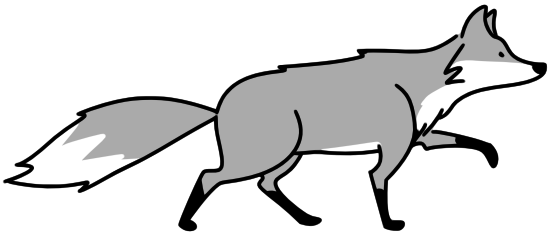
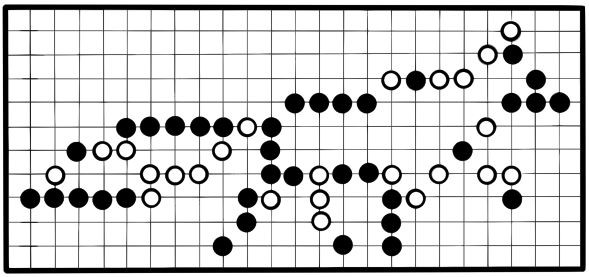
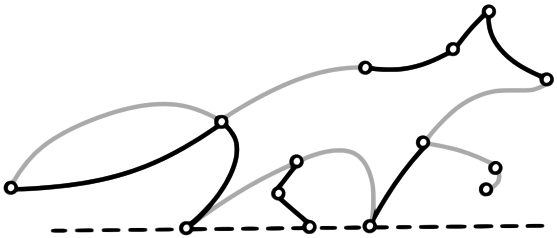
STUDENTSKÝ ČASOPIS A KORESPONDENČNÍ SEMINÁŘ

Ročník XXVI
Číslo 6

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, pošleme zpět s dalším číslem a ta nejzajímavější z nich otiskneme. Nejlepší řešitele zveme na podzim a na jaře na soustředění.

Milí řešitelé,

předně bychom se chtěli omluvit za takto pozdní vydání posledního čísla 26. ročníku. Přišlo nám důležitější dodat vám včas nová zadání ročníku 27., což se myslíme povedlo. Proto doufáme, že nám zpoždění odpustíte a najdete si chvíli na přečtení i tohoto čísla. Nová zadání v něm nejsou, přesto ale je na co se těšit!

Všechna témata a s nimi i celý ročník si uzavřeme, takže zde najdete zbylá vzorová řešení dříve zadaných úloh. V tématu týkajícím se her navíc také pohled do historie oboru a úplné výsledky turnaje spolu s krátkým komentářem k nejlepším skriptům. Děkujeme všem, kdo nám své skripty zaslali, a gratulujeme k umístění v turnaji!

Tím se dostáváme k věci úplně poslední, a tou jsou celkové výsledky ročníku. Děkujeme úplně všem, kdo se do tohoto ročníku zapojili a doufáme, že jste četli a řešili tak rádi, jako my psali a opravovali!

Výsledkovou listinu najdete na konci čísla. Zde zmíníme tři nejpilnější řešitele – Dr.$^{\mathsf{MM}}$ Václav Vladimír Janáček (96,9 bodu), Doc.$^{\mathsf{MM}}$ Jiří Kalvoda (134,4 bodu)… a úplně nejvíc bodů (143,2) nasbírala Doc.$^{\mathsf{MM}}$ Kateřina Vokálová, která měla zároveň také nejlepší skript herního turnaje! A konečně dort za nejlepší článek získává Mgr.$^{\mathsf{MM}}$ Vilém Starosta za pojednání o hledání optimálních strategií v maticových hrách, které jsme otiskli ve 4. čísle. Gratulujeme ještě jednou!

Pokud pokračujete ve studiu na střední škole, doufáme, že ve 27. ročníku už jsme od vás dostali nějaké řešení a těšíme se na další interakci. Maturantům přejeme mnoho štěstí na vysokých školách (nebo kdekoliv jinde, kam se vydáte) a také doufáme, že na M&M nezapomenete a ještě někdy se potkáme!

Mějte se co nejlépe to půjde,

Vaši organizátoři

Obsah

Řešení témat

Téma 1 – Teorie her

Díl 5: Historie teorie her

Milí řešitelé, máte v rukou závěrečný díl témátka Hry. Stručně si shrneme historii objevování teorie her a dozvíte se při tom o dalších disciplínách spadajících do tohoto oboru.

John von Neumann – vzestup génia (1903 ~ 1927)

1903
Na konci roku 1903, mezi Vánoci a Silvestrem, se v Maďarsku narodil John von Neumann [1], jeden z nejvýznamnějších vědců všech dob.

1910
Již od dětství John von Neumann mezi svými vrstevníky vynikal. V šesti letech uměl zpaměti dělit osmiciferná čísla a taky se dokázal s otcem bavit ve starořečtině. V osmi letech ovládal diferenciální a integrální počet. V devíti letech se učil zároveň anglicky, německy, francouzsky a italsky, ale nejvíc ze všeho ho zajímal dějepis. V jeho pokoji se tou dobou vršily knihy od podlahy až ke stropu. Do školy začal oficiálně chodit až v deseti letech, stejně jako ostatní maďarské děti v té době [1].

1919
V patnácti letech ho začal navštěvovat renomovaný matematik Gábor Szegő, aby ho učil pokročilou matematickou analýzu. Říká se, že když při jejich prvním setkání Gábor Szegő zjistil, s jakým neuvěřitelným matematickým talentem se setkal, dohnalo ho to k slzám. V osmnácti letech začal John von Neumann publikovat vlastní matematické objevy [2]. Jeho první publikace se týkala polynomů nad komplexními čísly [3], další se točily kolem teorie množin.

1921
Otec Johna von Neumanna se obával, že matematická dráha nepřinese jeho potomkovi dostatek peněz, proto ho přesvědčil, aby se raději dal na studium chemie [4]. John tedy studoval chemii na univerzitě v Berlíně, ale zároveň při tom zvládl dálkově vystudovat matematiku na univerzitě v Budapešti. Poté pokračoval studiem chemického inženýrství na ETH v Curychu, během něj se ale věnoval hlavně matematickému výzkumu s místními matematiky, za jednoho z nichž dokonce zaskakoval při přednášení.

1926
V roce 1926 obdržel John von Neumann inženýrský titul za chemii na ETH v Curychu a také doktorský titul za matematiku na univerzitě v Budapešti [5]. Zde začíná jeho kariéra úspěšného matematika a později také fyzika a informatika.

Klasická teorie her (1928 ~ 1959)

1928
John von Neumann jako první předložil teorii her jakožto vědeckou disciplínu, spadající do matematiky a ekonomie [6]. Ve svém článku „Zur Theorie der Gesellschaftsspiele“ [7] se John von Neumann zabýval hrami s nulovým součtem pro obecný počet hráčů. Pro hry dvou hráčů s nulovým součtem odvodil důležitý vztah „min max = max min“[1]. Tento poznatek se nám dnes může zdát jednoduchý, ale tehdy bez aparátu lineárního programování[2] muselo být těžké jej dokázat. Pro hry tří hráčů s nulovým součtem analyzoval to, které dvojici hráčů se vyplatí vytvořit tzv. koalici, čímž započal výzkum v oblasti kooperace. Také nastínil, k čemu by mohla být teorie her dobrá v praxi [7].

1944
Do našeho příběhu na chvilku vstupují dvě nové postavy. Poté, co Eduard Čech řekl Oskaru Morgensternovi o článku Johna von Neumanna z roku 1928, se Morgenstern začal intenzivně zajímat o Neumannovu práci [8]. Rozhodl se na něj navázat další prací o teorii her. Původně to měl být článek, který by Morgenstern s Neumannem konzultoval. Neumann ale tak dlouho navrhoval další poznatky, které by mohli přidat, až nakonec z jejich společného úsilí místo článku vzešla kniha „The Theory of Games and Economic Behavior“ [9] o více než 600 stranách.

V této knize Neumann a Morgenstern rozpracovali teorii v oblasti her více než dvou hráčů, ve které pokročili zejména pro případy tří a čtyř hráčů. Více než polovina knihy se týká kooperativních her, což jsou hry, ve kterých spolu mohou hráči ještě před oznámením strategie uzavírat závazné smlouvy [10]. Neumann a Morgenstern také zkoumali hry s neúplnou informací. A v neposlední řadě započali výzkum her s nenulovým součtem.

1947
John von Neumann se setkal s Georgem Dantzigem těsně poté, co George Dantzig publikoval simplexovou metodu[3]. Tou dobou John von Neumann o lineárním programování nic nevěděl, ale když mu Dantzig tento koncept popsal, Neumann

[1] Vzpomeňte si na úlohu z druhého dílu, kde jste si to sami ověřili na malém příkladu, nebo na první kapitolu z třetího dílu, kde jsme tento vztah vysvětlili v obecnosti.
[2] Jedná se o teorii vybudovanou kolem soustav lineárních nerovnic.
[3] Simplexová metoda je algoritmus pro řešení úlohy lineárního programování.

si jej okamžitě spojil se svým výzkumem maticových her. V tu chvíli vymyslel teorii duality a celou ji Georgi Dantzigovi během jedné hodiny spatra vysvětlil [1]. Georg Danzig poté publikoval tento objev společně s Alexem Ordenem [11].

1950
John Forbes Nash dokázal [12], že každá bimaticová hra (a obecněji každá nekooperativní současná hra n hráčů s nenulovým součtem) má stabilní řešení ve smíšených strategiích, zobecněné na tzv. Nashovo ekvilibrium. Jedná se o mnohem obecnější úlohu, o které bylo později dokázáno, že je NP-úplná, a to již pro n=2, tedy naše známé bimaticové hry. John Forbes Nash byl později za svou práci na ekvilibriích odměněn Nobelovou cenou za ekonomii.

Kombinatorická teorie her (1960 ~ 1975)

1960
V šedesátých letech se začala rozvíjet kombinatorická teorie her. Ta se zabývá hrami, ve kterých se hráči střídají v tazích a výsledná odměna je určena až po mnoha krocích, které určitým způsobem mění stav hry. Klasické příklady kombinatorických her jsou šachy, dáma, go, nim, žolík, kanasta, bridge, ...

1969
John Horton Conway vymyslel zvláštní hru jménem Hackenbush[4]. Zkoumání této hry ho dovedlo až k objevu tzv. *surreal numbers*, což je nestandardní model reálných čísel shodující se s reálnými čísly ve všech vlastnostech popsatelných predikátovou logikou prvního řádu, v němž existují podivná čísla jako ω (tj. číslo větší než všechna reálná čísla), ϵ (tj. číslo větší než nula ale menší než všechna kladná reálná čísla), $-\omega$, ϵ^2, $3+2\epsilon$, $\omega-5$, ω^2, $\omega^3+\sqrt{\omega}+\epsilon$, ω^ω ...

Jeho první inspirací prý bylo studium hry go, konkrétně jejích koncovek (yose), při němž ho napadlo, že by mohl jednotlivé části hrací plochy (goban) popsat čísly vyjadřujícími určité dílčí skóre, která by pak stačilo posčítat. Při pokusech o realizaci této myšlenku zjistil, že mu klasická reálná čísla nebudou stačit.

Na hru go se ovšem nepoužije plná šíře surreal numbers. Oproti tomu popis pozic ve hře Hackenbush potřebuje všechna surreal numbers [13].

Práce na poli kombinatorické teorie her mezitím nezastavila vývoj klasické teorie her. Dodnes se v klasické teorii her aktivně bádá, například na alternativních definicích ekvilibrií nebo v oblasti evoluční teorie her. Kromě toho, že pokračuje primární výzkum, teorie her nachází také nová využití. V posledním desetiletí se teorie her například začala využívat při ověřování bezpečnosti kryptoměn.

Umělá inteligence pro hry (1976 ~ 2020)

1976
Počítačové programy začaly v šachu porážet už i silnější amatéry [14].

[4]Spodní liška na obálce čísla vyjadřuje, jak může Hackenbush vypadat.

1994
Program Chinook porazil v dámě Mariona Tinsley, nejlepšího hráče dámy své doby [15]. Jeho hlavní silnou stránkou byla rozsáhlá databáze optimálních koncovek. Ta údajně obsahovala 250 miliard pozic zkomprimovaných do souboru menšího než 6 GB.

Utkání v roce 1994 se mělo hrát na 20 zápasů rozložených do pěti dní. Prvních 6 zápasů skončilo remízou. Po šestém zápase musel Marion Tinsley utkání přerušit z vážných zdravotních důvodů. Když mu později diagnostikovali rakovinu slinivky břišní, Marion Tinsley pověřil Dona Laffertyho, aby utkání dohrál za něj. Lafferty byl tou dobou druhý nejlepší hráč dámy ve světovém žebříčku a Tinsley byl jeho dlouhodobý rival [16]. Lafferty měl navíc zkušenosti s hrou proti předchozí verzi programu Chinook. Pověřit ho dohráním utkání se zdálo být jedinou rozumnou možností. Zatímco pokračování utkání bylo odloženo, Tinsley na své smrtelné posteli předával Laffertymu své zkušenosti s hrou proti nové verzi programu Chinook [15].

Když utkání pokračovalo, Don Lafferty obhájil další 3 remízy. V desáté hře prohrál, čímž se ručička hodin začala naklánět ve prospěch umělé inteligence. Následovalo 10 remíz v řadě. Program Chinook se tedy stal díky svým 19 remízám a 1 výhře těsným vítězem utkání. Přestože se obvykle uvádí, že Chinook tehdy porazil Mariona Tinsleyho, zvítězil spíše nad Člověkem než nad jedním konkrétním člověkem.

1997
Šachový počítač Deep Blue porazil Garryho Kasparova, nejlepšího hráče šachů své doby [17]. Deep Blue stavěl hlavně na ohromné hrubé síle k prohledávání stavového prostoru a na dobré ohodnocovací funkci pro kvalitu pozic.

Počítač se s velmistrem utkal nejprve v roce 1996, kdy Kasparov zvítězil $4:2$. Utkání Kasparova s vylepšenou verzí pak proběhlo v roce 1997, kdy Deep Blue zvítězil $3\frac{1}{2}:2\frac{1}{2}$. Utkání bylo ve své době kritizováno kvůli konání za nepříznivých herních podmínek pro Garryho Kasparova, ale jak potvrdila pozdější utkání člověka proti počítači uskutečněná za jiných podmínek, počítač se doopravdy stal schopným porazit šachového velmistra.

2006
Šachové programy začaly porážet profesionální šachové hráče už i na běžných osobních počítačích [18].

2007
Díky výpočtům, které běžely na až 200 osobních počítačích a jejichž dokončení trvalo řadu let, bylo dokázáno, že v dámě si oba hráči mohou vynutit remízu [19]. Dáma je tedy nyní považována za definitivně vyřešenou hru. V tom se dáma liší od šachů a go, kde sice máme (nyní již pro obě hry) umělou inteligenci schopnou

nadlidského výkonu, ale nikdo neví, k jakému výsledku vede skutečně optimální hra.

2016

AlphaGo porazilo v go Lee Sedola, nejlepšího hráče své doby [20]. Bylo toho dosaženo spojením hlubokých konvolučních neuronových sítí s pokročilými metodami stochastického prohledávání stavového prostoru.

Go je mnohými považováno za nejnáročnější hru všech dob. V této hře se hráči střídají v pokládání kamenů na čtvercovou plochu. Cílem je obklíčit soupeře a získat co největší území. Pro umělou inteligenci je go těžké, neboť má příliš rozsáhlý stavový prostor, mnohem větší než mají dáma nebo šachy. Lidé oproti tomu umí při svém uvažování stavový prostor velmi efektivně omezovat díky intuici na tvary. Taková intuice je u nás rozvinutější než sekvenční uvažování, které je nutné například při hraní dámy.

Hra go byla dlouhodobě proslulá jako údajně neřešitelný úkol pro umělou inteligenci. V době, kdy šachové programy porážely profesionální hráče, ve hře go ještě neuměly nejlepší programy porážet ani mírně pokročilé amatéry [21]. V roce 2015 odborníci na umělou inteligenci tvrdili, že nás dělí ještě 10 let od chvíle, kdy umělá inteligence porazí profesionální hráče.

Demis Hassabis, zakladatel start-upu DeepMind, se rozhodl pokořit go mnohem rychleji. Začal se svým týmem pracovat na programu AlphaGo [20]. O tomto projektu řekl: *We think of DeepMind as kind of an Apollo program effort for AI. Our mission is to fundamentally understand intelligence and recreate it artificially.*[5] Doufal, že pomocí nejnovějších metod posilovaného strojového učení vyvine algoritmus schopný nadlidského výkonu při hraní go. Nakonec sáhl po hlubokých neuronových sítích (tzv. deep learning), konkrétně po konvolučních sítích [22], které se v několika předchozích letech osvědčily pro automatické rozpoznávání obrazu. Tyto neuronové sítě použil jako „zázračnou krabičku“ dosazenou na dvě místa uvnitř již známého algoritmu Monte-Carlo Tree-search, který umí aproximovat očekávané výsledky v konečných kombinatorických hrách [23].

Utkání se mělo hrát na 5 zápasů, první z nich měl proběhnout 9. 3. 2016. Před ním si byl Lee Sedol jistý, že zvítězí a že by zápas neměl být moc těsný [20]. Tvrdil, že chce vyhrát 5 : 0 a že by bral jako neúspěch, pokud by vyhrál pouze 4 : 1. Těsně před začátkem Lee Sedol řekl, že lidská intuice je příliš pokročilá na to, aby ji nějaká umělá inteligence mohla dohnat. Přímý přenos zápasu Lee Sedol vs. AlphaGo sledovalo 200 milionů diváků [24].

K velkému překvapení hráčské komunity zvítězilo v první hře AlphaGo [25]. Když pak v druhém i třetím zápase opět a znovu vyhrálo AlphaGo, budilo to už mnohem menší překvapení. Odlišný byl čtvrtý zápas [26]. Začátek vypadal podobně. Lee Sedol opět agresivně útočil, ale AlphaGo skálopevně vzdorovalo a pomalu si budovalo malou teritoriální výhodu. Po 77 odehraných tazích však

[5] v překladu: DeepMind je pro nás něco jako projekt Apollo pro umělou inteligenci. Naší misí je hluboce porozumět inteligenci a pak ji stvořit uvnitř počítače.

nastal zvrat. Lee Sedol odhalil velmi dobrý netradiční tah. Zahrál ho jako tah 78 a tím AlphaGo zaskočil. Jak DeepMind zpětně prozradil, AlphaGo až o osm tahů později začalo odhadovat výrazně nižší šanci na výhru, což naznačuje, že ještě po dobu dalších sedmi tahů netušilo, k jakým následkům tah 78 vedl [20]. Diváci začali tahu 78 přidělovat heroické přezdívky jako „God's move", „Divine move", „The bot-confusing move", „The short-circuiting move" nebo „Humanity's last stand" [27]. Lee Sedol pak dovedl čtvrtý zápas ke zdárnému konci. V pátém zápase však ještě jednou vyhrálo AlphaGo [20], čímž zakončilo utkání se skóre 4 : 1. Březen 2016 se navždy zapsal do historie. Lidstvo bylo poraženo počítačem ve hře, která byla do té doby považována za poslední triumf lidské inteligence, intuice a kreativity [27].

V následujících letech DeepMind vynalezl ještě silnější verze AlphaGo [28], které dokázaly porazit 100 : 0 původní verzi, která porazila 4 : 1 Lee Sedola. Nezdá se, že by lidé ještě někdy byli schopni porazit počítače ve hře go.

Závěr

Jak asi už tušíte, tady naše výprava končí. Dostali jsme se do přítomnosti – a v ní to můžete být právě vy, kdo zanechá svůj otisk pro budoucí generace. Ať už se budete pokoušet posouvat hranici lidského poznání v teorii her nebo v jakékoliv jiné disciplíně, přejeme vám hodně zdaru! Vězte, že Riki bude pro vás vždycky držet tlapky sevřené.

Zdroje

[1] *Wikipedia – John von Neumann.* URL: https://en.wikipedia.org/wiki/John_von_Neumann.

[2] *Chess Programming Wiki.* URL: https://www.chessprogramming.org/John_von_Neumann.

[3] *MacTutor – Fekete.* URL: http://mathshistory.st-andrews.ac.uk/Biographies/Fekete.html.

[4] *MacTutor – Von Neumann.* URL: http://mathshistory.st-andrews.ac.uk/Biographies/Von_Neumann.html.

[5] *Mathematics Genealogy Project.* URL: https://www.genealogy.math.ndsu.nodak.edu/id.php?id=53213.

[6] *Wikipedia – Game theory.* URL: https://en.wikipedia.org/wiki/Game_theory.

[7] J. V. Neumann. „Zur Theorie der Gesellschaftsspiele". In: *Mathematische Annalen* 100 (1928), s. 295–320.

[8] *Wikipedia – Oskar Morgenstern.* URL: https://en.wikipedia.org/wiki/Oskar_Morgenstern.

[9] J. Neumann a O. Morgenstern. „Theory of Games and Economic Behavior.“ In: *Journal of the American Statistical Association* 40 (1944), s. 263.

[10] Harold W. Kuhn a A. W. Tucker. „John von Neumann's work in the theory of games and mathematical economics“. In: *Bulletin of the American Mathematical Society* 64 (1958), s. 100–122.

[11] G. B. Dantzig a A. Orden. „A Duality Theorem Based on the Simplex Method“. In: *Symposium on Linear Inequalities and Programming, Report 10, Project SCOOP* 64 (1952), s. 51–55.

[12] J. F. Nash. „Equilibrium Points in N-Person Games.“ In: *Proceedings of the National Academy of Sciences of the United States of America* 36 1 (1950), s. 48–49.

[13] J. H. Conway. „On Numbers and Games“. In: *London Academic Press* (1976).

[14] F. Hapgood. „Computer chess bad – human chess worse“. In: *New Scientist* (1982), s. 827–830.

[15] J. Propp. *Chinook*. URL: `https://web.archive.org/web/20060829085713/http://www.math.wisc.edu/~propp/chinook.html`.

[16] R. Fortman. *Don Morgan Lafferty*. URL: `https://webdocs.cs.ualberta.ca/~chinook/people/Lafferty.php`.

[17] J. Schaeffer a A. Plaat. *Kasparov versus Deep Blue: The Re-match*. URL: `http://webdocs.cs.ualberta.ca/~jonathan/PREVIOUS/Grad/Papers/db.html`.

[18] *Wikipedia – Human–computer chess matches*. URL: `https://en.wikipedia.org/wiki/Human%E2%80%93computer_chess_matches`.

[19] J. Schaeffer et al. „Checkers Is Solved“. In: *Science* 317 (2007), s. 1518–1522.

[20] *AlphaGo (movie)*. URL: `www.alphagomovie.com`.

[21] *History of Go-playing Programs*. URL: `https://www.britgo.org/computergo/history`.

[22] D. Silver et al. „Mastering the game of Go with deep neural networks and tree search“. In: *Nature* 529 (2016), s. 484–489.

[23] T. Pfeiffer. *What did AlphaGo do to beat the strongest human Go player?* URL: `www.alphagomovie.com`.

[24] *The Google DeepMind challenge match*. URL: `https://deepmind.com/alphago-korea`.

[25] BBC News. *Artificial intelligence: Google's AlphaGo beats Go master Lee Sedol*. URL: `https://www.bbc.com/news/technology-35785875`.

[26] *Humans strike back: How Lee Sedol won a game against AlphaGo*. URL: `https://www.newscientist.com/article/2080486-humans-strike-back-how-lee-sedol-won-a-game-against-alphago/`.

[27] *Match of the century - Lee Sedol vs Alpha Go.* URL: `https://youtu.be/I2WFvGl4y8c`.

[28] D. Silver et al. „A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play“. In: *Science* 362 (2018), s. 1140–1144.

Řešení 4. série

Problém 1

Zadání:
Upravte model konfliktu hrdlička-jestřáb, aby popisoval děj, ve kterém se dvě hrdličky nedělí o potravu napůl, ale soutěží v tom, kdo déle zůstane koukat na jídlo. Až to jedna z nich vzdá a odletí, druhá hrdlička sní celé jídlo. Rozdíl bude v tom, že se levá horní hodnota v bimatici sníží o nějaké malé číslo ε, jež bude vyjadřovat penalizaci za ztrátu času při čekání v případě setkání hrdličky s hrdličkou. Jaká bude v upravené hře optimální strategie pro jedince? A co by v upravené hře bylo nejlepší pro celou populaci v součtu?

Řešení:
Upravená bimatice vypadá následovně:

	H	J
H	$3-\varepsilon \setminus 3-\varepsilon$	$0 \setminus 6$
J	$6 \setminus 0$	$-6 \setminus -6$

Co je teď optimální strategií pro jedince? Stejně jako v původním příkladu si to vyjádříme v závislosti na z, které vyjadřuje poměr hrdliček v populaci. Průměrný zisk hrdličky je $(3-\varepsilon)\cdot z$. Průměrný zisk jestřába je $6\cdot z+(-6)\cdot(1-z)=12z-6$. Najdeme hodnotu z, při které se rovnají.

$$(3-\varepsilon)\cdot z = 12z-6$$

$$6 = (9+\varepsilon)\cdot z$$

$$z = \frac{6}{9+\varepsilon}$$

Jako zkoušku můžeme vypočítat, že při našem z vychází průměrný zisk hrdličky:

$$\frac{6\cdot(3-\varepsilon)}{9+\varepsilon}$$

Průměrný zisk jestřába vychází:

$$\frac{6\cdot 6}{9+\varepsilon} - \frac{(3+\varepsilon)\cdot 6}{9+\varepsilon} = \frac{(3-\varepsilon)\cdot 6}{9+\varepsilon}$$

Zisky se rovnají, při tomto poměru budou hrdličky i jestřábi prosperovat stejně.

Vidíme, že rovnovážným bodem je $z = \frac{2}{3} - \frac{2}{27}\varepsilon + \mathcal{O}(\varepsilon)$, což je v souladu s naším očekáváním, neboť bez penalizace za čekání bylo rovnovážným bodem $z = \frac{2}{3}$, a pokud čekání penalizujeme, dává to výhodu jestřábům (pro malé ε zhruba lineárně).

Co je optimem pro celou populaci? Bude to zase stav, ve kterém všichni používají strategii H? Pojďme si spočítat průměrný zisk r jako funkci z.

V dostatečně velké populaci je šance na setkání dvou hrdliček z^2, riziko střetu dvou jestřábů je $(1-z)^2$, s pravděpodobností $2z(1-z)$ dojde k zahnání hrdličky jestřábem.

$$r = (3-\varepsilon)\cdot z^2 + 3\cdot 2z(1-z) + (-6)\cdot(1-z)^2$$

Tuto funkci zjednodušíme.

$$r = 3z^2 - \varepsilon z^2 + 6z - 6z^2 + -6 + 12z - 6z^2$$

Nejprve jsme roznásobili závorky, nyní posčítáme jednočleny se stejnou mocninou z.

$$r = -(9+\varepsilon)z^2 + 18z - 6$$

Zderivujeme naši funkci podle z.

$$r' = -(9+\varepsilon)\cdot 2z + 18$$

Abychom našli maximum, položíme derivaci rovnou nule.

$$0 = -(9+\varepsilon)\cdot 2z + 18$$

Vyřešíme tuto snadnou lineární rovnici.

$$(9+\varepsilon)\cdot z = 9$$

Obdržíme přesný výsledek.

$$z = \frac{9}{9+\varepsilon}$$

Vidíme, že optimem pro celou populace je $z = 1 - \frac{1}{9}\varepsilon + \mathcal{O}(\varepsilon)$, což je opět v souladu s naším očekáváním.

Problém 2

Zadání:
Převeďte iterované vězňovo dilema na obyčejnou bimaticovou hru, popište několik metastrategií a vyjádřete jejich vzájemné zisky (průměrný počet bodů na kolo při dlouhé interakci s daným protějškem). Analyzujte optimální metastrategii v této „statické verzi“ vězňova dilematu.

Řešení:
Problém 2 nemá vzorové řešení. Každý z vás zvolil jinou množinu metastrategií. Hodnocena byla přesnost, rozsah, nápaditost a kvalita prezentace výsledků.

Problém 3

Zadání:
Najděte optimální (smíšenou) strategii pro každé hráče v této bimaticové hře bez opakování. K jakému průměrnému výsledku každého hráče hra povede, pokud se oba hráči budou rozhodovat čistě racionálně? (Tato úloha nevycházela z textu témátka.)

Řešení:
Uvažme prvního hráče. Žádný řádek nedominuje jiný řádek. Stejně tak žádná konvexní kombinace dvou řádků nedominuje zbývající řádek. Přesuňme se k druhému hráči. Zde je prostřední sloupec dominován průměrem levého a pravého sloupce. Škrtáme prostřední sloupec.

Podívejme se zpět na hráče 1. Nyní už je spodní řádek dominovaný (například prostředním řádkem). Škrtáme spodní řádek. Dostáváme bimaticovou hru 2×2.

$3 \setminus 9$	$6 \setminus 0$
$7 \setminus 0$	$5 \setminus 9$

Tato bimatice nemá „zobecněný sedlový bod“ (optimum v ryzích strategiích).

- Z bodu $3 \setminus 9$ bude první hráč vždy chtít přejít do bodu $7 \setminus 0$.
- Z bodu $7 \setminus 0$ bude druhý hráč vždy chtít přejít do bodu $5 \setminus 9$.
- Z bodu $5 \setminus 9$ bude první hráč vždy chtít přejít do bodu $6 \setminus 0$.
- Z bodu $6 \setminus 0$ bude druhý hráč vždy chtít přejít do bodu $3 \setminus 9$.

Podívejme se na smíšené strategie. Nechť první hráč volí strategii 1 s pravděpodobností x a druhý hráč volí svou strategii 1 s pravděpodobností y. Následující rovnice vyjadřuje, jaké má první hráč zvolit x, aby se druhému hráči vyplatily obě strategie stejně.

$$9x = 9\,(1 - x)$$
$$x = \frac{1}{2}$$

Následující rovnice vyjadřuje, jaké má druhý hráč zvolit y, aby se prvnímu hráči vyplatily obě strategie stejně.

$$3y + 6(1 - y) = 7y + 5(1 - y)$$
$$6 - 3y = 5 + 2y$$
$$1 = 5y$$

$$y = \frac{1}{5}$$

Dostali jsme optimální smíšenou strategii $(\frac{1}{2}; \frac{1}{2}; 0)$ pro prvního hráče a $(\frac{1}{5}; 0; \frac{4}{5})$ pro druhého hráče. Při jejich použití je průměrný zisk prvního hráče 5,4 bodu a průměrný zisk druhého hráče 4,5 bodu.

Výsledky turnaje

Sláva vítězům, čest poraženým, a velké poděkování všem účastníkům!

číslo turnaje	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Doc.MM Kateřina Vokálová	2.	1.	1.	4.	1.	1.	1.	1.	1.	1.
Hromislav	-	3.	2.	1.	3.	3.	4.	3.	3.	4.
Všemysl	-	-	7.	7.	2.	2.	5.	2.	2.	2.
Božetěch	1.	4.	3.	3.	4.	6.	3.	6.	6.	6.
Dr.MM Václav Janáček, Doc.MM Jiří Kalvoda	5.	6.	5.	2.	5.	5.	6.	5.	5.	5.
Mgr.MM Vilém Starosta	3.	2.	4.	5.	6.	7.	7.	7.	7.	7.
Mgr.MM Lukáš Veškrna	-	-	-	-	-	4.	2.	4.	4.	3.
Dr.MM Martin Fof	4.	5.	6.	8.	8.	9.	9.	8.	9.	9.
Soběrad	-	-	-	6.	7.	8.	8.	9.	8.	8.
Nezabud	-	-	9.	9.	9.	10.	10.	10.	10.	10.
Mgr.MM Klára Pernicová	6.	7.	8.	10.	10.	11.	11.	11.	11.	11.

V tabulce je uvedeno pořadí mezi všemi „lidskými hráči“. Řešitele M&M uvádíme celým jménem. Ostatním účastníkům turnaje jsme kvůli ochraně osobních údajů změnili jména na staročeská.

Komentář

Jistě vás zajímá, jakou strategii volili při psaní svých skriptů ostatní účastníci turnaje. Zmiňme tedy v krátkosti první tři, jejichž skripty jsou shodou okolností navzájem velmi rozdílné. Doc.[MM] Kateřina Vokálová napsala dlouhý skript s mnoha podmínkami, který nejprve klasifikoval aktuálního soupeře a poté cíleně reagoval na konkrétní hráče, kteří v turnaji byli. Hromislav (uveden v tabulce na druhém místě) oproti tomu napsal velmi jednoduchý ale účinný skript, jehož rozhodovací mechanismus má pouhých osm řádků. A nakonec Všemysl, jenž je uveden v tabulce na třetím místě, ale přibližně celou druhou polovinu turnajů končil na druhém místě, přistoupil k problému úplně jinak – použil neuronové sítě. Je zajímavé, jak se takto odlišné přístupy osvědčily v turnaji a získaly téměř stejný počet bodů. Hromislavův skript pro jeho jednoduchost otiskujeme:

```
1  def reward(self, result):
2      self.opponent_moves.append(result.opp_move)
3
4  def next_move(self):
5      if len(self.opponent_moves) < 2:
6          return Move.cooperate
7      elif self.opponent_moves[-1] == Move.deceive and
           ↪ self.opponent_moves[-2] == Move.deceive:
8          return Move.betray
9      elif self.opponent_moves[-1] == Move.betray and
           ↪ self.opponent_moves[-2] == Move.betray:
10         return Move.a_safe_way
11     else:
12         return self.opponent_moves[-1]
```

Markét, Martin; martin.dvorak@matfyz.cz
e-mailová konference: hry@mam.mff.cuni.cz

Téma 2 – Výpočetní modely

Vzorová řešení 4. série

Problém 1

Zadání:
Zamyslete se nad tím, jaké všechny vlastnosti od RNG chceme. Ukažte, že námi popsaná definice je splňuje. Pokud bychom měli jiné rozšíření, které některé z nich nesplňuje, umíme pomocí něj sestrojit stroj, jehož výstup by tyto vlastnosti měl? Pro představu uvádíme dva příklady alternativních rozšíření: „V každém kroku si nejdříve vygeneruje skutečně náhodný bit. Pokud je to 1, tak vygeneruje ještě jeden a ten vypíše. Pokud je to 0, tak zopakuje ten, co vypsal posledně.“ „Kdyby měl vypsat třetí jedničku v řadě, tak místo ní vypíše nulu.“

Řešení:
Od generátoru náhodných čísel (v našem případě spíše generátoru náhodných bitů) většinou chceme následující vlastnosti:

- **Nestrannost**

  Všechny výstupní hodnoty (v našem případě nula a jednička) mají vždy stejnou pravděpodobnost. Tuto podmínku nesplňuje generátor, který ve 3/4 případů vrací jedničku ani generátor, který vždy začne nulou.

- **Nezávislost**

  Předchozí vygenerované bity nemají žádný vliv na následující bity. Generátor, který si na začátku férově vygeneruje jeden bit a ten pak opakuje by tuto podmínku nesplňoval.

Z uvedených příkladů alternativních rozšíření první není nezávislý - má pravděpodobnost 3/4, že zopakuje předchozí bit. (S pravděpodobností 1/2 opakuje poslední vygenerovaný a ve zbývající 1/2 případů má pořád šanci 1/2 se trefit.) Druhý příklad pak není ani nestranný (nuly jsou o něco pravděpodobnější než jedničky), ani nezávislý (pokud vygeneroval dvě jedničky, určitě bude následovat nula).

Námi navrhovaná definice „dokonalé“ náhodnosti naopak obě tyto vlastnosti splňuje.

Pokud všechny posloupnosti délky n mají pravděpodobnost $\frac{1}{2^n}$, tak sekvence končící nulou mají celkovou pravděpodobnost 1/2, stejně tak sekvence končící nulou a n-tý bit je tudíž nutně nestranný.

Pro libovolnou posloupnost délky $n-1$ mají obě její možná pokračování stejnou pravděpodobnost, a tudíž n-tý bit je nutně nezávislý.

Klasický problém je to, že můžeme mít generátor náhodných čísel, který je nezávislý, ale není nestranný. Dejme tomu, že máme generátor, který vrací jedničku s pravděpodobností p a nulu s pravděpodobností $1-p$. Pokud potřebujeme

nestranný generátor, tak místo čtení jednoho bitu přečteme rovnou dva a pokračujeme podle následující tabulky:

00	přečteme další dva bity
01	vrátíme 0
10	vrátíme 1
11	přečteme další dva bity

Všimněte si, že 01 i 10 mají stejnou pravděpodobnost $p(1-p)$.

Problém 2

Zadání:
Co nám RNG umožňuje řešit (nebo alespoň usnadňuje)?

Řešení:
Z „klasických“ problémů nám RNG umí usnadnit třeba třízení pomocí algoritmu QuickSort[6]. Pokročilejší příklady pak můžou být algoritmy typů Monte Carlo[7] a Las Vegas[8].

Problém 3

Zadání:
Popište stroj na řešení problému dvou loupežníků. Uveďte jeho asymptotickou složitost.

Řešení:
Nejjednodušší deterministické řešení dvou loupežníků je postupně zkoušet všechny možná rozdělení na dvě hromady. Dokonce existuje domněnka, že v obecném případě tento problém ani nejde řešit o moc rychleji. (Následující úlohy mají snazší řešení.)

`vstup`
`fronta` (Sem uložíme celý vstup abychom jej mohli opakovaně číst. Při čtení z fronty tudíž budeme předměty vracet zpátky.)
`registr (počáteční hodnota 0)` (stav výpočtu)
`ALU (počáteční hodnota 0)` (příští testované rozdělení)
`ALU (počáteční hodnota 0)` (jedna hromada)
`ALU (počáteční hodnota 0)` (druhá hromada)
`ALU (počáteční hodnota 0)` (aktuální bit rozdělení)
`ALU (počáteční hodnota 0)` (zbývající bity rozdělení)
`výstup`

[6] `https://cs.wikipedia.org/wiki/quicksort`
[7] `https://cs.wikipedia.org/wiki/Algoritmus_typu_Monte_Carlo`
[8] `https://cs.wikipedia.org/wiki/Algoritmus_typu_Las_Vegas`

```
$,_,0,_,_,_,_,_ → #, ,1,2,0,+,0,0,+,0,0,+,0,0,+,1,0,+,
```
(na konec fronty umístíme mřížku jako zarážku a začneme řešit první rozdělení)
```
a,_,0,_,_,_,_,_ → a, ,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,
```
(vstup kopírujeme do fronty)
```
$,#,1,_,x,x,_,_ → _,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,ANO
```
(úspěšně nalezené řešení)
```
$,#,1,r,_,_,0,0 → #,>,1,r,1,+,0,0,+,0,0,+,r,2,/,r,2,%,
```
(zahájení dalšího pokusu)
```
$,#,1,_,_,_,_,_ → _,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,NE
```
(všechna rozdělení byla vyzkoušena)
```
$,a,1,r,x,y,d,0 → a,>,1,r,0,+,x,a,+,y,0,+,d,2,/,d,2,%,
```
(přiřazení prvku k první hromadě)
```
$,a,1,r,x,y,d,1 → a,>,1,r,0,+,x,0,+,y,a,+,d,2,/,d,2,%,
```
(přiřazení prvku k druhé hromadě)

Problém 4

Zadání:
Popište stroj na nedeterministické řešení problému dvou loupežníků. Uveďte jeho asymptotickou složitost.

Řešení:
Nedeterministické řešení dvou loupežníků náhodně rozdělí předměty ze vstupu na dvě hromady a poté zkontroluje, zda se ceny hromad rovnají. Používáme dvě ALU pro průběžné počítání cen hromad. (Existuje řešení, kterému stačí jedno rozšíření ALU. Dokážete jej vymyslet?)

```
vstup
RNG
ALU (počáteční hodnota 0)
ALU (počáteční hodnota 0)
výstup
```

```
$,_,x,x → , , , , , ,ANO
$,_,x,y → , , , , , ,NEVÍM
a,0,x,y → x,a,+,y,0,+,
a,1,x,y → x,0,+,y,a,+,
```

V každém kroku zpracujeme jeden předmět ze vstupu. Asymptotická složitost je tudíž $\mathcal{O}(n)$.

Problém 5

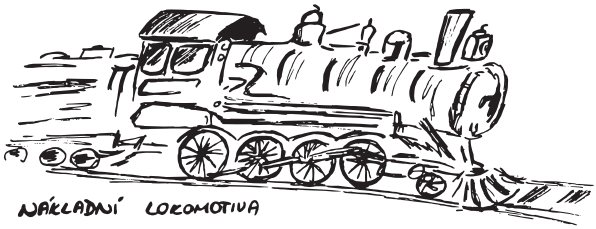
Zadání:
Popište stroj, který nedeterministicky poznává palindromy za použití pouze rozšíření RNG, IO a zásobník.

Řešení:
Asi nejelegantnější řešení vyžaduje použití jednoho zásobníku jako registru. Stroj zapisuje vstup do zásobníku až do první jedničky z RNG. Pokud je jednička následována nulou, testujeme možnost sudé délky palindromu. Pokud je jednička následována další jedničkou, testujeme možnost liché délky palindromu. Pro jednoduchost předpokládáme, že vstup neobsahuje číslice.

```
zásobník                                          („registr“)
IO
zásobník                                          („hlavní“ zásobník)
RNG


$,$,$,_ → ,ANO,                          (tohle ošetřuje prázdný vstup)
$,a,$,0 → 0, ,a   (nula v „registru“ indikuje zpracovávání první půlky vstupu)
$,a,$,1 → a, ,     (při potkání jedničky uložíme aktuální vstup do „registru“)
0,a,b,0 → 0, ,ba
0,a,b,1 → a, ,b
1,a,a,_ → 1, ,      (1 v „registru“ indikuje zpracovávání druhé půlky vstupu)
b,a,b,0 → 1, ,a      (sudá délka palindromu a uprostřed jsou dva znaky „b“)
b,a,a,1 → 1, ,         (lichá délka palindromu a uprostřed je sekvence „aba“)
_,_,_,_ → ,NEVÍM,                (cokoliv jiného znamená selhání postupu)
```

Možné konce a jejich význam (s trochu volnější notací):

```
_,ne-$,$,ne-$ = půlku vstupu jsme tipli moc pozdě
_,$,ne-$,ne-$ = půlku vstupu jsme tipli moc brzo
_,a,b,1 = vstup není symetrický kolem tipnutého středu
0,a,b,ne-b = na tipnutém středu jsou dva různé znaky
1,a,ne-a,b = kolem tipnutého středu jsou dva různé znaky
```



Problém 6

Zadání:
Popište stroj, který nedeterministicky pozná, zda množina čísel na vstupu obsahuje svůj průměr. Nepoužívejte zásobníky, fronty, pásky ani pole.

Řešení:
Náhodně si vybereme, který prvek zkusíme prohlásit za průměr.

```
RNG
vstup
ALU (počáteční hodnota 0)                                    (součet)
ALU (počáteční hodnota 0)                                    (součet)
registr (počáteční hodnota 0)
                  (tip na průměr, X po spočítání očekávaného součtu do 2. ALU)
výstup

_,_,x,x,X ->  , , , , , , ,ANO
_,_,x,y,X ->  , , , , , , ,NEVÍM
_,$,s,p,t -> s,0,+,p,t,*,X,                  (ověříme, že je průměr správně)
0,a,s,p,t -> s,a,+,p,1,+,t,                   (normální zpracování prvku)
1,a,s,p,_ -> s,a,+,p,1,+,a,       (nahrazení dosavadního tipu na průměr)
```

Problém 7

Zadání:
Představte si, že máte registr, který začíná na hodnotě 1, a vaším úkolem je zjistit, zda je možné jej vynulovat. Zádrhel je v tom, že registr má nějaká omezení na to, jak jej lze měnit. Dostanete číslo n na vstupu a pole obsahující $n \times n$ bitů, kde bit na pozici $i \cdot n + j$ (číslováno od nuly) určuje, zda můžete změnit hodnotu v registru z i na j, přičemž žádné jiné změny nejsou povoleny. Instance problému tedy může vypadat například takto:

$n = 4$
Obsah pole: 0000 0010 0001 1000
Pole reprezentuje následující tabulku:

		nová			
		0	1	2	3
původní	0	0	0	0	0
	1	0	0	1	0
	2	0	0	0	1
	3	1	0	0	0

Odpověď: ANO (1 -> 2 -> 3 -> 0)

Jak rychle umíte tento problém řešit? Jak rychle jej umíte řešit nedeterministicky? Pokud vám to pomůže, tak můžete předpokládat, že n je mocnina dvojky.

Řešení:
Tento problém je už pro naši notaci příliš komplikovaný, což je důvod, proč jsem se neptal na popis stroje. Nastíním možné algoritmy a jejich očekávanou časovou složitost.

- **Deterministický algoritmus - dosažitelné stavy**

  Algoritmus má frontu stavů, na které se má ještě podívat a pro každý stav si pamatuje, zda jej již prozkoumal. Na začátku není žádný stav prozkoumaný a máme se podívat na stav 1.

  ```
  Dokud fronta není prázdná:
     Odeber stav z fronty
     Pokud tento stav ještě není prozkoumaný,
        přidej všechny z něj přímo dosažitelné stavy do fronty
        a označ tento stav za prozkoumaný
  ```

  Celková složitost: $\mathcal{O}(n^2)$

- **Nedeterministický algoritmus - uhodnutí cesty**

  Z RNG vždy přečteme $\log n$ bitů a zkontrolujeme, zda se jedná o validní krok. Pokud takhle dojdeme do nuly, tak jsme vyhráli. Pokud se pokusíme o nevalidní krok nebo po $n-1$ krocích pořád nejsme v nule, tak jsme prohráli.

Problém 8

Zadání:
Jakými jinými způsoby lze nahlížet na nedeterminismus? Jak by se to dalo implementovat v našich strojích?

Pro příklad uvedu dva další pohledy na nedeterminismus:

Místo rozšíření RNG můžeme mít rozšíření Fork, které umožňuje našemu stroji se naklonovat. Postup výpočtu si můžeme představit jako větvící se cestu. Když se stroj dopracuje k rozcestí, tak se rozkopíruje, aby mohla každou větví pokračovat jedna kopie stroje. Opět koukáme na to, zda alespoň jedna kopie najde cíl.

Trochu zajímavější alternativa je místo RNG mít další vstup, který čte takzvaný certifikát. Idea je taková, že pokud je odpovědí ANO, tak existuje certifikát, který způsobí, že náš stroj vstup přijme. Pokud je odpovědí NE, tak naopak bude vstup odmítnut pro libovolný certifikát. Můžete si představit, že správný certifikát odpovídá tomu výstupu RNG, který dovede nedeterministický stroj k správnému výsledku.

Vzorová řešení 5. série

Problém 1

Zadání:
Jak najít maximum z čísel uložených na pozicích 1 až n ve sdílené paměti?

- *1 stroj v čase* $\mathcal{O}(n)$
- $\mathcal{O}(n)$ *strojů v čase* $\mathcal{O}(\log n)$
- $\mathcal{O}(n^2)$ *strojů v čase* $\mathcal{O}(1)$
- $\mathcal{O}(\sqrt{n})$ *strojů v čase* $\mathcal{O}(\sqrt{n})$
- $\mathcal{O}(n/\log n)$ *strojů v čase* $\mathcal{O}(\log n)$
- $\mathcal{O}(n^{1,5})$ *strojů v čase* $\mathcal{O}(1)$

Řešení:

- Jeden stroj prostě jenom přečte celý vstup a průběžně si pamatuje nejvyšší nalezenou hodnotu. Tohle je klasický sekvenční algoritmus.

- Tady začínáme využívat paralelizmus - $n/2$ strojů si rozdělí vstup na dvojice čísel a každý stroj vybere větší číslo ze své dvojice, čímž zkrátí délku vstupu na polovinu. Po $\log n$ opakováních zbyde jen jedno číslo.

- Více strojů nám umožňuje rychlejší výpočet. Začneme tedy používat týmy. Seženeme si n týmů po n strojích. i-tý stroj j-tého týmu zkontroluje, zda je j-tý prvek alespoň roven i-tému. Pokud ne, tak na $n+j$-tou pozici v paměti zapíše jedničku. Pokud nějaký tým zjistí, že na jejich $n+j$-té pozici se neobjevila jednička, tak jejich číslo je maximem. Tohle řešení je úžasně rychlé, ale potřebuje příliš mnoho práce.

- Pojďme se opět pokusit o lineární množství práce. Pokud použijeme $\sqrt{n}$ strojů, tak ty si můžou vstup rozdělit na $\sqrt{n}$ dlouhé úseky. Každý stroj sekvenčně najde maximum svého úseku a pak jeden z nich najde maximum z těchto mezivýsledků.

- Kam až se dokážeme dostat bez zvýšení celkové práce? $n/\log n$ strojů si může rozdělit vstup na $\log n$ dlouhé úseky, najít maximum každého úseku a pak použít postup z prvního paralelního řešení k vyřešení celého problému.

- Pro řešení v konstantním čase je potřeba více než lineární počet strojů, ale ne o moc. Můžeme si například pořídit $\sqrt{n}$ týmů, kde každý tým má $\sqrt{n}$ podtýmů po $\sqrt{n}$ strojích. (Úkol pro zvídavé: Rozmyslete si, jak tohle dál zlepšovat.)

Problém 2

Zadání:
Jak setřídit čísla uložená na pozicích 1 až n ve sdílené paměti?

- *1 stroj v čase* $\mathcal{O}(n \log n)$
- $\mathcal{O}(n)$ *strojů v čase* $\mathcal{O}(n)$
- $\mathcal{O}(n^2)$ *strojů v čase* $\mathcal{O}(\log n)$
- $\mathcal{O}(n)$ *strojů v čase* $\mathcal{O}(\log^2 n)$

Řešení:

- Tradiční sekvenční algoritmus. Příklady jsou mergesort nebo heapsort.
- Použijeme $n/2$ strojů. Můžeme si to představit tak, že každý stroj má na starosti svou sudou pozici v poli. Všechny stroje pravidelně srovnávají svou pozici se střídavě levou a pravou sousední pozicí a podle potřeby prohazují čísla směrem k uspořádanějšímu pořadí. Stroje tímto posouvají čísla směrem k jejich cílovým pozicím a střídání směrů zajišťuje, že si stroje nebudou překážet.
- Tentokrát přijdou na řadu týmy. Každý tým spočítá pozici, na kterou jeho číslo patří a pak jej tam přesune. $\mathcal{O}(\log n)$ času zabere kombinování informací od jednotlivých členů týmu.
- Tohle už je hodně náročné. Nejlépe dokumentované řešení se jmenuje bitonické třídění a buď se mě zeptejte, nebo si počkejte na magisterské studium na Matfyzu. :)

Problém 3

Zadání:
Jak rychle umíte deterministicky řešit problém 7 ze čtvrtého čísla?

- *1 stroj v čase* $\mathcal{O}(n^2)$
- $\mathcal{O}(n^2)$ *strojů v čase* $\mathcal{O}(n)$
- $\mathcal{O}(n^3)$ *strojů v čase* $\mathcal{O}(\log n)$

> **Problém 7 ze čtvrtého čísla:** *Představte si, že máte registr, který začíná na hodnotě 1, a vaším úkolem je zjistit, zda je možné jej vynulovat. Zádrhel je v tom, že registr má nějaká omezení na to, jak jej lze měnit. Dostanete číslo n na vstupu a pole obsahující $n \times n$ bitů, kde bit na pozici $i \cdot n + j$ (číslováno od nuly) určuje, zda můžete změnit hodnotu v registru z i na j, přičemž žádné jiné změny nejsou povoleny.*

Instance problému tedy může vypadat například takto:

n = 4
Obsah pole: 0000 0010 0001 1000
Pole reprezentuje následující tabulku:

		nová			
		0	1	2	3
původní	0	0	0	0	0
	1	0	0	1	0
	2	0	0	0	1
	3	1	0	0	0

Odpověď: ANO (1 -> 2 -> 3 -> 0)

Jak rychle umíte tento problém řešit? Jak rychle jej umíte řešit nedeterministicky? Pokud vám to pomůže, tak můžete předpokládat, že n je mocnina dvojky.

Řešení:

- První možnost je uvedená výše ve vzorových řešeních čtvrté série.

- Vezmeme sekvenční řešení a každému povolenému přechodu přiřadíme jeden stroj. Hned jak je stav i dosažitelný, tak pokud je přechod z i do j povolený, tak $ni + j$-tý stroj rovnou prohlásí stav j za dosažitelný. Ano, dochází ke spoustě plýtvání výkonem a kolizních zápisů, ale nevadí nám to.

- Na začátku upravíme matici tak, aby pozice i,i obsahovala jedničku pro každé i. Každý stroj se pak snaží ověřit, zda se dá dostat ze stavu i do stavu k přes stav j pro nějaké i,j,k. V každém „kole“ se tedy podívá na pozici i,j a j,k v tabulce a pokud jsou obojí jedničky, změní pozici i,k na jedničku. Na začátku prvního kola pozice i,j v matici udává, zda existuje cesta z i do j na nejvýše jeden krok. Po prvním kole matice ukazuje existenci cest na nejvýše dva kroky, po druhém na čtyři kroky, pak na osm kroků a tak dále. Po $\mathcal{O}(\log n)$ krocích máme zaručeně nalezené všechny možné cesty. Tento postup je znám jako Floyd-Warshallův algoritmus.[9]

[9] `https://cs.wikipedia.org/wiki/Floyd%C5%AFv%E2%80%93Warshall%C5%AFv_algoritmus`

Problém 4

Zadání:
Jak pomocí ALU a konstantně mnoha registrů simulovat zásobník, pásku, nebo dokonce pole bitů?

Řešení:
Nejdříve si postavíme zásobník na bity. Uložené bity budeme skladovat jako bity v binárním zápisu čísla v registru (a možná budeme chtít druhý registr pro sledování počtu vložených bitů). Vložení bitu vyžaduje zdvojnásobení obsahu registru a přičtení nového bitu. Ze dvou zásobníků se dá postavit páska a následující stroj simuluje pole. Stejně jako v dřívějších sériích předpokládáme, že se vstup mění až po naší odpovědi.

```
vstup                                                    (pozice)
vstup                                                    (bit)
registr                                                  („pole bitů“)
registr                                                  (aktuální krok)
registr                                                  (mocnina dvojky)
ALU
výstup

a,_,x,0,X,0 → x,1,X,2,a,^,
                                  (spočítáme pozici odpovídající mocninu dvojky)
_,_,x,1,X,m → x,2,m,x,m,/,
_,_,x,2,m,y → x,3,m,y,2,%,                (vytáhneme požadovanou pozici)
_,0,x,3,m,0 → x,0,X,0,0,+,0               (našli jsme nulu a nic neměníme)
_,0,x,3,m,1 → x,4,X,x,m,-,1     (našli jsme jedničku a nastavujeme nulu)
_,1,x,3,m,0 → x,4,X,x,m,+,0     (našli jsme nulu a nastavujeme jedničku)
_,1,x,3,m,1 → x,0,X,0,0,+,1               (našli jsme jedničku a nic neměníme)
_,_,_,4,_,x → x,0,X,0,0,+,                (uložení nového stavu pole)
```

Problém 5

Zadání:
Jak pomocí ALU a konstantně mnoha registrů setřídit n b-bitových čísel v čase $\mathcal{O}(n+b)$?

Řešení:
Tohle opět jedna z těžších úloh. Následující skripta obsahují alespoň základní myšlenky řešení: `http://mj.ucw.cz/vyuka/1920/ds2/02-integer.pdf` (strany 5 a 6)

Problém 6

Zadání:
Proč jsou nedeterministické algoritmy zajímavé i z pohledu realisticky silných výpočetních modelů?

Řešení:
Asi největší využití má onen náhled pomocí certifikátů – vzhledem k tomu, že najít certifikát je (domníváme se) mnohem těžší než jeho ověření, tak se související myšlenky často používají v kryptografii. Také se hodí umět poznat, že je problém těžký, a tudíž vyžaduje jiný přístup.

Problém 7

Zadání:
Jak by se dalo využít $\mathcal{O}(n \cdot n!)$ procesorů k řešení problémů 2 a 3 v čase $\mathcal{O}(1)$?

Řešení:
Hrubá myšlenka postupu:

Problém 2 (třídění): Každý tým má svou konkrétní permutaci, o které si myslí, že vstup setřídí. Stroje v týmu jen ověří, že po zpermutování by vstup byl setřízen. (Každý stroj kontroluje dvě sousední pozice výsledku.) Každý tým má vyhrazenou pozici, kam zapíše jedničku libovolný stroj, který najde nesetřízenou dvojici. Pokud v týmu žádný stroj jedničku nezapíše, daný tým může provést svou permutaci a setřídit vstup.

Problém 3 (hledání nulující posloupnosti): Každý tým má svůj tip na možnou cestu k cíli. (Rozmyslete si, že máme dost týmů pro cesty všech možných délek.) Pokud žádný stroj v týmu nenajde na týmové cestě neplatný krok, tak daný tým může vypsat, že se mu podařilo najít cestu.

Problém 8

Zadání:
Kolik procesorů potřebujete k sečtení n b-bitových čísel v čase $\mathcal{O}(1)$?

Řešení:
Přibližně $2^{n(b+\log n)} \cdot n(b + \log n)$. Celkový součet nebude delší než $\mathcal{O}(b + \log n)$ bitů. Každý tým si „tipne“ všechny mezisoučty, což jsou $\mathcal{O}(b+\log n)$-bitová čísla. Týmy se rozdělí na podtýmy, kde každý podtým ověřuje, že dva po sobě jdoucí mezisoučty se liší právě o příslušný prvek vstupu. Každý stroj pak už jen kontroluje změnu jednoho konkrétního bitu v mezisoučtu - podívá se (ač neověří), zda nižší bit přetekl a díky tomu může ověřit chovaní svého bitu. Pokud si v celém týmu žádný stroj nevšimne ničeho podezřelého, tak jejich tip je správně včetně tipnutého výsledku.

Téma 3 – Zpracování obrazových dat ze senzorů

V tomto závěrečném díle už nehledejte žádné další zadání. Postupně si projdeme úlohy ze čtvrtého a pátého dílu a ukážeme si jejich řešení. Veškerá vstupní data i zdrojový kód si můžete stáhnout na `https://mam.mff.cuni.cz/media/prilohy/26-6-3-data.zip`.

Vzorová řešení 4. dílu

Problém 1

Zadání:
Naprogramujte v libovolném programovacím jazyce (doporučuji Python) funkci, která na vstupu vezme 2D pole s naměřenými daty a na výstupu vrátí vizualizaci těchto dat v polárních souřadnicích. Vstupní data najdete ve formátu .npy pro Python (knihovna numpy) nebo ve formátu .csv pro ostatní programovací jazyky. Odevzdejte zdrojový kód kompletního programu (ne jen implementaci této funkce), který přečte vstupní data ze souboru a vizualizaci uloží jako obrázek nebo ji alespoň zobrazí na obrazovce.

Řešení:
V první úloze bylo vaším úkolem zobrazit dvojrozměrný obrázek v polárních souřadnicích. V tomto řešení jsem opět použil knihovnu Matplotlib v jazyce Python.

Pokud nemáte knihovny `numpy` a `matplotlib` nainstalované z minulých dílů, je třeba to napravit příkazem v terminálu (nebo v PowerShell konzoli na Windows v administrátorském režimu):

```
pip install numpy matplotlib
```

Implementaci vizualizace v polárních souřadnicích ukazuje funkce `ppiPlot()`.

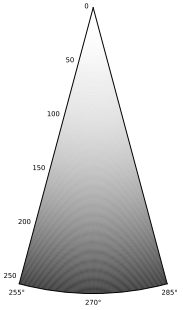
```
1  import matplotlib.pyplot as plt
2  import numpy as np
3  import math
4
5  def ppiPlot(
6      frame = None,
7      minDistance = 0, maxDistance = 256,
8      minAngle = 255, maxAngle = 285,
9      numAngleValues = 3, cmap = 'binary'
10 ):
11     """Vykresleni 2D np.matrix hodnot do grafu v polarnich
           ↪ souradnicich.
12
13
```

```
14      Args:
15          frame: np.matrix (2D pole) hodnot typu float (vstupni
                ↪ obrazek).
16          minDistance: Minimalni vzdalenost od pocatku souradnic.
17          maxDistance: Maximalni vzdalenost.
18          minAngle: Minimalni zobrazeny uhel.
19          maxAngle: Maximalni zobrazeny uhel.
20          numAngleValues: Pocet uhlu vypsanych na ose.
21          cmap: Barevna mapa, viz dokumentace matplotlib.
22
23      Returns:
24          None
25
26      """
27      # Pokud neni specifikovan obrazek, vytvorime nejaky priklad
28      if frame is None:
29          distanceValues = 256
30          angleValues = 256
31      # Jinak precteme rozmery obrazku
32      else:
33          distanceValues = frame.shape[0]
34          angleValues = frame.shape[1]
35
36      # Nastavime vlastnosti polarniho grafu
37      ax = plt.subplot(111, polar=True)
38      ax.set_thetamin(minAngle)
39      ax.set_thetamax(maxAngle)
40
41      # Hodnoty uhlu, ktere se vypisou na ose
42      plt.thetagrids(
43          np.linspace(minAngle, maxAngle, numAngleValues)
44      )
45
46      # Spocitame souradnice vsech pixelu
47      radiusSpace = np.linspace(minDistance, maxDistance,
            ↪ distanceValues)
48      azimuthSpace = np.linspace(
49          math.radians(minAngle),
50          math.radians(maxAngle),
51          angleValues
52      )
53      radius, theta = np.meshgrid(radiusSpace, azimuthSpace)
54
```

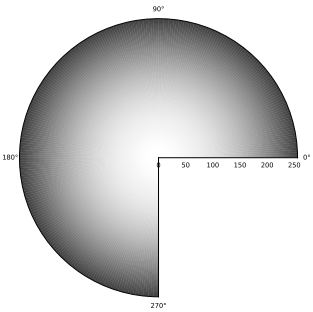
```
55      # Pokud neni specifikovan obrazek, vytvorime nejaky priklad
56      if frame is None:
57          displayedValue = radius ** 2.0
58      # Jinak vytvorime pole zobrazovanych hodnot
59      else:
60          displayedValue = frame.T
61
62      # nakonec obrazek vykreslime
63      plt.pcolormesh(theta, radius, displayedValue, cmap=cmap)
```

Funkčnost výše popsané funkce si můžeme ověřit třeba na těchto příkladech:

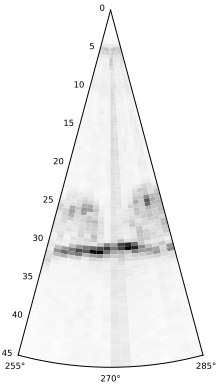
```
1  ppiPlot()
2  plt.show()
```



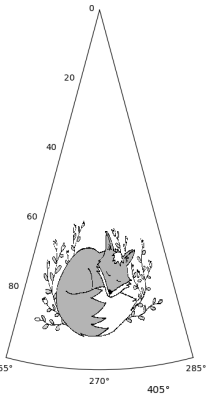

```
1  ppiPlot(
2      minAngle = 0,
3      maxAngle = 270,
4      numAngleValues = 4)
5  plt.show()
```



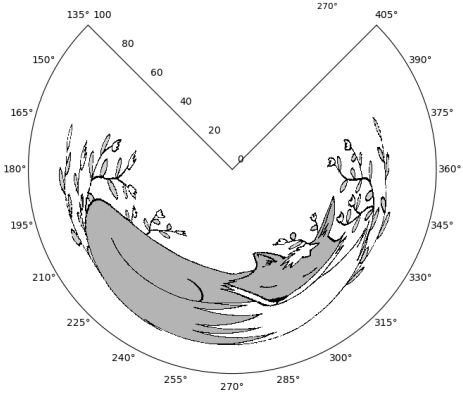

```
1  ppiPlot(
2      np.load('3_2D.npy')[:96,:],
3      0, 45, 255, 285, 3, 'binary')
4  plt.show()
```



```
# pip install imageio
import imageio
riki = imageio.imread('Riki.png')
riki2D = np.sum(riki[:,:,:2],
    ↪ axis=2)
ppiPlot(riki2D, 50, 100, 255,
    ↪ 285, 3, 'binary_r')
plt.show()
```




```
ppiPlot(
    riki2D, 25, 100,
    180-45, 360+45, 19,
    'binary_r')
plt.show()
```



Problém 2

Zadání:
Libovolným způsobem vizualizujte 3D data jako 3D graf srozumitelný pro lidské oko.

Řešení:
Ve druhé úloze jste naopak dostali za úkol vizualizovat trojrozměrný snímek libovolným způsobem. Mohli jste se vydat cestou vlastní implementace nebo zkusit najít již existující nástroj. Zde opět využiji síly knihovny `matplotlib`. Na podrobnější prozkoumání různých 3D vizualizací pomocí knihovny `matplotlib` vás odkážu na kapitolu 4.12 v této knize:

`https://jakevdp.github.io/PythonDataScienceHandbook`

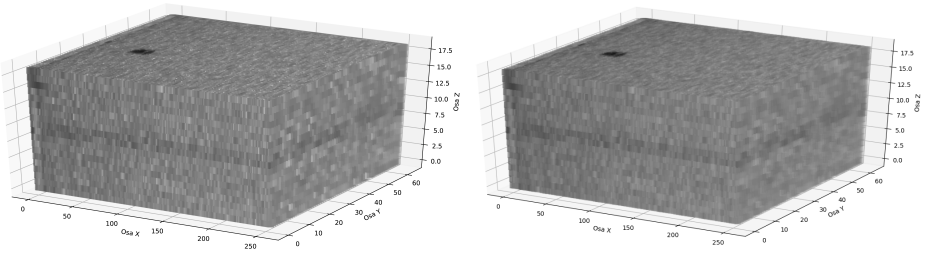
Jedna z možných implementací 3D vizualizace jako funkce `plot3D()`:

```
# Zde ziskame 3D projekci, ale import jinak zustava nepouzity
from mpl_toolkits.mplot3d import Axes3D # noqa: F401 unused import

import matplotlib.pyplot as plt
import numpy as np

```

```
7  def plot3D(image3D):
8      # Jednotliva pole s X, Y, Z souradnicemi a jejich hodnotami
9      xs, ys, zs, vs = [], [], [], []
10     for x in range(image3D.shape[0]):
11         for y in range(image3D.shape[1]):
12             for z in range(image3D.shape[2]):
13                 xs.append(x)
14                 ys.append(y)
15                 zs.append(z)
16                 vs.append(image3D[x,y,z])
17     # Prevedeme na pole knihovny numpy
18     xs = np.array(xs)
19     ys = np.array(ys)
20     zs = np.array(zs)
21
22     fig = plt.figure()
23     ax = fig.add_subplot(111, projection='3d')
24
25     # Volitelne popiseme jednotlive osy
26     ax.set_xlabel('Osa X')
27     ax.set_ylabel('Osa Y')
28     ax.set_zlabel('Osa Z')
29
30     # Zobrazime jako scatter plot
31     ax.scatter(xs, ys, zs, c=vs, marker='s', cmap='binary')
32
33 # Nacteme data ze souboru,
34 # logaritmus kvuli lepsimu rozlozeni barev
35 image3D = np.log10(np.load("3_3D.npy"))
36
37 # Nakonec funkci plot3D zavolame a vykreslime vizualizaci
38 plot3D(image3D)
39 plt.show()
```

Výsledek můžete vidět na obrázku 1a. Problém je, že takto vidíme jen obrysové plochy zobrazovaného kvádru, ale hodnoty uvnitř nám zůstávají utajeny. K hodnotám uvnitř kvádru se dostaneme pomocí průhlednosti. Řekněme, že nízkým hodnotám přiřadíme vysokou průhlednost a vysokým hodnotám naopak nízkou průhlednost. Díky tomu se nám podaří tato data vizualizovat pomocí trojrozměrné mlhy s různou hustotou v různých oblastech. (Pomocí pomyslné mlhy se mimochodem zobrazují například 3D snímky lidského těla pořízené pomocí lékařských přístrojů.)

Průhlednost do našeho grafu nyní vložíme pomocí barevné mapy, modifikujeme právě přidáním průhlednosti:



(a) Původní 3D snímek

(b) Po aplikaci lineární průhlednosti

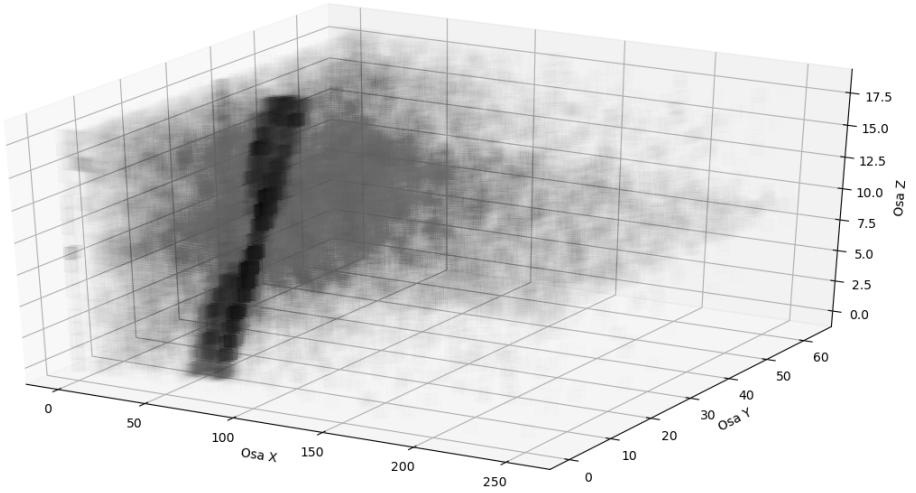
Obrázek 1

```
(...)
import numpy as np
import matplotlib.pylab as pl
from matplotlib.colors import ListedColormap

# Vybereme z podporovanych barevnych palet
cmap = pl.cm.binary
# Pouzijeme jeji barvy pro nasi novou paletu
my_cmap = cmap(np.arange(cmap.N))
# Pridame pruhlednost
my_cmap[:,-1] = np.linspace(0, 1, cmap.N)
# Vytvorime nasi novou barevnou paletu
my_cmap = ListedColormap(my_cmap)

def plot3D(image3D):
    (...)
        ax.scatter(xs, ys, zs, c=vs, marker='s', cmap=my_cmap)

# Nacteme data ze souboru,
# logaritmus kvuli lepsimu rozlozeni barev
image3D = np.log10(np.load("3_3D.npy"))
# Nakonec funkci plot3D zavolame a vykreslime vizualizaci
plot3D(image3D)
plt.show()
```

Výsledek můžete vidět na obrázku 1b. Moc jsme si nepomohli, vyrenderovaná mlha je příliš hustá a kvůli tomu vidíme skrz jen několik prvních voxelů. Problém vyřešíme například nelineárním rozdělením průhlednosti. Malou průhlednost chceme přiřadit jen opravdu významným voxelům, aby nám nezakryly jiné vzdálenější významnější voxely. Lineární rozdělení průhlednosti $\alpha = v$ nyní změníme na $\alpha = v^{10}$, kde $v \in (0,1)$:

```
(...)
# Pridame pruhlednost
my_cmap[:,-1] = np.linspace(0, 1, cmap.N)**10
(...)
```

Nyní už v datech začínáme vidět smysluplný objekt – vizte obrázek 2.



Obrázek 2: Po aplikaci nelineární průhlednosti x^{10}

Bonus:
Jako bonus jste se měli zamyslet nad tím, proč jsou data v 3D prostoru rozložena zrovna tímto způsobem. Navíc jste měli z předchozích částí informaci, že třetí rozměr odpovídá odražené frekvenci. Tato odražená frekvence koresponduje díky Dopplerově efektu s radiální rychlostí zobrazovaného předmětu vůči radaru. Uprostřed obrázku je zachycena cesta, která je z obou stran lemována stromy. Radar se v danou chvíli pohyboval konstantní rychlostí nad terénem, což znamená, že předměty před radarem se k němu přibližovaly a naopak předměty za radarem se vzdalovaly. Když se radar nacházel přímo nad cestou a pořídil tento snímek, tak jeden strom byl akorát před radarem a druhý strom už za radarem. Proto v naměřených datech vidíme různou rychlost obou stromů.

To neznamená, že se tyto stromy vůči sobě pohybovaly. Znamená to jen, že jeden strom se k radaru přibližoval a druhý vzdaloval.

Problém 3

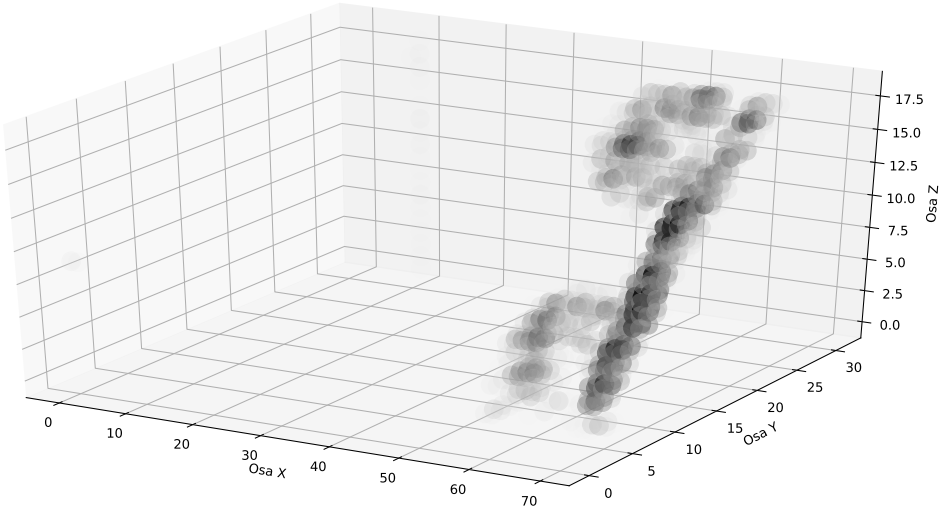
Zadání:
Zkuste odfiltrovat šum z 3D snímku ještě před tím, než ho začnete převádět na 2D obrázek. Měli byste tak dosáhnout lepšího výsledku než v předchozím díle.

Řešení:

Ještě lepšího výsledku dosáhneme odfiltrováním šumu. Jako šum prohlásíme zanedbatelně nízké naměřené hodnoty. Naši vykreslovací funkci upravíme tak, aby ignorovala všechny voxely s hodnotou menší než jedna:

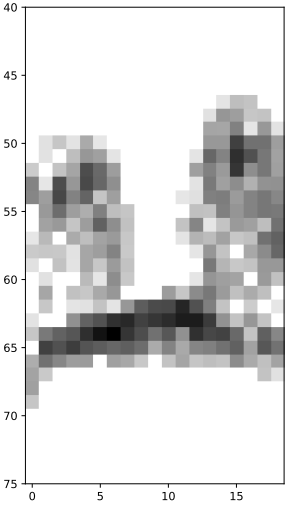
```
(...)
# Nyni opet staci linearni rozdeleni pruhlednosti
my_cmap[:,-1] = np.linspace(0, 1, cmap.N)
(...)
def plot3D(image3D):
    xs, ys, zs, vs = [], [], [], []
    for x in range(image3D.shape[0]):
        for y in range(image3D.shape[1]):
            for z in range(image3D.shape[2]):
                # Data filtrujeme timto IFem
                if image3D[x,y,z] > 1.0:
                    xs.append(x)
                    ys.append(y)
                    zs.append(z)
                    vs.append(image3D[x,y,z])
    (...)
(...)
```

Výsledek vypadá zase o něco lépe, jak ukazuje obrázek 3.

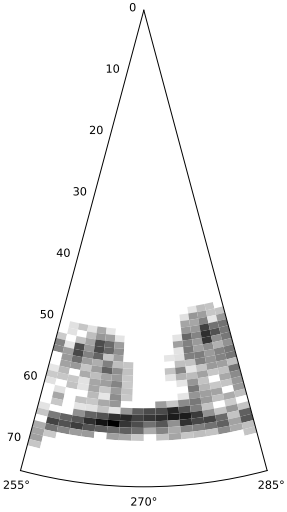


Obrázek 3: Po aplikaci threshold filtru

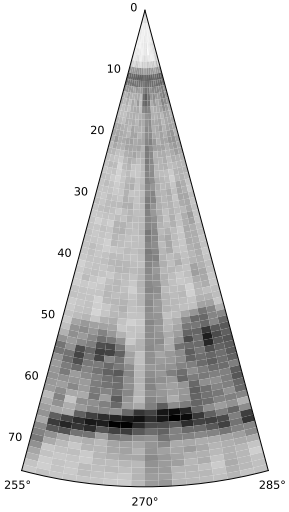
Možná si pamatujete druhý díl tohoto tématka, kde bylo vaším úkolem odstranit šum z dvojrozměrného obrázku. Odstranění šumu tehdy nebylo zas tak snadné, přitom tady v originálních datech nám bohatě stačil obyčejný threshold filtr. Po odstranění šumu hned ve 3D snímku získáme 2D snímek, který vidíte na obrázku 4.



Filtrovaný v kartézských souřadnicích



Filtrovaný v polárních souřadnicích



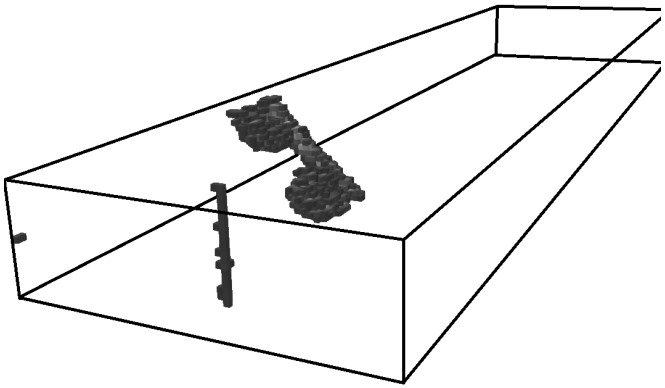
Originál v polárních souřadnicích

Obrázek 4

Poznámka:
K renderování trojrozměrných dat můžete použít robustnější nástroje, které jsou pro tyto účely přímo stvořené. Zde vám ukážu příklad vyexportování dat do programu ParaView a následné vyrenderování:

```
#pip3 install pyevtk
import numpy as np
import os
from pyevtk.hl import gridToVTK, VtkGroup

image3D = np.log10(np.load("3_3D.npy"))
x = np.arange(image3D.shape[0]+1)
y = np.arange(image3D.shape[1]+1)
z = np.arange(image3D.shape[2]+1)
filepath = os.path.join('.', '3_3D.vtr')
gridToVTK(filepath, x, y, z, cellData={'data':image3D.copy()})
# Vyrenderovat soubor '3_3D.vtr' v ParaView
```

Výsledek vidíme na obrázku 5



Obrázek 5: Vyrenderovaný výsledek pomocí nástroje ParaView

Problém 4

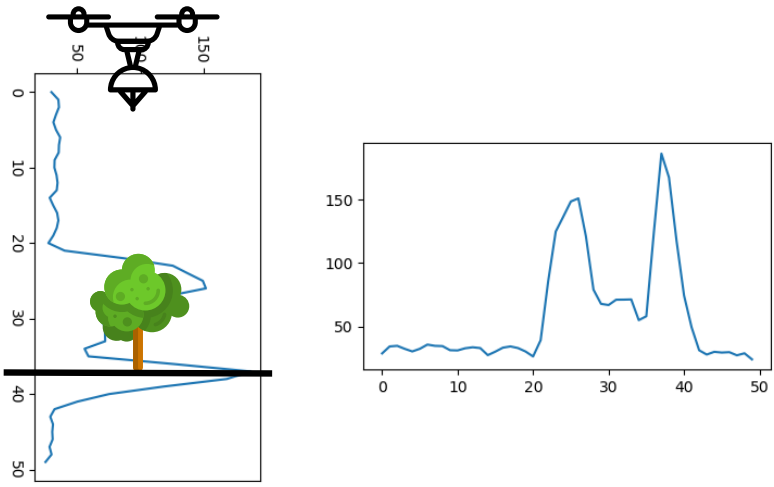
Zadání:
Vraťme se zpátky k prvnímu dílu, kde jsme naměřili jednorozměrný seznam hodnot. Tento seznam reprezentoval intenzitu odraženého signálu s narůstající vzdáleností od senzoru. Pokud se například 20 m *od senzoru nachází jediná překážka, tak seznam naměřených hodnot bude obsahovat nízké hodnoty (šum) pro všechny vzdálenosti menší než 20 m. Naopak hodnoty reprezentující vzdálenost 20 m budou výrazně vyšší, neboť z této vzdálenosti dorazil signál odražený od překážky. Pro upřesnění připomenu obrázkem 6 z prvního dílu.*

Zdrojová data k tomuto obrázku jsou uložena jako textový soubor, ve kterém je každá naměřená hodnota na samostatném řádku. Obsah souboru můžete přímo kopírovat do Excelu, Rka či jiného nástroje.

Vaším úkolem je pro daný vstup najít dvě čísla reprezentující vzdálenost (souřadnice X). První číslo reprezentuje vzdálenost nejbližší a druhé číslo nejvzdálenější naměřené překážky. Například pro obrázek výše je vzdálenost nejbližší překážky někde v intervalu 20 m – 25 m *a nejvzdálenější někde v intervalu* 38 m – 42 m.

Řešení:
U tohoto problému stačilo vrátit nejmenší a největší indexy, na kterých byla hodnota větší než zvolená konstanta. Jak říká zadání, je celkem jedno, jestli jste k řešení této úlohy použili Excel nebo skript v libovolném jazyce.

Volba této konstanty bývá součástí kalibrace celého senzoru. Dynamická kalibrace v každém snímku není dobrý nápad hlavně v případě, kdy senzor nevidí žádné předměty. V tom případě se nemá podle čeho nakalibrovat a může snadno začít vnímat vyšší hodnoty šumu jako reálné předměty.



Obrázek 6: Význam naměřených dat

Kalibrace tohoto senzoru závisí na změnách atmosférických podmínek. Tato problematika je ale vhodná spíš na fyzikální konferu na soustředění než na tématko. Takže tady se kalibrací nebudeme více zabývat.

Obrázek 7 ukazuje jedno z možných řešení tohoto problému pomocí Excelu. Sloupec B obsahuje jen vzestupné číslování, tedy indexy indikující vzdálenost. Sloupec C obsahuje naměřená data a sloupec D vznikl pomocí této formule: `=IF(C1>$H$1, B1, 0)`

Díky tomu sloupec D obsahuje všechny indexy vzdáleností, na kterých byla detekována překážka, neboli naměřená hodnota je větší než zvolený threshold. Nyní z tohoto sloupce stačí vybrat minimum a maximum, pokud budeme ignorovat nulové hodnoty.

Minimum: `=MIN(IF(D15:D256<>0,D15:D256))`

Maximum: `=MAX(D15:D256)`

Vzorová řešení 5. dílu

Doposud jsme měli příležitost pracovat s obrazovými daty, která nebyla pořízena přímo fotoaparátem. Zjistili jsme, že třetí (čtvrtá, ...) dimenze nemusí reprezentovat jen čas, ale třeba barvu, frekvenci, rychlost, teplotu nebo jakoukoliv jinou veličinu. Vyzkoušeli jsme si vizualizaci vícerozměrných obrázků a jejich analýzu (hlavně za účelem odstraňování šumu). Nyní se vrátíme zpět ke klasickým 2D obrázkům z fotoaparátu a zkusíme na nich využít dovednosti získané z minulých dílů tohoto tématka.

B	C	D	E	F	G	H
46	39,0377971	0				
47	31,87381376	0				
48	33,02532891	0				
49	43,4390459	0				
50	53,26685745	0			Threshold	100
51	82,19119722	0			**Minimum**	**52**
52	105,3705606	52			**Maximum**	**66**
53	164,8365802	53				
54	205,3800453	54				
55	187,3589776	55				

Obrázek 7: Tabulka z Excelu s řešením problému 4

Problém 1

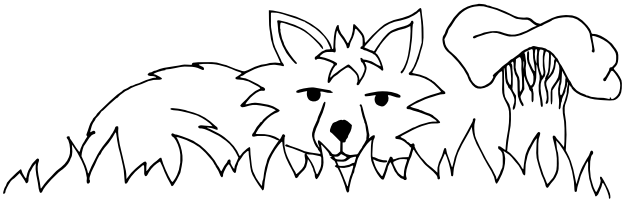
Zadání:

Vytvořte program (nebo najděte existující nástroj), který dostane zadaný vstupní obrázek v podobě vybarvených dětských omalovánek. Váš program tento obrázek rozdělí na regiony podle barvy. Pro každý nalezený region spočítá jeho velikost a obvod v pixelech. Region je v tomto případě ohraničená oblast vybarvená stejnou barvou. Testovací obrázek omalovánek Rikiho najdete na konci v datech k tomuto dílu pod názvem „Riki1.png“ (zde obrázek 8).

Obrázek 8: Vstupní obrázek, ve kterém jste měli najít všechny spojité regiony stejné barvy

Řešení:
Cílem tohoto problému bylo seznámit vás se strojovým vybarvováním obrázků. Regiony, které máme vybarvit jednou barvou, můžeme najít pomocí prohledávání do šířky. Prohledávání do šířky pracuje s frontou, do které si ukládáme všechny nově objevené pixely. Jako první krok si vybereme pixel, od kterého chceme začít prohledávat. Tento pixel je automaticky součástí hledané plochy. Pokud chceme najít oblast se stejnou barvou, uložíme si do fronty všechny sousedy tohoto prvního pixelu, které mají stejnou barvu. Potom vybereme další pixel z fronty a opět provedeme totéž. Musíme si ale pamatovat všechny už navštívené pixely, protože jinak se zacyklíme. Až se naše fronta vyprázdní, tak jsme s hledáním hotovi. Neexistuje už totiž žádný nenavštívený pixel, který by měl stejně vybarveného souseda.

Tímto způsobem můžeme postupně prohledat celý obrázek a rozdělit ho tak na regiony stejných barev.



```
1  import matplotlib.pyplot as plt
2  import numpy as np
3  from queue import *
4  import imageio
5
6  # Funkce vrati vsechny sousedy daneho pixelu
7  def neighbors(x, y, maxX, maxY):
8      output = []
9      if x > 0:
10         output.append((x-1,y))
11     if y > 0:
12         output.append((x,y-1))
13     if x < maxX-1:
14         output.append((x+1,y))
15     if y < maxY-1:
16         output.append((x,y+1))
17     # Volitelne pridame i rohove sousedy
18     if x > 0 and y > 0:
19         output.append((x-1,y-1))
20     if x > 0 and y < maxY-1:
21         output.append((x-1,y+1))
22     if x < maxX-1 and y > 0:
```

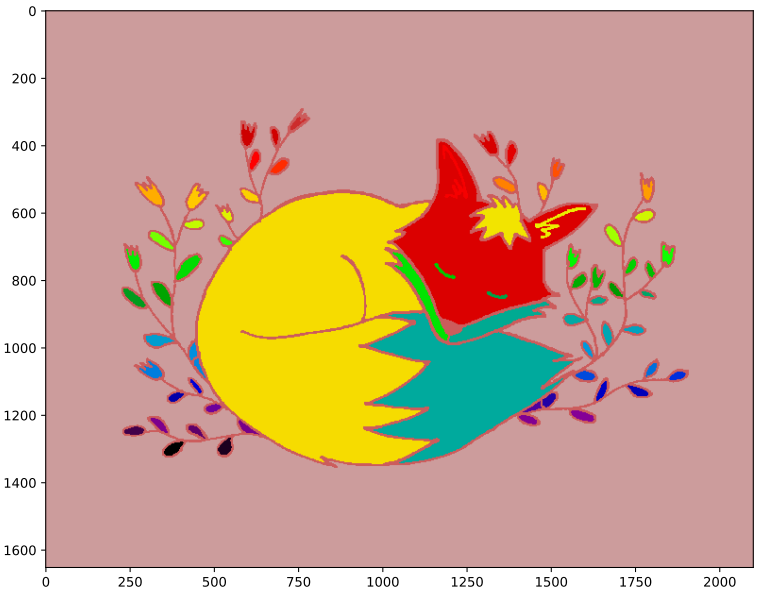
```
23          output.append((x+1,y-1))
24      if x < maxX-1 and y < maxY-1:
25          output.append((x+1,y+1))
26      return output
27
28  # Vrati seznam vsech sousedicich bodu stejne barvy
29  def findArea(startX, startY, img, mask, areaIndex=1):
30      area = []
31      q = Queue()
32      q.put((startX,startY))
33      while not q.empty():
34          x,y = q.get()
35          area.append((x,y))
36          for i,j in neighbors(x,y,img.shape[0],img.shape[1]):
37              if mask[i,j] == 0 and (img[x,y] == img[i,j]).all():
38                  q.put((i,j))
39                  mask[i,j] = areaIndex
40      return area
41
42  # Najde v dane plose vsechny okrajove pixely
43  def countEdgePixels(mask, area):
44      edgePixels = 0
45      for (x,y) in area:
46          isEdge = True
47          nei = neighbors(x,y,mask.shape[0],mask.shape[1])
48          if len(nei) != 8:
49              isEdge = False
50          else:
51              for (i,j) in nei:
52                  if mask[i,j] != mask[x,y]:
53                      isEdge = False
54              if isEdge:
55                  edgePixels += 1
56      return edgePixels
57
58  # Naceteme obrazek ze souboru
59  image = imageio.imread('Riki1.png')
60
61  # Zajimaji nas pouze kanaly RGB, kanal alpha vynechame
62  image = image[:,:,:3]
63
64  # Maska, kde si ulozime navstivene body
65  mask = np.zeros(image.shape[:2])
```

```
print("Pocitam...")
counter = 0
regions = []
for x in range(mask.shape[0]):
    for y in range(mask.shape[1]):
        if mask[x,y] == 0:
            counter += 1
            area = findArea(x,y,image,mask,counter)
            regions.append(
                (len(area), countEdgePixels(mask, area))
            )
regions.sort(reverse = True)
print(regions)
print(len(regions))

# Zobrazime vysledek, kde kazdy region ma vlastni unikatni cislo
plt.imshow(mask, cmap='nipy_spectral_r')
plt.show()
```

Díky tomuto algoritmu můžeme zadaný obrázek Rikiho rozdělit na 73 regionů, které můžete vidět zde seřazené podle velikosti od největšího po nejmenší. První číslo udává velikost regionu v pixelech a druhé číslo jeho obvod v pixelech (bez okraje celého obrázku). Mimochodem, všimli jste si i dvou schovaných regionů velikosti 1 pixel na souřadnicích $(694, 400)$ a $(1537, 1087)$?

```
[(2572994, 2540939), (385264, 378934), (166784, 115032),
(127740, 124185), (119994, 115042), (10381, 9515),
(9827, 8942), (7208, 6301), (3017, 2184), (2638, 1923),
(2420, 2133), (2003, 1650), (1909, 1669), (1897, 1658),
(1682, 1401), (1655, 1415), (1619, 1313), (1533, 1296),
(1513, 1234), (1510, 1253), (1462, 1255), (1409, 1212),
(1368, 1183), (1328, 1067), (1278, 1080), (1271, 1076),
(1137, 970), (1114, 937), (1104, 943), (1098, 870),
(1096, 923), (1094, 921), (1093, 908), (1082, 919),
(1080, 857), (1035, 814), (1026, 862), (985, 817),
(975, 806), (960, 770), (941, 761), (936, 785), (935, 780),
(932, 771), (911, 754), (897, 688), (868, 658), (862, 714),
(849, 691), (782, 642), (777, 619), (777, 606), (774, 628),
(769, 628), (745, 556), (739, 592), (738, 599), (727, 537),
(726, 575), (726, 505), (671, 490), (624, 492), (617, 492),
(616, 436), (610, 465), (607, 436), (593, 448), (567, 439),
(525, 406), (486, 341), (359, 251), (1, 0), (1, 0)]
```

Jako bonus přidávám upravený obrázek 9, kde každý region má přiřazenou trošku jinou barvu.



Obrázek 9: Stejný obrázek, ve kterém je každý nalezený region vybarvený náhodnou unikátní barvou

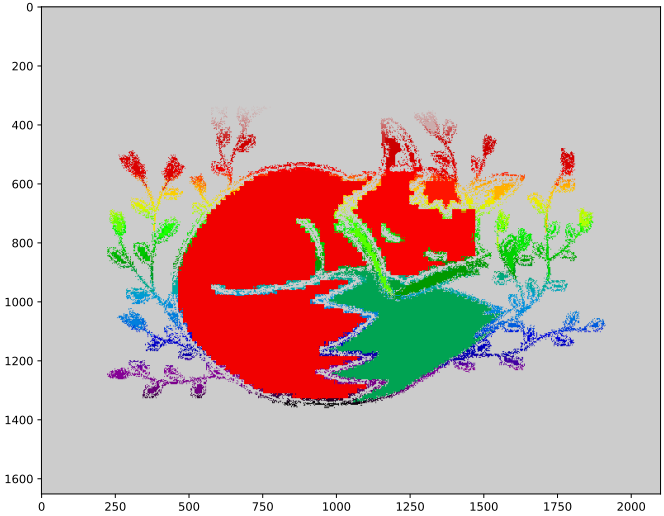
Poznámka:
Možná jste byli při práci s Pythonem lehce znepokojeni pomalejším výpočtem než u stejného algoritmu, který je zapsaný v jazyce C/C++. Kód v jazyce Python je možné optimalizovat tak, aby běžel srovnatelně rychle, ale bez předchozí kompilace kódu přímo pro daný procesor je prakticky nemožné dosáhnout stejného výkonu. Tomuto problému se v tématku věnovat nebudeme. Zdrojový kód v těchto ukázkách je optimalizován pro přehlednost, nikoliv pro výkon.

Problém 2

Zadání:
Upravte předchozí program tak, aby dokázal pracovat se stejným obrázkem, který ale prošel JPG kompresí. Testovací obrázek najdete v datech k tomuto dílu pod názvem „Riki2.jpg“.

Řešení:
Pokud stejný algoritmus aplikujeme na tento zdánlivě stejný obrázek, tak zjistíme, že počet nalezených regionů je výrazně vyšší. Formát JPG bohužel používá ztrátovou kompresi, což v praxi znamená, že během ukládání může na obrázku některé pixely drobně pozměnit. Drobně pozměnit znamená, že změna by neměla být viditelná lidským okem. Z praxe už pravděpodobně víte, že tyto artefakty v JPG obrázcích občas viditelné jsou.

Stejný algoritmus najde v obrázku `Riki2.jpg` (obrázek 10) celkem 228639 regionů, z nichž (jen) 43818 má velikost větší než 1 pixel.



Obrázek 10: Takto se první algoritmus rozbije na obrázku, který prošel JPG kompresí

Zkusme nyní změnit předchozí algoritmus tak, aby toleroval podobné barvy až do určité, předem dané hranice. Řekněme, že u každé barevné složky budeme tolerovat 1/6 spektra, toleranci tedy nastavíme na $256 \cdot 3/6 = 128$.

```
(...)
# Vrati seznam vsech sousedicich bodu stejne barvy
def findArea(startX, startY, img, mask, areaIndex=1):
    area = []
    q = Queue()
    q.put((startX,startY))
    sumColor = [0,0,0]
    while not q.empty():
        x,y = q.get()
        area.append((x,y))
        sumColor += img[x,y,:]
        for i,j in neighbors(x,y,img.shape[0],img.shape[1]):
            # sumColor/len(area) dava prumernou barvu
            # v hledanem regionu
            if mask[i,j] == 0 and sum(abs(sumColor/len(area) -
                ↪ img[i,j])) < 128:
                q.put((i,j))
                mask[i,j] = areaIndex
    return area
```

```
19
20 (...)
21 # Naceteme obrazek ze souboru
22 image = imageio.imread('data5/Riki2.jpg')
23 (...)
24 # Zobrazime vysledek, kde kazdy region ma vlastni unikatni cislo
25 plt.imshow(mask, cmap='nipy_spectral_r')
26 plt.show()
```

Všimněte si, že algoritmus porovnává aktuální pixel s průměrnou barvou v celém regionu. Kdyby porovnával s náhodným pixelem uvnitř regionu jako v minulém příkladu, mohlo by se stát, že zrovna tento zvolený pixel bude nabývat nějaké sporné hodnoty na hranici mezi dvěma regiony.

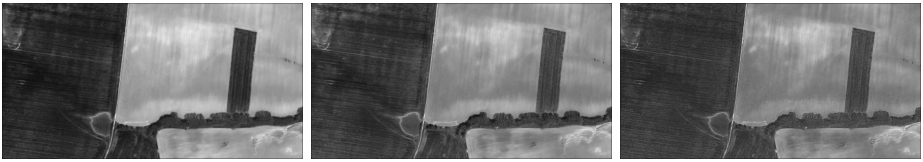
Nyní už upravený algoritmus našel jen 73 regionů, což je stejně jako v předchozím příkladu. Dokonce i velikosti a tvary regionů jsou úplně stejné.

Problém 3

Zadání:
Upravte předchozí program tak, aby dokázal podle barvy rozpoznat hnědé pole bez úrody a zelené pole s rostlinami. Váš program opět spočítá celkovou plochu, kterou zabírají zelená pole s rostlinami respektive hnědá pole s holou hlínou. Obvod počítat nebudeme hlavně z toho důvodu, že hranice mají fraktálovité vlastnosti, které nám výpočet obvodu výrazně zkomplikují. Testovací obrázek najdete v datech k tomuto dílu pod názvem „pole.jpg“ (zde obrázek 11).

Obrázek 11: Letecký snímek zeleného a hnědého pole, které má váš algoritmus rozlišit

Řešení:
V předchozím obrázku jsme nemuseli řešit problémy spojené s různým osvětlením povrchu. Nyní bychom chtěli všechny odstíny zelené vnímat jako jednu a všechny odstíny hnědé jako druhou barvu.



(R) (G) (B)

Obrázek 12: Barevný režim RGB

(H) (S) (V)

Obrázek 13: Barevný režim HSV

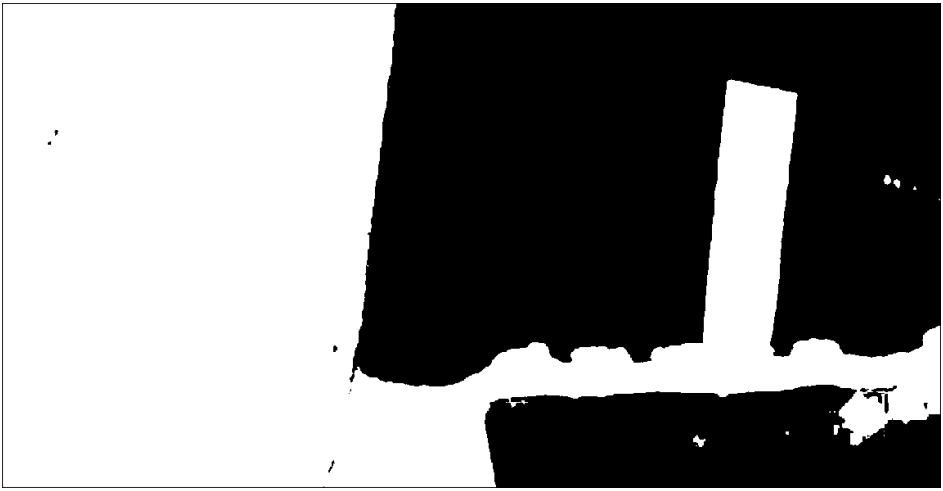
K tomu se nám bude hodit převod obrázku z barevného režimu RGB do HSV. Režim RGB reprezentuje barvy pomocí kombinace základních barev červené, zelené a modré. Režim HSV reprezentuje barvu pomocí odstínu (Hue) podobně jako v duze, sytosti (Saturation) od šedé do plně syté barvy a jasu (Value) od černé až po plně rozsvícené barvy. Jednotlivé kanály v obou režimech zobrazují obrázky 12 a 13.

Konkrétně u tohoto obrázku není rozdíl mezi jednotlivými barevnými kanály až tak zásadní a hledání zelených a hnědých regionů by nám pravděpodobně fungovalo u každého z těchto obrázků. My si pro další zpracování vybereme kanál H z režimu HSV.

```
1  import matplotlib.pyplot as plt
2  import matplotlib.colors
3  import numpy as np
4  import imageio
5  import colorsys
6
7  image = imageio.imread('pole.jpg')
8  image = matplotlib.colors.rgb_to_hsv(image)
9  image = image[:,:,0] # pouze kanal H
10
11 # Threshold zvolen v procentech tak,
12 # aby vohovoval zadani zelena vs. hneda barva
13 image[image < 0.25] = 0
14 image[image != 0] = 1
15
```

```
16 plt.imshow(image, cmap='binary_r')
17 plt.show()
```

Výsledek je znázorněn na obrázku 14.

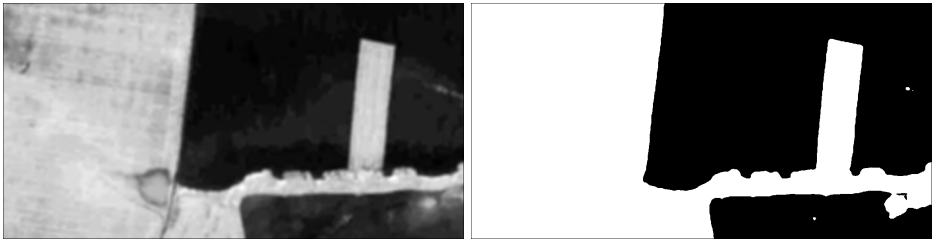


Obrázek 14: Rozlišení barev polí pomocí barevného režimu HSV a threshold filtru

Tímto způsobem jsme zjistili, že zelená plocha zabírá 832225 pixelů a hnědá 779334 pixelů. Možná namítnete, že plochy zanedbatelné velikosti nechceme počítat, neboť jsou to často jen drobné mezery ve vegetaci, cesty nebo větve dřevin. Podíváme se ještě na jednu úpravu, která zohledňuje i nejbližší okolí právě prohlíženého pixelu.

Řekněme, že když bude průměrná barva v okolí daného pixelu podobná (bude se lišit maximálně o předem danou konstantu), uznáme tento pixel jako součást aktuálního regionu. Volba velikosti tohoto okolí nám způsobí možnost zvolit si úroveň detailů, která nás zajímá. V této ukázce budeme používat okolí velikosti 16 pixelů. To znamená, že zarostlé (zelené) oblasti menší než 16×16 pixelů budeme zanedbávat.

```
1  (...)
2  radius = 16
3  outImg = np.zeros((image.shape[0]-radius, image.shape[1]-radius))
4  for x in range(image.shape[0]-radius):
5      for y in range(image.shape[1]-radius):
6          temp = image[x:x+radius, y:y+radius].reshape(1,-1)[0]
7          outImg[x,y] = sum(temp)/len(temp)
8
9  plt.imshow(outImg, cmap='binary_r')
10 plt.show()
```



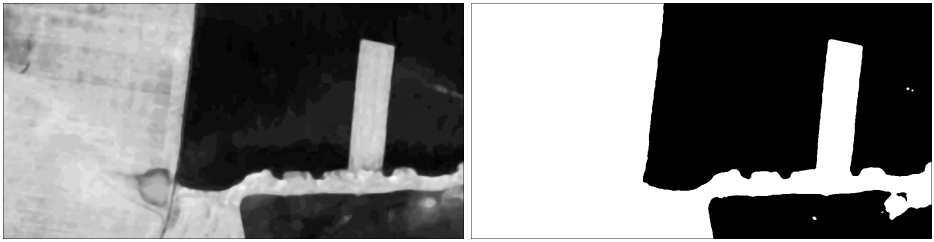
(a) Před aplikováním thresholdu **(b)** Po aplikování thresholdu

Obrázek 15: Threshold filtr aplikovaný na průměrnou barvu v okolí daného pixelu

Obrázek 15a na nás působí trošku rozmazaně. Když bereme v úvahu průměrnou barvu z okolí 16 pixelů, tak tato operace je opravdu ekvivalentní operaci rozmazání. Obecně se lineárním operacím nad okolím daného pixelu říká konvoluce.

O něco lepší je vzít místo průměrné hodnoty medián. Medián je statistická veličina, která není tolik náchylná na změny jako již zmíněný aritmetický průměr. Na druhou stranu počítání mediánu je pro procesor výpočetně náročnější. Výměna průměru za medián ve zdrojovém kódu vypadá takto:

```
1 (...)
2 radius = 16
3 outImg = np.zeros((image.shape[0]-radius, image.shape[1]-radius))
4 for x in range(image.shape[0]-radius):
5     for y in range(image.shape[1]-radius):
6         temp = image[x:x+radius, y:y+radius].reshape(1,-1)[0]
7         temp.sort()
8         outImg[x,y] = temp[len(temp)//2]
9 (...)
```



(a) Před aplikováním thresholdu **(b)** Po aplikování thresholdu

Obrázek 16: Threshold filtr aplikovaný na medián barev v okolí daného pixelu

Když se na 16 podíváme z dálky, možná budeme mít pocit, že se jedná o stejné snímky jako v případě průměru. Při pohledu zblízka ale zjistíme, že obrázek už se nejeví tak rozmazaně a jednotlivé regiony nejsou tolik zatížené šumem.

Problém 4

Zadání:
Napište program, který na výše uvedeném obrázku co nejpřesněji rozliší les od louky. Zamyslete se nad tím, jaké vlastnosti můžeme u tohoto obrázku využít. Váš program opět vypočítá celkovou plochu v pixelech, kterou zabírá les respektive louka. Obvod opět ze stejných důvodů počítat nebudeme. Testovací obrázek najdete v datech k tomuto dílu pod názvem "lesLouka.jpg"(obrázek 17).

Obrázek 17: Obrázek ze zadání, u kterého má váš program rozlišit les od louky

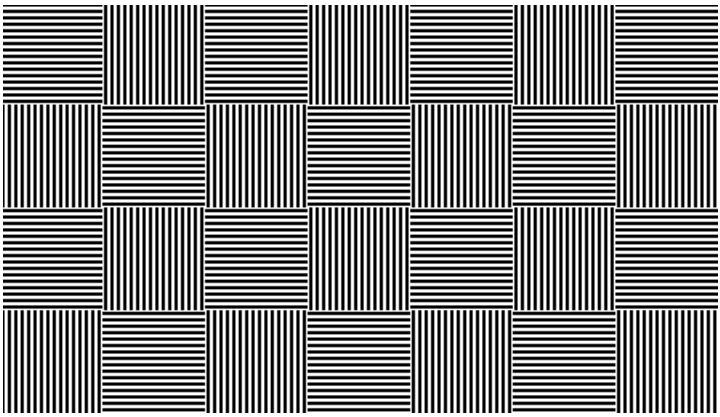
Řešení:
Zde se dostáváme k vrcholu a podle mého názoru k nejsložitějšímu příkladu celého tématka. Snažíme se rozlišit regiony velmi podobné nebo dokonce stejné barvy. Přesto se tyto regiony nějak liší, neboť lidské oko je hravě rozpozná.

Jednotlivé regiony se mohou lišit i jinak než barvou. Podívejme se třeba na obrázek 18.

Taky v obrázku vidíte šachovnici? Zkuste se na tento obrázek podívat z větší dálky a šachovnice se pravděpodobně promění v šedou plochu. [10] U většiny obrázků hodně záleží na tom, jakou míru detailů máme k dispozici, tedy z jaké dálky se na daný obrázek díváme. To nám napovídá, že pro hlubší analýzu nějakého bodu budeme muset brát v úvahu i jeho okolí.

Když známe okolí daného bodu, tak kromě průměrné nebo mediánové barvy můžeme sledovat i celou škálu dalších statistických veličin. Například v případě

[10] Při prohlížení tohoto obrázku v elektronické podobě se může stát, že se v něm objeví artefakty způsobené různým rozlišením obrázku a vašeho monitoru. Tyto artefakty bohužel pokazí zde rozebíraný efekt. Pokud artefakty vidíte, zkuste si dokument zobrazit v měřítku 1:1. Některé, převážně mobilní operační systémy, tuto volbu nemusí podporovat.



Obrázek 18: Šachovnice

rozlišování mezi lesem a loukou se nám bude hodit veličina zvaná rozptyl (variance). Bude nás zajímat, jak moc se jednotlivé barvy v okolí vybraného pixelu liší. Při pohledu na obrázek můžeme pozorovat, že na louce vidíme výrazně menší změny v barvě nejbližšího okolí než v lese.

Přesně tohoto jevu využijeme při rozlišování mezi zelenou loukou a zeleným lesem. Abych ukázal, že se zde nemusíme omezovat jen na běžné dobře známé statistické veličiny, budu zde používat vlastní statistickou veličinu podobnou rozptylu. S klasickým rozptylem vám níže ukázaný algoritmus bude fungovat taky.

Vezmeme všechny pixely v okolí vybraného pixelu a zjistíme jejich nejnižší a nejvyšší hodnotu. Taková analýza bude bohužel velmi náchylná na extrémy a na pixely s chybnou hodnotou (šum). Proto si všechny hodnoty z okolí vybraného pixelu seřadíme podle velikosti (u vícebarevných systémů musíme definovat vlastní porovnávací funkci). Následně nás bude zajímat rozdíl mezi hodnotou v první a v poslední osmině. Takto „uřízneme“ extrémní hodnoty na obou stranách a předpokládáme, že šumem zatížených pixelů bude méně než 1/8. Pomocí tohoto rozdílu nakonec rozhodneme, jestli je daný region spíše les nebo louka.

Možná vás napadne, že pokud tuto technologii zdokonalíme a budeme sledovat výrazně více statistických veličin, tak se nám podaří také výrazně přesněji rozlišovat v obrázcích regiony s různými vlastnostmi. V tomto bodě začíná celá věda o zpracování obrazu a počítačovém vidění.

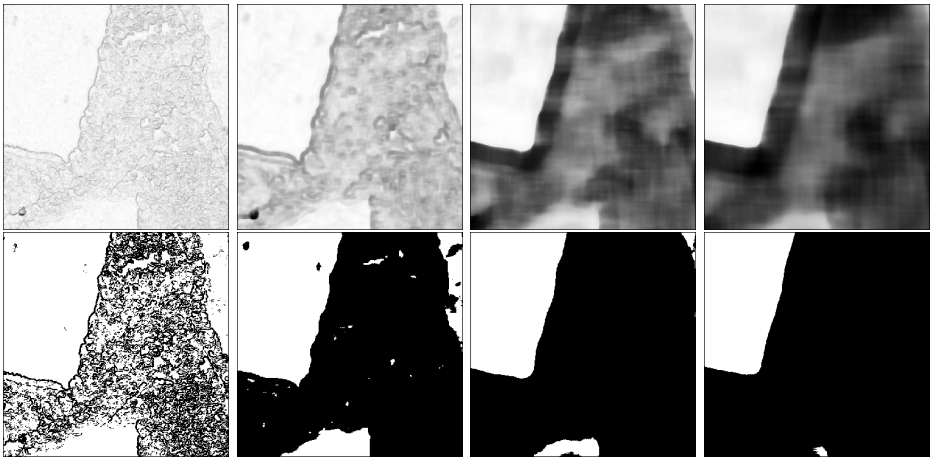
Algoritmus pro rozpoznání zeleného lesa od zelené louky v ortofoto snímku:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import imageio
4
5 # Zvolena velikost okoli bodu
6 neighbourhood = 32
```

```
7
8  # Spocitame statistickou velicinu pro vsechny pixely
9  def regions(sizeX, sizeY, image):
10     imageSizeX, imageSizeY = image.shape
11     output = []
12     for x in range(imageSizeX-sizeX):
13         line = []
14         for y in range(imageSizeY-sizeY):
15             # Indexy okraju aktualniho okoli
16             left, top, right, bottom = x, y, x+sizeX, y+sizeY
17             # Serazene pole hodnot pixelu v okoli
18             values = np.sort(
19                 image[x:x+sizeX, y:y+sizeY].reshape(1, -1)[0]
20             )
21             # Vypocet statisticke veliciny pro dany pixel
22             diff = abs(values[len(values)-len(values)//8] \
23                     - values[len(values)//8])
24             # Ulozeni vysledku do noveho obrazku
25             line.append(diff)
26         output.append(line)
27     # Vratime vysledky jako 2D pole (novy obrazek)
28     return output
29
30 # Nacteme obrazek ze souboru a pro jednoduchost
31 # secteme jednotlive RGB barvy do jedne slozky.
32 # Tim obrazek prevedeme na stupne sedi.
33 image = np.sum(imageio.imread('lesLouka.jpg'), axis=2)
34
35 # Vyhledame regiony pomoci zmineneho algoritmu
36 computedRegions = np.array(
37     regions(neighbourhood, neighbourhood, image)
38 )
39
40 # Vsechny pixely s hodnotou <50 oznacime jako louku
41 computedRegions[computedRegions < 50] = 0
42 # Ostatni jako les
43 computedRegions[computedRegions > 0] = 1
44
45 # Zobrazime vysledek
46 plt.imshow(computedRegions, cmap='binary')
47 plt.show()
48
49
```

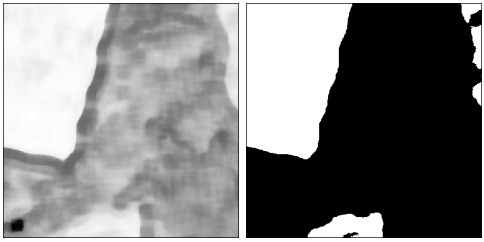
```
50 # Spocitame plochu lesa a louky
51 c0, c1 = 0, 0
52 for i in range(computedRegions.shape[0]):
53     for j in range(computedRegions.shape[1]):
54         if computedRegions[i,j] == 0:
55             c0 += 1
56         else:
57             c1 += 1
58 print("Louka:␣", c0)
59 print("Les:␣", c1)
```

Výsledek pro velikost okolí pixelu = 32 vidíme na obrázku 20. Pro zajímavost přidávám i výsledek pro jiné velikosti okolí pixelu, je znázorněn obrázkem 19.



(a) Okolí = 03 **(b)** Okolí = 16 **(c)** Okolí = 50 **(d)** Okolí = 75

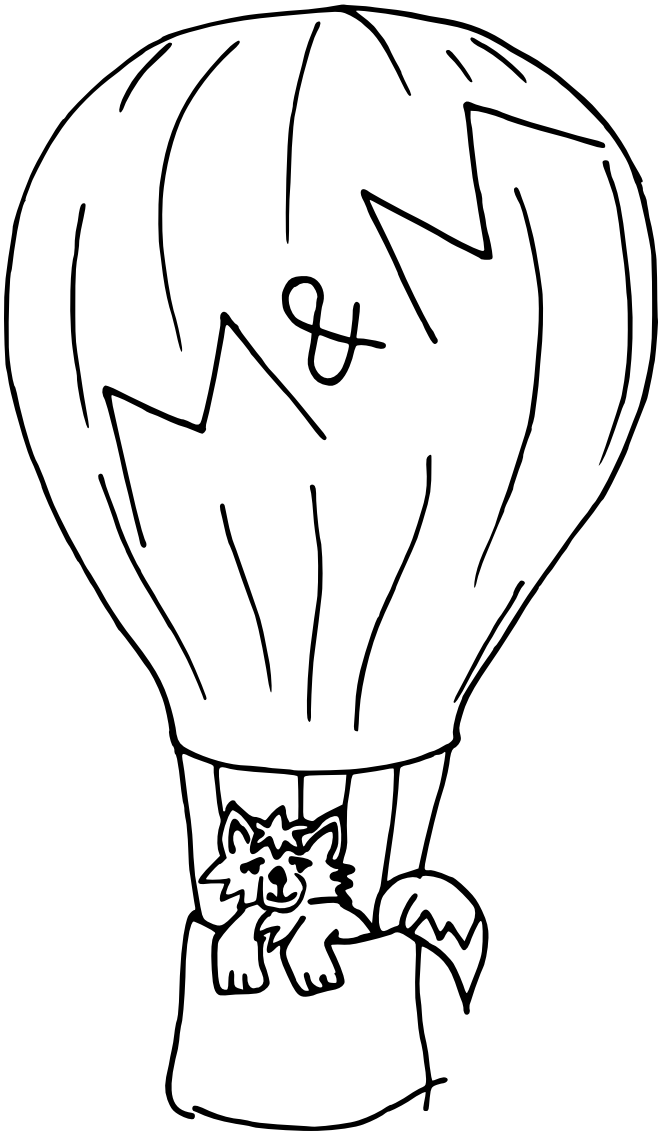
Obrázek 19: Výsledek algoritmu pro různou volbu velikosti okolí



(a) Hodnota statistické veličiny **(b)** Threshold

Obrázek 20

Tímto jsme dorazili na úplný konec tématka o zpracování obrazových dat ze senzorů. Pokud jste se díky tomuto tématku dozvěděli něco nového ze světa strojového vnímání, tak toto tématko splnilo svůj účel. Děkuji vám a mějte se krásně!



Téma 4 – Vybrané kapitoly z elektromagnetismu

Řešení 4. série

Problém 1

Zadání:
Máme vodič zanedbatelné tloušťky a z něj zformujeme kruhovou cívku se 150 závity o poloměru 2 cm. Na konce drátu přiložíme voltmetr a cívku vložíme do vnějšího magnetického pole. Velikost vektoru magnetické indukce zvyšujeme s časem podle rovnice $B(t) = 12t - 0{,}003$ *(jednotka mT). Směr vektorů magnetické indukce je konstantní – svisle dolů. Rovina plochy smyčky je natočená oproti vodorovnému směru o* $30°$*. Vypočtěte, jaké napětí bychom naměřili na koncích drátu v tomto stavu.*

Řešení:
Potřebujeme vypočítat napětí generované v proměnném poli na cívce s N závity, použijeme tedy vztah

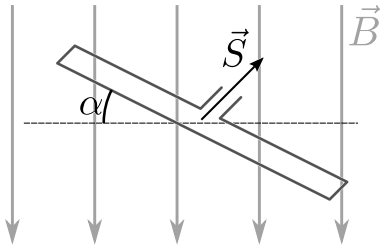
$$\varepsilon_i = -\frac{\mathrm{d}\Phi}{\mathrm{d}t},$$

kde

$$\Phi = \oint_S NB\cos\alpha \mathrm{d}S.$$

Magnetická indukce je vůči ploše smyčky neměnná (mění se s časem, ne v závislosti na ploše závitu), tudíž ji lze vytknout před integrál. Nezávislost na ploše se vztahuje i na úhel natočení smyčky vůči magnetickým indukčním čarám, takže ten lze také vytknout. Integrujeme přes celou plochu, výpočet integrálu vypadá takto:

$$\Phi = B\cos\alpha \oint_S \mathrm{d}S = NBS\cos\alpha.$$



Obrázek 21: Nákres smyčky podle situace popsané v Problému 1.

Tyto úvahy nám integraci velmi zjednodušily a vztah pro magnetický indukční tok jde přepsat na

$$\Phi = NBS\cos\alpha.$$

Spojení vzorců nám dá vztah

$$\varepsilon_i = -\frac{\mathrm{d}(NBS\cos\alpha)}{\mathrm{d}t}.$$

Vzhledem k tomu, že počet závitů N, plocha závitu smyčky S a úhel natočení α jsou v čase neměnné, můžeme je vytknout a derivovat jen magnetickou indukci podle předpisu ze zadání. Postup vypadá takto:

$$\varepsilon_i = -\frac{\mathrm{d}(NBS\cos\alpha)}{\mathrm{d}t} = -NS\cos\alpha\frac{\mathrm{d}B}{\mathrm{d}t}$$

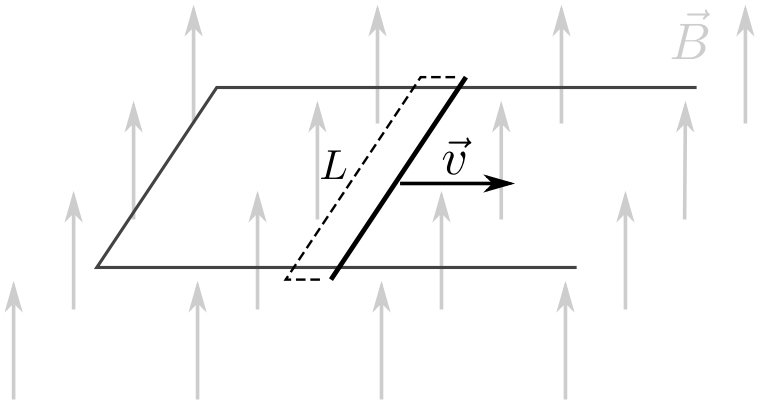
Po zderivování předpisu pro $\boldsymbol{B}$ (lineární funkce času), převedení na základní jednotky, dosazení hodnot ze zadání a vypočtení plochy závitu smyčky $S = \pi r^2$, kde $r = 2\,\mathrm{cm}$, se dostaneme ke vztahu

$$\varepsilon_i = -150 \cdot 0{,}012 \cdot \pi \cdot 0{,}02^2 \cdot \cos 30^\circ = -1{,}959\,\mathrm{mV}.$$

Na koncích cívky bychom naměřili 1,959 mV.

Problém 2

Zadání:
Napište, jak byste vypočetli indukované napětí a jaký směr bude mít výsledný proud v situaci na obrázku 22.



Obrázek 22: Modelová situace pro indukci napětí v homogenním vnějším magnetickém poli. Pohyb vodičem o délce L bude měnit velikost plochy smyčky. Plocha smyčky je kolmá na vektory magnetické indukce vnějšího magnetického pole.

Řešení:
Podle obrázku 22 ze zadání je nám jasné, že plocha smyčky bude v čase narůstat. Malý přírůstek plochy za čas $\mathrm{d}t$ je roven $\mathrm{d}S = Lv\mathrm{d}t$. Abychom se vyhnuli

integraci vztahu pro magnetický indukční tok, tak si pomůžeme matematickou úpravou: celkový magnetický indukční tok je součet jeho nekonečně malých přírůstků, které značíme $\mathrm{d}\Phi$. Tento symbol se vyskytuje i ve vzorci pro výpočet napětí indukovaného vnějším magnetickým polem. Navíc namísto celkového magnetického toku nyní pracujeme jen s jeho nekonečně malou částí, takže si můžeme dovolit vynechat znaménko integrálu a použít jen vztahy s diferenciály pro malé přírůstky. Jinými slovy: z celkového magnetického indukčního toku jsme udělali jen jeho nekonečně malou část, takže jsme si mohli dovolit vynechat integrál na pravé straně vztahu, který jsme doteď používali ve tvaru $\Phi = \oint_S NB\cos\alpha \mathrm{d}S$.

Této shody využijeme a vzorec pro výpočet ε_i upravíme dosazením vztahu $\mathrm{d}\Phi = B\mathrm{d}S = BvL\mathrm{d}t$. Kosinus úhlu jsme mohli vynechat, protože vektory plochy a vnější magnetické indukce jsou rovnoběžné.

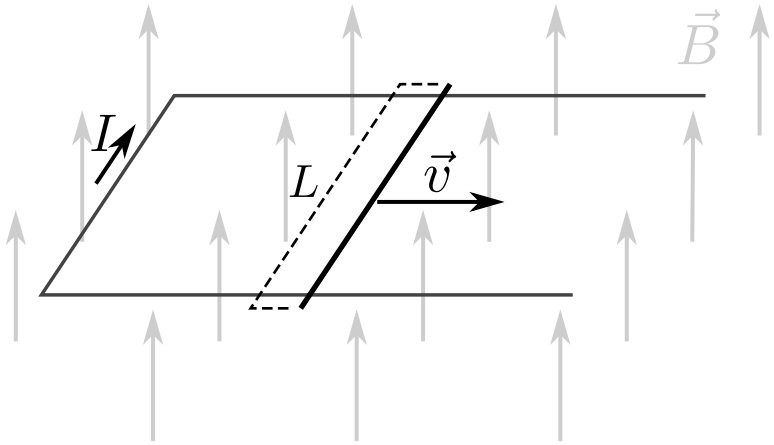
Nyní jen dosadíme:

$$\varepsilon_i = -\frac{\mathrm{d}\Phi}{\mathrm{d}t} = -\frac{BvL\mathrm{d}t}{\mathrm{d}t} = -BvL$$

.

Nyní určíme směr protékajícího indukovaného proudu podle pravidla pravé ruky a znalosti, že vektory indukované magnetické indukce $\boldsymbol{B'}$ musí mít opačný směr než vektory vnější magnetické indukce $\boldsymbol{B}$.

K obrázku přiložíme pravou ruku tak, že palec ukazuje proti směru vektorů $\boldsymbol{B}$ a stočené prsty nám ukazují směr toku proudu I smyčkou, viz obrázek 23.

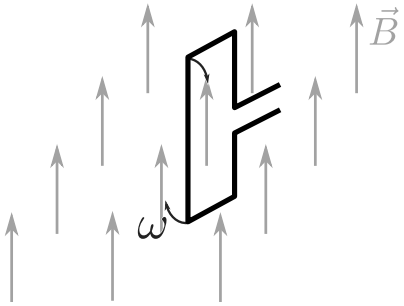


Obrázek 23: Smyčka s vyznačeným protékajícím proudem I.

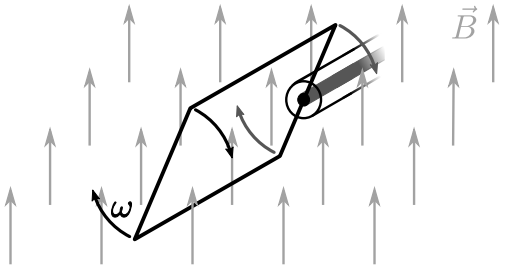
Problém 3

Zadání:
Na vás je zamyslet se, jaký bude časový průběh tohoto napětí při konstantní velikosti veličin B a ω. Plocha smyčky má povrch S. Vyneste své závěry do grafu závislosti generovaného napětí na čase. Porovnejte s průběhem velikosti složky magnetické indukce, která se podílí na indukci napětí. (Nápověda: Rozmyslete si, kam budou směřovat vektory normály plochy a vektor magnetické indukce $\boldsymbol{B}$.)

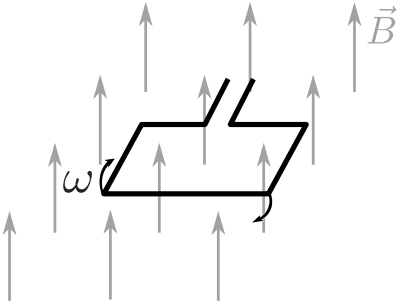
Řešení:
Pro rotující smyčku platí, že složka vektoru plochy smyčky kolmá na magnetické pole bude proměnná v čase. Tato proměnnost bude charakterizovaná změnou kosinu úhlu α, se kterou souvisí úhlová rychlost ω.



(a) zobrazuje námi zvolený počáteční stav



(b) zobrazuje obecnou polohu smyčky během rotace



(c) jeden z extrémních případů rovnoběžnosti vektorů

Obrázek 24: Vybrané momenty při rotaci smyčky v magnetickém poli.

Popis problému charakterizuje obrázek 24. Jako počáteční stav v nulovém čase si zvolíme ten, kdy je vektor plochy smyčky kolmý na vektor magnetické indukce, tedy $\alpha = 90°$. Rotace smyčky bude v záporném směru, tedy po směru hodinových ručiček.

Nyní si sestavíme obecný vztah pro výpočet indukovaného napětí. V problému 1 jsme si již ukázali, že platí

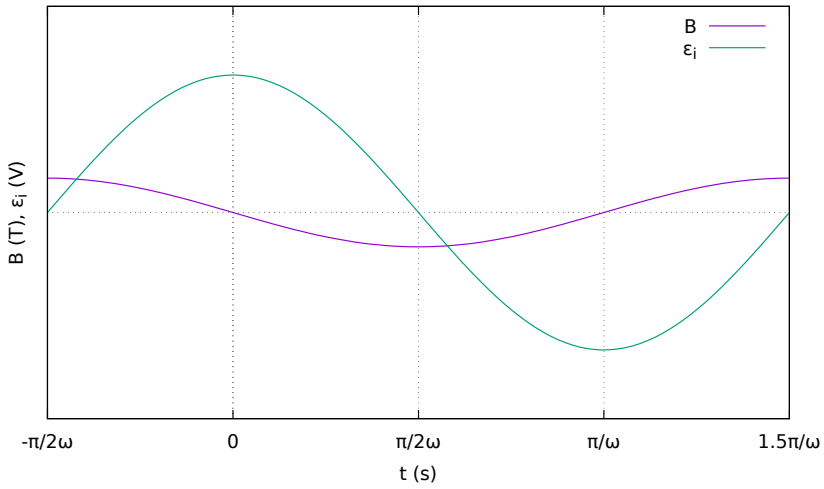
$$\varepsilon_i = -\frac{\mathrm{d}(NBS\cos\alpha)}{\mathrm{d}t}.$$

Pro smyčku platí $N = 1$ a dál tuto proměnnou nebudeme psát. Jediná proměnná na čase je zde úhel mezi vektorem plochy smyčky a vektorem $\boldsymbol{B}$. Tento úhel můžeme charakterizovat jako $\alpha = \omega t + \varphi$ (v radiánech), kde $\varphi = \pi/2$ rad je natočení vektorů v čase $t = 0$ s. Při rostoucím čase narůstá úhel α. Přepišme si vztah pro napětí a zderivujme:

$$\varepsilon_i = -\frac{\mathrm{d}(BS\cos(\omega t + \varphi))}{\mathrm{d}t}$$

$$\varepsilon_i = \omega BS\sin(\omega t + \varphi)$$

Co se týká velikosti účinné složky vektoru $\boldsymbol{B}$, tak nás zajímá ta složka, která je ve směru vektoru plochy magnetické indukce, tedy složka $B\cos(\omega t + \varphi)$. Vynesení závislostí do grafu vidíme na obrázku 25. Argument goniometrických funkcí sinus a kosinus obsahuje proměnnou ω, která zkracuje nebo prodlužuje vzájemný fázový posun sinusoidy a kosinusoidy, stejně tak jako jejich vzájemnou hodnotu maxim. Obecně lze říct, že vzájemný posun maxim hodnot B a ε_i je roven $\frac{\pi}{2\omega}$.



Obrázek 25: Zobrazení závislosti B a ε_i (osa y) na čase (osa x).

Pája, Fanda; fandazajic@gmail.com
e-mailová konference: elmag@mam.mff.cuni.cz

Výsledková listina 4. čísla

Poř.	Jméno	R.	$\sum_{-1}$	Témata t2	t3	t4	turnaj	$\sum_0$	$\sum_1$
1.	Doc.MM K. Vokálová	4	155,6			11,0	16,0	27,0	143,2
2.	Doc.MM J. Kalvoda	3	134,4	28,0			6,6	34,6	134,4
3.	Dr.MM V. Janáček	3	96,9	15,0			6,6	21,6	96,9
4.	Dr.MM M. Fof	2	58,7				4,2	4,2	58,7
5.	Mgr.MM V. Starosta	2	46,0				5,9	5,9	46,0
6.	Mgr.MM O. Piroutek	2	45,4					0	45,4
7.	Mgr.MM K. Pernicová	3	41,8				2,6	2,6	41,8
8.	Mgr.MM J. Knillová	1	35,9	2,3				2,3	35,9
9.	Mgr.MM D. Perout	3	32,0					0	32,0
10.	Mgr.MM L. Veškrna	2	31,8		9,0		9,8	18,8	31,8
11.	Mgr.MM A. Cmielová	1	25,6					0	25,6
12.	Doc.MM J. Havelka	4	129,4					0	25,0
13.	Dr.MM K. Hloušková	4	52,5					0	22,0
14.	Mgr.MM A. Opl	2	20,5					0	20,5
15.–16.	Dr.MM O. Chlubna	3	54,7					0	19,9
	Mgr.MM J. Piroutek	4	33,1					0	19,9
17.	Dr.MM M. Kalousková	4	69,8					0	19,5
18.	Mgr.MM T. Flídr	2	43,9					0	17,9
19.	Mgr.MM L. Kunčarová	4	20,6					0	17,0
20.	Dr.MM M. Holubička	4	54,4	1,0		9,0		10,0	16,6
21.	Bc.MM K. Grinerová	3	16,5					0	16,5
22.	Mgr.MM E. Vítková	4	47,6					0	16,3
23.	Bc.MM J. Kaifer	4	19,8					0	16,0
24.	Mgr.MM O. Gonzor	3	45,4	2,7				2,7	15,5
25.	Bc.MM P. Hladík	2	13,0					0	13,0
26.	Bc.MM J. Jedlička	2	12,7					0	12,7
27.	Bc.MM J. Kvapil	2	15,7					0	12,0
28.	Mgr.MM B. Kopčák	4	28,6					0	11,1
29.	Dr.MM M. Souza de Joode	3	51,3					0	11,0
30.	A. Žáčková	3	9,5					0	9,5
31.	Bc.MM L. Vomelová	4	15,3					0	8,2
32.	Bc.MM V. Žák	4	11,6					0	8,0
33.	Mgr.MM M. Boček	1	29,2					0	7,8
34.	J. Bláhová	3	6,5					0	6,5
35.	Bc.MM E. Neumannová	3	15,3					0	5,2
36.	M. Bučková	3	9,9					0	5,0

Poř.	Jméno	R.	$\sum_{-1}$	Témata t2	t3	t4	turnaj	$\sum_0$	$\sum_1$
37.	Bc.[MM] F. Bujnovský	2	12,2					0	3,4
38.	V. Jůzková	3	8,5	3,0				3,0	3,0
39.	M. Turinská	3	3,0					0	2,0
40.	Bc.[MM] M. Vícha	1	12,4					0	1,4

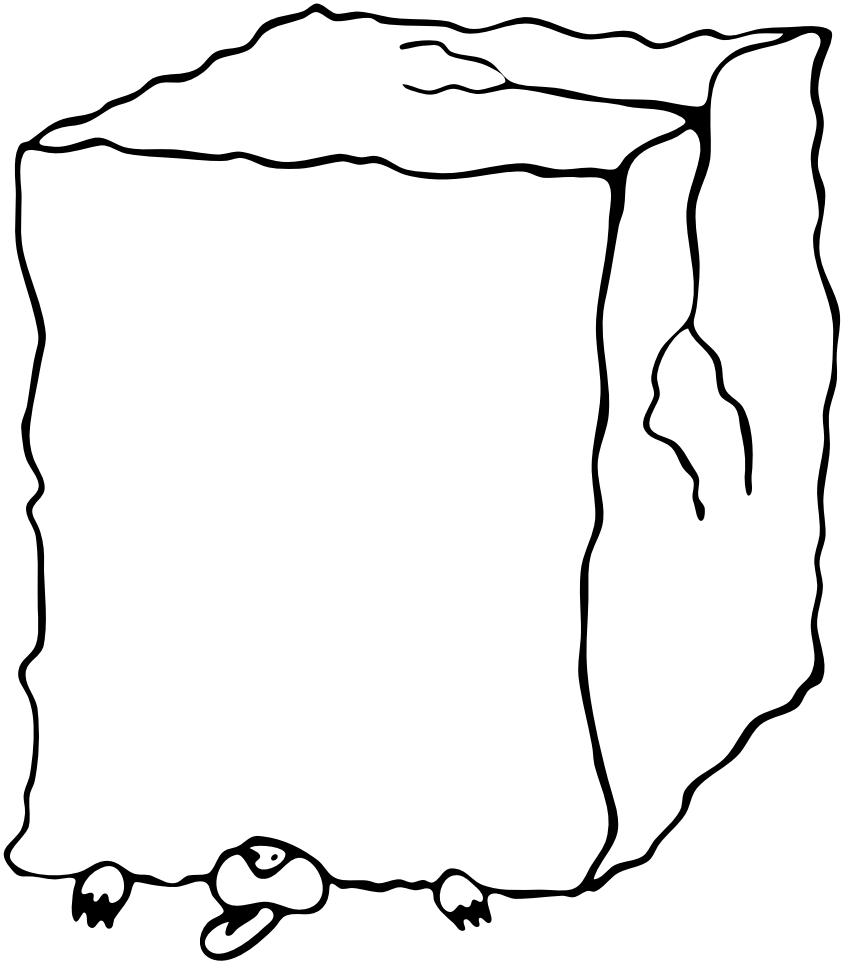
Sloupeček $\sum_{-1}$ je součet všech bodů získaných v našem semináři, $\sum_0$ je součet bodů v aktuální sérii a $\sum_1$ součet všech bodů v tomto ročníku. Tituly uvedené v předchozím textu slouží pouze pro účely M&M.

Výsledková listina 25. ročníku

Poř.	Jméno	R.	$\sum_{-1}$	Číslo 2	4	5	6	$\sum_1$
1.	Doc.[MM] K. Vokálová	4.	155,6	21,2	36,1	58,9	27,0	143,2
2.	Doc.[MM] J. Kalvoda	3.	134,4	5,0	52,0	42,8	34,6	134,4
3.	Dr.[MM] V. Janáček	3.	96,9	10,5	22,0	42,8	21,6	96,9
4.	Dr.[MM] M. Fof	2.	58,7	9,7	21,5	23,3	4,2	58,7
5.	Mgr.[MM] V. Starosta	2.	46,0	0	12,3	27,8	5,9	46,0
6.	Mgr.[MM] O. Piroutek	2.	45,4	10,0	7,6	27,8	0	45,4

Poř.	Jméno	R.	$\sum_{-1}$	Číslo 2	4	5	6	$\sum_{1}$
7.	Mgr.[MM] K. Pernicová	3.	41,8	12,4	6,3	20,5	2,6	41,8
8.	Mgr.[MM] J. Knillová	1.	35,9	20,1	0	13,5	2,3	35,9
9.	Mgr.[MM] D. Perout	3.	32,0	12,8	2,2	17,0	0	32,0
10.	Mgr.[MM] L. Veškrna	2.	31,8	0	0	13,0	18,8	31,8
11.	Mgr.[MM] A. Cmielová	1.	25,6	0	3,6	22,0	0	25,6
12.	Doc.[MM] J. Havelka	4.	129,4	25,0	0	0	0	25,0
13.	Dr.[MM] K. Hloušková	4.	52,5	17,9	4,1	0	0	22,0
14.	Mgr.[MM] A. Opl	2.	20,5	11,5	0	9,0	0	20,5
15.–16.	Dr.[MM] O. Chlubna	3.	54,7	8,0	3,9	8,0	0	19,9
	Mgr.[MM] J. Piroutek	4.	33,1	8,5	11,4	0	0	19,9
17.	Dr.[MM] M. Kalousková	4.	69,8	8,0	7,5	4,0	0	19,5
18.	Mgr.[MM] T. Flídr	2.	43,9	13,5	4,4	0	0	17,9
19.	Mgr.[MM] L. Kunčarová	4.	20,6	17,0	0	0	0	17,0
20.	Dr.[MM] M. Holubička	4.	54,4	6,5	0,1	0	10,0	16,6
21.	Bc.[MM] K. Grinerová	3.	16,5	0	0	16,5	0	16,5
22.	Mgr.[MM] E. Vítková	4.	47,6	14,3	0	2,0	0	16,3
23.	Bc.[MM] J. Kaifer	4.	19,8	16,0	0	0	0	16,0
24.	Mgr.[MM] O. Gonzor	3.	45,4	12,8	0	0	2,7	15,5
25.	Bc.[MM] P. Hladík	2.	13,0	0	0	13,0	0	13,0
26.	Bc.[MM] J. Jedlička	2.	12,7	5,7	0	7,0	0	12,7
27.	Bc.[MM] J. Kvapil	2.	15,7	12,0	0	0	0	12,0
28.	Mgr.[MM] B. Kopčák	4.	28,6	11,1	0	0	0	11,1
29.	Dr.[MM] M. Souza de Joode	3.	51,3	0	0	11,0	0	11,0
30.	A. Žáčková	3.	9,5	9,5	0	0	0	9,5
31.	Bc.[MM] L. Vomelová	4.	15,3	8,2	0	0	0	8,2
32.	Bc.[MM] V. Žák	4.	11,6	8,0	0	0	0	8,0
33.	Mgr.[MM] M. Boček	1.	29,2	7,8	0	0	0	7,8
34.	J. Bláhová	3.	6,5	6,5	0	0	0	6,5
35.	Bc.[MM] E. Neumannová	3.	15,3	5,2	0	0	0	5,2
36.	M. Bučková	3.	9,9	5,0	0	0	0	5,0
37.	Bc.[MM] F. Bujnovský	2.	12,2	3,4	0	0	0	3,4
38.	V. Jůzková	3.	8,5	0	0	0	3,0	3,0
39.	M. Turinská	3.	3,0	2,0	0	0	0	2,0
40.	Bc.[MM] M. Vícha	1.	12,4	1,4	0	0	0	1,4

Sloupeček $\sum_{-1}$ je součet všech bodů získaných v našem semináři, $\sum_{0}$ je součet bodů v aktuální sérii a $\sum_{1}$ součet všech bodů v tomto ročníku. Tituly uvedené v předchozím textu slouží pouze pro účely M&M.



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