NOFY080 poznámky k přednášce

Emil Varga, Marek Talíř

13. listopadu 2025

Obsah

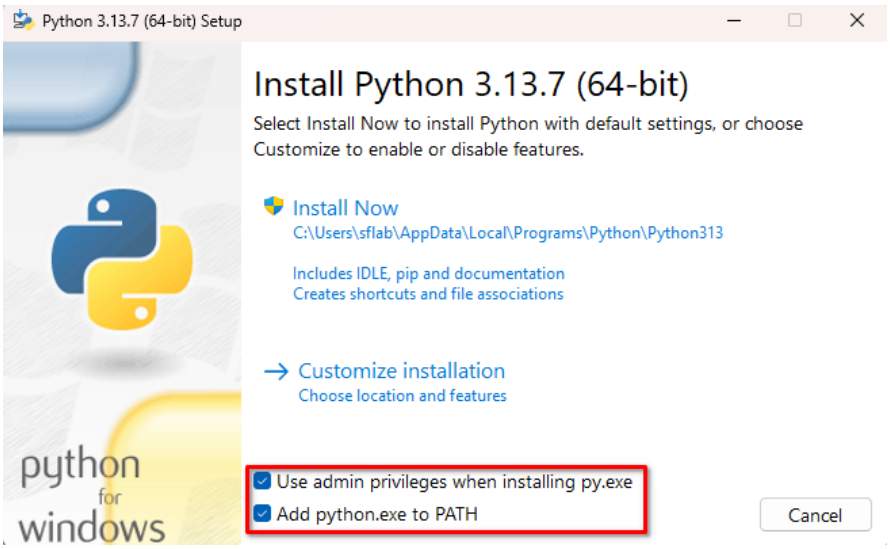
Python Jazykové Intermezza

Code Snippets

1 Nastavení prostředí

Pokud nemáte nainstalovaný Python, můžete si jej stáhnout z `https://www.python.org/downloads/` (konkrétně nejnovější verzi pro Windows zde).

Po spuštění instalátoru povolte volbu "Add python.exe to PATH"(a také doporučuji povolit administrátorská oprávnění), viz screenshot



Instalaci proklikejte, otevřete Windows PowerShell a ověřte funkčnost následujícími dvěma příkazy.

```
python --version
```

a

```
pip --version
```

Oba příkazy by měly vypsat verzi programu a žádnou chybovou hlášku. Pokud používáte Linux, Python již pravděpodobně máte nainstalovaný. Pokud ne, použijte balíčkovací systém vaší distribuce, např. na Fedoře:

```
sudo dnf install python3
```

Běžné programování v Pythonu silně spoléhá na externí balíčky, které mohou mít různé verze a závislosti. Aby se předešlo konfliktům mezi různými projekty, doporučuje se pro každý projekt používat virtuální prostředí (*virtual environment*, alebo ve zkratce venv). V tomto kurzu budeme takové virtuální prostředí používat, abychom zajistili konzistentní vývojové prostředí pro všechny. Virtuální prostředí jsou doporučeným způsobem správy závislostí v projektech v Pythonu.

Pro správu těchto virtuálních prostředí budeme používat open-source nástroj `uv`, dostupný na `https://github.com/astral-sh/uv`. Pokud již máte nainstalovanou jakoukoli verzi Pythonu, můžete `uv` nainstalovat jednoduchým spuštěním

```
pip install uv
```

Doporučuji postup pomocí `pip`, můžete ale také použít samostatné instalátory na `https://github.com/astral-sh/uv`.

Pro otestování instalace spusťte v příkazovém řádku

```
uv --version
```

a mělo by se objevit něco podobného jako `uv 0.5.30`.

Nyní vytvoříme projekt s názvem `NOFY080_2025`, který bude obsahovat všechny soubory související s tímto kurzem, spuštěním

```
uv init NOFY080_2025
```

který vytvoří nový adresář se stejným názvem, s jednoduchým souborem "Hello World"v Pythonu, několika dalšími soubory, které `uv` používá ke sledování závislostí vašeho projektu, a nastaví v něm git repozitář (v tomto kurzu nemusíte používat git, ale zájemci se mohou naučit základy v dodatku **??**).

Nyní se přesuňte do adresáře projektu

```
cd NOFY080_2025
```

a přidejte balíčky, které budeme v průběhu tohoto kurzu potřebovat

```
uv add numpy scipy matplotlib
```

což vytvoří virtuální prostředí a nainstaluje zadané balíčky (v adresáři projektu by se měl objevit adresář `.venv`).

Pokud byste chtěli vytvořit virtuální prostředí bez instalace jakýchkoli balíčků, dá se použít příkaz

```
uv venv <name of the virtual environment>
```

Pro aktivaci virtuálního prostředí spusťte následující příkaz v Linuxu nebo Mac OS

```
source .venv\bin\activate
```

a ve Windows

```
.venv\Scripts\activate
```

Když nyní spustíte `python`, spustí se verze z virtuálního prostředí, nikoli vaše systémová verze. Pro deaktivaci virtuálního prostředí jednoduše spusťte `deactivate`.

1.1 Spouštění kódu v Pythonu

Python je interpretovaný jazyk, který není třeba před spuštěním kompilovat. Kód v Pythonu ukládáme do textových souborů s příponou `.py` nebo do takzvaných Jupyter notebooků (přípona souboru `.ipynb`).

Pro otestování naší instalace vytvořte v našem projektovém adresáři nový soubor Pythonu (textový soubor s příponou `.py`) `test.py` s následujícím obsahem

```
import matplotlib.pyplot as plt

plt.plot([1, 2, 3], [4, 5, 6], '-o')
plt.show()
```

Pro jeho spuštění můžete buď manuálně aktivovat virtuální prostředí a spustit jej pomocí `python test.py` nebo použít `uv run test.py`. Měl by se objevit jednoduchý graf. Ve zbytku těchto skript myslíme příkazem `python` ten, který spouštíme z virtuálního prostředí .

Pro spuštění kódu v Pythonu přímo, bez uložení do souboru, otevřete vhodný terminál a zadejte `python` - objeví se výzva (*prompt*)

```
    >>>
```

Toto je Read-Evaluate-Print-Loop (REPL); jakýkoli zadaný kód bude spuštěn a výsledek vytištěn na obrazovku. Pro ukončení zadejte `quit()`. Pro spuštění souboru `myfile.py` se nejprve přesuňte do jeho adresáře a poté jej jednoduše spusťte pomocí `python myfile.py` (všimněte si, že pokud máte aktivované virtuální prostředí, skutečný skript Pythonu se nemusí nacházet ve vašem projektovém adresáři).

Cvičení 1.1. Vytvořte textový soubor `hello.py`, který obsahuje jeden řádek

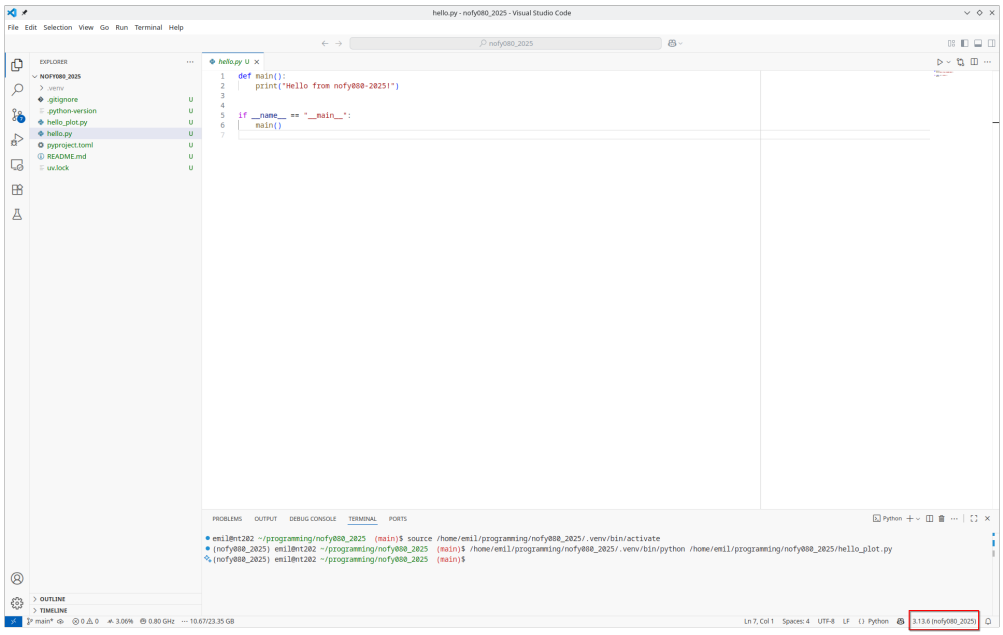
```
1         print("Hello world!")
2
```

a poté tento soubor spusťte.

Někdy nechceme, aby se interpreter Pythonu ukončil ihned po dokončení běhu našeho programu (např. chceme zkontrolovat proměnné vytvořené během běhu programu); to lze provést pomocí `python -i file.py`, kde `-i` znamená *interaktivní*. Alternativně lze uživatelsky přívětivější (barevně odlišená syntaxe, automatické doplňování atd.) verzi interaktivního Python REPL vyvolat pomocí `ipython` (nebo `ipython3`, v závislosti na vaší instalaci), který lze také použít k interaktivnímu spouštění souborů pomocí `ipython3 -i myfile.py`.

1.2 Krátká poznámka k textovým editorům

K psaní kódu v Pythonu lze použít jakýkoli textový editor, včetně výchozího Poznámkového bloku ve Windows. Pro vaše duševní zdraví je však přínosné používat alespoň něco se zvýrazňováním syntaxe, jako je Notepad++, nebo editor s více funkcemi, jako je VS Code nebo Spyder, kde můžete soubor spustit bez přepínání do terminálu. V tomto kurzu doporučuji používat VS Code s nainstalovaným rozšířením pro Python, protože usnadňuje práci s virtuálními prostředími. Při otevření adresáře VS Code automaticky detekuje virtuální prostředí a použije ho při spuštění souboru. Název aktuálně používaného virtuálního prostředí je uveden v pravém dolním rohu (zvýrazněno červeně):



Práce s prostředími ve Spyderu je poněkud složitější; viz průvodce zde.

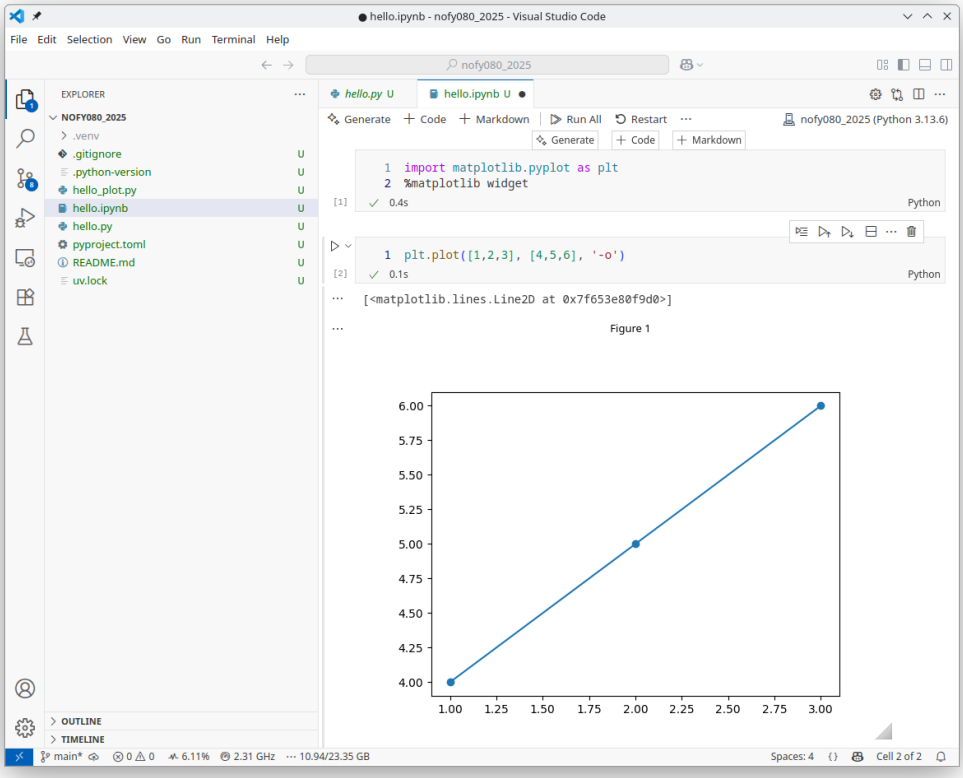
1.2.1 Jupyter Notebooky

Jupyter notebooky (spouštěné pomocí `jupyter-notebook` v příkazovém řádku) poskytují snadno použitelné interaktivní prostředí s rozhraním běžícím ve webovém prohlížeči, a jsou dobré pro jednoduché

vyzkoušení kódu[1]. Jupyter notebooky také dobře fungují s virtuálními prostředími a VS Code. Přejděte do adresáře svého projektu a spusťte

```
uv pip install jupyter ipympl
```

Balíček `ipympl` umožňuje použití interaktivních grafů Matplotlib v Jupyter noteboocích. Dále si také nainstalujte rozšíření `jupyter` ve VS Code. Nyní jednoduše vytvořte soubor s příponou `.ipynb` a můžete jej používat ve VS Code stejně jako v prohlížeči. Při prvním spuštění budete požádáni o výběr virtuálního prostředí, ve kterém by měl notebook běžet; to, které jsme vytvořili, by mělo být jednou z možností, pokud je notebook uložen v projektovém adresáři (stejný adresář jako adresář `.venv`).



[1]V tomto kurzu budeme (většinou) používat skripty kvůli snazšímu rozdělení do modulů a menšímu počtu problémů s paralelním programováním, se kterými se setkáme později.

2 Základy Pythonu

2.1 Proměnné a typy

Proměnné se vytvářejí přiřazením hodnoty k dosud nepoužitému jménu proměnné. Deklarace, jako v C/C++ nebo Pascalu, nejsou nutné.

```
my_integer = 5
my_float = 3.14
my_string = "double quote string"
my_string2 = 'single quote string'
my_string3 = "you can use 'single' quote in double quote string"
my_bool = False
```

Listing 1: Definování proměnných

Všimněte si, že proměnné obecně nemají pevný typ (tj. celé číslo nebo řetězec) a provedení něčeho jako `my_integer = "a string"` nezpůsobí typovou chybu. Samotný Python se nestará o to, jakého typu proměnná je, pokud jsou operace, které s ní provádíme, podporovány. Pokud však potřebujeme znát typ proměnné `v`, můžeme jej zjistit pomocí `type(v)`.

Základní aritmetické operace `+,-,*,/` fungují na číslech podle očekávání. Umocňování je `**`. Všimněte si však, že `/` automaticky *povýší (promotion)* celá čísla na desetinná (float). Pro celočíselné dělení můžeme použít `//` a pro zbytek po dělení `%`. Aritmetické operace jsou také definovány pro některé nečíselné typy, kde "dávají smysl", tj. řetězce lze spojovat pomocí `+`. Porovnávací operátory jsou `==` pro rovnost (**nezaměňovat s =, což je přiřazení hodnoty proměnné**), `!=` pro nerovnost a `<,<=,>,>=` pro uspořádání. Booleovské operátory jsou `and, or, not`.

Existují také **bitové** operátory `&,|,~,^` (bitové AND, OR, NOT a XOR). Tyto operují na jednotlivých bitech čísla. Dejte si pozor na záměnu mezi umocňováním `**` a bitovým exkluzivním *or* (XOR) `^`.

Komplexní čísla jsou v Pythonu podporována. Syntaxe pro zadání komplexního čísla je např. `3 + 5j`, což se překládá do matematického zápisu $3 + 5i$. Konkrétně imaginární jednotka je `1j`. Všimněte si, že písmeno `j` následuje bezprostředně za číslem.

Cvičení 2.1. Vyzkoušejte si základní aritmetiku v REPL. Zkuste vynásobit řetězec celým číslem. Co se stane, když se ho pokusíte vydělit?

Shrnutí základních vestavěných operátorů

`+,-,*,/, **`	základní aritmetika, umocňování
`and, or, not`	booleovské operace
`<, <=, ==, >, >=`	porovnávání
`&,|, ~, ^`	bitové operace, bitové AND, OR, NOT a XOR (exkluzivní nebo)

2.2 Uživatelský vstup

Většinou budeme psát neinteraktivní skripty, nicméně pro získání základního textového vstupu od uživatele můžeme použít funkci `input()`, **která vždy vrací řetězec**. Musíme jej sami převést na číslo (pomocí `int()` nebo `float()`), pokud očekáváme, že uživatel zadá číslo, např.:

```
age = int(input("How old are you? "))
print("You will be 100 in", 100-age, "years.")
```

2.3 Výpis a řetězce

Již jsme se setkali s funkcí `print()`, která na standardní výstup vypíše textovou reprezentaci objektů, které jí předáme. Přijímá jeden nebo více argumentů.

Pro převod jakékoli hodnoty na řetězec můžeme použít funkci `str()`. Speciální znaky, jako je nový řádek nebo tabulátor, je třeba *escapovat*, např. `\n` a `\t`. Zpětná lomítka je také třeba escapovat `\\`. Abychom zabránili Pythonu v interpretaci zpětných lomítek jako speciálních znaků (užitečné např. pro zadávání cest k souborům ve Windows), můžeme použít tzv. raw řetězce, např.:

```
print("s\tri\ng") #\t and \n are interpreted as tab and new line
print(r"s\tri\ng") # raw string, note the r immediately before "
```

Pro vytváření složitějších výpisů našich proměnných můžeme použít *formátovací* řetězce (nebo f-stringy) tak, že proměnnou, kterou chceme vložit do řetězce, uzavřeme do `{}`, např.:

```
vari = 5
varf = 3.14
fstr = f"An interger value {vari:05d}, a float value {varf:010.6f}, exponential
form {varf:g}"
```

Nahrazení textu hodnotou proměnné (tzv. interpolace) se specifikuje jako
`{výraz:specifikátor_formátu}`,
přičemž specifikátory formátu jsou volitelné. Zde specifikátor formátu `05d` znamená celé číslo s celkovou šířkou alespoň 5 znaků a `010.6f` znamená číslo s plovoucí desetinnou čárkou se 6 číslicemi za desetinnou čárkou a celkovou šířkou alespoň 10 znaků.

Pro úplnou specifikaci možných formátovacích specifikátorů viz dokumentaci.

2.4 Funkce

Funkce se definují pomocí klíčového slova `def`. Funkce mohou přijímat argumenty a vracet hodnotu.

```
def my_function(my_string):
    print("Someone wants to know how long \"" + my_string + "\" is!")
    return len(my_string) #returns the length of a my_string
```

Všimněte si odsazení, kterým se v Pythonu identifikují bloky kódu, tj. neexistuje zde `begin,end` jako v Pascalu nebo `{}` jako v C/C++.

Více argumentů se odděluje čárkou a argumenty mohou mít výchozí hodnotu, např. argument c bude 1, pokud není uvedeno jinak.

```
def calculate(a, b, c=1):
    """
    A function that calculates c*(a + b)

    Parameters:
    -----------
    a, b, c: numbers
        arguments of the calculation

    Returns:
    --------
    result: number
        The result of calculating c*(a + b)
    """
    return c*(a + b)

>>> calculate(2, 3)
5
>>> calculate(2, 3, 4)
20
```

Listing 2: Definice funkce

Zde je dlouhý řetězec ohraničený trojitými uvozovkami `"""` takzvaný dokumentační řetězec funkce (nebo zkráceně **docstring**). Trojité uvozovky lze použít kdekoli a znamenají pouze to, že řetězec se rozkládá na více řádků. K docstringu lze poté přistupovat pomocí funkce `help()` nebo pomocí textových

editorů. Formátování použité ve výše uvedeném příkladu je běžně srozumitelné pro IDE, např. Spyder jej zobrazí pěkně naformátovaný v panelu Nápověda (standardně v pravém horním rohu okna).

Argumenty s výchozími hodnotami musí následovat za povinnými argumenty. S více volitelnými argumenty můžeme funkci volat pomocí *pojmenovaných argumentů* (keyword arguments):

```
    def calculate2(a, b, c=1, d=1):
        return c*(a + b)**d

    >>> calculate(2, 3, d=2) # necha vychozi hodnotu c, ale nastavi d
    25
```

Listing 3: Volitelné a pojmenované argumenty.

Funkce lze předávat v proměnných a argumentech, konkrétně funkce mohou přijímat jiné funkce jako argumenty, např.:

```
    def add1(x):
        return x + 1

    def change_number(x, func):
        return func(x)

    >>> change_number(1, add1)
    2
```

Listing 4: Funkce vyššího řádu.

Krátké, jednoduché funkce (které se vejdou na jeden řádek) lze definovat přímo jako takzvané anonymní neboli *lambda* funkce, např.

```
    >>> change_number(1, lambda x: 2 + x)
    3
```

Listing 5: Lambda funkce.

Lambda funkce jsou obzvláště užitečné pro částečné vyplnění seznamů argumentů existujících funkcí, např.

```
    >>> change_number(1, lambda x: calculate2(x, 4, c=4, d=0.5))
    8.94427190999916
```

2.5 Složené typy

2.5.1 Seznami a n-tice (Lists and tuples)

Seznam (**list**) je kolekce jakýchkoli pythonovských objektů definovaná pomocí hranatých závorek `[]`,

```
    my_list = [1, 2.17, "a string", ['another', 'list']]
```

K hodnotám v seznamu lze přistupovat pomocí indexování seznamu **začínajícího od 0**, např. `my_list[1] == 2.17`. Záporné indexy se počítají od konce, tj. `my_list[-1]` je poslední prvek, `my_list[-2]` předposlední atd.

Seznamy jsou příkladem *objektu*, objekty mají přidružené funkce zvané metody, které se volají pomocí tečkové notace. Užitečné metody pro seznamy jsou:

- `append` and `pop`

```
>>> my_list = [1, 2.0, 'three']
>>> my_list.append('one more')
>>> my_list.pop() # odstrani posledni prvek a vrati ho
"one more"
```

- `my_list.sort()`, seřadí seznam na místě, pokud lze všechny prvky seznamu porovnat; pro vytvoření nového seznamu s seřazenými prvky můžeme použít `sorted(my_list)`

- `my_list.index('a')` najde index prvního výskytu 'a' nebo vrátí chybu

Kromě jednoduchého indexování lze k podmnožinám seznamů přistupovat pomocí *řezů* (slicing), např.:

```
l = [1,2,3,4,5,6,7,8,9]
l[2:5] # [3,4,5]
l[-5::2] # [5,7,9]
```

Obecně je syntaxe `my_list[start_idx:stop_idx:step]`, `start_idx` je včetně, `stop_idx` je vyjma. Všechny tři jsou volitelné, výchozí hodnota start je 0, výchozí hodnota stop je -1 a výchozí hodnota step je 1.

N-tice (**Tuples**) se v mnoha ohledech chovají podobně jako seznamy, s výjimkou, že jsou **neměnné** (immutable), tj. samotnou n-tici ani hodnoty v ní nelze měnit. Indexování a řezy však fungují stejně. N-tice se definují pomocí kulatých závorek `()`. Pokud n-tice obsahuje pouze jednu hodnotu, je třeba přidat čárku, aby se odlišila od výrazu v závorkách, tj. `single_tuple = (1,)`.

Běžným použitím n-tic je vracení více hodnot z funkce, tj.:

```
def add_and_subtract(x, y):
    return x+y, x-y # Vsimnete si, ze zavorky () nejsou potreba v nekterych pripadech

# tu rozbalime vracenou dvojici do dvou promennych
a, s = add_and_subtract(3, 2) # a==5, s==1
```

Všimněte si, že jsme neuložili návratovou hodnotu jako n-tici, ale okamžitě jsme ji rozdělili do dvou samostatných proměnných. Tomu se říká **rozbalování** (unpacking) a lze jej použít pro jakýkoli složený typ.

2.5.2 Slovníky (Dictionaries)

Slovníky (Dictionaries) jsou kolekce přiřazení klíče k hodnotě libovolného typu, často jsou klíče řetězce, např.:

```
my_dict = {
    'key1': value1,
    'key2': value2,
}
```

which can then be accessed as `my_dict['key2']` etc.

2.6 Kontrola toku programu

2.6.1 if-elif-else

Příkaz `if` přijímá booleovský výraz a spustí kód ve svém bloku, pokud se podmínka vyhodnotí jako `True`.

```
if condition:
    do_if_true()
elif condition2: # nepovinne
    do_if_2true() # spusti se jenom kdyz condition je False a condition2 je True
else: # nepovinne
    do_if_false() # spusti se jenom kdyz oboje condition and condition2 jsou False
```

2.6.2 match

Obecnější přiřazení hodnoty k akci je k dispozici pomocí `match`[2].

[2]Dostupné od Pythonu 3.10

```
    command = 'yell'
    match command:
        case 'talk':
            print("hello")
        case 'yell':
            print("HELLO!!")
        case _: # vychozi chovini
            print("neznamy prikaz")
```

`match` může být obzvláště silný v kombinaci s rozbalováním (unpacking), kde se `match` snaží co nejlépe najít vzoru svého argumentu. To vám umožňuje například zpracovávat příkazy a jejich argumenty, např.:

```
def process_command(command):
    match command:
        case ('speak', value):
            print(f"Hello {value}")
        case 'yell':
            print('HELLO!!')
        case ('yell', value):
            print(f"HELLO!! {value}")
        case (command, value):
            print(f"I'm supposed to \"{command}\" with \"{value}\")
```

což se chová jako

```
    >>> process_command(('speak', 1))
    Hello 1
    >>> process_command('yell')
    HELLO!!
    >>> process_command(('yell', 2))
    HELLO!! 2
    >>> process_command(('shout', 20))
    I'm supposed to \"shout\" with \"20\"
```

Všimněte si však, že pokud argument pro `match` neodpovídá žádnému poskytnutému vzoru, nic se nestane, tj. chová se podobně jako `if` s chybějící klauzulí `else`:

```
    >>> process_command('shout')
    # zadny vystup
```

Cvičení 2.2. Napište funkci, která na vstupu přijme koeficienty a, b, c kvadratické rovnice $ax^2 + bx + c = 0$ a vypíše řesení x_1, x_2. Pokud reálné řešení neexistuje, funkce do příkazové řádky vypíše chybu.

2.7 Smyčky (Loops)

Pro cyklení s daným počtem iterací můžeme použít cykly `for` s funkcí `range()`. Můžeme také cyklit přes cokoli iterovatelného, např. seznam nebo slovník.

```
    for k in range(10):
        print(k)

    for value in my_list:
        print(value)
```

Obecný tvar `range()` je `range(start, stop, step)`, start je včetně, stop je vyjma a krok je standardně 1. `range()` nevrací seznam, ale něco, co se nazývá *iterátor*. Můžeme jej převést na seznam pomocí `list(range(start, stop, step))`.

Jestliže potřebujeme iterovat přes nějaké hodnoty a zároveň pracovat s indexem můžeme použít `enumerate`.

```
    for index, value in enumerate(my_list):
        print(f"my_list[{index}] = {value}")
```

Pro cyklení, dokud je podmínka pravdivá, můžeme použít cyklus `while`,

```
    while condition:
        do_stuff()
```

Běh cyklů lze upravit pomocí `continue`, které přeskočí zbytek kódu v iteraci a přejde na další, nebo `break`, které cyklus okamžitě ukončí. Běžné použití je s nekonečnými cykly, např.:

```
    while True: # bude bezet do nekonecna
        do_stuff()
        if should_we_stop: # pokud to neukoncime
            break
```

Cykly se často používají k vytváření seznamů, což lze zjednodušit pomocí **list comprehension** (generování seznamu), jako je

```
    [expression(k) for k in iterable if condition(k)] # if podminka je nepovinna
```

For example,

```
    [k**2 for k in range(5)] # seznam druhych mocnin pro k < 5
    [k**2 for k in range(5) if k % 2 == 0] # seznam druhych mocnin pro sude k
```

Listing 6: List comprehension.

Cvičení 2.3. Napište program, který se zeptá uživatele na jméno a odpoví `"Hello <jméno>!"`.

1. Změňte program tak, aby pokud je jméno uživatele "Andrej", pozdrav se změnil na "Ciao Andrej!".
2. Oddělte logiku vytváření pozdravu do samostatné funkce.

Cvičení 2.4. Napište program, který přijme libovolný počet jmen z příkazového řádku a na konci je vypíše v abecedním pořadí.

1. Uživatel předem zadá počet jmen, která chce zadat. *Nápověda*: `range()` a cyklus `for`
2. Program přestane žádat o nová jména, jakmile uživatel zadá prázdný řetězec. *Nápověda:* `break`
3. Pokud uživatel zadal jméno "Emil", bude ve výsledném výpisu přeskočeno. *Nápověda:* `continue`.

2.8 Používaní externích modulů

Python je známý tím, že má velké množství dostupných knihoven pro téměř cokoli pod sluncem.[3] Abychom je mohli použít, musíme nejprve importovat *modul*, který potřebujeme, např. pro použití funkcí, které pracují s časem, můžeme udělat

```
    import time as t #importuj modul time a dej mu skraceny nazev t
    print(t.time()) #nazev modulu pripajime z leva ke jmenu funkce
```

Pokud chceme z daného modulu použít jen malý počet funkcí, můžeme je importovat konkrétně a nemusíme pro jejich volání používat název modulu, např.

[3] A brzy se naučíme, jak si vytvořit vlastní.

```
    from module import func1, func2
    func1()
    func2()
```

Cvičení 2.5. Napište program pro výpočet Fibonacciho čísla F_n pomocí cyklů i rekurze. Změřte čas potřebný k výpočtu prvních 20 F_n oběma metodami pomocí `time.time()`. *Nápověda:* $F_k = F_{k-1} + F_{k-2},\ F_1 = F_2 = 1$

Cvičení 2.6. Soubor exercises/e2.6.py obsahuje data o částicích v elektrickém poli $E = 1$ kV/m, které je orientováno podél osy z.

Data mají formu seznamu slovníků s následující strukturou:

```
[
    {'charge': -1,
    'id': 1,
    'mass': 0.4113513116324291,
    'position': [-1.5188809980735316e-06,
                 6.639994580333568e-06,
                 8.468849433621426e-06],
    'velocity': [-0.016976782951560687,
                 -0.43614541046795063,
                 0.24858497024060777]},
    {...}
]
```

Pozice a rychlosti jsou uvedeny v jednotkách metrů a metrů za sekundu jako vektory v prostoru $[x, y, z]$, hmotnost m je v MeV a náboj Q je v jednotkách elementárního náboje. Pole id slouží pouze k identifikaci.

Najděte id částice s nejvyšší energií danou vztahem

$$E = -\frac{1}{2}EQz + \frac{1}{2}mv^2,$$

kde z je pozice a v je celková rychlost.

Nápověda: $m[\text{kg}] = m[\text{eV}]\frac{e}{c^2}$, kde e a c jsou elementární náboj a rychlost světla v jednotkách SI. Tyto jsou již definovány v `scipy.constants`

```
from scipy.constants import elementary_charge, c
```

2.9 Objektově orientované programování

(OOP) je programovací technika pro spojování dat s funkcemi a oddělování implementačních detailů dílčích problémů od zbytku kódu. Při správném použití může pomoci psát snadno čitelný, rozšiřitelný a znovupoužitelný kód.

V jazyce OOP jsou objekty instancemi tříd. Všechno v Pythonu je objekt. Například číslo 5 je instance třídy `int`. V moderním Pythonu (tj. verze 3 a vyšší) mají pojmy typ a třída stejný význam. Vestavěná funkce `type()` vrací třídu (nebo typ) objektu, např.

```
    >>> type(2)
    <class 'int'>
```

Pro vytvoření nových tříd používáme klíčové slovo `class`, např. pro vytvoření třídy, která reprezentuje rozdíly

```
class Difference:
    def __init__(self, x, y):
        self.x = x
```

```
        self.y = y

    def __str__(self):
        return f"{self.x} - {self.y}"

    def value(self):
        return self.x - self.y
```

Funkce definované uvnitř tříd se nazývají **metody**. První argument, konvenčně nazývaný `self`, odkazuje na objekt, který metodu volá. `__init__` je speciální metoda, která vytváří objekt, `__str__` je speciální funkce, která by měla vytvořit čitelnou textovou reprezentaci objektu (používá se s `print`).

Používání tříd a objektů je přímočaré, např.

```
#__init__() se zavolá s x=3, y=5 a self odkazuje na d1
d1 = Difference(3, 5)
d2 = Difference(30, 50)

print(d1.value()) # -2
print(d2) # "30 - 50"
```

Existuje několik dalších speciálních názvů metod (více viz dokumentace), např. `__add__(self, other)`, která se volá pro `x + y`, kde `x` je `self` a `y` je `other`. Podobně existují `__sub__`, `__mul__` a `__truediv__` pro `-`, `*`, `/`.

Cvičení 2.7. Implementujte třídu `Fraction`, která reprezentuje zlomek, který je inicializován dvěma čísly – čitatelem a jmenovatelem. Třída by měla podporovat základní aritmetiku (`+,-,*,/`) s čísly a dalšími `Fraction`.

Intermezzo 1: *Vytváření vlastních modulů.* Jakýkoli Python soubor může být importován jiným Python souborem jako modul pomocí klíčového slova `import`; název modulu je název souboru (bez koncovky .py). Při `import`u se soubor nejprve spustí, tj. jakýkoli kód, který se nachází mimo definice funkcí, se vykoná. Například vezměme dva soubory ve stejném adresáři

`module.py`:

```
def my_module_function(x):
    return 1 + x

print("Ahoj moduly!")
```

`module_user.py`:

```
import module as m
print(m.my_module_function(1))
```

Po spuštění `module_user.py` se vytiskne "Ahoj moduly!". To obvykle není žádoucí. Abychom zabránili spuštění jakéhokoli kódu při importu a povolili jeho spuštění pouze tehdy, když je soubor spuštěn přímo, můžeme udělat

```
if __name__ == '__main__':
    print("Hello modules.")
```

Zde je `__name__` speciální proměnná definovaná v Pythonu, která obsahuje název modulu přiřazený současnému souboru. Má speciální hodnotu `'__main__'` pouze tehdy, když byl soubor spuštěn přímo.

Aby bylo možné soubor importovat jako modul, Python ho musí být schopen najít. Standardně Python hledá v aktuálním pracovním adresáři (`os.getcwd()`) a v adresářích uvedených v seznamu `sys.path` v modulu `sys`. Pokud chceme načíst balíček z jiného umístění, můžeme přidat jeho adresář do proměnné `sys.path` pomocí `sys.path.append(cesta)`.

3 NumPy, SciPy a matplotlib

3.1 NumPy

Tyto tři knihovny tvoří základ mnoha (pravděpodobně většiny) vědeckých skriptů v Pythonu. NumPy poskytuje numpy pole (*numpy arrays*), rychlý způsob ukládání numerických dat v paměti, a funkce pro práci s nimi. SciPy poskytuje velké množství algoritmů včetně fitování metodou nejmenších čtverců, zpracování signálu nebo prostorového shlukování (*clustering*) a matplotlib umožňuje vytváření vysoce kvalitních a přizpůsobitelných grafů. Všechny tři jsou navrženy pro práci s poli NumPy.

Intermezzo 2: *Instalace balíčků – pip* Balíčky, které nejsou standardně nainstalovány, lze nainstalovat pomocí nástroje příkazového řádku `pip`, který stahuje balíčky registrované v Python Package Index (PyPI, `https://pypi.org/`). Například pro instalaci balíčku `lmfit` spusťte v příkazovém řádku následující příkaz (funguje také v ipython REPL, např. ve Spyderu)

```
    pip install lmfit
```

Pro odstranění balíčku použijte `uninstall` a pro upgrade `install -U`. Spusťte `pip help` pro úplný seznam dostupných příkazů. Pokud je vaše virtuální prostředí aktivováno, pip instaluje balíčky do tohoto virtuálního prostředí.

Pokud používáte `uv`, příkaz `uv pip` je kompatibilní se standardním `pip`, např. můžete použít `uv pip install lmfit`. Nebo, pokud chcete svůj projekt zabalit pro použití jinými lidmi, můžete použít `uv add lmfit`, což také přidá `lmfit` jako závislost vašeho balíčku.

Abychom mohli NumPy používat, musíme ho importovat pomocí

```
import numpy as np
```

Existuje více způsobů vytváření polí

```
#přímo ze seznamů
arr = np.array([1,2,3,4])
#2D pole
arr_2D = np.array([[1,2,3],
                   [4,5,6],
                   [7,8,9]])
# podobné list(range()), ale ani start, stop, ani step nemusí být celá čísla
arr = np.arange(start, stop, step)
#lineárně rozložené hodnoty
arr = np.linspace(start, stop, count)
# vyplněné nulami
np.zeros(length)
# vyplněné jedničkami
np.ones(length)
```

Listing 7: Tvorba polí.

Standardně numpy pole ukládají typ float. Pro jedničky a nuly je často užitečné použít booleovské hodnoty (True a False) explicitním specifikovaním datového typu pomocí `dtype`, t.j., `np.ones(10, dtype=bool)` vytvoří pole o délce 10 vyplněné hodnotami True. Aritmetika funguje po prvcích, proto při sčítání/násobení/dělení/porovnávání musí mít pole stejnou délku, např.

```
>>> np.array([1,2,3]) + np.array([4,5,6])
array([5,6,7])
```

Pokud je však jeden z operandů skalární číslo, operace se *rozesílá* (broadcasts), např.

```
>>> np.array([1, 2, 3]) + 10
array([11, 12, 13])
```

Řezání polí (slicing) funguje podobně jako řezání seznamů, např. `arr[start:stop:step]`. Navíc můžeme také indexovat do pole pomocí seznamu indexů, např. `arr[[0, 2, 4]]` vrátí pole obsahující 1., 3. a 5.

prvek pole `arr`. Pomocí booleovských polí můžeme taky vytvořit podmnožinu prvků pole, kde je indexovací pole pravdivé, např.

```
arr1 = np.linspace(0, 10, 50)
b = np.sqrt(arr1) > 3 # pole booleovských hodnot True, kde np.sqrt(arr1) > 3
arr1[b] # pouze prvky arr1, jejichž druhá odmocnina je větší než 3
```

Cvičení 3.1. Napište program, který (ve smyčce) sečte čísla $1 + 2 + 3 + ... + n$ pro libovolné n pomocí polí `numpy` a porovná výsledek s Gaussovým vzorcem $n(n + 1)/2$ pro uživatelem zadané n. *Nápověda:* `np.arange`

Cvičení 3.2. Napište funkci, která vrátí všechna prvočísla menší než N pomocí metody Eratosthenova síta. *Nápověda:* `enumerate()`, `np.ones(N, dtype=bool)`, maskování booleovským polem.

3.1.1 Základní míry a operace s poli

Jelikož jsou pole numpy určena především pro numerická data, mají v sobě zabudováno několik metod, které počítají některé základní míry, např.

`arr.mean()`	průměr pole
`arr.std()`	směrodatná odchylka, vychýlený odhad rozptylu
`arr.sum(), arr.cumsum()`	suma a kumulativní suma
`arr.max(), arr.min()`	maximum a minimum
`arr.argmax(), arr.argmin()`	index maxima a minima

Jako u každého kontejnerového typu vrací `len(arr)` délku pole. Tyto metody existují také ve formě funkcí (např. můžeme volat `np.mean(arr)`). Dále existují užitečné funkce, které pracují s poli v samotném modulu: `np.diff(arr)` vrací pole rozdílů mezi nejbližšími sousedy (výsledek je o jeden prvek kratší než `arr`); pro obecnější vážené průměrování lze použít `np.average(arr, weights)`. Většina těchto funkcí má své odpovídající nan-verze, které ignorují hodnoty NaN v polích (např. `np.nansum`, `np.nanmax` atd.). `np.isnan(arr)` vrací pole stejné délky jako `arr` s hodnotou True všude tam, kde `arr` obsahovalo NaN. Pro pole s booleovskými hodnotami slouží místo operátorů `and` a `or` funkce `np.logical_and(arr1, arr2)` a `np.logical_or(arr1, arr2)` a podobně `np.logical_not(arr)`, které vytvářejí nové pole s logickou operací aplikovanou na každý prvek vstupních polí. Logické operátory samy o sobě s poli nefungují.

Intermezzo 3: *NaNs and Infs.* Matematické operace mohou vést buď k Not-a-Number (NaN) nebo nekonečnům, které jsou v numpy reprezentovány jako `np.nan` a `np.inf`. Jsou to speciální hodnoty indikující, pro NaN, že byla provedena nedefinovaná operace, např. `np.log(-1)` (Python a Numpy podporují komplexní čísla, ale předpokládá se, že operace s reálnými čísly vedou také k reálným číslům – pokud bychom chtěli použít komplexní logaritmus, museli bychom mu dát komplexní -1, tj. `np.log(-1 + 0j)`).

Jednoduché dělení nulou, např. 1/0, způsobí chybu ZeroDivisionError. Pokud je to však součástí `np.array`, výsledkem bude ±np.inf a varování (Warning), např. zkuste spustit `np.array(1)/np.array(0)`.

Myšlenka nehavarovat při nedefinovaných operacích nebo dělení nulou spočívá v tom, že často je poškozena pouze část pole (např. -1 může být zástupný symbol pro nedostupnou hodnotu) a může být jednodušší a čistší pokračovat za předpokladu, že všechny operace jsou platné, a na konci jednoduše vyhodit NaN hodnoty.

Pole lze seřadit na místě pomocí `arr.sort()` nebo lze vytvořit nové seřazené pole pomocí `sorted_arr = np.sort(arr)`. Pro získání pole indexů, které by pole seřadilo, můžeme použít `np.argsort()`, což je užitečné, když chceme seřadit jedno pole podle druhého, např.

```
    import numpy as np
    items = np.array(["Eggs", "Bread", "Apples"])
    prices = np.array([50, 20, 40])

    sort_ix = np.argsort(prices)
    print("Items sorted according to prices:")
    print(items[sort_ix])
```

3.1.2 Vstup/výstup se soubory

Budeme používat numpy a matplotlib k analýze experimentálních dat. Nejprve musíme data získat, obvykle z nějakého souboru na disku. Za předpokladu, že máme data uložena jako textový soubor v `data.txt` ve dvou sloupcích čísel oddělených tabulátory, můžeme je načíst do 2D pole numpy pomocí

```
    arr = np.loadtxt("data.txt", comments='%', skiplines=3)
```

což načte `data.txt` do 2D pole `arr`, ignoruje všechny řádky začínající % a přeskočí první tři datové řádky. Pokud jsou naše data oddělena čárkami (tj. soubor `.csv`) místo tabulátorů, můžeme přidat klíčový argument `delimiter=','`.

Intermezzo 4: *Paths* Cesty k souborům nebo adresářům mohou být absolutní nebo relativní. Absolutní cesta specifikuje absolutní pozici souboru v souborovém systému, na Windows bude typicky začínat něčím jako `C:\` a na Linuxu nebo Macu kořenovým adresářem `/`.

Relativní cesta je vztažena k aktuálnímu pracovnímu adresáři, který lze získat pomocí `getcwd()` z modulu `os`.

Různé operační systémy používají různé oddělovače adresářů, tj. Windows používá `\`, Linux a Mac `/` a japonské počítače s Windows používají `¥`. Pro spojení názvů adresářů způsobem, který bude fungovat všude, můžeme použít `os.path.join('dir1', 'dir2', 'dir3', ...)`.

Pro nalezení více souborů, jejichž název odpovídá vzoru, můžeme použít funkci `glob()` z modulu `glob`. Například `glob(os.path.join("photos", "photo*.png"))` vrátí seznam všech souborů, které odpovídají vzoru `"photos/photo<libovolný počet libovolných znaků>.png"`.

Použití funkcí pro vstup/výstup souborů ze standardní knihovny Pythonu je také možné. Funkce `file = open(filename, mode)` otevře soubor v režimu daným `mode`, nejběžnější jsou `'r'` pro čtení, `'w'` pro (pře)psaní a `'a'` pro připojení obsahu na konec souboru. Veškerý obsah souboru lze přečíst najednou pomocí `file.readlines()` nebo můžeme iterovat přes řádky souboru pomocí cyklu for (viz příklad níže). Inverzně existuje `file.write(some_string)` (pokud je soubor otevřen v režimu 'r', toto selže). Zvláště pro zápis je důležité zavolat `file.close()` poté, co jsme se souborem hotovi, jinak se změny nemusí zapsat na disk kvůli cachování.

Funkce `np.loadtxt()` s výchozími argumenty je zhruba ekvivalentní

```
    import numpy as np
    def my_load(fn):
        with open(fn, 'r') as file:
            rows = []
            for line in file:
                # řádek po řádku proměníme všechny hodnoty oddělené mezerami na čísla
                row = [float(s) for s in line.strip().split()]
                rows.append(row)
        return np.array(rows)
```

Intermezzo 5: *with statement (basic)* `with` je příkladem tzv. správy kontextu, která zajišťuje, že zdroje (v tomto případě soubor) jsou po dokončení práce s nimi řádně uvolněny. Později se naučíme vytvářet vlastní správce kontextu, prozatím je kód

```
    with open(fn, 'r') as file:
        do_stuff()
```

zhruba ekvivalentní

```
    file = open(fn, 'r')
    do_stuff()
    file.close()
```

s tou výhradou, že `with` zajišťuje, že `file.close()` se spustí, i když `do_stuff()` způsobí pád programu.

Cvičení 3.3. Rozšiřte funkci `my_load(fn)` z příkladu výše tak, aby podporovala komentáře v souborech.

3.2 matplotlib

matplotlib umožňuje vykreslování dat. Data vykreslujeme do os (axes), které jsou obsaženy v obrázcích (figures). Obrázek může obsahovat více os. Nejjednodušší způsob, jak vytvořit nový obrázek s novými osami, je [4]

```
    import matplotlib.pyplot as plt
    fig, ax = plt.subplots()
```

Pokud máme data v polích `x` a `y` stejné délky, můžeme je vykreslit například pomocí:

```
    ax.plot(x, y, '-') # spojí dvojice bodů dané x a y plnými čarami
    # logaritmická osa y, čárkovaná čára se čtvercovými značkami
    ax.semilogy(x, y, '--s')
    ax.loglog(x, y, ':') # logaritmická osa x a y, tečkovaná čára
    ax.scatter(x, y) # bodový graf, nespojuje body čarami
```

Listing 8: Základní vykreslování.

Formát vykreslování je specifikován pomocí formátovacího řetězce bezprostředně za daty, který má formát `fmt = [marker][line][color]`.[5] Sílu čáry můžeme specifikovat pomocí klíčového slova `lw=`, velikost značky pomocí `ms=` a barvu pomocí `color=`. [6] Všimněte si, že matplotlib nezajímá, zda dvojice polí x a y představuje matematickou funkci; jednoduše spojí body v pořadí čarami.

Pro popis os můžeme použít `ax.set_xlabel()` a `ax.set_ylabel()`. Data lze označit předáním klíčového argumentu `label=` s řetězcem kterémukoli z vykreslovacích příkazů a následným zavoláním `ax.legend()`. Obrázek můžeme uložit pomocí `fig.savefig(filename)`.

Například pro vykreslení Gaussovy křivky s purpurovými čarami a azurovými body můžeme udělat

```
import numpy as np
import matplotlib.pyplot as plt

xs = np.linspace(0, 10, 100)
ys = np.exp(-(xs - 5)**2)

plt.close('all') #zavře všechny dosud otevřené obrázky
fig, ax = plt.subplots()

#velikost bodu (marker size, ms) 5
ax.plot(xs, ys, '--s', ms=5, lw=2, color='magenta', label=r'ax.plot($e^{-x^2}$)')
#scatter umožňuje proměnnou velikost bodů pomocí argumentu s
#Všimněte si, že na vrcholu peaku jsou azurové body větší, než na okrajích
ax.scatter(xs, ys, s=ys*10, marker='o', color='cyan', zorder=3,
           label=r'ax.scatter($e^{-x^2}$)')

```

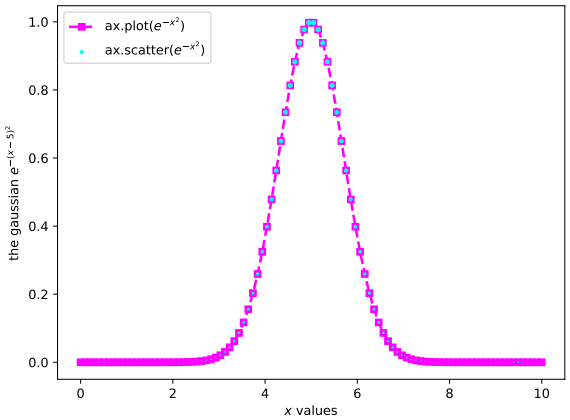
[4] Všimněte si, že `plt.subplots()` vrací dvě hodnoty. Vrací pouze jednu n-tici, která je destrukturována neboli rozbalena do dvou proměnných `fig` a `ax`.

[5] Viz dokumentaci pro úplný seznam možných formátů.

[6] Viz dokumentaci pro všechny pojmenované barvy.

```
ax.set_xlabel('$x$ values')
ax.set_ylabel(r'the gaussian $e^{-(x-5)^2}$')
#kam umístit legendu, můžeme také použít "best" a nechat matplotlib hádat
ax.legend(loc='upper left')

fig.tight_layout() #redukuje některé bílé místo kolem okrajů
fig.savefig('gaussian.pdf') #formát souboru obrázku je odvozen z přípony
```

Listing 9: Kompletní příklad vykreslování.



Všimněte si, že můžeme ovládat, které objekty se kreslí nad kterými, pomocí klíčového argumentu `zorder=` a že je podporována základní sazba matematiky pomocí LaTeX, ale příkazy latexu začínající zpětným lomítkem je třeba buď escapovat (tj. `"$\\frac{a}{b}$"`) nebo použít v raw řetězcích (tj. `r"$\frac{a}{b}$"`).

Cvičení 3.4. Napište program, který pomocí NumPy a Matplotlib vykreslí výrazy $y = x$, $y = x^2$ a $y = \sqrt{x}$ od $x = 0$ do $x = 5$ s různými styly čar (např. plná, čárkovaná, tečkovaná) a uživatelem zadaným počtem bodů. Vyzkoušejte lineární, semilogaritmické a logaritmické osy. Dejte osám popisky a křivkám legendu.

Cvičení 3.5.

- Napište funkci, která zintegruje libovolnou závislost danou body x_n, y_n, t.j., diskrétní verzi $\int y(x)dx$, a použije ji na spočtení integrálu $y = x^2$ od 0 do 5. (*Nápověda: přibližná hodnota integrálu je dána* $\sum_{n=0}^{N-2} 0.5 dx_n(y_{n+1} + y_n); dx_n = x_{n+1} - x_n$, $n = 0 \ldots N-1$)
- Použitím funkce z předchozího bodu napište funkci, která zintegruje libovolnou Python funkci jedné proměnné, která vrací číslo, mezi body a a b. Počet bodů N by měl být volitelný, s výchozí hodnotou 100. Nápověda: funkce jsou objekty.
- Graficky ukažte konvergenci integrálu z předchozího bodu ke skutečné hodnotě se zvyšujícím se počtem diskretizačních bodů N a porovnejte ji s funkcí `scipy.integrate.quad`. Zobrazte absolutní chybu v logaritmickém měřítku jako funkci N. Pro test konvergence můžete použít integrál

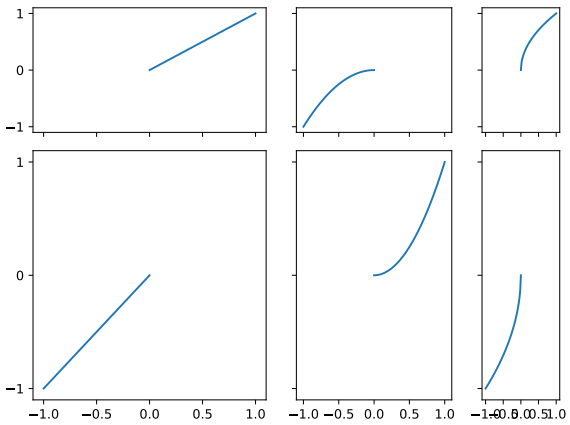
$$\int_0^1 e^{-x} \mathrm{d}x$$

Funkce z předchozích bodů dejte do samostatného .py souboru a použijte ho jako modul.

Jeden obrázek může obsahovat více os vytvořených předáním volitelných argumentů rows a columns funkci `plt.subplots()`. Více os je vytvořeno v pravidelné mřížce se zadaným počtem řádků a sloupců. Pro osy nerovných velikostí můžeme specifikovat poměry jejich šířek a výšek pomocí `width_ratios` a `height_ratios` (viz `gridspec` [7] pro složitější rozvržení obrázků). Osy mohou sdílet rozsahy x a y specifikováním booleovských klíčových argumentů `sharex` a `sharey`, například

```
import numpy as np
import matplotlib.pyplot as plt
nrows = 2
ncols = 3
fig, axes = plt.subplots(nrows, ncols, sharex=True, sharey=True,
                         width_ratios=[3, 2, 1], height_ratios=[1,2])
#axes is now a 2D array of axes which all share the same x and y range

xs = np.linspace(0, 1)

axes[0,0].plot(xs, xs)
axes[1, 1].plot(xs, xs**2)
axes[0, 2].plot(xs, np.sqrt(xs))

axes[1,0].plot(-xs, -xs)
axes[0, 1].plot(-xs, -xs**2)
axes[1, 2].plot(-xs, -np.sqrt(xs))

fig.tight_layout()
fig.savefig('multiax.pdf')
```

Listing 10: Více os v jednom obrázku.



Cvičení 3.6. Napište program, který načte data ze souboru `data.txt`, který obsahuje tři sloupce čísel oddělené tabulátory. Nazveme tyto sloupce frekvence, X a Y. Nakreslete frekvenční závislosti X a Y

1. ve stejných osách (X plná čára, Y čárkovaná čára)
2. v oddělených osách ve stejném obrázku se sdíleným rozsahem x a y

[7] `https://matplotlib.org/3.5.0/tutorials/intermediate/gridspec.html`

Vykreslete také X vs. Y v samostatném grafu. Osám dejte vhodné popisky.

Cvičení 3.7. Najděte maximum a minimum absolutní hodnoty $R = \sqrt{X^2 + Y^2}$ a vyznačte jejich pozice svislými čarami v grafu frekvenční závislosti X a Y z předchozího cvičení. Odhadněte plnou šířku v polovině maxima (full width at half maximum, FWHM) píku a činitel jakosti f_0/FWHM, kde f_0 je frekvence maximální odezvy.

Hint: `np.argmin, np.argmax, ax.axvline, ax.axhline`

Cvičení 3.8. Zpracujte všechny soubory z adresáře `lots_of_data` podobně jako v Cvič. 7. Vykreslete FWHM jako funkci f_0. Použijte řešení Cvič. 7 jako modul.

Hint: `glob`

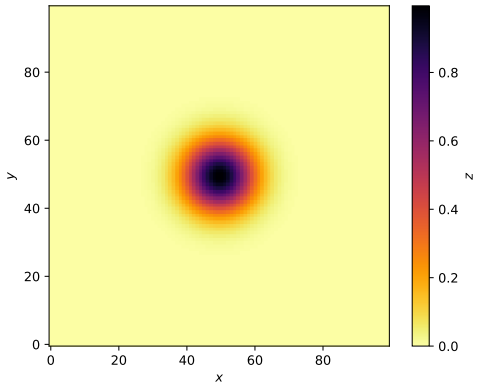
Veličiny, které závisí na dvou řídicích proměnných, lze často vykreslit pomocí teplotních map (heat maps), pro které můžeme použít `ax.imshow(arr2D)`, která vezme 2D pole čísel a mapuje je na barvu pixelu pomocí barevné mapy (colormap)[8] Všimněte si však, že `imshow()` se primárně používá pro obrázky, které konvenčně začínají v levém horním rohu s levotočivými osami. Data typicky začínají v levém dolním rohu s pravotočivými osami. To lze změnit specifikováním klíčového slova `origin='lower'` pro `imshow()`. Alternativně, pro vykreslování dat, která nejsou na pravidelné mřížce, můžeme použít `ax.pcolormesh(X, Y, Z)`, kde `X, Y, Z` jsou 2D pole.

Například pro vykreslení 2D Gaussovy křivky s barevnou mapou `'inferno_r'`,

```
import numpy as np
import matplotlib.pyplot as plt

#x and y axis
_xs = np.linspace(0, 10, 100)
_ys = np.linspace(0, 10, 100)
# my ale potřebujeme 100 x 100 bodů pro x i y které vzorkují
# celý interval (0, 10) x (0, 10), to lze provést pomocí
#meshgrid
xs, ys = np.meshgrid(_xs, _ys) #xs, ys jsou 2D pole
zs = np.exp(-(xs - 5)**2 - (ys - 5)**2) # a zs taky

plt.close('all') #zavře všechny dosud otevřené obrázky
fig, ax = plt.subplots()

plot = ax.imshow(zs, cmap='inferno_r', origin='lower')
#podobně
#plot = ax.pcolormesh(xs, ys, zs, cmap='inferno_r')
cbar = fig.colorbar(plot) # barevná škála (colorbar)
cbar.set_label('$z$')
ax.set_xlabel('$x$')
ax.set_ylabel('$y$')
```

Listing 11: Příklad dvourozměrného vykreslování.

[8] Viz `https://matplotlib.org/stable/users/explain/colors/colormaps.html` pro úplný seznam názvů barevných map.



Cvičení 3.9. Vykreslete Mandelbrotovu množinu s konfigurovatelným rozsahem a rozlišením.

Nápověda: Mandelbrotova množina je množina komplexních čísel c, pro která řada $z_{n+1} = z_n^2 + c,\ z_0 = 0$ nediverguje. Pokud $|z_n| \geq 2$ pro jakékoli n, řada bude divergovat. Jako kritérium konvergence můžeme použít, že $|z_n| < 2\ \forall n < N_{\max}$ ($N_{\max} = 100$, například). Vykreslete množinu pomocí `plt.imshow(c)`, kde `c[i,j]` je počet iterací potřebných k překročení $|z_n| = 2$ (nebo $N_{\max}$). Celá množina je obsažena v obdélníku s levým dolním rohem $-2-i$ a pravým horním rohem $1+i$ v komplexní rovině.

3.2.1 Přehled základních příkazů pro vykreslování

Za předpokladu, že obrázek a osy byly vytvořeny jako `fig, ax = plt.subplots()`, základní příkazy pro manipulaci s grafem jsou

`ax.xlabel("popisek x"), ax.ylabel("popisek y")`	nastaví popisky os
`ax.set_xlim(xmin, xmax), ax.set_ylim(ymin, ymax)`	nastaví limity os, xmin, xmax atd. lze také použít jako klíčová slova
`ax.set_aspect('equal')`	nastaví poměr stran os na stejný, užitečné, když obě osy obsahují kvalitativně podobná data
`ax.legend(loc=location)`	zobrazí legendu na místě `location` ∈ `"upper|lower left|right"` nebo `"best"`
`fig.tight_layout()`	upraví velikost os tak, aby se vešly všechny popisky a zmenšil se bílý prostor
`fig.supxlabel('xlabel'), fig.supylabel('ylabel')`	nastaví společné popisky os pro celý obrázek s více osami
`fig.colorbar()`	vytvoří barevnou škálu v obrázku
`plt.close('all')`	zavře všechny otevřené obrázky

4 Fitování a interpolace

Mějme data reprezentovaná sadou bodů $[(x_i, y_i)]$, kde x_i je řídicí proměnná a y_i je pozorovatelná veličina (např. řídím proud rezistorem a pozoruji napětí). Fit metodou nejmenších čtverců na funkci $f(x; p_1, p_2, ...p_n)$ minimalizuje sumu čtverců reziduí, tj.

$$R^2 = \sum_i |f(x_i, \{p\}) - y_i|^2 . \tag{1}$$

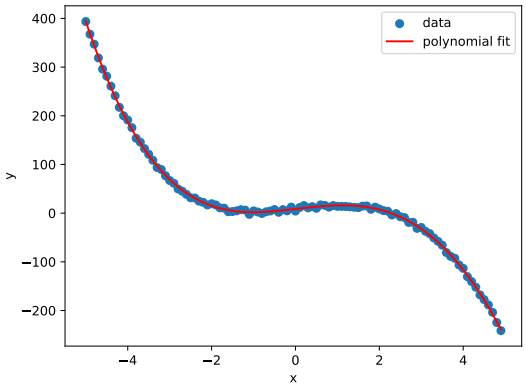
Výsledkem fitu je sada parametrů $\{p\} = p_1, p_2, p_3, ...$, které minimalizují R^2. V závislosti na tom, jak funkce f závisí na parametrech p_k, mluvíme o *lineárním* nebo *nelineárním* fitování. Důležitý rozdíl je v závislosti na parametrech p, nikoli na řídicí proměnné x.

4.1 Polynomiální fitování

Fitování polynomem v x, kde parametry fitu jsou koeficienty polynomu, je velmi častým příkladem lineárního fitování. Rozhraní pro použití polynomů je obsaženo v podmodulu `np.polynomial` ve třídě `Polynomial`, která poskytuje metodu `fit`. Koeficienty polynomu získáme pomocí metody `Polynomial.convert().coef`. Volání metody `convert()` je nutné, protože `Polynomial` vnitřně mapuje rozsah dat x na interval $[-1, 1]$ (typicky). Příklad použití:

```
import numpy as np
import matplotlib.pyplot as plt

from numpy.polynomial import Polynomial as P

rng = np.random.default_rng() # random number generator

#%% create some noisy data and plot them
noise_amplitude = 10
xs = np.arange(-5, 5, 0.1)
ys = 0.1*xs**4 - 3*xs**3 + 10*xs + 4 + rng.random(len(xs))*noise_amplitude

fig, ax = plt.subplots()
ax.scatter(xs, ys, label='data')

#%% fit the polynomial
poly = P.fit(xs, ys, deg=4)
ax.plot(xs, poly(xs), color='r', label='polynomial fit')
#the coefficients of poly are scaled for a particular domain/window
#to get the ordinary coefficients we need to use .convert()
coef = poly.convert().coef
poly_string = ' + '.join([f'({c:.1f})*x^{k}' for k, c in enumerate(coef)])
print("The estimated polynomial is", poly_string)

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.legend(loc='best')
fig.savefig('polynomial_fit.pdf')
```

Listing 12: Polynomiální fitování



Cvičení 4.1. Odečtěte pozadí od píků ve cvičení 7 pomocí polynomiálního fitu.
Nápověda: `np.polynomial.Polynomial.fit()` a booleovská pole

4.2 Nelineární fitování křivek

Když fitovací funkce závisí nelineárně na parametrech fitu, musíme použít nelineární fitování. Pro tento úkol existuje několik knihoven. Pro základní fitování ale můžeme použít podmodul `scipy.optimize` z `scipy`.

Pro fitování dané funkce na data můžeme použít `scipy.optimize.curve_fit`, např.

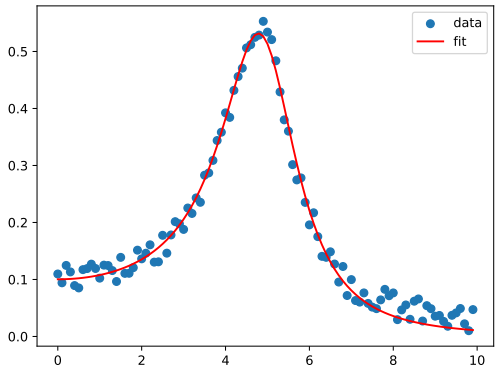
```
import numpy as np
import matplotlib.pyplot as plt

from scipy.optimize import curve_fit

rng = np.random.default_rng()

#%% function to fit
def lorentzian(xs, height, center, width):
    return height*center**2*width**2/((center**2 - xs**2)**2 + width**2*xs**2)

#%% generate some data and plot
xs = np.arange(0, 10, 0.1)
height = 0.5
center = 5
width = 2
noise = 0.05
ys = lorentzian(xs, height, center, width) + rng.random(len(xs))*noise

fig, ax = plt.subplots()
ax.scatter(xs, ys, label='data')

#%% fit the data
popt, pcov = curve_fit(lorentzian, xs, ys, p0=[1, 1, 1])
# curve_fit returns the optimized parameters (popt) and the covariance matric (pcov)
# the diagonal of the covariance matrix can be used as simple error estimates of the
    parameters
ax.plot(xs, lorentzian(xs, *popt), color='r', label='fit')
#                          ^
# this substitutes (positionally) elements of the iterable into the function call

print(f"Estimated height: {popt[0]:.3f} +/- {np.sqrt(pcov[0,0]):.3f}")
print(f"Estimated center: {popt[1]:.3f} +/- {np.sqrt(pcov[1,1]):.3f}")
print(f"Estimated width: {popt[2]:.3f} +/- {np.sqrt(pcov[2,2]):.3f}")
```

```
34
35 ax.legend(loc='best')
36 fig.savefig('curve_fit.pdf')
```

Listing 13: Nelineární fitování křivek

Výstup:

```
1     Estimated height: 0.518 +/- 0.007
2     Estimated center: 5.039 +/- 0.016
3     Estimated width: 2.195 +/- 0.036
```

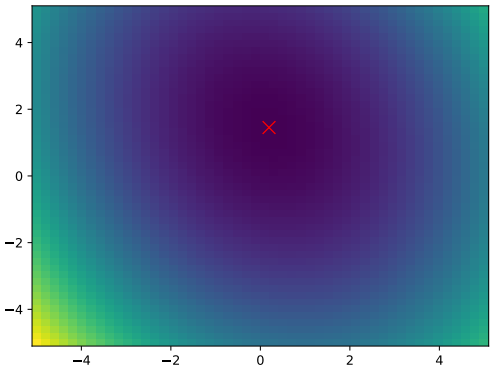


Na rozdíl od lineárních nejmenších čtverců je nelineární fitování křivek iterativní a zastaví se, jakmile parametry skonvergují. U některých kombinací dat a funkce je možné, že fit nemusí nikdy skonvergovat, proto po překročení nakonfigurovaného maximálního počtu iterací fitovací rutiny *vyvolají výjimku* (viz 6), která shodí program, pokud není ošetřena.

Pro obecnější optimalizační problémy můžeme použít `scipy.optimize.minimize()`. Funkce `minimize()` přijímá jednu **skalární** funkci (t.j., vracící jedno číslo) a počáteční odhad optimálních parametrů. Například pro nalezení minima dvourozměrné paraboly,

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  import scipy.optimize as optim
4
5  def parabola(x, y):
6      return 4*x**2 + 2*y**2 + x*y - 6*y -3*x + 5
7
8  #the lambda is there to simply turn a function of two parameters into a
9  #function of a single parameter
10 xmin = optim.minimize(lambda z: parabola(*z), x0=[0,0])
11
12 XX, YY = np.meshgrid(np.linspace(-5, 5), np.linspace(-5, 5))
13 p = parabola(XX, YY)
14 fig, ax = plt.subplots()
15 ax.pcolormesh(XX, YY, p)
16 ax.plot(*xmin.x, 'x', color='r', ms=10)
17 fig.savefig('parabola-minimize.pdf')
```

Listing 14: Minimalizace skalární funkce více parametrů



Odhad chyby parametrů Fitovací funkce `curve_fit` vrací dvě hodnoty – pole parametrů, které nás zajímají, a kovarianční matici parametrů, tj. $\mathrm{cov}(p_i, p_j) = \langle (p_i - \bar{p}_i)(p_j - \bar{p}_j) \rangle$. Diagonální hodnoty kovarianční matice lze použít jako odhad nejistot parametrů fitu, tj.

```
    #assuming that f is defined as f(x, p1, p2)
    p, cov = curve_fit(f, x, y)
    print(f"p1 = {p[0]} +/- {np.sqrt(cov[0,0])}")
```

Kovarianční matice se počítá z *reziduí* $r_i = y_i - f(x_i, \{p_j\})$, kde p_j jsou optimalizované parametry, jako ($\mathbf{r}$ je vektor s prvky r_i; $i = 1 \ldots N$, $j = 1 \ldots M$; N je počet datových bodů a M je počet parametrů) škálovaná inverze Hessovy matice H optimalizované funkce $\chi^2 = \sum r_i^2$[9]

$$\mathrm{cov}(p_i, p_j) = \frac{1}{N - M} \begin{pmatrix} \frac{\partial \chi^2}{\partial p_1 \partial p_1} & \frac{\partial \chi^2}{\partial p_1 \partial p_2} & \cdots \\ \frac{\partial \chi^2}{\partial p_1 \partial p_1} & \frac{\partial \chi^2}{\partial p_2 \partial p_2} & \cdots \\ \vdots & \vdots & \ddots \end{pmatrix}^{-1}, \tag{2}$$

Obvykle nás zajímá pouze diagonála pro přímý odhad chyby parametrů fitu, všimněte si, však ale, že pokud je například studovaná veličina součtem dvou parametrů fitu $z = A + B$, pak rozptyl z je

$$\mathrm{var} z = \langle (A+B-\bar{A}-\bar{B})^2 \rangle = \langle (A-\bar{A})^2 \rangle + \langle (B-\bar{B})^2 \rangle + 2\langle (A-\bar{A})(B-\bar{B}) \rangle = \mathrm{var} A + \mathrm{var} B + 2\mathrm{cov}(A, B), \tag{3}$$

a musí být použity i mimodiagonální členy kovarianční matice.

Odhad chyb vypočtený pomocí kovarianční matice však může být často podhodnocen, protože výše uvedená metoda je platná pouze tehdy, když je modelová funkce f správná a data mají skutečně tvar $y_i = f(x_i, p) + e_i$, kde chyby dat e_i mají nulovou střední hodnotu a normální rozdělení. Robustnější, ale také výpočetně náročnější metodou odhadu chyb parametrů je **bootstrap**, kde pro několik opakování vytvoříme náhodný výběr dat (stejné délky), provedeme fit na každé vytvořené sadě a vypočítáme průměr a směrodatnou odchylku (a v principu i kovarianci) na výsledné sadě parametrů fitu. Příklad implementace je ukázán v Lst. 15.

```
#%% fit the data
#...
bootstrap_N = 100
parameter_N = 3
data_N = len(xs)
ps = np.empty((bootstrap_N, parameter_N)) # here we will save all fit parameters
for k in range(bootstrap_N):
    # some data will be omitted, some will occur more than once
    ix = rng.choice(range(data_N), data_N)
```

[9] Výpočet je mírně složitější, ale podobný, když mají datové body různé váhy, viz dokumentace

```
    ix.sort()
    popt, _ = curve_fit(lorentzian, xs[ix], ys[ix], p0=[1, 1, 1])
    ps[k,:] = popt

# the fit function does not depend on the sign of the parameters center and width
# and curve_fit will randomly find one or the other. Make sure that we are averaging
# the same signs
ps = abs(ps)
# do the statistics on the set of parameters ps, which is a 2D array.
# work only along one axis, we don't want to average everything together
popt_bootstrap = np.mean(ps, axis=0) #1st axis (0th) averages over the rows
perr_bootstrap = np.std(ps, axis=0) #the same for standard deviation
ax.plot(xs, lorentzian(xs, *popt_bootstrap), color='r', label='fit')

print(f"Estimated height: {popt_bootstrap[0]:.3f} +/- {perr_bootstrap[0]:.3f}")
print(f"Estimated center: {popt_bootstrap[1]:.3f} +/- {perr_bootstrap[1]:.3f}")
print(f"Estimated width: {popt_bootstrap[2]:.3f} +/- {perr_bootstrap[2]:.3f}")
```

Listing 15: Odhad chyb parametrů fitu pomocí bootstrapu. Data jsou vytvořena stejným způsobem jako v Lst. 13.

Intermezzo 6: *Výjimky a ošetřování chyb* Výjimky jsou mechanismus, který Python používá k signalizaci chyb nebo jiných událostí, které musí váš kód ošetřit. Pokud *vyvolaná* výjimka není *zachycena*, program spadne. K ošetření výjimek používáme bloky `try`, `except` a `finally`. Výjimku můžeme vyvolat pomocí `raise`. Všimněte si, že výjimky nemusí vždy znamenat chybu, například cyklus `for` je ukončen pomocí výjimky `StopIteration`.

Příklad zachycení a vyvolání výjimek a použití bloku `finally`.

```
def faulty_function():
    raise ValueError("blergh!")

xs = [-2, -1, 0, 1, 2]
one_over_xs = []
try:
    for x in xs:
        try:
            one_over_xs.append(1/x)
            if x > 1:
                faulty_function()
        except ZeroDivisionError:
            print("Can't divide by zero!")
        except:
            print("something else went wrong")
            raise #propagate the exception further up
finally:
    print("I will always run")

# unless the exception that is raised on line 9 and then sent forward
# on line 16 isn't  handled, this line will not run
print(one_over_xs)
```

Všimněte si několika věcí:

`except` Může zachytit buď specifický typ výjimky, nebo jakýkoli typ (pokud typ výjimky není specifikován).

Blok `finally` se vždy spustí, bez ohledu na to, zda uvnitř bloku `try` došlo k výjimce, a je obecně určen pro řádné uvolnění zdrojů (např. otevřených souborů).

Bloky `try` mohou být vnořené: Když je vyvolána výjimka, nejvnitřnější blok `try-except` se ji pokusí ošetřit. Pokud není nalezen vhodný `except` nebo je výjimka znovu `raise`-nuta, pokusí se ji ošetřit další obklopující blok `try-except` a tak dále. Pokud se výjimka dostane ze všech vnořených obklopujících bloků `try-except` bez ošetření, program spadne.

`raise` **lze použít kdekoli**, např. ve funkcích, které neobsahují `try`. Je na volajícím kódu, aby rozhodl, co s výjimkami udělá.

Cvičení 4.2. Fitujte absolutní hodnotu $r(f) = \sqrt{x^2 + y^2}$ odezvy ve cvičení 7 na odezvu lineárního harmonického oscilátoru plus polynomiální pozadí (stupeň polynomu 3) a vykreslete výsledky podobně jako ve cvičení 7. Použijte jednoduchý odhad ze cvičení 6 jako počáteční odhady parametrů fitu. Použijte řešení cvičení 6 jako modul.

Bonusové cvičení: udělejte stupeň polynomu pozadí nastavitelný.

Komplexní amplituda odezvy lineárního harmonického oscilátoru na sílu F je (viz Dodatek A)

$$x(\omega) = \frac{F/m}{\omega_0^2 - \omega^2 + i\omega\gamma}, \tag{4}$$

kde ω_0 je (úhlová) rezonanční frekvence, ω je frekvence síly F, m je hmotnost oscilátoru a γ je tlumení.

Cvičení 4.3. Odhadněte chyby parametrů fitu získaných ve cvičení 2 pomocí bootstrapu.

5 Digitální zpracování signálu

5.1 Digitální reprezentace spojitého signálu

Předpokládejme signál $y(t)$ (např. napětí), který se spojitě mění v čase. Pro uložení a zpracování tohoto signálu na počítači ho měříme v sadě časových okamžiků t_i, což nám dává hodnoty signálu y_i. Výsledkem je diskrétní reprezentace signálu jako série párů (t_i, y_i). Signál je vzorkován rovnoměrně, pokud $t_i = i\Delta t$, kde Δt je **vzorkovací (*sampling*) perioda** a $f_s = 1/\Delta t$ je **vzorkovací (sampling) frekvence**.

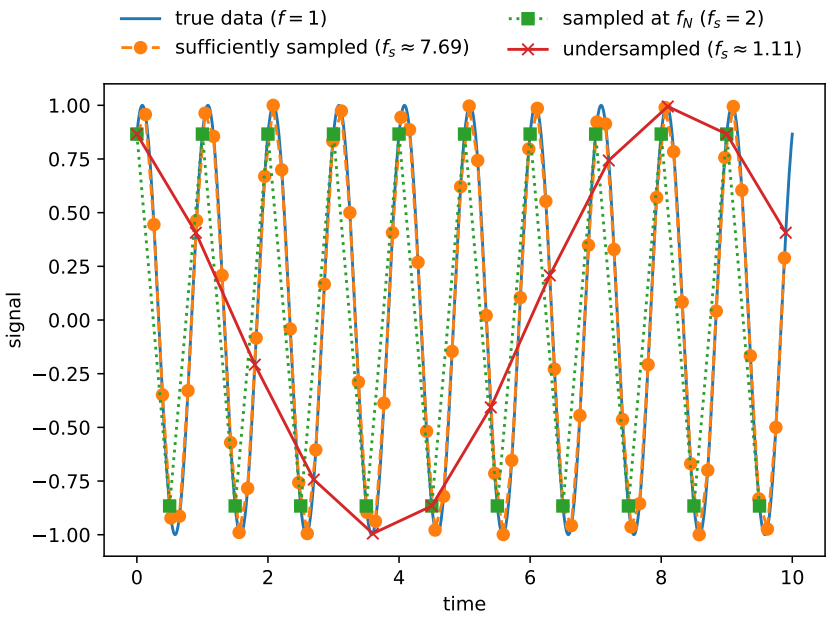
Jakákoli informace o původním signálu $y(t)$, která se mění v čase rychleji než Δt, je ztracena. Pokud však $y(t)$ osciluje na frekvenci f, nestačí mít $f_s \geq f$. Frekvence signálu $f > f_s/2$ se objeví posunuté na $f - f_s/2$, viz výstup Lst. 16. Vzorkovaný signál obsahuje fiktivní frekvence, které v původním signálu nejsou přítomny.

Toto zkreslení se nazývá **aliasing** a *Nyquistův teorém* říká, že aby se předešlo aliasingu, nejvyšší frekvence obsažená v signálu f nesmí být vyšší než polovina vzorkovací frekvence $f_s/2$, která se nazývá **Nyquistova frekvence**

$$f_N = f_s/2. \tag{5}$$

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.interpolate as intp

#create "continuous" data (very high sampling rate)
T = 10 #total length of data in "time" units
xs = np.linspace(0, T, 10000)
ys = np.sin(2*np.pi*xs + 7*np.pi/3)

#interpolate the signal at high sampling rates
signal = intp.interp1d(xs, ys)

def sample(signal, dx, T=T):
    """
    Samples the quasi-continuous signal with
    total length T with sample spacing dx
    """
    xs = np.arange(0, T, dx)
    return xs, signal(xs)

fig, ax = plt.subplots()
ax.plot(xs, ys, '-', label='true data ($f = 1$)')

ax.plot(*sample(signal, 0.13), '--o',
        label=r'sufficiently sampled ($f_s\approx 7.69$)')
ax.plot(*sample(signal, 0.5), ':s',
        label=r'sampled at $f_N$ ($f_s = 2$)')
#this signal will be aliased
ax.plot(*sample(signal, 0.9), '-x',
        label=r'undersampled ($f_s \approx 1.11$)')

ax.set_xlabel('time')
ax.set_ylabel('signal')

ax.legend(ncol=2, frameon=False,
          loc='lower left', bbox_to_anchor=(0.0, 1.01))
fig.tight_layout()
fig.savefig('sampling.pdf')
```

Listing 16: Efekt vzorkovací frekvence



5.2 Spektrální analýza

Jakýkoli periodický signál $y(t)$ (který může být komplexní) s periodou T, tj. $y(t+T) = y(t)$, může být reprezentován jako součet sinových a kosinových členů oscilujících na úhlových frekvencích $2\pi n/T$, kde n je celé číslo,

$$y(t) = \frac{1}{T}\sum_{k=0}^{\infty}\left[A'_k \sin\left(\frac{2\pi k}{T}t\right) + B'_k \cos\left(\frac{2\pi k}{T}t\right)\right], \tag{6}$$

což se nazývá **Fourierova řada**. Ekvivalentně lze Fourierovu řadu vyjádřit pomocí komplexních exponenciál

$$y(t) = \frac{1}{T}\sum_{k=-\infty}^{\infty} A_k e^{2\pi i k t/T}, \tag{7}$$

kde pro reálný signál y jsou kladné a záporné členy komplexně sdružené, $A_k = A^*_{-k}$ (faktor $1/T$ a znaménko uvnitř exponenciální funkce jsou dány konvencí).

Fourierovy koeficienty A_k lze vypočítat jako

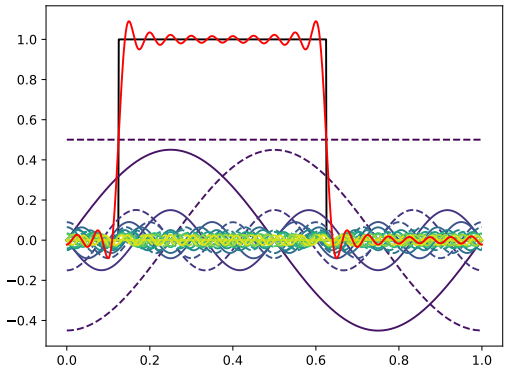
$$A_k = \int_0^T y(t) e^{-2\pi i k t/T} \mathrm{d}t \tag{8}$$

Například, zvažme přímo vypočtenou Fourierovu řadu v Lst. 17

```
import numpy as np
import matplotlib.pyplot as plt

# create a square pulse from t=0.125 to t=0.625
ts = np.linspace(0, 1, 1000)
dt = ts[1] - ts[0]
ys = np.zeros_like(ts)
start = 0.125
ys[np.logical_and(ts > start, ts < start + 0.5)] = 1

plt.close('all')
fig, ax = plt.subplots()
```

```
ax.plot(ts, ys, 'k')

#we will represent our signal y(t) with a Fourier series
# y(t) = sum_k A_k sin(2*pi*k*t) + B_k cos(2*pi*k*t)

#to calculate the individual A_k and B_k, we simply have to integrate
#the signal with the the appropriate sine or cosine term
def Ak(ts, ys, k):
    sint = np.sin(2*np.pi*k*ts)
    # we multiply by 2 because sin(...)^2 averaged over a period is 1/2
    return 2*np.sum(ys*sint)*dt

def Bk(ts, ys, k):
    if abs(k) > 0:
        cost = np.cos(2*np.pi*k*ts)
        return 2*np.sum(ys*cost)*dt
    else:
        #if k==0 (cos(...) == 1) the factor 2 is not needed
        return np.sum(ys)*dt

#calculate the first K fourier coefficients A and B
K = 20
As = [Ak(ts, ys, k) for k in range(K)]
Bs = [Bk(ts, ys, k) for k in range(K)]

#total will be the total sum of the Fourier series
total = 0

#prepare a colormap for plotting
cmap = plt.get_cmap('viridis')
norm = plt.Normalize(vmin=0, vmax=K)

#iterate over all coefficients and frequencies
for k, (A, B) in enumerate(zip(As, Bs)):
    sint = A*np.sin(2*np.pi*k*ts)
    cost = B*np.cos(2*np.pi*k*ts)

    # plot the oscillating terms with the color given
    # by the frequency
    ax.plot(ts, sint, color=cmap(norm(k)))
    ax.plot(ts, cost, '--', color=cmap(norm(k)))

    total += sint + cost

ax.plot(ts, total, '-', color='r')
fig.savefig('../fourier_series_square_pulse.pdf')

#finally, plot the Fouerier spectrum itself. Try changing the phase of the
#signal by changing the start variable on line 7. Individual As and Bs will
#change, but the modulus of the complex number |A + iB| stays the same.
fig_spec, ax_spec = plt.subplots()
ax_spec.plot(range(K), As, 'o', label='sine terms, $A$')
ax_spec.plot(range(K), Bs, 'o', label='cosine terms, $B$')
ax_spec.plot(range(K), np.abs(As + 1j*np.array(Bs)), 's', label='$|A + iB|$')
ax_spec.legend(loc='best')
plt.show()
```

Listing 17: Fourierova řada obdélníkového pulzu.



Tyto definice lze rozšířit na aperiodické signály (které si lze představit jako signály s nekonečně dlouhou periodou), což vede k **Fourierově transformaci**

$$\tilde{y}(\omega) = \int_{-\infty}^{\infty} y(t) e^{-i\omega t} \mathrm{d}t, \tag{9}$$

a **inverzní Fourierově transformaci**

$$y(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \tilde{y}(\omega) e^{i\omega t} \mathrm{d}\omega \tag{10}$$

Absolutní hodnota Fourierových koeficientů $|A|$ je amplituda oscilace signálu na dané frekvenci a jejich komplexní fáze je fázové posunutí sinových a kosinových členů. Pro Fourierovu transformaci, jelikož se frekvence nyní mění spojitě, má $|\tilde{y}|$ význam *hustoty* (nazývané spektrální hustota). V analogii s elektrickým výkonem $P = V^2/R$ potřebným k vytvoření napětí V na rezistoru R, se $|\tilde{y}|^2$ nazývá *výkonová spektrální hustota*, tj. $\int_{\omega_0}^{\omega_1} |\tilde{y}(\omega)|^2 \mathrm{d}\omega$ je výkon signálu ve frekvenčním pásmu (ω_0, ω_1).

Pro digitální signály mluvíme o *diskrétních* Fourierových transformacích. Pro rovnoměrně vzorkované signály, $t_n = n\Delta t$, jsou tyto v SciPy definovány jako (v podmodulu `scipy.fft`)

$$\tilde{y}[k] = \sum_{n=0}^{N-1} y[n] e^{-2\pi i k n/N}, \tag{11}$$

kde $y[n]$ je hodnota signálu y měřená v čase $t_n = n\Delta t$ a N je celkový počet vzorků. Bezrozměrné celočíselné frekvence k běží od 0 do $N-1$ – to znamená, že Fourierova transformace má stejnou délku jako původní signál.

Inverzní diskrétní Fourierova transformace je definována podobně,

$$y[n] = \frac{1}{N} \sum_{k=0}^{N-1} \tilde{y}[k] e^{+2\pi i k n/N}. \tag{12}$$

Celočíselné frekvenční indexy k reprezentují frekvence $f'_k = k/N$ pro $k = 0 \ldots N/2$ a $f'_k = -k/N$ pro $k = N/2 \ldots N-1$. Pro získání skutečných frekvencí stačí škálovat f'_k skutečnou vzorkovací frekvencí $1/\Delta t$. Všimněte si, že definice diskrétní Fourierovy transformace a její inverze nezávisí na vzorkovací frekvenci, pouze na tom, že signál je vzorkován rovnoměrně.

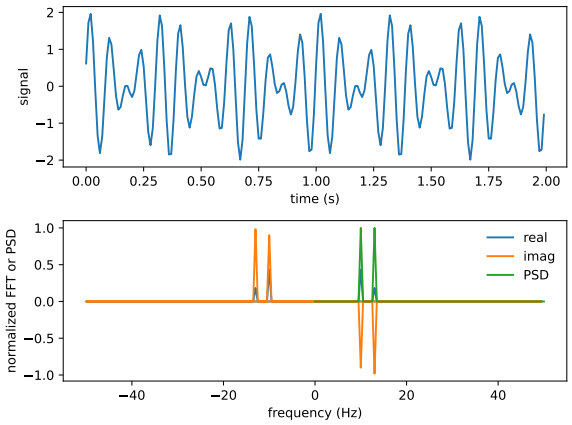
Fourierovy transformace jsou implementovány v NumPy a SciPy pomocí algoritmu **Rychlá Fourierova transformace – FFT**[10] v podmodulu `scipy.fft`. Fourierova transformace se počítá pomocí `scipy.fft.fft()` a skutečné frekvence lze vypočítat pomocí pomocné funkce `scipy.fft.fftfreq()`. Obě funkce `fft` a

[10] FFT je jedním z nejdůležitějších algoritmů, na kterých závisí celý digitální svět – např. kódování zvuku a videa a bezdrátová komunikace na něm silně spoléhají.

`fftfreq` vracejí kladné i záporné frekvence. Pro reálné signály neposkytují záporné frekvence žádné další informace, proto můžeme použít `scipy.fft.rfft()` a `rfftfreq()`, které vracejí pouze kladné frekvence (tj. výsledek má poloviční délku původního signálu). Pokud nás zajímá pouze výkonová spektrální hustota, existuje `scipy.signal.periodogram()`, která také počítá frekvence. Ve výchozím nastavení pro reálné signály vrací `periodogram` pouze kladné frekvence a pro komplexní signály kladné i záporné frekvence. Viz Lst. 18 pro příklad použití a Lst. 19 pro verzi programu v Lst. 17, ale s použitím FFT místo ručního výpočtu koeficientů.

```
import numpy as np
import scipy.fft as fft
from scipy.signal import periodogram

import matplotlib.pyplot as plt

#first create some signal
N = 200 #total number of points
dt = 0.01 # sampling period, sampling frequency = 1/dt = 100

#time
ts = np.arange(N)*dt
#signal: sum of two sine waves
ys = np.sin(2*np.pi*10*ts + np.pi/7) + np.sin(2*np.pi*13*ts + np.pi/17)

#plot the signal and label the axes
fig, (ax_sig, ax_fft) = plt.subplots(2, 1)
ax_sig.plot(ts, ys)
ax_sig.set_xlabel('time (s)')
ax_sig.set_ylabel('signal')

ax_fft.set_xlabel('frequency (Hz)')
ax_fft.set_ylabel('normalized FFT or PSD')

#compute the discrete Fourier transform of the signal
fft_frequencies = fft.fftfreq(N, dt)
Y_fft = fft.fft(ys)
ax_fft.plot(fft_frequencies, Y_fft.real/abs(Y_fft).max(), label='real')
ax_fft.plot(fft_frequencies, Y_fft.imag/abs(Y_fft).max(), label='imag')

#power spectral density
#periodogram() also returns the frequences
psd_freq, psd = periodogram(ys, fs=1/dt) #fs = sampling frequency
ax_fft.plot(psd_freq, psd/psd.max(), label='PSD')
ax_fft.legend(loc='best', frameon=False)
fig.tight_layout()
fig.savefig('fouriers.pdf')
plt.show()
```

Listing 18: Výpočet Fourierovy transformace a výkonové spektrální hustoty.




```python
import numpy as np
import matplotlib.pyplot as plt
import scipy.fft as fft

ts = np.linspace(0, 1, 1001)
dt = ts[1] - ts[0]
N = len(ts)
ys = np.zeros_like(ts)
start = 0.125
ys[np.logical_and(ts >= start, ts < start + 0.5)] = 1

plt.close('all')
fig, ax = plt.subplots()
ax.plot(ts, ys, 'k')

# FFT returns all frequencies it can
#ys is real, so we can use rfft
Y = fft.rfft(ys)
frequencies = fft.rfftfreq(N, d=dt)

#how many fourier terms do we want o look at
K = 50

total = 0
cmap = plt.get_cmap('viridis')
norm = plt.Normalize(vmin=0, vmax=K)

for k in range(K):
    #the fourier transform calculates the integral with exp(-i*omega*t)
    #so the signes and real/imag relation to the A and B of the sines and
    #cosines is slightly different
    A = -Y[k].imag/N
    B = Y[k].real/N
    #for oscillating terms we have to double the amplitude for the same
    #reason as when we were calculating A's and B's directly
    if k > 0:
        A *= 2
        B *= 2
    # the 2/N factor comes from the particular normalization used to
    # define fourier transform in SciPy. Various definitions exist.
    sint = A*np.sin(2*np.pi*frequencies[k]*ts)
    cost = B*np.cos(2*np.pi*frequencies[k]*ts)

    ax.plot(ts, sint, color=cmap(norm(k)))
    ax.plot(ts, cost, '--', color=cmap(norm(k)))

    total += sint + cost
```

```
ax.plot(ts, total, '-', color='r')

# However, to get the total sum of the first K fourier terms, we don't
# have to use use the loop explicitly. We can simply calculate the inverse
# fourier transform with frequencies larger than K to zero
#IRfft = inverse real fft
Y_K = np.copy(Y)
Y_K[K:] = 0
# this calculates the inverse fourier transform, i.e., the same sum were
# were building up in the total variable in the for loop
total_irfft = fft.irfft(Y_K, len(ts))
#                           ^^^^^^^
# for INVERSE REAL fft, there is some confusion about what should be the
# correct length of the inverse transform, so it's best to specify the
# expected length of the output explicitly.
ax.plot(ts, total_irfft, ':', color='g', lw=3)
```

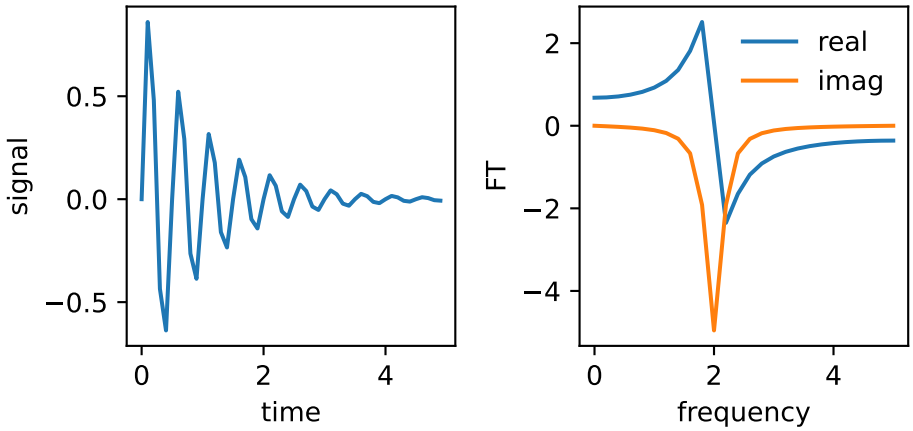
Listing 19: Fourierův koeficient obdélníkové vlny vypočtený pomocí FFT

Jedna obzvláště důležitá Fourierova transformace je transformace exponenciálně tlumené oscilace, tj.

$$s(t) = e^{-t/\tau} \sin(2\pi f t), \tag{13}$$

což dává komplexní lorentzián, který je odezvou lineárního harmonického oscilátoru, viz sekce A. Tlumená oscilace je samozřejmě pohyb tlumeného lineárního harmonického oscilátoru. Příklad:

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.fft as fft

fig, (axt, axf) = plt.subplots(1, 2, figsize=(5, 2.5))

tau = 1
dt = 0.1
t = np.arange(0, 5, dt)
y = np.exp(-t/tau)*np.sin(2*np.pi*2*t)
axt.plot(t, y)

Y = fft.rfft(y)
freq = fft.rfftfreq(len(t), dt)
axf.plot(freq, Y.real, label='real')
axf.plot(freq, Y.imag, label='imag')

axt.set_xlabel('time')
axt.set_ylabel('signal')

axf.set_xlabel('frequency')
axf.set_ylabel('FT')
axf.legend(loc='best', frameon=False)
fig.tight_layout()

fig.savefig('../decaying_exponential.pdf')
```



Intermezzo 7: *Ukládání dat a metadat do binárních souborů numpy.* Doposud jsme při načítání dat z disku narazili pouze na jednoduché, čitelné textové soubory. To je ale velmi omezující způsob ukládání dat: data musí mít formu obdélníkové tabulky, soubory zabírají více místa, než je nutné (např. text `1.234567890` vyžaduje 11 bajtů paměti, ale `float` reprezentující stejné číslo potřebuje pouze 8), a je nepohodlné ukládat metadata.

Mnoho formátů souborů tyto problémy řeší (např. HDF5 je populární). Zde použijeme řešení, které nevyžaduje žádné další externí knihovny – ukládání slovníků do komprimovaných binárních souborů `.npy`, např. uložení

```
raw_data = np.linspace(0, 1, 50)
my_data = {
    'date of measurement': '20241224',
    'was Mercury in retrograde': False,
    'data': raw_data
}
np.save("my_data.npy", my_data)
```

a načtení

```
my_data = np.load("my_data.npy", allow_pickle=True).item()
print(my_data['date of measurement'])
print(my_data['data'])
```

Pickling je pythonovský termín pro ukládání téměř jakéhokoli objektu Pythonu na disk jako binární data. Načítání „picklovaného" objektu může být za určitých okolností bezpečnostním rizikem (tj. vyhněte se čtení „picklovaných" dat, která jste stáhli z podivných míst na internetu), proto to musíme explicitně povolit. Funkce `np.load` a `np.save` pracují s poli, proto `np.load()` vrací pole délky 1, kde náš slovník je jediným prvkem. Metoda `array.item()` vrací `array[0]`, pokud má `array` délku 1, jinak vyvolá výjimku `ValueError`. Jejím účelem je sémanticky naznačit, že očekáváme, že pole bude mít pouze jeden prvek, a pokud ne, něco je špatně.

Cvičení 5.1. Načtěte data z adresáře `timeseries_data` a vykreslete časovou závislost přibližné střední teploty během měření. Načtěte soubory pomocí `d = np.load(filename, allow_pickle=True).item()`. Teplota na začátku a na konci měření je `d['Ti']` a `d['Tf']`.

Nápověda: Pro získání časového znaku (*timestamp*, počet sekund od 00:00 1.1.1970) můžete použít `time.mktime(time.strptime(fn, 'DM_%Y%m%d-%H%M%S.npy'))`, kde `fn` je název souboru.

Cvičení 5.2. Vykreslete časovou řadu a spektrální hustotu pulzu použitého k buzení rezonance v binárních datových souborech `data/timeseries_data` (viz Ukládání a načítání binárních souborů 7) pomocí `scipy.signal.periodogram` a pomocí funkcí (r)FFT z `scipy.fft`. Pulz lze načíst jako `d['pulse']`, počet bodů lze získat jako `d['samples']` (stejné jako `len(d['pulse'])`) a vzorkovací frekvenci jako `d['rate']`. Vykreslete pouze jeden soubor (pulz je pro všechny stejný).

Cvičení 5.3. Vykreslete časovou řadu (jako funkci skutečného času) odezvy rezonátoru (`d['timeseries']`) a frekvenční závislost její Fourierove transformace (reálnou i imaginární složku) pro soubor odpovídající nejnižší teplotě v adresáři `timeseries_data`. Vypočítejte frekvenční odezvu jako poměr Fourierových transformací odezvy rezonátoru a budícího pulzu. Vykreslete pouze frekvence $|f| < 3000$, hledaná rezonance je v rozsahu přibližně 2000 - 2400 Hz.

Cvičení 5.4. Vykreslete velikost (tj. absolutní hodnotu) odezvy rezonátoru r jako 2D teplotní mapu v závislosti na frekvenci a teplotě (tj. každá řádka v 2D grafu by měla být jedno spektrum odpovídající jedné teplotě).

Cvičení 5.5. Jako cvičení 4, ale zprůměrujte všechna spektra, jejichž teplota je bližší než 50 mK (tato technika vyhlazování se nazývá klouzavý průměr nebo sousední průměrování).

Amplituda, výkon, a decibel Při zpracování signálu často mluvíme o útlumu nebo zisku (nebo zesílení, *gain*) systému, např. kabeláže, zesilovačů nebo atenuátorů. Zisk je definován jako poměr výstupního a vstupního signálu a často se měří v decibelech, definovaných jako

$$g = 10 \log_{10} \frac{s_{\text{out}}}{s_{\text{in}}} [\text{dB}] \tag{14}$$

nebo pro výkon, tj. $s = V^2$,

$$g = 20 \log_{10} \frac{V_{\text{out}}}{V_{\text{in}}} \tag{15}$$

Pro „absolutní" veličiny měřené v dB (např. amplituda zvuku je běžná) je měření definováno vzhledem k nějaké dohodnuté referenční hodnotě. Pro zvuk se akustický výkon (tj. druhá mocnina tlaku) měří relativně k 20 µPa ve vzduchu. V elektronice, zejména v radiofrekvenčním (RF) inženýrství, je běžnou jednotkou dBm, kde 0 dBm odpovídá 1 mW výkonu, tj. pro zátěž 50 Ω asi 0.224 V_{RMS}.

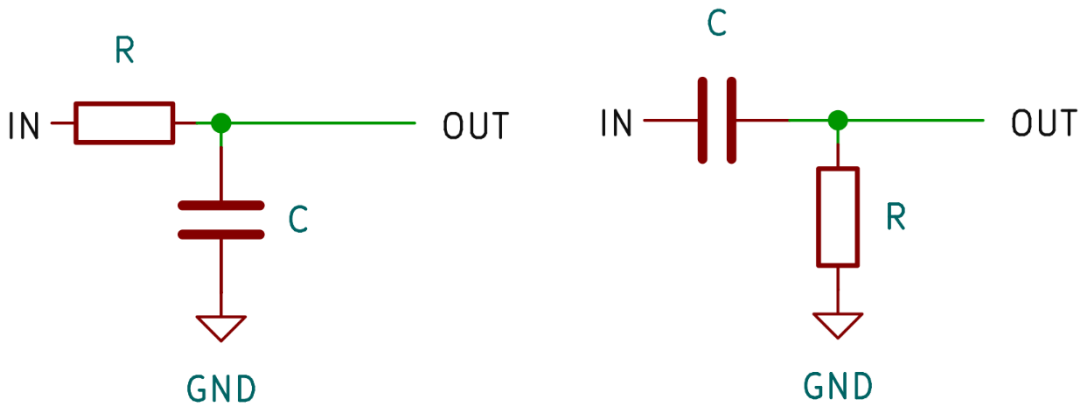
Pro amplitudy odpovídá zdvojnásobení signálu +3 dB a poloviční signál -3 dB a pro výkon je to +6 dB a -6 dB.

5.3 Filtrování

Filtrování je postup, kterým odstraňujeme určité frekvenční rozsahy ze vstupního signálu. Tyto postupy jsou obecné, ale nejjednodušší příklady jsou založeny na analogii s jednoduchými elektronickými RC filtry, viz Obr. 5.3. V závislosti na uspořádání rezistoru a kondenzátoru vytvoříme obvod, který buď zeslabuje nízké, nebo vysoké frekvence.

Použitím impedance kondenzátoru $Z_C = (i\omega C)^{-1}$ pro napětí oscilující na úhlové frekvenci ω dostaneme pro dolní propust

$$\frac{V_{\text{out}}}{V_{\text{in}}} = \frac{1}{1 + i\omega RC}, \tag{16}$$



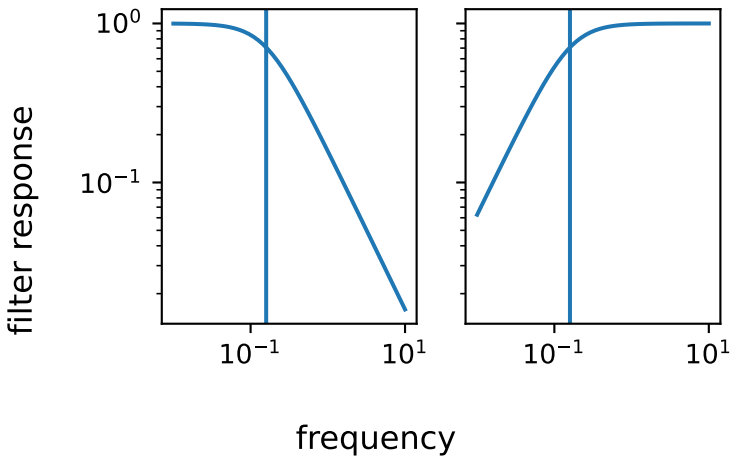
Obrázek 1: Dolní propust (vlevo) a horní propust (vpravo) RC filtr.

a pro horní propust

$$\frac{V_{\text{out}}}{V_{\text{in}}} = \frac{i\omega RC}{1 + i\omega RC}. \tag{17}$$

Výše uvedené výrazy se nazývají přenosové funkce filtru. Veličina $RC = \tau$ se nazývá časová konstanta. Mezní frekvence $f_c = 1/(2\pi\tau)$ je frekvence, při které filtr začíná působit. Odezvu filtrů můžeme vykreslit následujícím kódem

```
tau=1
freqs = np.logspace(-2, 1)
low_pass = 1/(1 + 1j*2*np.pi*freqs*tau)
high_pass = 1j*2*np.pi*freqs*tau/(1 + 1j*2*np.pi*freqs*tau)
fig, (axL, axH) = plt.subplots(1, 2, sharex=True, sharey=True, figsize=(4,2.5))
axL.loglog(freqs, abs(low_pass))
axH.loglog(freqs, abs(high_pass))
axL.axvline(1/(2*np.pi*tau))
axH.axvline(1/(2*np.pi*tau))
```



Obrázek ukazuje absolutní hodnotu přenosové funkce pro dolní a horní propust RC filtrů. Svislá čára je mezní frekvence.

Všimněte si, že přenosové funkce jsou funkcemi frekvence a obvody, které reprezentují, jsou lineární, tj. působí na každou frekvenci nezávisle. Proto, pokud je $s(t)$ náš signál a $\hat{s}(f)$ jeho Fourierova transformace, Fourierova transformace signálu filtrovaného přenosovou funkcí $H(f)$ je jednoduše

$$\hat{s}'(f) = H(f)\hat{s}(f), \tag{18}$$

tj. filtry ve frekvenční doméně jednoduše násobí spektrum. V časové doméně $s'(t) = \mathcal{F}^{-1}[H\hat{s}]$; Fourierova transformace násobení je konvoluce, tj.

$$s'(t) = (h * s)(t) \equiv \int_{-\infty}^{t} s(t')h(t-t')\mathrm{d}t, \tag{19}$$

kde h je inverzní Fourierova transformace H a nazývá se konvoluční jádro. FFT je často nejrychlejší způsob výpočtu konvoluce. Nemusíme ji implementovat sami, protože existuje `scipy.signal.convolve(a , b)`, která počítá konvoluci signálů `a` a `b`.

Pro dolní propust RC filtru lze dokázat, že

$$h(t) = \begin{cases} e^{-t/\tau} & \text{for } t > 0 \\ 0 & \text{for } t < 0 \end{cases} \tag{20}$$

což lze také snadno implementovat pro data která přichází v reálném čase (streaming data), pro která se často nazývá *exponenciální vyhlazování*.

Kromě dolní a horní propusti existují také pásmové propusti a pásmové zádrže (*band pass* a *band stop*). Pásmová propust propouští pouze určité frekvenční pásmo (tj. interval) a pásmová zádrž propouští vše kromě určitého frekvenčního pásma. Pásmovou propust i zádrž si můžete představit jako horní propust následovanou dolní propustí v sérii s mezními frekvencemi danými propustným nebo zadržovaným pásmem.

Jednoduché RC filtry jsou tzv. prvního řádu. Řád filtru udává, jak rychle ořezává signál mimo *propustné pásmo* (tj. interval frekvencí, které filtr propouští). To se často měří v decibelech na oktávu (dB/okt), což udává, o kolik decibelů je signál zeslaben, pokud se jeho frekvence zdvojnásobí, pro dolní propust, nebo zmenší na polovinu pro horní propust, dostatečně daleko od mezní frekvence. Oba RC filtry jsou 6 dB/okt, protože každé zdvojnásobení (nebo půlení) frekvence odstraní 6 dB přenášeného výkonu.

Rychlejší filtry lze konstruovat jak elektronicky, tak digitálně, ale jak jste viděli v příkladech s krokovou funkcí, ostrý přechod ve spektru vede k oscilacím, které jsou obvykle nežádoucí. Filtry, které jsou maximálně ploché v propustném pásmu jsou tzv. Butterworthovy filtry, pojmenované po Stephenu Butterworthovi, které mají amplitudový zisk

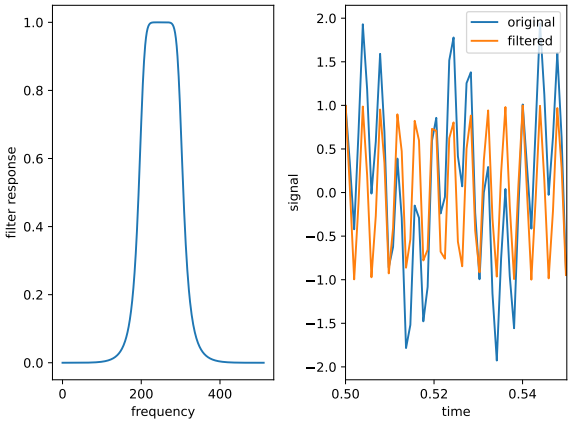
$$G_n(\omega) = \frac{1}{\sqrt{1 + \frac{\omega^{2n}}{\omega_c^{2n}}}},$$

kde ω_c je mezní (úhlová) frekvence. Tyto filtry můžeme zkonstruovat pomocí `scipy.signal.iirfilter` a aplikovat je na náš signál pomocí `scipy.signal.sosfilter`

```
import scipy.signal as sig
N = 4  # filter order, this would be 24 dB/oct
fs = 1024 # samplig frequency
Wn = [200, 300] # corner frequencies of a band-pass filter

# sos = second order sections
# internal representation of the transfer function
# recommended by scipy
sos = sig.iirfilter(N, Wn, fs=fs, btype='bandpass', output='sos')

#create some signal and filter it
ts = np.arange(0, 1, 1/fs)
```

```
ys = np.sin(2*np.pi*50*ts) + np.cos(2*np.pi*250*ts)
ys_filt = sig.sosfilt(sos, ys)

freq, H = sig.sosfreqz(sos, fs=fs)
axH.plot(freq, abs(H))
axsig.plot(ts, ys, label='original')
axsig.plot(ts, ys_filt, label='filtered')
```

:



Cvičení 5.6. Napište funkci, která bude filtrovat vstupní signál pomocí

1. dolně propustního RC filtru s nastavitelnou mezní frekvencí
2. horně propustního RC filtru podobně jako v bodě 1.
3. Butterworthova pásmové propusti zkonstruovaného pomocí `scipy.signal.iirfilter()` a aplikovaného pomocí `scipy.signal.lfilter()`

Pro 1. a 2. vykreslete přenosovou funkci filtru. Otestujte filtrování na přiloženém `noisy_data.npy` a vykreslete signál v časové i frekvenční doméně před a po filtrování. Pro dolní propust odfiltrujte vše nad 100 Hz. Pro horní propust vše pod 1000 Hz a pro pásmovou propust ponechte pouze rozsah (4310, 4330) Hz.

Cvičení 5.7. Demodulujte vyčištěný signál z Cvič. 6 (pomocí Butterworthovy pásmové propusti) na nosné frekvenci 4321 Hz (vynásobte $e^{i\omega t}$ a filtrujte dolní propustí) a vykreslete obálku modulující nosnou vlnu.

Cvičení 5.8. Vypočítejte obálku signálu z Cvič. 7 pomocí Hilbertovy transformace.

5.4 Interpolation and smoothing.

TODO: `scipy.ndimage.gaussian_filter()`, `scipy.signal.savgol_filter()`, `scipy.interpolate.interp1d()`, `scipy.interpolate.UnivariateSpline()`

6 Komunikace s přístroji

Přístroje lze k počítači připojit pomocí široké škály rozhraní, včetně USB, GPIB, RS-232, Ethernetu nebo emulovaného sériového portu přes USB. Každé z těchto rozhraní vyžaduje specifickou podporu operačního systému, ovladače nebo knihovny. To naštěstí do značné míry vyřešila architektura VISA (Virtual Instrument Software Architecture), která abstrahuje a sjednocuje většinu rozhraní do tzv. VISA „sezení" (sessions).

Většina komunikace přes VISA sezení je textová. Přístroji se pošle textový příkaz, přístroj provede nějakou akci a volitelně odpoví textovou odpovědí. Pro použití VISA budeme používat python modul `pyvisa`. Ten je však pouze python rozhraním k systémové knihovně VISA. Existuje několik implementací knihovny VISA (někdy nazývaných VISA backend) od různých výrobců, např. od společností National Instruments (NI-VISA, nejběžnější) nebo Keysight. My budeme používat open-source, plně pythonový VISA backend `pyvisa-py`. Pro naše účely budeme také potřebovat `pyserial`. Pokud chcete používat přístroje s rozhraním USB, Ethernet nebo GPIB, je pravděpodobně nejlepší cestou jedna z volně stažitelných proprietárních implementací.

V tomto kurzu budeme místo skutečných přístrojů používat Raspberry Pi Pico. Pico je naprogramováno tak, aby reagovalo na několik příkazů, které zhruba odpovídají syntaxi SCPI (viz Dodatek B). Jednoduchý VISA program, který komunikuje s přístrojem, může vypadat takto:

```
import pyvisa as vi

rm = vi.ResourceManager()

print("Available resources:")
resources = rm.list_resources()
for k, res in enumerate(resources):
    print(f'VISA address {k}: {res}')

#assuming that our pico is on the last address (usually the case)
pico = rm.open_resource(resources[-1],
                        read_termination='\n',
                        write_termination='\n')
print("Identification of the last device:")
pico.write('*IDN?')
resp = pico.read()
print(resp)

# performs the same in one step
# query = write + read
print("IDN via query:")
print(pico.query("*IDN?"))

pico.close()
rm.close()
```

Na prvním řádku importujeme modul `pyvisa`. Vytváření nových sezení se provádí pomocí Správce zdrojů (Resource Manager) `rm`. V názvosloví VISA se přístroje nazývají „zdroje" (resources) a jsou identifikovány adresou zdroje, která obvykle udává typ použitého rozhraní, např. adresy začínající `ASRL` jsou sériové přístroje, `GPIB` a `USB` atd.

Správce zdrojů poskytuje metodu `list_resources()`, která vrací seznam všech dostupných zdrojů. Pro USB a GPIB to jsou obvykle přístroje skutečně připojené k počítači. Pro sériové rozhraní mohou být uvedeny všechny dostupné porty bez ohledu na to, zda je k nim něco připojeno.

Pro spuštění VISA sezení použijeme metodu správce zdrojů `open_resource()`. Zde navíc specifikujeme ukončovací znaky pro čtení a zápis, což jsou znaky, které indikují, že přenos je ukončen (představte si vojenské filmy, kde říkají „přepínám" do vysílačky). Běžné jsou nový řádek `'\n'`, návrat vozíku `'\r'` a jejich kombinace `'\r\n'`. Naše Pico očekává, že příkazy budou ukončeny novým řádkem `'\n'` a své odpovědi také ukončuje novým řádkem. U jiných přístrojů je to něco, co musíte najít v manuálu, ale `'\n'` je nejběžnější.

Otevřené VISA sezení je reprezentováno objektem `pico` vráceným metodou `open_resource()`, jehož nejdůležitější metody jsou `write()`, `read()` a `query()`, což je jednoduše zápis okamžitě následovaný čtením.

Většina laboratorních přístrojů komunikuje pomocí textových řetězců, které obvykle dodržují syntaxi specifikovanou Standardními příkazy pro programovatelné přístroje (SCPI), která se snaží definovat společnou syntaxi pro přístroje podobného typu. Příkazy SCPI mají typicky formát

```
:COMMand:SUBCommand:SUBCommand ARG1 ARG2 ...
```

kde velkými písmeny psané podřetězce lze použít jako zkratku. Například na většině digitálních multimetrů (např. ctihodná řada Keithley 2000) příkaz

```
:MEAS:VOLT:DC?
```

změří stejnosměrné napětí. Otazník na konci značí, že příkaz vrací hodnotu. Přístroje, které podporují SCPI, navíc podporují několik základních příkazů, jako je `*IDN?`, který vrací identifikační řetězec, `*STB?`, který vrací jeden bajt s různými stavovými bity přístroje, nebo `*RST`, který přístroj resetuje. Na konci by mělo být jak VISA sezení, tak správce zdrojů řádně uzavřeno pomocí metody `close()`.

Ve většině případů není pro úspěšné otevření sezení metodou `rm.open_resource()` potřeba nic jiného než adresa zdroje. Nicméně, zejména u některých starších přístrojů připojených přes RS-232, jsou často vyžadovány další informace. Za předpokladu, že zařízení bylo otevřeno jako `dev = rm.open_resource(address)`, je možné, že před zahájením komunikace musíme nastavit následující:

přenosová rychlost (baud rate) Počet změn komunikačního signálu za sekundu (tj. změny z nízkého na vysoké napětí). Lze nastavit pomocí `dev.baud_rate` na celé číslo.

datové bity (data bits) Počet skutečných datových bitů v jednom „paketu" dat vyměňovaných mezi přístrojem a počítačem (obvykle 8). Lze nastavit pomocí `dev.data_bits` na celé číslo.

stop bity (stop bits) Počet bitů označujících začátek a konec znaku přenášeného po sériovém kabelu (obvykle 1). Lze nastavit pomocí `dev.stop_bits` na jednu z hodnot `pyvisa.constants.StopBits.SB`, kde `SB` je `one`, `one_and_a_half` nebo `two`.

parita (parity) Jednoduchá kontrola poškozeného přenosu. Některé přístroje posílají bit indikující, zda součet bitů v posledním znaku (tj. jeho ASCII kód) byl lichý nebo sudý. Lze nastavit pomocí `dev.parity` na jednu z hodnot `pyvisa.constants.Parity.P`, kde `P` je `none` (nejběžnější, žádná kontrola parity), `even` nebo `odd`.

Například, pokud bychom měli přístroj na adrese `COM16` a jeho manuál říká, že očekává přenosovou rychlost 19200, 8 datových bitů, jeden stop bit a žádnou paritu, mohli bychom komunikaci otevřít takto:

```
import pyvisa as vi

rm = vi.ResourceManager()
dev = rm.open_resource('COM16')
dev.baud_rate = 19200
dev.data_bits = 8
dev.stop_bits = vi.constants.StopBits.one
dev.parity = vi.constants.Parity.none

dev.query('*IDN?')
...
```

Cvičení 6.1. Vytvořte program, který automaticky najde Pico, pokud je připojeno k počítači. Pošlete dotaz `*IDN?` každému dostupnému VISA zdroji a najděte ten, který odpoví `'PICO'`. Zdroje (sériové porty), které nejsou připojeny k žádnému přístroji, vyvolají timeout výjimku (timeout), která musí být ošetřena bez pádu programu.

Cvičení 6.2. Otevřete komunikaci s Pico a nechte bílé a modré LED diody blikat ve stylu „policejního auta" (tj. dvě rychlá bliknutí bílé následovaná dvěma rychlými bliknutími modré) v opakující se smyčce. Použijte příkaz `:LED n x`, kde $n = 0 \ldots 4$ je číslo LED a x je buď 0 nebo 1, což LED vypne nebo zapne. Ujistěte se, že když je váš program ukončen (buď pomocí Ctrl-C nebo tlačítkem pro přerušení v Spyderu), všechny LED diody jsou zhasnuté.

Cvičení 6.3. Napište program, který bude indikovat aktuální teplotu (získanou pomocí příkazu `:READ:T?`) pomocí vestavěných LED diod. Namapujte rozsah 20-27°C na 0 až 5 rozsvícených LED. Pokud teplota překročí 28°C, vyvolejte `RuntimeError`, což ukončí program. Ujistěte se, že žádná z LED diod nezůstane svítit po ukončení programu.

Cvičení 6.4. Napište „digitální vodováhu", tj. indikujte aktuální náklon podél osy y (podélně s PICO) pomocí LED diod (můžete se rozhodnout, jaký způsob použití LED považujete za nejlepší). Všimněte si, že akcelerometr není na desce připájen dokonale vodorovně. Předpokládejte, že při spuštění programu je Pico ve vodorovné poloze a použijte vektor zrychlení změřený na začátku jako referenční hodnotu.

Cvičení 6.5. Napište objektové rozhraní pro Pico, které by mělo být inicializováno pouze pomocí instance správce zdrojů a mělo by si samo najít správnou adresu, tj.:

```
import pyvisa as vi

class Pico:
    def __init__(self, rm):
        ...
    def led(self, led_id, onoff):
        #should raise ValueError if led_id > 4
        ...
    def getT(self):
        ...
    def getP(self):
        ...
    def getACC(self):
        ...
    def getGYR(self):
        ...
    def close(self):
        "Turn off all LEDs and close the session."
        ...

rm = vi.ResourceManager()
pico = Pico(rm)
pico.led(2, 1) #turns the middle LED ON

```

6.1 Context Management Protocol

Zejména u přístrojů, které ovládají skutečný laboratorní hardware, je velmi důležité správné vypnutí a úklid, a to i v případě softwarové chyby. Představte si pec, která se stále zahřívá, nebo motor, který se nekontrolovatelně točí kvůli překlepu ve vašem programu. Ve cvičeních 27 – 29 jsme viděli, že řízené vypnutí lze řešit pomocí klauzule `finally` za blokem `try`. To je neoptimální, protože si musíme pamatovat, že v každém programu, ve kterém daný přístroj používáme, musíme napsat blok `finally`, který se často příliš nemění.

Python na to má řešení, které jsme již viděli, nazývané *protokol pro správu kontextu* (context management protocol), který používá příkaz `with`. Vzpomeňte si na použití souborů:

```
    with open("hello.txt", 'w') as file:
        file.write('Hello Context Management.')
```

kde příkaz `with` zajišťuje, že po dokončení práce se souborem je řádně `close()`d (uzavřen).

Abychom mohli správu kontextu používat s našimi objekty, stačí definovat dvě speciální metody, `__enter__()` a `__exit__()`, které se spouštějí na začátku a na konci příkazu `with`. Metoda enter by měla vrátit objekt, který chceme použít (tj. `file` v příkladu výše), často jednoduše vrací `self`. Metoda exit by měla provést veškerý potřebný úklid. Kromě `self` přijímá tři další argumenty, které obsahují informace o tom, zda došlo k výjimce a jakého typu. Většinou můžeme výjimky ignorovat a jednoduše je nechat propagovat dál. Pokud z metody `__exit__` vrátíme `False`, výjimka je potlačena.

Vezměme si jednoduchý příklad. Řekněme, že chceme mít objekt podobný souboru, který lze použít s `with` jako běžný soubor, ale metoda `write()` také podporuje pole NumPy. Mohli bychom to udělat následovně:

```
import numpy as np

class MyNumpyFile:
    def __init__(self, filename, mode='w'):
        print("Opening file.")
        self.file = open(filename, mode)

    def write(self, data):
        np.savetxt(self.file, data)

    def __enter__(self):
        print("Entering with statement")
        return self
    def __exit__(self, *exc):
        print("Closing.")
        self.file.close()

print("I'm about to open the file.")
with MyNumpyFile("my_array.txt", 'w') as file:
    file.write(np.arange(5))
print("The end.")
```

což vypíše:

```
I'm about to open the file.
Opening file.
Entering with statement
Closing.
The end
```

Všimněte si, že metoda `__enter__()` jednoduše vrací `self`. Je to proto, že správce kontextu a objekt reprezentující soubor jsou stejné. V metodě `__exit__()` jednoduše shromáždíme všechny informace o jakékoli výjimce, která mohla nastat, do seznamu argumentů `*exc` a ignorujeme je, provádíme pouze nezbytný úklid. Nevrácení ničeho je ekvivalentní vrácení `None`, což není `False`, takže pokud dojde k jakékoli výjimce, bude jednoduše pokračovat ve své cestě k nejbližší klauzuli `except`.

Cvičení 6.6. Rozšiřte třídu Pico tak, aby podporovala protokol pro správu kontextu.

Pro úplnost, příklad s `open()` je v podstatě ekvivalentní následujícímu kódu:

```
    manager = open("hello.txt", 'w')
    file = manager.__enter__()
    try:
        file.write("Hello Context Management.")
```

```
    except:
        if not manager.__exit__(exception_info)
            raise
    else:
        manager.__exit__(None, None, None)
```

6.2 Často se vyskytující problémy

Komunikace s externím hardwarem může selhat z několika důvodů. Za předpokladu, že samotný hardware a propojovací kabely fungují, jsou některé běžné příčiny problémů:

výjimka při otevírání VISA sezení V závislosti na operačním systému a použitém VISA backendu může `rm.open_resource()` vyvolat výjimku s tvrzením, že přístup ke zdroji byl odepřen (*access denied*) nebo že zařízení je zaneprázdněno (*resource busy*). To je nejčastěji způsobeno dříve otevřeným VISA sezením, které nebylo uzavřeno. V jednom okamžiku je povoleno pouze jedno VISA sezení s daným přístrojem.

Toto je běžný problém v IDE, jako je Spyder, které spouštějí soubory v interaktivní konzoli, kde proměnné zůstávají dostupné i po neočekávaném ukončení programu. Buď zajistěte, aby se sezení správně uzavírala i v případě pádu programu (tj. pomocí `finally` nebo správy kontextu), nebo zavřete konzoli, ve které byl program spuštěn.

zdroj se otevře, ale komunikace vyprší (timeout) Pokus o čtení vyprší, pokud zařízení neodpoví očekávaným způsobem, nejčastěji kvůli chybě v příkazu. Pokud je odeslaný příkaz určitě správný, je timeout výjimka obvykle příznakem nesprávně nastavené konfigurace VISA sezení, např. ukončovacích znaků pro čtení/zápis nebo některých z výše uvedených možností konfigurace sériového portu.

zdroj se otevře, nedojde k timeoutu, ale odpověď je špatná Zvažte následující kód:

```
    import pyvisa as vi
    rm = vi.ResourceManager()
    pico = rm.open_resource(rm.list_resources()[-1])
    pico.read_termination = '\n'
    pico.write_termination = '\n'
    pico.write(':READ:T?')
    print(pico.query('*IDN?'))
```

který vypíše naměřenou teplotu místo očekávaného identifikačního řetězce. Je to proto, že odpověď na `:READ:T?` nebyla nikdy přečtena, takže zůstala ve vstupní vyrovnávací paměti (buffer) až do prvního volání `read`, které přišlo v rámci `query()`. Pokud to backend a typ zdroje podporují, můžete zavolat `pico.clear()` pro vyprázdnění bufferů, nebo můžete zavolat `pico.read(timeout=0)` a zahodit timeout výjimku, pokud je buffer náhodou prázdný.

7 Paralelní běh programů

Python rozlišuje dva typy paralelismu: vícevláknový (multithreading) a víceprocesový (multiprocessing) běh. Vlákna poskytují pouze iluzi skutečného paralelismu: pro jeden proces Pythonu (tj. jedno spuštění `python file.py`) běží v jednom okamžiku pouze jedno vlákno, ale běh se přepíná mezi více vlákny tak, aby se vytvořila iluze paralelismu. I když váš skript běží na vícejádrovém (nebo víceprocesorovém) počítači, bude použito pouze jedno jádro[11]. Víceprocesový paralelismus na druhé straně může spouštět více procesů Pythonu skutečně paralelně, současně na více jádrech CPU, pokud jsou k dispozici.

Ačkoli se vlákna mohou zdát zbytečná, často se snadněji používají a zejména pro aplikace, které čekají na hardwarové I/O, síť nebo interakci s uživatelem, jsou často lepší volbou. Víceprocesové aplikace mohou být rychlejší pro dostatečně velké problémy, avšak pro krátce běžící programy jsou často *pomalejší*, protože správa více procesů vyžaduje *režii*, která může být srovnatelná s řešením samotného problému.

7.1 Vícevláknový paralelismus (Multithreading)

`Thread` objekty z modulu `threading` reprezentují vlákna. Vlákna provádějí danou *cílovou* (target) funkci, když jsou *spuštěna*, a po dokončení musí být *připojena* (joined) zpět k rodičovskému vláknu. Příklad použití:

```
import threading as th

def func(thread_name):
    print(f"I'm inside {thread_name}")

threads = []
# create and start 5 threads and save them to a list
for k in range(5):
    thread = th.Thread(target=func, args=(f"thread {k}",))
    thread.start()
    threads.append(thread)

# wait for all threads to finish
for thread in threads:
    thread.join()
```

Často je však objektově orientovaný přístup pohodlnější než target funkce. Můžeme definovat vlastní objekty vláken jednoduchým děděním z `Thread` a definováním metody `run()`. Funkčně ekvivalentní příklad k výše uvedenému:

```
import threading as th

class MyThread(th.Thread):
    def __init__(self, thread_name):
        # It is necessary to initialize the parent class as well
        super().__init__()
        self.thread_name = thread_name
    def run(self):
        print(f"I'm inside {self.thread_name}")

threads = []
for k in range(5):
    thread = MyThread(f"thread {k}")
    thread.start()
    threads.append(thread)

```

[11] Od verze Pythonu 3.13 lze *globální zámek interpretu, global interpreter lock* (GIL), který způsobuje, že vlákna běží bez skutečného paralelismu, volitelně vypnout, ačkoli GIL zůstane výchozím chováním v dohledné budoucnosti. Více podrobností viz zde.

```
for thread in threads:
    thread.join()
```

Vlákna musí být schopna vzájemně komunikovat a předvídatelně sdílet zdroje. Zvažte VISA funkci `query()`, což je jednoduše `write()` okamžitě následované `read()`. Pokud existují dvě vlákna komunikující se stejným zařízením Pico, jedno provádí `query(':READ:P?')` a druhé `query(':READ:T?')`, existuje šance, že skutečné pořadí provedených čtení a zápisů bude:

```
    # vlákno 1
    write(':READ:P?')
    # běží vlákno 2
    # běží vlákno 2
    read()
```

```
    # vlákno 2
    # běží vlákno 1
    write(':READ:T?')
    read()
    # běží vlákno 1
```

a vlákno, které žádalo o tlak, dostane teplotu a naopak. Tato třída chyb se nazývá *souběh* (race conditions) – tj. dvě vlákna se „předhánějí“ v soutěži o zdroj a vítěz je náhodný. Abychom tomu zabránili, musíme Pythonu říci, že bychom neměli být na chvíli přerušeni, dokud neskončíme se zápisem a čtením. Toho se dosahuje pomocí *zámků* (nebo mutexů, z MUTual EXclusion), které jsou k dispozici v modulu `threading` jako třída `Lock` a používají se následovně

```
    from threading import Lock
    lock = Lock()
```

```
    # vlákno 1
    lock.acquire()
    # běží vlákno 2
    write(':READ:P?')
    read()
    lock.release()
    # běží vlákno 2
    # běží vlákno 2
    # běží vlákno 2
```

```
    # vlákno 2
    # běží vlákno 1
    lock.acquire() # blokuje, dokud není zámek uvoln
    ěn
    # běží vlákno 1
    # běží vlákno 1
    # běží vlákno 1, lock.acquire() se vrátí
    write(':READ:T?')
    read()
    lock.release()
```

Všimněte si, že používáme objekt vytvořený třídou `Lock`, nikoli třídu samotnou. To platí pro všechny synchronizační a komunikační mechanismy – `Locks`, `Events` a `Queues` a další. Zámky podporují protokol správy kontextu, takže je obvykle používáme v příkazu `with` místo přímého volání `acquire()` a `release()`. Úplnější příklad,

```
import pyvisa as vi
from threading import Thread, Lock

rm = vi.ResourceManager()
# Assuming that our pico is on the last address (usually the case)
pico = rm.open_resource(rm.list_resources()[-1],
                        read_termination='\n',
                        write_termination='\n')

def readP(lock):
    with lock: # lock is acquired
        # this will run uninterrupted
        P = pico.query(':READ:P?')
        print(P)
    #lock is released

def readT(lock):
    with lock:
        T = pico.query(':READ:T?')
        print(T)

lock = Lock()
t1 = Thread(target=readP, args=(lock,))
t2 = Thread(target=readT, args=(lock,))

t1.start()
t2.start()
t1.join()
```

```
t2.join()
```

Všimněte si, že pokud bychom použili dva zámky k uzamčení dvou samostatných zdrojů, mohli bychom se dostat do situace, kdy dva zámky na sebe navzájem čekají věčně, jako je tato

```
    # vlákno 1
    # snaží se získat lock1 a lock2
    # v tomto pořadí
    lock1.acquire()
    # běží vlákno 2
    # běží vlákno 2
    lock2.acquire() # blokuje navždy
```

```
    # vlákno 2
    # snaží se získat lock2 a lock1
    # v tomto pořadí
    # běží vlákno 1
    lock2.acquire()
    lock1.acquire() # blokuje navždy
    # běží vlákno 1
```

což se nazývá *zablokování* (deadlock). Buďte zvláště opatrní, když používáte více než jeden zámek.

Nejjednodušší metodou komunikace mezi vlákny jsou globální proměnné. To se však může rychle stát nepřehledným a matoucím, proto je obvykle lepší používat typy určené pro komunikaci mezi vlákny. Nejjednodušší je `threading.Event`, což je booleovský příznak, který může být nastaven jedním vláknem a na který může reagovat jiné, např.:

```
import pyvisa as vi
from threading import Thread, Lock, Event
import time()

rm = vi.ResourceManager()
pico = rm.open_resource(rm.list_resources()[-1],
                        read_termination='\n',
                        write_termination='\n')

def keep_reading_P(lock, end_event):
    #keep running until the end_event is set
    while not end_event.is_set():
        with lock: # lock is acquired
            # this will run uninterrupted
            P = pico.query(':READ:P?')
            print(P)
        #lock is released
        time.sleep(1)

lock = Lock()
end = Event()
thr = Thread(target=keep_reading_P, args=(lock,end))
t0 = time.time()
thr.start()
# measure for five second and then signal the thread to end
while True:
    if time.time() - t0 > 5:
        end.set()
        break
    time.sleep(0.1)
thr.join()
```

Pro posílání dat mezi vlákny jsou užitečné `Queue`s z modulu `queue`. Hodnotu můžeme vložit do fronty v jednom vlákně pomocí metody `put()` a vyjmout ji jinde pomocí metody `get()`, která blokuje, pokud je fronta prázdná. Můžeme také zkontrolovat, zda je fronta prázdná, pomocí metody `empty()`. Minimální příklad:

```
from threading import Thread
from queue import Queue

# function that the thread will run
def thread_func(q):
    while True:
        # waits until a message becomes available
        msg = q.get()
        print(f"Received message {msg}")
        if msg == 'quit':
```

```
            break

# create the queue and start the thread
q = Queue()
thread = Thread(target=thread_func, args=(q,))
thread.start()

messages = ['hello', 'world', 'quit']
for msg in messages:
    # send the message
    q.put(msg)

thread.join()
```

Úplnější příklad, který čte data z Pico v jednom vlákně a vykresluje je v jiném:

```
import pyvisa as vi
import matplotlib.pyplot as plt
from threading import Thread, Event
from queue import Queue
import time

# A thread that plots data received through a queue
class Plotter(Thread):
    def __init__(self, queue):
        # Initialize the parent class
        super().__init__()

        # Create a figure and axis for plotting and
        # data storage
        self.fig, self.ax = plt.subplots()
        self.xdata = []
        self.ydata = []
        self.line, = self.ax.plot(self.xdata, self.ydata, 'o')

        # queue for receiving data and an event to signal the end
        self.queue = queue
        self.end = Event()

    # replot the data and update axes limits
    def update_plot(self):
        if len(self.xdata) > 0:
            self.line.set_xdata(self.xdata)
            self.line.set_ydata(self.ydata)
            xmin, xmax = min(self.xdata), max(self.xdata)
            ymin, ymax = min(self.ydata), max(self.ydata)
            xmid = 0.5*(xmin + xmax)
            ymid = 0.5*(ymin + ymax)
            dx = xmax - xmin
            dy = ymax - ymin
            self.ax.set_xlim(xmid - 0.55*dx, xmid + 0.55*dx)
            self.ax.set_ylim(ymid - 0.55*dy, ymid + 0.55*dy)
            # plt.draw()

    def pull_data(self):
        # keep reading the data from the queue as long
        # as anything is available
        while not self.queue.empty():
            # if get() is called on an empty queue, it blocks
            # until something becomes available (or a timeout occurs)
            data = self.queue.get()
            print("Received ", data)
            #we can send anything through the queue, for example
            #either a data point or a command to quit
            match data:
                case (x, y):
```

```
                    self.xdata.append(x)
                    self.ydata.append(y)
                case 'quit':
                    print("Quitting")
                    self.end.set()

    def run(self):
        while not self.end.is_set():
            self.pull_data()
            self.update_plot()
            time.sleep(0.5)


rm = vi.ResourceManager()
# change the address to match your system
pico = rm.open_resource('ASRL/dev/ttyACM1::INSTR',
                        read_termination='\n',
                        write_termination='\n')

data_queue = Queue()
plotter = Plotter(data_queue)
plotter.start()
try:
    t0 = time.time()
    while True:
        t = time.time() - t0
        P = float(pico.query(':READ:P?'))
        print("Sending ", t, P)
        data_queue.put((t, P))
        # the following line updates all open matplotlib plots
        # it MUST be run from the main thread
        plt.pause(0.01)
finally:
    data_queue.put('quit')
    plotter.join()
```

Cvičení 7.1. Napište program, který bude blikat všemi 5 LED diodami separátně v intervalech 0.1, 0.2, 0.5, 1 a 2 s.

Cvičení 7.2. Napište program, který bude zobrazovat aktualizovaný graf tlaku. Během běhu programu by měl být schopen přijímat textové příkazy a měl by podporovat: clear, který vymaže aktuální graf, a quit, který program čistě ukončí.

7.2 víceprocesový paralelismus (Multiprocessing)

Víceprocesový paralelismus může využívat více jader CPU, avšak existují omezení na to, jaké druhy proměnných lze sdílet mezi procesy. Nové procesy můžeme vytvářet pomocí třídy `Process` z modulu `multiprocessing` velmi podobným způsobem jako vlákna. Modul `multiprocessing` také poskytuje synchronizační a komunikační prostředky podobné `threading`, tj. `Event`, `Queue` atd. Musíte však používat třídy z modulu multiprocessing pro komunikaci mezi procesy. Minimální příklad, kde hlavní proces vytváří sadu procesů a posílá jim všem zprávy prostřednictvím sdílené `Queue`:

```
from multiprocessing import Process, Event, Queue
from queue import Empty
import time

# The same as for threads:
# Inherit from Process, initialize the parent class in __init__
# and write your own run() method
```

```
class MyProcess(Process):
    def __init__(self, procname, end, data):
        super().__init__()
        self.procname = procname
        self.end = end
        self.data = data
    def run(self):
        while True:
            try:
                msg = self.data.get(timeout=1)
                print(f"{self.procname} received: {msg}")
            except Empty:
                print(f"{self.procname}: Nothing in queue")
                time.sleep(0.1)

            if self.end.is_set() and self.data.empty():
                break

# Using processes is very similar to using threads
# However, we must use multiprocessing.Event and multiprocessing.Queue
# instead of the threading versions
# Also, note that, generally "if __name__ == '__main__':" guard is required
# to avoid recursive spawning of subprocesses
if __name__ == '__main__':
    data_queue = Queue()
    end_event = Event()
    process_pool = []
    for k in range(5):
        p = MyProcess(f'proc{k}', end_event, data_queue)
        p.start()
        process_pool.append(p)

    for k in range(10):
        data_queue.put(f"message {k}")
        time.sleep(0.2)

    end_event.set()
    for p in process_pool:
        p.join()
```

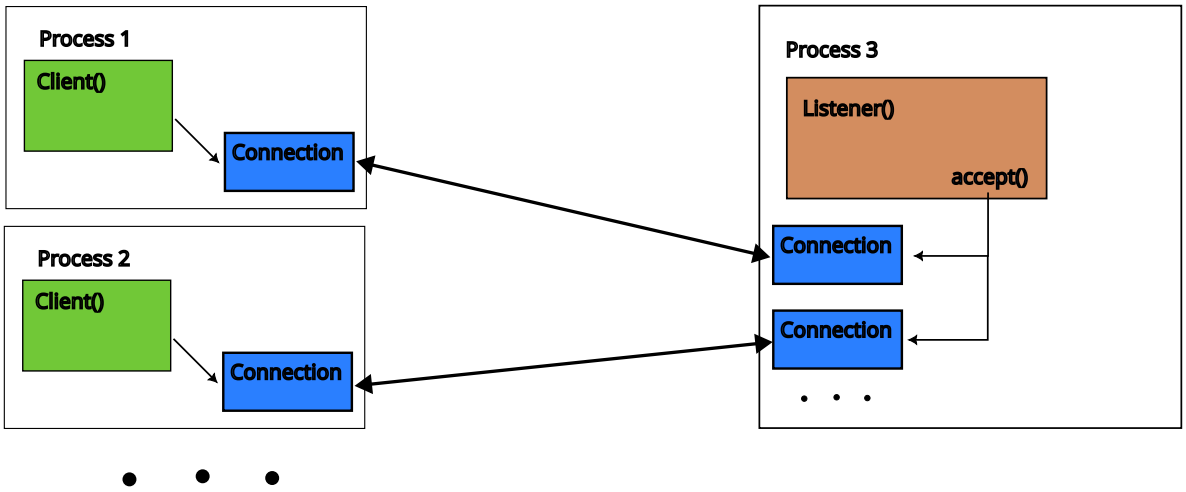
Výše uvedený kód je příkladem *fondu procesů* (process pool), což je často nejjednodušší způsob, jak urychlit problémy, které zahrnují více nezávislých výpočtů. Multiprocessing již poskytuje obecný fond process pool pro tento úkol:

```
from multiprocessing import Pool

def function(x):
    return some_calculation(x)

z = [... data ...]

with Pool(6) as pool:
    result = pool.map(function, z)
```

který aplikuje funkci `function(x)` na každý prvek sekvence (např. seznam, pole, ...) v 6 paralelních procesech a sbírá výsledky.

Úplnější příklad, který počítá Fibonacciho posloupnost pomalaým rekurzivním algoritmem:

```
import numpy as np
import matplotlib.pyplot as plt
from multiprocessing import Pool
import time

# A slow recursive function to compute Fibonacci numbers
def fib(x):
```



Obrázek 2: Vztahy mezi posluchači, klienty a spojeními

```
    if x < 2:
        return 1
    return fib(x-1) + fib(x-2)

# Function to compute Fibonacci numbers in parallel using a pool of processes
def calc_fibs(z, nproc):
    with Pool(nproc) as pool:
        fibs = pool.map(fib, z)
    return fibs

# Time the computation with different numbers of processes
def timeit(z, nproc):
    t0 = time.time()
    fibs = calc_fibs(z, nproc)
    t1 = time.time()
    print(f"Calculation with {nproc} processes took {t1 - t0:.3f} s")

z = np.arange(35)
timeit(z, 1)
timeit(z, 2)
timeit(z, 4)
timeit(z, 8)
timeit(z, 16)
```

všimněte si, že pro přílišné zvětšení počtu procesů může výpočet ve skutečnosti začít *spomalovat*.

7.3 Meziprocesová komunikace

Doposud jsme vytvářeli nové procesy z python skriptu, který jsme spustili. Avšak jakékoli dva procesy Pythonu, tj. samostatné běhy jakýchkoli programů v Pythonu, mohou spolu komunikovat. Existuje několik způsobů, jak toho dosáhnout, například přímé sdílení paměti (pomocí modulu `multiprocessing.shared_memory`, viz dokumentaci pro příklad použití s poli numpy) nebo *sokety*. Mechanismus soketů je nějakým způsobem poskytován všemi operačními systémy jako obecný způsob komunikace mezi procesy, lokálně nebo přes síť, viz Obr. 7.3. Obecně navážeme spojení s adresou (může být *localhost*, pokud nekomunikujeme přes síť) a číslem portu. Jeden proces funguje jako *posluchač* (tj. server), který *přijímá* spojení od *klientů* a oba si pak mohou navzájem posílat a přijímat zprávy.

Níže uvedený příklad ukazuje jednoduchý echo server – server, který jednoduše posílá zpět klientovi to, co obdrží. Bude přijímat pouze spojení z lokálního počítače na portu 6000. Nejprve je ob-

jekt posluchače vytvořen pomocí příkazu `with`, který je poté použit ke spuštění samostatného vlákna, které přijímá spojení a odpovídá klientům. Hlavní vlákno bude jednoduše čekat na signál k ukončení. Pokud máme skončit, zavřeme posluchače (musíme také odblokovat `accept()`, které by čekalo věčně) a skončíme. Všimněte si, že hesla jsou volitelná, pokud nezadáte `authkey` pro `Listener`, nemusíte ho zadávat ani pro `Client`.

Klient je v porovnání mnohem jednodušší. Jednoduše se připojte k posluchači se známou adresou, portem a heslem a můžete začít posílat a přijímat data. Metody `send()` a `recv()` provádějí serializaci a deserializaci (pickling a unpickling, vzpomeňte si na pickling ze souborů `.npy`), takže lze posílat téměř všechny objekty Pythonu.

Pro otestování níže uvedeného příkladu spusťte `python server.py` v jedné konzoli a `python client.py` v druhé.

`server.py`:

```
from multiprocessing.connection import Listener, Client
from threading import Thread
import time

# we will accept the connections in a separate thread
def handle_connections(listener):
    while True:
        # listener.accept() blocks until someone
        # tries to connect
        try:
            # accept the connection, echo back what we receive
            # and then close the connection (automatically, using 'with')
            with listener.accept() as conn:
                msg = conn.recv()
                print(f"Received {msg}")
                conn.send(f'Echo {msg}')
        except OSError:
             # OSError is raised when we try to accept
             # with a closed listener
             print("Stopping listening")
             break

# the address and port
# 'localhost' is local computer, use empty string ''
# if you want to access it over the netowrk
# port number 0 means automatic assignment by the OS
address = ('localhost', 6000)
password = b'password'
with Listener(address, authkey=password) as listener:
    print('Address :', listener.address)

    t = Thread(target=handle_connections, args=(listener,))
    t.start()
    # now the main thread will just idly sleep, waiting for
    # keyboard interrupt
    try:
        while True:
            time.sleep(1)
    except KeyboardInterrupt:
        print("Quitting...")
    # this will signal the listener to close, however, it
    # will remain open as long as .accept() is blocking
    listener.close()
    # so create a dummy connection to unblock accept()
    # and let the listener close
    try:
        with Client(address, authkey=password) as c:
            c.send('')
            c.recv()
    except ConnectionRefusedError:
```

```
        # someone connected before our dummy connection
        # and the listener shut down, don't do anything
        pass
    t.join()
```

client.py

```
import numpy as np
from multiprocessing.connection import Client

address = ('localhost', 6000)
passowrd = b'password'
with Client(address, authkey=passowrd) as conn:
    # send and recv pickle and unpickle, respectively
    # the objects we try to send. So we can send essentially
    # arbitrary data
    conn.send({'key1': 'hello', 'key2': np.arange(5)})
    resp = conn.recv()
    print("Response :", resp)
```

Pro povolení spojení ze sítě jednoduše zadejte adresu posluchače jako `''` (prázdný řetězec) a připojte se s klientem k IP adrese počítače, na kterém běží proces posluchače. Pozor na to, že pickling a unpickling v `send()` a `recv()` může vést k potenciálním bezpečnostním problémům, takže je nejlepší použít heslo, pokud přijímáte spojení přes síť.

Cvičení 7.3. Server běží na počítači s IP adresou `<IP>`, na portu `<PORT>` s heslem `<PASSWORD>`. Tento server ovládá Pico přímým odesláním jakéhokoli řetězce, který obdrží, do Pico a odesláním odpovědi z Pico zpět. Připojte se k tomuto serveru s klientem, odešlete příkaz `*IDN?` pro dotaz na identifikaci přístroje a blikněte LED diodou, pokud je odpověď správná.

`instrument_server.py`: Kód přijímá spojení v samostatném vlákně. Pro každé přijaté spojení spustí nové vlákno pro obsluhu klienta. Všechna vlákna pro obsluhu klienta sdílejí stejné Pico, takže ho musíme před pokusem o komunikaci řádně uzamknout, protože nás může kdykoli přerušit jiný klient. Server se ukončí, když obdrží KeyboardInterrupt (Ctrl-C).

```
from multiprocessing.connection import Listener, Client
from threading import Thread, Lock, Event
import time

import pyvisa as visa

#for wrong passwords
from multiprocessing.context import AuthenticationError

rm = visa.ResourceManager()
pico = rm.open_resource(rm.list_resources()[-1],
                        write_termination='\n',
                        read_termination='\n')
pico_lock = Lock()

handlers_lock = Lock()
client_handlers = []

class ClientHandler(Thread):
    def __init__(self, conn, name=None):
        super().__init__()
        self.name = name
        self.conn = conn
        self._end = Event()

    def stop(self):
        self._end.set()

```

```
    def run(self):
        try:
            while not self._end.is_set():
                if self.conn.poll():
                    msg = self.conn.recv()
                    print(f"Received : {msg}")
                    if msg == '':
                        self.conn.send('')
                    else:
                        with pico_lock:
                            resp = pico.query(msg)
                        self.conn.send(resp)
                time.sleep(0.1)
        except EOFError:
            print(f"Client {self.name} quit")
        finally:
            self.conn.close()
            with handlers_lock:
                client_handlers.remove(self)

def run(listener):
    while True:
        try:
            print("Waiting for connection")
            conn = listener.accept()
            if hasattr(listener, 'last_accepted'):
                client_name = listener.last_accepted
            else:
                client_name = None
            handler = ClientHandler(conn, client_name)
            handler.start()
            client_handlers.append(handler)
        except OSError:
             print("Stopping listening")
             break
        except AuthenticationError:
             print("Connection attempt with wrong password.")

address = ('', 0)
password = b'NOFY080_2024'
try:
    with Listener(address, authkey=password) as listener:
        actual_address = listener.address
        print('Address :', actual_address)

        t = Thread(target=run, args=(listener,))
        t.start()

        try:
            while True:
                time.sleep(1)
        except KeyboardInterrupt:
            print("Quitting...")

        listener.close()

        try:
            #dummy connection to force the listener to close
            with Client(actual_address, authkey=password) as c:
                c.send('')
                c.recv()
        except ConnectionRefusedError:
            pass
        t.join()
        for handler in client_handlers:
```

TODO

Obrázek 3: Jeden počítač ovládající více experimentů.

```
94              handler.stop()
95
96          for handller in client_handlers:
97              handler.join()
98  finally:
99      pico.close()
100     rm.close()
```

7.3.1 Reálný případ použití v laboratoři

Na Obr. 7.3.1 je fotografie jednoho počítače, který ovládá tři experimenty připojené ke dvěma různým experimentálním sestavám. Každý experiment patří jinému studentovi, který může potřebovat spouštět a upravovat své měřicí skripty v Pythonu současně, takže jednoduché sdílení vzdálené plochy není k dispozici.

Nakonec jsme použili server pro přístroje podobný zjednodušenému příkladu výše, který studentům umožnil spouštět měřicí skripty na jakémkoli počítači ve stejné síti, dokonce i na jejich vlastních noteboocích.

A Lineární harmonický oscilátor

Lineární harmonický oscilátor s třením je popsán dynamickou rovnicí

$$\ddot{x} + \omega_0^2 x + \gamma \dot{x} = F(t)/m, \tag{21}$$

kde m je hmotnost oscilátoru, ω_0 je (úhlová) rezonanční frekvence, γ koeficient tření a $F(t)$ vnější síla. Za předpokladu, že síla má tvar $F(t) = \Re(\tilde{F}e^{i\omega t})$ a že řešení má tvar $x(t) = \Re(\tilde{x}e^{i\omega t})$ (nebo aplikací Fourierovy transformace na obě strany rovnice), dostaneme jednoduchým přeuspořádáním

$$\tilde{x} = \frac{1}{m}\frac{\tilde{F}}{\omega_0^2 - \omega^2 + i\gamma\omega} = \chi(\omega)\tilde{F}, \tag{22}$$

kde $\chi(\omega)$ se nazývá susceptibilita.

Protože rovnice 21 je lineární rovnice, odezva na součet sil bude součtem odezev na každou sílu.

B Nastavení komunikace s přístroji

Ke komunikaci s přístroji budeme používat knihovnu VISA. Doporučený postpu pro Windows je nainstalovat NI-VISA a `pyvisa` pomocí `pip`.

Je ale také možno použít čistě pythonovou open-source implementaci této knihovny, pro kterou potřebujeme nainstalovat alespoň `pyvisa, pyvisa-py` a `pyusb` pomocí

```
pip install pyvisa pyvisa-py pyusb
```

který byste měli spustit v příkazovém řádku, který zná vaši instalaci pythonu (např. ve virtuálním prostředí nebo v Anaconda Prompt, pokud jste na Windows s distribucí Anaconda Python). Na Linuxu byste se také měli přidat do skupiny `dialout` (nezapomeňte na přepínač `-a`),

```
sudo usermod -a <your username> -G dialout
```

a odhlásit se a znovu přihlásit.

Pro otestování instalace spusťte z příkazového řádku `pyvisa-info`. Mělo by se vypsat množství informací, hledejte řádek, který vypadá jako

```
USB INSTR: Available via PyUSB (1.2.1). Backend: libusb1
```

Na Windows býva nutno nainstalovat knihovnu `libusb1` separátně, např. pomocí

```
pip install libusb-package
```

Dále vytvořte Python skript `list_resources.py` s následujícím kódem

```
import pyvisa as visa
rm = visa.ResourceManager()
print(rm.list_resources())
rm.close()
```

a spusťte ho. Měl by vypsat buď nic, nebo několik portů COM (nebo tty na Linuxu). Poté připojte Pico a spusťte program znovu, měla by se objevit nová adresa, to je adresa, kterou budeme používat pro komunikaci s Pico.

Dále spusťte následující kód a nahraďte `"ASRL/dev/ttyACM0::INSTR"` adresou, kterou jste našli v předchozím kroku.

```
import pyvisa as visa
rm = visa.ResourceManager()
pico = rm.open_resource('ASRL/dev/ttyACM0::INSTR',
                        read_termination='\n',
                        write_termination='\n')
print(pico.query('*IDN?'))
pico.close()
rm.close()
```

Program by měl vypsat "PICO"a ukončit se bez chyby.

Pro plnohodnotnější implementaci VISA můžete zvážit implementaci od National Instruments (NI). Knihovna NI-VISA je zdarma, ale není open source a vyžaduje registraci. Podpora Linuxu je také omezena na zastaralá jádra.

B.1 Podporované příkazy

***IDN?** Dotaz na identifikační řetězec. Měl by odpovědět PICO.
:LED n m Zapne (m=1) nebo vypne (m=0) LED diodu n. Číslo LED $n = 0 \ldots 4$, červená LED je 0.
:READ:P? Přečte tlak v Pa.
:READ:T? Vrací teploty jako $100T$, kde T je teplota v °C.
:READ:PT? Přečte teplotu i tlak.
:READ:ACC? Přečte akcelerometr. Vrací tři hodnoty oddělené mezerou v rozsahu -32768 až 32768, což odpovídá $-2g$ až $2g$.
:READ:GYR? Přečte gyroskop. Vrací tři hodnoty oddělené mezerou v rozsahu -32768 až 32768, což odpovídá $-500°/s$ až $500°/s$.

B.2 Hackování firmwaru

Zdrojový kód programu běžícího na Pico je dostupný zde. Pro jeho kompilaci je třeba nastavit Pico SDK, postupujte podle pokynů zde.

Alternativně můžete použít MicroPython pro spouštění Python kódu přímo na Pico, postupujte podle pokynů pro nastavení zde.

LED diody jsou připojeny k pinům `GP0` – `GP4` a senzory jsou připojeny k řadiči `I2C0` na pinech 16 a 17.

C Základy Linuxu

Linux je jádro operačního systému – část, která přímo spravuje přístup k hardwaru a nemůže být přímo použita uživatelem. V kombinaci s běžně používanou sadou programů vyvinutých např. projektem GNU vzniká skutečný operační systém, někdy nazývaný "GNU/Linux" (ačkoli téměř každý ho nazývá prostě Linux).

Přidáním dalších funkcí (typicky instalátor, správce balíčků a grafické uživatelské rozhraní) vzniká *linuxová distribuce* (nebo *distro*), např. Ubuntu, Fedora, OpenSUSE, Mint, Red Hat Enterprise Linux, ... nebo Gentoo, v případě laboratorních počítačů.

C.1 Struktura souborového systému

Linux a obecně systémy podobné Unixu nepoužívají strukturu souborového systému, která může být známá z Microsoft Windows. Konkrétně, souborový systém začíná v **kořenovém adresáři** označeném jediným lomítkem `/`, ze kterého se větví všechny ostatní adresáře. Více pevných disků nebo oddílů na jednom disku (které by se ve Windows jmenovaly něco jako `C:\` atd.) vypadají a chovají se jako běžné adresáře.

Důležité adresáře jsou `/home`, kde si uživatelé mohou ukládat svá data ve svých **domovských adresářích**, nebo `/bin`, kde jsou uloženy spustitelné programy.

C.2 Přesun souborů mezi laboratoří a osobním počítačem

Všichni studenti mají účty pro ukládání svých souborů na fakultních serverech. K nim lze přistupovat přes web na adrese `https://su.mff.cuni.cz/`

Můžete k nim také přistupovat pomocí správce souborů v Linuxu přechodem na `sftp://su.mff.cuni.cz/home/university_username` nebo `\\su.mff.cuni.cz\home\university_username` ve Windows a přihlášením se svými univerzitními přihlašovacími údaji. `university_username` je slovo založené na vašem jméně, nikoli ID číslo.

C.3 Rozhraní příkazového řádku

Operační systém nám poskytuje přístup ke svým službám (např. zápis na disk nebo tisk na obrazovku) prostřednictvím sady programů nazývaných **shell**. Shelly mohou být buď grafické (grafické uživatelské rozhraní, GUI) nebo rozhraní příkazového řádku (CLI). V Linuxu existuje mnoho grafických shellů, nejoblíbenější jsou Gnome, KDE (používané na laboratorních počítačích) a (hádám) Xfce. Nejčastěji používaný textový shell se nazývá **bash**. Pro použití CLI můžeme použít „emulátor terminálu" z GUI. Každé GUI poskytuje svou vlastní verzi (např. `konsole` pro KDE, `gnome-terminal` pro Gnome). Téměř všechny linuxové distribuce přiřazují spuštění emulátoru terminálu klávesové zkratce Ctrl-Alt-T.

Základní příkazy - Navigace Pro základní navigaci můžeme použít

- `ls` – vypíše obsah aktuálního adresáře
- `cd` – změní adresář
- `pwd` – vypíše aktuální pracovní adresář
- `tree` – vypíše stromovou reprezentaci aktuálního adresáře a jeho obsahu

Chování příkazů lze upravit pomocí *voleb*, obvykle začínajících jedním nebo dvěma pomlčkami, např. `ls -l` vypíše aktuální adresář v jednosloupcovém formátu s některými dalšími informacemi; `ls -la` také vypíše skryté soubory (ty, jejichž název začíná tečkou `.`) a `ls -lh` vypíše velikosti souborů v lidsky čitelném formátu (kilobajty, megabajty…) místo pouhých bajtů. Všimněte si, že je poměrně běžné, že názvy souborů v Linuxu nemají přípony (např. .exe, .txt atd.). Příkaz `tree` ve výchozím nastavení vypíše celý obsah všech podadresářů, rekurzivně od aktuálního adresáře, a není tedy příliš užitečný v kořeni větších adresářů (např. vašeho `~`). Volitelný argument `-L n` omezuje hloubku, do které se podadresáře vypisují, např. `tree -L 2` vypíše obsah aktuálního adresáře, jeho podadresářů a jejich podadresářů.

I když je často pohodlnější pracovat s grafickým správcem souborů, někdy je nutné přejít do CLI, například pro spuštění programu. Většina správců souborů to nějakým způsobem podporuje (obvykle pravým kliknutím > otevřít v terminálu nebo něco podobného).

Speciální názvy adresářů jsou `.` (tečka), což znamená aktuální adresář, a `..` (dvě tečky), což odkazuje na nadřazený adresář, a `~` (tilda, nad klávesou Tab), což odkazuje na váš domovský adresář (což by pro mě na mém počítači bylo `/home/vargaem`).

Např. `cd ..` změní adresář o jednu úroveň výš.

Většina emulátorů terminálu vám pomůže ušetřit psaní pomocí automatického doplňování názvů souborů, složek nebo příkazů. Stisknutím klávesy Tab se doplní aktuální slovo, jak jen to je možné jednoznačně. Dvojitým stisknutím klávesy Tab získáte seznam možných doplnění.

Cvičení C.1. Vytvořte soubor s nějakým textem v podadresáři ve vašem domovském adresáři pomocí GUI. Poté tento soubor najděte pomocí CLI.

Cesty Při pohybu v souborovém systému máme dvě možnosti, jak specifikovat cesty k přesnému souboru nebo adresáři, který chceme – *absolutní* nebo *relativní* cesty. To není specifické pro Linux, všechny operační systémy poskytují tyto dvě možnosti nějakým způsobem.

- absolutní cesty – cesta, která specifikuje absolutní pozici souboru v souborovém systému. V Linuxu tyto cesty vždy začínají `/`, tj. `/etc/fstab` (soubor, který v Linuxu obsahuje informace o rozdělení souborového systému). Ve Windows tyto cesty začínají písmenem jednotky, tj. `C:\Windows\system32`.
- relativní cesty – cesta relativní k aktuální pozici, jak je dána příkazem `pwd`. Soubory a složky se hledají v aktuálním pracovním adresáři. Pokud potřebujeme odkazovat na soubor v nadřazeném adresáři, musíme použít `..`

Zástupné znaky Někdy chceme pracovat pouze s názvy souborů nebo adresářů, které odpovídají určitému vzoru. Tento vzor lze nejsnadněji specifikovat pomocí *zástupných znaků* - `*` (hvězdička) nahrazuje libovolný počet (včetně nuly) libovolných znaků; `?` nahrazuje přesně jeden znak. Tyto zástupné znaky lze použít s `ls`, např.

```
ls *.txt
```

vypíše všechny soubory `.txt` v aktuálním adresáři

```
ls ab?.pdf
```

vypíše všechny třípísmenné PDF soubory, které začínají na `ab`. Složitější vzory lze vytvořit specifikací, které znaky mohou být do vzoru dosazeny, pomocí hranatých závorek `[]`, např. `data_[1-9].txt` bude odpovídat `data_1.txt`, `data_2.txt`,...`data_9.txt`.

Cvičení C.2. Vypište všechny soubory z adresáře `/usr/include`, které začínají na 'a' a mají příponu '.h'.

nápověda a manuálové stránky Příkazy mohou být od poměrně jednoduchých, jako je `cd`, až po poměrně složité, jako je `find` (viz níže), kde je často třeba nahlédnout do dokumentace. Většina příkazů přijímá volbu `-h` nebo `--help`, která vypíše krátkou nápovědu. Pro podrobnější dokumentaci můžeme použít příkaz `man`, který zobrazí tzv. *manuálovou stránku* daného příkazu (zkratka z **man**uál), např.

```
man ls
```

zobrazí dokumentaci pro příkaz `ls`.

Čtení souborů Textové soubory lze číst a zobrazovat přímo v CLI (pro úpravy viz další sekce) pomocí příkazu `cat` (zkratka z con**cat**enate), který vypíše jeden nebo více souborů jako text přímo na příkazový řádek.

Pokud chceme číst dlouhý textový soubor, možná budeme potřebovat možnost posouvat zobrazení. K tomu můžeme použít `less`, který přijímá jeden název souboru a umožňuje posouvat zobrazení pomocí šipek a ukončit stisknutím *q*.

Kopírování, přesouvání a mazání Soubory **kopírujeme** pomocí `cp`. Pro zkopírování souboru s názvem `source` z podadresáře `path/to` do aktuálního adresáře a jeho pojmenování `destination` použijeme

```
cp path/to/source destination
```

pokud je cíl adresář, soubor se do něj zkopíruje s původním názvem. Pokud chceme zkopírovat adresář a vše v něm, můžeme použít `cp -R` (R pro **R**ekurzivní).

Podobný vzor používá příkaz pro **přesun** `mv` (kromě toho, že `R se nepoužívá pro přesun adresářů`). Jak `mv`, tak `cp` podporují zástupné znaky, např. `mv *.txt texts/` přesune všechny soubory `*.txt` do adresáře `texts`.
Nové adresáře lze vytvořit pomocí `mkdir`.
Soubory se mažou pomocí `rm`, který také podporuje zástupné znaky. Pouze prázdné adresáře lze odstranit pomocí `rmdir`. Adresáře s obsahem lze odstranit pomocí `rm -rf`. POZOR `rm` maže soubory trvale, nepřesouvá je do koše nebo něčeho podobného. `rm -rf` v kombinaci se zástupnými znaky může vést k katastrofám, pokud nejste opatrní, např.

```
rm -rf tmp*
```

by odstranil všechny soubory a adresáře začínající na ,,tmp'' (možná nějaké dočasné soubory, které již nejsou potřeba), ale

```
rm -rf tmp *
```

by odstranil soubor s názvem tmp a vše v aktuálním adresáři bez žádosti o potvrzení.

Cvičení C.3. Vytvořte adresář, např. „exercise1", a zkopírujte do něj všechny `.h` soubory z `/usr/include`, které začínají na a nebo b. Poté smažte všechny soubory začínající na b v exercise1; poté smažte celý adresář exercise1 (s některými soubory stále uvnitř).

editace textových souborů

- `nano`
  Nano je často nainstalován ve výchozím nastavení. Spodní řádek zobrazuje klávesové zkratky pro základní operace. Stříška ^ znamená Ctrl, M znamená Alt, tj. pro ukončení nám nano říká, abychom udělali ^X, což znamená stisknutí Ctrl-X současně. Pro vrácení zpět nebo opakování bychom měli udělat M-U nebo M-E, což znamená stisknutí Alt-U nebo Alt-E. nano odkazuje na zobrazený obsah jako na *buffery*, které se pak ukládají do souborů na disku.
- `emacs`
  Mnoho funkcí, které mohou být obtížně nastavitelné. Lepší se vyhnout.
- `vim` -- Vim rozlišuje mezi *editačním režimem* a *příkazovým režimem*. Ve výchozím nastavení se spouští v příkazovém režimu, kde editor očekává nějaké příkazy. Pro vstup do editačního režimu stiskněte i a dole se objeví --INSERT--, což indikuje editační režim.
  Pro uložení souboru vstupte do příkazového režimu stisknutím Esc (--INSERT-- dole zmizí) a napište :w+Enter. Pro ukončení použijte :q. Pro ukončení bez uložení :q!. Pro ukončení a uložení :wq.
  Soubory lze prohledávat pomocí Esc-/ *vyhledávací vzor*

vyhledávání příkaz `find` lze použít k provádění poměrně složitých úkolů, ale pro jednoduché vyhledávání lze použít

```
find directory_name -name 'name_pattern'
```

např. `find . -name '*.pdf'` vypíše všechny PDF soubory v aktuálním adresáři včetně podadresářů.
Vyhledávání je citlivé na velikost písmen. Pro vyhledávání bez ohledu na velikost písmen nahrad'te `-name` za `-iname`.

Cvičení C.4. Najděte všechny soubory, které končí na '.conf' v /etc a jeho podadresářích.

Skládání a rozšiřování Bash umožňuje přesměrovat výstup jednoho příkazu někam jinam než jen na standardní výpis na obrazovku. Pokud chceme uložit výstup příkazu do souboru, můžeme použít přesměrování `>` např.

```
ls -l > directory_contents.txt
```

uloží výstup `ls -l` do souboru s názvem `directory\_contents.txt`. Přesměrování lze použít s příkazem `echo`, který jednoduše vypíše zpět svůj vstup, pro rychlé vytváření jednoduchých souborů, např.

```
echo A single line in a simple file > file.txt
```

vytvoří soubor file.txt, který bude obsahovat ,,A single line in a simple file''. *Poznámka:* prázdný soubor lze rychle vytvořit pomocí touch empty_file. Pokud chceme použít výstup jednoho příkazu jako vstup jiného, můžeme použít rouru (pipe) \|, např.

```
ls -l | less
```

nám umožní procházet výstup ls -l šipkami díky příkazu less.
Spojení více souborů lze provést jednoduše pomocí cat

```
cat file1.txt file2.txt file3.txt \> concatenated_file.txt
```

Pro třídění (užitečné s find) můžeme použít sort, např.

```
find . -name '*.pdf' | sort
```

vypíše všechny PDF soubory v aktuálním adresáři, včetně podadresářů, v abecedním pořadí.
Běžně používané příkazy s rourami jsou head a tail, které zobrazují prvních nebo posledních 10 řádků svého vstupu. Oba přijímají volbu -n, která mění výchozích 10 řádků, např.

```
cat long_text_file.txt | head -n 50
```

zobrazí prvních 50 řádků ze souboru long_text_file.txt.

Cvičení C.5. Jako v Úloze 4, ale seřaďte conf soubory abecedně a uložte je do textového souboru ve vašem domovském adresáři.

C.3.1 Uživatelé a oprávnění

Uživatelé Běžní uživatelé obecně nemají oprávnění provádět libovolné změny v operačním systému. Uživatel, který může s počítačem dělat cokoli, se nazývá root(nezaměňovat s *kořenovým adresářem* /). Pro zjištění, pod jakým uživatelským jménem jsme přihlášeni, můžeme použít příkaz whoami.

Oprávnění a skupiny Operační systém řídí, kdo co může dělat s kterými soubory, pomocí oprávnění. Oprávnění mohou být jakákoli a všechna z *čtení*, *zápisu* a *spuštění* (nebo *výpisu* pro adresáře). Oprávnění se zobrazují pomocí ls -l v prvním sloupci, např. pro náhodný soubor na mém počítači

```
ls -l /bin/bash
-rwxr-xr-x. 1 root root 1444200 Feb  6  2023 /bin/bash
```

Všimněte si, že oprávnění se zdají být uvedena třikrát. První pomlčka ,,-'' označuje běžný soubor; následuje ,,rwx'', což znamená, že vlastník souboru (v tomto případě uživatel ,,root'') má oprávnění ke čtení, zápisu a spuštění; další ,,r-x'' znamená, že uživatelé, kteří patří do skupiny (v tomto případě také nazývané ,,root''), mohou soubor číst a spouštět, ale ne do něj zapisovat. Nakonec poslední ,,r-x'' znamená, že všichni ostatní mohou soubor číst a spouštět, ale ne do něj zapisovat.

sudo I když používáte svůj osobní počítač, z bezpečnostních důvodů je obecně špatný nápad pracovat s účtem root pro běžné úkoly. Když potřebujeme spustit příkaz, který smí provést pouze root (např. instalovat program), použijeme sudo, např.

```
ls sudo apt-get install vim
```

vás požádá o heslo a poté nainstaluje textový editor vim na distribucích Linuxu založených na Debianu (např. Ubuntu). Většina osobních instalací Linuxu používá pouze jedno heslo pro uživatele a sudo. Pouze administrátor má heslo roota na laboratorních počítačích.

C.4 Přístup ke vzdáleným serverům pomocí CLI

Pokud máme na vzdáleném stroji běžný účet, můžeme se na něj přihlásit pomocí

```
ssh username@remote.server.cz
```

Pokud můžeme pouze přesouvat soubory (případ univerzitního studentského úložiště), v CLI, pro zkopírování souboru z lokálního počítače na vzdálený server, kde máme účet, můžeme použít scp, např.

```
scp file username@su.mff.cuni.cz:
```

vás požádá o heslo a poté zkopíruje soubor do vašeho studentského úložiště. Pro stažení souboru

```
scp username@su.mff.cuni.cz:path/to/a/file .
```

stáhne soubor ze serveru do aktuálního adresáře (tečka na konci).
Další možností je použít sftp, pomocí kterého se můžete přihlásit na server, což vám poskytne omezenou sadu příkazů (ls, mkdir, cp, mv atd.), pomocí kterých můžete organizovat své soubory na vzdáleném serveru a možnost nahrávat soubory pomocí příkazu put nebo stahovat soubory pomocí příkazu get.

Cvičení C.6. Zkuste nahrát soubor a ověřte, že je přítomen na vzdáleném serveru pomocí grafického průzkumníka souborů. Naopak, vytvořte soubor na vzdáleném serveru pomocí GUI a stáhněte ho pomocí CLI.

Alternativou ke studentskému úložišti je také univerzitní cloud OneDrive s webovým rozhraním (součást Office 365). Klient pro Windows funguje dobře, existuje také klient pro Linux, který je trochu složitější na nastavení, ale také funguje.