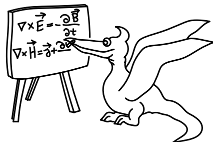


Seriál: Rostou nám diferenciální rovnice

V tomto díle seriálu pokročíme k numerickému tématu, které je ve fyzice všudypřítomné: řešení obyčejných diferenciálních rovnic (ODR). Ačkoli nejsou tak složité jako parciální diferenciální rovnice, vyžaduje jejich analytické řešení v mnoha případech pokročilé matematické metody. Numerika nám umožní se matematickým komplikacím vyhnout – pomocí diskretizace derivací získáme integrační schémata, pomocí nichž dokážeme vyřešit každou ODR se zvolenou přesností.

Simulační část seriálu nepokročí, ale naopak zůstane u tématu z minulého dílu. Ukážeme si, že jiné modely růstu povrchů vedou na kvalitativně odlišný vývoj hrubosti povrchu a také si předvedeme, jak použít růstové modely v mikrobiologii. Poté se vrátíme k náhodným procházkám a využijeme je k simulaci růstu fraktálního krystalu.

Obyčejné diferenciální rovnice

V tomto díle se naučíme základy numerického řešení obyčejných diferenciálních rovnic. Jde o poměrně atraktivní problematiku, ostatně mnoha lidem se při vyslovení pojmu „počítačové simulace“ vybaví právě řešení diferenciálních rovnic. My již víme, že numerická matematika je daleko rozsáhlejším oborem, to ale nijak nesnižuje význam a potřebu numerického řešení diferenciálních rovnic. Nicméně jde o zvlášť zákeřnou oblast numerické matematiky, narazit zde na nestabilní řešení není nic výjimečného a opravdu platí rčení: „Důvěřuj, ale prověřuj!“

Diferenciální rovnice jsou jednoduše řečeno takové rovnice, ve kterých vystupuje derivace. Na rozdíl od algebraických rovnic, jejichž řešením je číslo, řešením diferenciálních rovnic je funkce. Příkladem diferenciální rovnice je například pohybová rovnice svislého vrhu $\ddot{y} = -g$, kde $\ddot{y}$ značí druhou derivaci $y(t)$ vůči času. Řešením této rovnice pak je funkce $y(t) = y_0 + v_0 t - \frac{1}{2}gt^2$, kde y_0 a v_0 jsou tzv. integrační konstanty, tedy volné parametry řešení. V případě svislého vrhu pak mají jasný fyzikální význam počáteční polohy a rychlosti. Sami si pak můžete snadno ověřit, že druhá derivace dané funkce podle času je skutečně rovna $-g$. Pokud k zadání přidáme i tzv. počáteční podmínky, např. $y(0) = 1$ a $\dot{y}(0) = 0$, dokážeme jednoznačně určit i hodnotu integračních konstant. V našem případě pak řešení bude $y(t) = 1 - \frac{1}{2}gt^2$. Platí přitom, že počátečních podmínek potřebujeme tolik, jaký je řád diferenciální rovnice, tedy řád nejvyšší derivace v ní vystupující. V našem případě rovnice obsahuje druhou derivaci, rovnice je tedy druhého řádu a potřebujeme dvě nezávislé počáteční podmínky. Takto přesně analyticky vyřešit ovšem umíme pouze některé jednoduché rovnice, na zbytek musíme použít aproximace, rozvoje, nebo právě numerické metody.

Diferenciální rovnice dělíme na *obyčejné* (zkr. ODR), které obsahují neznámou funkci pouze jedné proměnné a derivace podle této proměnné, a *parciální* (zkr. PDR), obsahující neznámou funkci více proměnných a (parciální) derivace podle nich. Řešení parciálních diferenciálních rovnic je zpravidla složitější úloha, než řešení ODR, a to i při numerickém přístupu, nadále se tedy budeme zabývat pouze řešením ODR. Konkrétně budeme hledat řešení soustavy N rovnic, na které můžeme nahlédnout jako na vektorovou rovnici

$$\dot{\boldsymbol{u}}(t) = \boldsymbol{f}(\boldsymbol{u}(t), t)$$

s počáteční podmínkou $\boldsymbol{u}(0) = \boldsymbol{u}_0$. To, že jde o soustavu rovnic, se prakticky projeví pouze tak, že všechny kroky řešení neprovedeme jednou, ale N-krát, tedy pro každou z rovnic soustavy. Při popisu metod si tedy nemusíte zadání představovat jako soustavu, ale jen jako jednu rovnici. Tento přístup má dokonce jisté výhody, jak uvidíme při praktické implementaci metod. Ač to tak na první pohled nevypadá, do výše uvedeného tvaru lze převést naprostou většinu soustav obyčejných diferenciálních rovnic libovolného řádu[1], nijak se tedy v úloze neomezujeme. Jak tento převod prakticky provést si ukážeme na rovnici $\ddot{y} = -g$. Nejprve zavedeme substituci $\dot{y}(t) = v(t)$ a dosadíme do původní rovnice, čímž získáme $\dot{v} = -g$. Máme tedy soustavu dvou rovnic prvního řádu $\dot{y} = v$ a $\dot{v} = -g$ s neznámými $y(t)$ a $v(t)$. Nyní jsme připraveni k popisu samotných metod řešení, ještě předtím si ale pokusíme osvojit dovednost pro fyzika možná i důležitější, a to, jak formulovat problém ve formě diferenciální rovnice.

Derivace nějaké veličiny podle času vyjadřuje rychlost změny této veličiny. Jako příklad si můžeme vzít model radioaktivity, kdy z fyziky víme, že za jednotku času se rozpadne takové množství atomů, které je přímo úměrné počtu nerozpadlých atomů. Příslušná diferenciální rovnice tedy je

$$\frac{\mathrm{d}N(t)}{\mathrm{d}t} = -\lambda N(t)$$

s počáteční podmínkou $N(0) = N_0$, kde $N(t)$ je počet ještě nerozpadlých atomů v čase t a λ je rozpadová konstanta vyjadřující počet přeměn za jednotku času přepočtený na jeden atom. Mínus pak značí, že nerozpadlé atomy postupně ubývají. Řešením této rovnice je pak známá rovnice radioaktivní přeměny $N(t) = N_0 \exp(-\lambda t)$.

Dalším důležitým příkladem jsou pohybové rovnice vycházející z druhého Newtonova zákona, který je vlastně diferenciální rovnicí $\dot{\boldsymbol{p}} = \boldsymbol{F}$, slovně: „Časová změna hybnosti hmotného bodu je v každém okamžiku rovna výslednici sil na tento bod působících“. Pokud budeme uvažovat konstantní hmotnost, pak dostáváme méně obecnou verzi $\dot{\boldsymbol{p}} = \dot{(m\boldsymbol{v})} = m\dot{\boldsymbol{v}} = m\ddot{\boldsymbol{x}} = \boldsymbol{F}$, tedy $\ddot{\boldsymbol{x}} = \boldsymbol{F}/m$. Nyní stačí dosadit za sílu dle konkrétního fyzikálního problému a získáme soustavu pohybových rovnic daného hmotného bodu. Například pro šikmý vrh v homogenním gravitačním poli bude síla za použití kartézských souřadnic rovna $\boldsymbol{F} = (0, 0, -g)$. Musíme pak ještě samozřejmě zadat 6 počátečních podmínek, např. polohy a rychlosti na počátku a rovnice případně převést na soustavu rovnic prvního řádu, jak bylo ukázáno výše.

Eulerova metoda

První a nejjednodušší metodou, kterou si zde představíme, je explicitní Eulerova metoda. Vyjdeme ze soustavy $\dot{\boldsymbol{u}}(t) = \boldsymbol{f}(\boldsymbol{u}(t), t)$ a derivaci nahradíme dopřednou diferencí. Po úpravě pak dostaneme

$$\frac{\boldsymbol{u}(t+h) - \boldsymbol{u}(t)}{h} = \boldsymbol{f}(\boldsymbol{u}(t), t) + O(h)\,,$$

$$\boldsymbol{u}_{n+1} = \boldsymbol{u}_n + h\boldsymbol{f}(\boldsymbol{u}_n, t_n) + O(h^2)\,,$$

kde jsme označili $t_n \stackrel{\text{def}}{=} nh$ (bez újmy na obecnosti můžeme začít v čase $t_0 = 0$), $\boldsymbol{u}_n \stackrel{\text{def}}{=} \boldsymbol{u}(t_n)$ a $\boldsymbol{u}_{n+1} \stackrel{\text{def}}{=} \boldsymbol{u}(t_{n+1})$. Nyní již máme přímočarý návod k řešení diferenciální rovnice. Vezmeme počáteční podmínku $\boldsymbol{u}_0$ a dosazením $\boldsymbol{u}_n = \boldsymbol{u}_0$ a $t_n = 0$ do rovnice výše vypočteme $\boldsymbol{u}_1$, to opět dosadíme (nyní s časem $t_1 = h$) a iterujeme, dokud nezískáme hodnoty až do kýženého času. Metodu lze implementovat například takto:

[1] Snad s výjimkou některých obzvlášť ošklivých rovnic, které jsou nelineární v nejvyšším řádu derivace.

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

def euler_step(f,y,t,h):
        return y+h*f(y,t)

def ODE_solve(f,y0,t0,tf,steps):
        times = np.linspace(t0,tf,steps)
        h = (tf-t0)/steps
        result = []
        y = y0
        for t in times:
                result.append(y)
                y = euler_step(f,y,t,h)
        return times,np.array(result)

def vrh(y,t):
        return np.array((y[2],y[3],0.0,-9.81))

X0,Y0 = 0.0,0.0
VX0,VY0 = 5.0,10.0
t0,tf,steps = 0.0,10.0,1000
y0 = np.array((X0,Y0,VX0,VY0))

times, res = ODE_solve(vrh, y0, t0, tf, steps)
plt.plot(times, res[:,1]) #vykresli Y(t)
plt.show()
```

Všimněme si, že veškerá fyzika je obsažena v definici funkce $f(u,t)$ (v našem případě za ni dosazujeme funkci `vrh()`) a v počátečních podmínkách. Toto oddělení fyziky a numerického řešiče (integrátoru) umožňuje flexibilně používat stejný integrátor pro různé problémy a naopak.

Eulerova metoda funguje na principu lineární extrapolace, kdy v bodě (t_n, u_n) sestrojíme tečnu k řešení u (její směrnice je vyjádřena hodnotou funkce $f(u_n, t_n)$) a posuneme se podél této tečny o časový krok h do bodu (t_{n+1}, u_{n+1}). Je jasné, že jde pouze o hrubou aproximaci skutečného řešení, pro dostatečnou přesnost tedy musíme volit poměrně malý krok h.

Rungovy-Kuttovy metody

Přesnost dokážeme vylepšit například s použitím Rungových-Kuttových metod vyššího řádu. Předpis R-K metody s-tého řádu má tvar

$$u_{n+1} = u_n + h \sum_{i=1}^{s} b_i k_i \,, \text{ kde}$$

$$k_i = f\left(u_n + h \sum_{j=1}^{s} a_{ij} k_j, t_n + c_i h\right) ,$$

přičemž a_{ij}, b_i a c_i jsou číselné koeficienty příslušející dané metodě. Často jsou v literatuře zapisovány ve formě tzv. *Butcherovy tabulky.* Všimněme si, že pro $j \geq i$ potřebujeme znát hodnoty k_j, které při postupném vyhodnocování ještě neznáme. Tyto metody se nazývají *implicitní* a mají určité nemalé výhody, které si zmíníme později, ty jsou však vykoupeny složitějším způsobem vyhodnocování. Tento problém ale zmizí, pokud pro $j \geq i$ je $a_{ij} = 0$. Pak můžeme bez problémů vyčíslit k_1, poté k_2, ..., k_s a nakonec u_{n+1}. Takové metody se nazývají *explicitní.*

Zkusme nyní dosadit $s = 1$, $a_{11} = 0$, $b_1 = 1$ a $c_1 = 0$. Pak dostaneme nám již známý předpis explicitní Eulerovy metody $u_{n+1} = u_n + hf(u_n, t_n)$. Explicitní Eulerova metoda je tedy Rungovou-Kuttovou metodou 1. řádu.

Pokusme se nyní ilustrovat, jak Rungovy-Kuttovy metody fungují, a to na příkladu R-K metody 4. řádu, která je jednou z nejpoužívanějších. Její předpis je

$$\begin{aligned} k_1 &= f(u_n, t_n)\,, \\ k_2 &= f\left(u_n + \frac{h}{2}k_1, t_n + \frac{h}{2}\right)\,, \\ k_3 &= f\left(u_n + \frac{h}{2}k_2, t_n + \frac{h}{2}\right)\,, \\ k_4 &= f(u_n + hk_3, t_n + h)\,, \\ u_{n+1} &= u_n + \frac{1}{6}hk_1 + \frac{1}{3}hk_2 + \frac{1}{3}hk_3 + \frac{1}{6}hk_4\,. \end{aligned}$$

Nejprve vypočteme směrnici k_1 v bodě (t_n, u_n), podobně, jako v případě Eulerovy metody. Dále ale vypočteme další směrnice k_i v mezibodech určených předchozími směrnicemi. Nakonec všechny směrnice zprůměrujeme s různými vahami a posuneme se podél této výsledné směrnice o krok h z počátečního bodu kroku (t_n, u_n). Lze intuitivně předpokládat, že toto odhadování v mezibodech a následné průměrování je přesnější, než prostý dlouhý skok jako v Eulerově metodě, metematický důkaz zde ale neuvedeme. Dále by se mohlo zdát, že by bylo jednodušší a možná i přesnější použít Eulerovu metodu s třeba čtyřikrát kratším krokem. Vtip je v tom, že R-K metoda s-tého řádu má globální zaokrouhlovací chybu (chybu nasčítanou přes všechny kroky) řádu $O(h^s)$. Pro velmi velký krok tedy bude skutečně Eulerova metoda lepší,[2] ale pro menší krok (prakticky pro každý rozumně velký krok) budou R-K metody vyššího řádu přesnější. V praxi se pak často používá zmíněná R-K metoda 4. řádu, která představuje rozumný kompromis mezi přesností (zlepšující se s řádem) a stabilitou a počtem vyčíslení funkce f v jednom kroku (což jsou s řádem se zhoršující vlastnosti). Dále je používána například Dormandova-Princeova metoda, což je jedna z adaptivních R-K metod, tedy metod, které dokáží odhadnout chybu v daném kroku a na jejím základě adaptivně upravovat délku kroku v průběhu výpočtu.

Tab. 1: Butcherova tabulka

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \cdots & a_{1s} \\ c_2 & a_{21} & a_{22} & \cdots & a_{2s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & a_{s2} & \cdots & a_{ss} \\ \hline & b_1 & b_2 & \cdots & b_s \end{array}$$

Lineární vícekrokové metody

Dalším typem pokročilejších metod jsou lineární vícekrokové metody. Metody, které jsme doposud poznali, byly jednokrokové, protože pro výpočet stavu u_{n+1} stačilo znát pouze stav u_n,

[2] V takovém případě by byla lepší už z toho důvodu, že by R-K metody vyššího řádu byly nestabilní, tj. nevedly by ke správnému řešení.

starší stavy jsme znát nepotřebovali. Myšlenka vícekrokových metod pak spočívá v tom, že tyto starší stavy známe a s jejich pomocí lépe odhadneme vývoj do následujícího kroku než pouze se znalostí stavu a jeho derivace v jednom bodě. Předpis obecné lineární s-krokové metody je

$$\sum_{j=0}^{s} \alpha_j u_{n+j} = h \sum_{j=0}^{s} \beta_j f(u_{n+j}, t_{n+j}) \,,$$

kde α_j a β_j jsou koeficienty a platí $\alpha_s = 1$, $\alpha_0 \neq 0$ a $\beta_0 \neq 0$ (pomocí $u_n, \ldots, u_{n+s-1}$ a $f(u_n, t_n)$, $\ldots$, $f(u_{n+s}, t_{n+s})$ tedy chceme spočíst u_{n+s}). Pokud navíc platí $\beta_s = 0$, je metoda explicitní, jinak je implicitní.

Praktickým problémem při implementaci vícekrokových metod je jejich nastartování. Dokud jsme totiž neudělali $s-1$ kroků, nemůžeme metodu použít pro nedostatek známých u_i. Prvním řešením je, že máme k dispozici nadbytek počátečních podmínek, ideálně přímo ve formě potřebných u_i. To se stává zřídkakdy, protože to většinou znamená znalost řešení ODR, kterou chceme řešit. Častějším způsobem je použití nějaké jednokrokové metody pro prvních pár kroků. Dalším, hůře řešitelným problémem je, že vícekrokové metody předpokládají stálý krok h po celou dobu výpočtu, nelze tedy použít adaptivní změnu kroku dle aktuálního odhadu chyby.

Ne všechny metody splňující toto schéma jsou automaticky použitelné. Samozřejmým požadavkem je, aby metoda dostatečně přesně aproximovala danou ODR, musí tedy mít malou chybu metody (chyba musí být dostatečného řádu v h). Ovšem ani taková metoda nemusí automaticky fungovat dobře, protože metoda může neúměrně zvětšovat zaokrouhlovací chybu danou diskretizací problému, a to až tak, že řešení metody nebude vůbec kopírovat skutečné řešení. Říkáme pak, že je metoda *nestabilní*. Vyšetřování stability metod sahá nad rámec tohoto textu, spokojme se s tvrzením, že často používanými zástupci lineárních vícekrokových metod jsou explicitní Adamsovy-Bashforthovy metody a implicitní Adamsovy-Moultonovy metody, jejichž předpisy a vlastnosti (např. řád) najdete v literatuře.

Problémy s problémy se silným tlumením

Uvažujme diferenciální rovnici ve tvaru

$$y' = -cy \,,$$

kde $c > 0$ je reálná konstanta. Analytické řešení této rovnice je

$$y(x) = A\mathrm{e}^{-cx} \,,$$

kde A je integrační konstanta, kterou určíme z počátečních podmínek. Vidíme, že funkce $y(x)$ rychle klesá k nule. Takovému problému se říká problém se silným tlumením[3], anglicky *stiff equation*. Pokusme se jej nyní vyřešit pomocí explicitní Eulerovy metody. Přímým dosazením za $f(u,t)$ dostaneme

$$u_{n+1} = (1 - ch) u_n \,.$$

Pokud budeme volit $h > 2/c$, pak posloupnost $|u_n| \to +\infty$ pro $n \to +\infty$ pro libovolné počáteční podmínky, což rozhodně nedává správné řešení, metoda je *nestabilní*. Pokusme se nyní ten samý problém vyřešit s pomocí implicitní Eulerovy metody s předpisem $u_{n+1} = u_n + hf(u_{n+1}, t_{n+1})$,

[3] Název pochází z tlumení v harmonickém oscilátoru, které se projevuje právě takto.

kde ale můžeme vyjádřit u_{n+1} explicitně díky tomu, že známe vztah pro $f()$. Po dosazení dostáváme

$$u_{n+1} = \frac{u_n}{1 + ch}\,.$$

V tomto případě pro libovolně velký krok h posloupnost u_n klesá k nule. Zdá se tedy (a prakticky to tak je), že použití implicitních metod by mohlo být univerzálním lékem na problém silného tlumení.

Ukázali jsme tedy, že za určitých (ne zcela vzácných) okolností jsou implicitní metody nedocenitelné. Zbývá vyřešit problém, jak je řešit, když se zdá, že potřebujeme znát u_{n+1} dříve, než jej vypočítáme (a nemůžeme využít trik s dosazením za $f()$). První, co nás napadne, je použít iterativní metody, kdy nastřelíme nějaký odhad u^0_{n+1} a iterujeme předpisem implicitní metody, dokud u^i_{n+1} není dostatečně blízko u_{n+1} (horní index je jen index iterace, ne mocnina). Hodilo by se, aby už prvotní odhad byl dostatečně blízko, abychom dokonvergovali rychle, nabízí se tedy pro prvotní odhad použít explicitní metodu. Tím se dostáváme k metodám *prediktor-korektor*, které fungují přesně na tomto principu. Nejprve použijeme explicitní metodu (prediktor) pro určení hrubého nástřelu u_{n+1}, ten poté zpřesníme použitím jedné, nebo více iterací implicitní metody (korektoru). Díky použití implicitní metody máme prakticky zajištěnu stabilitu, nezdržujeme se ale příliš mnoha iteracemi (většinou stačí jedna), metoda je tedy rychlá. Při jejich použití je třeba dát pozor na správné namixování prediktoru a korektoru, mimo jiné na řád metod. Doporučujeme tedy používat nějaké standardní kombinace, či ještě lépe použít numerickou knihovnu, která má metody implementovány. V případě kombinace Adamsovy-Bashforthovy a Adamsovy-Moultonovy metody je takovou standardní kombinací, například

$$u_{n+2} = u_{n+1} + \frac{1}{2}h(3f(u_{n+1}, t_{n+1}) - f(u_n, t_n))\,,$$
$$u_{n+2} = u_{n+1} + \frac{h}{12}(5f(u_{n+2}, t_{n+2}) + 8f(u_{n+1}, t_{n+1}) - f(u_n, t_n))\,,$$

čímž dostaneme prediktor-korektor 3. řádu.

Zachování energie a Verletův algoritmus

Jak jsme zmínili v úvodu (a po diskusi stability je nejspíš jasné), je třeba v průběhu výpočtu kontrolovat, jestli nám příliš neroste chyba. U fyzikálních dějů se nám často nabízejí různé zachovávající se veličiny, například u oběhu planety kolem Slunce se (mimo jiné) zachovává celková energie a moment hybnosti. Pokud si budeme tyto veličiny v průběhu výpočtu vypisovat, měly by být konstantní, samozřejmě až na nějakou numerickou chybu, která ale nesmí být velká.

Pokud ale řešíme diferenciální rovnici typu $u'' = f(u, t)$, což je mj. i častý případ mechanických pohybů, pak můžeme s výhodou použít metodu, která automaticky ze své podstaty zachovává energii. Napišme si Taylorův rozvoj $u(t + h)$ a $u(t - h)$ a sečtěme je.

$$u(t + h) = u(t) + hu'(t) + \frac{h^2}{2}u''(t) + \frac{h^3}{6}u'''(t) + O(h^4)$$
$$u(t - h) = u(t) - hu'(t) + \frac{h^2}{2}u''(t) - \frac{h^3}{6}u'''(t) + O(h^4)$$
$$u(t + h) = 2u(t) - u(t - h) + h^2u''(t) + O(h^4)$$

Poslední rovnice nám již dává předpis pro Verletovu metodu

$$u_{n+1} = 2u_n - u_{n-1} + h^2 f(u_n, t_n) \,.$$

Důležitou vlastností Verletovy metody je, že je symetrická vůči směru toku času, neboli *časově reverzibilní.* Skutečně, pokud zaměníme h za $-h$, dostaneme

$$u_{n-1} = 2u_n - u_{n+1} + h^2 f(u_n, t_n) \,,$$

což po úpravě dává vztah výše. Díky této vlastnosti pak z důležité matematické věty, které se říká *teorém Noetherové*, plyne, že Verletův vztah zachovává energii, ovšem samozřejmě pouze za předpokladu, že energii zachovává i simulovaný děj, reprezentovaný funkcí f.

Nejspíš jste se již s teorémem Noetherové setkali v populární literatuře. Pak si nejspíš vzpomínáte na tvrzení, že každé symetrii odpovídá nějaká zachovávající se veličina. Zde je tou symetrií invariance vůči směru toku času, odpovídající se zachovávající veličinou je pak energie.

Růst povrchů II – balistická depozice a Edenův model

Než začneme vysvětlovat nové pojmy, zopakujme si nejprve ve stručnosti poznatky z minulého dílu seriálu. Říkali jsme si, že růst povrchu lze modelovat pomocí stochastického vícestavového buněčného automatu. Automat je stochastický, protože buňku, která v daném kroce bude růst, vybíráme náhodně. Buňkou je zde myšlen jeden sloupeček molekul na mřížce. Že je automat vícestavový, znamená, že výška h_i sloupečku, kterou ztotožňujeme se stavem buňky, může nabývat během vývoje více než dvou různých hodnot ($h_i \in \mathbb{N}$). Hrubost povrchu v čase t (po t krocích) na 1D mřížce velikosti L definujeme jako

$$W(t, L) = \sqrt{\frac{1}{L}\sum_i h_i^2 - \left(\frac{1}{L}\sum_i h_i\right)^2} \,. \tag{1}$$

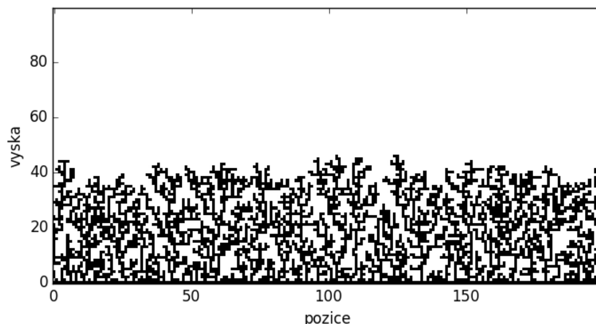
Model náhodné depozice náhodně vybere buňku i a zvětší hodnotu jejího stavu o jedna. Balistická depozice má mírně složitější pravidlo,

$$h_i(t+1) = \max\left(h_{i-1}(t),\, h_i(t) + 1,\, h_{i+1}(t)\right) \,. \tag{2}$$

Růst povrchu modelovaný metodou balistické depozice si můžeme představit tak, že částice sestupují z nekonečna na povrch a zastaví se tehdy, když je nějaká částice ihned pod nimi, ale také tehdy, nachází-li se nějaká částice v sousední buňce nalevo nebo napravo. V materiálu tak vznikají dutiny, což se děje i ve skutečných experimentech, například při růstu povrchu pomocí naprašování (epitaxe). Porózní materiál má pak samozřejmě jiné vlastnosti. Ukázka materiálu vzniklého balistickou depozicí je na obrázku 1.

Balistická depozice[4] se však od náhodné depozice neliší pouze porozitou. Podívejme se, jak se vyvíjí hrubost povrchu. Na obrázku 2 je vykreslen vývoj hrubosti vygenerovaný pomocí kódu níže. Vidíme, že na počátku hrubost poměrně rychle narůstá, ale růst zpomaluje a po cca 10 000 krocích se zastaví – došlo k saturaci (nasycení). Tento efekt jsme nepozorovali u náhodné depozice (viz řešení úlohy ke 4. dílu seriálu), kdy hrubost neustále rostla jako mocnina $t^{1/2}$. Obecně můžeme rozdělit vývoj hrubosti na tři časové úseky na základě času přechodu t_x:

[4] Pokud budete hledat o balistické depozici něco na internetu, možná narazíte na jinou definici. Zde jsme definovali NN (nearest neighbour) depozici.

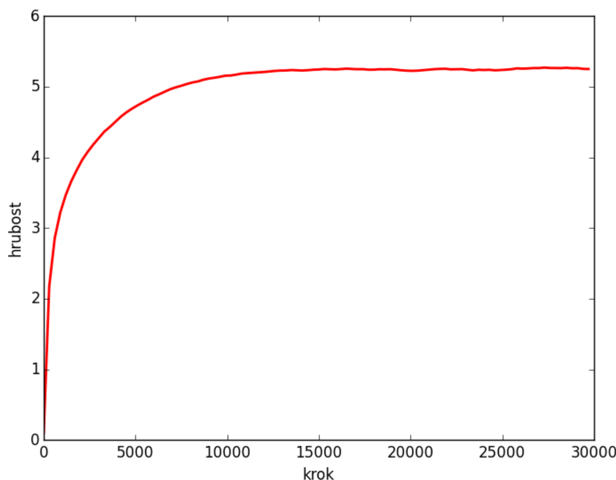


Obr. 1: Struktura povrchu získaného na základě modelu balistické depozice. Všimněte si vysoké porozity.

1. $t \ll t_\mathrm{x} \rightarrow W(t,L) \sim t^\beta$;
2. $t \approx t_\mathrm{x} \rightarrow$ přechodová oblast, růst se zpomaluje;
3. $t \gg t_\mathrm{x} \rightarrow W(t,L) \sim L^\alpha$.

Parametry α a β se nazývají *kritické škálovací parametry*. Dále se zavádí *dynamický škálovací parametr* $z = \alpha/\beta$, který popisuje chování času přechodu jako $t_\mathrm{x} \sim L^z$. Nalezení parametrů α a β pro model balistické depozice bude vaším úkolem v seriálové úloze.

```
# BALISTICKA DEPOZICE
# nacteme grafickou knihovnu a numericku knihovnu
import matplotlib as mpl
from matplotlib import pyplot as plt
import numpy as np

# definujeme rozmer mrizky a pocet kroku a interval pro vypocet hrubosti
pocet = 100
roz = 100
krok = 30000
inter = 30
# vytvorime pole naplnene nulami pro stavy a pro hrubost
stav = np.zeros(roz,dtype=float)
hrub = np.zeros(krok/inter,dtype=float)
sumhrub = hrub

for j in range(pocet):
        # v kazdem kroku nahodne vybereme bunku a zvysime jeji hodnotu o 1
        hrub = np.zeros(krok/inter,dtype=float)
        stav = np.zeros(roz,dtype=float)
        for i in range(krok):
```



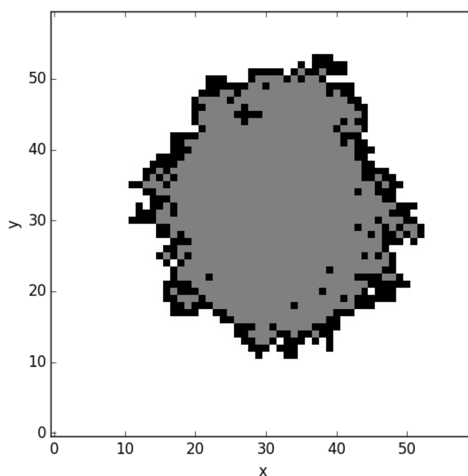
Obr. 2: Příklad vývoje hrubosti povrchu pro simulaci růstu metodou balistické depozice. Průměrováno přes 1000 běhů.

```
                nahod = np.random.randint(roz)
                stav[nahod] = max(stav[nahod]+1.,stav[(nahod-1) % roz],stav[(nahod+1) %
                     roz])
                if i % inter == 0:
                        hrub[i/inter] = np.sqrt(1./roz*np.sum(stav**2) - (1./roz*np.sum(
                            stav))**2)
        sumhrub += hrub

sumhrub *= 1./pocet

# nakreslime graf vyvoje hrubosti
# funkce linspace rovnomerne rozdeli pocet kroku na intervaly
body = np.linspace(0,krok-inter,krok/inter)
hrubplot = plt.plot(body,sumhrub,'r',linewidth=2,label='hrubost')
plt.xlabel('krok')
plt.ylabel('hrubost')
plt.show()
```

Druhým růstovým modelem, o kterém se zde blíže zmíníme, je Edenův model. Jeho princip je jednoduchý – částice se může přilepit na libovolné místo na povrchu (tj. seshora i zboku sloupečku). Ale protože zde bude naším cílem simulovat růst bakteriálních kolonií, jak jsme již v úvodu předeslali, nemůžeme použít 1D mřížku. Model si proto představíme jako binární automat (pouze stavy 0 a 1) na 2D mřížce a povolíme růst ze všech směrů. Během simulace je tedy potřeba pamatovat si celý povrch, a to včetně dutin uvnitř (bakterie se mohou množit kdekoli, kde mají životní prostor). Správným přístupem je ukládat si souřadnice buněk na povrchu, ale programátorsky jednodušší je vytvořit pole $N \times N$, kde N je zvolený rozměr mřížky, a ukládat hodnotu každé buňky. Výpočet je pak samozřejmě pomalý a musíme se omezit na malé mřížky. Příklad povrchu vytvořeného pomocí Edenova modelu vidíte na obrázku 3.

K vygenerování byl použit kód níže.

Edenův model přiřazuje růstu uvnitř dutin v materiálu stejnou pravděpodobnost jako růstu na volném povrchu. Porozita tak jednak zvyšuje celkovou délku povrchu, kterou musíme udržovat v paměti, a také nemusí odpovídat skutečnému chování fyzikálního systému, který studujeme. Proto se často zavádějí úpravy modelu, které zvyšují pravděpodobnost zaplňování dutiny úměrně její vzdálenosti od povrchu. Tato úprava samozřejmě ovlivní hodnoty škálovacích koeficientů ve vztazích pro hrubost, ale ne příliš výrazně.



Obr. 3: Výsledek simulace růstu podle Edenova modelu po 2000 krocích. Šedou barvou jsou označeny ty buňky, které byly náhodně vybrány, ale už neměly neobsazené sousedy (bylo by možné vykreslit je také černou barvou, jde jen o ukázku toho, jak funguje použitý algoritmus). Porozita je v porovnání s balistickou depozicí minimální.

```
# EDENUV MODEL
# nacteme grafickou knihovnu a numericku knihovnu
import matplotlib as mpl
from matplotlib import pyplot as plt
import numpy as np

# definujeme rozmer mrizky a pocet kroku
roz = 200
krok = 20000
# vytvorime pole nul, do ktereho budeme ukladat stavy bunek
# zavorka (kroky, rozmer) udava velikost 2D pole
pole = np.zeros((roz, roz),dtype=np.int)
# inicializujeme
pole[100,100] = 1

for i in range(krok):
```

```
        # funkce where vrati indexy prvku, ktere maji hodnotu jedna,
        # ve forme seznamu obsahujiciho dve pole, kde kazde z techto poli
        # obsahuje prislusne x,y souradnice
        indexy = np.where(pole == 1)
        # nahodne vybereme index
        rand = np.random.randint(len(indexy[0]))
        ind = [(indexy[0])[rand],(indexy[1])[rand]]
        # prozkoumame okoli odpovidajiciho prvku
        soused1 = pole[(ind[0]+1) % roz,ind[1]]
        soused2 = pole[ind[0],(ind[1]+1) % roz]
        soused3 = pole[(ind[0]-1) % roz,ind[1]]
        soused4 = pole[ind[0],(ind[1]-1) % roz]
        sousedi = np.array([soused1,soused2,soused3,soused4])
        sous_indexy = np.where(sousedi == 0)
        # pokud jsou sousedi zaplneni, nastavime pole[ind] na 2,
        # abychom ho jiz priste neprohledavali
        if len(sous_indexy[0]) == 0:
                pole[ind[0],ind[1]] = 2
        # nahodne vybereme prazdnou sousedni bunku a zaplnime ji
        else:
                sous_ind = (sous_indexy[0])[np.random.randint(len(sous_indexy[0]))]
                if sous_ind == 0:
                        pole[(ind[0]+1) % roz,ind[1]] = 1
                elif sous_ind == 1:
                        pole[ind[0],(ind[1]+1) % roz] = 1
                elif sous_ind == 2:
                        pole[(ind[0]-1) % roz,ind[1]] = 1
                elif sous_ind == 3:
                        pole[ind[0],(ind[1]-1) % roz] = 1

# definujeme barevne schema pomoci vyctu, v nasem pripade cernobile
barvy = mpl.colors.ListedColormap(['white','black','grey'])
# definujeme prechod mezi barvami kdekoli mezi 0 a 1
hranice=[0,0.5,1.5,2]
# pouzijeme barvy a hranice k normovani barevne mapy
norma = mpl.colors.BoundaryNorm(hranice, barvy.N)
# vykreslime barevnou mapu a zobrazime ji
mapa = plt.imshow(pole,cmap=barvy,interpolation='None',norm=norma,origin='lower')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

Kvůli radiální geometrii nelze přímo aplikovat vzorec pro hrubost (1). Proto se nebudeme hrubostí zabývat, ale zavedeme do modelu malou změnu: s pravděpodobností p_1 změníme stav buňky na rozhraní z 0 (prázdná) na 1 (obsazená) a s pravděpodobností p_2 naopak změníme stav buňky na rozhraní z 1 na 0. Tento model, někdy nazývaný Williams-Bjerknesův, je vhodný pro simulaci růstu nádorových buněk nebo bakterií v Petriho misce. V seriálové úloze bude vaším úkolem zkoumat vývoj nádorových buněk pomocí právě popsaného modelu v závislosti na p_1, p_2.

Pro úplnost uveďme, že modelovat růst povrchů lze i jinými metodami než pomocí buněčných automatů. Ke každému zmíněnému růstovému modelu existuje ekvivalentní diferenciální rovnice, kterou lze v některých případech řešit analyticky a získat tak stejné škálovací parametry jako numerickým modelováním. Jedná se však o stochastické diferenciální rovnice, tj. rovnice obsahující člen, který reprezentuje šum. K řešení takových rovnic je potřeba zavést speciální integrální počet, tzv. Itův stochastický kalkulus.

DLA – difúzí limitovaná agregace

Zatím zmíněné růstové modely dokázaly poměrně dobře simulovat laboratorní tvorbu materiálů nebo růst bakteriálních kolonií. Pokud bychom ale chtěli modelovat růst krystalu vznikajícího v roztocích, např. pomocí elektrolýzy nebo sedimentací, bude lepší volbou model DLA (Diffusion limited aggregation).

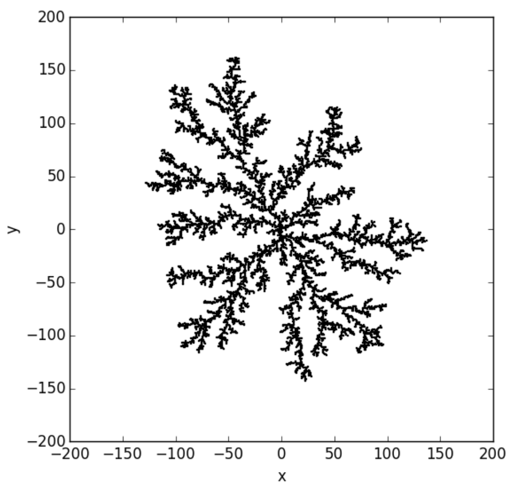
Fyzikální představa procesu je následující: částice z nekonečna se brownovsky pohybují prostorem, dokud nedojde k jejich kontaktu s krystalizačním jádrem, ke kterému se přilepí. Hustota částic je malá, vzájemně spolu nekolidují (brownovský pohyb zajišťují molekuly v pozadí, které na jádru neulpívají). V počítačové praxi nemůžeme poslat částici z nekonečna, vytvoříme ji proto v určité vzdálenosti od krystalu a necháme ji pohybovat se tak dlouho, dokud nedojde ke kolizi s krystalem, nebo se nedostane za určitou radiální hranici – potom částici přestaneme sledovat a vytvoříme novou.

Níže je příklad kódu (výsledek simulace je na obrázku 4), kde máme jednu částici v počátku hexagonální mřížky a nové částice generujeme ve vzdálenosti, která je rovna radiální vzdálenosti od počátku nejvzdálenější částice krystalu (plus jedna). Částici přestaneme sledovat, pokud se vzdálí na dvojnásobek počáteční vzdálenosti. Snížíme-li maximální povolenou vzdálenost, běh kódu se výrazně zrychlí, ale krystal potom začne vykazovat preferovaný směr růstu. Kód je poměrně komplikovaný kvůli implementaci šestiúhelníkové mřížky. V seriálové úloze budete simulovat růst na čtvercové mřížce, tím se kód podstatně zjednoduší.

```
# DIFFUSION LIMITED AGGREGATION
# nacteme grafickou knihovnu a numericku knihovnu
import matplotlib as mpl
from matplotlib import pyplot as plt
import numpy as np

# definujeme rozmer mrizky a maximalni pocet castic
roz = 20
mass_max = 100
# pole pro spiralni 1D index, bezpecne vetsi nez je potreba
roz_1d = 6*roz**2
# pole, do kterzch budeme ukladat prepoctene kartezske souradnice
# definujeme vektory pomahajici pri prepoctu z hexa na ctvercovou mrizku
xs = np.zeros(mass_max,dtype=float)
ys = np.zeros(mass_max,dtype=float)
xhelp = [np.sqrt(3.)/2.,0.,-np.sqrt(3.)/2.,-np.sqrt(3.)/2.,0.,np.sqrt(3.)/2.]
yhelp = [-0.5,-1.,-0.5,0.5,1.,0.5]
# pole nul, do ktereho budeme ukladat stavy bunek
pole = np.zeros(roz_1d,dtype=np.int)
# inicializujeme bunky, hmotnost a nejvetsi vzdalenost od stredu
pole[0] = 1
mass = 1
rmax = 0

while mass < mass_max:
        # pocatecni vzdalenost castice
        r = rmax+1
        # pricist k hranici v radialnim smeru, za kterou castice nesmi utect
        rkill = rmax
        # vyber nahodny prvek z sestiuhelniku o rozmeru r
        i = int(6*r*np.random.random() + 1)
        nn = 0
        # pohyb difundujici castice
        while nn == 0:
                i_full = i+3*r*(r-1) # pozice v poli 'pole'
                # nejsme-li ve vrcholu sestiuhelniku
```

```
        if (i % r) != 0:
                # nalezneme sousedy
                if i == 1:
                 ileft = 6*r
                 idownleft = 3*r*(r-1)
                else:
                 ileft = i-1
                 idownleft = (i*(r-1))//r + 3*(r-1)*(r-2)
                # dvojite lomitko vraci floor
                pole_nn = [i+1+3*r*(r-1),ileft+3*r*(r-1),idownleft,
                 (i*(r-1))//r+3*(r-1)*(r-2)+1,
                 (i*(r+1))//r+3*r*(r+1),
                 (i*(r+1))//r+3*r*(r+1)+1]
                obsazeno = np.where(pole[pole_nn] == 1)
                nn = len(obsazeno[0])
                # je-li nejaky soused obsazen, prilep se
                if nn > 0:
                 pole[i_full] = 1
                 rmax = max([rmax,r])
                 mass += 1
                # jinak difunduj dal
                else:
                 step = int(np.random.random()*6)
                 # radialni sestup
                 if (step == 2 or step == 3):
                 r -= 1
                 # radialni vzestup
                 elif (step == 4 or step == 5):
                 r += 1
                 i = pole_nn[step] - 3*r*(r-1)
        # pokud jsme ve vrcholu sestiuhelniku
        else:
                if i == 6*r:
                 iright = 1
                 iupright = 1
                else:
                 iright = i+1
                 iupright = i*(r+1)/r+1
                if i == 1:
                 ileft = 6
                else:
                 ileft = i-1
                pole_nn = [iright+3*r*(r-1),ileft+3*r*(r-1),
                 i*(r-1)//r+3*(r-1)*(r-2),i*(r+1)//r+3*r*(r+1),
                 iupright+3*r*(r+1l),i*(r+1)//r-1+3*r*(r+1)]
                obsazeno = np.where(pole[pole_nn] == 1)
                nn = len(obsazeno[0])
                if nn > 0:
                 pole[i_full] = 1
                 rmax = max([rmax,r])
                 mass += 1
                else:
                 step = int(np.random.random()*6)
                 if step == 2:
                 r -= 1
                 elif ((step == 3 or step == 4) or step == 5):
                 r += 1
                 i = pole_nn[step] - 3*r*(r-1)
        # pokud castice utekla, zacneme znovu
        if r > (rmax+rkill):
                r = rmax+1
                i = int(6*r*np.random.random()+1)
                continue
```

```
        x0 = r*np.sin(2.*np.pi/(6.*r)*(i - (i % r)))
        y0 = r*np.cos(2.*np.pi/(6.*r)*(i - (i % r)))
        x = x0 + (i % r)*xhelp[min(int(i//r),5)]
        y = y0 + (i % r)*yhelp[min(int(i//r),5)]
        xs[mass-1] = x
        ys[mass-1] = y

# vykreslime castice pomoci sestiuhelnikovych symbolu
dlaplot = plt.scatter(xs,ys,c='k',s=30000/(roz**2),marker=(6,0,30))
plt.xlim(-roz, roz)
plt.ylim(-roz, roz)
plt.axes().set_aspect('equal')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



Obr. 4: Výsledek simulace růstu podle DLA modelu na hexagonální mřížce. Počet částic v krystalu je 8000. Všimněte si, že i přes poměrně benevolentní limit na maximální vzdálenost difundující částice je na krystalu vidět preferovaný směr růstu.

Na vzniklý krystal (shluk částic) můžeme pohlížet jako na fraktál. Nebudeme zde zabíhat do hluboké teorie, fraktál si prostě představíme jako objekt, který při bližším pohledu (myšleno ve větším měřítku) vykazuje stále stejnou strukturu – je soběpodobný. Příkladem z přírody je například list kapradiny, který se skládá z mnoha menších listů, a ty zase z menších lístků. V matematické abstrakci lze na objekt „zoomovat“ donekonečna, vždy budeme nacházet stále stejnou geometrickou strukturu. Podobně na obrázku 4 najdeme na každé větvi rostoucí z počátku další větev, ze které rostou další větve atd.

Pro fraktály je typické, že jejich dimenze je větší než u „obyčejných“ geometrických objektů. Například kružnice má dimenzi 1, ale Kochova vločka (obr. 5), což je vlastně jenom jinak

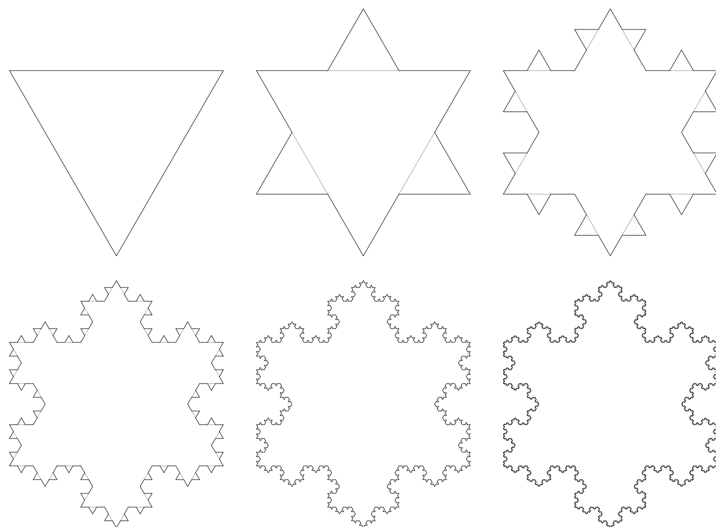
zakřivená čára, má dimenzi přibližně 1,26. Na Kochově vločce si vysvětlíme definici dimenze.[5] Nechť má strana počátečního trojúhelníku délku $\varepsilon = 1$. Počet úseček v jedné straně je samozřejmě $N = 1$. V dalším kroku již jedna strana obsahuje $N = 4$ úsečky délky $\varepsilon = 1/3$, v dalším kroku $N = 16$ úseček délky $\varepsilon = 1/9$ atd. Fraktální dimenze je pak definována jako

$$D = \lim_{\varepsilon \to 0} \frac{\log N(\varepsilon)}{\log \frac{1}{\varepsilon}}\,, \tag{3}$$

přičemž pro vločku můžeme psát $\varepsilon(n) = 1/3^n$, $N(n) = 4^n$, a tedy

$$D_{\mathrm{Koch}} = \lim_{n\to\infty} \frac{\log 4^n}{\log 3^n} = \frac{\log 4}{\log 3} \doteq 1{,}26\,.$$

Všimněte si, že se ve výpočtu mocniny n zkrátily. To je dáno faktem, že princip výstavby Kochovy vločky spočívá v aplikaci zcela identické geometrické deformace na jednotlivé hrany vločky v každém kroku. Limitní proces tedy nemá v případě matematicky exaktní vločky smysl (tj. dimenzi by šlo definovat jinak, např. jako $\log N(\varepsilon)/\log(1/\varepsilon)$ bez limity), ale má smysl v případě fraktálu vytvořeného DLA. Při „zoomování“ zde totiž nepozorujeme zcela identické obrazce, ale pouze velmi podobné, protože výstavba krystalu probíhá náhodně. Navíc krystal není nekonečně velký, resp. nemá nekonečně detailní strukturu, a také v důsledku nutného omezení délky náhodné procházky vstupují do procesu další odchylky. Při výpočtu dimenze budeme nyní místo strany trojúhelníku měřit lineární rozměr krystalu (radiální vzdálenost nejvzdálenější částice od počátku), místo počtu úseček máme počet částic. Přítomnost limity nám říká, že pro co nejpřesnější výsledek chceme nechat krystal růst co nejdéle. Pro krystal na obrázku 4 vychází dimenze $D \doteq 1{,}75$.



Obr. 5: Prvních šest iterací Kochovy vločky. Tenké čáry jsou vždy z předchozího kroku.

[5] Existuje mnoho způsobů, jak definovat fraktální dimenzi, zde jsme vybrali jeden z nich.

Nakonec ještě dodejme, že v případě materiálu získaného na základě Edenova modelu můžeme pohlížet na jeho povrch jako na fraktál, tělo takového materiálu ale není fraktální.

Fyzikální korespondenční seminář je organizován studenty MFF UK. Je zastřešen Oddělením propagace a mediální komunikace MFF UK a podporován Ústavem teoretické fyziky MFF UK, jeho zaměstnanci a Jednotou českých matematiků a fyziků.

Toto dílo je šířeno pod licencí Creative Commons Attribution-Share Alike 3.0 Unported.
Pro zobrazení kopie této licence navštivte http://creativecommons.org/licenses/by-sa/3.0/.