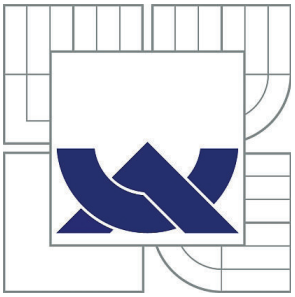


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ

ÚSTAV MECHANIKY TĚLES, MECHATRONIKY A BIOMECHANIKY

FACULTY OF MECHANICAL ENGINEERING

INSTITUTE OF SOLID MECHANICS, MECHATRONICS AND BIOMECHANICS

VÝVOJ ALGORITMŮ PRO ODHAD STAVU EXPERIMENTÁLNÍHO VOZIDLA

DEVELOPMENT OF ALGORITHMS STATE ESTIMATION OF EXPERIMENTAL VEHICLE

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

AUTOR PRÁCE — Bc. VOJTĚCH LAMBERSKÝ

AUTHOR

VEDOUCÍ PRÁCE — Ing. ROBERT GREPL, Ph.D.

SUPERVISOR

BRNO 2010

Vysoké učení technické v Brně, Fakulta strojního inženýrství

Ústav mechaniky těles, mechatroniky a biomechaniky
Akademický rok: 2009/2010

ZADÁNÍ DIPLOMOVÉ PRÁCE

student(ka): Bc. Vojtěch Lamberský

který/která studuje v **magisterském navazujícím studijním programu**

obor: **Mechatronika (3906T001)**

Ředitel ústavu Vám v souladu se zákonem č.111/1998 o vysokých školách a se Studijním a zkušebním řádem VUT v Brně určuje následující téma diplomové práce:

Vývoj algoritmů pro odhad stavu experimentálního vozidla

v anglickém jazyce:

Development of algorithms state estimation of experimental vehicle

Stručná charakteristika problematiky úkolu:

Práce se bude zabývat vývojem algoritmů pro odhad stavu a řízení trakce vozidla se 4 plně řiditelnými koly. Pro dodaný nelineární dynamický model vozidla budou vytvořeny stavové estimátory na bázi Kalmanova filtru. Algoritmy vyvinuté a testované v prostředí Matlab/Simulink budou implementovány v 16-ti bitovém mikrokontroleru dsPIC.

Cíle diplomové práce:

1) Rešerše v oblastech: odhad stavů nelineárních dynamických systémů s využitím znalosti modelu, nástrojů pro dsPIC, Fixed Point nástroje v Simulinku.
2) Simulační model a odhad stavu pro několik základních úloh dynamiky vozidla.
3) Simulace stavového odhadu pro dodaný nelineární dynamický model zahrnující čtyři hnaná a řízená kola.
4) Implementace stavového odhadu do reálné řidicí jednotky experimentálního vozidla.
5) Testování vlastností na reálném experimentálním vozidle.

Seznam odborné literatury:

- Valášek, M.: Mechatronika, Vydavatelství CVUT 1995
- Noskievic: Modelování a identifikace systému
- Vlk, F.: Dynamika motorových vozidel. Brno 2000
- Vlk, F.: Asistenční a informační systémy motorových vozidel. Automobilová elektronika 1. Brno 2006
- Vlk, F.: Systémy řízení podvozku a komfortní systémy. Automobilová elektronika 2. Brno 2006

Vedoucí diplomové práce: Ing. Robert Grepl, Ph.D.

Termín odevzdání diplomové práce je stanoven časovým plánem akademického roku 2009/2010.

V Brně, dne

L.S.

prof. Ing. Jindřich Petruška, CSc. Ředitel ústavu	prof. RNDr. Miroslav Doupovec, CSc. Děkan fakulty

Abstrakt

Tato práce se zabývá studiem filtračních algoritmů používajících matematický model systému k zlepšení kvality filtrace. Navržené filtrační algoritmy jsou použité v řídící jednotce experimentálního vozidla (filtrování signálu pro zpětnovazební regulátory). Na experimentálním vozidle je demonstrováno zlepšení odhady polohy při použití Kalmanova filtru. V další části je popsán způsob návrhu algoritmu pro mikrokontroléry dsPIC z prostředí Matlabu.

Abstract

This thesis deals with the filter algorithm design, implementing mathematical model to improve algorithm performance. Designed algorithms are implemented in a control unit of the experimental vehicle (filters signal used in the closed-loop controller). The improvement of the position estimation using Kalman Filter is demonstrated on the experimental vehicle. In the next part the design process of algorithm developing for dsPIC microcontroller using Matlab is described.

Klíčová slova

Unscentovaný Kalmanův filtr, Rozšířený Kalmanův filtr, Kalmanův filtr, dsPIC, MPC555, filtrování signálu, odhad parametru, identifikace systému

Keywords

Unscented Kalman Filter, Extended Kalman Filter, Kalman Filter, dsPIC, MPC555, signal filtering, parametr estimation, system identification

LAMBERSKÝ, V. Vývoj algoritmů pro odhad stavu experimentálního vozidla. Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, 2010. 78 s. Vedoucí diplomové práce Ing. Robert Grepl, Ph.D.

Čestné prohlášení

Čestně prohlašuji, že jsem tuto práci vypracoval samostatně s použitím uvedené literatury a pod vedením vedoucího DP.

Bc. Vojtěch Lamberský, Brno, 2010

Poděkování

Chtěl bych tímto poděkovat všem, kteří mi při vytváření práce pomáhali. Především děkuji svému vedoucímu diplomové práce, Ing. Robertu Greplovi, Ph.D. za trpělivost, cenné rady a věnovaný čas, dále spolupracovníkům z Laboratoře Mechatroniky za výdrž a inspiraci.

Obsah

Úvod do problematiky, kterou se tato práce zabývá.

1 Úvod

V okamžiku, kdy se pokusíme změřit nějakou veličinu, narazíme na problém. Ukazuje se že nikdy není možné určitou veličinu, například délku, změřit přesně. (I když vyloučíme hrubé a systematické chyby měření) S tímto problémem je nutné se nějakým způsobem vyrovnat. Zlepšení přesnosti měření je možné dosáhnout jednak použitím přesnějšího měřicího zařízení (větší rozlišení, menší úroveň šumu) nebo matematickým zpracováním. Jelikož měřící jednotka je obvykle vyrobena na hranici technických možností a zlepšení jejich vlastností není možné nebo pořízení lepší měřicí jednotky je příliš nákladné, věnujeme pozornost možnosti použití matematického zpracování signálu.

Pokud například měřím délku tyče a mám dost času, mohu provést několik měření a s použitím nástrojů statistického zpracování dat zjistit, v jakém intervalu a s jakou pravděpodobností leží skutečná hodnota. Bohužel, abychom měli dostatečně malý interval, ve kterém leží hledaná hodnota musíme provést velké množství měření a ještě nesmírně rychle, protože měřená veličina se mění v čase (tyč se rozpíná vlivem změny teploty). Tento způsob měření lze provést pouze pro vlastnosti, které se mění pomalu, například změna pokojové teploty.

Častěji však potřebujeme měřit veličiny, které se mění velmi rychle (například poloha auta). Je patrné, že v určitý časový okamžik stihneme provést pouze jediné měření. Abychom mohli nějakým způsobem zlepšit měření, („přiblížit se skutečné hodnotě“) zkombinujeme znalost předchozí naměřené hodnoty a znalost dynamiky systému. Představitelem tohoto postupu je například výpočet průměru se „zapomínáním“ starších hodnot.

Nicméně výsledky dosahované výše popsaným postupem nejsou příliš uspokojivé. Pro zlepšení odhadu skutečné hodnoty soustavy se dnes používá znalost systému (jeho matematického modelu), měřená data a informace o jejich kvalitě (obvykle reprezentována rozptylem).

Dalším problémem je, jak toto matematické zpracování provést. Pokud potřebujeme zlepšit přesnost dat (tuto operaci obvykle označujeme jako filtrování) zpracovávaných v počítači nebo jiném „výkonném“ zařízení (osciloskop), je situace velmi jednoduchá. Můžeme napsat prakticky jakýkoliv algoritmus, podle kterého se pro-

vede výpočet. Pokud ovšem potřebujeme filtrovat signál na jednoduchém zařízení (mobilní telefon) musíme počítat s omezeným výpočetním výkonem, který máme k dispozici a návrh takového algoritmu se značně komplikuje.

Jednotlivé kapitoly se zabývají problémy představenými výše. Hodně pozornosti je věnováno studiu metod používaných pro filtrování signálu, implementaci těchto algoritmů ve výpočetních jednotkách zařízení (dsPIC a MPC555) a studiu vlastností těchto platforem. Na závěr je pak algoritmus filtrování signálu otestován na experimentálním vozidle *Car4* postaveným v laboratoři mechatroniky. Díky tomuto vozidlu můžeme sledovat chování algoritmů na soustavě, která je svým charakterem velmi blízká praktickým aplikacím.

Jak je patrné z předchozího textu, velká část této práce je součástí projektu *Car4*. Cílem tohoto projektu je sestavit model vozidla se čtyřmi hnanými a zatáčejícími koly, na kterém bude možné vyvíjet algoritmy jízdní bezpečnosti. (Například zlepšení jízdní stability, zkrácení brzdné dráhy a podobné.) V době psaní této práce byl kompletně sestaven podvozek vozidla, zprovozněna elektronika a komunikace s počítačem, takže bylo možné začít testovat vlastnosti vozidla při použití s jednoduchými typy algoritmů.

Tento projekt vznikl díky spolupráci lidí z laboratoře mechatroniky, kteří pracovali na různých částech tohoto projektu.

> *Práce se zabývá návrhem a realizací doplňkového senzorického systému pro reálný projekt:„Experimentální čtyřkolové vozidlo“. Jedná se o senzory polohy tlumičů, teplotní senzory a kolizní senzory. Zároveň je vytvořen systém pro zabránění střetu vozidla s překážkou, který je implementován do modelu řízení.*
> Matěj Šimurda [1]

> *Práce popisuje návrh, vývoj a realizaci palubní elektroniky pro řízení pohonů, jejich výkonové buzení, elektroniku dálkového ovládání vozidla, implementaci základního firmware pro řídicí jednotky a testování řídicí jednotky metodikou HIL.*
> Josef Vejlupek [2]

> *Práce se zabývá vývojem dynamických modelů pro odhad stavu a řízení trakce vozidla se čtyřmi plně řiditelnými koly. Z mnoha variant popsaných v literatuře jsou vybrány modifikovány takové modely, které dostatečně věrohodně postihují vlastnosti reálného experimentálního vozidla a zároveň jsou výpočetně zvládnutelné v reálném čase.*
> Michal Jasanský [3]

Hlavní nálpní této práce je konstrukční návrh podvozku vozidla se čtyřmi řízenými a hnanými koly. Předpokládá se provoz ve vnitřním i vnějším prostředí.
Filip Vadlejch [4]

Zpracování signálu pomocí KF, EKF a UKF filtrů.

2 Metody používané k filtrování dat

2.1 Úvod k filtrování dat

V dnešní době je kolem nás velké množství zařízení, která nějakým způsobem zpracovávají signál (data). Od jednoduchých algoritmů implementovaných na levných signálových procesorech (mobilní telefony) po sofistikovaná řešení implementovaná na výkonných počítačích (automobily, letadla).

Nejjednoduššími typy filtrů jsou pásmové propustě nebo zádrže. Při jejich použití se vychází z úvahy, že výstup určité soustavy vzhledem k její dynamice a typu buzení může obsahovat pouze určité frekvence. Pokud se v měřených datech objeví i jiné frekvence (vyšší) je zřejmé, že se jedná o chybu v měření - šum. Můžeme tedy použít takový filtr, který bude propouštět pouze frekvence, které do dané soustavy patří. Pro filtrování hodnot měřených na mechanických soustavách se obvykle používá filtr dolní propust (případně pásmová propust). Tyto metody lze zařadit do širší kategorie filtrů *FIR* a *IIR* (konečná, respektive nekonečná odezva na skok). Do kategorie *FIR* patří například výpočet diference (2.1) (tento algoritmus je často používaný jako „náhrada“ za derivaci při použití numerických metod). Přičemž a_x v rovnici (2.1) je převrácená hodnota časového kroku Δt. Jak je zřejmé pro výpočet se použijí pouze aktuální a minulá změřená hodnota, proto filtr typu *FIR*.

$$y(t) = a_k x(t) - a_{k-1} x(t - \Delta t) \tag{2.1}$$

Jednoduché filtry prvního řádku lze navrhnout intuitivně, metodika návrhu složitějších typů filtrů je popsána například [5]).

Naši pozornost dále zaměříme zejména na filtrování signálu založené na znalosti systému (jeho matematického modelu). Tento přístup, oproti výše popsaným metodám, poskytuje pro většinu úloh podstatně lepší výsledky.

2.1.1 Šum v měření

Měřená veličina je obvykle závislá na mnoha parametrech. Při modelování systému pak parametry které známe (napětí, které jsme přivedli na motor) nebo můžeme mě-

řit označíme jako vstupní. Oproti tomu vstupní parametry jejichž velikost neznáme (nemůžeme nebo nechceme měřit) budeme považovat za rušení. Tyto parametry v modelu označujeme jako chyby - *disturbance*. Takovéto chyby obvykle způsobují náhodné pohyby atomů, měnící se elektromagnetická pole. Tento typ chyb je v technické literatuře obvykle označován jako šum.

Podle frekvencí, které rušení obsahuje, rozlišujeme šum bílý a barevný. Bílý šum je takový, který má energii signálu rozloženou přes všechny frekvence stejnoměrně. Barevný má většinu energie soustředěnou pouze v určitém frekvenčním pásu (pásech). V reálných aplikacích se vyskytuje šum barevný, ale není „příliš" barevný. Proto se i při řešení problémů s barevným šumem pracujeme jako by se jednalo o šum bílý. Navíc většina modelů používá tzv. *Gaussian white noise*, což je proměnná s vlastnostmi normálového rozdělení. Přitom se vychází z vlastnosti normálního rozdělení, které implikuje rovnoměrné rozdělení energie přes všechny frekvence (opak neplatí). Podobně i v dalším textu budeme pro modelování šumu náhodnou proměnnou používat Gaussovo rozdělení.

2.1.2 Použité značení

Nadále budeme pracovat téměř výhradně s diskrétními systémy. Abychom zjednodušili a zpřehlednili zápis, použijeme konvenci pro zápis časové proměnné (2.2). Přičemž ΔT je časový krok a k celé číslo.

$$x(k \cdot \Delta T) = x(k) = x_k \tag{2.2}$$

Abychom mohli využít nástroje statistické matematiky, budeme proces považovat za stochastický. Pokud měříme statickou veličinu, můžeme odhadnout s jakou pravděpodobností a jak moc se blíží skutečné hodnotě na základě rozptylu (nebo taky směrodatná odchylka) spočítá se podle vzorce (2.4) ve kterém střední hodnota se spočítá podle (2.3). V těchto vzorcích je n počet prvků. Pro naše účely nepotřebujeme rozlišovat mezi směrodatnou odchylkou a výběrovou směrodatnou odchylkou. Podrobněji je daná problematika rozebrána například v [6].

$$\mu = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{2.3}$$

$$\sigma^2 = \frac{1}{n} \sum_{i=1}^{n} (x_i - \mu)^2 \tag{2.4}$$

Pomocí výše popsaných nástrojů se pokusíme určit „kvalitu" signálu. Představme si že měříme určitou veličinu a mohli bychom provádět měření nekonečně rychle (všechny v jednom okamžiku). Pokud bychom pro určitý okamžik dokázali naměřit nekonečně mnoho hodnot, střední hodnota by reprezentovala skutečnou hodnotu dané veličiny. A rozptyl (2.4) „rozložení" - druhý kvadratický moment

změřených dat kolem skutečné hodnoty. Navíc mějme dvě čidla pro měření stejné veličiny. Jelikož jsou měřicí soustavy vždy vyrobeny s určitou nepřesností, dávají i různě zkreslené výsledky. Uvažujme že obě čidla změří stejnou střední hodnotu, ale jejich výstupy mají různý rozptyl. Nyní můžeme na základě porovnání rozptylů hodnot, rozhodnout o jejich kvalitě.

V dalších úvahách bude vždy používat rozptyly k porovnání dvou signálů. Obecně lze říci, že mohu více „věřit“ hodnotě, která má menší rozptyl a obráceně. Jinými slovy, hodnota naměřená čidlem s menším rozptylem je pravděpodobně blíž skutečné hodnotě a mohu se na ni více spolehnout.

2.2 Kalmanův filtr

2.2.1 Princip funkce

Jak bylo zmíněno výše, Kalmanův filtr je založený na filtrování dat pomocí znalosti systému, ze kterého jsou data měřena. Prakticky výhradně se k popisu systému používá stavový model v diskrétní podobě. Postup vytváření matematických modelů a stavového popisu lze najít v [7]. Samotný princip je velmi jednoduchý, matematický model se počítá v reálném čase a pokud by byl tento model dokonalý vlastně bychom ani nepotřebovali provádět žádná měření na skutečné soustavě, protože by dával stejné výsledky jako soustava, kterou modeluje. Nicméně takový model nelze vytvořit, a proto se provádějí při výpočtu v modelu korekce, tak aby jeho stavy odpovídaly stavům skutečné soustavy. V zásadě se jedná o strukturu pozorovatele s proměnným zesílením – *Kalmanovým* zesílením, jak je patrné z obrázku 2.1.

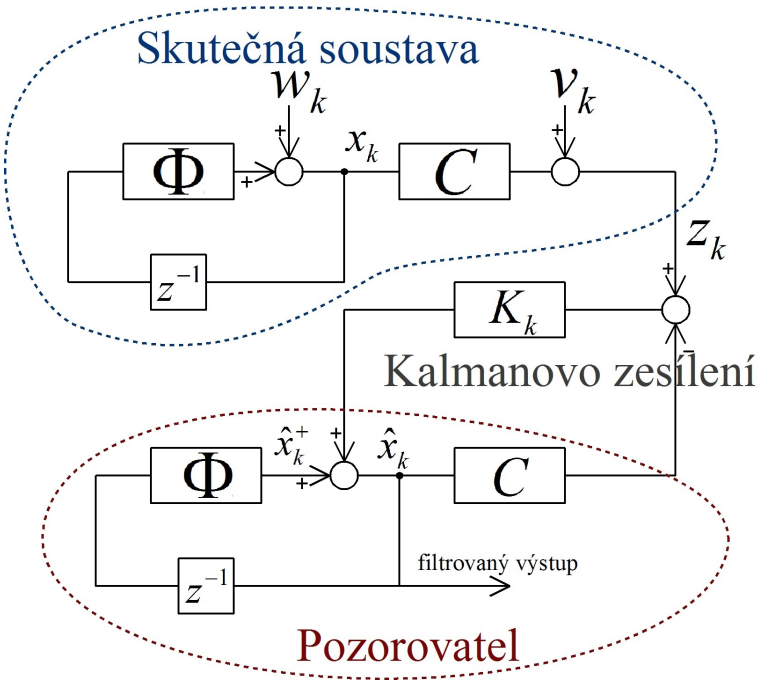
2.2.2 Matematický model

Vyjděme z modelu reálné soustavy, do jejíž stavů a měření vstupuje šum. Tento model (část obrázku 2.1 označená jako Skutečná soustava) popisuje rovnice (2.5).

$$\begin{aligned} x_{k+1} &= \Phi x_k + w_k \\ y_k &= C x_k + v_k \end{aligned} \tag{2.5}$$

V této rovnici představují proměnné w_k a v_k šum vstupující do systému. Tyto proměnné budou reprezentovat realizace z Gausova rozdělení, které ve většině případů velmi přesně modeluje skutečný charakter disturbancí. U většiny soustav není šum vstupující do stavů a měření závislý (korelovaný) kovarianční matice bude tedy mít na vedlejší diagonále nulové prvky (2.6).

$$\begin{bmatrix} w_k \\ v_k \end{bmatrix} \sim N \left[\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} Q & 0 \\ 0 & R \end{bmatrix} \right] \tag{2.6}$$



Obr. 2.1: Reálná soustava a pozorovatel

2.2.3 Kalmanovo zesílení

Jediným problémem tedy je, jak zvolit Kalmanovo zesílení. Pokud šum vstupuje do stavů i do měření, je vhodné zvolit zesílení pozorovatele tak, aby se minimalizovala kovarianční matice stavů P_k (velikosti variance a kovariance stavů). Tedy takový stav, při kterém známe velikost stavových veličin co nejpřesněji.

Filtrované $\hat{x}_k$ získáme podle rovnice (2.7). Což je v podstatě výstup z pozorovatele korigovaný Kalmanovým zesílením, přičemž z_k je vstup z reálné soustavy (měřený). Tento vstup modeluje rovnice (2.8).

$$\hat{x}_k = \hat{x}_k^+ + K_k \left(z_k - C\hat{x}_k^+\right) \tag{2.7}$$

$$z_k = Cx_k + v_k \tag{2.8}$$

Pokud dosadíme rovnici (2.7) do (2.8) získáme (2.9).

$$\hat{x}_k = \hat{x}_k^+ + K_k \left(Cx_k + v_k - C\hat{x}_k^+\right) \tag{2.9}$$

V rovnici (2.9) jsme spočítali novou střední hodnotu stavů (výstup pozorovatele korigovaný na základě měření stavů skutečné soustavy). Můžeme tedy spočítat varianci stavů P_k. Určí se podle rovnice (2.10). Pokud dosadíme rovnici (2.8) do (2.10) získáme (2.11)

$$P_k = E\left[e_k e_k^T\right] = E\left[\left(x_k - \hat{x}_k\right)\left(x_k - \hat{x}_k\right)^T\right] \tag{2.10}$$

$$\begin{aligned} P_k &= E\left[\left(x_k - \left(\hat{x}_k^+ + K_k\left(Cx_k + v_k - C\hat{x}_k^+\right)\right)\right)(...)^T\right] \\ P_k &= E\left[\left(\hat{x}_k^+ - K_kC\hat{x}_k^+ + x_k - K_kCx_k - K_kv_k\right)(...)^T\right] \\ P_k &= E\left[\left((I - K_kC)\left(x_k - \hat{x}_k^+\right) - K_kv_k\right)(...)^T\right] \end{aligned} \tag{2.11}$$

Protože stavové veličiny jsou nezávislé na šumu (jejich kovariance je nulová) důkaz je proveden například v [8]. Získáme po roznásobení výrazu (2.11) výraz (2.12). Šum vstupující do měření jsme označili maticí R_k. (Ve vzoreci (2.6) předpokládáme stejnou velikost kovariance šumu během celého měření, nicméně obecně může být i časově závislá.) Navíc výraz $E\left(\hat{x}_k - \hat{x}_k^+\right)$ je vzorec pro výpočet matice P_k^+. Tato matice je „odhadem“ matice P_k (variance stavů). Substitujeme-li dříve uvedené matice do výrazu (2.12), získáme výraz (2.13) a ten dále upravíme roznásobením na (2.14)

$$P_k = (I - K_kC)\,E\left(\hat{x}_k - \hat{x}_k^+\right)(I - K_kC) + K_kE\left[v_kv_k^T\right]K_k \tag{2.12}$$

$$P_k = (I - K_kC)\,P_k^+\,(I - K_kC) + K_kR_kK_k^T \tag{2.13}$$

$$P_k = P_k^+ - K_kCP_k^+ - P_k^+C^TK_k^T + K_k\left(CP_k^+C^T + R\right)K_k^T \tag{2.14}$$

Nyní máme rovnici pro výpočet rozptylu stavových proměnných. Na hlavní diagonále matice P_k jsou kovariance jednotlivých stavů. Jak už bylo zmíněno dříve, naším cílem je minimalizovat jejich velikost. Abychom toto mohli provést, potřebujeme určit normu matice, což je způsob jak určit její „velikost“. Použijeme funkci *trace*, která sečte prvky na hlavní diagonále. (podrobněji je funkce popsána například v [9]. Pochopitelně pokud transponujeme argument funkce *trace*, její hodnota se nezmění. Rovnici (2.14) můžeme upravit do tvaru (2.15). Jelikož hledáme minimum, celou rovnici zderivujeme podle K_k (2.16) a položíme rovnu nule (2.17). Dále z této rovnice vyjádříme K_k (2.18).

$$T\left[P_k\right] = T\left[P_k^+\right] - 2T\left[K_kCP_k^+\right] + T\left[K_k\left(CP_k^+C^T + R\right)K_k^T\right] \tag{2.15}$$

$$\frac{dT[P_k]}{dK_k} = -2\left(CP_k^+\right)^T + 2K_k\left(CP_k^+C^T + R\right) \tag{2.16}$$

$$\left(CP_k^+\right)^T = K_k\left(CP_k^+C^T + R\right) \tag{2.17}$$

$$K_k = \left(CP_k^+\right)^T\left(CP_k^+C^T + R\right)^{-1} \tag{2.18}$$

Tento výsledný vztah můžeme dosadit do rovnice (2.7) a (2.14), čímž získáme vztahy pro výpočet filtrovaných hodnot $\hat{x}_k$ a P_k, rovnice (2.19) respektive (2.20).

$$\hat{x}_k = \hat{x}_k^+ + \left(CP_k^+\right)^T\left(CP_k^+C^T + R\right)^{-1}\left(y_k - C\hat{x}_k^+\right) \tag{2.19}$$

$$P_k = \left(CP_k^+\right)^T\left(CP_k^+C^T + R\right)^{-1}CP_k^+ \tag{2.20}$$

Abychom mohli předchozí dvě rovnice spočítat, potřebujeme znát hodnoty $\hat{x}_k^+$ a P_k^+ (předpovědi hodnot). První z nich, $\hat{x}_k^+$ určíme snadno z matematického modelu (2.5)

nebo z pohledu na obrázek 2.1. Odhad nové hodnoty $\hat{x}_k$, tedy $\hat{x}^+_{k+1}$ bude (2.21).

$$\hat{x}^+_{k+1} = \Phi\hat{x}_k \tag{2.21}$$

Nyní spočítejme velikost P^+_k. Tato matice je dána (2.22).

$$\hat{x}^+_{k+1} = \Phi\hat{x}_k \tag{2.22}$$

A tento výraz upravíme (2.23).

$$\begin{aligned} P^+_{k+1} &= E\left[\left(x_{k+1} - \hat{x}^+_{k+1}\right)\left(x_{k+1} - \hat{x}^+_{k+1}\right)^T\right] \\ P^+_{k+1} &= E\left[\left((\Phi x_k + w_k) - \Phi\hat{x}_k\right)\left((\Phi x_k + w_k) - \Phi\hat{x}_k\right)^T\right] \\ P^+_{k+1} &= E\left[\left(\Phi e_k + w_k\right)\left(\Phi e_k + w_k\right)^T\right] \\ P^+_{k+1} &= E\left[\left(\Phi e_k\right)\left(\Phi e_k\right)^T\right] + E\left[w_k w_k^T\right] \end{aligned} \tag{2.23}$$

Substituujeme kovarianční matici zavedenou ve výrazu (2.6). A získáme výsledný výraz pro výpočet predikce matice P^+_k (2.24).

$$P^+_{k+1} = \Phi P_k \Phi^T + Q \tag{2.24}$$

Nyní máme všechny potřebné vztahy připravné. Opakovanou iterací spustíme výpočet podle rovnic (2.21), (2.24) a (2.19), (2.20).

2.3 Rozšířený Kalmanův filtr

Kalmanův filtr, tak jak byl navržen ve své původní podobě [10], lze použít pouze pro lineární systémy. Nelineární systémy (2.25) totiž nelze popsat přenosovou matici Φ. Je to dáno tím, že změna stavu nezávisí pouze na lineární kombinaci ostatních stavů. Pokud nemůžeme dané matice zkonstruovat, nelze použít Kalmanův filtr. Bohužel většina systémů, se kterými potřebujeme pracovat, je výrazně nelineární. Není tedy možné je jednoduše v celém pracovním rozsahu aproximovat lineárním modelem. Jednoduchý způsob jak toto obejít, je linearizace modelu v okolí aktuálního stavu v každém kroku výpočtu.

$$\begin{aligned} x_{k+1} &= f\left(x_k, u_k\right) + w_k \\ z_k &= h\left(x_k\right) + v_k \end{aligned} \tag{2.25}$$

Potřebujeme tedy v každém kroku výpočtu určit jakobián stavového modelu (Což může být problém, jelikož derivace některých funkcí je značně složitá). Jelikož analytické vyjádření je většinou příliš složité, používají se numerické metody (diference

v „nekonečně“ malém okolí stavu). Abychom urychlili výpočet, obvykle se počítá diference v komplexním oboru. Víme, že pro konformní funkce platí, když se přibližujeme z libovolného směru v komplexním prostoru, je derivace ve všech směrech stejná. Pokud tedy vezmeme pouze imaginární část, nemusíme od ní odečítat (hodnota stavu, kolem kterého počítáme derivaci měla nulovou komplexní část).

Část kódu z Matlabu uvedená níže tento postup demonstruje. $x1$ je stavová proměnná (vektor) a k němu přičteme eps (proměnná ve které je uložené „strojové delta“ - rozdíl dvou po sobě jdoucích čísel typu double). V dalším kroku vydělíme komplexní část nově získaného čísla velikostí delty a tím určíme gradient v daném bodě.

```
x1(k)=x1(k)+eps*i;
A(:,k)=imag(fun(x1))/eps;
```

Výpočet predikovaných stavů můžeme provést pomocí rovnice (2.26) a jejich korekci (2.27). Matice C obvykle bývá lineární, jinak ji získáme podobně jako matici Φ.

$$\hat{x}_{k+1}^{+} = f\left(\hat{x}_k, u_k\right) \tag{2.26}$$

$$\hat{x}_k = \hat{x}_k^{+} + P_k C^T \left(CPC^T + R\right)^{-1} \left(y_k - h\left(\hat{x}_k^{+}\right)\right) \tag{2.27}$$

Oproti tomu při výpočtu predikce kovariance stavů (2.28) a její aktualizace (2.29) nelze použít žádnou část nelineárního modelu a všechny matice (Φ a C) je nutné získat linearizací.

$$P_{k+1}^{+} = \Phi P_k \Phi^T + Q \tag{2.28}$$

$$P_k = P_k^{+} C^T \left(C P_k C^T + R\right)^{-1} C P_k^{+} \tag{2.29}$$

Tento přístup funguje velmi dobře na výkonných výpočetních jednotkách. Signál potřebujeme filtrovat prakticky všude a je tedy snaha používat co nejlevnější (méně výkonné) procesory. Bohužel, většina počítačů neumí pracovat s aritmetikou komplexních čísel, a tak je nutné výpočet emulovat a složitě optimalizovat. Navíc v případě použití „jednočipů“ je situace vzhledem k jejich výpočetnímu výkonu a instrukčním sadám daleko složitější.

2.4 Unscentovaný Kalmanův filtr

Jak je patrné, Rozšířený Kalmanův filtr je výpočetně poměrně náročný, proto se začalo vyvíjet několik alternativních výpočetních algoritmů, které by tento nedostatek alespoň částečně odstranily. Principy algoritmu Unscentovaného Kalmanova filtru, zkráceně *UKF* byly poprvé použity roku 1994 v Robotické výzkumné skupině (RRG)

na Oxfordu (Anglie). Po té se tento algoritmus objevil v několika článcích pod různými jmény jako například Sigma bodový filtr nebo FAB filtr. *UKF* je v dnešní době velmi „oblíbeným“ algoritmem pro filtrování dat nelineárních systémů. Tento filtr byl úspěšně aplikován v několika různých aplikacích jak je dokumentováno v [11], [12] nebo [13]. Jak je uvedeno v přednášce *Pekka Janise*, o konečném pojmenování algoritmu rozhodli členové *Robotic Research Group* a od té doby se používá výhradně označení Unscentovaný Kalmanův filtr.

2.4.1 Pravděpodobnostní model

Hlavní rozdíl oproti předchozím typům Kalmanova filtru je v tom, že *UKF* neaproximuje stavový model, ale rozdělení. Na stavové veličiny se můžeme dívat jako na veličiny pravděpodobnostní funkce (reprezentované střední hodnotou a rozptylem). Na základě stavového modelu (2.5) můžeme tyto funkce přímo určit. Kovarianci (X, Y) určíme podle rovnice (2.30) a kovarianci (Y, Y) podle rovnice (2.31). Kovariance (Y, X) je stejná jako (X, Y) s tím rozdílem, že tato matice je transponovaná.

$$\operatorname{cov}(X,Y) = E\left(XY^T\right) = E\left(X[CX]^T\right) = E\left(XX^TC^T\right) = PC^T \tag{2.30}$$

$$\begin{aligned} \operatorname{var}(Y) &= E\left(YY^T\right) = E\left(CX[CX]^T\right) = \\ &= E\left(CXX^TC^T\right) = CE\left(XX^T\right)C = CPC^T \end{aligned} \tag{2.31}$$

Máme tedy pravděpodobnostní rozdělení popisující rozložení stavových proměnných (2.32).

$$\begin{aligned} \begin{bmatrix} x_k \\ y_k \end{bmatrix} &\sim N\left(\begin{bmatrix} \hat{x}_k \\ \hat{y}_k \end{bmatrix}, \begin{bmatrix} P_k^{XX} & P_k^{XY} \\ P_k^{YX} & P_k^{YY} \end{bmatrix}\right) \\ \begin{bmatrix} x_k \\ y_k \end{bmatrix} &\sim N\left(\begin{bmatrix} \hat{x}_k \\ C\hat{x}_k \end{bmatrix}, \begin{bmatrix} P_k & P_kC^T \\ CP_k & CP_kC^T + R \end{bmatrix}\right) \end{aligned} \tag{2.32}$$

Výpočet nové střední hodnoty provedeme podle rovnice (2.33). Tuto rovnici bychom snadno odvodili dosazením (2.32) do (2.5).

$$\begin{aligned} \hat{x}_k &= \hat{x}_k^+ + P_kC^T\left(CPC^T + R\right)^{-1}\left(y_k - C\hat{x}_k^+\right) \\ \hat{x}_k &= \hat{x}_k^+ + P_k^{XY}\left(P_k^{YY}\right)^{-1}\left(y_k - \hat{y}_k^+\right) \end{aligned} \tag{2.33}$$

Kromě toho lze vycházet z „intuitivní“ úvahy a dospět ke stejné rovnici. V zásadě lze tuto rovnici chápat tak, že se snažíme upřesnit predikci budoucího stavu $\hat{x}_k^+$ na základě rozdílu mezi předpovědí budoucího výstupu $C\hat{x}_k^+$ a naměřenou hodnotu y_k. Nejdůležitější je rozhodnutí, jak na základě znalosti rozdílu naměřené a předpovězené hodnoty zpřesnit (opravit) střední hodnotu stavů. Je zřejmé, že platí:

- Čím větší je velikost variance y (P_k^{YY}), tím menší váhu lze přisuzovat měřenému stavu, tedy rozdílu předpovědi a změřené hodnoty. Pokud má nějaká veličina velkou varianci, znamená to, že hodnoty leží ve velkém intervalu, pokud naměříme hodnotu vzdálenou průměrné hodnotě a variance je velká, není to příliš závažné a nemusíme moc upravovat střední hodnotu. Pokud je naproti tomu variance malá a rozdíl výsledku oproti předpokladu velký musíme střední hodnotu opravit významně.
- Další parametr je kovariance vstupů a výstupů $P_k C^T$ (P_k^{XY}). Tato hodnota vyjadřuje, jak moc jsou výstupy závislé na stavech. Pokud například je výstup ze soustavy „nezávisí“ na stavech, rozdíl mezi předpovězeným výstupem a neposkytne „příliš“ dobrou informaci o skutečné střední hodnotě stavů. Na základě této úvahy je zřejmé, že čím více jsou stavy a výstupy systému závislé, tím větší váhu přisoudíme naměřené hodnotě (větší změna stavů na základě rozdílu naměřené hodnoty a předpovězené).

Z předchozích úvah vyplývá, že korekce střední hodnoty bude přímo úměrná kovarianci vstupů a výstupů $P_k C^T$ a nepřímo úměrná varianci výstupů (P_k^{YY}), což je ostatně vidět v rovnici (2.33).

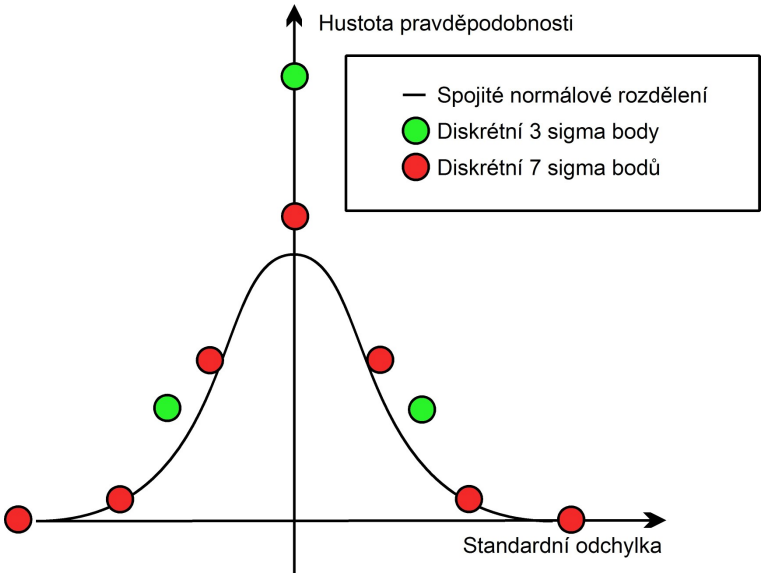
2.4.2 Aproximace rozdělení sigma body

V dalším kroku tedy potřebujeme určit kovarianci určité proměnné. Velmi dobře umíme pracovat s normálním rozdělením, které je spojité. Nicméně mimo to existuje i diskrétní normální rozdělení definované pro určitý počet bodů. Takové rozdělení je znázorněné na 2.2. Pro normálové rozdělení se používá lichý počet sigma bodů (středově souměrný). Je zřejmé, že potřebujeme alespoň tři sigma body, abychom mohli spočítat první kvadratický moment (rozptyl). Nyní tedy zvolíme sigma body tak, aby aproximovali dané rozdělení (jeho střední hodnotu a rozptyl), musí tedy platit (2.34).

$$\begin{aligned} \bar{x} &= \sum w_i x^{(i)} \\ P_x &= \sum w_i \left(x^{(i)} - \bar{x}\right) \left(x^{(i)} - \bar{x}\right)^T \end{aligned} \tag{2.34}$$

Pro vektor proměnných se tato úvaha analogicky rozšíří. Budeme potřebovat 2n+1 bodů (n je počet stavů, proměnných). Výraz $[M]_i$ v rovnici (2.35). představuje i-tý sloupec matice M. Navíc pokud platí $A = \sqrt{P}$, pak $P = AA^T$.

$$\begin{aligned} x^{(0)} &= \bar{x} \\ x^{(i)} &= \bar{x} + \left[\sqrt{(n+\lambda)\, P_x}\right]_i | i = 1..\ n \\ x^{(i)} &= \bar{x} - \left[\sqrt{(n+\lambda)\, P_x}\right]_i | i = n+1..\ 2n \end{aligned} \tag{2.35}$$



Obr. 2.2: Normálové rozdělení a sigma body.

Váhy pro jednotlivé body pro výpočet střední hodnoty a kovariance se spočítají podle vztahu (2.36).

$$\begin{aligned} W_m^{(0)} &= \frac{\lambda}{(n+\lambda)} \\ W_m^{(i)} &= \frac{\lambda}{2\,(n+\lambda)} | i = 1..\ 2n \\ W_c^{(0)} &= \frac{\lambda}{2\,(n+\lambda)} + 1 - \alpha^2 + \beta \\ W_c^{(i)} &= \frac{\lambda}{2\,(n+\lambda)} | i = 1..\ 2n \end{aligned} \tag{2.36}$$

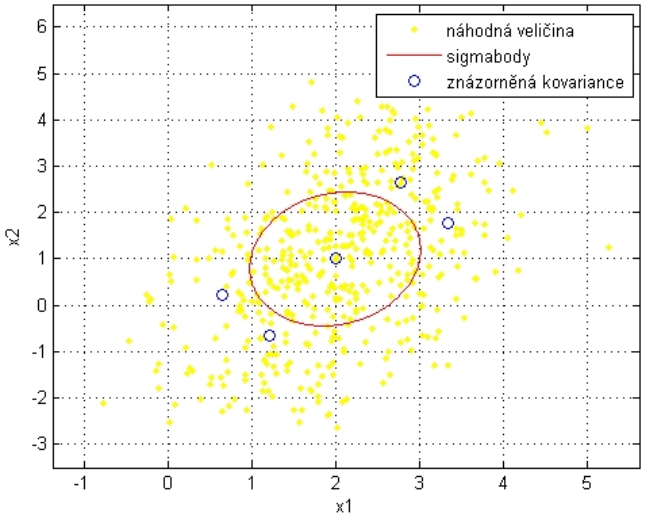
V předchozích rovnicích je parametr λ, *škálovací* parametr, který je definovaný (2.37).

$$\lambda = \alpha^2\,(n+k) - n \tag{2.37}$$

Konstanty α, β a k definují rozdělení. Jsou voleny jako nezáporné. Pro normální rozdělení se obvykle použije $\beta = 2$. Parametr α, ovlivňuje rozložení sigma bodů kolem střední hodnoty. Obvykle se volí malá hodnota (0,01 až 1). Poslední parametr je k. Většinou se volí 0 nebo $3 - n$.

Rozložení takto získaných sigma bodů je patrné z obrázku 2.3. Žluté body jsou body normálového rozdělení a modré jsou sigma body. Tyto body jsou zvoleny tak, aby pomocí nich šla snadno spočítat kovariance. Pokud bychom počítali kovarianci

žlutých bodů a modrých bodů, získali bychom dva shodné výsledky. Nicméně výpočet kovariancí z modrých (sigma) bodů je podstatně méně náročný. Tyto body byly vygenerovány tak, aby odpovídali pravoúhlým souřadnicím objektu v ploše.



Obr. 2.3: Naměřené hodnoty a sigma body před transformací.

2.4.3 Transformace bodů

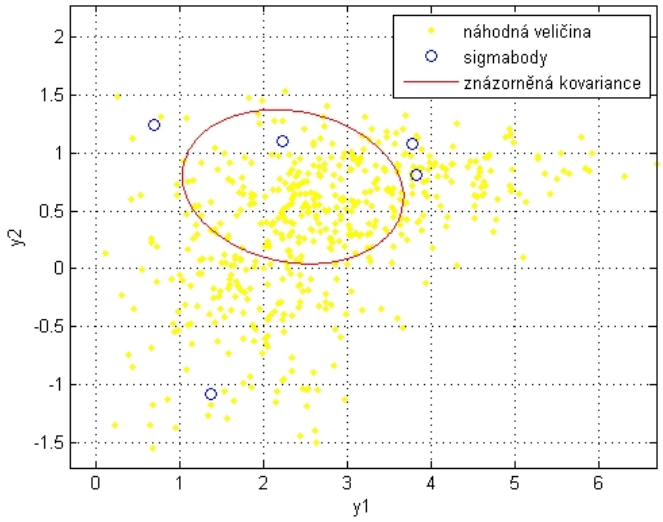
Nyní tedy máme potřebné body reprezentující pravděpodobnostní rozdělení stavů. Zobrazme je tedy nelineární stavovou funkcí (2.38).

$$y^{(i)} = g\left(x^{(i)}\right) | x = 0..\ 2n \tag{2.38}$$

Nyní máme všechny potřebné body a můžeme spočítat střední hodnotu a kovariance (2.39), které potřebujeme pro výpočet do rovnice (2.33).

$$\begin{aligned} \mu_y &= \sum_{i=0}^{2n} w_m^{(i)} y^{(i)} \\ P_{yy} &= \sum_{i=0}^{2n} w_c^{(i)} \left(y^{(i)} - \mu_y\right) \left(y^{(i)} - \mu_y\right)^T \\ P_{xy} &= \sum_{i=0}^{2n} w_c^{(i)} \left(x^{(i)} - \bar{x}\right) \left(y^{(i)} - \mu_y\right)^T \end{aligned} \tag{2.39}$$

Takto transformované body jsou zobrazené na obrázku 2.4. Pro zobrazení naměřených dat byla použita nelineární funkce převádějící pravoúhlé souřadnice do polárních. Tímto simulujeme měření dat radarem (známe vzdálenost a úhel, ze kterého se vrátil odražený paprsek).



Obr. 2.4: Naměřené hodnoty a sigma body po transformaci.

2.4.4 Algoritmus výpočtu

Unscentovaný Kalmanův filtr funguje tak, že v prvním kroku předpoví novou střední hodnotu stavů a jejich rozptyl (2.40) pomocí transformace sigma bodů.

Nejdříve se vytvoří sigma body (první rovnice). Ty se dále transformují nelineární funkcí popisující daný systém (druhá rovnice v (2.40) a na závěr se z takto transformovaných sigma bodů spočítá predikovaná střední hodnota stavů a kovariance.

$$\begin{aligned} X_k &= [m_k \ \dots \ m_k] + \sqrt{c}\left[0 \quad \sqrt{P_{k-1}} \quad -\sqrt{P_{k-1}}\right] \\ X^+_{k+1} &= f\left(X_k\right) \\ \mu^+_{k+1} &= X_{k+1} \cdot W_m \\ {P_{YY}}^+_{k+1} &= X^+_{k+1} \cdot W_m \cdot {X^+_{k+1}}^T + Q_k \\ {P_{XY}}^+_{k+1} &= X^+_{k+1} \cdot W_m \cdot {Y^+_{k+1}}^T \end{aligned} \tag{2.40}$$

Ve druhém kroku se aktualizuje střední hodnota a velikost rozptylu stavů x na základě naměřených výstupů soustavy (2.41). V první rovnici se koriguje střední

hodnota stavů a ve druhé se aktualizuje kovariance stavů.

$$\begin{aligned} m_k &= m_k^+ + P_{XY\,k} P_{YY\,k}^{-1} \left[y_k - \mu_k \right] \\ P_k &= P_k^+ - \left[P_{XY\,k} P_{YY\,k}^{-1} \right] P_{YY_k} \left[P_{XY\,k} P_{YY\,k}^{-1} \right]^T \end{aligned} \tag{2.41}$$

Samotný výpočet filtrování se provádí iterováním rovnic (2.40) a (2.41).

Srovnání výkonu nejpoužívanějších typů filtrů na silně nelineární úloze.

3 Testování filtrů na jednoduché úloze

Výkon algoritmů otestujeme na jednoduché modelované úloze *odhadování koeficientu tření* mezi pneumatikou auta a vozovkou. Modelovaná úloha znamená, že data použitá pro simulaci nebyla naměřena na reálné soustavě ale jsou vygenerována pomocí matematického modelu soustavy. Šum v senzorech simulují proměnné s normálovým rozdělením.

Nejedná se o základní filtrování signálu tak jak jej běžně chápeme (zpřesnění měřené veličiny) ale o aplikaci při které známe model a snažíme se odhadnout určitý stav (parametr) v závislosti na jiných (měřených) stavech.

Mějme jednoduchý model auta (čtvrtinový), kterým budeme přejíždět na různých typech povrchu a naším cílem je odhadnout koeficient tření povrchu.

3.1 Rovnice a data popisující úlohu

3.1.1 Použité rovnice

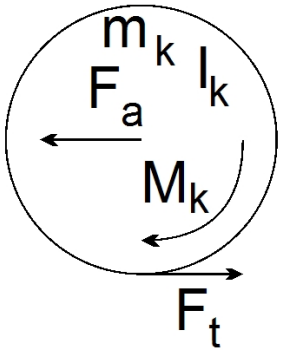
Použijeme zjednodušený model auta, které bude mít jedno kolo a karoserii (spojené rotační vazbou) navíc možnost pohybu auta je pouze ve směru x. V obrázku 3.1 a 3.2 jsou zobrazeny síly působící na jednotlivé části a veličiny použité v následujících rovnicích.

Pro daná tělesa platí rovnice pro kolo (3.1) a pro karoserii (3.2). (Uvažujeme zjednodušený pohyb - kolo neodskakuje, normálová síla v místě dotyku kola vozovky je konstantní.)

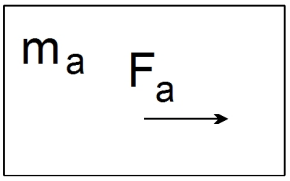
$$I_k \alpha = M_k - F_t r \tag{3.1}$$

$$F_t = (m_a + m_k)\, \ddot{x}_a \tag{3.2}$$

Zásadní nelinearita soustavy je v modelování velikosti tření. Použijeme Pacejkův model [14], který modeluje třecí sílu funkcí se čtyřmi parametry (B, C, D a E) v závislosti na skluzu s – vztah (3.3). Tyto parametry jsou konstanty definující



Obr. 3.1: Kolo



Obr. 3.2: Karoserie auta

vlastnosti pneumatiky a povrchu.

$$\mu = D \sin\left(C \arctan\left(Bs - E\left(Bs - \arctan\left(Bs\right)\right)\right)\right) \tag{3.3}$$

Výsledná třecí síla se pak spočítá podle (3.4), přičemž N (normálovou sílu) budeme dále považovat za konstantní a μ se mění v závislosti na druhu povrchu. V dalších úvahách a modelech předpokládáme, že třecí síla se bude měnit v závislosti na povrchu pouze lineárně (bude se měnit pouze parametr D „koeficientu tření“ (3.3). Třecí síla se tedy bude měnit v závislosti na skluzu a typu povrchu pouze lineárně. Tato závislosti je zobrazena na obrázku 3.3.

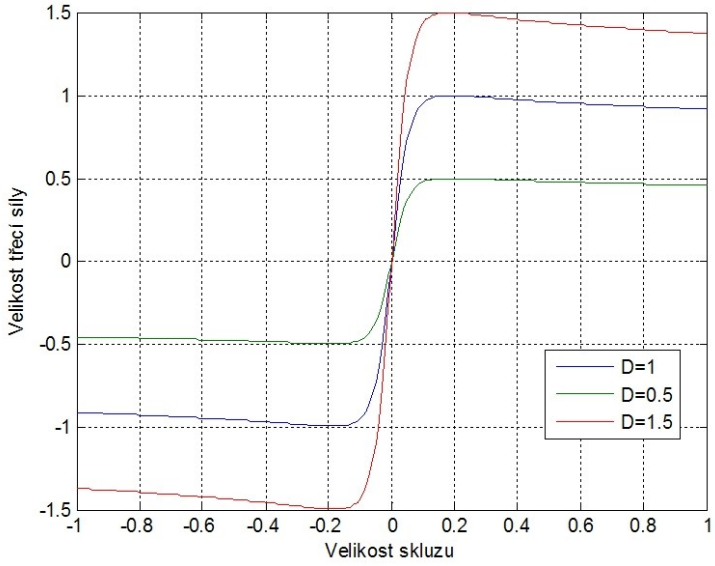
$$F_t = \mu N \tag{3.4}$$

Skluz, který je klíčovým parametrem pro výpočet třecí síly, v rovnici (3.3) se spočítá podle vztahu (3.5). Kde v_a je rychlost auta, v_b rychlost kola a δ je malá konstanta bránící dělení nulou.

$$s = \frac{v_a - v_k}{\max\left(\left|v_a\right|, \left|v_k\right|, \delta\right)} \tag{3.5}$$

Jak bylo uvedeno v úvodu, naším úkolem je odhadovat parametr B, charakterizující „přilnavost“ povrchu vozovky. S výhodou použijeme Kalmanův filtr, tak, že zavedeme nový stav jako parametr, který chceme odhadovat. Napíšeme tedy rovnice soustavy (3.6).

$$\begin{aligned} J\alpha &= M - F_t\left(v_a, v_k\right) Br \\ m_{\sum} a &= F_t\left(v_a, v_k\right) k_t \\ \frac{dx}{dt} &= v \end{aligned} \tag{3.6}$$



Obr. 3.3: Závislost velikosti třecí síly na skluzu.

Označení konstant v této rovnici je podle 3.1 a 3.2. A $m_{\sum}$ označuje celkovou hmotnost (karoserie a kola) a k_t koeficient tření povrchu. Rychlost otáčení přepočítáme na rychlost bodu na kraji kola (3.7).

$$v_k = \omega_k r \tag{3.7}$$

Tyto rovnice zdiskretizujeme aproximací spojité funkce mezi výpočetními kroky ΔT polynomem nultého stupně (zoh) [15] a přidáme parametr f (ekvivalent koeficientu tření povrchu) jako stav. Moment, kterým kroutí motor kolem, označíme jako vstup u (3.8).

$$\begin{aligned}
x_1 &= x_{auta} = x_1 + x_3 \Delta T \\
x_2 &= x_{kola} = x_2 + x_4 \Delta T \\
x_3 &= v_{auta} = x_6 \Delta T + x_3 \\
x_4 &= v_{kola} = -\frac{1}{J} F_t\left(x_3, x_4\right) x_5 r^2 \Delta T + x_4 + \frac{1}{J} r \Delta T u \\
x_5 &= f_{k.\ tření} = x_5 + q(t) \\
x_6 &= a_{auta} = \frac{1}{m} F_t\left(x_3, x_4\right) x_5
\end{aligned} \tag{3.8}$$

Proměnná $q(t)$ modeluje bílý šum (Do filtrovacího algoritmu zahrneme tuto informaci do matice Q - šum vstupující do stavů).

3.1.2 Vstupní data

Auto se pohybuje po povrchu s koeficientem tření 1, který se v průběhu brzdění tangent hyperbolicky změní na hodnotu 0.5 (přejezd na ledovatou plochu – menší přilnavost povrchu). Měřené veličiny jsou moment, kterým kroutí motor kolem, poloha kola (data z inkrementálního enkodéru) a měřené zrychlení auta (data z akcelerometru).

Pomocí výše uvedených rovnic modelu vygenerujeme data pro filtrování. Jako vstup do modelu použijeme moment motoru a koeficient povrchu vozovky. Výstupem je pak poloha kola a zrychlení auta. Nakonec k tomuto výstupu přičteme šum (simuluje náhodné chyby v měření).

3.2 Dosažené výsledky

Na první pohled není příliš zásadní rozdíl mezi výstupem filtru UKF a EKF. Dále jsou podrobněji popsané výstupy jednotlivých typů filtrů.

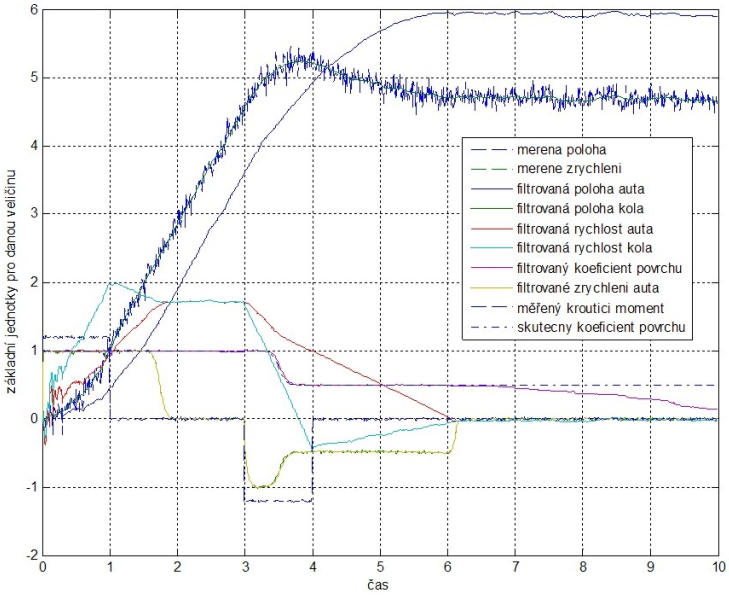
3.2.1 Výsledky filtrování pomocí EKF

Získaná data byla filtrována pomocí EKF Matice pro nastartování filtrování($P0$, $x0$) a matice šumu(Q a R) byly odhadnuté (v praxi nikdy nelze změřit přesně jak „zašumělé“ jsou měřené hodnoty, určení kovariance a variance šumu experimentálně by bylo náročné. Navíc tyto hodnoty nepotřebujeme znát úplně přesně pro správný chod filtrovacího algoritmu. Proto se dané hodnoty odhadují. Výsledný průběh filtrovaných veličin je v grafu 3.4.

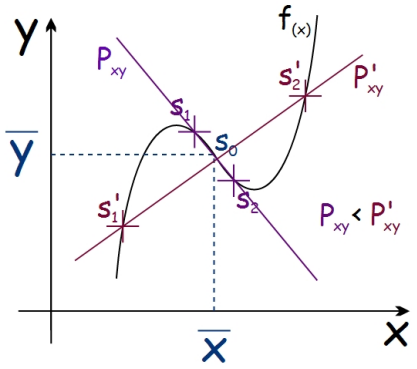
3.2.2 Výsledky filtrování pomocí UKF

Volba parametrů pro výpočet pomocí UKF je náročnější než filtrování pomocí EKF. Výpočet Kalmanova zesílení v EKF filtru závisí pouze na poměru velikosti matic Q a R. Při filtrování pomocí UKF je situace složitější.

Kovariance vstupu a výstupu a variance výstupu se počítají pomocí sigma bodů jejichž rozložení je odvozené od matic šumu a je závislé na absolutní hodnotě těchto matic. Tento problém je demonstrován na obrázku 3.5. Kde S jsou sigma body. Pokud je variance veličiny X malá, sigma body se zobrazí blízko střední hodnoty (S_0). Pokud je variance veličiny X větší, body se zobrazí dále (v obrázku znázorněné S'). Jak je vidět, výsledné kovariance P_{xy} budou pro různě vzdálené body různé. Navíc pokud by nastala situace s obdobně závažným rozdílem jako na obrázku 3.5, filtrační algoritmus by nefungoval. Znaménko u kovariancí je opačné.



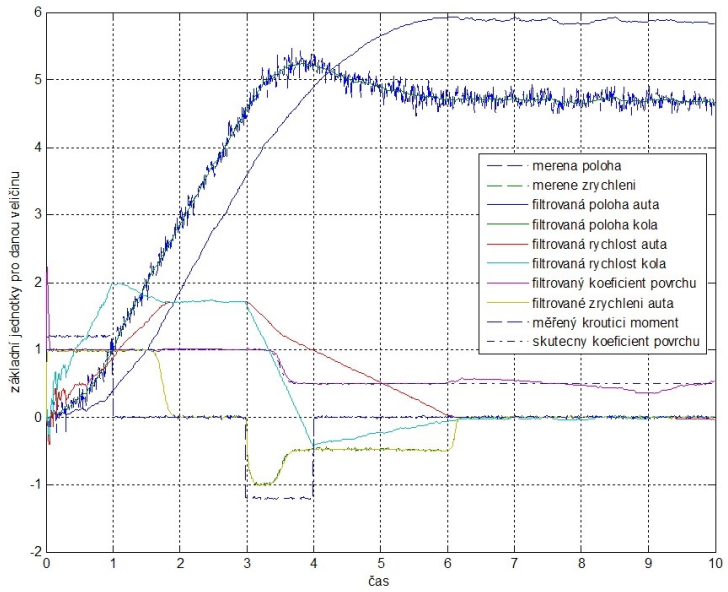
Obr. 3.4: Průběh veličin při filtrování EKF.



Obr. 3.5: Závislost vypočítané kovariance na poloze sigma bodů.

Pokud bychom počítali Kalmanovo zesílení podle vzdálených bodů, korekční zásah filtru by byl špatným směrem. Pokud by rozdíl odhadnuté veličiny a měřené byl kladný, výsledná filtrovaná hodnota by divergovala (po filtraci bychom dostali ještě horší výsledek než z měření).

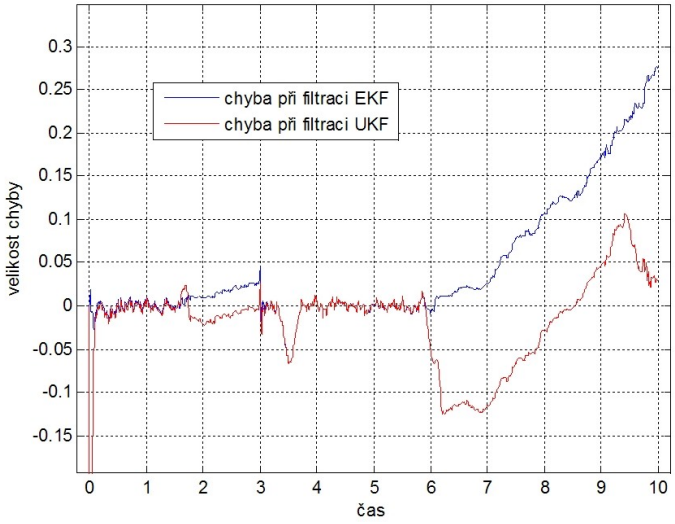
Získaná data byla filtrována pomocí UKF Matice $P0$, $x0$, Q a R byly stejně jako v předchozím případě odhadnuté. Průběh filtrovaných veličin je patrný z grafu 3.6.



Obr. 3.6: Průběh veličin při filtrování UKF.

3.2.3 Porovnání výsledků

Při pohledu na rozdíl mezi skutečným koeficientem povrchu a vyfiltrovanou hodnotou 3.7 není patrný příliš zásadní rozdíl. Pouze v zadní části kdy je výpočet velmi nestabilní (auto jede konstantní rychlostí a skluz se blíží nule) se algoritmus UKF se zdá být stabilnější v oblastech s vysokou nelinearitou (EKF diverguje, UKF se snaží vrátit). Navíc je vidět, že pro UKF se nám nepodařilo dostatečně dobře odhadnout počáteční variaci stavů (matice $P0$), než dojde k aktualizaci na správnou velikost objeví se ve filtrovaných datech chyba, která se ale poměrně rychle opraví. Tento problém je patrný u hodnot filtrovaných pomocí algoritmu UKF i EKF. Než dojde ke zkorigování kovariance stavů $P0$, filtrování v několika prvních iteracích neprobíhá příliš kvalitně (3.4, 3.6).



Obr. 3.7: Průběh veličin při filtrování UKF.

Způsob práce při generování kódu pro dsPIC a MPC555.

4

Implementace algoritmů do výpočetních jednotek

V předchozí části jsme se zabývali návrhem algoritmu, který bude v reálném čase zpracovávat filtrovat data. Tato data se používají pro nejnižší řídící smyčky (například řízení rychlosti, polohy nebo momentu). V dalším kroku je potřeba zvolit správnou platformu pro vyhodnocení a zpracování těchto dat. Přitom k dispozici je celá řada typů výpočetních jednotek, lišících se architekturou výpočetním výkonem a cenou.

Vývoj algoritmu pro jednotky s vysokým výkonem a podporující operace s čísly s plovoucí desetinnou čárkou je poměrně jednoduchý. Lze bez zásadních změn použít stejný kód jaký se používá pro simulace spuštěné na procesoru stolního počítače. Oproti tomu levnější varianta těchto jednotek umí pracovat pouze s číslem, které má menší počet bitů a neumí některé instrukce (operace s daty).

V praxi se pro vývoj nebo kusovou výrobu používají výkonné výpočetní jednotky. Použití levnějších platforem je výhodné pouze u sériové výroby, kde neúměrně větší náklady na vývoj a ladění kódu dostatečně kompenzuje snížení ceny jednoho kusu.

4.1 Generování kódu

Kód pro procesory lze napsat v několika jazycích. Obecně největšího výkonu dosahují programy napsané v assembleru, nicméně jelikož je tento programovací jazyk nejnižším, který lze použít (používají se přímo instrukce procesoru) je vývoj takovéhoto programu nesmírně zdlouhavý a náročný. Víceméně standardem pro programování „jednočipů“ je použití jazyka C, případně některé často používané procedury napsané v assembleru. Poměrně nový přístup spočívá v použití *Matlab Embedded Coder*.

4.1.1 Matlab Embedded Coder

Obrovskou předností *Matlab Embedded Coder* je rychlost, kterou generuje kód. Hlavní myšlenkou je poskytnout nástroj pro rapidní vývoj aplikací, tedy co nejvíce zkrátit vývojový cyklus, zejména ve smyčce testování, oprava chyb. Stačí změnit bloček v *Simulinku* a změní se celý program. Oproti tomu stejná změna v jazyce *C* znamená časově náročnou změnu velké části programového kódu.

Real-Time Workshop Embedded Coder je nástroj, který vygeneruje C kód, ten se dál přeloží pomocí C překladače do strojového kódu pro cílovou platformu. Hlavní rozdíl oproti kódu, který generuje samotný *Matlab* při simulacích, je jeho kompaktnost. Další rozdíl pak spočívá v přidání sekce kódu, která nastaví registry procesoru (použití periferií). Navíc přidá několik voleb pro optimalizaci generovaného kódu (například *inline* parametry místo konstant nebo „rozbalení“ for cyklů s malým počtem opakování).

4.1.2 Nastavení Matlabu pro generování kódu pro dsPIC

Bohužel *Matlab* neobsahuje *toolbox* s podporou pro *dsPIC* procesory, je proto nutné použít software třetí strany, v tomto případě *Kerhuel Toolbox*. Přesto, že se jedná o komerční software, obsahuje v sobě několik chyb, nicméně bločky pro jednoduché periferie fungují spolehlivě. Vzhledem k tomu, že je tento software zatím jen v beta verzi[1], lze čekat ještě další zlepšení. Navíc není podporována 64-bitová verze *Matlabu*, což by mělo být ve finální verzi bločků také odstraněno.

K vygenerování kódu pro dsPIC z Matlabu budeme potřebovat následující programy.

1. Matlab
   Je doporučeno použít operační systém Windows XP, u Windows Vista se objevují nekompatibility se softwarem. Kromě standardních součástí Matlabu je potřeba doinstalovat následující toolboxy.
   - Simulink
   - Real-Time Workshop
   - Real-Time Workshop Embedded Coder
   - Simulink Fixed Point (doporučeno)
2. MPLAB IDE
   MPLAB používáme především jako nahrávač, není tedy třeba instalovat všechny nástroje. Chceme-li šetřit místem na disku, zvolíme vlastní instalaci a ve volbě komponent necháme vybranou pouze podpora 16 bitových čipů a

[1] Informace platná v květnu 2010.

ICD3. Automatická instalace nainstaluje všechny komponenty. To je výhodné zejména, pokud budeme v budoucnu používat i jiné procesory.

Mplab automaticky nainstaluje do systému ovladače pro programátor ICD3 (pokud je při instalaci vybrán, což defaultně je). Ovladače pro ICD3 se nainstalují automaticky po zastrčení ICD3 do USB portu (stačí odklepat instalační dialogy). U Vist není instalace automatická, nicméně po ukončení instalace se automaticky zobrazí dokument s nápovědou k instalaci daného zařízení. Je doporučené používat pouze jeden a ten samý USB port pro připojení programátoru k počítači.

Je velmi výhodné používat nejnovější verzi MPLABu. Starší verze mohou obsahovat (obsahují) i závažné chyby. Nové verze lze zdrama (po registraci) stáhnout na stránkách výrobce [16].

3. Kompiler C30
   Studentská verze (zdarma) funguje dobře. Pro správnou funkci *Kerhuelova Toolboxu* je důležité použít verzi alespoň 3.11. Pozor verze na CD přiložená k vývojové desce je příliš stará a nefunguje s novou verzí *Matlabu*. Ke stažení ze stránek výrobce [16]. Instalace proběhne automaticky. Je důležité nainstalovat *C30 compiler* dříve než *Kerhuelův toolbox*.

4. Kerhuelův toolbox pro Matlab
   Aktuální verzi lze stáhnout z [17] zdarma dostupná verze obsahuje několik omezení (počet použitých periferií procesoru, navíc nepodporuje S funkce v projektu). Instalace se provede spuštěním staženého skriptu. Aby instalace proběhla v pořádku, je nutné spustit *Matlab* jako administrátor.

4.2 Vygenerování a nahrání kódu do dsPIC

4.2.1 Ověření správnosi instalace

K ověření funkčnosti a správnosti instalace je připraven demo program vytvořený při instalaci Kerhuelova toolboxu. Při instalaci se vytvoří složka `examples` (v místě kde je instalační skript). V té je soubor `dsPIC 33f Explorer16.mdl`. Po jeho otevření se provede automatická konfigurace parametrů (výběr překladače, nastavení Embedded Coder) poklepáním na bloček `Configure Model for dsPIC`. (Nepoužívejte ke konfiguraci bloček Configure Model for dsPIC1 - nastaví parametry špatně). Po nastavení parametrů zkompilujeme projekt kliknutím na ikonu `incremental build` v nástrojové liště okna modelu. Pokud všechno proběhlo úspěšně (žádná chybová hláška), měl by se vedle souboru projektu v pracovním adresáři objevit soubor s příponou „.hex“. Ten můžeme už přímo nahrát do procesoru. Pokud se kompilace nedaří, nejpravděpodobnější příčinou je příliš stará verze *C30* překladače.

Druhá varianta je kompilace C souborů vytvořených nástrojem *Embedded Coder* v *MPLAB IDE* tato varianta je vhodná, chceme-li použít nástroje pro debagování implementované v *MPLABU*. Velmi se nedoporučuje editovat přímo *C* kód. Připravíme se tím o výhody kódu generovaného ze *Simulinku* (drobná změna kódu nevyžaduje několik hodin programování). Při kliknutí na ikonku `incremental build` se kromě „.hex“ souboru vytvoří ještě C soubory. Ty můžeme přeložit v *MPLAB IDE* (integrovaným *C30* compilerem). Poté v `MPLABU` spustíme `Project` a vybereme `Project Wizard`. Zvolíme další a vybereme jednotku `33fJ256GP710` (námi používaný čip). V dalším kroku výběr nástrojů pro překlad vybereme `C30 Compiler`. Následuje importování souborů a knihoven. Vybereme všechny soubory vygenerované ve složce `Sources` u daného projektu. Průvodce skončí. Dále je potřeba přidat linkovací skript. Klikneme pravým tlačítkem myši na `linker script` (ve stromu složek projektu) a z kontextové nabídky vybereme `add` a zvolíme adresář `C:FilesC30`. Pokud bude potřeba měli by se přidat ještě soubory dsPIC knihovny, přidáme stejným způsobem jako předchozí zvolíme cestu: `C:\Program Files\Microchip\MPLAB C30\li` (Překlad většinou proběhne i bez přidání této knihovny). A pak už jen `Project` a položka `Make`, program se přeloží. Takto přeložený program se nahraje stejně, jako se nahrává *.hex* soubor do procesoru z prostředí *MPLABu*.

Efektivita s jakou je program ze *Simulinku* přeložen závisí na kvalitě vytvořeného *C* kódu *Matlabem* a v další fázi na překladači do strojového kódu z jazyka *C*. Kvalitu překladu do strojového kódu prakticky nemůžeme ovlivnit. Pouze lze nastavit optimalizaci (pro rychlost) pomocí přepínačů překladače (zadávají se jako parametry pro překlad). Lepších výsledků je možné dosáhnout pouze zakoupením kvalitnějšího překladače. Oproti tomu efektivitu *C* kódu generovaného *Matlabem* můžeme zásadně ovlivnit použitím vhodných datových typů jak bude ukázáno později.

4.2.2 Nahrání programu do dsPIC

Vytvořený program nahrajeme do procesoru pomocí ICD 3. Otevřeme tedy `MPLAB IDE`. V nástrojové liště vybereme `configure` pak `select device` a v rozbalovacím menu vybereme `dsPIC33FJ256GP710`. V dalším kroku je potřeba vybrat programátor. V nástrojové liště `programmer` pak `select programmer` a vybereme `MPLAB ICD3`. Po připojení programátoru ke konektoru na desce je MPLAB nastavený pro použití s vybraným procesorem. Otevřeme v předchozím kroku vytvořený binární soubor (přípona „.hex“) volba `File` v hlavní nástrojové liště pak `import`, najdeme soubor a `Ok`. Nahrání importovaného souboru do procesoru se provede volbou `programmer` a následně položka `programm` (nebo ikonka `programm` na hlavní nástrojové liště).

4.3 Procesor MPC555

Jak je patrné z výše uvedeného, výpočetní výkon procesorů *PIC* je nedostatečný pro složitější výpočty (abs, nelineární řízení apod.), které nelze jednoduše a efektivně počítat pomocí celočíselné aritmetiky. Proto současně používáme další výpočetní jednotku, která umí operace s čísly typu *double*. Procesor *MPC555* osazený na kartě od společnosti *PhyTec* má integrované jednotky pro výpočet v plovoucí desetinné čárce a umí pracovat s 32 bitovými čísly.

4.3.1 Použití výpočetní jednotky MPC555

Pro používání karty s procesorem *MPC555* stačí do *Matlabu* doinstalovat balíček `Target Support Package FM5`. Navíc je potřeba překladač (za jazyka C do strojového kódu pro daný procesor) například *CodeWarrior* který se dodává i s překladačem přímo pro tento procesor.

Při instalaci je potřeba stáhnout aktualizace některých souborů a ty ručně přehrát, nicméně poté karta funguje výborně. Postup jak kartu připojit k počítači s *Matlabem* a *Windows xp* je popsána na stránkách [18]. Jako nahrávač používáme čip integrovaný přímo na kartě (*BDM*), který komunikuje s *CodeWarriorem* po paralelním portu. Tento nahrávač se zapíná pomocí pinu přímo na desce karty. Další obrovskou výhodou oproti řešení pomocí *PIC* procesorů je propojení s *Matlabem*. Projekt v *Simulinku* se automaticky nahraje a spustí po přeložení. Navíc *Real Time Target* bločky jsou vytvořené firmou *Matworks*, takže neobsahují prakticky žádné chyby.

Karta *MPC555* je vybavena dvěma typy pamětí. Použití paměti *RAM* je vhodná k testování. Relativně rychle se nahrává a má neomezený počet cyklů přepsání. Oproti tomu nahrání paměti *FLASH* trvá déle, ale data v ní zůstanou i po odpojení napájení.

Při používání této karty je potřeba mít na paměti, že nesmíme zapomenout nahrát „bootovací“ sektor ze kterého se spouští kód vygenerovaný *Matlabem*. Ten obvykle stačí nahrát pouze jednou ale pokud se celá paměť přepíše nebo smaže, je potřeba jej nahrát znovu.

4.4 Aritmetika s pevnou desetinnou čárkou

Simulink standardně používá datový typ *double*. Bohužel, zejména levné procesory tento datový typ nepodporují, jsou pouze 16 bitové a k tomu ještě neumí pracovat s desetinou čárkou. Pokud chceme použít operace s datovým typem double, na procesoru s celočíselnou aritmetikou musí se složitě simulují pomocí celočíselných

operací, což je nesmírně neefektivní. (Podobná situace byla ještě nedávno[2] i na stolních počítačích, než se stala samozřejmostí podpora *x86* instrukcí)

4.4.1 Simulace desetinných čísel celými

Pokud bychom chtěli používat celočíselnou aritmetiku místo desetinné, znamenalo by to přenásobit rozsahy všech proměnných, tak, aby se s dostatečnou přesností daly zaokrouhlit na celá čísla. Například vstupní napětí, které je mezi hodnotami 0 až 1, bychom „převedli“ tak, aby pokrylo interval 0 až 2 na 16. Což nevypadá jako problém a také není. První potíže nastanou, budeme-li chtít taková čísla sčítat nebo odčítat. Pokud bychom znali intervaly všech vstupních hodnot, mohli bychom správným „škálováním“ docílit funkčnosti programu (tak aby hodnoty při sčítání a odčítání nepřetékali). Bohužel při naší práci s daty potřebujeme operace násobení a dělení. V takovém případě převod na celá čísla a následná správná interpretace výsledku je i pro jednoduché úlohy velmi náročná.

Naštěstí přišel Matworks s nástroji, které se snaží usnadnit práci s desetinnými čísly při použití celočíselné aritmetiky (*Fixed Point Toolbox* a *Simulink Fixed Point*). Při překladu se použije sada nástrojů pro co nejefektivnější převod operací do celočíselné aritmetiky (sečtení celých čísel a následné bitové posuvy desetinných bitů, je-li to třeba).

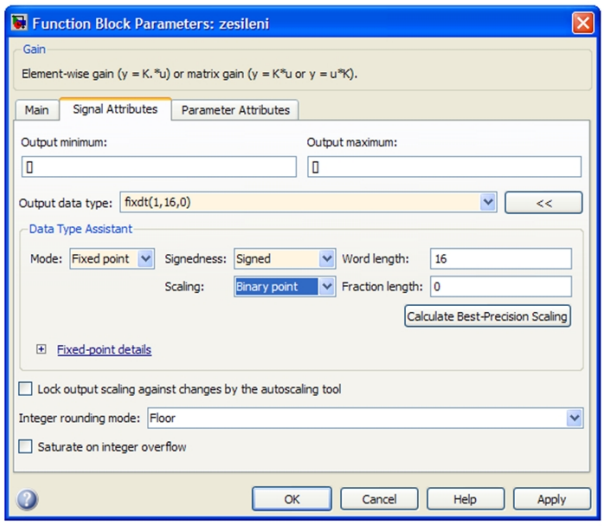
Datový typ fixdt

Simulink fixed point datový typ se použije tak, že v jakémkoliv bločku Simulinku zvolíme `Function Block Parameters` (otevře se poklepáním na bloček) a záložku `Signal Attributes` a v položce `Output Data type` zvolíme fixdt. Tento dialog je zobrazený na obrázku 4.1

Pomocí datového typu fixdt lze definovat „změnu vzdálenosti celých čísel“ nebo ještě přidat k tomuto zobrazení posunutí. Použití posunutí není příliš výpočetně efektivní, proto budeme přednostně používat puoze změnu rozsahu. Desetinné číslo se popíše třemi parametry *fixdt(a,b,c)*, kde *a* je znaménkový byt. Znaménkový byt nastavíme na nulu, pokud pracujeme pouze s kladnými čísly, jinak je jednička. Parametr *b* je počet bitů proměnné, při použití šestnáctibytového procesoru nemá smysl používat jiné hodnoty než 16 eventuelně 32 a 48 (použije se více registrů na proměnnou). Poslední parametr *c* určuje počet desetinných bytů. Pokud například napíšeme fixdt (1,8,2) datový typ bude mít osm bytů z toho první je znaménkový a dva desetinné (4.1). V této rovnici je bx x – tá cifra v bytu.

$$b1 \cdot (-1) \cdot 2^5 + b2 \cdot 2^4 + ... + b7 \cdot 2^{(-1)} + b8 \cdot 2^{(-2)} \tag{4.1}$$

[2]První procesor firmy Intel pro osobní počítač s integrovanou jednotkou pro výpočet v plovoucí desetinné čárce s označením 80486 (nebo i486) byl v prodeji od podzimu roku 1989.



Obr. 4.1: Dialog pro změnu datového typu v Simulinku.

V Matlabu lze nastavit ekvivalentní typ datových proměnných jako v Simulinku příkazem:

```
fi (číslo,1,8,2)
```

což je základní verze příkazu umožňující nastavit vyjádření hodnoty *číslo* pomocí znaménkového osmibitového datového typu s dvěma desetinnými bity.

Použití průvodce

Ruční nastavení proměnných a modelu je relativně náročné. Musíme vědět, jak velké bude největší číslo, se kterým budeme pracovat, abychom optimálně využili bity. (Pokud zvolíme příliš velký rozsah, bude několik bitů následujících za znaménkovým nulových, což je škoda, protože můžeme zvýšit počet desetinných bitů a tím i přesnost.)

Simulink obsahuje průvodce, který (pokud se mu to povede) nastaví datové typy automaticky. Šance na úspěšný automatický převod lze zvýšit několika úpravami modelu. První z nich je co nejjednodušší převáděný model, vytvoříme subsystémy a v těch teprve budeme měnit datové typy. Dále je vhodné převáděný úsek uzavřít mezi bločky convert. Před spuštěním průvoce je lepší projekt uložit. Pokud se průvodce v půlce zasekne (což se stává poměrně často) musíme ručně vracet už změněné nastavení některých datových typů. Tohoto průvodce spustíme z menu `Tools` podmenu `Fixed Point` a `Fixed Point Advisor`. Průvodce navíc sám popisuje chyby a nabízí jejich řešení, takže lze převod jednoduše provést.

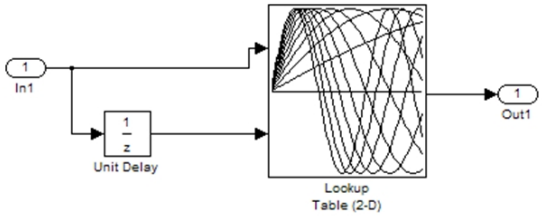
Dalším nástrojem je Fixed Point Tool. Spouští se ze stejného umístění jako Fixed Point Advisor. Slouží zejména k testování správného nastavení datových typů.

Umožňuje sledovat přetečení proměnných a zaokrouhlovací chyby (porovná data spočítaná při nastavení všech proměnných na typ *double* a na *fixdt*).

4.5 Výpočetní náročnost jednoduchých operací

4.5.1 Paměť

Velikost paměti obvykle nepředstavuje vzhledem k typu programů, které používáme na této platformě, problém. Pro představu byla do čipu s 256 tisíci byty paměti nahrána tabulka s tisíci hodnotami 4.2. Hodnoty v tabulce jsou vygenerovány pomocí



Obr. 4.2: Tabulka hodnot v Simulinku.

náhodné funkce. Navíc je důležité o jaký datový typ se jedná. Pro testování byl použit procesor *33fJ256GP710* a výsledek je zobrazen níže.

Použitý datový typ	Obsazená paměť
double	15%
int16	7%

Tab. 4.1: Velikost obsazené paměti při použití různých datových typů

4.5.2 Překlad ze Simulinku do C

Pro získání představy o výpočetním výkonu používaných bylo vytvořeno několik jednoduchých úloh, které demonstrují časovou náročnost na použitých jednotkách.

Sčítání

Jednou z nejpoužívanějších operací je sečtení čísel. Abychom mohli sledovat dobu výpočtu byl vytvořen jednoduchý program v *Simulinku*, kdy je proměnná různého

typu inkrementována o jedničku. Navíc na této úloze je dobře vidět jak ovlivní použití datového typu v *Simulinku* vygenerovaný *C* kód. Kvůli jednoduššímu zápisu byla použita pro vygenerování kódu vnořená Matlabovská funkce.

Označování datových typů v jazyce *C* je poměrně intuitivní. Jejich detailnější popis je například v [19].

Sčítání desetinných čísel (reálných) Datový typ *double* je výchozím datovým typem. Pokud neoznačíme použité proměnné jinak, standardně se definují jako *double*. Níže je část kódu, která provede přičtení jedna k datovému typu double.

```
s=s+1;
```

Odpovídající vygenerovaný C kód je následující.

```
real_T rtb_y;
rtb_y++;
```

Pokud chceme použít datový typ *single* musíme jej v Matlabu definovat. Jedna z možností jak toto udělat je přiřazení určitého typu do proměnné při inicializaci (prvním použití). Inkrementace o jedničku se dál provede stejným způsobem jako při použití typu *double*.

```
s=single(promenna);
s=s+1;
```

Vygenerovaný C kód je pak stejný až na jiný použitý typ.

```
real32_T rtb_y;
rtb_y++;
```

Sčítání celých čísel V zásadě postupujeme stejně jako při použití datového typu *single*. Pouze změníme typ proměnné při prvním použití. V tomto případě celočíselné šestnácti bytové číslo. Tento postup je stejný pro použití libovolného celočíselného typu (*int32*, *uint32*, ...).

```
s=int16(promenna);
s=s+1;
```

Vygenerovaný C kód je trochu překvapiví. Přičtení se provede v datovém typu s vyšším rozsahem a po kontrole přetečení se výsledek převede do proměnné požadovaného typu. Při použití *Simulink* bločků je možné tuto ochranu snadno vypnout zaškrtnutím `Saturate on integer overflow` na kartě `Function block parameters`.Vypnutí ochrany přetečení u kódu generovaného z *Embedded Matlabu* je komplikovanější. Je nutné definovat jakým způsobem se bude provádět sčítání v (*fimath* objektu). Bohužel tento objěkt není zatím příliš podporován v *Embedded Matlab* funkcích.

```
int16_T rtb_y;
int32_T tmp_0;

tmp_0 = (int32_T)rtb_y + 1L;

if (tmp_0 > 32767L) {rtb_y = MAX_int16_T;}
               else {rtb_y = (int16_T)tmp_0;}
```

Sčítání desetinných čísel pomocí celočíselné aritmetiky Podobné použití jako v předchozích příkladech. Definujeme datový typ s dvěma desetinnými místy a znaménkovým bitem. V matlabu definujeme datový typ proměnné ke které budeme přičítat i datový typ přičítaného čísla (oba operandy musí mít stejný datový typ).

```
s=fi(promenna,1,32,2);
s=s+fi(1,1,32,2);
```

V jazyku *C* samozřejmě žádná proměnná s dvěma desetinnými byty neexistuje. Sečtení se tedy provede v celočíselných proměnných s tím že dva poslední byty se považují za desetinné (číslo se musí násobit 2^2).

```
int32_T rtb_y;
rtb_y += 4L;
```

Pokud chci toto desetinné číslo převést zpět na celé číslo (zahodit desetinné bity), stačí provést bitový posun doprava (zleva se doplní nuly). Navíc pro tuto operaci je na většině procesorů přímo instrukce, takže se provádí velmi rychle.

```
rtb_y >> 2;
```

Jak je zřejmé, není příliš rozumné používat jiný rozsah datového typu než celočíselný násobek obvyklých typů používaných v jazyce *C*. Pokud použijeme například osmnácti bitové číslo, bude se používat pouze prvních osmnáct bitů v třiceti dvou bitové proměnné. V příkladě níže je k osmnáctibitovému číslu přičtena jednička.

```
int32_T rtb_y;
int16_T tmp;

tmp = (int16_T)(((rtb_y << 16U) + 65536L) >> 16);
rtb_y = (int32_T)tmp & 131072L ? (int32_T)tmp |
        -131072L : (int32_T)tmp & 131071L;
```

4.5.3 Doba výpočtu na různých typech procesorů

Způsob měření doby výpočtu

Zjištění doby kterou se vykonává jeden výpočetní krok na kartě MPC555 je velmi jednoduché. V nastavení *Realtime Embedded Coderu* zaškrteneme volbu `Execution Profiling` a přidáme odpovídající bloček `Execution Profiling` podle rozhraní, které budeme používat pro komunikaci karty s počítačem.

Data o výpočtu získáme z karty zadáním následujícího příkazu do *Matlabu*.

```
> PROFDATA = PROFILE_MPC555('serial', 'BitRate', 115200)
```

Matlab poté vygeneruje *html* zprávu, ve které je mimo jiné průměrná doba jednoho výpočetního cyklu.

Získání doby výpočtu procesoru PIC je o něco náročnější. K dispozici v *Kerhuel Toolboxu* je nástroj, který po skončení doby výpočetního kroku odečte čas z časovače který vyvolává přerušení spouštící nový výpočetní krok. Pokud tedy víme, při jaké hodnotě se spouští další výpočetní krok a hodnotu, při které skončila výpočetní operace, můžeme na základě jejich poměru a znalosti času periody výpočtu spočítat skutečnou dobu výpočtu.

Naměřené hodnoty

Inkrementace čísla V tabulce 4.2 je zobrazena doba výpočtu při použití různých datových typů, ve kterém je číslo zvýšeno o jedničku.

Použitý typ procesoru	Datový typ	doba výpočtu
dsPIC	int16	5,66e-7s
	int32	7,66e-7s
	single	9,30e-6s
	double	9,30e-6s
MPC555	int16	6,704e-7s
	int32	5,824-7s
	single	3,744e-7s
	double	4,504e-7s

Tab. 4.2: Doba inkrementace čísla v závislosti na použitém datovém typu a platformě.

Jak je vidět z naměřených výsledků, doba inkrementace celého i desetinného čísla je srovnatelná při použití procesoru *MPC555* s numerickou jednotkou. Oproti

tomu *dsPIC* takový numerický koprocesor nemá. Pokud simuluje sčítání desetinných čísel doba výpočtu je řádově desetkrát pomalejší.

Maticové operace Jednou ze základních matematických operací v *Matlabu* jsou operace s maticemi. Dále budeme sledovat jak dlouho trvají základní maticové operace.

Násobení matic V tabulce 4.3 je provedeno shrnutí doby výpočtu při násobení dvou matic.

Použitý typ procesoru	Datový typ	Velikost matice	Doba výpočtu
dsPIC	double	1x1	3,3667e-5s
		5x5	8,6215e-4s
		10x10	0,007s
		15x15	0,0243s
	fixdt	1x1	4,1e-7s
		5x5	5,1628e-5s
		10x10	3,0819e-4s
		15x15	9,4199e-4s
MPC555	double	1x1	$<$ 1e-6s
		5x5	6,956e-4s
		10x10	3,6336e-3s
		15x15	0,0108s
	fixdt	1x1	$<$ 1e-6s
		5x5	1,996-4s
		10x10	1,3916e-3s
		15x15	4,95e-3s

Tab. 4.3: Doba násobení matic v závislosti na jejich velikosti a použité platformě

Mohlo by se zdát zvláštní, že násobení matic je rychlejší na procesoru *dsPIC*. To je dané zejména tím, že použitý datový typ *fixdt* se zpracovává na různých procesorech s různou přesností (*dsPIC* výchozí 16 bitů, *MPC555* výchozí 32 bitů). Jelikož implementace násobení vektorů není tak přímočará jako sčítání (velmi rychle přetékají datové typy) použití vyšší přesnosti mnohonásobně zvýší nároky na výpočetní výkon.

Inverze matice V tabulce 4.4 je uvedena doba výpočtu inverzní matice s čísly v plovoucí desetinné čárce. Je docela překvapivé, jak malý rozdíl je v době výpočtu na obou platformách při použití stejného datového typu. K inverzi matice se obvykle používá algoritmus vycházející z L-U dekompozice [20]. Jak je vidět z naměřených časů, implementace v plovoucí desetinné čárce je poměrně efektivní. Doba výpočtu je pouze několikrát pomalejší, než při použití operací v plovoucí desetinné čárce.

Použitý typ procesoru	doba výpočtu
MPC555	0,00168s
dsPIC	0,0061s

Tab. 4.4: Doba výpočtu inverzní matice 10x10.

4.6 Výpočet speciálních funkcí

Velmi často se řeší výpočet určité funkce, která není příliš jednoduchá (nelze ji jednoduše na používaném definičním oboru aproximovat polynomem prvního stupně). Jako příklad pro srovnání použijeme funkci sinus. Potřebujeme použít tuto funkci v intervalu -90 .. 90 stupňů. První možnost je použít přímo zabudovanou funkci.

4.6.1 Zabudovaná funkce

Pokud použijeme funkci *Matlabu* $sind(hodnota)$ vygenerovaný C kód bude následující.

```
eml_x = (real32_T)floor(rtb_DataTypeConversion / 90.0F + 0.5F);
rtb_DataTypeConversion -= eml_x * 90.0F;
eml_x = (real32_T)floor(eml_x + 0.5F);
rtb_DataTypeConversion = (real32_T)sin(1.745329238E-002F *
rtb_DataTypeConversion);
tmp = fmod(floor((real_T)rtb_DataTypeConversion), 65536.0);
```

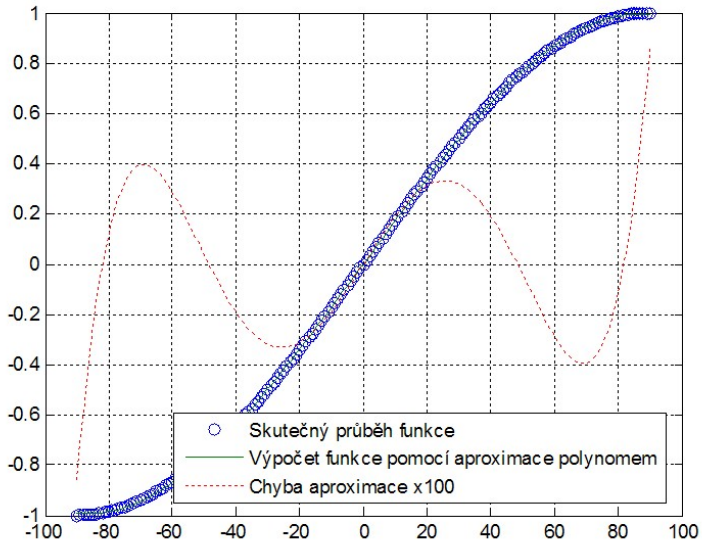
Je zřejmé, že v knihovně C je definovaná funkce sinus pro radiány a musí se tedy převádět. Pokud bychom použili funkci sinus s argumentem v radiánech $sin(hodnota)$ vygenerovaný kód bude.

```
tmp = fmod(floor((real_T)(real32_T)sin((real32_T)
```

Pokud se podíváme na dobu výpočtu funkce sinus ve stupních a sinus v radiánech, výpočet v radiánech je o 22μs rychlejší při použití procesoru dsPIC. Na procesoru MPC555 je rychlejší o 1,4μs.

4.6.2 Aproximace polynomem

Jelikož je funkce poměrně „klidná“, k její aproximaci stačí polynom třetího stupně, přičemž tento polynom aproximuje danou funkci na zvoleném definičním oboru s přesností na dvě desetinná místa. Tato funkce je znázorněna na obrázku (4.3).



Obr. 4.3: Průběh funkce aproximované polynomem a její chyba

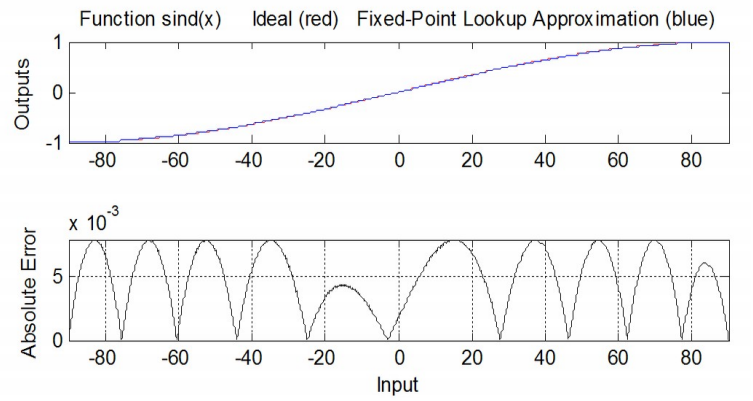
Vygenerovaný *C* kód při aproximaci polynomem je následující.

```
tmp = -7.7004763292330999E-007 * rt_pow_snf((real_T)eml_x, 3.0) +
-1.5766433249266400E-020 * rt_pow_snf((real_T)eml_x, 2.0) +
1.7253552992155100E-002 * (real_T)eml_x + 3.2493953697218003E-017;
```

4.6.3 Aproximace tabulkou

Existují dva způsoby, jak sestavit tabulku. Buď optimalizujeme umístění bodů tabulky pro rychlost výpočtu, nebo pro malé obsazení paměti.

Minimální počet bodů V tomto případě je tabulka generována tak, aby tam kde je velká druhá derivace funkce (hodně se mění gradient) byly body hustěji a v místě kde je gradient konstantní řidší. Tento přístup zajistí minimální chybu oproti aproximované funkci při co nejmenším počtu tabulkových bodů. Průběh funkce a její



Obr. 4.4: Průběh funkce aproximované tabulkou a její chyba

chyby je zobrazen na obrázku (4.4). Funkce je definována v deseti bodech, hodnoty mezi nimi se dopočítávají lineární interpolací.

Zápis této funkce v Matlabu je následující.

```
y=interp1(xdata,ydata,xin);
```

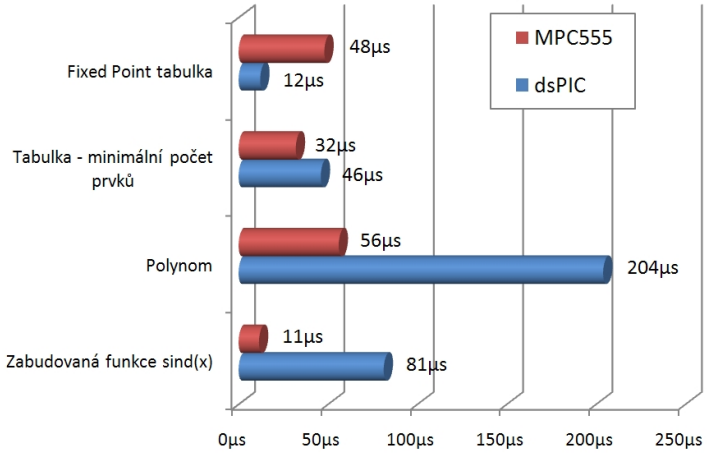
Tato funkce vygeneruje poměrně dlouhý *C* kód.

```
rtb_y = (rtNaNF);
eml_exitg = 0L;
if (!(rtb_DataTypeConversion1 > 90)) {
 eml_low_i = 1UL; eml_low_ip = 2UL; eml_high_i = 10UL;
 while (eml_high_i > eml_low_ip) {
  eml_mid_i = eml_low_i + eml_high_i >> 1;
  if ((real_T)rtb_DataTypeConversion1 >=
  tmp_0[(int16_T)eml_mid_i - 1])
   {eml_low_i = eml_mid_i;
    eml_low_ip = eml_mid_i + 1UL;}
  else {eml_high_i = eml_mid_i;}}
 eml_exitg = (int32_T)eml_low_i;}
if ((uint32_T)eml_exitg > 0UL) {
 eml_np = eml_exitg + 1L;
 rtb_y = (real32_T)tmp_1[(int16_T)eml_exitg - 1] + ((real32_T)
 rtb_DataTypeConversion1 - (real32_T)tmp_0[(int16_T)eml_exitg - 1]) /
  (real32_T)(tmp_0[(int16_T)eml_np - 1] -
  tmp_0[(int16_T)eml_exitg - 1]) *
  (real32_T)(tmp_1[(int16_T)eml_np - 1] -
  tmp_1[(int16_T)eml_exitg - 1]);}
```

„Fixed point“ tabulka Matlab nabízí generování tabulky optimalizované pro celočíselnou aritmetiku. Její hlavní rozdíl spočívá v tom, že body v tabulce jsou rozložené rovnoměrně. To výrazně urychlí výpočet interpolovaných hodnot. Vygenerovný C kód této funkce je ještě o něco delší. Počet hodnot tabulky je $2^4 + 1$. (obecně pro optimální rychlost výpočtu je vhodné volit počet bodů $2^N + 1$, kde N je celé číslo. Je vidět, že pro srovnatelnou přesnost je potřeba použít téměř dvojnásobný počet bodů, pokud je rozložíme rovnoměrně.

4.6.4 Srovnání výpočetní rychlosti pro různé typy aproximací

Srovnání doby výpočtu funkce sinus různými metodami je v tabulce (4.5).



Obr. 4.5: Doba výpočtu funkce sinus různými metodami.

Z naměřených dat je zřejmé, že pro výpočet funkce sinus na procesoru podporující operace s plovoucí desetinnou čárkou je výhodnější použít přímo zabudované funkce, které jsou dostatečně optimalizované pro rychlost výpočtu. Oproti tomu při výpočtu na procesoru s celočíselnou aritmetikou je nejlepší použít tabulku optimalizovanou přímo pro tento typ procesorů. Navíc aproximace funkce pomocí polynomu i relativně nízkého řádu (třetího) je pro výpočet velmi neefektivní (pomalé).

Obecně lze říci, že při používání procesoru *MPC555* je výhodné používání přímo funkcí definovaných v *Matlabu*. Pro generování kódu pro *dsPIC* je vhodné použít alternativní výpočet, přičemž nejlépe vychází aproximace funkce pomocí tabulky optimalizované pro procesory s celočíselnou aritmetikou.

Návrh regulátoru momentu implementovatelného do dsPIC jednotky.

5 Regulátor momentu

Vzhledem k charakteru řízení experimentálního vozidla je požadavek na momentové řízení pohonu kol. Implementovaný algoritmus vyšší smyčky řízení pohonu mění velikost hnacího momentu kola u kterého dochází k prokluzu.

5.1 Návrh PID regulátoru

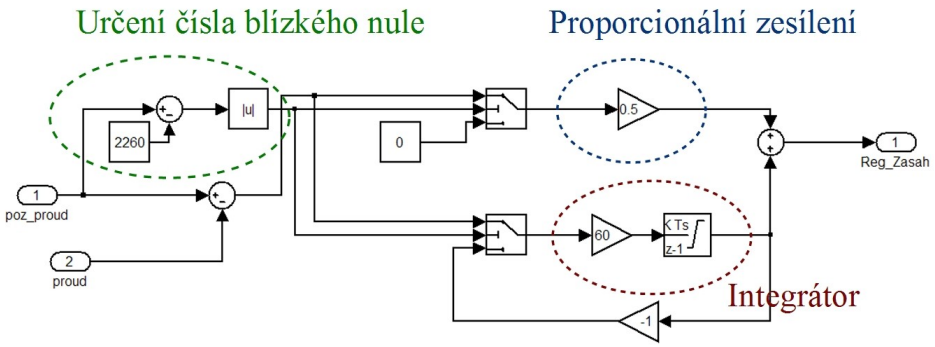
Nejjednodušším možným typem regulátoru je PID regulátor. Oproti jiný metodám návrhu regulátoru nepotřebujeme znát informaci o regulované soustavě (její model). Díky tomu je návrh velmi rychlý.

Metodou ladění jsme zvolili konstanty zesílení pro proporcionální a integrační složku. Navíc je tento regulátor doplněn o „pásmo necitlivosti“ v okolí nuly. Jeho účelem je vypnout regulátor, pokud je požadovaný proud blízký nulové hodnotě (brání „chvění“ kola pokud stojí na místě). To se provede tak, že do P složky regulátoru se přivede nulová hodnota a do I složky se přičte záporná velikost výstupu integrátoru. Tím docílíme toho, že obě složky budou mít nulový výstup. Tento mechanizmus je dobře patrný z obrázku 5.1.

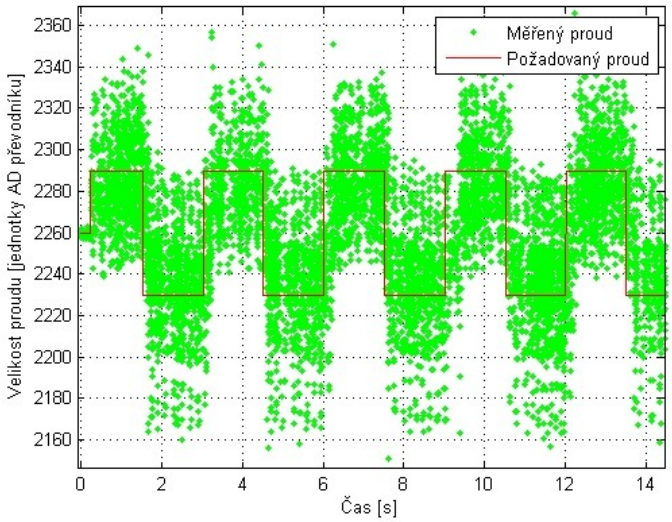
K regulátoru je přidán jednoduchý filtr měřeného proudu. Během jednoho pracovního cyklu se naměří z AD převodníku 16 hodnot, ze kterých se spočítá průměrná a tu použijeme jako zpětnou vazbu do regulátoru.

Schéma PID regulátoru je na obrázku 5.1. Vzhledem k principu návrhu lze použít přímo celočíselné datové typy. Velikost proudu pro zpětnou vazbu je z AD převodníku v datovém typu *uint16* a požadovaná hodnota pro střídu PWM je typu *int16*. Jediné místo, kde vznikají platné desetinné bity, je při násobení P složky. Vzhledem k tomu, že výsledek požadované střídy musí být celé číslo, je v tomto případě možné desetinné bity zahodit.

Na obrázku 5.2 je zobrazen průběh regulovaných veličin při skokové změně požadovaného momentu. Jak je patrné, proud je velmi zarušený (spínání výkonových



Obr. 5.1: Schéma použitého PI regulátoru



Obr. 5.2: Odezva soustavy na řídící signál

tranzistorů). Abychom mohli dosáhnout lepších výsledků, je mimo jiné nutné použít lepší způsob filtrace proudu.

5.2 Sestavení modelu

Existují dva přístupy modelování systému. První z nich je fyzikální modelování. Tehdy se na základě znalosti fyzikálních rovnic popisujících chování systému sestaví model. Druhým identifikace systému, tedy hledáním takové funkce, která popisuje závislost mezi známým vstupem a výstupem soustavy. Při tvorbě modelu vyjdeme ze známých fyzikálních stavů a konstanty určíme pomocí metod odhadu parametru.

5.2.1 Fyzikální rovnice

Vyjdeme z obecně známých rovnic (5.1) popisujících dynamické chování stejnosměrného motoru. Ve kterých I je moment setrvačnosti, C_Φ konstanta budicího toku motoru, R odpor vinutí, L indukčnost vinutí a u napětí na svorkách motoru.

$$\begin{aligned} I\frac{d\omega}{dt} &= C_\Phi i - M_b \\ u &= Ri + L\frac{di}{dt} + C_\Phi \omega \end{aligned} \tag{5.1}$$

Tento stavový model bude v konečné fázi implementován v diskrétním regulátoru s délkou výpočetního kroku 0,005s. Časová konstanta vinutí je $\tau = \frac{L}{R}$. Indukčnost vinutí L můžeme snadno změřit RLC metrem stejně jako jeho odpor R. Naměřené velikosti jsou $R = 1,8\Omega$ a $L = 0,0021H$. Výsledná časová konstanta vinutí je zhruba 0,00112s. Jelikož je tato konstanta daleko menší než délka časového kroku můžeme dynamiku proudu v modelu zanedbat.

Navíc zátěžný moment budeme uvažovat pouze viskózní tření, které se modeluje lineární závislostí na rychlosti (5.2).

$$M_b = b\omega \tag{5.2}$$

Z měřených dat mám k dipozici ještě informaci o natočení kola, do stavového modelu přidáme ještě polohu. Jak už bylo uvedeno výše, máme k dispozici informaci o velikosti proudu. Do stavového modelu tedy zavedeme „redundantní“ stav proudu. Tyto rovnice převedeme do diskrétního tvaru metodou *ZOH* (5.3).

$$\begin{aligned} \omega_{k+1} &= \omega_k + \Delta t\frac{C_\Phi}{I} i_k - \Delta t\frac{b}{I}\omega_k \\ \varphi_{k+1} &= \varphi_k + \Delta t\omega_k \\ i_{k+1} &= \frac{1}{R}u - \frac{C_\Phi}{R}\omega_k \end{aligned} \tag{5.3}$$

V modelu označíme neznámé konstanty, které budeme hledat (5.4).

$$\begin{aligned} \omega_{k+1} &= a_1\omega_k + a_2 i_k \\ \varphi_{k+1} &= \varphi_k + b_1\omega_k \\ i_{k+1} &= d_1 u - c_1\omega_k \end{aligned} \tag{5.4}$$

5.2.2 Převod jednotek měřených veličin na fyzikální

Správný převod měřených jednotek na fyzikální je důležitý, zejména pokud budeme dále používat filtrovaný signál. Například může být výhodné, když můžeme přepočítat úhel natočení kola přepočítat na ujetou vzdálenost a podobně.

Měření natočení kola

Pro začátek je důležité zvětšit velikost proměnné, do které se ukládá počet tiků z enkodéru umístěných na kolech. Při použití 16 bitové proměnné dojde poměrně rychle k jejímu přetečení, zvýšíme-li její velikost na 32 bitů, tato velikost už bude dostatečná (přeteče až po pěti dnech jízdy auta plnou rychlostí). Vzhledem k tomu, že inkrementace proměnné polohy se provádí vždy při vyvolání přerušení enkodéru a použitý procesor nepodporuje přímo práci s 32 bitovými proměnnými není vhodné použít 32 bitovou proměnnou přímo, ale její inkrementaci provádět pouze před prováděním výpočetního kroku.

Jedno otočení kola kolem své osy odpovídá 1060 tiků enkodéru. Úhlu natočení v radiánech tedy odpovídá počet dílků vynásobený 0,005927533.

Měření proudu

Ukazuje se že závislost mezi protékaným proudem a měřenou velikostí proudovým senzorem je lineární. Pro výpočet hodnotu pouze posuneme (na nulovou hodnotu - nulový proud je při 2270 dílcích) a určíme měřítko (1A odpovídá 42,8852 dílků na výstupu z proudového čidla).

Požadované napětí

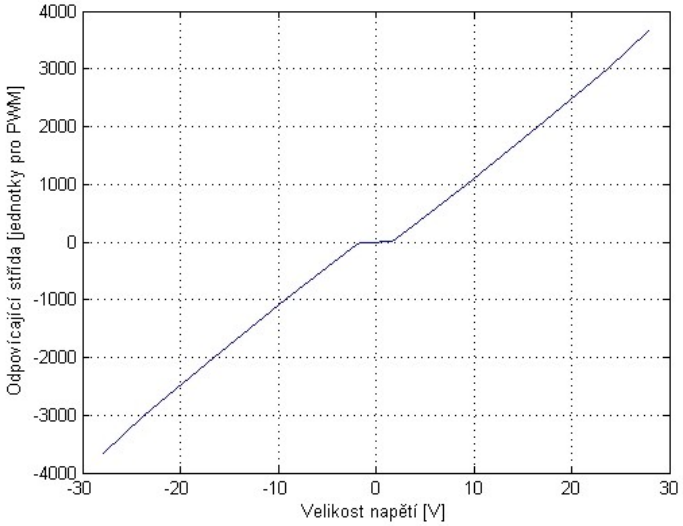
Ukázalo se, že velikost napětí není lineární funkcí střídy předepsanou pro spínání tranzistorů. Z naměřených hodnot sestrojíme tabulku, a tu dále použijeme pro převod požadovaného napětí a střídy PWM. Převodní charakteristika je zobrazena na obrázku 5.3.

5.3 Odhad parametrů

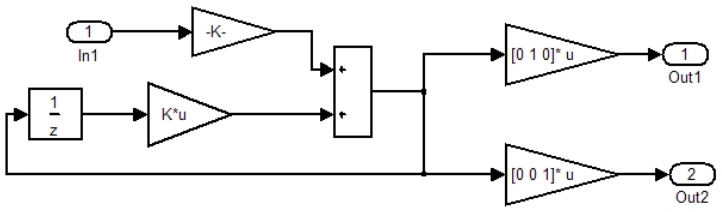
Poté co jsme znormovali a zlinearizovali měřené veličiny přistoupíme k odhadu parametrů. Použijeme k tomu nástroj *Simulinku - Control and Estimation Tools*.

Nejdřív naměříme data pro odhad parametrů. Vstupem jsou skoky napětí na motoru (+12V a -6V) a výstupem je měřený proud a poloha. Tyto hodnoty použijeme pro odhad parametrů. Model upravíme do podoby pro použití s *Control and Estimation Tools*, obrázek 5.4. Pochopitelně při estimaci parametrů je důležité nezapomenout zvolit diskrétní řešič bez spojitých stavů a správně nastavit časový krok aby souhlasil s naměřenými daty.

Při hledání parametrů minimalizujeme rozdíl mezi výstupem matematického modelu a měřenými hodnotami. Průběh naměřených výstupů a predikovaných hod-



Obr. 5.3: Závislost střídy PWM a napětí.



Obr. 5.4: Model pro estimaci parametrů.

not (podle modelu s nově nalezenými konstantami) je na obrázku 5.5. Šedou barvou jsou naměřená data a modlou výstup z matematického modelu.

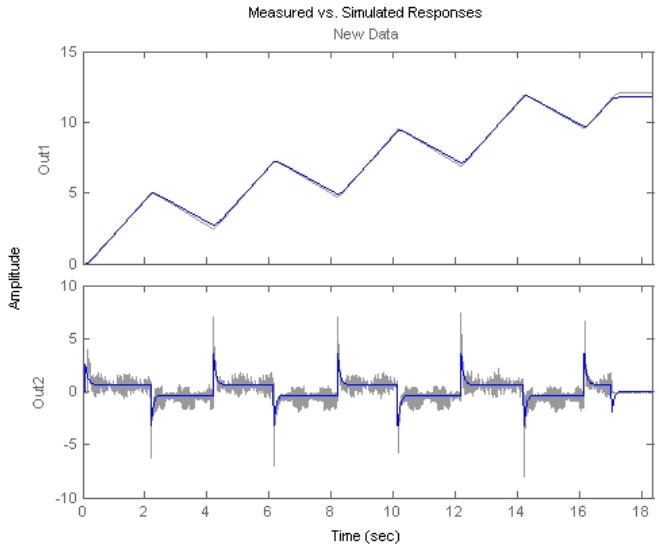
5.4 Filtrování proudu pomocí KF

Jelikož máme k dispozici lineární model popisující chování proudu v motoru, můžeme jej snadno implementovat do algoritmu *Kalmanova filtru*.

5.4.1 Použitý algoritmus KF

Kalmanův Filtr implementujeme pomocí *Embedded Matlab* funkce. Filtrační algoritmus zapsaný v *Matlabu* zabere pouze několik řádků.

```
function [proud, rychlost]  = Kalmanfilt(poloham,proudm,u)
% eml
y=[poloham;proudm];

persistent Pkn;
persistent xkn;

if isempty(Pkn) || isempty(xkn), % inicializace matic
xkn = single([0; 0; 0]);
Pkn = single(eye(3));       end;

% Šum vstupující do modelu
Q = [ 1000 0 0;
      0 100 0;
      0 0 1]*1;

% Šum vstupující do měření
R = [10 0;
      0 100000];

%Stavový model soustavy
a1=0.969;a2=0.111;c1=0.791;d1=0.2136;

A=[a1  0  a2; 0.01  1  0; -c1  0  0];
B=[0;  0; d1];
C=[0  1  0; 0  0  1];

% Predikce stavů a jejich kovariance
%==================================
xk = A*xkn+B*u;
Pk = A*Pkn*A.'+Q;
%==================================

% Úprava predikovanych veličin podle naměřených hodnot
%=====================================================
xkn = xk + (Pk*C.')*((C*Pk*C.'+R)\(y-C*xk));
Pkn = Pk - (Pk*C.')*((C*Pk*C.'+R)\C*Pk);
%=====================================================

proud=xkn(3);rychlost=xkn(1);end;
```



Obr. 5.5: Naměřené hodnoty a výstup modelu.

Volba konstant

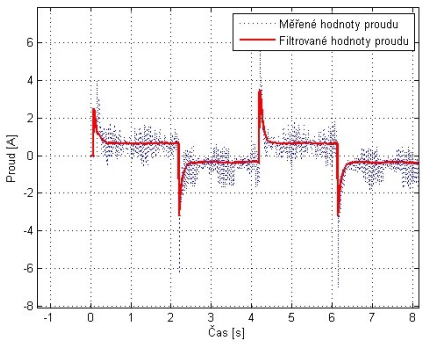
Jak už bylo dříve uvedeno v kapitole 2.2, pro chod algoritmu je klíčové správné nastavení matic Q a R.

Jelikož měřený proud je velmi zarušený, odpovídající prvek matice R bude mít velkou hodnotu (velká variance šumu). Výpočet proudu je velmi citlivý na rychlost otáčení kola. Nastavení matice Q provedeme podle podobné úvahy. Výpočet polohy a proudu je poměrně přesný, nicméně případná porucha způsobí změnu rychlosti, proto nastavíme velkou varianci šumu ve stavu rychlosti (první prvek na hlavní diagonále zleva).

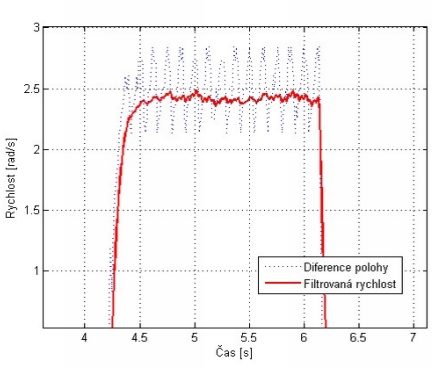
Filtrovaná data jsou rychlost a proud. Informace o rychlosti není použita v momentovém regulátoru, nicméně používá se v jiných algoritmech implementovaných do řídící jednotky vozidla. Výkon algoritmu při filtrování proudu je dokumentován na obrázku 5.6. Pro srovnání byla vypočítaná rychlost pomocí diference polohy (rozdíl dvou po sobě jdoucích hodnoty za dobu výpočetního kroku). Ta byla porovnána s rychlostí, kterou počítá filtrační algoritmus - obrázek 5.7.

5.4.2 Regulace s Filtrovanými hodnotami

V obrázku 5.8 je znázorněn průběh proudů při požadovaném obdélníkovém průběhu proudu. Oproti přímo měřenému proudu 5.2 je průběh filtrovaného proudu nesrovnatelně méně zašumělý, díky tomu je možné zvolit v PI regulátoru větší zesílení

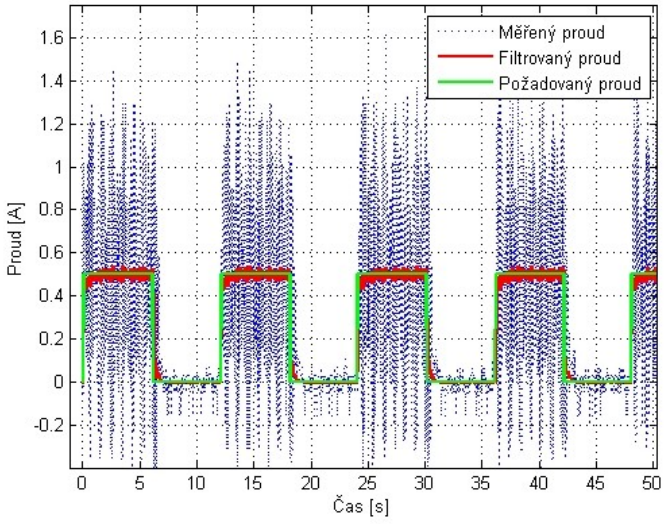


Obr. 5.6: Filtrování proudu



Obr. 5.7: Výpočet rychlosti pomocí KF

jednotlivých složek aniž by se soustava rozkmitala a tím docílit rychlejšího dosažení požadované hodnoty.



Obr. 5.8: Průběh požadované a filtrované hodnoty proudu.

Výpočetní zátěž procesoru

Vzhledem k typu úlohy není prakticky možné jednoduše převést operace z plovoucí aritmetiky do celočíselné (maticové násobení by se muselo provádět pomocí cyklů a

Použitý typ procesoru	doba výpočtu
MPC555	0,0003048s
dsPIC	0,003172s

Tab. 5.1: Doba výpočtu jednoho cyklu filtrace dat.

každý stav v závislosti na jeho rozsahu uchovávat v proměnné odpovídajícího typu. V tabulce 5.1 je doba výpočtu jednoho cyklu filtračního algoritmu.

Naměřený výsledek je poměrně překvapivý, pokud totiž porovnáváme pouze rychlosti výpočtu s jednotlivými operacemi výsledné časy jsou srovnatelné. (Jak je patrné z měření v kapitole 4.5.3.) V takto komplexním výpočtu se projeví i další parametry procesoru, jako například rychlost přístupu do paměti. Výsledná doba výpočtu se pak zásadně změní.

Vzhledem k tomu, že na jednom procesoru dsPIC potřebujeme řídit proudy pro dvě nezávislá kola, je nutné zvolit výpočetní krok délky (0,01s) a nebo použít méně kvalitní informaci o proudu ve zpětné vazbě.

Určení polohy auta na základě měřených veličin.

6 Filtrování dat na reálné soustavě

V praxi je velmi často potřeba určit polohu nějakého vozidla. Bohužel většinou k tomu nelze použít přímé metody měření (radar nebo laserový měřič vzdálenosti). Musíme proto použít signály z čidel integrovaných v soustavě. Dnes vyráběná auta jsou standardně vybavena enkodérem měřícím úhel natočení kol, případně MEMS akcelerometrem[1].

V následující úloze je zpracováván a vyhodnocen signál získaný z měřicích prvků umístěných v experimentálním vozidle *Car4*.

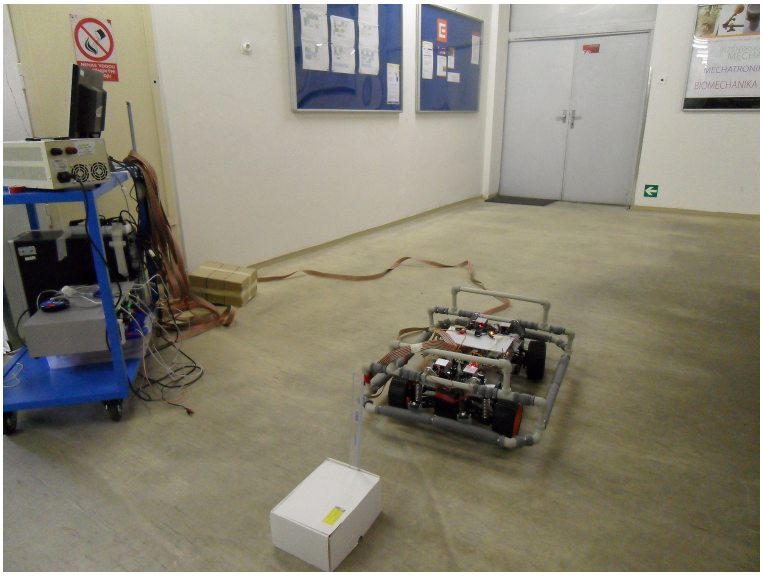
6.1 Popis experimentu

S experimentálním vozidlem byl proveden jízdní manévr a po té se vrátilo do původní polohy. Po zpracování naměřených dat by měla vycházet nulová poloha při zastavení auta v konečné poloze. Kvalita měření potom odpovídá chybě, o kolik se liší konečná poloha od výchozí (čím blíže výchozí poloze, tím lépe). Abychom přesněji zhodnotili kvalitu filtrovaných dat, budeme navíc sledovat chybu odhadnuté a skutečné polohy během jízdy vozidla. Fotka z průběhu experimentu je na obrázku 6.1.

Z obrázku 6.1 je dále patrný charakter experimentální oblasti. Začátek a současně konec jízdní dráhy vymezuje bílá krabice. Na té je umístěno pravítko zjednodušující lokalizaci vztažného bodu.

Jelikož je výpočetní výkon jednotek instalovaných přímo na vozidle příliš malý, jsou data získaná z měření dále zpracovávána na počítači. Přenos dat mezi počítačem a experimentálním vozidlem je prováděn po sériové lince *RS-232*. Toto rozhraní umožňuje přenášet pouze 8 bitové zprávy. Abychom mohli přenést větší množství dat byla vytvořena jednoduchá datová konstrukce „rámec“, která se po přenesení do počítače zpracovávala v prostředí *Matlabu*. Z této aplikace je také vozidlo pomocí klávesnice ovládáno.

[1] Zrychlení se určuje na základě měření účinků setrvačných sil. Celá měřící soustava je integrována v pouzdře o velikosti několika milimetrů [21]



Obr. 6.1: Průběh měření na experimentálním vozidle.

Napájení procesorů a motorů je realizováno ze stabilizovaného zdroje připojeného kabelem. Serva nebylo možné připojit přes takto dlouhý vodič ke zdroji. Vzhledem k charakteru jejich práce dochází k významnému kolísání odebíraného proudu. Jelikož odpor vodiče je nezanedbatelný, dojde na něm při větším odběru proudu k poklesu napětí a serva přestanou pracovat. Navíc konstrukce elektroniky serv neumožňuje jejich napojení na napětí vyšší než 6V. Proto jsou serva napájena zvlášť z baterie umístěné v zadní části vozidla.

6.1.1 Měřená data

Při experimentu byla poháněna pouze přední náprava. Při pohonu všech kol takovým momentem, aby došlo k jejich prokluzu, by byla výsledná rychlost auta po několika sekundách natolik velká, že by nebylo možné v daných podmínkách zaručit, že nedojde k poškození auta, pokud nastane neočekávaná situace.

Měřenými veličinami je úhel otočení kol přední nápravy, proudy protékající motory a velikost zrychlení z akcelerometru. Jelikož v tomto experimentu uvažujeme pouze pohyb v jedné ose, úloha se navíc zjednoduší tak že uvažujeme pouze jedno kolo (průměr natočení kol přední nápravy) a stejným způsobem určíme i proud protékající motorem. Z akcelerometru měříme zrychlení v podélné ose auta.

Jelikož zadní kola nejsou poháněna, pouze se volně otáčejí, nedojde k jejich prokluzu. Hodnotu z enkodérů zadních kol tedy používáme k určení skutečné polohy auta.

6.2 Matematický model

Standardně používané modely pneumatiky (jejich studium například v [3]) pro simulování jejího chování nelze použít při filtraci v reálném čase. Tyto algoritmy používají algebraickou smyčku, kterou nelze jednoduše odstranit zavedením zpoždění do části výpočtu. (Třecí síla je funkcí rozdílu rychlostí vozidla a kola a tyto rychlosti jsou funkcí třecí síly.) Pokud bychom takové zpoždění zavedly muselo by být nesmírně malé (což není možné vzhledem k rychlosti vzorkování měřených dat). Při zavedení většího zpoždění začne být výpočet nestabilní – oscilovat.

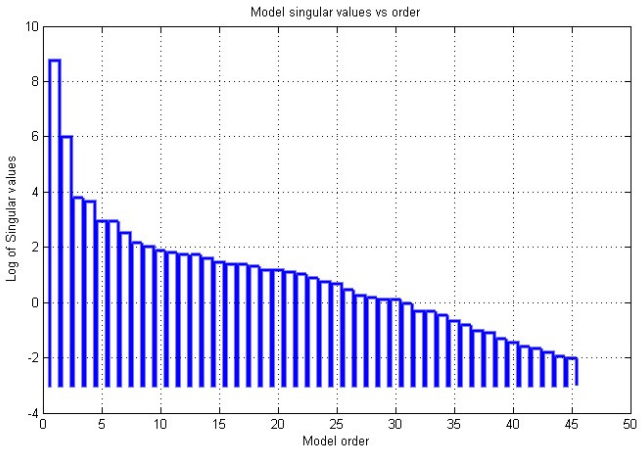
Vzhledem k těmto problémům aproximujeme model lineárním modelem v oblasti ve které je soustava provozována.

6.2.1 Identifikace systému

Jelikož chceme polohu dat filtrovat pomocí algoritmu KF, je vhodné připravit stavový model popisující chování daného systému.

Podle referenčních dat odhadneme řád modelu. Model vyššího řádu aproximuje daný systém s vyšší přesností, ale pokud zvolíme příliš mnoho parametrů (stavů), víc než systém skutečně má, dojde k modelování šumu.

Podle obrázku 6.2, kde je zobrazena závislost velikosti „chyby“ na počtu stavů odhadovaného systému, zvolíme vhodný počet stavů. Ideální počet se zdá být čtyři stavy. Při zvyšování počtu stavů už nedochází k příliš významnému snižování chyby.



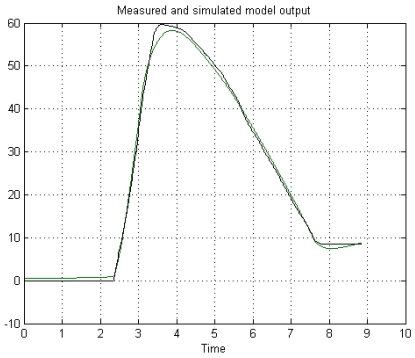
Obr. 6.2: Velikost chyby v závislosti na řádu systému.

K identifikaci systému použijeme nástroj integrovaný v *Matlabu* – `System Identification Toolbox`. Pomocí metody iterativního hledání (minimalizace

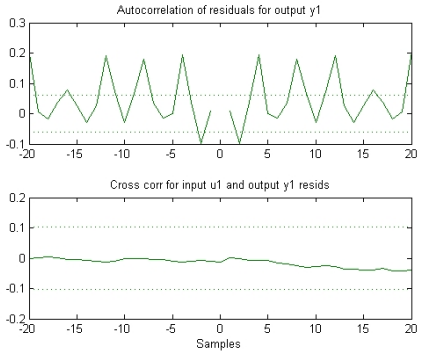
chyby predikce modelu) najdeme koeficienty matic *LTI*[2] modelu s jedním vstupem a více výstupy. Označování matic je obvyklým způsobem (zobrazeno níže), navíc *Matlab* spočítá i matici odpovídající zesílení šumu vstupujícího do systému – K. Tu nebudeme v dalších výpočtech potřebovat.

```
x(t+Ts) = A x(t) + B u(t) + K e(t)
y(t)    = C x(t) + D u(t) + e(t)
```

Zkontrolujeme, jaká je chyba predikce modelu 6.3, a jestli není šum na vstupu a výstupu výrazněji závislý 6.4. Z předchozích grafů není patrný žádný problém, model tedy nebudeme dále měnit.



Obr. 6.3: Predikce výstupu a naměřené hodnoty



Obr. 6.4: Závislost šumu na vstupu a výstupu modelu

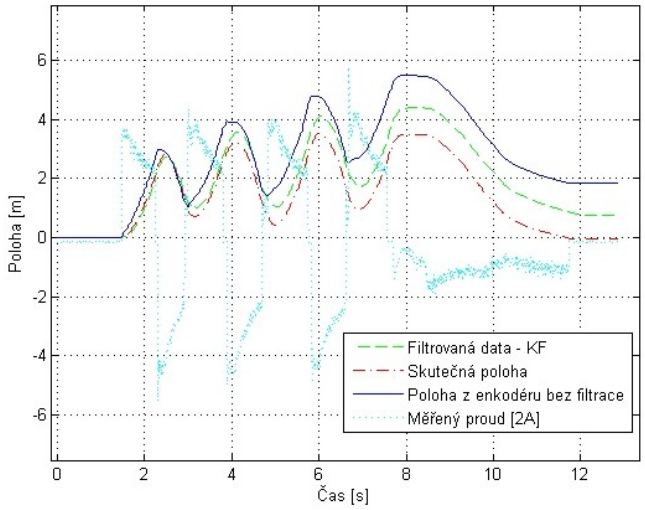
6.3 Odhad polohy z měřených dat

S autíčkem byl proveden jízdní manévr jízdy dopředu a dozadu několikrát za sebou. Přitom při jízdě dopředu docházelo k prokluzu kol.

6.3.1 Určení polohy z natočení kola

V tomto případě určíme polohu pouze na základě znalosti natočení prokluzujícího kola. Pro srovnání je odhad polohy filtrován s použitím lineárního modelu získaného identifikací v předchozí části 6.2.1. Naměřené průběhy jsou na obrázku 6.5.

[2]Lineární model s konstantními (časově nezávislými) koeficienty.



Obr. 6.5: Srovnání kvality odhadu polohy vycházející z měření natočení kola.

Nastavení filtrovacího algoritmu

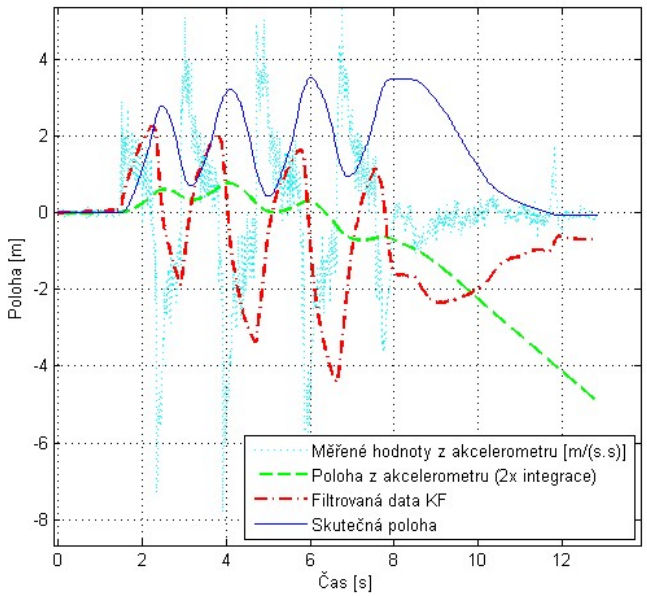
Vstupem do filtračního algoritmu je měřená poloha prokluzujícího kola, měřený proud protékající motorem a známé napětí na motoru (nastavujeme jej pomocí střídy spínání tranzistorů H můstku). Vzhledem k charakteru měřených dat byly koeficienty šumu v maticích nastaveny následovně: Velká důležitost byla přiřazena měření úhlu otočení prokluzujícího kola, a malá důležitost měření proudu (tyto data jsou velmi zatížená šumem). Velikost šumu vstupujícího do matematického modelu (do stavů) byla zvolena tak, aby byla zhruba mezi hodnotami velikosti šumu měření polohy a proudu.

6.3.2 Určení polohy z akcelerometru

V tomto měření byly pro určení polohy použity hodnoty z akcelerometru. Získaná data jsou zobrazena v grafu na obrázku 6.6.

Nastavení Filtru

Jak je patrné z průběhů v grafu 6.6, na akcelerometru se během měření začne posunovat hodnota nulového zrychlení (odhad polohy z integrace utíká směrem do záporných hodnot. Výsledný odhad polohy se nedá příliš zlepšit ani při použití filtračního algoritmu s informací o soustavě. Filtrovací algoritmus byl nastaven tak,



Obr. 6.6: Srovnání kvality odhadu polohy vycházející z měření zrychlení.

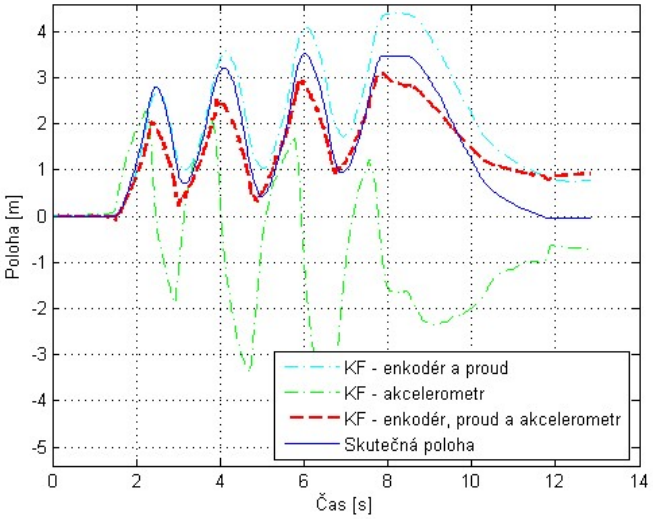
aby se preferovaly hodnoty z matematického modelu (odhad budoucích stavů) oproti měřenému zrychlení.

6.3.3 Určení polohy kombinováním měřených veličin

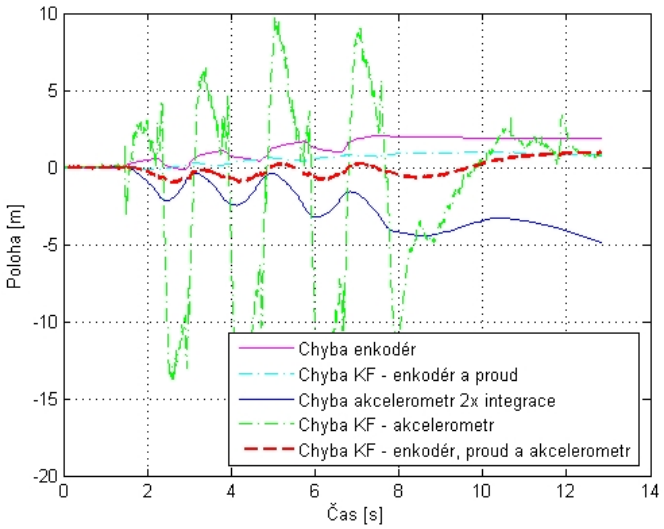
V této úloze byly pro určení aktuální polohy kombinovány data z akcelerometru a enkodéru (otočení prokluzujícího kola). Navíc pro srovnání byly do grafu přidány průběhy veličin filtrovaných pomocí metod představených výše. Nastavení kovariančních matic pro algoritmus *KF* bylo zvoleno podobně jako v předchozích úlohách. Největší důraz je kladen na měření otočení prokluzujícího kola, menší na matematický model, poté data z akcelerometru a nejmenší váha je přiřazena měření proudu. Průběh filtrované polohy je zobrazen na obrázku 6.7.

6.4 Zhodnocení kvality odhadu polohy

V grafu na obrázku 6.8 je zobrazen průběh chyby odhadované veličiny v závislosti na čase a použitém typu vstupů do KF.



Obr. 6.7: Kvalita odhadu polohy při použití různých měřených veličin.



Obr. 6.8: Chyba odhadu polohy při použití různých měřených veličin.

Trochu překvapivý je fakt, že chyba při určování polohy pomocí dvojité integrace akcelerometru je menší, než chyba při filtraci pomocí KF s využitím informace o zrychlení. Tento problém je pravděpodobně způsoben nekvalitním matematickým modelem. Při identifikaci systému se nepodařilo najít správnou závislost mezi vstupním napětím, polohou auta a zrychlením. Proto při filtrování s použitím tohoto modelu je výsledná chyba poměrně velká.

Abychom mohli srovnat, jak kvalitně je daná poloha určená, zavedeme chybu odhadu polohy jako průměrnou hodnotu druhé mocniny rozdílu určené a skutečné polohy. Takto určená chyba je zobrazena v tabulce 6.1.

Použitá metoda	Chyba
Měření otočení kola a proudu	$2{,}1034m^2$
KF - otočení kola a proud	$0{,}4491m^2$
Měření zrychlení (dvojitá integrace)	$8{,}0704m^2$
KF - zrychlení z akcelerometru	$34{,}4015m^2$
KF - otočení kola, proud a zrychlení	$0{,}293m^2$

Tab. 6.1: Druhá mocnina průměrné chyby při určování polohy různými způsoby

Z naměřených údajů vyplývá, že čím máme k dispozici více informací o stavech soustavy (měřených stavů), tím kvalitnější odhad skutečného stavu získáme. Navíc, pokud zahrneme do výpočtu i poměrně nekvalitní informaci, které přiřadíme odpovídajícím způsobem důležitost, dojde k zpřesnění odhadované veličiny.

Shrnutí problematiky, která byla zpracována v této práci.

7 Závěr

Poměrně hodně úsilí bylo věnováno studiu nástroje *Matlab Embedded Target*. S tím souvisí analýza možností a nástrojů, které nabízí *Matlab* pro simulaci výpočtů s desetinnými čísly pomocí celočíselné aritmetiky. Ukázalo se, že výpočet některých funkcí (sinus), které potřebujeme počítat na procesoru *dsPIC*, trvá neúměrně dlouho. Navržená alternativa je aproximace této funkce pomocí tabulky na definičním oboru, který používáme. Doba výpočtu touto alternativou je oproti původní funkci volané z knihovny *Microchip* přibližně sedmkrát rychlejší.

Dalším tématem bylo studium filtračních algoritmů využívajících znalosti matematického modelu popisujícího soustavu. Tyto filtry vychází z „obyčejného“ Kalmanova filtru a nějakým způsobem jej rozšiřují pro použití s nelineárním stavovým modelem. Ukazuje se, že kvalitu filtrovaného signálu příliš neovlivní použitý typ Kalmanova filtru (*UKF* nebo *EKF*). Při výběru konkrétní modifikace *KF* se vychází spíše z implementační náročnosti, než z rozdílu v kvalitě vyfiltrovaného signálu.

Pro mikrokontroléry *dsPIC*, které byly použity v experimentálním vozidle byl navržen proudový regulátor. Nicméně vzhledem k velikosti šumu přítomném v měřeném proudu bylo nutné signál před použitím ve zpětné vazbě regulátoru zpracovat. To bylo provedeno pomocí KF. I když celý algoritmus nebyl převeden do celočíselné aritmetiky, výpočet probíhá dostatečně rychle, aby stíhal pracovat v reálném čase. Zavedení tohoto filtru umožňuje zlepšit vlastnosti regulátoru (zkrátit dobu odezvy na řídící signál).

Díky experimentálnímu vozidlu, které bylo postaveno ve školní laboratoři bylo možné studovat chování algoritmů při použití s reálnou soustavou. Byl sestaven algoritmus, který se pomocí měřených dat snaží co nejpřesněji odhadnout skutečnou polohu vozidla. Ukazuje se, že čím více máme k dispozici měřených veličin (i relativně málo kvalitní signál) tím kvalitnější odhad skutečného stavu můžeme získat. Tento závěr je ve shodě s tím, co bychom předpokládali.

8

Literatura a odkazy

[1] ŠIMURDA, M.: *Návrh a realizace doplňkového senzorického systému pro experimentální vozidlo.*, Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, 35 s., 2010

[2] VEJLUPEK, J.: *ývoj elektroniky pro řízení trakce experimentálního vozidla.*, Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství., 2010

[3] JASANSKÝ, M.: *Návrh dynamických modelů pro řízení trakce experimentálního vozidla.*, Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, 107 s., 2010.

[4] VADLEJCH, F.: *Design of experimental vehicle undercarriage with four wheel steering.*, Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, 56 p., 2010.

[5] Steven W. Smith: *The scientist and engineer's guide to digital signal processing*, California Technical Publishing, San Diego, CA„ 1997

[6] Karpíšek, Z.: *Matematika IV. Statistika a pravděpodobnost*, Brno : FSI VUT v CERM, 2003

[7] Lennart Ljung and Torkel Glad: *Modeling of Dynamic Systems*, rentice Hall, Inc., Englewood Cliffs, NJ, 1994

[8] Aström K.J., Wittenmark B.: *Computer Controlled Systems - Theory and Design*, 3rd edition, Prentice-Hall, Englewood Cliffs, N.J. ..., 1997

[9] L. Čermák, R. Hlavička: *Numerické metody*, CERM, Brno, 2006

[10] R. E. Kalman: *A New Approach to Linear Filtering and Prediction Problems*, Transactions of the ASME–Journal of Basic Engineering, 82 (Series D): 35-45, 1960

[11] H. Hyoung, K. Kiwan Choi Seok, W. Yong, B. Lee Sang, R. Kim: *Constrained Kalman Filter for Mobile Robot Localization with Gyroscope*, Proceeding of Intelligent Conference on Robots and Systems, IEEE/RSJ, 2007

[12] Deng Kun, Li Kaijun, and Xia Qunsheng: *Application of Unscented Kalman Filter for the State Estimation of Anti-lock Braking System*, IEEE 1-4244-0759-1, pp. 130–133, 2006

[13] Sun Zhen-jun Zhu Tian-jun and Zheng Hong-yan: *Research on Road Friction Coefficient Estimation algorithm Based on Extended Kalman Filter*, IEEE 978-0-7695-3357-5/08, pp. 418–422, 2008

[14] Pacejka, H.B.: *Tyre and Vehicle Dynamics*, Butterworth-Heinemann, ISBN 0-7506-5141-5, Third improved imprint, 2005

[15] Wikipedia Zero-order hold:, `http://en.wikipedia.org/wiki/Zero-order_hold`, 1010-05

[16] MPLAB:, `http://www.microchip.com/MPLAB`, 1010-05

[17] Kerhuel website:, `http://www.kerhuel.eu/wiki/Download#Requirement`, 1010-05

[18] Mechlab:, `http://www.umt.fme.vutbr.cz/mechlab`, 1010-05

[19] Pavel Herout: *Učebnice jazyka C*, Kopp, ISBN 80-7232-220-6, 1995

[20] Press, William H.; Flannery, Brian P.; Teukolsky, Saul A.; Vetterling, William T. : *LU Decomposition and Its Applications*, Cambridge University Press, pp. 34–42, 1992

[21] Ing. Pavel Houška, Ph.D.: *Senzorika a prvky umělé inteligence*, Prezentace - přednáška 6, 2010

Často používané zkratky (symboly) a jejich popis.

9 Použité zkratky a symboly

AD převodník - Převodník, který převádí spojitý (analogový) signál na digitální
bit - Jednotka, která může nabývat dvou hodnot (0 nebo 1)
byte - Jednotka, která může nabývat 2^8 hodnot (skupina 8 bitů)
C30 - Překladač jazyka C do strojového kódu (Microchip)
dsPIC - Označení procesoru - použitý typ p33FJ128MC804
EKF - Rozšířený Kalmanův filtr
fix - Celočíselný datový typ Matlabu simulující desetinné číslo
IIR, FIR - Nekonečná odezva na impulz, resp. konečná odezva na impulz
int16 int32 - Celočíselný datový typ (16 respektive 32 bitů)
KF - Kalmanův filtr
MPC555 - Označení procesoru - použitý typ 40Mhz
single, double - Proměnná s plovoucí desetinnou čárkou (16 resp 32 bitů)
uint16 uint32 - Stejně jako int16 (int32) ale pouze pro kladná čísla
UKF - Unscentovaný Kalmanův filtr
ZOH - Metoda diskretizace (konstanta během periody výpočtu)