FACULTY
OF APPLIED SCIENCES
UNIVERSITY
OF WEST BOHEMIA

University of West Bohemia
Department of Computer Science and Engineering
Univerzitní 8
30614 Pilsen
Czech Republic

Geometrické výpočty a čísla s plovoucí desetinnou čárkou

Technická zpráva

Petr Lobaz

Technical Report No. DCSE/TR-2014-04
August, 2014

Distribution: public

Technical Report No. DCSE/TR-2014-04

August 2014

Geometrické výpočty a čísla s plovoucí desetinnou čárkou

Petr Lobaz

Abstract

The technical report describes floating precision problems in geometric calculations. First, practical issues of ignoring floating point nature of computer calculations are presented. Second, properties of floating point numbers are presented along with practical tips how to use floating point calculations efficiently. Third, geometrical consequences of limited precision calculations are presented and techniques how to avoid problems are described. Finally, two general techniques are presented: sign evaluation and simulation of simplicity.

.

Copies of this report are available on
http://www.kiv.zcu.cz/en/research/publications/
or by surface mail on request sent to the following address:

University of West Bohemia
Department of Computer Science and Engineering
Univerzitní 8
30614 Pilsen
Czech Republic

Obsah

1 O čem text pojednává a kdo by jej měl číst

Je všeobecně známo, že počítače mohou zpracovat pouze omezené množství informace. V případě počítačového zpracování čísel se nejedná pouze o jejich omezené množství, ale také o omezené možnosti při vyjadřování jednoho čísla. Je tedy celkem pochopitelné, že celá čísla umíme zpracovávat pouze v jistém rozsahu. Například současné počítače mají mimořádně dobrou podporu pro práci s čísly ležícími v intervalu -2147483648 až 2147483647 (32bitové vyjádření); je asi také zřejmé, že rozsah reprezentovatelných čísel můžeme zvětšovat tak dlouho, dokud budeme mít k dispozici volnou paměť. V každém případě ale musíme akceptovat skutečnost, že počítač může kvůli omezené paměti pouze aproximovat celočíselnou aritmetiku; zejména v kombinatorických výpočtech se můžeme velmi snadno dostat mimo rozsah reprezentovatelných čísel a výpočetní algoritmus musí být vybaven prostředky na tuto skutečnost zareagovat.

Podobné omezení bude zřejmě platit i pro výpočty s reálnými čísly. I zde se musíme smířit s tím, že umíme reprezentovat pouze čísla z omezeného rozsahu. Navíc musíme přijmout nové omezení – číslo z povoleného rozsahu umíme reprezentovat pouze s omezenou přesností. Tato dvě omezení přináší mnohé těžkosti a mnohé otázky.

První otázka se přímo nabízí: má smysl zabývat se se odlišnostmi aritmetiky reálných čísel a její počítačové aproximace? Bohužel se ukazuje, že je to v mnoha případech nezbytné. Bohužel zde nemůžeme udělat stejnou úvahu, kterou můžeme udělat u celočíselné aritmetiky a kde ji často činíme: pokud budeme pracovat s dostatečně malými čísly, poskytuje počítačová aproximace celočíselné aritmetiky přesné výpočty. Naneštěstí počítačová aproximace reálných čísel poskytuje přesné výsledky spíše výjimečně, ve všech ostatních případech poskytuje více či méně nepřesný výsledek.

Druhá otázka bezprostředně navazuje: má smysl zabývat se nepřesností, když jsou počítače standardně vybaveny mechanismy pro výpočty s přesností na 15 desetinných míst? Pro představu – takovou přesnost potřebujeme, chceme-li měřit průměr Země s přesností na setinu mikrometru. Není takto malá nepřesnost z fyzikální podstaty věci skutečně zanedbatelná? Bohužel si ukážeme, že při nevhodném programování (a ignorování rozdílu mezi přesnou a nepřesnou aritmetikou) může být skutečná chyba celého výpočtu mnohem větší; někdy tak velká, že výsledek je nesmyslný. Jindy se kvůli ignorování rozdílů může výpočet zacyklit, skončit chybou, zkrátka může vést k nepředvídatelným důsledkům, ačkoliv algoritmus je na první pohled správný.

Tento text tedy za prvé ukazuje různé chyby, ke kterým může vést nepozorné zacházení s počítačovou aritmetikou. Hned v úvodu budiž řečeno, že nepůjde o výpočty z „kosmického výzkumu“, kde by se daly blíže neurčené numerické potíže čekat; naopak, půjde o obyčejné úlohy, s kterými se může potkat kterýkoliv programátor.

Za druhé si ukážeme některé důležité aspekty reprezentace čísel s plovoucí

desetinnou čárkou podle standardu IEEE 754; budeme předpokládat, že čtenář je s koncepcí pojmu „plovoucí desetinná čárka“ obeznámen, že zná pojmy exponent a mantisa a že přibližně tuší, jak probíhají základní aritmetické operace s čísly s plovoucí desetinnou čárkou. Také si všimneme často přehlížených hodnot INF a NaN a ukážeme si, jak je prakticky využívat.

Za třetí si ukážeme některé standardní numerické potíže a způsoby, jak se dají řešit.

Ve zbytku textu se budeme věnovat zvláštní disciplíně, a to geometrickým výpočtům v počítačové aritmetice. Oproti jiným výpočtům mají totiž své zvláštnosti: často se na základě (nepřesně) vypočtené hodnoty musíme rozhodovat ano/ne, konvexní/nekonvexní a podobně. V „počítačové“ geometrii také nemusí platit běžné geometrické zákony: dvě různoběžky mohou mít více než jeden průsečík, anebo nemusejí mít žádný; ostatně není vlastně vůbec zřejmé, co znamená pojem „přímka“ atd.

Ačkoliv bude poslední část poměrně úzce zaměřená, nabízí velmi inspirativní pohled na počítačovou aritmetiku pro všechny mírně pokročilé programátory či programující matematiky. Tolik tedy k obsahu textu a předpokládanému publiku.

2 Můžeme si dovolit ignorovat zvláštnosti počítačové aritmetiky?

Jak všichni programátoři vědí, pro prakticky každý program se dají zkonstruovat prapodivná vstupní data, která jej dokonale zmatou. Například hackerské útoky na jakoukoliv infrastrukturu jsou toho skvělým dokladem. Nejinak je tomu u numerických výpočtů.

Katastrofální odečtení

Článek [3] ukazuje pozoruhodný výpočet demonstrující šikovně vybranou úlohu a katastrofální důsledky ignorování zvláštností počítačové aritmetiky. Úloha je až směšně jednoduchá: výpočet funkční hodnoty funkce

$$y = f(a,b) = 333{,}75b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) + 5{,}5b^8 + \frac{a}{2b} \qquad (1)$$

pro $a = 77617$ a $b = 33096$. Podle toho, na jaké platformě počítáme a používáme-li jednoduchou (float, 32 bitů, 24bitová mantisa), dvojitou (double, 64 bitů, 53bitová mantisa), rozšířenou (interní, 80 bitů, 64bitová mantisa) nebo čtyřnásobnou (float128, 128 bitů, 113bitová mantisa) přesnost, obdržíme různé výsledky – před dalším čtením si je prohlédněme v tabulce 1.

Jak z posledního řádku tabulky plyne, ani jeden numerický výpočet nebyl správný, a to ani přibližně! Článek [3] nejenom problém detailně analyzuje, ale

Maple V, Intel	$y = -1{,}18059\ldots \times 10^{21}$
C++, Intel, 24 bitů mantisa	$y = \ \ 6{,}33825\ldots \times 10^{29}$
C++, Intel, 53 bitů mantisa	$y = \ \ 1{,}1726$
C++, Intel, 64 bitů mantisa	$y = \ \ 5{,}76461\ldots \times 10^{17}$
C++, Sun, 24 bitů mantisa	$y = \ \ 6{,}33825\ldots \times 10^{29}$
C++, Sun, 53 bitů mantisa	$y = \ \ 1{,}1726$
C++, Sun, 113 bitů mantisa	$y = \ \ 1{,}1726$
PROFIL/BIAS analýza intervalu	$y \in [-8{,}264\ldots \times 10^{21}; 7{,}0835\ldots \times 10^{21}]$
přesný výpočet	$y = -0{,}82739\ldots$

Tabulka 1: *Výsledky vyhodnocení výrazu* (1) *při použití různých výpočetních prostředků.*

také poukazuje na příčinu: označíme-li si dva z členů funkce f jako z, x:

$$y = f(a,b) = \underbrace{333{,}75b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2)}_{z} + \underbrace{5{,}5b^8}_{x} + \frac{a}{2b},$$

snadno potíž lokalizujeme. Pro daná $a = 77617$, $b = 33096$ totiž platí

$$z = -79171113406689613611011347015249428 50$$
$$x = +79171113406689613611011347015249428 48$$

Pro korektní výpočet $z + x = -2$ tedy potřebujeme aritmetiku, která zvládne 37 platných desítkových cifer; aritmetika s horší přesností povede k nepředvídatelným výsledkům.

Uvedený příklad je zajisté poněkud umělý. Názorně ale demonstruje skutečnost, že ačkoliv jsou všechny vstupy i mezivýsledky bezpečně v rozsahu používané aritmetiky, může být výsledek zcela špatně.

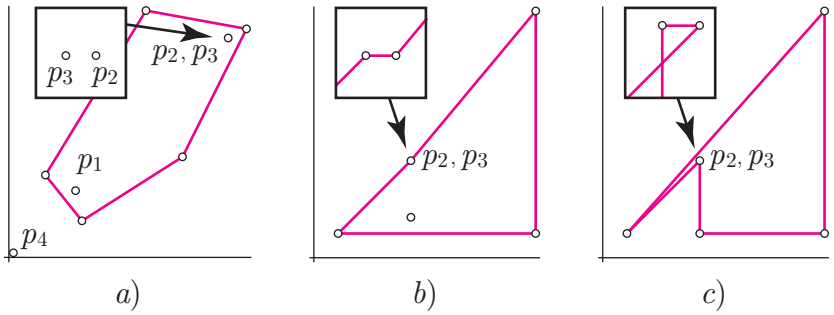
Podívejme se proto na realističtější příklady.

Konvexní obálka

Jednou ze základních urychlovacích metod v geometrických výpočtech je test na konvexní obálku. Například při pohybu robota je třeba detekovat, zda nenarazí do překážky. Jelikož je tvar robota obvykle komplikovaný, vyplatí se jej napřed „obalit“ do konvexního útvaru a test provést nejprve s ním. Pokud text dopadne negativně, máme jistotu, že ani vnitřek konvexního útvaru do ničeho nenarazí. Konvexní útvary se používají proto, že algoritmy pracující s nimi jsou podstatně jednodušší (a rychlejší) než algoritmy pro práci s obecnými tvary.

Algoritmus pracující s konvexními útvary pochopitelně musí na vstupu dostat konvexní útvar. Článek [4] ukazuje různé chyby, které může standardní algoritmus výpočtu konvexní obálky několika bodů udělat, viz obr. 1: obálka je sice konvexní,

ale zjevně neobaluje všechny vstupní body; obálka je nekonvexní; obálka je nekonvexní a dokonce sebeprotínající. Dá se očekávat, že další výpočty založené na chybné konvexní obálce nepovedou ke smysluplným výsledkům.



Obrázek 1: *Příklad několika typů chyb v konstrukci konvexní obálky. a) Konvexní obálka neobsahuje bod p_4. Všimněme si, že body p_1, p_2, p_3, p_4 leží prakticky na stejné přímce jako hrana $p_2 - p_3$. b) Obálka obsahuje zjevnou nekonvexitu. c) Obálka obsahuje zjevnou nekonvexitu, navíc je sebeprotínající.*

Je velmi inspirativní pochopit, jak k takovým chybám může dojít.

Výše uvedené obrázky byly vytvořeny následujícím (správným!) inkrementálním algoritmem založeným na následujících skutečnostech:

- Bod p leží nalevo od hrany ab, je-li trojúhelník abp orientovaný proti směru hodinových ručiček.
- Bod p leží uvnitř konvexního polygonu $c_1c_2 \ldots c_m$ orientovaného proti směru hodinových ručiček, leží-li bod p nalevo od všech hran c_1c_2, c_2c_3, $\ldots$, c_mc_1.
- Bod p leží vně konvexního polygonu $c_1c_2 \ldots c_m$ orientovaného proti směru hodinových ručiček, leží-li bod p napravo od alespoň jedné z hran c_1c_2, c_2c_3, $\ldots$, c_mc_1.
- Pokud bod p leží vně konvexního polygonu $c_1c_2 \ldots c_m$ orientovaného proti směru hodinových ručiček, potom bod p leží napravo od několika hran c_1c_2, c_2c_3, $\ldots$, c_mc_1. Tyto hrany tvoří neprázdný spojitý řetěz. Ostatní hrany tvoří také neprázdný spojitý řetěz.

Samotný postup konstrukce konvexní obálky je jednoduchý:

- Vstup: body p_1, p_2, $\ldots$, p_n, $n \geq 3$.
- Výstup: posloupnost bodů c_1, c_2, $\ldots$, c_m konvexního polygonu s hranami c_1c_2, c_2c_3, $\ldots$, c_mc_1.
- Algoritmus:

1. Inicializuj první verzi obálky: $c_1 = p_1$, $c_2 = p_2$, $c_3 = p_3$, $m = 3$.
2. Uprav konvexní obálku tak, aby orientace c_1, c_2, c_3 byla proti směru hodinových ručiček.
3. Opakuj pro $i \in \{4,\ 5,\ \ldots,\ n\}$:
   (a) Najdi body c_A, c_B $(A < B)$ takové, že bod p_j leží napravo od všech hran $c_j c_{j+1}$ $(A \leq j < B)$. Sčítání v indexu chápeme cyklicky.
   (b) Pokud takové body c_A, c_B nebyly nalezeny, je bod p uvnitř stávající konvexní obálky. Pokračuj další iterací.
   (c) Ve stávající konvexní obálce nahraď body c_A, c_{A+1}, $\ldots$, c_B posloupností bodů c_A, p, c_B. Uprav m a pokračuj další iterací.

Pokud vstup neobsahuje kolineární trojici bodů, pracuje správně. A zde se skrývá kámen úrazu. Například na obrázku 1a začal algoritmus s body p_1, p_2, p_3 a správně vyhodnotil, že tvoří trojúhelník orientovaný proti směru hodinových ručiček. Když pak měl rozhodovat o bodu p_4, který leží téměř na spojnici p_1p_2 (resp. p_1p_3), kvůli numerické chybě špatně vyhodnotil orientaci trojúhelníku $p_3p_1p_4$. Tím nesprávně usoudil, že bod p_4 leží uvnitř trojúhelníku $p_1p_2p_3$ a pokračoval dále. Přidávání dalších bodů už nemohlo tuto chybu zvrátit.

Před uvedením algoritmu jsme formulovali skutečnosti, na kterých spočívalo jeho geometrické uvažování. Všechny skutečnosti spočívaly na vyhodnocení orientace trojúhelníku abc, $a = [a_x, a_y]$, $b = [b_x, b_y]$, $c = [c_x, c_y]$:

$$\text{orientace}(a, b, c) = \text{sign} \begin{vmatrix} 1 & a_x & a_y \\ 1 & b_x & b_y \\ 1 & c_x & c_y \end{vmatrix}$$

Je-li znaménko determinantu kladné, je trojúhelník abc orientován proti směru hodinových ručiček; je-li záporné, má orientaci opačnou; je-li determinant nulový, jsou body a, b, c kolineární. Bohužel, v aritmetice s omezenou přesností může být výsledek testu nesprávný a jediné špatné rozhodnutí může znehodnotit celý výpočet. Skutečnosti, které jsme si před uvedením algoritmu formulovali, jsou tedy platné pouze při použití přesných výpočtů; v nepřesné aritmetice můžeme ke každé skutečnosti najít protipříklad.

K uvedenému příkladu je vhodné doplnit tři poznámky.

První: Algoritmus pro tvorbu konvexní obálky vůbec neuvažoval situaci, kdy jsou tři body kolineární. I kdyby ji však uvažoval, víme už, že v nepřesné aritmetice je vyhodnocení kolinearity přinejmenším problematické. Nestačí tedy do algoritmu přidat další větve pro případ kolinearity; je třeba úplně přehodnotit celé geometrické uvažování.

Druhá: S kolineárními body se můžeme setkat velmi často, zejména tehdy, je-li množina bodů generována z CAD modelu, kde přímky jsou dokonale rovné. Tam kolinearita plyne z podstaty věci. U takové množiny bodů se můžeme navíc setkat

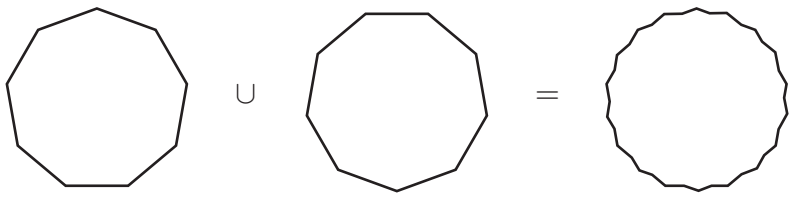
s dalším jevem: pokud mělo několik bodů geometricky ležet na jedné přímce, ale souřadnice těchto bodů nelze v použité aritmetice vyjádřit přesně, budeme pracovat s body, které ve skutečnosti kolineární nejsou.

Třetí: V praxi obvykle pracujeme s rozsáhlými daty, například s množinami milionů bodů. V tak velkých množinách je již značná pravděpodobnost, že některá trojice bodů bude shodou okolností téměř kolineární – a problém je na světě.

Geometrické množinové operace

V geometrickém modelování se často oblé křivky aproximují lomenou čarou, zejména tehdy, jde-li o modely získané 3D skenem reálného předmětu. Dejme tomu, že kružnici máme aproximovanou n-úhelníkem a chceme vypočítat sjednocení tohoto n-úhelníku se svou kopií pootočenou o úhel α, viz obr. 2. Výsledky testů provedených v zavedených programových balících (viz [8]) hovoří za své:

program	n	α [rad]	čas výpočtu	výsledek
ACIS	1000	10^{-4}	5 min	správný
ACIS	1000	10^{-5}	4,5 min	správný
ACIS	1000	10^{-6}	30 min	**zacyklení?**
Microstation95	100	10^{-2}	2 s	správný
Microstation95	100	$0{,}5 \times 10^{-2}$	3 s	**nesprávný**
Rhino3D	200	10^{-2}	15 s	správný
Rhino3D	400	10^{-2}	—	**havárie programu**

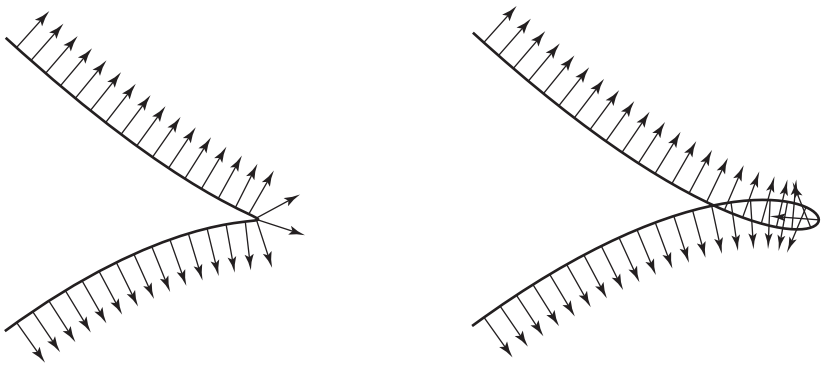


Obrázek 2: *Sjednocení n-úhelníku se svou kopií pootočenou o úhel α.*

Zdroj problémů je celkem zřejmý. Při konstrukci sjednocení je třeba vyhodnocovat průsečíky hran původních objektů, přičemž některé hrany jsou téměř rovnoběžné. Výpočet takového průsečíku je náchylný k numerické chybě. Program to může ignorovat a výsledek může být jakýkoliv (Microstation95, Rhino3D). Jiný program si toho může být vědom a na situaci reagovat; to ale může vést k výraznému prodloužení doby výpočtu (ACIS). Zdroj [11] uvádí, že řešení netriviálních situací zabírá až 90 % času výpočtu.

Simulace vlnění

Při modelování šíření vlny se sleduje čelo vlny (viz [8]), které dělí prostor na dvě části – část „za vlnou“ a část „před vlnou“, viz obr. 3. Jakmile se projeví kvalitativní chyba a vlnoplocha se začne protínat, nelze rozhodnout, která část prostoru už byla vlnou zasažena. Jakékoliv rozhodování založené na topologicky špatném tvaru vlnoplochy pozbývá smysl.



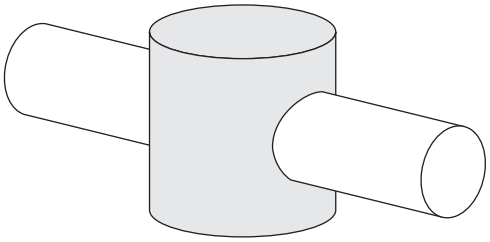
Obrázek 3: *Simulace čela vlnoplochy s vektorem směru šíření. Při sebeprotínající se vlnoploše nelze ani určit, kterým směrem se bude vlna dále šířit.*

Modelování hladkých ploch

Komplikovanější plochy v CAD se typicky tvoří pomocí ořezových křivek (trim curves); například válcová plocha s dírou (viz obr. 4) je definována jako parametrická válcová plocha s dodatečnou informací, podle které povrch obsahuje díru pro jisté hodnoty parametrů plochy. Ořezové křivky je v praxi nutné aproximovat, a zde je kámen úrazu. Mají-li se napojit dvě oříznuté plochy, nedá se jednoduše zajistit, aby byl spoj přesný; může se stát, že mezi nimi vznikne mikroskopická škvíra. Na druhou stranu například algoritmy pro výpočet mechanických vlastností těles předpokládají, že těleso je definováno uzavřenou plochou. Nedodržení předpokladu může vést k libovolnému výsledku výpočtu.

Testování kvality

Některé výrobky prochází náročnou výstupní kontrolou, při které se na základě naměřených hodnot (například rozměrů) musí posoudit, zda je výrobek v předepsané toleranci. Při neopatrné implementaci výpočtů se může snadno stát, že se nekvalitní výrobek prohlásí za bezvadný. Naopak příliš opatrný výpočet může označit výrobek v toleranci za vadný. Jsou-li výrobky velmi drahé, obojí je pochopitelně značný problém (viz [8]).



Obrázek 4: *Válec s dírou, kterou prochází jiný válec. Problematické těleso je válec s dírou – jak zajistit, aby plocha díry přesně navazovala na vnější plášť válce?*

Shrnutí

Ačkoliv je následující odhad z roku 1998 (viz [8]), je přesto alarmující: při výpočtech proudění vzduchu kolem křídla letadla se používají sítě s typicky 50 miliony elementů. Povrchová síť se generuje 10–20 minut, objemová síť další 3–4 hodiny, samotný výpočet proudění zabere cca hodinu a další 2–4 týdny se hledají a opravují chyby způsobené numerickými chybami v generování sítí. Z roku 2002 pochází odhad, podle kterého způsobí numerické chyby softwaru v automobilovém průmyslu ročně škodu jedné miliardy dolarů (viz [8]).

Numerické chyby mohou mít i doslova katastrofální důsledky; v roce 1996 se zřítila raketa Ariane, protože software nereagoval na přetečení při převodu desetinného čísla (double) na celé číslo (int).

Co s tím?

Kupodivu existují typy programů, kde rozumnou odpovědí je „nic“. Jde zejména o programy, kde preferujeme rychlost před přesností, kde se případná chyba projevuje pouze lokálně a nemá významný vliv na další průběh programu. Uvedeme si dva příklady.

- V aplikacích pro zpracování zvuku (či obecně signálu) pracujeme se vzorky často reprezentovanými hodnotami s plovoucí desetinnou čárkou. Algoritmy zpracování signálu se obvykle nevětví podle hodnot vzorků, a pokud ano, pak malá nepřesnost na vstupu způsobí malou nepřesnost na výstupu. I kdyby však byla výstupní nepřesnost velká, chyba v jednom vzorku málokdy způsobí slyšitelnou degradaci zvuku.

- Programy pro rychlou vizualizaci musí na základě (nepřesné) hodnoty s plovoucí desetinnou čárkou rozhodovat, zda pixel obarvit, nebo neobarvit. Raději se ale spokojí se špatně vypočteným pixelem, než aby za cenu značného

zpomalení řešily jeho korektní zobrazení. Na obrázku bude totiž typicky nanejvýš několik promile špatně zobrazených pixelů, což je akceptovatelná daň za rychlost.

Odpověď „nic“ má však dva důležité předpoklady. Za prvé, jeden výpočet neovlivní další výpočty, tedy chybný výsledek se nešíří dál. Za druhé, výsledek celého výpočtu (tj. obrázek, zvuk apod.) neslouží ke kritickému rozhodování, tedy lokální chyba skutečně nemůže způsobit škodu.

Zde by neměl vzniknout mylný dojem, že audiovizuální aplikace nepřesnosti řešit nemusí! Coby ilustraci si uveďme příklad výstupu tzv. rasterizéru, v našem případě programu pro konverzi PDF dokumentu do řídicích příkazů produkčního tiskového stroje, viz obr. 5. Vinou chyby rasterizéru se dva objekty se společnou hranou vyhodnotily jako oddělené a mezera mezi nimi se nevyplnila barvou. Sice je pravda, že se kvůli tenké čáře v ploše jednolité barvy žádná raketa nezřítí, zákazník se však dá stěží přesvědčit, že má za takové výtisky platit plnou cenu.

Obrázek 5: *Sken části vytištěného dokumentu, 4× zvětšeno. Světlá čára vznikla nepřesností rasterizace stínu od žlutého objektu nahoře. Ačkoliv je velmi tenká, je na tiskovině zřetelně vidět.*

Druhým extrémem k odpovědi „nic“ je počítat vše přesně. To má pochopitelně také svá úskalí. I pokud pomineme, že některé výpočty se na počítači přesně vyhodnotit nedají, zejména výpočty s transcendentními čísly (Ludolfovo číslo, Eulerova konstanta apod.), přesto dospějeme k poznání, že pro většinu úloh se „přesné“ výpočty nehodí. Hlavním důvodem je časová náročnost přesných výpočtů; bývají řádově 10000× pomalejší oproti výpočtům s plovoucí desetinnou čárkou, viz [7]. Zpomalení je obzvlášť nepřijatelné, uvědomíme-li si, že většina

výpočtů s plovoucí desetinnou čárkou proběhne dostatečně přesně, extrémní přesnost je typicky zapotřebí pouze u velmi malého množství případů.

V praxi je tedy nejužitečnější kombinovat oba přístupy, tedy počítat pokud možno vše s čísly v plovoucí desetinné čárce a nepřesnost ignorovat, a pouze v kritických případech přistoupit s přesnějšímu vyhodnocení. Nepřesnosti jsou i tam nevyhnutelné, ale robustnost programu lze přesto zařídit pečlivou implementací. Katastrofální chyby způsobené nepřesností jsou sice vzácné, ale při obrovském množství zpracovávaných dat se objevují relativně často.

3 Obecné vlastnosti čísel s plovoucí desetinnou čárkou

Na úvod kapitoly podotkněme, že reálná čísla můžeme v počítači reprezentovat mnoha způsoby: zlomky, čísly s pevnou desetinnou čárkou (fixed point), čísly s plovoucí desetinnou čárkou (floating point), čísly s adaptivní přesností, čísly s definovaným intervalem nejistoty atd., viz [9, 5]. Jelikož současné hardwarové architektury zdaleka nejlépe podporují práci s plovoucí desetinnou čárkou (floating point), bude velmi vhodné si této reprezentace všímat blíže. V této kapitole si připomeneme některé známé skutečnosti a poukážeme na méně známá fakta. Zejména první část kapitoly pojednávající mimo jiné o speciálních kódech definovaných standardem IEEE 754 je mimořádně důležitá pro *všechny* programátory, kteří do programu napíší klíčové slovo *float*.

3.1 Vlastnosti aritmetiky podle IEEE 754

Aproximace reálných čísel čísly s plovoucí desetinnou čárkou napadla většinu tvůrců prvních výpočetních systémů. Bohužel, každého napadla trochu jinak. Proto bylo kdysi prakticky nemožné přenést program z jedné počítačové architektury na jinou, protože ta implementovala základní aritmetické operace jinak a výsledky nebyly identické. A jak jsme již viděli, i drobná chyba může vést k zásadnímu poškození výpočtu.

Zejména proto byl v roce 1985 schválen standard IEEE 754 [1], který popisuje několik datových typů pro práci s čísly s plovoucí desetinnou čárkou a popisuje, jaké vlastnosti mají mít základní operace s nimi. Vlastnostmi se myslí zejména (ale nejenom) zaokrouhlování.

Definované operace jsou:

- sčítání, odčítání, násobení, dělení, zbytek po dělení,
- druhá odmocnina,
- porovnávací operátory,

- konverze mezi celočíselnými typy a typy s plovoucí desetinnou čárkou,
- konverze mezi různými typy s plovoucí desetinnou čárkou definovanými IEEE 754,
- konverze mezi desítkovým (textovým) a binárním tvarem čísla s plovoucí desetinnou čárkou,
- řešení výjimečných situací ve výpočtu (např. dělení nulou).

Kromě toho standard IEEE 754 některé jevy výslovně nespecifikuje, například jisté detaily kódu NaN (viz dále). Je tedy zřejmé, že maximálně přenositelná aplikace se musí spolehnout výhradně na seznam specifikovaných vlastností, přenositelnost aplikací využívajících nestandardizované operace (výpočet logaritmu, goniometrických funkcí atd.) se nedá zaručit.

Bohužel to není vše. Výše uvedené by platilo, pokud by programátor využíval elementární operace přímo, tj. pomocí strojového kódu. Prakticky všechny výpočetní programy jsou ale psané v nějakém vyšším programovacím jazyku, například v C/C++, Fortranu, Matlabu a podobně. Naneštěstí tvůrci překladačů obvykle nebývají experty na detaily práce s plovoucí desetinnou čárkou a programátor má obvykle velmi malou představu o tom, jak výpočet *skutečně* probíhá.

Různé záludné chyby mohou nastat zejména při neopatrné práci s datovými typy (přičemž za neopatrnost často může tvůrce překladače) a při ignorování speciálních hodnot čísel (hlavně INF a NaN). Na detaily se postupně zaměříme.

Základní datové typy

Standard IEEE 754 definuje několik základních typů. Pro běžného programátora mají největší význam tři z nich, single (jednoduchá přesnost, v C++ obvykle datový typ *float*), double (dvojitá přesnost, v C++ obvykle datový typ *double*) a double extended (rozšířená dvojitá přesnost, přímá podpora v C++ není běžná) ve verzi implementované na mikroprocesorech Intel.

Všechny aproximují reálné číslo znaménkem a dvěma celými čísly: exponentem a mantisou; číslo je pak dáno jako

$$(znaménko)\frac{mantisa}{2^{počet\ bitů\ mantisy-1}} \times 2^{exponent} \tag{2}$$

Standard IEEE 754 předpokládá základ exponování 2 a i naše úvahy budou tento základ předpokládat. Pro jiné základy je totiž nutné některá tvrzení týkající se přesnosti výpočtů přeformulovat, viz např. [2]. Zájemci mohou rovněž nahlédnout do normy IEEE 854, která je rozšířením normy IEEE 754 pro základ exponování 10.

Pro naše účely bude stačit, když si připomeneme, že jednotlivé datové typy se liší zejména počtem bitů pro exponent a mantisu. Až na výjimečné případy také

platí, že čísla se ukládají v tzv. *normalizované* podobě, kdy je nejvýznamnější bit mantisy roven jedné. Tato jednička se někdy ukládá a někdy ne; tabulka 2 ji do počtu bitů mantisy započítává.

	single	double	extended double (Intel)
celkový počet bitů	32	64	80
počet bitů mantisy	24	53	64
počet bitů exponentu	8	11	15
rozsah exponentu	$[-126; +127]$	$[-1022; +1023]$	$[-16382; +16383]$
platná desetinná místa (přibližně)	7	16	19

Tabulka 2: *Nejběžnější formáty podle normy IEEE 754.*

Napřed si všimněme v tabulce 2 některých detailů.

Tabulka uvádí přibližný počet platných desetinných míst. Je to hodnota, která byla jednoduše určena z počtu bitů mantisy: jestliže například v přesnosti *single* můžeme reprezentovat $2^{24} = 16\,777\,216$ různých mantis, odpovídá to asi 7–8 cifrám v desítkové soustavě, protože $\log 2^{24} \doteq 7{,}2$. To ovšem neznamená, že by číslo v přesnosti *single* šlo přesně reprezentovat osmiciferným desetinným číslem! Například číslo

$$1{,}11111\,11111\,11111\,11111\,111_2 \times 2^{-126} = 2^{-125} - 2^{-149}$$

kde dolní index 2 značí binární vyjádření, má přesné desítkové vyjádření dlouhé 112 platných cifer:

$$2{,}35098856151472858345576598207153302664571798551798085536592623685000612993034607711706485133618116378784179687 5 \times 10^{-38}\ .$$

Údaj o počtu platných desetinných míst je tedy třeba brát s jistou rezervou. Jistou důležitost, byť v mírně odlišném kontextu, má ovšem přesný počet desetinných míst, který potřebujeme k jednoznačnému vyjádření libovolného čísla v dané přesnosti. Například pro přesnost *single* potřebujeme 9 cifer; pro přesnost *double* potřebujeme 17 cifer, viz [2]. Algoritmus pro jednoznačný převod mezi binárním a desítkovým vyjádřením musí pracovat velmi opatrně; i proto je tento převod zařazen mezi standardními IEEE 754 operacemi a vyžaduje se jeho precizní implementace.

Jak tuto vědomost prakticky využít? Většina programátorů si je vědoma existence různých binárních formátů pro čísla s plovoucí desetinnou čárkou, v nejjednodušším případě lišícím se pořadím bytů (malý endián vs. velký endián). Proto se v „přenositelných“ souborech často vyjadřují čísla v textové podobě, tj. v zápisu desítkovým číslem. Platformní nezávislost takových souborů je ovšem

velmi diskutabilní; musí totiž být zaručeno, že převody mezi desetinnou a binární reprezentací čísla budou probíhat bezpečně. Zajistit bezpečnou konverzi je však mnohem složitější než přeuspořádat data z malého na velký endián, případně provést triviální bitové manipulace. Proto je u kritických dat mnohem vhodnější úplně se textové reprezentaci vyhnout.

Dále si pozorný čtenář z tabulky 2 jednak uvědomí, že rozsah exponentů každého typu nevyčerpává rozsah umožněný počtem bitů exponentu, jednak si všimne, že rozsah exponentů je nesymetrický.

Díky menšímu rozsahu exponentů je možné reprezentovat speciální hodnoty. Například v přesnosti *single* indikuje hodnota exponentu -127 hodnotu 0; jelikož čísla v přesnosti *single* mají uloženou mantisu bez jedničky na nejvýznamnější pozici (reálně tedy ukládají 23 bitů mantisy), nebylo by jinak možné hodnotu nuly přesně zaznamenat. Naopak hodnota exponentu 128 indikuje speciální kódy (INF, NaN), kterým se budeme věnovat později.

Díky nesymetrii exponentů je celkový počet všech možných exponentů sudý; protože potřebujeme dvě hodnoty exponentů rezervovat (pro nulu a speciální kódy), je tato volba přirozená. Díky zvolenému typu asymetrie ($|-126| < 127$) je systém odolnější vůči přetečení; převrácená hodnota nejmenšího možného čísla, tj. $1/2^{-126} = 2^{+126}$, je v přesnosti *single* reprezentovatelná. Je sice pravda, že převrácená hodnota maximálního čísla způsobí podtečení, ale obecně vzato je podtečení méně kritické než přetečení [2].

Podívejme se nyní na avizované potíže s datovými typy.

Zkusme si představit, co udělá následující program v C/C++ (viz [2]):

```
1  int main() {
2    double q;
3    q = 3.0/7.0;
4    if (q == 3.0/7.0) printf("Rovnost\n");
5                      else printf("Nerovnost\n");
6    return 0;
7  }
```

V závislosti na architektuře a překladači může program vypsat jak „Rovnost“, tak „Nerovnost“! Analyzujme, proč tomu tak je.

Mikroprocesory Intel interně počítají operace v pohyblivé řádové čárce v *extended double* přesnosti (není-li řečeno jinak) a před uložením výsledek zaokrouhlí na požadovanou (*single* nebo *double*) přesnost. Má to rozumný důvod. Jak jsme viděli na straně 5, velkým problémem numerických výpočtů je ztráta platných číslic při odečítání podobných čísel. Proto má obvykle smysl provádět výpočty s vyšší přesností a výsledek zaokrouhlit. Podobně to má smysl u výpočtů s transcendentními funkcemi (logaritmus, goniometrické funkce apod.), kde nemáme zaručeno přesné zaokrouhlení (viz str. 30). Mikroprocesor má proto několik registrů pro

práci s čísly s plovoucí desetinnou čárkou, v případě mikroprocesorů Intel s *extended double* přesností. Tam se pokud možno ukládají mezivýsledky při výpočtu komplikovanějšího vztahu.

Nyní je asi pochopitelné, proč se může uvedený program chovat nevyzpytatelně. Výpočet 3,0/7,0 na řádce 3 proběhne v *extended double* přesnosti, zaokrouhlí se na *double* přesnost a uloží do proměnné `q`. I podruhé (na řádce 4) proběhne výpočet 3,0/7,0 v *extended double* přesnosti, do pomocného *extended double* registru se načte hodnota proměnné `q` (zaokrouhlená) a čísla se porovnají. Není divu, že se nemusí rovnat! Pečlivě napsané překladače proto musí před porovnáváním provést zaokrouhlení na požadovanou přesnost.

Obvyklou námitkou je názor, podle kterého se nesmí čísla s plovoucí desetinnou čárkou testovat na rovnost. Něco pravdy na tom je, námitka ovšem nenavrhuje postup, jak rovnost otestovat. Ostatně, relace „větší než“ bude srovnatelně spolehlivá, jakmile budou testovaná čísla téměř stejná.

Nejjednodušší pomůckou je tzv ϵ-test: místo $a = b$ se testuje $|a - b| < \epsilon$. Jednoduché pomůcky obvykle skrývají různé nástrahy a nejinak je tomu i zde. První problém je samozřejmě s volbou prahu ϵ. Obvyklý postup naivního programátora $\epsilon \leftarrow 0{,}0001$ (nebo podobně) samozřejmě selhává, jakmile se pracuje s velmi malými čísly (viz rozsah exponentů IEEE datových typů). Druhý problém je naprosté rozbití relace „rovná se“. U relace „rovná se“ pochopitelně platí implikace

$$a = b \quad \wedge \quad b = c \quad \Rightarrow \quad a = c.$$

Jakmile místo „rovná se“ používáme ϵ-test, implikace neplatí:

$$|a - b| < \epsilon \quad \wedge \quad |b - c| < \epsilon \quad \not\Rightarrow \quad |a - c| < \epsilon.$$

Při naivním nasazení ϵ-testu je tedy možné, že v jednom místě se program rozhodne, že $a \neq c$ (protože $|a - c| > \epsilon$), zatímco na jiném místě se implikací rozhodne o opaku (protože $|a - b| < \epsilon$ a $|b - c| < \epsilon$). Jelikož robustní program se musí rozhodovat konzistentně, není zřejmě ϵ-test dlouhodobě udržitelný.

Další problémy s *extended double* přesností následují. Pokud výpočet výrazu obsahuje část, kterou lze vyhodnotit pouze za jistých okolností (např. druhou odmocninu jen tehdy, je-li argument nezáporný), je obvyklé před výpočtem ověřit kritické části výrazu (konkrétně například test znaménka argumentu odmocniny). Pokud překladač před testem argument zaokrouhlí např. na přesnost *double*, zjevně se netestuje správné číslo – v samotném počítaném výrazu bude argument odmocniny bez zaokrouhlení.

Programátor si často kvůli zpřehlednění kódu nebo zvýšení efektivity výpočtu komplikovanější výraz rozdělí na podvýrazy, vypočítá je odděleně a uloží do pomocných proměnných; samozřejmě se skrytým zaokrouhlením. Komplikovanější výraz se pak zjednoduší náhradou podvýrazů za hodnoty v proměnných – ale je zřejmé, že výsledek už nebude identický.

S *extended double* se pojí jedna další nepříjemnost.

Základní operace mají být podle standardu IEEE 754 implementovány tak, aby výpočet proběhl jakoby dokonale přesně a výsledek se zaokrouhlil na požadovanou přesnost. To znamená, že všechny cifry (bity) výsledku jsou platné. Pokud ale interně počítáme v *extended double* přesnosti, výsledky operací jsou průběžně zaokrouhlovány na *extended double* přesnost a výsledek je navíc zaokrouhlen na požadovanou, například *double* přesnost. Dvojí závěrečné zaokrouhlení ovšem může vést ke zneplatnění poslední cifry. Můžeme to demonstrovat jednoduchým příkladem. Dejme tomu, že přesný výsledek je 15,46 a požadovaná přesnost je celočíselná. Správně zaokrouhlený výsledek je tedy 15. Pokud ale výsledek napřed zaokrouhlíme na *extended* přesnost, v našem příkladu to bude přesnost na jedno desetinné místo, obdržíme napřed 15,5, a pak finálním zaokrouhlením 16. Dalo by se sice argumentovat, že chyba na posledním místu není důležitá, ale jednak jsme již vliv podobných „nepodstatných“ chyb viděli, jednak mnohé přesné algoritmy spoléhají na platnost *všech* cifer.

Závěrem můžeme povědět, že neopatrná manipulace s čísly interně reprezentovanými různou přesností může vést k prapodivným výsledkům. To ovšem není povelem k zavržení *extended precision* a jiných technik dočasné práce s vyšší přesností – jen je třeba postupovat opatrně. Mimo jiné proto vznikla v roce 2008 revize standardu IEEE 754, která podobné zádrhele řeší (viz [13]).

Kódy INF, NaN a další

Během výpočtů s reálnými čísly může program úmyslně či častěji neúmyslně vyžadovat operaci, jejíž výsledek není matematicky definovaný. Jsou to zejména dělení nulou a odmocnina ze záporného čísla, při použití jiných než IEEE 754 operátorů může být takových případů mnohem více. Kromě toho může sice matematicky výsledek existovat, ale nepůjde vyjádřit žádným platným číslem se zvolenou přesností.

S nejjednodušším případem jsme se již setkali; například výsledek 1,0/3,0 nelze vyjádřit žádným IEEE 754 kódem. Jak víme, operace musí vrátit výsledek zaokrouhlený na nejbližší možné číslo ve zvolené přesnosti. Co se však má stát, požadujeme-li v jednoduché přesnosti výsledek $1{,}0/10^{-40}$? Má se rovněž zaokrouhlit na nejbližší možné číslo, tj. cca $3{,}4 \times 10^{38}$, nebo se má volajícímu programu oznámit přetečení? Má se výsledek $1{,}0/10^{+40}$ zaokrouhlit na nulu, nejmenší možné číslo (cca $1{,}2 \times 10^{-38}$), nebo se má oznámit podtečení? Má se při snaze o výpočet 1,0/0,0 vrátit největší možná hodnota, speciální kód, nebo se má volající program zastavit? Rozhodnout tyto a další otázky není vůbec snadné, protože každá volba způsobí problémy. Tvůrci standardu IEEE 754 se snažili, aby tyto problémy způsobily co nejméně rozdílů mezi zákony reálné a přibližné aritmetiky.

Při pokusu o operaci, která vede k výjimečnému výsledku (např. dělení nulou), by samozřejmě šlo výpočet zastavit. Zdaleka ne vždy je to ale nejrozumnější možnost. Například při numerickém řešení rovnice $f(x) = 0$ musí algoritmus

vyhodnocovat funkci f v různých bodech zadaného intervalu. Je jistě příjemnější, když algoritmus pracuje i tehdy, když se při vyhodnocování omylem „strefí“ do hodnoty mimo definiční obor funkce f. Pokud by každá taková „trefa“ byť i jen vyvolala mechanismus výjimky, trvalo by hledání kořenu funkce typicky mnohem déle než při bezproblémovém chodu.

Proto se v IEEE 754 zavádí speciální veličiny: NaN (not a number; nedefinovaná hodnota), INF (nekonečno), −INF (minus nekonečno), +0 („kladná nula“), −0 („záporná nula“) a denormalizovaná čísla. Budeme se jim postupně věnovat, ukážeme si důvody jejich zavedení a praktické ukázky použití. Důležitým důsledkem je jasně definovaná hodnota libovolné operace s libovolnými operandy, tj. standardem IEEE 754 je definován součet INF a běžného čísla, podíl dvou NaN atd.

Kód INF se generuje vždy, je-li výsledkem operace číslo v absolutní hodnotě větší, než je maximální možné. To je mnohem bezpečnější než zaokrouhlení na nejvyšší možné číslo; například v jednoduché přesnosti by výsledkem $\sqrt{(2^{80})^3}$ bylo číslo $1{,}8^{19}$, zatímco správný výsledek je $1{,}3 \times 10^{36}$. Oproti tomu je argument odmocniny v IEEE 754 aritmetice roven INF a je definováno, že $\sqrt{\text{INF}} = \text{INF}$. Obecně platí, že v případě $\lim_{x\to\infty} f(x) = \infty$ je i výsledek vyhodnocení $f(\text{INF}) = \text{INF}$. Se stejnou logikou bylo zavedeno, že $1{,}0/0{,}0 = \text{INF}$, protože $\lim_{x\to 0+} 1/x = +\infty$.

Praktickým příkladem použití INF ve výpočtu je vyčíslení funkce (viz [2])

$$f(x) = \frac{x}{x^2 + 1} \ .$$

Takový zápis je špatný ze dvou důvodů. Za prvé pro velká x dojde snadno k přetečení ve jmenovateli. Za druhé bude pro velká x kvůli přetečení výsledkem $x/\text{INF} = 0$ místo přesnějšího $1/x$. Vztah můžeme opravit jednoduše přepsáním na

$$f(x) = \frac{1}{x + 1/x} \ .$$

Pro velká x bude pracovat přesněji a bez přetečení, pro malá x rovněž a díky INF aritmetice bude správně pracovat i pro $x = 0$, protože se správně vyhodnotí $1/(0+\text{INF}) = 0$. Bez INF aritmetiky by byl třeba test na $x = 0$, který pochopitelně nedělá dobrotu v pipeline nebo paralelním zpracování.

Analogicky ke kódu INF je definován kód −INF, který je výsledkem např. $-1{,}0/0$. Se zavedením −INF se pojí zajímavý problém: mají se lišit hodnoty $1{,}0/\text{INF}$ a $1{,}0/-\text{INF}$? Ve standardu IEEE 754 se liší, neboť nula má definované znaménko; existuje tedy +0 a −0. Díky tomu je například pro všechna x platná rovnost $1/(1/x) = x$. Kromě toho je možné rozlišovat malá kladná a záporná čísla $+e$ a $-e$, která se vlivem podtečení zaokrouhlila na nulu. Počítáme-li například odmocninu takového čísla, můžeme snadno rozlišit $\sqrt{+e} = 0$ od $\sqrt{-e} = \text{NaN}$. Na druhou stranu je pravda, že zavedením −0 a +0 se některé věci komplikují. Například test $x = y$ nestačí provádět porovnáváním bit po bitu, protože IEEE 754

aritmetika definuje $-0 = +0$. Tím se také zanáší podivné chování, kdy z $x = y$ neplyne $f(x) = f(y)$, například $1/+0 \neq 1/-0$. Tvůrci IEEE 754 ale usoudili, že výhody znaménkové nuly převažují její nevýhody; tak nebo tak, se znaménkovou nulou musíme počítat.

Kód NaN se generuje v případech, kdy není možné konzistentně rozhodnout o výsledku. Je známo, že pokud $\lim_{x\to 0} f(x) = 0$ a $\lim_{x\to 0} g(x) = 0$, potom $\lim_{x\to 0} f(x)/g(x)$ může nabývat libovolné hodnoty. Obdobně pro $\lim_{x\to\infty} f(x) = \infty$ a $\lim_{x\to\infty} g(x) = \infty$ může být hodnota $\lim_{x\to\infty} f(x) - g(x)$ libovolná. V takových případech generují IEEE 754 operace kód NaN. Stejně tak jej generují pro odmocninu či logaritmus záporného čísla a podobně. Přesněji řečeno, generují *některý* z kódů NaN, protože může být užitečné rozlišovat, jakou operací kód NaN vznikl. Standard IEEE 754 ovšem jednotlivé kódy NaN nerozlišuje, respektive nechává jejich počet a význam na dílčí implementaci.

Poslední zvláštní kategorií kódů jsou denormalizovaná čísla. Zmíníme je jen pro úplnost; pro praktické výpočty je jejich význam poměrně malý a i většina přesných algoritmů s nimi příliš neoperuje.

Při reprezentaci čísel v normalizovaném tvaru je nejmenším reprezentovatelným číslem $n_{\min} = 1{,}0 \times 2^{e_{\min}}$, kde $e_{\min}$ je minimální možný exponent; například u přesnosti *single* je $e_{\min} = -126$. Mezi nulou a $n_{\min}$ je tedy poměrně velká mezera. Pokud ale umožníme zapisovat čísla s exponentem $e_{\min} - 1$ v nenormalizovaném tvaru (tj. nejvýznamnější bit mantisy je explicitně 0), plynule mezeru mezi nulou a $n_{\min}$ zaplníme.

Má takové chování praktické opodstatnění? Ano. Počítáme-li například v přesnosti *single* hodnotu $x/(x - y)$ pro velmi blízká x a y, může se v rozdílu ve jmenovateli většina cifer odečíst a výsledek $x - y$ bude menší než $n_{\min}$. Výsledkem $x - y$ tedy bude po zaokrouhlení nula i přesto, že $x \neq y$, a tedy výsledek dělení bude INF! V aritmetice s normalizovanými čísly tedy platí $x = y \not\Leftrightarrow x - y = 0$; negací takto elementárního aritmetického pravidla se samozřejmě veškeré úvahy o správnostech výpočtů stávají nespolehlivými. Právě tento nedostatek řeší denormalizovaná čísla; dá se ukázat, že jejich zavedením opět pro libovolná x a y platí $x = y \Leftrightarrow x - y = 0$, viz [2].

S denormalizovanými čísly se pojí jedna perlička. Mnoho programovacích jazyků zavádí konstanty typu `DBL_MIN`, `FLT_MIN` apod. s významem minimálního reprezentovatelného čísla v přesnosti *single*, *double* apod. Pozor! Tyto konstanty mohou označovat nejmenší možné číslo v normalizovaném tvaru! Je tedy možné regulérně definovat hodnotu proměnné `float x`, pro kterou $0 <$ `x` $<$ `FLT_MIN`. Konkrétně např. Microsoft Visual Studio definuje v souboru `float.h` hodnotu `FLT_MIN` $\doteq 1{,}18 \times 10^{-38}$, ovšem nejmenší možná hodnota v přesnosti *single* je přibližně $1{,}4 \times 10^{-45}$.

Zabývejme se nyní bezprostředními praktickými důsledky speciálních kódů definovaných v IEEE 754.

Běžnou praxí při výpočtu komplikovanější hodnoty $f(x)$ je před samotným

výpočtem otestovat, zda x patří do definičního oboru funkce f. Například při výpočtu kořenů kvadratické rovnice $ax^2 + bx + c$ se typicky testuje znaménko diskriminantu $b^2 - 4ac$ a nenulovost koeficientu a. Jak již víme, takové testy jednak nefungují vždy dobře při výpočtech v *extended precision*, jednak jakékoliv testování komplikuje paralelní výpočet několika funkčních hodnot (rozbití instrukční pipeline, potíže s SIMD instrukcemi). Pokud naopak funkční hodnotu napřed vypočteme a až následně otestujeme, zda je výsledkem konečné číslo, je problém vyřešen. Programovací jazyky obvykle takové testy nabízejí, jen o nich většina programátorů neví. Například v C/C++ jsou tyto testy definovány v `float.h` pod názvy `_finite()`, `_isnan()` atd.

Při zpracování dvourozměrných dat potřebujeme často vyhodnocovat údaje v jiné než obdélníkové oblasti. Například program pro vizualizaci funkce $f(x, y)$, jehož vstupem bude (obdélníková) matice s funkčními hodnotami, potřebuje nějak vizuálně odlišit body s regulérními funkčními hodnotami od bodů, kde funkce není definována. Mnoho programátorů si pomáhá pomocným dvourozměrným polem s příznaky „hodnota funkce existuje/neexistuje“ nebo jinými zbytečnými postupy; přirozené řešení je neplatné funkční hodnoty označit kódem NaN a zajistit, aby vizualizační program na tuto hodnotu korektně reagoval.

Hodnoty INF a NaN nepřinášejí jen šikovné vlastnosti, ale i některé méně zřejmé jevy při vyhodnocování relací $=$, $<$, $>$, $\leq$ a $\geq$. Pokud se například kód NaN porovnává s číslem y, je výsledkem vždy *false*. Proto se může nezkušený programátor podivit, že program

```
1 int main() {
2   float x = 3;
3   while (x >= 0) {
4     x = sqrt(x - 1);
5    }
6 }
```

proběhne korektně, zatímco na první pohled ekvivalentní podmínka v cyklu povede k nekonečnému opakování:

```
1 int main() {
2   float x = 3;
3   while (!(x < 0)) {
4     x = sqrt(x - 1);
5    }
6 }
```

Uvedeným příkladem také poskytujeme návod, jak bez pomoci interních funkcí otestovat, zda je $x = \mathrm{NaN}$:

```
1 if (x==x)
2   { /* x není NaN */}
3 else
4   { /* x je NaN */}
```

Poslední poznámka, kterou uvedeme v kapitole o vlastnostech aritmetiky podle IEEE 754, se týká rozdílů mezi přesnou a nepřesnou aritmetikou.

Kvůli omezené přesnosti čísel je zřejmé, že například nemusí platit

$$1 + (10^{30} - 10^{30}) \stackrel{?}{=} (1 + 10^{30}) - 10^{30} \ . \tag{3}$$

Současné překladače však často v rámci optimalizace přemýšlí, jak výraz vyhodnotit co nejefektivněji, případně zda výraz vůbec vyhodnocovat. Bohužel při svém uvažování obvykle používají vlastnosti přesné aritmetiky. Proto se může chování programu přeloženého s optimalizacemi značně lišit od neoptimalizovaného kódu.

3.2 Obecné vlastnosti výpočtů s čísly s nepřesnou aritmetikou

Jak již bylo řečeno, nepřesnostem se v praktických výpočtech nevyhneme. Programátor však může postupovat tak, aby se mnohým potížím vyhnul s vynaložením minimálního úsilí. V této části textu se seznámíme s několika užitečnými způsoby uvažování.

Zaokrouhlování

Víme, že celočíselná aritmetika poskytuje přesné výsledky, pokud během výpočtu nedojde k přetečení nebo podtečení. Pokud proto potřebujeme vyhodnotit výpočet, jehož vstupem jsou celá čísla, a očekáváme opět celá čísla, je velmi riskantní výpočet kvůli pohodlnosti vyhodnocovat v nepřesné aritmetice. To se týká zejména výpočtů se zlomky.

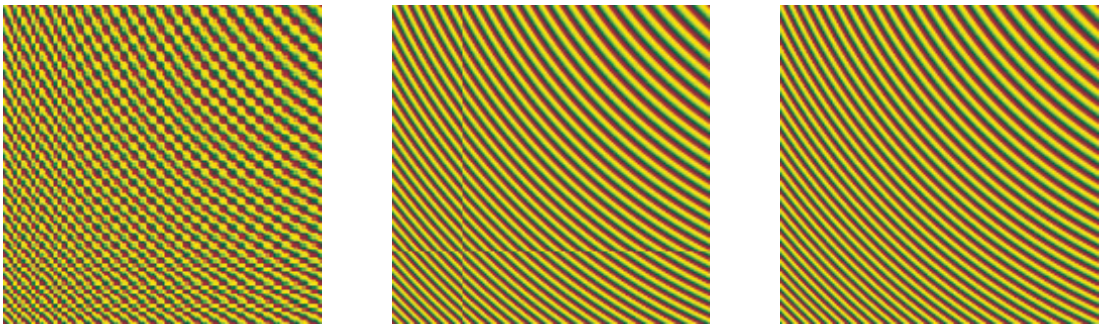
Coby příklad si můžeme uvést převzorkování obrazu. Na obrázku 6 se můžeme přesvědčit, že nesprávná implementace (obrázky 6a, 6b) může vést k neočekávaným výsledkům. Podívejme se, jak problémy vznikly.

Vstupem úlohy je obrázek definovaný maticí I, jejíž prvky $I[k,l]$ představují vzorky funkce f v bodech $[k\Delta_I + i_x, l\Delta_I + i_y]$, kde k, l jsou celočíselné indexy prvku, Δ_I je vzorkovací vzdálenost a reálná čísla i_x, i_y určují pozici prvku $I[0,0]$. Hodnotu funkce f v obecném bodu $[x,y]$ definujme stylem „nejbližší soused“, tj.

$$\begin{aligned} f(x,y) &\stackrel{\text{def}}{=} f\left(\text{round}\left(\frac{x-i_x}{\Delta_I}\right)\Delta_I + i_x, \text{round}\left(\frac{y-i_y}{\Delta_I}\right)\Delta_I + i_y \right) \\ &= I\left[\text{round}\left(\frac{x-i_x}{\Delta_I}\right), \text{round}\left(\frac{y-i_y}{\Delta_I}\right) \right], \end{aligned}$$

kde funkce round představuje zaokrouhlení k nejbližšímu celému číslu. Pro jednoduchost předpokládejme, že indexy k, l nejsou omezené.

Naším úkolem je vytvořit „obrázek zvětšený a posunutý“. Popíšeme jej maticí O s prvky $O[m,n]$, které představují vzorky funkce f v bodech $[m\Delta_O + o_x, n\Delta_O +$



a) b) c)

Obrázek 6: *Různé metody převzorkování obrazu: a) matematicky korektní, ale programátorsky naivní implementace využívající čísla s plovoucí desetinnou čárkou, b) lepší, ale stále nedokonalá implementace redukující počet operací, c) korektní implementace založená na celých číslech.*

$o_y]$, kde m, n jsou opět celočíselné indexy prvku, Δ_O je vzorkovací vzdálenost a o_x, o_y jsou reálná čísla popisující pozici prvku $O[0,0]$. Platí tedy zřejmě

$$\begin{aligned} O[m,n] &= f(m\Delta_O + o_x, n\Delta_O + o_y) \\ &= f\left(\text{round}\left(\frac{m\Delta_O + o_x - i_x}{\Delta_I}\right)\Delta_I + i_x, \text{round}\left(\frac{n\Delta_O + o_y - i_y}{\Delta_I}\right)\Delta_I + i_y \right) \\ &= I\left[\text{round}\left(\frac{m\Delta_O + o_x - i_x}{\Delta_I}\right), \text{round}\left(\frac{n\Delta_O + o_y - i_y}{\Delta_I}\right) \right]. \end{aligned}$$

Naivní výpočet všech prvků $O[m,n]$ bychom tedy mohli provést takto:

```
1 for (m = ...; m < ...; m++) {
2   for (n = ...; n < ...; n++) {
3     x = m * delta_o + o_x;
4     y = n * delta_o + o_y;
5     index_x = round((x - i_x)/delta_i);
6     index_y = round((y - i_y)/delta_i);
7     o[m,n] = i[index_x, index_y];
8 }}
```

Pokud ovšem zkusíme kód spustit pro $\Delta_i = \Delta_o = 0{,}1$, $i_x = i_y = 0$ a $o_x = o_y = 0{,}05$, dostaneme výsledek jako na obrázku 6a (srovnej s korektním výstupem na obrázku 6c).

Původ chyby je zjevný: číslo $\Delta_i = 0{,}1$ nelze korektně reprezentovat číslem podle standardu IEEE 754. Číslo $o_x = 0{,}05$ je přesně polovinou vzorkovací vzdálenosti, a navíc jej také nejde ve standardu IEEE 754 reprezentovat. Výpočty na řádcích 3–6 jsou pak zatíženy zaokrouhlovacími chybami. Jelikož je výpočet

funkce round v okolí rozhodovacího prahu velmi citlivý na vstup, je výběr vzorku na řádku 7 téměř dílem náhody.

O něco lepší řešení bere v potaz chyby zaokrouhlování a snaží se výpočet přeuspořádat tak, aby se při $\Delta_I = \Delta_O$ násobily indexy m, n jedničkou:

```
1 for (m = ...; m < ...; m++) {
2   for (n = ...; n < ...; n++) {
3     index_x = round(m*(delta_o/delta_i) + (o_x - i_x)/delta_i);
4     index_y = round(n*(delta_o/delta_i) + (o_y - i_y)/delta_i);
5     o[m,n] = i[index_x, index_y];
6 }}
```

Kód poskytuje pro výše uvedené parametry ($\Delta_i = \Delta_o = 0{,}1$, $i_x = i_y = 0$ a $o_x = o_y = 0{,}05$) uspokojivé výsledky. Jakmile ale zvolíme matematicky ekvivalentní parametry $i_x = i_y = 100$ a $o_x = o_y = 100{,}05$ dostaneme výsledek jako na obrázku 6b – pro dostatečně vysoké hodnoty indexů (v našem případě je prahem 256) výpočet „přeskočí“ a úplně vynechá jeden řádek, resp. sloupec vstupu.

Problematické místo je nyní skryto v rozdílu $100{,}05 - 100$. Jelikož číslo 100,05 nelze ve standardu IEEE 754 přesně reprezentovat a část bitů mantisy je třeba věnovat hodnotě 100, je hodnota $100{,}05 \ominus 100$ rozdílná od hodnoty $0{,}05 \ominus 0$, tj. několik bitů přesnosti bylo ztraceno. Nepatrná chyba se projeví právě na rozhraní, kde tyto „ztracené bity“ začnou hrát roli.

Proto je jediným spolehlivým řešením vyhnout se konverzi celé číslo $\rightarrow$ desetinné číslo $\rightarrow$ celé číslo. Program by tedy měl detekovat případy, kdy Δ_I a Δ_O jsou v jednoduchém celočíselném poměru a výpočet proměnných `index_x`, `index_y` založit na celočíselné aritmetice.

Periodické funkce

Uvedený problém zaokrouhlování je však interně přítomný i ve výpočtech, kdy bychom jej na první pohled nečekali.

Ve výpočtu funkční hodnoty $f(x)$ periodické funkce f, např. $\tan(x)$, je typicky nutné napřed hodnotu x normalizovat na hodnotu x_n, tj. omezit na interval základní periody, tedy např. na interval $[0, \pi)$. Normalizaci teoreticky vypočteme snadno: $x_n = (x \bmod \pi) = x - \pi\lfloor x/\pi \rfloor$. Pokud je tedy x mnohem větší než π, tj. délka periody funkce f, dojde při normalizaci intervalu ke značné chybě a výsledek $f(x)$ je nepřesný až nesmyslný. Pro ilustraci problému se podívejme na tabulku 3 shrnující výsledky výpočtu $\sin(10^{22})$.

Uvedený příklad výpočtu funkce $\tan(x)$ skrývá další úskalí: pro x blízká $\pi/2$ je funkce $\tan(x)$ velmi strmá, čili nepatrná nepřesnost v x způsobí značnou nepřesnost výsledku. Tato jednoduchá úvaha se dá říci i jinak: pro x blízké $\pi/2$ je výpočet $\tan(x)$ *špatně podmíněný*.

Podívejme se na pojem podmíněnosti blíže.

platforma	$\sin(10^{22})$
přesný výsledek	$-0{,}8522008497671888017727\ldots$
Vax VMS (g or h format)	$-0{,}852200849\ldots$
HP 48 GX	$-0{,}852200849762$
HP 700	$0{,}0$
HP 375, 425t (4.3 BSD)	$-0{,}65365288\ldots$
Matlab V.4.2 c.1 for Macintosh	$0{,}8740$
Matlab V.4.2 c.1 for SPARC	$-0{,}8522$
PC: Borland TurboC 2.0	$4{,}67734 \times 10^{-240}$
Sharp EL5806	$-0{,}090748172$
DECstation 3100	NaN
TI 89	Trig. arg. too large

Tabulka 3: *Výpočet* $\sin(10^{22})$ *na různých platformách. Výňatek z tabulky převzaté z [15]. Jak autor [15] poznamenává, číslo* 10^{22} *se dá v přesnosti* double *reprezentovat přesně. Výsledky byly zaslány mnoha uživateli a nebyly diskutovány s výrobci.*

Podmíněnost

Počítáme-li $f(x)$, ale x známe pouze v aproximaci x' s *absolutní chybou* $\hat{e}$, tj. $x' = x + \hat{e}$, dostaneme výsledek $f(x') = f(x + \hat{e}) \doteq f(x) + \hat{e}f'(x)$, kde $f'(x)$ je první derivací funkce $f(x)$. Je tedy pochopitelné, že se vyhýbáme situacím, kdy je $|f'(x)| > 1$. Toho lze často snadno dosáhnout pouhým přeformulováním vztahu.

Například článek [6] navrhuje tři alternativní vzorce pro výpočet úhlu θ svíraného vektory $\mathbf{r}$ a $\mathbf{s}$:

$$\theta = \operatorname{acos} \frac{\mathbf{r} \cdot \mathbf{s}}{|\mathbf{r}||\mathbf{s}|} \tag{4a}$$

$$\theta = \operatorname{asin} \frac{2A}{|\mathbf{r}||\mathbf{s}|} \tag{4b}$$

$$\theta = \operatorname{atan} \frac{2A}{\mathbf{r} \cdot \mathbf{s}} = \operatorname{atan2}(2A, \mathbf{r} \cdot \mathbf{s}), \tag{4c}$$

kde A je plocha trojúhelníku svíraného úhly $\mathbf{r}$ a $\mathbf{s}$; v článku [6] je rovněž uveden postup přesného výpočtu A. Nejznámější vzorec (4b) plyne přímo z vlastností skalárního součinu; málokdy se však hovoří o tom, že výpočet $\operatorname{acos}(x)$ je velmi špatně podmíněný pro $|x|$ v okolí 1. Podobné vlastnosti má i funkce $\operatorname{asin}(x)$. Naproti tomu první derivace funkce $\operatorname{atan}(x)$ je všude menší než 1 a z hlediska podmíněnosti je tedy nejvýhodnější. Pokud navíc pro výpočet využijeme funkci $\operatorname{atan2}(x, y)$, kterou nabízí většina matematických knihoven, není ani třeba speciálně ošetřovat případy s nulou ve jmenovateli zlomku.

Detailnější pohled na pojem podmíněnosti nabízí například [14], ve stručnosti uvedeme nejdůležitější body.

Při výpočtu $f(x)$ je často x výsledkem nějakého výpočtu; musíme tedy předpokládat, že je nějakým způsobem zaokrouhlené na hodnotu x'. Zajímá nás, v jakém vztahu jsou $f(x)$ a $f(x')$.

Budeme předpokládat, že x' je srovnatelně velké jako x; zavedeme tedy *relativní chybu* e:

$$\frac{x'}{x} = 1 + e \quad \Leftrightarrow \quad \frac{x' - x}{x} = e \quad \Leftrightarrow \quad x' - x = ex \ .$$

Relativní chybu e jsme zavedli místo absolutní chyby $\hat{e}$ používané v předchozích odstavcích proto, že je pro následující analýzu výhodnější. Zajímat nás bude absolutní hodnota relativní chyby výsledku, tj.

$$\left| \frac{f(x') - f(x)}{f(x)} \right| \ .$$

Korektně vzato bychom sice měli předpokládat, že i samotný výpočet funkce f je zatížen zaokrouhlovací chybou, pro jednoduchost ale tento předpoklad zanedbejme.

Provedeme formální úpravu

$$\frac{|f(x') - f(x)|}{|f(x)|} = \frac{|f(x') - f(x)|}{|x' - x|} \times \frac{|x|}{|f(x)|} \times \frac{|x' - x|}{|x|} \ .$$

První člen součinu je zřejmě odhadem $|f'(x)|$, poslední člen je absolutní hodnota relativní chyby x'. Proto můžeme psát

$$\frac{|f(x') - f(x)|}{|f(x)|} \doteq \kappa_f(x)\,|e| \ ,$$

kde

$$\kappa_f(x) = \frac{|f'(x)| \times |x|}{|f(x)|} \ .$$

Číslo $\kappa_f(x)$ nazýváme *relativním číslem podmíněnosti* funkce f v bodu x; název se často zkracuje na *číslo podmíněnosti*. Přibližně udává, jak se zvětší relativní chyba výsledku oproti relativní chybě x.

Jako příklad použití vypočítejme relativní podmíněnost funkce $f(x) = x - y$ (y je konstanta), tedy výpočtu rozdílu dvou čísel. Zřejmě platí $f'(x) = 1$, a proto

$$\kappa_f(x) = \frac{|x|}{|x - y|} \ .$$

Relativní číslo podmíněnosti diverguje k nekonečnu pro $x \to y$, tedy i teoreticky máme potvrzeno, že rozdíl dvou blízkých čísel je zatížen značnou relativní chybou. Stejným postupem zjistíme, že operace násobení ($f(x) = x \times y$) a dělení

($f(x) = y/x$) jsou dobře podmíněné ($\kappa_f(x) = 1$ pro všechna x). Rovněž zjistíme, že výpočet druhé odmocniny ($f(x) = \sqrt{x}$) je velmi dobře podmíněný, protože $f'(x) = 1/(2\sqrt{x})$ a tedy $\kappa_f(x) = 0{,}5$ pro všechna x.

Pozorný čtenář si ovšem uvědomí, že závěrečný výsledek je jistým způsobem v rozporu se začátkem kapitoly, kde jsme tvrdili, že je vhodné vyhýbat se výpočtu funkce $f(x)$ pro $|f'(x)| > 1$ – což pro funkci odmocniny znamená pro $x < 0{,}25$.

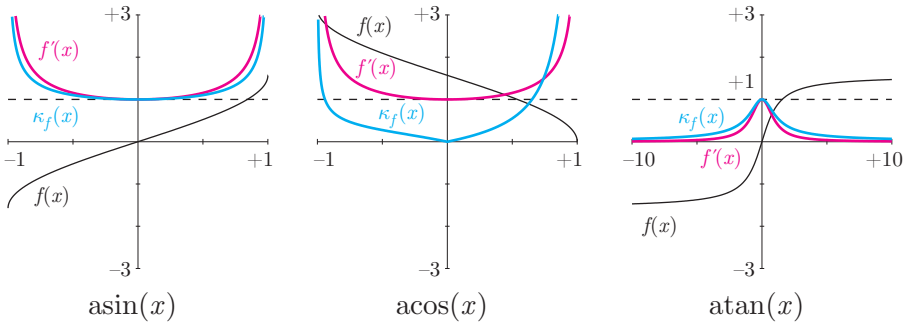
Zdánlivý nesoulad je způsobený odlišným pojetím pojmu *chyba výpočtu*. Zatímco v prvních úvahách jsme se zabývali absolutní chybou $\hat{e} = f(x') - f(x)$, v úvahách o podmíněnosti jsme vycházeli z formulace relativní chyby $e = (f(x') - f(x))/f(x)$. Oba dva pohledy na chybu mají svoje opodstatnění a své užití; pojmem chyby výpočtu bychom se tedy měli zabývat hlouběji. Věnujeme mu proto následující podkapitolu.

Tento oddíl uzavřeme zpětným pohledem na výpočet úhlu mezi dvěma vektory podle vzorců (4a–4c) na straně 25. Podobně, jako jsme definovali relativní číslo podmíněnosti $\kappa_f(x)$ coby míru zvětšení relativní chyby výpočtu funkce f v bodu x, mohli bychom definovat absolutní číslo podmíněnosti $\hat{\kappa}_f$, které bude udávat míru zvětšení absolutní chyby výpočtu. Měli bychom je zřejmě zavést jako poměr absolutní chyby výpočtu funkce $f(x)$ ku absolutní chybě argumentu funkce. Snadno zjistíme, že

$$\hat{\kappa}_f(x) = \frac{|f(x') - f(x)|}{|x' - x|} \quad \overset{x' \to x}{\longrightarrow} \quad |f'(x)|$$

a tedy (jak již víme), výpočet funkce $f(x)$ v bodu x, pro který $|f'(x)| > 1$, bude zatížen větší absolutní chybou.

Podíváme-li se detailně na všechny vzorce (4a–4c) a nakreslíme si jejich průběhy, průběhy prvních derivací a čísla podmíněnosti (viz obr. 7), zjistíme, že univerzální je skutečně vzorec (4c) založený na výpočtu funkce $\mathrm{atan}(x)$, resp. $\mathrm{atan2}(x, y)$. Když si navíc uvědomíme, že by kvůli nepřesnostem výpočtu mohl být argument funkcí $\mathrm{asin}(x)$ či $\mathrm{acos}(x)$ mimo rozsah $[-1, 1]$ (a tedy výsledkem výpočtu by byl kód NaN), jeví se vzorec (4c) jako ještě užitečnější.



Obrázek 7: *Závislost zvětšování velikosti absolutní (purpurová) a relativní (azurová) chyby při výpočtu* $\mathrm{asin}(x)$, $\mathrm{acos}(x)$ *a* $\mathrm{atan}(x)$.

3.3 Přesnost výpočtu s plovoucí desetinnou čárkou

V předchozí podkapitole jsme se zabývali obecnou problematikou chyby výpočtu bez ohledu na konkrétní implementaci. Jelikož je však hardwarová podpora operací s čísly s plovoucí desetinnou čárkou tak velká, je vhodné se zaměřit úžeji a při analýze chyb vzít tuto implementaci v potaz.

Pro analýzu chyby výpočtu je vhodné zavést pojmy *absolutní* a *relativní chyby*. Už jsme se s nimi setkali, ale pro přehlednost si je definujme znovu.

Mějme veličinu x a její přibližné vyjádření x'. *Absolutní chybou* $\hat{e}$ rozumíme veličinu

$$\hat{e} = x' - x \qquad \Leftrightarrow \qquad x' = x + \hat{e}\,,$$

relativní chybou e rozumíme veličinu

$$e = \frac{x' - x}{x} \qquad \Leftrightarrow \qquad x' = x(1 + e)\,.$$

Povšimněme si, že absolutní chyba $\hat{e}$ má stejný fyzikální rozměr jako veličina x, zatímco relativní chyba e je bezrozměrným číslem.

Hovoříme-li o velikosti chyby (absolutní i relativní), obvykle zanedbáváme její znaménko. Někdy se ve vztazích s chybami $\hat{e}$ či e explicitně pracuje s absolutními hodnotami; protože ale absolutní hodnoty značně komplikují manipulaci s těmito vztahy, často se absolutní hodnoty vynechávají a jistá volnost ve znaménku chyby se tiše předpokládá.

Smysl absolutní chyby je zřejmý, jde typicky o rozdíl mezi přesnou a nepřesnou veličinou. Není už však tolik zřejmé, zda je například absolutní chyba 1 mm akceptovatelná, nebo ne – pro veličinu v řádu kilometrů bude typicky akceptovatelná, naopak pro veličinu v řádu mikrometrů bude neakceptovatelná. To je hlavní důvod zavedení relativní chyby; je-li relativní chyba mnohem menší než 1, je výsledek obvykle dostatečně přesný.

Relativní chyba ovšem také není univerzální. Za prvé není definována, je-li veličina x rovna nule. Za druhé, a o tom se příliš často nehovoří, je výhodná pouze u tzv. lineárních veličin, v počítačové grafice například u jasu. Naopak u veličin perceptuálních, v počítačové grafice například u vnímaného jasu, je výhodnější absolutní chyba; chyba ± 1 má totiž stejnou váhu jak u malých vnímaných jasů, tak u velkých. Podobně je relativní chyba zavádějící, má-li x význam souřadnice; má-li být díra vyvrtána na souřadnici x, může být akceptovatelná souřadnice $x \pm \hat{e}$, ale nemá opodstatnění posuzovat chybu souřadnice díry v závislosti na absolutní pozici jejího umístění, tj. vztahem $x(1 \pm e)$.

Jistým kompromisem může být vztažení absolutní chyby ke vhodně zvolené konstantě K. Například pro délkové tolerance ve stavebnictví se může zvolit jako 1 cm, pro tolerance v jemné mechanice jako 10 μm. Volba konstanty K, která nezávisí na velikosti x, přirozeně vede k zavedení reprezentace čísla x s pevnou

desetinnou čárkou (fixed point) – ještě před výpočtem totiž víme, že potřebujeme čísla s jistým počtem platných cifer, přičemž pro zvolené K bude několik z nich za desetinnou čárkou.

V jiných aplikacích, zejména v takových, kde není předem zřejmý rozsah veličiny x, je vhodnější zavést pojem *počet platných (desetinných) míst*. To platí zejména tehdy, není-li předem známo, v jakých jednotkách bude veličina x zadána. V běžném životě si obvykle vystačíme se třemi platnými místy – čtvrtá a další platná číslice by byla „pod rozlišovací schopností“ například pro výšku člověka 176 cm, hmotnost 12,3 t, cenu 4,56 Kč za kWh apod. Volba počtu platných míst přirozeně vede k reprezentaci čísla x s plovoucí desetinnou čárkou (floating point). „Konstanta“ K je nyní závislá na velikosti x, přičemž nejvýhodnější je definovat ji jako velikost spojenou s posledním platným desetinným místem; obvykle ji značíme zkratkou ulp z anglického *unit in the last place*. Je-li tedy skutečná hmotnost 12 312 kg zapsána jako 12,3 t, zvolili jsme ulp = 100 kg a chyba vyjádření je zřejmě $(12\,312 - 12\,300)/100 = 0{,}12$ ulp.

V reprezentaci čísel podle standardu IEEE 754 má ulp velikost, která odpovídá poslednímu bitu mantisy. Například číslo 1,9 je v přesnosti *single* vyjádřeno jako

$$1,\underbrace{111\,001\,100\,110\,011\,001\,100\,11}_{23\text{ bitů}}{}_2 \times 2^0 \doteq 1{,}899\,999\,976_{10}\,,$$

kde dolní indexy u čísel značí základ použité číselné soustavy. Zde je zřejmě ulp $= 2^{-23} \times 2^0 = 2^{-23} \doteq 1{,}19 \times 10^{-7}$, a tedy chyba reprezentace je rovna $1{,}9 - 1{,}899\,999\,976 \doteq 2{,}38 \times 10^{-8} \doteq 0{,}2$ ulp . Obecně platí, že pro číslo x' vyjádřené normalizovanou mantisou délky p (pro přesnost *single* je $p = 24$) a exponentem E je ulp $= 2^{E-p+1}$.

Reálné číslo, které má být reprezentováno číslem podle standardu IEEE 754, musí být zaokrouhleno. Při zaokrouhlování dolů nebo nahoru je zaokrouhlovací chyba menší než 1 ulp, při zaokrouhlení k nejbližšímu reprezentovatelnému číslu nejvýše 0,5 ulp. Kromě toho také standard IEEE 754 předepisuje, že výsledky základních operací mají být takové, jako by byly operace provedeny přesně a výsledky následně korektně zaokrouhleny; při zaokrouhlení k nejbližšímu reprezentovatelnému číslu je tedy absolutní chyba výsledku aritmetické operace nejvýše 0,5 ulp.

Jednotka ulp tedy dobře slouží k odhadu maximální absolutní chyby. Nyní si zavedeme protějšek k odhadu maximální relativní chyby.

Číslo x si vyjádříme pomocí reálného čísla m ($1 \le m < 2$) a celého čísla E jako $x = m2^E$. Číslo x budeme aproximovat číslem x'. To vyjádříme normalizovanou mantisou m' ($1 \le m' < 2$) o délce p bitů (např. pro přesnost *single* 24 bitů) a stejným celým číslem E, tedy $x' = m'2^E$. Po korektním zaokrouhlení x k nejbližšímu reprezentovatelnému číslu x' je absolutní chyba aproximace $\hat{e}$ maximálně 0,5 ulp, tedy $|x' - x| = \hat{e} \le 0{,}5$ ulp. Při délce mantisy p bitů je pro dané x, jak již

víme, ulp $= 2^{E-p+1}$. Pro relativní chybu e pak můžeme určit

$$\begin{aligned} e &= \left|\frac{x' - x}{x}\right| = \left|\frac{\hat{e}}{x}\right| \\ &\leq \left|\frac{0{,}5\,\mathrm{ulp}}{x}\right| = \left|\frac{2^{E-p}}{m2^E}\right| \\ &\leq 2^{-p} = \epsilon\,. \end{aligned}$$

Veličinu ϵ, kterou jsme zavedli, nazýváme *strojovým epsilon*. Kromě významu omezení relativní chyby má i jiný, snadno uchopitelný význam: je to největší takové číslo, pro které platí (v nepřesné aritmetice s mantisou délky p bitů) $1 + \epsilon = 1$.

Pro $x = 1$ je pochopitelně relativní chyba e číselně rovna absolutní chybě $\hat{e}$. Připomeňme si však, že ani pro $x = 1$ nelze psát $e = \hat{e}$, protože relativní a absolutní chyba mají jiné fyzikální jednotky. Rovnost $1 + \epsilon = 1$ v předchozím odstavci tak musíme chápat jako speciální formu zápisu $x(1 + \epsilon) = x'$ pro $x = 1$, nikoliv jako práci s absolutní chybou.

Poznamenejme, že mnozí nezkušení programátoři nesprávně chápou strojové ϵ jako „nejmenší reprezentovatelné číslo“. Občas se lze setkat s konstrukcí, kdy místo testu $x = y$ použije programátor berličku $|x - y| < \epsilon$, kde „sofistikovaně“ použije strojové ϵ. Jak z předchozího textu plyne, obě chyby pramení ze zásadní neznalosti pojmu.

Zaokrouhlování stojí za všemi problémy spojenými s aritmetikou podle standardu IEEE 754, a to i přesto, že jeho implementaci byla věnována velká péče. Zájemci se mohou různé podrobnosti dozvědět v [2], například:

- IEEE 754 definuje a podporuje zaokrouhlování nahoru, dolů i k nejbližšímu reprezentovatelnému číslu. Poslední zaokrouhlovací režim sice vede k nejmenší chybě, zbývající dva se naopak vhodně využívají v intervalové aritmetice, tj. v definici intervalu, kde výsledek určitě leží.

- Při zaokrouhlování k nejbližšímu reprezentovatelnému číslu je vždy problém, jak zaokrouhlit číslo přesně uprostřed; například při zaokrouhlování na celá čísla je problém, jak zaokrouhlit 0,5. IEEE 754 standard zvolil v mezních případech strategii „zaokrouhlit k sudé mantise“, protože pak například nebude opakované provádění přiřazení $x \leftarrow (x - y) + y$ měnit hodnotu proměnné x.

- Pro základní operace definované IEEE 754 lze najít algoritmy, které vrátí přesně zaokrouhlený výsledek. Naproti tomu je to komplikované nebo to nelze udělat pro transcendentní funkce jako $\sin x$, $\log x$ apod. Zde se naplno projevují výhody aritmetiky s rozšířenou přesností (*extended precision*), protože je relativně snadné najít algoritmy poskytující výsledek s přesností v řádu stovek ulp.

My se do podrobností pouštět nebudeme; uvedený výčet sloužil jen jako ukázka hloubky úvah, které stojí za implementací jakéhokoliv přesného algoritmu. Stran zaokrouhlování ještě dodejme, že nebude-li řečeno jinak, budeme v celém tomto textu předpokládat zaokrouhlování k nejbližšímu reprezentovatelnému číslu. Navíc zavedeme následující notaci: symboly x, y, … budou značit (přesná) reálná čísla a operátory $+$, $-$, $\times$, $/$, $\sqrt{\ }$ operace s nimi. Symboly x', y', … budou značit zaokrouhlená čísla x, y, … Operátory $\oplus$, $\ominus$, $\otimes$, $\oslash$, $\mathring{\sqrt{\ }}$ pak budou značit operaci, jejíž výsledek je zaokrouhlený.

Díky přesnému zaokrouhlování výsledků základních operací tak můžeme psát

$$a \oplus b = (a + b)(1 + e) \qquad \text{kde } |e| \leq \epsilon \tag{5}$$

a podobně pro operace rozdílu, součinu, podílu, odmocniny a zbytku po dělení. Alternativně můžeme psát (viz [9])

$$a + b = (a \oplus b)(1 + e) \qquad \text{kde } |e| \leq \epsilon \tag{6}$$

a podobně pro ostatní základní operace. Tyto vztahy tvoří základ veškerého dalšího posuzování chyby výpočtu; kdyby je IEEE 754 aritmetika nezaručovala, nešlo by chybu složitějších výpočtů odhadnout.

Zdálo by se, že zaručenou přesností základních aritmetických operací je problém přesnosti výpočtu vyřešen. Bohužel, není to pravda. Je sice pravda, že například $x \oplus y = (x + y)(1 + e)$, kde $|e| < \epsilon$; výsledkem zaokrouhleného součtu je ovšem nepřesné číslo z'. Jakmile se nepřesné číslo použije v jiné operaci, chyba se začne kumulovat a výsledná nepřesnost může být významně vyšší než ϵ pro relativní chybu nebo 0,5 ulp pro absolutní chybu. Podívejme se proto, jak kumulace chyby probíhá. Následující odvozování nebudou samoúčelná; ukazují, jakým stylem se provádí analýza chyby výpočtu.

Nechť $x' = x(1 + e_x)$ a $y' = y(1 + e_y)$ jsou nepřesné hodnoty reálných čísel x a y s relativními chybami e_x a e_y. Vypočítejme, jak se bude lišit $z = x + y$ od aproximace $z' = x' \oplus y' = (x' + y')(1 + e_z)$ s relativní chybou e_z:

$$\begin{aligned} z' - z &= x' \oplus y' - (x + y) = \Big(x' + y'\Big)\Big(1 + e_z\Big) - (x + y) \\ &= \Big(x(1 + e_x) + y(1 + e_y)\Big)\Big(1 + e_z\Big) - (x + y) \\ &= e_x x + e_y y + e_z(x + y) + e_x e_z x + e_y e_z y\,. \end{aligned}$$

Pokud byla hodnota x' získána jedinou nepřesnou operací, je zřejmě $|e_x| \leq \epsilon$. Totéž se dá říct o e_y a samozřejmě o e_z. Proto bude pro $x > 0$, $y > 0$ relativní chyba $(z' - z)/z$ největší pro maximální hodnoty dílčích chyb, tj. pro $e_x = e_y = e_z = \epsilon$. Po dosazení:

$$e_{\max} = \max\left(\frac{z' - z}{z}\right) = \frac{(x + y)(2\epsilon + \epsilon^2)}{x + y} = 2\epsilon + \epsilon^2\,.$$

Obdobně zjistíme, že relativní chyba nabývá minimální hodnoty pro $e_x = e_y = e_z = -\epsilon$:

$$e_{\min} = \min\left(\frac{z'-z}{z}\right) = \frac{(x+y)(-2\epsilon+\epsilon^2)}{x+y} = -2\epsilon+\epsilon^2\,.$$

Proto je relativní chyba e výsledku omezena číslem

$$|e| = \left|\frac{z'-z}{z}\right| \le 2\epsilon+\epsilon^2 \qquad \text{pro } z' = x' \oplus y' \quad (x>0, y>0)\,.$$

Stejným způsobem zjistíme chování pro $x > 0$, $y < 0$; abychom novou analýzu odlišili od předchozí, počítejme odhad chyby pro ekvivalentní operaci $x \ominus y$ pro $x > 0$, $y > 0$. Zjednodušme si navíc úvahu předpokladem $x - y > 0$:

$$\begin{aligned} z' = x' \ominus y' &= \big(x'-y'\big)\big(1+e_z\big) = \big(x(1+e_x)-y(1+e_y)\big)\big(1+e_z\big) \\ &= x - y + e_x x - e_y y + e_z(x-y) + e_x e_z x - e_y e_z y\,. \end{aligned}$$

Nyní bude relativní chyba zřejmě největší pro $e_x = e_z = \epsilon$, $e_y = -\epsilon$:

$$\begin{aligned} e_{\max} = \max\left(\frac{z'-z}{z}\right) &= \frac{\big(x-y+\epsilon x+\epsilon y+\epsilon(x-y)+\epsilon^2(x-y)\big)-(x-y)}{x-y} \\ &= \frac{2x\epsilon+\epsilon^2(x-y)}{x-y} = \epsilon\frac{2x}{x-y}+\epsilon^2\,, \end{aligned}$$

nejmenší pro $e_x = e_z = -\epsilon$, $e_y = \epsilon$:

$$e_{\min} = \min\left(\frac{z'-z}{z}\right) = \epsilon\frac{-2x}{x-y}+\epsilon^2\,,$$

a proto

$$|e| = \left|\frac{z'-z}{z}\right| \le \epsilon\frac{2x}{x-y}+\epsilon^2 \qquad \text{pro } z' = x' \ominus y' \quad (x>0,\ y>0,\ x-y>0)\,.$$

Opět zjišťujeme, že operace odčítaní (rozuměj kladných čísel) vede k dramatickému nárůstu relativní chyby, je-li přesný výsledek $x-y \doteq 0$. Potom se relativní chyba výsledku blíží nekonečnu. Při sčítání se sice relativní chyba také zvětšuje, ale její velikost nezávisí na hodnotách sčítanců a zůstává v řádu strojového ϵ.

Nyní však vidíme problém odčítání v lepším světle: problém s odčítáním nastává jen tehdy, jsou-li operandy pouhými aproximacemi přesných hodnot. Rozdíl přesných hodnot počítaný podle standardu IEEE 754 má velikost relativní chyby stále omezenou strojovým ϵ, tj. pro většinu aplikací je skutečně chyba zanedbatelná. Podotkněme (viz [2]), že samo odčítání hodnot x' a y' problém nezpůsobí,

zanesená relativní chyba je omezena ϵ; odečtení ale významně zvýrazní nepřesnosti, které vedly k hodnotám x' a y'.

Analogicky bychom odvodili relativní chyby pro součin $z' = x' \otimes y'$:

$$|e| = \left|\frac{z'-z}{z}\right| \leq 3\epsilon + 3\epsilon^2 + \epsilon^3 \qquad \text{pro } z' = x' \otimes y' \quad (x > 0,\ y > 0)\,,$$

pro podíl $z' = x' \oslash y'$:

$$|e| = \left|\frac{z'-z}{z}\right| \leq \epsilon \qquad \text{pro } z' = x' \oslash y' \quad (x > 0,\ y > 0)$$

a pro odmocninu $z' = \mathring{\sqrt{x'}}$

$$|e| = \left|\frac{z'-z}{z}\right| \leq \epsilon\sqrt{1+\epsilon} < \epsilon + \epsilon^2 \qquad \text{pro } z' = \mathring{\sqrt{x'}} \quad \text{a libovolné } \epsilon > 0\,.$$

I zde zjišťujeme, že operace součinu a podílu se chovají „dobře“, tj. jejich relativní chyba se zvětšuje jen mírně, pokud vůbec. Totéž platí o odmocnině. Jedinou kritickou základní operací tak zůstává rozdíl nepřesných hodnot.

Uvedená zjištění můžeme intuitivně aplikovat, aniž bychom se pouštěli do nepříjemné analýzy chyby ve stylu předchozích výpočtů.

- Výpočet $z = x^2 - y^2$ vztahem $(x \otimes x) \ominus (y \otimes y)$ může být nepřesný, protože hodnoty čtverců nedokážeme spočítat přesně. Naopak algebraicky ekvivalentní vztah $z = (x-y)(x+y)$ vyjádřený postupem $(x \ominus y) \otimes (x \oplus y)$ bude fungovat mnohem bezpečněji, budou-li x a y přesné hodnoty.

  Analýza provedená stejným způsobem jako výše uvedené (viz [2]) by odhalila, že chyba výpočtu $(x \ominus y) \otimes (x \oplus y)$ je v řádu 5ϵ (zanedbáváme vyšší mocniny ϵ), zatímco v odhadu chyby pro $(x \otimes x) \ominus (y \otimes y)$ se vyskytuje člen $\epsilon y^2/(x^2 - y^2)$, který pro $x^2 - y^2$ blízké nule výsledek zcela znehodnotí.

- Výpočet plochy S trojúhelníku se stranami a, b, c se dá vyjádřit Heronovým vzorcem

  $$S = \sqrt{s(s-a)(s-b)(s-c)} \qquad \text{kde } s = \frac{a+b+c}{2}\,.$$

  Bez hlubší analýzy okamžitě vidíme, že výpočet bude značně nepřesný, pokud se některý z rozdílů pod odmocninou bude blížit nule.

  Otázkou samozřejmě je, jak se případnému problému vyhnout. Zde žádné jednoduché recepty nejsou. Například [2] uvádí vztah

  $$S = \frac{\sqrt{(a+(b+c))(c-(a-b))(c+(a-b))(a+(b-c))}}{4} \quad \text{pro } a \geq b \geq c\,,$$

a	b	c	přímý výpočet	Kahanova úprava
10	10	10	43,30127019	43,30127020
−3	5	2	2,905	chyba
100000	99999,99979	0,00029	17,6	9,999999990
100000	100000	1,00005	50010,0	50002,50003
99999,99996	99999,99994	0,00003	chyba	1,118033988
99999,99996	0,00003	99999,99994	chyba	1,118033988
10000	50000,000001	15000	0	612,3724358
99999,99999	99999,99999	200000	0	chyba
5278,64055	94721,35941	99999,99996	chyba	0
100002	100002	200004	0	0
31622,77662	0,000023	31622,77661	0,447	0,327490458
31622,77662	0,0155555	31622,77661	246,18	245,9540000

Tabulka 4: *Srovnání přesnosti výpočtu plochy trojúhelníka o stranách a, b, c přímým dosazením do Heronova vzorce a do jeho Kahanovy úpravy.*

jehož relativní chyba je v IEEE 754 aritmetice nejvýše 11ϵ. Všechny závorky ve vztahu mají smysl a udávají pořadí operací; dá se dokázat, že dílčí chyby se pak mají tendenci vzájemně vyrušit. Představu, jaký je rozdíl mezi přímým výpočtem Heronova vztahu a výpočtem Kahanovou úpravou, ukazuje tabulka 4 převzatá z [9].

Od čtenáře tohoto textu se samozřejmě nečeká, že bude schopen podobné vzorce rutinně odvozovat. Očekává se ale, že bude schopen na první pohled rozpoznat problém ve vztahu původním.

- Výpočet kořenů x_1, x_2 kvadratické rovnice $ax^2 + bx + c$ pro $a > 0$ se dá vyjádřit známými vztahy

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \qquad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}\,,$$

respektive po rozšíření zlomkem $(-b \pm \sqrt{b^2 - 4ac})/(-b \pm \sqrt{b^2 - 4ac})$

$$x_1 = \frac{2c}{-b - \sqrt{b^2 - 4ac}} \qquad x_2 = \frac{2c}{-b + \sqrt{b^2 - 4ac}}\,.$$

Ve všech vztazích se opakují dva typy rozdílů: „pod odmocninou“ ($b^2 - 4ac$) a „s odmocninou“ ($-b \pm \sqrt{\ldots}$). Typu „pod odmocninou“ se bohužel nevyhneme, viz [2]. Pokud ale bude $b \doteq \pm\sqrt{\ldots}$, můžeme si vybrat mezi alternativními vzorci pro x_1, resp. x_2, a vyhnout se nebezpečnému rozdílu „s odmocninou“ vedoucímu k téměř nulovému výsledku.

Kromě zásadní nepřesnosti způsobené typicky rozdílem téměř stejných (nepřesných) hodnot nám zbývají nepřesnosti způsobené ostatními operacemi, tj. součtem, součinem, podílem a odmocninou. Sice jsme je označili za *bezpečné*, to ovšem neznamená, že je můžeme ignorovat úplně.

Například víme, že relativní chyba součtu $x_1 + x_2$ dvou přesných kladných hodnot x_1, x_2 je omezena strojovým ϵ. Dá se snadno ukázat, že součet nepřesné hodnoty $x_1 \oplus x_2$ s přesnou kladnou hodnotou x_3 má už relativní chybu 2ϵ. Podobně se pak dá zjistit, že součet kladných sčítanců

$$\Big(\big(([x_1 \oplus x_2] \oplus x_3) \cdots\big) \oplus x_{n-1}\Big) \oplus x_n$$

má již relativní chybu řádově $(n-1)\epsilon$, viz [2]. To může být značný problém, je-li počet sčítanců velký.

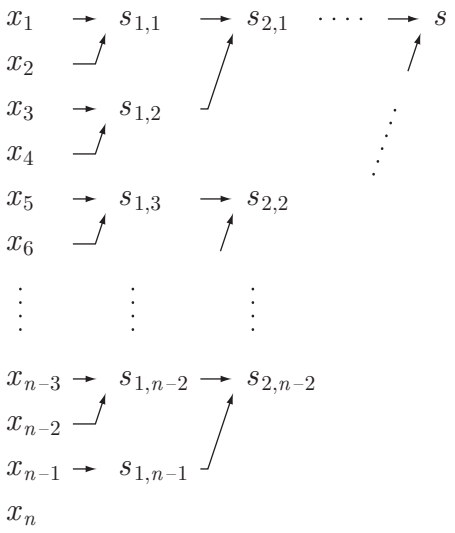
V následujících ukázkách si na problému součtu kladných $x_1 + x_2 + \cdots + x_n$ předvedeme základní techniky, které se pro omezení chyby běžně používají.

- Podstatně lepších výsledků než prosté sečtení dosahuje algoritmus založený na technice *rozděl a panuj*. Ten kladné vstupní hodnoty x_1 až x_n rozdělí na dvě přibližně stejně velké skupiny, rekurzivně vypočítá jejich součty a tyto dílčí součty sečte. Výpočet sumy je tedy přeuspořádán do podoby vyváženého binárního stromu (viz obr. 8). Jestliže jsou vstupní hodnoty x_1 až x_n přesné, potom dílčí součty $s_{1,1}$ až $s_{1,n-1}$ jsou zatíženy relativní chybou ϵ, součty $s_{2,1}$ až $s_{2,n-2}$ chybou v řádu 2ϵ, součty $s_{3,1}$ až $s_{3,n-3}$ chybou v řádu 3ϵ atd. Součet $s_{i,j}$ je tedy zatížen relativní chybou v řádu $2^{i-1}\epsilon$. Poslední dílčí součet s, tj. finální výsledek, je proto zatížen chybou $\lceil \log_2 n \rceil \epsilon$, kde multiplikativní faktor je dán hloubkou stromu o n listech.

- Odhad chyby, který jsme doposud dělali, zkoumá nejhorší možný případ; je tedy značně pesimistický. Víme sice, že že výsledky základních operací jsou zaokrouhleny a jejich relativní chyba je menší než strojové ϵ. Na druhou stranu ale také tušíme, že k největší chybě dojde jen někdy, a dokonce že některé operace mohou proběhnout přesně; přesné jsou například násobení nulou nebo změna znaménka. Zkoumejme tedy přesnost operací podrobněji.

  Algoritmus sčítání (resp. odčítání) dvou čísel s plovoucí desetinnou čárkou začíná, jak známo, úpravou obou sčítanců. Po úpravě mají oba sčítance stejný exponent; úprava se dotkne sčítance s menším exponentem. Je tedy zřejmé, že menší ze sčítanců ztratí nejméně významné bity; tato ztráta bude tím větší, čím budou exponenty sčítanců před úpravou rozdílnější. Dá se proto očekávat, že součet (resp. rozdíl) dvou přibližně stejných čísel bude přesnější.

  Chybu součtu kladných čísel x_1, x_2, …, x_n můžeme proto vylepšit následujícím algoritmem: z posloupnosti odebereme dvě nejmenší čísla, sečteme



Obrázek 8: *Schéma součtu* $s = x_1 + x_2 + \cdots + x_{n-1} + x_n$ *ve stromovém uspořádání.*

je, a tento součet do posloupnosti vrátíme. Jakmile v posloupnosti zbyde jediné číslo, algoritmus končí.

Algoritmus můžeme efektivně naprogramovat, ukládáme-li členy součtu do min-haldy. Algoritmus můžeme samozřejmě rozšířit na součet kladných i záporných čísel; stačí nejdřív sečíst všechna kladná, pak všechna záporná čísla a dílčí součty sečíst.

Dá se očekávat, že chyba takto získaného součtu bude menší než chyba součtu získaného přímým postupem. Bohužel není snadné analyzovat, jak velká tato chyba bude; bližší informace lze nalézt v [13], kap. 6.3. To je nepříjemné zejména tehdy, je-li výsledek vstupem další aritmetické operace; celkovou chybu pak můžeme těžko odhadnout.

Přesto má technika s min-haldou své opodstatnění – dá se použít jako užitečný stavební kámen při konstrukci přesných algoritmů, které chybu jen neodhadují, ale přímo ji vyčíslují a pracují s ní (příklad takového algoritmu si ukážeme za chvíli). Jestliže totiž přesný algoritmus vyhodnocuje operaci, jejíž chyba je malá nebo dokonce nulová, má mnohem snazší práci, než když je velikost chyby značně proměnlivá.

Již jsme naznačili, že některé operace s čísly s plovoucí desetinnou čárkou mohou vyjít přesně. Buďme konkrétní. Známé Sterbenzovo lemma (viz např. [2, 13]) ukazuje, že

$$x \ominus y = x - y \qquad \text{pro } \frac{y}{2} \le x \le 2y\,. \tag{7}$$

Bohužel neexistují podobně jednoduché předpoklady, které by zaručily přes-

nost sčítání (viz [9]). Sice se dá ukázat (viz [5]), že platí

$$x \oplus y = x + y \qquad \text{pro} \quad |x+y| \leq |x| \quad \wedge \quad |x+y| \leq |y|$$

při bližším zkoumání ale zjistíme, že jde jen o mírně rozvedené Sterbenzovo lemma.

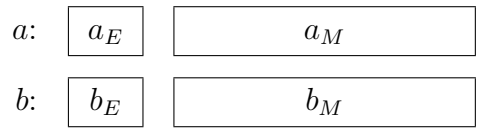
Algoritmus násobení a dělení má snadnou práci, je-li jeden z argumentů ve tvaru 2^k pro celočíselné k – celá práce spočívá v úpravě exponentu druhého argumentu. Pokud nedojde k přetečení nebo podtečení exponentu, je výsledek operace přesný.

Konečně se dá ukázat, že i obecný součin dvou čísel s pohyblivou řádovou čárkou o mantisách délky p může být za jistých okolností přesný: pokud je posledních a bitů mantisy čísla x a posledních b bitů mantisy čísla y nulových, a navíc platí $a + b \geq p$, potom $x \otimes y = x \times y$ (viz např. [13]).

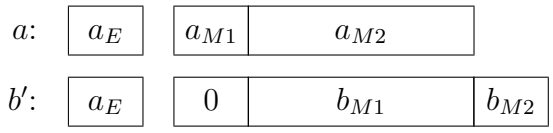
- Posledním základním postupem pro omezení chyby výsledku je sledování přesné hodnoty chyby a reakce na ni, nikoliv jen na její (pesimistické) odhady. Ačkoliv jde o metodu nejspolehlivější, jde také o metodu jednoznačně nejsložitější, protože vyžaduje poměrně složitou analýzu výpočtu. Ukažme si příklad.

  Kvůli našemu problému sumy $x_1 + x_2 + \cdots + x_n$ napřed analyzujme, jak vlastně probíhá součet dvou čísel a, b pro $a > b > 0$.

  Vstupem operace $\oplus$ je tedy číslo a vyjádřené exponentem a_E a mantisou a_M délky p bitů. Číslo b je vyjádřeno exponentem $b_E \leq a_E$ a mantisou b_M stejné délky p. Obrázkem:



  Před sečtením mantis je třeba upravit čísla na společný exponent a_E. Bity původních mantis a_M a b_M formálně rozdělíme na části a_{M1}, a_{M2} a b_{M1}, b_{M2}, protože bude třeba mantisu b_M posunout o $a_E - b_E$ bitů doprava. Bitová délka b_{M2} proto bude $a_E - b_E$, bitová délka b_{M1} pak $p - (a_E - b_E)$. Upravíme tedy číslo b na reprezentaci b':



  Sečteme-li $a \oplus b'$, dostaneme přibližnou hodnotu součtu s mantisou délky p. Pro zjednodušení předpokládejme, že mezi jednotlivými částmi nedošlo k přesunu jedničky (carry). Rovněž ignorujme vliv zaokrouhlení podle standardu IEEE 754; zaokrouhlení je pochopitelně dáno hodnotou b_{M2}.

a:	a_E	a_{M1}	a_{M2}
b':	a_E	0	b_{M1} b_{M2}
$s' = a \oplus b$:	a_E	a_{M1}	$a_{M2} + b_{M1}$

Chyba součtu je zřejmě dána „ztracenými bity“ uloženými v číslu b_{M2}. K jejich obnovení si napřed vytvořme číslo t, jehož mantisa je b_{M1}:

$s' = a \oplus b$:	a_E	a_{M1}	$a_{M2} + b_{M1}$
a:	a_E	a_{M1}	a_{M2}
$t = s' \ominus a$:	a_E	0	b_{M1}

Poznamenejme, že čísla a a s' jsou reprezentována stejným exponentem a_E, a proto podle Sterbenzova lemmatu (7) na straně 36 je výpočet $s' \ominus a$ přesný, tj. $s' \ominus a = s' - a$.

Jelikož standard IEEE 754 uchovává čísla přednostně s normalizovanou mantisou, bude vlastně číslo t graficky vypadat takto:

$t = s' \ominus a$:	b_E	b_{M1}	0

Nyní již číslo b_{M2} snadno obnovíme výpočtem čísla $\hat{e}$. Opět si uvědomíme, že výpočet probíhá podle Sterbenzova lemmatu přesně.

b:	b_E	b_{M1}	b_{M2}
t:	b_E	b_{M1}	0
$\hat{e} = b \ominus t$:	b_E	0	b_{M1}

I přes různá zjednodušení jsme se dostali k podstatě Dekkerova lemmatu (viz např. [10, 2]), které platí pro libovolná čísla a, b za podmínky $|a| \geq |b|$:

$$a + b = \underbrace{\Big[a \oplus b\Big]}_{s'} + \underbrace{\Big[b \ominus \Big([a \oplus b] \ominus a\Big)\Big]}_{\hat{e}} \tag{8}$$

kde číslo s' je zaokrouhlený součet a číslo $\hat{e}$ je přesná hodnota absolutní chyby. Důkaz tvrzení samozřejmě musí vzít v potaz i zjednodušení, která

jsme kvůli jasnosti výkladu zavedli; zájemci jej mohou najít např. v [2, 12, 19]. Mimochodem: na straně 62 ukážeme podobný vztah pro vyjádření $a \times b$.

Znalost velikosti chyby využívá Kahanův algoritmus pro součet $a = x_1 + x_2 + \cdots + x_n$, viz např. [2, 10]:

```
1  a = x[1];                          /* průběžný součet */
2  e = 0;                             /* chyba výpočtu */
3  for (j = 2; j <= N; j++) {
4    b  = x[j] + e;                   /* korigovaný sčítanec */
5    s' = a + b;                      /* odhad součtu */
6    e  = b - (s' - a);               /* chyba součtu */
7    a  = s';
8  }
```

Činnost algoritmu je jednoduchá. Před první iterací se nastaví průběžná hodnota součtu a na hodnotu x_1, dosud zjištěná chyba operací e je samozřejmě nulová. Při první iteraci se spočítá (protože $e = 0$)

$$\begin{aligned} s' &= a \oplus b \\ e &= b \ominus (s' \ominus a) = b \ominus \Big((a \oplus b) - a \Big) , \end{aligned}$$

čili zaokrouhlená hodnota součtu s' a přesná hodnota chyby e. Na řádku 7 aktualizujeme hodnotu průběžného součtu a a pokračujeme další iterací.

V každé další iteraci se na řádce 4 pokusíme dosud zjištěnou chybu e napřed přidat k dalšímu sčítanci. Tato operace samozřejmě není přesná; je to ale lepší než nic. Smysl řádek 5 až 7 je stejný jako v první iteraci.

Pozorný čtenář si také jistě všiml, že výpočet chyby na řádce 6 není korektní, protože není zaručena podmínka Dekkerova lemmatu $|a| \geq |b|$. I přesto se dá dokázat (viz [2]), že při použití Kahanova algoritmu platí

$$\begin{aligned} x_1 \oplus x_2 \oplus \cdots \oplus x_n = x_1(1 + e_1) + x_2(1 + e_2) + \cdots + x_n(1 + e_n) \\ + O(n\epsilon^2)(|x_1| + |x_2| + \cdots + |x_n|) \end{aligned}$$

kde $|e_i| \leq 2\epsilon$. Jsou-li všechny sčítance kladné, je zřejmě relativní chyba součtu nanejvýš rovna 2ϵ (při zanedbání vyšších mocnin ϵ).

Uvedený postup je zdaleka nejlepší ze všech uvedených; připomeňme, že naivní implementace sumy je zatížena relativní chybou v řádu $(n-1)\epsilon$, sčítání metodou rozděl a panuj chybou $\lceil \log_2 n \rceil \epsilon$, součet seřazených sčítanců odhad chyby neposkytuje, zatímco Kahanův algoritmus je zatížen chybou v řádu 2ϵ (ve všech případech jsme zanedbali vyšší mocniny ϵ). Kahanův algoritmus přitom vyžaduje jen $4\times$ více aritmetických operací než rozděl a panuj nebo naivní přístup a jeho časová složitost je lineární (naproti tomu

algoritmus s řazením je $O(n \log n)$). Cenou jeho a algoritmů podobných je – a to je pro nás podstatné – jejich netriviální vymýšlení.

Ještě než problematiku opustíme, povšimněme si důležitého detailu. Kahanův algoritmus (a jiné obdobné) je založen na důkladném porozumění vlastností aritmetiky podle IEEE 754 a bude fungovat jen tehdy, bude-li počítač provádět přesně to, co algoritmus předepisuje, a navíc v předepsaném pořadí. Jakmile se ovšem do hry vloží nedbale napsaný překladač se „sofistikovanou“ optimalizací kódu, který usoudí, že

$$e = b - (s' - a) = b - \Big((a - b) - a\Big) = 0\,,$$

přijde veškerý důmysl algoritmu vniveč. Podobné algebraické úvahy dělají překladače poměrně často a nutno přiznat, že obvykle jsou ku prospěchu věci; při programování podobných kritických algoritmů je ale nutné být ve střehu.

Na závěr podkapitoly přidejme několik užitečných vztahů, které se uplatní při běžném programování.

Axiomy čísel s plovoucí desetinnou čárkou Již několikrát jsme narazili na odlišnost chování přesné a nepřesné aritmetiky. Asi nejkřiklavějším případem je neplatnost asociativního zákona, čili např.

$$x \oplus (y \oplus z) \neq (x \oplus y) \oplus z\,.$$

Podobně neplatí asociativní zákon pro násobení. Neplatí ani distributivní zákon, tj.

$$x \otimes (y \oplus z) \neq (x \otimes y) \oplus (x \otimes z)\,.$$

Konečně zejména výpočty založené na lineární algebře je vhodné vědět, že neplatí ani Cauchy-Schwarzova nerovnost:

$$(x_1^2 + x_2^2 + \cdots + x_n^2)(y_1^2 + y_2^2 + \cdots + y_n^2) \not\geq (x_1 y_1 + x_2 y_2 + \cdots + x_n y_n)^2\,.$$

Na druhou stranu mnoho jiných zákonů platí; kompletní výčet podává [12]:

$$
\begin{aligned}
x \oplus y &= y \oplus x \\
x \ominus y &= x \oplus (-y) \\
-(x \oplus y) &= (-x) \oplus (-y) \\
x \oplus y = 0 \quad &\Leftrightarrow \quad x = -y \\
x \oplus 0 &= x \\
x \le y \quad &\Rightarrow \quad x \oplus z \le y \oplus z \\
x \otimes y &= y \otimes x \\
(-x) \otimes y &= -(x \otimes y) \\
1 \otimes x &= x \\
x \otimes y = 0 \quad &\Leftrightarrow \quad x = 0 \quad \vee \quad y = 0 \\
(-x) \oslash y &= x \oslash (-y) = -(x \oslash y) \\
0 \oslash x &= 0 \\
x \oslash 1 &= x \\
x \oslash x &= 1
\end{aligned}
$$

Korektní ϵ-testy Víme, že testovat dvě čísla s plovoucí desetinnou čárkou na rovnost je nemoudré (až na případ, kdy si uvědomujeme, co přesně děláme). Trochu lepší implementací testu $x = y$ může být naivní ϵ-test v podobě $|x - y| < \hat{e}$, kde testujeme velikost absolutní chyby; její mezní velikost ale musíme kvalifikovaně určit. Jelikož je IEEE 754 aritmetika spíše orientovaná na udržení relativní chyby, máme i alternativu $|x-y| < e|x|$; mezní velikost relativní chyby e se určuje o něco lépe.

Stejným způsobem bychom potřebovali zavést i ostatní testy jako $x < y$ nebo $x \le y$ – je totiž zřejmé, že pro blízká x a y nebude standardní test relace úplně spolehlivý. Například „nepřesnou“ variantu testu $x < y$ bychom mohli zavést jako $x + \hat{e} < y$, případně $x(1 + e) < y$.

Zásadní nevýhodou výše uvedených testů je jejich nejasné chování, jakmile se začnou spojovat logickými operátory a z výsledků několika testů se usuzuje na další vlastnosti. Například očekáváme, že pro $x < y$ a $y < z$ platí $x < z$. Platí to ale i pro výše uvedené testy, jsou-li implementované IEEE 754 aritmetikou?

Knuth [12] proto uvádí takové relace, které mají definované chování. Jsou-li x a y definovány mantisami x_M, y_M a exponenty x_E, y_E, potom definuje:

$$
\begin{aligned}
x \prec_e y \quad &(x \text{ je určitě menší než } y) \quad \Leftrightarrow \quad y - x > e \max(2^{x_E}, 2^{y_E}) \\
x \succ_e y \quad &(x \text{ je určitě větší než } y) \quad \Leftrightarrow \quad x - y > e \max(2^{x_E}, 2^{y_E}) \\
x \approx_e y \quad &(x \text{ je v podstatě rovno } y) \quad \Leftrightarrow \quad |x - y| \le e \min(2^{x_E}, 2^{y_E}) \\
x \sim_e y \quad &(x \text{ je přibližně rovno } y) \quad \Leftrightarrow \quad |x - y| \le e \max(2^{x_E}, 2^{y_E})
\end{aligned}
$$

Zaručeně pak platí mnoho důsledků, kompletní výčet viz [12]:

$$\begin{array}{rcl} x \prec_e y & \Rightarrow & y \succ_e x \\ x \approx_e y & \Rightarrow & y \sim_e x \\ x \prec_e y & \Rightarrow & x < y \\ x \approx_{e1} y \quad \wedge \quad y \approx_{e1} z & \Rightarrow & x \sim_{(e1+e2)} z \end{array}$$

Pokud potřebujeme porovnávat dvě čísla s plovoucí desetinnou čárkou a nechceme nebo nemůžeme si dovolit podrobnou numerickou analýzu, jsou uvedené testy rozumným kompromisem. Po chvíli pátrání lze obvykle tyto relace najít implementované v různých knihovnách, např. pro C++ je implementuje Boost Test Library.

Odhad chyby součtu Jak již víme z Dekkerova lemmatu (8), pro pro $|a| \geq |b|$ můžeme vypočítat $a + b = \underbrace{(a \oplus b)}_{s} + e$ algoritmem

```
1  s  = a + b;      /* aproximace součtu */
2  tb = s - a;      /* oříznuté číslo b */
3  e  = b - tb;     /* celková chyba */
```

Verzi, která nevyžaduje podmínku $|a| \geq |b|$, uvádí [12, 5]:

```
1  s  = a + b;      /* aproximace součtu */
2  tb = s - a;      /* oříznuté číslo b */
3  eb = b - tb;     /* první část chyby */
4  ta = s - tb;     /* speciálně oříznuté číslo a */
5  ea = a - ta;     /* druhá část chyby */
6  e  = ea + eb;    /* celková chyba */
```

Oba algoritmy tiše předpokládají, že nedojde k přetečení ani podtečení exponentu, respektive že žádné číslo nebude v nenormalizovaném tvaru.

Jak uvádí [5], lze první algoritmus používat i bez podmínky $|a| \geq |b|$, pokud napřed velikosti sčítanců otestujeme a pro výpočet je případně prohodíme. Test na absolutní hodnotu pak lze asi nejrychleji napsat jako

```
1  if ((a>b) == (a>-b))
```

Bohužel ale nelze apriori rozhodnout, který z uvedených algoritmů bude rychlejší; dá se ale očekávat, že v masivně paralelním prostředí typu GPU nebo na procesorech s instrukční pipeline se bude lépe chovat verze bez podmíněného příkazu.

4 Geometrie světa s omezenou přesností výpočtu

Viděli jsme, že v aritmetice s omezenou přesností neplatí mnohé aritmetické zákony, například asociativní nebo distributivní. Dá se tedy očekávat, že geometrie reprezentovaná nepřesnými čísly se nebude řídit týmiž zákony jako geometrie přesná.

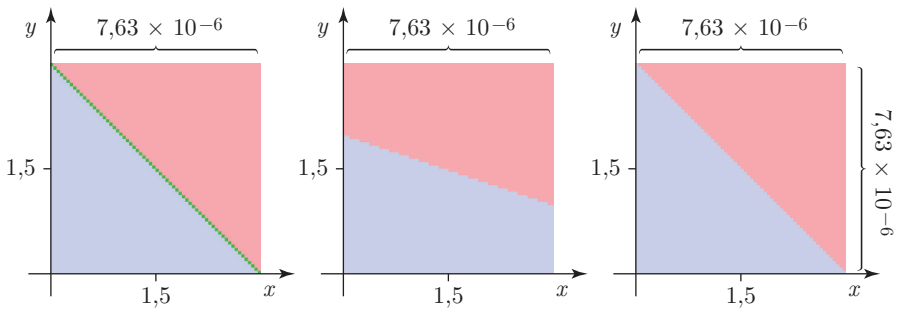
Hlavním problémem nepřesné geometrie je nejasná definice základních pojmů. V analytické geometrii je to snadné: bod v rovině je vyjádřen dvojicí čísel $[x, y]$, přímka je tvořena všemi body, pro které platí $ax + by + c = 0$ (je-li $a^2 + b^2 > 0$) a tak dále.

Abychom zjistili, jak jednoduchá definice přímky obstojí v nepřesné aritmetice, zkusíme jednoduchý test. Vybereme náhodně přímku a otestujme, zda na ní najdeme bod:

- náhodně vygeneruj $a \in [1; 2)$, $b \in [1; 2)$, $c \in [1; 2)$,
- náhodně vygeneruj $x \in [1; 2)$,
- urči $y = -(a \otimes x \oplus c) \oslash b$,
- vypočti $h = a \otimes x \oplus b \otimes y \oplus c$.

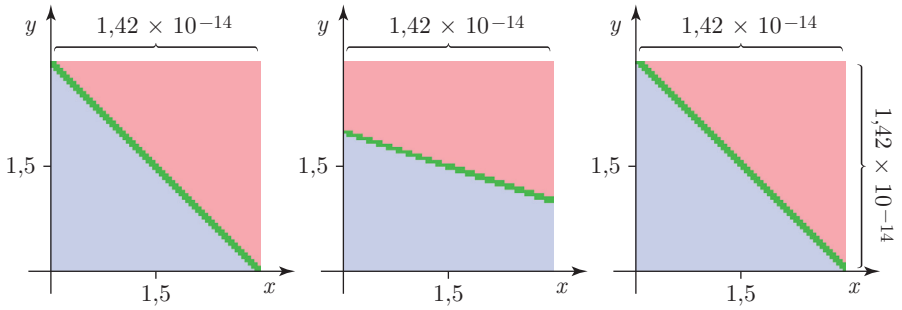
Výsledek testu je poměrně tristní. Z 10^9 pokusů byl jen 70× nalezen bod $[x, y]$, pro který platí $h = a \otimes x \oplus b \otimes y \oplus c = 0$, tj. úspěšnost je menší než miliontina procenta. V ostatních případech byla hodnota h v polovině případů kladná, v polovině záporná.

Podívejme se proto na vyhodnocení testu „bod na přímce“ podrobněji: zvolme si taková a, b, c, aby přímka $ax + by + c = 0$ procházela bodem $[1{,}5; 1{,}5]$ a zkoumejme, jak se bude měnit znaménko výrazu $a \otimes x \oplus b \otimes y \oplus c$ pro x, y v jeho okolí. Výsledek zobrazíme zelenou (vypočtená hodnota je nulová), modrou (hodnota je záporná) nebo červenou (hodnota je kladná). Připomeňme si, že mantisa je celé číslo, a proto můžeme snadno testovat všechna reprezentovatelná čísla v okolí zvoleného. Testujme tedy 64×64 bodů v okolí bodu $[1{,}5; 1{,}5]$:



$x + y - 3 = 0$ $\quad x/3 + y - 2 = 0$ $\quad 1{,}0101x + y - 3{,}0151 = 0$

V jednoduché přesnosti zjišťujeme, že pro $a = 1$, $b = 1$ najdeme taková x, y, pro která je test splněn. Výsledek experimentu je docela hezký . Už ale pro mírně odlišné koeficienty takové body nemusíme najít. Tentýž experiment provedený ve dvojité přesnosti ale ukazuje, že test je splněn v poměrně širokém pásu kolem ideální přímky:



$x + y - 3 = 0$ $\qquad$ $x/3 + y - 2 = 0$ $\qquad$ $1{,}0101x + y - 3{,}0151 = 0$

Odpověď na otázku, co je vlastně přímka, se začíná komplikovat. Uvědomme si navíc, že náš experiment měl velmi jednoduchou práci, neboť explicitně znal hodnoty koeficientů a, b, c. Takové štěstí obvykle nemíváme; častěji máme přímku (či úsečku) zadanou body $A = [a_x, a_y]$, $B = [b_x, b_y]$. Bod $[x, y]$ na ní leží tehdy, platí-li

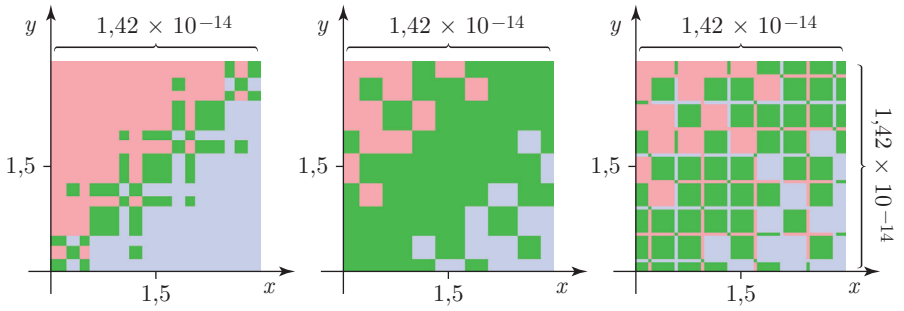
$$\begin{vmatrix} 1 & x & y \\ 1 & a_x & a_y \\ 1 & b_x & b_y \end{vmatrix} = 0. \tag{9}$$

K vyhodnocení determinantu můžeme přistoupit přinejmenším dvojím způsobem:

$$\begin{aligned} h_1 &= \begin{vmatrix} 1 & x & y \\ 1 & a_x & a_y \\ 1 & b_x & b_y \end{vmatrix} = \begin{vmatrix} a_x - x & a_y - y \\ b_x - x & b_y - y \end{vmatrix} \\ &= (a_x - x)(b_y - y) - (b_x - x)(a_y - y) \end{aligned}$$

$$\begin{aligned} h_2 &= (a_x - x)(b_y - y) - (b_x - x)(a_y - y) \\ &= x(a_y - b_y) + y(b_x - a_x) + (a_x b_y - a_y b_x) \end{aligned}$$

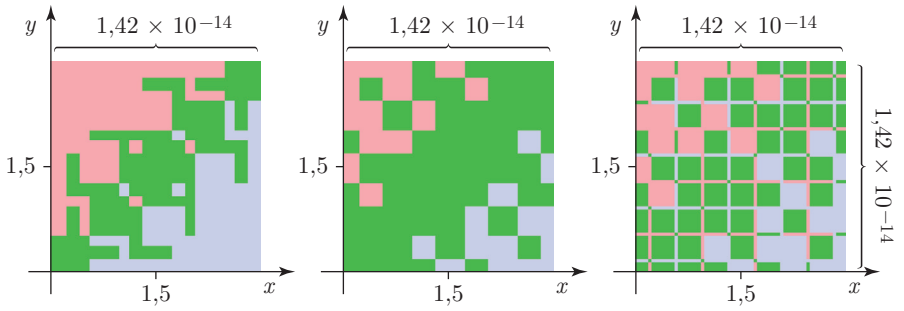
Zkoumejme, zda bod $[x, y]$ v okolí bodu $[1{,}5; 1{,}5]$ leží na dané úsečce; budeme testovat opět 64×64 bodů vyjádřených nejbližšími reprezentovatelnými čísly. Pro vyhodnocení použijme vztah h_1:



A: [12; 12] *B*: [24; 24] *A*: [12; 12] *B*: [13; 13] *A*: [12; 12] *B*: [13,1; 13,1]

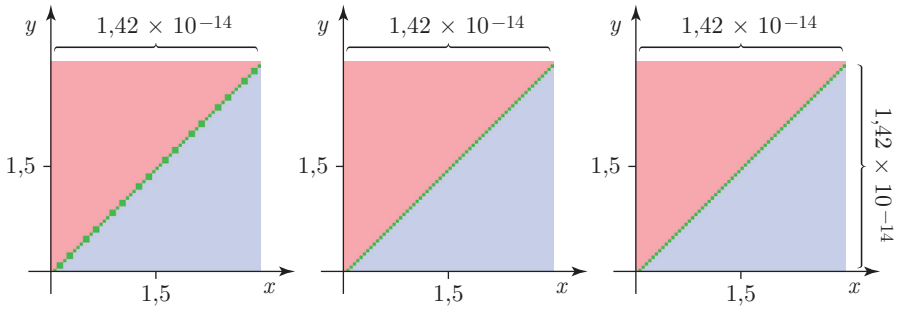
Úsečka AB by teoreticky měla procházet levým dolním a pravým horním rohem vizualizované plochy. Povšimněme si, že v prvním případě se může stát, že h_1 se vyhodnotí kladné místo správného záporného (a naopak). V případě druhém se sice vždy vyhodnotí znaménko buď dobře, nebo je $h_1 = 0$, ale oblast špatně vyhodnoceného znaménka se zvětšila. Třetí případ je značně chaotický, což je dáno mimo jiné tím, že hodnota 13,1 má plně obsazenou mantisu. Podrobnější diskuse chování je uvedena v [4].

Nezkušený programátor by se možná pokusil situaci vyřešit naivním ϵ testem, čili za nulu by považoval libovolné h_1 menší než zvolený práh. Příklad, kde byl práh zvolen 5×10^{-14} hovoří za své: v prvním případě se jen plocha špatně vyhodnocovaných výsledků zvětšila, ve zbývajících případech se nestalo nic. Případné zvětšování prahu situaci těžko zachrání, viz následující vizualizace.



A: [12; 12] *B*: [24; 24] *A*: [12; 12] *B*: [13; 13] *A*: [12; 12] *B*: [13,1; 13,1]

Všímavý programátor naopak okamžitě uvidí, že ve výpočtu h_1 se odečítají dvě nepřesná čísla, což je vždy problematické. Výpočet h_2 na první pohled takový nedostatek neobsahuje. Skutečně, pro uvedené situace je výsledek s h_2 podstatně lepší:



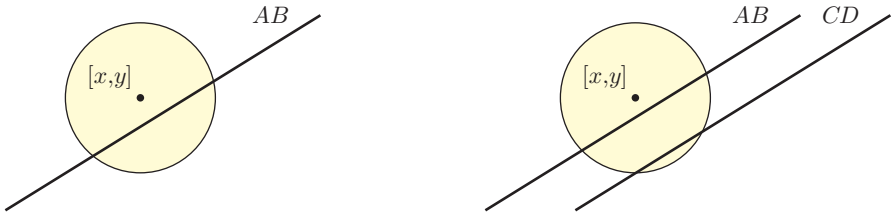
A: [12; 12] B: [24; 24] A: [12; 12] B: [13; 13] A: [12; 12] B: [13,1; 13,1]

Čtenář by neměl nabýt dojmu, že výpočet h_2 je bezproblémový! Nanejvýš můžeme říci, že pro uvedené případy a sledované oblasti se h_2 chová lépe. Podrobnější diskusi kvality výsledku na vyhodnocení h_1 uvádí [4]; pro nás je důležité, že neexistuje jednoduchý zápis vyhodnocení determinantu (9), který by byl zcela bezproblémový.

Uvedené experimenty nám především odhalily další potíž geometrie reprezentované nepřesnou aritmetikou. Nejenom, že není úplně zřejmé, jak například definovat bod na přímce; smysluplnost definice vždy navíc silně závisí na konkrétní implementaci.

Zkusme nyní rozvíjet úvahy a sledovat důsledky rozhodnutí.

Dejme tomu, že jsme vybrali nějakou „rozumnou“ implementaci pro detekci bodu na přímce. Taková implementace by zřejmě měla „bod“ $[x, y]$ detekovat na přímce, leží-li v jejím těsném sousedství ve smyslu omezené přesnosti. Dvojici $[x, y]$ bychom si potom mohli představovat jako jakousi miniaturní oblast kolem bodu $[x, y]$. Rozhodnutí „$[x, y]$ leží na přímce AB“ bychom pak interpretovali jako „miniaturní oblast kolem bodu $[x, y]$ je protnuta přímkou AB“. Na první pohled rozumná konstrukce pak ale vede k situaci, kdy umíme najít bod společný rovnoběžkám AB a CD:

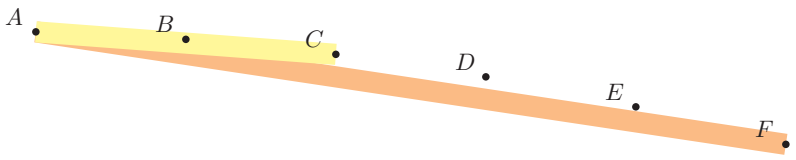


Podobně bychom mohli zjistit, že dvě různoběžky mají více než jeden průsečík a tak dále.

Rovněž alternativní postup, podle kterého bychom „nepřesnou přímku“ chápali jako jakýsi úzký pás kolem přesné přímky, nevede ke konzistentní geometrické představě. Dospěli bychom například k tvrzení, že dvě rovnoběžky mají nekonečně mnoho průsečíků; blíže viz [8].

Zásadní problém „tlustých“ geometrických primitiv uvádí [7]. Pokud kvůli představě nepřesných geometrických primitiv usoudíme, že body A, B, C jsou

kolineární a že body B, C, D jsou rovněž kolineární, mělo by vyplývat, že všechny body A, B, C, D jsou kolineární; a tak dále. To pochopitelně vede úsudku, že body A až F leží na téže přímce. Na druhou stranu tatáž geometrická představa tvrdí, že body A, D, F na jedné přímce neleží:



„Geometrie tlustých primitiv“ je tedy vnitřně nekonzistentní, respektive řídí se natolik odlišnými pravidly, že jako model přesné geometrie rozhodně nemůže sloužit.

Chtě nechtě musíme opustit představu, že mírnou modifikací pojmů „bod“ nebo „přímka“ zázračně odstraníme problémy. Geometrické útvary a vztahy mezi nimi musíme popisovat přesnými vztahy známými z analytické geometrie a s počítačovými nepřesnostmi se musíme vypořádat jinak.

Především si musíme položit otázku, co vlastně od výpočtu (programu) očekáváme. Odpovědět můžeme různým způsobem (viz [5]), a podle toho budeme muset přemýšlet nad implementačními možnostmi.

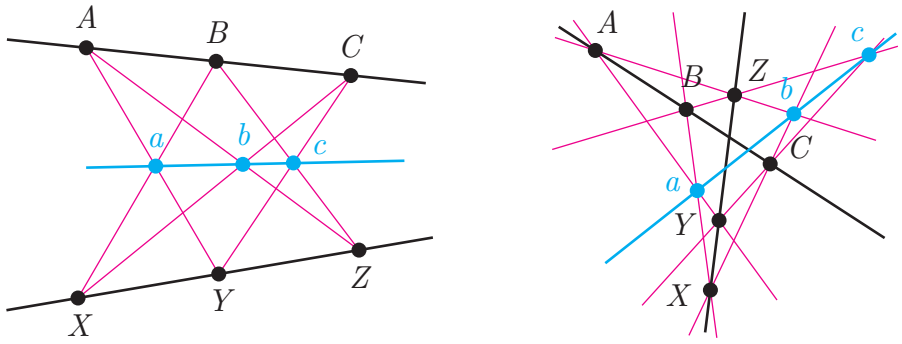
- Výsledky chceme přesné. Dá se očekávat, že za přesnost budeme muset zaplatit značnou cenu, a to jak v podobě náročné implementace, tak v podobě výpočetního času. Přesné výpočty, obzvlášť takové, které mají na základě vstupních dat něco konstruovat, se musí spoléhat na výpočty v symbolické podobě podobně, jako je dělají systémy počítačové algebry typu Mathematica, Maple nebo Maxima. Jelikož se při práci s nelineárními útvary (elipsami, spline křivkami apod.) může snadno stát, že systém dojde ke vztahu, který nelze symbolicky řešit (například hledání kořene polynomu stupně vyššího než 4), omezují se přesné algoritmy víceméně jen na práci s lineárními útvary.

  Ultimativní požadavek na přesné výstupy také samozřejmě klade ultimativní požadavky na přesnost vstupu. Zejména aplikace zpracovávající naměřené údaje nemají přesné vstupy nikdy; pak je samozřejmě legitimní otázka, jaké výhody vlastně přesná práce s nepřesnými daty přináší.

- Výsledky chceme přesné, ale uznáváme, že vstupy mohou být nepřesné. Výstupem programu tedy bude výsledek, který bychom dostali přesným výpočtem s mírně odlišným vstupem, než jaký jsme zadali. Takovým programům říkáme *robustní*. Je-li navíc zvoleným výpočetním postupem zaručeno, že odchylka vstupu skutečného a „mírně odlišného“ bude malá, hovoříme o programu *stabilním*.

Téměř neznatelná změna formulace požadavků vede k dalekosáhlým důsledkům. Uveďme si dva příklady.

První příklad: pokud zadáme tři kolineární body A, B, C přesnému programu, musí je vyhodnotit jako kolineární. Robustní program buď vyhodnotí, že bod C leží na přímce AB, nebo že leží na jedné či druhé straně od ní. Všechna tato rozhodnutí jsou v rámci formulace robustnosti korektní. Robustní program je ale ve veškerých svých úvahách konzistentní, nemůže se mu tedy stát, aby z vyhodnocení poloh jiných vstupních bodů usoudil na opak. To se snadno řekne, hůře se však udělá – obrázek 9 (převzato z [7]) ukazuje netriviální situaci, kdy z kolinearity jedné a kolinearity druhé skupiny bodů plyne kolinearita třetí skupiny bodů. Jak by si měl takovou situaci (a jiné) promyslet program, to definice robustnosti neřeší.

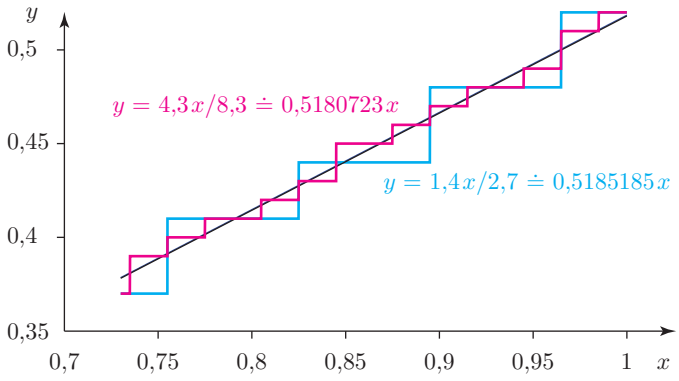


Obrázek 9: *Dva příklady Pappovy věty o šestiúhelníku. Jsou-li body A, B, C kolineární, body X, Y, Z kolineární, potom body a, b, c jsou rovněž kolineární, kde $a = AY \cap BX$, $b = AZ \cap CX$, $c = CY \cap BZ$. Poněkud záhadný název věty plyne z její ekvivalentní formulace, která hovoří o šestiúhelníku $aBZbCY$ a vlastnostech bodů A, X, c (viz příklad vpravo). Pro naše účely je ale důležité uvědomit si, že kolinearita bodů a, b, c není úplně zřejmá.*

Druhý příklad: mnoho geometrických algoritmů příliš nepracuje se singulárními případy; například v konstrukci triangulace se často neuvažuje případ tří (a více) kolineárních bodů, v konstrukci Voroného diagramu se neuvažuje případ více než tří bodů ležících na stejné kružnici a podobně. Má to svůj důvod: zatímco algoritmus bez singulárních případů bývá jednoduchý, jejich zahrnutí vede ke značným komplikacím. Například algoritmus ořezávání úsečky nekonvexním mnohoúhelníkem (viz [27]) musí v regulárním případě (ořezávaná úsečka protíná několik hran mnohoúhelníku) řešit jeden typ situace; pro řešení singulárních případů (např. hrana mnohoúhelníku leží na ořezávané úsečce) je třeba přidat ošetření dalších 16 situací. Proto se robustní algoritmy často uchylují k úskoku – předpokládají, že přesné pozice vstupních bodů by k singularitám nevedly. To v některých úlohách může vadit, v jiných nemusí.

- Výsledky chceme přesné, ale geometrické útvary se mohou od ideálních lišit. Tento ne zcela zřejmý popis si objasníme příkladem.

  Od přímky očekáváme jednak „přímost“, jednak schopnost jednoduchého rozdělení roviny na dvě části (resp. tři, zahrneme-li případ „na přímce“). Pokud ale první vlastnost zeslabíme, čili připustíme, že se „přibližná přímka“ může mírně vlnit, získáme při implementaci jistou volnost; například můžeme relativně snadno definovat přímku jako množinu bodů, pro kterou je výpočet vhodně zvolené formule roven nule. Na druhou stranu nemůžeme spoléhat na na takové vlastnosti jako na jasný počet průsečíků dvou přímek. Demonstrovat to můžeme na příkladu z [7] (kde byl použit v jiném kontextu), viz obr. 10: pokud v aritmetice se dvěma platnými desetinnými číslicemi aproximujeme dvě přímky, jež by měly být na daném intervalu prakticky identické, dostaneme dvě velmi odlišné čáry s mnoha vzájemnými průsečíky.



Obrázek 10: *Aproximace přímek* $y_1 = 1{,}4x/2{,}7$ *a* $y_2 = 4{,}3x/8{,}3$ *v aritmetice se zaokrouhlováním na dvě desetinná místa. Na intervalu* $[0{,}73; 1{,}0]$ *by měly být prakticky stejné, konkrétně* $y_1 \in [0{,}37852; 0{,}51852]$, $y_2 \in [0{,}37819; 0{,}51807]$. *Při zaokrouhlování všech operací na dvě desetinná místa ale dostáváme výrazně odlišné průběhy.*

- Samozřejmě se můžeme také rozhodnout pro implementaci, která bude tiše doufat, aby ve vstupních datech nebyly zapeklité případy. Takové implementaci říkáme *křehká*. Protože nezaručuje vůbec nic a může se znenadání zacyklit nebo může být ukončena operačním systémem, je přinejmenším slušné průběžně automaticky zálohovat uživatelovu práci, nabízet funkci „undo“ a podobně. Aniž by autor tohoto textu chtěl křehký přístup doporučovat, musíme připustit, že v nenáročných aplikacích typu interaktivní kreslicí program ocení uživatelé spíše jiné vlastnosti než dokonalou robustnost programu. Pro jakoukoliv neinteraktivní aplikaci nebo aplikaci s výstupem větším, než který může uživatel jedním pohledem posoudit, se však hazardování s křehkou implementací nevyplácí.

Podle rozhodnutí o očekávaném chování programu vybíráme implementační prostředky. Křehké programy mohou využívat běžnou naivní práci s čísly s plovoucí desetinnou čárkou; v naprosté většině případů poskytnou správné výsledky. Ostatní typy programů typicky vedou k následujícím obecným technikám.

Velká čísla. Zatímco čísla reprezentovaná běžnými typy (32bitová celá čísla, čísla podle IEEE 754 apod.) pracují s předem pevně danou přesností, u velkých čísel je přesnost volitelná a omezená pouze dostupnou pamětí. Program si tedy přesnost volí podle momentálních požadavků.

Vnitřní reprezentace velkých čísel může být různá, používají se zejména rozšířené ekvivalenty běžných datových typů a symbolické součty několika čísel reprezentovaných podle IEEE 754. Příkladem první skupiny je n-bitové celé číslo, kde n je volitelné. Konkrétním příkladem z druhé skupiny je součet $2^{100} + 2^{-100}$; pokud bychom takové číslo chtěli reprezentovat ekvivalentem čísla s plovoucí desetinnou čárkou, muselo by disponovat alespoň 201bitovou mantisou.

Celočíselné výrazy. Každé číslo s plovoucí desetinnou čárkou můžeme vyjádřit racionálním číslem. Pokud tedy budeme veškeré výpočty dělat ve formě zlomků a s (celočíselnými) čitateli a jmenovateli budeme pracovat přesně, což je snadné, můžeme teoreticky počítat beze ztráty přesnosti. Počet platných cifer čitatelů a jmenovatelů samozřejmě velmi rychle roste, proto se typicky musí zapojit také „velká čísla“. Jiným příkladem celočíselných výrazů jsou zlomky s odmocninami celých čísel. Ty využijeme jak pro jednoduchou práci s kuželosečkami, tak pro reprezentaci délek úseček.

Odhady přesnosti. Používáme-li ve výpočtu operace zaručující jistou přesnost (např. základní operace IEEE 754), můžeme analyzovat vliv nepřesností a výpočet mu přizpůsobit. Programátor se tedy může apriori rozhodnout, že daný výpočet musí proběhnout ve dvojité přesnosti, jiný výpočet musí využívat velká čísla s 200 platnými ciframi apod. Programátor musí samozřejmě brát v potaz nejhorší možný případ; dá se proto předpokládat, že ne všechny vstupní hodnoty budou vyžadovat číselnou reprezentaci zvolené přesnosti.

Alternativně si může požadovanou přesnost číselné reprezentace diktovat program sám na základě konkrétních vstupních hodnot. Může tak činit trojím způsobem, přičemž každý má své přednosti i nedostatky.

- O přesnosti se rozhoduje před výpočtem. Technika se nejlépe uplatní, jsou-li velké rozdíly mezi minimální a maximální požadovanou přesností a výpočet v základní přesnosti by téměř nikdy nedopadl dobře.
- Výpočet se provede v základní přesnosti, následně se analyzuje chyba a v případě nutnosti se výpočet přepočítá. Technika se dobře uplatní,

pokud většinou výpočet v základní přesnosti stačí a jen výjimečně se musí použít náročnější techniky. Oproti předchozímu případu je analýza chyby *po výpočtu* šikovnější tehdy, je-li možné během výpočtu v základní přesnosti střádat kritické mezivýsledky a jejich jednoduchým testem ověřit, zda je základní přesnost dostačující. Je-li třeba výpočet provést znovu, může se buď určit rovnou požadovaná přesnost číselné reprezentace, nebo se přesnost výpočtu iteračně zvyšuje tak dlouho, dokud nejsou splněna daná kritéria.

- Výpočet se provede v základní přesnosti, následně se analyzuje chyba a v případě nutnosti se spočítá korekční člen. Technika je podobná předchozí a má proti ní zřejmou výhodu: při opakovaném přepočítávání se nevyužívá práce vykonaná při výpočtu s nižší přesností. Na druhou stranu zde musí být výpočet organizovaný tak, aby šlo korekční členy snadno počítat. Dá se tedy očekávat, že metoda spoléhající na korekční členy bude implementačně nejnáročnější.

Odhady intervalů. Pokud neumíme vypočítat výsledek přesně, ale umíme určit rozsah hodnot, kde se určitě nachází, můžeme být v mnoha praktických aplikacích spokojení, a to zejména tehdy, je-li rozsah hodnot menší než jistá předepsaná tolerance. Ukažme si základní myšlenky techniky odhadu intervalů na příkladech.

Typicky umíme určit, s jakou přesností jsou určeny vstupní hodnoty. Dejme tomu že pro vstupy $x > 0$ a $y > 0$ platí $x \in [x_1, x_2]$, $y \in [y_1, y_2]$. Pak zřejmě platí $x + y \in [x_1 + y_1, x_2 + y_2]$. Podobně můžeme napsat vztahy pro ostatní elementární operace a vyjádřit, v jakém rozsahu hodnot se nachází výsledek libovolně složitého postupu. To je podstata *intervalové aritmetiky*, viz např. [21]. Při vyhodnocování intervalů můžeme navíc dobře využít standard IEEE 754, který nařizuje možnost volby typu zaokrouhlení výsledku libovolné základní operace. Pro vyhodnocování dolní meze tedy zvolíme zaokrouhlení směrem dolů, pro vyhodnocování horní meze směrem nahoru.

Intervalová aritmetika bohužel často vede k příliš pesimistickým odhadům intervalu (tedy zbytečně velkým intervalům). Například pro $x > 0$, $y > 0$, $x \in [x_1, x_2]$, $y \in [y_1, y_2]$ platí $x - y \in [x_1 - y_2, x_2 - y_1]$. Pro speciální případ $x = y$ pak dostáváme $x - x \in [x_1 - x_2, x_2 - x_1]$, zatímco samozřejmě vždy platí $x - x = 0$. Uvedený problém se snaží řešit *afinní aritmetika* (viz [21]), která určuje interval hodnot jako $x = x_0 + x_E\xi_x$, $y = y_0 + y_E\xi_y$, kde x_0, y_0 jsou střední hodnoty čísel x a y, pro ξ_x, ξ_y platí $\xi_x \in [-1, 1]$, $\xi_y \in [-1, 1]$ a čísla x_E, y_E pak určují interval, kde se hodnoty x a y určitě nacházejí. Potom samozřejmě platí $x - y = x_0 - y_0 + x_E\xi_x - y_E\xi_y$; výpočet intervalu, kam $x - y$ určitě spadá, je snadný. Naproti tomu platí $x - x = x_0 - x_0 + x_E\xi_x - x_E\xi_x = 0$, což je podstatně přesnější výsledek než s užitím intervalové aritmetiky. Afinní aritmetika tedy umí zacházet se souvislostmi

mezi argumenty výrazu.

Znaménkové testy. Většina algoritmů obsahuje větvení, v jistém okamžiku se tedy musí rozhodnout ano/ne. Musí tedy vyhodnotit nějaký výraz a jednoznačně určit, zda je roven nějaké hodnotě, případně je větší než nějaká hodnota a podobně. Bez újmy na obecnosti předpokládejme, že musí přesně vyhodnotit znaménko jistého výrazu.

Ačkoliv se na první pohled může zdát, že jde stále o problém přesného vyhodnocování, není tomu tak. Zatímco při přesném vyhodnocování nám jde o nulovou chybu, při vyhodnocení znaménka nám chyba nevadí, pokud nepovede současně ke změně znaménka. Pro tento účel se znamenitě hodí kontrola relativní chyby; připomeňme, že nepřesná hodnota x' je zatížena relativní chybou e vůči přesné hodnotě x, platí-li

$$x' = x(1 + e)\,.$$

Je-li tedy $e > -1$, případně praktičtěji $|e| < 1$, potom mají x a x' stejné znaménko a algoritmus se rozhodne správně.

V průběhu celé kapitoly 3 jsme se zaměřovali na vyhodnocování výrazů s co nejmenší relativní chybou; viděli jsme, že pro základní operace v IEEE 754 je relativní chyba menší než strojové epsilon, tedy číslo mnohem menší než 1. Při vyhodnocování pouhého znaménka tak máme mnohem volnější ruce než při snaze o přesné výpočty. Protože je technika správného vyhodnocení znaménka tak důležitá, věnujeme jí celou kapitolu 5.

S vyhodnocováním znaménka se pojí i následující postřeh. Pokud algoritmus nekonstruuje nová geometrická primitiva, tj. rozhoduje se pouze na základě „přesně“ vyhodnocených vstupů, může si vystačit se znaménkovými testy. Příkladem takového algoritmu je triangularizace množiny bodů; jejím výstupem je totiž pouhá topologická informace o propojení vstupních bodů do trojúhelníků.

Pokud ale algoritmus konstruuje nová geometrická primitiva, nepřesně si je ukládá coby mezivýsledky a na jejich základě se dále rozhoduje, je zřejmé, že ani korektní znaménkové testy nepomohou – jakmile je vstup do testu zatížen chybou, můžeme se dočkat nejrůznějších výsledků. Příkladem algoritmu je konstrukce Voroného diagramu z Delaunayovy triangulace: algoritmus musí *konstruovat* bisektory hran trojúhelníků, počítat *jejich* průsečíky a zvažovat vzájemné polohy průsečíků. Potíž samozřejmě nastává při vyhodnocení vzájemné polohy průsečíků. Klíčem k úspěšnému řešení je proto při vyhodnocení uvažovat celý proces, který k výpočtu průsečíků vedl. Stejnou myšlenku můžeme použít pro libovolný algoritmus.

S technikami obecnými souvisí doplňkové techniky.

Výpočetní strom. Techniku už známe z diskuse znaménkových testů, pro úplnost ji uvedeme znovu: je-li to možné, je vhodné pamatovat si kompletní výpočetní strom, který vedl od vstupů až k finálnímu výsledku. Při detekci problému je pak možné strom znovu vyhodnotit s vyšší přesností, použít jiný typ výpočtu apod. Naopak nevhodné je ukládat si mezivýsledky do proměnných s omezenou přesností a používat je jako vstup dalších výpočtů, aniž by program bral v potaz jejich nepřesnost.

Racionální rotace. Jakékoliv úvahy o přesných výpočtech narážejí na zásadní omezení: čísla, s kterými pracujeme, musí být vyjádřitelná omezeným počtem celých čísel. To samozřejmě splňují čísla celá, čísla racionální i čísla v reprezentaci s plovoucí desetinnou čárkou. Teoreticky vzato to splňují i algebraická čísla, tj. kořeny polynomů s celočíselnými koeficienty; například číslo $\sqrt{2}$ můžeme vyjádřit omezeným počtem celých čísel jako „kořen polynomu x^2-2 ležící v intervalu $[0,2]$“. V praxi je však komplikované pracovat s algebraickými čísly, jež jsou řešeními polynomu stupně čtvrtého a vyššího. Konečně existují ještě čísla transcendentní, která se konečným počtem celých čísel vyjádřit nedají, příkladem jsou π nebo Eulerova konstanta.

Podívejme se, jak s teoretickou klasifikací reálných čísel souvisí praktické geometrické výpočty.

Dejme tomu, že potřebujeme vyjádřit body na přímce svírající s osou x úhel 15°. Jejich souřadnice jsou zřejmě

$$\begin{aligned} x &= t\cos 15^\circ \\ y &= t\sin 15^\circ \end{aligned} \qquad \text{pro } t \in (-\infty,\infty)\,.$$

Při vyčíslení narážíme na problém – všechny běžné algoritmy pro výpočet goniometrických funkcí předpokládají argument zadaný v radiánech. Úhel 15° odpovídá $\pi/12$ radiánům, což je pochopitelně transcendentní číslo. Jelikož jej neumíme vyjádřit přesně, nemůžeme očekávat ani přesné hodnoty souřadnic bodů.

Jelikož mnohé technicky významné úhly (vyjádřené ve stupních omezeným počtem cifer, např. 90° nebo 45°) trpí stejným problémem, zkusme si představit, že vymyslíme algoritmus pro výpočet goniometrických funkcí (omezme se na sinus a kosinus), jehož vstup bude zadán v racionálních násobcích π. Tím jsme sice vyřešili přesné zadávání technicky významných úhlů, ale problém vznikl jinde – siny a kosiny takových úhlů jsou algebraickými čísly. Naneštěstí jen několik málo úhlů vede k „použitelným“ algebraickým číslům, tj. kořenům polynomu do stupně tři, viz [17].

Ať tedy zadáváme úhel jedním či druhým způsobem, neumíme (až na speciální případy) přesně vyčíslit jeho sinus a kosinus. Nejenom, že neumíme

vyjádřit body na výše uvedené přímce; nedovedeme ani spočítat souřadnice bodu $[b_x, b_y]$ vzniklého rotací bodu $[a_x, a_y]$ o úhel ϕ kolem počátku souřadnic, protože platí

$$b_x = a_x \cos\phi - a_y \sin\phi$$
$$b_y = a_x \sin\phi + a_y \cos\phi\,.$$

Důsledky v CAD systémech mohou být zničující: program sice může usoudit, že bod C leží napravo od přímky AB, ale po pootočení celé situace o úhel ϕ může vlivem nepřesností usoudit na opak.

Praktickým řešením je třetí způsob zadávání úhlu – nepřímo. Dá se ukázat, že k libovolnému úhlu ϕ a pro libovolné $\epsilon > 0$ dokážeme najít úhel ϕ', $|\phi - \phi'| < \epsilon$, jehož sinus i kosinus jsou racionálními čísly, viz [16, 10]. Sice se tím zbavujeme možnosti přesného vyjádření většiny technicky významných úhlů, ale za prvé to v mnoha aplikacích nemusí vadit, za druhé si s nimi stejně neumíme jednoduše poradit.

Odkládání výpočtů. Často je možné některé aritmetické operace odložit a vyhnout se tak nepřesnému mezivýsledku. Například práce se zlomky je vlastně pouhým odložením operace dělení; je-li navíc v jistém místě algoritmu nutné dva zlomky porovnat, můžeme to udělat jen s pomocí celočíselných násobení jejich čitatelů a jmenovatelů.

Na podobném principu jsou založeny homogenní souřadnice známé z počítačové grafiky, které bod $[x, y]$ afinního (eukleidovského) prostoru vyjadřují trojicí $[wx, wy, w]$, kde $w \neq 0$ je vhodně zvolené reálné číslo. Jako příklad jejich použití si můžeme uvést výpočet průsečíku přímek AB a CD daných body $A = [a_x, a_y]$, $B = [b_x, b_y]$, $C = [c_x, c_y]$, $D = [d_x, d_y]$. V afinním prostoru bychom museli vypočítat koeficienty přímek

$$AB: \qquad (a_y - b_y)x + (b_x - a_x)y + \begin{vmatrix} a_x & a_y \\ b_x & b_y \end{vmatrix} = 0$$
$$CD: \qquad (c_y - d_y)x + (d_x - c_x)y + \begin{vmatrix} c_x & c_y \\ d_x & d_y \end{vmatrix} = 0$$

a následně tuto soustavu dvou dvou rovnic vyřešit. Při jejím řešení bychom se samozřejmě nevyhnuli dělení.

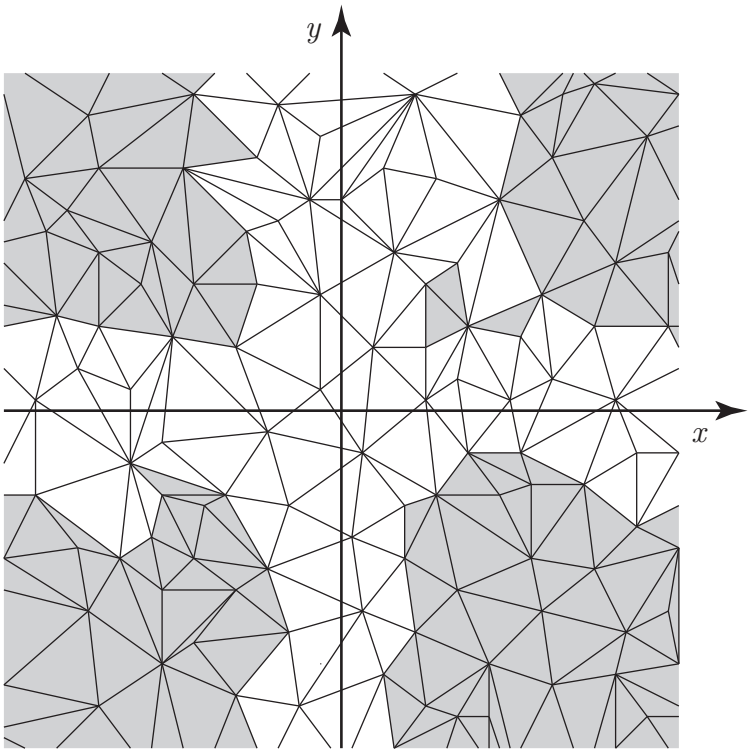
V homogenních souřadnicích rovnou spočítáme

$$AB \cap CD: \qquad \Big([a_x, a_y, 1] \times [b_x, b_y, 1]\Big) \times \Big([c_x, c_y, 1] \times [d_x, d_y, 1]\Big)\,,$$

kde $\times$ je vektorový součin.

Označíme-li si složky výsledku jako $[p_x, p_y, p_w]$, získáme souřadnice průsečíku v afinním prostoru snadno jako $[p_x/p_w, p_y/p_w]$. Výsledek výpočtu v homogenních souřadnicích je samozřejmě ekvivalentní výpočtu v afinním prostoru, jen operaci dělení jsme si odložili až na konec.

Transformace úlohy. Zejména geometrické výpočty jsou obvykle nezávislé na zvoleném souřadnicovém systému; například trojúhelník ABC je orientovaný po směru nebo proti směru hodinových ručiček nezávisle na volbě souřadnicového počátku či natočení os. Vhodně zvoleným souřadným systémem můžeme výpočet jednak zjednodušit, jednak zpřesnit. Ukažme si to na příkladu.



Obrázek 11: *U šedě vybarvených trojúhelníků proběhne translace do počátku pomocí odčítání v IEEE 754 přesně. Obrázek je převzat z [6].*

Úlohu orientace trojúhelníku ABC, $A = [a_x, a_y]$, $B = [b_x, b_y]$, $C = [c_x, c_y]$ vyřešíme vyhodnocením znaménka funkce Orient2D(A, B, C) (název je převzat z [5, 6]):

$$\text{Orient2D}(A, B, C) = \begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix}. \tag{10}$$

Posuneme-li souřadný systém počátkem do bodu C, znaménko se nezmění,

ale determinant se zjednoduší:

$$\begin{aligned}\text{Orient2D}(A,B,C) &= \text{Orient2D}(A-C,B-C,\mathbf{0})\\ &= \begin{vmatrix} a_x-c_x & a_y-c_y & 0\\ b_x-c_x & b_y-c_y & 0\\ 0 & 0 & 1\end{vmatrix}\\ &= \begin{vmatrix} a_x-c_x & a_y-c_y\\ b_x-c_x & b_y-c_y\end{vmatrix}. \qquad (11)\end{aligned}$$

Na první pohled se jeví verze (11) jako horší, protože prvky determinantu jsou zaokrouhlenými výsledky odčítání. Pokud si ale uvědomíme, že rozdíl dvou blízkých čísel se v IEEE 754 vyhodnocuje přesně (viz Sterbenzovo lemma (7) na straně 36), zjistíme, že pro většinu typických trojúhelníků se argumenty prvky determinantu vyhodnotí přesně, viz obr. 11. Jelikož výpočet determinantu 2×2 vyžaduje méně operací než výpočet determinantu 3×3, bude výpočet (11) přesnější (viz [6]).

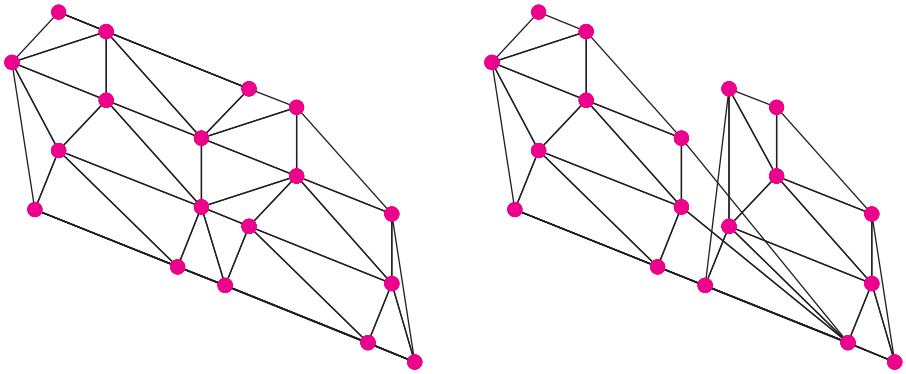
Jistou pozornost je třeba věnovat výběru bodu, který bude přesunut do počátku (v našem případě to byl bod C); jde o tzv. *problém výběru pivotu*. Protože chceme maximálně využít Sterbenzova lemmatu, bude zřejmě nejvhodnější seřadit body A, B, C podle souřadnice x (nebo y) a coby pivot zvolit prostřední bod. Analýzu je možné najít v [10]; obecně je třeba volit pivot tak, aby byl výsledný výpočet co nejstabilnější.

5 Znaménkové testy

Existuje mnoho typů úloh, které nekonstruují nové geometrické útvary, ale pouze vyhodnocují vzájemné polohy útvarů vstupních. Příklady takových úloh jsou „bod v polygonu“, triangularizace množiny bodů, detekce průsečíků geometrických útvarů, plánování cest a jiné. Jako všechny netriviální úlohy obsahují podmíněné příkazy, čili v jistém okamžiku se musí program na základě vstupních hodnot rozhodnout ano/ne. Bez újmy na obecnosti můžeme předpokládat, že další běh programu závisí na vyhodnocení znaménka jistého výrazu.

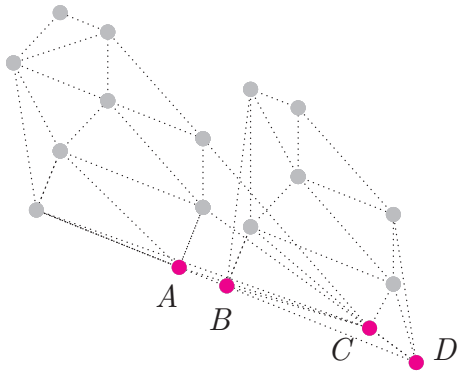
Klasický příklad ukazující vliv jediného špatného rozhodnutí na výsledek triangularizace množiny bodů ukazuje [5], viz obr. 12 a 13. Je-li algoritmus tvořen jedním dlouhým řetězcem rozhodnutí, je zřejmé, že chyba na počátku řetězce může vést k nesmyslnému výsledku; tehdy musíme každý test provést obzvlášť pečlivě. O něco méně pozorní můžeme být jen u algoritmů vyhodnocujících množství oddělených podúloh, takže chyba vyhodnocení v jedné nemůže mít žádný vliv na vyhodnocení ostatních.

V této kapitole se ukážeme tři přístupy, jak se vypořádat s korektním vyhodnocením znaménka. První bude založeno především na logice výpočtu a ukážeme



a) korektní triangulace *b) reálný výstup programu*

Obrázek 12: *Delaunayova triangulace množiny bodů, převzato z [5].*



Obrázek 13: *Zvýraznění oblasti, jejíž vinou došlo k chybě znázorněné obrázkem 12b. Program nesprávně vypočítal, že bod A leží uvnitř kružnice opsané trojúhelníku BCD (poloha bodu C je pro názornost mírně pozměněna). Jediné rozhodnutí vedlo k topologicky nesmyslné triangulaci.*

si jej na vyhodnocení přesného znaménka sumy. Druhý bude založen na analýze chyby, ukážeme si jej na vyhodnocení determinantu matice 2×2. Třetí bude založen na adaptivní přesné aritmetice a ukážeme si jej na vyhodnocení testu orientace trojúhelníku.

5.1 Přesné znaménko sumy

Mnoho výpočtů, například výpočet determinantu, vede k součtu několika čísel. Ukazovali jsme si různé přístupy k výpočtu sumy; ale i nejlepší z nich, Kahanův algoritmus na straně 3.3, nezaručoval přesný výsledek. Zejména při sčítání kladných a záporných čísel může dojít ke katastrofálnímu odečtení, které zvětší relativní chybu výpočtu nad 1 a ani znaménko výsledku nemusí odpovídat skutečnosti.

Pokud nám jde pouze o znaménko sumy, nemusíme hned nasazovat přesnou aritmetiku. Příkladem algoritmu, který znaménko vyhodnotí přesně za použití základních IEEE 754 operací, je ESSA (Exact Sign of Sum Algorithm). Jeho původní verze je hezky popsána v [10], my si obdobným způsobem vysvětlíme efektivnější modifikovanou verzi popsanou v [18].

- Vstup: sestupně seřazené seznamy kladných čísel $\{a_i\}_{i=1}^{k}$, $\{b_j\}_{j=1}^{l}$ ($a_i \geq a_{i+1}$, $b_j \geq b_{j+1}$).
- Výstup: znaménko $S = \operatorname{sign}\left(\sum_{i=1}^{k} a_i - \sum_{j=1}^{l} b_j\right)$
- Algoritmus:

  1. Rozhodni triviální případy:
     (a) Jsou-li oba seznamy prázdné ($k = l = 0$), pak $S = 0$.
     (b) Je-li $k > l = 0$, pak $S = +1$.
     (c) Je-li $0 = k < l$, pak $S = -1$.
     (d) Je-li $a_1 > l \otimes b_1$, pak $S = +1$. Hodnota $l \otimes b_1$ totiž omezuje sumu všech b_i. Pokud je samotné a_1 větší než než toto omezení, je znaménko bez dalších výpočtů jasné.
     (e) Je-li $b_1 > k \otimes a_1$, pak $S = -1$. Zdůvodnění je analogické.
  2. Vyjmeme největší členy seznamů $\{a_i\}$ a $\{b_j\}$ a později do nich vrátíme jejich rozdíl tak, aby celková suma zůstala zachována.
     $\alpha \leftarrow a_1$
     $\beta \leftarrow b_1$
     Vyjmi a_1 a b_1 ze seznamů $\{a_i\}$, $\{b_j\}$.
  3. Budeme postupovat podle Dekkerova lemmatu (8) na straně 38, tj. budeme počítat přesný rozdíl $\alpha - \beta$, výsledek uložíme do proměnných x, y. Napřed vypočtěme x:
     $x \leftarrow \alpha \ominus \beta$
  4. Dekkerovo lemma říká, jak vypočítat y takové, aby $x + y = \alpha - \beta$ pro $\alpha \geq \beta$. Pokud je tedy x záporné, musíme výpočet y mírně pozměnit prohozením α a β.
     (a) Je-li $x > 0$, vypočítej $y \leftarrow (\alpha \ominus x) \ominus \beta$.
     Zařaď x do seznamu $\{a_i\}$.
     Je-li $y > 0$, zařaď y do seznamu $\{a_i\}$.
     Je-li $y < 0$, zařaď y do seznamu $\{b_i\}$.
     (b) Je-li $x < 0$, vypočítej $y \leftarrow \alpha \ominus (\beta \oplus x)$.
     Zařaď x do seznamu $\{b_i\}$.
     Je-li $y > 0$, zařaď y do seznamu $\{a_i\}$.
     Je-li $y < 0$, zařaď y do seznamu $\{b_i\}$.

(c) Je-li $x = 0$, není třeba dělat nic, protože $\alpha = \beta$, a proto $y = 0$.

5. Uprav hodnoty k, l, aby odpovídaly skutečné délce seznamů $\{a_i\}$, $\{b_j\}$.

Text [18] se podrobně zabývá důkazem, že algoritmus je přesný (to by čtenář ostatně měl být schopen posoudit sám), že po konečném počtu kroků skončí, efektivním zapojením max-haldy pro implementaci „seřazeného seznamu“ apod. My se spokojíme s konstatováním, že algoritmus je po všech stránkách v pořádku.

5.2 Standardní vs. přesná aritmetika – znaménko determinantu matice 2×2

V mnoha aplikacích se setkáme s výpočtem $ab - cd$. Může jít o výpočet determinantu matice s prvky a, b, c, d; může jít o výpočet součinu komplexních čísel $(a + bi) \times (c + di)$; může jít o výpočet skalárního součinu vektorů $(a, b) \cdot (c, -d)$; výpočet diskriminantu $b^2 - 4ac$ ve výpočtu kořenů kvadratické rovnice $ax^2 + bx + c = 0$ má ostatně stejný tvar; a tak dále. V některých aplikacích je důležité přesně rozhodnout o jeho znaménku. My si ukážeme dva přístupy; jeden standardní, založený na rutinní analýze chyby, a druhý vylepšený o důkladnou analýzu chyby. Druhý postup však bude uveden pouze pro zajímavost; můžeme si jej dovolit, protože výraz $ab - cd$ je jednoduchý. U složitějších výrazů by pravděpodobně byla analýza neprakticky složitá.

Oba dva přístupy ale postupují podle stejné logiky. Za prvé vyhodnotí výraz pomocí IEEE 754 operací, tj. vyhodnotí $a \otimes b \ominus c \otimes d$, za druhé vyhodnotí, zda mohlo dojít k velké chybě, a pak případně vypočtou výraz přesnou aritmetikou.

Rutinní analýza chyby

Je-li $t = t_1 + t_2$ přesná (*true*) hodnota součtu a $x = t_1 \oplus t_2$ její zaokrouhlená (*approximate*) hodnota, víme, že $x = t + et$, kde $e \leq \epsilon$, viz (5) na straně 31. Podobně je tomu pro ostatní IEEE 754 operace.

Podobně jako v [5] (kde také může čtenář najít další příklady analýzy chyby), zaveďme si pro vztah mezi x a t užitečnou zkrácenou notaci a pišme $x = t \pm \epsilon|t|$. Praktičtější pro nás bude ekvivalentní

$$t = x \pm \epsilon|t| \,. \tag{12}$$

Díky vlastnostem IEEE 754 operací také víme, že

$$t = x \pm \epsilon|x| \,, \tag{13}$$

viz (6) na straně 31. Vztahy (12) i (13) se v analýze chyby často využívají.

Označme si přesné i zaokrouhlené části výpočtu $t_A = ab - cd$ a $A = (a \otimes b) \ominus (c \otimes d)$. Předpokládejme přitom, že čísla a, b, c, d jsou přesně reprezentovatelná

v IEEE 754 proměnných, tj. nejsou zatížena chybou.

$$\begin{aligned} t_1 &= ab & x_1 &= a \otimes b \\ t_2 &= cd & x_2 &= c \otimes d \\ t_A &= t_1 - t_2 & A &= x_1 \ominus x_2 \end{aligned}$$

Díky zaručené přesnosti IEEE 754 operací můžeme psát

$$\begin{aligned} t_A = t_1 - t_2 &= x_1 \pm \epsilon|x_1| - x_2 \pm \epsilon|x_2| \\ &= x_1 - x_2 \pm \epsilon(|x_1| + |x_2|) \\ &= A \ \pm \epsilon|A| \pm \epsilon(|x_1| + |x_2|)\,. \end{aligned}$$

Kdy budou znaménka t_A a A stejná? Je-li $A \geq 0$, potom $|A| = A$ a

$$t_A = A\ (1 \pm \epsilon) \pm \epsilon(|x_1| + |x_2|)$$

čili t_A a A budou mít stejné znaménko, pokud $|A|(1-\epsilon) > \epsilon(|x_1| + |x_2|)$. Musíme totiž brát v potaz nejhorší možnou kombinaci znamének u ϵ. Naopak je-li $A < 0$, potom $|A| = -A$ a

$$t_A = -A(-1 \pm \epsilon) \pm \epsilon(|x_1| + |x_2|)$$

čili znaménka t_A a A budou stejná, když $-A(-1+\epsilon) < -\epsilon(|x_1| + |x_2|)$. To je totéž jako $|A|(1-\epsilon) > \epsilon(|x_1| + |x_2|)$. Pro oba případy tedy pro shodu znamének získáváme stejné kritérium

$$|A|(1-\epsilon) > \epsilon(|x_1| + |x_2|)\,.$$

To bohužel neumíme vyhodnotit přesně, protože k němu potřebujeme přesné operace – například hodnotou $1-\epsilon$ nelze reprezentovat číslem v IEEE 754. Test ale můžeme přepsat do podoby

$$|A| > 2\epsilon(|x_1| + |x_2|)$$

protože potom platí

$$\begin{aligned} |A|(1-\epsilon) &> (1-\epsilon)2\epsilon(|x_1| + |x_2|) \\ &= (2\epsilon - 2\epsilon^2)(|x_1| + |x_2|) \\ &> \epsilon(|x_1| + |x_2|) \end{aligned}$$

přičemž poslední nerovnost platí tehdy, je-li $2\epsilon - 2\epsilon^2 > \epsilon$, čili pro $\epsilon < 1/2$. To je pro libovolný IEEE 754 formát bezpečně splněno.

Hodnotu $2\epsilon \times (|x_1| + |x_2|)$ bohužel také neumíme spočítat přesně, protože součin a součet umíme obecně spočítat pouze s relativní chybou ϵ. Vyřešme nejprve

součet: jelikož platí $|x_1| + |x_2| = (1 \pm \epsilon)(|x_1| \oplus |x_2|)$, můžeme kritérium upravit na

$$|A| > 2\epsilon(1+\epsilon) \times (|x_1| \oplus |x_2|)\,.$$

Podobně operaci $\times$ nahradíme operací $\otimes$ a opět započteme relativní chybu ϵ:

$$|A| > 2\epsilon(1+\epsilon)^2 \otimes (|x_1| \oplus |x_2|)\,.$$

Přesnou hodnotu členu $2\epsilon(1+\epsilon)^2 = 2\epsilon + 4\epsilon^2 + 2\epsilon^3$ bohužel neumíme vyjádřit číslem s plovoucí desetinnou čárkou. Jelikož platí $\epsilon = 2^{-p}$, kde p je počet bitů mantisy, můžeme psát

$$\begin{aligned} 2\epsilon + 4\epsilon^2 + 2\epsilon^3 &= 2^{-p+1} + 2^{-2p+2} + 2^{-3p+1} = 2^{-p+1}(1 + 2^{-p+1} + 2^{-2p}) \\ &< 2^{-p+1}(1 + 2^{-p+2}) = 2\epsilon(1+4\epsilon)\,. \end{aligned}$$

Výsledné číslo je přímo zapsáno v normalizovaném tvaru (s exponentem $-p+1$ a p-bitovou mantisou $1 + 2^{-p+2}$), a proto jej můžeme uložit do běžného čísla podle IEEE 754.

Pro přehlednost shrneme, co jsme zjistili. Chceme-li zjistit znaménko výrazu $ab - cd$, vypočteme jeho přibližnou hodnotu $A = a \otimes b \ominus c \otimes d$. Znaménko A je v pořádku, pokud platí

$$|A| > 2\epsilon(1+4\epsilon) \otimes (|a \otimes b| \oplus |c \otimes d|)\,.$$

Přesný výpočet znaménka

Už víme, za jakých podmínek má hodnota $A = a \otimes b \ominus c \otimes d$ určitě stejné znaménko jako přesná hodnota $ab - cd$. Pokud test odvozený v předchozím oddílu neplatí, musíme o znaménku rozhodnout jinak.

Hlavním problémem při výpočtu $ab - cd$ je neznalost přesné hodnoty součinů ab a cd. Případné malé zaokrouhlovací chyby se totiž můžou operací odečtení mnohonásobně zesílit. Klíčem k přesnému vyhodnocení znaménka je proto přesné vyjádření součinu.

Pro přesné vyjádření součtu $a + b$ známe Dekkerovo lemma (8) ze strany 38. O součinu zatím jen ze strany 37 víme, že může za jistých okolností proběhnout přesně. Konkrétně: $ab = a \otimes b$, je-li posledních z_a bitů mantisy a nulových, posledních z_b bitů mantisy b nulových a současně $z_a + z_b \geq p$, kde p je počet bitů mantisy.

Kdybychom ale uměli vyjádřit čísla a, b jako

$$a = x_a + y_a \qquad\qquad b = x_b + y_b$$

kde čísla x_a, x_b, y_a, y_b mají posledních $p/2$ bitů mantisy nulových, mohli bychom přesnou hodnotu ab zapsat jako součet několika čísel s plovoucí desetinnou čárkou:

$$ab = (x_a + y_a)(x_b + y_b) = x_a x_b + x_a y_b + x_b y_a + y_a y_b\,, \tag{14}$$

To je svým způsobem podobné Dekkerovu lemmatu (8), které součet $a + b$ vyjadřuje součtem $x + y$. Dekker se problému přesného násobení věnoval rovněž a odvodil následující postup; často je označovaný jako Dekker-Veltkampův. Jeho podrobné zdůvodnění je hezky popsáno v [5]; čtenář samozřejmě může studovat i původní článek [19] z roku 1971, ale podobně jako ostatní články starší než norma IEEE 754 je i tento hůře čitelný než novější důkazy.

Rozdělme nejprve číslo a na čísla x_a a y_a. První bude mít $(p - s)$-bitovou mantisu, druhé $(s - 1)$-bitovou mantisu a bude platit $a = x_a + y_a$ a $|x_a| \geq |y_a|$. Konkrétně zvolíme $s = \lceil p/2 \rceil$. Čtenáře nemusí znepokojovat, že původní číslo s p-bitovou mantisou bude rozděleno do dvou čísel s celkem $p - 1$ bity v mantisách – „chybějící bit“ je ukryt ve znaménku čísla y_a.

```
1  tmp = (2^s + 1)*a;
2  a' = tmp - a;
3  xa = c - a';
4  ya = a - xa;
```

Program je založen na stejných myšlenkách jako úvahy ze strany 37, které vedly k Dekkerovu lemmatu. Stačí si uvědomit, že $(2^s + 1)a = 2^s a + a$, čili jde o součet dvou čísel s rozdílnými exponenty, která jsme na straně 37 označovali jako a a b.

Obdobně rozdělíme i číslo b na x_b a y_b, opět volíme $s = \lceil p/2 \rceil$.

Součin ab nyní vyjádříme součtem $x + y$, kde opět x bude vyjadřovat zaokrouhlenou hodnotu součinu a y korekční člen.

```
1  x  = a * b;
2  e1 = -x + (xa * xb);
3  e2 = e1 + (ya * xb);
4  e3 = e2 + (xa * yb);
5  y  = e3 + (ya * yb);
```

Z kódu je vidět, že jde v podstatě o akumulaci jednotlivých členů z (14). Důkazem přesnosti algoritmu se ale zabývat nebudeme.

Vraťme se proto k našemu problému, tj. výpočtu znaménka $ab - cd$. Jelikož nyní umíme vyjádřit $ab = x + y$ a $cd = x' + y'$, stačí vyhodnotit pouze znaménko sumy $x + x' + y + y'$. K tomu můžeme samozřejmě použít algoritmus ESSA ze strany 58. Tím jsme dokončili ukázku standardního přístupu k vyhodnocení znaménka výrazu.

Důkladná analýza chyby

Výhodou rutinní analýzy chyby, kterou jsme si ukázali, je její univerzálnost. Ačkoliv se může postup jevit na první pohled jako „spadlý z nebe“, dá se bez větších potíží aplikovat na mnohem složitější výpočty. Na druhou stranu může být odhad chyby příliš pesimistický, tj. jsme nuceni přejít k přesnému výpočtu častěji, než je skutečně nutné.

Pro náš jednoduchý problém si ovšem můžeme dovolit chybu analyzovat důkladněji. Je-li $t_A = ab - cd$ a $A = (a \otimes b) \ominus (c \otimes d)$, může při srovnání jejich znamének nastat několik situací; vyjádřeme si je tabulkou 5. Ještě než se na ni podíváme, analyzujme případy, kdy některá z čísel a, b, c, d jsou nulová. To proto, že při analýze relativní chyby komplikují nuly důkazy.

Pokud je ve výrazu $a \otimes b \ominus c \otimes d$ některé z čísel a, b, c, d rovno nule, je příslušný součin také roven nule a výraz degeneruje na součet dvou nul nebo na součet nuly a nenulového součinu. Součet s nulou proběhne vždy přesně a případný jediný nenulový součin je přesně zaokrouhlený, tj. jeho znaménko je platné. Můžeme tedy konstatovat, že v takovém případě platí $\operatorname{sign} A = \operatorname{sign} t_A$ a nadále se zabývat pouze situací, kdy žádné z čísel a, b, c, d není rovno nule.

		$\operatorname{sign} t_A$		
		-1	0	1
	-1	OK	1	2
$\operatorname{sign} A$	0	3	OK	3
	$+1$	2	1	OK

Tabulka 5: *Různé situace při srovnání znamének A a t_A*

Je zřejmé, že při $\operatorname{sign} A = \operatorname{sign} t_A$ (diagonála v tabulce 5) je situace v pořádku. Pak mohou hypoteticky nastat tři situace, které v pořádku nejsou:

1. $t_A = 0$, ale $A \neq 0$ (v tabulce 5 číslo 1),
2. znaménka t_A a A jsou opačná (v tabulce 5 číslo 2),
3. $A = 0$, ale $t_A \neq 0$ (v tabulce 5 číslo 3)

Podívejme se postupně na jednotlivé případy.

1. $\mathbf{t_A = 0}$, ale $\mathbf{A \neq 0}$: Je-li $ab - cd = 0$, potom $ab = cd$. Protože jsou výsledky IEEE 754 operací přesně zaokrouhlené, plyne z toho i $a \otimes b = c \otimes d$. Rozdíl dvou stejných čísel je ale roven nule, čili $\operatorname{sign}(a \otimes b \ominus c \otimes d) = 0$, což je spor. K situaci 1 tedy nemůže dojít.

2. Znaménka $\mathbf{t_A}$ a A jsou opačná: Jde o nejhorší možný případ, výpočet došel k opačnému závěru než měl. Bez újmy na obecnosti předpokládejme, že $\operatorname{sign} t_A = 1$ a $\operatorname{sign} A = -1$. Důkaz opačné situace je analogický.

Označme si pro jednoduchost $x = ab$, $x' = a \otimes b = \operatorname{round} x$, $y = cd$, $y' = c \otimes d = \operatorname{round} y$. Má-li být $\operatorname{sign} t_A = \operatorname{sign}(x - y) = 1$, potom musí platit $x > y$. Jelikož je funkce round neklesající, platí implikace

$$x > y \quad \Rightarrow \quad \operatorname{round} x \geq \operatorname{round} y\,,$$

jinými slovy platí $x' \geq y'$. Potom ale musí platit $x' \ominus y' \geq 0$, což je spor. Ani k situaci 2 tedy nemůže dojít.

3. **$A = 0$, ale $t_A \neq 0$:** Kdyby nemohlo dojít ani k situaci 3, byli bychom spokojeni – znamenalo by to, že výpočet znaménka A je vždy korektní. To bohužel není pravda. Nejlépe se o tom přesvědčíme konkrétním příkladem.

   Je-li $a = b = 1$ potom $ab = 1$ i $a \otimes b = 1$. Druhá rovnost snadno plyne ze znalosti algoritmu násobení a skutečnosti, že mantisa čísla 1 má většinu bitů nulových.

   Čísla $c = 1+2\epsilon$ a $d = 1-2\epsilon$, kde $\epsilon = 2^{-p}$, jsou reprezentovatelná s p-bitovou mantisou. Platí pro ně

$$cd = (1 + 2\epsilon)(1 - 2\epsilon) = 1 - 4\epsilon^2 = 1 - 2^{-2p+2}\,.$$

   Číslo cd by ovšem vyžadovalo $2p - 1$ bitů mantisy; po zaokrouhlení na p bitů máme $c \otimes d = 1$, a proto

$$ab - cd > 0 \quad \wedge \quad a \otimes b \ominus c \otimes d = 0\,.$$

   K situaci 3 tedy dojít může.

Analyzovali jsme tři možné typy chyby výpočtu a došli k závěru, že chyba může být pouze typu 3. Proto přepočítání $ab - cd$ přesnou aritmetikou spustíme pouze v případě, kdy $a \otimes b \ominus c \otimes d = 0$.

5.3 Adaptivní aritmetika – orientace trojúhelníku

Při výpočtu determinantu matice 2×2, jejíž prvky jsme znali přesně, jsme postupovali přímočaře:

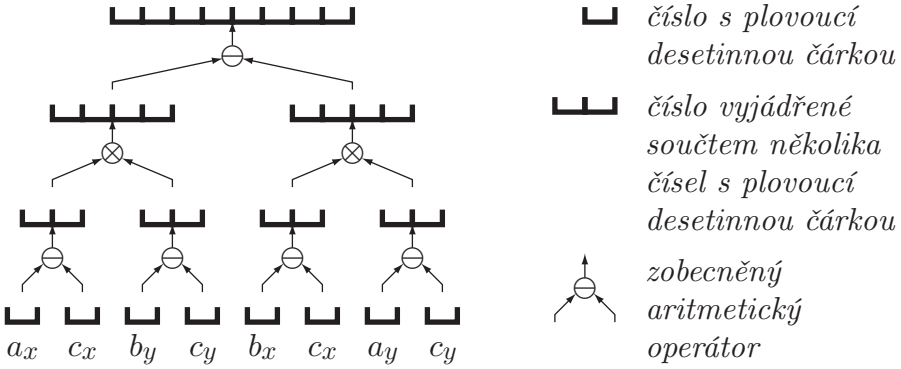
1. vypočti determinant IEEE 754 aritmetikou,
2. rozhodni, zda mohlo dojít k závažné zaokrouhlovací chybě,
3. v případě nutnosti determinant přepočítej přesnou aritmetikou.

Mohli jsme si to dovolit, protože přesná aritmetika vyžadovala operandy vyjádřené nejvýše dvěma čísly s plovoucí desetinnou čárkou. Co ale dělat v případě složitějších výrazů? Například rozhodnutí, zda bod $C = [c_x, c_y]$ leží vpravo či vlevo od přímky dané body $A = [a_x, a_y]$ a $B = [b_x, b_y]$ (neboli rozhodnutí o orientaci trojúhelníku ABC) vede na výpočet znaménka determinantu

$$\operatorname{sign} \begin{vmatrix} a_x - c_x & a_y - c_y \\ b_x - c_x & b_y - c_y \end{vmatrix}. \tag{15}$$

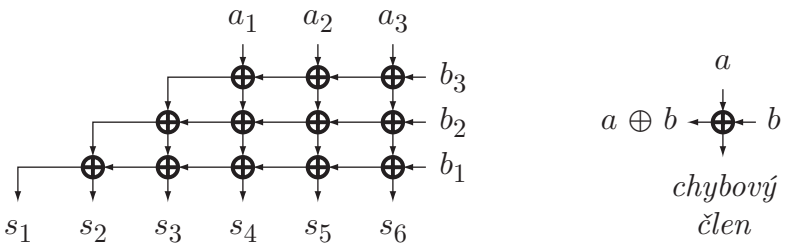
Jsou-li souřadnice bodů A, B, C známé přesně, vyžadují už prvky matice vyjádření dvěma čísly s plovoucí desetinnou čárkou (viz Dekkerovo lemma (8) na straně 38), dílčí součiny ve výpočtu determinantu pak vyžadují reprezentaci čtyřmi čísly s plovoucí desetinnou čárkou (viz Dekker-Veltkampův součin na straně 62) a tak dále.

Dospěli jsme tedy k závěru, že pro přesné výpočty je užitečné čísla reprezentovat součtem několika čísel s plovoucí desetinnou čárkou (tzv. komponent) a že bude zapotřebí implementovat nad nimi zobecněné aritmetické a relační operátory. Výpočet determinantu v (15) pak můžeme graficky znázornit obrázkem 14.



Obrázek 14: *Výpočetní strom pro přesný výpočet* $(a_x - c_x)(b_y - c_y) - (b_x - c_x)(a_y - c_y)$*, kde přesnost mezivýsledků zajišťuje jejich reprezentace součtem několika čísel s plovoucí desetinnou čárkou.*

Zobecněné aritmetické a logické operátory musí předpokládat vícekomponentové operandy, výsledek poskytovat pochopitelně rovněž vícekomponentový. Jejich základním stavebním kamenem bude typicky Dekkerův součet (dva vstupy, dva výstupy) a Dekker-Veltkampův součin (opět dva vstupy, dva výstupy). Naivní přístup k součtu vícekomponentových operandů ukazuje obrázek 15; lepší postup a další operace ukazuje [5].



součet dvou tříkomponentových čísel

Dekkerův součet dvou čísel

Obrázek 15: *Naivní přístup k součtu* $s = s_1 + s_2 + \cdots + s_6$ *dvou tříkomponentových čísel* $a = a_1 + a_2 + a_3$ *a* $b = b_1 + b_2 + b_3$.

Pokračujme v úvahách dále. Víme, že výpočet determinantu (15) vede na číslo, které je třeba reprezentovat až osmi komponentami, a že máme rozhodnout o jeho znaménku. Také víme, že o znaménku můžeme často rozhodnout jen na

základě přibližného výpočtu pomocí IEEE 754 aritmetiky, víme-li, že zaokrouhlovací chyby nepřekročily jistý práh. Je tedy nasnadě otázka: pokud víme, že zaokrouhlovací chyby byly příliš závažné, je skutečně zapotřebí všech osm komponent výsledku, abychom o znaménku rozhodli? Zajisté ne.

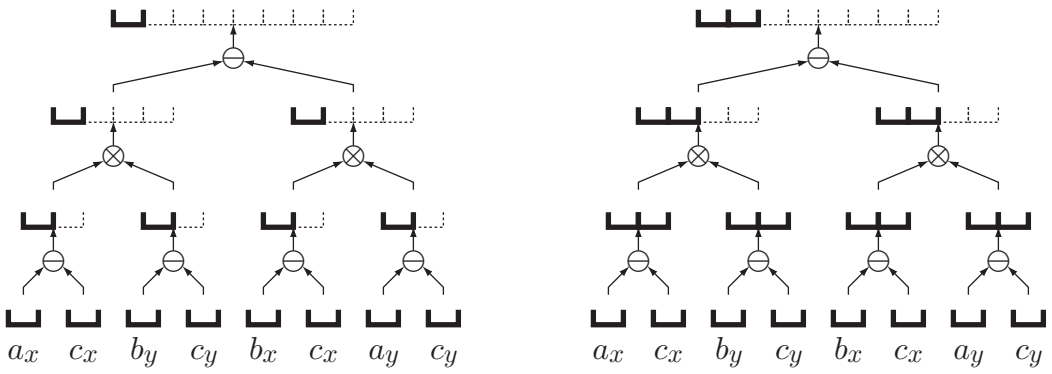
Pokud bude mít vyjádření čísla součtem několika komponent vhodné vlastnosti, můžeme k přesnému výsledku dojít v několika krocích, kde každý krok zpřesní předchozí odhad. Jedna z „vhodných vlastností“ je maximalizace přesnosti dílčích součtů. Co se tím myslí?

Je-li číslo a vyjádřeno součtem n komponent (každá, je vyjádřená jedním číslem s plovoucí desetinnou čárkou), tj.

$$a = a_1 + a_2 + \cdots + a_n \,,$$

potom by bylo příjemné, kdyby a_1 byla nejlepší aproximace čísla a jediným číslem s plovoucí desetinnou čárkou. Podobně by bylo užitečné, kdyby $a_1 + a_2$ byla nejlepší aproximace čísla a vyjádřená dvěma čísly s plovoucí desetinnou čárkou. A tak dále.

Pokud by rozklad čísla na komponenty splňoval uvedenou vlastnost, mohli bychom přibližný výpočet celého výrazu standardními IEEE 754 operacemi graficky vyjádřit obrázkem 16 vlevo. Tučně vyznačené jsou komponenty, které se skutečně počítají (tj. jen nejdůležitější komponenty každého mezivýsledku), čárkovaně pak komponenty, které se nepočítají.



Obrázek 16: *Výpočty výrazu* $(a_x - c_x)(b_y - c_y) - (b_x - c_x)(a_y - c_y)$ *s různou přesností. Čárkovaně jsou naznačeny komponenty, které se ve skutečnosti nepočítají.*

Rutinní analýzou zaokrouhlovací chyby bychom došli k tvrzení (viz [5]), že výrazy

$$\begin{aligned} t_A &= (a_x - c_x) \times (b_y - c_y)(b_x - c_x) \times (a_y - c_y) \\ A &= (a_x \ominus c_x) \otimes (b_y \ominus c_y) \ominus (b_x \ominus c_x) \otimes (a_y \ominus c_y) \end{aligned}$$

mají stejné znaménko, platí-li

$$|A| \geq (3\epsilon + 16\epsilon^2) \otimes \Big(\Big| (a_x \ominus c_x) \otimes (b_y \ominus c_y) \Big| \oplus \Big| (b_x \ominus c_x) \otimes (a_y \ominus c_y) \Big| \Big) .$$

Jakmile by podmínka splněna nebyla, zkusili bychom vypočítat více komponent, abychom aproximaci výsledku zpřesnili. Zřejmě bychom museli zejména zpřesnit výpočty členů matice, tj. podvýrazy $(a_x - c_x)$ atd., protože sebemenší chyba na začátku výpočtu může výsledek znehodnotit. Dekkerovým lemmatem bychom tedy vypočetli dvoukomponentová čísla $x_i + y_i$:

$$\begin{aligned} a_x - c_x &= x_1 + y_1 \\ b_y - c_y &= x_2 + y_2 \\ b_x - c_x &= x_3 + y_3 \\ a_y - c_y &= x_4 + y_4 \, . \end{aligned}$$

a počítali jejich součiny. Pro jednoduchost si rozepišme pouze součin $(x_1 + y_1) \times (x_2 + y_2)$:

$$(x_1 + y_1) \times (x_2 + y_2) = x_1 x_2 + x_1 y_2 + x_2 y_1 + y_1 y_2 \, ,$$

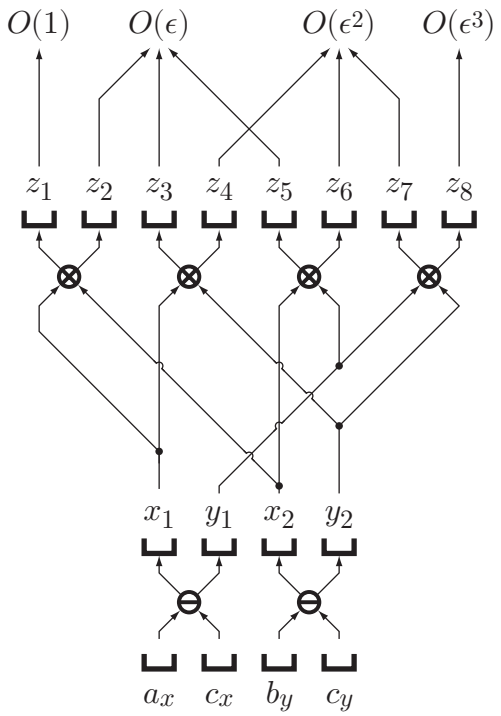
kde dílčí součiny vyjádříme Dekker-Veltkampovým součinem jako

$$\begin{aligned} x_1 x_2 &= z_1 + z_2 \\ x_1 y_2 &= z_3 + z_4 \\ x_2 y_1 &= z_5 + z_6 \\ y_1 y_2 &= z_7 + z_8 \, . \end{aligned}$$

Z vlastností Dekkerova součtu plyne, že čísla y_i jsou v absolutní hodnotě menší než $\epsilon|x_i|$; zkráceně můžeme zapisovat, že čísla x_i jsou velikosti $O(1)$ a čísla y_i velikosti $O(\epsilon)$. Stejnou vlastnost má i Dekker-Veltkampův součin. Proto můžeme čísla z_1 až z_8 roztřídit:

$$\begin{aligned} z_1 \quad &\in \quad O(1) \\ z_2, \ z_3, \ z_5 \quad &\in \quad O(\epsilon) \\ z_4, \ z_6, z_7 \quad &\in \quad O(\epsilon^2) \\ z_8 \quad &\in \quad O(\epsilon^3) \, . \end{aligned}$$

Dá se tedy očekávat, že přesnější vyjádření determinantu (15) získáme započtením z_2, z_3, z_5 z jedné větve součinu a příslušných komponent z větve druhé; schematicky je započtení $O(\epsilon)$ členů naznačeno obrázkem 16 vpravo, podrobně je pak jedna z výpočetních větví rozkreslena na obrázku 17. Opět by bylo třeba analyzovat maximální možnou zaokrouhlovací chybu a opět rozhodnout, zda nám toto zpřesnění stačí, nebo zda bude k rozhodnutí o znaménku determinantu vzít v úvahu i menší členy.



Obrázek 17: *Detail výpočtu* $(a_x - c_x)(b_y - c_y)$. *Výpočetní strom ale není dotažen do konce; neřeší, jak z osmi čísel* z_1 *až* z_8 *získat čtyři výsledné komponenty součinu.*

Touto podkapitolou jsme se pouze seznámili se základními myšlenkami adaptivní aritmetiky, aniž bychom se zabývali důležitými detaily; například jsme vůbec neřešili, jak z osmi čísel z_1 až z_8 získat čtyři komponenty velikosti $O(1)$, $O(\epsilon)$, $O(\epsilon^2)$ a $O(\epsilon^3)$. Neřešili jsme, zda je při adaptivním zpřesňování efektivnější postupovat pozvolným zvyšováním přesnosti, nebo se k přesnému výsledku přibližovat menším počtem kroků. Tím a mnoha dalšími detaily a postupy se zabývá například [5]. Tam také čtenář najde mimo jiné dokončení úvah o výpočtu orientace trojúhelníku a odkaz na jeho implementaci.

Z uvedených náznaků problematiky by mělo být zřejmé, že implementace adaptivní aritmetiky je sice technicky náročná, ale ještě s rozumně velkým úsilím proveditelná.

6 Konzistentní výchylka vstupu

V předcházejícím textu jsme řešili problém, jak vyhodnotit výraz s dostatečnou přesností; v krajním případě nám stačila znalost znaménka výsledku. Pokud jsme problém přesnosti výpočtu vyřešili, stále ještě nemáme bohužel vyhráno.

Uvažujme následující problém a jeho jednoduché řešení. Máme dán obecný mnohoúhelník, jehož hrany se neprotínají, a bod X. Máme rozhodnout, zda bod

X leží uvnitř mnohoúhelníku. Rozhodneme snadno:

1. vytvoř libovolnou polopřímku XY začínající v bodu X,
2. zjistit počet n průsečíků polopřímky XY s hranicí mnohoúhelníku,
3. je-li n liché, je X uvnitř mnohoúhelníku.

I když umíme průsečíky zjišťovat přesně, stále zbývá vyřešit mnoho nepříjemných otázek:

- Jak se má řešit případ, leží-li X na hraně mnohoúhelníku?
- Jak se má řešit případ, leží-li X v některém vrcholu mnohoúhelníku?
- Jak se má řešit případ, leží-li některá hrana mnohoúhelníku částečně či úplně na polopřímce XY?
- Jak se má řešit případ, prochází-li polopřímka XY vrcholem mnohoúhelníku?
- Co dělat v případě, má-li některá hrana mnohoúhelníku nulovou délku a je protnuta polopřímkou XY?
- ...

Jak poslední bod naznačuje, není vůbec zřejmé, zda je výčet otázek kompletní a ani není zřejmé, jak se o tom obecně přesvědčit. Všechny otázky se týkají degenerovaných (singulárních) případů. Degenerovaným případem se myslí situace, kdy se má program o něčem rozhodnout na základě znaménka jistého výrazu, který ovšem nabývá nulové hodnoty. Za regulární případ budeme považovat situaci, kdy je znaménko výrazu (při přesném vyhodnocení) plus nebo minus.

Potíž samozřejmě spočívá v původní formulaci algoritmu, která degenerované případy neuvažuje. A to jde o triviální algoritmus! Co dělat v případě, kdy i základní formulace algoritmu je komplikovaná, například delaunayovská „triangularizace“ v n-dimenzionálním prostoru? Odhaduje se, že řešení degenerovaných případů je věnováno až 90 % kódu; současně je známo, že naprostá většina případů není degenerovaná (viz [11]), čili že kód pro řešení degenerovaných situací se spouští málokdy, pokud vůbec.

Pro podobné situace vznikla metoda konzistentního vychýlení (či symbolické perturbace) vstupu (viz [11]), jejíž myšlenku můžeme popsat následovně:

1. Předpokládej, že vstupní body (či jiné elementy) jsou nepatrně vychýlené ze zadané pozice.
2. Proveď algoritmus, který neuvažuje degenerované geometrické konfigurace. Výchylka zavedená v prvním bodu proto musí být nejen tak velká, aby k degenerovaným případům skutečně nedošlo, ale i tak malá, aby se výstup algoritmu prakticky nelišil od ideálního výstupu.
3. Zkontroluj výstup a koriguj případy, k nimž došlo kvůli umělé výchylce (například: vrať pozice bodů na místa popsaná vstupem, zruš úsečky nulové délky, zruš trojúhelníky nulové plochy apod.).

Při implementaci bychom teoreticky mohli postupovat dvojím způsobem. Za prvé: mohli bychom skutečně vstupní hodnoty nějak vychýlit a pak spustit běžný algoritmus. Samozřejmě bychom potřebovali dopředu vědět, že zvolená výchylka bude „tak akorát“, což zřejmě nebude snadné zařídit. Proto bude prakticky vhodnější zvolit druhý způsob: vstupní údaje nechat v původním stavu a nechat pracovat běžný algoritmus tak dlouho, dokud nenarazí na degenerovaný případ. Jakmile na něj narazí, tj. narazí na výraz V, který nabývá pro jisté parametry nulové hodnoty, musí se algoritmus rozhodnout: má nulu považovat za „kladnou“, nebo „zápornou“?

K rozhodnutí může použít právě techniku vychýlení vstupu. Algoritmus tedy nepatrně pozmění vstupy inkriminovaného výrazu V, tedy de facto vstupní hodnoty algoritmu, a vyhodnotí výraz V znovu. Zde je ale třeba nejvyšší opatrnosti – změna musí být taková, aby jimi nebyla dotčena již provedená rozhodnutí. To znamená: kdyby algoritmus od začátku věděl, že vstup bude jistým způsobem vychýlen (aby výraz V vyšel nenulový), na jeho běhu by se až do vyhodnocení V nic nezměnilo.

Jednou z možností, jak výchylku volit, je technika Simulation of simplicity, viz [20, 11]. Vysvětlíme si ji na jednoduchém příkladu.

Dejme tomu, že algoritmus má na vstupu čísla a_1 až a_n a v průběhu činnosti potřebuje vyhodnocovat znaménka determinantu

$$\begin{vmatrix} a_i & a_j \\ a_k & a_l \end{vmatrix},$$

kde $1 \leq i,\ j,\ k,\ l \leq n$. Pokud místo původního použijeme mírně odlišný vstup, počítáme vlastně

$$\begin{vmatrix} a_i + \hat{e}_i & a_j + \hat{e}_j \\ a_k + \hat{e}_k & a_l + \hat{e}_l \end{vmatrix} = \begin{vmatrix} a_i & a_j \\ a_k & a_l \end{vmatrix} + \hat{e}_i a_l - \hat{e}_j a_k - \hat{e}_k a_j + \hat{e}_l a_i + \begin{vmatrix} \hat{e}_i & \hat{e}_j \\ \hat{e}_k & \hat{e}_l \end{vmatrix}, \quad (16)$$

kde $\hat{e}_i$, $\hat{e}_j$, $\hat{e}_k$, $\hat{e}_l$ jsou malé výchylky vstupních hodnot.

Zvolme $\hat{e}_p = \hat{e}^{2^p}$, kde $1 \leq p \leq n$ a $\hat{e}$ je „dostatečně malé číslo“. Nepotřebujeme znát hodnotu $\hat{e}$ explicitně; musí být jen tak malé, aby jednotlivé sčítance pravé strany (16) měly „důležitost“ závislou na indexu – čím větší index, tím menší důležitost. Pro $p < q$ to znamená

$$\hat{e}_p |a_p| > \hat{e}_q |a_q| \,,$$

tedy

$$\hat{e}^{2^p} |a_p| > \hat{e}^{2^q} |a_q| \,.$$

To je splněno, pokud

$$\hat{e}^{2^p} |a_p| > \hat{e}^{2^p\, 2^1} |a_{p+1}| \,,$$

čili pokud pro všechna p platí

$$|a_p|/|a_{p+1}| > \hat{e}^{2^p} .$$

Je zřejmé, že pro libovolná nenulová $a_1, a_2, \ldots, a_n$ můžeme zvolit tak malé $\hat{e}$, aby byla podmínka splněna. Je-li tedy $\hat{e}$ dostatečně malé a první sčítanec pravé strany (16) (tedy „přesný determinant“) nenulový, nemohou ostatní členy součtu na znaménku nic změnit. Je-li „přesný determinant“ nulový (tj. jde o degenerovaný případ) a $a_{\min(i,\ j,\ k,\ l)} \neq 0$, je znaménko (16) dáno právě znaménkem $a_{\min(i,\ j,\ k,\ l)}$. A tak dále.

Navržený postup tedy rozhodování o znaménku determinantu v regulárních případech nezmění, v degenerovaných případech určí „znaménko nuly“ konzistentně se všemi minulými i budoucími rozhodnutími. Pro jistotu připomeňme, že hodnotu $\hat{e}$ není nutné explicitně určovat; stačí předpokládat, že je tak malá, aby uvedený postup fungoval.

Simulation of simplicity je celkem jednoduchý postup, který odstraní problémy s degenerovaným vstupem. Na druhou stranu ale není bez vad na kráse; za prvé stále vyžaduje přesné vyhodnocování výrazů (v našem příkladu determinantu matice 2×2), za druhé „vyčištění výstupu algoritmu“ nemusí být triviální, za třetí nefunguje se všemi typy algoritmů. Typickou třídou algoritmů, které se simulation of simplicity příliš nefungují, jsou heuristické žravé (greedy) algoritmy, například greedy triangularizace minimalizující součet délek hran triangulace (viz [11]).

7 Závěr

V textu jsme procházeli různé nástrahy, které číhají na programátora (nejen) geometrických algoritmů. Text rozhodně neměl být úplným přehledem používaných a použitelných technik; soustředili jsme se jen na takové, které je možné s relativně rozumným úsilím aplikovat na libovolný typ algoritmu. I tak je ale asi zřejmé, že pro komplikovanější algoritmy je cena za eliminaci závažných chyb dost vysoká, a to jak rychlostí výpočtu, tak složitostí implementace.

Proto je velmi vhodné prozkoumat možnosti knihoven, které usnadňují programování přesných výpočtů; za všechny uveďme knihovny CGAL [22], LEDA [23], GMP [24], MPFR [25] nebo ARPREC [26]. Odkazy na jejich domovské stránky a na další zdroje jsou k nalezení například na

`http://cs.nyu.edu/exact/links/`

Ačkoliv tyto knihovny nabízejí spoustu hotového a prověřeného kódu, neměl by jim jejich uživatel (tj. programátor) slepě důvěřovat. Znalost principů, na kterých fungují, tedy obsah tohoto textu, snad umožní neočekávat od nich nemožné, a vědět, co od nich chtít.

Mnoho zdaru v programování přeje autor.

P.S. Čtenáři se tímto doporučuje podívat se do seznamu použité literatury, kde najde náměty k dalšímu studiu. Oproti běžným seznamům literatury je tento komentovaný a snad poslouží jako startovací bod k opravdovému studiu přesné aritmetiky a přesných geometrických výpočtů.

Reference

[1] IEEE Standard for Floating-Point Arithmetic IEEE Std 754-2008. In *IEEE Std 754-2008* (29 August 2008), pp. 1–58. doi:10.1109/ieeestd.2008.4610935.
Samotná norma pro aritmetiku s plovoucí desetinnou čárkou, nejnovější revize. Jako normativní text není pro studium příliš čitelná ani užitečná; lepší je podívat se do některé z knih či článků, které k normativním tvrzením doplňují vysvětlení, příklady a komentáře. Pro implementátory nízkoúrovňových aplikací je ale samozřejmě znalost textu normy povinná.

[2] GOLDBERG, David. *What every computer scientist should know about floating-point arithmetic.* ACM Computing Surveys, Vol. 23, No. 1., March 1991, pp. 5–48. doi:10.1145/103162.103163
Velmi užitečný úvod do problematiky IEEE 754 od autora nanejvýš povolaného (D. Goldberg se na přípravě normy podílel). Obsahuje mnoho příkladů dokumentujících důvody zavedení různých důležitých detailů normy. Doplněnou verzi textu lze nalézt jako dodatek D manuálu *Numerical Computation Guide*, Sun One Studio 2005, `http://docs.oracle.com/cd/E19957-01/806-3568/ncgTOC.html`, `http://docs.oracle.com/cd/E19422-01/819-3693/819-3693.pdf`

[3] CUYT, Annie A. M.; VERDONK, Brigitte; BECUWE, Stefan, KUTERNA, Peter. *A remarkable example of catastrophic cancellation unraveled.* Computing, Volume 66, Issue 3, May 2001, pp. 309–320. doi:10.1007/s006070170028.
Analýza fatálního selhání pozoruhodně jednoduchého výpočtu. V tomto textu je inkriminovaný výpočet (bez analýzy selhání) uveden na straně 2.

[4] KETTNER, Lutz; MEHLHORN, Kurt; PION, Sylvain; SCHIRRA, Stefan; YAP, Chee. *Classroom examples of robustness problems in geometric computations.* Computational Geometry, Volume 40, Issue 1, May 2008, pp. 61–78. `http://dx.doi.org/10.1016/j.comgeo.2007.06.003`
Velmi přínosný článek s příklady důsledků numerických nepřesností v geometrických výpočtech. Obsahuje i analýzy, proč k chybě v daném příkladu došlo; nevěnuje se ale řešení.

[5] SHEWCHUK, Jonathan R. *Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates.* Discrete & Computational Geometry, Volume 18, 1996, pp. 305–363. doi:10.1007/PL00009321.
Pěkný a podrobný článek, který popisuje práci s čísly vyjádřenými součtem několika čísel s plovoucí desetinnou čárkou. Obsahuje algoritmy pro operace součtu a součinu, které následně aplikuje v adaptivním výpočtu znaménka determinantu, resp. ve výpočtu orientace trojúhelníku (čtyřstěnu) a testu,

zda bod leží uvnitř kružnice (koule). Text je k dispozici na `www.cs.cmu.edu/~quake/robust.html` spolu s implementací v jazyce C.

[6] SHEWCHUK, Jonathan R. *Lecture Notes on Geometric Robustness.* University of California at Berkeley, 2009. Dostupné z `www.cs.berkeley.edu/~jrs/274`
Mnoho užitečných rad, čeho se vyvarovat v geometrických výpočtech. Zejména obsahuje návody, jak dělat některé výpočty lépe. Mnoho z technik je založeno na technikách uvedených v [5].

[7] SCHIRRA, Stefan. Robustness and Precision Issues in Geometric Computation. Kapitola 14 z knihy:
SACK, J. R.; URRUTIA, J. (eds.) *Handbook on Computational Geometry.* Elsevier, 1999. ISBN 9780080529684.
Pěkný přehled problematiky robustních geometrických výpočtů s mnoha odkazy. Je orientován spíše na čtenáře zběhlého v problematice než na úplného začátečníka.

[8] MEHLHORN, Kurt; YAP, Chee. Introduction to Geometric Nonrobustness. Předběžná verze kapitoly 1 knihy *Robust Geometric Computation* (předpokládaný titul). Dostupné z `http://cs.nyu.edu/yap/bks/egc`
Úvod do problematiky robustních geometrických výpočtů s mnoha příklady, co se může pokazit.

[9] MEHLHORN, Kurt; YAP, Chee. Modes of Numerical Computation. Předběžná verze kapitoly 2 knihy *Robust Geometric Computation* (předpokládaný titul). Dostupné z `http://cs.nyu.edu/yap/bks/egc`
Úvod do vyjádření čísel s pohyblivou desetinnou čárkou, čísel s volitelnou přesností a intervalové aritmetiky.

[10] MEHLHORN, Kurt; YAP, Chee. Arithmetic Approaches. Předběžná verze kapitoly 4 knihy *Robust Geometric Computation* (předpokládaný titul). Dostupné z `http://cs.nyu.edu/yap/bks/egc`
Ukázky aritmetického (algebraického) přístupu k implementaci robustních geometrických algoritmů.

[11] MEHLHORN, Kurt; YAP, Chee. Perturbation. Předběžná verze kapitoly 4 knihy *Robust Geometric Computation* (předpokládaný titul). Dostupné z `http://cs.nyu.edu/yap/bks/egc`
Úvod do metody perturbací čili konzistentní výchylky vstupu.

[12] KNUTH, Donald E. *Umění programování: Seminumerické algoritmy.* Vyd. 1. Brno: Computer Press, 2010. ISBN 978-80-251-2898-5.
Klasická kniha klasického autora o základech programování. Kapitola o počítačové aritmetice s plovoucí desetinnou čárkou obsahuje jak

axiomatický výklad jejích vlastností, tak důkladnou analýzu základních aritmetických operací.

[13] MULLER, Jean-Michel; et. al. *Handbook of Floating-Point Arithmetic.* Springer, 2009. ISBN 9780817647056.
Velmi pěkně napsaná kniha shrnující prakticky vše, co programátor potřebuje vědět o číslech s plovoucí desetinnou čárkou. Její hlavní výhodou je rok vydání – obsahuje i podrobnosti z poslední revize IEEE 754.

[14] OVERTON, Michael L. *Numerical Computing with IEEE Floating Point Arithmetic.* SIAM, 2001. ISBN 9780898718072.
Velmi jednoduchá úvodní kniha (104 stran) do problematiky čísel s plovoucí desetinnou čárkou. Je orientovaná na nematematicky orientované programátory.

[15] MULLER, Jean-Michel. *Elementary Functions: Algorithms and Implementation.* 2nd ed. Springer, 2006. ISBN 9780817643720.
Kniha o implementaci základních aritmetických operací a elementárních funkcí s čísly s plovoucí desetinnou čárkou.

[16] CANNY, John; DONALD, Bruce; RESSLER, Eugene K. *A rational rotation method for robust geometric algorithms.* SCG 1992 Proceedings of the eighth annual symposium on Computational geometry, pp. 251–260. ACM New York, NY, USA. ISBN 0-89791-517-8.
doi:10.1145/142675.142726
Pěkný článek analyzující možnosti přibližně vyjádřit úhel nepřímo pomocí jeho sinu a kosinu z oboru racionálních čísel.

[17] JAHNEL, Jörg. *When is the (co)sine of a rational angle equal to a rational number?* Dostupné z `http://arxiv.org/abs/1006.2938`.
Jednoduchá analýza problému jasně popsaného názvem článku.

[18] MÖRIG, Marc; SCHIRRA, Stefan. *Engineering an Exact Sign of Sum Algorithm.* Technical report nr. FIN-002-2010. Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg, 2010. ISSN 1869-5078.
Analýza různých způsobů rozhodnutí znaménka sumy čísel s plovoucí desetinnou čárkou.

[19] DEKKER, Theodorus J. *A floating-point technique for extending the available precision.* Numerische Mathematik, 1971/72, Volume 18, Issue 3, pp. 224–242. doi:10.1007/BF01397083.
Původní článek s návrhem algoritmu součtu a součinu dvou čísel s plovoucí desetinnou čárkou. Je psán velmi obecně, norma IEEE 754 vznikla o 14 let později.

[20] EDELSBRUNNER, Herbert; MÜCKE, Ernst P. *Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms.* ACM Transactions on Graphics, Volume 9 Issue 1, Jan. 1990, pp. 66–104. doi:10.1145/77635.77639.
Původní článek navrhující metodu simulation of simplicity. Obsahuje mnohem více podrobností než [11].

[21] FIGUEIREDO, Luiz H. de; STOLFI, Jorge. *Self-Validated Numerical Methods and Applications.* Brazilian Mathematics Colloquium monograph, IMPA, Rio de Janeiro, Brazil, July 1997. Dostupné z `www.ic.unicamp.br/~stolfi/EXPORT/projects/affine-arith`
Původní popis techniky afinní aritmetiky, obsahuje i úvod do intervalové aritmetiky.

[22] *CGAL: Computational Geometry Algorithms Library.* `www.cgal.org`
Knihovna orientovaná na C++ pro efektivní implementaci spolehlivých geometrických algoritmů. Obsahuje jak funkcionalitu pro obecnou algebru, tak specializované části zaměřené na geometrické výpočty. Poskytuje rovněž mnoho geometrických algoritmů, například triangularizace, konstrukce konvexních obálek apod.

[23] *LEDA.* `www.algorithmic-solutions.com/leda`
Knihovna orientovaná na C++ zejména pro geometrické algoritmy, grafové algoritmy a kombinatorickou optimalizaci.

[24] *GMP: The GNU Multiple Precision Arithmetic Library.* `https://gmplib.org`
Knihovna orientovaná na C/C++ implementující aritmetiku s volitelnou přesností. Podporuje celá čísla, racionální čísla a čísla s plovoucí desetinnou čárkou. Knihovna je orientovaná zejména na kryptografii, počítačovou algebru apod., ale užití najde i v geometrických výpočtech.

[25] *The GNU MPFR Library.* `www.mpfr.org`
Knihovna orientovaná na ANSI C implementující práci s čísly s plovoucí desetinnou čárkou s volitelnou přesností. Zaměřuje se na korektně zaokrouhlené výpočty.

[26] *ARPREC.* `http://crd-legacy.lbl.gov/~dhbailey/mpdist` Knihovna orientovaná na Fortran-90 a C++ implementující aritmetiku s volitelnou přesností. Podporuje čísla celá, reálná a komplexní (ta jsou implementovaná polem čísel s plovoucí desetinnou čárkou).

[27] SKALA, Václav. *Algoritmy počítačové grafiky 2.* Skripta. Západočeská univerzita v Plzni, 1992. ISBN 80-7082-059-4.

Stále užitečná skripta s popisem mnoha základních algoritmů počítačové grafiky. Online k dispozici na `www.vaclavskala.eu`