Ovládání laboratorní úlohy v prostředí Python

Control of laboratory task in Python

BAKALÁŘSKÁ PRÁCE

Studijní program: Bakalářský studijní program "STROJÍRENSTVÍ"

Studijní obor: Informační a automatizační technika

Vedoucí práce: Ing. Ivo Bukovský, Ph.D.

Jakub Čepela

Chybějící přílohy na vyžádání

OBSAH

1. Úvod 11
2. Laboratorní úloha „Batyskaf“ 12
    2.1. Popis realizace úlohy 12
    2.2. Současné řešení ovládání úlohy 13
        2.2.1. Simulační model 15
    2.3. Současné řešení komunikace úlohy s PC 16
3. Programovací jazyk Python 18
    3.1. Proč Python 18
    3.2. Historie a vývoj 20
    3.3. Používané verze 22
    3.4. Implementace a multiplatformnost 24
    3.5. Komunikační možnosti 25
    3.6. Řízení a regulace 28
    3.7. Grafické uživatelské rozhraní 29
4. Realizace ovládací aplikace pro "Batyskaf" 34
    4.1. Struktura ovládacího programu pro Batyskaf 34
    4.2. Výběr modulů 35
    4.3. Návrh řešení komunikace s úlohou 36
    4.4. Zpracování získaných dat 39
    4.5. Grafické uživatelské rozhraní 42
    4.6. Popis výsledné aplikace a možnosti zlepšení 46
5. Závěr 54
6. Použité zdroje 55
7. Přílohy (na vyžádání).
    7.1. Zdrojový kód
    7.2. CD s prací a programem v elektronické podobě.

1. ÚVOD

První část této práce popisuje současné programové i hardwarové řešení úlohy Batyskaf. Dále zkoumá možnosti jazyka Python, jako prostředí pro návrh a vývoj aplikací použitelných pro ovládání úloh v reálném čase. V poslední části práce popisuji návrh a realizaci ovládacího programu v prostředí Python, za pomoci nabytých znalostí z předchozích částí práce.

Návrh ovládací aplikace obnáší řešení programu na několika úrovních. Na nejzákladnější úrovni je třeba zajistit komunikaci se zařízením, v mém případě byla komunikace řešena přes sériovou linku. Existuje řada dalších možností přenosu dat pro laboratorní a měřící účely, například ethernetová síť, USB nebo i bezdrátové technologie jako bluetooth. V konečném důsledku je třeba zajistit komunikaci s uživatelem, která je nejpohodlnější a nejintuitivnější přes grafické rozhraní.

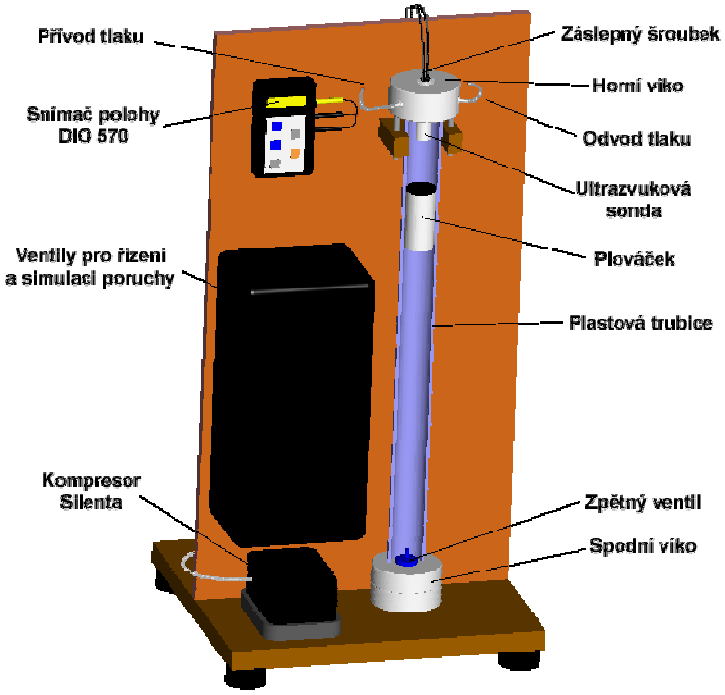
Jazyk Python se často používá jako skriptovací jazyk pro webové aplikace, pro 3D a 2D grafické balíčky a nespočet dalších rozsáhlých programů. V této pozici ovšem slouží jen jako doplněk programu napsaného v jiném jazyce. Vývojáři si ho oblíbili spíše jako skriptovací jazyk právě díky jeho jednoduchosti i pro běžné uživatele, kteří nejsou nebo nechtějí být programátory. Aplikace napsané pouze v Pythonu jsou relativně vzácné a běžný uživatel se s nimi téměř nesetká. Ještě méně se používá v oblasti programů pracujících v reálném čase, na tuto oblast se ve své práci zaměřím. Právě přehlednost a čitelnost kódu Pythonu byla jedním z faktorů, proč jsem si tento jazyk vybral pro bakalářskou práci. Také jsem si chtěl vyzkoušet sám naprogramovat větší aplikaci a potíže s tím spojené. Abych mohl možnosti jazyka vyzkoušet a ukázat na praktickém příkladu, tak jsem v rámci práce napsal nástroj pro ovládání laboratorní úlohy „Batyskaf“.

Doufám, že i tato práce přispěje k rozšíření povědomí o jazyku Python na českých vysokých školách i mezi veřejností, kde zatím není příliš rozšířen. Díky své jednoduchosti je přímo předurčen jak pro výuku programování, tak pro vědecké účely, kdy je často třeba rychle napsat a doladit různé ovládací a regulační aplikace.

2. LABORATORNÍ ÚLOHA „BATYSKAF“

Tato část práce ukazuje současné hardwarové i programové řešení úlohy.

2.1. Popis realizace úlohy



Obr. 1 - Úloha Batyskaf [1]

Laboratorní úloha Batyskaf je umístěna v laboratoři ústavu Přístrojové a řídící techniky odboru Automatické řízení a inženýrské informatiky. Je zde využívána pro výuku studentů, kteří se zde seznamují s návrhem řízení zajišťující regulaci polohy plováčku. Stručný popis funkce úlohy z článku Prof. Ing. Milana Hofreitera, CSc., Netradiční laboratorní modely pro výuku automatického řízení [2]:

„Hlavním konstrukčním prvkem laboratorního modelu Batyskaf (Obr. 1) je vertikálně umístěná plastová trubice naplněná vodou, v níž se pohybuje plováček, tj. dutý válec s otevřeným dnem vyplněný vzduchem a vodou, přičemž podíl mezi objemem vzduchu a vody v plováčku závisí na okolním hydrostatickém tlaku. Pohyb plováčku je řízen pomocí tlaku vzduchu přiváděného kompresorem do trubice nad hladinu vodního sloupce. Změna tlaku vzduchu nad hladinou způsobí změnu objemu vzduchu uzavřeného v plováčku, čímž podle Archimédova zákona dojde ke změně silových poměrů působících na plováček. Tlak vzduchu nad hladinou je závislý na otevření dvou jehlových ventilů. Jeden slouží pro vykonávání akčního zásahu a druhý pro simulaci poruchy. Otvírání a zavírání ventilů je realizováno servomotory, které jsou ovládány z počítače pomocí programu pracujícím v prostředí Matlab/Simulink přes sériovou komunikaci. Tlak vzduchu v trubici je snímán tlakovým čidlem a předáván

přes sériový port do počítače. Poloha plováčku je měřena vysílací/přijímací ultrazvukovou sondou a za pomocí měřicí jednotky, propojené sériovým kabelem s počítačem, jsou údaje předávány do programu pracujícím v prostředí Matlab/Simulink.“

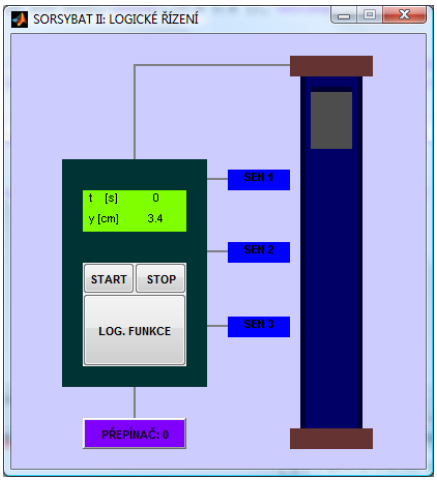
2.2. Současné řešení ovládání úlohy

Program pro ovládání úlohy zpracoval již před několika lety Ing. Jaroslav Jirkovský jako součást své diplomové práce [3]. Simulační program se nazývá Sorsybat II, v laboratoři se nachází verze 12.2, je navržen a pracuje v prostředí MATLAB - Simulink. Název je složen ze slov „Soubor Regulací Systému Batyskaf“, který i přesně vystihuje funkci programu. Lze si vyzkoušet následují metody regulace:

Logické řízení

Spojité řízení

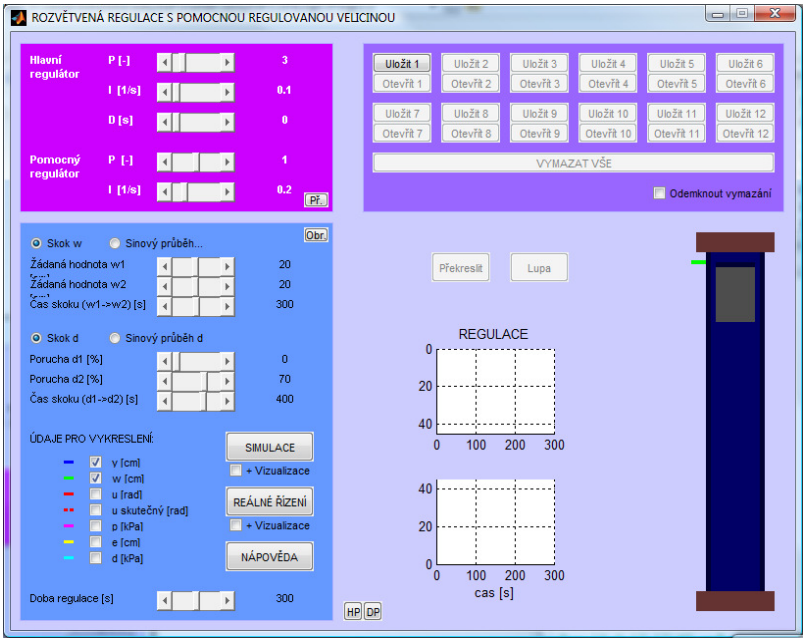
1. Dvoupolohová regulace
    a) Dvoupolohová regulace s plným akčním zásahem
    b) Dvoupolohová regulace s polovičním akčním zásahem
3. Čtyř-polohová regulace
4. PID regulace
    a) P regulace
    b) I regulace
    c) PI regulace
    d) PD regulace
    e) PID regulace
5. Nespojitá číslicová PI regulace
6. Jednorozměrová rozvětvená regulace
    a) S pomocnou regulovanou veličinou
    b) S měřením poruchy



Obr. 2 – Uživatelské rozhraní pro logické řízení [3]

Logické řízení je založeno na simulaci tří hloubkových senzorů a jednoho přepínače. Senzory vrací hodnoty logická „1“ nebo „0“ podle polohy plováčku, spínací polohy jsou 15, 25 a 35 cm. Přepínač lze ovládat z uživatelského rozhraní. Samotné řízení zde není pevně naprogramováno, ale tento krok je ponechán na studentovi. Pod tlačítkem „Log. funkce“ se skrývá velice intuitivní rozhraní, kde jsou předdefinovány pouze vstupy (senzory a tlačítko) a výstup (řídící servo). Studentovi jsou zde nabídnuty základní logické funkční bloky – AND, OR, NAND, NOR, NOT a SR klopný obvod, pomocí kterých lze sestavit požadovanou řídící funkci. Lze zde například sestavovat úlohy typu, aby se plováček pohyboval mezi polohami 15 a 25 cm, pokud je přepínač stisknutý a mezi polohami 25 a 35 cm, pokud není. Samotná regulace polohy plováčku zde realizovatelná není, protože je zde výstupem logická „1“ nebo „0“, což může odpovídat stavům, kdy plováček klesá nebo stoupá [4].

Cílem spojité regulace je co nejpřesněji a nejrychleji dosáhnout žádanou hodnotu, v tomto případě polohu plováčku. Toto umožňují všechny metody regulace, ale značně se liší jejich přesnost a rychlost.



Obr. 3 – Uživatelské rozhraní rozvětvené regulace [3]

Dvoupolohová regulace je velice jednoduchá a nevyžaduje prvek s plynule měnitelným zásahem, ale má kmitavý regulační pochod s delší dobou ustálení.

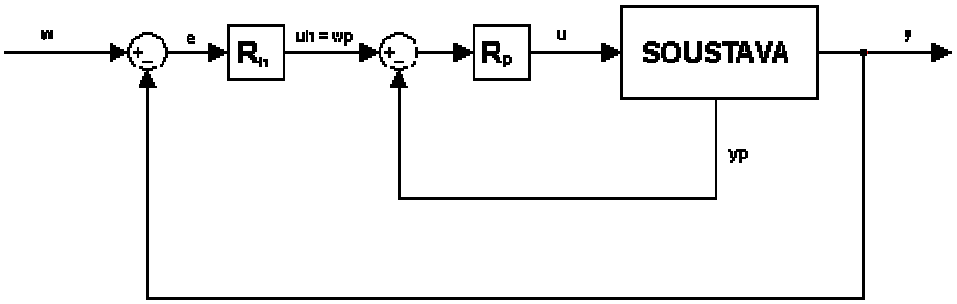
Čtyř-polohová regulace je též velice jednoduchá a nevyžaduje prvek s plynule měnitelným zásahem, regulační pochod je méně kmitavý než u dvoupolohové regulace, ale je více citlivá na poruchu.

P regulátor dosahuje při správném nastavení parametru zesílení hladkého průběhu regulace, ale je nutný prvek s plynule měnitelným zásahem a po regulaci zůstává trvalá regulační odchylka.

Pro PI regulátor existují propracované metody nastavování parametrů, dosahuje hladkých průběhů regulace a pracuje bez trvalé regulační odchylky, ale je nutný prvek s plynule měnitelným akčním zásahem a odstranění poruchy je zdlouhavé.

PID regulátor je shodný s PI regulátorem, jen obsahuje navíc derivační složku, která zrychluje reakci na změnu žádané veličiny.

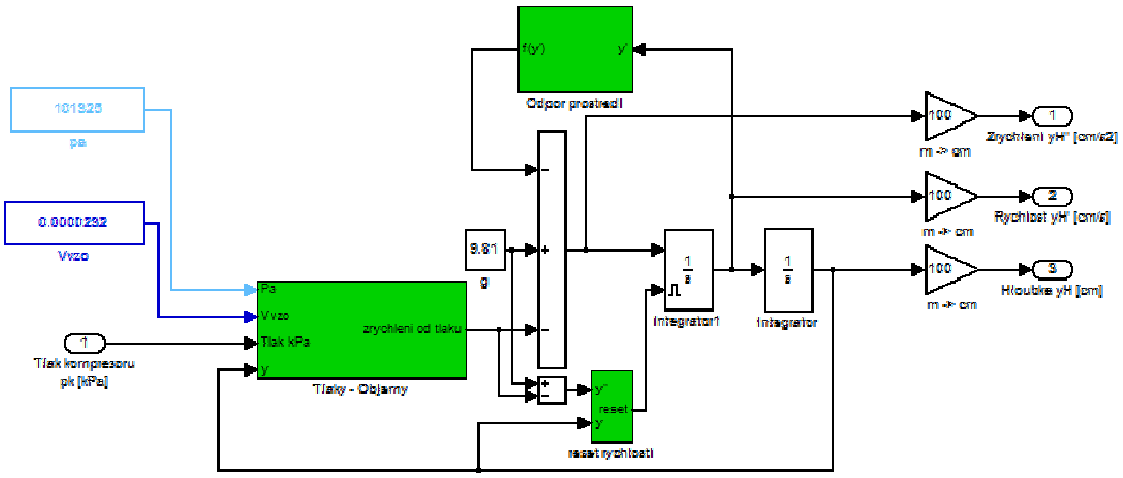
Rozvětvená regulace je v tomto případě realizována zapojením PID a PI regulátoru za sebou, což by mělo přinést zkvalitnění regulačního procesu. Umožňuje rychlé odstranění poruchy s nepatrným vlivem na průběh regulované veličiny, ale je nutné zajistit odměřování pomocné regulované veličiny, v tomto případě tlak nad hladinou. Také je nákladnější, protože vyžaduje více regulátorů.



Obr. 4 - Schéma rozvětvené regulace s pomocnou veličinou [1]

2.2.1. Simulační model

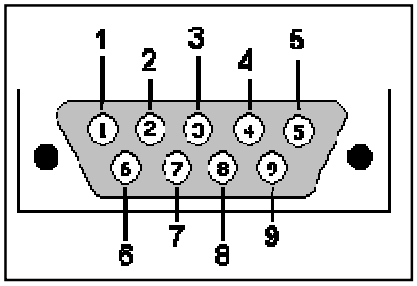
Software SORSYBAT také obsahuje simulační model soustavy Batyskaf, který se dá využít pro všechny metody regulace, jako testovací prostředek. Díky rychlejšímu průběhu simulované soustavy umožňuje získat přehled o chování sytému v kratším čase, než pozorováním na reálné soustavě. Autor programu získal model pomocí matematicko-fyzikální analýzy, který zpřesnil porovnáváním se skutečnou soustavou.



Obr. 5 - Schéma zapojení simulačního modelu v prostředí Simulink [3]

2.3. Současné řešení komunikace úlohy s PC

Zařízení je osazeno dvěma konektory D-Sub DE9 female, na kterých Batyskaf komunikuje pomocí standardu RS232C. Využity jsou pouze signály RxD (pin 2), TxD (pin 3) a Signal GND (pin 5), viz Obr. 6. Pro připojení úlohy k počítači je nutné, aby PC mělo dva sériové porty. Současné počítače, bez rozšiřujících karet, mají pouze jeden nebo žádný COM port. Proto lze využít konvertorů RS232 – USB pro připojení k libovolnému modernímu počítači.



Obr. 1 - Konektor CANON 9 pin

Asynchronní standard RS232 vyžaduje pro správnou funkčnost nastavení shodných parametrů komunikace na přijímači i vysílači. Batyskaf pracuje s přenosovou rychlostí 600 bd, která sice není vysoká, ale vzhledem k rychlosti změn v Batyskafu dostačující. Dále pracuje s délkou znaku 8 bitů, bez paritního bitu, s jedním STOP bitem a velikostí vyrovnávací paměti 1024 B.

Batyskaf využívá jednu sériovou linku pro ovládání polohy řídícího serva a přenosu informace o poloze plováčku. Druhá linka ovládá servo simulující poruchu a přenáší informace o tlaku nad hladinou. Číselné informace v obou směrech jsou přenášeny ve formě ASCII znaků a jako oddělovací znak je použita hvězdička „*“. Řídící jednotka vyžaduje pro zahájení komunikace, aby byl na obě linky poslán znak „S“ a naopak pro ukončení komunikace znak „F“. Pokud se komunikace nesprávně ukončí (například pokud spadne ovládací program), začne řídící jednotka posílat na obě linky náhodné znaky a je třeba komunikaci znovu řádně spustit a ukončit. Další zajímavostí je, že pokud zahájíme komunikaci na jedné lince, tak ve skutečnosti se

spustí na druhé a naopak, toto v praxi nijak nevadí, protože se obvykle spouští komunikace na obou portech zároveň. [6]

Otevření ventilu je řízeno natočením serva, to lze regulovat od 0 do 6 otáček. To odpovídá 0 až 12*π radiánů, údaje je ale nutné posílat v rozsahu od 0 do 400, proto je zde potřebný přepočet. Poloha plováčku je přenášena v setinách milimetru, zde není žádný přepočet třeba. Tlak nad hladinou je přenášen v jednotkách převodníku, tj. v rozsahu 0 až 1023. Pro převod na tlak v Pascalech je nutná kalibrace.

V prostředí Simulink je použit pro komunikaci Real Time Toolbox od Humusoftu. Real Time Toolbox byl navržen hlavně pro použití se speciální měřící kartou, ale podporuje i sériové rozhraní, které je použito u této úlohy. Konkrétně jsou použity bloky „Serial In“ a „Serial Out“. Přenosový proces zde probíhá na pozadí, takže nezpomaluje Matlab a naopak výpočty v Matlabu neruší komunikaci. [7]

3. PROGRAMOVACÍ JAZYK PYTHON

Tato část práce slouží k získání přehledu o momentálních možnostech prostředí Python z komplexního pohledu, to znamená nejen základního jazyka, ale včetně možností rozšiřujících modulů. Výstupem je volba prostředí a modulů pro tvorbu ovládací aplikace pro Batyskaf. Kapitola 3 je výsledkem rešerše dostupných materiálů v anglickém jazyce s využitím zdrojů [1] až [39].

3.1. Proč Python

Tento programovací jazyk je dostatečně jednoduchý, aby se dal učit na základních školách a zároveň dost obsáhlý pro napsání objektové databáze (například ZODB) a podobných komplexních projektů. Dalo by se říci, že je flexibilní jako jazyk Smalltalk, přesto rozmanitý jako Perl a zároveň jednoduchý jako Basic. [8]

Použitelnost pro nováčky je pouze část z jeho schopností. Další a hlavní část tvoří možnost vytváření skutečných rozsáhlých aplikací, používaných pro náročné úkoly. Python umožňuje vytvářet struktury propracovaných abstrakcí stručně a ve srozumitelném formátu.

Proč, i přes všechny tyto výhody, je všechen skvělý nový software psán v C nebo Javě? Částečně proto, že lidé prostě jazyk Python neznají. I to je jeden z cílů mé práce, rozšířit povědomí o jazyce Python. Další z důvodů je, že tyto jazyky mají navrch v efektivitě, ale to se může změnit, pokud bude Python více rozšířen a zvýší se jeho podpora.

Aktuální situace není optimální. Na průměrném Linuxovém systému jsou programy napsané až v šesti různých jazycích. To způsobuje duplicity v kódu, nekompatibility dat a velké zpomalení procesu pochopení pochodů v programu.

Python vždy nabízí pro řešení konkrétního problému několik různých způsobů. Ostřílený programátor tuto možnost zajisté ocení a vybere si tu nejvhodnější možnost pro jeho rozsáhlý projekt, ale pro obyčejného člověka, který pouze potřebuje rychle napsat malý zkušební program, je toto značně zatěžující. Muset se každou chvíli zastavovat a rozhodovat jakou cestou jít dál, může velice snížit produktivitu. Nebylo by proto lepší používat jazyk s jasně danými hranicemi jako například Matlab? Dovolím si použít přirovnání, že to je jako učit se lyžovat nebo jezdit na snowboardu. První den na snowboardu je vždy utrpení plné pádů, zatímco lyžař už jezdí po okolí. O několik týdnů později, ale člověk na snowboardu nacvičuje obdivuhodné kousky a lyžař stále jen jezdí. Možná trochu přehnané přirovnání, ale odpovídá tomu, že pokud člověk vloží více úsilí do učení Pythonu, není problém v něm

vytvořit komplexní projekty.

3.2. Historie a vývoj

Jazyk Python nevznikl jako většina jiných jazyků, kdy velká společnost sestavila tým lidí s jediným cílem. Základy položil pouze jeden člověk, jménem Guido van Rossum. Po vysoké škole Guido pracoval na projektu Amoeba (distribuovaný systém), kde také vznikla motivace a potřeba vyššího jazyka. Uvědomil si, že vývoj administračních programů v C trvá příliš dlouho a navíc se objevovaly velké problémy s propojením s jádrem operačního systému. Bylo potřeba najít jazyk, který by vytvořil „most“ mezi jádrem a C. To se později stalo na dlouhou dobu hlavním cílem Pythonu. [9]

Nabízí se otázka, proč pro tyto potřeby neupravil již existující jazyk. Se zkušenostmi s množstvím jiných jazyků, které by vyžadovaly neúměrné množství energie na úpravy, se rozhodl, že nejrychlejší bude vytvořit jazyk vlastní. Využil všech svých zkušeností z předchozích projektů, snažil se vyvarovat všemu, co mu na ostatních jazycích vadilo a výsledkem je Python.

Prvním problémem bylo jméno. Python znamená v překladu krajta. Jak by se mohlo na první pohled zdát, Guido by mohl být milovníkem hadů. Ve skutečnosti pouze nechtěl příliš dramatizovat výběr jména a vybral první věc, která ho napadla a to byl Monty Python’s Flying Circus. Je to totiž jeho oblíbená komediální skupina. Slovo „Python“ je dobře zapamatovatelné a spadá do tradice pojmenovávání programovacích jazyků po slavných osobnostech. Například Pascal, Ada nebo Eiffel. Guido dokonce několik let bojoval proti pokusům o spojování jazyka s hady. Nakonec se vzdal, když nakladatelství O’Reilly chtělo vydat knihu "Programming Python" a na obálku dát hada. [10]

Obr. 2 - Aktuální logo Pythonu [12]

Samotné práce na jazyku začaly v prosinci roku 1989 a během prvních měsíců existovala první fungující verze. V raných fázích vývoje používali Python převážně lidé z firmy CWI, kde v té době pracoval. Také klíčoví vývojáři Sjoerd Mullender a Jack Jansen pocházeli ze stejné firmy. První veřejně vydaná verze 0.9.0 v únoru 1991 spatřila světlo světa v diskusní skupině *alt.sources*. Vydání proběhlo pod licencí téměř doslovné kopie MIT licence, která zaručovala v podstatě podmínky open-source ještě dříve než tento termín vůbec existoval. Okamžitě po vydání dostal Guido velké množství ohlasů, díky kterým pokračoval několik dalších let ve vývoji.

Python vznikl hlavně pod vlivem jazyka ABC, který měl sloužit jako náhrada za Basic pro výuku programování. Typická vlastnost – odsazování, pochází přímo z ABC, ale původní myšlenku propagoval už Donald Knuth. Odsazování v podstatě znamená, že záleží, kolik mezer se nachází před každým řádkem, dokonce udává celou strukturu programu. Tato vlastnost často překvapí, až odradí, programátory, kteří mají zkušenosti z jiných programovacích jazyků. Pro mě je tato vlastnost jednoznačně přínosem, protože velice zpřehledňuje kód. V prvních verzích se nepoužívala dvojtečka před následujícím odsazeným kódem, ale při testech bylo zjištěno, že začátečníkům potom nebyl jasný význam a důvod odsazené části. Proto se nyní používá dvojtečka, vždy když následuje odsazený blok.

3.2.1. Základní filozofie návrhu Pythonu

Protože vývoj začínal pouze jeden člověk, bez žádného oficiálního rozpočtu, Guido si určil několik pravidel pro úsporu času: [11]

- Vypůjčovat si nápady kde se dá.
- „Věci by měly být jednoduché jak je jen možné, ale ne jednodušší.“ (Einstein)
- Dělej pouze jednu věc, ale dobře. (filozofie UNIXu)
- Nezabývej se příliš výkonem, optimalizuj později, až to bude třeba.
- Nebojuj s prostředím a jdi s proudem.
- Nesnaž se o dokonalost, protože „ujde to“ je často dokonalé.
- Je v pořádku něco „odfláknout“, obzvlášť pokud se k tomu jde později vrátit.

Několik dalších pravidel, které nebyly určeny přímo pro úsporu času. Některé spíše způsobovaly opak:

- Implementace Pythonu by neměla být závislá na platformě.
- Neobtěžuj uživatele s detaily, které může obsloužit počítač.
- Podporuj platformě nezávislý kód, ale využívej všechny možnosti konkrétní platformy. (velký rozdíl oproti Javě)
- Velký složitý systém by měl mít několik úrovní rozšiřitelnosti.
- Chyby by neměly způsobovat zbytečné ukončování programu.
- Ale zároveň by chyby neměly projít bez povšimnutí.
- Chyba v kódu uživatele by neměla vést k neočekávanému chování překladače. Pád překladače není nikdy chyba uživatele.

Filozofie, kterou Guido použil při návrhu Pythonu, je pravděpodobně jedním z hlavních důvodů jeho velkého úspěchu. První uživatelé zjistili, že Python je pro jejich účely dostačující, místo toho, aby toužili po dokonalosti. S tím jak se zvětšovala uživatelská základna, návrhy na zlepšení byly postupně zapracovávány do jazyka.

Několik těchto vylepšení dokonce znamenalo podstatné změny a přepracování některých částí jádra jazyka. Python pokračuje ve vývoji i dnes a stále se rozšiřuje.

3.3. Používané verze

V době psaní této bakalářské práce jsou vývojáři Pythonu udržované a aktualizované tři větve. Konkrétně se jedná o verze 2.6, 2.7 a 3.2. K tomuto rozdělení došlo již v počátku vývoje verze 3, kde bylo naplánováno tolik změn, že se autoři rozhodli, kvůli zpětné kompatibilitě větve rozdělit. Ve větvi 3 totiž nemusí fungovat všechny programy psané pro starší verze, zatímco u 2.6 a 2.7 by mělo stačit minimum nebo žádné úpravy.

Větev 3, také známá jako „Python 3000“ nebo „Python 3K“ je první vůbec záměrně zpětně nekompatibilní vydání. Obsahuje více změn než běžné vydání, ale tyto změny jsou důležité pro všechny uživatele. Ve výsledku se Python moc nezměnil, ale bylo odstraněno mnoho starého nepoužívaného kódu a opraveny neduhy jazyka jako takového. [13]

3.3.1. Problémové změny v nové verzi Pythonu 3

Zde nastíním jen několik málo nejdůležitějších změn provedených ve verzi 3.

Výraz *print* se změnil na funkci *print()*. V praxi to znamená většinou pouze přidání závorek, ale přidává i nové možnosti, jako například volba oddělovače.

```
Dříve:  print 'x y'              # vypíše řetězec 'x y'
Nyní:   print('x','y' sep=" ")  # vypíše řetězec 'x y'
```

Zjednodušení pravidel pro příkazové porovnávání. Operátory *(<, <=, >=, >)* vyvolají *TypeError* pokud výraz nemá smysl. Nelze tedy použít výrazy jako *1 < ''*, *0 > None* nebo *len <= len*. To také znamená, že není možné třídit seznam s neporovnatelnými položkami. Toto ovšem neplatí pro operátory *==* a *!=*, neporovnatelné objekty si vždy nejsou rovny.

Změna názvu integeru *long* na *int*. Dále například *1/2* vrací typ float, když ve starších verzích to značilo celočíselné dělení. Pro celočíselné dělení se nyní používá *1//2*. Čísla osmičkové soustavy se nyní zapisují ve formě *0o720* místo *0720*.

Ve starších verzích bylo nutné řetězce obsahující speciální znaky (diakritiku) značit na počátku znakem u, například *u"…"*, nyní jsou všechny řetězce automaticky v unicode a 8 bitové řetězce lze použít na vyžádání pomocí *b"…"*. Z této změny vyplývá celá řada dalších změn, které ovlivňují téměř všechny programy. Nyní jsou jasně odděleny datové typy pro řetězce a binární data a proto je nutné používat pro převod mezi nimi metody *str.encode()* a *bytes.decode()*. Pro soubory otevřené pomocí funkce open() jako textové metody je nutné nastavit správně mód, ve kterém

má soubor otevřít. Jestli bude obsahovat řetězce nebo binární data. Také došlo ke změně základního kódování kódu na UTF-8.

3.3.2. Python 2.6

Vývojový cyklus verzí 2.6 a 3.0 byl synchronizovaný, alfa i beta verze byly vydány ve stejný den. To znamená, že funkce, které jsou navrženy pro verzi 3.0, byly zpětně převedeny do verze 2.6 [15]. Výčet několika vybraných prvků:

- Výraz *with* se už nemusí importovat, pokud ho chceme použít
- Modul multiprocessing, vzniknul jako kopie threading modulu, ale místo vláken používá procesy
- Formátování řetězců pomocí metody *format()*
- *print()* jako funkce
- metoda __complex__() pro převod objektů na komplexní čísla
- alternativní syntaxe zachytávání výjimek: except TypeError as exc

3.3.3. Python 2.7

Verze 2.7 je poslední plánovaná verze ve vývojové větvi 2.x, proto obsahuje nejvíce nových prvků z větve 3. Díky tomu se předpokládá, že verze 2.7 bude udržována mnohem delší dobu než předchozí verze. Stejně jako verze 2.6 přebírá vlastnosti z 3.0, tak 2.7 přebírá prvky z 3.1 [14]. Několik prvků z verze 3.1:

- nová verze knihovny io, přepracovaná do jazyka C pro lepší výkon
- nový datový typ – řazený slovník (ordered-dictionary)
- zaveden oddělovač řádu tisíců pro lepší přehlednost
- převod z float na řetězec se nyní správně zaokrouhluje

3.3.4. Použít verzi 2.x nebo 3.x ?

Volba verze je hlavně závislá na našich potřebách, nedá se jednoznačně říci, že jedna je lepší. Pokud vše co potřebujeme je dostupné pro 3.x a máme k dispozici prostředí s verzí 3.x, potom je volba jasná. Často se ale může stát, že není k dispozici knihovna nebo modul pro novější Python a jsme nuceni použít starší verzi. Také můžeme být závislí na prostředí, kde bude program používán, například webový server, který stále používá starší verzi. Pořád ještě existuje řada hojně používaných modulů, které verzi 3.x stále nepodporují. Například modul Twisted používaný pro produkční servery, je velký balíček zakořeněný hluboko do Pythonu a jeho přepsání pro novější verzi není vůbec jednoduché. Na druhou stranu několik hlavních knihoven už je přepsáno pro Python 3. [16]

Pro účely této práce jsem se rozhodl použít verzi 2.6 a to hlavně z důvodu široké uživatelské podpory a velkého množství použitelných knihoven. Některé klíčové

moduly, které používám nebo jsem testoval při návrhu, bohužel novou verzi Pythonu nepodporovaly.

3.4. Implementace a multiplatformnost

Možnosti různých implementací a chodu na rozličných platformách jsou jedny z nejsilnějších stránek Pythonu.

Implementace se dá také přeložit jako realizace nebo zavedení a ve skutečnosti se ani o nic jiného nejedná. Přesněji by se tento výraz dal popsat jako prostředí nebo program, který poskytuje podporu pro spouštění programů napsaných v jazyce Python. Nejpoužívanější a zároveň referenční implementace je CPython. Je napsána v programovacím jazyku C a je považována za vzor pro ostatní implementace. Nikoho asi nepřekvapí, že autor této realizace je shodný s tvůrcem Pythonu - Guido van Rossum. Existuje celá řada dalších implementací, nebudu zde vyjmenovávat všechny, zaměřím se pouze na několik nejpoužívanějších. Problémem neoficiálních implementací je, že jsou často nedokončené, nepodporují všechny prvky jazyka, nebo jsou sice kompletní, ale již několik let neaktualizované. V době psaní této práce jsou aktuální pouze čtyři implementace. První z nich je IronPython, je to open-source implementace, která je úzce svázána s prostředím .NET. V tomto případě je Python považován za silný doplněk k .NET Frameworku používaného hlavně pro webové aplikace. Jython je další implementací, tentokrát realizované v jazyce Java. Výhodou Jythonu je, že programy v něm napsané je možné spustit kdekoliv, kde je Java, tedy v dnešní době téměř kdekoliv. Další realizací Pythonu je PyPy. Její hlavní výhodou je rychlost programů v ní spouštěných. Používá technologii Just-In-Time kompilování, která ve většině případů umožní rychlejší běh programů s menšími nároky na paměť. Poslední implementací, kterou popíši, je Stackless Python, který se používá převážně díky velice propracované podpoře mikrovláken a práce s nimi. [17][18]

Pojem multiplatformnost by se dal popsat jako podpora mnoha různých platforem – operačních systémů. Toto je velice silná stránka Pythonu, protože je možné spustit stejný kód na běžném počítači s libovolným operačním systémem, stejně jako například na mobilním telefonu nebo herní konzoli. Pravidelně vydávané verze podporují tři základní operační systémy, Windows, Mac OS a Unixové systémy. Různé modifikace jazyka, je ale také možné spouštět například na iPodu, pod dříve hojně používaným systémem MS-DOS, na Paml OS používaném pro PDA, na herních konzolích PlayStation nebo Xbox, na platformě Series 60 založené na Symbianu, která se používá převážně v mobilních telefonech Nokia a na nespočtu dalších. Pro lepší podporu různých systémů existuje v Pythonu knihovna *platform*, která umožňuje

získat maximum informací o operačním systému, na kterém je zrovna program spuštěn.

3.5. Komunikační možnosti

Zde platí, že komunikační možnosti Pythonu jsou tak široké, jak široké jsou možnosti zařízení, na kterém je spuštěn. Proto uvedu výčet nejpoužívanějších komunikačních rozhraní a knihoven pro Python.

3.5.1. Sériové rozhraní

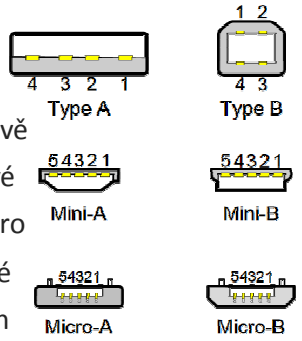
Sériová linka se dříve používala převážně pro tiskárny, počítačové terminály, kamery, modemy apod. V dnešní době se používá jako spolehlivé rozhraní pro přenos dat na krátké vzdálenosti mezi libovolnými zařízeními.

Pro Python existuje knihovna pySerial, která umožňuje přístup k sériovému portu. Podporuje většinu nejpoužívanějších operačních systémů a implementací Pythonu. Umožňuje nastavení různých délek datového bytu, stop bity i paritní bity. Možnost práce s nebo bez čekání na příchozí data. Přístup k sériovému portu je podobný práci se soubory, pomocí instrukcí „*read*“ a „*write*“, také podporuje přečtení pouze jedné „řádky“ pomocí „*readline*“. Tato knihovna je napsána pouze v čistém Pythonu, bez podpory dalších jazyků. [20]

3.5.2. Rozhraní USB

USB sběrnice je moderní a velice rozšířený způsob připojení periferií k počítači. V dnešní době se používá místo starších rozhraní pro připojování periferií, jako například sériový a paralelní port, gameport apod. Díky své jednoduchosti se pole působnosti USB stále rozšiřuje na více různých zařízení. Existují čipy o velikosti komunikačního portu USB, které se starají o celou komunikaci, takže USB lze využít i pro účely měřících a laboratorních zařízení.

Modul PyUSB se zaměřuje na jednoduchý přístup k USB portu v Pythonu. Vývoj této knihovny je nyní rozdělen na dvě větve. Stabilní verze 0.x a vývojová 1.x, ale obsahující nové funkce. Obsahuje jednoduché aplikační rozhraní pro komunikaci se zařízeními. Podporuje vlastní koncové knihovny a izochronní přenos. Opět je celý modul napsán pouze v čistém Pythonu. [22]



Obr. 3 - Typy USB konektorů [21]

3.5.3. Paralelní rozhraní

Paralelní port byl od svého vzniku určen pro připojení tiskáren, v dnešní době už je téměř úplně vytlačen USB portem. Používá velký konektor DB-25. Komunikace

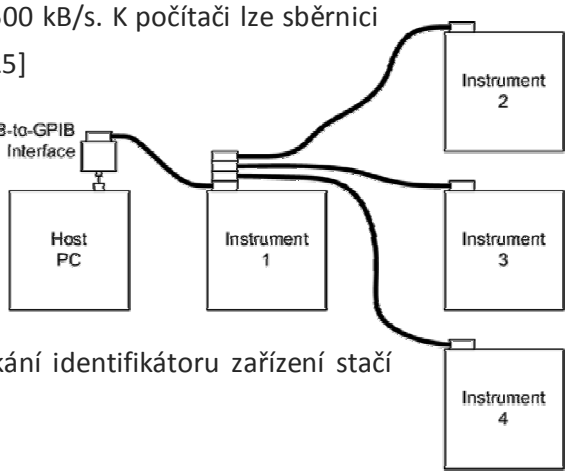
probíhá na 8 bitové paralelní datové sběrnici, dále obsahuje 4 vodiče pro ovládání výstupu a 5 vodičů pro ovládání vstupu.

Modul pyParallel byl vyvíjen společně s modulem pySerial, ale zřejmě kvůli malému rozšíření paralelního portu nebyla již několik let vydána nová verze. Poslední verze je 0.2 z roku 2005. Stejně jako pySerial i tento modul je možné používat pod různými operačními systémy a v různých implementacích. Číslování portů začíná od nuly, a proto není nutné znát přesný název portu. Pokud není vhodné číslování portů lze použít i název zařízení. [23]

3.5.4. Sběrnice GPIB

GPIB (General Purpose Interface Bus) neboli také sběrnice pro obecné účely je rozhraní vyvinuté společností Hewlett-Packard pro měřící a zkušební zařízení, umožňující přenos dat mezi dvěma nebo více přístroji. Paralelní sběrnice umožňuje připojení maximálně 15 přístrojů, přenosová rychlost je závislá na délce kabelu, v praxi se dosahuje rychlostí 200 až 500 kB/s. K počítači lze sběrnici připojit pomocí převodníku do USB. [25]

Balíček PyVISA umožňuje řízení různých druhů měřících zařízení nejen pomocí sběrnice GPIB, ale i pomocí sériového nebo USB portu. Používání knihoven je velice jednoduché, například pro získání identifikátoru zařízení stačí pouze tři krátké řádky kódu. [24]



Obr. 4 - Zapojení zařízení do hvězdy pomocí GPIB [19]

3.5.5. TCP / UDP protokol

Protokoly TCP a UDP jsou součástí internetového protokolu IP. Použití těchto protokolů pro řízení a sledování zařízení na velké vzdálenosti je v poslední době na vzestupu. K internetu se už dá připojit prakticky odkudkoliv na světě a to je vše, co je nutné k připojení se k zařízení. Protokol TCP používá připojení, které je nutné otevřít a zavřít. Výhodou tohoto modelu je možnost kontroly bezchybnosti dat a možnost požádání o chybějící části dat. Naopak protokol UDP nevytváří žádné připojení, ale pouze odešle takzvaný datagram na určenou adresu. Díky tomu méně zatěžuje síť, ale je zde horší kontrola dat a problémy s pořadím příchozích paketů.

V Pythonu je již zabudovaný modul socket, který umožňuje nejen komunikaci přes protokoly TCP nebo UDP. Modul socket je možné používat na všech systémech podporujících Berkeley sockets, což zahrnuje všechny běžně používané operační systémy. Berkeley sockets jsou aplikační rozhraní pro internetový protokol. Použí-

vání je, jako tradičně v Pythonu, velice jednoduché. Jednoduchý TCP klient nebo i server je možné napsat asi na 15 řádků kódu. [26]

3.5.6. Bluetooth

Bezdrátová technologie Bluetooth umožňuje propojení dvou zařízení na kratší vzdálenosti. Nejnovější revize specifikace Bluetooth dokáže dosáhnout až 24 Mbit/s. Pracuje v rádiovém rozhraní 2,4 GHz a pro přenos dat je použita technologie přenosu v rozprostřeném spektru. Tato technologie je vhodná pro ovládání a kontrolu zařízení bezdrátovým přenosem dat.

Pokud chceme použít Bluetooth v Pythonu, je nutné použít například knihovnu PyBluez. Tento rozšiřující modul pro Python je napsán v jazyku C. Poskytuje přístup k systémovým prostředkům Bluetooth. Primárně byl napsán pro Windows XP a Linux. Jeho používání je opět velmi jednoduché. Připojit se k zařízení lze na několika řádcích kódu. [27]

3.5.7. Karty pro získávání dat

Abychom mohli nějaký proces ovládat, nejprve potřebujeme znát potřebné vstupní veličiny a to nejlépe s dostatečně velkou vzorkovací frekvencí a mít možnost s daty pohodlně a rychle pracovat. Tyto požadavky splňují různé karty pro získávání dat, které se buď připojují například přes USB, nebo se vkládají přímo do počítače například do sběrnice PCI.

Pro ukázku jsem vybral kartu DaqBoard/1000 od firmy Measurement Computing. Tato karta se připojuje pomocí rozhraní PCI. Obsahuje 16 bitový A/D převodník pracující na frekvenci 200 kHz. Možnost až 16 analogových vstupů, 4 čítače, 2 časovače. V Pythonu lze tuto kartu používat s pomocí knihovny pydaqboard, která je napsaná v jazyce C pro operační systém Windows. [28][29]

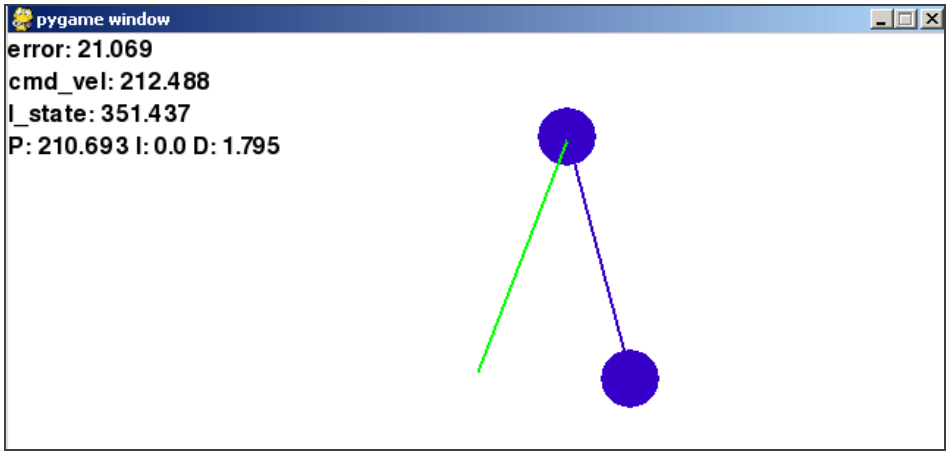
Obr. 10 - Karta pro získávání dat LabJack U3-HV [30]

Dalším příkladem může být karta U3 (Obr. 10) od firmy LabJack. Připojit ji lze přes USB, umožňuje použití až 16 vstupů, 2 čítačů a 2 časovačů. Přímo firma LabJack nabízí modul LabJackPython [31], který lze použít pro jakékoliv zařízení od této firmy.

Knihovnu lze používat na všech běžných operačních systémech. Připojení a získávání dat je opět více než jednoduché, většinou si vystačíme s třemi řádky kódu.

3.6. Řízení a regulace

V Pythonu lze s jistou dávkou kreativity napsat libovolný řídící program nebo regulační algoritmus bez speciálních podpůrných modulů. Jako příklad lze uvést práci Eda Pradadise [32], který využil modul PyODE (3D grafická fyzikální simulace) k napsání jednoduchého kyvadla, které následně řídil PID regulátorem napsaným v Pythonu (Obr. 11).



Obr. 11 – Simulace kyvadla řízená PID regulátorem napsaným v Pythonu [32]

Existují ovšem i moduly, které nám velice usnadní, ať už vytváření modelů soustavy nebo regulátorů. Jako příklad lze uvést knihovnu python-control [34], která přináší funkce z toolboxu *Řídicí systémy* v MATLABu. Tento balíček nabízí základní funkce pro analýzu a návrh zpětnovazebních systémů a byl vytvořen jako doplněk k učebnici *Zpětnovazební systémy* od autorů Karl J. Åström a Richard M. Murray. Výčet několika funkcí, které tato knihovna nabízí:

- Lineární systémy ve stavovém prostoru a frekvenční oblasti
- Výpočty na základně blokových schémat
- Časová odezva: počáteční, skoková, impulzní
- Frekvenční odezva: Bodeho a Nyquistova charakteristika
- Analýza modelu: stabilita, dostupnost, meze stability
- Návrh modelu: hledání vlastních čísel

Jedním z dalších modulů je pypid [33], který se dá považovat za plnohodnotnou náhradu klasického PID regulátoru. Autorův záměr je, že uživatel nepotřebuje znát vnitřní funkce regulátoru, pouze stačí nastavit parametry. Modul podporuje komunikaci pomocí protokolu Modbus přes sériovou linku a automatické nastavování parametrů regulátoru. Je napsaný v čistém Pythonu, orientovaný převážně na Linux.

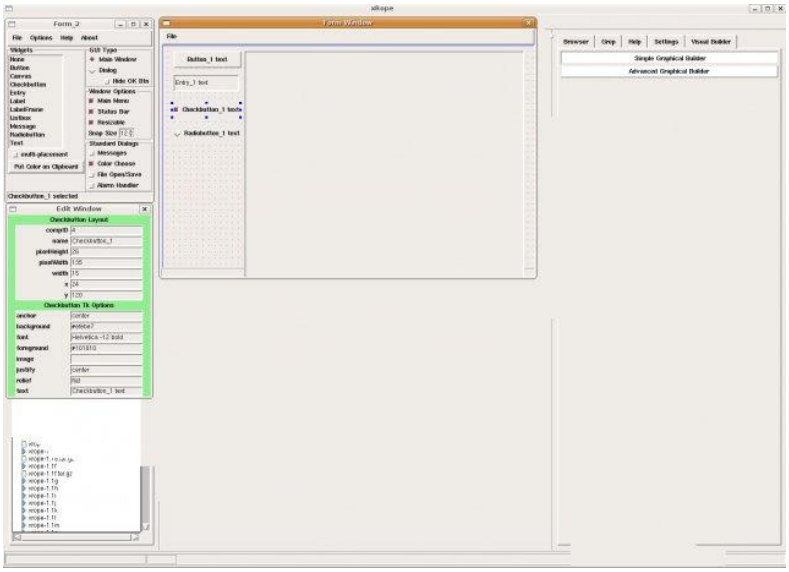
Jak je z předchozího textu znát, Python nachází zatím využití spíše v oblasti simulace řízení, než řízení samotného, ale i to se poslední dobou mění.

3.7. Grafické uživatelské rozhraní

Žádná moderní aplikace by nebyla kompletní bez grafického uživatelského rozhraní, dále jen GUI (Graphical User Interface) a to nejen kvůli pohodlnosti používání, ale také díky rozsáhlým vizuálním možnostem. Pro Python existuje velké množství [35] frameworků[1], které se liší svým koncovým zaměřením nebo podporou platforem. Zde se zaměřím převážně na multiplatformní frameworky, protože to je i jeden z hlavních kladů Pythonu samotného.

Nejčastěji se lze setkat s frameworkem **Tkinter**, je totiž dodáván jako součást základní instalace Pythonu. Tkinter napsal Fredrik Lundh a jeho název se skládá ze slov „Tk interface“. Z toho je zřejmé, že Tkinter vychází z open-source frameworku Tk. Tk je platformně nezávislý framework, který byl vyvinut pro programovací jazyk Tcl.

Pro Tkinter existuje vývojové prostředí nazvané xRope (Obr. 12). Neslouží pouze pro návrh GUI, ale je to plnohodnotné prostředí pro vývoj celých aplikací. xRope je typický program typu Python pro Python. Je totiž napsán v čistém Pythonu a jeho GUI je vytvořen pomocí Tkinteru. Je plně multiplatformní, rychlý a nenáročný na systémové prostředky.



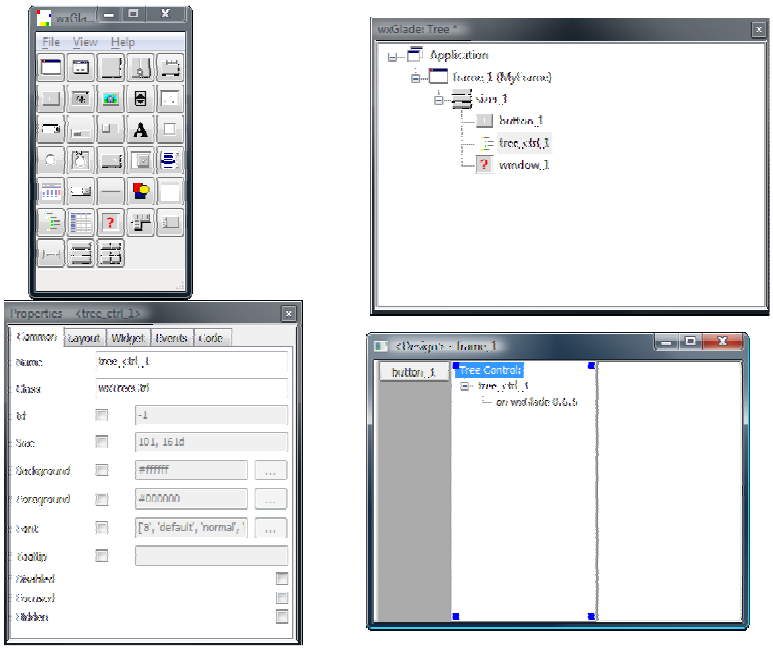
Obr. 12 – Návrh GUI v xRope [36]

Velice oblíbený je taktéž GUI framework **wxPython**. Pro vykreslování GUI používá knihovny wxWidgets napsané v C++, které jsou taktéž multiplatformní. wxPython je,

[1] Framework je softwarová struktura, která slouží jako podpora při programování, vývoji a organizaci jiných softwarových projektů.

stejně jako Python, open-sourceový. Robin Dunn napsal první verzi wxPythonu, když potřeboval během několika týdnů vytvořit GUI pro program ve Windows 3.1 a Linux.

Z velkého výběru vývojových prostředí pro wxPython, mě nejvíce zaujal wxGlade. Je to jednoduchý, malý nástroj, který slouží pouze k návrhu GUI. I přes jeho jednoduchost dovoluje vygenerování kódu nejen v Pythonu, ale i v C++, Perlu, Lispu a XRC. wxGlade jde opět příkladem, je napsán v Pythonu. Tento nástroj jsem si vybral pro návrh GUI ovládacího programu pro Batyskaf.



Obr. 13 – Návrh GUI ve wxGladu.

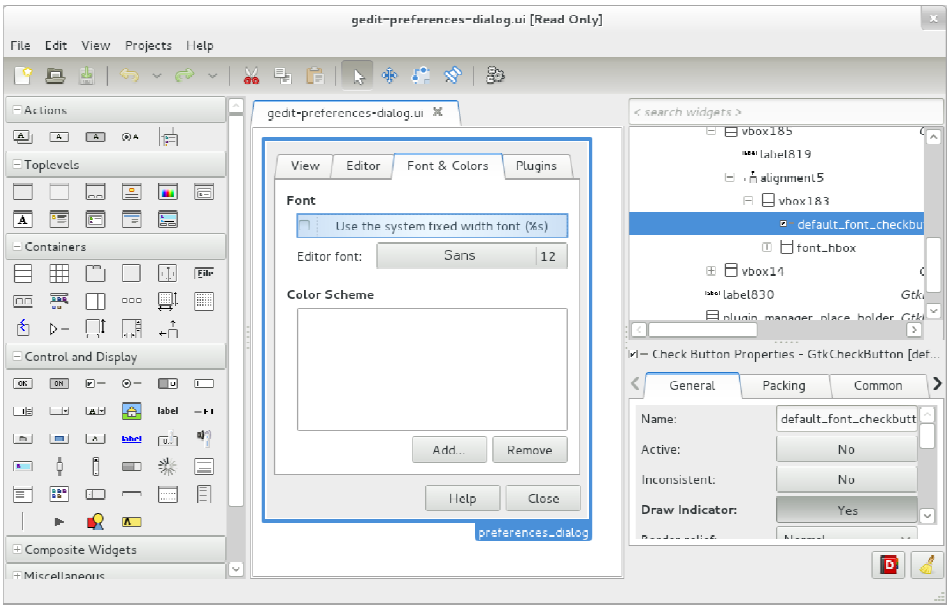
Dalším oblíbeným frameworkem je **PyQt** vyvinutý společností Riverbank Computing. PyQt je v podstatě sada vazeb pro Qt framework od společnosti Nokia zaměřený na C++. PyQt je stejně jako Qt plně multiplatformní. PyQt i Qt samotné jsou rozdělené do dvou větví, verze 3 a 4. PyQt obsahuje až 300 tříd a přes 6000 funkcí. PyQt je ke stažení zdarma, ale pouze pro soukromé účely. Komerční licence PyQt stojí 350 liber, dále by bylo nutné si pořídit i licencovanou kopii Qt samotného.

Plnohodnotné vývojové prostředí Eric napsané v Pythonu podporuje nejen Python, ale i Ruby. Je navrženo jak pro každodenní použití jako rychlý editor, tak ho lze stejně dobře používat pro velké profesionální projekty. Vývojové prostředí Eric je pojmenováno po členu skupiny Monty Python Ericu Idleovi.

Posledním z nejznámějších frameworků je **PyGTK**, který je dostupný zdarma. Autor PyGTK je zároveň vývojářem GNOME grafického prostředí pro Linux. To znamená, že PyGTK nabízí nejširší možnosti integrace na Linuxu. Jak název napovídá, PyGTK je založeno na frameworku GTK+, který je napsaný v jazyce C. Pro GTK+ exis-

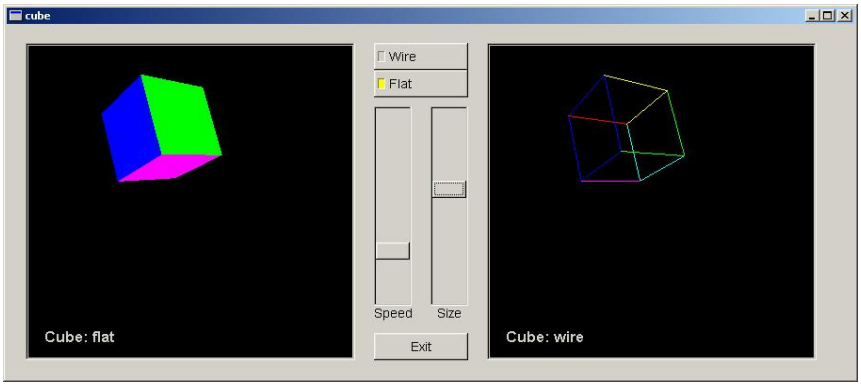
tuje celá řada aplikačních rozhraní pro všechny více i méně používané programovací jazyky.

Pro návrh a vývoj GUI lze použít známý nástroj Glade (Obr. 14), který je ke stažení zdarma. Starší verze Gladu byly ke stažení i pro Windows, nyní lze stáhnout pouze zdrojové kódy pro Linux. Glade se řadí do kategorie RAD (Rapid application development) nástrojů, která by měla zaručit maximálně kvalitní výsledky při minimalizaci nutného plánování a času. Výstupem z Gladu jsou XML soubory, které je nutné následně pomocí GtkBuilderu sestavit. Díky tomuto postupu lze výsledné GUI sestavit téměř v libovolném programovacím jazyce podporovaném GtkBuilderem.



Obr. 14 – Návrh GUI pomocí nástroje Glade [37]

Jedním z méně známých a rozšířených frameworků je **pyFLTK** (Obr. 15), jehož hlavní cíle jsou rychlost, nenáročnost, multiplatformnost a jednoduchost použití. Z názvu je opět jasné, že je založen na sadě FLTK (Fast Light Tool Kit) napsané v C++ pro všechny hlavní platformy. Vazby pyFLTK a FLTK jsou vytvořeny pomocí nástroje SWIG, který usnadňuje vytváření rozhraní mezi programy napsanými v C/C++ s vyššími jazyky jako Python, Perl, PHP a Ruby.



Obr. 15 – Příklad použití pyFLTK s OpenGL grafikou [38]

Na konec pojednání o grafických uživatelských rozhraních bych rád zmínil zajímavou alternativu **Kivy** (Obr. 16). Kivy je open-source projekt vydaný pod licencí LGPL3, která zaručuje možnost vytváření a prodej aplikací zdarma, ale kvůli slabé dokumentaci a malé uživatelské komunitě není příliš rozšířen mezi obyčejnými uživateli. Autoři projektu se více zaměřují na podporu komerčních projektů. Silná stránka Kivy spočívá v podpoře více-dotykového ovládání (Obr. 16), to žádný jiný z Python frameworků zatím neumožňuje. Tento model ovládání lze využít nejen pouze pro jednoho uživatele s více prsty, ale i pro více různých uživatelů.

Obr. 16 – Grafické uživatelské rozhraní Kivy na více-dotykové obrazovce [39]

Díky silné podpoře dotykových obrazovek je Kivy převážně rozšířeno na mobilních platformách jako například Android nebo IOS. Podporuje většinu používaných vstupních protokolů a zařízení. Grafické rozhraní je postaveno na technologii OpenGL, která zaručuje vysoký výkon i při náročných grafických projektech. Kivy využívá jednoduchosti a rychlosti programování v Pythonu, ale kritické části programů bývají překompilovány do jazyka C, který zaručí nejlepší optimalizaci kódu. Jako další příklad použití Kivy lze uvést hru piškvorky (Obr. 17), která je sestavena z pneumatických prvků Festo, řízena PLC Siemens S7-300 pomocí sítě ethernet a ovládání je napsáno pomocí Kivy s využitím 24 palcové více-dotykové obrazovky.

Obr. 17 – Piškvorky řízené PLC Siemens S7-300 [39]

4. REALIZACE OVLÁDACÍ APLIKACE PRO "BATYSKAF"

Závěrečná část práce obsahuje samotný návrh a realizaci ovládací aplikace pro úlohu Batyskaf. Dále popisuje možnosti využití již hotového programu a návod k jeho používání.

4.1. Struktura ovládacího programu pro Batyskaf

Nejdříve stručně popíši strukturu ovládacího programu, pro základní představu funkce.

Celý program je rozdělen do tří vláken. První vlákno zajišťuje čtení a zápis dat pomocí sériové linky a druhé vlákno zpracování dat. Třetí, hlavní vlákno obsahuje grafické uživatelské rozhraní s vykreslováním grafů.

Z Batyskafu přichází data ve formě „nekonečného“ řetězce, například *„50*51*50*53*49*“* atd. Prvního vlákno *com_monitor* tento řetězec zachytává a „rozsekává“ na jednotlivá čísla. V momentě, kdy se podaří získat celé číslo, tak je mu zároveň přiřazena časová značka. Tato dvojice je pomocí fronty poslána do druhého vlákna.

Druhé vlákno pracuje nezávisle na prvním a jeho vstupním parametrem je hodnota nastavené vzorkovací frekvence. Jeho cílem je, aby se vnitřní cyklus vlákna spouštěl v co nejpřesnějších časových intervalech podle nastavené vzorkovací frekvence. Slouží, ke zpřesnění vzorkování, které je u přicházejících čísel velice proměnlivé. Pokud je nastavené vzorkování menší než vzorkovací frekvence Batyskafu, tak se z příchozích hodnot počítá průměr, naopak pokud je vyšší, tak je hodnot nedostatek a vždy když chybí hodnota, tak se použije hodnota z předchozího cyklu. Tímto způsobem je zaručené vzorkování přesné v rámci možností operačního systému. Výstupem z tohoto vlákna je jakýsi zásobník řady čísel, a to: původní časová značka, nová přesná časová značka, poloha plováčku, natočení řídícího serva, natočení poruchového serva. Velikost tohoto zásobníku je dána nastavením „Délka dat“ v GUI.

Hlavní vlákno obsahuje pouze grafické uživatelské rozhraní a vykreslování grafů. GUI se skládá ze tří záložek. Na první záložce je samotné ovládání Batyskafu, lze zde spustit a zastavit ovládací proces. První dva posuvníky slouží k nastavení natočení řídícího serva a serva poruchy. Další dva k nastavení délky dat k uložení a délce zobrazení na grafu. Poslední slouží k nastavení frekvence pro převzorkovávání v druhém vlákně. Dále je zde možnost nastavení souborů pro ukládání dat. Grafy vykreslují průběhy vstupních a výstupních veličin včetně poruchy.

Druhá záložka slouží pouze pro vykreslení statické charakteristiky, kterou lze naměřit na první záložce.

Třetí záložka obsahuje grafy pro kontrolu vzorkování. V prvním grafu je vidět původní a nová vzorkovací frekvence, je zde také pěkně vidět jak moc je frekvence časově stálá. Druhý graf vykresluje linearitu vzorkování, strmost křivky odpovídá velikosti vzorkovací frekvence. Křivka by měla být vždy lineární, pokud není, tak ve vzorkovací frekvenci došlo k neočekávaným výkyvům.

4.2. Výběr modulů

Dříve než jsem začal psát vlastní program, bylo nutné si určit, které knihovny a moduly použiji pro jednotlivé části programu. Jak je zmíněno v rešerši výše, nejdůležitější části jsou komunikační modul a grafické uživatelské rozhraní.

Batyskaf komunikuje s počítačem pomocí sériové linky, proto bylo třeba najít modul rozšiřující Python o sériovou komunikaci. Zdaleka nejznámější a nejspolehlivější je knihovna pySerial, která poskytuje jednoduché a pohodlné rozhraní pro používání a ovládání sériové linky.

Abych měl představu o celkové koncepci programu, musel jsem si určit, který framework použiji pro grafické uživatelské rozhraní. Jako kritéria výběru jsem si určil nejen široké programovací možnosti a stabilitu, ale také rozsáhlou dokumentaci a širokou uživatelskou základnu, která mi pomůže najít odpověď na všechny mé dotazy. Tyto požadavky sice splňovaly tři nejrozšířenější frameworky wxPython, PyQt a PyGTK, ale dále jsem se také zajímal o dostupná vývojová prostředí, kde mě nejvíce přesvědčilo o své jednoduchosti a flexibilitě prostředí wxGlade. PyQt a PyGTK jsou také více zaměřeny na Linux, ale moje aplikace byla vyvíjena na Windows, kde by s těmito frameworky mohly být jisté komplikace. Proto jsem se rozhodl pro grafické uživatelské rozhraní použít knihoven wxPython s pomocí prostředí wxGlade.

Ovládání Batyskafu by nebylo možné, kdybychom neviděli průběh výstupních veličin. Proto je nutné zvolit modul pro vykreslování grafů. Nejrozšířenějším modulem je matplotlib, který nabízí široké možnosti vytváření grafů, ale možná i díky tomu je částečně těžkopádný. Bohužel žádný jiný modul s obdobně propracovanou dokumentací není, takže jsem neměl jinou možnost než použít tento. Další možností by bylo napsat si vlastní rychlé rozhraní pro vykreslování grafů, ale na to v rámci této práce nezbyl čas.

Již od začátku jsem předpokládal, že bude nutné program řešit ve více oddělených vláknech. V Pythonu je možné vytvářet vlákna a procesy, zvolil jsem si vlákna, podrobněji vysvětlím dále v textu. Nutnost více vláken je dána potřebou vykonávat

více věcí najednou. Například jedno vlákno se stará o přístup a čtení dat ze sériové linky, další zpracovává data a poslední se stará o zobrazení grafického uživatelského rozhraní. Python nabízí pro tvorbu vláken dva základní moduly, thread a threading. Modul threading je v podstatě vyšší nadstavba nad modulem thread, proto jsem zvolil ten.

Pro početní operace jsem zvolil modul numpy, který přináší výpočetní možnosti MATLABu do Pythonu.

4.3. Návrh řešení komunikace s úlohou

Nejdříve jsem na velice jednoduchém prográmku vyzkoušel funkci modulu pySerial. Skládal se pouze z otevření portu, nastavení parametrů, zahájení komunikace s batyskafem, přečtení dat a následné ukončení komunikace.

```
import serial

com1 = serial.Serial(10)  # otevření sériového portu COM11
com1.baudrate=600    # nastavení přenosové rychlosti
com1.timeout=0.5  #  doba čekání na příchozí data
com1.write("S") # zahájení komunikace s Batyskafem
for i in range(0,5):    # cyklus proběhne pětkrát
    com1r = com1.read(100)  #  přečtení maximálně 100 bytů
    print(com1r) # vypíše vše co bylo v zásobníku
com1.write("F") # ukončení komunikace s Batyskafem
com1.close()    # ukončení připojení k sériovému portu
```

Těchto několik řádků kódu stačilo, abych mohl získávat data z Batyskafu. Stačilo přidat jeden příkaz pro ovládání ventilů:

```
com1.write("200*") # nastavení polohy serva (otevření ventilu)
```

Takto řešený program by ale nikoho příliš nenadchl. Bylo by nutné vždy ručně přepisovat skript a vykreslovat data až po ukončení programu. Z toho vyplývá potřeba práce více různých částí programu najednou. Jedna část pro čtení dat a další pro vykreslování dat. Python nabízí několik možností, jak toto řešit.

V základní instalaci Pythonu se nacházejí moduly threading (vlákna) a multiprocessing (procesy), dále ještě existuje speciálně upravená verze Pythonu nazvaná Stackless Python, nabízející takzvaná mikrovlákna. Stackless Python jsem vyloučil hned z počátku, protože jeho zaměření je spíše na velké množství (stovky tisíc) mikrovláken. Například pro online servery s velkým množstvím uživatelů – pro každého uživatele jedno mikrovlákno.

Multiprocessing:

- Výhody: oddělené místo v paměti, srozumitelný kód, umí využít více jader CPU, je bez omezení GIL[2], není nutné řešit synchronizaci, procesy lze ukončit
- Nevýhody: komunikace mezi procesy je komplikovaná, vyšší nároky na paměť

Threading:

- Výhody: nenáročné na paměť, jednoduché sdílení dat mezi vlákny, vhodné pro vstupně/výstupní aplikace
- Nevýhody: omezení GIL, vlákna nelze ukončit, pokud se pro přenos dat nepoužívá fronta - nutné řešit synchronizaci, méně přehledný kód

Přestože mají vlákna více nevýhod, pro moje použití se hodí jednoznačně více. Vlákna se využívají právě pro případy, kdy je nutné oddělit grafické uživatelské rozhraní a komunikační část programu. Procesy se používají, když je opravdu nutné vykonávat více věcí najednou. Například pokud by bylo třeba ovládat více Batyskafů najednou, potom by každý Batyskaf měl svůj proces.

Inspiraci řešení komunikačního modelu jsem našel v článku Eli Benderskyho, „*A "live" data monitor with Python, PyQt and PySerial*“ [40]. V tomto článku je popsán jednoduchý program pro vykreslování teploty do grafu. Grafické uživatelské rozhraní je realizováno v PyQt a získávání dat pomocí sériové linky.

Jak už bylo řečeno, komunikaci na sériovém portu jsem řešil pomocí zvláštního vlákna, které se spustí po stisknutí tlačítka *Spustit*. Samotné vlákno se skládá ze tří hlavních částí:

__init__ - deklarace potřebných proměnných, nastavení počátečních hodnot

run – obsahuje samotný kód pro komunikaci

join – zastavení běhu vlákna

První část *__init__* je metoda povinná pro každou třídu v objektově orientovaném programovacím jazyce Python, funguje jako konstruktor[3]. V mém případě definuje vstupní parametry vlákna. První tři parametry jsou objekty typu Queue (fronta pro přenos dat mezi vlákny), další dva parametry obsahují číslo sériového portu, ke kterému se chceme připojit. Poslední parametry určují nastavení sériové komunikace – přenosovou rychlost, STOP bity, paritu a důležitý parametr timeout. Timeout určuje, jak dlouho bude program čekat při čtení dat, čím menší je nastaven, tím lepší je časové rozlišení dat, ale naopak stoupá hardwarová náročnost. Dále je v této části vlákna nastaven znak „*alive*“, podle kterého vlákno pozná, jestli je spuštěno nebo zastaveno.

[2] GIL – Global Interpreter Lock je v podstatě zámek, který oddělí paměťové prostory jednotlivých procesů překladače a vlákna si potom nemohou mezi sebou vyměňovat informace

[3] Konstruktor je speciální metoda, která je automaticky volána pří vzniku objektu. Jindy ji volat nelze.

Nejdůležitější část *run* určuje dění po spuštění vlákna. Nejdříve se vlákno pokusí připojit k oběma sériovým portům na Batyskafu a na oba pošle znak „*S*“ pro zahájení komunikace. Pokud se toto nepodaří, pošle do fronty „*error_q*“ chybovou zprávu, kterou potom převezme GUI a vypíše chybovou zprávu, kterou vidí uživatel. Dále následuje samotný *while* cyklus, který běží, dokud je znak „*alive*“ nastaven. Tento cyklus obsluhuje celé čtení dat z Batyskafu i zápis hodnot do Batyskafu. Podle doporučení z dokumentace modulu pySerial se nejdříve přečte pouze jeden byte pomocí *read(1)* a následně vše ostatní, co čeká v zásobníku. Právě tuto část ovlivňuje parametr timeout. V tomto místě přichází nekonzistentní data, která ještě nelze vykreslit. Batyskaf posílá data ve formátu například „*50*51*50*53*49**“, ale díky velké rychlosti čtení dat dostávám při každém průběhu cyklem pouze jeden znak, ale někdy to může být více znaků nebo žádný. Proto je nutné data skládat do vlastního zásobníku nazvaného *data1_buff*, s kterým dále pracuje menší podcyklus pro oddělení jednotlivých čísel.

```
while '*' in data1_buff:
                    # cyklus běží dokud je v zásobníku *
   pozice=data1_buff.find('*')
                    # najde číslo znaku, kde se nachází *
   if pozice != 0:
                    # kontrola, aby nebyla * na začátku
    cislo=int(data1_buff[0:pozice])/1000.0
                    # vytažení čísla z řetězce a převod na cm
    data1_buff=data1_buff[pozice+1:]
                    # umaže již získané číslo ze zásobníku
    timestamp = time.clock()
                    # časová značka kdy jsem číslo získal
    self.data_q1.put((cislo, timestamp))
                    # odeslání čísla se značkou pro další zpracování
   else:
                    #  pokud je * na začátku řetězce
    data1_buff=data1_buff[1:]
                    #  umaže * ze začátku řetězce
```

Tento jednoduchý a rychlý cyklus zaručuje, že pro další zpracování projdou vždy jen celá čísla, která byla ohraničena hvězdičkami. Díky použití cyklu *while* místo podmínky *if* si algoritmus poradí, i pokud dojde k nějakému pozdržení dat a ke zpracování přijde delší řetězec obsahující více hodnot. Součástí hlavního cyklu je také podmínka pro zapisování hodnot do Batyskafu, pro ovládání jednotlivých ventilů. To je realizováno pomocí sdílené proměnné *DB.zapis*, pokud je nastavena na *True*, tak si program přečte hodnoty nastavené na posuvnících a pošle je do Batyskafu. Na konci podmínky se *DB.zapis* nastaví na hodnotu *False*, aby se hodnoty nezapisovaly stále znova, ale pouze při změně. Poslední část metody *run* zajišťuje správné ukončení připojení k sériovému portu, tato část už je mimo hlavní cyklus a spustí se, když

je zavolána metoda vlákna *Stop*. Zajistí nastavení počátečních hodnot otevření ventilů Batyskafu a správné ukončení komunikace pomocí znaku „*F*“.

Poslední část vlákna *join* je zavolána při použití metody *Stop*. Zruší nastavení znaku „*alive*“ a zastaví vlákno.

Potom už je nutné pouze vytvořit instanci a vlákno spustit:

```
DB.com_monitor = ComMonitorThread(
                          # vytvoření instance nazvané com_monitor
    DB.data_q1,DB.data_q2,self.error_q,
                          # objekty typu Queue pro přenos dat
    "\\\\.\\COM10","\\\\.\\COM11", # názvy sériových portů
    600)                  # přenosová rychlost
DB.com_monitor.start()    # spuštění vlákna
```

4.4. Zpracování získaných dat

Dále je nutné data, přijímaná pomocí vlákna *com_monitor,* zpracovat. To znamená upravit je do vhodného formátu pro vykreslování a ukládat je do seznamů. A hlavně také umožnit nastavení vzorkování jaké si uživatel přeje. Ukládané seznamy obsahují nejen data získaná ze sériových portů, ale také všechna ostatní, která se dále používají pro vykreslování do grafů a další zpracování. Konkrétně se zpracovávají a ukládají data do tří oddělených seznamů. První seznam *DB.samples* obsahuje informace o aktuální poloze plováčku, uživatelem nastavené natočení řídícího a poruchového serva a původní a novou časovou značku. Druhý seznam, *DB.sampling*, slouží pro vykreslování grafů kontroly vzorkování. Poslední seznam, *DB.tlak*, sbírá hodnoty naměřených tlaků nad hladinou. Délka seznamů se řídí nastavením uživatele pomocí posuvníku „Délka dat“.

Tuto část kódu jsem přepisoval nejvíce-krát ze všech, hlavně kvůli problémům s přesností nastaveného vzorkování, které se lišilo až o 4 Hz od nastavené hodnoty. Hlavním důvodem, proč k tomuto docházelo, se nakonec ukázal operační systém. Zde se projevuje, že hlavní výhoda Pythonu může způsobit i problémy. Problém je totiž v tom, že stejná funkce se chová různě na různých platformách. A jak jsem i později zjistil, operační systém Windows, na kterém jsem aplikaci realizoval, není vhodný pro úlohy v reálném čase. Jde o to, že Windows není systém RTOS[4], který zaručuje, že časově kritické úlohy se spustí vždy, když mají. Windows je více-úlohový systém, kde některá z úloh může kdykoliv ovlivnit průběh časovače a tím ho například zpozdit.

[4] RTOS – „Real-Time Operating Systém“ Real-time systém je systém, ve kterém správnost výstupu je závislá nejen na správnosti výsledku výpočtu, ale též na čase, v němž je výsledek spočten.

Protože jsem už byl rozhodnut pro platformu Windows, hledal jsem řešení, jak vyřešit vzorkování alespoň s dostatečnou přesností. Nejdříve jsem vyzkoušel několik různých funkcí pro časování programu, které nabízí Python, například *sleep, wait, timer, usleep*, ale všechny vracely stejně nepřesné výsledky. Když totiž nahlédneme do zdrojového kódu těchto funkcí, tak všechny ve výsledku používají základní funkci *sleep*, ve které je „zakopán pes“. Abych se této funkci vyhnul, našel jsem v článku [41] od Microsoftu velice zajímavou funkci *QueryPerformanceCounter*, což je vlastně jen čítač cyklů procesoru. Po vydělení frekvencí procesoru *QueryPerformanceFrequency* lze získat čas, ale to je pouze jedna hodnota, kterou je nutné porovnávat s druhou. Kdybych napsal cyklus, který by neustále porovnával tyto časy, dokud by nenastal správný čas pro uložení dat, tak by byl procesor neustále vytížen na 100% a aplikace by byla velice pomalá až nepoužitelná. Proto je stejně nutné zahrnout do cyklu nějakou čekací funkci, která nebude tolik zatěžovat procesor. Zde jsem ale narazil na problém jakéhosi minimálního rozdílu časování funkce *sleep*, který právě způsobuje nepřesnosti vzorkování.

Bylo nutné jít ještě hlouběji do kódu, funkce *sleep* využívá pro časování systémovou funkci *select*. *Select* na Windows se používá pro výběr a kontrolu dostupnosti socketů[5], pokud je socket nedostupný, tak čeká po nastavenou dobu. Právě toto čekání lze využít, pokud vytvořím neexistující socket, pak funkce bude vždy čekat po nastavenou dobu. Do cyklu jsem tedy přidal *select* s nastavenou dobou na 9 miliontin sekundy, s tím, že by to mělo stačit na dostatečnou přesnost, ale toto byla příliš malá hodnota a CPU byl opět vytížen 100%. Při zvyšování doby čekání jsem narazil na zajímavý problém. Při jisté zlomové hodnotě se rázem snížilo vytížení procesoru, ale zároveň skokem opadla přesnost na původní špatné hodnoty. Tento skok si vysvětluji jako jakési minimální vzorkování systému Windows a použitím například Linuxového operačního systému by se tento problém vyřešil.

Protože na změnu operačního systému bylo už pozdě, podařilo se navrhnout elegantní metodu, jak se k dostatečně přesnému času postupně dostat. Naprogramoval jsem dvoustupňový časovač, který na prvním nepřesném stupni čeká 99 % času a v druhém stupni jen poslední procento času čeká s rozlišením 9 miliontin sekundy. Toto rozdělení dovoluje dostatečnou přesnost vzorkování s minimálním zatěžováním procesoru.

Nyní popíši konkrétní realizaci zpracování dat. Pro maximální nezávislost na ostatních procesech je zpracování dat opět odděleno od zbytku programu ve zvláštním vláknu, stejně jako *com_monitor*.

[5] Sockety se používají pro přenos dat po sítích nezávisle na použitém protokolu.

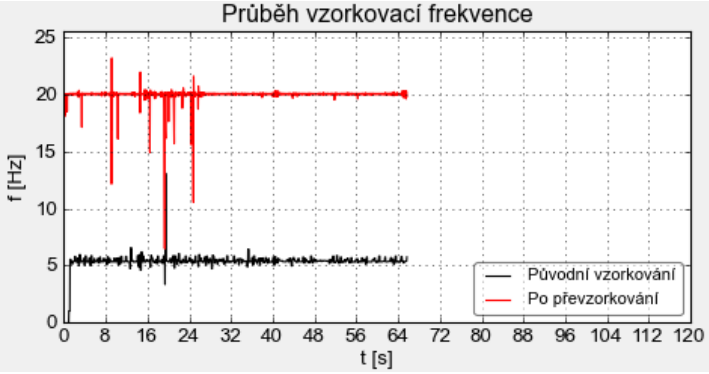
Vlákno je opět uvedeno metodou *__init__*, kde se zavádí potřebně počáteční hodnoty a také si zde připravuji socket pro funkci *select*.

Metoda *run* obsahuje kód běhu vlákna, který je celý tvořen *while* cyklem s kontrolou nastavení značky *alive*. Základní časovací struktura cyklu je, že si nejdříve uložím čas začátku cyklu, potom se provede zpracování dat a čekání 99 % nastaveného intervalu. Následně si uložím druhý čas. Odečtením rozdílu těchto dvou časů od celkového intervalu zjistím, jak dlouho má ještě čekat druhý přesný stupeň. Pokud náhodou zpracování dat a čekání 99 % intervalu trvá déle než celý interval, tak druhý stupeň se úplně přeskočí a rovnou se spustí další cyklus.

```
while self.alive.isSet():    # hlavní cyklus zpracování dat
   self.time1=time.clock()   # čas začátku cyklu
   #
   # zde se nachází funkce zpracování dat
   #
   # select čeká většinu nastaveného intervalu
   select.select([self.s], [], [], self.interval - 0.01)
   # čas po zpracování dat a blokování selectem
   self.time2 = time.clock()
   # výpočet doby po kterou bude čekat druhý stupeň
   doba = self.interval - (self.time2 - self.time1)
   # cyklus pro přesné dosažení celého intervalu
   while doba > (time.clock() - self.time2):
       select.select([self.s], [], [], .0000009)
```

Kód časování vzorkování

Tento způsob vrací výsledky s dostatečně přesnou vzorkovací frekvencí, ale i tak je stále přesnost závislá na zatížení procesoru. Na Obr. 18 je názorně vidět ovlivnění vzorkovací frekvence, prvních 30 sekund jsem zatížil procesor na 100 % a dále je bez zátěže, kde už je přesnost velice dobrá. Proto je nutné při práci s programem nezatěžovat počítač jinými věcmi.



Obr. 18 – Průběh vzorkovací frekvence při zátěži procesoru

Jak už bylo řečeno, zpracovaná data se ukládají do tří seznamů. První seznam, *DB.samples*, obsahuje informace o aktuální poloze plováčku, uživatelem nastavené natočení řídícího a poruchového serva a původní a novou časovou značku. Poloha

plováčku je získávána z datové fronty, do které data přidává vlákno *com_monitor*. Vybráním dat z fronty se automaticky tato data z fronty odstraní. Pokud je nastavená vzorkovací frekvence nižší než vzorkovací frekvence Batyskafu, potom se z příchozích dat počítá průměr a použije se průměrná hodnota. Naopak pokud je vzorkování vyšší, není dostatek hodnot a je nutné použít vždy hodnotu z minulého cyklu. Tímto je zaručena úplná nezávislost zpracování dat a příchozích dat z Batyskafu. Hodnoty natočení poruchového a řídícího serva se načítají podle nastavení posuvníků v grafickém rozhraní. V každém cyklu je také kontrola délky seznamu, pokud je delší než nastavená, odebere se vždy z konce seznamu jeden řádek.

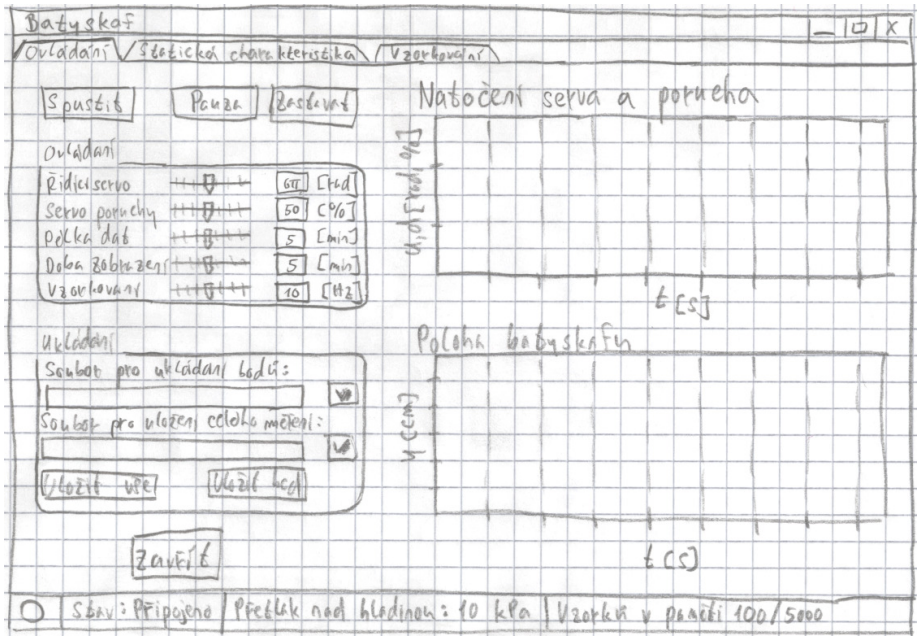
Druhý seznam, *DB.sampling*, slouží pro vykreslování grafů kontroly vzorkování. Hodnota aktuálního vzorkování se počítá jako převrácená hodnota z rozdílu časů spuštění posledních dvou cyklů. Dále se počítá hodnota původního vzorkování Batyskafu, která je opět převrácenou hodnotou posledních dvou časových značek příchozích dat z Batyskafu.

Poslední seznam, *DB.tlak*, sbírá hodnoty naměřených tlaků nad hladinou. Z příchozích dat se opět počítá průměr nebo se používají hodnoty z předešlého cyklu, podle nastavené vzorkovací frekvence. Délka tohoto seznamu je pevně nastavena na 30 vzorků, ze kterých se počítá průměrný tlak. Hodnoty tlaku jsou značně nestálé a orientační, proto je použit průměr z 30 hodnot, který dává lepší výsledky než přímé zobrazení.

Tyto tři seznamy jsou dále využity v grafickém rozhraní pro vykreslování a zobrazování hodnot.

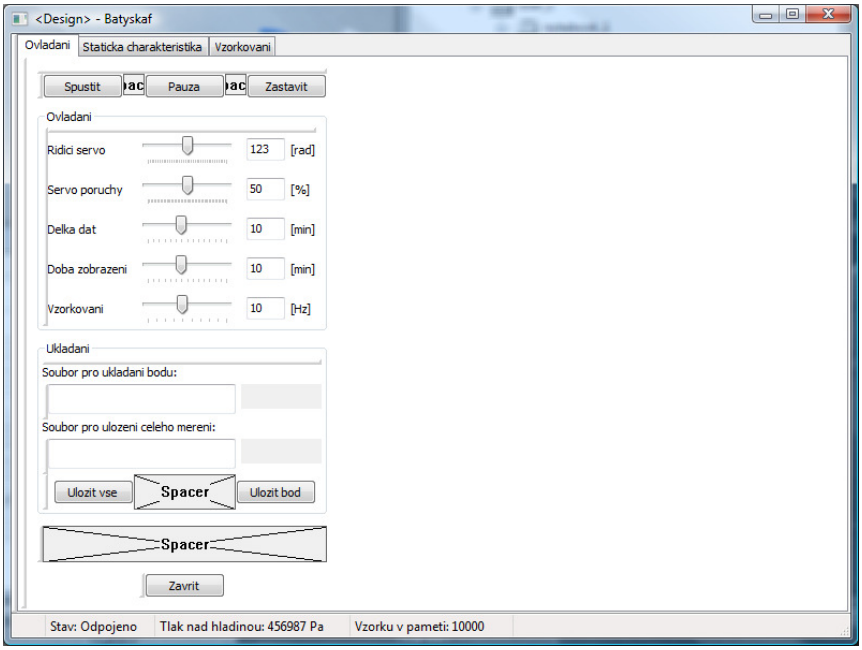
4.5. Grafické uživatelské rozhraní

Grafické rozhraní je nedílná součást dnešních aplikací. Jen obtížně si lze představit efektivní ovládání pouze v textovém režimu. Po zvolení programu wxGlade, jako nástroje pro tvorbu GUI, si bylo nutné rozmyslet strukturu a rozložení prvků v okně programu. Takže nejdříve jsem si načrtl, jak by okno mohlo vypadat:



Obr. 19 – Prvotní náčrt rozložení prvků grafického rozhraní

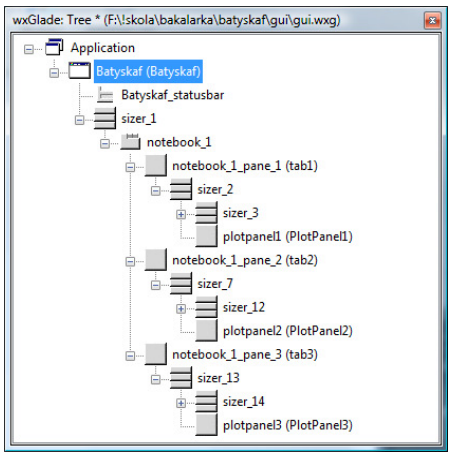
Nyní už pouze stačilo takto připravené grafické rozhraní vytvořit v prostředí wx-Glade, kde se vytvoří pouze GUI bez funkcí, které se musí ručně dopsat do kódu.



Obr. 20 – Vytvořený návrh GUI v programu wxGlade

Systém tvorby GUI ve wxPythonu je založen na takzvaných sizerech. Sizery udávají, co a jak se má zmenšit, zvětšit nebo přesunout při změně velikosti okna programu. Sizer se dá popsat jako několik buněk tabulky, s tím, že lze do sebe vnořit libovolné množství sizerů. Každému sizeru lze nastavit různé vlastnosti, základní jsou orientace (vertikální nebo horizontální), pozice (pokud je například více sizerů pod

sebou, tak kolikátý bude) a proporce (pokud je volné místo, tak jestli se má roztáhnout nebo si zachovat minimální velikost). Díky nekonečným možnostem kombinování různých sizerů, lze vytvořit velice komplexní grafické rozhraní. Na Obr. 21 je přehledně zobrazena struktura programu, skládajícího se ze sizerů.



Obr. 21 – Základní struktura ovládacího programu

Každá aplikace může mít více podoken, v mém případě jsem si vystačil pouze s jedním, nazvaným Batyskaf, které jsem rozdělil na tři přepínací záložky. Dále jsem použil stavový řádek pro zobrazování údajů o aktuálním stavu programu i Batyskafu. Lze zde nalézt informace o tom, jestli je program zrovna připojen k Batyskafu, o přetlaku nad hladinou, počet vzorků v seznamu, které lze uložit, a posledním údajem je hodnota vzorkovací frekvence. Dále už je program tvořen pomocí sizerů, do kterých se postupně vkládají jednotlivé prvky nebo další sizery.

Na Obr. 20 je vidět, že zde ještě nejsou grafy, ani žádné popisky k nim. To je dáno tím, že grafy jsou vykreslovány pomocí odděleného modulu a proto je vhodné si ve wxGladu pouze připravit panel (v mém případě prvky nazvané PlotPanel), do kterého se později ručně připíše kód pro vložení grafu se všemi popisky.

Po umístění všech požadovaných prvků stačí pouze vygenerovat kód. Zde je možnost volby, jestli chceme kód mít v jednom souboru nebo ve více. Mně se osvědčilo mít kód rozdělen do několika souborů pro větší přehlednost. Rozdělení do souborů probíhá podle struktury programu, viz Obr. 21. wxGlade používá techniku takzvaných střežených regionů, což v praxi znamená, že lze do již vygenerovaného kódu přidávat vlastní úpravy mimo tato vyznačená místa a wxGlade by při případném opětovném generování kódu neměl tyto úpravy přepsat. Bohužel v tomto ohledu wxGlade zatím není příliš propracovaný a proto jsem radši po prvotním vygenerování kódu všechny úpravy dopisoval ručně.

```
# begin wxGlade: dependencies
  from PlotPanel1 import PlotPanel1
```

```
        # end wxGlade
```

Označení střeženého regionu wxGladem

Struktura vygenerovaného kódu je velice přehledná. Celou aplikaci zastřešuje soubor *app.py*, do kterého je importován soubor *Batyskaf.py*, kde je definován stavový řádek a časovač pro jeho aktualizaci. Dále je zde ošetřeno správné ukončování celého programu, aby se správně ukončila vlákna a odpojil Batyskaf. Také jsou zde samozřejmě importovány soubory jednotlivých záložek tab1, tab2 a tab3. V inicializační části souboru záložky se definují a umisťují veškeré použité prvky, včetně sizerů a událostí pro tlačítka a další prvky. Dále jsou pouze metody k veškerým událostem. Soubor každé záložky importuje každý svůj *plotpanel*, do kterého je vložen graf. Soubor *plotpanelu* se opět skládá z inicializační části, ve které je vytvořen kompletní graf včetně ovládacích prvků. Ke grafu lze přidat veškeré popisky i nadpis, včetně legendy. Dále stačí metoda pro překreslování grafu, která se spouští buď tlačítkem, nebo časovačem pro plynulý pohyb grafu.

Plynulost animace grafu a hardwarová nenáročnost byla jednou z dalších překážek. Díky velké svobodě volby modulů je možno si vybrat z několika balíčků pro GUI a stejně tak z několika balíčků pro vykreslování grafů. Tato svoboda má ale své nevýhody. Pokud potřebuji určitou spolupráci více modulů (například animace grafu), tak autoři nemohou vědět, které moduly si vyberu, aby připravili hotové řešení. To znamená, že je každý uživatel nucen si toto řešení najít a realizovat. Ve většině případů řešení existuje, dokonce ne jen jedno a právě toto hledání nejlepšího řešení dokáže zabrat nejvíce času.

V článku [42] z dokumentace SciPy lze nalézt velice pěkný přehled animačních metod a dokonce i porovnání výkonu jednotlivých GUI. Podle tohoto porovnání by mělo podávat nejlepší výsledky rozhraní GTK, ale i přesto jsem se rozhodl použít backend WXAgg, kvůli jeho přehlednosti kódu a jednoduchosti instalace.

Při zkoušení různých realizací animace jsem neustále narážel na dva hlavní problémy. Buď byla animace až nepoužitelně hardwarově náročná nebo sice plynulá a nenáročná, ale zase velice nestabilní a jakýkoliv zásah způsobil pád programu. Nakonec jsem se inspiroval článkem [40], kde Eli použil jednoduchou a účinnou metodu. Princip spočívá v tom, že si nejdříve v inicializační části vytvoříme kompletní plochu grafu i s popisky a legendou a v metodě pro aktualizaci vždy pouze překreslíme data.

```
def update(self, filename):     # funkce aktualizace grafu
  data=np.loadtxt(filename)     # načtení dat z txt souboru
  miny,maxy=min(data[:,3]),max(data[:,3])   # získání mezí osy y
  minx,maxx=min(data[:,2]),max(data[:,2])   # získání mezí osy x
```

```
self.axes2.set_ylim((miny*0.9, maxy*1.1))# nastavení rozsahu osy y
self.axes2.set_xlim((minx*0.9, maxx*1.1))# nastavení rozsahu osy x
self.axes2.invert_yaxis()     # otočení osy y, plováček klesá..

self.plot_data3.set_xdata(xdata)    # nastavení dat pro osu x
self.plot_data3.set_ydata(ydata)    # nastavení dat pro osu y

self.figure2.canvas.draw() # vynucení překreslení grafu
```

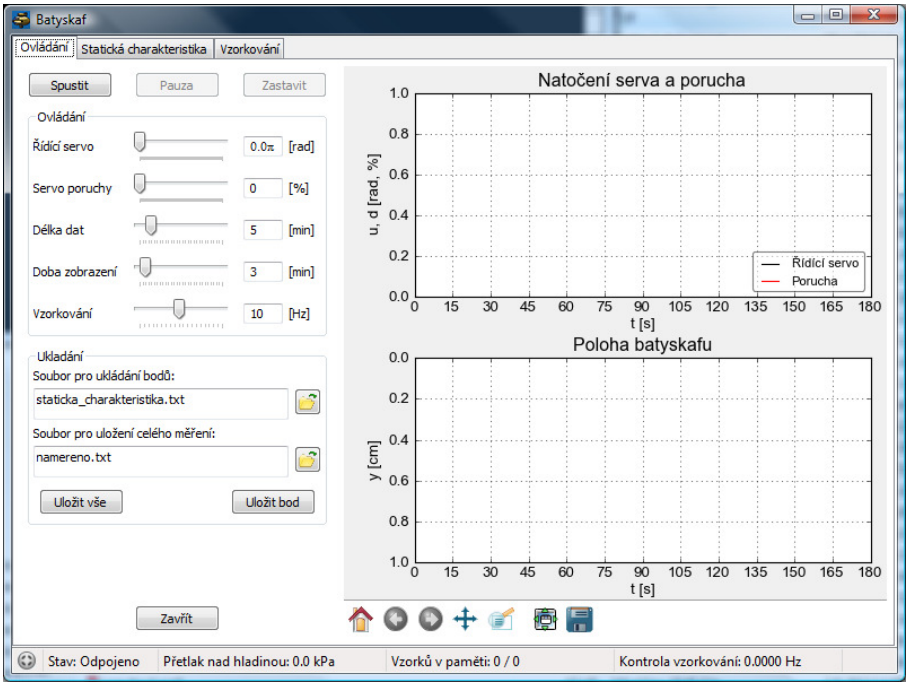
Ukázka metody pro překreslování grafu

Nyní stačí pouze, vždy když chceme obnovit graf, zavolat tuto jednoduchou metodu. Původně byl nastaven časovač, který každých 100 ms obnovil graf, tato rychlost byla dostatečná pro plynulou animaci se zatížením procesoru kolem 30 %. Po přidání všech popisek a legendy se ale až dvojnásobně zvýšilo zatížení procesoru, což už více ovlivňovalo přesnost vzorkovací frekvence. Proto jsem byl nucen zvětšit čas obnovy na 200 ms, s tím, že animace už není tak plynulá, ale zatížení procesoru kleslo na přijatelnou úroveň. Toto zvýšení je dáno tím, že funkce *draw()* vždy nepřekreslí pouze data, ale i všechny popisky a legendy.

Dále bylo nutné pro optimální chod programu ošetřit, aby se všechny grafy neobnovovaly neustále, ale jen když jsou vidět, to znamená, když je aktivní záložka, na které jsou. Z pohledu programu jsou totiž všechny záložky pořád aktivní, i když nejsou vidět. Ošetření bylo jednoduché, pomocí události, která se zavolá vždy při změně záložky.

4.6. Popis výsledné aplikace a možnosti zlepšení

Po úspěšném překonání všech úskalí, se podařilo napsat program, který je schopný dostatečně spolehlivě ovládat laboratorní úlohu Batyskaf a zároveň s ním lze měřit i přechodovou a statickou charakteristiku.



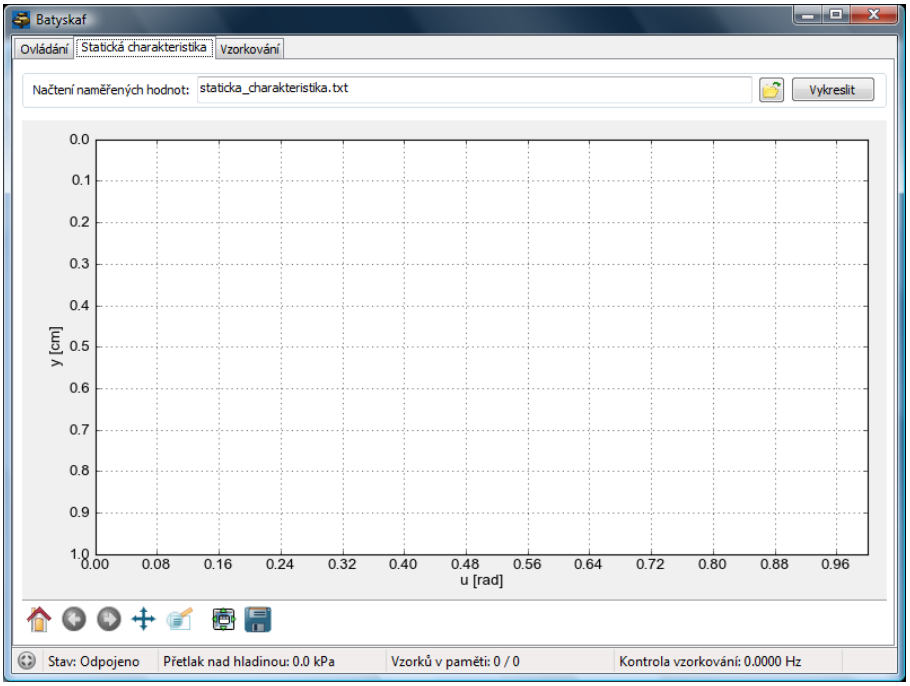
Obr. 22 – Hlavní ovládací záložka programu

Jak je vidět na Obr. 22 – Hlavní ovládací záložka programuObr. 22, cílem bylo přehledné, jasně rozdělené rozhraní. Levá polovina slouží pro vstupy uživatele a pravá je z pohledu uživatele výstupní. Dole ve stavovém řádku se nachází základní informace o stavu programu. První ikona a zároveň textový stav indikují, zda jsme připojeni k úloze přes sériový port. Pokud jsme připojeni, další pole ukazují tlak nad hladinou v Batyskafu a počet vzorků v paměti. Maximální hodnota „Vzorků v paměti“ se řídí nastavením posuvníků „Délka dat“ a „Vzorkování“. Po spuštění programu se vzorky ukládají do paměti rychlostí danou nastavenou vzorkovací frekvencí, po dobu danou délkou dat. Jakmile se naplní maximální hodnota vzorků v paměti, tak se vždy umaže jeden vzorek z konce a jeden přidá na počátek seznamu. Pole „Kontrola vzorkování“ ukazuje skutečnou vzorkovací frekvenci, kterou lze porovnávat s nastavenou.

Horní tlačítka „Spustit“ a „Zastavit“ řídí samotné připojení úlohy pomocí sériové linky, tedy připojení a odpojení od úlohy. Tlačítko „Pauza“ pouze pozastaví vykreslování grafu, ale záznam dat i ovládání stále funguje. Tato funkce se hodí, pokud bychom si chtěli prohlédnout detailněji některé části grafu. Následuje rámeček „Ovládání“, ve kterém jsou všechny důležité ovládací prvky. První dva posuvníky slouží pro ovládání otevření ventilů Batyskafu. Posuvník „Řídící servo“ řídí natočení hlavního serva v rozsahu 0 až 6 (12π) otáček. Druhé servo slouží k simulaci poruchy (úniku vzduchu), nastavit lze v rozsahu 0 až 100 %. Jak už bylo zmíněno, posuvník „Délka dat“ určuje maximální počet vzorků, které se budou ukládat do paměti. Pro

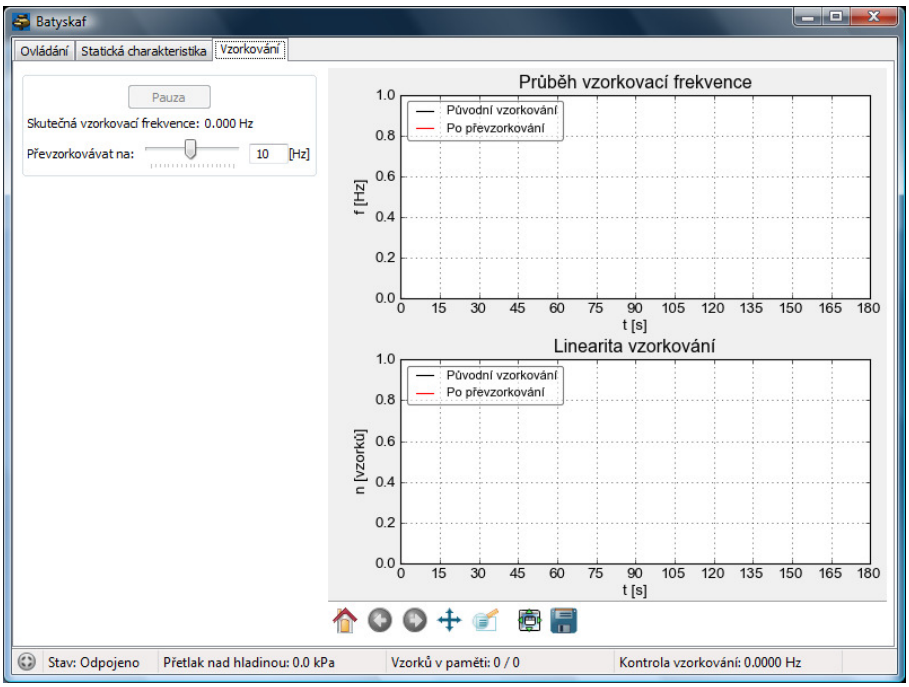
lepší představu je údaj v minutách. „Doba zobrazení“ ovlivňuje pouze viditelnou část osy x v grafech. Lze nastavit čas od jedné až třicet minut, který chceme vidět v grafech. Toto nastavení nijak neovlivňuje ukládaná data, pouze zobrazení. Vzorkovací frekvenci lze nastavit posledním posuvníkem v rozsahu 1 až 20 Hz. Kterýkoli z těchto údajů lze ručně přepsat ve vstupních polích vedle jednotlivých posuvníků. Naměřená data lze ukládat dvěma způsoby. Buď lze uložit vše, co se zrovna nachází v paměti do textového souboru. Toto zahrnuje vše, co je v prvním zpracovávaném seznamu (aktuální poloha plováčku, uživatelem nastavené natočení řídícího a poruchového serva a původní a nová časová značka). Druhá možnost slouží pro uložení všech těchto hodnot jen v aktuálním časovém okamžiku, to znamená vždy první řádek ze seznamu. Tuto funkci lze nejlépe využít pro měření statické charakteristiky, kdy chceme zachytit bod, ve kterém je soustava stabilní. Poslední tlačítko „Zavřít“ ukončí celou aplikaci, v případě, že jsme se zapomněli odpojit, tak vše korektně ukončí.

Se všemi grafy nejen na této záložce lze také libovolně manipulovat pomocí panelu, který lze nalézt ve spodní části grafu. Dovoluje libovolný posun do stran nebo nahoru a dolů, stejně tak i přibližování a oddalování grafu. Dále si lze uložit graf jako obrázek v jednom z mnoha nabízených formátů, včetně vektorových souborů. Tento panel je viditelný, pouze pokud je program odpojen od Batyskafu nebo je vykreslování pozastaveno. Při běhu animace by s grafem nešlo rozumně manipulovat.



Obr. 23 – Druhá záložka slouží pro vykreslení naměření statické charakteristiky

Pokud jsme si na předchozí záložce uložili několik různých bodů, kde se plováček ustálil, pak stačí pouze pomocí tlačítka s obrázkem složky vybrat textový soubor s body a stisknout tlačítko „Vykreslit“. Program následně ukáže v grafu závislost výstupu na vstupu. S grafem lze opět plnohodnotně manipulovat.

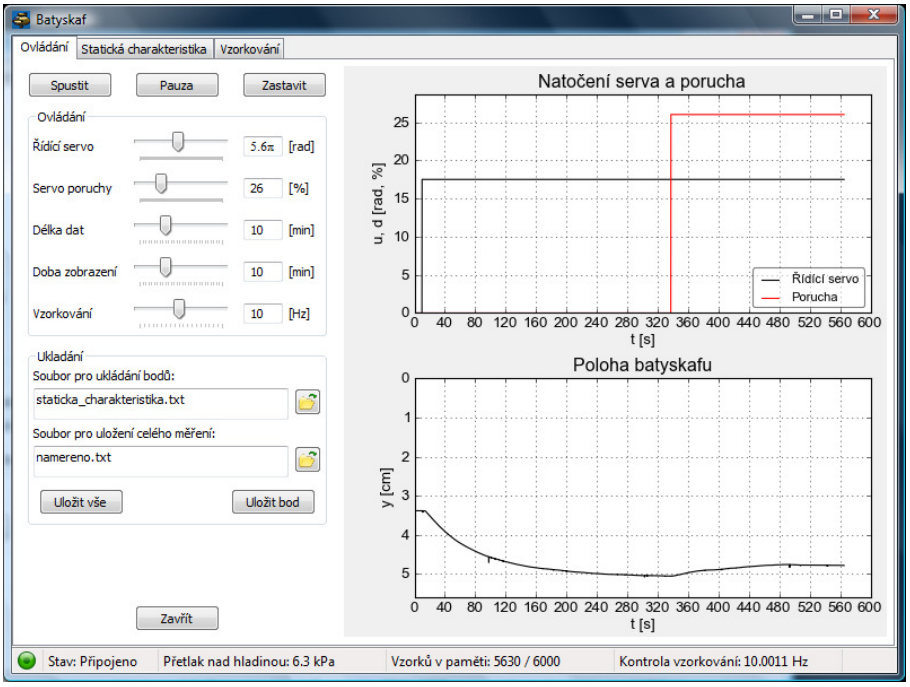


Obr. 24 – Poslední záložka slouží pro kontrolu vzorkovací frekvence

Záložka „Vzorkování“ už neslouží přímo k ovládání ani měření na Batyskafu. Je to pomůcka pro sledování a kontrolu vzorkovací frekvence. Je zde pouze jeden posuvník, který je shodný s posuvníkem na první záložce pro nastavení vzorkovací frekvence. Tlačítko „Pauza“ opět pouze pozastaví vykreslování grafů, ale záznam dat běží dále. Oba grafy zobrazují dvě datové linie. Černou, která značí vzorkovací frekvenci, s kterou přichází data z Batyskafu a červenou, která se řídí nastavením uživatele a přesností programu. První graf slouží pro kontrolu stálosti vzorkovací frekvence v čase a druhý představuje závislost jednotlivých vzorků na jejich poloze v čase. To znamená, že výsledkem by měla být vždy přímka stoupající pod různým úhlem podle vzorkovací frekvence. Pokud dojde k neočekávaným změnám ve vzorkovací frekvenci, lze je z grafu na první pohled vyčíst.

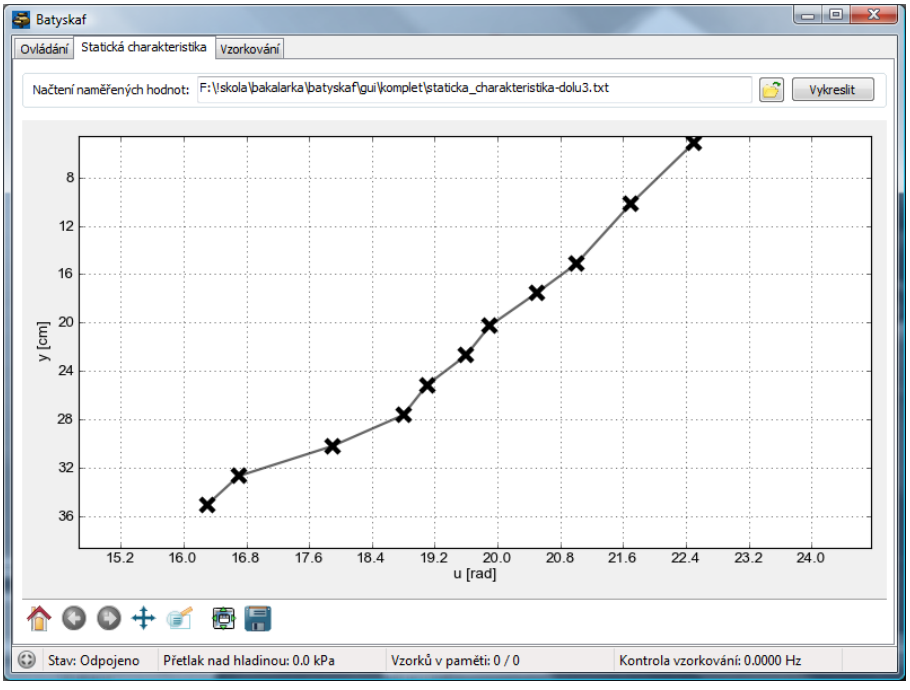
Používání programu

Abych tento program nepředstavoval pouze v teoretické rovině, nyní předvedu několik možností využití aplikace.



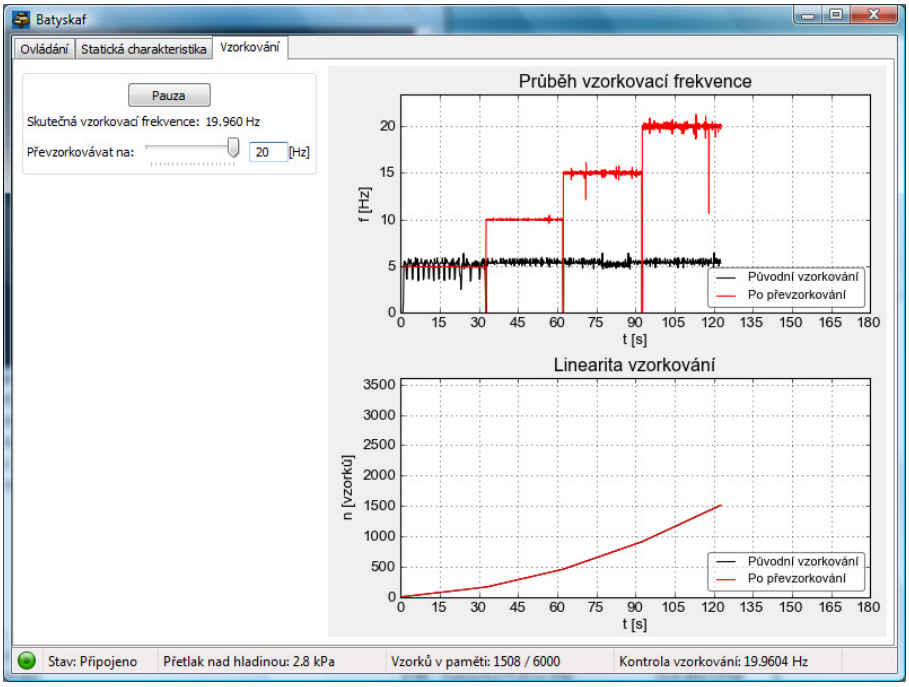
Obr. 25 – Měření přechodové charakteristiky s následným zapůsobením poruchy

Na první záložce horní graf zobrazuje průběh vstupních veličin a dolní výstupní – polohu plováčku. Osa y je invertovaná pro lepší představu polohy. Jako ukázku (Obr. 25) předvádím měření přechodové charakteristiky. Nastavením řídícího serva na 17,6 radiánu vznikl skok a po asi 6 minutách se plováček stabilizoval v hloubce 5 cm. Po této době jsem také do soustavy zanesl působení poruchy, které změnilo stabilní polohu plováčku.



Obr. 26 – Vykreslení statické charakteristiky

Měření statické charakteristiky je na Batyskafu poměrně obtížné, protože je to velice pomalá soustava a poznat kdy už je plováček ve stabilní poloze není jednoduché. Dalším důvodem je velká citlivost otevření ventilu, kde i změna jen o jeden bod dokáže způsobit velké změny polohy. Výslednou statickou charakteristiku, kterou jsem měřil při pohybu plováčku směrem dolů lze vidět na Obr. 26.



Obr. 27 – Ukázka změny vzorkovací frekvence

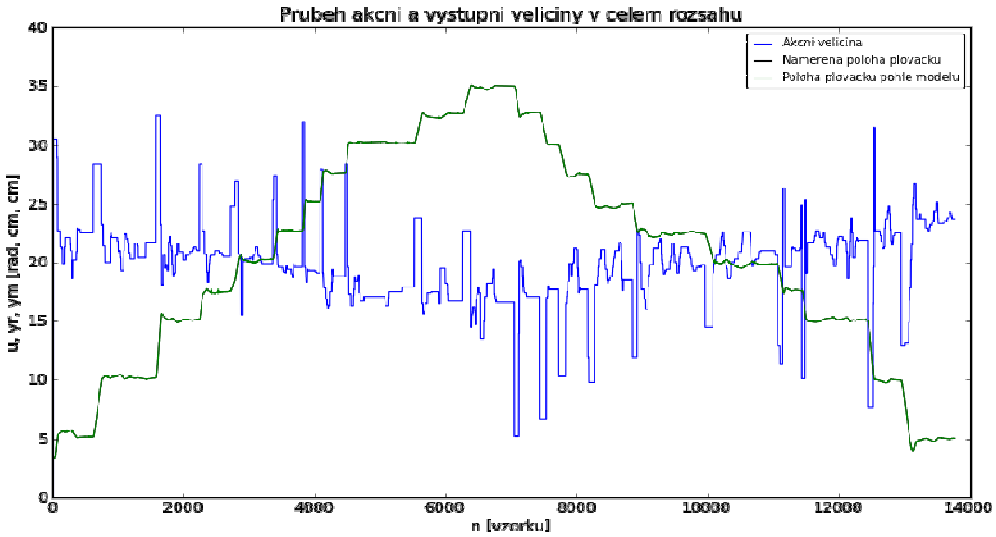
Poslední záložka slouží pro kontrolu vzorkovací frekvence. Ukázku změn vzorkovací frekvence a jejich promítnutí do grafů lze najít na Obr. 27. Na horním grafu je názorně vidět jak se stoupající frekvencí klesá její stálost a je více rozkmitaná. To je dáno hlavně tím, že vyšší frekvence jsou více citlivé na zatížení procesoru, například už jen to, které způsobí můj program. Pokud bychom pozastavili vykreslování grafu a pak opět spustili, uvidíme, že během pozastavení vykreslování byla frekvence klidná a bez roztřesení. Samotná animace grafu totiž na testovacím počítači zatěžovala procesor asi na 30 až 40 %. Původní frekvence Batyskafu se nemění, protože na tu změna vzorkovací frekvence programu nemá vliv. Pouze po prvních 30 sekundách je vidět jistá změna, která je daná tím, jak s daty zachází program, pokud je nastavená vzorkovací frekvence nižší nebo vyšší než původní. Na dolním grafu lze sledovat změnu strmosti křivky se změnou vzorkovací frekvence.

Možnosti dalšího vývoje programu

V rámci práce jsem napsal funkční a dostatečně stabilní ovládací prostředí pro Batyskaf, ale v programu je stále mnoho částí, které by šlo napsat jinak, lépe a také nepřeberné možností nových funkcí, o které bych program rád rozšířil.

Jsem přesvědčen, že komunikační část programu je dostatečně stabilní a robustní a další vylepšení není momentálně nutné. V části zpracování dat by bylo vhodné se zamyslet nad vylepšením časovacího algoritmu, a aby nebyl tak snadno ovlivnitelný zátěží procesoru. Pokud by měl být program opravdu používán v provozu, bylo by nevyhnutelné změnit operační systém, například na linuxově založený systém. Tato změna by celkově přispěla k lepší stabilitě programu a i jeho větší přesnosti. Bylo by ovšem nutné celý program projít a optimalizovat pro multiplatformní chod. Jako příklad lze uvést časové funkce modulu *Time*, které se chovají různě na různých platformách, dále je také mnoho rozdílů v zobrazování GUI, které by také potřebovalo upravit. V případě provozů, kde by kriticky záleželo na přesnosti časování a rychlosti odezvy, by se musely klíčové části programu přepsat do jiného jazyka, například C. Toto omezení je dáno tím, že Python je interpretovaný programovací jazyk, který se nikdy nemůže rovnat výkonu C. Má ovšem mnoho jiných nesporných výhod, proto aplikační oblast Pythonu leží trochu jinde. Vzhled grafického rozhraní je dostatečně přehledný a funkční, ale jisté zlepšení by mohla přinést změna prostředí na GTK, které by mělo podávat lepší výkony při menší zátěži procesoru. Hlavní rozdíl by měl být vidět na pohyblivých grafech. Zmenšení zátěže procesoru by také přineslo zlepšení přesnosti časování programu. Další možností by bylo použít pro vykreslování grafů přímo grafické knihovny, ale to by obnášelo napsání vlastních funkcí pro vykreslování všech prvků grafu.

Nyní je Batyskaf ovládán přímo akční veličinou – natočením serva. Tento přístup není zdaleka optimální a pro pohodlnější ovládání je vhodnější nastavovat přímo polohu plováčku v centimetrech. Nastavení žádané veličiny, místo akční umožní regulátor. V původním programu k Batyskafu je na výběr z několika různých metod regulace. Během vývoje a testování programu v laboratoři na úloze Batyskaf jsem pozoroval, že úloha je silně závislá na několika vnějších vlivech, které velkou měrou ovlivňují chování celé soustavy. Konkrétně jde například o okolní tlak ovzduší, množství vody ve válci nebo množství vzduchu v plováčku. Tato velká proměnlivost soustavy vylučuje použití regulátorů založených na pevném modelu, ale je nutné použít adaptivní regulaci. Řízení úlohy Batyskaf pomocí neuro-regulátoru již řešil Ing. Smetana ve své diplomové práci [43]. Proto by další vývoj programu mohl zahrnovat integraci nelineárního adaptivního řízení. Princip by spočíval v adaptivním regulátoru, který by na počátku neměl žádné informace o soustavě, a jeho ladění by probíhalo za chodu pomocí systému adaptovaných vah, které by se postupně zpřesňovaly.



Obr. 28 – Zaznamenaná data, která je možno použít pro identifikaci modelu

Příklad identifikace adaptivního modelu soustavy Batyskafu z naměřených hodnot lze vidět na Obr. 28. Modrá křivka značí akční veličinu (natočení řídícího serva), její „roztřesenost“ je dána tím, že byla ovládána přímo, ručně a pro rychlejší pohyb mezi jednotlivými polohami bylo nutné nastavit dočasně skokově vyšší hodnotu. Černá a zelená křivka se kryjí, jsou to křivky výstupní veličiny (poloha plováčku) podle identifikovaného modelu a skutečnosti.

5. ZÁVĚR

Cílem této bakalářské práce bylo vytvoření grafického uživatelského rozhraní, včetně komunikačních algoritmů pro ovládání laboratorní úlohy Batyskaf v programovacím jazyku Python. Také jsem zhodnotil vhodnost jazyka Python pro tento typ úloh a hledal jsem jeho silné a slabé stránky. Programovací jazyk Python má velice rozsáhlé možnosti použití, jde o vyspělý jazyk, ve kterém lze naprogramovat cokoliv. Jeho filozofie je založena na open-source, to nám dává na jednu stranu možnost vždy si cokoliv upravit k obrazu svému a téměř nekonečné možnosti výběru řešení. Na druhou stranu je tato svoboda i na škodu, protože v případě, kdy je nutné robustní a rychlé řešení, tak je takové řešení nutné hledat nebo vytvářet. Python nabízí pouze nástroje k řešení problému, na použití a kombinaci těchto nástrojů už musí každý přijít sám, případně za pomoci rozsáhlé uživatelské základny. Díky tomuto charakteru jazyka se nyní spíše používá jako skriptovací jazyk nebo ho používají domácí kutilové, které netlačí čas a hledání řešení je baví. V současné době, kdy se rozšiřuje nejnovější, třetí verze jazyka, začíná být jazyk připraven i pro profesionální použití ve velkých firmách a pro projekty, kde je nutné zaručit stabilitu a rychlost reakce. Jednou z nejsilnějších stránek Pythonu je přehlednost kódu, která zaručuje maximální efektivitu a dokáže vyrovnat časové ztráty při hledání řešení problému. Pro aplikace ovládání a řízení v reálném čase lze Python použít, ale pouze v omezené míře. Díky své povaze interpretovaného jazyka nemůže zaručit vždy okamžité a přesné reakce, které jsou nutné u řídících aplikací většího rozsahu. Naopak v laboratořích, kde je důležitá spíše rychlost vývoje než stabilita se Python uplatní velice dobře i pro úlohy řízení.

V rámci této práce jsem také napsal program pro ovládání úlohy Batyskaf. Tento program umožňuje plné ruční ovládání úlohy s možností flexibilního zobrazení průběhu veličin v reálném čase. Akční veličinou je natočení řídícího serva, které ovládá jehlový ventil. Poruchovou veličinu simuluje druhé servo, které ovládá druhý jehlový ventil. Výstupní veličinou je odměřená poloha plováčku. Všechny tyto veličiny lze sledovat v grafech v reálném čase. Pro příklad použití programu je možné měřit přechodovou nebo statickou charakteristiku soustavy s následným zobrazením přímo v programu. Během testování aplikace jsem nenarazil na žádné závažné nedostatky a lze ji bezpečně používat pro ovládání úlohy. Jisté problémy dělala pouze velká citlivost soustavy na vnější vlivy, a i na malé akční zásahy. To způsobuje, že přímé ovládání, bez regulátoru, je velice obtížné a zdlouhavé. Pro tento typ soustavy je vhodný adaptivní typ regulátoru, který je možné v budoucnu do programu zapracovat.

6. POUŽITÉ ZDROJE

[1] JIRKOVSKÝ, J. *Batyskaf* [online]. 3. 9. 2006 [cit. 17. 4. 2012]. Dostupné na WWW: <www.ar-batyskaf.wz.cz>

[2] HOFREITER, M. *Netradiční laboratorní modely pro výuku automatického řízení*. Automatizace, 2006, roč. 49, č. 1, s. 10-11. ISSN 0005-125X.

[3] JIRKOVSKÝ, J. *Počítačové modelování a řízení laboratorního modelu "Batyskaf" : diplomová práce.* Praha : ČVUT, Fakulta Strojní, 2005. 110 l., 3 l. příl. Vedoucí diplomové práce Doc. Ing. Milan HOFREITER, CSc.

[4] TRNKA, P., KOPECKÝ, M. *Laboratorní úloha "Batyskaf 2, 3"* [online]. 15. 8. 2009 [cit. 17. 4. 2012]. Dostupné na WWW: <fsid.cvut.cz/cz/u12110/ar/ulohy/bat2.htm>

[5] URBÁNEK, D. *Laboratorní úloha "Batyskaf"* [online]. 2010 [cit. 17. 4. 2012]. Dostupné na WWW: <vlab.fsid.cvut.cz/cz/ulohy/batyskaf.php>

[6] WASSERBAUEROVÁ, P. *Matematický model zařízení „BATYSKAF" : diplomová práce.* Pardubice : Univerzita Pardubice, Fakulta chemicko-technologická, 2008. 51 l., 4 l. příl., Vedoucí diplomové práce Doc. Ing. František Dušek, CSc.

[7] HUMUSOFT s.r.o. *Real Time Toolbox for use with MATLAB and Simulink* [online]. 20. 8. 2004 [cit. 17. 4. 2012]. Dostupné na WWW: <humusoft.cz/www/rt/datasheet/rtt312.pdf>

[8] PRESCOD, P. *Why I Promote Python* [online]. 5. 3. 2000 [cit. 20. 4. 2012]. Dostupné na WWW: <www.prescod.net/python/why.html>

[9] ROSSUM, G. *Personal History - part 1, CWI* [online]. 20. 1. 2009 [cit. 20. 4. 2012]. Dostupné na WWW: <python-history.blogspot.com/2009/01/personal-history-part-1-cwi.html>

[10] ROSSUM, G. *Early Language Design and Development* [online]. 3. 2. 2009 [cit. 20. 4. 2012]. Dostupné na WWW: <python-history.blogspot.com /2009/02/early-language-design-and-development.html>

[11] ROSSUM, G. *Python's Design Philosophy* [online]. 13. 1. 2009 [cit. 20. 4. 2012]. Dostupné na WWW: <python-history.blogspot.com/2009/01/pythons-design-philosophy.html>

[12] PYTHON COMMUNITY. *The Python Logo* [online]. [cit. 20. 4. 2012]. Dostupné na WWW: <www.python.org/community/logos/>

[13] ROSSUM, G. *What's New In Python 3.0* [online]. 20. 4. 2012 [cit. 22. 4. 2012]. Dostupné na WWW: <docs.python.org/py3k/whatsnew/3.0.html>

[14] KUCHLING, A. *What's New in Python 2.7* [online]. 20. 4. 2012 [cit. 22. 4. 2012]. Dostupné na WWW: <docs.python.org/whatsnew/2.7.html>

[15] KUCHLING, A. *What's New in Python 2.6* [online]. 20. 4. 2012 [cit. 22. 4. 2012]. Dostupné na WWW: <docs.python.org/whatsnew/2.6.html>

[16] BEIGUI, P. *Should I use Python 2 or Python 3?* [online]. 20. 4. 2012 [cit. 22. 4. 2012]. Dostupné na WWW: <wiki.python.org/moin/Python2orPython3>

[17] BODDIE, P. *Python Implementations* [online]. 21. 4. 2012 [cit. 24. 4. 2012]. Dostupné na WWW: <wiki.python.org/moin/PythonImplementations>

[18] PYTHON COMMUNITY. *Download Python for Other Platforms* [online]. [cit. 24. 4. 2012]. Dostupné na WWW: <www.python.org/getit/other/>

[19] HUGHES, J.M. *Real World Instrumentation with Python*. First Edition. Sebastopol (California) : O'Reilly Media, Inc., 2010, 597p. ISBN 978-0-596-80956-0

[20] LIECHTI, C. *pySerial*. [online]. 3. 8. 2010 [cit. 24. 4. 2012]. Dostupné na WWW: <pyserial.sourceforge.net/pyserial.html>

[21] DUYÉ, B. *USB-types*. [online]. 1. 1. 2009 [cit. 24. 4. 2012]. Dostupné na WWW: <cs.wikipedia.org/wiki/Soubor:Types-usb_new.svg>

[22] LAIRSON, W. *PyUSB*. [online]. 22. 4. 2012 [cit. 24. 4. 2012]. Dostupné na WWW: <sourceforge.net/apps/trac/pyusb/>

[23] LIECHTI, C. *pyParallel.* [online]. 27. 1. 2005 [cit. 24. 4. 2012]. Dostupné na WWW: <pyserial.sourceforge.net/pyparallel.html>

[24] BRONGER, T. Python GPIB etc. support with PyVISA. [online]. 21. 11. 2006 [cit. 24. 4. 2012]. Dostupné na WWW: <pyvisa.sourceforge.net>

[25] WIKIPEDIE: Otevřená encyklopedie. *GPIB* [online]. c2012 [cit. 24. 04. 2012]. Dostupný z WWW: <cs.wikipedia.org/w/index.php?title=GPIB>

[26] PYTHON SOFTWARE FOUNDATION. *socket — Low-level networking interface*. [online]. 20. 4. 2012 [cit. 24. 4. 2012]. Dostupné na WWW: <docs.python.org/library/socket.html>

[27] HUANG, A. pybluez. [online]. 15. 10. 2009 [cit. 24. 4. 2012]. Dostupné na WWW: <code.google.com/p/pybluez>

[28] MEASUREMENT COMPUTING. *DaqBoard/1000 Series*. [online]. 9. 3. 2011 [cit. 24. 4. 2012]. Dostupné na WWW: <mccdaq.com/products/db1000s.htm>

[29] HAYES, N. *pydaqboard*. [online]. 11. 11. 2010 [cit. 24. 4. 2012]. Dostupné na WWW: <code.google.com/p/pydaqboard>

[30] LABJACK CORPORATION. *U3*. [online]. 28. 4. 2011 [cit. 24. 4. 2012]. Dostupné na WWW: <labjack.com/u3/specs>

[31] LABJACK CORPORATION. *LabJackPython*. [online]. 26. 8. 2011 [cit. 24. 4. 2012]. Dostupné na WWW: <labjack.com/support/labjackpython>

[32] PARADIS, E. *Using Python to Simulate Mechanical Things*. [online]. 22. 4. 2009 [cit. 28. 4. 2012]. Dostupné na WWW: <www.edparadis.com/pyode/>

[33] KING, W. T. *pypid 0.3, A modular PID control library*. [online]. 27. 7. 2011 [cit. 28. 4. 2012]. Dostupné na WWW: <pypi.python.org/pypi/pypid/>

[34] MURRAY, R. M. *Python Control Systems Library (python-control).* [online]. 7. 8. 2011 [cit. 28. 4. 2012]. Dostupné na WWW: <source-forge.net/apps/mediawiki/python-control/>

[35] CUTHBERT, D. *GUI Programming in Python*. [online]. 15. 4. 2012 [cit. 28. 4. 2012]. Dostupné na WWW: <wiki.python.org/moin/GuiProgramming>

[36] HULNE, S. *xRope: A lightweight IDE for Python*. [online]. 29. 7. 2008 [cit. 28. 4. 2012]. Dostupné na WWW: <sourceforge.net/projects/xrope/>

[37] UGARTE, J. P. *Glade - A User Interface Designer*. [online]. 26. 3. 2012 [cit. 28. 4. 2012]. Dostupné na WWW: <glade.gnome.org>

[38] HELD, A. *pyFLTK - Python wrapper for the Fast Light Tool Kit*. [online]. 19. 2. 2012 [cit. 29. 4. 2012]. Dostupné na WWW: <pyfltk.sourceforge.net>

[39] VIRBEL, M. *Kivy: Crossplatform Framework for NUI*. [online]. 2. 4. 2012 [cit. 29. 4. 2012]. Dostupné na WWW: <kivy.org/#gallery>

[40] BENDERSKY, E. *A "live" data monitor with Python, PyQt and PySerial*. [online]. 7. 8. 2009 [cit. 29. 4. 2012]. Dostupné na WWW: <eli.thegreenplace.net/2008/08/01/matplotlib-with-wxpython-guis/>

[41] MICROSOFT DEVELOPER NETWORK. *How To Use QueryPerformanceCounter to Time Code*. [online]. 20. 1. 2007 [cit. 22. 5. 2012]. Dostupné na WWW: <sup-port.microsoft.com/kb/172338/en-us?fr=1>

[42] SCIPY.ORG - COOKBOOK. *Animations*. [online]. 13. 5. 2012 [cit. 23. 5. 2012]. Dostupné na WWW: <www.scipy.org/Cookbook/Matplotlib/Animations>

[43] SMETANA, L. *Nelineární Neuro-regulátor pro úlohy automatického řízení* : diplomová práce. Praha : ČVUT, Fakulta Strojní, 2008. 63l., 1 l. příl. Vedoucí diplomové práce Ing. Ivo Bukovský, Ph.D.

7. PŘÍLOHY

… NA VYŽÁDÁNÍ