Zápisky k předmětu Využití počítačů ve fyzice

Pavel Stránský

26. dubna 2021

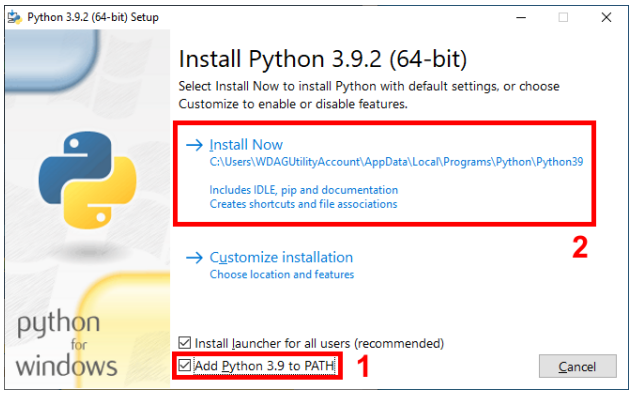
Obsah

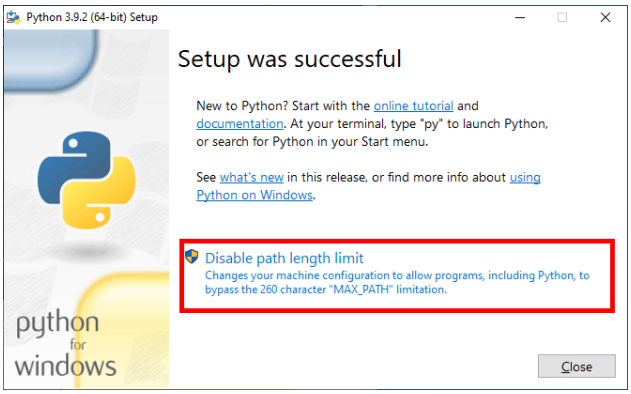
1 Instalace používaných nástrojů

Příklady k cvičení budou demonstrovány v nejnovější verzi programovacího jazyka Python. Jako vývojové prostředí doporučuji Visual Studio Code. Tento volně dostupný program lze nainstalovat na všechny neužívanější operační systémy (Linux, Windows, macOS). Má nepřeberné možnosti při editaci zdrojových souborů, překladu a ladění snad ve všech známých programovacích jazycích. Bohaté možnosti nastavení umožnují přizpůsobit si práci svým potřebám (například zvýrazňování syntaxe, klávesové zkratky či vzhled prostředí). Pomocí doplňků ho můžete integrovat s dalšími službami, například s verzovacím programem Git, či vzdálenými repozitáři, čehož také využijeme. Komunita, která toto vývojové prostředí používá a spravuje, je obrovská, což zaručuje dobrou podporu a rychlé přidávání nových funkcí.

1.1 Instalace Pythonu

Instalační soubor pro svůj operační systém stáhnete ze stránky python.org. Při instalaci na počítač s Windows doporučuji zvolit „Add Python to PATH“, což zjednoduší práci s Pythonem z příkazové řádky, a na poslední obrazovce zvolit „Disable path length limit“:





Pro ověření instalace napíšeme v příkazové řádce příkaz `python`.[1] Tím se spustí REPL[2] Pythonu, ve které můžeme již psát všechny příkazy programovacího jazyka, které se po zadání ihned provedou a vypíšou výsledek.

1.1.1 Instalace doplňujících knihoven

Samotná instalace Pythonu obsahuje jen minimální množství nejnutnějších knihoven. My budeme využívat ještě následující rozšiřující knihovny:

- `NumPy` (**Num**erical **Py**thon: numerická matematika, řady a vícedimenzionální datové typy),
- `SciPy` (**Sci**entific **Py**thon: algoritmy pro optimalizaci, statistiku, řešení diferenciálních rovnic, lineární algebru, atd.),
- `Matplotlib` (vizualizace, grafy).

K jejich doinstalování slouží modul `pip`. V příkazové řádce napíšeme

```
python -m pip install numpy scipy matplotlib
```

čímž se nainstalují naráz všechny tři knihovny.[3]

Existuje samozřejmě celá řada dalších užitečných a používaných knihoven, jako je například `Pandas` pro analýzu dat nebo `SymPy` pro symbolické výpočty, na které v tomto kurzu nedojde.

[1] Na počítačích s Linuxem je příkaz v terminálu `python3`.

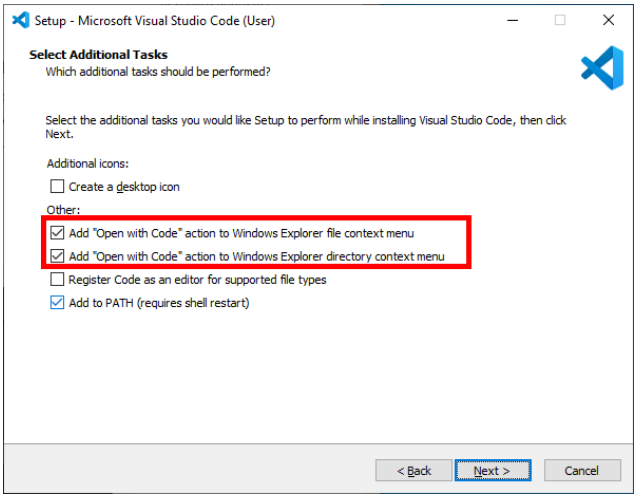
[2] **R**ead-**E**valuate-**P**rint-**L**oop.

[3] Pokud na počítači s Linuxem uvedený postup nebude fungovat, je potřeba nejprve nainstalovat instalátor `pip` pomocí příkazu `sudo apt install python3-pip`. Pak lze použít buď výše uvedený příkaz, nebo stručnější `pip3 install numpy`.

1.2 Instalace Visual Studio Code

Instalace jazyka Python obsahuje jednoduché vývojové prostředí nazvané IDLE.[4] To však poskytuje jen omezené možnosti co se týče ladění, psaní rozsáhlejších projektů s více zdrojovými soubory nebo integrace s verzovacími programy.

Pro serióznější práci budeme používat Visual Studio Code s doplňkem pro programovací jazyk Python. Instalační soubor stáhnete ze stránky code.visualstudio.com. Během instalace na počítač s Windows doporučuji zvolit obě možnosti „Add Open with Code action to...“,



což zjednoduší otevírání složek s projekty nebo samostatných souborů pomocí pravého tlačítka myši.

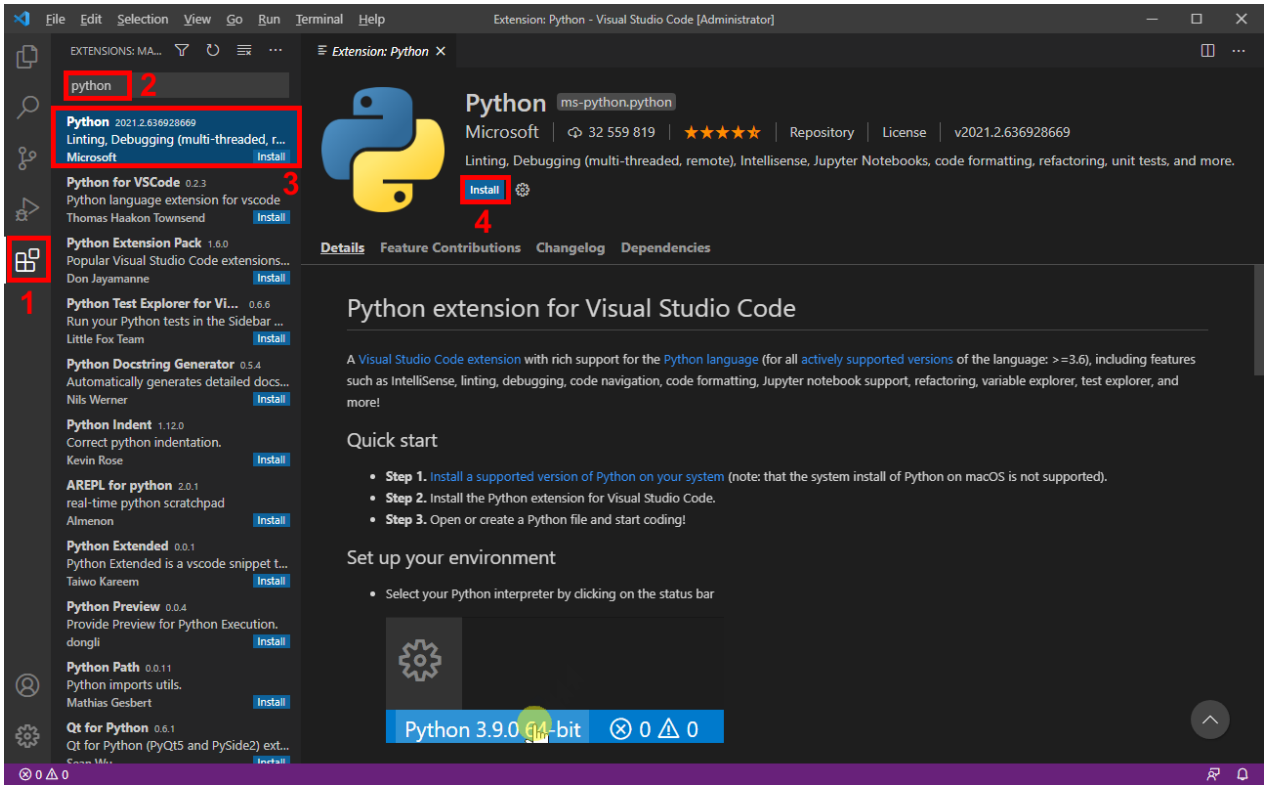
Na operačním systému Linux je nejjednodušší provést instalci pomocí Snap Store příkazem v terminálu

```
sudo snap install --classic code
```

nebo použít tento návod.

[4] **I**ntegrated **D**evelopment and **L**earning **E**nvironment.

1.2.1 Instalace doplňku pro Python

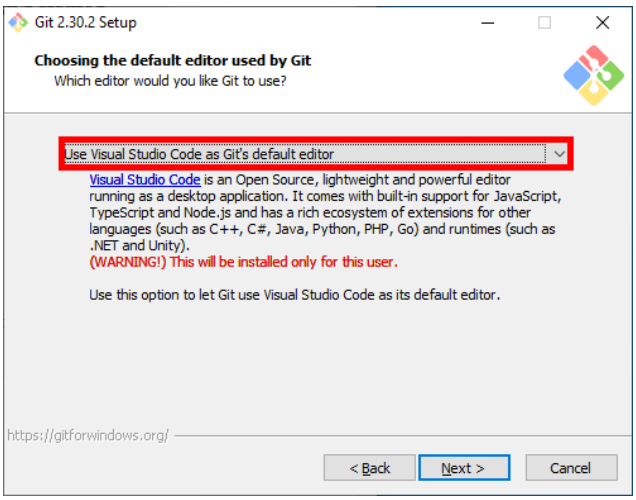


Ke správě doplňků (extensions) se dostanete kliknutím na ikonku 1 nebo stisknutím `Ctrl+Shift+X`. Vyhledáte doplněk Python 2 od Microsoftu, vyberete ho 3 a nainstalujete 4.

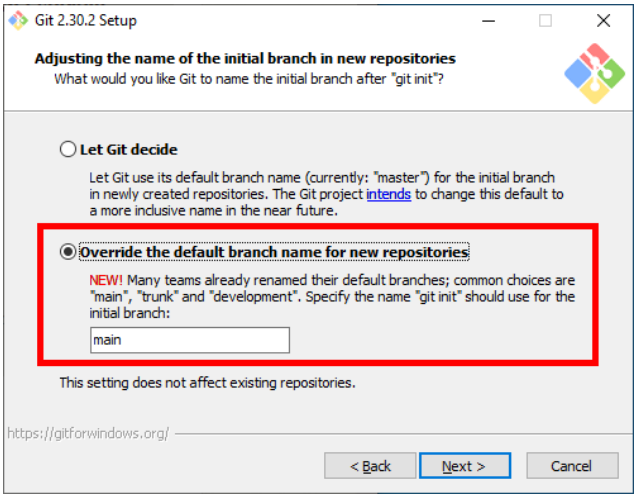
1.3 Instalace Git

Instalační soubor verzovacího systému Git stáhnete z webové stránky git-scm.com. K instalaci Gitu potřebujete administrátorská práva.

V následujícím postupu zobrazuji jen ty snímky obrazovky, na kterých je vhodné vybrat jinou volbu, než jaká je instalátorem standardně nabízena.



Jako editor zvolíme dříve nainstalované Visual Studio Code. Systém Git vyžaduje editor jednak pro povinný komentář každé zapsané změny (commit), jednak pro ošetření kolizí při slučování větví (merge).



Hlavní větev repozitáře se donedávna standardně jmenovala `master`. Vzhledem k negativním konotacím tohoto slova v angličtině se přechází na neutrálnější označení. Nejběžnější je `main`, na které již přešel i populární správce vzdálených repozitářů GitHub. Doporučuji tedy zvolit pojmenování `main`.

Všechna nastavení lze samozřejmě kdykoliv po instalaci změnit.

2 Úvod do používaných nástrojů

V této sekci naleznete základy použití verzovacího systému Git a úvod do syntaxe a idiomatiky jazyka Python. Na rozdíl od zbytku poznámek bude tato sekce v průběhu cvičení postupně doplňována podle toho, s jakými technikami se seznámíme v hlavní části cvičení.

2.1 Verzovací systém Git

Každý si patrně už někdy v životě zasteskl, že nemá uložené dřívější verze svých souborů. Změny, které se ukazují být nevhodné či provedené omylem (kočka nepozorovaně přejde po klávesnici a my pak soubor uložíme), kamarád, který z nějakých důvodů chce tu verzi vašeho programu, kterou jste mu poskytli před rokem, a vy jste mezitím kód zásadně přepsali, to vše jsou důvody k předsevzetí nějakým způsobem důležité milníky v práci na projektu archivovat.

Triviální verzování může spočívat například v ručním ukládání kopií projektu s daty či jinými označeními jednotlivých verzí. Takovýto postup je ale velmi těžkopádný, jelikož v každé kopii musí být uloženy všechny soubory projektu, i pokud se v nich od předchozí verze nic nezměnilo. Obtížně se vyhledává, co přesně se mezi jednotlivými verzemi změnilo a kdo změnu provedl, pokud na projektu pracuje větší tým, přičemž obtížnost roste s velikostí a komplexností projektu. Pro usnadnění uchovávání historie změn proto vznikly verzovací systémy.

Verzovací systém pomáhá udržet historii změn souborů projektu, přičemž ke každému snímku historie se lze vrátit. Vše provádí chytře a v maximální míře automaticky. Systém sám sleduje, v jakých souborech byly provedeny změny. Pokud se jedná o textový soubor, umí porovnat aktuální a libovolnou historickou verzi řádek po řádku a zvýraznit odlišnosti. Umožňuje pracovat s nezávislými vývojovými větvemi (*branches*), ve kterých lze například zkoušet různý přístup k řešení daného problému a mezi kterými lze jednoduše přepínat. Vybrané řešení lze pak snadno začlenit (*merge*) do hlavní vývojové větve.

Soubory projektu a informace o jejich historických změnách se nazývá souhrnně *repozitář*. V něm se uchovávají nejen verze souborů, ale také údaje o tom, kdo a kdy změny provedl. To předurčuje verzovací systémy pro efektivní správu týmových projektů, kdy každý člen týmu pracuje na určité části projektu a své změny následně do projektu včleňuje.

My budeme používat verzovací systém Git. Ten patří mezi *distribuované* verzovací systémy[5], což znamená, že každý uživatel má na svém počítači celý obsah repozitáře a pouze ve chvílích, kdy uzná za vhodné nebo kdy je k tomu příležitost, může své změny synchronizovat se *vzdáleným repozitářem*, ve kterém se shromažďují změny od všech členů týmu, začleňují do hlavní vývojové větve projektu a hlídají případné kolize. Výhody tohoto přístupu oproti centralizovaným verzovacím systémům[6] jsou následující:

1. Práce na projektu nevyžaduje připojení k nějakému centrálnímu serveru s repozitářem, a tedy můžete pracovat offline klidně někde v džungli nebo na Marsu.

2. Není potřeba spravovat zvlášť server s repozitářem a zvlášť klientské počítače. Vše je na jednom místě.

3. Při práci na týmovém projektu není případná porucha počítače spojená se ztrátou dat žádná katastrofa, protože ostatní kolegové mají celou kopii repozitáře na svých počítačích.

Git nejefektivněji funguje na textové soubory, ale zvládne verzovat i soubory binární (například obrázky).

Git je v dnešní době jeden z nejpopulárnějších verzovacích systémů. Pod jedho správou jsou vyvíjeny i velké projekty, například samo Visual Studio Code. Tento projekt je navíc otevřený (open source), což znamená, že kdokoliv, tedy i vy nebo já, má přístup k celému repozitáři, tedy ke všem

[5] Další distribuované verzovací systémy jsou například Mercurial, Bazaar či Darcs.
[6] Nejznámější reprezentant centralizovaného verzovacího systému je Subversion.

zdrojovým kódům a k jejich kompletní historii. Může se do práce na projektu zapojit, přispět k jeho vývoji a začít psát historii sám.

Program Git pochází z prostředí Linuxu a je navržený tak, aby se s ním dalo pracovat z příkazové řádky (terminálu) pomocí jednoduchých textových příkazů. Tímto způsobem lze používat všechny dostupné funkce Gitu. My se s nejdůležitějšími funkcemi seznámíme právě pomocí textových příkazů, protože na nich se nejlépe naučí, jak tento Git funguje a jaké jsou jeho možnosti. Jakmile si člověk ujasní principy verzování pomocí Gitu, může využít některý z množství nástrojů a doplňků pro různé programy, které práci s repozitáři usnadní a zrychlí. Obsluha Git repozitářů je integrována i do vývojového prostředí Visual Studio Code.

Na stránkách projektu Git najdete podrobný interaktivní návod ke všem funkcím verzovacího systému (a to částečně i v češtině).

2.1.1 Prvotní nastavení

Git nelze používat do té doby, než jsou nastaveny základní informace o uživateli. To se provede pomocí příkazů v příkazové řádce (terminálu)

```
git config --global user.name "..."
git config --global user.email "..."
```

přičemž za ... doplníte své jméno (přezdívku) a email. Těmito údaji se podepisují všechny zapsané změny v repozitáři. Pokud tedy spolupracujete na projektu s jinými lidmi, je dobré zvolit takové údaje, pomocí kterých vás kolegové snadno identifikují a úspěšně kontaktují.

Výpis všech nastavení získáte příkazem

```
git config --list
```

Uvidíte, že v mezi nastaveními je i název hlavní větve repozitáře (`init.defaultbranch=main`) a cesta k nastavenému textovému editoru (v našem případě Visual Studio Code).

Všechna nastavení a práci s příkazem `config` najdete v tomto podrobném návodu nebo zadáním příkazu

```
git help config
```

2.1.2 Čtyři možné stavy souborů v repozitáři

- *Nesledovaný (untracked):* Systém Git historii tohoto souboru neuchovává. Každý nové vytvořený soubor je v tomto stavu. Speciální podskupinu tvoří *ignorované (ignored)* soubory, což bývají v drtivé většině pomocné soubory nebo soubory s citlivými údaji. Seznam ignorovaných souborů se uvede do speciálního souboru **`.gitignore`**, o kterém pojednává sekce 2.1.4.

- *Připravený k zapsání (staged):* Soubor je systémem Git sledován (verzován). Od předchozího zapsání v něm došlo ke změnám (nebo byl nově vytvořen) a k zapsání byl připraven pomocí příkazu

  ```
  git add soubor
  ```

  Hromadný příkaz

  ```
  git add *
  ```

  připraví k zapsání všechny změněné soubory a všechny nové soubory kromě těch, které jsou uvedeny v souboru **`.gitignore`** (viz sekce 2.1.4).

  Přehled změn v souborech připravených k zapsání oproti naposledy uložené verzi dá příkaz

  ```
  git diff --staged
  ```

Verze souborů se změnami připravenými k zapsání jsou uloženy mimo pracovní adresář: při dodatečných modifikacích souboru se k těmto verzím souboru lze vrátit příkazem

```
git restore --staged soubor
```

(pozor, tímto příkazem se nenávratně ztratí všechny změny v souboru, které byly provedeny od jeho poslední přípravy k zapsání).

- *Změněný (modified):* Soubor je systémem Git sledován (verzován). Od předchozího zapsání v něm došlo ke změnám, avšak soubor ještě nebyl označen k zapsání. Přehled změn v souborech oproti naposledy uložené verzi či oproti verzi připravené k zapsání dá příkaz

  ```
  git diff
  ```

- *Zapsaný (committed):* Soubor je systémem Git sledován (verzován) a jeho historie je zaznamenána v repozitáři. Jedná se o verzi souboru, ke které je vždy možné se vrátit. Zapsání (commit) všech souborů připravených k zapsání (staged) se provede příkazem

  ```
  git commit [m "popis revize"]
  ```

  kde `"popis revize"` je povinné označení ukládaného historického snímku. Pokud se neuvede část v hranatých závorkách, systém Git spustí textový editor zvolený při jeho instalaci a v něm otevře soubor, do kterého název revize a případně nějaké podrobnější komentáře uvedete. Po uložení souboru a uzavření textového editoru se provede zapsání.

2.1.3 Životní cyklus souborů v repozitáři

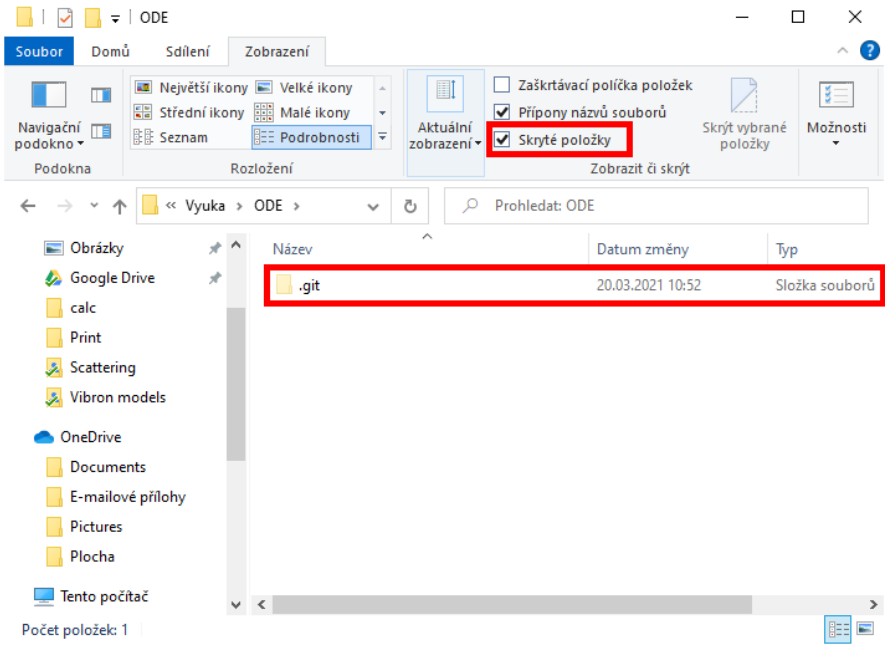
Práci s repozitářem budu demonstrovat na příkladu projektu pro integraci diferenciálních rovnic ze sekce 3.

2.1.3.1 Vytvoření nového repozitáře

Ve složce, jejíž historii chceme ukládat, se repozitář vytvoří jednoduchým příkazem

```
git init
```

Existenci repozitáře lze vždy odhalit pomocí skryté podsložky s názvem `.git`:



V této složce je uložena veškerá zapsaná historie vašeho projektu od jeho vytvoření. Pokud tuto složku smažete, přijdete tím o „paměť“ systému Git pro daný projekt, avšak aktuální soubory projektu zůstanou tak, jak jsou. Pokud naopak přesunete nebo překopírujete složku s projektem včetně podsložky `.git` kamkoliv jinam, klidně na jiný počítač, kopírujete zároveň celou historii projektu. Nedoporučuji jakkoliv měnit soubory v této složce, protože tím můžete nevratně narušit integritu repozitáře a systém Git přestane být schopen repozitář používat.

2.1.3.2 Stav repozitáře

Aktuální stav repozitáře kdykoliv zjistíte pomocí

```
git status
```

Příkaz vypíše název aktuální větve a stav všech souborů v projektu. Pro čerstvě vytvořený repozitář se dozvíte něco takového:

```
D:\Pavel\Documents\Fyzika\Vyuka\ODE>git status
On branch main

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

2.1.3.3 Vytvoření a první zapsání nového souboru

Ve složce s projektem otevřeme Visual Studio Code (napsáním `code` do příkazové řádky nebo stiskem pravého tlačítka myši nad otevřenou složkou a výběrem „Open with Code“), vytvoříme nový soubor a uložíme ho jako `ode.py`. Příkaz `git status` ukáže

```
D:\Pavel\Documents\Fyzika\Vyuka\ODE>git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        ode.py

nothing added to commit but untracked files present (use "git add" to track)
```

Systém Git si je vědom toho, že si uživatel nemusí pamatovat všechny příkazy, a proto se snaží práci usnadnit a napovídá, které další akce lze v další fázi použít a jak.

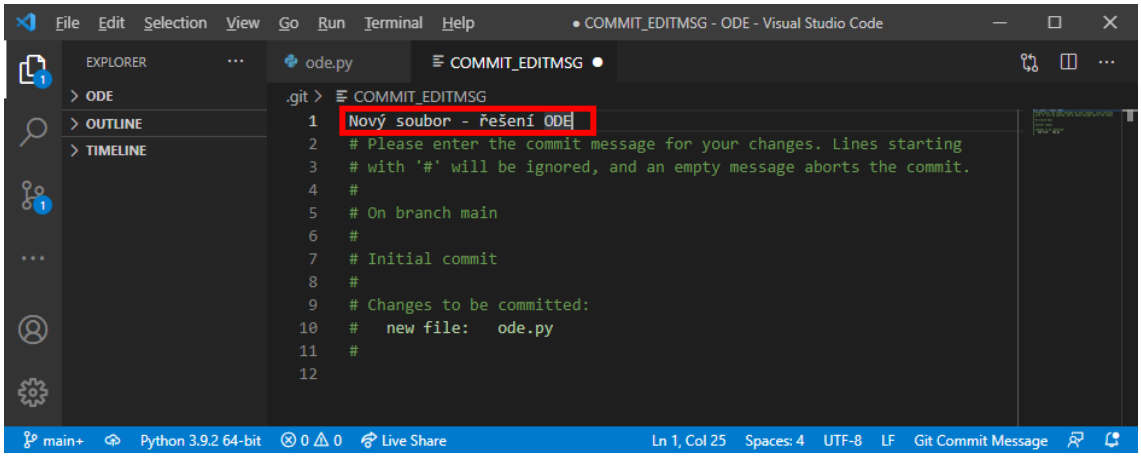
Příkazem `git add *` se nový soubor připraví k zapsání. Stav repozitáře se změní takto:

```
D:\Pavel\Documents\Fyzika\Vyuka\ODE>git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   ode.py
```

Zapsání se provede příkazem `git commit`. Otevře se editor (nový soubor ve Visual Studiu Code), do kterého uvedeme stručný a výstižný popis změn:



Po uložení a zavření souboru se vytvoří první snímek historie našeho projektu. V příkazové řádce vidíme stručný souhrn, co se zapsalo.

```
D:\Pavel\Documents\Fyzika\Vyuka\ODE>git commit
hint: Waiting for your editor to close the file...
[main (root-commit) 1b69421] Nový soubor - řešení ODE
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 ode.py

D:\Pavel\Documents\Fyzika\Vyuka\ODE>git status
On branch main
nothing to commit, working tree clean
```

2.1.3.4 Změny v souboru

Do souboru **ode.py** vepíšeme první část kódu, kterým bude řešení jedné diferenciální rovnice Eulerovou metodou 1. řádu.

```
def euler_1(model, y, t, dt):
    y1 = y + model(y, t) * dt
    t1 = t + dt
    return y1, t1

def ode_solve(model, initial_condition, integrator=runge_kutta_4, dt=0.1, maxt=10):
    y = initial_condition                    # Initial conditions
    ys = [y]                                 # List with results

    t = 0                                    # Actual time
    ts = [t]                                 # List with times

    while t < maxt:
        y, t = integrator(model, y, t, dt)   # Step

        ys.append(y)                         # Store position
        ts.append(t)                         # Store time

    return ys, ts

def relaxation(y, t):
    return -y

ys, ts = ode_solve(relaxation, 1)
```

Po spuštění kód vyřeší diferenciální rovnici relaxace (17) a v proměnných **ys** a **ts** vrátí seznam y-ových hodnot a odpovídajících časových okamžiků t.

Kód funguje a dává smysluplné výsledky ($y(t)$ klesá s rostoucím časem k nule), zapíšeme tedy změny do „historie“. Stav repozitáře je

```
D:\Pavel\Documents\Fyzika\Vyuka\ODE>git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   ode.py

no changes added to commit (use "git add" and/or "git commit -a")
```

Soubor je ve stavu změněný (modified). Před zapsáním je nutné jeho stav změnit na k zapsání (staged) pomocí příkazu `git add *`. Lze také použít zkrácený příkaz

```
git commit -a
```

který všechny změněné soubory převede do stavu k zapsání a rovnou zapsání provede. Změnu označním popisem „Eulerova metoda 1. řádu“.

```
D:\Pavel\Documents\Fyzika\Vyuka\ODE>git commit -a
hint: Waiting for your editor to close the file...
[main 509b08b] Eulerova metoda 1. řádu
 1 file changed, 27 insertions(+)
```

Dobrá programátorská praxe je zapisovat vždy jen takový kód, který lze spustit, tj. který neobsahuje žádné syntaktické chyby.

2.1.3.5 Přidání dalšího souboru do projektu

V programu na řešení diferenciálních rovnic je dobré oddělit univerzální výpočetní funkce od funkcí specifických pro daný model. Vytvoříme tedy nový soubor **`relaxation.py`**, do kterého přesuneme kód

```
import ode

def relaxation(y, t):
    return -y

ys, ts = ode.ode_solve(relaxation, 1)
```

Nesmíme zapomenout náš „modul“ naimportovat pomocí příkazu `import ode`. Program lze spustit a funguje, je tedy dobrý čas tyto změny zapsat. Nejdříve se ale podíváme na stav repozitáře:

```
D:\Pavel\Documents\Fyzika\Vyuka\ODE>git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   ode.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        __pycache__/
        relaxation.py

no changes added to commit (use "git add" and/or "git commit -a")
```

Soubor **`ode.py`** se změnil, soubor **`relaxation.py`** byl vytvořen a zatím je ve stavu nesledován. Navíc se ještě vytvořila celá složka **`__pycache__/`**.[7] Jedná se o pomocnou složku, kterou verzovat nechceme. Toho se docílí buď tím, že ji ponecháme v nesledovaném stavu (untracked) a začneme sledovat jen nový soubor. Tento postup však znemožní používání hromadných příkazů `git add *` či `git commit -a`. Gitu lze speciálně naznačit, jaké soubory a jaké složky jsou jen pomocné, a má je tedy ignorovat. Za tím účelem se vytvoří speciální soubor pojmenovaný **`.gitignore`**, do kterého zapíšeme následující pravidlo

[7] Pro urychlení provádění celého programu ukládá do tohoto adresáře interpret Pythonu speciálně předpřipravené používané moduly.

```
__pycache__/
```

a uložíme. Stav repozitáře se změní následovně:

```
D:\Pavel\Documents\Fyzika\Vyuka\ODE>git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   ode.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore
        relaxation.py

no changes added to commit (use "git add" and/or "git commit -a")
```

Pomocná složka už není uvedena mezi nesledovanými soubory a Git ji ignoruje. Můžeme již připravit k zapsání všechny nové a změněné soubory pomocí hromadného příkazu `git add *` a následně zapsat všechny změn pomocí `git commit`, přičemž změnu pojmenuji „Oddělení řešitele ODE a modelu“.

2.1.3.6 Zobrazení historie revizí

Historie zobrazíme příkazem

```
git log
```

```
D:\Pavel\Documents\Fyzika\Vyuka\ODE>git log
commit 419e936eac9715111090af19b7e347b31a12ff29 (HEAD -> main)
Author: Pavel Stránský <pcfyzika@pavelstransky.cz>
Date:   Tue Mar 30 10:42:13 2021 +0200

    Oddělení řešitele ODE a modelu

commit 509b08b5551c9b48362b7c31690e04f1947c5f6b
Author: Pavel Stránský <pcfyzika@pavelstransky.cz>
Date:   Tue Mar 30 10:38:36 2021 +0200

    Eulerova metoda 1. řádu

commit 1b694213d728693b000e269a82c972f26f4214c4
Author: Pavel Stránský <pcfyzika@pavelstransky.cz>
Date:   Tue Mar 30 10:35:05 2021 +0200

    Nový soubor - řešení ODE
```

Vypsaný seznam obsahuje všechny údaje o jednotlivých snímcích historie (zapsáních), tj. datum a čas, autora a jeho e-mail a hash kód podepisující zapsání.

Příkaz log má velké možnosti, jak formátovat a filtrovat informace o zapsáních, přičemž všechny jsou uvedeny v referenční příručce. Nejdůležitější jsou

- `git log --oneline` naformátuje vše hezčím způsobem (každé zapsání na jednom řádku) a z hashe zobrazí jen prvních 7 znaků, což stačí k identifikaci jednotlivých zapsání,
- `git log -2` vypíše jen poslední dva záznamy,
- `git log --stat` vypíše statistiky o jednotlivých verzích (počty změn a výpis změněných souborů),
- `git log --graph` zobrazí výsledek „graficky“, což se hodí v případě více větví (větve budou vysvětleny v sekci 2.1.5),
- `git log --follow ode.py` vypíše všechny historické změny v souboru **`ode.py`**,
- `git log --since "2 years 1 day 3 minutes ago"`

- `git log --until "2019-01-15"` vypíše všechny změny od, resp. do uvedeného časového údaje,
- `git log --author="Pavel"` vypíše všechny změny, které zapsal uživatel Pavel,
- `git log --grep="Euler"` vypíše všechny změny, které obsahují v popisu slovo „Euler“.

Uvedené filtry lze samozřejmě libovolně kombinovat.

2.1.4 .gitignore

Překladače programovacích jazyků často vytvářejí v adresáři 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 *` či `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 2.1: *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.*

2.1.5 Větve

Verzovací systém Git umožňuje pracovat paralelně na několika různých verzích kódu a mezi nimi dokáže efektivně přepínat. Větvení se hodí v případě, že chceme vyzkoušet více různých přístupů k řešení úlohy. Pro každé vytvoříme vlastní větev úprav, což dovolí řešení porovnávat a nakonec vybrat to, které vyhovuje nejvíce. Každou větev můžeme dále větvit. Perspektivní větve lze následně slučovat dohromady, neperspektivní lze nechat být.

Častá programátorská praxe je taková, že v hlavní větvi[8] se nachází vždy jen odladěná a otestovaná verze programu. Pro každou změnu v projektu se vytvoří nová větev. Na té se může pracovat hodinu, ale třeba několik měsíců. V této „vývojové“ větvi se změny provedou, odladí, pošlou testerům, a teprve poté se začlení do hlavní větve. Mezitím je stále k dispozici funkční program hlavní větve, který lze bez obav používat k ostrému provozu.

2.1.5.1 Vytvoření nové větve

Budeme pokračovat ve vývoji programu pro řešení obyčejných diferenciálních rovnic. Předpokládejme, že řešení pro jednu rovnici máme hotové a chceme ho rozšířit tak, bych fungovalo i na soustavu více diferenciálních rovnic 1. řádu, kterým se věnuje sekce 4. K tomu vytvoříme novou větev `odesystem`

```
git branch odesystem
```

a přepneme se na ni příkazem[9]

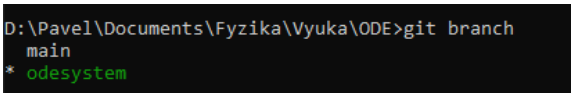
```
git switch odesystem
```

Zpět na hlavní větev se lze vrátit příkazem `git switch main`.[10]

2.1.5.2 Výpis větví

Seznam všech větví se vypíše příkazem

```
git branch
```



Aktivní větev je vyznačena hvězdičkou i barevně.

Další užitečná upřesnění výpisu větví jsou

- `git branch --merged` vypíše jen větve spojené s aktuální větví,
- `git branch --no-merged` vypíše jen větve nespojené s aktuální větví.

2.1.5.3 Změny a sloučení větví

Poté, co provedeme a odladíme změny související s rozšířením programu na výpočet soustavy diferenciálních rovnic ve větvi `odesystem`, můžeme tyto změny začlenit do hlavní větve `main`. K tomu slouží příkaz

```
git merge odesystem
```

spuštěný v hlavní větvi. Pokud během slučování nedojde k žádným konfliktům, vše proběhne hladce. Může se však stát, že byly provedeny změny na stejném místě jednoho souboru v obou větvích (konflikt). V tom případě Git označí konflikty v souborech a jejich stav změní na modifikovaný. Pak je potřeba tyto soubory projít, vybrat z konfliktních verzí jednu, případně kód upravit tak, aby odrážel obě konfliktní změny, a zapsat příkazem `git commit -a`. Tím je slučování dokončeno.

[8] Hlavní větev se donedávna pojmenovávala `master`, dnes se spíše prosazuje pojmenování `main`.
[9] Vytvoření nové větve a přepnutí se do ní lze jedním příkazem `git switch -c odesystem`.
[10] Git umožňuje přepínání mezi větvemi také pomocí příkazu `git checkout`.

2.1.5.4 Odstranění větve

Větev odstraníme prostým

```
git branch -d odesystem
```

Odstraňovat větve má smysl jen v případě, kdy jsme přesvědčeni, že změny v ní provedené byla slepá cesta.

2.1.6 Návrat k dřívějším verzím

Pozor, při neopatrném použití těchto příkazů hrozí ztráta dat! Pokud si budete následující příkazy zkoušet, doporučuji vám zazálohovat si celou složku s projektem včetně repozitáře (složky `.git`).

- `git commit --amend` umožňuje změnit popis posledního zapsání.
- `git restore soubor` vrátí změny uvedeného souboru na posledně zapsanou verzi (lze použít i hromadné `git restore *`[11]).
- `git reset HEAD~1` zruší poslední[12] zapsání, avšak soubory ve složce projektu ponechá tak, jak jsou.[13]
- `git reset --hard HEAD~1` zruší poslední zapsání a soubory projektu uvede do stavu, v jakém byly na začátku této poslední revize. Všechny provedené změny jsou nenávratně ztraceny. Pokud byl nějaký soubor v poslední revizi vytvořen, je smazán. Toto je nejnebezpečnější příkaz, proto používejte jen v opravdu odůvodněných případech.
- `git reset 946cb25` se vrátí na zapsání, jehož hash začíná uvedeným číslem.
- `git reset ORIG_HEAD` se vrátí na tu zapsanou verzi, na které jsme byli bezprostředně před použitím posledního příkazu `git reset`.
- `git checkout -b oldrev 946cb25` vytvoří novou větev nazvanou „oldrev“, ve které budou soubory ve stavu, kdy bylo provedeno zapsání s uvedeným hashem (hash zjistíme například pomocí `git log --oneline` popsaného v sekci 2.1.3.6).
- `git checkout 946cb25` vrátí se do tohoto zapsání, ale ponechá historii tak, jak je.[14]. K návratu zpět stačí použít příkaz pro přepnutí do hlavní větve `git switch main`.

2.1.7 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. Vlastníkem firmy vyvíjející tento systém je Microsoft. Z toho důvodu je základní práce s ním integrována i co vývojového prostředí Visual Studio Code.
- GitLab: Open source řešení, základní verze rovněž zdarma, k pokročilé verzi má MFF UK licenci.

[11] `git restore *` se chová podobně jako `git reset --hard`.
[12] Číslo udává, o kolik revizí se vracíme zpět.
[13] HEAD vlastně značí ukazatel na poslední zapsání.
[14] Tzv. Detached HEAD state.

Oba 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. 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/PCInPhysics2021`. 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/PCInPhysics2021`

  V aktuální složce vytvoří podsložku `PCInPhysics` a do ní stáhne celý repozitář. V případě těchto zápisků se jedná o tento soubor v LaTeXu, výsledné PDF, EPS či PNG 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 `main`, 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 umístěný na GitHubu.

- `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 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 2.2: *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 2.3: *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`.

2.1.8 Cheat Sheet

Pro snadnou orientaci v použití Gitu bez nutnosti pamatovat si všechny příkazy je dostupný Git Cheat Sheet nebo tento interaktivní web.

2.2 Programovací jazyk Python

Python je v dnešní době velmi populární programovací jazyk, který pronikl do spousty rozmanitých odvětví, a to zejména díky bohatosti a pokročilosti knihoven, které jsou vyvíjeny komunitou a jsou díky tomu aktuální a odladěné (či průběžně odlaďované). Python existuje na mnoha platformách od osobních počítačů po mikrokontroléry a dá se v něm naprogramovat téměř cokoliv: pokročilé vědecké výpočty používající nejnovější numerické algoritmy, dobře vypadající grafy, statistická analýza, analýza velkých dat a strojové učení, ale také webové služby, okénkové programy či editace obrázků a videí. Je běžné, že i komerční programy obsahují Python coby skriptovací jazyk, čímž umožňují uživateli používat své vlastní kódy „uvnitř“ komerčního produktu a integrovat různé programy mezi sebou.[15]

Python lze charakterizovat jako jazyk:

- *Interpetovaný*, což znamená, že provádění programů probíhá přímo ze zdrojového kódu.[16] Ke spouštění kódů je tedy potřeba mít nainstalovaný interpret, což jsme učinili v sekci 1.1. Pokud kód obsahuje syntaktickou chybu, je odhalena až ve chvíli, kdy se na ni interpret při provádění kódu narazí. U interpretovaných jazyků se neztrácí čas překladem do strojového kódu a je pro ně přirozené dynamické typování proměnných. Tím, že interpet čte text kódu příkaz po příkazu, je provádění programů pomalejší, ale vzhledem k tomu, že velká část časově náročných funkcí a knihoven bývá napsána v rychlejších programovacích jazycích, není to zásadní nevýhoda.

  Opakem intepretovaných jazyků jsou jazyky kompilované, přičemž kompilace se provádí buď přímo do strojového kódu daného procesoru[17] nebo do mezikódu. Ten ke spuštění vyžaduje dodatečnou kompilaci, která však může být vysoce optimalizovaná přímo pro danou hardwarovou konfiguraci použitého počítače.[18]

- *Dynamicky typovaný*, což znamená, že proměnná nemusí mít předem daný typ a typ proměnné se může za běhu programu dokonce měnit.

- *S automatickou správou paměti*, takže programátor se nemusí starat o alokování a uvolňování paměti. Python při inicializaci proměnné paměť alokuje automaticky a ve chvíli, kdy proměnnou přestaneme používat, paměť uvolní.[19]

- V Pythonu lze pythonovským způsobem používat tři základní programovací paradigmata: *procedurální, objektové a funkcionální.*

Python klade důraz na jednoduchost, stručnost a čitelnost kódu. Filosofie programovacího jazyka je shrnuta v Zenu Pythonu,[20] se kterým vám doporučuji se seznámit.

2.2.1 Vytvoření a spuštění kódu

Ve Visual Studio Code vytvoříme nový soubor a uložíme si ho s příponou `*.py`. Podle této přípony VS Code pozná, že se jedná o pythonovský kód a použije k práci s ním správný doplněk.

Hotový kód lze spustit ve VS Code jedním z následujících tří způsobů.

1. *Kliknutím na zelenou šipku na nástrojové liště*: Tímto způsobem se spustí celý soubor kódu.

2. *Stiskem* `F5` (a výběrem Python File): Kód se spustí v režimu ladění (debugger). Zastaví se na každém kontrolním bodě (breakpoint, který se vloží na vybranou řádku kódu klávesou `F9`) nebo na každé chybě. Pro pokračování provádění kódu stačí stisknout buď znovu `F5`, nebo `F10` či `F11` pro jeden krok.

[15] Jako příklad poslouží nejnovější verze programu Mathematica, Origin či SAS.
[16] Mezi další známé interpetované jazyky patří například JavaScript nebo PHP.
[17] Například C/C++, Pascal nebo Fortran.
[18] Zde se jedná například o jazyky Java, C# a celý .NET framework nebo Julia.
[19] Algoritmus se nazývá *Garbage collection.*
[20] Zen se také vypíše, pokud do svého kódu zadáte `import this`.

3. *Označením části kódu a stiskem Ctrl+Enter*: V okně `TERMINAL` spustí REPL Pythonu a v něm označenou část kódu. REPL se po provedení kódu nezavře, takže v něm lze psát dodatečné příkazy. Pro ukončení REPL do něj stačí napsat `exit()` nebo v něm stisknout `Ctrl+Z` a potvrdit. REPL musí být ukončen, než se použije jakákoliv z dvou dříve popsaných metod spuštění kódu.

2.2.2 Základy syntaxe Pythonu

Soubor se vzorovými příklady najdete v repozitáři ke cvičení: `introduction.py`.[21] Doporučuji vám si ho stáhnout nebo jeho obsah překopírovat a důsledně si ho řádek po řádku projít a spouštět nejlépe pomocí označení a stiskem `Ctrl+Enter`, jak bylo popsáno v předchozí sekci. Některé části kódu odkazují na proměnné zavedené v dřívější části souboru, proto kód procházejte postupně od začátku do konce.

Z vzorového kódu bych vypíchl následující body:

1. *Základní datové typy a operátory:* Python obsahuje jen dva základní číselné typy: `int` a `float`. Zbytek je podobný jako v jiných programovacích jazycích.

2. *Formátování řetězců:* Python umožňuje několik různých způsobů, jak formátovat řetězce (vkládat do nich čísla či jiné proměnné). Doporučuji se seznámit s velmi čitelným způsobem pomocí f-řetězce. Pro více informací k formátování řetězců doporučuji tento výborný návod.

3. *Proměnné a kolekce:* Důležité standardní kolekce jsou *seznam* (list) [...] a *slovník* (dicctionary) {...}. Dále existuje typ *n-tice* (tuple) (...) a *množina* (set) {...}. O jaký typ kolekce se jedná poznáte podle typu závorek, kterými je uzavřena. Python nabízí bohaté možnosti indexování prvků kolekcí.

   Python nemá typ „řada“ (jedno či vícerozměrný soubor hodnot stejných typů). Ten je implementován až v rozšiřujících knihovnách, například v knihovně **`numpy`**, které se budeme věnovat později.

4. *Podmínky a cykly*: Příkaz uvozující blok kódu vždy končí znakem „:“. Každý blok je definován svým odsazením, které musí být na každém řádku stejné (tj. musí obsahovat stejný počet odsazujících znaků, mezi které patří buď mezera nebo tabulátor). Konvence je používat k odsazení 4 mezery.

   Cykly jsou možné pouze přes iterovatelné objekty. Pro cyklus přes přirozená čísla (indexy) musíme vytvořit odpovídající iterovatelný objekt příkazem `range`. V drtivé většině se lze při programování v Pythonu indexům zcela vyhnout.

5. *Funkce*: Python umožnuje velkou variabilitu co se týče argumentů funkce a návratových hodnot díky své funkci automatického sbalení a rozbalení kolekcí. Funkce se navíc chová jako objekt, lze ji tedy přiřadit jakékoliv proměnné. To je jeden z prvků funkcionálního programování.

   Funkcionální programování také obsahuje koncept anonymní funkce, což je jednoduchá funkce definovaná na jednom řádku, která nemá vlastní jméno. V Pythonu se vytváří pomocí klíčového slova `lambda`.

6. *Moduly*: Při programování je důležité vhodně strukturovat kód. Python obsahuje jednoduchý koncept modulů, přičemž každý soubor s příponou `*.py` lze použít jako modul. Modul je vlastně speciální typ objektu.

   Dobře se seznamte se způsoby, jak modul načíst a používat, jelikož většina funkcí Pythonu se nachází právě v modulech. Jedná se například o matematické funkce v modulu **`math`** nebo rozšiřující knihovny **`numpy`**, **`scipy`** a **`matplotlib`**.

[21] Vzorový soubor je inspirován příkladem Nauč se Python v Y minutách.

7. *Třídy*: Python umožňuje objektově orientované programování. Práce s třídami se však v mnohém liší od striktně objektově orientovaných programovacích jazyků, jakými jsou například C++, C# nebo Java. Důležitý rozdíl je například v přístupnosti atributů (všechny atributy v Pythonu jsou veřejné). Rovněž dědičnost a dědění metod se chová odlišně. Dále je nutné pamatovat na to, že každá metoda třídy musí mít v deklaraci jako první argument odkaz na instanci, se kterou je volána (konvenčně se označuje `self`). Ve vzorovém kódu je k objektovému programování opravdu jen to nejnutnější minimum. Pokud chcete v Pythonu využívat objekty seriózně, doporučuji pročíst si odpovídající kapitolu v manuálu.

Uvedený kód s příklady obsahuje jen ty struktury jazyka Python, které budeme používat. V budoucích cvičeních se ještě seznámíte se *správou kontextu* (klíčové slovo `with`). Python umožňuje navíc

- ošetření chyb pomocí *výjimek* (klíčová slova `raise`, `try`, `except`, `finally`),
- tvorbu *generátorů* (klíčové slovo `yield`),
- *asynchronní programování*, korutiny a úkoly (klíčová slova `await`, `async`)
- či tvorbu a použití *dekorátorů*.

Pro plné ovládnutí jazyka vám doporučuji se v budoucnu s těmito koncepty seznámit, a to buď ze specializovaných přednášek, z tutoriálů a návodů na webu, nebo studiem cizích kódů.

Na oficiálních stránkách Pythonu najdete tutoriál k nejnovější verzi programovacího jazyka.

2.2.3 Pojmenování proměnných a formátování

Formátování kódu Pythonu je celkem volné. Povinné je dodržet jen správné odsazení bloků. Pro snazší čitelnost kódu byla nicméně vytvořena řada doporučení, která najdete souhrnně na stránce PEP 8.[22] Python obsahuje dokonce knihovnu `autopep8`, která váš zdrojový soubor podle těchto pravidel naformátuje. Automatické formátování podle PEP 8 lze nastavit i ve Visual Studiu Code.

Pro pojmenování proměnných existuje jediné závazné pravidlo, které zní, že indentifikátor se nesmí shodovat s žádným z 35 klíčových slov jazyka. Kromě toho jsou ve výše uvedeném dokumentu další nezávazná pravidla k pojmenování proměnných:[23]

- *Proměnné a funkce* se doporučuje pojmenovávat malými písmeny a jednotlivá slova spojovat podtržítky, tzv. underscore style (například `pocet_bodu`).
- *Konstanty* se doporučuje pojmenovávat velkými písmeny a jednotlivá slova spojovat podtržítky (například `PLANCKOVA_KONSTANTA`).
- *Třídy* se doporučuje pojmenovávat tak, že jednotlivá slova názvu začínají velkým písmenem a mezi nimi není žádný znak, tzv. Pascal style (například `PostovniAdresa`).
- *Moduly* se doporučuje pojmenovávat krátkými výstižnými názvy složenými jen z malých písmen.
- Kvůli čitelnosti je dobré se vyhnout jednopísmenným označením `l`, `I` a `O`, jelikož takto označené proměnné mohou být snadno zaměněny s čísly `1` a `0`.

Pokud vám vyhovuje jiný styl pojmenovávání, lze ho použít. Je však dobré být v pojmenovávání konzistentní napříč celým projektem.

[22] PEP = **P**ython **E**nhancement **P**roposal

[23] Čitelnosti různých stylů pojmenování proměnných se věnují i seriózní vědecké články, například http://www.cs.kent.edu/~jmaletic/papers/ICPC2010-CamelCaseUnderScoreClouds.pdf.

Častá otázka je, zda proměnné označovat *česky či anglicky*. Jelikož nikdy nevíte, kdo v dnešním propojeném světě bude chtít váš kód použít a třeba i upravit pro své potřeby, doporučuji používat anglická pojmenování.

Snadná čitelnost kódu je podpořena i vhodným psaním *komentářů*. Na drouhou stranu je vhodné vycházet z předpokladu, že dobře napsaný kód je čitelný i bez komentářů. Čitelnost totiž zaručují zejména vhodně a výstižně označené proměnné a správně navržená struktura kódu. Komentovat je dobré jen ty části kódu, kde se používá nějaký ne všeobecně známý trik nebo postup. Nadbytek komentářů je velmi těžké udržovat při úpravách a může vést i k tomu, že po několika změnách kódu nebudou komentáře v souladu s tím, co kód vykonává.

Naopak doporučené je nešetřit popisem, co dělají jednotlivé funkce, jaké jsou jejich argumenty a co vracejí na výstupu. K tomu slouží komentář typu *docstring*, který je vysvětlený v souboru `introduction.py` v sekci o funkcích.[24] Použití *docstringu* usnadní i tvorbu a správu doprovodné dokumentace k celému projektu. Existují například nástroje, které vám z těchto komentářů vytvoří strukturované webové stránky.

2.2.4 Řady (knihovna numpy)

Jak bylo zmíněno v předchozí sekci, definice jazyka Python neobsahuje typ *řada*, tj. jedno či vícerozměrně pole hodnot stejného typu.

Základy práce s řadami jsou vysvětleny přímo v souboru `arrays.py` v repozitáři.

2.2.5 Grafy (knihovna matplotlib)

K vykreslení grafů slouží knihovna `Matplotlib`. V základní instalaci Pythonu není obsažena, je nutné ji doinstalovat podle návodu v sekci 1.1.1.

Jednoduchý příklad vykreslení grafů je v souboru `plot.py` v repozitáři.

2.2.6 Pseudonáhodná čísla

V Pythonu lze pseudonáhodná čísla generovat pomocí několika knihoven.

- Pro základní použití je k dispozici knihovna `random`. 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`.
- Ke generování řad či vícerozměrných objektů, jejichž elementy jsou náhodná čísla, lze použít i knihovnu **`numpy`**, viz sekce 2.2.4 a soubor s příklady `numpy.py`.
- Pro pokročilejší použití je knihovna `numpy.random`. Ta umožňuje zvolit vlastní generátor pseudonáhodných čísel, generovat čísla z celé řady statistických rozdělení a generovat naráz celé vektory či matice.

[24] Stejně jako v mluvených jazycích se zlepšujete čtením literatury, v programovacích jazycích pomáhá čtení cizích kódů. Pro přiklad použití komentářů se koukněte třeba na zdrojový soubor funkce `optimize` z knihovny **`scipy`**.

- `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.

3 Obyčejné diferenciální rovnice 1. řádu

V této části se budeme věnovat řešení jedné diferenciální rovnice prvního řádu,

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

s počáteční podmínkou

$$y(t_0) = y_0. \tag{2}$$

Zde $y = y(t)$ je hledaná funkce a t je nezávisle proměnná.

Numerické řešení diferenciální rovnice spočívá v nahrazení infinitezimálních přírůstků přírůstky konečnými:

$$\frac{\Delta y}{\Delta t} = \phi(y,t) \tag{3}$$

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.

3.1 Pár důležitých pojmů

- **Explicitní algoritmy:** K výpočtu hodnoty funkce y_{i+1} se vyžadují pouze hodnoty z aktuálních a minulých kroků, tj. y_i, y_{i-1}, atd.

- **Jednokrokové algoritmy:** K výpočtu hodnoty funkce y_{i+1} v následujícím kroku vyžadují pouze znalost hodnoty funkce v aktuálním kroku y_i. Rozepsáním (3) dostaneme

$$\boxed{y_{i+1} = y_i + \underbrace{\phi(y_i,t_i)}_{\phi_i}\,\Delta t}, \tag{4}$$

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

- **Lokální diskretizační chyba:**

$$\mathcal{L} = y(t+\Delta t) - y(t) - \phi(y(t),t)\Delta t, \tag{5}$$

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

- **Akumulovaná diskretizační chyba:**

$$\epsilon_i = y_i - y(t_i) \tag{6}$$

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

$$L(\Delta t) = \mathcal{O}(\Delta t^{p+1}). \tag{7}$$

- **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.

3.2 Eulerova metoda 1. řádu

$$\phi_i = f(y_i, t_i), \tag{8}$$

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.

3.3 Eulerova metoda 2. řádu

$$\begin{aligned} k_1 &= f(y_i, t_i) \\ k_2 &= f\left(y_i + k_1 \Delta t, t + \Delta t\right) \\ \phi_i &= \frac{1}{2}\left(k_1 + k_2\right), \end{aligned} \tag{9}$$

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:

$$\begin{aligned} k_1' &= f(y_i, t_i) \\ k_2' &= f\left(y_i + k_1' \frac{\Delta t}{2}, t + \frac{\Delta t}{2}\right) \\ \phi_i &= k_2' \end{aligned} \tag{10}$$

3.4 Runge-Kuttova metoda 4. řádu

$$\begin{aligned} k_1 &= f(y_i, t_i) \\ k_2 &= f\left(y_i + k_1 \frac{\Delta t}{2}, t + \frac{\Delta t}{2}\right) \\ k_3 &= f\left(y_i + k_2 \frac{\Delta t}{2}, t + \frac{\Delta t}{2}\right) \\ k_4 &= f\left(y_i + k_3 \Delta t, t + \Delta t\right) \\ \phi_i &= \frac{1}{6}\left(k_1 + 2k_2 + 2k_3 + k_4\right) \end{aligned} \tag{11}$$

- 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.

3.5 Odvození metod

Metody se odvozují na základě rozvoje do Taylorovy řady. Volně řečeno platí, že kolik členů Taylorova rozvoje se vezme, takový bude řád metody p.

- *Eulerova metoda 1. řádu.* Odvození je triviální:

$$y(t+\Delta t) \approx y(t) + \underbrace{y'(t)}_{f(y,t)} \Delta t, \tag{12}$$

takže

$$y_{i+1} = y_i + f(y_i, t_i)\Delta t. \tag{13}$$

- *Eulerova metoda 2. řádu.*

$$\begin{aligned} y(t+\Delta t) &\approx y(t) + y'(t)\Delta t + \frac{1}{2} \underbrace{y''(t)}_{\approx \frac{y'(t+\Delta t)-y'(t)}{\Delta t}} \Delta t^2 \\ &\approx y(t) + f(y,t)\Delta t + \frac{1}{2}[f(\underbrace{y(t+\Delta t)}_{\approx y(t)+y'(t)\Delta t}, t+\Delta t) - f(y,t)]\Delta t \\ &\approx y(t) + \frac{1}{2}\left[f(y,t) + f(y(t) + f(y,t)\Delta t, t+\Delta t)\right]\Delta t, \end{aligned} \tag{14}$$

a odtud

$$y_{i+1} = y_i + \underbrace{\frac{1}{2}\left[f(y_i,t_i) + f(y_i + f(y_i,t_i)\Delta t, t_{i+1})\right]}_{\phi_i}\Delta t, \tag{15}$$

což je jen jinak napsané vztahy (9).

Druhá Eulerova metoda 2. řádu (10) se obdrží stejným způsobem, jen při aproximaci druhé derivace se použije poloviční krok:

$$y''(t) \approx \frac{y'\left(t + \frac{\Delta t}{2}\right) - y'(t)}{\frac{\Delta t}{2}}. \tag{16}$$

Úkol 3.1: *Dokončete odvození vztahů (10).*

Analogicky se postupuje při odvození metod vyššího řádu. Jeho složitost však narůstá exponenciálně s řádem metody. U Eulerovy metody 2. řádu se navíc ukázalo, že lze odvodit několik stejně dobrých vztahů. Tato volnost s řádem metody narůstá.

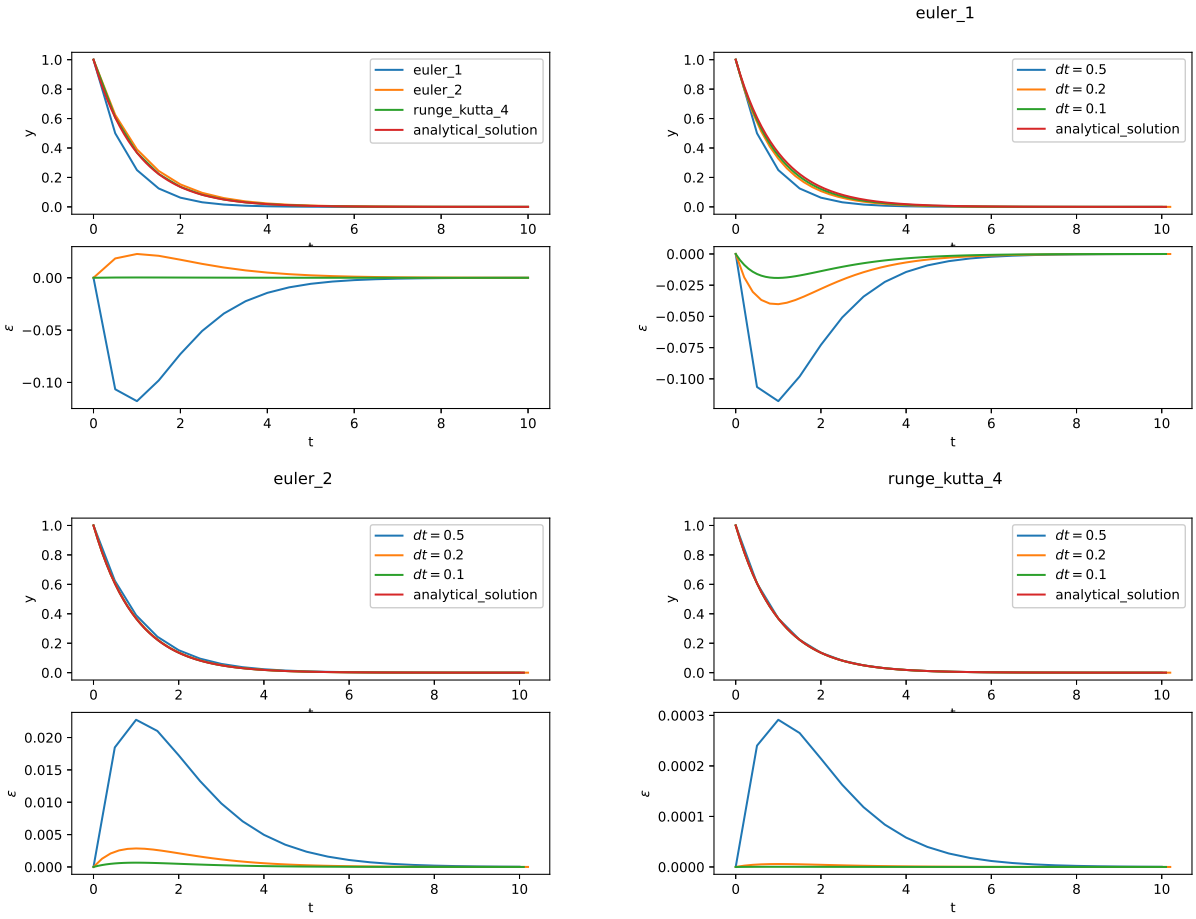
3.6 Domácí úkol na 23.3.2021

Úkol 3.2: *Naprogramujte Eulerovu metodu 1. a 2. řádu a Runge-Kuttovu metodu. Vyřešte numericky diferenciální relaxační rovnici*[25]

$$\frac{\mathrm{d}y}{\mathrm{d}t} = -y \tag{17}$$

s počátečními podmínkami $y_0 = 1$ (analytickým řešením je funkce e^{-t}). Integrační krok Δt ponechte jako volný parametr. Nakreslete grafy řešení $y(t)$ pro rozdílné hodnoty integračních kroků, například $\Delta t = 0.01$ a $\Delta t = 0.1$ pro čas $t \in \langle 0; 10 \rangle$.

Řešení 3.2: *Jedno možné řešení v souborech* `ode.py` *(integrační metoda),* `graphs.py` *(vykreslování grafů) a* `relaxation.py` *(hlavní soubor, který obsahuje definici modelu a volá všechny výpočetní a vykreslovací funkce). Na řešení této úlohy je také ukázán základ práce s větvemi v sekci 2.1.5.*



Obrázek 1: Integrace relaxační diferenciální rovnice (17) různými metodami a s různými časovými kroky.

- `ode.py`*: Modul napsaný maximálně obecně, aby ho bylo možné použít na řešení libovolné diferenciální rovnice, a také soustav diferenciálních rovnic probíraných v sekci 4.*
  - `euler_1`*: Integrační krok Eulerovy metody 1. řádu.*
  - `euler_2`*: Integrační krok Eulerovy metody 2. řádu.*
  - `runge_kutta_4`*: Integrační krok Runge-Kuttovy metody 4. řádu.*

[25] Rovnice popisuje například Newtonův zákon chladnutí tělesa, aktuální množství prvku při radioaktivním rozpadu nebo různé populační modely.

  - `ode_solve`: *Integruje diferenciální rovnici danou pravou stranou prvního parametru* `model` *s počáteční podmínkou předanou parametrem* `initial_condition`. *Integrační metodu lze specifikovat parametrem* `integrator`, *což je vlastně funkce* $\phi(y_i, t_i)$ *z rovnice* (4). *Délku kroku udává parametr* `dt`. *Funkce vrací pole hodnot řešení rovnice (soustavy rovnic) v jednotlivých časech a pole časů.*
  - `scipy_ode_solve`: *Integruje diferenciální rovnici pomocí funkce* `odeint` *z knihovny* `scipy.integrate`. *Pozor, parametr* `dt` *zde neznačí 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.*

- `graphs.py`: *Vykresluje grafy srovnávající různé metody nebo různé kroky výpočtu.*
  - `plot_solution_error`: *Vykreslí graf řešení diferenciální rovnice. Parametr* `analytical_solution` *je odkaz na přesné řešení dané diferenciální rovnice. Je-li zadán, vykreslí se do grafu dva panely: jeden s hodnotami numerického řešení, druhý s rozdílem řešení numerického a přesného* ϵ *(akumulovaná diskretizační chyba dle rovnice* (6)*).*
  - `plot_compare_methods`: *Vykreslí do jednoho obrázku řešení zadané diferenciální rovnice různými metodami. Z důvodu efektivnějšího kódu se v této funkci používá trochu složitější práce s panely, než jak je uvedeno ve vzorovém souboru* `plot.py`.
  - `plot_compare_steps`: *Vykreslí do jednoho obrázku řešení zadané diferenciální rovnice s různými integračními kroky.*
  - `plot_cummulative_errors`: *Řešení úlohy 3.3.*

- `relaxation.py`: *Soubor, který využívá integračních funkcí z modulu* `ode.py` *a vykreslovacích funkcí z modulu* `graphs.py` *pro řešení diferenciální rovnice pro relaxaci.*
  - *Vyřeší diferenciální rovnici různými metodami, řešení nakreslí do jednoho grafu a porovná s teoretickou funkcí.*
  - *Vyřeší diferenciální rovnici s různým časovým krokemm, řešení nakreslí do jednoho grafu a porovná s teoretickou funkcí.*
  - *Nakreslí graf k úloze 3.3.*

Příslušné grafy jsou zobrazeny na obrázku 1. Je vidět, že chyba opravdu klesá s řádem metody a že čím je metoda vyššího řádu, tím rychleji chyba klesá se zmenšujícím se krokem.

Úkol 3.3: *Rozšiřte kód tak, aby počítal průměrnou kumulovanou chybu*

$$\mathcal{E} = \sqrt{\frac{1}{n}\sum_{i=0}^{n-1}\left(y_i - y(t_i)\right)^2}, \tag{18}$$

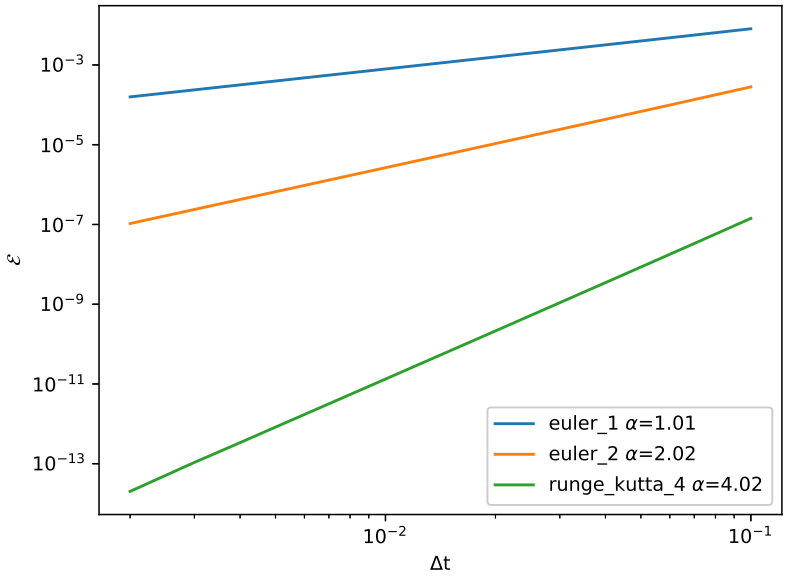
kde $y(t)$ *je analytické řešení diferenciální rovnice. Nakreslete závislost* $\mathcal{E}(\Delta t)$ *pro* $\Delta t \in \langle 0.002; 0.1\rangle$ *a pro různé metody. Jelikož očekáváme mocninnou závislost dle* (7), *kde exponent je tím větší, čím větší je řád metody, je výhodné graf* $\mathcal{E}(\Delta t)$ *kreslit v log-log měřítku. V Pythonu použijete místo* `plot(...)` *funkci* `loglog(...)` *z knihovny* `matplotlib.pyplot`. *Ověřte, že získané křivky jsou v souladu s řády použitých metod.*

Řešení 3.3: *Řešení této úlohy vykreslí funkce* `plot_cummulative_error` *z modulu* `graphs.py`. *Graf je zobrazen na obrázku 2. Body log-log grafu byla proložena přímka pomocí funkce lineární regrese* `linregress` *z knihovny* `scipy.stats`. *Nafitovaná směrnice* α *je uvedena v obrázku.*

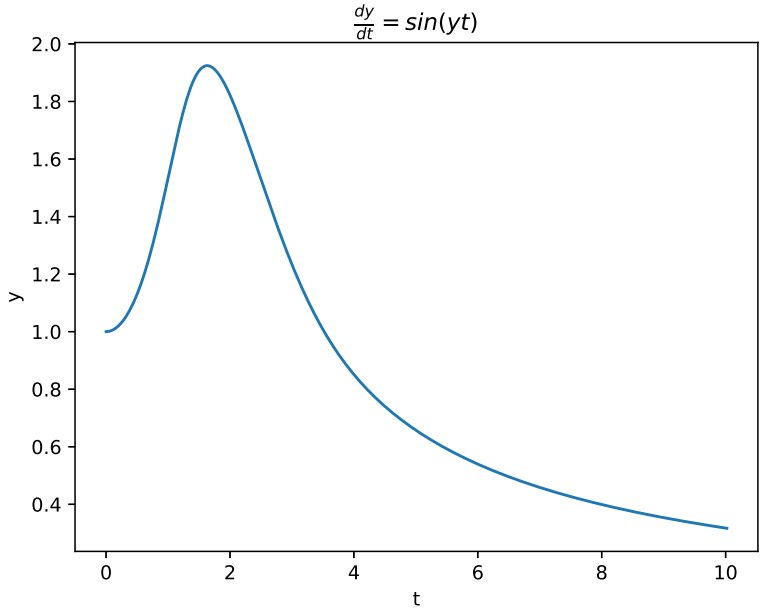
Úkol 3.4: *Pomocí naprogramovaných metod vyřešte nelineární diferenciální rovnici*

$$\frac{\mathrm{d}y}{\mathrm{d}t} = \sin(ty) \tag{19}$$

s počáteční podmínkou $y_0 = 1$, $t_0 = 0$ *a vykreslete graf jejího řešení.*



Obrázek 2: Závislost průměrné kumulované chyby (18) na délce kroku Δt vypočítaná a vykreslená pomocí funkce pro relaxační diferenciální rovnici.



Obrázek 3: Řešení diferenciální rovnice (19) s počáteční podmínkou $y(0) = 1$. Použitá metoda: Runge-Kutta, časový krok $\Delta t = 0.02$.

Řešení 3.4: *Řešení je v souboru* `sinyt.py`. *Graf vypočítané funkce je na obrázku 3.*

4 Soustavy diferenciálních rovnic 1. řádu

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

$$\frac{\mathrm{d}\boldsymbol{y}}{\mathrm{d}t} = \boldsymbol{f}(\boldsymbol{y}, t), \tag{20}$$

kde $\boldsymbol{y} = \boldsymbol{y}(t)$ je vektor hledaných funkcí a $\boldsymbol{y}(t_0) = \boldsymbol{y}_0$ vektor počáteční podmínky.

Příklad 4.1: *Pohybovou rovnici*

$$Ma = F(x), \tag{21}$$

kde M je hmotnost pohybujícího se tělesa, $x = x(t)$ jeho poloha a $a = a(t) = \mathrm{d}^2x/\mathrm{d}t^2$ zrychlení převedeme na dvě diferenciální rovnice prvního řádu zavedením rychlosti $v = v(t) = \mathrm{d}x/\mathrm{d}t$:[26]

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} x \\ v \end{pmatrix} = \begin{pmatrix} v \\ \frac{1}{M}F(x) \end{pmatrix}, \tag{22}$$

tj. vektor funkce pravých stran podle rovnice (20) je

$$\boldsymbol{f}(\boldsymbol{y}, t) = \begin{pmatrix} v \\ \frac{1}{M}F(x) \end{pmatrix} \tag{23}$$

kde $\boldsymbol{y} \equiv \begin{pmatrix} x \\ v \end{pmatrix}$.

Příklad 4.2: *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*

$$a = \frac{\mathrm{d}^2x}{\mathrm{d}t^2} = -x \qquad \Longleftrightarrow \qquad \frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} x \\ v \end{pmatrix} = \begin{pmatrix} v \\ -x \end{pmatrix} \tag{24}$$

Úkol 4.1: *Převeďte na soustavu obyčejných diferenciálních rovnic prvního řádu rovnici třetího řádu pro Hiemenzův tok*

$$x''' + xx'' - x'^2 + 1 = 0. \tag{25}$$

Řešení 4.1: *Hledaná soustava diferenciálních rovnic je*

$$\frac{\mathrm{d}}{\mathrm{d}t}\begin{pmatrix} x \\ v \\ a \end{pmatrix} = \begin{pmatrix} v \\ a \\ -xa + v^2 - 1 \end{pmatrix}. \tag{26}$$

Drtivá většina knihoven a algoritmů pro integraci obyčejných diferenciálních rovnic počítá s rovnicemi ve tvaru (20).

4.1 Symplektické algoritmy

Jedná se o speciální algoritmy navržené pro řešení pohybových differenciálních 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 nejčastěji *Verletův algoritmus*. Pro diferenciální rovnici 2. řádu ve tvaru

$$M\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = F(x), \tag{27}$$

[26] Tato soustava rovnic odpovídá Hamiltonovým pohybovým rovnicím, se kterými se seznámíte v přednášce Teoretická mechanika.

což je pohybová rovnice pro jeden hmotný bod o hmotnosti M, na který působí časově neproměnná síla $F(x)$, má Verletův algoritmus tvar

$$\begin{aligned} x_{i+1} &= x_i + v_i \Delta t + \frac{1}{2} a_i \Delta t^2, \\ v_{i+1} &= v_i + \frac{1}{2} \left(a_{i+1} + a_i \right) \Delta t, \end{aligned} \tag{28}$$

kde $a_i \equiv F(x_i)/M$ je zrychlení částice.

Verletův algoritmus se používá nejčastěji v molekulární dynamice k simulaci pohybu velkého množství vzájemně interagujících částic. Jedná se o velmi rychlou a efektivní metodu.

Řád Verletova algoritmu je $p = 2$. Symplektické algoritmy s vyšším řádem existují, avšak v praxi se nepoužívají.

4.2 Domácí úkol na 30.3.2021

Úkol 4.2: *Rozšiřte kód naprogramovaný v úkolu 3.2 tak, aby fungoval pro libovolně velkou soustavu obyčejných diferenciálních rovnic. Využijte k tomu funkce pro práci s řadami z knihovny* **numpy** *popsané v sekci 2.2.4. Vyřešte diferenciální rovnici harmonického oscilátoru*

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2} = -x \tag{29}$$

s počátečními podmínkami $x_0 = 0$, $x_0' \equiv v_0 = 1$, $t_0 = 0$ a porovnejte řešení různými metodami s analytickým řešením $x(t) = \sin t$. Jako časový krok volte například $\Delta t = 0.1$ a $\Delta t = 0.01$ a počítejte na časovém intervalu $t \in \langle 0; 30 \rangle$.

Řešení 4.2: *Metody jsou naprogramovány v souborech* `ode.py` *(integrační metoda),* `graphs.py` *(vykreslování grafů) a jsou podrobně diskutování v řešení úlohy 3.2. Soubor* `ho.py` *obsahuje pravé strany diferenciálních rovnic harmonického oscilátoru (funkce* `ho`*) a volá všechny výpočetní a vykreslovací funkce. Navíc obsahuje následující funkce:*

- `ho_solution`*: Přesné řešení pohybové rovnice harmonického oscilátoru s počáteční podmínkou $x_0 = 0, x_0' = 1, t_0 = 0$, jímž je funkce $x(t) = \sin t$.*
- `ho_energy`*: Pro vstupní polohu a rychlost (nebo pole poloh a rychlostí) vrátí energii harmonického oscilátoru.*

Příslušné grafy jsou zobrazeny na obrázku 4.

Úkol 4.3: *Naprogramujte Verletův algoritmus (28). Ukažte, že zatímco při použití Eulerovy metody nebo Runge-Kuttovy metody energie systému v průběhu výpočtu roste, Verletův algoritmus energii zachovává. Energie bezrozměrného harmonického oscilátoru (29) je dána vzorcem*

$$E = \frac{1}{2}\left(x^2 + v^2\right). \tag{30}$$

Řešení 4.3: *Symplektický Verletův algoritmus je naprogramován v souboru* `symplectic.py`*. Toto vzorové řešení bude fungovat* jen *pro jednu pohybovou rovnici, tj. pro jednu diferencální rovnici původně druhého řádu přepsanou na dvě diferenciální rovnice prvního řádu, přičemž první rovnice musí být pro souřadnici, druhá pro rychlost. Řešení lze samozřejmě rozšířit na více pohybových rovnic.*

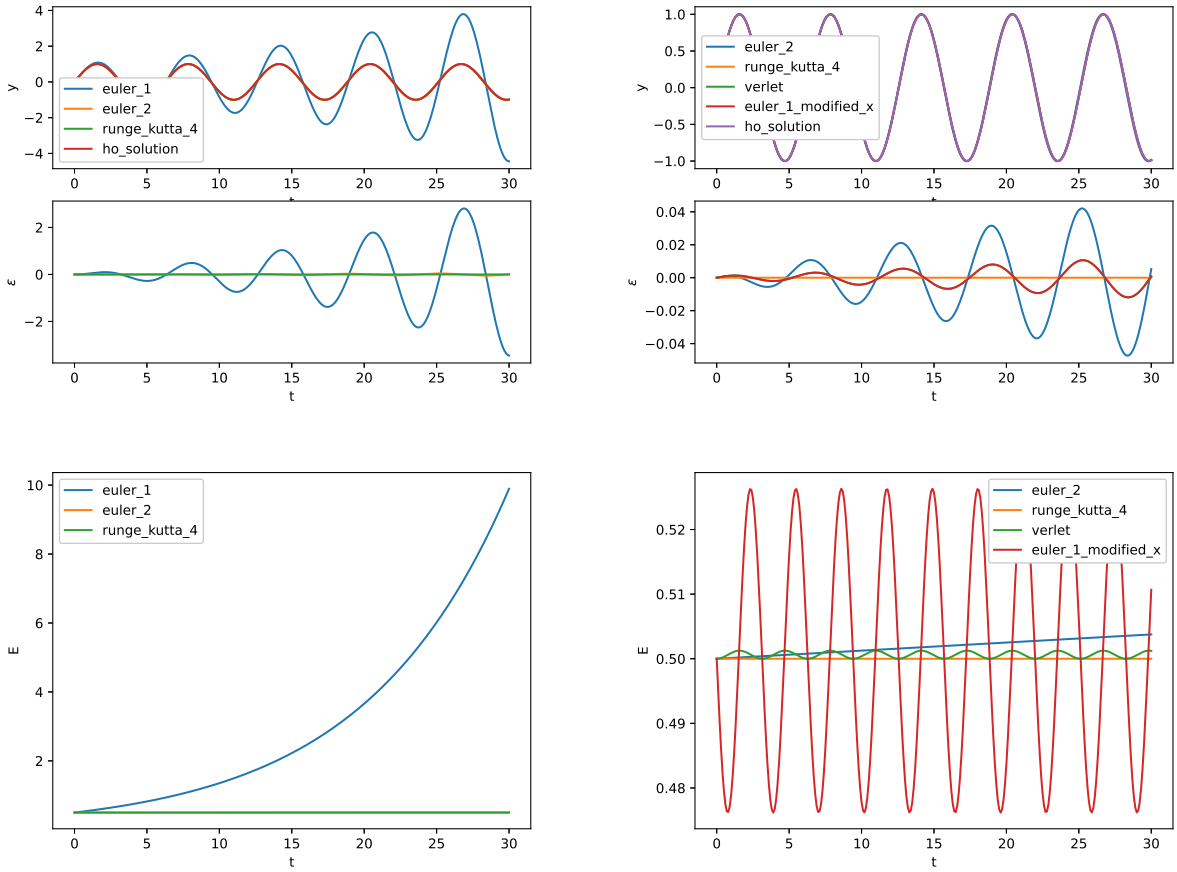
V souboru `symplectic.py` *je i funkce* `plot_energy`*, která vykreslí graf závislosti $E(t)$. Energie je znázorněna v panelech na třetím řádku na obrázku 4. Harmonický oscilátor je konzervativní systém (zachovává energii), pozorovaná rostoucí energie je způsobena nepřesností integračních metod. U symplektických algoritmů energie slabě osciluje okolo střední hodnoty, která však i pro velmi dlouhé časy zůstává na přesné hodnotě $E = \frac{1}{2}$.*

Úkol 4.4: *Eulerovu metodu 1. řádu lze pro soustavy dvou diferenciálních rovnic 1. řádu vylepšit následující záměnou:*

$$\begin{aligned} x_{i+1} &= x_i + v_i\Delta t \\ v_{i+1} &= v_i - x_i\Delta t \end{aligned} \quad \longrightarrow \quad \begin{aligned} x_{i+1} &= x_i + v_i\Delta t \\ v_{i+1} &= v_i - x_{i+1}\Delta t \end{aligned} \tag{31}$$

(vypočítáme x_{i+1} a tuto hodnotu použijeme namísto hodnoty x_i pro výpočet rychlosti v_{i+1}). Naprogramujte tuto metodu a pomocí výsledků úlohy 3.3 ukažte, že pro harmonický oscilátor se jedná o metodu 2. řádu.

Řešení 4.4: *„Vylepšené“ Eulerovy metody jsou naprogramovány v souboru* `symplectic.py`*. Metoda s předbíhající se souřadnicí je označena* `euler_1_modified_x`*, metoda s předbíhající rychlostí pak*



Obrázek 4: Integrace diferenciální rovnice harmonického oscilátoru (29) různými metodami. Časový krok je $\Delta t = 0.1$. *Levý sloupec:* všechny metody. *Pravý sloupec:* Bez eulerovy metody 1. řádu a včetně symplektických metod. *1. řádek:* hodnoty $y(t)$. *2. řádek:* Akumulované diskretizační chyby dle (6). Pro Eulerovu metodu 1. řádu je obalová křivka divergence numerického od analytického řešení očividně exponenciální v čase. *3. řádek:* energie (30). Pro Eulerovy metody energie roste. Energie se mění i pro Runge-Kuttovu metodu (pro tento systém energie s časem klesá), avšak změna je řádově menší než pro ostatní metody, a tudíž není na grafech při daném měřítku svislé osy vidět. Naopak pro Verletův algoritmus a pro „předbíhající“ Eulerovu metodu energie osciluje okolo počáteční energie $E = \frac{1}{2}$.

`euler_1_modified_v`. *Obě metody dávají velmi podobné výsledky, v grafech je proto uvedena jen první z nich.*

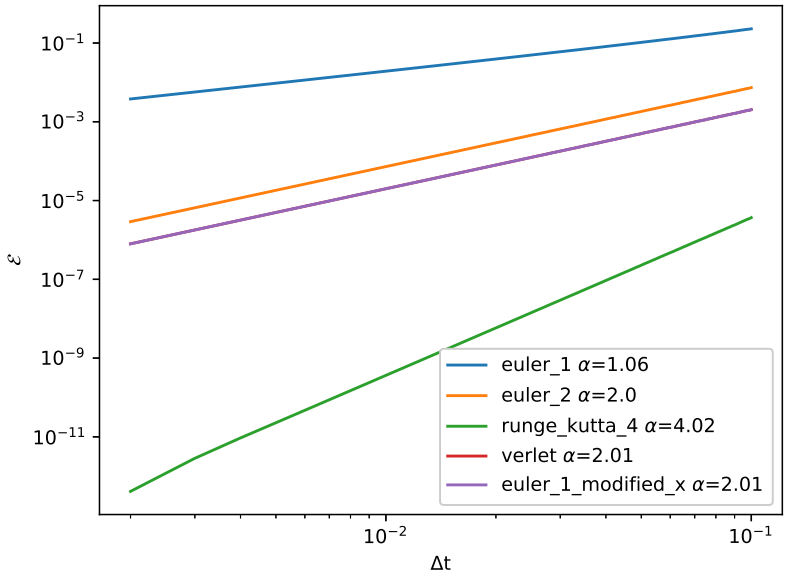
Řešení diferenciální rovnice harmonického osciátoru je znázorněno v pravém sloupci obrázku 4. Je vidět, že so se přesnosti týče, vylepšená Eulerova metoda je srovnatelná s Verletovou metodou. Energie však osciluje trochu víc.

Řád metody lze určit na základě obrázku 5. Jak u Verletovy metody, tak u vylepšené Eulerovy metody klesá průměrná kumulovaná chyba s druhou mocninou délky časového kroku δt, jedná se tedy o metody 2. řádu.

Úkol 4.5: *Pohrajte si s řešením rovnice pro klesající exponenciálu*

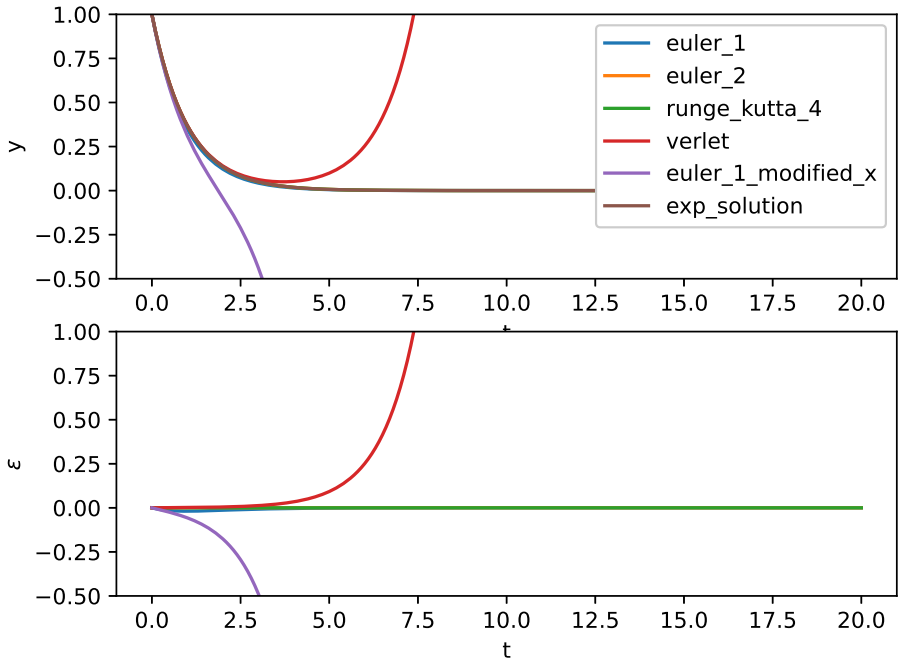
$$\frac{\mathrm{d}^2 x}{\mathrm{d}t^2} = x \tag{32}$$

s počátečními podmínkami $x_0 = 1$, $x'_0 = -1$. Přesvědčte se, že Verletova metoda a vylepšená Eulerova metoda z předchozího úkolu jsou nestabilní — pro tuto rovnici v relativně krátkém čase začnou řešení exponenciálně divergovat.



Obrázek 5: Závislost průměrné kumulované chyby (18) na délce kroku Δt vypočítaná a vykreslená pro soustavu diferenciálních rovnic pro harmonický osciátor. Zobrazeny jsou i symplektické metody.

Řešení 4.5: *Řešení rovnice (32) je v souboru* `exp.py`. *Soustava rovnic je analogická k soustavě harmonického oscilátoru — liší se pouze znaménkem. Systém popsaný touto rovnicí* není konzervativní *— nelze nadefinovat zachovávající se veličinu, která by měla význam energie.*



Obrázek 6: Totéž jako v obrázku 4, avšak pro exponenciálně klesající systém daný rovnicí (32). Symplektické algoritmy jsou nestabilní.

Z obrázku 6 je vidět, že symplektické algoritmy jsou nestabilní, a tudíž nejsou na tento typ úlohy vhodné, což je pochopitelné, protože jsou navrženy pouze pro energii zachovávající systémy. Příčinu

nestability lze nahlédnout i z obecného řešení rovnice (32),

$$y(t) = A\,\mathrm{e}^{t} + B\,\mathrm{e}^{-t}, \tag{33}$$

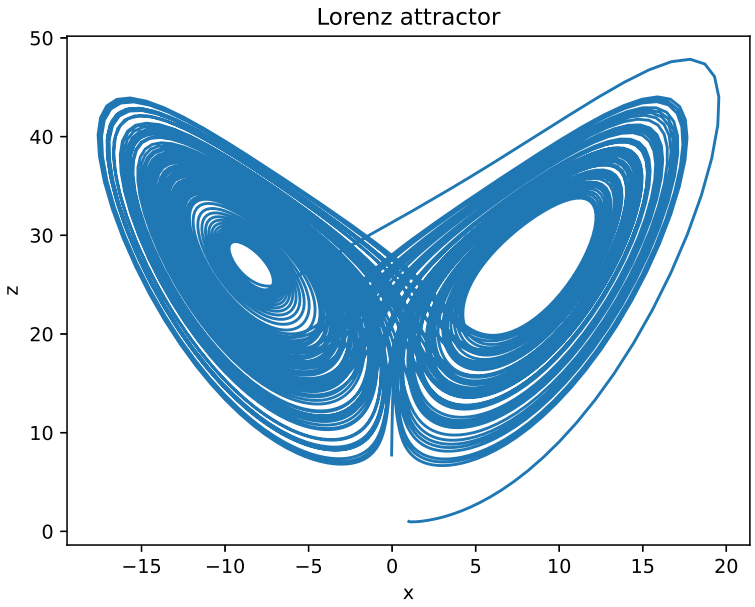
přičemž my speciálními počátečními podmínkami vybíráme pouze exponenciálně klesající řešení. Symplektické algoritmy však v určitou chvíli „překmitnou“ na exponenciálně rostoucí řešení, což způsobí pozorovanou divergovat.

Úkol 4.6: *Vyřešte nelineární soustavu tří diferenciálních rovnic pro jednoduchý Lorenzův model vedení tepla v atmosféře*

$$\begin{aligned} \frac{\mathrm{d}x}{\mathrm{d}t} &= \sigma(y - x), \\ \frac{\mathrm{d}y}{\mathrm{d}t} &= x(\rho - z) - y, \\ \frac{\mathrm{d}z}{\mathrm{d}t} &= xy - \beta z \end{aligned} \tag{34}$$

s hodnotami parametrů $\sigma = 10$, $\rho = 28$ a $\beta = 8/3$, počátečními podmínkami $(x_0, y_0, z_0) = (1, 1, 1)$ (na počátečních podmínkách zase tolik nezáleží), s krokem $\Delta t = 0.01$ a na časovém intervalu $t \in \langle 0, 100\rangle$. Vykreslete graf $z(x)$. Výsledná křivka je slavný Lorenzův podivný atraktor ve tvaru motýlích křídel, krerý zpopularizoval teorii klasického chaosu.

Řešení 4.6: *Řešení Lorenzova systému diferenciálních rovnic je jednoduchou aplikací kódu naprogramovaného v úloze 4.2 a lze ho nalézt v souboru* `lorenz.py`*. Příslušnou projekci trojrozměrné trajektorie do 2D roviny (x, z) lze nalézt na obrázku 7.*



Obrázek 7: Lorenzův systém (34) integrovaný pomocí Runge-Kuttovy metody 4. řádu na časovém intervalu $t \in \langle 0; 100\rangle$ s krokem $\Delta t = 0.01$.

4.3 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 dle potřeby měnit délku kroku (metody s adaptivním krokem).
- 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.

A hlavně, nyní již umíte jen pomocí sčítání a násobení vypočítat a nakreslit průběh goniometrické funkce sinus.

5 Náhodná procházka

Náhodná procházka je jeden ze základních prostředků, jak simulovat velké množství nejen fyzikálních procesů (například pohyb Brownovské částice, fluktuace akciového trhu, cestu 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 s se volí buď náhodná, nebo konstantní.

K simulování náhodných procesů slouží algoritmy generující pseudonáhodná čísla, což jsou čísla, která mají statistické vlastnosti blízké vlastnostem skutečných náhodných čísel, avšak jsou počítána jednoduchými deterministickými algoritmy. Náhodná čísla se generují od počáteční tzv. *násady* (seed), kterou lze explicitně zadat, a tím posloupnost náhodných čísel přesně zreprodukovat (díky tomu, že generující algoritmy jsou deterministické). Pokud násada není explicitně zadána, knihovny pro generování pseudonáhodných čísel většinou volí systémový čas, takže při každém spuštění programu dostáváme posloupnost odlišnou.

Základní funkce a postupy pro generování náhodných čísel v Pythonu jsou uvedeny v sekci 2.2.6.

5.1 Domácí úkol na 6.4.2021

Úkol 5.1: *Naprogramujte náhodnou procházku ve 2D rovině. Délku kroku volte konstantní $s = 1$, směr volte náhodně. Začněte z bodu $(x, y) = (0, 0)$ a procházku ukončete poté, co opustíte oblast tvaru čtverce o hraně délky $2a$. Uchovávejte celou procházku v poli či seznamu. Nakonec trajektorii vykreslete do grafu.*

Řešení 5.1: *Řešení je naprogramováno v souboru* `2d.py`.

- `random_direction_2d` *vrátí náhodný směr ve 2D rovině [generuje náhodný úhel ϕ, směr je dán jednotkovým vektorem se složkami $(\cos\phi, \sin\phi)$].*
- `random_walk_2d` *vykreslí do grafu náhodnou procházku s* `num_steps` *kroky omezenou ve čtverci rozměru* `2*box_size`$\times$`2*box_size` *a náhodnou procházku vrátí.*
- `random_walk_2d_interactive` *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. Tato funkce implementuje i cyklické okrajové podmínky.*

Vzorový kód je napsán takovým způsobem, aby mohl být přímočaře rozšířen pro vícerozměrnou náhodnou procházku.

Úkol 5.2: *Upravte náhodnou procházku tak, aby počítala s cyklickými okrajovými podmínkami. To znamená, že pokud opustíte oblast čtverce jednou jeho stranou, objevíte se na straně protilehlé (jako kdybyste okraje čtverce zavinuli a slepili). Výpočet ukončete poté, co uděláte n kroků.*

Řešení 5.2: *Cyklické okrajové podmínky jsou naprogramovány ve funkci* `random_walk_2d_interactive` *v souboru* `2d.py`. *Stejně je lze naprogramovat i do funkce* `random_walk_2d`, *výsledný graf však nebude vypadat příliš hezky.*

Úkol 5.3: *Zamyslete se nad tím, jak byste realizovali náhodnou procházku s konstantní délkou kroku v $d > 2$ rozměrech, a zkuste své řešení naprogramovat. Zásadní 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 5.3: *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 nesmírně 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*

   $$V_d^{(krychle)} = 2^d, \tag{35}$$

   objem vepsané hyperkoule o poloměru 1 *je*

   $$V_d^{(koule)} = \frac{\pi^{\frac{d}{2}}}{\Gamma\left(\frac{d}{2}+1\right)}, \tag{36}$$

   kde Γ *je Eulerova gama funkce. Vzájemný poměr*

   $$\eta_d \equiv \frac{V_d^{(krychle)}}{V_d^{(koule)}} = \left(\frac{2}{\sqrt{\pi}}\right)^d \Gamma\left(\frac{d}{2}+1\right) \tag{37}$$

   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 nalézá 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

$$\begin{aligned} \phi &= 2\pi\xi_1, \\ \theta &= \arccos\left(1 - 2\xi_2\right), \end{aligned} \tag{38}$$

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

$$\begin{aligned} \hat{n}_x &= \sin\theta\cos\phi = \sqrt{1-(1-2\xi_2)^2}\cos 2\pi\xi_1, \\ \hat{n}_y &= \sin\theta\sin\phi = \sqrt{1-(1-2\xi_2)^2}\sin 2\pi\xi_1, \\ \hat{n}_z &= \cos\theta = 1 - 2\xi_2. \end{aligned} \tag{39}$$

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 `nd.py`. *Funkce* `random_direction` *generuje směr pomocí 1. metody, funkce* `random_direction_gaussian` *pomocí 2. metody. Spočítejte si náhodnou procházku pro $d = 10$ oběma metodami. Uvidíte, že i pro takto relativně „malou“ dimenzi je rozdíl ve výpočetních časech je dramatický.*

6 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.

Jednoduchá náhodná procházka funguje dobře pro funkce s jedním minimem. V obecném případě ale může mít funkce více lokálních minim a právě uvedený algoritmus skončí náhodně v jednom z nich, ze kterého se již nedokáže dostat ven. Přitom 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*.

6.1 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

$$p = \mathrm{e}^{\frac{f - f_{\text{nová}}}{T}}, \tag{40}$$

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 $p = 1$, 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.

6.2 Minimalizace pomocí knihovny SciPy

Python obsahuje funkci pro hledání minima `minimize` v knihovně **`scipy.optimize`**.

6.3 Domácí úkol na 13.4.2021

Úkol 6.1: *Rozšiřte sv;j program z minulého cvičení pro náhodnou procházku tak, aby hledal minimum funkce dvou proměnných $f(x, y)$. Otestujte svůj program pro kvadratickou funkci*

$$f(x, y) = x^2 + y^2 \tag{41}$$

a pro Rosenbrockovu funkci

$$g(x, y) = (a - x)^2 + b\left(y - x^2\right)^2 \tag{42}$$

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 6.1: *Vzorový kód naleznete v souboru* `minimize.py` *a vychází z náhodné vícerozměrné procházky z modulu* `nd.py` *(z tohoto modulu kód využívá funkci* `random_direction_gaussian`*). K minimalizaci jsou v kódu dvě různé metody:*

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

- `minimize_adaptive`*: Předchozí metoda hledání minima má chybu $\Delta x_i \approx$* `step_size`*. 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. $T \propto 1/$*`step_size`*). Mnohem vhodnější je začít s velkým krokem a krok zmenšovat postupně. Tak postupuje tato funkce: začne s krokem* `initial_step_size` *a pokaždé, když se jí nepodaří* `max_failed_steps`*-krát provést úspěšný krok, zmenší délku kroku na polovinu. Výpočet probíhá do chvíle, dokud je délka kroku větší než* `final_step_size`*. Konečná chyba je tedy $\Delta x_i \approx$* `final_step_size` *a doba výpočtu roste pouze logaritmicky se zmenšováním této chyby, tj. $T \propto 1/\log$* `final_step_size`*.*

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

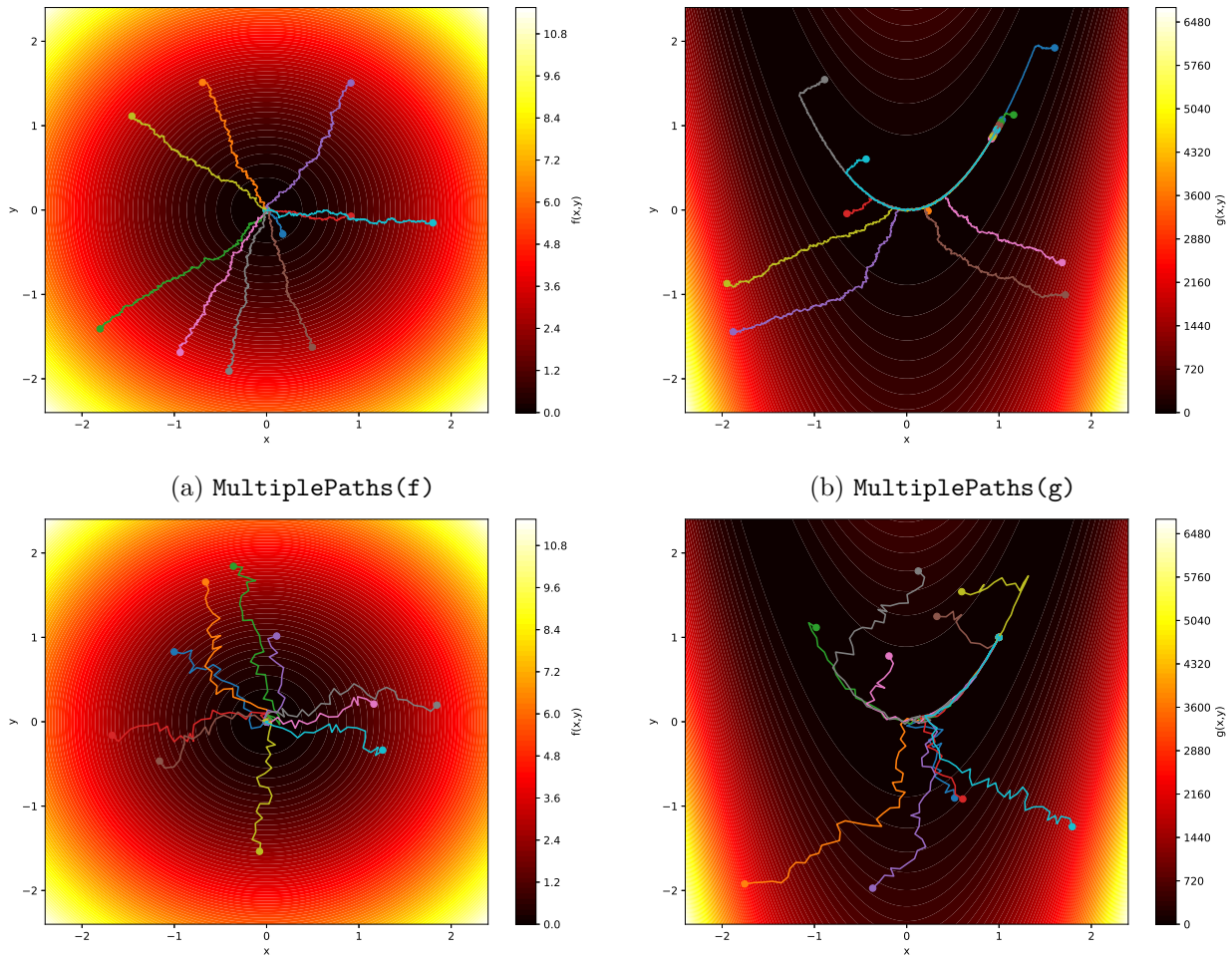
Minimalizace konkrétních funkcí $f(x, y)$ a $g(x, y)$ ze zadání úlohy je provedena v souboru `min_functions.py`*.*

Úkol 6.2: *Náhodnou procházku zakreslete jako čáru do grafu společně s konturovým grafem potenciálu. Návod na nakreslení konturového grafu v Pythonu pomocí funkce* `matplotlib.pyplot.contourf` *naleznete v souboru* `contourf.py`*.*

Řešení 6.2: *Vzorový kód je v souboru* `min_functions.py` *a využívá jako modul* `minimize.py`*. Obsahuje následující funkce:*

- `show_graph`*: 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* `box_size`*.*

- `multiple_paths`*: Vypočítá celkem* `num_paths` *náhodně vybraných jednoduchých minimalizačních procházek a vykreslí je v grafu společně s konturovým grafem minimalizované funkce. Parametrem* `method` *lze určit minimalizaci s pevným krokem nebo s adaptivním krokem.*



(a) `MultiplePaths(f)` (b) `MultiplePaths(g)`

(c) `MultiplePathsAdaptive(f)` (d) `MultiplePathsAdaptive(g)`

Obrázek 8: Minimalizace kvadratické funkce (41) a Rosenbrockovy funkce (42) pomocí náhodné procházky a deseti trajektorií náhodně vybraných ze čtverce `initial_box_size` $= 2$. *1. řádek:* Minimalizace metodou `minimize` s délkou kroku `step_size` $= 0.01$ a kritériem ukončení `max_failed_steps` $= 100$. Chyba určení minima je $\Delta x, y \approx 0.01$. V případě Rosenbrockova minima je chyba velká: Rosenbrockovo „údolí“ je velmi pozvolné podél a strmé napříč, což způsobuje, že trajektorie končí v širokém okolí skutečného minima $(x_{\min}, y_{\min}) = (1, 1)$. Počet kroků, než je výpočet ukončen, je zde ≈ 1000. *2. řádek:* Minimalizace vylepšeným algoritmem s postupně se zmenšující krokem. Na počátku je krok velký, `initial_step_size` $= 0.1$, ale postupně se zmešuje až k `final_step_size` $= 10^{-6}$. To stačí i pro přesné nalezení minima Rosenbrockovy funkce. Počet nutných kroků zde je ≈ 3000.

- `multiple_paths`*:*

Výsledky pro funkce f a g ze zadání jsou zobrazeny na obrázku 8.

Úkol 6.3: *Použijte kód pro vícerozměrnou náhodnou procházku a najděte pomocí něho minimum funkce čtyř proměnných*

$$\begin{aligned} h(s,t,u,v) = {} & \frac{1}{4}\left(s^2+t^2+u^2+v^2\right) \\ & - \frac{1}{2}\left[\left(s^2+t^2\right)\left(2-s^2-t^2-u^2-v^2\right)+(su-tv)^2\right] \\ & + \frac{s}{2}\sqrt{2-s^2-t^2-u^2-v^2}. \end{aligned} \tag{43}$$

Řešení 6.3: *Volání funkce* `minimize.minimize_adaptive(h, dimension=4)` *vypočítá minimum v bodě*

$$\begin{aligned} s_{min} &\approx -0.913 \\ t_{min} = u_{min} = v_{min} &\approx 0.000 \\ h_{min} \equiv h(s_{min}, t_{min}, u_{min}, v_{min}) &\approx -0.771. \end{aligned} \tag{44}$$

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. Důležitá podmínka však je, že počáteční bod náhodné procházky musí ležet v definičním oboru funkce* h*.*

Úkol 6.4: *Naprogramujte Metropolisův algoritmus a odlaďte ho na případu funkce*

$$r(x, y) = x^4 - 2x^2 + x + y^2. \tag{45}$$

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

Řešení 6.4: *Pro ukázku, že standardní minimalizace pomocí náhodné procházky vede náhodně do různých lokálních minim, slouží obrázek 9(a).*

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

- `minimize`*: Minimalizuje funkci* `function` *se zadanou teplotou* `temperature` *a s fixní délkou kroku* `step_size`*. Výpočet je ukončen přesně po* `max_steps` *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 9(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 silně závisí na kombinaci teploty a délky kroku.

- `minimize_adaptive`*: Vylepšení spočívající v postupném zmenšování teploty a zároveň kroku. Výpočet začne s krokem* `initial_step_size` *a teplotou* `initial_temperature` *a po každých* `num_steps_change` *krocích déku kroku i teplotu vydělí dvěma. Výpočet je ukončen po dosažení délky kroku* `final_step_size`*. Příklad realizovaných procházek pro funkci (45) je na obrázku 9(c).*

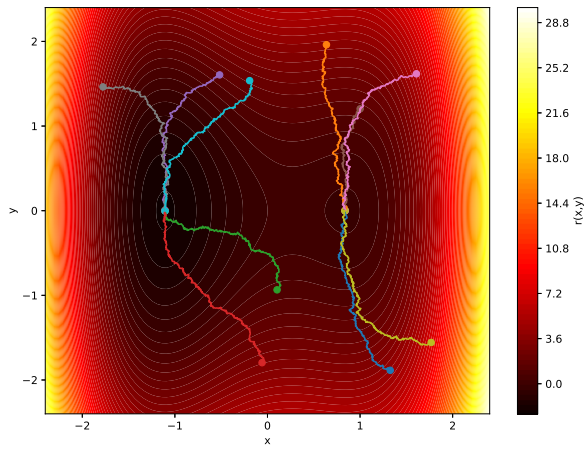
Úkol 6.5: *Prostudujte dokumentaci k funkci* `minimize` *a vytvořte kód, který tuto funkci využije k najití minima všech doposud studovaných funkcí dvou a více proměnných. Pokud programujete v jiném programovacím jazyku, nalezněte odpovídající minimalizační funkci či knihovnu a použijte ji.*

Řešení 6.5: *Knihovní funkci voláme příkazem* `minimize(f, initialCondition)`*, kde parametr* `initialCondition` *je n-tice udávající počáteční bod, ze kterého minimalizační procedura vystartuje.*

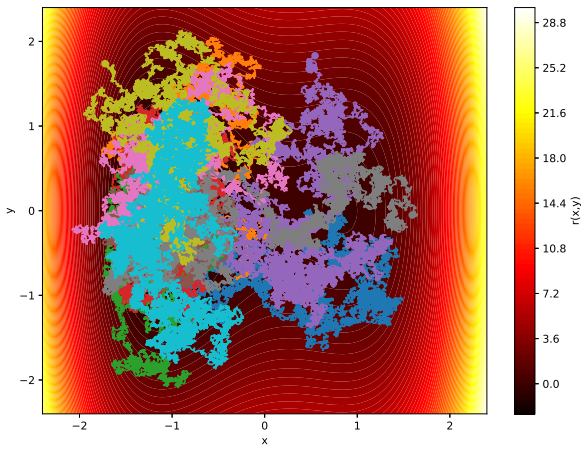
V případě funkce s více lokálními minimy tato knihovní funkce nenajde nutně minimum globální. Lze však pomocí volitelného parametru `method` *vybrat metodu minimalizace, která bude úspěšnější. Pokročilá varianta Metropolisova algoritmu je v knihovně* `scipy.optimize` *naprogramována pod názvem* `basinhopping`*.*

6.4 Shrnutí

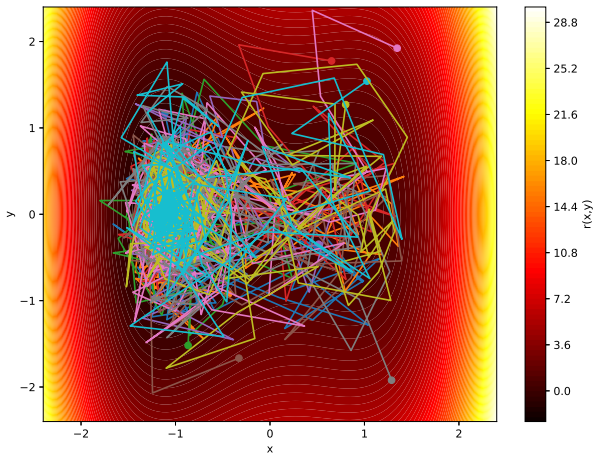
- Jeden z nejjednodušších algoritmů na hledání minima (maxima) funkce je pomocí náhodné procházky. K její implementaci stačí mít generátor náhodných čísel.



(a) `multiple_paths(r, method=minimize.minimize)`



(b) `multiple_paths(r, method=metropolis.minimize)`



(c) `multiple_paths(r, method=metropolis.minimize_adaptive)`

Obrázek 9: Porovnání různých metod minimalizace dvoujámové funkce (45). (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.

- 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.

7 Histogram

V tomto cvičení budeme pokračovat s náhodnými čísly, se kterými již pracovali při programování náhodné procházky v 5. sekci. Zde 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.

7.1 Základní definice a tvrzení z teorie pravděpodobnosti

V následujícím textu budeme značit X spojitou náhodnou veličinu[27] s hodnotami v intervalu $x \in \langle a, b\rangle$[28]. 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

$$\Pr\left[x_1 \leq X \leq x_2\right] = \int_{x_1}^{x_2} \rho(x)\mathrm{d}x. \tag{46}$$

  Hustota pravděpodobnosti je normalizovaná na definičním oboru,

$$\int_a^b \rho(x)\mathrm{d}x = 1 \tag{47}$$

  (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[29]

$$F(x) = \int_a^x \rho(x')\mathrm{d}x'. \tag{49}$$

  Platí tedy díky normalizaci (47)

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

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

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

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

[28] 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.

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

$$\rho(x) = \frac{\mathrm{d}F}{\mathrm{d}x}. \tag{48}$$

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

$$\Pr\left[x_1 \leq X \leq x_2\right] = F(x_2) - F(x_1). \tag{50}$$

- **Střední hodnota:**[30]

$$\mathrm{E}\left[X\right] = \int_{-\infty}^{\infty} x\rho(x)\mathrm{d}x. \tag{51}$$

- **Rozptyl:**

$$\sigma_X^2 = \mathrm{E}\left[X^2\right] - \mathrm{E}\left[X\right]^2 = \int_{-\infty}^{\infty} x^2\rho(x)\mathrm{d}x - \left[\int_{\infty}^{\infty} x\rho(x)\mathrm{d}x\right]^2. \tag{52}$$

- **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,

$$\overline{X} = \frac{1}{n}\sum_{j=1}^{n} x_n. \tag{53}$$

Čím mohutnější máme výběr, tím lépe výběrová střední hodnota aproximuje střední hodnotu,

$$\overline{X} \xrightarrow{n\to\infty} \mathrm{E}\left[X\right]. \tag{54}$$

- **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,

$$\rho_{X,Y}(x, y) = \rho(x)\rho(y). \tag{55}$$

Například věk a výška náhodně vybrané 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

$$Z = X + Y \tag{56}$$

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

$$\rho_Z(z) = \int_{-\infty}^{\infty} \rho_X(\xi)\rho_Y(z - \xi)\mathrm{d}\xi. \tag{57}$$

Střední hodnota a rozptyl náhodné veličny Z jsou dány součtem

$$\begin{aligned} \mathrm{E}\left[Z\right] &= \mathrm{E}\left[X\right] + \mathrm{E}\left[Y\right], \\ \sigma_Z^2 &= \sigma_X^2 + \sigma_Y^2. \end{aligned} \tag{58}$$

[30] **E**xpectation value

- **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

$$Y \sim N(m\mu, m\sigma^2), \tag{59}$$

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

$$U \equiv \frac{Y - m\mu}{\sqrt{m\sigma^2}} \xrightarrow{n\to\infty} N(0, 1). \tag{60}$$

Hustota pravděpodobnosti normálního rozdělení je dána vzorcem (64).

7.2 Příklady náhodných veličin

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

$$\rho_R(x) = \frac{1}{b-a} = \text{konst.} \tag{61}$$

$$\mathrm{E}\left[R\right] = \frac{a+b}{2} \tag{62}$$

$$\sigma^2_R = \frac{(b-a)^2}{12}. \tag{63}$$

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

$$\rho_N(x) = \frac{1}{\sqrt{2\pi\sigma^2}}\, \mathrm{e}^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{64}$$

$$\mathrm{E}\left[N\right] = \mu \tag{65}$$

$$\sigma^2_N = \sigma^2. \tag{66}$$

- **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

$$P_k = \frac{\lambda^k}{k!}\, \mathrm{e}^{-\lambda}, \tag{67}$$

$$\mathrm{E}\left[P\right] = \lambda, \tag{68}$$

$$\sigma^2_P = \lambda, \tag{69}$$

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

Úkol 7.1: *Dokažte vztahy (62)–(63) a (68)–(63).*

Řešení 7.1: *K dokázání (62)–(63) vyjdeme z definičních vztahů pro střední hodnotu a rozptyl (51)–(52):*

$$\mathrm{E}\left[R\right] = \int_a^b \frac{x}{b-a}\mathrm{d}x = \frac{1}{b-a}\left[\frac{x^2}{2}\right]_a^b = \frac{1}{b-a}\frac{b^2-a^2}{2} = \frac{a+b}{2}, \tag{70}$$

$$\begin{aligned}\sigma^2_R &= \int_a^b \frac{x^2}{b-a}\mathrm{d}x - \left(\frac{a+b}{2}\right)^2 = \frac{1}{b-a}\frac{b^3-a^3}{3} - \frac{(a+b)^2}{4} \\ &= \frac{a^2+ab+b^2}{3} - \frac{a^2+2ab+b^2}{4} = \frac{a^2-2ab+b^2}{12} = \frac{(b-a)^2}{12}.\end{aligned} \tag{71}$$

Poissonovo rozdělení je diskrétní, je tudíž nutné použít diskrétní analogii vztahů (51)–(52) (nahrazení integrálů sumami):

$$\mathrm{E}\,[P] = \sum_{k=0}^{\infty} kP_k = \sum_{k=1}^{\infty} \frac{\lambda^k}{(k-1)!}\,\mathrm{e}^{-\lambda} = \left| \begin{array}{l} \textit{substituce} \\ l = k-1 \end{array} \right| = \sum_{l=0}^{\infty} \frac{\lambda^{l+1}}{l!}\,\mathrm{e}^{-\lambda} = \lambda, \tag{72}$$

$$\begin{aligned} \sigma_P^2 &= \sum_{k=0}^{\infty} k^2 P_k - \lambda^2 = \sum_{k=1}^{\infty} \frac{k\lambda^k}{(k-1)!}\,\mathrm{e}^{-\lambda} - \lambda^2 \\ &= \sum_{k=1}^{\infty} \frac{(k-1)\lambda^k}{(k-1)!}\,\mathrm{e}^{-\lambda} + \sum_{k=1}^{\infty} \frac{\lambda^k}{(k-1)!}\,\mathrm{e}^{-\lambda} - \lambda^2 \\ &= \sum_{k=2} \frac{\lambda^k}{(k-2)!}\,\mathrm{e}^{-\lambda} + \lambda - \lambda^2 = \lambda^2 + \lambda - \lambda^2 = \lambda. \end{aligned} \tag{73}$$

7.3 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 $\langle a, b\rangle \times \langle c, d\rangle$[31], 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 ještě potkáme 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

  $$x = F^{-1}(y), \tag{74}$$

  kde F^{-1} je inverzní funkce k F, bude z rozdělení s danou F. Pokud neznáme inverzní funkci, řešíme numericky rovnici

  $$F(x) = y, \tag{75}$$

  která však s ohledem na vlastnosti distribuční funkce popsané v sekci 7.1 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

  $$X = F^{-1}(R) \tag{76}$$

  je náhodná veličina s rozdělením daným distribuční funkcí F.

[31] 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.

7.4 Domácí úkol na 20.4.2021

Úkol 7.2: *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. Nejefektivnějšímu algoritmu stačí jeden průchod vstupním polem (jeden cyklus). Pokuste se na něj přijít.*

Výstupní pole funkce vykreslete jako čárový graf.[32]

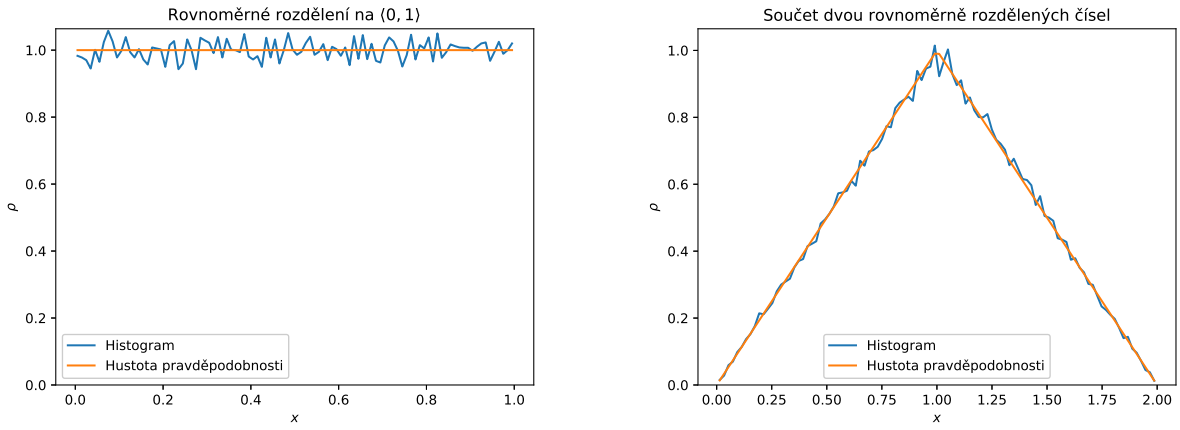
Řešení 7.2: *Vzorový výpočet histogramu je naprogramován v souboru* `histogram.py` *ve funkci* `histogram`. *Klíčové jsou dva řádky:*

```
index = int((d - min_value) / bin_width
histogram[index] += 1
```

Hodnota, kterou chceme zařadit do příslušného okénka histogramu, se nachází v proměnné `d`. *Z ní spočítáme celočíselný index v intervalu mezi* $0 \leq$ `index` $<$ `num_bins` *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*

$$\sum_{i=1}^{\texttt{num\_bins}} h_i w_i = 1, \tag{77}$$

kde h_i *je hodnota histogramu v i-tém okénku a* $w_i \equiv$ `bin_width` *šířka okénka.*



(a) (b)

Obrázek 10: Srovnání histogramu získaného z $n = 10^5$ hodnot vybraných z daného rozdělení a porovnání s teoretickou hustotou pravděpodobnosti (a) rovnoměrného rozdělení (61) ($a = 0, b = 1$) a (b) součtu dvou rovnoměrných rozdělení (79). Počet intervalů histogramu je v obou případech $N = 100$.

Příklady histogramů a jejich vykreslení do grafu jsou v řešení následujícího úkolu a na obrázku 10.

Úkol 7.3: *Otestujte funkci z předchozího úkolu na následujících vstupních datech:*

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 2.2.6).*

[32] Knihovna `matplotlib` obsahuje funkce na přímé vykreslení sloupcového histogramu, například `hist`.

2. *Výběr ze* **součtu dvou** *rovnoměrných rozdělení na intervalu* $\langle 0, 1\rangle$. *Hustota pravděpodobnosti výsledného rozdělení je dána konvolucí* (57). *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* (59) *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 7.3: *Testování funkce* `histogram` *pro různá rozdělení je v souboru* `distributions.py`.

1. *Hustota pravděpodobnosti rovnoměrného rozdělení je zobrazena funkcí* `uniform`. *Výsledný graf je na obrázku 10.*

2. *Hustota pravděpodobnosti součtu dvou rovnoměrných rozdělení je podle* (57)

$$\rho_2(z) = \int_0^1 \rho_1(\xi)\rho_1(z-\xi)\mathrm{d}\xi = \int \rho_1(z-\xi)\mathrm{d}\xi, \tag{78}$$

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:*

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

To lze zapsat zjednodušeně jako

$$\rho_2(z) = \begin{cases} 1 - |1-z| & 0 < z < 2,\\ 0 & \textit{jinak.}\end{cases} \tag{79}$$

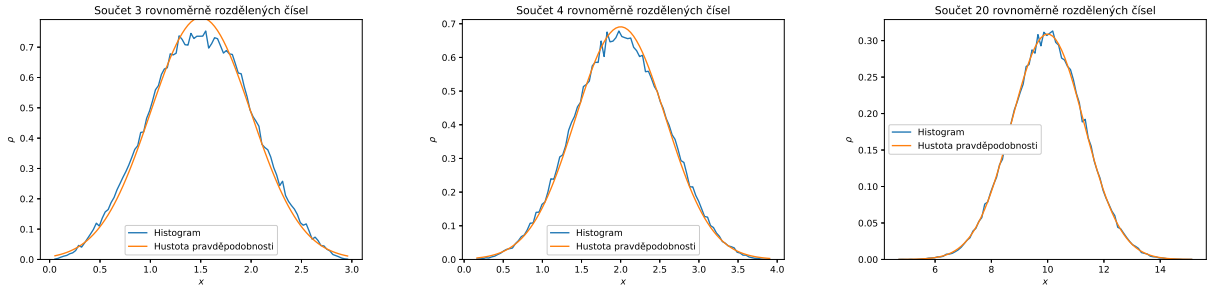
Rozdělení má trojúhelníkový tvar, což odráží skutečnost, že při součtu dvou rovnoměrně rozdělených čísel na intervalu $\langle 0, 1\rangle$ *je mnohem více možností, jak realizovat součet okolo* 1, *než součet na krajích intervalu (stejně jako při hodu dvěma kostkami je největší pravděpodobnost součtu rovna* 7, *jak vědí všichni hráči deskové hry Osadníci z Katanu a je větší než pravděpodobnost součtu* 12, *neboť součet* 7 *můžeme realizovat šesti způsoby z* 36 *celkových možných výsledků hodu dvěma kostkami, zatímco součet* 12 *jen jedním způsobem). Hustota pravděpodobnosti se počítá funkcí* `sum_2_uniform` *a výsledný graf je zobrazen na obrázku 10.*

3. *Výpočet hustoty pravděpodobnosti součtu více rovnoměrných rozdělení je naprogramován ve funkci* `sum_m_uniform` *a porovnání se správně nanormovanou Gaussovkou je na obrázku 11.*

Úkol 7.4: *Na základě centrální limitní věty* (60) *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?*

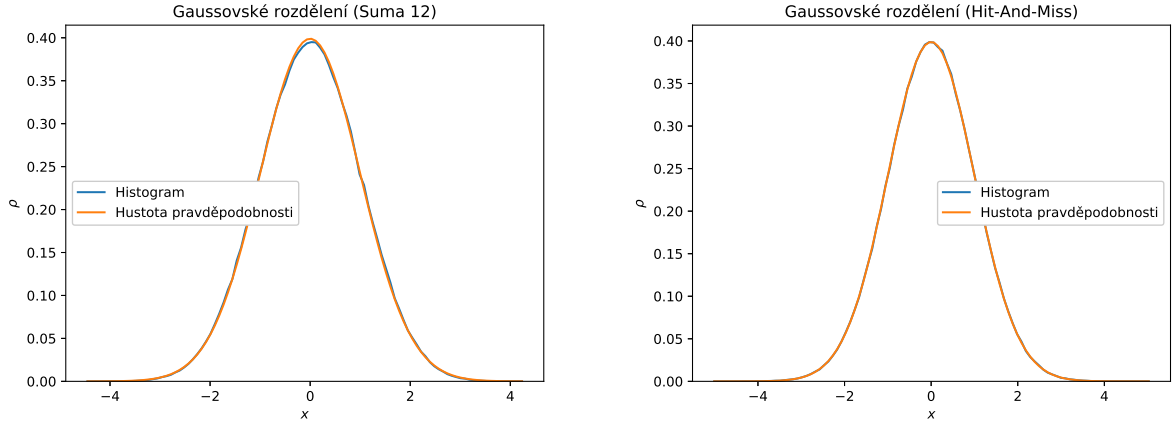
Řešení 7.4: *Ideální počet sečtených čísel z rovnoměrného rozdělení pro získání velmi dobré aproximace čísla z normálního Gaussova rozdělení je* $m = 12$, *neboť pro tuto hodnotu:*

- *Rozptyl výsledného rozdělení je* $\sigma = 1$ *díky vztahům* (60) *a* (63).



(a) (b) (c)

Obrázek 11: Srovnání histogramu získaného z $n = 10^5$ hodnot daných součtem (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 (59) je přibližně splněna. V případě $m = 20$ je již rozdíl od Gaussovského rozdělení prakticky nepozorovatelný. Počet intervalů histogramu je ve všech případech $N = 100$.



(a) (b)

Obrázek 12: Srovnání jednoduchých generátorů Gaussovsky rozdělených náhodných čísel. (a) Generátor založený na použití Centrální limitní věty (60): 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 7.6.

- *Hodnota nagenerovaného čísla je v intervalu $\langle 0, 12\rangle$, se střední hodnotou 6. Pokud tuto střední hodnotu odečteme dle vzorce (60), obdržíme rozdělení se střední hodnotou 0 a zahrnující interval 6σ, který pokrývá 99.9999998% Gaussovského rozdělení.*

Generátor je naprogramován v souboru `gaussian.py`*, funkce* `generator_clt`*. Srovnání generátoru s odpovídající Gaussovkou je na obrázku 12(a).*

Úkol 7.5: *Nakreslete histogram pro rozdělení hodnot v jednotlivých intervalech histogramů z úlohy 7.3. Jaké očekáváte statistické rozdělení v tomto případě?*

Řešení 7.5: *Počet hodnot v jednotlivých intervalech splňuje předpoklady pro Poissonovo rozdělení, očekáváme tedy Poissonovo rozdělení se parametrem*

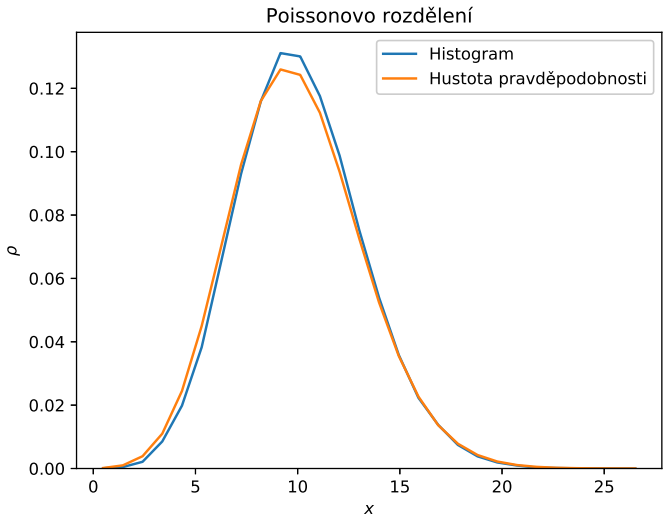
$$\lambda = \frac{\texttt{num\_values}}{\texttt{num\_bins}}, \tag{80}$$

který udává zároveň střední hodnotu (68) a zároveň rozptyl (69). 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).*
- *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 (67) je vypočteno funkcí `poisson` *ze souboru* `distributions.py` *a je vykresleno na obrázku 13.*



Obrázek 13: 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 $\lambda = 10$.

Úkol 7.6: *Na základě známé hustoty pravděpodobnosti (64) vytvořte generátor čísel s Gaussovským normálním rozdělením. Porovnejte jeho rychlost s generátorem založeným na centrální limitní větě, který jste naprogramovali v úloze 7.4 a s generátorem z některé z knihoven.*[33]

Řešení 7.6: *Generátor je naprogramován v souboru* `gaussian.py`*, funkce* `generator_hm`*, a porovnání histogramu takto nagenerovaných náhodných hodnot s teoretickou hustotou pravděpodobnosti je na obrázku 12(b). Nagenerování 10^6 Gaussovsky rozdělených náhodných čísel trvá na mém PC*[34]

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

Úkol 7.7: *Vytvořte generátor čísel z rozdělení daném distribuční funkcí*

$$F(x) = \frac{1}{2}\left(1 + \frac{2}{\pi}\arctan x\right). \tag{81}$$

Jak vypadá analyticky hustota pravděpodobnosti? Nakreslete histogram a porovnejte.

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

[34] 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.

Řešení 7.7: *Hustota pravděpodobnosti je dána derivací distribuční funkce podle vztahu (48), tj.*

$$\rho(x) = \frac{\mathrm{d}F}{\mathrm{d}x} = \frac{1}{\pi}\frac{\mathrm{d}}{\mathrm{d}x}\arctan x = \frac{1}{\pi}\frac{1}{1+x^2}. \tag{82}$$

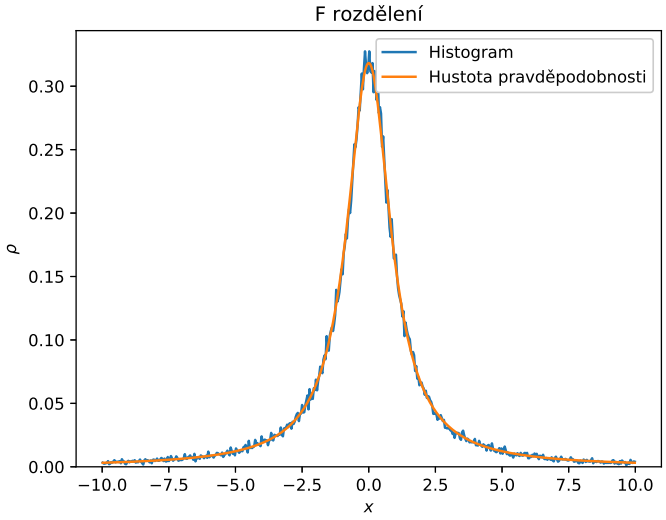
Toto rozdělní se nazývá Cauchyho *či v kvantové fyzice* Breit-Wignerovo. *Popisuje šířku energetických hladin exponenciálně se rozpadajících systémů.*

Ke generování čísel z tohoto rozdělení využijeme vztahu (74). Musíme tedy určit funkci inverzní k distribuční funkci (81), která je

$$F^{-1}(y) = \tan\left[\frac{\pi}{2}(2y-1)\right], \tag{83}$$

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 (82) je počítán funkcí `cauchy` *souboru* `distributions.py` *a je zobrazen na obrázku 14. 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 $x \in \langle -10, 10\rangle$.*



Obrázek 14: Cauchyho (Breit-Wignerovo) rozdělení nagenerované z distribuční funkce (81) a porovnané s teoretickou hustotou pravděpodobnosti (82). Počet hodnot pro histogram je $n = 10^5$, počet intervalů $N = 500$.

8 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.[35] 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: pomocí náhodného házení jehly (či jakékoliv tyčky) na síť rovnoběžných čar nakreslenou na zemi lze určit číslo π,

$$\pi \approx \frac{2l}{h}\frac{N_{\text{zásah}}}{N_{\text{celkem}}}, \tag{84}$$

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 metody Monte-Carlo, konkrétně varianty nazývané hit-and-miss. Té jsme se již dotkli v sekci 7.3 a nyní si rozebereme podrobněji.

Úkol 8.1: *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*

$$p \approx \frac{N_{zásah}}{N_{celkem}} \tag{85}$$

(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%.*

8.1 Hit-And-Miss

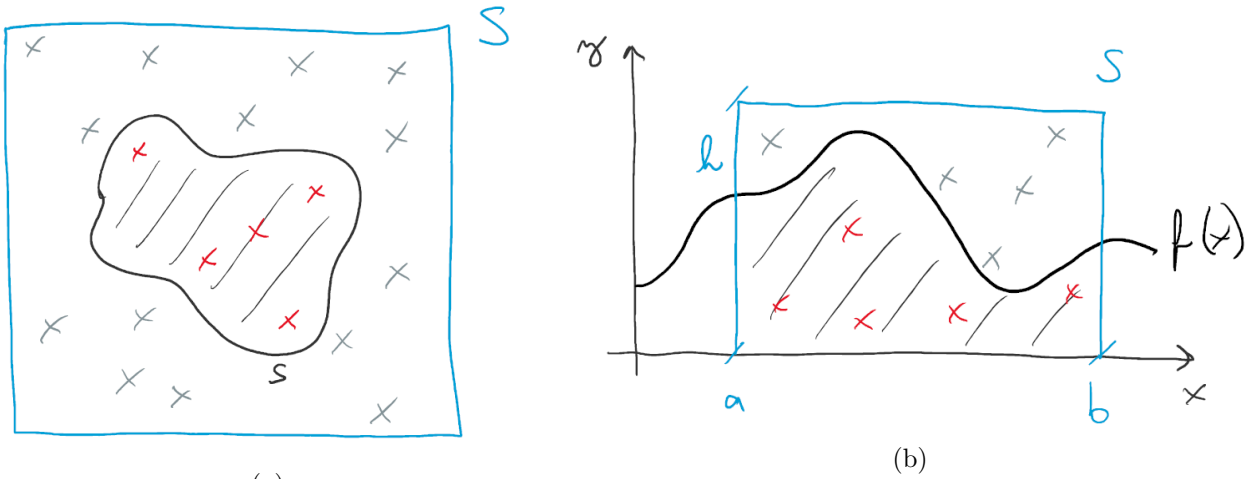
Tato metoda spočívá v jednoduché aplikaci zákona velkých čísel. Její esence je načrtnuta na obrázku 15. Uvažujme nejprve, že chceme změřit plochu s černého obrazce složitého tvaru znázorněného na panelu (a). Obrazec vepíšeme do jiného jednoduchého obrazce, jehož plochu S známe (nejčastěji obdélník, případně 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

$$s \approx S\frac{N_{\text{zásah}}}{N_{\text{celkem}}}. \tag{86}$$

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 (86) vypočítáme hodnotu integrálu:

$$\int_a^b f(x)\mathrm{d}x \approx S\frac{N_{\text{zásah}}}{N_{\text{celkem}}} = (b-a)h\frac{N_{\text{zásah}}}{N_{\text{celkem}}}. \tag{87}$$

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



(a) (b)

Obrázek 15: Metoda hit-and-miss (a) pro výpočet plochy složitého obrazce a (b) pro výpočet integrálu jednorozměrné funkce. Podrobnosti k obrázku jsou uvedeny v textu.

8.1.1 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 (68) a rozptylem (69)

$$\lambda = \mathrm{E}\left[N_{\text{zásah}}\right] = \sigma^2_{N_{\text{zásah}}}. \tag{88}$$

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

$$\Delta N_{\text{zásah}} = \sqrt{\sigma^2_{N_{\text{zásah}}}} \approx \sqrt{N_{\text{zásah}}}. \tag{89}$$

Relativní chyba pak je

$$\delta N_{\text{zásah}} = \frac{\Delta N_{\text{zásah}}}{N_{\text{zásah}}} \approx \frac{1}{\sqrt{N_{\text{zásah}}}}. \tag{90}$$

Tento vzorec platí i pro relativní chybu ve výpočtu plochy (86) či integrálu (87).

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.

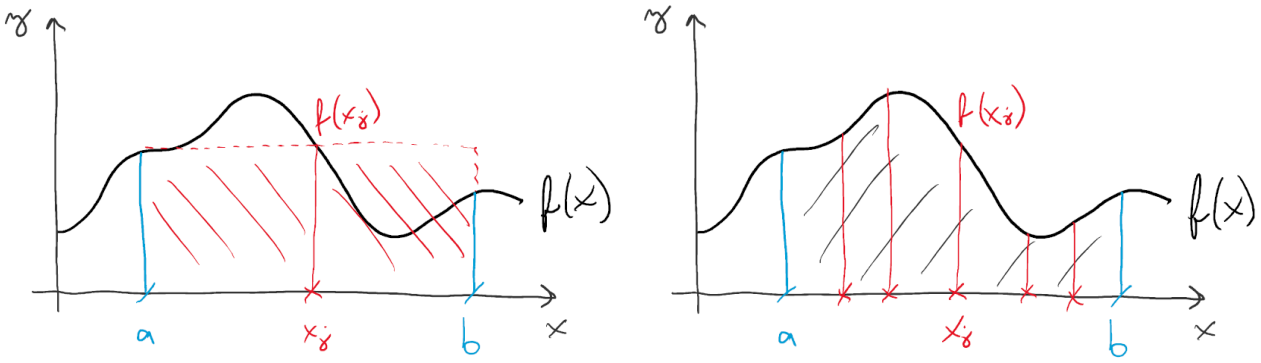
8.1.2 Použití

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 (90).

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

Úkol 8.2: *Vytvořte program na výpočet objemu d-rozměrné jednotkové koule metodou Monte-Carlo.*[36] *Pro jakou dimenzi bude tento objem největší číslo?*

8.2 Monte-Carlo integrace



(a) (b)

Obrázek 16: Monte-Carlo integrace. Vysvětlení je v hlavním textu.

Pro hledání integrálu je mnohem efektivnější metoda, již lze vysvětlit pomocí obrázku 16. 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),

$$\int_a^b f(x)\mathrm{d}x \approx (b-a)f(x_j). \tag{91}$$

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,

$$\int_a^b f(x)\mathrm{d}x \approx \frac{1}{N}\sum_{j=1}^{N}(b-a)f(x_j) = \frac{b-a}{N}\sum_{j=1}^{N} f(x_j). \tag{92}$$

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

$$\int_a^b f(x)\mathrm{d}x \approx \frac{1}{N}\sum_{j=1}^{N}\frac{f(x_j)}{\rho(x_j)}, \tag{93}$$

přičemž nejvhodnější je volit takové pravděpodobnostní rozdělení, jehož hustota pravděpodobnosti co nejlépe kopíruje integrovanou funkci.[37] 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 (61), která po dosazení dá předchozí vzorec (92).

[36] 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.

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

Chybu metody integrace lze odhadnout pomocí směrodatné odchylky

$$\Delta \equiv \sigma = \sqrt{\overline{f^2} - \overline{f}^2}, \tag{94}$$
$$\overline{f^2} = \frac{1}{N}\sum_{j=1}^{N} f^2(x_j),$$
$$\overline{f}^2 = \left[\frac{1}{N}\sum_{j=1}^{N} f(x_j)\right]^2.$$

Úkol 8.3: *Metodou Monte-Carlo spočítejte integrály*

$$I_1 \equiv \int_0^{2\pi} \mathrm{e}^{-x} \sin x \,\mathrm{d}x, \tag{95}$$
$$I_2 \equiv \int_0^{\sqrt{10\pi}} \frac{\sin x^2}{\sqrt{1+x^4}} \mathrm{d}x. \tag{96}$$

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.

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 8.4: *Spočítejte čtyřrozměrný integrál*

$$I_3 \equiv \int_\Omega \sin\sqrt{\ln\left(x+y+z+w+2\right)}\,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z\,\mathrm{d}w, \tag{97}$$

kde integrační oblast je hyperkoule

$$\Omega : \left(x-\frac{1}{2}\right)^2 + \left(y-\frac{1}{2}\right)^2 + \left(z-\frac{1}{2}\right)^2 + \left(w-\frac{1}{2}\right)^2 \leq \frac{1}{4}. \tag{98}$$

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 (92) 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 (86).

9 Paralelizace

Rychlost výpočtu lze v zvyšovat dvěma základními způsoby:

1. Zvyšováním rychlosti procesoru.
2. Zvětšováním počtu procesorů.

Zatímco první způsob již narazil na fyzikální limity a kupředu postupuje jen zvolna,[38] 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*[39]). 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*.[40] V moderních operačních systémech je počet současně běžících procesů neomezený. Pokud je procesů více, než dostupných procesorových jader, tak operační systém velmi rychle přepíná mezi prováděním procesu, čímž 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 17.

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).[41]

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 ní zapisovalo). To by totiž vedlo k nejednoznačnému výsledku, protože nelze a priori

[38] 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 je navíc zpomalováno přístupem programu do operační paměti, proto dnešní procesory zvládnou řádově nanejvýš stovky milionů instrukcí za vteřinu. Jeden jednoduchý příkaz v Pythonu může vyžadovat stovky až tisíce procesorových instrukcí, takže Python vykoná řádově maximálně stovky tísíc příkazů za vteřinu. Ke zrychlení procesoru se využívají nejrůznější sofistikované metody. Procesor například odhaduje, kam se program vydá, a instrukce se snaží předpočítávat 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).

[39] 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.

[40] 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.

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

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 má pro sebe nějakou proměnná zamčenou.

Jiný 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.[42]

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`**.

- `integrate_1D_Pool`: Nejjednodušší způsob paralelizace je pomocí objektu `Pool`. Při vytváření instance tohoto objektu mu předáme parametr `processes` udávající maximální počet procesů, které bude objekt obsluhovat.[43] 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`.[44] 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 `integrate_1D` musíme předat argumenty čtyři, proto volíme složitější `starmap`.

  Při vytvoření instance objektu `Pool` jsme použili konstrukci s klíčovým slovem `with`.

- `integrate_1D_process`: Složitější, avšak univerzálnější je použití objektů `Process`.[45] Ty vyžadují vykonání tři kroků:

  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.

[42] 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.

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

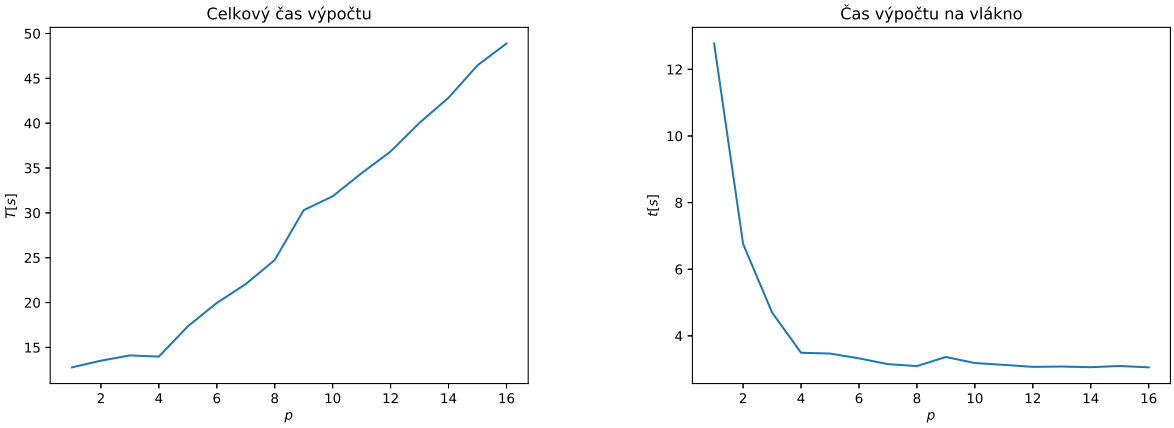
[44] Pokud je n-tic s argumenty víc než počet dostupných procesů, bude objekt `Pool` spouštět volání postupně.

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

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` pro všechny.

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 `integrate_1D_parallel` a která nám zavolá naši funkci `integrate_1D` a její návratovou hodnotu uloží do sdílené proměnné. Naší funkci `integrate_1D` chceme předat beze změny všechny parametry, které pomocná funkce `integrate_1D_parallel` 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.



(a) (b)

Obrázek 17: Výpočetní čas integrace metodou Monte-Carlo při použití různého počtu paralelních procesů p, přičemž v každém vlákně je spuštěn výpočet integrálu I_1 (95) 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 $4 \leq p \leq 8$ začíná narůstat 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.

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 vidět pomocí funkce `plot_integrate_1D_duration`. Tato funkce spouští postupně paralelní výpočet pro různé počty navzájem běžících paralelních vláken, a výsledky zobrazí do grafů, které jsou vykresleny v obrázku 17. Pozorujeme, že nejlepší přesnosti (nejvyššího poštu Monte-Carlo bodů pN) za jednotku času dosáhneme, pokud vytížíme všechna dostupná jádra.

9.1 Domácí úkol na 4.5.2021

Úkol 9.1: *Prostudujte si vzorový příklad v souboru* `multiprocessing.py` *a na jeho základě vytvořte kód, který bude počítat hodnotu integrálu* I_3 (97) *ve více vláknech. Zjistěte si, kolik výpočetních jader má procesor na vašem počítači, a spusťe výpočet tak, aby zaměstnal všechna jádra.*

Úkol 9.2: *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).*

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).

[3] B. P. Abbott *et al.* (LIGO Scientific Collaboration and Virgo Collaboration), *Observation of Graviational Waves from a Binary Black Hole Merger*, Physical Review Letters **116**, 061102 (2016).