České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra řídicí techniky

Bakalářská práce

# Automatické generování kódu palubních regulátorů pro kolonu autodráhových autíček

***Michal Staněk***

Vedoucí práce: doc. Ing. Zdeněk Hurák, Ph.D.

25. května 2017

České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra řídicí techniky

# ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: **Staněk Michal**

Studijní program: Kybernetika a robotika
Obor: Systémy a řízení

Název tématu: **Automatické generování kódu palubních regulátorů pro kolonu autodráhových autíček**

Pokyny pro vypracování:

Cílem práce je umožnit využití kódu v jazyce C automaticky vygenerovaného Simulinkem pro realizaci regulátorů na palubě autodráhových autíček tvořících distribuovaně řízenou kolonu. Jelikož je palubní software realizován v jazyce Java, bude samotná práce spočívat v napsání efektivních "wrapperů" obalujících vygenerovaný kód v jazyce C. Konkrétní pokyny:

1. Napište wrapper v jazyce Java tak, aby uživatel mohl přímo v Matlabu/Simulinku připravit vygenerovaný C-kód regulátoru pro nahrání do palubního počítače autíčka (RaspberryPi).
2. Wrapperů nachystejte více pro různé množství vstupů: dvojsměrný regulátor, regulátor s přímou vazbou od předchůdce, regulátor s přímou vazbou od vedoucího vozu. Zvažte i implementaci wrapperu pro regulátor, který bude mít proměnné množství vstupů.
3. Připravte export stavových regulátorů z Matlabu.
4. Implementujte možnost nastavování parametrů regulátorů přímo z Matlabu přes konfigurační XML soubor.
5. Funkčnost navrženého řešení demonstrujte na příkladu několika typů regulátorů, a to včetně regulátorů s více vstupy a/nebo více výstupy a regulátorů s nelinearitami.

Seznam odborné literatury:

[1] Repositář projektu na https://gitlab.fel.cvut.cz/SlotcarPlatooning.

Vedoucí: doc. Ing. Zdeněk Hurák, Ph.D.

Platnost zadání: do konce letního semestru 2017/2018

L.S.

prof. Ing. Michael Šebek, DrSc.
vedoucíkatedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 21. 2. 2017

# Poděkování

Chtěl bych poděkovat panu doc. Ing. Zdeňku Hurákovi, Ph.D. za odborné vedení mé práce, za pomoc a věcné rady při zpracování této práce.

# Prohlášení

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

V Praze dne 25. května 2017 .....

České vysoké učení technické v Praze
Fakulta elektrotechnická


### Odkaz na tuto práci

Staněk, Michal. *Automatické generování kódu palubních regulátorů pro kolonu autodráhových autíček*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta elektrotechnická, 2017.

# Abstrakt

Práce se zabývá vytvořením Java wrapperů pro simulinkové regulátory kvůli usnadnění návrhu regulátorů. C kód generovaný ze Simulinku pomocí pluginu Embedded Coder je obalen JNI rozhraním a implementován do hlavního programu jako sdílená knihovna. Dále se práce zabývá nahráním těchto vytvořených souborů do autodráhového vozidla, ve kterém je výpočetní modul Raspberry Pi s operačním systémem Linux.

**Klíčová slova** Java, JNI, Simulink, Raspberry Pi, regulátor, wrapper

# Abstract

The purpose of this thesis is to create Java wrappers for Simulink controllers to ease the process of controllers design. C code generated from Simulink with Embedded Coder is wrapped with JNI interface and implemented into the main program as a shared library. The thesis also looks at the problem of transfering these files to a slotcar running on a Raspberry Pi Compute Module with operating system Linux.

**Keywords** Java, JNI, Simulink, Raspberry Pi, controller, wrapper

# Obsah

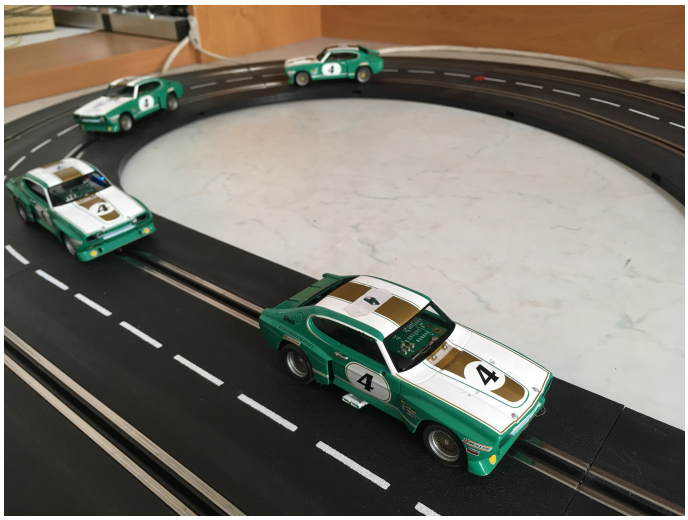
Kapitola 1

# Úvod a motivace

Má bakalářská práce byla vedena jako součást projektu Slotcar platooning, který má za cíl vytvořil testovací kolonu autodráhových vozidel, na které by bylo možné zkoušet různé typy a metody distribuovaného řízení. Projekt vznikl na katedře řídicí techniky ČVUT v Praze pod vedením pana doc. Ing. Zdeňka Huráka Ph.D.

## 1.1 Popis systému

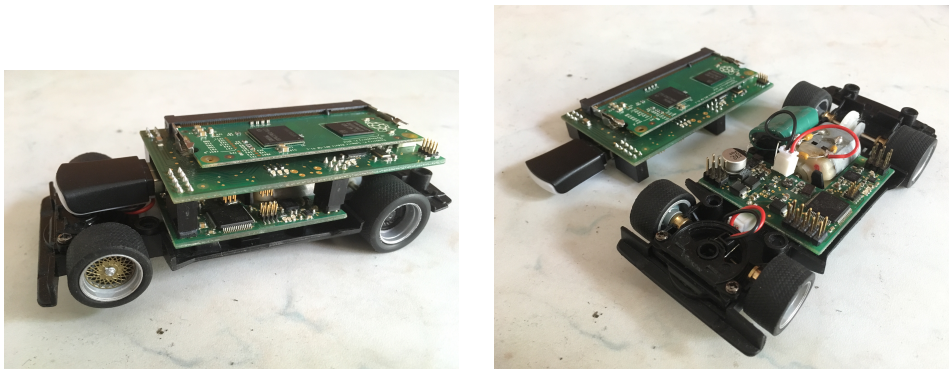
Platforma se skládá z autodráhových vozidel typu Carrera Ford Capri (Obr.1.1).



Obrázek 1.1: Kolona vozidel

Auta jezdí po autodráze vsazené v liště, která jim umožňuje pouze pohyb vpřed nebo vzad, a pomocí této lišty jsou i napájena (v současné době použí-

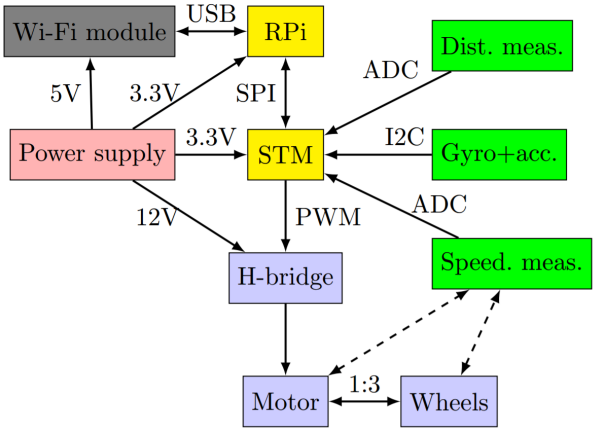
váme napětí 11V). O pohon se stará stejnosměrný elektromotor, který ovládá zadní nápravu kol. Vnitřek aut byl pro potřeby projektu upraven (Obr.1.2). Každé autíčko obsahuje dva výpočetní moduly – ARM Cortex M4 STM32 mikroprocesor a Raspberry Pi Compute Module.



(a) Pohled po odmontování kapoty (b) Spodní STM modul a DC motor

Obrázek 1.2: Vnitřek vozidel

První zmiňovaný (STM) má na starost zpracovávání dat ze senzorů a ovládání DC motoru pomocí PWM (puls-width modulation) signálu. Výpočetní modul Raspberry Pi, na kterém běží operační systém Linux, má za úkol komunikaci s počítačem a ostatními auty v koloně. Komunikace vozidla s okolím je zajišťována pomocí WI-FI modulu (standard IEEE 802.11g), který je připojen k Raspberry Pi pomocí USB. Každé auto obsahuje senzory, které zajišťují sběr informací z okolí. Mezi měřené veličiny patří rychlost vozu, aktuální zrychlení a vzdálenost od sousedních účastníků kolony (před a za vozidlem). Na Obr.1.3 je schématicky znázorněna vnitřní struktura upraveného vozu platformy.




Obrázek 1.3: Vnitřní struktur upraveného vozu[1]

Kolona se ovládá přes hlavní program, který běží na počítači a je psán v jazyce Java. Pomocí tohoto programu lze mimo jiné nastavovat požadované parametry řízení nebo spouštět a zastavovat kolonu. [1][2][3]

## 1.2 Tým pracující na projektu

Práce na projektu probíhají již delší dobu, takže se na jeho tvorbě už podílelo více studentů nebo zaměstnanců školy. Platforma, tak jak jsem jí výše popsal, již tedy byla při mém začlenění do týmu vytvořena. V současnosti na projektu spolu se mnou pracují dva studenti – Šimon Wernish a Bc. Martin Lád. Šimon pracuje na projektu stejně jako já v rámci své bakalářské práce. Zabývá se zlepšením bezdrátové komunikace. Martin na projektu pracuje již delší dobu. Navrhnul například řídící systém vozidel a podílel se na psaní hlavního programu, který stále vylepšuje.

## 1.3 Motivace

Když se s projektem začínalo, rozhodli se zakládající členové o implementaci hlavního programu v Javě. V současné době je budoucí vize taková, že bychom mohli platformu poskytovat pro výukové účely školám, popřípadě firmám, které by projevily zájem. Pokud by měl projekt takto fungovat, bylo by dobré poskytnout uživatelům více možností při návrhu regulátorů. Psaní regulátorů v Javě vyžaduje znalost základů programování a vzhledem ke struktuře projektu i zdlouhavé procházení dokumentací. Mým úkolem tedy bylo umožnit návrh regulátorů v Simulinku a začlenit tyto regulátory do hlavního programu.

## 1.4 Řešení problému

Jako první krok řešení jsem musel převést simulinkové schéma regulátoru do jazyka C. K tomu jsem použil plugin Embedded Coder[4], který je obsažen v matlabovské licenci katedry řídící techniky. Takto připravené soubory bylo nutné propojit s hlavním programem, který očekává regulátory napsané v Javě. Implementování C kódu do Javy lze dosáhnout pomocí JNI rozhraní (Java Native Interface)[5]. Vytvořil jsem tedy pomocné soubory tohoto rozhraní, které jsem spolu s vygenerovaným regulátorem zabalil do knihovny. Dále jsem vytvořil Java wrapper, který knihovnu načítá a volá metody generovaného regulátoru. Tento wrapper se sám tváří jako regulátor a je snadné ho načíst v grafickém rozhraní. Přenesení souborů do řízených aut jsem vyřešil pomocí skriptu. Vytvořené wrappery jsem otestoval na třech základních regulátorech. Nakonec jsem do projektu přidal možnost řízení vozidel pomocí stavového regulátoru, který jsem také otestoval.

Kapitola 2

# Návrh a implementace wrapperů

Tuto kapitolu jsem rozdělil do čtyř hlavních bodů – propojení simulinkového schéma s hlavním programem v Javě, vytvoření skriptu pro přenos souborů, implementace stavových regulátorů a nastavování parametrů regulátoru přes XML soubor.

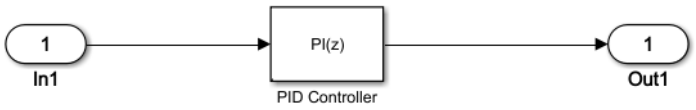
## 2.1 Propojení Simulinku s Javou

Pro propojení simulinkových regulátorů s Javou jsem použil Embedded Coder a Java Native Interface. Embedded Coder je matlabovský plugin, který dokáže zkompilovat simulinkové schéma do jazyka C. Takto generované soubory jsem spolu s pomocnými soubory rozhraní JNI zabalil do sdílené knihovny a tu načetl do obalovací třídy napsané v Javě. Wrappery jsem připravil tři. Pro jednovstupový regulátor, dvouvstupový regulátor a pro regulátor s proměnným počtem vstupů.

### 2.1.1 Jednovstupový regulátor

#### Generování kódu ze Simulinku

V Simulinku jsem vytvořil nové schéma, v kterém jsem nakreslil velmi jednoduchý PI regulátor (Obr.2.1) s jedním vstupem (In1) a jedním výstupem (Out1).




Obrázek 2.1: Jednovstupový PI regulátor

Konstanty jsem nastavil podle již dříve vytvořeného PI regulátoru,

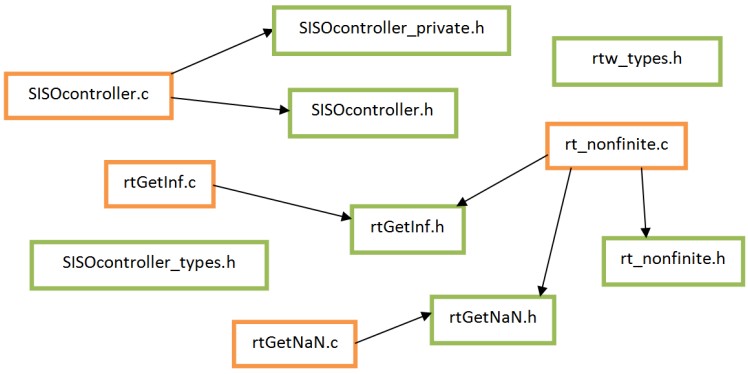
$$\begin{aligned} k_p &= 3 \\ k_i &= 0.8 \end{aligned} \tag{2.1}$$

kde $k_p$ značí proporcionální konstantu a $k_i$ konstantu integrační. Dále jsem regulátoru přidal saturační limit ($\pm 1.5$m/s) a jako anti-windup metodu jsem zvolil clamping. Schéma jsem uložil jako `SISOcontroller` a pomocí pluginu Embedded Coder jsem ho převedl na C kód. Plugin ve složce se schématem vytvoří plně spustitelný projekt, který obsahuje soubory implementující simulinkové bloky do jazyka C, hlavní smyčku programu a jiné pomocné soubory. Na Obr. 2.2 je vidět, jak vypadá vygenerovaná metoda z takového simulinkového bloku.

```
void SISOcontroller_step(void) {

  SISOcontroller_Y.Out1 = 3.0 * SISOcontroller_U.In1
     + SISOcontroller_DW.Integrator_DSTATE;

  SISOcontroller_DW.Integrator_DSTATE +=
     0.8 * SISOcontroller_U.In1;
  }
```

Obrázek 2.2: Implementace bloku PI regulátor v jazyce C

Metoda `SISOcontroller_step()` počítá výstup regulátoru za jednu vzorkovací periodu. Pro moje účely jsou potřeba pouze definující soubory, protože hlavní smyčka programu je již vytvořena v Javě. Jedná se o jeden soubor, který implementuje simulinkové bloky a další pomocné soubory (Obr.2.3).




Obrázek 2.3: Generované soubory potřebné pro wrapper

### Obalovací kód v Javě

Součástí Javy je rozhraní JNI (Java Native Interface), pomocí kterého lze do projektu implementovat metody napsané v jazyce C. V našem projektu jsem vytvořil obalovací třídu, kterou jsem nazval `SimulinkControllerSISO`. Tato třída, stejně jako jakýkoliv jiný již napsaný regulátor, rozšiřuje abstraktní třídu Controller, která v našem projektu zaštiťuje všechny napsané regulátory. Navíc obsahuje deklaraci nativní metody `generatedStep()` a načtení sdílené knihovny `libSISOController.so` (Obr.2.4).

```
private native float generatedStep(float e);
static {
  try{
    System.load("/slotcar/lib/libSISOController.so");
  }
  catch(UnsatisfiedLinkError e){
    System.out.println("Cannot load library.");
  }
}
```

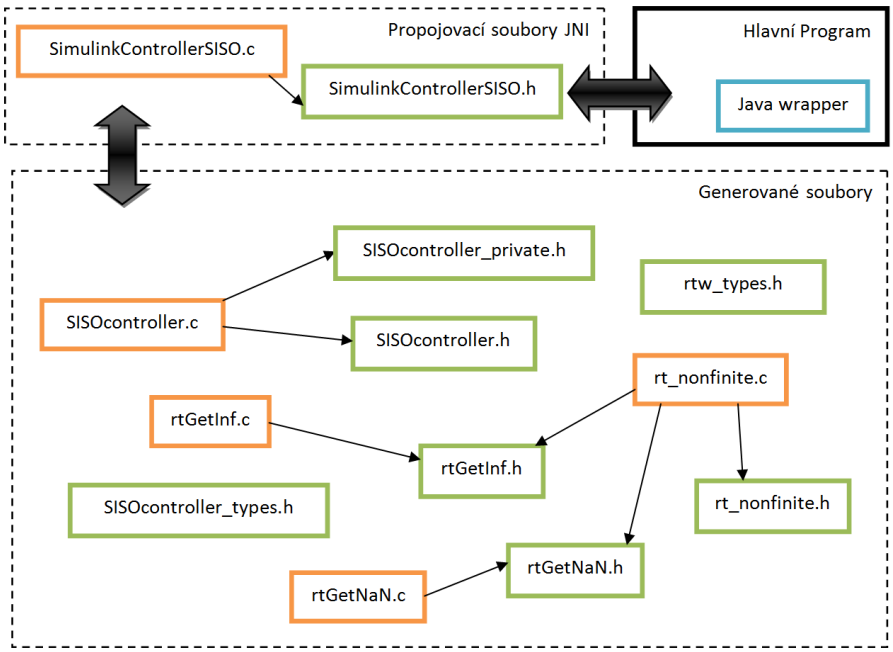
Obrázek 2.4: Deklarace nativní metody a načtení sdílené knihovny

Metoda očekává na vstupu regulační odchylku $e$. Specifikátor `native` říká, že metoda není implementována zde, nýbrž v nahrávané knihovně, která je uložena ve složce `/slotcar/lib` v řízeném autíčku. Tato knihovna obsahuje kromě již vygenerovaných souborů ze Simulinku jeden hlavičkový a jeden .c soubor, které tvoří tzv. propojovací můstek mezi programem v jazyce C a Javou. Soubor s implementovanou nativní metodou `generatedStep()` lze vidět na Obr. 2.5.

```
#include <jni.h>
#include "SimulinkControllerSISO.h"
#include "SISOcontroller.h"

JNIEXPORT float JNICALL Java_slotcar_pi_controller_
SimulinkControllerSISO_generatedStep
  (JNIEnv *env, jobject obj, jfloat e) {
     SISOcontroller_U.In1 = e;
     SISOcontroller_step();
     return SISOcontroller_Y.Out1;}
```

Obrázek 2.5: Implementace metody generatedStep() v propojovacím souboru

Hlavička implementované metody podléhá pevně dané sytnaxi nativního rozhraní. JNIEXPORT a JNICALL jsou makra, která říkají, že metoda je generována a volána pomocí Java Native Interface a jméno metody odkazuje na deklarovanou nativní metodu uloženou ve volající Java třídě, nacházející se v určitém balíku. První argument metody je pointer na JNI rozhraní a druhý odkazuje na objekt volající třídy, v podstatě sám na sebe.[7] Třetí vstupní argument *e* je regulační odchylka. Metoda tedy předá regulační odchylku proměnné `SISOcontroller_U.In1`, což je vstupní blok simulinkového schématu (Obr.2.1), zavolá generovanou metodu, kterou jsem popsal v předchozí sekci (Obr.2.2) a vrací výstup regulátoru. Takto vytvořené propojovací soubory jsem spolu se soubory simulinkového regulátoru přeložil v autíčku do sdílené knihovny, kterou jsem nazval `libSISOController.so`. Kompletní výčet souborů, které obsahuje sdílená knihovna je tedy vidět na relačním diagramu (Obr.2.6).

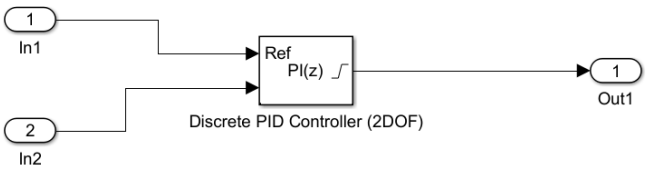



Obrázek 2.6: Relační schéma vytvořených souborů

### 2.1.2 Dvouvstupový regulátor

#### Generování kódu ze Simulinku

V Simulinku jsem podobně jako u jednovstupového regulátoru vytvořil nové schéma pro regulátor dvouvstupový se vstupy In1 a In2 a výstupem Out1 (Obr.2.7).



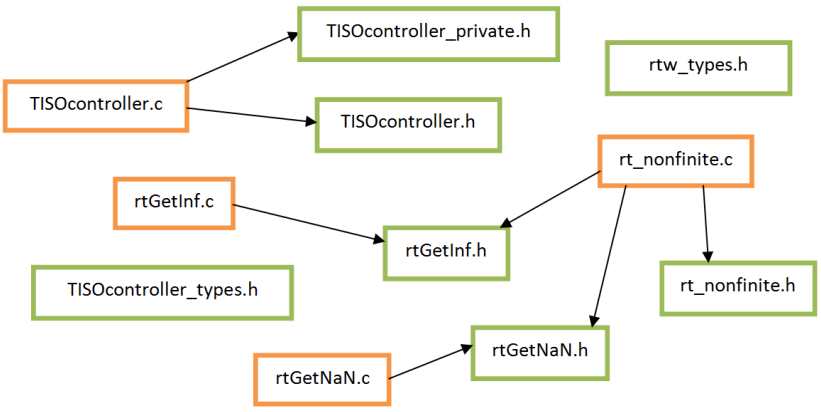

Obrázek 2.7: Dvouvstupový PI regulátor

Pro implementaci jsem použil simulinkové schéma pro diskrétní dvouvstupový PID regulátor. Saturaci i konstanty (proporcionální a integrační) jsem nastavil stejně, jako u předchozího případu (Saturace ±1.5m/s a konstanty podle rovnice 2.1). Schéma jsem uložil jako `TISOController` (TISO zkráceně pro two-input single-output) a pomocí pluginu Embedded Coder jsem schéma převedl do projektu v jazyce C.Implementace metody, která ze vstupů počítá výstup za jednu vzorkovací periodu je velmi podobná jednovstupovému regulátoru (Obr.2.2) a lze si jí prohlédnout na Obr.2.8.

```
void TISOcontroller_step(void) {

  TISOcontroller_Y.Out1 = (TISOcontroller_U.In1
     - TISOcontroller_U.In2) * 3.0
     + TISOcontroller_DW.Integrator_DSTATE;

  TISOcontroller_DW.Integrator_DSTATE +=
      0.8 * TISOcontroller_U.In1;
  }
```

Obrázek 2.8: Implementace bloku PI regulátor v jazyce C

Název metody je dán názvem schématu. V metodě je do proměnné Out1 uložen rozdíl vstupů In1 a In2 přenásobený proporcionální konstantou regulátoru ($k_p$) a je k němu připočten stav integrátoru. V dalším kroku je tento stav přepočítán pomocí integrační konstanty ($k_i$) a uložen do proměnné. Tato názorná implementace je bez nelinearit, abych se vyhnul vkládání dlouhých pasáží kódu. Kód s ošetřenou saturací je obohacený o sérii podmínek (příkazy if), které kontrolují překročení povolených mezí. Spolu se souborem, ve kterém jsou implementovány simulinkové bloky, potřebuji k propojení s Javou ještě vytvořené pomocné soubory (Obr.2.9).




Obrázek 2.9: Relační schéma vytvořených souborů

### Obalovací kód v Javě

Pro generované soubory zbývá v našem projektu vytvořit obalovací třídu a dále propojovací soubory nativního rozhraní (JNI). V projektu jsem vytvořil wrapper `SimulinkControllerTISO`, který rozšiřuje abstraktní třídu `Controller`. Oproti třídě reprezentující regulátor psaný kompletně v Javě obsahuje navíc kód pro načtení sdílené knihovny `libTISOController.so`, ve které jsou generované soubory ze Simulinku pro dvouvstupový regulátor, a deklaraci nativní metody `generatedStep()`, která očekává na vstupu dvě proměnné (Obr.2.10).

```
private native float generatedStep(float e, float f);

static {
  try{
    System.load("/slotcar/lib/libTISOController.so");
  }
  catch(UnsatisfiedLinkError e){
     System.out.println("Cannot load
        libTISOController.so, file does not exist.");
  }
}
```

Obrázek 2.10: Deklarace nativní metody a načtení sdílené knihovny

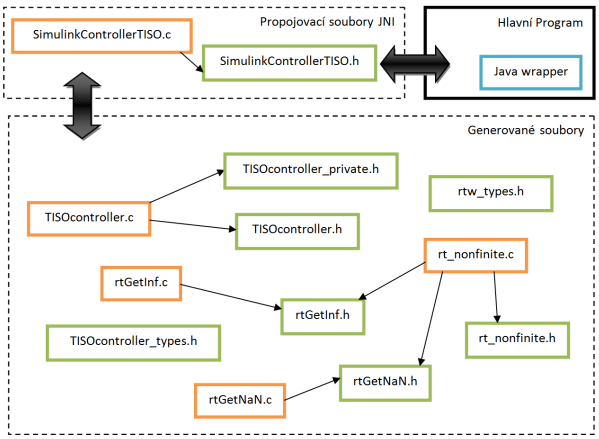
Metoda není implementována zde, ale v načítané knihovně, o čemž nás informuje specifikátor `native`. V této knihovně se kromě vygenerovaných souborů ze Simulinku nachází i propojovací soubory nativního rozhraní. Imple-

mentační soubor .c, v kterém je tělo metody `generatedStep()` zachycuje Obr.2.11.

```
#include <jni.h>
#include "SimulinkControllerTISO.h"
#include "TISOcontroller.h"

JNIEXPORT float JNICALL Java_src_slotcar_pi_controller_
  SimulinkControllerTISO_generatedStep
  (JNIEnv *env, jobject obj, jfloat e, jfloat f) {
     TISOcontroller_U.In1 = e;
     TISOcontroller_U.In2 = f;
     TISOcontroller_step();
     return TISOcontroller_Y.Out1;}
```

Obrázek 2.11: Implementace metody generatedStep() v propojovacím souboru

Soubor obsahuje přidání hlavičkových souborů a samotnou metodu. Hlavičku metody jsem již popisoval u jednovstupového wrapperu (text k Obr.2.5). Zde je navíc vstupní parametr $f$, který reprezentuje druhý vstup regulátoru. Metoda přiřazuje oba vstupy proměnným, které reprezentují vstupní bloky simulinkového schématu (Obr.2.7), volá metodu `TISOcontroller_step()` (Obr. 2.8) a vrací výstup regulátoru. Všechny soubory jazyka C, pomocné JNI i generované ze Simulinku, jsem nahrál do autíčka a přeložil je zde do sdílené knihovny `libTISOController.so`, kterou jsem uložil do složky s knihovnami `/slotcar/lib`. Kompletní výčet souborů v této knihovně je vidět na Obr.2.12.




Obrázek 2.12: Relační schéma vytvořených souborů

### 2.1.3 Regulátor s proměnným počtem vstupů

V rámci mého zadání jsem se měl také zamyslet nad možností implementovat wrapper pro regulátor s proměnným počtem vstupů. Takový wrapper se mi vytvořit povedlo, byl jsem ale limitován možnostmi použitého programovacího jazyku. Má původní představa byla taková, že by obalovací třída v Javě obsahovala deklaraci více nativních metod, které by byly přetížené a volaly by se na základě počtu vstupních parametrů. Propojovací .c soubor nativního rozhraní by rovněž obsahoval implementaci všech přetížených metod, a uvnitř každé z nich by se přiřadil příslušný počet vstupů metody externímu vstupu simulinkového regulátoru. Můj předpoklad počítal s tím, že bych byl schopný na začátku implementované metody dynamicky zjistit, kolik externích vstupů očekává generovaný regulátor. Avšak při pročítání dokumentace k Embedded Coderu a procházení generovaných souborů jsem zjistil, že plugin při vytváření kódu v jazyce C převádí všechny externí vstupy do jedné proměnné typu `struct`. Na obrázku 2.13 je vidět struktura vygenerovaná pro dvouvstupový regulátor.

```
typedef struct {
  real_T In1;
  real_T In2;
} ExtU_TISOcontroller_T;
```

Obrázek 2.13: Vygenerovaná struktura externích vstupů

Pokud by plugin pracoval s proměnnými typu `array`, předpokladu bych dosáhl. Vždy bych zjistil počet prvků v poli a poté každému z nich přiřadil jeden vstupní parametr. Počet prvků ve struktuře jsem teoreticky taky schopný určit. Pokud platí, že struktura obsahuje všechny prvky o stejné velikosti, mohl bych výpočet provést pomocí:

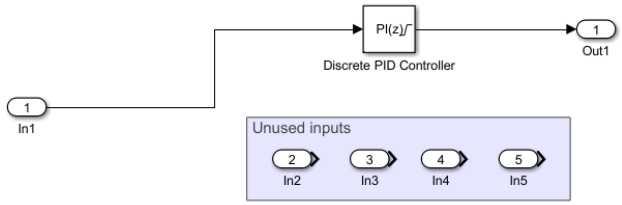
$$struct\_length = sizeof(struct)/sizeof(element)$$

Takto počítaná délka struktury v jazyce C ale není robustní, protože celková velikost struktury se nemusí rovnat součtu velikostí prvků uvnitř. V závislosti na kompilátoru mohou být v paměti na konci každého elementu uloženy další redundantní byty a výsledek tedy nemusí být vždy shodný s reálným počtem prvků ve struktuře. Ani kdybych tento fakt opomenul, nedokázal bych strukturu dynamicky naplnit, protože se k prvkům nedá přistupovat ani pomocí indexace ani pomocí pointerové aritmetiky. Řešení jsem tedy navrhnul tak, že simulinkové schéma regulátoru musí za každých podmínek obsahovat všechny externí vstupy, ikdyž se s nimi zrovna nepracuje. Takto docílím vždy vygenerování struktury se stejným počtem prvků. Nemožnost přistupovat ke generovaným prvkům struktury externích vstupů pomocí indexace má také za

následek to, že názvy externích vstupů ve schématu nesmějí být přejmenovány. Plugin při tvorbě C kódu pojmenovává prvky struktury podle názvů ve schématu, na což by moje propojovací soubory nedokázaly dynamicky reagovat. Ze stejného důvodu se nesmí pojmenovat ani signál, který ze vstupu vychází. V konečném řešení jsem počítal s wrapperem pro regulátor s maximálně pěti možnými vstupními parametry.

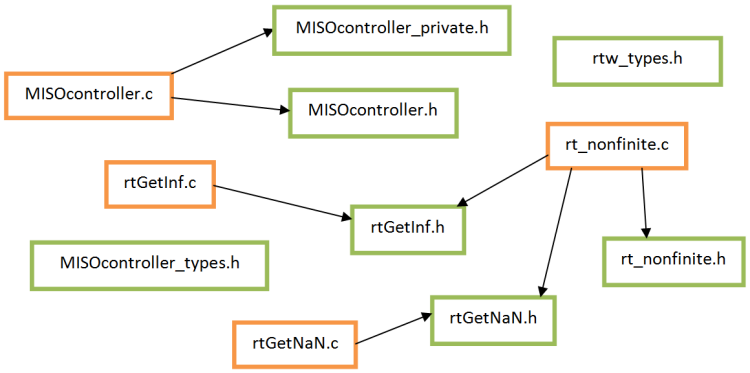
### Generování kódu ze Simulinku

Jak jsem přednesl dříve, aby wrapper mohl správně fungovat, musí být v simulinkovém schématu vždy všech pět externích vstupů. Vstup (In1) jsem spojil s blokem `PID Controller` a přivedl na výstup (Out1), čímž mi vznikl stejný jednovstupový regulátor jako na Obr.2.1. Vytvořené schéma (Obr.2.14) jsem uložil jako `MISOcontroller` (Multiple-input single-output) a vygeneroval z něj pomocí pluginu Embedded Coder projekt v jazyce C.




Obrázek 2.14: Vícevstupový regulátor

Díky přítomným nezapojeným vstupům obsahuje vygenerovaná struktura `MISOcontroller_U` pět prvků, se kterými mohou zacházet propojovací soubory nativního rozhraní. Relaci generovaných souborů zachycuje Obr.2.15.




Obrázek 2.15: Relační schéma vytvořených souborů

### Obalovací kód v Javě

Do hlavního projektu jsem stejně jako v předchozích případech přidal obalovací třídu, kterou jsem nazval `SimulinkControllerMISO` (Obr.2.16).

```
private native float generatedStep(float e);

private native float generatedStep
 (float e1, float e2);

private native float generatedStep
 (float e1, float e2, float e3);

private native float generatedStep
 (float e1, float e2, float e3, float e4);

private native float generatedStep
 (float e1, float e2, float e3, float e4, float e5);

static {
 try{
   System.load("/slotcar/lib/libMISOController.so");
   }
 catch(UnsatisfiedLinkError e){
   System.out.println("Cannot load
      libMISOController.so, file does not exist.");
   }
 }
```

Obrázek 2.16: Deklarace nativních metod a načtení sdílené knihovny

Při inicializaci třídy se načítá sdílená knihovna `libMISOController.so`, která obsahuje pomocné soubory nativního rozhraní a generované soubory vícevstupového regulátoru. Obalovací třída dále obsahuje deklaraci pěti nativních metod `generatedStep()`, které jsou přetížené a mají různý počet vstupních argumentů. Implementace těchto metod je v samostatném souboru, který je rozhraním JNI vyžadován ke správnému připojení souborů s kódem v jazyce C do projektu v Javě. Soubor je stejný, jako dříve zmíněné propojovací soubory (Obr. 2.5 a 2.11). Namísto jedné metody ovšem obsahuje implementaci všech pěti přetížených. Na Obr.2.17 uvádím pouze metodu s pěti vstupy.

Jazyk C v principu nepodporuje přetěžování metod tak jako Java, takže v kódu nemůže být deklarováno více metod se stejným názvem. Avšak rozhraní JNI tuto funkcialitu přidává. Na konec názvu metody, jehož syntaxe je

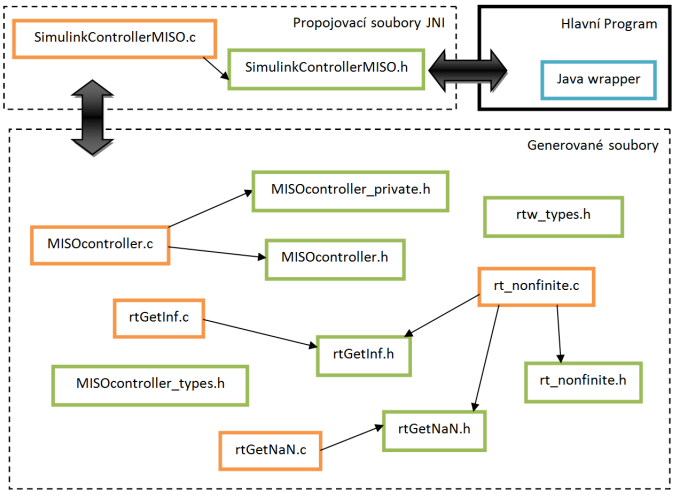
```
JNIEXPORT float JNICALL Java_slotcar_pi_controller_
SimulinkControllerMISO_generatedStep__FFFFF
(JNIEnv *env, jobject obj, jfloat e1, jfloat e2,
   jfloat e3, jfloat e4, jfloat e5) {
      MISOcontroller_U.In1 = e1;
      MISOcontroller_U.In2 = e2;
      MISOcontroller_U.In3 = e3;
      MISOcontroller_U.In4 = e4;
      MISOcontroller_U.In5 = e5;
      MISOcontroller_step();
      return MISOcontroller_Y.Out1;}
```

Obrázek 2.17: Implementace metody generatedStep() s pěti vstupy

striktně daná, se musí přidat specifikátory, které souvisejí s počtem vstupních argumentů a odlišují od sebe názvy přetížených metod, takže pro kompilátor jazyka C se metody jeví jako nepřetížené. V tomto případě přetížené metody, která má pět vstupních parametrů typu `float`, je vyžadována koncovka $\_\_\_FFFFF$. Celý název přetížené metody je tedy dán následovně:

Java_package_ClassName_methodName___numOfArgs

Takto vytvořený implementační soubor jsem spolu s hlavičkovým přidal ke zbytku vygenerovaného kódu ze Simulinku a zkompiloval do již zmíněné knihovny `libMISOController.so` (Obr.2.18).




Obrázek 2.18: Relační schéma vytvořených souborů

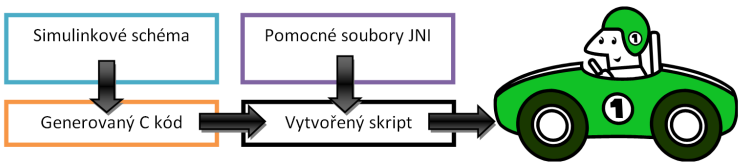
## 2.2 Přenos generovaných souborů do autíčka

### 2.2.1 Rozbor problematiky

Problémem přenosu souborů do autodráhového vozidla jsem se zabýval delší dobu. Z počátku jsem si myslel, že nejlepší řešení by bylo vytvořit nový, nebo použít již existující target, což je nástroj, který je součástí pluginu Embedded Coder. Target slouží k nahrávání zkompilovaného kódu ze Simulinku nebo Matlabu do externích zařízení.[6] Hlavní výhodou targetu by byla možnost přenosu souborů přímo ze Simulinku. Při řešení jsem ale narazil na problém, který jsem nedokázal překonat, a tak jsem se uchýlil k variantě vytvoření skriptu. Vzhledem k tomu, že ve vozidlech je operační systém Linux a na pracovním PC Windows, musí dojít k překladu generovaných souborů do knihovny až na cílovém autíčku. Kompilátor vytváří knihovnu, která je závislá na platformě. V případě Windows se jedná o knihovnu s koncovkou .dll, s kterou operační systém Linux nedokáže pracovat (pro něj jsou typické koncovky .so). Další výhoda skriptu může spočívat v tom, že dokáže soubory přenést na více aut zároveň.

### 2.2.2 Tvorba skriptu

Dávkový soubor jsem psal ve skriptovacím jayzce Batch, který je vyvinut pro operační systémy od Microsoftu. Mé řešení má tedy nevýhodu v tom, že je závislé na platformě Windows, na které se náš projekt vyvíjí. Tuto variantu jsem zvolil hlavně kvůli tomu, že jsem se již s tímto jazykem setkal, takže pro mě nebyl úplně cizí. Do budoucna bych možná skript předělal, aby byl nezávislý na platformě, nebo bych se více zabýval problematikou vytvoření targetu, který by do projektu přinesl pro uživatele snazší zacházení s generovanými soubory a skript by zcela nahrazoval. V současnosti je tedy přenos souborů navrhnut jako na Obr.2.19.




Obrázek 2.19: Relační schéma přenosu souborů

Vytvořený dávkový soubor pracuje s programy *plink* a *pscp*, které jsou součástí open source SSH klientu *PuTTY*[8]. Utilita *pscp* (PuTTY secure copy client) slouží k přenosu souborů mezi dvěma počítači pomocí protokolu SSH ([8],Chapter 5). Nástroj *plink* tento protokol používá ke vzdálenému připojení a k automatickému vykonání zadaných příkazů na připojeném zařízení ([8],Chapter 7). V mém případě jsem nástroj použil ke kompilaci přenesených

souborů do sdílené knihovny na autíčku. Jak jsem předestřel dříve, je to z důvodu, že projekt je vyvíjen na operačním systému Windows, ale v autíčkách je systém Linux. Funkce skriptu je tedy přenést všechny potřebné soubory a v autíčku je přeložit do knihovny. Na začátku skriptu je deklarován list aut, do kterých se kód nahrává, takže pokud uživatel chce list cílových IP adres měnit, musí to opravit zde (Obr.2.20).

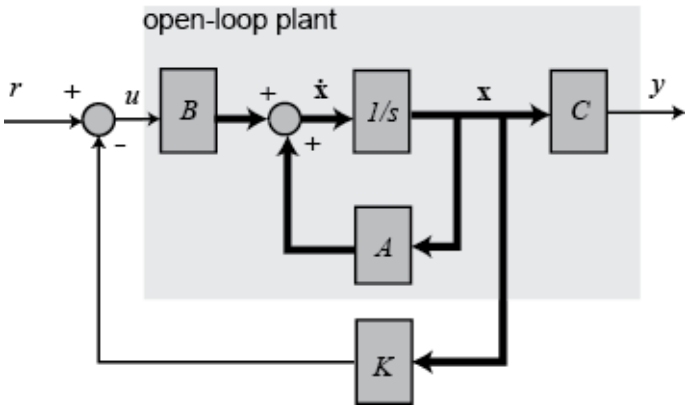
```
::variables for the script
set carList=192.168.1.102 192.168.1.103 192.168.108

::username and password used for login into slotcars
set carUser=pi
set carPass=raspberry
```

Obrázek 2.20: Úprava skriptu pro přenos souborů

Skript může být spuštěn se třemi možnými přepínači – *siso*, *tiso* a *miso* pro cílené přeložení pouze jednoho vygenerovaného regulátoru. Pokud se program pustí bez přepínačů, dojde k překladu všech tří. Dávkový soubor jsem se snažil napsat co nejrobustnější, aby dokázal reagovat na chyby uživatele, které vzniknou nevědomostí nebo nepozorností. Skript tedy upozorňuje uživatele na špatně zadaný argument a před začátkem přenosu kontroluje, zda jsou cílové složky vytvořeny a zda existují všechny potřebné soubory. Pro vyvolání nápovědy lze použít přepínač *help*.

## 2.3 Stavové regulátory

Pro implementování stavových regulátorů do našeho projektu jsem se snažil napsat regulátor v Javě, který by fungoval na stejném principu jako simulinkové schéma (Obr.2.21).




Obrázek 2.21: Schéma stavového regulátoru[9]

Schéma je stavový popis systému, který dostává na vstupu regulační odchylku $r$, obohacený o zpětnovazební regulátor $K$. Pro diskrétní systém s jedním vstupem a jedním výstupem můžu takto vytvořený model přepsat do diferenčních rovnic

$$\begin{aligned} \mathbf{x}[k+1] &= (\mathbf{A} - \mathbf{BK})\mathbf{x}[k] + \mathbf{B}r[k] \\ y[k] &= \mathbf{Cx}[k] + \mathbf{D}u[k], \end{aligned} \tag{2.2}$$

kde matice $A, B, C$ a $D$ jsou matice, které definují systém a matice $K$ určuje regulátor. V projektu jsem vytvořil třídu `StateSpaceController`, která tyto rovnice implementuje. Obsahuje parametry, které reprezentují výše zmíněné matice, a které se dají měnit pro testování různých vlastností stavových regulátorů. Třída rozšiřuje abstraktní třídu `Controller`, takže obsahuje metodu `step()`, která v každém kroku vrací výstup regulátoru, a dá se tak použít pro řízení autíček.

## 2.4 Konfigurace regulátorů přes XML soubor

V rámci zadání jsem se měl také zamyslet nad možností nastavování parametrů generovaných regulátorů pomocí konfiguračního XML souboru přímo z Matlabu. Pro uživatele by bylo jednodušší, kdyby mohl schéma s regulátorem přeložit, nahrát do řízeného autíčka, a potom pouze měnit konstanty regulátoru v Simulinku bez nutnosti nové kompilace. V průběhu semestru jsem ale došel k závěru, že toho nepůjde dosáhnout. Embedded Coder konstanty regulátoru překládá do C kódu přímo jako čísla, nikoliv jako proměnné, s kterými by se dalo zacházet a měnit je. Při zkompilování vzniká tedy kód, který je velmi těžko editovatelný. Při změně konstant regulátoru se tedy musí schéma znovu přeložit a v autíčku vytvořit nová knihovna.

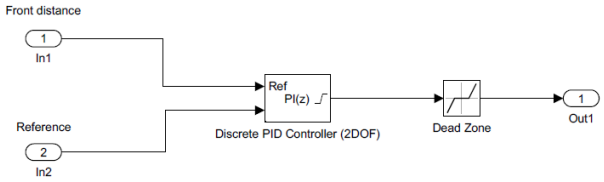
Kapitola 3

# Testování a experimentální ověření

Pro odzkoušení funkčnosti napsaných wrapperů jsem v Simulinku navrh tři regulátory - vzdálenostní regulátor, dvousměrný vzdálenostní regulátor a regulátor vzdálenosti s informacemi o rychlosti vedoucího vozu kolony. Pro první zmíněný jsem použil dvouvstupový wrapper, pro zbylé dva wrapper s proměnným počtem vstupů. Všechny tři pokusy jsem prováděl na koloně obsahující tři vozidla, jedno řízené počítačem a zbylé dvě autonomně.

## 3.1 Vzdálenostní regulátor

Simulinkové schéma (Obr. 3.1) prvního testovaného regulátoru má dva externí vstupy – měřená vzdálenost před autem (In1) a referenční vzdálenost (In2). Jedná se o koncept ACC (Adaptive Cruise Control), kdy auto reguluje svou rychlost podle měřené vzdálenosti před vozidlem.

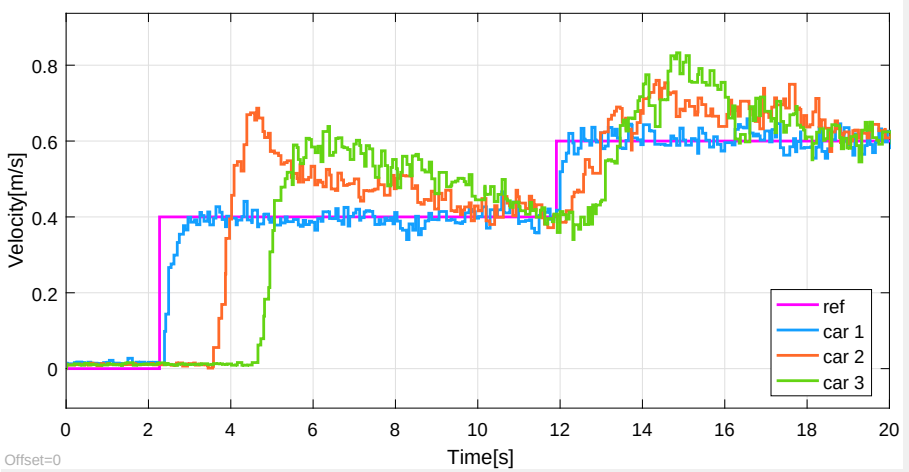



Obrázek 3.1: Schéma vzdálenostního regulátoru

Konstanty PI bloku jsem nastavil na hodnoty, které jsem již zmiňoval dříve, ale pro lepší přehlednost je zde uvedu znovu.

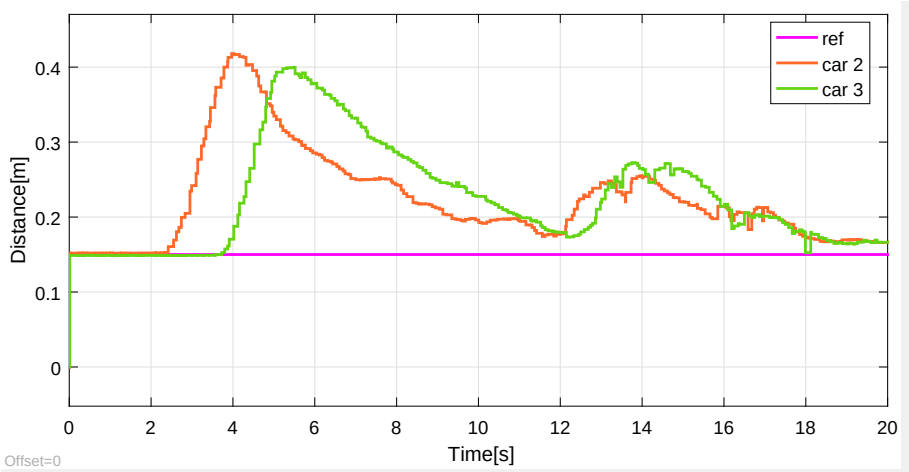
$$\begin{aligned} k_p &= 3 \\ k_i &= 0.8 \end{aligned} \tag{3.1}$$

Při tvorbě regulátoru jsem uvažoval i nelinearity. Saturaci jsem nastavil na ±1.5m/s a pásmo necitlivosti na ±0.1m/s. V obou případech vycházím z již navržených regulátorů.




Obrázek 3.2: Průběh rychlostí vzdálenostního regulátoru

Na grafech 3.2 a 3.3 je vidět časový průběh rychlostí, respektive vzdáleností před vozidlem. Pro pokus jsem použil řídící signál, který zprvu udržuje čelní vozidlo kolony v klidu, přibližně po dvou sekundách změní referenční rychlost na 0.4m/s a na dvanácti sekundách na 0.6m/s.

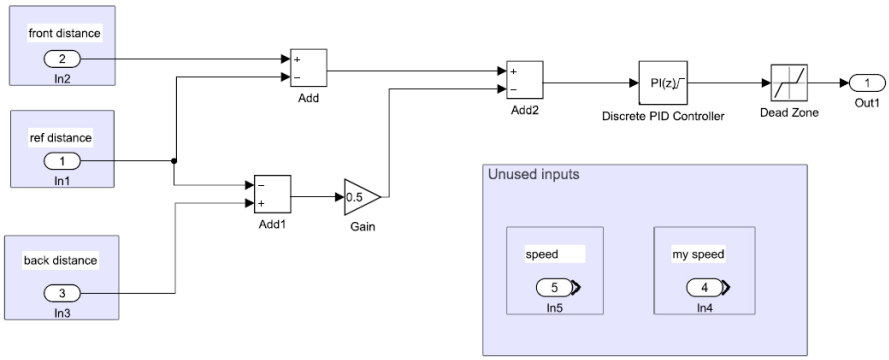



Obrázek 3.3: Porovnání měřené a referenční vzdálenosti

Z obou grafů lze pozorovat, že obě řízená vozidla (car 2, car 3) reagují s určitým zpožděním. Z grafu rychlostí je patrné, že by mohl být regulátor rychlejší, protože při rozjezdu byla měřená vzdálenost více než dvojnásobkem referenční. Odzkoušení dvouvstupového wrapperu ale proběhlo úspěšně.

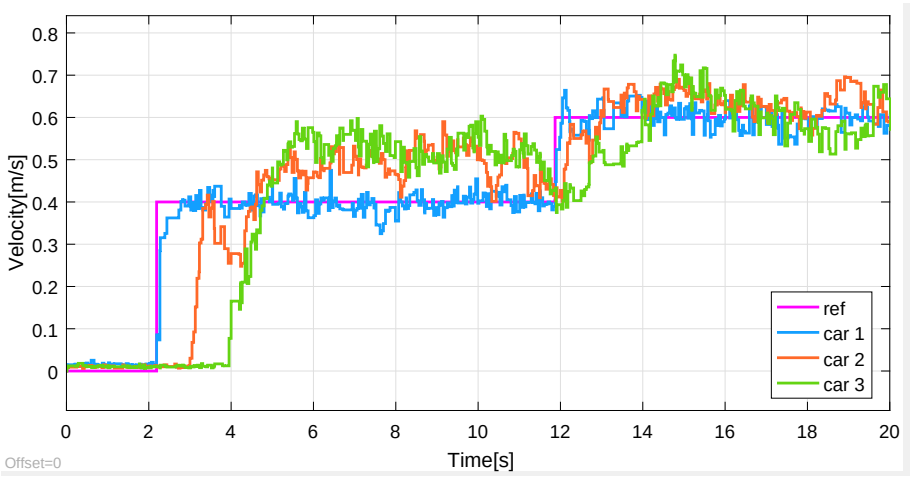
## 3.2 Dvousměrný regulátor

V druhém testovaném regulátoru jsem uvažoval měření vzdálenosti před i za vozidlem. Pro implementaci jsem použil vícevstupový wrapper (Obr.3.4) se třemi externími vstupy – referenční vzdálenost (In1), měřená vzdálenost před vozidlem (In2) a měřená vzdálenost za vozidlem (In3).




Obrázek 3.4: Schéma dvousměrného regulátoru

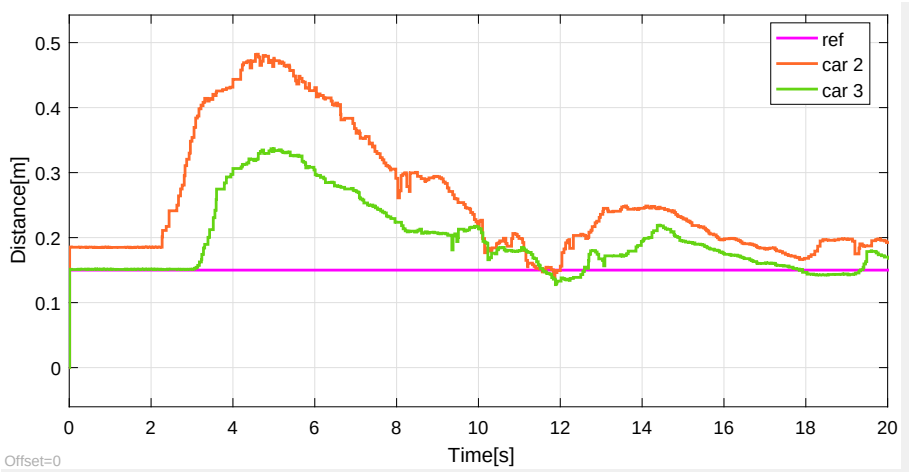
Konstanty PI bloku jsem nastavil na stejné hodnoty jako v předchozím případě (3.1), a stejně tak i saturaci (±1.5m/s) a pásmo necitlivosti (±0.1m/s). Řídící signál jsem také použil stejný a i zde si lze povšimnout reakčního zpoždění u obou řízených vozidel.




Obrázek 3.5: Průběh rychlostí dvousměrného regulátoru

Z grafu rychlostí (Obr.3.5) jde vidět, že regulátor jsem navrhl špatně, protože obě řízená autíčka mají velké kmitání v průběhu rychlostí. Špatné vlastnosti regulátoru lze pozorovat i v průběhu prvního řízeného autíčka (car 2)

okolo čtvrté vteřiny, kdy vozidlo kvůli měření zadní vzdálenosti čeká na rozjezd třetího vozu.

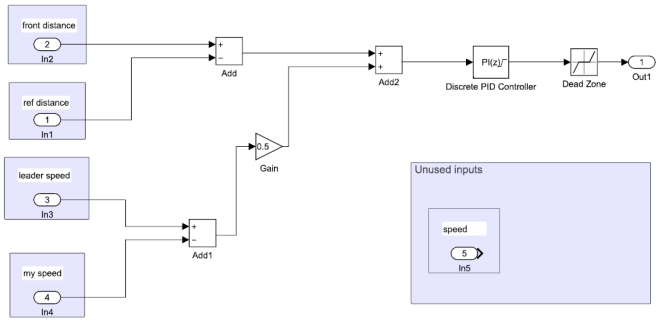



Obrázek 3.6: Porovnání měřené a referenční vzdálenosti

Této chyby si lze všimnout i v grafu měřených vzdáleností před vozidly (3.6). První řízené vozidlo nabralo kvůli čekání velké ztráty a měřená vzdálenost před vozidlem byla velmi vysoká.

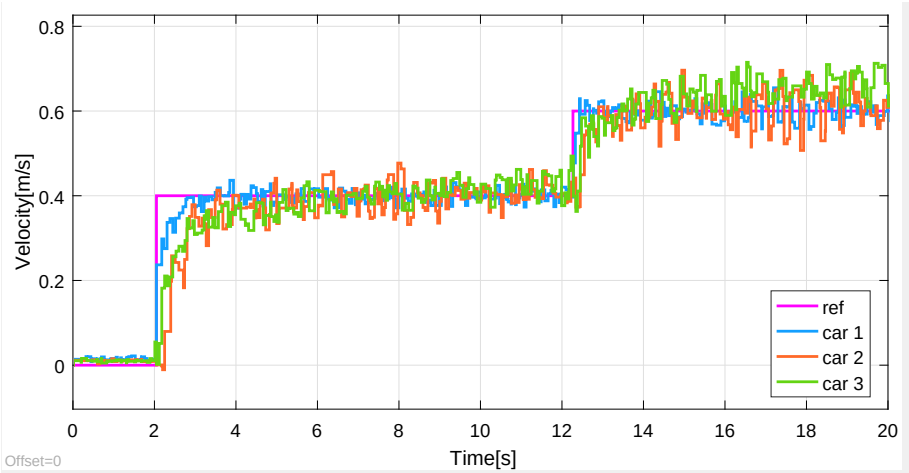
## 3.3 Regulátor s informacemi o rychlosti leadera

Třetí regulátor využívá k řízení informace o rychlosti vedoucího vozu. Jedná se o obdobou konceptu CACC (Cooperative Adaptive Cruise Control), kdy auto při regulaci rychlosti komunikuje s ostatními účastníky provozu. Testovaný regulátor jsem opět implementoval pomocí vícevstupového wrapperu (Obr.3.7).



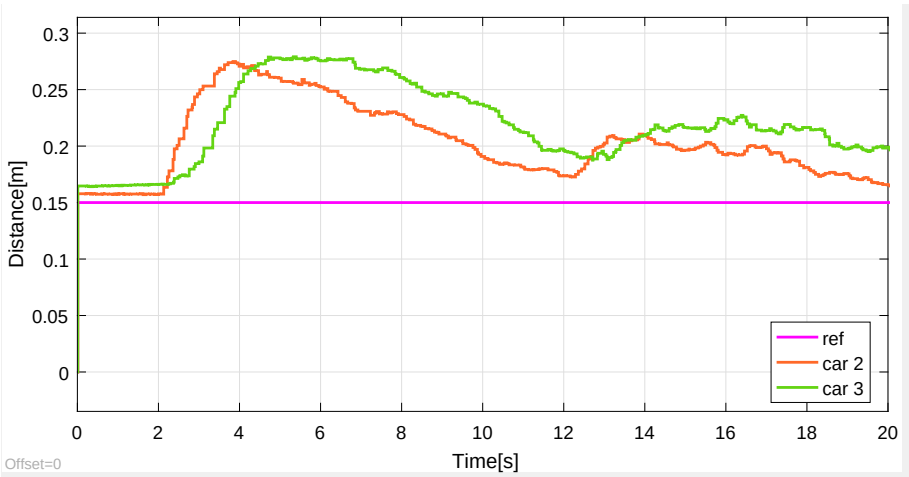

Obrázek 3.7: Schéma regulátoru s informacemi o rychlosti vedoucího vozu

Konstanty regulátoru jsou totožné s předchozími případy (3.1), stejně tak i nelinearity – saturační limit jsem nastavil na ±1.5m/s a pásmo necitlivoti na ±0.1m/s. Totožný je i řídící signál.




Obrázek 3.8: Průběh rychlostí regulovaných vozů

Na grafu zaznamenávajícím rychlosti vozů v průběhu času (3.8) lze pozorovat velmi dobré výsledky při rozjezdu vozidel, kdy je reakční zpoždění oproti prvním dvěma regulátorům téměř minimální.




Obrázek 3.9: Porovnání měřené a referenční vzdálenosti

Avšak při pohledu na graf porovnávající referenční a měřené vzdálenosti je vidět, že regulátor má s udržováním vzdálenosti větší problémy než předchozí dva typy a pro lepší výsledky bych ho musel odladit. Mou snahou zde ale bylo ukázat funkčnost navržených wrapperů, kterou jsem díky tomuto testování dokázal.

Kapitola 4

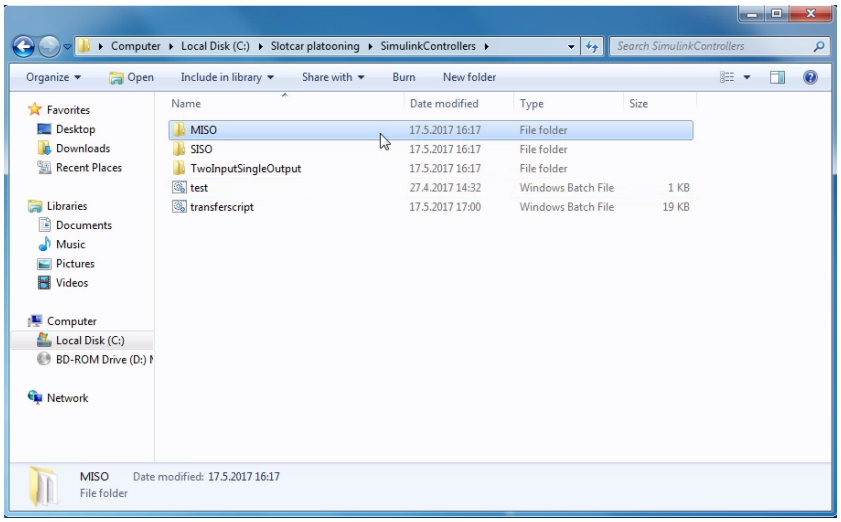
# Instrukce k zacházení

Pro snadné zacházení s wrappery bych nakonec uvedl stručný návod. Na konci kapitoly uvádím i odkaz na instruktážní video, ve kterém ukazuji jeden modelový příklad zacházení s wrappery.

## 4.1 Návod

### 4.1.1 Výběr wrapperu

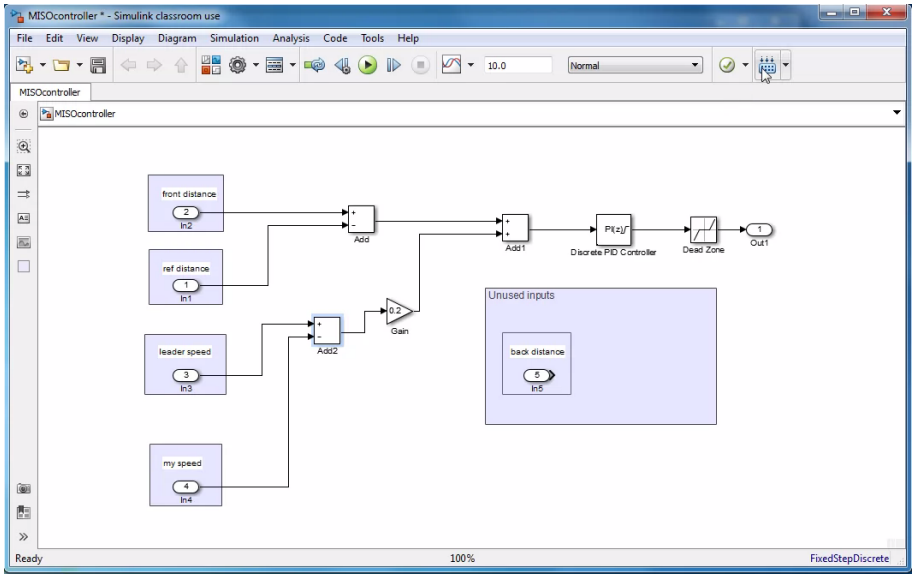
Ve složce `SimulinkControllers`, která je umístěna v kořenovém adresáři projektu, se nachází všechny tři vytvořené wrappery – jednovstupový (`SISO`), dvouvstupový (`Two-Input Single-Output`) a wrapper s proměnným počtem vstupů (`MISO`). Uživatel si zvolí, se kterým wrapperem chce zacházet a otevře si příslušné simulinkové schéma.




Obrázek 4.1: Složka s wrappery

### 4.1.2 Úprava schématu a generování C kódu

Schéma si upraví podle potřeby a pomocí pluginu Embedded Coder z něj vygeneruje C kód.




Obrázek 4.2: Simulinkové schéma vybraného regulátoru

Je nutné dát si pozor na podporované funkce a bloky, protože Embedded Coder neumí překládat úplně vše (seznam podporovaných funkcí je uveden v dokumentaci pluginu[10]). Po vygenerování kódu uživatel spustí `transferscript` (viz. Obr.4.1), který vygenerovaný kód pošle do vozidla a zkompiluje jej do knihovny. Pokud se zachází s novými vozidly, musí se jejich IP adresy přidat do `carListu` uvnitř skriptu.

```
::variables for the script
set carList=192.168.1.102 192.168.1.103 192.168.108

::username and password used for login into slotcars
set carUser=pi
set carPass=raspberry
```

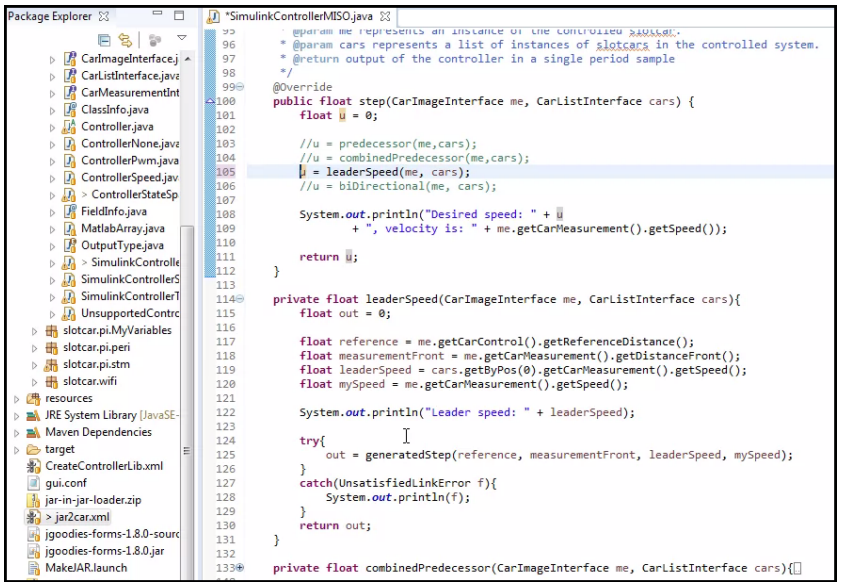
Obrázek 4.3: Úprava skriptu pro přenos souborů

Samotný skript lze pustit se třemi přepínači v závislosti na tom, zda chce uživatel přeložit jen jeden konkrétní regulátor. Bez přepínače přeloží skript všechny tři. Podporované přepínače jsou *siso tiso miso*.

### 4.1.3 Úprava Java wrapperu

Vytvořené Java wrappery se nacházejí v balíku `slotcar.pi.controller` uvnitř projektu `slotcar-sw`. Uživatel si vybere příslušný wrapper napsaný pro simu-

linkové schéma, které použil, a v metodě `step()` předá požadované vstupní signály generovanému kódu (nativní metoda `generatedStep()`).

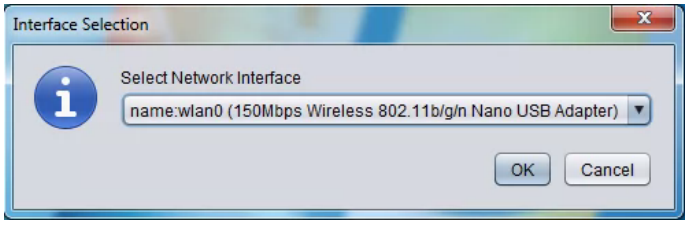



Obrázek 4.4: Kód Java wrapperu

Nakonec musí uživatel spustit skript `MakeJAR.launch`, který zabalí třídy projektu do .jar souboru, a následně skript `jar2car.xml`, který odešle novou verzi programu do řízených vozidel. Oba zmíněné skripty byly v projektu vytvořeny kolegou Martinem Ládem a nacházejí se uvnitř projektu `slotcar-sw` (viz Package Explorer na Obr. 4.4).

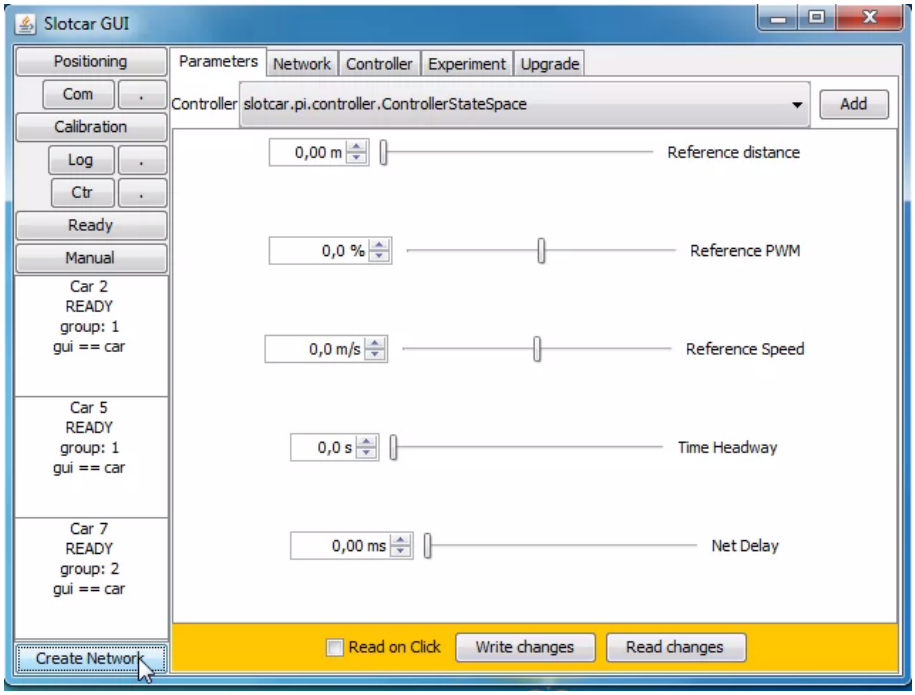
### 4.1.4 Komunikace s vozidly pomocí grafického rozhraní

Po skončení posledního skriptu lze spustit grafické rozhraní (hlavní třída projektu). Grafické rozhraní jsem také nenavrhoval já, ale z drtivé většiny patří zásluhy již zmiňovanému kolegovi Martinu Ládovi.




Obrázek 4.5: Výběr sítě

Po zapnutí nabídne program uživateli vybrat z dostupných síťových rozhraní (Obr.4.5) a posléze naběhne okno s hlavními funkcemi (Obr.4.6).




Obrázek 4.6: Grafické uživatelské rozhraní komunikačního programu

Zde si lze pro jednotlivá auta volit typ regulátoru a nastavovat mu různé parametry. Příkaz pro zahájení řízení se vyšle po zmáčknutí tlačítka `Ctrl` a tlačítkem stop (označeném tečkou) se řízení ukončí. Pokud navržený regulátor počítá s komunikací mezi vozidly (např. informace o rychlosti čelního vozu), je nutné před začátkem řízení zapnout i komunikaci (tlačítko `Com`), jinak regulátor nebude fungovat podle navrženého modelu.

## 4.2 Insktruktážní video

Ve videu ukazuji, jak zacházet s wrappery od návrhu simulinkového schéma až po řízení vozidel na autodráze. Video jsem uložil na portál YouTube (odkaz na video `https://youtu.be/MsdfnZxJcNo`) a lze jej nalézt i na přiloženém CD.

# Závěr

Bakalářská práce spočívala v napsání funkčních wrapperů, které by obalily kód v jazyce C generovaný ze Simulinku a začlenily ho tak do projektu realizovaného v jazyce Java jakožto palubní regululátory autodráhových autíček.

Prvním úkolem bylo vytvořit zmiňovaný Java wrapper pro C kód, který jsem ze Simulinku generoval díky pluginu Embedded Coder. Kód jsem do Java projektu začlenil pomocí Java nativního rozhraní ve formě sdílené knihovny. K tomu jsem musel vytvořit pomocné propojovací soubory v jazyce C, které jsem také přidal do zkompilované knihovny. Pro přenos generovaných souborů jsem vytvořil skript. Tento skript je závislý na platformě Microsoft Windows, což bych chtěl v budoucnu opravit, nebo skript nahradit napsáním targetu.

Druhým bodem zadání bylo vytvořit více typů wrapperů pro různé druhy regulátorů. Vytvořil jsem tedy wrappery pro jednovstupový a dvouvstupový regulátor. V rámci tohoto bodu zadání jsem se měl také zamyslet nad možností vytvořit wrapper pro regulátor s proměnným počtem vstupů. Takový wrapper se mi nakonec vytvořit povedlo, avšak práce s ním není tak intuitivní, jak se na začátku očekávalo. Obalovací kód jsem vytvořil pro maximálně pětivstupový regulátor a kvůli struktuře generovaného kódu musí být v simulinkovém schéma tohoto wrapperu vždy přítomny všechny externí vstupy, ikdyž nejsou využívány.

Dalším bodem bylo začlenit do projektu stavové regulátory. Vytvořil jsem tedy regulátor, který svůj výstup počítá pomocí stavového popisu s tím, že jednotlivé stavové matice $A, B, C, D$ a matice regulátoru $K$ jsou implementovány jako atributy třídy stavového regulátoru a musejí se měnit v kódu.

Předposledním úkolem bylo zamyslet se nad možností implementace nastavování parametrů generovaných regulátorů přímo z Matlabu přes konfigurační XML soubor, aby se nemuselo vžy při každé změně parametrů překládat celé

schéma. V průběhu mé práce jsem ale došel k závěru, že tato realizace nebude možná. Java wrappery pro řízení vozidel využívají generované funkce, které jsou zabalené v knihovně a není možné nijak měnit jejich strukturu. Problém vidím ve způsobu, jakým Embedded Coder generuje C kód z navrženého schématu.

Posledním úkolem zadání bylo demonstrovat funkčnost navržených wrapperů na několika typech regulátorů. Úspěšné navržení jsem předvedl na třech regulátorech. Dvouvstupový wrapper jsem testoval na vzdálenostním regulátoru, který porovnával měřenou vzdálenost před vozidlem s referenční hodnotou. Wrapper pro regulátor s proměnným počtem vstupů jsem odzkoušel na dvousměrném regulátoru a regulátoru s informací o rychlosti vedoucího vozu kolony. Stavový regulátor jsem odzkoušel na velmi jednoduchém sledovači, který se zpožděním přenášel zesílený vstup na výstup.

# Literatura


[1] Lád, Herman, Hurák. *Vehicular platooning experiments using autonomous slot cars*. ©2017, Dostupné z: http://aa4cc.dce.fel.cvut.cz/sites/default/files/downloads/publications/ifac2017.pdf

[2] Moravec Jan. *Distribuované řízení kolon vozidel na autodráze*. ©2014, České vysoké učení technické v Praze, vedoucí práce Ing. Ivo Herman, Dostupné z: https://dspace.cvut.cz/bitstream/handle/10467/24299/F3-BP-2014-Moravec-Jan-prace.pdf

[3] Lád Martin. *Návrh a implementace řídicího systému pro autodráhové vozidlo*. ©2014, České vysoké učení technické v Praze, vedoucí práce Ing. Dan Martinec, Dostupné z: https://support.dce.felk.cvut.cz/mediawiki/images/a/a4/Bp_2014_lad_martin.pdf

[4] MathWorks. *Embedded Coder*. ©1994-2017, Dostupné z: https://se.mathworks.com/products/embedded-coder.html

[5] Oracle. *Java Native Interface*. ©1993,2016, Dostupné z: https://docs.oracle.com/javase/8/docs/technotes/guides/jni/

[6] MathWorks. *What is a target*. ©1994-2017, Dostupné z: https://se.mathworks.com/help/supportpkg/armcortexa/ug/what-is-a-target.html

[7] Chua Hock-Chuan. *Java Programming Tutorial*. ©2014, Dostupné z: https://www3.ntu.edu.sg/home/ehchua/programming/java/JavaNativeInterface.html

[8] Simon Tatham. *PuTTY User Manual*. ©1997-2017, Dostupné z: https://the.earth.li/~sgtatham/putty/0.69/htmldoc/

[9] Control Tutorials for Matlab and Simulink. *State-Space Methods for Controller Design*. CC Attribution-ShareAlike, Dostupné z: http://ctms.engin.umich.edu/CTMS/Content/Introduction/Control/StateSpace/figures/StateSpaceTutorial_ControlDesign_BlockDiagram.png

[10] MathWorks. *Embedded Coder Functions*. ©1994-2017, Dostupné z: https://se.mathworks.com/help/ecoder/functionlist.html

Příloha **A**

# Obsah přiloženého CD

- `slotcar-sw` ..... adresář s Java projektem
- `SimulinkControllers`
  - `SISO` ..... jednovstupový regulátor
  - `TwoInputSingleOutput` ..... dvouvstupový regulátor
  - `MISO` ..... vícevstupový regulátor
  - `transferscript.bat` ..... skript pro přenos souborů
- `text`
  - `thesis.pdf` ..... text práce ve formátu PDF
  - `thesis.tex` ..... text práce ve formátu LaTeX
  - `pictures` ..... zdrojové obrázky pro formát LaTeX
- `video`
  - `tutorial.mp4` ..... instruktážní video