Pedagogická fakulta
Faculty of Education

Jihočeská univerzita v Českých Budějovicích
University of South Bohemia in České Budějovice

Jihočeská univerzita v Českých Budějovicích

Pedagogická fakulta

Katedra aplikované fyziky a techniky

Diplomová práce

Základy jazyka Python s příklady jeho aplikací ve fyzice

Vypracoval: Bc. Tomáš Luksch

Vedoucí práce: doc. RNDr. Josef Blažek, CSc.

České Budějovice 2018

Prohlášení

Prohlašuji, že svoji diplomovou práci jsem vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury.

Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své diplomové práce, a to v nezkrácené podobě elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách, a to se zachováním mého autorského práva k odevzdanému textu této kvalifikační práce. Souhlasím dále s tím, aby toutéž elektronickou cestou byly v souladu s uvedeným ustanovením zákona č. 111/1998 Sb. zveřejněny posudky školitele a oponentů práce i záznam o průběhu a výsledku obhajoby kvalifikační práce. Rovněž souhlasím s porovnáním textu mé kvalifikační práce s databází kvalifikačních prací Theses.cz provozovanou Národním registrem vysokoškolských kvalifikačních prací a systémem na odhalování plagiátů.

V Českých Budějovicích dne 21. dubna 2018

Bc. Tomáš Luksch

Anotace

Cílem diplomové práce je tvorba výukového materiálu k programovacímu jazyku Python, určeného primárně pro studenty fyziky na pedagogických fakultách. Výukové materiály zahrnují obecný popis jazyka, včetně několika jeho matematických knihoven, úvod do principů programování v Pythonu spolu s příklady použití standardních příkazů a funkcí a konkrétní příklady využití jazyka v matematických úlohách a fyzikálních simulacích.

Klíčová slova

Python, programování, fyzika, matematika, simulace

Abstract

The aim of this thesis is to provide learning materials for programming in Python, primarily for physics students at the faculties of education. Created materials include general description of the language, including several mathematical libraries, basics of the programming in Python along with examples of standard commands and functions and particular examples of application of the language in mathematical problems and physical simulations.

Keywords

Python, programming, physics, mathematics, simulation

Poděkování

Rád bych poděkoval panu doc. RNDr. Josefu Blažkovi, CSc. za věcné připomínky k podobě a obsahu práce, za značnou pomoc se správnou prezentací hlavně matematických a statistických metod a za neméně nápomocné doporučení a zapůjčení odborné literatury. Další velký dík patří rodině, a to hlavně za trpělivost a podporu. Nakonec nesmím opomenout ani kruh přátel za mnoho cenných rad.

Obsah

Úvod

Výpočetní technika a fyzika jsou dvě oblasti z principu úzce propojené. První by bez druhé nejspíše nemohla vzniknout, druhá by bez první zase zažívala mnohem pomalejší rozvoj. Jedním z cílů, které si tato práce proto klade, je přiblížit využití výpočetní techniky ve fyzice a matematice a rozšířit možnosti počítačem podporované výuky fyziky, mimo jiné v oblasti modelování a simulace fyzikálních dějů.

Jazyk MATLAB, který se na KAFT PF JU v Českých Budějovicích vyučuje, sice tomuto účelu vyhovuje, je ale komerční a tím je pro studenty a většinu učitelů prakticky nedostupný. Je proto dobré uvažovat, zda v budoucnu jazyk MATLAB nenahradit jiným jazykem, který by byl dostatečně jednoduchý a přitom volně dostupný.

Jazyk Python tyto podmínky splňuje. Hlavním cílem této práce je proto zpřístupnit a zpopularizovat Python. Ten je díky své jednoduché syntaxi vhodný pro začátečníky, je zdarma dostupný včetně plejády externích vědeckých i dalších knihoven a může se pyšnit vstřícnou a neustále rostoucí komunitou.

1 Programovací jazyk Python

Python je programovací (dle některých definicí spíše skriptovací) jazyk, navržený Guidem van Rossumem v roce 1989 [1]. Název ‚Python' částečně vychází z autorovy obliby britského komediálního uskupení Monty Python's Flying Circus [2].

Python je zařazován mezi vysokoúrovňové programovací jazyky, neboť přichází s poměrně jednoduchou a v základu přímočarou syntaxí, která v mnohém (na rozdíl od „nižších" jazyků) může připomínat skutečný jazyk. Python je oproštěn od nutnosti předem deklarovat proměnné (tzn. určovat jejich datový typ), takže je možné do proměnné, která v sobě původně měla uložené celé číslo, uložit například řetězec (Př. 1).

```
1 a = 2
2 print(a)
3
4 a = "retezec"
5 print(a)
```

Výstup:

```
2
retezec
```

Příklad 1: Dynamický datový typ

A právě tohle je jeden z důvodů, proč se o Pythonu často mluví nikoli jako jazyku programovacím, nýbrž skriptovacím. Dalším rozdílem a pro začátečníky i značnou výhodou oproti nižším programovacím jazykům, jakým je například jazyk C, je i nepotřebnost se v základu ručně starat o správu paměti [1]. Samozřejmostí je i velmi dobrá přenositelnost kódu mezi zařízeními a různými operačními systémy[3].

Mezi další výhody patří třeba rozsáhlá knihovna modulů, které zásadně rozšiřují funkcionalitu Pythonu, přičemž spousta z nich je součástí jeho základní knihovny, se kterou instalace Pythonu přichází. Ostatní knihovny a moduly, včetně mnoha určených právě pro matematické a fyzikální využití, je možno zdarma stáhnout.

Přes velké množství výhod oproti ostatním programovacím jazykům musí mít Python samozřejmě i nějaké nevýhody. Jako jednu z nich můžeme jmenovat například jeho rychlost. Během mnoha aplikací si pomalejšího výkonu jazyka vůbec nevšimneme, ovšem obecně je Python pomalejší než plně kompilované jazyky, jakým je například C. Tento rozdíl se ale výrazně kompenzuje při porovnání času, který je potřeba vyhradit na napsání programu v Pythonu a v jazyce C. Druhý jmenovaný bude, pro svou mnohem složitější syntaxi a nutnost starat se o více věcí, od programátora vyžadovat daleko více práce. [1]

Další nevýhodou v mnoha případech může být složitost skrytí nebo znečitelnění

zdrojového kódu tak, aby jej ostatní nemohli kopírovat a upravovat[1].

K Pythonu se váže nevýhoda, spojená s tím, že jazyk je velmi rychle vyvíjen, což vede k následným problémům v kompatibilitě, kdy jsou mezi jednotlivými verzemi jazyka mnohdy zásadní rozdíly. Zpětná kompatibilita jazyka by ovšem zajištěna být měla, takže program napsaný ve starší verzi by měl být bez problémů spustitelný i ve verzi novější. V obráceném případě už bychom ale s největší pravděpodobností narazili na potíže.[1]

V této souvislosti je nutné zmínit, že tato práce je psána pro Python ve verzi 3.5.

1.1 Instalace Pythonu

Instalaci Pythonu je možno provést mnoha způsoby a z mnoha zdrojů v závislosti například na účelu a zaměření naší práce s ním. První a zřejmě nejstandardnější možností je získat základní instalaci Pythonu z jeho oficiálních stránek python.org, kde lze mimo jiné najít i rozsáhlou dokumentaci k jazyku. Zde distribuovaná verze obsahuje základní knihovny a moduly. Další, které budou potřeba pro některé z příkladů dál, bude tedy nutno následně doinstalovat.

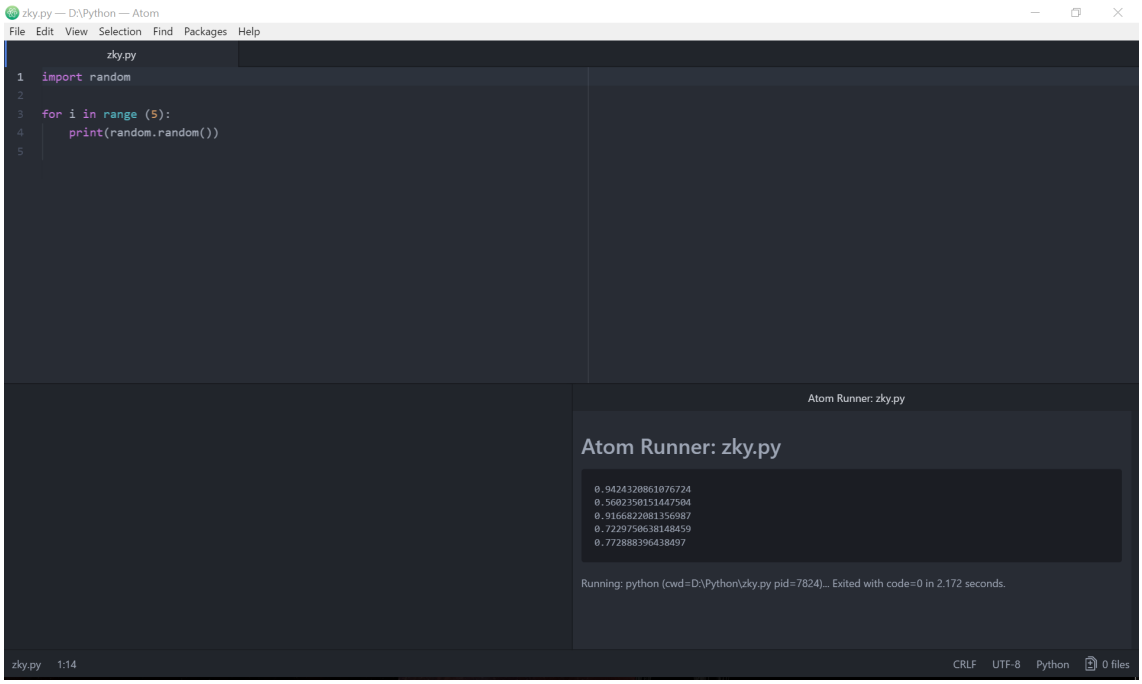
Pro „vědecké“ programování je doporučována [1] například distribuce *Anaconda* nebo *Enthought canopy*, které obsahují i knihovny NumPy, SciPy a Matpotlib, které budeme hojně využívat.

Tato práce však bude počítat s původně čistou instalací Pythonu a proto bude dále ve stručnosti popsáno i ruční přidávání nových knihoven.

Během instalace je dobré nezapomenout zaškrtnout volbu *Add Python 3.5* (nebo jiné verze) *to PATH*, pro zjednodušení následné konfigurace.

Pro práci se samotným kódem i jeho spouštěním lze doporučit textový editor Atom, který lze stáhnout z oficiálních stránek atom.io. Po jeho nainstalování je však potřeba jej ještě lehce donastavit a doinstalovat knihovny pro spouštění pythonovských skriptů. Nejprve si tedy v nastavení (*File -> Settings*) v submenu *Editor* zaškrtneme volby *Show Indent Guide* (pro lepší orientaci v odsazeních), *Soft Tabs* a *Tab Length* nastavíme na hodnotu 4. Dále v submenu *Packages* najdeme balíček *Autosave*, který aktivujeme. Nakonec v submenu *Install* necháme vyhledat balíček *atom-runner*, který nainstalujeme. Ten nám dovolí spouštět skripty přímo v editoru bez nutnosti zaobírat se spouštěním přes příkazový řádek. Po absolvování těchto kroků je dobré restartovat počítač.

Po napsání kódu v Atomu je potřeba daný soubor uložit s koncovkou .py, aby editor věděl, v jakém jazyce je program napsán. Následně je možné jej spustit pomocí klávesové zkratky *alt+r*. Výsledek by měl vypadat podobně jako na screenshotu (Obr. 1), kde vlevo nahoře vidíme spouštěný kód a vpravo dole jeho výsledek.



Obrázek 1: Kód v Atomu

1.2 Instalace modulů

Jak již bylo řečeno výše, existují distribuce Pythonu, které rovnou přicházejí s mnohými moduly jakožto jejich součástmi. My jsme si však z didaktických důvodů nainstalovali čistou verzi Pythonu, takže si budeme muset požadované moduly doinstalovat.

To uděláme tak, že si otevřeme příkazový řádek ve složce, kde máme Python nainstalovaný, a vstoupíme do podsložky *Scripts*. Zde příkazem `pip install numpy` nainstalujeme modul NumPy. Stejně tak příkazem `pip install matplotlib` nainstalujeme modul Matplotlib.

Obdobným způsobem lze nainstalovat i další moduly Pythonu. V případě potíží nebo nejasností existuje k většině modulů i množství návodů a popisů dalších způsobů instalace, včetně instalace celých distribucí, které byly zmíněny výše.

1.3 Zaměření modulů

V předchozí podkapitole jsme se dozvěděli o existenci několika modulů, o kterých si nyní povíme něco více.

1.3.1 NumPy

NumPy je v dnešní době standardním a pravděpodobně nejpoužívanějším modulem při běžných vědeckých operacích v Pythonu. Nabízí numerické nástroje potřebné pro generování a analyzování dat [4]. Využíván je také pro svou podporu

tvorby N-prostorových matic a polí. Objekt, který v NumPy toto zajišťuje, se nazývá `ndarray`, což je multidimenzionální pole, které může být upravováno, včetně změny jeho rozměrů, a přeskládáno, lze s ním provádět matematické operace a statistické analýzy, zapisovat a číst s ním ze souborů a mnoho dalšího. Hlavní výhodou používání objektů NumPy oproti těm v samotném jádře Pythonu je jejich předkompilování jakožto C kódu [1], což může značně urychlit chod programu.

Další výhodou NumPy je podpora vektorizace. To znamená, že provádění matematických operací lze vykonat s celou maticí bez toho, aby bylo nutné programovat cyklus, který by tu samou operaci provedl samostatně pro každý z prvků matice. To samozřejmě jednak zpřehledňuje a zjednodušuje napsaný kód, jednak je i tato funkčnost předkompilována v jazyce C, tudíž probíhá daleko rychleji, než kdybychom se o to pokoušeli pomocí zabudovaných funkcí Pythonu[1].

Pro použití NumPy je potřeba jej do kódu importovat, což lze udělat příkazem `import numpy` či `import numpy as np`. Pak je možno na modul v kódu odkazovat pomocí prefixu `numpy` – například `numpy.array`, či pomocí prefixu `np` – například `np.array`, v závislosti na tom, jaký zápis pro import si zvolíme.

Při importu je místo `np` možno použít i jiný prefix, pak se ovšem ten samý musí použít i při následném volání modulu.

1.3.2 Matplotlib

Tato knihovna je spolu s jejími součástmi *PyPlot* a *pylab* používána k vizualizaci dat [4]. Primárně je určená pro tvorbu 2D grafů, ovšem s určitými omezeními je možné ji používat i pro třírozměrné grafy.

Pro import modulu můžeme použít příkaz `import matplotlib.pyplot as plt`.

1.3.3 SciPy

Knihovna modulů *SciPy* je, jak již název napovídá, také určena pro vědecké výpočty. Na rozdíl od modulu *NumPy*, který obsahuje generické datové struktury a matematické algoritmy, *SciPy* nabízí daleko specifičtější součásti pro vyhodnocování speciálních funkcí, se kterými se můžeme ve vědecké a technické sféře setkat, například modul `scipy.constants` obsahuje fyzikální konstanty, které je z něj možno vytáhnout a použít. Podobně jako v případě NumPy, i SciPy má velké množství algoritmů předkompilovaných v jazyce C, tudíž při používání jeho součástí dochází k rychlejšímu chodu programu. [1]

2 Úvod do programování v Pythonu

V prvé řadě je nutno si uvědomit, že Python je programovacím jazykem a jako takový má určitá pravidla zápisu. Z Javy nebo C# můžeme být zvyklí na oddělování jednotlivých příkazů pomocí středníků, případně uzavírání složitějších výrazů (cyklů, podmínek, ...) do složených závorek. V případě Pythonu jsme si ovšem na ukázaných příkladech (Př. 1, Obr. 1) mohli všimnout, že za jednotlivými inicializacemi proměnných ani za příkazy se středníky nevyskytují, stejně tak zapsaný cyklus není uzavřený ve složené závorce. Python totiž jednotlivé příkazy čte po řádcích. Pokud se pak jedná o cykly a podmínky, případně o příkazy v nich vnořené, určuje jejich příslušnost podle odsazení řádků – na to je tedy při psaní složitějších kódů potřeba dávat pozor.

Porovnejte kód v Pythonu a v C# (Př. 2 a Př. 3).

```
1 for i in range(10):
2     i = i + 1
3
4 print(i)
```

Příklad 2: Zápis cyklu v Pythonu

```
1 for (int i = 0; i < 10; i++) {
2     i = i + 1
3 }
4
5 Console.WriteLine(i);
```

Příklad 3: Zápis cyklu v C♯

2.1 Objekty

Zjednodušeně můžeme říct, že veškerá čísla, řetězce, nebo třeba i výše zmíněné matice, jsou objekty. Takže jakmile pracujeme i s pouhým číslem, pracujeme s objektem. Tato skutečnost bude důležitá v následujících podkapitolách, kde budeme notně využívat právě objektového pojetí Pythonu.

Objekty jako takové totiž nemusí obsahovat uložené hodnoty čísel, znaky a podobně, ale mohou obsahovat metody, funkce a dokonce i další objekty.

2.2 Proměnné

V Pythonu stejně jako v ostatních programovacích jazycích pracujeme s proměnnými, do kterých – zjednodušeně řečeno – ukládáme námi používané objekty. Pokud v kódu pracujeme například s nějakým číslem, počítač jej musí mít uložené v paměti. Budeme-li tedy pracovat například s číslem 10, toto číslo se v tu chvíli do paměti uloží na určitou adresu, kterou můžeme nechat vypsat pomocí příkazu `id` (Př. 4). Z příkladu si můžeme zároveň všimnout stylu psaní komentářů do kódu pomocí znaku *hash*.

```
1 print(id(10))     #vypsání adresy čísla 10
```

Výstup:

```
1705793408
```

Příklad 4: Adresa čísla v paměti

Pokud bychom však na toto číslo chtěli odkazovat, používat adresy v paměti počítače by bylo poměrně nešikovné. Proto je výhodné danému číslu přiřadit nějaký název, který jej bude zastupovat a pomocí kterého s ním budeme nadále pracovat (Př. 5). Tomuto názvu potom říkáme právě proměnná (nebo, pokud bychom chtěli být přesnější, identifikátor nebo identifikátorový název [1]).

```
1 print(id(10))     # vypsání adresy čísla 10
2
3 a = 10
4 print(id(a))      # vypsání adresy na níž ukazuje a
```

Výstup:

```
1705793408
1705793408
```

Příklad 5: Adresa proměnné v paměti

Vidíme, že adresa proměnné je stejná jako adresa čísla, se kterým pracujeme, tedy nám proměnná skutečně odkazuje na tuto hodnotu.

Takto inicializovanou proměnnou pak můžeme dále používat a provádět s ní další operace, používat ji jako argument ve funkcích apod.

Důležité je také vědět, že do jedné proměnné mohu uložit druhou (Př. 6).

```
1  a = 10
2  b = a
3
4  print(b)      # vypsání hodnoty b
```

Výstup:

```
10
```

Příklad 6: Uložení proměnné do proměnné

Když však takto přiřadíme proměnnou `a` do proměnné `b` a pak hodnotu proměnné `a` změníme, v proměnné `b` zůstane původní hodnota (Př. 7).

```
1  a = 10
2  b = a
3  a = 20
4
5  print(b)      #vypsání hodnoty b
```

Výstup:

```
10
```

Příklad 7: Změna původní proměnné

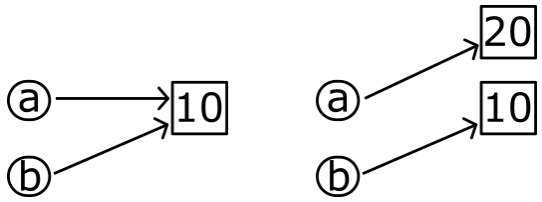
Je to tím, že změnou hodnoty v `a` neměníme hodnotu na daném místě v paměti, ale říkáme tím proměnné `a`, že se vlastně má zaměřit jinam – tam, co se do paměti uložila její nová hodnota, kterou jsme jí přidělili (Př 8., Obr. 2)[1].

```
1  a = 10
2  b = a
3
4  print(id(a))      #zobrazení adresy na níž ukazuje proměnná a
5  print(id(b))      #zobrazení adresy na níž ukazuje proměnná b
6
7  a = 20
8
9  print(id(a))      #zobrazení adresy na níž ukazuje proměnná a
10 print(id(b))      #zobrazení adresy na níž ukazuje proměnná b
```

Výstup:

```
1869961088
1869961088
1869961248
1869961088
```

Příklad 8: Změna adresy paměti



Obrázek 2: Přidělování hodnot proměnných

2.2.1 Názvy proměnných

Při pojmenovávání proměnných musíme respektovat určitá pravidla. V první řadě musíme brát na vědomí skutečnost, že jejich názvy jsou *case-sensitive*, to znamená, že záleží na tom, zda v názvu použijeme velké nebo malé písmeno – proměnná s názvem `obvod` je jiná než ta s názvem `Obvod`.

Námi vytvořené proměnné se nesmí jmenovat jako tzv. rezervovaná slova [1], která jsou v Pythonu používána mimo jiné pro základní funkce nebo příkazy pro cykly, podmínky a zabudované konstanty. Patří sem například námi často používaný `print` nebo `if`, `for`, `break`, atd.

Názvy proměnných mohou obsahovat všechna písmena, čísla a znak podtržítka, nesmí ovšem číslem začínat [1].

2.3 Čísla

Na rozdíl od jazyků jako je Java nebo C není v Pythonu nutné striktně určovat datové typy (Př. 1). Přesto Python pracuje se třemi druhy čísel.

Prvním z nich je *integer*, který zastupuje celá čísla. V praxi to znamená, že pokud pracujeme s čísly jako 2, 12, -123, ale i 65478965467, pracujeme s čísly typu `int`. Od Pythonu verze 3 a výše navíc ani není omezeno, jak velká čísla do proměnné typu `int` můžeme uložit, omezeni jsme pouze pamětí počítače [1].

Druhým číselným typem je *float*, neboli typ s *plovoucí* desetinnou čárkou. Patří sem tedy čísla jako 2.1, -7.8 nebo 1.7865×10^{-5} (pozor na zápis desetinné čárky, která se v Pythonu zapisuje jako tečka). Poslední uvedené číslo je v Pythonu také možno zapsat jako 1.7865e-5.

Třetím typem jsou čísla typu *complex* složená z reálné a imaginární části označené písmenem j. Může to být například číslo $5 + 2j$.

Jak ale určíme typ čísla, když víme, že se to v Pythonu neprovádí přímou deklarací? Jednoduše samotnou inicializací proměnné, ve které je uloženo (Př. 9).

```
1 a = -12        #číslo typu int
2 b = -1.2e3     #číslo typu float
3 c = 3 + 2j     #číslo typu complex
4 d = complex(3, 2)    #také číslo typu complex
5
6 print(a)
7 print(b)
8 print(c)
9 print(d)
```

Výstup:

```
-12
-1200.0
(3+2j)
(3+2j)
```

Příklad 9: Typy čísel

Jak vidíme, pokud do proměnné zadáme číslo s desetinnou tečkou, tedy typu *float*, na výstupu se vypíše také s desetinnou tečkou, přestože se jedná o celé číslo -1200, kde bychom ji vůbec nepotřebovali. To je dáno právě tím, že jsme číslo zadali jako *float*, a tak se i nadále prezentuje.

Také vidíme, že stejně jako *float* můžeme i komplexní čísla zadávat různými způsoby. Buď to standardně ve známém formátu s již zmíněnou reálnou částí a imaginární částí s písmenem *j*, nebo příkazem `complex` se dvěma parametry v závorce, které reálnou a imaginární část reprezentují. Zde už písmeno *j*, označující imaginární část, nepíšeme.

Pokud si nejsme jistí datovým typem různých proměnných, můžeme jej zjistit příkazem `type` (Př. 10).

```
1  a = -12        #číslo typu int
2  b = -1.2e3     #číslo typu float
3  c = 3 + 2j     #číslo typu complex
4  d = complex(3, 2)    #také číslo typu complex
5  e = "toto je string"
6
7  print("a je typu " + str(type(a)))
8  print("b je typu " + str(type(b)))
9  print("c je typu " + str(type(c)))
10 print("d je typu " + str(type(d)))
11 print("e je typu " + str(type(e)))
```

Výstup:

```
a je typu <class 'int'>
b je typu <class 'float'>
c je typu <class 'complex'>
d je typu <class 'complex'>
e je typu <class 'str'>
```

Příklad 10: Zjištění datového typu

Na výstupu je patrné, že dané proměnné jsou skutečně určitého typu. Příkaz `type` se neomezuje pouze na číselné datové typy, ale lze jej použít na jakýkoli jiný datový typ, což nám v příkladu dokazuje proměnná *e*, která je typu *string* (o řetězcích až později).

2.3.1 Operace s čísly

Kromě základních operací, jako je sčítání, odčítání a násobení, kterými není třeba se více zaobírat, Python v základu nabízí i další možnosti práce s čísly.

Jsou to především dva druhy dělení – desetinné a celočíselné (Př. 11).

```
1  a = 3
2  b = 2
3
4  print(a / b)     #desetinné dělení
5  print(a // b)    #celočíselné dělení
```

Výstup:

```
1.5
1
```

Příklad 11: Druhy dělení

Desetinné dělení provádíme pomocí jednoho lomítka a návratovou hodnotou je vždy číslo typu `float` nebo `complex`, i když bychom jako dělence a dělitele měli oba `int`. Celočíselné dělení provádíme dvojitým lomítkem a jeho výsledkem je `float` zaokrouhlený dolů. Pouze pokud jsou oba operátory typu `int`, bude i výsledek *integer*.

Stejné pravidlo platí i u operace *modulo*, která vrací zbytek po celočíselném dělení (Př. 12.)

Pokud bychom chtěli číslo umocnit na jiné číslo, použijeme k tomu dvojici hvězdiček – `**` (Př. 12).

```
1 a = 3
2 b = 2
3
4 print(a % b)
5 print(a**b)
```

Výstup:

```
1
9
```

Příklad 12: Modulo a mocniny

2.3.2 Metody a atributy pro práci s čísly

Čísla (a nejen čísla, ale vlastně vše) v Pythonu jsou objekty [1]. To znamená, že mají určité atributy a metody, ke kterým můžeme přistupovat. Uděláme to tak (podobně jako v jiných programovacích jazycích) pomocí takzvané tečkové notace. Ukázat si to můžeme na příkladu komplexních čísel, která v sobě ukrývají atributy, ve kterých je uložená jejich reálná a imaginární část – `real` a `imag` (Př. 13).

```
1 print((3-2j).real)   #výpis atributu reálné části
2 print((3-2j).imag)   #výpis atributu imaginární části
```

Výstup:

```
3.0
-2.0
```

Příklad 13: Atributy čísel

K metodám v objektech přistupujeme stejně. Říkáme však, že metodu *voláme*, to znamená že k ní nepřistoupíme pouze pomocí tečky a jejího názvu, ale musíme za ní ještě připsat závorku – tím Pythonu dáváme najevo, že k ní chceme nejen přistoupit, ale zároveň ji i spustit (Př 14). V závislosti na různých metodách závorka může obsahovat i atributy, se kterými může pracovat.

```
1 print((3-2j).conjugate())    #použití metody v komplexním čísle
```

Výstup:

```
(3+2j)
```

Příklad 14: Metody čísel

2.3.3 Matematické funkce

Pokud bychom s čísly chtěli provádět pokročilejší operace, Python nám k tomu v základu nabízí několik zabudovaných funkcí. Jednou z nich je funkce `abs`, která nám vrátí absolutní hodnotu vloženého čísla. Další funkcí je `round`, která slouží k zaokrouhlování k nejbližšímu celému číslu (Př. 15).

```
1 print(abs(-8.9))        #absolutní hodnota
2 print(abs(-8))          #absolutní hodnota
3 print(round(-8.9))      #zaokrouhlení
```

Výstup:

```
8.9
8
-9
```

Příklad 15: Metody čísel

Funkcí, ovšem ne matematickou, je i námi hojně používaný příkaz `print`. Vidíme, že způsob zápisu a použití je totožný a že vlastně – stejně jako v některých předchozích příkladech – vnořujeme jednu funkci do druhé.

Při práci samozřejmě nejsme omezeni pouze na tyto dvě matematické funkce. Musíme si ovšem Python rozšířit o příslušný modul. Pro základní matematické funkce nám poslouží modul `math`, který musíme do našeho programu importovat, následně jej můžeme využít (Př. 16).

```
1 import math
2
3 print(math.sqrt(81))
4 print(math.log(3))
```

Výstup:

```
9.0
1.0986122886681098
```

Příklad 16: Modul math

Tabulka 1: Porovnávací operátory

Operátor	Význam
==	rovná se
!=	nerovná se
<	je menší než
>	je větší než
<=	je menší nebo rovno
>=	je větší nebo rovno

Modul `math` samosebou nabízí velké množství funkcí dalších. Jejich kompletní seznam lze najít v online dokumentaci na stránkách Pythonu.

2.3.4 Porovnávání a logické operace

V určitých případech budeme čísla, nebo obecně objekty, porovnávat. K tomu máme v Pythonu množství porovnávacích operátorů (Tab. 1). Porovnávání vrací objekt typu boolean, který může nabývat přesně dvou hodnot – `True` (v případě pravdivého tvrzení) a `False` (v případě nepravdivého) (Př 17).

```
1 a = 2
2 b = 3
3
4 print(a < b)      #je 2 menší než 3? Ano -> True
5 print(a > b)      #je 2 větší než 3? Ne -> False
6 print(a == b)     #je 2 rovno 3? Ne -> False
7 print(a != b)     #je 2 nerovno 3? Ano -> True
```

Výstup:

```
True
False
False
True
```

Příklad 17: Porovnávání

Naše porovnávání může samozřejmě nabývat i složitějšího charakteru. K tomu využíváme logických operátorů `or`, `and` a `not`, které nám dovolují námi použitá porovnání dále řetězit a různě kombinovat.

```
1 a = 2
2 b = 3
3
4 print(a < b or a > b)    #vypíše True, pokud je alespoň jedno z tvrzení True
5 print(a < b and a > b)   #vypíše True, pokud jsou obě tvrzení True
6 print(not a > b)         #vypíše True, pokud 2 není větší než 3
```

Výstup:

```
True
False
True
```

Příklad 18: Logické operátory

Vidíme, že operátor `or` vrátí `True`, pokud je alespoň jedno z tvrzení pravdivé. Naopak `and` vrací `True` pouze tehdy, kdy jsou všechna (v praxi to není omezeno pouze na dvě – porovnávání je možno řetězit i dál) porovnávaná tvrzení pravdivá. Operátor `not` vrátí `True`, pokud je tvrzení neplatné.

2.4 Řetězce

Řetězec je objekt typu `str` obsahující sekvenci znaků[1]. Stejně jako číslo i text můžeme ukládat do proměnných. Námi ukládaný znak nebo jeho sekvence však musí být uzavřena v jednoduchých nebo dvojitých uvozovkách (Př. 19).

```
1 a = 'Toto je prvni retezec'
2 b = "Toto je druhy retezec"
3
4 print(a)
5 print(b)
```

Výstup:

```
Toto je prvni retezec
Toto je druhy retezec
```

Příklad 19: Inicializace řetězců

Řetězce můžeme „sčítat“ a dokonce i násobit (Př. 20).

```
1 a = "Toto je "
2 b = "retezec."
3
4 print(a + b)
5 print(b * 3)
```

Výstup:

```
Toto je retezec.
retezec.retezec.retezec.
```

Příklad 20: Sčítání a násobení řetězců

Stejně tak řetězce můžeme dělit a získávat z nich podřetězce (*substring*). Pokud chceme z řetězce získat jediný znak, stačí za jeho název do hranatých závorek připsat pozici požadovaného znaku (Př. 21). Musíme si ale uvědomit, že indexování pozic začíná od čísla 0, takže pokud budeme chtít například třetí znak, jeho index bude 2. Pokud chceme získat delší podřetězec, do hranaté závorky zapíšeme rozsah podřetězce oddělený dvojtečkou (Př. 21).

```
1 a = "Retezec."
2
3 print(a[2])
4
5 print(a[1:5])
```

Výstup:

```
t
etez
```

Příklad 21: Získávání podřetězců

Je potřeba dát pozor na zadávaný rozsah, neboť levý index označuje polohu prvního znaku, nicméně pravý index označuje polohu prvního znaku, který už v podřetězci nebude.

2.4.1 Metody práci s řetězci

Stejně jako čísla v předchozí podkapitole i řetězce jsou objekty s vlastními metodami, které můžeme využít. Jmenujme například funkci `capitalize` (Př. 22), která nahradí první písmeno v řetězci písmenem velkým.

```
1 a = "retezec."
2
3 print(a.capitalize())
```

Výstup:

```
Retezec.
```

Příklad 22: `capitalize`

Kromě funkce `capitalize` řetězce obsahují i další užitečné metody. Jmenujme například `endswith(suffix)` zjišťující, zda řetězec končí zadanou příponou; `index(substring)` vracející index, na kterém se nachází hledaný podřetězec nebo `upper` vracející kopii řetězce s písmeny přepsanými na velká [1]. Kompletní seznam metod je opět k nalezení v dokumentaci k Pythonu.

2.5 List

Ti, kdo se již aktivně setkali s jiným programovacím jazykem, jistě znají datové struktury, kterým se říká *pole* nebo *array*, ve kterých je možné ukládat objekty podobně jako jednotlivé znaky v řetězcích. V Pythonu tuto strukturu zastupuje *list (seznam)*. Významným rozdílem oproti některým jiným programovacím jazykům je v Pythonu to, že jeho *list* je v sobě schopen držet objekty rozdílných datových typů – tedy do něj můžeme vedle sebe uložit například řetězec a číslo.

Vytvoření *listu* není o nic těžší než vytvoření proměnné s číslem nebo řetězcem, jen s tím rozdílem, že vkládané objekty máme všechny uzavřené v hranaté závorce a mezi sebou oddělené čárkami (Př. 23).

```
1 list1 = [2, (3 + 2j), "retezec"]
2
3 print(list1)
4
5 a = 2378
6 list2 = [a, 2, (3 + 2j), "retezec"]
7 a = 1
8 print(list2)
```

Výstup:

```
[2, (3+2j), 'retezec']
[2378, 2, (3+2j), 'retezec']
```

Příklad 23: Vytvoření listu

Vidíme, že do listu můžeme ukládat i samotné proměnné, kdy se v případě vypsání listu vypíše hodnota, kterou proměnná měla v době jeho vytvoření.

V případě, že bychom chtěli vypsat nebo jinak získat jen některou z hodnot uložených v listu, můžeme k ní přistoupit stejně jako v případě řetězce přes hranatou závorku a index (Př. 24). I zde se indexuje od nuly.

```
1 list1 = [2, (3 + 2j), "retezec"]
2
3 a = list1[2]
4
5 print(list1[2])
6 print(list1[0])
7 print(list1[1])
8 print(a)
```

Výstup:

```
retezec
2
(3+2j)
retezec
```

Příklad 24: Přístup k hodnotám v listu

Na rozdíl od řetězců však můžeme hodnoty na jednotlivých pozicích v listech i měnit (Př. 25).

```
1 list1 = [2, (3 + 2j), "retezec"]
2 a = "zmena"
3
4 print(list1)
5
6 list1[0] = a
7 list1[2] = 10
8
9 print(list1)
```

Výstup:

```
[2, (3+2j), 'retezec']
['zmena', (3+2j), 10]
```

Příklad 25: Změna hodnoty v listu

Můžeme tak učinit přímo nebo pomocí proměnné, dokonce vidíme, že nejsme vázáni na datový typ na daném indexu, takže číslo můžeme bez problémů nahradit řetězcem a naopak – platí to i pro všechny ostatní datové typy.

2.6 Cyklus for

Cyklus `for` samozřejmě funkčně odpovídá cyklům z jiných jazyků, ovšem na rozdíl od klasického zápisu v Javě nebo C# je zápis v Pythonu na první pohled jednodušší

(Př. 26) a ve své základní podobě de facto odpovídá cyklu *for each* používanému ve zmíněných jazycích.

```
1 list1 = [2, (3 + 2j), "retezec"]
2
3 for prvek in list1:
4     print(prvek)
```

Výstup:

```
2
(3+2j)
retezec
```

Příklad 26: For cyklus

Zjednodušeně se tento zápis dá přečíst jako: „Pro každý prvek v *list1* udělej to, co je za dvojtečkou“. Je to další ze způsobů, jak vypisovat objekty v seznamu nebo k nim jednotlivě přistupovat. Takto ovšem postupně přistoupíme vždy ke všem prvkům v tomto seznamu. Z jiných jazyků víme, že chceme-li pomocí cyklu přistoupit třeba jen k několika prvkům, využíváme takzvaný iterátor, pomocí kterého procházíme seznam z požadovaného místa do požadovaného místa. Tento způsob procházení seznamu pomocí cyklu už je pro zápis o něco složitější a lze k němu využít například zabudovanou funkci `range` [1] (Př. 27).

```
1 list1 = [2, (3 + 2j), "retezec"]
2
3 for prvek in range(len(list1)):
4     print(list1[prvek])
5
6 print("␣")
7
8 for prvek in range(1, len(list1)-1):
9     print(list1[prvek])
```

Výstup:

```
2
(3+2j)
retezec

(3+2j)
```

Příklad 27: Iterace ve for cyklu

2.7 Cyklus while

Z příkladů vidíme, že cyklus `for` nám vždy nabízí určený a daný počet iterací. Pokud bychom chtěli cyklus, který běží, dokud platí námi zadaná podmínka, použili

bychom cyklus `while` (Př. 28).

```
1 a = 0
2
3 while a < 5:
4     print("Probehl jsem!")
5     a = a + 1
6
7 print("Skoncil jsem, protoze a = " + str(a))
```

Výstup:

```
Probehl jsem!
Probehl jsem!
Probehl jsem!
Probehl jsem!
Probehl jsem!
Skoncil jsem, protoze a = 5
```

Příklad 28: Cyklus while

2.8 Podmínkování

Použití podmínek probíhá opět velmi podobně jako v jiných nejpoužívanějších programovacích jazycích, pouze se znovu mírně odlišným zápisem (Př. 29).

```
1  a = 0
2
3  while a < 5:
4      if a < 3:
5          print("Jsem mensi nez 3!")
6      elif a == 3:
7          print("Jsem roven 3!")
8      else:
9          print("Neplati predchozi dve podminky,"
10         " takze musim byt cislo 4!")
11     a = a + 1
12
13 print("Skoncil jsem, protoze a = " + str(a))
```

Výstup:

```
Jsem mensi nez 3!
Jsem mensi nez 3!
Jsem mensi nez 3!
Jsem roven 3!
Neplati predchozi dve podminky, takze musim byt cislo 4!
Skoncil jsem, protoze a = 5
```

Příklad 29: Podmínka

V podmínkových konstrukcích lze samozřejmě využívat a kombinovat všechny porovnávací a logické operace, které byly uvedeny v předchozích podkapitolách.

2.9 Funkce

Za funkci můžeme označit sadu seskupených a pojmenovaných příkazů, které můžeme v programu volat pod tímto názvem [1].

Python v základu přichází se sadou základních funkcí, z nichž jsme jich již mnoho použili, a tento seznam funkcí si navíc ještě můžeme obohatit o funkce další, například ze zmiňovaných doinstalovatelných knihoven. Stejně tak si v případě potřeby ale můžeme napsat i funkce vlastní.

Takový postup se volí ze dvou hlavních důvodů[1]. Prvním je zjednodušení celkového kódu v případě, že jeho některou část potřebujeme použít vícekrát. Právě v takovém případě danou část „zabalíme“ do naší funkce a pak voláme pouze ji bez toho, abychom znovu psali nebo kopírovali kus kódu, který může mít třeba jen pár řádků, ale klidně i několik desítek.

Druhý důvod je celkové zpřehlednění kódu, který lze tímto způsobem rozdělit do jednodušších, tematicky uspořádaných částí.

Vlastní funkci definujeme příkazem `def` následovaným názvem funkce a závorkou (Př. 30), ve které jsou (v případě potřeby) argumenty, které funkci při zavolání chceme předat (Př. 31).

```
1 def pozdrav():
2     print("Ahojda!")
3
4 pozdrav()
```

Výstup:

```
Ahojda!
```

Příklad 30: Funkce bez argumentů

Vidíme, že funkce po zavolání jejího jména, včetně závorky, provede příkazy, které jsme do ní napsali. Může to být například ukázaný textový výpis, ale třeba i kontrola rovnosti proměnných nebo provedení nějakého výpočtu. Ne vždy však budeme chtít, aby nám program okamžitě vypsal vypočtenou hodnotu – v takovém případě musíme výsledek provedených operací z funkce zase dostat ven. K tomu nám poslouží příkaz `return` (Př. 31).

```
1  def obsahLichobezniku(a, c, v):
2      obsah = ((a+c)/2)*v
3      return obsah
4
5
6  dolniPodstava = 10
7  horniPodstava = 5
8  vyska = 4
9
10 obsah = obsahLichobezniku(dolniPodstava, horniPodstava,
11 vyska)
12 print("Obsah␣lichobeznika␣je␣" + str(obsah) + "␣cm^2")
```

Výstup:

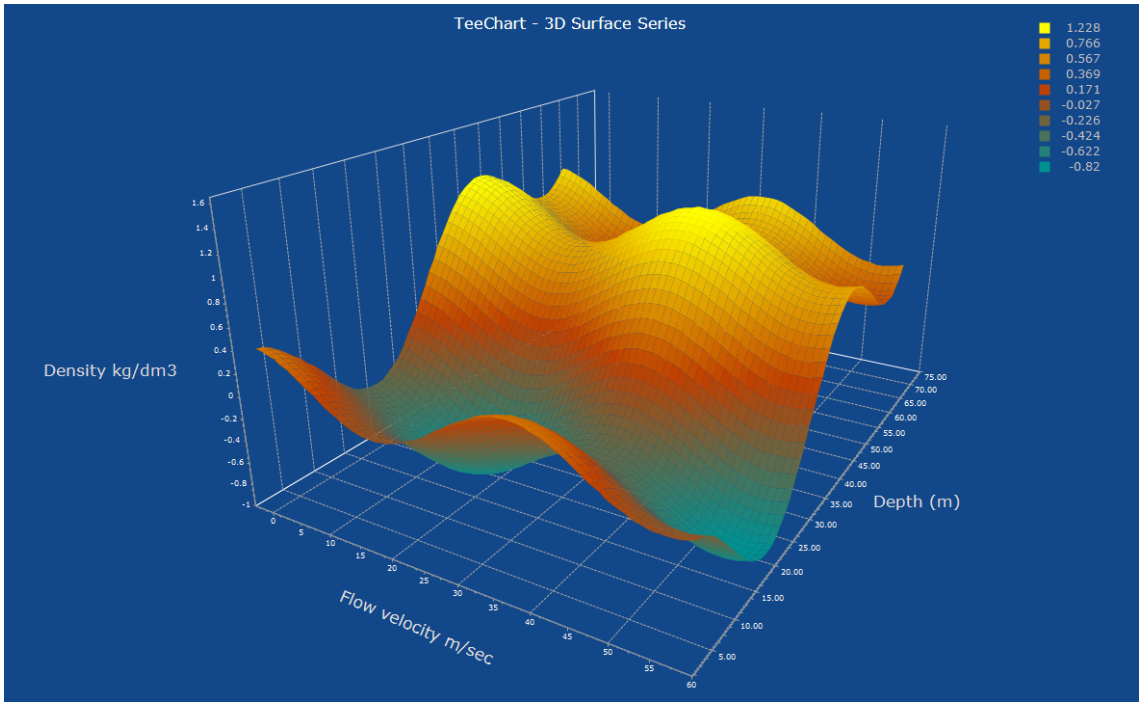
```
Obsah lichobeznika je 30.0 cm^2
```

Příklad 31: Funkce s argumenty a návratovou hodnotou

3 Vizualizace dat

Schopnost ovládat programovací jazyk, jakým je Python, nám nenabízí pouze možnosti jednoduchých či složitějších výpočtů, jaké jsme viděli v předchozích kapitolách. Důležitým aspektem, například při fyzikálních výpočtech a modelování, je i vizualizace zadaných, vypočtených nebo jinak získaných a zpracovávaných dat. Ta nám pomáhá zkoumaná data lépe a snáze pochopit, nalézt v nich důležité informace a uvědomit si jejich vzájemné vztahy.

Zřejmě nejpoužívanějším způsobem je vizualizace pomocí grafů. Ty můžeme dělit dle prostorového uspořádání na dvourozměrné, které pracují s daty ve dvou dimenzích a přiřazují je dvěma osám, a třírozměrné, zobrazující data v trojrozměrném prostoru. Výsledný graf pak může připomínat „výškovou“ mapu či povrch (Obr. 3).



Obrázek 3: 3D graf
Převzato z:
https://www.steema.com/uploads/products/3D_Surface_Chart.png

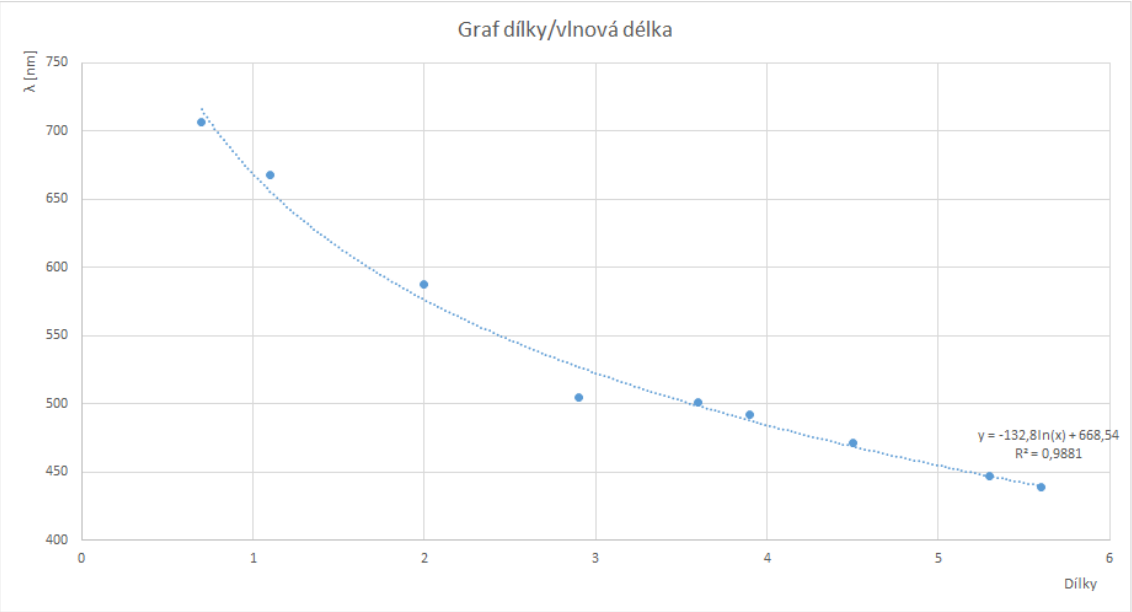
Tyto 3D grafy si však nesmíme plést s takzvanými pseudo-3D grafy [5], které sice vizuálně působí prostorově, ale data zobrazují pouze ve dvou osách. Prostorovost je zde použita ve snaze graf vizuálně zkrášlit nebo z(ne)přehlednit (Obr. 4).

Obrázek 4: Pseudo-3D graf
Převzato z:
http://st2.depositphotos.com/2121483/7368/v/450/depositphotos_73686375-stock-illustration-3d-chart-graph-bar-infographic.jpg

Dále můžeme grafy dělit podle typů, přičemž každý typ je určený pro jiný druh dat.

Zřejmě nejpoužívanějším typem grafů je graf sloupcový [6]. Tento typ grafu je používán pro porovnávání jednotlivých hodnot, kupříkladu počtu žáků v jednotlivých třídách. Nehodí se však pro data kontinuální, spojitá, zobrazující nějakou změnu, například v čase.

Pro taková data bychom použili graf bodový (Obr. 5), jinde označovaný také jako kartézský [7], či graf čárový neboli spojnicový. Tyto grafy zobrazují vztahy mezi dvěma proměnnými – jednu vyobrazovanou na ose x, druhou na ose y, přičemž výsledný graf je tvořen propojením těchto hodnot. V případě následného pospojování jednotlivých bodů grafu můžeme zpřehlednit vzájemný vztah a vývoj pozorovaných veličin. Hodnoty grafu zároveň nejsou omezeny pouze na kladná čísla s počátkem v souřadnicích [0;0], ale mohou velmi často přecházet i do záporných hodnot, a to na obou osách.



Obrázek 5: Bodový graf se spojnicí trendu

Ve fyzikálních analýzách bývá také často využívána tzv. spojnice trendu (Obr. 5), sloužící k lepšímu grafickému zobrazení trendu daného problému, zároveň může dopomoci i při odhadech následného vývoje dalších dat. Grafické zpracování dat jde obvykle ruku v ruce se zpracováním statistickým. Základní statistické pojmy si připomeneme v pozdějších kapitolách.

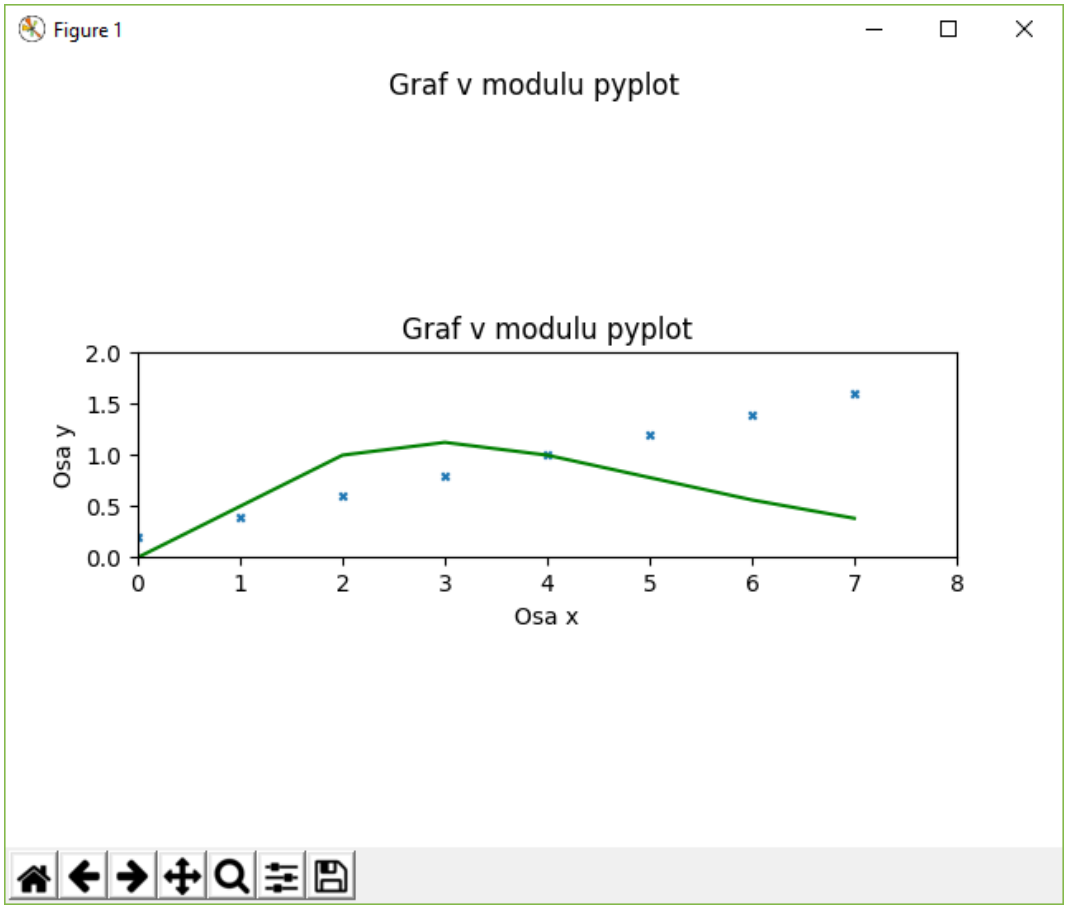
3.1 Vizualizace dat pomocí Pythonu

K tvorbě grafů v Pythonu lze použít zmíněnou knihovnu *matplotlib* spolu s jejím modulem *pyplot*. Základní principy práce s tímto modulem ukazuje následující příklad (Př. 32, Obr. 6).

```
1  from matplotlib import pyplot
2
3  X = [0, 1, 2, 3, 4, 5, 6, 7]
4  Y1 = [0.2, 0.4, 0.6, 0.8, 1.0, 1.2, 1.4, 1.6]
5  Y2 = []
6
7  for x in X:
8      Y2.append((x**2)/2**x)
9
10 pyplot.plot(X, Y2, "g")
11 pyplot.scatter(X, Y1, s=10, marker="x")
12
13 pyplot.xlabel("Osa x")
14 pyplot.ylabel("Osa y")
15 pyplot.axis("scaled")
16 pyplot.axis([0, 8, 0, 2])
17 pyplot.suptitle("Graf v modulu pyplot")
18 pyplot.title("Graf v modulu pyplot")
19
20 pyplot.show()
```

Výstup:

Příklad 32: Graf v Pythonu



Obrázek 6: Graf v Pythonu

Pro vykreslení grafu pochopitelně potřebujeme sadu dat, kterou v příkladu máme zastoupenou seznamem pro hodnoty na ose x, a dvěma seznamy pro hodnoty y. *Pyplot* nabízí množství různých druhů grafů, v příkladu máme zastoupen graf bodový, vytvořený funkcí `scatter`, a spojnicový, vytvořený funkcí `plot`. Do atributů těchto funkcí můžeme kromě samotných souřadnic uvést i atributy, kterými jednotlivé řady nastavíme – zde máme u spojnicového grafu nastavenou barvu (`"g"`) a u bodového velikost jednotlivých bodů (`s=10`) a jejich tvar (`marker="x"`). Všimnout si můžeme i dalších nastavovacích příkazů, kterými můžeme například nastavit názvy os (`xlabel("Osa x")`, `ylabel("Osa y")`), jejich poměr (`axis("scaled")`), zobrazovaný rozsah (`axis([0, 8, 0, 2])`) atd. Pro úplný přehled všech možností, které *pyplot* nabízí, je dobré navštívit stránky modulu[14], kde je vše poměrně přehledně popsáno, včetně mnoha dalších příkladů.

Pyplot také nabízí jistou interaktivnost zobrazeného grafu pomocí ovládacích prvků v dolní části okna, ve kterém se graf vykreslí. Obrázek grafu tak lze jednoduše uložit, přiblížit či oddálit, poupravit rozsahy os atd.

4 Základy statistického zpracování dat

4.1 Střední hodnota

Zde střední hodnotou budeme rozumět aritmetický průměr. Ten nám při opakovaném měření téže náhodné veličiny zprostředkovává její nejpravděpodobnější hodnotu, čímž dopomáhá ke zpřesnění provedeného měření. Jednoduchým příkladem může být obyčejné měření doby dopadu olověné kuličky z výšky jednoho metru. Měření provedeme několikrát, přičemž nejen vlastní chybou, založenou na rychlosti reakcí ruky, která spouští a vypíná stopky, naměříme pokaždé trochu jiný čas. Aritmetickým zprůměrováním všech hodnot dostaneme nejpravděpodobnější čas pádu kuličky a do určité míry tak eliminujeme nepřesnost při měření.

Aritmetický průměr měřené veličiny X značíme $\overline{x}$ a vypočítáme jej podle vzorce[11]: $\overline{x} = \frac{1}{n}(x_1 + x_2 + x_3 + ... + x_n) = \frac{1}{n}\sum_{i}^{n} x_i$.

4.2 Medián

Pokud mluvíme o středních hodnotách naměřených dat, jistě bychom měli zmínit i medián. Medián je hodnota stojící uprostřed řady hodnot, uspořádaných podle velikosti. Značkou pro medián z měřených hodnot X je $\widetilde{x}$. V případě lichého počtu hodnot je snadné medián získat – seřadíme hodnoty od nejmenší po největší a vezmeme tu, která je přímo uprostřed. V případě sudého počtu hodnot však žádnou hodnotu uprostřed nenajdeme. V takové situaci vezmeme dvě prostřední hodnoty a vypočteme z nich aritmetický průměr, který bude hledaným mediánem [8].

4.3 Modus

Termínem modus označujeme hodnotu, která se ve zpracovávané řadě vyskytuje nejčastěji, jinými slovy: má nejvyšší četnost. V případě, že soubor nabízí několik hodnot s nejvyšší četností zároveň, považujeme za modus každou z nich. V případě modu navíc můžeme pracovat i s nečíselnými hodnotami, protože na rozdíl od aritmetického průměru a mediánu nepotřebujeme znát a počítat se samotnou hodnotou, stačí nám pouze počet jejích výskytů ve zpracovávaných datech. Modus tedy můžeme uplatnit nejen na čísla, ale i například názvy měst, barvy, jména a další.

4.4 Směrodatná odchylka

Protože samotný aritmetický průměr nám nedává žádné informace o rozložení dat v souboru, počítáme často i směrodatnou odchylku. Její kvadrát odhadujeme z naměřených dat ve formě výběrového rozptylu σ^2 – ten vypočteme podle vzorce

$\sigma^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2$ [11].

Následně můžeme přikročit k výpočtu samotné směrodatné odchylky, kterou získáme odmocněním výběrového rozptylu: $\sigma = \sqrt{\sigma^2}$.

4.5 Korelace

Zatím jsme se drželi statistických výpočtů, týkajících se pouze jedné proměnné (v našem případě x). Pokud ovšem registrujeme dvě náhodné veličiny (x a y), můžeme u nich určit korelační koeficient ϱ_{xy}. To je parametr, který popisuje stupeň lineární závislosti mezi těmito veličinami. Pokud se dostáváme k hodnotám blízkým -1 a +1, víme, že se statistická závislost blíží závislosti lineární. Na druhou stranu, pokud se bude korelační koeficient blížit hodnotě 0, proměnné budou statisticky nezávislé [11].

Pro výpočet samotného korelačního koeficientu musíme nejprve vypočítat výběrovou kovariaci. Tu vypočteme podle vzorce: $\sigma_{xy} = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})$. Následně můžeme přistoupit k výpočtu korelačního koeficientu: $\varrho_{xy} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}$[11].

4.6 Příklad v Pythonu

Vypočtěme koeficient korelace pro dvojici veličin zaznamenaných v tabulce (Tab. 2).

Tabulka 2: Data pro výpočet korelačního koeficientu
Převzato z:[11]

i	1	2	3	4	5	6
$\mathbf{x_i}$	214,66	211,47	215,20	218,75	219,59	216,71
$\mathbf{y_i}$	4281,33	4318,90	4294,68	4432,63	4491,27	4333,93
i	7	8	9	10	11	
$\mathbf{x_i}$	216,50	217,21	209,88	210,95	211,65	
$\mathbf{y_i}$	4396,79	4399,75	4356,53	4338,73	4338,73	

```
1  import numpy
2  import statistics
3  import math
4
5  x = [
6      214.66,  211.47, 215.20, 218.75, 219.59, 216.71,
7      216.50, 217.21, 209.88,
8      210.95, 211.65
9      ]
10 y = [
11     4281.33, 4318.90, 4294.68, 4432.63, 4491.27,
12     4333.93, 4396.79, 4399.75,
13     4356.53, 4338.73, 4338.73
14     ]
15
16 korelacniKoeficient = numpy.corrcoef(x, y)
17
18 prumerX = numpy.average(x)
19 prumerY = numpy.average(y)
20
21 rozptylX = statistics.variance(x)
22 rozptylY = statistics.variance(y)
23
24 smerOdchylkaX = math.sqrt(rozptylX)
25 smerOdchylkaY = math.sqrt(rozptylY)
26
27
28 print("Korelacni koeficient: " + str(korelacniKoeficient))
29
30 print("Prumer x: " + str(prumerX))
31 print("Prumer y: " + str(prumerY))
32
33 print("Rozptyl x: " + str(rozptylX))
34 print("Rozptyl y: " + str(rozptylY))
35
36 print("Smerodatna odchylka x: " + str(smerOdchylkaX))
37 print("Smerodatna odchylka y: " + str(smerOdchylkaY))
```

Výstup:

```
Korelacni koeficient: [[ 1.         0.62940398]
 [ 0.62940398  1.        ]]
Prumer x: 214.779090909
Prumer y: 4362.11545455
Rozptyl x: 11.11078909090911
Rozptyl y: 3913.266267272743
Smerodatna odchylka x: 3.3332850299530508
Smerodatna odchylka y: 62.55610495605319
```

Příklad 33: Korelační koeficient

Jak vidíme (Př. 33), k výpočtu korelačního koeficientu lze využít funkce, obsažené v modulu `numpy`. Podobně jsme využili dalších funkcí ze zabudovaných modulů `math` a `statistics` pro výpočet průměrů, rozptylů a směrodatných odchylek. Tyto

hodnoty nejsou pro výpočet korelačního koeficientu nutné, jsou uvedeny pouze pro příklad. Samotný korelační koeficient je možno vypočítat jednoduše pomocí funkce `corrcoef()` z modulu `numpy`.

Funkci `str()` je v příkladu nutné použít pro možnost vypsání vypočtených číselných hodnot ve spojení s textem. Tato funkce totiž provádí *přetypování* jinak číselných nebo seznamových hodnot na textový typ `string`.

5 Algoritmy obecných numerických metod

V této kapitole si uvedeme příklady řešení různých matematických problémů za pomoci Pythonu. Pro lepší pochopení jazyka a procvičení si práce s ním budou jednotlivé příklady vyřešeny „od základu“, ačkoli Python a jeho knihovny v mnoha případech nabízí snadnější vyřešení za pomoci již hotových funkcí.

5.1 Kvadratická rovnice

Máme-li řešit kvadratickou rovnici ve tvaru $ax^2 + bx + c = 0$, kde a se nerovná nule, pro výpočet kořenů v programu nám stačí znalost koeficientů a, b a c. Z nich musíme podle vzorce $d = b^2 - 4ac$ vypočítat diskriminant, který rozhodne o tom, kam program dále větvit, tedy zda má rovnice dva různé reálné kořeny, jeden dvojnásobný reálný kořen nebo dva kořeny komplexně sdružené. V prvním případě kořeny vypočteme podle vzorce $x_{1,2} = \frac{-b \pm \sqrt{d}}{2a}$, ve druhém jako $x_1 = x_2 = \frac{-b}{2a}$ a ve třetím jako $x_{1,2} = \frac{-b}{2a} \pm i\frac{\sqrt{|d|}}{2a}$.

```
1  import math
2
3  a = 2
4  b = 4
5  c = 6
6
7  d = (b**2) - (4 * a * c)                    #vypocet diskriminantu
8
9  if d > 0:                                   #diskriminant je vetsi nez 0
10     x1 = (-b + math.sqrt(d))/(2*a)
11     x2 = (-b - math.sqrt(d))/(2*a)
12
13 if d == 0:                                  #diskriminant je roven 0
14     x1 = -b/(2*a)
15     x2 = x1
16
17 if d < 0:                                   #diskriminant je mensi nez 0
18     x1 = complex((-b/(2 * a)),((math.sqrt(abs(d)))/(2 * a)))
19     x2 = complex((-b/(2 * a)),-((math.sqrt(abs(d)))/(2 * a)))
20
21 print("x1 = " + str(x1))
22 print("x2 = " + str(x2))
```

Výstup:

```
x1 = (-1+1.4142135623730951j)
x2 = (-1-1.4142135623730951j)
```

Příklad 34: Kvadratická rovnice

Jak vidíme (Př. 34), pro každou z možných hodnot diskriminantu budou kořeny vypočteny klasickým způsobem podle výše uvedených vzorců. Před použitím

odmocniny ve vzorci nesmíme zapomenout importovat matematickou knihovnu Pythonu, pak z ní můžeme využít funkci `sqrt()`. Dále bylo potřeba využít několika přetypování – a to hlavně ve třetím „`if u`“, kde kořeny vychází jako komplexní čísla, proto je potřeba reálnou a imaginární část výsledku vypočítat zvlášť a pak je pomocí přetypovací funkce `complex()` „složit“ dohromady a uložit jako proměnné `x1` a `x2`. Další přetypování je provedeno ve vypisovací funkci `print()`, kde, aby bylo možné výsledek, který vychází jako číslo (ať už *int*, *float* nebo *complex*), vypsat spolu s textovým řetězcem (typu *str*) v uvozovkách, je potřeba samotné výsledky přetypovat taktéž do datového typu *string*, a to pomocí příkazu `str()`. Další použitou funkcí v příkladu je `abs()`, která dříve vloženou hodnotu vrátí v absolutní hodnotě.

Jak již bylo řečeno, Python nebo jeho knihovny v mnoha případech nabízí funkce, kterými si v případě nutnosti můžeme práci na řešení značně ušetřit. Pro tento případ je možné využít knihovnu NumPy, která obsahuje funkci `roots()`(Př. 35).

```
1 import numpy as np
2
3 koeficienty = [2, 4, 6]
4 print (np.roots(koeficienty))
```

Výstup:

```
[-1.+1.41421356j -1.-1.41421356j]
```

Příklad 35: Kvadratická rovnice pomocí NumPy

Do funkce `roots()` se jako argument vkládá *seznam* obsahující koeficienty počítané rovnice. Výsledek funkce vrací také jako seznam kořenů. Výhodou jejího použití je kromě urychlení a usnadnění práce i to, že zvládá vypočítávat i kořeny polynomů vyšších řádů.

5.2 Metoda půlení intervalu

Metoda půlení intervalu je jedna z nejjednodušších pro hledání kořene funkce v zadaném intervalu [11], v němž se kořen nachází. Základním principem této metody je dělení daného intervalu $< a, b >$ na dvě poloviny $< a, x >$ a $< x, b >$, přičemž $x = \frac{a+b}{2}$, tedy x je středem zadaného intervalu. Následně je potřeba spočítat součin hodnot funkce v krajních bodech jednoho z intervalů. Pokud $f(a) \cdot f(x) < 0$, víme, že se kořen nachází v intervalu $< a, x >$. Pokud výpočet vyjde větší než nula, víme, že se kořen nachází v intervalu $< x, b >$ (za předpokladu, že funkce má někde v intervalu $< a, b >$ kořen). Pokud vyjde přesně nula, kořen jsme našli v půlícím bodě x. Aby byla aproximace kořene nějak omezená a program neběžel nekonečně dlouho,

je potřeba si ještě určit, s jakou přesností chceme kořen najít. Pokud je rozdíl $x - a$ nebo $b - x$ menší, než zadaná přesnost, program (Př. 36) se ukončí, jinak se vše opakuje znovu na intervalu, ve kterém je hledaný kořen.

```
1  funkce = "x**3 - 2*x - 5"
2  a = 2
3  b = 3
4  presnost = 0.0001
5
6  def vypocetFce(x):
7      fce = eval(funkce)
8      return fce
9
10 if (vypocetFce(a) * vypocetFce(b)) < 0:
11     while True:
12         x = (a + b)/2
13         fce = vypocetFce(x)
14
15         if fce == 0 or b-a < presnost:
16             print(x)
17             break
18         elif fce*vypocetFce(a) < 0:
19             b = x
20         else:
21             a = x
22 else:
23     print("Funkce nemá v zadaném intervalu žádný kořen.")
```

Výstup:

```
2.09454345703125
```

Příklad 36: Metoda půlení intervalu

V příkladech 34 a 35 jsme rovnici zadávali ve formě jejích koeficientů, uložených do jednotlivých proměnných nebo do seznamu. V příkladu 36 vidíme další možnost zápisu, a to v podobě řetězce. Dále jsme museli určit i interval, na kterém chceme kořeny aproximovat a samozřejmě již zmíněnou přesnost.

Následuje definování vlastní funkce. Ta slouží pro výpočet hodnoty naší matematické funkce. Kvůli tomu je použita vlastní funkce Pythonu `eval()`, která je schopná matematicky vyhodnotit operace zadané v podobě řetězce a dokonce během toho dosadit za proměnnou (v našem případě za x). Návratová hodnota nám vrací číslo, vyjadřující vypočtenou hodnotu vložené funkce.

Pro opakování algoritmu výpočtu můžeme použít cyklus `while`. V předchozích kapitolách bylo řečeno, že tento cyklus se opakuje do té doby, dokud není porušena podmínka uvedená v jeho zápise, respektive dokud je uvedená podmínka vyhodnocována jako `True`. My pro určité potřeby můžeme tento cyklus poněkud ošálit a rovnou mu hodnotu `True` do zápisu dát, čímž způsobíme to, že by mohl běžet do-

nekonečna. Abychom se ovšem nekonečnému zacyklení vyhnuli, ve vhodnou chvíli – zde v případě dosažení zadané přesnosti aproximace kořene na daném intervalu či přímo nalezení kořene – cyklus přerušíme příkazem `break`. Pokud jsme ovšem kořen se zadanou přesností nenašli, předefinujeme krajní body intervalu pro další průběh cyklu tak, abychom v dalším kroku dělili ten interval, ve kterém se kořen nachází, a vše znovu opakujeme.

5.3 Metoda sečen

Další z možných metod, kterými lze určit kořen funkce, je metoda sečen, vycházející z metody regula-falsi. Oproti metodě půlení intervalu konverguje rychleji, ovšem může se stát, že při špatné počáteční aproximaci kořene nebude konvergovat vůbec [11]. Při ní nedělíme zadaný interval na poloviny a neurčujeme, ve které z nich se nachází průsečík s osou x, ale křivku funkce mezi body x_1 a x_2 nahrazujeme sečnou procházející body x_1 a x_2. Její průsečík s osou nazvěme x_3. Tento bod použijeme v další iteraci výpočtu, kdy vedeme další sečnu mezi body x_2 a x_3. S každou další iterací se nám průsečík osy x blíží k samotnému kořeni funkce. Vzorec výpočtu, který následně použijeme pro výpočetní algoritmus, bude vypadat takto[11]:

$$x_3 = x_2 - \frac{(x_2 - x_1)f(x_2)}{f(x_2) - f(x_1)}$$

```
1  from matplotlib import pyplot
2  from matplotlib import lines
3  import numpy
4
5  funkce = "x**3 - 2*x - 5"
6  x1 = 2
7  x2 = 3
8  presnost = 0.0001
9
10 def vypocetFce(funkce, x):
11     fce = eval(funkce)
12     return fce
13
14 while True:
15     x3 = x2 - ((x2 - x1) * vypocetFce(funkce, x2)) / (
16     vypocetFce(funkce, x2) - vypocetFce(funkce, x1))
17
18     if abs((x3 - x2)) < presnost:
19         break
20
21     x1 = x2
22     x2 = x3
23
24 print(x3)
```

Výstup:

```
2.094551481227599
```

Příklad 37: Metoda sečen

V kódu (Př. 37) použijeme opět vlastní funkci `vypocetFce(funkce, x)`, která nám pomocí zabudované funkce `eval()` vyhodnotí výraz se zadaným x. K opakování výpočetního algoritmu znovu využijeme cyklus `while`, který příkazem `break` přerušíme, jakmile u výsledku dosáhneme požadované přesnosti.

5.4 Výpočet druhé odmocniny

U výpočtu druhé odmocniny můžeme využít Newtonova iteračního vzorce [11], který pro následné zalgoritmování můžeme zapsat takto: $x_2 = \frac{1}{2}(x_1 + \frac{a}{x_1})$, kde x_1 je vstupní mezivýsledek či počáteční odhad, x_2 je výstupní mezivýsledek a a je číslo, z něhož odmocninu počítáme (Př. 38).

```
1  a = 3
2  presnost = 0.00001
3  x2 = 1
4
5  while True:
6      x1 = x2
7      x2 = (a/x1 + x1)/2
8      if abs(x1-x2) <= presnost:
9          break
10
11 print(x2)
```

Výstup:

```
1.7320508075688772
```

Příklad 38: Druhá odmocnina

Výsledek opět hledáme s určitou přesností. Jakmile jí dosáhneme, což zjišťujeme opět v podmínce `if`, celý zdánlivě nekonečný cyklus `while` ukončíme příkazem `break`.

5.5 Aproximace přímkou

Často se setkáváme s úkolem proložit naměřenými daty x_i, y_i přímku a zároveň určit její rovnici $F(x) = a \cdot x + b$. Z odvození [11] získáme dvě rovnice pro koeficienty a a b:

$$b = \frac{\sum_{i=1}^{n} x_i^2 \sum_{i=1}^{n} y_i - \sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i \cdot x_i}{n \sum_{i=1}^{n} x_i^2 - \sum_{i=1}^{n} x_i \cdot \sum_{i=1}^{n} x_i}$$

$$a = \frac{n \sum_{i=1}^{n} y_i x_i - \sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{n \sum_{i=1}^{n} x_i^2 - \sum_{i=1}^{n} x_i \cdot \sum_{i=1}^{n} x_i}.$$

Pro danou aproximaci určíme střední lineární chybu:

$$\xi_i = [\sum_{i=1}^{n} (y_i - a \cdot x_i - b)]/(n-1),$$

a střední kvadratickou chybu:

$$\xi_K = [\sum_{i=1}^{n} (y_i - a \cdot x_i - b)^2]/(n-1).$$

Nyní zbývá tyto rovnice převést ve funkční algoritmus (Př. 39).

```
data = [[4, 8.1],
        [5, 8.9],
        [6, 10.2],
        [7, 11.01],
        [8, 11.991],
        [9, 13.2],
        [10, 14.11]]

a = 0           # koeficient a
b = 0           # koeficient b
sx2 = 0         # suma x^2
sy = 0          # suma y
sxy = 0         # suma xy
sx = 0          # suma x
slch = 0        # stredni linearni odchylka
skch = 0        # stredni kvadraticka odchylka


for prvek in data:
    sx2 += prvek[0]**2
    sy += prvek[1]
    sxy += prvek[1] * prvek [0]
    sx += prvek[0]

b = (sx2 * sy - sx * sxy)/(len(data) * sx2 - sx * sx)
a = (len(data) * sxy - sx * sy)/(len(data) * sx2 - sx * sx)

for prvek in data:
    slch += (prvek[1] - a * prvek[0] - b)
    skch += (prvek[1] - a * prvek[0] - b)**2

slch = slch/(len(data) - 1)
skch = skch/(len(data)- 1)

print("b = " + str(b))
print("a = " + str(a))
print("stredni linearni chyba = " + str(slch))
print("stredni kvadraticka chyba = " + str(skch))
```

Výstup:

```
b = 3.9677500000000006
a = 1.0150357142857125
stredni linearni chyba = 1.3618735768735254e-14
stredni kvadraticka chyba = 0.011441327380952314
```

Příklad 39: Aproximace přímkou

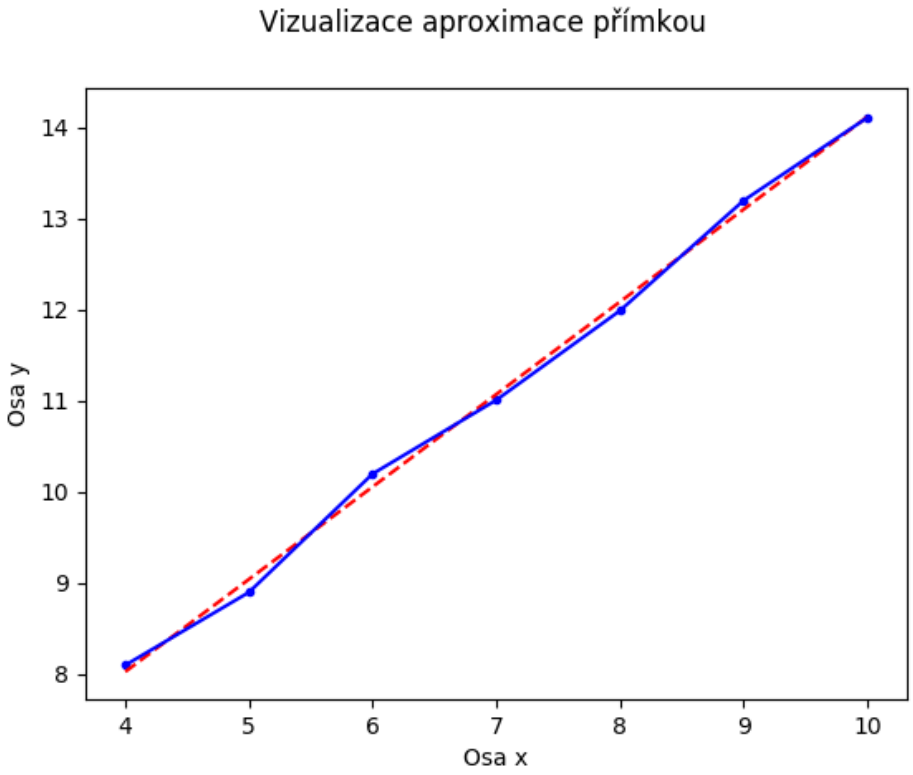
Naměřená data bylo v tomto případě vhodné zadat do dvourozměrného *listu* (*seznamu*), čili vlastně do matice. V levém sloupci jsou naměřené hodnoty x, v pravém y. Samotná inicializace listu může být zapsána v jednom řádku, v příkladu je sloupcový zápis proveden kvůli přehlednosti. Dále je potřeba definovat si proměnné, se kterými budeme pracovat. Pak už následuje samotný výpočet. V prvním cyklu nejprve vypočteme všechny sumy, které se ve vzorcích nachází, poté z nich určíme

oba koeficienty funkce.

Druhý cyklus slouží pro výpočet sum, které se následně použijí k dopočítání středních chyb. Výraz `len(data)` zde vyjadřuje proměnou n z původních vzorců, což je počet jednotlivých měření, z nichž se sumy počítají, v případě programu je to zároveň délka listu `data`, v němž jsou hodnoty uložené. Protože je to tzv. list listů, tedy `list`, jehož jednotlivé položky jsou opět typu `list`, vrátí nám příkaz `len(data)` hodnotu 7, která nám říká, že v listu *data* se nachází 7 dalších listů (z nichž každý už by měl délku 2), čili při dodržení navrženého zápisu kódu to odpovídá sedmi řádkům tabulky.

5.5.1 Vykreslení grafu funkce a přímky trendu

Pomocí dříve zmiňovaných modulů Pythonu můžeme výše zmíněná data zobrazit ve formě grafu, včetně aproximační přímky (Obr.7). Níže uvedený kód je možno po lehkých úpravách buď to vložit přímo do výše uvedeného kódu, nebo jej můžeme zapsat do samostatného souboru (Př. 40), který by měl být uložený ve stejné složce, jako kód pro aproximaci přímkou. U obou souborů nezapomeňme na koncovku .py. Názvy souborů lze volit libovolně (zde *aproximacePrimkou.py* a *aproximacePrimkou_Graf.py*).



Obrázek 7: Aproximace přímkou

```
1  import aproximacePrimkou
2  from matplotlib import pyplot
3  import numpy
4
5  x = []
6  y = []
7  y_namerene = []
8  i = 0
9  a = aproximacePrimkou.a
10 b = aproximacePrimkou.b
11
12 for prvek in aproximacePrimkou.data:
13     y_namerene.append(prvek[1])
14     x.append(prvek[0])
15     y.append(numpy.polyval([a, b], x[i]))
16     i += 1
17
18 pyplot.xlabel("Osa x")
19 pyplot.ylabel("Osa y")
20 pyplot.suptitle("Vizualizace aproximace přímkou")
21 pyplot.plot(x, y, "r", linestyle="dashed")
22 pyplot.scatter(x, y_namerene, c="b", marker=".")
23 pyplot.plot(x,y_namerene, "b")
24 pyplot.show()
```

Výstup:

```
b = 3.9677500000000006
a = 1.0150357142857125
stredni linearni chyba = 1.3618735768735254e-14
stredni kvadraticka chyba = 0.011441327380952314
```

Příklad 40: Aproximace přímkou

Protože jsme kód pro vykreslení grafu uložili do samostatného souboru, potřebujeme Pythonu nějak říct, aby využíval i data z původního souboru, kde se provádí samotné výpočty metody. Proto použijeme již známý `import` s tím, že tentokrát nebudeme importovat knihovnu Pythonu, ale náš původní soubor *aproximacePrimkou.py* (při importu však již bez koncovky). Pro samotné vykreslení grafu je potřeba z knihovny *matplotlib* importovat modul *pyplot* a pro výpočty s polynomy modul *numpy*. K vykreslení grafu potřebujeme přístup k některým hodnotám z původního souboru. K těm přistupujeme přes tečkovou notaci. V příkladu do proměných a a b takto uložíme hodnoty vypočtené v programu *aproximacePrimkou*. Tyto hodnoty jsou v počátku ovšem nastavené na 0 a až ke konci programu dojde k jejich výpočtu a uložení. Z toho plyne důležitý poznatek, že během importování daného souboru dojde rovnou k jeho spuštění a vykonání programu, který je v něm napsaný. Proto po spuštění programu, ve kterém je kód pouze pro vykreslení grafu, dojde na výstupu i k výpisu výsledků z programu původního, neboť i ten se spustí a provede.

Podobně jako ke koeficientům a a b je nutno přistoupit i k seznamu, v němž

jsou uložena naměřená data. Tento seznam procházíme pomocí cyklu `for` a data ukládáme do dvojice seznamů oddělených. Jako poslední krok v cyklu je potřeba vypočítat hodnoty y aproximační přímky za pomoci funkce `polyval()` z modulu `numpy`, jejímiž vstupními argumenty jsou koeficienty polynomu a prvek x, který je uložen v seznamu tvořeném o řádek výše. Z tohoto seznamu je vždy brán ten prvek x, který svou polohou v něm odpovídá indexu i, navyšovaném v poslední části cyklu.

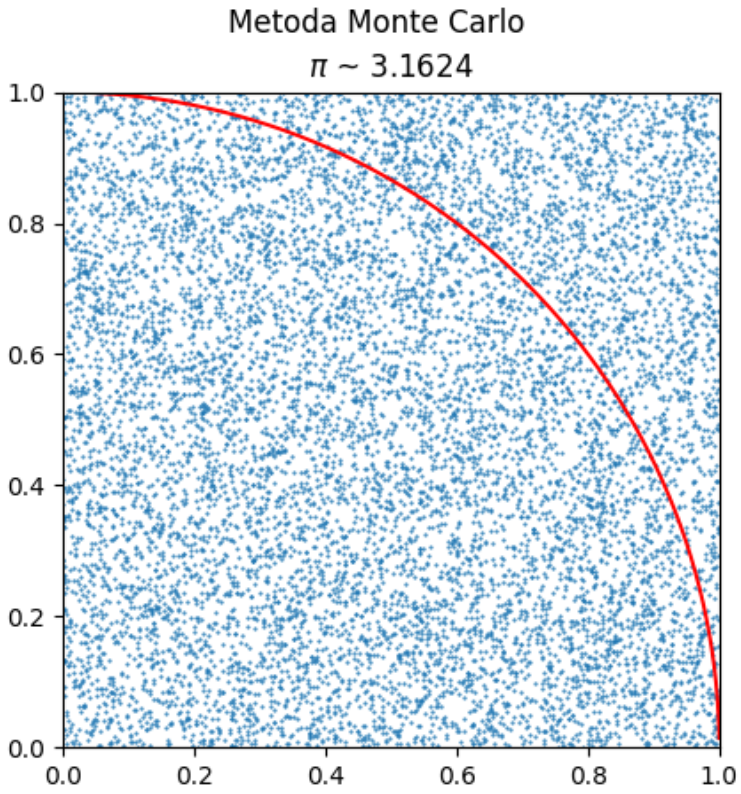
Následně už jen stačí vytvořit graf pomocí funkcí modulu `pyplot`. Funkce `xlabel` a `ylabel` přidávají popisky k osám grafu, `suptile` nastavuje popisek celého grafu, funkce `plot` a `scatter` vykreslují hodnoty do grafu. `Plot` používáme pro vykreslení grafu spojnicového, `scatter` pro vykreslení grafu bodového. Každá z těchto funkcí má kromě vykreslovaných dat ještě i argumenty pro nastavení typu čáry (`linestyle="dashed"`) nebo bodu (`marker="."`), barvy(”r”, c=”b”,...) atd. Každá z funkcí však může využívat trošku jiný zápis – což lze vidět například u rozdílného nastavování barvy – a co funguje v jedné funkci, nemusí fungovat v jiné. Proto je velmi výhodné používat dokumentaci k modulu `pyplot`, kde je vše poměrně detailně popsané.

Příkaz `show()` už jen zobrazí námi naprogramovaný obrázek.

5.6 Metoda Monte Carlo

Metodu Monte Carlo lze využít například k výpočtu hodnoty čísla π (Př. 41). Základní myšlenka je založena na geometrické pravděpodobnosti (Obr. 8), kdy si určíme čtvercovou plochu o rozměrech 1×1 a obsahu $S_1 = 1$, do níž bude vepsaná jednotková čtvrtkružnice o obsahu $S_2 = \pi/4$. Do plochy necháme vygenerovat množství bodů určených pseudonáhodnými čísly (algoritmus, generující „náhodná“ čísla, je ve skutečnosti kauzální). Tyto body nám rovnoměrně pokryjí plochu čtverce. Následně určíme, kolik z nich se nachází uvnitř čtvrtkruhu. Poměr počtu bodů uvnitř čtvrtkruhu vůči celkovému počtu bodů nám dá přibližnou hodnotu čtvrtiny čísla π. Vycházíme z toho, že v případě rovnoměrného rozdělení bodů ve čtverci je pravděpodobnost výskytu bodu uvnitř čtvrtkruhu dána vztahem $P = \frac{S_2}{S_1}$, tedy π je (přibližně) 4 krát n/N, kde N je celkový počet bodů ve čtverci a n je jejich počet uvnitř čtvrtkruhu:

$$\pi = 4 \cdot \frac{n}{N}$$



Obrázek 8: Metoda Monte Carlo

```
1  import numpy as np
2  from matplotlib import pyplot
3  import math
4  import random
5
6  i = 0
7  pocetPodKrivkou = 0
8  pocetCelkove = 10000
9
10 x = np.array(range(0, pocetCelkove, 1))
11 x = x/pocetCelkove
12 y = [math.sqrt(1-i**2) for i in x]
13
14 xBody = [random.random() for j in range(pocetCelkove)]
15 yBody = [random.random() for j in range(pocetCelkove)]
16 for ypsilon in yBody:
17     if ypsilon <= y[i]:
18         pocetPodKrivkou += 1
19     i += 1
20
21 pi = 4 * (pocetPodKrivkou/pocetCelkove)
22 print(pi)
23
24 pyplot.plot(x, y, "r")
25 pyplot.scatter(xBody, yBody, s=1, marker=".")
26 pyplot.axis("scaled")
27 pyplot.axis([0, 1, 0, 1])
28 pyplot.suptitle("Metoda␣Monte␣Carlo")
29 pyplot.title("$\pi$␣~␣" + str(pi))
30 pyplot.show()
```

Výstup:

```
3.1304
```

Příklad 41: Metoda Monte Carlo

Nutno poznamenat, že při každém spuštění programu dostaneme trochu jiný výsledek, což je dáno právě tím, že výpočet provádíme na základě odhadu hodnoty pravděpodobnosti P z konečného počtu vygenerovaných bodů. Ovšem čím více bodů necháme vygenerovat, tím přesnější hodnotu π získáme, což je možno vyzkoušet změnou proměnné `pocetCelkove`.

K naplnění seznamu x-ových souřadnic pro vykreslení čtvrtkružnice lze použít funkci `range()`, jejímiž argumenty jsou počátek číselné řady, konec a krok mezi jednotlivými čísly. Tato funkce ovšem dokáže generovat pouze čísla typu *integer*, tedy celá čísla, my však potřebujeme čísla desetinná v rozsahu (0; 1). Z tohoto důvodu nemůžeme využít klasický základní *seznam*, ale objekt typu *array* z modulu numpy, který nabízí daleko více možností pro práci – v našem případě možnost jednoduše každé číslo v něm uložené vydělit počtem těchto čísel, čímž získáme hodnoty v požadovaném rozsahu (0; 1). Pro získání souřadnic bodů čtvrtkružnice už jen stačí

dopočítat hodnoty příslušných souřadnic y ze vzorce $y = \sqrt{1 - x^2}$ (v kódu je x nahrazeno i, neboť proměnnou x již používáme). Protože je potřeba výpočet provést pro tolik hodnot y, kolik je hodnot x, zapíšeme jej do cyklu `for`. V příkladu je uveden jiný druh zápisu tohoto cyklu, funkčně je ovšem naprosto totožný zápisu původnímu.

Následuje generování náhodných bodů pomocí funkce `random()` z modulu *random*, která vrací pseudonáhodné hodnoty v rozsahu (0; 1), a zjištění, zda se nachází pod křivkou kružnice. Pokud ano, zvýší se hodnota proměnné `pocetPodKrivkou` o 1. Pak už jen stačí vypočítat hodnotu π.

Samozřejmě není od věci si vykreslit i graf. Oproti předchozímu příkladu s grafem jsme zde navíc pomocí funkce `axis()` zadali poměr a rozsah os a ve funkci `scatter()` jsme argumentem `s=1` nastavili velikost vykreslení bodů.

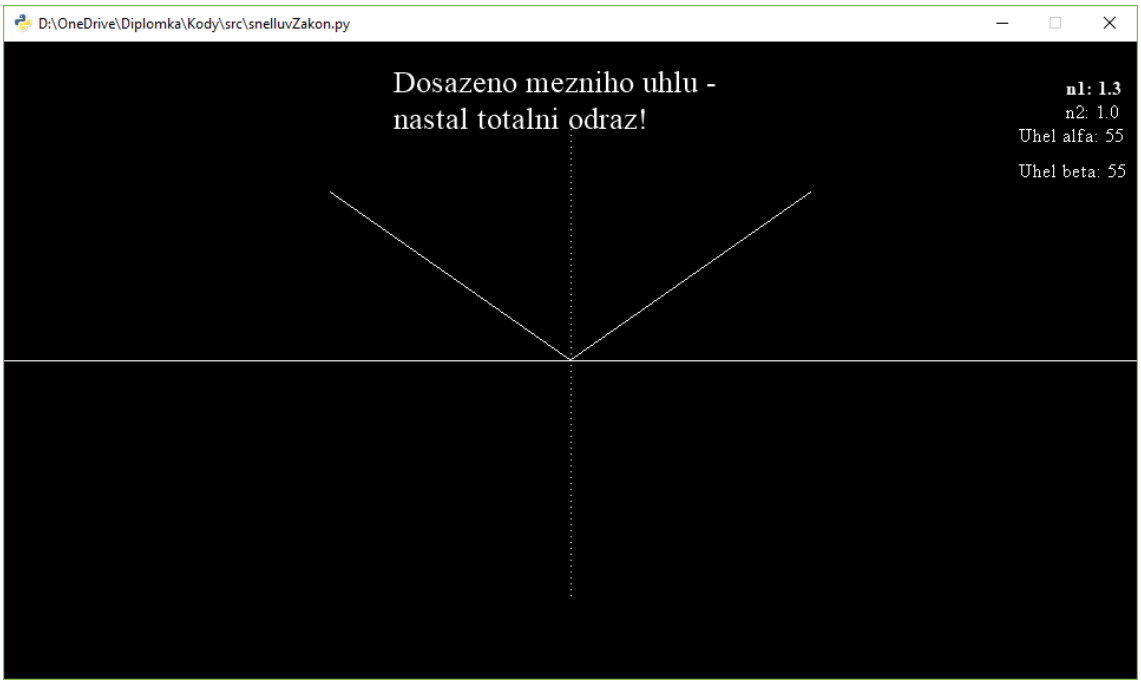
6 Tvorba interaktivní aplikace

Kromě „klasických“ matematických a fyzikálních výpočtů lze Python využít k tvorbě méně či více složitých interaktivních programů, simulujících a znázorňujících rozličné fyzikální jevy. Ty je možno následně využít například i ve výuce samotné fyziky jako doplnění daného tématu či nahrazení reálného experimentu, který v dané situaci není možné provést.

6.1 Snellův zákon lomu

Možnosti Pythonu v této oblasti si budeme demonstrovat na simulaci Snellova zákona lomu. Naším cílem bude jeho jednoduché zobrazení, viz Obr. 9 a Obr. 10.

Obrázek 9: Snellův zákon lomu – lom ke kolmici



Obrázek 10: Snellův zákon lomu – totální odraz

Z teorie nám pro jeho znázornění postačí vzorec: $n_1 \cdot \sin\alpha = n_2 \cdot \sin\beta$ a z něj vyjádřený úhel: $\beta = \arcsin(\frac{n_1 \cdot sin\alpha}{n_2})$, kde n_1 je index lomu prostředí, ze kterého se paprsek šíří, n_2 je index lomu prostředí, do kterého se paprsek šíří, a úhly α a β označují úhly měřené vůči kolmici na rozhraní těchto dvou prostředí.

V následující části (Př. 42) uvádíme kód programu.

```
1  import pyglet
2  from pyglet.window import key
3  import ctypes
4  import math
5
6  user32 = ctypes.windll.user32
7  screensize = user32.GetSystemMetrics(0), user32.GetSystemMetrics(1)
8
9  i = 0
10 delkaPaprsku = 250
11 n1 = 1.0
12 n2 = 1.0
13 alfa = math.radians(45)
14 beta = 0
15
16 ui = ["n1: ", "n2: ", "Uhel alfa: ", "Uhel beta: "]
17 hodnoty = [n1, n2, alfa]
18 totalniOdraz = False
19
20 window = pyglet.window.Window(screensize[0]//2, screensize[1]//2)
21
22 label = pyglet.text.Label(ui[0],
23                           font_name='Times New Roman',
24                           font_size=12,
25                           x=window.width, y=window.height-30,
26                           anchor_x='right', anchor_y='top',
27                           multiline = True, width = 60)
28
29 label2 = pyglet.text.Label(ui[1],
30                           font_name='Times New Roman',
31                           font_size=12,
32                           x=window.width, y=window.height-50,
33                           anchor_x='right', anchor_y='top',
34                           multiline = True, width = 60)
35
36 label3 = pyglet.text.Label(ui[2],
37                           font_name='Times New Roman',
38                           font_size=12,
39                           x=window.width, y=window.height-70,
40                           anchor_x='right', anchor_y='top',
41                           multiline = True,
42                           width = 100)
43
44 label4 = pyglet.text.Label(ui[3],
45                           font_name='Times New Roman',
46                           font_size=12,
47                           x=window.width, y=window.height-100,
48                           anchor_x='right', anchor_y='top',
49                           multiline = True,
50                           width = 100)
51
52 label5 = pyglet.text.Label("",
53                           font_name='Times New Roman',
54                           font_size=20,
55                           x=window.width/2, y=window.height-20,
```

```
56                              anchor_x='center', anchor_y='top',
57                              multiline = True,
58                              width = 300)
59
60  labels = [label, label2, label3, label4, label5]
61
62
63  def update(time):
64      """Funkce starajici se o aktualizaci stavu
65      programu.
66      """
67
68      global hodnoty
69      global alfa
70      global i
71      global totalniOdraz
72      global beta
73
74      if hodnoty[2] > math.pi/2:
75          hodnoty[2] = math.pi/2
76
77      if hodnoty[2] < 0:
78          hodnoty[2] = 0
79
80      n1 = hodnoty[0]
81      n2 = hodnoty[1]
82      alfa = hodnoty[2]
83
84      labels[0].text = ui[0] + str(round(hodnoty[0],1))
85      labels[1].text = ui[1] + str(round(hodnoty[1],1))
86      labels[2].text = ui[2] + str(round(math.degrees(hodnoty[2])))
87
88      labels[i].bold = True
89      try:
90          labels[i-1].bold = False
91          labels[i+1].bold = False
92      except:
93          pass
94
95
96      if n2<n1:
97          mezniUhel = math.asin(n2/n1)
98      else:
99          mezniUhel = math.radians(90)
100
101     if alfa >= mezniUhel:
102         beta = alfa
103         totalniOdraz = True
104         labels[4].text = "Dosazeno␣mezniho␣uhlu␣-␣nastal␣totalni␣odraz!
                "
105     else:
106         try:
107             sinBeta = (n1*math.sin(alfa))/n2
108             beta = math.asin(sinBeta)
109             totalniOdraz = False
110             labels[4].text = ""
```

```
111             except:
112                 pass
113
114         labels[3].text = ui[3] + str(round(math.degrees(beta)))
115
116
117 def zmenHodnotu(oKolik):
118         """Funkce menici hodnoty promennych po stisku
119         ovladaciho prvku"""
120
121         global n1
122         global n2
123         global alfa
124         global i
125
126         if i == 0 or i == 1:
127             hodnoty[i] = hodnoty[i] + oKolik/10
128         elif i == 2:
129             if math.pi/2 >= alfa >= 0:
130                 hodnoty[i] = hodnoty[i] + math.radians(oKolik)
131
132
133 def pressed(symbol, modifiers):
134         """Funkce starajici se o ovladani programu
135         pomoci klavesnice.
136         """
137
138         global hodnoty
139         global i
140
141         if symbol == key.DOWN and i < len(labels)-1:
142             i = i + 1
143
144         if symbol == key.UP and i > 0:
145             i = i - 1
146
147         if symbol == key.LEFT and modifiers == 17:
148             zmenHodnotu(-1)
149         elif symbol == key.LEFT:
150             zmenHodnotu(-5)
151
152         if symbol == key.RIGHT and modifiers == 17:
153             zmenHodnotu(1)
154         elif symbol == key.RIGHT:
155             zmenHodnotu(5)
156
157
158 def draw():
159         """Funkce, ktera se stara o vykreslovani okna
160         """
161
162
163         global alfa
164         global totalniOdraz
165         global beta
```

```
167     window.clear()
168
169     for label in labels:
170         label.draw()
171
172     pyglet.graphics.draw(2, pyglet.gl.GL_LINES,
173             ("v2f", (0, window.height/2, window.width,
174             window.height/2)))
175
176     kolmice = list(range(window.height//2 -200,
177     window.height//2 + 200, 5))
178     for y in kolmice:
179         pyglet.graphics.draw(1, pyglet.gl.GL_POINTS,
180                 ("v2f", (window.width/2, float(y))))
181
182     pyglet.graphics.draw(2, pyglet.gl.GL_LINES,
183             ("v2f", (window.width/2, window.height/2,
184             window.width/2 - delkaPaprsku * math.sin(alfa),
185             window.height/2 + delkaPaprsku * math.cos(alfa))))
186
187     if not totalniOdraz:
188         pyglet.graphics.draw(2, pyglet.gl.GL_LINES,
189             ("v2f", (window.width/2, window.height/2,
190             window.width/2 + delkaPaprsku * math.sin(beta),
191             window.height/2 - delkaPaprsku * math.cos((beta)))))
192     elif totalniOdraz:
193         pyglet.graphics.draw(2, pyglet.gl.GL_LINES,
194             ("v2f", (window.width/2, window.height/2,
195             window.width/2 + delkaPaprsku * math.sin(beta),
196             window.height/2 + delkaPaprsku * math.cos((beta)))))
197
198
199 window.push_handlers(on_key_press=pressed)
200
201 def main():
202     """Funkce, jejimz zavolanim na konci kodu
203     spustime samotny program.
204     """
205
206     @window.event
207     def on_draw():
208         draw()
209     pyglet.clock.schedule_interval(update, 1/120.0)
210     pyglet.app.run()
211
212 main()
```

Výstup:

Příklad 42: Snellův zákon lomu

Základem pro tvorbu interaktivní grafické aplikace je výběr knihovny, na níž budeme stavět. Mimo námi použité knihovny *Pyglet* můžeme zmínit ještě například knihovnu *PyGame*. Tyto knihovny v sobě zahrnují moduly pro práci s grafickým

rozhraním, oknem, vstupem z klávesnice a myši atd. Pyglet je před importováním potřeba nainstalovat, což zahrnuje stejný postup, jako v případě instalace předchozích modulů, případně je na jeho stránkách [12], kromě detailní dokumentace, popsaný i postup instalace.

Kromě knihovny *Pyglet* a některých modulů z této knihovny je výhodné ještě importovat modul *ctypes* (není jej potřeba instalovat – je součástí základní instalace Pythonu), který využijeme k získání rozměrů obrazovky, na které budeme program spouštět (proměnné `user32` a `screensize`) a následnému určení velikosti okna programu, jež inicializujeme do proměnně `window` (v příkladu jsou pro některé proměnné nebo funkce použity standardní zavedené anglické názvy; obecně je v programech doporučeno používat pouze anglické názvy, zde je částečně použito i českých názvů pro lepší srozumitelnost).

Pro začátek musíme zavést některé proměnné, které nám budou určovat vlastnosti prostředí a směr a délku zobrazovaného paprsku. Seznam `ui` a pygletovské objekty typu `Label` (Label v grafickém okně vykreslí text) si zavedeme pro jednoduchou správu uživatelského rozhraní, pomocí kterého budeme aplikaci ovládat. Naším cílem bude možnost měnit v programu oba indexy lomu a úhel dopadu světelného paprsku. Podle těchto proměnných se pak vypočte a zobrazí lom paprsku.

Objekty typu `Label` opět pro snazší správu umístíme do seznamu s názvem `labels`.

Po zavedení základních proměnných a objektů, které budeme v programu využívat, následují funkce, které se starají o běh programu.

6.1.1 Funkce `update()`

Jako první si popíšeme funkci `update()`. Je to funkce, která se po spuštění programu neustále opakuje, takže v zásadě zajišťuje běh programu a hlavně aktualizaci jeho stavu. To znamená, že je vhodné sem umístit kód, který je potřeba často přepočítávat, či z něj volat jiné funkce, ve kterých k přepočtům dochází. Pro správnou funkčnost je zde potřeba znovu zavést proměnné, které budeme používat, a opatřit je klíčovým slovem `global`. To je nutné z toho důvodu, že se nacházíme uvnitř funkce a proměnné zde vytvořené patří pouze této funkci a nejsou vidět z jejího vnějšku, takže k nim program nemůže přistupovat. Tím, že je nastavíme jako globální, programu říkáme, že s těmito proměnnými chceme pracovat v celém programu, čímž je vlastně „sloučíme“ se stejně pojmenovanými proměnnými vně funkce, a tedy pokud je na nějakém místě programu změníme, změníme je v celém programu. Takto je (mimo jiné) možno předávat proměnné mezi funkcemi.

Následně je nutno zkontrolovat velikost úhlu α, uloženého v seznamu `hodnoty`. To je vhodné udělat kvůli situacím, kdy by uživatel zadal úhel větší než 90° nebo naopak menší než 0°, což by neodpovídalo standardnímu znázornění zákona lomu a navíc

by mohl nastat problém při výpočtech. Touto kontrolou tak zamezíme nechtěným situacím, které by mohly vyústit v chybu.

Další část kódu slouží k získání zbylých hodnot. Poté aktualizujeme text uvnitř „labelů“ tak, aby odpovídal aktuálním hodnotám indexů lomu a úhlu α. V této části ovšem musíme jednotlivé hodnoty převést na text pomocí funkce `str()` a navíc je vhodné číslo ještě předtím zaokrouhlit, neboť by nám program často zobrazoval hodnoty na zbytečně mnoho desetinných míst.

V dalším kroku nastavujeme uživatelské rozhraní tak, aby byl vždy tučným písmem označen řádek, v němž chceme měnit hodnoty. Vybraný řádek nám reprezentuje proměnná `i` (viz další podkapitoly).

Následuje výpočet mezního úhlu pro totální odraz pro daná prostředí a poté konečně výpočet samotného úhlu β. Zde použijeme nová klíčová slova `try` a `except`, která nám zajistí, že pokud by někde v bloku `try` došlo k chybě (například k dělení nulou), program místo ukončení a vypsání chybové hlášky vynechá neproveditelný příkaz a vykoná kód v bloku `exception` – v tomto případě neudělá nic, což mu zadáme slovem `pass`.

Nakonec zapíšeme vypočtený úhel do dalšího *labelu*, který se nám bude v programu zobrazovat.

6.1.2 Funkce `zmenHodnotu()`

Tato funkce slouží ke změně hodnot v seznamu `hodnoty` uživatelem, tedy k nastavování obou indexů lomů a úhlu α. Funkce samotná se ovšem zavolá až po stisknutí určitého tlačítka, což je řešeno v následující funkci s názvem `pressed()`.

6.1.3 Funkce `pressed()`

Funkce, která je zavolána při stisknutí tlačítka. O zavolání této funkce se stará takzvaný *push_handler* (řádek 199), který zaznamenává události a následně volá k nim přiřazené funkce – v našem případě je v závorce *push_handleru* uvedeno `on_key_press=pressed`, což znamená, že při stisknutí klávesy chceme zavolat funkci `pressed()`. Té navíc ještě musíme předat proměnné, ve kterých jsou uchované informace o stisknuté klávese `(symbol, modifiers)`, tedy jaká klávesa byla stisknuta a zda během jejího stisknutí byl zároveň aktivní některý modifikátor, jako například klávesa *SHIFT*.

Následně už nám jen stačí porovnávat proměnnou `symbol` a případně i `modifiers` s jejich odpovídajícími hodnotami v modulu `key` a při dané shodě provést požadovanou operaci. V tomto případě sledujeme primárně klávesy *šipka nahoru* a *šipka dolů*, kterými se posunujeme v řádcích uživatelského rozhraní pomocí indexu `i`, a dále sledujeme použití *šipky vlevo* a *vpravo* i s jejich modifikátory (modifikátoru

SHIFT odpovídá číslo 17). Při stisknutí jedné ze šipek se zavolá výše zmíněná funkce `zmenHodnotu()`, která buď to přičte nebo odečte určitou hodnotu v daném řádku od té stávající v závislosti na tom, co bylo stisknuto a zda byl použit i modifikátor *SHIFT*. Se *SHIFTem* chceme dělat menší kroky, bez *SHIFTu* větší.

6.1.4 Funkce `draw()`

Zde zajišťujeme správné vykreslování okna. Na začátku je vždy potřeba zavolat funkci `window.clear()`, která celé okno smaže, aby další vykreslování probíhalo načisto. Následujícím cyklem vykreslujeme objekty typu `Label`, které nám slouží jako uživatelské rozhraní.

Nakonec je potřeba vykreslit samotné znázornění Snellova zákona. To provedeme příkazy `pyglet.graphics.draw()`. Zápis celého příkazu se může zdát poněkud krkolomný, ovšem při bližším rozboru smysl dává. První číslo v jeho závorce udává počet bodů, pomocí kterých budeme daný obrazec vykreslovat. V případě vykreslení úsečky potřebujeme body dva, pokud budeme vykreslovat pouze tečku (viz řádek 179 pro tečkovanou čáru), bude nám stačit bod jeden. Následující atribut `pyglet.gl.GL_LINES` musíme použít pro vykreslení přímky, v případě vykreslení bodu by za poslední tečkou bylo `GL_POINTS`, pokud bychom chtěli třeba trojúhelníky, použili bychom `GL_TRIANGLES`[13]. Dále řetězcem `"v2f"` říkáme, že chceme nastavovat souřadnice vertexů (čili bodů úsečky) a zadávat je chceme ve formátu typu float, tedy desetinného čísla. Podobně bychom mohli nastavovat například barvu, pokud bychom použili například výraz `"c3I"`. Za každým z těchto uvozujících řetězců následuje závorka, kterou určujeme požadované souřadnice, barvu, atd. My zde pouze nejprve vykreslíme rozhraní dvou prostředí a poté tečkovanou čárou kolmici na toto rozhraní a následně i paprsky, jejichž souřadnice jsou dopočítávány pomocí goniometrických funkcí.

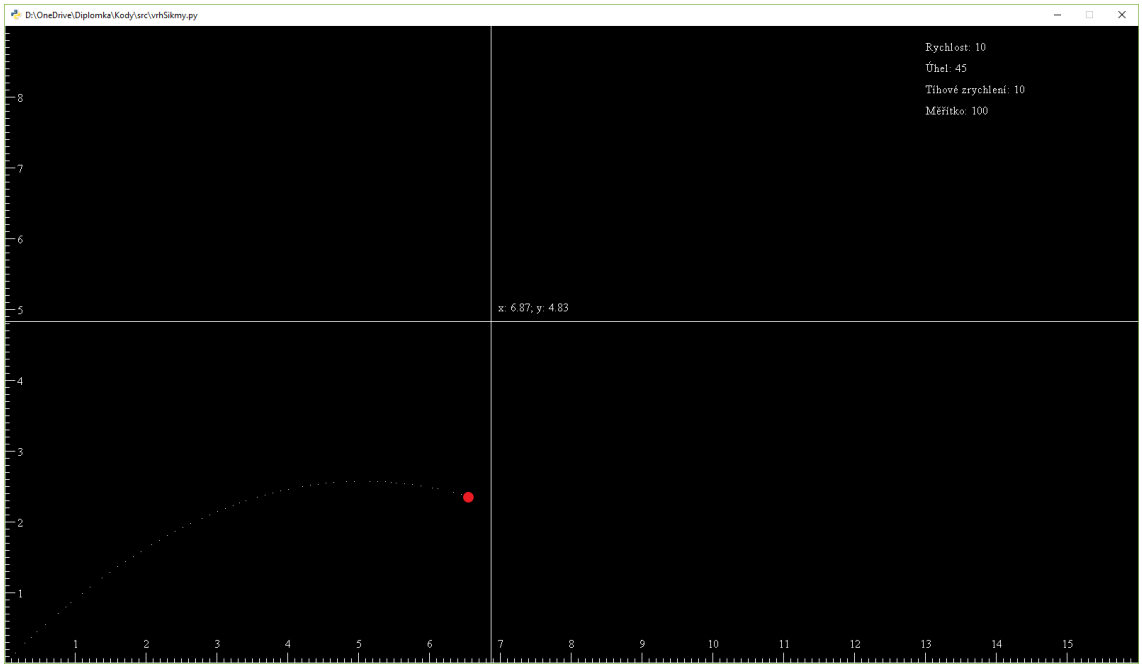
6.1.5 Funkce `main()`

Jde o funkci, která se stará o spuštění a běh programu. Zde nastavujeme například jak často se má volat funkce `update()` nebo spouštíme modul `app`.

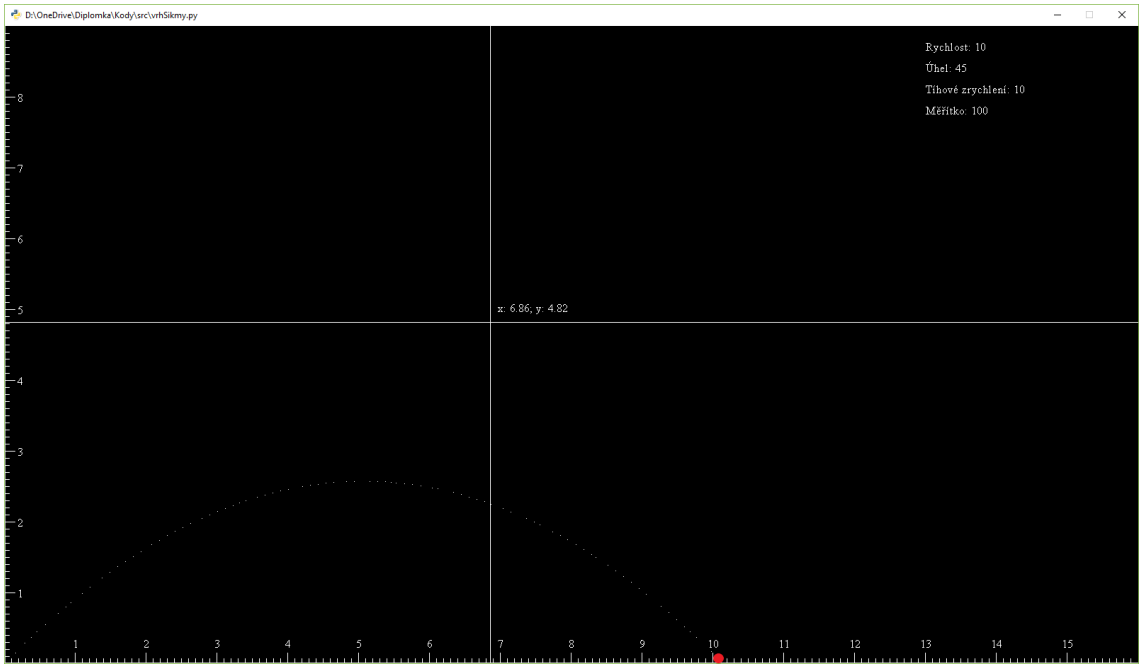
Aplikace byla psána pokud možno jednoduše, proto neodpovídá běžným standardům ani optimálně psanému kódu a zahrnuje pouze zlomek možností, které knihovna pyglet nabízí. V odkazu [13] na dokumentaci lze nalézt mnoho dalších užitečných rad, postupů a popisů jednotlivých modulů a funkcí této knihovny, kterou je v případě pokročilejší práce s ní vhodné detailněji prostudovat.

6.2 Vrh šikmý

Dalším příkladem bude zobrazení šikmého vrhu tělesa v homogenním tíhovém poli. Jako základní vlastnosti aplikace si stanovme možnost měnit počáteční rychlost tělesa, úhel vrhu, tíhové zrychlení na něj působící a změnu měřítka, ve kterém se simulace bude přepočítávat a zobrazovat. Okno simulace by mělo být ohraničeno měřítky pro určování délky a výšky vrhu, které se budou se změnou nastavení automaticky měnit. Letící těleso by také pro názornost mělo zanechávat stopu své trajektorie. Pro uživatelské pohodlí nakonec vybavíme kurzor myši vodícími přímkami a funkcí pro určování pozice, na které se ukazatel nachází. Obr. 11 a Obr. 12 ukazují cílovou podobu aplikace.



Obrázek 11: Vrh šikmý 1



Obrázek 12: Vrh šikmý 2

Než se pustíme do psaní programu, je potřeba zvážit přístup, který k simulaci zvolíme. První možností by bylo kompletně nasimulovat pohyb tělesa v tíhovém poli, takže by výsledný program nebyl striktně závislý na počátečních souřadnicích a dokázal by tak simulovat nejen požadovaný šikmý vrh, ale i volný pád, vodorovný vrh atd., a to prakticky z kteréhokoli místa na obrazovce. Tento přístup by byl rozhodně efektivnější a zároveň by to mohl být první krok k tvorbě daleko komplexnějších simulací pohybu tělesa, kam by šlo implementovat například i působení dalších sil, nicméně bychom si zbytečně znesnadnili splnění zadání, které po nás tyto možnosti nepožaduje. Aby totiž vše fungovalo jak má a aby výsledný pohyb tělesa zároveň nepůsobil „podivně“, bylo by potřeba k přepočtům rychlostí ze zrychlení přidat i (alespoň) zjednodušenou simulaci tření, odrazů a s tím i systém detekce kolizí tělesa s podložkou a stěnami okna, nehledě na nutnost komplexnějšího ovládání.

Proto je v tomto případě vhodnější zvolit přístup, který se bude přesně držet stanoveného zadání, což sice omezí případné rozšíření možností simulace, ovšem značně nám zjednoduší implementaci, a to o minimálně několik stovek řádků kódu. K omezení jednoduchého rozšíření kódu i na jiné druhy pohybu dojde, protože nám pro tento přístup budou stačit známé vztahy pro výpočet *x-ových* a *y-ových* souřadnic v závislosti na čase a vztah pro výpočet času dopadu t_d vrženého tělesa, které odpovídají pouze šikmému vrhu v homogenním tíhovém poli, takže pomocí nich nemůžeme v programu popsat i jiné druhy pohybu, které mohou nastat. Pro výpočet času dopadu využijeme vzorec $t_d = \frac{2v_0 \sin \alpha}{g}$, pro výpočet *x-ové* souřadnice vztah: $x = v_0 t \cos \alpha$ a nakonec vzorec: $y = v_0 t \sin \alpha - \frac{1}{2} g t^2$ pro výpočet souřadnice *y-ové*. Po zvolení postupu je možné začít psát kód (Př. 43).

```
1  import pyglet
2  from pyglet.window import key, mouse
3  import ctypes
4  import math
5
6  user32 = ctypes.windll.user32
7  screensize = user32.GetSystemMetrics(0), user32.GetSystemMetrics(1)
8
9  window = pyglet.window.Window(round(screensize[0]//1.2),
10 round(screensize[1]//1.2))
11
12 batch = pyglet.graphics.Batch()
13
14 teleso = pyglet.resource.image("teleso.png")
15 teleso.anchor_x = teleso.width/2
16 teleso.anchor_y = teleso.height/2
17 startX = teleso.width//2
18 startY = teleso.height//2
19 sprite = pyglet.sprite.Sprite(teleso, x=startX,
20                                y=startY, batch=batch)
21
22 seznamStop = []
23 osy = []
24 souradnice = []
25
26 mouseCoords = [0, 0]
27
28 meritko = 100                    #100 px = 1 m
29 skutecneG = 10
30 t = 0
31 td = 0
32 skutecneV = 10
33 alfa = 45
34 start = False
35 odstartovano = False
36 restartovano = True
37
38 hodnoty = [skutecneV, alfa, skutecneG, meritko]
39
40 #Měřítko: 100 px = 1 m
41 drahaVMetrech = sprite.x / meritko
42
43 ui = ["Rychlost:␣", "Úhel:␣", "Tíhové␣zrychlení:␣", "Měřítko:␣"]
44 labels = []
45
46
47 def initialize():
48     """Nastavení os s měřítky a labelů ihned
49     po startu programu
50     """
51     global labels
52
53     for y in range(4):
54         label = pyglet.text.Label(ui[y],
55                   font_name='Times␣New␣Roman',
```

```
56                  font_size=12,
57                  x=window.width, y=window.height-20-(30 * y),
58                  anchor_x='right', anchor_y='top',
59                  multiline = True, width = 300, batch=batch)
60          labels.append(label)
61
62      vertex_list = batch.add(2, pyglet.gl.GL_LINES, None,
63              ("v2f", (0, 1, window.width, 1)))
64
65      vertex_list = batch.add(2, pyglet.gl.GL_LINES, None,
66              ("v2f", (1, 0, 1, window.height)))
67
68      meritkoOs()
69
70
71  def meritkoOs():
72      """Aktualizace os, která se spustí
73      jen po změně měřítka
74      """
75      global hodnoty
76      global osy
77
78      meritko = hodnoty[3]
79
80      for prvek in osy:
81          prvek.delete()
82
83      osy = list()
84
85      i = meritko
86      while i < window.width:
87          if meritko >= 20:
88              prvek = batch.add(2, pyglet.gl.GL_LINES, None,
89              ('v2i', (i, 0, i, window.height//60)))
90              popisek = pyglet.text.Label(str(i//meritko),
91                    font_name='Times New Roman',
92                    font_size=12,
93                    x=i, y=window.height//50,
94                    anchor_x='center', anchor_y='bottom',
95                    multiline = False, width = 10, batch=batch)
96              osy.append(prvek)
97              osy.append(popisek)
98          elif i % 50 == 0:
99              prvek = batch.add(2, pyglet.gl.GL_LINES, None,
100             ('v2i', (i, 0, i, window.height//60)))
101             popisek = pyglet.text.Label(str(i//meritko),
102                   font_name='Times New Roman',
103                   font_size=12,
104                   x=i, y=window.height//50,
105                   anchor_x='center', anchor_y='bottom',
106                   multiline = False, width = 10, batch=batch)
107             osy.append(prvek)
108             osy.append(popisek)
```

```
111             i = i + meritko
112
113         i = meritko//10
114         if meritko >=20:
115             while i < window.width:
116                 prvek = batch.add(2, pyglet.gl.GL_LINES, None,
117                         ("v2f", (i, 0, i, window.height//120)))
118                 osy.append(prvek)
119                 i = i + meritko//10
120
121         j = meritko
122         while j < window.height:
123             if meritko >= 20:
124                 prvek = batch.add(2, pyglet.gl.GL_LINES, None,
125                         ("v2i", (0, j, window.height//60, j)))
126                 popisek = pyglet.text.Label(str(j//meritko),
127                           font_name='Times New Roman',
128                           font_size=12,
129                           x=window.height//50, y=j,
130                           anchor_x='left', anchor_y='center',
131                           multiline = False, width = 10, batch=batch)
132                 osy.append(prvek)
133                 osy.append(popisek)
134             elif j % 50 == 0:
135                 prvek = batch.add(2, pyglet.gl.GL_LINES, None,
136                         ("v2i", (0, j, window.height//60, j)))
137                 popisek = pyglet.text.Label(str(j//meritko),
138                           font_name='Times New Roman',
139                           font_size=12,
140                           x=window.height//50, y=j,
141                           anchor_x='left', anchor_y='center',
142                           multiline = False, width = 10, batch=batch)
143                 osy.append(prvek)
144                 osy.append(popisek)
145             j = j + meritko
146
147         j = meritko//10
148         if meritko >= 20:
149             while j < window.height:
150                 prvek = batch.add(2, pyglet.gl.GL_LINES, None,
151                         ("v2i", (0, j, window.height//120, j)))
152                 osy.append(prvek)
153                 j = j + meritko//10
154
155
156 def restartTeleso():
157     """Vrátí těleso do výchozí pozice
158     """
159     global startX
160     global startY
161     global sprite
162     global t
163     global restartovano
164
165     sprite.x = startX
166     sprite.y = startY
```

```
167
168     t = 0
169     restartovano = True
170
171
172 def smazStopu():
173     """Smaže stopy zanechané tělesem
174     """
175     global seznamStop
176
177     for stopa in seznamStop:
178         stopa.delete()
179     seznamStop = list()
180
181
182 def vypoctiTD(v, alfa, g):
183     """Vypočte čas dopadu tělesa
184     """
185     global td
186     alfa = math.radians(alfa)
187
188     td = (2 * v * math.sin(alfa))/g
189
190
191 def kriz():
192     """Vytváří batch vodítek a souřadnic
193     u kurzoru"""
194     global souradnice
195     global hodnoty
196
197     for prvek in souradnice:
198         prvek.delete()
199
200     souradnice = list()
201
202     cara = batch.add(2, pyglet.gl.GL_LINES, None,
203             ("v2i", (mouseCoords[0], 0, mouseCoords[0], window.height))
                 )
204     souradnice.append(cara)
205     cara = batch.add(2, pyglet.gl.GL_LINES, None,
206             ("v2i", (0, mouseCoords[1], window.width, mouseCoords[1])))
207     souradnice.append(cara)
208
209     popisek = pyglet.text.Label(("x: " + str(mouseCoords[0]/hodnoty[3])
        +
210         "; y: "+ str(mouseCoords[1]/hodnoty[3])),
211           font_name='Times New Roman',
212           font_size=12,
213           x=mouseCoords[0] + 10, y=mouseCoords[1] + 10,
214           anchor_x='left', anchor_y='bottom',
215           multiline = False, width = 10, batch=batch)
216
217     souradnice.append(popisek)
218
219
220 def update(dt):
221     """Funkce starajici se o aktualizaci stavu
222     programu.
```

```
223     """
224     global t
225     global td
226     global start
227     global odstartovano
228     global startX
229     global startY
230     global drahaVMetrech
231     global seznamStop
232     global hodnoty
233     global restartovano
234 
235     """Omezení nastavovaných hodnot
236     na určité rozsahy
237     """
238     if hodnoty[0] < 0:
239         hodnoty[0] = 0
240     if hodnoty[1] < 0:
241         hodnoty[1] = 0
242     if hodnoty[1] > 90:
243         hodnoty[1] = 90
244     if hodnoty[2] <= 0:
245         hodnoty[2] = 0.1
246     if hodnoty[3] <= 0:
247         hodnoty[3] = 5
248 
249     """Přepočet nastavovaných "reálných" hodnot
250     na hodnoty programové v daném měřítku (metry
251     na pixely, m/s^2 na px/s^2, stupně na radiány)
252     """
253 
254     v = hodnoty[0] * hodnoty[3]
255     alfa = math.radians(hodnoty[1])
256     g = hodnoty[2] * hodnoty[3]
257 
258     labels[0].text = ui[0] + str(round(hodnoty[0],1))
259     labels[1].text = ui[1] + str(round(hodnoty[1]))
260     labels[2].text = ui[2] + str(round(hodnoty[2],1))
261     labels[3].text = ui[3] + str(round(hodnoty[3],1))
262 
263     if sprite.y <= teleso.height//2 and odstartovano:
264         sprite.y = teleso.height//2
265         start = False
266         odstartovano = False
267 
268     if start and t <= td:
269         odstartovano = True
270         restartovano = False
271         t = t + dt
272         sprite.x = startX + v * t * math.cos(alfa)
273         sprite.y = startY + v * t * math.sin(alfa) - 1/2 * g * t**2
274         stopa = batch.add(1, pyglet.gl.GL_POINTS, None,
275                 ("v2f", (sprite.x, sprite.y)))
276         seznamStop.append(stopa)
```

```
279 def draw():
280     """Funkce, ktera se stara o vykreslovani okna
281     """
282 
283     window.clear()
284     batch.draw()
285     kriz()
286 
287 
288 def pressed(symbol, modifiers):
289     """Funkce starajici se o ovladani programu
290     pomoci klavesnice.
291     """
292     global start
293     global hodnoty
294     global restartovano
295 
296     if symbol == key.SPACE:
297         if restartovano:
298             vypoctiTD(hodnoty[0], hodnoty[1], hodnoty[2])
299             start = True
300 
301     if symbol == key.R:
302         restartTeleso()
303 
304     if symbol == key.T:
305         smazStopu()
306 
307     if symbol == key.DOWN:
308         hodnoty[1] = hodnoty[1] - 5
309 
310     if symbol == key.UP:
311         hodnoty[1] = hodnoty[1] + 5
312 
313     if symbol == key.LEFT and modifiers == 17:
314         hodnoty[0] = hodnoty[0] - 1
315     elif symbol == key.LEFT:
316         hodnoty[0] = hodnoty[0] - 5
317 
318     if symbol == key.RIGHT and modifiers == 17:
319         hodnoty[0] = hodnoty[0] + 1
320     elif symbol == key.RIGHT:
321         hodnoty[0] = hodnoty[0] + 5
322 
323     if symbol == key.W and modifiers == 17:
324         hodnoty[2] = hodnoty[2] + 0.1
325     elif symbol == key.W:
326         hodnoty[2] = hodnoty[2] + 1
327 
328     if symbol == key.S and modifiers == 17:
329         hodnoty[2] = hodnoty[2] - 0.1
330     elif symbol == key.S:
331         hodnoty[2] = hodnoty[2] - 1
332 
333     if symbol == key.D and modifiers == 17:
334         hodnoty[3] = hodnoty[3] + 5
```

```
334             hodnoty[3] = hodnoty[3] + 5
335             if hodnoty[3] <= 0:
336                 hodnoty[3] = 5
337             meritkoOs()
338         elif symbol == key.D:
339             hodnoty[3] = hodnoty[3] + 50
340             if hodnoty[3] <= 0:
341                 hodnoty[3] = 5
342             meritkoOs()
343 
344         if symbol == key.A and modifiers == 17:
345             hodnoty[3] = hodnoty[3] - 5
346             if hodnoty[3] <= 0:
347                 hodnoty[3] = 5
348             meritkoOs()
349         elif symbol == key.A:
350             hodnoty[3] = hodnoty[3] - 50
351             if hodnoty[3] <= 0:
352                 hodnoty[3] = 5
353             meritkoOs()
354 
355 
356 def mouseMotion(x, y, dx, dy):
357     mouseCoords[0] = x
358     mouseCoords[1] = y
359 
360 
361 window.push_handlers(on_key_press=pressed, on_mouse_motion=mouseMotion)
362 
363 
364 def main():
365     """Funkce, jejimz zavolanim na konci kodu
366     spustime samotny program.
367     """
368 
369     @window.event
370     def on_draw():
371         draw()
372     initialize()
373     pyglet.clock.schedule_interval(update, 1/120.0)
374     pyglet.app.run()
375 
376 
377 main()
```

Výstup:

Příklad 43: Vrh šikmý

Kód bude mít stejnou strukturu jako v předchozím příkladu (Př. 42) a bude využívat stejných základních programových funkcí (`update()`, `draw()`, `main()` atd). Na začátku si opět nejprve naimportujeme moduly, které budeme potřebovat, zde navíc ještě s modulem zahrnujícím myš. Stejně tak si musíme vytvořit proměnné, které budeme v programu využívat.

V této části si můžeme všimnout dvou novinek. První z nich je vytvoření objektu s názvem `batch`. Tento objekt nám jednak usnadní práci s vykreslovanými čarami, body a tělesy, protože je při správném použití budeme moci volitelně mazat a znovu vykreslovat jinde a jinak, a jednak místo programátora provádí optimalizaci při práci s grafickou pamětí, takže zrychluje běh programu[13].

Druhou novinkou je blok kódu, kterým do programu vložíme objekt typu *sprite*, což pro nás bude grafická reprezentace tělesa, které se bude v simulaci pohybovat. Vkládaný *png* obrázek zde v jednoduchosti stačí vytvořit v programu *Malování* a uložit do složky, kde se nachází i soubor s kódem. U vytvořeného objektu `sprite` musíme nejen nastavit jeho startovní souřadnice, ale musíme jej „propojit“ s objektem `batch`, který poté bude obrázek vykreslovat.

6.2.1 Funkce `initialize()`

Tato funkce se spustí ihned po startu programu a provede nastavení grafického rozhraní simulace. Můžeme si všimnout, že místo příkazu *pyglet.graphics.draw()*, kterou jsme používali v předchozím příkladu, zde používáme příkaz `vertex_list = batch.add()`. Tím vytváříme seznam vertexů a jejich nastavení (to je stejné jako u `draw()`), jenž se ale nevykreslí rovnou, pouze se předá objektu `batch`, který se postará o vykreslení všech k němu takto přiřazených objektů při zavolání příkazu `batch.draw()`. Do proměnné, kterou při nastavování vertexů vytváříme, se ukládá odkaz na tento seznam vertexů, díky kterému s ním můžeme později dále pracovat. V tomto případě to ale není potřeba, dále tuto možnost již však využijeme.

Tvorbu *labelů* již známe, zde je příkaz pro jejich vytvoření rozšířen o atribut `batch=batch`, který daný `label` propojí s objektem `batch`, stejně jako v případě obrázku, abychom jej mohli později vykreslit.

Část zde nastavených hodnot se v programu už nikdy nebude měnit a proto je zbytečné hodnoty znovu a znovu nastavovat ve funkci `update()` – značně tím šetříme výkon počítače. Proto zavádíme tuto samostatně stojící funkci, kterou voláme pouze po spuštění programu.

6.2.2 Funkce `meritkoOs()`

Funkce, ve které budeme nastavovat stupnice na osách v závislosti na změněném měřítku. Ani tuto část kódu není vhodné provádět při každé aktualizaci stavu programu, neboť obsahuje velké množství cyklů, používaných k výpočtu souřadnic pro jednotlivé čáry a *labely*, které nám ve výsledku dohromady vytvoří stupnice pro obě osy. Nám stačí, pokud k přepočtům dojde jen tehdy, když uživatel změní měřítko, tedy při stisku klávesy k této funkci přiřazené (viz funkce `pressed()`). Měřítko je v základu nastaveno na hodnotu 100, tedy 100 px na obrazovce odpovídá vzdálenosti

1 m. Hlavní stupnice takto bude dělená po 100 px a u každé značky bude označená i vzdálenost v metrech. Tento výpočet nám zajišťuje první cyklus `while`, který navíc obsahuje ještě dvě podmínky, které se starají o to, aby v případě, kdy uživatel zvolí příliš malé měřítko, se jednotlivé popisky os nezačaly překrývat. Pokud by tento případ měl nastat, stupnice přejde z dělení na metry na dělení po deseti metrech.

Druhý cyklus `while`, který ještě obalíme podmínkou `if`, zajišťuje – při dostatečné velikosti měřítka – vykreslení vedlejší stupnice x, pro jemnější určení vzdálenosti.

Druhá polovina této funkce provádí to samé, pouze pro svislou stupnici.

V každém z těchto cyklů si můžeme povšimnout, že jednotlivé čáry opět nevykreslujeme rovnou, ale znovu tento úkon dáváme na starost objektu `batch`. Zde navíc ještě hodnotu, kterou nám tento příkaz vrací, ukládáme do seznamu `osy`, abychom se přes ni mohli dostat zpátky k nastaveným vertexům, a v případě potřeby je smazat a nahradit novými – zde to provádíme na začátku funkce, tedy hned po jejím zavolání, které nastane při změně měřítka, příkazem `delete()`, který ovšem musíme provést pro každý prvek v seznamu `osy`, kam jsme nastavení jednotlivých čar a labelů uložili. Následně musíme celý seznam znovu inicializovat, jinak by nám v něm zůstaly prázdné objekty `batch`e.

6.2.3 Funkce `restartTeleso()`

Pomocí této funkce restartujeme těleso zpět do počáteční pozice vždy, když bude stisknuta příslušná klávesa. Zároveň s restartem musíme vynulovat i čas `t`, aby při dalším vrhu tělesa došlo ke správnému výpočtu nové trajektorie. Proměnnou `restartovano` hlídáme, kdy můžeme těleso znovu vrhnout.

6.2.4 Funkce `smazStopu()`

Touto funkcí smažeme stopu zobrazující trajektorii vrženého tělesa. Postup je stejný jako v případě mazání os při změně jejich měřítka.

6.2.5 Funkce `vypoctiTD()`

Zde se dostáváme k jednomu ze vzorců, které jsme si na začátku připomenuli. Při výpočtu času dopadu tělesa nesmíme zapomenout převést úhel na radiány, neboť modul *math* počítá právě s nimi.

6.2.6 Funkce `kriz()`

Tato funkce bude sloužit k vykreslování vodítek kurzoru myši a vypsání jeho souřadnic v metrech poblíž kurzoru. Opět při tom využíváme objektu `batch` a průběžného

vymazávání vertexů. Ty máme uloženy v jiném seznamu, abychom mohli mazat pouze ty, které se vážou ke grafice spojené s kurzorem myši.

Stávající souřadnice kurzoru, které následně předáváme objektu `batch`, máme uloženy v seznamu `mouseCoords`. V něm uložené souřadnice se mění při každém *eventu* pohybu myši, k čemuž dochází ve funkci `mouseMotion()` níže.

6.2.7 Funkce `update()`

Zde, jak již víme z předchozího příkladu, dochází ke aktualizaci stavu programu a výpočtům s tím spojených.

V první části musíme kontrolovat rozsahy hodnot, které uživatel zadal, aby nemohl zadat například zápornou rychlost nebo záporné tíhové zrychlení. Poté je potřeba podle měřítka přepočítat zadané jednotky „reálného“ světa (tedy m a ms^{-2}) na px a úhel ve stupních na radiány. Další část aktualizuje výpis hodnot v uživatelském rozhraní a první `if` funguje jako jednoduchá kolize pro detekci momentu, kdy se těleso dotýká podložky.

Poslední `if` slouží k výpočtu souřadnic tělesa v daném čase. Čas se s každým průběhem funkce zvýší o hodnotu `dt`, díky čemuž jsme schopni zobrazovat změnu pozice tělesa v průběhu času. Po výpočtu obou souřadnic tělesa dochází i k předání těchto souřadnic do `batch`e, abychom je pak mohli vykreslit a zaznamenat tak trajektorii, kterou se těleso pohybuje. Odkaz na tyto souřadnice musí být opět uložen do seznamu (`seznamStop`), abychom jej mohli později smazat.

6.2.8 Funkce `draw()`, `pressed()` a `main()`

Funkce nám známé již z předchozího příkladu. Funkce `draw()` je zde však daleko jednodušší kvůli využití objektu `batch`, který spravuje veškeré vykreslované objekty, a zavoláním jeho vlastní funkce `draw()` dojde k jejich vykreslení.

Ve funkci `pressed()` opět definujeme ovládání programu pomocí klávesnice – mezerník pro vrhnutí, R pro návrat tělesa do výchozí pozice, T pro smazání jeho stopy, šipky nahoru a dolů pro změnu úhlu vrhu, doleva a doprava pro změnu rychlosti, W a S pro změnu tíhového zrychlení a A a D pro změnu měřítka. Některé z veličin lze navíc měnit o menší hodnoty za pomoci modifikátoru SHIFT.

Ve funkci `main()` navíc ještě oproti předchozímu programu voláme funkci `inicialize()`, aby ihned po spuštění došlo k prvotnímu rozvržení a vykreslení uživatelského rozhraní.

Závěr

Jak jsem naznačil v úvodu, tato práce by měla sloužit studentům fyziky a učitelství fyziky jako úvod do programování v jazyce Python. Zahrnuje proto to nejdůležitější, co by každý začínající uživatel měl vědět – obecný popis jazyka a některých modulů, průvod instalací, styl syntaxe, následovaný stručným popisem jednotlivých příkazů, funkcí a základních objektů v Pythonu. Druhá polovina práce se již věnuje konkrétní aplikaci jazyka na matematické výpočty a končí příklady jednoduchých simulací fyzikálních jevů.

Z principu je nemožné, aby se tato práce Pythonu věnovala více do hloubky a v plné komplexnosti, neboť už jen popis jeho možností na poli matematických a fyzikálních výpočtů by vystačil na několik tlustých knih. Mým přáním pouze je, aby tato práce studentům usnadnila vstup do světa programování a probudila v nich zájem o hlubší studium a využívání Pythonu, ale třeba i dalších programovacích jazyků, které k začátečníkům nejsou již tak vstřícné.

Použitá literatura a zdroje

[1] HILL, Christian. *Learning scientific programming with Python*. Cambridge University Press, 2015. ISBN 978-1-107-42822-5.

[2] Foreword for „Programming Python“ (1st ed.). *Python.org* [online]. [cit. 2017-03-18]. Dostupné z: https://www.python.org/doc/essays/foreword/

[3] ASCHER, David *Naučte se Python - pohotová příručka*. Grada Publishing a.s., 2003. ISBN 802470367x.

[4] KINDER, Jesse M. a Philip Charles NELSON *A student's guide to Python for physical modeling*. Princeton University Press, 2015. ISBN 978-0691170503.

[5] MAREK, Tomáš. *Efektivní vizualizace dat se zaměřením na základní typy grafů* [online]. MASARYKOVA UNIVERZITA, 2014 [cit. 2017-09-01]. Dostupné z: https://is.muni.cz/th/362075/ff_m/362075_magisterska-prace.pdf. Magisterská diplomová práce. MASARYKOVA UNIVERZITA, Filozofická fakulta. Vedoucí práce Mgr. Jan Boček.

[6] Data Visualization – How to Pick the Right Chart Type? *EazyBI* [online]. [cit. 2017-09-03]. Dostupné z: https://eazybi.com/blog/data_visualization_and_chart_types/

[7] Graphs and Charts. *Skills you need* [online]. [cit. 2017-09-12]. Dostupné z: https://www.skillsyouneed.com/num/graphs-charts.html

[8] Medián. *Matematika.cz* [online]. [cit. 2017-09-13]. Dostupné z: https://matematika.cz/median

[9] Modus. *Matematika.cz* [online]. [cit. 2017-09-13]. Dostupné z: https://matematika.cz/modus

[10] Rozptyl, směrodatná odchylka a variační koeficient. *Statistika & my: Měsíčník českého statistického úřadu* [online]. [cit. 2017-09-24]. Dostupné z: http://www.statistikaamy.cz/2017/01/rozptyl-smerodatna-odchylka-a-variacni-koeficient/

[11] OLEHLA, Miroslav a Jan TIŠER. *Praktické použití Fortranu. 2. upravené vydání*. Praha: Nakladatelství dopravy a spojů, 1979.

[12] Bitbucket [online]. 2017 [cit. 2018-03-03]. Dostupné z: https://bitbucket.org/pyglet/pyglet/wiki/Home

[13] Graphics. *Pyglet Documentation* [online]. [cit. 2018-03-11]. Dostupné z: http://pyglet.readthedocs.io/en/pyglet-1.3-maintenance/programming_guide/graphics.html

[14] *Matplotlib* [online]. 2018 [cit. 2018-03-29]. Dostupné z: https://matplotlib.org/index.html

Seznam příkladů

Seznam obrázků

Seznam tabulek

Přílohy

Na přiloženém DVD se nachází plné znění diplomové práce, dále jsou na DVD umístěny všechny kódy, které byly v práci použity a instalační soubor základní verze Pythonu.