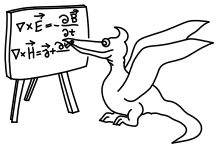

Seriál: Numerické metody a počítačové simulace

Úvod

S třicátým prvním ročníkem FYKOSu přichází i nový seriál, v němž se budeme zabývat tématem, které zasahuje téměř do všech ostatních fyzikálních oborů – jedná se o počítačovou fyziku. Fyzika byla jednou z prvních oblastí vědy, která začala počítače naplno využívat a dnes jim vděčíme za velkou část pokroku, kterého jsme na poli experimentu i teorie v posledním půlstoletí dosáhli. Proto jsme usoudili, že ačkoli bylo toto téma již ve FYKOSích seriálech zpracováno[1] (ročníky VIII. a XXI.), neuškodí se k němu opět vrátit.

V průběhu roku budeme sledovat paralelně dvě různé linie: v jedné se budeme zabývat numerickými metodami a v druhé počítačovými simulacemi. V numerice vás seznámíme se základními úlohami, které by měl každý fyzik umět řešit, jako je hledání kořenů algebraických rovnic, integrace diferenciálních rovnic a další. Simulace budou poněkud hravější se zaměřením především na stochastickou metodu Monte Carlo a na jednoduché modely fyzikálních i ne-tak-fyzikálních procesů, jako je perkolace, růst krystalů, tvorba kolon v silničním provozu, šíření lesních požárů a mnoha dalších.

Nemá smysl zastírat, že vzhledem ke zvolenému tématu nebude možné se zcela vyhnout programování. Ačkoli se budeme snažit, aby každý díl seriálu obsahoval dostatek analyticky řešitelných úloh (tj. řešitelných bez pomoci počítače), bude s příběhem roku přibývat těch, které vyžadují sepsání krátkého skriptu. Nemíníme zde suplovat KSP[2], proto Vás po krátkém úvodu do programování prezentovaném níže odkazujeme na internetové tutoriály. Pokud to myslíte s fyzikou vážně, tak se programování v budoucnu nevyhnete, a je lepší začít dříve než později. Pro bojácné ještě uveďme, že na vyřešení některých numerických úloh bude stačit Microsoft Excel či jiný tabulkový editor (nicméně tento přístup nedoporučujeme).

To je vše, co jsme Vám chtěli úvodem sdělit. Doufáme, že pro Vás bude letošní seriál poučný i zábavný.

Základy počítačové fyziky

Počítače jsou dnes již běžnou součástí našich životů, nejinak je tomu ve vědě a s počítačovou fyzikou a numerickými metodami se setká nejspíše každý fyzik. Do této oblasti totiž nespadají pouze numerické simulace, ale i všednější problémy jako vyčíslení nějaké složitější funkce, či nalezení kořenů algebraické rovnice, kterou nelze vyřešit analyticky.[3] Může se zdát, že jde v tomto případě o triviální problémy řešitelné selským rozumem, a nějaké speciální studium numerických metod k jejich vyřešení snad ani není potřeba. Nicméně, jak sami brzy poznáte, numerické metody bývají mocné, ale zrádné, ve výjimečných případech může jít i o život.[4] Proto je vhodné se s nimi alespoň trochu seznámit a zjistit, jak fungují. **Ale naprosto nutné je jim nikdy bezmezně nevěřit a vždy mít nějakou kontrolu, či intuici plynoucí z daného fyzikláního problému, že výsledek získaný numerickou cestou není zcela špatně.**

[1] `http://fykos.cz/ulohy/archiv`

[2] Korespondenční seminář z programování MFF UK, `https://ksp.mff.cuni.cz/`.

[3] Jako příklad poslouží rovnice $x = \sin x$.

[4] `http://www-users.math.umn.edu/~arnold/disasters/`

Tato zrádnost numerických metod plyne z numerických chyb. Ty mohou vznikat dvěma hlavními způsoby:

- chyby metody – způsobené aproximacemi oproti analytickému řešení
- zaokrouhlovací chyby

Reprezentace čísel v počítači

Zaokrouhlovací chyby jsou způsobeny tím, že desetinná čísla v počítači nejsou uložena přesně. Nejmenší jednotkou informace v počítači, se kterou lze přímo operovat, je *byte*. Ten obsahuje 8 bitů, chlívečků, do kterých můžeme zapsat jedničku, nebo nulu. Přirozenou číselnou soustavou počítače tedy není desítková soustava, jako u člověka,[5] ale soustava dvojková.

Čísla v počítači jsou dvou běžných typů, celá (*integer*) a reálná (s plovoucí desetinnou čárkou, *float*). Celá čísla dokážeme uložit přesně v nějakém rozsahu, závisejícím na velikosti použitého bloku paměti. Např. do jednoho bytu dokážeme uložit čísla od 0 do 255, nebo od -128 do 127. Většinou se ale používají celá čísla o délce 4 byty (integer) nebo 8 bytů (long).

Reálná čísla se ukládají v tzv. semilogaritmickém tvaru

$$s \cdot m \cdot \beta^e \,,$$

kde s je znaménko, m je mantisa, e exponent a β základ číselné soustavy (v našem případě $\beta = 2$). Mantisa i exponent pak mají pevně danou velikost paměti, závisející na použitém datovém typu. Ve skutečnosti jsou čísla uložena o něco složitěji [6], nám ale postačí tato základní představa. Tento způsob uložení zajišťuje udržení počtu platných cifer i pro čísla lišící se o mnoho řádů. Například typ *double*, který budeme výhradně používat, dokáže uložit kladná i záporná čísla s absolutní velikostí od přibližně 10^{-324} do 10^{308} s přesností na 15 až 16 platných cifer. Tento údaj se nazývá *strojová přesnost* a je hlavním údajem ovlivňujícím velikost zaokrouhlovacích chyb.

Jako ilustrace, že k zaokrouhlení dochází, poslouží výpočet 0,1 + 0,2. V double precision obdržíme výsledek 0,30000000000000004. Proč zde ale k zaokrouhlení došlo, když 0,1 i 0,2 jsou přesné tvary? Potíž je v tom, že jde o přesné tvary v desítkové soustavě, počítač ale pracuje ve dvojkové soustavě, a v ní mají obě čísla nekonečný periodický rozvoj. Tento příklad demonstruje, že nemá smysl zjišťovat, zda se dvě reálná čísla přesně rovnají.

Typy zaokrouhlovacích chyb

Pokud bychom byli schopni udržet přesnost na 16 platných cifer po celou dobu výpočtu, měli bychom vyhráno. Problém je, že existují operace, které tuto přesnost za určitých podmínek nezachovávají. Nyní se podíváme na nejběžnější typy takových zaokrouhlovacích chyb, konkrétně na cancellation, ordering a smearing.

Pokud se omezíme na základní matematické operace sčítání, násobení, dělení a odčítání dvou kladných čísel, zjistíme, že první tři operace pro libovolně velké operandy přesnost více méně zachovají.[7] Pokud se ale rozhodneme odečíst dvě blízká čísla, lišící se například až na 10. platné cifře, zjistíme, že výsledek bude mít přesnost pouze $5 - 6$ platných cifer, neboť pouze o těchto cifrách byla nějaká informace v původních operandech. Tento problém se nazývá *cancellation*.

[5] I když my také dokážeme prsty „vypnout“ a „zapnout“ jako bity. Sami si zkuste, že na prstech rukou dokážete napočítat do $2^{10} - 1 = 1023$.

[6] https://www.topcoder.com/community/data-science/data-science-tutorials/representation-of-integers-and-reals-section-2

[7] Nějaká chyba vznikne při každé operaci, v ideálním případě ale bude v řádu strojové přesnosti.

Náš příklad je poněkud přehnaný, nicméně v praxi je cancellation problémem i pro čísla, co jsou jen řádově stejná, neboť se vzniklá chyba bude pravděpodobně s přibývajícími operacemi dále zvětšovat. Cancellation lze odstranit pouze tak, že problematickou operaci přepíšeme, aby neobsahovala odčítání blízkých čísel. Pro ilustraci cancellation si vezměme kvadratickou rovnici $x^2 + 10^6 x + 2 = 0$. Pokud pro nalezení kořenů použijeme klasický vzorec

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a},$$

pak, protože a i c jsou malá v porovnání s b, nastane pro jeden z kořenů cancellation při odečítání $-b$ a diskriminantu. Protože pro kvadratickou rovnici platí $x_1 x_2 = c/a$, můžeme problematický kořen vypočítat jako

$$x_2 = \frac{c}{a x_1}.$$

V původním výpočtu při použití double precision získáme kořeny $x_1 = -9{,}9999999999799998 \cdot 10^5$ a $x_2 = -2{,}0000152289867401 \cdot 10^{-6}$, zatímco s použitím opravy $x_2 = -2{,}0000000000040000 \cdot 10^{-6}$. Výsledky se tedy liší již na šesté platné cifře. Poznamenejme, že jsme se hrozby cancellation zcela nezbavili. Pokud totiž $b^2 \approx 4ac$, pak nastane těžko odstranitelná cancellation již při výpočtu diskriminantu.

Dalším častým problémem je *ordering*. Jeho důsledkem je, že u sčítání závisí na pořadí sčítanců. Nejlépe si jej demonstrujeme na příkladu, kde naším úkolem bude spočítat částečný součet prvních N členů řady

$$\sum_{k=1}^{N} \frac{1}{k}.$$

Použijeme $N = 10^7$ a single precision (datový typ float, strojová přesnost 10^{-7})[8]. Pokud budeme řadu sčítat klasicky „odpředu“, tedy od nejmenších k, dostaneme výsledek[9] $1{,}5403682709 \cdot 10^1$. Protože pro $N \to \infty$ řada diverguje (její součet je nekonečno), očekávali bychom, že pro $N = 10^8$ obdržíme větší částečný součet. Realita je ale taková, že od $N = 10^7$ se součet nijak nezměnil. A nezměnil by se ani pro libovolné větší N, obdrželi jsme tedy vlastně „výsledek“, že $\infty \doteq 15{,}4$!

Co se vlastně pokazilo? V každém kroku sčítáme dvě čísla, dosavadní součet a další člen posloupnosti. A zatímco součet postupně roste, členy posloupnosti klesají. Při součtu velkého a malého čísla pak z menšího čísla neuložíme celou jeho přesnost, ale jen to, co se vejde do přesnosti většího z čísel. Při jednom součtu by byla tato chyba zanedbatelná,[10] ale jak součet provádíme mnohokrát, chyba narůstá. Řešením je sčítat řadu „odzadu“. Tak postupně roste součet i členy posloupnosti a my v každém kroku sčítáme ne tak rozdílná čísla. Tímto postupem dostaneme pro $N = 10^7$ součet $1{,}4392651558 \cdot 10^1$ a pro $N = 10^8$ součet $1{,}8807918549 \cdot 10^1$. Pokud použijeme double precision, dostaneme pro $N = 10^8$ součet $1{,}8997896414 \cdot 10^1$. Je tedy vidět, že ani druhý způsob není zcela ideální, rozhodně je ale přesnější, než způsob první. Chyba v druhém postupu je nejspíš již neodstranitelnou zaokrouhlovací chybou. Pokud vezmeme v úvahu, že v každém kroku může vzniknout chyba 10^{-8} a my provedli 10^8 kroků, pak můžeme očekávat chybu až řádu 1. Budiž toto varováním proti používání single precision ve vědeckých výpočtech.

[8] Problém by samozřejmě nastal i při použití double precision, ale nebyl by tak výrazný.

[9] Pro názornost je uvedeno více platných cifer, než je strojová přesnost.

[10] Na úrovni přesnosti většího z čísel.

Posledním problémem, který zmíníme, je *smearing*. Nastává, pokud sčítáme mnoho kladných a záporných čísel, jejichž součet je malý. Opět si jej ukážeme na příkladu. Funkci $\exp(x)$ lze zapsat pomocí tzv. Taylorova rozvoje

$$\exp(x) = 1 + x + \frac{x^2}{2} + \cdots = \sum_{k=0}^{\infty} \frac{x^k}{k!}\,.$$

Pokud se pokusíme v double precision spočítat $\exp(20)$ pomocí prvních 100 členů rozvoje, dostaneme výsledek $4{,}85165195409790158{\cdot}10^8$, což je blízko správné hodnotě $4{,}85165195409790277{\cdot}10^8$. Pokud se ale pokusíme stejnou metodou spočítat $\exp(-20)$, dostaneme $6{,}13825973860946424{\cdot}10^{-9}$, což se od správné hodnoty $2{,}06115362243855786 \cdot 10^{-9}$ velmi liší. Problém spočívá v tom, že se v rozvoji střídají velké kladné a záporné členy, které se navzájem téměř vyruší, což vede ke ztrátě přesnosti. V tomto konkrétním případě lze využít identity $\exp(-x) = 1/\exp(x)$, což vede na $\exp(-20) = 2{,}06115362243855828 \cdot 10^{-9}$. Tento výsledek je již blízko správné hodnotě.

Stabilita algoritmu a podmíněnost úlohy

Jak jsme viděli, různé algoritmy jsou různě citlivé na chybu vstupu. Této vlastnosti se říká *stabilita algoritmu*. Algoritmus je stručně řečeno[11] stabilní, pokud malá chyba vstupu vždy vyprodukuje pouze malou chybu výstupu. Naším cílem je tedy pokud možno nalézt pro náš problém stabilní algoritmus, který jej řeší. Otázka zní, existuje pro každý problém stabilní algoritmus? Odpověď bohužel zní, ne. Problémům, pro které existuje stabilní algoritmus říkáme *dobře podmíněné*, ostatním *špatně podmíněné*. Příkladem dobře podmíněné úlohy je numerická integrace, příkladem špatně podmíněné úlohy řešení diferenciálních rovnic, či numerická derivace. I špatně podmíněné úlohy ale potřebujeme občas řešit, jen je potřeba si v takovém případě počínat obzvlášť opatrně, jak uvidíme v dalších dílech seriálu.

Složitost algoritmu

Dnešní vědecké výpočty mohou běžet i několik týdnů, či dokonce měsíců a spotřebují gigabyty operační paměti. Proto je důležité algoritmy navrhovat s ohledem na jejich časovou a paměťovou složitost. My tu sice takto náročné výpočty dělat nebudeme, přesto je dobré si jisté základy osvojit. Zaměříme se přitom především na časovou složitost.

Začněme jednou radou. Pokud budete v praxi řešit jakýkoliv problém, pro který existuje standardní algoritmus, nesnažte se jej implementovat sami, ale vždy použijte nějakou knihovnu pro numerické výpočty. Ušetříte si tím spoustu času a nervů, máte jistotu, že algoritmus je implementovaný správně a navíc výpočet bude rychlejší, protože autoři knihoven použili mnoho různých triků a optimalizací. V tomto seriálu si ale z pedagogických důvodů napíšeme implementace vlastní.

Nyní již k časové složitosti. *Časová složitost* algoritmu je doba jeho běhu v závislosti na velikosti vstupu. Dobu jeho běhu můžeme odhadnout počtem základních operací, které se musejí vykonat. Některé operace[12] jsou pomalejší než jiné, v numerických výpočtech nás ale zajímají především aritmetické operace. Tam platí, že dělení je o chlup pomalejší než násobení a to je o chlup pomalejší než sčítání a odčítání. Vždy ale mějme na mysli, že prioritou je zajistit

[11] Existují různé druhy stability, nicméně nám postačí tato základní intuitivní představa.

[12] Především ty, které interagují s něčím mimo náš program, například výpis do souboru, či alokace paměti.

stabilitu algoritmu, až poté jeho rychlost. Rychle získaný výsledek je nám k ničemu, pokud je špatně.

Učebnicovým příkladem, kde se optimalizace počtu operací může vyplatit, je vyčíslování polynomů. Mějme polynom $5x^3 + 2x^2 + 8x + 6$. Pokud jej vyčíslíme pro nějaké x takto přímočaře, budeme potřebovat $3 + 2 + 1 = 6$ násobení[13] a 3 sčítání. Také ale můžeme výraz zapsat jako $6 + x(8 + x(2 + 5x))$. Tímto způsobem potřebujeme pouze 3 násobení a 3 sčítání. Tomuto způsobu se říká *Hornerovo schéma*.

Většinou nás ale nezajímá přesný počet proběhlých operací, ale pouze trend, jak časová složitost roste pro velký vstup. Tomuto trendu se pak říká *asymptotická časová složitost*. Jako příklad si vezměme výše zmíněné vyčíslení polynomu. Velikostí vstupu je zde počet členů polynomu. Při tradičním způsobu je počet sčítání $N - 1$, kde N je počet členů polynomu. Počet násobení je ale $N(N - 1)/2$. Celkově tedy potřebujeme $0{,}5N^2 + 0{,}5N - 1$ operací, což se pro velká N chová zhruba jako N^2. (Asymptotická) časová složitost prvního algoritmu je tedy kvadratická. Pro Hornerovo schéma potřebujeme $N - 1$ násobení a $N - 1$ sčítání, dohromady tedy $2N - 2$ operací. Časová složitost je tedy lineární. Pro velké polynomy se nám tedy vždy vyplatí Hornerovo schéma.

Předvedli jsme si, že pro potřeby asymptotické časové složitosti nás zajímá pouze nejrychleji rostoucí člen, a to bez konstanty před ním. To můžeme formálně vyjádřit pomocí tzv. notace velkého O. Lineární, resp. kvadratickou časovou složitost bychom zapsali jako $O(N)$, resp. $O(N^2)$. S touto notací se ještě setkáme i mimo kontext časové složitosti. Vždy ale bude znamenat to, že nás zajímá pouze trend a uvádíme tedy pouze nejvíce signifikantní člen bez konstanty.[14]

Simulace v počítačové fyzice

Fyzika popisuje přírodní jevy za pomoci matematického aparátu. Obvykle je exaktní popis situace příliš složitý a pracujeme pouze s modelem, který zachycuje všechny podstatné vlastnosti studovaného jevu. Pokud není možné tento model v ruce na papíře v rozumném čase vyřešit a jeho další redukce není přípustná, vstupují do hry počítačové simulace. Jako *počítačovou simulaci* budeme tedy nadále označovat program, který na základě matematického modelu počítá veličiny charakteristické pro studovaný systém. Pro konkrétní představu může jít o časovou závislost teploty chladnoucího hrnku čaje nebo o střední počet částic vyzářený radioaktivním izotopem za určitý časový interval.

Existuje nepřeberné množství úloh, které lze řešit za pomoci simulací, a ještě větší množství numerických a statistických metod, které k jejich řešení můžeme použít. My se zde budeme věnovat takovým úlohám, které nevyžadují oborové znalosti z fyziky a pouze základní znalosti programování. Vyhneme se proto například molekulové dynamice, elektromagnetickému vlnění a dalším zajímavým, ale náročným oblastem. Budeme ovšem dbát na to, aby naše modely měly fyzikální motivaci a aplikaci.

Klasifikace simulací

Než se začneme věnovat konkrétním metodám a modelům, rozdělíme si simulace do několika kategorií.

[13] Mocnění celým číslem je v podstatě opakované násobení a také se jej tak většinou vyplatí implementovat. Cvičení pro zájemce: jak rychleji počítat hodně velké mocniny?

[14] Detailní čtení: `https://www.topcoder.com/community/data-science/data-science-tutorials/computational-complexity-section-2/`

Simulace můžeme dělit na diskrétní a spojité. *Spojitá simulace* je obvykle založena na diferenciální rovnici, pomocí níž dokážeme předpovědět stav v systému v libovolném čase. Jednoduchým příkladem nám budiž Newtonova pohybová rovnice

$$\boldsymbol{F}(t) = m\ddot{\boldsymbol{x}}(t) = m\frac{\partial^2 \boldsymbol{x}}{\partial t^2}\,, \tag{1}$$

kde $\ddot{\boldsymbol{x}}$ značí zrychlení hmotného bodu (druhá derivace polohového vektoru $\boldsymbol{x}$). Při numerickém řešení této rovnice[15] volíme konečné časové kroky, v nichž hledáme polohu bodu, ale nejsme omezeni tím, jak velké časové kroky zvolíme. Omezuje nás pouze přesnost ukládání číselných hodnot v počítači a časový horizont. Oproti tomu *diskrétní simulace* hledají stav v systému pouze v předem pevně daných časových úsecích. Například model predikující počet obyvatel Evropy v následujících letech by mohl být založen na rovnici

$$P_{i+1} = AP_i\left(1 - \frac{P_i}{B}\right)\,. \tag{2}$$

Proměnná P_i zde označuje počet lidí v roce i, A a B jsou parametry. Tyto typy rovnic se nazývají diferenční[16] a jsou diskrétní obdobou diferenciálních rovnic. Jelikož v našem světě plyne čas spojitě, jsou diskrétní časové vývoje vždy aproximací, která s sebou přináší ztrátu přesnosti, ale také zrychlení výpočtu.

Pro zvídavé: Rovnice (2) představuje takzvaný logistický růst. Časový vývoj získáme nejsnáze převedením na diferenciální rovnici

$$\frac{\mathrm{d}P}{\mathrm{d}t} = AP(1 - P/B)\,.$$

Separací proměnných a integrací parciálních zlomků získáme řešení

$$P = \frac{BP_0\mathrm{e}^{At}}{B + P_0\left(\mathrm{e}^{At} - 1\right)}\,.$$

Tento model se používá se například v biologii k simulaci vývoje zvířecích populací. Parametr A pak představuje růstový koeficient a parametr B koeficient saturace (přemnožení vede k nedostatku potravy a životního prostoru). Zajímavější model získáme přidáním predace. Model kořist-predátor je dán soustavou rovnic

$$\begin{aligned} K_{i+1} &= AK_i - BK_iP_i\,,\\ P_{i+1} &= CK_iP_i - DP_i\,. \end{aligned} \tag{3}$$

Kořist je K (např. králík), predátor P (např. puma) a A, B, C, D jsou parametry. K tomuto modelu se vrátíme v pozdějších dílech seriálu.

Diskretizace se netýká pouze času, ale také prostoru a dalších parametrů. Dobrým příkladem je Isingův model, který má význam při simulování magnetických vlastností pevných látek ve vnějším magnetickém poli. V tomto modelu je prostor rozdělen n-rozměrnou mříží a pro každou z jejích buněk určujeme hodnotu elektronového spinu.[17] Isingův model patří do významné kategorie mřížkových modelů, na něž ještě přijde řeč.

Dále existuje dělení na deterministické a stochastické simulace. *Deterministické simulace* jsou založené na systému matematických pravidel, pomocí kterého dokážeme v každém časovém

[15] Řešit Newtonovu rovnici se naučíte později v tomto seriálu.

[16] Obecně můžeme mluvit o rekurentních vztazích, kde každý člen sekvence je popsán pomocí funkce závisející na (ne nutně všech) předchozích členech.

[17] Kvantově-mechanická veličina, která se svými vlastnostmi podobá momentu hybnosti.

kroku určit, v jakém stavu se bude systém nacházet v kroku následujícím. Pro případ diskrétního modelu můžeme obecně psát

$$x_{t+1} = f(t, x_t, x_{t-1}, \ldots, x_0)$$

kde f je funkce, která jednoznačně určuje hodnotu veličiny x. *Stochastická simulace* je založena na náhodě. Máme sice předpis, který udává vývoj systému (jinak bychom nemohli simulovat!), ale tento předpis mluví pouze o pravděpodobnosti, se kterou se bude systém v následujícím časovém kroku nacházet v určitém stavu. Příkladem stochastického procesu je sekvence hodů kostkou. Pokud jsme již například v předchozích hodech naházeli v součtu hodnotu 30, tak víme, že v příštím hodu dosáhne celkový součet hodnoty v rozmezí od 31 do 36, nedokážeme však říci, jaká z těchto hodnot to bude. Víme pouze to, že pravděpodobnost nabytí jakékoli jedné z těchto hodnot je 1/6 (není-li kostka „cinklá“).

K tomu, abychom správně pochopili stochastické modely, si potřebujeme osvojit základy pravděpodobnosti a statistiky. Než se do toho pustíme, tak dodejme, že simulace lze dělit do dalších skupin. Prudce rozvíjejícím se oborem jsou *kvantové simulace*. Kvantová statistika nebude předmětem tohoto seriálu, proto zmíníme-li někdy v dalších dílech seriálu simulace kvantových systémů, tak pouze jako zajímavost. Není-li simulace kvantová, nazývá se *klasická simulace*.

Ještě se na chvíli zdržme u pojmů kvantová simulace a deterministická simulace. S kvantovou fyzikou je neodmyslitelně spjata Schrödingerova rovnice. Možná vás nyní zarazí, že simulace založená na Schrödingerově rovnici je deterministická. Každý systém popsaný diferenciální rovnicí je deterministický. Platí však, že vlnová funkce, která je výsledkem řešení Schrödingerovy rovnice, nepopisuje jednoznačně stav fyzikálního systému, tj. vztah mezi vlnovou funkcí a pozorovatelnou veličinou je nedeterministický, naše simulace nikoli.

Podobné zmatení může nastat ohledně simulací chaotických systémů – tyto simulace jsou též deterministické. Chaotičnost spočívá ve vysoké citlivosti systému na počátečních hodnotách. Proto pokud pro stejné počáteční podmínky dostaneme jiný výsledek simulace, nejedná se o náhodné chování, ale o nepřesnou počítačovou reprezentaci čísel popisujících stav systému.

Pokud v sobě rovnice popisující časový vývoj systému zahrnuje náhodné chování, jedná se o stochastickou rovnici. Metodami řešení těchto rovnic se zabývá stochastický kalkulus.

Základní pojmy z pravděpodobnosti a statistiky

Statistika a teorie pravděpodobnosti tvořili významnou část seriálu minulého ročníku. Zde shrneme pouze to, co budeme potřebovat pro pochopení stochastických procesů a pro zpracování výsledků simulací. Cílem je pouze intuitivní pochopení a praktická aplikace – pokud vám bude připadat následující text příliš složitý, zaměřte se pouze na význam výrazů (8) až (11), tj. na pojmy střední hodnota, rozptyl, zákon velkých čísel a statistická definice pravděpodobnosti. Pokud vás naopak pravděpodobnost zajímavá více do hloubky, přečtěte si seriál 30. ročníku.[18]

Abychom se neutopili v abstrakci, budeme si pojmy z teorie pravděpodobnosti vysvětlovat na házení šestistěnnou hrací kostkou. Událost, kdy hodíme kostkou a padne konkrétní číslo, nazýváme *elementární jev*. Množina $\Omega = \{1, 2, 3, 4, 5, 6\}$ se potom nazve *prostor elementárních jevů*. Označme $\mathcal{A}$ množinu všech neprázdných podmnožin[19] vybraných z Ω. Jako *náhodný jev* A označíme libovolnou množinu $A \subseteq \mathcal{A}$. Například náhodný jev „Na kostce padlo sudé číslo“

[18] Seriál naleznete na adrese `http://fykos.cz/ulohy/archiv`. Pokud vám ani to nestačí, doporučujeme knihu *Zvára, Štěpán: Pravděpodobnost a matematická statistika.*

[19] Počet těchto podmnožin je $2^n - 1$, kde n je počet prvků množiny Ω.

je reprezentován množinou $A = \{2, 4, 6\}$. *Doplňkový jev* k jevu A je množina $A^c = \Omega - A$ a v uvedeném příkladu by šlo o náhodný jev „Na kostce padlo liché číslo“, $A^c = \{1, 3, 5\}$.

Je zřejmé, že jev „Padlo číslo menší než čtyři“ bude nastávat častěji než jev „Padla šestka“. Abychom tuto skutečnost nějak kvantifikovali, zavádíme *pravděpodobnostní míru* P. Pravděpodobnostní míra má tři důležité vlastnosti:

- Je nezáporná, $P(A) \geq 0$.
- Je normovaná, $P(\Omega) = 1$.
- Je aditivní na disjunktních[20] množinách, $P(A_1 + A_2 + \dots) = P(A_1) + P(A_2) + \dots$

Výraz $P(A)$ označuje *pravděpodobnost, že nastane jev* A. Teorie nám neříká, jak volit Ω, $\mathcal{A}$, P. Pro případ házení kostkou bude platit $P(A) = |A|/|\Omega|$, kde svislé čáry značí kardinalitu, tedy počet prvků v množině.

Dále zavádíme *podmíněnou pravděpodobnost*[21]

$$P(A|B) = \frac{P(A \cap B)}{P(B)}\,. \tag{4}$$

Čteme: Pravděpodobnost, že za předpokladu nastání jevu B nastane jev A, je rovna pravděpodobnosti, že nastanou jevy A a B zároveň, dělené pravděpodobností jevu B. Nechť jev A znamená, že padne sudé číslo, a B znamená, že padne číslo větší než 3. Potom

$$P(A|B) = P(\{2,4,6\}|\{4,5,6\}) = \frac{P(\{2,4,6\} \cap \{4,5,6\})}{P(\{4,5,6\})} = \frac{P(\{4,6\})}{P(\{4,5,6\})} = \frac{\frac{2}{6}}{\frac{3}{6}} = \frac{2}{3}\,.$$

O jevech říkáme, že jsou *nezávislé*, pokud platí $P(A|B) = P(A)$.

Mějme množinu neslučitelných jevů B_1, B_2, …, z nichž alespoň jeden nastává (tedy $B_i \cap B_j = \emptyset\, \forall i, j$ a $B_1 \cup B_2 \cup \dots = \Omega$). Pak *Věta o úplné pravděpodobnosti* tvrdí

$$P(A) = \sum_i P(A|B_i)P(B_i)\,. \tag{5}$$

Toto tvrzení vám může pomoct při řešení jedné z bonusových úloh.

Označme nyní prvky množiny Ω malým písmenem ω. Nechť existuje funkce X, která každému prvku ω přiřadí reálné číslo, formálně $X : \Omega \to \mathbb{R}$. Funkce X se nazývá *náhodná veličina* a číslo $X(\omega)$ je *realizace náhodné veličiny*. Význam náhodné veličiny si opět ilustrujeme na hodu kostkami. Nyní budeme mít dvě šestistěnné kostky a množina Ω bude tedy obsahovat celkem 36 prvků: (1,1), (1,2), (1,3), (1,4), (1,5), (1,6), (2,1), (2,2), (2,3), …, (6,6). Jednotlivé prvky množiny Ω značme ω_i a jejich složky ω_i^j. Tedy například $\omega_9^1 = 2$, $\omega_9^2 = 3$. Ve většině stolních her nás zajímá součet ok na hozených kostkách. Tento součet představuje náhodnou veličinu X definovanou vztahem

$$X(\omega_i) = \omega_i^1 + \omega_i^2 \;\forall i = 1, \dots, 36\,.$$

Pro každé $x \in \mathbb{R}$ definujeme *distribuční funkci* $F(x) = P(\{\omega :\, X(\omega) \leq x\})$. V našem příkladu se jedná o diskrétní úlohu, náhodná veličina X tedy nabývá hodnot $x_1 = 2$, $x_2 = 3$, …, $x_{11} = 12$ a píšeme[22]

$$P(\{\omega :\, X(\omega) = x_i\}) = p_i \equiv P(x_i)\,. \tag{6}$$

[20] Disjunktní množiny nemají žádný společný prvek, tj. jejich průnikem je prázdná množina.
[21] Každou pravděpodobnost můžeme chápat jako podmíněnou, protože $P(A|\Omega) = P(A)$.
[22] Často se také používá zkrácený zápis $P(X = x_i)$.

Funkce $P(x)$ se nazývá *pravděpodobnostní funkce*. Pokud bychom pracovali se spojitou náhodnou veličinou, definovali bychom distribuční funkci pomocí integrálu

$$F(x) = \int_{-\infty}^{x} f(t)\mathrm{d}t\,, \tag{7}$$

kde se $f(t)$ nazývá *hustota pravděpodobnosti* a je spojitou obdobou pravděpodobnostní funkce.

Náhodnou veličinu lze charakterizovat pomocí mnoha různých ukazatelů, zde si uvedeme pouze dva. *Střední hodnota* se definuje pro diskrétní a spojitá rozdělení jako

$$\mathrm{E}X = \sum_i x_i p_i \equiv \mu\,, \quad \mathrm{E}X = \int_{-\infty}^{\infty} x f(x)\mathrm{d}x \tag{8}$$

a *rozptyl* jako

$$\mathrm{Var}X = \sum_i (x_i - \mu)^2 p_i = \mathrm{E}(X-\mu)^2 \equiv \sigma^2\,, \quad \mathrm{Var}X = \int_{-\infty}^{\infty} (x-\mu)^2 f(x)\mathrm{d}x \tag{9}$$

přičemž odmocnina rozptylu σ se nazývá *směrodatná odchylka*. Laicky řečeno rozptyl udává, jak moc široce jsou jsou x_i rozprostřeny okolo střední hodnoty.

Význam střední hodnoty nejlépe pochopíme pomocí tzv. *zákona velkých čísel*. Ten nám říká, že aritmetický průměr výsledků získaných měřením náhodné veličiny X se s rostoucím počtem pokusů N blíží střední hodnotě $\mathrm{E}X$. Formálně zapsáno

$$E \lim_{N\to\infty} \left(\frac{1}{N} \sum_i^N X_i \right) = \mu\,. \tag{10}$$

Tento zákon nám zaručuje, že pro dostatečně velký počet pokusů získáme dobrý odhad střední hodnoty, což je nezbytný předpoklad pro simulace i fyzikální experimenty.

V této kapitole jsme prozatím vycházeli z tzv. Kolmogorovovy axiomatické definice pravděpodobnosti, která je, jak jsme si ukázali, velmi obecná a neříká nám, jak pro účely simulace definovat Ω, $\mathcal{A}$ a P. Proto v praxi používáme *statistickou definici pravděpodobnosti*, podle níž je pravděpodobnost jevu dána jeho *relativní četností* při dostatečném počtu pokusů. Tedy pokud v N pokusech nastane M-krát jev A, pak platí

$$\lim_{N\to\infty} \frac{M}{N} = P(A)\,. \tag{11}$$

Nakonec si uveďme dvě důležitá diskrétní rozdělení:

- *Rovnoměrné rozdělení* je takové rozdělení, v němž náhodná veličina může nabývat n hodnot, a to s pravděpodobností $1/n$ pro každou z nich. Pro pohodlnost popisu řekněme, že množinu realizací náhodné veličiny tvoří všechna celá čísla z intervalu $\langle a, b\rangle$, kde a, b jsou také celá, tedy $n = b - a + 1$. Pravděpodobnostní funkci potom můžeme zapsat jako

$$P(k) = \begin{cases} \frac{1}{n} & \forall k \in \{\mathbb{Z} \cap \langle a, b\rangle\} \\ 0 & \text{jinak} \end{cases} \tag{12}$$

  a distribuční funkci jako

$$F(k) = \frac{\lfloor k \rfloor - a + 1}{n}\, \forall k \in \langle a, b\rangle\ . \tag{13}$$

Příkladem náhodné veličiny s rovnoměrným rozdělením je hod jednou šestistěnnou kostkou. Střední hodnota rovnoměrného rozdělení je $\mu = (a+b)/2$, rozptyl $\sigma^2 = (n^2-1)/12$.

- *Binomické rozdělení* s parametry n, p je dáno pravděpodobnostní funkcí

$$P(k) = \binom{n}{k} p^k (1-p)^{n-k} \,. \tag{14}$$

Parametr $p \in \langle 0, 1\rangle$ zde představuje pravděpodobnost, že v pokusu s binárním výsledkem 0 nebo 1 dostaneme číslo 1 (a číslo 0 pak s pravděpodobností $1-p$). Parametr n udává, kolikrát jsme pokus provedli. Náhodná veličina X s binomickým rozdělením $P(k)$ potom udává počet úspěchů (hodnota 1) v těchto n pokusech. Typickým příkladem jsou opakované hody mincí, kde např. 1 je panna a 0 orel. Střední hodnota binomického rozdělení je $\mu = np$, rozptyl $\sigma^2 = np(1-p)$.

Generátory pseudonáhodných čísel

Představme si, že žijeme v sedmnáctém století a přijde za námi Jacob Bernoulli a zeptá se nás, s jakou pravděpodobností nám při hodu šesti šestistěnnými kostkami padne číslo dvacet. Nyní máme na výběr: buďto jsme dobří počtáři a nalezneme všechny kombinace, které vedou na tuto hodnotu, nebo jsme hodně trpěliví a budeme kostkami házet tak dlouho, dokud nedostaneme výsledek s uspokojivou přesností. V dnešní době máme naštěstí počítače, a ty mohou kostkou házet za nás. Je tu ovšem jeden problém: jak řekneme počítači, aby při každém hodu kostkou *náhodně* vybral, jaká hodnota padne? Při hodu skutečnou kostkou spočívá náhodnost ve vysoké citlivosti výsledku na rychlosti rotace kostky a úhlu dopadu (kostka se chová chaoticky). Co použijeme jako zdroj náhody v počítači?

Vytvářet náhodná čísla za pomoci počítače můžeme dvěma způsoby. První z nich využívá fyzikální procesy, například tepelný šum v hardwarových součástkách nebo kvantové jevy. Takto získaná náhodná čísla se v angličtině označují jako *true random*, opravdu náhodná (jejich hodnotu nedokážeme předpovědět o nic lépe než výsledek hodu kostkou). Nevýhodou těchto generátorů je, že nedokáží poskytnout dostatečně velké množství náhodných čísel v krátkém čase nebo vyžadují specializovaný hardware.

Obvyklejší je proto používat *generátory pseudonáhodných čísel*. Ty jsou založeny na jistém deterministickém algoritmu, který na základě zadané počáteční hodnoty (tzv. *seed*, může být získán pomocí hardwarového generátoru) generuje sekvenci čísel, která má statistické vlastnosti blízké sekvenci opravdových náhodných čísel. Existuje mnoho testů náhodnosti a zde se jimi nebudeme příliš dopodrobna zabývat. Pro úplnost uveďme, že se může jednat o testování četnosti výskytu jednotlivých čísel a jejich řetězců či rozdělení délky mezer mezi stejnými hodnotami. Jednoduchým vizuálním testem je vykreslení náhodných čísel pomocí černobílého kódování do plochy. Výsledek by měl připomínat šum analogové televize.[23]

Nejjednodušším generátorem pseudonáhodných čísel (a jediným, který si podrobně popíšeme) je *lineární kongruenční generátor*.[24] Tento generátor vytváří sekvenci čísel $\{x_n\}$ na základě rekurentního vztahu

$$x_{n+1} = (ax_n + c) \bmod m \,, \tag{15}$$

[23] Náhodnost generátoru ve vašem internetovém prohlížeči si můžete pomocí vizuálního testu ověřit na adrese `http://www.psychicscience.org/vt2.aspx`.

[24] Název pochází z matematického pojmu kongruence zbytkových tříd. Říkáme, že a je kongruentní s b modulo n, pokud rozdíl $a-b$ je beze zbytku dělitelný číslem n.

kde parametry a, c jsou celá čísla, m je přirozené číslo a mod je operátor udávající zbytek po dělení výrazu číslem m. x_0 je seed, který musí zadat uživatel. Parametry volíme tak, abychom získali co nejdelší neopakující se sekvenci čísel pro libovolný seed. Je zřejmé, že perioda nemůže být delší než m. Pokud chceme, aby generátor měl tuto délku periody pro všechny seedy, musí parametry splňovat následující podmínky:

- Čísla m a c jsou nesoudělná.
- $a - 1$ je dělitelné všemi prvočísly z rozkladu čísla m.
- Pokud je m dělitelné 4, je také $a - 1$ dělitelné čtyřmi.

Fungování generátoru si ukážeme na dvou jednoduchých příkladech. Nejprve volme $a = 6$, $c = 1$, $m = 13$. Tato volba zřejmě nesplňuje druhé kritérium. Pro $x_0 = 0$ dostaneme sekvenci

$$1, 7, 4, 12, 8, 10, 9, 3, 6, 11, 2, 0, \ldots$$

Vidíme, že tato sekvence má periodu $m - 1 = 12$, přičemž v ní chybí číslo 5. Podobný výsledek dostaneme pro každý seed kromě $x_0 = 5$, kdy sekvence obsahuje pouze číslo 5. Volba parametrů $a = 13$, $c = 1$, $m = 6$ splňuje všechna kritéria a sami si jistě snadno ověříte, že pro libovolný seed je perioda maximální. Má ovšem délku pouze $m = 6$. Pokud bychom zvolili $c = 0$, hovořili bychom o *multiplikativním kongruenčním generátoru*. Výpočet čísel v sekvenci je poté jednodušší, ale nesmíme volit $x_0 = 0$.

Nejvyšší efektivity kongruenční generátory dosahují, pokud je m mocnina dvou, např. 2^{32}. Poté totiž v binární soustavě získáme zbytek po dělení prostě tak, že od 31. pozice všechny hodnoty nalevo[25] vynulujeme (v praxi je prostě vůbec nepočítáme). Většina implementací však nevrací čísla v rozsahu od 0 do $m - 1$, ale dělí je hodnotou m, čímž je přeškáluje do intervalu $\langle 0, 1)$. To je velmi výhodné zvláště tehdy, když generátor používáme k simulaci stochastického chování, protože hodnoty pravděpodobnosti jsou také v rozsahu od 0 do 1.

Pečlivě musíme volit i čísla a a c. Výše uvedené podmínky, které na ně klademe, zaručují pouze dlouhou periodu, ne její náhodnost. Pro volbu $a = 1$, $c = 1$, $m = 2^{31}$, $x_0 = 0$, která podmínky splňuje, dostaneme sekvenci 1, 2, 3, 4, ... I bez diskuse je jasné, že nejde o náhodná čísla. Kongruenční generátory, které se v praxi používají, mají pečlivě zvolené hodnoty parametrů a prošly mnoha testy náhodnosti.

Před použitím generátoru pseudonáhodných čísel ve vašem oblíbeném programovacím jazyku si nejprve zkontrolujte, jakou metodu vlastně používáte. Například funkce `rand()` z knihovny `glibc` pro jazyk C je obyčejný lineární kongruenční generátor, který může být pro některé aplikace nedostačující. Python používá ve své funkci `random()` generátor založený na algoritmu Mersenne Twister, který využívá vlastnosti Mersennových prvočísel. Seed se získává ze systémového času.

Na závěr ještě zmiňme, že existují také *generátory kvazináhodných čísel*, jejichž cílem není přiblížit se co nejvíce opravdovým náhodným číslům, ale spíše co nejrovnoměrněji pokrýt zvolenou oblast n-dimenzionálního prostoru. Příkladem jsou například Sobolovy sekvence. Kvazináhodná čísla se s výhodou používají například v metodě Monte Carlo – o té si ale povíme více až v příštím díle.

Drsný úvod do programování

Program je sekvence příkazů, které má počítač vykonat. Aby je mohl vykonat, musí jim rozumět. Počítač ovšem rozumí pouze sekvenci jedniček a nul, tzv. strojovému kódu (posloupnost

[25] Nebo napravo. Záleží na tom, jak ukládáme data do paměti. Více viz heslo *endianita*.

instrukcí pro procesor). Dříve se v něm skutečně programy psaly, nicméně bylo to pro programátora nepohodlné, navíc každá generace počítačů rozuměla jinému strojovému kódu. Proto byly vynalezeny programovací jazyky, které jsou bližší lidskému uvažování a řeči. Dnes tedy programátor napíše pomocí nějakého programovacího jazyka tzv. zdrojový kód. Ten se poté, v závislosti na zvoleném jazyce, buď přímo za běhu interpretuje počítači, či je nejprve přeložen pomocí překladače (kompilátoru) do strojového kódu.

Volba jazyka

Jednotlivé úlohy seriálu můžete řešit v libovolném jazyce, vždy je ale dobré k takovému řešení přiložit zdrojový kód. My budeme v ukázkách používat buď programovací jazyk Python, nebo tzv. pseudokód. Pseudokód je v podstatě zdrojový kód, kde jsou ale některé pasáže z důvodu stručnosti a přehlednosti popsány pouze slovně, nikoliv v daném jazyce. Takový kód samozřejmě není přímo spustitelný na počítači.

Jazyk Python byl zvolen, protože jde o poměrně jednoduchý a přehledný jazyk, navíc dnes dominuje v oblasti data science. Pravděpodobně se s ním v praxi tedy setkáte. Chtěli bychom dopředu upozornit, že styl, jakým budeme zezačátku v Pythonu psát, není pro vědecké výpočty v Pythonu běžný. V praxi je hojně využívaná knihovna NumPy[26] s knihovnou pro vědecké výpočty SciPy[27] a pokročilejší funkcionální konstrukce jazyka. Správné použití těchto prostředků dokáže zrychlit výpočty, nicméně podmínkou je znalost toho, jak je v Pythonu „pod kapotou“ nakládáno s operační pamětí. My si v prvních ukázkách vystačíme bez těchto knihoven, náš zdrojový kód tedy nebude zcela optimalizovaný, zato bude intuitivní a přehledný i pro ty, kteří nemají s Pythonem zkušenost. Později ovšem z důvodu stručnosti začneme knihovnu NumPy v ukázkách používat tam, kde to bude vhodné.

Python je interpretovaný jazyk, to znamená, že zdrojový kód nepřekládáme, ale rovnou spouštíme pomocí tzv. interpreteru. Interpreter Pythonu a kompletní dokumentaci jazyka včetně tutoriálů pro začátečníky najdete na `https://www.python.org`. Python je dnes používán ve dvou vzájemně nekompatibilních verzích 2 a 3. V našich ukázkách budeme používat novější verzi 3. Protože zdrojový kód je čistý text, potřebujeme kromě interpreteru již pouze libovolný textový editor, například Poznámkový blok[28] a můžeme začít programovat. Naše programy budou poměrně krátké, nezatěžujte se tedy výběrem nějakých pokročilejších vývojových prostředí (IDE). Nevyužijeme jejich plný potenciál a pouze by nás mátly.

Základy syntaxe jazyka Python

Jak již bylo řečeno, nebudeme zde suplovat tutoriály, kterých lze na internetu nalézt spousta. Místo toho si rovnou předvedeme ukázku zdrojového kódu a rozebereme si, co tento kód dělá.

```
import math

def e_z_limity(n):
  if n < 1:
    print("My brain hurts")
    exit()
```

[26] `http://www.numpy.org`

[27] `https://www.scipy.org/scipylib/index.html`

[28] V nejhorším případě. Lepší je nainstalovat si kvalitní editor, např. Sublime Text, nebo použít jeden z editorů nainstalovaných v Linuxu.

```
  vnitrek = 1.0+1.0/n
  vysledek = 1.0
  for i in range(n):
    vysledek *= vnitrek
  return vysledek

def e_z_taylora(n):
  suma = 0.0
  faktorial = 1.0
  for i in range(1,n+2):
    suma += 1.0/faktorial
    faktorial *= i
  return suma

for i in range(1,10000):
  print("{} {} {}".format(i,math.e-e_z_limity(i),math.e-e_z_taylora(i)))
```

Program je vykonáván shora dolů po jednotlivých příkazech. Příkaz `import math` zajistí import matematické knihovny. V našem programu ji potřebujeme kvůli konstantě `math.e`, což je Eulerovo číslo. Dále máme prázdný řádek. Ten je uveden pouze pro přehlednost a je počítačem ignorován. Stejně tak jsou ignorovány řádky začínající znakem `#`. Jde o tzv. komentáře a slouží pouze pro snadnější orientaci v kódu. Nicméně doporučujeme naučit se je používat.

Další řádek začíná klíčovým slovem `def`. Jde o definici funkce, kterou jsme nazvali `e_z_limity()`. Tato funkce má jeden parametr `n`. Funkce je označení kusu kódu, tzv. těla funkce, který se provede, pokud tuto funkci zavoláme. Volání funkce se dělá uvedením názvu funkce s kulatými závorkami, které mohou obsahovat hodnoty parametrů. Pokud je parametrů více, jsou odděleny čárkami, jak vidíme například při volání funkcí `range()` nebo `format()`. Tělo funkce[29] je označeno odsazením, tělo funkce `e_z_limity()` tedy končí řádkem `return vysledek`. Příkaz `return` znamená ukončení vykonávání funkce a případné vrácení nějaké hodnoty. Funkce `e_z_limity()` například vrací hodnotu uloženou v proměnné `vysledek`.

Na dalším řádku vidíme konstrukci `if`. Pokud je splněna podmínka, v našem případě `n < 1`, provede se blok příkazů za ní, v našem případě příkazy `print("My brain hurts")` a `exit()`. Pokud podmínka splněna není, blok se přeskočí. Konstrukce `if` může obsahovat i blok `else`, který se provede, pokud podmínka splněna není. Podmínkou může být cokoliv, co vrací hodnoty `True` (pravda) a `False` (nepravda).

Příkaz `print("My brain hurts")` vypíše na obrazovku řádek s textem „My brain hurts“ (bez uvozovek). Uvozovky zde označují, že jde o tzv. řetězec znaků (string). Příkaz `exit()` pak ukončí program. Další dva příkazy vytvoří proměnné pojmenované `vysledek` a `vnitrek` a přiřadí jim hodnotu 1, resp. $1 + 1/n$, kde n je aktuální hodnota parametru `n`. Proměnné jsou zjednodušeně řečeno „škatulky“, kam si můžete ukládat nějakou hodnotu, ať už jde o číslo, řetězec, ... K proměnným můžeme přistupovat vždy pouze v rámci bloku, ve kterém byly definovány, v tomto případě jde o tělo funkce `e_z_limity()`. Povšimněte si, že přiřazujeme hodnotu „1.0“ a ne „1“. Rozdíl je v tom, že „1“ je celé číslo, zatímco „1.0“ je reálné číslo (v double precision). Jak později uvidíme, z kontextu výpočtu chceme, aby `vysledek` a `vnitrek` byly reálné.[30]

[29] Stejně jako každý blok kódu v Pythonu.

[30] Ve skutečnosti by stačilo psát „1“. Důvodem jsou hlubší nuance jazyka Python, který v konkrétních situacích konvertuje celá čísla na reálná nebo naopak. Python 2 se v tom chová jinak než Python 3. Jestli neumíte

Další řádek začíná klíčovým slovem `for`. Jde o tzv. cyklus, tedy blok příkazů, který se dokola opakuje. Ve většině jazyků je opakování prováděno, dokud je splněna podmínka, podobně jako u konstrukce `if`. Python je v tomto odlišný, cyklus je prováděn pro všechny prvky seznamu, či obecně libovolné kolekce hodnot, kterou lze postupně procházet. V našem případě funkce `range(n)` vrátí seznam celých čísel od 0 do $n-1$ včetně. Cyklus je pak v tomto pořadí postupně prochází, dosadí danou hodnotu do proměnné `i` a vykoná blok příkazů, v našem případě příkaz `vysledek *= vnitrek`. Ten vynásobí hodnoty proměnných `vysledek` a `vnitrek` a výsledek uloží znovu do proměnné `vysledek`. Jde pouze o zkrácení zápisu `vysledek = vysledek * vnitrek`. Poznamenejme, že v tomto cyklu proměnnou `i` vůbec nevyužíváme, celý cyklus slouží pouze k tomu, aby příkaz `vysledek *= vnitrek` proběhl n-krát za sebou. Cyklus tedy představuje pouhý výpočet n-té mocniny hodnoty proměnné `vnitrek`.

Poslední příkaz těla funkce je příkaz `return vysledek`. Kdykoliv se při vykonávání funkce narazí na příkaz `return`, nebo se dojde na konec těla funkce, vykonávání funkce se ukončí a program pokračuje v místě, odkud byla funkce zavolána. Pokud je za příkazem return uvedena nějaká konstanta nebo proměnná, je její hodnota dosazena na místo volání funkce. Říkáme, že funkce vrátila hodnotu. Příkladem mohou být různé matematické funkce, například `math.cos()`, která vrátí kosinus parametru, či funkce `range()`, která vrací seznam čísel.

Pokud budeme program sledovat dále, tak následuje definice funkce `e_z_taylora()`. Jejímu tělu byste již měli rozumět, za povšimnutí stojí pouze volání `range()` se dvěma parametry. V tomto případě funkce vrátí seznam celých čísel od 1 do $n+1$ včetně (hodnota $n+2$ už v seznamu není).

Následuje poslední cyklus `for i in range(1,10000)`. Protože se doposud vyskytovaly pouze definice funkcí a příkaz `import`, je toto místem, kde se skutečně začne vykonávat program. S funkcí `print()` jsme se již setkali, cílem tohoto cyklu je tedy 9999krát něco vypsat na obrazovku. Povšimněme si konstrukce `"{} {}".format(. . . )`. Ta vytvoří řetězec, dle vzoru v uvozovkách, ale místo každé dvojice znaků `{}` dosadí v pořadí argumenty předané funkci `format()`. Ta může mít libovolný počet argumentů. Cílem je tedy tisknout hodnoty `i`, `math.e-e_z_limity(i)` a `math.e-e_z_taylora(i)` oddělené mezerou, a to pro `i` nabývající postupně hodnot 1 až 9999.

Seznámili jsme se s konstrukcemi `if` a `for`, definicí funkce, proměnnými a výpisem na obrazovku. Rozhodně nejde o plný výčet konstrukcí jazyka Python, nicméně se jedná o ty nejpoužívanější a nejspíše si s nimi vystačíme po velkou část seriálu. Jsme si vědomi, že šlo opravdu o rychlokurz, proto silně doporučujeme projít si nějaký tutoriál a pokusit se napsat si pár cvičných programů. Strávíte tím sice několik večerů, nicméně základní znalost programování vám brzy vloženou námahu mnohokrát vrátí.

And now for something completely different

Nyní, když zvládneme zdrojový kód přečíst, se podívejme na to, co má tento kód za úkol. Jak nám napoví názvy funkcí, patrně se snažíme vypočítat Eulerovo číslo různými způsoby, vypisujeme pak rozdíl těchto výpočtů od skutečné hodnoty.

Funkce `e_z_limity()` využívá vztahu

$$\mathrm{e} = \lim_{n\to\infty} \left(1+\frac{1}{n}\right)^n ,$$

kdy vyčíslujeme výraz

$$\left(1+\frac{1}{n}\right)^n$$

dobře programovat, raději se na to nespoléhejte.

pro různě velký parametr n. Pro dostatečně velké n bychom měli přibližně dostat hodnotu e. Podmínka zde testuje, jestli hodnota parametru dává smysl. Je zde uvedena zcela účelově pouze pro ukázku konstrukce `if`. Všimněme si, že není zcela neprůstřelná, mohli bychom totiž v parametru předat například řetězec, či reálné číslo. Při psaní programů je obecně dobrým návykem kontrolovat, jestli je vstup validní, nicméně v našich prográmcích se tímto nebudeme většinou zabývat.

Funkce `e_z_taylora()` využívá Taylorova rozvoje exponenciály, který byl zmíněn v kapitole o smearingu. Pokud tento rozvoj vyčíslíme v bodě 1, dostaneme

$$\exp(1) = \mathrm{e} = \sum_{k=0}^{\infty} \frac{1}{k!}\,.$$

Funkce pak skutečně přímočaře vyčísluje částečné součty rozvoje

$$\sum_{k=0}^{n} \frac{1}{k!}$$

v závislosti na hodnotě parametru n vyjadřujícím počet členů rozvoje. Mohli byste v tuto chvíli namítnout, že použitý postup bude náchylný k orderingu. Nicméně pokud program skutečně spustíme, zjistíme, že se pro dostatečně velké n liší výsledek výpočtu od skutečné hodnoty Eulerova čísla v řádu 10^{-16}, tedy v řádu strojové přesnosti. Ordering je tedy v tomto případě zanedbatelný (nejspíš díky tomu, že $1/k!$ klesá děsně prudce) a nebudeme se jím zabývat.

Spuštění programu

Uložme zdrojový kód do textového souboru, například `euler.py`. Poté spusťme příkazový řádek v adresáři, kam jsme uložili náš program. Ten nyní spusťme příkazem `python euler.py`. Uvidíme očekávaný výpis čísel. Program chvíli poběží, neboť výpis na obrazovku je poměrně pomalá operace, zvlášť když ji provádíme téměř desettisíckát[31]. Gratulujeme, právě jste spustili svůj první program. Pokud se vám program nedaří spustit, neváhejte se na nás obrátit, případně postupujte podle některého z tutoriálů pro instalaci a první spuštění pythonu.

Mít výpis čísel pouze na obrazovce je nepraktické. Proto znovu spusťme program, tentokrát příkazem `python euler.py > euler-data.txt`. Tím jsme přesměrovali výstup z programu do souboru `euler-data.txt`, který se vytvořil ve stejném adresáři, jako `euler.py`. Takto uložený výstup je již jednoduché dále používat, například jej můžeme zobrazit v grafu pomocí programu Gnuplot nebo importovat do Excelu, či jiného tabulkového editoru.

Pokud si výstup prohlédneme, zjistíme, že nám stačí pouze 17 členů Taylorova rozvoje pro dosažení strojové přesnosti. Oproti tomu i pro $n = 9999$ je výsledek funkce `e_z_limity()` přesný pouze na několik málo platných cifer. Nicméně tyto dvě metody nemůžeme zcela srovnávat, neboť jde v obou případech o zcela jiný postup a parametr `n` má v obou případech různý význam.

Fyzikální korespondenční seminář je organizován studenty MFF UK. Je zastřešen Oddělením pro vnější vztahy a propagaci MFF UK a podporován Ústavem teoretické fyziky MFF UK, jeho zaměstnanci a Jednotou českých matematiků a fyziků.

[31] Problém není velikost výstupu, ale to, že program musí volat systémovou funkci pro výpis z programu. Výpis tedy můžeme zrychlit tím, že si výstup nejdřív vytvoříme jako řetězec a ten pak vypíšeme jednou. V Pythonu existují alternativní způsoby čtení a výpisu dat, které to opravdu dělají a jsou tak o hodně rychlejší