VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

NUMERICKÝ INTEGRÁTOR NA PLATFORMĚ .NET

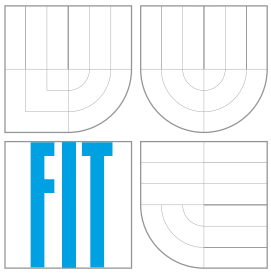
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JIŘÍ KOPECKÝ

BRNO 2012

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

NUMERICKÝ INTEGRÁTOR NA PLATFORMĚ .NET

PLATFORM .NET FOR NUMERICAL INTEGRATION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JIŘÍ KOPECKÝ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. VÁCLAV ŠÁTEK, Ph.D.

BRNO 2012

Abstrakt

Tato bakalářská práce se zabývá numerickým řešením soustav obyčejných diferenciálních rovnic 1. řádu. V první části práce jsou popsány vybrané jednokrokové integrační metody. Druhá část práce se věnuje jazyku pro popis diferenciálních rovnic. Nejprve popisuje zkoumání jazyků systémů MATLAB, Maple a TKSL/386. Na základě těchto znalostí byl následně navržen jazyk nový. Předposlední část práce se věnuje návrhu a implementaci systému určeného pro výpočet soustav diferenciálních rovnic. V poslední části je pak ukázáno použití tohoto systému při řešení příkladů z oblasti teorie obvodů.

Abstract

This bachelor thesis deals with numeric solution of ordinary first-order differential equations and their systems. The first part of this thesis contains description of selected one-step integration methods. The second part is devoted to a language intended for differential equation notation. This part at first describes the study of languages of MATLAB, Maple and TKSL/386 simulation systems. Later, based on this study it presents a design of a new language. The penultimate part of the thesis deals with design and implementation of a system intended for the calculation of systems of differential equations. In the final part is then shown usage of this system to solve exercises from the Circuits Theory domain.

Klíčová slova

diferenciální rovnice, numerické řešení diferenciálních rovnic, MATLAB, spojitá simulace, problémy s počáteční podmínky, numerické integrační metody

Keywords

differential equations, numeric solution of differential equations, MATLAB, continuous simulation, initial condition problem, numeric integration methods

Citace

Jiří Kopecký: Numerický integrátor na platformě .NET, bakalářská práce, Brno, FIT VUT v Brně, 2012

Numerický integrátor na platformě .NET

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Václava Šátka, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jiří Kopecký
15. května 2012

Poděkování

Na tomto místě bych rád poděkoval Ing. Vlastimilu Kalužovi, díky kterému vzniklo zadání této práce. Dále bych chtěl poděkovat vedoucímu práce Ing. Václavu Šátkovi, Ph.D., dřívější vedoucí Ing. Pavle Sehnalové, Ph.D. a Ing. Václavu Vopěnkovi za poskytnutí mnoha cenných rad a připomínek. Velký dík také patří Martinu Kučerovi, který se mnou spolupracoval v rámci své bakalářské práce na tvorbě systému Integrátor.NET. V neposlední řadě bych také chtěl poděkovat své rodině a přátelům za podporu a pomoc po celou dobu studia.

Obsah

Kapitola 1

Úvod

Práce se zabývá numerickým řešením obyčejných diferenciálních rovnic 1. řádu. Pomocí diferenciálních rovnic popisujeme mnoho dějů především ve fyzice a elektrotechnice. Řešení těchto úloh se provádí nejčastěji numericky. Cílem této práce je navrhnout a implementovat systém, který tyto výpočty usnadní. Velký důraz je kladen především na vytvoření nového jazyka pro popis soustav diferenciálních rovnic 1. řádu, který bude sloužit jako vstup systému.

Pro pochopení celé práce je nutné seznámit čtenáře s nezbytnou teorií. Tato zahrnuje především různé typy diferenciálních rovnic a jejich řešení. Dále pak numerické integrační metody pro řešení obyčejných diferenciálních rovnic 1. řádu a jejich vlastnosti. Tuto teorii předloží čtenáři kapitola 2.

Kapitola 3 se nejprve zabývá již existujícími jazyky, které slouží pro zápis diferenciálních rovnic. Následně představí návrh nového jazyka, který má sloužit jako vstup simulačního systému Integrátor.NET.

Návrhem a implementací simulačního systému Integrátor.NET se zabývá kapitola 4. Nejprve bude představen návrh systému jako celku, poté jednotlivých komponent a v druhé části této kapitoly bude čtenář seznámen s implementací tohoto systému.

V kapitole 5 jsou pak pomocí systému Integrátor.NET řešeny příklady z oblasti teorie obvodů. V této kapitole je také srovnání výstupu ze systému Integrátor.NET s již existujícím řešením.

Kapitola 2

Diferenciální rovnice

Diferenciální rovnice využíváme v mnoha oborech lidské činnosti. Velmi širokou oblast jejich uplatnění představuje popis fyzikálních a chemických dějů. Text této kapitoly čerpá především z [1], [2] a [11].

2.1 Typy a řád diferenciální rovnice

Rozlišujeme obyčejné diferenciální rovnice a parciální diferenciální rovnice. Obyčejné diferenciální rovnice obsahují nezávisle proměnnou t a neznámou funkci $y = y(t)$ včetně jejích derivací. Parciální diferenciální rovnice pak obsahují derivace podle více nezávislých proměnných. Řádem diferenciální rovnice pak myslíme řád nejvyšší derivace, která se v rovnici vyskytuje.

V následujícím textu se, pokud nebude uvedeno jinak, budeme zabývat pouze obyčejnými diferenciálními rovnicemi 1. řádu a jejich soustavami.

2.2 Analytické řešení diferenciálních rovnic

Analytické řešení je možné nalézt pouze pro některé typy diferenciálních rovnic. V reálném prostředí je však většinou potřeba řešit složitější rovnice, jejichž analytické řešení buď neznáme, nebo by bylo natolik složité, že není vhodné jej použít. Více informací o analytickém řešení diferenciálních rovnic lze najít například v [8].

2.3 Numerické řešení obyčejných diferenciálních rovnic

K řešení složitějších diferenciálních rovnic a jejich soustav se z těchto důvodů používají metody numerické. Tyto metody řešení aproximují – tedy hledají přibližné řešení. Při numerickém řešení problémů je tedy nutné toto uvážit a určit jak přesný výsledek potřebujeme. Od toho se poté odvíjí volba metody a dalších parametrů výpočtu.

Společným znakem všech dále uvedených metod je, že řešení nehledáme jako spojitou funkci, ale jako diskrétní funkci definovanou v konečném počtu bodů $t_n, n = 0, 1, \ldots, N$. Tyto body budeme volit ekvidistantní, tj. položíme

$$t_n = a + nh, \quad n = 0, 1, \ldots, N\,, \tag{2.1}$$

kde h je konstanta, kterou nazveme *integrační krok*.

2.3.1 Vlastnosti numerických metod

Všechny dále popsané metody mají určité vlastnosti, které určují vhodnost metody pro řešení konkrétních problémů.

Globální a lokální chyby

Při výpočtu pomocí numerických metod je třeba dbát několika typů chyb, které se mohou vyskytnout.

Jestliže $y(t)$ je přesné řešení dané úlohy, pak rozdíl

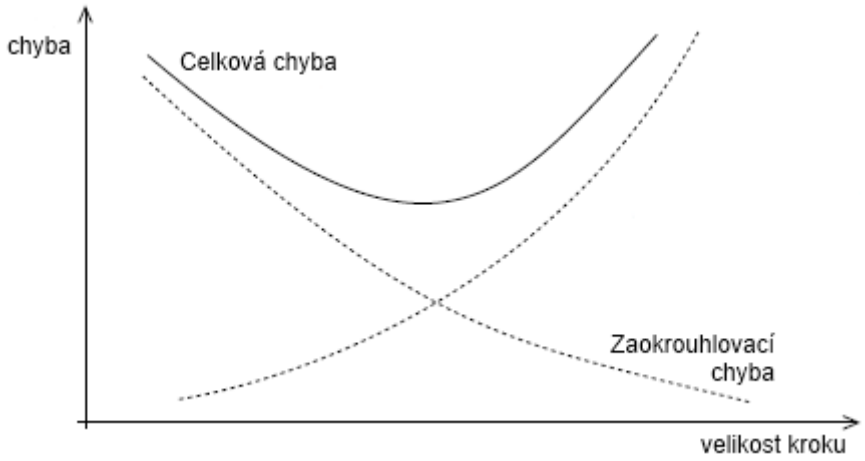
$$e_n = y_n - y(t_n) \tag{2.2}$$

nazýváme globální chybou aproximace nebo globální diskretizační chybou. Tato chyba představuje nakupení chyb tzv. lokálních. Lokální diskretizační chyba představuje chybu, která vznikla v rámci výpočtu jednoho kroku metody za předpokladu, že všechny hodnoty nutné pro výpočet tohoto kroku jsou přesné.

Jiný typ chyby je způsoben tím, že čísla y_n většinou nemůžeme počítat s libovolnou přesností. Pokud označíme $\tilde{y}_n$ čísla, která jsou skutečně počítána místo y_n, pak rozdíl

$$r_n = \tilde{y}_n - y_n \tag{2.3}$$

nazýváme zaokrouhlovací chybou. Celková chyba je závislá na obou těchto faktorech a délce kroku metody. Tuto závislost ilustruje obrázek 2.1. Z tohoto poznatku vyplývá, že pro každou metodu existuje určitý optimální rozsah velikosti kroku. Při volbě menšího kroku by již vlivem zaokrouhlovacích chyb rostla globální chyba výpočtu.



Obrázek 2.1: Závislost chyby metody na délce kroku

Stabilita metody

Stabilita metody vyjadřuje chování metody při řešení některých problémů v závislosti na délce kroku. Pokud krok zvětšíme a chování metody se začne skokově měnit, označujeme toto řešení jako nestabilní – výsledky jsou zcela chybné. Pro některé metody a typy problémů nemusí stabilní řešení vůbec existovat.

Explicitní metody

Explicitní metody se vyznačují tím, že pro výpočet nového kroku y_{i+1} používají pouze hodnotu jednoho či více kroků předchozích. Jinými slovy se ve výrazu pro výpočet kroku y_{i+1} vyskytuje maximálně hodnota y_i.

Implicitní metody

Implicitní metody používají při výpočtu nové hodnoty aproximace tuto neznámou hodnotu. Jsou tedy výpočetně náročnější než metody jednokrokové (je třeba numericky řešit rovnici pro výpočet nové aproximace). Implicitní metody jsou však při stejné velikosti integračního kroku přesnější než metody explicitní a zároveň jsou implicitní metody stabilnější.

Řád metody

Definici řádu metody převezmeme z [1]:

> Pro popis rychlosti konvergence metody používáme pojem **řád metody**. Zhruba řečeno je řád metody přirozené číslo p takové, že pro malá h je lokální diskretizační chyba d_i řádově velikosti h^{p+1}. U jednokrokových metod p-tého řádu lze dokázat, že globální diskretizacní chyba je řádově velikosti h^p.

2.3.2 Explicitní Eulerova metoda

Jedná se o nejjednodušší jednokrokovou integrační metodu. Díky tomu je vhodná spíše pro demonstraci principů platných i pro složitější metody, než pro skutečný výpočet. Explicitní Eulerova metoda je definována rekurentním vztahem

$$y_{n+1} = y_n + hf(t_n, y_n), \quad n = 0, 1, 2, \ldots, N-1 \tag{2.4}$$

přičemž platí počáteční podmínka

$$y_0 = c.$$

Výpočet pomocí Eulerovy metody je velmi jednoduchý. Nejlépe ho demonstrujeme na následujícím jednoduchém příkladu.

Příklad použití explicitní Eulerovy metody

Explicitní Eulerovou metodou s krokem $h = 0{,}1$ řešte úlohu

$$y' = t^2 - y, \qquad y_0 = 1$$

na intervalu $\langle 0; 0{,}5 \rangle$.

Řešení: V našem případě je $t_0 = 0$, $y_0 = 1$ a $f(t, y) = t^2 - y$. Přibližné hodnoty řešení v dalších bodech budeme počítat podle vzorce (2.4), konkrétně

$$y_{i+1} = y_i + 0{,}1 \cdot (t_i^2 - y_i), \qquad i = 0, 1, \ldots, 5$$

Hodnoty prvního kroku již známe ze zadání. Konkrétně to jsou

$$t_0 = 0$$
$$y_0 = y(0) = 1$$

Hodnoty pro další krok vypočteme prostým dosazením do vzorce (2.4).

$$t_1 = 0{,}1$$
$$y_1 = y_0 + 0{,}1 \cdot (t_0^2 - y_0) = 1 + 0{,}1 \cdot (0 - 1) = 0{,}9$$

Pro další kroky postupujeme stejným způsobem. Vypočtené výsledky zapisujeme do tabulky. Všechny výsledky jsou zaokrouhleny na 4 desetinná místa.

i	0	1	2	3	4	5
t_i	0	0,1	0,2	0,3	0,4	0,5
y_i	1	0,9	0,811	0,7339	0,6695	0,6186

2.3.3 Metoda Taylorova rozvoje

V některých případech je vhodné využít tzv. metodu Taylorova rozvoje. Předpokládejme zadání rovnice $y' = f(t, y)$. Je li funkce f diferencovatelná do dostatečně vysokého řádu, lze psát

$$\begin{aligned} y'' &= \tfrac{\mathrm{d}f}{\mathrm{d}t}(t,y) = f_t(t,y) + f_y(t,y)y' = \\ &= f_t(t,y) + f_y(t,y)f(t,y)\,, \end{aligned} \tag{2.5}$$

kde index t resp. y znamená parciální derivaci vzhledem k t resp. y. Třetí derivaci y''' lze analogicky získat

$$y''' = f_{tt} + 2ff_{ty} + f_{yy}f^2 + f_x f_y + f f_y^2 \tag{2.6}$$

atd. Přírůstek řešení $y(t)$ na intervalu $\langle t, t+h\rangle$ lze poté napsat ve tvaru Taylorova rozvoje:

$$\begin{aligned} y(t_n + h) &\doteq y_{n+1} = \\ &= y_n + hy'(t_n) + \tfrac{h^2}{2}y''(t_n) + \cdots + \tfrac{h^p}{p!}y^{(p)}(t_n) + \mathcal{O}(h^{p+1})\,. \end{aligned} \tag{2.7}$$

Metoda (2.7) se nazývá metoda Taylorova rozvoje p-tého řádu. Její globální chyba je řádu p, tedy $\mathcal{O}(h^p)$.

Pro obecné typy rovnic však není tato metoda příliš vhodná, protože analytické derivování může být značně pracné pro $p > 3$. Nutným požadavkem pro použití této metody je také existence alespoň p-tého řádu derivace funkce $y(t)$.

Dosazením $p = 1$ do rovnice (2.7) dostaneme explicitní Eulerovu metodu popsanou v části 2.3.2.

Podrobnější informace o metodě Taylorova rozvoje a jejích použitích lze nalézt např. v [11].

2.3.4 Runge-Kuttovy metody

Jedná se o jednu z nejdůležitějších a nejpoužívanějších skupin jednokrokových metod. Obecný tvar Runge-Kuttovy metody je

$$y_{n+1} = y_n + h(\gamma_1 k_1 + \cdots + \gamma_s k_s)\,, \tag{2.8}$$

kde

$$\begin{aligned} k_1 &= f(t_n, y_n) \\ k_i &= f(t_n + \alpha_i h, y_n + h\sum_{j=1}^{i-1}\beta_{ij}k_j), \quad i = 2, \ldots, s \end{aligned} \tag{2.9}$$

a γ_i, α_i a β_{ij} jsou konstanty volené tak, aby metoda měla maximální řád. Více informací o metodách z rodiny Runge-Kutta lze nalézt např. v [1].

Schéma Runge-Kuttových metod

Pro přehledné vyjádření Runge-Kuttových metod se používá speciální schéma. Obecná podoba tohoto schématu je uvedena v tabulce 2.1.

α_1	β_{11}				
α_2	β_{21}	β_{22}			
α_3	β_{31}	β_{32}	β_{33}		
$\vdots$					
α_j	β_{j1}	β_{j2}	$\cdots$	β_{jj}	
	γ_1	γ_2	$\cdots$	γ_j	γ_{j+1}

Tabulka 2.1: Obecné schéma Runge-Kuttových metod

Tabulka 2.2 pak uvádí konkrétní podobu tohoto schématu pro explicitní Eulerovu metodu.

0	
	1

Tabulka 2.2: Schéma Eulerovy metody

2.3.5 Heunova metoda

V [2] je uváděna jako zlepšená Eulerova metoda. Na rozdíl od klasické Eulerovy metody počítá hodnotu funkce v dalším kroku pomocí dvou bodů a ne jen jednoho. Jedná se o explicitní metodu druhého řádu. Hodnotu aproximace v dalším kroku počítá pomocí dvou bodů. Předpis této metody je uveden v soustavě rovnic (2.10). Zjednodušený zápis pomocí schématu pak představuje tabulka 2.3.

$$\begin{aligned} k_1 &= hf(t_n, y_n) \\ k_2 &= hf(t_n + h, y_n + k_1) \\ y_{n+1} &= y_n + \tfrac{1}{2}(k_1 + k_2) \end{aligned} \tag{2.10}$$

1	1	
	$\frac{1}{2}$	$\frac{1}{2}$

Tabulka 2.3: Schéma Heunovy metody

2.3.6 Metoda Runge-Kutta 3. řádu

První ze zde uvedených metod, která se používá v praxi. Předpis metody uvádí soustava rovnic (2.11). Zjednodušený zápis pomocí schématu je uveden v tabulce 2.4.

$$\begin{aligned} k_1 &= f(t_n, y_n) \\ k_2 &= f(t_n + \tfrac{1}{3}h, y_n + \tfrac{1}{3}hk_1) \\ k_3 &= f(t_n + \tfrac{2}{3}h, y_n + \tfrac{2}{3}hk_2) \\ y_{n+1} &= y_n + \tfrac{1}{4}h(k_1 + 3k_3) \end{aligned} \tag{2.11}$$

1/3	1/3		
2/3	0	2/3	
	1/4	0	3/4

Tabulka 2.4: Schéma metody Runge-Kutta 3. řádu

2.3.7 Metoda Runge-Kutta 4. řádu

Jedná se o nejznámější metodu z rodiny Runge-Kutta. Při zmínce o Runge-Kuttově metodě bez udání řádu je většinou myšlena tato metoda. Předpis metody Runge-Kutta 4. řádu je uveden v soustavě rovnic (2.12), zápis pomocí schématu pak v tabulce 2.5

$$\begin{aligned} k_1 &= f(t_n, y_n) \\ k_2 &= f(t_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}hk_1) \\ k_3 &= f(t_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}hk_2) \\ k_4 &= f(t_n + h, y_n + hk_3) \\ y_{n+1} &= y_n + \tfrac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned} \tag{2.12}$$

1/2	1/2			
1/2	0	1/2		
1	0	0	1	
	1/6	1/3	1/3	1/6

Tabulka 2.5: Schéma metody Runge-Kutta 4. řádu

2.3.8 Implicitní Eulerova metoda

Implicitní Eulerova metoda dosahuje lepších výsledků a stability, než explicitní Eulerova metoda popsaná části 2.3.2. Předpis implicitní Eulerovy metody je uveden v rovnici (2.13).

$$y_{i+1} = y_i + h \cdot f(t_{i+1}, y_{i+1}) \tag{2.13}$$

Oproti vzorci (2.4) se změnil parametr funkce f, který u implicitní varianty metody obsahuje právě počítanou hodnotu aproximace. Z tohoto důvodu je nutné v každém kroku numericky vyřešit tuto další rovnici. Přínosem implicitní metody je však větší stabilita, jak již bylo uvedeno v části 2.3.1.

2.3.9 Příklad numerického výpočtu

Nyní si ukážeme způsob výpočtu pomocí numerické integrační metody Runge-Kutta 4. řádu. Princip výpočtu je však pro všechny explicitní jednokrokové metody stejný, a tedy zde uvedený postup je možné aplikovat na libovolnou metodu.

Mějme následující zadání:

$$\begin{aligned} y' &= -y \\ y(0) &= 1 \\ y(0.02) &= ? \end{aligned}$$

Zvolíme krok integrační metody $h = 0{,}01$. Číslo kroku označíme i. Do tabulky řešení zaznamenáme počáteční hodnoty proměnných.

i	t	y
0	0	1
1		
2		

Nyní podle vztahu (2.12) vypočítáme hodnotu y v čase t_1.

$$k_1 = f(t_0, y_0) = -1$$
$$k_2 = f(t_0 + \tfrac{1}{2}h, y_0 + \tfrac{1}{2}hk_1) = -\left(1 + \tfrac{1 \cdot 0{,}01}{2}\right) = -1{,}005$$
$$k_3 = f(t_0 + \tfrac{1}{2}h, y_0 + \tfrac{1}{2}hk_2) = -\left(1 + \tfrac{-1{,}005 \cdot 0{,}01}{2}\right) = -0{,}994975$$
$$k_4 = f(t_0 + h, y_0 + hk_3) = -\left(1 - 0{,}994975 \cdot 0{,}01\right) = -0{,}99005025$$
$$y_1 = y_0 + \tfrac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4) = 1 + \tfrac{0{,}01}{6}\left(-1 - 1{,}005 - 0{,}994975 - 0{,}99005025\right) =$$
$$= 0{,}9933499579$$

Výsledek zaznamenáme do tabulky řešení a stejným způsobem vypočteme hodnotu y v čase t_2.

i	t	y
0	0	1
1	0,01	0,9933499579
2		

$$k_1 = f(t_1, y_1) = 0{,}9933499579$$
$$k_2 = f(t_1 + \tfrac{1}{2}h, y_1 + \tfrac{1}{2}hk_1) = -\left(0{,}9933499579 + \tfrac{0{,}9933499579 \cdot 0{,}01}{2}\right) = -0{,}9983167077$$
$$k_3 = f(t_1 + \tfrac{1}{2}h, y_1 + \tfrac{1}{2}hk_2) = -\left(0{,}9933499579 + \tfrac{-0{,}9983167077 \cdot 0{,}01}{2}\right) = -0{,}9883583744$$
$$k_4 = f(t_1 + h, y_1 + hk_3) = -\left(0{,}9933499579 - 0{,}9883583744 \cdot 0{,}01\right) = -0{,}9834663742$$
$$y_2 = y_1 + \tfrac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4) =$$
$$= 0{,}9933499579 + \tfrac{0{,}01}{6}(0{,}9933499579 - 0{,}9983167077 - 0{,}9883583744 -$$
$$- 0{,}9834663742) = 0{,}9900553054$$

i	t	y
0	0	1
1	0,01	0,9933499579
2	0,02	0,9900553054

Výsledek zaznamenáme do tabulky. Tento příklad mimo jiné demonstruje, že numerický výpočet diferenciálních rovnic je velmi pracné provádět ručně.

Kapitola 3

Návrh jazyka pro popis ODR

Klíčovým prvkem celého systému je vstupní jazyk pro popis obyčejných diferenciálních rovnic 1. řádu. Při návrhu tohoto jazyka jsem nejprve prostudoval již existující jazyky systémů MATLAB, Maple a TKSL/386. Následující kapitola obsahuje poznatky získané studiem těchto jazyků a také návrh jazyka pro simulační systém Integrátor.NET.

3.1 Programový vstup systému MATLAB

Systém MATLAB (**mat**rix **lab**oratory) od společnosti MathWorks je programovací prostředí pro vývoj algoritmů, analýzu dat, jejich vizualizaci a numerické výpočty. V současné době je k dispozici verze pro všechny tři majoritní operační systémy (Windows, Linux, MacOS) a to pro jejich 32bitové i 64bitové varianty. Více informací o tomto systému lze nalézt na [6].

Výpočet obyčejných diferenciálních rovnic 1. řádu je pouze jedna z mnoha funkcí systému MATLAB. Tomu také odpovídá jeho programový vstup, který je velmi komplexní. Pro numerický výpočet diferenciálních rovnic slouží v systému MATLAB příkazy z rodiny `ode` (např. `ode23` a `ode45`). Číslo na konci názvu metody udává řády metod, které se při výpočtu použijí. Syntaxe použití metod z rodiny `ode` závisí na použitém počtu parametrů a typech výstupu. Zde je uvedena nejjednodušší forma zápisu, kde se nepoužívají doplňující volby.

```
[T,Y] = solver(odefun,tspan,y0)
```

Tento příkaz předpokládá zadání rovnice v explicitním tvaru, tedy $y' = f(t, y)$. Význam jednotlivých parametrů je následující:

`T`	Vektor časů, pro které byl proveden výpočet
`Y`	Vektor hodnot vypočtených v čase na odpovídající pozici ve vektoru `T`
`solver`	Název metody z rodiny `ode` (např. `ode45`)
`odefun`	Ukazatel na funkci, která provede vyhodnocení pravé strany zadané rovnice
`tspan`	Vektor určující časový rámec integrace. Integrátor předpokládá, že počáteční podmínky platí pro `tspan(1)`
`y0`	Vektor počátečních podmínek.

3.1.1 Příklad výpočtu

Zadání

Vyřešte numericky diferenciální rovnici

$$y' = y, y(0) = 1,$$

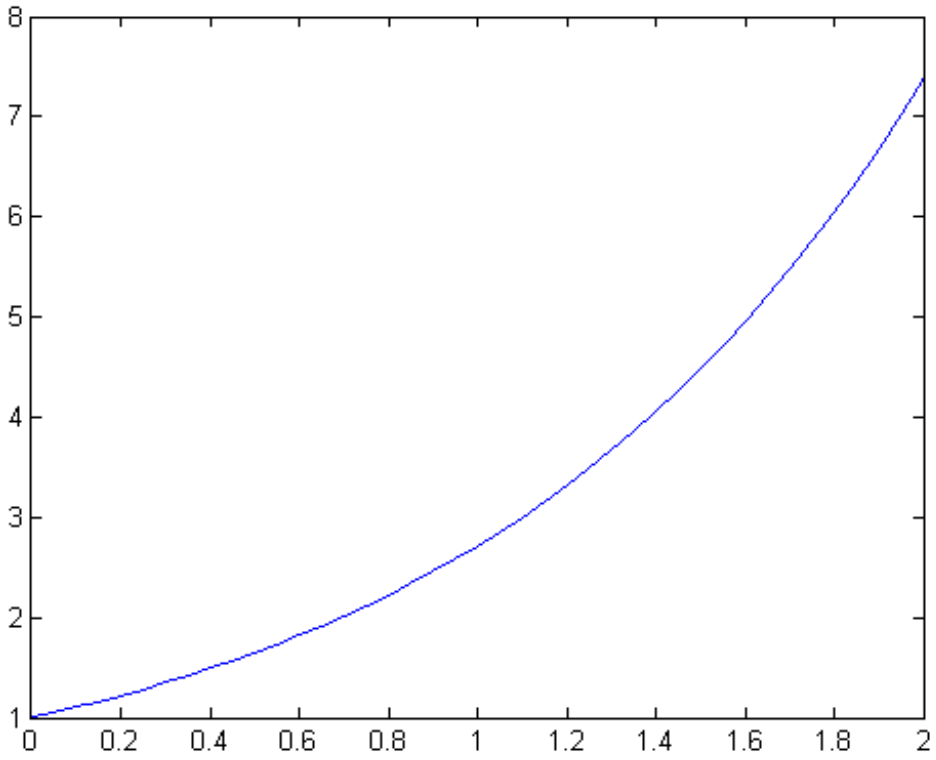
s nezávisle proměnnou $t \in \langle 0, 2 \rangle$. Výsledek prezentujte formou grafu funkce $f(t) = y$.

Řešení v systému MATLAB

Toto zadání je v systému MATLAB možné vyřešit následující posloupností příkazů:

```
fn = @(t,y) y;
[T, Y] = ode45(fn, [0 2], [1]);
plot(T, Y);
```

Výsledný graf je pak zobrazen na obrázku 3.1



Obrázek 3.1: Řešení v systému MATLAB

Výsledek je také možné uložit tak, aby bylo možné s daty dále pracovat. Vhodným formátem pro uložení dat je CSV, neboť jej umí zpracovat mnoho nástrojů. Export dat ve formátu CSV provedeme příkazem

```
csvwrite('soubor.csv', [T Y]);
```

3.1.2 Shrnutí vlastností systému MATLAB

Následující seznam shrnuje hlavní vlastnosti systému MATLAB vzhledem k zápisu diferenciálních rovnic a jejich numerickému řešení.

- Diferenciální rovnice se nezapisují přímo, ale jako funkce jejich pravé strany.
- Při výpočtu je možné zvolit integrační metody.
- Při použití metod z rodiny `ode` si integrační krok určuje systém sám, není možné použít fixní krok.
- Výsledky jsou vráceny formou vektorů (samostatný vektor pro časové údaje).
- Graf je nutné explicitně vytvořit.
- Výsledky je možné exportovat ze systému např. ve formátu CSV.

3.2 Vstupní jazyk systému Maple

Maple od kanadské společnosti MapleSoft je matematický systém určený pro výuku a využití algebry. Umožňuje symbolické i numerické výpočty. Stejně jako systém MATLAB je i Maple univerzální software, který umožňuje řešit nejrůznější matematické problémy.

K řešení diferenciálních rovnic a jejich soustav slouží v systému Maple příkaz `dsolve`. Ve výchozím nastavení řeší tento příkaz zadanou rovnici symbolicky. Pro numerické řešení je třeba uvést parametr `numeric`. Detailní informace o tomto příkazu a všech jeho podporovaných zápisech se nachází v manuálu systému na [4].

3.2.1 Příklad výpočtu

Zde provedeme výpočet stejného příkladu jako v části 3.1.1 pomocí systému Maple.

Řešení v systému Maple

Zadání přepíšeme do kódu systému Maple.

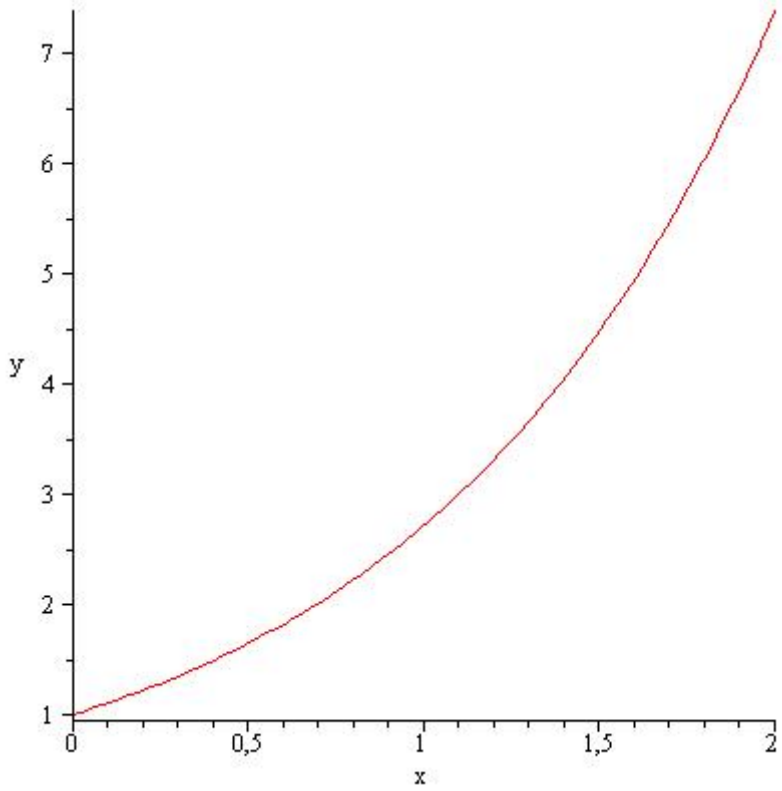
```
with(plots);
ode := {y(0) = 1, (D(y))(x) = y(x)};
s := dsolve(ode, type = numeric, range = 0 .. 2);
odeplot(s);
```

První řádek zavede knihovnu `plots`, která obsahuje příkaz `odeplot`. Na druhém řádku je popsána řešená diferenciální rovnice. Na třetím řádku je pak proveden numerický výpočet. Graf vytvořený příkazem `odeplot` je zobrazen na obrázku 3.2.

3.2.2 Shrnutí vlastností systému Maple

Následující seznam shrnuje hlavní vlastnosti systému Maple vzhledem k zápisu diferenciálních rovnic a jejich numerickému řešení.

- Diferenciální rovnice se zapisují přímo. V prostředí Maple lze využít přehledný symbolický zápis, avšak tento zápis není čistě textový.



Obrázek 3.2: Řešení v systému Maple

- Při výpočtu je možné zvolit integrační metody.
- Výsledek je vrácen formou funkce. Tuto funkci lze snadno zobrazit v grafu.
- Data z výsledků není možné intuitivně exportovat ze systému.

3.3 Vstupní jazyk systému TKSL/386

Simulační systém TKSL/386 vytvořil Doc. Ing. Jiří Kunovský, CSc. na Fakultě informačních technologií VUT v Brně. Jeho účelem je testování algoritmů využívajících Taylorovu řadu pro řešení diferenciálních rovnic. Tento systém je vytvořen v prostředí TurboVision, které je určeno pro 16bitové systémy DOS a Windows. Díky tomu není možné tento systém na moderních 64bitových systémech spustit nativně a je nutné používat emulátor DOSBox. Stejný emulátor je pak možné využít pro spuštění systému na platformách Linux a MacOS.

Vstupní jazyk tohoto systému je pak přímo navržen pro popis obyčejných diferenciálních rovnic 1. řádu. Rovnice se zadávají v explicitním tvaru a počáteční podmínka se uvádí u každé rovnice zvlášť za znakem `&`. Jako nezávislá proměnná je v systému brán simulační čas `t`. Simulace probíhá vždy od $t_0 = 0$ do času určeného konstantou `tmax`. Další parametry simulace určují konstanty `dt` a `eps`. Konstanta `dt` určuje integrační krok výpočtu a konstanta `eps` určuje přesnost výpočtu. Krok integrační metody není pevný. V každé iteraci je velikost kroku upravena tak, aby byla zachována požadovaná přesnost. Po dokončení simulace je ihned zobrazen graf časového průběhu proměnných uvedených na levých stranách zadaných rovnic. Výsledky simulace je též možné zobrazit formou tabulky.

Struktura zdrojového souboru je velmi podobná struktuře zdrojových souborů v jazyce PASCAL. Všechny použité proměnné musejí být deklarovány v hlavičce programu. Tamtéž se také definují konstanty. Řešená rovnice nebo soustava rovnic se pak zapíše mezi závorky `system` a `sysend`. Za závorkou `sysend` se vkládá tečka, která označuje konec programu.

3.3.1 Příklad výpočtu

Zde provedeme výpočet stejného příkladu jako v části 3.1.1, nyní ale s pomocí systému TKSL/386.

Řešení v systému TKSL/386

Zadání převedeme na vstupní program pro TKSL/386.

```
var y;
const t0=0,dt=0.1,tmax=2,eps=1e-20;
system
y'= y & 1;
sysend.
```

Tento program v systému TKSL/386 nejdříve zkompilujeme (F9) a následně spustíme (Ctrl+F9). Zobrazí se nám graf časového průběhu funkce $f(x) = y$, který je zobrazen na obrázku 3.3. Výsledek je též možné zobrazit jako číselnou tabulku, což ukazuje obrázek 3.4. Oba obrázky byly pro lepší přehlednost barevně invertovány.

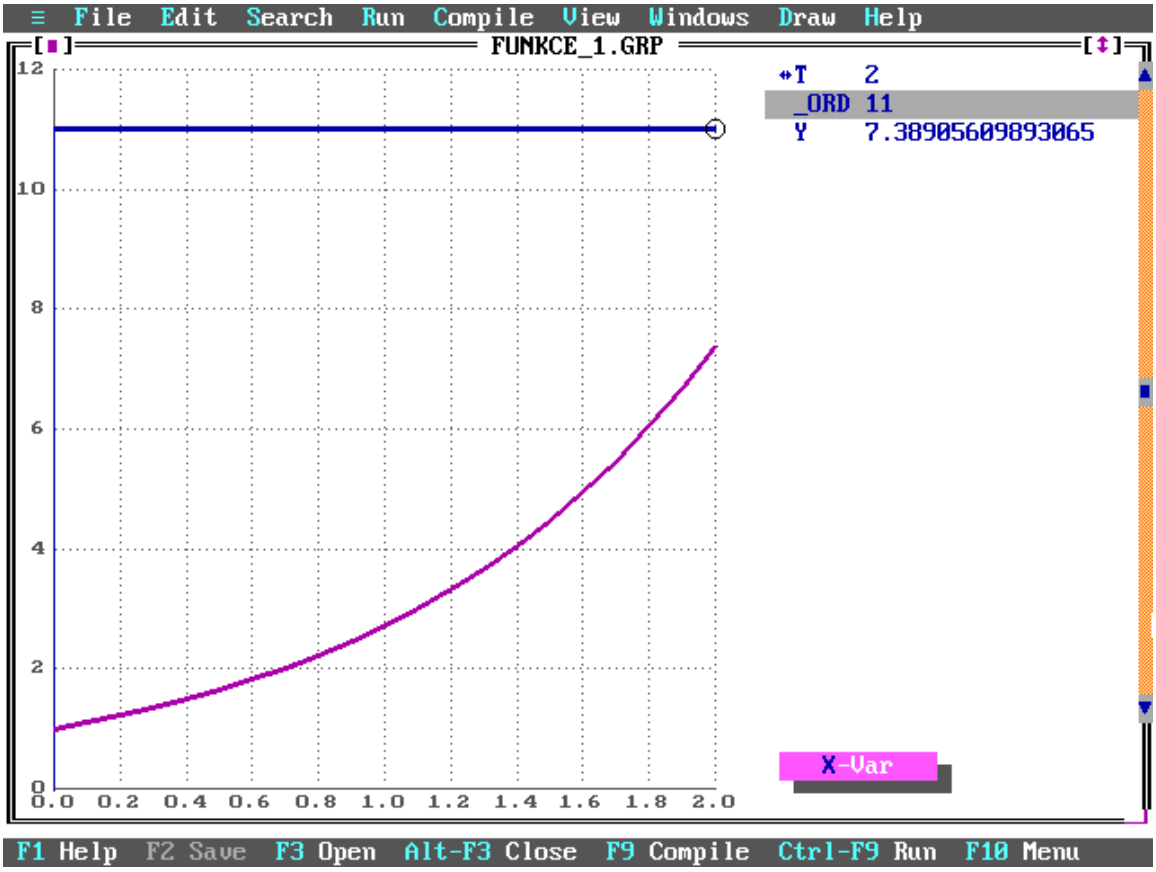
3.3.2 Shrnutí vlastností systému TKSL/386

Následující seznam shrnuje hlavní vlastnosti jazyka systému TKSL/386 vzhledem k popisu diferenciálních rovnic a jejich numerickému řešení.

- Diferenciální rovnice jsou popisovány přímo (není potřeba zvlášť uvádět počáteční podmínky a pravou stranu rovnice).
- Je potřeba deklarovat všechny použité proměnné.
- Parametry simulace se nastavují pomocí speciálně pojmenovaných konstant.
- Výsledky jsou zaznamenány po předem daných krocích.
- Simulace je spuštěna automaticky, není potřeba volat speciální příkazy.
- Výsledek výpočtu je formou grafu či tabulky zobrazen okamžitě.
- Číselné hodnoty nelze ze systému snadno exportovat.

3.4 Návrh jazyka pro systém Integrátor.NET

Návrh vstupního jazyka pro tento systém vychází ve velké míře z vlastností jazyka systému TKSL/386. Některými prvky se pak inspiruje jak u jiných simulačních jazyků (např. MATLAB) tak moderních programovacích jazyků (např. C#).

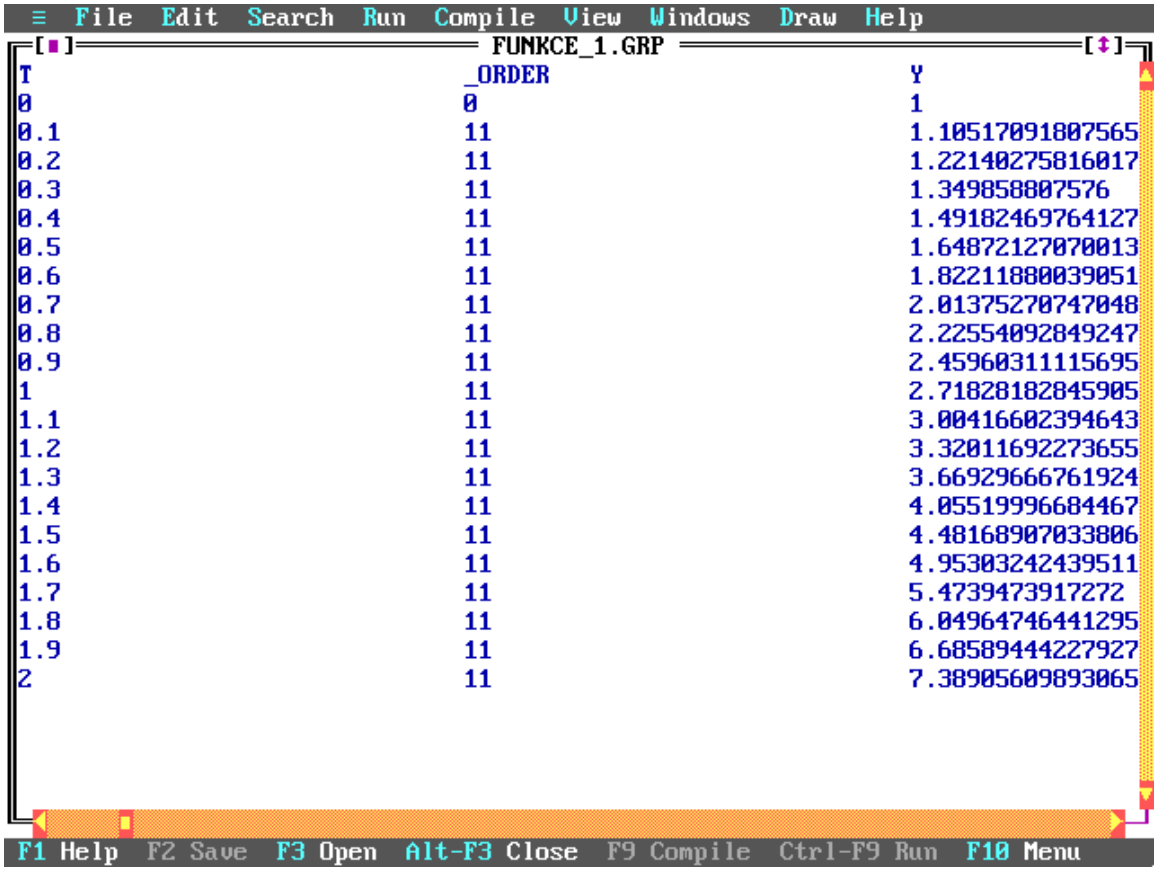


Obrázek 3.3: Řešení v systému TKSL/386

3.4.1 Požadavky na jazyk

Jazyk musí splňovat několik základních požadavků. Prvním z nich je snadný a srozumitelný zápis diferenciálních rovnic. Z tohoto zápisu musí být na první pohled patrné, která proměnná se v rovnici nachází v podobě 1. derivace a také jaká počáteční podmínka pro danou rovnici platí. Vhodnou formu zápisu diferenciálních rovnic můžeme převzít ze systému TKSL/386, především kvůli jeho srozumitelnosti. Další vhodnou vlastností je explicitní deklarace všech použitých proměnných, především díky následnému snadnému odhalení překlepů při zápisu jmen proměnných. Nutným požadavkem jsou potom standardní aritmetické funkce a operátory.

Jedním z hlavních požadavků bylo umožnit v jazyce volání funkčních celků, především pak funkcí ze systémové matematické knihovny. Tyto funkce pak mohou být použity například pro porovnání přesnosti integračních metod se systémovou implementací matematických funkcí. Tento požadavek jsem se rozhodl rozvinout a umožnit také definici vlastních funkčních bloků, aby bylo možné oddělit zápis některých pomocných výpočtů od hlavního řešeného problému. Tyto pomocné výpočty pak mohou být použity opakovaně, čímž se snižuje obsah kódu nutného pro popsání některých problémů a také možnost zanesení chyby způsobené neustálým přepisováním kódu.



Obrázek 3.4: Řešení v systému TKSL/386 zobrazené jako tabulka

3.4.2 Základní vlastnosti jazyka

Na základě uvedených požadavků a konzultací s vedoucími práce jsem vytvořil jazyk pro vstup simulačního systému Integrátor.NET. Tento jazyk je popsán formou LL(1) gramatiky. Tato gramatika je v podobě přepisovacích pravidel uvedena v příloze A. Více informací o LL gramatikách nalezneme například v [7].

Následující seznam uvádí několik důležitých faktů o jazyce sytému Integrátor.NET:

- Každý příkaz končí středníkem.
- Funkční celky se v jazyce nazývají bloky.
- Pro oddělení bloků se využívají složené závorky.
- Pro identifikátory platí stejná pravidla jako v jazyce C.
- Jazyk systému Integrátor.NET rozlišuje velikost písmen.

3.4.3 Syntaxe a sémantika jazykových konstrukcí

Nyní si popíšeme hlavní konstrukce jazyka a uvedeme i jejich příklady.

Komentáře

Jazyk podporuje jak jednořádkové tak víceřádkové komentáře ve stylu jazyka C.

```
y = a + 1; // toto je komentář
/* Toto je delší komentář,
     klidně víceřádkový. */
```

Konstanty

V jazyce existují dva typy konstant – pojmenované a anonymní.

Anonymní konstantou je libovolné číslo zapsané ve zdrojovém textu. Jako oddělovač desetinné části je používána tečka. Přípustný je jak zápis desetinnou formou (např. `3.14`), tak v exponenciální notaci (např. `1.3e-10`).

Definice pojmenované konstanty je uvozena klíčovým slovem `const` následovaným identifikátorem a přiřazením hodnoty této konstantě. V jedné definici může být uvedeno více konstant oddělených čárkou. Hodnota pojmenované konstanty je definována výrazem, avšak tento musí být vyčíslitelný již v době překladu. Je tedy možné hodnotu definovat anonymní konstantou, ale také odvodit hodnotu jedné pojmenované konstanty od jiné pojmenované konstanty.

```
const a = 1, b = 2;
const c = a+b;
```

Dle místa definice rozlišujeme lokální a globální konstanty. Globální konstanty jsou definovány vně bloku a platí ve všech překládaných souborech. Lokální konstanty jsou definovány uvnitř bloku na který je také omezena jejich platnost.

Proměnné

V systému Integrátor.NET existuje jediný typ proměnné – reálné číslo. Deklarace samotná je uvozena klíčovým slovem `var` následovaným seznamem proměnných oddělených čárkami.

```
var a,b,c;
```

Parametry simulace

Systém nezavádí žádnou speciální syntaxi pro zadání parametrů simulace. Způsob zadávání parametrů je zde stejný jako v systému TKSL/386 – tedy s využitím speciálně pojmenovaných konstant. V systému Integrátor.NET musí být tyto konstanty definovány jako globální a jejich přehled uvádí tabulka 3.1.

Jméno	Význam
`tmax`	Koncový čas simulace (počáteční je vždy 0)
`record`	Velikost kroku pro záznam výsledků
`step`	Velikost kroku numerické integrační metody
`eps`	Přesnost výpočtu[1]

Tabulka 3.1: Přehled konstant ovlivňujících simulaci

[1] Vždy musí být zadán právě 1 parametr z dvojice `step` a `eps`.

Zápis diferenciální rovnice

Diferenciální rovnice tvoří jádro systému Integrátor.NET. Syntaxe vychází z jazyka systému TKSL/386. Jako počáteční podmínka může být uveden výraz, ale tento výraz musí být konstantní (tedy vyčíslitelný již v době překladu).

```
promenna' = vyraz & poc.podminka;
```

Definice bloku

Definice bloku je uvozena klíčovým slovem `block`, následuje identifikátor bloku a jeho parametry. Vzhledem k funkci jazyka musí mít každý blok alespoň jeden parametr.

```
block foo(<<definice parametrů>>)
{
    <<příkazy>>
}
```

Parametry bloku jsou dvojího typu – vstupní a výstupní. Typ parametru je uveden při jeho definici pomocí klíčových slov `in` a `out`. Při definici bloku pak musí být uvedeny vždy nejprve vstupní parametry a poté výstupní.

```
block foo(in vstup, out vystup) { ... }
```

Volání bloku

Volání bloku je provedeno vložením jeho jména a platného počtu argumentů. Výstupní argumenty jsou i při volání explicitně označeny klíčovým slovem `out`.

```
block sum(in a, in b,  out c) { c = a+b; }

system {
    var a = 1, b = 5, c;
    sum(a, b, out c);
}
```

Při překladu se provede vložení kódu bloku do všech míst, kde je blok použit. Rekurze je zakázána.

Cyklus for

Cyklus `for` slouží pro jednoduché vygenerování soustavy podobných rovnic. Jeho funkčnost nejlépe ilustruje následující příklad. Zápis

```
var a1=1,a2=9;
for (x,y;2;a) {
  x'=a-y*9 & 1;
  y'=(x+y)^(a-1) & 0;
}
```

je při překladu nahrazen následující soustavou diferenciálních rovnic:

```
var a1=1,a2=9;
x1'=a1-y1*9 & 1;
y1'=(x1+y1)^(a1-1) & 0;
x2'=a2-y2*9 & 1;
y2'=(x2+y2)^(a2-1) & 0;
```

Přiřazení hodnoty proměnné

Přiřazení je značeno jedním znakem = a na rozdíl od jazyka C není výrazem, ale příkazem.

```
a = 2;
```

3.4.4 Výrazy

Výrazem je předpis, který po vyhodnocení vrací hodnotu.

Aritmetické výrazy

Aritmetický výraz se skládá z operandů spojených pomocí operátorů. Jako operand se mohou vyskytovat proměnné a anonymní či pojmenované konstanty. Přehled operátorů definovaných jazykem je uveden v tabulce 3.2

Operátor	Význam
+	součet
-	rozdíl
*	násobek
/	podíl
^	mocnina

Tabulka 3.2: Aritmetické operátory

Logické výrazy

Logické operace mohou být využity především v podmíněném příkazu (viz dále). Každý logický výraz může nabývat dvou hodnot - pravda a nepravda. Vzhledem k tomu, že jazyk nedefinuje zvláštní datový typ pro logické hodnoty, jsou tyto reprezentovány číselně. Převody mezi těmito typy hodnot jsou uvedeny v tabulce 3.3.

Logická hodnota	platí pro hodnoty x	Je převedeno na
pravda	$x \geq 0$	1
nepravda	$x < 0$	-1

Tabulka 3.3: Interpretace logických hodnot

Jazyk definuje dva typy logických operátorů - porovnávací a spojovací. Porovnávací operátory slouží k porovnání hodnot dvou výrazů a jejich přehled je uveden v tabulce 3.4.

Spojovací operátory umožňují tvorbu složitějších logických výrazů. Podporují dvě formy zápisu. První je převzata z jazyka C, druhá potom z jazyka PASCAL. Funkčně jsou tyto formy ekvivalentní a je tedy pouze na uživateli, který zápis preferuje. Přehled spojovacích operátorů je uveden v tabulce 3.5.

Operátor	Význam
`a == b`	Přesné porovnání
`a != b`	Přesná nerovnost
`a > b`	Větší než
`a >= b`	Větší nebo rovno
`a < b`	Menší než
`a <= b`	Menší nebo rovno

Tabulka 3.4: Logické porovnávací operátory

Operátor	Význam
`&&` / `and`	Logický součin
`||` / `or`	Logický součet
`!` / `not`	Logická negace

Tabulka 3.5: Logické spojovací operátory

Podmíněný výraz

Na základě podmínky vrátí jeden ze dvou výrazů. Syntaxe vychází z ternárního operátoru jazyka C. Použití tohoto výrazu je značně omezeno. Podmínka musí obsahovat proměnnou `t`, tedy simulační čas, a oba vstupní výrazy musí být konstantní. Podmíněný výraz tedy slouží například ke generování různých skokových změn na vstupu systému.

Následující příklad vrací `x1` pokud platí `cond`.

```
cond ? x1 : x2
```

3.4.5 Záznam výsledků

V komplikovaných výpočtech může nastat situace, kdy budeme mít velké množství rovnic, ale zajímat nás budou výsledky pouze pár z nich. Bylo by proto výhodné zavést nutnost explicitně označit, které hodnoty si uživatel přeje ve výsledcích zobrazit.

K tomuto účelu vznikla v jazyce konstrukce `watch`. Tato konstrukce podporuje dvě formy zápisu. První z nich pouze označí proměnnou, kterou si uživatel přeje zobrazit.

```
var x;
x' = x & 1;
watch x;
```

Druhá forma zápisu je velmi podobná deklaraci konstanty. V této formě totiž uživatel specifikuje výraz, který bude určovat hodnotu výsledné `watch`.

```
var x,y;
y' = x & 0;
x' = -y & 1;
watch w = x + y;
```

Při použití tohoto zápisu má výsledná `watch` jméno, které bylo použito při deklaraci.

Kapitola 4

Návrh a implementace systému

V této kapitole se budeme věnovat návrhu systému jako celku. Poté rozebereme návrh jednotlivých součástí systému a nakonec se budeme věnovat implementaci systému Integrátor.NET.

4.1 Celkový návrh

V této podkapitole se zaměříme na celkový návrh a členění systému. Nejprve probereme volbu cílové platformy, následně obecné principy návrhu systému. Poslední část se pak věnuje návrhu všech hlavních částí systému Integrátor.NET.

4.1.1 Zvolená platforma a programovací jazyk

Pro vývoj systému byla zvolena platforma Microsoft .NET a jazyk C#. Toto prostředí bylo vybráno především kvůli možnostem kompilace kódu za běhu programu, jejichž využití bude ukázáno později v této kapitole. Jazyk C# patří mezi moderní čistě objektově orientované jazyky a platforma .NET nabízí velmi rozsáhlou systémovou knihovnu. Díky těmto vlastnostem je možné se plně soustředit na řešení daného problému. Zajímavá je též možnost prozkoumat možnosti této platformy pro implementaci náročných numerických výsledků.

Více informací o jazyce C# a platformě Microsoft .NET lze získat v oficiální dokumentaci umístěné na [9].

Prostředí .NET je nativně podporováno pouze v systémech Microsoft Windows. Existuje však projekt s volným zdrojovým kódem MONO, který poskytuje prostředí pro běh .NET aplikací na mnoha dalších platformách (např. Linux, MacOS, Solaris, ...).

4.1.2 Obecné principy dodržované při vývoji

Během návrhu systému Integrátor.NET jsem se snažil dodržovat různé zásady a principy sloužící pro lepší čitelnost a udržovatelnost výsledného kódu. Zde bych chtěl zmínit dva principy, které jsou používány napříč celým systémem. Jedná se o *princip jedné odpovědnosti* a *princip inverze závislosti*.

Princip jedné odpovědnosti (Single Responsibility Principle - SRP) říká, že jedna třída by měla mít pouze jedinou zodpovědnost a tuto zodpovědnost také celou zapouzdřovat. Uplatnění tohoto principu především zamezí vzniku obrovských a špatně udržovatelných tříd, které se starají o velké množství činností zároveň.

Princip inverze závislosti (Dependency Inversion Principle - DIP) říká, že třída by měla záviset pouze na veřejném rozhraní a ne na konkrétní implementaci. Díky dodržování tohoto principu jsou výsledné třídy tzv. volně svázány. To znamená, že je možné nahradit jednu konkrétní implementaci daného rozhraní jinou, aniž by bylo potřeba jakkoliv upravovat třídu, která toto rozhraní používá. Tohoto efektu lze velmi výhodně využít například při testování, kdy reálnou implementaci rozhraní nahradíme testovací verzí, u které jsme schopni přesně určit chování. Následně ověřujeme reakce testované třídy na toto nám přesně známe chování.

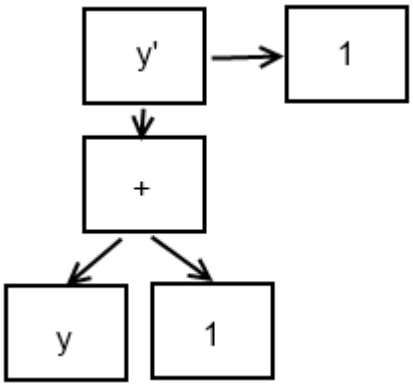
O těchto principech a především o jejich aplikaci v praxi pojednává [5].

4.1.3 Základní členění systému

Aplikace je rozčleněna do několika modulů. Každý modul má na starosti jinou část procesu simulace. První úsek představuje převedení zdrojového textu do strojově snadno zpracovatelné podoby – vnitřní reprezentace kódu. Nad tímto vnitřním kódem následně proběhnou nutné transformace a optimalizace (např. nahrazení volání bloků jejich obsahem, výpočet konstantních výrazů, atd.). Tento vnitřní kód bude přeložen do spustitelné podoby tak, aby bylo možné provést simulaci. V průběhu samotné simulace si aplikace zaznamená výsledky, které budou po ukončení simulace zobrazeny uživateli.

4.1.4 Vnitřní reprezentace kódu

Dříve než se pustíme do návrhu jednotlivých částí aplikace, je potřeba navrhnout vhodné datové struktury pro reprezentaci vstupního programu v paměti. Jako nejvhodnější mi přišla forma výrazového stromu. Kořenem tohoto stromu je v tomto případě příkaz jazyka. Jeho jednotlivé větve pak odpovídají parametrům tohoto příkazu. Například výraz `y' = y + 1 & 1` je převeden na výrazový strom uvedený na obrázku 4.1.



Obrázek 4.1: Struktura vnitřního kódu

4.1.5 Převedení textového vstupu do vnitřního kódu

Vstupní textový soubor musí být analyzován a převeden do vnitřní reprezentace, tedy je třeba provést lexikální a syntaktickou analýzu vstupního souboru tak, jak je popsáno v [7]. Lexikální analyzátor bude vytvořen jako konečný automat, který bude po znacích číst vstupní soubor. Syntaktická analýza vstupního programu pak bude probíhat metodou rekurentního sestupu (tedy metodou shora dolů). Toto ovšem nemůže platit pro výrazy, jelikož

nejsou popsány LL(1) gramatikou. Z tohoto důvodu jsou výrazy překládány pomocí precedenční analýzy (tedy metodou zdola nahoru). Pokud tedy hlavní syntaktický analyzátor očekává v daném místě výraz, zavolá na jeho zpracování precedenční analyzátor. Výstupem syntaktické analýzy je vnitřní kód, který přesně odpovídá zápisu ve zdrojovém textu.

4.1.6 Sémantická kontrola a optimalizace

Dalším prvkem překladu je sémantická kontrola. Tato kontrola bude provedena v několika krocích a jejím účelem je zajistit platnost výrazového stromu z hlediska simulace. Ihned po sémantické kontrole bude následovat posloupnost optimalizací. Tyto lze v zásadě rozdělit do dvou skupin. První skupina pokrývá transformaci kódu (např. vkládání obsahu bloků do místa jejich volání a generování rovnic z `for` cyklu). Druhou skupinu tvoří optimalizace výkonnostní, například výpočet konstantních výrazů. Kód po optimalizacích projde znovu sémantickou analýzou z důvodu odstranění chyb, které teoreticky mohou při optimalizacích vzniknout.

4.1.7 Generování spustitelného kódu

Poslední operací samotného překladu je vygenerování spustitelného kódu. Zároveň je však potřeba omezit množství generovaného kódu na nezbytné minimum, aby se zamezilo vzniku těžko odhalitelných chyb. Tento model lze splnit například s využitím bázové třídy, která bude poskytovat veškerou infrastrukturu nutnou pro simulaci. Následně při překladu bude vytvořena odvozená třída, která bude obsahovat pouze implementaci potřebných metod (vyhodnocení pravé strany rovnice, hodnota počátečních podmínek). Tato třída pak bude následně za pomoci standardních nástrojů platformy .NET zkompilována. Tento postup tak využívá aritmetických optimalizací překladače jazyka C#.

4.1.8 Simulace

Takto zkompilovaný kód je možno načíst do aplikace a následně spustit jeho provádění. Simulace bude předávat data svému nadřazenému objektu pomocí událostí. Z technických a bezpečnostních důvodů pak musí být celá simulace spuštěna ve vlastní aplikační doméně. Aplikační doména je jednotka izolace v rámci aplikací na platformě .NET. Více informací o tomto tématu se nachází v dokumentaci na [9]. Simulační algoritmus bude vycházet z algoritmu pro spojitou simulaci uvedeném v [10].

4.1.9 Zobrazení výsledků

Po úspěšném dokončení simulace budou zobrazeny výsledky. Zobrazení výsledků bude primárně ve formě grafu, ale musí být zachována možnost uložit výsledky jako číselná data. Toho lze dosáhnout například uložením dat ve formátu CSV. Tento formát je vhodný především pro svou jednoduchost a také proto, že s ním umí pracovat tabulkové procesory jako Microsoft Excel nebo LibreOffice Calc.

4.2 Implementace systému Integrátor.NET

V této podkapitole se seznámíme s implementací jednotlivých součástí systému.

4.2.1 Grafické uživatelské rozhraní

Pro tvorbu uživatelského rozhraní jsem se rozhodl využít technologii Windows Presentation Foundation (WPF), jelikož se jedná o nejmodernější způsob tvorby uživatelského rozhraní na platformě .NET. Aplikace vytvořené touto technologií jsou však spustitelné pouze na platformě Microsoft .NET, která je dostupná jen pro operační systém Microsoft Windows. Aby byla zachována přenositelnost klíčových komponent systému, rozhodl jsem se jej rozdělit do dvou částí. První část tvoří knihovna nezávislá na platformě, obsahující veškerou funkcionalitu nutnou pro překlad a simulaci. Druhou část tvoří grafické uživatelské rozhraní (GUI) vytvořené pomocí WPF. Toto rozhraní je však pouze jakousi tenkou obálkou nad třídami v knihovně.

Díky tomuto rozdělení je možné maximálně využít prostředků pro tvorbu uživatelského rozhraní na konkrétní platformě. Při tom není nutné pro každou platformu znovu vytvářet klíčové komponenty systému, ve kterých je implementován překlad a simulace.

Zároveň s knihovnou vzniklo i konzolové rozhraní, které je nezávislé na platformě. To umožňuje alespoň v omezené míře tento systém používat i v jiných operačních systémech, než je Microsoft Windows.

4.2.2 Moduly systému

Jak bylo uvedeno v části 4.1.3, je systém Integrátor.NET složen z volně provázaných modulů. Hlavní funkcionalita systému se nachází v modulu `Core`. Tento modul obsahuje všechny třídy, které se používají při práci s vnitřním kódem. Dále obsahuje zastřešující třídy, které zapouzdřují celé procesy kompilace a simulace. V neposlední řadě pak obsahuje všechna rozhraní, pomocí kterých systém komunikuje s ostatními moduly. Systém se skládá ze dvou dalších skupin modulů. Tyto moduly poskytují implementace rozhraní `IFrontEnd` a `IBackEnd`. Proto jsou označeny jako front-end a back-end moduly.

Modul front-end se stará o převod zdrojového kódu do vnitřní reprezentace. V současné době existují dvě implementace tohoto modulu. První z nich je `FrontEnd.Text`. Ten slouží ke kompilaci z textového zdrojového kódu, který je popsán v kapitole 3. Druhá implementace tohoto modulu – `FrontEnd.Block` – slouží pro kompilaci z blokových schémat. Popis této implementace naleznete v [3].

Modul back-end slouží pro převod vnitřního kódu do spustitelné podoby a také se stará o spouštění zkompilované simulace. V současné době existuje jediná implementace tohoto modulu – `BackEnd.CIL`[1], která převádí vnitřní kód do podoby spustitelné na platformě .NET a více se jí budu věnovat později v této kapitole.

4.2.3 Vnitřní reprezentace kódu

Vstupní kód je při překladu převeden do tzv. vnitřní reprezentace kódu. Příkazy jsou převedeny do podoby výrazových stromů. Dále jsou ve vnitřním kódu uloženy deklarace a definice konstant, proměnných a `watch`. Všechny zmíněné informace jsou zapouzdřeny ve třídě `CompilationUnit`. Tato třída je vytvořena vždy jednou pro celou kompilaci a nese v sobě vnitřní kód odpovídající všem kompilovaným souborům. Nad touto třídou jsou také prováděny veškeré optimalizace.

[1] Zkratka CIL znamená *Common Intermediate Language*. Jedná se o jakousi obdobu jazyka assembler v prostředí .NET.

4.2.4 Vnitřní reprezentace výrazů

Výrazy jsou reprezentovány stromem objektů. Každý uzel tohoto stromu je objektem třídy, která implementuje rozhraní `IExpression`. Toto rozhraní poskytuje přístup k jednotlivým podvýrazům daného uzlu, k typu výrazu v uzlu a také k jeho hodnotě. Tuto hodnotu pak využívají výrazy jako konstanta nebo čtení hodnoty proměnné.

Díky implementaci pomocí rozhraní je možné využít dva na první pohled protichůdné přístupy. První z nich zahrnuje vytvoření obecných nástrojů pro práci s výrazovým stromem. Tento přístup je nezbytný pro implementaci nástrojů, které mají s vnitřním kódem dále pracovat. Pokud by však byl strom implementován pouze jako obecný objekt, velmi by to ztížilo práci s jednotlivými uzly při jejich vytváření. Kompromisu lze dosáhnout právě pomocí rozhraní `IExpression`. Konkrétní uzly jsou implementovány konkrétními třídami (např. `BinaryOperation`, `TernaryOperation`, `Constant`, `DifferentialEquation`, atd.) ale díky jednotnému rozhraní mohou s výrazovým stromem všechny další vrstvy systému pracovat jednotně.

4.2.5 Lexikální analyzátor

Lexikální analyzátor je implementován třídou `BasicScanner`. Lexikální analýza je v ní implementována formou konečného automatu. Tato třída přijímá jako vstup zdrojový text. Jejím výstupem je posloupnost lexikálních symbolů. Lexikální symbol (třída `Token`) obsahuje veškeré informace o aktuálním kusu zpracovávaného kódu. Jeho hlavními atributy jsou typ symbolu (např. identifikátor, pravá závorka) a jeho hodnota, která může být prázdná. Dále nese informace o pozici symbolu v rámci zdrojového kódu, což je užitečné v případě výskytu chyby. Ostatní třídy komunikují s lexikálním analyzátorem přes rozhraní `IScanner`.

4.2.6 Syntaktický analyzátor

Hlavní syntaktický analyzátor je implementován třídou `Parser`. Tato třída potřebuje pro svou funkci třídu implementující rozhraní `IScanner` . Syntaktická analýza je prováděna formou rekurentního sestupu tak, jak je popsána v [7]. V případě, že je očekáván výraz, je řízení syntaktické analýzy předáno třídě `ExpressionParser`, která provádí precedenční syntaktickou analýzu.

Vstup syntaktického analyzátoru

Třída `Parser` musí být před použitím inicializována. Přitom jsou jí předány veškeré závislosti, které potřebuje pro svou správnou funkčnost. Tyto závislosti zapouzdřuje rozhraní `IParserContext`. Třída implementující toto rozhraní nese jméno vstupního souboru (nutné pro chybová hlášení), objekt třídy `CompilationUnit` a inicializovaný objekt třídy implementující rozhraní `IScanner`.

4.2.7 Sémantická kontrola

Sémantickou kontrolu implementuje třída `SemanticChecker`. Stará se především o volání dalších kontrol. Následující seznam uvádí třídy, které provádí tyto kontroly, a také význam jednotlivých kontrol.

`FlagsSetter` Kontroluje deklarace proměnných a jejich použití. Dále kontroluje volání bloků (počet a typ parametrů).

`FlagBasedChecker` Generuje varování podle příznaků nastavených třídou `FlagSetter`.

`WalkerChecker` Představuje druhý průchod stromem. Při prvním průchodu nebylo např. u soustav diferenciálních rovnic možné odhalit proměnné s nedefinovanou hodnotou.

`WatchesChecker` Kontroluje kolize názvů `watch`, které byly definovány výrazem, s názvy proměnných.

Více informací o sémantických kontrolách v systému Integrátor.NET lze nalézt v [3].

4.2.8 Optimalizace a transformace kódu

Před vytvořením simulační třídy je třeba provést transformace a optimalizace kódu tak, jak byly zmíněny v podkapitole 4.1.6. Vstupní data pro tyto transformace představuje instance třídy `CompilationUnit`, která obsahuje data ze syntaktické analýzy. Následující seznam pak uvádí posloupnost jednotlivých transformačních operací.

1. kontrola integrity
2. nastavení příznaků (kromě `for` cyklů)
3. zpracování volání tzv. trivial bloků[2]
4. generování soustav rovnic z `for` cyklů
5. odstranění sekvencí vzniklých vygenerováním soustav rovnic z `for` cyklů
6. první sémantická kontrola – viz 4.2.7
7. výpočet hodnot konstant (nejprve globálních, potom lokálních)
8. vložení kódu bloků do místa jejich volání
9. závěrečné úpravy v bloku `system`
10. závěrečná sémantická kontrola

Více informací o jednotlivých krocích transformace a jejich implementaci lze nalézt v [3].

4.2.9 Generování spustitelného kódu

Poslední součástí překladu je generování simulace v podobě strojově proveditelného kódu. Systém Integrátor.NET používá velmi jednoduchý princip spojité simulace, tak jak je popsán v [10]. Tato funkčnost je implementována v abstraktní třídě `SimulationBase`. Při kompilaci je pak vygenerována konkrétní třída, která dědí od `SimulationBase`. Tato třída, zapsaná v jazyce C#, je poté zkompilována pomocí standardních nástrojů poskytovaných knihovnou .NET do podoby knihovny.

[2] Jedná se o zápis základních operátorů (+, -, *, ...) pomocí speciálně pojmenovaných bloků.

Vygenerování simulační třídy

Výpočet pravých stran rovnic je v simulační třídě prováděn metodou `DerivationFunction`. Tělo této metody tvoří přeložené výrazové stromy. Stromy výrazů se překládají pomocí standardních operátorů jazyka C# a funkcí z matematické knihovny .NET. Generování výrazů jazyka C# z vnitřní reprezentace kódu provádí třída `CSharpExpressionTranslator`. Překlad volání vestavěných bloků na volání funkcí z matematické knihovny pak provádí třída `BuiltInBlockTranslator`.

Simulační třída je vytvořena vložením menších vygenerovaných kusů kódu do textové šablony. Výsledkem této operace je pak třída zapsaná v jazyce C#. Celé generování této třídy zapouzdřuje třída `CSharpCodeGenerator`.

Kompilace simulační třídy

Textový zápis simulační třídy je zkompilován pomocí standardního překladače jazyka C#. Velkou výhodou platformy .NET je, že tento překladač je v ní přímo obsažen, a není tedy potřeba žádný program třetí strany. Zkompilovaný kód je uložen ve formě DLL knihovny, kterou je možné za běhu načíst do systému Integrátor.NET a spustit takto simulaci. V systému Integrátor.NET je tato funkcionalita zapouzdřena ve třídě `CSharpCompiler`, která nastaví správné parametry překladače jazyka C# a následně spustí samotný překlad.

4.2.10 Simulace

Simulaci v systému Integrátor.NET má na starosti několik tříd. Toto je způsobeno rozčleněním simulace do několika vrstev. V tomto modelu předávají vyšší vrstvy těm nižším příkazy a obdrží od nich výsledky simulace. Vícevrstevný model je nutný z důvodu jistých omezení platformy .NET. V prostředí .NET totiž není možné uvolnit konkrétní knihovnu z paměti. Uvolněny mohou být pouze celé aplikační domény. Z tohoto důvodu je simulační třída načtena do speciální aplikační domény. Komunikace mezi aplikačními doménami však probíhá formou tzv. remotingu[3], kdy musí být veškerá data přecházející mezi aplikačními doménami serializována. Aby z těchto důvodů nedocházelo k poklesu výkonu výpočtu, běží tento ve vlastním vlákně.

Nyní si popíšeme jednotlivé vrstvy simulace. Uvedeme si také, ve kterém modulu se tato vrstva nachází.

Simulační třída - `SimulationBase`

Všechny simulační třídy dědí od třídy `SimulationBase`. Tato třída se nachází v modulu `BackEnd.CIL.Runtime`. Jak již bylo řečeno implementuje základní funkcionalitu nutnou pro spuštění simulace. Výsledky simulace předává pomocí událostí vyšším vrstvám.

Spouštěč simulace - `SimulationRunner`

Třída `SimulationRunner` se nachází v modulu `BackEnd.CIL.Runtime`. Slouží pro spuštění simulace ve speciální aplikační doméně a následnou komunikaci s ní. Komunikace se simulační třídou je spuštěna ve vlastním vlákně. Dochází zde však ke klasickému problému producent–spotřebitel. Tento problém jsem vyřešil použitím třídy `BlockingCollection`,

[3] Způsob komunikace mezi uzly na platformě .NET. Lze jej využít i pro komunikaci po síti atd. Více informací o této problematice naleznete na [9].

kterou poskytuje knihovna .NET. Pokud nějaká metoda žádá data z této kolekce a kolekce je prázdná, je volající vlákno zablokováno. Ve chvíli, kdy jsou do kolekce vložena nová data, je volající vlákno opět probuzeno.

Poskytovatel simulace - `ISImulationProvider`

Rozhraní `ISimulationProvider` umožňuje genericky přistupovat k simulaci nezávisle na modulu, který simulaci poskytuje. Jedinou implementaci tohoto rozhraní však v současné době představuje třída `CILSimulationProvider` z modulu `BackEnd.CIL`. Tato třída se stará o vytvoření aplikační domény pro simulaci a vytvoření spouštěče v ní. Výsledky simulace pak prostřednictvím rozhraní poskytuje vyšší vrstvě.

Hlavní simulátor - `Simulator`

Třída `Simulator` z modulu `Core` má na starosti především zaznamenání výsledků. Výsledky získává od objektu implementujícího rozhraní `ISimulationProvider`. Vzhledem k tomu, že nezná implementaci tohoto rozhraní je i zde simulace spuštěna ve vlastním vlákně. Řešení problému producent–spotřebitel je zde stejné jako v případě třídy `SimulationRunner`.

Třída `Simulator` také při zaznamenávání výsledků spouští výpočet hodnot pro `watch`.

Podporované integrační metody

Systém Integrátor.NET podporuje pouze explicitní jednokrokové integrační metody. Konkrétně Eulerovu metodu, Heunovu metodu a Runge-Kuttovy metody 3. a 4. řádu.

4.2.11 Výsledky simulace

Výsledky simulace jsou ukládány do třídy implementující rozhraní `IResults`. V současné implementaci systému je toto rozhraní poskytováno třídou `ListOfArraysResults`. Tato třída ukládá výsledky do pole čísel pro každou zaznamenanou proměnnou.

Tato data je v systému Integrátor.NET možné zobrazit formou grafu. Tento graf je možné uložit jako obrázek ve formátech PNG, BMP, JPEG, TIFF, GIF a SVG. Pro potřeby dalšího zpracování je pak možné výsledky exportovat ve formátu CSV.

Více informací o uložení a zobrazení výsledků lze nalézt v [3].

Kapitola 5

Příklady

Jak již bylo zmíněno dříve nelze pro některé složitější soustavy diferenciálních rovnic řešení vůbec nalézt. Proto byly vybrány vhodné demonstrační příklady z teorie obvodů u kterých je nám znám průběh řešení.

5.1 Srovnání přesnosti s matematickou knihovnou .NET

V této části provedeme srovnání přesnosti výpočtu implementovaného v systému Integrátor.NET s funkcí z matematické knihovny platformy .NET. Toto srovnání provedeme pro všechny implementované integrační metody.

Srovnávat budeme exponenciální funkci $f(t) = \mathrm{e}^x$ pro $t \in \langle 0; 2 \rangle$. Této funkci odpovídá diferenciální rovnice

$$y' = y, \quad y(0) = 1 \tag{5.1}$$

Tuto diferenciální rovnici přepíšeme do programu systému Integrátor.NET. Zvolíme simulační krok $h = 0{,}01$.

```
const tmax=2, step=0.01, record=0.1;
system {
    var y,x;

    y' = y & 1; // výpočet diferenciální rovnicí

    exp(t, out x); // výpočet pomocí matematické knihovny

    watch d = y - x; // zajímá nás rozdíl v~obou řešeních
}
```

Přeložíme tento kód a postupně spustíme simulaci se všemi implementovanými integračními metodami. Hodnoty `d` ve vybraných bodech jsou uvedeny v tabulce 5.1. Pro přehlednost byly tyto výsledky zaokrouhleny na 6 platných číslic.

5.2 Řešení příkladů z teorie obvodů

Nyní pomocí simulačního systému Integrátor.NET vyřešíme vybrané příklady z teorie obvodů. Příklady budeme řešit pomocí metody Runge-Kutta 4. řádu.

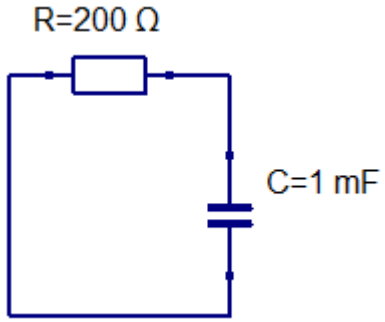
t	Eulerova metoda	Heunova metoda	Runge-Kutta 3. řádu	Runge-Kutta 4. řádu
$0,5$	$-4,08945 \cdot 10^{-3}$	$-1,36367 \cdot 10^{-5}$	$-3,40747 \cdot 10^{-8}$	$-6,81264 \cdot 10^{-11}$
$1,0$	$-1,34680 \cdot 10^{-2}$	$-4,49659 \cdot 10^{-5}$	$-1,12360 \cdot 10^{-7}$	$-2,24643 \cdot 10^{-10}$
$1,5$	$-3,32662 \cdot 10^{-2}$	$-1,11204 \cdot 10^{-4}$	$-2,77874 \cdot 10^{-7}$	$-5,55563 \cdot 10^{-10}$
$2,0$	$-7,30382 \cdot 10^{-2}$	$-2,44458 \cdot 10^{-4}$	$-6,10849 \cdot 10^{-7}$	$-1,22130 \cdot 10^{-9}$

Tabulka 5.1: Globální chyba aproximace jednotlivých metod

5.2.1 Sériový obvod RC - vybíjení kondenzátoru

Zadání

Řešte sériový RC obvod uvedený na obrázku 5.1 připojený ke stejnosměrnému zdroji napětí. Na větvi tvořené sériově zapojeným odporem $R = 200\ \Omega$ a kondenzátorem $C = 10^{-3}$ F v okamžiku $t = 0$ zkratujeme zdroj napětí. Napětí na kondenzátoru v čase $t = 0$ je $u_C(0) = 10$ V.



Obrázek 5.1: Obvod RC

Řešení v systému Integrátor.NET

Pro numerické řešení musíme vyjádřit průběh hodnoty napětí na kondenzátoru pomocí diferenciální rovnice.Obecnému tvaru této rovnice pak odpovídá rovnice (5.2).

$$u'_C = -\frac{1}{RC}u_C, \quad u_C(0) = U \tag{5.2}$$

R, C a U jsou konstanty jejichž hodnoty známe ze zadání.

Tuto rovnici zapíšeme v jazyce systému Integrátor.NET.

```
const tmax=10,step=10e-3,record=0.1;
const C=1e-5, R=200;

system {
    var uc;
    uc' = (-1/(R*C))*uc & 10;
    watch uc;
}
```

Řešení v systému MATLAB

Pro porovnání výsledků spočtených aplikací Integrátor.NET spočítáme stejný příklad i v systému MATLAB a porovnáme výsledky.

```
fn = @(t, uc) (-1/(200*10e-3))*uc;
[T, Y] = ode45(fn, [0 10], [10]);
```

Výsledky z obou systémů byly v několika vybraných bodech porovnány. Toto srovnání naleznete v tabulce 5.2. Orientačně také byla změřena délka výpočtu v jednotlivých systémech. Toto měření je však pouze orientační. Výsledky tohoto orientačního měření jsou v tabulce 5.3.

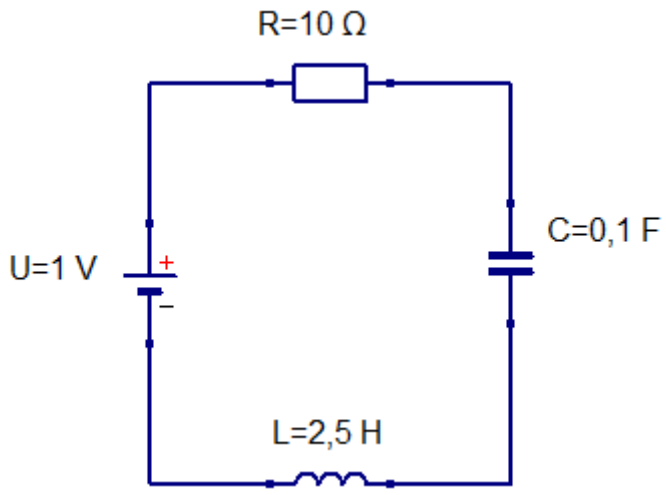
t	MATLAB	Integrátor.NET	Rozdíl
0,9	6,37	6,3762496349	$6{,}25 \cdot 10^{-3}$
2,4	3,0091	3,011942119121980	$2{,}84 \cdot 10^{-3}$
4,9	0,86212	0,862931550325168	$8{,}12 \cdot 10^{-4}$
7,4	0,24702	0,192546055021188	$5{,}45 \cdot 10^{-2}$
9,4	0,090874	0,090952316254243	$7{,}83 \cdot 10^{-5}$

Tabulka 5.2: Řešení příkladu vybíjení kondenzátoru v RC obvodu

Systém	Délka výpočtu
MATLAB	0.0066s
Integrátor.NET	0,2s

Tabulka 5.3: Délka výpočtu

5.2.2 Sériový obvod RLC - nabíjení kondenzátoru



Obrázek 5.2: Obvod RLC

Zadání

Řešte sériový obvod RLC na větvi tvořené sériově spojeným odporem $R = 10\,\Omega$, cívkou $L = 2{,}5\text{H}$ a kondenzátorem $C = 0,1\,\text{F}$. V okamžiku $t = 0\,\text{s}$ připojíme zdroj stejnosměrného napětí. Napětí na kondenzátoru v okamžiku $t = 0\,\text{s}$ je $u_C(0) = 0\,\text{V}$, $i_L(0) = 0\,\text{A}$ a $U = 1\,\text{V}$. Obvod je znázorněn na obrázku 5.2.

Řešení v systému Integrátor.NET

Proud protékající cívkou a napětí na kondenzátoru popisují následující diferenciální rovnice.

$$\begin{aligned} u_L &= U - R \cdot i_L - u_C \\ i'_L &= \tfrac{1}{L} \cdot u_L, \quad i_L(0) = 0 \\ u'_C &= -\tfrac{1}{C} \cdot i_L, \quad u_C(0) = 0 \end{aligned}$$

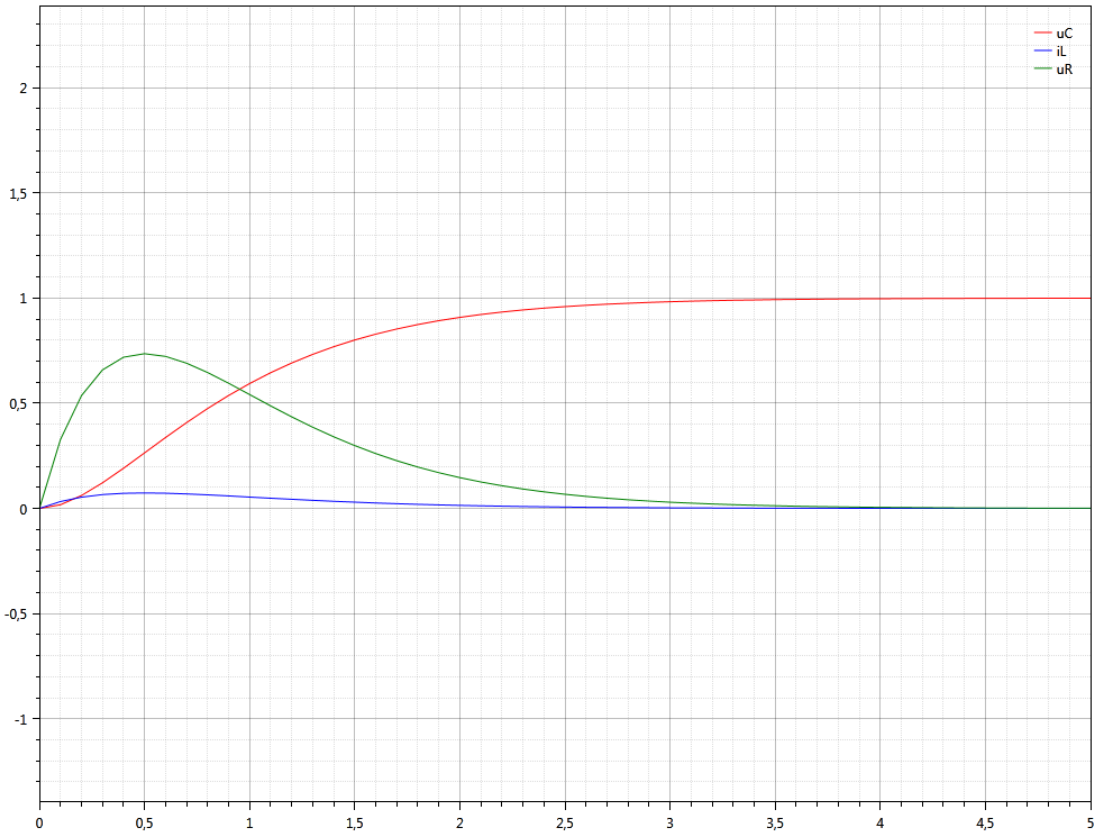
Tyto rovnice převedeme do programového zápisu systému Integrátor.NET a vhodně zvolíme krok integrační metody.

```
const U=1, R=10,L=2.5,C=0.1, record=0.1,tmax=5, step=1e-5;

system {
    var uL, uC, iL;

    uL =U-R*iL-uC;
    uC'=1/C*iL &0;
    iL' =1/L*uL &0;
    watch uC,iL;
    watch uR=10*iL;
}
```

Po přeložení a spuštění simulace získáme stejný graf jako je zobrazen na obrázku 5.3.



Obrázek 5.3: Řešení zadaného RLC obvodu

Kapitola 6

Závěr

Cílem této práce bylo navrhnout a vytvořit systém pro výpočty soustav obyčejných diferenciálních rovnic 1. řádu. Současně s tímto systémem vznikl i nový jazyk pro popis diferenciálních rovnic. Použitelnost tohoto systému pro řešení reálných problémů byla ukázána v kapitole 5. I ve srovnání s pokročilým simulačním systémem MATLAB dopadl systém Integrátor.NET obstojně.

Hlavní přínos simulačního systému Integrátor.NET vidím v překladu kódu do nativní podoby za běhu aplikace. I když současná implementace není dokonalá, jedná se o správný směr vývoje.

V budoucím vývoji tohoto systému by bylo dobré se zaměřit na zdokonalení simulace jako takové. Největší slabinu současného systému vidím v tom, že simulace používá fixní krok. Dalším možným rozšířením je pak podpora pro vícekrokové a implicitní integrační metody.

Literatura

[1] Fajmon, B. a Růžičková, I. *Matematika 3.* Brno: FEKT VUT v Brně, 2005.

[2] Kubíček, M., Dubcová, M. a Janovská, D. *Numerické metody a algoritmy.* 2. oprav. vyd. Praha: Vydavatelství VŠCHT Praha, 2005. 188 s. ISBN 80-708-0558-7.

[3] Kučera, M. *Grafický editor pro blokový vstup numerického integrátoru v .NET.* Brno: FIT VUT v Brně, 2012. Bakalářská práce.

[4] MapleSoft. *Nápověda k systémům Maple a MapleSim* [online]. 2012 [cit. 14. května 2012]. Dostupné na: <http://www.maplesoft.com/support/help/>.

[5] Martin, R. C. *Čistý kód.* 1. vyd. Brno: Computer Press, 2009. 423 s. ISBN 978-80-251-2285-3.

[6] MathWorks, Inc. *MATLAB: The Language of Technical Computing* [online]. 2012 [cit. 11. května 2012]. Dostupné na: <http://www.mathworks.com/products/matlab/>.

[7] Meduna, A. a Lukáš, R. *Formální jazyky a překladače – Studijní opora.* 2010.

[8] Melkes, F. a Řezáč, M. *Matematika 2.* Brno: FEKT VUT v Brně, 2002.

[9] Microsoft Corporation. *MSDN Library* [online]. 2012 [cit. 11. května 2012]. Dostupné na: <http://msdn.microsoft.com/en-us/library>.

[10] Peringer, P. *Modelování a simulace – Studijní opora.* listopad 2008.

[11] Vitásek, E. *Numerické metody.* 1. vyd. Praha: Státní nakladatelství technické literatury, 1987. 512 s. Technický průvodce, sv. 67.

Příloha A

Gramatika jazyka systému Integrátor.NET

Přepisovací pravidla pro gramatiku jazyka Integrátor.NET. Tato gramatika obsahuje i několik navrhovaných rozšíření, která však nejsou zatím v systému Integrátor.NET implementována.

Struktura programu

<program> → <program-parts> <sysblock> <program-parts>

<sysblock> → system { <block-body> }
<sysblock> → ε

<program-parts> → ε
<program-parts> → <program-part> <program-parts>

<program-part> → <const-stat>;
<program-part> → <block-definition>

Definice uživatelského bloku

<block-definition> → block <block-name>(<block-params>) { <block-body> }

<block-name> → <id>

<block-params> → <block-params-list>

<block-params-list> → <block-params-type> <id> <block-params-list-cont>
<block-params-list-cont> → ε
<block-params-list-cont> → , <block-params-list>

<block-params-type> → in
<block-params-type> → out

<block-body> → <statments-sequention>

Příkazy

<statments-sequention> → <statment-with-semicolon><statments-sequention>
<statments-sequention> → ε

```
<statment-with-semicolon> → <statment>;
<statment-with-semicolon> → <block>
<statment-with-semicolon> → <for>

<statment> → <equation-or-block-call-or-unop>
<statment> → <var-stat>
<statment> → <const-stat>
<statement> → <watch-stat>

<for> → for ( <for-variables> ; <const> ; <for-variables> )
                <statment-with-semicolon>
<for-variables> → ε
<for-variables> → <for-variables-one>
<for-variables-one> → <id> <for-variables-list>
<for-variables-list> → , <for-variables-one> <for-variables-list>
<for-variables-list> → ε

<foreach> → foreach ( var <id> in <id (type=array)> )
                <statment-with-semicolon>

<block> → { <statments-sequention> }


 Definice proměnných
<var-stat> → var <var-stat-one-var> <var-stat-list>
<var-stat-one-var> → <id> <var-stat-set-val-opt>
<var-stat-set-val-opt> → ε
<var-stat-set-val-opt> → = <expression>
<var-stat-list> → ε
<var-stat-list> → , <var-stat-one-var> <var-stat-list>

 Definice konstant
<const-stat> → const <const-stat-one-const> <const-stat-list>
<const-stat-one-const> → <id> = <expression (type=const)>
<const-stat-list> → ε
<const-stat-list> → , <const-stat-one-const> <const-stat-list>

 Definice watches
<watch-stat> → watch <watch-stat-one-watch> <watch-stat-list>
<watch-stat-one-watch> → <id> <watch-stat-set-val-opt>
<watch-stat-set-val-opt> → ε
<watch-stat-set-val-opt> → = <expression>
<watch-stat-list> → ε
<watch-stat-list> → , <watch-stat-one-watch> <watch-stat-list>

 Začátek přiřazení/integrace, nebo volání bloku
<equation-or-block-call-or-unop> → <id>
                <equation-or-block-call-or-unop-sec-part>
<equation-or-block-call-or-unop-sec-part> → <block-call>
<equation-or-block-call-or-unop-sec-part> → <equation>

 Volání bloku
<block-call> → (<block-call-params>)
<block-call-params> → <block-call-params-list>
```

```
<block-call-params-list> → <block-call-params-list-one-arg>
                           <block-call-params-list-cont>
<block-call-params-list-one-arg> → <expression>
<block-call-params-list-one-arg> → out <id>
<block-call-params-list-cont> → ε
<block-call-params-list-cont> → , <block-call-params-list>

 Přiřazení/integrace
<equation> → = <expression>
<equation> → ' = <expression> & <expression (type=const)>

 Výrazy (nejsou LL)
<expression> → <id>
<expression> → <const>
<expression> → ( <expression> )
<expression> → <operator-un-pre> <expression>
<expression> → <expresion> <operator-bin> <expression>
<expression> → <expression (type=bool)> ? <expression> : <expression>
<expression> → <built-in-function>
<expression> → <block-call>
<expression> → <operator-un-var>
<expression> → <expression> in <expression>..<expression>

<operator-bin> → <operator-bin-ar> | <operator-bin-log>
<operator-bin-ar> → + | - | * | / | ^
<operator-bin-log> → && | || | and | or
<operator-bin-log> → < | > | <= | >= | == | !=
<operator-un-pre> → <operator-un-pre-ar> | <operator-un-pre-log>
<operator-un-pre-ar> → -
<operator-un-pre-log> → ! | not

 Číselná konstanta
<const> → <const-dec-number><const-e-opt>
<const-dec-number> → <const-minus-opt><num-rep><const-dec-part-opt>
<const-dec-part-opt> → ε
<const-dec-part-opt> → .<num-rep>
<const-minus-opt> → ε
<const-minus-opt> → -
<const-e-opt> → ε
<const-e-opt> → (e|E)<const-minus-opt><num-rep>

<num-rep> → <num><num-rep-cont>
<num-rep-cont> → ε
<num-rep-cont> → <num><num-rep-cont>

 Identifikátor
<id> → <char-or-under> <id-cont-opt>
<id-cont-opt> → ε
<id-cont-opt> → <num-or-char-or-under> <id-cont-opt>
<num-or-char-or-under> → <num-or-char> | _
<num-or-char> → <num> | <char>
<char-or-under> → <char> | _
<num> → 0..9
<char> → a..z | A..Z
```

Gramatika pro precedenční analýzu
Bez priority

<E> → i
<E> → c
<E> → (<E>)
<E> → s
<L> → <P> , <L>
<L> → <P>
<P> → out i
<P> → <E>

Prioritní skupiny, čím výš, tím vyšší priorita

<E> → i(<L>)

<E> → − <E>
<E> → ! <E>

<E> → <E> ^ <E>

<E> → <E> * <E>
<E> → <E> / <E>

<E> → <E> + <E>
<E> → <E> − <E>

<E> → <E> < <E>
<E> → <E> > <E>
<E> → <E> <= <E>
<E> → <E> >= <E>

<E> → <E> == <E>
<E> → <E> != <E>

<E> → <E> and <E>
<E> → <E> or <E>

<E> → <E> ? <E> : <E>

Příloha B

Simulační třída v jazyce C#

V této příloze si představíme šablonu pro simulační třídu tak, jak ji používá aplikace. Dále si ukážeme jak vypadá třída vygenerovaná z této šablony pro konkrétní příklad.

B.1 Šablona simulační třídy

V následujícím výpisu je uvedena šablona pro simulační třídu.

```
using Core;
using BackEnd.CIL.Runtime;

namespace Simulation {
    [SimulationRuntime]
    public class <0> : SimulationBase {
        public <0>() : base(<1>,
                simulationName: "<5>",
                newAuxiliaryVariablesCount: <7>,
                newAuxVarsToResultsCount: <8>)
        {
            <4>
        }

        public override double DerivationFunction(
                int integratorNumber,
                double actualTime,
                double integratorOutput)
        {
            <3>
        }

        protected override void ComputeAuxiliaryVariables(
                double actualTime) {
            <6>
        }

        protected override void InitialConditions() {
            <2>
        }
    }
}
```

V tomto výpisu jsou označena místa, na která vkládá systém při kompilaci vygenerovaný kód. Následující seznam uvádí význam jednotlivých pozic:

0. Validní C# identifikátor. Je unikátní pro každou vygenerovanou simulaci.
1. Počet integrátorů v této simulaci.
2. Vygenerovaný kód. Nastaví počáteční podmínky.
3. Vygenerovaný kód. Vypočítá pravou stranu rovnice se zadaným pořadovým číslem.
4. Vygenerovaný kód. Nastaví parametry simulace.
5. Nastaví popis simulace. Většinou se jedná o název vstupního souboru.
6. Vygenerovaný kód. Obsahuje výpočet vestavěných bloků.
7. Nastaví počet pomocných proměnných (používají se pro práci s vestavěnými bloky).
8. Určuje kolik pomocných proměnných (bráno od 0) je potřeba vrátit do výsledků kvůli použití v některé `watch`.

Simulační třída je navíc označena atributem `SimulationRuntime`. Díky tomu pak systém pomocí reflexe tuto třídu nalezne a může tak spustit výpočet.

B.2 Příklad použití šablony

Nyní si ukážeme, jak vypadá vygenerovaná simulační třída pro konkrétní zdrojový kód.

Následující zdrojový kód uložený v souboru `exp.txt`

```
const tmax = 5, step = 1e-2, record = 1e-1;

system {
        var x;
        x' = x & 1;
    watch x;
}
```

bude přeložen do podoby zapsané v následujícím výpisu.

```
using Core;
using BackEnd.CIL.Runtime;

namespace Tksl.Simulation
{
    [SimulationRuntime]
    public class Simulation_fb3bdfb6059b498faf2033ec445d77ea
        : SimulationBase
    {
        public Simulation_fb3bdfb6059b498faf2033ec445d77ea()
            : base(1,
                simulationName: "exp.txt",
                newAuxiliaryVariablesCount: 0,
                newAuxVarsToResultsCount: 0)
```

```
        {
            base.SetDefaultParameters(
            0.1d, 5d, PrecisionMode.MethodStep, 0.01d);
        }

        public override double DerivationFunction(
            int integratorNumber,
            double actualTime,
            double integratorOutput)
        {
            switch (integratorNumber)
            {
              case 0:
                return integratorOutput;

              default:
                throw new SimulationException(
                                "Invalid integrator number.");
            }
        }

        protected override void ComputeAuxiliaryVariables(
                 double actualTime)
        {
        }

        protected override void InitialConditions()
        {
            this.output[0] = 1d;
        }
    }
}
```