

Zápisky k předmětu Využití počítačů ve fyzice

Pavel Stránský

28. března 2021

Obsah

1	Instalace používaných nástrojů	3
1.1	Instalace Pythonu	3
1.1.1	Instalace doplňujících knihoven	3
1.2	Instalace Visual Studio Code	4
1.2.1	Instalace doplňku pro Python	5
1.3	Instalace Git	5
2	Úvod do používaných nástrojů	7
2.1	Verzovací systém Git	7
2.1.1	Prvotní nastavení	8
2.1.2	Stav souborů v repozitáři	8
2.1.3	Životní cyklus souborů v repozitáři	9
2.1.3.1	Vytvoření nového repozitáře	9
2.1.3.2	Stav repozitáře	9
2.1.3.3	Vytvoření a první zapsání nového souboru	10
2.1.3.4	Změny v souboru	11
2.1.3.5	Přidání dalšího souboru do projektu	11
2.1.3.6	Zobrazení historie revizí	12
2.1.4	.gitignore	13
2.1.5	Větve	14
2.1.5.1	Vytvoření nové větve	14
2.1.5.2	Výpis větví	15
2.1.5.3	Změny a sloučení větví	15
2.1.5.4	Odstranění větve	15
2.1.6	Návrat k dřívějším verzím	15
2.1.7	Cheat Sheet	16
2.2	Programovací jazyk Python	16
2.2.1	Vytvoření a spuštění kódu	17
2.2.2	Základy syntaxe Pythonu	17
2.2.3	Pojmenování proměnných a formátování	18
2.2.4	Řady (knihovna numpy)	19
2.2.5	Grafy (knihovna matplotlib)	19
2.2.6	Pseudonáhodná čísla	20
3	Obyčejné diferenciální rovnice 1. řádu	21
3.1	Pár důležitých pojmů	21
3.2	Eulerova metoda 1. řádu	22
3.3	Eulerova metoda 2. řádu	22
3.4	Runge-Kuttova metoda 4. řádu	22

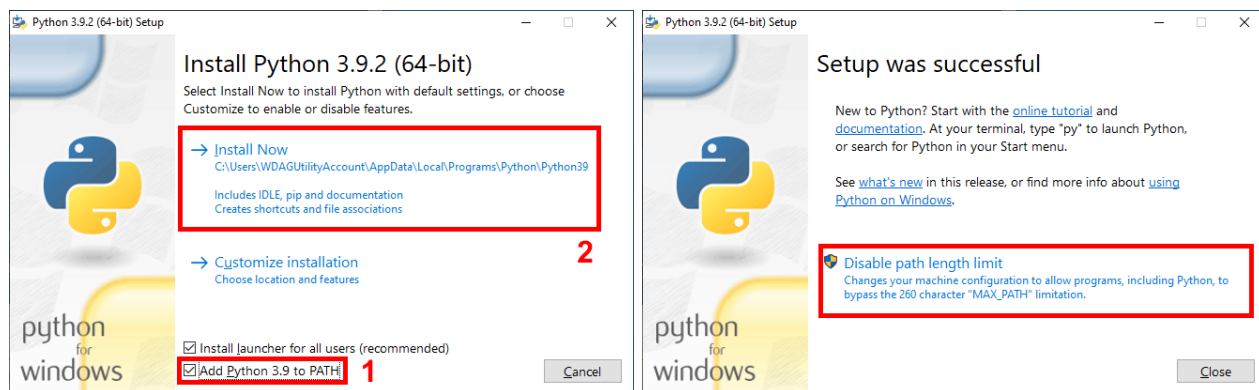
3.5	Odvození metod	23
4	Soustavy diferenciálních rovnic 1. řádu	27
4.1	Symplektické algoritmy	27
5	Náhodná procházka	30

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 neuznávané operační systémy (Linux, Windows, macOS). Má nepřehledné možnosti při editaci zdrojových souborů, překladu a ladění snad ve všech známých programovacích jazycích. Bohaté možnosti nastavení umožňují 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 REPL² Pythonu, ve které můžeme již psát všechny příkazy programovacího jazyka, které se po zadání ihned provedou a vypíší výsledek.

1.1.1 Instalace doplňujících knihoven

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

- [NumPy](#) (**N**umerical **P**ython: numerická matematika, řady a vícedimenzionální datové typy),
- [SciPy](#) (**S**cientific **P**ython: 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.

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

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

³Pokud 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 instalaci pomocí Snap Store příkazem v terminálu

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

nebo použít tento [návod](#).

⁴Integrated Development and Learning Environment.

1.2.1 Instalace doplňku pro Python

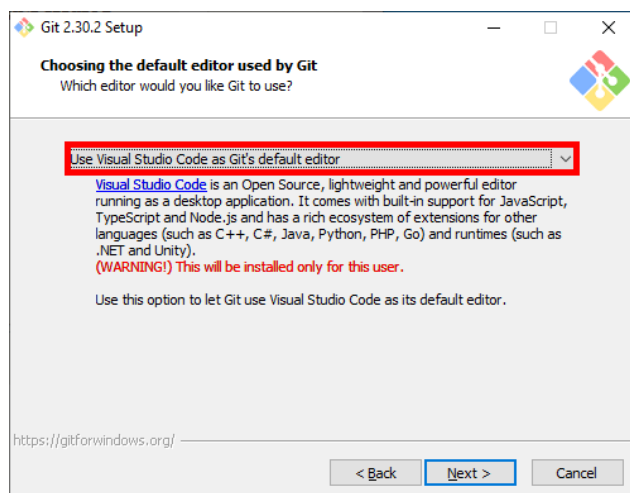


Ke správě doplňků (extensions) se dostanete kliknutím na ikonku 1 nebo stisknutím **Ctrl+Shift+X**. Vyhledáte doplněk **Python** 2 od Microsoftu, vyberete ho 3 a nainstalujete 4.

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 idiomatické 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, 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 centrálně řízeným verzovacím systémům 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 nejpobulárnějších 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 kdokoli, tedy i vy nebo já, má přístup k celému repozitáři, tedy ke všem 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 proto 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, nabízí se spousta nástrojů a doplňků pro různé programy, které práci s repozitáři usnadní a zrychlí. Obsluha Git repozitářů je ostatně 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 "..."
```

příčemž za ... doplníte své jméno (přezdívkou) 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 Stav souborů v repozitáři

- *Nesledovaný (untracked)*: Systém Git historii tohoto souboru neuchovává. Každý nově vytvořený soubor je v tomto stavu.
- *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
```

- *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 dá příkaz

```
git diff
```


- *Zapsaný (committed)*: Soubor je systémem Git sledován(verzován) a je zapsaný 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

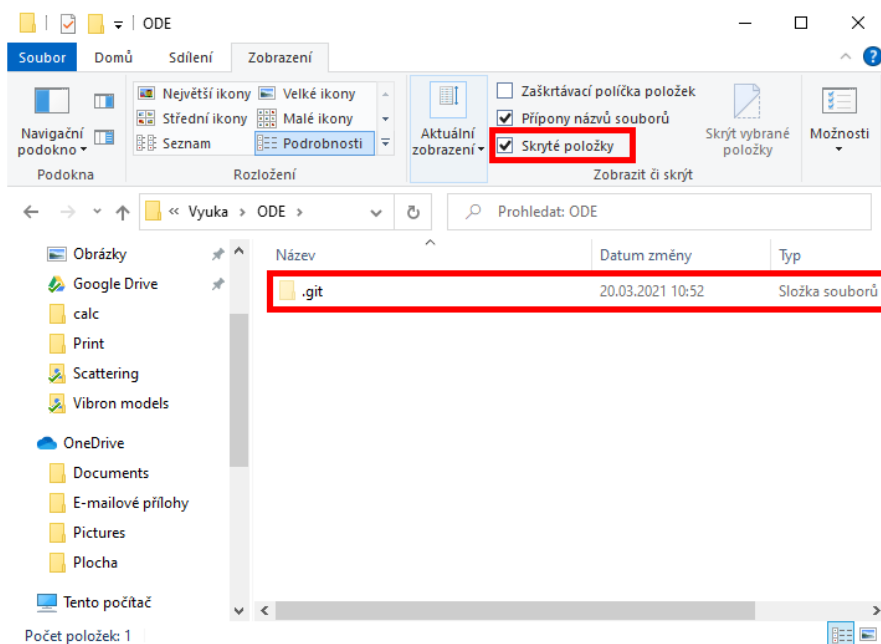
Práci s repozitářem budu demonstrovat na příkladu projektu pro integraci diferenciálních rovnic ze sekce 3.

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 přepokopírujete složku s projektem včetně podsložky `.git` kamkoliv jinam, klidně na jiný počítač, kopírujete zároveň celou historii projektu.

2.1.3.2 Stav repozitáře

Aktuální stav repozitáře kdykoliv zjistíte pomocí příkazu

```
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 napovídat, které další akce lze v další fázi použít a jak.

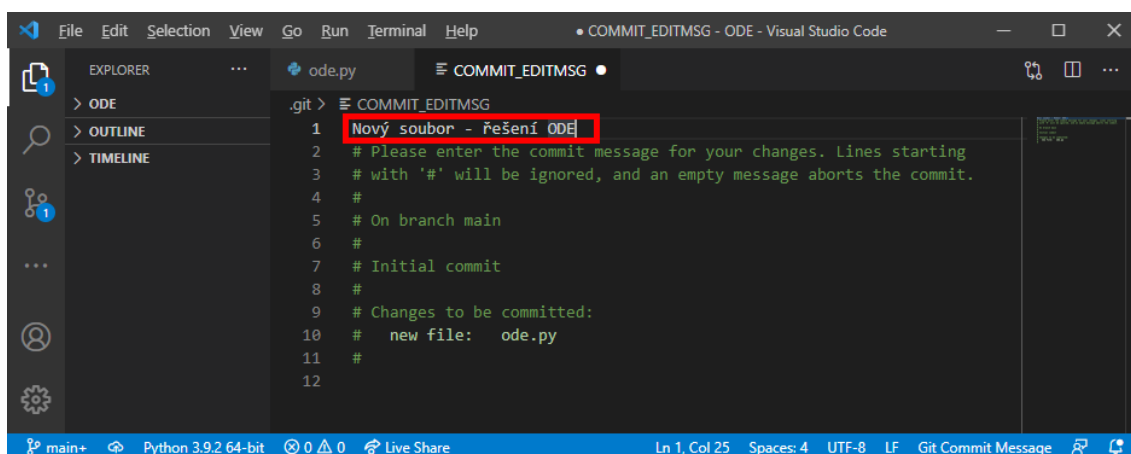
Příkazem **git add *** se nový soubor připraví k zapsání. Stav repozitáře se změní takto:

```
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) 1ef9756] 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

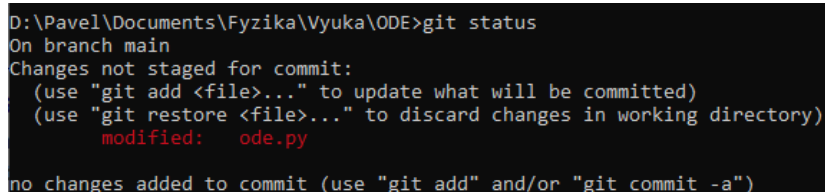
    return ys, ts

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



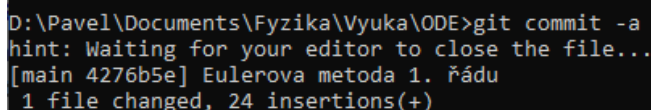
```
D:\Pavel\Documents\Fyzika\Vyuka\ODE>git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   ode.py

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 označím popisem „Eulerova metoda 1. řádu“.



```
D:\Pavel\Documents\Fyzika\Vyuka\ODE>git commit -a
hint: Waiting for your editor to close the file...
[main 4276b5e] Eulerova metoda 1. řádu
1 file changed, 24 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 přeneseme kód

```
import ode

def relaxation(y, t):
    return -y

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

Nesmíme zapomenout náš „modul“ nainportovat 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__`.⁵ 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

```
__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
```

⁵Pro urychlení provádění celého programu ukládá do tohoto adresáře interpret Pythonu speciálně předpřipravené používané moduly.

```

D:\Pavel\Documents\Fyzika\Vyuka\ODE>git log
commit 946cb25a6e75a08aa68e8b50bd4ad0956fc1a908 (HEAD -> main)
Author: Pavel Stránský <microsoft@pavelstransky.cz>
Date: Sat Mar 20 15:02:54 2021 +0100

    Oddělení řešitele ODE a modelu

commit 4276b5e9248801f197b9c85ff80d14798bc1aedc
Author: Pavel Stránský <microsoft@pavelstransky.cz>
Date: Sat Mar 20 13:12:51 2021 +0100

    Eulerova metoda 1. řádu

commit 1ef9756ba45d85b17f3ddcadb74f2fffd2218de0a
Author: Pavel Stránský <microsoft@pavelstransky.cz>
Date: Sat Mar 20 12:28:20 2021 +0100

    Nový soubor - řešení ODE

```

Vypsání seznamu 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 (větvě 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, $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ 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 větvi⁶ 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 testům, 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. Předpokládáme, 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 příkazem⁷

⁶Hlavní větev se donedávna pojmenovávala `master`, dnes se spíše prosazuje pojmenování `main`.

⁷Vytvoření nové větve a přepnutí se do ní lze jedním příkazem `git switch -c odesystem`.

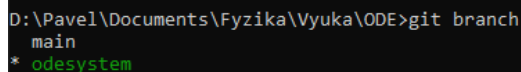
```
git switch odesystem
```

Zpět na hlavní větev se lze vrátit příkazem `git switch main`.⁸

2.1.5.2 Výpis větví

Seznam všech větví se vypíše příkazem

```
git branch
```



```
D:\Pavel\Documents\Fyzika\Vyuka\ODE>git branch
main
* odesystem
```

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 větví `odesystem`, můžeme tyto změny začlenit do hlavní větve `main`. K tomu slouží příkaz

```
git merge odesystem
```

spuštěný v hlavní větví. 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 odrazil obě konfliktní změny, a zapsat příkazem `git commit -a`. Tím je slučování dokončeno.

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 poslední zapsanou verzi (lze použít i hromadné `git restore *`⁹).
- `git reset HEAD~1` zruší poslední¹⁰ zapsání, avšak soubory ve složce projektu ponechá tak, jak jsou.¹¹

⁸Git umožňuje přepínání mezi větvemi také pomocí příkazu `git checkout`.

⁹`git restore *` se chová podobně jako `git reset --hard`.

¹⁰Číslo udává, o kolik revizí se vracíme zpět.

¹¹HEAD vlastně značí ukazatel na poslední zapsání.

- `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.¹² K návratu zpět stačí použít příkaz pro přepnutí do hlavní větve `git switch main`.

2.1.7 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 odladované). 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.¹³

Python lze charakterizovat jako jazyk:

- *Interpretovaný*, což znamená, že provádění programů probíhá přímo ze zdrojového kódu.¹⁴ Ke spuštění kódu 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 interpret 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 interpretovaných jazyků jsou jazyky kompilované, přičemž kompilace se provádí buď přímo do strojového kódu daného procesoru¹⁵ nebo do mezikódu. Ten ke spuštění vyžaduje dodatečnou kompilaci, která však může být vysoce optimalizovaná přímo pro danou hardwarovou konfiguraci použitého počítače.¹⁶

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

¹²Tzv. Detached HEAD state.

¹³Jako příklad poslouží nejnovější verze programu [Mathematica](#), [Origin](#) či [SAS](#).

¹⁴Mezi další známé interpretované jazyky patří například JavaScript nebo PHP.

¹⁵Například C/C++, Pascal nebo Fortran.

¹⁶Zde se jedná například o jazyky Java, C# a celý .NET framework nebo Julia.

- *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.¹⁷
- V Pythonu lze pythonovským způsobem používat tři základní programovací paradigmat: *procedurální, objektové a funkcionální*.

Python klade důraz na jednoduchost, stručnost a čitelnost kódu. Filosofie programovacího jazyka je shrnuta v [Zenu Pythonu](#),¹⁸ 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.
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 popsanych metod spuštění kódu.

2.2.2 Základy syntaxe Pythonu

Soubor se vzorovými příklady najdete v repozitáři ke cvičení: [introduction.py](#).¹⁹ 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*: Zaměřte se na možnosti formátování řetězců. Python obsahuje jen dva základní číselné typy: `int` a `float`. Zbytek je podobný jako v jiných programovacích jazycích.
2. *Proměnné a kolekce*: Důležité standardní kolekce jsou *seznam* (list) `[...]` a *slovník* (dictionary) `{...}`. 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.

¹⁷ Algoritmus se nazývá *Garbage collection*.

¹⁸ Zen se také vypíše, pokud do svého kódu zadáte `import this`.

¹⁹ Vzorový soubor je inspirován příkladem [Nauč se Python v Y minutách](#).

3. *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.

4. *Funkce*: Python umožňuje 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 jakémukoli 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`.

5. *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 seznámte 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`.

6. *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 nejnужnější 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](#).²⁰ Python obsahuje dokonce knihovnu `autopep8`, která váš zdrojový soubor podle těchto pravidel naformátuje. Automatické formátování podle PEP 8 lze nastavit i ve [Visual Studio Code](#).

²⁰PEP = Python Enhancement Proposal

Pro pojmenování proměnných existuje jediné závazné pravidlo, které zní, že identifikátor se nesmí shodovat s žádným z [35 klíčových slov](#) jazyka. Kromě toho jsou ve výše uvedeném dokumentu další nezávazná pravidla k pojmenování proměnných:²¹

- *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 `0`, 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.

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 druhou 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.²² 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 vícerozměrné 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.

²¹Čitelnosti různých stylů pojmenování proměnných se věnují i seriózní vědecké články, například <http://www.cs.kent.edu/~jmaletic/papers/ICPC2010-CamelCaseUnderScoreClouds.pdf>.

²²Stejně jako v mluvených jazycích se zlepšujete čtením literatury, v programovacích jazycích pomáhá čtení cizích kódů. Pro příklad použití komentářů se koukněte třeba na zdrojový soubor funkce [optimize](#) z knihovny [scipy](#).

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 $x \in \langle 0; 1 \rangle$.
 - `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 μ a směrodatnou odchylkou σ .
 - `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 ná sada 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.
 - `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 $\langle 0; 1 \rangle$.
 - `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,

$$\frac{dy}{dt} = f(y, t) \quad (1)$$

s počáteční podmínkou

$$y(t_0) = y_0. \quad (2)$$

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

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

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

kde ϕ je funkce, která udává směr, podél kterého se při numerickém řešení vydáme. Volba 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

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

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

- **Lokální diskretizační chyba:**

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

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

- **Akumulovaná diskretizační chyba:**

$$\epsilon_i = y_i - y(t_i) \quad (6)$$

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

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

- **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

$$\phi_i = f(y_i, t_i), \quad (8)$$

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

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

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:

$$\begin{aligned} k'_1 &= f(y_i, t_i) \\ k'_2 &= f\left(y_i + k'_1 \frac{\Delta t}{2}, t + \frac{\Delta t}{2}\right) \\ \phi_i &= k'_2 \end{aligned} \quad (10)$$

3.4 Runge-Kuttova metoda 4. řádu

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

- 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 $p = 8, 9$ vyžaduje $p + 2$ 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:

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

takže

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

- *Eulerova metoda 2. řádu.*

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

a odtud

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

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:

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

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.

Domácí úkol na 23.3.2021

Úkol 3.2: Naprogramujte Eulerovu metodu 1. a 2. řádu a Runge-Kuttovu metodu. Vyřešte numericky diferenciální relaxační rovnici²³

$$\frac{dy}{dt} = -y \quad (17)$$

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 $t \in \langle 0; 10 \rangle$.

Úkol 3.3: Rozšířte kód tak, aby počítal průměrnou kumulovanou chybu

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

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.

Úkol 3.4: Pomocí naprogramovaných metod vyřešte nelineární diferenciální rovnici

$$\frac{dy}{dt} = \sin(ty) \quad (19)$$

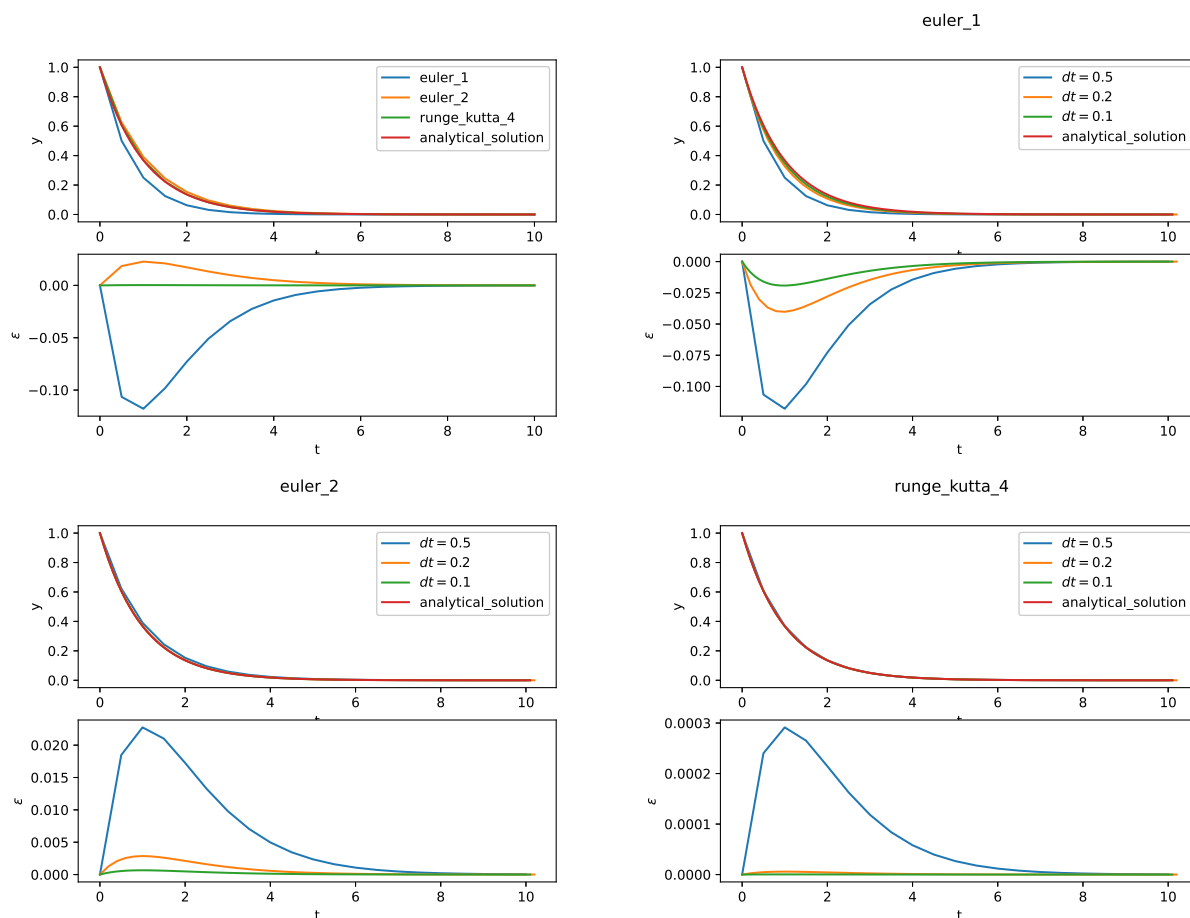
s počáteční podmínkou $y_0 = 1$, $t_0 = 0$ a vykreslete graf jejího řešení.

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.

- `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.
 - `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.

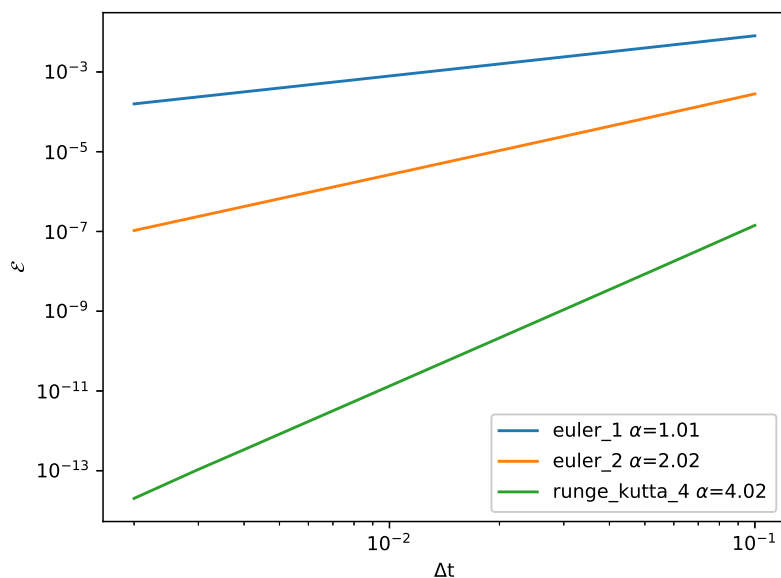
²³Rovnice popisuje například Newtonův zákon chlazení tělesa, aktuální množství prvku při radioaktivním rozpadu nebo různé populační modely.

- `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 krokem, řešení nakreslí do jednoho grafu a porovná s teoretickou funkcí.
 - Nakreslí graf k úloze 3.3.



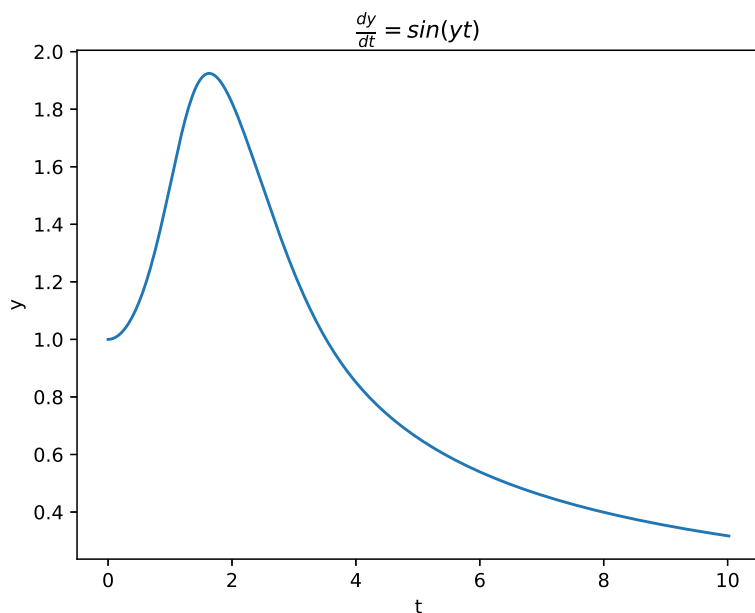
Obrázek 1: Integrace relaxační diferenciální rovnice (17) různými metodami a s různými časovými kroky.

Průslušné 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.



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

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



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

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(\mathbf{y}, t), \quad (20)$$

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

Příklad 4.1: Pohybovou rovnici

$$Ma = F(x), \quad (21)$$

kde M je hmotnost pohybujícího se tělesa, $x = x(t)$ jeho poloha a $a = a(t) = d^2x/dt^2$ zrychlení převedeme na dvě diferenciální rovnice prvního řádu zavedením rychlosti $v = v(t) = dx/dt$:

$$\frac{d}{dt} \begin{pmatrix} x \\ v \end{pmatrix} = \begin{pmatrix} v \\ \frac{1}{M}F(x) \end{pmatrix}, \quad (22)$$

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

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

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

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

$$a = \frac{d^2x}{dt^2} = -x \quad \Longleftrightarrow \quad \frac{d}{dt} \begin{pmatrix} x \\ v \end{pmatrix} = \begin{pmatrix} v \\ -x \end{pmatrix} \quad (24)$$

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

$$x''' + xx'' - x'^2 + 1 = 0. \quad (25)$$

Řešení 4.1: Hledaná soustava diferenciálních rovnic je

$$\frac{d}{dt} \begin{pmatrix} x \\ v \\ a \end{pmatrix} = \begin{pmatrix} v \\ a \\ -xa + v^2 - 1 \end{pmatrix}. \quad (26)$$

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 diferenciálních rovnic. Od běžných algoritmů je odlišuje to, že zachovávají objem fázového prostoru, a tedy i energii (zatímco u obecných algoritmů se energie s integračním časem mění a většinou roste).

V praxi se ze symplektických algoritmů používá nejčastěji *Verletův algoritmus*. Pro diferenciální rovnici 2. řádu ve tvaru

$$M \frac{d^2x}{dt^2} = F(x), \quad (27)$$

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

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

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.

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

$$\frac{d^2x}{dt^2} = -x \quad (29)$$

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 $t \in \langle 0; 30 \rangle$.

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

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

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

$$\begin{array}{l} x_{i+1} = x_i + v_i \Delta t \\ v_{i+1} = v_i - x_i \Delta t \end{array} \quad \longrightarrow \quad \begin{array}{l} x_{i+1} = x_i + v_i \Delta t \\ v_{i+1} = v_i - x_{i+1} \Delta t \end{array} \quad (31)$$

(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.

Úkol 4.5: Pohrajte si s řešením rovnice pro klesající exponenciálu

$$\frac{d^2x}{dt^2} = x \quad (32)$$

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.

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

$$\begin{aligned} \frac{dx}{dt} &= \sigma(y - x), \\ \frac{dy}{dt} &= x(\rho - z) - y, \\ \frac{dz}{dt} &= xy - \beta z \end{aligned} \quad (33)$$

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.

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 $\mathbf{y}_i \rightarrow \mathbf{y}_{i+1}$ takovým způsobem, aby pravděpodobnost pohybu do všech směrů byla stejná. Délka kroku d 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.

Úkol 5.1: *Naprogramujte náhodnou procházku ve 2D rovině. Délku kroku volte konstantní, například $l = 1$, směr volte náhodně. Začněte z bodu $[0; 0]$ 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.*

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.*

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 jakéhokoli směru nastával se stejnou pravděpodobností (esence úlohy tedy spočívá v generování náhodného směru v d -rozměrném prostoru).*