STUDENTSKÝ ČASOPIS A KORESPONDENČNÍ SEMINÁŘ

Ročník XXXIII
Číslo 1

M&M

MATEMATIKA FYZIKA INFORMATIKA

Uvnitř najdete několik témat a s nimi souvisejících úloh. Zamyslete se nad nimi a pošlete nám svá řešení. My vám je opravíme a ta nejzajímavější z nich otiskneme. Nejlepší řešitele zveme na podzim a na jaře na soustředění.

Milá čtenářko, milý čtenáři,

právě otevíráš první číslo již 33. ročníku časopisu M&M!

Na následujících stránkách tě netrpělivě vyhlížejí tématka, díky nimž můžeš porozumět různým zajímavým oblastem matematiky, fyziky a informatiky. S tématky přichází nové úlohy, jejichž řešením procvičíš svůj důvtip i nově nabyté znalosti, a konečně i problémy otvírající ti dveře k hlubšímu bádání. Na úlohy a problémy tě upozorní zvoneček na okraji stránky.

Tak na co čekáš? Vyzkoušíš si nejprve v tématku o programovacím jazyce *Python*, jak používat cykly (nebo jak je nepoužívat, pokud to už umíš)? Nebo se raději v *rozměrové analýze* naučíš, jak jednoduše předpovědět rychlost šíření vln na hladině rybníka? Hledáš-li matematiku a teoretickou informatiku, jsou tu pro tebe *grafy*, a chceš-li nahlédnout pod pokličku fungování počítačů, nalistuj na tématko o *operačních systémech*. Nebo tě láká *obecná teorie relativity*? První díl ti představí tenzory, které jsou (nejen) pro tuto teorii nesmírně důležité.

Svá řešení neváhej sepsat a odevzdat nám je do odevzdávátka, které najdeš po přihlášení na našem webu. Podrobnější návod najdeš v přiloženém letáčku[1] nebo na našich stránkách `https://mam.mff.cuni.cz`. Můžeš také řešit dohromady s kamarádem, ke spojení s ostatními řešiteli i s autorem tématka ti pomůže náš Discord. A pokud se rozhodneš nějakému tématu věnovat ještě nadšeněji, můžeš nám o něm poslat článek.

A co bude dál? Řešení ti opravíme a obodujeme. Za každé číslo, jehož řešením získáš alespoň π bodů, od nás dostaneš ponožku jedné barvy. Zvládneš za rok nasbírat stejnobarevný pár? Kromě dalších cen ti za body třeba promineme přijímačky na Matfyz nebo tě pozveme na týdenní soustředění. Na to podzimní (17.–25. října 2026) se můžeš dostat ještě díky bodům z tohoto čísla (za řešení odevzdaná do prvního deadlinu), tak hurá do řešení!

Na závěr ještě jedna **NOVINKA**: v letošním ročníku přibyla nová možnost, jak získat legendární M&Mí předměty! Stačí najít kamaráda, který M&M ještě neřešil, a ukázat mu, o co přichází. Když se pak tento kamarád zaregistruje, uvede tvé jméno v kolonce *Jak ses o M&M dozvěděl(a)* a získá v nějakém čísle alespoň π bodů, získáš tím zbrusu novou odměnu (a tvůj kamarád samozřejmě ponožku).

Přejeme ti léto plné radosti z objevování.

Tvoji orgové M&M

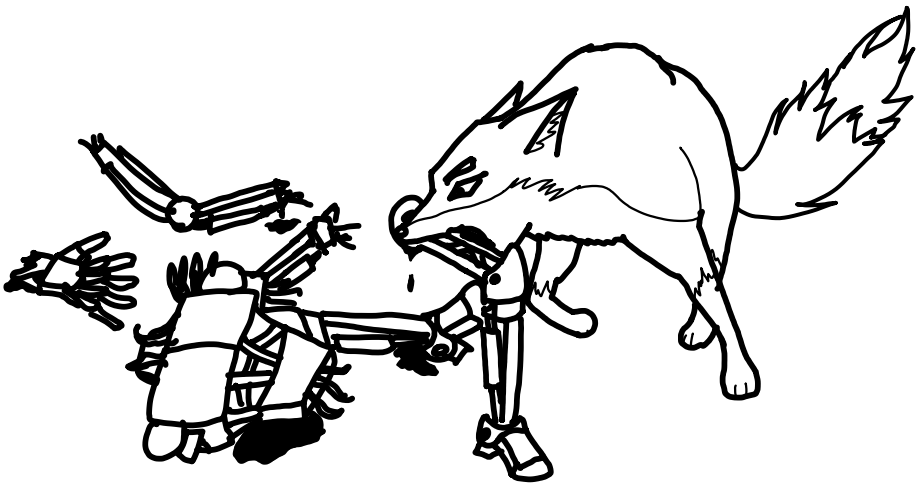
Odkaz na Discord: `https://discord.gg/6Y8TcDg3qP`

[1] Elektronickou verzi najdeš na webu `https://mam.mff.cuni.cz/jak-resit/`.

Obsah

Pravidla použití AI

Při řešení úloh a problémů z M&M je zakázáno používat generativní umělou inteligenci (ChatGPT a podobně), není-li uvedeno jinak. Využití AI jako nástroje pro dohledávání detailů je však povoleno.



Zadání témat

1. deadline: 10. září 2026 | 2. deadline: 6. října 2026
Řešení odevzdaná do 10. září se započítají pro účast na soustředění.

Téma 1 – Grafy

Díl 1: Základy teorie grafů

V tomto tématku se budeme věnovat grafům. Nebude se však jednat o grafy funkcí, které znáte ze školy. Budeme se bavit o tématu na pomezí matematiky a informatiky, nás však bude zajímat spíše matematický a čistě teoretický pohled na věc. V tomto tématku tedy nebudeme programovat, přesto bude přínosné i pro informatiky. Zároveň k tomuto tématku nepotřebujete skoro žádné předchozí znalosti (i když znalost kombinatoriky se může hodit – bohatě vám bude stačit první díl stejnojmenného tématka z minulého roku). V prvním čísle si vysvětlíme základní pojmy a představíme jednu z nejčastějších důkazových metod, která se v teorii grafů používá – indukci. Podobně jako v tématku z minulého roku občas narazíte na text psaný *kurzívou*, který vás bude vybízet k zamyšlení. Zkuste se v těchto místech na chvíli zastavit ve čtení, zamyslet se nad odpovědí na otázku a až poté pokračovat a ověřit si správnost svých úvah.

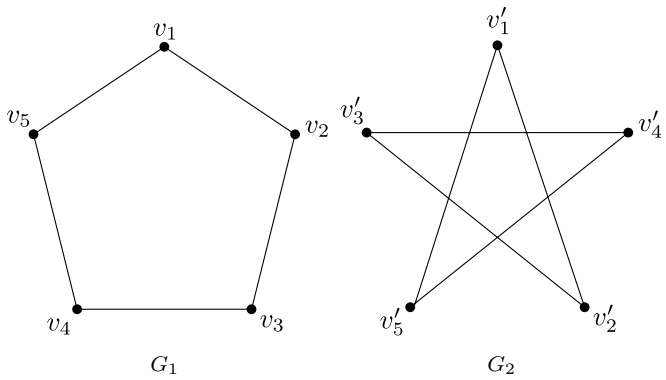
Základní pojmy

Graf ve své podstatě jsou body spojené čárami, viz např. obrázek 1. Formálněji se jedná o matematickou strukturu tvořenou dvěma množinami: množinou **vrcholů** V (tedy naše body) a množinou neuspořádaných dvojic vrcholů E, kterým říkáme **hrany** (to jsou naše čáry). Graf G na vrcholech V a hranách E značíme $G = (V, E)$. V běžných grafech mohou být každé dva vrcholy spojené maximálně jednou hranou a každá hrana spojuje dva různé vrcholy (existují i struktury, kde to tak není, ale těmi se zatím nebudeme zabývat). Pro každý vrchol definujeme jeho **stupeň** jako počet hran, které z tohoto vrcholu vedou.

Ukažme si praktický příklad struktury reprezentovatelné grafem. Řekněme, že máme skupinu 5 lidí a někteří se mezi sebou znají a někteří ne. To může krásně reprezentovat graf. Na papír uděláme 5 teček (každá bude reprezentovat jednoho člověka) a mezi lidmi, kteří se navzájem znají, nakreslíme čáru. V tomto obrázku jsou tečky vrcholy našeho grafu a čáry jeho hrany. Pokud by tuto situaci kreslilo více lidí, pravděpodobně by výsledný obrázek vypadal u každého trochu jinak, přestože všechny zobrazují „to stejné“.

K tomu se nám hodí pojem **izomorfismus**. Ten se v matematice používá často a říká nám, že dvě matematické struktury jsou nějakým způsobem ekvivalentní, tedy stejné. Pro grafy specificky platí, že dva grafy jsou izomorfní, pokud mají grafy stejný počet vrcholů a můžeme pro každý vrchol z jednoho grafu zvolit jednoznačně (tedy pro žádné dva vrcholy z prvního grafu nezvolíme stejný vrchol z druhého grafu) vrchol z druhého a zachovají se nám hrany. Formálně řečeno grafy G_1 a G_2 jsou izomorfní, pokud pro každý vrchol v_i z grafu G_1 najdeme

vrchol z grafu G_2, který nazveme v'_i, tak, že žádné dva vrcholy z G_1 nespárujeme se stejným vrcholem a každé dva vrcholy v'_i a v'_j z G_2 mají mezi sebou hranu, právě když ji mezi sebou mají i vrcholy v_i a v_j. (Pro ty z vás, kdo znají pojem zobrazení, se jedná o prosté zobrazení mezi vrcholy těchto grafů, a protože oba grafy mají stejný počet vrcholů, tak se jedná dokonce o bijekci – ovšem pozor, ne každá bijekce je izomorfismus kvůli podmínce s hranami.) Tomu říkáme izomorfní zobrazení (neboli izomorfismus) jednoho grafu na druhý. Pokud se jedná o izomorfismus grafu sama na sebe, tak mu říkáme automorfismus. Každý graf má alespoň jeden automorfismus, kdy se každý vrchol zobrazí sám na sebe. Dále v textu budeme izomorfní grafy považovat za tentýž graf.



Obrázek 1: Izomorfismus grafu G_1 na graf G_2

Indukce

V matematice musíme platnost všech tvrzení dokazovat formálním důkazem. Pokud z jednoho tvrzení vyvodíme druhé pomocí logických operací (např. úpravou výrazů), tak tomu říkáme **přímý důkaz**, naopak pokud předpokládáme, že naše původní tvrzení neplatí, a poté tím získáme něco, co nedává smysl, tak tomu říkáme **důkaz sporem**. Krásným příkladem je například tvrzení, že neexistuje největší přirozené číslo. Pro spor předpokládáme, že takové číslo existuje, a označíme si ho n. Poté ale $n + 1$ je také přirozené číslo, které je očividně větší, a tedy n nemůže být největší a tím jsme dostali spor (neboli něco, co nedává smysl).

V teorii grafů však hojně využíváme ještě jeden typ důkazu, který není tak jednoduchý na pochopení, a to **důkaz indukcí**. V indukci musíme nejprve dokázat, že nějaké tvrzení platí pro nějaké počáteční číslo (zpravidla 1), poté předpokládáme, že platí pro všechna čísla menší nebo rovná $n \in N$, a z toho odvodíme, že tvrzení platí i pro $n+1$. Tím dokážeme, že tvrzení platí pro všechna $n \in N$ (nebo od našeho počátečního čísla): z první části důkazu víme, že to platí pro 1, ale pak pokud si zvolíme $n = 1$, tak z druhé poloviny důkazu zjistíme, že to platí pro 2, a pokud si zvolíme $n = 2$, tak to platí pro 3, a tak dále pro každé přirozené číslo. Dále v čísle si indukci ukážeme v praxi.

Typy grafů

V grafech často chceme nějakým způsobem hledat „cesty“ mezi vzdálenějšími vrcholy tak, že se budeme pohybovat mezi sousedními vrcholy po hranách, které je spojují. To si můžeme představit jako posloupnost, ve které se vždy střídají vrcholy a hrany: třeba posloupnost v_1, e_1, v_2, e_2, v_3 je pohyb, ve kterém jsme z vrcholu v_1 přešli přes hranu e_1 do vrcholu v_2 a z něj přes hranu e_2 do vrcholu v_3. Pokud se nám v tomto pohybu můžou opakovat hrany i vrcholy, nazveme ho **sled**, pokud se nám můžou opakovat pouze vrcholy, říkáme mu **tah**, a pokud se nám nic opakovat nemůže, říkáme mu **cesta**. Vynechali jsme ovšem možnost, kdy se můžou opakovat pouze hrany. *Zkuste se zamyslet proč.*

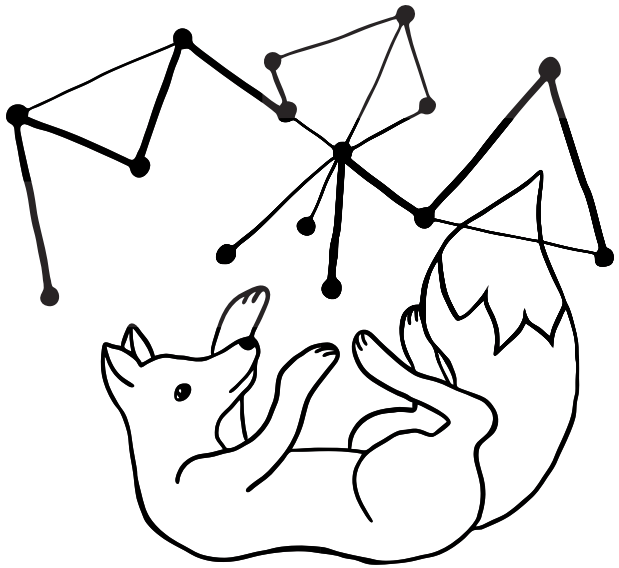
Abychom mohli projít stejnou hranou víckrát, museli bychom víckrát projít i vrcholy, které tato hrana spojuje, což jsme si ale zakázali. Proto nám tato situace nemůže nastat. Výše zmíněné posloupnosti hran a vrcholů nám jistým způsobem „spojují“ vrcholy v grafu. Pokud jsme schopni spojit nějaké dva vrcholy v grafu tahem, sledem nebo cestou, tak jsme schopni je spojit i zbylými dvěma typy posloupností. *Zamyslete se proč.*

Protože cesta je současně tahem i sledem, tak pokud umíme spojit vrcholy cestou, tak tvrzení jistě platí. Pokud je můžeme spojit sledem nebo tahem, tak nám stačí pouze najít vrcholy, které se opakují, a vynechat ty části posloupnosti mezi první a poslední návštěvou každého opakovaného vrcholu. Každý vrchol tak bude v posloupnosti nejvýše jednou. Zároveň jsme si už dokázali, že nemůžeme projít hranou víckrát, aniž bychom prošli vrcholem víckrát. Pokud tedy v posloupnosti nedochází k opakování vrcholů, neopakují se ani hrany a nalezli jsme cestu. Část grafu, která je takto spojená cestami, neboli část grafu, kde jsme se schopni z každého vrcholu dostat do každého jiného cestou, nazveme **komponenta**. Pokud je graf tvořený pouze jednou komponentou, říkáme mu **souvislý graf**.

Další často používané grafy jsou takzvaný **úplný** graf, nebo také **klika**, ve kterém je hrana mezi každými dvěma vrcholy, nebo naopak **prázdný** graf, který žádné hrany nemá. Další typy grafů jsou **cesta**, která se skládá pouze z jedné cesty spojující nějaké dva vrcholy, a **cyklus**, který je prakticky cesta, ale počáteční a konečný vrchol jsou ten stejný (můžeme si ho představit například jako pravidelný n-úhelník). Řekneme, že graf H je **podgraf** grafu G, pokud z grafu G můžeme smazat nějakou množinu hran a vrcholů (pokud smažeme vrchol, tak automaticky smažeme hrany z něj vedoucí) tak, abychom dostali graf H (nebo graf jemu izomorfní). Říkáme, že podgraf je **indukovaný**, pokud se skládá z nějaké množiny vrcholů původního grafu a všech hran mezi těmito vrcholy, které byly v původním grafu. Pokud je nějaký graf podgrafem G, tak někdy také říkáme, že ho graf G obsahuje. Poslední termín, který si v tomto díle představíme, je pojem **strom**. Jedná se o souvislý graf, který neobsahuje žádné cykly. Jeho vrcholy stupně 1 se nazývají **listy**. Ty mají speciální název, protože při práci se stromy často využíváme vlastnosti listů. Stromy mají jednu zajímavou vlastnost, kterou si dokážeme použitím indukce. Každý strom na n vrcholech má $n-1$ hran. *Zkuste si rozmyslet proč.*

Očividně graf na jednom vrcholu má 0 hran, tedy první část indukce je splněná. Nyní předpokládejme, že každý strom na n vrcholech má $n-1$ hran, a dokažme, že strom na $n+1$ vrcholech musí mít n hran. Podíváme se na to, jak vypadá strom na $n+1$ vrcholech. Zvolíme si nějaký jeho vrchol jako startovní bod a budeme dělat libovolné pohyby tak, že se nikdy nevrátíme stejným vrcholem, kterým jsme přišli. Tímto postupem se nikdy nemůžeme dostat do žádného vrcholu, ve kterém už jsme byli – tím bychom totiž našli cyklus, který ale strom nemůže obsahovat. Tedy s každým pohybem víme, že máme méně a méně vrcholů, do kterých může vést cesta, a tedy tento počet musí někdy klesnout na 0. Ocitli jsme se tedy ve vrcholu, ze kterého už nemůžeme jít nikam dál. Jediná hrana, kterou nesmíme využít, je ta, po které jsme přišli. Protože ale neexistuje jiná cesta, kterou bychom mohli využít, je právě tato hrana tou jedinou, která z „našeho“ vrcholu vychází. Nacházíme se tedy ve vrcholu stupně 1. Pokud tento vrchol a z něj vycházející hranu smažeme, tak dostaneme strom na n vrcholech (protože jistě nezkazíme souvislost a protože náš původní graf nemá cykly, tak ani po tom, co smažeme vrchol, v něm nemůžou být). Tento strom má dle indukčního předpokladu $n-1$ hran, a protože vrchol, který jsme smazali, měl stupeň 1, tak jsme smazali právě jednu hranu. Náš graf měl tedy původně n hran. Tím je důkaz dokončen.

Na závěr si představíme ještě několik typů grafů, se kterými se můžete často setkat. Řekneme, že graf je **k-regulární**, pokud každý jeho vrchol má stupeň k. Řekneme, že graf $\overline{G}$ je **doplněk** grafu G, pokud je na stejných vrcholech a má hrany právě mezi těmi vrcholy, kde je graf G nemá.



Úlohy

V tomto díle jsou řešení úloh snadno dohledatelná na internetu, zkuste ale prosím vymyslet své vlastní postupy a neposílejte nám řešení zkopírovaná z Wikipedie :D.

Úloha 1.1 [1b]: *Rozhodněte, jestli existují grafy s vrcholy stupňů 1, 1, 2, 3, 3 a 1, 1, 1, 1, 2, 2, 3. Pokud ano, můžou to být stromy?*

Úloha 1.2 [2b]: *a) Dokažte, že graf je strom právě tehdy, když mezi libovolnými dvěma vrcholy existuje právě jedna cesta. b) Dokažte, že graf je strom právě tehdy, když je souvislý, ale po odebrání libovolné hrany už není. U obou tvrzení musíte dokázat obě implikace, tedy že každý strom má tuto vlastnost a každý graf s touto vlastností je strom.*

Úloha 1.3 [1b]: *Dokažte, že pokud strom má vrchol stupně k, tak má alespoň k listů.*

Úloha 1.4 [2b]: *Pro která n existuje 7-regulární graf na n vrcholech?*

Úloha 1.5 [3b]: *Kolik různých cyklů (cykly nemusí být na všech vrcholech) obsahuje klika na n vrcholech?*

Úloha 1.6 [3b]: *Dokažte následující tvrzení: Existuje nekonečně mnoho grafů, které jsou izomorfní se svým doplňkem.*

Lukáš, Terka; troj.lukas@gmail.com
odevzdávejte do odevzdávátka

Téma 2 – Python

Díl 1: První a druhé krůčky

Začneme prvními krůčky: rychlým průletem proměnnými, čísly, řetězci, funkcemi a základním řízením toku. Druhé krůčky pak navážou čtyřmi tématy, ve kterých si vyzkoušíte s Pythonem pracovat: podmínky, rekurze, smyčky a list comprehensions.

Klasicky vám poskytneme úlohy. Protože někteří z vás už Python znají, úlohy mají lehkou a těžkou variantu. Lehká varianta je pro ty z vás, kteří jste se s jejím obsahem ještě nesetkali. V těžké variantě pro změnu dostanete nějaké omezení, které vám řešení úlohy ztíží. Příkladem: jedna z úloh vás žádá, abyste implementovali Eratosthenovo síto. Námět úlohy jsou cykly, takže byste se o vhodné obtížnosti měli rozhodnout podle toho, zdali znáte cykly, a nikoli podle toho, zda znáte Eratosthenovo síto. Body jsou u úloh zadány ve formátu lehká/těžká. U každé úlohy řešíte vždy právě jednu z variant, kterou si sami vyberete; pokud zvládnete jen lehkou, nevadí, body za ni stejně dostanete. Pokud je u úlohy pouze jedno číslo, můžete ji řešit vždy.

V rámci jednoho tématka bohužel nemáme čas projít každý koncept do takové hloubky, jak by se slušelo. Některé pojmy jen nakousneme a odkážeme na Wikipedii nebo na oficiální dokumentaci. Počítáme s tím, že si chybějící detaily dohledáte sami, ať už přes vyhledávač, dokumentaci na `https://docs.python.org/3/`, nebo přes ukecání nějakého chatovacího jazykového modelu. Samotná řešení úloh ale musíte vymyslet a napsat sami; jazykový model je tu od toho, abyste pochopili látku a dohledali detaily, ne aby za vás vygeneroval hotové řešení. Pokud vás bude trápit nějaký konkrétní koncept, zpomalte a dohledejte si na internetu nějaké příklady.

Než se ponoříme do prvních krůčků, krátce o tom, jak Python spustit. Pro instalaci stáhněte interpreter z `https://www.python.org/downloads/` (na Linuxu většinou stačí balíčkovač, např. `sudo apt install python3`); minimální verze pro tento výklad je 3.10. Skript spustíte příkazem `python3 soubor.py`; když do terminálu napíšete jen `python3`, naskočí interaktivní REPL[2], kde se píše výraz po výrazu a Python hned po stisku Enteru ukáže, co spočítal. Pro zkoušení ukázek z výkladu doporučujeme právě REPL.

Na Windows při instalaci z `https://www.python.org/downloads/` nezapomeňte zaškrtnout „Add Python to PATH“, jinak vám terminál `python` nenajde; v PowerShellu nebo `cmd.exe` se pak interpreter volá zkráceně `python` (ne `python3`). Komu se nechce psát do terminálu, má dvě pohodlné cesty: buď *IDLE*, malé prostředí, které se nainstaluje rovnou s Pythonem a v menu Start ho najdete pod tímto jménem; po spuštění se otevře okno s REPLem a přes `File` → `New File` se v něm dají psát i celé skripty. Druhá možnost je *Visual Studio Code* (`https://code.visualstudio.com/`) s oficiálním rozšířením „Python“ od Micro-

[2] Read–Eval–Print Loop, tedy smyčka „přečti výraz, vyhodnoť ho, vypiš výsledek“; chová se jako kalkulačka, do které se píše rovnou Python.

softu: po jeho instalaci stačí otevřít soubor s příponou `.py`, vlevo dole zvolit interpreter a zelený trojúhelník vpravo nahoře skript spustí. Interaktivní REPL se ve VS Code otevře přes paletu příkazů (Ctrl + Shift + P) volbou „Python: Start Terminal REPL“.

První krůčky

Python je interpretovaný jazyk s minimalistickou syntaxí. Bloky kódu se nevyznačují závorkami, ale odsazením: stejně odsazené řádky patří do jednoho bloku, větší odsazení otevírá vnořený blok. Interpret neřeší „kolik milimetrů“, ale přesně, z jakých znaků odsazení skládáte: všechny řádky v jednom bloku musí začínat *identickou* sekvencí mezer a tabulátorů, a vnitřní blok musí být odsazen hlouběji než řádek s hlavičkou (`if`, `for`, `def`, …). Mezera a tabulátor nejsou zaměnitelné, a nikdy by se neměly používat ve stejném souboru. Konvence PEP 8 proto doporučuje čtyři mezery na každou úroveň vnoření a tabulátory v odsazení nikdy nepoužívat.

```
if x > 0:
    print("vnejsi blok: kladne")
    if x > 10:
        print("vnoreny blok: velke kladne")
    print("zpatky ve vnejsim bloku (ne ve vnorenem)")
print("uz uplne mimo oba if")
```

Proměnné a typy. Proměnnou vytvoříme přiřazením, typ se odvodí z hodnoty. Hlavní vestavěné typy jsou `int`, `float`, `str`, `bool` a speciální `None`.

```
n = 42                  # int
pi = 3.14               # float
jmeno = "Maria"         # str (jednoduche apostrofy taky plati)
hotovo = True           # bool
nic = None              # absence hodnoty
```

Aritmetika: `+ - *`, dělení `/` (vrací float), celočíselné dělení `//`, zbytek `%`, mocnina `**`. Porovnání `== != < <= > >=` a logické spojky `and`, `or`, `not` fungují, jak byste čekali.

```
x = 7
y = 10
x < y                  # True, 7 je mensi nez 10
x == y                 # False, 7 se nerovna 10
3 <= x <= 9            # True, x lezi mezi 3 a 9 vcetne

s = "abc"
s == "abc"             # True, retezce jsou stejne
s != "ABC"             # True, "abc" a "ABC" nejsou stejne

vek = 20
ma_obcanku = True
vek >= 18 and ma_obcanku   # True, je plnolety a ma obcanku
```

Sekvence a slovníky. Indexujeme od nuly hranatými závorkami, záporný index počítá od konce. *Slicing* `xs[a:b]` vrátí podposloupnost od `a` (včetně) do `b` (mimo).

```
s = "Pythonek"
s[0]       # 'P'
s[-1]      # 'k'
s[2:5]     # 'tho'
len(s)     # 8

xs = [1, 2, 3, 4]
xs.append(5)                  # xs je ted [1, 2, 3, 4, 5]
xs[1:3]                       # [2, 3]

cisla = {"jedna": 1, "dve": 2}
cisla["jedna"]                # 1
cisla["tri"] = 3              # pridani polozky
```

Ne všechny typy se ale chovají stejně, když do nich chceme něco zapsat. Řetězce `str` a n-tice `tuple` (kulaté závorky) jsou *neměnné* (immutable): jakmile vznikly, jejich obsah se už nedá přepsat – operace, které „vrací změněnou verzi“, ve skutečnosti vyrábí upravenou kopii a původní nechávají být. Naproti tomu seznamy `list` a slovníky `dict` jsou *měnitelné* (mutable): jdou na místě upravit (přidat, smazat, přepsat položku) a všechna jména, která na ně ukazují, tu změnu uvidí. Rozdíl demonstruje následující příklad:

```
s = "Pythonek"
s[0] = "M"              # TypeError: 'str' object does not support item
    assignment
s = "M" + s[1:]         # 'Mythonek' -- vyrobili jsme NOVY retezec

t = (1, 2, 3)
t[0] = 99               # TypeError: 'tuple' object does not support item
    assignment

xs = [1, 2, 3]
ys = xs                 # ys je tentyz seznam jako xs
xs[0] = 99              # zmena na miste
ys                      # [99, 2, 3] -- ys vidi tutez zmenu

cisla = {"jedna": 1}
cisla["dve"] = 2        # zmena na miste, slovnik je ted {"jedna": 1, "dve":
     2}
```

U neměnných typů tedy přiřazení `s = "M" + s[1:]` jen „přesměruje“ jméno `s` na nově vyrobený řetězec; u měnných stačí jediný odkaz, abyste obsah přepsali „na místě“ a všichni ostatní to viděli.

Pro skládání textu se hodí *f-stringy*: řetězcový literál začíná předponou `f` těsně před otevíracími uvozovkami a uvnitř zápisu Python ve složených závorkách `{...}` vyhodnotí libovolný výraz a jeho výsledek převede na text stejným způsobem jako při volání `str(...)` – může jít o proměnnou, aritmetiku, volání funkce, indexování a podobně. Za výrazem lze (jako u metody `str.format`) uvést dvojtečku a *specifikátor formátu*: např. `f"{pi:.2f}"` zaokrouhlí `float` na dvě desetinná místa, `f"{n:04d}"` celé číslo doplní nulami na čtyři znaky. Chcete-li do výsledného textu dostat samotnou složenou závorku, znak v řetězci zdvojte: `f"{{ ... }}"` ve výstupu vyrobí `{ ... }`. Od Pythonu 3.8 lze při ladění psát `f"{x=}"`; program doplní jméno proměnné, znak `=` a *repr* hodnoty.

```
x, pi, n = 7, 3.14159, 3
f"x = {x}, druha mocnina = {x * x}"
f"{pi:.2f}"                 # "3.14"
f"{n:04d}"                  # "0003"
f"{{ bez interpolace }}"    # "{ bez interpolace }"
f"{x=}"                     # "x=7"
```

Funkce. Funkci zavádíme klíčovým slovem `def`, hodnotu vrátíme `return`; pokud `return` chybí, funkce vrátí `None`. Parametr může mít *výchozí hodnotu*, která se použije, když volající argument neuvede; pak ho stačí psát jen pro případy, kdy chcete jinou hodnotu než výchozí. Argumenty se navíc dají předávat dvěma způsoby: *pozičně* v tom pořadí, jak jsou parametry napsané, nebo *jménem* ve tvaru `jmeno_parametru=hodnota`.

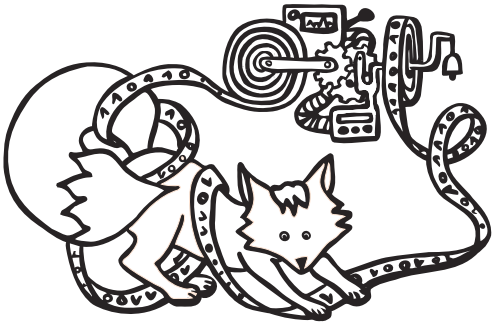
```
def pozdrav(jmeno, hlasite=False):
    text = f"Ahoj, {jmeno}!"
    if hlasite:
        return text.upper()
    return text

pozdrav("Marie")                        # 'Ahoj, Marie!'
pozdrav("Marie", True)                  # 'AHOJ, MARIE!'
pozdrav(jmeno="Marie", hlasite=True)    # totez, parametry jmenem
```

Větvení a smyčky. Z jiných jazyků poznáte `if`/`elif`/`else`, `while` a `for`; místo závorek za podmínkou je dvojtečka a tělo se odsadí. Příkaz `for` je v Pythonu *foreach*, prochází přes položky kolekce. Pro průchod přes čísla se hodí `range(start, stop, step)`.

```
i = 0
while i < 5:
    if i % 2 == 0:
        print(i, "sude")
    else:
        print(i, "liche")
    i += 1

for slovo in ["jedno", "dve", "tri"]:
    print(slovo)
```



Vstup a výstup. Každý spuštěný program má dva pomyslné kanály, kterými komunikuje s okolím: *standardní vstup* (`stdin`) a *standardní výstup* (`stdout`). Když skript pustíte z terminálu, čtení ze `stdin` typicky znamená „počkat, až uživatel něco napíše a stiskne Enter“, a zápis do `stdout` znamená „vypsat to do toho samého terminálu“. Oba kanály jdou přesměrovat: `python3 skript.py < vstup.txt` dodá `stdin` ze souboru, `python3 skript.py > vystup.txt` naopak `stdout` přesměruje do souboru; obojí jde použít naráz (`python3 skript.py < vstup.txt > vystup.txt`). Příkaz `print` je standardní cesta na `stdout` (argumenty oddělí mezerou a přidá konec řádku), `input` přečte ze `stdin` jeden řádek a vrátí ho jako řetězec; pro číslo ho převedeme funkcí `int` či `float`:

```
print("Ahoj,", "svete!")          # vypise na stdout: Ahoj, svete!
n = int(input("n = "))            # vypise vyzvu, pak prevezme cislo ze
    stdin
```

U úloh, které ze vstupu čtou víc hodnot, počítejte s tím, že každé volání `input` přečte jeden řádek: není-li u úlohy řečeno jinak, očekávejte každou hodnotu na samostatném řádku (ne několik čísel na jednom řádku oddělených mezerou).

Pojmenování. Python rozlišuje malá a velká písmena (`Promenna` a `promenna` jsou různé). Kolem pojmenovávání má Python ustálené zvyky (viz PEP 8 – styl pojmenovávání[3]): proměnné a funkce v `snake_case` (slova malými písmeny oddělená podtržítkem, např. `pocet_pruchodu`), třídy v `CamelCase` (např. `Bod`), konstanty velkými písmeny (`MAX_HLOUBKA`). Navíc je zvykem psát identifikátory bez diakritiky. Je to sice technicky povolené, ale některá prostředí neumí diakritiku správně zobrazit, takže je lepší být s použitými symboly konzervativní.

Úloha 1.1 [2b]: *Na závěr prvních krůčků si zkuste malý miniprojekt: napište program, který se zeptá na jméno a věk uživatele, pozdraví ho („Ahoj, Alice!“) a podle věku vypíše, jestli je „plnoletý/á“, nebo „neplnoletý/á“. Pokud uživatel zadá záporný věk nebo zjevný nesmysl (např. víc než* 130*), místo pozdravu vypište zprávu o neplatném vstupu. Pokud byl věk zadán správně, vypište, co si myslíte, že už uživatel prožil, a vyvěštěte, co ho v životě ještě čeká.*

[3] `https://peps.python.org/pep-0008/#naming-conventions`

Druhé krůčky

Podmínky. Kromě `if/elif/else` z prvních krůčků má Python ještě trojčlenný (ternární) výraz `a if c else b`, který vrací `a`, pokud `c` platí, jinak `b`. Od Pythonu 3.10 přibylo `match` pro porovnávání se vzory:

```
match cislo:
    case 0:           return "nula"
    case n if n < 0:  return "zaporne"
    case _:           return "kladne"
```

Pravdivostní hodnotu má v Pythonu skoro všechno: `0`, `0.0`, `""`, `[]`, `{}`, `None` jsou *falsy* (chovají se jako `False`), zbytek je *truthy*. Operátory `and` a `or` vracejí svůj operand, ne jen `True`/`False`, takže `x or default` je idiom „vezmi `x`, pokud má smysl, jinak `default`“.

Úloha 1.2 [3b/3b]: *Následující úlohy by měly otestovat vaši schopnost použít podmínky. Pokud podmínky už znáte, vyřešte úlohy bez `if`, `elif`, `else`, `match` i bez ternárního `x if c else y`.*

1. *[1b/1b] Načtěte tři celá čísla a, b, c a vypište největší z nich. Pokud je největších víc (např. $a = b > c$), vypište místo čísla text „vice stejnych maxim“.*

2. *[1b/1b] Načtěte celé číslo „hodina“ z intervalu 0 až 23 a vypište, jaká část dne to je: „noc“ pro $[0, 5]$, „rano“ pro $[6, 11]$, „odpoledne“ pro $[12, 17]$ a „vecer“ pro $[18, 23]$. Pro čísla mimo tento rozsah vypište „neplatny cas“.*

3. *[1b/1b] Naprogramujte hru na hádání čísla. Počítač si zvolí číslo 1 až 100, hráč zadává tipy ze vstupu; po každém tipu vypište „moc málo“, „moc hodně“, nebo „trefa!“ a v posledním případě hru ukončete.*

Rekurze. Rekurze nazýváme typ chování programu, kde funkce volá sama sebe. Klasický faktoriál:

```
def factorial(n):
    if n <= 1:
        return 1
    return n * factorial(n - 1)
```

Vždy musí být dvě části: kotvící podmínka (`n <= 1`), která rekurzi zastaví, a rekurzivní volání s „menší“ úlohou. Bez kotvy (nebo když se k ní nikdy nedostaneme) Python po asi tisíci voláních ukončí program chybou `RecursionError`. Limit lze zvýšit voláním `sys.setrecursionlimit(10000)`, ale obvykle je lepší si algoritmus rozmyslet znovu.

Úloha 1.3 [4.5b/4.5b]: *Následující úlohy by měly otestovat vaši schopnost použít rekurzi. Pokud rekurzi už znáte, vyřešte úlohy bez ní.*

1. *[1b/1b] „Zploštěte“ seznam obsahující libovolně vnořené seznamy.*
2. *[1b/0.5b] Setřiďte pole prvků pomocí Quicksortu[4] nebo Mergesortu[5]. Na konkrétní variantě nezáleží (u Quicksortu třeba na tom, jak volíte pivota); stačí libovolná.*
3. *[0.5b/1b] Vyhodnoťte Ackermannovu funkci[6] $Ack(n, m)$ pro zadaná n, m.*
4. *[2b/2b] Adaptivní lichoběžníková integrace[7]: pro funkci f a interval $[a, b]$ spočítejte přibližně $\int_a^b f(x)\,dx$. Porovnejte lichoběžníkovou aproximaci na celém intervalu s aproximací na jeho dvou polovinách. Pokud se dost shodují, vraťte jejich součet, jinak rekurzivně integrujte obě poloviny zvlášť a jejich výsledky sečtěte.*

Cykly. `for` a `while` z prvních krůčků uvidíme i tady, ale s pár pythonovskými idiomy navrch. Když potřebujeme zároveň index i hodnotu, použijeme `enumerate`; pro paralelní procházení dvou (a více) seznamů je `zip`.

```
for i, x in enumerate(["a", "b", "c"]):  # (0,"a"), (1,"b"), (2,"c")
    print(i, x)

for x, y in zip([6, 7, 9], ["a", "b", "c"]):
    print(x, y)                          # 6 a / 7 b / 9 c
```

`while` se chová jako v jiných jazycích. Příkaz `break` uvnitř těla okamžitě ze smyčky vyskočí (a pokračuje se až za jejím tělem); to se hodí v tradičním vzoru `while True: …break`, který Pythonu nahrazuje `do-while`: smyčka běží napořád, dokud něco uvnitř těla neřekne „a teď dost“ (typicky čtení vstupu „do konce souboru“).

```
# "do-while" styl: aspon jednou se provede telo smycky
while True:
    s = input("Zadej kladne cislo (0 konci): ")
    n = int(s)
    if n == 0:
        break
    if n < 0:
        print("To neni kladne cislo.")
        continue
    print("Dvojnasobek:", 2 * n)
```

[4] https://cs.wikipedia.org/wiki/Quicksort
[5] https://cs.wikipedia.org/wiki/Mergesort
[6] https://cs.wikipedia.org/wiki/Ackermannova_funkce
[7] https://en.wikipedia.org/wiki/Adaptive_quadrature

Úloha 1.4 [3b/3.5b]: *Následující úlohy by měly otestovat vaši schopnost použít cykly. Pokud cykly už znáte, vyřešte úlohy bez* `for` *a* `while`*.*

1. *[1b/1b] Zakódujte řetězec metodou „run-length[8]“: např. „aaabbc“ odpovídá „a3b2c1“. Napište kódování i dekódování.*

2. *[1b/0.5b] Vypište všechna prvočísla menší než N pomocí Eratosthenova síta[9].*

3. *[1b/2b] Spočítejte jeden krok Conwayovy Game of life[10] na konečné mřížce $n \times m$. Buňky za okrajem mřížky považujte za permanentně mrtvé.*

List comprehensions. List comprehension je hutný způsob, jak z jedné posloupnosti spočítat druhou. Místo `for`-cyklu s `append`…

```
result = []
for x in xs:
    if x % 2 == 0:
        result.append(x * x)
```

… napíšeme jediný výraz:

```
result = [x * x for x in xs if x % 2 == 0]
```

Tři základní tvary jsou *map* (`[f(x) for x in xs]`), *filter* (`[x for x in xs if p(x)]`) a *kartézský součin* (`[(x, y) for x in xs for y in ys]`). Vnořením vznikne matice: `[[f(i, j) for j in J] for i in I]`. Stejnou syntaxí se staví množiny (`{...}`) a slovníky (`{k: v for k, v in ...}`).

Úloha 1.5 [3b]: *Následující úlohy by vás měly seznámit s list comprehension (případně generátorovými či slovníkovými výrazy). Každá z nich má přirozeně tvar jediné comprehension.*

1. *[1b] Pro daný seznam celých čísel vraťte seznam čtvrtých mocnin těch jeho prvků, které jsou liché.*

2. *[1b] Pro zadané n sestavte násobilku $n \times n$ jako seznam seznamů, tedy matici, jejíž prvek na souřadnicích (i, j) je $i \cdot j$.*

3. *[1b] Pro zadané N vypište všechny pythagorejské trojice (a, b, c) splňující tyto podmínky $1 \leq a \leq b \leq c \leq N$ a $a^2 + b^2 = c^2$.*

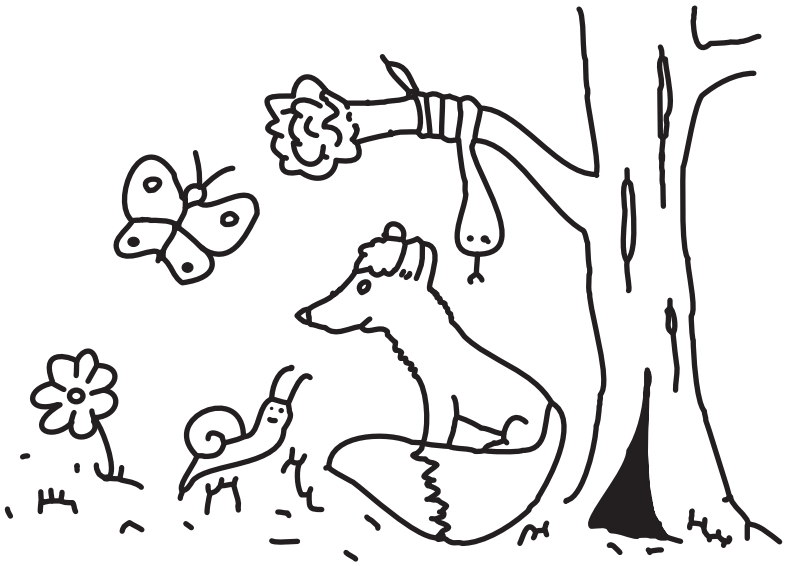
[8] https://cs.wikipedia.org/wiki/RLE
[9] https://cs.wikipedia.org/wiki/Eratosthenovo_síto
[10] https://cs.wikipedia.org/wiki/Hra_života

Comprehensions ve velkém. Předchozí část list comprehensions stačila pro úlohy, kde se s každým prvkem dělá stejná věc nezávisle. Tato sekce je pro řešitele, kteří hledají výzvu: ukáže, kam až se dá comprehension protáhnout, když potřebujeme něco akumulovat nebo provázat iterace mezi sebou. Klíčový trik spočívá v tom, že výraz tvaru `(vyraz for x in xs if podminka)` je *generátorový výraz*: syntakticky vypadá jako list comprehension, ale neukládá celý seznam do paměti, hodnoty vyrábí postupně až při průchodu. Když takový generátor zabalíme do funkcí `sum`, `any`, `all`, `max`, `min` nebo `math.prod`, dostaneme výpočet jednoho skaláru.

Test prvočíselnosti přes `all` demonstruje, jak zkombinovat generátorový výraz s `all`:

```
def is_prime(n):
    return n >= 2 and all(n % d for d in range(2, int(n**0.5) + 1))
```

Pro každého kandidáta `d` z rozsahu vrátíme `n % d`; nula znamená „dělí“, nenula „nedělí“. Příkaz `all` chce všechny pravdivé, takže výsledkem je „žádný dělitel z 2 až $\lfloor\sqrt{n}\rfloor$ nedělí `n`“, čili prvočíselnost.



Úloha 1.6 [5b]: *Tyto úlohy zkouší, kam až comprehensions dosáhnou. Každou z nich vyřešte jediným výrazem, bez pomocných cyklů, mezivýsledků a přiřazení: buď samotnou comprehension, nebo generátorovým výrazem obaleným do jediného volání* `sum`, `any`, `all`, `max`, `min` *nebo* `math.prod`.

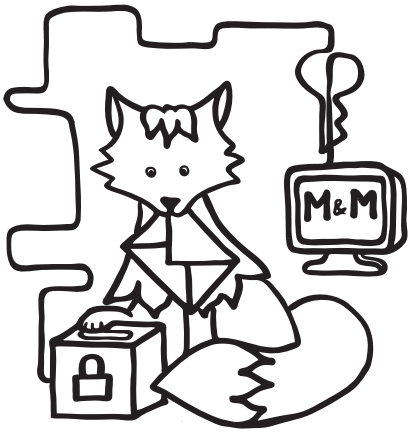
1. *[1b] Rozložte přirozené číslo $n \geq 2$ na prvočinitele s násobnostmi: vraťte seznam dvojic (p, e), kde p jsou prvočísla a $n = \prod_p p^e$. Např. pro $n = 12$ vraťte $[(2, 2), (3, 1)]$.*

2. *[2b] Pro zadaný seznam kořenů $[r_0, r_1, \ldots, r_{n-1}]$ vraťte koeficienty monického*[11] *polynomu $\prod_i (x - r_i) = \sum_k a_k x^k$ jako seznam $[a_0, a_1, \ldots, a_n]$. Např. pro kořeny $[1, 2]$ vraťte $[2, -3, 1]$ (tedy polynom $x^2 - 3x + 2$).*

3. *[2b] Pro pravidelnou mřížku $W \times H$ komplexních čísel c (rovnoměrně rozesetých např. přes obdélník $[-2, 1] \times [-1{,}5, 1{,}5]$) vraťte 2D pole počtů iterací do „úniku" Mandelbrotovy*[12] *posloupnosti $z_0 = 0$, $z_{n+1} = z_n^2 + c$, tedy nejmenšího i, pro které $|z_i| > 2$. Pokud takové i nenajdete do iterace i_{max}, vraťte i_{max}.*

Pokud v tématku najdete chybu, něco vám nebude jasné nebo budete mít jakoukoli připomínku, napište na M&M Discord do channelu `python`.

Jan Koška; `jan.koska@email.cz`
odevzdávejte do odevzdávátka



[11] Monický polynom je polynom, jehož vedoucí koeficient (u nejvyšší mocniny x) je roven jedné.
[12] `https://cs.wikipedia.org/wiki/Mandelbrotova_množina`

Téma 3 – Operační systémy

Díl 1: Programy

Toto je první ze série tématek pojednávajících o vybraných aspektech fungování počítačů, a to především z pohledu softwaru a toho, čemu říkáme *jádro operačního systému*. Tato témata jsou jistě nevšední pro časopis jako je M&M a vůbec obzor nás teoretických informatiků, ale právě to je na škodu. Shledávám totiž, že i takto zdánlivě ryze inženýrský obor stojí na matematice, která je pro něj stejně nezbytná jako fyzika pro návrháře hardwaru o úroveň níže. Ani teoretické informatice se dlouho po jejím vzniku nedostávalo od zaběhlých oborů té úcty, kterou si zasluhuje. Teprve po tom období skvělých vynálezů a zázračných pokroků, kdy programátoři začali narážet na skutečné překážky vyvolávající fundamentální otázky, se mohla prosadit jako opravdový předmět zájmu. Stejně tak v návrhu operačních systémů a v příbuzných programátorských odvětvích jsou už pomyslné karty rozdány a hlavním hybatelem je jejich tvrdá „matematizace“ (což je samozřejmě umocněno současnou předávkou „pouhého programování“ do rukou strojů). Proto bych rád zaujal širší okruh čtenářů, než jen ty, kteří již přirozeně tíhnou k informatice nebo přímo k systémovému programování. Jedině tak budeme mít šanci objevovat dál tento krásný obor, jehož klíčové otázky jsou už teď především matematického (mezioborového) rázu a obejdou se zcela bez pachuti psaní instrukcí v assembleru či ladění céčkového kódu. Přeji příjemné čtení a čistou mysl.

Čemu říkáme programy

Všichni používáme počítače, a tudíž máme všichni nějakou představu o jejich fungování. Některé technikálie se nechtěně dostanou i k běžnému uživateli – především vědomost, že software obsahuje chyby – od jiných je zase téměř dokonale odstíněn (ovšem neobsahují-li chyby). Chci, aby se naše povídání ubíralo způsobem postupného odkrývání těchto „samozřejmých“ aspektů a zkoumání jejich nesamozřejmosti.

Tedy začněme u základního pojmu programu. Pokud jste si již někdy napsali sebemenší skript v Pythonu či podobném jazyce, tušíte, že za grafickým oknem na obrazovce, kterému běžně říkáme program, se skrývá skutečný „program“ (kód), který se v principu neliší od onoho Python skriptu, jen je o malinko složitější[13]. Záleží na úhlu pohledu, ale v konečném důsledku o moc složitější být skutečně nemůže, neboť samotný procesor počítače, který programy spouští, o moc složitějšímu „jazyku“ nerozumí. Dokonce pracuje s ještě mnohem jednodušším jazykem, než je Python, s tzv. strojovým kódem, nicméně pro naše pochopení nyní stačí, že v hrubých rysech vypadá běh libovolného programu podobně jako běh programu Pythonového – program sestává ze stavu (proměnných) a jednoduchých příkazů,

[13] Nemá-li čtenář skutečně žádnou zkušenost s algoritmizací, nemusí smutnit, stačí otočit na stranu 9 a dostane se k tématku o Pythonu, které v tomto ročníku bude rovněž vycházet. Pro další čtení stačí mít skutečně jen základní představu o Pythonu (nebo jiném imperativním jazyce) a o programátorském pojmu funkce. To je bohatě pokryto už v tomto čísle.

které jsou popořadě vykonávány, pokud zrovna neskákají mezi místy v programu.

Ve většině našich úvah tak bohatě postačí přemýšlet nad libovolnými počítačovými programy jako nad Python programy[14], pokud máme na paměti, že v žádném počítači není Python skutečně „zadrátovaný“, nýbrž je překládán do strojového kódu[15], který závisí na typu procesoru a který bychom museli při programování konkrétního operačního systému důkladně znát (což je však jen technická dovednost). *Stejně tak očekáváme řešení programovacích úloh a problémů v Pythonu, přičemž jej bereme pouze jako prostředek vyjádření konceptů, o které nám jde především. (Často půjde o „hypotetické“ programy, které ani nelze spustit.)*

Jeden počítač, mnoho programů

Dosud jsme mluvili o programech samostatně, nicméně víme, že počítače, s nimiž se setkáváme, umožňují spouštět mnoho (prakticky libovolně) programů naráz. To je první pozoruhodná vlastnost, jíž se budeme zabývat. Postupně totiž uvidíme, že mnoho zbylých rolí OS (operačního systému) tento tzv. multitasking vlastně předznamenává. Ale v čem vlastně tkví problém? Nešlo by prostě všechny naše programy propojit jakoby do jednoho velkého Python programu? Zkusme si to:

Problém 1.1: *Samoobslužný kiosek ve fastfoodu má dva displeje, jeden z každé strany. Chceme ho ale řídit pouze jedním počítačem. Máme k dispozici funkci* `wait_for_order(display_number)`*, která pro daný displej (1 nebo 2) počká, než zákazník dokončí objednávku, načež objednávku vrátí. Dále máme funkci* `pass_to_kitchen(order)`*, která získanou objednávku předá kuchyni.*

Nejdříve napišme dva (vskutku kratičké) programy, které budou opakovaně obsluhovat jednotlivé displeje, tj. vždy počkají na objednávku a pak ji odešlou do kuchyně (programy budou spuštěny nezávisle, „najednou“, jak jsme zvyklí).

Nyní napišme jediný, opět jednoduchý program, jenž bude dělat to samé, jako tyto dva programy běžící najednou, tj. bude obsluhovat oba displeje. Pokud se vám to nepodaří, vysvětlete, v čem je problém. Šlo by zadání smysluplně upravit tak, aby to šlo? (A vyhneme se tak skutečně původnímu problému?)

I kdyby tedy teoreticky šlo napsat jeden program, který by se choval jako dva či více předem daných programů, narazíme na problém udržitelnosti.

- Jednak jako uživatelé nechceme, aby byla množina současně běžících programů na našem počítači předem daná (chceme spouštět a ukončovat různé programy).

[14] Slovo „programy“ v tomto tématku používáme v několika lehce odlišných významech, abychom ulehčili terminologii. Především se o ně v tomto čísle zajímáme ve smyslu *vláken* (threads). Tento a jiné pojmy však představíme v souvislostech příště, resp. až se k nim přirozeně propracujeme.

[15] Překladače a interprety programovacích jazyků jsou speciálními druhy programů, které tvoří samy o sobě bohatý obor.

- Druhak jako vývojáři těchto programů nechceme na ostatní brát ohled, naopak by se nám hodilo z pohledu jednoho programu stále uvažovat tak, jako kdybychom měli celý počítač pro sebe.

Jak vytvoříme takovou „iluzi“ běhu více programů naráz, aby obě strany byly šťastné?

Kontext programu

Programy se mezi sebou navzájem budou „přepínat“. Nejdřív poběží chvíli jeden, pak druhý, třetí, načež po nějaké době zase ten první a tak dále. Takové přepnutí je ale zřejmě složitější než třeba volání funkce. Jak jsme řekli, nechceme kvůli tomu drasticky měnit strukturu programu. Tedy musíme být schopni vyskočit z libovolného místa v programu (třeba i uprostřed nějakého výpočtu) a pak se na stejné místo zcela neporušeně vrátit. Jaké informace o programu si pro takové navrácení musíme zapamatovat?

Čtenář, který již něco ví o tom, jak fungují interprety programovacích jazyků, má nejspíš jasno. Pokusme se k tomu ale dojít od začátku. Nepotřebujeme toho mnoho. Pro jednoduchost začněme u programu bez funkcí. Takový program je jen řetězec nějakých výpočtů s proměnnými, popřípadě podmíněných příkazů a cyklů. Například:

```
1 n = 1
2 while True:
3     m = n
4     while m != 1:
5         if m % 2 == 0:
6             m = (m // 2)
7         else:
8             m = (3*m + 1)
9     n = (n + 1)
```

Příklad 1: Kód, jenž nevolá funkce, tedy nikdy „nevyskakuje pryč“

Protože není zřejmé, po kolika krocích vnitřní cyklus (řádek 4) vždy doběhne, slušelo by se umět po tomto řádku vždy „vyskočit“ (a vrátit se, až na nás zase přijde řada). Aby výpočet zůstal neporušen, musíme si uložit nějaký „stav programu“. Například v desátém průběhu vnějšího cyklu po druhém průběhu vnitřního cyklu si musíme zapamatovat (ověřte hodnoty):

- Pokračuj na řádku 5.
- Pokračuj s hodnotami n = 10, m = 16.

Tedy stačí si kdykoliv uložit *číslo řádku* a *stav proměnných*. (Ověřte, že toho nepotřebujeme víc.) Tomuto balíčku pak říkáme kontext.

Úloha 1.2 [2+1b]: *Napište program, který začne s prázdným seznamem a postupně jej (pomocí nějakého algoritmu) naplní prvními 100 čísly nějaké vaší oblíbené celočíselné posloupnosti. Nevytvářejte ani nevolejte přitom žádné funkce (pro přidání prvku do seznamu můžete použít např.* `seznam += [prvek]`*). Pak popište celkový kontext programu v bodě, kdy se délka seznamu právě zvýšila na 7. (+1 bod za hezkou posloupnost!)*

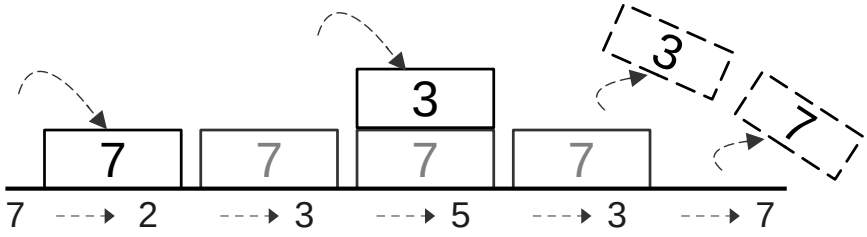
Zásobník volání

```
1 def f(a):
2     b = (a + 1)
3     g(b + 1)
4 def g(a):
5     if a % 2 == 1: # "a je liche"
6         f(a)
7 f(0)
```

Příklad 2: Kód, jenž využívá funkce a rekurzi, a vyžaduje tak bohatší kontext

Nyní začněme uvažovat o programech s funkcemi. Funkce lze volat z různých míst (i z nich samotných) s různými argumenty. Rozmysleme si, proč nám zde pro zachycení kontextu již nestačí pouhé číslo řádku a stav proměnných. Především by tento kontext opomíjel informaci, odkud jsme funkci zavolali, která je nutná k tomu, abychom se z ní mohli někdy vrátit. Neboť funkce mohou volat další funkce, toto v obecnosti vede na datovou strukturu zásobníku (stack), která funguje následovně:

- Kdykoliv zavoláme nějakou funkci, položíme navrch zásobníku číslo řádku, odkud jsme ji zavolali. (Tuto konvenci pro smysluplnost používáme zde v příkladech, ale samozřejmě tím nemyslíme, že se chceme navracet na totéž místo, odkud jsme funkci zavolali, ale „těsně za něj“. V praxi se skutečně používá číslo následující instrukce.) Tomuto „číslu řádku“ obecně říkáme *návratová adresa*.
- Kdykoliv se vracíme z nějaké funkce (doběhla na konec, nebo byl proveden `return`), odstraníme návratovou adresu shora zásobníku a vrátíme se na ni.

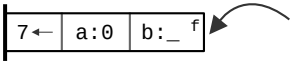


Zbývá už jen jeden detail. Chápeme, že dvě proměnné, které se stejně jmenují, ale nacházejí se každá v jiné funkci, spolu zřejmě nemají nic společného. Neočekáváme, že jedna funkce bude měnit obsah proměnné v jiné funkci (hovoříme o čistě *lokálních proměnných*, na rozdíl od *globálních*, jejichž použití bychom v Pythonu ohlašovali příkazem `global`). Při ukládání hodnot proměnných do kontextu tedy již nestačí říkat např. „proměnná `a` má hodnotu 10“, ale „proměnná `a` z funkce `f` má hodnotu 10“. Nicméně ani toto zcela nestačí. Obecně chceme, aby funkce mohla volat sebe samou (třeba i skrze jiné funkce). S tímto principem zvaným *rekurze* se čtenář jistě již setkal anebo setká při řešení mnoha různých algoritmických problémů[16]. Používáme-li v takové rekurzivní funkci proměnné, je podobně nežádoucí, aby si jednotlivá rekurzivní „zavolání“ téže funkce navzájem sahala do proměnných (do těch zahrnujeme i argumenty)!

Nejvhodnější je opět pořídit si zásobník, na který si funkce budou moci odkládat hodnoty svých lokálních proměnných, aby se k nim po návratu z nimi volaných funkcí mohly vracet. Všimneme si, že chování tohoto zásobníku je vlastně totožné se zásobníkem návratových adres, který již máme. Stačí nám tedy zásobník jediný, na který si budeme odkládat všechno, obsažené v tzv. *rámcích* (stack frame). Každý rámec odpovídá nějaké zavolané funkci, která se dosud nevrátila, a obsahuje jednak návratovou adresu, jednak lokální proměnné funkce včetně argumentů. Všimněme si, že při přidání nového rámce na zásobník (tj. zavolání funkce) musí jednu část informací dodat volající, druhou si pak spravuje sám volaný. Pro úplnost dodejme, že v praxi může mít tento obecný *zásobník volání* i další účely, jako např. předávání návratových hodnot (funkce při návratu nahradí svůj rámec vracenou hodnotou).

Ukážeme si chování zásobníku na příkladu. Sledujeme běh ukázkového programu z Příkladu 2 (výše).

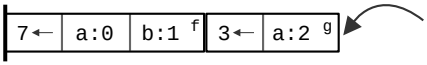
1. Na začátku je zásobník prázdný.
2. Program začne voláním `f(0)` (ř. 7). Na zásobník tedy umístí nový rámec funkce `f` a předvyplní návratovou adresu na 7 a argument `a` na hodnotu 0.



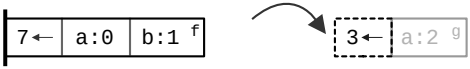
3. Jsme ve funkci `f`. Na vrchu zásobníku (v našem rámci) vidíme návratovou adresu, argument `a` a dosud neinicializovanou (nevyplněnou) proměnnou `b` (standardně se rámce neroztahují a nesmršťují, nýbrž jejich velikost je již předem vypočítaná tak, aby pojala všechny proměnné dané funkce). Proměnné `b` je vzápětí přiřazena hodnota $a + 1 = 1$.

[16] Nezasvěcený čtenář se s rekurzí může seznámit i v Python tématku v tomto čísle na straně 15.

4. Voláme `g`. Navrch zásobníku přibyde nový rámec s návratovou adresou 3 a argumentem `a` s hodnotou $b + 1 = 2$.



5. Podmínka (ř. 5) selže a funkce `g` se tedy vrací. Ještě než odstraní svůj rámec ze zásobníku, podívá se do něj, že se má vrátit na (těsně za) řádek 3.



6. Za řádkem 3 již funkce `f` končí, tedy vrací se i ta, a to stejným způsobem – podívá se na návratovou adresu ve svém rámci (tj. 7) a odstranivši jej ze zásobníku se na ni vrátí. To je již konec programu. Zásobník je opět prázdný.

A je to! Tedy pro zachycení kontextu obecného programu nám stačí znát: číslo řádku, zásobník volání (na kterém žijí mj. lokální proměnné) a hodnoty globálních proměnných (žijících mimo funkce). (Přesvědčte se o tom.)

Úloha 1.3 [2b]: *Co kdybychom na řádku 7 volali `f` s argumentem 1? Popište, jak vypadá celý zásobník (tj. všechny rámce) ve chvíli, kdy jeho výška dosáhne 1000 rámců.*

Úloha 1.4 [1+1b]: *Jak bychom mohli omezit používání funkcí v programu tak, abychom nepotřebovali zásobník volání (a vystačili si nadále jen s jednoduchým kontextem)? (1 bod) A co kdybychom si ho dovolili, ale pouze pro návratové adresy a ne lokální proměnné? (1 bod)*

Úloha 1.5 [3+1b]: *Zásobník pochopitelně nemůže být neomezený. I kdybychom si ho mohli na krátko „zvětšit“, vždy bude existovat riziko, že nám dojde paměť a program nebude moci smysluplně pokračovat. V praxi bývá velikost zásobníku pevně daná a programátoři musejí ověřit, že ji program nepřekročí. (Často za to může rekurze, ale ne vždy, máme-li obecně příliš velké rámce nebo příliš malý zásobník!) Pro následující funkci určete (maximální) spotřebu paměti zásobníku (počítejme prostě rámce) v závislosti na počátečních argumentech (3 body). Poté funkci ekvivalentně přepište bez rekurze tak, aby dovolovala prakticky libovolně velké vstupy a nemuseli jsme spotřebu paměti řešit (1 bod).*

```
1 def f(x, y): # x, y jsou prirozena cisla
2     if y == 0:
3         return 0
4     elif y % 2 == 0: # "y je sude"
5         return 2 * f(x, y // 2) # // = celociselne deleni
6     else:
7         return 2 * f(x, y // 2) + x
```

Problém 1.6: *Některé rekurzivní funkce jsme schopni snadno přepsat na nerekurzivní. Toto jednoduché, avšak významné povšimnutí dovoluje programovacím jazykům být mnohem efektivnější. Pokuste se charakterizovat co nejobecněji tuto třídu funkcí (co musejí splňovat) a popište algoritmus převodu na nerekurzivní tvar (stačí slovy, avšak dostatečně přesně). Nápověda: Funkce z předchozí úlohy není takovou funkcí. Cílem zadání není vymýšlet závratně složité algoritmy, naopak.*

Závěr

Snad čtenář při zkoumání kontextu nezapomněl, proč jsme se o něj vůbec začali zajímat. Nyní totiž máme vše potřebné ke spouštění mnoha programů „naráz“. Máme-li k dispozici funkce `save_context()` a `restore_context(c)`[17], jde pomocí nich již snadno napsat funkci `program_switch()`, jejímž voláním lze kdekoliv v programu vlastně označit bod, kde se „vyskakuje“, resp. „naskakuje“ zpět do programu. Detaily vyzradíme v příštím čísle – zkuste na ně přijít sami (nejdříve třeba jen pro dva pevně dané programy a pak obecně). Budeme rádi, když nám své pokusy do té doby zašlete:

Problém 1.7: *Diskutujte a předveďte možnosti implementace funkce `program_switch()` popsané výše. Jaká klíčová rozhodnutí to obnáší? (Vážně by nemělo být potřeba používat žádné pokročilé funkce kromě našich (smyšlených) `save_context()` a `restore_context(c)` a vašich vlastních. Např. chcete-li podporovat dynamické spouštění programů, můžete implementovat funkci `start_program(...)` apod.)*

Problému se nebojte – je vskutku otevřený a slouží především k ověření, jak dobře jste koncepty pochopili, resp. jak dobře jsme je vysvětlili.

Všimněme si ale, že jsme ještě nedosáhli původního cíle – stále potřebujeme, aby programy byly ochotné se samy „přepínat“. (Proto se tomuto přístupu říká *kooperativní* multitasking, který je zajímavý i sám o sobě.) Nicméně jsme na dobré cestě. Skládačku doplníme příště, kdy si povíme o klíčovém konceptu přerušení (interrupts). Uvidíme, že existují algoritmy pro „plánování“ běhu programů s různými vlastnostmi. Nakonec dáme do souvislosti s multitaskingem vstup/výstup (příslušenství), načež již plně porozumíme úvodnímu „problému kiosku“.

[17] Ty ve vyšších jazycích ve skutečnosti nenajdeme, ale ve strojovém kódu je kontext přirozeně dostupný a ještě v C existují ekvivalenty v podobě `setjmp` a `longjmp`, se kterými si čtenář znalý jazyka může pohrát.

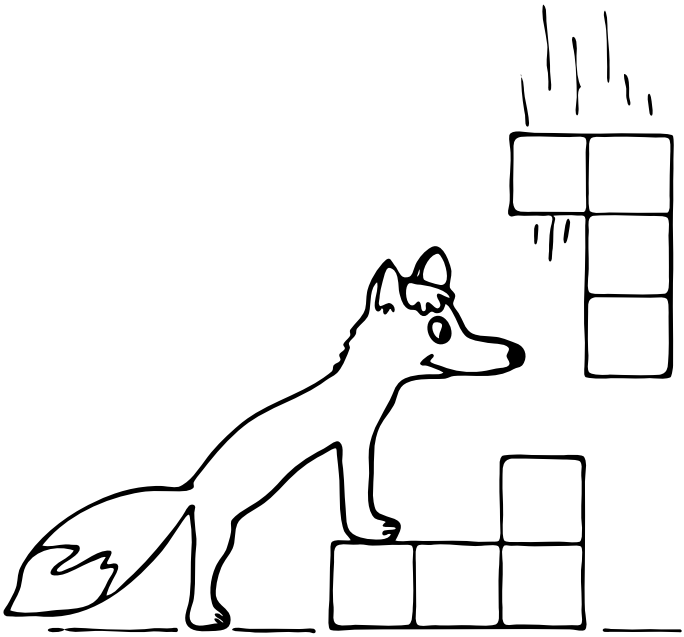
Bonusové úlohy

Úloha 1.8 [1+2b]: *Říkali jsme, že v ukázkovém programu z Příkladu 1 není zřejmé, po kolika krocích vždy doběhne vnitřní cyklus. Ve skutečnosti není zřejmé, jestli vůbec vždy doběhne (pro libovolné n). Jedná se o známý problém. Vyhledejte bez použití AI, jak se jmenuje. (1 bod) Pak dokažte, že pokud $n = 8k + 4$ (pro nějaké přirozené číslo k), pak vnitřní cyklus doběhne (jestli vůbec) pro n po stejně mnoha krocích jako pro $n + 1$. (2 body)*

Úloha 1.9 [2b]: *Přepište váš generátor hezké posloupnosti z úlohy 1.2, aby (smysluplně) využíval rekurzi namísto cyklů. Popište celý kontext i zásobník v momentě po přidání osmého prvku do seznamu.*

Budu moc rád za zpětnou vazbu!

Jakub; `jakub.sebek.o@seznam.cz`
odevzdávejte do odevzdávátka



Téma 4 – Rozměrová analýza

Díl 1: Odhadování vztahů bez pochopení fyziky

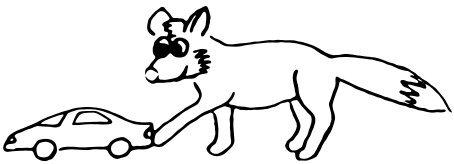
Jak poznáte, který návrh zaoceánské lodi je nejlepší, když nemáte počítačové simulace a nemůžete si dovolit postavit tucet drahých plavidel v plné velikosti a zkoušet je na moři? Překvapivě daleko se dostanete jenom tím, že pečlivě hlídáte jednotky. Metoda, které se říká **rozměrová analýza**, z „uhádnutí“ dělá skoro exaktní disciplínu – a navíc přesně určí, v jakém režimu musí běžet zmenšený model, aby se choval jako skutečnost.

Rozměrová analýza se fundamentálně používá na dvě věci:

- Uhodnutí fyzikálního vztahu.
- Navržení experimentu, kde se budou relevantní jevy chovat stejně, což nám umožní změřit všechny konstanty v uhodnutém vztahu.

Ty spolu velmi souvisí, protože často dostaneme trochu jinou podobu vztahu, než je standardní, protože všechny konstanty musí být bezrozměrné, a tato podoba je nejvíce užitečná pro návrh experimentů v praktičtějším měřítku.

Rozměrovou analýzou se nejprve z proměnných poskládají bezrozměrné skupiny[18], a v modelu nastavujeme volné proměnné, aby celkem bezrozměrná skupina vycházela stejně jako v reálné verzi. V nádrži se vlečený model lodi proto vláčí znatelně pomaleji, v jiném experimentu létají modely letadel ve vzduchu o teplotě $-160\,^{\circ}\mathrm{C}$. Hlavní reálné využití rozměrové analýzy je tedy škálování modelů pro jevy, které nejdou spočítat, ale je potřeba simulovat. V současnosti NASA stále používá kryogenní hypersonický větrný tunel.



[18] Bezrozměrná skupina je násobek několika veličin v nějakých mocninách, kde se všechny rozměry vykrátí. Například to může být něco jednoduchého jako $l^{-1} \cdot d$, kde l i d mají rozměr v metrech, takže $\mathrm{m}^{-1} \cdot \mathrm{m} = \mathrm{m}^0 = 1$. Nebo to může být složitější jako $Q^2 \cdot \varrho^{-2} \cdot g^{-1} \cdot D^{-5}$. Rozepíšeme rozměry a postupně krátíme:

$$
\begin{aligned}
&(\mathrm{kg/s})^2 \cdot (\mathrm{kg/m^3})^{-2} \cdot (\mathrm{m/s^2})^{-1} \cdot \mathrm{m}^{-5} \\
\to\ &\mathrm{kg}^2 \cdot \mathrm{s}^{-2} \cdot \mathrm{kg}^{-2} \cdot \mathrm{m}^6 \cdot \mathrm{m}^{-1} \cdot \mathrm{s}^2 \cdot \mathrm{m}^{-5} \\
\to\ &\cancel{\mathrm{kg}^2} \cdot \mathrm{s}^{-2} \cdot \cancel{\mathrm{kg}^{-2}} \cdot \mathrm{m}^6 \cdot \mathrm{m}^{-1} \cdot \mathrm{s}^2 \cdot \mathrm{m}^{-5} \\
\to\ &\mathrm{s}^{-2} \cdot \mathrm{m}^6 \cdot \mathrm{m}^{-1} \cdot \mathrm{s}^2 \cdot \mathrm{m}^{-5} \\
\to\ &\cancel{\mathrm{s}^{-2}} \cdot \mathrm{m}^6 \cdot \mathrm{m}^{-1} \cdot \cancel{\mathrm{s}^2} \cdot \mathrm{m}^{-5} \\
\to\ &\mathrm{m}^6 \cdot \mathrm{m}^{-1} \cdot \mathrm{m}^{-5} \\
\to\ &\mathrm{m}^{6-1-5} = \mathrm{m}^0 = 1.
\end{aligned}
$$

Jak rychle chodíte?

Cílem tu pro nás bude:

1. najít vztah mezi rychlostí chůze a proměnnými popisujícími nohu,
2. prověřit ten vztah v praxi.

Co všechno by rychlost chůze v mohlo ovlivnit?
Než začneme počítat, zastavme se u téhle otázky. Napnutá noha se chová jako kyvadlo: v kyčli je zavěšená k tělu a chodidlo opisuje oblouk. Pohodlná (energeticky nejlevnější) rychlost chůze v (m/s) bude souviset s tím, jak rychle toto kyvadlo dokáže přehazovat nohu z jedné strany na druhou.

Napišme co nejdelší rozumný seznam kandidátů na proměnné a u každého jeho rozměr v základních SI jednotkách:

- délka nohy l (m),
- výška těžiště nohy nad zemí h (m),
- průměr nohy d (m),
- hmotnost nohy m (kg),
- tíhové zrychlení g ($\mathrm{m\,s^{-2}}$).

Teď pozorujeme dvě věci.

Hmotnost musí vypadnout. m je jediná proměnná, která má v sobě rozměr kg. Výsledná rychlost má rozměr m/s; kilogramy v odpovědi nemáme čím „umazat“, takže se hmotnost v rozměrově správném vztahu nesmí objevit. (Fyzikálně se to dalo tušit: kyvadlo v tíhovém poli je stejně jako padající kámen na hmotnosti nezávislé.)

Délky můžeme zjednodušit na jednu. Délky l, h, d mají stejný rozměr m. To se dá vyřešit ve dvou případech: ty 3 délky na sobě nejsou nezávislé, nebo dokážeme říci, jak je složit dohromady. Naštěstí u většiny lidí i kráčejících zvířat jsou v přibližně konstantních poměrech. Takže platí, že jsou závislé. Ten druhý případ uvidíme v úloze 1.7 na konci tématka. Nahradíme tedy h za $c_1 \cdot l$ a d za $c_2 \cdot l$. Bezrozměrné konstanty můžeme všechny složit dohromady při skládání rovnice, takže zbude jediná nezávislá délka l.

Zbyly nám jen l a g. Teď nějak potřebujeme poskládat bezrozměrné skupiny – násobek / zlomek mocnin našich proměnných, kde se všechny rozměry vyškrtají. Jelikož máme akorát o jednu proměnnou (l, g, v) víc, než máme jednotek (m, s), tak víme, že budeme mít právě jednu bezrozměrnou skupinu $l^a \cdot g^b \cdot v^c$. Vychází to z toho, že když vezmeme do jedné skupiny právě dva rozměry, tak třetí rozměr zůstane ve skupině samotný a nemůže se vykrátit. Z jedné skupiny můžeme rovnici udělat triviálně:

$$1 = \mathrm{konst} \cdot l^a \cdot g^b \cdot v^c.$$

Abychom vykrátili proměnné, tak nám stačí 2 neznámé exponenty, a jelikož nás zajímá vztah pro v, tak můžeme nastavit $c = 1$ a přeskládat rovnici na:

$$v = \text{konst} \cdot l^a \cdot g^b.$$

Další krok je porovnat jednotky:

$$\mathrm{m\,s^{-1}} = \mathrm{m}^a \cdot (\mathrm{m\,s^{-2}})^b = \mathrm{m}^{a+b} \cdot \mathrm{s}^{-2b}.$$

Porovnáním exponentů[19] u metru a sekundy dostaneme $a + b = 1$ a $-2b = -1$, odkud $b = 1/2$, $a = 1/2$. Výsledek je

$$v = \xi \cdot \sqrt{g \cdot l},$$

kde ξ je bezrozměrná konstanta. Tu rozměrová analýza neprozradí, ale stačí ji odhadnout z jediného měření. Ekvivalentně: kombinace $\xi = v/\sqrt{g \cdot l}$ je u chůze univerzální bezrozměrné číslo pro všechny chodící tvory.

Úloha 1.1 [2b]: *Stopkami a metrem změřte rychlost pohodlné chůze několika co nejvíc různě vysokých kamarádů a délku jejich nohy (od kyčle k zemi). Vyneste v proti $\sqrt{l}$. Vyjde přímka? Odhadněte z ní konstantu ξ.*

Úloha 1.2 [1b]: *Na Měsíci je g přibližně šestkrát menší než na Zemi. Jakou rychlostí byste tam chodili? Srovnejte s tím, co popisovali měsíční astronauti – proč „normální“ chůze na Měsíci nefungovala a raději přešli na charakteristické poskakování?*

Problém 1.3: *Najděte rychlost pohodlné chůze a délku nohy pro co nejvíce zvířat (slon, kráva, pes, kočka, kuře, …). Vyneste do grafu v proti $\sqrt{g \cdot l}$ a ověřte, zda body leží na společné křivce (nebo přímce na log-log grafu). Pozor: do datasetu patří jen zvířata, která **skutečně kráčejí** (rovná, svisle stojící noha během opory) – ne ta, která skáčou, cválají nebo chodí s nohama horizontálně jako někteří plazi. Pasuje to?*[20]

Problém 1.4: *Proč je běh rychlejší než chůze, když náš vztah pro kyvadlo dává jedinou „přirozenou“ rychlost?*

Nápověda: *Co dělá běžcova noha v okamžiku, kdy se vrací dopředu – změnila se efektivní délka? Jaké dva efekty ovlivňuje délka a který byl ovlivněn?*

[19] Jelikož proměnné násobíme, tak exponenty můžeme sčítat.

[20] Srovnávací dataset rychlostí chůze a rozměrů končetin u savců najdete např. v Alexander, R. McN.: *Principles of Animal Locomotion* (Princeton UP, 2003), graf 7.1.

Postup v rozměrové analýze

Co jsme právě udělali, se dá zformulovat jako návod:

1. **Sepište pravděpodobné proměnné.** Radši víc než míň.
2. **Zapište rozměry všech proměnných** v základních jednotkách (metr, sekunda, kilogram).
3. **Vyřaďte proměnné s unikátními rozměry.** Není možné je s ničím vykrátit pro tvorbu bezrozměrných skupin.
4. **Zkuste najít vztahy mezi veličinami se stejným rozměrem.** Někdy se jich můžeme zbavit úplně jako v prvním příkladě, jindy se nám povede najít mezi nimi vztah případně obsahující bezrozměrnou konstantu (uvidíme v posledním příkladě).
5. **Sestavte skupiny, které splňují *počet proměnných* = *počet rozměrů* + 1.**
6. **Spočítejte exponenty, aby skupiny byly bezrozměrné.** Porovnáním rozměrů se mocniny doladí tak, aby se exponenty každého rozměru sečetly na nulu.
7. Pokud máte pouze jednu bezrozměrnou skupinu, můžete triviálně sestavit rovnici $1 = \xi \cdot \{S\}$. Pokud více, tak to potřebujete dořešit jinak než rozměrovou analýzou.

Tuhle kostru teď uvidíme na příkladu, kde seznam veličin není hned zřejmý.

Vlny na hladině

Podívejme se teď na jiný systém, kde se stejný rozměrový trik hodí. Hoďte kamínek do dostatečně hlubokého rybníka a sledujte kružnice, které se rozběhnou od místa dopadu. Jakou rychlostí c se takové vlny šíří?

Omezíme se na hluboké vlny (tedy na situaci, kdy je hloubka vody větší než půlka vlnové délky) a na vlny delší než 5 cm (protože v menších dělá velký efekt povrchové napětí).

Co může rychlost c ovlivnit? Kamínek hladinu vychýlí z rovnováhy a gravitace ji tlačí zpět dolů; setrvačnost rozhýbané vody rozkmit udržuje. Sepišme kandidáty:

- vlnová délka λ s rozměrem m,
- tíhové zrychlení g s rozměrem $\mathrm{m\,s^{-2}}$,
- hustota vody ϱ s rozměrem $\mathrm{kg\,m^{-3}}$.

A teď se stane něco povědomého. Hustota musí vypadnout. Proměnná ϱ je jediná, která v sobě nese kilogramy; rychlost c kilogramy neobsahuje, takže ϱ nemá čím své kilogramy zkrátit a musí se objevit s nulovým exponentem. Je to přesně stejný krok jako u chůze, kde ze stejného důvodu vypadla hmotnost. Fyzikálně to dává smysl: obnovující síla na vlnu škáluje s ϱ (tíha vychýleného sloupce vody) a setrvačnost rozkmitané vody taky s ϱ, takže se v poměru, který určuje rychlost, vykrátí – stejně jako se u kyvadla vykrátí hmotnost.

Zbyly nám tři proměnné c, λ, g a dva rozměry (m, s), tedy jedna bezrozměrná skupina. Hledáme $c = \text{konst} \cdot \lambda^a \cdot g^b$. Srovnáním rozměrů:

$$\mathrm{m\,s^{-1}} = \mathrm{m}^a \cdot (\mathrm{m\,s^{-2}})^b = \mathrm{m}^{a+b} \cdot \mathrm{s}^{-2b}.$$

Z exponentu u sekundy $-2b = -1$, tedy $b = 1/2$; z exponentu u metru $a + b = 1$, tedy $a = 1/2$. Vychází

$$c = \varphi \cdot \sqrt{g \cdot \lambda},$$

kde φ je bezrozměrná konstanta.

Vyšlo nám, že dlouhé vlny utíkají před krátkými – tomuhle jevu se říká disperze a je to důvod, proč se z neuspořádaného šplouchnutí časem rozjede úhledný vlnový balík: dlouhé komponenty běží napřed, krátké se opožďují.

Problém 1.5: *Ověřte vztah $c = \varphi \cdot \sqrt{g \cdot \lambda}$ experimentálně a najděte φ. Najděte si klidnou vodní hladinu – rybník, okraj jezera, bazén nebo velkou vanu – s hloubkou alespoň 10 cm. Ideálně bez větru a vln, ať jsou vaše vlny vidět. Hoďte do vody hladký oblázek (průměr zhruba 1–5 cm) ze zvolené výšky mezi 20 cm a 1 m a natočte dopad zpomaleným videem na mobilu. Potřebujete nějak určit měřítko – buď umístíte pravítko poblíž dopadu, nebo si změříte vzdálenost mobilu od dopadu a spočítáte velikost pixelu. Vyzkoušejte několik kombinací kamínků a výšek, vyberte si, v jakých případech jsou rozbíhající se vlny dobře vidět. Rychlost vlny bude záležet pouze na vlnové délce a každý hod vytvoří několik vln o různých vlnových délkách. Z videí odečtěte pro několik různých hřebenů jejich vlnovou délku λ (vzdálenost mezi sousedními hřebeny) a jejich rychlost c (vzdálenost uraženou mezi dvěma snímky dělenou časem mezi nimi). Vyneste c^2 proti λ na lineárních osách a změřte sklon. Z toho spočítejte φ. Odevzdejte taky fotky, videa, případně s anotacemi.*

Vibrace struny

Představte si napnutou strunu – třeba strunu kytary, gumičku přes krabičku od sirek nebo prádelní šňůru. Má lineární hmotnost μ (v kg/m) a je napnutá silou T (v $\mathrm{N} = \mathrm{kg\,m\,s^{-2}}$). Cvrnknete do ní příčně a po struně se rozběhne vlna – podobně jako se v předchozím experimentu rozbíhaly vlnky po hladině, akorát teď jednodimenzionálně. Seznam proměnných je tentokrát krátký, postup ale zůstává stejný jako u kapilárních vlnek: sepište kandidáty, zapište jejich rozměry, slepte bezrozměrnou kombinaci a přečtěte odpověď.

Úloha 1.6 [3b]: *Jakou rychlostí v se rozběhnou vlny po prádelní šňůře? Předpokládejte, že gravitace se dá zanedbat, jelikož je napnutí struny dost silné.*

Když jsou faktory dva

Rozměrová analýza bývá nejzajímavější tam, kde se jeden jev rozpadá na dva nezávislé škálovací zákony. Zpět k parníku z úvodu: loď při pohybu vodou táhne dvě odlišně se škálující složky odporu: **vlnový odpor** (loď si na hladině vytváří soustavu vln, jejichž energie mizí za zádí) a **třecí odpor** (tenká vrstva vody u trupu se pohybuje s lodí, vzniká viskózní tření po celé smáčené ploše). Každá má vlastní bezrozměrnou kombinaci: vlnový odpor škáluje s $v/\sqrt{g \cdot L}$ (rychlost lodi, gravitační zrychlení, délka lodi), třecí odpor na jiné kombinaci rychlosti, délky a vlastností vody. Obě kombinace nelze na zmenšeném modelu dodržet současně.

Jedno řešení, co se nabízí, je naškálovat jeden model, aby byl vlnový odpor zanedbatelný a třecí správně, druhý model, aby byl třecí odpor zanedbatelný, ale vlnový správně. Problém je, že abychom třecí odpor naškálovali správně, natož zanedbatelně, tak potřebujeme kapaliny řádově méně viskózní než vodu, a takové kapaliny se špatně hledají.

Druhé řešení, které v tomto případě je nutno využít, je, že třecí odpor se vlastně dá docela dobře spočítat. Vlnový odpor je ten složitý, který velmi záleží na přesném tvaru lodi. Takže naškálujeme model, aby vlnový odpor odpovídal (loď jede patřičně pomaleji, aby byly vlny geometricky podobné), od změřeného odporu odečteme třecí odpor pro model, a zbylý vlnový odpor naškálujeme. Tento postup se používal pro vývoj lodí od druhé půlky 19. století.

Modelování sila

Zemědělské silo, dávkovač léků nebo obyčejná slánka – všechno to jsou úlohy o tom, jak rychle vypadává sypký materiál otvorem dole. Jelikož je silo velká stavba, je to příklad, kde se rozměrová analýza používala v praxi pro pochopení chování z vhodného modelu. Budeme se snažit zjistit, jaký hmotnostní průtok Q (v kg/s) vyteče kruhovou aperturou o průměru D.

První si ale povíme o sypkých materiálech. Sypká hmota se chová jinak než kapalina. Zrna se o sebe opírají a přenášejí si zatížení přes **silové řetězce** – náhodné řetězy bodových dotyků, po kterých většina tíhy sloupce neputuje rovnou dolů, ale šikmo v těch směrech, kde je více opory.

- **Proč výška náplně nehraje roli.** Jakmile se nad otvorem vytvoří krátká klenba (zrnka se dočasně zaklíní do klenby), proudí materiál jen rychlostí, jakou se tato klenba hroutí. Svislé napětí ve sloupci se s hloubkou navíc saturuje – přestává růst, protože tíhu přebírají boční stěny přes tření. Tlak u dna nezávisí na tom, jak vysoko je silo naplněné (proto je v hromadě obilí spodní zrno rozdrcené méně, než by člověk naivně čekal). Průtok otvorem tedy taky nezávisí na výšce náplně – rozhoduje jen lokální geometrie kolem otvoru.

- **Sypná hmotnost.** U hustoty sypké hmoty je dobré rozlišovat dvě veličiny. Hustota materiálu zrna (třeba křemen u písku, $\approx 2650\,\mathrm{kg\,m^{-3}}$) popisuje, z čeho je jedno zrno udělané. Pro proudění ze sila ale potřebujeme sypnou hmotnost (někdy též sypnou hustotu) ϱ – hmotnost volně nasypané hmoty na celkový objem, který zaujímá, včetně dutin mezi zrny. Typicky je 50–65 % hustoty pevného zrna. Toto je ta veličina, kterou chcete použít pro hustotu. Pokud ji budete potřebovat měřit, tak porovnáváte objem sypkého materiálu včetně dutin a jeho hmotnost, musíte si tedy změřit objem v odměrce.

Úloha 1.7 [4b]: *Pomocí rozměrové analýzy odvoďte vztah pro hmotnostní průtok Q. Dospějte k formuli v uzavřeném tvaru (až na 2 bezrozměrné konstanty).*

Nápověda: *Jsou tam dvě relevantní délkové proměnné – průměr otvoru D a velikost zrna d. Jelikož velikost zrna zmenšuje efektivní otvor, stačí tyto dvě proměnné používat pouze ve tvaru $(D-\theta\cdot d)$, kde θ je bezrozměrná konstanta.*

Problém 1.8: *Sestavte si doma vlastní malé silo a ověřte škálování z úlohy 1.7. Chcete docela rovné dno a možnost měnit velikost díry nebo několik sil s různě velkými dírami, cca mezi 3 mm a 3 cm. Sypejte různé materiály – mouku (ale ne hladkou, tam už příliš působí molekulární síly), krystalový cukr, kuskus, čočku, cizrnu nebo cokoli jiného, co se sype alespoň jednou z dírek.*

Pro každou kombinaci změřte, za jak dlouho vyteče známé množství materiálu, a spočítejte Q. Q bude docela konstantní, než sypké hmoty v silu začne docházet. Vyneste $\log Q$ proti $\log D$ a porovnejte sklon s teoretickou předpovědí (používáme log-log graf, protože z x^a se stane přímka ax' a je tedy mnohem snazší změřit proměnnou a z grafu). Odhadněte obě bezrozměrné konstanty a diskutujte odchylky. Pro jeden zvolený otvor navíc ověřte, že Q nezávisí na výšce náplně. Odevzdejte taky fotky, videa, případně s anotacemi.

Kuba S.; suchanek989@seznam.cz
odevzdávejte do odevzdávátka

Téma 5 – Obecná teorie relativity

Díl 1: Tenzory

Milý čtenáři, obecná teorie relativity je teorií prostoru, času a gravitace. Jakési jádro této teorie je docela jednoduché, gravitace je geometrie. Efekty, které přisuzujeme gravitaci, jsou pouze projevem zakřivení prostoru a času. V tomto tématku se mimo jiné pokusíme nastínit, proč by tomu tak mělo být. Ovšem pracovat v zakřivených prostorech (například na sféře) umí být velmi matematicky náročné. Proto má toto tématko jisté prerekvizity, které prostě není možné všechny shrnout zde. Velmi tedy doporučujeme přečíst si lingebraické tématko[21], tématko o derivacích a integrálech[22] a samozřejmě tématko o speciální teorii relativity[23]. Jistě, že je číst nemusíte, abyste se pokusili o pochopení tohoto tématka, ovšem pokud se někde třeba ztratíte, jsou to námi doporučené zdroje, kde se potřebný materiál můžete doučit.

V tomto díle ještě moc fyziky ovšem nebude, protože se zde budeme bavit o té nejdůležitější prerekvizitě, o tenzorech. Ty jsou pro OTR (obecnou teorii relativity, tuto zkratku nejspíše ještě použijeme) tak důležité, že jsou svým způsobem zakomponovány do jednoho ze dvou výchozích principů OTR. O těchto principech se budeme bavit v dalších dílech, jen věžte, že zmiňovaný princip lze chápat jako požadavek, aby všechny fyzikální zákony šly zapsat pomocí tenzorových rovnic.

Motivace

Pro motivaci k používání tenzorových veličin k popisu světa kolem nás se nemusíme nutně obracet na OTR. Úplně nám bude stačit i klasická mechanika. Uvažme nejjednodušší typ tenzoru, skalár. Rovnou budeme uvažovat celé pole těchto skalárů, abychom si trochu zvykli na složitější tenzorová pole. Skalární pole je jednoduše **funkce**, která každému bodu v prostoru přiřadí skalár (číslo). Klasickým příkladem je například pole teploty v místnosti nebo příčná výchylka na membráně bubnu, když do něj udeříme paličkou (tedy, udeřit do něj nemusíme, jen jsme toho názoru, že nulové pole je poněkud nezajímavé). Abychom mohli každému bodu přiřadit číslo, musíme nutně zavést nějaký souřadnicový systém. Klasicky bychom nejspíše vzali kartézký, ovšem uvažte, že například pro případ s bubnem by byl vhodnější polární souřadnicový systém. Dobrá, ovšem zvolme si pro jednoduchost kartézské souřadnice. Co když jejich počátek posuneme o vektor $\vec{x}$? Jistě se kvůli naší představě počátku jen tak nezmění teplota v místnosti, tedy nové pole teploty bude v jistém smyslu pořád to stejné pole teploty. Ano, funkce popisující teplotu bude vypadat jinak, ovšem fyziku to nijak nezmění (a představte si, kdyby změnilo!). Pro ty z vás, kteří už se chtějí vrhnout na matematiku, nová funkce teploty T' bude v novém souřadném systému triviálně $T'(\vec{r}\,') = T(\vec{r} - \vec{x})$, kde si uvědomte, že $\vec{r}\,'$ je polohový vektor v nových souřadnicích a vektory $\vec{r}$ a $\vec{x}$ jsou ve starých souřadnicích.

[21] Odkaz na první číslo tématka: `https://mam.mff.cuni.cz/media/cislo/pdf/31/31-1.pdf`
[22] Odkaz na první číslo tématka: `https://mam.mff.cuni.cz/media/cislo/pdf/29/29-1.pdf`
[23] Odkaz na první číslo tématka: `https://mam.mff.cuni.cz/media/cislo/pdf/31/31-2.pdf`

Idea, že by nemělo záležet na tom, jak si definujeme souřadnicový systém, je právě motivací pro užívání tenzorů ve fyzice. Dá se říct, že to jsou právě takové objekty, které se při transformaci souřadnic nezmění. Pozor, to neznamená, že se nezmění jejich složky, jak jsme viděli u teploty (byť ta má ovšem složku pouze jednu).

Velmi důležitým objektem je ve fyzice také vektor, o kterém vězte, že je též tenzorem. Proto se v další sekci podíváme na to, jak ho transformovat.

Úloha 1.1 [1b]:

Mějme transformaci kartézských souřadnic (x, t) *zadanou rovnicemi:*

$$x' = x + b_0 t,$$
$$t' = t,$$

kde b_0 *je konstanta. Diskutujte, zda-li je rychlost vektorem vůči těmto transformacím. Diskutujte, zda-li je zrychlení vektor.*
(Hint. Definice rychlosti je $v = \frac{dx}{dt}$ *a definice zrychlení je* $a = \frac{d^2x}{dt^2}$*. Zkuste tedy spočítat* $\frac{dx'}{dt'}$ *a* $\frac{d^2x'}{dt'^2}$*.)*

Transformace vektorů

Pro jednoduchost budeme až do konce dílu pracovat v kartézských souřadnicích. Jak víme, vektory „žijí“ ve vektorových prostorech. Každý takový prostor má bázi. Pracujme po zbytek dílu v ortonormální bázi (byť to často není nutná podmínka platnosti následujících tvrzení). To znamená, že bázové vektory jsou všechny jednotkové a jsou na sebe kolmé. Mějme vektor $\mathbf{v}$ v našem vektorovém prostoru. Z definice báze existuje unikátní sada čísel v^i (zde i značí index, ne mocninu), pro které platí:

$$\mathbf{v} = \sum_{i=1}^{n} v^i \mathbf{e}_i, \tag{1}$$

kde $\mathbf{e}_i$ značí i-tý bázový vektor.

Protože při práci s tenzory budeme používat velké množství sum, zavedeme zde takzvanou Einsteinovu sumační konvenci. To znamená, že pokud je někde psaný index nahoře a násobí se s něčím, co má ten samý index dole, automaticky se předpokládá, že se přes tento index sčítá. Rovnice (1) má potom tvar:

$$\mathbf{v} = v^i \mathbf{e}_i. \tag{2}$$

Nyní se budeme zabývat transformacemi báze a u toho zjistíme, proč jsme začali rozlišovat indexy psané nahoře a dole.

Mějme tedy dvě báze, jednu čárkovanou $\mathbf{e}'_j$ a jednu nečárkovanou $\mathbf{e}_i$. Jak přejdeme z jedné báze do báze druhé? Jak se u toho změní složky vektoru $\mathbf{v}$? Začneme úvahou, že vektory čárkované báze jdou (jaksi z definice báze) vyjádřit jako lineární kombinace vektorů báze nečárkované, tedy:

$$\mathbf{e}'_j = R^i{}_j \mathbf{e}_i, \tag{3}$$

kde $R^i{}_j$ značí příslušné rozvojové koeficienty. Nepřekvapí nás, že existují i takové koeficienty $S^j{}_k$, že:

$$\mathbf{e}_k = S^j{}_k \mathbf{e}'_j. \tag{4}$$

Dosazením vztahu (3) do vztahu (4) pak:

$$\mathbf{e}_k = S^j{}_k R^i{}_j \mathbf{e}_i, \tag{5}$$

z čehož je vidět, že $S^j{}_k R^i{}_j = \delta^i{}_k$, kde δ značí tzv. Kroneckerovo delta, tedy složky jednotkové matice. Z toho dále plyne, že koeficienty $S^j{}_k$, zapsány do matice, by tvořily matici inverzní k R (což by značilo matici z koeficientů $R^i{}_j$). Budeme tedy koeficienty $S^j{}_k$ od teď značit jako $(R^{-1})^j{}_k$.

Nyní tedy umíme přecházet mezi bázemi. Teď odpovíme na otázku o složkách vektorů. Začneme u vztahu (2), přepíšeme ho pomocí Kroneckerovy delty (to „nic neudělá“, protože je to vlastně násobení složek vektoru jednotkovou maticí) a dosadíme za ni:

$$\mathbf{v} = v^i \mathbf{e}_i = v^k \delta^i{}_k \mathbf{e}_i = v^k (R^{-1})^j{}_k R^i{}_j \mathbf{e}_i = v^k (R^{-1})^j{}_k \mathbf{e}'_j = v'^j \mathbf{e}'_j. \tag{6}$$

V předposledním kroku jsme jen identifikovali bázové vektory čárkované báze. Poslední krok je zde pouze jakési připomenutí, že i v nové bázi bude platit vztah (2), ovšem „očárkovaný“. Z toho však ihned plyne, že složky vektoru $\mathbf{v}$ vůči čárkované bázi jsou:

$$v'^j = (R^{-1})^j{}_k v^k. \tag{7}$$

To znamená, že složky vektoru se transformují s inverzní transformací, než jak se transformuje báze! To má samozřejmě dobrý smysl. Jen uvažte, že pokud třeba dvakrát zvětšíme bázové vektory, složky daného vektoru se musí dvakrát zmenšit, aby to byl pořád ten samý vektor.

To nás přivádí také k notaci s horními a dolními indexy. O objektech, které se transformují stejně jako báze vektorového prostoru, řekneme, že jsou kovariantní a budeme je značit s dolním indexem, neboli kovariantním indexem. Naopak objekty transformující se opačně, než jak se transformuje báze, nazýváme kontravariantní a značíme je s indexem nahoře, neboli s kontravariantním indexem.

Lineární formy a jejich transformace

Dobrá, nyní jsme jaksi matematicky rozcvičeni, abychom začali probírat věci, o kterých předpokládáme, že budou pro některé čtenáře nové. Objekty, které budeme nyní probírat, se nazývají lineární formy a jsou jaksi druhou podstatnou ingrediencí pro obecné tenzory.

Lineární forma je **funkce**, která si jako input vezme vektor a jako output nám dá číslo. Je důležité, že tato funkce je lineární, tedy pro lineární formu α, vektory $\mathbf{v}$, $\mathbf{u}$ a čísla a, b platí:

$$\alpha(a\mathbf{v} + b\mathbf{u}) = a\alpha(\mathbf{v}) + b\alpha(\mathbf{u}). \tag{8}$$

Úloha 1.2 [1b]: *Není těžké se přesvědčit, že lineární formy tvoří vektorový prostor. Najděte si na internetu, co všechno musí platit, aby byla nějaká množina vektorovým prostorem, a každou vlastnost ověřte pro lineární formy.*

Z předchozí úložky víte, že prostor lineárních forem tvoří vektorový prostor. Pojďme se tedy pobavit o bázi tohoto prostoru. Budeme dokonce uvažovat speciální případ báze prostoru lineárních forem a to bázi tzv. duální.

Nechť $B = \{\mathbf{e}_i\}_{i=1}^n$ je bází vektorového prostoru V. Označme prostor lineárních forem, které působí na vektory z V, jako V^*. Báze prostoru V^*, ozn. $B^* = \{\mathbf{e}^j\}_{j=1}^n$ se nazývá duální báze k bázi B, pokud platí:

$$\mathbf{e}^j(\mathbf{e}_i) = \delta^j_{\ i}. \tag{9}$$

Abychom si na lineární formy trochu zvykli, pojďme odvodit pár identit.

Spočtěme, co se stane, aplikujeme-li j-tou bázovou formu na vektor $\mathbf{v}$:

$$\mathbf{e}^j(\mathbf{v}) = \mathbf{e}^j(v^i\mathbf{e}_i) = v^i\mathbf{e}^j(\mathbf{e}_i) = v^i\delta^j_{\ i} = v^j, \tag{10}$$

kde jsme v prvním kroku využili toho, že každý vektor lze rozepsat jako lineární kombinaci bázových vektorů. Poté jsme využili linearitu lineárních forem a následně definici duální báze. Platnost posledního kroku plyne triviálně z toho, že Kroneckerovo delta je 0 pro všechny $i \neq j$ a tedy nám v sumě zbyde pouze jeden nenulový člen s hodnotou v^j.

Protože B^* je báze, lze každou formu z V^* vyjádřit jako lineární kombinaci této báze. Tedy pro libovolnou formu α platí:

$$\alpha = \alpha_j\mathbf{e}^j, \tag{11}$$

kde α_j jsou rozvojové koeficienty, které nazýváme složkami formy α vůči bázi B^*.

Spočtěme, co se stane, aplikujeme-li formu α na i-tý bázový vektor:

$$\alpha(\mathbf{e}_i) = \alpha_j\mathbf{e}^j(\mathbf{e}_i) = \alpha_j\delta^j_{\ i} = \alpha_i, \tag{12}$$

kde jsme pouze využili vztahu (11), definice duální báze a vlastností Kroneckerovy delty.

Úloha 1.3 [1b]: *Odvoďte ve složkách výsledek působení formy α na vektor $\boldsymbol{v}$.*

Jak jsme psali na začátku této sekce o lineárních formách, lineární formy jsou podstatnou ingrediencí obecných tenzorů. Proto by nás ovšem zajímalo, jak se transformují, změníme-li bázi vektorového prostoru. Tím samozřejmě změníme i odpovídající duální bázi. Z poloh indexů to samozřejmě dokážete uhodnout, ovšem pojďme si transformační vztahy odvodit.

Nechť tedy máme bázi $\mathbf{e}_i$ a k ní duální bázi $\mathbf{e}^j$ a samozřejmě čárkovanou bázi $\mathbf{e}'_k$ a k ní duální čárkovanou bázi $\mathbf{e}'^l$. Mějme taktéž formu $\alpha = \alpha_j \mathbf{e}^j$. Jak bude vypadat k-tý rozvojový koeficient formy α v čárkované bázi? Jednoduše dle vztahu (12):

$$\alpha'_k = \alpha(\mathbf{e}'_k) = \alpha(R^i{}_k \mathbf{e}_i) = R^i{}_k \alpha(\mathbf{e}_i) = R^i{}_k \alpha_i. \tag{13}$$

Je tedy jasné, že se složky forem transformují kovariantně.

Dále chceme transformační formuli pro bázové formy. Využijeme stejného triku jako v rovnici (6):

$$\alpha = \alpha_j \mathbf{e}^j = \alpha_i \delta^i{}_j \mathbf{e}^j = \alpha_i R^i{}_l (R^{-1})^l{}_j \mathbf{e}^j = \alpha'_l (R^{-1})^l{}_j \mathbf{e}^j. \tag{14}$$

Z toho je už snadné vykoukat:

$$\mathbf{e}'^l = (R^{-1})^l{}_j \mathbf{e}^j. \tag{15}$$

Což znamená, že se báze prostoru lineárních forem transformuje kontravariantně.

Toto je vše, co v tomto díle stíháme říci o lineárních formách, ovšem vyzýváme vás, abyste si zkusili najít jejich geometrickou interpretaci a jak souvisí s vrstevnicemi na mapách, hybností nebo s integrováním.

Obecné tenzory

Nyní máme potřebné ingredience k uvaření obecného tenzoru. Ovšem ještě musíme trochu prozkoumat proces samotného vaření. K tomu zavedeme operaci tenzorového součinu, kterou budeme značit $\otimes$. K zavedení tenzorového součinu ani nebudeme potřebovat formy a vektory, budou nám stačit obecné funkce.

Mějme tedy funkce $f : X \to \mathbb{R}$ a $g : Y \to \mathbb{R}$, pak jejich tenzorový součin vyrobí následující funkci:

$$\begin{aligned} f \otimes g : X \times Y &\longrightarrow \mathbb{R} \\ (x, y) &\to f(x) g(y). \end{aligned}$$

Pro tenzorový součin jaksi platí všechna pravidla jako pro součin normální (tedy součin dvou reálných čísel). Jen uvažte, jak by se toto z definice asi dokazovalo, když v definici normální součin přímo vystupuje. Musíme se mít malinko na pozoru s komutativitou, neboť při prohazování funkcí musíme prohodit i inputy, protože například $f(y)$ ani nemusí být definováno.

Pojďme si nyní představit bilineární formy. To, jak název napovídá, je forma, která si vezme dva vektory a vyrobí z nich číslo. Taktéž je samozřejmě v obou inputech lineární. S pomocí tenzorového součinu můžeme nějakou bilineární formu h vyrobit jednoduše: $h = \alpha \otimes \beta$. Uvědomme si následující:

$$h : V \times V \longrightarrow \mathbb{R}$$
$$(\mathbf{u}, \mathbf{v}) \to \alpha(\mathbf{u})\beta(\mathbf{v}).$$

Bilineární formy taktéž tvoří vektorový prostor. Bázi tohoto prostoru zkonstruujeme pro naše účely jako tenzorové součiny bázových lineárních forem. Obecná bilineární forma pak ve složkách vypadá takto:

$$h = h_{ij}\mathbf{e}^i \otimes \mathbf{e}^j. \tag{16}$$

(Uvědomte si, že tenzorový součin dvou bázových forem vypadá jako $\mathbf{e}^i \otimes \mathbf{e}^j$. Pro obecnou bilineární formu pak jen vysčítáme přes všechny kombinace.)

Obdobně můžeme tvořit i trilineární formy, tetralineární atd. Ovšem to by nám pořád jaksi nemuselo stačit. Ideálně bychom chtěli do těchto součinů ještě zahrnout i vektory samotné. To by ale z definice znamenalo dívat se na vektor jako na zobrazení. Toto zobrazení chceme z duálního vektorového prostoru V^* do reálných čísel. Protože jsme v konečné dimenzi, lze akci vektoru na lineární formu definovat následovně:

$$\mathbf{v}(\alpha) = \alpha(\mathbf{v}). \tag{17}$$

Nyní už má dobrý smysl výraz $\mathbf{e}_i \otimes \mathbf{e}^j$. Je to jednoduše funkce, která si vezme bod z $V^* \times V$ a vrátí číslo.

Obdobně vznikají obecné smíšené tenzory. Ve složkách je zapisujeme takto:

$$T = T^{ijk\ldots}{}_{lmn\ldots}\mathbf{e}_i \otimes \mathbf{e}_j \otimes \mathbf{e}_k \otimes \ldots \otimes \mathbf{e}^l \otimes \mathbf{e}^m \otimes \mathbf{e}^n \otimes \ldots \tag{18}$$

Ještě než vás pošleme tyto objekty zkoumat, zavedeme elegantní značení, které vám, ale hlavně nám, usnadní zápis. Tenzorový součin bázových vektorů budeme jaksi psát bez znaku tenzorového součinu. Prostě jen napíšeme všechny sčítací indexy k jednomu $\mathbf{e}$. Tedy například $\mathbf{e}_i \otimes \mathbf{e}_j = \mathbf{e}_{ij}$, $\mathbf{e}^k \otimes \mathbf{e}_m \otimes \mathbf{e}_l = \mathbf{e}^k{}_{ml}$ atp.

Úloha 1.4 [1b]: *Lze každou bilineární formu zapsat jako tenzorový součin dvou lineárních forem?*

Úloha 1.5 [1b]: *Dokažte, že platí následující tvrzení (Zde bychom si dovolili napovědět, abyste dokazovali jaksi postupně):*

$$h_{ij} = h(\boldsymbol{e}_i, \boldsymbol{e}_j),$$
$$h'_{kl} = R^i{}_k R^j{}_l h_{ij}$$
$$T^{ijk\ldots}{}_{lmn\ldots} = T(\boldsymbol{e}^i, \boldsymbol{e}^j, \boldsymbol{e}^k, \ldots, \boldsymbol{e}_l, \boldsymbol{e}_m, \boldsymbol{e}_n, \ldots)$$
$$T'^{ijk\ldots}{}_{lmn\ldots} = (R^{-1})^i{}_o (R^{-1})^j{}_p (R^{-1})^k{}_q \ldots R^r{}_l R^s{}_m R^t{}_n \ldots T^{opq\ldots}{}_{rst\ldots}$$

Zdvihání a spouštění indexů

Jistě jste si někteří všimli jistých divností během tohoto dílu. Například váš učitel na střední pravděpodobně psal složky vektorů s indexy dole a matematika mu fungovala; nebo že akce lineární formy na vektor je nápadně podobná skalárnímu součinu. Zde je nutné dodat, že se o skalární součin nejedná, nebo ne úplně.

V této sekci se pokusíme vysvětlit dané podivnosti a proč jsou naprosto v pořádku. Taktéž tím završíme naše povídání o tenzorech (ne však o tenzorových polích).

Začneme zlehka, a to částečným dosazením vektoru do bilineární formy. Co tím myslíme, je že inputneme pouze jeden vektor a druhý input necháme volný. Dobrá tedy:

$$h(\cdot,\mathbf{v}) = h_{ij}(\mathbf{e}^i \otimes \mathbf{e}^j)(\cdot,\mathbf{v}) = h_{ij}\mathbf{e}^i\mathbf{e}^j(\mathbf{v}) = h_{ij}v^j\mathbf{e}^i, \tag{19}$$

což jaksi očekávatelně vyrobilo lineární formu.

Speciálním případem bilineární formy je skalární součin. Jen se zamyslete nad jeho vlastnostmi. Taktéž pro „klasický“ skalární součin platí:

$$g(\mathbf{u},\mathbf{v}) = u_i v^i, \tag{20}$$

kde jsme dali do rovnosti $u_i = u^i$. Ovšem, zcela obecně je bilineární forma g jistě vyjádřitelná ve složkách. Potom dostaneme:

$$g(\mathbf{u},\mathbf{v}) = g_{ij}u^i v^j. \tag{21}$$

Označíme-li $u_j = g_{ij}u^i$, formulka pro skalární součin $(u_j v^j)$ pak musí být nutně zcela správně. Uvažte, že klasicky je $g_{ij} = \delta_{ij}$, tedy vskutku $u_j = u^j$. Skalárním součinem je pak motivováno právě spouštění indexů. Tedy dosadíme-li částečně vektor $\mathbf{u}$ do bilineární formy skalárního součinu, dostaneme lineární formu, která ovšem, po aplikaci na vektor, dává skalární součin vektoru $\mathbf{u}$ s daným vektorem.

Spouštění indexů pak prostě definujeme analogicky i pro složky tenzorů vyšších řádů. Jednoduše:

$$T_i{}^{jk} = g_{il}T^{ljk}. \tag{22}$$

Nebo pokud spouštíme indexů více najednou:

$$h_{ij} = g_{ik}g_{jl}h^{kl}. \tag{23}$$

Dobrá, co když chceme ale index z jakéhosi důvodu zdvihnout? Jednoduše by asi mělo platit, že pokud nějaký index zdvihneme, řekněme pomocí objektu se složkami f^{ij}, a následně daný index zase spustíme, dostaneme ten stejný objekt. Mělo by tedy platit:

$$u_i = g_{ij}f^{jk}u_k, \tag{24}$$

z čehož je ihned patrné, že:

$$g_{ij}f^{jk} = \delta_i{}^k, \tag{25}$$

tedy pokud g_{ij} jsou složky matice, pak f^{ij} jsou složky matice inverzní. Bývá zvykem značit složky této inverzní matice jako g^{ij}.

Zdvih nějakého indexu složky nějakého tenzoru pak jednoduše definujeme následovně:

$$T_{j\,k}^{\ i} = g^{il} T_{jlk}. \tag{26}$$

Úloha 1.6 [2b]: *Na determinant čtvercové matice se lze koukat jako na n-lineární formu, která je v každých dvou inputech antisymetrická (tedy $D(...,\boldsymbol{v},...,\boldsymbol{u},...) = -D(...,\boldsymbol{u},...,\boldsymbol{v},...)$) a platí:*

$$D(\boldsymbol{e}_1,\boldsymbol{e}_2,\boldsymbol{e}_3,...,\boldsymbol{e}_n) = 1.$$

Odvoďte složky formy determinantu ve 2 a 3 dimenzích.

Pokud do formy determinantu ve 2 dimenzích částečně dosadíme vektor, co je zajímavé na složkách výsledné formy? Pokud částečně dosadíme dva vektory do formy determinantu ve 3 dimenzích, co je zajímavé na složkách výsledné formy? Jak se této operaci běžně říká? (Zde myslíme operaci částečného dosazení dvou vektorů do formy determinantu ve 3 dimenzích.)

Úloha 1.7 [1.5b]: *Zaveďme zde kovariantní elektromagnetický tenzor jako:*

$$F_{\mu\nu} = \left(\begin{bmatrix} 0 & E_x & E_y & E_z \\ -E_x & 0 & B_z & -B_y \\ -E_y & -B_z & 0 & B_x \\ -E_z & B_y & -B_x & 0 \end{bmatrix} \right)_{\mu\nu},$$

kde jaksi pravou stranou myslíme $\mu\nu$-tou složku matice v závorkách. Uvažujte skalární součin daný Minkowského metrikou, tedy:

$$g_{\mu\nu} = \eta_{\mu\nu} = \left(\begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \right)_{\mu\nu}.$$

Spočtěte hodnotu výrazu v proměnných E a B:

$$\frac{1}{4} F_{\mu\nu} F^{\mu\nu} = ?$$

Definujme Hodgeův duál k $F_{\mu\nu}$ jako:

$$\tilde{F}^{\mu\nu} = \left(\begin{bmatrix} 0 & -B_x & -B_y & -B_z \\ B_x & 0 & -E_z & E_y \\ B_y & E_z & 0 & -E_x \\ B_z & -E_y & E_x & 0 \end{bmatrix} \right)_{\mu\nu} .$$

Spočtěte hodnotu výrazu v proměných E a B:

$$\frac{1}{4} F^{\mu\nu} \tilde{F}_{\mu\nu} = ?$$

Radim N; `radim05@post.cz`
odevzdávejte do odevzdávátka

Kontakty:

M&M, OPMK, MFF UK
Ke Karlovu 3
121 16 Praha 2

E-mail: mam@matfyz.cz
Web: mam.matfyz.cz
FB: casopis.MaM

matfyz
M&M