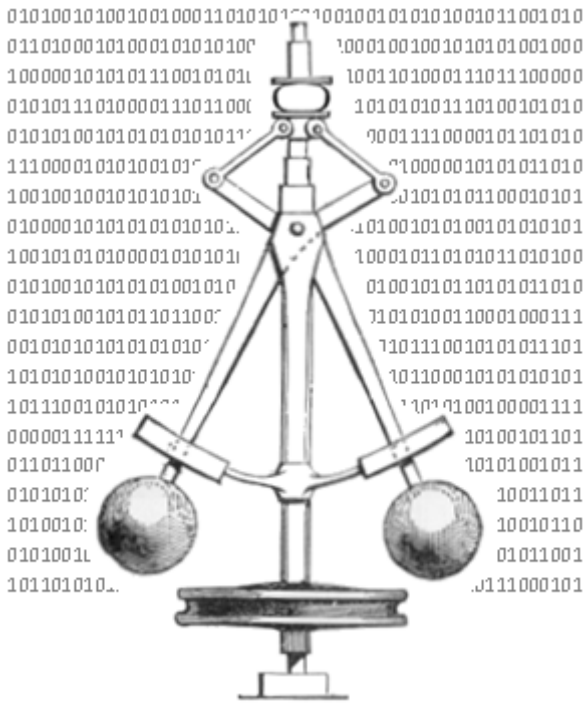
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA ELEKTROTECHNICKÁ

Katedra řízení

Bakalářská práce



Podpora návrhu číslicových regulátorů

Jan Martinec

Ing. Ondřej Holub, vedoucí práce

Praha, 2008

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra řídicí techniky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: **Jan Martinec**

Studijní program: Elektrotechnika a informatika (bakalářský), strukturovaný
Obor: Kybernetika a měření

Název tématu: **Podpora návrhu číslicových regulátorů**

Pokyny pro vypracování:

1. Napište obecný algoritmus pro zpětnovazební řízení tak, aby bylo možno do něj snadno zadávat navržené parametry pro konkrétní úlohy řízení. Použijte programovací jazyk C. Vytvořte dvě verze kódu, jednak s plovoucí řádovou čárkou, jednak s pevnou řádovou čárkou.
2. Zopakujte předchozí postup pro dodané algoritmy diskrétního PID regulátoru a Kalmanova filtru.
3. Ověřte správnost návrhu simulacemi v prostředí Matlab (Simulink) a zejména implementací navržených algoritmů do desky s mikroprocesorem LPC firmy Philips.

Seznam odborné literatury:

Dodá vedoucí práce

Vedoucí: Ing. Ondřej Holub

Platnost zadání: do konce zimního semestru 2008/2009

L.S.

prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

v.z. M. Šimková
doc. Ing. Boris Šimák, CSc.
děkan

V Praze dne 25. 2. 2008

Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

V Praze, dne 18. 8. 2008

podpis

Poděkování

Chtěl bych tímto předně poděkovat vedoucímu bakalářské práce panu Ing. Ondřeji Holubovi za jeho kvalitní vedení a odbornou pomoc. Mé poděkování také náleží panu Ing. Ondřeji Špinkovi a všem co mi při práci pomáhali, ať už přímo nebo nepřímo.

Abstrakt

Tato práce se zabývá implementací několika druhů číslicových regulátorů a Kalmanova filtru v programovacím jazyce C a to jak v klasickém binární reprezentaci čísel formátu plovoucí řádové čárky, tak i ve formátu pevné řádové čárky. Významná pozornost je věnována právě formátu pevné řádové čárky z důvodu jeho nepříliš velkého rozšíření v aplikacích číslicové techniky a implementaci knihovny jazyka C pro práci s aritmetikou tohoto formátu. Cílem práce je navrhnout číslicové algoritmy pro obecně použitelnou formu regulátoru se dvěma stupni volnosti a filtrací vstupu, modifikovanou verzi PID regulátoru a obecného Kalmanova filtru a především pak ukázat vliv použití formátu pevné řádové čárky na průběhy regulace a filtrace. K účelům testování navržených algoritmů jsou kromě jiných prostředků použity tzv. systémové funkce poskytované programem Matlab, jejichž problematice je věnována také podstatná část této práce.

Abstract

This thesis describes an implementation of several kinds of digital controllers and Kalman filter in numeric binary representation of floating point and fixed point data format via programming language C. The main attention is paid to the fixed point number representation by reason of not very wide distribution in digital application and to implementation of a C language library for fixed point arithmetic computation. The goal of this thesis is designing of algorithms for widely usable form of controller with two degrees of freedom and input filtering, modified version of PID controller and common Kalman filter and mainly showing the influence of using fixed point format on controlling and filtering. Among other instruments are for testing of designed algorithms used so-called system functions provided by Matlab, which are also described in this thesis.

Obsah

1. ÚVOD 1
2. REPREZENTACE ČÍSEL V POČÍTAČI 2
   - 2.1 FORMÁT S PLOVOUCÍ ŘÁDOVOU ČÁRKOU 3
   - 2.2 FORMÁT S PEVNOU ŘÁDOVOU ČÁRKOU 3
     - *2.2.1 Základní aritmetické operace ve formátu pevné řádové čárky* 5
     - *2.2.2 Implementace a popis knihovny fixedpoint* 7
     - *2.2.3 Testování knihovny fixedpoint* 11
3. MATLAB A S-FUNKCE 17
   - 3.1 PRINCIP C MEX S-FUNKCÍ 17
   - 3.2 METODY IMPLEMENTACE C MEX S-FUNKCÍ 20
     - *3.2.1 Ručně psané C MEX S-funkce* 20
     - *3.2.2 S-function Builder* 21
   - 3.3 KOMPILACE A LADĚNÍ C MEX S-FUNKCÍ 23
     - *3.3.1 Kompilace C MEX S-funkcí* 23
     - *3.3.2 Ladění C MEX S-funkcí* 24
   - 3.4 POUŽÍVÁNÍ C MEX S-FUNKCÍ A JEJICH MASKOVÁNÍ 25
4. OBECNÝ REGULÁTOR 27
   - 4.1 STRUKTURA OBECNÉHO REGULÁTORU 28
     - *4.1.1 Číslicové FIR filtry* 28
     - *4.1.2 Číslicové IIR filtry* 29
   - 4.2 IMPLEMENTACE OBECNÉHO REGULÁTORU 30
     - *4.2.1 Implementace ve formátu plovoucí řádové čárky* 30
     - *4.2.2 Implementace ve formátu pevné řádové čárky* 33
   - 4.3 POUŽITÍ A TESTOVÁNÍ OBECNÉHO REGULÁTORU 34
     - *4.3.1 S-funkce obecného regulátoru* 36

5. PID REGULÁTOR 39
5.1 STRUKTURA POUŽITÉHO REGULÁTORU 39
5.2 ALGORITMUS PID REGULÁTORU A JEHO ÚPRAVY 43
5.2.1 Popis knihovny regulátoru v FP formátu a její úpravy 43
5.2.2 Přepis knihovny PID do FX formátu 46
5.3 OVĚŘENÍ FUNKCE A TESTOVÁNÍ KNIHOVEN PID A FIXPID 46
5.3.1 Porovnání funkce knihoven PID a fixPID se skutečnými daty 47
5.3.2 S-funkce PID regulátoru 50
6. KALMANŮV FILTR 52
6.1 POPIS FUNKCE KALMANOVA FILTRU 53
6.2 IMPLEMENTACE KALMANOVA FILTRU 55
6.2.1 Implementace ve formátu plovoucí řádové čárky 55
6.2.2 Implementace ve formátu pevné řádové čárky 58
6.3 OVĚŘENÍ A TESTOVÁNÍ FUNKCE KNIHOVEN KALMAN A FIXKALMAN 58
6.3.1 S-funkce Kalmanova filtru 62
7. ZÁVĚR 63
LITERATURA 65
SOFTWARE 67

PŘÍLOHY

A OBSAH PŘILOŽENÉHO CD I

Seznam obrázků

Obrázek 2.1: *Rozdělení binárního slova u formátu pevné řádové čárky* 4
Obrázek 2.2: *Příklad rozsahu a přesnosti hodnot u formátu pevné řádové čárky při délce bitového slova N = 8 a A=5, B=3* 4
Obrázek 2.3: *Znázornění aritmetické operace součinu u formátu s pevnou řádovou čárkou* 6
Obrázek 2.4: *Ukázka použití masky pro detekci přetečení* 10
Obrázek 2.5: *Ukázka textového výstupu programu Řády čísel pro testování knihovny fixedpoint* 12
Obrázek 2.6: *Graficky znázorněný výstup programu Řády čísel pomocí Matlab skriptu view_results.m při nastavení FX formátu: A = 16 a B = 16* 13
Obrázek 2.7: *Nákres protínajících se dvou kružnic* 14
Obrázek 2.8: *Graficky znázorněný výstup programu Kružnice pomocí Matlab skriptu circles_view_results.m* 15
Obrázek 2.9: *Ukázka textového výstupu programu Kružnice pro testování knihovny fixedpoint* 16

Obrázek 3.1: *Diagram simulační smyčky S-funkce* 18
Obrázek 3.2: *Blok Inicializace a jeho vnitřní metody pro C MEX S-funkce*[1] 19
Obrázek 3.3: *Dialogové okno S-function Builder* 22
Obrázek 3.4: *Postup pro připojení k procesu Matlab v prostředí Microsoft Visual Studio* 24
Obrázek 3.5: *Dialogové okno pro nastavení parametrů S-function bloku v Simulinku* 25
Obrázek 3.6: *Editor pro tvorbu masky S-function bloku* 26

[1] převzato z [5] *Simulink 7: Writing S-functions. Documentation for MathWorks Products*

Obrázek 4.1: *Struktura obecného regulátoru* 28
Obrázek 4.2: *Porovnání výsledků regulace obecného regulátoru se simulací Matlabu pomocí skriptu compare_PSD.m* 35
Obrázek 4.3: *Výsledky testu funkčnosti knihoven system a filter s porovnáním se simulací v Matlabu pomocí skriptu compare_systems.m* 36
Obrázek 4.4: *Schéma Simulink modelu pro testování obecného regulátoru s použitím S-funkcí* 37
Obrázek 4.5: *Maska S-funkce obecného regulátoru pracujícího ve formátu pevné řádové čárky* 38

Obrázek 5.1: *Regulační obvod s PID regulátorem* 40
Obrázek 5.2: *Regulační obvod s PID regulátorem s filtrovanou derivační složkou ve zpětné vazbě* 41
Obrázek 5.3: *PID regulátor s „vážením reference“* 42
Obrázek 5.4: *Vliv parametru b na odezvu PID regulátoru s „vážením reference“*[2] 42
Obrázek 5.5: *Stavový model PID regulátoru s derivační složkou ve zpětné vazbě a „vážením reference“* 43
Obrázek 5.6: *Porovnání práce regulátorů z knihovny PID a fixPID pro nastavení FX formátu A=12, B=20* 48
Obrázek 5.7: *Porovnání práce regulátorů z knihovny PID a fixPID pro nastavení FX formátu A=16, B=16* 49
Obrázek 5.8: *Porovnání práce regulátorů z knihovny PID a fixPID pro nastavení FX formátu A=20, B=12* 49
Obrázek 5.9: *Schéma Simulink modelu pro testování PID regulátorů s použitím S-funkcí* 51
Obrázek 5.10: *Maska S-funkce realizující PID regulátor* 51

Obrázek 6.1: *Stavový model obecného diskrétního systému* 53
Obrázek 6.2: *Stavové schéma Kalmanova filtru a jeho připojení k pozorované soustavě* 54
Obrázek 6.3: *Filtrace zarušeného signálu Kalmanovým filtrem v FP a FX formátu s nastavením A=16,B=16* 60
Obrázek 6.4: *Filtrace zarušeného signálu Kalmanovým filtrem v FP a FX formátu s nastavením A=12,B=20* 60
Obrázek 6.5: *Filtrace zarušeného signálu Kalmanovým filtrem v FP a FX formátu s nastavením A=20,B=12* 61
Obrázek 6.6: *Filtrace zarušeného signálu Kalmanovým filtrem v FP a FX formátu s nastavením A=22,B=10* 61
Obrázek 6.7: *Ohlášená chyba po spuštění S-funkce Kalmanova filtru v Simulinku* 62

[2] převzato z [9] *Computer-Controlled Systems : Theory and Design*

Kapitola 1

Úvod

Regulace a řízení je v dnešní době jedním z nejdůležitějších a nejperspektivnějších technických oborů v oblasti průmyslu i vědeckém bádání. Po masivním nástupu počítačů a číslicové techniky se postupem času začalo i v tomto odvětví přecházet od analogové reprezentace regulátorů k jejich ekvivalentní číslicové podobě. Číslicové regulátory, jejichž podob a modifikací je v dnešní době opravdu velké množství, jsou nejpoužívanějšími systémy pro úlohy řízení.

Tato práce se zabývá implementací algoritmů několika druhů regulátorů a Kalmanova filtru v programovacím jazyce C, a to jak v klasickém číselném formátu plovoucí řádové čárky, tak i v praxi méně používaném formátu pevné řádové čárky. I přesto, že použití formátu pevné řádové čárky s sebou nese řadu výhod, jeho podpora není ze strany výrobců číslicových systémů příliš rozšířena. Je tomu tak z toho důvodu, že realizace specializované výpočetní jednotky není jednoduchou a levnou záležitostí. Na druhou stranu, funkci aritmetiky formátu pevné řádové čárky je možné relativně jednoduchým způsobem implementovat v některém z programovacích jazyků. Proto jedním z pilířů této práce je realizace speciální knihovny poskytující možnosti používání tohoto formátu při implementaci programů v jazyce C. Použití této knihovny přinese možnost tvorby algoritmů pro jednoúčelové mikroprocesorové systémy, jejichž cena i složitost je výrazně nižší oproti použití klasické binární reprezentace čísel ve formátu plovoucí řádové čárky.

Součástí zadání je i podmínka možnosti jednoduchého testování navržených algoritmů číslicových regulátorů. Proto jsou všechny knihovny vytvořeny ve formě použitelné pro počítačové testovací programy. K testování jsem se rozhodl používat i tzv. C MEX systémové funkce, které poskytují možnost používat programy implementované v programovacím jazyce C přímo v modelačním prostředí Matlabu, tedy Simulinku. Vysvětlení principu funkce a způsobu realizace systémových funkcí věnuji v této práci také výraznou pozornost.

Kapitola 2

Reprezentace čísel v počítači

V této kapitole se pokusím vysvětlit, jakými způsoby lze v operačních pamětech počítače a v registrech mikroprocesoru (*CPU*) či matematického koprocesoru (*FPU*) reprezentovat numerické hodnoty. Číselné hodnoty se v počítačích ukládají v binárním formátu, to znamená, že pro vyjádření nějakého stavu z reálného světa, v našem případě tedy čísel, se použije N binárních číslic (bitů), což je základní jednotka informace nabývající hodnot *0* nebo *1* (*false* nebo *true*). Pomocí N-tice bitů je tedy možné vyjádřit 2^N jednoznačných stavů (číslic). Je pak jen na nás nebo na programátorovi, jakým způsobem bude dané stavy interpretovat.

Nejjednodušší interpretací binárního čísla je jeho zobrazení do množiny celých čísel (*Integers*), které můžeme vyjadřovat přímo binární kombinací bitů. Číselné hodnoty reálných čísel je v dnešní době možné ukládat nejčastěji ve dvou binárních formátech, a to v *pevné řádové čárce* a dnes asi v nejpoužívanějším formátu *plovoucí řádové čárky*. V anglické literatuře se formát pevné řádové čárky označuje zkratkou *FX* nebo *FXP* (*fixed point*) a formát plovoucí řádové čárky zkratkou *FP* (*floating point*). Oba formáty se liší jak ve způsobu binární reprezentace čísla, tak i v aritmetických operacích prováděných *CPU* nebo *FPU*.

Nejdůležitějším problémem, kterému musejí programátoři čelit při volbě binárního formátu, je efektivita využití všech dostupných bitů určených pro vyjádření čísla. Je to způsobeno tím, že počítače mají vždy omezený počet bitů, tedy i omezený rozsah hodnot, které lze binárně interpretovat, a také tím, že procesory jsou navrhovány tak, aby vždy pracovali s konstantním počtem bitů. Bude-li například procesor s *32*bitovou architekturou pracovat s čísly uloženými na *20* bitech, efektivita se výrazně snižuje. Příkladem můžou být například dnes hojně používané mikroprocesory řady *x86*.

2.1 Formát s plovoucí řádovou čárkou

Tento formát je dnes nejpoužívanějším formátem pro ukládání reálných čísel. Základní myšlenkou tohoto formátu je to, že každá uložená hodnota si s sebou nese informaci o poloze řádové čárky. Řádová čárka tedy není určena pevně, ale může se její poloha měnit – „plavat“ – od toho tedy název formát s plovoucí řádovou čárkou. Toto je základní rozdíl od formátu s pevnou řádovou čárkou.

Numerická hodnota se v binární reprezentaci skládá z několika částí. První z nich se nazývá *mantisa*, druhá část je *exponent*. *Mantisa* nese informaci o významných číslicích ukládané numerické hodnoty, *exponent* pak udává mocninu o určitém základu (bázi), kterou jsou významné číslice násobeny nebo děleny. Původní hodnota reálného čísla se získá podle (2.1). Největší výhodou a důvodem širokého používání tohoto formátu je široký rozsah číselných hodnot, které je možné zobrazit.

$$v = b^e \times m \tag{2.1}$$

kde

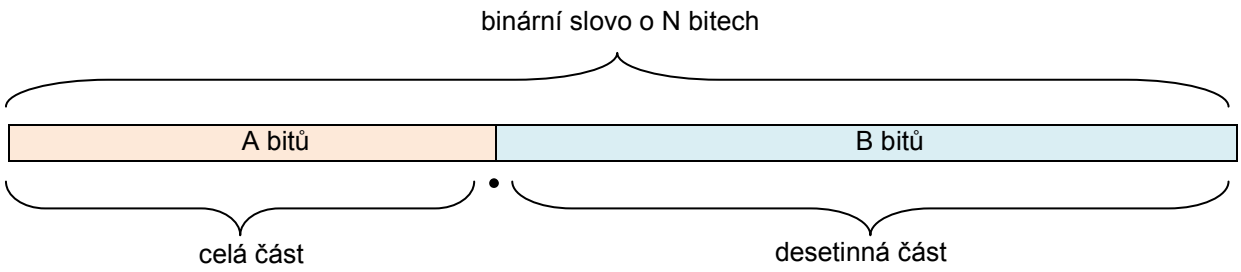
- v je hodnota reálného čísla
- b je báze
- e je exponent
- m je mantisa

Podrobněji se o formátu s plovoucí řádovou čárkou v tomto textu zmiňovat nebudu, jelikož to není hlavním předmětem této práce. Podrobnější informace mohou zájemci nalézt v normě IEEE 754, což je standard popisující binární aritmetiku *FP* formátu.

2.2 Formát s pevnou řádovou čárkou

Této kapitole se budu věnovat výrazně hlouběji, neboť celá má práce stojí právě na použití formátu s pevnou řádovou čárkou. Zaměřím se na vysvětlení principu uchovávání číselných dat v tomto formátu, funkci aritmetických operací s těmito daty a celkovou implementaci knihovny v programovacím jazyce C pro práci s *FX* formátem.

Jak již bylo popsáno v předchozím textu, formát *FX* je vhodný pro binární reprezentaci reálných čísel a od formátu *FP* se liší hlavně tím, že má přesně definovanou polohu řádové čárky, která je pro všechna čísla stejná. Binární slovo o *N* bitech je rozděleno opět na dvě části. První část o *A* bitech obsahuje informaci o celé části reálného čísla. Druhá část, která má *B* bitů, pak reprezentuje desetinnou část čísla (obr. 2.1).



Obrázek 2.1: Rozdělení binárního slova u formátu pevné řádové čárky

Změnou počtu bitů *A* a *B* je možno měnit rozsahy a přesnost zobrazovaných čísel. Zvolení správných hodnot je velice důležité, aby nedocházelo k výpočetním chybám. Každý, kdo by chtěl používat tento číselný formát, by si měl dopředu velice dobře rozmyslet, jaké přesnosti a rozsahu mají jeho číselné hodnoty dosahovat. Každý bit celé části představuje kladnou mocninu o základu *2*, lze tedy pomocí *A* bitů měnit rozsah čísel. Naopak každý bit desetinné části představuje zápornou mocninu o základu *2* a je tedy možno pomocí *B* bitů měnit přesnost. Situaci je lépe vidět na příkladě s *8*bitovou délkou slova na (obr. 2.2), kde poloha řádové čárky je mezi třetím a čtvrtým bitem.

Rozsah zobrazitelných čísel je dán intervalem podle (2.2) pro čísla se znamínkem, nebo podle (2.3) pro čísla bez znaménka. Mezi oběma formáty lze přecházet pomocí jednoduchých operací. Pro převod z *FX* formátu do *FP* formátu platí vztah (2.4) a pro opačný převod pak analogicky vztah (2.5).

bit	8	7	6	5	4	3	2	1
váha bitu	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}
desítková váha bitu	16	8	4	2	1	0.5	0.25	0.125

Obrázek 2.2: Příklad rozsahu a přesnosti hodnot u formátu pevné řádové čárky při délce bitového slova $N = 8$ a A=5, B=3

$$-2^{A-1} < X_{signed} < 2^{A-1} - 2^{-B} \tag{2.2}$$

$$0 < X_{unsigned} < 2^{A} - 2^{-B} \tag{2.3}$$

$$\mathrm{X_{FP}} = \frac{\mathrm{X_{FX}}}{2^{\mathrm{B}}} \tag{2.4}$$

$$\mathrm{X_{FX}} = \mathrm{X_{FP}} \times 2^{\mathrm{B}} \tag{2.5}$$

Výhodou oproti *FP* formátu je to, že poloha řádové čárky je známa již před použitím algoritmu, který s čísly pracuje. To znamená, že tak uspoříme velkou část bitového slova, která je u *FP* formátu vyhrazena pro uchování informace o poloze řádové čárky. Výhodné je nasazení tohoto formátu u *embedded* mikrokontrolerů, které neobsahují matematický koprocesor (*FPU*). Použitím *FX* formátu můžeme také zajistit, že algoritmus bude pracovat vždy s čísly se stejnou přesností, což je také u mnoha aplikací velice důležité.

Výraznou nevýhodou formátu pevné řádové čárky je jeho nízká podpora ze strany výrobců mikroprocesorů a jiných číslicových systémů, jelikož vytvoření specializovaného matematického koprocesoru pracujícího v *FX* formátu je velice komplikované. Existuje jen několik specializovaných zařízení, které obsahují kromě klasické matematické jednotky *FPU* také jednotku *FXU*, která umožňuje provádět výpočty v tomto formátu. Také u programovacích jazyků, jako je například C nebo C++, se nedostává tomuto formátu veliké podpory.

2.2.1 Základní aritmetické operace ve formátu pevné řádové čárky

V následujícím textu popíši principy funkce základních aritmetických i jiných operací pro práci s formátem pevné řádové čárky, zejména těch, které jsem později implementoval v programovacím jazyce C.

- *Převody z FX formátu do FP a naopak*

Tyto operace samozřejmě nepatří do základních číselných aritmetických operací, ale zmiňuji je zde, jelikož v mnou vytvořené knihovně pro práci s *FX* formátem jsou velice důležité. Vztahy pro převod mezi oběma formáty jsem již zmiňoval v předchozí části kapitoly. Pro převod z *FX* do *FP* platí vztah (2.4), pro opačný převod pak vztah (2.5).

- *Aritmetické operace sčítaní a odčítání*

Popis těchto dvou aritmetických operací jsem spojil dohromady, jelikož pro obě platí stejná pravidla. Abychom mohli sčítat (odčítat) dvě čísla ve formátu pevné řádové čárky, musíme zajistit splnění podmínky, že se budou sčítat (odčítat) jen čísla ve stejném *FX* formátu, to znamená se stejným nastavením počtu bitů celé (*A*) a desetinné části (*B*). Pak lze aritmetickou operaci sčítání vyjádřit podle (2.6). Analogicky pak i operaci odčítání podle (2.7). Tyto vztahy ukazují, že sčítání (odčítání) dvou hodnot v *FX* formátu je stejné, jako kdybychom sčítali (odčítali) dvě celočíselné hodnoty *integer*. Lze tak využít instrukcí aritmetickologické jednotky mikroprocesoru (*ALU*).

$$X \times 2^B + Y \times 2^B = (X + Y) \times 2^B \tag{2.6}$$

$$X \times 2^B - Y \times 2^B = (X - Y) \times 2^B \tag{2.7}$$

- *Aritmetická operace násobení*

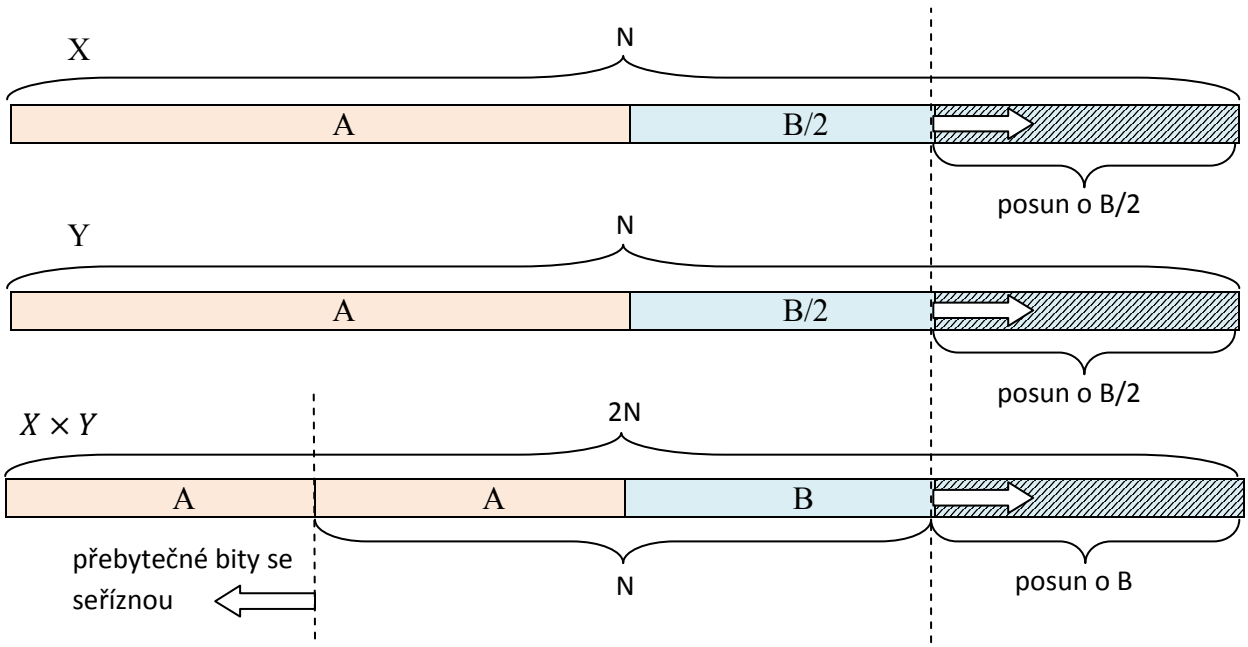
Implementace této operace není tak jednoduchá, jako tomu bylo u sčítání a odčítání, i přesto, že zde nemusíme splňovat podmínku stejného *FX* formátu obou činitelů. Pokud totiž násobíme dvě *N* bitová binární čísla, pak výsledek má *2N* bitů. Pokud tedy budeme násobit dvě *32*bitová čísla, dostaneme *64*bitový výsledek. Toto tvrzení zapíšeme jako (2.8). Takováto situace je ve většině případů problematická. Problém lze řešit tak, že zmenšíme přesnost obou činitelů. V řeči *FX* formátu to znamená, že zmenšíme velikost desetinné části (*B*) obou činitelů a celé číslo bitově posuneme o zmenšenou desetinnou část směrem doprava.

Předpokládejme, že pracujeme vždy s čísly, které jsou vzájemně ve stejném *FX* formátu. Pak bitový posun směrem doprava provedeme o *B/2* bitů. Tím zmenšíme logickou velikost obou činitelů, tedy i velikost výsledku jejich součinu, ale pouze na úkor přesnosti výpočtu. Přebytečné bity v *A* části výsledku se nakonec seříznou do konečné velikosti *FX* formátu. Řešení je graficky znázorněno na (obr. 2.3). Poznamenejme, že u tohoto formátu bude velice často docházet k přetečení, respektive podtečení, což je překročení maximální délky bitového slova *N*. Tomu je při implementaci této aritmetické operace potřeba věnovat pozornost.

pro čísla bez znaménka: $X(A1, B1) \times Y(A2, B2) = Z(A1 + A2, B1 + B2)$

pro čísla se znaménkem: $X(A1, B1) \times Y(A2, B2) = Z(A1 + A2 + 1, B1 + B2)$

(2.8)



Obrázek 2.3: Znázornění aritmetické operace součinu u formátu s pevnou řádovou čárkou

- *Aritmetická operace dělení*

U aritmetické operace dělení je velikost výsledku dána vztahy (2.9). Dělení má podobné vlastnosti jako předchozí operace násobení. Dělit můžeme opět čísla s různým *FX* formátem, tedy s různým poměrem celé a desetinné části. Ze vztahů (2.9) je patrné, že i u této operace se budeme potýkat s příliš velkou velikostí výsledku stejně tak jako tomu bylo u součinu. V tomto případě je možné problém řešit tak, že děliteli opět zmenšíme přesnost bitovou rotací směrem doprava o *B/2* bitů. Tím sice ztratíme přesnost čísla, ale zamezíme tak častému přetečení (podtečení) výsledku. To ale neznamená, že k přetečení (podtečení) nemůže dojít, a tak je nezbytné jeho detekci implementovat.

pro čísla bez znaménka: $\frac{X(A1,B1)}{Y(A2,B2)} = Z(A1 + B1, |\log_2(2^{A2+B1} - 2^{B1-B2})|)$

pro čísla se znaménkem: $\frac{X(A1,B1)}{Y(A2,B2)} = Z(A1 + B2 + 1, A2 + B1)$

(2.9)

2.2.2 Implementace a popis knihovny fixedpoint

Implementování knihovny v programovacím jazyce C, která bude zajišťovat aritmetiku formátu pevné řádové čárky, a její použití v řídících algoritmech číslicových regulátorů je hlavním pilířem této práce.. V následující části kapitoly popíši jednotlivé funkce knihovny.

```
int fixInit(int a, int b, int report)
```

Tato funkce inicializuje celou knihovnu *fixedpoint*. Argumenty funkce *a*, *b* typu *integer* nastavují formát pevné řádové čárky - počet bitů celé a desetinné části (*A* a *B*). Argument *report*, který je také typu *integer*, definuje, zda se mají při práci knihovny vypisovat chybová hlášení. Pokud je jeho hodnota rovna *0*, pak se chybová hlášení vypisovat nebudou. U funkce se kontroluje pouze součet argumentů *a* a *b*, který musí být roven celkové velikosti bitového slova, v tomto případě je to konstanta *MAX_BITS*. Pokud je tato podmínka splněna, provede se nastavení *FX* formátu a funkce skončí s návratovou hodnotou *0*. V opačném případě funkce vrátí hodnotu *-1*.

```
fixp float2fix(double a)
```

Funkce *float2fix* převádí číselnou hodnotu uloženou ve formátu plovoucí řádové čárky ve vstupní proměnné *a*, do formátu pevné řádové čárky. Návratová hodnota této funkce je převedené číslo datového typu *fixp* (definovaný jako *signed integer*). Před samotným převodem se zkontroluje, zda hodnota reprezentovaná v *FP* formátu leží v zobrazitelném rozsahu nastaveného *FX* formátu daného intervalem (2.2). Je-li toto číslo větší než je horní

mez intervalu (2.2), funkce vrátí právě hodnotu této meze. Analogicky pak platí to samé i pro dolní mez. Leží-li číslo v rozsahu zobrazitelných hodnot, provede se převod podle (2.5), jen s tím rozdílem, že místo mocniny 2^B se použije ekvivalentní operace bitové rotace čísla *1* o *B* bitů vlevo.

```
double fix2float(fixp a)
```

Tato funkce je inverzní funkcí k předchozí. To znamená, že převádí číslo v *FX* do *FP* formátu. U funkce není třeba žádné kontroly a vstupní argument *a* typu *fixp* je převeden na datový typ *double* podle vztahu (2.4), kdy je opět místo mocniny použita bitová rotace.

```
void printfix_bin(fixp a)
void printfix_dec(fixp a)
```

Tyto funkce jsou pouze pomocné, které jsem používal při odlaďování knihovny *fixedpoint*. V knihovně jsem je zanechal z důvodu jejich možného využití uživatelem. První z nich, *printfix_bin*, vypíše hodnotu čísla v *FX* formátu na výstupní terminál v binárním formátu se zobrazenou polohou řádové čárky. Druhá funkce pak provádí podobnou operaci s tím rozdílem, že na výstupním terminálu je číslo zobrazeno v dekadickém formátu.

```
fixp addfix(fixp a, fixp b)
```

Je první z funkcí realizujících aritmetickou operaci. V tomto případě se jedná o sčítání. Někdo by mohl namítat, proč realizuji ve své knihovně zvláštní funkci pro sčítání dvou čísel v *FX* formátu, když v předchozí části kapitoly bylo řečeno, že pro součet je možné použít klasickou sčítací instrukci nabízenou aritmetickologickou jednotkou procesoru (*ALU*). Je to z toho důvodu, že je potřebné detekovat přetečení hodnoty součtu a také proto, aby se dodržela určitá forma celé knihovny. Vstupem do funkce jsou dva sčítance *a* a *b* datového typu *fixp*. Ještě před provedením součtu a vrácení výsledku je nutné zjistit, zda u výsledku nedojde k přetečení nebo podtečení *FX* formátu. K přetečení horní meze může dojít pouze v případě, že oba sčítance jsou kladná čísla a platí-li nerovnice (2.10). Opačně je to u podtečení, ke kterému dojde, jsou-li sčítance záporné a platí (2.11).

$$a > 0 \;\; AND \;\; b > 0 \;\; AND \;\; (FIXP\_MAX - a) < b \tag{2.10}$$

$$a < 0 \;\; AND \;\; b < 0 \;\; AND \;\; (FIXP\_MIN - a) > b \tag{2.11}$$

```
fixp subfix(fixp a, fixp b)
```

Jedná se o realizaci aritmetické operace rozdílu. Funkce je téměř totožná s předchozí funkcí *addfix*, pouze s tím rozdílem, že místo znaménka + je použito znaménko – a podmínky pro přetečení se změní z (2.10) na (2.12), respektive pro případ podtečení z (2.11) na (2.13).

$$a > 0 \;\; AND \;\; b < 0 \;\; AND \;\; (FIXP\_MAX - a) < b \tag{2.12}$$

$$a < 0 \;\; AND \;\; b > 0 \;\; AND \;\; (FIXP\_MIN - a) > b \tag{2.13}$$

```
fixp mulfix(fixp a, fixp b)
```

Jak už napovídá název funkce, jedná se o aritmetickou operaci součinu. Vychází se z metody o součinu dvou čísel v *FX* formátu popsané v předchozí podkapitole. Hodnotám dvou činitelů *a* a *b*, které jsou vstupními parametry funkce datového typu *fixp*, se nejprve sníží jejich přesnost bitovou rotací o *B/2* bitů směrem doprava. Tím tedy částečně omezíme celkovou logickou velikost výsledku součinu. Dalším krokem je kontrola přetečení. Tu opět musíme rozdělit na dvě části, a to na kontrolu přetečení horní meze a kontrolu přetečení dolní meze. Horní mez *FX* formátu výsledek překročí pouze v případě, že oba činitele mají stejná znaménka, podmínka je pak dána podle (2.14). V opačném případě hrozí přetečení meze dolní a kontroluje se podmínkou (2.15). Pokud podmínky splněny nejsou, provede se součin a funkce vrátí jeho výsledek. Při přetečení se vrací příslušné mezní hodnoty, tedy maxima nebo minima *FX* formátu.

$$\left(a > 0 \;\; AND \;\; b > 0 \;\; AND \;\; \frac{FIX\_MAX}{a} < b\right) OR \left(a < 0 \;\; AND \;\; b < 0 \;\; AND \;\; \frac{FIXP\_MAX}{a} > b\right) \tag{2.14}$$

$$\left(a < 0 \;\; AND \;\; b > 0 \;\; AND \;\; \frac{FIXP\_MIN + 1}{a} < b\right) OR \left(a > 0 \;\; AND \;\; b < 0 \;\; AND \;\; \frac{FIXP\_MIN + 1}{b} < a\right) \tag{2.15}$$

```
fixp divfix(fixp a, fixp b)
```

Poslední z výčtu základních aritmetických operací, které jsou potřebné pro běžné výpočetní operace, je dělení. U této operace se potýkáme v podstatě se stejnými problémy jako u násobení. Pro zamezení častého přetékání hodnot se děliteli *b* opět sníží přesnost bitovou rotací směrem doprava o *B/2* bitů. Následně se provede kontrola, zdali takto upravený dělitel *b_new* není roven nule. Pokud ano, v závislosti na znamínku dělence *a* je jako výsledek vrácena hodnota maxima nebo minima *FX* formátu. Je to z toho důvodu, že při dělení nulou se výsledek limitně blíží k nekonečnu. V našem případě je nekonečno možné vyjádřit právě maximálními zobrazitelnými hodnotami *FX* formátu. Vzniká tak při výpočtu výrazná chyba, ale je tím alespoň zajištěna jakási kontinuita složitějšího výpočtu. Pokud je upravený dělitel různý od nuly, provede se klasické dělení *a / b_new*.

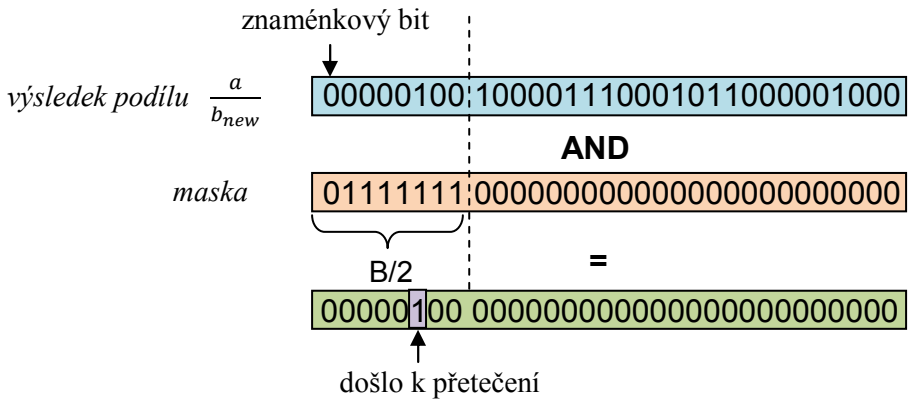
Nyní nastává čas pro kontrolu přetečení. Ta je výrazně složitější, než tomu dosud bylo u ostatních aritmetických operací. První operací je kontrola znaménka výsledku. Pokud je výsledek záporný a přitom oba s operandů měli shodné znaménko, je to jasná známka toho, že došlo k přetečení. Podmínka se tedy zapíše podle (2.16). Pro opačný případ, kdy při kladném znamínku výsledku a různých znamínkách operandů můžeme detekovat podtečení, platí podmínka (2.17).

Jelikož bity dělitele jsme před dělením rotovali, došlo i ke změně polohy řádové čárky u výsledku. Pro získání správné polohy, musíme výsledek podílu rotovat o stejnou vzdálenost, tedy o *B/2* bitů, tentokrát ale směrem doleva. Toto je druhý případ, kdy musíme detekovat možnost vzniku přetečení (podtečení) *FX* formátu. Rotací se ztratí horních *B/2* bitů reprezentovaného čísla. Pokud těchto *B/2* bitů nese nějakou informaci o výsledku (bity nejsou nulové), znamená to, že upravený výsledek pak tedy bude nesprávný. K detekci jsem použil bitovou masku o stejné délce bitového slova, jakou má výsledek podílu. Maska má horních *B/2* bitů, kromě nejvyššího znaménkového, jednotkových. Zbytek bitů je nulový. Logickým součinem výsledku podílu *a / b_new* a masky získáme nulový nebo nenulový výsledek. Pokud součinem dostaneme nulu, znamená to, že rotací výsledku podílu neztratíme žádnou informaci a můžeme ji tedy provést. V opačném případě dojde rotací k přetečení. Tento postup platí ale pouze pro nezáporné výsledky podílu. V případě, že je výsledek dělení záporné číslo, jeho binární reprezentace je ve formě doplňku. To znamená, že k přetečení, respektive tedy v tomto případě podtečení, dojde v chvíli, kdy horních *B/2* bitů bude obsahovat alespoň jednu nulu. V tom je tedy rozdíl v detekci oproti případu kladného výsledku, kdy je nutné hlídat výskyt jedniček. K detekci nul v horních *B/2* bitech je možné použít naprosto stejného způsobu maskování, jako v předchozím případě, jen s tím rozdílem, že všechny bity maskovaného výsledku podílu je nutné znegovat.

Ukázka použití masky pro *32*bitovou reprezentaci *FX* formátu s nastavením *A=16* a *B=16* a případ kladného výsledku podílu *a / b_new* je na (obr. 2.4)

$$(res < 0)\ AND\ (\ (a > 0\ AND\ b > 0)\ OR\ (a < 0\ AND\ b < 0)\ ) \tag{2.16}$$

$$(res > 0)\ AND\ (\ (a > 0\ AND\ b < 0)\ OR\ (a < 0\ AND\ b > 0)\ ) \tag{2.17}$$



Obrázek 2.4: Ukázka použití masky pro detekci přetečení

2.2.3 Testování knihovny fixedpoint

Knihovnu *fixedpoint* bylo před nasazením do algoritmů číslicových regulátorů nutné otestovat. Tím se myslí ověření jejího chování a přesnosti výsledků pro složitější výpočty a škálování. Pod pojmem „škálování“ rozumějme nastavení různých poměrů logické velikosti celé a desetinné části (A a B) formátu pevné řádové čárky.

Pro testovací účely jsem vytvořil následující dva programy:

- *Řády čísel*

Tento program testuje cyklicky všechny aritmetické operace definované v knihovně *fixedpoint*. Cyklů je celkem dvanáct a v každém z nich se vygenerují dvě náhodná čísla v číselném řádu od 10^{-6} do 10^{5}. Řád vždy odpovídá aktuálnímu cyklu. S vygenerovanými čísly se poté provádí matematické úkony sčítání, odčítání, násobení a dělení ve formátu pevné řádové čárky. Všechny tyto matematické operace se navíc provádí pro všechny kombinace znamének obou čísel. V jednom cyklu se tedy provede celkem *16* výpočtů, jejichž výsledky se porovnávají s výsledky vypočtenými klasicky ve formátu plovoucí řádové čárky formou relativní odchylky.

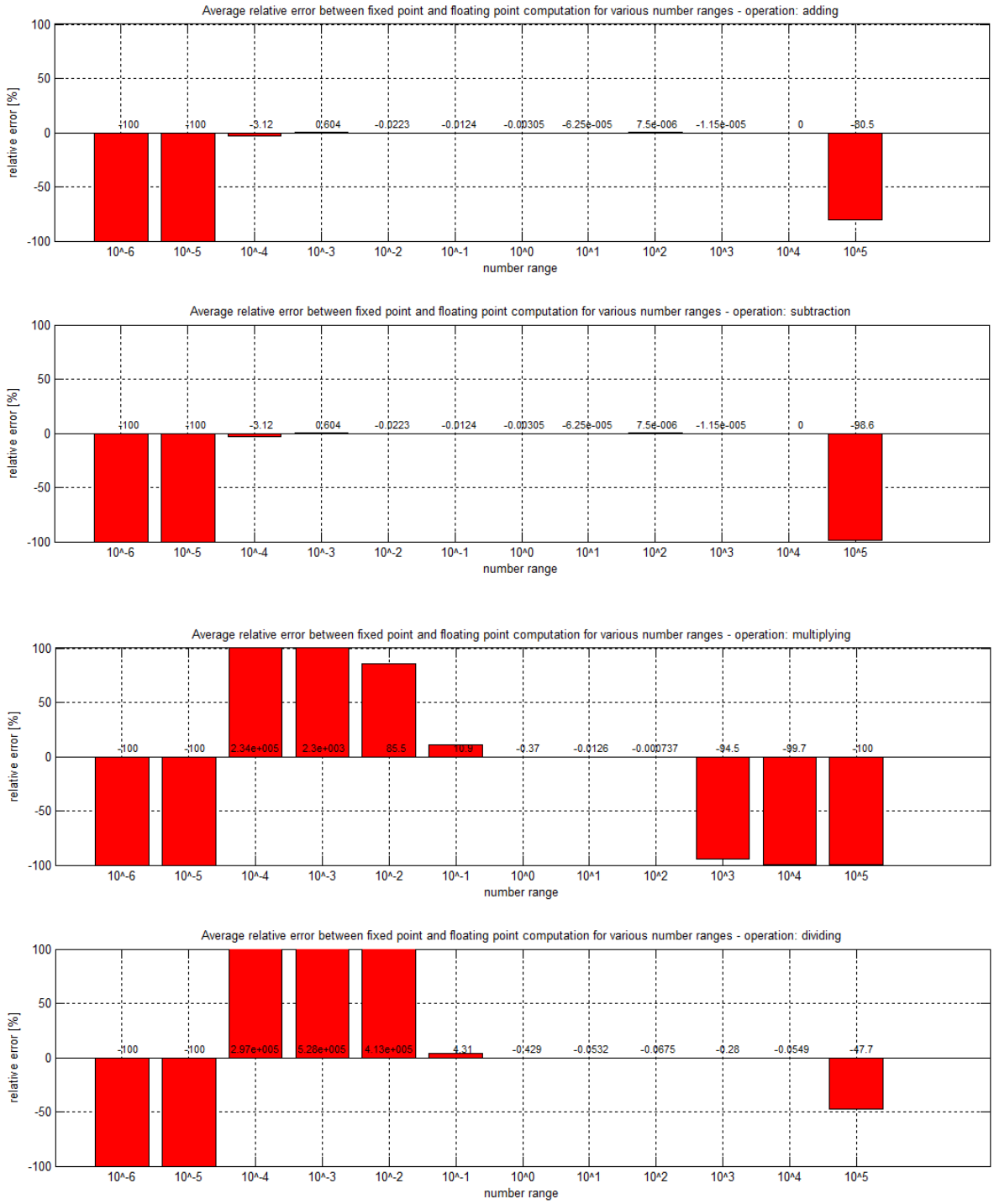
Po spuštění programu je uživatel vyzván k zadání velikosti celé a desetinné části *FX* formátu. Velikost se zadává v bitech. Pak již program běží automaticky a uživatel pouze potvrzuje stisknutím klávesy *Enter* běh dalšího cyklu. Na terminálu jsou vypisovány jednotlivé výpočty jak v *FX* formátu, tak v *FP* formátu, společně s relativní odchylkou obou řešení. Po skončení posledního cyklu se program ukončí.

Výsledky se kromě terminálu také zapisují do čtyř, pro každou aritmetickou operaci zvlášť textových souborů. Pomocí připraveného Matlab skriptu *view_results.m* můžeme výsledky zobrazit přehledně v grafu. Ukázka textového výstupu programu je na (obr. 2.5). Ukázka grafu ze skriptu je pro nastavení hodnot $A = 16$ a $B = 16$ na (obr. 2.6).

Tímto testovacím programem je možné sledovat chování a přesnost výpočtu za použití knihovny *fixedpoint* pro různé řády čísel. Je jasné, že chování bude různé, jelikož *FX* formát má vždy vymezený určitý počet bitů pro celou a desetinnou část čísla. Z (obr. 2.6) je pro nastavení *FX* formátu $A = 16$ a $B = 16$ vidět, že pro aritmetickou operaci součtu a rozdílu dochází v pásmu řádu čísel 10^{-4} až 10^{4} k zanedbatelné chybě při výpočtu. U operací součinu a podílu je toto pásmo, kde dochází k malé chybě, výrazně užší, řekněme od 10^{-1} do 10^{3}. Je to způsobeno právě úpravou přesnosti jednotlivých operandů pro výpočet (viz. podkapitola 2.2.1 a 2.2.2).

```
Max range of numbers: 1.000000e-02

ADDING:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/ Floatingpoint:        4.947661e-03    +       1.246986e-03    =       6.194647e-03
\ Fixedpoint:           4.943848e-03    +       1.235962e-03    =       6.179810e-03    Error: -0.239521 %
/ Floatingpoint:        -4.947661e-03   +       1.246986e-03    =       -3.700674e-03
\ Fixedpoint:           -4.943848e-03   +       1.235962e-03    =       -3.707886e-03   Error: 0.194864 %
/ Floatingpoint:        4.947661e-03    +       -1.246986e-03   =       3.700674e-03
\ Fixedpoint:           4.943848e-03    +       -1.235962e-03   =       3.707886e-03    Error: 0.194864 %
/ Floatingpoint:        -4.947661e-03   +       -1.246986e-03   =       -6.194647e-03
\ Fixedpoint:           -4.943848e-03   +       -1.235962e-03   =       -6.179810e-03   Error: -0.239521 %

SUBTRACTION:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/ Floatingpoint:        4.947661e-03    -       1.246986e-03    =       3.700674e-03
\ Fixedpoint:           4.943848e-03    -       1.235962e-03    =       3.707886e-03    Error: 0.194864 %
/ Floatingpoint:        -4.947661e-03   -       1.246986e-03    =       -6.194647e-03
\ Fixedpoint:           -4.943848e-03   -       1.235962e-03    =       -6.179810e-03   Error: -0.239521 %
/ Floatingpoint:        4.947661e-03    -       -1.246986e-03   =       6.194647e-03
\ Fixedpoint:           4.943848e-03    -       -1.235962e-03   =       6.179810e-03    Error: -0.239521 %
/ Floatingpoint:        -4.947661e-03   -       -1.246986e-03   =       -3.700674e-03
\ Fixedpoint:           -4.943848e-03   -       -1.235962e-03   =       -3.707886e-03   Error: 0.194864 %

MULTIPLYING:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/ Floatingpoint:        4.947661e-03    *       1.246986e-03    =       6.169665e-06
\ Fixedpoint:           4.943848e-03    *       1.235962e-03    =       0.000000e+00    Error: -100.000000 %
/ Floatingpoint:        -4.947661e-03   *       1.246986e-03    =       -6.169665e-06
\ Fixedpoint:           -4.943848e-03   *       1.235962e-03    =       0.000000e+00    Error: -100.000000 %
/ Floatingpoint:        4.947661e-03    *       -1.246986e-03   =       -6.169665e-06
\ Fixedpoint:           4.943848e-03    *       -1.235962e-03   =       -1.525879e-05   Error: 147.319565 %
/ Floatingpoint:        -4.947661e-03   *       -1.246986e-03   =       6.169665e-06
\ Fixedpoint:           -4.943848e-03   *       -1.235962e-03   =       3.051758e-05    Error: 394.639130 %

DIVIDING:
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
/ Floatingpoint:        4.947661e-03    /       1.246986e-03    =       3.967695e+00
Error: function divfix(): Dividing by zero!
\ Fixedpoint:           4.943848e-03    /       1.235962e-03    =       3.276800e+04    Error: 825770.021821 %
/ Floatingpoint:        -4.947661e-03   /       1.246986e-03    =       -3.967695e+00
Error: function divfix(): Dividing by zero!
\ Fixedpoint:           -4.943848e-03   /       1.235962e-03    =       -3.276800e+04   Error: 825770.022206 %
/ Floatingpoint:        4.947661e-03    /       -1.246986e-03   =       -3.967695e+00
\ Fixedpoint:           4.943848e-03    /       -1.235962e-03   =       -1.265625e+00   Error: -68.101753 %
/ Floatingpoint:        -4.947661e-03   /       -1.246986e-03   =       3.967695e+00
\ Fixedpoint:           -4.943848e-03   /       -1.235962e-03   =       1.265625e+00    Error: -68.101753 %

####################################################################################################
Press Enter to continue...
```

Obrázek 2.5: Ukázka textového výstupu programu *Řády čísel* pro testování knihovny *fixedpoint*

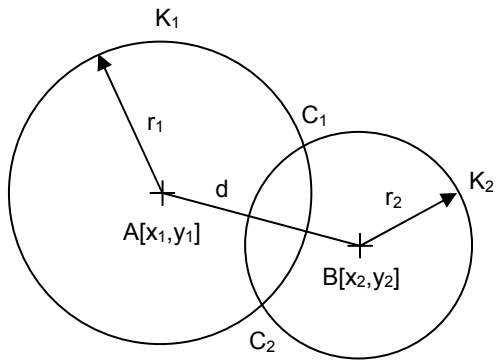


Obrázek 2.6: Graficky znázorněný výstup programu *Řády čísel* pomocí Matlab skriptu *view_results.m* při nastavení FX formátu $A = 16$ a $B = 16$

- *Kružnice*

Je důležité také otestovat, jak se knihovna *fixedpoint* chová pro složitější výpočty než je prosté sčítání, odčítání, násobení a dělení. Myslím tím zakomponování všech těchto matematických úkonů do složitějšího výpočetního algoritmu a sledování akumulace chyb jednotlivých operací do celkového výsledku.

Při výběru vhodné matematické úlohy mě zaujal postup pro výpočet průsečíků dvou kružnic podle (obr. 2.7) bez nutnosti řešení soustavy kvadratických rovnic. Tento postup jsem já osobně používal pro řešení inverzní kinematické úlohy planárního manipulátoru, kde bylo pro řešení kvadratické soustavy, z důvodu nízké rychlosti, nevhodné používat výpočetní nástroje poskytované programem Matlab. Základní vztahy pro výpočet jsou uvedeny v (2.18).



Obrázek 2.7: Nákres protínajících se dvou kružnic

$$k_1: \quad {r_1}^2 = (x - x_1)^2 + (y - y_1)^2$$
$$k_2: \quad {r_2}^2 = (x - x_2)^2 + (y - y_2)^2$$

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$
$$m = \frac{{r_1}^2 - {r_2}^2}{2d} + \frac{d}{2}$$
$$v = \sqrt{{r_1}^2 - m^2}$$
$$S_x = x_1 + \frac{m}{d}(x_2 - x_1)$$
$$S_y = y_1 + \frac{m}{d}(y_2 - y_1)$$

$$C_{x1,2} = S_x \pm \frac{v}{d}(y_1 - y_2)$$
$$C_{y1,2} = S_y \pm \frac{v}{d}(x_1 - x_2)$$

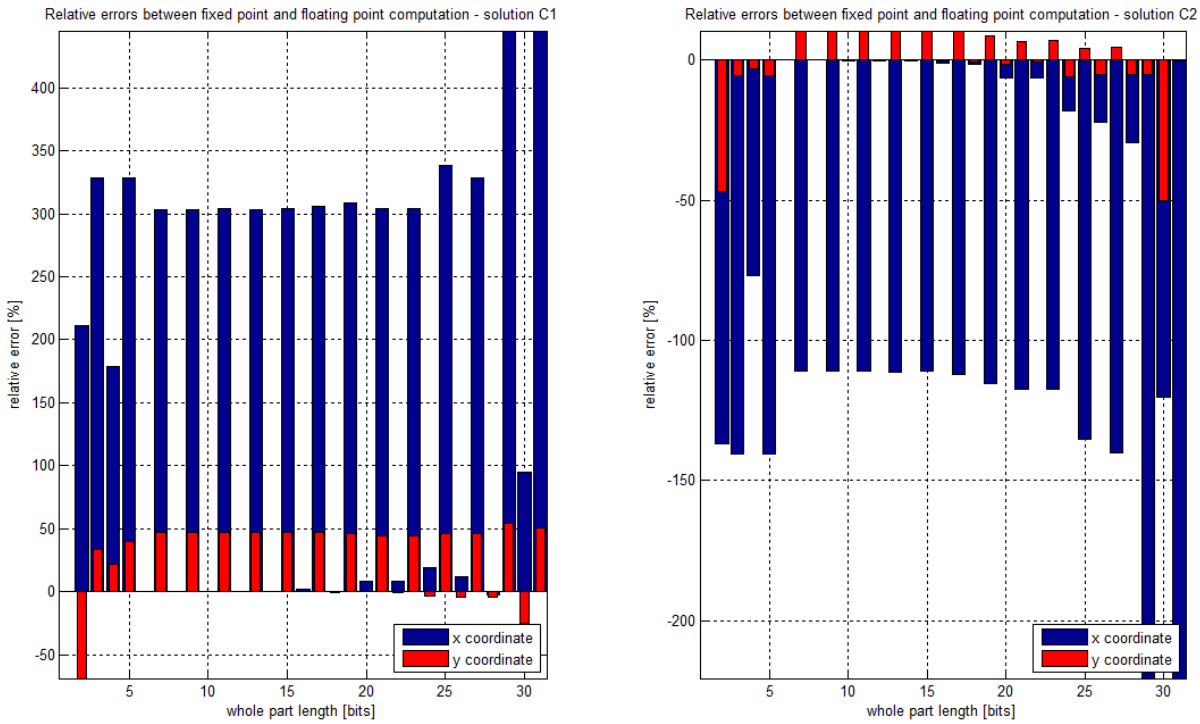
(2.18)

Program spočítá ze zadaných parametrů dvou kružnic jejich průsečíky. Výpočet se nejprve provede ve formátu plovoucí řádové čárky, jež se považuje jako referenční. Referenční výsledek se porovnává s výsledky spočtenými pomocí knihovny *fixedpoint* a to pro jedenatřicet různých nastavení poměru velikosti celé a desetinné části *FX* formátu.

Po spuštění programu uživatel zadá souřadnice středů obou kružnic a jejich poloměry. Po jejich zadání se na terminálu postupně pro všechna nastavení zobrazí souřadnice spočtených průsečíků společně s relativní odchylkou od referenčního řešení. Po vypsání všech jedenatřiceti výsledků se program ukončí.

Všechna data se opět ukládají do textového souboru pro možnost grafického znázornění výsledků pomocí skriptu *circles_view_results.m*. Tento skript zobrazí sloupcový graf relativních odchylek všech spočtených řešení v závislosti na nastavení *FX* formátu. Jelikož vzájemná poloha dvou kružnic může mít maximálně dvě řešení (pokud neuvažujeme

singulární případy nekonečně mnoha řešení), jsou grafy dva. Ukázka grafu a textového výstupu programu je na (obr. 2.8) a (obr. 2.9). V grafu je jasně vidět, že největší chyby vznikají při výrazných nepoměrech celé a desetinné části *FX* formátu. Všimněme si, že výrazné chyby ve výpočtu vznikají také pro „lichá“ nastavení (např. *A=15* a *B=17*). Je to pravděpodobně způsobeno snižováním přesnosti operandů u operací součinu a podílu (viz. podkapitola 2.2.1 a 2.2.2). Tato čísla bitově rotujeme o hodnotu *B/2* bitů směrem doprava. Pokud je ale *B* liché, vzniká tu chyba, která se projeví na výsledku.



Obrázek 2.8: Graficky znázorněný výstup programu *Kružnice* pomocí Matlab skriptu *circles_view_results.m*

```
Floating point results: C1 = [ 0.641917 ; 2.990054 ]    C2 = [ -1.244090 ; -2.510799 ]

Error: function flow2fix(): Maximum of fixedpoint format value was exceeded!
Error: function flow2fix(): Maximum of fixedpoint format value was exceeded!
Error: function flow2fix(): Maximum of fixedpoint format value was exceeded!
Error: function flow2fix(): Maximum of fixedpoint format value was exceeded!
Error: function mulfix(): Maximum of fixedpoint format value was exceeded!
Error: function addfix(): Maximum of fixedpoint format value was exceeded!
whole 1,fraction 31:    C1 = [   NaN   ;  NaN   ]       C2 = [   NaN   ;  NaN   ]       Error: C1 = [   NaN   ;  NaN   ]       C2 = [   NaN   ;  NaN   ]

Error: function flow2fix(): Maximum of fixedpoint format value was exceeded!
Error: function flow2fix(): Maximum of fixedpoint format value was exceeded!
Error: function flow2fix(): Maximum of fixedpoint format value was exceeded!
Error: function addfix(): Maximum of fixedpoint format value was exceeded!
Error: function mulfix(): Minimum of fixedpoint format value was exceeded!
Error: function mulfix(): Maximum of fixedpoint format value was exceeded!
Error: function mulfix(): Maximum of fixedpoint format value was exceeded!
Error: function mulfix(): Maximum of fixedpoint format value was exceeded!
Error: function addfix(): Maximum of fixedpoint format value was exceeded!
whole 2,fraction 30:    C1 = [2.000000; 0.933952]       C2 = [0.460782; -1.333940]      Error: C1 = [211.566922%; -68.764712%]  C2 = [-137.037717%; -46.871898%

Error: function flow2fix(): Maximum of fixedpoint format value was exceeded!
Error: function mulfix(): Maximum of fixedpoint format value was exceeded!
Error: function addfix(): Maximum of fixedpoint format value was exceeded!
Error: function mulfix(): Maximum of fixedpoint format value was exceeded!
Error: function mulfix(): Maximum of fixedpoint format value was exceeded!
Error: function mulfix(): Maximum of fixedpoint format value was exceeded!
Error: function mulfix(): Maximum of fixedpoint format value was exceeded!
Error: function addfix(): Maximum of fixedpoint format value was exceeded!
whole 3,fraction 29:    C1 = [2.747706; 4.000000]       C2 = [0.502721; -2.373917]      Error: C1 = [328.047191%; 33.776869%]   C2 = [-140.408738%; -5.451731%]

Error: function mulfix(): Maximum of fixedpoint format value was exceeded!
Error: function addfix(): Maximum of fixedpoint format value was exceeded!
Error: function mulfix(): Maximum of fixedpoint format value was exceeded!
Error: function mulfix(): Maximum of fixedpoint format value was exceeded!
Error: function mulfix(): Maximum of fixedpoint format value was exceeded!
whole 4,fraction 28:    C1 = [1.789191; 3.631091]       C2 = [-0.289191; -2.431104]     Error: C1 = [178.726384%; 21.439006%]   C2 = [-76.754804%; -3.174093%]

Error: function mulfix(): Maximum of fixedpoint format value was exceeded!
Error: function addfix(): Maximum of fixedpoint format value was exceeded!
Error: function mulfix(): Maximum of fixedpoint format value was exceeded!
Error: function mulfix(): Maximum of fixedpoint format value was exceeded!
Error: function mulfix(): Maximum of fixedpoint format value was exceeded!
whole 5,fraction 27:    C1 = [2.747891; 4.174130]       C2 = [0.502963; -2.373843]      Error: C1 = [328.076023%; 39.600518%]   C2 = [-140.428204%; -5.454648%]

whole 6,fraction 26:    C1 = [0.642283; 2.989919]       C2 = [-1.243480; -2.510447]     Error: C1 = [0.057133%; -0.004486%]     C2 = [-0.049039%; -0.014012%]

whole 7,fraction 25:    C1 = [2.590160; 4.390357]       C2 = [0.135182; -2.770287]      Error: C1 = [303.504091%; 46.832055%]   C2 = [-110.865919%; 10.334914%]

whole 8,fraction 24:    C1 = [0.642857; 2.990395]       C2 = [-1.243199; -2.510825]     Error: C1 = [0.146509%; 0.011436%]      C2 = [-0.071607%; 0.001060%]

whole 9,fraction 23:    C1 = [2.591015; 4.390610]       C2 = [0.136036; -2.770034]      Error: C1 = [303.637207%; 46.840526%]   C2 = [-110.934603%; 10.324826%]

whole 10,fraction 22:   C1 = [0.643227; 2.989834]       C2 = [-1.241860; -2.509678]     Error: C1 = [0.204107%; -0.007348%]     C2 = [-0.179256%; -0.044636%]

whole 11,fraction 21:   C1 = [2.591888; 4.389487]       C2 = [0.138581; -2.767739]      Error: C1 = [303.773276%; 46.802965%]   C2 = [-111.139132%; 10.233425%]

whole 12,fraction 20:   C1 = [0.642258; 2.988125]       C2 = [-1.240891; -2.507969]     Error: C1 = [0.053126%; -0.064504%]     C2 = [-0.257158%; -0.112701%]
```

Obrázek 2.9: Ukázka textového výstupu programu *Kružnice* pro testování knihovny *fixedpoint*

Kapitola 3

Matlab a S-funkce

Pro testování navržených algoritmů číslicových regulátorů bylo zapotřebí použít uživatelsky příjemné, nejlépe grafické prostředí, kde by bylo možné jednoduše používat tyto regulátory jako celek, nastavovat jejich parametry a modelovat funkci na různých regulačních obvodech. Všechny tyto požadavky je možné splnit použitím simulačního programu Matlab, konkrétně jeho části Simulink, a tzv. S-funkcí.

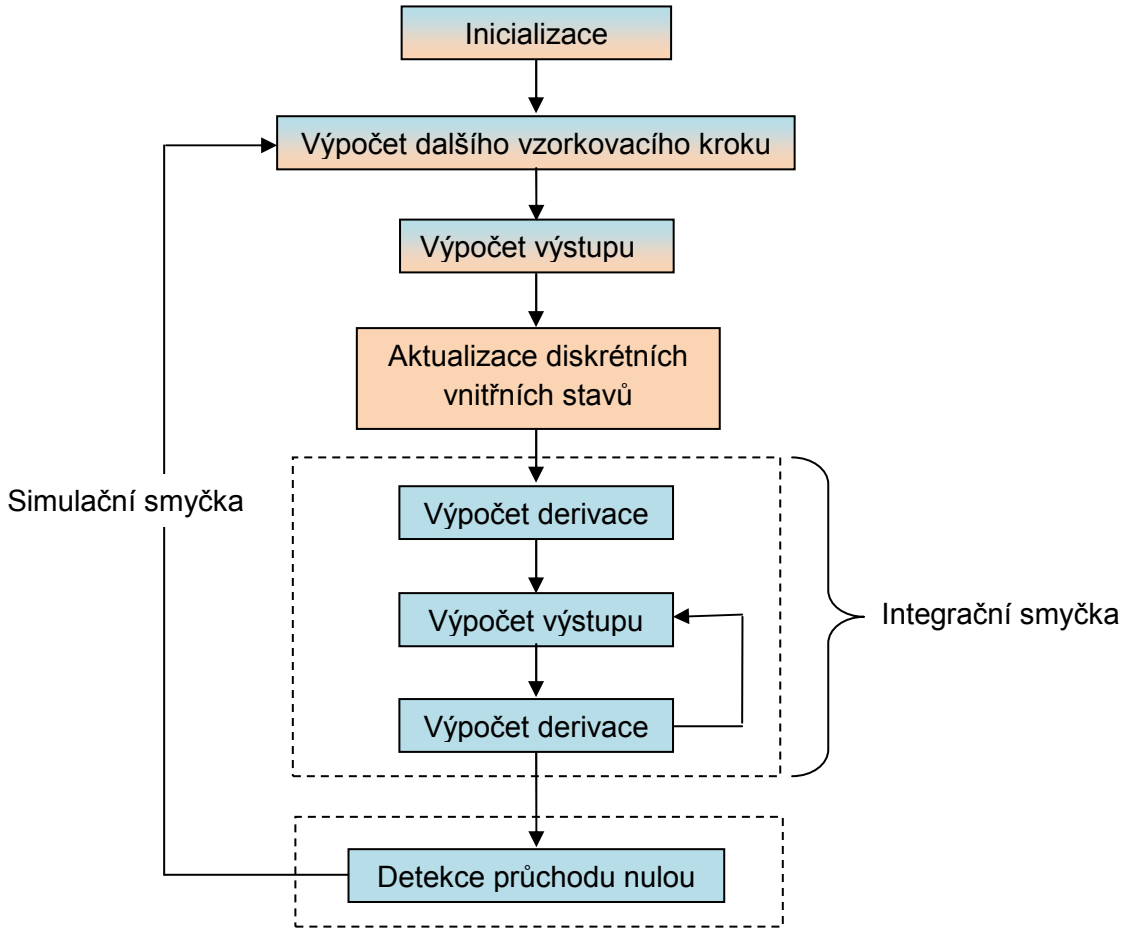
Systémové funkce (S-funkce) jsou užitečným nástrojem rozšiřujícím možnosti Simulinku. Všechny bloky z knihovny Simulinku jsou ve svém jádru realizovány právě těmito funkcemi. Implementací v programovacích jazycích Matlab, C, C++, Ada nebo Fortran, které jsou pro tvorbu S-funkcí podporovány, je možné si vytvořit své vlastní funkční bloky. Aby bylo možné s těmito programy pracovat, je nutné je přeložit pomocí příkazu *mex* na spustitelné *MEX*-soubory. To ale neplatí v případě, že S-funkci píšeme přímo jako *M*-soubor, tedy přímo v Matlabu. Při psaní systémových funkcí je také nutné dodržovat určitou formu a pravidla, které korespondují s principem Simulinku. Tato pravidla jsou definována standardem *S-function API*. Při práci jsem používal S-funkce programované v jazyce C, jejichž tvorba je v této kapitole stručně popsána.

3.1 Princip C MEX S-funkcí

C MEX S-funkce je označení pro systémové funkce, které jsou realizovány v programovacím jazyce C. V této kapitole vysvětlím, jak takovéto funkce fungují.

Každá S-funkce představuje jeden funkční blok Simulinku. Blok se sestává ze sady vstupů, vnitřních stavů a výstupů. Každý výstup je pak dán časovou funkcí vstupu a vnitřních

stavů. Po spuštění simulace je obecně postupováno podle diagramu (obr. 3.1). Pro každý krok má S-funkce přesně definované metody, které jsou v průběhu simulace volány.

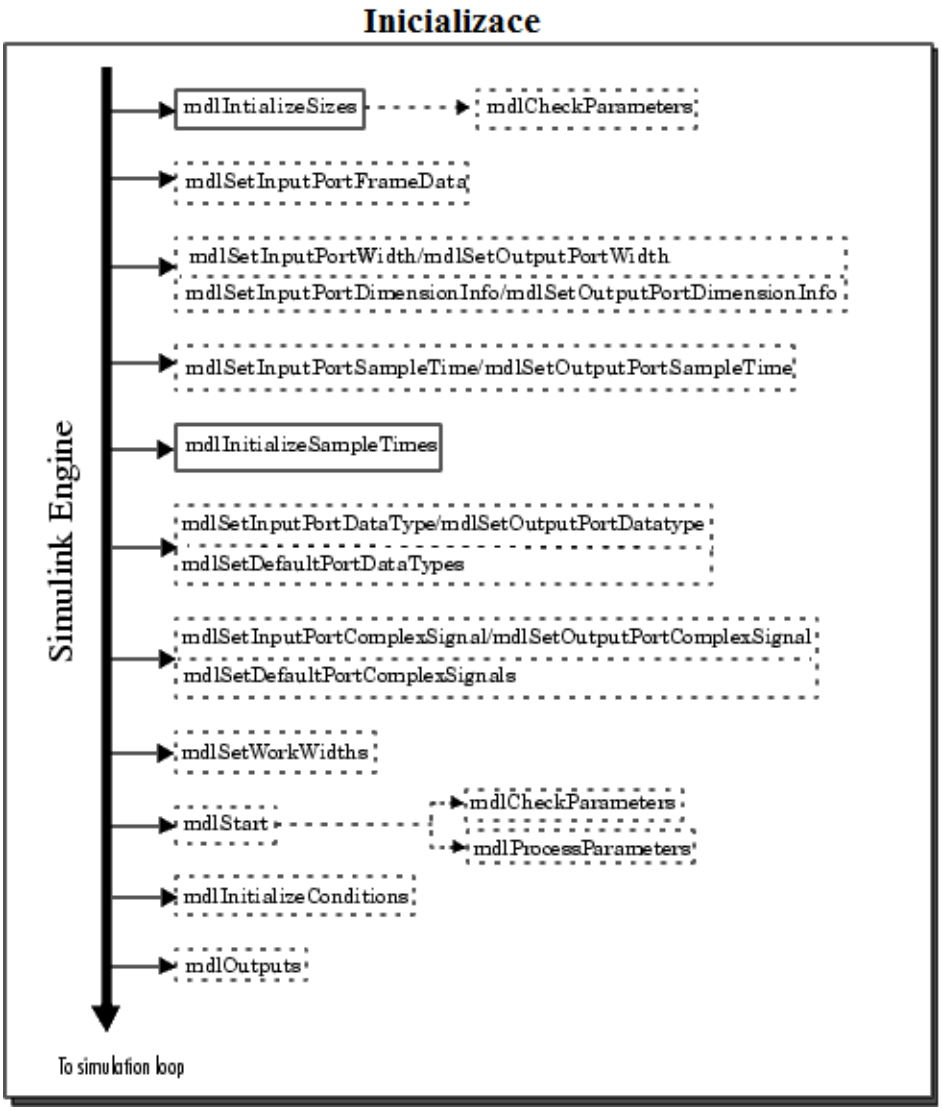


Obrázek 3.1: Diagram simulační smyčky S-funkce

Blok *Inicializace* představuje počáteční nastavení celého systému. Definuje se zde, mimo jiného, velikost (dimenze) vstupů a výstupů, datové typy nebo vzorkovací perioda. Kontrolují se nastavené parametry systému a probíhá zde alokace potřebné paměti. Po inicializaci program vkročí do simulační smyčky, kde se v každém kroku spočte aktuální hodnota výstupu, a aktualizují se vnitřní stavy. Zde se musí rozlišovat, zda systém pracuje v diskrétním nebo spojitém čase. V případě spojitého systému je do simulační smyčky zařazena ještě integrační smyčka, pro výpočet spojitých derivací. Pro lepší přehlednost diagramu (obr. 3.1), jsem jednotlivé bloky označil barevně podle spojitosti. Modré bloky náleží spojitým systémům, červené pak diskrétním. Kombinovaná barva označuje bloky potřebné pro oba typy spojitosti. Popsaný děj se neustále opakuje až do skončení celé simulace.

Pro všechny části simulační smyčky jsou přesně definovány metody v C programu, které jsou v průběhu simulace volány. Formát těchto metod se musí striktně řídit podle definovaného standardu *S-function API*. Proto je vhodné při implementaci používat

připravené šablony uložené v adresáři Matlabu nebo intuitivní nástroj Simulinku zvaný *S-function Builder*. Abych ukázal, jak může vypadat vnitřní struktura některé části simulačního pochodu, tím myslím její volané metody, vybral jsem na ukázku blok *Inicializace* (obr. 3.2).



Obrázek 3.2: Blok Inicializace a jeho vnitřní metody pro C MEX S-funkce[1]

Obrázek (obr. 3.2) zobrazuje seznam nejčastěji volaných metod v průběhu inicializace a to v pořadí, v jakém jsou volány. Výčet metod ale není pro všechny S-funkce stejný. Jejich volání je podmíněno typem systému, rozměry vstupů nebo výstupů a dalšími parametry. Funkce *mdlInitializeSizes* a *mdlInitializeSampleTimes* jsou zvýrazněny, jelikož se volají jak při počáteční inicializaci, tak vždy i v průběhu simulace. Ostatní metody jsou pak většinou volány jen na začátku celého pochodu. Tyto dvě funkce jsou zvýrazněny také z toho důvodu,

[1] převzato z [5] *Simulink 7: Writing S-functions. Documentation for MathWorks Products*

že patří k těm nejdůležitějším metodám, bez kterých žádná S-funkce nemůže existovat. Mezi takové metody se řadí také *mdlOutputs* a *mdlTerminate*, které se volají při běhu simulační smyčky.

3.2 Metody implementace C MEX S-funkcí

V této části bych chtěl ukázat, jaké možnosti nabízí Matlab pro psaní systémových funkcí. Existují tři způsoby:

- *Ručně psané S-funkce* – využití připravených šablon
- *S-function Builder* – nástroj Simulinku pro vytváření S funkcí
- *Legacy code tool* – nástroj pro dědění zdrojových kódů

Při vytváření svých vlastních S-funkcí pro testování navržených algoritmů číslicových regulátorů jsem používal kombinaci prvních dvou způsobů implementace.

3.2.1 Ručně psané C MEX S-funkce

Tento způsob implementace C MEX systémových funkcí je podle mého názoru sice nejobtížnější, ale z důvodu možnosti plné editovatelnosti také nejlepší. Uživatel ale nepíše celý zdrojový kód S-funkce od úplného začátku, jak by se mohlo na první pohled zdát. Bylo by to velice obtížné z důvodu nutnosti zachování standardu *S-function API*. Proto se využívá připravených šablon, které tvoří jakousi kostru celého kódu.

Šablony jsou uloženy v adresáři `matlabroot/simulink/src`. Šablon je v tomto adresáři několik a jsou obecně označeny jménem *sfuntmpl_XXX.c*. Nejdůležitější je pak soubor *sfuntmpl_doc.c*. Jedná se o kompletně předpřipravenou šablonu, která obsahuje všechny metody poskytované standardem společně s velice podrobným popisem jejich účelu a implementace. Méně zkušený uživatel může využít šablony *sfuntmpl_basic.c*, která má ve svém obsahu pouze nejdůležitější metody.

Šablony programátor může editovat podle vlastního uvážení. Je pouze nutné zachovat ty části, které musí každá S-funkce bezpodmínečně obsahovat. První podmínkou je definice jména systémové funkce a zahrnutí knihovny *simstruc.h*. Tato knihovna zajišťuje komunikaci mezi prostředím Simulinku a S-funkcemi. V řeči programovacího jazyka C tomu odpovídá zápis podle (3.1) a je vždy umístěn na začátku zdrojového kódu. Naopak na konci každé S-funkce je vždy definováno podle (3.2), zda se má zdrojový kód překládat jako *MEX*-soubor nebo jako *M*-soubor. Poslední podmínkou je pak implementace těch nejzákladnějších metod v těle kódu potřebných pro každou simulaci.

Jedná se o metody:

- *mdlInitializeSizes*
- *mdlInitializeSampleTimes*
- *mdlOutputs*
- *mdlTerminate*

To jsou tedy hlavní tři náležitosti, které musí z hlediska standardu libovolná S-funkce splňovat, aby mohla spolehlivě fungovat.

```
#define S_FUNCTION_NAME název S-funkce
#define S_FUNCTION_LEVEL 2
#include "simstruc.h"
```

(3.1)

```
#ifdef MATLAB_MEX_FILE
#include "simulink.c"
#else
#include "cg_sfun.h"
#endif
```

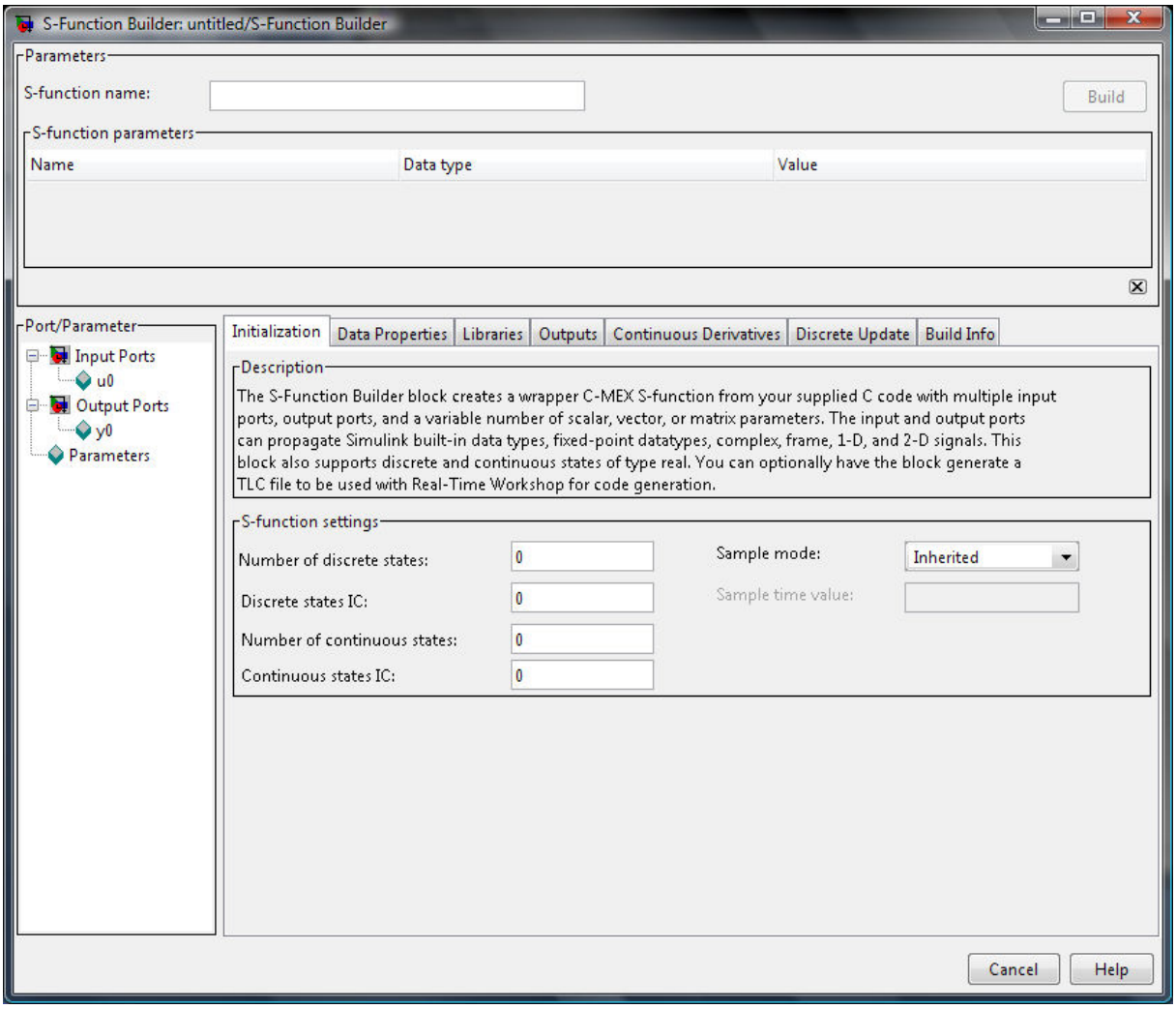
(3.2)

3.2.2 S-function Builder

S-function Builder je nástroj Simulinku pro rychlou implementaci jednodušších S-funkcí. Pokud chceme začít pracovat s tímto nástrojem, vybereme z knihovny Simulinku blok s názvem *S-function Builder* a přemístíme ho na pracovní plochu. Po jeho otevření se zobrazí dialogové okno (obr. 3.3), ve kterém se posléze navrhuje celá S-funkce. Nejprve je nutné definovat jméno navrhované S-funkce. Název se zadává do kolonky *S-function name*. Návrh se pak provádí v záložkách:

- *Initialization* – definuje se počet spojitých a diskrétních vnitřních stavů včetně jejich počátečních hodnot a také typ a hodnota vzorkování.
- *Data Properties* – v této záložce se nastavuje počet vstupů, výstupů a vnitřních parametrů navrhované S-funkce. Definují se jejich jména, dimenze a datové typy.
- *Libraries* – uživatel zde specifikuje, které programové knihovny nebo externí funkce, kromě standardních, chce ve své S-funkci používat, např. knihovnu *math.h*.
- *Outputs* – definuje výstupní funkci systému. Funkce se zapisuje přímo ve formě programovacího jazyka C.
- *Continuous Derivatives* – nastavuje se pouze v případě použití minimálně jednoho spojitého vnitřního stavu. Podobně jako u záložky *Outputs* se definuje funkce pro aktualizaci spojitých vnitřních stavů.

- *Discrete Update* – obdoba předchozího nastavení, v tomto případě ale pro diskrétní stavy.
- *Build Info* - zobrazení informací o průběhu kompilace S-funkce a nastavení doplňkových možností týkajících se způsobu překladu.



Obrázek 3.3: Dialogové okno S-function Builder

Po provedení všech nastavení a specifikací je možné vygenerovat *MEX*-soubor navržené S-funkce stiskem tlačítka *Build*. *MEX*-soubor se vygeneruje podle požadavků definovaných v záložce *Build Info* a uloží se v pracovním adresáři Matlabu.

Výhodou použití této metody je jednoduchost a rychlost návrhu S-funkcí. Po vygenerování *MEX*-souboru se blok *S-function Builder* automaticky změní na blok přímo realizující navrženou S-funkci, se kterým je možné okamžitě pracovat. Blok dostane dokonce i jednoduchou masku (viz. podkapitola 3.4). Velkou nevýhodou použití tohoto nástroje je velmi malé využití všech možností, které S-funkce poskytují. Pro tvorbu mých vlastních S-

funkcí jsem *S-function Builder* používal, ale pouze pro vygenerování základní kostry programu. Tím mám na mysli vygenerování základních potřebných metod a deklaraci vstupních a výstupních portů a vnitřních parametrů, která by při ručním psaní zabrala mnohem více času. Ručně jsem pak většinou pouze implementoval metodu *mdlOutputs*, případně i *mdlInitializeSizes*.

3.3 Kompilace a ladění C MEX S-funkcí

O kompilaci, tedy překladu, budu mluvit pouze pro případ ručně psaných S-funkcí, jelikož při použití *S-function Builderu* je kompilace zajištěna automaticky tímto nástrojem. Dále se pak ukážu, jakým způsobem lze odlaďovat uživatelské S-funkce.

3.3.1 Kompilace C MEX S-funkcí

Pro překlad C MEX systémových funkcí slouží matlabovský příkaz *mex*, který volá tzv. *MEX*-kompilátor. Kompilovat je možné ve třech módech:

- *MATLAB_MEX_FILE* – překladem se vytvoří *MEX*-soubor použitelný v Simulinku
- *RT* – překladem se vytvoří produkt tzv. *Real-Time Workshopu* pro aplikace pracující v reálném čase
- *NRT* – překladem se vytvoří produkt tzv. *Real-Time Workshopu* pro aplikace nepracující v reálném čase

Pro mé aplikace jsem používal vždy první mód, tedy generování *MEX*-souborů.

Před prvotním použitím příkazu *mex* je dobré nastavit výchozí kompilátor. To se provede příkazem *mex –setup*. Po jeho spuštění se Matlab nejprve uživatele dotáže, zda chce vyhledat ve svém počítači všechny nainstalované kompilátory nebo automaticky vyhledat pouze ty, které podporují *mex* překlad. Osobně doporučuji druhou možnost. Matlab pak vypíše všechny podporované překladače. Uživatel si z nabídky zvolí, který kompilátor chce pro překlad svých S-funkcí používat. V mém případě jsem používal kompilátor *Microsoft Visual C++ 2005*.

Po základním nastavení již nic nebrání v kompilaci připravených S-funkcí. To se provede tak, že do příkazové řádky Matlabu napíšeme příkaz *mex*, za nímž následují názvy všech souborů obsahujících zdrojové kódy realizující S-funkci. Při úspěšné kompilaci se příkaz provede bez jakéhokoliv hlášení. Budeme-li naše S-funkce chtít ladit, překladači toto oznámíme přepínačem *–g*. Celý příkaz pak bude obecně vypadat podle (3.3).

```
mex -g soubor1.c soubor2.c soubor3.c … souborX.c
```

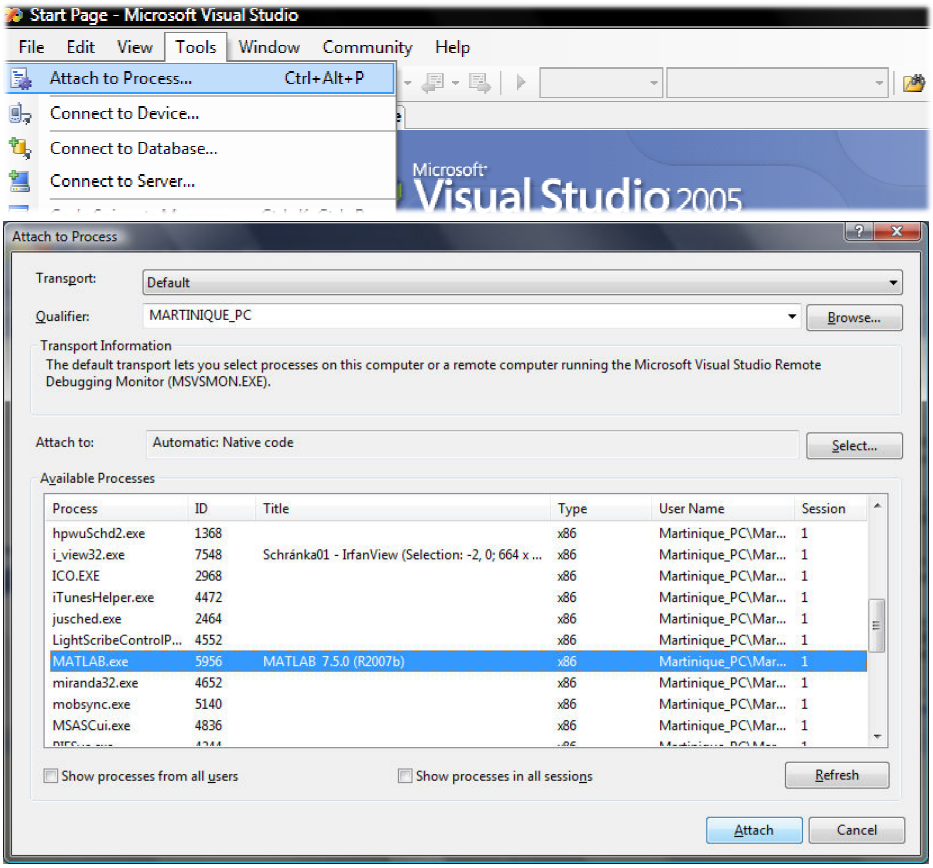
(3.3)

3.3.2 Ladění C MEX S-funkcí

Každý programátor často používá při vývoji svých programů nějaké specializované vývojové prostředí, které mu většinou poskytuje i užitečné nástroje pro ladění a hledání chyb. V případě S-funkcí ale Matlab bohužel nenabízí žádný kvalitní *debugger*. Za pomoci jiných programových a vývojových prostředků je ale možné si ladící nástroj vytvořit. Jedním ze způsobů je využití vývojového prostředí Microsoft Visual C/C++ .NET.

Předpokládejme, že máme v Simulinku vytvořený model obsahující blok s S-funkcí zkompilovanou jako laditelný *MEX*-soubor (viz. podkapitola 3.3.1). Nyní je potřeba propojit prostředí Microsoft Visual C/C++ .NET s prostředím Matlabu, respektive Simulinku. Spustíme program Microsoft Visual Studio. V menu *Tools* zvolíme položku *Attach to Process*. Po otevření dialogového okna vybereme ze seznamu spuštěných procesů program Matlab. Postup je ukázán na (obr. 3.4). Tím dojde k propojení obou procesů. Nyní již stačí otevřít v prostředí Visual Studia ty části zdrojového kódu S-funkce, které chceme ladit a označit *breakpointy*. Po spuštění simulace v Simulinku a průchodu zdrojovým kódem, který je v prostředí Microsoft Visual označený *breakpointem*, se běh simulace pozastaví a je tak možné přistoupit k ladění, které se provádí přímo v prostředí Visualu.

Tento způsob ladění se může zdát komplikovaný, ale je jediným způsobem *debuggingu* S-funkcí v prostředí Windows.

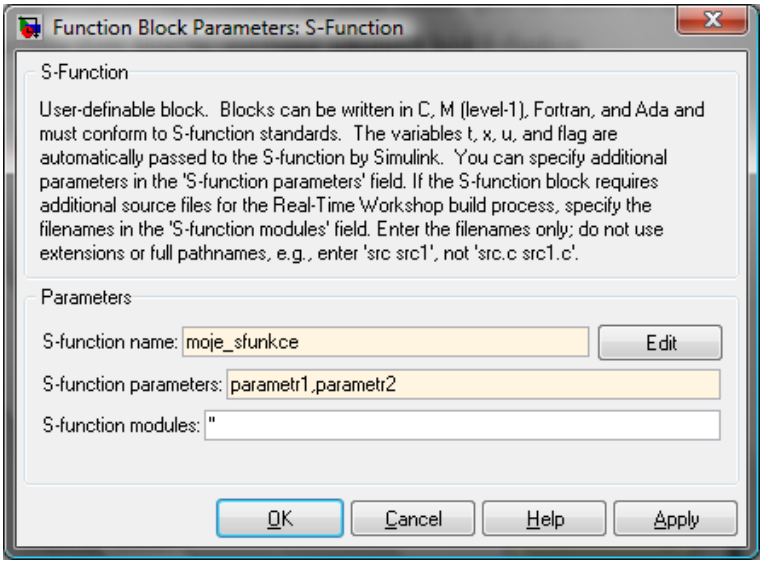


Obrázek 3.4: Postup pro připojení k procesu Matlab
v prostředí Microsoft Visual Studio

3.4 Používání C MEX S-funkcí a jejich maskování

Poslední podkapitolu o S-funkcích bych věnoval popisu, jak s nimi v Matlabu pracovat a jakým způsobem z nich lze použitím tzv. masky vytvořit uživatelsky příjemný systém pro modelování v Simulinku.

Předpokládejme, že máme vytvořenou a zkompilovanou S-funkci ve formě *MEX*-souboru uloženého v pracovním adresáři Matlabu. Z knihovny Simulinku vybereme blok označený *S-function* a přesuneme ho na pracovní plochu. Dvojitým kliknutím levým tlačítkem myši na tento blok se otevře dialogové okno (obr. 3.5). Do kolonky označené jako *S-function name* vepíšeme název vytvořené S-funkce. Do další kolonky s názvem *S-function parameters* se vyplní hodnoty vnitřních parametrů systému, realizovaného připravenou S-funkcí, oddělené čárkami. V některých případech (například při použití maskování) je vhodnější místo přímých hodnot zadat názvy proměnných, které pak budou jednotlivé parametry systému zastupovat. Poslední kolonka *S-function modules* je vyplňována pouze při použití *Real-Time* S-funkcí. Po zadání všech parametrů se nastavení potvrdí tlačítkem *OK*. Jednoduchý blok, který jsme si před tím vložili na pracovní plochu, se nyní změní do podoby požadovaného systému s odpovídajícím počtem vstupů a výstupů.

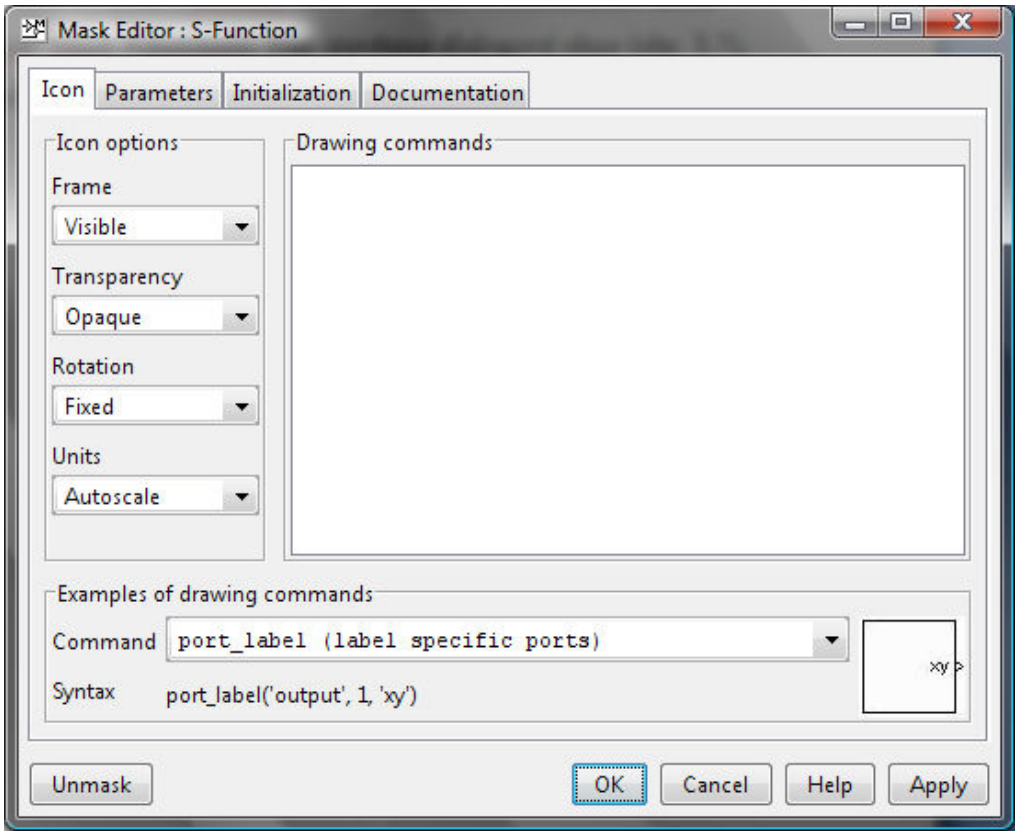


Obrázek 3.5: Dialogové okno pro nastavení parametrů S-function bloku v Simulinku

Pokud by uživatel chtěl vytvořit profesionální blok, který by mohli používat i ostatní uživatelé, je vhodné použití tzv. maskování. Maskování znamená vytvoření uživatelsky příjemného dialogu pro nastavení parametrů systému tvořeného S-funkcí. Pokud by S-funkce realizovala systém s několika vstupními parametry, použití klasického nastavovacího dialogu *S-function* bloku by nebylo příliš vhodné. Je to z toho důvodu, že při vyplňování kolonky *S-function parameters* musí uživatel znát přesnou implementaci dané S-funkce, to znamená

přesné množství, datové typy, dimenze a pořadí zadávaných parametrů. Uživateli tento problém je možné jednoduše ulehčit právě vytvořením masky.

Po kliknutí pravým tlačítkem myši na *S-function* blok lze z rozbaleného menu vybrat položku *Mask S-function*. Tím se otevře editor (obr. 3.6), ve kterém se provádí celý návrh masky.



Obrázek 3.6: Editor pro tvorbu masky S-function bloku

První záložka s názvem *Icon* dává možnost grafického návrhu *S-function* bloku. Návrh se provádí jednoduchými příkazy, jejichž syntaxe je popsána v dolní části editoru. Je možné si tak definovat například popisky vstupních a výstupních portů nebo si tělo bloku vyplnit nějakým vhodným názvem či obrázkem. Druhá záložka *Parameters* slouží k definici proměnných, které budou zastupovat vnitřní parametry systémové funkce a jejichž hodnoty se budou zadávat v nastavovacím dialogovém okně. U každé proměnné je pak možné především definovat její jméno a popis, který se v dialogovém okně bude zobrazovat u příslušné kolonky. Další záložka *Initialization* slouží pro nastavení počátečních hodnot proměnných a záložka *Documentation* pro vytvoření popisu a nápovědy maskované S-funkce.

Maskování je tedy užitečným nástrojem, jak vytvářet obecně použitelné a jednoduše ovladatelné funkční bloky pro simulaci v Simulinku.

Kapitola 4

Obecný regulátor

Prvním z hlavních úkolů této práce bylo navrhnout obecný algoritmus regulátoru pro úlohy zpětnovazebního řízení v programovacím jazyce C. Pojem „obecný regulátor“ se může zdát poněkud zavádějící, ale v této práci ho budu používat pouze pro jednodušší označení regulátoru se dvěma stupni volnosti a na vstupu zařazeným *FIR* filtrem pro tvarování referenčního signálu. Podmínkou implementace byla možnost jednoduchého testování algoritmu na různých úlohách řízení a možnost snadného nastavování parametrů celého systému. Dalším požadavkem pak byla realizace dvou verzí tohoto regulátoru. První z nich je verzí pracující s binární reprezentací čísel v klasickém formátu plovoucí řádové čárky (tedy použití datového typu *double*). Druhá pak má za úkol používat formát pevné řádové čárky. Při její implementaci jsem tedy využil mou knihovnu *fixedpoint* popsanou v kapitole 2.

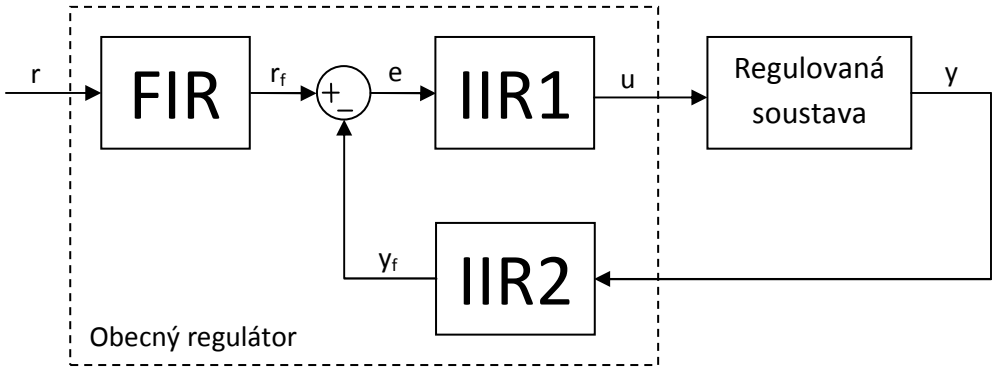
Obě verze jsou, co se týče uživatelského ovládání a použití, téměř totožné, liší se pouze v reprezentaci číselných hodnot a způsobu provádění aritmetických výpočtů. Pro ověření správné funkce navrženého algoritmu a testování zde nalézá své uplatnění samozřejmě program Matlab, s jehož řešením budu mé výsledky ze simulačních úloh porovnávat. K účelům porovnávání výsledků využiji jak vytvořený skript, který graficky porovnává spočtená data z obou verzí (plovoucí a pevné řádové čárky) implementovaného regulátoru s vlastním výpočtem dané úlohy, tak i pro každou verzi vytvořené S-funkce (viz. kapitola 3), které umožňují používat naprogramované regulátory přímo v Simulinku.

V následujícím textu vysvětlím, co je myšleno pod pojmem obecný regulátor. Dále pak také objasním způsob mé implementace regulátoru v programovacím jazyce C a jeho testování. Zároveň také popíši tvorbu S-funkce, která bude navržený algoritmus regulátoru využívat pro simulace v Simulinku.

4.1 Struktura obecného regulátoru

Vnitřní struktura obecného regulátoru, ze které budu vycházet při pozdější implementaci programu, má tvar podle (obr. 4.1). Referenční signál *r* je na vstupu filtrován číslicovým *FIR* filtrem, tedy filtrem s konečnou délkou impulsní odezvy. Filtr slouží nejčastěji k odstranění vysokých frekvencí referenčního signálu. Z filtrované žádané hodnoty r_f a měřeného výstupu regulované soustavy y_f, který je také filtrován, tentokrát ale číslicovým filtrem s nekonečnou délkou impulsní odezvy *IIR2*, je vypočtena regulační odchylka *e*. Výstup z regulované soustavy je obvykle filtrován ze stejného důvodu jako signál referenční, tedy z důvodu možného superponovaného vysokofrekvenčního šumu v signálu. Filtrace je vhodná zejména v případě, že by byl regulátor navržen s derivační složkou ve zpětné vazbě, kdy by docházelo k velkému nežádoucímu zesilování vysokých frekvencí. Budeme-li uvažovat klasické uspořádání regulačního obvodu, můžeme říct, že jádro regulátoru je tvořeno blokem *IIR1*, jehož vstupem je spočtená regulační odchylka *e* a výstupem odpovídající akční zásah *u*. Jelikož regulátor je v podstatě vždy systém s definovaným přenosem, který má určité frekvenční vlastnosti, můžeme blok *IIR1* nazvat také filtrem.

Z předchozího popisu vyplývá, že náš obecný regulátor bude tvořen třemi číslicovými filtry, a to jedním *FIR* filtrem a dvěma filtry *IIR*. Jejich parametry bude možné nastavovat, což umožní tvorbu libovolných řídících zpětnovazebních systémů.



Obrázek 4.1: Struktura obecného regulátoru

4.1.1 Číslicové FIR filtry

Číslicové filtry s konečnou délkou impulsní odezvy jsou lineární diskrétní časově invariantní systémy s nerekurzivní strukturou. Nerekurzivní filtry jsou takové filtry, které neobsahují žádnou zpětnou vazbu. Diskrétní přenos *FIR* filtru je dán vztahem (4.1). Pokud přenos vyjádříme v kladných mocninách parametru *z* podle (4.2), je z tohoto tvaru vidět, že diskrétní *FIR* filtry mají *M*-násobný pól v nule. Znamená to, že *FIR* filtry jsou vždy stabilní. *M* nazýváme řádem filtru. Z diferenční rovnice (4.3), kterou lze odvodit z přenosu (4.1) zpětnou *Z*-transformací, je patrné, že výstupní hodnota signálu závisí na aktuální hodnotě

vstupu a *M* předchozích hodnotách vstupu. K ustálení výstupního signálu dojde tedy až po *M* krocích. V praxi se ale tyto filtry používají s velmi vysokým řádem, což způsobuje velké zpoždění signálu na výstupu filtru. To je jejich velkou nevýhodou. Naopak velkou výhodou je možnost jejich návrhu s lineární frekvenční fázovou charakteristikou, jejich stabilita a relativně snadná realizace.

$$H(z) = \sum_{k=0}^{M} b_k z^{-k} \tag{4.1}$$

$$H(z) = \frac{\sum_{k=0}^{M} b_k z^{M-k}}{z^M} \tag{4.2}$$

$$y(k) = b_0 x(k) + b_1 x(k-1) + \cdots + b_M x(k-M) \tag{4.3}$$

4.1.2 Číslicové IIR filtry

Filtry s nekonečnou délkou impulsní odezvy se od *FIR* filtrů liší tím, že obsahují minimálně jednu zpětnovazební smyčku, mají tedy rekurzivní strukturu. Přenos *IIR* filtru je dán podílem dvou polynomů a řád filtru je vyšší ze stupňů obou polynomů vyjádřených v kladných mocninách parametru *z*. Díky zavedené zpětné vazbě nemusí být tento druh filtru navrhován s tak velkým řádem jako je tomu u *FIR* filtrů. Mají tedy rychlejší přechodovou charakteristiku a jsou tak schopni rychleji reagovat na změny vstupního signálu. Jejich nekonečně dlouhá impulsní odezva je dána umístěním pólů, které nejsou umístěny v nule, ale v prostoru jednotkové kružnice *Z*-roviny. Přenos je v záporných mocninách parametru *z* vyjádřen podle (4.4) a tomu odpovídající diferenční rovnice (4.5). Pro kauzální filtr pak musí platit, že N ≥ M. Nevýhodou těchto filtrů je jejich obtížnější realizace a také možnost vzniku nestability. Při použití rekurzivních filtrů využívajících algoritmy pracující v binárním formátu pevné řádové čárky mohou vznikat také tzv. *mezní cykly*, což je jev, při kterém se za určitých okolností na výstupu filtru objevuje periodický signál, i když vstup je konstantní nebo nulový. Zmiňuji se o tomto problému, jelikož se tu budu v dalších kapitolách zabývat právě implementací *IIR* filtrů v aritmetice pevné řádové čárky.

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_M z^{-M}}{1 - a_1 z^{-1} - a_2 z^{-2} - \cdots - a_N z^{-N}} \tag{4.4}$$

$$y(k) = \sum_{i=0}^{M} b_i x(k-i) + \sum_{i=1}^{N} a_i y(k-i) \tag{4.5}$$

4.2 Implementace obecného regulátoru

Algoritmus obecného regulátoru jsem implementoval v programovacím jazyce C. Nejprve popíši implementaci regulátoru ve formátu plovoucí řádové čárky, tedy klasickou realizaci programu pomocí datového typu *double*. Tím získám podklad i pro druhou verzi programu, kterou pomocí knihovny *fixedpoint* (viz. kapitola 2) jednoduše přepracuji na algoritmus pracující s binárním formátem pevné řádové čárky.

4.2.1 Implementace ve formátu plovoucí řádové čárky

Jádro programu realizujícího regulátor tvoří knihovny *PSD.c* a *filter.c*. Knihovna *PSD* představuje samotný obecný regulátor a používá při tom knihovnu *filter*, která implementuje funkci číslicových filtrů. Regulátor bylo možné realizovat přímo v jedné knihovně, která by prováděla výpočty jak z pohledu řízení tak i filtrace, ale z důvodu možného obecného využití knihovny *filter* pro jiné aplikace, jsem programy rozdělil.

- *Knihovna filter*

```
int filter_reset_params ( Filter_params_structure *filter_params,
                          int num_order,
                          int den_order,
                          double *b_const,
                          double *a_const )
```
(4.6)

```
int FIR_init ( Filter_params_structure *FIR_params )
```
(4.7)

```
int IIR_init ( Filter_params_structure *IIR_params )
```
(4.8)

```
int filter_tf_print ( Filter_params_structure *filter_params )
```
(4.9)

```
double filter_response ( const double input,
                         Filter_params_structure *filter_params )
```
(4.10)

```
void filter_terminate ( Filter_params_structure *filter )
```
(4.11)

Knihovna realizuje číslicové filtry *FIR* a *IIR*. Je tedy důležitou součástí pro implementaci mého obecného regulátoru. Podstatnou částí knihovny je definice struktury *Filter_params_structure*. Struktura uchovává parametry každého filtru. Konstanty polynomů čitatele a jmenovatele přenosové funkce jsou zastoupeny formou ukazatelů *a* a *b* na paměťová

pole, která budou alokována funkcí *malloc* (*memory allocation*) až při spuštění inicializační funkce knihovny *filter* podle požadavků uživatele. Tento způsob dynamického alokování paměti je pro realizaci číslicových filtrů, respektive i dalších systémů používající paměťová pole proměnné velikosti, velice výhodný, neboť jeho použitím zajistíme maximální univerzálnost použití knihovny. Dále jsou ve struktuře stejným způsobem deklarovány ukazatele *u* a *y* na pole pro uchovávání historie vstupních a výstupních dat. Jelikož se historie vstupů a výstupů bude neustále měnit, jsou ve struktuře definovány dvě proměnné *actual_u* a *actual_y*, které plní funkci jakéhosi ukazatele, respektive indexu, ukazující v poli historie na aktuální hodnoty. Díky těmto ukazatelům se nemusí při změně historie neustále měnit celý obsah polí a tím se tak celý algoritmus zefektivní. Poslední informací, kterou struktura nese, je skutečný řád čitatele (proměnná *numerator*) a jmenovatele (proměnná *denominator*) navrženého filtru.

Knihovna *filter* obsahuje kromě zmíněné definice struktury, také pět metod. První z nich je funkce *filter_reset_params*. Hlavička funkce je uvedena v (4.6). Tuto metodu jsem implementoval hlavně z důvodu její potřeby při pozdějším psaní S-funkce, ale nalezne určitě i jiná uplatnění. Funkce provádí nastavení parametrů filtru definovaného strukturou *filter_params*. Funkce je vhodná pro nastavení filtru bez použití inicializační metody, která vyžaduje spolupráci s obsluhovatelem programu, nebo při potřebě kopírování parametrů dvou filtrů. Prvním ze vstupních argumentů funkce je ukazatel na strukturu filtru, kterou chceme nastavit. Kromě struktury se funkci zadávají také hodnoty všech nastavovaných parametrů, a to řád čitatele, řád jmenovatele a pole s konstantami polynomů přenosové funkce filtru. V těle funkce je nejprve alokováno odpovídající místo v paměti pro všechny konstanty. Potom jsou veškeré parametry zkopírovány do zadané struktury *filter_params*.

Dalšími dvěma metodami knihovny jsou inicializační funkce *FIR_init* a *IIR_init*, které provádí nastavení *FIR* a *IIR* filtrů pomocí komunikace s uživatelem. Funkce jsou implementovány ve tvaru (4.7) a (4.8). Po jejich zavolání se nejprve vynulují všechny parametry inicializovaného filtru dané strukturou *FIR_params*, případně *IIR_params*. Následně je uživatel dotázán, zda chce parametry filtru zadat ručně pomocí terminálu nebo zda je má program načíst z textového souboru. V případě použití textového souboru musí být striktně dodržen pro každý typ filtru určitý formát zápisu, který je popsán v přiložených souborech *FIR_help.txt*, popřípadě *IIR_help.txt*. Při ručním zadávání, je uživatel postupně vyzýván k zadání vždy jednoho z parametrů. Všechny načtené parametry jsou zkontrolovány a v případě výskytu nějaké chyby je funkcí vrácen odpovídající chybový kód. V opačném případě funkce ukončí svou činnost s návratovou hodnotou *0*. Funkce *FIR_init* a *IIR*_init se liší pouze tím, že u *FIR_init* nedochází k načítání konstant jmenovatele přenosu filtru.

Třetí metoda *filter_tf_print*, jež má tvar podle (4.9), je pouze doplňkovou funkcí, která pomáhá kontrolovat správnost nastavení filtru tím, že vypíše jeho přenos ve formě zlomku polynomů v kladných mocninách parametru *z* na textový výstup programu. Vstupním argumentem funkce je ukazatel na strukturu *filter_params* držící informace o vypisovaném přenosu filtru.

Další metodou v knihovně je *filter_response*, počítající odezvu filtru na vstupní signál. Metoda je volána pro každý vzorkovací krok. Vstupními parametry funkce je aktuální hodnota vstupního signálu reprezentovaná proměnnou *input* a ukazatel *filter_params* ukazující na strukturu filtru, jehož odezva má být spočtena. Hlavička funkce má tvar podle

(4.10). Výpočet se provádí přesně podle definice diferenční rovnice (4.5) *IIR* filtru za použití *for* cyklů. Algoritmus výpočtu se používá stejný pro oba typy filtrů. Je to pochopitelné, jelikož *FIR* filtry jsou pouze speciálním případem *IIR* filtrů. Po dokončení celého výpočtu funkce vrátí vypočtenou aktuální hodnotu výstupu filtru.

Poslední metodou knihovny je *filter_terminate* ve tvaru podle (4.11). Tuto metodu je nutné vždy zavolat, chceme-li ukončit veškerou práci s filtrem definovaným ve struktuře *filter_params*. Jelikož je paměť pro uložení parametrů filtru alokována až v průběhu běhu programu, to znamená, že není známa v době překladu, je nutné alokovanou paměť po skončení práce znovu uvolnit. To provádí právě tato metoda pomocí funkce *free*.

- *Knihovna PSD*

```
int PSD_reset_params ( PSD_params_structure *PSD_params,
                       Filter_params_structure *FIR,
                       Filter_params_structure *IIR1,
                       Filter_params_structure *IIR2,
                       double low_bound,
                       double high_bound )
```
(4.12)

```
int PSD_init ( PSD_params_structure *PSD_params )
```
(4.13)

```
double PSD_control ( double r,
                     double y,
                     PSD_params_structure *PSD_params )
```
(4.14)

```
void PSD_terminate(PSD_params_structure *PSD_params)
```
(4.15)

Taro knihovna představuje přímo obecný regulátor. Při použití v programech pro konkrétní úlohy řízení voláme právě metody této knihovny. Podstatnou částí knihovny je opět struktura, v tomto případě pojmenovaná *PSD_params_structure*, ve které budeme uchovávat všechny parametry regulátoru. Těmito parametry jsou převážně opět struktury, tentokrát ale typu *Filter_params_structure*, které uchovávají nastavení číslicových filtrů. Filtry potřebujeme v našem obecném regulátoru tři, jeden *FIR* filtr a dva *IIR* filtry (viz. obr. 4.1). Dále je ve struktuře *PSD_params_structure* definováno saturační omezení výstupu regulátoru, tedy akčního zásahu.

Knihovna obsahuje samozřejmě i několik metod. Jednou z nich je metoda *PSD_reset_params*. Tato metoda má téměř totožnou funkci jako *filter_reset_params*, o které jsem mluvil v popisu knihovny *filter*, jen s tím rozdílem, že v tomto případě se nastavují parametry regulátoru. Hlavička funkce má tvar podle (4.12).

Další metodou této knihovny je inicializační funkce *PSD_init* implementovaná ve tvaru (4.13). Tato funkce provádí nastavení regulátoru za přímé účasti uživatele. Má pouze jeden vstupní argument a tím je ukazatel na strukturu nastavovaného regulátoru. Při zavolání této

metody se na terminálu uživatele vykreslí schéma zapojení regulátoru s popisky jednotlivých bloků. Je to jen z toho důvodu, aby uživatel věděl, co nastavuje. Následně se začnou volat inicializační metody pro každý druh používaného filtru, tedy jedna inicializace *FIR* filtru a dvě inicializace filtrů *IIR*. Tyto metody byly vysvětleny v popisu knihovny *filter*. Nakonec je uživatel vyzván k zadání hodnot horní a dolní saturační meze. Po kontrole všech parametrů se funkce ukončí s odpovídajícím návratovým kódem.

Třetí metodou knihovny je *PSD_control*, jejíž tvar je uveden v (4.14). Je to hlavní funkce regulátoru realizující řídící algoritmus popsaný obrázkem (obr. 4.1). Funkce počítá aktuální hodnotu akčního zásahu z hodnot referenčního signálu a výstupního signálu z regulované soustavy. Referenční signál reprezentovaný vstupní proměnnou r je vyfiltrován *FIR* filtrem s parametry definovanými ve struktuře *PSD_params*, která se funkci předává rovněž ve formě vstupního parametru. Tato struktura také nese informace o dalších dvou *IIR* filtrech regulátoru. Filtrem *IIR2* se vyfiltruje výstupní signál z řízené soustavy, jehož hodnota je reprezentována další vstupní proměnnou y. Rozdílem filtrovaných signálů r_f a y_f je získána regulační odchylka, která se použije jako vstup do posledního filtru *IIR1*, jehož odezva je požadovaným akčním zásahem celého regulátoru. Spočtená hodnota je případně ještě saturačně omezena a nakonec vrácena jako návratová hodnota funkce.

Poslední metodou je stejně jako v případě knihovny *filter* ukončovací funkce *PSD_terminate*. Tato funkce je volána na konci programu používajícího knihovnu *PSD*. Funkce provádí uvolnění dynamicky alokované paměti pro parametry všech tří filtrů regulátoru. Uvolnění paměti se provede voláním metody *filter_terminate* z knihovny *filter*. Metoda je implementována ve tvaru podle (4.15).

4.2.2 Implementace ve formátu pevné řádové čárky

Nyní máme tedy již hotovou verzi obecného regulátoru pracující ve formátu plovoucí řádové čárky. To znamená, že pro reprezentaci číselných hodnot, se kterými se v řídícím algoritmu počítá, se používá datový typ *double* a jeho aritmetika. Pro verzi programu ve formátu pevné řádové čárky využiji, z hlediska algoritmu, naprosto stejné implementace knihoven *PSD* a *filter*. Pro rozlišení obou verzí jsem obě knihovny pro tento případ přejmenoval na *fixPSD* a *fixFilter*. Dávám tak najevo, že se jedná o verzi ve formátu pevné řádové čárky.

K nové verzi je připojena knihovna *fixedpoint* (viz. kapitola 2). Všechny důležité proměnné typu *double* jsou v tomto případě deklarovány jako datový typ *fixp*, reprezentující *32*bitový celočíselný typ *integer*, definovaný v knihovně *fixedpoint*. Jedná se o datový typ pro formát pevné řádové čárky. Pro mé účely jsem používal *32*bitový formát, ale jeho velikost je možné volit libovolně.

Další změnou, kterou je nutné provést na původní implementaci, je redefinování všech aritmetických operací. Doteď byly pro operace sčítání, rozdílu, součinu a podílu čísel v *FP* formátu používány klasické aritmetické operátory +, -, *, /. Jelikož nyní je zapotřebí provádět výpočty v *FX* formátu, je nutné tyto operátory nahradit funkcemi z knihovny *fixedpoint*. Například přepis části kódu (4.16) z knihovny *PSD* bude pak v knihovně *fixPSD* vypadat

podle (4.17). Zbytek přepracování knihoven spočívá už jen v menších úpravách původního kódu. Například u inicializačních funkcí, kde uživatelem zadávané parametry jsou načítány v *double* formátu, musíme tyto parametry pomocí funkce *float2fix* z knihovny *fixedpoint* převést do *FX* reprezentace.

```
filter_params->y[filter_params->actual_y] /= filter_params->
a[filter_params->denominator];
```

(4.16)

```
fixFilter_params->y[fixFilter_params->actual_y] =
divfix ( fixFilter_params->y[fixFilter_params->actual_y],
fixFilter_params->a[fixFilter_params->denominator] );
```

(4.17)

4.3 Použití a testování obecného regulátoru

Pro ukázku používání obecného regulátoru, tedy knihovny *PSD* i *fixPSD*, jsem vytvořil jednoduchý zkušební program aplikující řídící algoritmus na modelu libovolné soustavy.

Pro jeho používání jsem ale nejprve musel vytvořit speciální knihovnu *system*, která bude realizovat řízenou soustavu. Nebudu zde podrobně popisovat tuto knihovnu, jelikož pracuje naprosto na stejném principu jako realizace *IIR* filtrů v knihovně *filter*, jejíž popis byl uveden v předchozí části kapitoly.

Po spuštění testovacího programu jsou vytvořeny struktury *PSD_params* a *fixPSD_params* reprezentující parametry obou verzí regulátoru. Ke každému regulátoru je deklarována také vždy jedna regulovaná soustava opět ve formě struktury. Všechny struktury jsou následně uživatelem nastaveny inicializačními metodami skrze textové uživatelské prostředí. Aby bylo zajištěno, že obě verze regulátoru budou, z důvodu možnosti porovnání výsledků, regulovat stejné řízené soustavy, je uživatelem nastavena pouze jedna struktura uchovávající informace o systému. Druhá je pak vytvořena kopírováním parametrů první struktury metodou *system_reset_params* z knihovny *system*. Zároveň je také inicializována knihovna *fixedpoint* pro výpočty ve formátu pevné řádové čárky. Po nastavení všech parametrů program pokračuje dotazem na jméno textového souboru obsahujícího vstupní data, tedy referenční signál. Jelikož jsem se ve většině případů snažil změřit přechodovou charakteristiku, používal jsem soubor dat *input.txt*, který představuje jednotkový skok. Jakmile program získá všechny parametry potřebné pro simulaci, vkročí do simulační smyčky, kde se pro všechna vstupní data spočítají akční zásahy obou regulátorů a odpovídající odezva řízených systémů. Výsledky každého kroku regulace se přehledně zobrazují společně s informací o absolutní chybě mezi řešením *FP* a *FX* aritmetiky v uživatelském okně a zároveň se ukládají do textového souboru pod názvem *output.txt* pro možnost dalšího zpracování. To nám umožní i grafické zobrazení výsledků pomocí připraveného Matlab skriptu *compare_PSD.m*, který provede porovnání regulace s modelem

implementovaným v Matlabu. Ukázka práce regulátoru a porovnání výsledků pro nastavení podle (4.18) je na (obr. 4.2).

Abych ukázal, že implementované knihovny *filter* a *system* pracují správně, vytvořil jsem speciálně jen pro ně obdobný testovací program. Tento program spočítá odezvu diskrétního systému, *FIR* filtru a *IIR* filtru pro uživatelem zadané specifikace a vstupní signál. Spočtené výsledky je možné opět porovnat se simulací Matlabu pomocí skriptu *compare_systems.m*. Ukázka funkčnosti knihoven je pro nastavení podle (4.19) na (obr. 4.3).

Přenos regulovaného systému: $S = \frac{-0{,}002716\, z^3 + 0{,}003069\, z^2 + 0{,}002483\, z - 0{,}002524}{z^4 - 3{,}699\, z^3 + 5{,}208\, z^2 - 3{,}311\, z + 0{,}8026}$

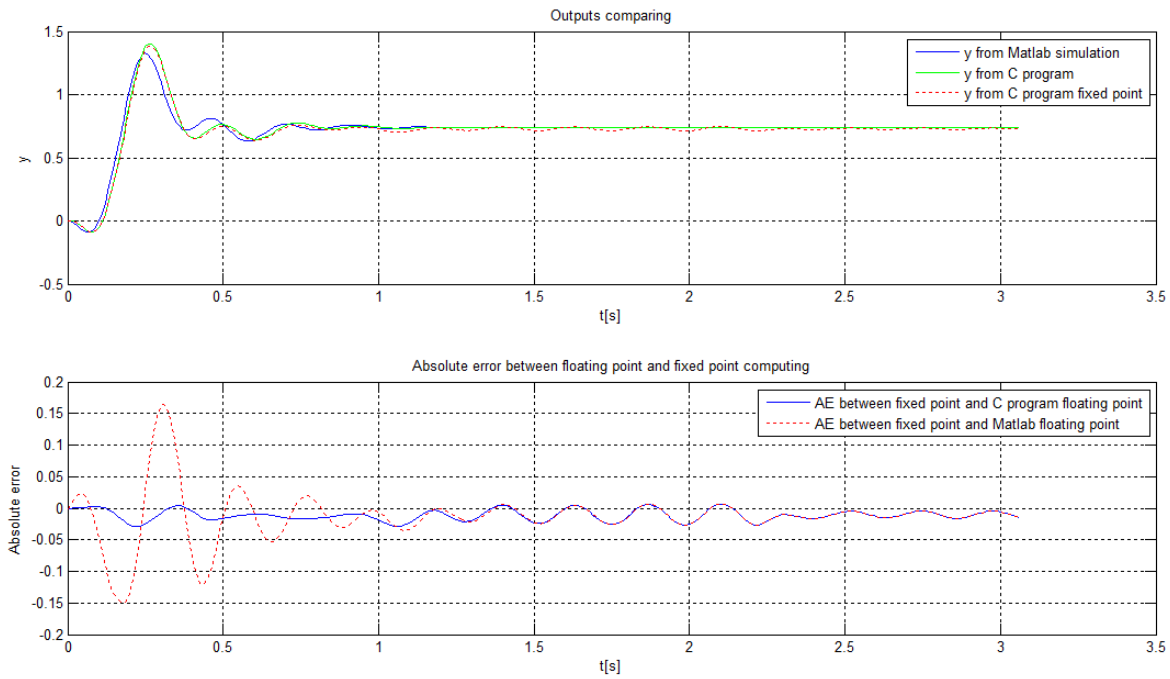
Přenos FIR filtru regulátoru: $S = \frac{0{,}1197\, z^{15} + 0{,}1197\, z^{14} + 0{,}1197\, z^{13} + \ldots + 0{,}1197\, z + 0.1197}{z^{15}}$

Přenos IIR1 filtru regulátoru: $S = \frac{12{,}91\, z^2 - 21{,}02\, z + 8{,}449}{z^2 - 0{,}9454\, z + 0{,}2244}$

Přenos IIR2 filtru regulátoru: $S = \frac{z^2 - 1{,}63\, z + 0{,}7291}{z^2 - 1{,}372\, z + 0{,}4705}$

Saturační meze: +20/-20
Perioda vzorkování: 0.01s
Vstupní signál: 1(t)
Nastavení FX formátu: A = 16, B = 16

(4.18)



Obrázek 4.2: Porovnání výsledků regulace obecného regulátoru se simulací Matlabu pomocí skriptu *compare_PSD.m*

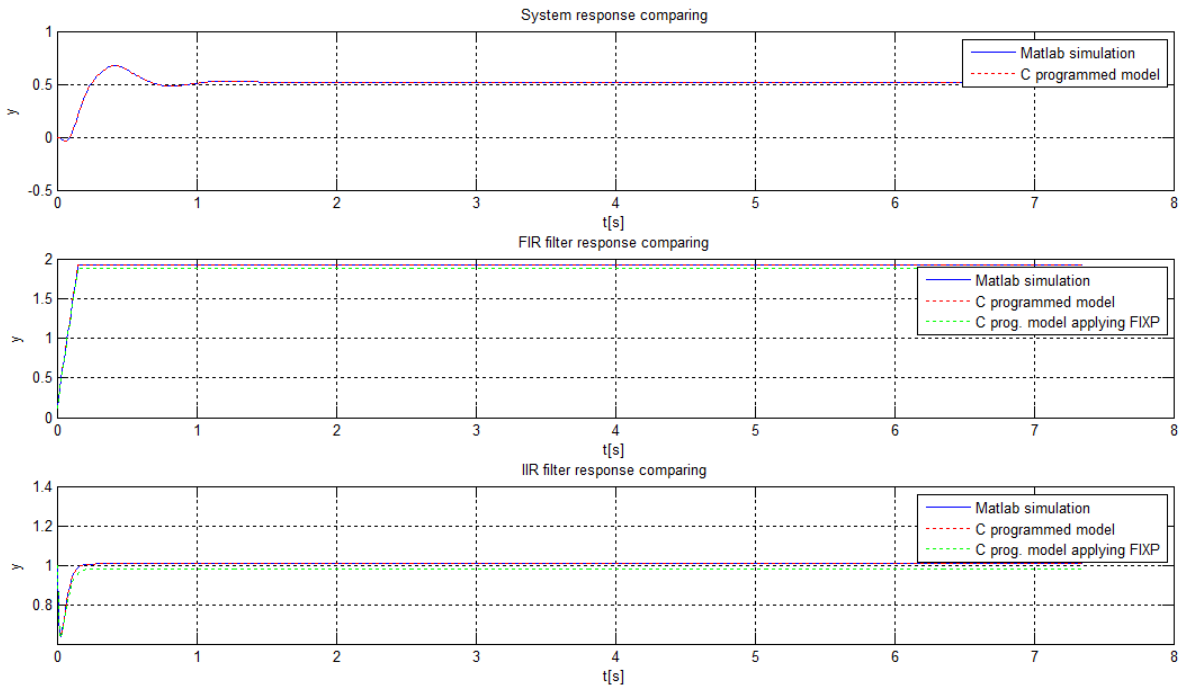
Systém: $S = \frac{-0{,}002716\, z^3 + 0{,}003069\, z^2 + 0{,}002483\, z - 0{,}002524}{z^4 - 3{,}699\, z^3 + 5{,}208\, z^2 - 3{,}311\, z + 0{,}8026}$

FIR filtr: $S = \frac{0{,}1197\, z^{15} + 0{,}1197\, z^{14} + 0{,}1197\, z^{13} + \ldots + 0{,}1197\, z + 0.1197}{z^{15}}$

IIR filtr: $S = \frac{z^2 - 1{,}63\, z + 0{,}7291}{z^2 - 1{,}372\, z + 0{,}4705}$

Perioda vzorkování: 0.01s
Vstupní signál: 1(t)
Nastavení FX formátu: A = 16, B = 16

(4.19)



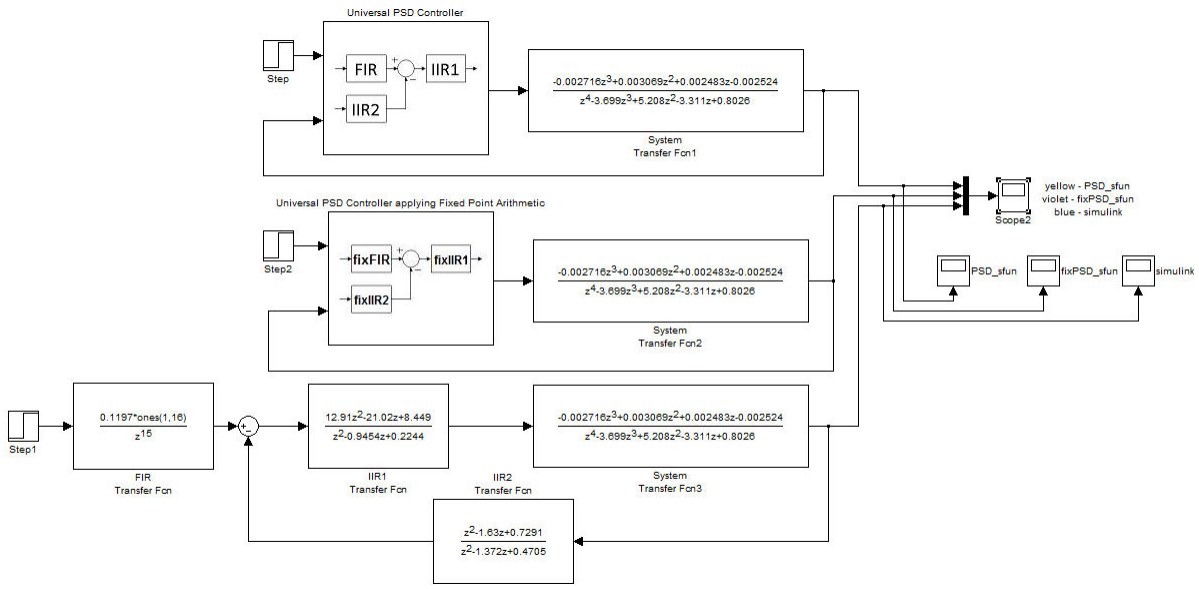
Obrázek 4.3: Výsledky testu funkčnosti knihoven *system* a *filter* s porovnáním se simulací v Matlabu pomocí skriptu *compare_systems.m*

4.3.1 S-funkce obecného regulátoru

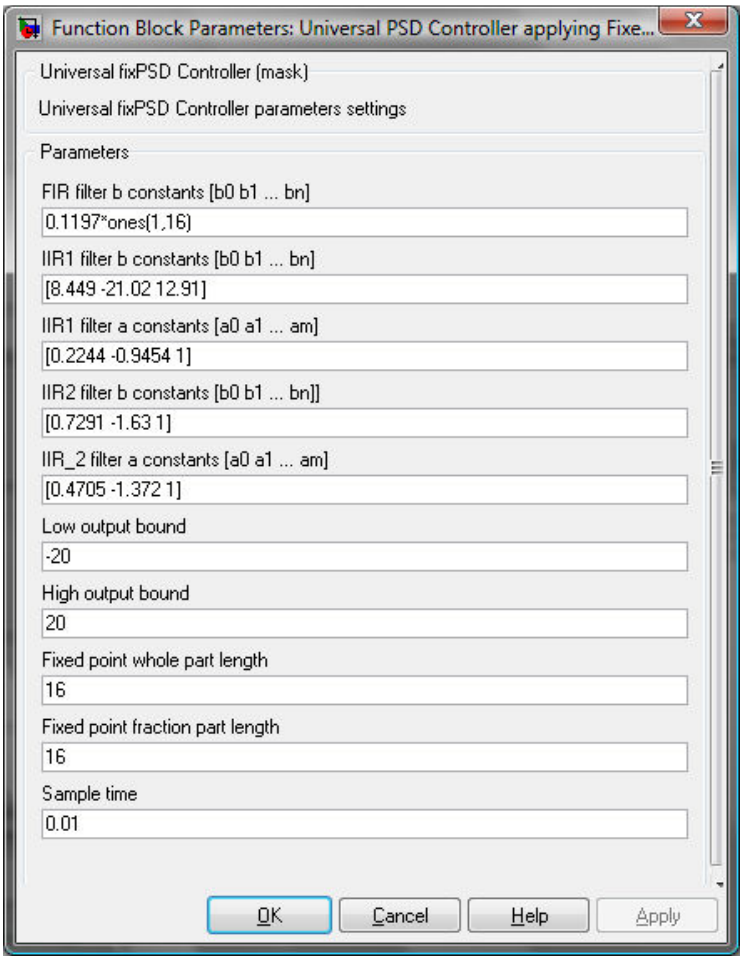
Speciální podkapitolu bych chtěl věnovat tvorbě S-funkce (viz. kapitola 3) pro možnost testování a používání implementovaného algoritmu obecného regulátoru přímo v Matlabu, respektive v Simulinku. Vytvořil jsem dvě verze S-funkcí, přičemž první z nich představuje obecný regulátor pracující ve formátu plovoucí řádové čárky, tedy využívá knihovnu *PSD*. Druhá verze pak používá knihovnu regulátoru pracujícího ve formátu pevné řádové čárky *fixPSD*. Popis implementace bude pro obě verze téměř totožný.

Základní kostru obou verzí S-funkce jsem vytvořil použitím nástroje *S-function Builder* z důvodu rychlejšího definování všech vnitřních parametrů a vstupně-výstupních portů. Zbytek programu, jako je definice základního řídícího algoritmu a inicializace knihovny *PSD*, jsem implementoval ručně. Nutná byla editace metod *mdlInitializeSizes*, *mdlOutputs* a *mdlTerminate*. V inicializační metodě S-funkce *mdlInitializeSizes* probíhá, kromě základního nastavení portů a kontroly vstupních parametrů, i nastavení regulátoru reprezentovaného strukturou *PSD_params*, případně u druhé verze strukturou *fixPSD_params*. Metoda *mdlOutputs* je volána při běhu simulace pro výpočet akčního zásahu regulátoru. Funkce *PSD_control* (*fixPSD_control*) z knihovny *PSD* (*fixPSD*) je pro výpočet řízení použita v externí funkci *PSD_sfun_Outputs_wrapper* (*fixPSD_sfun_Outputs_wrapper*), které se v metodě *mdlOutputs* zadají všechny důležité parametry pro výpočet akčního zásahu. Poslední metodou S-funkce, kterou jsem editoval, je *mdlTerminate*. Zde je zapotřebí uvolnit paměť, která byla v průběhu inicializace dynamicky alokována pro parametry filtrovacích komponent regulátoru, použitím funkce *PSD_terminate* (*fixPSD_terminate*).

Hotovou S-funkci jsem použil v modelu Simulinku s názvem *PSD_simulink_sfun.mdl*, jehož struktura je zobrazena na (obr. 4.4). Model je přímo připraven pro testování regulátoru včetně vytvořených masek bloků s S-funkcemi (viz. kapitola 3) pro jednoduché zadávání testovaných parametrů. Ukázka nastavovacího dialogu regulátoru je na (obr. 4.5). Jedná se o masku k verzi S-funkce pracující v *FX* formátu. U druhé verze dialog neobsahuje kolonky pro nastavení *FX* formátu.



Obrázek 4.4: Schéma Simulink modelu pro testování obecného regulátoru s použitím S-funkcí



Obrázek 4.5: Maska S-funkce obecného regulátoru pracujícího ve formátu pevné řádové čárky

Kapitola 5

PID regulátor

Mým druhým úkolem při realizaci této práce bylo upravit již hotový algoritmus číslicového *PID* regulátoru implementovaného v programovacím jazyce C v číselném formátu plovoucí řádové čárky, který mi byl dodán vedoucím práce a jehož autorem je Ing. Ondřej Špinka. Úprava spočívá v přepracování zdrojového kódu programu tak, že řídící algoritmus bude pracovat v binárním formátu pevné řádové čárky. Využiji tak tedy mnou vytvořenou knihovnu *fixedpoint* (viz. kapitola 2).

Jelikož obecná výuková forma *PID* regulátoru je v praktických úlohách v podstatě nepoužitelná, jeho tvar se určitým způsobem vždy modifikuje. I v tomto případě se jedná o speciální tvar *PID* regulátoru. Je zde použit regulátor s derivační složkou umístěnou ve zpětné vazbě a s tzv. vážením reference. Jeho funkci se v této kapitole pokusím částečně vysvětlit.

V další části textu pak vysvětlím moji implementaci tohoto regulátoru v programovacím jazyce C v *FX* formátu. Správnost funkce ověřím především porovnáním regulace s daty pořízenými z praktické úlohy řízení, kdy dodaný algoritmus *PID* regulátoru byl použit pro řízení skutečného modelu helikoptéry. Pro další testování *PID* regulátoru je vytvořen obecný testovací program, kde je algoritmus kontroléru možné testovat i na jiných úlohách řízení.

Pro možnost použití obou verzí regulátoru pro simulace v Simulinku jsem opět vytvořil S-funkce (viz. kapitole 3), jejichž implementaci a způsob použití zde také popíši.

5.1 Struktura použitého PID regulátoru

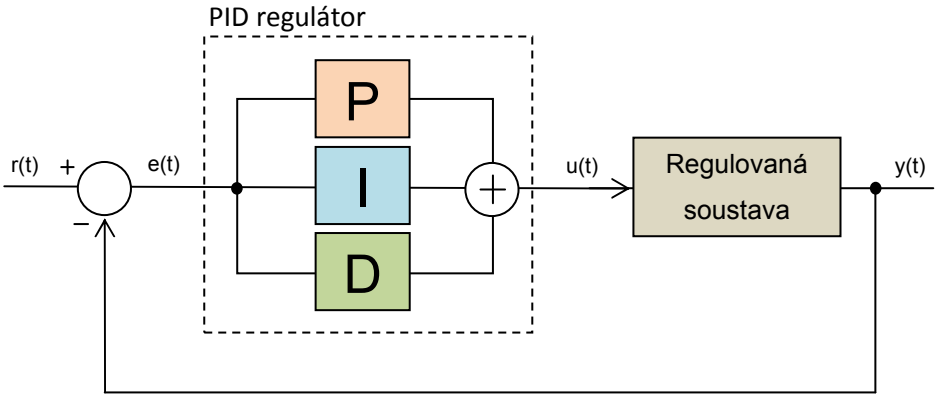
Pro účely řízení diskrétních dynamických systémů je možné použít několik druhů číslicových regulátorů, které vychází z jejich ekvivalentních tvarů ve spojité oblasti. Mezi

nejčastěji používané regulátory pro zpětnovazební řízení patří tzv. třísložkový regulátor, neboli regulátor s proporcionální, integrační a derivační složkou (*PID*).

Obecnou funkci *PID* regulátoru je možné popsat rovnicí (5.1), případně přenosem (5.2). Ukázka zapojení regulačního obvodu s *PID* regulátorem je na (obr. 5.1).

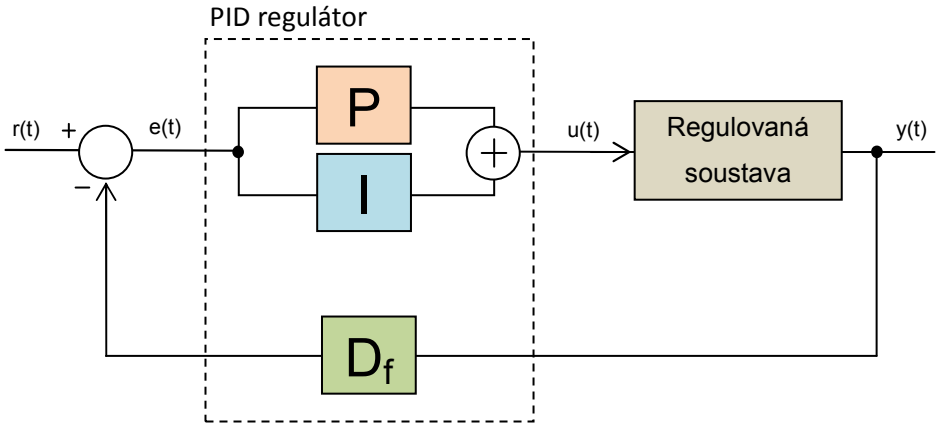
$$u(t) = k_P e(t) + k_I \int_{t_0}^{t} e(\tau) d\tau + k_D \frac{de(t)}{dt} \tag{5.1}$$

$$C(s) = \frac{U(s)}{E(s)} = k_P + \frac{k_I}{s} + k_D s \tag{5.2}$$



Obrázek 5.1: Regulační obvod s *PID* regulátorem

Algoritmus, který mi byl dodán k přepracování, představuje modifikovanou verzi klasického *PID* regulátoru. První úpravou klasického tvaru je přemístění derivační složky regulátoru do zpětné vazby podle (obr. 5.2). Je to obvyklá úprava *PID* regulátorů z důvodu zamezení reakce derivační složky na referenční signál. Pokud by referenční signál obsahoval výrazné vysokofrekvenční složky, tak při použití klasického tvaru *PID* regulátoru podle (obr. 5.1) by derivační složka způsobovala nežádoucí příliš velké akční zásahy. Umístěním *D* složky do zpětné vazby bude derivace počítána z výstupu regulované soustavy. Jelikož k měření výstupního signálu řízené soustavy je zapotřebí použít odpovídající snímač, který do signálu vždy zanáší vysokofrekvenční šum, je nutné tento šum odfiltrovat, aby nedocházelo vlivem derivační složky k jeho zesilování. K filtraci se ve většině případů používá dolnopropustný filtr prvního řádu, který se zařazuje společně s derivační složkou přímo do zpětnovazební smyčky.



Obrázek 5.2: Regulační obvod s *PID* regulátorem s filtrovanou derivační složkou ve zpětné vazbě

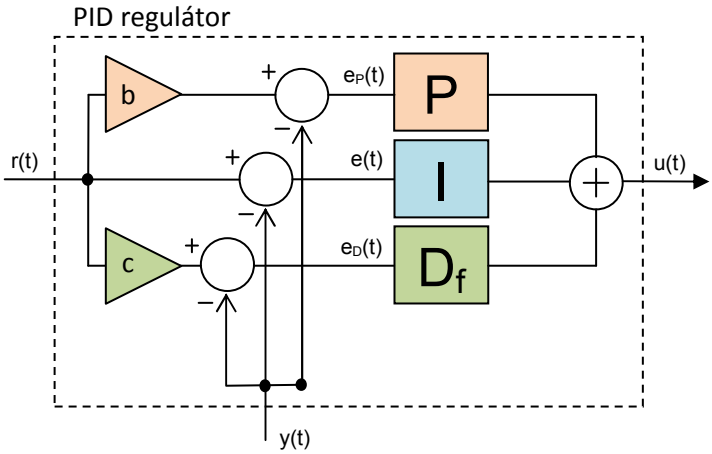
Poslední odlišností dodaného regulátoru od obecné formy *PID* je použití tzv. vážení reference (v anglické literatuře je tento termín označován jako *set-point weighting*). Princip úpravy spočívá v tom, že každá složka regulátoru používá pro výpočet dílčího akčního zásahu svou vlastní regulační odchylku *e*. Proporcionální složka pracuje s regulační odchylkou e_P, která je spočtena podle (5.3). Derivační složka spočte svůj akční zásah na základě odchylky e_D vyjádřené jako (5.4). Koeficienty *b* a *c* se nazývají *váhy*. Regulační odchylka pro integrační složku je vypočtena obvyklým způsobem, tedy jako rozdíl reference a měřeného výstupu. Je to z důvodu zachování specifické vlastnosti integrační složky, a to udržení nulové ustálené odchylky.

$$e_P = b \cdot r - y \tag{5.3}$$

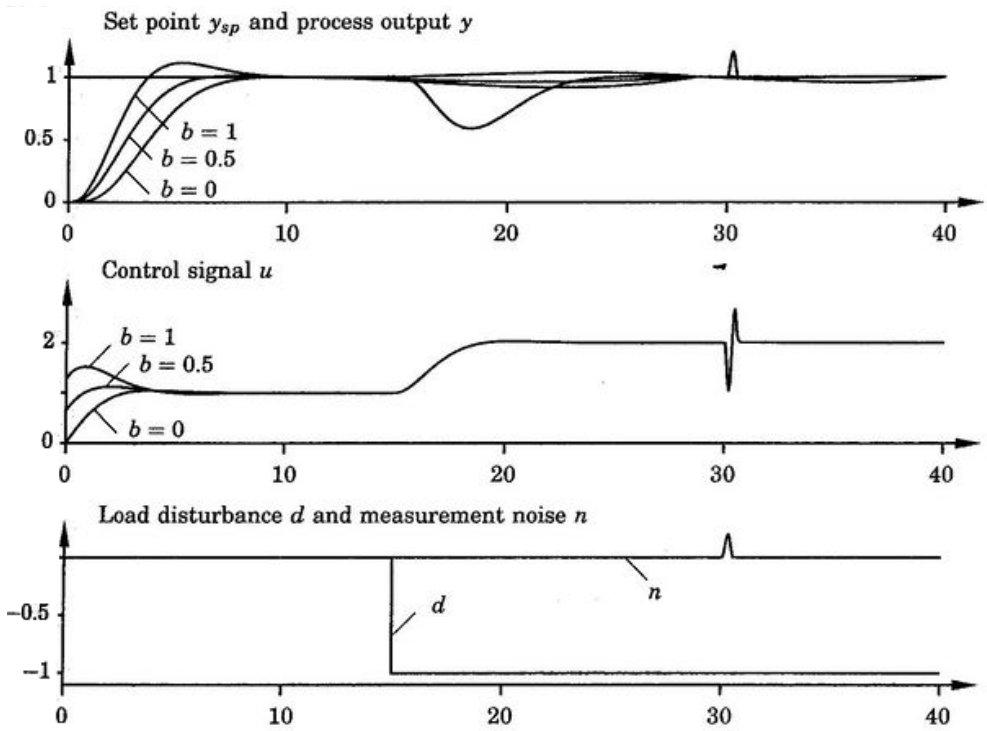
$$e_D = c \cdot r - y \tag{5.4}$$

Strukturu regulátoru s „vážením reference“ je možné vyjádřit ve tvaru podle (obr. 5.3). Změnou hodnot *b* a *c* můžeme měnit váhu závislosti odezvy regulátoru na jednotlivých složkách *PID*. Jejich různé nastavení může zlepšit průběh regulace, při kterém dochází k častému vzniku poruch či zašumění signálů. Ukázka vlivu velikosti *váhy b* na regulační pochod je na (obr. 5.4). Obrázek ukazuje odezvu *PID* regulátoru na jednotkový referenční signál při vzniku poruchy a šumu měření. Parametr *c* je velmi často volen rovný nule, což zaručí pozvolné akční zásahy regulátoru při náhlých změnách reference.

Konečný tvar diskrétní verze *PID* regulátoru s derivační složkou ve zpětné vazbě a tzv. vážením reference, který byl implementován v programovacím jazyce C a mně dodán k dalším úpravám, je možné vyjádřit stavovým modelem podle (obr. 5.5). Tento model dává nejlepší představu, jak tento regulátor pracuje.

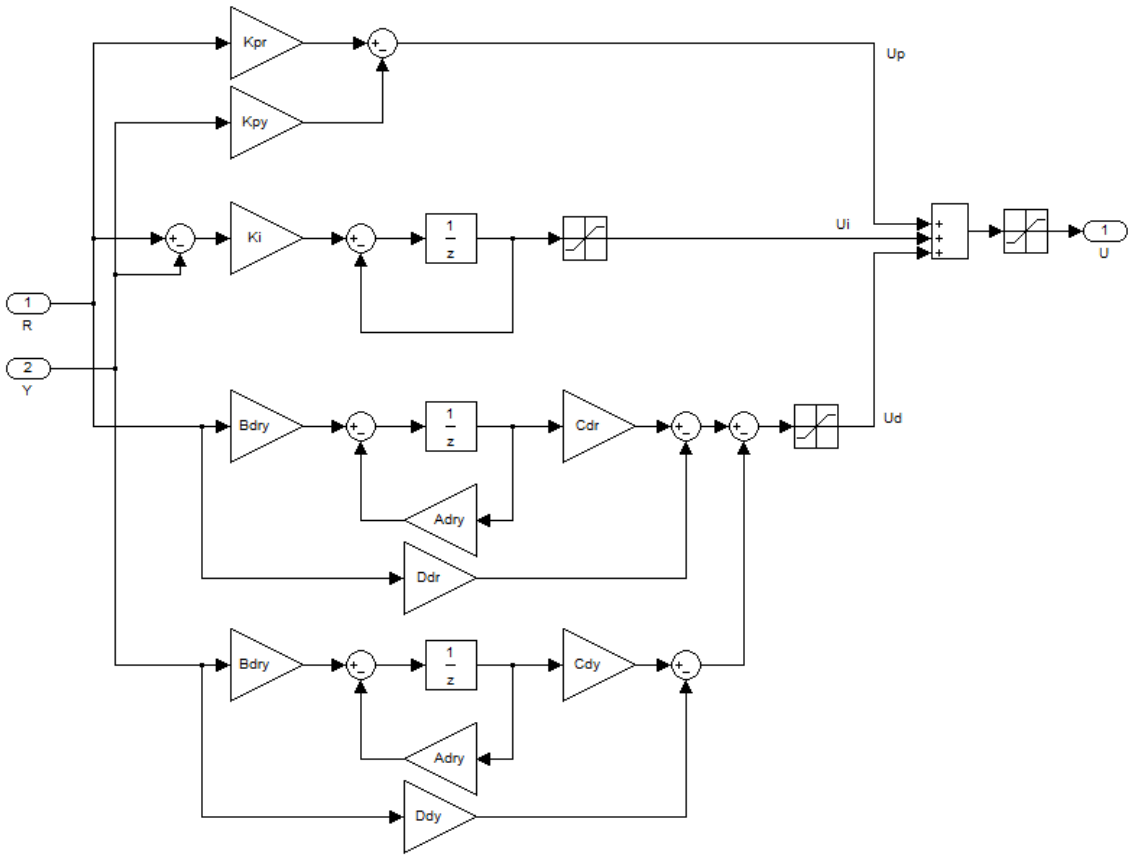


Obrázek 5.3: *PID* regulátor s „vážením reference“



Obrázek 5.4: Vliv parametru *b* na odezvu *PID* regulátoru s „vážením reference“[2]

[2] převzato z [9] *Computer-Controlled Systems : Theory and Design*



Obrázek 5.5: Stavový model *PID* regulátoru s derivační složkou ve zpětné vazbě a „vážením reference“

5.2 Algoritmus PID regulátoru a jeho úpravy

V následující části této kapitoly se již dostávám od teoretického popisu dodaného algoritmu regulátoru k praktické stránce této práce. Nejprve popíši, jakým způsobem byl *PID* regulátor implementován v programovacím jazyce C v binární reprezentaci čísel plovoucí řádové čárky a jak jsem tento program upravil pro obecné používání a testování. Dále se zaměřím zejména na můj hlavní úkol, a to na přepis kódu programu na verzi pracující ve formátu pevné řádové čárky, k čemuž využiji mou knihovnu *fixedpoint* (viz. kapitola 2).

5.2.1 Popis knihovny regulátoru v FP formátu a její úpravy

Knihovna *PID* regulátoru, kterou jsem převzal, měla ve svém obsahu definici struktury pro uchovávání parametrů regulátoru, metodu pro inicializaci a metodu obsahující samotný

řídící algoritmus. Při vytváření verze knihovny pracující ve formátu plovoucí řádové čárky jsem nikterak výrazně zdrojový kód programu neměnil, pouze ho doplnil o další možnosti.

- *Knihovna PID*

```
int PID_reset_params ( PID_params_structure *PID_params,
                       double K,
                       double omega_i,
                       double omega_d,
                       double b,
                       double c,
                       double N,
                       double I_low_bound,
                       double I_high_bound,
                       double D_low_bound,
                       double D_high_bound,
                       double out_low_bound,
                       double out_high_bound,
                       double samp_freq )
```

(5.5)

```
int PID_init ( PID_params_structure *PID_params )
```

(5.6)

```
double PID_control ( double r,
                     double y,
                     double man_action,
                     int auto_control,
                     PID_params_structure *PID_params )
```

(5.7)

Jednou z nejdůležitějších částí knihovny je definice struktury *PID_params_structure*, která slouží k ukládání všech konstant regulátoru, které jsou potřebné pro řídící algoritmus. Tyto konstanty jsou dány stavovým modelem podle (obr. 5.5). Z téhož obrázku je patrné, že struktura musí nést také informace o vnitřních stavech a mezích saturačních omezení celkového i dílčích akčních zásahů regulátoru. Prvním z kroků, které tedy musí uživatel udělat, pokud chce knihovnu *PID* používat pro číslicové řízení, je vytvoření instance této struktury.

Nyní nastává potřeba parametry uvnitř struktury nastavit. K tomuto účelu lze použít hned dvě metody, a to *PID_reset_params* a *PID_init*. Původní knihovna obsahovala pouze metodu *PID_init*, ale z hlediska funkce připomínala spíše nynější metodu *PID_reset_params*. Inicializace konstant regulátoru se v této metodě provádí bez přítomnosti uživatele. Ukazatel na nastavovanou strukturu a parametry pro výpočet všech konstant se funkci zadávají formou vstupních argumentů ve tvaru podle (5.5). V těle funkce *PID_reset_params* se nejprve vstupní parametry zkontrolují několika podmínkami. Pokud minimálně jedné z podmínek

nevyhoví, funkce jako svou návratovou hodnotu vrátí odpovídající chybový kód. V opačném případě se ze vstupních argumentů provede výpočet všech parametrů regulátoru v zadané struktuře *PID_params*.

Druhá inicializační metoda *PID_init* provádí naprosto stejné nastavení zadané struktury, ale tentokrát formou dotazové komunikace s uživatelem. Obsluhovatel programu je nejprve dotázán, zda hodlá zadat všechny parametry regulátoru ručně, nebo zda si přeje načíst konstanty z textového souboru. Zároveň je upozorněn, že v případě použití souboru je nutné zachovat formu vnitřního zápisu parametrů, která je popsána v pomocném souboru *Constants_help.txt*. Pokud je použit textový soubor, parametry jsou z něho načteny automaticky. V případě ručního zadávání je uživatel postupně vyzýván k zadání všech parametrů. Po načtení parametrů je postupováno stejně jako v metodě *PID_reset_params*, tedy nejprve je provedena kontrola všech zadaných hodnot a následně i výpočet a uložení konstant ve struktuře *PID_params*. Tato je předána funkci formou vstupního parametru. Hlavička funkce *PID_init* má tvar podle (5.6).

Poslední metodou knihovny *PID* je *PID_control* ve tvaru podle (5.7) a je přesnou kopií stejnojmenné metody z původní knihovny. Jak již název funkce napovídá, jedná se metodu reprezentující celý algoritmus řízení popsaný stavovým modelem podle (obr. 5.5). Funkce je vždy volána pro výpočet akčního zásahu v jednom aktuálním vzorkovacím kroku pro zadané hodnoty reference r a měřeného výstupu regulované soustavy y. Vstupní proměnnou *auto_control* je definováno, v jakém režimu má regulátor pracovat, zda v automatickém (proměnná *auto_control* má hodnotu různou od nuly), kdy regulátor přejímá veškeré řízení, nebo ručním (*auto_control* je nulové), kdy je řízení kompletně pod kontrolou uživatele. V případě ručního ovládání je pro možnost tzv. beznárazového zapojení regulátoru zpět do regulace nutné funkci předávat informace o manuálním akčním zásahu pomocí proměnné *man_action*. Posledním vstupním argumentem funkce je samozřejmě ukazatel na strukturu s parametry regulátoru *PID_params*. Algoritmus regulátoru je v těle funkce rozdělen do několika částí. Nejprve je spočten podíl proporcionální části regulátoru na velikosti akčního zásahu. Dále je vypočtena odezva derivační části, která je případně omezena saturací, a jsou aktualizovány její vnitřní stavy. Výpočet integrační složky závisí na režimu práce regulátoru, to znamená, jestli pracuje v manuálním nebo automatickém režimu. V případě manuálního režimu je vnitřní stav integrace určen ze zadaného manuálního zásahu a dosud vypočtené proporce a derivace. Regulátor je díky tomu schopen sledovat manuální řízení a při přepnutí do automatického režimu na něj plynule navázat. V dalším kroku algoritmu je proveden výpočet celkového akčního zásahu součtem všech tří složek, tedy proporce, derivace a sumace. Velikost akčního zásahu je v případě potřeby také saturačně omezena. Posledním krokem podmíněným prací regulátoru v automatickém režimu je aktualizace integrační složky a její eventuální omezení.

Tím je tedy popsána funkce celé knihovny *PID*.

5.2.2 Přepis knihovny PID do FX formátu

Popsaná knihovna *PID* pracuje v číselném formátu plovoucí řádové čárky. To znamená, že všechna neceločíselná data jsou reprezentována datovým typem *double*. Mým hlavním úkolem této části práce, bylo upravit algoritmus regulátoru, tak aby pracoval ve formátu pevné řádové čárky. K přepisu jsem samozřejmě využil mou knihovnu *fixedpoint* (viz. kapitola 2).

O přepracování programu z *FP* do *FX* formátu jsem již mluvil v popisu implementace obecného regulátoru (viz. kapitola 4). I zde jsem postupoval stejným způsobem. Pro vytvoření knihovny *fixPID* jsem použil přesnou kopii knihovny *PID*, ale s tím rozdílem, že všechny metody a definici struktury jsem kvůli odlišení přejmenoval přidáním přípony *fix*. Všechny proměnné, které jsou v původní verzi knihovny datového typu *double*, jsou přetypovány na datový typ *fixp* reprezentující číslo v *FX* formátu. Tento datový typ je definován v knihovně *fixedpoint* jako *32*bitový celočíselný typ *integer*. V celé knihovně bylo nutné také nahradit veškeré aritmetické operátory užívané ve výpočtech ekvivalentními funkcemi z knihovny *fixedpoint*. Takto upravený algoritmus řízení definovaný metodou *PID_control*, respektive nyní již *fixPID_control*, se může zdát komplikovaný a nepřehledný, ale v principu funkce se naprosto shoduje s původní implementací.

To jsou tedy všechny základní úpravy, kterými musela původní knihovna *PID* projít, aby byla schopna provádět stejný algoritmus řízení ve formátu pevné řádové čárky.

5.3 Ověření funkce a testování knihoven PID a fixPID

Pro účely testování knihovny *PID* a *fixPID* jsem implementoval testovací program velice podobný tomu, jaký jsem vytvořil pro testování obecného regulátoru (viz. kapitola 4). Program testuje funkci obou knihoven naráz, aby bylo možné porovnávat rozdíly výsledků obou číselných formátů. Jako řízené systémy jsou použity dvě instance modelů (pro každý regulátor vlastní) diskrétních soustav z knihovny *system*, jejichž parametry jsou uživatelsky nastaveny inicializační metodou *system_init*. Inicializací projdou také oba typy regulátoru deklarované formou struktur. Pokud proběhnou všechny inicializace v pořádku, tzn. jejich návratová hodnota je rovna nule, program vyzve uživatele k nastavení *FX* formátu a zadání názvu souboru se vstupními daty, které budou použity jako referenční signál regulátorů. Po jeho zadání testovací program vstoupí do smyčky, kde pro každou hodnotu reference spočte akční zásah každého regulátoru a odpovídající odezvy regulovaných systémů. Výsledky se pro každý krok v přehledné formě vypisují na terminál společně s údajem o absolutní chybě mezi oběma řešeními a zároveň se zapisují do výstupního textového souboru *output.txt* pro možnost dalšího zpracování. Po skončení regulace, kdy na vstupu nejsou již žádná data, se program po potvrzení ukončí.

5.3.1 Porovnání funkce knihoven PID a fixPID se skutečnými daty

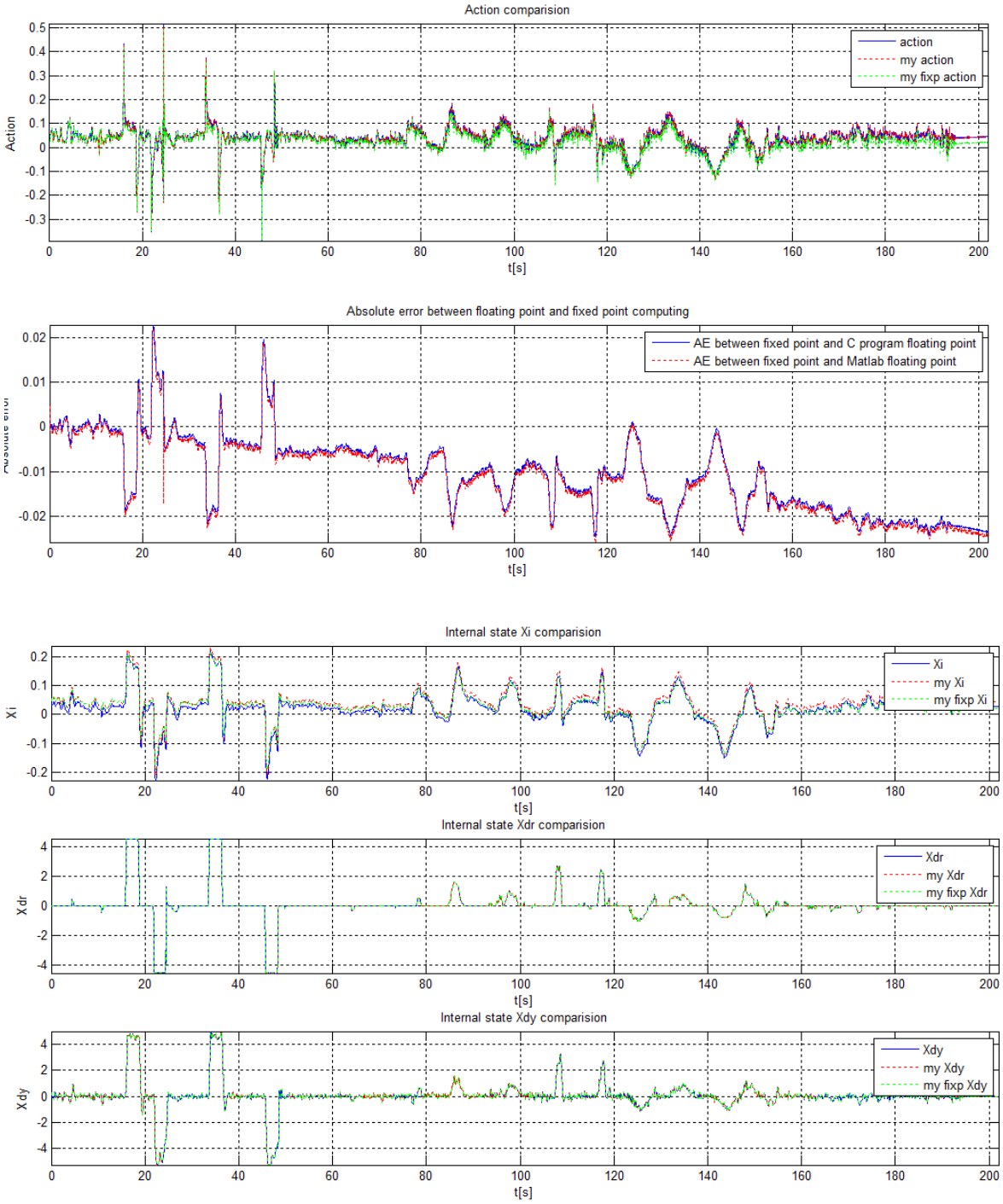
Původní verze regulátoru, která mi byla dodána k přepracování, byla již odzkoušena na skutečné úloze řízení. Algoritmus byl nasazen při úloze řízení letu modelu helikoptéry, přičemž veškerá důležitá data byla nahrávána do paměti počítače. Nenabízí se tedy lepší možnost, jak otestovat správnost funkce mých knihoven, než nasimulování průběhu letu a porovnání regulace s daty pořízenými z reálného modelu.

Pro tyto účely jsem nepatrně modifikoval výše popsaný testovací program. Modifikace spočívá v tom, že se zde nepoužívá knihovna *system* pro reprezentaci regulovaného systému, ale měřený výstup, stejně tak jako referenční signál, se načítají ze souboru dat získaných z části záznamu letu. Regulátor tak bude pracovat v rozpojené zpětné vazbě. Tato data se ve smyčce postupně načtou a použijí se pro výpočet akčních zásahů regulátorů jak ve formátu plovoucí řádové čárky, tak i ve formátu pevné řádové čárky. Výsledky jsou opět přehledně vypisovány na textový výstup programu a zároveň ukládány do souboru *myaction.txt*.

Aby bylo možné zobrazit spočtená data regulace graficky a porovnat je se skutečnými, vytvořil jsem v Matlabu skript *compare_flight.m*. Po jeho spuštění se zobrazí dvě okna s několika grafy. V prvním z grafů se provede porovnání akčních zásahů uložených v souboru *myaction.txt* se skutečnými daty pořízenými za letu. V dalším grafu je graficky znázorněn nárůst absolutní chyby mezi *FP* a *FX* řešením výpočtu akčních zásahů v průběhu regulace. V druhém okně je nakonec ještě provedeno porovnání všech tří vnitřních stavů regulátoru.

Před spuštěním testu je také důležité si uvědomit, že data ze záznamu letu, která jsem dostal k dispozici, nebyla pořízena za nulových počátečních podmínek. V případě, že by byl test prováděn přímo s regulovaným systémem se zapojenou zpětnou vazbou, nemuseli bychom se o počáteční podmínky vnitřních stavů zajímat, jelikož vlivem integrační složky by se v průběhu regulace způsobená chyba vykompenzovala. Jelikož ale tento testovací program pracuje bez zpětné vazby, je nutné nastavit počáteční hodnoty vnitřních stavů ručně přímo v kódu inicializační metody.

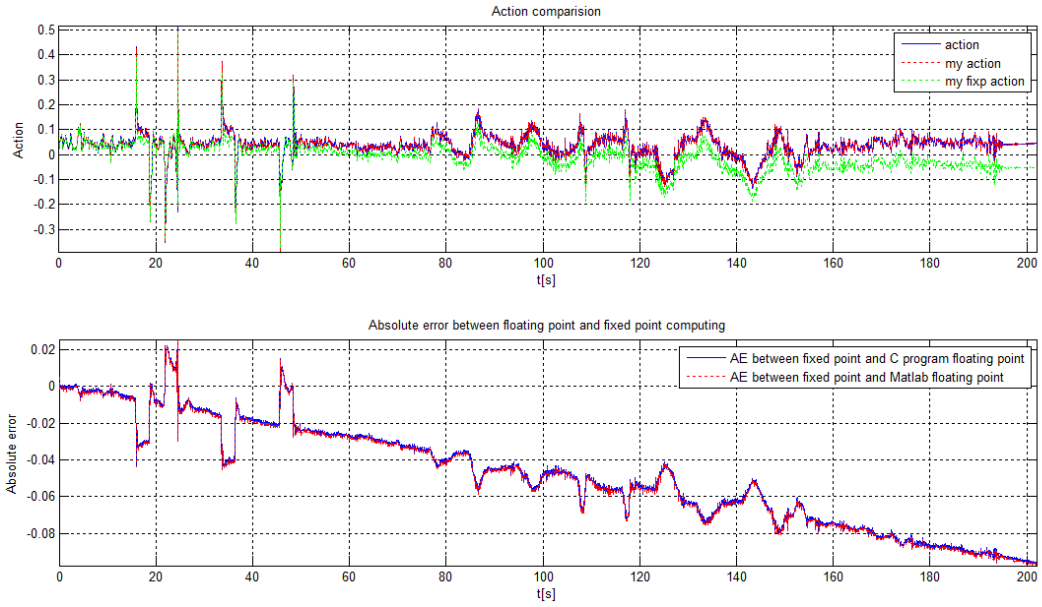
První test jsem provedl pro nastavení formátu pevné řádové čárky *A=12* a *B=20*, tedy vymezení *12* bitů pro celočíselnou část a *20* bitů pro desetinnou část. Výsledek testu je na (obr. 5.6). První graf ukazuje porovnání akčních zásahů. Je vidět, že mezi referenčními daty z letu a akčními zásahy spočtenými v *FP* formátu regulátorem z knihovny *PID* není téměř žádný rozdíl. Obstojně si počíná pro dané nastavení *FX* formátu i regulátor *fixPID*. Nárůst absolutní chyby mezi oběma číselnými formáty ukazuje druhý graf na (obr. 5.6). Z grafu je patrné, že průměrná chyba lineárně narůstá s časem, což je způsobeno akumulací chyb při dílčích výpočtech aritmetiky pevné řádové čárky. Poslední tři grafy na (obr. 5.6) zobrazují porovnání všech tří vnitřních stavů regulátorů. Z výsledků je jasné, že dané nastavení *A* a *B* *FX* formátu je pro tento případ rozumné.



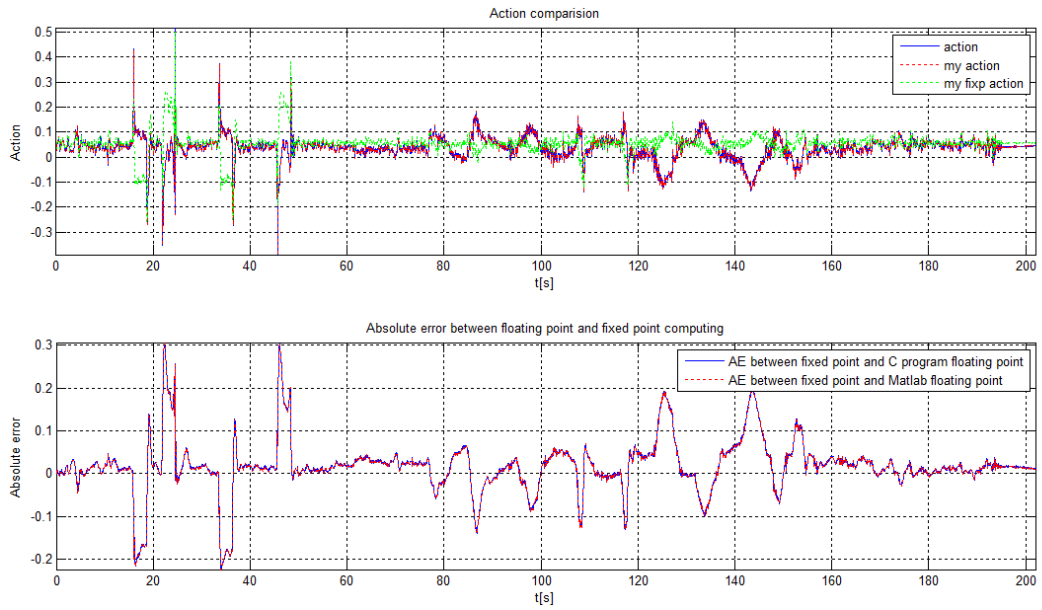
Obrázek 5.6: Porovnání práce regulátorů z knihovny *PID* a *fixPID* pro nastavení *FX* formátu *A=12*, *B=20*

Abych ukázal vliv nastavení formátu pevné řádové čárky na průběh regulace pomocí knihovny *fixPID*, provedl jsem test ještě pro dvě různá nastavení *A* a *B*. Obrázek (obr. 5.7) ukazuje průběh akčního zásahu regulátoru *fixPID* v porovnání s regulátorem *PID* a referencí pro nastavení *A=16* a *B=16*. Zmenšením přesnosti čísla v *FX* formátu se docílilo vyšší rychlosti vzrůstu absolutní chyby v průběhu regulace, což jednoznačně ukazuje druhý graf.

Třetí test jsem provedl pro nastavení *A=20* a *B=12*, jehož výsledky jsou na (obr. 5.8). Extrémní snížení přesnosti a zbytečné zvýšení rozsahu čísel způsobilo na první pohled nevelkou průměrnou absolutní chybu od referenčního řešení, ale při pohledu na průběh akčních zásahů regulátoru *fixPID* je vidět, že průběh naprosto neodpovídá signálu získanému výpočtem ve formátu plovoucí řádové čárky.



Obrázek 5.7: Porovnání práce regulátorů z knihovny *PID* a *fixPID* pro nastavení *FX* formátu *A=16*, *B=16*



Obrázek 5.8: Porovnání práce regulátorů z knihovny *PID* a *fixPID* pro nastavení *FX* formátu *A=20*, *B=12*

5.3.2 S-funkce PID regulátoru

Aby bylo možné používat navržené regulátory praktičtějším způsobem, například pro modelování v Matlabu, vytvořil jsem S-funkce (viz. kapitola 3) pro Simulink, které poskytují možnost využívat knihovny *PID* a *fixPID*. Popis realizace S-funkcí je téměř totožný pro obě verze knihoven regulátoru, proto v následujícím textu budu popisovat implementaci pouze jedné z nich. Na případné odlišnosti upozorním.

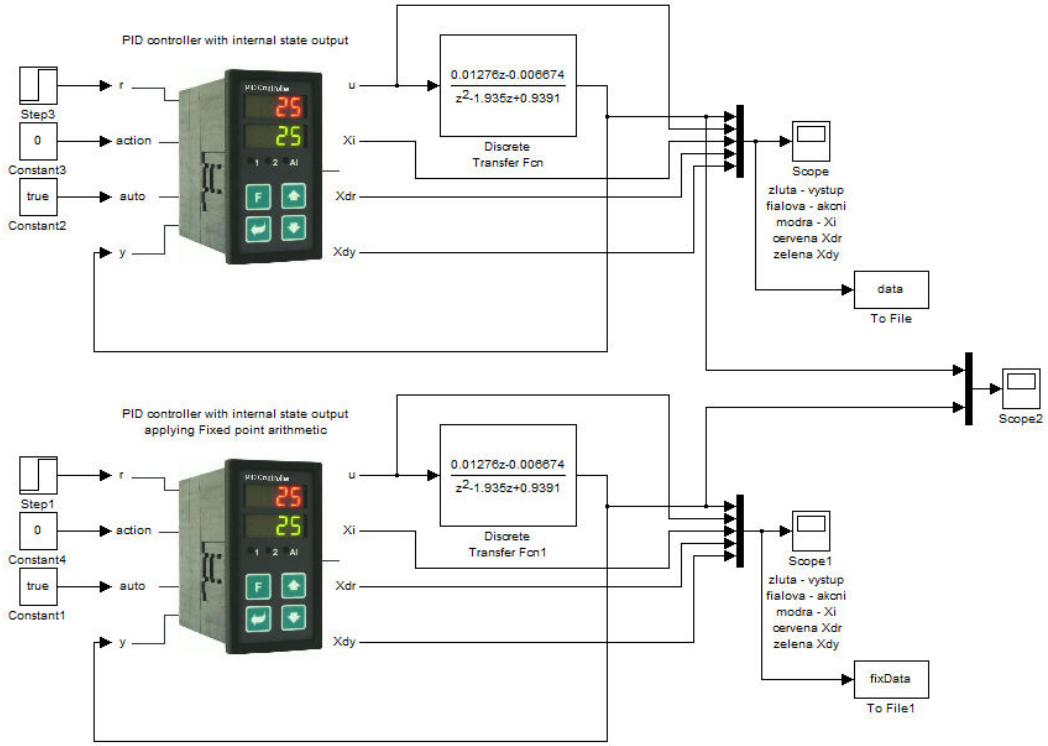
Základní kostru zdrojového kódu S-funkce jsem vytvořil opět s pomocí *S-function Builderu* z důvodu snazšího a rychlejšího definování základní struktury a parametrů S-funkce. S-funkce má definovány čtyři vstupní porty (*r*, *action*, *automatic* a *y*), čtyři výstupní porty (*output*, *Xi*, *Xdr* a *Xdy*) a třináct vnitřních parametrů, odpovídajících konstantám regulátoru. V případě S-funkce používající knihovnu *fixPID* je nutné deklarovat o dva vnitřní parametry více pro definici nastavení *FX* formátu.

Vygenerovaný základní kód S-funkce jsem dále modifikoval ručně. V metodě *mdlInitializeSizes* jsou, kromě základních inicializačních úkonů, načteny hodnoty parametrů a použity pro výpočet konstant regulátoru, které jsou po kontrole uloženy ve struktuře *PID_params* (*fixPID_params*) voláním funkce *PID_reset_params* (*fixPID_reset_params*). V případě S-funkce realizující regulátor *fixPID* je v této metodě inicializována i knihovna *fixedpoint* voláním její funkce *fixInit*.

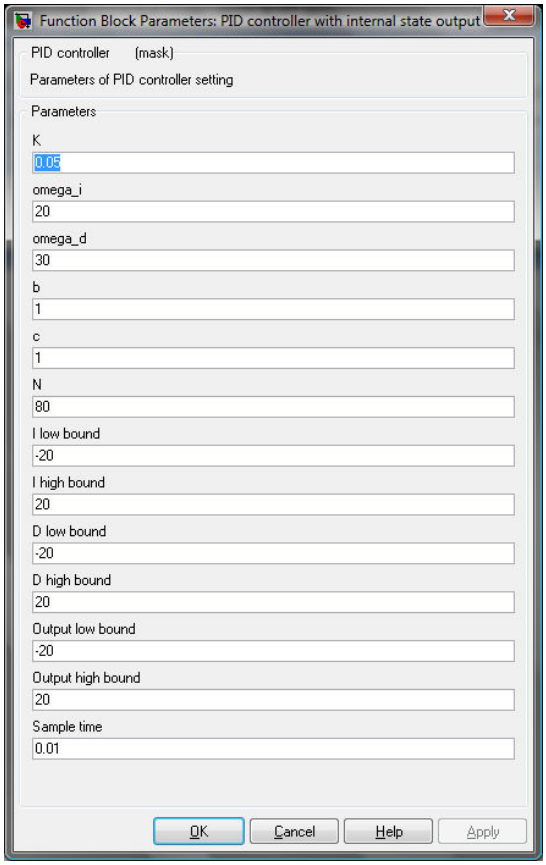
Druhou metodou, kterou bylo nutné nepatrně upravit, bylo *mdlInitializeSampleTimes*, kde bylo nutné nastavit periodu vzorkování podle hodnoty příslušného parametru.

V metodě *mdlOutputs* je volána standardní externí funkce *PID_sfun_Outputs_wrapper* (*fixPID_sfun_Outputs_wrapper*), které se předávají ukazatelé na paměťová místa všech vstupně-výstupních portů a struktury uchovávající informace o regulátoru. Tělo této externí funkce pak obsahuje řídící algoritmus regulátoru formou metody *PID_control* (*fixPID_control*) z knihovny *PID* (*fixPID*).

Obě hotové a zkompilované verze systémových funkcí jsem použil v obecném simulinkovém modelu uloženým jako *PIDcontroller.mdl*, jež lze využít pro modelování široké škály řídících obvodů. Ukázka modelu je na (obr. 5.9). Jak je z obrázku vidět, pro bloky představující implementované S-funkce jsem připravil i uživatelské masky (viz. kapitola 3) pro snadnější ovládání a používání regulátorů. Dvojitým kliknutím levým tlačítkem myši na blok S-funkce se otevře dialogové okno (obr. 5.10), s jehož pomocí lze jednoduše měnit parametry každého regulátoru.



Obrázek 5.9: Schéma Simulink modelu pro testování PID regulátorů s použitím S-funkcí



Obrázek 5.10: Maska S-funkce realizující *PID* regulátor

Kapitola 6

Kalmanův filtr

V předposlední kapitole této práce pojednám o mém posledním úkolu, a to o implementaci algoritmu Kalmanova filtru v programovacím jazyce C. Jelikož problematika Kalmanova filtru částečně přesahuje hranice mých vědomostí, základní nástin řešení algoritmu formou Matlab skriptu mi byl dodán vedoucím práce. Podmínkou implementace, je navrhnout daný algoritmus jak v binární reprezentaci čísel formátu plovoucí řádové čárky, tak i formátu řádové čárky pevné.

V první části kapitoly se pokusím vysvětlit důvody použití a základní princip funkce Kalmanova filtru, z něhož jsem vycházel při implementaci knihovny v jazyce C. Popisu navržené knihovny realizující Kalmanův filtr věnuji samozřejmě další podstatnou část kapitoly. Jak ukážu v první části textu, Kalmanův filtr je většinou navrhován ze stavového popisu sledovaného diskrétního systému, který je reprezentován nejčastěji odpovídajícími stavovými maticemi. Z toho důvodu je nutné vytvořit také pomocnou knihovnu pro práci s velikostně neomezenými maticemi v jazyce C v obou číselných formátech. Tím vznikne skutečně univerzální systém vhodný pro použití v úlohách regulace a řízení.

Testování probíhá velice podobným způsobem jako u předchozích regulátorů *PID* a *PSD* (viz. kapitola 4 a 5). Implementoval jsem jednoduchý testovací program, který algoritmus Kalmanova filtru aplikuje na libovolný diskrétní systém. Výsledky je možné porovnávat se simulací Matlabu znovu pomocí připraveného skriptu. Jak již u této práce začíná být pravidlem, pokusím se implementovat také S-funkci, která poskytne možnost knihovnu filtru využívat pro možnosti simulace v Simulinku.

6.1 Popis funkce Kalmanova filtru

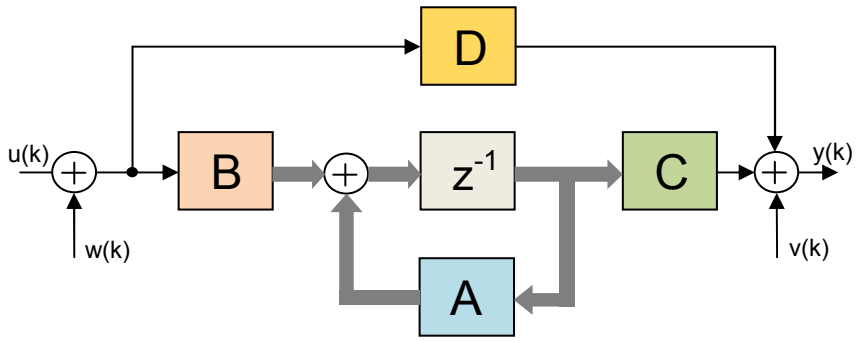
Kalmanův filtr, někdy nazýván také jako pozorovatel stavů nebo estimátor stavů, je zajímavý nástroj použitelný jak v řídící, tak i v měřicí technice.

V oblasti měřicí techniky se Kalmanova filtru využívá nejčastěji k filtraci zašuměných stacionárních harmonických signálů, ale dovoluje filtrovat i signály kvazistacionární. Výhodou filtrace je to, že se provádí přímo v časové oblasti bez nutnosti převodu do oblasti frekvenční (*FFT*), a to kvalitativně srovnatelným způsobem v porovnání s klasickými filtračními algoritmy.

V oboru řídící techniky se Kalmanův filtr používá pro účely řízení jako tzv. pozorovatel stavů. V případě, že chceme řídit diskrétní systém například metodou stavové zpětné vazby, ale vnitřní stavy systému jsou z nějakého důvodu neměřitelné, přichází na řadu právě použití pozorovatele, který je schopen na základě vstupních a výstupních signálů řízeného systému odhadnout jeho vnitřní stavy, které jsou následně použity pro řízení.

Jak již bylo řečeno výše, Kalmanův filtr je schopen předpovídat chování dynamického systému na základě historie jeho vstupních a výstupních dat. Princip funkce je založen na průměrování odchylek naměřených hodnot od odhadovaných hodnot a jejich nejistot. Algoritmus pak běží ve smyčce, kdy je nejprve proveden odhad a po něm následuje jeho korektura podle skutečných hodnot.

Návrh filtru vyhází ze stavového modelu lineárního dynamického diskrétního systému podle (obr. 6.1) popsaného stavovými rovnicemi (6.1), jehož vstup a výstup je zatížen bílými šumy *w(k)* a *v(k)*. Tyto bílé šumy lze vyjádřit kovariančními maticemi Q a R definovanými podle (6.2), které udávají míru důvěryhodnosti odhadu stavů. Poměr těchto matic udává podle (6.3) tzv. Kalmanovo zesílení L. Toto zesílení je někdy označováno také pojmem „matice injekce z výstupu“.



Obrázek 6.1: Stavový model obecného diskrétního systému

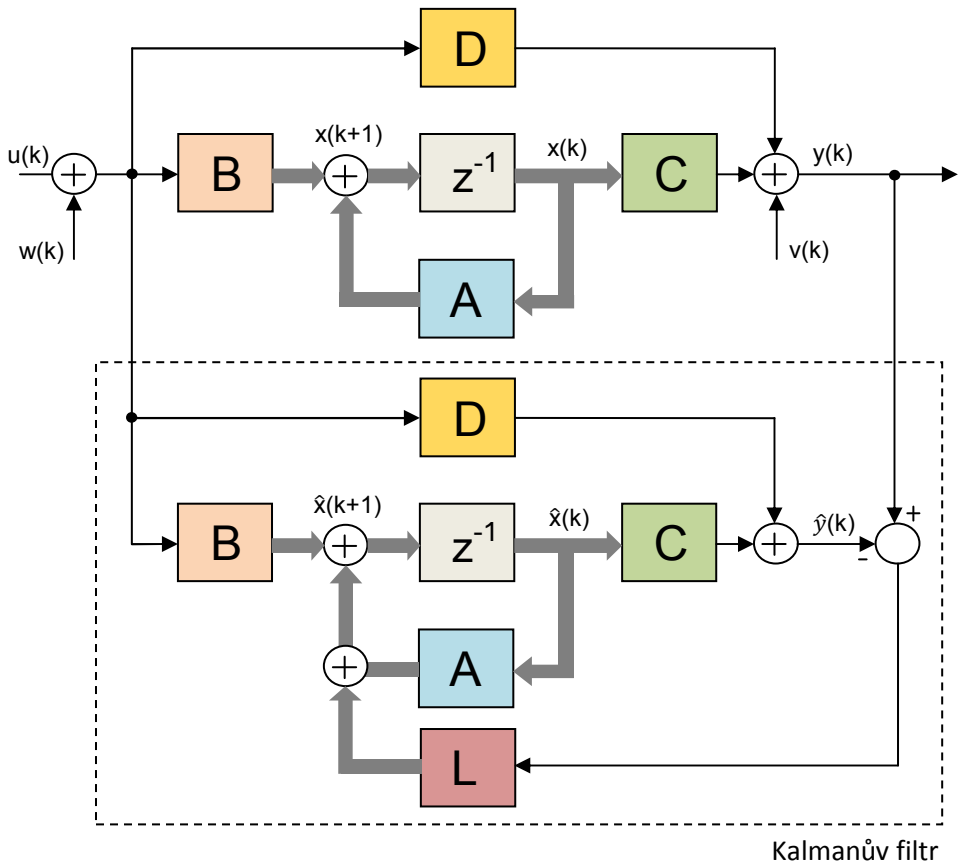
$$x(k+1) = \boldsymbol{A} \cdot x(k) + \boldsymbol{B} \cdot (u(k) + w(k))$$
$$y_{noise}(k) = \mathbf{C} \cdot \mathrm{x(k)} + \mathbf{D} \cdot \big(\mathrm{u(k)} + \mathrm{w(k)}\big) + \mathrm{v(k)} \tag{6.1}$$

$$\begin{pmatrix} w(k) \\ v(k) \end{pmatrix} \cdot (w(k) \quad v(k)) = \begin{pmatrix} Q & 0 \\ 0 & R \end{pmatrix} \tag{6.2}$$

$$L = Q \cdot R^{-1} \tag{6.3}$$

Kalmanův filtr lze definovat jako diskrétní systém, který lze popsat rovnicemi (6.4). Parametry tohoto systému, tedy matice *A*,*B*,*C*,*D*, jsou stejné jako parametry pozorované soustavy. Systém Kalmanova filtru se liší pouze přidáním tzv. injekce z výstupu do výpočtu vnitřních stavů, což je odchylka mezi skutečným a odhadovaným výstupem obou soustav, která je násobena Kalmanovým zesílením *L*. Navržený Kalmanův filtr se formou stavového modelu k pozorovanému systému připojí způsobem ukázaným na (obr. 6.2).

$$\begin{gathered} \hat{x}(k+1) = \boldsymbol{A} \cdot \hat{x}(k) + \boldsymbol{B} \cdot u(k) + \boldsymbol{L} \cdot (y_{noise}(k) - \boldsymbol{C} \cdot \hat{x}(k) - \boldsymbol{D} \cdot u(k)) \\ \hat{y}(k) = \boldsymbol{C} \cdot \hat{x}(k) + \boldsymbol{D} \cdot u(k) \end{gathered} \tag{6.4}$$



Obrázek 6.2: Stavové schéma Kalmanova filtru
a jeho připojení k pozorované soustavě

Pro návrh Kalmanova filtru, respektive určení koeficientů matice injekce, lze použít Matlab a jeho funkci *kalman*. Tato funkce vypočte ze zadaných parametrů pozorovaného systému a kovariančních matic šumů Q a R přenos navrženého pozorovatele stavů a matici zesílení L.

Při implementaci algoritmu Kalmanova filtru v programovacím jazyce C budu vycházet z rovnic jeho stavového popisu (6.4).

6.2 Implementace Kalmanova filtru

V následující části této kapitoly popíši mou implementaci knihovny realizující Kalmanův filtr v programovacím jazyce C. Stejně jako tomu bylo v kapitolách o obecném i *PID* regulátoru (viz. kapitola 4 a 5), nejprve popíši implementaci ve formátu plovoucí řádové čárky. Z této verze pak budu vycházet při přepisu do formátu pevné řádové čárky.

6.2.1 Implementace ve formátu plovoucí řádové čárky

Jelikož při realizaci Kalmanova filtru v programovacím jazyce C vycházím z jeho stavového popisu podle (6.4), kde je důležitá maticová aritmetika, bylo před samotnou implementací nejprve nutné vytvořit speciální pomocnou knihovnu pro práci s maticemi. Tato knihovna má název *matrix*. S její pomocí jsem mohl přistoupit přímo k tvorbě knihovny *kalman*, která reprezentuje samotný Kalmanův filtr.

- *Knihovna matrix*

```
Matrix new_matrix ( mx_size size )
```
(6.5)

```
void free_matrix ( Matrix_structure *mat )
```
(6.6)

```
Matrix_structure transpose_matrix ( Matrix_structure mat )
```
(6.7)

```
Matrix_structure mul_matrix ( Matrix_structure first,
                              Matrix_structure second )
```
(6.8)

```
Matrix_structure dotprod_matrix ( Matrix_structure first,
                                  Matrix_structure second )
```
(6.9)

```
Matrix_structure add_matrix ( Matrix_structure first,
                              Matrix_structure second )
```
(6.10)

```
Matrix_structure sub_matrix ( Matrix_structure first,
                              Matrix_structure second )
```
(6.11)

Knihovna *matrix* představuje užitečný nástroj pro práci s maticemi v programovacím jazyce C, kterou bylo nutné implementovat pro účely Kalmanova filtru.

V knihovně jsou definovány vlastní datové typy *Matrix* a *mx_size*. *Matrix* představuje dvojrozměrný ukazatel typu *double* ukazující na paměťová místa s uloženými prvky matice. Datový typ *mx_size* reprezentuje pole o dvou prvcích, které slouží k uchování informace o rozměru matice. Jelikož při maticovém počtu je zapotřebí znát jak samotné matice, tak i jejich rozměry, k reprezentaci matice v knihovně *matrix* je použita definovaná struktura *Matrix_structure*, která všechny informace o matici uchovává.

Aby bylo možné vytvářet libovolně velké matice, je v knihovně definována metoda *new_matrix* podle (6.5). Tato metoda provede alokaci paměti pro matici podle zadané velikosti *size*. Jako výsledek je vrácen ukazatel na alokované místo formou datového typu *Matrix*. Jelikož matice jsou alokovány dynamicky, to znamená až po kompilaci, je nutné alokovanou paměť po skončení práce uvolnit. K tomuto účelu slouží metoda *free_matrix* ve tvaru podle (6.6). Této funkci se jako vstupní parametr předává ukazatel na strukturu *Matrix_structure*, která obsahuje matici, jejíž alokovanou paměť je potřeba uvolnit.

Nyní se již popis bude týkat metod realizujících samotné operace s maticemi. Knihovna obsahuje většinu potřebných operací kromě výpočtu inverzní matice, což je pro implementaci v jazyce C relativně náročná operace. Všechny následující metody vrací jako výsledek operace strukturu *Matrix_structure*.

První definovanou operací je metoda *transpose_matrix*. Její hlavička je ve tvaru podle (6.7). Tato metoda provádí transpozici zadané matice *mat*. Transpozice je jednoduchou operací pro implementaci. Je využito dvojitého *for* cyklu, který provede přesun prvků zadané matice do transponované v souladu s transpozičním pravidlem, kdy řádky matice jsou nahrazeny sloupci a naopak.

Další metodou je *mul_matrix* (6.8) reprezentující násobení dvou matic. U této operace je nutné kontrolovat rozměry obou násobených matic, kdy musí být počet sloupců první matice roven počtu řádků matice druhé. Pokud tato podmínka splněna není, je uživatel o této skutečnosti informován chybovou hláškou a jako výsledek je vrácena prázdná matice. V opačném případě je alokováno odpovídající místo pro výslednou matici a je proveden samotný výpočet podle pravidel pro součin dvou matic.

Součin dvou matic provádí i metoda *dotprod_matrix* (6.9). Tentokrát se ale jedná o tzv. *dot product*, tedy operaci, kdy je součin proveden po prvcích. Z toho vyplývá i podmínka kontroly zadaných maticových operandů, kdy se jejich rozměry musí sobě rovnat. Pokud ne, je vrácena společně s chybovým hlášením opět prázdná matice. Algoritmus výpočtu je

samozřejmě mnohem jednodušší než tomu bylo u metody *mul_fix*. Jsou použity opět dva *for* cykly, ve kterých se postupně násobí prvky s odpovídajícími souřadnicemi.

Naprosto stejným způsobem jako předchozí metoda jsou implementovány i poslední dvě metody *add_matrix* a *sub_matrix* pro součet a rozdíl dvou matic. Jediný rozdíl je ve změně aritmetických operátorů, kdy násobení je nahrazeno součtem, respektive rozdílem. Hlavičky obou metod jsou ve tvaru podle (6.10) a (6.11).

- *Knihovna kalman*

```
int kalman_init ( Kalman_params_structure *kalman_params )
```
(6.12)

```
Matrix_structure kalman_estimate (
                                Matrix_structure u,
                                Matrix_structure y,
                                Kalman_params_structure *kalman_params )
```
(6.13)

```
void kalman_terminate ( Kalman_params_structure *kalman_params )
```
(6.14)

Nyní se již dostávám k hlavní a podstatné části této kapitoly, a to k popisu implementace knihovny *kalman*, která realizuje algoritmus Kalmanova filtru.

Parametry filtru se opět uchovávají ve formě definované struktury s názvem *Kalman_params_structure*. V nitru struktury jsou deklarovány matice A,B,C,D, obsahující informace o filtrovaném diskrétním systému. Dále matice L, reprezentující Kalmanovo zesílení. Nutné je také rezervovat paměť pro historii vstupních a výstupních dat (matice *prev_u* a *prev_y*) a samozřejmě i matici X pro odhad vnitřních stavů.

První z metod knihovny je *kalman_init* (6.12) pro inicializaci parametrů filtru uložených ve struktuře *kalman_params* předávané funkci formou ukazatele. Inicializační metoda má velice podobnou implementaci jako inicializační funkce knihoven obecného a *PID* regulátoru (viz. kapitola 4 a 5). Nastavení parametrů probíhá formou komunikace s obsluhovatelem programu. Uživatel se nejprve rozhodne, zda hodlá zadat parametry ručně, nebo zda nechá program parametry automaticky načíst z textového souboru. V případě, že se rozhodne pro automatické nastavení, musí zadat název souboru obsahující údaje o filtru. Uvnitř souboru musí být tyto informace zapsány opět v předepsaném pořadí a formátu popsaném ukázkovým souborem *Parameters_help.txt*. Po zadání správného názvu souboru jsou všechny parametry postupně, bez nutnosti jakéhokoliv zásahu načteny a podle nich vytvořeny i odpovídající matice pro strukturu *kalman_params*. Při načítání jsou parametry kontrolovány a v případě nesrovnalostí funkce vypíše chybové hlášení a jako výsledek inicializace vrátí odpovídající chybový kód. Při ručním zadávání je postup obdobný, pouze s tím rozdílem, že k zadání jednotlivých parametrů je uživatel vyzýván programem. Pokud inicializace struktury

kalman_params proběhne v pořádku, metoda ukončí svou činnost s návratovou hodnotou rovnou nule.

Druhá metoda knihovny *kalman* pod názvem *kalman_estimate* (6.13) provádí samotný algoritmus Kalmanova filtru podle vztahu (6.4). Funkci se zadávají matice u a y, které reprezentují aktuální hodnoty vstupního a měřeného výstupního signálu filtrovaného systému. Zároveň také funkce dostává informace o nastavení Kalmanova filtru formou ukazatele na strukturu *kalman_params*. V těle metody je prováděn výpočet odhadu přesně podle vztahu (6.4) za pomoci maticových operací definovaných v knihovně *matrix*. Pouze z důvodu přehlednosti jsem výpočet rozdělil do několika částí. Po provedení výpočtu metoda vrátí hodnotu odhadované filtrace výstupního signálu soustavy.

Poslední metodou je *kalman_terminate*. Jelikož jsou paměťové prostory pro matice ze struktury *Kalman_params_structure* alokovány až za běhu programu, je nutné alokovanou paměť po skončení práce uvolnit voláním metod *free_matrix* z knihovny *matrix*. Hlavička této metody je uvedena v (6.14).

6.2.2 Implementace ve formátu pevné řádové čárky

Jelikož způsob přepisu zdrojového kódu knihoven *kalman* a *matrix* do formátu pevné řádové čárky je naprosto stejný jako v případě knihoven obecného i *PID* regulátoru (viz. kapitola 4 a 5), bylo by zbytečné ho dopodrobna znovu vysvětlovat. Proto popíši jen základní úpravy.

Pro konverzi se použijí kopie již implementovaných knihoven *kalman* a *matrix*, které se pro odlišení od původní verze přejmenují na *fixKalman* a *fixMatrix*. Příponu *fix* dostanou i názvy všech definovaných metod a struktur z důvodu zabránění kolize s metodami původních knihoven při jejich současném použití v praktických programech. Základ přepracování do *FX* formátu spočívá hlavně v zakomponování použití knihovny *fixedpoint*. Všechny proměnné typu *double* použité v původních knihovnách se v nové verzi přetypují na datový typ *fixp*. Zbytek úprav se zakládá na použití metod aritmetických operací formátu pevné řádové čárky z knihovny *fixedpoint*, kterými se nahradí původní aritmetické operátory +, -, /, * podporované formátem plovoucí řádové čárky.

6.3 Ověření a testování funkce knihoven kalman a fixKalman

Nezbytnou součástí mé práce je samozřejmě ukázat správnost funkce mých implementovaných knihoven. Ani v tomto případě neudělám výjimku.

Funkci ukážu pomocí jednoduchého testovacího programu, který jsem v jazyce C implementoval. Tento program je schopen testovat pozorovací i filtrovací funkci obou

implementovaných verzí Kalmanova filtru. Jak je již u mých programů zvykem, pro grafické znázornění výsledků testu jsem vytvořil pomocné skripty v Matlabu.

Po spuštění programu je uživatel dotázán, zda si přeje aplikovat algoritmus Kalmanova filtru na modelu soustavy, reprezentovaném knihovnou *system*, nebo zda chce k simulaci použít sadu dat uloženou v textovém souboru. V případě použití knihovny *system* se bude jednat o testování pozorovací schopnosti knihovny *kalman*, jelikož data odezvy systému nebudou zatížena šumem. Zarušený výstupní signál je možné simulovat použitím textového souboru, ve kterém si data můžeme volit libovolně, například exportem z Matlabu.

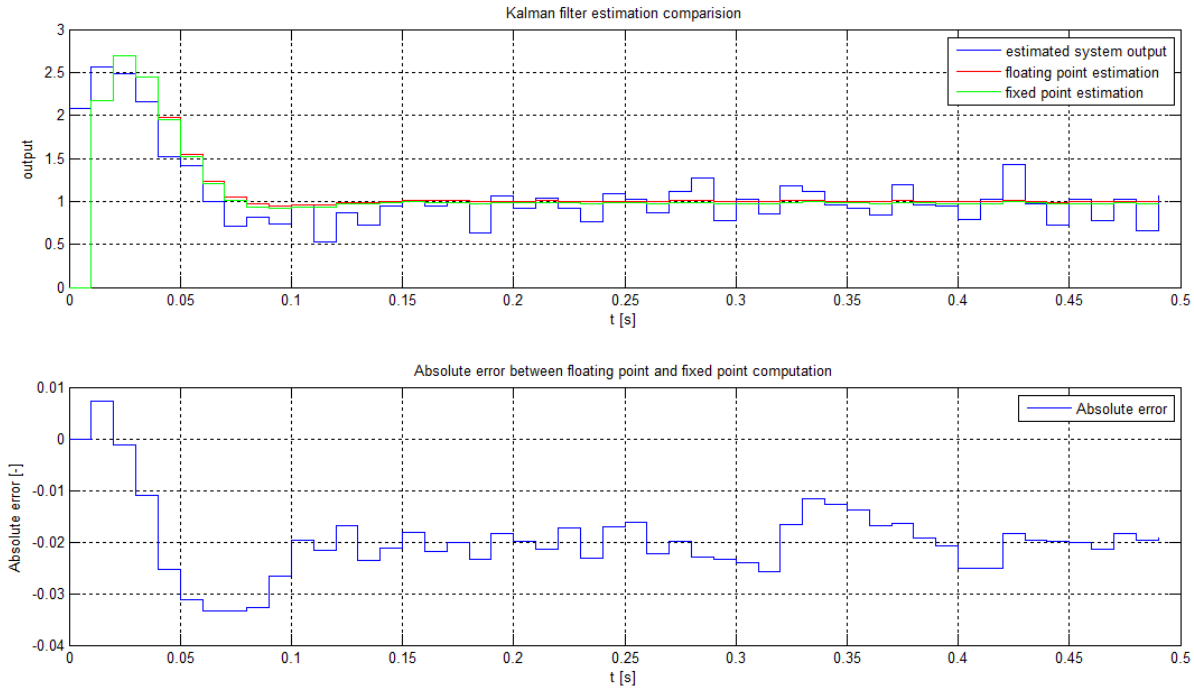
Podle toho, jakou možnost si uživatel zvolí, se provede či neprovede inicializace knihovny *system*, tedy modelu diskrétní soustavy. Před spuštěním inicializačních metod obou verzí Kalmanova filtru je nutné nastavit formát pevné řádové čárky. Nastavení uživatel provede zadáním poměru celé a desetinné části *FX* formátu. Po jeho úspěšném nastavení se rozběhnou inicializační funkce *kalman_init* a *fixKalman_init*, které za účasti uživatele provedou nastavení vnitřních parametrů struktur *kalman_params* a *fixKalman_params*.

Nyní je již vše připraveno k simulaci. Program se dotáže na jméno souboru s daty reprezentujícími vstupní signál a v závislosti na počáteční volbě případně i na jméno souboru obsahující výstupní data filtrované soustavy. V simulační smyčce jsou vzorky ze souborů postupně načítány nebo případně spočteny a použity pro výpočet odhadu Kalmanova filtru, a to jak v *FP* formátu, tak i v *FX* formátu. Výsledky se jako obvykle vypisují na terminál a zapisují do výstupního textového souboru pro další využití. Simulace běží až do vyčerpání všech dat ve vstupních souborech.

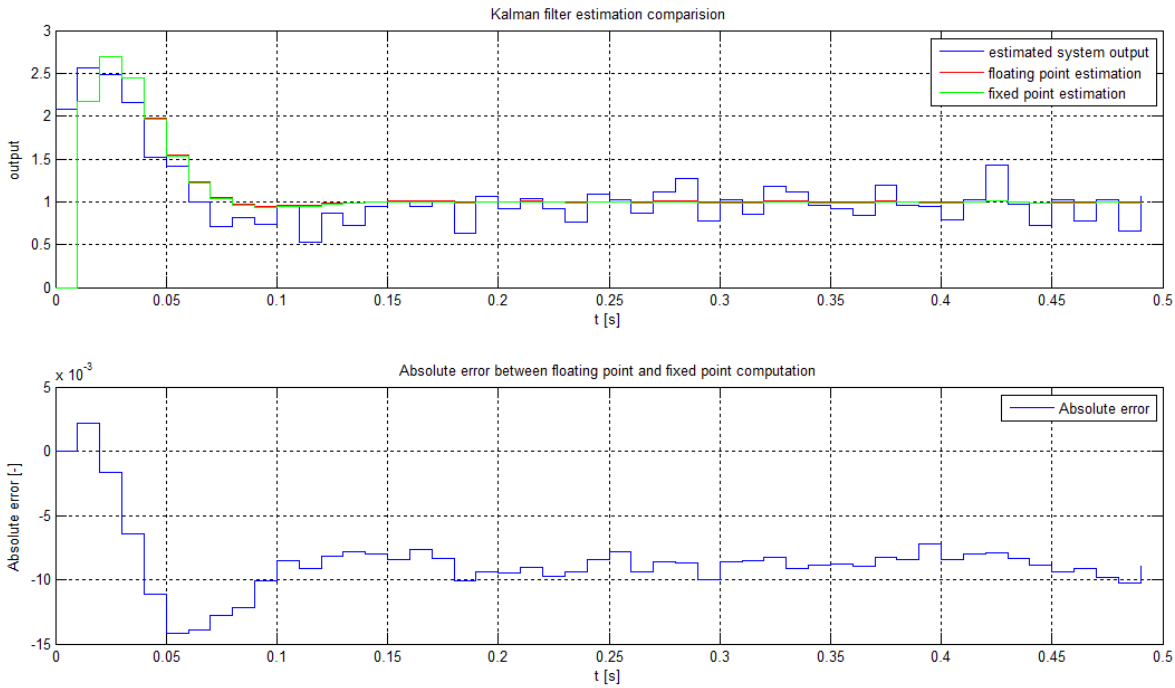
Před ukončením celého testovacího programu se zavolají ukončovací funkce *kalman_terminate* a *fixKalman_terminate* pro uvolnění dynamicky alokované paměti.

Pro ukázku funkčnosti filtrace implementovaného algoritmu Kalmanova filtru jsem použil data zarušeného signálu odezvy diskrétního systému popsaného přenosem (6.15). Tento signál jsem vyexportoval z Matlabu skriptem *kalman_generate.m*, který krom toho provádí i návrh Kalmanova filtru pro daný systém. Navržené parametry filtru jsem použil pro simulaci. Výsledky je možné graficky znázornit pomocí připraveného skriptu *compare_kalman.m*. Řešení filtrace pomocí knihoven *kalman* a *fixKalman* není v tomto skriptu porovnáváno s řešením Matlabu, jelikož správnost funkce je patrná již z průběhu filtrovaného signálu. Výsledek filtrace pro výše uvedené parametry simulace pro nastavení *FX* formátu *A=16* a *B=16* je na (obr. 6.3). Z obrázku je patrné, že filtrace opravdu proběhla úspěšně a rozdíly mezi řešením spočteným *FP* a *FX* aritmetikou jsou téměř zanedbatelné. To, jaký má vliv nastavení *FX* formátu na průběh filtrace pomocí knihovny *fixKalman*, ukazují (obr. 6.4), (obr. 6.5) a (obr. 6.6). Z těchto obrázků je dobře vidět, že chování filtru je uspokojivé i při výrazných nepoměrech nastavení celé a desetinné části *FX* formátu.

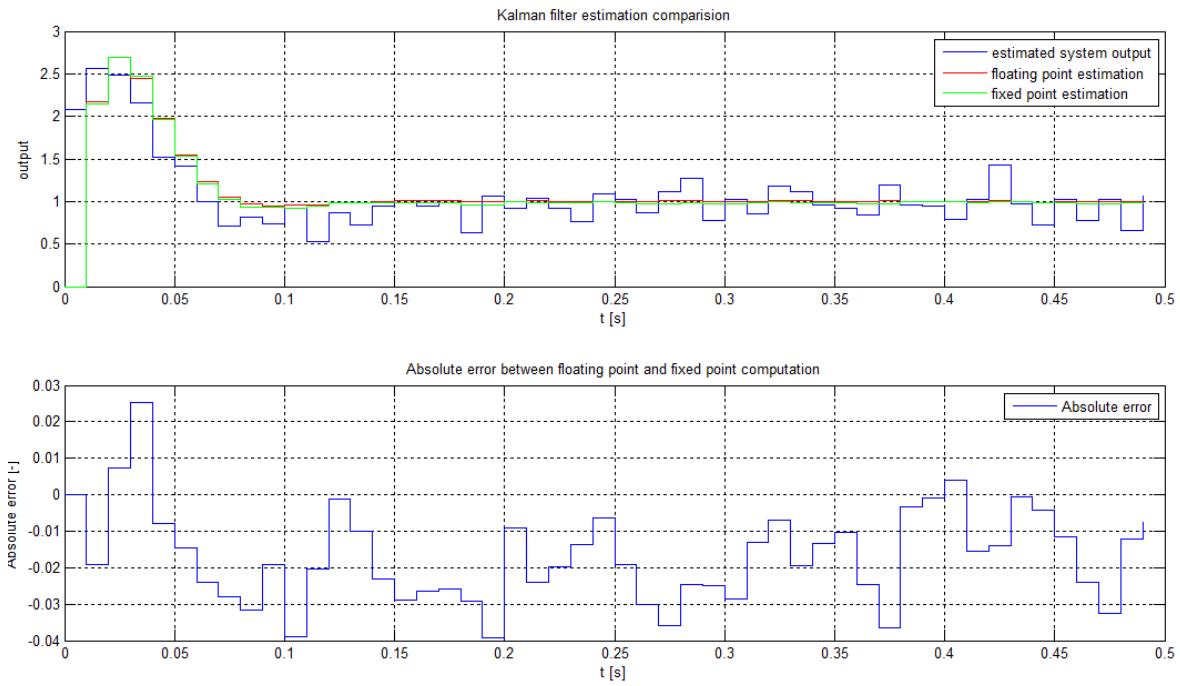
$$S = \frac{2{,}094z - 1{,}841}{z^2 - 1{,}158z + 0{,}4112} \tag{6.15}$$



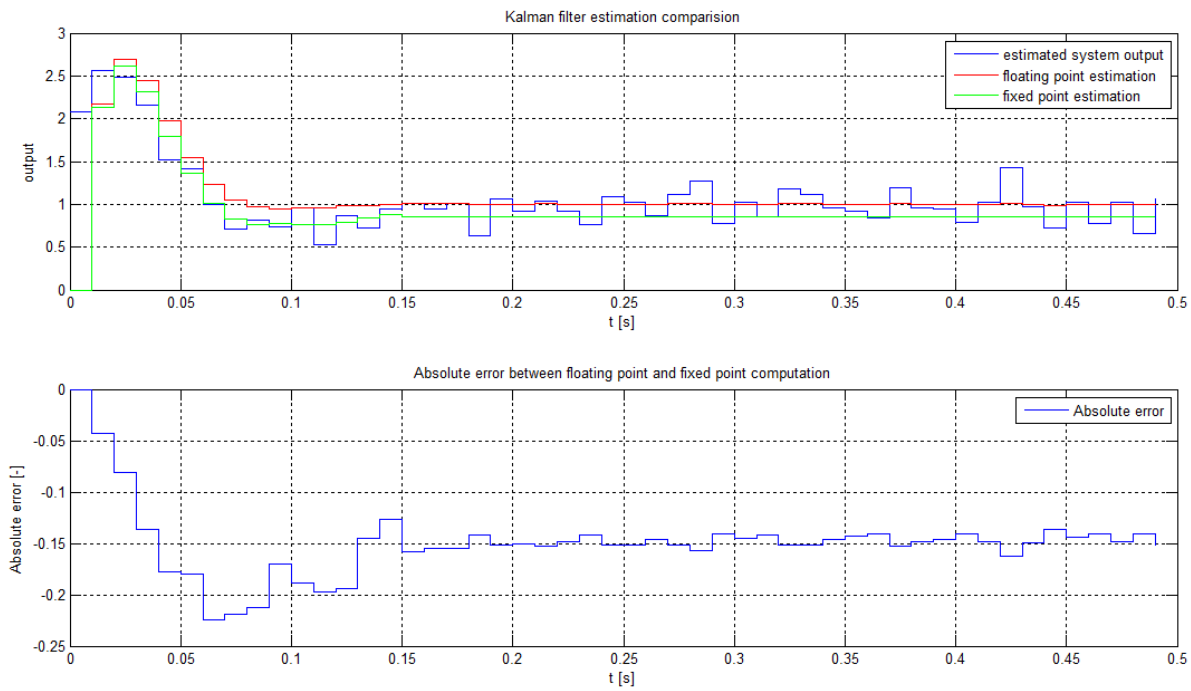
Obrázek 6.3: Filtrace zarušeného signálu Kalmanovým filtrem v *FP* a *FX* formátu s nastavením *A=16,B=16*



Obrázek 6.4: Filtrace zarušeného signálu Kalmanovým filtrem v *FP* a *FX* formátu s nastavením *A=12,B=20*



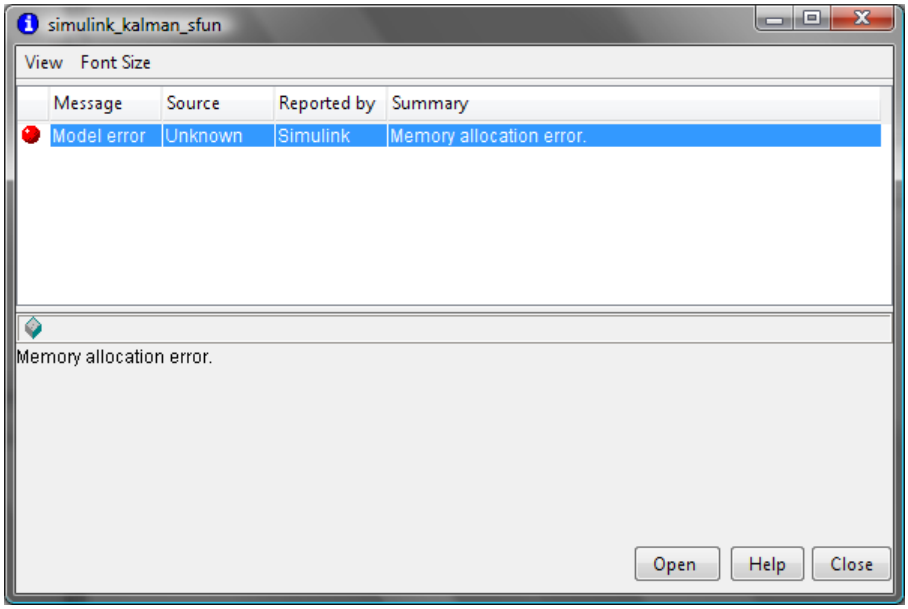
Obrázek 6.5: Filtrace zarušeného signálu Kalmanovým filtrem v *FP* a *FX* formátu s nastavením *A=20,B=12*



Obrázek 6.6: Filtrace zarušeného signálu Kalmanovým filtrem v *FP* a *FX* formátu s nastavením *A=22,B=10*

6.3.1 S-funkce Kalmanova filtru

Také pro knihovny *kalman* a *fixKalman* jsem se snažil vytvořit S-funkce (viz. kapitola 3) pro možnost jejich používání v modelačních úlohách Simulinku. S-funkce jsem implementoval obvyklým způsobem a i úspěšně zkompiloval *mex* kompilátorem. Problém se objevil až při používaní těchto S-funkcí v Simulinku, kdy po spuštění simulace Simulink oznámil chybu: „*Memory allocation error*“ podle (obr. 6.7), nebo v některých případech se celý Matlab dokonce zhroutil. Snažil jsem se zjistit bližší informace o této chybě v referenční příručce Matlabu i na jeho oficiálním internetovém diskusním fóru, ale bez valného úspěchu. Možný problém by podle mého názoru mohl být v používaní dynamického alokování paměti v knihovnách Kalmanova filtru. Bohužel se mi ale z časových důvodů nepodařilo S-funkce odladit do použitelného tvaru.



Obrázek 6.7: Ohlášená chyba po spuštění S-funkce Kalmanova filtru v Simulinku

Kapitola 7

Závěr

Cílem této práce bylo navrhnout algoritmy dvou číslicových regulátorů a Kalmanova filtru v programovacím jazyce C. Návrh jsem měl za úkol provést jak ve formátu plovoucí řádové čárky, tak ve formátu pevné řádové čárky. Správnost implementace jsem ověřil pomocí testovacích programů realizujících jednoduché úlohy řízení. Zároveň pro možnost používaní všech algoritmů v Matlabu, respektive v Simulinku, jsem implementoval tzv. C MEX systémové funkce.

Jelikož programovací jazyk C neposkytuje žádnou podporu formátu pevné řádové čárky, mým prvním a možná nejdůležitějším úkolem byla implementace knihovny *fixedpoint*, která tento formát a jeho aritmetiku reprezentuje. Vývoj knihovny je popsán v kapitole 2. Největším problémem bylo asi vůbec pochopení funkce aritmetiky formátu pevné řádové čárky, poněvadž to není zrovna obvyklý způsob reprezentace čísel. Při implementaci knihovny byla náročná především definice detekce přetečení, respektive podtečení reprezentovaných hodnot u jednotlivých aritmetických operací, zejména u operace dělení dvou čísel v *FX* formátu. Výsledek testování knihovny ukázal její vcelku rozumné chování pro aplikace obecných výpočetních úkonů.

Kapitola 3 popisuje princip funkce systémových funkcí Matlabu psaných v programovacím jazyce C, které jsem používal pro testování navržených algoritmů regulátorů a Kalmanova filtru přímo v prostředí Simulinku. Kapitola poskytuje také informace o možnostech implementace C MEX S-funkcí a jejich ladění. Kapitola 3 je pojata víceméně formou jakéhosi návodu, jak systémové funkce psát v jazyce C a používat v prostředí Simulinku jako uživatelsky příjemně ovladatelné funkční bloky. Jelikož lidé z Katedry řízení ČVUT, se kterými jsem měl tu čest spolupracovat, neměli s problematikou S-funkcí psaných v jazyce C žádné dosavadní zkušenosti, musel jsem tomuto z počátku

složitému tématu čelit bez jakýchkoliv jejich rad a pomoci s problémy, což mi zabralo výrazné množství času.

Ve čtvrté kapitole popisuji mou implementaci obecného regulátoru v obou číselných reprezentacích. Obecný regulátor představuje soustavu složenou ze tří číslicových filtrů typu *FIR* a *IIR*. Pro účely filtrace jsem musel tedy vytvořit speciální pomocné knihovny *filter* a *fixFilter*, které jsem pak použil v samotném algoritmu obecného regulátoru. Největším problémem, ke kterému jsem se musel několikrát vracet a opravovat, byla implementace algoritmu pro výpočet odezvy filtru, nebo potažmo obecného diskrétního systému popsaného přenosem, na vstupní signál. Nakonec se mi podařilo odladit tento algoritmus do konečné podoby, o čemž svědčí i obrázek s ukázkou jak odezvy jednotlivých filtrů, tak i funkce celého regulátoru. Funkční knihovnu jsem pak použil při implementaci S-funkce, která poskytuje možnost používání obecného regulátoru pro modelování v Simulinku.

Pátá kapitola pojednává o implementaci algoritmu modifikovaného *PID* regulátoru, jehož prvotním autorem je Ing. Ondřej Špinka, který mi svůj regulátor poskytl k úpravám a přepracování do formátu pevné řádové čárky. Při odlaďování funkce knihovny jsem se v tomto případě potýkal s častými problémy. Funkci regulátoru jsem měl možnost porovnat a vyzkoušet na reálných datech pořízených při aplikaci původní verze regulátoru na praktické úloze řízení modelu helikoptéry. Při několika počátečních testech jsem ale nedokázal dosáhnout kýženého výsledku. Až po konzultaci s panem Ing. Špinkou se mi podařilo konečně odladit všechny chyby a uvést tak mou verzi *PID* regulátoru do provozu. V této kapitole jsem chtěl také ukázat vliv nastavení formátu pevné řádové čárky na průběh regulace. Z provedených testů vyplynulo, že pro vhodně zvolené poměry celé a desetinné části *FX* formátu se průběh regulace výrazně neliší od verze počítané klasicky formátem plovoucí řádové čárky. Jelikož se ale výpočet každého kroku provede s určitou chybou, celková chyba se v průběhu času akumuluje a po delší regulaci se začne výsledek odchylovat od správného řešení. Stejně jako v předchozím případě obecného regulátoru jsem nezapomněl i na implementaci systémové funkce pro Simulink.

Kapitola 6 popisuje můj poslední navržený algoritmus, tentokrát ale týkající se Kalmanovy filtrace. Základní nástin řešení Kalmanova filtru mi dodal vedoucí práce Ing. Ondřej Holub formou Matlab skriptu. Před samotnou realizací bylo nejprve nutné implementovat pomocnou knihovnu pro práci s maticemi v jazyce C. I přesto, že si myslím, že standardní knihovny jazyka C podporují práci s maticemi, rozhodl jsem se implementovat svou vlastní z důvodu nutnosti přepisu této knihovny i do formátu pevné řádové čárky. Při řešení jsem se nepotýkal s žádnými výraznějšími problémy, algoritmus pracoval správně téměř na první pokus. Z testování se ukázalo dokonce i kvalitní chování verze pracující v *FX* formátu, kdy i při výrazných nepoměrech nastavení celé a desetinné části filtrace fungovala jen s nepatrnou chybou. Bohužel se mi ale nepodařilo z důvodu nedostatku zbývajícího času kompletně odladit S-funkce používající navrženou knihovnu Kalmanova filtru.

Při této práci jsem získal neocenitelné praktické zkušenosti z teorie řídící techniky i teorie číslicové filtrace. Obohatil jsem výrazně také své znalosti v oblasti programování v jazyce C. Nejvíce mě zaujala tématika Kalmanova filtru, se kterým jsem doposud neměl žádné výraznější praktické zkušenosti. Myslím, že je to jeden z nejzajímavějších číslicových systémů, se kterým jsem se setkal.

Literatura

[1] *YATES, Randy. Technical Reference Fixed-Point Arithmetic : An Introduction. [s.l.] : [s.n.], 2007. 13 s. Dostupný z WWW: <www.digitalsignallabs.com/fp.pdf >.*

[2] *TIŠNOVSKÝ, Pavel. Seriál Fixed Point Arithmetic. Root.cz [online]. 2006 [cit. 2008-07-22]. Dostupný z WWW: <http://www.root.cz/serialy/fixed-point-arithmetic/>.*

[3] *Fixed-point arithmetic. Wikipedia [online]. 2008 [cit. 2008-07-22]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Fixed-point_arithmetic>.*

[4] *Floating point. Wikipedia [online]. 2008 [cit. 2008-07-22]. Dostupný z WWW: <http://en.wikipedia.org/wiki/Floating_point>.*

[5] *Simulink 7 : Writing S-functions. Documentation for MathWorks Products [online]. 2008 [cit. 2008-07-27]. Dostupný z WWW: <http://www.mathworks.com/access/helpdesk/help/pdf_doc/simulink/sfunctions.pdf>.*

[6] *HLAVÁČ, Václav, SEDLÁČEK, Miloš. Zpracování signálů a obrazů. Praha : Nakladatelství ČVUT, 2007. 249 s. ISBN 978-80-01-03110-0.*

[7] *KERNIGHAN, Brian W., RITCHIE, Dennis M. Programovací jazyk C : ANSI C99. Miroslav Virius, Josef Novák; Martin Domes, Jiří Matoušek; Zbyněk Šťáva; Martin Hanslian, Petr Klíma, René Kašík, Martin Sodomka. Brno : Computer Press, a.s., 2006. 281 s. ISBN 80-251-0897-X.*

[8] *FRANKLIN, Gene F., POWELL, J. David, EMAMI-NAEINI, Abbas. Feedback control of dynamic systems. 5th edition. [s.l.] : Upper Saddle River : Prentice Hall, 2006. 910s. ISBN 0-13-149930-0.*

[9] *ÅSTRÖM, Karl J., WITTENMARK, Björn. Computer-Controlled Systems : Theory and Design. 3rd edition. New Jersey : Prentice-Hall, 1997. 557 s. Prentice Hall Information and System Sciences Ser.. ISBN 0-13-314899-8.*

[10] *Matlab : Control System Toolbox : Function kalman. Documentation for MathWorks Products [online]. 2008 [cit. 2008-08-12]. Dostupný z WWW: <http://www.mathworks.com/access/helpdesk/help/toolbox/control/>.*

[11] *VOJÁČEK, Antonín. Co je to Kalmanova filtrace. Automatizace.hw.cz : Měření a regulace [online]. 2007 [cit. 2008-08-12]. Dostupný z WWW: <http://automatizace.hw.cz/mereni-a-regulace/ART327-co-je-to-kalmanova-filtrace-.html>.*

[12] *Aplikace Kalmanova filtru pro zpracování signálu v měřicí technice. [online]. [cit. 2008-08-12]. Dostupný z WWW: <http://measure.feld.cvut.cz/groups/edu/vmd/Kalmankor.htm>*

[13] *WINKLER, Zbyněk. Měření rychlosti. Robotika.cz [online]. 2005 [cit. 2008-08-12]. Dostupný z WWW: <http://robotika.cz/guide/filtering/en>.*

Software

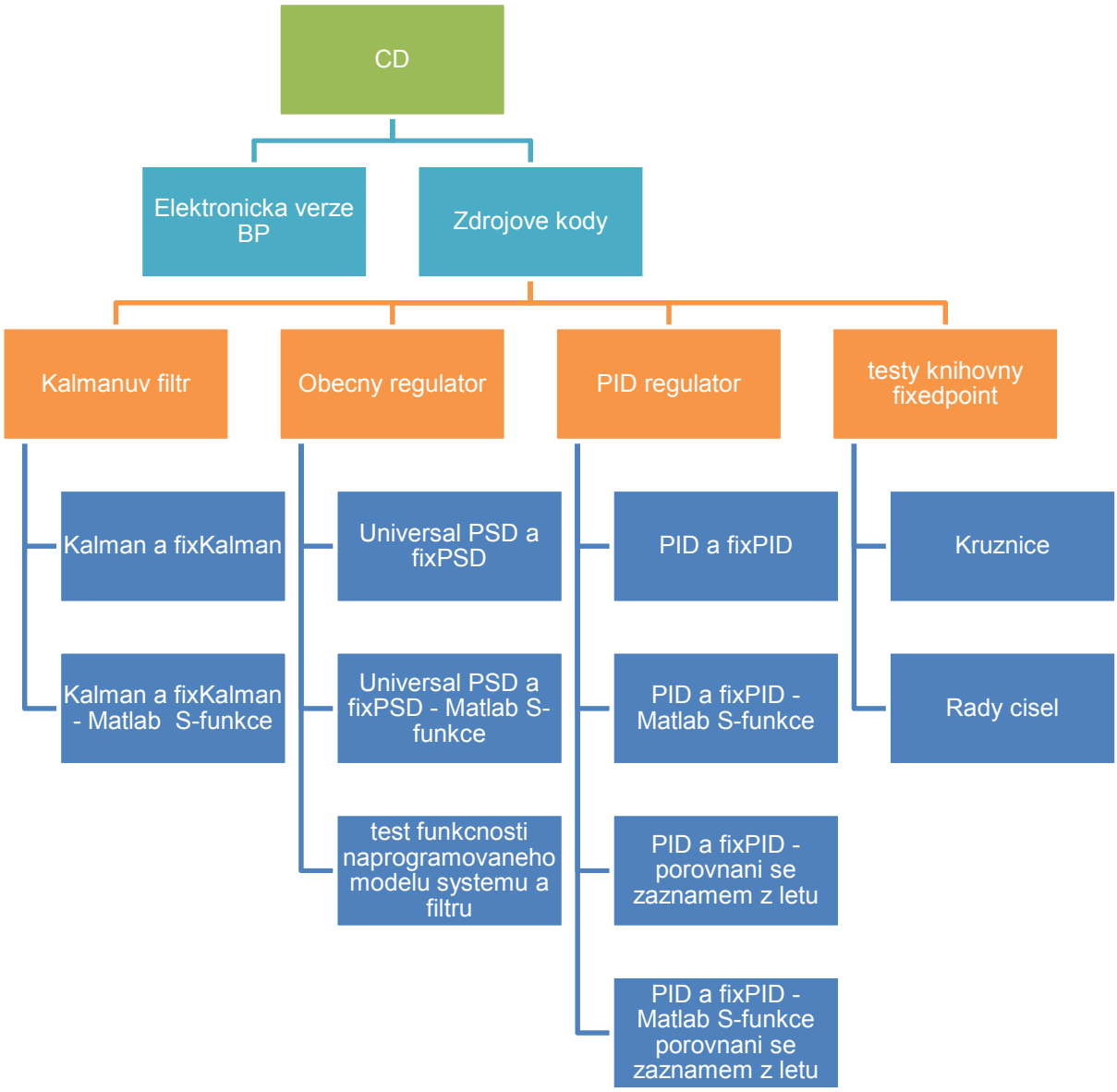
[I] **LabWindows/CVI Version 7.1.0 (306) Demonstration program**

[II] **MATLAB Version 7.5.0.342 (R2007b)**

MATLAB	Version 7.5 (R2007b)
Simulink	Version 7.0 (R2007b)
Control System Toolbox	Version 8.0.1 (R2007b)
Extended Symbolic Math Toolbox	Version 3.2.2 (R2007b)
Filter Design Toolbox	Version 4.2 (R2007b)
Fixed-Point Toolbox	Version 2.1 (R2007b)
Fuzzy Logic Toolbox	Version 2.2.6 (R2007b)
Gauges Blockset	Version 2.0.5 (R2007b)
Image Acquisition Toolbox	Version 3.0 (R2007b)
Instrument Control Toolbox	Version 2.5 (R2007b)
MATLAB Compiler	Version 4.7 (R2007b)
Model Predictive Control Toolbox	Version 2.3 (R2007b)
Neural Network Toolbox	Version 5.1 (R2007b)
Optimization Toolbox	Version 3.1.2 (R2007b)
Real-Time Workshop	Version 7.0 (R2007b)
Real-Time Workshop Embedded Coder	Version 5.0 (R2007b)
Robust Control Toolbox	Version 3.3 (R2007b)
Signal Processing Blockset	Version 6.6 (R2007b)
Signal Processing Toolbox	Version 6.8 (R2007b)
Simulink Control Design	Version 2.2 (R2007b)
Simulink Fixed Point	Version 5.5 (R2007b)
Simulink Response Optimization	Version 3.1.2 (R2007b)
Simulink Verification and Validation	Version 2.2 (R2007b)
Stateflow	Version 7.0 (R2007b)
Stateflow Coder	Version 7.0 (R2007b)
Statistics Toolbox	Version 6.1 (R2007b)
Symbolic Math Toolbox	Version 3.2.2 (R2007b)
System Identification Toolbox	Version 7.1 (R2007b)
Virtual Reality Toolbox	Version 4.6 (R2007b)

[III] **Microsoft Visual Studio 2005 Version 8.0.50727.867**
Microsoft .NET Framework Version 2.0.50727 SP1

Microsoft Visual Basic 2005
Microsoft Visual C# 2005
Microsoft Visual C++ 2005
Microsoft Visual J# 2005
Microsoft Visual Studio 2005 Tools for Applications
Microsoft Visual Web Developer 2005
Microsoft Web Application Projects 2005

Příloha A

Obsah přiloženého CD