ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

FAKULTA DOPRAVNÍ

Zdeněk Barnet

APROXIMACE S VYUŽITÍM GENETICKÉHO PROGRAMOVÁNÍ

Diplomová práce

2015

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta dopravní
d ě k a n
Konviktská 20, 110 00 Praha 1

K614 ..... Ústav aplikované informatiky v dopravě

ZADÁNÍ DIPLOMOVÉ PRÁCE
(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení studenta (včetně titulů):

Bc. Zdeněk Barnet

Kód studijního programu a studijní obor studenta:

N 3710 – ID – Inženýrská informatika v dopravě a spojích

Název tématu (česky): **Aproximace s využitím genetického programování**

Název tématu (anglicky): An approximation using genetic programming

Zásady pro vypracování

Při zpracování diplomové práce se řiďte osnovou uvedenou v následujících bodech:

- prostudujte principy genetického programování
- seznamte se s různými metodami aproximace
- navrhněte a ověřte možnost aproximace průběhu hledáním vhodné diferenciální rovnice pomocí genetického programování
- porovnejte výsledky s metodou gramatické evoluce
- navrhněte využití této metody v dopravě

Rozsah grafických prací: podle pokynů vedoucího práce

Rozsah průvodní zprávy: minimálně 55 stran textu (včetně obrázků, grafů a tabulek, které jsou součástí průvodní zprávy)

Seznam odborné literatury:

Mařík V. a kol.: Umělá inteligence 3, Academia 2001

Mařík V. a kol.: Umělá inteligence 4. Academia 2003

Zelinka I. a kol.: Evoluční výpočetní techniky - principy a aplikace, BEN 2009

Langdon W.B, Poli R.: Foundations of Genetics Programming, Springer-Verlag 2002

Vedoucí diplomové práce: **doc. Ing. Vít Fábera, Ph.D.**

Datum zadání diplomové práce: **30. června 2014**
(datum prvního zadání této práce, které musí být nejpozději 10 měsíců před datem prvního předpokládaného odevzdání této práce vyplývajícího ze standardní doby studia)

Datum odevzdání diplomové práce: **31. května 2015**
a) datum prvního předpokládaného odevzdání práce vyplývající ze standardní doby studia a z doporučeného časového plánu studia
b) v případě odkladu odevzdání práce následující datum odevzdání práce vyplývající z doporučeného časového plánu studia

L. S.

doc. Dr. Ing. Tomáš Brandejský
vedoucí
Ústavu aplikované informatiky v dopravě

prof. Dr. Ing. Miroslav Svítek
děkan fakulty

Potvrzuji převzetí zadání diplomové práce.

Bc. Zdeněk Barnet
jméno a podpis studenta

V Praze dne .....30. června 2014

Poděkování

Chtěl bych velmi poděkovat vedoucímu diplomové práce Ing. Vítu Fáberovi, Ph.D. za odborné konzultace a vedení.

Prohlášení

Předkládám tímto k posouzení a obhajobě diplomovou práci zpracovanou na závěr studia na ČVUT v Praze Fakultě dopravní.

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

Nemám závažný důvod proti užívání tohoto školního díla ve smyslu § 60 Zákona č.121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne …..... …....

podpis

Abstrakt

Diplomová práce se zaměřuje na aproximaci signálu využitím jedné z evolučních technik, genetického programování, a porovnání výsledků s technikou druhou, gramatickou evolucí. Metoda je implementována v jazyce C++ s pomocí objektově orientovaného programování pro reprezentaci a operace s diferenciálními rovnicemi, které signál popisují.

Abstract

This diploma thesis is focused on signal approximation using one of the evolutionary techniques – genetic programming, and comparing the results with another technique – grammatical evolution. The method is implemented in C++ with the help of object-oriented programming to represent and work with differential equations, that describe the signal.

Klíčová slova

Genetické algoritmy, Genetické programování, Aproximace signálu, Diferenciální rovnice

Keywords

Genetic algorithms, Genetic programming, Signal Approximation, Differential equations

Obsah

1 Úvod....7
2 Teorie....8
2.1 Úvod do teorie....8
2.2 Evoluční výpočetní techniky (EVT)....8
2.3 Genetické algoritmy (GA)....11
2.4 Genetické programování (GP)....12
2.4.1. Stromy....12
2.4.2. Počáteční populace stromů v GP....14
2.4.3. Rekombinační operátory v GP....14
2.5 Gramatická evoluce (GE)....17
2.6 Metody aproximace....19
2.7 Numerické řešení diferenciálních rovnic....20
2.7.1. Eulerova metoda....21
2.7.2. Metoda Runge-Kutta....22
3 Řešení....23
3.1 Popis funkce výsledného programu....23
3.2 Struktura souborů celého programu....23
3.3 Implementace diferenciální rovnice – soubor dif_rce.cpp....24
3.3.1. Tvar diferenciální rovnice....24
3.3.2. Diff_R – Třída implementující diferenciální rovnici....25
3.4 Implementace GP – soubor gp.cpp....27
3.4.1. GPTree - Třída pro reprezentaci stromu....27
3.4.2. Generování stromu pomocí třídy GPTree....28
3.4.3. Genetické operátory....30
3.4.4. Objektový diagram....31
3.5 Implementace Fitness funkce....31
3.5.1. Třída Fitness....31
3.5.2. Použití třídy Fitness ve funkcích....32
3.6 Funkce v hlavním modulu – soubor diff_rce_gp.cpp....34
3.6.1. Implementace řešení diferenciální rovnice metodou Runge-Kutta....34
3.6.2. Funkce pro turnajovou selekci....37
3.6.3. Funkce main....38
4 Používání programu....43

4.1 Spouštění souboru diff_rce_gp.exe....43
4.2 Spouštění programu pomocí dávkového souboru gp.bat....44
4.3 Ukázkový výpis výstupu....44
4.4 Vysvětlivky k výstupu....46
5 Experimenty a porovnání výsledků s GE....47
5.1 Použité testovací rovnice....47
5.2 První sada testovacích rovnic....48
5.3 Druhá sada testovacích rovnic....51
5.4 Třetí sada testovacích rovnic....52
5.5 Rozbor situací, kdy nebylo nalezeno správné řešení....53
6 Porovnání s výsledky GE....56
6.1 Krátké shrnutí výsledků GE pro účely porovnání....56
6.2 Porovnání výsledků obou metod....56
7 Závěr....57
7.1 Shrnutí provedených experimentů....57
7.2 Srovnání GP a GE....57
7.3 Zhodnocení a návrhy pro zlepšení....57
8 Použité zdroje....60
9 Seznam obrázků....61
10 Seznam tabulek....61
11 Přílohy....62
11.1 Pomocný program pro generování průběhů diferenciálních rovnic....62

SEZNAM POUŽITÝCH ZKRATEK

EVT	Evoluční výpočetní techniky
GA	Genetické algoritmy
GP	Genetické programování
GE	Gramatická evoluce
OOP	Objektově orientované programování
UML	Unified Modeling Language
EEG	Elektro-encefalogram

1 Úvod

Diplomová práce se zabývá aproximací signálů pomocí genetických algoritmů a vyjádření výsledku ve formě diferenciální rovnice. Obsahově navazuje na předchozí práci bakalářskou [6] a přejímá některé její základní postupy, které se týkají numerického řešení diferenciálních rovnic. S ohledem k výsledkům bakalářské práce, které naznačují horší využitelnost gramatické evoluce při řešení problematiky aproximace signálů, je zde použita jiná evoluční technika, a to genetické programování. Algoritmus je implementován pomocí programovacího jazyka C++. Výsledný program přijímá sadu hodnot v čase a výstupem je diferenciální rovnice, která nejlépe popisuje daný průběh signálu. Tato rovnice může být libovolného řádu, zaměřuji se ale nejvíce na rovnice druhého řádu s ohledem na možné využití programu při aproximaci signálu EEG.

2 Teorie

2.1 Úvod do teorie

Tato část práce přibližuje a objasňuje základní principy použité při řešení problému. Popsány jsou v následujících kapitolách: Evoluční výpočetní techniky, Genetické algoritmy, Genetické programování, Gramatická evoluce, Metody aproximace, Numerické metody řešení diferenciálních rovnic. Jedná se o kompilaci a parafrázi[1] informací z různých odborných zdrojů[2].

2.2 Evoluční výpočetní techniky (EVT)

Evoluční výpočetní techniky jsou stochastické prohledávací algoritmy. Slouží tedy pro řešení úloh optimalizace. V situacích, kdy je prohledávaný prostor možných řešení úlohy příliš velký a prozkoumání všech možností již není časově výhodné, nabízí EVT jednu z alternativ. „Stochastické“ znamená, že při prohledávání prostoru řešení se využívá náhodných jevů, respektive pravděpodobností a to tak, že se aplikují mechanismy známé z procesu evoluce v přírodě. Konkrétně jde o předávání vlastností rodičů potomkům, o vznik nových mutací a o přirozený výběr, který zajistí, že přežijí jen „silní“ jedinci. Každý jedinec v tomto případě představuje jedno možné řešení úlohy.

Samotný algoritmus EVT vychází z počáteční množiny, označované jako populace nebo také generace. Jedná se o náhodně vygenerované jedince s tím, že v některých případech je vhodné do této počáteční generace zasáhnout a zajistit, aby byla co nejrozmanitější. V opačném případě by hledání mohlo uváznout pouze v lokálním extrému. V jednotlivých iteracích algoritmu je potom vždy z aktuální populace vytvořena nová vylepšená pomocí mechanismů evoluce neboli rekombinačních operátorů, jimiž jsou nejčastěji mutace a křížení.

Při mutaci dochází k vývoji jedince drobným pozměněním v jeho genetickém kódu.

[1] kompilace a parafráze dle definice, kterou udává Metodický pokyn č. 1/2009 "O dodržování etických principů při přípravě vysokoškolských závěrečných prací", dostupné online: *http://www.fd.cvut.cz/pro-studenty/dokumenty/ssz/metodicky-pokyn-1-2009.pdf*

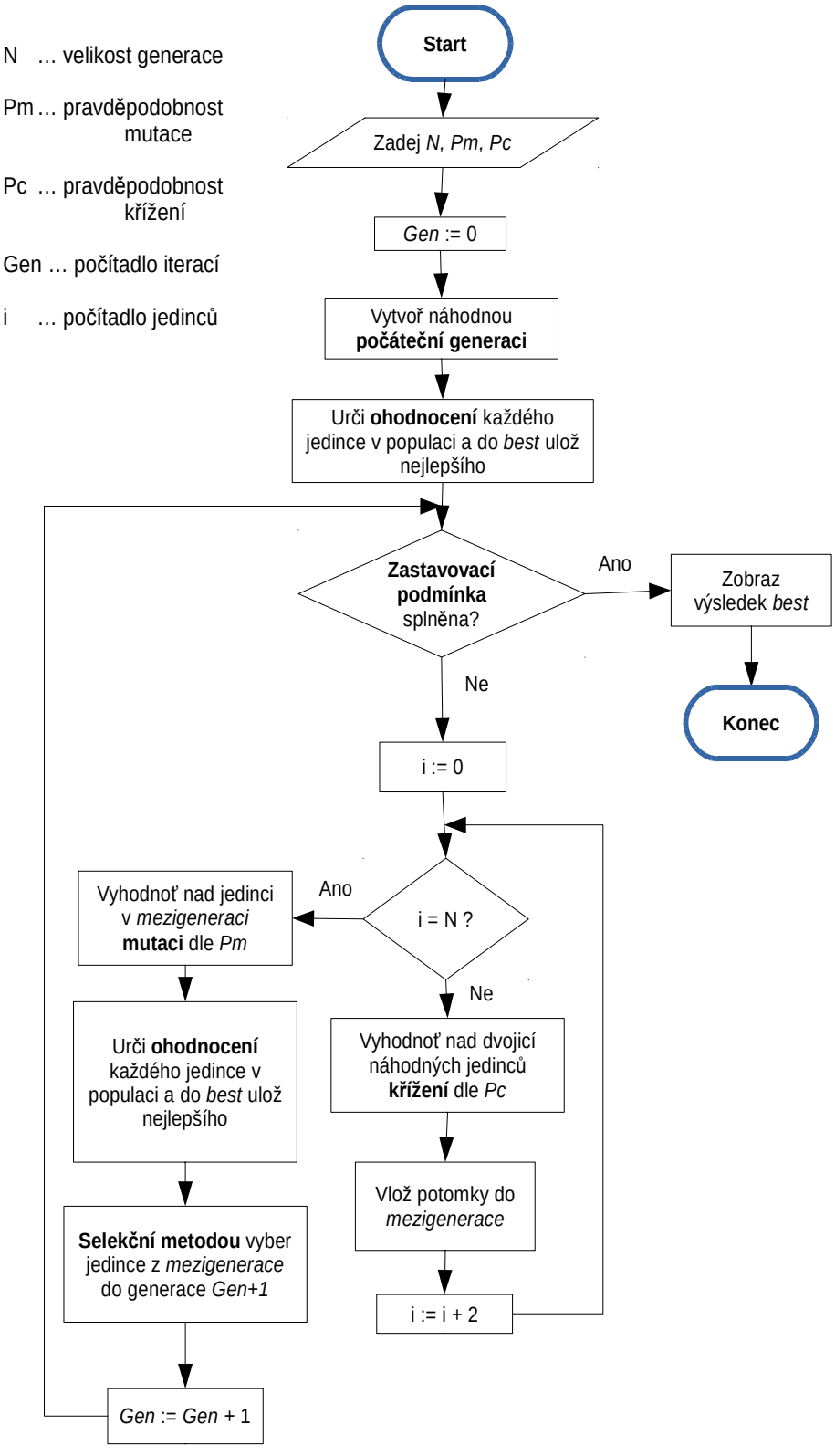
[2] zdroje [1] a [2] pro kapitoly 2.2 – 2.5 a zdroje [3] a [4] pro kapitoly 2.6 – 2.7

Při křížení předávají dva či více jedinců - „rodičů“ některé své vlastnosti novým jedincům - „potomkům“.

Výběr konkrétních způsobů křížení i mutace závisí na podstatě řešené úlohy a také volbě řešitele. Rodičovští jedinci jsou vybíráni náhodně a ne vždy dojde mezi nimi ke křížení. To určí pravděpodobnost křížení, která je jedním z parametrů EVT. Stejně tak každý jedinec neprojde mutací. To určí další z parametrů – pravděpodobnost mutace. Jedinci, kteří byli vybráni jako rodiče, ale kterým náhodný proces určil, že se nebudou křížit ani mutovat, postupují do nové generace v nezměněném stavu. Výsledná generace se typicky opět skládá ze stejného počtu jedinců (ale existují i algoritmy, kde se velikost generací v průběhu výpočtu mění). Velikost generace je dalším parametrem EVT.

Všichni jedinci jsou ohodnoceni pomocí fitness funkce (také účelová nebo kriteriální funkce), která vybraným způsobem určí, jak je které řešení kvalitní. Na základě toho můžeme jedince mezi sebou porovnávat a do nové generace vybírat jen ty nejlepší s tím, že jedinec může být vybrán i vícekrát. Tomuto procesu se říká selekce. Ta by měla, ze stejného důvodu jako bylo zmíněno výše u počáteční generace, udržet rozmanitost jedinců, ale zároveň zajistit zkvalitnění populace a přežití nejlepšího jedince.

Jakmile je algoritmus spuštěn, musí existovat ukončující podmínka, kdy se má zastavit hledání. Možností je více: byl nalezen optimální výsledek, bylo dosaženo limitu počtu iterací (= počet generací, které vzniknou) nebo nedošlo-li za posledních několik generací k žádnému zlepšení. Celý postup EVT znázorňuje vývojový diagram na Obr. 2.1.



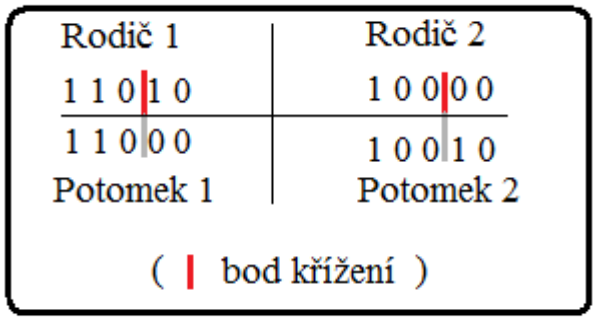
Obr. 2.1: Vývojový diagram GA s křížením a mutací

Pozn. Takto popsaný algoritmus s křížením vyžaduje, aby velikost generace *N* byla sudé číslo.

2.3 Genetické algoritmy (GA)

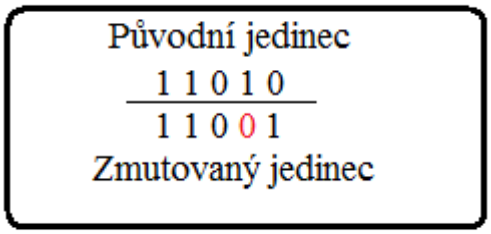
GA jsou jednou z metod EVT. Jejich hlavní charakteristiku je reprezentace jedince jako řetězce hodnot nazývaného chromozom. Každá pozice v řetězci (alela) obsahuje hodnotu z pevně daného rozsahu (gen), přičemž nejběžněji se používají binární hodnoty (0 nebo 1).

Křížení se typicky provede metodou jednobodového křížení, kdy se náhodně určí bod, kterým vedeme řez v obou chromozomech rodičů. Potomek následně vznikne z první části jednoho a druhé části z druhého rodiče a analogicky z prohozených částí vznikne druhý potomek. (viz.Obr. 2.2)



Obr. 2.2: Jednobodové křížení

Mutace se typicky provede inverzí jednoho náhodně zvoleného bitu jedince. (viz. Obr. 2.3)



Obr. 2.3: Mutace

Mezi nejběžnější metody **selekce** patří ruletová selekce, pořadová selekce nebo turnajová selekce [2]. V současnosti se nejčastěji používá **turnajová selekce**, při které se náhodně vybere několik jedinců z původní populace (obvykle dva), porovná se jejich kvalita

a vítěz(ové) tohoto turnaje postupují do nové generace. Respektive stanoví se pravděpodobnost (běžně 100%, ale není to podmínkou), že dále postoupí vítěz turnaje, v opačném případě by mohl postoupit naopak poražený. Důvodem pro postup slabšího jedince je udržení již zmiňované rozmanitosti a zabránění uváznutí v lokálním extrému, ke kterému by mohlo dojít, pokud by se generace skládala jen z většího množství stejných nebo velmi podobných jedinců. Turnaje se pořádají tak dlouho, dokud není zaplněna celá nová generace.

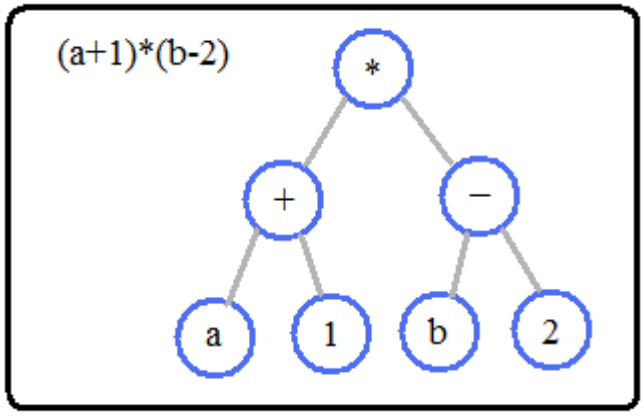
2.4 Genetické programování (GP)

Tato EVT vychází z klasických GA, ale má odlišný způsob reprezentace jedince a tím svoje specifické využití, kterým je syntéza programů, algoritmů nebo výrazů. Jedince v GP představují tzv. stromy, popřípadě orientované grafy nebo síť uzlů v kartézských souřadnicích [2].

2.4.1. Stromy

Strom je souvislý graf bez kružnic (může být orientovaný i neorientovaný). Na vrcholu je jeden uzel nazývaný kořen, ze kterého vychází další, **vnitřní uzly**, a nakonec potom i koncové neboli **listové uzly** (u orientovaného grafu má kořen vstupní stupeň hodnotu 0 a list výstupní stupeň hodnotu 0). Počet úrovní uzlů pod kořenem se nazývá **hloubka** a zároveň je to délka nejdelší cesty od kořene k listu.

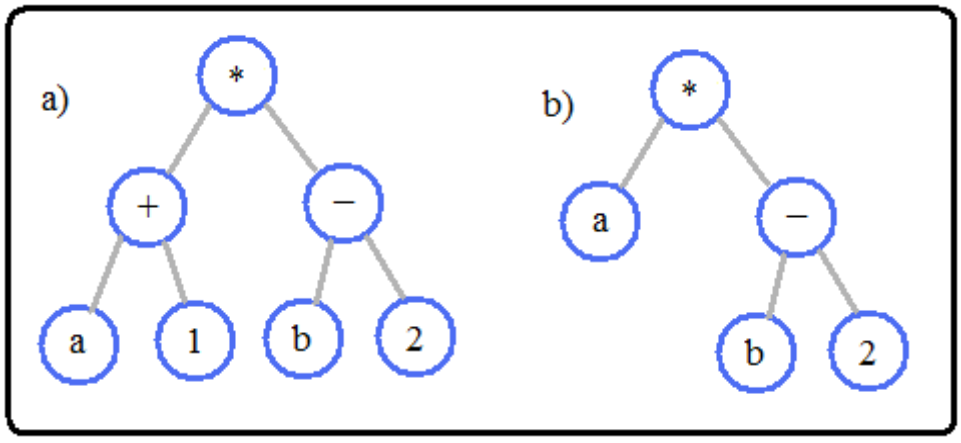
V GP má strom ohodnocené uzly symboly z množin terminálních a neterminálních symbolů. Představíme-li si, že jedincem je matematický výraz, potom neterminálními symboly (neterminály) jsou funkce a matematické operace a jsou jimi ohodnoceny kořen a vnitřní uzly, terminálními symboly (terminály) jsou proměnné a konstanty a jsou jimi ohodnoceny listy. Neterminály mohou mít různou aritu (počet argumentů funkce) a tím i různý počet větví. Oproti tomu terminály mají logicky aritu nulovou. Ukázka stromu, který reprezentuje výraz *(a+1)*(b-2)* je na Obr. 2.4.



Obr. 2.4: Stromová struktura reprezentující výraz (a+1)*(b-2)

V průběhu evoluce GP je u stromů důležité sledovat jejich hloubku. Necháme-li algoritmus generovat stromy zcela náhodně bez jakéhokoli dalšího zásahu, dojde k tomu, že vzniknou stromy s tak velkou hloubkou, že jim odpovídající výrazy jsou až nesmyslně složité a zbytečně tím vzrůstají požadavky na výpočetní výkon. Z toho důvodu se využívají různá omezení pro maximální hloubku stromu a její kontrolu během rekombinací.

Stromy se generují od kořene, kterým je v netriviálním případě neterminální symbol, který zároveň určuje počet následovníků. Existují dvě metody generování stromů: **úplná** a **růstová**. (viz. Obr. 2.5)

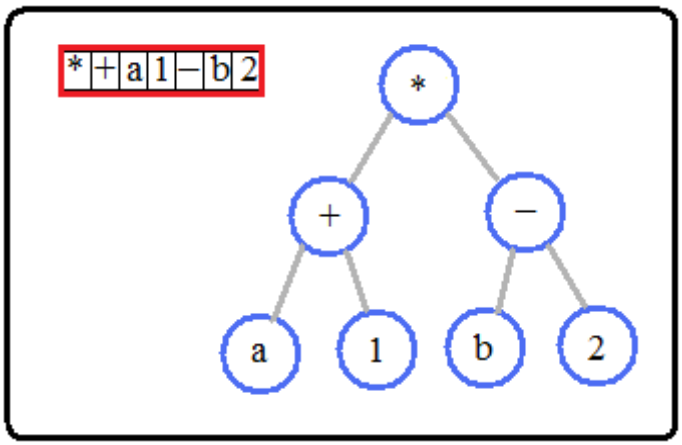


Obr. 2.5: Strom hloubky 2 generovaný (a) úplnou a (b) růstovou metodou

Úplná metoda vybírá symboly pouze ze skupiny neterminálů a až ve chvíli dosažení

maximální povolené hloubky se vybere terminál. Růstová metoda oproti tomu povoluje výběr terminálu i jinde než na nejspodnějším listovém uzlu.

V programátorské praxi se strom reprezentuje **lineárně**, tedy polem (vektorem), jehož každý prvek odpovídá jednomu uzlu grafu. Při převodu stromu na lineární reprezentaci se strom prochází **rekurzivně metodou pre-order**, tj. v pořadí kořen, levý podstrom, pravý podstrom a v tomto pořadí se uzly ukládají do pole. Uvedený zápis se označuje termínem „polská notace“, kdy první prvek pole odpovídá kořenu stromového grafu, druhý a následující prvky vždy nejprve levému a až poté pravému podstromu. (viz. Obr. 2.6)



Obr. 2.6: Lineární reprezentace stromu

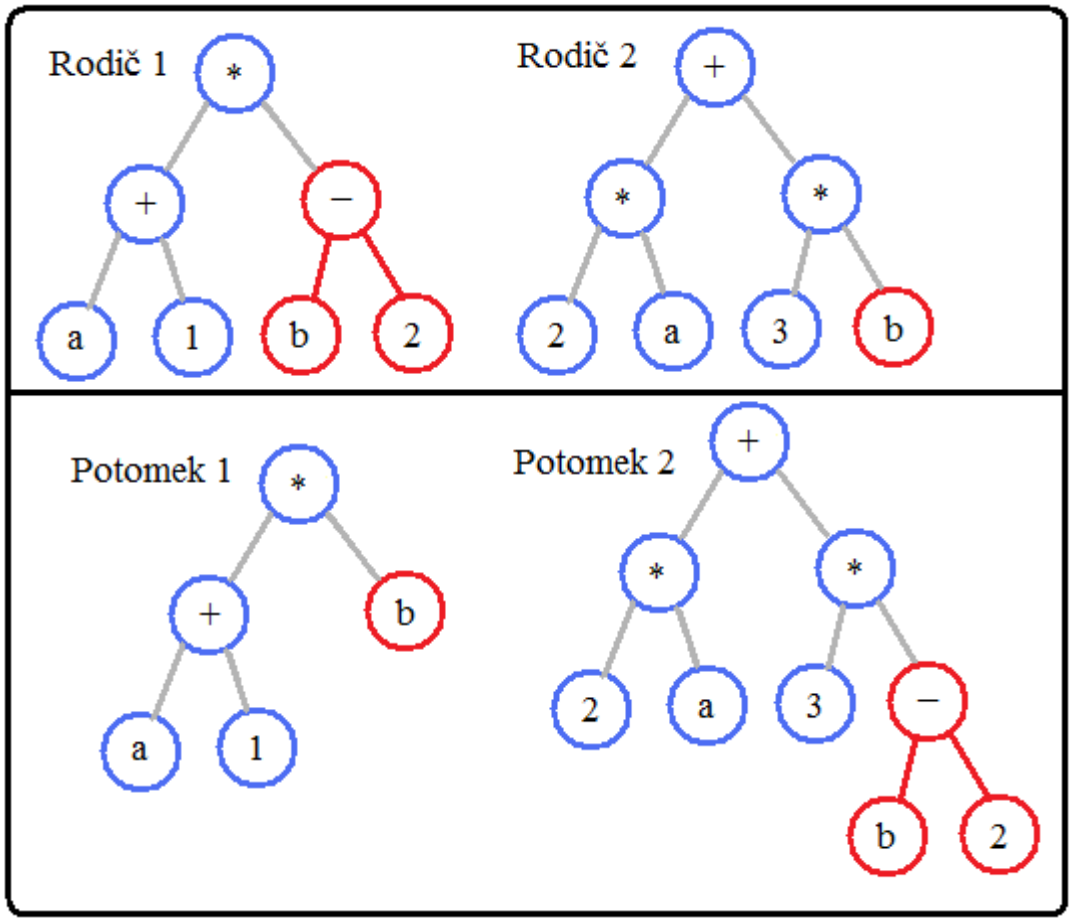
2.4.2. Počáteční populace stromů v GP

Úvodní (nultá) generace má velký vliv na další vývoj všech ostatních generací. Důležitým faktorem je zde hloubka stromů. Je vhodné jednak zajistit zastoupení stromů co nejrůznější hloubky a jednak omezit maximální.

2.4.3. Rekombinační operátory v GP

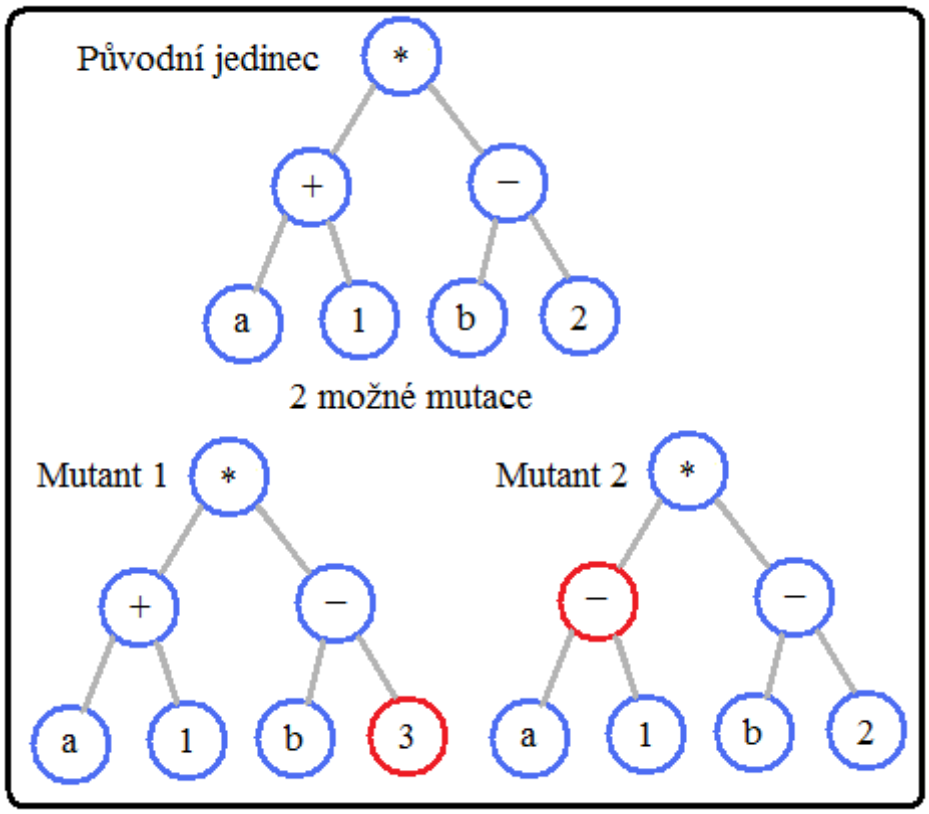
Primárně se v GP jako rekombinační operátory využívají stejně jako u GA: křížení a mutace. Při křížení dochází k prohození dvou podstromů. Zvláštním případem je, pokud oba podstromy jsou koncové uzly. Tomu bychom pro lepší průběh mohli chtít zabránit, protože jde jednak jen o velmi malou změnu a jednak velmi podobně funguje mutace.

Křížení v GP ilustruje obrázek Obr. 2.7. Na obrázku jsou v horní části rodičovské stromy. Červeně jsou označeny křížící se podstromy. V dolní části obrázku jsou potomci křížení. V ukázce pro jednoduchost nijak nezohledňujeme maximální povolenou hloubku stromu.



Obr. 2.7: Křížení stromových struktur

Algoritmus **mutace** může být různý, většinou jde o záměnu náhodně zvoleného uzlu nově vygenerovaným podstromem. V tomto případě provádím mutaci jednodušeji nahrazením uzlu novým uzlem se stejnou aritou, tedy terminálu terminálem a funkce funkcí se stejným počtem argumentů, které zůstanou zachovány z původního jedince. Situaci ilustruje následující obrázek Obr. 2.8.



Obr. 2.8: Mutace stromových struktur

Kromě těchto výše zmíněných primárních rekombinačních operátorů se v GP někdy využívají také další sekundární, nejčastějšími jsou např. **permutace, editace, zapouzdření a decimování**. Ty ale v práci použity nebyly, proto jsou uvedeny jen pro informaci.

Při rekombinacích může snadno docházet k nekontrolovatelnému nárůstu hloubky stromů, když se nahradí listový uzel podstromem nebo celý podstrom, který zasahuje až do největší hloubky, podstromem s větší hloubkou. To má velmi negativní vliv na výpočetní výkon a dobu a pravděpodobně to ani nepřinese lepší výsledky, spíše naopak. Proto se zavádí do rekombinací pravidla, která tomuto zamezí.

2.5 Gramatická evoluce (GE)

Gramatická evoluce je další, poněkud mladší, evoluční technikou, která v sobě spojuje prvky GP a gramatik používaných v kybernetice. Touto metodou jsem se zabýval v bakalářské práci [6] a díky tomu bude možno v závěru této diplomové práce porovnat výsledky obou metod.

Pro principiální porozumění GE uvádím zkrácený popis, podrobněji se lze o tématu dočíst ve výše zmíněné bakalářské práci nebo v knize Umělá inteligence 3 a 4 [1] [2].

Gramatika je v kybernetice definována jako čtveřice G = <N,T,P,S>, kde N je množina neterminálních symbolů, T je množina terminálních symbolů, P je množina pravidel pro přepis neterminálních symbolů, S je počáteční (startovací) neterminální symbol gramatiky, kterým generování „vět“ pomocí gramatiky začíná. Existují 4 typy gramatik dle Chomského klasifikace: neomezené, kontextové, bezkontextové a regulární, v informatice mají význam poslední dvě jmenované. Pro generování matematických výrazů použijeme bezkontextovou gramatiku.

Příklad jednoduché bezkontextové gramatiky:

G = <N,T,P,S>
N = { *výraz, operace, proměnná* }
T = {+, -, a, b}

P = {

výraz = proměnná operace proměnná	(0)
výraz = (výraz) operace (výraz)	(1)
operace = +	(0)
operace = -	(1)

proměnná = a (0)

proměnná = b (1)

}

S = { *výraz* }

GE potom funguje tak, že každý jednotlivý prvek chromozomu říká, které pravidlo se má na aktuální neterminální symbol použít, čímž se postupně rozvíjí celý řetězec, dokud nejsou všechny neterminální symboly nahrazeny terminálními.

Příklad funkce GE na gramatice z předchozího příkladu:

Jedinec je řetězec binárních hodnot, kde hodnota *0* znamená použít první pravidlo pro daný neterminál a *1* druhé, chápáno podle pořadí v němž jsou vypsány ve výčtu.
Mějme jedince = 0 1 0 0 1.

čtení jedince		rozvoj výrazu		
0 1 0 0 1		*výraz*		
0 1 0 0 1	=>	*proměnná*	*operace*	*proměnná*
0 1 0 0 1	=>	b	*operace*	*proměnná*
0 1 0 0 1	=>	b	+	*proměnná*
0 1 0 0 1	=>	b	+	a
0 1 0 0 1				

Jak je vidět, je-li jedinec delší, než kolik je potřeba přepsat neterminálů, zbylé hodnoty se nevyužijí. Příklad byl také ideální v tom, že pro každý neterminál byla právě dvě pravidla. V reálné situaci je pravidel různý počet, proto jsou potřeba některé další úpravy[3], které už ale zde není potřeba rozvádět.

Jak je patrné, hlavním využitím GE je syntéza a hledání výrazů libovolného „jazyka“ nebo jakéhokoliv problému, v němž řešitel spatří podobnost s jazykem a funkcí gramatiky.

[3] podrobně rozebrané v [6]

2.6 Metody aproximace

Záměrem této práce je aproximovat signál pomocí diferenciální rovnice, pro úvod do problematiky by ale bylo dobré se také seznámit s nezbytnou teorií a různými metodami aproximace.

Základní úloha aproximace je následující. Je dáno *n* tzv. uzlových bodů $x\ (x_0, x_1 \dots x_{n-1})$ a jim odpovídajících hodnot $y\ (y_0, y_1 \dots y_{n-1})$. Aproximace potom spočívá v tom, že z předem zvolené třídy funkcí se vybere taková funkce φ, jejíž hodnoty v *x* mají nejblíže k y_i.

Speciálním případem aproximace je **interpolace**, při které se požaduje přesná shoda v uzlových bodech. Toho lze docílit různým způsobem, nejjednodušším je pomocí lineární kombinace aproximačních funkcí φ_i. Tato metoda bývá označována jako prostá interpolace a omezuje se na hledání koeficientů lineární kombinace. Častější je ale interpolace polynomem s omezením maximálního řádu. Výhodou této metody je, že výsledný polynom lze potom snadno derivovat, integrovat nebo zobrazovat ve Fourierově transformaci. Pro sestrojení polynomu existuje několik postupů. Jsou jimi např. Lagrangeova konstrukce a Newtonova konstrukce a v případě, že nehledáme celý polynom, ale jen další bod, můžeme využít Nevillův algoritmus (detaily o všech metodách viz. skripta Numerické metody [3]). Polynomem aproximuje funkci také z matematické analýzy oblíbená Taylorova řada, ta ale není velmi vhodná, nabízí dobrou přesnost jen v okolí bodu x_0 na úkor vzdálenějších bodů. Navíc vyžaduje analytické zadání funkce, aby šlo spočítat potřebné derivace. Vylepšením těchto metod je tzv. (kubický) spline. Pro něj je charakteristické, že každý bod ovlivňuje pouze svoje blízké okolí a na vzdálenější už větší vliv nemá, na rozdíl od polynomů. Někdy požadavek na přesný průchod všemi body není úplně nezbytný a lepšího výsledku můžeme dosáhnout obecným aproximačním kritériem minimalizace rozdílu hodnoty aproximační funkce $\varphi(x_i)$ a zadaného y_i.

Jedním z klasických způsobů aproximace je pomocí **metody nejmenších čtverců,** kde je výše zmíněné kritérium nahrazeno kritériem kvadratickým H_2 neboli kvadrátem rozdílu $\varphi(x_i)$ a zadaného y_i v sumě přes všechna $i < n$.

$$H_2 = \Sigma(\varphi(x_i) - y_i)^2$$

V praxi se ale použijí i jiná kritéria, např. $H_1 = \Sigma|\varphi(x_i) - y_i|$, popřípadě někdy může dávat větší smysl minimalizace maximální odchylky $\varphi(x_i)$ a y_i, tedy minimalizace výrazu

$H_0 = max|\varphi(x_i) - y_i|$,

která se využije v **Čebyševově aproximaci polynomem**. Výběr potom záleží na úvaze řešitele a vhodnosti pro řešenou úlohu.

Abychom mohli aproximovat průběh funkce zadaný uzlovými body diferenciální rovnice, potřebujeme diferenciální rovnici vyřešit. K tomu využijeme numerické řešení.

2.7 Numerické řešení diferenciálních rovnic

Numerické metody řešení se často používají pro simulace reálných dynamických systémů popsaných diferenciálními rovnicemi. Poskytnou nám hodnoty y v tomto případě s konstantním krokem. Hledané řešení je tedy funkce diskretizovaná. Numerických metod existuje několik, např. Eulerova, Stoermerova, Adamsovy metody a mezi nejvíce používané metody patří metoda Runge-Kutta 4. řádu, která je zde použita.

Předpokládám obecný tvar diferenciální rovnice prvního řádu

$$y'(x)=f(x,y(x)) \quad , \qquad (2.7.1)$$

kde $f(x,y(x))$ je funkce pravé strany o dvou reálných proměnných a počáteční podmínkou je $y(x_0) = y_0$. V případě, že f nezávisí na y, tj. $f(x,y) = g(x)$ dostávám rovnici

$$y'(x) = g(x) \ ,$$

jejímž řešením je:

$$y(x)= y_0+\int_{x_0}^{x} g(t)\,dt, \quad x\in\langle x_0,x_n\rangle \qquad (2.7.2)$$

Příklad: $y'(x)=\sqrt{y(x)}, \ y(0)=0$

Pro $x\geqslant 0$ má triviální řešení $y(x)=0$, ale i $y(x)=\frac{x^2}{4}$.

Je tedy vidět, že počáteční podmínka nezaručuje jednoznačnost řešení. K tomu slouží

Lipschitzova podmínka. Ta říká, že pro Cauchyho počáteční úlohu: $y'(x)=f(x,y(x))$ s počáteční podmínkou $y(x_0) = y_0$ existuje právě jedna reálná funkce y na intervalu $\langle x_0, x_n\rangle$, která je jejím řešením. Lipschitzova podmínka zní:

$$\exists L\in\Re \;\; \forall x \in\langle x_0, x_n\rangle \;\; \forall y_1, y_2 \in \Re: \;\; |f(x, y_1) - f(x, y_2)| \;\le\; L|y_1 - y_2| \qquad (2.7.3)$$

Cauchyho úlohu přepíši na integrální rovnici:

$$y(x)=y_0+\int_{x_0}^{x} f(t, y(t))\,dt, \qquad (2.7.4)$$

Navíc se počítá jen s konečným počtem hodnot, rozdělím proto interval $\langle x_0, x_n\rangle$ na n intervalů délky h, kterou budu dále označovat jako **krok**. Postupně generuji posloupnost y_0 , … y_n. V analogii se vzorcem (2.7.4) dostávám:

$$y(x_{i+1}) = y(x_i) + \int_{x_i}^{x_{i+1}} f(t, y(t))\,dt \qquad (2.7.5)$$

$y(x_i)$ dále pro zjednodušení značím pouze jako y_i . Řešení diferenciální rovnice spočívá ve výpočtu: $y_{i+1} = y_i + \Delta y_i$, kde Δy_i značí právě odhad integrálu z (2.7.5), y_i mám odhadnutý z předchozího kroku a pro $i=0$ z počáteční podmínky. Způsob, jakým se odhad integrálu provádí, je to, čím se jednotlivé metody liší.

2.7.1. Eulerova metoda

Je nejjednodušší metodou pro řešení diferenciálních rovnic. Integrál počítá metodou levého odhadu, kdy integrovanou funkci $f(t, y(t))$ nahradím konstantou rovnou její hodnotě v levém krajním bodě uvažovaného intervalu :

$$\Delta y_i = \int_{x_i}^{x_{i+1}} f(t, y(t))\,dt = \int_{x_i}^{x_{i+1}} f(x_i, y_i)\,dt = h\cdot f_i \qquad (2.7.6)$$

a potom $y_{i+1} = y_i + h\cdot f_i$. Geometrický význam je takový, že f_i je směrnicí úsečky z bodu (x_i , y_i) do bodu (x_{i+1} , y_{i+1}). Toto je metoda 1. řádu.

Následující metoda je **první modifikace Eulerovy metody** a metodou 2. řádu. Funkce $f(t, y(t))$ se v ní nahrazuje opět konstantou ale tentokrát jde o hodnotu uprostřed intervalu integrace tzn. za $t = x_i + h/2$. Otázkou je co dosadit za y(t), tedy $y(x_i + h/2)$. Proto udělám pomocný krok Eulerovou metodou a tím odhad v bodě $x_i + h/2$, který má hodnotu

$y_i + \frac{h}{2}\cdot f_i$. Ve vzorci (2.7.6) se dosadí za $f(t, y(t)) = f(x_i + \frac{h}{2},\ y_i + \frac{h}{2}\cdot f_i)$.

Poslední metoda před Runge-Kuttovou je **druhá modifikace Eulerovy metody** a je také 2. řádu. Funkce $f(t, y(t))$ se nahrazuje lineární funkcí proloženou hodnotami v obou krajních bodech intervalu. Levým krajním bodem je $f(x_i, y_i)$ jako u klasické Eulerovy metody a pravý krajní bod získám opět pomocným odhadem ale v bodě x_{i+1} neboli $x_i + h$.

Výsledný odhad integrálu: $\Delta y_i = \frac{h}{2}\cdot(f(x_i, y_i) + f(x_{i+1}, y_i + h\cdot f_i))$ (2.7.7)

2.7.2. Metoda Runge-Kutta

Jedná se také o jednokrokovou metodu, při které se počítá nová hodnota opět pomocí hodnoty předchozího kroku a několika pomocných bodů. Vychází z Taylorova rozvoje a Eulerovy metody, do výpočtu zahrnuje i členy vyšších řádů. Vyskytuje se ve více variantách, nejběžnější je metoda Runge-Kutta 4. řádu. Použijí se při ní 4 pomocné kroky $k_{i,j}$, podobné těm v předchozích modifikacích:

$$
\begin{aligned}
k_{i,1} &= f(x_i, y_i),\\
k_{i,2} &= f(x_i + \frac{h}{2}, y_i + \frac{h}{2}\cdot k_{i,1}),\\
k_{i,3} &= f(x_i + \frac{h}{2}, y_i + \frac{h}{2}\cdot k_{i,2}),\\
k_{i,4} &= f(x_i + h, y_i + h\cdot k_{i,3})
\end{aligned}
\qquad (2.7.8)
$$

$k_{i,1}$ je levý krajní bod intervalu.

$k_{i,2}, k_{i,3}$ jsou obojí poloviční krok *h/2*, liší se pouze způsobem pomocného odhadu $y(x_i + h/2)$ obojí vždy z předchozího kroku. Ve výsledném odhadu (2.7.9) mají váhu 2.

$k_{i,4}$ je celý krok *h*, s pomocným odhadem z předchozího kroku

Hledaný integrál je potom: $\Delta y_i = \frac{h}{6}\cdot(k_{i,1} + 2k_{i,2} + 2k_{i,3} + k_{i,4})$. (2.7.9)

A výsledný vzorec:

$$y_{i+1} = y_i + \frac{h}{6}\cdot(k_{i,1} + 2k_{i,2} + 2k_{i,3} + k_{i,4}) \qquad (2.7.10)$$

3 Řešení

3.1 Popis funkce výsledného programu

V úvodu této kapitoly nejprve popisuji, jak bude hotový program fungovat pro získání lepšího přehledu o tom, k čemu budou sloužit jednotlivé následně popisované části. Výsledný program má za úkol najít diferenciální rovnici, která nejlépe aproximuje zadaný průběh. Průběh je zadán jako posloupnost vzorků v čase s konstantním krokem. Pro testování používáme průběhy, které jsme si předem vygenerovali jako řešení zvolených diferenciální rovnic. Ke generování slouží pomocný program **diff_rce.cpp** (i s popisem je k nahlédnutí v příloze 11.1) V rámci fitness funkce se potom porovnávají dvě řady vzorků. První je získaná numerickým řešením rovnice, kterou generuje GP, druhá jsou vstupní data. Přesná metoda tohoto porovnání bude dále popsána v sekci o fitness funkci. Ve výsledku je každé vygenerované rovnici přiřazeno číselné ohodnocení (rozuměj výsledek fitness funkce). Na jeho základě se nakonec vybere rovnice, jejíž řešení nejlépe odpovídá vstupním datům.

3.2 Struktura souborů celého programu

Celý program se skládá ze souborů se **zdrojovými kódy** (soubory *.cpp*), k nim příslušných **souborů hlavičkových** (soubory *.h*) a souborů vzniklých při překladu (soubory *.o* a soubor Makefile.win). Dále zahrnujeme textové soubory se vstupními daty a dávkový soubor **gp.bat**, který nám usnadňuje spouštění programu (více viz. Kapitola 4 Používání programu).

Zdrojové kódy:

- dif_rce.cpp - implementuje třídu pro diferenciální rovnici (viz. Kapitola 3.3)
- gp.cpp - implementuje třídu pro reprezentaci jedince (viz. Kapitola 3.4)
- fitness.cpp - implementuje třídu pro fitness funkci (viz. Kapitola 3.5)
- diff_rce_gp.cpp - hlavní program (viz. Kapitola 3.6)

- diff_rce.cpp - pomocný program pro generování průběhů (viz. Kapitola 11.1)

3.3 Implementace diferenciální rovnice – soubor dif_rce.cpp

3.3.1. Tvar diferenciální rovnice

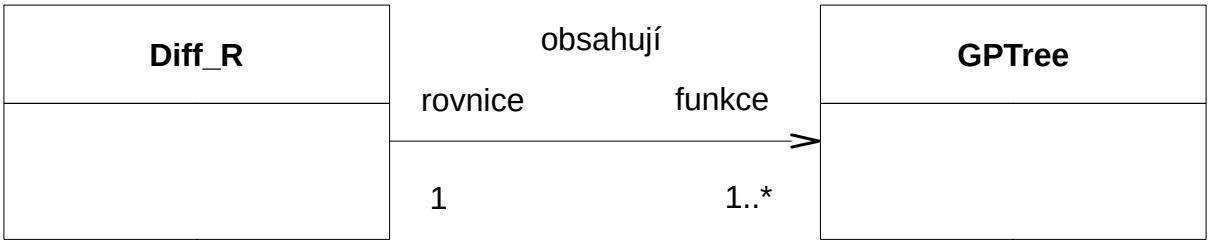
Program generuje libovolné diferenciální rovnice podle zadání buď s nenulovou pravou stranou nebo bez ní a s daným maximálním řádem. Pro potřeby následného numerického vyhodnocování pomocí metody Runge-Kutta se rovnice používá ve tvaru, kdy na levé straně je samotná derivace nejvyššího řádu a ostatní derivace se převedou na pravou stranu. Tedy:

$$y^{(n)}(t) = f_{(n-1)}(t)*y^{(n-1)}(t) + f_{(n-2)}(t)*y^{(n-2)}(t) + \ldots f_{(0)}(t)*y^{(0)}(t) + u(t)$$

Kde horní index značí derivaci, dolní index k ní příslušící funkci a „t“ v závorce je argument (zpravidla interpretovaný jako čas, jde tedy o funkci času t), “u(t)“ je případná funkce pravé strany, pokud se nejedná o homogenní diferenciální rovnici.

Aby byla úloha řešitelná, je nutné zadat počáteční podmínky, ty jsou zadány spolu se vstupními daty. Mimo to je pro jednoznačnost řešení požadováno splnění Lipschitzovy podmínky, která byla zmíněna v kapitole 2.7 Numerické řešení diferenciálních rovnic. Její kontrola ale z důvodu úspory výpočetního výkonu zahrnuta není. Předpokládáme, že GP si z principu (potlačení špatných řešení) s těmito nevhodnými rovnicemi poradí.

V kódu programu používám pro uložení každé diferenciální rovnice jeden objekt ze třídy **Diff_R** (bude popsána v následující kapitole 3.3.2) Samotná rovnice je v něm reprezentována polem, v němž každý jeho prvek nese funkci $f_{(i)}(t)$ u jedné z derivací, popřípadě u prostého *y* nebo pravé strany u nehomogenní rovnice. Každá z těchto funkcí je definována vlastním stromem, který je objektem třídy **GPTree**. Relaci mezi oběma zmíněnými třídami zachycuje diagram na Obr. 3.1. Tento diagram je zjednodušený v tom, že nezobrazuje výčet atributů a metod obou tříd, ty budou popsány v samostatných kapitolách.



Obr. 3.1: Zjednodušený diagram tříd

Jeden objekt třídy Diff_R má jednosměrnou vazbu s několika objekty třídy GPTree. Jednosměrnou v tom smyslu, že třída Diff_R v sobě nese odkaz na jí příslušející objekty třídy GPTree, ale samotná třída GPTree už nenese žádnou informaci o tom, že je částí nadřazené třídy Diff_R.

3.3.2. Diff_R – Třída implementující diferenciální rovnici

Popis třídy **Diff_R** v notaci jazyka[4] UML zobrazuje *Tabulka 1:Definice třídy Diff_R dle UML 2*. Třída je definována v souborech **dif_rce.cpp** a **dif_rce.h**, které jsou součástí elektronické .zip přílohy.

Diff_R
- degree : int - right_side : bool - functions : GPTree* - f_count : int
+ Diff_R(degree : int, right_side : bool) + get_degree() : int + evaluate(t : double, y[] : const double) : double + print(&s : string) + print(*f : FILE) + generate_random(depth : int) + Mutation(Pm : float) + friend Crossover(*rodic1 : Diff_R, *rodic2 : Diff_R, *potomek : Diff_R, *potomek2 : Diff_R) + &operator=(&r : const Diff_R) : Diff_R + ~Diff_R()

Tabulka 1: Definice třídy Diff_R dle UML 2

Atributy třídy jsou v Tabulce 1 v tomto pořadí:

- *degree* - uchovává řád rovnice,
- *right_side* - přítomnost pravé strany,
- **functions* - dynamické pole funkcí, které jsou uloženy jako stromy pomocí třídy GPTree (viz. kapitola 3.4.1 GPTree - Třída pro reprezentaci stromu). Prvky pole *functions* mají následující význam: *functions[0]* je funkce u *y(t)*, *functions[1]* je funkce $f^{(1)}(t)$ u 1. derivace $y^{(1)}(t)$ atd. až *functions[degree-1]*, kde je uložena funkce u nejvyšší

[4] Unified Modeling Language, [5]

derivace a ve *functions[degree]* je případná funkce pravé strany.

- *f_count* - celkový počet funkcí v poli *functions[]*.

Metody (operace dle UML) třídy jsou v Tabulce 1 v tomto pořadí:

- konstruktor - tvoří objekty třídy, inicializuje atributy
- vrácení hodnoty řádu,
- vyhodnocení rovnice pro vstupní data,
- tisk rovnice jako řetězce na obrazovku,
- tisk do souboru,
- generování náhodných rovnic s omezenou maximální hloubkou stromu,
- mutace se zadáním pravděpodobnosti mutace,
- křížení se zadáním rodičů a proměnnými pro potomky,
- přetížení operátoru přiřazení - důvodem je potřeba realokace více paměti
- destruktor

3.4 Implementace GP – soubor gp.cpp

3.4.1. GPTree - Třída pro reprezentaci stromu

Reprezentace jedince je implementována v souborech **gp.cpp** a **gp.h** pomocí třídy **GPTree**, znázorněno je to opět v notaci UML 2 v Tabulce 2.

GPTree
- buffer : int * - aloc_length : int - length : int - constants : static double * + num_const : static int + P_liter : static float = 0,5 + P_prom : static float = 0,5 + STEP : static const int = 20
- run(&position : int, t : double) : int - expand(int position, int depth) : int - translate_subtree(position : int, &s : string) : int - traverse(position : int) : int + evaluate(t : double) : double + print(&s : string) + gener_random(depth : int) + copy_buffer(*b : int) : int + *get_buffer() : int + get_length() : int + gener_const(min : double, max : double) : static void + free_const() : static void + crossover(&parent2 : GPTree, &offspring : GPTree) + mutation(&offspring : GPTree, Pm : float) + GPTree() + GPTree(&g : const GPTree) + &operator=(&right : const GPTree) : GPTree + ~GPTree()

Tabulka 2: Definice třídy GPTree dle UML 2

Atributy třídy GPTree jsou v Tabulce 2 v tomto pořadí:

- **buffer* – je pole, které obsahuje **lineární reprezentaci stromu**
- *aloc_length* - alokovaná délka pole jedince
- *length* – skutečně využitá délka pole
- **constants* – dynamické pole konstant v rozsahu min … max
- *num_const* – délka pole *constants*, maximálně 9999
- *P_liter* - pravděpodobnost generování literálu (proměnné, konstanty) vůči funkci
- *P_prom* - pravděpodobnost generování proměnné vůči konstantě
- *STEP* - krok, o který se zvětší velikost pole pro strom, když už je zaplněna celá alokovaná délka

Vybrané **metody** třídy GPTree z Tabulky 2:

- *run()* a *evaluate()* - vyhodnotí podstrom/strom pro hodnotu proměnné t
- *expand()* - vygeneruje náhodný podstrom
- *gener_random()* - náhodně vygeneruje strom o zadané max. hloubce, alokuje pameť a zavolá metodu *expand()*
- translate_subtree() - přeloží lineární reprezentaci stromu na formu tisknutelné rovnice
- *traverse()* - na základě pozice uzlu vyhledá konec příslušného podstromu, použije se při křížení
- *gener_const()* - vygeneruje konstanty do atributu *constants* v rozsahu od min do max
- *crossover()* - implementuje křížení v této třídě
- *mutation()* - implementuje mutaci v této třídě
- *GPTree()* a *GPTree(const GPTree &g)* – konstruktor a kopírující konstruktor
- *&operator=()* - přetížení operátoru přiřazení kvůli realokaci více paměti
- *~GPTree()* - destruktor

3.4.2. Generování stromu pomocí třídy GPTree

Nejprve podrobněji popíši jedince. Jak již bylo zmíněno, je uložen v poli *buffer.* Obsahuje hodnoty z rozsahu $\langle 0; num\ const \rangle \cup \langle 10\,000; 10\,006 \rangle$ a jejich význam znázorňuje Tabulka 3:Kódování hodnot jedince.

Hodnota	Význam
0	t - proměnná
1 – *num_const*	odkaz na pole číselných konstant (viz. dále)
10 000	ADD - sčítání
10 001	SUB - odčítání
10 002	MUL - násobení
10 003	DIV - dělení
10 004	SIN - sinus
10 005	COS - cosinus
10 006	PI - a

Tabulka 3: Kódování hodnot jedince

Konstanty se generují následujícím způsobem. Metoda *gener_const(min,max)* naplní pole *constants* náhodně konstantami ze zadaného rozsahu *min – max*. Toto pole má *num_const* prvků. Jedinec, uložený v poli *buffer*, potom obsahuje hodnotu 1 až *num_const*, která odkazuje na pozici v poli *constants*.

Ukázka1:

Je definováno:

num_const = 5

příkaz *gener_const(1,3)* vytvořil náhodně pole *constants*: 3, 1, 2, 1, 2

Jedinec(10000, 1 , 4) potom odpovídá výrazu:

3 + 1

Došlo se k tomu následovně:

10 000 odpovídá operaci sčítání, 1 ukazuje na první (nultý v notaci C++) prvek pole *constants* tedy konstantu 3 a nakonec 4 ukazuje na čtvrtý prvek pole, tedy 1.

Pro převod hodnot z rozsahu 10 001 – 10 006 na funkce definuji v hlavičkovém souboru **gp.h** také proměnnou výčtového typu nazvanou Funkce, začíná hodnotou FCE_START = ADD = 10 000, končí hodnotou FCE_END = PI = 10 006 a obsahuje všechny funkce, jak to zobrazuje předchozí Tabulka 3:Kódování hodnot jedince. Funkce, které se používají, byly vybrány na základě zkušeností popsaných v publikaci A Field Guide to Genetic Programming ([8] příloha B - Tiny GP) Navíc se vzhledem k účelu použilo funkcí sin, cos a implementace π.

Ukázka2:

Mějme *constants[2]*= 5, *constants[3]*= 20 a jedince:

10 000	3	10 004	10 002	2	10 002	10 006	0

Výraz, který tento jedinec reprezentuje je: 20 + sin(5 * π * t)

Jedinci jsou tvořeni náhodně růstovou metodou s pravděpodobnostmi generování terminálů/funkcí definovanými tak, že každá z možností má definovanou (stejnou nebo rozdílnou) šanci výběru - *P_liter, P_prom*.

3.4.3. Genetické operátory

Genetické operátory, mutace a křížení, se implementují na několika úrovních:

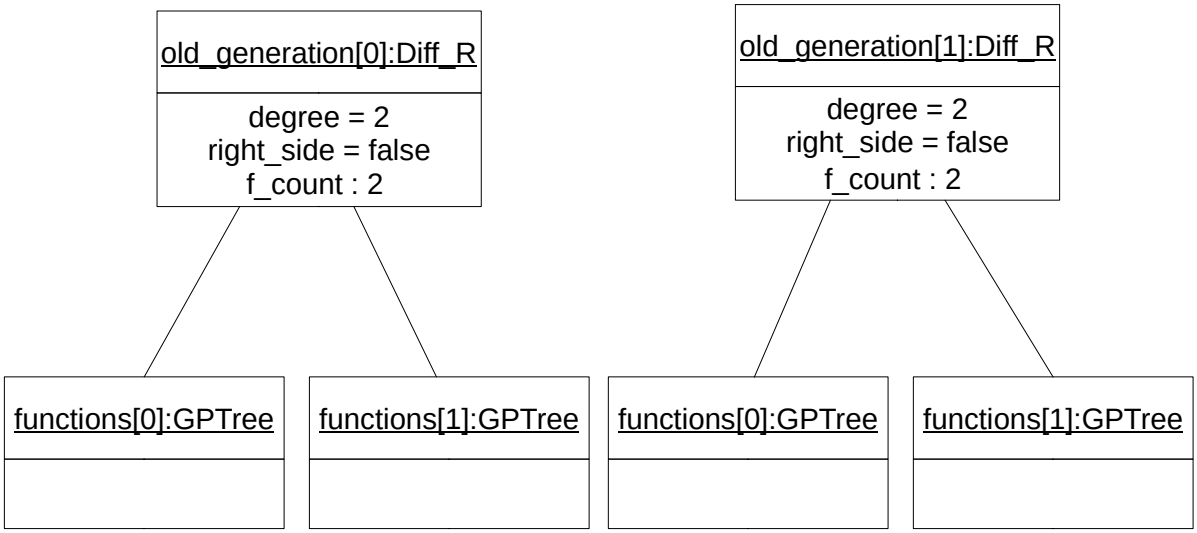
- V hlavním programu se vyhodnotí pravděpodobnost křížení a volá se funkce Crossover z třídy Diff_R. Pravděpodobnost mutace se předá jako parametr metodě Diff_R::Mutation, která se zavolá nad každým jedincem.

- Ve třídě Diff_R se náhodně vygeneruje, ve které z funkcí *functions[i]* (pro připomenutí, každá patří k jedné z derivací $y^{(i)}$, prostému y nebo pravé straně) se vyskytuje bod křížení. Potomek pak obsahuje funkce u nižších derivací od prvního rodiče, potom následuje křížený strom u derivace, ve které se nachází bod křížení a nakonec funkce u vyšších derivací od druhého rodiče. Mutace se vyhodnotí jednotlivě nad každou z funkcí. Obě metody dále volají metody na nižší úrovni, tedy ve třídě GPTree. Pojmenovány jsou podobně: GPTree::crossover a GPTree::mutation.

- Ve třídě GPTree pro provedení křížení a mutace na stromu konkrétní funkce z *functions[i]*. Křížení se musí provést správně na celém podstromu. Mutace se provede výměnou uzlu.

3.4.4. Objektový diagram

Na závěr kapitoly ještě pro přehled uvádím ukázkový objektový diagram Obr. 3.2. Parametry zadání jsou: rovnice 2.řádu bez pravé strany a velikost generace, kterou zde tvoří pole *old_generation[]*, je pro ukázku 2 jedinci.



Obr. 3.2: Objektový diagram

3.5 Implementace Fitness funkce

3.5.1. Třída Fitness

Fitness funkce využívá **metodu nejmenších čtverců** popsanou v Kapitole 2.6 Metody aproximace. Krom toho je také implementována jako alternativa **korelace** signálu, ale jde spíše o pokus a porovnání s metodou první, která se v práci převážně používá. Korelaci provádí tento jednoduchý cyklus:

```
for(i=0;i<pocet;i++) kor += y_given[i]*y_solved[i];
```

Neboli, do proměnné *kor* se provádí sumace součinů *y* zadaného z dat (*y_given[i]*) a *y* vyřešného numericky (*y_solved[i]*). Počet iterací daný proměnnou *pocet* je načten z délky vektoru s daty. Před provedením výpočtu fitness tímto způsobem je ale nutné signál pro účely korelace ještě normalizovat do rozsahu <-1;1>. Třetí a poslední je metoda váženého

součtu z hodnoty nejmenších čtverců a převrácené hodnoty korelace obou předchozích metod.

K výběru metody i ukládání hodnot slouží třída **Fitness** definovaná v souborech **fitness.cpp** a **fitness.h**. Třída je poměrně jednoduchá, pro úplnost i zde uvádím popis pomocí UML 2 v Tabulce 4.

Fitness
+ hodn_nejmensi_ctverce : double + hodn_korelace : double + w1 : static double + w2 : static double + metoda : static TMetoda
+ je_lepsi(&druhy : Fitness) : bool

Tabulka 4: Implementace třídy Fitness dle UML 2

Atributy třídy jsou hodnota fitness pro metodu nejmenších čtverců a korelace, a k nim příslušné váhy *w1* a *w2* pro případ metody váženého součtu a nakonec atribut *metoda*, který je typu TMetoda. Ten je definován na začátku fitness.h jako výčtový typ s hodnotami CTVERCE, KORELACE, VAZSOUCET.

Jediná **metoda**, *je_lepsi(),* slouží k porovnání fitness jedince s fitness druhého jedince, předanou jako argument. V programu se využije na porovnání s aktuální nejlepší hodnotou fitness.

Z hlediska vzájemných relací mezi třídami naznačených v předchozích diagramech (Obr. 3.1 a Obr. 3.2) stojí třída Fitness samostatně, tedy neexistuje přímá vazba ani na jednu z výše zmíněných tříd. Propojení je pouze nepřímé: jedinec, objekt třídy Diff_R, na indexu *n* v poli reprezentujícím generaci, má hodnotu své fitness funkce uloženu na indexu *n* pole obsahujícího objekty třídy Fitness.

3.5.2. Použití třídy Fitness ve funkcích

Použití této třídy je ve funkcích v hlavním programu **diff_rce_gp.cpp**. Jsou to funkce realizující obě metody výpočtu fitness ***least_squared***, ***correlation***, souhrnná funkce **Calc_fitness** a **Calc_all_fitness**.

```
double least_squared(vector<double> &y_given,vector<double> &y_solved)
double correlation(vector<double> &y_given, vector<double> &y_solved)
```

Obě tyto funkce přijímají stejné parametry, kterými jsou vektor *y_given* ze vstupních dat a vektor *y_solved* získaných z vygenerované rovnice. Vrací hodnotu porovnání, v případě nejmenších čtverců jde o druh minimalizace odchylek a v případě korelace je naopak hodnota tím vyšší, čím jsou si signály podobnější.

```
    void Calc_fitness(Diff_R *dr, vector<double> &t_dane,
vector<double> &y_dane, const double y_poc[], Fitness &fitn)
{
...
}
```

Argumentem funkce je jedinec ve formě diferenciální rovnice, která se má vyhodnotit a porovnat. Dále vektory se vstupními daty *t*, *y* a počáteční podmínky. Posledním argumentem je objekt třídy Fitness pro uložení hodnot. V průběhu funkce Calc_fitness se nejprve numericky vyhodnotí diferenciální rovnice (viz. Kapitola 3.6.1 Implementace řešení diferenciální rovnice metodou Runge-Kutta), pak se porovná výsledek se vstupními daty jednou z metod. Při použití korelace se data nejdříve normalizují tak, aby signál byl v rozsahu -1 až +1. Poté se spočítá fitness, uloží se hodnoty, také se ošetří výjimky a nakonec se ještě klasicky dealokuje pomocná pameť.

```
int Calc_all_fitness(Diff_R **gener, vector<double> &t_dane,
vector<double> &y_dane, const double y_poc[], Fitness fitness[], int
n)
{
...
}
```

Navíc je deklarována funkce *Calc_all_fitness()*, která uvnitř volá funkci *Calc_fitness()* na každého jedince v generaci a vrací ukazatel na nejlepšího jedince. Přijímat proto musí logicky stejné argumenty jako funkce *Calc_fitness()*, s rozdílem, že místo jedince potřebuje ukazatel na generaci a stejně tak místo jednoho objektu pole objektů třídy *Fitness*.

3.6 Funkce v hlavním modulu – soubor diff_rce_gp.cpp

Některé důležité funkce z hlavního modulu již byly popsány v předchozích kapitolách. *Calc_fitness, Calc_all_fitness, least_squared, correlation* v Kapitole 3.5.2 Použití třídy Fitness ve funkcích. Některé není pro porozumění nutné podrobněji popisovat – funkce print_param() tiskne do výstupního souboru parametry GP. Ostatní funkce budou popsány v následujících kapitolách. Na úvod ještě výčet zahrnutých knihoven:

```
#include <string>                  - pro práci z řetězci
#include <string.h>

#include <time.h>                  - pro měření doby chodu programu

#include <stdio.h>                 - standardní knihovny pro vstup/výstup
#include <cstdlib>
#include <iostream>

#include <fstream>                 - pro práci s proudy (do souboru a do řetězce)
#include <sstream>

#include <vector>                  - práce s vektory
#include <limits>                  - implementace hodnot typu nekonečno
#include <cmath>                   - matematické funkce
#include <gsl/gsl_matrix.h>        - GSL knihovna
#include <gsl/gsl_odeiv.h>

#include "gp.h"                    - vlastní knihovny
#include "dif_rce.h"
#include "definitions.h"           - obsahuje jen úpravu generátoru náhodných čísel
#include "fitness.h"
```

3.6.1. Implementace řešení diferenciální rovnice metodou Runge-Kutta

Metodu Runge-Kutta, které je použito pro numerické řešení diferenciálních rovnic, implementuji **v hlavním modulu** (diff_rce_gp.cpp) pomocí numerické knihovny GSL (GNU

Scientific Library [7]), která je volně dostupná na internetu a obsahuje více než 1000 matematických a statistických funkcí. Užití knihovních funkcí pro řešení rovnic vyžaduje definovat specifické proměnné a volat funkce předepsaným způsobem. Nejprve se definuje proměnná pro metodu:

```
const gsl_odeiv_step_type *T = gsl_odeiv_step_rk4;
```

Tento řádek určuje typ krokovací funkce, která bude použita, *gsl_odeiv_step_rk4* je označení pro metodu Runge-Kutta 4. řádu. Následně je nutné alokovat paměť pro novou instanci krokovací funkce:

```
s = gsl_odeiv_step_alloc (T, rad);
```

Kde *rad* je řád rovnice (*T* již bylo uvedeno výše). Poté se vytvoří systém rovnic:

```
gsl_odeiv_system sys = {func, NULL, (unsigned int)rad, dr};
```

Má tyto parametry:

- *func* předává diferenciální rovnice ve formě struktury, která uchovává vektor hodnot *y* a *t* a některé další parametry.
- Následujícím parametrem je jakobián, který ale není potřebný pro všechny typy krokovacích funkcí, proto zde má hodnotu *NULL*.
- *rad* je dimenze soustavy rovnic
- *dr* je odkaz na objekt třídy, která je použita pro diferenciální rovnici.

Poté se do vektoru *y* vloží čas 0 a řešení v čase 0.

```
for(int i=0;i<rad;i++) y[i] = y_poc[i];
t_res.push_back(0);
y_res.push_back(y[0]);
```

Následujícími řádky už se spouští samotné vyhodnocení diferenciální rovnice a jednotlivých derivací. Proměnná *status* nese informaci o úspěchu a v opačném případě se vyhodí (*throw*) výjimka, jinak pokračuji zvýšením hodnoty času *t* o krok *h* až do času t1, který je načten z

velikosti pole dostupných hodnot, a vložením dalšího bodu řešení. Nakonec se nesmí zapomenout dealokovat veškerou dynamicky přidělenou paměť pomocí *free* a *delete*.

```
GSL_ODEIV_FN_EVAL(&sys, t, y, dydt_in);
    while (t < t1)
    {
        status = gsl_odeiv_step_apply (s, t, h, y, y_err, dydt_in,
dydt_out,&sys);

       for(int i=0;i<rad;i++) dydt_in[i] = dydt_out[i];
       t += h;
       if (status != GSL_SUCCESS) throw ERR_evolve();

       t_res.push_back(t);
       y_res.push_back(y[0]);

     }

  gsl_odeiv_step_free (s);
  delete [] y;
  delete [] y_err;
  delete [] dydt_in;
  delete [] dydt_out;
```

3.6.2. Funkce pro turnajovou selekci

Důležitou funkcí v hlavním modulu je také implementace turnajové selekce.

```
int Tournament(Diff_R **gener, Fitness fitness[], int popul_size,
int n, float p)
```

Vstupními hodnotami jsou:

- *gener* - pole ukazatelů na jedince (generace)
- fitness - pole s hodnocením jedinců
- popul_size - velikost generace
- n – počet účastníků turnaje
- p – pravděpodobnost označení lepšího jedince jako vítěze (použito 100%)

Funkce uspořádá turnaj mezi *n* náhodně vybranými jedinci a s pravděpodobností *p* vybere nejlepšího, jinak dalšího v pořadí, a vrátí index vybraného jedince. V tomto provedení pracuji s turnajem o dvou účastnících, ale funkce je napsána tak aby dokázala pracovat i s většími turnaji. Jedinci jsou do turnaje vybráni náhodně:

```
index1 = random(popul_size);
index2 = random(popul_size);
```

Následně použiji metodu *Fitness::je_lepsi*, popsanou v kapitole 3.5.1, na porovnání 2 jedinců a prohodím jedince tak, aby v *index1* byl nejlepší. V případě většího turnaje, provádím tak dlouho, dokud nebyli otestováni všichni a na první pozici je opravdu nejlepší jedinec. Nakonec vyhodnotím pravděpodobnost p, že zvítězí nejlepší.

```
if (random(100)/100.0 < p) return index1; else return index2;
```

3.6.3. Funkce main

Chod hlavního programu provádí funkce main. Její průběh se řídí algoritmem popisujícím obecně všechny EVT, tak jak byl znázorněn v Kapitole 2.2 Evoluční výpočetní techniky (EVT) na diagramu Obr. 2.1.

V průběhu využíváme různé **datové struktury**, nejdůležitější jsou:

- **Dynamická pole** pro uchování staré a nové generace jako matice jedinců a nejlepšího jedince.

```
Diff_R **old_generation;
Diff_R **new_generation;
Diff_R  *best_solution;
```

- **Implicitní hodnoty** parametrů GP

```
int popul_size = 100;     - velikost populace
int iter_num = 100;       - počet iterací
int rad=2;                - řád diferenciální rovnice
bool prava_strana=false; - generovat i pravou stranu rovnice
float Pc=0.8;             - pravděpodobnost křížení
float Pm=0.2;             - pravděpodobnost mutace
```

- Vektory **vstupních dat** a počáteční podmínky

```
vector<double> y_dane;    - vektor daných y
vector<double> t_dane;    - vektor času
double *y_poc;
```

- Proměnné potřebné pro **měření času** chodu programu

```
time_t start_time;  - počáteční čas běhu GA
time_t stop_time;   - konečný čas běhu GA
time_t run_time;    - rozdíl konečného a počátečního času
```

Nyní už se můžeme zabývat samotným chodem programu. Nejprve se **otestují vstupní parametry**, jsou-li chybně zadány, je na to uživatel upozorněn a popř. se použijí implicitní hodnoty parametrů. Zadají se některé parametry, které jsou uživateli skryty.

```
GPTree::num_const = 500;          - počet generovaných konstant
GPTree::gener_const(-20,20);      - rozmezí hodnot použitých konstant
GPTree::P_liter = 0.3;            - pravděpodobnost generování literálu
                                    (proměnná/konstanta) vůči funkci
```

Poté se **načte obsah souboru** se vstupními daty. **Dynamicky se alokuje paměť** pro generaci, mezigeneraci a fitness, pro níž se také vybere metoda.

```
old_generation = new Diff_R*[popul_size];
new_generation = new Diff_R*[popul_size];
fitness = new Fitness[popul_size];

for(i=0;i<popul_size;i++)
{
  old_generation[i] = new Diff_R(rad,prava_strana);
  new_generation[i] = new Diff_R(rad,prava_strana);
}
best_solution = new Diff_R(rad,prava_strana);
```

Nyní už začíná samotný průběh GA. V tuto chvíli se také spouští měření času. Nejprve se vytvoří náhodně **počáteční generace** s omezením maximální hloubky stromu, ohodnotí se všichni jedinci a výbere se aktuálně nejlepší jedinec.

```
  for(i=0;i<popul_size;i++)
  {
    old_generation[i] -> generate_random(depth);
  }
```

Následuje hlavní cyklus, který realizuje vznik nových generací z předchozích s uchováváním nejlepšího výsledku. Nejprve se aplikuje **křížení**.

```
while(iter<=iter_num)
  {

for(i=2;i<popul_size;i=i+2)
     {
                i1 = random(popul_size); i2 = random(popul_size);
          if(random(100)/100.0<Pc)
                {
        Crossover(old_generation[i1],old_generation[i2],new_generatio
n[i],new_generation[i+1]);
                }
          else
          {
            *new_generation[i]=*old_generation[i1];
            *new_generation[i+1]=*old_generation[i2];
          }
     }
```

Jedinci jsou vybíráni náhodně po dvou a buďto se stanou rodiči dvou potomků křížení nebo sami postoupí do nové generace na základě pravděpodobnosti *Pc*. Poté se provede **mutace**.

```
for(i=2;i<popul_size;i++)
  {
      new_generation[i]->Mutation(Pm);
  }
```

Vypočítají se fitness koeficienty celé generace a na základě těchto hodnot proběhne mezi jedinci **turnajová selekce**. Tím vznikne nová generace, která se stane základem pro další iteraci algoritmu.

```
best_index                                                    =
Calc_all_fitness(new_generation,t_dane,y_dane,y_poc,fitness,popul_si
ze);

for(i=2;i<popul_size;i++)
  *old_generation[i]=*new_generation[Tournament(new_generation,
     fitness,popul_size,2,1.0)];
```

Tím končí jedna iterace a postupuje se k další s tím, že nejlepšího jedince uložím na první dvě místa nové generace, abych ho uchoval pro případ, že nebyl vybrán do turnaje (tzv. technika *eliticismu*).

```
      *old_generation[0]=*new_generation[best_index];
      *old_generation[1]=*new_generation[best_index];
```

Během každé iterace vypisuji na obrazovku aktuální statistiky a na závěr nejlepší výsledek a čas běhu programu. Ukázka výstupu je v kapitole 4.3.

Na závěr je důležité nezapomenout dealokovat dynamicky přidělenou paměť a zavřít otevřené soubory.

- dealokace

```
for(i=0;i<popul_size;i++) delete old_generation[i];
 for(i=0;i<popul_size;i++) delete new_generation[i];
 delete [] old_generation;
 delete [] new_generation;
 delete [] fitness;
 delete best_solution;
 delete [] y_poc;
 GPTree::free_const();
```

- zavření výstupního souboru se statistikou

```
if (file_stat != NULL) fclose(file_stat);
```

4 Používání programu

4.1 Spouštění souboru diff_rce_gp.exe

Program se spouští z příkazové řádky následovně:

diff_rce_gp.exe vst_soubor rad hloubka prava_str Pc Pm N I metoda [-l log_soubor],

kde:

vst_soubor	... jméno vstupního souboru s daty
rad	... řád diferenciální rovnice
hloubka	... max. hloubka stromu
prava_str	... generovat pravou stranu rovnice ano/ne
Pc	... pravděpodobnost křížení 0.0 - 1.0
Pm	... pravděpodobnost mutace 0.0 - 1.0
N	... velikost jedné generace (sudé číslo)[5]
I	... počet iterací (generaci)
metoda	... metoda ve fitness funkci [CTVERCE, KORELACE, VAZSOUCET]
log_soubor	... jméno výstupního souboru pro statistiku

V případě, že některý parametr zadán není nebo je zadán chybně, má program nastaveny implicitní hodnoty těchto parametrů, které použije, a upozorní na to uživatele. Vstupní soubor s daty je nezbytný. Soubor pro výstupní statistiku je volitelný parametr.

Vstupní soubory jsou textové, první řádek obsahuje počáteční podmínky, každý další řádek vždy dvojici čas a příslušná hodnota *y* oddělené tabulátorem. Tyto soubory generujeme pomocným programem **diff_rce.cpp** (viz. Příloha 11.1). Textové soubory s vygenerovanými daty rovnic použitých při experimentech lze nalézt ve složce /inputs.

[5] Během křížení se plní generace po dvou jedincích

4.2 Spouštění programu pomocí dávkového souboru gp.bat

Pro usnadnění práce s programem byl vytvořen dávkový soubor **gp.bat**, který zadává parametry GP. Důvodem je jednak to, že často s programem pracuji tak, že jej opakovaně spouštím se stejnými parametry pro porovnání výsledků a jednak také fakt, že po několika spuštěních se některé parametry GP ukáží jako optimální, a proto je chci ponechat jako výchozí pro další pokusy i s jinými vstupními daty a rovnicemi. Vyhnu se tak opakovanému ručnímu zadávání stejných hodnot. Jediný parametr, který soubor gp.bat nezadává je vstupní soubor s daty, je proto nutné jej zadat při volání gp.bat **z příkazové řádky** jako parametr dávky. Otevření souboru gp.bat v textovém editoru umožňuje snadnou úpravu jen některých parametrů se zachováním stávajících hodnot ostatních. Dávka je napsána tak, že pokud není v textu změněn soubor pro výstup, aktuální výstup přepisuje obsah souboru log.txt.

Ukázka volání: *C:\gp.bat cost.txt*
(soubory gp.bat a cost.txt musí být v této ukázce v adresáři C:\ )

Kromě změn v hlavních parametrech je možné při testování vyzkoušet měnit i parametr pravděpodobnost generování literálu/funkce. Vzhledem k růstové metodě tvorby stromů má tento parametr velký vliv na počáteční generaci. Úprava hodnoty se provádí změnou ve zdrojovém kódu diff_rce_gp.cpp na řádku 464:

```
GPTree::P_liter = 0.2;
```

4.3 Ukázkový výpis výstupu

Po spuštění program vypisuje pro každou generaci hodnotu fitness aktuálně nejlepšího jedince. Jakmile je vytvořena poslední generace, vypíše se opět nejlepší fitness a příslušný jedinec ve formě rovnice.

Ukázka výstupu:

Příkaz: C:\diff_rce_gp.exe cost.txt 2 4 ne 0.8 0.02 10000 10 CTVERCE
Výstup na obrazovku:

```
Parametry algoritmu:
--------------------
Rad rovnice:                    2
Hloubka stromu:                 4
Generovat pravou stranu:        ne
Pravděpodobnost krizeni:        Pc=0.800
Pravdepodobnost mutace:         Pm=0.020
Velikost generace:              N=10000 jedincu
Pocet iteraci:                  I=10
************************

**********************

Gener.  Ctverce Korelace
0       0.0000  18.8589  index = 8527
1       0.0000  18.8589  index = 1914
2       0.0000  18.8589  index = 0
.
.
.
10    0.0000  18.8589  index = 0
Cas behu: 118 sec.,  (0:01:58)

Nejlepsi reseni:
(PI-PI)*y(1)+cos(t)*y(0)

Fitness: ctverce = 0.000000, korelace = 18.858858
```

4.4 Vysvětlivky k výstupu

- Během výpisu si lze povšimnout, že se index po chvíli změní na hodnotu 0. Je to z toho důvodu, jak bylo již zmíněno v popisu funkce main[6], že nejlepšího jedince do nové generace předám vždy automaticky a uložím ho právě na první místo v poli.

- Zvláštní se může zdát v zápisu řešení na uvedeném příkladu funkce u první derivace, tedy *(PI-PI)*, která je už logicky nulová. Po konzultaci s vedoucím práce jsem se ale rozhodl uvažovat toto řešení jako přípustné.

- V rozboru řešení v kapitole 5.2 si lze všimnout dalšího zvláštního jevu. Hodnota fitness správného řešení metodou nejmenších čtverců není 0 ale 944602925995.2366. Jedná se pravděpodobně o chybu, kterou se nepodařilo zatím odhalit. Nicméně program i s touto hodnotou pracuje správně a výsledek to neovlivní.

[6] Kapitola 3.6.3 Funkce main

5 Experimenty a porovnání výsledků s GE

5.1 Použité testovací rovnice

Zhodnocení efektivity a použitelnosti naprogramovaného algoritmu bude provedeno na vybraných testovacích rovnicích. Vzhledem k tomu, že z předchozí bakalářské práce [6] mám k dispozici výsledky řešení aproximace signálu pomocí metody gramatické evoluce (GE), využiji stejné testovací rovnice. To mi umožní porovnat funkčnost obou metod, čas, za který se dojde k výsledku a při jakých parametrech GP to bude. Zejména pak půjde o velikost generace, počet iterací, popřípadě také o parametry rekombinací – pravděpodobnost mutace a křížení. Pro složitější rovnice, kde se přesná shoda s hledanou rovnicí nenajde, bude možno porovnat, jaký typ rovnic, každá z metod upřednostňuje jako nejbližší hledané rovnici. Z metod výpočtu fitness funkce používám v experimentech převážně metodu nejmenších čtverců a hodnotu korelace jen pro kontrolu. Během průběhu testování byly změněny i některé uživateli skryté parametry, jako je např. rozsah a počet konstant nebo pravděpodobnost generováni literálu vůči funkci a tím byly provedeny různé zásahy do počáteční generace. Vše je vždy popsáno v rozboru řešení u dané úlohy.

Testovací rovnice jsou roztříděny podle složitosti do tří kategorií. První kategorie jsou rovnice nejjednodušší. Jde o funkce $y''(t)= 0*y(t)$, $y''(t)= 2*y(t)$ a $y''(t)= t*y(t)$. Druhá kategorie jsou funkce $y''(t)= sin(t)*y(t)$ a $y''(t)= cos(t)*y(t)$. Vzhledem k tomu, že jednou ze zamýšlených aplikací programu je aproximace signálu EEG, směřujeme testování k rovnici Mathieu, kdy se průběh jejího řešení podobá signálům typu EEG. Obecný analytický zápis této rovnice je $y''(t)= -(a + 16q\ cos(2t))\ y(t)$, kde a a q jsou koeficienty z oboru reálných čísel. Tomu odpovídá poslední, třetí, kategorie. Analytický zápis zde použité rovnice typu Mathieu je $y''(t)= -(16+16*cos(2t))*y(t)$.

5.2 První sada testovacích rovnic

V prvním experimentu se hledají tři jednoduché funkce.

Analytický zápis hledaných rovnic:

$$y''(t) = 0 * y(t)$$

$$y''(t) = 2 * y(t)$$

$$y''(t) = t * y(t)$$

Měřený časový interval a krok: interval < 0,0 ; 20,0 > s krokem 0,1

Testováno s parametry GP:

Rad rovnice:	2		
Hloubka stromu:	4		
Generovat pravou stranu:	ne		
Pravděpodobnost krizeni:	Pc	=	0.400 - 0.800
Pravdepodobnost mutace:	Pm	=	0.200 - 0.800
Velikost generace:	N	=	100 až 10000 jedincu
Pocet iteraci:	I	=	10 až 100

****

a skryté parametry:

```
GPTree::num_const = 100;
GPTree::gener_const(0,5);
GPTree::P_liter = 0.2;
```

Spuštěno: několik pokusů pro každou rovnici s různými parametry

Některá nalezená řešení:

- $y^{(2)}t) = 0*y(t)$
    - (0.000000*4.300000)*y(1) + (cos((PI-(t+4.450000)))*0.000000)*y(0)
    - sin(PI)*y(1) + (((cos(t)*t)/t)*(sin(PI)*4.500000))*y(0)
    - 0.000000*y(1) + (((0.000000*(t+3.050000))-PI)*((PI+t)*sin((t-t))))*y(0)
    - sin(sin(sin((t-t))))*y(1) + (t-t)*y(0)
- $y^{(2)}t) = 2*y(t)$
    - 0.000000*y(1) + 2.000000*y(0)
    - sin(0.000000)*y(1) + 2.000000*y(0)
    - (1.150000-1.150000)*y(1) + 2.000000*y(0)
- $y^{(2)}t) = t*y(t)$
    - (sin(PI)*(4.900000/(t-(t-1.550000))))*y(1) + t*y(0)
    - (sin(PI)/(PI/(t*(t-3.700000))))*y(1) + t*y(0)
    - (sin(((t-t)/(3.850000-t)))/PI)*y(1) + t*y(0)

Nejlepší nalezená řešení: Pro každou rovnici bylo nalezeno správné řešení.

Rozbor řešení:

Nejlepší řešení bylo nalezeno při generaci 1000 jedinců, obou pravděpodobnostech 0,8 a pouhých 10 iteracích. Navíc je třeba dodat, že to nebylo z důvodu, že by bylo vygenerováno náhodně už do první generace, ale právě naopak vzniklo postupným vylepšováním.

Pro ilustraci, vzhledem k tomu, že není velmi dlouhý, uvádím celý výpis průběhu:

Gener.	Ctverce	Korelace	
0	259484714761213280000000.0000	4.0512	index = 806
1	8912771057702333800000.0000	4.0479	index = 232
2	8912771057702333800000.0000	4.0479	index = 0
3	8912771057702333800000.0000	4.0479	index = 0
4	8912771057702333800000.0000	4.0479	index = 0
5	88259011693691137000000.0000	4.0349	index = 998
6	66469432123702536000000.0000	4.0165	index = 291
7	2451288323767946800000.0000	4.0650	index = 24
8	2451288323767946800000.0000	4.0650	index = 0
9	944602925995.2366	4.0591	index = 51
10	944602925995.2366	4.0591	index = 0

Cas behu: 10 sec., (0:00:10)

Nejlepsi reseni:
0.000000*y(1)+2.000000*y(0)

V tomto průběhu se našlo řešení i **v přesném znění**, ale jak bylo vidět v ukázce ostatních nalezených řešení, většinou se vyskytuje ve tvaru komplikovanějším, který teprve úpravou vede na přesné znění, ale i tato řešení považuji za správná a z důvodu úspory výkonu neimplemetuji žádný mechanismus, který by nad každým jedincem prováděl zjednodušení jednotlivých výrazů. **Doba běhu programu**[7] se při testování pohybovala v rozmezí od několika sekund při generaci 1000 jedinců a 10 iteracích, kolem 4 minut při velikosti generace 10000 jedinců a stejném počtu iterací a více než 10 minut při kombinaci 1000 jedinců a 100 iterací. V případě nízkého počtu iterací a malé velikosti generace z pochopitelných důvodů program výsledek najít nestihne. K tomu může dojít i v případě, že se počáteční náhodná generace velmi nepovede, nebo jsou-li výsledky rekombinací příliš zavádějící. Zajímavé je proto i porovnání běhu při velké generaci a malém počtu iterací s během při malé generaci a velkém počtu iterací. Druhá varianta je velmi závislá na kvalitě úvodní generace. Optimálních výsledků jsem při pokusech dosahoval při vyšších pravděpodobnostech obou rekombinací.

[7] testováno na notebooku s procesorem Intel(R) Core(TM)2 Duo CPU T7500 2,2 GHz, 2 GB RAM

5.3 Druhá sada testovacích rovnic

Druhý experiment přidává goniometrické funkce sinus a kosinus.

Analytický zápis hledaných rovnic: *y''(t)= sin(t)*y(t)*
*y''(t)= cos(t)*y(t)*

Měřený časový interval a krok: interval < 0,0 ; 20,0 > s krokem 0,1

Testováno s parametry GP:

Rad rovnice:	2		
Hloubka stromu:	4		
Generovat pravou stranu:	ne		
Pravděpodobnost krizeni:	Pc	=	0.200 až 0.800
Pravdepodobnost mutace:	Pm	=	0.800
Velikost generace:	N	=	100 až 10000 jedincu
Pocet iteraci:	I	=	10, 40, 100, 200, 500

****

a skryté parametry:

```
GPTree::num_const = 100;
GPTree::gener_const(0,5);
GPTree::P_liter = 0.2;
```

Spuštěno: přibližně 20 pokusů, se stejnými parametry spuštěno vždy alespoň 3 krát

Některá nalezená řešení:

- *$y^{(2)}t) = sin(t)*y(t)$*
  - sin(PI)*y(1)+sin(t)*y(0)
  - (t-t)*y(1)+sin(t)*y(0)

- *$y^{(2)}t) = cos(t)*y(t)$*
  - (cos(((t-1.750000)*2.100000))*sin(PI))*y(1) + cos(t)*y(0)
  - sin(PI)*y(1) + cos(t)*y(0)
  - 0.000000*y(1) + cos(t)*y(0)

Nejlepší nalezené řešení: Správné řešení nalezeno

Rozbor řešení:

Tento typ rovnic se velice často vyskytuje už v počáteční generaci a řešení je proto nalezeno okamžitě. Kvůli tomu jsem se rozhodl experimentovat s malými generacemi, kde je menší pravděpodobnost výskytu řešení už v počátku, a s větším počtem iterací. Z toho logicky vyplývá menší úspěšnost každého z pokusů a vzhledem k malé generaci také velmi pomalé zlepšování a potřeba velkého počtu iterací s nejistým výsledkem. Touto cestou se řešení podařilo najít, dle předpokladu, opravdu jen v malém procentu experimentů.

5.4 Třetí sada testovacích rovnic

V posledním experimentu se zaměřuji na funkci typu Mathieu.

Analytický zápis hledané rovnice: *y''(t)= - (16+16*cos(2t))*y(t)*

Měřený časový interval a krok: testováno s daty pro intervaly < 0,0 ; 10,0 >, < 0,0 ; 20,0 > a < 0,0 ; 40,0 > s krokem 0,1

Testováno s parametry GP:

Rad rovnice:	2		
Hloubka stromu:	6		
Generovat pravou stranu:	ne		
Pravděpodobnost krizeni:	Pc	=	0.200 až 0.800
Pravdepodobnost mutace:	Pm	=	0.200 až 0.800
Velikost generace:	N	=	1000 až 20000 jedincu
Pocet iteraci:	I	=	10 až 200

****

a skryté parametry:

```
GPTree::num_const = 200;
GPTree::gener_const(-20,20);
```

```
GPTree::P_liter = 0.2 až 0.6;
```

Spuštěno: ve velkém množství kombinací parametrů a úprav kódu více než 20 pokusů

Nejlepší nalezená řešení (zápis je oproti přímému výstupu zkrácen):

*(0) * y(1) + ((-13+(t/4)*sin(-4t)) * y(0)*
(-1) y(1) + (-19+sin(-5)+2t) * y(0)*
*(0) * y(1) + ((-16+(2t)*sin(3t-4)) * y(0)*
*(0) * y(1) + ((-16 + (-16)*cos(2t))* y(0)*

Rozbor řešení:

V případě této úlohy bylo v ojedinělém případě nalezeno přesné řešení, což ale přisuzuji do jisté míry spíše náhodě, která samozřejmě v EVT hraje velkou roli. Ve většině ostatních pokusů vznikaly spíše rovnice, které se tvarem hledanému řešení podobají. Lze předpokládat, že po několika správně zvolených rekombinacích by vedly k přesnému řešení. Volba rekombinací je ale řízena náhodným jevem a proto se často stane, že se v daném počtu iterací k řešení nedojde. Zvyšování iterací vede k přesnějšímu výsledku, ale v dlouhodobějším horizontu vede na stromy s příliš velkou hloubkou, kde jsou ale většinou konstanty nahrazeny rozsáhlými podstromy. Doporučuji tedy v těchto případech snížit hodnotu pravděpodobnosti křížení a spoléhat více na mutace při větší počáteční generaci. V tomto experimentu byla vyzkoušena i změna ve zdrojovém kódu - datový typ z double na int, kvůli velkému rozsahu konstant a protože celá čísla nám na přesné znění testované rovnice stačí. Nakonec se ale ukázalo, že pro reálná čísla jsou nalezeny lepší aproximace. Pro podrobnější pohled na to, jakým způsobem a jestli dané výsledky konvergují s hledaným řešením odkazuji na následující kapitolu 5.5 Rozbor situací, kdy nebylo nalezeno správné řešení.

5.5 Rozbor situací, kdy nebylo nalezeno správné řešení

K rozboru jsem vybral několik situací z řešení rovnice typu Mathieu, kdy řešení neodpovídalo přesně zadání ale bylo vzorcem podobné a mělo dobré fitness ohodnocení. Rozbor provedu tak, že nalezenou rovnici numericky vyhodnotím pomocí programu diff_rce.exe (Kapitola

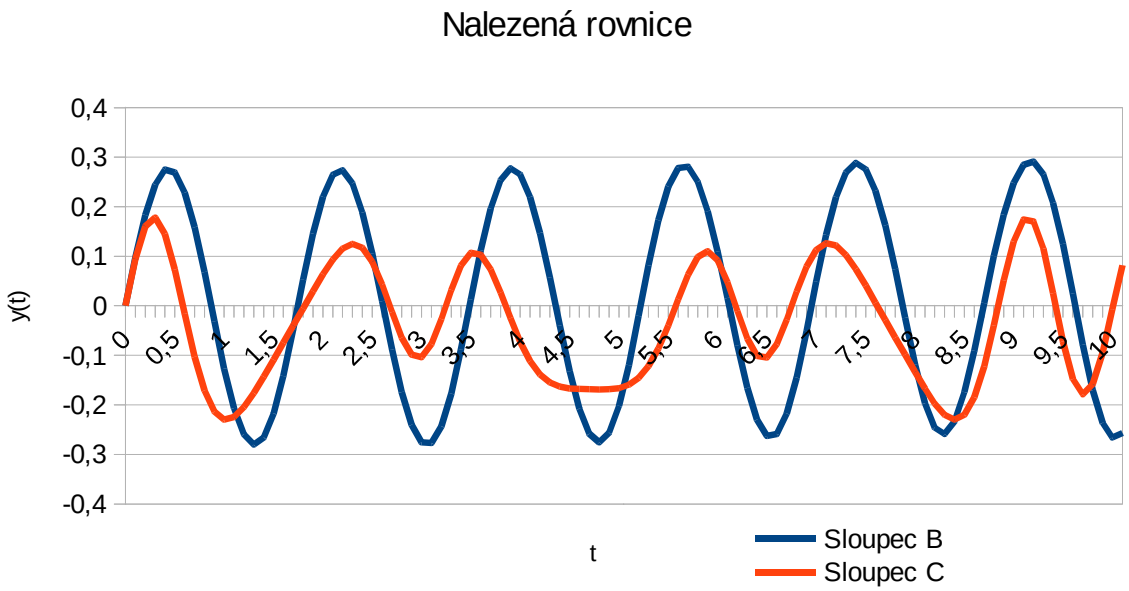
11.1). Výstupní data použiji k sestavení grafu v němž zobrazím jak nalezenou tak hledanou rovnici.

Hledaná rovnice:

*(0) * y(1) + ((-16 + (-16)*cos(2t))* y(0)*

Nalezená rovnice:

*(0) * y(1) + ((-13+(t/4)*sin(-4t)) * y(0)*



Obr. 5.1: Graf 1

Sloupec B – modře - nalezené řešení

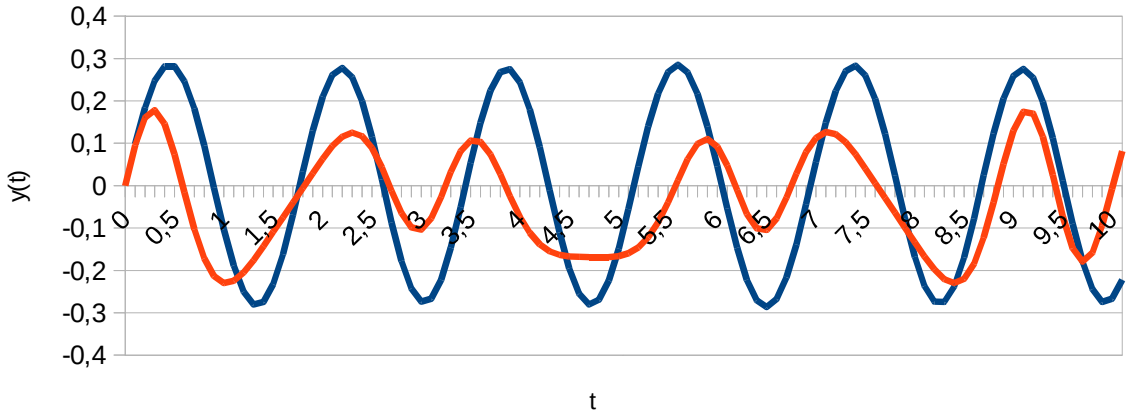
Sloupec C – červeně - hledané řešení

Hledaná rovnice:

*(0) * y(1) + ((-16 + (-16)*cos(2t))* y(0)*

Nalezená rovnice:

*(0) * y(1) + ((-8.800000)+((-4.400000)+cos(t))) * y(0)*



Obr. 5.2: Graf 2

V obou případech je perioda průběhu velmi blízká až stejná a amplituda je také podobná, je proto pochopitelné, že jsou tyto funkce ohodnoceny dobrou hodnotou fitness funkce. Vyskytují se ale i případy, kdy podobné ohodnocení dostane i funkce, jejíž průběh je hodně odlišný.

6 Porovnání s výsledky GE

6.1 Krátké shrnutí výsledků GE pro účely porovnání

Nejprve jen krátce shrnu výsledky pokusů metody GE, které jsou uvedené v celém znění v bakalářské práci [6].

Úloha 1 a rovnice *y''(t)= 0*y(t), y''(t)= 2*y(t) a y''(t)= t*y(t)* byly vzhledem k povaze a provedené implementaci nalezeny vždy již v počáteční generaci.

Úloha 2 s rovnicemi *y''(t)= sin(t)*y(t)* a *y''(t)= cos(t)*y(t)*. Řešení bylo v různých obměnách nalezeno při generaci o velikosti 1000 a výše s *Pc = 0.6 - 0.8* a *Pm = 0.2 - 0.6*. Často bylo řešení opět nalezeno už v počáteční generaci, ale pokud ne, nebyla úspěšnost pokusu velmi vysoká.

Úloha 3 a rovnice Mathieu *y''(t)= - (16+16*cos(2t))*y(t)*. V tomto případě GE nenalezla správné řešení, pouze jeho aproximaci *-(sin(46))*y'(t)-(15)*y(t)*, která vykazovala některé znaky podobnosti s originálem, které jí zaručili dobré fitness ohodnocení o použitelný výsledek ale nešlo.

6.2 Porovnání výsledků obou metod

Pro nejjednodušší rovnice typu experimentu 1 jsou obě metody srovnatelné, protože řešení je buďto již v počáteční generaci nebo vznikne hned při provedení jedné mutace. Pro rovnice typu druhého experimentu se využívá v jednom případě více křížení v druhém mutace, ve výsledku je ale výstup stejný s tím, že GP se podařilo v některých experimentech dostat k řešení rekombinacemi i při menší generaci. V případě hledání rovnice typu Mathieu se už obě metody odlišují. GP se v případě „neomezeného“ času více blíží ke správnému řešení a i po větším počtu vzniklých generací se stále objevují zlepšení, bohužel je to vykoupeno nárůstem hloubky stromů a komplikováním jednotlivých funkcí složitými výrazy, které značně ztěžují orientaci ve výsledku, přičemž po zkrácení odpovídají konstantě. Přesto považuji výsledky GP za použitelnější než u GE s odkazem na úpravy navržené v kapitole 7.3 Zhodnocení a návrhy pro zlepšení.

7 Závěr

7.1 Shrnutí provedených experimentů

Provedl jsem experimenty při nichž jsem hledal některé vybrané rovnice. Pro jednodušší stromy výrazů bylo řešení nalezeno snadno. Ať už to bylo v počáteční generaci tak i během iterací genetického programování a aplikováním rekombinací. V tomto typu úloh byla efektivnějším rekombinačním operátorem mutace, protože šlo především o změny konstant. U úloh typu $y^{(2)}t) = sin(t)*y(t)$ je mutace nastavena tak, že *sin* zamění za *cos* a obráceně, proto je potřeba zahrnout ve větší míře i křížení.

Pro nejkomplikovanější úlohu, rovnici typu Mathieu, jsou nezbytné oba operátory a navíc větší generace jedinců s poměrně velkým počtem iterací. Zvýšení počtu iterací obvykle přinese lepší výsledek a s časem se tedy správnému řešení více a více přibližuje. Rychlost tohoto přibližování sice kolísá, ale lze předpokládat, že v lokálním extrému neuvázne z toho důvodu, že použité rekombinace jsou aplikovány s dostatečnou pravděpodobností a přináší dostatečně velkou změnu.

7.2 Srovnání GP a GE

Provedu-li srovnání obou metod EVT, je třeba brát v úvahu především výsledky složitějších experimentů. Z hlediska porovnávaných implementací vychází GP jako metoda s větším potenciálem k úspěšnému hledání výsledku řešení aproximace signálu vyjádřeného diferenciální rovnicí než metoda GE.

7.3 Zhodnocení a návrhy pro zlepšení

Naprogramovaná implementace GP není zřejmě zcela ideální, ale s hledaným řešením, i když v dlouhém čase výpočtu, většinou konverguje. Pro praktické využití by bylo nutné mít co nejvíce informací o hledané rovnici a program tomu na míru upravit, nejen co se týká nastavitelných parametrů programu jako je např. hloubka stromu, ale i podstatně znatelnější úpravy jako např. cílený zásah do úvodní generace s tvorbou jen určitého typu stromů, jejichž kombinováním budou vznikat rovnice očekávaného tvaru.

Po provedení velkého množství experimentů s různými parametry GP, různými parametry programu obecně a s jeho drobnými úpravami jsem nalezl některá možná vylepšení a změny, které by mohly pomoci k rychlejšímu a efektivnějšímu hledání výsledků. Tato vylepšení metody spatřuji v:

- **úpravě metod rekombinace**

  Je zřejmé, že k vylepšování nejlepšího výsledku generace dochází jen velmi pozvolna. Na základě sledování vlivu rekombinací na populaci by mohlo být užitečné vyzkoušet metody, kterými se obě operace provádějí, pozměnit, popř. přidat dle uvážení některé jiné genetické operátory.

- **fixaci hodnot spočítatelných výrazů**

  Při prozkoumání nalezených řešení si často můžeme v jednotlivých rovnicích všimnout výrazů typu „ *(5-5)*(… “* , které lze snadno spočítat. V práci sice tato řešení, pokud vedou na správný výsledek, akceptujeme, ale je možné, že mají negativní vliv na generované potomky. Mohlo by tedy být výhodné nahradit tyto zbytečné podstromy jedním uzlem se snadno spočítatelnou hodnotou. Nevýhodou tohoto je, že se prodlužuje doba výpočtu o dobu práce potřebnou na tuto úpravu s každým jedincem a v každé generaci.

- **větší kontrole hloubky stromů vznikajících během chodu programu**

  Během chodu programu hloubka stromů stále narůstá a aby nedocházelo k jejímu nekontrolovatelnému nárůstu je nutné stanovit limit maximální povolené hloubky. I s ním ale vznikají během křížení stále jedinci s hloubkou spíše u horní hranice limitu a jednodušší výrazy, jako je například funkce $f_{(i)}(t) = 0$ u členů derivací, které nejsou v rovnici zastoupeny, se objeví jen zřídka. Místo toho tomto místě vzniknou stromy, které v ideálním případě po vyhodnocení hodnotu *0* mít budou, ale i tak zavádějí hledání řešení jiným směrem.

- **větší kontrole počáteční generace**

  Vliv počáteční generace na výsledek už byl v práci často zmiňován. Předpokládalo se ale, že růstovou metodou náhodně vytvořené stromy představují dostatečný základ. Mohlo by ale stát za úvahu nenechat celou generaci náhodě a uměle vložit dostatečně rozmanité jedince, kteří by poskytli dobré východisko pro další generace.

- **návrhu jiné metody vyhodnocení fitness**

  Jak bylo naznačeno v kapitole 5.5, někdy jsou i poměrně zcestná řešení ohodnocena dobrou fitness, proto by návrh jiné metody výpočtu mohl přispět k lepším výsledkům.

Na úplný závěr bych chtěl jenom shrnout veškeré poznatky získané experimenty, které naznačují, že má smysl řešit úlohy aproximace signálu touto metodou GP s přihlédnutím k navrženým úpravám.

8 Použité zdroje

[1] MAŘÍK, V., O. ŠTĚPÁNKOVÁ, J. LAŽANSKÝ a kol.: *Umělá inteligence (3)*. Praha, Academia, 2001. ISBN 80-200-0472-6.

[2] MAŘÍK, V., O. ŠTĚPÁNKOVÁ a J. LAŽANSKÝ a kol.: *Umělá inteligence (4)*. Praha, Academia, 2003. ISBN 80-200-1044-0.

[3] NAVARA, Mirko a Aleš, Němeček. *Numerické metody.* Praha: Vydavatelství ČVUT, 2003. ISBN 80-01-02689-2.

[4] DONT, Miroslav. *Numerické metody – cvičení.* Praha: ČVUT, 1990. ISBN 80-01-00400-7.

[5] ARLOW, Jim a Ila, Neustadt. *UML 2 a unifikovaný proces vývoje aplikací*. Brno: Computer Press a.s., 2007. ISBN 978-80-251-1503-9.

[6] BARNET, Zdeněk.: *Aproximace signálu s využitím evolučních technik,* Praha, 2013, Bakalářská práce, FD ČVUT.

[7] GALASSI, M. et al: *GNU Scientific Library Reference Manual* [online]. 3. vyd.
[cit. 2015-05-28]. ISBN 0954612078.
Dostupné z: http://www.gnu.org/software/gsl/manual/html_node/

[8] POLI, R., W.B. Langdon, and N.F. McPhee.: *A Field Guide to Genetic Programming* [online]. [cit. 2015-05-28]. ISBN 978-1-4092-0073-4.
Dostupné z: http://www.gp-field-guide.org.uk.

9 Seznam obrázků

Obr. 2.1: Vývojový diagram GA s křížením a mutací.....10
Obr. 2.2: Jednobodové křížení.....11
Obr. 2.3: Mutace.....11
Obr. 2.4: Stromová struktura reprezentující výraz (a+1)*(b-2).....13
Obr. 2.5: Strom hloubky 2 generovaný (a) úplnou a (b) růstovou metodou.....13
Obr. 2.6: Lineární reprezentace stromu.....14
Obr. 2.7: Křížení stromových struktur.....15
Obr. 2.8: Mutace stromových struktur.....16
Obr. 3.1: Zjednodušený diagram tříd.....24
Obr. 3.2: Objektový diagram.....31
Obr. 5.1: Graf 1.....54
Obr. 5.2: Graf 2.....55

10 Seznam tabulek

Tabulka 1: Definice třídy Diff_R dle UML 2.....25
Tabulka 2: Definice třídy GPTree dle UML 2.....27
Tabulka 3: Kódování hodnot jedince.....29
Tabulka 4: Implementace třídy Fitness dle UML 2.....32

11 Přílohy

V přílohách jsou uvedeny některé pomocné zdrojové kódy. Kompletní program je k dispozici v elektronické podobě v komprimovaném souboru .zip.

11.1 Pomocný program pro generování průběhů diferenciálních rovnic

Program diff_rce.cpp, v jehož zdrojovém kódu upravíme rovnici (`int func()` – ve zdrojovém kódu v elektronické příloze je připraveno několik funkcí, ze kterých lze vybírat), krok a rozsah generovaných dat. Výstup programu je na obrazovku, proto při volání z příkazové řádky použijeme přesměrování výstupu do textového souboru (C:\diff_rce.exe > nazev_souboru.txt). Ten již lze použít jako vstup hlavního programu.

Kompletní zdrojový kód:

```
#include <cstdlib>
#include <iostream>
#include <cmath>

using namespace std;


#include <stdio.h>
     #include <gsl/gsl_errno.h>
     #include <gsl/gsl_matrix.h>
     #include <gsl/gsl_odeiv.h>

     int func (double t, const double y[], double f[], void *params)
     {

       f[0] = y[1];
       f[1] = -(16+16*cos(2*t))*y[0];
       return GSL_SUCCESS;
     }
```

```
int jac (double t, const double y[], double *dfdy,
         double dfdt[], void *params)
    {
      double mu = *(double *)params;
      gsl_matrix_view dfdy_mat
        = gsl_matrix_view_array (dfdy, 2, 2);
      gsl_matrix * m = &dfdy_mat.matrix;
      gsl_matrix_set (m, 0, 0, 0.0);
      gsl_matrix_set (m, 0, 1, 1.0);
      gsl_matrix_set (m, 1, 0, -2.0*mu*y[0]*y[1] - 1.0);
      gsl_matrix_set (m, 1, 1, -mu*(y[0]*y[0] - 1.0));
      dfdt[0] = 0.0;
      dfdt[1] = 0.0;
      return GSL_SUCCESS;
    }

    int
    main (void)
    {
      const gsl_odeiv_step_type * T
        = gsl_odeiv_step_rk4;

      gsl_odeiv_step * s
        = gsl_odeiv_step_alloc (T, 2);
      //gsl_odeiv_control * c
      //  = gsl_odeiv_control_y_new (1e-4, 0.0);
      //gsl_odeiv_evolve * e
      //  = gsl_odeiv_evolve_alloc (2);

      double mu = 10;
      gsl_odeiv_system sys = {func, /*jac*/ NULL, 2, &mu};

      double t = 0.0, t1 = 40.0;
      double h = 1e-1;
      double y[2] = { 0.0, 1.0 };
```

```
        double y_err[2];
        double dydt_in[2], dydt_out[2];

        int status;

          //printf ("control method is '%s'\n",gsl_odeiv_control_name
(c));

        GSL_ODEIV_FN_EVAL(&sys, t, y, dydt_in);

       printf ("%.5e %.5e %.5e\n", t, y[0], y[1]);

        while (t < t1)
          {
                //status = gsl_odeiv_evolve_apply (e, c, s,&sys,&t,
t1,&h, y);
            status = gsl_odeiv_step_apply (s, t, h, y, y_err,
                                           dydt_in,
                                           dydt_out,
                                           &sys);
            dydt_in[0] = dydt_out[0];
            dydt_in[1] = dydt_out[1];

            t += h;
            if (status != GSL_SUCCESS)
                break;

            printf ("%.5e\t%.5e \n", t, y[0]/*, y[1]*/);
          }

        gsl_odeiv_step_free (s);
        //system("PAUSE");
        return 0;
      }
```