LCOV: the LTP GCOV extension. *SourceForge* [online]. 2018, December 20, 2016 [cit. 2018-02-12]. Dostupné z: `http://ltp.sourceforge.net/coverage/lcov.php`

[10] MAT-File Versions. *MathWorks* [online]. 2018 [cit. 2018-02-03]. Dostupné z: `https://www.mathworks.com/help/matlab/import_export/mat-file-versions.html`

[11] MATIO. *GitHub* [online]. 2018 [cit. 2018-02-03]. Dostupné z: `https://github.com/tbeu/matio`

[12] MATLAB. *MathWorks* [online]. USA: The MathWorks, 2018 [cit. 2018-02-04]. Dostupné z: https://www.mathworks.com/products/matlab.html

[13] *Nvidia* [online]. 2018 [cit. 2018-02-12]. Dostupné z: https://www.nvidia.com

[14] Qt Creator IDE: Making software development fast, easy & fun. *Qt* [online]. 2018 [cit. 2018-02-04]. Dostupné z: https://www.qt.io/qt-features-libraries-apis-tools-and-ide/#ide

[15] QUndoStack Class. *Qt Documentation* [online]. 2018 [cit. 2018-02-13]. Dostupné z: http://doc.qt.io/qt-5/qundostack.html

[16] *Valgrind* [online]. 2017 [cit. 2018-01-31]. Dostupné z: http://valgrind.org/

[17] *VirtualBox* [online]. 2018 [cit. 2018-02-12]. Dostupné z: https://www.virtualbox.org/

[18] *WxWidgets: Cross-Platform GUI Library* [online]. 2018 [cit. 2018-01-31]. Dostupné z: https://www.wxwidgets.org/

[19] BAI, Haoqi. SignalResampler. *GitHub* [online]. 2018 [cit. 2018-01-31]. Dostupné z: https://github.com/terrygta/SignalResampler

[20] BÁRTA, Martin. Alenka: A Visualisation System for Biosignals. *GitHub* [online]. 2018 [cit. 2018-02-04]. Dostupné z: https://github.com/machta/Alenka

[21] BÁRTA, Martin. *Specializovaný systém pro zobrazování biologických signálů pacientů zařazených do epilepto-chirurgického programu: rozšiřovací moduly*. Praha, 2015. Bakalářská práce. ČVUT.

[22] BEELEN, Teunis van. EDFlib. *Teunis van Beelen* [online]. 2018 [cit. 2018-02-02]. Dostupné z: https://github.com/Teuniz/EDFlib

[23] DRÁBEK, Jakub. *Multiplatformní implementace spike detektoru pro intraoperační elektrokortikografii*. Praha, 2015. Bakalářská práce. ČVUT.

[24] FETTE, Ian a Alexey MELNIKOV. RFC 6455: The WebSocket Protocol. *IETF Tools* [online]. 2011 [cit. 2018-01-28]. Dostupné z: https://tools.ietf.org/html/rfc6455

[25] JANČA, Radek, Petr JEŽDÍK a et al. Detection of Interictal Epileptiform Discharges Using Signal Envelope Distribution Modelling: Application to Epileptic and Non-Epileptic Intracranial Recordings. *Brain Topography* [online]. 2015, (28), 172-183 [cit. 2018-01-31]. Dostupné z: https://link.springer.com/article/10.1007%2Fs10548-014-0379-1

[26] KAPOULKINE, Arseny. *Pugixml* [online]. 2016 [cit. 2018-02-04]. Dostupné z: https://pugixml.org/

[27] KHRONOS GROUP. *KHRONOS GROUP: The open standard for parallel programming of heterogeneous systems* [online]. c2015 [cit. 2018-01-30]. Dostupné z: https://www.khronos.org/opencl/

[28] KHRONOS GROUP. *OpenGL* [online]. c1997-2015 [cit. 2018-01-30]. Dostupné z: https://www.opengl.org/

[29] LOPO, Erik de Castro. Libsamplerate. *GitHub* [online]. 2018 [cit. 2018-01-31]. Dostupné z: https://github.com/erikd/libsamplerate

[30] Design of Linear-Phase FIR Filters by the Frequency-Sampling Method. PROAKIS, John G a Dimitris G MANOLAKIS. *Digital signal processing*. 4th ed. Upper Saddle River: Pearson Prentice Hall, c2007, s. 671-678. ISBN 0-13-187374-1.

[31] SCHLÖGL, Alois. *GDF: A GENERAL DATAFORMAT FOR BIOSIGNALS* [online]. Graz, 2013 [cit. 2018-01-28]. Dostupné z: https://arxiv.org/abs/cs/0608052. Paper. Graz University of Technology.

[32] SVOBODOVÁ, Lenka. *Mobilní aplikace pro automatickou detekci epileptiformních výbojů v intraoperační elektrokortikografii*. Praha, 2017. Bakalářská práce. ČVUT.

[33] THE QT COMPANY. *Qt* [online]. 2015 [cit. 2018-01-29]. Dostupné z: http://www.qt.io/

[34] QChartView Class. *Qt Documentation* [online]. The Qt Company, 2017 [cit. 2018-02-01]. Dostupné z: https://doc.qt.io/archives/qt-5.8/qchartview.html

[35] Command pattern. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2018-02-13]. Dostupné z: https://en.wikipedia.org/wiki/Command_pattern

[36] Elektroencefalogram. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2018-01-30]. Dostupné z: https://cs.wikipedia.org/wiki/Elektroencefalogram

[37] GeForce 900 series. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2018-02-12]. Dostupné z: https://en.wikipedia.org/wiki/GeForce_900_series

[38] MacOS. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2018-02-04]. Dostupné z: https://en.wikipedia.org/wiki/MacOS

[39] Neuromarketing. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2018-01-30]. Dostupné z: https://cs.wikipedia.org/wiki/Neuromarketing

[40] Radeon HD 5000 Series. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2018-02-12]. Dostupné z: https://en.wikipedia.org/wiki/Radeon_HD_5000_Series

Příloha **A**

Seznam použitých zkratek

CPU Central processing unit

BP Bakalářská práce

DP Diplomová práce

EEG Elektroencefalogram

FFT Fast Fourier transform

FIR Finite impulse response

FN Fakulní nemocnice

GPU Graphics processing unit

GUI Graphical user interface

IDE Integrated development environment

ISARG Intracranial Signal Analysis Research Group

OS Operační systém

RAM Random access memory

UML Unified modeling language

XML Extensible markup language

Příloha B

Ovládání

Nápověda k nastavení se zobrazí specifikováním přepínače `--help` (viz příloha C) při spuštění programu z příkazové řádky. Toto nastavení lze specifikovat, buď přímo z příkazové řádky, nebo v konfiguračním souboru.

Výběr montáže k zobrazení a aktivní typ anotace se mění pomocí panelu nástrojů Select.

Frekvence filtrů se nastavují pomocí panelu nástrojů Filter.

V oknech managerů se nastavují různé doplňující informace k vytvářeným objektům. Za zmínku zejména stojí sloupec Save v MontageManager, který řídí, zda se anotace dané montáže ukládají do primárního souboru.

B.1 Klávesové zkratky pro ovládání hlavního zobrazení

`MouseWheel` posunování po časové ose

`PageUp` posunování po časové ose

`PageDown` posunování po časové ose

`Šipky doleva a doprava` posunování po časové ose

`Ctrl + MouseWheel` změna amplitudy jedné stopy

`Shift + MouseWheel` změna amplitud všech stop

`Alt + MouseWheel` změna přiblížení zobrazení

`Ctrl + LeftMouse` přidání jedno-kanálové anotace

`Shift + LeftMouse` přidání vše-kanálové anotace

`C` zapnutí/vypnutí kříže

`T` posunutí pozičního indikátor na pozici kurzoru

`Ctrl + Z` zpět/undo

`Ctrl + Shift + Z` znovu/redo

B.2 Klávesové zkratky pro ovládání oken managerů

`Ctrl + C` zkopírování označených buněk do schránky

`Ctrl + V` vložení obsahu ze schránky na pozici aktuálně označené buňky

`Delete` vymazání označených řádků

`G` posunutí hlavního zobrazení na začátek označené anotace

Příloha C

Nápověda programu Alenka

Následuje výstup příkazu `./Alenka --help`.

```
Usage:
  Alenka [OPTION]... [FILE]...
  Alenka --spikedet OUTPUT_FILE [SPIKEDET_SETTINGS]... FILE [FILE]...
  Alenka --help|--clInfo|--version
Command line options:
  --help                          help message
  --config path                   override default config file path
  --spikedet OUTPUT_FILE          Spikedet only mode
  --clInfo                        print OpenCL platform and device info
  --glInfo                        print OpenGL info
  --version                       print version number
  --printTiming                   print the time it took to redraw Canvas
Configuration:
  --mode val (=desktop)           desktop|tablet|tablet-full
  --locale lang (=en_us)          mostly controls decimal number format
  --uncalibratedGDF bool (=0)     assume uncalibrated data in GDF
  --autosave seconds (=120)       interval between saves; 0 to disable
  --kernelCacheSize count (=0)    if 0, the existing file is removed
  --kernelCacheDir path           default is install dir
  --gl20 bool (=0)                use OpenGL 2.0 instead of 3.0
  --gl43 bool (=0)                use OpenGL 4.3 instead of 3.0; disabled
  --cl11 bool (=0)                use OpenCL 1.1 instead of 1.2
  --glSharing bool (=1)           use cl_khr_gl_sharing extension
  --clPlatform ID (=0)            select OpenCL platform
  --clDevice ID (=0)              OpenCL device
  --blockSize val (=32768)        samples per channel per block
  --gpuMemorySize MB (=0)         allowed GPU memory; 0 means no limit
  --parProc val (=2)              parallel signal processor queue count
  --fileCacheSize MB (=0)         allowed RAM for caching signal files
  --notchFrequency f (=50)        power interference filter
  --resOptions list (=1 2 ...)    resolution combo options
  --screenPath path               screenshot output dir path
  --screenType type (=jpg)        screenshot file type; jpg, png, or bmp
  --matData val...                data var names for MAT files; default is 'd'
  --matFs val (=fs)               sample rate var name
```

```
  --matMults val (=mults)         channel multipliers var name
  --matDate val (=tabs)           start date var name
  --matLabel val (=header.label)  labels var name
  --matEvtPos val (=out.pos)      event position var name in seconds
  --matEvtDur val (=out.dur)      event duration var name in seconds
  --matEvtChan val (=out.chan)    one-based event channel index var name
Spikedet settings:
  --fl f (=10)                    lowpass filter frequency
  --fh f (=60)                    highpass filter frequency
  --k1 val (=3.65)                K1
  --k2 val (=3.65)                K2
  --k3 val (=0)                   K3
  --w val (=5)                    winsize
  --n val (=4)                    noverlap
  --buf seconds (=300)            buffering
  --h f (=50)                     main hum. freq.
  --dt val (=0.005)               discharge tol.
  --pt val (=0.12)                polyspike union time
  --dec f (=200)                  decimation
  --sed seconds (=0.1)            spike event duration
  --osd bool (=1)                 use orginal Spikedet implementation
```

Příloha D

Build Instructions

Tato příloha byla vygenerována z wiki stránky projektu na GitHubu[20].

This page describes steps needed to build Alenka from source. The command-line examples can be run using bash (or git-bash on Windows which is included in git's installer).

Install Qt via the installer on their website. Select the `Qt 5.8 msvc2015 64/32-bit` package for Windows, or `Desktop gcc 64/32-bit` for Linux and Mac. Also select the `QtCharts` module.

Then download the third-party libraries using the preprepared script:

```
cd libraries
./download-libraries.sh
cd ..
```

You can use cmake-gui in place of cmake to change some of the following options to customize the build configuration:

- set `CMAKE_PREFIX_PATH` to specify Qt's install directory if needed (for example `/opt/Qt/5.8/gcc_64` on Linux, `C:/Qt/5.8/msvc2015_64` on Windows)
- on some systems you need to add to `CMAKE_PREFIX_PATH` the location of AMD APP SDK (e.g. on Linux set it to `/opt/Qt/5.8/gcc_64; /opt/AMDAPPSDK-3.0/lib/x86_64/sdk`)
- set `CMAKE_BUILD_TYPE` to switch between debug and release
- check ALENKA_STATIC_LINK to make a standalone Linux binary that works on both Ubuntu 14 and 16
- check ALENKA_BUILD_TESTS to build unit-tests

The rest of the instructions are OS specific.

D.1 Linux

First install the required tools and matio library. On some Linux distributions you need to also install OpenGL headers. On a Debian-like system you can do this by running:

```
sudo apt install git cmake-gui build-essential libmatio-dev curl
libgl1-mesa-dev
```

Then use the usual cmake/make combination to make an out-of-source build. From the repository's root directory use:

```
mkdir build-Release && cd build-Release
cmake -DCMAKE_BUILD_TYPE=Release ..
make
```

That should be all you need to do to build Alenka. Additionally you can use Qt Creator (or some other IDE) to open CMakeLists.txt file as a project. During project configuration redirect to the build directory we have just created. Then you can rebuild, run and debug the program directly from the IDE.

D.2 Windows

MSVC++ compiler can be acquired by installing Visual C++ Build Tools 2015. Choose `Custom Installation`, and uncheck all options but `Windows 8.1 SDK`. If you already have Visual Studio 2015, you probably don't need to install this.

Now you have two options: you can use Qt Creator to generate a makefile-based project, or use cmake to generate a Visual Studio solution. The first approach can use only a single thread for compilation which can lead to some annoying build times. By choosing the second option you lose some of Qt Creator's functionality designed to work specifically with Qt.

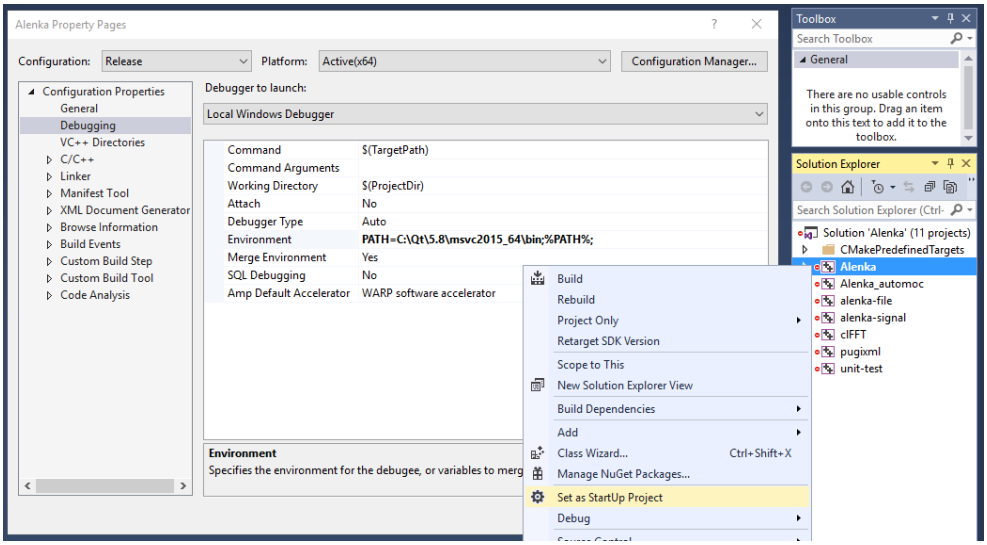
D.2.1 Qt Creator Project

Open CMakeLists.txt via Qt Creator and fill the configuration form that appears. Qt Creator with its default settings should take care of everything.

D.2.2 Visual Studio Solution

These commands generate a MS Visual Studio solution and then build Alenka.

```
mkdir build-Release && cd build-Release
cmake -G "Visual Studio 14 2015 Win64" ..
cmake --build . --config Release
```



Obrázek D.1: Nastavení projektu ve Visual Studiu

Or you can skip the cmake build step and open Alenka.sln located in the build directory. Then select Alenka as the StartUp project, and set PATH to contain Qt's library installation directory. (See the picture D.1 for details. You can also set PATH system wide as an environment variable). Now you can run Alenka via the debugger button or F5.

Příloha **E**

Obsah přiloženého DVD

- `DP_Bárta_Martin_2018.pdf` ..... text práce ve formátu PDF
- `Ubu14-Alenka.zip` ..... obraz virtuálního počítače s předinst. Alenkou
- `doc`
  - `index.html` ..... domovská stránka dokumentace
- `exe` ..... distribuční balíčky pro Windows a Ubuntu
- `samples` ..... testovací soubory pro vyzkoušení programu
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `testCovRep` ..... zpráva o pokrytí testy