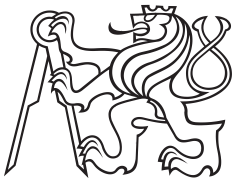
Diplomová práce



České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačů

Nelineární dynamické systémy

Bc. David Passler

Vedoucí: prof. RNDr. Petr Kulhánek, CSc.
Obor: Softwarové inženýrství
Studijní program: Otevřená informatika
Květen 2018

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra počítačů

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Passler David

Studijní program: Otevřená informatika
Obor: Softwarové inženýrství

Název tématu: Nelineární dynamické systémy

Pokyny pro vypracování:

Proveďte analýzu stávajícího programu pro zobrazování grafových výstupů nelineárních dynamických systémů.
Navrhněte řešení v programovacím jazyce Java se zachováním stávající funkcionality a s přihlédnutím k rozšiřování aplikace v budoucnosti o další dynamické systémy.
Implementujte navržené řešení v programovacím jazyce Java, které bude zobrazovat grafové výstupy na základě uživatelských vstupů do programu, včetně zadávání počátečních podmínek pomocí klikáni myší do grafového výstupu.
Aplikace bude obsahovat časový vývoj, fázový portrét, u podivných atraktorů také 3D zobrazení situace.
Otestujte aplikaci jak pomocí jednotkových testů, tak i uživatelským testováním na zvoleném vzorku lidí.
Řešení řádně zdokumentujte (zdrojové kódy ve formátu javadoc, uživatelská dokumentace v PDF).

Seznam odborné literatury:

[1] Petr Kulhánek: Vybrané kapitoly z teoretické fyziky; AGA 2016
[2] Brian D. Storey: Numerical Methods for Differential Equations; Franklin W. Olin College of Engineering , 2015, online: http://faculty.olin.edu/bstorey/Notes/DiffEq.pdf
[3] Walter Fendt: Java Programs on Physics; 2016 Java 6, online: http://www.walter-fendt.de/ph6en/

Vedoucí: prof. RNDr. Petr Kulhánek, CSc.

Platnost zadání do konce zimního semestru 2018/2019

L.S.

prof. Dr. Michal Pěchouček, MSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 18.4.2017

Poděkování

Chtěl bych poděkovat vedoucímu práce panu prof. RNDr. Petru Kulhánkovi, CSc. za odborný dohled a pomoc při realizaci této diplomové práce. Dále bych chtěl poděkovat své rodině a přátelům za podporu, kterou mi projevovali nejen při studiu, ale i při psaní této práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškerou použitou literaturu.

V Praze, 17. května 2018

Abstrakt

Tato diplomová práce si klade za cíl provést analýzu stávajícího programu pro simulaci nelineárních dynamických systémů. Dále si klade za cíl vyvinout novou verzi tohoto programu v programovacím jazyce Java, ve kterém bude zachována stávající funkcionalita programu. K této stávající funkcionalitě bude přidána možnost postupného vykreslování simulace, paralelního vykreslování simulací pomocí zadání počatečních podmínek kliknutím myší a v neposlední řadě zobrazení též 3D modelů u podivných atraktorů. Vytvořený program bude sloužit pro podporu výuky fyziky na FEL ČVUT. To umožní názorná ukázka simulace chování částic v daném systému při daných počátečních podmínkách a parametrech systému.

Klíčová slova: Java aplikace, simulace dynamických systémů, 3D znázornění podivných atraktorů

Vedoucí:
prof. RNDr. Petr Kulhánek, CSc.
Praha
Technická 2
B2-43

Abstract

This diploma thesis is devoted to analysis of existing program for nonlinear dynamical systems simulation. Furthermore it is aimed at developing new version of this program in Java programming language, in which the current functionality will be preserved. In addition to this preserved functionality, following functionalities will be added: continuous simulation drawing, parallel simulation drawing through the use of inputting start conditions via mouse click and finally, but not least showing 3D Strange atractor models. Created program will be used for education purposes on FEE CTU due to illustrative simulation of the particle behavior in given system with given starting conditions and system parameters.

Keywords: Java application, dynamic system simulation, Strange atractor 3D visualization

Title translation: Nonlinear dynamical systems

Obsah

Obrázky

Tabulky

Kapitola 1

Úvod

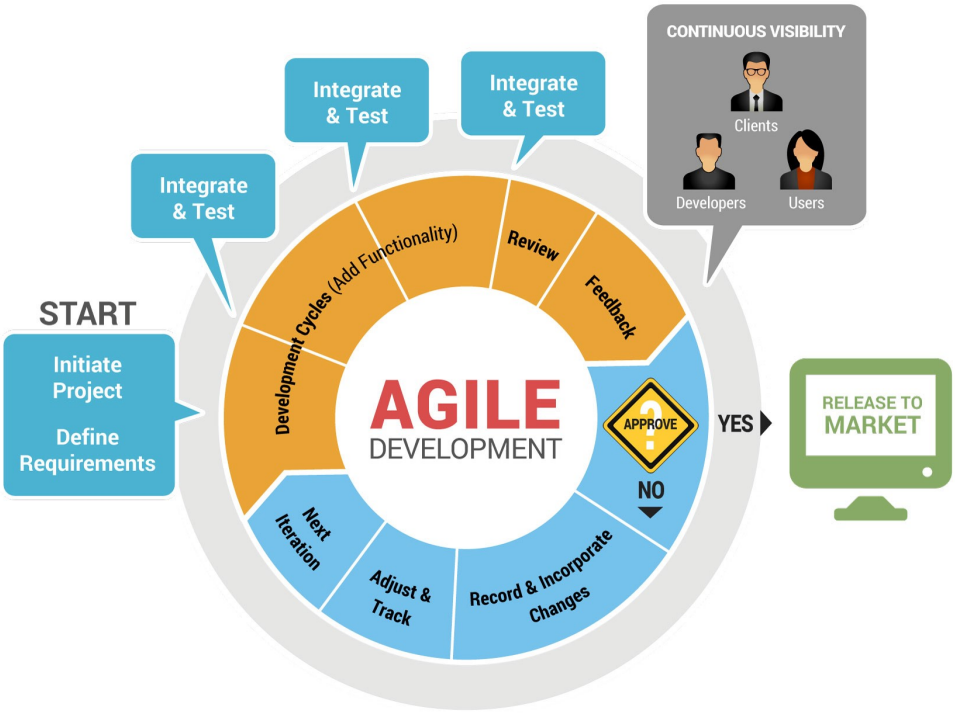
S rozvojem počítačů se výuka přesouvá od klasických principů, jakými byly třeba kreslení výkresů pomocí tuše, k těm elektronickým. Jedná se o rychlejší a pohodlnější přístup k výuce [35]. Proto je důležité stále zdokonalovat nástroje k výuce a dát studentům i možnost samostatně si probíranou látku vyzkoušet. Pro tyto účely jsem vyvinul novou verzi aplikace pro simulaci nelineárních dynamických systémů. Jedná se o simulaci chování částic v jednotlivých systémech podle zadaných parametrů a při daných počátečních podmínkách. Pro zvýšení interaktivnosti aplikace je též možné následně pomocí kliknutí myši udat novou počáteční podmínku pro simulaci. Díky této funkcionalitě je možné též paralelně vykreslovat více různých simulací s různými počátečními podmínkami. Aplikace není limitována platformou spuštění, takže ji může využít uživatel systému Windows, Linux i macOS. Díky použití nejnovějších technologií je dokonce možné zobrazit situaci ve 3D u podivných atraktorů.

1.1 Motivace

Motivací pro vytvoření nové verze aplikace je její jednodušší budoucí rozšiřitelnost o nové systémy, vykreslení situace přehledně ve 3D, paralelní vykreslení simulace systémů s danými parametry a různými počátečními podmínkami, přenositelnost aplikace na jiné platformy a v neposlední řadě i modernizace vzhledu aplikace. Intuitivnost ovládání aplikace a přehlednost jsou samozřejmostí.

1.2 Pracovní postup

Jelikož se jedná pouze o vylepšení původní aplikace (v programovacím jazyce Fortran 90 [6]) jejím přepsáním do modernějšího programovacího jazyka (Java [11]), svou práci začnu analýzou stávající aplikace. Identifikuji její silné (které zachovám) a slabé (které vylepším) stránky. Následně spolu se zadavatelem stanovíme funkční a nefunkční požadavky na novou verzi aplikace a provedu návrh řešení. Tento návrh budu konzultovat se zadavatelem a následně se iterativním způsobem [9] pustím do samotného vývoje nové verze aplikace.



Obrázek 1.1: Iterativní způsob vývoje [10]

Kapitola 2

Analýza

V této kapitole jsem se zaměřil na analýzu stávající aplikace. Identifikoval jsem její silné a slabé stránky, které jsem následně s doporučeným postupem jejich řešení diskutoval s vedoucím práce. V dalším kroku jsem provedl analýzu použitých systémů pro simulaci, zejména odvození vztahů pro popis fungování daného systému v závislosti na čase. Následně jsem zhodnotil možné přístupy k řešení diferenciálních rovnic pomocí programovacího jazyka Java a vybral z nich ten nejvíce vhodný pro danou problematiku. Také jsem se seznámil s programovací metodou dynamického časového kroku pro automatickou úpravu časového kroku. Pomocí této techniky jsem vyhladil simulaci, což přispělo k její větší realističnosti.

2.1 Identifikace silných a slabých stránek stávající aplikace

Pro přehlednější znázornění a kategorizaci jsem svá zjištění rozdělil do dvou tabulek.

2.1.1 Silné stránky

V následující tabulce uvádím silné stránky aplikace. Každý záznam tabulky je uvozen identifikací s počátečním písmenem S (z anglického slova *Strengths* - silné stránky), následovaném číslem, které udává jeho pořadí v tabulce. U každého záznamu je také jeho popis.

Identifikace	Popis
S1	Realistická simulace průběhu
S2	Definování parametrů systému
S3	Definování počátečních podmínek
S4	Definování parametrů modelu
S5	Volba diagramu (fázový, časový vývoj)
S6	Volba samotného systému pro simulaci skrze menu
S7	Možnost udání nových počátečních podmínek kliknutím myši

Tabulka 2.1: Silné stránky aplikace

2.1.2 Slabé stránky

V následující tabulce uvádím slabé stránky aplikace. Každý záznam tabulky je uvozen identifikací s počátečním písmenem W (z anglického slova *Weaknesses* - slabé stránky), následovaném číslem, které udává jeho pořadí v tabulce. U každého záznamu je také jeho popis.

Identifikace	Popis
W1	Nepřenositelnost aplikace na jiné operační systémy (Linux, MacOS)
W2	Nemožnost zobrazení 3D modelu
W3	U časového diagramu není možnost volby proměnné pro vykreslení
W4	Nemožnost vykreslení jiného řezu u vícedimensionálních systémů
W5	Nemožnost postupného vykreslování simulace
W6	Grafické nerozlišení různých simulací jednoho systému
W7	Chybějící validace na přípustné hodnoty zadané uživatelem

Tabulka 2.2: Slabé stránky aplikace

2.2 Analýza použitých systémů pro simulaci

V této kapitole popíšu blíže všechny použité dynamické systémy v aplikaci. Pro první verzi aplikace jsme spolu s vedoucím práce vybrali následující systémy

- Nelineární oscilátor
- Van der Polův oscilátor
- Rösslerův atraktor
- 2D bruselátor
- 4D bruselátor
- Dravec - kořist
- Elektron v periodickém poli
- Lorenzův atraktor

2.2.1 Nelineární oscilátor

Ne každý oscilátor má přesně parabolický průběh energie, který vede na lineární závislost mezi silou a výchylkou (tzv. harmonický oscilátor). Většina oscilací není harmonická a nevede na sinové a kosinové kmity. Příkladem mohou být jak různé elektronické oscilátory, tak obyčejná pružina vytažená za mez pevnosti. V rovnicích na pravé straně pak přibudou další členy, jednou z možností je systém popsaný následující soustavou rovnic:

$$\frac{d\xi_1}{dt} = \xi_2$$
$$\frac{d\xi_2}{dt} = -\xi_1 + \alpha\xi_1^2$$

[34]

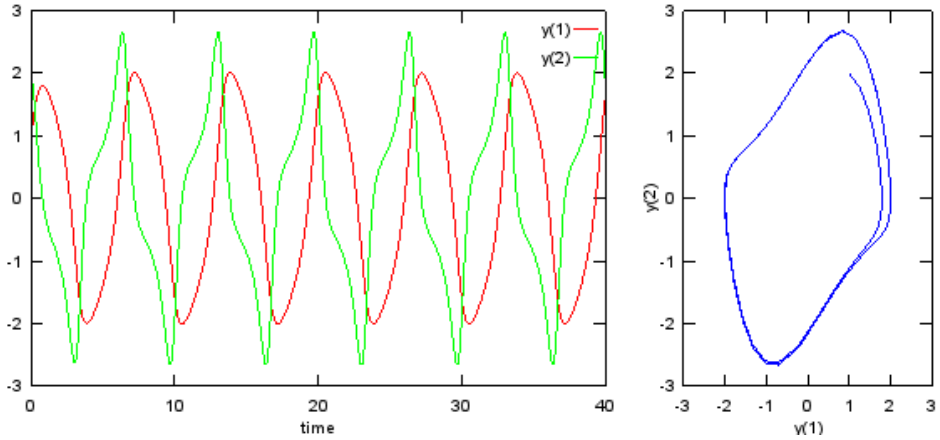
2.2.2 Van der Polův oscilátor

Jedná se o systém popisující chování elektrického obvodu nelineární vakuuové trubice. Model popisující toto chování byl vyvinut nizozemským fyzikem Balthasarem van der Polem. [26]. Systém je popsán následujícími rovnicemi

$$\frac{d\xi_1}{dt} = \xi_2$$
$$\frac{d\xi_2}{dt} = -\xi_1 + \alpha(1 - \delta\xi_1^2)\xi_2; \qquad \delta > 0$$

[34]

Následující obrázek zobrazuje jak fázový, tak i časový vývoj proměnných, kde $y(1) = \xi_1, y(2) = \xi_2$



Obrázek 2.1: Fázový a časový diagram Van der Polova oscilátoru [18]

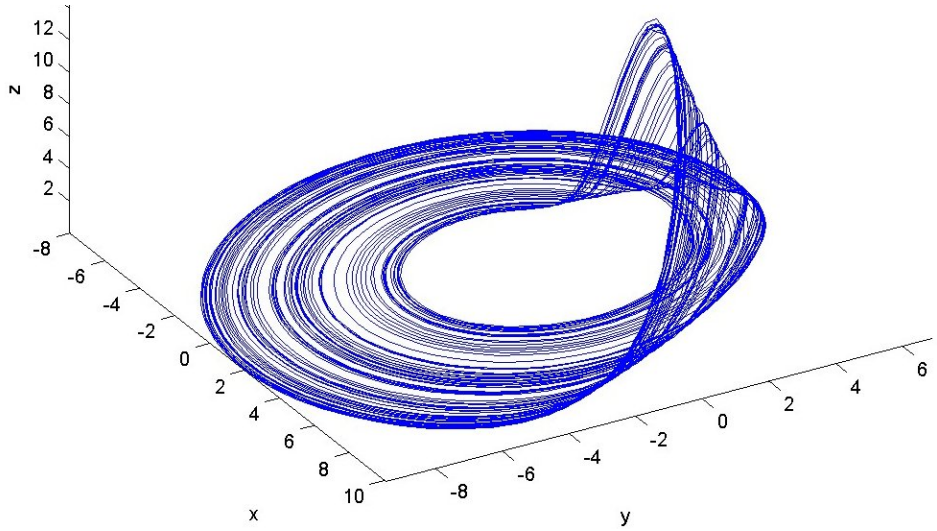
2.2.3 Rösslerův atraktor

Jedná se o atraktor pro Rösslerův systém tří diferenciálních rovnic popisujících dynamický systém ve spojitém čase s chaotickým chováním [21]. Atraktor je popsán následujícími rovnicemi

$$\frac{d\xi_1}{dt} = \xi_2 - \alpha\xi_3$$
$$\frac{d\xi_2}{dt} = -\xi_1 + \beta\xi_2$$
$$\frac{d\xi_3}{dt} = 1 + \xi_1\xi_3 - \gamma\xi_3$$

[20]

Následující obrázek zobrazuje fázový diagram Rösslerova atraktoru



Obrázek 2.2: Fázový diagram Rösslerova atraktoru [14]

2.2.4 Bruselátor 2D

Jedná se o teoretický model autokatalytické chemické reakce (reakce, ve které jako katalyzátor působí jeden z produktů reakce [1]), kterou jako první popsal Ilya Prigogine, belgicko-americký fyzikální chemik v roce 1968. Bruselátor je minimální matematický model zahrnující oscilační chování [2].

$$
\begin{aligned}
A &\xrightarrow{k_1} X \\
B + X &\xrightarrow{k_2} Y + D \\
2X + Y &\xrightarrow{k_3} 3X \\
X &\xrightarrow{k_4} E
\end{aligned}
$$

kde $k_1, ..., k_4$ jsou rychlosti reakcí, koncentraci výchozích látek a produktů označíme $n_A, ..., n_D$. Proměnné budou koncentrace látek X a Y: $\xi_1 = n_X, \xi_2 = n_Y$. Z tvaru reakcí sestavíme soustavu diferenciálních rovnic

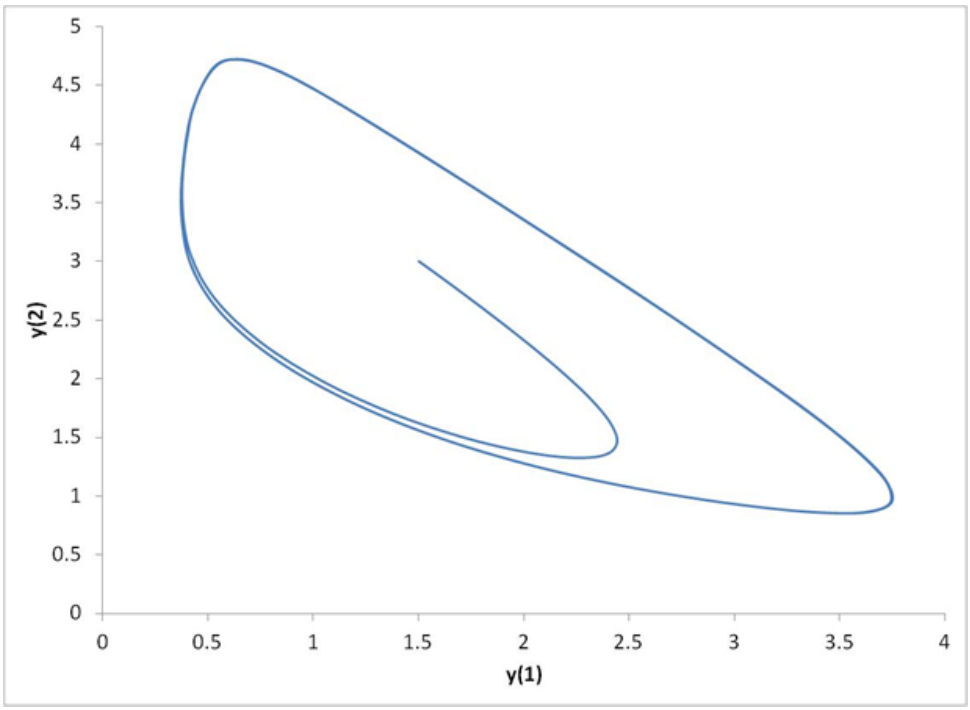
$$\frac{d\xi_1}{dt} = k_1 n_A - k_2 n_B \xi_1 + k_3 \xi_1^2 \xi_2 - k_4 \xi_1$$
$$\frac{d\xi_2}{dt} = k_2 n_B \xi_1 - k_3 \xi_1^2 \xi_2$$

Po zanedbání nepodstatných konstant dostaneme výsledné vztahy

$$\frac{d\xi_1}{dt} = \alpha - \beta \xi_1 + \xi_1^2 \xi_2$$
$$\frac{d\xi_2}{dt} = \gamma \xi_1 - \xi_1^2 \xi_2$$

[34]

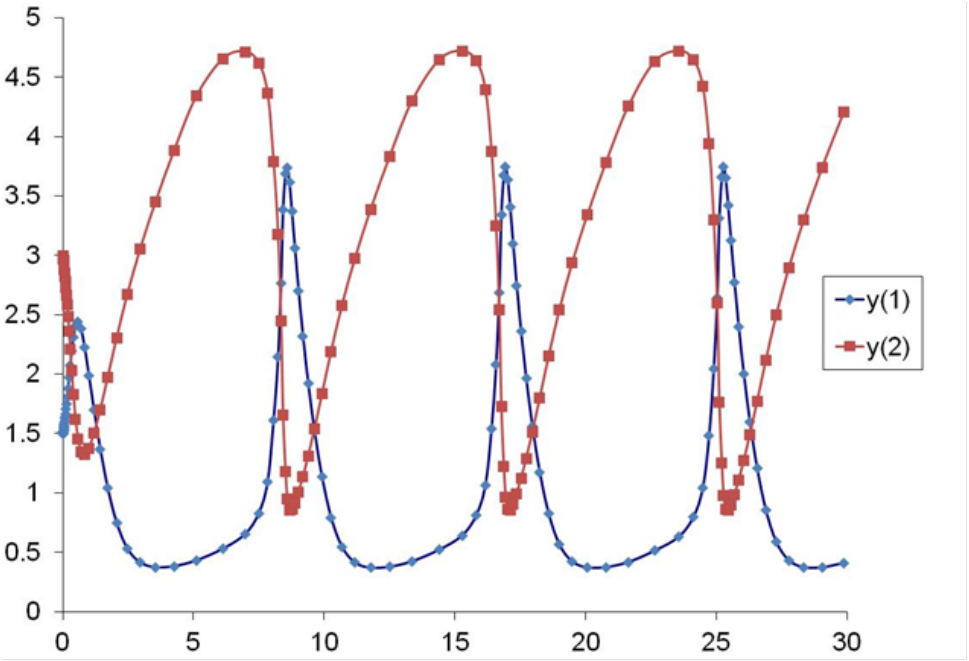
Následující obrázek ukazuje fázový diagram 2D bruselátoru pro $\alpha = 1, \beta = 3$.



Obrázek 2.3: Fázový diagram 2D bruselátoru [36]

K němu též odpovídající časový vývoj obou proměnných, kde $y(1) =$

$\xi_1, y(2) = \xi_2$.



Obrázek 2.4: Časový diagram 2D bruselátoru [36]

2.2.5 Bruselátor 4D

Předpokládejme, že reakce z kapitoly 2.2.4 probíhá současně ve dvou reaktorech s možností výměny látky X rychlostí δ_1 a látky Y rychlostí δ_2. Koncentrace látek X a Y v reaktorech 1 a 2 označíme takto: $\xi_1 = n_{X1}, \xi_2 = n_{Y1}, \xi_3 = n_{X2}, \xi_4 = n_{Y4}$. Výchozí rovnice budou

$$\begin{aligned}
\frac{d\xi_1}{dt} &= \alpha - (\beta + 1)\xi_1 + \xi_1^2\xi_2 + \delta_1(\xi_3 - \xi_1) \\
\frac{d\xi_2}{dt} &= \beta\xi_1 - \xi_1^2\xi_2 + \delta_2(\xi_4 - \xi_2) \\
\frac{d\xi_3}{dt} &= \alpha - (\beta + 1)\xi_3 + \xi_3^2\xi_4 + \delta_1(\xi_1 - \xi_3) \\
\frac{d\xi_4}{dt} &= \beta\xi_3 - \xi_3^2\xi_4 + \delta_2(\xi_2 - \xi_4)
\end{aligned}$$

[34]

V programu jsou zvoleny konkrétní hodnoty

$$\begin{aligned}\alpha &= 2\\ \beta &= 5,9\\ \delta_1 &= 1,21\\ \delta_2 &= 12,1\end{aligned}$$

2.2.6 Dravec - kořist

Jedná se o systém simulující biologické procesy typu dravec-kořist. Systém je založen na 4 základních konstantách

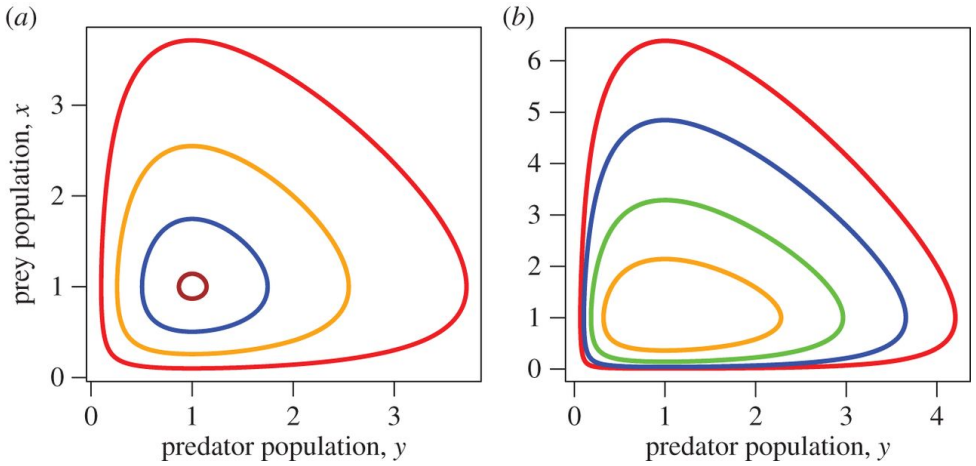
- α - přírustek kořisti
- β - poměr, kterým dravci loví kořisti
- γ - poměr úhynu dravců
- δ - poměr přírustku dravců

Následující rovnice, známé též jako Lotka-Volterra, popisují tento biologický proces

$$\begin{aligned}\frac{d\xi_1}{dt} &= \alpha\xi_1 - \beta\xi_1\xi_2\\ \frac{d\xi_2}{dt} &= -\gamma\xi_2 + \delta\xi_1\xi_2\end{aligned}$$

[13].

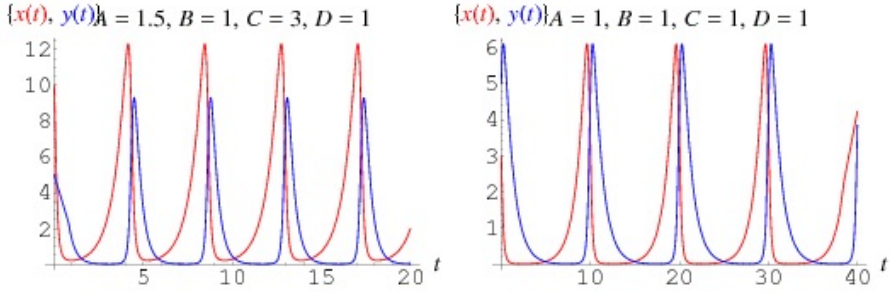
Následující obrázek ukazuje fázový diagram tohoto systému



Obrázek 2.5: Fázový diagram systému Dravec-kořist [38]

K němu také odpovídající časový diagram, kde

$$x(t) = \xi_1, y(t) = \xi_2, A = \alpha, B = \beta, C = \gamma, D = \delta$$



Obrázek 2.6: Časový diagram systému Dravec-kořist [13]

2.2.7 Elektron v periodickém poli

Příkladem pohybu v periodickém poli může být například pohyb nabité částice v krystalové mřížce. Budeme předpokládat, že se částice nachází v poli potenciální energie dané vztahem

$$V(x) = -V_0 \cos \frac{2\pi x}{a}$$

Perioda potenciálu je a a výška je V_0. Příslušné Hamiltonovy rovnice budou:

$$
\begin{aligned}
H &= \frac{p^2}{2m} - V_0 \cos \frac{2\pi x}{a} \\
&\Downarrow \\
\dot{x} &= \{x, H\} = \frac{\delta H}{\delta p} = \frac{p}{m} \\
\dot{p} &= \{p, H\} = -\frac{\delta H}{\delta x} = -\frac{2\pi V_0}{a} \sin \frac{2\pi x}{a}
\end{aligned}
$$

Pokud odhlédneme od nepodstatných konstant, dostaneme se na rovnice ve tvaru

$$
\begin{aligned}
\dot{\xi_1} &= \xi_2 \\
\dot{\xi_2} &= -\sin \xi_1
\end{aligned}
$$

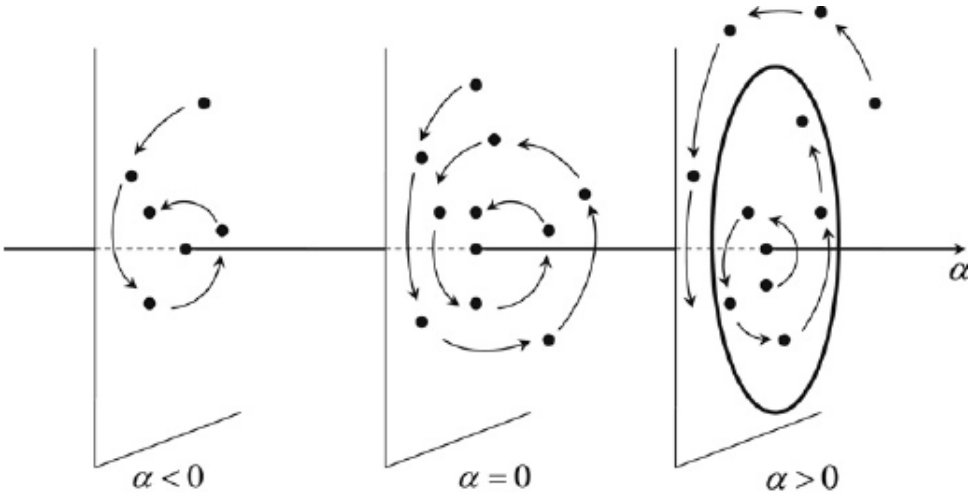
[34]

2.2.8 Hopfova bifurkace

Pojem Hopfova bifurkace (též známý jako Poincaré-Andronov-Hopfova bifurkace) se týká lokálního zrodu a zániku periodického řešení ekvilibria, jakmile parametr překročí kritickou hodnotu [7]. Následující soustava diferenciálních rovnic popisuje pohyb systému v polárních souřadnicích

$$
\begin{aligned}
\dot{r} &= r(\alpha + \delta r^2) \\
\dot{\varphi} &= \omega; \qquad \delta > 0, r \geq 0, \varphi \in R
\end{aligned}
$$

Jedná se o soustavu rovnic pro pohyb systému v polárních souřadnicích [34].



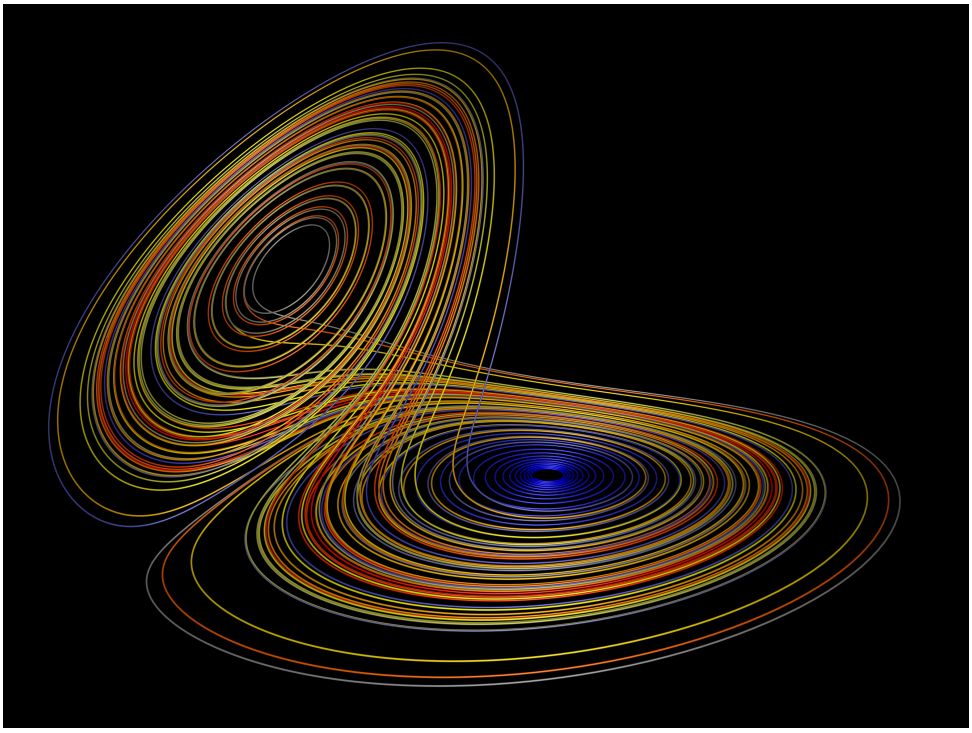
Obrázek 2.7: Hopfova bifurkace [22]

2.2.9 Lorenzův atraktor

Lorenzův atraktor byl odvozený ze zjednodušeného modelu proudění v zemské atmosféře. Prvním, kdo studoval toto chování, byl americký meteorolog Edward Norton Lorenz kolem roku 1963 [12]. Jde o nejznámější příklad podivného atraktoru vyjádřeného výchozí sadou rovnic

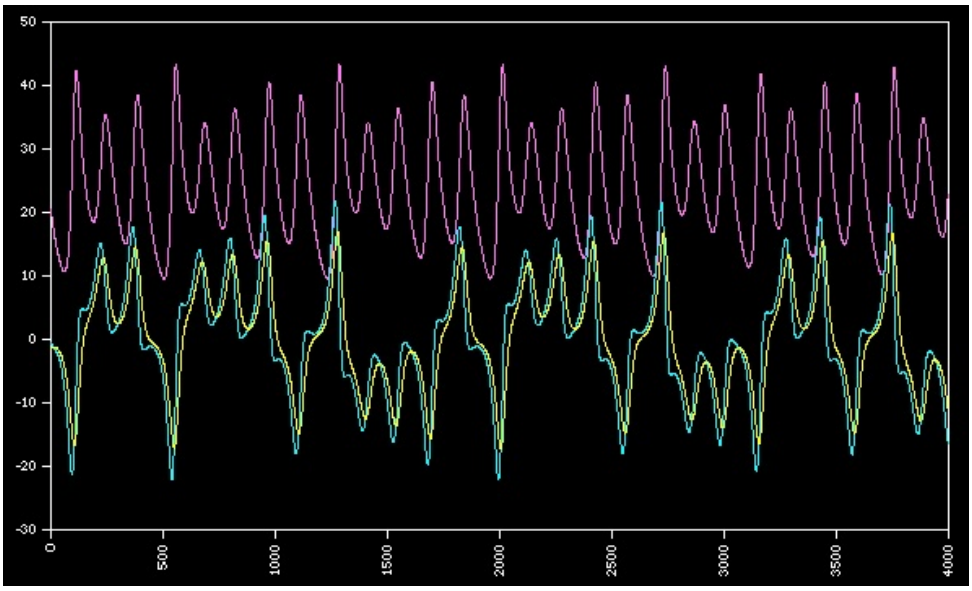
$$\begin{aligned} \frac{d\xi_1}{dt} &= \alpha(\xi_2 - \xi_1) \\ \frac{d\xi_2}{dt} &= -\xi_1\xi_3 + \beta\xi_1 - \xi_2 \\ \frac{d\xi_3}{dt} &= \xi_1\xi_2 - \gamma\xi_3 \end{aligned}$$

[34]. Následující obrázek ukazuje fázový diagram Lorenzova atraktoru



Obrázek 2.8: Fázový diagram Lorenzova atraktoru [12]

Příklad časového vývoje Lorenzova atraktoru



Obrázek 2.9: Časový vývoj Lorenzova atraktoru [12]

2.3 Přístupy k numerickému řešení diferenciálních rovnic

Diferenciální rovnice popisují chování systémů v závisloti na čase. [15] Pro simulaci výše uvedených systémů je nutné strojově řešit diferenciální rovnice. K tomuto účelu existují numerické metody pro řešení diferenciálních rovnic, jejichž hlavní zástupce níže představím a vyberu z nich tu nejvhodnější. U každé metody pro numerické řešení diferenciálních rovnic se jedná pouze o aproximaci, čili jakýsi odhad. Každá metoda má ale jinou přesnost. Existují samozřejmě i jiné metody řešení obyčejných diferenciálních rovnic založené i na zcela odlišných principech. Příkladem mohou být Monte Carlo metody, které hledají řešení rovnic jako optimalizační problém pro nějaký funkcionál. Jiným směrem jsou poruchové metody, které hledají řešení jako odchylky od známého řešení v nějaké základní modelové situaci. Řešení lze také hledat formou nekonečných rozvojů do vhodné báze. Pro naše účely jsou ale výše popsaná diferenční schémata plně dostačující.

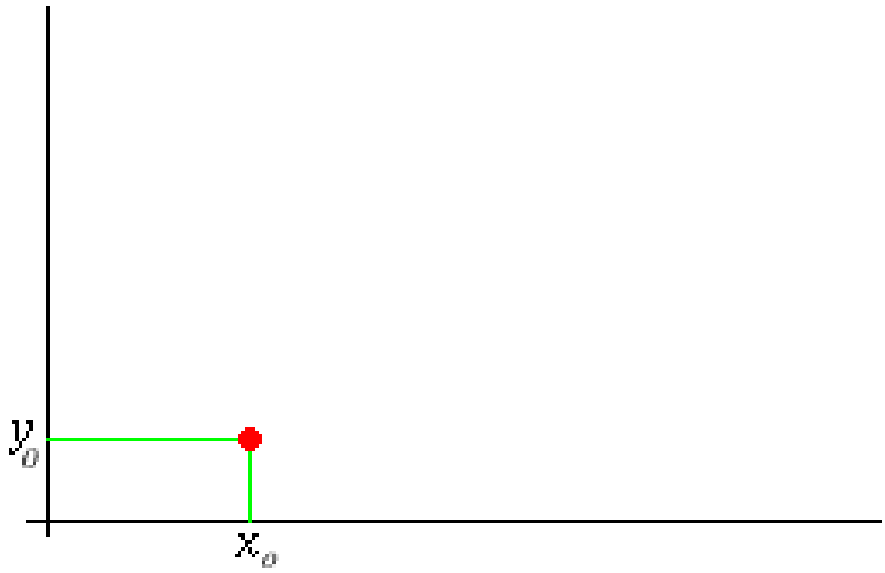
2.3.1 Newtonovo-Eulerovo schéma

Grafické znázornění

Nejjednodušší způsob řešení diferenciálních rovnic. Je založeno na předevení na diferenční rovnice (místo derivací napíšeme jen rozdíly veličin) [33]. Použití Newtonovo-Eulerovy metody ukážu na příkladu:
Mějme diferenciální rovnici ve tvaru:

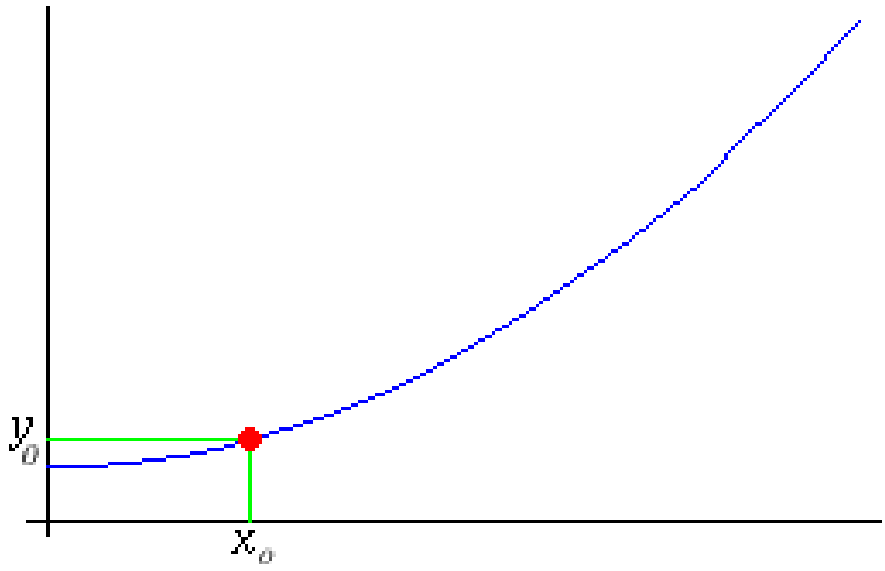
$$y' = f(x, y)$$
$$y(x_0) = y_0$$

Když se podíváme na zadání diferenciální rovnice, tak známe skutečnou hodnotu pro jeden bod. Jedná se o bod daný počáteční podmínkou x_0, ve kterém je hodnota funkce y_0. Můžeme si tuto situaci znázornit pomocí grafu:



Obrázek 2.10: Počáteční podmínka [16]

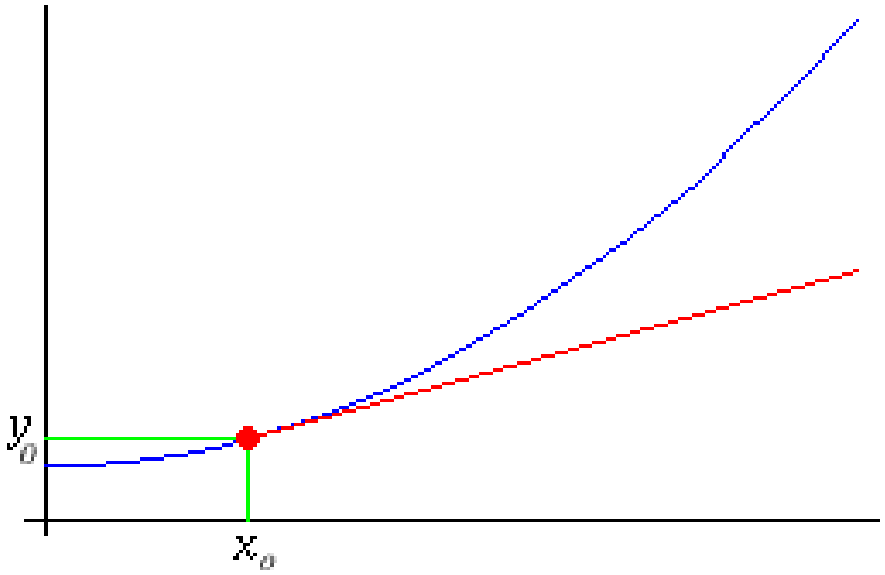
Ve skutečnosti neznáme skutečný průběh funkce $f(x, y)$, ale dejme tomu, že ho známe. Předpokládejme, že skutečný průběh funkce je znázorněn modrým grafem:



Obrázek 2.11: "Skutečný průběh funkce"[16]

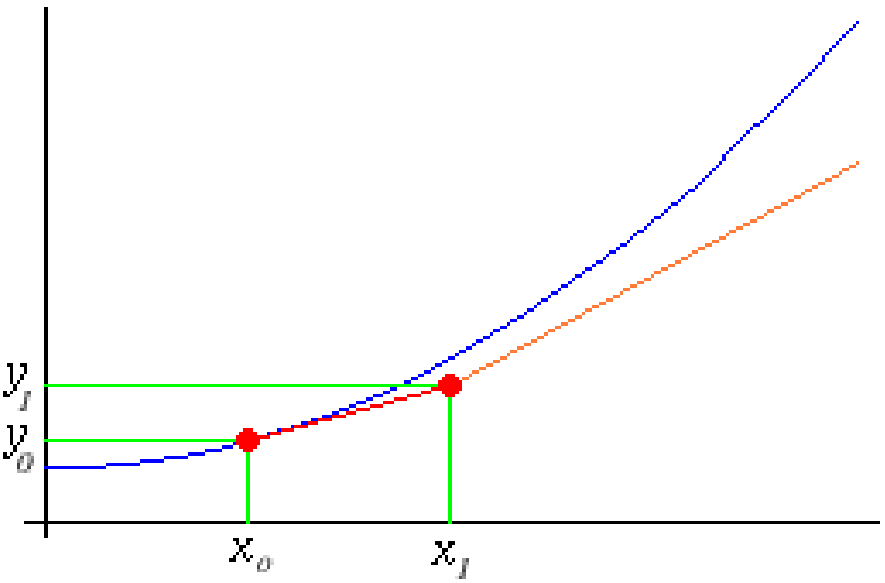
Nyní je potřeba najít způsob, jakým najdeme další body řešení (resp. další

hodnoty funkce $f(x, y)$). Využijeme způsobu, že v bodě x_0 sestrojíme tečnu ke skutečnému (neznámému) průběhu funkce (znázorněnou červeně)



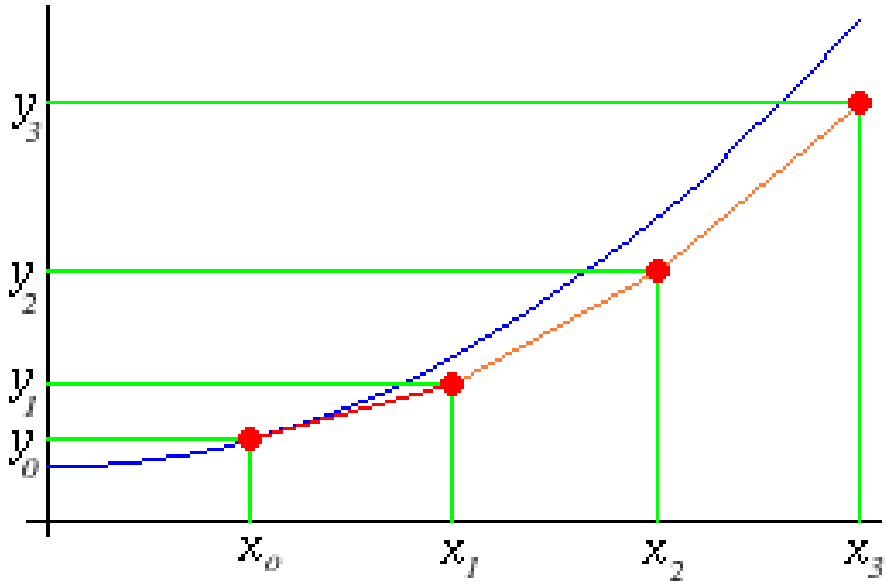
Obrázek 2.12: Tečna ke skutečnému (neznámému) průběhu funkce [16]

Pohledem na tento obrázek můžeme zjistit, že pokud se nepohneme o velký kus po tečně, neodchýlíme se od skutečného průběhu funkce o tolik. Posuneme-li se tedy po tečně do bodu x_1 dostaneme další bod řešení aproximace



Obrázek 2.13: x_1 [16]

Budeme-li postup konstrukce tečen v aproximačních bodech opakovat dostatečně dlouho, dostaneme celkovou aproximaci průběhu



Obrázek 2.14: Postupná konstrukce aproximace [16]

Odvození vztahu

V této kapitole budu používat značení

- (x_n, y_n) pro známý bod
- (x_{n+1}, y_{n+1}) pro bod následující

Rovnice popisující pohyb po ose X je přímočará:

$$x_{n+1} = x_n + h$$

kde h je velikost pohybu po ose X. Dále můžeme vyjádřit vztah pro sklon tečny ke grafu funkce jako

$$f(x_n, y_n) = \frac{\Delta y}{h}$$

Z této rovnice vyjádříme Δy jako

$$\Delta y = h f(x_n, y_n)$$

Z předchozí rovnice je zřejmé, že

$$y_{n+1} = y_n + h f(x_n, y_n)$$

V tuto chvíli už máme popsané pohyby po obou osách X a Y.

[17]

Zhodnocení metody

Tento postup se zdá být jednoduchý a přímočarý, ale má své problémy. Po nějaké době se generovaná trajektorie začne poněkud lišit od skutečné trajektorie a simulace začne být nepřesnou. Důvodem jsou jednak zaokrouhlovací chyby a jednak nahrazení derivací diferencemi. Tato chybovost se dá ovlivnit například volbou bezrozměrných proměnných, případně volbou menšího časového kroku. Tento časový krok nemusí být v průběhu celé simulace konstantní. Vždy jde o kompromis mezi počtem provedených operací a dosaženou přesností [33]. Největším problémem je, že metoda je prvního řádu a chyba roste lineárně s časem, tj. jako $O(\Delta t)$.

2.3.2 Skákající žába aneb Leap-Frog schéma

Problémy Newtonovo-Eulerova schématu lze celkem snadno napravit. Při výpočtu nové hodnoty y_{n+1} využijeme hodnoty y v "mezičase"$t_{n/2}$, stejně tak při výpočtu nové hodnoty x_{n+1} využijeme hodnoty x v "mezičase"$t_{n/2}$. Tento

postup má ale jednu nevýhodu. Na počátku musíme spočítat hodnotu y v čase $t_0 - \Delta t/2$. Až poté lze počítat nové hodnoty, podobně jako v Newtonově-Eulerově schématu [33]. Tato metoda je druhého řádu, chyba roste s druhou mocninou použitého časového kroku, tj. jako $O((\Delta t)^2)$.

Zhodnocení metody

Tato metoda je ve fyzice plazmatu velmi oblíbená. Ponechává si dobré vlastnosti Newtonova-Eulerova schématu. Jeho přesnost je ale na rozdíl od Newtonova-Eulerova schématu o řád vyšší [33].

2.3.3 Runge-Kutta

Pokud máme vyšší požadavky na přenost a kvalitu simulace, musíme využít některé z numerických schémat s vyšším řádem přesnosti. Jedním z nejoblíbenějších schémat je právě Runge-Kuttova metoda 4. řádu.
Předpokládejme, že máme všechny rovnice převedeny na diferenciální rovnice prvního řádu:

$$\frac{d\xi_k}{dt} = f_k(t, \xi_1, ...\xi_N)$$

Nejprve určíme pro každou proměnnou ξ_k v čase t čtveřici konstant

$$\begin{aligned}
K_{1,k} &= f_k(t, \xi_1(t), ..., \xi_N(t)) \\
K_{2,k} &= f_k(t + \frac{1}{2}\Delta t, \xi_1(t) + \frac{1}{2}K_{1,1}\Delta t, ..., \xi_N(t) + \frac{1}{2}K_{1,N}\Delta t) \\
K_{3,k} &= f_k(t + \frac{1}{2}\Delta t, \xi_1(t) + \frac{1}{2}K_{2,1}\Delta t, ..., \xi_N(t) + \frac{1}{2}K_{2,N}\Delta t) \\
K_{4,k} &= f_k(t + \frac{1}{2}\Delta t, \xi_1(t) + K_{3,1}\Delta t, ..., \xi_N(t) + K_{3,N}\Delta t)
\end{aligned}$$

Tyto konstanty napočítáme v uvedeném pořadí po jednom, z důvodu jejich vzájemné závislosti, pro každou hledanou proměnnou. Numerické řešení v čase $t + \Delta t$ dostaneme ze vztahů

$$\xi_k(t+\Delta t) \cong \xi_k(t) + \frac{1}{6}(K_{1,k} + 2K_{2,k} + 2K_{3,k} + K_{4,k})\Delta t; \qquad k = 1, ..., N$$

Nyní známe řešení v čase $t + \Delta t$ a postup můžeme iterativně opakovat [33].

Zhodnocení metody

Pro obecnou aproximaci řešení diferenciálních rovnic pomocí počítače je tato metoda nejvhodnější ze zkoumaných metod. Jedná se o velmi dobrý poměr mezi kvalitou a přesností simulace a výpočetní náročností. Proto jsem také tuto metodu vybral jako aproximační metodu pro simulace ve své diplomové práci.

2.3.4 Borisovo-Bunemanovo schéma

Borisovo-Bunemanovo schéma se používá pro výpočty trajektorií nabitých částic v elektrickém a magnetickém poli. Pro daný problém bylo také původně navrženo. Jeho princip spočívá v tom, že je spočítána polovina urychlení v elektrickém poli, následuje rotace částice v magnetickém poli a v posledním kroku je uskutečněna zbývající část urychlení v elektrickém poli [33].

Zhodnocení metody

Tato metoda, díky svému úzkému zaměření, je pro obecné použití nevhodná. Ovšem pro použití k simulaci, pro kterou byla navržena, je velmi účinná.

2.4 Programovací metoda dynamického časového kroku

V aproximacích se velmi často může vyskytovat též numerická nestabilita způsobená velkým časovým krokem. Jde o jev, který znehodnotí celkovou

simulaci. Typicky, pokud bude velký časový krok mezi jednotlivými body simulace, simulace bude velmi nepřesná a odkloní se k řešení, které není řešením původní rovnice. Je tedy velmi vhodné implementovat nějakou techniku pro automatickou korekci časového kroku. Tato technika nám zajistí to, že v oblastech velkého výkyvu simulace (oblasti s náhlou změnou směru pohybu) bude simulace stále korektní. Velmi častou technikou je tzv. metody dynamického časového kroku [37]. Technika je založená na následujícím principu: Zvolíme přesnost ε, se kterou chceme počítat v simulaci a jednou za několik kroků (např. 10) se zkusí simulace posunout jak o Δt, tak i dvakrát o $\Delta\frac{t}{2}$. Porovnáme mezi sebou oba dva posuny a podle výsledku se rozhodneme:

- pokud je rozdíl posunů o Δt a o $\Delta\frac{t}{2}$ o řád vyšší než zadané ε, víme, že časový krok je velký a snížíme ho na polovinu. Dále také zahazujeme poslední krok simulace a tento přepočítáme znovu s poloviční časovým krokem
- pokud je rozdíl posunů o Δt a o $\Delta\frac{t}{2}$ o řád nižší než zadané ε, víme, že časový krok je zbytečně malý a zvýšíme ho na dvojnásobek

Rozdíl posuvů se spočítá následovně:

Označíme A jako souřadnice posuvu o Δt a B jako souřadnice posuvu o dvakrát $\Delta\frac{t}{2}$.

$$A = (a_1, a_2, ..., a_n)$$
$$B = (b_1, b_2, ..., b_n)$$

Potom celková vzdálenost je:

$$\Delta = \sqrt{\frac{\sum_{i=1}^{n}(b_i - a_i)^2}{\sum_{i=1}^{n}(a_i)^2}}$$

Pozn.: Ve jmenovateli uvnitř odmocniny je též možné zvolit i souřadnice posuvu B

Tuto vzdálenost (označenou jako Δ) porovnáme se zadaným ε a podle výsledku se rozhodneme, jak dále postupovat (viz. výše 2.4).

Kapitola 3

Návrh řešení

Ve fázi návrhu jsem se zadavatelem definoval funkční a nefunkční požadavky na aplikaci. Tyto jsem zpracoval do přehledných tabulek a s přihlédnutím k těmto požadavkům jsem vytvořil architekturu celé aplikace.

3.1 Požadavky na aplikaci

Požadavky na aplikaci jsem s pomocí vedoucího práce definoval s přihlédnutím k již stávajícímu řešení aplikace. Jak jsem psal v úvodu práce, hlavním požadavkem bylo zachovat stávající funkcionalitu aplikace a tuto vylepšit o nové prvky.

3.1.1 Funkční požadavky

V následující tabulce uvádím definované funkční požadavky na aplikaci. Funkčním požadavkem rozumíme takový požadavek na aplikaci, který definuje, co má aplikace dělat [29]. Každý funkční požadavek jsem doplnil o jeho jednoznačný identifikátor. Tento identifikátor pomůže v závěru práce zhodnotit, zda byl požadavek splněn, či nikoliv.

Identifikace	Popis
FR1	Zachovat stávající funkcionalitu aplikace
FR2	Možnost 3D simulace
FR3	Validovat vstupy od uživatele do aplikace
FR4	Výběr řezu do 2D grafu pro vícedimensionální systémy
FR5	Výběr proměnné pro znázornění časového průběhu
FR6	Postupné vykreslení simulace definovanou rychlostí

Tabulka 3.1: Funkční požadavky na aplikaci

3.1.2 Nefunkční požadavky

V následující tabulce uvádím definované nefunkční požadavky na aplikaci. Nefunkčním požadavkem rozumíme takový požadavek na aplikaci, který definuje, jak se má aplikace chovat [29]. Každý nefunkční požadavek jsem doplnil o jeho jednoznačný identifikátor. Tento identifikátor pomůže v závěru práce zhodnotit, zda byl požadavek splněn, či nikoliv.

Identifikace	Popis
NR1	Přenositelnost mezi různými operačními systémy
NR2	Paralelní vykreslování různých simulací
NR3	Distribuce ve formátu .jar jako stand-alone verze
NR4	Jednoduché rozšíření aplikace o nové systémy

Tabulka 3.2: Nefunkční požadavky na aplikaci

3.2 Architektura aplikace

V závěrečné fázi návrhu aplikace jsem se zaměřil na samotnou architekturu. S přihlédnutím k definovaným funkčním a nefunkčním požadavkům jsem se nakonec rozhodl pro architekturu MVC.

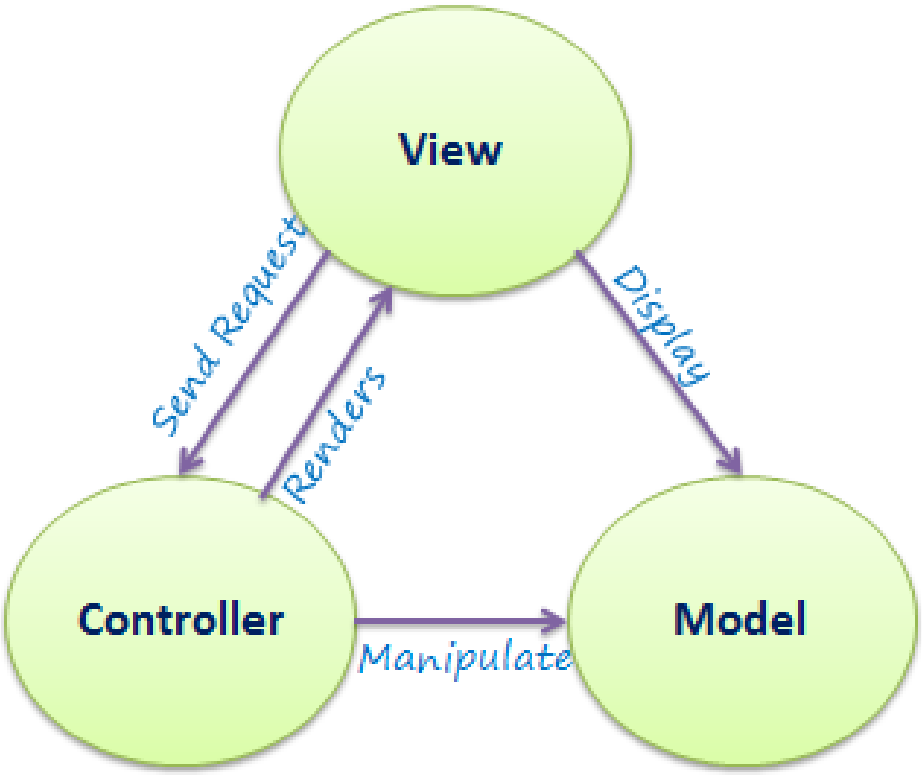
3.2.1 MVC

MVC architektonický vzor je obecně používaný architektonický vzor pro návrh aplikací s uživatelským prostředím [28]. MVC rozděluje aplikaci do 3 částí:

- Model

Reprezentuje data a business logiku celé aplikace. Na této vrstvě se provádí hlavní logika kódu a funkcionalita celé aplikace.

- View
  Prezentační vrstva. Tato vrstva se používá pro zobrazení dat uživateli a také dovoluje uživateli manipulovat jak s daty, tak s akčními prvky (tlačítky).
- Controller
  Obstarává reakci na události v aplikaci. Typicky uživatel klikne na tlačítko ve View vrstvě, vyvolá se událost, kterou zachytí a zpracuje právě Controller. Tento také po zpracování požadavku vytvoří nové View s daty pro uživatele.



Obrázek 3.1: MVC architektura [28]

Kapitola 4

Implementace zvoleného řešení

4.1 Obecné informace k implementaci

V této sekci popíšu obecné informace týkající se implementace.

4.1.1 Programovací jazyk

Pro svou diplomovou práci jsem spolu s vedoucím práce zvolil programovací jazyk Java ve verzi 8. Tento jazyk jsme zvolili z důvodu nezávislosti na operačním systému, velké rozšířenosti a oblíbenosti [31] a také kvůli široké škále frameworků dostupných pro pohodlnější práci. Nejdůležitějším modulem, který jsem ve své práci využil, byl bezesporu JavaFX, modul pro tvorbu grafických aplikací [32]. Z tohoto modulu jsem použil funkce pro tvorbu desktopové aplikace, jejích oken a hlavně 2D grafů.

4.1.2 Systém pro správu a sestavení aplikace

Jako systém pro získávání tzv. 3rd party závislostí (závislostí na projektech třetích stran) jsem použil Maven [27] od firmy Apache. Tento software umož-

ňuje popsat závislosti projektu pomocí Project Object Modelu (POM), tyto závislosti sám získá z centrálního repozitáře a přibalí do sestavení projektu.

4.1.3 Spring framework

Pro řízení závislostí mezi jednotlivými třídami aplikace jsem použil framework Spring. Spring framework odstraňuje zbytečné explicitní vytváření objektů tříd, ale pracuje na principu tzv. Dependency Injection kontejneru. Dependency injection kontejner uchovává všechny označené objekty u sebe (tzv. beany) a tyto pak na vyžádání vkládá (injectuje) jako závislost do žádající třídy. Timto také pomáhá k naplnění principu IoC (Inversion of Control) [8].

4.1.4 Knihovna pro tvorbu 3D grafů

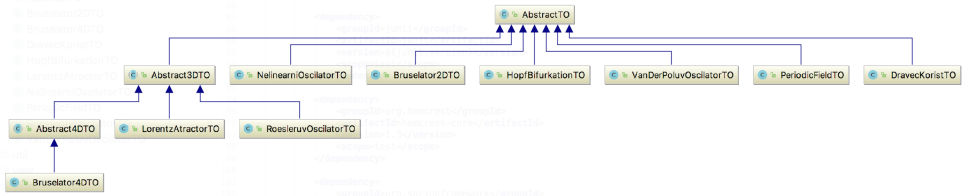
Jelikož JavaFX v základní podobě nemá pro účely aplikace vhodnou implementaci možnosti tvorby 3D grafů, musel jsem využíl knihovnu třetí strany. Po zralé úvaze a testu několika knihoven jsem si zvolil knihovnu JZY3D [19]. Jedná se o Open Source API pro tvorbu 2D grafů v jazyce Java.

4.2 Model

V této sekci ukážu návrh modelové vrstvy aplikace a popíšu účel jednotlivých navrhnutých tříd aplikace.

4.2.1 Balíček pro přenos dat

V tomto balíčku jsem podle návrhového vzoru DTO (Data transfer object) implementoval funkcionalitu přenosu dat mezi jednotlivými komponentami. Transfer object je jednoduché POJO (Plain Old Java Objects), které má pouze atributy, get, set metody a je serializovatelný [3]. Následující diagram zobrazuje hierarchii těchto tříd.



Obrázek 4.1: Data Transfer Objects

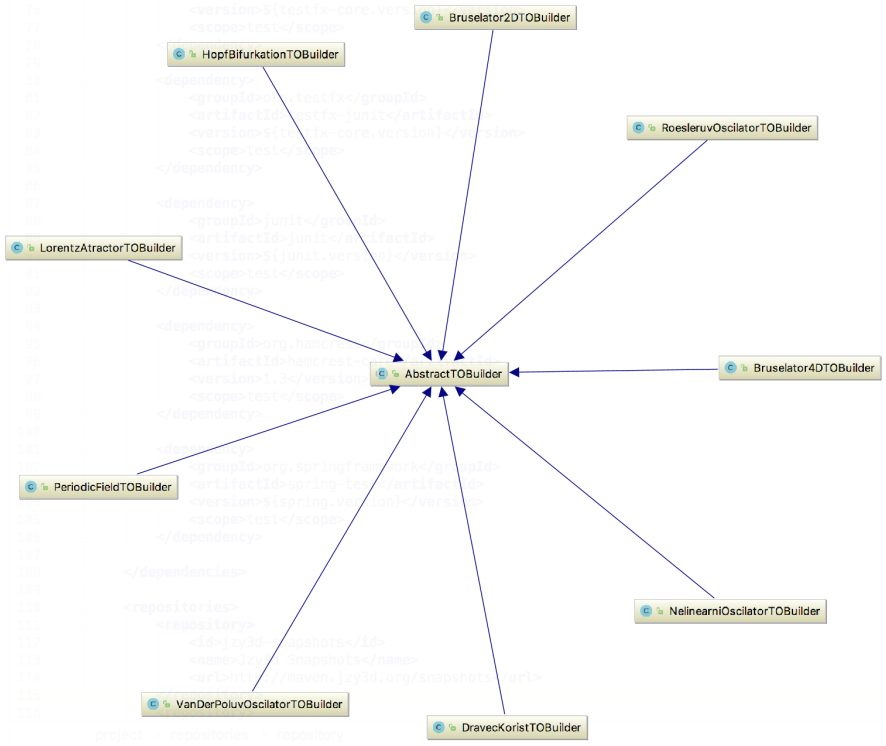
- *AbstractTO*
  Abstraktní předek všech transfer objektů, který v sobě drží informace společné pro všechny ostatní transfer objekty.
- *Abstract3DTO*
  Abstraktní předek všech 3D transfer objektů, který dědí od *AbstractTO* a drží v sobě navíc informace platné pro všechny 3D transfer objekty.
- *Abstract4DTO*
  Abstraktní předek všech 4D transfer objektů, který dědí od *Abstract3DTO* a drží v sobě navíc informace platné pro všechny 4D transfer objekty.
- *Bruselator4DTO*
  Transfer object, dědící od *Abstract4DTO*, který je platný pro systém 4D bruselátoru.
- *LorentzAtractorTO*
  Transfer object, dědící od *Abstract3DTO*, který je platný pro systém Lorentzova atraktoru.
- *RoesleruvOscilatorTO*
  Transfer object, dědící od *Abstract3DTO*, který je platný pro systém Rösslerova oscilátoru.
- *NelinearniOscilatorTO*
  Transfer object, dědící od *AbstractTO*, který je platný pro systém nelineárního oscilátoru.
- *Bruselator2DTO*
  Transfer object, dědící od *AbstractTO*, který je platný pro systém 2D bruselátoru.
- *HopfBifurkationTO*
  Transfer object, dědící od *AbstractTO*, který je platný pro systém Hopfovy bifurkace.
- *VanDerPoluvTO*
  Transfer object, dědící od *AbstractTO*, který je platný pro systém Van der Polova oscilátoru.

- *PeriodicFieldTO*
  Transfer object, dědící od *AbstractTO*, který je platný pro systém elektronu v periodickém poli.

- *DravecKoristTO*
  Transfer object, dědící od *AbstractTO*, který je platný pro systém dravec-kořist.

4.2.2 Balíček pro tvorbu doménových objektů

V tomto balíčku jsem podle návrhového vzoru Builder implementoval logiku tvorby příslušných doménových objektů v závislosti na zvoleném systému. Návrhový vzor Builder tvoří komplexní objekty postupně na základě volání jednoduchých metod [4]. Hlavní logikou, kterou jsem použil v builderech, je převod textové formy vstupu od uživatele do číselné reprezentace ve zdrojovém kódu.



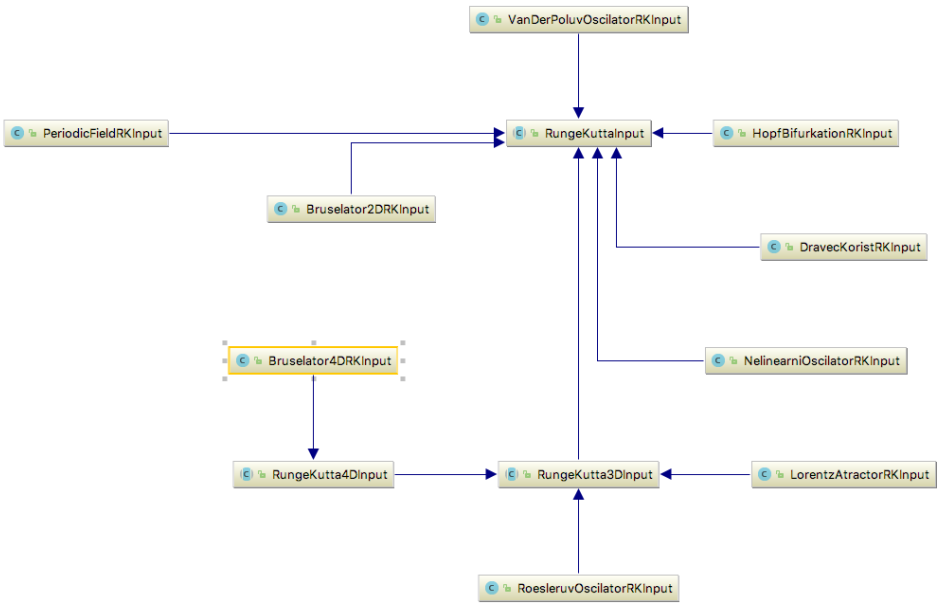
Obrázek 4.2: Builder

4.2.3 Balíček pro matematické výpočty

Tento balíček je rozdělen na 4 logické celky podle jejich zaměření. Každá část má jiný podíl na matematickém výpočtu. Všechny části nyní podrobně proberu.

Vstupní objekty

Objekty představující vstup do matematického výpočtu. Pro každý systém obsahují všechny nutné parametry výpočtu.



Obrázek 4.3: Vstupní objekty pro matematické výpočty

- *RungeKuttaInput*
  Základní vstupní objekt do řešení pomocí metody Runge-Kutta. Obsahuje společné atributy (neznámou x, y, posun času Δt a typ vypočítávaného systému) pro všechny ostatní vstupní objekty.

- *RungeKutta3DInput*
  Základní vstupní objekt do řešení pomocí metody Runge-Kutta pro 3D systémy. Obsahuje navíc neznámou z představující 3. dimensi řešení.

- *RungeKutta4DInput*
  Základní vstupní objekt do řešení pomocí metody Runge-Kutta pro 4D systémy. Obsahuje navíc neznámou w představující 4. dimensi řešení.

- *NelinearniOscilatorRKInput* Vstupní objekt pro výpočet systému Nelineárního oscilátoru. Obsahuje pouze parametr α, viz. též odvození výpočtu 2.2.1.

- *VanDerPoluvOscilatorRKInput* Vstupní objekt pro výpočet systému Van der Polova oscilátoru. Obsahuje pouze parametr α, viz. též odvození výpočtu 2.2.2.

- *RoesleruvOscilatorRKInput* Vstupní objekt pro výpočet systému Rösslerova atraktoru. Obsahuje parametry α, β, γ, viz. též odvození výpočtu 2.2.3.

- *Bruselator2DRKInput* Vstupní objekt pro výpočet systému 2D bruselátoru. Obsahuje parametry α, β, γ, viz. též odvození výpočtu 2.2.4.

- *Bruselator4DRKInput* Vstupní objekt pro výpočet systému 4D bruselátoru. Obsahuje parametry $\alpha, \beta, \delta_1, \delta_2$, viz. též odvození výpočtu 2.2.5. V programu jsou zvoleny konkrétní hodnoty daných parametrů pro praktické znázornění situace:

$$\begin{aligned} \alpha &= 2 \\ \beta &= 5,9 \\ \delta_1 &= 1,21 \\ \delta_2 &= 12,1 \end{aligned}$$

- *DravecKoristRKInput* Vstupní objekt pro výpočet systému Dravec-kořist. Obsahuje parametry $\alpha, \beta, \gamma, \delta$, viz. též odvození výpočtu 2.2.6.

- *PeriodicFieldRKInput* Vstupní objekt pro výpočet systému Elektron v periodickém poli. Neobsahuje žádný parametr, viz. též odvození výpočtu 2.2.7.

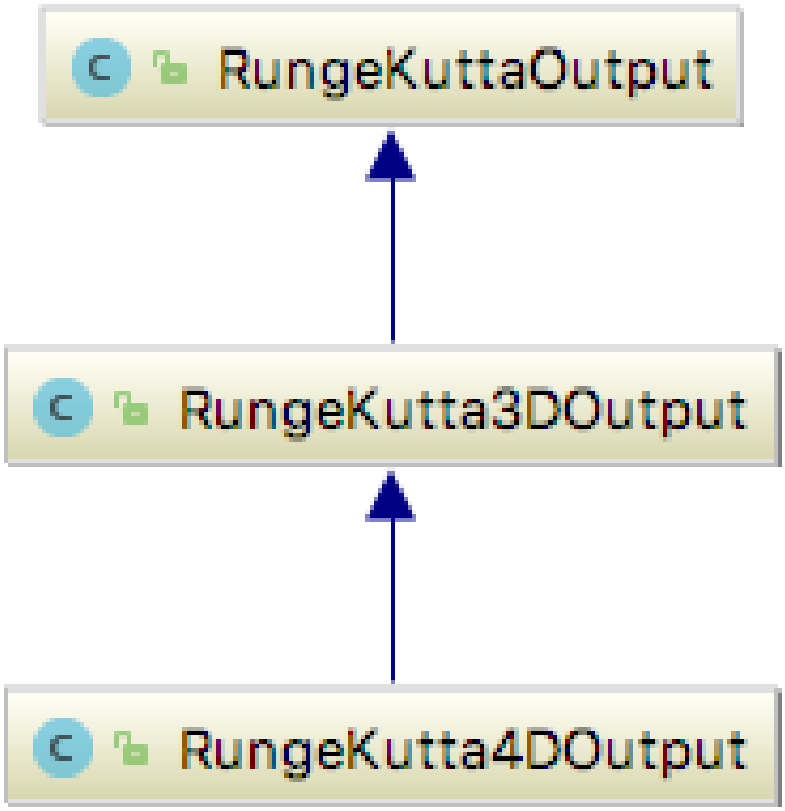
- *HopfBifurkationRKInput* Vstupní objekt pro výpočet systému Hopfovy bifurkace. Obsahuje parametry $\alpha, \delta; \alpha = 1, \delta = 0,01$, viz. též odvození výpočtu 2.2.8.

- *LorentzAtractorRKInput* Vstupní objekt pro výpočet systému Lorenzova atraktoru. Obsahuje parametry α, β, γ, viz. též odvození výpočtu 2.2.9. V programu jsou zvoleny konkrétní hodnoty daných parametrů pro praktické znázornění situace:

$$\alpha = 3$$
$$\beta = 26,5$$
$$\gamma = 1$$

Výstupní objekty

Následující objekty se používají jako výstup po výpočtu metodou Runge-Kutta.



Obrázek 4.4: Výstupní objekty pro matematické výpočty

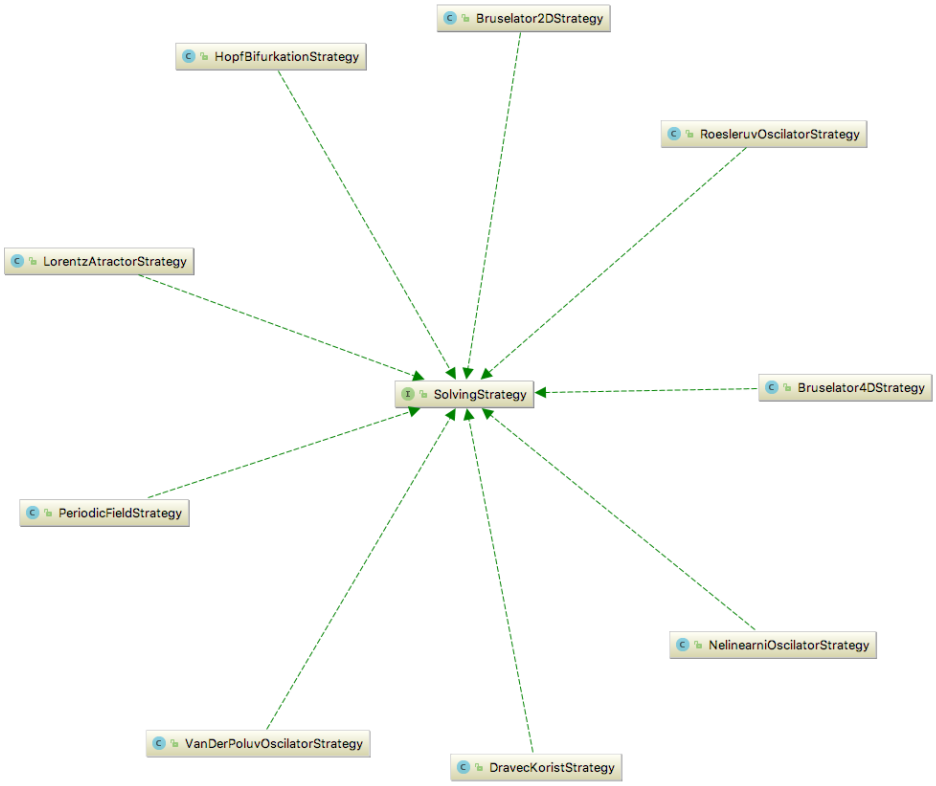
- *RungeKuttaOutput*
  Základní výstupní objekt pro všechny výpočty metodou Runge-Kutta.

Obsahuje nové hodnoty x a y po posuvu o Δt při daných parametrech systému.

- *RungeKutta3DOutput*
  Základní výstupní objekt pro všechny výpočty metodou Runge-Kutta pro 3D systémy. Obsahuje navíc parametr z po posuvu o Δt při daných parametrech systému.
- *RungeKutta4DOutput*
  Základní výstupní objekt pro všechny výpočty metodou Runge-Kutta pro 4D systémy. Obsahuje navíc parametr w po posuvu o Δt při daných parametrech systému.

Balíček pro uskutečnění výpočtů

Balíček pro výpočty ve svém návrhu ctí návrhový vzor Strategy. Návrhový vzor strategy je založen na principu určení aktuálního chování za běhu programu. Jednotlivé třídy reprezentují odlišné strategie chování a za běhu programu je určena ta strategie, která se použije v aktuálním běhu [5].



Obrázek 4.5: Uskutečnění výpočtů

- *SolvingStrategy*
  Interface, který určuje kontrakt všech ostatních strategií. V těchto objektech probíhá samotný výpočet simulace metodou Runge-Kutta. Obsahuje jednu jedinou metodu *solve*(), která spustí danou strategii s parametrem vstupu *RungeKuttaInput* a po dokončení výpočtu vrací *RungeKuttaOutput*.

- *NeliearniOscilatorStrategy*
  Obsahuje implementaci logiky výpočtu z 2.2.1.

- *VanDerPoluvOscilatorStrategy*
  Obsahuje implementaci logiky výpočtu z 2.2.2.

- *RoesleruvOscilatorStrategy*
  Obsahuje implementaci logiky výpočtu z 2.2.3.

- *Bruselator2DStrategy*
  Obsahuje implementaci logiky výpočtu z 2.2.4.

- *Bruselator4DStrategy*
  Obsahuje implementaci logiky výpočtu z 2.2.5.

- *DravecKoristStrategy*
  Obsahuje implementaci logiky výpočtu z 2.2.6.

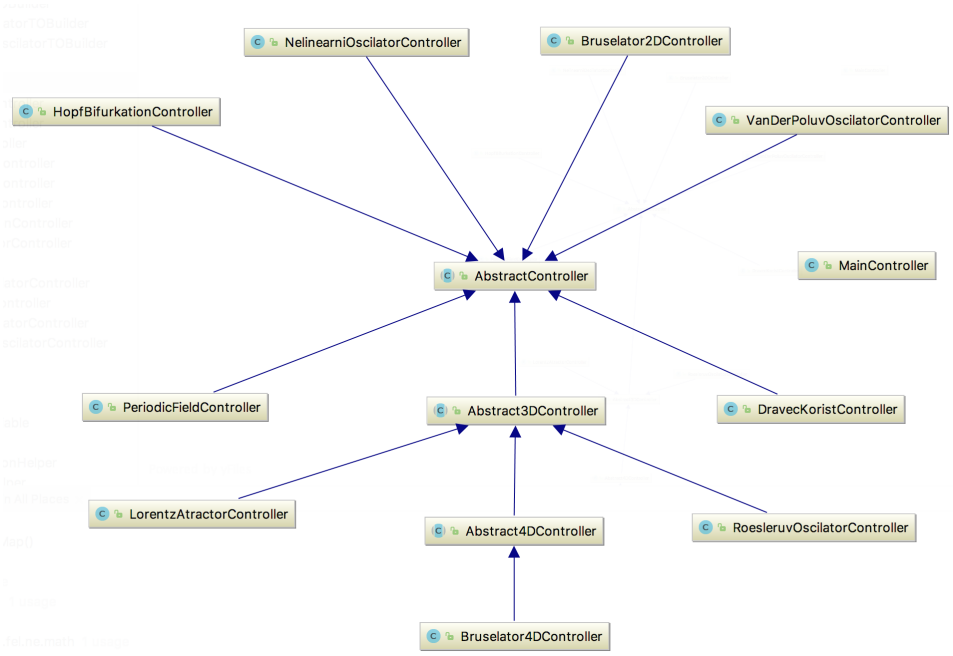
- *PeriodicFieldStrategy*
  Obsahuje implementaci logiky výpočtu z 2.2.7.

- *HopfBifurkationStrategy*
  Obsahuje implementaci logiky výpočtu z 2.2.8.

- *LorentzAtractorStrategy*
  Obsahuje implementaci logiky výpočtu z 2.2.9.

4.2.4 Controller

V této kapitole popíšu vrstvu Controller pro obsluhu klientských požadavků.



Obrázek 4.6: Balíček controller

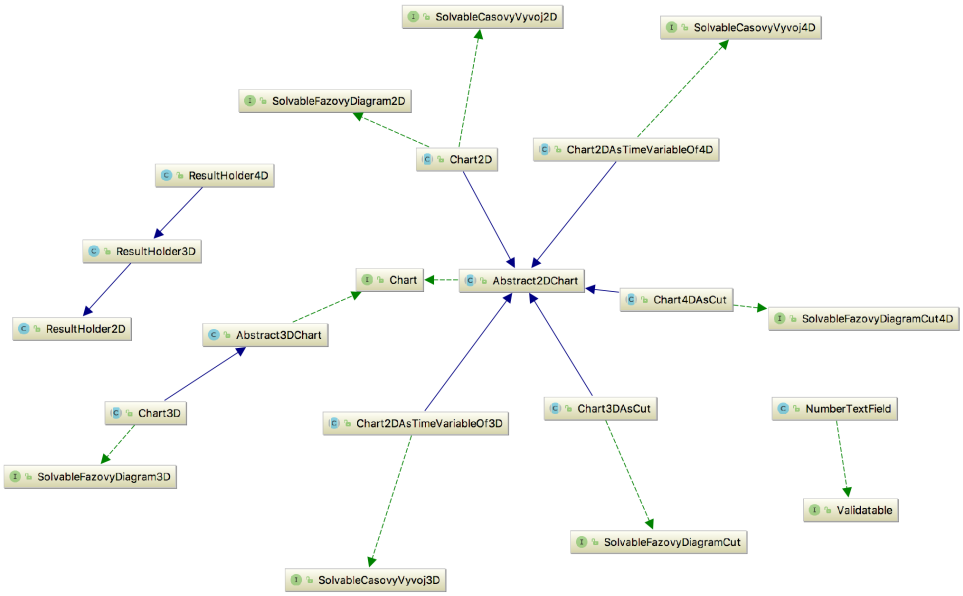
- *AbstractController*
  Controller obsahující společnou funkcionalitu všech ostatních controllerů. Obsahuje mapování společných polí z obrazovky uživatele a další společné chování.

- *Abstract3DController*
  Controller obsahující společnou funkcionalitu pro controllery obsluhující 3D systémy. Obsahuje například mapování výběrového pole pro řez, pole pro počáteční hodnotu z .

- *Abstract4DController*
  Controller obsahující společnou funkcionalitu pro controllery obsluhující 4D systémy. Obsahuje například mapování pole pro počáteční hodnotu w .

4.2.5 View

Poslední vrstvou v návrhu je vrstva prezentační pro uživatele. V této vrstvě najdeme třídy pro vykreslení do FXML souborů. Detailnější rozpis hlavních tříd následuje:



Obrázek 4.7: Balíček view

- *Chart*
  Interface, který definuje kontrakt veškerých grafů. Obsahuje jedinou metodu
  `Type getType();`
  Tato metoda definuje, jakého je graf typu. V současné době je možné v aplikaci vykreslit pouze grafy typu 2D a 3D.

- *Abstract2DChart*
  Abstraktní třída definující společné chování 2D grafů. Obsahuje reakci na kliknutí uživatele do kreslícího okna, automatické nastavení měřítka os, nastavení samotného vzhledu vykreslovaného grafu apod. Definuje jedinou povinnou metodu pro implementaci potomky:
  `EventHandler<MouseEvent> createClickEventHandler(AbstractTO to, DiagramType diagramType);`
  Metoda má za úkol vytvořit handler pro událost kliknutí uživatelem do kreslící plochy grafu. Kliknutím uživatel nastaví novou počáteční podmínku simulace.

- *Chart2D*
  Abstraktní třída dodefinovávající společné chování 2D grafu. Obsahuje 1 abstraktní metodu pro definici potomky:
  `boolean runDynamicStepCheck(ResultHolder2D resultHolder, AbstractTO to);`
  Tato metoda má za úkol provést kontrolu, zda se simulace neodchýlila od skutečného výpočtu o více, než je parametrem dovoleno. Návratová hodnota je typu `boolean`, která má hodnotu **true** ve chvíli, kdy se

výpočet odchýlil o více, než je parametrem dovoleno a **false** ve chvíli, kdy je výpočet stále v normě.

- *Chart2DAsTimeVariableOf3D*
  Abstraktní třída dodefinovávající společné chování 2D grafu, který je použit jako graf pro vykreslení vybrané časové proměnné 3D grafu. Obsahuje 1 abstraktní metodu pro definici potomky:
  `boolean runDynamicStepCheck(ResultHolder3D resultHolder, Abstract3DTO to);`
  Tato metoda má za úkol provést kontrolu, zda se simulace neodchýlila od skutečného výpočtu o více, než je parametrem dovoleno. Návratová hodnota je typu `boolean`, která má hodnotu **true** ve chvíli, kdy se výpočet odchýlil o více, než je parametrem dovoleno a **false** ve chvíli, kdy je výpočet stále v normě.

- *Chart2DAsTimeVariableOf4D*
  Abstraktní třída dodefinovávající společné chování 2D grafu, který je použit jako graf pro vykreslení vybrané časové proměnné 4D grafu. Obsahuje 1 abstraktní metodu pro definici potomky:
  `boolean runDynamicStepCheck(ResultHolder4D resultHolder, Abstract4DTO to);`
  Tato metoda má za úkol provést kontrolu, zda se simulace neodchýlila od skutečného výpočtu o více, než je parametrem dovoleno. Návratová hodnota je typu `boolean`, která má hodnotu **true** ve chvíli, kdy se výpočet odchýlil o více, než je parametrem dovoleno a **false** ve chvíli, kdy je výpočet stále v normě.

- *SolvableFazovyDiagram2D*
  Interface, který umožňuje na grafu zobrazit jeho fázový vývoj ve 2D. Definuje 1 metodu:
  `void solveFazovyDiagramIteration(ResultHolder2D resultHolder, AbstractTO to, XYChart.Series<Number, Number> series);`
  Tato metoda má za úkol spočítat jednu iteraci pro zobrazení fázového vývoje.

- *SolvableFazovyDiagram3D*
  Interface, který umožňuje na grafu zobrazit jeho fázový vývoj ve 3D. Definuje 1 metodu:
  `void solveFazovyDiagramIteration(ResultHolder3D resultHolder3D, Abstract3DTO to, List<Coord3d> series);`
  Tato metoda má za úkol spočítat jednu iteraci pro zobrazení fázového vývoje.

- *SolvableFazovyDiagramCut*
  Interface, který umožňuje na 3D grafu zobrazit fázový vývoj řezu dvou proměnných ve 2D. Definuje 1 metodu:
  `void solveFazovyDiagramIteration(ResultHolder3D resultHolder3D, AbstractTO to, XYChart.Series<Number, Number> series);`

Tato metoda má za úkol spočítat jednu iteraci pro zobrazení fázového vývoje.

- *SolvableFazovyDiagramCut4D*
  Interface, který umožňuje na 4D grafu zobrazit fázový vývoj řezu dvou proměnných ve 2D. Definuje 1 metodu:
  `void solveFazovyDiagramIteration(ResultHolder4D resultHolder4D, AbstractTO to, XYChart.Series<Number, Number> series);`
  Tato metoda má za úkol spočítat jednu iteraci pro zobrazení fázového vývoje.

- *SolvableCasovyVyvoj2D*
  Interface, který umožňuje zobrazení časového vývoje jedné proměnné 2D systému ve 2D grafu. Definuje 2 metody:

  - `void solveCasovyVyvojIteration(double t, ResultHolder2D resultHolder, AbstractTO to, XYChart.Series<Number, Number> series, TimeGraphVariable variable);`
    Tato metoda má za úkol spočítat jednu iteraci pro zobrazení časového vývoje vybrané proměnné.
  - `void addPointBasedOnTimeVariable(double t, double x, double y, XYChart.Series<Number, Number> series TimeGraphVariable variable);`
    Metoda přidávající bod na časový graf v závislosti na výběru proměnné.

- *SolvableCasovyVyvoj3D*
  Interface, který umožňuje zobrazení časového vývoje jedné proměnné 3D systému ve 2D grafu. Definuje 2 metody:

  - `void solveCasovyVyvojIteration(double t, ResultHolder3D resultHolder, AbstractTO to, XYChart.Series<Number, Number> series, TimeGraphVariable variable);`
    Tato metoda má za úkol spočítat jednu iteraci pro zobrazení časového vývoje vybrané proměnné.
  - `void addBasedOnTimeGraphVariable(double t, double x, double y, double z,`

```
XYChart.Series<Number, Number> series,
TimeGraphVariable variable);
```
Metoda přidávající bod na časový graf v závislosti na výběru proměnné.

- *SolvableCasovyVyvoj4D*
  Interface, který umožňuje zobrazení časového vývoje jedné proměnné 4D systému ve 2D grafu. Definuje 2 metody:

  - `void solveCasovyVyvojIteration(double t,`
    `ResultHolder4D resultHolder4D,`
    `AbstractTO to,`
    `XYChart.Series<Number, Number> series,`
    `TimeGraphVariable variable);`
    Tato metoda má za úkol spočítat jednu iteraci pro zobrazení časového vývoje vybrané proměnné.
  - `void addBasedOnTimeGraphVariable(double t,`
    `double x,`
    `double y,`
    `double z,`
    `double w,`
    `XYChart.Series<Number, Number> series,`
    `TimeGraphVariable variable);`
    Metoda přidávající bod na časový graf v závislosti na výběru proměnné.

- *Abstract3DChart*
  Abstraktní třída definující společné chování 3D grafů.

- *Chart3D*
  Abstraktní třída dodefinovávající společné chování 3D grafu. Obsahuje 1 abstraktní metodu pro definici potomky:
  `boolean runDynamicStepCheck(ResultHolder3D resultHolder,`
  `Abstract3DTO to);`
  Tato metoda má za úkol provést kontrolu, zda se simulace neodchýlila od skutečného výpočtu o více, než je parametrem dovoleno. Návratová hodnota je typu `boolean`, která má hodnotu **true** ve chvíli, kdy se výpočet odchýlil o více, než je parametrem dovoleno a **false** ve chvíli, kdy je výpočet stále v normě.

- *Chart3DAsCut*
  Abstraktní třída dodefinovávající společné chování grafu, který bude vykreslený jako 2D řez 3D systému. Obsahuje 1 abstraktní metodu pro definici potomky:
  `boolean runDynamicStepCheck(ResultHolder3D resultHolder,`
  `Abstract3DTO to);`

Tato metoda má za úkol provést kontrolu, zda se simulace neodchýlila od skutečného výpočtu o více, než je parametrem dovoleno. Návratová hodnota je typu `boolean`, která má hodnotu **true** ve chvíli, kdy se výpočet odchýlil o více, než je parametrem dovoleno a **false** ve chvíli, kdy je výpočet stále v normě.

- *Chart4DAsCut*
  Abstraktní třída dodefinovávající společné chování grafu, který bude vykreslený jako 2D řez 4D systému. Obsahuje 1 abstraktní metodu pro definici potomky:
  `boolean runDynamicStepCheck(ResultHolder4D resultHolder, Abstract4DTO to);`
  Tato metoda má za úkol provést kontrolu, zda se simulace neodchýlila od skutečného výpočtu o více, než je parametrem dovoleno. Návratová hodnota je typu `boolean`, která má hodnotu **true** ve chvíli, kdy se výpočet odchýlil o více, než je parametrem dovoleno a **false** ve chvíli, kdy je výpočet stále v normě.

- *ResultHolder2D*
  Třída, jejíž instance drží hodnoty proměnných 2D systému.

- *ResultHolder3D*
  Třída, jejíž instance drží hodnoty proměnných 3D systému.

- *ResultHolder4D*
  Třída, jejíž instance drží hodnoty proměnných 4D systému.

- *Validatable*
  Interface, který definuje kontrakt pro všechny validovatelné komponenty. Pro tento účel definuje 1 metodu:
  `boolean validate();`
  Metoda vrací hodnot **true** v případě, že validovaný komponenta obsahuje přípustnou hodnotu, **false** v případě, že validovaná komponenta obsahuje nepřípustnou hodnotu. Tuto metodu spouští *AbstractController* před vykreslením grafu pro kontrolu správnosti zadaných hodnot uživatelem. Pokud nějaká komponenta obsahuje nepřípustnou hodnotu, zobrazí chybovou hlášku u této komponenty a nepokusí se zobrazit graf systému.

- *NumberTextField*
  Komponenta pro zadání čísla. Konfigurací se dá nastavit, jestli do této komponenty smí uživatel nastavit desetinné číslo, nebo jen celé číslo. Ve výchozím nastavení je komponenta nastavena pro použití s desetinnými čísli.

4.2.6 Ukázka přidání nového systému do aplikace

Cílem této sekce je ukázat, jak do aplikace lze přidat nový systém. Jedním z požadavků bylo, aby šel do aplikace jednoduše přidat nový systém. Ukážu tedy postup, jakým takového požadavku lze docílit. Tento postup ukážu na příkladu Nelineárního oscilátoru.

Úvodní obrazovka pro výběr simulovaného systému

Prvním krokem bude vložení konstanty identifikující daný systém do enumerace `StartWindowOption`. Následně pro zobrazení textace je také vhodné tuto konstantu přeložil v souboru `Application.properties`.

```
StartWindowOption.NELINEARNI_OSCILATOR =
   Nelineární oscilátor
```

O vykreslení nové hodnoty se automaticky postará `MainController` v metodě `initialize()`, konkrétně

```
for (StartWindowOption value :
   StartWindowOption.values()) {
    final RadioButton radioButton = new
       RadioButton( localizationHelper.getMessage(
       value.getDeclaringClass().getSimpleName() +
       "." + value ) );
    radioButton.setUserData(value);
    radioButton.setToggleGroup(selected);

    radioButtons.getChildren().add(radioButton);
}
```

Nyní je potřeba pro nový systém vytvořit jeho controller a samotnou obrazovku. Začneme controllerem:

```
@Controller
public class NelinearniOscilatorController
   extends AbstractController
```

Abstraktní třída `AbstractController` předepisuje implementaci 2 metod:

- `protected Chart createChart(AbstractTO to, DiagramType diagramType)`
  Metoda pro vytvoření instance třídy, která bude zobrazovat graf na obrazovce. Protože se v tomto případě jedná o 2D graf, budeme tvořit instanci třídy `Chart2D`.
- `protected AbstractTO createTo()`
  Metoda pro vytvoření instance třídy transportního objektu pro přenos dat konkrétního systému. V aplikaci je slušným zvykem vytvořit tento složitý objekt pomocí Builderu.

Dále, třída `AbstractController` nemá defaultní konstruktor, tudíž je nutné vytvořit speciální konstruktor, který bude schopen splnit kontrakt konstruktoru třídy `AbstractController`.

```
@Autowired
public
   NelinearniOscilatorController(LocalizationHelper
   localizationHelper, MyExecutorService
   executorService, RungeKuttaSolver solver) {
    super(localizationHelper, executorService,
       solver);
}
```

Vidíme, že většina závislostí je vyřešena pomocí anotace `@Autowired`, tedy technikou vložení závislostí ze Spring IoC kontejneru.

Tvorba FXML souboru

Vytvoříme obrazovku pro tento systém ve formátu a specifikaci FXML. FXML je značkovací jazyk postavený na technologii XML, který poskytuje jazyk pro tvorbu uživatelského rozhraní oddělenou od aplikační logiky [25]. Definice prvků je velmi podobná Swingu, ze kterého JavaFX vychází. Ukázku souboru můžete najít v přiloženém zdrojovém kódu práce na cestě `src/main/resources/screens/NelinearniOscilatorDialog.fxml`.

Přidání do Springové konfigurace

Nyní máme hotové všechny potřebné souboru pro přidání systému do Spring kontejneru. Spring konfiguraci mám v diplomové práci udělanou cestou Java

konfigurace. Jedná se o Java třídu, která je anotovaná anotací `@Configuration`.
Do této třídy přidáme novou závislost na FXML obrazovce:

```
@Value("classpath:screens/NelinearniOscilatorDialog.fxml")
private Resource
   nelinearniOscilatorDialogResource;
```

Dále přidáme nový controller do seznamu controllerů pro systémy:

```
@Bean(name = "systemControllerMap")
public Map<StartWindowOption, AbstractController>
   systemControllerMap() {
    final Map<StartWindowOption,
       AbstractController> solvingSystemMap = new
       HashMap<>();

    ...

    solvingSystemMap
        .put(StartWindowOption.NELINEARNI_OSCILATOR,
           new
           NelinearniOscilatorController(localizationHelper(),
           myExecutorService(),
           rungeKuttaSolver()));

    ...

    return solvingSystemMap;
}
```

Následně přidáme obrazovku do seznamu obrazovek aplikace:

```
@Bean(name = "systemResourceMap")
public Map<StartWindowOption, Resource>
   systemResourceMap() {
    final Map<StartWindowOption, Resource>
       solvingSystemMap = new HashMap<>();

    ...

    solvingSystemMap.put(StartWindowOption.NELINEARNI_OSCILATOR,
       nelinearniOscilatorDialogResource);

    ...

    return solvingSystemMap;
```

```
}
```

Transfer objekt pro systém

Pro nový systém musíme do balíčku `cz.passler.fel.ne.to` přidat tranfer objekt, který bude držet aktuální parametry tohoto systému. Pro nelineární oscilátor máme pouze jeden speciální atribut `alpha`:

```
public class NelinearniOscilatorTO extends
    AbstractTO {

    private double alpha;

    public double getAlpha() {
        return alpha;
    }

    public void setAlpha(double alpha) {
        this.alpha = alpha;
    }
}
```

Tento transfer objekt dědí od svého předka `AbstractTO`, čímž dostává automaticky atributy společné všem 2D systémům.

Builder

Pro nový transfer objekt též vytvoříme jeho builder do balíčku `cz.passler.fel.ne.builder`. Tento builder zpravidla pouze vytvoří výše uvedený transfer objekt:

```
public class NelinearniOscilatorTOBuilder extends
    AbstractTOBuilder {

    public static NelinearniOscilatorTO build(
        TextField alpha,
        TextField xmin,
        TextField xmax,
        TextField x0,
        TextField ymin,
```

```
            TextField ymax,
            TextField y0,
            TextField startTime,
            TextField endTime,
            TextField timeStep,
            TextField exactness,
            TextField speed,
            ComboBox<TimeGraphVariable>
               timeGraphVariable
        ) {
            final NelinearniOscilatorTO to = new
               NelinearniOscilatorTO();
            setupCommonParameters(to,
              Double.valueOf(xmin.getText()),
              Double.valueOf(xmax.getText()),
              Double.valueOf(x0.getText()),
              Double.valueOf(ymin.getText()),
              Double.valueOf(ymax.getText()),
              Double.valueOf(y0.getText()),
              Double.valueOf(startTime.getText()),
              Double.valueOf(endTime.getText()),
              Double.valueOf(timeStep.getText()),
              Double.valueOf(exactness.getText()),
              Integer.valueOf(speed.getText()),
              timeGraphVariable
                .getSelectionModel()
                .getSelectedItem());

            to.setAlpha(Double.valueOf(alpha.getText()));

            return to;
        }

}
```

Strategie výpočtu systému

Poslední, co chybí k vykreslení grafu pro systém je vytvoření strategie výpočtu simulace daného systému. Vytvoříme tedy strategii:

```
public class NelinearniOscilatorStrategy
   implements SolvingStrategy {
```

```
    @Override
    public RungeKuttaOutput solve(RungeKuttaInput
       input) {
        return solve((NelinearniOscilatorRKInput)
           input);
    }

    private RungeKuttaOutput
       solve(NelinearniOscilatorRKInput input) {
        double k1 = input.getY();

        double l1 = -input.getX() +
           input.getAlpha() *
           Math.pow(input.getX(), 2);

        double k2 = input.getY() + 0.5 *
           input.getDt() * l1;

        double y12 = input.getX() + 0.5 *
           input.getDt() * k1;
        double l2 = -y12 + input.getAlpha() *
           Math.pow(y12, 2);

        double k3 = input.getY() + 0.5 *
           input.getDt() * l2;

        double y13 = input.getX() + 0.5 *
           input.getDt() * k2;
        double l3 = -y13 + input.getAlpha() *
           Math.pow(y13, 2);

        double k4 = input.getY() + input.getDt()
           * l3;

        double y14 = input.getX() + input.getDt()
           * k3;
        double l4 = -y14 + input.getAlpha() *
           Math.pow(y14, 2);

        double newX = input.getX() + (k1 + 2 * k2
           + 2 * k3 + k4) * input.getDt() / 6;
        double newY = input.getY() + (l1 + 2 * l2
           + 2 * l3 + l4) * input.getDt() / 6;

        return new RungeKuttaOutput(newX, newY);
    }
```

```
}
```

Tuto strategii založíme do Spring kontejneru, aby ji mohl systém automaticky detekovat a použít:

```
@Bean(name = "solveStrategyMap")
public Map<SolvingSystem, SolvingStrategy>
   solvingStrategyMap() {
    final Map<SolvingSystem, SolvingStrategy>
       strategyMap = new HashMap<>();

    ...

    strategyMap.put(SolvingSystem.NELINEARNI_OSCILATOR,
       new NelinearniOscilatorStrategy());

    ...

    return strategyMap;
}
```

Kapitola 5

Testování a dokumentace

Testování a dokumentace softwaru je jedním ze základních a slučných návyků softwarového vývojáře. Jak se aplikace rozvíjejí a jsou stále složitější, je velmi složité dohlédnout všechny možné dopady určitých budoucích změn v aplikaci. Právě zde nám mohou dopomoci automatizované testy. V případě, že zásahem do aplikace změníme fungování jiné, třeba i nesouvisející části aplikace, správně napsané testy by nás na to měly upozornit. Pokud neexistuje kvalitní dokumentace k softwaru, může to být také příčinou mnoha problémů. Pokud se k týmu připojí nový člen, velmi těžko se mu bude orientovat v aplikaci, která není zdokumentovaná. Tím, namísto aby pomáhal, spíše bude škodit a brzdit samotný vývoj nových funkčností.

5.1 Testování

Existuje několik druhů automatizovaných testů pro softwarové aplikace. V této kapitole představím druhy testů, které jsem použil ve své diplomové práci a ukážu příklady.

5.1.1 Jednotkové (unit) testy

Jednotkové, neboli unit, testy jsou takové, které testují malé kousky kódu. Pro vytvoření takového testu musíme izolovat jednotlivé kusy aplikace a tyto

otestovat a tím ověřit jejich funkčnost [24]. Pro vytvoření tohoto druhu testů jsem použil ve své práci framework jUnit `https://junit.org/junit5/`. Jako příklad ukážu test třídy pro výpočet jednoho kroku simulace. K vygenerování testovacích případů jsem použil nástroj Allpairs, který vygeneruje testovací případy technikou párového testování od Jamese Bacha [23].

Ukázka testu

Vytvoření testovacích dat:

```
@Parameterized.Parameters(name = "Instance
   {index}")
public static Collection<Object[]> data() {
    return Arrays.asList(
        new Object[][]{
            {new Bruselator2DRKInput(0, 0,
               timeDifference, 10, 20, 30), new
               RungeKuttaOutput(9.999000066663334E-5,
               1.499900004999969E-8)},
            {new Bruselator2DRKInput(0, -5,
               timeDifference, 0, 0, 0), new
               RungeKuttaOutput(0.0, -5.0)},
            {new Bruselator2DRKInput(0, 5,
               timeDifference, -10, -20, -30), new
               RungeKuttaOutput(-1.000100005E-4,
               5.0000000150008335)},
            {new Bruselator2DRKInput(-5, 0,
               timeDifference, 0, -20, -30), new
               RungeKuttaOutput(-5.000999912447293,
               0.0014999624443749954)},
            {new Bruselator2DRKInput(-5, -5,
               timeDifference, 10, 20, -30), new
               RungeKuttaOutput(-5.000149678728171,
               -4.997250283825374)},
            {new Bruselator2DRKInput(-5, 5,
               timeDifference, 10, 0, 0), new
               RungeKuttaOutput(-4.9986504935400315,
               4.998750493540031)},
            {new Bruselator2DRKInput(5, 0,
               timeDifference, -10, 0, -30), new
               RungeKuttaOutput(4.999899812521881,
               -0.0014997975031329311)},
```

```
            {new Bruselator2DRKInput(5, -5,
               timeDifference, 0, 20, 30), new
               RungeKuttaOutput(4.997751130592175,
               -4.9972512430544835)},
            {new Bruselator2DRKInput(5, 5,
               timeDifference, 10, -20, 0), new
               RungeKuttaOutput(5.002350666363637,
               4.998749568680785)},
            {new Bruselator2DRKInput(-5, 5,
               timeDifference, -10, 20, 30), new
               RungeKuttaOutput(-4.997851095620153,
               4.997251203083628)},
            {new Bruselator2DRKInput(0, -5,
               timeDifference, -10, 0, 30), new
               RungeKuttaOutput(-1.0000000016666665E-4,
               -5.000000014999833)},
            {new Bruselator2DRKInput(0, 0,
               timeDifference, 0, 20, 0), new
               RungeKuttaOutput(0.0, 0.0)},
            {new Bruselator2DRKInput(0, 5,
               timeDifference, 0, 0, -30), new
               RungeKuttaOutput(0.0, 5.0)},
            {new Bruselator2DRKInput(-5, -5,
               timeDifference, -10, -20, 0), new
               RungeKuttaOutput(-5.002350666363637,
               -4.998749568680785)},
        }
    );
}
```

V ukázce je vidět použití anotace `@Parametrized.Parameters`, která se používá k parametrizovatelnému spuštění jUnit testu. Tato anotace, pokud je testovací třída anotovaná pomocí `@RunWith(Parameterized.class)` pro každý záznam v kolekci vloží do instanční proměnné třídy danou hodnotu pomocí jejího indexu:

```
@Parameterized.Parameter(value = 0)
public RungeKuttaInput input;

@Parameterized.Parameter(value = 1)
public RungeKuttaOutput output;
```

Potom v samotném testu je možné použít automaticky tyto hodnoty:

```
@Test
public void runTest() {
    final RungeKuttaOutput actualOutput =
       strategy.solve(input);

    assertEquals("Instance incorrect!", output,
       actualOutput);
}
```

5.1.2 Integrační testy

Dalším typem testu použitým v diplomové práci je integrační test. Tento typ testu testuje, zda jednotlivé komponenty (fungující samostatně) po spojení spolupracují korektně [30]. Pro realizaci těchto testů jsem ve spolupráci s jUnit použil framework Mockito `http://site.mockito.org`. Pro ukázku jsem použil třídu `RungeKuttaSolver` sloužící pro získání správné strategie pro výpočet (viz. kapitola 4.2.3).

Ukázka testu

Pro aktivaci anotací z Mockito frameworku je nutné test spouštět Runnerem `MockitoJUnitRunner`.

```
@RunWith(MockitoJUnitRunner.class)
public class RungeKuttaSolverTest {
```

Dále vytvoříme mock objekt ze závilosti a tento vložíme do testované třídy:

```
@Mock
private Map<SolvingSystem, SolvingStrategy>
   solveStrategy;

@InjectMocks
private RungeKuttaSolver solver;
```

Nyní mám třídu připravenou se svou závislostí a spustím test pro známý systém:

```
@Test
public void testSolve_knownSystem() {
    SolvingStrategy strategy =
       Mockito.mock(Bruselator2DStrategy.class);
    final Bruselator2DRKInput input = new
       Bruselator2DRKInput(0, 0, 0, 0, 0, 0);
    final RungeKuttaOutput output = new
       RungeKuttaOutput(1, 2);

    Mockito.when(solveStrategy.containsKey(
       SolvingSystem.BRUSELATOR_2D) ).thenReturn(
       true );
    Mockito.when(solveStrategy.get(
       SolvingSystem.BRUSELATOR_2D ) ).thenReturn(
       strategy );
    Mockito.when(strategy.solve( input
       )).thenReturn( output );

    final RungeKuttaOutput actualOutput =
       solver.solve( input );

    assertSame(output, actualOutput);

    Mockito.verify(solveStrategy).containsKey(
       SolvingSystem.BRUSELATOR_2D );
    Mockito.verify(solveStrategy).get(
       SolvingSystem.BRUSELATOR_2D );
    Mockito.verifyNoMoreInteractions(
       solveStrategy );
}
```

Nejprve vytvořím testovací objekty v první sekci testu. Dále nastavím chování mock objektů pro průchod, jaký požaduji. Následně spustím výkonnou metodu, kterou chci otestovat. Nakonec ověřím, že metoda vrátila co měla a zároveň u výpočtu použila volání metod, které jsem nastavil a žádné jiné.

Přidám ještě test pro neznámý systém:

```
@Test
public void testSolve_unknownSystem() {
    final Bruselator2DRKInput input = new
       Bruselator2DRKInput(0, 0, 0, 0, 0, 0);

    Mockito.when(solveStrategy.containsKey(
       SolvingSystem.BRUSELATOR_2D ) ).thenReturn(
```

```
        false );

    try {
        solver.solve(input);
        fail();
    } catch (IllegalArgumentException e) {}

    Mockito.verify(solveStrategy).containsKey(
       SolvingSystem.BRUSELATOR_2D );
    Mockito.verifyNoMoreInteractions(solveStrategy);
}
```

V tomto testu očekávám vyhození výjimky `IllegalArgumentException`, proto poslední řádek sekce *try* končí volání metody `fail()`.

5.1.3 Testy uživatelského rozhraní

Posledním typem testů použitých v diplomové práci jsou automatizované testy uživatelského rozhraní. Tyto testy mají za úkol otestovat správné chování aplikace simulací uživatelské interakce (klikáním, vpisováním hodnot apod.). K otestování těchto funkcionalit jsem použil framework testFX `https://github.com/TestFX/TestFX/wiki`. Pro ukázku použiji dva testy, jeden ověří, že jsou na obrazovce správné rovnice a druhý ověří, že do číselných polí nelze zapsat písmeno.

Test správných rovnic

Test musí dědit od abstraktní třídy framworku testFX:

```
public abstract class AbstractTest extends
   ApplicationTest
```

Nyní implementuji metodu pro start testu a tím i aplikace:

```
@Override
public void start(Stage stage) throws Exception {
    super.start(stage);
```

```
    final FXMLLoader loader = new
       FXMLLoader(getFXMLResource().getURL());
    loader.setController(getController());

    stage.setScene(new Scene(loader.load(),
       stage.getWidth(), stage.getHeight()));
    stage.show();
}
```

A spustím test:

```
@Test
public void testCorrectEquations() {
    final List<String> correctEquations =
       getCorrectEquations();

    final VBox equationsBox =
       lookup("#equations").query();
    final Set<Label> equations =
       from(equationsBox).lookup((Label l) ->
       l.getParent().equals(equationsBox)).queryAll();

    assertEquals(getCorrectEquations().size(),
       equations.size());

    for (Label equation : equations) {
        assertTrue(correctEquations.contains(
           equation.getText() ) );
    }
}
```

Test se pokusí najít element s ID *equations*. Poté se pokusí naleznout v něm umístěný *Label*. Následně zkontroluje, zda obsah tohoto *Labelu* odpovídá požadovaným rovnicím.

Test správnosti omezení číselných polí

Test už má spuštěnou aplikaci a proto stačí jen jeho samotná výkonná část:

```
@Test
public void
   testDecimalValidation_cannotInsertLetter() {
```

```
    final Map<String, String> defaultValues =
        getDefaultValues();
    final Set<String> numericInputs =
        getNumericInputs();

    for (String numericInput : numericInputs) {
        logger.debug("Testing " + numericInput);

        final TextField textField = lookup("#" +
            numericInput).query();

        clickOn(textField, Motion.DEFAULT,
            MouseButton.PRIMARY).type(KeyCode.A);

        // check that value is not changed
        assertEquals(defaultValues.get(numericInput),
            textField.getText());
    }
}
```

Pokusím se najít element podle ID (jež vrátí metoda `getNumericInputs()`) a do tohoto elementu se pokusí zapsat písmeno *A*. Poté zkontroluje, že se obsah elementu nezměnil.

5.2 Dokumentace

Jak jsem nastínil v úvodu kvalitní dokumentace je základem pro kvalitní a rychlý rozvoj aplikace. Pro svou diplomovou práci jsem vytvořil dokumentaci zdrojového kódu ve formátu javaDoc a uživatelskou dokumentaci pro správné používání softwaru. Obě zmíněné dokumentace jsou k nalezení na přiloženém CD k této práci.

Kapitola 6

Závěr

V mojí diplomové práci jsem měl za úkol provést analýzu stávajícího programu pro simulaci nelineárních dynamických systémů. Dále jsem měl za cíl vytvořit novou verzi tohoto programu v programovacím jazyce Java se zachováním stávající funkcionality. K této funkcionalitě měla přibýt možnost postupného vykreslování simulace, paralelního vykreslování simulací pomocí zadání počatečních podmínek kliknutím myší a zobrazení 3D modelů u podivných atraktorů. V kapitole 2 jsem identifikoval silné a slabé stránky aplikace, zhodnotil jsem možné přístupy k řešení diferenciálních rovnic pomocí programovacího jazyka Java a vybral z nich ten nejvíce vhodný pro danou problematiku. Také jsem se seznámil s programovací metodou dynamického časového kroku pro vyhlazení simulace. V kapitole 3 jsem se zadavatelem práce definoval funkční a nefunkční požadavky na aplikaci. Níže v této kapitole vyhodnotím splnění, resp. nesplnění jednotlivých požadavků. V kapitole 4 jsem popsal postup implementace a rozebral základní balíčky a třídy aplikace s popisem jejich funkčnosti. V poslední kapitole 5 jsem popsal postup testování aplikace a také jaké formy dokumentace jsem zvolil a přiložil k této diplomové práci.

6.1 Zhodnocení splnění/nesplnění požadavků na aplikaci

V této sekci zhodnotím míru splněnosti všech požadavků na aplikace. Za tímto účelem si znovu připomeneme tabulky ze sekce 3.1.

6.1.1 Funkční požadavky

Identifikace	Popis	Splněno?
FR1	Zachovat stávající funkcionalitu aplikace	✓
FR2	Možnost 3D simulace	✓
FR3	Validovat vstupy od uživatele do aplikace	✓
FR4	Výběr řezu do 2D grafu pro vícedimensionální systémy	✓
FR5	Výběr proměnné pro znázornění časového průběhu	✓
FR6	Postupné vykreslení simulace definovanou rychlostí	✓

Tabulka 6.1: Vyhodnocení splnění funkčních požadavků na aplikaci

6.1.2 Nefunkční požadavky

Identifikace	Popis	Splněno?
NR1	Přenositelnost mezi různými operačními systémy	✓
NR2	Paralelní vykreslování různých simulací	✓
NR3	Distribuce ve formátu .jar jako stand-alone verze	✓
NR4	Jednoduché rozšíření aplikace o nové systémy	✓

Tabulka 6.2: Vyhodnocení splnění funkčních požadavků na aplikaci

6.1.3 Návrhy na rozšíření aplikace

Do aplikace v budoucnu přibydou další systémy pro jednotlivé simulace. Jak tyto systémy v budoucnu přidat si lze připomenout v sekci 4.2.6. Dalším možným budoucím rozšířením je, že všechny parametry systémů, které byly zvoleny jako konstanty, budou dynamické. Na pevno zvolené hodnoty budou pouze výchozími s možností změny. Dále do aplikace je možné postupně přidávat další systémy, jako například saturační problémy, difúze singulárního píku atd. Aplikace bude nasazena ve výuce hned v zimním semestru akademického roku 2018/2019 a jistě z praktického používání aplikace vyplynou další návrhy na vylepšení.

Příloha A

Literatura

[1] *Autocatalysis*, https://www.sciencedaily.com/terms/autocatalysis.htm, [online], cit. 1. 4. 2018.

[2] *Brusselator*, https://www.slideshare.net/RohitAggarwal9/brusselator, [online], cit. 1. 4. 2018.

[3] *Design pattern - transfer object pattern*, https://www.tutorialspoint.com/design_pattern/transfer_object_pattern.htm, [online], cit. 22. 4. 2018.

[4] *Design patterns - builder pattern*, https://www.tutorialspoint.com/design_pattern/builder_pattern.htm, [online], cit. 22. 4. 2018.

[5] *Design patterns - strategy pattern*, https://www.tutorialspoint.com/design_pattern/strategy_pattern.htm, [online], cit. 23. 4. 2018.

[6] *Fortran 90*, http://www.fortran90.org/, [online], cit. 26. 3. 2018.

[7] *Hopf bifurcation*, http://www.ucl.ac.uk/~ucesgvd/hopf.pdf, [online], cit. 4. 4. 2018.

[8] *The ioc container*, https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/beans.html, [online], cit. 23. 4. 2018.

[9] *Iterative development*, https://www.agilealliance.org/glossary/iterative-development, [online], cit. 26. 3. 2018.

[10] *Iterative development: The future of project management*, https://www.netsville.com/wp-content/uploads/2015/10/agile.jpg, [online], cit. 26. 3. 2018.

[11] *Java*, https://java.com, [online], cit. 26. 3. 2018.

[12] *The lorenz attractor in 3d*, http://paulbourke.net/fractals/lorenz/, [online], cit. 4. 4. 2018.

[13] *Lotka-volterra equations*, http://mathworld.wolfram.com/Lotka-VolterraEquations.html, [online], cit. 4. 4. 2018.

[14] *M-638 lectures*, https://carretero.sdsu.edu/teaching/M-638/lectures/lectures.html, [online], cit. 4. 4. 2018.

[15] *Numerical methods for differential equations*, http://faculty.olin.edu/bstorey/Notes/DiffEq.pdf, [online], cit. 10. 4. 2018.

[16] *Numerical methods for solving differential equations - euler's method - theoretical introduction*, http://calculuslab.deltacollege.edu/ODE/7-C-1/7-C-1-h-a.html, [online], cit. 10. 4. 2018.

[17] *Numerical methods for solving differential equations - euler's method - theoretical introduction*, http://calculuslab.deltacollege.edu/ODE/7-C-1/7-C-1-h-b.html, [online], cit. 10. 4. 2018.

[18] *Octave/matlab - differential equation*, http://www.sharetechnote.com/html/Octave_Matlab_DifferentialEquation.html, [online], cit. 4. 4. 2018.

[19] *Open source api for 3d charts*, http://www.jzy3d.org, [online], cit. 23. 4. 2018.

[20] *Rossler attractor*, http://www.scholarpedia.org/article/Rossler_attractor, [online], cit. 4. 4. 2018.

[21] *Rössler attractor*, https://en.wikipedia.org/wiki/Rössler_attractor, [online], cit. 4. 4. 2018.

[22] *Supercritical hopf bifurcation diagram*, https://www.researchgate.net/figure/Supercritical-Hopf-bifurcation-diagram_fig21_264118954, [online], cit. 4. 4. 2018.

[23] *Test tools*, http://www.satisfice.com/tools.shtml, [online], cit. 8. 5. 2018.

[24] *Unit testing tutorial - learn in 10 minutes*, https://www.guru99.com/unit-testing-guide.html, [online], cit. 8. 5. 2018.

[25] *Using fxml to create a user interface*, https://docs.oracle.com/javafx/2/get_started/fxml_tutorial.htm, [online], cit. 28. 4. 2018.

[26] *The van der pol oscillator*, http://www2.me.rochester.edu/courses/ME406/webexamp5/vanpol.pdf, [online], cit. 4. 4. 2018.

[27] *Welcome to apache maven*, https://maven.apache.org, [online], cit. 23. 4. 2018.

[28] Daniel Deutsch, *Understanding mvc architecture with react*, https://medium.com/of-all-things-tech-progress/understanding-mvc-architecture-with-react-6cd38e91fefd, [online], cit. 21. 4. 2018.

[29] Ulf Eriksson, *Functional vs non functional requirements*, https://reqtest.com/requirements-blog/functional-vs-non-functional-requirements/, [online], cit. 21. 4. 2018.

[30] Martin Fowler, *Integrationtest*, https://martinfowler.com/bliki/IntegrationTest.html, [online], cit. 8. 5. 2018.

[31] Stephen Cass Nick Diakopoulos, *Interactive: The top programming languages 2017*, https://spectrum.ieee.org/ns/IEEE_TPL_2017/index/2017/1/1/1/0/1/50/1/50/1/50/1/30/1/30/1/30/1/20/1/20/1/5/1/5/1/20/1/100/, [online], cit. 23. 4. 2018.

[32] Monica Pawlan, *What is javafx?*, https://docs.oracle.com/javafx/2/overview/jfxpub-overview.htm, [online], cit. 23. 4. 2018.

[33] Petr Kulhánek, *Úvod do teorie plazmatu*, vol. 1, Aldebaran Group for Astrophysics (AGA), March 2011.

[34] ———, *Vybrané kapitoly z teoretické fyziky*, vol. 1, Aldebaran Group for Astrophysics (AGA), August 2016.

[35] David Robson, *How computers change the way we learn*, http://www.bbc.com/future/story/20141022-are-we-getting-smarter, 2014, [online], cit. 26. 3. 2018.

[36] Anand Srini, *The brusselator problem*, https://rotordynamics.wordpress.com/2014/05/09/the-brusselator-problem/, [online], cit. 4. 4. 2018.

[37] William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery, *Numerical recipes in c: The art of scientific computing*, Cambridge University Press, 2002.

[38] Hong Qian Yi-An Ma, *A thermodynamic theory of ecology: Helmholtz theorem for lotka–volterra equation, extended conservation law, and stochastic predator–prey dynamics*, http://rspa.royalsocietypublishing.org/content/471/2183/20150456, [online], cit. 4. 4. 2018.

Příloha B

Seznam příloh

- `user-guide.pdf`
  Uživatelská příručka aplikace