MASARYKOVA UNIVERZITA

Přírodovědecká fakulta
Ústav teoretické fyziky a astrofyziky

DIPLOMOVÁ PRÁCE

Nelineární procesy v akrečních discích

Bc. Jiří Květoň

Vedoucí práce: **Mgr. Viktor Votruba, Ph.D.**

Brno 2014

Chtěl bych v prvé řadě poděkovat vedoucímu práce Viktoru Votrubovi za jeho cenné rady, motivaci a směrující podněty nejen k této práci. Také bych chtěl poděkovat Káje za její podporu a neutuchající důvěru ve vše, čemu se věnuji, a své rodině za jejich nikdy nekončící trpělivost.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Brně dne 16. 5. 2014 Jiří Květoň

Bilbliografický záznam

Autor:	Bc. Jiří Květoň Přírodovědecká fakulta, Masarykova univerzita Ústav teoretické fyziky a astrofyziky

Název práce:	Nelineární procesy v akrečních discích
Studijní program:	Fyzika
Studijní obor:	Teoretická fyzika a astrofyzika
Vedoucí práce:	Mgr. Viktor Votruba, Ph.D.
Rok obhajoby:	2014
Klíčová slova:	Chaos, akreční disk, simulace kapajícího kohoutku

Bibliographic entry

Author:	Bc. Jiří Květoň Faculty of Science, Masaryk University Department of Theoretical Physics and Astrophysics

Title of thesis:	Nonlinear processes in acretion discs
Degree programme:	Physics
Field of study:	Theoretical Physics and Astrophysics
Supervisor:	Mgr. Viktor Votruba, Ph.D.
Year of defense:	2014
Keywords:	Chaos, accretion disk, dripping faucet simulation

Abstrakt

Cílem této práce je vytvořit jednoduché nelineární modely kapajícího kohoutku, které v modifikované podobě mohou být jedním z možných vysvětlení nepravidelného chování akrečních disků. Detailněji jsou studovány dva modely kapajícího kohoutku. První z nich je založen na dynamickém popisu chování kapaliny v diskretizovaném systému kapky pomocí Lagrangeových rovnic. Druhý z předložených modelů popisuje systém kapky jako závaží o proměnné hmotnosti na pružině s nastavenými kritickými parametry odtržení. Výsledky simulací jsou následně podrobeny další analýze za účelem získání některých charakteristických invariantů popisujících nelineární systémy. V závěru práce jsou diskutovány možnosti budoucí vícerozměrné modifikace těchto modelů jako tzv. modelu kapajícího disku, jenž je vhodný k popisu chování akrečních disků.

Klíčová slova: Chaos, akreční disk, simulace kapajícího kohoutku

Abstract

The aim of this work is to develop a simple non-linear models of dripping faucet, which in modified form may be one possible explanation for the irregular behavior of the accretion disks. Two models of dripping facuet are studied in detail. The first one is based on a dynamic description of the behavior of fluid in the discrete system of drop using Lagrange equations. Second presented model describes the system of drop as the weight with the variable mass on a spring with a critical break parameters. The results of simulation are then subjected to further analysis in order to obtain some characteristic invariants describing nonlinear systems. The conclusion discusses the possibility of future multidimensional modifications of these models such as the dripping disc model, which is suitable to describe the behavior of accretion disks.

Keywords: Chaos, accretion disk, dripping faucet simulation

Obsah

1. Úvod

"I accept chaos, I'm not sure whether it accepts me."
Bob Dylan

1.1 Oficiální zadání

Úkolem studenta bude analyzovat světelnou křivku kvasaru 3C 273 s pomocí metod nelineární analýzy časových řad, stanovit hodnoty charakteristických invariantů jako je korelační dimenze, Lyapunovy exponenty a rekonstruovat stavový portrét. Dále pak porovná výsledky této analýzy s jednoduchými nelineárními modely akrečních disků a získané poznatky podrobí kritické diskusi.

1.2 Motivace

Výzkumy v oblasti teorie chaosu a nelineárních dynamických systémů zažily v posledních pár desetiletích značný pokrok. V astrofyzice lze nalézt mnoho příkladů, kde nelinearita systému hraje podstatnou roli a je zodpovědná za jeho případné nepravidelné chování. Jedním z možných příkladů mohou být akreční disky a jejich projev v podobě nepravidelných změn pozorovaných světelných křivek.

Chování akrečního disku je dobře možné aproximovat kapajícím modelem, který v podstatě vychází z modelu kapajícího kohoutku. Cílem této práce je pochopit chování různých kapajících modelů, určit možnosti jejich použitelnosti na různé případy akreujících systémů, k tomuto účelu vytvořit simulační software takových modelů a poté případně srovnat získaná modelová data s reálnými pozorováními.

1.3 Struktura práce

Po krátkém stručném seznámení se strukturou práce a použitým softwarem v úvodní kapitole následuje druhá kapitola věnující se teoretickému výkladu na téma teorie chaosu a nelineární dynamiky. Třetí kapitola seznamuje s některými numerickými metodami řešení obyčejných diferenciálních rovnic, z nichž jedna je přímo využita při výpočtech v této práci. Hlavní význam této kapitoly je hlouběji seznámit čtenáře s teoretickým pozadím výpočtů v pozdějších kapitolách,

které by jinak byly jen *černou skříňkou*. Čtvrtá kapitola se věnuje zpracování dvou variant modelu kapajícího kohoutku a následné analýze výsledků ze simulací těchto modelů. Pátá kapitola uvádí teoretický popis modelu kapajícího disku a diskutuje možnosti budoucího rozšíření modelů popsaných v předchozí čtvrté kapitole. Poté následuje závěr s diskuzí výsledků práce a nakonec je zařazena kapitola s přílohami, která obsahuje některé vybrané scripty v jazyce Python použité v při výpočtech a simulacích v této práci. Tyto scripty jsou také dostupné v elektronické podobě na adrese:

http://physics.muni.cz/~kveton/

1.4 Použitý software

K realizaci jednoduchých výpočtů i komplexnějších simulací modelů kapajících systémů je v této práci využíván programovací jazyk Python [1] ve verzi 2.7.5, který je hojně využíván vědci a inženýry po celém světě. Jeho hlavní výhody tkví především v jeho širokém spektru použitelnosti. Aplikace jazyka Python lze nalézt od zpracování velkého množství ekonomických nebo sociologických dat až k uplatnění v počítačových hrách. Díky tomu existuje mnoho doplňkových knihoven a balíků nejrůznějšího zaměření, za kterými stojí rozsáhlá komunita uživatelů.

V této práci je použita široká škála knihoven jazyka Python. Pro výpočty a numerické simulace jsou to knihovny NumPy [13] a SciPy [9], k vizualizaci dat balík Matplotlib [8], pro symbolické výpočty a hromadné úpravy výrazů knihovna SymPy [5] a dále některé další běžně používané knihovny jazyka Python.

Další použité nástroje, které je dobré zmínit, jsou program FFmpeg [17] k tvorbě videí z výsledků některých simulací. Velmi důležitý je také programový balík Tisean [6] ve verzi 3.0.0 distribuovaný ve formě zdrojových kódů v jazycích C a Fortran. Balík Tisean slouží k analýze nelineárních časových řad a je v něm obsažena poměrně široká paleta nástrojů.

2. Teorie chaosu a nelineární dynamika

"Too bad the post office isn't as efficient as the weather service."
Dr. Emmett Lathrop Brown, Ph.D.

Ve druhé polovině 17. století položili Issac Newton a Gottfried Wilhelm von Leibniz svými průlomovými pracemi na poli diferenciálního počtu základy modelování vývoje fyzikálních systémů pomocí diferenciálních rovnic. Newtonovými největšími úspěchy byly objevy pohybových zákonů a gravitačního zákona. S jejich pomocí pak mohl vysvětlit Keplerovy zákony pohybu planet kolem Slunce [24]. Newton vlastně vyřešil tzv. problém dvou těles.

Následující generace fyziků a matematiků se marně pokoušely, stejně jako sám Newton už v roce 1687 ve svých Principiích, rozšířit řešení na systém tří těles (například Slunce, Země a Měsíc). Švédský král Oscar II. dokonce v roce 1887 u příležitosti svých 60. narozenin vyhlásil odměnu za nalezení řešení. Nakonec se dospělo k závěru, že problém tří těles je nemožné vyřešit ve smyslu nalezení explicitního vyjádření pohybových rovnic jednotlivých těles.

Cenu švédského krále nakonec získal francouzský fyzik a matematik Henri Poincaré. Přestože vlastně požadovaného řešení nedosáhl, velmi výrazně přispěl k rozvoji nebeské mechaniky. Poincaré objevil, že mohou existovat orbity, které jsou neperiodické a nejsou ani neustále vzrůstající ani se neblíží k pevnému bodu. Poincaré byl tak první kdo se dotkl problematiky chaotického chování v deterministickém systému a jeho práce obsahovala mnoho myšlenek důležitých pro pozdější rozvoj teorie chaosu.

Teorie chaosu rychle postupovala vpřed ve druhé polovině minulého století, kdy se stalo pro některé vědce zřejmé, že lineární teorie, převažující teorie systémů v tomto období, prostě nemůže vysvětlit pozorované chování v určitých experimentech. Hlavním katalyzátorem vývoje teorie chaosu byl elektronický počítač. Studium chování chaotického systému ve většině případů vyžaduje mnoho relativně jednoduchých výpočtů, které je ale třeba provádět v opakovaných iteracích na určitém časovém intervalu. Provádět takové výpočty ručně je velice nepraktické. Elektronický počítač tak doslova vytrhl trn z paty všem výzkumníkům na poli teorie chaosu, protože jim neuvěřitelným způsobem usnadňuje a zrychluje práci [18].

Počítač ENIAC, který byl jedním z prvních, byl využíván ke studiu jednoduchých modelů předpovědi počasí. Jedním z výzkumníků teorií týkajících se předpovědi počasí byl i Edward Lorenz, který do styku s chaosem přišel víceméně náhodou. V roce 1961 prováděl na svém počítači Royal McBee LPG-30 simulaci modelu předpovědi počasí. Jednou, když opětovně spouštěl simulaci, zadal jako počáteční podmínky data zhruba z poloviny běhu předcházejícího výpočtu, aby ušetřil čas. Dočkal se ale obrovského překvapení, když výsledky z této simulace byly zcela odlišné od předešlých, i když zadal stejná data. Nakonec zjistil, že tištěný přepis výsledků byl zaokrouhlen na 3 desetinná místa, zatímco počítač vnitřně pracoval s 5 desetinnými

místy. Rozdíl je to velmi malý a dalo by se tak předpokládat, že bude mít velmi malý nebo žádný vliv na výsledek výpočtů. Ukázalo se však, že existují systémy, v nichž i velmi malé variace v počátečních podmínkách způsobí velké změny v dlouhodobém vývoji simulace [16].

Lorenz později v roce 1963 popsal jednoduchý model odvozený ze simulací počasí, tzv. *Lorenzův systém*. Jedná se o nelineární třídimenzionální deterministický dynamický systém odvozený ze zjednodušených rovnic vynucené konvekce v atmosféře. Zároveň by se dalo říci, že jde o jeden z nejslavnějších příkladů systémů, generujících chaotické chování. Lorenzův systém je popsán soustavou tří obyčejných diferenciálních rovnic [23]

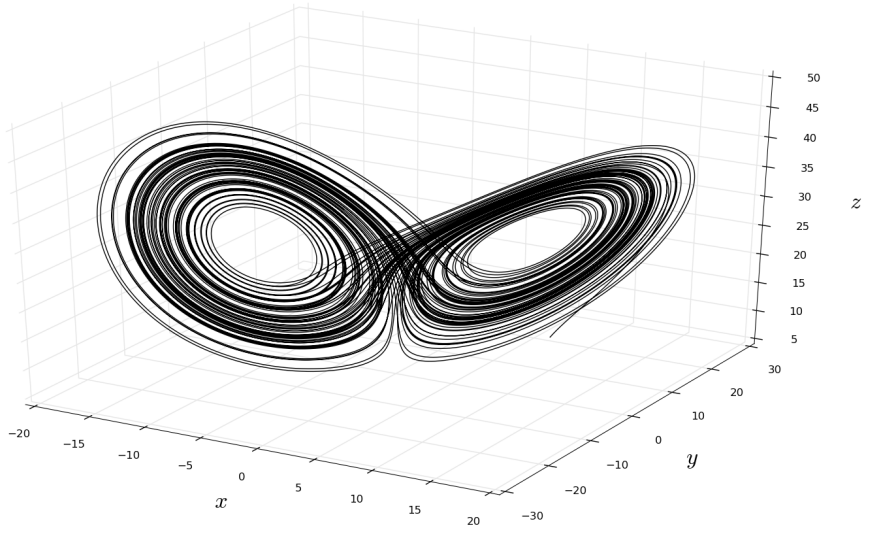
$$\begin{aligned} \dot{x} &= \sigma(y - x) \\ \dot{y} &= x(R - z) - y \\ \dot{z} &= xy - \beta z, \end{aligned} \tag{2.1}$$

kde σ je Prandtlovo číslo, R je Rayleighovo číslo a β je nepojmenovaný bezrozměný parametr. Prandtlovo číslo σ určuje poměr mezi kinematickou viskozitou a tepelnou vodivostí. Rayleighovo číslo R souvisí s konvekcí tepla v atmosféře.

V dynamických systémech mohou trajektorie končit na nízkodimenzionální podmnožině stavového prostoru, jsou k této podmnožině *přitahovány (angl. atract)*, a proto tuto podmnožinu nazýváme *atraktor*. Atraktorem ve stavovém prostoru může být bod, limitní cyklus, limitní torus nebo tzv. *podivný atraktor*. Podivný atraktor je takový, který má fraktální strukturu a většinou je výrazem chaotického chování dynamického systému. Mezi podivné atraktory patří například atraktor generovaný soustavou rovnic (2.1) nazývaný Lorenzův, který je zobrazen na obrázku 2.1. Ukázka jednoduchého skriptu v programovacím jazyku Python k řešení Lorenzova atraktoru je vložena v přílohách k této práci.

Díky Lorenzovu atraktoru se také vžil často používaný termín *butterfly effect* neboli *efekt motýlích křídel*. Toto slovní spojení vychází z podobnosti Lorenzova atraktoru s křídly motýla a také svým způsobem vyjadřuje vysokou citlivost systému na hodnoty počátečních podmínek. Pro názorné vysvětlení se často uvádí příklad, že i mávnutí motýlích křídel v Japonsku může o několik týdnů později způsobit tornádo v Americe.

Až do dnešních dnů prošel výzkum na poli nelineární dynamiky a teorie chaosu významným vývojem. Akcelerován doposud nezpomaleným rozvojem výpočetní techniky, který přesně kopíruje křivku Mooreova zákona, a také rostoucí popularitou *chaotického přístupu* mezi vědci. Dnes lze nalézt myšlenky a postupy z teorie chaosu v mnoha oblastech lidského bádání. Uplatnění nacházejí například i v ekonomii, sociologii nebo biologii [18].



Obrázek 2.1: Stavový portrét Lorenzova atraktoru pro $\sigma = 10{,}0$; $R = 28{,}0$ a $\beta = 8/3$. Řešeno v časovém intervalu 100s při kroku $h = 0{,}001$s a počátečních podmínkách $x_0 = 10{,}0$; $y_0 = 10{,}0$; $z_0 = 10{,}0$.

2.1 Stavový prostor, stavový portrét a fázový prostor

Jako *stavový prostor* je označován prostor definovaný stavovými souřadnicemi x_i a geometrické zobrazení trajektorií dynamického systému do stavového prostoru nazýváme *stavový portrét*. Můžeme se také setkat s označením fázový portrét, které se používá zejména u klasických dynamických systémů. Zároveň je zde označován prostor souřadnic x_i a hybností p_i jako fázový prostor.

Stavový portrét je mocný nástroj ke studiu chování chaotických systémů nebo i dynamických systémů obecně. Uvažujme systém, jehož vývoj je popsán diferenciální rovnicí s jednou dimenzí. Řešení můžeme zobrazit do stavového prostoru s jednou dimenzí a stavovým portrétem pak bude zobrazení hodnot stavové proměnné x_1 na přímce. Hodnoty souřadnice x_1 se na této přímce mohou shlukovat kolem nějakého bodu. Říkáme, že jsou k takovému bodu přitahovány, a tento bod pak nazýváme *atraktor*.

Dále můžeme naši úvahu rozšířit na dvě dimenze. Uvažujme vývoj systému popsaný diferenciálními rovnicemi se stavovými proměnnými x_1 a x_2. Stavovým portrétem řešení bude pak zobrazení stavových proměnných do roviny. Atraktorem tentokrát může být nejen bod, ale i křivka.

Pokud postoupíme dále k dimenzi 3 a více, začne být situace o poznání komplikovanější. Ve stavovém prostoru můžeme v různých případech nalézt nejen předchozí dva typy atraktorů, ale i tzv. *podivný atraktor* jehož korelační dimenze je neceločíselná–fraktální [3][12]. Definici pojmu korelační dimenze rozvedeme dále.

2.2 Disipativní a Hamiltonovské systémy

Termínem *disipace* označujeme ztrátu případně únik energie ze systému, nejčastěji ve formě tepla. Dynamické disipativní systémy jsou takové, u kterých se celková energie systému nezachovává a tyto systémy ve fázovém prostoru nezachovávají objem, takže dochází ke kontrakci objemu. Vlivem kontrakce objemu fázového prostoru dochází k soustřeďování trajektorií v okolí relativně malé podmnožiny prostoru nebo přímo na ní, a právě tuto podmnožinu pak nazýváme *atraktor*.

Systémy, které naopak zachovávají objem ve fázovém prostoru se nazývají *konzervativní* nebo též *Hamiltonovské*. Označení Hamiltonovské je u těchto systémů použito proto, že jejich časový vývoj lze popsat Hamiltonovými kanonickými rovnicemi, které nesou název po skotském matematikovi Williamu Hamiltonovi. Zmíněné Hamiltonovy rovnice ukazují výrazy 2.2 [7]

$$\begin{aligned} \frac{\mathrm{d}q_i}{\mathrm{d}t} &= \frac{\partial H}{\partial p_i} \\ \frac{\mathrm{d}p_i}{\mathrm{d}t} &= \frac{\partial H}{\partial q_i}, \end{aligned} \tag{2.2}$$

kde H je Hamiltonova funkce tzv. *Hamiltonián* systému, který v nejjednodušších případech odpovídá celkové mechanické energii systému. Zobecněné souřadnice jsou označeny q_i a zobecněné hybnosti p_i. V třídimenzionálním systému obsahujícím N částic platí $i \in \langle 1; 3N \rangle$. Hamiltonova funkce je s Lagrangeovou funkcí svázána vztahem

$$H = \sum_{i=1}^{N} \dot{q}_i p_i - L(\dot{q}_i, q_i, t). \tag{2.3}$$

Lagrangeova funkce je definována jako rozdíl celkové kinetické energie $T(\dot{q}_i)$ a celkové potenciální energie $V(q_i, t)$ rovnicí

$$L(\dot{q}_i, q_i, t) = T(\dot{q}_i) - V(q_i, t). \tag{2.4}$$

Otázkou je, jestli i v přírodě je možné nalézt příklady Hamiltonovských systémů. Odpověď na tuto otázku není zcela jednoznačná a v podstatě závisí na konkrétním popisu systému. Víme sice, že celková energie izolovaného systému zůstává zachována, ale její podoba se může měnit a často chování celého systému je až příliš komplexní. Proto je mnohdy schůdnější popisovat pouze nějaký subsystém a zbytek systému uvažovat jako zdroj disipace.

Jedním ze zvláštních případů disipativního systému je například naše Sluneční soustava. Pokud bychom sledovali dynamiku solárního systému na relativně krátké časové škále v řádu maximálně tisíců let, můžeme ji považovat za systém konzervativní. Efekt disipace energie se zde projevuje až v řádech milionů a více let, a proto je z krátkodobého hlediska tyto efekty možné zanedbat. Disipace probíhá například prostřednictvím slunečního větru nebo vlivem slapových sil na planety [7].

2.3 Definice chaosu

Výraz *chaos* je v běžném smyslu tohoto slova většinou chápán jako *stav neuspořádanosti*. Ovšem z matematického a fyzikálního pohledu je význam slova *chaos* diametrálně velmi odlišný. Přestože univerzální a obecně přijímaná definice neexistuje, podle [3] lze o dynamickém systému říci, že je chaotický, pokud má následující vlastnosti.

1. Chování systému je popsáno deterministicky (například soustavou diferenciálních rovnic).
2. Chování systému je aperiodické.
3. Stav systému je popsán pomocí stavových veličin. Vývoj stavových veličin generuje trajektorie ve stavovém prostoru. Tyto trajektorie jsou ve stavovém prostoru ohraničené.
4. Systém je vysoce citlivý na počáteční podmínky.

První ze zmíněných vlastností znamená, že stav systému v čase t_n je určen nějakým přesně definovaným pravidlem na základě stavu sytému v čase t_{n-1}. Příkladem takového pravidla může být výše zmíněný Lorenzův atraktor, který získáme řešením soustavy rovnic (2.1), kde i výpočet provádíme postupnými iteracemi v čase.

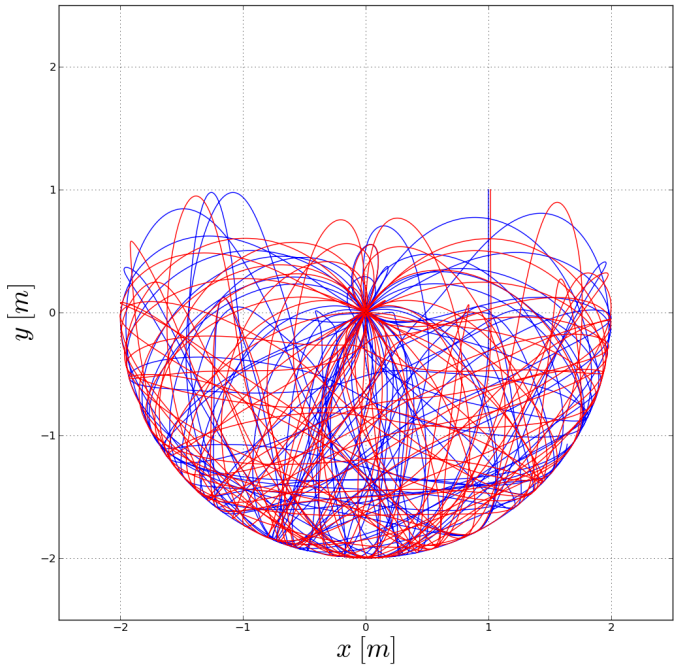
Aperiodické orbity jsou takové, které se nikdy neopakují. Tedy, že každá další orbita systému je odlišná od všech předchozích. Jako příklad lze opět brát výše zmíněný Lorenzův atraktor na obrázku 2.1, kde výsledná trajektorie v systému nikdy neprotne sama sebe. Jinými slovy, stav systému se nikdy přesně nevrátí do žádného ze stavů předchozích. S tím úzce souvisí ohraničenost orbit. Přestože jsou orbity aperiodické, nevzdalují se k nekonečnu ani se neblíží k pevnému bodu. I na Lorenzově atraktoru je dobře vidět, že trajektorie jsou kumulovány na relativně malé části stavového prostoru.

Poslední ze zmiňovaných vlastností je vysoká citlivost na počáteční podmínky, která je často označována jako *efekt motýlích křídel (butterfly effect)*. V podstatě tím říkáme, že i velmi malé variace v hodnotách počátečních podmínek vedou ve vývoji systému na velmi odlišné trajektorie.

2.4 Invarianty systému

2.4.1 Lyapunovův exponent

Pokud je systém citlivý na počáteční podmínky, můžeme říci, že dvě trajektorie ve stavovém prostoru s velmi malými rozdíly v hodnotách počátečních podmínek se od sebe během vývoje systému dříve nebo později vzdálí. Otázkou ale je, jak rychle se vzdálí. Jako příklad lze uvést systém dvojitého kyvadla. Dvojité kyvadlo je složeno ze dvou pohyblivých závěsů v řadě, mezi kterými jsou zavěšena dvě závaží. Simulaci časového vývoje provádíme na pohybových rovnicích [19], ve kterých vystupují úhly Θ_1 a Θ_2, udávající úhel mezi závěsem a svislou rovinou, a úhlové rychlosti závěsů ω_1 a ω_2. Budeme sledovat vývoj trajektorií s velmi blízkými počátečními podmínkami, například $\Theta_{1A} = \Theta_{1B} = 90^\circ$, $\Theta_{2A} = 180^\circ$ a $\Theta_{2B} = 179^\circ$. Hmotnost obou závaží zadejme $m_1 = m_2 = 1\text{kg}$ a sledujme trajektorie závaží m_2 systému pro oba případy počátečních podmínek po dobu jedné minuty. Výsledek simulace nám ukazuje stavový portrét na obrázku 2.2.



Obrázek 2.2: Srovnání trajektorií dvojitého kyvadla s velmi blízkými počátečními podmínkami.

Je dobře patrné, že i přes rozdíl počátečních podmínek pouze o 1° se trajektorie v od sebe velmi rychle vzdálí a vyvíjí se naprosto odlišným způsobem.

Nyní uvažujme v dynamickém systému dvě hodnoty počátečních podmínek x_0 a y_0 začínající blízko sebe. Jejich počáteční vzdálenost separace budeme označovat jako δ_0 a definujme ji výrazem

$$\delta_0 = | x_0 - y_0 | . \tag{2.5}$$

Vzdálenost separace po nějakém čase t označíme $\delta(t)$ a zapíšeme ji jako

$$\delta(t) = | x_t - y_t | . \tag{2.6}$$

Ukazuje se, že pro většinu systému lze chování $\delta(t)$ popsat přibližně rovnicí

$$\delta(t) \approx \delta_0 e^{\lambda t}, \tag{2.7}$$

kde koeficient λ nazýváme *Lyapunovův exponent*. Pokud je kladný, znamená to, že sledované trajektorie se od sebe vzdalují. Záporný naopak značí jejich přibližování. Čím větší nebo menší je hodnota Lyapunovova exponentu, tím rychleji se trajektorie vzdalují nebo naopak přibližují. V systému dimenze n existuje právě n Lyapunovových exponentů, protože v různých směrech se trajektorie budou vzdalovat různou rychlostí [3]. Největší z Lyapunovových exponentů pak označíme jako $\lambda_{\max}$. Se znalostí $\lambda_{\max}$ lze vypočítat tzv. *Lyapunovův čas* určující nám dobu, po kterou lze předpovídat vývoj systému [12]. Lyapunovův čas je dán vztahem

$$L_t = \frac{1}{\lambda_{\max}}. \tag{2.8}$$

Logaritmováním vztahu (2.7) získáme výraz pro výpočet Lyapunovova exponentu

$$\lambda = \lim_{t\to\infty,\delta_0\to\infty} \left(\frac{1}{t}\right) \ln\left(\frac{\delta(t)}{\delta_0}\right). \tag{2.9}$$

Pokud je největší z Lyapunovových exponentů systému kladný, značí to chaotické chování systému. V případě diskrétních systémů se předchozí vztah zjednodušuje na výraz

$$\lambda \equiv \lim_{N\to\infty} \frac{1}{N} \ln\left(\frac{\delta_N}{\delta_0}\right). \tag{2.10}$$

Nyní uvažujme dvě blízké trajektorie x a y. Jejich vývoj je dán pomocí rovnic

$$\begin{aligned} x_{i+1} &= F(x_i) \\ y_{i+1} &= F(y_i), \end{aligned} \tag{2.11}$$

generujících sadu $i = 0, 1, \ldots, N$ diskrétních časových hodnot. Pro rozdíl výsledných hodnot v daném časovém kroku můžeme psát

$$y_{i+1} - x_{i+1} = F(y_i) - F(x_i) \approx \frac{\mathrm{d}F}{\mathrm{d}x}(y_i - x_i). \tag{2.12}$$

Po N krocích lze napsat obdobný výraz

$$y_N - x_N = F^N(y_0) - F^N(x_0) \approx \left[\frac{\mathrm{d}}{\mathrm{d}x}F^N\right](x_0 - y_0), \tag{2.13}$$

kde F^N označuje $F(F(F(\ \ldots F(x)\ \ldots)))$. Vztah můžeme ještě dále rozepsat s použitím

$$\frac{\mathrm{d}}{\mathrm{d}x}F^N(x_0) = \frac{\mathrm{d}}{\mathrm{d}x}F(x_{N-1}) \cdot \frac{\mathrm{d}}{\mathrm{d}x}F(x_{N-2}) \cdot \ldots \cdot \frac{\mathrm{d}}{\mathrm{d}x}F(x_0) \tag{2.14}$$

a upravit na vzorec pro stanovení rozdílu dvou blízkých počátečních hodnot u diskrétního dynamického systému

$$\delta x_N = \left[\prod_{j=0}^{N-1} \frac{\mathrm{d}}{\mathrm{d}x}F(x_j)\right] \delta x_0. \tag{2.15}$$

Dosazením výrazu (2.15) do rovnice (2.10) obdržíme vztah pro výpočet Lyapunovova exponentu jednodimenzionálního diskrétního systému

$$\lambda = \lim_{N\to\infty} \frac{1}{N} \sum_{j=0}^{N-1} \ln\left|\, \frac{\mathrm{d}}{dx}F(x_j)\,\right|. \tag{2.16}$$

Pro n–dimenzionální systémy bychom chtěli popsat chování ve všech směrech, tedy místo $\frac{\mathrm{d}}{\mathrm{d}x}F(x_j)$ můžeme napsat Jacobiho matici derivací $\frac{\partial F}{\partial x_f}$. Pro výpočet Lyapunových exponentů tak obdržíme vztah

$$\lambda_k = \lim_{N\to\infty} \frac{1}{N} \ln \left[\prod_{j=0}^{N-1} [\mu_k(x_j)] \right], \tag{2.17}$$

kde μ_k jsou vlastní hodnoty matice $\mathscr{M}$ definované jako součin Jacobiho matic podle rovnice

$$\mathscr{M} = \mathscr{J}(x_0)\mathscr{J}(x_1) \ldots \mathscr{J}(x_{N-1}). \tag{2.18}$$

2.4.2 Korelační dimenze

Z běžné zkušenosti jsme zvyklí na celočíselné dimenze. Například přímka má dimenzi 1, čtverec má dimenzi 2 a krychle dimenzi 3. Podivné atraktory, jako například Lorenzův, mají už z definice dimenzi neceločíselnou–fraktální. Fraktály jsou *soběpodobné* geometrické objekty, které mohou být, stejně jako trajektorie v chaotických systémech, generovány relativně jednoduchými rovnicemi. Oba obory tak mají k sobě blízko, přestože jsou v mnoha ohledech rozdílné. Mezi fraktály patří například i slavná Mandelbrotova množina. Fraktály ale můžeme pozorovat i v přírodě a v běžném životě. Nacházíme je například u rostlin (větev je zmenšená podoba větší struktury stromu) nebo živočichů (tvary schránek různých měkkýšů) [3]. Soběpodobnost znamená, že menší části fraktálního objektu se podobají částem větším a tato podobnost se může dále opakovat na menší a menší části. Podivné atraktory také vykazují znaky fraktální struktury.

K přibližnému určení dimenze sledovaných trajektorií lze použít různé postupy. Díky jednoduchosti výpočtu a relativně malé výpočetní náročnosti se velmi oblíbenou stala tzv. *korelační dimenze*.

Hledanou korelační dimenzi nějakého objektu ve stavovém prostoru si označme jako d. Element objemu bychom tedy získali přibližně ze vztahu

$$V \sim L^d \tag{2.19}$$

a po zlogaritmování se dostaneme k výrazu,

$$d = \frac{\log V}{\log L}, \tag{2.20}$$

který nám udává předpis pro výpočet korelační dimenze [10][12]. V praxi výpočet provedeme tak, že si nejprve definujeme tzv. *korelační sumu* rovnicí

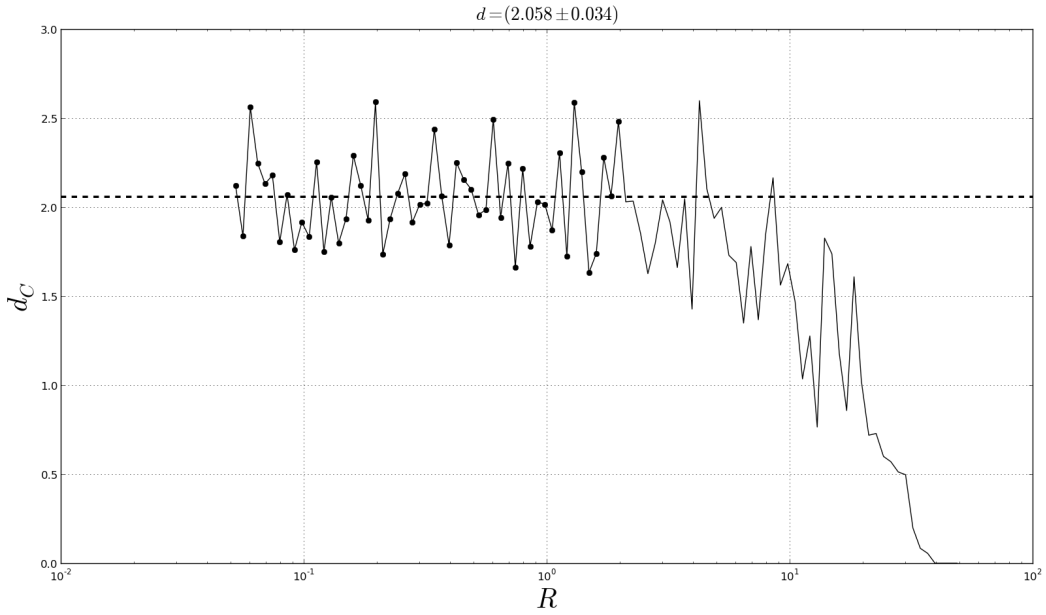
$$C(R) = \frac{1}{N} \sum_{i=1}^{N} \frac{N_i(R)}{N-1}. \tag{2.21}$$

Sledujeme trajektorii v N bodech, a proto v okolí i–tého bodu s poloměrem R se bude nacházet $N_i(R)$ bodů. Výraz uvnitř sumy nám tak udává relativní četnost bodů v okolí bodu i. Korelační suma nabývá hodnot od 0 do 1. Pokud ve sledovaném okolí R není žádný bod $C(R) = 0$ a pokud do okolí spadají všechny body, tak $C(R) = 1$. Analogicky k rovnici (2.19) nám tedy R vystupuje v roli L a $C(R)$ nám plní funkci objemu. Korelační dimenzi pak udává rovnice

$$d_C = \lim_{R \to 0} \frac{\log C(R)}{\log R}. \tag{2.22}$$

Je zcela zřejmé, že limita v rovnici (2.22) je nerealizovatelná, protože nemůžeme sledované okolí R nekonečně zmenšovat. Víme ale, že při velmi malých hodnotách R bude korelační suma rovna nule, protože se v okolí dané velikosti nebude nacházet žádný bod. Naopak ve chvíli, kdy bude okolí obsahovat všechny body, dojde k saturaci korelační sumy na 1. Když sestrojíme tzv. *log–log graf*, ve kterém zobrazíme závislost $\log C(R)$ na $\log R$, bude korelační dimenze dána směrnicí přímky fitované ve střední části křivky(zanedbáme hodnoty blízké 0 a 1), kde tato závislost přibližně odpovídá skutečnosti.

Přesněji lze korelační dimenzi určit tak, že budeme do grafu vynášet různé hodnoty korelační dimenze d_C v závislosti na okolí R. Odhad korelační dimenze poté získáme z ploché části grafu fitováním na konstantní funkci. Obrázek 2.3 ukazuje příklad určení korelační dimenze pro Lorenzův atraktor s využitím nástroje *d2* z balíku Tisean. Stejný postup budeme později aplikovat i na data získaná z našich modelů.



Obrázek 2.3: Určení korelační dimenze d Lorenzova atraktoru s dimenzí vnoření $m = 3$. Zvýrazněné body ukazují sadu dat ploché části grafu fitované konstantní funkcí.

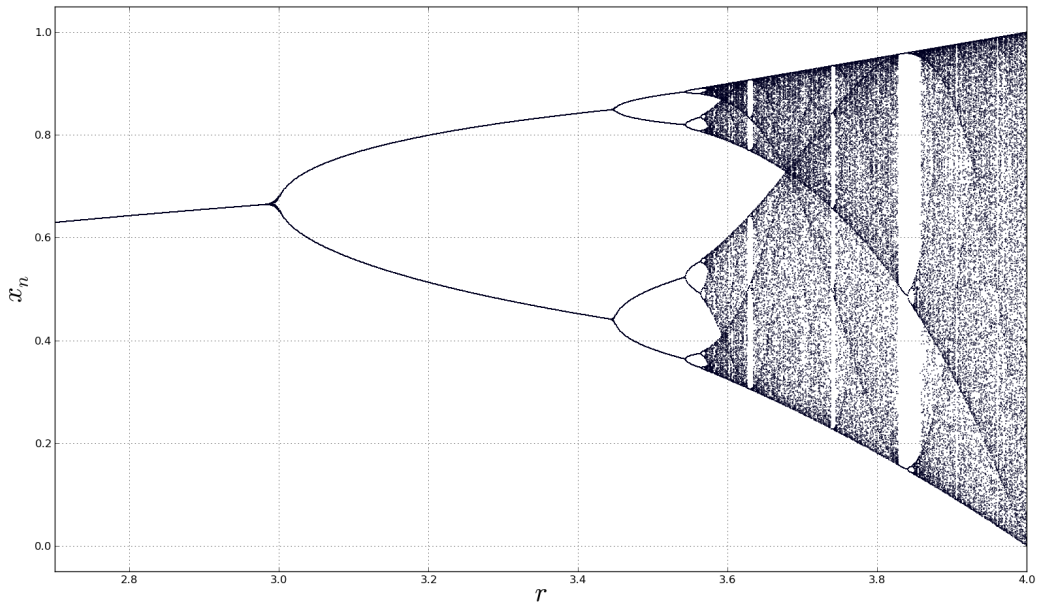
2.5 Bifurkační diagram

Při studiu dynamických systémů nám bifurkační diagram ukazuje chování hodnot ve sledovaném systému v závislosti na zvoleném bifurkačním parametru. S pomocí bifurkačního diagramu lze hledat v systému fixní body, periodické orbity, nebo například rozlišit oblasti stabilních a nestabilních řešení. Má také velký význam v rozlišení, jestli systém vykazuje chaos či nikoliv. Sestavení bifurkačního diagramu budeme demonstrovat na příkladu tzv. *logistické funkce*, kterou zapíšeme vztahem [3]

$$x_{n+1} = r x_n (1 - x_n). \tag{2.23}$$

Logistická funkce je typickým příkladem systému, u kterého velice komplexní chaotické chování vzniká i na základě jednoduché nelineární rovnice. Poprvé ji představil v roce 1838 francouzský matematik Pierre François Verhulst. V této rovnici x_n reprezentuje současnou populaci ve studovaném prostředí vyjádřenou jako poměr k maximální populaci, kterou je dané prostředí schopno uživit a nabývá hodnot mezi 0 a 1. Parametr r udává rychlost růstu sledované populace. Řešení provádíme iteracemi na vhodně zvoleném intervalu n.

Bifurkačním parametrem bude v případě logistické funkce proměnná r. Pokud je $r < 1$ populace klesá, až nakonec dosáhne $x_n = 0$. Pro hodnoty $1 \leq r \leq 4$ má logistická funkce pro některá r jedno stabilní řešení, pro některá více a pro některá přechází v čistě chaotické chování. Pomocí bifurkačního diagramu uvidíme rozložení různých řešení v závislosti na r. Výsledný bifurkační diagram logistické funkce ukazuje graf na obrázku 2.4.



Obrázek 2.4: Bifurkační diagram pro logistickou mapu. Řešeno s $\Delta r = 0{,}001$. Zobrazeny jsou stavy x_n pro $n \in \langle 250; 400 \rangle$.

Bifurkační diagram logistické funkce má smysl zobrazovat pro hodnoty $r \leq 4$, protože při hodnotách vyšších se x_n dostává mimo svůj platný interval $x_n \in \langle 0; 1 \rangle$. V grafu 2.4 můžeme pozorovat přechod systému k chaosu přes postupné zdvojování period řešení.

Nevýhodou bifurkačního diagramu může v některých případech být velká výpočetní náročnost, protože musíme propočítat řešení pro mnoho hodnot bifurkačního parametru v námi zvoleném intervalu.

3. Numerické metody řešení obyčejných diferenciálních rovnic

"It may than seem that the only choice left is computer simulation and thus one may think that the linear paradise of powerfull analytical methods is lost and what remains is just a numerical hell."

Odev Regev

Simulace dynamických systémů ve většině případů vyžadují numerické řešení diferenciálních rovnic s využitím postupných iterací. Vstupem pro výpočet příštího stavu je stav aktuální a interval, o který se posuneme, je určen jako $h \equiv \Delta t$ a nazýváme jej *iterační krok*. K aproximaci hodnot stavu následujícího je vhodné použít některou z iteračních numerických metod, jejichž aplikace by bez využití počítače byla časově velmi náročná.

Existují případy, kdy většina zvolených metod poskytne stabilní výsledky, ale speciálně u systémů vykazujících chaotické chování s vysokou citlivostí na počáteční podmínky může volba metody přímo ovlivnit výsledky simulace. Naší snahou vždy je minimalizovat chyby výpočtu. V dynamických systémech s chaotickým chováním i malé chyby vedou k obrovským rozdílům ve výpočtu v dlouhodobém měřítku.

Mějme systém se zadanými počátečními podmínkami definovaný podle

$$\dot{y} = f(t, y),\ y(t_0) = y_0. \tag{3.1}$$

Řešení (3.1) lze provést mnoha různými metodami. Naším cílem je co nejlépe aproximovat hodnoty y v závislosti na čase. Zvolme si iterační krok $h > 0$ a výpočet proveďme s využitím několika metod.

3.1 Eulerova metoda

Jedná se o jednoduchou metodu prvního řádu, kde aproximace řešení je dána dle [20]

$$\begin{aligned} y_{n+1} &= y_n + hf(t_n, y_n) \\ t_{n+1} &= t_n + h. \end{aligned} \tag{3.2}$$

Eulerova metoda je v porovnání se sofistikovanějšími metodami zmiňovanými dále poměrně nepřesná. Nevyužívá vícestupňovou aproximaci ani adaptivní krok ke zvýšení přesnosti výpočtu, a proto je pro naše potřeby naprosto nevhodná. Eulerova metoda se hodí maximálně k aproximacím řešení na krátkém intervalu nebo, díky své jednoduchosti, k výpočtům *ručně*.

3.2 Runge–Kuttova metoda čtvrtého řádu (RK4)

Na rozdíl od metody Eulerovy využívá Runge–Kuttova metoda čtvrtého řádu několik dílčích aproximací k získání hodnoty y_{n+1}. Tato metoda je definována jako [21]

$$\begin{aligned} y_{n+1} &= y_n + \frac{1}{6}h\left(k_1 + 2k_2 + 2k_3 + k_4\right) \\ t_{n+1} &= t_n + h, \end{aligned} \tag{3.3}$$

kde dílčí koeficienty k_1 až k_4 získáme podle

$$\begin{aligned} k_1 &= f(t_n, y_n) \\ k_2 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\right) \\ k_3 &= f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2\right) \\ k_4 &= f\left(t_n + h, y_n + hk_3\right). \end{aligned} \tag{3.4}$$

Runge-Kutova metoda čtvrtého řádu poskytuje poměrně dobré výsledky, ale existují i situace, kdy řešení diferenciální rovnice je numericky nestabilní, například když hledané řešení nabývá minima nebo maxima. Relativní chyby výpočtu pak velmi rychle narůstají a výpočet zkolabuje do příliš velkých nebo malých čísel. V takových případech je nutné zmenšovat velikost iteračního kroku h, aby se chyby v problematických místech udržely v únosných hodnotách pro stabilitu výpočtu. Na druhé straně příliš malý krok v oblastech řešení, kde to není potřeba jen zbytečně zvyšuje náročnost výpočtu. Tento problém řeší například dále zmiňovaná Dormand–Princova metoda.

3.3 Dormand–Princova metoda (DOPRI)

Tato metoda patří do rodiny Runge–Kutových metod. Metoda má 7 stupňů a využívá 6 dílčích aproximací k výpočtu řešení čtvrtého a pátého řádu. Rozdíl těchto řešení je brán jako chyba řešení čtvrtého řádu. Na základě této chyby lze poté implementovat algoritmus adaptivního kroku h k udržení požadované přesnosti. Řešení čtvrtého řádu v $t_n + h$ získáme podle

$$y_{n+1} = y_n + h\left(\frac{35}{384}k_1 + \frac{500}{1113}k_3 + \frac{125}{192}k_4 - \frac{2187}{6784}k_5 + \frac{11}{84}k_6\right) \tag{3.5}$$

a řešení pátého řádu v $t_n + h$ podle

$$z_{n+1} = y_n + h\left(\frac{5179}{57600}k_1 + \frac{7571}{16695}k_3 + \frac{393}{640}k_4 - \frac{92097}{339200}k_5 + \frac{187}{2100}k_6 + \frac{1}{40}k_7\right), \tag{3.6}$$

kde dílčí koeficienty k_1 až k_7 získáme podle

$$\begin{aligned}
k_1 &= f(t_n, y_n)\\
k_2 &= f\left(t_n + \frac{1}{5}h, y_n + \frac{1}{5}k_1\right)\\
k_3 &= f\left(t_n + \frac{3}{10}h, y_n + \frac{3}{40}k_1 + \frac{9}{40}k_2\right)\\
k_4 &= f\left(t_n + \frac{4}{5}h, y_n + \frac{44}{45}k_1 - \frac{56}{15}k_2 + \frac{32}{9}k_3\right)\\
k_5 &= f\left(t_n + \frac{8}{9}h, y_n + \frac{19372}{6561}k_1 - \frac{25360}{2187}k_2 + \frac{64448}{6561}k_3 - \frac{212}{729}k_4\right)\\
k_6 &= f\left(t_n + h, y_n + \frac{9017}{3168}k_1 - \frac{355}{33}k_2 - \frac{46732}{5247}k_3 + \frac{49}{176}k_4 - \frac{5103}{18656}k_5\right)\\
k_7 &= f\left(t_n + h, y_n + \frac{35}{384}k_1 + \frac{500}{1113}k_3 + \frac{125}{192}k_4 - \frac{2187}{6784}k_5 + \frac{11}{84}k_6\right).
\end{aligned} \tag{3.7}$$

Rozdíl řešení čtvrtého a pátého řádu nám určuje chybu řešení čtvrtého řádu výrazem

$$\delta y_{n+1} = | y_{n+1} - z_{n+1} | . \tag{3.8}$$

Pokud chyba přesáhne námi zvolenou mez přesnosti $\delta_{\max}$, je třeba snížit velikost iteračního kroku h, typicky zmenšit na polovinu, a výpočet provést znovu. Snižování h děláme tak dlouho, dokud není dosaženo námi požadované přesnosti.

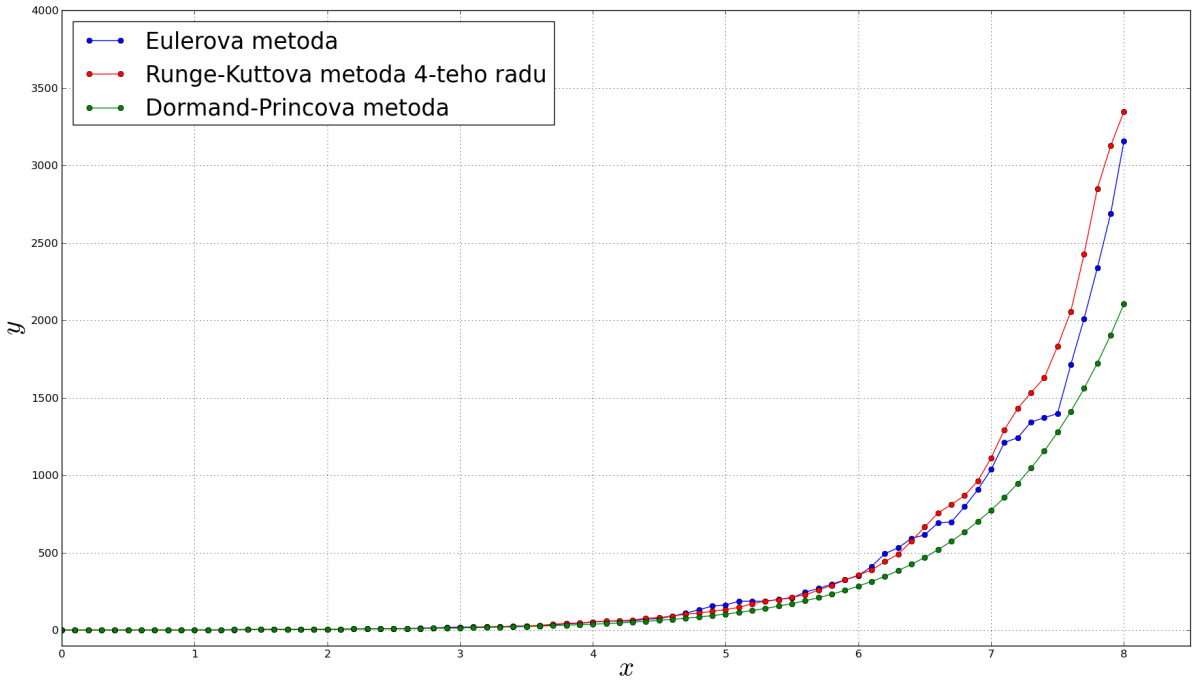
Naopak pokud chyba klesne pod námi požadovanou minimální hodnotu $\delta_{\min}$, můžeme iterační krok zvětšovat, typicky zdvojnásobovat, tak dlouho, dokud chyba není v námi požadovaném intervalu přesnosti. Tímto způsobem jsme schopni dosáhnout velmi dobré přesnosti výpočtu a zároveň snížit jeho náročnost v místech, kde je řešení stabilní.

3.4 Srovnání přesnosti numerických metod

Pro většinu problémů, na které při řešení můžeme narazit, nám budou stačit i jednodušší metody s konstantním krokem a poskytnou stabilní a přesné výsledky. Existují ale případy obyčejných diferenciálních rovnic se silným tlumením, v angličtině označované jako *stiff equations*, u kterých jednoduché metody mnohdy i po několika málo krocích ztratí stabilitu a výsledky pak neodpovídají skutečnému řešení [22]. K tomuto účelu je pak třeba využít některou ze složitějších metod s adaptivním krokem. Názorně lze problém ilustrovat na diferenciální rovnici

$$\begin{aligned} \frac{\mathrm{d}y}{\mathrm{d}x} &= e^x + \sin x + y\cos(2y) \\ y(0) &= 1.0, \end{aligned} \tag{3.9}$$

která je příkladem rovnice se silným tlumením (stiff equations). Její řešení provedeme s použitím výše popsaných metod s totožnými parametry výpočtu a budeme sledovat, jaké výsledky jednotlivé metody poskytnou zobrazením do grafu na obrázku 3.1. Jednoduchý script v jazyce Python k řešení tohoto příkladu je uveden v přílohách.



Obrázek 3.1: Srovnání různých metod numerických řešení obyčejné diferenciální rovnice 3.9. Řešeno v intervalu $x \in \langle 0; 8 \rangle$ s iteračním krokem $h = 0{,}1$.

Z grafu 3.1 je patrné, že metody bez adaptivního kroku pro hodnoty $x > 4$ začínají ztrácet stabilitu a postupné hromadění numerických chyb vede k výsledkům, které neodpovídají skutečnému řešení. Naproti tomu Dormand-Princova metoda, využívající adaptivního kroku, ve sledovaném případě poskytuje relativně velmi přesné výsledky, které v podstatě překrývají skutečné analytické řešení obyčejné diferenciální rovnice.

Je ještě nutné zmínit, že Dormand-Princova metoda využívá adaptivního kroku jen tehdy, když numerická nepřesnost překročí požadovanou maximální hodnotu a jako výsledky se zobrazují jen hodnoty po celých krocích $h = 0{,}01$. Bylo by samozřejmě možné dosáhnout stejné přesnosti i s použitím ostatních dvou metod, ovšem jedině za cenu stále jemnějšího kroku a tím pádem časově náročnějšího výpočtu. Výhodou Dormand-Princovy metody tak je dobrý kompromis mezi přesností a náročností výpočtu, protože ke zmenšování iteračního kroku dochází jen když je to potřeba.

4. Modely kapajícího kohoutku

"Models are not right or wrong, they are always wrong."
Gavin Schmidt

Formování a chování kapek nejrůznějších kapalin je velmi komplikovaný fenomén, který můžeme pozorovat takřka každý den. Mohlo by se tedy zdát, že tento jev bude fyzikálně velmi dobře popsán, ale opak byl donedávna pravdou. Přestože se studie na toto téma datují až do 17. století, výrazný pokrok byl učiněn teprve nedávno [4]. Jeans Eggers navrhl v roce 1993 teorii univerzálně použitelnou k popisu osově symetrických viskózních kapek [2]. Následná numerická řešení se s vysokou přesností shodují s experimentálně pozorovanými tvary kapek.

Robert Shaw svou přelomovou prací [15] ukázal, že kapající kohoutek z dlouhodobého hlediska může vykazovat jak periodické tak i chaotické chování. Hlavním kontrolním parametrem v systému kapajícího kohoutku je množství přitékající kapaliny do formující se kapky. I velmi malé variace v přítoku se projevují výraznými změnami v atraktoru systému.

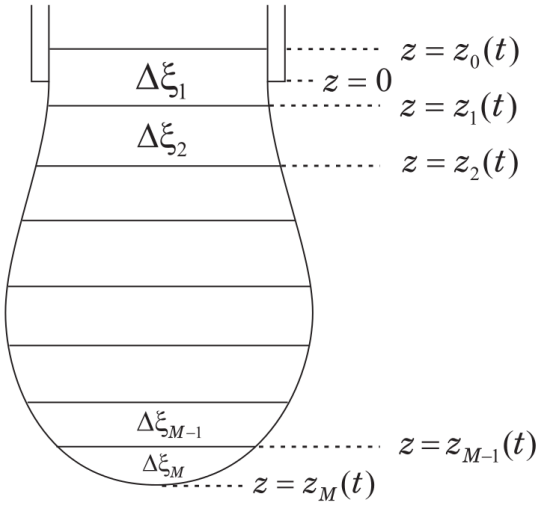
K tvorbě modelu kapajícího kohoutku je možné přistupovat více způsoby. Například z popisu kapaliny pomocí Navier-Stokesových rovnic nebo z rovnic Lagrangeových jako diskretizovaný systém pod vlivem gravitace, povrchového napětí a viskozity. Model označovaný jako *Mass–Spring Model* (dále jen MSM), který původně navrhl Shaw [15], je určen rovnicemi

$$\begin{aligned} \frac{d}{dt}\left(m\frac{dz}{dt}\right) &= -kz - \gamma\frac{dz}{dt} + mg \\ \frac{dm}{dt} &= Q = konst., \end{aligned} \tag{4.1}$$

kde z je souřadnice těžiště formující se kapky, m její hmotnost, g gravitační zrychlení, k určuje tuhost pružiny a závisí na aktuální hmotnosti kapky a γ je tlumící parametr [11]. Zjednodušeně řečeno, MSM vlastně aproximuje chování kapky jako zaváží na pružině. Hmotnost m je v průběhu času zvyšována konstantním přítokem Q, který závisí především na průměru uvažovaného kohoutku. Pokud poloha těžiště kapky z dosáhne určité zvolené kritické hodnoty, je část hmoty oddělena v závislosti na její aktuální rychlosti. Chování modelu se odvíjí od nastavení parametrů k, g a γ a samozřejmě nastavení počátečních podmínek. Ovšem hodnoty jednotlivých parametrů nebyly z provedených simulací přesně určeny.

K lepšímu pochopení procesu formování kapky a přesnějšímu určení řídících parametrů MSM je lépe použít model sofistikovanější. Tím může být například model, který budeme zkráceně označovat jako FDM (z anglického *Fluid Dynamical Model*). Ten uvažuje formující se kapku

jako diskrétní sadu vzájemně vázaných disků. Hlavním kontrolním parametrem je opět hodnota přítoku Q. Vysvětlující schéma rozdělení kapky na sadu disků je na obrázku 4.1.



Obrázek 4.1: Schéma rozdělení kapky na M disků pomocí $M-1$ horizontálních rovin. Součadnice z_0 je zvolena a určuje horní hranici modelu. $\Delta\xi_j$ označuje objem konkrétního disku. Schéma převzato z [4].

Přestože oba modely spolu úzce souvisí, jsou vhodné pro víceméně odlišné účely. Zatímco jednodušší MSM se hodí ke studiu chování kapajícího kohoutku na řádově mnohem větších časových škálách, FDM je vhodnější spíše k pochopení formování, vývoje tvaru a také procesů vedoucích k odtržení kapky. Tento fakt vychází v prvé řadě z mnohem větší výpočetní náročnosti FDM. Při numerických výpočtech systému MSM řešíme jedinou diferenciální rovnici druhého řádu, protože model uvažuje pouze pohyb těžiště kapky. Naproti tomu u FDM spočívá simulace v řešení soustavy desítek až stovek diferenciálních rovnic druhého řádu (pro každý disk).

Protože k správnému nastavení parametrů MSM je vhodné nejprve dobře pochopit a interpretovat výsledky simulace FDM, bude právě tento sofistikovanější model ten, kterým začneme. Prvním krokem při tvorbě tohoto modelu je řešení rovnovážných stavů kapek. Jedná se vlastně o tvary visících kapek v klidu v čase $t = 0$, které pak využijeme jako vstupní data do simulace FDM.

4.1 Řešení rovnovážných stavů

Rovnovážný stav kapky je takový, kdy se vzájemně vyrovnají síly působením gravitace a povrchového napětí kapaliny. Můžeme uvažovat tak, že musí existovat nějaká kritická hodnota celkového objemu kapky V_c a potažmo její hmotnosti. Pokud tato kritická hodnota bude překročena je rovnovážný stav neudržitelný.

Před samotným řešením je nutné dobře definovat systém v klidu visící kapky. Budeme vycházet z definice zavedené v [4]. Uvažujme kapku, která je osově symetrická a hustota kapaliny ρ je v celém jejím objemu konstantní. Tlak v kapce v závislosti na vertikální souřadnici z můžeme popsat výrazem

$$P = \rho g z. \tag{4.2}$$

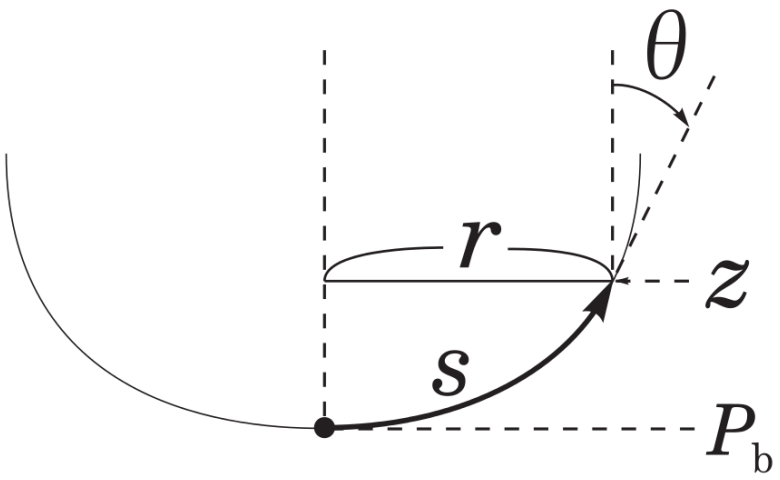
Rozdíl tlaku na rozhraní kapky uvnitř a vně lze vyjádřit pomocí poloměrů křivosti R_1 a R_2 povrchu kapky jako rovnici

$$P = \Gamma \left(\frac{1}{R_1} + \frac{1}{R_2} \right), \tag{4.3}$$

kde Γ je povrchové napětí kapaliny kapky. Díky vhodně zvoleným základním jednotkám délky, tlaku, hmotnosti a času můžeme říci, že $g = \rho = \Gamma = 1$. Zde g vyjadřuje gravitační zrychlení a ρ hustotu kapaliny tvořící kapku. Tyto základní jednotky jsou definovány vztahy

$$\begin{aligned} l_0 &\equiv \sqrt{\frac{\Gamma}{\rho g}} = 0{,}27\text{cm} \\ m_0 &\equiv \rho l_0^3 = 0{,}02\text{g} \\ P_0 &\equiv \sqrt{\rho g \Gamma} = 270\text{dyn} \cdot \text{cm}^{-2} \\ t_0 &= 0{,}017\text{s}. \end{aligned} \tag{4.4}$$

Řešení je prováděno v soustavě CGS, a proto i my budeme simulace provádět v CGS, aby bylo možné výsledky efektivně porovnávat s výsledky v [4]. Nadále tak budeme v modelech používat bezrozměrné jednotky, ze kterých lze jednoduše dopočítat reálné hodnoty podle (4.4), například kvůli srovnání s reálnými experimenty. Proměnné v systému visící kapky definuje schéma na obrázku 4.2.



Obrázek 4.2: Definice proměnných v systému visící kapky v klidu. Zde s je vzdálenost měřená podél tvaru kapky, z vertikální souřadnice směřující v kladném směru dolů, r poloměr kapky a θ je úhel, který svírá tečna v daném místě s osou z. Schéma převzato z [4].

Poloměry křivosti R_1 a R_2 jsou dány vztahy

$$\begin{aligned}\frac{1}{R_1} &= -\frac{\mathrm{d}\Theta}{\mathrm{d}s} \\ \frac{1}{R_2} &= \frac{\cos\Theta}{r}.\end{aligned} \tag{4.5}$$

Poté lze vyjádřit soustavu rovnic

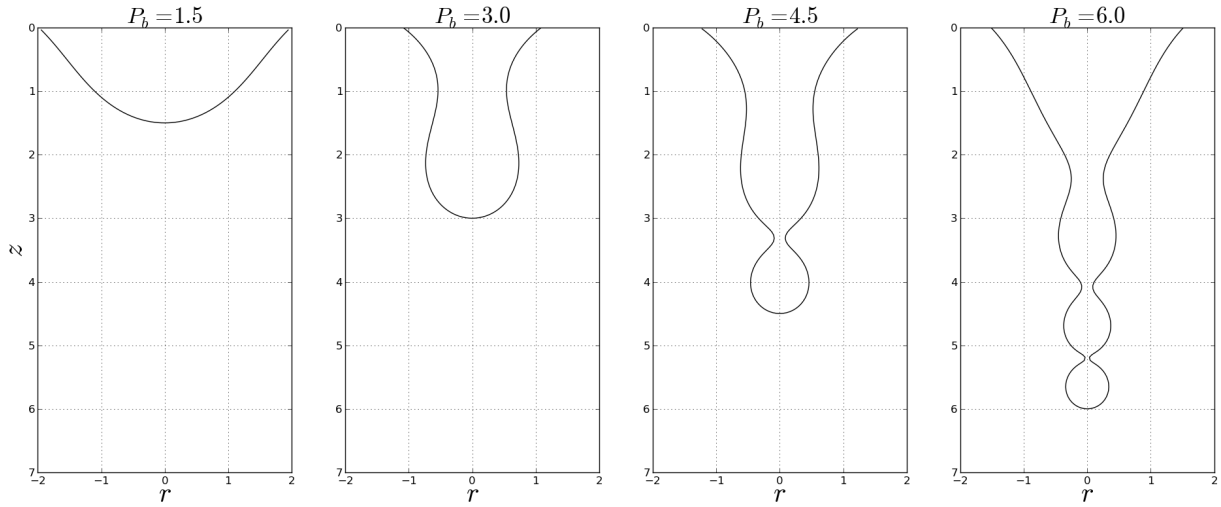
$$\begin{aligned}\frac{dr}{ds} &= \sin\theta \\ \frac{dz}{ds} &= -\cos\theta \\ \frac{d\theta}{ds} &= \frac{\cos\theta}{r} - z,\end{aligned} \tag{4.6}$$

jejímž řešením získáme křivky rovnovážných tvarů kapek. Numerické řešení rovnic 4.6 provedeme s použitím počátečních podmínek

$$\begin{aligned}z(0) &= P_b \\ \theta(0) &= \pi/2 \\ r(0) &= 1 \cdot 10^{-20},\end{aligned} \tag{4.7}$$

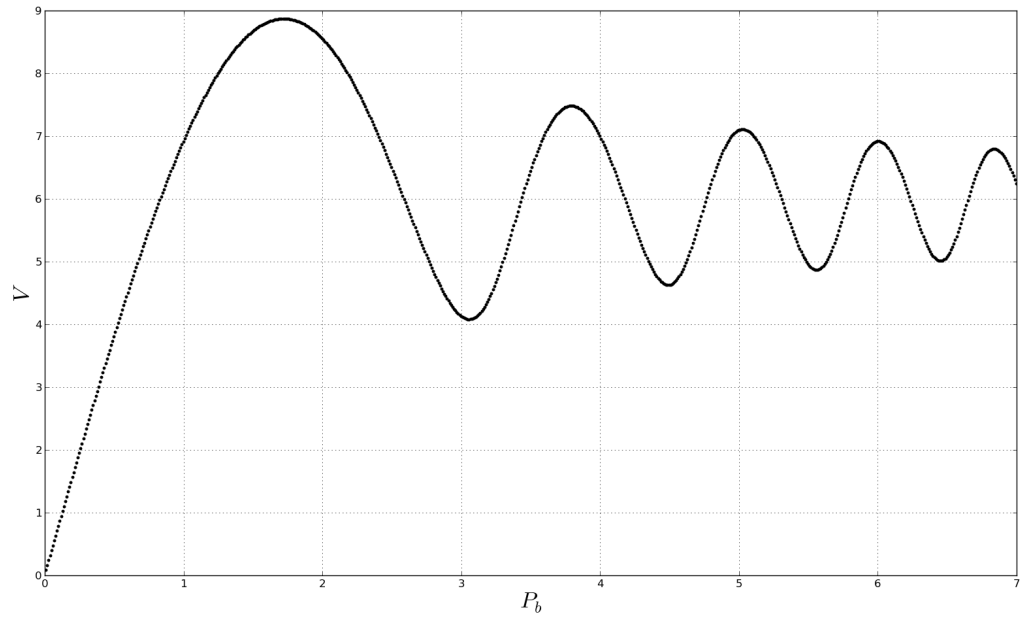
kde je ještě nutné si uvědomit, že nelze zvolit $r(0) = 0$, protože takový případ je výpočetně neřešitelný. Proto je nutné použít nějakou velice malou hodnotu r blízkou nule. V našem případě použijeme $r(0) = 1 \cdot 10^{-20}$. Jako počáteční podmínku z budeme zadávat tlak na spodku kapky P_b, protože díky volbě základních jednotek je $z_M \equiv P_b$. Výpočet pak provedeme pro různé hodnoty P_b v intervalu $P_b \in \langle 0{,}01; 7{,}0\rangle$ s krokem $\Delta P_b = 0{,}01$. Jednotlivá řešení provedeme na integračním oboru $s \in \langle 0{,}0; 8{,}0\rangle$ s krokem $\Delta s = 0{,}0008$.

Pro potřeby výpočtu byl vytvořen jednoduchý objektový kód v programovacím jazyku Python, který je součástí příloh k této práci. Výsledné tvary omezíme horní hranicí $z = 0$ a získáme tak rovnovážné tvary kapek přiléhající k nekonečné rovině $z = 0$, můžeme říci, že se jedná o *kapky na stropě*. Příklady několika takových řešení ukazuje obrázek 4.3. Zde konkrétně pro hodnoty $P_b = 1{,}5$; $P_b = 3{,}0$; $P_b = 4{,}5$ a $P_b = 6{,}0$.



Obrázek 4.3: Příklady řešení rovnovážných tvarů kapek pro různé hodnoty P_b.

Dále se můžeme ptát, jaká je závislost celkového objemu kapky V na jejím tlaku P_b a jaká je výše zmiňovaná hodnota kritického objemu kapky V_c. K tomuto účelu vyneseme do grafu závislost $V_c = f(P_b)$. Tuto závislost ukazuje graf na obrázku 4.4.



Obrázek 4.4: Závislost $V = f(P_b)$. Každý bod reprezentuje jedno řešení visící kapky shora omezené rovinou $z = 0$.

Z grafu 4.4 můžeme jasně vidět, že existují řešení, která mají stejnou hodnotu celkového objemu V při rozdílné hodnotě tlaku na spodku kapky P_b. Jak ukázali J. F. Padday a A. R. Pitt, je stabilní

a realizovatelné vždy jen jedno z takových řešení [14]. V našem případě budou stabilní řešení v intervalu $P_b \in \langle 0{,}0; 1{,}72 \rangle$. Přičemž k $P_b = 1{,}72$ náleží kritická hodnota objemu $V_c = 8{,}87$.

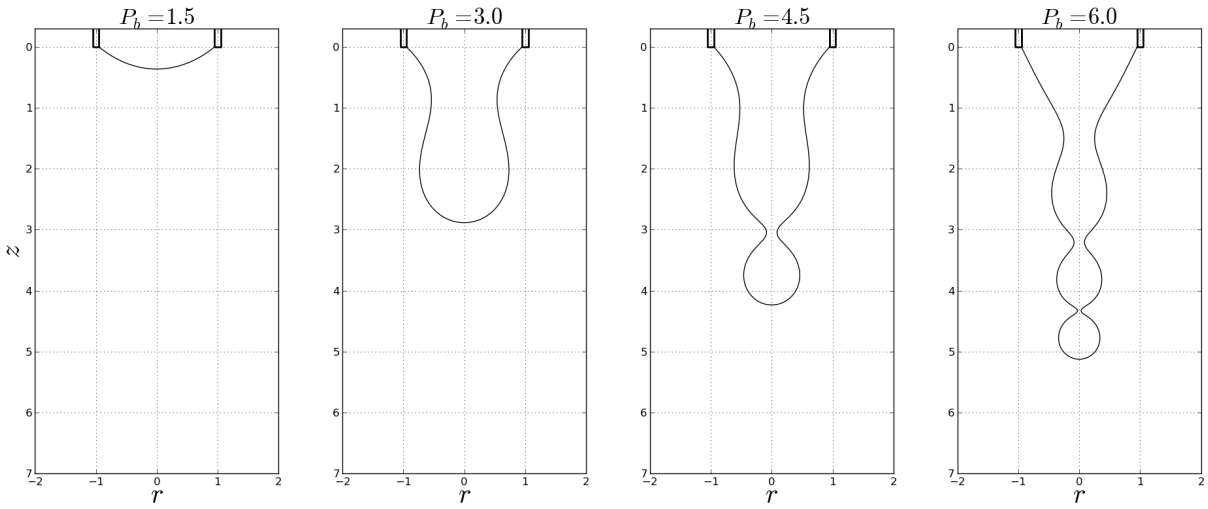
Tvary kapek přiléhající k rovině $z = 0$ ale nejsou příliš vhodné jako vstup do následné dynamické simulace, protože přítok Q, který je v simulaci hlavním kontrolním parametrem, není v systému dobře definovaný pomocí rychlosti kapaliny. Je proto třeba, abychom získané tvary dále omezili nejen rovinou $z = 0$, ale také pomocí vazby na kohoutek prostřednictvím jeho poloměru, který budeme označovat jako r_a.

Omezení tvarů kapek v podmínkách $z_j \leqq 0$ a $r_j \approx r_a$ je opět implementováno v přiloženém kódu. Pro každé řešení kapek na stropě tak získáme nejméně jeden možný tvar pro kapku na kohoutku, ale v závislosti na volbě poloměru kohoutku lze nalézt i více bodů, které splňují podmínky vazby.

Bod na křivce tvaru kapky, kde jsou splněny podmínky vazby, označme například jako C a jeho souřadnice $(r_C; z_C)$. Pro souřadnice bodu C platí $r_C \approx r_a$ a $z_C > 0$. Protože chceme, aby pro vazbu platilo $z_C = 0$, je třeba všechny body z_j na křivce tvaru posunout o hodnotu z_C podle vztahu

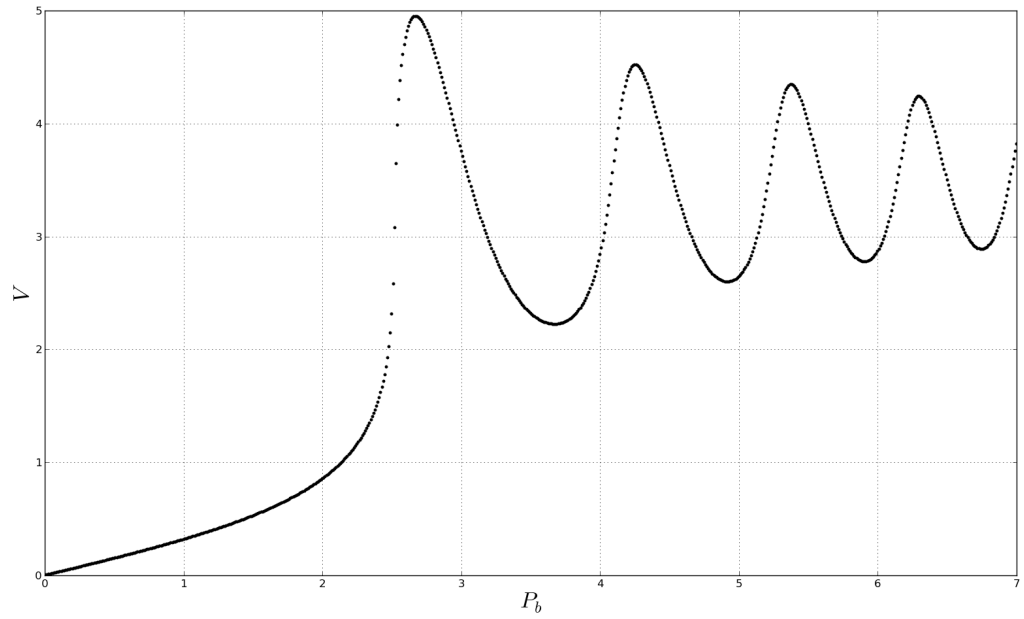
$$z'_j = z_j - z_C. \tag{4.8}$$

Příklady několika takto získaných tvarů ukazuje graf na obrázku 4.5. Jsou zvoleny stejné hodnoty P_b jako na obrázku 4.3.



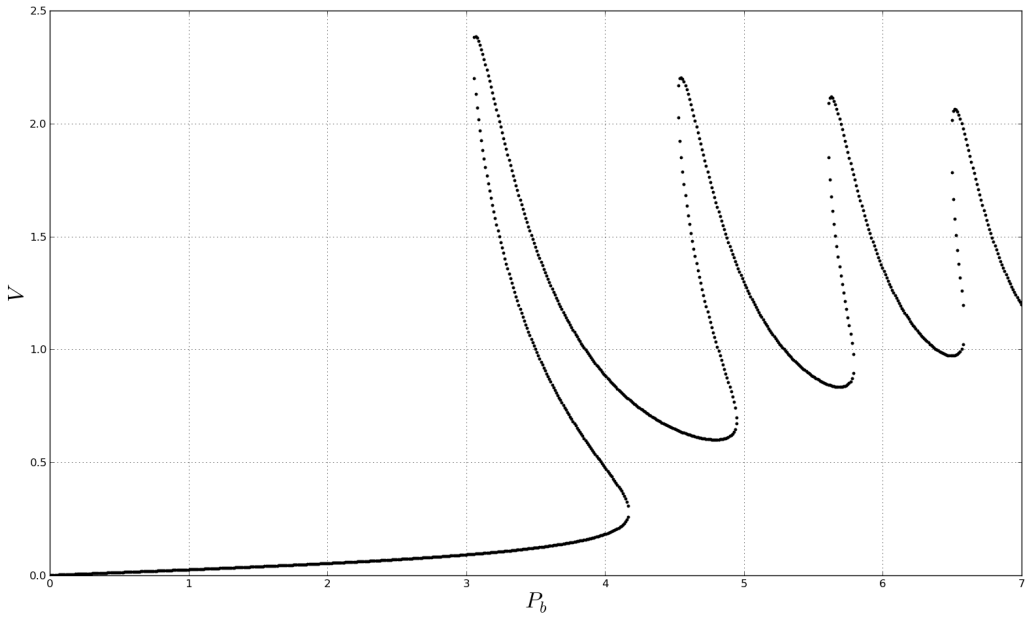
Obrázek 4.5: Příklady řešení rovnovážných tvarů kapek pro různé hodnoty P_b pro vazbu s volbou $r_a = 0{,}952$.

Stejně jako u kapek na stropě nás u kapek na kohoutku bude zajímat závislost $V = f(P_b)$, ze které následně určíme hodnotu kritického objemu V_c. Tato závislost je vynesena v grafu 4.6.



Obrázek 4.6: Závislost $V = f(P_b)$. Každý bod reprezentuje jedno řešení visící kapky na kohoutku s poloměrem $r_a = 0{,}952$.

Ze závislosti 4.6 lze opět určit interval stability, který je zde $P_b \in \langle 0{,}0; 2{,}67 \rangle$ s hodnotou kritického objemu $V_c = 4{,}95$. Protože poloměr kohoutku r_a je důležitý parametr, který určuje jednak vazbu kapky ke kohoutku a jednak se od něho odvíjí hodnota přítoku Q, bude nás nyní zajímat jakým způsobem ovlivňuje hodnota r_a závislost $V = f(P_b)$. Výpočet provedeme ještě jednou s volbou například $r_a = 0{,}5$. Výsledný graf ukazuje obrázek 4.7.



Obrázek 4.7: Závislost $V = f(P_b)$. Každý bod reprezentuje jedno řešení visící kapky na kohoutku s poloměrem $r_a = 0{,}5$.

Je vidět, že snížení poloměru r_a se projeví v prvé řadě změnou hodnoty kritického objemu V_c, který výrazně poklesl na 2.39. Zároveň ale pozorujeme, že interval stability v P_b již není definován jako v předešlém případě pomocí minimální a maximální hodnoty. Tentokrát jako stabilní lze označit pouze stavy na křivce před dosažením kritické hodnoty obejmu V_c. Tato kritická hodnota pro nás později bude velmi důležitá, protože jako vstupní data do dynamického kapalinového modelu budeme používat kapku s kritickou hodnotou nebo hodnotou blížící se kritické hodnotě na stabilní části křivky $V = f(P_b)$. Totéž platí i pro předešlý případ s $r_a = 0{,}952$.

4.2 Dynamický kapalinový model (FDM)

Z řešení rovnovážných stavů jsme získali vstupní data do dynamického kapalinového modelu, který zkráceně označujeme jako FDM (z angl. *Fluid Dynamical Model*). Uvažujeme tedy kapku visící na kohoutku poloměru r_a, jejíž objem je roven kritickému objemu V_c. Tento tvar je zvolen proto, že se jedná o kapku, která je těsně před přechodem k nestabilitě a její vývoj do okamžiku odtržení tak bude nejkratší. Vstupní tvar rozdělíme na M disků. Každý takový disk bude popsán pomocí souřadnice z_j, rychlosti $\dot{z}_j$ a hmotnosti m_j, kde $j \in \langle 1; M \rangle$. Rozdělení je realizováno tak, že všechny disky budou mít stejnou délku Δs měřenou podél tvaru kapky, v našem případě $\Delta s = 0{,}08$. Hmotnosti jednotlivých disků získáme jako $m_j = \rho \Delta \xi_j$. Protože ale uvažujeme $\rho = 1$, lze výraz zjednodušit na $m_j \equiv \Delta \xi_j$.

4.2.1 Lagrangián diskretizovaného systému kapky

Lagrangián systému kapky složené z M disků zapíšeme ve tvaru

$$L = E_k - U_g - U_\Gamma, \tag{4.9}$$

kde E_k je celková kinetická energie kapky, U_g je její celková potenciální energie a U_Γ její energie povrchová. Proměnné v systému jsou definovány podle obrázku 4.2. Kinetickou energii E_k vyjádříme jednoduše jako sumu kinetických energií jednotlivých disků vztahem

$$E_k = \frac{1}{2}\sum_{j=1}^{M} \Delta\xi_j \dot{z}_j^2. \tag{4.10}$$

Potenciální energii U_g zapíšeme opět jako sumu potenciálních energií jednotlivých disků vztahem

$$U_g = -g\sum_{j=1}^{M} \Delta\xi_j z_j. \tag{4.11}$$

Jako poslední potřebujeme znát výraz pro povrchovou energii U_Γ. Protože přesný výraz je značně složitý a obsahuje derivace vyšších řádů v prostorových souřadnicích, budeme povrchovou energii aproximovat výrazem

$$U_\Gamma = \Gamma S, \tag{4.12}$$

kde S je vyjadřuje celkovou plochu kapky. Tuto plochu budeme podle [4] počítat jako sumu dílčích ploch S_j. Plochu S_j v intervalu $\langle (z_{j+1}+z_j)/2; (z_j+z_{j-1})/2\rangle$ vyjádříme jako obsah pláště komolého kuželu a po úpravách dostaneme

$$S_j = \pi(r_j + r_{j+1})\sqrt{\frac{1}{4}(z_{j+1}-z_{j-1})^2 + (r_j - r_{j+1})^2}. \tag{4.13}$$

Problematickými místy jsou disky $j = 1$ a $j = M$. V případě $j = M$ nelze dosadit r_{j+1} ani z_{j+1}. Víme ale, že hodnota $z_M - z_{M-1}$ je velmi malá, a proto budeme vyjadřovat S_M jako obsah kruhu s poloměrem r_M rovnicí

$$S_M = \pi r_M^2. \tag{4.14}$$

V případě disku $j = 1$ platí vztah (4.13) pouze pokud $z_1 =\mid z_0 \mid$, kdy pro horní hranici plochy S_1 platí

$$\frac{z_0 + z_1}{2} = 0. \tag{4.15}$$

S rostoucím časem se ale tato hranice posunuje rychlostí v_1 do kladných hodnot, protože pak platí $z_1 >\mid z_0 \mid$. V celkové ploše S by tak chyběl právě úsek mezi nulou (hranice kohoutku) a hranicí plochy S_1. Vzorec (4.13) proto můžeme použít pouze pro výpočet ploch disků $j \in \langle 2; M-1\rangle$. Plochu S_1 pak vypočteme v intervalu $\langle (z_2+z_1)/2; 0\rangle$ obdobným vzorcem

$$S_1 = \pi(r_1 + r_a)\sqrt{(r_1 - r_a)^2 + z^2{}_1} + \frac{\pi}{2}(3r_1 + r_2)\sqrt{\frac{1}{4}(r_2 - r_1)^2 + \frac{1}{4}(z_2 - z_1)^2}, \tag{4.16}$$

kde r_a reprezentuje poloměr kohoutku. Dosadíme rovnice (4.13), (4.14) a (4.16) do rovnice (4.12) a pro výpočet celkové plochy S obdržíme výraz

$$S = S_1(r_j, r_{j+1}, z_j, z_{j+1}) + \sum_{j=2}^{M-1} S_j(r_j, r_{j+1}, z_{j-1}, z_j, z_{j+1}) + S_M(r_M). \tag{4.17}$$

Poloměry jednotlivých disků r_j, které vystupují ve výpočtech dílčích ploch, nejsou konstantní, ale jsou funkcí souřadnic z_j. Objem $\Delta\xi_j$ se pro každý disk zachovává, ale současně víme, že se disky v různých částech kapky budou pohybovat různými rychlostmi, a proto i výška každého disku $z_j - z_{j-1}$ se bude měnit, a tím pádem se bude měnit i jeho poloměr. Poloměr disků vyjádříme jako

$$r_j = \sqrt{\frac{\Delta\xi_j}{\pi(z_j - z_{j-1})}}. \tag{4.18}$$

Problematickým místem je disk $j = 1$, protože před začátkem simulace posuneme hodnoty z_j tak, že platí $z_0 < 0 < z_1 < z_j \ldots < z_M$, a hodnota z_0 nám poslouží jako horní okrajová podmínka, která bude přesněji definována dále. K výpočtu poloměru r_1 tedy využijeme pouze část objemu $\Delta\xi_1$, která se nachází v intervalu $\langle 0; z_1\rangle$, protože poloměr pod hranicí kohoutku má konstantní hodnotu r_a, získáme r_1 podle rovnice

$$r_1 = \sqrt{\frac{\Delta\xi_1 - \pi r_a^2 \mid z_0 \mid}{\pi z_1}}. \tag{4.19}$$

Dosazením (4.18) a (4.19) do (4.17) získáme výraz pro výpočet celkové plochy S závislý pouze na souřadnicích z_j

$$S = S_1(z_{j-1}, z_j, z_{j+1}) + \sum_{j=2}^{M-1} S_j(z_{j-1}, z_j, z_{j+1}) + S_M(z_M, z_{M-1}). \tag{4.20}$$

Lagrangián našeho diskretizovaného systému kapky má výsledný tvar

$$L = \frac{1}{2}\sum_{j=1}^{M} \Delta\xi_j \dot{z}_j^2 + g\sum_{j=1}^{M} \Delta\xi_j z_j - \Gamma S\ . \tag{4.21}$$

4.2.2 Pohybové rovnice

Nyní známe Lagrangián diskretizovaného systému kapky. Pohybové rovnice získáme z Lagrangeovy rovnice

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\partial L}{\partial \dot{z}_j} = \frac{\partial L}{\partial z_j} + \frac{1}{2}\frac{\partial \dot{E}_k}{\partial \dot{z}_j}, \tag{4.22}$$

kde poslední člen vyjadřuje disipativní funkci, která nám udává vliv viskozity na chování systému. Disipativní funkce je podle [4] dána rovnicí

$$\dot{E}_k = \sum_{j=1}^{M} \frac{(\dot{z}_j - \dot{z}_{j-1})}{(z_j - z_{j-1})} \Delta\xi_j. \tag{4.23}$$

Řešení rovnice (4.22) provedeme následujícím způsobem. Nejprve vypočteme derivace na pravé straně a výsledek zapíšeme jako

$$\Delta\xi_j \ddot{z}_j = \frac{\partial L}{\partial z_j} + \frac{1}{2}\frac{\partial \dot{E}_k}{\partial \dot{z}_j}. \tag{4.24}$$

Protože derivace na pravé straně by bylo velmi zdlouhavé počítat ručně a také velmi náchylné na chyby, vyjádříme zrychlení $\ddot{z}_j$ jako

$$\ddot{z}_j = \frac{1}{\Delta\xi_j}\left(\frac{\partial L}{\partial z_j} + \frac{1}{2}\frac{\partial \dot{E}_k}{\partial \dot{z}_j}\right), \tag{4.25}$$

kde derivace na pravé straně budeme počítat pomocí balíku SymPy na začátku simulace a také v případě, že se změní počet disků M (například zvýší přítokem z kohoutku). Získáme tak soustavu M diferenciálních rovnic druhého řádu. Abychom ji mohli numericky řešit, převedeme ji na soustavu $2M$ diferenciálních rovnic prvého řádu

$$\begin{aligned} \dot{z}_j &= v_j \\ \dot{v}_j &= \frac{1}{\Delta\xi_j}\left(\frac{\partial L}{\partial z_j} + \frac{1}{2}\frac{\partial \dot{E}_k}{\partial v_j}\right). \end{aligned} \tag{4.26}$$

Dále je ještě třeba specifikovat horní okrajovou podmínku systému. Víme, že kapalina z kohoutku přitéká konstantní rychlostí v_0, a tím pádem i disk $M = 0$ se pohybuje bez zrychlení rychlostí v_0. Soustavu tedy upravíme do výsledného tvaru

$$\begin{aligned} \dot{z}_0 &= v_0 \\ \dot{v}_0 &= 0 \\ \dot{z}_j &= v_j \\ \dot{v}_j &= \frac{1}{\Delta\xi_j}\left(\frac{\partial L}{\partial z_j} + \frac{1}{2}\frac{\partial \dot{E}_k}{\partial v_j}\right). \end{aligned} \tag{4.27}$$

4.2.3 Simulace vývoje v čase

Soustavu rovnic (4.27) není možné řešit analyticky, protože všechny rovnice jsou vzájemně provázané. Je tedy nutné použít některou z numerických metod řešení obyčejných diferenciálních rovnic. My k tomuto účelu využijeme Dormand–Pricovu metodu s adaptivním krokem, která byla podrobněji popsána v kapitole 2.6.3. Použití právě této metody má několik výhod. První výhodou je, že iterační krok je adaptivně přizpůsobován aktuálním potřebám výpočtu, což vede ke spolehlivějším výsledkům, a druhou výhodou je, že tato metoda je hojně implementována v knihovnách a výpočetních balících různých programovacích jazyků a její použití je tak mnohem pohodlnější.

V programovacím jazyce Python se k numerickému řešení obyčejných diferenciálních rovnic výborně hodí nástroje obsažené v knihovně SciPy, jejíž součástí je i výše zmíněná Dormand–Princova medota s širokými možnostmi nastavení.

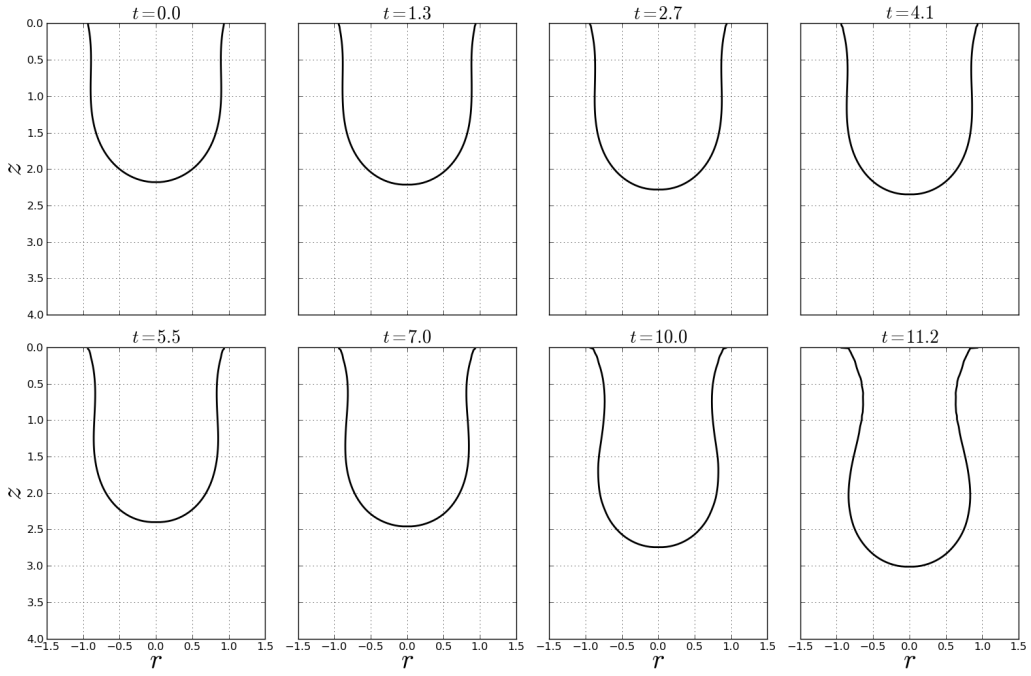
K samotné simulaci byl vytvořen program v jazyce Python, který je součástí příloh. S použitím tohoto scriptu provedeme ukázkový příklad simulace s nastavenými parametry

$$\begin{aligned} r_a &= 0{,}943 \\ v_0 &= 0{,}01 \\ h &= 0{,}01 \\ \eta &= 0{,}002, \end{aligned} \tag{4.28}$$

kde r_a je poloměr kohoutku, v_0 je rychlost přítoku kapaliny, h je velikost iteračního kroku a η je viskozita kapaliny, která při přepočtu do skutečných jednotek odpovídá viskozitě vody při $20^\circ C$.

Během simulace se postupně zvyšuje počet disků M jednak přítokem kapaliny z kohoutku a jednak dělením disků. Dělení se provádí v případě, že nějaký disk se příliš protáhne v souřadnici z a naopak smrští v souřadnici r (protože jeho objem se zachovává). Takový disk je rozdělen na dva tak, aby prostorová derivace rychlosti mezi disky zůstala zachována. Dělení provádíme proto, aby byla zajištěna přesnost simulace i místech, kde se tvar kapky stává postupem simulace komplikovanější.

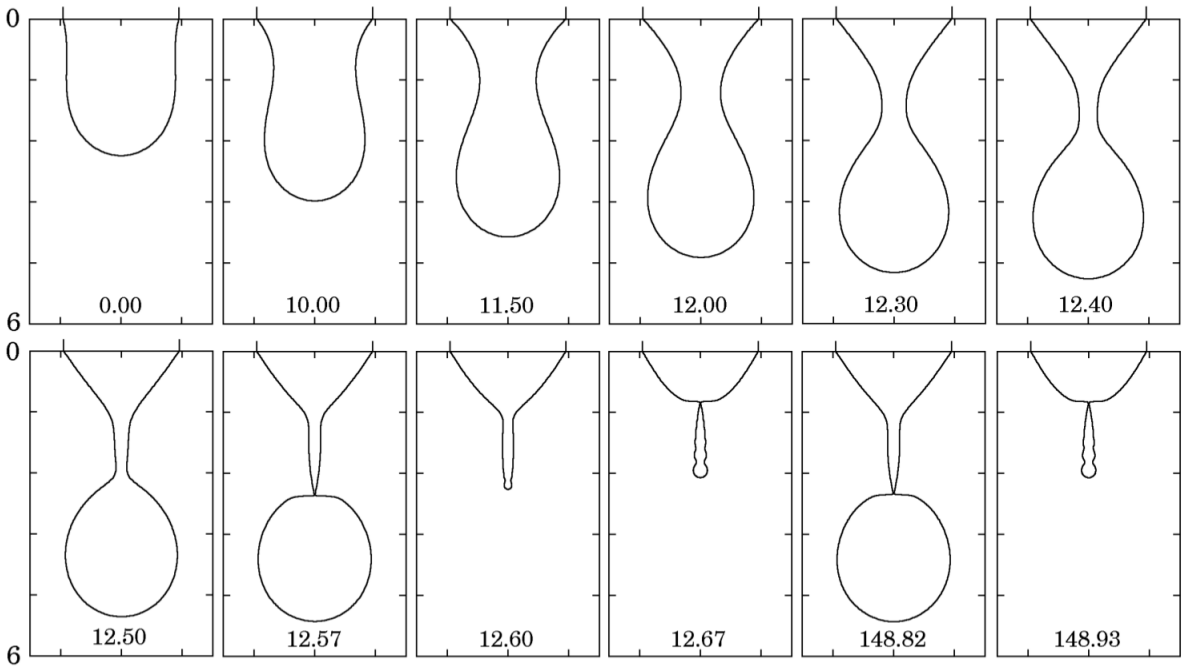
Postupné zvyšování počtu disků s sebou logicky nese větší výpočetní náročnost k provedení numerického řešení. Výpočetní náročnost zároveň narůstá s tím, jak se tvar kapky komplikuje. Následkem toho jsme během simulace narazili na velmi výrazné zvýšení času potřebného k výpočtu jednoho numerického řešení v jednom iteračním kroku. Potřebný čas rostl až do té míry, že z původních maximálně desítek sekund potřebných k výpočtu jednoho kroku tento čas narostl až do řádu několika hodin. Výsledky provedené simulace, které ukazuje graf na obrázku 4.8, jsou z tohoto důvodu zobrazeny pouze do okamžiku $t = 11{,}2$, kdy se kapka začne výrazněji urychlovat a protahovat. Následkem toho se její tvar komplikuje a počet potřebných disků v simulaci roste až řádově. Z tohoto důvodu bylo prakticky nemožné na dostupném hardwaru (pouze běžný stolní počítač) efektivně pokračovat v simulaci.



Obrázek 4.8: Výsledky simulace FDM provedené s parametry $r_a = 0{,}943$; $v_0 = 0{,}01$; $h = 0{,}01$ a $\eta = 0{,}002$. Jednotlivé grafy ukazují tvar formující se kapky v různých časech během simulace.

Přesto se podařilo potvrdit, že soustava diferenciálních rovnic použitá při simulaci správně popisuje chování kapky, což můžeme ověřit srovnáním se simulací provedenou v [4]. Výsledky této srovnávací simulace ukazují grafy na obrázku 4.9, kde pozorujeme velmi podobný vývoj tvaru kapky v závislosti na čase. Parametry obou simulací jsou shodné až na poloměr kohoutku r_a, který v našem případě byl zvolen $r_a = 0{,}943$ a v případě srovnávací simulace $r_a = 0{,}952$. Z tohoto důvodu pozorujeme u srovnávací simulace nepatrně rychlejší formování kapky, protože přítok kapaliny z kohoutku je větší.

Na základě zjištěných poznatků o možnostech našeho modelu budeme jeho vývoj směřovat k lepší optimalizaci výpočtu a snížení jeho časové náročnosti. Již nyní lze navrhnout několik možných přístupů jak optimalizace dosáhnout. Logickým postupem je paralelizace numerických výpočtů, například použitím knihovny *multiprocessing* v jazyce Python. Jako ještě lepší se jeví implementace námi ověřeného modelu v jazyce C++ a paralelizace výpočtu s využitím technologií CUDA nebo OpenCL pro paralelní výpočty na grafické kartě, protože dnešní grafické karty již dosahují výpočetních výkonů v řádech několika tisíc GFLOPS a jsou tak srovnatelné s některými staršími superpočítači.



Obrázek 4.9: Výsledky srovnávací simulace FDM provedené s parametry $r_a = 0{,}952$; $v_0 = 0{,}01$; $h = 0{,}01$ a $\eta = 0{,}002$. Jednotlivé grafy ukazují tvar formující se kapky v různých časech během simulace. Převzato z [4].

4.3 Modifikovaný Mass–spring model (MSMM)

Jednodušším typem modelu kapajícího kohoutku, který se lépe hodí na simulace dlouhodobého vývoje, je *Mass–spring model*. Ten modeluje chování kapky jako zavaží s měnící se hmotností na pružině. Podoba originálního modelu, který navrhl Shaw [15], je určena rovnicemi (4.1). Na jeho základě navrhli K. Kiyono a N. Fuchikami vylepšený model [11] popsaný rovnicemi

$$\begin{aligned} \frac{\mathrm{d}p}{\mathrm{d}t} &= -kz - \gamma\frac{\mathrm{d}z}{\mathrm{d}t} + mg \\ \frac{\mathrm{d}p}{\mathrm{d}t} &= m\frac{\mathrm{d}^2z}{\mathrm{d}t^2} + \left(\frac{\mathrm{d}z}{\mathrm{d}t} - v_0\right)\frac{\mathrm{d}m}{\mathrm{d}t}, \end{aligned} \tag{4.29}$$

který budeme zkráceně označovat MSMM (z angl. *Mass–spring model modified*). Abychom mohli s rovnicemi (4.29) dále pracovat a numericky je řešit, přepíšeme je do tvaru

$$-kz - \gamma\dot{z} + mg = m\ddot{z} + \left(\dot{z} - v_0\right)Q, \tag{4.30}$$

kde Q je hodnota přítoku z kohoutku definovaná rovnicí

$$\frac{\mathrm{d}m}{\mathrm{d}t} = Q = \pi r_a^2 v_0. \tag{4.31}$$

Rovnici (4.30) dále upravíme a vyjádříme ji do podoby funkce $\ddot{z} = f(\dot{z}, z)$. Dostaneme tak výraz

$$\ddot{z} = g - \frac{1}{m}\left[kz + \gamma\dot{z} + (\dot{z} - v_0)Q\right]. \tag{4.32}$$

Abychom mohli provést numerické řešení rovnice (4.32), musíme ji převést na soustavu obyčejných diferenciálních rovnic prvého řádu

$$\begin{aligned} \dot{z} &= v \\ \dot{v} &= g - \frac{1}{m}\left[kz + \gamma v \ + (v \ - v_0)Q\right], \end{aligned} \tag{4.33}$$

kde vystupují parametry k a γ, jejichž hodnoty mají výrazný vliv na chování simulace. Parametr k reprezentuje tuhost uvažované pružiny a γ je tlumící konstanta, která je během simulace konstantní s hodnotou $\gamma = 0{,}05$. Tuhost pružiny k se odvíjí od aktuální hodnoty hmotnosti m. Jejich vzájemný vztah vyjadřuje rovnice

$$k \ = \begin{cases} -11{,}4m + 52{,}5 & (m < 4{,}61) \\ 0 & (m \geq 4{,}61). \end{cases} \tag{4.34}$$

Je vidět, že k je před dosažením kritické hodnoty hmotnosti na hmotnosti závislé a po jejím překročení klesá na nulu. Toto lze velice jednoduše vysvětlit tak, že kapka s vyšší hmotností než kritická přešla k nestabilitě a její hmota pak podléhá volnému pádu. Závislost k před překročením kritické hodnoty je odvozena z experimentů a simulací složitějších modelů [4].

Pro potřeby simulace byl opět vytvořen jednoduchý objektový script v jazyce Python, který je obsahem příloh k této práci. Při výpočtu jsou nejprve nastaveny počáteční podmínky a další parametry. Jedno z možných nastavení, na kterém budeme demonstrovat průběh a výsledky simulace, je $r_a = 0{,}916$; $m_0 = 4{,}61$; $z = z_0 = 2{,}0$; $v = v_0 = 0{,}12$; $g = 1{,}0$, $\gamma = 0{,}05$; $z_c = 5{,}5$; kde r_a reprezentuje poloměr kohoutku, m_0 počáteční hodnotu hmotnosti, kterou zvolíme rovnu kritické hodnotě. Dále z a v vyjadřují souřadnici těžiště kapky a jeho rychlost, v_0 je rychlost přítoku kapaliny z kohoutku, která je hlavním regulovaným parametrem, g je gravitační zrychlení, γ tlumící konstanta a z_c je hodnota, při které dojde k odtržení kapky. Hodnota z_c je zvolena na základě reálných pozorování a složitějších dynamických simulací.

V momentě kdy dojde k odtržení, je nutné snížit hmotnost m. Odtržení neproběhne tak, že by se oddělila veškerá hmotnost, ale část vždy zůstane. Ukazuje se, že doposud nejlépe odpovídá skutečnosti vztah

$$m_r = 0{,}2m + 0{,}3; \tag{4.35}$$

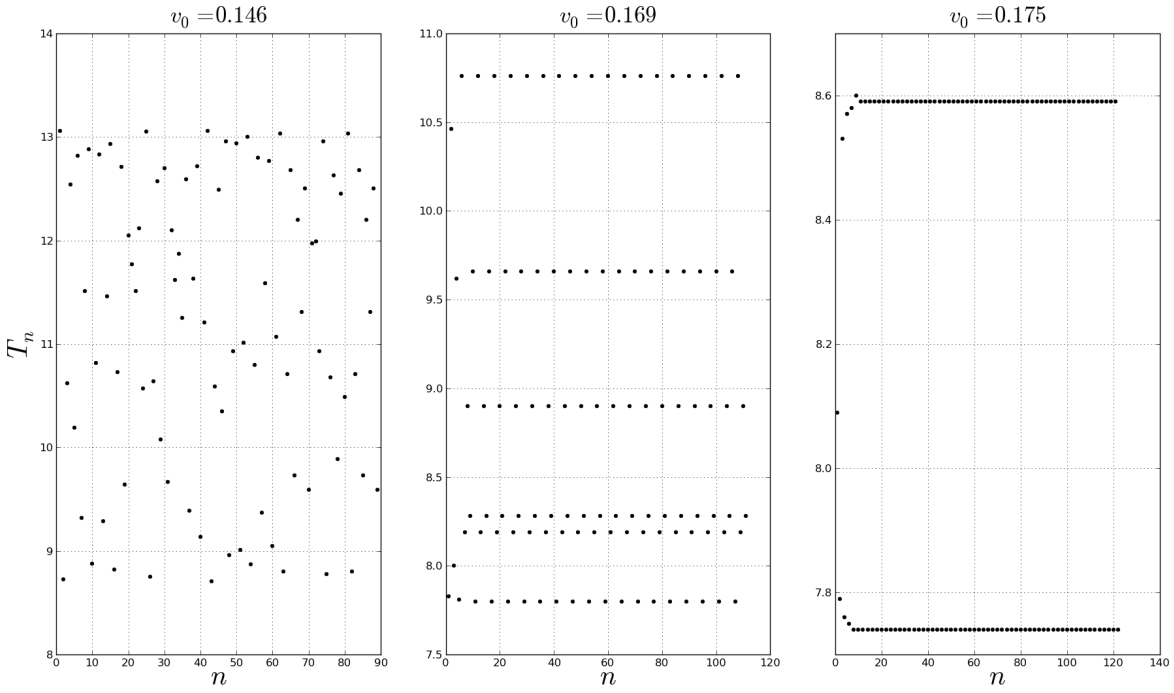
kde m_r vyjadřuje hmotnost, která po odtržení zůstává. Pro rychlost a souřadnici těžiště po odtržení bude platit

$$\begin{aligned} z \ &= z_0 \\ v \ &= 0. \end{aligned} \tag{4.36}$$

Provedeme tedy simulaci pro různé hodnoty v_0 s použitím zmíněných parametrů v rozsahu $t \in \langle 0; 1000 \rangle$ při iteračním kroku $h = 0{,}01$ a dále budeme studovat její výsledky.

4.3.1 Intervaly održení

Při prvním pohledu na výsledky simulací nás bude zajímat, jaké jsou intervaly odtržení kapky, které označíme T_n, a jaká je případná závislost a vliv různých hodnot v_0 na intervaly odtržení. Do grafu vyneseme hodnoty $T_n = t_n - t_{n-1}$, které nám tak udávají časy mezi po sobě jdoucími odtrženími, n označuje pořadí každého odtržení. Jako ukázku použijeme výsledky pro $v_0 = 0{,}146$, $v_0 = 0{,}169$, $v_0 = 0{,}175$. Výsledné závislosti jsou zobrazeny v grafech na obrázku 4.10.

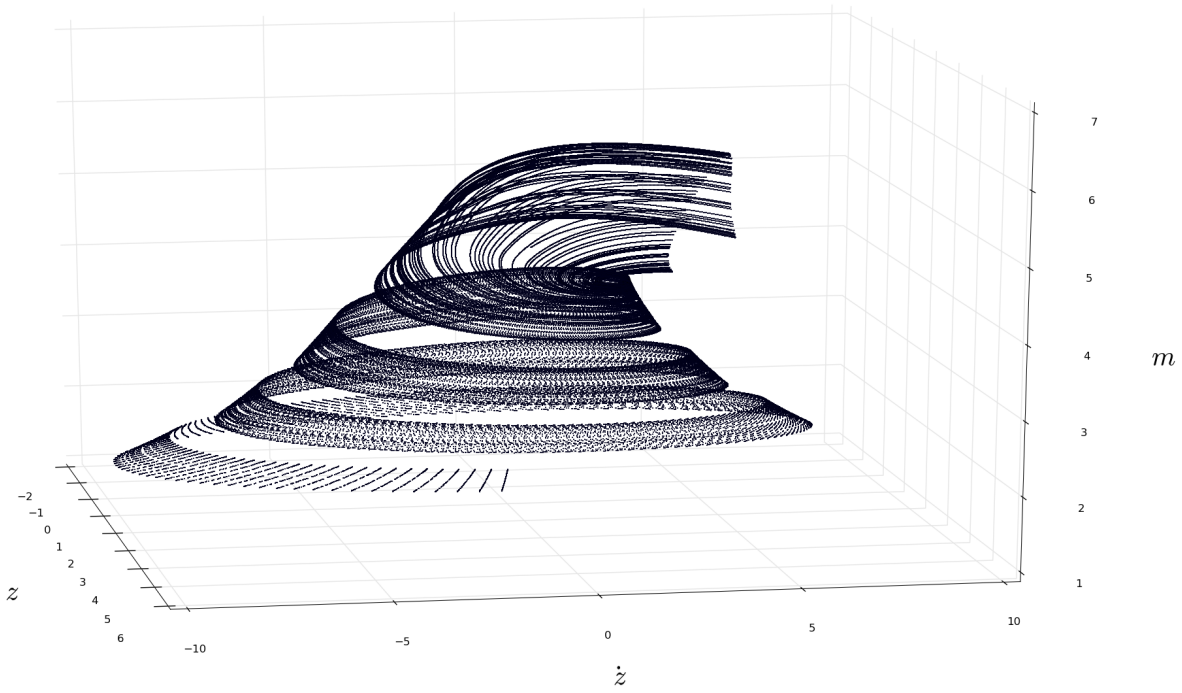


Obrázek 4.10: Srovnání intervalů odtržení pro různé hodnoty v_0.

Je dobře patrné, že zvolená hodnota v_0 má na chování systému velký vliv. Zatímco při $v_0 = 0{,}146$ můžeme pozorovat neperiodické chování s v podstatě rovnoměrně rozloženými hodnotami T_n v určitém intervalu, u $v_0 = 0{,}169$ a $v_0 = 0{,}175$ je vidět chování periodické, kdy jsou hodnoty T_n soustředěny u dvou nebo více hodnot.

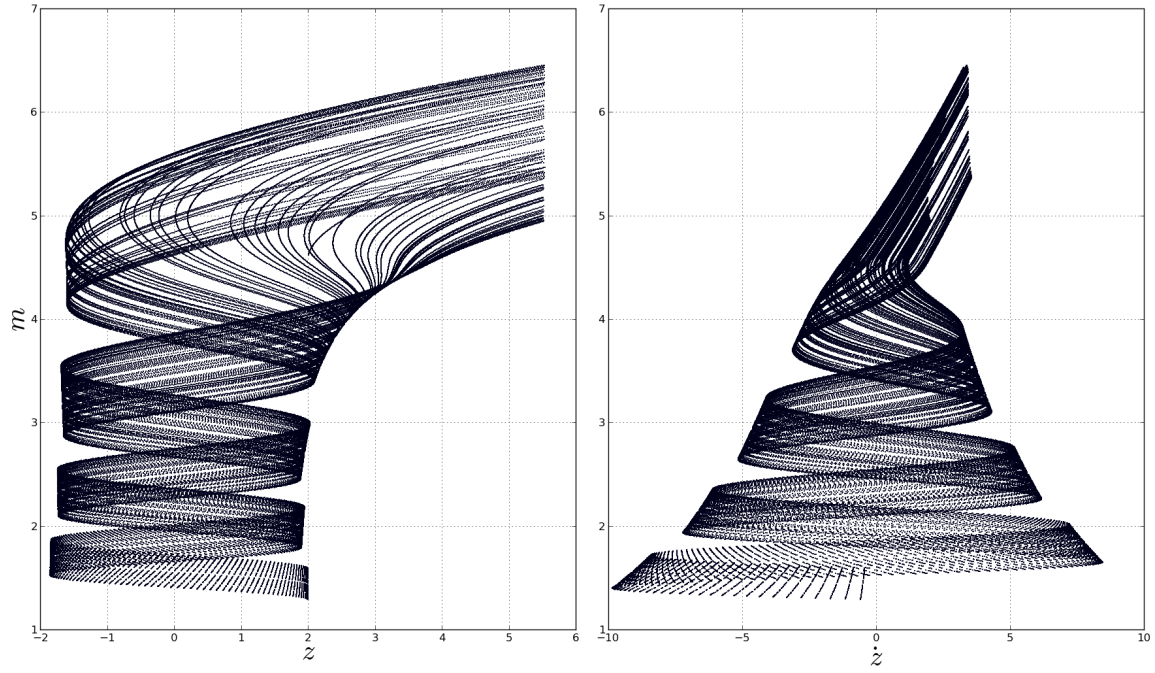
4.3.2 Zpracování stavového portrétu

Stavový portrét zobrazíme ve stavovém prostoru souřadnic z, $\dot{z}$ a m nejprve z výsledků simulace s $v_0 = 0{,}146$, který ukazuje graf na obrázku 4.11. Každá trajektorie v tomto stavovém portrétu představuje jeden cyklus kapky od předchozího odtržení k následujícímu. Trajektorie začínají v grafu dole, kde hmotnost kapky m těsně po odtržení je malá a po zužující se trajektorii spirálovitého tvaru pokračují směrem vzhůru s konstantně rostoucí homotností m. Z grafu je patrné, že těžiště kapky, jehož poloha je dána souřadnicí z, se nepohybuje pouze jedním směrem, ale osciluje. Stejně tak hodnota rychlosti $\dot{z}$ se mění mezi kladnými a zápornými hodnotami.



Obrázek 4.11: Stavový portrét výsledků simulace pro $\nu_0 = 0{,}146$.

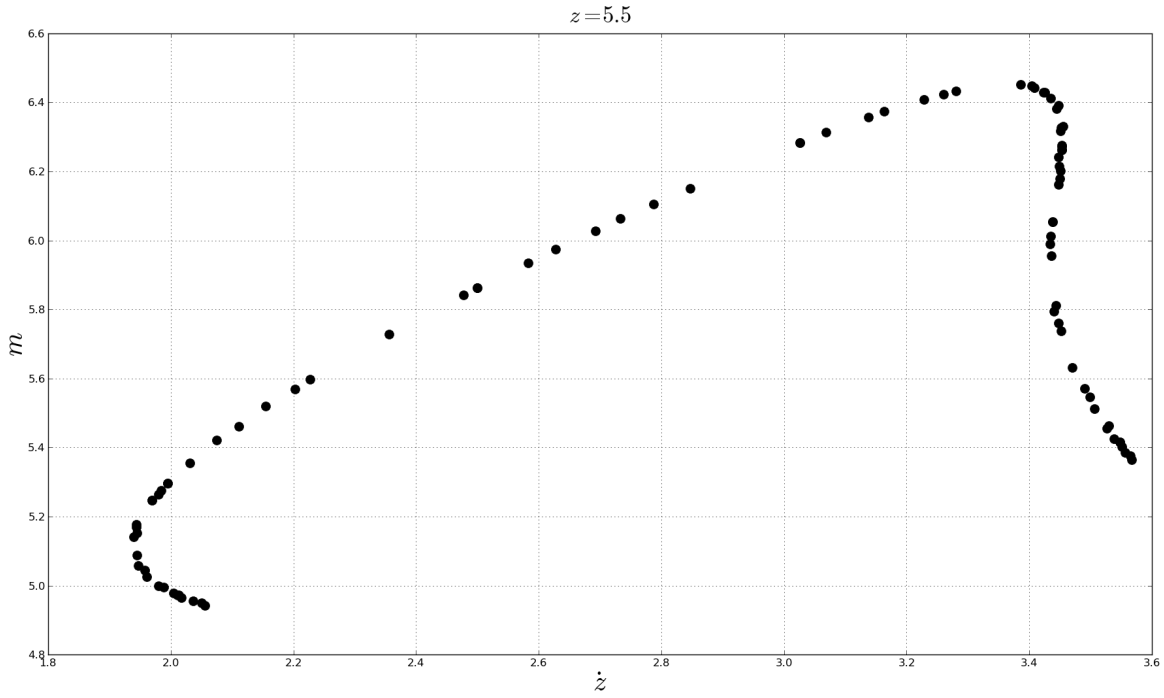
Pro získání lepší představy o povaze pohybu těžiště kapky v souřadnici z, vyneseme do grafu pouze průmět stavového portrétu do roviny $\dot{z} = 0$. Totéž provedeme i pro průmět do roviny $z = 0$. Tato zobrazení ukazuje graf na obrázku 4.12.



Obrázek 4.12: Průměty stavového portrétu do rovin $\dot{z} = 0$ (levý) a $z = 0$ (pravý) z výsledků simulace pro $\nu_0 = 0{,}146$.

Z grafu na obrázku 4.12 jsou oscilace stavových proměnných z a $\dot{z}$ dobře patrné. Je vidět, že z po většinu času simulace osciluje mezi zruba stejnými hodnotami až do okamžiku kdy hmotnost přesáhne $m = 4{,}61$ a tuhost pružiny k klesne na nulu. Kapka v tomto okamžiku začíná padat volným pádem. Oscilace $\dot{z}$ se postupně utlumují až do momentu, kdy $m = 4{,}61$, poté rychlost začíná růst se zrychlením $g = 1$.

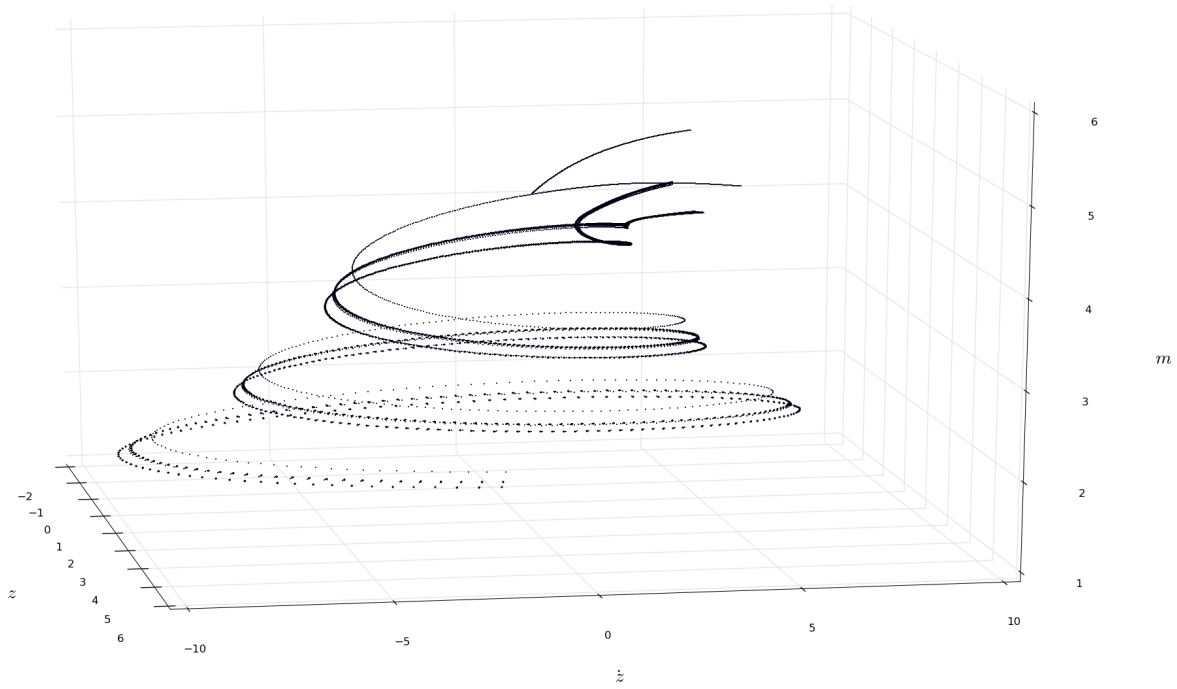
Dále zobrazíme tzv. *Poincarého mapu*, která nám poskytne dobrou představu o atraktoru systému ve vybrané oblasti. Poincarého mapa je průnik trajektorií do zvoleného prostoru s nižší dimenzí. V našem případě nás bude zajímat průnik do roviny $z = 5{,}5$ tedy v okamžiku odtržení. Tuto mapu ukazuje graf na obrázku 4.13.



Obrázek 4.13: Poincarého mapa v momentě odtržení $z = 5{,}5$ pro výsledky simulace $v = 0{,}146$.

Vidíme, že hmotnost jednotlivých kapek v momentě odtržení může být značně rozdílná a stejně tak i jejich rychlost může nabývat různých hodnot. Atraktorem je tedy plocha, jejíž průnik do roviny $z = 0$ tvoří křivku, na které se kumulují body v momentě odtržení. Jak můžeme vidět například z grafu na obrázku 4.11, tato plocha s rostoucí hmotností m značně mění svůj tvar.

Pro srovnání si ještě zobrazíme stavový portrét výsledků simulace s periodickým chováním, například pro $v = 0{,}175$. Tento stavový portrét je zobrazen v grafu na obrázku 4.14.



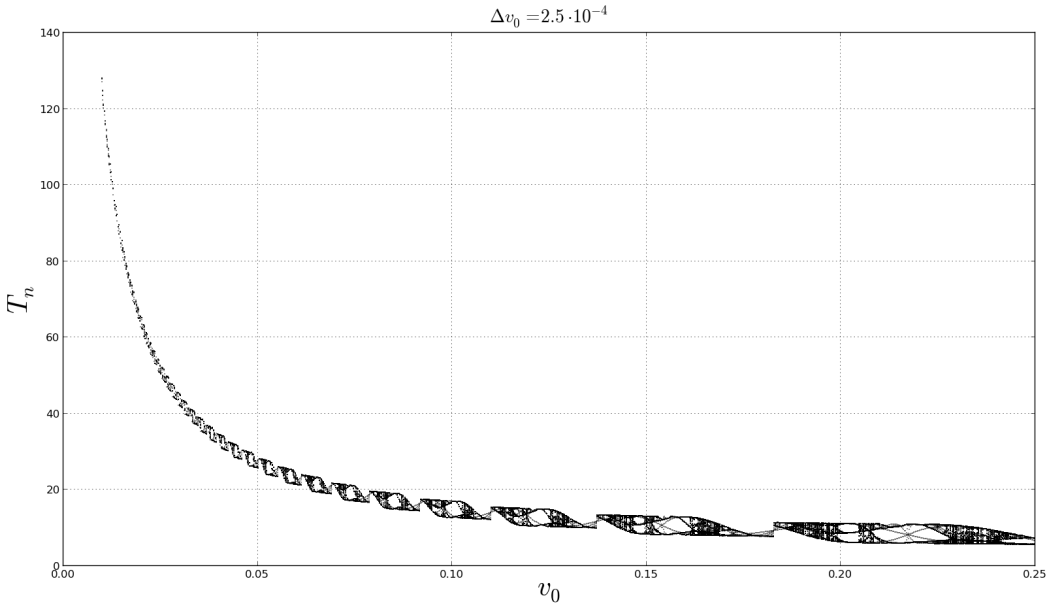
Obrázek 4.14: Stavový portrét výsledků simulace pro $\nu_0 = 0{,}175$.

Vidíme, že jsou trajektorie i zde kumulovány ve stejné oblasti stavového prostoru jako v předchozím případě, ale tentokrát na periodicky se opakujících křivkách limitních cyklů.

4.3.3 Zpracování bifurkačního diagramu

Bifurkačním parametrem v systému kapajícího kohoutku je rychlost přítoku ν_0. Ukazuje se, že právě rychlost přítoku určuje, zda chování kapajícího kohoutku povede v dlouhodobém měřítku k chaosu či nikoli.

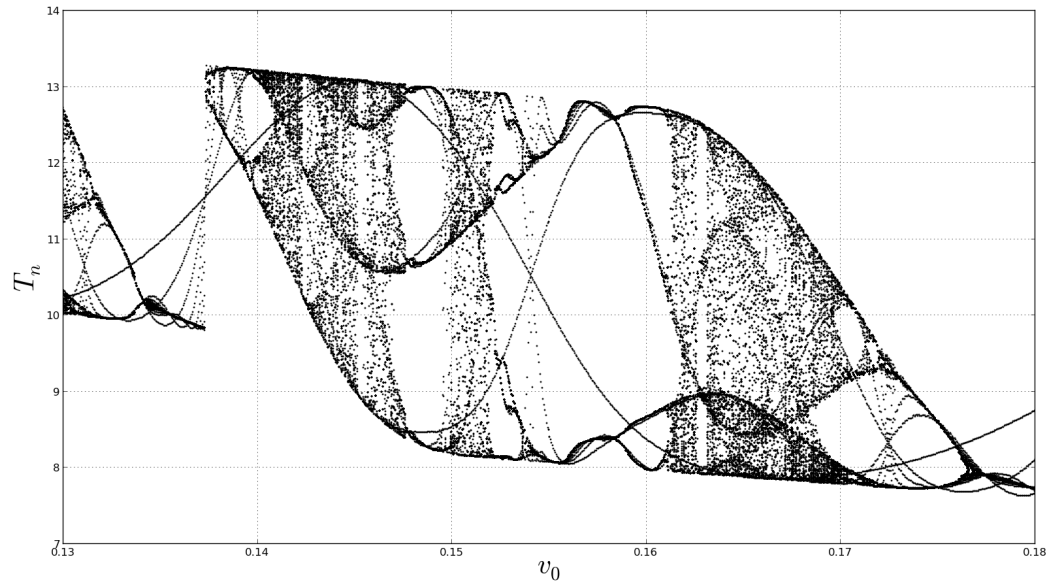
Zpracování bifurkačního diagramu provedeme tak, že ve zvoleném intervalu ν_0 budeme simulovat MSMM vždy pro jednu hodnotu ν_0, a tímto způsobem projdeme celý interval. Takový výpočet je časově velmi náročný, je proto třeba rozumně zvolit úroveň diskretizace intervalu ν_0. Vhodným přístupem bude provést nejprve simulaci na větším intervalu s větším krokem $\Delta\nu_0$. Ze získaných dat sestrojit bifurkační diagram závislosti intervalů odtržení T_n na rychlosti přítoku ν_0. Z grafu na větším intervalu vybrat zajímavé úseky, a ty poté zpracovat ve větším rozlišení. Graf na obrázku 4.15 ukazuje zobrazení v menším rozlišení na intervalu $\nu_0 \in \langle 0{,}01; 0{,}25\rangle$ s krokem $\Delta\nu_0 = 2{,}5 \cdot 10^{-4}$.



Obrázek 4.15: Bifurkační diagram MSMM pro $v_0 \in \langle 0{,}01; 0{,}25 \rangle$ s rozlišením $\Delta v_0 = 2{,}5 \cdot 10^{-4}$.

Z grafu 4.15 je vidět, že v některých oblastech dochází ke zdvojování period T_n a poté naopak půlení period. S rostoucí hodnotou v_0 je tento efekt stále více patrný. Naopak je zřejmé že s klesající rychlostí v_0 se intervaly odtržení limitně blíží k nekonečnu.

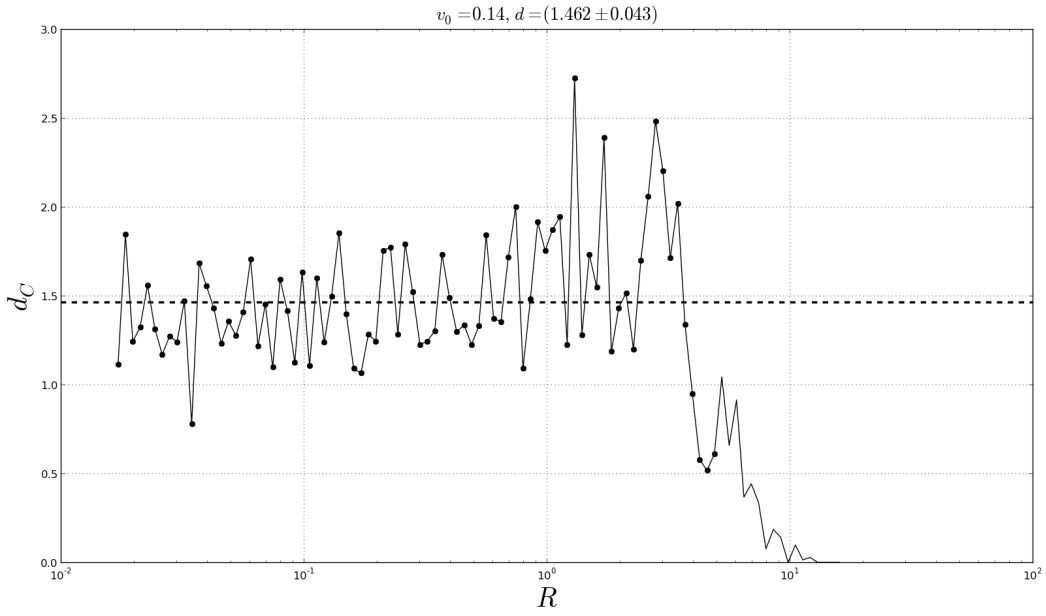
Pro zpracování ve větším rozlišení jsme vybrali interval $v_0 \in \langle 0{,}13; 0{,}18 \rangle$ a výpočet provedli s krokem $\Delta v_0 = 5{,}0 \cdot 10^{-5}$. Bifurkační diagram vybraného regionu ukazuje obrázek 4.16.



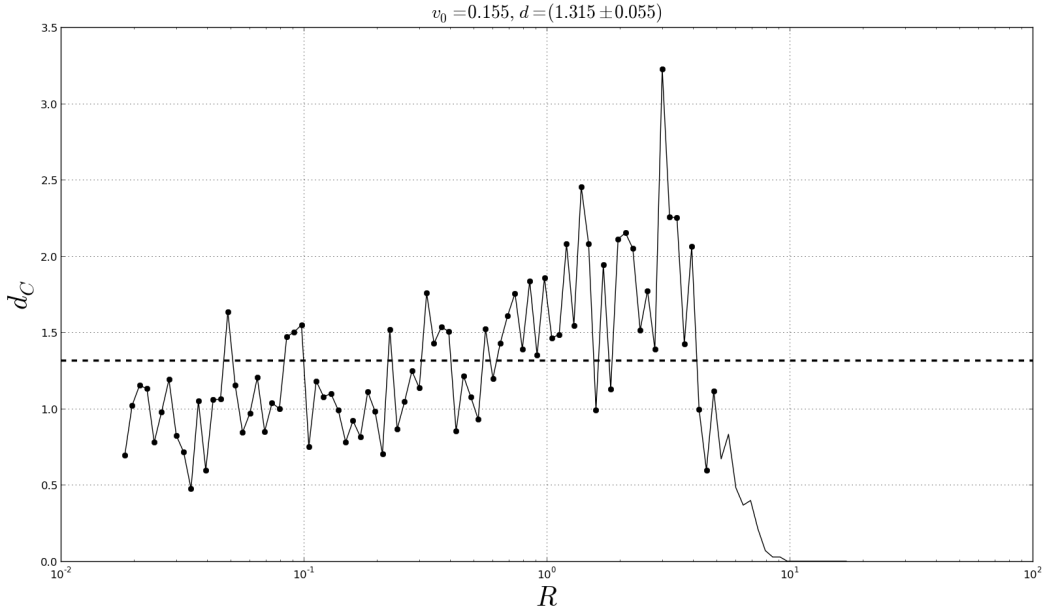
Obrázek 4.16: Bifurkační diagram MSMM v rozsahu $v_0 \in \langle 0{,}13; 0{,}18 \rangle$ s rozlišením $\Delta v_0 = 5{,}0 \cdot 10^{-5}$.

4.3.4 Určení korelační dimenze

Stejným postupem jako v případě Lorenzova atraktoru určíme korelační dimenzi pro výsledky simulací MSMM s několika rozdílnými hodnotami v_0. Obrázky 4.17 a 4.18 ukazují vybrané dva grafy těchto zpracování, konkrétně pro $v_0 = 0{,}140$ a $v_0 = 0{,}155$. Tabulka 4.1 pak obsahuje kompletní výpis určených korelačních dimenzí d i jejich absolutní chyby.



Obrázek 4.17: Určení korelační dimenze pro výsledky simulace s $v_0 = 0{,}140$.



Obrázek 4.18: Určení korelační dimenze pro výsledky simulace s $v_0 = 0{,}155$.

$\mathbf{v_0}$	$\mathbf{d}$	$\delta\mathbf{d}$
0,140	1,462	0,043
0,143	1,409	0,048
0,148	1,407	0,042
0,150	1,774	0,035
0,155	1,315	0,055
0,159	1,474	0,042
0,165	1,737	0,044
0,169	1,384	0,048
0,175	1,203	0,050
0,178	1,068	0,039

Tabulka 4.1: Korelační dimenze pro různé hodnoty v_0.

Když postavíme proti sobě výsledky v tabulce 4.1 a bifurkační diagram v grafu 4.16, vidíme, že existuje zjevná souvislost nižších hodnot korelační dimenze d s nižším počtem limitních cyklů T_n. Jinými slovy, korelační dimenze je nižší pro simulace, kde se objevují stabilní řešení, a naopak vyšší pro řešení vykazující chaotické chování, jako například pro $v_0 = 0{,}150$. V rámci stabilních řešení navíc s rostoucím v_0 pozorujeme pokles hodnot korelační dimenze d.

4.3.5 Určení maximálních Lyapunovových exponentů

K určení maximálních Lyapunovových exponentů pro model MSMM využijeme nástroj *lyap_k* z programového balíku Tisean. Tento nástroj pracuje na základě tzv. *Kantzova algoritmu*, který počítá funkci $S(\Delta n)$ tak, že zjišťuje průměrnou vzdálenost mezi bodem s_{n_0} a body v určitém okolí $R(s_{n_0})$ pro postupně rostoucí diskrétní čas Δn. Následně tyto hodnoty zlogaritmuje a kvůli snížení vlivu fluktuací zprůměruje přes všechny hodnoty s_n [12]. Výsledná funkce má tvar

$$S(\Delta n) = \frac{1}{N}\sum_{n_0=1}^{N} \ln\left(\frac{1}{|\,R(s_{n_0})\,|}\sum_{s_n \in R(s_{n_0})} |\,s_{n_0+\Delta n} - s_{n+\Delta n}\,|\right). \tag{4.37}$$

Do grafu vyneseme závislost $S(\Delta n)$, kde nám směrnice přímky fitované v lineární části závislostí bude udávat maximální Lyapunovův exponent.

Určení maximálních Lyapunovových exponentů provedeme na výsledcích simulace MSMM pro $v_0 = 0{,}146$. Nástroj *lyap_k* z balíku Tisean spustíme příkazem

lyap_k data.dat -M3 -m3 -d8 -s200 -o "lyap.res",

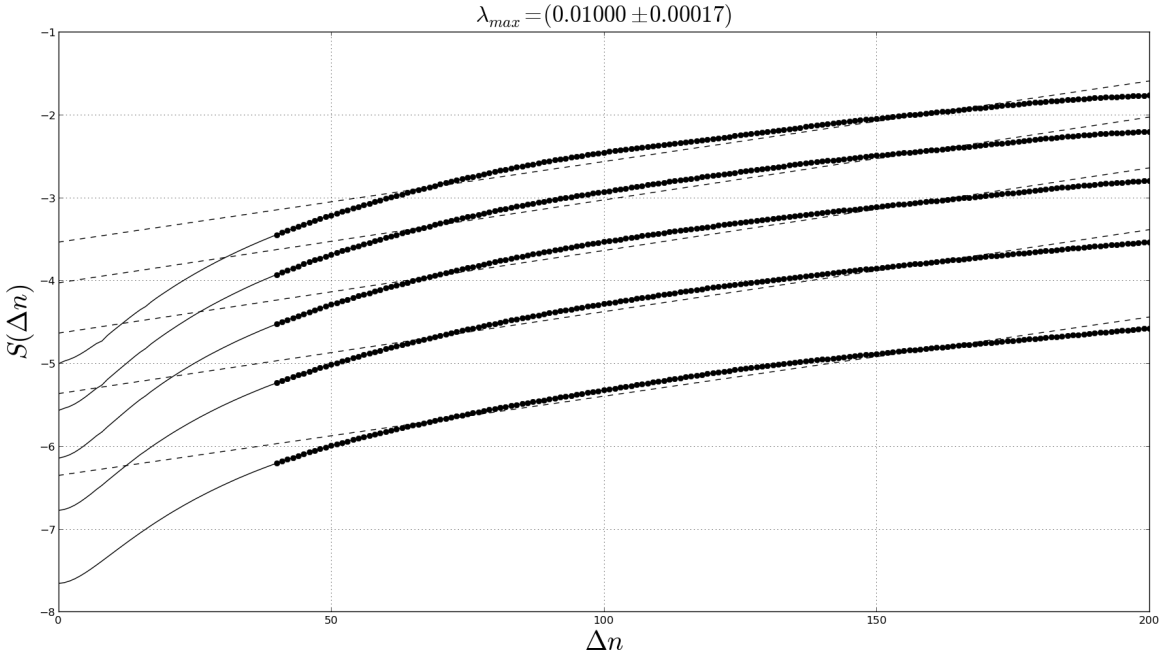
kde důležité jsou parametry -m udávající minimální dimenzi vnoření, -M maximální dimenzi vnoření, -c použitý sloupec ze vstupních dat, -o udává název výstupního souboru s výsledky a -s udává počet iterací Δn. Podrobnější popis tohoto nástroje a všech jeho parametrů je uveden na adrese:

www.mpipks-dresden.mpg.de/~tisean/Tisean_3.0.1/index.html

Určený maximální Lyapunovův exponent λ_{max} je roven

$$\lambda_{\text{max}} = (0{,}01000 \pm 0{,}00017). \tag{4.38}$$

Vidíme, že λ_{max} je kladný, což ukazuje na chaotické chování v systému. Graf na obrázku 4.19 ukazuje zpracované a nafitované závislosti $S(\Delta n)$.



Obrázek 4.19: Určení maximálního Lyapunovova exponentu. Fitovaná lineární oblast je v grafu zvýrazněna body.

5. Model kapajícího disku

"Computer science is no more about computers than astronomy is about telescopes."
Edsger Dijkstra

Model kapajícího disku je možným vysvětlením nepravidelného chování některých systémů s akrečním diskem pozorovaných astronomy.

Jedna z prvních prací zabývajících se touto problematikou byla práce autorů Karla Younga a Jeffrey Scargleho [25]. Autoři zde navrhli tzv. *model odkapávajícího hraničního prstence* (v anglickém originále *dripping handrail*), který je podle anglického názvu označován jako DHR. Jedná se o jednoduchý deterministický model, který uvažuje vnitřní okraj akrečního disku jako prstenec rozdělený na jednotlivé buňky. Tyto buňky jsou vzájemně provázané a proud hmoty buňkami závisí na zvolených parametrech Γ (difuzní koeficient) a ω (akreční koeficient). Tento model je popsán rovnicemi [25]

$$\rho_{n+1}^{i} = [\rho_n^i + \Gamma(\rho_n^{i+1} + \rho^{i-1_n} - 2\rho_n^i) + \omega] \mod 1, \tag{5.1}$$

kde ρ_n^i reprezentuje hustotu hmoty v i–té buňce v čase n. U každé buňky je sledována kritická hodnota hustoty hmoty v buňce. Po překročení této kritické hodnoty dojde k *odkápnutí* hmoty z této buňky a k následnému resetování hustoty v buňce na původní zvolenou hodnotu. Protože hmota z disku odkapává, patří tento model mezi disipativní.

Young a Scargle tento model využili k modelování fluktuací u málo hmotných rentgenových dvojhvězd (LMXBs z angl. *Low Mass X-Ray Binaries*) například Sco X-1. Předpokládali, že změny světelné křivky jsou způsobeny právě odkapáváním zahřátého plynu z vnitřního okraje akrečního disku na centrální těleso. Zatímco difuze *vyhlazuje* distribuci hmoty v buňkách, *odkapávání* hmoty opět zvyšuje nehomogenitu rozložení hmoty.

5.1 Model kapajícího disku z MSMM

V předcházejících kapitolách jsme popsali a realizovali fungující *modifikovaný mass–spring model* kapajícího kohoutku. V mnoha ohledech je tento model podobný jedné uvažované buňce ve výše zmíněném modelu DHR. Funkci limitní kritické hodnoty zastává kritická hodnota z_c, v našem případě nastavená na $z_c = 5{,}5$. Také zde probíhá shromažďování hmoty, kterou kvantifikujeme nikoli pomocí hustoty ρ ale pomocí hmotnosti m. Uvažujeme přítok do kohoutku

Q, který je v případě modelu DHR rozdělen mezi akreci a difuzi. Zaveďme tedy rozšíření MSMM s nekonstantním přítokem Q.

Uvažujme disk složený z N buňek. Každá tato buňka představuje jeden model MSMM popsaný soustavou obyčejných diferenciálních rovnic

$$\begin{aligned} \dot{z}_i &= v_i \\ \dot{v}_i &= g - \frac{1}{m_i}\left[kz_i + \gamma v_i + v_i Q\right], \end{aligned} \tag{5.2}$$

kde $i \in \langle 1, N\rangle$ a člen $v_0 Q$ jsme podle [11] zanedbali. Přítok Q už nebude mít konstantní hodnotu závislou na rychlosti přítoku a poloměru kohoutku, ale bude součtem množství hmoty přidaného procesem akrece a procesem difuze mezi buňkami. Můžeme tedy psát

$$Q = A + D, \tag{5.3}$$

kde A je přítok hmoty akrecí a D je přítok hmoty difuzí. Přítok hmoty akrecí budeme v nejjednodušším případě považovat za konstantní a přítok hmoty difuzí bude závislý na gradientu hmoty mezi jednotlivými buňkami. Celkový přítok bude mít tvar

$$Q = A + D(m_{i-1}, m_i, m_{i+1}). \tag{5.4}$$

Dále tak můžeme rovnici (5.2) přepsat do tvaru

$$\begin{aligned} \dot{z}_i &= v_i \\ \dot{v}_i &= g - \frac{1}{m_i}\left[kz_i + \gamma v_i + v_i A + v_i D(m_{i-1}, m_i, m_{i+1})\right]. \end{aligned} \tag{5.5}$$

Protože jsou buňky uspořádané v prstenci, sousedí s buňkou $i = 1$ buňka $i = N$, takže musíme soustavu (5.6) dále přepsat do tvaru

$$\begin{aligned} \dot{z}_1 &= v_1 \\ \dot{v}_1 &= g - \frac{1}{m_1}\left[kz_1 + \gamma v_1 + v_1 A + v_1 D(m_N, m_1, m_2)\right] \\ &\cdots \\ \dot{z}_i &= v_i \\ \dot{v}_i &= g - \frac{1}{m_i}\left[kz_i + \gamma v_i + v_i A + v_i D(m_{i-1}, m_i, m_{i+1})\right] \\ &\cdots \\ \dot{z}_N &= v_N \\ \dot{v}_N &= g - \frac{1}{m_N}\left[kz_N + \gamma v_N + v_N A + v_N D(m_{N-1}, m_N, m_1)\right]. \end{aligned} \tag{5.6}$$

Přesný tvar funkce popisující difuzi hmoty D zatím určen nebyl, ale jeho volba bude mít velký vliv na chování simulace, protože proces difuze funguje jako vyhlazovací mechanismus rozložení hmoty. Tím pádem bude mít vliv i na intervaly odkapávání hmoty z jednotlivých buňek a v neposlední řadě na výslednou modelovou světelnou křivku. Nalezení přesného tvaru difuzní funkce pro modifikaci MSMM jako odkapávajícího disku bude jedním z cílů našeho dalšího studia v oblasti modelování nelineárních procesů v akrečních discích a porovnání výsledků těchto modelů se skutečnými daty.

6. Závěr

"The drops of rain make a hole in the stone, not by violence, but by oft falling."
Lucretius

V této práci jsme se věnovali popisu různých nelineárních modelů kapajícího kohoutku a jejich možných modifikací na model kapajícího disku, který by mohl pomoci vysvětlit nepravidelné fluktuace objevující se ve světelných křivkách některých astronomických zdrojů. Podařilo se vytvořit sadu dobře použitelných skriptů v jazyce Python, pomocí kterých tyto modely lze simulovat.

Konkrétně se jedná o dynamický kapalinový model(FDM) a modifikovaný mass–spring model(MSMM). Pro potřeby výpočtu vstupních dat modelu FDM, kterými jsou rovnovážné stavy kapek, byl vytvořen skript na jejich řešení. Ten by navíc bylo možné využít i k výpočtům rovnovážných stavů kapalin s jinými parametry než v této práci a tím pádem například k porovnávání s nejrůznějšími experimenty. Samotný model FDM umožňuje provádět simulace s různým stupněm diskretizace systému a nastavovat mnoho parametrů simulace. Dalším plánovaným krokem při vývoji tohoto modelu je paralelizace výpočtu, například využitím knihovny *multiprocesing* pro jazyk Python nebo ještě lépe s použitím technologie CUDA případně OpenCL pro paralelní výpočty na grafické kartě. Tím by se dosáhlo značného zrychlení výpočtu, protože jak se ukázalo výpočetní náročnost simulace velmi výrazně roste s rostoucím počtem disků v simulaci.

Dále byl vytvořen relativně jednoduchý script na řešení modelu MSMM. Analýzou výsledků ze simulací dlouhodobého vývoje MSMM jsme získaly některé charakteristické invarianty systému kapajícího kohoutku popsaného tímto modelem. Byl sestrojen bifurkační diagram na relativně velkém rozsahu rychlostí přítoku v_0 s velmi dobrým rozlišením Δv. Na základě tohoto diagramu je možné dobře sledovat chování modelu kapajícího kohoutku v závislosti na zvoleném v_0 a rozlišovat tak oblasti se stabilními a nestabilními řešeními intervalů odtrhávání kapek T_n. Je vidět, že výsledky simulací se dobře shodují s výsledky uváděnými v literatuře [4][11].

V páté kapitole byl prezentován návrh možné modifikace modelu MSMM do podoby modelu kapajícího disku, který by mohl popsat chování některých akrečních disků. Tento navrhovaný model bude předmětem našeho dalšího studia na téma modelování nelineárních procesů v akrečních discích, kde jednou z hlavních otázek je tvar difuzní funkce D, která určuje proces přenosu hmoty mezi jednotlivými uvažovanými buňkami v tomto modelu.

Některé vybrané scripty jsou vloženy v závěrečné kapitole *Přílohy*. Všechny skripty použité při zpracování této práce jsou navíc k dispozici v elektronické podobě na adrese:

http://physics.muni.cz/~kveton/

7. Přílohy

7.1 Lorenzův atraktor

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import numpy as np
import scipy.integrate as integrate
import matplotlib as mpl
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

class lorenz():
    """
    Reseni Lorenzova atraktoru.

    Autor:
    ------
    Bc. Jiri Kveton
    """
    def __init__(self):
        """
        Inicializace a nastaveni parametru vypoctu.
        """
        self.sigma = 10.0
        self.R = 28.0
        self.b = 8.0 / 3.0

    def diff_system(self, state, t):
        """
        System diferencialnich rovnic Lorenzova atraktoru.
        """
        dxdt = np.zeros_like(state)
        dxdt[0] = self.sigma*(state[1] - state[0])
        dxdt[1] = self.R*state[0] - state[1] - state[0]*state[2]
        dxdt[2] = state[0]*state[1] - self.b*state[2]
        return dxdt

    def solve(self, state, t_start, t_end, dt):
        """
        Provede reseni v zadanem casovem intervalu.
        """
```

```
40          return integrate.odeint(self.diff_system, state, np.arange(t_start, t_end
    , dt))
41
42      def plot(self, data):
43          """
44          Vykresli vyslednou krivku Lorenzova atraktoru.
45          """
46          fig = plt.figure(figsize=(19.2, 10.8))
47          ax = fig.gca(projection='3d')
48          ax.plot(data[:,0], data[:,1], data[:,2], color="black")
49          ax.set_xlabel("$x$", fontsize=25)
50          ax.set_ylabel("$y$", fontsize=25)
51          ax.set_zlabel("$z$", fontsize=25)
52          ax.w_xaxis.set_pane_color((1.0, 1.0, 1.0, 0.0))
53          ax.w_yaxis.set_pane_color((1.0, 1.0, 1.0, 0.0))
54          ax.w_zaxis.set_pane_color((1.0, 1.0, 1.0, 0.0))
55          plt.savefig("lorenzuv_atraktor.png")
56          plt.close(fig)
57
58  if(__name__ == '__main__'):
59      lr = lorenz()
60
61      #nastaveni pocatecnich podminek
62      init_state = np.array([10.0, 10.0, 10.0])
63
64      #reseni Lorenzova atraktoru
65      data = lr.solve(init_state, 0.0, 100.0, 0.001)
66
67      #vykresleni krivky Lorenzova atraktoru
68      lr.plot(data)
```

7.2 Logistická funkce

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import os
5  import numpy as np
6  import scipy as sc
7  import math as mt
8  import matplotlib.pyplot as plt
9
10 class logistic_function():
11     """
12     Zpracovani bifurkacniho diagramu logisticke funkce.
13
14     Autor:
15     ------
16     Bc. Jiri Kveton
17     """
18     def __init__(self):
```

```
        """
        Inicializace a nastaveni parametru vypoctu.
        """
        self.dr = 0.001

    def solve(self, r=2.6, x0=0.5, count=400):
        """
        Reseni zadaneho poctu kroku pro jednu hodnotu r.
        """
        tmp = []
        x = x0
        i = 1
        while(i <= count):
            x = r * x * (1 - x)
            tmp.append([r, x])
            i += 1
        return np.array(tmp)

    def plot_bifurcation(self):
        """
        Vykresleni bifurkacniho diagramu z vypoctenych dat.
        """
        files = os.listdir("data")
        data = np.array([[0, 0]])

        for f in files:
            data = np.concatenate((data, np.genfromtxt("data/"+f, delimiter=",")
    [250:]), axis=0)
        data = np.delete(data, 0, 0)

        fig = plt.figure(figsize=(19.2, 10.8))
        ax = fig.gca()
        ax.plot(data[:,0], data[:,1], marker="o", markersize=0.6, linestyle="")
        ax.set_xlabel("$r$", fontsize=30.0)
        ax.set_ylabel("$x_n$", fontsize=30.0)
        ax.set_xlim(2.7, 4.0)
        ax.set_ylim(-0.05, 1.05)
        ax.grid()
        plt.savefig("logisticka_funkce.png")

if(__name__ == '__main__'):
    lm = logistic_function()

    #reseni v intervalu r
    r = 2.7
    while(r <= 4.0):
        print "r = "+str(r)
        data = lm.solve(r=r)
        np.savetxt("data/data_"+str(r)+".csv", data, delimiter=",")
        r += lm.dr

    #vykreslit bifurkacni diagram
    lm.plot_bifurcation()
```

7.3 Srovnání numerických metod řešení obyčejných diferenciálních rovnic

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import math
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import ode

class ode_compare:
    """
    Srovnani presnosti numerickych metod reseni obycejnych diferencialnich rovnic
    .

    Autor:
    ------
    Bc. Jiri Kveton
    """
    def __init__(self):
        """
        Inicializace a nastaveni parametru vypoctu.
        """
        self.h = 0.1

    def euler(self, x0, y0):
        """
        Eulerova metoda.
        """
        x = x0
        y = y0
        tmp = [[x, y]]
        while(x < 7.9):
            y += self.h * self.func(x, y)
            x += self.h
            tmp.append([x, y])
        return np.array(tmp)

    def rk4(self, x0, y0):
        """
        Runge-Kuttova metoda 4-teho radu.
        """
        x = x0
        y = y0
        tmp = [[x, y]]
        while(x < 7.9):
            k1 = self.func(x, y)
            k2 = self.func(x+0.5*self.h, y+0.5*self.h*k1)
            k3 = self.func(x+0.5*self.h, y+0.5*self.h*k2)
            k4 = self.func(x+self.h, y+self.h*k3)
            y += (1.0 / 6.0) * self.h * (k1 + 2.0 * k2 + 2.0 * k3 + k4)
            x += self.h
```

```
50              tmp.append([x, y])
51          return np.array(tmp)
52
53      def dopri(self, x0, y0):
54          """
55          Dormand-Princova metoda.
56          """
57          x = x0
58          y = y0
59          tmp = [[x, y]]
60          while(x < 7.9):
61              sol = ode(self.func).set_integrator("dopri5", nsteps=10000)
62              sol.set_initial_value(y, x)
63              sol.integrate(x+self.h)
64              y = sol.y
65              x += self.h
66              tmp.append([x, y])
67          return np.array(tmp)
68
69      def func(self, x, y):
70          """
71          Srovnavaci rovnice.
72          """
73          return math.exp(x) + y * math.cos(2.0 * y) + math.sin(x)
74
75      def plot(self, data_1, data_2, data_3):
76          """
77          Graf srovnani vysledku.
78          """
79          fig = plt.figure(figsize=(19.2, 10.8))
80          ax = fig.add_subplot(1,1,1)
81          plt.subplots_adjust(top=0.95, bottom=0.05, hspace=0.08, left=0.05, right
        =0.95)
82          ax.plot(data_1[:, 0], data_1[:, 1], color="blue", linewidth=1.0, marker="
        o")
83          ax.plot(data_2[:, 0], data_2[:, 1], color="red", linewidth=1.0, marker="o
        ")
84          ax.plot(data_3[:, 0], data_3[:, 1], color="green", linewidth=1.0, marker=
        "o")
85          ax.set_xlim([0.0, 8.5])
86          ax.set_ylim([-100.0, 4000.0])
87          ax.set_xlabel("$x$", fontsize=30)
88          ax.set_ylabel("$y$", fontsize=30)
89          ax.legend(["Eulerova metoda", "Runge-Kuttova metoda 4-teho radu", "
        Dormand-Princova metoda"], loc=2, fontsize=25)
90          ax.grid()
91          plt.savefig("ode_srovnani.png")
92          plt.close(fig)
93
94  if(__name__ == '__main__'):
95      comp = ode_compare()
96
97      #reseni s pouzitim ruznych metod
98      data_1 = comp.euler(0.0, 1.0)
```

```
    data_2 = comp.rk4(0.0, 1.0)
    data_3 = comp.dopri(0.0, 1.0)

    #graf srovnani vysledku
    comp.plot(data_1, data_2, data_3)
```

7.4 Roznovážné stavy kapek

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import math
import os
import time
import re
import scipy as sp
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import ode
from sympy import *
from sympy.utilities.lambdify import lambdify

class shapes:
    """
    Reseni rovnovaznych tvaru visicich kapek.

    Autor:
    ------
    Bc. Jiri Kveton
    """
    def __init__(self):
        """
        Inicializace a nastaveni parametru vypoctu.
        """
        #nejvyssi pracovni adresar
        self.path = "."

        #parametry vypoctu
        self.P_range = [0.01, 7.0] # interval P
        self.P_step = 0.01 # krok P
        self.s = sp.linspace(0.0, 8.0, 2E2) # interval s
        self.r0 = 1E-20 # pocatecni polomer
        self.th0 = math.radians(90) # pocatecni uhel tecny s osou z

        #inicializace pracovniho adresare data
        if(self.check_dir(self.path+"/data")):
            self.clear_dir(self.path+"/data")

        #inicializace pracovniho adresare plot
        if(self.check_dir(self.path+"/plot")):
            self.clear_dir(self.path+"/plot")
```

```
44
45      def check_dir(self, dir_path):
46          """
47          Zkontroluje existenci zadaneho adresare, pokud neexistuje tak ho vytvori.
48          """
49          if(os.path.isdir(dir_path) == False):
50              os.mkdir(dir_path)
51              return False
52          else:
53              return True
54
55      def clear_dir(self, dir_path):
56          """
57          Vymaze obsah zadaneho adresare.
58          """
59          os.system("rm -R "+dir_path+"/*")
60
61      def diff_system(self, init_vector, s):
62          """
63          Soustava diff. rovnic popisujici tvar kapky, r(0) = 0, z(0) = Pb tzn.
    tlak na spodu kapky, th(0) = pi / 2 .
64          """
65          r0 = init_vector[0]
66          z0 = init_vector[1]
67          th0 = init_vector[2]
68          return sp.array([math.sin(th0),  (-1.0) * math.cos(th0), (math.cos(th0) /
     r0) - z0])
69
70      def solve(self):
71          """
72          Provede reseni tvaru kapek v zadanem rozsahu.
73          """
74          P = self.P_range[0]
75          n = 1
76
77          #vystupni adresar nenavazanych tvaru
78          self.check_dir(self.path+"/data/shape")
79
80          #reseni tvaru v rozsahu P
81          while(P <= self.P_range[1]):
82              print "shape: P = "+str(P)
83
84              # reseni
85              init_vector = sp.array([self.r0, P, self.th0])
86              data, infodict = sp.integrate.odeint(self.diff_system, init_vector,
    self.s, full_output=True)
87
88              # ulozit data
89              np.savetxt(self.path+"/data/shape/shape_"+self.lead_zero(n, 4)+"_"+
    self.end_zero(P, 5)+".csv", np.insert(data, 0, self.s, axis=1), delimiter=",")
90
91              P += self.P_step
92              n += 1
93
```

```
94      def lead_zero(self, num, n):
95          """
96          Doplni retezec na pozadovanou delku nulami na zacatku.
97          """
98          num = str(num)
99          while(len(num) < n):
100             num = "0" + num
101         return num
102
103     def end_zero(self, num, n):
104         """
105         Doplni retezec na pozadovanou delku nulami na konci.
106         """
107         num = str(num)
108         while(len(num)<n):
109             num = num + "0"
110         return num
111
112     def bound_ceiling(self):
113         """
114         Urci mozne vazby kapek na kohoutek.
115         """
116         import re
117
118         #vystupni adresar navazanych tvaru
119         self.check_dir(self.path+"/data/shape_ceiling")
120
121         #nalezt mozne vazby pro vsechny tvary
122         for file_name in os.listdir(self.path+"/data/shape"):
123             out_name = re.sub("\.csv", "", file_name)
124             out_name = re.sub("original", "ceiling", out_name)
125
126             P = out_name.split("_")[2]
127             data = np.genfromtxt(self.path+"/data/shape/"+file_name, delimiter=',
    ')
128
129             print "bound to ceiling: P = "+str(P)+" file = "+str(file_name)
130
131             tmp = []
132
133             i = 0
134             while(data[i][2] >= 0.0):
135                 tmp.append(data[i])
136                 i += 1
137
138             np.savetxt(self.path+"/data/shape_ceiling/"+out_name+".csv", np.array
    (tmp), delimiter=",")
139
140     def bound_faucet(self, r_faucet=1):
141         """
142         Urci mozne vazby kapek na kohoutek.
143         """
144         import re
145
```

```
146         #vystupni adresar navazanych tvaru
147         self.check_dir(self.path+"/data/shape_faucet")
148 
149         #nalezt mozne vazby pro vsechny tvary
150         for file_name in os.listdir(self.path+"/data/shape"):
151             out_name = re.sub("\.csv", "", file_name)
152             out_name = re.sub("original", "faucet", out_name)
153 
154             P = out_name.split("_")[2]
155             data = np.genfromtxt(self.path+"/data/shape/"+file_name, delimiter=',
    ')
156 
157             print "bound to faucet: P = "+str(P)+" file = "+str(file_name)
158 
159             for line in data:
160                 if(line[1] >= r_faucet):
161                     z1 = line[2]
162                     break
163                 else:
164                     z1 = 0.0
165             for line in data:
166                 if(line[1] <= r_faucet and line[2] < z1):
167                     z2 = line[2]
168                     break
169                 else:
170                     z2 = 0.0
171 
172             data_1 = []
173             data_2 = []
174             data_3 = []
175 
176             for line in data:
177                 if(line[1] <= r_faucet and line[2] >= z1):
178                     data_1.append(line)
179                 elif(line[1] > r_faucet and line[2] <= z1 and line[2] >= z2):
180                     data_2.append(line)
181                 elif(line[1] <= r_faucet and line[2] <= z2 and line[2] >= 0):
182                     data_3.append(line)
183 
184             tmp_1 = np.array(data_1)
185             tmp_1, shift_1 = self.shift_to_zero(tmp_1)
186             np.savetxt(self.path+"/data/shape_faucet/"+out_name+"_1.csv", tmp_1,
    delimiter=",")
187             if(z2 != 0):
188                 tmp_2 = np.array(data_1+data_2)
189                 tmp_3 = np.array(data_1+data_2+data_3)
190 
191                 tmp_2, shift_2 = self.shift_to_zero(tmp_2)
192                 tmp_3, shift_3 = self.shift_to_zero(tmp_3)
193 
194                 np.savetxt(self.path+"/data/shape_faucet/"+out_name+"_2.csv",
    tmp_2, delimiter=",")
195                 np.savetxt(self.path+"/data/shape_faucet/"+out_name+"_3.csv",
    tmp_3, delimiter=",")
```

```
196
197     def shift_to_zero(self, data):
198         """
199         Posune kapku po vazbe k nule
200         """
201         shift = data[len(data) - 1][2]
202         data[:, 2] = data[:, 2] - shift
203         return data, shift
204
205     def volume(self):
206         """
207         Vypocte objemy vsech kapek
208         """
209         import re
210
211         data_out = []
212         for file_name in os.listdir(self.path+"/data/shape_faucet"):
213             P = float(re.sub(r"\.csv", "", file_name).split("_")[2])
214
215             print "volume: P = "+str(P)+" file = "+str(file_name)
216
217             data = np.genfromtxt(self.path+"/data/shape_faucet/"+file_name,
    delimiter=',')
218
219             r = data[:, 1]
220             z = data[:, 2]
221
222             V = 0.0
223             i = len(r)-1
224             while(i>0):
225                 if(z[i] >= 0):
226                     V = V + math.pi * math.pow(r[i], 2) * (z[i-1]-z[i])
227                 i = i-1
228             data_out.append([P, V])
229
230         data_out = np.array(data_out)
231         np.savetxt(self.path+"/data/volume.csv", data_out, delimiter=",")
232
233     def plot_volume(self):
234         """
235         Plot grafu V = f(P)
236         """
237         print "plot: V = f(P)"
238
239         data = np.genfromtxt(self.path+"/data/volume.csv", delimiter=',')
240         fig = plt.figure(figsize=(19.2, 10.8))
241         ax1 = fig.add_subplot(1,1,1)
242         ax1.plot(data[:, 0], data[:, 1], marker="o", markersize=3, linestyle="",
    color="blue")
243         ax1.set_xlabel("$P_{b}$")
244         ax1.set_ylabel("$V$")
245         ax1.grid()
246         plt.savefig(self.path+"/plot/volume_P_faucet.png")
247         plt.close(fig)
```

```
248
249     def critical_volume(self):
250         """
251         Nalezne kapku s kritickou hodnotou objemu a zkopiruje soubor na vstup do
        simulace vyvoje kapky
252         """
253         data = np.genfromtxt(self.path+"/data/volume.csv", delimiter=',')
254         crit = data[np.where(data[:,1] == np.max(data[:,1]))]
255         print crit
256         os.system("cp "+self.path+"/data/shape_faucet/shape_*_"+str(crit[0][0])+"
        _1.csv "+self.path+"/../move/data_in.csv")
257
258 if(__name__ == "__main__"):
259     sh = shapes()
260
261     #provest reseni
262     sh.solve()
263
264     #urcit vazbu na strop
265     sh.bound_ceiling()
266
267     #urcit mozne vazby na kohoutek v zadanem polomeru
268     sh.bound_faucet(r_faucet=0.5)
269
270     #vypocitat objemy kapek
271     sh.volume()
272
273     #vykreslit graf funkce V = f(P)
274     sh.plot_volume()
275
276     #nalezt kapku s kritickym objemem a zkopirovat soubor na vstup do simulace
        vyvoje kapky
277     sh.critical_volume()
```

7.5 Dynamický kapalinový model (FDM)

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  from sympy import *
5  from sympy.utilities.codegen import codegen
6  from sympy.utilities.lambdify import lambdify
7  from scipy.integrate import ode
8  import numpy as np
9  import time
10 import math
11 import os
12
13 class drop_move:
14     """
15     Simulace modelu kapajiciho kohoutu
```

```
16
17      Autor:
18      ------
19      Bc. Jiri Kveton
20      """
21      def __init__(self):
22          """
23          Inicializace a nastaveni parametru vypoctu
24          """
25          # nevyssi pracovni adresar
26          self.path = "."
27
28          # parametry vypoctu
29          self.eta = 0.002
30          self.rho = 1.0
31          self.Gamma = 1.0
32          self.g = 1.0
33          self.v0 = 0.01
34          self.t_interval = [0.0, 1000.0]
35          self.h = 0.01
36
37          #povolit sestaveni soustavy rovnic
38          self.create_system = True
39
40          np.set_printoptions(precision=4, threshold=10, linewidth=200)
41
42      def log(self, par, mode):
43          """
44          Vlozi zaznam do logu
45          """
46          f = open(self.path+"/log.csv", mode)
47          f.write(time.strftime('%Y-%m-%d %H:%M:%S') + "," + str(par["count"]) + ",
    " + str(par["M"]) + "," + str(par["t"]) + "," + par["data_file"] + "\n")
48          f.close()
49
50      def shift_z0(self, data):
51          """
52          Posune kapku a polovinu prvniho disku do zapornych hodnot a nastavi z_0 v
     zapornych
53          """
54          self.shift = data[1][2] * 0.5
55          data[:, 2] = data[:, 2] - self.shift
56
57          #zvysit z_0 10x
58          data[0][2] = 3.0 * data[0][2]
59          self.z0 = data[0][2]
60          return data
61
62      def data_in(self, target=None):
63          """
64          Pripravi vstupni data podle souboru data_in.csv
65          """
66          data = np.genfromtxt(self.path+"/data_in.csv", delimiter=',')
67          data = data[::-1,:]
```

```
68          data = self.shift_z0(data)
69          data = np.delete(data, 0, 1)
70          data = np.delete(data, 2, 1)
71
72          # uchovat polomer kohoutku
73          self.ra = data[0][0]
74
75          tmp = []
76          i = 0
77          while(i < len(data)):
78              if(i == 0):
79                  V = 0.0
80              else:
81
82                  if(i == 1):
83                      V = np.pi * math.pow(data[0][0], 2) * abs(data[0][1]) + np.pi
     * data[i][0] * data[i][0] * data[i][1]
84                  else:
85                      V = np.pi * data[i][0] * data[i][0] * (data[i][1] - data[i
    -1][1])
86
87              tmp.append([0.0, data[i][0], data[i][1], self.v0, V])
88              i += 1
89
90          #ulozit vstupni data
91          np.savetxt(self.path+"/data/data_0.csv", np.array(tmp), delimiter=",")
92          return len(tmp) - 1
93
94      def solve_L(self, data):
95          """
96          Formuluje rovnice pro vsechny disky a sestavi z nich system
97          """
98          M = len(data) - 1
99
100         # vseobecne symboly
101         sym = {}
102         sym["g"] = Symbol("g")
103         sym["rho"] = Symbol("rho")
104         sym["Gamma"] = Symbol("Gamma")
105         sym["eta"] = Symbol("eta")
106         sym["pi"] = Symbol("pi")
107         sym["ra"] = Symbol("ra")
108
109         #charakteristiky disku 0
110         sym["z0"] = Symbol("z0")
111         sym["v0"] = Symbol("v0")
112
113         # charakteristiky disku 1 az M
114         j = 1
115         while(j <= M):
116             sym["r"+str(j)] = Symbol("r"+str(j))
117             sym["z"+str(j)] = Symbol("z"+str(j))
118             sym["v"+str(j)] = Symbol("v"+str(j))
119             sym["V"+str(j)] = Symbol("V"+str(j))
```

```
120             j += 1
121 
122         par = []
123         for key in sym:
124             par.append(key)
125 
126         #celkova kin. energie kapky
127         E_kin = 0
128         j = 1
129         while(j <= M):
130             E_kin += 0.5 * sym["rho"] * sym["V"+str(j)] * (sym["v"+str(j)])**2
131             j += 1
132 
133         #celkova pot. energie kapky
134         E_pot = 0
135         j = 1
136         while(j <= M):
137             E_pot += ( -1.0 * sym["g"] * sym["rho"] * sym["V"+str(j)] * sym["z"+
    str(j)])
138 
139             j += 1
140 
141         #celkova pov. energie kapky
142         S = 0
143         j = 1
144         while(j <= M):
145             if(j == M):
146                 rj = sqrt( sym["V"+str(j)] / (sym["pi"] * (sym["z"+str(j)] - sym[
    "z"+str(j-1)])) )
147                 S += sym["pi"] * rj**2
148             else:
149                 rjp1 = sqrt( sym["V"+str(j+1)] / (sym["pi"] * (sym["z"+str(j+1)]
    - sym["z"+str(j)])) )
150                 if(j == 1):
151                     rj = sqrt( (sym["V"+str(j)] - sym["pi"] * sym["ra"]**2 * Abs(
    sym["z0"]) ) / (sym["pi"] * (sym["z"+str(j)])) )
152                     S += sym["pi"] * (rj + sym["ra"]) * sqrt((rj - sym["ra"])**2
    + sym["z1"]**2)
153                     S += sym["pi"] * 0.5 * (3*rj + rjp1) * sqrt(0.25*(rjp1-rj)**2
     + 0.25*(sym["z2"] - sym["z1"])**2)
154                 else:
155                     rj = sqrt( sym["V"+str(j)] / (sym["pi"] * (sym["z"+str(j)] -
    sym["z"+str(j-1)])) )
156                     S += sym["pi"] * (rj + rjp1) * sqrt( 0.25 * (sym["z"+str(j+1)
    ] - sym["z"+str(j-1)])**2 + (rj - rjp1)**2 )
157 
158             j += 1
159         E_pov = sym["Gamma"] * S
160 
161         #Lagrangian
162         L = E_kin - E_pot - E_pov
163 
164         #disipacni funkce
165         dsp = 0
```

```
166         j = 1
167         while(j <= M):
168             dsp += -3.0 * sym["eta"] * (( sym["v"+str(j)] - sym["v"+str(j-1)] )
    **2 / ( sym["z"+str(j)] - sym["z"+str(j-1)] )**2) * sym["V"+str(j)] * sym["rho
    "]
169             j += 1
170 
171         #pohybove pro vsechny disky z1 az zM
172         self.system = {}
173         j = 0
174         while(j <= M):
175             if(j == 0):
176                 aj = 0.0
177             else:
178                 aj = ( diff(L, sym["z"+str(j)]) + 0.5 * diff(dsp, sym["v"+str(j)
    ]) ) / (sym["V"+str(j)] * sym["rho"])
179             self.system["a"+str(j)] = lambdify(par, aj)
180             j += 1
181         self.create_system = False
182 
183     def solve(self, y0, t, par):
184         """
185         Provede numericke reseni systemu
186         """
187         #reseni
188         sol = ode(self.diff_system).set_integrator("dopri5", nsteps=10000)
189         sol.set_initial_value(y0, t).set_f_params(par)
190         sol.integrate(t+self.h)
191 
192         #pretvarovat vysledek na 2 sloupce
193         i = 0
194         y1 = []
195         while(i < len(sol.y)):
196             y1.append([sol.y[i], sol.y[i+1]])
197             i += 2
198 
199         return np.array(y1)
200 
201     def diff_system(self, t, y0, par):
202         """
203         System diff. rovnic k reseni
204         """
205         system = []
206 
207         i = 0
208         while(i < len(y0) / 2):
209             par["z"+str(i)] = y0[2*i]
210             par["v"+str(i)] = y0[2*i + 1]
211             i += 1
212 
213         j = 0
214         for a in self.system:
215             #dz/dt = v
216             system.append(par["v"+str(j)])
```

```
217
218             #dv/dt = a
219             system.append(self.system["a"+str(j)](**par))
220
221             j += 1
222         return np.array(system)
223
224     def disc_increase(self, data):
225         """
226         Postupne zvysuje pocet disku kapky podle pritoku od kohoutku
227         """
228         tmp = []
229         V_b = data[1][4] - np.pi * math.pow(self.ra, 2) * abs(data[0][2])
230         if(V_b < 0.01):
231             return data
232         else:
233             tmp = []
234             t = data[0][0]
235             z_0 = self.z0
236             z_1 = data[1][2] * 0.3
237             z_2 = data[1][2]
238             r_2 = data[1][1]
239             V_2 = np.pi * r_2 * r_2 * (z_2 - z_1)
240             V_b1 = V_b - V_2
241             V_1 = np.pi * math.pow(data[0][1], 2) * abs(z_0) + V_b1
242             r_1 = math.sqrt(V_b1 / (np.pi * z_1))
243             v_1 = self.v0
244             v_2 = data[1][3]
245             tmp.append([t, self.ra, z_0, self.v0, 0.0])
246             tmp.append([t, r_1, z_1, v_1, V_1])
247             tmp.append([t, r_2, z_2, v_2, V_2])
248             j = 2
249
250             while(j < len(data)):
251                 tmp.append(data[j])
252                 j = j + 1
253
254             self.create_system = True
255             return np.array(tmp)
256
257     def disc_divide(self, data):
258         """
259         Provede rozdeleni prilis roztazenych disku na polovinu
260         """
261         tmp = []
262
263         tmp.append(data[0])
264         tmp.append(data[1])
265         tmp.append(data[2])
266
267         i = 3
268         while(i < len(data)-1):
269             #vzdelenost sousednich bodu
```

```
270             dl = math.sqrt( math.pow(data[i][1]-data[i-1][1], 2) + math.pow(data[
    i][2]-data[i-1][2], 2) )
271
272             #print dl
273
274             if(dl > 0.1):
275                 V_1 = data[i-1][4]
276                 V_2 = data[i][4]
277                 V_3 = data[i+1][4]
278                 r_1 = data[i-1][1]
279                 r_2 = data[i][1]
280                 r_3 = data[i+1][1]
281                 z_1 = data[i-1][2]
282                 z_2 = data[i][2]
283                 z_3 = data[i+1][2]
284                 v_1 = data[i-1][3]
285                 v_2 = data[i][3]
286                 v_3 = data[i+1][3]
287                 r_21 = (r_1 + r_2) / 2.0
288                 r_22 = r_2
289                 z_21 = (z_1 + z_2) / 2.0
290                 z_22 = z_2
291                 dv = (v_3 - v_1) / (z_3 - z_1)
292                 v_21 = dv * (z_21 - z_1) + v_1
293                 v_22 = v_3 - dv * (z_3 - z_22)
294                 V_22 = np.pi * math.pow(r_22, 2) * (z_22 - z_21)
295                 V_21 = V_2 - V_22
296                 tmp.append([data[i][0], r_21, z_21, v_21, V_21])
297                 tmp.append([data[i][0], r_22, z_22, v_22, V_22])
298             else:
299                 tmp.append(data[i])
300             i += 1
301         tmp.append(data[len(data)-1])
302
303         self.create_system = True
304         return np.array(tmp)
305
306     def data_out(self, data, tmp):
307         """
308         Zpracuje vystupni data z reseni do formatu pro ulozeni
309         """
310         data_out = []
311         t = data[0][0] + self.h
312
313         i = 0
314         while(i < len(tmp)):
315             data_out.append([t, data[i][1], tmp[i][0], tmp[i][1], data[i][4]])
316             i += 1
317
318         return data_out
319
320     def solve_init(self, data):
321         """
322         Pripravi sadu vsupnich parametru a pocatecni vektor do reseni
```

```
323         """
324         y0 = []
325         par = {}
326 
327         #pripravit init vektor
328         i = 0
329         while(i < len(data)):
330             y0.append(data[i][2])
331             y0.append(data[i][3])
332             i += 1
333         y0 = np.array(y0)
334 
335         #dalsi parametry pro reseni
336         par["ra"] = self.ra
337         par["pi"] = np.pi
338         par["eta"] = self.eta
339         par["Gamma"] = self.Gamma
340         par["rho"] = self.rho
341         par["g"] = self.g
342         par["z0"] = data[0][2]
343         par["v0"] = self.v0
344 
345         j = 1
346         while(j < len(data)):
347             par["r"+str(j)] = data[j][1]
348             par["V"+str(j)] = data[j][4]
349             j += 1
350 
351         return y0, par
352 
353     def renew_rj(self, data):
354         i = 1
355         while(i < len(data)):
356             if(i == 1):
357                 if(data[i][2] == 0.0):
358                     data[i][1] = self.ra
359                 else:
360                     data[i][1] = math.sqrt( (data[i][4] - np.pi * self.ra * self.
    ra * abs(data[0][2]) ) / (np.pi * data[i][2]))
361             else:
362                 data[i][1] = math.sqrt( data[i][4] / ( np.pi * (data[i][2] - data
    [i-1][2]) ) )
363             i += 1
364 
365         return data
366 
367 if(__name__ == "__main__"):
368     solver = drop_move()
369 
370     #pokud existuje zaznam o simulaci tak zavazat jinak vytovit novou
371     if(os.path.isfile(solver.path+"/log.csv")):
372         log = np.genfromtxt(solver.path+"/log.csv", delimiter=",")
373         i = len(log)-1
374         count = int(log[i][1])
```

```
375         data_0 = np.genfromtxt(solver.path+"/data/data_0.csv", delimiter=",")
376         solver.z0 = data_0[0][2]
377         data = np.genfromtxt(solver.path+"/data/data_"+str(count)+".csv",
    delimiter=",")
378         solver.ra = data[0][1]
379         t = float(data[len(data)-1][0])
380     else:
381         M = solver.data_in()
382         solver.log({"count":0, "M":M, "data_file":"data_0.csv", "t":0.0}, "w")
383         count = 0
384         t = solver.t_interval[0]
385 
386     while(t <= solver.t_interval[1]):
387 
388         #nacist data
389         data = np.genfromtxt(solver.path+"/data/data_"+str(count)+".csv",
    delimiter=',')
390 
391         #sestavit system rovnic (jen pokud je to poprve nebo pokud se pocet
    rovnic zmenil)
392         if(solver.create_system):
393             solver.solve_L(data)
394 
395         #ziskat vstupni parametry do reseni
396         y0, par = solver.solve_init(data)
397 
398         # reseni systemu o jeden krok
399         tmp = solver.solve(y0, t, par)
400         t += solver.h
401         count += 1
402 
403         #zpracovat vystupni data
404         data_out = solver.data_out(data, tmp)
405 
406         #zvysovani poctu disku pritokem z kohoutku
407         data_out = solver.disc_increase(data_out)
408 
409         #deleni prilis roztazenych disku
410         data_out = solver.disc_divide(data_out)
411 
412         #prepocet rj na aktualni hodnoty
413         data_out = solver.renew_rj(data_out)
414 
415         #ulozit data
416         np.savetxt(solver.path+"/data/data_"+str(count)+".csv", np.array(data_out
    ), delimiter=",")
417 
418         #zapis do logu
419         solver.log({"count":count, "M":len(data_out), "data_file":"data_"+str(
    count)+".csv", "t":t}, "a")
420 
421         #informacni hlaska
422         print("t = " + str(t) + ", M = " + str(len(data_out)))
```

7.6 Modifikovaný Mass-spring model (MSMM)

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import math
import scipy as sp
import numpy as np
from scipy.integrate import ode

class mass_spring:
    """
    Mass-Spring model kapajiciho kohoutku.

    Autor:
    ------
    Bc. Jiri Kveton
    """
    def __init__(self):
        """
        Inicializace a nastaveni parametru simulace.
        """
        #parametry modelu
        self.ra = 0.916
        self.m0 = 4.61
        self.z0 = 2.0
        self.g = 1.0
        self.gamma = 0.05
        self.z_crit = 5.5

        #datove containery
        self.drop_data = [] #data o pohybu kapky [t, z, v, m, k]
        self.break_data = [] #data odtrzeni [t, z, v, m, k]

    def get_k(self, m):
        """
        Urceni tuhosti pruziny podle aktualni hmotnosti.
        """
        if(m < 4.61):
            return -11.4 * m + 52.5
        else:
            return 0.0

    def diff_system(self, t, y, p):
        """
        System diferencialnich rovnic Mass-Spring modelu.
        """
        m = p[0]
        k = p[1]
        v = y[1]
        a = self.g - k * y[0] / m - self.gamma * y[1] / m - (np.pi * math.pow(
    self.ra, 2) * self.v0 * (y[1] - self.v0)) / m
        return np.array([v, a])
```

```
51
52      def solve(self, v0=0.1, t_start=0.0, t_end=1000.0, h=0.01):
53          """
54          Provede reseni v urcenem casovem intervalu s predem nastavenymi parametry
        .
55          """
56          self.v0 = v0
57          t = t_start
58          m = self.m0
59          y = np.array([self.z0, self.v0])
60
61          while(t <= t_end):
62              k = self.get_k(m)
63
64              sol = ode(self.diff_system).set_integrator("dop853")
65              sol.set_initial_value(y, t).set_f_params([m, k])
66              sol.integrate(t + h)
67
68              y = sol.y
69              t += h
70              m += np.pi * math.pow(self.ra, 2) * self.v0 * h
71
72              data = [t , y[0], y[1], m, k]
73              self.drop_data.append(data)
74
75              #kontrola odtrzeni --> reset parametru
76              if(y[0] >= self.z_crit):
77
78                  self.break_data.append(data)
79
80                  print data
81
82                  m = 0.2 * m + 0.3
83                  y[0] = 2.0
84                  y[1] = 0.0
85
86          self.drop_data = np.array(self.drop_data)
87          self.break_data = np.array(self.break_data)
88
89          np.savetxt("drop_data_"+str(self.v0)+".csv", self.drop_data, delimiter=",
        ")
90          np.savetxt("break_data_"+str(self.v0)+".csv", self.break_data, delimiter=
        ",")
91
92  if(__name__ == "__main__"):
93      ms = mass_spring()
94
95      #simulace se zadanou rychlosti pritoku v0
96      ms.solve(v0=0.146)
```

Literatura

[1] D. M. Beazley. *Python Essential Reference*. Addison Wesley, fourth edition, 2008.

[2] J. Eggers. Universal pinching of 3d axisymmetric free-surface flow. *Physical Review Letters*, 71(21):3458–3460, 1993.

[3] D. P. Feldman. *Chaos ond Fractals*. Oxford University Press, first edition, 2012.

[4] N. Fuchikami, S. Ishioka, and K. Kiyono. Simulation of a dripping faucet. *Journal of the Physical Society of Japan*, 68(4):1185–1196, 1998.

[5] I. group of developers. Sympy. http://www.sympy.org/, 2007–2014.

[6] R. Hegger, H. Kantz, and T. Schreiber. Practical implementation of nonlinear time series methods: The tisean package. *CHAOS*, 9:413–435, 1999.

[7] R. C. Hilborn. *Chaos and Nonlinear Dynamics*. Oxford University Press, Oxford, second edition, 2000.

[8] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[9] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001–2014.

[10] H. Kantz and T. Schreiber. *Nonlinear Time Series Analysis*. Cambridge University Press, second edition, 2004.

[11] K. Kiyono and N. Fuchikami. Dripping faucet dynamics clarified by an improved mass-spring model. *Journal of the Physical Society of Japan*, 68(10):3259–3270, 1999.

[12] Mgr. Viktor Votruba PhD. Osobní sdělení.

[13] T. Oliphant et al. Numpy. http://www.numpy.org/, 1995–2014.

[14] J. F. Padday and A. R. Pitt. Philosophical transactions of the royal society. *Physical Review Letters*, 275(1253):489–528, 1973.

[15] R. Shaw. *The Dripping Faucet as a Model Chaotic System*. Aerial Press, first edition, 1984.

[16] S. H. Strogatz. *Nonlinear Dynamics and Chaos*. Perseus Books, Reading, Massachusetts, first edition, 1994.

[17] F. team. Ffmpeg. http://www.ffmpeg.org/, 2000–2014.

[18] Wikipedia. Chaos theory — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Chaos_theory&oldid=603135090, 2014. [Online; navštíveno 27.2.2014].

[19] Wikipedia. Double pendulum — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Double_pendulum&oldid=607752407, 2014. [Online; navštíveno 13. 5. 2014].

[20] Wikipedia. Euler method — wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Euler%27s_method, 2014. [Online; navštíveno 12. 4. 2014].

[21] Wikipedia. Runge–kutta methods — wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods, 2014. [Online; navštíveno 12. 4. 2014].

[22] Wikipedia. Stiff equation — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Stiff_equation&oldid=601474336, 2014. [Online; navštíveno 17. 4. 2014].

[23] Wikipedie. Lorenzův atraktor — wikipedie: Otevřená encyklopedie. http://cs.wikipedia.org/w/index.php?title=Lorenz%C5%AFv_atraktor&oldid=10755307, 2013. [Online; navštíveno 27. 2. 2014].

[24] J. A. Yorke et al. *CHAOS: An Introduction to Dynamical Systems*. Springer, New York, third edition, 1996.

[25] K. Young and J. D. Scargle. The dripping handrail model: Transient chaos in accretion systems. *The Astrophysical Journal*, 468:617–632, 1996.