Zápisky k předmětu Využití počítačů ve fyzice

Pavel Stránský

31. března 2020

Obsah

1 Obyčejné diferenciální rovnice

Každou obyčejnou diferenciální rovnici n-tého řádu lineární v nejvyšší derivaci lze převést na soustavu n obyčejných diferenciálních rovnic prvního řádu ve tvaru

$$\frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}t} = \boldsymbol{f}(\boldsymbol{x}, t), \tag{1}$$

kde $\boldsymbol{x} = \boldsymbol{x}(t)$ je vektor hledaných funkcí.

Příklad 1.1: *Pohybovou rovnici*

$$Ma = F(y), \tag{2}$$

kde M je hmotnost pohybujícího se tělesa, $y = y(t)$ jeho poloha a $a = a(t) = \mathrm{d}^2y/\mathrm{d}t^2$ převedeme na dvě diferenciální rovnice prvního řádu triviálním zavedením rychlosti $v = v(t) = \mathrm{d}y/\mathrm{d}t$:

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} y \\ v \end{pmatrix} = \begin{pmatrix} v \\ \frac{1}{M}F(y) \end{pmatrix}, \tag{3}$$

tj. vektor funkce pravých stran podle rovnice (1) je

$$\boldsymbol{f}(\boldsymbol{x}, t) = \begin{pmatrix} v \\ \frac{1}{M}F(y) \end{pmatrix} \tag{4}$$

kde $\boldsymbol{x} \equiv (y, v)$.

Příklad 1.2: *Pohybová rovnice pro harmonický oscilátor (matematické kyvadlo s malou výchylkou) při volbě jednotek $M = \Omega = 1$, kde M je hmotnost kmitající částice a Ω její rychlost, zní*

$$a = \frac{\mathrm{d}^2 y}{\mathrm{d}t^2} = -y \qquad \Longleftrightarrow \qquad \frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} y \\ v \end{pmatrix} = \begin{pmatrix} v \\ t \end{pmatrix} \tag{5}$$

Úkol 1.1: *Převeďte na soustavu obyčejných diferenciálních rovnic prvního řádu rovnici třetího řádu pro Hiemenzův tok*

$$y''' + yy'' - y'^2 + 1 = 0. \tag{6}$$

Řešení 1.1: *Hledaná soustava diferenciálních rovnic je*

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} y \\ v \\ a \end{pmatrix} = \begin{pmatrix} v \\ a \\ -ya + v^2 - 1 \end{pmatrix}. \tag{7}$$

1.1 Diferenciální rovnice prvního řádu

Drtivá většina knihoven a algoritmů pro integraci diferenciálních rovnic počítá s rovnicemi ve tvaru (1). Zde se omezíme na jednu rovnici

$$\frac{\mathrm{d}y}{\mathrm{d}t} = f(y, t), \tag{8}$$

přičemž rozšíření na soustavu je triviální: místo skalárů y a f vezmeme vektory.

Řešení diferenciální rovnice spočívá v nahrazení infinitezimálních přírůstků přírůstky konečnými:

$$\frac{\Delta y}{\Delta t} = \phi(y, t) \tag{9}$$

kde ϕ je funkce, která udává směr, podél kterého se při numerickém řešení vydáme. Volbá této funkce je klíčová a záleží na ní, jak přesné řešení dostaneme a jak rychle ho dostaneme.

1.1.1 Pár důležitých pojmů

- **Jednokrokové algoritmy:** Algoritmy, které výpočtu následujícího kroku hodnoty funkce y_{i+1} vyžadují znalost pouze aktuálního kroku y_i. Rozepsáním (9) dostaneme

$$\boxed{y_{i+1} = y_i + \underbrace{\phi(y_i, t)}_{\phi_i} \Delta t}, \tag{10}$$

  přičemž počáteční hodnota y_0 je dána počáteční podmínkou. My se omezíme pouze na tyto algoritmy.

- **Lokální diskretizační chyba:**

$$\mathcal{L} = y(t + \Delta t) - y(t) - \phi(y(t), t)\Delta t, \tag{11}$$

  kde $y(t)$ udává přesné řešení v čase t.

- **Akumulovaná diskretizační chyba:**

$$\epsilon_i = y_i - y(t_i) \tag{12}$$

- **Řád metody:** Metoda je p-tého řádu, pokud

$$L(\Delta t) = \mathcal{O}(\Delta t^{p+1}). \tag{13}$$

- **Symplektické algoritmy:** Speciální algoritmy navržené pro řešení pohybových diferenciálních rovnic. Od běžných algoritmů je odlišuje to, že zachovávají objem fázového prostoru, a tedy i energii (zatímco u obecných algoritmů se energie s integračním časem mění a většinou roste). V praxi se ze symplektických algoritmů používá pouze Verletův algoritmus 1.3.
- **Kontrola chyby řešení:** Chybu numerického řešení diferenciální rovnice lze zmenšit 1) menším krokem, 2) lepší metodou (metodou vyššího řádu). Menší krok však znamená vyšší výpočetní čas. Sofistikované metody proto průběžně mění velikost kroku: když se funkce mění pomalu, krok prodlouží, když se mění rychle, krok zkrátí (tzv. **metody s adaptivním krokem**). Tím se docílí vysoké přesnosti při co nejmenším výpočetním čase.

1.1.2 Eulerova metoda 1. řádu

$$\phi_i = f(y_i, t_i), \tag{14}$$

tj. krok do y_{i+1} děláme vždy ve směru tečny v bodě y_i.

- Nejjednodušší metoda integrace diferenciálních rovnic.
- Chyba je obrovská, k dosažení přesných hodnot je potřeba velmi malého kroku, což znamená dlouhý výpočetní čas.

1.1.3 Eulerova metoda 2. řádu

$$\begin{aligned} k_1 &= f(y_i, t_i) \\ k_2 &= f\left(y_i + k_1 \Delta t, t + \Delta t\right) \\ \phi_i &= \frac{1}{2}\left(k_1 + k_2\right), \end{aligned} \tag{15}$$

tj. uděláme jednoduchý Eulerův krok ve směru k_1, spočítáme derivaci k_2 po tomto kroku a vyrazíme z bodu y_i ve směru, který je průměrem obou směrů (doporučuji si nakreslit obrázek).

Ekvivalentní je udělat „Eulerův půlkrok“ a vyrazit z bodu y_i ve směru derivace spočtené po tomto půlkroku:

$$\begin{aligned} k_1' &= f(y_i, t_i) \\ k_2' &= f\left(y_i + k_1' \frac{\Delta t}{2}, t + \frac{\Delta t}{2}\right) \\ \phi_i &= k_2' \end{aligned} \tag{16}$$

1.2 Runge-Kuttova metoda 4. řádu

$$\begin{aligned} k_1 &= f(y_i, t_i) \\ k_2 &= f\left(y_i + k_1 \frac{\Delta t}{2}, t + \frac{\Delta t}{2}\right) \\ k_3 &= f\left(y_i + k_2 \frac{\Delta t}{2}, t + \frac{\Delta t}{2}\right) \\ k_4 &= f\left(y_i + k_3 \Delta t, t + \Delta t\right) \\ \phi_i &= \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \end{aligned} \tag{17}$$

- Jedna z nejčastěji používaných metod.
- Vysoká rychlost a přesnost při relativní jednoduchosti.
- Existují i Runge-Kuttovy metody vyššího řádu p, avšak vyžadují výpočet více než p dílčích derivací k_j. Obecně platí, že metoda řádu $p \leq 4$ vyžaduje p derivací, metoda řádu $5 \leq p \leq 7$ vyžaduje $p+1$ derivací a metoda řádu $p = 8, 9$ vyřaduje $p+2$ derivací.

1.3 Verletova metoda

Pro rovnici 2. řádu ve tvaru (pohybovou rovnici)

$$M\frac{\mathrm{d}^2 y}{\mathrm{d}t^2} = F(y), \tag{18}$$

kde M je hmotnost pohybující se částice a F síla, která na ni působí. Algoritmus je

$$\begin{aligned} y_{i+1} &= y_i + v_i \Delta t + \frac{1}{2} a_i \Delta t^2, \\ v_{i+1} &= v_i + \frac{1}{2}\left(a_{i+1} + a_i\right)\Delta t, \end{aligned} \tag{19}$$

kde $a_i \equiv F(y_i)/M$.

- Symplektický algoritmus, tj. algoritmus zachovávající energii (pokud systém popsaný rovnicí (18) energii zachovává).
- Užívá se nejčastěji v molekulární dynamice k simulaci pohybu velkého množství vzájemně interagujících částic.
- Řád této metody je $p = 2$. Symplektické algoritmy s vyšším řádem existují, avšak v praxi se nepoužívají.

Úkol 1.2: *Naprogramujte Eulerovu metodu 1. a 2. řádu*[1]*, Runge-Kuttovu metodu a Verletovu metodu a vyřešte diferenciální rovnici harmonického oscilátoru*

$$\frac{\mathrm{d}^2 y}{\mathrm{d}t^2} = -y \tag{20}$$

s počátečními podmínkami $y_0 = 0$, $y_0' \equiv v_0 = 1$ (analytickým řešením je funkce $\sin t$). Časový krok ponechte jako volný parametr. Nakreslete grafy řešení $y(t)$ a grafy energie $E(t)$ pro rozdílné hodnoty integračních kroků, například $\Delta t = 0.01$ a $\Delta t = 0.1$ pro čas $t \in \langle 0; 30 \rangle$. Energie harmonického oscilátoru je dána vzorcem

$$E = \frac{1}{2}\left(y^2 + v^2\right). \tag{21}$$

Přesvědčte se, že jediná Verletova metoda skutečně zachovává energii. Pro ostatní metody energie roste.

Řešení 1.2: *Jedno možné řešení je rozděleno do dvou souborů* `ODE.py` *a* `Oscillator.py`*, které můžete stáhnout z GitHubu* `https://www.github.com/PavelStransky/PCInPhysics` *(o GitHubu bude pojednáno v sekci 2.1).*

- `ODE.py`*: Modul napsaný dostatečně obecně, aby ho bylo možné použít na řešení libovolných soustav diferenciálních rovnic.*
  - `StepEuler1`*: Integrační krok Eulerovy metody 1. řádu.*

[1] Tyto metody jsme již naprogramovali na cvičení minulý týden.

  - `StepEuler2`: *Integrační krok Eulerovy metody 2. řádu.*
  - `StepVerlet`: *Integrační krok Verletovy metody.*
  - `StepRungeKutta`: *Integrační krok Runge-Kuttovy metody 4. řádu.*
  - `ODESolution`: *Integruje diferenciální rovnici danou pravými stranami prvního parametru* `Derivatives` *pomocí kroku (metody) dané parametrem* `Step` *a délkou* `dt`*, přičemž vrátí pole hodnot řešení soustavy rovnic v jednotlivých časech, pole časů a jméno integračního kroku (pro snazší pozdější porovnání různých metod).*
  - `ScipyODESolution`: *Integruje diferenciální rovnici pomocí funkce* `odeint` *z knihovny* `scipy.integrate`*. Pozor, parametr* `dt` *zde neznamená integrační krok, nýbrž časový krok výsledného pole. Funkce* `odeint` *používá sofistikovaný řešitel diferenciálních rovnic s proměnným krokem. Pro podrobnosti můžete mrknout na dokumentaci k této funkci.*
  - `ShowGraphSolutions`: *Vykreslí graf řešení diferenciální rovnice (jako první parametr* `odeSolutions` *lze zadat seznam více řešení různými metodami či s různým krokem). Parametr* `ExactFunction` *je odkaz na přesné řešení dané diferenciální rovnice. Je-li specifikován, vykreslí se do grafu dva panely: jeden s hodnotami numerického řešení, druhý s rozdílem řešení numerického a přesného.*

  Funkce `StepEuler1`*,* `StepEuler2` *a* `StepRungeKutta` *fungují pro obecnou soustavu n obyčejných diferenciálních rovnic prvního řádu. Funkce* `StepVerlet` *funguje jen pro pohybovou rovnici, tj. pro jednu diferencální rovnici původně druhého řádu přepsanou na dvě diferenciální rovnice prvního řádu.*

- **`Oscillator.py`**: *Soubor, který využívá obecných funkcí z modulu* **`ODE.py`** *pro integraci harmonického oscilátoru různými metodami.*
  - `Energy`: *Pro zadanou polohu a rychlost vrátí energii harmonického oscilátoru.*
  - `Derivatives`: *Pravá strana soustavy diferenciálních rovnic harmonického oscilátoru.*
  - `CompareMethods`: *Vyřeší diferenciální rovnici harmonického oscilátoru různými metodami a řešení nakreslí do jednoho grafu. Následně vykreslí grafy $E(t)$. Harmonický oscilátor je konzervativní systém (zachovává energii), rostoucí energie je způsobena nepřesností integrační metody.*

Příslušné grafy jsou zobrazeny v obrázku 1.

Úkol 1.3: *Rozšiřte kód tak, aby počítal průměrnou kumulovanou chybu*

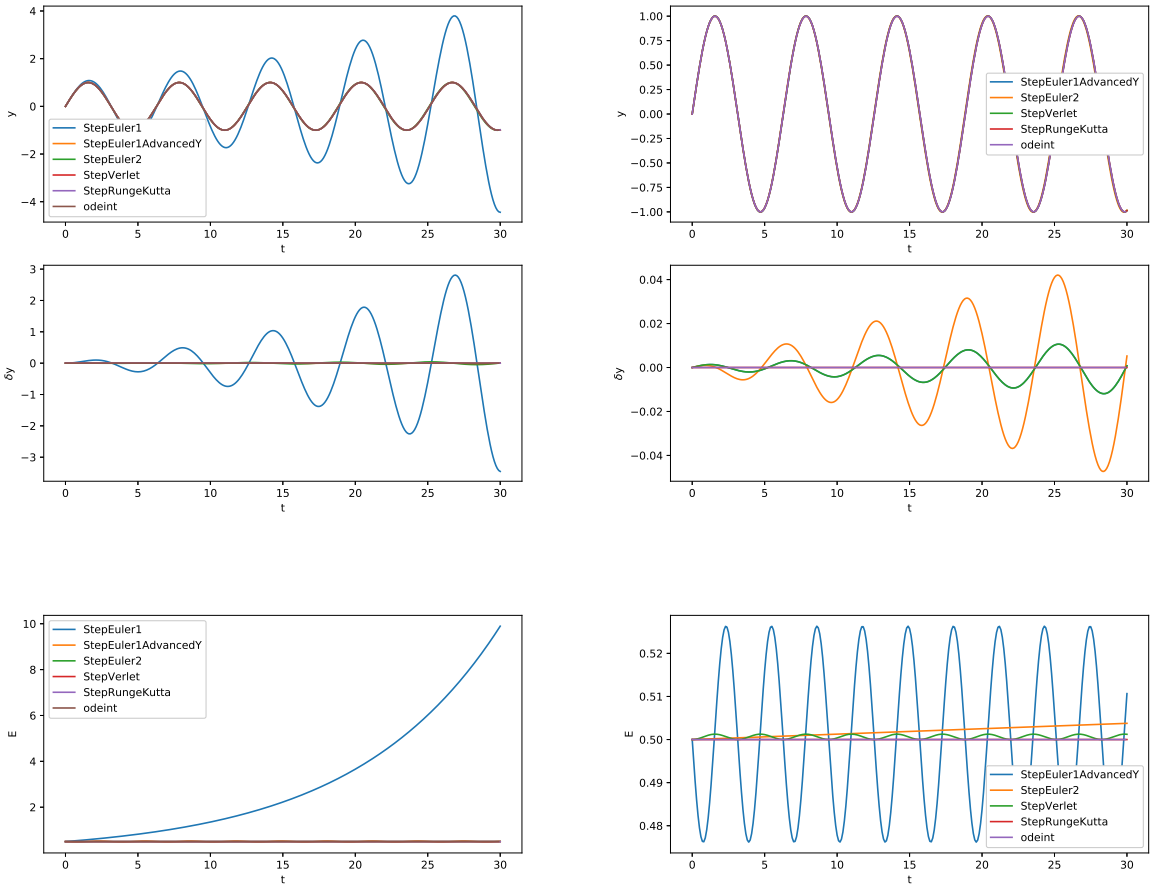
$$\mathcal{E} = \sqrt{\frac{1}{n}\sum_{i=0}^{n-1}(y_i - \sin t_i)^2} \tag{22}$$

a nakreslete závislost $E(\Delta t)$ pro $\Delta t \in \langle 0.002; 0.1\rangle$ a pro různé metody. Jelikož očekáváme mocninnou závislost dle (13), kde exponent je tím větší, čím větší je řád metody, je výhodné graf $E(\Delta t)$ kreslit v log-log měřítku. V Pythonu použijete místo `plot(...)` *funkci* `loglog(...)` *z knihovny* `matplotlib.pyplot`*.*

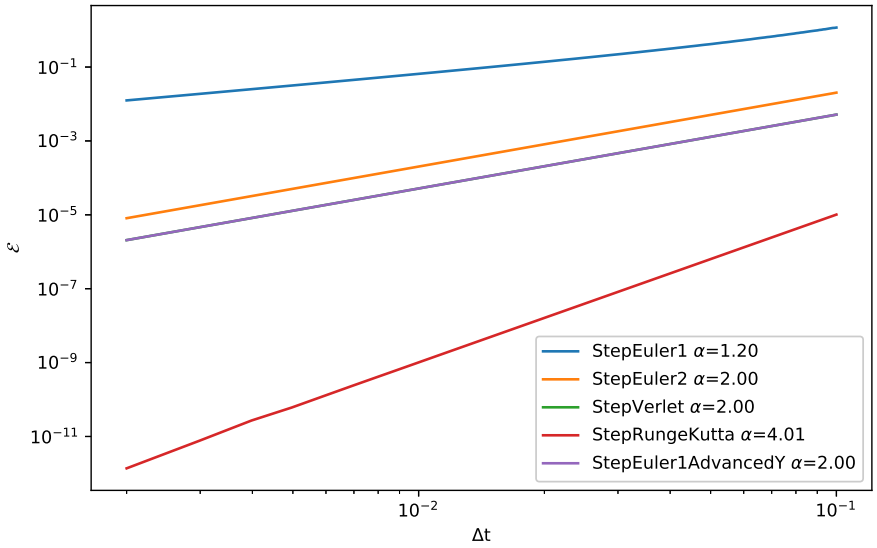
Řešení 1.3: *Průměrnou kumulovanou chybu počítá funkce* `CumulativeError` *z modulu* **`ODE.py`***. Srovnání řešení diferenciální rovnice harmonického oscilátoru různými integračními metodami zakresluje do grafu funkce* `ShowGraphCumulativeErrors` *ze souboru* **`Oscillator.py`***. Výsledný graf je znázorněn na obrázku 2. Vypočítanými křivkami $\ln\mathcal{E}(\ln\Delta t)$ je proložena přímka, jejíž sklon α udává exponent moncinného zákona*

$$\mathcal{E} \propto (\Delta t)^{\alpha} \tag{23}$$

(k proložení přímky je využita funkce pro lineární regresi `linregress` *z knihovny* `scipy.stats`*). Tento exponent výborně odpovídá řádu metody p dle rovnice (13).*



Obrázek 1: Integrace diferenciální rovnice harmonického oscilátoru (20) různými metodami. Časový krok je $\Delta t = 0.1$. *Levý sloupec:* všechny metody. *Pravý sloupec:* bez Eulerovy metody 1. řádu. *1. řádek:* hodnoty $y(t)$. *2. řádek:* rozdíly $\delta y(t) = y(t) - \sin t$. Pro Eulerovu metodu 1. řádu je divergence numerického od analytického řešení očividně exponenciální v čase. *3. řádek:* energie (21). Pro Eulerovy metody energie roste. Energie se mění i pro Runge-Kuttovu metodu (pro tento systém energie s časem klesá) a pro integraci pomocí funkce `odeint`, avšak tyto změny jsou řádově menší, a tudíž je není na grafech při daném měřítku svislé osy vidět. Naopak pro Verletův algoritmus a pro „předbíhající“ Eulerovu metodu energie osciluje okolo počáteční energie $E = \frac{1}{2}$.



Obrázek 2: Závislost průměrné kumulované chyby (22) na délce kroku Δt vypočítaná a vykreslená pomocí funkce `ShowGraphCumulativeErrors` pro harmonický oscilátor (soubor **`Oscillator.py`**). Křivka pro Verletovu metodu je „schovaná“ za křivkou pro předbíhající Eulerovu metodu.

Úkol 1.4: *Eulerovu metodu 1. řádu lze pro harmonický oscilátor vylepšit následující záměnou:*

$$\begin{aligned} y_{i+1} &= y_i + v_i \Delta t \\ v_{i+1} &= v_i - y_i \Delta t \end{aligned} \quad \longrightarrow \quad \begin{aligned} y_{i+1} &= y_i + v_i \Delta t \\ v_{i+1} &= v_i - y_{i+1} \Delta t \end{aligned} \tag{24}$$

(vypočítáme y_{i+1} a tuto hodnotu použijeme namísto hodnoty y_i pro výpočet rychlosti v_{i+1}). Naprogramujte tuto metodu u ukažte, že pro harmonický oscilátor se jedná o metodu 2. řádu. Využijte srovnání v grafu z předchozí úlohy.

Řešení 1.4: *Tato metoda je naimplementována v modulu* **`ODE.py`** *funkcemi* `StepEuler1AdvancedY` *a* `StepEuler1AdvancedV`. *Ze srovnání s ostatními metodami zobrazené v obrázcích 1 a 2 vyplývá, že tato metoda je*

- *symplektická (energie sice osciluje a osciluje s větší amplitudou než pro Verletovu metodu, ale pořád osciluje okolo počáteční hodnoty),*
- *2. řádu.*

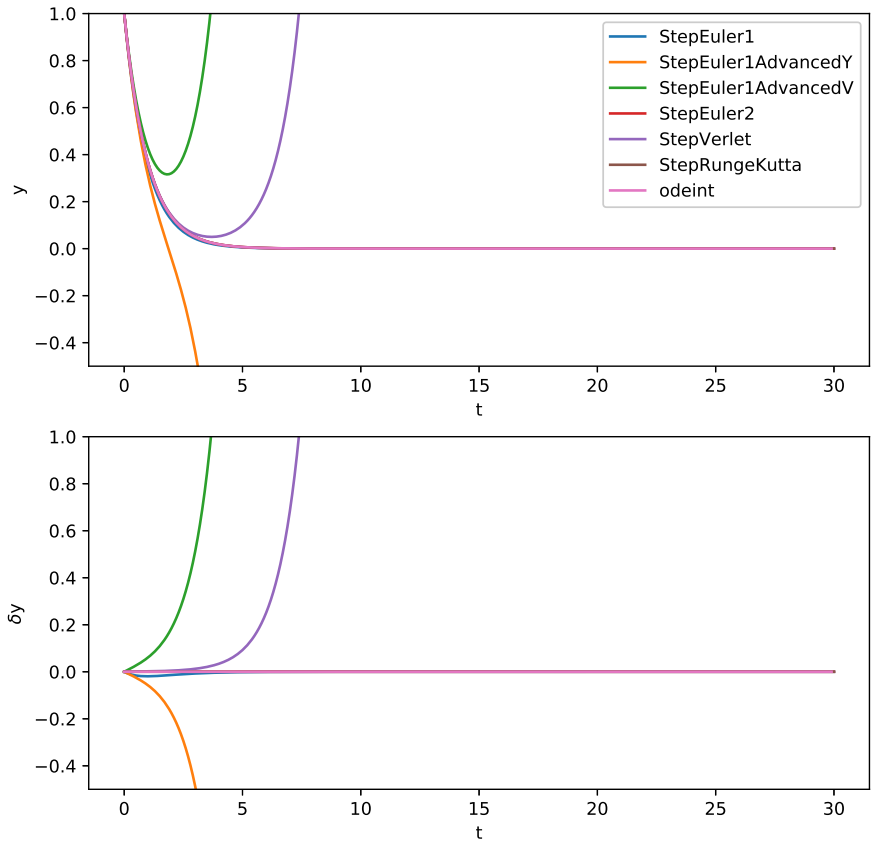
V obrázcích jsou výsledky pouze pro metodu předbíhající v souřadnici. Díky symetrii jsou výsledky pro metodu předbíhající v rychlosti identické.

Úkol 1.5: *Využijte hotové kódy a pohrajte si s řešením rovnice pro klesající exponenciálu*

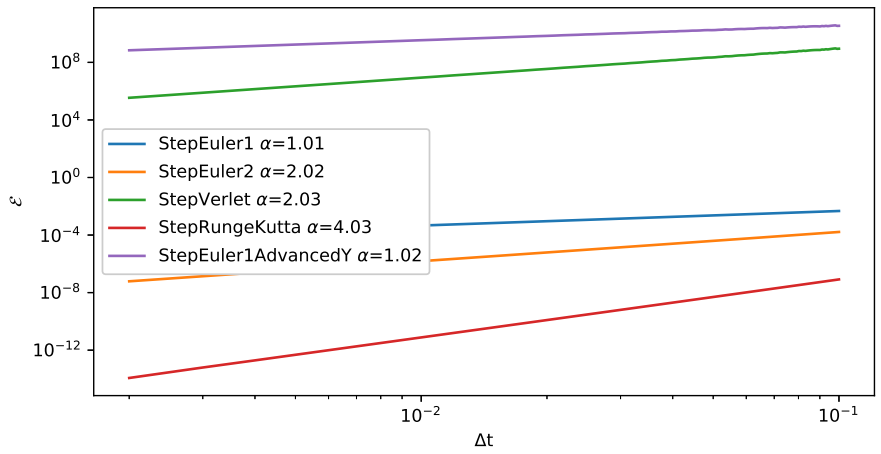
$$\frac{\mathrm{d}^2 y}{\mathrm{d}t^2} = y \tag{25}$$

s počátečními podmínkami $y_0 = 1$, $y'_0 = -1$. Přesvědčte se, že Verletova metoda a vylepšená Eulerova metoda z posledního bodu jsou nestabilní — pro tuto rovnici v relativně krátkém čase začnou řešení exponenciálně divergovat.

Řešení 1.5: *Řešení analogické k příkladu harmonického oscilátoru je v souboru* **`Exp.py`**. *Tento systém není konzervativní — nelze nadefinovat zachovávající se veličinu, která by měla význam energie.*



Obrázek 3: Totéž jako v obrázku 1, avšak pro exponenciálně klesající systém daný rovnicí (25). Symplektické algoritmy jsou nestabilní.



Obrázek 4: Totéž jako v obrázku 2, avšak pro exponenciálně klesající systém daný rovnicí (25). Pro symplektické algoritmy uvedený graf nedává příliš smysl, jelikož chyba je o řády vyšší než hledané řešení. Přesto stojí za povšimnutí, že předbíhající Eulerův algoritmus, který výborně funguje pro harmonický oscilátor a je pro něj metodou 2. řádu, se zde chová jako metoda 1. řádu.

Z obrázku 3 je vidět, že symplektické algoritmy jsou zde nestabilní, a tudíž nejsou na tento typ úlohy vhodné, což je pochopitelné, protože symplektické algoritmy jsou navrženy pouze pro energii zachovávající systémy. Obecné řešení rovnice (25) má tvar

$$y(t) = A\,\mathrm{e}^{t} + B\,\mathrm{e}^{-t}, \tag{26}$$

přičemž my speciálními počátečními podmínkami vybíráme pouze exponenciálně klesající řešení. Symplektické algoritmy v určitou chvíli „překmitnou“ na exponenciálně rostoucí řešení a začnou divergovat.

Průměrná kumulovaná chyba je znázorněna na obrázku 4.

1.4 Shrnutí

- Řešitelé obyčejných diferenciálních rovnic převážně pracují se soustavami diferenciálních rovnic prvního řádu. Na tento tvar není obtížné diferenciální rovnici vyššího řádu převést.
- Nejčastěji se používají jednokrokové metody, jejichž hlavní výhoda je v možnosti jednoduše měnit délku kroku.
- Přesnost řešení závisí na řádu metody p a na délce integračního kroku Δt. Čím je řád metody vyšší, tím rychleji klesá chyba se zmenšujícím se krokem. V praxi, pokud nechcete svěřit svůj probém černé skříňce ve formě nějaké hotové knihovny, se velmi často používá Runge-Kuttova metoda 4. řádu, která je jednoduchá na implementaci, je stabilní a rychlá.
- Symplektické metody, z nichž nejběžnější je Verletova metoda, jsou výhodné k modelování fyzikálních systémů zachovávajících energii. Pro nekonzervativní systémy nejsou vhodné.
- V Pythonu se procvičilo:
  - předávání odkazů na funkce v argumentu,
  - vracení více hodnot z funkce a jejich následné zpracování,
  - vykreslování grafů a práce s více panely pomocí funkce `subplot`; argument této funkce přijímá trojciferné číslo, ve kterém první cifra udává počet panelů v řádcích, druhá cifra počet panelů ve sloupcích a třetí cifra pořadové číslo vykreslovaného panelu,
  - funkce `lineregression` z knihovny `scipy.stats` pro výpočet lineární regrese.

A nyní již umíte vypočítat a nakreslit průběh funkce sinus :-)

2 Git (pokračování)

2.1 Vzdálené repozitáře

Verzovací systém GIT umí kromě sledování a uchovávání historie změn souborů vašeho projektu i koordinovat změny, které provádí na projektu více řešitelů (pracovní tým). K tomu slouží vzdálené repozitáře. Mezi nejrozšířenější systémy patří

- GitHub: Na něm si každý může zdarma zřídit vlastní repozitáře a používat je například k synchronizaci svého projektu mezi různými počítači, pro sdílení vlastních výtvorů s komunitou nebo právě pro spolupráci ve vlastním týmu.
- GitLab: Open source řešení, základní verze rovněž zdarma, k pokročilé verzi má MFF UK licenci.

Oba tyto systémy disponují webovým rozhraním, ze kterého lze projekty jednoduše spravovat, a rovněž desktopovými aplikacemi, které umožňují pracovat s lokálními repozitáři pomocí jednoduchého rozhraní. Pro GitHub existuje například GitHub Desktop, ale i spousta dalších. Projekty mohou být soukromé (přístup k nim máte pouze vy či ti, kterým pošlete pozvánku) nebo veřejné (přístup má kdokoliv).

Tyto zápisky a vzorové ukázky kódů jsou veřejně na GitHubu a můžete k nim dostat na adrese `https://github.com/PavelStransky/PCInPhysics`. K dispozici je samozřejmě celý repozitář s historií se všemi commity a se všemi vývojovými větvemi. Repozitář můžete stáhnout buď z uvedené webové stránky (zelené tlačítko `Clone or download`), nebo pomocí programu git.

- `git clone https://github.com/PavelStransky/PCInPhysics`

  Vytvoří adresář `PCInPhysics` a do něj stáhne celý repozitář. V případě těchto zápisků se jedná o tento soubor v LaTeXu, výsledné PDF, EPS verze všech obrázků a všechny zdrojové kódy.

- `git remote -v`

  V adresáři s lokálním repozitářem ukáže, na jaký vzdálený repozitář je navázaný. Stejně jako základní větev se standardně jmenuje `master`, vzdálený repozitář se standardně jmenuje `origin`. Vy můžete mít na jeden projekt navázáno více vzdálených repozitářů, každý pak samozřejmě musíte pojmenovat jinak.

- `git remote add origin https://github.com/Uzivatel/VzdalenyRepozitar`

  Přidá do vašeho projektu odkaz na vzdálený repozitář.

- `git remote show origin`

  Zobrazí informace o vzdáleném repozitáři.

- `git pull`

  Do adresáře s lokálním repozitářem stáhne aktuální verzi aktuální větve. Pokud máte lokálně rozpracované změny, stažení se nepovede. Pokud máte uložené změny (commit), git se automaticky pokusí vaše změny sloučit se změnami v globálním repozitáři (merge).

- `git fetch`

  Stáhne celý vzdálený repozitář (všechny větve).

- `git push origin master`

  Do vzdáleného repozitáře `origin` zapíše vaši větev `master`. Vzdálený repozitář může být nastaven tak, že vaše změny musí ještě někdo schválit.

- `git push`

  Zkrácený zápis, pokud jste dříve nastavili pomocí příkazu `git push --set-upstream origin master` název lokální větve a příslušného vzdáleného repozitáře.

Další informace najdete například v dokumentaci.

Úkol 2.1: *Stáhněte si do svých počítačů (naklonujte si) z GitHubu repozitář s poznámkami k tomuto cvičení. V budoucnu si pomocí příkazu* `git pull` *stahujte aktuální verze. Můžete si vytvořit pracovní větev poznámek a v ní si s kódy hrát. V hlavní větvi* `master` *se vám uchová originální verze ze vzdáleného repozitáře.*

Úkol 2.2: *Vytvořte si účet na GitHubu, vytvořte prázdný projekt a navažte si ho s dříve na cvičení vytvořeným lokálním repozitářem pomocí příkazů* `git remote add` *(plný příkaz výše). Následně si do vzdáleného repozitáře nahrajte lokální repozitář pomocí příkazu* `git push`*.*

2.2 .gitignore

Překladače programovacích jazyků často vytvářejí v adresáři vašeho projektu dočasné pomocné soubory, které nechcete, aby se staly součástí repozitáře (tyto soubory nenesou žádnou relevantní informaci, navíc mohou na různých počítačích vypadat jinak podle toho, jaký překladač či jaké vývojové prostředí zrovna použijete). Abyste mohli používat příkazy pro hromadné sledování či zapisování souborů `git add *` a `git commit -a`, musíte GITu naznačit, jaké soubory má ignorovat. K tomu slouží soubor `.gitignore`.

Každé pravidlo v souboru `.gitignore` zabírá jeden řádek. Řádek, který začíná znakem #, je ignorován a může sloužit například jako komentář. Příklady jednotlivých řádků:

- `tajne.txt`

  Ignoruje soubor s názvem `tajne.txt` (může obsahovat třeba přihlašovací údaje k nějaké službě a ty rozhodně nechceme sdílet ani archivovat; nezapomeňte, že co je jednou zapsané v repozitáři, z něj až na výjimky nelze odstranit).

- `*.log`

  Ignoruje všechny soubory s příponou `log`,

- `!important.log`

  ale neignoruje soubor `important.log`.

- `*.[oa]`

  Ignoruje všechny soubory s příponou `o` nebo `a`.

- `temp/`

  Ignoruje všechny soubory v podadresáři `temp`.

- `doc/**/*.pdf`

  Ignoruje všechny soubory s příponou `pdf` v podadresáři `doc` a ve všech jeho podadresářích. Neignoruje však soubory s příponou `pdf` v hlavním adresáři projektu.

Další příklady jsou například zde.

Pokud na GitHubu zakládáte nový projekt, můžete upřesnit, jaký programovací jazyk budete používat a GitHub automaticky vytvoří optimální soubor `.gitignore`.

Úkol 2.3: *Podívejte se do souboru `.gitignore` v repozitáři k těmto zápiskům. Zatímco Python si téměř žádné pomocné soubory nevytváří, LaTeX jich generuje požehnaně. Proto je tento soubor celkem dlouhý.*

3 Náhodná procházka

Náhodná procházka je jeden ze základních prostředků, jak simulovat velké množství nejen fyzikálních procesů (například pohyb Brownovské částice, fluktuace akciového trhu, pohyb opilce z hospody atd.). V dalších cvičeních si ukážeme, jak se pomocí náhodné procházky dá jednoduše hledat minimum funkcí (a to i funkcí více proměnných).

Algoritmus pro náhodnou procházku je následující: v každém časovém kroku uděláme krok v d-rozměrném prostoru $\boldsymbol{y}_i \to \boldsymbol{y}_{i+1}$ takovým způsobem, aby pravděpodobnost pohybu do všech směrů byla stejná. Délka kroku l se volí buď náhodná, nebo konstantní.

3.1 Pseudonáhodná čísla

V Pythonu lze pseudonáhodná čísla generovat pomocí několika knihoven.

- Pro základní použití se používá knihovna **`random`** (dokumentace). Z ní nejdůležitější funkce jsou tyto:
  - `random()`: *reálné* pseudonáhodné číslo x rovnoměrně z intervalu $x \in \langle 0; 1)$.
  - `uniform(a,b)`: *reálné* pseudonáhodné číslo x rovnoměrně z intervalu $x \in \langle a; b\rangle$.
  - `gauss(mu,sigma)`: pseudonáhodné číslo z Gaussovského rozdělení se střední hodnotou `mu` a směrodatnou odchylkou `sigma`.
  - `randint(a,b)`: *celé* pseudonáhodné číslo d z intervalu $a \leq d < b$.
  - `seed(s)`: nastaví počáteční násadu generátoru podle parametru `s` (pro jednu konkrétní násadu bude generátor dávat stejnou sekvenci čísel). Parametr může být jakéhokoliv typu, tedy číslo, řetězec atd. Pokud se parametr neuvede, použije se jako násada systémový čas.
  - `choice(l)`: vybere pseudonáhodně element ze seznamu `l`.
  - `shuffle(l)`: promíchá elementy v seznamu `l`.
- Pro pokročilejší použití je výhodnější modul **`numpy.random`** (dokumentace). Ta umožňuje zvolit vlastní generátor pseudonáhodných čísel, generovat čísla z celé řady statistických rozdělení a generovat naráz celé vektory či matice.
  - `generator = default_rng()`: Inicializuje standardní generátor pseudonáhodných čísel.
  - `generator = Generator(PCG64())`: Inicializuje specifický generátor pseudonáhodných čísel (v tomto případě PCG-64, což je O'Neillův permutační kongruenční generátor).
  - `generator.random(size=10)`: vektor délky 10 s elementy z rovnoměrného rozdělení z intervalu $\langle 0; 1)$.
  - `generator.normal(size=10)`: vektor délky 10 s elementy z normálního Gaussova rozdělení se střední hodnotou 0 a směrodatnou odchylkou 1.
  - `generator.normal(loc=1, scale=2, size=(10,10))`: matice rozměru 10×10 s elementy z normálního Gaussova rozdělení se střední hodnotou 1 a směrodatnou odchylkou 2.

Úkol 3.1: *Naprogramujte náhodnou procházku ve 2D rovině. Délku kroku volte konstantní, například $l = 1$, směr volte náhodně. Začněte například z bodu $(0; 0)$ a procházku ukončete po N krocích. Případně ji můžete ukončit poté, co se dostanete z předem zadané oblasti, například ze čtverce o hraně délky 100. Uchovávejte celou procházku v poli či seznamu. Nakonec trajektorii vykreslete do grafu.*

Řešení 3.1: *Řešení je naimplementováno v souboru* **`RandomWalk2D.py`**.

- `RandomDirection2D` *vrátí náhodný směr ve 2D rovině [generuje náhodný úhel ϕ, směr je dán jednotkovým vektorem se složkami $(\cos\phi, \sin\phi)$].*
- `RandomWalk2D` *vykreslí do grafu náhodnou procházku s* `numSteps` *kroky omezenou ve čtverci rozměru* `2 boxSize`$\times$`2 boxSize` *a náhodnou procházku vrátí.*
- `RandomWalk2DInteractive` *generuje náhodnou procházku a vykresluje ji do grafu krok po kroku. Musí být zapnutý interaktivní mód vykreslování* `plt.ion()` *a v prostředí Spyder vypnuto použití inline grafů příkazem* `%matplotlib auto` *v konzoli REPL.*

Vzorový kód je napsán takovým způsobem, aby mohl být přímočaře rozšířen pro vícerozměrnou náhodnou procházku. Důležité body kódu jsou tyto:

- *Cyklus* `for` *v Pythonu prochází jakýkoliv objekt nazvaný* **iterátor**. *Už jsme se seznámili s cyklem přes prvky seznamu, pole či řádky textového souboru. Pokud chceme jednoduchý cyklus přes po sobě jdoucí celá čísla, použijeme iterátor* `range(start,stop[,step])` *(iteruje se přes celá čísla počínající* `start` *a končící posledním číslem ostře menším než* `stop`*).*

- `numpy.allclose(a,b)` *porovnává prvek po prvku řad* `a` *a* `b` *a pokud jsou všechny prvky blízko sebe v rámci zadané tolerance, vrátí* `True`*. Konečná tolerance je důležitá proto, aby byly ošetřeny případy, kdy důsledkem konečné strojové přesnosti některá fakticky stejná čísla běžné porovnání vyhodnotí jako rozdílná, například* `x = 101*0.1` *a* `y = 10.1`*. Toleranci pro porovnávání lze nastavit.*

  Pro porovnání jednotlivých čísel v mezích tolerance slouží funkce `numpy.isclose(x,y)`*.*

- *Operátor* `%`*: zbytek po dělení.*

- *Rozmyslete si dobře podmínku, která zjišťuje, zda jsme uvnitř omezujícího čtverce. Podmínka by dobře posloužila i v případě, kdybychom chtěli implementovat cyklické okrajové podmínky.*

Úkol 3.2: *Zamyslete se nad tím, jak byste realizovali náhodnou procházku s konstantní délkou kroku v $d > 2$ rozměrech. Důležité je dodržet požadavek, aby pohyb do jakéhokoliv směru nastával se stejnou pravděpodobností (esence úlohy tedy spočívá v generování náhodného směru v d-rozměrném prostoru).*

Řešení 3.2: *Zatímco pro směr ve 2D rovině stačí náhodě generovat jeden úhel (předchozí úloha), vícerozměrné úlohy jsou komplikovanější. Přímé rozšíření 2D případu do 3D (či do vyšších dimenzí) za generování více úhlů a použití (hyper-)sférických souřadnic k cíli nevede — takto otrocky generované směry upřednostňují okolí pólů před rovníkem (rozmyslete). K úspěšnému generování náhodného kroku je nutné využít jeden z následujících algoritmů:*

1. *Hyperkoule vepsaná v hyperkrychli.*

   (a) *Nagenerujeme bod v hyperkrychli o hraně délky 2, tj. generujeme vektor $\boldsymbol{v}$ s d složkami, přičemž každá složka je náhodné číslo z rovnoměrného rozdělení $(-1, 1)$.*

   (b) *Zkontrolujeme, zda bod leží uvnitř vepsané jednotkové koule například tak, že spočítáme jeho normu $v = |\boldsymbol{v}|$ a porovnáme, zda $n \leq 1$.*

   (c) *Pokud ne, opakujeme postup od začátku. Pokud ano, nagenerovaný bod promítneme na jednotkovou kouli (jinými slovy vektor $\boldsymbol{v}$ nanormujeme) a získaný jednotkový vektor $\hat{\boldsymbol{v}} \equiv \boldsymbol{v}/v$ udává hledaný směr.*

   Tato metoda je obrovsky neefektivní, pokud je dimenze d vysoká, poněvadž v tom případě většina nagenerovaných bodů leží vně vepsané hyperkoule a je zahozena. Poměr celkového počtu nagenerovaných bodů ku úspěšným zásahům vnitřku hyperkoule lze snadno spočítat. Objem hyperkrychle o hraně délky 2 je

$$V_d^{(krychle)} = 2^d, \tag{27}$$

   objem vepsané hyperkoule o poloměru 1 je

$$V_d^{(koule)} = \frac{\pi^{\frac{d}{2}}}{\Gamma\left(\frac{d}{2}+1\right)}, \tag{28}$$

   kde Γ je Eulerova gama funkce. Vzájemný poměr

$$\eta_d \equiv \frac{V_d^{(krychle)}}{V_d^{(koule)}} = \left(\frac{2}{\sqrt{\pi}}\right)^d \Gamma\left(\frac{d}{2}+1\right) \tag{29}$$

udává, kolik bodů musíme průměrně nagenerovat, abychom se trefili do hyperkoule (reciproká hodnota $1/\eta_d$ určuje pravděpodobnost, že se do hyperkoule trefíme). Zatímco pro $d = 3$ je $\eta_3 \approx 1.91$, pro $d = 10$ již $\eta_{10} \approx 401$, tj. pro nalezení jednoho náhodného směru v desetirozměrném prostoru musíme nagenerovat v průměru přes 4000 náhodných čísel. Z posledního vztahu je vidět, že s rostoucí dimenzí roste η_d exponenciálně.

2. *Náhodný Gaussovský vektor.*

   (a) *Nagenerujeme vektor $\boldsymbol{n}$ s d složkami, přičemž každá složka je číslo z normálního Gaussovského rozdělení $N(0, 1)$.*

   (b) *Vektor nanormujeme a získáme hledaný náhodný směr $\hat{\boldsymbol{n}} \equiv \boldsymbol{n}/n$.*

   Tato metoda je mnohem přímočařejší než předchozí, předpokladem je jen mít k dispozici generátor čísel vybraných z normálního rozdělení.

   Důkaz, že tato metoda dává opravdu náhodný směr v d dimenzích, a další informace o metodě se naleznete v článcích [1, 2].

3. *Speciální případ $d = 3$ (náhodný let).*

   (a) *Generujeme dvě náhodná čísla $\xi_{1,2}$ z rovnoměrného rozdělení na intervalu $\langle 0; 1)$.*

   (b) *Sférické úhly jednotkového směru jsou pak*

$$\begin{aligned} \phi &= 2\pi\xi_1, \\ \theta &= \arccos\left(1 - 2\xi_2\right), \end{aligned} \tag{30}$$

   takže hledaný jednotkový vektor $\hat{\boldsymbol{n}}$ do náhodného směru má komponenty

$$\begin{aligned} \hat{n}_x &= \sin\theta\cos\phi = \sqrt{1 - (1 - 2\xi_2)^2}\cos 2\pi\xi_1, \\ \hat{n}_y &= \sin\theta\sin\phi = \sqrt{1 - (1 - 2\xi_2)^2}\sin 2\pi\xi_1, \\ \hat{n}_z &= \cos\theta = 1 - 2\xi_2. \end{aligned} \tag{31}$$

   Ve více rozměrech je tento přístup prakticky nerealizovatelný (vede na problém inverzních funkcí k funkcím daným řadou goniometrických funkcí).

Náhodná procházka v d-rozměrném prostoru je naprogramována v souboru **`RandomWalk.py`**. *Funkce* `RandomDirection` *generuje směr pomocí 1. metody, funkce* `RandomDirectionGaussian` *pomocí 2. metody. Zkuste si spočítat náhodnou procházku pro $d = 10$ oběma metodami. Uvidíte, že i pro takto relativně „malou“ dimenzi je rozdíl ve výpočetních časech je dramatický.*

4 Hledání minima funkce

Náhodnou procházku lze úspěšně použít k hledání minima funkce obecně více proměnných. Představte si funkci dvou proměnných jako zvlněnou „krajinu“ v noci. Potřebujete se vrátit k chatě, která se nachází pod vámi hluboko v úkolí. Je tma a nevidíte jakým směrem se vydat. Zkusíte tedy udělat náhodný krok a pokud povede dolů, vykročíte. Pokud by však krok vedl nahoru, zůstanete na místě a zkusíte nový směr.

Úkol 4.1: *Rozšiřte program pro náhodnou procházku tak, aby hledal minimum funkce dvou proměnných $f(x, y)$. Otestujte svůj program pro kvadratickou funkci*

$$f(x, y) = x^2 + y^2 \tag{32}$$

a pro Rosenbrockovu funkci

$$f(x, y) = (a - x)^2 + b\left(y - x^2\right)^2 \tag{33}$$

vypadající jako velmi pozvolna klesající hluboké údolí ve tvaru paraboly. Tato funkce se používá k testování rychlosti a efektivity minimalizačních algoritmů. Její minimum se nachází v bodě (a, a^2) *a hodnoty parametrů nejčastěji se volí* $a = 1, b = 100$.

Implementujte vhodným způsobem ukončení náhodné procházky, tj. okamžik, kdy jste již dorazili do minima funkce.

Úkol 4.2: *Náhodnou procházku zakreslete jako čáru do grafu společně s konturovým grafem potenciálu. Návod na nakreslení konturového grafu pomocí funkce* `matplotlib.pyplot.contourf` *naleznete v souboru* **`Contourf.py`**.

Úkol 4.3: *Rozšiřte kód tak, aby počítal i vícerozměrnou náhodnou procházku, a najděte pomocí něho minimum funkce čtyř proměnných*

$$\begin{aligned} f(s,t,u,v) = {} & \frac{1}{4}\left(s^2+t^2+u^2+v^2\right) \\ & - \frac{1}{2}\left[\left(s^2+t^2\right)\left(2-s^2-t^2-u^2-v^2\right)+(su-tv)^2\right] \\ & + \frac{s}{2}\sqrt{2-s^2-t^2-u^2-v^2}. \end{aligned} \tag{34}$$

Jednoduchá náhodná procházka funguje dobře pro funkce s jedním minimem. V obecném případě má však funkce více lokálních minim a právě uvedený algoritmus skončí náhodně v jednom z nich, ze kterého se již nedokáže dostat ven. Při tom rozhodně nemusí jít o minimum nejhlubší (globální).

Hledání globálního minima funkce mnoha proměnných je obecně velmi komplexní problém. Dva nejjednodušší postupy, kterými můžeme vylepšit stávající metodu pomocí náhodné procházky, jsou následující:

- Provedeme několik náhodných procházek, které obecně dojdou do různých lokálních minim. Následně porovnáme konečné funkční hodnoty a vybereme to minimum, které má hodnotu nejnižší.
- Provedeme jednu náhodnou procházku doplněnou o *Metropolisův algoritmus*.

4.1 Metropolisův algoritmus

Metropolisův algoritmus rozšiřuje náhodnou procházku o konečnou teplotu. Je inspirován termodynamickým Boltzmannových rozdělením energie: máme tepelnou energii, díky které můžeme při náhodné procházce s určitou pravděpodobností udělat krok i „do kopce“, avšak čím je kopec strmější, tím bude pravděpodobnost takovéhoto kroku menší.

Předpokládejme, že jsme na vrstevnici s funkční hodnotou f a nová funkční hodnota po provedení kroku náhodné procházky by byla $f_{\text{nová}} > f$. Při minimalizaci pomocí obyčejné náhodné procházky bychom tento krok neprovedli. V Metropolisově algoritmu krok provedeme s pravděpodobností

$$p = \mathrm{e}^{\frac{f-f_{\text{nová}}}{T}}, \tag{35}$$

kde T je parametr, který má roli „teploty“: pokud $T = 0$, žádný tepelný pohyb neexistuje, krok do kopce nikdy neprovedeme a vracíme se tak k obyčejné minimalizaci. Pokud $T \to \infty$, uděláme krok do kopce s pravděpodobností $p \to 1$, což znamená, že tepelný pohyb zcela převládá, my se pohybujeme zcela náhodně a potenciál pod sebou vůbec necítíme.

V praxi je největší umění zvolit správnou hodnotu teploty. Pokud zvolíme teplotu nízkou, skončíme v lokálním minimu a už se z něj nedostaneme, pokud naopak příliš vysokou, budeme chaoticky procházet krajinou naší funkce a žádné minimum nenajdeme. Dobrá volba je začít spíš s vyšší teplotou a teplotu postupně snižovat. Jakmile se ocitneme zaseklí v nějakém minimu, můžeme teplotu zase trochu zvýšit a tím vyzkoušet, zda se nepřesuneme do nějakého minima hlubšího.

Úkol 4.4: *Naprogramujte Metropolisův algoritmus a odlaďte ho na případu funkce*

$$f(x, y) = x^4 - 2x^2 + x + y^2. \tag{36}$$

Tato funkce má dvě lokální minima (jedná se o vzorovou funkci ze souboru `Contourf.py`*).*

4.2 Minimalizace pomocí knihovny SciPy

Python obsahuje funkci pro hledání minima `minimize` v knihovně `scipy.optimize` (z této knihovny jsme v jednom z prvních cvičení využívali funkci `least_squares` pro hledání optimální hodnoty parametrů funkce fitující zadaná data metodou nejmenších čverců).

Úkol 4.5: *Prostudujte dokumentaci k funkci* `minimize` *a vytvořte kód, který tuto funkci využije k najití minima všech doposud studovaných funkcí dvou a více proměnných.*

Reference

[1] M.E. Muller, *A note on a method for generating points uniformly on n-dimensional spheres*, Communications of the Asociation for Computing Machinery **2**, 19 (1959).

[2] G. Marsaglia, *Choosing a Point from the Surface of a Sphere*, The Annals of Mathematical Statistics **43**, 645 (1972).