VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ
FACULTY OF MECHANICAL ENGINEERING

ÚSTAV AUTOMATIZACE A INFORMATIKY
INSTITUTE OF AUTOMATION AND COMPUTER SCIENCE

ANALYTICKÁ ŘEŠENÍ VYBRANÝCH TYPŮ DIFERENCIÁLNÍCH ROVNIC: SOFTWAROVÁ PODPORA PRO STUDENTY TECHNICKÝCH OBORŮ

ANALYTICAL SOLUTIONS FOR SELECTED TYPES OF DIFFERENTIAL EQUATIONS: SOFTWARE SUPPORT TOOL FOR STUDENTS IN TECHNICAL FIELDS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — Daniel Neuwirth, DiS.
AUTHOR

VEDOUCÍ PRÁCE — Ing. Jan Roupec, Ph.D.
SUPERVISOR

BRNO 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA STROJNÍHO INŽENÝRSTVÍ

Zadání bakalářské práce

Ústav:	Ústav automatizace a informatiky
Student:	**Daniel Neuwirth, DiS.**
Studijní program:	Strojírenství
Studijní obor:	Aplikovaná informatika a řízení
Vedoucí práce:	**Ing. Jan Roupec, Ph.D.**
Akademický rok:	2016/17

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

Analytická řešení vybraných typů diferenciálních rovnic: softwarová podpora pro studenty technických oborů

Stručná charakteristika problematiky úkolu:

Vývoj softwaru umožňujícího řešení vybraných typů diferenciálních rovnic, za použití algoritmů určených pro ruční neautomatizovaný výpočet, sloužící jako výuková pomůcka pro kontrolu postupu výpočtu, určená zejména pro studenty technických oborů vysokých škol.

Cíle bakalářské práce:

Cílem práce je vytvoření softwarové pomůcky určené pro podporu výuky v podoblasti matematiky diferenciálního počtu, která se zabývá řešením vybraných typů lineárních diferenciálních rovnic. Klíčovým výstupem tohoto softwaru bude zobrazení nejen výsledku, ale i správného a úplného postupu řešení. Zvolená metoda má pomoci studentům snadněji pochopit principy, na nichž jsou použité způsoby řešení založeny.

Seznam doporučené literatury:

DAWKINS, Paul. Differential equations [online]. Beaumont, Texas: Lamar University, 2007. Dostupné z: http://tutorial.math.lamar.edu/.

Termín odevzdání bakalářské práce je stanoven časovým plánem akademického roku 2016/17.

V Brně, dne 20. 10. 2016

L. S.

doc. Ing. Radomil Matoušek, Ph.D.
ředitel ústavu

doc. Ing. Jaroslav Katolický, Ph.D.
děkan fakulty

ABSTRAKT

Cílem této práce bylo vytvořit počítačovou aplikaci s jednoduchým uživatelským rozhraním, určenou pro řešení vybraných typů diferenciálních rovnic, jejíž výstup není omezen na prosté zobrazení konečného výsledku, nýbrž zahrnuje i kompletní postup výpočtu, a díky tomu může sloužit jako podpůrná výuková pomůcka pro studenty vysokých škol.

ABSTRACT

The aim of this thesis was to create a computer application with a simple user interface for solving selected types of differential equations, whose output is not limited to display a final result only, but also includes a complete calculation procedure and therefore can serve as a supportive teaching aid for university students.

KLÍČOVÁ SLOVA

diferenciální rovnice, počítačový algebraický systém, programovací jazyk C++

KEYWORDS

differential equation, computer algebra system, C++ programming language

BIBLIOGRAFICKÁ CITACE

NEUWIRTH, Daniel. *Analytická řešení vybraných typů diferenciálních rovnic: softwarová podpora pro studenty technických oborů.* Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, 2017. 77 s. Vedoucí bakalářské práce Ing. Jan Roupec, Ph.D.

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu své bakalářské práce Ing. Janu Roupcovi, Ph.D. za odborné vedení při tvorbě této práce, za cenné rady a návrhy ke zlepšení, drobné korektury a zejména za jeho entuziasmus, se kterým k tomuto procesu přistupoval.

ČESTNÉ PROHLÁŠENÍ

Prohlašuji, že tato práce je mým původním dílem, zpracoval jsem ji samostatně pod vedením Ing. Jana Roupce, Ph.D. a s použitím literatury uvedené v seznamu literatury.

V Brně dne 25. 5. 2017 ……

Daniel Neuwirth

OBSAH

1 ÚVOD

Současná fáze vývoje lidstva, ve které hrají nezastupitelnou roli informační technologie, má mnoho výhod, jež nám usnadňují náš každodenní život (a zvyšují hladinu stresu). Nejnovější zprávy a novinky z celého světa jsou prakticky okamžitě k dohledání na Internetu, velkou část administrativních úkonů lze řešit elektronickou cestou, komunikace s přáteli či obchodními partnery je okamžitá, ať již prostřednictvím e-mailu, Facebooku či jiných kanálů. Dříve těžko řešitelný či zcela neřešitelný problém lze dnes, díky všudypřítomnosti počítačů a dostupnosti softwarových řešení, zvládnout relativně snadno, způsoby v minulosti těžko představitelnými. Přes všechen dosažený pokrok však pro některé specifické úkoly není dostupnost takovýchto řešení vždy samozřejmostí, bývá omezena finančními nároky příp. dalšími požadavky, a právě proto se v takovýchto oblastech nalézá skrytý potenciál, který je možno zužitkovat.

Mezi výše zmíněné oblasti spadá i problematika diferenciálního počtu, na kterou se právě tato práce významně zaměřuje, konkrétně pak způsoby řešení diferenciálních rovnic. Ač to nemusí být na první pohled zjevné, diferenciální rovnice se nalézají všude kolem. Lze je použít pro popis mnoha, i zdánlivě nesouvisejících přírodních dějů, a proto je především pro studenty a v návaznosti tedy i budoucí absolventy technických oborů vysokých škol, nezbytná znalost principů jejich řešení.

Z pohledu běžného uživatele, který nemá potřebu proniknout do dané problematiky hlouběji, je v případě nutnosti postačujícím řešením použít některý ze softwarových produktů, jež se zaměřují pouze na zobrazení konečného výsledku zadaného výpočtu. Takovýchto aplikací je v současné době v nabídce dostatečné množství. Z pohledu studenta, který potřebuje nejprve pochopit principy a postupy, na jejichž základě lze dospět ke konečnému výsledku, a zejména si potřebuje ověřit, zda při vlastním řešení postupoval správně, se ovšem tato situace jeví býti zcela neuspokojivou. Softwarových řešení, která nabízejí možnost zobrazit kompletní postup, jakým bylo konkrétního výsledku po jednotlivých výpočetních krocích dosaženo, je poměrně málo. Navíc drtivá většina z nich jsou produkty komerční, jejichž pořízení se vyplatí pouze velkým firmám či korporacím.

Cílem této práce proto bylo vytvořit a popsat jednoduchou, volně dostupnou aplikaci, plnící funkci podpůrné výukové pomůcky pro studenty, jež se pravidelně setkávají s potřebou řešit právě takové problémy, které lze s výhodou popsat pomocí diferenciálních rovnic. Zobrazení úplného postupu řešení tak bylo jedním z klíčových požadavků, jež měl software, jehož návrh a implementaci popisuje tato práce, splnit. Ta je z důvodu přehlednosti a logické návaznosti členěna do čtyř hlavních kapitol, jež jsou navíc doplněny mnoha poznámkami, jejichž úkolem je ozřejmit použitou terminologii příp. uvést problematiku do historických souvislostí.

V kapitole č. 2 je nejprve proveden stručný výklad matematického aparátu, který slouží jako základ při řešení problémů popsaných diferenciálními rovnicemi. Především jsou zde zmíněna hlediska, dle kterých lze tyto rovnice dělit na jednotlivý typy, a s tím také související metody řešení některých konkrétních, často se vyskytujících variant. Tato kapitola pochopitelně není, a vzhledem ke svému rozsahu, ani nemůže být, vyčerpávající. Zaměřuje se téměř výhradně na výpočetní metody, které jsou použity v popisované aplikaci.

V následující kapitole je pojednáno o softwarových produktech podobného zaměření, jež jsou momentálně na trhu k dispozici, ať již ve formě komerční či volně dostupné aplikace. Stručně je popsán princip jejich funkce a zhodnocení jejich výhod a nevýhod. Jsou uvedeny též adresy webů, ze kterých lze tyto aplikace, resp. v případě komerčních produktů jejich zkušební verze s omezenou funkčností, stáhnout a vyzkoušet si, jak se s nimi pracuje. Ještě jednodušší situace je u on-line služeb, pro jejichž použití postačí uživateli běžný webový prohlížeč.

V kapitole č. 4 jsou nastíněny možnosti technických řešení, z nichž lze při návrhu obdobné aplikace volit. Jedná se především o volbu vhodného programovacího jazyka, integrovaného vývojového prostředí resp. frameworku, způsobu návrhu grafického uživatelského rozhraní a použití dalších nástrojů, které usnadňují vývoj softwaru. Podrobněji je pojednáno zejména o programovacím jazyce C++, konkrétně o jeho revizi C++11, a též vývojovém frameworku Qt, jež byly pro vývoj této aplikace zvoleny.

Rozhodnutí použít pro vývoj kombinaci Qt a C++ je podloženo několika důvody. Jazyk C++ patří dlouhodobě mezi nejpopulárnější programovací jazyky, je k němu k dispozici ohromné množství dokumentace, ať již v elektronické či tištěné formě, a je nejen doprovázen rozsáhlou standardní knihovnou s mnoha vestavěnými funkcemi, nýbrž je k němu zdarma k dispozici i množství dalších knihoven třetích stran (vč. výpočtových a vědeckých), jež značně urychlují vývoj. Navíc umožňuje zvolit různá programovací paradigmata, čímž přenechává vývojářům možnost vybrat si vlastní styl programování. Nelze také opomenout fakt, že C++ je jazykem kompilovaným a poměrně nízkoúrovňovým, který vyniká rychlostí zpracování.

Programový framework Qt umožňuje, mimo jiné, vývoj aplikací právě ve spojitosti s užitím jazyka C++, disponuje vynikající vestavěnou i on-line nápovědou a má integrovány veškeré potřebné vývojové nástroje vč. automatické kompletace kódu. Díky vestavěnému návrhovému modulu též značně usnadňuje vývoj uživatelského rozhraní. Je navíc multiplatformní. Tato všechna kritéria pochopitelně hrála roli při volbě tohoto vývojového prostředí, kromě dalšího vlivu, který je sice značně subjektivní, avšak ne zanedbatelný, a to osobní preference autora založená na předchozích zkušenostech s vývojem aplikací za užití tohoto prostředí.

Vlastní implementace je popsána v kapitole č. 5. Postupně je vysvětleno, jakým způsobem je aplikace členěna z hlediska logické struktury, jsou nastíněny programátorské techniky použité při implementaci některých funkcí a je též popsán návrh grafického uživatelského rozhraní, doplněný názornými ilustracemi. Následuje několik praktických ukázek použití, na kterých je demonstrováno, jakým způsobem aplikace pracuje a jaký výstup uživateli podává. V závěru je zmíněno, kterým směrem by se měl ubírat její budoucí vývoj. Celý aplikační kód je připojen jako elektronická příloha a je bohatě komentován.

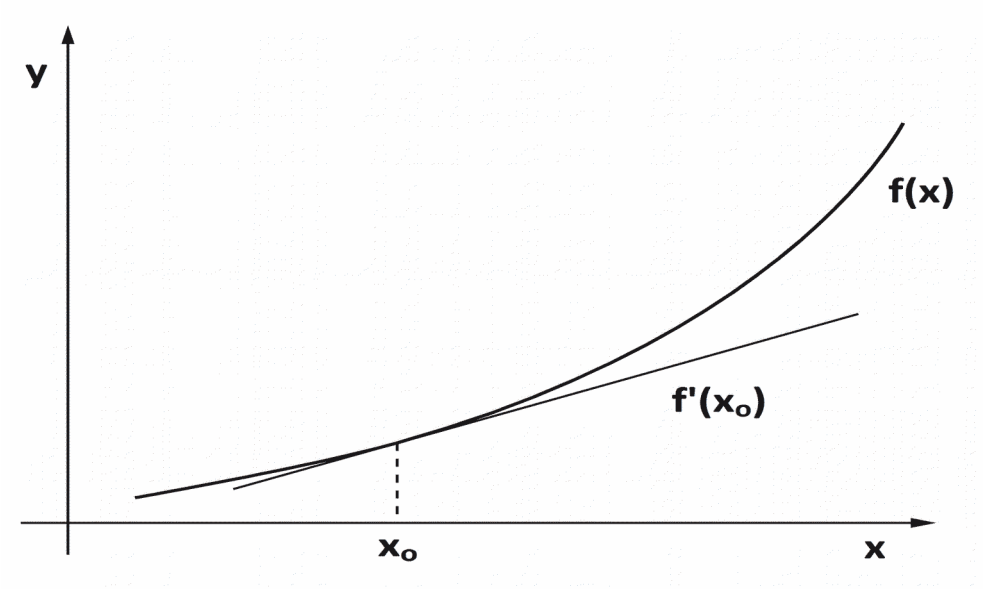
2 DIFERENCIÁLNÍ POČET

Diferenciální počet (*differential calculus*) je jedním ze dvou podoborů tzv. **infinitesimálního počtu** (*calculus,* též matematická analýza), který se zabývá studiem změn funkčních hodnot *f(x)* v závislosti na změnách vstupních hodnot nezávisle proměnné/ých. Druhým podoborem infinitesimálního počtu je **integrální počet** (*integral calculus*).

Základním stavebním kamenem studia diferenciálního počtu je **derivace** (*derivative*) funkce, jež popisuje charakter změny této funkce v bezprostředním okolí zvoleného bodu x_0 (hodnoty nezávisle proměnné).

2.1 Derivace

Derivace funkce reálné proměnné určuje míru, jakou reaguje funkční hodnota *f(x)* na změny vstupních hodnot nezávisle proměnné *x*. Pokud existuje, lze ji snadno interpretovat graficky jako tečnu ke grafu této funkce v daném vstupním bodě x_0, jak je znázorněno na obr. 1.



obr. č. 1: Grafická interpretace derivace jako tečny ke křivce v daném bodě

Pro označení derivace se používá několik různých způsobů zápisu. O rozvoj infinitesimálního počtu se zasloužilo mnoho významných matematiků (G.W. Leibniz, I. Newton ad.)[1], z nichž každý používal vlastní systém označování. Nejběžnější z nich jsou:

$$\frac{dy}{dx} \text{ dle Leibnize} \quad f'(x) \text{ dle Lagrange} \quad \dot{y} \text{ dle Newtona} \tag{1}$$

Hodnotu derivace lze určit nejen pro funkci jedné reálné proměnné, nýbrž i pro funkce více reálných proměnných. V takovém případě hovoříme o tzv. **parciálních derivacích** (*partial derivatives*). Pokud je určitá funkce v daném bodě **diferencovatelná** (*differentiable*), má zde tolik parciálních derivací, kolik má nezávisle proměnných.

Proces určování derivace se nazývá **derivování** (*differentation*).

1 řazeno abecedně, pořadí v žádném případě nesouvisí s dávným sporem těchto dvou učenců, který z nich vlastně založil tuto matematickou disciplínu [1, 2], ačkoli může naznačit, kdo je autorovým favoritem; přehled významných vědců zmíněných v této práci viz Příloha A

2.2 Vztah mezi diferenciálním a integrálním počtem

Diferenciální a integrální počet jsou vzájemně svázány pomocí **základní věty integrálního počtu** (*fundamental theorem of calculus*). První část této věty říká, že neurčitý integrál spojité funkce *f(t)* definované na uzavřeném intervalu $\langle a,b\rangle \in \mathbb{R}$ je roven primitivní funkci *F(x)*, jež je v uzavřeném intervalu $\langle a,b\rangle$ spojitá, v otevřeném intervalu (a,b) diferencovatelná, a jejíž derivace pro všechna *x* z tohoto intervalu je rovna *f(x)*. [3]

$$F(x)=\int_a^x f(t)\,dt \qquad F'(x)=f(x) \tag{2}$$

Z (2) pak jednoznačně vyplývá, že integrování je zpětným procesem derivování.

2.3 Diferenciální rovnice

Jakákoli rovnice obsahující derivace, ať již obyčejné či parciální, je nazývána **diferenciální rovnicí** (*differential equation*). Diferenciální rovnice [4] jsou jedním ze základních nástrojů, které používáme pro popis přírodních dějů.

Jako příklad diferenciální rovnice možno uvést **Bernoulliho diferenciální rovnici** [5], kterou sestavil roku 1695 švýcarský matematik Jacob Bernoulli (1654-1705).

$$y'+P(x)\,y=Q(x)\,y^n \tag{3}$$

2.3.1 Základní pojmy

Pro usnadnění dalšího výkladu je vhodné nejprve zavést některé důležité pojmy.

řád diferenciální rovnice (*order of differential equation*)
Řád diferenciální rovnice je určen řádem nejvyšší derivace, která se v této rovnici vyskytuje:

příklad rovnice 1. řádu: $\cos(x)\,y'+\sin(x)\,y=2\cos^3(x)\sin(x)-1$ [2]

příklad rovnice 3. řádu: $y'''-12\,y''+48\,y'-64\,y=12-32e^{-8t}+2e^{4t}$

příklad rovnice 4. řádu: $2\,y^{(4)}+11\,y'''+18\,y''+4\,y'-8\,y=0$ (4)

řešení diferenciální rovnice (*solution of differential equation)*
Řešením diferenciální rovnice je jakákoli funkce *y(x)*, která danou rovnici splňuje.

počáteční podmínky (*initial conditions*)
Množina podmínek, která nám umožní rozhodnout, které konkrétní řešení z množiny všech možných řešení je tím, které hledáme. Počáteční podmínky jsou hodnotami závisle proměnné *y(x)* příp. jejích derivací v konkrétně zadaných bodech (např. *y(0) = 1, y'(0) = 0*). Pro úspěšné vyřešení dané diferenciální rovnice potřebujeme znát takový počet počátečních podmínek, jako je řád této rovnice.

obecné řešení (*general solution*)
Obecným řešením diferenciální rovnice je, jak plyne již z jeho názvu, řešení v obecné formě s nevyčíslenými konstantami, tj. takové řešení, u nějž nebyly aplikovány počáteční podmínky.

---

2 vědecká komunita není, co se týká označování derivací, jednotná; v závislosti na konkrétní vědní disciplíně a autorovi se použitá notace mění, proto i tato práce používá v různých svých částech rozdílnou notaci (nikdy však ne v rámci jednoho příkladu)

aktuální řešení (*actual solution*)
Aktuální řešení diferenciální rovnice je specifickým řešením, které splňuje zadanou rovnici nejen obecně, nýbrž konkrétně pro danou množinu počátečních podmínek. Z řešení obecného ho snadno získáme prostým aplikováním zadaných počátečních podmínek, což nám umožní numericky určit hodnoty jednotlivých konstant (viz. kap. 2.3.6).

Např. rovnice $y'''+4y''-7y'-10y=0$
má při počátečních podmínkách $y(0)=0, y'(0)=1, y''(0)=0$
obecné řešení $y=c_1e^{-5x}+c_2e^{-x}+c_3e^{2x}$ a
aktuální řešení $y=-0.035714e^{-5x}-0.25e^{-x}+0.285714e^{2x}$ (5)

2.3.2 Využití v technických oborech

Diferenciální počet je hojně využívaným nástrojem ve většině exaktních vědeckých disciplín. Široké uplatnění nalézá zejména ve fyzice. V kinematice se užívá např. při určování okamžité rychlosti a zrychlení pohybujících se těles.

Okamžitá rychlost (*velocity*) je dána změnou **polohy** (*displacement,* též posunutí) v čase [6]:

$$\vec{v}=\lim_{\Delta t\to 0}\frac{\Delta\vec{r}}{\Delta t}=\frac{d\vec{r}}{dt}\,, \tag{6}$$

kde $\vec{r}$ značí posunutí (změnu polohy) a t čas.

Okamžité zrychlení (*acceleration*) je pak derivací vektoru rychlosti podle času (tj. druhou derivací vektoru posunutí podle času) [6]:

$$\vec{a}=\frac{d^2\vec{r}}{dt^2}=\frac{d\vec{v}}{dt}$$ [3] (7)

Dalším obecně známým příkladem je **Newtonův druhý pohybový zákon** (*Newton's second law of motion*), označovaný též jako Zákon síly, který říká, že síla působící na těleso je dána časovou změnou, tj. derivací, hybnosti [6]:

$$\vec{F}=\frac{d\vec{p}}{dt}=m\frac{d\vec{v}}{dt}=m\vec{a}\,, \tag{8}$$

kde p představuje hybnost tělesa a m jeho hmotnost, která je v klasické (newtonovské) fyzice konstantní, proto je možné ji vytknout před derivaci. Ostatní proměnné vyplývají z (6) a (7).

S potřebou řešit diferenciální rovnice (v tomto případě 2. řádu) se setkáváme též při popisu a zkoumání soustav s vibracemi, jež se zpravidla vyjadřují pomocí tzv. **pohybové rovnice pro soustavu s mechanickými kmity** (*equation of motion for a vibrating system*). [7]

Pro soustavu s volnými netlumenými kmity platí: $m\ddot{q}+kq=0$, (9)
pro soustavu s volnými tlumenými kmity platí: $m\ddot{q}+b\dot{q}+kq=0$, (10)
pro soustavu s nucenými kmity platí: $m\ddot{q}+b\dot{q}+kq=Q(t)$, (11)

kde m vyjadřuje hmotnost, b součinitel útlumu, k tuhost pružiny, q vyjadřuje výchylku (změnu polohy) a $Q(t)$ představuje součet vnějších sil působících na soustavu.

---

3 v praxi se lze někdy setkat též s použitím třetí derivace vektoru posunutí (změna zrychlení), označované v anglosaském prostředí jako *jerk* [8] (vhodný český překlad tohoto pojmu bude možno nalézt jen s obtížemi), který je ukotven v normě ISO 2041:2009

Rovnice (9), (10) jsou typickými příklady homogenní diferenciální rovnice, na rozdíl od (11), která je rovnicí nehomogenní (blíže viz. kap. 2.3.3.4).

2.3.3 Druhy diferenciálních rovnic

Diferenciální rovnice lze dělit dle mnoha různých hledisek. Nejčastější způsoby klasifikace zahrnují dělení na obyčejné a parciální, lineární a nelineární či homogenní a nehomogenní. Dalším častým hlediskem je rozlišení, zda se jedná o rovnici s konstantními či proměnnými koeficienty. Všechny zmíněné typy diferenciálních rovnic budou v následujících kapitolách stručně popsány a bude také vysvětleno, v čem se odlišují.

2.3.3.1 Obyčejné a parciální diferenciální rovnice

Obyčejnou diferenciální rovnicí (*ordinary differential equation*) je označována diferenciální rovnice, která obsahuje neznámou funkci (a její derivace) pouze jedné nezávisle proměnné.

$$a_0 \frac{d^2 y}{dx^2} + a_1 \frac{dy}{dx} + a_2 y = 0 \tag{12}$$

Naproti tomu, **parciální diferenciální rovnicí** (*partial differential equation*) je diferenciální rovnice, která obsahuje neznámou funkci (a její derivace) více než jedné nezávisle proměnné.

Jedním z příkladů takovéto rovnice je **obecná rovnice vedení tepla** (*heat equation*) [9], která popisuje šíření tepla tělesem v závislosti na čase.

$$\frac{\partial u}{\partial t} = k \partial^2 \frac{u}{\partial x^2} + \frac{Q(x,t)}{c\rho}, \tag{13}$$

kde k vyjadřuje koeficient tepelné vodivosti, c měrnou tepelnou kapacitu, ρ je hustota a $Q(x,t)$ představuje hustotu tepelných zdrojů; znak ∂ se používá pro označení parciální derivace.

2.3.3.2 Lineární a nelineární diferenciální rovnice

V případě **lineární diferenciální rovnice** (*linear differential equation*) jsou hledaná funkce $y(x)$ i všechny její derivace vždy v první mocnině a nevyskytují se v ní ani součiny této funkce s jejími derivacemi, ani součiny jednotlivých derivací této funkce.

Všechny ostatní rovnice, které výše udané podmínky nesplňují, jsou nazývány **nelineárními diferenciálními rovnicemi** *(nonlinear differential equation*). Nelineární rovnice jsou většinou obtížně řešitelné. Jako příklad možno uvést třeba rovnici Laguerreovu [10], pojmenovanou po významném francouzském matematikovi Edmondu Nicolasi Laguerreovi (1834-1886).

$$x y'' + (1-x) y' + k y = 0, \quad k \in \mathbb{R} \tag{14}$$

2.3.3.3 Koeficienty v diferenciálních rovnicích

Každá diferenciální rovnice obsahuje buď **proměnné koeficienty** (*non-constant coefficients*) nebo, v případě jednodušších závislostí, **konstantní koeficienty** (*constant coefficients*).

Lineární diferenciální rovnice s proměnnými koeficienty je v obecném tvaru (zde uvedeno na příkladu rovnice 2. řádu) vyjádřena následovně:

$$p(x) y'' + q(x) y' + r(x) y = 0 \tag{15}$$

Diferenciální rovnice s konstantními koeficienty je ve své podstatě speciálním případem (15), ve kterém jsou $p(x)$, $q(x)$ a $r(x)$ konstantní.

2.3.3.4 Homogenní a nehomogenní diferenciální rovnice

Homogenní diferenciální rovnicí (*homogeneous differential equation*)[4] je nazývána rovnice s nulovou pravou stranou. Příkladem lineární homogenní diferenciální rovnice je tedy i (15).

Obecný tvar nehomogenní diferenciální rovnice (zde 2. řádu) lze vyjádřit následovně:

$$p(x)\,y''+q(x)\,y'+r(x)\,y=g(x)\,,\quad g(x)\neq 0 \tag{16}$$

Řešení nehomogenních diferenciálních rovnic (někdy zvaných též úplné) se skládá ze dvou kroků, tzv. **homogenního** a **partikulárního** řešení: $y=y_h+y_p$ (17)

Podstata homogenního řešení spočívá v prosté záměně nenulové pravé strany nulovou pravou stranou, čímž tuto rovnici převedeme na homogenní (někdy též nazývanou zkrácená rovnice).

Pro určení partikulárního řešení, tj. řešení pravé strany rovnice, se v praxi nejčastěji používají **Lagrangeova metoda variace konstant** (*variation of parameters*), pojmenovaná po italsko-francouzském matematikovi Josephovi-Louisi Lagrangeovi (1736-1813)[5], či obecně o něco méně použitelné **Metoda neurčitých koeficientů** (*method of undetermined coefficients*) a s ní příbuzná **Metoda anihilátorů** (*annihilator method*). [11]

2.3.4 Způsoby řešení diferenciálních rovnic

V následujících subkapitolách budou naznačeny základní způsoby řešení některých vybraných typů diferenciálních rovnic.

2.3.4.1 Diferenciální rovnice 1. řádu

Diferenciálními rovnicemi 1. řádu (*first-order differential equation*) nazýváme, v souladu s názvoslovím zavedeným v kap. 2.3.1, takové diferenciální rovnice, ve kterých se hledaná funkce *y(x)* vyskytuje max. v první derivaci. Pro řešení těchto rovnic neexistuje žádný obecně použitelný postup řešení, pro každý typ rovnice je nutno použít specifický přístup.

Mezi nejčastější představitele tohoto typu rovnic patří **lineární rovnice 1. řádu** (*linear first-order*), **rovnice se separovanými** resp. **separovatelnými proměnnými** (*separable equation*), **homogenní rovnice** (*homogeneous equation*) řešené pomocí substituce či **exaktní rovnice** (*exact equation*) [7]. Příkladem rovnice 1. řádu je též (3).

2.3.4.2 Diferenciální rovnice vyšších řádů

Obecný zápis lineární diferenciální rovnice n-tého řádu s konstantními koeficienty, na které se především tento text zaměřuje, je následující:

$$a_0\,y^{(n)}+a_1\,y^{(n-1)}+a_2\,y^{(n-2)}+\ldots+a_{n-1}\,y'+a_n\,y=g(x) \tag{18}$$

Homogenním řešením (y_h) takovéto rovnice je poté výraz:

$$y_h(x)=c_1\,y_1(x)+c_2\,y_2(x)+c_3\,y_3(x)+\ldots+c_{n-1}\,y_{n-1}(x)+c_n\,y_n(x)\,, \tag{19}$$

---

4 homogenní rovnicí bývá v literatuře často označována také nelineární diferenciální rovnice prvního řádu, kterou lze řešit substitucí např. $x\,y\,y'=x^2-y^2$ [12], viz též kap. 2.3.4.1

5 tento významný matematik, člen Francouzské akademie, jejíž členové jsou označováni jako *les immortels* (nesmrtelní) byl zmíněn již v kap. 2.1 v souvislosti s jednou z hojně užívaných metod označování derivací; narodil se v italském Turíně jako Giuseppe Lodovico Lagrangia a do Francie, konkrétně do Paříže, přesídlil až ve svých 51 letech

za předpokladu, že jednotlivá řešení jsou **lineárně nezávislá** (*linearly independent*). Lineární nezávislost se určí dle kap. 2.3.5.

K homogennímu řešení (18) dospějeme pomocí metody objevené švýcarským matematikem a fyzikem Leonhardem Eulerem (1707-1783) spočívající v sestavení **charakteristické rovnice** (*characteristic equation*), což je algebraická rovnice n-tého stupně, ve které jsou jednotlivé derivace nahrazeny příslušnými mocninami.

$$a_0 y^n + a_1 y^{n-1} + a_2 y^{n-2} + \ldots + a_{n-1} y + a_n = 0 \tag{20}$$

Vyřešením (20) získáme *n* kořenů (r_1–r_n) ve tvaru

$$y_1(x) = e^{r_1 x}, y_2(x) = e^{r_2 x}, \ldots, y_n = e^{r_n x}\ ^{6} \tag{21}$$

Další postup řešení se odvíjí od typu kořenů, které mohou nabývat několika různých forem:

a) reálné různé $r_1 \neq r_2 \neq \ldots \neq r_n, \quad r_i \in \mathbb{R}$

b) reálné násobné $r_1 = r_2 = \ldots = r_n, \quad r_i \in \mathbb{R}$

c) komplexní sdružené $r_{j,j+1} = a_j \pm b_j i\ ^{7}, \quad r_j \in \mathbb{C}, \quad a_j, b_j \in \mathbb{R}$

Reálné různé kořeny (*real distinct roots*)
Reálné různé kořeny jsou již ze své podstaty lineárně nezávislé, takže k hledanému řešení lze dospět v tomto případě velmi snadno, prostou aplikací **principu superpozice** (*principle of superposition*):

$$y_h(x) = c_1 e^{r_1 x} + c_2 e^{r_2 x} + \ldots + c_n e^{r_n x} \tag{22}$$

Příklad rovnice 3. řádu se třemi různými reálnými kořeny je na obr. 2.

$$y''' - 2y'' - y' + 2y = 0$$

roots: -1.000000, 1.000000, 2.000000

solutions: e^{-x}, e^{x}, e^{2x}

$$y = c_1 e^{-x} + c_2 e^{x} + c_3 e^{2x}$$

obr. č. 2: Reálná různá řešení LODR 3. řádu (výstup z programu)

Reálné násobné kořeny (*real multiple roots*)
Řešení ve tvaru (22) předpokládá, že jednotlivé kořeny jsou lineárně nezávislé. Jelikož tento předpoklad není v případě vícekrát se opakujících kořenů splněn, je nutno pro tento případ řešení upravit do následujícího tvaru:

$$y_h(x) = c_1 [x^a] e^{r_1 x} + c_2 [x^a] e^{r_2 x} + \ldots + c_n [x^a] e^{r_n x}, \quad a \in \mathbb{Z}, a \in (0, n-1) \tag{23}$$

---

6 skutečnost, že kořeny lineární obyčejné diferenciální rovnice (zkr. LODR) s konstantními koeficienty jsou vždy ve tvaru e^{rx}, přičemž $r \in \mathbb{C}$, jako první správně odvodil … ano, přesně tak … Leonhard Euler, žák švýcarského matematika Johanna Bernoulliho (1667-1748), který byl bratrem již dříve zmíněného Jacoba Bernoulliho (viz (3) v kap. 2.3)

7 zde *i* představuje tzv. **imaginární jednotku** (*imaginary unit*), $i^2 = -1$

Výše uvedená úprava je nezbytná, neboť jednotlivé kořeny (z nichž všechny nebo alespoň některé jsou stejné), by při aplikaci (22) nesplňovaly podmínku lineární nezávislosti a rovnice by se tím pádem jevila jako neřešitelná.

Z výše uvedeného vyplývá, že je třeba provést takovou úpravu násobných kořenů, která splní podmínku lineární nezávislosti, ale přitom bude stále platným řešením dané charakteristické rovnice. Obě tyto podmínky jsou splněny, pokud kořeny vynásobíme určitým koeficientem[8], přičemž nejjednodušší způsob je součin řešení a nezávisle proměnné x v příslušné mocnině a, podle toho o kolikátý násobek kořenu se jedná, jak je patrné z obr. 3.

$$y^{(4)}+12y'''+54y''+108y'+81y = 0$$
roots: -3.000000, -3.000000, -3.000000, -3.000000
solutions: e^{-3x}, e^{-3x}, e^{-3x}, e^{-3x}
$$y = c_1e^{-3x}+c_2xe^{-3x}+c_3x^2e^{-3x}+c_4x^3e^{-3x}$$

obr. č. 3: Reálné násobné řešení LODR 4. řádu (výstup z programu)

Řešení (23) je vlastně zobecněným řešením pro všechny, nejen násobné, typy kořenů. Pokud je hodnota a pro všechny prvky nulová (což nastává právě u neopakujících se kořenů), přejde toto řešení v (22).

Komplexní sdružené kořeny (*complex conjugate roots*)
Charakteristická rovnice nemusí mít vždy řešení z oboru reálných čísel. Zaměřme se proto nyní podrobněji na případ, kdy jsou její kořeny ve tvaru $r_{1,2}=\lambda\pm\mu i$. [7]

Jelikož řešení musí mít formu $y(x)=e^{rx}$, po dosazením za $r_{1,2}$ platí:

$$y_1(x)=e^{(\lambda+\mu i)x}, \quad y_2(x)=e^{(\lambda-\mu i)x}\ ^{9} \tag{24}$$

Pomocí **Eulerova vzorce** (*Euler's formula*)[10] $e^{(i\theta)}=\cos\theta+i\sin\theta$, (25)

a několika základních matematických operací, lze (24) přepsat do podoby:

$$y_h(x)=c_1e^{(\lambda x)}\cos(\mu x)+c_2e^{(\lambda x)}\sin(\mu x) \quad \text{(viz obr. 4)} \tag{26}$$

type: 4th order linear homogeneous ordinary differential equation with constant coefficients
characteristic equation: quartic (biquadratic) equation
type of roots: 4 complex roots
roots of characteristic equation: 1.000000-1.000000i, 1.000000+1.000000i, -1.000000+1.000000i, -1.000000-1.000000i
solutions of differential equation: $e^x\sin(x)$, $e^x\cos(x)$, $e^{-x}\cos(x)$, $e^{-x}\sin(x)$
according to Principle of superposition: $y = c_1e^x\sin(x)+c_2e^x\cos(x)+c_3e^{-x}\cos(x)+c_4e^{-x}\sin(x)$

obr. č. 4: Parametry a řešení rovnice $y^{(4)}+4y=0$ (výstup z programu)

8 zde lze vyjít z prosté skutečnosti, že řešení (kořen) dané rovnice, vynásobené libovolnou konstantou, je také platným řešením [7]

9 pro zjednodušení je uvažována kvadratická charakteristická rovnice (se dvěma kořeny)

10 a zase ten Euler; šíře záběru tohoto génia je skutečně ohromující [13], jak ilustruje i citát z knihy Clifforda Truesdella: *"Approximately one-third of the entire corpus of research on mathematics and mathematical physics and engineering mechanics published in the last three-quarters of the eighteenth century is by him."* [14]

2.3.5 Wronskián

V předchozích kapitolách jsme vycházeli ze skutečnosti, že jednotlivé kořeny charakteristické rovnice musí být vzájemně lineárně nezávislé, aby po aplikaci principu superpozice tvořily obecné řešení dané diferenciální rovnice. Přestože v mnoha případech je toto zřejmé na první pohled, nelze spoléhat pouze na „matematickou intuici". Je třeba znát nějakou metodu, která tuto vlastnost umožní jednoznačně určit. K tomu právě slouží tzv. **Wronskián** (*Wronskian*, též Wronského determinant). Jedná se o determinant tzv. **fundamentální matice** (*fundamental matrix*), kterou lze sestavit následujícím způsobem:

$$\begin{pmatrix} y_1(x_0) & y_2(x_0) & \dots & y_n(x_0) \\ y_1'(x_0) & y_2'(x_0) & \dots & y_n'(x_0) \\ \dots & \dots & \dots & \dots \\ y_1^{(n-1)}(x_0) & y_2^{(n-1)}(x_0) & \cdots & y_n^{(n-1)}(x_0) \end{pmatrix} \tag{27}$$

Pokud determinant matice (27) je nenulový ($W \neq 0$), jsou jednotlivé kořeny lineárně nezávislé a lze poté určit obecné řešení dané diferenciální rovnice. Z výše uvedeného však jednoznačně nevyplývá, že v případě nulového Wronskiánu ($W = 0$) jsou řešení lineárně závislá! [7]

Na druhou stranu je nutno podotknout, že i v těch nemnoha případech, kdy lineárně nezávislá řešení generují nulový Wronskián, není bohužel možno, z důvodu **dělení nulou** (*division by zero*), určit hodnoty jednotlivých konstant c_1–c_n (viz kap. 2.3.6).

2.3.6 Určení konstant

Pokud jsou zadány počáteční podmínky, je možno určit hodnoty konstant c_1–c_n aplikací těchto podmínek na řešení (19), resp. jeho derivace. [7] Konstanty lze v zásadě určit dvěma způsoby. Buďto algebraickým řešením soustavy n rovnic o n neznámých nebo pomocí Cramerova pravidla.

Určení konstant pomocí **Cramerova pravidla** (*Cramer's rule*)
Příslušná konstanta je rovna podílu determinantů dvou matic: ve **jmenovateli** (*denominator*) fundamentální matice (27), v **čitateli** (*nominator*) pak matice, jež má vždy příslušný sloupec, odpovídající číslu konstanty, kterou je v daném okamžiku třeba určit, nahrazený oproti (27) hodnotami funkce $y(x)$ resp. jejích derivací v daném počátečním bodě x_0.

Vzor sestavení matice pro konstantu c_1:

$$\begin{pmatrix} y_0 & y_2(x_0) & \dots & y_n(x_0) \\ y_0' & y_2'(x_0) & \dots & y_n'(x_0) \\ \dots & \dots & \dots & \dots \\ y_0^{(n-1)} & y_2^{(n-1)}(x_0) & \cdots & y_n^{(n-1)}(x_0) \end{pmatrix} \tag{28}$$

Dosazením vypočítaných hodnot c_1–c_n do (19) vznikne aktuální řešení (viz také kap. 2.3.1). Zároveň z výše uvedeného logicky vyplývají dvě důležité skutečnosti. Jednak, že pro určení aktuálního řešení je potřeba znát tolik počátečních podmínek, kolik toto řešení obsahuje konstant. A za druhé, jak již bylo naznačeno v kap. 2.3.5, že $W = 0$ možnost určení konstant zcela vylučuje.

3 DOSTUPNÉ SOFTWAROVÉ NÁSTROJE

Pro **symbolické** (*symbolic*) i **numerické řešení** (*numeric computation*) diferenciálních rovnic, jakož i dalších problémů z oblasti matematické analýzy, je v dnešní době k dispozici poměrně široká nabídka softwarových nástrojů, z nichž některé jsou vyvíjeny jako komerční placené aplikace, některé jsou nabízeny zdarma v rámci open-source[11] komunity.

Takovýto typ softwaru je obvykle nazýván **počítačový algebraický systém** (CAS, *computer algebra system*) neboli systém pro počítačové zpracování matematických výrazů. Původně se tyto programy vyvinuly ze specializovaných matematických aplikací pro superpočítače, dnes jsou však, díky nárůstu výpočetního výkonu[12], k dispozici i pro běžné osobní počítače.

CAS programy lze používat mnoha různými způsoby - jako prostou náhradu kalkulačky, jako nástroj na tvorbu grafů či pro pokročilejší symbolické a numerické výpočty. Většina těchto systémů má navíc vestavěný vlastní programovací jazyk, s jehož pomocí lze vytvářet vlastní algoritmy a funkce.

Tabulka č. 1 uvádí některé z často používaných CAS programů, z nichž některé se hodí spíše pro symbolické (Maple, Mathematica, Maxima) a některé spíše pro numerické (MATLAB, GNU Octave) výpočty.

Název	Autor, výrobce	Vývoj započal	Poslední verze	Označení aktuální verze	Podporované operační systémy
Axiom	Tim Daly	2002	2014[1]	August-2014	BSD, Linux, MacOS
FriCAS	Waldek Hebisch	2007	2017	1.3.2	BSD, Linux, MacOS
GNU Octave	John W.Eaton	1988	2017	4.2.1	BSD, Linux, MacOS, Windows
Mathics	Jan Pöschko	2011	2017	1.0	online aplikace[2]
Maple	MapleSoft	1982	2016	Maple 2016	Linux, MacOS, Windows
Mathematica	Wolfram Research	1988	2017	11.1	Linux, MacOS, Windows
MATLAB	MathWorks	1984	2017	R2017a (9.2)	Linux, MacOS, Windows
Maxima	Macsyma Group	1999[3]	2016	5.39.0	Android, Linux, MacOS, Win
SageMath	William Stein	2005	2017	7.6	Linux, MacOS, Windows
Scilab	Scilab Enterprises	1994	2017	6.0.0	Linux, MacOS, Windows
WolframAlpha	Wolfram Research	2009	2017		online aplikace

1] v dubnu 2017 byla vydána nová verze, prozatím dostupná pouze pro BSD
2] v říjnu 2016 byla vydána i první desktopová verze
3] navazuje na systém Macsyma vyvíjený na MIT již od roku 1968

tab. č. 1: Matematický software

11 jedná se o tzv. **otevřený software** (*open-source*) neboli počítačový software s otevřeným zdrojovým kódem; jeho součástí je vždy i licence vymezující práva uživatele s ohledem na možnosti modifikace kódu a jeho redistribuci; protikladem k open-source je tzv. **proprietární software** (*proprietary, closed-source)*, software s uzavřeným zdrojovým kódem

12 v současné době se stále považuje za platný tzv. **Moorův zákon** (*Moore's law*), vyslovený roku 1965 pozdějším spoluzakladatelem společnosti Intel Corporation Gordonem E. Moorem, který říká, že počet tranzistorů (a tedy i výpočetní výkon) v „běžném“ počítačovém procesoru se každé dva roky zdvojnásobí [15]

3.1 Webové aplikace

Do stále většího popředí zájmu uživatelů i vývojářů se dostávají tzv. **webové aplikace** (*web-based application*), což jsou aplikace používající jako svůj **front-end** prostředí **webového prohlížeče** (*web browser*).

Pro uživatele má použití takové aplikace mnoho výhod, zejména dostupnost z jakéhokoli PC připojeného k Internetu (nehledě na použitý operační systém)[13], a to vč. mobilních telefonů, tabletů či PDA[14]. Odpadá také nutnost instalace (upgradu) nebo v některých krajních případech dokonce obměny hardwaru, pokud má nově vydaná verze dané aplikace vyšší požadavky na **systémové prostředky** (*system resources*).

3.1.1 WolframAlpha

Nejznámější a patrně i nejpoužívanější webovou aplikací pro (nejen) matematické výpočty je produkt společnosti Wolfram Research zvaný **WolframAlpha**, dostupný zdarma[15] na webové adrese http://www.wolframalpha.com/.

Pro náročnější uživatele je systém WolframAlpha dostupný též v placených[15] verzích Pro a Pro Premium, které nabízejí rozšířenou funkcionalitu zahrnující zobrazení postupu řešení či možnost uploadovat (nahrát na server provozovatele) vlastní data pro podrobnou analýzu.

Webové rozhraní WolframAlpha je schopno rozpoznat jak slovní zadání (pouze v angličtině), tak symbolické zadání ve vlastním specializovaném jazyce **Wolfram Language**, pro který je k dispozici rozsáhlá online dokumentace (přes 50.000 stran).

Popularitu WolframAlpha pomáhá navýšit obrovský záběr použití, který sahá od matematiky přes fyziku, strojírenství, astronomii až k historii či jazykovědě. Mezi další podstatné výhody patří velmi podrobná dokumentace s mnoha příklady, a jednoduché a přehledné rozhraní (viz obr. 5). Nevýhodou pro studenty mnoha, zejména však technických, oborů je, že v základní bezplatné verzi tento systém neumožňuje zobrazení postupu daného výpočtu.

zdroj: www.wolframalpha.com

obr. č. 5: Webové rozhraní WolframAlpha

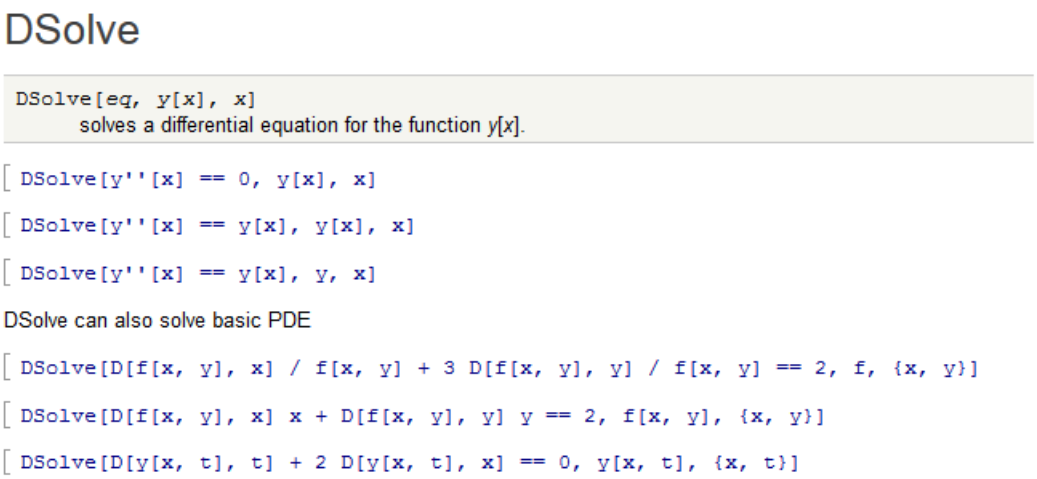
13 zařízení musí být pochopitelně vybaveno podporovaným webovým prohlížečem

14 **osobní digitální pomocník** (*personal digital assistant*) – malý kapesní počítač obvykle vybavený perem (stylus) a dotykovou obrazovkou

15 v základní verzi je systém bezplatný; pokročilejší verze Pro a Pro Premium jsou nabízeny v rámci měsíčního předplatného v cenách od $5 do $12

3.1.2 Mathics

Mathics je volně dostupným webovým řešením CAS systému, jež je od počátku svého vývoje prezentován jako „odlehčená“ alternativa [16] ke komerčnímu systému Mathematica, který je svým uživatelům k dispozici i v online verzi (více o tomto softwaru v kap. 3.2.2). Snaží se tomuto systému přiblížit zejména syntaxí symbolického jazyka (ukázka z dokumentace viz obr. 6), aby tak uživatelům ulehčil případný pozdější přechod na tento sofistikovanější CAS.



obr. č. 6: Výpis z dokumentace k Mathics - příkaz `Dsolve` k řešení diferenciálních rovnic

Mathics vč. zmíněné dokumentace je dostupný na adrese https://mathics.angusgriffith.com/. V průběhu roku 2016 byl tento systém vydán nově též v desktopové variantě, obě verze jsou založené na jazyce **Python**[16].

Jednoduchost a dostupnost Mathicsu (poskytován zdarma pro všechny uživatele) je bohužel do jisté míry vyvážena i jeho nevýhodami - menší škálou dostupných funkcí (týká se i oblasti diferenciálního počtu) a pomalejším zpracováním. [16]

3.2 Komerční desktopové aplikace

V oblasti komerčně vyvíjených desktopových aplikací je trh CAS systémů v současné době poměrně jednoznačně rozdělen mezi trojici velkých „hráčů“: Maplesoft a jeho produkt Maple, Wolfram Mathematica od Wolfram Research a MATLAB od společnosti MathWorks.

Základní přehled rekapituluje tabulka č. 2.

Název	Cena	Dostupné z
Maple	$99 - $2275	http://www.maplesoft.com/products/maple/
Mathematica	$70 - $2495	http://www.wolfram.com/mathematica/
MATLAB	€69 - €2000	https://www.mathworks.com/products/matlab.html

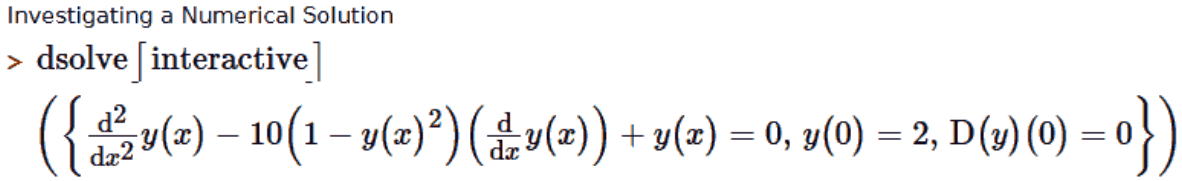
tab. č. 2: Komerční desktopové aplikace

16 populární programovací jazyk vytvořený nizozemským programátorem G. van Rossumem; jedná se o jazyk interpretovaný, v současné době ve verzi 3.6.1; méně obvyklým prvkem, který jej odlišuje od většiny jiných jazyků, je použití horizontálního odsazování pro naznačení oblasti programového bloku; uživatelé Pythonu mají k dispozici velmi rozsáhlou **standardní knihovnu** (*standard library*)

3.2.1 Maple

Maple je vlajkovým produktem kanadské společnosti Maplesoft. Maple je velmi rozsáhlý systém s více než 5.000 vestavěnými matematickými funkcemi pro řešení téměř jakéhokoli problému z oblasti teorie čísel, matematické analýzy, algebry, lineárního programování, teorie množin, diferenciální geometrie, zpracování signálu a mnoha dalších.

Maple je mnohými uživateli [17] považován za nejlepší systém pro symbolická i numerická řešení diferenciálních rovnic. Maple lze použít nejen pro řešení obyčejných diferenciálních rovnic a **okrajových úloh** (*boundary value problem*), tj. úloh splňujících **okrajové podmínky** (*boundary conditions*), ale i pro komplikované výpočty systémů **diferenciálně-algebraických rovnic** (*system of differential-algebraic equations*) [18]. Způsob zadání nelineární homogenní obyčejné diferenciální rovnice 2. řádu (vč. počátečních podmínek) je patrný z obr. 7.



obr. č. 7: Příklad zadání diferenciální rovnice 2. řádu v systému Maple

Tento software je dodáván v několika různých verzích (Academic, Personal, Professional a Student) lišících se jednak množstvím dostupných funkcí a jednak cenou, která však pro mnoho potenciálních uživatelů bývá často rozhodujícím kritériem, zda daný systém pořídit.

3.2.2 Mathematica

Wolfram Mathematica (zkráceně pouze Mathematica) je počítačový systém pro symbolické výpočty, zaměřující se především na uživatele z oblastí vědy, výzkumu a strojírenství. Tento produkt byl původně navržen Stephenem Wolframem, spoluzakladatelem a CEO společnosti Wolfram Research, která jej nadále vyvíjí.

Mathematica obsahuje přes 5.000 vestavěných funkcí, více jak 150.000 konkrétních příkladů a je schopna zpracovat přes 180 vstupních formátů dat. Mathematica je provázána s webovým prostředím **Mathematica Online**, které umožňuje uživatelům ukládat data do **cloudového úložiště** (*cloud storage*).

Mathematica je založena na specializovaném symbolickém jazyku Wolfram Language, který sdílí s webovým řešením WolframAlpha (viz kap. 3.1.1). Tento jazyk se vyznačuje vskutku neobvyklou mírou zpětné kompatibility. Kód, který případný uživatel napsal před více než 30 lety v systému Mathematica 1, je i v nejnovější verzi stále plně funkční.

Stejně jako u systému Maple, se kterým je Mathematica velmi často srovnávána [19, 20], je největší nevýhodou poměrně vysoká cena.

3.2.3 MATLAB

MATLAB (psáno velkými písmeny; zkratka z MATrix LABoratory) je názvem numerického výpočetního prostředí společnosti MathWorks a současně i názvem programovacího jazyka, jež toto prostředí využívá. MATLAB podporuje nejen operace s maticemi, jak již ostatně napovídá jeho název, nýbrž i další matematické výpočty vč. vykreslování grafů a spolupráce s částmi kódu napsanými v jiných programovacích jazycích (především C, C++, Java, Python, Fortran a C#).

MATLAB se primárně zaměřuje na numerické výpočty. Symbolické výpočty lze provádět za pomoci volitelného balíčku **Symbolic Math Toolbox** (pochopitelně za příplatek) využívající jazyka MuPAD. Dalším hojně využívaným přídavným balíčkem je **Simulink**, prostředí pro modelování, simulace a následnou analýzu dynamických systémů. Simulink využívá grafický programovací jazyk blokových schémat, často využívaný při programování **jednodeskových počítačů** (*single-board computer*) platformy Arduino [21], Raspberry Pi [22] a dalších.

MATLAB je značně rozšířen jak v automobilovém, tak v leteckém průmyslu, vč. systémů pro meziplanetární lety, využívají jej aplikace monitorující zdravotní stav pacientů i systémy pro mobilní sítě čtvrté generace.

Podobně jako Maple je nabízen ve čtyřech různých verzích (Education, Standard, Student a Home), s možností rozšíření pomocí **přídavných balíčků** (*optional toolbox*), které rozšiřují možnosti použití. Nevýhodou je vysoká náročnost na systémové prostředky, u stávající verze R2017a je to 4–6 GB prostoru na pevném disku a až 4 GB operační paměti.

3.3 Open-source desktopové aplikace

Co se týká open-source aplikací, je nabídka CAS systémů přece jen širší, než je to v případě produktů komerčních. Nemusí to však být pouze ku prospěchu věci, neboť kromě nabídky se bohužel zvyšují i rozdíly v kvalitě. Je proto třeba pečlivě zvažovat, který software zvolit.

Název	Typ licence	Dostupné z
Axiom	BSD-like	http://www.axiom-developer.org/
FriCAS	BSD-like	http://fricas.sourceforge.net/
GNU Octave	GNU GPL	https://www.gnu.org/software/octave/
Mathics	GNU GPL	https://mathics.angusgriffith.com/
Maxima	GNU GPL	http://maxima.sourceforge.net/
SageMath	GNU GPL	http://www.sagemath.org/index.html
Scilab	CeCILL	http://www.scilab.org/

tab. č. 3: Open-source desktopové aplikace

Nejznámější (a časem již prověřené) programy jsou uvedeny v tabulce č. 3. Pochopitelně se nejedná o výběr úplný, neboť dynamika vývoje softwaru je v open-source prostředí mnohem vyšší, než je zvykem u aplikací proprietárních.

3.3.1 Maxima

Maxima je odvozena z prostředí **Macsyma**, vyvíjeného na MIT[17] v letech 1968-1982 v rámci projektu MAC[18]. Tento software byl ve své době naprosto unikátní a inspiroval vznik mnoha dalších podobných aplikací, mezi nimi Maple (viz kap. 3.2.1) či Mathematica (viz kap. 3.2.2).

17 soukromá vysoká škola **Massachusetts Institute of Technology** (MIT) ležící v Cambridge nedaleko Bostonu ve státě Massachusetts (USA); založena roku 1861; společně se **Stanford University** se řadí k absolutní špičce mezi technickými univerzitami [23]

18 projekt MAC (Mathematics And Computation) byl založen roku 1963 a proslavil se hlavně úspěšným výzkumem v oblasti umělé inteligence; jeho prvním ředitelem se stal Robert Fano, italsko-americký počítačový odborník, spolutvůrce tzv. **Shannon-Fanova kódování** (způsob tvorby prefixového kódu) [24]

Maxima je výpočtovým prostředím pro vyhodnocování symbolických i numerických výrazů, jako jsou množiny, vektory, matice, polynomy, tenzory či systémy lineárních rovnic.

Maxima využívá pro grafickou reprezentaci dat freewarový nástroj **gnuplot**, který generuje zobrazení ve formě 2D či 3D grafů. Většina zdrojového kódu je napsána v jazyce **Common Lisp**, vyvinutém taktéž na MIT, držitelem Turingovy ceny[19] za rok 1971 Johnem McCarthym.

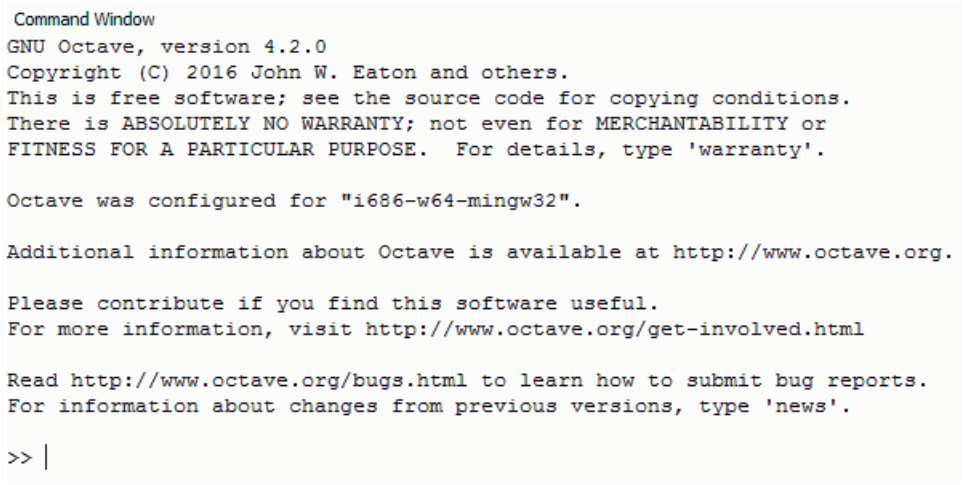
Jako **grafické uživatelské rozhraní** (*graphical user interface,* GUI) lze použít externí front-end modul **wxMaxima**, který implementuje zobrazení nabídek a dialogů pro většinu příkazů a nabízí také funkci **automatického dokončování** (*auto-completion*).

Jádro systému Maxima pro algebraické výpočty využívá i další z open-source aplikací **Euler Math Toolbox** (EMT) nabízený rovněž pod **GPL** (General Public License) licencí, který je určen výhradně pro počítače s operačním systémem Windows.

3.3.2 GNU Octave

GNU Octave je vývojové prostředí (a programovací jazyk) určený převážně pro numerické výpočty. Typicky se využívá pro operace s polynomy, vektory a maticemi, při řešení soustav lineárních i nelineárních rovnic, určování derivací, řešení diferenciálních rovnic či statistickou analýzu. **Syntaxe jazyka** připomíná jazyk MATLAB, se kterým se snaží být do značné míry kompatibilní.

Původním záměrem bylo využívat Octave pouze jako výukový prostředek k učebnici Jamese B. Rawlingse a Johna G. Ekerdta [25], zabývající se návrhem chemického reaktoru. Vývoj byl započat na jaře 1992, verze 1.0 spatřila světlo světa o rok později.



obr. č. 8: Vývojové prostředí GNU Octave 4.2.0 (Command Window)

19 nejprestižnější ocenění udělované v oblasti počítačové vědy; o jejím příjemci rozhoduje **Asociace pro výpočetní techniku** (*Association for computing machinery*); pojmenována po geniálním anglickém matematikovi Alanu M. Turingovi (1912-1954), který se za 2. světové války podílel na prolomení kódu německých šifrovacích strojů **Enigma**; mezi oceněnými lze nalézt většinu průkopníků tohoto oboru: Richard Hamming (vynálezce lineárního kódu pro detekci a opravu chyb, zvaného **Hammingův kód**) [24], Edsger W. Dijkstra (algoritmus pro nalezení nejkratší cesty v grafu, tzv. **Dijkstrův algoritmus**), Dennis Ritchie, Ken Thompson, Vinton Cerf, Robert Kahn, Martin Hellman, Whitfield Diffie, Donald E. Knuth (autor systému pro počítačovou sazbu **TEX**) či Tim Berners Lee

Prvotně bylo možné Octave ovládat pouze prostřednictvím **příkazové řádky**. Současná verze nabízí plnohodnotné grafické uživatelské rozhraní kompletního **integrovaného vývojového prostředí** (*integrated development environment*, IDE) obsahující editor kódu (viz obr. 8), vestavěný debugger[20] a interní prohlížeč dokumentace.

Funkcionalitu Octave lze jednoduše rozšířit pomocí **uživatelsky definovaných funkcí** (*user-defined function*) nebo prostřednictvím knihoven (modulů) napsaných v jazycích C, C++ či Fortranu[21]. Samotný Octave je vyvíjen v C++ (více o tomto jazyce v kap. 4.1.4).

3.3.3 Další alternativy

Kromě výše zmíněných open-source řešení Maxima a GNU Octave, které si již byly schopny za dobu své existence vytvořit širokou a stabilní uživatelskou základnu, lze vyzkoušet i další, z hlediska funkčnosti neméně pokročilé alternativní nástroje.

Axiom
Tento CAS byl původně vyvíjen světoznámou společností IBM pod názvem **Scratchpad** [26]. Ta jej na počátku 90.let 20.století odprodala britské firmě NAG, jež software přejmenovala na Axiom a snažila se s ním prosadit na trhu s komerčními matematickými výpočetními systémy. Jak se později ukázalo, tato strategie nefungovala, proto bylo následně rozhodnuto, že budou zdrojové kódy Axiomu uvolněny v rámci open-source.

Na obr. 9 je uvedeno ukázkové řešení obyčejné nehomogenní diferenciální rovnice, převzaté z uživatelské příručky. [27]

```
deq := x**3 * D(y x, x, 3) + x**2 * D(y x, x, 2) - 2 * x * D(y x, x) + 2 * y x = 2 * x**4
```

$$x^3y'''(x) + x^2y''(x) - 2xy'(x) + 2y(x) = 2x^4$$

Type: Equation Expression Integer

```
solve(deq, y, x)
```

$$[\text{particular} = \frac{x^5 - 10x^3 + 20x^2 + 4}{15x}, \text{basis} = \left[\frac{2x^3 - 3x^2 + 1}{x}, \frac{x^3 - 1}{x}, \frac{x^3 - 3x^2 - 1}{x}\right]]$$

Type: Union(Record(particular: Expression Integer,basis: List Expression Integer),...)

obr. č. 9: Řešení nehomogenní LODR 3. řádu v systému Axiom

FriCAS
FriCAS je jednou z oddělených vývojových větví (fork) Axiomu. Vznikl v průběhu roku 2007 poté co jeden z vývojářů, Waldek Hebisch, nebyl spokojen s nastavenou **vývojovou strategií** (*development strategy*) a rozhodl se pokračovat ve vývoji samostatně. Od té doby byly mnohé části kódu zásadně přepracovány, dle [28] bylo změněno asi 25 % kódu, který je z velké části implementován pomocí programovacího jazyka Spad, jež vychází ze systému Scratchpad.

---

20 specializovaný softwarový nástroj, který se používá pro ladění (hledání a identifikaci chyb) při vývoji softwaru; v dnešní době většinou součástí IDE (podrobněji o ladění v kap. 4.2.3)

21 jeden z nejstarších, do dnešních dnů používaných, programovacích jazyků; byl vyvinut již v 50.letech 20.století společností IBM; vhodný zejména pro vědecké (numerické) výpočty, je proto používán např. pro systémy predikce počasí

SageMath
SageMath je specificky vyvíjený programový balík, který využívá, zastřešuje a pomocí jazyka Cython[22] propojuje jiné volně dostupné **programové balíčky** (*package*) vč. knihoven jazyka Python pro zpracování numerických (NumPy), symbolických (SymPy) a vědeckých výpočtů (SciPy), knihovnu téhož jazyka pro vykreslování dat (matplotlib), dále knihovnu jazyka C pro aritmetické výpočty (FLINT)[23], funkce diferenciálního a integrálního počtu systému Maxima a mnoho dalších.

Scilab
Scilab historicky vychází z projektu Francouzského institutu pro výzkum výpočetní techniky IRIA[24] zvaného **Blaise** (název měl nepochybně evokovat vzpomínku na francouzského fyzika a matematika Blaise Pascala), jehož vývoj byl započat v 80. letech minulého století za účelem vytvoření vhodného nástroje pro **automatické řízení** (*automatic control*). První verze Scilabu byla uvolněna k použití počátkem roku 1994.

Scilab je, podobně jako MATLAB či GNU Octave, orientován na numerické výpočty. Ovšem díky návaznosti na výše zmíněný projekt Blaise, ze kterého pochází jádro systému, nabízí, na rozdíl od svých konkurentů, specializovaný modul pro modelování a simulace mechanických a hydraulických systémů (grafické návrhové prostředí Xcos).

22 Cython je **rozšířením** (*superset*) Pythonu, které si klade za cíl dosáhnout, prostřednictvím kódu psaného převážně v Pythonu, takové rychlosti provedení instrukcí, která je srovnatelná s implementací obdobného kódu v jazyce C; z výše zmíněného důvodu má Cython zabudován tzv. *foreign function interface*, což je rozhraní umožňující volat funkce ze sdílených knihoven jiných programovacích jazyků

23 zkratka z Fast Library for Number Theory

24 zkratka pro *Institut de recherche en informatique et en automatique,* založeného roku 1967 v obci Rocquencourt nedaleko Paříže, zodpovědného, mimo jiné, za vývoj programovacího jazyka **OCaml**; v roce 1979 byl institut přejmenován na INRIA

4 PROSTŘEDKY PRO VÝVOJ APLIKACE

Oblast **vědeckého výzkumu a vývoje** (*scientific research and development*), do které spadá i vývoj počítačových aplikací pro řešení matematických problémů, má na použité prostředky vývoje (z nichž jedním z klíčových je zvolený programovací jazyk) specifické požadavky. Zásadním se jeví zejména požadavek na vysokou přesnost výstupu matematických operací v souběhu s jejich velmi rychlým zpracováním. Z výše uvedeného, přirozeně však i z dalších méně závažných, důvodů se ve vědecké komunitě mezi často používané programovací jazyky řadí mj. Fortran[25] [29], C, C++ (viz kap. 4.1.4), Common Lisp [30, 31], Haskell, Python[16], Julia či OpenCL.

Důležitá není pochopitelně jen volba konkrétního programovacího jazyka (více o výhodách a nevýhodách jednotlivých jazyků v kap. 4.1), ale též dostupnost dalších potřebných nástrojů – **kompilátoru** (*compiler*, též překladač), **sestavovacího programu** (*linker*), **ladícího prostředí** (*debugger*)[20] (v současnosti jsou běžně všechny tyto nástroje integrovány v jednotném IDE), **systému pro kontrolu verzí** (*version control*), **profilovacího nástroje** (*profiler*) [32] apod.

4.1 Programovací jazyky

> *„There are really two kinds of languages: languages that are designed to minimize the time programmers spend programming and languages that are designed to minimize the time computers spend computing.“* [33]

Toto “hrubé” rozdělení programovacích jazyků je sice, pro správnou volbu jazyka použitého pro konkrétní aplikaci, nedostatečné, nicméně může alespoň naznačit hlavní směry, kterými se vývoj jazyků ubírá (a zároveň slouží jako jisté odlehčení tohoto odborného textu). Konkrétní rozhodovací proces bude brát v úvahu i další prvky, popsané v následujících kapitolách.

4.1.1 Paradigmata programovacích jazyků

Programovací paradigma (*programming paradigm*) je základní způsob programování, které daný jazyk podporuje. Programovací jazyk může pochopitelně podporovat více paradigmat, poté je nazýván **multiparadigmatický** (*multi-paradigm*). Mezi nejčastější paradigmata patří:

deklarativní (*declarative*)
Zaměřuje se především na to, čeho se má dosáhnout, ne na to, jakým způsobem má být tohoto cíle dosaženo. Příklady: Prolog, SQL, Wolfram Language

funkcionální (*functional*)
Považuje všechny výrazy za vyhodnocení matematických rovnic a funkcí. Pracuje výhradně s **neměnnými objekty** (*immutable object*). Příklady: Common Lisp, Erlang, Haskell, ML

---

25 během let bylo postupně vydáno několik revizí, z nichž přelomové byly verze FORTRAN 66 (prvotní standardizace organizací **ANSI**) a FORTRAN 77* (zavádí prvky strukturovaného programování); v dalších letech postupně následovaly Fortran 90, Fortran 95, Fortran 2003, v němž se poprvé objevily objektové prvky, a Fortran 2008; ANSI je zkratkou pro American National Standards Institute – americkou neziskovou organizaci pro standardizaci

* proměnné, jejichž název v této verzi Fortranu začíná na I–N, jsou implicitně deklarovány jako typ **integer**, zatímco všechny ostatní jako typ **real** (pokud programátor toto implicitní nastavení nezmění), což vedlo ke vzniku známého úsloví: *"In Fortran, GOD is REAL (unless declared INTEGER)."* [34]

generické (*generic*)
Kód algoritmů je chápán jako zcela obecný, bez ohledu na to s jakými datovými typy pracuje. Konkrétní kód algoritmu se z něj stává dosazením datového typu. Příklady: C++, D

procedurální (*procedural*)
Program je strukturován/rozdělen do jednotlivých procedur (podprogramů, funkcí), které jsou postupně volány z hlavní procedury. Každá procedura obsahuje sled programových instrukcí, které se mají vykonat. Příklady: COBOL[26], C, C++, Pascal

objektově-orientované (*object-oriented*, OOP)
Používá k popisu datových struktur tzv. objekty, jež mají simulovat skutečné objekty reálného světa. Tyto objekty mohou obsahovat jak **proměnné** (*member variable*), tak funkce, nazývané **metody** (*member function* příp. *method*). Příklady: C++, Java, Smalltalk [35]

ezoterické (*esoteric*)
Jazyky s nekonvenční syntaxí a funkcionalitou pohybující se na samotné hraně použitelnosti. Napsat v nich i jednoduché programy je nesmírně obtížné. Příklady: Brainfuck, Malbolge

4.1.2 Dělení programovacích jazyků

Programovací jazyky lze dělit dle mnoha různých kritérií. Různou kombinací těchto kritérií, kterými je daný jazyk charakterizován, je, společně s jeho paradigmatem, dáno, pro jaké typy úkolů je optimální ho zvolit.

Kompilované a interpretované jazyky
Kód **kompilovaného jazyka** (*compiled language*) je, pomocí nástroje zvaného kompilátor, přeložen přímo do **strojového kódu** (*machine code*), a proto je výsledný program zpravidla velmi rychlý. Nevýhodou je, zvláště u rozsáhlejších projektů, samotná doba kompilace, která může, dle velikosti **kódové základny** (*code base*), trvat i desítky minut či několik hodin[27].

U interpretovaného jazyka (*interpreted language*) je při každém spuštění čten, nástrojem zvaným **interpret** (*interpreter*), jeho zdrojový kód a je jím přímo i prováděn. Výhodou je, že program lze díky tomuto mechanismu spustit na jakékoli platformě, pro kterou je k dispozici interpret daného jazyka, provádění programu je však výrazně pomalejší (ve srovnání s jazyky kompilovanými).

Oba přístupy lze úspěšně kombinovat. V takovém případě je zdrojový kód programu přeložen do tzv. **bajtkódu** (*bytecode* či *p-code*), jež je jakýmsi mezistupněm mezi zdrojovým kódem a strojovým kódem, a jehož zpracování interpretem je tedy rychlejší, než v případě standardně interpretovaného jazyka. Tento přístup využívají např. jazyky Lua a Tcl.

Dalším vývojový stupněm je tzv. **JIT kompilovaný jazyk** (*just-in-time compiled language),* u něhož jsou před spuštěním programu některé (masivně využívané) části jeho kódu přeloženy přímo do strojového kódu, kdežto zbytek je ponechán ke zpracování pro interpret. Typickými představiteli tohoto přístupu jsou Java a Smalltalk.

---

26 zkratka pro *common business-oriented language* [36]*;* poprvé se objevil roku 1959 a je do dnešních dnů používán především ve finančním sektoru (bankovnictví, pojišťovnictví), i když i tam už dochází k postupnému přechodu na modernější systémy; používá velice zajímavou syntaxi založenou na užití anglického jazyka (např. `x IS GREATER THAN y`)

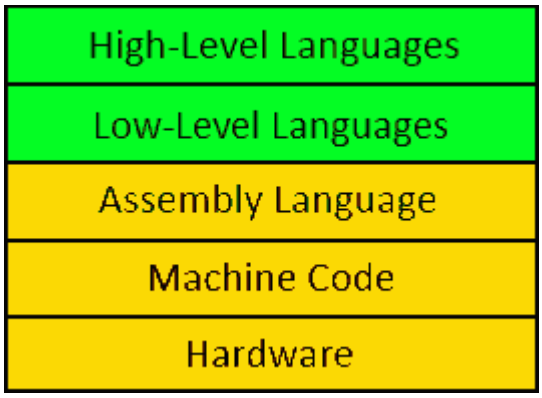
27 sestavení Qt 5.7 (více o tomto frameworku v kap. 4.2) ze zdrojového kódu, který obsahuje přes 8 milionů řádků (viz https://www.openhub.net/p/qt5/analyses/latest/languages_summary) trvá na počítači s procesorem AMD A8-4500M (1,90 GHz) a 4 GB RAM přes 8 hodin

Nízkoúrovňové (nižší) a vysokoúrovňové (vyšší) jazyky
Nízkoúrovňový jazyk (*low-level language*) je obecně mnohem blíže strojovému kódu než jazyk vysokoúrovňový a je proto vhodnější pro „kritické“ aplikace jako jsou **ovladače** (*device driver*) nebo programy s požadavkem na velmi vysokou rychlost zpracování (např. v leteckém průmyslu). Z toho logicky plyne, že nízkoúrovňové jazyky nejsou téměř nikdy interpretované.

Jazykem nejnižší úrovně je tzv. **jazyk symbolických adres** (*assembly language* či *symbolic machine code*, zvaný též assembler). [37] Jednotlivé symboly jazyka odpovídají strojovým instrukcím pro daný procesor (téměř vždy) v poměru 1:1. Jazyk je vždy specifický pro danou platformu/architekturu. Pro překlad do strojového kódu se používá program zvaný **assembler**.

Vysokoúrovňové jazyky (*high-level language*) se soustřeďují především na koncepty, jež jsou blízké lidskému způsobu myšlení, bývají proto srozumitelnější, snadněji pochopitelné a většinou umožňují i rychlejší vývoj daného softwaru. Jedním ze znaků vyšších programovacích jazyků je např. automatická **správa paměti**[28] (*memory management*).

Jak lze roztřídit programovací jazyky z hlediska „výšky“ úrovně ilustruje obr. 10.



obr. č. 10: Třídění programovacích jazyků

4.1.3 Typový systém

Typový systém[29] (*type system*) značí skupinu pravidel, která vymezují chování **proměnných** (*variable*) přiřazením určitého **datového typu** (*data type*).

Silně typovaný systém (*strong type system*) klade větší omezení na možnosti konverzí mezi jednotlivými typy proměnných. Takovýto systém by neměl umožňovat **implicitní konverze** (*implicit cast*), tj. konverze bez použití specifických příkazů daného jazyka, mezi typy, které spolu logicky nesouvisejí (např. konverze z textového řetězce na numerickou hodnotu). **Slabě typovaný systém** (*weak type system* či *loose type system*) takovéto konverze umožňuje a je jen na odpovědnosti programátora, aby byl výsledek takové operace smysluplný (příkladem slabě typovaného jazyka je např. C). Výše uvedený způsob dělení jazyků na silně typované a slabě typované není jednoznačně definováno v žádné normě a bývá v praxi někdy nesprávně zaměňováno za rozdělení na jazyky se statickou a dynamickou typovou kontrolou (viz dále).

Jazyky s **explicitním typováním** (*manifest typing*) vyžadují, aby byl při deklaraci proměnné povinně uveden její typ, zatímco jazyky, které toto nevyžadují, používají pro určení datového typu metodu zvanou **typové odvozování** (*inferred typing*). Programovací jazyk C++, který byl tradičně zástupcem první kategorie, je díky změnám ve standardu (viz kap. 4.1.4.2) nově zařazován i do kategorie druhé.

---

28 rozdíl mezi vyšším a nižším programovacím jazykem, ve vztahu ke správě paměti, krásně ilustruje poznámka autorů publikace The Annotated C++ Reference Manual:

> *„C programmers think memory management is too important to be left to the computer. Lisp programmers think memory management is too important to be left to the user.“* [38]

29 jazyky symbolických adres (assemblery) typy nepoužívají

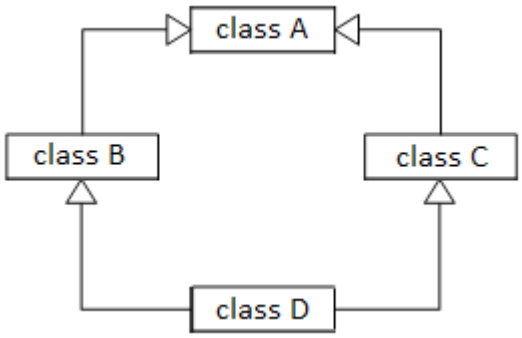
Jazyk se **statickou typovou kontrolou** (*statically type checked*) provádí kontrolu typů během kompilace programu. Odhalí sice jen chyby vyhodnotitelné během překladu, ale výhodou je, že odpadá nutnost opakování typové kontroly při každém spuštění programu, což umožňuje jeho rychlejší vykonání. Opakem je jazyk s **dynamickou typovou kontrolou** (*dynamically type checked*), který provádí kontrolu typů až za běhu programu. Dynamická kontrola probíhá pouze v použitých částech kódu (záleží na větvení programu).

4.1.4 Programovací jazyk C++

C++ [39–42] je multiparadigmatický (viz kap. 4.1.1), kompilovaný (viz kap. 4.1.2) jazyk se statickou typovou kontrolou (viz kap. 4.1.3). Byl vyvinut začátkem 80.let 20.století v AT&T Bell Labs[30] dánským počítačovým odborníkem **Bjarnem Stroustrupem**.

Charakteristické rysy jazyka C++:

- patří mezi nejrychlejší počítačové jazyky
- podporuje procedurální, generické i objektově-orientované paradigma
- k dispozici je velké množství překladačů pro mnoho různých platforem
- C++ překladače jsou schopny zpracovat většinu kódu napsanou v jazyce C
- pro C++ existuje nepřeberné množství knihoven (Boost, Loki, SFML, SDL, ad.)



obr. č. 11: Schéma diamantového problému

C++[31] vzniklo jako modifikace jazyka C. Mezi nové prvky byla zahrnuta podpora **objektově-orientovaného programování** (*object-oriented programming*, OOP), **přetěžování funkcí** (*function overloading*), **přetěžování operátorů** (*operator overloading*), nové operátory `new` a `delete` pro de/alokování paměti na **hromadě** (*heap* či *free store*) a další.

V roce 1989 bylo C++ obohaceno o další užitečné prvky: **vícenásobnou dědičnost** (*multiple inheritance*)[32], **abstraktní třídy** (*abstract class*), **statické metody** (*static member function*) a nový typ **oprávnění přístupu** (*access modifier* či *specifier*) `protected`. Později přibyly též **šablony** (*template*), **výjimky** (*exception*) a **jmenné prostory** (*namespace*), byl přidán datový typ `boolean` a nové typy konverzních operátorů (`reinterpret_cast`, `dynamic_cast`, `static_cast`, `const_cast`).

---

30 pojmenovány po vynálezci telefonu a fotofonu (přístroj přenášející zvuk pomocí paprsku světla) Alexanderu Grahamu Bellovi (1847-1922); od roku 2016 dceřinná společnost Nokie

31 ++ v názvu je odkazem na **inkrementační operátor** (*increment operator*), který navyšuje hodnotu operandu, na který je aplikován, o 1; jedná se o **unární** (*unary*) operátor

32 zda se v tomto případě skutečně jedná o užitečný prvek je námětem k diskuzi; vícenásobná dědičnost je poměrně kontroverzním nástrojem, jehož použití je třeba dobře uvážit; zásadním nedostatkem je možnost vzniku tzv. **diamantového problému** (*diamond problem*): jedná se o situaci, kdy dvě třídy B a C dědí ze společné rodičovské třídy A; pokud další třída D (jejich potomek) chce použít metodu z A, kterou B i C **překrývají** (*method overriding*) a pokud třída D neobsahuje vlastní implementaci této metody, tak se nedá určit, která verze dané metody se má použít (z nadřízené třídy B nebo z nadřízené třídy C?); tento problém ilustruje obr. 11

4.1.4.1 Standardy jazyka C++

Jazyk C++ je standardizován pracovní skupinou organizace ISO JTC1/SC22/WG21. Dosud byly vydány čtyři revize standardu, aktuálně se dokončuje příprava nové revize C++17 [43].

První standardizovaná verze C++ byla vydána v roce 1998 jako *ISO/IEC 14882:1998*. V roce 2003 byla vydána nová verze *ISO/IEC 14882:2003* opravující zejména v mezidobí objevené chyby. Jednou z hlavních změn bylo, že pro prvky vektoru, což je jeden z nejčastěji užívaných **kontejnerů** (*container*), byla nově zavedena podmínka *contiguous storage*, tj. že jednotlivé prvky vektoru musí být v paměti uloženy za sebou, neboť praxe ukázala, že takovéto chování většina programátorů logicky předpokládá. Většina kompilátorů `std::vector` takovýmto způsobem sice implementovala již dříve, standardem ale takové chování garantováno nebylo.

Další verze, před přijetím neformálně nazývaná C++0x, přinesla skutečně masivní změny, a to jak v **jádře** (*core language*), tak ve standardní knihovně (více o ní v kap. 4.1.4.3). Revize byla nakonec vydána v roce 2011 jako *ISO/IEC 14882:2011* (C++11) [44]. Zásadní resp. zajímavé změny budou detailněji popsány v kap. 4.1.4.2.

V prosinci 2014 pak byla vydána současně platná verze *ISO/IEC 14882:2014* (C++14) [45], neobsahující žádné dramatické změny, spíše jen opravy a drobná vylepšení.

4.1.4.2 Nové prvky v C++11

Tato revize standardu ISO pro jazyk C++ zavedla tak ohromné množství změn a novinek, že se dá s nadsázkou říci, že C++11 je novým programovacím jazykem. Důležité však je, že přes všechny tyto změny, zůstává veškerý kód vytvořený dřívějšími revizemi jazyka C++, plně funkční. Z důvodu nedostatku místa není pochopitelně možné všechny tyto změny podrobně probrat, uveďme tedy alespoň ty nejzajímavější:

R-value reference (*r-value reference*)
C++11 přidává nový typ reference zvaný r-value reference (označení `T &&`). Jde o referenci na **dočasnou proměnnou** (*temporary variable*), která je modifikovatelná. Toho využívá **move konstruktor** (*move constructor*), jehož použitím lze předejít zbytečnému kopírování obsahu kontejneru např. `std::vector`, který je návratovou hodnotou funkce (jedná se o dočasnou proměnnou, která bude ihned zlikvidována). Move konstruktor zkopíruje **ukazatel** (*pointer*) na interní **pole** (*array*) z dočasného objektu do nového objektu daného kontejneru (do kterého se přiřazuje výsledek funkce, jež byla volána) a nastaví hodnotu tohoto ukazatele v dočasném objektu na nulu resp. **nullptr**[33] (což by bez modifikovatelné r-value reference nebylo možné).

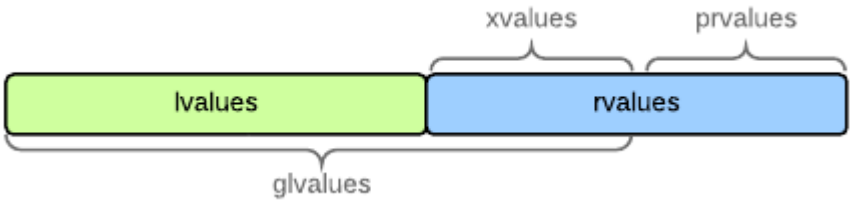
Dalším důsledkem zavedení r-value referencí je, že lze přetěžovat funkce v závislosti na tom, jestli je jako argument při jejich volání použita hodnota l-value nebo r-value (čím se tyto typy hodnot liší bude vysvětleno níže v textu). Toto je ilustrováno na následujícím příkladu:

```
// prototyp funkce pro argument typu l-value reference
void foo(T &);
// prototyp funkce pro argument typu r-value reference
void foo(T &&);
// deklarace proměnné x (l-value) a funkce y (vrací r-value)
T x;
T y();
// volání varianty funkce foo() dle typu argumentu
foo(x);     // argument je l-value; volá se foo(T &)
foo(y());   // argument je r-value: volá se foo(T &&)
```

33 nullptr – konstanta označující nulový ukazatel (další z novinek v C++11)

Rozšířený okruh pro hodnoty výrazů
Kromě tradičních l-value a r-value byly zavedeny tři nové typy nazývané gl-value, pr-value a x-value [46]. Vztahy mezi nimi názorně vyjadřuje obr. 12.

- **l-value** představuje funkci nebo objekt
- **x-value** (*eXpiring value*) představuje objekt, který je výsledkem výrazu obsahujícího r-value reference (viz výše) např. jako výsledek volání funkce, jejíž návratová hodnota je r-value reference
- **gl-value** (*generalized l-value*) je l-value nebo x-value
- **r-value** je x-value, **dočasný objekt** (*temporary object*), nebo hodnota, která není svázána s žádným objektem
- **pr-value** (*pure r-value*) je r-value, která není x-value



zdroj: josephmansfield.uk/articles/lvalue-rvalue-metaphor.html

obr. č. 12: Kategorie hodnot výrazů v C++11

Odvozování typů (*type inference*)
Definice proměnné, která je inicializována, nemusí obsahovat explicitní vyjádření jejího typu, místo něj je možno použít klíčové slovo `auto`[34].

```
const std::string proverb = "Pride comes before a fall.";
auto it = proverb.cbegin();
```

Cyklus `for` pro iteraci přes všechny prvky v daném rozmezí (*range-based `for` loop*)
Syntaxe příkazu `for` byla rozšířena, aby umožňoval snadnější způsob iterace přes všechny prvky v daném rozsahu. Lze ji využít jak pro klasická pole s pevným počtem prvků, tak pro jakýkoliv kontejner, který má definovány iterátory `begin()` a `end()`.

Lambda funkce (*lambda function*)
C++11 nově nabízí možnost vytvářet **anonymní funkce**, zvané lambda funkce [44], ve tvaru

[capture](parameters) -> return_type { function_body }, např.

```
[](int a, int b) -> int { return (a*b); }
```

Mimo výše uvedeného byly v rámci specifikace C++11 dále zavedeny **silně typované výčty** (*strongly-typed enumeration*), **externí šablony** (*extern template*), **`constexpr`** specifikátor či označení konstruktorů tříd `default` resp. `deleted`. Byly také rozšířeny možnosti použití klíčového slova `using` a **inicializačního seznamu** (*initializer list*).

34 bylo využito klíčového slova, které jazyk C++ již obsahoval, ale v praxi nebylo používáno; dle původní specifikace sloužilo k označení **automatické proměnné** tj. proměnné vytvořené na **zásobníku**, jejíž životnost je limitována programovým blokem, v němž je deklarována; vzhledem k tomu, že lokální proměnné jsou vždy automatické, nebyl důvod tento specifikátor používat (pozn. `auto` spadalo do skupiny tzv. **storage class specifier**, stejně jako `static`)

4.1.4.3 Standardní knihovna

Standard jazyka C++ je tvořen dvěma částmi: jednak jádrem, jednak standardní knihovnou. Standardní knihovna [47] je tvořena kolekcí kontejnerů (std::vector, std::map, std::multimap, std::list, std::queue, std::set ad.), **iterátorů** (*iterator*), které umožňují snadný přístup do těchto kontejnerů, a **algoritmů** (*algorithm*)[35], které podporují základní operace, jako je hledání či třídění (např. std::find, std::for_each, std::binary_search, std::stable_sort ad.).

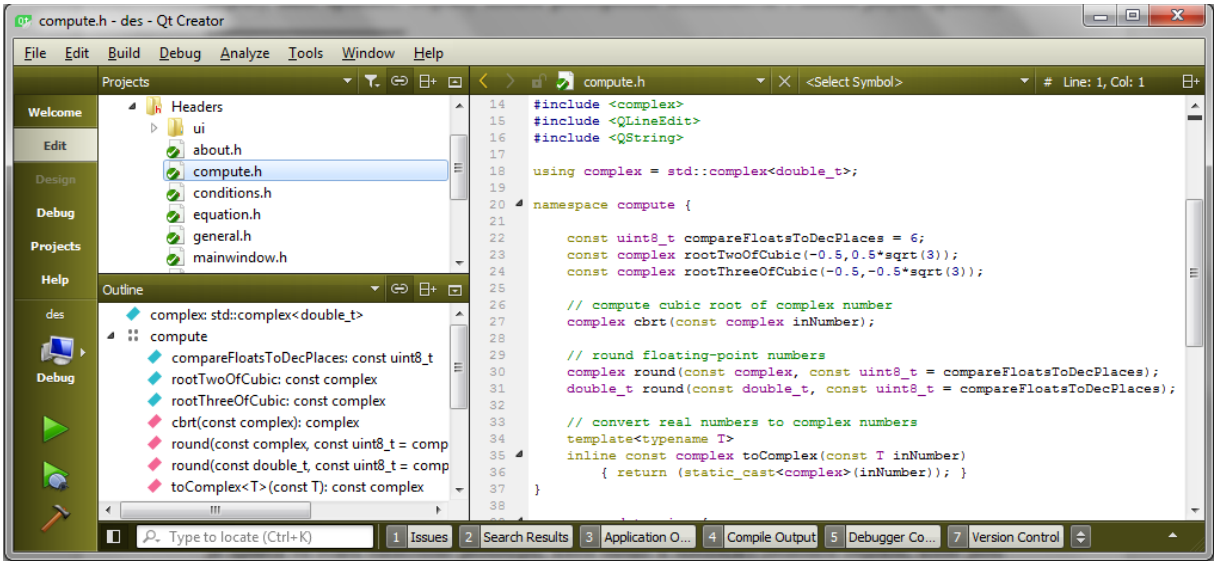
Standardní knihovna dále usnadňuje provádění **vstupně-výstupních operací** (*input-output*, zkráceně I/O) pomocí `std::iostream`, poskytuje **chytré ukazatele** (*smart pointer*) [48] pro automatickou správu paměti, nabízí podporu **regulárních výrazů** (*regular expression*) [49], obsahuje prvky podporující **vícevláknové zpracování** (*multi-threading*) a (pseudo)**generátor náhodných čísel** (*random number generator*).

Do standardní knihovny je zabudována též modifikovaná verze standardní knihovny jazyka C. Jednotlivé prvky standardní knihovny lze použít po vložení příslušného hlavičkového souboru pomocí direktivy `#include`.

4.2 Vývojový framework Qt

Qt je aplikační vývojový framework pro vývoj aplikací s grafickým uživatelským rozhraním, umožňující sestavení aplikace pro více různých platforem s využitím jediné společné kódové základny. Qt framework primárně podporuje dva programovací jazyky: multiparadigmatický jazyk C++ (viz kap. 4.1.4) a deklarativní jazyk QML[36].

V Qt frameworku [50–52] je pochopitelně možné, kromě aplikací s grafickým uživatelským rozhraním, vyvíjet i konzolové aplikace či služby využívající jen jeho jádro příp. jej využívat pouze jako nástroj pro tvorbu uživatelských rozhraní již existujících aplikací.



obr. č. 13: Vývojové prostředí Qt Creator

35 algoritmus = výraz používaný programátory v případech, kdy nechtějí (anebo ani neumí) vysvětlit, co vlastně udělali

36 **značkovací jazyk** (*markup language*) používaný při návrhu uživatelského rozhraní pro mobilní aplikace (vč. ovládání dotykové obrazovky); interně využívá **JavaScript**

Qt obsahuje širokou kolekci mnoha různých tříd, funkcí a dalších datových struktur, které se nabízí pro použití při tvorbě vlastních aplikací. K tvorbě uživatelského rozhraní obsahuje Qt nepřebernou škálu přizpůsobitelných **widgetů**. Mimo to má vestavěnou podporu pro databáze (QSql), sítě (QNetwork), rastrovou i vektorovou grafiku, zpracování zvuku a videa ad.

Hlavní komponentou Qt frameworku je vývojové prostředí **Qt Creator** (uživatelské rozhraní je zobrazeno na obr. 13), v němž jsou integrovány veškeré nástroje potřebné pro kompletní vývoj aplikace, od editace zdrojového kódu, vč. funkce automatického dokončování, přes jeho překlad a ladění, až po možnost vizuálního návrhu grafického uživatelského rozhraní pomocí **Qt Designeru**. Pro práci s dokumentací slouží nástroj **Qt Assistant**.

Qt framework je dostupný pod několika typy licencí, takže je v něm možné vyvíjet aplikace s otevřeným i uzavřeným zdrojovým kódem, pro komerční i nekomerční použití.

4.2.1 Mechanismus signálu a slotu

Pro vývoj uživatelského rozhraní, a zejména interakci s ním, je klíčový **mechanismus signálu a slotu** (*signal & slot mechanism*), který zajišťuje snadnou komunikaci mezi jednotlivými objekty dané aplikace. Objekty mohou pochopitelně komunikovat i mnoha jinými způsoby, mechanismus signálu a slotu má však tu výhodu, že je nezávislý na rozhraní jednotlivých objektů.

Princip funkce
Pokaždé, když nějaký objekt změní svůj stav, reaguje na tuto situaci vysláním signálu. Signál je zpráva ve tvaru funkčního prototypu, která slouží k notifikaci ostatních objektů, které jsou určeny jako příjemci tohoto signálu, že nastala událost, na kterou je třeba zareagovat. Vyslání signálu aktivuje sloty, což jsou většinou metody daných tříd, které signál obslouží. K určitému signálu může být přiřazeno i více slotů než jeden. Tyto funkce se vykonají v takovém pořadí, v jakém byly registrovány.

Registrace mechanismu signálu a slotu se provádí v konstruktoru widgetu, kterého se týkají, jednou ze dvou následujících metod:

```
// starší metoda – využívá makra SIGNAL a SLOT
// nevýhoda: run-time checking
connect(ui->pushButton_Clear, SIGNAL(clicked()),
            this, SLOT(clear());

// novější metoda – využívá reference
// výhoda: compile-time checking
connect(ui->pushButton_Clear, &QPushButton::clicked,
            this, &MainWindow::clear);

// novější metoda - slot implementován jako lambda funkce
connect(ui->pushButton_Clear, &QPushButton::clicked,
            this, [this]() -> void { result = 0; });
```

4.2.2 Kompilace a sestavení aplikace v Qt

K překladu zdrojového kódu vyvíjeného programu lze v Qt frameworku využít libovolného kompilátoru pro jazyk C++, který je na daném zařízení (počítači) nainstalován. V prostředí Linuxu/MacOS to nejčastěji bude LLVM[37] nebo GCC[37] [53], v operačním systému Windows pak MinGW[37](klon GCC) resp. MSVC[37].

37 nejčastěji používané kompilátory jazyka C++ (pro vývoj desktopových aplikací)

Kompilátor je program, který překládá zdrojový kód do formy proveditelných instrukcí tzv. **objektového souboru** (*object file*). Objektové soubory ještě nejsou spustitelné, jsou pouze jakýmsi mezistupněm při tvorbě **spustitelného souboru** (*executable*), neboť nejsou svázány s ostatními moduly a knihovnami, které potřebují pro svůj běh. K tomu je třeba použít linkeru, který propojí všechny sestavované moduly aplikace, převede relativní adresy jednotlivých modulů na absolutní, dodá specifický inicializační kód (pro různé platformy se liší), a tím teprve vznikne výsledný program ve spustitelném tvaru.

Práce linkeru (resp. jeho konfigurace) je založena na zpracování tzv. **projektového souboru**. Tento konfigurační textový soubor (s příponou .pro) je vytvořen buď automaticky při tvorbě nového projektu v Qt Creatoru příp. manuálně programátorem. Projektový soubor obsahuje seznam **hlavičkových souborů** (*header file*), **souborů s programovým kódem** a **zdrojových souborů** (*resource file*), seznam použitých modulů Qt, názvy a cesty výstupních binárních souborů, konfiguraci sestavovacího nástroje **qmake** apod.

4.2.2.1 Statické sestavení aplikace v systému Windows

Prostředí Qt frameworku nainstalované pomocí předpřipraveného instalačního balíčku pro OS Windows je **dynamicky linkované** (*dynamically linked*). Z toho plyne, že i aplikace vyvinuté v tomto prostředí se budou při svém běhu odkazovat na dynamické knihovny prostředí Qt a takováto aplikace, pokud nebudou potřebné knihovny distribuovány společně s ní, nebude na systému bez instalovaného Qt frameworku pracovat (i kdyby tam instalován byl, tak by ještě musel mít uživatel to štěstí, že se jedná o stejnou verzi, v jaké byl program vyvíjen)[38].

Výše uvedená situace se dá řešit dvěma způsoby. První možnost je použít nějaký softwarový nástroj k vyhledání závislostí, např. **Dependency Walker**[39], a distribuovat potřebné knihovny, jak již bylo naznačeno výše, společně s aplikací. Druhou možností je sestavit statickou verzi dané aplikace. Problém je ovšem v tom, že předinstalované (dynamicky linkované) prostředí Qt frameworku nám toto neumožňuje, je proto nutné stáhnout si z depozitáře Qt zdrojové soubory a statickou verzi Qt si sestavit vlastními silami[40].

Vlastnosti a omezení staticky linkovaných aplikací

- první omezení plyne ze systému licencování Qt; v případě open-source aplikací problém nenastává, v případě proprietárních aplikací je však možnost statického linkování výrazně omezena - je třeba se řídit licenčními podmínkami Qt
- není možné rozšíření funkčnosti aplikace za běhu pomocí pluginů, ať již vlastních či dodaných třetí stranou
- spustitelný soubor statické verze je větší, než spustitelný soubor dynamické verze, jelikož veškerý potřebný kód Qt knihoven je vnořen ve spustitelném souboru
- celková velikost aplikace je však v případě statické verze menší (pouze spustitelný soubor), neboť dynamická verze obsahuje mimo spustitelného souboru i veškeré externí dynamické knihovny, které obsahují i kód v aplikaci nepoužitý/nepotřebný

---

38 kdyby měl ještě větší štěstí, tak by se mohlo jednat i o jinou verzi Qt frameworku, přičemž odkazovaná dynamická knihovna se mezi verzí instalovanou na spouštěném systému a verzí, která byla použita pro vývoj aplikace, nezměnila; pochopitelně by záleželo také na nastavení **cesty** (*path*) k adresářům, kde jsou potřebné knihovny umístěny

39 volně ke stažení z http://www.dependencywalker.com/ (pouze pro Windows)

40 pro zkušenějšího uživatele to není zase tak moc obtížné; na webových stránkách Qt jsou k dispozici **skripty** (*script*), které provedení celé operace značně usnadňují

- statická verze aplikace nevyžaduje žádné knihovny dodané Qt či MinGW (MSVC), přesto však obsahuje reference na knihovny systému Windows (např. kernel32.dll nebo user32.dll); tyto knihovny jsou běžnou součástí OS, není třeba ověřovat, zda jsou k dispozici
- statické sestavení je doporučeno používat pouze v kombinací s tzv. **release**[41] verzí aplikace; používat ho v kombinace s **debug**[41] verzí, která obsahuje množství kódu určeného k usnadnění ladění, nedává valný smysl – snižuje to rychlost aplikace a neúměrně zvyšuje její velikost

4.2.3 Ladění aplikace v Qt

Ladění (*debugging*) zahrnuje hledání a řešení chyb, jež způsobují nefunkčnost či nekorektní chování programu. K ladění se používá specializovaný softwarový nástroj zvaný **debugger**. Účelem použití debuggeru je umožnit programátorovi snadněji identifikovat stav, v jakém se laděný software nacházel v okamžiku, kdy **havaroval** (*crashed*) příp. se u něj objevilo jiné nežádoucí chování. Zastavení debuggeru v určitém místě programového kódu lze dosáhnout nastavením tzv. **bodů přerušení** (*breakpoint*). Provádění kódu je v takovém místě zastaveno a debugger poskytne programátorovi užitečné informace o aktuálním stavu programu – zejména jaké jsou aktuální hodnoty jednotlivých **lokálních** (*local*) a **globálních** (*global*) proměnných, které proměnné se nachází v registrech, jaký je stav **zásobníku paměti** (*memory stack*), jak je kód rozdělen pro zpracování v jednotlivých **vláknech** (*thread*) apod.

Pokud program, který je spuštěný v režimu ladění, nemůže být z důvodu **programové chyby** (*programming bug*)[42] příp. **vadných dat** (*invalid data*) dále prováděn, dojde v místě kódu, kde došlo k detekování chyby, k jeho automatickému zastavení, a to bez ohledu na nastavené body přerušení. To může být způsobeno např. pokud se laděný program pokouší použít instrukce, jež **instrukční sada** (*instruction set*) daného procesoru nepodporuje nebo se pokouší o přístup do oblasti paměti, která je nedostupná či chráněná proti zápisu (read-only)[43].

Qt Creator používá **zásuvný modul** (*plugin*) pro ladění, který slouží jako **rozhraní** (*interface*) mezi jádrem Qt Creatoru a externím ladícím nástrojem. Mezi podporované debuggery patří GDB (GNU Symbolic Debugger), CDB (Microsoft Console Debugger) či LLDB (debugger projektu LLVM). Debugger plugin automaticky vybere vhodný nativní debugger z těch, které jsou na daném systému instalovány (toto lze pochopitelně manuálně změnit).

GDB běží jak na Windows, tak na mnoha Unix-like systémech (FreeBSD, Linux, MacOS) a podporuje širokou škálu programovacích jazyků, mezi jinými Ada[44], C, C++, Objective-C, D, Fortran, Go, Pascal, Rust či Modula-2.

---

41 debug verze obsahuje oproti release verzi navíc informace usnadňující ladění, zejména tzv. **ladící symboly** (*debugging symbol*); nejsou také povoleny některé z optimalizačních technik, např. **inlining** (umístění kódu funkce přímo do místa jejího volání), **loop unrolling** (nahrazení cyklu opakováním kódu) ad.

42 označení *bug* pochází z konce 40.let 20.století, kdy bylo jako příčina nefunkčnosti elektro-mechanického počítače Mark II označeno relé, ve kterém byl uvězněn mol [54]

43 pokus o zápis do oblasti paměti určené pouze pro čtení typicky končí chybou známou jako **chyba paměťové ochrany** (*segmentation fault*)

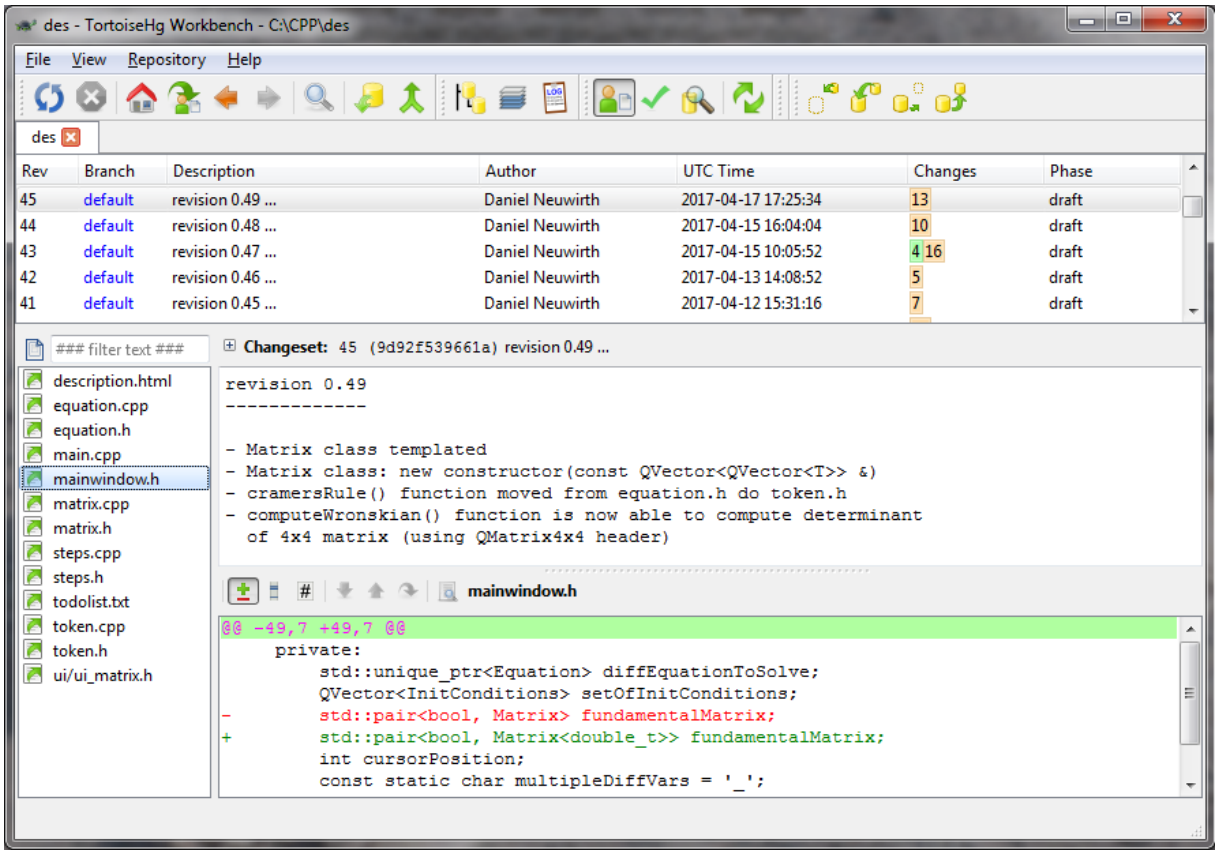
44 pojmenován po matematičce Adě Lovelace (1815–1852), dceři anglického básníka George Gordona Byrona (1788-1824), která se podílela na vývoji mechanického počítače Charlese Babbage (1791-1871), a je proto všeobecně považována za první programátorku v historii

4.3 Systém kontroly verzí Mercurial

Systém kontroly verzí (*version control system*, VCS) je software či metoda mající na starost sledování změn v programovém kódu (v širším vyjádření změn v jakýchkoli dokumentech). Jednotlivé verze jsou označovány jako revize a jsou identifikovány pomocí tzv. **čísla revize** (*revision number*). Každá revize je označena **časovým razítkem** (*timestamp*) a kódem osoby, která ji uložila, tj. provedla tzv. **commit**. Jednotlivé revize lze mezi sebou snadno porovnávat, obnovovat či slučovat.

Vývojové prostředí Qt Creator podporuje velké množství systémů kontroly verzí, mezi jinými to jsou všeobecně oblíbené služby Bazaar, CVS [55], Git, Mercurial a Subversion.

Mercurial je multiplatformní, distribuovaný systém kontroly verzí, s otevřeným zdrojovým kódem. Z větší části je implementován pomocí jazyka Python. Mercurial je systém ovládaný primárně z **příkazové řádky**[45] (*command-line,* CLI), nicméně jsou k dispozici i rozšíření resp. nadstavby nabízející grafické uživatelské rozhraní. Jedním z nejrozšířenějších je TortoiseHg[46], jehož rozhraní zvané **TortoiseHg Workbench** je zobrazeno na obr. 14.



obr. č. 14: TortoiseHg Workbench

45 program, který se stará o zpracování příkazů zadaných prostřednictvím příkazové řádky, je nazýván **interpret příkazové řádky** (*command language interpreter*, též *shell*)

46 vyvíjeno v Pythonu pomocí **PyQt**, **jazykové vazby** (*language binding*) pro Qt framework; další často využívanou jazykovou vazbou je **QtSharp** pro jazyk C#

5 IMPLEMENTACE APLIKACE

Pro vývoj aplikace bylo zvoleno prostředí Qt Creator 4.0.2, integrální součást Qt frameworku, v kombinaci s multiparadigmatickým programovacím jazykem C++, s rozsáhlým využitím nových prvků, jež byly do jazyka implementovány v rámci revize C++11. Jako kompilátor byl použit MinGW 5.3.0, jako debugger GDB 7.10.1. Z uvedeného je patrné, že vývoj probíhal pod operačním systémem Microsoft Windows. Z dalších podpůrných nástrojů byly použity systém kontroly verzí Mercurial 3.8.4 (s nadstavbou TortoiseHg) a textový editor **Notepad++**.

Mimo vlastních knihoven Qt frameworku nebylo nezbytné použití žádných jiných (externích) knihoven či výpočtových modulů třetích stran, což pochopitelně značně usnadnilo sestavení aplikace. Ikony použité v programu pochází od autorského týmu **Oxygen Team**, jejich použití je, v rámci licence **LGPL** (Lesser General Public License), možné pro komerční i nekomerční projekty. Ikony vytvořené tímto francouzským týmem jsou k dispozici ke stažení na webové adrese: http://www.iconarchive.com/artist/oxygen-icons.org.html.

Aktuální verze aplikace **0.55** (z 20. dubna 2017) podporuje výpočet lineárních (kap. 2.3.3.2) homogenních (kap. 2.3.3.4) obyčejných (kap. 2.3.3.1) diferenciálních rovnic s konstantními koeficienty (kap. 2.3.3.3), druhého, třetího, a s jistými omezeními i čtvrtého řádu (podrobnosti viz kap. 5.1.2.3), a to vč. výpočtu konstant c_1–c_n v případě zadání počátečních podmínek. Jiné typy diferenciálních rovnic momentálně nejsou v této aplikaci řešitelné, nicméně je alespoň schopna je správně identifikovat (viz kap. 5.3).

5.1 Jádro aplikace

Jádrem aplikace je **třída** (*class*) `Equation`, definovaná v hlavičkovém souboru equation.h, která zahrnuje jednak členské proměnné, jež popisují parametry dané diferenciální rovnice, a jednak skupinu funkcí, z části určených pro jednotlivé kroky, z nichž sestává výpočet, z části sloužících pro rozbor dané rovnice a určení jejího typu.

Pro určení homogenního řešení zadané diferenciální rovnice je klíčová její **charakteristická rovnice** reprezentovaná třídou `CharEq`, definovanou v hlavičkovém souboru general.h (více v kap. 5.1.2). K dalším (pomocným) třídám patří `Roots` (pro uložení kořenů charakteristické rovnice), `Derivatives` (pro uložení derivací), `InitConditions` (počáteční podmínky) a `Token` (rozdělení zadané rovnice na jednotlivé prvky).

V dalším textu budou popsány pouze ty programové konstrukce, které jsou pro běh aplikace nezbytné nebo nějakým zajímavým či méně obvyklým způsobem využívají možností daných zvoleným programovacím jazykem. Kompletní zdrojový kód je v elektronické příloze.

5.1.1 Třída Equation

Třída `Equation` používá pro uložení konkrétně zadané diferenciální rovnice, a dalších údajů s ní logicky svázaných, mj. v následujících odstavcích popsané členské proměnné:

Počáteční podmínky:
Množina počátečních podmínek (každá ze zadaných podmínek je určena jednak hodnotou nezávislé, a jednak hodnotou závislé proměnné v daném bodě) je uložena ve vektoru objektů k tomu určené třídy `InitConditions`, který je navíc svázán s proměnnou typu `boolean`, která určuje, zda se mají zadané počáteční podmínky při výpočtu rovnice skutečně aplikovat – pro uložení této dvojice údajů je využit kontejner standardní knihovny `std::pair`, jež je definován v hlavičkovém souboru `<utility>`.

```
std::pair<bool,QVector<InitConditions>>[47] initialConditions;
```

Charakteristická rovnice:
Charakteristická rovnice je uložena jako objekt třídy `CharEq` (více v kap. 5.1.2).

```
CharEq characteristicEquation;
```

Tokeny:
Jednotlivé elementy (subvýrazy) zadané diferenciální rovnice jsou, z důvodu usnadění určení typu rovnice, i kvůli usnadnění následných početních operací (derivování ad.), během procesu zvaného **tokenizace** (*tokenization*) od sebe odděleny a samostatně uloženy ve vektoru objektů třídy `Token`, která tyto elementy reprezentuje.

```
QVector<Token> tokens;
```

Typ rovnice:
K uložení typu rovnice jsou potřeba jen 4 paměťové bity[48] – rovnice je lineární či nelineární, obyčejná či parciální, homogenní či nehomogenní a s konst. nebo proměnnými koeficienty (druhy diferenciálních rovnic viz kap. 2.3.3), je proto užito typu `std::bitset` definovaného v hlavičkovém souboru `<bitset>`. Řád rovnice je uložen zvlášť (jako `uint8_t`).

```
std::bitset<4> type;
```

Nastavení odpovídající hodnoty 1.bitu (tohoto `bitset`u) v závislosti na tom, jestli je zadaná diferenciální rovnice lineární či nelineární, je provedeno pomocí členské metody `set()`:

```
// is linear?
    type.set(0, isLinear());
    if (type.test(LINEAR))
        description += "linear ";
    else {
        description += "nonlinear ";
        canBeSolved = false;
    }
```

Ke klíčovým členským metodám patří funkce `generalSolution()` zastřešující jednotlivé postupné kroky vedoucí k výpočtu zadané diferenciální rovnice, `computeDerivatives()` pro symbolické i numerické řešení derivací (jež jsou potřeba v případě zadání počátečních podmínek k určení aktuálního řešení), `computeWronskian()` sloužící k určení determinantu fundamentální matice, `determineValuesOfConstants()` k výpočtu hodnot jednotlivých konstant použitím tzv. Cramerova pravidla, a analyzační funkce `parseEquation()`.

5.1.1.1 Funkce generalSolution

Z důvodu následného zpracování, a správného zobrazení výsledku, je třeba vypočítané kořeny charakteristické rovnice nejprve setřídit. Je použito funkce `std::stable_sort`, definované v hlavičkovém souboru `<algorithm>`, která na rozdíl od `std::sort`, zaručuje zachování pořadí stejných prvků (to je nezbytné v případě násobných kořenů). Třídící funkce používá vlastní **komparační objekty**.

---

47 díky úpravě pravidel pro **syntaktickou analýzu** (*parsing*) v C++11 již není nadále nutné používat mezi dvěma pravými ostrými závorkami oddělovací mezeru [44]; před příchodem C++11 byl totiž výraz >> zpracován analyzátorem jako **operátor bitového posunu doprava** (*right shift operator*) a byla hlášena **syntaktická chyba** (*syntax error*)

48 bit = **bi**nary digi**t** – základní jednotka informace

```
// declare non_const iterators
auto from = characteristicEquation.getRootsModifiable().begin();
auto to = characteristicEquation.getRootsModifiable().end();

// phase 1: move real roots towards the top of the vector
// and complex roots to the bottom of the vector
sortRealRootsOverComplex customSortPhaseOne;
std::stable_sort(from, to, customSortPhaseOne);

// phase 2: sort real roots (ie. only part of the vector)
to = from + characteristicEquation.noOfRealRoots();
sortRealRootsInAscOrder customSortPhaseTwo;
std::stable_sort(from, to, customSortPhaseTwo);
```

Jako datový typ komparačního objektu se používá **struktura** (*struct*), která porovnává dané elementy pomocí přetíženého operátoru `()`:

```
struct sortRealRootsOverComplex {

    bool operator()(const Roots inFirst, const Roots inSecond)
        { return (std::abs(inFirst.getImagPartOfRoot())) <
                  std::abs(inSecond.getImagPartOfRoot()); }
};
```

Funkci pro výpočet absolutní hodnoty (`std::abs`) je možno použít po vložení hlavičkového souboru `<cmath>`. Je však třeba důsledně dbát na uvedení jmenného prostoru `std`, protože hlavičkový soubor `<cmath>` obsahuje dvě různé funkce pro výpočet absolutní hodnoty. První z nich je převzata z hlavičkového souboru `<stdlib.h>`, tudíž z té části standardní knihovny, která má původ v jazyce C, a má **návratovou hodnotu** (*return value*) typu **int**. [56] Právě tato funkce je užita v případě, kdy není uveden specifikátor `std`, který uvozuje druhou, „originál“ C++, funkci, která má návratovou hodnotu **float**. [57]

5.1.1.2 Funkce computeDerivatives

Pro procházení jednotlivých kořenů charakteristické rovnice, u nichž je třeba určit derivace, lze s výhodou využít specifikátoru `auto` (nově v C++11, viz kap. 4.1.4.2):

```
for (auto & it: characteristicEquation.getRootsModifiable()) { ... }
```

Programový kód je větven v závislosti na tom, zda jsou kořeny charakteristické rovnice reálné anebo komplexní, jestli jsou násobné, příp. zda je reálná část komplexního kořenu nulová. Pro potřeby větvení jsou porovnávány zaokrouhlené[49] hodnoty jednotlivých kořenů (resp. jejich reálné a imaginární části), čímž se předchází nesprávnému vyhodnocení podmínky z důvodu, jakým způsobem jsou čísla s pohyblivou řádovou čárkou uložena v paměti počítače[50].

```
// root is REAL
if (compute::round(real) != 0 && compute::round(imag) == 0) { ... }
```

---

49 zaokrouhlené hodnoty se pochopitelně použijí pouze během srovnávací operace; skutečné hodnoty těchto proměnných použité ve výpočtech zůstávají uložené s přesností dle použitého datového typu (`double_t` resp. `long double`)

50 **číslo s pohyblivou řádovou čárkou** (*floating-point number*) lze v paměti počítače uložit s naprostou přesností pouze v případě, že jej lze vyjádřit ve tvaru $k\,2^n$, $k,n\in\mathbb{Z}$; je to dáno použitou binární reprezentací čísel; v ostatních případech dochází k určité drobné nepřesnosti, kterou je však možno ve většině případů zanedbat (ne však při testu na rovnost!)

Funkce pro zaokrouhlení je implementována následujícím způsobem:

```
double_t compute::round(const double_t inNumber,
                        const uint8_t inDecPlaces) {
    const long int modifier = std::pow(10,inDecPlaces);
    double_t roundedNumber = std::round(inNumber*modifier);
    roundedNumber /= modifier;
    return (roundedNumber);
}
```

U komplexních kořenů je třeba při určování derivací pracovat s goniometrickými funkcemi[51]. Pro tento případ se hodí použít **ukazatele na funkci** (*function pointer*). [58]

```
double_t (*gonFunctionFirstPart)(double_t) = &(std::sin);
double_t (*gonFunctionSecondPart)(double_t) = &(std::cos);
```

Tyto ukazatele[52] jsou při symbolickém výpočtu derivace (v rámci větvení programu je použito pouze pro komplexní kořeny s nulovou reálnou částí) použity jako argument volání funkce `evalDerivativeImagPart()`, přičemž jsou u každé druhé derivace prohozeny s pomocí `std::swap` (tato funkce je definována v hlavičkového souboru `<algorithm>`).

```
if (row % 2 == 0)
    std::swap(gonFunctionFirstPart, gonFunctionSecondPart);
```

V případech, kdy se mezi řešeními dané charakteristické rovnice vyskytují komplexní kořeny, které mají nenulovou reálnou část, je dvojice řešení diferenciální rovnice, určená příslušnou dvojicí komplexních sdružených kořenů této ch.r., dána součinem konstanty s exponenciální funkcí a goniometrickou funkcí *sin(x)* resp. *cos(x)*.

$$y_k = c_k e^{ax} \cos(bx), \quad y_{k+1} = c_{k+1} e^{ax} \sin(bx) \tag{29}$$

Pro druhou a každou další derivaci obecného řešení vzniklého aplikací principu superpozice na (29), pak platí

$$y^{(m)} = koef_1 e^{ax} \sin(bx) + koef_2 e^{ax} \cos(bx), \tag{30}$$

přičemž pro hodnoty koeficientů $koef_1$ a $koef_2$ lze odvodit, že

$$koef_1 = koef_2^{(m-1)} b + koef_1^{(m-1)} a \quad \text{a} \tag{31}$$

$$koef_2 = koef_2^{(m-1)} a - koef_1^{(m-1)} b, \tag{32}$$

kde $^{(m-1)}$ značí stav koeficientu z předchozího výpočetního kroku, což se odráží v kódu takto:

```
coeff1 = coeff2 * imag + coeff1 * real;
coeff2 = coeff2 * real - lastCoeff1 * imag;
```

5.1.1.3 Funkce computeWronskian

Wronskián je determinantem fundamentální matice (více v kap. 2.3.5). Výpočet Wronskiánu zastřešuje funkce `computeWronskian()`, pro matice druhého nebo třetího řádu volá funkci `determinant3()`, jejíž definice se nachází v hlavičkovém souboru token.h.

---

51 nejčastěji používanými goniometrickými funkcemi jsou sinus (*sin*), kosinus (*cos*), tangens (*tg* či *tan*) příp. kotangens (*cotg*); poměrně málo známé jsou sekans (*sec*) a kosekans (*cosec*)

52 někteří autoři preferují výraz **směrník** [59]

```
template <class T>
const T Matrix<T>::determinant3() const {

    T determinant = 0;
    // run only once for 2x2 matrix (determinant = ad - bc)
    // run three times for 3x3 matrix (computed using Sarrus' rule)
    for (uint8_t i = 0; i < size()-(size()%3)/2; ++i) {
        T partToAdd = 1, partToSubtract = 1;
        for (int j = 0; j < size(); ++j) {
            partToAdd *= matrix[j % size()]
                                [(j + i) % size()];
            partToSubtract *= matrix[j % size()]
                                    [(size()-j-1 + i) % size()];
        }
        determinant += partToAdd - partToSubtract;
    }
    return (determinant);
}

template class Matrix<double_t>;
template class Matrix<int>;
```

Složitější situace nastává u matic čtvrtého řádu. Pro výpočet Wronskiánu je v tomto případě použito metody ze třídy `QMatrix4x4` (součást Qt frameworku). Tato třída, bohužel, může být inicializována pouze pomocí klasického pole (resp. ukazatele na toto pole). Naplnění tohoto pole hodnotami řeší níže uvedený kód – matice je vytvářena po sloupcích (tzv. *column-major order*), protože je to programátorsky jednodušší, než po řádcích (tzv. *row-major order*). Tento postup nemá vliv na výpočet determinantu, jelikož platí, že: **determinant matice A je roven determinantu matice k ní transponované A^T**.

```
float values[16]; float * p = values;

for (uint8_t column = 0; column < fMatrix.size(); ++column) {

    // first row of fundamental matrix = individual roots
    *p++ = fMatrix.at(0,column)
         = characteristicEquation.getRoots().
           at(column).getNumValue();

    // other rows of individual matrix = derivatives of roots
    for (uint8_t row = 1; row < fMatrix.size(); ++row)
        *p++ = fMatrix.at(row,column)
             = characteristicEquation.getRoots().
               at(column).evalDerivativeAtInit(row-1);
}
```

5.1.1.4 Funkce determineValuesOfConstants

V případě, že jsou zadány počáteční podmínky, je třeba určit numerické hodnoty jednotlivých konstant c_1–c_n. To lze provést dvěma způsoby – buď algebraickým řešením soustavy rovnic nebo jako podíly determinantů matic zkonstruovaných dle kap. 2.3.6. Vzhledem k tomu, že ve jmenovateli tohoto podílu je determinant fundamentální matice tj. Wronskián, jehož hodnota je již známa (viz kap. 5.1.1.3), je použití druhé metody výhodnější. Matice pro určení druhého determinantu (v čitateli) se vytvoří jednoduchou záměnou jednoho ze sloupců, čehož využívá i následující implementace funkce `determineValuesOfConstants()`.

```
// repeat "number-of-constants" times
for (uint8_t constantNo = 0; constantNo < inConstants.size();
     ++constantNo) {

    float * p = values + constantNo * inConstants.size();
    // constantNo-th column contains function values
    // at initial points
    for (uint8_t i = 0; i < matrix.size(); ++i)
        *p++ = matrix.at(i,constantNo)
             = initialConditions.second.at(i).getFunctionValue();
    p = values;
    for (uint8_t column = 0; column < matrix.size(); ++column) {

        // constantNo-th column is already filled in (see above)
        if (column == constantNo)
            { p+=4; continue; }
        // first row (from second column onwards) contains
        // function values of individual roots at initial points
        *p++ = matrix.at(0,column)
             = characteristicEquation.getRoots().
               at(column).getNumValue();
        // other rows (from second column onwards) contain
        // values of individual root's derivatives at initial points
        for (uint8_t row = 1; row < matrix.size(); ++row)
            *p++ = matrix.at(row,column)
                 = characteristicEquation.getRoots().
                   at(column).evalDerivativeAtInit(row-1);
    }
}
```

Vlastní aplikace Cramerova pravidla je poté už velmi jednoduchá:

```
inConstants[constantNo] = determinant / wronskian;
```

5.1.2 Třída CharEq

Třída `CharEq` je určena pro práci s charakteristickou rovnicí. Pro uložení jejích kořenů slouží vektor objektů třídy `Roots` (jejich počet je dán nejvyšší mocninou neznámé proměnné):

```
QVector<Roots> rootsOfCharEquation;
```

Třída `Roots` neukládá jen samotné kořeny ch.r.[53] (ve formě komplexních čísel), ale i hodnoty z toho vycházejících řešení d.r.[53] v bodě daném počátečními podmínkami a také symbolické i numerické (opět v bodech daných počátečními podmínkami) hodnoty jednotlivých derivací.

```
complex root;
double_t rootNumValueAtInit;
QVector<Derivatives> derivatives;
```

Pro výpočet kořenů charakteristické rovnice je, s ohledem na to, o jaký typ rovnice se jedná, tj. zda o rovnici kvadratickou (2.řádu), kubickou (3.řádu) či kvartickou (4.řádu), volána jedna z následujících funkcí.

---

53 ch.r. = charakteristická rovnice, d.r. = diferenciální rovnice; v tomto případě je nezbytné důsledně rozlišovat, o jaké řešení se jedná; např. kořen (řešení) ch.r. $y=2+5i$ odpovídá řešení d.r. $y_j=c_j e^{2x}\cos(5x)$, kořen ch.r. $y=3$ odpovídá řešení d.r. $y_j=c_j e^{3x}$

```
const QString rootsOfQuadraticEquation(const QVector<Token> &,
    const initConditionsStruct &, const equationDegree = QUADRATIC);

const QString rootsOfCubicEquation(const QVector<Token> &,
    const initConditionsStruct &);[54]

const QString rootsOfQuarticEquation(
    const QVector<Token> &, const initConditionsStruct &, bool &);
```

5.1.2.1 Funkce rootsOfQuadraticEquation

U rovnic kvadratických je výpočet kořenů velmi jednoduchý, je použito známého vztahu:

$$y_{1,2}=\frac{-b\pm\sqrt{b^2-4ac}}{2a} \tag{33}$$

5.1.2.2 Funkce rootsOfCubicEquation

U rovnic kubických [60] je situace o poznání komplikovanější.

Pokud je její **determinant** (*determinant*) Δ

$$\Delta=18abcd-4b^3d+b^2c^2-4ac^3-27a^2d^2 \tag{34}$$

nulový, obsahuje řešení rovnice reálné násobné kořeny.

V případě, že je kromě (34) nulový i determinant Δ_0

$$\Delta_0=b^2-3ac\,, \tag{35}$$

má rovnice jeden trojnásobný reálný kořen ve tvaru

$$y_{1,2,3}=-\frac{b}{3a}\,. \tag{36}$$

Pokud je (35) nenulový, má rovnice jeden dvojnásobný reálný kořen ve tvaru

$$y_{1,2}=-\frac{9ad-bc}{2\Delta_0} \tag{37}$$

a jeden jednoduchý reálný kořen ve tvaru

$$y_3=\frac{4abc-9a^2d-b^3}{a\Delta_0}\,. \tag{38}$$

Ve všech ostatních případech má jeden z kořenů kubické rovnice tvar

$$y_1=-\frac{1}{3a}\left(b+C+\frac{\Delta_0}{C}\right), \tag{39}$$

přičemž hodnota C je rovna výrazu

$$C=\sqrt[3]{\frac{\Delta_1\pm\sqrt{\Delta_1^2-4\Delta_0^3}}{2}}\,. \tag{40}$$

54 kompletní kód funkce `rootsOfCubicEquation()` je uveden v příloze B; útržky kódu s vysvětlivkami a popisem použitého matematického aparátu jsou v kap. 5.1.2.2

Determinant Δ_1 v (40) se určí následujícím způsobem:

$$\Delta_1 = 2b^3 - 9abc + 27a^2d \tag{41}$$

Zbývající dva kořeny této rovnice lze poté spočítat s využitím C takto:

$$y_2 = \left(-\frac{1}{2} + \frac{1}{2}i\sqrt{3}\right)C \tag{42}$$

$$y_3 = \left(-\frac{1}{2} - \frac{1}{2}i\sqrt{3}\right)C \tag{43}$$

Přestože ve (40) lze použít pro výpočet kořene libovolné znaménko před výrazem

$$\sqrt{\Delta_1^2 - 4\Delta_0^3}, \tag{44}$$

determinant Δ_1 a (44) se nesmí navzájem vyrušit, což je v kódu ošetřeno takto:

```
// firstly calculate only part of C expression (that is under sqrt)
double_t sqrtPart = std::sqrt(-temp1.real()*discriminant.real());

// round floating point numbers before comparing them
if (compute::round(disD1.real()) == compute::round(sqrtPart))

    // expressions in the numerator part of the fraction under cbrt
    // ie. discriminant D1 and sqrtPart must not cancel each other
    C = compute::toComplex(std::cbrt(((disD1.real()+sqrtPart)/2)));
else
    C = compute::toComplex(std::cbrt(((disD1.real()-sqrtPart)/2)));
```

Další komplikace nastává v případě třech různých reálných kořenů. Jejich výpočet s využitím (39) totiž probíhá v oboru komplexních čísel, jež se sice v průběhu tohoto výpočtu navzájem vyruší, nicméně i tak je pochopitelně nutné použít funkci pro výpočet třetí odmocniny, jež je schopna přijímat argumenty právě ve tvaru komplexních čísel. Standardní knihovna jazyka C++ bohužel takovouto funkci nenabízí. Z tohoto důvodu, a také vzhledem k tomu, že použití externí knihovny, byť i nepříliš objemné, by bylo v tomto případě značně neefektivní[55], bylo nakonec implementováno vlastní řešení, které využívá možnosti přepisu komplexního čísla do **trigonometrické formy** [61]:

```
complex compute::cbrt(const complex inNumber) {

    double_t a = inNumber.real();
    double_t b = inNumber.imag();
    double_t r = std::sqrt(a*a + b*b);
    double_t fi = std::atan(b/a);

    a = std::cbrt(r) * std::cos(fi/3);
    b = std::cbrt(r) * std::sin(fi/3);

    // quadrant adjustment
    if (inNumber.real() < 0)
        a = -a;

    complex result(a,b);
    return result;
}
```

55 **výpočtová knihovna** (*numerical library*) typicky obsahuje desítky až stovky funkcí

5.1.2.3 Funkce rootsOfQuarticEquation

Vzorec pro obecné řešení kvartické rovnice (neboli rovnice 4. řádu) je natolik komplikovaný, že ho není možné, vzhledem k rozsahu této práce, zde uvést.[56] Popisovaná aplikace jej stejně nepoužívá, schopnost řešit kvartické rovnice je v její aktuální verzi omezena na dva speciální případy, tzv. **bikvadratickou rovnici** (*biquadratic equation*)

$$y = a_0 x^4 + a_2 x^2 + a_4 , \tag{45}$$

u které jsou koeficienty u lichých mocnin hledané proměnné nulové, a tudíž ji lze za použití **substituce** převést na rovnici 2. řádu

```
if (compute::round(b.real())== 0 && compute::round(d.real())== 0) {

    // biquadratic
    const QString equationType =
      rootsOfQuadraticEquation(inTokens, inConditions, BIQUADRATIC);
    return equationType;
}
```

a na případ, kdy je řešení tvořeno čtyřnásobným reálným kořenem ve tvaru

$$y_{1-4} = -\frac{b}{4a} , \tag{46}$$

což je v programovém kódu řešeno následovně:

```
const complex C4 = compute::toComplex(4);

if (compute::round(disDelta0.real()) == 0 &&
    compute::round(disD.real()) == 0) {

    // all four roots are equal (quadruple root)
    const complex root = -b / (C4*a);
    for (uint8_t i = 0; i < 4; ++i) {

        // even though all four roots are equal, newRoot must be
        // re-initialized each time (its numerical value changes)
        const Roots newRoot = solution(root, inConditions);
        rootsOfCharEquation.push_back(newRoot);
    }
}
```

Pro rovnice vyšších řádů (pátého, šestého ad.) neexistuje obecné **algebraické řešení** tj. řešení, k němuž lze dospět pomocí **konečných vzorců s odmocninami** (*solution in radicals*). Důkaz tohoto tvrzení byl podán norským matematikem Nielsem Henrikem Abelem (1802-1829), jež rozpracoval předchozí myšlenku italského učence a filosofa Paola Ruffiniho. Všeobecně je tento důkaz znám jako **Abelův-Ruffiniho teorém**. [63]

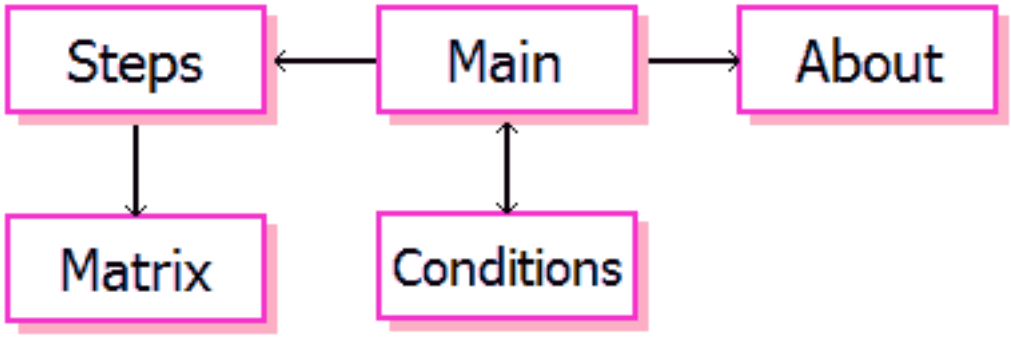
---

56 volná parafráze na proslulého francouzského matematika Pierra de Fermata (1601-1665), jenž na okraj výtisku knihy *Aritmetika* (autorem je řecký matematik Diofantos z Alexandrie), napsal poznámku týkající se důkazu o pravdivosti slavné **Velké Frematovy věty** (*Fermat's last theorem*): „Objevil jsem opravdu tak podivuhodný důkaz, že tento okraj je příliš malý, aby se do něj vešel.“ [62]; Fermat však takový důkaz pravděpodobně nikdy nenalezl, Velkou Fermatovu větu se podařilo dokázat až roku 1994 britskému matematikovi Andrew Wilesovi

5.2 Uživatelské rozhraní

Uživatelské rozhraní aplikace je tvořeno několika **okny** (*window*), která si vzájemně předávají informace, jak je naznačeno na obr. 15. Hlavní okno (objekt třídy `MainWindow`) zpracovává vstupní data tj. zadanou diferenciální rovnici, provádí její předběžnou analýzu (*pre-parsing*), jenž slouží k ověření korektnosti vstupních dat, umožňuje zadání resp. povolení aplikovat počáteční podmínky (objekt třídy `ConditionsWindow`) a zobrazuje konečný výsledek.

K zobrazení kompletního podrobného postupu výpočtu slouží dialogové okno Steps (objekt třídy `StepsWindow`), z něhož je možné vyvolat další navazující dialog sloužící pro zobrazení fundamentální matice (objekt třídy `MatrixWindow`). Zobrazení základních údajů o programu nabízí dialogové okno About (objekt třídy `AboutWindow`).



obr. č. 15: Struktura uživatelského rozhraní aplikace (šipky naznačují směr komunikace)

5.2.1 Hlavní okno aplikace

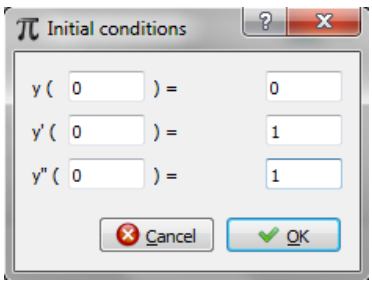
Hlavní okno aplikace, zobrazené na obr. 17, je reprezentováno objektem třídy `MainWindow`, která je **potomkem** (*child*) nadřízené **rodičovské** (*parent*) třídy `QMainWindow`. Tato třída je jednou z konstrukcí dodaných Qt frameworkem. Úkolem hlavního okna je prvotní zpracování a analýza zadané diferenciální rovnice. Paměť pro uložení těchto dat je dynamicky alokována prostřednictvím tzv. **chytrého ukazatele**[57].

```
std::unique_ptr<Equation> diffEquationToSolve;
```

Pro uložení počátečních podmínek (jsou-li uživatelem zadány) slouží vektor objektů k tomu určené třídy:

```
QVector<InitConditions> setOfInitConditions;
```

Počáteční podmínky se zadávají pomocí dialogu zobrazeného na obr. 16. Dialogové okno, resp. počet jeho položek, je tvořené dynamicky na základě řádu zadané diferenciální rovnice, neboť právě tomu odpovídá počet potřebných počátečních podmínek.



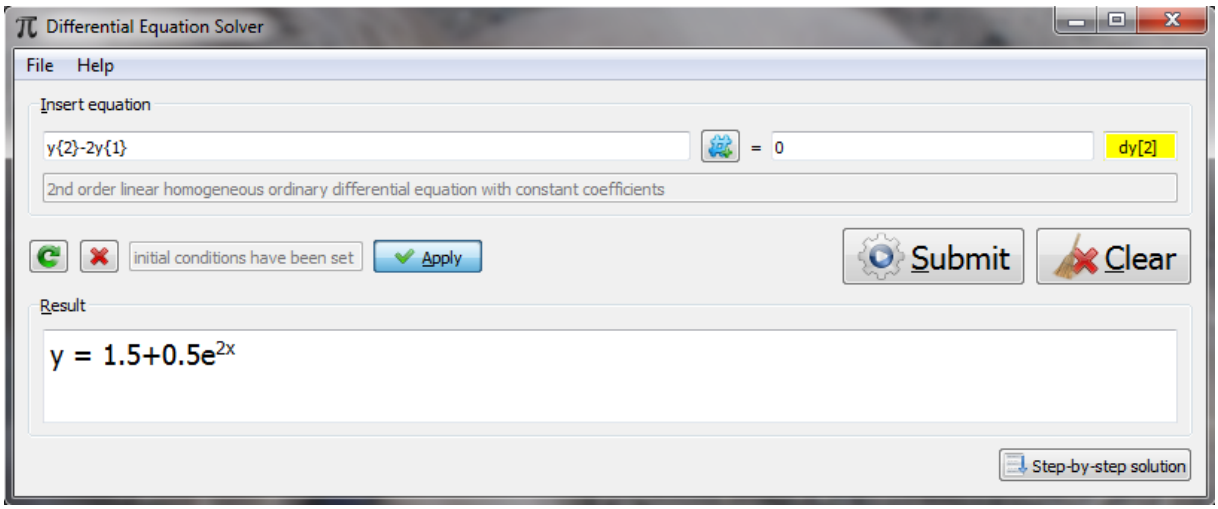
obr č.16: Počáteční podmínky

57 použití chytrých ukazatelů je zpřístupněno po vložení hlavičkového souboru `<memory>`; jejich výhodou je, že se programátor nemusí starat o dealokaci paměti takto deklarovaných proměnných; existují tři druhy chytrých ukazatelů: `unique_ptr`, `shared_ptr` a `weak_ptr`; v minulosti používaný typ (třída) `auto_ptr` je počínaje revizí C++11 považován za zastaralý (standard označuje takové prvky jazyka jako *deprecated*) a jeho použití v nově vyvíjeném softwaru se nedoporučuje

Pro uložení hodnot potřebných k výpočtu fundamentální matice pak slouží:

```
std::pair<bool, Matrix<double_t>> fundamentalMatrix;
```

Analýza zadaného výrazu (rovnice) spočívá, mimo jiné, v kontrole na přítomnost povolených a nepovolených znaků, k čemuž je využíváno regulárních výrazů. Ty jsou podporovány jak standardní knihovnou (hlavičkový soubor `<regex>`), tak Qt frameworkem (třída `QRegExp` resp. novější implementace `QRegularExpression`). Regulární výrazy v Qt jsou založeny na syntaxi jazyka **Perl**, vyvinutého americkým programátorem Larry Wallem.



obr. č. 17: Hlavní okno aplikace (MainWindow) v systému Windows 7

Seznam povolených znaků je uložen jako textový řetězec ve statické[58] proměnné:

```
const QString MainWindow::allowedSpecialCharacters = "+-^. ";
```

5.2.1.1 Funkce checkForSyntaxErrors

Tato funkce testuje, zda vložený text (diferenciální rovnice) neobsahuje zakázané znaky, zda se **proměnná, dle které se derivuje** (*variable we are differentiating with respect to*) nachází i na pravé straně rovnice (nutno převést na levou stranu), a také kontroluje umístění **složených závorek** (*brace* či *curly bracket)*, u nichž musí mj. každé otevírací závorce odpovídat jedna uzavírací, jak ilustruje následující výpis z kódu (jedná se o jednotlivé útržky kódu použité pro ilustraci, nejedná se o kompletní implementaci této funkce).

```
int8_t countBraces = 0;
uint16_t countFrom = 0;

// check for matching curly braces
if (*it == '{') {
    // opening brace must be preceded by a letter
    if (it == inEquation.constBegin() || !((it-1)->isLower()))
        return (NO_VARIABLE_TO_DIFF);
    ++countBraces;
    countFrom = 0;
}
```

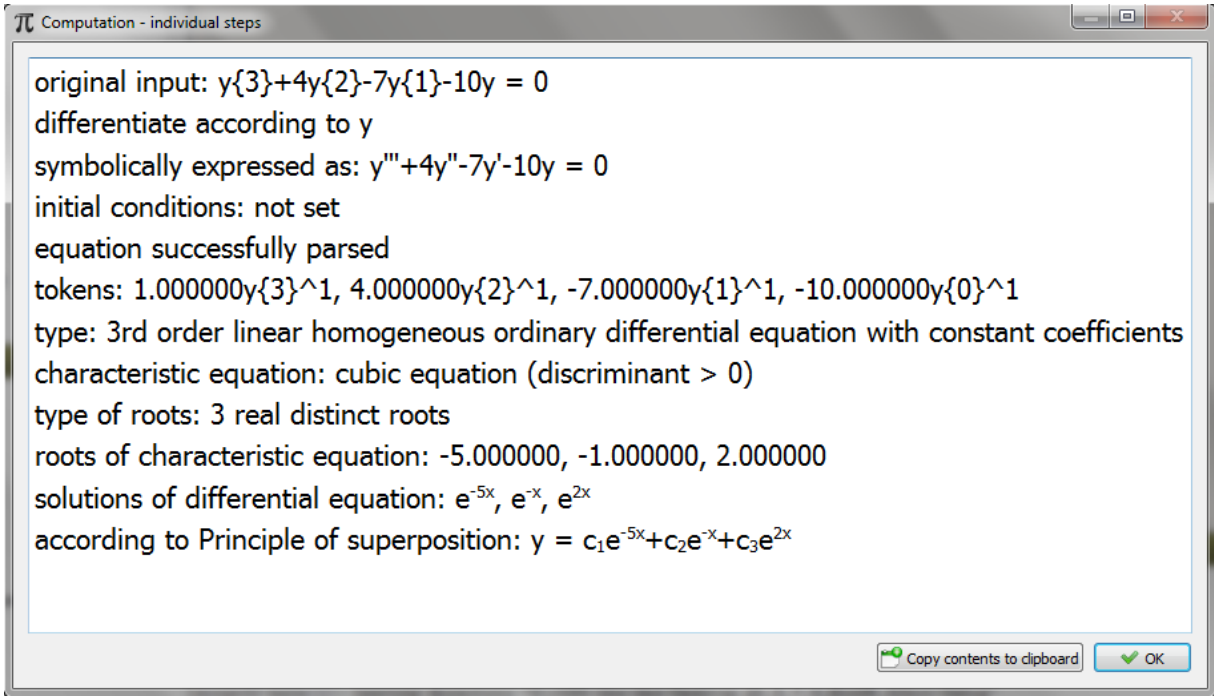
58 uvedený programový kód je **definicí** (*definition*) dané proměnné (typu `const QString`), která se nachází v souboru .cpp, proto neobsahuje specifikátor `static`; ten se uvádí pouze při **deklaraci** (*declaration*) v příslušném hlavičkovém souboru

```
if (*it == '}') {
    // just one character [in range 1-9] is allowed between braces
    if (countFrom != 2 || !isCharAllowed(*(it-1)))
        countBraces = 0;
    --countBraces;
}
```

5.2.1.2 Funkce submitEquationClicked

Tato funkce, jež je volána po odeslání zadané diferenciální rovnice ke zpracování (tlačítkem Submit), vykonává jakýsi „dohled“ nad správným průběhem postupu výpočtu, jinými slovy na základě zadaných hodnot a logických testů postupně volá jednotlivé (podřízené) funkce, které řeší konkrétní úkoly související s analýzou, zpracováním a výpočtem. Mezi ně patří:

- deklarace objektu třídy `Equation` s danými parametry (počáteční podmínky ad.)
- odstranění nadbytečných **bílých znaků** (*whitespace*)[59]
- test, zda se jedná o homogenní rovnici (její pravá strana je prázdná resp. nulová)
- určení proměnné, vzhledem ke které se derivuje, příp. určení, zda není takovýchto proměnných více tj. zda se nejedná o parciální diferenciální rovnici
- volání funkce `parseEquation()`, která provádí tokenizaci
- určení typu diferenciální rovnice
- volání funkce `generalSolution()`, která má na starost vlastní početní řešení
- zobrazení výsledku



obr. č. 18: Zobrazení postupu výpočtu (`StepsWindow`)

59 **Whitespace** je též názvem ezoterického programovacího jazyka, který na rozdíl od většiny ostatních jazyků, které bílé znaky ignorují, přiřazuje význam pouze mezerám, **tabulátorům** a **novým řádkům** (*newline*), a ignoruje veškeré ostatní znaky; nejstarším ezoterickým jazykem je **INTERCAL** (Compiler Language With No Pronounceable Acronym), který byl vytvořen roku 1972 dvěma studenty Princetonské univerzity Donem Woodsem a Jamesem M. Lyonem

5.2.2 Zobrazení postupu výpočtu

K zobrazení postupu výpočtu slouží dialogové okno `StepsWindow` (potomek `QDialog`u), jehož vzhled je patrný z obr. 18. Jedná se z velké části o pasivní prvek uživatelského rozhraní, má za úkol zobrazovat jednotlivé výpočtové kroky, s možností zkopírovat je do systémové **schránky** (*clipboard*), příp. umožňuje vyvolat zobrazení fundamentální matice (viz obr. 19).

Vlastní implementace je v podstatě velmi jednoduchá, nicméně může posloužit jako praktická demonstrace použití mechanismu signálu a slotu (princip funkce viz kap. 4.2.1), jenž v kódu probíhá v následujících třech krocích.

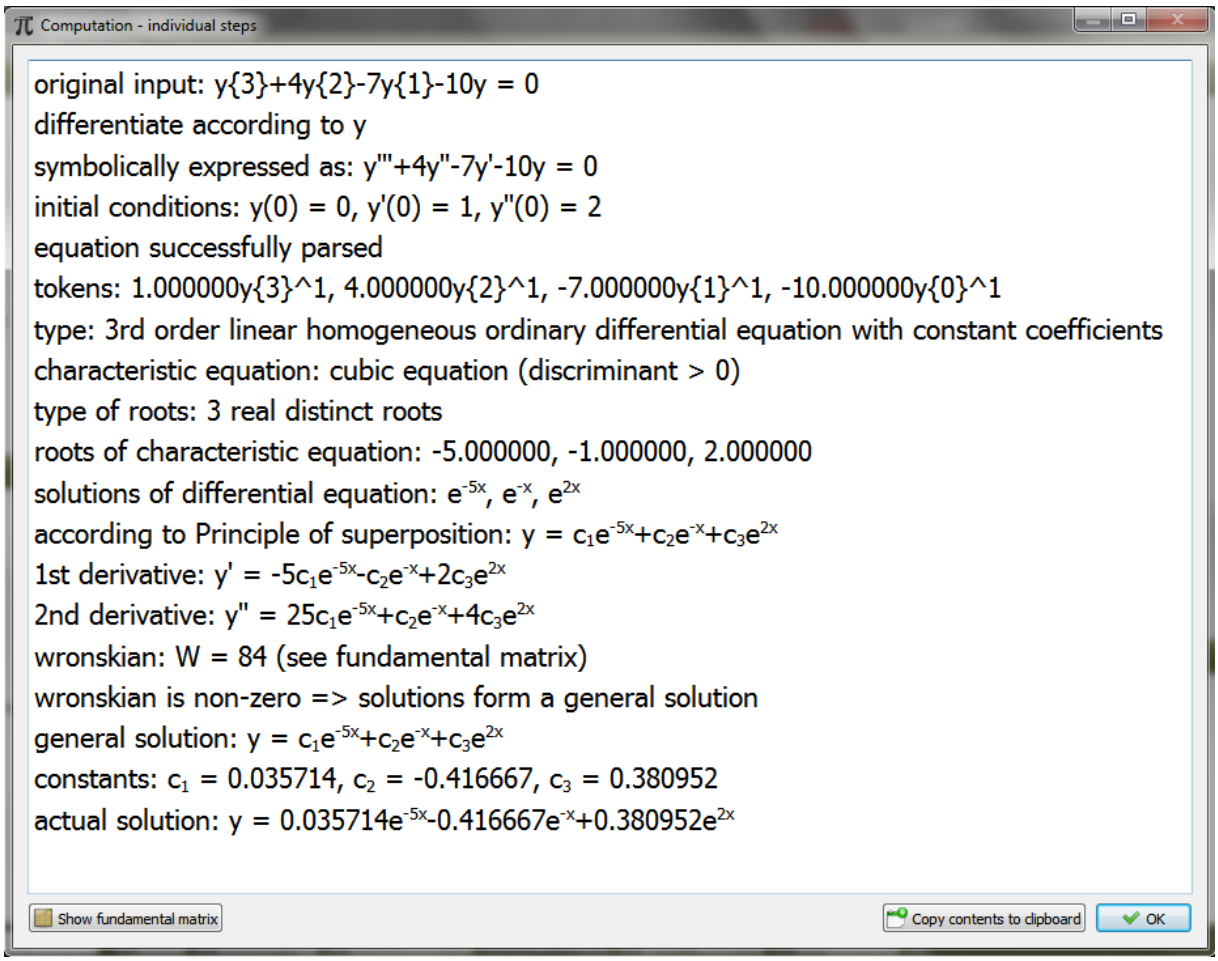
Krok č.1: třída StepsWindow

```
class StepsWindow: public QDialog {

    Q_OBJECT

    public: { ... }
    private:
        QChar diffVariable;
        Matrix<double_t> fundamentalMatrix;
        Ui_StepsWindow * ui;

    private slots:
        int fundamentalMatrix_clicked();
};
```



obr. č. 19: Příklad výpočtu rovnice se zadanými počátečními podmínkami

Krok č.2: tělo konstruktoru

```
{ ... }
// OK button clicked
connect(ui->pushButton_OK, SIGNAL(clicked()), this, SLOT(close()));
// select and copy displayed text
connect(ui->pushButton_CopyToClipboard, SIGNAL(clicked()),
        ui->textEdit_ComputationSteps, SLOT(selectAll()));
connect(ui->pushButton_CopyToClipboard, SIGNAL(clicked()),
        ui->textEdit_ComputationSteps, SLOT(copy()));
// FundamentalMatrix button clicked
connect(ui->pushButton_FundamentalMatrix, SIGNAL(clicked()),
        this, SLOT(fundamentalMatrix_clicked()));
```

Krok č.3: implementace slotu

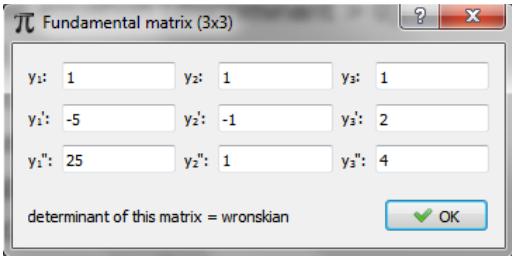
```
int StepsWindow::fundamentalMatrix_clicked() {

    MatrixWindow fmWindow(diffVariable, fundamentalMatrix, this);
    return fundamentalMatrixWindow.exec();
}
```

5.3 Praktické ukázky použití aplikace

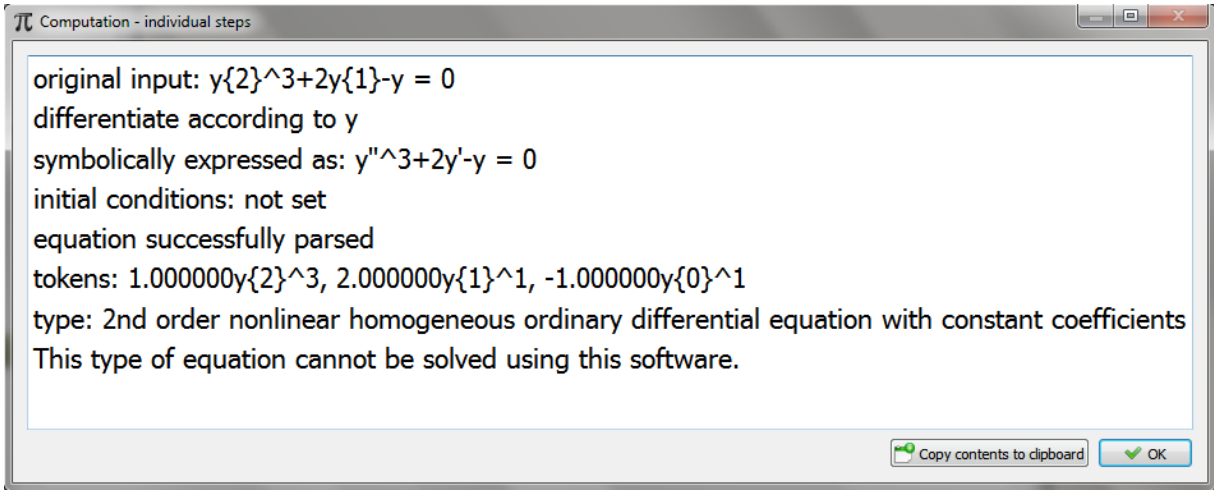
Obr. 19 uvádí příklad lineární homogenní obyčejné diferenciální rovnice 3. řádu s aktuálním řešením, tj. jedná se o rovnici se zadanými počátečními podmínkami. Fundamentální matice, sestavená dle kap. 2.3.5, jež je základem pro výpočet Wronskiánu, je uvedena na obr. 20.

Příklad výpočtu samotného obecného řešení diferenciální rovnice je demonstrován na obr. 18, jenž byl uveden v předcházející kap. 5.2.2.



obr. č. 20: Fundamentální matice ➔

Popisovaná aplikace je schopna rozpoznat i některé typy diferenciálních rovnic, jež nejsou v její aktuální verzi řešitelné, jako příklad možno uvést nelineární diferenciální rovnici, jejíž zadání je patrné z obr. 21.



obr. č. 21: Příklad rovnice, jíž aplikace nedokáže řešit

Další vzorové příklady jsou uvedeny v Příloze C.

5.4 Budoucí vývoj

Přestože je aplikace v podstatě plně funkční, není její vývoj zdaleka ukončen. Do budoucna je plánováno provést mnoho změn a vylepšení, jak v oblasti aplikačního jádra (týká se zejména výpočetních operací a návrhu vhodných struktur pro ukládání dat), tak v oblasti uživatelského rozhraní. Některé ze zamýšlených úprav, jejichž implementace se v současné chvíli nachází v různých fázích rozpracovanosti, jsou uvedeny v následujících podkapitolách.

5.4.1 Změny v programové logice

V závislosti na typu kořenů charakteristické rovnice existuje několik variant řešení LODR s konstantními koeficienty, které jsou detailně popsány v kap. 2.3.4.2. Nutno však podotknout, že v případě čtvrtého (příp. vyššího) řádu zadané rovnice, může nastat ještě jedna varianta, a to sice nepříliš častý, přesto se v praxi občas vyskytující případ, **násobných komplexních kořenů**. Jako příklad je možno uvést třeba následující rovnici:

$$y^{(4)}+8\,y''+16\,y=0\,, \tag{47}$$

jejíž charakteristická rovnice má kořeny:

$$y_{1,2}=-2i\,,\quad y_{3,4}=2i \tag{48}$$

a jejíž řešení vypadá následovně:

$$y=c_1\cos(2x)+c_2\,x\cos(2x)+c_3\sin(2x)+c_4\,x\sin(2x) \tag{49}$$

Takovýto typ rovnice momentálně není v popisované aplikaci řešitelný, je tedy třeba doplnit její funkčnost, aby bylo možné řešit i příklady tohoto specifického typu.

Dalším, dlouhodobějším, cílem je implementace řešení lineárních nehomogenních obyčejných diferenciálních rovnic, tzn. doplnit programový kód o funkce pro výpočet tzv. partikulárního řešení. Existující metody pro určení partikulárního řešení jsou zmíněny v kap. 2.3.3.4.

5.4.2 Změny v uživatelském rozhraní

Názory na kvalitu a funkčnost (příp. i estetickou stránku) uživatelského rozhraní se zcela jistě budou lišit od jednoho uživatele k druhému. Při takovém hodnocení se totiž nelze nikdy zcela vyhnout subjektivnímu pohledu na danou věc, z čehož logicky vyplývá nemožnost uspokojit požadavky všech. Přesto by bylo vhodné některé programové konstrukce přepracovat, aby se zvýšil uživatelský komfort a přehlednost výstupu.

Jednou ze zamýšlených novinek je rozšíření postupu výpočtu o možnost zobrazit matice, které slouží pro výpočet hodnot konstant (viz kap. 2.3.6). Vzhled tohoto dialogového okna by měl korespondovat se způsobem zobrazení fundamentální matice, jež je zřejmý z obr. 20. Vyvolat tento dialog bude možné pomocí klasického **stisknutelného tlačítka** (*push button*).

5.4.3 Webová prezentace

Vzhledem k tomu, že popisovaná aplikace je vyvíjena jakožto nekomerční produkt k volnému použití, jeví se použití celosvětové počítačové sítě **Internet** jako distribučního kanálu pro její rozšíření mezi odbornou veřejnost jako optimální řešení. Z tohoto důvodu je ve střednědobém plánu vytvoření dedikované **webové stránky** (prezentace), která by měla sloužit jednak jako zdroj informací ohledně problematiky nutné k pochopení principů funkce tohoto softwaru, ať již se jedná o nástroje matematické či vývojářské, jednak jako **on-line dokumentace**.

Mezi uvažovanými návrhy, jež by postupem času mohly být zapracovány do takovéto webové prezentace, jsou např. komentované útržky kódu, diskuzní fórum či jiná vhodná forma zpětné vazby, hodnocení podobně zaměřených aplikací jiných vývojářů, odkazy na webové stránky zaměřené na teorii diferenciálního počtu či jiná související témata apod. Zároveň by měla tato webová stránka umožňovat stažení aktuální verze aplikace příp. i zdrojového kódu.

Práce na grafickém i technickém návrhu této prezentace sice již byly započaty, v současné době se však nachází ve velmi rané fázi, která ještě není připravena k publikování. Navíc by to již přesahovalo rámec této práce, která se soustředí spíše na vlastní implementaci. Lze snad alespoň zmínit, že vývoj bude probíhat za použití standardních nástrojů pro vytváření webu: **značkovacího jazyka HMTL**[60] v aktuální verzi HTML 5.0, **kaskádových stylů** (*cascading style sheets*, CSS) používaných pro popis a umisťování jednotlivých elementů, či jazyků umožňujících programování dynamických webových stránek, ať již na straně klienta (JavaScript) či na straně serveru (PHP).

60 HyperText Markup Language

6 ZÁVĚR

Cílem této bakalářské práce bylo vytvořit jednoduchou, ale přesto výkonnou, volně dostupnou matematickou aplikaci disponující uživatelsky přívětivým rozhraním, zaměřenou na výpočty některých základních typů diferenciálních rovnic, s výstupem ve formě obecného, či v případě zadání počátečních podmínek, i aktuálního řešení. Klíčovým požadavkem, kterým se aplikace odlišuje od jí podobných, mělo být zobrazení úplného postupu řešení, neboť záměrem bylo poskytnout studentům vysokých škol podpůrnou výukovou pomůcku pro jejich studium.

Stanoveného cíle bylo dosaženo v podstatě v plném rozsahu, ten totiž nebyl stanoven naprosto exaktně tím, že by taxativně vyjmenovával konkrétní typy rovnic, ale spíše ideově, s hlavní myšlenkou dospět k řešení formou jednotlivých dokumentovaných kroků. V rámci vývoje této aplikace se podařilo implementovat vesměs plnou funkčnost pro lineární homogenní obyčejné diferenciální rovnice s konstantními koeficienty, druhého, třetího a s jistými omezeními též čtvrtého řádu, se kterými se studenti velmi často setkávají.

Zvolená kombinace vývojového prostředí Qt s využitím programovacího jazyka C++ se, dle předpokladů, ukázala jako správná. V případě potřeby bylo možno konzultovat mnoho zdrojů, ať již ve formě tištěné či elektronické, vč. aktuálně platného standardu, on-line referenčních příruček, internetových diskuzních fór, vývojářských blogů nebo velkého množství aplikací s otevřeným zdrojovým kódem. Jediné, co se v průběhu vývoje ukázalo jako nevýhodné, bylo použití externích knihoven, i když toto bylo původně zvažováno, a to zejména v souvislosti s potřebou řešit symbolické výpočty derivací. Nicméně nutnost nastudovat takovou knihovnu a přizpůsobit jí rozhraní vyvíjeného softwaru, aby byl schopen knihovnu používat, se nakonec ukázalo jako časově náročnější, než navrhnout a implementovat vlastní řešení. Během toho byly, v rámci vlastního výzkumu, objeveny zákonitosti pro tvorbu libovolného řádu derivace takového řešení, které bylo sestaveno na základě dvojice komplexních sdružených kořenů charakteristické rovnice, jak bylo popsáno v kapitole o implementaci.

Pochopitelně ještě zbývá mnoho oblastí, které by bylo možné zlepšit či vhodným způsobem upravit, ať již se to týká funkčnosti popisované aplikace, především s ohledem na podporu většího rozsahu typů zadávaných diferenciálních rovnic, nebo v jejím uživatelském rozhraní, které by zajisté bylo možno učinit více intuitivním příp. i lépe přizpůsobitelným požadavkům a potřebám individuálních uživatelů.

7 SEZNAM POUŽITÉ LITERATURY

[1] Dolnick, Edward. *The clockwork universe: Isaac Newton, the Royal society, and the birth of the modern world*. New York: Harper Perennial, 2012. ISBN 978-0061719523.

[2] Ball, Walter William Rouse. *A short account of the history of mathematics*. 4th edition. London: MacMillan, 2010. ISBN 978-0486206301.

[3] *Základní věta integrálního počtu* [online]. Praha: ČVUT, Fakulta elektrotechnická, Katedra matematiky. [cit. 2017-04-24]. Dostupné z: http://math.feld.cvut.cz/mt/txtd/1/txc3da1d.htm

[4] Zwillinger, Daniel. *Handbook of Differential Equations.* 3rd edition. London: Academic Press Limited, 1997. ISBN 978-0127843964.

[5] Pahikalla, J. *Bernoulli equation* [online]. PlanetMath, 2005. Editováno 2013. [cit. 2017-04-24]. Dostupné z: http://planetmath.org/bernoulliequation

[6] Walker, Jearl. D. Halliday. R. Resnick. *Fundamentals of physics.* 10th edition. Hoboken, NJ: John Wiley & Sons, 2014. ISBN 978-1118230725.

[7] Dawkins, Paul. *Differential equations* [online]. Beaumont, Texas: Lamar University, 2007. [cit. 2017-04-24]. Dostupné z: http://tutorial.math.lamar.edu/download.aspx

[8] Gibbs, Philip: *What is the term used for the third derivative of position?* [online]. Riverside, California: University of California, 1996. Poslední změna 1998 [cit. 2017-04-24]. Dostupné z: http://math.ucr.edu/home/baez/physics/General/jerk.html

[9] Rokyta, Mirko. *Rovnice vedení tepla a vlnová rovnice* [online]. Praha: Univerzita Karlova, Matematicko-fyzikální fakulta, Katedra matematické analýzy. [cit. 2017-04-24]. Dostupné z: http://www.karlin.mff.cuni.cz/~rokyta/vyuka/general/tahaky/tahak3re.ps

[10] Singh, Sushil Kumar. *Laguerre differential equation* [online]. New Delhi: University of Delhi, S.G.T.B Khalsa College, Institute of Lifelong Learning, 2015. [cit. 2017-04-24]. Dostupné z: http://vle.du.ac.in/mod/resource/view.php?inpopup=true&id=13155

[11] Dummit, Evan: *Differential equations: Linear differential equations* [online]. Rochester, New York: University of Rochester, 2012. Poslední změna 2016 [cit. 2017-04-24]. Dostupné z: https://web.math.rochester.edu/people/faculty/edummit/docs/lindiff_5_linear_differential_equations.pdf

[12] Doležalová, Jarmila. *Diferenciální rovnice* [online]. Ostrava: Vysoká škola báňská, 2015. [cit. 2017-04-24]. Dostupné z: homen.vsb.cz/~dol30/MII-difrovnice.pdf

[13] James, Ioan. *Remarkable mathematicians: From Euler to von Neumann*. Cambridge: Cambridge University Press, 2003. ISBN 978-0521520942.

[14] Truesdell, Clifford: *An idiot's fugitive essays on science*. New York, Berlin, Heidelberg, Tokio: Springer-Verlag, 1984. ISBN 978-1461381853.

[15] Brock, David C.: *Understanding Moore's law: Four decades of innovation*. Berkeley: Chemical heritage foundation, 2006. ISBN 978-0941901413.

[16] The Mathics Team: *Mathics* (user manual) [online]. 2013-10-27 [cit. 2017-05-02]. Dostupné z: http://mathics.org/doc/mathics.pdf

[17] Norton, Douglas. *Why MAPLE?* [online]. Villanova, PA: Villanova University, Department of Mathematics and Statistics. [cit. 2017-05-02]. Dostupné z: http://www1.villanova.edu/villanova/artsci/mathematics/resources-and-opportunities/maple/whymaple.html

[18] Simerská, Carmen. *Systémy algebro-diferenciálních rovnic* [online]. Pokroky matematiky, fyziky a astronomie, 2005, roč. 50, č. 3, s. 182-192. Dostupné z: http://dml.cz/dmlcz/141270

[19] Foster Kenneth: *Mathematica 8 and Maple 15* [online]. IEEE Spectrum, 2001 [cit. 2017-05-02]. Dostupné z: http://spectrum.ieee.org/geek-life/tools-toys/mathematica-8-and-maple-15

[20] Mihailovs, Alec <alec@mihailovs.com>. *Re: Maple or Mathematica* (e-mail) [online]. 2002 [cit. 2017-05-02]. Dostupné z: http://shell.cas.usf.edu/~wclark/maple_or_mathematica.html

[21] Moncaro, Hever (and students). *Embry-Riddle students and Gulfstream engineers design and implement embedded flight control systems on the Arduino platform* [online]. Daytona Beach, Savannah: Embry-Riddle aeronautical university, Gulfstream aerospace, 2017 [cit. 2017-05-02]. Dostupné z: https://www.mathworks.com/company/user_stories/embry-riddle-students-and-gulfstream-engineers-design-and-implement-embedded-flight-control-systems-on-the-arduino-platform.html

[22] Holoubek, Tomáš. *Raspberry Pi: programování v prostředí Matlab/Simulink* (bakalářská práce). Brno: VUT v Brně, Fakulta strojního inženýrství, Ústav automatizace a informatiky, 2017.

[23] Gill, John (editor). *Best universities in the United States 2017* [online]. London: Times higher education, 2016 [cit. 2017-05-02].
Dostupné z: https://www.timeshighereducation.com/student/best-universities/best-universities-united-states

[24] Matoušek, Radomil. *Metody kódování* [elektronický dokument]. Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, Odbor aplikované informatiky, 2006.

[25] Rawlings, James. J. Ekerdt. *Chemical Reactor Analysis and Design Fundamentals*. Madison, Winsconsin: Nob Hill Publishing, 2002. ISBN: 978-0615118840.

[26] Griesmer, James. R. Jenks. *SCRATCHPAD/1: An interactive facility for symbolic mathematics*. Proceedings of the second ACM symposium on symbolic and algebraic manipulation, 1971, č. 3, s. 42-58. Dostupné z: http://dl.acm.org/citation.cfm?id=806266

[27] Daly, Tim. *Axiom* (user manual) [online]. New York: CAISS, 2003 [cit. 2017-05-02].
Dostupné z: http://www.axiom-developer.org/axiom-website/bookvol1.pdf

[28] Hebisch, Waldek <hebisch@math.uni.wroc.pl>. *Re: [Axiom-developer] Compiling Axiom on Ubuntu 14.04, 64 bit* (e-mail) [online]. 2015 [cit. 2017-05-02].
Dostupné z: https://lists.gnu.org/archive/html/axiom-developer/2015-10/msg00000.html

[29] Phillips, Lee. *Scientific computing's future: Can any coding language top a 1950s behemoth?* [online]. Ars Technica, 2014 [cit. 2017-05-08].
Dostupné z: https://arstechnica.com/science/2014/05/scientific-computings-future-can-any-coding-language-top-a-1950s-behemoth/3/

[30] Steele, Guy. *Common LISP. The Language.* 2nd edition. Oxford, Waltham (MA): Butterworth-Heinemann (Digital Press), 1990. ISBN: 978-1555580414.

[31] Seibel, Peter. *Practical Common Lisp*. New York: Apress Media, 2005. ISBN: 978-1590592397.

[32] Seward Julian, N. Nethercote, J. Weidendorfer and the Valgrind development team. *Valgrind 3.3 – Advanced Debugging and Profiling for GNU/Linux applications*. Bristol, UK: Network Theory Ltd., 2008. ISBN: 978-0954612054.

[33] D'Avezac, M. *How can I choose the right programming language for a computational physics project?* (internetové fórum) [online]. ResearchGate, 2014 [cit. 2017-05-08].
Dostupné z: https://www.researchgate.net/post/How_can_I_choose_the_right_programming_language_for_a_computational_physics_project

[34] MisanthropicScott (nick). *GOD is REAL, unless declared INTEGER (FORTRAN)* (internetové fórum) [online]. Reddit, 2015 [cit. 2017-05-08].
Dostupné z: https://www.reddit.com/r/atheism/comments/34ozzu/god_is_real_unless_declared_integer_fortran/

[35] Goldbert, Adele. D. Robson. *SMALLTALK-80. The language and its implementation.* Reading, Menlo Park, London, Amsterdam, Don Mills, Sydney: Addison-Wesley Publishing Company, 1983. ISBN: 0-201-11371-6.

[36] Reilly, Edwin D. *Concise Encyclopedia of Computer Science.* Hoboken, NJ: John Wiley & Sons, 2004. ISBN 978-0470090954.

[37] Bartlett, Jonathan. *Programming from the Ground Up* [elektronická kniha]. Bartlett Publishing, 2003. ASIN: B014ICMTEK.

[38] Ellis, Margaret A. B. Stroustrup. *The Annotated C++ Reference Manual.* Boston, MA: Addison-Wesley Professional, 1990. ISBN 978-0201514599.

[39] Stroustrup, Bjarne. *The C++ programming language*. 3rd edition. Reading, MA: Addison-Wesley Longman Inc., 1997. ISBN: 0-201-88954-4.

[40] Solter, Nicholas A. S. Kleper. *Professional C++*. Indianapolis, IN: Wiley Publishing, Inc., 2005, ISBN: 0-7645-7484-1.

[41] Pohl, Ira. *C++ by dissection*. Boston, MA: Addison-Wesley, 2002. ISBN 0-201-74396-5.

[42] Henricson, Mats. E. Nyquist. *Industrial strength C++*. Upper Saddle River, NJ: Prentice Hall, 1997. ISBN: 0-13-120965-5.

[43] ISO/IEC. *Working Draft, Standard for Programming Language C++* (revize č. N4659) [online]. Washington: Standard C++ Foundation, 2017-03-21 [cit. 2017-05-08]. Dostupné z: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf

[44] ISO/IEC. *Working Draft, Standard for Programming Language C++* (revize č. N3337) [online]. Washington: Standard C++ Foundation, 2012-01-16 [cit. 2017-05-08]. Dostupné z: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf

[45] ISO/IEC. *Working Draft, Standard for Programming Language C++* (revize č. N4296) [online]. Washington: Standard C++ Foundation, 2014-11-19 [cit. 2017-05-08]. Dostupné z: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf

[46] Miller, William M. *A Taxonomy of Expression value categories* [elektronický dokument]. Edison design group, 2010-03-12 [cit. 2017-05-08].

[47] Josuttis, Nicolai M. *The C++ Standard Library: A Tutorial and Reference*. 2nd edition. Boston, MA: Addison-Wesley Professional, 2012. ISBN: 978-0321623218.

[48] Kieras, David. *Using C++11's Smart Pointers* [online]. Ann Arbor, MI: University of Michigan, 2016 [cit. 2017-05-08]. Dostupné z: http://umich.edu/~eecs381/handouts/C++11_smart_ptrs.pdf

[49] Friedl, Jeffrey E.F. *Mastering regular Expressions*. 3rd edition. Sebastopol, CA: O'Reilly Media, 2006. ISBN: 978-0596528126.

[50] Chroboczek, Martin. *Grafická uživatelská rozhraní v Qt a C++*. Brno, Computer Press, 2013. ISBN: 978-8025141243.

[51] Blanchette, Jasmin. M. Summerfield. *C++ GUI Programming with Qt 4*. 2nd edition. Upper Saddle River, NJ: Prentice Hall, 2008. ISBN: 978-0132354165.

[52] Rischpater, Ray. *Application Development with Qt Creator.* 2nd edition. Birmingham, UK: Packt Publishing Ltd., 2014. ISBN: 978-1784398675.

[53] Gough, Brian J. *An introduction to GCC*. 2nd edition. Bristol, UK: Network Theory Ltd., 2005. ISBN: 978-0954161798.

[54] Danis, Sharron Ann. *Rear Admiral Grace Murray Hopper.* Norfolk, VA: Virginia Tech/Norfolk State University, 1997-02-16 [cit. 2017-05-08].

[55] Cederqvist, Per et al. *Version Management with CVS*. Bristol, UK: Network Theory Ltd., 2005. ISBN: 978-0954161712.

[56] cppreference.com. *abs, labs, llabs, imaxabs* [online referenční příručka]. 2012. Poslední změna 2015 [cit. 2017-05-15]. Dostupné z: http://en.cppreference.com/w/c/numeric/math/abs

[57] cppreference.com. *std::abs(float), std::fabs* [online referenční příručka]. 2011. Poslední změna 2016 [cit. 2017-05-15]. Dostupné z: http://en.cppreference.com/w/cpp/numeric/math/fabs

[58] Haendel, Lars. *Introduction to C and C++ Function Pointers, Callbacks and Functors* [online]. Bochum: 2015. Dostupné z: http://www.newty.de/fpt/zip/e_fpt.pdf

[59] Roupec, Jan. *Jazyky C a C++* [elektronický dokument]. Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, Odbor aplikované informatiky, 2012.

[60] Wikipedia: The free encyclopedia. *Cubic function* [online]. Wikimedia Foundation, Inc., 2003. Poslední změna 2017 [cit. 2017-05-15].
Dostupné z: https://en.wikipedia.org/wiki/Cubic_function

[61] Landers, Mara et al. *Math analysis FlexBook* [online]. Palo Alto, CA: CK-12 Foundation, 2012. Poslední změna 2014. Dostupné z: http://www.ck12.org/book/ck-12-math-analysis/section/4.6/

[62] Singh, Simon. *Fermat's Enigma*. New York: Anchor Books, 1998. ISBN: 978-0385493628.

[63] Rosen, Michael I. *Niels Hendrik Abel and Equations of the fifth degree* [online]. Providence, RI: Brown University, Mathematics Department, 1995.
Dostupné z: https://www.maa.org/sites/default/files/pdf/upload_library/22/Chauvenet/Rosen.pdf

8 SEZNAM PŘÍLOH

Příloha A:
Přehled významných osobností z oboru matematiky, fyziky, výpočetní techniky a příbuzných odvětví, které se podstatným způsobem zasloužili o rozvoj lidského poznání, a o nichž se tato práce zmiňuje.

Příloha B:
Kompletní programový kód funkce `rootsOfCubicEquation()` určené pro výpočet kořenů charakteristické rovnice 3. řádu (obecně jakékoli kubické rovnice). Jedná se o implementaci postupu popsaného v kap. 5.1.2.2, kterému odpovídá větvení v kódu na základě vypočtených hodnot diskriminantů. Uvedeno vč. komentářů a původního formátování.

Příloha C:
Konkrétní příklady výpočtů několika základních typů lineárních obyčejních diferenciálních rovnic s konst. koeficienty (s nulovou pravou stranou tj. homogenních) pomocí popisované aplikace. Jsou uvedeny příklady rovnic druhého, třetího i čtvrtého řádu, s různými variantami kořenů charakteristické rovnice (reálné i komplexní, jednoduché i násobné, příp. jejich různé kombinace). Výsledky výpočtů vč. jednotlivých derivací jsou přímými výstupy popisovaného softwaru, pouze jsou z důvodu přehlednosti prezentovány v odlišném formátu.

Příloha D:
Ukázka jednoho z mnoha ručních kontrolních výpočtů, který byl použit při hledání, kontrole a optimalizaci algoritmu pro určování derivací.

Elektronická příloha:
Staticky linkovaná verze popisované softwarové aplikace je k dispozici na přiloženém CD.

PŘÍLOHA A

Abel, Niels H.	norský matematik, je po něm pojmenována **Abelova cena**[61]
Babbage, Charles	anglický matematik, vynálezce a filosof
Bell, Alexander G.	skotsko-americký vědec a inženýr, vynálezce telefonu
Berners Lee, Tim	vynálezce systému www (World Wide Web)
Bernoulli, Jacob	švýcarský matematik
Bernoulli, Johann	švýcarský matematik
Cerf, Vinton	společně s Robertem Kahnem vyvinuli TCP/IP protokol
Cramer, Gabriel	švýcarský matematik
de Fermat, Pierre	francouzský matematik a právník
Diffie, Whitfield	průkopník asymetrické kryptografie
Dijkstra, Edsgar W.	nizozemský počítačový vědec, autor Dijkstrova algoritmu
Diofantos z Alexandrie	starořecký matematik
Euler, Leonhard	švýcarský matematik, fyzik a astronom
Fano, Robert	spoluautor Shannon-Fanova kódování
Hamming, Richard	americký matematik a počítačový vědec
Hellman, Martin	průkopník asymetrické kryptografie
Hoene-Wroński, Józef M.	polský matematik a filosof
Kahn, Robert	společně s Vintonem Cerfem vyvinuli TCP/IP protokol
Knuth, Donald E.	navrhl systém pro počítačovou sazbu TEX

61 toto prestižní ocenění patří, společně s **Fieldsovou medailí**, k nejvýznamnějším oceněním udělovaným významným matematikům; o jejím příjemci rozhoduje každoročně norská vláda; v minulosti ji obdrželi např. John Nash (2015) či Andrew Wiles (2016)

Lagrange, Joseph-Louis	italsko-francouzský matematik a astronom
Laguerre, Edmond N.	francouzský matematik
Leibniz, Gottfried W.	německý matematik a filosof
Lovelace, Ada	anglická matematička, autorka prvního algoritmu
McCarthy, John	autor programovacího jazyka Common Lisp
Moore, Gordon E.	spoluzakladatel Intel Corporation, vyslovil Moorův zákon
Nash, John F.	americký matematik, držitel Nobelovy ceny za ekonomii (1994), objevitel konceptu **Nashovy rovnováhy** (*Nash equilibrium*)
Newton, Isaac	anglický matematik, fyzik a astronom
Pascal, Blaise	francouzský matematik a fyzik
Ritchie, Dennis	spolutvůrce operačního systému **Unix**, autor jazyka C
Ruffini, Paolo	italský matematik a filosof
Shannon, Claude	„otec“ teorie informace
Stroustrup, Bjarne	autor programovacího jazyka C++
Thomson, Ken	spolutvůrce operačního systému **Unix**
Turing, Alan M.	anglický počítačový vědec, matematik, logik a kryptoanalytik, autor **Turingova stroje** (*Turing machine*) a souvisejícího konceptu **Turingovské úplnosti** (*Turing completeness*)
van Rossum, Guido	autor programovacího jazyka Python
Wall, Lary	počítačový programátor, autora jazyka Perl
Wiles, Andrew	matematik, jenž dokázal platnost Velké Fermatovy věty
Wolfram, Stephen	britsko-americký počítačový odborník

PŘÍLOHA B

```
const QString CharEq::rootsOfCubicEquation(const QVector<Token> &
              inTokens, const initConditionsStruct & inConditions) {

    const complex a = inTokens.at(0).getComplexCoef(),
                  b = inTokens.at(1).getComplexCoef(),
                  c = inTokens.at(2).getComplexCoef(),
                  d = inTokens.at(3).getComplexCoef();

    // compute discriminant
    const complex temp1 = compute::toComplex(27)*a*a;
    const complex discriminant = compute::toComplex(18)*a*b*c*d -
        compute::toComplex(4)*std::pow(b,3)*d + b*b*c*c -
        compute::toComplex(4)*a*std::pow(c,3) - temp1*d*d;
    const complex disD0 = b*b - compute::toComplex(3)*a*c;
    const complex disD1 = compute::toComplex(2)*std::pow(b,3) -
                          compute::toComplex(9)*a*b*c + temp1*d;

    // compute roots
    if (compute::round(discriminant.real()) == 0) {

        // discriminant is zero
        if (compute::round(disD0.real()) == 0) {

            // one triple real root
            const complex root = -b / compute::toComplex(3)*a;
            for (uint8_t i=0; i<3; ++i)
                rootsOfCharEquation.push_back(
                  solution(root,inConditions));
        }
        else {
            // one double real root + one distinct real root
            const complex temp2 = compute::toComplex(9)*a*d;
            complex root1 =
                (temp2 - b*c) / (compute::toComplex(2)*disD0);
            rootsOfCharEquation.push_back(
                solution(root1,inConditions));
            rootsOfCharEquation.push_back(
                solution(root1,inConditions));
            complex root2 = (compute::toComplex(4)*a*b*c -
                             temp2*a - std::pow(b,3)) / (a*disD0);
            rootsOfCharEquation.push_back(
                solution(root2,inConditions));
        }
    }
    else {
        // discriminant is non-zero
        complex C;
        bool tripleRoot = false;

        if (discriminant.real() < 0) {

            // one distinct real root + two complex conjugate roots

            // use standard cbrt function =>
            // computes cbrt of real numbers
```

```
            // firstly calculate only part of C expression
            double_t sqrtPart =
                std::sqrt(-temp1.real()*discriminant.real());

            // round floating point numbers before comparing them
            if (compute::round(disD1.real()) ==
                compute::round(sqrtPart))

                // expressions in the numerator part of the fraction
                // under cbrt ie. discriminant D1 and sqrtPart must
                // not cancel each other
                C = compute::toComplex(
                    std::cbrt(((disD1.real()+sqrtPart)/2)));
            else
                C = compute::toComplex(
                  std::cbrt(((disD1.real()-sqrtPart)/2)));
        }
        else {
            // three distinct real roots

            // use custom-made cbrt function =>
            // computes cbrt of complex numbers
            C = compute::cbrt(
                ((disD1 + std::sqrt(-temp1*discriminant)) /
                compute::toComplex(2)));

            // although computed root is a real number it could (in
            // theory) have non-zero imag-part because intermediate
            // result is a complex number and due to the way
            // floating-point numbers are stored in memory there
            // could have emerged imag-part residuals during
            // performing mathematical operations => result must be
            // normalized ie. its imag-part must be zeroed out via
            // cutOffImagPart() function
            tripleRoot = true;
        }
        complex root1 =
            (compute::toComplex(-1.0/3)/a)*(b+C+disD0/C);
        if (tripleRoot) cutOffImagPart(root1);
        rootsOfCharEquation.push_back(solution(root1,inConditions));
        complex Cx = compute::toComplex(C)*compute::rootTwoOfCubic;
        complex root2 =
            (-compute::toComplex(1.0/3)/a)*(b+Cx+disD0/Cx);
        if (tripleRoot) cutOffImagPart(root2);
        rootsOfCharEquation.push_back(solution(root2,inConditions));
        Cx = compute::toComplex(C)*compute::rootThreeOfCubic;
        complex root3 =
           (-compute::toComplex(1.0/3)/a)*(b+Cx+disD0/Cx);
        if (tripleRoot) cutOffImagPart(root3);
        rootsOfCharEquation.push_back(solution(root3,inConditions));
    }

    QString EquationType =
        QStringLiteral("cubic equation(discriminant ") +
        stateOfDiscriminant(discriminant.real()) + ')';
    return (EquationType);
}
```

PŘÍLOHA C

výsledek: $c_1e^{-3x}+c_2e^{3x}$
první derivace: $-3c_1e^{-3x}+3c_2e^{3x}$

2 **1 dvojnásobný reálný kořen** 1.000 (2x)
zadání: $y'' - 2y' + y = 0$
výsledek: $c_1e^{x}+c_2xe^{x}$
první derivace: $c_1e^{x}+c_2e^{x}+c_2xe^{x}$

2 **2 komplexní kořeny (reálná část je nulová)** 0.000 ± 2.000i
zadání: $y'' + 4y = 0$
výsledek: $c_1\sin(2x)+c_2\cos(2x)$
první derivace: $2c_1\cos(2x)-2c_2\sin(2x)$

2 **2 komplexní kořeny** 1.000 ± 2.000i
zadání: $y'' - 2y' + 5y = 0$
výsledek: $c_1e^{x}\cos(2x)+c_2e^{x}\sin(2x)$
první derivace: $c_1e^{x}\cos(2x)-2c_1e^{x}\sin(2x)+c_2e^{x}\sin(2x)+2c_2e^{x}\cos(2x)$

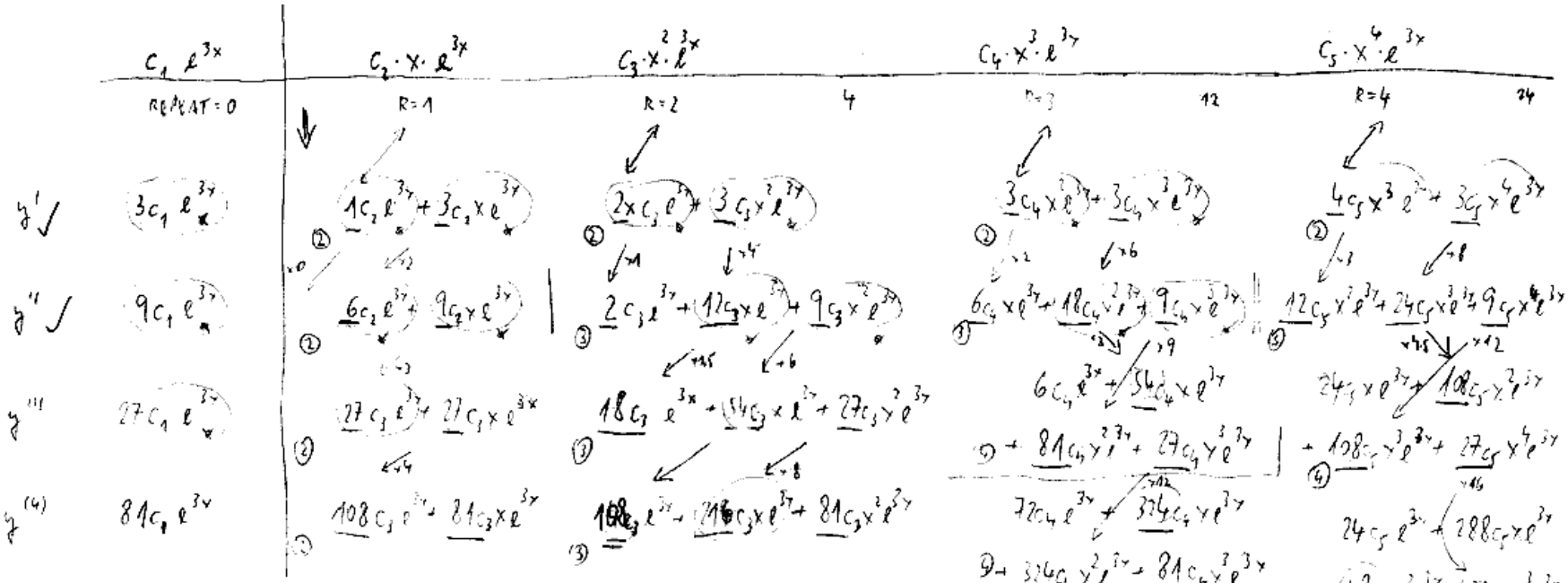
3 **3 reálné kořeny** -1.000; 1.000; 2.000
zadání: $y''' - 2y'' – y' + 2y = 0$
výsledek: $c_1e^{-x}+c_2e^{x}+c_3e^{2x}$
první derivace: $-c_1e^{-x}+c_2e^{x}+2c_3e^{2x}$
druhá derivace: $c_1e^{-x}+c_2e^{x}+4c_3e^{2x}$

3 **1 reálný (nulový) a 2 komplexní kořeny (reálná část je nulová)** 0.000
zadání: $y''' + 9y' = 0$ 0.000 ± 3.000i
výsledek: $c_1+c_2\cos(3x)+c_3\sin(3x)$
první derivace: $-3c_2\sin(3x)+3c_3\cos(3x)$
druhá derivace: $-9c_2\sin(3x)-9c_3\cos(3x)$

3 **1 reálný a 2 komplexní kořeny** 1.000
zadání: $y''' - 3y'' + 7y' - 5y = 0$ 1.000 ± 2.000i
výsledek: $c_1e^{x}+c_2e^{x}\cos(2x)+c_3e^{x}\sin(2x)$
první derivace: $c_1e^{x}+c_2e^{x}\cos(2x)-2c_2e^{x}\sin(2x)+ c_3e^{x}\sin(2x)+2c_3e^{x}\cos(2x)$
druhá derivace: $c_1e^{x}-3c_2e^{x}\cos(2x)-4c_2e^{x}\sin(2x)-3c_3e^{x}\sin(2x)+4c_3e^{x}\cos(2x)$

stupeň	a typy kořenů char.rovnice	vzorový příklad	kořeny
3	**1 trojnásobný reálný kořen**		2.000 (3x)
	zadání:	$y''' - 6y'' + 12y' - 8y = 0$	
	výsledek:	$c_1e^{2x}+c_2xe^{2x}+c_3x^2e^{2x}$	
	první derivace:	$2c_1e^{2x}+c_2e^{2x}+2c_2xe^{2x}+2c_3xe^{2x}+2c_3x^2e^{2x}$	
	druhá derivace:	$4c_1e^{2x}+4c_2e^{2x}+4c_2xe^{2x}+ 2c_3e^{2x}+8c_3xe^{2x}+4c_3x^2e^{2x}$	
3	**1 reálný a 1 dvojnásobný reálný kořen**		2.000 (2x); 3.000
	zadání:	$y''' - 7y'' + 16y' - 12y = 0$	
	výsledek:	$c_1e^{2x}+c_2xe^{2x}+c_3e^{3x}$	
	první derivace:	$2c_1e^{2x}+c_2e^{2x}+2c_2xe^{2x}+3c_3e^{3x}$	
	druhá derivace:	$4c_1e^{2x}+4c_2e^{2x}+4c_2xe^{2x}+9c_3e^{3x}$	
4	**4 reálné kořeny (jeden z nich je dvojnásobný nulový)**		0.000 (2x); ± 2.000
	zadání:	$y^{(4)} - 4y'' = 0$	
	výsledek:	$c_1e^{-2x}+c_2+c_3x+c_4e^{2x}$	
	první derivace:	$-2c_1e^{-2x}+c_3+2c_4e^{2x}$	
	druhá derivace:	$4c_1e^{-2x}+4c_4e^{2x}$	
	třetí derivace:	$-8c_1e^{-2x}+8c_4e^{2x}$	
4	**4 reálné kořeny**		± 2.000; ± 3.000
	zadání:	$y^{(4)} - 13y'' + 36y = 0$	
	výsledek:	$c_1e^{-3x}+c_2e^{-2x}+c_3e^{2x}+c_4e^{3x}$	
	první derivace:	$-3c_1e^{-3x}-2c_2e^{-2x}+2c_3e^{2x}+3c_4e^{3x}$	
	druhá derivace:	$9c_1e^{-3x}+4c_2e^{-2x}+4c_3e^{2x}+9c_4e^{3x}$	
	třetí derivace:	$-27c_1e^{-3x}-8c_2e^{-2x}+8c_3e^{2x}+27c_4e^{3x}$	
4	**2 reálné (nulové) a 2 komplexní kořeny (reálná část je nulová)**		0.000 (2x)
	zadání:	$y^{(4)} + 4y'' = 0$	0.000 ± 2.000i
	výsledek:	$c_1+c_2x+c_3\cos(2x)+c_4\sin(2x)$	
	první derivace:	$c_2-2c_3\sin(2x)+2c_4\cos(2x)$	
	druhá derivace:	$-4c_3\cos(2x)-4c_4\sin(2x)$	
	třetí derivace:	$8c_3\sin(2x)-8c_4\cos(2x)$	
4	**2 reálné (necelé) a 2 komplexní kořeny (reálná část je nulová)**		-1.952; 1.952
	zadání:	$-2y^{(4)} + 5y'' + 10y = 0$	0.000 ± 1.145i
	výsledek:	$c_1e^{-1.952x}+c_2e^{1.952x}+c_3\sin(1.145x)+c_4\cos(1.145x)$	
	první derivace:	$-1.952c_1e^{-1.952x}+1.952c_2e^{1.952x}+$ $1.145c_3\cos(1.145x) -1.145c_4\sin(1.145x)$	
	druhá derivace:	$3.812c_1e^{-1.952x}+3.812c_2e^{1.952x}$ $-1.312c_3\sin(1.145x) -1.312c_4\cos(1.145x)$	
	třetí derivace:	$-7.442c_1e^{-1.952x}+7.442c_2e^{1.952x}$ $-1.502c_3\cos(1.145x)+1.502c_4\sin(1.145x)$	

stupeň a typy kořenů char.rovnice	vzorový příklad	kořeny
4	**2 dvojnásobné reálné kořeny**	± 1.000 (2x)
	zadání: $y^{(4)} - 2y'' + 1 = 0$	
	výsledek: $c_1e^{-x}+c_2xe^{-x}+c_3e^{x}+c_4xe^{x}$	
	první derivace: $-c_1e^{-x}+c_2e^{-x}-c_2xe^{-x}+c_3e^{x}+c_4e^{x}+c_4xe^{x}$	
	druhá derivace: $c_1e^{-x}-2c_2e^{-x}+c_2xe^{-x}+c_3e^{x}+2c_4e^{x}+c_4xe^{x}$	
	třetí derivace: $-c_1e^{-x}+3c_2e^{-x}-c_2xe^{-x}+ c_3e^{x}+3c_4e^{x}+c_4xe^{x}$	
4	**4 komplexní (necelé) kořeny**	± 0.702 ± 1.320i
	zadání: $2y^{(4)} + 5y'' + 10y = 0$	
	výsledek: $c_1e^{0.702x}\sin(1.320x)+c_2e^{0.702x}\cos(1.320x)+$ $c_3e^{-0.702x}\cos(1.320x)+c_4e^{-0.702x}\sin(1.320x)$	
	první derivace: $0.702c_1e^{0.702x}\sin(1.320x)+1.320c_1e^{0.702x}\cos(1.320x)+$ $0.702c_2e^{0.702x}\cos(1.320x)-1.320c_2e^{0.702x}\sin(1.320x)$ $-0.702c_3e^{-0.702x}\cos(1.320x)-1.320c_3e^{-0.702x}\sin(1.320x)$ $-0.702c_4e^{-0.702}x\sin(1.320x)+1.320c_4e^{-0.702x}\cos(1.320x)$	
	druhá derivace: $-1.25c_1e^{0.702x}\sin(1.320x)+1.854c_1e^{0.702x}\cos(1.320x)$ $-1.25c_2e^{0.702x}\cos(1.320x)-1.854c_2e^{0.702x}\sin(1.320x)$ $-1.25c_3e^{-0.702x}\cos(1.320x)+1.854c_3e^{-0.702x}\sin(1.320x)$ $-1.25c_4e^{-0.702x}\sin(1.320x)-1.854c_4e^{-0.702x}\cos(1.320x)$	
	třetí derivace: $-3.326c_1e^{0.702x}\sin(1.320x)-0.348c_1e^{0.702x}\cos(1.320x)$ $-3.326c_2e^{0.702x}\cos(1.320x)+0.348c_2e^{0.702x}\sin(1.320x)+$ $3.326c_3e^{-0.702x}\cos(1.320x)+0.348c_3e^{-0.702x}\sin(1.320x)+$ $3.326c_4e^{-0.702x}\sin(1.320x)-0.348c_4e^{-0.702x}\cos(1.320x)$	
4	**4 komplexní kořeny**	± 1.000 ± 1.000i
	zadání: $y^{(4)} + 4y = 0$	
	výsledek: $c_1e^{x}\sin(x)+c_2e^{x}\cos(x)+c_3e^{-x}\cos(x)+c_4e^{-x}\sin(x)$	
	první derivace: $c_1e^{x}\sin(x)+c_1e^{x}\cos(x)+c_2e^{x}\cos(x)-c_2e^{x}\sin(x)$ $-c_3e^{-x}\cos(x)-c_3e^{-x}\sin(x)-c_4e^{-x}\sin(x)+c_4e^{-x}\cos(x)$	
	druhá derivace: $2c_1e^{x}\cos(x)-2c_2e^{x}\sin(x)+2c_3e^{-x}\sin(x)-2c_4e^{-x}\cos(x)$	
	třetí derivace: $-2c_1e^{x}\sin(x)+2c_1e^{x}\cos(x)-2c_2e^{x}\cos(x)-2c_2e^{x}\sin(x)+$ $2c_3\,e^{-x}\cos(x)-2c_3e^{-x}\sin(x)+2c_4e^{-x}\sin(x)+2c_4e^{-x}\cos(x)$	
4	**1 čtyřnásobný reálný kořen**	-3.000 **(**4x**)**
	zadání: $y(4) + 12y''' + 54y'' + 108y' + 81y = 0$	
	výsledek: $c_1e^{-3x}+c_2xe^{-3x}+c_3x^2e^{-3x}+c_4x^3e^{-3x}$	
	první derivace: $-3c_1e^{-3x}+c_2e^{-3x}-3c_2xe^{-3x}+2c_3xe^{-3x}$ $-3c_3x^2e^{-3x}+3c_4x^2e^{-3x}-3c_4x^3e^{-3x}$	
	druhá derivace: $9c_1e^{-3x}-6c_2e^{-3x}+9c_2xe^{-3x}+2c_3e^{-3x}-12c_3xe^{-3x}+$ $9c_3x^2e^{-3x}+6c_4xe^{-3x}-18c_4x^2e^{-3x}+9c_4x^3e^{-3x}$	
	třetí derivace: $-27c_1e^{-3x}+27c_2e^{-3x}-27c_2xe^{-3x}-18c_3e^{-3x}+54c_3xe^{-3x}$ $-27c_3x^2e^{-3x}+6c_4e^{-3x}-54c_4xe^{-3x}+81c_4x^2e^{-3x}-27c_4x^3e^{-3x}$	



1. DERIVACE: REPEAT × KONST. × X^{R-1} × KOŘEN + (REALPART)^{DER} × KONST. × X^{R} × KOŘEN

2. DERIVACE: REPEAT (REPEAT−1) × KONST. × X^{R-2} × KOŘEN + (REPEAT^2 × REPEAT)^{DER} × KONST. × X^{R-1} × KOŘEN + (REALPART)^{DER} × KONST. × X^{R} × KOŘEN

POSLEDNÍ ČLEN: (REALPART)^{DER} × X^{R}

PŘEDPOSLEDNÍ: (REALPART × DER × REPEAT) × X^{R-1}