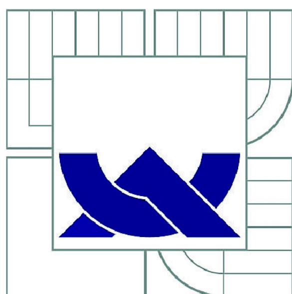
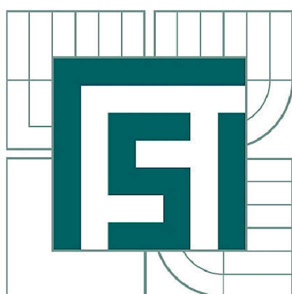


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA STROJNÍHO INŽENÝRSTVÍ
ÚSTAV MECHANIKY TĚLES, MECHATRONIKY A BIOMECHANIKY

FACULTY OF MECHANICAL ENGINEERING
INSTITUTE OF SOLID MECHANICS, MECHATRONICS AND BIOMECHANICS

SOFTWARE PRO SIMULACI ŘÍDICÍCH ALGORITMŮ S GENEROVÁNÍM C-KÓDU PRO MIKROKONTROLÉR

SOFTWARE FOR SIMULATION OF CONTROL ALGORITHMS WITH C CODE GENERATION FOR MICROCONTROLLER

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE MARTIN FIALA
AUTHOR

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

BRNO 2011

Vysoké učení technické v Brně, Fakulta strojního inženýrství

Ústav mechaniky těles, mechatroniky a biomechaniky
Akademický rok: 2010/2011

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

student(ka): Martin Fiala

který/která studuje v **bakalářském studijním programu**

obor: **Mechatronika (3906R001)**

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 bakalářské práce:

Software pro simulaci řídicích algoritmů s generováním C-kódu pro mikrokontrolér

v anglickém jazyce:

Software for simulation of control algorithms with C code generation for microcontroller

Stručná charakteristika problematiky úkolu:

Práce se bude zabývat návrhem a implementací vlastního prostředí pro simulaci, testování a generování kódu pro mechatronické aplikace.
Předpokládáme:
- využití open-source prostředí (např. jazyk Python)
- propojení s knihovnou Open Dynamic Engine (ODE) pro simulaci dynamiky mechanických soustav
- generování bloků C kódu a časování pro jednoduché prvky řídicích obvodů.

Student musí mít zájem o programování a vhodná (ne nutná) je i minimální zkušenost s některým z mikrokontrolérů (např. Atmel AVR).

Více také na http://www.umt.fme.vutbr.cz/mechlab
Pro podrobnější informace kontaktujte emailem garanta.

Cíle bakalářské práce:

- Rešerše v oblasti použitelných komponent pro návrh softwaru.
- Studium ODE (k dispozici je několik BP).
- Návrh struktury.
- Implementace a ověření fungování na jednoduchých příkladech.

Seznam odborné literatury:

- Mann, B.: C pro mikrokontroléry, Nakladatelství BEN, 2003
- Váňa, V.: Mikrokontroléry ATMEL AVR - assembler, Nakladatelství BEN, 2003
- Herout, P.: Učebnice jazyka C

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

Termín odevzdání bakalářské práce je stanoven časovým plánem akademického roku 2010/2011.

V Brně, dne 29.4.2011

L.S.

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

Abstrakt

Tato práce se zabývá simulací průběhu signálu v zadané jednoduché struktuře s jedním regulátorem, řízenou soustavou a zpětnou vazbou pomocí open-source programovacího jazyka Python. V další části je rozebíráno generování C kódu regulátoru pro mikrokontroléry a následnou kontrolu průběhu signálu, který mikrokontrolér zpracovává. Tato kontrola je prováděna pomocí sériového rozhraní.

Abstract

This work is concerned with simulation of signal processing in a simple scheme with one controller, plant and feedback by open-source programming language Python. Moreover it is concerned with generating C-code for microcontrollers and in consequent check up of signal processing which is processed by microcontroller. This check up is done by serial interface.

Klíčová slova

simulace, mikrokontrolér, dsPIC, AVR, C, Python, ODE

Keywords

simulation, microcontroller, dsPIC, AVR, C, Python, ODE

FIALA, M. *Software pro simulaci řídicích algoritmů s generováním C-kódu pro mikrokontrolér*. Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, 2011. 60 s. Vedoucí bakalářské práce Ing. Robert Grepl, Ph.D.

Prohlášení:

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně na základě svých vědomostí, studia literárních zdrojů a pokynů vedoucího mé práce.

V Brně dne:

…………………
Martin Fiala

Poděkování

Děkuji vedoucímu bakalářské práce Ing. Robertu Greplovi Ph.D. za cenné rady, připomínky a metodické vedení práce.

OBSAH

Zadání závěrečné práce ... 3
Abstrakt ... 5
Obsah ... 9
1 Úvod ... 11
2 Jazyk Python a použité nástroje ... 13
2.1 Python ... 13
2.2 NumPy ... 13
2.2.1 Použité funkce z knihovny NumPy ... 13
2.3 SciPy ... 13
2.3.1 Použité funkce z knihovny SciPy ... 14
2.4 MatPlotLib ... 14
2.4.1 Použité funkce z knihovny MatPlotLib ... 14
2.5 pySerial ... 14
2.6 wxPython ... 14
2.6.1 wxGlade ... 14
2.7 PyODE ... 14
3 Open Dynamics Engine ... 15
3.1 Model v ODE ... 15
3.1.1 Dynamický svět ... 15
3.1.2 Tělesa ... 15
3.1.3 Vazby ... 15
3.1.4 Kolizní prostor ... 16
3.2 Simulace ... 16
4 Mikrokontroléry ... 17
4.1 Co je to mikrokontrolér ... 17
4.2 Použité mikrokontroléry ... 17
4.2.1 ATmega8 ... 17
4.2.2 dsPIC33FJ128MC804 ... 18
4.3 Jazyk C ... 19
5 Vlastní simulační program ... 21
5.1 Použití programu ... 21
5.2 Instalace programu ... 21
5.3 Popis programu ... 22
5.3.1 Simulace ... 22
5.3.2 Pole časů ... 23
5.3.3 Pole vstupů ... 23
5.3.4 Vykreslení výsledků ... 23
5.3.5 Generování C-kódu ... 23
5.3.6 Řízení mikrokontrolérem a jeho kontrola pomocí sériového portu ... 23
5.4 Grafické rozhraní ... 24
5.4.1 Struktura grafického rozhraní ... 24
5.5 Příklady simulace na počítači ... 27
5.5.1 Průběh signálu přes diskrétní regulátor se vstupem jednotkového skoku ... 27
5.5.2 Průběh signálu přes řízený systém se vstupem jednotkového skoku ... 28
5.5.3 Průběh signálu soustavou se vstupem jednotkového skoku bez zpětné vazby ... 28
5.5.4 Průběh signálu soustavou se vstupem jednotkového skoku se zpětnou vazbou 29
5.5.5 Průběh signálu soustavou se vstupem ve tvaru obdélníka se zpětnou vazbou ... 30
5.5.6 Průběh signálu soustavou se vstupem ve tvaru pily se zpětnou vazbou ... 30

5.5.7 Simulace nelineární soustavy - kyvadla 31
5.5.8 Návrh řízení kyvadla 32
5.5.9 Návrh řízení modelu kyvadla vytvořeného v ODE 32
5.5.10 Zobrazení více výstupních hodnot 35
5.5.11 Rozběh stejnosměrného motoru 36
5.5.12 Řízení polohy stejnosměrného motoru 37
5.6 Řízení modelu pomocí mikrokontroléru 38
5.6.1 Generování C-kódu s datovým typem double 39
5.6.2 Naprogramování mikrokontroléru 39
5.6.3 Regulace modelu pomocí mikrokontroléru 40
5.6.4 Regulace modelu pomocí mikrokontroléru s datovým typem integer 41
5.7 Řízení reálné aplikace pomocí mikrokontroléru 42
5.7.1 Regulace natočení motoru pomocí PID regulátoru se zpětnou vazbou 43
6 Závěr 45
7 Literatura a odkazy 47
8 Seznam použitých symbolů 49
Přílohy 51
I ODE 53
I.1 Typický postup vytváření a simulaci modelu ODE 53
I.1.1 Vytvoření modelu 53
I.1.2 Simulace modelu 54
II Program simulace 55
II.1 Definice funkcí ve skriptu Simulace.py 55
II.1.1 Funkce pro vytvoření pole času 55
II.1.2 Funkce pro vytvoření pole vstupů 55
II.1.3 Funkce pro simulaci 57
II.1.4 Funkce na vykreslení grafů 60
II.1.5 Funkce pro generování C-kódu 61
II.1.6 Funkce pro kontrolu mikrokontroléru po sériovém rozhraní 61
III Práce s mikrokontroléry 63
III.1 Modul s mikrokontrolérem ATmega8 63
III.2 Modul výkonového členu pro řízení motoru 64
III.3 Modul pro sériovou komunikaci 65
III.4 Modul s mikrokontrolérem dsPIC 66
III.5 Pracoviště 66

1 ÚVOD

Většina profesionálních aplikací pro simulaci a řízení mechatronických soustav je vytvářena v prostředí MATLAB/Simulink[20] vyvíjeném společností MathWorks. Využití tohoto prostředí s sebou nese klady i zápory. Jeho velkou výhodou je jeho univerzálnost. Je zde možné vytvořit model prakticky libovolně složité soustavy a následně ho i řídit. Další výhodou je také softwarová podpora ze strany velkého množství firem. Naopak jeho velká nevýhoda spočívá v jeho celkové vysoké ceně. Program MATLAB by byl zastupitelný nějakým volně šiřitelným nástrojem pro matematické výpočty, například programem Octave[21], ale nástroj Simulink pro simulaci průběhů signálů tak snadno nahradit nelze.

Záměrem této práce je vytvořit volně šiřitelný program pro simulaci průběhu signálu soustavou, která je tvořena regulátorem, řízeným systémem, zpětnou vazbou, šumem a generátorem referenční trajektorie. Model řízeného systému by měl mít možnost být zadán jako soustava diferenciálních rovnic vyjadřující jeho dynamiku. Simulátor by také měl zvládat řízení a simulaci modelu vytvořeného v Open Dynamics Engine.

Dalším cílem této práce je usnadnění vytváření řídících aplikací pro mikrokontroléry tím, že se bude generovat C-kód regulátoru, kterým bude možné mikrokontroléry naprogramovat a následně odsimulovat jejich řízení na sestaveném modelu.

K tomuto účelu je vhodné využít volně šiřitelný programovací jazyk Python, pro který existuje mnoho volně šiřitelných knihoven usnadňujících jeho použití.

2 JAZYK PYTHON A POUŽITÉ NÁSTROJE

Programovací jazyk Python byl na programování simulátoru vybrán kvůli jeho přehledné syntaxi, snadnosti vývoje aplikací a také díky širokému množství knihoven rozšiřující jeho funkce. Velkou výhodou tohoto jazyka byla i existence knihovny umožňující snadné propojení s knihovnou Open Dynamics Engine, jejíž použití je jedním z cílů zadání této práce.

2.1 Python

Python[3] je volně šiřitelný objektově orientovaný programovací jazyk, který je nezávislý na platformě. Běží tedy na systémech Windows, Linux, Mac OS a dalších. Díky přehledné syntaxi kódu a používání vysokoúrovňových datových typů probíhá vývoj programů v tomto jazyce rychleji než v tradičních jazycích, například v C nebo C++. Lze v něm vytvářet jak spustitelné aplikace, tak i skripty, které vyžadují instalaci kompletního programu Python.[1]

Pro lepší využití a rozšíření funkcí tohoto jazyka, byla pro matematické výpočty použita řada volně šiřitelných knihoven. Jde o knihovny NumPy, SciPy, MatPlotLib, pySerial, wxPython a PyODE. Jejich vlastnosti a v práci použité funkce jsou popsány v následujících kapitolách.

2.2 NumPy

NumPy[5] je matematická knihovna, která je potřebná k vědeckým numerickým výpočtům v Pythonu. Obsahuje funkce pro práci s maticemi, převody mezi různými typy dat a polí a mnoho dalších matematických funkcí (například Fourierova transformace nebo generování náhodných čísel).

2.2.1 Použité funkce z knihovny NumPy

numpy.ndarray

Datový objekt n-rozměrné pole (ndarray) z knihovny NumPy je v programu použit pro zjednodušení maticových výpočtů. Lze s ním provádět libovolné maticové operace, jako je například maticové násobení, dělení, sčítání nebo odečítání. [7][8]

numpy.rot90(A)

Funkce rot90 slouží k přetáčení matic o 90°. Tato funkce je použita pro převod řádkových vektorů na sloupcové, aby je bylo možné navzájem vynásobit. Do vstupního parametru se zadává proměnná typu ndarray. Stejný typ proměnné funkce následně i vrací.

numpy.dot(A, B)

Tato funkce slouží pro vynásobení dvou matic. Vstupy A a B jsou typu ndarray, výstupem je také typ ndarray.

2.3 SciPy

SciPy[9] znamená „Scientific Tools for Python“, neboli vědecké nástroje pro Python. Je to volně šiřitelná a jednoduše použitelná knihovna pro tento jazyk, která se využívá pro matematické a vědecké výpočty, například pro numerickou integraci, optimalizaci nebo pro práci s přenosy signálů. Je ovšem propojená s knihovnou NumPy (viz kapitola 2.2), ze které používá pro výpočty n-rozměrné pole (typ ndarray) a operace pro práci s ním.

2.3.1 Použité funkce z knihovny SciPy

scipy.signal.tf2ss(num, den)

Funkce slouží pro převod přenosové funkce na matice stavové rovnice. Vstup *num* znázorňuje čitatel a vstup *den* jmenovatel polynomu přenosové funkce. Výstupem jsou matice vyjadřující dynamiku stavového systému – *A*, *B*, *C*, *D*.[11]

scipy.integrate.odeint(func, y0, t, args = ())

Funkce slouží k integraci soustavy diferenciálních rovnic. Vstup *func* znázorňuje soustavu diferenciálních rovnic, vstup *y0* počáteční hodnotu, vstup *t* pole času, ve kterém se soustava integruje a vstup *args* představuj odkazy na další potřebné proměnné, které se vyskytují v soustavě diferenciálních rovnic.

2.4 MatPlotLib

MatPlotLib[13] je knihovna pro vykreslování 2D grafů, které lze následně uložit do souboru jako obrázek. Lze s ním vykreslit funkce, histogramy, barevná spektra nebo i sloupcové diagramy.

V projektu je použita funkce *pyplot* ke znázornění průběhu signálu, který prošel skrz danou soustavu.

2.4.1 Použité funkce z knihovny MatPlotLib

matplotlib.pyplot.plot(x, y)

Funkce vykresluje graf, kde *x* znázorňuje pole hodnot osy x a *y* pole hodnot osy y.[14]

matplotlib.pyplot.savefig(FileName)

Funkce ukládá vykreslený graf do souboru, ke kterému vede cesta zadaná v parametru *FileName*.

2.5 pySerial

PySerial[16] je knihovna umožňující jednodušší přístup a komunikaci po sériovém portu počítače.

2.6 wxPython

WxPython[26] je jedno z mnoha grafických rozhraní pro Python. Má otevřené zdrojové kódy a je použitelné jak pro platformu Windows, tak i pro Linux a Macintosh OS X. Je založeno na oblíbeném grafickém rozhraní wxWidgets určeném pro programovací jazyk C++.

2.6.1 wxGlade

Pomocí nástoje WxGlade[27] lze jednodušeji vytvářet grafické formuláře pro rozhraní wxPython. Jednotlivé prvky lze na formulář umístit pomocí myši a nástroj pak sám automaticky vygeneruje příslušný kód pro Python.

2.7 PyODE

PyODE[28] je rozhraní umožňující v Pythonu práci s volně šiřitelnou knihovnou Open Dynamics Engine[29] (zkr. ODE). Tato knihovna umožňuje simulaci dynamiky tuhých těles a bude blíže popsána v následující kapitole.

3 OPEN DYNAMICS ENGINE

Open Dynamics Engine je volně šiřitelná knihovna pro simulaci dynamiky tuhých těles. Je napsána pro programovací jazyky C/C++, ale s rozšiřující knihovnou pyODE je jednoduše použitelná i pro programovací jazyk Python. Knihovna ODE podporuje detekci kontaktů mezi tělesy, ale nemá žádné uživatelské prostředí, ani se nezabývá vykreslením simulace dynamiky těles. Na vykreslení se musí použít jiná grafická knihovna, například knihovna OpenGL.

Práce v ODE by se dala rozdělit na dvě části. Na vytváření modelu dynamického světa, který je blíže popsán v kapitole 3.1, a na vlastní simulaci dynamiky, která je blíže popsána v kapitole 3.2.

3.1 Model v ODE

Na začátku vytváření modelu v ODE se musí nejprve vytvořit dynamický svět. Do tohoto světa se pak přidávají tělesa. Mezi tělesy se mohou vytvářet různé typy vazeb, včetně vazeb kontaktních. Pokud je potřeba v simulaci počítat též s kontakty mezi tělesy, musí se v dynamickém světě vytvořit kolizní prostor a nastavit geometrii všech vytvořených těles. Bližší popis jednotlivých částí modelu se nachází dále v této kapitole.

3.1.1 Dynamický svět

Při vytváření dynamického světa je dobré nastavit globální konstanty, které ovlivňují běh simulace světa. Těmito konstantami jsou tíhové zrychlení, ERP a CFM.

ERP (Error Reduction Parameter) je konstanta, která nastavuje pevnost vazeb. Při rozpojení vazby do ní přidává další sílu, která vazbu navrací do původní polohy. Konstanta může nabývat hodnot od 0 do 1. Čím větší bude její velikost, tím menší bude šance, že se vazba rozpojí.

CFM (Constraint Force Mixing) je konstanta nastavující pružnost vazby, neboli jak velké bude omezení ve vazbách. Při hodnotě 0 je omezení ve vazbě velké a neumožňuje žádný pohyb. Zvyšující se hodnotou tolerance pohybu ve vazbě vzrůstá.

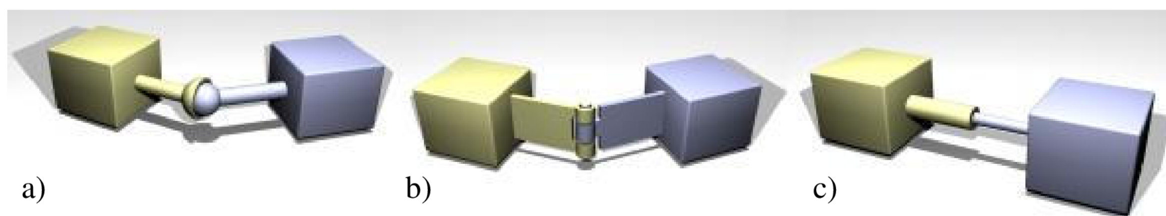
Do vytvořeného světa se pak mohou umístit další objekty, kterými jsou tělesa a vazby.

3.1.2 Tělesa

U každého vytvořeného tělesa v ODE umístěného ve světě lze nastavit několik parametrů, kterými jsou tvar, hmotnost, natočení tělesa či jeho poloha ve světě. K vytvoření požadovaného tvaru je na výběr z více geometrií. Lze vytvářet tělesa tvaru kvádru, koule, kapsle, válce, plochy nebo tělesa tvořená z trojúhelníkové sítě. Pomocí těchto geometrií se pak rozhoduje, zdali se nějaká tělesa dostávají do vzájemného kontaktu a vyhodnocují se kontaktní vazby.

3.1.3 Vazby

Vytvořená tělesa se dále mohou vzájemně pospojovat pomocí různých typů vazeb. V ODE se nachází vazby pevné, kontaktní, sférické, posuvné, rotační a rotační umožňující rotaci kolem dvou os.



Obr. 1 Vazby. a) Sférická. b) Rotační. c) Posuvná.[32]

U rotačních vazeb lze nastavit osy rotace.

U kontaktních vazeb se nastavují dva základní parametry – *Bounce* a *Mu*. Parametr *Bounce* určuje, jak silná bude reakce při srážce dvou těles. Jeho hodnota se pohybuje v rozmezí 0 až 1. Při hodnotě rovné 0 nevyvolá srážka mezi tělesy žádnou reakci, při hodnotě rovné 1 bude reakce maximální. Druhý parametr *Mu* ve vazbě představuje třecí konstantu.

3.1.4 Kolizní prostor

Kolizní prostor se v dynamickém světě vytváří proto, aby mohly být vyhodnocovány vznikající kolize mezi tělesy. Při každém kroku simulace pak v něm lze kontrolovat nově vzniklé kolize těles a vytvářet mezi tělesy kontaktní vazby.

3.2 Simulace

Simulace světa ODE je uskutečňována integrací rovnic charakterizujících daná tělesa po malých časových krocích. Zvětšováním časových kroků se ztrácí přesnost výsledků, naopak při velmi malém kroku může dojít k nestabilitě simulace.

Před každým krokem simulace světa se musejí znovu vytvořit síly a momenty působící na tělesa a zkontrolovat nově vzniklé kolize mezi tělesy, protože po provedení integračního kroku se všechny síly a momenty působící na tělesa vymažou a kontakty mezi tělesy se za daný časový krok mohou též změnit.

4 MIKROKONTROLÉRY

4.1 Co je to mikrokontrolér

Mikrokontrolér je logický obvod, který lze naprogramovat, aby vykonával určitou činnost. Hlavní výhodou jeho použití je, že jím můžeme nahradit složitější zapojení, které by se jinak muselo skládat z více jednoduchých logických obvodů.

Oproti mikroprocesoru, který se podílí jen na výpočtech, je mikrokontrolér plně soběstačná součástka, která obsahuje mnoho dalších periferií. Nejčastěji to jsou čítače a časovače, vstupně-výstupní porty, AD převodníky a spousta dalších periferií.

4.2 Použité mikrokontroléry

Ke splnění a dosažení cílů této bakalářské práce se pracovalo se dvěma typy mikrokontrolérů. Byly to mikrokontroléry z rodiny AVR a dsPIC. V této kapitole je dále uveden jejich bližší popis.

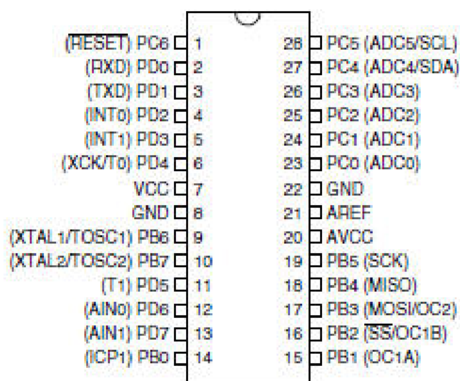
4.2.1 ATmega8

ATmega8 je osmibitový mikrokontrolér rodiny AVR od firmy Atmel. Dosahuje výkonu až 16MIPS (16 milionů instrukcí za sekundu) při maximální taktovací frekvenci 16MHz. Jeho napájecí napětí je 4,5V až 5,5V.

Mikrokontrolér je postavený na architektuře RISC, což znamená redukovaná instrukční sada. Návrh této instrukční sady je zaměřen na jednoduchost a zároveň optimalizaci strojových instrukcí. Délka provádění jedné instrukce trvá většinou jeden cyklus mikrokontroléru. Také šířka slova neboli bitová délka všech instrukcí je stejná.

Uvnitř mikrokontroléru jsou 3 druhy pamětí. První je paměť programu typu FLASH, která v sobě uchovává i po odpojení napájení instrukce programu a která má velikost 8kB. Druhá paměť je typu EEPROM. Je to trvalá paměť dat o velikosti 512kB, která také uchovává svůj obsah i po odpojení napájení. Avšak přístup k ní je pomalejší, než k paměti dat typu SRAM. Paměť SRAM má velikost pouze 1kB a její obsah se po odpojení napájení ztratí.

V mikrokontroléru je zabudována řada periferií. Obsahuje například vstupně-výstupní porty, dva osmibitové a jeden šestnáctibitový čítač a časovač s PWM výstupy, sériové rozhraní, 6 analogově digitálních převodníků, analogový komparátor atd.



Obr. 2 Schéma vývodů mikrokontroléru ATmega8 v pouzdře PDIP[30]

Různé funkce těchto periferií pak mohou vyvolat přerušení mikrokontroléru. To znamená, že se přeruší běh hlavního programu a začne se vykonávat podprogram, který dané přerušení obsluhuje. U tohoto mikrokontroléru jsou všechny přerušení na stejné úrovni. Z toho vyplývá, že obsluha druhého přerušení bude spuštěna teprve poté, co se ukončí obsluha prvního přerušení.

Mikrokontrolér také umožňuje naprogramování sebe samého bez použití programátoru pomocí tzv. bootloaderu. Na konci paměti programu se totiž nachází bootovací sekce. Do ní lze nahrát program umožňující přepsání stávajících dat, které se nacházejí v paměti programu před bootovací sekcí. Pomocí tohoto programu pak umí mikrokontrolér přeprogramovat sám sebe. [30]

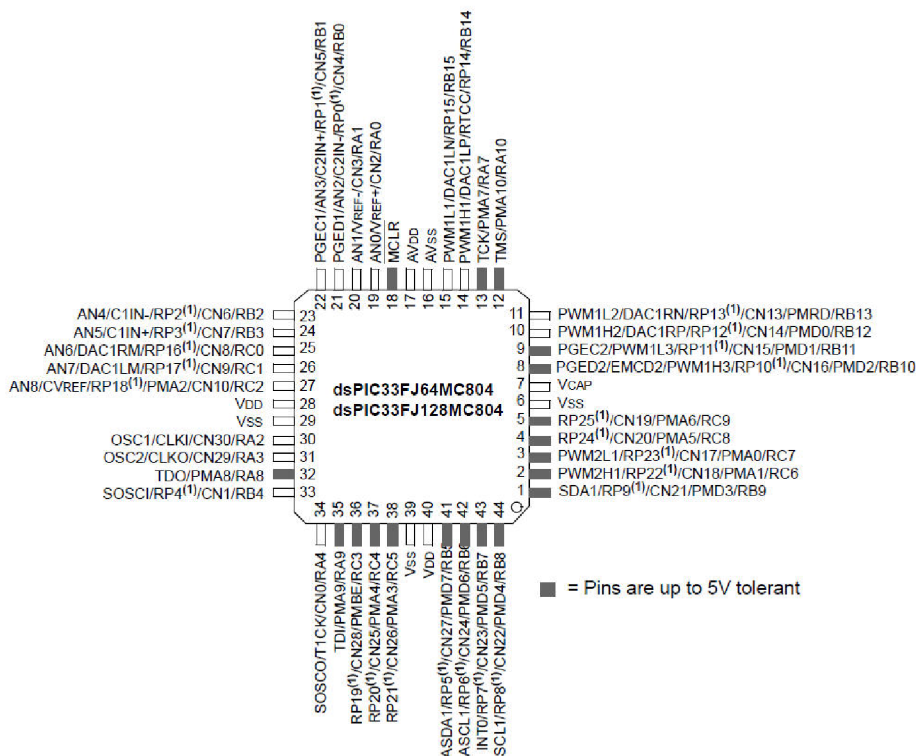
4.2.2 dsPIC33FJ128MC804

Mikrokontrolér dsPIC33FJ128MC804 je vyráběn firmou Microchip a patří do rodiny digitálních signálových mikrokontrolérů dsPIC[18]. Je to šestnáctibitový mikrokontrolér pracující stejně jako mikrokontrolér ATmega8 na architektuře RISC, ale oproti němu je napájen nižším napětím o velikosti 3V až 3,6V. Při maximální taktovací frekvenci může dosahovat výkonu až 40MIPS.

Mikrokontrolér má 35 vstupně-výstupních digitálních pinů, z nichž 26 je programovatelných. Instrukce programu jsou uchovávány v paměti typu FLASH o velikosti 128kB. Dále mikrokontrolér obsahuje paměť RAM, ve které jsou za chodu mikrokontroléru uchovávána data.

Součástí mikrokontroléru je také 5 šestnáctibitových čítačů a časovačů, 2 sériová rozhraní, 9 dvanáctibitových analogově digitálních převodníků, analogový komparátor, dvoukanálový šestnáctibitový digitálně analogový převodník pro audio výstup a šesti kanálová šestnáctibitová jednotka pro řízení motoru pomocí PWM signálu, pomocí níž je možno řídit i střídavé třífázové motory.

Oproti mikrokontroléru ATmega8, kde byly všechny přerušení na stejné úrovni, umožňuje tento mikrokontrolér nastavení priority přerušení až v sedmi úrovních.[31]



Obr. 3 Schéma vývodů mikrokontroléru dsPIC33FJ128MC804 v SMD provedení[31]

4.3 Jazyk C

Každý typ mikrokontroléru má svoji instrukční sadu neboli seznam strojových instrukcí, podle kterých řídí svoji činnost. Tyto instrukce se zapisují v jazyce symbolických adres neboli v assembleru. Je to programovací jazyk, ve kterém každý příkaz vyjadřuje právě jednu strojovou instrukci.

U každého typu mikrokontroléru se příkazy assembleru liší. Proto se nedají programy napsané v assembleru pro nějaký konkrétní typ mikrokontrolérů dobře přenášet na jiné typy mikrokontrolérů. Protože se tato bakalářská práce zabývá vytvářením obecného regulátoru, který by byl použitelný na více typů mikrokontrolérů, bylo potřeba zvolit k vytváření kódu pro mikrokontrolér univerzálnější programovací jazyk než je assembler. Na generování kódu regulátoru pro mikrokontrolér byl zvolen programovací jazyk C, protože je v oblasti programování mikrokontrolérů nejrozšířenější.

Programovací jazyk C je univerzální jazyk nízké úrovně. Není specializovaný na jednu určitou oblast využívání. Oproti assembleru je čitelnější a snáze přenositelný na jiné architektury. Vzhledem k jeho jednoduchosti je efektivita jeho kódu velká. Téměř se rovná efektivitě kódu v assembleru. Toto platí zvláště u mikrokontrolérů z rodiny AVR. Pro ně byla totiž instrukční sada vytvářena speciálně tak, aby ji kód psaný v jazyku C využil co nejefektivněji.

Mikrokontroléry AVR i dsPIC ovšem oproti výpočtům v simulaci hardwarově nepodporují práci s plovoucí desetinnou čárkou. Kdyby se naprogramovaly kódem pracujícím v této aritmetice, trval by výpočet velmi dlouhou dobu. Výše zmíněné mikrokontroléry podporují pouze celočíselnou aritmetiku, která bohužel na praktické aplikace většinou nestačí. Proto se u takovýchto aplikací používá aritmetika založená na pevné desetinné čárce, která umožňuje uchovat i desetinné číslo. Stačí, aby bylo v programu určeno, od kterého bitu číslo představuje hodnotu za desetinnou čárkou.

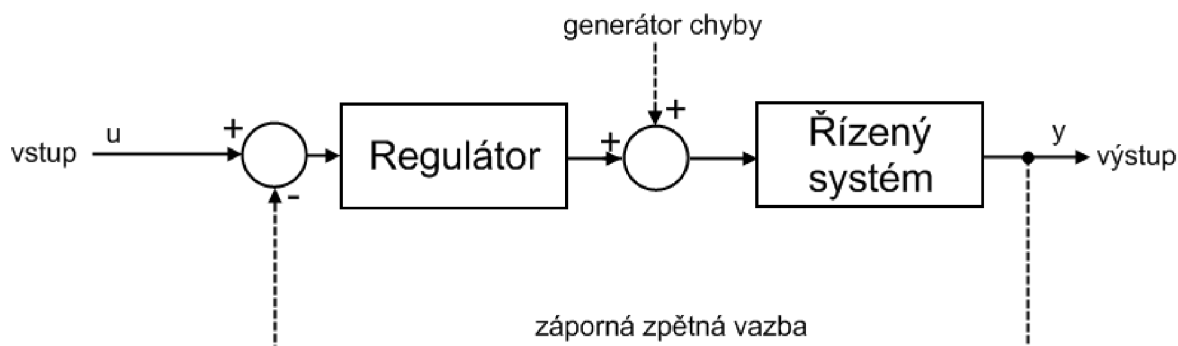
5 VLASTNÍ SIMULAČNÍ PROGRAM

5.1 Použití programu

Pomocí programu z této práce lze simulovat změnu průběhu signálu skrz soustavu znázorněnou na Obr. 4. Soustava obsahuje regulátor, řízený systém, zpětnou vazbu, kterou lze zapnout či vypnout, a generátor chyby. Přestože je tento systém jednoduchý, a jeho simulace neprobíhá v reálném čase, v mnoha použitích zcela postačuje.

Pokud má regulátor tvar diskrétní přenosové funkce či PID regulátoru, lze kód těchto funkcí vygenerovat přímo pro reálný mikrokontrolér ve formě C-kódu.

Dále lze blok regulátoru či řízeného systému nahradit komunikací po sériovém portu s libovolným zařízením, které přijímá číselné hodnoty typu float nebo integer a navrací zpět hodnoty přepočítané. Toto řešení lze použít například při práci s reálným regulátorem, pokud je potřeba zkontrolovat jeho správnou funkci.



Obr. 4 Schéma použití programu

Simulátor je naprogramován ve formě skriptu pro Python, tudíž je pro jeho použití nutné mít Python nainstalovaný.

5.2 Instalace programu

Simulační program se nachází ve dvou souborech. Jsou to soubory *Simulace.py* a *Win.py*. Tyto soubory jsou skripty programovacího jazyka Python (dále jen skripty), proto je pro jejich spuštění nutné mít na počítači nainstalovaný programovací jazyk Python verze 2.6[4]. Protože skripty využívají i funkce, které v pythonu standardně obsaženy nejsou, musí se také k němu doinstalovat knihovny NumPy[6], SciPy[10], MatPlotLib[15] a pySerial[17]. Tyto knihovny se dají volně stáhnout z internetu.

Pro využívání grafického prostření simulátoru, které se nachází v souboru *Win.py*, se musí doinstalovat také grafická knihovna wxPython[26].

Pokud je však model řízeného systému vytvořený pro Open Dynamics Engine, je také nutné mít v počítači nainstalovanou knihovnu pyODE[28].

Veškeré funkce simulátoru se nacházejí v souboru *Simulace.py*. Aby byly funkce z tohoto souboru přístupné z kteréhokoli místa na počítači a mohly se využívat ve všech vytvořených skriptech spouštějící simulaci, je třeba tento soubor nakopírovat do složky, ve které je programovací jazyk Python nainstalovaný, do podsložky „*.\Lib\site-packages*“ (např. „*C:\Program Files\Python265\Lib\site-packages*“).

Aby byly tyto funkce při vytváření vlastního skriptu dostupné, stačí soubor *Simulace.py* importovat:

```
from simulace import *
```

V souboru Win.py se nalézá grafické rozhraní usnadňující práci se simulátorem. Dají se zde nastavit všechny potřebné hodnoty pro simulaci a následně lze vygenerovat skript, který využívá funkce ze souboru *Simulace.py*. Spuštěním tohoto vygenerovaného skriptu simulace započne.

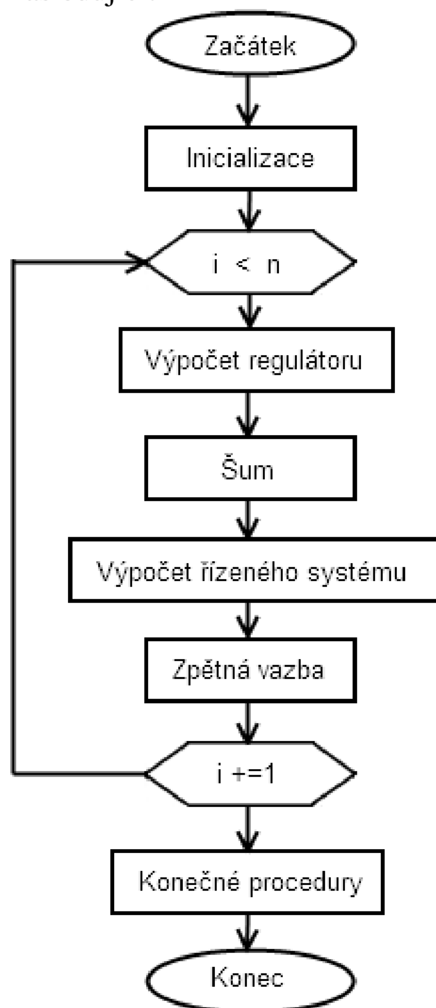
5.3 Popis programu

V souboru *Simulace.py* se nachází několik funkcí, které se mohou využít pro simulaci zadané soustavy. Pro lepší pochopení toho, jak simulátor funguje, je v této kapitole blíže rozebrána struktura programu.

5.3.1 Simulace

Vlastní simulaci soustavy zajišťuje funkce *Sim*. V ní lze nastavit pomocí jejích parametrů funkce jednotlivých bloků i celé soustavy. Je to např. časování bloků, zpětná vazba, či požadované vstupy. Každý z těchto parametrů je zde symbolizován jednou proměnnou (viz příloha kap. II.1.3).

Vnitřní struktura funkce je následující:



Obr. 5 Diagram funkce Sim

Na začátku probíhá inicializace. V té, pokud je to nutné, se upraví tvar vložené funkce regulátoru a řízeného systému. Dále se inicializují vstupní hodnoty a počáteční podmínky regulátoru i řízeného systému.

Po inicializaci následuje cyklus *For*, který se opakuje, dokud neproběhne simulace až do konce. Probíhají v něm postupně následující kroky. Provádí se výpočet výstupu regulátoru. K němu se může připočíst šum. Tato hodnota pak slouží jako vstup pro výpočet výstupu řízeného systému. Hodnota z tohoto výstupu pak může jít zpětnou vazbou na vstup.

Po proběhnutí celé simulace probíhající v cyklu *For* se provedou konečné procedury, které upravují výstupní hodnoty do správného formátu. Nakonec funkce *Sim* tyto hodnoty navrací.

5.3.2 Pole časů

Aby bylo možné měnit krok výpočtu, třeba jen v určitém úseku simulace, je funkce *Sim* řešena tak, že jeden z jejích parametrů je pole časů. Do něj lze nahrát posloupnost hodnot probíhajícího času simulace, kde rozdíl mezi po sobě jdoucími čísly je vždy takový, jaký je požadovaný krok.

Pokud tedy bude požadovaný krok simulace 0,01s, pak musí být hodnota každého následujícího čísla v poli časů o 0,01 vyšší než předchozí, např. [0,0; 0,01; 0,02; …]. Tato posloupnost čísel by pak trvala až do času ukončení simulace.

Ke zjednodušenému vytvoření pole časů slouží funkce *time*, která je více rozebrána v příloze v kapitole II.1.1.

5.3.3 Pole vstupů

Pro každou hodnotu v poli časů musí být také definována hodnota vstupu. Jelikož hodnoty vstupů mohou být závislé na čase, obsahuje program také funkce pro vytvoření průběhů vstupních hodnot v závislosti na již zmíněném poli časů. Funkce jsou schopny generovat obdélníkový, pilovitý a sinusový signál nebo jednotkový skok (viz příloha kap. II.1.2).

5.3.4 Vykreslení výsledků

Vzniklé pole hodnot, ať už vstupů či výstupů lze vykreslit v grafu v závislosti na poli časů pomocí funkce *plot*. Ta je více popsána v příloze v kapitole II.1.4.

5.3.5 Generování C-kódu

Pro libovolný PID regulátor nebo diskrétní přenosovou funkci lze vygenerovat univerzální podprogram v C-kódu pomocí funkce *GenerateC*. Popis této funkce naleznete v příloze v kapitole II.1.5. Vygenerování pouze podprogramu regulátoru, a ne kompletního programu pro mikrokontrolér, je voleno z toho důvodu, že mikrokontrolér může využívat k řízení jiné vstupní či výstupní periferie a porty, nebo může používat rozdílné názvy nastavovacích registrů. Vygenerovaný podprogram se pak musí přidat do ručně vytvořeného celkového programu pro používaný mikrokontrolér.

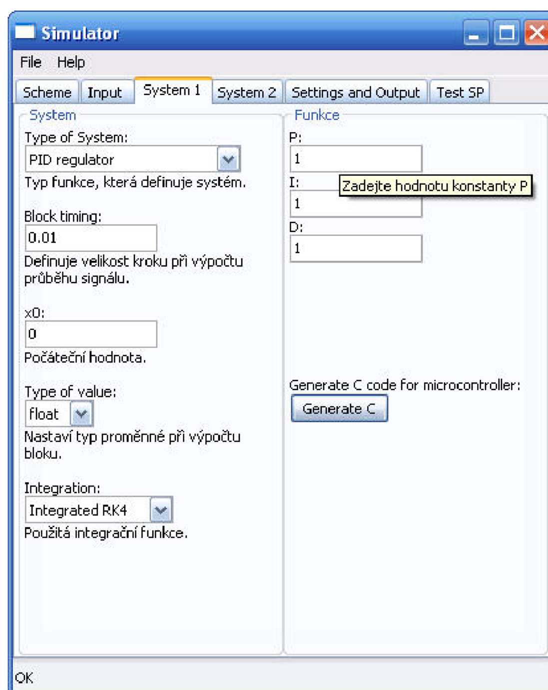
5.3.6 Řízení mikrokontrolérem a jeho kontrola pomocí sériového portu

Pomocí sériového portu lze zjistit z již naprogramovaného mikrokontroléru průběh výstupních hodnot nebo do něj naopak posílat požadované vstupní hodnoty. Komunikace s mikrokontrolérem probíhá v reálném čase pomocí funkce *status*, jak je blíže popsáno v příloze v kapitole II.1.6.

5.4 Grafické rozhraní

Aby se zjednodušila práce se simulátorem, to znamená, aby se nemusel zdlouhavě vypisovat programovací kód a pamatovat si všechny typy proměnných, byla pro něj vytvořena grafická nadstavba. Ta umí potřebný programovací kód pro simulaci sama vygenerovat. Rozhraní ovšem umí vygenerovat pouze základní funkce. Proto při složitějších simulacích, například při vytváření specifického pole vstupů, či při vytváření řízeného modelu v ODE je nutné upravit vygenerovaný skript ručně.

Grafické rozhraní se nachází v souboru *Win.py*.



Obr. 6 Grafické rozhraní simulátoru

5.4.1 Struktura grafického rozhraní

Po spuštění skriptu *Win.py* je vidět na vrchní liště menu. Pod ním se nachází 6 záložek, ve kterých jsou přehledně uspořádány položky pro nastavení simulace (viz Obr. 6). U každé položky, pokud se na ni najede kurzorem myši, se zobrazí nápověda. Ta podrobněji popisuje, co která položka symbolizuje a jakou hodnotu je do ní třeba zadat.

Menu

Obsahuje několik položek:

- **Load values**

  Slouží k načtení hodnot všech nastavených parametrů potřebných pro simulaci. Načítá tedy hodnoty do všech parametrů, které se nachází v záložkách *Input*, *System 1*, *System 2* a *Settings and Output*, z textového souboru.

- **Save values**

  Slouží k uložení hodnot všech nastavených parametrů potřebných pro simulaci do textového souboru.

- **Generate Python file**

  Vygeneruje skript, pomocí kterého lze spustit simulaci se zadanými parametry.

- **Exit**

  Ukončí program.

Záložky

Parametry simulace jsou rozděleny do šesti záložek:

- **Scheme**

  V první záložce se nachází schéma simulátoru s popisem, které napovídá, v kterých záložkách se nachází jednotlivé potřebné parametry pro simulaci.

- **Input**

  Tato záložka slouží k vytvoření vstupních hodnot pro simulaci. V položce *Type of input signal* je možno vybrat typ generovaného vstupního signálu. Je zde na výběr buď z automaticky generovaných signálů typu skok, sinusovka či pulz, nebo ručně vytvořený signál.

  Při vybraném signálu typu skok lze nastavit, v jakém čase se skok provede a jaká bude velikost signálu po skoku i před ním.

  U sinusového signálu lze nastavit jeho periodu, velikost amplitudy a posunutí signálu v časové ose udávaný v sekundách.

  U pulzního signálu lze nastavit periodu, po které se bude impulz znovu spouštět, maximální velikost signálu impulsu, posun v časové ose a délku pulzu. Kromě maximální velikosti signálu se vše udává v sekundách. Dále lze pomocí políčka *Saw* přepínat, zda bude mít pulz obdélníkový či trojúhelníkový tvar.

  U ručně vytvořeného signálu stačí zvolit za typ vstupního signálu *From file*, a dále už jen zadat do pole *Name of variable* název vytvořené proměnné obsahující vstupní signál. Proměnná se musí nacházet v souboru „data.py“.

  V položce *Type of variable* se vybírá číselný typ vstupního signálu. Jsou na výběr celá čísla *int* nebo čísla s plovoucí desetinnou čárkou *float*.

- **System 1, 2**

  V těchto záložkách se nastavuje, jakého typu budou funkce prvního i druhého bloku celé soustavy a jejich příslušné parametry (viz Obr. 6).

  V levém sloupci *System* můžeme nastavit typ bloku, u kterého se vybírá z následujících možností: přenosová funkce, stavový systém, diskrétní přenosová funkce, PID regulátor, sériová komunikace s jiným zařízením, ručně vytvořený model v ODE či ručně vytvořená nelineární funkce z externího souboru. Ručně vytvořené funkce se musejí nacházet v souboru s názvem „daty.py“.

  Dále se tu lze nastavit časování bloku, které značí, po jakém časovém kroku bude pro tento blok prováděn výpočet. Čím menší časový krok, tím přesnější bude výpočet, ale na druhou stranu bude výpočet trvat déle.

  V položce *x0* lze nastavit počáteční hodnotu (nebo i více hodnot oddělených čárkou), do které se blok na začátku simulace nastaví.

  V položce *Type of value* se vybírá číselný typ, se kterým bude výpočet tohoto bloku probíhat. Jsou na výběr celá čísla *int* nebo čísla s plovoucí desetinnou čárkou *float*.

  Nevyhovuje-li integrovaná funkce pro výpočet diferenciálních rovnic metodou Runge-Kutta 4, může být v poli *Integration* nastavena externí ručně vytvořená funkce. Ta musí uložená v souboru s názvem „data.py“. Pak je samozřejmě nutné v dalším poli zadat název této funkce.

  V pravém sloupci s názvem *Function* volíme parametry zvoleného typu funkce reprezentující blok. Je-li zvolen PID regulátor nebo diskrétní stavová funkce, objeví se v záložce tlačítko umožňující pro tuto konkrétní funkci vygenerovat C-kód. Ten je pak možné přidat do hlavního programu v jazyku C vytvořeného pro mikrokontrolér.

- **Settings and Output**

  V této záložce lze nastavit zbylé parametry simulace.

  Je to čas jejího trvání, který se zapisuje v sekundách.

  Dále je to vypnutí či zapnutí zpětné vazby. Při zapnutí se od hodnot ze vstupu odečítají hodnoty z výstupu a tento rozdíl je vstupem do prvního bloku.

  Lze také zapnout šum, který se dostává mezi první a druhý blok a ruší mezi nimi signál. V polích *Max. Value* a *Min. Value* lze nastavit jeho mezní hodnoty.

  Nakonec se v této záložce také nalézá tlačítko, které vygeneruje simulační skript a spustí samotnou simulaci.

- **Test SP**

  Záložka *Test SP* již nesouvisí se simulátorem. Je to samostatné rozhraní, které slouží pro kontrolu a načítání dat z již naprogramovaného regulátoru v mikrokontroléru po sériové lince. Pro jeho nastavení slouží několik parametrů. První z nich je čas, po který probíhá načítání dat z mikrokontroléru. Dále to je časový interval, v kterém se bude načítání hodnot opakovat. Oba dva parametry se dosazují v sekundách. Třetí parametr je název sériového portu, přes které bude komunikace probíhat. Dále lze nastavit, jaké hodnoty se budou z mikrokontroléru načítat. Mohou to být buď jen výstupní hodnoty, rozdíl vstupních a výstupních hodnot, nebo obě tyto hodnoty.

  Do mikrokontroléru se dá také zároveň vysílat průběh požadovaných vstupních hodnot. Na vytvoření tohoto průběhu používá rozhraní společně se simulátorem záložku *Input*, jejíž popis byl uveden výše.

  Aby se usnadnilo ladění naprogramovaného PID regulátoru v mikrokontroléru a nemusel se při každé změně PID konstant mikrokontrolér přeprogramovávat, je v této záložce možné tyto konstanty upravit. Nové konstanty se vyšlou do mikrokontroléru hned na začátku komunikace před zahájením načítání dat.

  Nakonec se v této záložce vyskytuje tlačítko *Start*, které vygeneruje skript, který se automaticky spustí. Spuštěním skriptu začne postupné načítání dat z mikrokontroléru. Po uplynutí nastavené doby shromažďování dat se hodnoty v závislosti na čase vykreslí do grafu.

5.5 Příklady simulace na počítači

Pro lepší pochopení funkcí simulátoru je zde na několika příkladech podrobněji vysvětleno, jakým způsobem se simulátorem pracovat a jaké funkce v daných chvílích použít. Nacházejí se zde příklady použití různých typů bloků nebo příklady řízení modelů vytvořených v různých formátech.

5.5.1 Průběh signálu přes diskrétní regulátor se vstupem jednotkového skoku

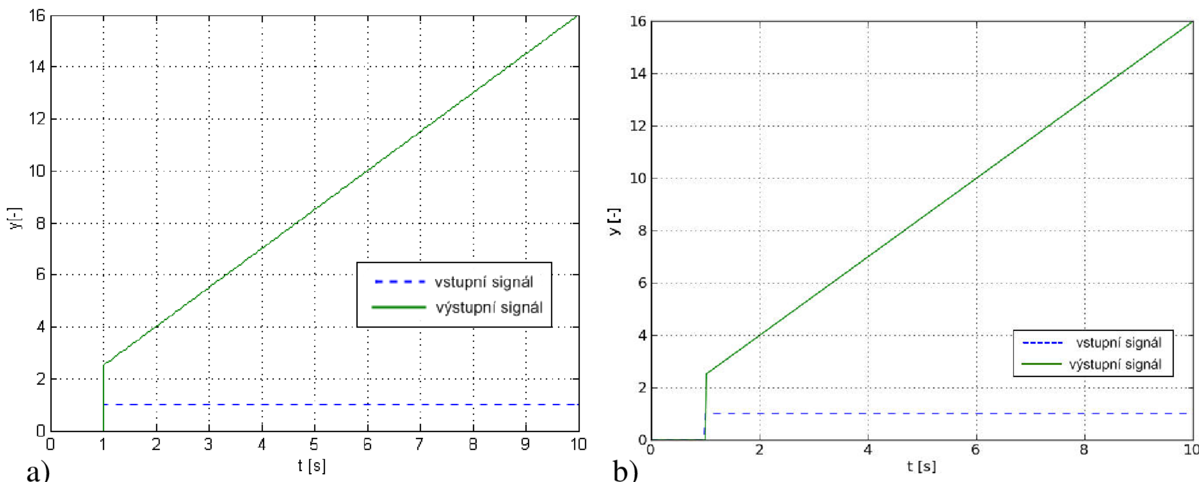
Pokud je potřeba simulovat jednotkový skok přes diskrétní regulátor s přenosovou funkcí

$$F(z) = \frac{2{,}515 - 2{,}5z^{-1}}{1 - 1z^{-1}}, \tag{1}$$

je nutné nejprve vytvořit pole času, například s výpočetním krokem 0,01s do času 10s. Dále se vytvoří pole vstupů pro tyto časy s jednotkovým skokem v čase 1s. Nyní je na řadě zapsat přenosovou funkci regulátoru do formátu pro jazyk Python a vložit ji do funkce *Sim* do parametru prvního bloku. Protože je potřeba odsimulovat jen průběh signálu přes tento regulátor, do druhého bločku pro řízený systém zapíšeme funkci jednotkového přenosu. Tato funkce nám zaručí, že na výstupu z druhého bločku se objeví stejné hodnoty, jako na jeho vstupu. Nakonec už zbývá jen pomocí funkce *Sim* vypočítat průběh signálu v soustavě a příkazem *plot* signál vykreslit. Na Obr. 7 je vidět porovnání výpočtu v Simulinku a v tomto simulátoru.

Kód v Pythonu:

```
h = 0.01                          # nastavení výpočetního kroku 0.01s
tmax = 10                         # max. čas, do kterého trvá výpočet
t = time(tmax, h)                 # vytvoření pole času
u = step(t, 1)                    # vytvoření pole vstupů s jednotkovým
                                  #  skokem v čase 1s

TF1 = [[2.515, -2.5], [1, -1],‘z‘]     # přenos přes regulátor
TF2 = [[1], [1]]                       # přenos přes systém

y = Sim(TF1, TF2, t, u)                # výpočet výstupu
plot(t,y)                              # vykreslení grafu
```



Obr. 7 Průběh signálu přes regulátor se vstupem jednotkového skoku. a) V Simulinku. b) V tomto simulátoru.

5.5.2 Průběh signálu přes řízený systém se vstupem jednotkového skoku

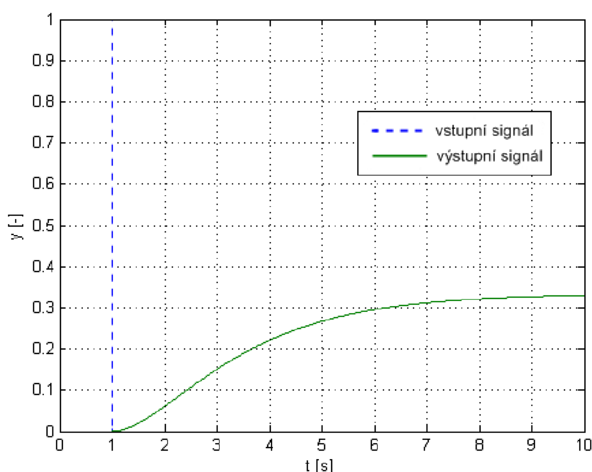
Je-li potřeba simulovat jednotkový skok přes řízený systém, který má přenosovou funkci

$$F(p) = \frac{1}{5p^2 + 8p + 3}, \quad (2)$$

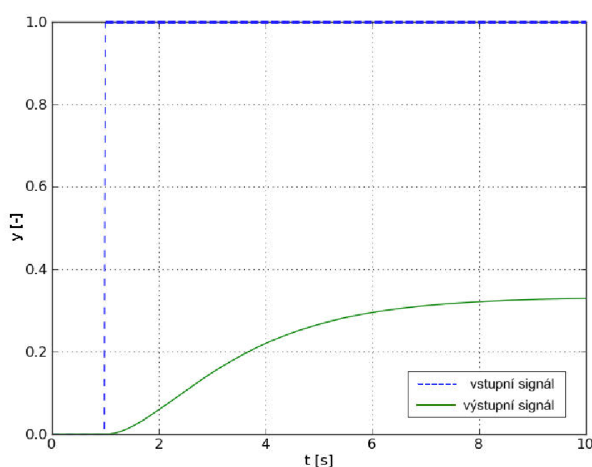
postupuje se stejným způsobem jako v předcházejícím případě. Pouze za přenos regulátoru se dosadí jednotkový přenos a za přenos řízeného systému se dosadí převedená přenosová funkce ve formátu pro jazyk Python. Na Obr. 8 je vidět porovnání výpočtu v Simulinku a v tomto simulátoru.

Kód v Pythonu:

```
TF1 = [[1],[1]]                       # přenos přes regulátor
TF2 = [[1],[5, 8, 3]]                 # přenos přes systém

y = Sim(TF1, TF2, t, u)               # výpočet výstupu
plot(t, y)                            # vykreslení grafu
```



a)



b)

Obr. 8 Průběh signálu přes řízený systém se vstupem jednotkového skoku. a) V Simulinku. b) V tomto simulátoru.

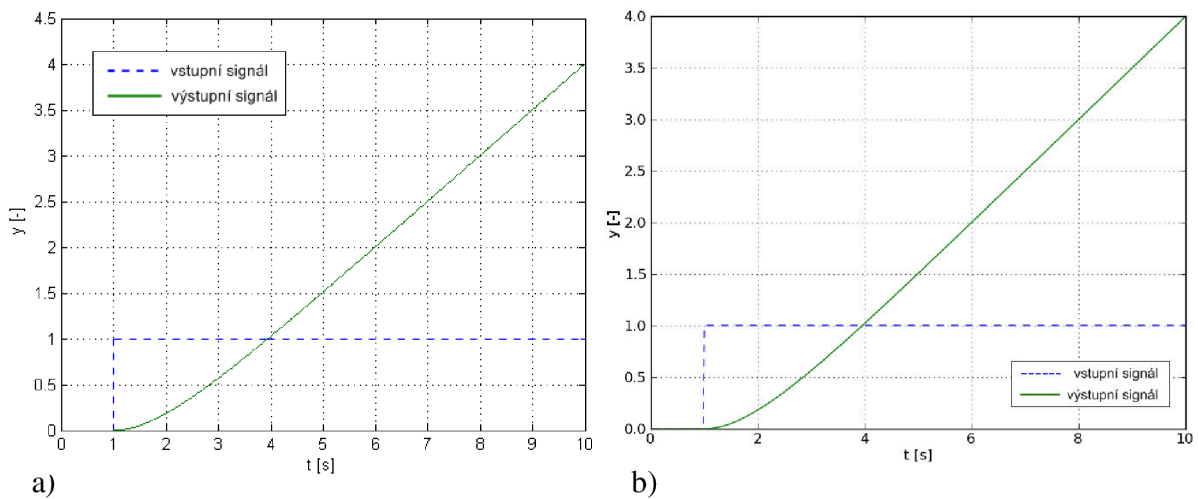
5.5.3 Průběh signálu soustavou se vstupem jednotkového skoku bez zpětné vazby

Pokud je potřeba simulovat přenos celého tohoto systému obsahující diskrétní regulátor z příkladu 5.5.1 a řízený systém tvaru přenosové funkce z příkladu 5.5.2 bez zpětné vazby, stačí tyto bloky systému dosadit opět do parametrů funkce *Sim*.

Na Obr. 9 je vidět porovnání výpočtu v Simulinku a v tomto simulátoru.

Kód v Pythonu:

```
Tf1 = [[2.515, -2.5], [1, -1],‘z‘] # přenos přes regulátor
Tf2 = [[1],[5, 8, 3]]               # přenos přes systém

y = Sim(Tf1, Tf2, t, u)             # výpočet výstupu
plot(t, y)                          # vykreslení grafu
```



Obr. 9 Průběh signálu soustavou se vstupem jednotkového skoku bez zpětné vazby.
a) V Simulinku. b) V tomto simulátoru.

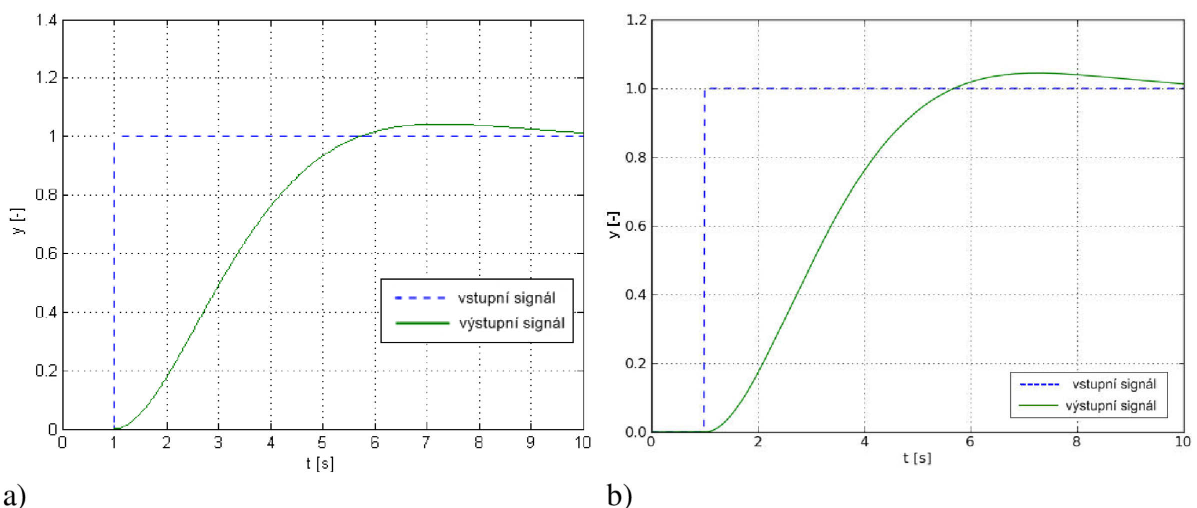
5.5.4 Průběh signálu soustavou se vstupem jednotkového skoku se zpětnou vazbou

Pokud je potřeba simulovat přenos celého tohoto systému se zpětnou vazbou, postupuje se stejně jako v předcházejícím případě 5.5.3. Pouze u funkce *Sim* se musí zapnout zpětná vazba tím, že se do pátého parametru funkce zadá hodnota *True*. Na

Obr. 10 je vidět porovnání výpočtu v Simulinku a v tomto simulátoru.

Kód v Pythonu:

```
Tf1 = [[2.515, -2.5], [1, -1],‘z‘]  # přenos přes regulátor
Tf2 = [[1],[5, 8, 3]]                # přenos přes systém

y = Sim(Tf1, Tf2, t, u, True)        # výpočet výstupu
plot(t,y)                            # vykreslení grafu
```



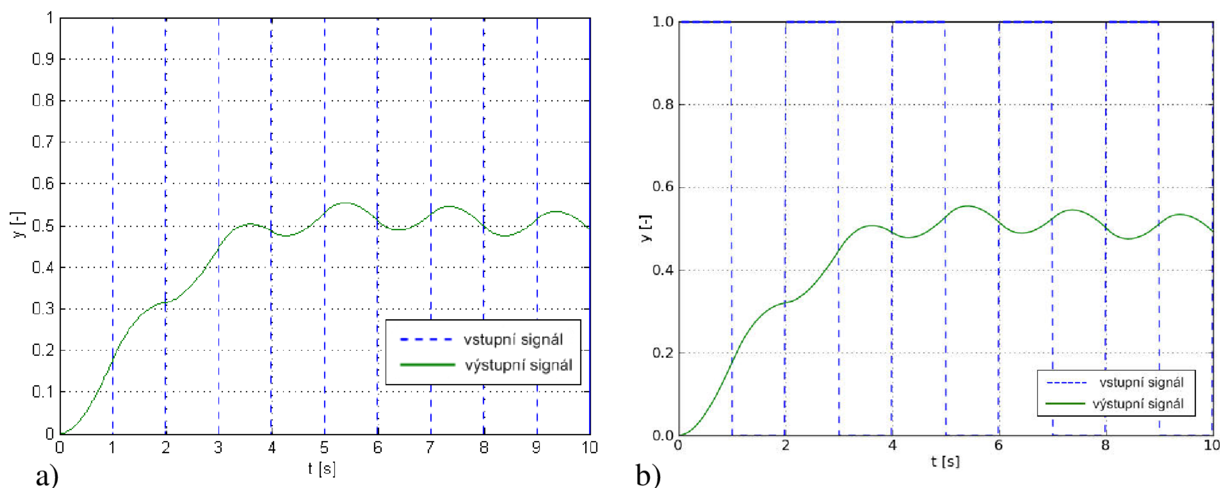
Obr. 10 Průběh signálu soustavou se vstupem jednotkového skoku se zpětnou vazbou. a) V Simulinku.b) V tomto simulátoru.

5.5.5 Průběh signálu soustavou se vstupem ve tvaru obdélníka se zpětnou vazbou

Pokud je potřeba simulovat jiný typ vstupního signálu než jednotkový skok, pak lze použít například obdélníkový průběh s nastavitelnou periodou i délkou sepnutého stavu. V tomto případě stačí místo příkazu *step* použít příkaz *pulse*. Ostatní příkazy zůstávají stejné jako v předchozím případě. Na Obr. 11 je vidět porovnání výpočtu v Simulinku a tomto simulátoru.

Kód v Pythonu:

```
u = pulse(t, 2, 1)      # vytvoření pole vstupů pulzním signálem
                        # s periodou 2s a délkou sepnutého stavu 1s
```



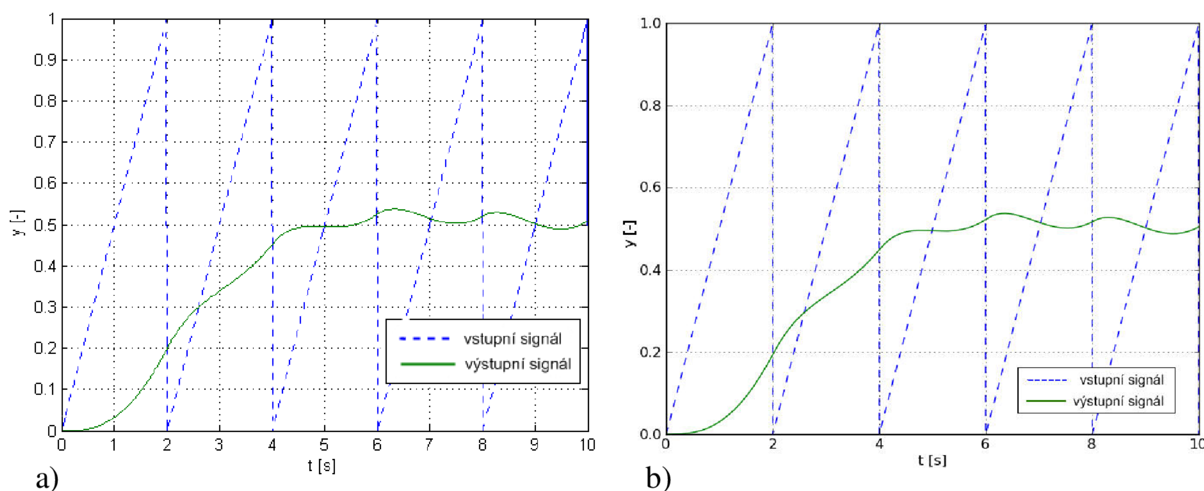
Obr. 11 Průběh signálu soustavou se vstupem ve tvaru obdélníka se zpětnou vazbou. a) V Simulinku. b) V tomto simulátoru.

5.5.6 Průběh signálu soustavou se vstupem ve tvaru pily se zpětnou vazbou

Další možnost je použít generování signálu ve tvaru pily. Toho lze dosáhnout tak, že u příkazu *pulse* zapneme ve čtvrtém parametru volbu *saw* pomocí hodnoty *True*. Na Obr. 12 je vidět porovnání výpočtu v Simulinku a v tomto simulátoru.

Kód v Pythonu:

```
u = pulse(t,2,2,True) # vytvoření pole vstupů s pilovým signálem
                      # s periodou 2s a délkou sepnutého stavu 2s
```



Obr. 12 Průběh signálu soustavou se vstupem ve tvaru pily se zpětnou vazbou. a) V Simulinku. b) V tomto simulátoru.

5.5.7 Simulace nelineární soustavy - kyvadla

V tomto programu lze také simulovat nelineární soustavu, například matematické kyvadlo, které popisuje rovnice

$$\ddot{\varphi} = -\frac{g}{l}\sin\varphi + u\,, \tag{3}$$

kde $\ddot{\varphi}$ je úhlové zrychlení, φ je úhel natočení, g je tíhové zrychlení, l je délka kyvadla a u je vstup.

Tato rovnice se dá snadno přepsat do formátu funkce pro jazyk Python. Za počáteční podmínky kyvadla se může považovat počáteční výchylka φ kyvadla a jeho rychlost $\dot{\varphi}$. Pro názornost můžeme použít tyto hodnoty počátečních podmínek:

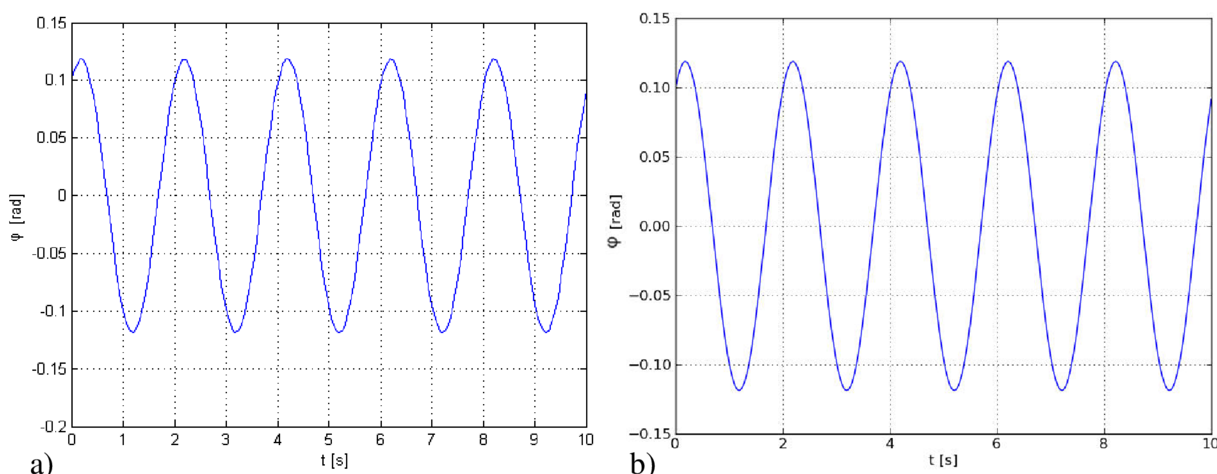
$$\varphi = 0{,}1 rad\,, \tag{4}$$

$$\dot{\varphi} = 0{,}2 rad \cdot s^{-1}. \tag{5}$$

Na Obr. 13 je vidět porovnání výpočtu v Simulinku a v tomto simulátoru.

Kód v Pythonu:

```
def kyvadlo(x,t,u):                      # definice funkce
    dx = [0.0] * 2                       # vytvoření výstupního vektoru

    g = 9.81                             # zadání konstant
    l = 1

    dx[0] = x[1]                         # zápis rovnic
    dx[1] = (-g/l) * sin(x[0]) + u[0]
    return dx

x0 = [[0.0], [0.1, 0.2]]                 # zadání počátečních podmínek
TF = [[1], [1]]                          # jednotkový přenos přes regulátor
t = time(10, 0.01)                       # vytvoření pole času
u = [0.0] * len(t)                       # vytvoření pole vstupů
y = Sim(TF,kyvadlo,t,u,False, x0)        # výpočet průběhu signálu
plot(t,y)                                # vykreslení grafu
```



Obr. 13 Simulace kmitání kyvadla. a) V Simulinku. b) V tomto simulátoru.

5.5.8 Návrh řízení kyvadla

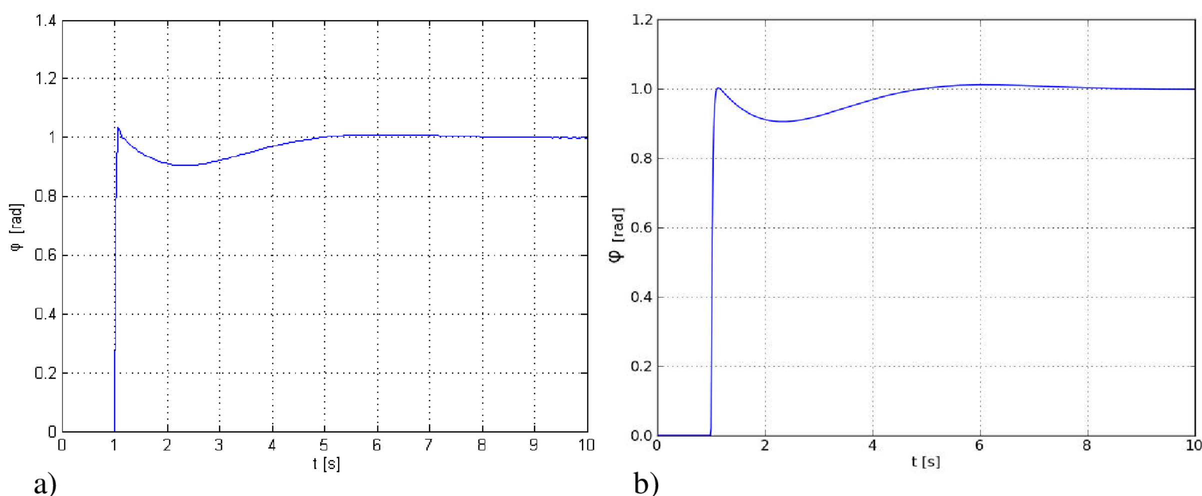
Pro kyvadlo z minulého příkladu 5.5.7 lze navrhnout regulátor pomocí PID prvku. Stačí do parametru regulátoru ve funkci *Sim* dosadit pole se třemi prvky, které odpovídají konstantám proporcionálního, integračního a derivačního členu PID regulátoru. Na Obr. 14 je pak znázorněno porovnání řízení jednotkového skoku natočení kyvadla v Simulinku a tomto simulátoru s PID regulátorem, který má následující složky:

$$K_p = 40,\ K_i = 40,\ K_d = 40, \tag{6}$$

kde, K_p je konstanta proporcionálního členu, K_i konstanta integračního členu a K_d konstanta derivačního členu.

Kód v Pythonu:

```
x0 = [[0.0], [0.0, 0.0]]               # zadání počátečních podmínek
PID = [40.0, 40.0, 40.0]               # zadání PID regulátoru

t = time(10, 0.01)                     # vytvoření pole času
u = step(t, 1)                         # vytvoření pole vstupů

y = Sim(PID,kyvadlo,t,u,True,x0)       # výpočet průběhu signálu
plot(t,y)                              # vykreslení grafu
```



Obr. 14 Simulace řízení kyvadla. a) V Simulinku. b) V tomto simulátoru.

5.5.9 Návrh řízení modelu kyvadla vytvořeného v ODE

Ten samý model kyvadla z příkladu 5.5.7 lze navrhnout i pomocí ODE. Aby však simulátor s tímto modelem dokázal pracovat, musí být model rozdělen do dvou funkcí.

V první funkci se musí vytvořit dynamický svět a těleso modelu, vytvořit vazby a nastavit počáteční podmínky. Pro nastavení počátečních podmínek slouží jediný parametr, který funkce obsahuje. V tomto případě se jedná o vytvořenou funkci *ODE_init* s parametrem počátečních podmínek *x0*.

Druhá funkce pak musí mít 2 vstupní parametry. První z nich je velikost časového kroku *dt*, pro který bude simulace probíhat. Druhý parametr je aktuální vstupní hodnota do systému modelu. Velikost časového kroku se zadává do příkazu k výpočtu jednoho kroku ve světě ODE – *world.step(dt)*. Tento příkaz musí funkce obsahovat též. Aby byla simulace úspěšná, musí funkce také navracet požadované výstupní hodnoty.

Nakonec už jen stačí do funkce *Sim* odeslat proměnnou typu pole o dvou prvcích, ve kterých se budou nacházet odkazy na výše jmenované funkce.

Příklad vytvoření inicializační funkce ODE na modelu kyvadla

Při vytváření modelu se postupuje následovně. Nejprve se vytvoří svět přiřazením třídy *ode.world()* do proměnné *world*. Pak se v něm nastaví velikost gravitačního zrychlení v ose y pomocí funkce *setGravity*.

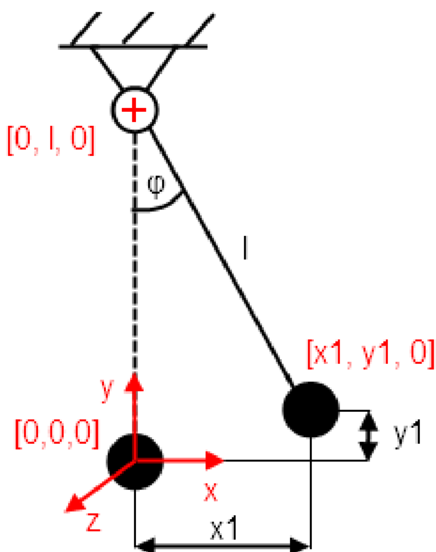
Jakmile je vytvořený svět, tak se do něj mohou začít přidávat tělesa, ze kterých se sestaví model. Na kyvadlo stačí jen jedno těleso, kterým je hmotný bod. Vytvoří se přiřazením třídy *ode.body(world)* do proměnné *body*. Tím se hmotný bod objeví na počátečních souřadnicích x, y, z = [0, 0, 0] souřadné soustavy, která je naznačena na Obr. 15.

Aby vzniklo celé kyvadlo, musí se spojit vytvořený hmotný bod s dynamickým světem pomocí rotační vazby. Rotační vazba se vytvoří přiřazením třídy *ode.HingeJoint(world)* do proměnné *joint* a umístí se nad hmotný bod ve vzdálenosti délky kyvadla l. V souřadné soustavě naznačené na Obr. 15 bude vazba umístěna do bodu [0, l, 0] pomocí funkce *setAnchor*. Po vytvoření rotační vazby se musí zvolit osa kolem které bude vazba rotovat pomocí funkce *setAxis*. V našem případě to je osa z.

Na závěr inicializační funkce se ještě uvede kyvadlo podle počátečních podmínek *x0* do správné polohy úhlu natočení φ a nastaví se počáteční úhlová rychlost $\dot{\varphi}$.

Ač by se zdálo na nastavení pozice lepší využít funkci *setRotation*, která těleso o určitý úhel pootočí, není tomu tak. Tato funkce totiž nenastaví polohu hned, ale způsobí, že těleso začne postupně rotovat až při vyvolání dalšího kroku výpočtu ve světě *world*. Proto se musí pomocí goniometrických funkcí přepočítat natočení kyvadla na pevné souřadnice, jak je naznačeno v rovnicích (7) a použít funkci *setPosition*.

Nastavování úhlové rychlosti kyvadla je daleko jednodušší. Vektor úhlové rychlosti kyvadla stačí nastavit pomocí *setAngularVel*.



Obr. 15 Model kyvadla s vyznačenými souřadnicemi ve světě ODE.

$$x_1 = l \cdot \sin\varphi$$
$$y_1 = l - l \cdot \cos\varphi \qquad (7)$$

Programový kód inicializační funkce:

```
def ODE_init(x0):                        # název inicializační funkce
   global world, body, joint, l          # použití globálních proměnných

   l = 1                                 # nastavení délky kyvadla - 1m
   v = x0[1]*l                           # výpočet počáteční rychlosti

   # ---------- Svět ----------
   world = ode.World()                   # vytvoření světa
   world.setGravity((0,-9.81,0))         # nastavení gravitačního pole

   # ---------- Kyvadlo ----------
   body = ode.Body(world)                # vytvoření tělesa kyvadla

   # ---------- Vazba ----------
   joint = ode.HingeJoint(world)        # vytvoř. rotační vazby ve světě
   joint.attach(body, ode.environment)# připoj. vazby k tělesu kyvadla
   joint.setAnchor( (0,l,0) )           # nastav. souřadnic polohy vazby
   joint.setAxis( (0,0,1) )             # zvolení osy rotace z

   # ---------- Zadání počátečních podmínek ----------
   body.setPosition((l*sin(x0[0]),l-l*cos(x0[0]),0)) # poč. souřadnice
   body.setAngularVel((0,0,x0[1]))       # poč. úhlová rychlost
```

Příklad vytvoření simulační funkce modelu ODE

V simulační funkci se už jen využívá vytvořený model ODE předchozí funkcí.

Na začátku funkce lze pomocí vstupních hodnot libovolně nastavovat různé parametry vytvořeného modelu. V tomto případě to je moment, který na těleso kyvadla působí. Všechny momenty a síly se před začátkem každého kroku musí nastavovat znovu, protože se po dokončení výpočtu hned odstraní. Proto je na začátek funkce umístěn příkaz *addTorgue*, který umístí na těleso kyvadla moment o velikosti vstupní hodnoty.

Po nastavení všech potřebných hodnot lze vyvolat další výpočtový krok simulace světa *world* příkazem *world.step(dt)*.

V posledním kroku musí vytvořená simulační funkce navracet novou vypočítanou hodnotu. V tomto příkladu simulace kyvadla stačí vracet hodnotu natočení, která se musí pomocí goniometrických funkcí přepočítat ze souřadného systému zpět na úhel. Při tomto výpočtu se ovšem vyskytl problém. K zpětnému výpočtu úhlu bylo zapotřebí znát délku kyvadla. Ta se ale vlivem odstředivé síly od původně zadané délky prodloužila. Proto bylo zapotřebí pomocí Pythagorovy věty vypočítat skutečnou hodnotu délky kyvadla a uložit do proměnné *lu*.

Programový kód simulační funkce:

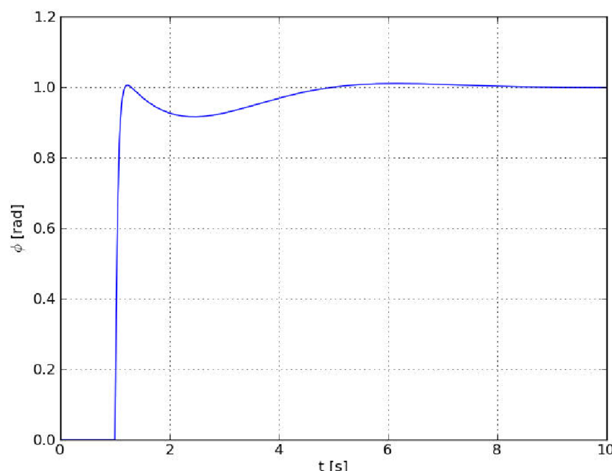
```
def ODE_sim(dt, u):
   global world, body, join, l

   body.addTorque((0,0,u))              # přid. momentu na těleso kyvadla

   world.step(dt)                       # další krok simulace
   x1,y1,z1 = body.getPosition()        # zjištění pozice

   lu = sqrt(x1*x1 + (l-y1)*(l-y1))     # výpoč. skutečné délky kyvadla
   y = asin(x1/lu)                      # výpočet úhlu natočení kyvadla
   return [y]                           # navrácení úhlu natočení kyvadla
```

Pro názornost příkladu můžeme použít k simulaci stejný PID regulátor, který je použit i v příkladu 5.5.8.

Příklad simulace PID regulátoru řídící model ODE:

```
x0 = [[0.0], [0.0, 0.0]]                   # zadání počátečních podmínek
PID = [40.0, 40.0, 40.0]                   # zadání PID regulátoru

t = time(10, 0.01)                         # vytvoření pole času
u = step(t, 1)                             # vytvoření pole vstupů
model = [ODE_init, ODE_sim]                # odkazy na ODE model

y = Sim(PID, model, t, u, True, x0)        # výpočet průběhu signálu
plot(t,y)                                  # vykreslení grafu
```

Pokud porovnáme výsledek řízení tohoto ODE modelu kyvadla znázorněný na Obr. 16 a výsledek řízení toho samého modelu kyvadla vyjádřeného pomocí diferenciálních rovnic a znázorněného na Obr. 14, zjistíme, že jsou oba dva průběhy téměř shodné.



Obr. 16 Simulace řízení modelu kyvadla v ODE.

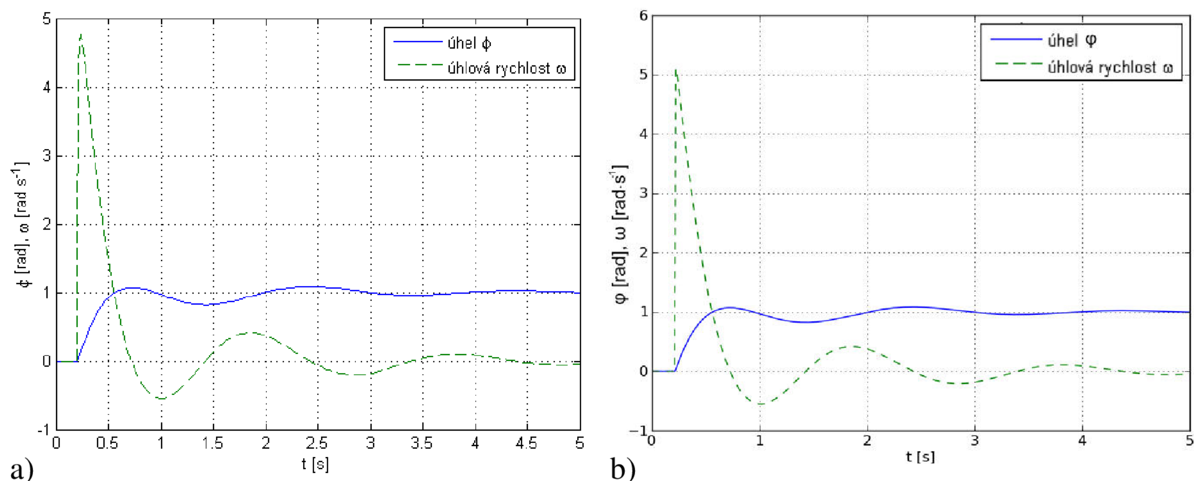
5.5.10 Zobrazení více výstupních hodnot

Další výhodou, kterou tento program nabízí je možnost dvoukanálového zobrazení výstupů. Například u kyvadla z příkladu 5.5.7 je možné do jednoho grafu současně zobrazit okamžitou polohu i tomu odpovídající okamžitou rychlost pohybujícího se kyvadla. Stačí vytvořit dvoukanálový signál pomocí funkce *step*, čímž celá soustava bude pracovat v dvoukanálovém režimu a na výstupu se pak objeví obě potřebné veličiny. Na Obr. 17 je znázorněna poloha a rychlost kyvadla při reakci na jednotkový skok nastávající v čase 0,2s s PID regulátorem, který má následující složky:

$$K_p = 10,\ K_i = 40,\ K_d = 5. \tag{8}$$

Kód v Pythonu:

```
x0 = [[0.0], [0.0, 0.0]]            # zadání počátečních podmínek
PID = [10.0, 40.0, 5.0]             # zadání PID regulátoru

t = time(10, 0.01)                  # vytvoření pole času

u = step(t, [0.2 ,10])              # vytvoření dvoukanálového vstupu
y = Sim(PID,kyvadlo,t,u,True,x0)    # výpočet průběhu signálu
plot(t, y)                          # vykreslení grafu
```



Obr. 17 Výstup dvou signálů z kyvadla. a) V Simulinku. b) V tomto simulátoru.

5.5.11 Rozběh stejnosměrného motoru

Dynamika stejnosměrného motoru je zachycena dvěmi diferenciálními rovnicemi (9) a (10), které vycházejí z rovnic pro elektrickou rovnováhu kotvy a mechanickou rovnováhu momentů na hřídeli:

$$i' = \frac{1}{L}(U - Ri - k_m \omega), \tag{9}$$

$$\omega' = \frac{1}{J}(k_m i - M_0), \tag{10}$$

kde U je napájecí napětí motoru, i je proud procházející kotvou, R odpor vinutí kotvy, L vlastní indukčnost kotvy, M_0 zatěžovací moment, J moment setrvačnosti motoru, ω úhlová rychlost motoru a k_m je mechanická konstanta motoru.

Aby byl na výstupu vidět průběh proudu i úhlové rychlosti, stačí pomocí příkazu *step* vytvořit pole dvou vstupů. Za první vstup se může zvolit napětí U a za druhý vstup zátěžný moment M_0. Následně je potřeba upravit rovnice (9) a (10) pro zápis funkce v jazyku Python. Simulace rozběhu motoru (viz Obr. 18) proběhla s následujícími parametry a počátečními podmínkami:

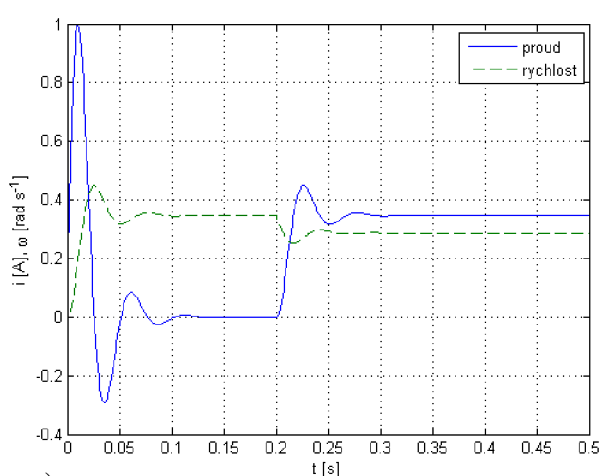
$$R = 0{,}5\Omega,\ L = 0{,}005H,\ k_m = 2{,}88,\ J = 0{,}1kgm^2,\ i_0 = 0A,\ \omega_0 = 0. \tag{11}$$

V čase $t = 0s$ bylo na vstup 1 přivedeno napětí $U = 1V$, a v čase $t = 0{,}2s$ byl na vstup 2 přiveden zátěžný moment $M_0 = 1Nm$.

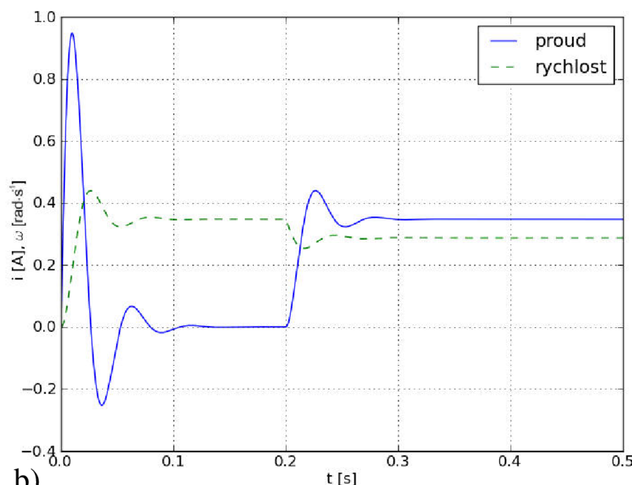
Kód v Pythonu:

```
def Motor(x, t, u):                    # definice funkce
    dx = [0.0] * 2                     # vytvoření výstupního vektoru

    km = 2.88                          # zadání konstant
    R = 0.5
    J = 0.1
    L = 0.005

    dx[0] = (u[0]-R*x[0]-km*x[1])/L # zápis rovnic
    dx[1] = (km*x[0]-u[1])/J
    return dx
```

```
x0 = [[0.0], [0.0]]                          # zadání počátečních podmínek
PID = [1, 0, 0]                              # jednotkový přenos přes regulátor

t = time(0.5, 0.001)                         # vytvoření pole časů s krokem
u = step(t, [0.0, 0.2])                      # vytvoření pole 2 vstupů

y = Sim(PID,Motor,t,u,False,x0)              # výpočet průběhu
plot(t, y)                                   # vykreslení grafu
```



a)



b)

Obr. 18 Rozběh stejnosměrného motoru. a) V Simulinku. b) V tomto simulátoru.

5.5.12 Řízení polohy stejnosměrného motoru

Aby bylo možné řídit polohu motoru, je nutné k rovnicím dynamiky motoru (9) a (10) ještě přidat rovnici natočení motoru

$$\dot{\varphi} = \omega , \qquad (12)$$

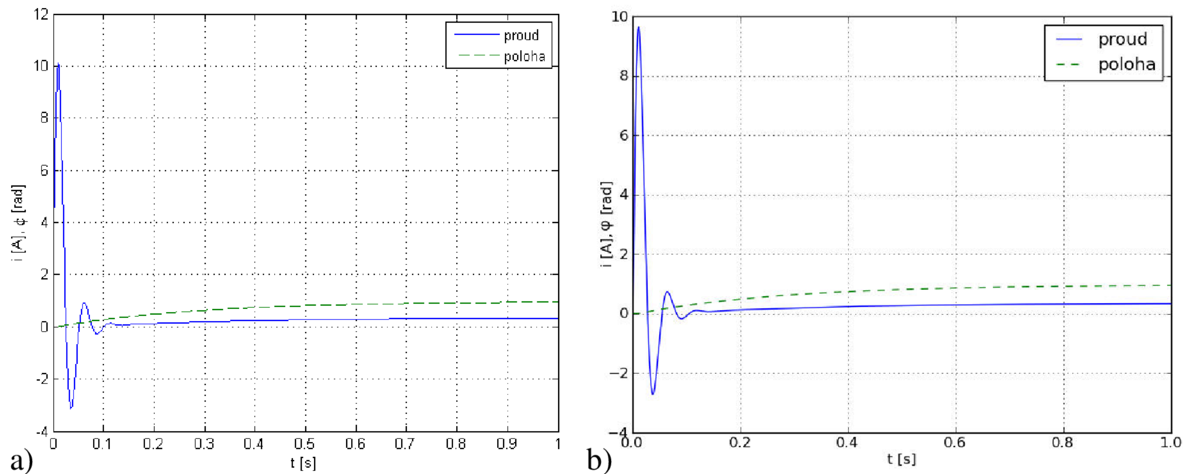
a všechny tyto rovnice přepsat do tvaru pro jazyk Python. Aby byla na výstupu vidět nejen poloha, ale i proud, byl vytvořen dvouvýstupový systém, kde první výstup z motoru vrací proud protékajícím motorem a druhý výstup vrací polohu. V modelu motoru je pak řízen pomocí zpětné vazby a PID regulátoru pouze druhý vstup znázorňující polohu motoru.

Parametry motoru zůstaly stejné, jako v předešlém příkladu 5.5.11, jen zátěžný moment M_0 byl zvolen jako konstantní s hodnotou $M_0 = 1Nm$. Výsledný průběh simulace je zobrazen na Obr. 19.

Kód v Pythonu:

```
def Motor(x, t, u):                          # definice funkce
    dx = [0.0] * 3                           # vytvoření výstupního vektoru

    km = 2.88                                # zadání konstant
    R = 0.5
    J = 0.1
    L = 0.005
    M0 = 1

    dx[0] = (u[1]-R*x[0]-km*x[2])/L# rovnice - proud
    dx[1] = x[2]                             #         - poloha
    dx[2] = (km*x[0]- M0)/J                  #         - rychlost
    return dx
```

```
x0 = [[0.0],[0.0]]                    # zadání počátečních podmínek
PID = [10, 0, 0]                      # jednotkový přenos přes regulátor

t = time(1, 0.001)                    # vytvoření pole časů s krokem
u = step(t, [1.0, 0.0])               # vytvoření pole 2 vstupů

y = Sim(PID,Motor,t,u,True,x0)        # výpočet průběhu
plot(t, y)                            # vykreslení grafu
```

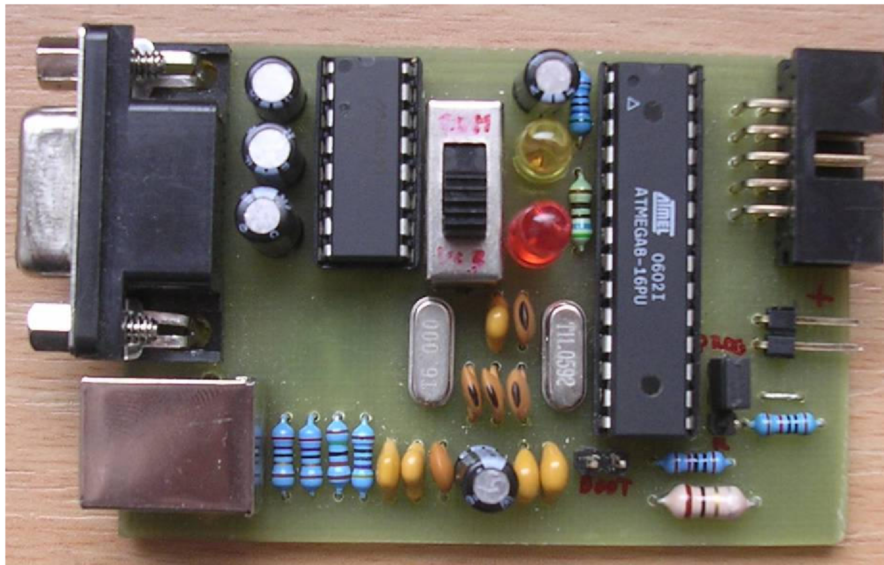


Obr. 19 Řízení polohy stejnosměrného motoru. a) V Simulinku. b) V tomto simulátoru.

5.6 Řízení modelu pomocí mikrokontroléru

Příklady v této kapitole ukazují, jakým způsobem lze vytvořit z mikrokontroléru regulátor a jak s tímto regulátorem řídit pomocí sériového portu model vytvořený v tomto simulátoru.

Jako regulátor byl použit mikroprocesor od firmy Atmel z rodiny AVR typu ATmega8.



Obr. 20 Programátor pro mikrokontrolér ATmega8.

Řízení modelu vytvořeném v tomto simulátoru po sériovém portu pomocí mikrokontroléru bude dobře vidět na modelu kyvadla z příkladu 5.5.7. Na řízení bude použit stejný PID regulátor jako v příkladu 5.5.10.

5.6.1 Generování C-kódu s datovým typem double

Aby byl mikrokontrolér použitelný, je nutné jeho funkci regulátoru nejprve naprogramovat. K tomuto účelu slouží funkce *CreateC*, která se také nachází v souboru *Simulace.py*. Tato funkce podle zadaných parametrů vytvoří podprogram regulátoru v C-kódu. Bližší popis této funkce se nachází v příloze v kapitole II.1.5.

V tomto případě se zadává do prvního parametru funkce cesta k souboru, do kterého se kód uloží. Do druhého parametru se zadává proměnná obsahující pole se třemi konstantami PID regulátoru. Do třetího parametru se zadává interval, ve kterém bude mikrokontrolér načítat hodnoty ze vstupu a podle nich regulovat výstupní hodnoty. Do posledního parametru se zadává číselný typ, se kterým bude mikrokontrolér pracovat.

Příklad kódu na vytvoření podprogramu PID regulátoru:

```
PID = [10.0,40.0,5.0]          # zadání konstant PID regulátoru
T = 0.01                       # zadání času 1 výp.kroku v sekundách
CreateC(‘C:\pid‘, PID, T, typ = 'double')
      # vygenerování podprogramu PID reg. do souboru ‘C:\pid.c‘,
      # který pracuje s datovým typem double s krokem 0.01s
```

Vygenerovaný kód pak vypadá následovně, zde je pro vysvětlení ještě doplněn o komentář:

```
/************************************************************************
* Regulator PID                                                         *
************************************************************************/

/************* DEFINE CONSTANTS ***************/

#define Kp 10.0              // konstanty pro výpočet
#define Ki 0.4
#define Kd 500
#define T 0.01

/************* GLOBAL VARIABLES ***************/

double P, I, D, S;           // proměnné

/************* PROCEDURES ********************/

void init_reg(void)          // inicializace proměnných
{
      P = 0;
      I = 0;
      D = 0;
      S = 0;
}
double calculate(double value)     // výpočet regulátoru
{
      P = value * Kp;              // P složka
      I = I + Ki * value;          // I složka
      D = Kd * (value - S);        // D složka
      S = value;                   // zápis vstupu pro D složku
      return(P + I + D);           // součet
}
```

5.6.2 Naprogramování mikrokontroléru

Vygenerovaný podprogram pak už jen stačí přidat do ručně vytvořeného hlavního programu pro mikrokontrolér. Hlavní program musí mít takovou strukturu, aby načetl ze sériového rozhraní vstupní hodnotu, která je vysílána ve formě řetězce po sobě jdoucích znaků zakončených ukončovacím znakem. Pak musí program tento řetězec převést na

číslo typu *double*, přepočítat ho pomocí podprogramu regulátoru, převézt zpět na řetězec znaků a vyslat ho zpět po sériové lince do počítače.

Hlavní program pro mikrokontrolér ATmega8 pak vypadá následovně:

```
void main(void)
{
      char s[16]; int i; double f; // deklarace proměnných

      uart_init();                 // inicializace sériového rozhraní
      init_reg();                  // inicializace regulátoru

      while(1)                     // hlavní program
      {
            if((UCSRA & (1 << RXC)) == 1 ) // pokud přijat znak
            {
                  i = 0;
                  s[i++] = UDR;              // přijmout do proměnné
                  while (s[i-1] != '\n')
                  {                // dokud nebude ukončovací znak
                        while ((UCSRA & (1 << RXC)) == 0) {};
                        s[i++] = UDR;        // přijmout do proměnné
                  }
                  s[i-1] = '\0';             // ukončit řetězec
                  f = atof(s);               // převést na číslo
                  f = calculate(f);          // výpočet
                  sprintf(s, "%.8f", f);     // převést na řetězec
                  putStr(s);                 // odeslat řetězec
            }
      }
}
```

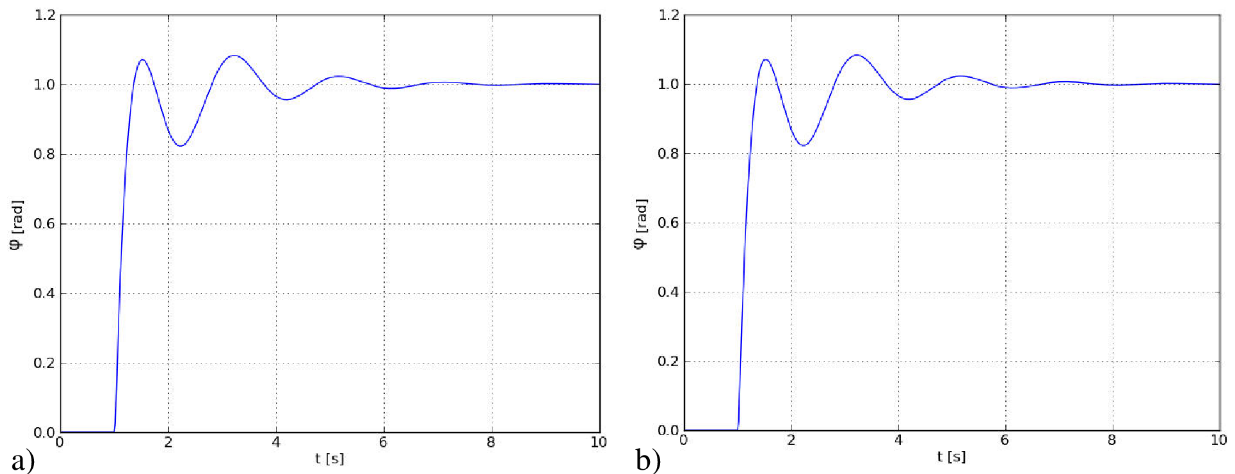
Celý kód lze zkompilovat v prostředí AVR Studio[23] pomocí kompilátoru AVR-GCC a zkompilovaný soubor nahrát do mikrokontroléru.

5.6.3 Regulace modelu pomocí mikrokontroléru

Naprogramovaným mikrokontrolérem teď už lze řídit v simulátoru vytvořený model kyvadla z příkladu 5.5.7. Stačí u funkce *Sim* dosadit do parametru prvního bloku regulátoru název sériového portu, na který je mikrokontrolér připojen, v tomto případě COM3. Do parametru druhého bloku řízeného systému se pak stačí zadat funkci kyvadla a výstup funkce *Sim* následně vykreslit. Výsledný průběh je možné porovnat se simulací PID regulátoru, který má stejné konstanty, jako PID regulátor v mikrokontroléru. Na Obr. 21 je vidět, že se tyto průběhy se od sebe neliší.

Kód v Pythonu:

```
x0 = [[0.0], [0.0, 0.0]]          # počáteční podmínky
t = time(10, 0.01)                # vytvoření pole času
u = step(t, 1)                    # vytvoření jednot. skoku
y = Sim(['COM3'],kyvadlo,t,u,True,x0) # výpočet průběhu
plot(t,y)                         # vykreslení
```



Obr. 21 Regulace modelu kyvadla. a) Mikrokontrolér. b) Simulovaný PID regulátor

5.6.4 Regulace modelu pomocí mikrokontroléru s datovým typem integer

Aby se zkrátila délka výpočtu regulátoru, je lepší, pokud je to možné, používat celočíselný datový typ *integer*. Na jeho použití se při programování mikrokontroléru postupuje prakticky stejně jako v případě práce s datovým typem double (viz kap. 5.6.3), pouze u funkce *CompileC* se změní hodnota parametru *typ* na *int*. Dále je třeba dbát na to, aby parciální složka, součin integrační složky s délkou kroku a podíl derivační složky s délkou kroku byla celá čísla. Tomuto požadavku při délce kroku 0,01s odpovídá např. regulátor obsahující tyto složky:

$$K_p = 40,\ K_i = 400,\ K_d = 40. \qquad (13)$$

Příklad vytvoření podprogramu PID regulátoru pracující s datovým typem integer:

```
PID = [40, 400, 40]          # zadání PID regulátoru
T = 0.01                     # zadání času 1 kroku v sekundách
CreateC(‘C:\pid‘, PID, T, typ = 'int')
```

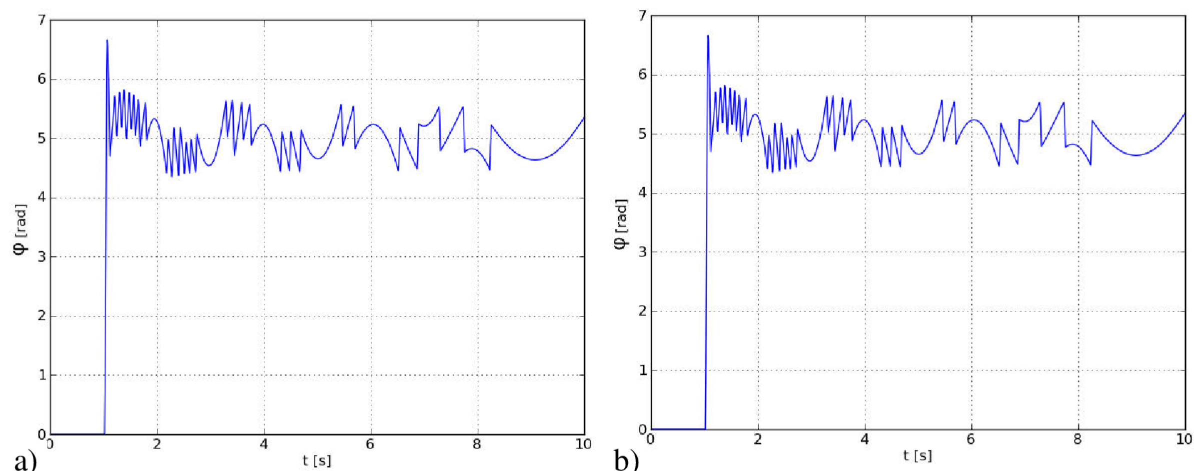
Po naprogramování mikrokontroléru je možné zkusit řídit model kyvadla tímto celočíselným regulátorem. Stejně jako ve funkci *CreateC* je nutné i ve funkci *Sim* zapnout v parametru *typ* pro blok regulátoru práci s celými čísly dosazením hodnoty *‘int‘*. Na Obr. 22 je vidět porovnání výsledků řízení kyvadla celočíselným regulátorem a celočíselnou simulací tohoto regulátoru.

Kód pro simulaci PID regulátoru v Pythonu:

```
PID = [40, 400, 40]          # zadání PID regulátoru
y = Sim(PID,kyvadlo,t,u,True,x0,False,[],['int','float'])
plot(t, y)                   # vykreslení
```

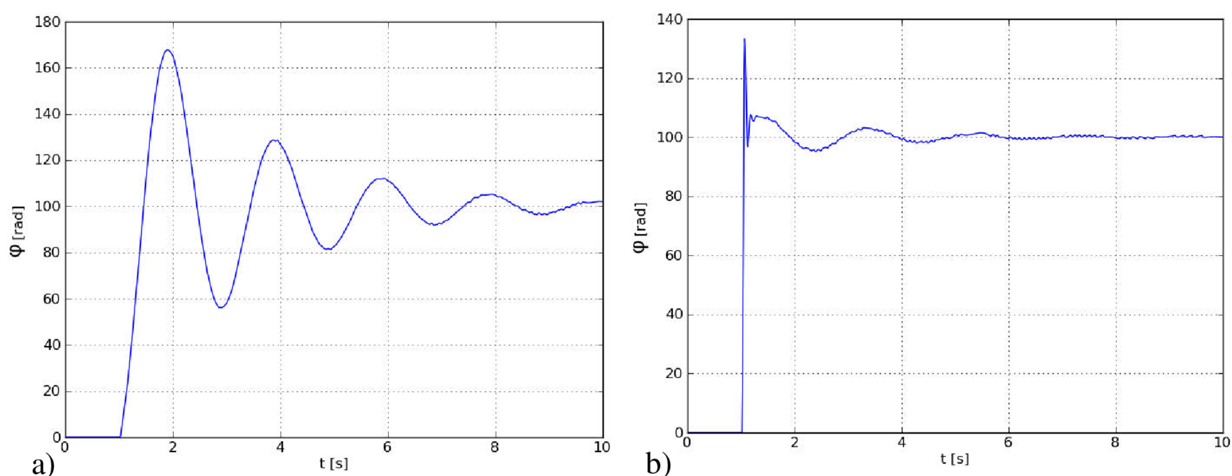
Kód pro simulaci mikrokontroléru v Pythonu:

```
y = Sim([‘COM3‘],kyvadlo,t,u,True,x0,False,[],['int','float'])
plot(t, y)                   # vykreslení
```



Obr. 22 Řízení celočíselným regulátorem. a) Mikrokontrolér. b) Simulátor.

Oba průběhy řízení pro skok při natočení 5 radiánů se neliší. Zlom nastává při větších natočeních, například při 100 radiánech, kdy si simulátor se skokem poradil o mnoho lépe než mikrokontrolér, kterému s největší pravděpodobností přetekl integrační zásobník. Porovnání průběhů je na Obr. 23.



Obr. 23 Řízení celočíselným regulátorem. a) Mikrokontrolér. b) Simulátor.

5.7 Řízení reálné aplikace pomocí mikrokontroléru

Zkouška řízení reálné aplikace pomocí mikrokontroléru fungujícího jako PID regulátor byla provedena na stejnosměrném motoru s vestavěným enkodérem.

Mikrokontrolér ovládal motor přes výkonový člen pomocí dvou výstupů. První výstup určoval směr otáčení motoru. Pokud byl nastaven do logické 1, motor se otáčel v kladném směru, pokud byl nastaven do logické 0, motor se otáčel v záporném směru. Na druhý výstup byl vysílán PWM signál, který ovládal velikost napětí na motoru, čímž reguloval jeho rychlost.

Z motoru byl vyveden do mikrokontroléru výstup z enkodéru, který vytvářel 512 impulzů na jednu otáčku motoru. Pomocí těchto impulzů dostával mikrokontrolér z motoru zpětnou vazbu, ze které mohl určit jeho aktuální polohu a adekvátně ho regulovat.

V této kapitole je rozebrán příklad regulace stejnosměrného motoru pomocí mikrokontroléru Microchip dsPIC33FJ128MC804.

5.7.1 Regulace natočení motoru pomocí PID regulátoru se zpětnou vazbou

V tomto případě mikrokontrolér dsPIC33FJ128MC804 načítal impulsy z enkodéru pomocí přerušení vyvolané vzestupnou hranou signálu na pinu INT1. Každých 0,01s pak mikrokontrolér porovnával načtený počet impulsů s požadovanou hodnotou natočení motoru. Časový interval byl nastaven pomocí časovače TIMER1, který každých 0,01s vyvolal přerušení.

Načtené vstupní hodnoty mikrokontrolér přepočítával podle funkce PID regulátoru, který měl nastaveny tyto konstanty:

$$K_p = 8\,,\ K_i = 0{,}003\,, K_d = 30\,, \qquad (14)$$

kde, K_p je konstanta proporcionálního členu, K_i konstanta integračního členu a K_d konstanta derivačního členu.

Pomocí přepočtených výstupních hodnot z funkce regulátoru se generovala velikost střídy PWM signálu generovaného pomocí časovače TIMER2, kterým se řídilo napětí přivedené na motor.

Zjednodušená struktura C kódu pro mikrokontrolér dsPIC vypadala následovně:

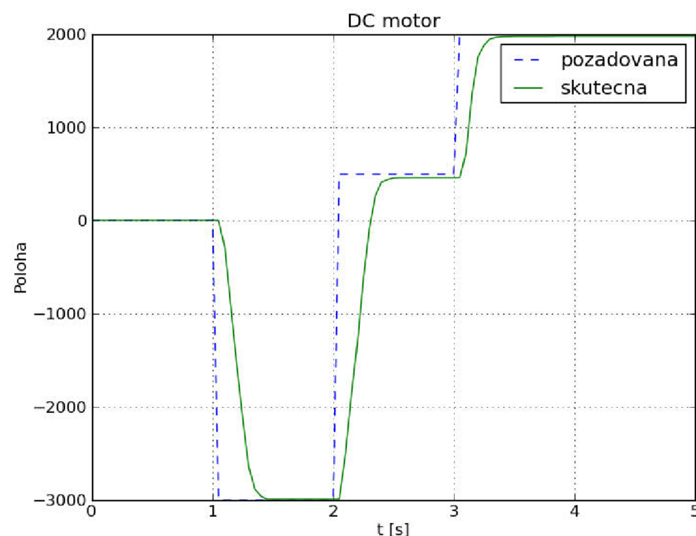
```
int vystup, poloha;                 // deklarace globálních proměnných

void main(void)                     // hlavní program
{
        char znaky[16]; int i;      // deklarace lokálních proměnných
        UART_init();                // inicializace sériového rozhraní
        INT_init();                 // inicializace portů
        PWM_init();                 // inicializace generování PWM
        Interrupt_init();           // inicializace přerušení
        T1_init();                  // inicializace Timeru 1

        while(1)                    // začátek nekonečné smyčky
        {
               getString(znaky);    // procedura pro přijetí řetězce ze
                                    // sériového portu

               if (znaky[0] == 'u')         // pokud vyslat vstup
               {
                      itoa(poloha, znaky, 10); // převést číslo na řetězec
                      sendString(znaky);       // poslat hodnotu vstupu
               }
               else if (znaky[0] == 'y')   // pokud vyslat výstup
               {
                      itoa(vystup, znaky, 10); // převést číslo na řetězec
                      sendString(znaky);       // poslat hodnotu výstupu
               }
               else if (znaky[0] == 'h')   // pokud nový požadovaný vstup
               {
                      hodnota = atoi(znaky); // zapsat přijatou hodnotu do
               }                             // proměnné hodnota
        }
}


void __attribute__((interrupt, no_auto_psv)) _T1Interrupt(void)
// přerušení vyvolané časovačem TIMER1
{
    double x; char s[16];
        x = hodnota - poloha;                // výp. vstupu do regulátoru
        sprintf(s, "%.8f", calculate(x));    // převod na vypočítané
        vystup = atoi(s);                    //   hodnoty na integer
        PWM(vystup);                         // řízení motoru pomocí PWM
}
```

```
void __attribute__ ((interrupt, no_auto_psv)) _INT1Interrupt(void)
// přerušení při vzestupné hraně signálu z enkodéru
{
   if (PORTCbits.RC7 == 1){        // aktualizace proměnné poloha
         poloha--;}               // motor se točí záporným směrem
   else{
         poloha++;}               // motor se točí kladným směrem
}
```

V tomto příkladu se pomocí funkce *status* načítaly z mikrokontroléru připojeného na COM3 hodnoty polohy motoru po dobu 5s v intervalu 0,05s. Pomocí spojení dvou polí vstupů vygenerovaných funkcemi *Step* byl vytvořen jeden průběh požadované polohy motoru. Nakonec byly do jednoho grafu pomocí funkce *plot* vykresleny průběhy požadované i skutečné polohy motoru. Graf je znázorněn na Obr. 24.

Kód v Pythonu pro sledování činnosti regulátoru:

```
# ----- Vytvoření složitějšího pole vstupů -----

t1 = time(2,0.05)              # vytvoření 2 úseků pole časů
t2 = time(3,0.05)

u1 = step(t1,1,-3000)          # pro oba úseky vstupní hodnoty
u2 = step(t2,1,2000,500)

t = time(5,0.05)               # vytvoření celkového pole časů
u = u1 + u2[1:]                # spojení dvou polí vstupů

# ----- Sledování regulace mikrokontroléru -----

y = status('COM3',t,True,False,u) # sledování regulace mikrokont.
y = [u,y]                      # spojení pole vstupů a výstupů
                               #  do jednoho grafu
plot(t, y)                     # vykreslení grafu
```



Obr. 24 PID regulátor pomocí dsPIC

6 ZÁVĚR

V této práci byl vytvořen program pro simulaci průběhu signálu soustavou, který obsahoval regulátor, řízený systém, zpětnou vazbu, šum a generátor referenční trajektorie. Dynamiku modelu řízeného systému lze definovat více způsoby, např. pomocí přenosové funkce, explicitních diferenciálních rovnic, pomocí modelu vytvořeném v ODE atd. Pro regulátor i řízený systém lze zvolit použitý řešič i délku výpočetního kroku zvlášť.

Funkce simulátoru byla znázorněna na příkladech simulace řízení natočení kyvadla i simulace řízení polohy stejnosměrného motoru. Výsledné hodnoty byly porovnány s výpočtem hodnot pomocí prostředí Matlab/Simulink. Všechny sledované nasimulované průběhy se nijak výrazně nelišily, což by mělo svědčit o správnosti naprogramování simulátoru. Rozdíl byl pozorován u jen některých velmi strmých průběhů, kde mohla chyba vzniknout volbou jiného řešiče diferenciálních rovnic.

Dále bylo dosaženo toho, že blok regulátoru i řízeného modelu může být nahrazen komunikací po sériové lince s libovolným zařízením. Při nahrazení bloku regulátoru tak lze například odzkoušet správnou funkci reálného mikroprocesoru naprogramovaného jako regulátor.

Program simulátoru je možné také využít pro zjednodušení naprogramování funkce regulátoru do mikrokontrolérů. Toho lze dosáhnout převodem diskrétní přenosové funkce nebo PID regulátoru do podprogramu pro programovací jazyk C. Tento podprogram podporuje práci buď s čísly s plovoucí desetinnou čárkou nebo s čísly celými.

Generování C-kódu diskrétní přenosové funkce a PID regulátoru proběhlo úspěšně. Správná funkčnost tohoto kódu byla odzkoušena na reálných mikrokontrolérech ATmega8 a dsPIC33FJ128MC804 k řízení reálného stejnosměrného motoru.

7 LITERATURA A ODKAZY

[1] HARMS, Daryl; MCDONALD, Kenneth. *Začínáme programovat v jazyce Python*. 1. vydání. Brno: Computer Press, 2003. 456 s. ISBN 80-7226-799-X

[2] SKLICKÝ, Jiří. *Teorie řízení*. 2. vydání, Brno: VUT FEKT, 2002. 98 s.

[3] PYTHON SOFTWARE FOUNDATION. *Python*. [online]. 2010-05-10.
Dostupné z < http://www.python.org/ >

[4] PYTHON SOFTWARE FOUNDATION. *Python install files*. [online]. 2010-05-10.
Dostupné z < http://www.python.org/ftp/python/2.6.5/python-2.6.5.msi >

[5] NUMPY DEVELOPERS. *NumPy*. [online]. 2010-05-10.
Dostupné z < http://numpy.scipy.org/ >

[6] NUMPY DEVELOPERS. *NumPy install files*. [online]. 2010-05-10.
Dostupné z < http http://sourceforge.net/projects/numpy/files/ >

[7] NUMPY COMMUNITY. *NumPy User Guide* [online]. 2010-04-15.
Dostupné z < http://docs.scipy.org/doc/numpy/numpy-user.pdf >

[8] NUMPY COMMUNITY. *NumPy Reference Guide* [online]. 2010-04-15.
Dostupné z < http://docs.scipy.org/doc/numpy/numpy-ref.pdf >

[9] SCIPY DEVELOPERS. *SciPy*. [online]. 2010-05-10.
Dostupné z < http://scipy.org/ >

[10] SCIPY DEVELOPERS. *SciPy install files*. [online]. 2010-05-10.
Dostupné z < http://sourceforge.net/projects/scipy/files/ >

[11] SCIPY COMMUNITY. *SciPy Reference Guide* [online]. 2010-04-15.
Dostupné z < http://docs.scipy.org/doc/scipy/scipy-ref.pdf >

[13] DALE, Darren a kolektiv. *Matplotlib*. [online]. 2010-05-10.
Dostupné z < http://matplotlib.sourceforge.net/ >

[14] DALE, Darren a kolektiv. *Matplotlib documentation*. [online]. 2010-05-10.
Dostupné z < http://matplotlib.sourceforge.net/Matplotlib.pdf >

[15] DALE, Darren a kolektiv. *Matplotlib install files*. [online]. 2010-05-10.
Dostupné z < http://sourceforge.net/projects/matplotlib/files/matplotlib/matplotlib-0.99.1/ >

[16] LIECHTI, Chris. *pySerial*. [online]. 2010-05-10.
Dostupné z < http://pyserial.sourceforge.net/ >

[17] LIECHTI, Chris. *pySerial install files*. [online]. 2010-05-10.
Dostupné z < http://sourceforge.net/projects/pyserial/files/ >

[18] WIKIPEDIA. *Mikrokontrolér PIC*. [online]. 2010-05-10.
Dostupné z < http://cs.wikipedia.org/wiki/Mikrokontrolér_PIC >

[19] MICROCHIP. *dsPIC33FJ128MCX02/X04 Data Sheet*. [online]. 2010-05-10.
Dostupné z < http://ww1.microchip.com/downloads/en/DeviceDoc/70291D.pdf >

[20] MATHWORKS. *MATLAB/Simulink*. [online]. 2010-05-10.
Dostupné z < http://www.mathworks.com/ >

[21] EATON, J.W. a kolektiv. *Octave*. [online]. 2010-05-10.
Dostupné z < http://www.gnu.org/software/octave/ >

[22] TIŠNOVSKÝ, Pavel. [online]. 2006-05-24
Dostupné z < http://www.root.cz/clanky/fixed-point-arithmetic/ >

[23] ATMEL. AVR Studio. [online] 2010-05-21.
Dostupné z < http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2725 >

[24] HEROUT, Pavel. *Učebnice jazyka C*. 4.vydání. České Budějovice: Kopp, 2007. 271 s.
ISBN 80-7232-220-6

[25] MM PRŮMYSLOVÉ SPEKTRUM. *Rotační snímače (enkodéry).* [online]. 2010-05-26.
Dostupné z < http://www.mmspektrum.com/clanek/rotacni-snimace-enkodery >

[26] WXPYTHON DEVELOPERS. *wxPython*. [online]. 2011-05-06.
Dostupné z < http://www.wxpython.org/ >

[27] WXGLADE DEVELOPERS. *wxGlade*. [online]. 2011-05-06.
Dostupné z < http://wxglade.sourceforge.net/ >

[28] PYODE DEVELOPERS. *PyODE*. [online]. 2011-05-06.
Dostupné z < http://pyode.sourceforge.net/ >

[29] SMITH, Russell. *Open Dynamics Engine*. [online]. 2011-05-06.
Dostupné z < http://www.ode.org/ >

[30] ATMEL. *ATmega8(L)*. [online]. 2011-05-06.
Dostupné z < http://www.atmel.com/dyn/resources/prod_documents/doc2486.pdf >

[31] MICROCHIP. *dsPIC33FJ128MCX02/X04 Data Sheet*. [online]. 2011-05-06.
Dostupné z < http://ww1.microchip.com/downloads/en/DeviceDoc/70291E.pdf >

[32] SMITH, Russell. *ODE Manual*. [online]. 2011-05-06.
Dostupné z < http://opende.sourceforge.net/wiki/index.php/Manual_%28All%29 >

[33] SCHREIBER, Petr. *Návrh automatického generátoru prostředí pro mobilní robot.* 1.vydání. Brno, 2008. 57 s.

8 SEZNAM POUŽITÝCH SYMBOLŮ

φ – úhlové zrychlení $[\mathrm{rad}]$
ω – úhlová rychlost $[\mathrm{rad} \cdot \mathrm{s}^{-1}]$
g – tíhové zrychlení $[\mathrm{m} \cdot \mathrm{s}^{-2}]$
l – délka $[\mathrm{m}]$
i – proud $[\mathrm{A}]$
U – napětí $[\mathrm{V}]$
R – elektrický odpor $[\Omega]$
L – indukčnost $[\mathrm{H}]$
k_m – konstanta motoru, která je dána konstrukčním provedením
M – moment $[\mathrm{Nm}]$
M_0 – zatěžovací moment $[\mathrm{Nm}]$
J – moment setrvačnosti $[\mathrm{kg} \cdot \mathrm{m}^2]$
t – čas $[\mathrm{s}]$
K_p – proporcionální složka PID regulátoru
K_i – integrační složka PID regulátoru
K_d – derivační složka PID regulátoru
bps – sériová datová rychlost – bitů za sekundu
explicitní ODE – explicitně vyjádřená soustava diferenciálních rovnic
PID regulátor – proporcionálně integračně derivační regulátor
PWM – pulzně šířková modulace
ASCII – kódy pro znaky v digitální technice
Enkodér – převodník převádějící rotační pohyb na číslicové impulzy[25]
ODE – The Open Dynamics Engine
FLASH – typ paměti
SRAM– typ paměti
EEPROM– typ paměti
MIPS – milion instructions per sekond - milion instrukcí za sekundu

Datové typy:

- Float – datový typ s plovoucí desetinnou čárkou
- Double – datový typ s plovoucí desetinnou čárkou se zvýšenou přesností
- Fixed point – datový typ s pevnou desetinnou čárkou
- Integer – celočíselný datový typ
- Boolean – datový typ s dvěmi možnostmi – pravda/nepravda (True/False)
- Ndarray – vícerozměrné pole z knihovny NumPy

PŘÍLOHY

I ODE

I.1 Typický postup vytváření a simulaci modelu ODE

Při používání Open Dynamics Engine se obvykle program dělí na 2 části. V první části se vytváří model dynamického světa obsahující všechny tělesa a vazby. Ve druhé části, která tvoří výpočetní smyčku, probíhá po jednotlivých krocích simulace dynamiky světa. Typický postup vytváření a simulace modelu ODE je uveden níže v této kapitole.

I.1.1 Vytvoření modelu

Při vytváření modelu v prostředí ODE obvykle postupuje v následujících krocích:

1. Vytvoření dynamického světa

```
world = ode.World()                   # vytvoření dynamického světa
world.setGravity((0,-9.81,0))         # vytvoření gravitačního pole
world.setERP(0.8)                     # zadání konstanty ERP
world.setCFM(1E-5)                    # zadání konstanty CFM
```

2. Vytvoření těles v dynamickém světě

```
# vytvoření tělesa body v dynamickém světě world
body = ode.Body(world)
# vytvoření hmoty tělesa
M = ode.Mass()
# tvaru koule s hustou 2500 a poloměrem 0.05
M.setSphere(2500.0, 0.05)
# Nastavení hmotnosti koule na 1kg
M.mass = 1.0
# přiřazení hmoty M tělesu body
body.setMass(M)
```

3. Nastavení pozice všech těles

```
# nastavení pozice tělesa body na souřadnice [0, 0, 0]
body.setPosition((0, 0, 0))
```

4. Vytvoření vazeb v dynamickém světě

```
# vytvoření rotační vazby v dynamickém světě world
joint = ode.HingeJoint(world)
```

5. Připojení vazeb k tělesům

```
# připojení vazby k tělesu a světu
joint.attach(body, ode.environment)
```

6. Nastavení parametrů všech vazeb

```
# nastavení polohy vazby na souřadnice [0, 1, 0]
joint.setAnchor((0, 1, 0))
# zvolení osy rotace z
joint.setAxis((0, 0, 1))
```

7. Při potřebě kolizí též vytvoření kolizního prostoru a kolizní geometrie objektů

```
# Vytvoření kolizního prostoru space
space = ode.Space()
# Vytvoření podlahy floor v kolizním prostoru space
floor = ode.GeomPlane(space, (0,1,0), 0)
```

```
# vytvoření kolizní geometrie tvaru koule o poloměru 0.05
geom = ode.GeomBox(space, 0.05)
# přiřazení kolizní geometrie tělesu body
geom.setBody(body)
```

8. Vytvoření skupiny vazeb pro kontaktní vazby

```
contactgroup = ode.JointGroup() # vytvoř. skupiny kontaktních vazeb
```

I.1.2 Simulace modelu

Když už je model vytvořen, probíhá v jednom cyklu simulace dynamického světa. V cyklu probíhají obvykle následující kroky:

1. Působení sil na tělesa

```
body.addTorque((0,0,3))    # vytvoř. momentu působící na těleso body
body.addForce((0,100,0))   # vytvoř. síly působící na těleso body
```

2. Je-li potřeba, tak upravení parametrů vazeb

```
joint.addTorque((0,0,3))    # vytvoř. momentu působící ve vazbě
```

3. Zavolání detekce kolizí

```
# Vyvolání funkce detekující kolize. Tato funkce vyvolává další
# funkci near_callback, ve které můžeme sami definovat konstanty
# tření a odrazivosti nalezených kolizí.
space.collide((world,contactgroup), near_callback)
```

4. Ve funkci detekce kolizí vytvoření kontaktní vazby pro každý kolizní bod a její přidání do skupiny kontaktních vazeb

```
def near_callback(args, geom1, geom2):

    # Kontrola, zda-li dochází ke kontaktu dvou geometrií těles
    contacts = ode.collide(geom1, geom2)

    world,contactgroup = args
    for c in contacts:            # pro každý nalezený kontaktní bod
        c.setBounce(0.2)          #   zadání konstanty odrazivosti
        c.setMu(5000)             #   zadání konstanty tření
        # Vytvoření kontaktních vazeb
        j = ode.ContactJoint(world, contactgroup, c)
        j.attach(geom1.getBody(), geom2.getBody())
```

5. Vyvolání kroku simulace

```
world.step(0.01)                # další krok (0.01s) simulace světa
```

6. Je-li potřeba, tak načtení hodnot dat

```
x1,y1,z1 = body.getPosition()   # zjištění nové pozice tělesa body
```

7. Odebrání všech vazeb ze skupiny kontaktních vazeb

```
contactgroup.empty()            # odebrání všech kontaktních vazeb
```

II PROGRAM SIMULACE

II.1 Definice funkcí ve skriptu Simulace.py

V této kapitole se nacházejí definice a bližší popis funkcí, které se nachází v souboru *Simulace.py*. Podle jejich použití jsou rozděleny do kapitol na funkci na vytvoření pole času, funkce pro vytvoření pole vstupů, funkci pro simulaci, funkci pro vykreslení grafů, funkci pro generování C-kódu a funkci pro kontrolu mikrokontroléru po sériovém rozhraní.

II.1.1 Funkce pro vytvoření pole času

time(tmax, step)

Funkce generuje řadu čísel typu float, které začínají hodnotou 0. Každé číslo se potom postupně se navyšuje o hodnotu *step* až do hodnoty *tmax*. Pomocí této funkce je možno vygenerovat tabulku časů, podle které se pak integruje ve funkci *Sim* průběh signálu.

Příklad použití:

```
t = time(10, 0.1)

>> t = [0.0, 0.1, 0.2, ... , 9.8, 9.9, 10.0]
```

II.1.2 Funkce pro vytvoření pole vstupů

step(t, start, max = 1.0, min = 0.0, typ = ‘float‘)

Tato funkce vygeneruje průběh signálu s jedním skokem pro tabulku času *t*. Skok se provede v čase *start*. Za nepovinný parametr *max* se dosazuje hodnota signálu po provedeném skoku, která defaultně nastavena na velikost 1. Do nepovinného parametru *min* se dosazuje hodnota signálu před provedením skoku, která defaultně nastavena na velikost 0. Nepovinný parametr *typ* určuje, jakého číselného typu se budou generovat prvky. Buď může generovat prvky s plovoucí desetinnou čárkou typu *float* nebo celá čísla typu *int*. Dále je také v této funkci možné generovat signál s více vstupy, a to tak, že se za parametry *start*, *max* a *min* nebudou dosazovat samostatné proměnné, ale vloží se za ně pole s příslušným počtem prvků. Tento počet se určí podle toho, kolik je potřeba v daném okamžiku výstupů.

Příklad použití – generování jednoho vstupu:

```
t = time(10, 0.1)
u = step(t, 1)

>> t = [0.0, 0.1, ... , 0.9, 1.0, 1.1, ..., 9.9, 10.0]
>> u = [0.0, 0.0, ... , 0.0, 1.0, 1.0, ..., 1.0, 1.0]
```

Příklad použití – generování více vstupů:

```
t = time(10, 1)
u = step(t, [1, 2], [2.0, 1.0])

>> t = [ 0.0,        1.0,        2.0,        ..., 10.0]
>> u = [[0.0, 0.0], [2.0, 0.0], [2.0, 1.0], ..., [2.0, 1.0]]
```

sine(t, period, amplitude = 1.0, shift = 0.0, typ = ‘float‘)

Tato funkce vygeneruje průběh sinusového signálu pro tabulku času *t*. Parametr *period* udává velikost periody sinusového průběhu. Nepovinný parametr *amplitude* udává velikost amplitudy signálu, která je defaultně nastavena na hodnotu 1. Nepovinný parametr *shift* udává velikost posunutí v časové ose. Nepovinný parametr *typ* opět určuje, jakého typu budou generovány prvky, zda-li čísla s plovoucí desetinnou čárkou typu *float*, či celá čísla typu *int*. I v této funkci je možné generovat signál s více vstupy. To se provádí tak, že za parametry *period*, *amplitude* a *shift* nebudeme dosazovat samostatné proměnné, ale pole s příslušným počtem prvků podle toho, kolik je potřeba výstupů.

Příklad použití:

```
t = time(10, 0.1)
u = step(t, 1)

>> t = [0.0, 0.1,             ..., 9.9,              10.0]
>> u = [0.0, 0.5877852522924, ..., -0.5877852522924, 0.0]
```

pulse(t, period, width, saw = False, amplitude = 1.0, shift = 0.0, typ = ‘float‘)

Funkce vygeneruje průběh PWM signálu pro tabulku času *t*. Proměnná *period* udává velikost periody pulzního průběhu, proměnná *width* určuje délku sepnutého stavu signálu. Funkce má nepovinný parametr *saw*. Pokud nabývá hodnoty True, pak funkce generuje pilový průběh, pokud nabývá hodnoty False, vygeneruje průběh obdélníkový. Nepovinný parametr *amplitude* vyjadřuje výšku hrany signálu a nepovinný parametr *shift* určuje posun v časové ose. Další nepovinný parametr *typ* určuje, jakého typu budou generovány prvky, zda-li čísla s plovoucí desetinnou čárkou typu *float*, či celá čísla typu *int*. I zde možné generovat signál s více vstupy tak, že za parametry *period*, *width*, *saw*, *amplitude* a *shift* se nebudou dosazovat samostatné proměnné, ale vloží se za ně pole s příslušným počtem prvků podle toho, jaký počet výstupů je potřeba.

Příklad použití – generování obdélníku s jedním výstupem:

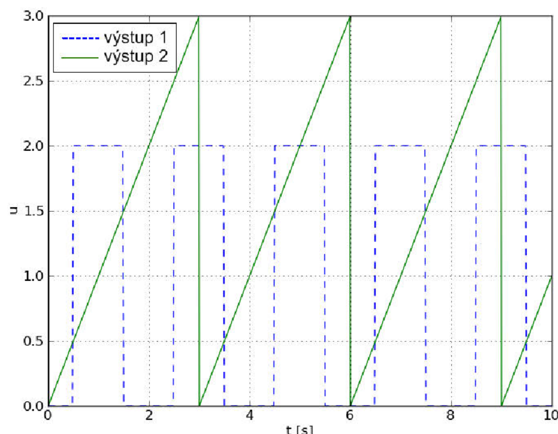
```
t = time(10, 0.1)
u = pulse(t, 0.4, 0.2)

>> t = [0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, ...]
>> u = [1.0, 1.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, ...]
```

Příklad použití – generování pily:

```
t = time(10, 0.1)
u = pulse(t, 0.4, 0.4, True)

>> t = [0.0, 0.1,  0.2, 0.3,  0.4, 0.5,  ...]
>> u = [0.0, 0.25, 0.5, 0.75, 0.0, 0.25, ...]
```

Příklad použití – generování signálu s více výstupy (viz. Obr. 2):

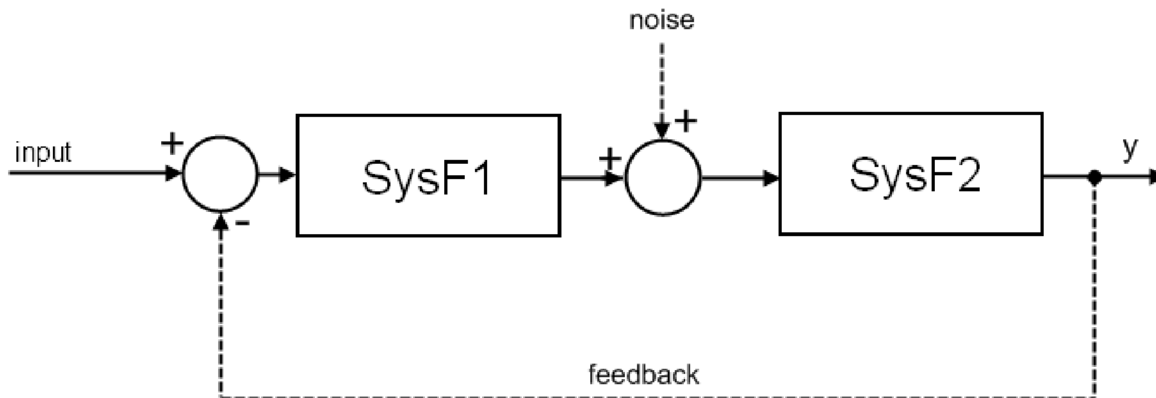
```
t = time(10, 0.01)
u = pulse(t,[2,3],[1,3],[False, True],[2.0,3.0],[0.5,0.0])
plot(t,u)
```



Obr. 25 Znázornění funkce pulse s více výstupy

II.1.3 Funkce pro simulaci

Sim(SysF1, SysF2, time, input, feedback = False, x0 = [[0.0],[0.0]], noise = False, noiseVal = [-0.01,0.01], typ = ['float','float'], integration = ['',''])



Obr. 26 Znázornění funkce Sim

Funkce vypočítává, jaký bude průběh signálu po průchodu soustavou znázorněnou na Obr. 26. Parametry *SysF1* a *SysF2* znázorňují obecné bloky dvou systémů, které upravují vstupní signál. Pro praktické použití se první blok využívá nejčastěji jako regulátor a druhý pak jako řízená soustava. Oba dva bloky lze nastavit tak, aby plnili jednu z následujících sedmi možných funkcí:

1. Přenosová funkce – zadává se do parametrů *SysF1* (*SysF2*) jako pole se dvěma prvky znázorňující čitatele a jmenovatele zlomku přenosové funkce.

   Příklad použití – přenosová funkce o rovnici:

$$F(p) = \frac{1}{5p^2 + 8p + 3} \tag{15}$$

– zápis v Pythonu:

```
SysF = [[1],[5, 8, 3]]
```

2. Matice stavové rovnice – zadává se jako pole se čtyřmi prvky, z nichž každý prvek znázorňuje jednu z matic stavové rovnice A,B,C,D.

   Příklad použití – stavové matice:

$$A=\begin{bmatrix}0 & 1\\ -0{,}6 & -1{,}6\end{bmatrix}\quad B=\begin{bmatrix}0\\ 1\end{bmatrix}\quad C=\begin{bmatrix}0{,}2 & 0\end{bmatrix}\quad D=\begin{bmatrix}0\end{bmatrix}\tag{16}$$

   – zápis v Pythonu:

```
SysF = [[[0,1],[-0.6,-1.6]], [[0],[1]], [0.2,0], [0]]
```

3. Diskrétní přenosová funkce – zadává se jako pole se třemi prvky, z nichž první dva znázorňují čitatele a jmenovatele zlomku přenosové funkce. Třetí prvek slouží jako znak k rozlišení této funkce od obyčejné přenosové funkce a může nabývat libovolné hodnoty, např. znaku ‘*z*‘.

   Příklad použití – diskrétní přenosová funkce o rovnici:

$$F(z)=\frac{1}{5z^2+8z+3}\tag{17}$$

   – zápis v Pythonu:

```
SysF = [[1],[5, 8, 3],‘z‘]
```

4. Nelineární soustava – dynamika samotné soustavy musí být v Pythonu definovaná zvlášť jako samostatná funkce. Funkce musí být ve formátu *f(x, t, u)*, kde *f* znázorňuje název funkce, *x* je pole veličin, pro které se přenos počítá, *t* je čas a *u* je pole vstupů. Funkce musí navracet pole derivací veličin *dx*. Do parametrů *SysF1* (*SysF2*) se pak pouze zadává odkaz na tuto funkci.

   Příklad použití – nelineární soustava s rovnicemi:

$$\begin{aligned}dx_1 &= x_2\\ dx_2 &= 6{,}8\cdot x_1+8.0\cdot x_2+u\cdot\sin(2\cdot t)\end{aligned}\tag{18}$$

   – zápis v Pythonu:

```
def funkce(x,t,u):        # nejprve je nutné definovat funkci
    dx = [0.0]*2          # vytvoření výstupního pole
    dx[0] = x[1]
    dx[1] = x[0]*6.8 + x[1]*8.0 + u[0]*sin(2*t)
    return dx

SysF = funkce             # pak lze na ni odkázat
```

5. Dynamická soustava definovaná pomocí ODE – dynamika musí být definována dvěmi funkcemi. První funkce je inicializační a vytváří se v ní dynamický model světa a těles. Musí obsahovat jeden parametr, do kterého se vkládají počáteční podmínky modelu. Druhá funkce pak slouží pro výpočet simulace modelu a musí obsahovat dva parametry. Do prvního parametru se vkládá

velikost kroku simulace, do druhého se zadávají vstupní hodnoty. Podrobnější informace pro vytváření funkcí pro výpočet simulace modelu ODE se nacházejí v kapitole 5.5.9. Do parametrů *SysF1* (*SysF2*) se pak pouze zadává pole o dvou prvcích, kde první prvek obsahuje odkaz na inicializační funkci, a druhý prvek obsahuje odkaz na funkci pro výpočet modelu.

Příklad použití – model ODE v Pythonu:

```
def ODE_init(x0):        # definice inicializační funkce
    world = ode.World()  # vytvoření světa v ODE...
    ...

def ODE_funkce(dt, u):   # definice funkce modelu ODE
    world.step(dt)       # krok výpočtu modelu
    ...
    return y             # vrátit výstup modelu

SysF = [ODE_init, ODE_funkce] # pole s odkazy na def. funkce
```

6. PID regulátor – zadává se jako pole se třemi prvky. Prvním prvkem je konstanta proporcionálního členu K_p, druhým je konstanta integračního členu K_i a posledním prvkem je konstanta derivačního členu K_d.

   Příklad použití – PID regulátor o konstantách:

$$K_p = 10, K_i = 50, K_d = 0{,}6. \tag{19}$$

– zápis v Pythonu:

```
TF = [10.0, 50.0, 0.6]
```

7. Komunikace po sériovém rozhraní – zadává se jako pole s jedním prvkem, který udává název sériového portu, po kterém program komunikuje. Odesílaná a přijímaná data jsou ve formátu textového řetězce, který vyjadřuje hodnotu čísla, zakončeného znakem pro konec řádku. Ten má podle ASCII tabulky číselnou hodnotu 10. Komunikace probíhá rychlostí 115200bps.

   Příklad použití – zápis komunikace na sériovém portu COM1 v Pythonu:

```
TF = [‘COM1‘]
```

Dalším parametrem funkce *Sim* je parametr *time*, za který se dosazuje pole časů, podle kterého se integruje výpočet průběhu signálu. Parametr *input* je pole vstupů v čase *time*. Nepovinný parametr *feedback* je typu boolean a jeho úkolem je zapínání či vypínání zpětné vazby. Defaultně je zpětná vazba vypnutá. Nepovinný parametr *x0* zadává počáteční podmínky pro oba bloky systémů. Nepovinný parametr *noise* zapíná šum za regulátorem, což je přičítání náhodných čísel k signálu v určitém rozsahu, které udává nepovinný parametr *noiseVal*. Ten je standardně nastaven na generování náhodných čísel v rozsahu od -0,01 do 0,01. Nepovinný parametr *typ* určuje, s jakým typem čísel budou pracovat jednotlivé bločky systému, buď s čísly s plovoucí desetinnou čárkou typu *float*, či s celými čísly typu *int*. Pokud je nutné použít jiný řešič diferenciálních rovnic, než je metoda Runge–Kutta, pak lze do posledního parametru zadat odkaz na funkci jiného ručně vytvořeného řešiče.

Pokud se simulace zdaří, je výstupem funkce pole výstupních hodnot pro časy *t*. Pokud se ve výpočtu objeví nějaká chyba, je výstupem řetězec oznamující informace o vzniklé chybě.

Příklad použití – zápis v Pythonu:

```
from numpy import array    # import funkce array (matice)

def sys1(x,t,u):           # definice funkce systému
    dx = [0.0]*2           # vytvoření výstupního pole
    dx[0] = x[1]           # zápis rovnic systému
    dx[1] = x[0]*6.8 + x[1]*8.0 + u[0]*sin(2*t)
    return dx              # navrácení hodnot

def Euler(f, x0, t, u):    # definice funkce Eulerovy metody řešení
    h = t[1]-t[0]
    fun = array(x0) + h * array(f(x0,t[0],u))
    return fun

SysF1 = [13, 0.02, 6.2]    # Blok 1 - PID regulátor
SysF2 = sys1               # Blok 2 - definován funkcí sys1
t = time(10, 0.1)          # vytvoření pole času 10s (skok po 0.1s)
u = step(t, 1)             # vytvoření pole vstupů pro čas t
feedback = True            # zapnutí zpětné vazby
x0 = [[0.0],[0.0]]         # počáteční podmínky
noise = True               # zapnut šum za regulátorem
nVal = [-0.2,0.2]          # velikost šumu
typ = ['float','float']    # datový typ čísel
integr = ['', Euler]       # pro výp. 2. bloku použita Eulerova metoda


# spuštění simulace
y = Sim(SysF1, SysF2, t, u, feedback, x0, noise, nVal, typ, integr)
```

II.1.4 Funkce na vykreslení grafů

plot(x, y, FileName = '', Title = 'Simulation', xLabel = 't [s]', yLabel = 'process')

Funkce vykresluje graf průběh signálu a ukládá ho do souboru jako obrázek. Parametr *x* znázorňuje hodnoty osy x (za tento parametr se dosazuje vygenerované pole času *t*), parametr *y* představuje hodnoty osy y (dosazují se zde např. vstupní či výstupní hodnoty simulace). Pokud se do nepovinného parametru *FileName* zadá název souboru, bude vygenerovaný graf do tohoto souboru uložen jako obrázek png. Nepovinný parametr *Title* obsahuje název grafu, parametr *xLabel* obsahuje popisek osy x a parametr *yLabel* obsahuje popisek osy y.

Funkce umožňuje vykreslit i průběh 2 signálů, pokud budou obsaženy v parametru *y*. Mohou zde být vyjádřeny buď formou 2 spojených polí, nebo jako pole vektorů.

Příklad použití – možné vykreslení různých signálů:

```
t = [1,2,3,4]
y = [1,2,3,4]                    # 1) vykreslení jednoho průběhu
y = [[1,2,3,4], [5,3,1,8]]       # 2) vykreslení 2 průběhů
y = [[1,5],[2,3],[3,1],[4,8]]    # 3) 2 průběhy - jiná forma zápisu
plot(t,y)                        # vykreslení grafu
```

II.1.5 Funkce pro generování C-kódu

CreateC(File, Sys, StepTime, Type = 'double')

Funkce generuje C-kód ve formě podprogramu pro výpočet výstupních hodnot, které jsou potřeba k řízení daného systému. Do parametru *Sys* se zadávají buď konstanty všech tří složek PID regulátoru , nebo přenos diskrétní přenosové funkce. Délka kroku výpočtu se zadává v sekundách do parametru *StepTime*. Do parametru *File* se zadává cesta k souboru, do kterého se vygenerovaný C-kód uloží.

Příklad použití – kompilace kódu diskrétní přenosové funkce a PID regulátoru:

```
TfZ = [[2.512, -2.5],[1, -1]]              # zadání diskrétní p.f.
CreateC('C:\main',TfZ,0.01,'double')       # kompilace – typ double

PID = [20,3,10]]                           # zadání PID regulátoru
CreateC ('C:\main',PID,0.01,'int')         # kompilace – typ integer
```

II.1.6 Funkce pro kontrolu mikrokontroléru po sériovém rozhraní

status(COM, t, input = True, output = True, signal = 0, PID = 0)

Funkce slouží ke zjištění aktuální hodnoty vstupu a výstupu na mikrokontroléru. Tyto hodnoty může zjišťovat opakovaně, a to vždy v časech, které udává pole času *t*. Parametry *input* a *output* jsou typu *boolean*. Pokud bude hodnota parametru *input* rovná hodnotě *True*, budou se zjišťovat vstupy mikrokontroléru. Pokud bude hodnota parametru *output* rovná hodnotě *True*, budou se zjišťovat jeho výstupy. Do nepovinného parametru *signal* lze zadat pole požadovaných vstupních hodnot, které jsou zapotřebí pro řízení pomocí mikrokontroléru. Do parametru *COM* se zapisuje název sériového rozhraní, po kterém program komunikuje s mikrokontrolérem. Do posledního nepovinného parametru *PID* lze zapsat konstanty PID regulátoru, které se na začátku vysílání pošlou do mikrokontroléru, čímž se původní konstanty přepíší.

Komunikace s mikrokontrolérem probíhá následovně. Program vyšle pro zjištění vstupu znak 'u' a znak pro ukončení řádku, který má v ASCII tabulce hodnotu 10. Pak očekává odezvu od mikrokontroléru ve formě řetězce obsahujícího hodnotu vstupu a ukončeného znakem pro ukončení řádku. Při zjišťování výstupu komunikace probíhá obdobně, jen program vyšle místo znaku 'u' znak 'y'. Při změně požadované vstupní hodnoty program vyšle znak 'h' a za ním požadovanou hodnotu převedenou na řetězec se znakem pro ukončení řádku. Výstupem funkce je tabulka načtených hodnot, které lze dále vykreslit funkcí *plot*.

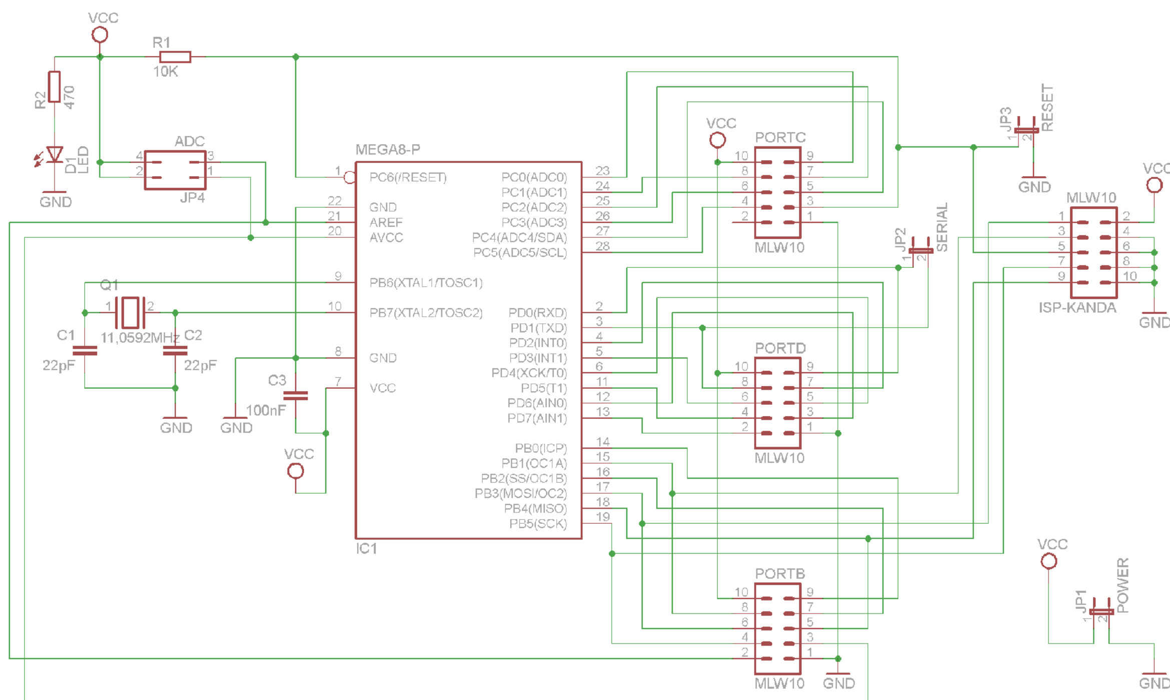
Příklad použití – načítání hodnot z mikrokontroléru:

```
t = time(5,0.1)                            # vytvoření pole času

# Po dobu 5s každých 0.1s načíst z mikrokontroléru připojeného na
# COM1 hodnotu pouze z výstupu
y = status('COM1', t, False, True)

plot(x,y)                                  # vykreslení hodnot
```

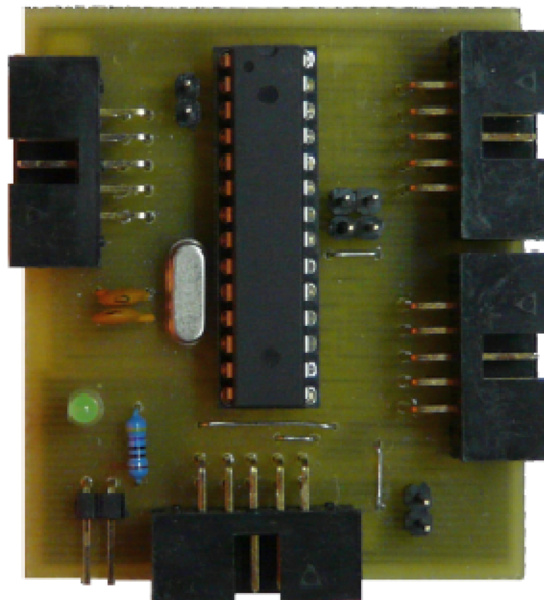
III PRÁCE S MIKROKONTROLÉRY

III.1 Modul s mikrokontrolérem ATmega8

K tomuto univerzálnímu modulu pro mikrokontrolér ATmega8 se přes konektory MLW10 připojoval modul pro sériovou komunikaci s počítačem a modul s výkonovým členem, pomocí něhož se převáděl PWM signál na napětí napájecí motor.



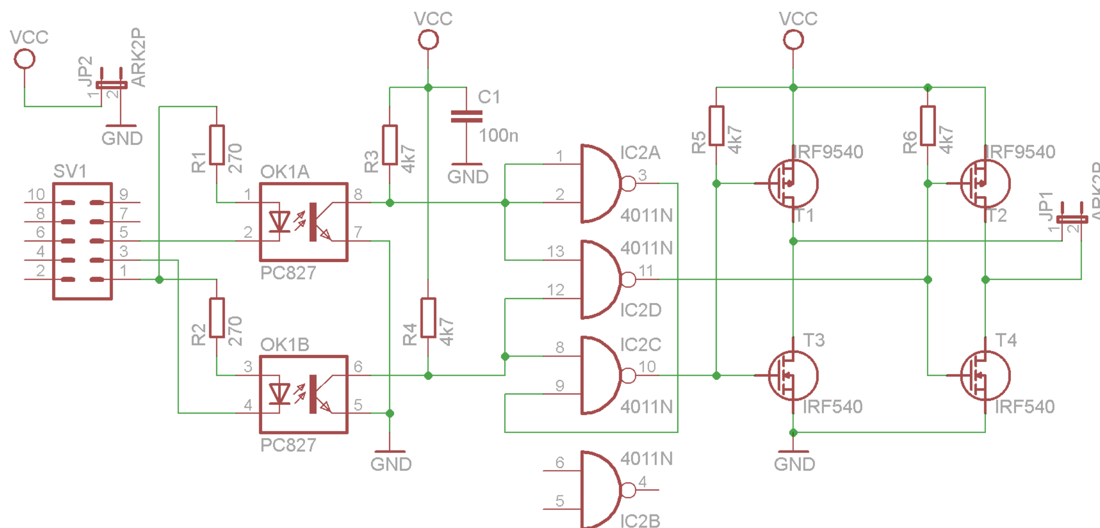
Obr. 27 Schéma modulu s mikrokontrolérem ATmega8



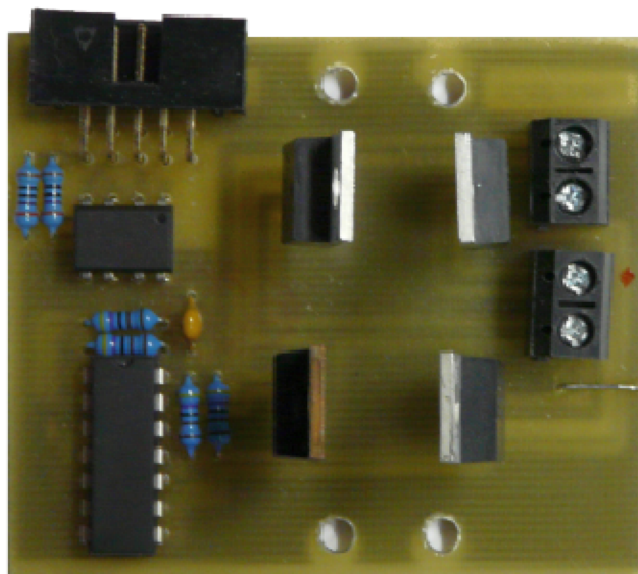
Obr. 28 Modul s mikrokontrolérem ATmega8

III.2 Modul výkonového členu pro řízení motoru

Na Obr. 29 se nachází schéma modulu s výkonovým členem, pomocí něhož se převádí PWM signál na napětí napájecí motor. Na konektoru SV1 na pinu 1 je napájecí napětí z modulu mikrokontrolérů +5V, z pinu 3 přichází z mikrokontroléru PWM signál a na pinu 5 přichází logická hodnota určující směr otáčení motoru.



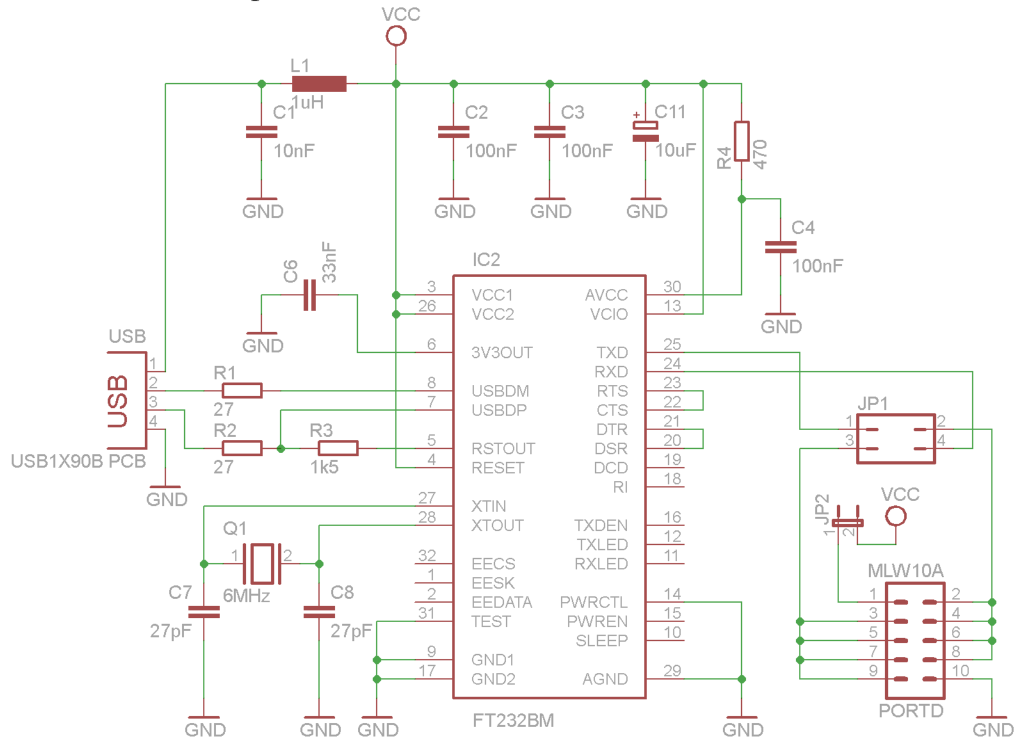
Obr. 29 Schéma výkonového modulu pro řízení motoru



Obr. 30 Výkonový modul pro řízení motoru

III.3 Modul pro sériovou komunikaci

Modul pro sériovou komunikaci s počítačem je řešen pomocí integrovaného obvodu FT232BM, který převádí univerzální sériový signál vycházející z mikrokontroléru na signál sériové komunikace po USB rozhraní.

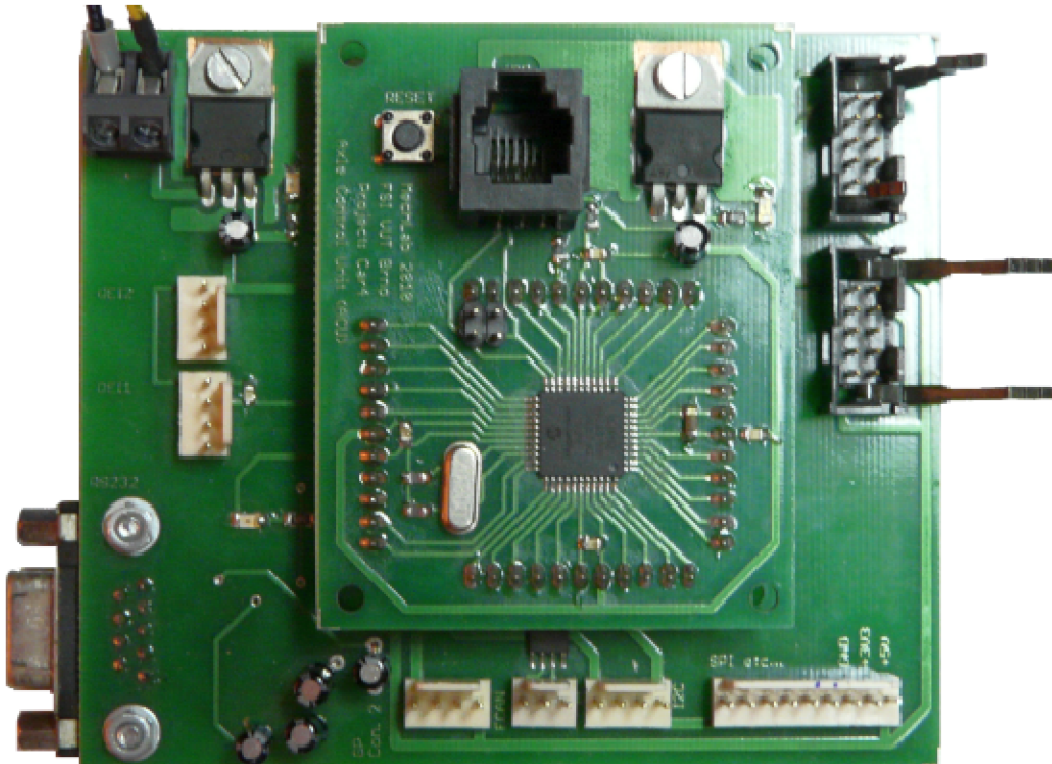


Obr. 31 Schéma modulu pro sériovou komunikaci mezi PC a mikrokontrolérem s USB rozhraním

Obr. 32 Modul pro sériovou komunikaci mezi PC a mikrokontrolérem s USB rozhraním

III.4 Modul s mikrokontrolérem dsPIC

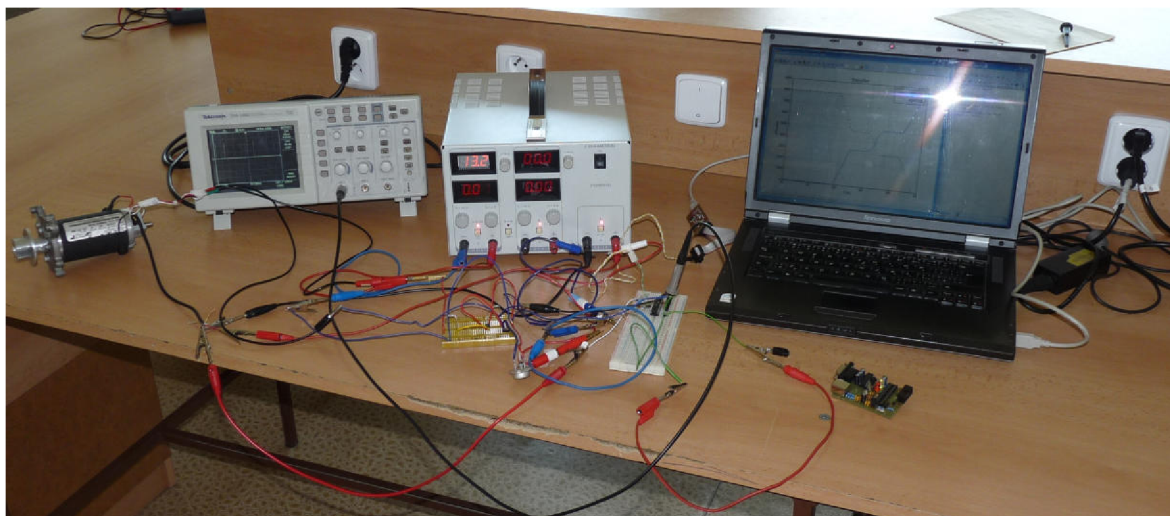
Na vyzkoušení regulace pomocí mikrokontroléru dsPIC33FJ128MC804 byl v laboratoři Mechlab vypůjčen modul *Axis Control Unit* obsahující zmíněný mikrokontrolér.



Obr. 33 Modul s mikrokontrolérem dsPIC33FJ128MC804

III.5 Pracoviště

Na Obr. 34 se nachází fotka testovacího pracoviště řízení stejnosměrného motoru.



Obr. 34 Testovací pracoviště