Zápisky k předmětu Využití počítačů ve fyzice

Pavel Stránský

5. května 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*

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

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).*

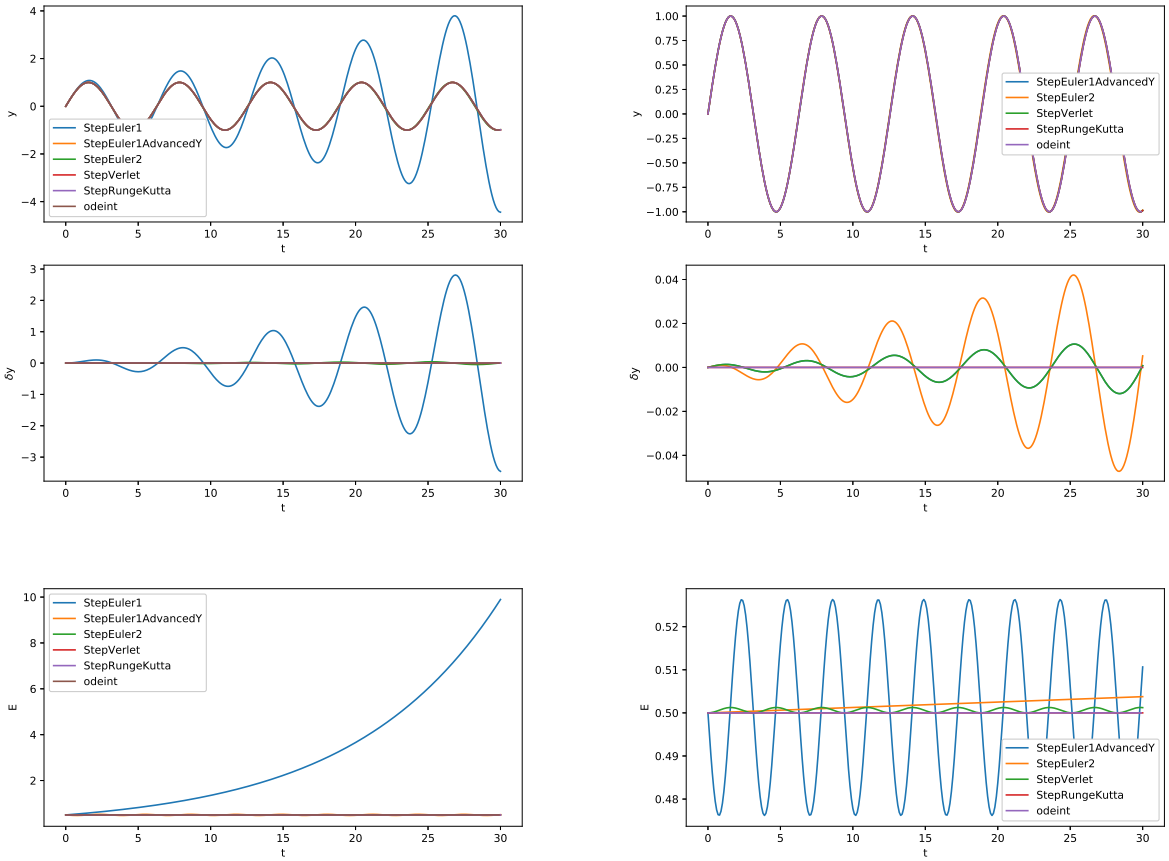
- `ODE.py`*: Modul napsaný dostatečně obecně, aby ho bylo možné použít na řešení libovolných soustav diferenciálních rovnic.*
  - `StepEuler1`*: Integrační krok Eulerovy metody 1. řádu.*
  - `StepEuler2`*: Integrační krok Eulerovy metody 2. řádu.*
  - `StepVerlet`*: Integrační krok Verletovy metody.*
  - `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.*

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

– `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.*



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 $E = \frac{1}{2}$.

Příslušné grafy jsou zobrazeny v obrázku 1.

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

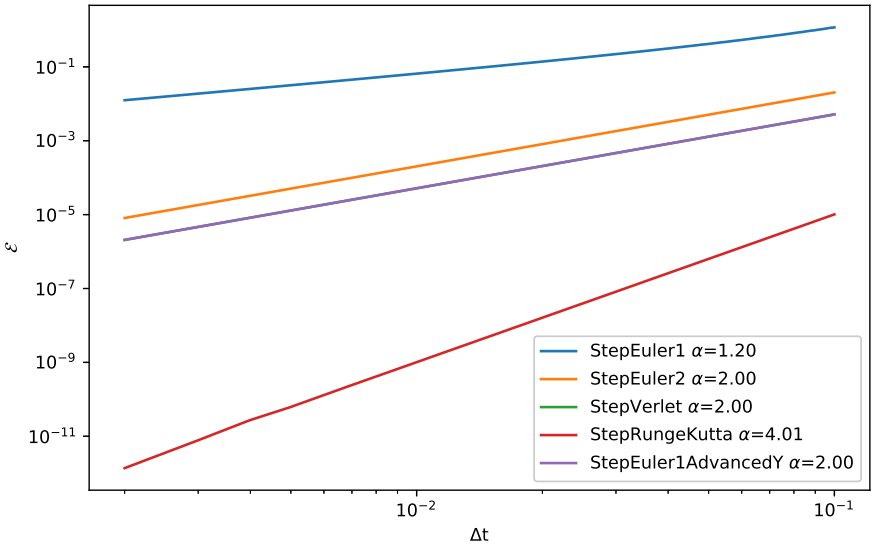
$$\mathcal{E} = \sqrt{\frac{1}{n}\sum_{i=0}^{n-1} (y_i - \sin t_i)^2} \tag{22}$$

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 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.*

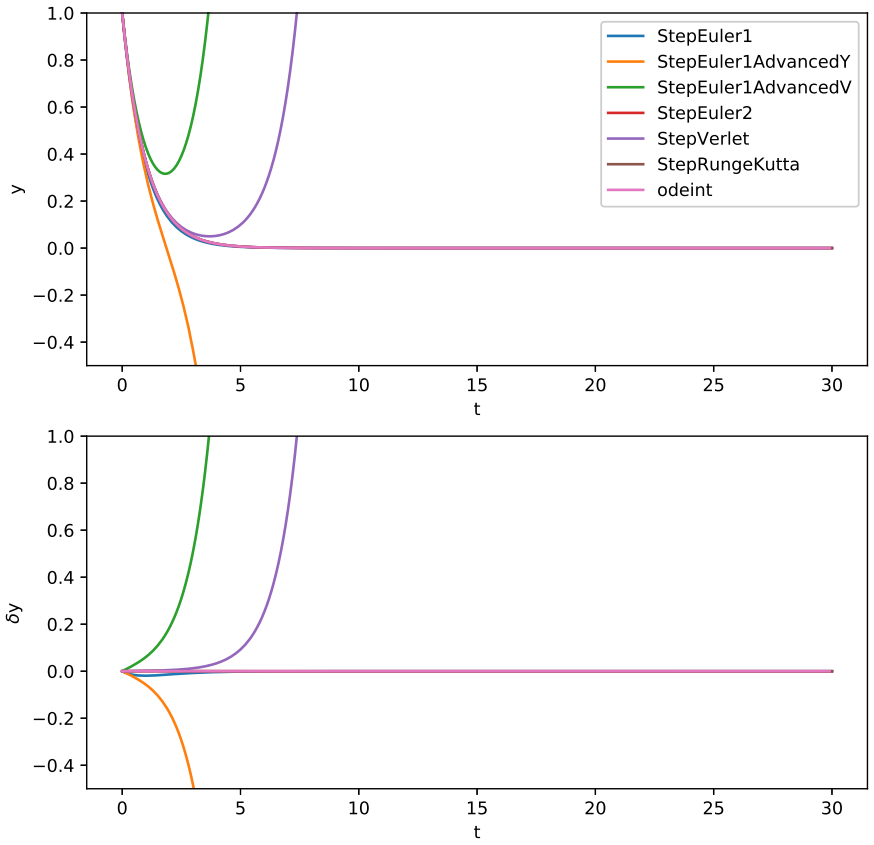
V obrázcích jsou výsledky pouze pro metodu předbíhající v souřadnici. Díky symetrii jsou výsledky pro metodu předbíhající v rychlosti identické.

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.*



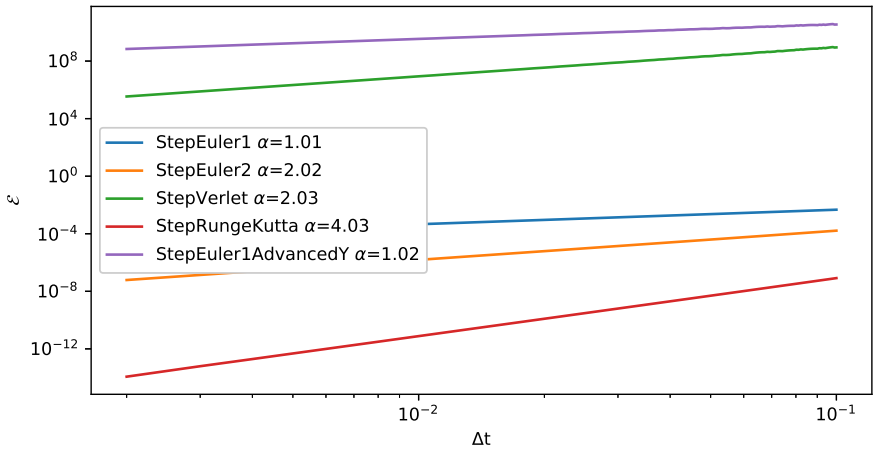
Obrázek 3: Totéž jako v obrázku 1, avšak pro exponenciálně klesající systém daný rovnicí (25). Symplektické algoritmy jsou nestabilní.

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.



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.

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 zmenšujícícm 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 (modulo).*

- *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.

Z tohoto výpočtu také vyplývá, že pokud bychom nezahazovali body ležící mimo hyperkouli, pak bychom ve výsledné procházce výrazně upřednostňovali pohyb podél diagonál, a to tím více, čím vyšší je dimenzionalita procházky (u $d = 10$ bychom podél diagonál vyrazili s více než 99% pravděpodobností).

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

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

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.

Řešení 4.1: *Vzorový kód je naimplementován v souboru* `RandomWalkMinimize.py` *a vychází z náhodné vícerozměrné procházky z modulu* `RandomWalk.py` *(z tohoto modulu kód využívá funkci* `RandomDirectionGaussian`*). K minimalizaci jsou v kódu dvě různé metody:*

- `Minimize`*: Hledá minimum funkce* `Function` *z počátečního bodu daného parametrem* `initialCondition`*. Pokud tento parametr není specifikován, zvolí se počáteční bod náhodně z* `dimension`*-rozměrné hyperkrychle se středem v počátku a délkou hrany* `2*initialConditionBox`*. Výpočet je ukončen, pokud se kódu* `maxFailedSteps`*-krát po sobě nepodaří udělat úspěšný krok (krok směrem k menší funkční hodnotě). Každý náhodný krok má konstantní délku danou parametrem* `stepSize`*.*

- `MinimizeAdaptive`*: Předchozí metoda hledání minima má chybu $\Delta x_i \approx$* `stepSize`*. Pro zmenšení chyby je v ní nutné zmenšit délku kroku náhodné procházky. Pokud tak učiníme, výpočetní čas T se výrazně prodlouží (desetkrát za každý řád zpřesnění výsledku, tj. $T \propto 1/$*`stepSize`*). Mnohem vhodnější je začít s velkým krokem a krok zmenšovat postupně. Tak postupuje tato funkce: začne s krokem* `initialStepSize` *a pokaždé, když se jí nepodaří* `maxFailedSteps`*-krát provést úspěšný krok, zmenší délku kroku na polovinu. Výpočet probíhá do chvíle, dokud je délka kroku větší než* `finalStepSize`*. Konečná chyba je tedy $\Delta x_i \approx$* `finalStepSize` *a doba výpočtu roste pouze logaritmicky se zmenšováním této chyby, tj. $T \propto 1/\log$* `finalStepSize`*.*

Obě funkce vracejí řadu s celou náhodnou procházkou. Nalezené minimum je tedy v posledním bodě této řady.

V kódu se vyskytuje jedna nová věc, a to je metoda `format` *řetězce. Tato metoda postupně na místa řetězce vyznačená dvojicí znaků {} vloží své argumenty, přičemž argumenty mohou být libovolné objekty. Pro více informací k formátování řetězců doporučuji tento výborný návod.*

Konkrétní funkce $f(x, y)$ a $g(x, y)$ ze zadání jsou v souboru `Minimize.py`*.*

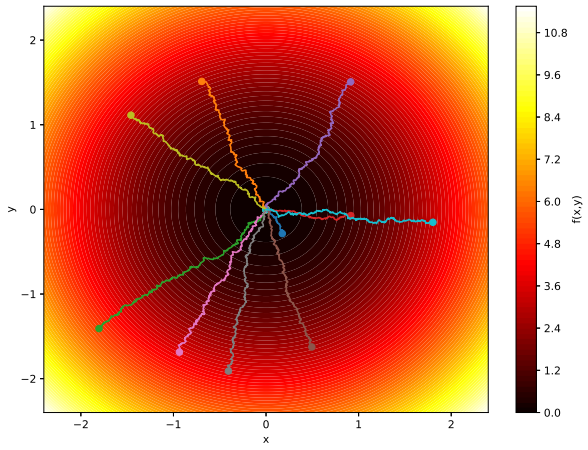
Úkol 4.2: *Náhodnou procházku zakreslete jako čáru do grafu společně s konturovým grafem potenciálu. Návod na nakreslení konturového grafu pomocí funkce* `matplotlib.pyplot.contourf` *naleznete v souboru* `Contourf.py`*.*

Řešení 4.2: *Vzorový kód je v souboru* `Minimize.py` *a využívá jako modul* `RandomWalkMinimize.py`*. Obsahuje následující funkce:*

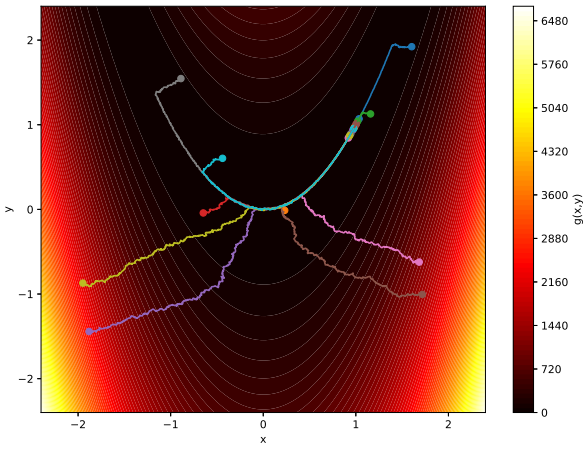
- `ShowGraph`*: Vykreslí konturový graf funkce* `Function` *a do něj zakreslí křivky (náhodné procházky) z parametru* `paths`*. Pokud tento parametr není specifikován, vykreslí pouze konturový graf s funkcí. Meze funkce pro vykreslení jsou dány parametrem* `boxSize`*.*

- `MultiplePaths`*: Vypočítá celkem* `numPaths` *jednoduchých náhodných minimalizačních procházek a vykreslí je v grafu společně s konturovým grafem minimalizované funkce.*

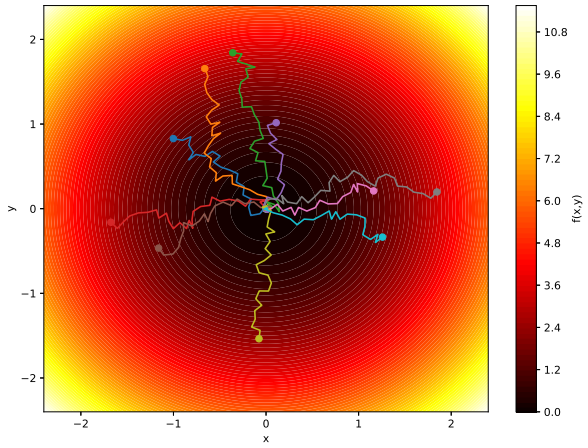
- `MultiplePathsAdaptive`*: Vykreslí totéž, avšak použije metodu s adaptivním krokem.*



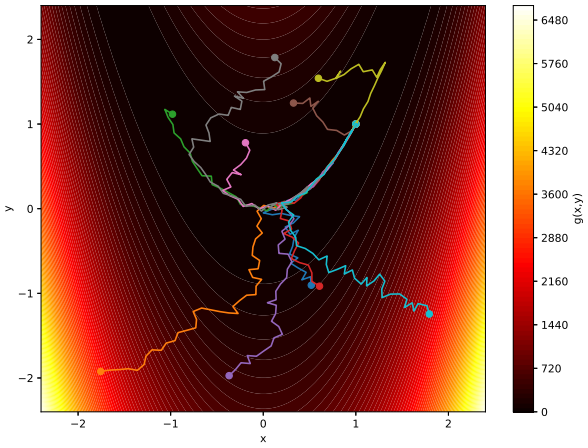
(a) `MultiplePaths(f)`



(b) `MultiplePaths(g)`



(c) `MultiplePathsAdaptive(f)`



(d) `MultiplePathsAdaptive(g)`

Obrázek 5: Minimalizace kvadratické funkce (32) a Rosenbrockovy funkce (33) pomocí náhodné procházky a deseti náhodných trajektorií. *1. řádek:* Minimalizace metodou `Minimize` s délkou kroku `stepSize` $= 0.01$ a kritériem ukončení `maxFailedSteps` $= 100$. Chyba určení minima je $\Delta x, y \approx 0.01$, což v případě Rosenbrockova minima dává velkou chybu: Rosenbrockovo „údolí“ je velmi pozvolné podél a strmé napříč, což způsobuje, že trajektorie končí v širokém okolí skutečného minima v bodě $(x, y) = (1, 1)$. Počet kroků, než je výpočet ukončen, je zde ≈ 1000. *2. řádek:* Minimalizace vylepšeným algoritmem s postupně se zmenšující krokem. Na počátku je krok velký, `initialStepSize` $= 0.1$, ale postupně se zmešuje až k `finalStepSize` $= 10^{-6}$. To stačí i pro přesné nalezení minima Rosenbrockovy funkce. Počet nutných kroků zde je ≈ 3000.

Výsledky pro funkce f a g ze zadání jsou zobrazeny na obrázku 5.
V kódu jsou následující vychytávky:

- `Function.__name__` *vrátí originální název funkce. Každá funkce v Pythonu má hodnotu tohoto atributu nastavenou.*
- *V cyklu* `for _ in range(0, numPaths)` *je proměnná označena podtržítkem* `_`*. To je v Pythonu běžně užívané označení pro proměnnou, jejíž hodnota se nikde nepoužívá.*
- *Z argumentů funkce* `MultiplePaths` *chceme většinu pojmenovaných argumentů předat přímo funkci* `Minimize`*, aniž by nás zajímalo, kolik takových argumentů je a jak se jmenují. K tomu slouží proměnná uvozená dvěma hvězdičkami* `**kwargs`[2]*. Ta v sobě nese všechny nepoužité pojmenované argumenty (nikoliv tedy* `initialConditionBox`*, který musíme při volání funkce* `Minimize` *předat explicitně). Typ proměnné* `kwargs` *je slovník.*

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

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

Řešení 4.3: *Volání funkce* `MinimizeAdaptive(h, dimension=4, initialConditionBox=1)` *vypočítá minimum v bodě*

$$\begin{aligned} s_{min} &\approx -0.913 \\ t_{min} = u_{min} = v_{min} &\approx 0.000 \\ h_{min} \equiv h(s_{min}, t_{min}, u_{min}, v_{min}) &\approx -0.771. \end{aligned} \tag{35}$$

V případě minimalizace této funkce je nutné ošetřit odmocninu, pod kterou se může během náhodné procházky objevit záporné číslo, což způsobí konec výpočtu s chybou. V kódu je to vyřešeno tak, že v případě kroku do této „nedovolené oblasti“ vrátí funkce `h` *hodnotu ∞ (v Pythonu* `float("inf")`*), která je větší než všechna možná jiná čísla, a tím tento krok zakáže. Ze stejného důvodu je zmenšena oblast hledání počáteční polohy pomocí parametru* `initialConditionBox=1`*.*

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*.

[2] kw je zkratka pro *keywored*; proměnná se samozřejmě může jmenovat jakkoliv, `kwargs` je jen standardně používané označení.

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

$$p = \mathrm{e}^{\frac{f - f_{\text{nová}}}{T}}, \tag{36}$$

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*

$$r(x, y) = x^4 - 2x^2 + x + y^2. \tag{37}$$

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

Řešení 4.4: *Pro ukázku, že standardní minimalizace pomocí náhodné procházky vede náhodně do různých lokálních minim, slouží obrázek 6(a).*

Metropolisův algoritmus je naprogramován v souboru `MetropolisMinimize.py`*.*

- `Minimize`*: Minimalizuje funkci* `Function` *se zadanou teplotou* `temperature` *a s fixní délkou kroku* `stepSize`*. Výpočet je ukončen přesně po* `maxSteps` *krocích. Kritérium ukončení výpočtu z metody bez teploty zde nebude fungovat, jelikož konečná teplota způsobuje neustálý „tepelný pohyb“, nikdy tedy nedojde k úplnému zastavení, ani pokud dosáhneme minima funkce.*

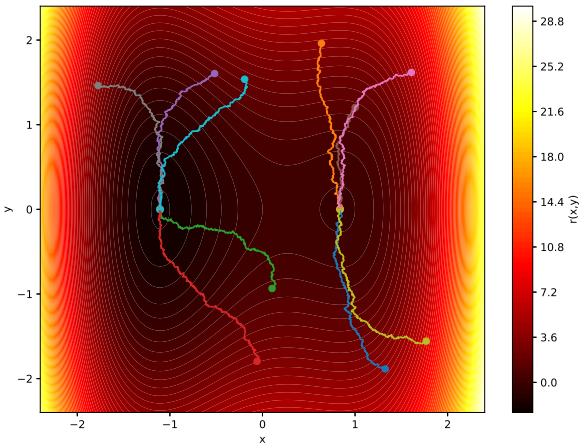
  Výsledek je zobrazen na obrázku 6(b). Je vidět, že oproti případu bez teploty se dostaneme častěji k okolí globálního minima, avšak důsledkem celkem vysoké teploty nalezená poloha výrazně fluktuuje. Navíc, a to zde není zobrazeno, výsledek velmi záleží na kombinaci teploty a délky kroku.

- `MinimizeAdaptive`*: Vylepšení spočívající v postupném zmenšování teploty a zároveň kroku. Výpočet začne s krokem* `initialStepSize` *a teplotou* `initialTemperature` *a po každých* `numStepsChange` *krocích déku kroku i teplotu vydělí dvěma. Výpočet je ukončen po dosažení délky kroku* `finalStepSize`*.*

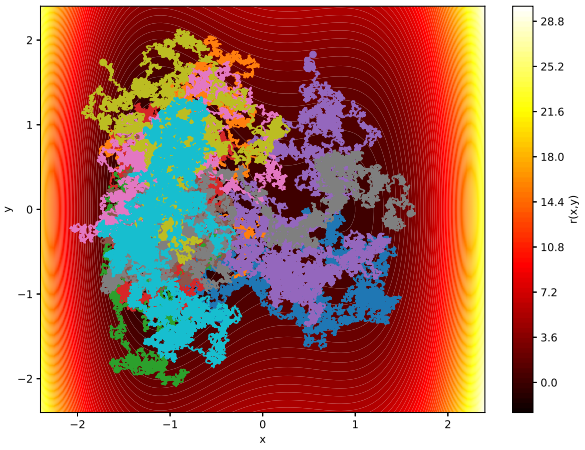
4.2 Minimalizace pomocí knihovny SciPy

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).

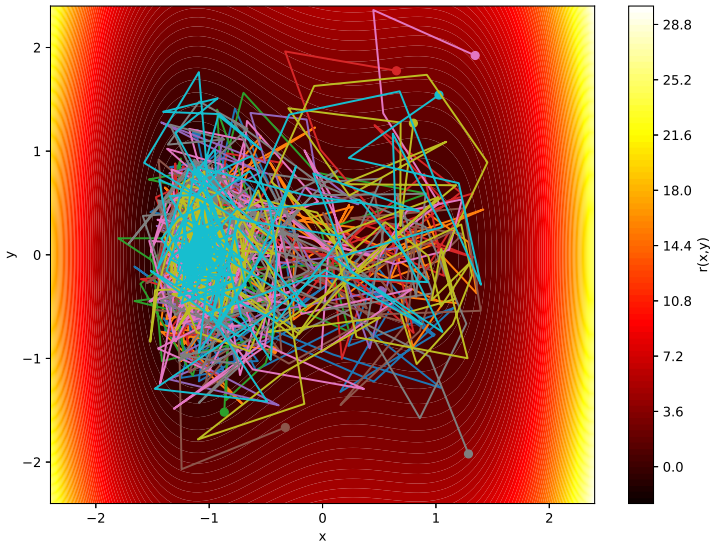
Úkol 4.5: *Prostudujte dokumentaci k funkci* `minimize` *a vytvořte kód, který tuto funkci využije k najití minima všech doposud studovaných funkcí dvou a více proměnných.*



(a) `MultiplePaths(r)` pro `RandomWalkMinimize`.



(b) `MultiplePaths(r, numSteps=30000)` pro `MetropolisMinimize`.



(c) `MultiplePathsAdaptive(r)` pro `MetropolisMinimize`.

Obrázek 6: Porovnání různých metod minimalizace dvoujámové funkce (37). (a) Minimalizace obyčejnou náhodnou procházkou, která dokonverguje do náhodného lokálního minima. Následně je možné vybrat to minimum, které je hlubší. (b) Metropolisův algoritmus s konstantní teplotou $T = 1$. Kvůli vysoké teplotě dráha výrazně fluktuuje, avšak postupně dokonverguje k okolí hlubšího globálního minima. (c) Metropolisův algoritmus s postupně se zmenšujícím krokem a teplotou. Zde do globálního minima postupně zkonvergují všechny trajektorie. Počet použitých kroků je 2000.

Řešení 4.5: *Knihovní funkci voláme příkazem* `minimize(f, initialCondition)`*, kde parametr* `initialCondition` *je n-tice udávající počáteční bod, ze kterého minimalizační procedura vystartuje.*

V případě funkce s více lokálními minimy tato knihovní funkce nenajde nutně minimum globální. Lze však pomocí volitelného parametru `method` *vybrat metodu minimalizace, která bude úspěšnější. Pokročilá varianta Metropolisova algoritmu je v v knihovně* `scipy.optimize` *naprogramována pod názvem* `basinhopping` *(dokumentace).*

4.3 Shrnutí

- Jeden z nejjednodušších algoritmů na hledání minima (maxima) funkce je pomocí náhodné procházky. K její implementaci stačí mít generátor náhodných čísel.
- Úspěch algoritmů založených na náhodné procházce je podmíněn tím, že každý krok procházky musí vést do libovolného směru se stejnou pravděpodobností. Při procházce v rovině toho lze docílit náhodným generováním úhlu. Ve vícerozměrné procházce je nejvýhodnější použít algoritmus založený na Gaussovském náhodném vektoru (za předpokladu, že umíme generovat čísla z Gaussovského rozdělení; jeden jednoduchý takový generátor bude obsahem dalších cvičení).
- Obecná funkce může mít více lokálních minim, přičemž náhodná procházka nás zavede do jednoho z nich, které nemusí být nejhlubší (globální). K nalezení globálního minima lze využít Metropolisův algoritmus, který k náhodné procházce přidá tepelný pohyb, a tím umožní „vyskočit“ z mělkého lokálního minima.

  Metropolisův algoritmus se využívá i pro jiné termodynamické úlohy, například k modelování spinových systémů při konečné teplotě, čímž lze studovat fázový přechod feromagnet $\leftrightarrow$ paramagnet.

5 LaTeX

Nedílnou součástí vědecké práce je prezentování výsledků, přičemž chceme, aby dokument bylo jednoduché napsat, a zároveň aby dobře vypadal. Jelikož fyzikální odborné texty obsahují velké množství rovnic a symbolů, není nejvhodnější volbou používat textové editory pro kancelářskou práci, jejichž možnosti psaní rovnic a použití vědeckých stylů jsou celkem omezené. Ve fyzice je nejběžnější psát texty v systému LaTeX.

Dnes se jedná o nesmírně obsáhlý balík různých knihoven a doplňků, pomocí kterého lze

- psát dobře vypadající a typograficky správné odborné texty s matematickými rovnicemi, obrázky a tabulkami,
- vybírat z množství profesionálně připravených stylů obsahujících podporu tvorby obsahu, rejstříku, poznámek pod čarou, kapitol, seznamů, referencí a dalších vychytávek,
- stáhnout si přímo styl časopisu či knihy, pro kterou text připravujeme,
- používat tisíce matematických symbolů či druhů písem (lze psát třeba ve švabachu, gotickém písmu či elfím písmu, jste-li fanoušci díla J.R.R. Tolkiena),
- využívat různá makra a doplňky k jednoduchému vytváření speciálních objektů (například chemických vzorců nebo Feynmannových diagramů),
- vytvářet robustní prezentace (například pomocí knihovny Beamer),
- nebo sázet notové materiály a kdovíco dalšího (rajčata však zatím ne).

Systém LaTeX má dvě vrstvy:

1. TeX: sazeč (zajišťuje, aby vše bylo na stránce tam, kde má být),

2. LATEX: typograf (zajišťuje, aby dokument dobře vypadal).

Program TEX začal vznikat v 70. letech 20. století a byl určen sázení textu a matematických rovnic při zachování vysoké typografické úrovně výsledného dokumentu. LATEX je pak nadstavba maker, která psaní dokumentů velmi zjednodušuje a zcela zakrývá sazečskou práci. Příkazy v TEXu jsou však stále dostupné.

Dokument napsaný v LATEXu striktně odděluje *text* (obsah) a *styl* (vzhled), přičemž my využijeme standardní LATEXovský styl a budeme se zabývat výhradně tím, jak napsat text. Jeho psaní se podobá programování: zdrojový soubor je textový dokument (nebo soubor textových dokumentů), který sestává z našeho textu a doplňujících příkazů. Chceme-li vidět, jak bude výsledný dokument vypadat, musíme soubor „přeložit“.

Existují různé *distribuce* LATEXu. Jedna z nejpoužívanějších je MiKTeX. Dále je nutné mít k dispozici textový editor, přičemž vhodný je ten, který rozumí TEXovským příkazům a bude umět zdrojový text poslat k překladu a zobrazit. Pokročilé editory umějí navigovat mezi zdrojovým textem k přeloženému výsledku (většinou do PDF souboru) a naopak. Z volně dostupných editorů jsou nejčastěji používané[3]

- **TeXworks**: Editor se základními funkcemi. Je součástí distribuce MiKTeX.
- TeXnicCenter: Pokročilejší editor, podpora použití projektů.
- TeXstudio: Pokročilý editor, podpora projektů, automatického doplňování, jednoduchá práce s obrázky, možnost tvorby záložek, pomocníci po tvorbu rovnic a tabulek, zvýrazňování syntaktických chyb.
- Visual Studio Code s pluginem LaTeX Workshop: Méně funkcí co se týče samotného LATEXu a komplikovaná instalace, avšak plná podpora všech funkcí tohoto rozšířeného vývojového prostředí (integrace s Gitem, pokročilé vyhledávání, programátorské možnosti editace).
- Overleaf: Online editor, ideální pro práci v týmu. Nemusíte nic isntalovat, potřebujete však připojení k internetu. Komplikované může být rozchodit nějaké méně rozšířené balíčky.

Úkol 5.1: *Nainstalujte si na svůj počítač nějakou distribuci LATEXu (doporučuji MiKTeX) a editor (doporučuji TeXstudio nebo TeXnicCenter).*

5.1 Formát TEXovského souboru

- ASCII nebo UTF-8 kódování (lze použít i obskurnější kódování, ale v dnešní době je UTF-8 dostatečně univerzální).
- *Mezery*: více mezer je interpretováno jako jedna mezera, jednoduché zalomení řádku také jako jedna mezera.
- *Nový odstavec*: dvě zalomení řádku po sobě.
- *Příkazy*: uvozeny znakem `\`, jejich povinné parametry ve složených závorkách `{...}`, volitelné parametry v hranatých závorkách `[...]`. U jmen příkazů záleží na velikosti písmen.
- *Rovnice v textu*: oddělena znaky `$...$`. Některé příkazy lze použít pouze uvnitř rovnic.
- *Komentář*: uvozen znakem `%`. Vše za tímto znakem až do konce řádky je ingnorováno.

[3] Kromě prvního editoru mám zkušenost se všemi ostatními. Tento text píšu ve Visual Studiu Code, avšak pro začátek doporučuji nejlépe TeXstudio nebo TeXnicCenter.

5.2 Struktura TEXovského souboru

5.2.1 Preambule

Jedná se o první řádky souboru, než začne vlastní tělo dokumentu. Obsahuje výčet všech použitých balíčků a parametry, které se použijí pro styl textu.

- `\documentclass[a4paper,twoside,11pt,twocolumn]{article}`: Základní specifikace stylu dokumentu (v tomto případě papír velikosti A4, dvoustránkový tisk, základní velikost písma 11 bodů, dvousloupcová sazba). Tento příkaz je povinný, musí být vždy přítomen.
- `\usepackage{epsfig}`: Použije se balíček `epsfig`.
- `\def\abs#1{\left|#1\right|}`: Definuje makro `\abs` s jedním parametrem (absolutní hodnota).

Nejčastěji používané balíčky jsou tyto:

- `\usepackage{amsfonts,amsmath,amssymb}`:[4] Rozšiřuje množství použitelných písem, matematických symbolů a struktur (např. snazší psaní matic, víceřádkových rovnic atd.).
- `\usepackage[utf8]{inputenc}`: Specifikuje UTF-8 jako vstupní kódování (jinak je očekáváno ASCII).
- `\usepackage[czech]{babel}`: Udávající české formátování (například české uvozovky) a české názvy (například Obsah, Rejstřík).
- `\usepackage{epsfig}`: Umožní vkládání vektorových EPS souborů.
- `\usepackage[unicode]{hyperref}`: Umožní vkládání hypertextových odkazů a učiní klikabilní i odkazy na rovnice, obrázky, stránky či kapitoly v textu pro snazší navigaci.

5.2.2 Tělo dokumentu

Tělo je uvozeno příkazy

```
\begin{document}
    ...
\end{document}
```

a do něj píšeme vlastní text dokumentu. Vše, co se nachází za příkazem `\end{document}`, je ignorováno.

Jednoduchý příklad LATEXovského dokumentu s vysvětlením základů psaní textu je v repozitáři v adresáři LaTeX a jmenuje se **`dokument.tex`**. K jeho přeložení budete potřebovat i přiložený EPS soubor **`kubik.eps`**.

Pokročilejším příkladem je přímo tento soubor (i jeho zdroják je v repozitáři).

Úkol 5.2: *Prostudujte si vzorový LATEXovský soubor* **`dokument.tex`** *a napište v LATEXu vlastní pojednání o nějakém svém oblíbeném vědci, fyzikálním (případně matematickém) teorému či rovnici. Dokument by měl obsahovat aspoň jednu rovnici a nejlépe i obrázek. Textová část stačí na jednu stránku.*

[4] AMS = American Mathematical Society.

5.3 Další návody a odkazy

- Ne příliš stručný úvod do sys1tému LaTeX 2_ε: Vynikající srozumitelný, přehledný a čtivý návod v češtině. Doporučuji prostudovat.
- Jak na LaTeX: Webový seriál, obsahuje i určité pokročilejší vychytávky.
- The comprehensive LaTeX Symbol List: Několikasetstránkový dokument se všemi možnými použitelnými matematickými i nematematickými symboly.
- LaTeX: Asi nejpodrobnější příručka dostupná na webu.

6 Histogram

V tomto cvičení se vrátíme k náhodným číslům, se kterými jsme se již setkali při programování náhodné procházky 3. Nyní se podíváme hlouběji na jejich vlastnosti, naučíme se zobrazit hustotu pravděpodobnosti jejich rozdělení (histogram), vytvoříme triviální generátor čísel z Gaussovského normálního rozdělení a generátor čísel z libovolného rozdělení zadaného hustotou pravděpodobnosti nebo distribuční funkcí.

Histogram je jeden z klíčových objektů v mnoha oblastech fyziky, kde se pracuje s náhodnými veličinami. To je vpodstatě celá kvantová mechanika, a tudíž obory jako je atomová, jaderná, či subjaderná fyzika. S náhodnými veličinami se setkáte samozřejmě také v klasické statistické fyzice, ale také například v meteorologii či dalších oborech. Stojí proto za to se s ním seznámit podrobně.

6.1 Základní definice a tvrzení z teorie pravděpodobnosti

V následujícím textu budeme značit X spojitou náhodnou veličinu[5] s hodnotami v intervalu $x \in \langle a, b\rangle$[6]. Důležité pojmy a vztahy pro nás budou:

- **Hustota pravděpodobnosti $\rho(x)$:** Pravděpodobnost, že náhodná veličina X bude nabývat hodnoty z intervalu $\langle x_1, x_2\rangle \subset \langle a, b\rangle$, je

$$\Pr\left[x_1 \leq X \leq x_2\right] = \int_{x_1}^{x_2} \rho(x)\mathrm{d}x. \tag{38}$$

  Hustota pravděpodobnosti je normalizovaná na definičním oboru,

$$\int_a^b \rho(x)\mathrm{d}x = 1 \tag{39}$$

  (pravděpodobnost, že bude náhodná veličina nabývat libovolné ze svých povolených hodnot, je 1 = jistý jev). Hustotu pravděpodobnosti lze vždy rozšířit na celou množinu $\mathbb{R}$, pokud dodefinujeme $\rho(x) = 0$ pro $x < a$ a $x > b$.

- **Distribuční funkce (kumulovaná hustota pravděpodobnosti) $F(x)$:** Neklesající spojitá funkce s oborem hodnot $\langle 0, 1\rangle$ daná integrálem hustoty pravděpodobnosti[7]

$$F(x) = \int_a^x \rho(x')\mathrm{d}x'. \tag{41}$$

[5] V teorii pravděpodobnosti se náhodné veličiny značí obvykle velkým písmenem.

[6] Náhodná veličina X je ve skutečnosti velmi abstraktní objekt. Obecně se definuje na měřitelném prostoru $(\mathcal{X}, \mathcal{A}, \mu)$, kde $\mathcal{X}$ je množina možných hodnot náhodné veličiny X, $\mathcal{A}$ je σ-algebra nad množinou $\mathcal{X}$ (neprázdný systém množin uzavřený na spočetné sjednocení a obsahující prázdnou množinu a množinu $\mathcal{X}$) a μ je míra množiny $\mathcal{M} \subset \mathcal{X}$ (nezáporná σ-aditivní množinová funkce nulová pro prázdnou množinu a jednotková pro celou množinu $\mathcal{X}$). Tato definice v sobě zahrnuje jak náhodné veličiny s diskrétními možnými hodnotami (jako je například hod kostkou), tak náhodné veličiny se spojitými možnými hodnotami, kterým se věnujeme v této sekci.

[7] Nebo obráceně, hustota pravděpodobnosti je derivace distribuční funkce,

$$\rho(x) = \frac{\mathrm{d}F}{\mathrm{d}x}. \tag{40}$$

Platí tedy díky normalizaci (39)

$$F(a) = 0,$$
$$F(b) = 1.$$

Rozšíříme-li obor hodnot náhodné veličiny na všechna reálná čísla stejným způsobem, jako jsme naznačili u hustoty pravděpodobnosti, platí navíc

$$F(x < a) = 0,$$
$$F(x > b) = 1.$$

Pravděpodobnost, že náhodná veličina X bude nabývat hodnoty z intervalu $\langle x_1, x_2 \rangle$, je pak jednoduše

$$\Pr\left[x_1 \leq X \leq x_2\right] = F(x_2) - F(x_1). \tag{42}$$

- **Střední hodnota:**[8]
$$\mathrm{E}\left[X\right] = \int_{-\infty}^{\infty} x\rho(x)\mathrm{d}x. \tag{43}$$

- **Rozptyl:**
$$\sigma_X^2 = \mathrm{E}\left[X^2\right] - \mathrm{E}\left[X\right]^2 = \int_{-\infty}^{\infty} x^2\rho(x)\mathrm{d}x - \left[\int_{\infty}^{\infty} x\rho(x)\mathrm{d}x\right]^2. \tag{44}$$

- **Výběrová střední hodnota:** Pokud máme soubor n hodnot náhodné veličiny X, které označíme $\{x_1, x_2, \ldots, x_n\}$ (výběr), pak výběrová střední hodnota je dána aritmetickým průměrem,
$$\overline{X} = \frac{1}{n}\sum_{j=1}^{n} x_n. \tag{45}$$

  Čím mohutnější máme výběr, tím lépe výběrová střední hodnota aproximuje střední hodnotu,
$$\overline{X} \xrightarrow{n\to\infty} \mathrm{E}\left[X\right]. \tag{46}$$

- **Histogram:** Graf (obvykle sloupcový), který aproximuje distribuční funkci náhodné veličiny X na základě hodnot výběru $\mathcal{V} = \{x_j, j = 1, \ldots, n\}$. Graf se skládá z $N \ll n$ intervalů (sloupců) obvykle konstantní šířky pokrývající obor hodnot náhodné veličiny $\langle a, b \rangle$, přičemž výška sloupce na konkrétním intervalu je rovna počtu hodnot z výběru $\mathcal{V}$, které do intervalu padnou. Pokud histogram správně nanormujeme, získáme (poněkud zubatou) aproximaci distribuční funkce.

- **Nezávislé náhodné veličiny:** Dvě náhodné veličiny X a Y jsou nezávislé, pokud jedna neovlivňuje druhou. Sdružená hustota pravděpodobnosti nezávislých náhodných veličin je dána součinem dílčích hustot pravděpodobnosti,
$$\rho_{X,Y}(x, y) = \rho(x)\rho(y). \tag{47}$$

  Například věk a výška osoby nejsou nezávislé veličiny (pro děti bude rozdělení jejich výšek jiné než pro dospělé), zatímco věk osoby a její krevní skupina nezávislé veličiny jsou.

- **Součet dvou náhodných veličin:** Pokud máme náhodnou veličinu X s hustotou pravděpodobnosti $\rho_X(x)$ a náhodnou veličinu Y s hustotou pravděpodobnosti $\rho_Y(y)$, přičemž obě náhodné veličiny jsou nezávislé, pak náhodná veličina
$$Z = X + Y \tag{48}$$

[8] **E**xpectation value

bude mít hustotu pravděpodobnosti ρ_Z danou *konvolucí* hustot ρ_X a ρ_Y,

$$\rho_Z(z) = \int_{-\infty}^{\infty} \rho_X(\xi)\rho_Y(z-\xi)\mathrm{d}\xi. \tag{49}$$

Střední hodnota a rozptyl náhodné veličny Z jsou dány součtem

$$\begin{aligned} \mathrm{E}[Z] &= \mathrm{E}[X] + \mathrm{E}[Y], \\ \sigma_Z^2 &= \sigma_X^2 + \sigma_Y^2. \end{aligned} \tag{50}$$

- **Centrální limitní věta:** Je-li náhodná veličina Y daná součtem m vzájemně nezávislých náhodných veličin $X^{(1)}, X^{(2)}, \ldots, X^{(m)}$ se shodným rozdělením s hustotou pravděpodobnosti $\rho(x) = \rho_{X^{(j)}}(x)$, jehož střední hodnota je $\mu \equiv \mathrm{E}\left[X^{(j)}\right] < \infty$ a $\sigma^2 \equiv \sigma^2_{X^{(j)}} < \infty$, $j = 1, \ldots, m$, pak

$$Y \sim N(m\mu, m\sigma^2), \tag{51}$$

kde $N(\mu, \sigma^2)$ je Gausovské normální rozdělení se střední hodnotu μ a rozptylem σ^2. Zcela ekvivalentně lze zavést náhodnou veličinu U jako přeškálovanou veličinu Y a psát

$$U \equiv \frac{Y - m\mu}{\sqrt{m\sigma^2}} \xrightarrow{n\to\infty} N(0,1). \tag{52}$$

Hustota pravděpodobnosti normálního rozdělení je dána vzorcem (56).

6.2 Příklady náhodných veličin

- **Rovnoměrné rozdělení $R(a,b)$ na intervalu $\langle a, b\rangle$:**

$$\rho_R(x) = \frac{1}{b-a} = \text{konst.} \tag{53}$$

$$\mathrm{E}[R] = \frac{a+b}{2} \tag{54}$$

$$\sigma_R^2 = \frac{(b-a)^2}{12}. \tag{55}$$

- **Gaussovo normální rozdělení $N(\mu, \sigma^2)$:**

$$\rho_N(x) = \frac{1}{\sqrt{2\pi\sigma^2}}\, \mathrm{e}^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{56}$$

$$\mathrm{E}[N] = \mu \tag{57}$$

$$\sigma_N^2 = \sigma^2. \tag{58}$$

- **Poissonovo rozdělení:** Diskrétní rozdělení udávající počet nezávislých jevů k v zadaném intervalu (například počet lidí, které potkáme na mostě cestou z Holešovic do Troji, nebo počet rozpadů radioaktivního prvku ve vzorku za jednotku času). Rozdělení pravděpodobnosti je

$$P_k = \frac{\lambda^k}{k!}\, \mathrm{e}^{-\lambda}, \tag{59}$$

$$\mathrm{E}[P] = \lambda, \tag{60}$$

$$\sigma_P^2 = \lambda, \tag{61}$$

přičemž parametr λ udává zároveň střední hodnotu a zároveň rozptyl rozdělení.

Úkol 6.1: *Dokažte vztahy (54)–(55) a (60)–(55).*

Řešení 6.1: *K dokázání (54)–(55) vyjdeme z definičních vztahů pro střední hodnotu a rozptyl (43)–(44):*

$$\mathrm{E}\,[R] = \int_a^b \frac{x}{b-a}\mathrm{d}x = \frac{1}{b-a}\left[\frac{x^2}{2}\right]_a^b = \frac{1}{b-a}\frac{b^2-a^2}{2} = \frac{a+b}{2}, \tag{62}$$

$$\begin{aligned}\sigma_R^2 &= \int_a^b \frac{x^2}{b-a}\mathrm{d}x - \left(\frac{a+b}{2}\right)^2 = \frac{1}{b-a}\frac{b^3-a^3}{3} - \frac{(a+b)^2}{4} \\ &= \frac{a^2+ab+b^2}{3} - \frac{a^2+2ab+b^2}{4} = \frac{a^2-2ab+b^2}{12} = \frac{(b-a)^2}{12}.\end{aligned} \tag{63}$$

Poissonovo rozdělení je diskrétní, je tudíž nutné použít diskrétní analogii vztahů (43)–(44) (nahrazení integrálů sumami):

$$\mathrm{E}\,[P] = \sum_{k=0}^{\infty} kP_k = \sum_{k=1}^{\infty} \frac{\lambda^k}{(k-1)!}\mathrm{e}^{-\lambda} = \left| \begin{array}{l} \textit{substituce} \\ l = k-1 \end{array} \right| = \sum_{l=0}^{\infty} \frac{\lambda^{l+1}}{l!}\mathrm{e}^{-\lambda} = \lambda, \tag{64}$$

$$\begin{aligned}\sigma_P^2 &= \sum_{k=0}^{\infty} k^2P_k - \lambda^2 = \sum_{k=1}^{\infty} \frac{k\lambda^k}{(k-1)!}\mathrm{e}^{-\lambda} - \lambda^2 \\ &= \sum_{k=1}^{\infty} \frac{(k-1)\lambda^k}{(k-1)!}\mathrm{e}^{-\lambda} + \sum_{k=1}^{\infty} \frac{\lambda^k}{(k-1)!}\mathrm{e}^{-\lambda} - \lambda^2 \\ &= \sum_{k=2} \frac{\lambda^k}{(k-2)!}\mathrm{e}^{-\lambda} + \lambda - \lambda^2 = \lambda^2 + \lambda - \lambda^2 = \lambda.\end{aligned} \tag{65}$$

Úkol 6.2: *Naprogramujte funkci pro výpočet histogramu: na vstupu bude pole hodnot (výběr z nějakého rozdělení) a počet intervalů histogramu; na výstupu bude pole, jehož každý prvek bude odpovídat jednomu intervalu histogramu a ponese počet hodnot, které do tohoto intervalu padnou ze vstupního pole. Zamyslete se nad co nejefektivnějším algoritmem.*

Výstupní pole funkce vykreslete jako čárový graf.[9]

Řešení 6.2: *Vzorový výpočet histogramu je naprogramován v modulu* `Histogram.py` *ve funkci* `Histogram`*. Klíčové jsou dva řádky:*

```
index = int((d - minValue) / (maxValue - minValue) * numBins)
histogram[index] += 1
```

Hodnotu, kterou chceme zařadit do příslušného okénka histogramu, máme v proměnné `d`*. Z ní spočítáme celočíselný index v intervalu mezi* $0 \leq$ `index` $<$ `numBins` *a pak v poli histogram na příslušném indexu přidáme jedničku. Histogramované pole hodnot* `data` *tedy stačí procházet jen jednou, algoritmus má tudíž časovou náročnost* $\mathcal{O}(N)$*. Nutným předpokladem pro použití tohoto algoritmu je, že všechna okénka histogramu musejí mít stejnou šířku. Funkce* `Histogram` *pak vrací pole s* x*-ovými hodnotami (středy histogramů) a pole s hotovým histogramem. Pro snazší porovnání s hustotami pravděpodobnosti lze navíc nastavit parametr* `normalize=True`*, výsledný histogram pak bude splňovat*

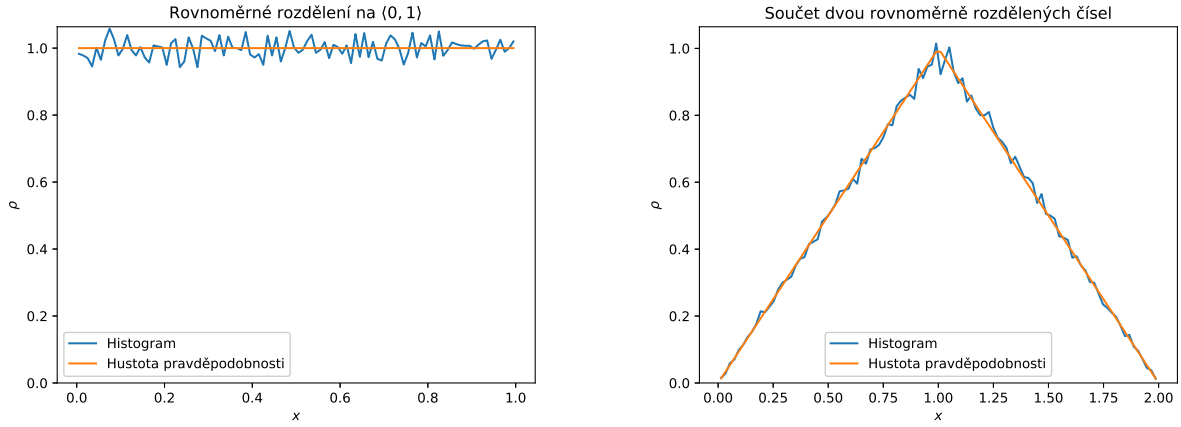
$$\sum_{i=1}^{\texttt{numBins}} h_i w_i = 1, \tag{66}$$

kde h_i *je hodnota histogramu v* i*-tém okénku a* $w_i =$ (`maxValue` $-$ `minValue`)/`numBins` *šířka okénka.*

Příklady histogramů a jejich vykreslení do grafu jsou v řešení následujícího úkolu a v obrázku 7.

Úkol 6.3: *Otestujte funkci z předchozího úkolu na následujících vstupních polích:*

[9] Knihovna `matplotlib` obsahuje funkce na přímé vykreslení sloupcového histogramu.



(a) (b)

Obrázek 7: Srovnání histogramu získaného z $n = 10^5$ hodnot vybraných z daného rozdělení a porovnání s teoretickou hustotou pravděpodobnosti (a) rovnoměrného rozdělení (53) ($a = 0, b = 1$) a (b) součtu dvou rovnoměrných rozdělení (68). Počet intervalů histogramu je v obou případech $N = 100$.

1. *Výběr z rovnoměrného rozdělení na intervalu $\langle 0, 1\rangle$ (v Pythonu generované pomocí* `random()` *z knihovny* **`random`**, *resp. pomocí* `generator.random()` *z knihovny* **`numpy.random`**, *jak je shrnuto v sekci 3.1).*

2. *Výběr ze* **součtu dvou** *rovnoměrných rozdělení na intervalu $\langle 0, 1\rangle$. Hustota pravděpodobnosti výsledného rozdělení je dána konvolucí (49). Vypočítejte analyticky pomocí tohoto vzorce, jak bude hustota pravděpodobnosti vypadat, a porovnejte se získaným histogramem.*

3. *Výběr ze* **součtu** *m rovnoměrných rozdělení na intervalu $\langle 0, 1\rangle$. Přesvědčte se, že již pro celkem malé m platí centrální limitní věta (51) a výsledné rozdělení se blíží normálnímu rozdělení $N(\mu, \sigma)$. Jaká bude střední hodnota μ a rozptyl σ tohoto rozdělení?*

Pro pěkné grafy volte alespoň $n = 10000$ (počet prvků výběru) a $N = 100$ (počet intervalů histogramu).

Řešení 6.3: *Testování funkce* `Histogram` *pro různá rozdělení je v modulu* **`Distributions.py`**.

1. *Hustota pravděpodobnosti rovnoměrného rozdělení je zobrazena funkcí* `Uniform`. *Výsledný graf je na obrázku 7.*

2. *Hustota pravděpodobnosti součtu dvou rovnoměrných rozdělení je podle (49)*

$$\rho_2(z) = \int_0^1 \rho_1(\xi)\rho_1(z-\xi)\mathrm{d}\xi = \int \rho_1(z-\xi)\mathrm{d}\xi, \tag{67}$$

kde $\rho_1(\xi) = 1$ pro $0 \le \xi \le 1$ a $\rho_1(\xi) = 0$ jinde, čímž se v integrálu zbavíme faktoru $\rho_1(\xi)$, neboť je rovný 1 na celém integračním intervalu. Integraci nyní rozdělíme na čtyři intervaly:

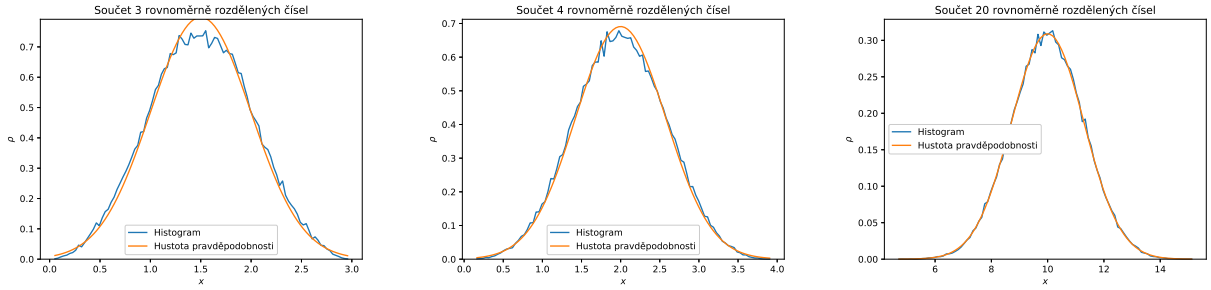
$$\begin{aligned}
z < 0: &\qquad \rho_2(z) = 0,\\
0 < z < 1: &\qquad \rho_2(z) = \int_0^z \mathrm{d}\xi = z,\\
1 < z < 2: &\qquad \rho_2(z) = \int_{z-1}^z \mathrm{d}\xi = 1 - (1-z) = 2 - z,\\
z > 2: &\qquad \rho_2(z) = 0.
\end{aligned}$$

To lze zapsat zjednodušeně jako

$$\rho_2(z) = \begin{cases} 1 - |1-z| & 0 < z < 2,\\ 0 & \textit{jinak.}\end{cases} \tag{68}$$

Rozdělení má trojúhelníkový tvar, což odráží skutečnost, že při součtu dvou rovnoměrně rozdělených čísel na intervalu $\langle 0, 1\rangle$ je mnohem více možností, jak realizovat součet okolo 1, než součet na krajích intervalu (stejně jako při hodu dvěma kostkami je pravděpodobnost součtu 7 větší než pravděpodobnost součtu 12, neboť součet 6 můžeme realizovat šesti způsoby, zatímco součet 12 jen jedním způsobem). Hustota pravděpodobnosti se počítá funkcí `Sum2Uniform` *a výsledný graf je zobrazen na obrázku 7.*

3. *Výpočet hustoty pravděpodobnosti součtu více rovnoměrných rozdělení je naprogramován ve funkci* `SumUniform` *a porovnání s příslušně nanormovanou Gaussovkou a je na obrázku 8.*



(a) (b) (c)

Obrázek 8: Srovnání histogramu získaného z $n = 10^5$ hodnot daných součtem m (a) $m = 3$, (b) $m = 4$ a (c) $m = 20$ rovnoměrně rozdělených náhodných čísel, s Gaussovskou hustotou pravděpodobnosti. Již pro $m = 3$ je shoda velmi dobrá a centrální limitní věta (51) je přibližně splněna. V případě $m = 20$ je již rozdíl od Gaussovského rozdělení prakticky nepozorovatelný. Počet intervalů histogramu je ve všech případech $N = 100$.

Úkol 6.4: *Na základě centrální limitní věty (52) vytvořte jednoduchý generátor čísel s normálním Gaussovským rozdělením $N(0, 1)$. Jaké je optimální hodnota m, abychom získali dostatečně přesnou aproximaci normálního rozdělení, a přitom použili co nejméně algebraických operací?*

Řešení 6.4: *Ideální počet sečtených čísel z rovnoměrného rozdělení pro získání velmi dobré aproximace čísla z normálního Gaussova rozdělení je $m = 12$, neboť pro tuto hodnotu:*

- *Rozptyl výsledného rozdělení je $\sigma = 1$ díky vztahům (52) a (55).*
- *Hodnota nagenerovaného čísla je v intervalu $\langle 0, 12\rangle$, se střední hodnotou 6. Pokud tuto střední hodnotu odečteme dle vzorce (52), obdržíme rozdělení se střední hodnotou 0 a zahrnující interval 6σ, který pokrývá 99.9999998% Gaussovského rozdělení.*

Generátor je naprogramován v modulu `GaussianGenerators`*, funkce* `GaussianGenerator1`*. Srovnání generátoru s odpovídající Gaussovkou je na obrázku 9(a).*

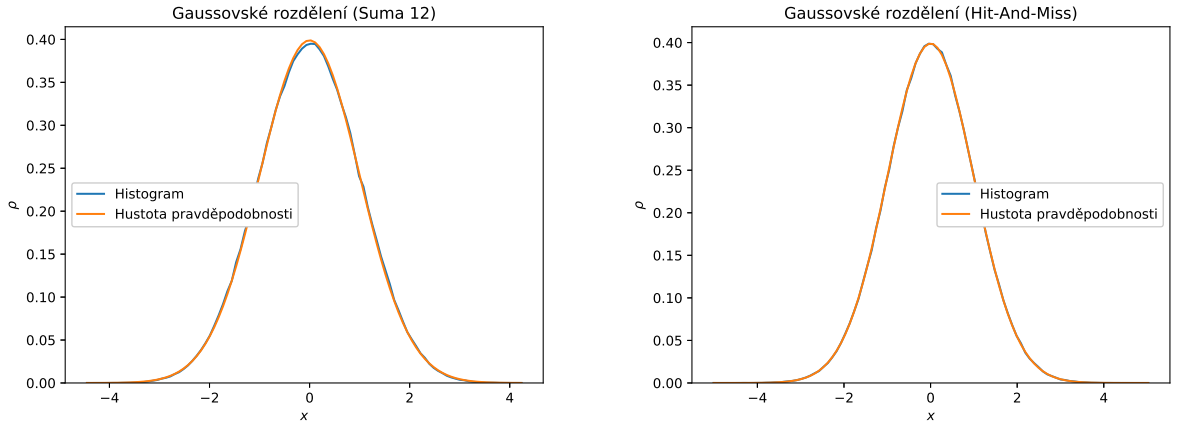
Úkol 6.5: *Nakreslete histogram pro rozdělení hodnot v jednotlivých intervalech histogramů z úlohy 6.3. Jaké očekáváte statistické rozdělení v tomto případě?*

Řešení 6.5: *Počet hodnot v jednotlivých intervalech splňuje předpoklady pro Poissonovo rozdělení, očekáváme tedy Poissonovo rozdělení se parametrem*

$$\lambda = \frac{\texttt{numValues}}{\texttt{numBins}}, \tag{69}$$

který udává zároveň střední hodnotu (60) a zároveň rozptyl (61). Chceme-li toto rozdělení zobrazit, je potřeba jisté delikátnosti.

- *Počty hodnot v okénkách (nenormovaného) histogramu jsou přirozená čísla. Pro zobrazení jejich rozdělení je tedy vhodné volit sekundární histogram se šířkou intervalu 1 (nebo s jinou celočíselnou šířkou).*

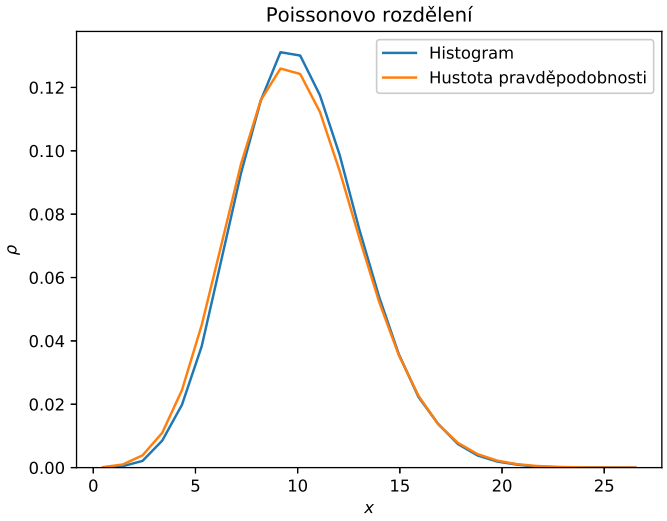


(a) (b)

Obrázek 9: Srovnání jednoduchých generátorů Gaussovsky rozdělených náhodných čísel. (a) Generátor založený na použití Centrální limitní věty (52): Gaussovské náhodné číslo je získáno jako součet $m = 12$ rovnoměrně rozdělených náhodných čísel. (b) Generátor založený na hit-and-miss metodě použité na hustotu pravděpodobnosti Gaussovského rozdělení. V obou případech je pro zobrazovaný histogram použito $n = 10^6$ náhodných čísel a počet intervalů histogramu je $N = 100$. U generátoru (a) je pozorována drobná odchylka v okolí maxima hustoty pravděpodobnosti, u generátoru (b) žádná odchylka pozorována není. Nutno zdůraznit, že generátor (a) je zhruba o řád rychlejší než generátor (b), jak je ukázáno v úloze 6.6.

- *Hezčí rozdělení získáme pro malé parametry λ (pro λ velké se Poissonovo rozdělení blíží Gaussovskému rozdělení). V kódu volím hodnoty tak, aby $\lambda = 10$.*

Srovnání příslušného histogramu s teoretickým rozdělením (59) je vypočteno funkcí `Poisson` *ze souboru* `Distributions.py` *a je vykresleno na obrázku 10.*



Obrázek 10: Poissonovo rozdělení, získané z fluktuací počtu hodnot histogramu s $N = 10^5$ intervaly a $n = 10^6$ rovnoměrně rozdělenými vstupními čísly. Střední hodnota a rozptyl takto získaného Poissonova rozdělení je $\lambda = 10$.

6.3 Výběr z neznámého rozdělení

V praxi se můžeme setkat se situací, kdy máme zadanou hustotu pravděpodobnosti či distribuční funkci nějakého komplikovaného rozdělení a chceme generovat výběr z tohoto rozdělení.

- **Známe-li hustotu pravděpodobnosti rozdělení $\rho(x)$:** Nejjednodušší metoda v tomto případě je vepsat funkci $\rho(x)$ do obdélníku $\langle a, b\rangle \times \langle c, d\rangle$[10], nagenerovat číslo rovnoměrně z tohoto obdélníku, a pokud padne pod křivku $\rho(x)$, vezmeme ho, v opačném případě ho zahodíme (tzv. *hit-and-miss metoda*, se kterou se potkáme v budoucnu u metody Monte-Carlo).

- **Známe-li distribuční funkcí $F(x)$:** V tomto případě využijeme skutečnosti, že obor hodnot distribuční funkce je $F(x) \in \langle 0, 1\rangle$. Stačí tedy generovat náhodné číslo y z rovnoměrného rozdělení na intervalu $\langle 0, 1\rangle$ a číslo
$$x = F^{-1}(y), \tag{70}$$
kde F^{-1} je inverzní funkce k F, bude z rozdělení s danou F. Pokud neznáme inverzní funkci, řešíme numericky rovnici
$$F(x) = y, \tag{71}$$
která však s ohledem na vlastnosti distribuční funkce popsané v sekci 6.1 má téměř vždycky jedno a pouze jedno řešení.

  Matematičtěji zapsáno: je-li $R = R(0, 1)$ náhodná veličina s rovnoměrným rodělením na intervalu $\langle 0, 1\rangle$, pak
$$X = F^{-1}(R) \tag{72}$$
je náhodná veličina s rozdělením daným distribuční funkcí F.

Úkol 6.6: *Pomocí právě popsané metody vytvořte generátor čísel s Gaussovským normálním rozdělením daným hustotou pravděpodobnosti* (56)*. Porovnejte jeho rychlost s generátorem založeným na centrální limitní větě, který jste naprogramovali v úloze 6.4 a s generátorem z některé z knihoven.*[11]

Řešení 6.6: *Generátor je naprogramován v modulu* `GaussianGenerators`*, funkce* `GaussianGenerator2`*, a porovnání histogramu takto nagenerovaných náhodných hodnot s teoretickou hustotou pravděpodobnosti je na obrázku 9(b). Nagenerování* 10^6 *Gaussovsky rozdělených náhodných čísel trvá na mém PC*[12]

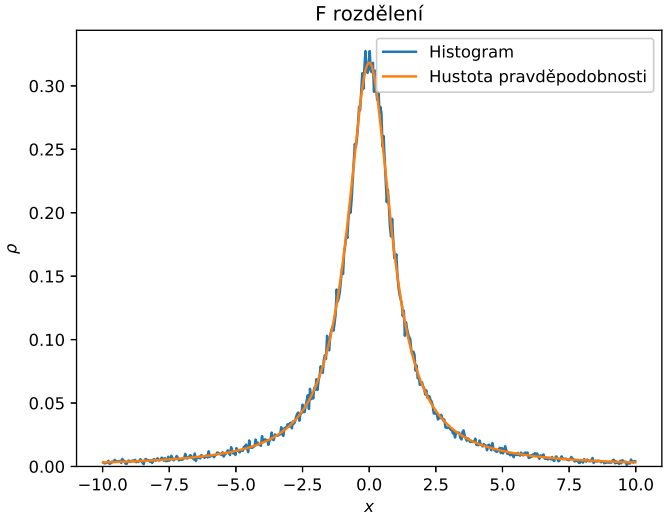
- **4s***: funkce* `generator.normal()` *z knihovny* `numpy`*,*
- **16s***: součet 12 rovnoměrně rozdělených čísel,*
- **111s***: hit-and-miss metoda.*

Úkol 6.7: *Vytvořte generátor čísel z rozdělení daném distribuční funkcí*
$$F(x) = \frac{1}{2}\left(1 + \frac{2}{\pi}\arctan x\right). \tag{73}$$
Jak vypadá analyticky hustota pravděpodobnosti? Nakreslete histogram a porovnejte.

[10] Hustota pravděpodobnosti bývá obvykle definovaná na neomezeném intervalu. Pak je nutné určitou část funkce $\rho(x)$ oříznout. To funguje dobře v případě náhodných veličin s rychle ubývající hustotou pravděpodobnosti, jako je například Gaussovo rozdělení $N(0, 1)$, kde při oříznutí $x \in \langle -3\sigma, 3\sigma\rangle = \langle -3, 3\rangle$ zanedbáme jen 3‰ možných hodnot. V případě dlouhodosahových rozdělení, jako je například lognormální rozdělení nebo Gamma rozdělení, musíme obdélník volit velmi dlouhý, čímž se tato metoda stává velmi neefektivní.

[11] Porovnání můžete provést tak, že nagenergujete větší množství čísel různými metodami, například $n = 10^6$, a změříte dobu výpočtu.

[12] Pokud k nagenerování n hodnot použijeme knihovní funkci s počtem v argumentu, `generator.normal(1000000)`, výpočet trvá zlomek vteřiny. Velká část výpočetního času je tudíž způsobena voláním funkcí a prováděním cyklu.



Obrázek 11: Cauchyho (Breit-Wignerovo) rozdělení nagenerované z distribuční funkce (73) a porovnané s teoretickou hustotou pravděpodobnosti (74). Počet hodnot pro histogram je $n = 10^5$, počet intervalů $N = 500$.

Řešení 6.7: *Hustota pravděpodobnosti je dána derivací distribuční funkce podle vztahu (40), tj.*

$$\rho(x) = \frac{\mathrm{d}F}{\mathrm{d}x} = \frac{1}{\pi}\frac{\mathrm{d}}{\mathrm{d}x}\arctan x = \frac{1}{\pi}\frac{1}{1+x^2}. \tag{74}$$

Toto rozdělní se nazývá Cauchyho *či v kvantové fyzice* Breit-Wignerovo *a tam modeluje šířku energetických hladin exponenciálně se rozpadajících systémů.*

Ke generování čísel z tohoto rozdělení využijeme vztahu (70). Musíme tedy určit funkci inverzní k distribuční funkci (73), která je

$$F^{-1}(y) = \tan\left[\frac{\pi}{2}(2y-1)\right], \tag{75}$$

a za y dosazovat čísla z rovnoměrného rozdělení na intervalu $\langle 0, 1\rangle$.

Histogram čísel nagenerovaných z Cauchyho rozdělení a srovnání s hustotou pravděpodobnosti (74) je počítán funkcí `Poisson` *z modulu* `Distribution.py` *a je zobrazen na obrázku 11. Rozdělení má dlouhý dosah, s rostoucím x klesá jen polynomiálně k nule, pravděpodobnost nagenerování hodnoty daleko od maxima je tudíž velká. V obrázku proto zobrazuji jen okno $x \in \langle -10, 10\rangle$.*

7 Monte-Carlo metoda

Pod Monte-Carlo metodou se rozumí, že namísto systematického (a obvykle zdlouhavého) procházení nějakého parametrického prostoru využíváme náhodně generované body a hledané vlastnosti našeho systému určíme statisticky.[13] V tomto cvičení zúročíme veškeré dosavadní zkušenosti s náhodnými čísly a budeme ze zabývat zejména integrací Monte-Carlo.

Možná jste se již setkali se problémem tzv. Buffonovy jehly: ppomocí náhodného házení jehly (či jakékoliv tyčky) na síť rovnoběžných čar nakreslenou na zemi lze určit číslo π,

$$\pi \approx \frac{2l}{h}\frac{N_{\text{zásah}}}{N_{\text{celkem}}}, \tag{76}$$

kde h je vzdálenost čar, $l \leq h$ délka jehly, N_{celkem} celkový počet hodů a $N_{\text{zásah}}$ počet hodů, při kterých jehla po dopadu kříží nějakou z čar. Buffonova jehla je názorné experimentální použití

[13] Monte Carlo je oblast Monaka, ve kterém se nacházely a doposud nacházejí slavná kasina. Odtud název.

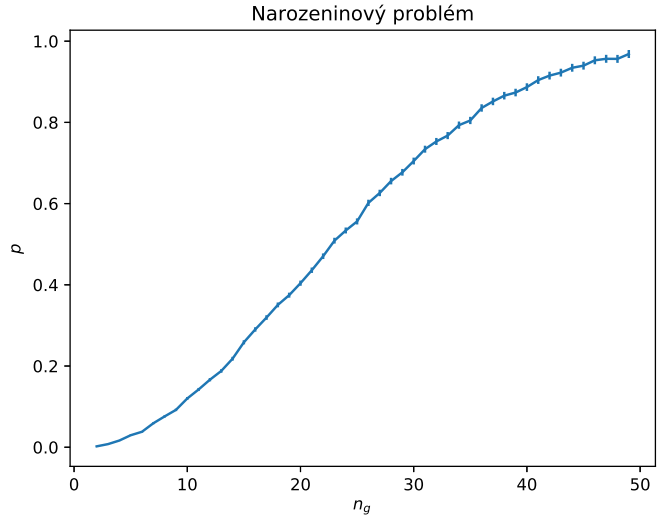
metody Monte-Carlo, konkrétně varianty nazývané hit-and-miss. Té jsme se již dotkli v sekci 6.3 a nyní si rozebereme podrobněji.

Úkol 7.1: *Metodou Monte-Carlo vyřešte tzv. narozeninový problém: Uvažujte skupinu n lidí. Jaká je pravděpodobnost, že dva lidi ve skupině budou mít narozeniny ve stejný den? Úloha se samozřejmě dá vyřešit exaktně, ale zkuste si úlohu vyřešit metodou Monte-Carlo: Pokud nagenerujete náhodně N_{celkem}-krát narozeniny n lidí a označíte $N_{zásah}$ případy, kdy alespoň dvoje narozeniny padnou na stejný den, bude podle zákona velkých čísel hledaná pravděpodobnost rovna*

$$p \approx \frac{N_{zásah}}{N_{celkem}} \tag{77}$$

(rovnost by nastala pro $N_{celkem} \to \infty$). Naprogramujte tuto úlohu a určete,

1. *jaká je pravděpodobnost pro skupinu 30 lidí a*
2. *jak velkou skupinu potřebujete, aby byla pravděpodobnost alespoň 30%.*



Obrázek 12: Pravděpodobnost, že ve skupině n_g lidí budou mít alespoň dva lidé narozeniny ve stejný den v roce. Svislými čarami je odhad chyby $\pm 1\sigma$. Počet pokusů pro metodu Monte-Carlo je $N_{\text{celkem}} = 10^4$.

Řešení 7.1: *Vzorové řešení je v souboru* `BirthdayProblem.py` *a obsahuje dvě funkce:*

- `BirthdayCoincidenceProbability` *určí metodou Monte-Carlo pravděpodobnost, že ve skupině o velikosti $n_g =$* `groupSize` *bude alespoň jedna dvojice, která slaví narozeniny ve stejný den v roce. Řešení probíhá ve dvou krocích:*
  1. *nagenergujeme řadu n_g čísel mezi 1 a 366, které udávají den narozenin,*[14]
  2. *zkontrolujeme, zda řada obsahuje dva stejné prvky (ve vzorovém řešení tak, že řadu seřadíme a zkontrolujeme, jestli obsahuje dvě či více stejných čísel v bezprostředně následujících prvcích).*

  Funkce vrací hodnotu pravděpodobnosti p a odhad chyby $\Delta p \approx 1\sigma_p$.
- `PlotBirthdayProblem` *vykreslí graf pravděpodobností pro interval velikostí skupin. Graf včetně chyb je v obrázku 12.*

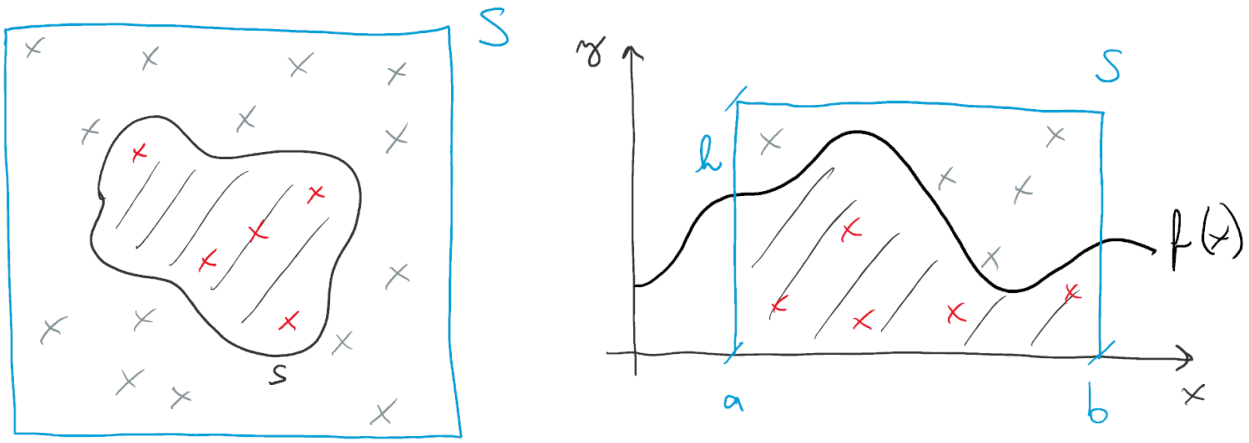
[14] Řešení počítá i s datem narození 29. února, přičemž předpokládá, že pravděpodobnost všech dat narození je stejná. To však pro narozeniny 29. února neplatí, pravděpodobnost narození s tímto datem je menší.

Na základě těchto funkcí se jednoduše spočítá, že

1. *Pro $n_g = 30$ je pravděpodobnost $p \approx 70\%$.*
2. *Pro $n_g = 17$ je pravděpodobnost $p \approx 31\%$.*

Program lze triviálně rozšířit a počítat například pravděpodobnost, že

- *alespoň d lidí bude mít narozeniny ve stejný den,*
- *alespoň 2 lidé budou mít narozeniny ve stejný den i rok.*



(a) (b)

Obrázek 13: Metoda hit-and-miss (a) pro výpočet plochy složitého obrazce a (b) pro výpočet integrálu jednorozměrné funkce. Podrobnosti k obrázku jsou uvedeny v textu.

7.1 Hit-And-Miss

Tato metoda spočívá v jednoduché aplikaci zákona velkých čísel. Její esence je načrtnuta na obrázku 13. Uvažujme nejprve, že chceme změřit plochu s černého obrazce složitého tvaru znázorněného na panelu (a). Obrazec vepíšeme do jiného jednoduchého obrazce, jehož plochu S známe (nejčastěji obdélník, případně kruh). Poté do tohoto obrazce S náhodně „házíme“ body (křížky) a počítáme, kolikrát se trefíme do černého obrazce (červené křížky). Neznámá hledaná plocha s je pak při již použitém značení

$$s \approx S \frac{N_{\text{zásah}}}{N_{\text{celkem}}}. \tag{78}$$

Analogicky postupujeme při integrování, což je ukázáno na panelu (b). Integrál v mezích (a, b) je plocha pod křivkou funkce $f(x)$ (na obrázku černě vyšrafovaná plocha ohraničená zespodu osou x, shora funkcí $f(x)$ a ze stran modrými čarami $x = a$ a $x = b$). Uvedenou oblast vepíšeme do obdélníku o hranách $l = b - a$ a h s plochou $S = (b - a)h$ a stejnou metodou jako u panelu (a) a pomocí stejného vzorce jako je (78) vypočítáme hodnotu integrálu:

$$\int_a^b f(x)\mathrm{d}x \approx S \frac{N_{\text{zásah}}}{N_{\text{celkem}}} = (b-a)h \frac{N_{\text{zásah}}}{N_{\text{celkem}}}. \tag{79}$$

7.1.1 Chyba

Počet zásahů je vlastně počet nezávislých „hodů“, kterými se trefíme do oblasti, jejíž plochu S hledáme. Pokud budeme opakovat metodu hit-and-miss s fixním N_{celkem}, bude mít náhodná veličina $N_{\text{zásah}}$ Poissonovo rozdělení se střední hodnotou (60) a rozptylem (61)

$$\lambda = \mathrm{E}\left[N_{\text{zásah}}\right] = \sigma^2_{N_{\text{zásah}}}. \tag{80}$$

Budeme-li mít jen jednu realizaci s dostatkem zásahů, můžeme střední hodnotu odhadnout pomocí této realizace, $\mathrm{E}\left[N_{\text{zásah}}\right] \approx N_{\text{zásah}}$ a absolutní chybu odhadneme směrodatnou odchylkou

$$\Delta N_{\text{zásah}} = \sqrt{\sigma^2_{N_{\text{zásah}}}} \approx \sqrt{N_{\text{zásah}}}. \tag{81}$$

Relativní chyba pak je

$$\delta N_{\text{zásah}} = \frac{\Delta N_{\text{zásah}}}{N_{\text{zásah}}} \approx \frac{1}{\sqrt{N_{\text{zásah}}}}. \tag{82}$$

Tento vzorec platí i pro relativní chybu ve výpočtu plochy (78) či integrálu (79).

Jelikož $N_{\text{zásah}} \propto N_{\text{celkem}}$, z uvedených úvah vyplývá, že

- relativní chyba klesá jako převrácená hodnota odmocniny celkového počtu pokusů N_{celkem} a
- chceme-li zpřesnit výsledek získaný touto metodou desetkrát, musíme zestonásobit počet pokusů.

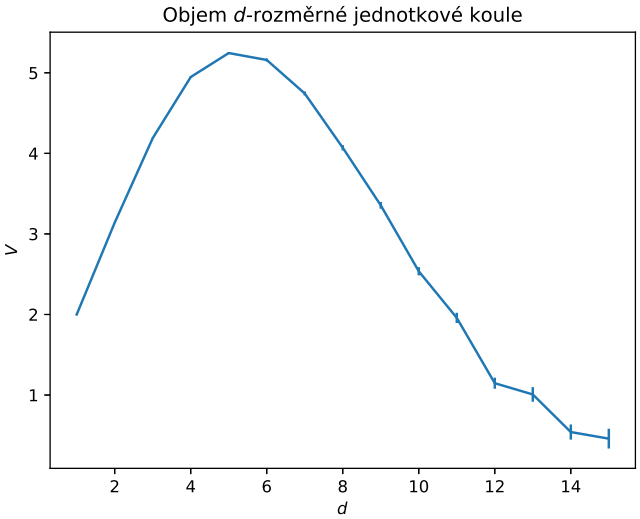
V praxi za použití běžných výpočetních prostředků lze dosáhnout nanejvýš $N_{\text{celkem}} \approx 10^{10}$, což dá výsledek s relativní chybou minimálně $\delta \approx 10^{-5}$, tj. pět desetinných míst.

7.1.2 Použití

Integrace pomocí metody hit-and-miss se nepoužívá, jelikož je příliš neefektivní. K jejímu úspěšnému použití totiž musíme znát maximum funkce na zadaném intervalu, abychom efektivně určili výšku obdélníku h, a obecně je $N_{\text{zásah}}/N_{\text{celkem}}$ velmi malé číslo (funkce mají dlouhý chvost nebo jsou příliš vysoké), většina „hodů“ jde tedy mimo a i při jejich velkém množství získáme málo zásahů, a tudíž velkou chybu podle vztahu (82).

Na druhou stranu se tato metoda hodí k výpočtu povrchů, objemů či hyperobjemů v případě vícerozměrných objektů.

Úkol 7.2: *Vytvořte program na výpočet objemu d-rozměrné jednotkové koule metodou Monte-Carlo.*[15] *Pro jakou dimenzi bude tento objem největší číslo?*



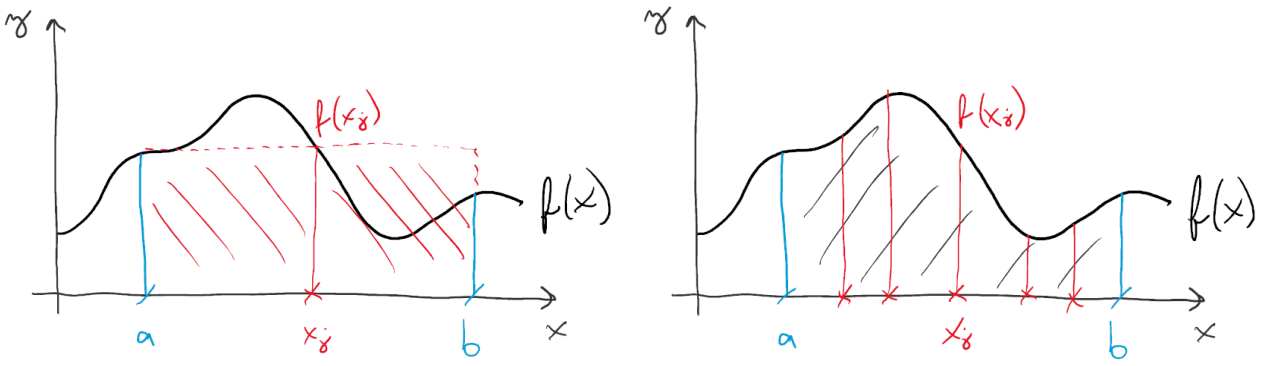
Obrázek 14: Závislost objemu jednotkové koule na dimenzionalitě prostoru. Svislými čarami je zobrazena chyba $\Delta V \approx 1\sigma_V$. Počet pokusů pro metodu Monte-Carlo je pro každý bod $N_{\text{celkem}} = 10^6$.

[15] Pod 1-rozměrnou jednotkovou koulí rozumíme úsečku délky 2, pod 2-rozměrnou jednotkovou koulí jednotkový kruh. Povrch jednotkového kruhu je $S = \pi$, výsledek lze tedy použít i k určení čísla π, aniž byste museli házet Buffonovou jehlou.

Řešení 7.2: *Vzorový kód je v souboru* `VolumeBall.py`. *Funkce* `VolumeBall` *spočítá objem jednotkové koule metodou Monte-Carlo a vrátí současně odhad chyby. Funkce* `PlotVolumes` *vykreslí graf objemů v závislosti na dimenzi, jak je ukázáno na obrázku 14. Největší objem $V \approx 5.27$ dostaneme pro $d = 5$. Zde je na místě jistá opatrnost: srovnáváme trochu „jablka s hruškami“, protože samozřejmě objem různěrozměrných koulí má různou dimenzionalitu. Pro jiné než jednotkové koule tato závislost platit nebude.*

Z obrázku je rovněž vidět, že s rostoucím d roste chyba výsledku. To je způsobeno tím, že s rostoucí dimenzionalitou se při metodě hit-and-miss čím dál častěji trefujeme mimo jednotkovou kouli, jinými slovy klesá při zadaném počtu pokusů počet zásahů.[16] *Tento jev a jeho příčiny jsme již diskutovali v případě generování náhodného směru v d dimenzích v řešení úlohy 3.2.*

7.2 Monte-Carlo integrace



(a) (b)

Obrázek 15: Monte-Carlo integrace. Vysvětlení je v hlavním textu.

Pro hledání integrálu je mnohem efektivnější metoda, již lze vysvětlit pomocí obrázku 15. Uvažujme neprve, že známe pár hodnot $(x_j, f(x_j))$, nic víc, nic míň. Na základě těchto hodnot můžeme učinit pouze velmi hrubý odhad integrálu, a to jako plochu červeně vyšrafovaného obdélníku jako na panelu (a),

$$\int_a^b f(x)\mathrm{d}x \approx (b-a)f(x_j). \tag{83}$$

Pokud budeme mít párů hodnot $(x_j, f(x_j))$ více, jak je znázorněno na panelu (b), vezmeme za odhad hodnoty integrálu průměr ploch takovýchto obdélníků,

$$\int_a^b f(x)\mathrm{d}x \approx \frac{1}{N}\sum_{j=1}^{N}(b-a)f(x_j) = \frac{b-a}{N}\sum_{j=1}^{N} f(x_j). \tag{84}$$

V právě uvedeném postupu tkví je podstata integrace Monte-Carlo. Obecně platí, že pokud hodnoty x_j vybíráme z rozdělení s hustotou pravděpodobnosti $\rho(x)$, je integrál odhadnutý výrazem

$$\int_a^b f(x)\mathrm{d}x \approx \frac{1}{N}\sum_{j=1}^{N}\frac{f(x_j)}{\rho(x_j)}, \tag{85}$$

přičemž nejvhodnější je volit takové pravděpodobnostní rozdělení, jehož hustota pravděpodobnosti co nejlépe kopíruje integrovanou funkci.[17] V praxi, jelikož na funkci nejčastěji pohlížíme jako na „černou skříňku“ a o jejím průběhu nic nevíme, se jako nevhodnější jeví volit rovnoměrné rozdělení s hustotou pravděpodobnosti (53), která po dosazení dá předchozí vzorec (84).

[16] Pro $d = 15$ je při $N_{\text{celkem}} = 10^6$ počet zásahů pouze řádově 10^1, tj. z každých sto tisíc pokusů se jen jednou trefíme do jednotkové koule, zbytek pokusů padá do „rohů“ opsaných krychlí.

[17] Tento postup se nazývá *importance sampling*.

Chybu metody integrace lze odhadnout pomocí směrodatné odchylky

$$\Delta \equiv \sigma = \sqrt{\overline{f^2} - \overline{f}^2}, \tag{86}$$
$$\overline{f^2} = \frac{1}{N}\sum_{j=1}^{N} f^2(x_j),$$
$$\overline{f}^2 = \left[\frac{1}{N}\sum_{j=1}^{N} f(x_j)\right]^2.$$

Úkol 7.3: *Metodou Monte-Carlo spočítejte integrály*

$$I_1 \equiv \int_0^{2\pi} \mathrm{e}^{-x} \sin x \,\mathrm{d}x, \tag{87}$$
$$I_2 \equiv \int_0^{\sqrt{10\pi}} \frac{\sin x^2}{\sqrt{1+x^4}} \mathrm{d}x. \tag{88}$$

První integrál má analytické vyjádření, které si můžete odvodit a porovnat s hodnotou získanou Monte-Carlo integrací; druhý integrál lze spočítat pouze numericky. Metodu můžete otestovat i na jiných známých integrálech.

Řešení 7.3: *Řešení je naprogramováno v souboru* `Itegration.py` *dvěma způsoby:*

1. `Integrate1D` *integruje postupným generováním* $N =$ `n` *náhodných bodů a výpočtem pomocí rovnice* (84).
2. `Integrate1DArray` *nageneruje řadu* N *bodů z intervalu* $\langle a, b\rangle$*, pro tuto řadu spočítá řadu funkčních hodnot a tu pak sečte a vynásobí příslušným prefaktorem, rovněž dle rovnice* (84)*. Tento postup je výrazně rychlejší, neboť obsluha cyklů a volání funkcí pro individuální body stojí v interpetovaných jazycích (jímž Python je) velké množství výpočetního času. Na druhou stranu tento postup vyžaduje mít v operační paměti uložená dlouhá pole bodů* x_j *a* $f(x_j)$ *a pro velké počty* N *nám dostupná paměť nemusí stačit. Řešením je samozřejmě oba postupy zkombinovat.*

Výsledky integrálů jsou

$$I_1 = \frac{1}{2}\left(1 - \mathrm{e}^{-2\pi}\right) \approx 0.499, \tag{89}$$
$$I_2 \approx 0.765. \tag{90}$$

Síla metody Monte-Carlo se naplno projeví při výpočtu vícerozměrných integrálů. Jak bylo ukázáno, chyba metody závisí jen na celkovému počtu pokusů $N = N_{\mathrm{celkem}}$. Zatímco u jiných metod při požadování určité dané přesnosti výsledku drasticky narůstá časová složitost integrace s rostoucí dimenzí integrované funkce, u Monte-Carla časová složitost na dimenzi závisí jen nepatrně. Ve více rozměrech bývá navíc integrační oblast složitější, k čemuž lze využít metodu hit-and-miss. To se nejlépe ukáže na příkladu.

Úkol 7.4: *Spočítejte čtyřrozměrný integrál*

$$I_3 \equiv \int_\Omega \sin\sqrt{\ln(x+y+z+w+2)}\,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z\,\mathrm{d}w, \tag{91}$$

kde integrační oblast je hyperkoule

$$\Omega : \left(x-\frac{1}{2}\right)^2 + \left(y-\frac{1}{2}\right)^2 + \left(z-\frac{1}{2}\right)^2 + \left(w-\frac{1}{2}\right)^2 \le \frac{1}{4}. \tag{92}$$

Při výpočtu postupujte tak, že nejprve danou hyperkouli vepíšete do hyperkvádru (či hyperkrychle) známých rozměrů, a tudíž známého objemu V*. Poté pro náhodně zvolený bod v hyperkvádru určíte,*

zda se trefí do hyperkoule Ω či nikoliv. Pokud ano, spočítáte funkční hodnotu integrandu v tomto bodě. Jedná se tedy o kombinaci integrace (84) a metody hit-and-miss. Budete-li si uchovávat počet hodů N_{celkem} a počet zásahů $N_{zásah}$, získáte jako vedlejší produkt objem integrační hyperoblasti pomocí vzorce (78).

Řešení 7.4: *Řešení pro tento konkrétní integrál je naprogramováno v souboru* `Integration.py` *ve funkci* `Integral2`. *Pro $N_{celkem} = 10^6$ tento integrál vychází přibližně*

$$I_3 \approx 0.921. \tag{93}$$

8 Paralelizace

Rychlost výpočtu lze v zvyšovat dvěma základními způsoby: zvyšováním rychlosti procesoru a zvětšováním počtu procesorů. Zatímco první způsob již narazil na fyzikální limity a kupředu postupuje jen zvolna,[18] druhý způsob lze použít takřka neomezeně. V dnešní době mají procesory osobních počítačů, ale i mobilních telefonů či dalších zařízení běžně dva až čtyři plnocenné podprocesory nazývané jádra, přičemž některé procesory navíc umožňují na každém jádře spustit dvě vlákna (technologie *hyperthreading*[19]). To znamená, že můžeme na procesoru spustit několik výpočtů najednou a výpočty budou probíhat paralelně.

Jeden probíhající výpočet se běžně nazývá *vlákno* (thread) nebo *proces*.[20] V moderních operačních systémech je počet současně běžících procesů neomezený. Pokud je procesů více, než dostupných procesorů, operační systém velmi rychle přepíná mezi jejich prováděním, a tak se zdá, že všechny procesy probíhají zároveň (tzv. *preemptivní multitasking*). Přepínání samozřejmě stojí určitý výpočetní čas. Optimální situace tedy je, pokud se počet procesů přesně rovná počtu procesorových jader. Pak jsou všechna jádra vytížena, a přitom operační systém není zatěžován nutností přepínat mezi jednotlivými vlákny. Toto je demonstrováno na obrázku 16.

Nejtriviálnější způsob paralelizace spočívá v tom, že spustíme nezávisle více programů najednou. Operační systém automaticky využije veškerá dostupná procesorová jádra. My jen počkáme na výsledky a ty pak zpracujeme. Tento způsob vpodstatě vylučuje jakoukoliv komunikaci mezi jednotlivými vlákny, a úlohy proto musejí být zcela nezávislé. Z vnějšku lze také jen velmi omezeně řídit provádění vláken, například spustit nový výpočet po doběhnutí konkrétního vlákna.

Pokročilejší postup je ten, kdy danou úlohu rozsekáme na nezávislé samostatné úseky, ty pak z našeho programu (tzv. *hlavního vlákna*) pošleme ke zpracování na více procesorů, počkáme na výsledky a ty následně zpracujeme. Takto lze jednoduše paralelizovat metodu Monte-Carlo: výpočet N Monte-Carlo bodů spustíme současně v p vláknech, a poté všechny výpočty spojíme dohromady tak, že jednoduše spočítáme aritmetický průměr výsledků jednotlivých vláken, přičemž tato hodnota odpovídá jednomu Monte-Carlo výpočtu s pN body. Jednotlivá vlákna výpočtu se sebou nemusejí nijak komunikovat, nemusejí sdílet žádnou část paměti, není tudíž potřeba řešit jejich vzájemnou sychronizaci (v anglické terminologii se pro takovýto typ problémů používá označení „embarassingly parallel“, trapně paralelní).[21]

[18] Maximální dosažitelná rychlost procesorů je dnes přibližně 5 GHz, tj. řádově miliardy strojových cyklů za vteřinu, a během posledních let se nemění. Strojová instrukce většinou trvá několik cyklů a jejich provádění navíc zpomaluje přístup do operační paměti, proto dnešní procesory zvládnou řádově nanejvýš stovky milionů instrukcí za vteřinu. Ke zrychlení procesoru se využívají nejrůznější sofistikované metody. Procesor například odhaduje, kam se program vydá, a instrukce se snaží předpočítat dopředu. Pokud se v odhadu trefí, dojde ke zrychlení. Jiný způsob je vytváření nových strojových instrukcí, které zrychlují určitý často používaný typ úloh (například instrukce rychlá Fourierova transformace, která se intenzivně používá při dekódování videa a při práci s obrázky).

[19] Technologie spočívá v tom, že v každém procesorovém jádře jsou různé výpočetní jednotky, které zpracovávají různé typy procesorových instrukcí. Zatímco se tedy v jednom vlákně násobí dvě čísla s desetinnou čárkou, v jiném vlákně na tomtéž jádře se může zpracovávat například cyklus přes celočíselný index.

[20] Ve skutečnosti je to složitější, jeden proces může obsahovat i více vláken, ale v těchto zápiscích budu předpokládat, že v každém procesu běží jen jedno vlákno, a tudíž budu obě označení brát jako synonyma.

[21] Ve striktně funkcionálním programování není dovoleno funkcím měnit hodnoty proměnných, a proto lze čistě funkcionální programy triviálně paralelizovat.

V úplně obecném případě je nutné řešit vzájemnou sychronizaci vláken, což přesahuje rámec tohoto kurzu. Pro ilustraci uvedu jen jeden z nejzákladnějších příkladů: Představte si, že dvě vlákna sdílejí některé proměnné, a tedy přistupují do stejné části paměti. Pak je nutné zabránit tomu, aby obě vlákna k jedné proměnné přistupovala zároveň (například jedno z proměnné četlo, zatímco druhé do ni zapisovalo). To by totiž vedlo k nejednoznačnému výsledku, protože nelze a priori říci, které vlákno bude operaci provádět dříve. K vyřešení kolize se používají tzv. *zámky* (lock). Než vlákno přistoupí ke sdílené proměnné, zamkne si ji pro sebe, provede svoji operaci a poté proměnnou odemkne. Pokud by mezitím k zamčené proměnné chtělo přistoupit jiné vlákno, jeho provádění se zastaví a vlákno čeká, než bude proměnná odemčena. Z toho ihned vyplývá, že při velkém počtu sdílených proměnných či při častém přístupu k nim dochází k významnému zpomalení paralelního zpracování, neboť velkou část výpočetního času vlákna čekají na přístup k momentálně zamčeným proměnným.

Zamykání proměnných s sebou nese problémy a potenciální obtížně odhalitelné chyby. Může se stát, že zapomeneme proměnnou odemknout, čímž zastavíme všechna ostatní vlákna, která chtějí k proměnné přistupovat. K zablokování programu může dojít i tak, že provádění vlákna skončí chybou ve chvíli, kdy je nějaká proměnná zamčená.

Další možný způsob zablokování paralelního výpočtu kvůli zámkům je tzv. *gridlock*: Jedno vlákno chce například sečíst hodnotu proměnných `a` a `b`. Zamkne proměnnou `a` a chce zamknout i proměnnou `b`, avšak tu má zrovna zamčenou druhé vlákno. Pokud v tu chvíli druhé vlákno potřebuje přistoupit k proměnné `a`, není mu to povoleno, protože tato proměnná je zamčena prvním vláknem. První vlákno tedy čeká na odemčení proměnné `b` aby mohlo odemknout proměnnou `a`, zatímco to druhé na odemčení proměnné `a`, bez čehož neuvolní proměnnou `b`. Obě vlákna jsou do sebe zakleslá, čekají a k uvolnění nedojde nikdy.

Pro pěkný podrobný úvod do paralelního programování doporučuji https://computing.llnl.gov/tutorials/parallel_comp/.

V Pythonu existují dvě základní knihovny pro zpracování programu ve více vláknech: threading a multiprocessing. První z nich obsahuje více funkcionalit, avšak spouští všechna vlákna jen na jednom procesoru (jádře), k paralelnímu provádění výpočtů se tedy nehodí.[22]

Paralelizace na více výpočetních jádrech je v Pythonu implementována v knihovně **`multiprocessing`**. Zde si ukážeme využití objektů `Pool`, `Process` a `Value` z této knihovny na příkladu paralelní integrace metodou. Monte-Carlo. Vzorový kód je naprogramován v souboru **`Multiprocessing.py`**.

- `Integrate1DPool`: Nejjednodušší způsob paralelizace je pomocí objektu `Pool`. Při vytváření instance tohoto objektu mu předáme parametr `processes` udávající maximální počet procesů, které bude objekt obsluhovat.[23] Následně zavoláme jeho metodu `starmap`, jejíž první parametr je funkce, kterou chceme spustit v jednotlivých procesech, a druhý parametr je seznam, jehož každý element obsahuje n-tici s argumenty naší funkce. Metoda `starmap` spustí postupně naši funkci se všemi dostupnými n-ticemi parametrů ze seznamu, přičemž použije všechny dostupné procesy objektu `Pool`, počká na výsledky z jednotlivých vláken a shromáždí je do seznamu, který přiřazujeme proměnné `results`.[24] Průměr ze všech hodnot dá finální hodnotu integrace Monte-Carlo.

  Pokud by naše funkce měla jen jeden argument, namísto metody `starmap` bychom použili jednodušší metodu `map`. Ve vzorovém příkladu však spouštěné funkci `Integrate1D` musíme předat argumenty čtyři, proto volíme `starmap`.

  Při vytvoření instance objektu `Pool` jsme použili konstrukci s klíčovým slovem `with`, se kterou jsme se již setkali v úvodních cvičeních při práci se soubory.

[22] Použití knihovny **`threading`** je vhodné v případě, kdy provádíme úlohy, ve kterých se obvykle čeká na výsledek. Chceme například stáhnout webovou stránku z nějakého serveru, což může trvat nedefinovaně dlouho, a nechceme přitom, aby náš program přestal po dobu čekání reagovat.

[23] Pokud parametr `processes` nezadáme, Python použije všechna dostupná jádra procesoru, dostupná také v globální proměnné `os.cpu_count()`.

[24] Pokud je n-tic s argumenty víc než počet dostupných procesů, bude objekt `Pool` spouštět volání postupně.

- `Integrate1DProcess`: Složitější, avšak univerzálnější je použití objektů `Process`.[25] V tom případě je potřeba vykonat tři kroky:

  1. Vytvoříme instanci objektu typu `Process`. Při tom musíme specifikovat parametr `target`, jímž předáme funkci, kterou chceme v procesu spustit, a parametr `args`, který obsahuje její argumenty.
  2. Výpočet v procesu spustíme pomocí metody `start`. Výpočet se spustí asynchronně, naše hlavní vlákno programu tedy nečeká, než se výpočet v novém procesu dokončí.
  3. Chceme-li počkat na výsledek, využijeme metodu `join`. Zavoláme-li ji pro daný proces, bude hlavní vlákno programu čekat na dokončení příslušného procesu. My chceme počkat na dokončení všech procesů, proto si musíme po vytvoření a nastartování procesy uschovat (ve vzorovém kódu je uschováváme v proměnné `processes`) a poté zavolat `join` nade všemi.

  Při použití objektu `Process` **není možné získat návratovou hodnotu volané funkce**. Proto musíme program ještě trochu zesložitit a volanou funkci vrátit ve sdílené proměnné — instanci objektu `Value`. Tento objekt slouží k přenosu hodnot mezi hlavním vláknem a ostatními procesy i mezi procesy navzájem. První parametr při vytváření instance objektu `Value` udává typ (`'d'` pro číslo s desetinnou čárkou, `'i'` pro celočíselný typ), druhý parametr počáteční hodnotu. Hodnota proměnné je uložena v atributu `value`. Vytvoříme takovouto proměnnou pro každý jednotlivý proces. Navíc musíme naprogramovat pomocnou funkci (*wrapper*), kterou pojmenujeme `Integrate1DP` a která nám zavolá naši funkci `Integrate1D` a její návratovou hodnotu uloží do sdílené proměnné. Naší funkci `Integrate1D` chceme předat beze změny všechny parametry, které pomocná funkce `Integrate1DP` dostane. K tomu slouží konstrukce `*args, **kwargs`.

  Objekt `Value` má v sobě implementovaný zámek. Stačilo by tedy použít jen jednu instanci `Value` pro všechny procesy a výsledky dílčích integrací do ní přičítat. Průměr bychom nakonec získali vydělením počtem procesů.

Důležité poznámky:

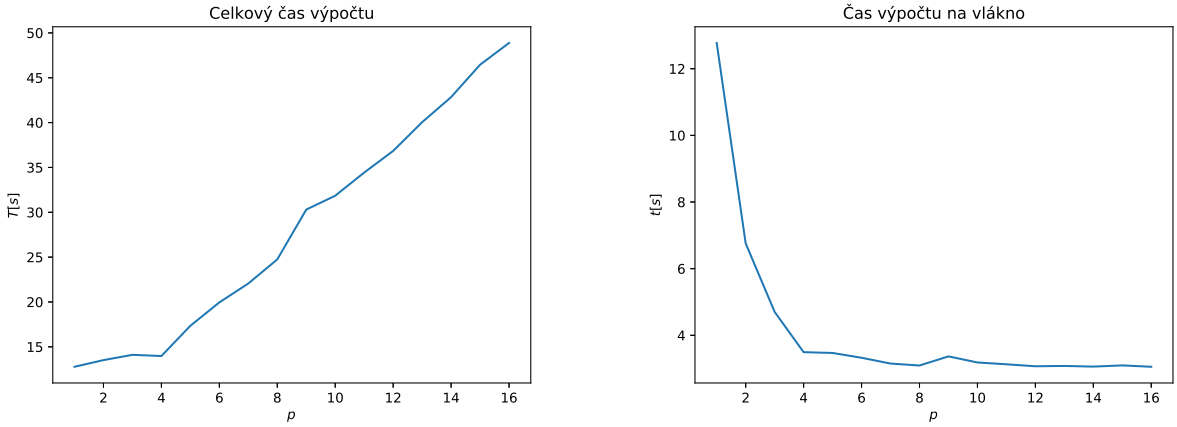
- Python postupuje tak, že v jednotlivých procesech spouští celé moduly obsahující spouštěnou funkci. V našem případě je v každém procesu spuštěn celý modul `Multiprocessing.py`. Tento modul obsahuje globální část kódu, která v našem případě spouští funkce, které spouštějí víceprocesorové zpracování, a došlo by tedy k zacyklení programu. Abychom tomuto zabránili, je nutné tu část kódu, která smí být spuštěna jen z hlavního vlákna, vložit do podmínky `if __name__ == "__main__":`
- Ve Windows ve vývojovém prostředí IDLE nefunguje funkce `print`, pokud ji používáme z jiného než z hlavního vlákna (nic nevypíše). V jiných prostředích či operačních systémech by to mělo fungovat.

Efekty paralelizace lze pozorovat pomocí funkce `PlotIntegrate1DDuration`. Tato funkce spouští postupně paralelní výpočet pro různé počty navzájem běžících paralelních vláken, a výsledky zobrazí do grafů, které jsou vykresleny v obrázku 16. Pozorujeme, že nejlepší přesnosti (nejvyššího poštu Monte-Carlo bodů pN) za jednotku času dosáhneme, pokud vytížíme všechna dostupná jádra.

Úkol 8.1: *Prostudujte si vzorový příklad v souboru* `Multiprocesing.py` *a upravte ho tak, aby počítal hodnotu integrálu* I_3 (91) *ve více vláknech. Zjistěte si, kolik výpočetních jader má procesor na vašem počítači, a spusťe výpočet tak, aby zaměstnal všechna jádra.*

Úkol 8.2: *Časově náročný, a přitom jednoduše paralelizovatelný je výpočet součinu dvou matic. Jelikož metoda* `starmap` *pracuje jednoduše jen s vektory, naprogramujte paralelní výpočet součinu matice a vektoru (výsledkem je vektor).*

[25] Podobným způsobem se používá knihovna `pthread` v programovacím jazyce C++.



(a) (b)

Obrázek 16: Výpočetní čas integrace metodou Monte-Carlo při použití různého počtu paralelních procesů p, přičemž v každém vlákně je spuštěn výpočet integrálu I_1 (87) s $N = 10^6$. Výpočet byl prováděn na PC se 4-jádrovým procesorem Intel se zapnutou technologií hyperthreading. Celková doba výpočtu je T, doba výpočtu vztažená na jádro je $t \equiv T/p$. Pozorujeme, že celková doba výpočtu se pro $p \leq 4$ téměř nemění, protože každý proces běží na svém vlastním jádře. Pro $4 \leq p \leq 8$ již narůstá celkový výpočetní čas, protože procesy se již musejí dělit o dostupná jádra, avšak výpočetní čas na vlákno stále nepatrně klesá díky hyperthreadingu. Pro $p \geq 8$ se již procesor saturuje a zvětšování počtu procesů nepřináší žádný efekt. Pro $p \gg 8$ bychom pozorovali naopak zhoršování času výpočtu na vlákno, protože by si pro sebe více a více výpočetního času brala režie operačního systému, aby všechna vlákna obhospodařila.

9 Fourierova transformace

Fourierova transformace je jednou ze základních metod zpracování signálu. Spočívá v převodu časově závislé funkce $h(t)$ na funkci udávající spektrum harmonických frekvencí $H(\omega)$. S Fourierovou transformací analytických funkcí se seznámíte v matematické analýze a teorii distribucí. My se zde budeme zabývat diskrétní verzí této transformace, která je se používá pro numerické zpracování naměřených signálů.

9.1 Diskrétní Fourierova transformace

Transformace a případná zpětná transformace $h_j \leftrightarrow H_k$ se provádějí předpisem

$$H_k = \frac{1}{N} \sum_{j=0}^{N-1} h_j \, \mathrm{e}^{-\frac{2\pi i j k}{N}},$$
$$h_j = \sum_{k=0}^{N-1} H_k \, \mathrm{e}^{+\frac{2\pi i j k}{N}}, \tag{94}$$

kde

- h_j, $j = 0, 1, \ldots, N-1$ je *vstupní signál (časová řada)* délky N naměřený v ekvidistantních okamžicích odpovídajících časům
$$t_j = \frac{j}{f_s}. \tag{95}$$
- f_s je *vzorkovací frekvence* a odpovídá počtu měření za sekundu.
- H_k, $k = -\frac{N-1}{2}, \ldots, 0, \frac{N-1}{2}$ (pro N liché) je spektrální rozklad vstupního signálu, kde
$$f_k = |k| \, \frac{f_s}{N-1} \tag{96}$$

je frekvence odpovídajícího příspěvku. Níže ukážeme, že k jedné frekvenci existují dvě komponenty H_k a H_{-k}, které se liší jen fází. Jelikož díky periodičnosti platí $H_k = H_{k+N}$, je při programování praktičtější uvažovat index $k = 0, \ldots, N-1$. Pak H_k a H_{N-k} odpovídají stejné frekvenci.

- Nejvyšší frekvence, kterou jsme schopni ze vstupního signálu určit, nezávisí na délce signálu N a je určena pouze vzorkovací frekvencí: $f_{\max} = f_s/2$. Nazývá se *Nyquistova frekvence*.
- Nejnižší frekvence, kterou ze signálu vyextrahujeme, je $f_{\min} = \frac{f_s}{N-1}$. Pro signály velmi nízkých frekvencí potřebujeme tedy dlouhé vstupní časové řady.
- Frekvence f_0 odpovídá tzv. *stejnosměrnému příspěvku* a odpovídající komponenta H_0 udává střední hodnotu signálu.

Přímá a zpětná transformace se liší jen znaménkem a normalizačním faktorem $1/N$, jehož umístění do zpětné Fourierovy transformace je věcí konvence.[26]

Úkol 9.1: *Dokažte, že provedením Fourierovy transformace a zpětné Fourierovy transformace dostanete původní časovou řadu.*

Z předpisu (94) vyplývá, že výsledek Fourierovy transformace je obecně sekvence *komplexních* čísel. Rozepsáním komplexní exponenciály dostaneme

$$\begin{aligned} H_k^R &= \frac{1}{N}\sum_{j=0}^{N-1}\left(h_j^R \cos\frac{2\pi jk}{N} + h_j^I \sin\frac{2\pi jk}{N}\right), \\ H_k^I &= \frac{\mathrm{i}}{N}\sum_{j=0}^{N-1}\left(-h_j \sin\frac{2\pi jk}{N} + h_j^I \cos\frac{2\pi jk}{N}\right), \end{aligned} \tag{97}$$

kde

$$\begin{aligned} h_j &= h_j^R + \mathrm{i}h_j^I, \\ H_k &= H_k^R + \mathrm{i}H_k^I \end{aligned} \tag{98}$$

je rozklad na reálnou a imaginární část. Analogický vztah bychom získali pro zpětnou Fourierovu transformaci.

Za předpokladu, že vstupní časová řada h_j je reálná (obecně reálná být nemusí, ale v praxi obvykle bývá), můžeme pro reálnou a imaginární složku spektrálního rozkladu H_k psát

$$H_k = \frac{1}{N}\sum_{j=0}^{N-1} h_j \cos\frac{2\pi jk}{N} - \frac{\mathrm{i}}{N}\sum_{j=0}^{N-1} h_j \sin\frac{2\pi jk}{N}. \tag{99}$$

Ze sudosti cosinu a lichosti sinu plyne, že $H_k^R = H_{-k}^R$ a $H_k^I = -H_{-k}^I$.

Komponenty H_k lze také rozepsat pomocí jiných dvou reálných čísel: velikosti $|H_k|$ a fáze ϕ_k,

$$H_k = |H_k|\,\mathrm{e}^{i\phi_k}, \tag{100}$$

$$|H_k| = \sqrt{\left(H_k^R\right)^2 + \left(H_k^I\right)^2},$$

$$\phi_k = \arctan\frac{H_k^I}{H_k^R}. \tag{101}$$

V praxi je nejdůležitější údaj velikost příspěvku k-té frekvence

$$A_k = |H_k| + |H_{-k}| = 2\,|H_k|\,, \qquad k = 1, \ldots, \frac{N-1}{2}, \tag{102}$$

[26] Jiná často používaná konvence je tento faktor rozdělit symetricky mezi obě transformace: pak bude před oběma sumami ve vztazích (94) stát $\frac{1}{\sqrt{N}}$ a Fourierova transformace bude unitární transformací mezi vektory $\boldsymbol{h} = (h_j)$ a $\boldsymbol{H} = (H_k)$ (unitární transformace zachovává délku komplexního vektoru).

který udává amplitudu odpovídající harmonické vlny. Místo amplitudy se také často používá kvadrát

$$S_k \equiv |A_k|^2 , \tag{103}$$

který se nazývá *výkonové spektrum* a udává, jak již název napovídá, výkon, který do signálu vnáší k-tá harmonická vlna.

Fourierova transformace tedy vyjadřuje signál jako součet prostých harmonických vln (sinů a cosinů) s frekvencemi f_k a amplitudami A_k. Hudební terminologií se jedná o rozklad souzvuku několika tónů na jednotlivé jednoduché tóny.

9.2 Použití Fourierovy transformace

- Získání dominantních frekvencí neznámého signálu (například délky slunečního cylku, vlastní frekvence kmitů složitého objektu atd.).
- Filtrování signálu, odstranění šumu (signál očekáváme na určitých frekvencích, šumu se zbavíme, pokud komponenty H_k od ostatních frekvencích vynulujeme).
- Komprese signálu: uchováme jen složky signálu s několika nejvyššími příspěvky H_k. Toho se využívá v kompresi zvukového signálu (například MP3) i obrazového signálu (formát JPEG).

My si vyzkoušíme Fourierovu transformaci na zvukovém souboru.

V Pythonu je velké množství knihoven, které umějí pracovat se zvukem. Pro naše účely postačí jednoduchá knihovna sounddevice.[27] Pokud ji nemáte nainstalovanou, doinstalujete ji příkazem

`python -m pip install sounddevice`	Windows
`pip install sounddevice`	Linux, Mac

Z této knihovny využijeme následující funkce:

- `play(data, fs)` přehraje časovou řadu uloženou v poli `data` při vzorkovací frekvenci `fs`. Jsou-li elementy přehrávané řady typu `float`, jejich hodnoty musejí ležet v rozmezí $h_j \in \langle -1, 1 \rangle$, jinak bude zvuk přebuzený a zkreslený. Vzorkovací frekvence zvukových souborů bývají nejčastěji násobky či podíly 44100 Hz nebo 48000 Hz.[28]
- Přehrávání probíhá asynchronně. Chceme-li počkat, než bude přehrávání dokončeno, použijeme funkci `wait()`.
- `stop()` okamžitě zastaví přehrávání.
- `rec(numsamples, samplerate=fs, channels=2)` nahraje časovou řadu s `numsamples` hodnotami. `channels=1` pro mono signál, `channels=2` pro stereo signál.

Jelikož zde budeme pracovat se zvukovými soubory typu WAV, budeme potřebovat ještě knihovnu soundfile k jejich načtení.

V repozitáři je ukázkový soubor **`Sound.py`**, který načte zadaný WAV soubor a přehraje ho.

Úkol 9.2: *Naprogramujte Fourierovu transformaci a zpětnou Fourierovu transformaci podle vztahů (94). Můžete počítat buď nezávisle reálnou a imaginární část, nebo využít toho, že funkce knihovny* **`numpy`** *rozumějí komplexním číslům. Imaginární jednotka v Pythonu je* `1j`*, komplexní číslo* $3+4\mathrm{i}$ *se tedy zapíše ve tvaru* `3 + 4j`*.*

[27] Další knihovny pro přehrávání a nahrávání zvuku jsou uvedeny na https://realpython.com/playing-and-recording-sound-python.

[28] 44100 Hz je vzorkovací frekvence používaná na CD, 48000 Hz na DVD.

Úkol 9.3: *Vytvořte časovou řadu délky $N = 2000$ se vzorkovací frekvencí $f_s = 2000\,Hz$ (signál tedy bude trvat přesně $1\,s$) danou součtem tří harmonických funkcí:*

$$h_j = \sum_{n=1}^{3} a_n \sin\left(2\pi F_n t_j\right), \tag{104}$$

kde

$$a_1 = 0.1, \qquad a_2 = 0.2 \qquad a_3 = 0.3,$$
$$F_1 = 440\,Hz \qquad F_2 = \frac{5}{4}F_1 \qquad F_3 = \frac{3}{2}F f_2$$

a časy t_j jsou dány vztahem (95). Přehrajte si ji pomocí funkcí knihovny `sounddevice`.[29] *Spočítejte Fourierovu transformaci a vykreslete do grafu amplitudy A_k na frekvencích f_k. Přesvědčte se, že graf bude mít tři vrcholy s výškami odpovídajícími zadaným amplitudám a_n a středy v místech zadaných frekvencí F_n.*

Zopakujte výpočet pro $N = 1000$ a $f_s = 1000\,Hz$. V tomto případě jsou již frekvence F_2, F_3 větší než Nyquistova frekvence f_{max}. Zvuk bude zkreslený a vrcholy ve frekvenčním diagramu po provedení Fourierovy transformace se posunou.

Úkol 9.4: *V adresáři* `sounds/` *v repozitáři jsou soubory* `a.wav`, `e.wav`, `i.wav`, `o.wav`, `u.wav`, *ve kterých jsou nahrané krátké vzorky samohlásek.*[30] *Vzorkovací frekvence nahrávek je $f_s = 16000\,Hz$, nahrávky mají délku okolo $N = 8000$. Soubory načtěte pomocí knihovny* `soundfile`, *přehrajte si je, vyřízněte z nich kvůli rychlosti okno délky $N = 2000$ (vezměte například prostřední část* `sound[3000:5000]`*), vypočítejte na něm Fourierovu transformaci a zakreslete získané amplitudy A_k do jednoho grafu pro všech pět samohlásek. Uvidíte, že každá samohláska má zcela jinak vysoké dominantní frekvence. Této skutečnosti se využívá pro strojovou analýzu hlasu.*[31]

Úkol 9.5: *V adresáři* `sounds/` *v repozitáři je soubor* `BlackHolesCollision.wav` *zahrnující nasimulovaný průběh gravitačních vln těsně před srážkou dvou černých děr.*[32] *Načtěte tento soubor, přehrajte si ho (uslyšíte charakteristický tzv.* chirp sound*). Pozor, soubor má dva kanály, pro následující analýzu vyberte pouze jeden z nich příkazem* `sound = sound[:,0]`. *Rozdělte časovou řadu na časová okna délky $N_W = 2000$ bodů a pro každé okno spočítejte Fourierovu transformaci a amplitudy A_k. Následně vykreslete konturový graf, kde na ose x bude čas (začátku nebo středu použitého časového okna), na ose y frekvence a na ose z (barevný kód) amplituda. Frekvence omezte pomocí příkazu* `plt.ylim(0, 500)` *na hodnoty $\langle 0\,Hz, 500\,Hz\rangle$.*

9.3 Rychlá Fourierova transformace

Fourierova transformace naprogramovaná pomocí definičních vztahů (94) má časovou složitost $\mathcal{O}(N^2)$, pro delší časové řady tedy doba výpočtu prudce roste (již pro $N = 10000$ si počkáte několik minut). Existuje mnohem efektivnější algoritmus s časovou složitostí $\mathcal{O}(N \log N)$. V Pythonu naleznete funkci počítající rychlou Fourierovu transformaci ze zadané časové řady v balíku `numpy`. Jmenuje se fft.fft.

Úkol 9.6: *Spočítejte Fourierovu transformaci pomocí funkce naprogramované v úkolu 9.2 a pomocí knihovní funkce* `numpy.fft.fft` *pro jednu z hlásek z úkolu 9.4. V grafu porovnejte získané amplitudy. Měly by být stejné.*

[29] Uvedené frekvence odpovídají po řadě tónům a^1, c#^{2}, e^2 v přirozeném ladění. Zaznít by tedy měl durový kvintakord.

[30] Nahrávky jsou staženy z https://homepages.wmich.edu/~hillenbr/voweldata.html.

[31] Pomocí funkce `rec` si můžete nahrát a zanalyzovat vlastní hlas.

[32] Soubor pochází z http://web.mit.edu/sahughes/www/sounds.html.

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).